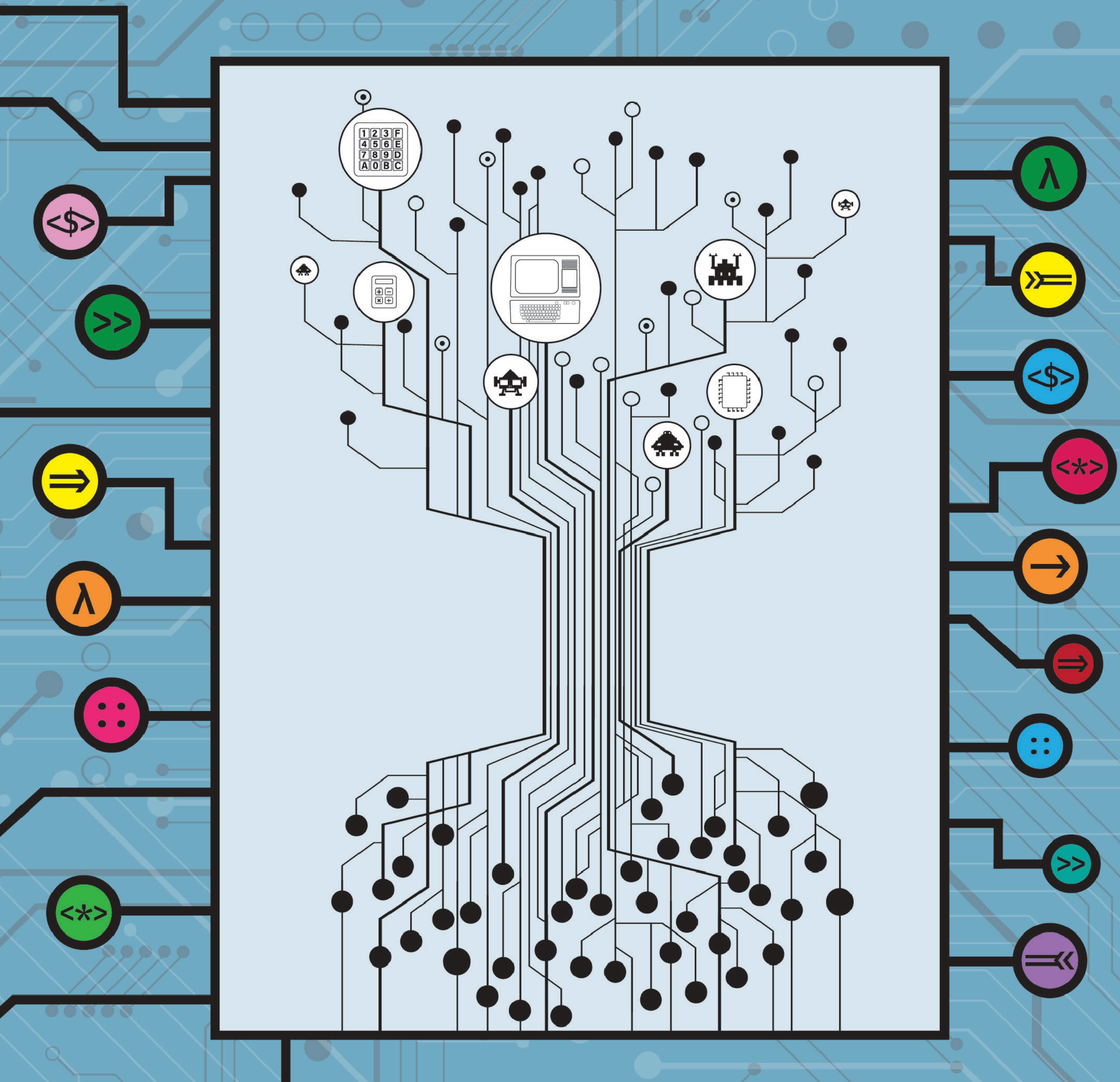
Retrocomputing with Clash

Haskell for FPGA Hardware Design



Gergő Érdi

Retrocomputing with Clash

Haskell for FPGA Hardware Design

GERGŐ ÉRDI

https://unsafePerform.IO/retroclash/

Retrocomputing with Clash: Haskell for FPGA Hardware Design

First edition, 2021.

Cover by Jutka Sallai

Bouncy pushbutton scope screenshot from https://commons.wikimedia.org/wiki/File:Bouncy_Switch.png licensed under the terms of the Creative Commons CC0 1.0 Universal Public Domain Dedication 🅭🄍

https://unsafePerform.IO/retroclash/

Contents

Introduction

Preliminaries

In this book, we are going to assume at least intermediate level familiarity with Haskell. We will feel free to use many of GHC's general-purpose language extensions like `DataKinds` or `TypeApplications`, without introducing them. In fact, Clash, the main tool we will use throughout this book, turns on these (and many more) extensions by default. On the other hand, Template Haskell, being a more specialized tool that is not necessarily used by, the working Haskell programmer, will get its own introduction before using it in anger in chapter 14.

We take a similar approach to libraries: we will use *Containers*, the *Monad Transformer Library*, and small parts of *Lens*, assuming readers are either already familiar with them, or can look up the documentation as needed. Libraries for more niche domains, such as *Barbies* for easier handling of higher-kinded data types, *Terminal* for text IO, and *SDL2* for interactive, graphical applications, are described in detail alongside the parts of the book that use them.

Notation

In text, `monospaced` typeface denotes code fragments; the most common case is referring to Haskell types and functions by name, such as `Bool` or `fmap`. For the actual definitions that make up our programs, we will use code blocks like this:

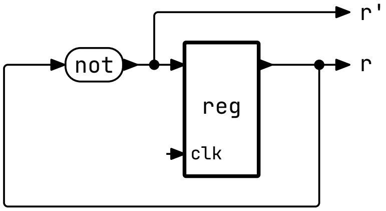
```
succIdx :: (Eq a, Enum a, Bounded a) => a -> Maybe a
succIdx x | x == maxBound = Nothing
          | otherwise = Just $ succ x
```

Due to the limitations of paginated media, code blocks sometimes necessarily spread across page boundaries. Good code should tell a story, so in these situations, the cliffhangers get their own "to be continued" title card by drawing the code block's frame dashed, as in the following example:

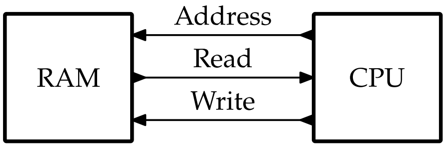
```
predIdx :: (Eq a, Enum a, Bounded a) => a -> Maybe a
predIdx x | x == minBound = Nothing
          | otherwise = Just $ pred x
```

Sometimes, we will build function or data type definitions piecewise over several blocks. In these cases, we will indicate where later parts are to be inserted, by including some of the previous code. Readers are very much encouraged to follow along by assembling the program from the parts and experimenting with the incomplete intermediate steps, not just looking at the final version.

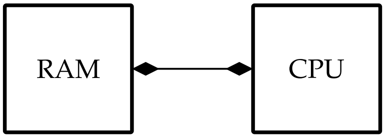
In illustrations, since we are creating hardware by writing code, the role of the schematic diagrams is just to provide overview. We will include more or less detail in these diagrams in different contexts. When we want to highlight the distinction between purely combinational circuits vs. stateful ones, we will enclose the pure parts in rounded shapes, like the `not` circuit in the following example:



When the details of memory access are the focus of discussion, we will draw separate connections for address, read, and write lines. In other contexts, we will keep our diagrams easier to read by using a single connection, denoting the read-only, write-only, or read/write access by the direction of the connection:



(1) Detailed memory connections



(2) Simplified memory connection

Resources

The website of this book is at `https://unsafePerform.IO/retroclash/` with links to repositories of the full source code of all programs appearing in the book.

Acknowledgements

I don't know if it takes a village to write a book, but a village certainly came together for this one.

Even before the idea of writing a book, I've been regularly bouncing my increasingly hare-brained schemes of describing hardware with all the abstractions afforded by Haskell with Marten Ågren, Nguyễn Hữu Hải, and Raphael Montelatici. Hải originally put the bug in my ear that I should tell not just them, but the whole world about these ideas.

Once I got to writing, Christiaan Baaij supported this endeavor from the get-go. The whole Clash development team and QBayLogic got busy to fix the various bugs I've uncovered on the way with very quick turnaround. Christiaan, Cosette Reczek, Andor Pénzes, Ellie Hermaszewska, Viktória Kiss, and Miëtek Bak provided invaluable feedback on the chapters as I wrote them. I have only myself to blame for any errors that might still have slipped through.

Károly Lőrentey helped me enter the world of typography beyond "whatever is default in LaTeX", and Viktória had great ideas for how to turn some of the concepts in my head into illustrations. I am also indebted to TEX.stackexchange.com in their assistance in turning these ideas into reality.

. . .

But before any of that, there was my father, bringing first an Atari 2600 home, then a short while later, a Commodore 64. Followed by countless nights figuring out *Impossible Mission 2*, and countless days designing BASIC programs and drawing sprites of red balloons on graphing paper. He would have enjoyed — not the technical details — but certainly the *idea* of this book.

1 Into the world of FPGAs

This book is about the intersection of FPGAs, retrocomputing, and Haskell; approaching the topic from the direction of Haskell. In other words, we expect readers to be familiar with traditional (software) programming in Haskell, and will concentrate on the hardware aspects. This chapter's goal is to introduce FPGAs, and to make our argument for picking retrocomputing as our topic.

1.1 Computers everywhere

We are surrounded by computers. Smartphones are computers. Gaming consoles are computers. Smart TVs are computers, but so are "dumb" ones as well. Washing machines are computers. Bluetooth headphones are computers, as are non-Bluetooth noise canceling ones. *Computers are computers*. In fact, computers are surrounded by smaller computers in their peripherals: there's a computer in the keyboard, there's one in the mouse, and there's one in the network adapter.

If everything is a computer, does that make the term meaningless? No. What we mean by the word "computer" here is an artificial physical device designed to implement a finite approximation of a Turing machine. There are many proposed ways of building devices that perform computation: we can build a computer from water running through an elaborate network of pipes; we can synthesize organic molecules whose interaction corresponds to computation; there are even children's toys in which a ball falling through a lattice of pins and spindles can perform universal computation achieving Turing-completeness.

However, there is only one substrate of computation that sees real-world use: digital electronics. If we look at all these computers around us as physical artifacts, we see that they contain a bunch of functionality-specific parts (the drum of the washing machine, the speaker of the headphone, or the LCD of the TV) and one or more fiberglass epoxy boards (the *printed circuit boards* or *PCBs*) containing small plastic boxes and electric components (mostly resistors and capacitors), with copper traces running between the parts. To the naked eye, the epoxy boards look very similar across devices, even though the washing machine is a lot less capable a computer than a smartphone.

Amazingly, this similarity runs much deeper. Resistors are resistors, capacitors are capacitors, copper is copper, so the difference must be inside those small plastic boxes, right? If we zoom in on the inside of one of those boxes, we find a tiny silicon plane called an *integrated circuit* or *IC* for short. And if we zoom in much, much more, we will see that on these planes, all ICs are made up of a large number of components: a small portion is tiny versions of the resistors, capacitors, and other electric components we've seen earlier, and the rest are transistors.

Different chips have different sized transistors; the rapid advancement in computing performance since the first computers is in large part because of improvements in integrated circuit manufacturing. Newer transistors are smaller, use less energy, and so more of them can be packed into the same area. But functionally, all these transistors, large and small, are exactly the same: a tiny version of an electric switch where being short or open is itself also controlled electronically.

What this means is that the difference between the microcontroller in a toaster and the processor of a top-of-the-line supercomputer is only in how many of these transistors are connected to each other, and what the connections are. The key corollary is that we can build our own computer just by deciding on the connections.

1.1.1 Digital electronics

As the saying goes, in theory, theory and practice are the same, but in practice, they are not. The above description of the insides of a computer is a gross simplification, glossing over a lot of the electrical components. However, we are in good company by making these simplifications: when we say that the computer is made of a network of transistors and nothing more, we are using the *digital abstraction*.

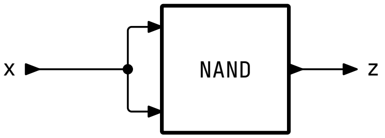
The idea of digital electronics is to pretend that every transistor lead and every pin visible to the outside world is always at one of only two voltage levels, referred to as "low" and "high" voltage. Of course, nature is messy, and it takes considerable electrical engineering effort to maintain this abstraction. In fact, for purely digital circuits, the role of everything else on the circuit board (resistors, capacitors, etc.) is to ensure that the digital abstraction isn't leaky.[1]

In this book, we will stay firmly inside the confines of the digital abstraction. When interfacing with the rest of the world, we will assume that the peripheral hardware contains sufficient circuitry to solve impedance mismatches, translate between voltage levels, and so on.

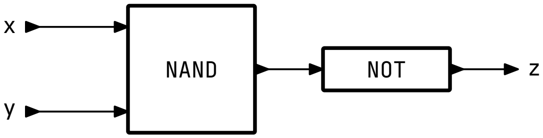
[1] There are also circuits where the functionality is intrinsically tied to non-digital electronics, e.g. a radio receiver. Here, we regard those parts in the same way as we regard the heating coil of a toaster: a peripheral that interfaces with the digital parts, but isn't part of the "computer" proper.

1.1.2 Universality

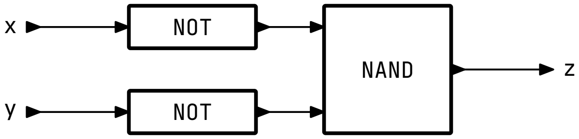
From two transistors, we can build a `NAND` gate: a two-input, one-output circuit where the output is the complement of the two input's conjunction (the name "`NAND`" stands for "`NOT-AND`"); in Haskell notation, we could write it as `\x y -> not (x && y)`. By using enough `NAND` gates, we can implement any function mapping n Boolean inputs to m outputs. This property of the `NAND` gate is called *universality*.[2] By way of example, the following circuit diagram implements a `NOT` gate from a single `NAND` gate; armed with `NOT` and `NAND`, we can directly implement all the usual binary Boolean functions, since e.g. `x AND y = NOT (x NAND y)` and `x OR y = NAND (NOT x) (NOT y)`.



(1) `z = NOT x`



(2) `z = x AND y`

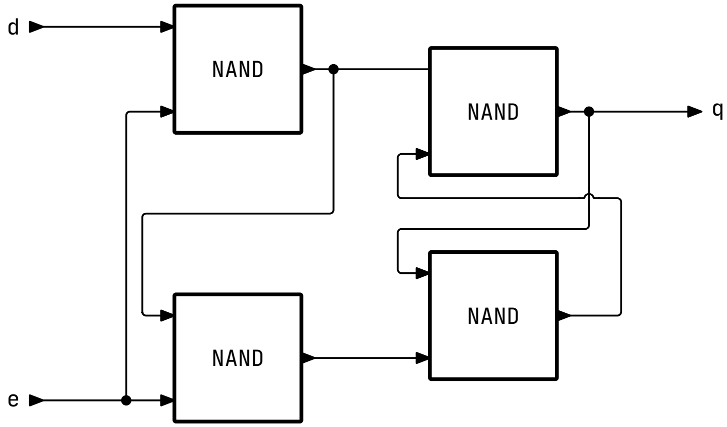


(3) `z = x OR y`

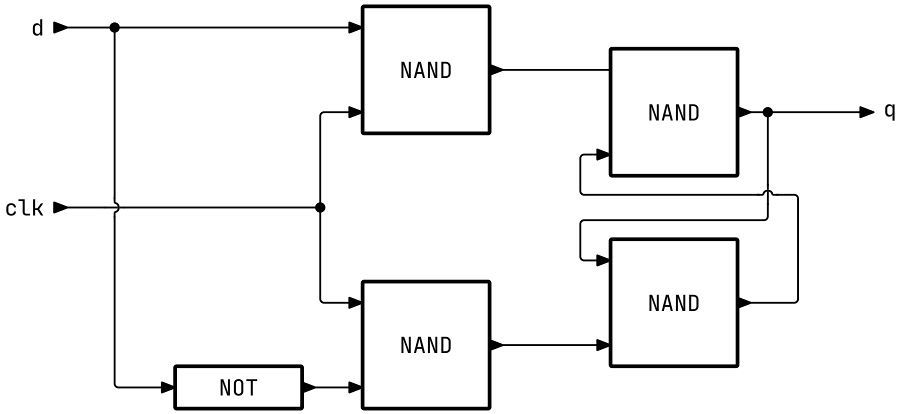
In most devices, however, the outputs are not purely a function of the inputs: there is also some *internal state*. By wiring together four `NAND` gates in a crossover configuration, we can create a so-called *latch*: a circuit with two inputs (*data* and *enable*) and one output where the output "remembers" what its data input was the last time the enable input was high:[3]

[2] For any given concrete function, it is entirely possible that we can design a circuit that implements that function using fewer transistors than the one built from `NAND` gates. The point here is that via the `NAND` gate's universality, we can show that *any* function can be built just by wiring together transistors, without needing any other components.

[3] More specifically, this configuration is called a *D latch*. Other latch types with slightly different behavior are also used in circuit design; for example, an *SR latch* also has two inputs, but the output "remembers" which of the two inputs was last seen high.



Adding one more `NAND` gate, we can build a *flip-flop*: a latch that is triggered not by a high enable input, but by the enable input (called the *clock* in this case) *changing* from low to high. As we will see in chapter 4, clocks are essential to compositional circuit design.



To summarize, it is possible to build this tower of abstractions where the higher levels correspond to the purely functional and the stateful parts of a given device's intended behavior, but at its foundation, everything is built up of the same basic element of the `NAND` gate.

1.2 Field-programmable Gate Arrays

The rest of this book is about building even higher abstractions to make it manageable to design large circuits that implement complicated behavior. But once we have such a design, how are we going to turn that into physical hardware?

As we have seen, one solution would be to climb all the way down into the world of transistors, and wire them together. This is exactly what happens in traditional chip design: the output of the development work is the placement and connection of every single transistor, in a blueprint that is then shrunk down onto silicon wafers. The high-level design happens within the confines of the digital abstraction, but since the end result is a physical artifact, it will be subject to all laws of physics, not just the ones we'd like to apply: part of the design work is to address the electrical requirements of *maintaining* the digital abstraction. The cost of a full roundtrip from design to physical chip is enormous: depending on the details of the manufacturing process, it can cost millions of dollars. This is very different from the world of software, where a full from-scratch compilation of a program is essentially free.

The alternative is to develop using a so-called *field-programmable gate array*, or *FPGA* for short. As the name implies, an FPGA is a large number of logic gates where the connections between the gates can be set "in the field", i.e. after the manufacturing process. From the outside, an FPGA looks like any other fixed-layout chip, but internally, the connections between the gates are configured electronically from an uploaded schematic. Trying out a new design is as simple as recompiling the design into its low-level gate-by-gate representation, running a tool that creates a circuit layout confirming to the intended connections, and uploading this new layout. Functionally, it's like having our own chip factory!

The compromise in FPGAs is that the programmable interconnects are much more complicated than fixed traces: they use up some of our transistor budget. The consequence is that an FPGA is more expensive (per unit) and uses more power than a purpose-built chip. It is also not quite a standalone component: since the whole point is to keep the configuration dynamic, external support elements are needed to provide the programming at power-up.

These drawbacks are less relevant to us in this book:

- Unit cost: we are not interested in designing for manufacturing, but in creating one-off circuits. Furthermore, our designs are simple and small enough that they use a small number of components, fitting onto low-tier hobbyist FPGAs instead of the expensive, top-of-the-line, very fast, very large offerings.

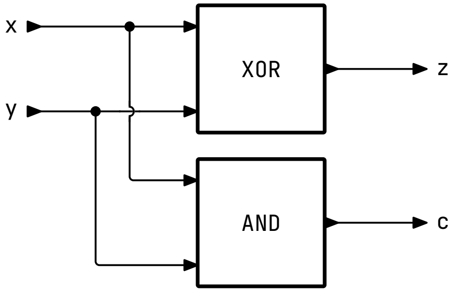
- Power consumption: we will not design for embedded applications, and because we will be implementing old computers that were very slow compared to today, our designs will not need to run anywhere near the power/heat limits of today's FPGAs.

- Supporting circuitry: tying back to building one-offs, we will use pre-made development boards instead of designing our own purpose-built PCBs around bare FPGA chips. These off-the-shelf development boards contain everything needed to make the FPGA work, freeing us up to care only about the circuit design inside the FPGA.

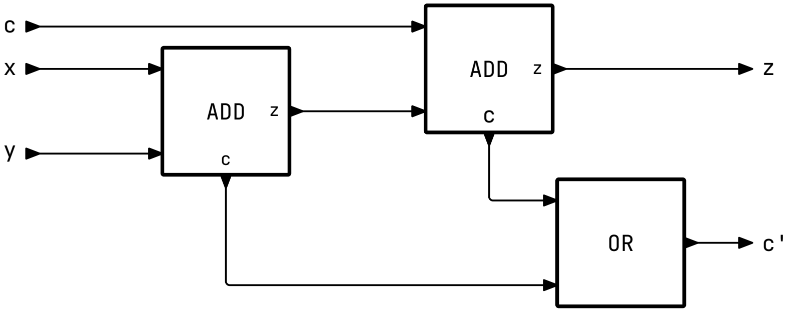
1.2.1 From design to a network of NANDs

Suppose we design a circuit that has 16 inputs and 25 outputs: the inputs are grouped into two 8-bit numbers `x` and `y`, and the output consists of the bits of `(complement x, complement y, x + y)`. The third component of the output, `x + y`, is 9 bits long, to account for the possibility of carry.

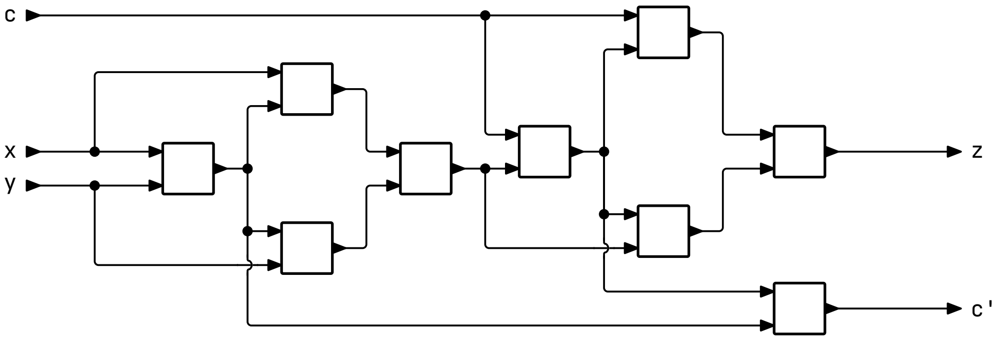
We have already seen how to use one `NAND` gate per bit for the complement. For the addition, we start with adding just two one-bit values: the so-called *half adder*:



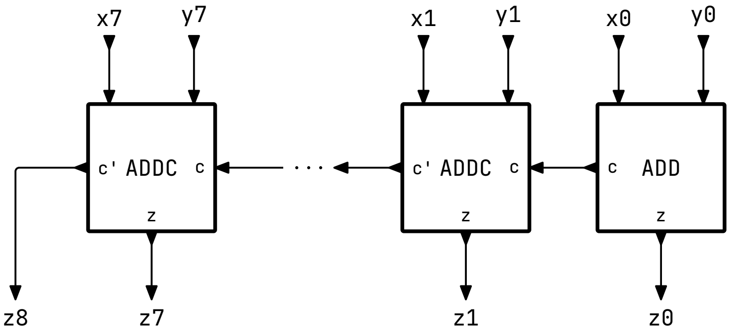
To extend this to multiple bits, we need a way to propagate the carry outputs from lower bits to higher ones. A *full adder* has an additional third input for the carry-in, i.e. the carry-out from the lower bit-pair:



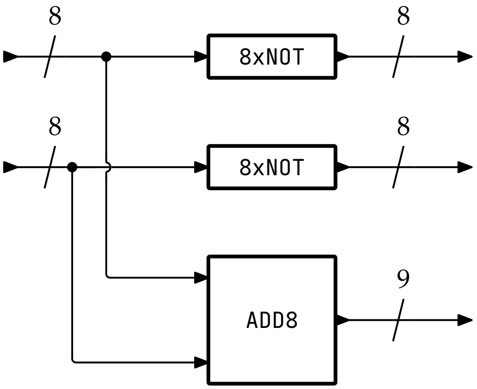
We can build circuits for `AND`, `OR`, and `XOR` using 2, 3 and 4 `NAND`s respectively. This gives us a total `NAND` count of 6 for the half-adder, and so 15 for the full adder. Alternatively, by breaking the abstractions, we can use just 9 `NAND` gates to implement the same functionality (in this diagram, every gate is a `NAND` gate, so we omit labels to save on space):



Regardless of which implementation we use for our full adder, we can then build the 8-bit adder by connecting 8 of our full adders in a so-called *ripple-carry configuration*:



We are now ready to assemble our complete circuit, using the components defined above:



The slashes on the lines here emphasize the fact that each line actually corresponds to multiple (8 or 9) wires. Pretending that we have "wide" (multi-bit) lines (called *bus*es) is a very useful abstraction in avoiding clutter. In fact, it is so useful, that in the rest of this book we will use it all the time without using these bus marker slashes to distinguish them; one-bit lines will be thought of as degenerate buses that just happen to be one-bit wide.

What this final image tells us is that we can take $8 \cdot 1 + 8 \cdot 1 + 8 \cdot 9 = 88$ `NAND` gates, connect them in the configuration that we see when all the abstract boxes are inlined, and the resulting 16-input 25-output circuit will implement our desired behavior.

1.2.2 Lookup tables as basic elements

Since the complicated part of an FPGA is the electronically configurable routing between elements, and of course the more elements we have, the more routing is needed between them, it makes sense to reduce that complexity by using fewer, larger elements. This is why FPGAs are not called field-programmable *transistor* arrays: although everything is built from transistors, the basic unit of addressing is not a single transistor, but larger elements built from it. The idea is that for any circuit we would actually want to design, it will have a lot more structure than just a network of transistors.

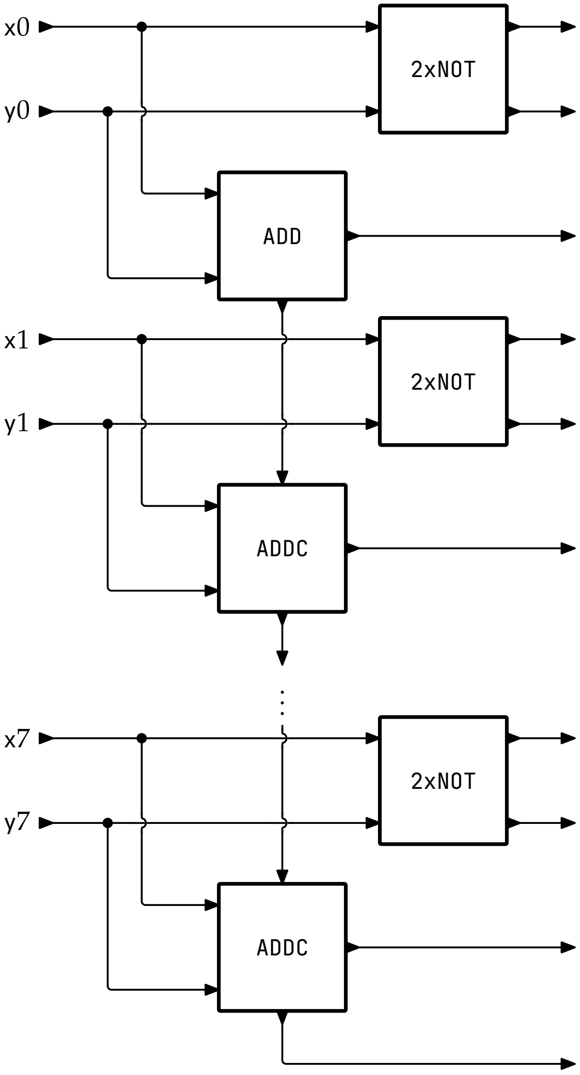
Looking at our previous example, suppose we also have `NOT` gates as basic elements, still implemented with a single `NAND` gate internally. While this doesn't save on the total number of `NAND` gates, it does cut down on the number of junctions: in contrast to a `NAND` element that has three leads connected in a certain way to implement a `NOT` gate, a dedicated `NOT` element only has two leads. Thus, for the 16 `NOT` gates, the number of junctions goes down from $16 \cdot 3 = 48$ to $16 \cdot 2 = 32$.

We can save a lot more junctions if we also have full adders as basic elements, since now we only need routing between $16 + 8 = 24$ elements instead of the original 88.

In this hypothetical example, the reduction in routing was achieved by the design of the circuit matching the underlying FPGA's structure perfectly: the FPGA had `NOT` gates and full adders as basic elements, and our circuit trivially decomposed into complements and addition. But the whole point of an FPGA is to be versatile.

The answer to this seeming contradiction is to make the basic elements themselves configurable. On an FPGA, instead of using fixed-function gates like `NOT` or `ADD`, each basic element is a tiny piece of memory called a *lookup table* (*LUT* for short) that is initialized together with the inter-element connections. For example, if each element is a memory with 3 address bits and 2 data bits, we can put a 3×2 lookup table into it. That is good for two `NOT` gates, or one full adder.

In the following diagram, the `2xNOT`, `ADD`, and `ADDC` blocks all refer to 3×2 LUTs loaded with the truth table of two parallel `NOT` gates, a half-adder, and a full adder, respectively. Unused inputs and outputs (such as the third input of a `2xNOT` element) are omitted.



So now we can take just 8 + 8 = 16 elements to implement our circuit, without requiring the FPGA to be designed with prescience to our intended use case. On the other hand, we have now introduced additional complexity to the design process itself. Although our problem statement was “the bits are complemented”, we now need to group that into two-bit units just to get better resource utilization. Then a new FPGA model comes out with 3-output basic elements, and now we need to rework our design to take advantage of being able to put three `NOT` gates on one lookup table.

What happens in practice is that the details of lookup table dimensions and routing distances are hidden from the developer, and handled by vendor- and model-specific toolchains. We write our circuit in terms of one-bit `NOT` gates, and let the compiler (commonly called *synthesizer* in the FPGA context) figure out the nicest way to map that onto the elements available on our target FPGA.[4]

1.2.3 FPGAs as hardware gadgets

Now that we have an idea of what goes on *inside* an FPGA, what is it actually, as a physical device? Although we include many simulators in this book, ultimately, the payoff is in synthesizing and uploading the designs into FPGAs. So what does that look like?

For development purposes, our best bet is to get an FPGA *development board* aimed at educational uses: a standalone hardware gadget that contains the FPGA; its support components such as flash ROM that will store its configuration between power cycles; some hardware peripherals like pushbuttons, toggle switches, LEDs, or seven-segment displays; connectors in standard format for further peripherals like USB for input devices or VGA for video output; a USB connector to upload the synthesis result from a computer; and some way of powering the board when not connected to a computer, usually in the form of a micro-USB socket or simple DC input.

Development boards are useful for us because they take care of all the electronic concerns that are outside the scope of this book. For example, lighting an LED on the board is as simple as setting the appropriate output pin's value to high, without having to worry about voltage levels and making sure the current won't be too large to fry the LED. They are also satisfyingly tactile: unlike a PC expansion card, we can hold the whole thing in our hands, disconnect it from the programming computer, and enjoy that it really is a standalone circuit that does everything by its own power.

1.2.4 Toolchains

In the software development world, we are used to open source, standardized compiler toolchains. Unfortunately, this is not at all the case for FPGAs. Each FPGA

[4] Just like in software, squeezing out the maximum performance sometimes requires reaching through abstractions and taking into account the specifics of the target device. One advantage of reproducing old computers is that in this book, we will not get even close to needing to do that.

vendor provides its own synthesis toolchain, containing a ton of model-specific knowledge.

The normal way of interacting with the toolchain is by writing the circuit description in a *hardware description language* (commonly abbreviated as *HDL*). The two big players in this space are VHDL and Verilog: both have IEEE standards describing them. The VHDL or Verilog input is then processed by vendor-specific tools, mapping parts of the circuit to various elements of the given FPGA model, and then producing an opaque binary blob called a *bitfile*, which can be uploaded with an (again, vendor-specific) tool to the FPGA.

Since all intermediate representations are non-standard and proprietary, anything we want to do will have to go via HDL. We can think of the situation similar to how we can use any programming language to run in the browser, as long as it is JavaScript. If we want to use some other language, our program has to be translated to JavaScript. This translation can be simple and direct, with the output being in direct correspondence with the input (e.g. TypeScript); or it can be very involved, as in the case of GHCJS that compiles Haskell into JavaScript.

Due to this non-standard, highly vendor-specific nature of the development environments, in this book, we will not look beyond creating the HDL source (and we will not even write any HDL by hand), and will assume the reader has read up on the specifics in the documentation provided with their FPGA development boards.

1.3 Retrocomputing

The motivating examples in this book all fall into one of two categories: simple circuits that demonstrate a single functionality, and complex, fully-worked-out systems based on old computers. "Old" can mean different things to different people; for us, it means the time period when computers just started entering the lives of everyday people: the late seventies arcade machines and home computers.

It would be a lie to say that this choice has nothing to do with the author's age and childhood. However, there are two objective benefits to studying old computers:

- CPUs and computers that were considered simple and cheap even in their own time were designed by very small teams, sometimes individuals. This makes it possible to understand them holistically.

- The performance gap between then and now is so large, that we can get something working without having to worry about doing everything the most efficient way. The goal of this book is not to train professional hardware designers, but to learn about quaint old computers and to write down what we learn in code that is not only functionally correct, but also nicely readable and concise for humans.

None of the techniques explored in this book are specific to retrocomputing; however, there are certainly some modern areas we don't venture into. As an example, we never even consider pipelined CPU designs.

1.4 Haskell meets Hardware

Since Verilog and VHDL, the two standardized hardware description languages, are the lingua franca of FPGA toolchains, if we want to use Haskell as a "better HDL", we need a way to meet the toolchains on their own turf. The two approaches to this are as an embedded DSL (e.g. Lava), or as a compilation target (e.g. Clash).

1.4.1 Lava

In the Lava approach[5], the role of Haskell is that of a macro language: the library provides types corresponding to circuit fragments, and combinators over these types, and the ambient Haskell language can be used in building abstractions for how these fragments are put together. Internally, signals are represented as data that describes their definitions; to create the HDL output, we write a Haskell program that uses the Lava entry point that traverses that description for the value representing our complete circuit design, and then when we *run* that program, it writes out a file containing Verilog or VHDL source.

Because the primitives exposed by Lava correspond directly to the primitives of the target HDL, it is easy to track provenance of the output. Error messages, or results from downstream testing tools, are straightforward to interpret in the context of the source code. Of course, this correspondence becomes less straightforward as the Haskell code that generates that final circuit value becomes more involved; but even then, since the HDL output is created by running our program, we can use Haskell-runtime debugging tools to do further tracking.

Pushing further, since we have a Turing-complete macro language with rich types in the form of Haskell, we can fully automate source code creation, for example by writing an interpreter from our own high-level description into Lava code. Since the interpreter runs before the output HDL is consumed, there is no limitation in the structures that can be generated, and no interpretation overhead in the resulting HDL.

On the flip side, Lava is a first-order language over primitive HDL types only. We can use arbitrary Haskell types to drive our macro code, but the result is com-

[5] There has been several dialects of Lava since its original 1998 paper, resulting in a whole family of implementations. At the level of granularity in this chapter, they can all be regarded as the same.

pletely disconnected from these types. For example, there is no concept of defining algebraic datatypes, or pattern matching on them, in Lava. There is no way to lift functions on pure values to functions on signals, short of generating a small ROM containing the complete graph of the function as a lookup table.

1.4.2 Clash

The Clash approach uses GHC as a compiler frontend, consuming its Core intermediate language, inlining definitions to dissolve recursion, and mapping the result to HDL constructs. In this model, we write straight Haskell code, and then *compile* it with Clash instead of GHC.

This results in the ability to use all features of Haskell in our circuit descriptions. Our circuits can be pure functions, or, if state is needed, we use the built-in `Signal` type constructor, which is an applicative functor, allowing pure Haskell function definitions to be lifted to the world of signals. Signals can contain arbitrary Haskell types: the Clash compiler applies supercompilation techniques to get rid of intermediate signals of non-representable contents. For example, in something this simple:

```
(&&) <$> xs <*> ys
```

the result of `(&&) <$> xs` is of type `Signal dom (Bool -> Bool)`, but we don't need to worry about time-varying functions, since the full term is a `Signal dom Bool` where the function is statically known to be the logical-AND function.

One drawback of this methodology is that some generalizations require quite involved type-level programming, compared to the Lava/DSL approach where ultimately, we're stitching together term-level HDL fragments. We will see quite early in this book that for most uses, lists have to be replaced by length-indexed vectors, to ensure they can be manipulated with circuits of statically known size. In chapter 14, we will also encounter a situation where Clash's supercompilation technique is simply not sophisticated enough to deal with certain recursively defined, but finite, circuits.

Nevertheless, this book puts its vote down firmly for Clash. One reason for this is that the intended audience is already familiar with Haskell and regards the type-level programming bits as fun and not scary at all. The second reason is that algebraic datatypes and pattern matching are simply too painful to live without. Starting at chapter 6, most of our circuit design will be based on translating input signals into a circuit-specific datatype, and consuming values of that datatype in the main state transition function.

2 Hello, Clash!

In this chapter, we will write our first Clash program to connect some simple inputs to simple outputs. "Simple" here means that the digital signal connected to the FPGA's pins correspond directly to the peripheral's state. For example, a two-state switch connected to an I/O pin will pull that pin to either `low` or `high` depending on the position of the switch. Similarly, an LED is either on (lit up) or off (dark). Later, we will interface with more complicated peripherals that require coordination between multiple pins and/or through time.

Most FPGA development boards have some pushbuttons, switches and LEDs directly accessible from the FPGA; for those that don't, we can use a breadboard to connect these components as standalone parts to the so-called *general purpose I/O* (*GPIO*) pins of our board.

2.1 Bit

Digital signal levels are represented by the `Bit` type in Clash. It has two values `low` and `high`, conversion functions `boolToBit` and `bitToBool`, and is an instance of standard Haskell typeclasses like `Eq`, `Ord`, `Show`, `Bounded`, and `Enum`. It is also an instance of `Num` as modulo-2 arithmetic, and `Bits` / `FiniteBits` in the straightforward way.

In this book, we will make an effort to keep `Bool` and `Bit` separate. For example, the signal level corresponding to a pushbutton connected to an I/O pin is a `Bit`, but whether the button is pushed or not is a `Bool`. Depending on how the pushbutton is wired in a given circuit, either `high` or `low` could be mapped to `True`. This can get tricky to track properly, so we will also define a newtype wrapper around `Bit` that tracks its polarity:

```
data Polarity = High | Low

newtype Active (p :: Polarity) = MkActive{ activeLevel :: Bit }
    deriving (Show, Eq, Ord, Generic, NFDataX, BitPack)
```

```
active :: Bit -> Active p
active = MkActive
```

This allows conversion of signal levels to logical `Bool` values, either by direct conversion (for `Active High`) or by taking the complement (for `Active Low`):

```
class IsActive p where
    fromActive :: Active p -> Bool
    toActive :: Bool -> Active p

instance IsActive High where
    fromActive = bitToBool . activeLevel
    toActive = MkActive . boolToBit

instance IsActive Low where
    fromActive = bitToBool . complement . activeLevel
    toActive = MkActive . complement . boolToBit
```

2.2 `Signal`

The `Signal` type represents time-varying values. Clash's model of time is *discrete* and *synchronous*: at every time step, all signals have some value and then in the next time step, all signals are updated. In Clash's simulator, signals are turned into streams (represented as infinite lists), with every element corresponding the the signal's value at a given timestep.

`Signal` is parameterized by the *clock domain* that a given signal belongs to. Clock domains are important because crossing over a signal from one domain to another requires special care. For example, imagine one domain operating at 100 MHz and another one at 1 MHz. If a signal in the faster domain changes value for a single (10 ns) cycle, and something is derived from it in the slower domain, what should happen during the whole 1 μs cycle of that domain? There is no single, general, good answer to this. By tagging signals with their clock domains, Clash enforces that only signals in the same domain can be connected without explicitly taking care of these issues.

2.3 Our first circuit

Armed with just `Bit` and `Signal`, we can now write our very first Clash program in just three lines:

```
-- This is the Clash equivalent of the Haskell Prelude
import Clash.Prelude

-- What `main` is for a Haskell program, `topEntity` is for Clash.
topEntity :: Signal System Bit -> Signal System Bit
topEntity = id
```

This defines a circuit which takes as input a one-bit signal, and returns it. Note the type of `topEntity`: it is a function from the `Signal` that represents the input to a `Signal` representing the output. Even though our circuits in this chapter are not going to be clocked (we'll learn about clocks in more detail in the next chapter), we still need to pick *some* clock domain, so we'll use `System`, which is defined for us in the Prelude.

If we connect a pushbutton to the input and an LED to the output, we get a simple electronic device that lights up an LED as long as the button is pressed. Or, depending on the wiring of the LED and the pushbutton, maybe it is going to light up the LED as long as the button is *not* pressed.

2.3.1 Simulation

In its present form, we can't load this configuration onto a real FPGA board yet; we'll see shortly what else we need to add. But we can already play around with this circuit in the Clash simulator by loading it into the interpreter `clashi`, the Clash equivalent of `ghci`:

```
$ clashi Hello.hs
Clashi, version 1.4.2 (using clash-lib, version 1.4.2):
http://www.clash-lang.org/  :? for help
Ok, one module loaded.
```

As far as the Clash simulator is concerned, `topEntity` is a bona fide function, so we can apply it on an argument of type `Signal System Bit`:

```
> :t topEntity
topEntity :: Signal System Bit -> Signal System Bit
```

One way of getting our hands on a `Signal System Bit` value is to simulate a certain fixed sequence of events specified as a `[Bit]`. Let's see how our circuit would behave if we waited for two time steps, then pressed the button for one time step:

```
> let input1 = fromList [low, low, high]
> topEntity input1
0 0 1 *** Exception: X: finite list
```

Simulation failed after three clock cycles for the very simple reason that our input was only specified for three cycles. Let's fix that by appending infinitely repeating `low` values to our list:

```
-- we import Data.List qualified to avoid name conflicts with other
-- definitions exported by Clash.Prelude
> import qualified Data.List as L
> let input2 = fromList ([low, low, high] <> L.repeat low)
> topEntity input2
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
...
```

The simulation will run indefinitely, until the heat death of the universe, or until we kill it with Ctrl + C ; whichever happens first. So far, this author has always gone for the latter option.

Printing out the value of a signal timestep-by-timestep is useful for interactive examination, which is a fancy way of saying eyeballing. If we want to process it programmatically, it is usually a better idea to turn it into a list of values. This is what `sample` and `sampleN` do; the first one returns an infinite list, while the latter limits the simulation to *n* steps.

```
> L.take 20 $ sample $ topEntity input2
[0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
> sampleN 10 $ topEntity input2
[0,0,1,0,0,0,0,0,0,0]
```

Both `sample` and `sampleN` are lazy, so they will work with our finite-length `input1` as well:

```
> L.take 3 $ sampleN 10 $ topEntity input1
[0,0,1]
```

2.3.2 Synthesis

Staring at signal traces in the simulator can be very useful, and in fact later on, we will do all our debugging in the simulator (and various different levels of simulation!) instead of on actual hardware. This is also how "real" ICs are developed, even more so than FPGA configurations, since round-tripping via a chip fab is both slow and expensive.

But we're here to put *fun* in *functional programming*! So let's put simulation behind us for a second, and try to get some LEDs on our FPGA dev board to light up.

We *could* take our program as-is, and tell Clash to compile it into some HDL we can then feed into our FPGA synthesis toolchain. However, because our input and output signals are not named, the resulting HDL will use the Clash-generated name `c$arg` for our input port and the more reasonable `result` as the output port. By adding annotations to `topEntity`'s type signature, the Template Haskell function `makeTopEntity` gives meaningful names to our I/O ports:

```
import Clash.Prelude
import Clash.Annotations.TH

topEntity
    :: "BTN" ::: Signal System Bit
    -> "LED" ::: Signal System Bit
topEntity = id

makeTopEntity 'topEntity
```

Note that the types of inputs and outputs are annotated with a type-level string denoting the port name to use. Clash supports currying for multi-input circuits; but the result type has to be a single type, since Haskell doesn't have multi-valued functions. Multiple outputs will have to be represented as a single output of some tuple type; more on that later. In later code listings, we will skip the `makeTopEntity 'topEntity` line.

We can use the meta-command `:verilog` in `clashi` to compile `topEntity` into Verilog:[1]

```
GHC: Parsing and optimising modules took: 0.603s
GHC: Loading external modules from interface files took: 0.000s
GHC: Parsing annotations took: 0.000s
Clash: Parsing and compiling primitives took 0.249s
GHC+Clash: Loading modules cumulatively took 1.301s
Clash: Compiling Main.topEntity
Clash: Normalization took 0.012s
Clash: Netlist generation took 0.000s
Clash: Total compilation took 1.315s
```

[1] Why Verilog? Clash supports three mainstream Hardware Definition Languages: VHDL, Verilog, and SystemVerilog. SystemVerilog is not supported by older versions of the Xilinx ISE, needed to program popular FPGA dev boards such as the Papilio family, that are based on older Spartan-3 and Spartan-6 FPGAs; and Clash doesn't support VHDL for file-initialized ROM operations on Altera/Intel FPGAs. We'll be using Verilog throughout this book as the sweet spot.

This will create a file `verilog/Main.topEntity/topEntity.v`. How this is synthesized into FPGA configuration depends on what toolchain we use, which in turn is determined by the vendor of the FPGA chip we are targeting. One common trait among FPGA vendor tools is that they range from horribly painful to painfully horrible. In this book, interaction with the FPGA toolchain is restricted to a minimum by writing everything we can in Clash. However, mapping the port names such as `BTN` and `LED` to physical FPGA pins needs to be done outside Clash. Different toolchains, sometimes even from the same vendor, all have different formats for describing that mapping; usually, the easiest is to find this so-called *constraints file* on the dev board manufacturer's website and then tweak it to our needs.

Supposing we got over that hurdle, in the constraints file we can use the name `BTN` for one of the FPGA pins that is connected to a pushbutton on our development board. Alternatively, if we don't have a pushbutton, map `BTN` to a GPIO pin and connect a stand-alone pushbutton component to it via a breadboard. Similarly, map `LED` to a pin connected to an LED either directly, or via a breadboard. Then let the synthesis tool do its job to generate a *bitfile* , which is what can ultimately be uploaded to our FPGA.

2.3.3 Running on real hardware

Finally, we have our bitfile and are ready to upload it! The details of uploading, again, are FPGA- and dev board-specific. Once it's on the board, we can press the button we chose in our constraint mapping, and observe the LED lighting up. Release the button, and the LED goes out as well. Reactivity!

Or maybe it is happening the other way round: if the pushbutton is wired such that it pulls the input `low` when pressed, but the LED is wired such that `high` turns it on, then it will go out when the button is held. The next chapter will show that `Signal` is an applicative functor, so we can use `fmap complement` to flip the input `Bit`.

Either way, this is not quite *blinkenlights* yet: all the *blinken* is still missing! Blinking requires *state*, which we will explore after the next chapter.

Exercises:

- Try out what happens if we change `topEntity`'s type to be `Signal System Bool -> Signal System Bool`. Does its behavior change? Does its intended meaning?

2.4 Summary

- `Bit`: signal level of a single digital line. `low` and `high` doesn't necessarily mean `False` and `True`. `Active p` keeps track of this.
- `Signal`: discrete & synchronous time-varying value. Type-tagged with its clock domain.
- A Clash program's main entry point is `topEntity`, mapping input `Signals` to output `Signals`.
- Clash simulator converts between `Signal` and lists.
- Type-level port name annotation gives names to `topEntity` input and output signals to interface with the outside world.
- Clash compiles Haskell into HDL, which is then consumed by some FPGA toolchain to produce the bitfile containing the full FPGA configuration.

3 Combinational Circuits are Applicative Functors

Our first Clash program connected its one-`Bit` input directly to its one-`Bit` output. But what if we want to do some processing in between? For example, what if we want to flip the pushbutton's behavior so that the LED lights up when the button is released?

Circuits whose output can be described as a pure function of their inputs are called *combinational circuits*. In the flipped pushbutton case, we can say that the LED output is the result of applying the `complement` function on the button input.

3.1 `Signal` is an applicative functor

The idiomatic Haskell way to write the input-flipping circuit would be to use a `Functor` interface, and indeed Clash's `Signal` type is an instance of `Functor`:

```
topEntity
    :: "BTN" ::: Signal System Bit
    -> "LED" ::: Signal System Bit
topEntity = fmap complement
```

In fact, `Signal` is also `Applicative`, with `pure` being the constant signal and `<*>` applying function-valued signal on argument-valued signals in lockstep at every clock cycle. This allows us to make a more complicated circuit which only lights up the LED if two buttons are pressed at the same time, using the `Bits` instance of the `Bit` type to implement the actual conjunction:

```
topEntity
    :: "BTN_1" ::: Signal System Bit
    -> "BTN_2" ::: Signal System Bit
    -> "LED"   ::: Signal System Bit
topEntity btn1 btn2 = (.&.) <$> btn1 <*> btn2
```

This example also shows the support for curried functions for multi-input circuits: `topEntity` is a two-argument function, and both of its arguments are annotated with port names. To make a two-output circuit, for example showing the conjunction of two buttons on one LED and their disjunction on the other one, we can use a tuple in the result type. The resulting names will be `LED_1` and `LED_2`:

```
topEntity
    :: "BTN_1" ::: Signal System Bit
    -> "BTN_2" ::: Signal System Bit
    -> "LED" :::
           ( "1" ::: Signal System Bit
           , "2" ::: Signal System Bit
           )
topEntity btn1 btn2 =
    ( (.&.) <$> btn1 <*> btn2
    , (.|.) <$> btn1 <*> btn2
    )
```

Of course, tuple annotations also work on input arguments, allowing a slightly more regular-looking version:

```
topEntity
    :: "BTN" :::
           ( "1" ::: Signal System Bit
           , "2" ::: Signal System Bit
           )
    -> "LED" :::
           ( "1" ::: Signal System Bit
           , "2" ::: Signal System Bit
           )
topEntity (btn1, btn2) = (both, either)
  where
    both = (.&.) <$> btn1 <*> btn2
    either = (.|.) <$> btn1 <*> btn2
```

Before we move on to richer datatypes than `Bit`, it is perhaps worth noting here that since the kind of `Signal` is `Domain -> Type -> Type`, technically it is not `Signal`, but `Signal domain` (for any given type-level index `domain :: Domain`) that is an instance of the `Applicative` typeclass. This is exactly how Clash makes sure clock domains are not crossed unwittingly: if we have `b1 :: Signal Domain1 Bit` and `b2 :: Signal Domain2 Bit`, trying to compute their conjunction as `(.&.) <$> b1 <*> b2` is a type error.

3.2 `BitVector`s and `Vector`s

Beside representing single digital signal levels, the `Bit` type has another important role in Clash: values of certain datatypes can be taken apart into some set number of bits. This property is captured by the `BitPack` typeclass:

```
class BitPack a where
  type family BitSize a :: Nat
  pack :: a -> BitVector (BitSize a)
  unpack :: BitVector (BitSize a) -> a
```

Crucially, anything we want to use as input or output has to be an instance of `BitPack`, since ultimately everything has to become a bundle of wires each carrying a single digital 1-bit signal. This is not going to be a worry for us in this book: all our Clash code will interface directly with the outside world, so everything will be directly in terms of single bits or vectors of them.

The length-indexed type `BitVector :: Nat -> Type` shows up in the definition of `BitPack`. A `BitVector n` is a vector of n bits: the size is statically determined, in this case, by `BitSize a`. For example, `BitSize Bit` is 1 as expected; and correspondingly, `BitSize (Maybe Bit)` is 2. We can use the `(!)` operator to index into a `BitVector n`, or anything else that has a `BitPack` instance, starting at the least significant bit.

A more generic container for n-ary vectors of arbitrary element type is `Vec :: Nat -> Type -> Type`. In many ways, this plays the same role in Clash as `[] :: Type -> Type` in Haskell, as the bread-and-butter container type; in fact, the Clash Prelude binds familiar names like `zipWith`, `(++)` or `map` to their `Vec` version.

In a computer program, traversing a list is something that can be done element by element without knowing at compile-time how long the list is going to be. However, in a hardware circuit, transforming an n-element vector requires n sub-circuits whose outputs are bundled together to be taken as the result, so the length needs to be statically known to lay out the correct number of parts. This also means that while we can (and should!) write length-polymorphic functions, everything will need to have a concrete known length when accessed through the main top-level definition. `Vec` can also be used at our circuit's boundaries, since `Vector`s of `BitPack`able elements are `BitPack`able themselves.

As an aside, in later chapters we will see how to sequence steps and how to create RAM. With these tools, it is going to be possible to "process a list elementwise", but at a very different level of abstraction than just `fmap f someVector`.

3.3 Controlling many LEDs

Armed with `Vec`, we can write a circuit that controls many LEDs with many buttons. We'll use 8 inputs to drive 8 LEDs. This works with dev boards with more peripherals as well, since unused buttons and LEDs are simply not going to be connected to anything.

If your dev board has two-state switches, I recommend using those instead of pushbuttons for this coming example, unless you are an octopus.[1]

```
topEntity
    :: "SWITCHES" ::: Signal System (Vec 8 Bit)
    -> "LEDS"     ::: Signal System (Vec 8 Bit)
topEntity switches = reverse <$> switches
```

Hopefully no surprises here: the name `reverse` here of course refers to the function that operates on `Vec`s instead of lists.

Note that we have a *single* input port and a *single* output port, but both has a vector type. This works out because Verilog and VHDL support so-called *buses*: "wide" wires carrying multiple signals in parallel. Buses are a purely virtual construct, the outside world still only sees single pins, so the constraint files has to map individual pins to individual wires of a given signal. The exact syntax for this, again, varies slightly between FPGA toolchains, but it is going to look something like `SWITCHES<0>` or `SWITCHES[0]`. Buses are 0-indexed, so the above example will need both `SWITCHES` and `LEDS` mapped up to 7.

Synthesizing and uploading this program to our dev board should give us a new toy that turns LEDs on or off with switches. If we didn't use `reverse`, we could have used `BitVector 8` in our input and output types, but most `Vec` functions have no equivalents for `BitVector`. In this book, we are only going to use `BitVector` in cases where we want to reinterpret between different, `BitPack`able representations; collections of bits as a data structure will be represented as `Vec n Bit` instead.

3.3.1 Taking buses apart and putting them back together: the `Bundle` typeclass

The `Functor` interface to `Signal` makes it possible to take a signal containing many things and turn it into many signals. For example, we can take a bus of a 2-tuple into its two components:

[1] Latest Clash developer survey showed a minuscule ratio of octopus users compared to humans.

```
unbundleTuple
    :: Signal dom (a, b)
    -> (Signal dom a, Signal dom b)
unbundleTuple bus = (fst <$> bus, snd <$> bus)
```

We can do the same for vectors as well. Since vectors are length-indexed, we have to do a bit of type-directed programming to generate the vector of indices $\langle 0, 1, \ldots, n-1 \rangle$ by iterating the successor function n times, using `iterateI` from the Clash Prelude.[2]

```
unbundleVec
    :: (KnownNat n)
    => Signal dom (Vec n a)
    -> Vec n (Signal dom a)
unbundleVec bus = map (\i -> (!!i) <$> bus) indices
  where
    indices = iterateI (+1) 0
```

Moreover, `Signal` is also `Applicative`, giving us a way to put it back together. For tuples, we can do it directly:

```
bundleTuple
    :: (Signal dom a, Signal dom b)
    -> Signal dom (a, b)
bundleTuple (fsts, snds) = (,) <$> fsts <*> snds
```

and for `Vec n`, we can take advantage of the fact that the latter is `Traversable` (if only $n \geq 1$), by using `sequenceA`:[3]

```
bundleVec
    :: (KnownNat n)
    => Vec n (Signal dom a)
    -> Signal dom (Vec n a)
bundleVec Nil = pure Nil
bundleVec xs@(Cons _ _) = sequenceA xs
```

That is a lot of code for something that does so little! In fact, all these functions should be *doing* nothing at all: the signal *values* are not changed by them, only their *geometry*. It should come as no surprise that we don't need to write any of the

[2] Clash also has `indicesI :: (KnownNat n) => Vec n (Index n)`; we are implementing `indices` ourselves here purely for educational reasons.

[3] We are using `Cons` here instead of `:>` like everywhere else, due to a subtle Clash bug. See https://github.com/clash-lang/clash-compiler/issues/867 and https://github.com/clash-lang/clash-compiler/pull/966 for the details.

above code: the Clash Prelude comes with a typeclass called `Bundle` that gives us an overloaded version of `bundle` and `unbundle`.

```
class Bundle a where
  type Unbundled (dom :: Domain) a :: Type
      = res
      | res -> dom a
  bundle :: Unbundled dom a -> Signal dom a
  unbundle :: Signal dom a -> Unbundled dom a
```

We can now write a version of our LED strip demo that computes each element of our resulting vector separately, before bundling them up into a vector.

```
topEntity
    :: "SWITCHES" ::: Signal System (Vec 2 Bit)
    -> "LEDS"     ::: Signal System (Vec 4 Bit)
topEntity switches = fmap (map boolToBit) . bundle $
    both :> either :> onlyOne :> onlyTheSecond :> Nil
  where
    sw1 :> sw2 :> Nil = unbundle $ map bitToBool <$> switches

    both = sw1 .&&. sw2
    either = sw1 .||. sw2
    onlyOne = sw1 ./=. sw2
    onlyTheSecond = (not <$> sw1) .&&. sw2
```

Here, we used `.&&.`, `.||.` and `./=.` which are `Signal`-lifted versions of `&&`, `||` and `/=`.

Exercises:

- Rewrite `LEDStrip2` using the `ApplicativeDo` language extension
- `LEDStrip2` is, of course, still a combinational circuit. Rewrite it without using `bundle` / `unbundle`, in the form of `f <$> switches`.

3.4 Seven-segment display

A segmented LED display is a bunch of LEDs arranged in multiple line segments in a layout that allows drawing interesting patterns that can be recognized as digits or letters. By far the most common setup is seven segments and everyone knows them from basic calculators, where the same layout is used in LCDs. Other displays use 14 or 16 segments by adding diagonals and breaking up the horizontal segments into two parts.

In this section, we will write Clash code to interface with a seven-segment LED display — commonly found on FPGA dev boards — to show a four-bit number as a single hexadecimal digit.

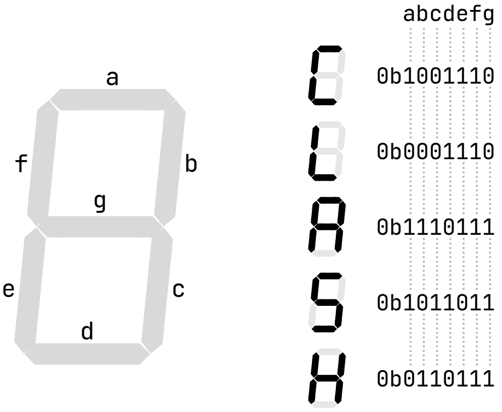
3.4.1 How a seven-segment LED display works

The fundamental idea behind a seven-segment LED is to be able to turn on or off each segment separately. So each segment gets its own LED, and each LED has a separate pin to the outside world. Of course, you actually need two leads for each LED, but keeping one of them constant and changing the other is enough to turn them on or off. Thus, seven-segment LEDs are either *common anode* or *common cathode* for a total of 8 pins (9 if there is also a decimal point).

On a common anode device, to display anything, the shared anode pin needs to be set to `high` and individual segments can be turned on by setting their pins to `low`. Setting a segment to `high` turns that segment off, and setting the shared anode to `low` turns the whole display off, regardless of the individual segment pins, since LEDs only light up if the voltage across them has the right direction. On a common cathode device, everything is reversed: the display is on if the shared cathode is `low` and individual segments are `high`.

In our code, we will represent the segments as a `Vec 7 Bool`, and convert that to active high or active low representation at the edges, using `toActive`.

As for mapping the segments themselves, on seven-segment display manufacturer's datasheets the segments are usually labeled `a` to `g` in a spiral sequence:



The following function takes the state of the seven segments and renders it using ASCII art. This is going to be useful for debugging in the Clash simulator. Note that we are using list functions and `String`s freely here, since this is not meant to be synthesized: this is plain-old Haskell code here.

```
import Clash.Prelude
import qualified Data.List as L

showSS :: Vec 7 Bool -> String
showSS (a :> b :> c :> d :> e :> f :> g :> Nil) = unlines . L.concat $
    [ L.replicate 1 $ horiz   a
    , L.replicate 3 $ vert  f   b
    , L.replicate 1 $ horiz   g
    , L.replicate 3 $ vert  e   c
    , L.replicate 1 $ horiz   d
    ]
  where
    horiz True  = " ###### "
    horiz False = " ...... "

    vert b1 b2 = part b1 <> "      " <> part b2
      where
        part True  = "#"
        part False = "."
```

Let's try drawing a `5`! It should have segments *a*, *f*, *g*, *c* and *d* set:

```
ss5 :: Vec 7 Bool
ss5 =  True :> False :> True :> True :> False :> True :> True :> Nil

> putStrLn $ showSS ss5
 ######
#      .
#      .
#      .
 ######
.      #
.      #
.      #
 ######
```

We can hook this up to real hardware by pairing it with a decimal point (unset) and the digit selector anodes/cathodes depending on the board. In the code below, we are using a `Vec 4 Bool` to select which digits are on; the number 4 should be the number of digits in our seven-segment display. You should also adapt the choice of `Active Low` and `Active High` types to suit your display wiring.

```
topEntity
    :: "SS" :::
           ( "AN"  ::: Signal System (Vec 4 (Active High))
           , "SEG" ::: Signal System (Vec 7 (Active Low))
           , "DP"  ::: Signal System (Active Low)
           )
topEntity =
    ( map toActive <$> anodes
    , map toActive <$> segments
    , toActive <$> dp
    )
  where
    anodes = pure $ False :> False :> False :> True :> Nil
    segments = pure ss5
    dp = pure False
```

The value of `False :> False :> False :> True :> Nil` means only the last digit (out of four) is on. This requires the digit selector pins to be numbered so that 0 is the **rightmost** digit. But the 0th should be the vector's **leftmost** element? That is true when indexing (so `('a' :> 'b' :> Nil) !! 0` returns `'a'`), but in buses, such as a vector-valued signal, the order is most to least significant bits going left to right, just like when writing out a digit. This works out very nicely here, because `False :> False :> False :> True :> Nil` matches visually the state of the digits resulting from it, that is, *off, off, off, ON*.

3.4.2 `Unsigned` and `Signed`

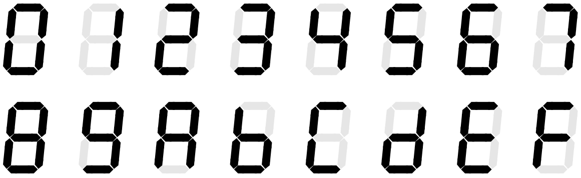
While the code above works fine, it is lacking a lot in interactivity: all it does is display the number 5. So let's change it a bit so that we can use four two-state switches to input a four-bit number, and then we can display that number as a single hexadecimal digit.

To do this, first of all we need to represent a four-bit number. Just like vectors that are tracked in the type system to have n elements, Clash offers types for n-bit `Signed` and `Unsigned` numbers. The only difference between these two types is that the range of representable values is $(-2^{n-1}) \ldots (2^{n-1} - 1)$ for `Signed n` (using two's-complement format) and $0 \ldots (2^n - 1)$ for `Unsigned n`.

Both types are instances of `Eq`, `Ord`, `Num` etc. as one would expect (the `Num` instance is overflowing, e.g. `6 + 4 :: Unsigned 3` is 2). They are also instances of `BitPack`, with `BitSize (Unsigned n) ~ n` and `BitSize (Signed n) ~ n`, no surprises here. At the edges, we can use this `BitPack` instance to convert from/to a `Vec n Bit`, possibly composed with a `map complement` to deal with active-low inputs/outputs.

For some special values of `n`, `Signed n` and `Unsigned n` behave exactly like the standard Haskell `IntN` and `WordN` types, e.g. `Unsigned 8` and `Word8` both represent the same values with the same semantics; the `BitPack` instance of both ensures we can go between them with `bitCoerce`. This can be very useful when interfacing with Haskell libraries.

If we want to represent a single hexadecimal digit to be shown on a seven-segment display, `Unsigned 4` is a great choice. In the next chapter when we get to driving multiple seven-segment displays, we will see that because `BitPack` is closed under products, we can use `bitCoerce` to turn e.g. an `Unsigned 16` into a four-tuple of `(Unsigned 4, Unsigned 4, Unsigned 4, Unsigned 4)` to display a 16-bit number using four hexadecimal digits. For now, we are going to concentrate on a single such digit.



Turning the 4-bit number into a seven-segment pattern is easy with a combinational circuit. We just implement a pure function to “render” hexadecimal digits into segments:

```
encodeHexSS :: Unsigned 4 -> Vec 7 Bool
encodeHexSS n = unpack $ case n of
    --        abcdefg
    0x0 ->  0b1111110
    0x1 ->  0b0110000
    0x2 ->  0b1101101
    0x3 ->  0b1111001
    0x4 ->  0b0110011
    0x5 ->  0b1011011
    0x6 ->  0b1011111
    0x7 ->  0b1110000
    0x8 ->  0b1111111
    0x9 ->  0b1111011
    0xa ->  0b1110111
    0xb ->  0b0011111
    0xc ->  0b1001110
    0xd ->  0b0111101
    0xe ->  0b1001111
    0xf ->  0b1000111
```

We can try it out e.g. to make sure `6` and `b` are discernibly different:

```
> putStrLn $ showSS $ encodeHexSS 0x6
 ######
#      .
#      .
#      .
 ######
#      #
#      #
#      #
 ######

> putStrLn $ showSS $ encodeHexSS 0xb
 ......
#      .
#      .
#      .
 ######
#      #
#      #
#      #
 ######
```

Looks good to me! Let's hook up some input switches to `encodeHexSS` and route it to the segments:

```
topEntity
    :: "SWITCHES" ::: Signal System (Vec 4 Bit)
    -> "SS" ::: ( "AN"  ::: Signal System (Vec 4 (Active High))
                , "SEG" ::: Signal System (Vec 7 (Active Low))
                , "DP"  ::: Signal System (Active Low)
                )
topEntity sws =
    ( map toActive <$> anodes
    , map toActive <$> segments
    , toActive <$> dp
    )
  where
    anodes = pure $ False :> False :> False :> True :> Nil
    digit = bitCoerce <$> sws
    segments = encodeHexSS <$> digit
    dp = pure False
```

Exercises:

- Wire up some of the unused switches to select which seven-segment digit to turn on
- Change the input to an `Unsigned 8`, show its lower 4 bits as the digit, but turn on the decimal point if the value is greater than 15.

3.4.3 What about multiple digits?

A fairly natural way of extending our circuit would be to use more of the seven-segment digits. Most FPGA development boards have seven-segment displays with four or eight digits; we control which ones are on by setting the `anodes` signal. However, much more useful would be to not only turn on all digits, but also to show different figures on each of them. However, at first glance there aren't enough degrees of freedom exposed for it: on a four-digit display, we have 7 segment inputs, one decimal point input, and four digit selectors; how does that allow setting each digit separately?

In the next chapter, we will learn about stateful circuits and sequencing, which will then allow us in the subsequent chapter to show multiple digits exploiting the persistence of vision: flashing each digit while turning on a different display, when done at a rapid enough rate, gives the illusion that each digit is shown at the same time.

3.5 Summary

- `Signal` is an instance of `Applicative`, corresponding to combinational circuits
- `BitPack` typeclass turns values into fixed-width `BitVector`s and back.
- `Bundle` typeclass makes it easy to take apart and put back together buses
- `Vec` is the statically sized equivalent to lists. Its known size makes it possible to traverse it in a circuit.
- `Unsigned` and `Signed` are integers of any, statically known, size.

4 State, Sequencing and Clocks: The Register Transfer-Level Model

Since a combinational circuit can be modeled as a pure function applied to an input signal, it follows that whenever the input signals have the same value, the output signals will also have the same value. But there are lots of useful circuits where this doesn't hold, for example:

- **Blinking LEDs** (this and next chapter): who doesn't like blinking LEDs?! In its simplest form, a blinking circuit has no inputs, yet its output is sometimes on, sometimes off.

- **Turning an LED on and off with a pushbutton switch** (this chapter). This is different from the example that used a toggle switch because now when the button is released, its result depends on what happened before:

Timestep	Button held?	LED on?
1.	False	False
2.	False	False
	. . .	. . .
10.	True	True
11.	False	True
12.	False	True
	. . .	. . .
20.	True	False

- **Edge detection**. This is used for synchronous communication protocols (e.g. the PS/2 protocol described in chapter 16) where there is an explicit clock signal between the communicating parties, and the data channel should be sampled on edges of the clock. *Clock edge* here means the instant when the clock signal goes from `high` to `low` (or vice versa depending on the protocol). But this requires knowing what the earlier value of the clock signal was: a `low` value means a "clock tick" only if the value was `high` up until that point.

- **Keeping time**. So-called asynchronous serial protocols (chapter 10) have no shared clock signal: each communicating party is responsible for keeping its own time to know when new data is available (or, in the case of the sender, should be made available) on the bus. A constant input of `high` values might represent one, two, or more bits of data depending on how long the value has been there.

4.1 Clocks and registers

In Haskell we can model stateful computations over some input as a function `f :: a -> s -> (b, s)`, that is, a pure function that takes as extra argument its current state and returns not just its result, but also its new state. In the setting of circuits, there are more details to work out:

1. How often is the state updated? In other words, if the input `a` doesn't change, what is the time period for which the output is held at the `b` value calculated from the current `s` value, before the calculation is re-done for the new `s` value?
2. Where do we store the `s` value calculated in the previous step, so that it can be fed back as the new state?

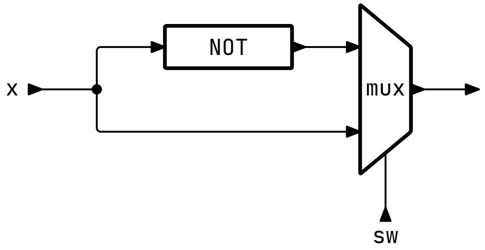
One possible way to answer these is to "just wire everything together", i.e. to feed the output `s` directly into the input. In that case, the update frequency is determined by the propagation delay of `f`: every lookup table, every transistor, even the lengths of the wires involved all contribute to some non-zero time delay it takes for voltage changes to propagate through the circuit. `s` is not explicitly stored anywhere: the *circuit itself* "stores" the state.

This "solution" has so many problems that it is not supported by Clash at all. The propagation delay of `f` is necessarily intensional: different implementations of the same function, yielding different circuits, will have different propagation delays. And since different FPGA models will have parts with different characteristics, calculating the delay of a given Clash function would require seeing all the way through the Clash compiler and the FPGA toolchain.

Basic composition becomes impossible: consider the simple example of a branch:

```
-- This is defined in the Clash Prelude as an easy way to use
-- if/then/else over applicative functors
mux :: (Applicative f) => f Bool -> f a -> f a -> f a
mux = liftA3 $ \cond thn els -> if cond then thn else els

myCircuit :: Signal dom Bool -> Signal dom Bit -> Signal dom Bit
myCircuit sw x = mux sw (complement <$> x) x
```

Here, `myCircuit` can be implemented as a circuit that splits the input wire into two, and feeds both wires into a multiplexer: one as-is, and the other via a `NOT` gate.



However, because the `NOT` gate has its own delay, the two inputs to the multiplexer will not be ready at the same time. But the multiplexer's selector input (`sw` in `myCircuit`'s case) would need to have the right delay so that regardless of which branch is taken, the new signal level is already at the multiplexer right when the selector changes. So now its delay would need to depend on the branch taken, otherwise the signal is unstable.

4.1.1 Synchronicity

A different approach to answering the details of stateful circuits is to use a synchronous model. Here, the state is updated whenever some external, regular clock ticks. This avoids all the problems with propagation delay as long as the clock period is long enough that everything settles in time before the next tick. Since all circuits use the same clock, they can be freely composed: no matter how signal changes propagate mid-period, new values are only considered at the end of the period. In our previous example with the multiplexer, it doesn't matter that the two branches have different propagation delays, because if `myCircuit` is used in the definition of a stateful computation, the state is only updated with the stabilized value after enough time has passed.

While this approach neatly solves all the problems we had with stability, it introduces two new problems:

- The clock rate must be selected correctly. If there are any signal paths through the circuit that have a propagation delay longer than the clock period, we are back to square one. However, the whole design can be written with the assumption that the clock period is correct, and then the maximal delay can be computed and checked for the circuit at the end (and FPGA toolchains do this as part of the synthesis process). So even if it is discovered that the targeted clock rate is too fast, we can either just decrease it (if the problem allows for it) or start optimizing the critical parts of the circuit, without having to rewrite everything to account for changed propagation delays.

- State needs to be stored somewhere during clock periods. Since the whole point of the synchronous model is to avoid intra-period instabilities to lead to oscillation, the newly computed state cannot be fed back directly as the input state. Instead, at every tick of the clock, the state output of the circuit is latched into a register, and it is the register's value that is fed back.

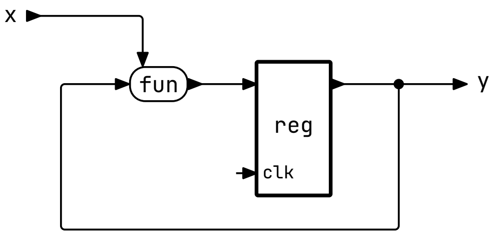
The two basic building blocks of this design are, thus, the *clock* and the *registers*.

The *clock* is an external entity that sets everything else in motion. Usually, in an electronic circuit there is a physical clock, with a set clock rate, based on an oscillating quartz crystal, and other components that can generate variable-rate clocks from that single shared base clock. The output of the clock is a one-bit square wave; the "tick" and the "tock" of the clock are the rising and falling edges of this signal. In this book, we will take the clock as given (i.e. as an input signal) in our Clash circuits; and for designs where it matters, we will configure it outside Clash to be at the right frequency.

A *register* holds a set number of bits at the last given value, until that value is overwritten at the next clock tick. A register, thus, has two inputs and one output at minimum: the clock and the write value are its inputs, and the currently held read value is its output. Clash adds two more inputs to each register: a *reset* input that loads the initial value of the register, regardless of the write value, and an *enable* input which controls whether the whole component is live or not.

4.2 The RTL model: `register` and delayed feedback

Armed with clocks and registers, we can implement synchronous stateful circuits using *feedback*, *delayed* via a register:



Now we can describe the whole circuit as a register whose read output and write input are connected to a combinational circuit. Similar to the combinational circuit model, this model is also closed under composition, by just composing the pure functions and storing the product of all states in a single register.

This model is widely used to describe digital circuits and is known as the *Register-Transfer level* model, or *RTL* for short, since it describes the design in terms of the combinational circuits through which signals are transferred between registers. RTL is the basis of not just Clash, but also Verilog/SystemVerilog and VHDL, the two mainstream hardware description languages. FPGA synthesis toolchains all work by consuming RTL descriptions; it is fair to say that the RTL model is the bread and butter of FPGA design, and this is no different in Clash.

The Clash primitive to create a register is called, unsurprisingly, `register`. Because writing code that connects the same clock, reset and enable signals to all registers would be very tedious, Clash provides a typeclass-based shorthand for all `register`-based operations. In this book, we are going to use these convenience functions; in fact, if we import `Clash.Prelude`, these are the versions brought in scope. But for this section only, we are first going to use `Clash.Explicit.Prelude` to make it more obvious how the code maps to the above RTL diagram.

The type of this explicitly-clocked version of `register` is:

```
Clash.Explicit.Prelude.register
    :: (KnownDomain dom, NFDataX a)
    => Clock dom -> Reset dom -> Enable dom
    -> a -> Signal dom a -> Signal dom a
```

The typeclass constraint `KnownDomain dom` prescribes that `dom` is a valid type-level tag for a clock domain. `NFDataX a` is the Clash simulator's requirement on `a` for simulating uninitialized or invalid values; we are not going to concern us with them in this book, other than using GHC's generics to derive the `NFDataX` instance for all our datatypes to satisfy `register`.

The first argument, of type `Clock dom`, is the clock signal driving the writing into the register value. The second argument, of type `Reset dom`, is the reset signal that, when asserted, replaces the register value with the initial one. The `Enable dom` argument gates the register transfer; it is basically a glorified `Signal dom Bool`, and setting it to `False` causes the `register` to ignore any incoming new value.

The fourth argument is the initial value of the register: this value is what is read from the register before the first write, and also the value loaded back on reset. The fifth argument is what is written into the register on each tick of the clock.

4.2.1 Flipping a `Bool` once

Let's make a circuit that takes the control signals (clock, reset and enable) as inputs, and creates possibly the most boring stateful circuit: one that outputs a `True` signal in the first cycle, and `False` afterwards. This is not quite using the RTL model to its fullest, since there is no feedback yet.

```
import Clash.Explicit.Prelude

helloRegister
    :: Clock System -> Reset System -> Enable System
    -> Signal System Bool
helloRegister clk rst en = register clk rst en True (pure False)
```

We didn't bother with port name annotations here because we are only going to play around with this code in the simulator. If we load it into `ghci` and try to use it with `sampleN`, the first hurdle is coming up with `clk`, `rst` and `en` inputs. In a real circuit, the clock would come from components such as the clock manager that turns the raw quartz crystal's oscillation into the right clock rate. Similarly, `rst` is often connected to a physical button, or there is a reset circuit which provides a reset pulse as the board is brought up. As for `en`, in the simplest case we want to keep our circuit ticking away unconditionally. In fact, for simplicity's sake, we are not going to use the `Enable` lines at all in this book. Instead, we will use simple `Bool`-valued signals in conjunction with `mux` whenever we need conditional updates.

Since we wouldn't otherwise have access to the clock and reset button peripherals inside the simulator, Clash provides the functions `clockGen` and `resetGen`. A third function, `enableGen`, is provided to create an always-enabled line. Note that `clockGen` is not synthesizable (we can't make a clock signal from thin air), so we can't use it in a real circuit we intend to run on a real FPGA, only for simulation. `resetGen` and `enableGen`, on the other hand, are usable even in a real circuit – `resetGen` simply generates a never-asserted reset line, and `enableGen` always enables every component.

In the simulator, then, we can do the following:

```
> sampleN 5 $ helloRegister clockGen resetGen enableGen
[True,True,False,False,False]
```

This is. . . quite unexpected, isn't it? Why is the output `True` for two cycles instead of just one? The reason is that the first simulated timestep is *before* the first clock cycle; then in the first cycle, the value coming from the register into the output is `True`, and `False` is written back to the register for the next cycle.

When designing integrated circuits instead of targeting an FPGA, register values might be undefined before being reset. The simulated behavior before the first clock cycle is configurable in the clock domain, but IC design is beyond the scope of this book. Luckily, the `System` domain's configuration matches the behavior of FPGAs.

4.2.2 Flipping a `Bool`, repeatedly

To write a more interesting circuit that doesn't just get stuck in one state after the initial clock cycle, let's create a proper RTL circuit that uses `register`-delayed feedback to keep flipping between `True` and `False`:

```
flippy
    :: Clock System -> Reset System -> Enable System
    -> Signal System Bool
flippy clk rst en = r
  where
    r = register clk rst en True (not <$> r)
```

The money shot is in the definition of `r`, which is recursive but its recursion is *guarded* under a call to `register`. To get an idea of what is happening, let's return both `r` and `not <$> r`, and look at it per timestep:

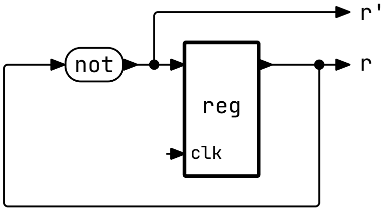
```
flippy
    :: Clock System -> Reset System -> Enable System
    -> Signal System (Bool, Bool)
flippy clk rst en = bundle (r, r')
  where
    r = register clk rst en True r'
    r' = not <$> r
```

In the simulator, we see:

```
> mapM_ print $ L.zip [0..] $ sampleN 8 $ flippy clockGen resetGen
    enableGen
(0,(True,False))
(1,(True,False))
(2,(False,True))
(3,(True,False))
(4,(False,True))
(5,(True,False))
(6,(False,True))
(7,(True,False))
```

In the first cycle, `r` is `True` (its initial value), while `r'` is `False`, as computed by the combinational circuit `fmap not` from `r`. By the second cycle, this value of `r'` is written into `r`, so now its value is `False`, which of course also means that `r'` is now changed to `True`.

We can instantiate our previous diagram for `flippy` as such:



Exercise:

- Write a "slow `flippy`" that goes `True`, `True`, `False`, `False`, `True`, `True`, . . . Hint: use `register` twice.

4.3 Finally blinkenlights!

We now have everything to finally do what every book on electronics for hobbyists must do in one of its first chapters: make some LEDs blink.

We could say that the `flippy` circuit already does that, but that one is too fast: if clock is running at 100 MHz, the LED will turn on and off 50 million times a second, which is not going to be perceptible. The idea here is to slow this down by counting to a sufficiently large number, and taking its most significant bit as the LED output. Because `Unsigned`'s `Num` instance implements overflow semantics, once it gets to the largest possible `Unsigned n` value, adding one more to it will simply reset it by wrapping around to 0.

We'll choose `n` such that 2^n clock periods takes roughly 1 second:

```
type SecondPeriods dom = 1_000_000_000_000 `Div` DomainPeriod dom

blinkingSecond
    :: forall dom. (KnownDomain dom)
    => (1 <= DomainPeriod dom, KnownNat (DomainPeriod dom))
    => (1 <= 1_000_000_000_000 `Div` (DomainPeriod dom))
    => Clock dom -> Reset dom -> Enable dom
    -> Signal dom Bit
blinkingSecond clk rst en = msb <$> r
  where
    r :: Signal dom (Unsigned (CLog 2 (SecondPeriods dom)))
    r = register clk rst en 0 (r + 1)
```

Here, we use the type-level function `DomainPeriod` to extract the clock period from the clock domain. The extended typeclass constraints on `DomainPeriod` are needed to be able to do the type-level computation to convert the period (in picoseconds) to periods per second (i.e. Hz). `CLog 2 periods` then computes the ceiling of the 2-base logarithm of `periods`, i.e. the number of bits needed to represent that number. For a 100 MHz clock, this will result in `r :: Signal dom (Unsigned 27)`.

The main circuit then is using `blinkingSecond` to drive the LED output:

```
topEntity
    :: "CLK" ::: Clock System
    -> "LED" ::: Signal System Bit
topEntity clk = blinkingSecond clk resetGen enableGen
```

If we synthesize that and load it onto a real FPGA board, we will find that while the LED does blink, it is not doing it at quite the required frequency. There are two reasons for that.

4.3.1 A more accurate timing counter

The first reason is that the power-of-2 approximation of 1 second can be quite inaccurate: for example, with a 100 MHz clock, counting to 2^{27} takes 1.34 seconds. We can improve that by using that same 27-bit counter to only count up to 100 million. Let's also use this opportunity to factor out the conversion of frequency (in Hertz) to clock period length (in picoseconds), i.e. a type-level version of the Clash prelude's `hzToPeriod` function:

```
type HzToPeriod (freq :: Nat) = 1_000_000_000_000 `Div` freq

type ClockDivider dom ps = ps `Div` DomainPeriod dom

blinkingSecond
    :: forall dom. (KnownDomain dom)
    => (1 <= DomainPeriod dom, KnownNat (DomainPeriod dom))
    => (1 <= HzToPeriod 1 `Div` DomainPeriod dom)
    => Clock dom -> Reset dom -> Enable dom
    -> Signal dom Bit
blinkingSecond clk rst en = msb <$> r
  where
    r :: Signal dom (Unsigned (CLog 2 (ClockDivider dom (HzToPeriod 1))))
    r = register clk rst en 0 $ mux (r .<. limit) (r + 1) 0

    limit = snatToNum (SNat @(ClockDivider dom (HzToPeriod 1)))
```

This is better in that the total LED blinking period is indeed going to be 1 second. However, our 27-bit counter's most significant bit will be unset for 671 ms (as it counts up to $2^{26} - 1$) and set for 329 ms (as it counts from 2^{26} to 100,000,000), so the LED will be off for roughly two-third of the period and on for one-third. Of course, if all we want to do is blink an LED "per second", that problem is underspecified enough that this could be good enough. But in general, we would like more control over the duty cycle. So let's aim for restoring the 50% duty cycle: we want our LED to be off for half a second and on for half a second.

A neat way of doing that is to have a counter that counts up to exactly half a second's worth of periods, and then changes state. In the half-and-half scenario, we could get away with the state being a `Bool`, but we'll use a sum type of two counters to show how this would generalize for arbitrary duty cycles.

Instead of an `Unsigned 27` to count up to 50,000,000, we are going to use a type that has exactly 50,000,000 values: this is the `Index 50_000_000` type from Clash's prelude. It is similar to an `Unsigned` type (and its `BitSize` is the same as a just-large-enough `Unsigned`), but its `Num` and `Bounded` instances implement the right behavior for the given cardinality. Also note how addition doesn't wrap over:

```
> [minBound .. maxBound] :: [Index 14]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13]
> (10 :: Index 14) + 3
13
> (10 :: Index 14) + 4
*** Exception: X: Clash.Sized.Index: result 14 is out of bounds: [0..13]
> succ (13 :: Index 14)
*** Exception: X: Clash.Sized.Index: result 14 is out of bounds: [0..13]
```

We can implement `succ`- and `pred`-like functions that handle `Index` types without exceptions; these functions are going to be useful in a lot of later chapters.

```
succIdx :: (Eq a, Enum a, Bounded a) => a -> Maybe a
succIdx x | x == maxBound = Nothing
          | otherwise = Just $ succ x

predIdx :: (Eq a, Enum a, Bounded a) => a -> Maybe a
predIdx x | x == minBound = Nothing
          | otherwise = Just $ pred x
```

With these, we can rewrite `blinkingSecond` to have 0.5 seconds (500 ms, or 500,000,000,000 ps) off, 0.5 seconds on. Our state will now be one of two `Index` counters: either counting how long the LED should be `On`, or counting how long it should be `Off`:

```
data OnOff on off
    = On  (Index on)
    | Off (Index off)
    deriving (Generic, NFDataX)

isOn :: OnOff on off -> Bool
isOn On{}  = True
isOn Off{} = False
```

In every clock cycle, we increment the current counter if possible; if not, we start the other counter at 0 instead.

```
countOnOff :: (KnownNat on, KnownNat off) => OnOff on off -> OnOff on off
countOnOff (On  x) = maybe (Off 0) On  $ succIdx x
countOnOff (Off y) = maybe (On  0) Off $ succIdx y
```

We also move to streamlined typeclass constraints using `PartialTypeSignatures`, to avoid the tedium of listing out all the combinations of `1 <=` and `KnownNat` that GHC happens to require:

```
blinkingSecond
    :: forall dom. (KnownDomain dom, _)
    => Clock dom -> Reset dom -> Enable dom
    -> Signal dom Bit
blinkingSecond clk rst en = boolToBit . isOn <$> r
  where
    r :: Signal dom
         (OnOff
             (ClockDivider dom (500_000_000_000))
             (ClockDivider dom (500_000_000_000)))
    r = register clk rst en (Off 0) $ countOnOff <$> r
```

Did we get the number of zeroes right in `500_000_000_000`? To avoid having to calculate with billions of picoseconds, we can add some larger units: a `Second` is a thousand `Milliseconds`, a `Millisecond` is a thousand `Microseconds` and so on.

```
type Seconds      (s  :: Nat) = Milliseconds (1_000 * s)
type Milliseconds (ms :: Nat) = Microseconds (1_000 * ms)
type Microseconds (us :: Nat) = Nanoseconds  (1_000 * us)
type Nanoseconds  (ns :: Nat) = Picoseconds  (1_000 * ns)
type Picoseconds  (ps :: Nat) = ps
```

This allows rewriting `blinkingSeconds` as:

```
blinkingSecond clk rst en = boolToBit . isOn <$> r
  where
    r :: Signal dom
         (OnOff
             (ClockDivider dom (Milliseconds 500))
             (ClockDivider dom (Milliseconds 500)))
    r = register clk rst en (Off 0) $ countOnOff <$> r
```

4.3.2 Customizing the clock

Uploading the final version of `blinkingSecond`, depending on the FPGA board, we will either see it work as expected (blinking at 1 Hz), or running slower or faster.

As we promised, there are two reasons for not getting the right blinking speed. The second one is using the `System` clock in Clash, when that might not have the right configuration for the actual clock on our dev board. The `System` clock is defined to run at 100 MHz, so all period calculations will use that as reference. If the real clock connected to the `CLK` input is running at, let's say, 32 MHz, then all calculations will be off by a factor of three, so our LED will blink at one-third of the designed speed.

This bridge can be crossed from either direction. The more straightforward one is to tell Clash what clock to use. Since the clock domains are defined at the type level, this would involve some mucking around with typeclass instances and singletons; it is much easier to just use the `createDomain` Template Haskell function, provided by the Clash standard library:

```
createDomain vSystem{vName="Dom32", vPeriod = hzToPeriod 32_000_000}

topEntity
    :: "CLK" ::: Clock Dom32
    -> "LED" ::: Signal Dom32 Bit
topEntity clk = blinkingSecond clk resetGen enableGen
```

Since `blinkingSecond`'s definition is polymorphic in the clock domain, this version of `topEntity` will do all type-level calculations with a 32 MHz clock, resulting in a 25-bit counter instead of the 27-bit one for `System`.

The second way to ensure Clash calculates all clock rate computations correctly is to change our clock to really run at 100 MHz. Most FPGAs have components to manage clock speed: these are components that take the raw clock as input, and using divisions and multiplications, produce the desired clock rate. The division part is straightforward counting just like we did in our `blinkingSeconds` function; the multiplication is more involved and requires some specialized physical parts and careful feedback loops to keep the multiplied clock regular. Luckily, all this is abstracted away behind the clock manager, and it is enough to configure it with just the input clock rate and the desired output clock rate.

In this book, some of the circuits will need to adhere to external timing constraints prescribed by various protocols. For example, asynchronous serial communication (chapter 10) at a set rate, let's say 9600 bits per second, requires both parties to have the same idea of how long 9.6 kHz is. The actual internal clock rate of the two parties doesn't matter, and 9.6 kHz is very slow compared to the tens or hundreds of MHz speeds that FPGA clocks use by default, so the same kind of counting we did in `blinkingSeconds` can be used instead with good enough accuracy, as long as Clash is configured for the right clock speed.

However, once we get to generating video output in chapter 7, we will have to start feeding Clash a clock at just the right speed, using an external clock manager. To see why, consider trying to generate a 640 × 480 VGA signal at 60 frames per second. As we will see in the chapter on VGA modes, this requires pumping out pixel data at 25.125 MHz. If, let's say, we start from a 32 MHz raw clock, there is no way to update the output at the desired rate of 25.125 MHz, since the closest we can get would be either 32 MHz (using no division) or 16 MHz (counting to 2). Of course, both 32 and 16 MHz rates are far far far beyond the timing tolerances of VGA devices. And so when we'll get to building computers with video output, we will always prescribe the exact clock speed, and use the targeted FPGA's clock manager to produce the right clock. For this reason, we will go into the practical details of using a clock manager in chapter 7.

4.4 Passing around Clock, Reset and Enable lines implicitly

Our blinking LED example is small enough that it is not too bothersome to pass around the `clk`, `rst` and `en` arguments. However, in a larger circuit, since every `register` call needs these three arguments, this can add up to quite a lot of noise. Moreover, there is not much we can do with these signals other than passing them around.

To mitigate this, Clash provides two version of `register`, and all other functions that use `register` internally: `Clash.Explicit.Prelude` exports them as straight functions from `clk`, `rst` and `en`; whereas `Clash.Prelude` exports them with the typeclass constraint `HiddenClockResetEnable dom` taking the place of explicit arguments.

Compare the two type signatures of `register`:

```
Clash.Explicit.Prelude.register
    :: (KnownDomain dom, NFDataX a)
    => Clock dom -> Reset dom -> Enable dom
    -> a -> Signal dom a -> Signal dom a

Clash.Prelude.register
    :: (HiddenClockResetEnable dom, NFDataX a)
    => a -> Signal dom a -> Signal dom a
```

The typeclass constraint `HiddenClockResetEnable dom` represents a clock domain that has an associated clock, reset and enable lines. We can go back and forth between the explicit and the implicit passing:

```
hideClockResetEnable
    :: (HiddenClockResetEnable dom)
    => (KnownDomain dom => Clock dom -> Reset dom -> Enable dom -> r)
    -> r

withClockResetEnable
    :: KnownDomain dom
    => Clock dom -> Reset dom -> Enable dom
    -> (HiddenClockResetEnable dom => r) -> r
```

Of course, this implicit typeclass-based clock passing still doesn't allow us to make something from nothing, so the top-level `topEntity` still has to get a clock signal (and, optionally, a reset signal) from somewhere outside.

In the rest of this book, we will take on the following conventions:

- Clock, reset, and enable lines will be passed implicitly, i.e. using `HiddenClockResetEnable` and using `Clash.Prelude` instead of `Clash.Explicit.Prelude`.
- Since we are not going to use `Enable` signals, we will use `enableGen` in every `topEntity` to enable all components at all times, and leave it to the individual circuits to implement conditional register updates using plain `Bool`-valued signals.
- We are only going to use an external `Reset` signal in designs that use a clock manager, since they usually have an output signal to mark that the clock is locked at the desired frequency and is ready to use. For circuits that use the raw clock, we'll just use `resetGen` to start the circuit immediately as soon as the FPGA is powered up.

- All functions will be domain-polymorphic, with sufficient constraints as needed. Of course, topEntity can't be domain-polymorphic since that is where the clock domain is specified; so all through this book, topEntity will be defined as withClockResetEnable clk resetGen enableGen board, i.e. as a domain-polymorphic, implicitly-clocked function with input signals applied.

Following is the full code listing of the LED blinker rewritten in the same style that we will use for the rest of this book. Note that this imports the definition of succIdx from RetroClash.Utils, and ClockDivider and Milliseconds from RetroClash.Clock: we will be implicitly collecting a library of reusable functions throughout the book.

```
{-# LANGUAGE NumericUnderscores, PartialTypeSignatures #-}
{-# OPTIONS_GHC -Wno-partial-type-signatures #-}
module Blink where

import Clash.Prelude
import RetroClash.Utils (succIdx)
import RetroClash.Clock (ClockDivider, Milliseconds)
import Data.Either
import Data.Maybe

-- Change this to the raw clock rate of the FPGA board you are targeting
createDomain vSystem{vName="Dom100", vPeriod = hzToPeriod 100_000_000}

data OnOff on off
    = On  (Index on)
    | Off (Index off)
    deriving (Generic, NFDataX)

isOn :: OnOff on off -> Bool
isOn On{}  = True
isOn Off{} = False

countOnOff :: (KnownNat on, KnownNat off) => OnOff on off -> OnOff on off
countOnOff (On  x) = maybe (Off 0) On  $ succIdx x
countOnOff (Off y) = maybe (On  0) Off $ succIdx y

topEntity
    :: :CLK100MHZ" ::: Clock Dom100
    -> "LED"       ::: Signal Dom100 Bit
topEntity clk =
    withClockResetEnable clk resetGen enableGen blinkingSecond
```

```
blinkingSecond
    :: forall dom. (HiddenClockResetEnable dom, _)
    => Signal dom Bit
blinkingSecond = boolToBit . isOn <$> r
  where
    r :: Signal dom
         (OnOff
             (ClockDivider dom (Milliseconds 500))
             (ClockDivider dom (Milliseconds 500)))
    r = register (Off 0) $ countOnOff <$> r
```

In fact, let's factor out this pattern so we can write `topEntity` point-free:

```
withResetEnableGen
    :: (KnownDomain dom)
    => (HiddenClockResetEnable dom => r)
    -> Clock dom -> r
withResetEnableGen board clk =
    withClockResetEnable clk resetGen enableGen board

topEntity
    :: Clock Dom100
    -> Signal Dom100 Bit
topEntity = withResetEnableGen blinkingSecond
```

Exercises:

- Blink only a set number of times
- Blink multiple LEDs at different speeds
- Synchronized LED blinking. Write fun patterns like a left-to-right then right-to-left sweep, or a sweep from both sides to the center and then outwards again.
- Sequence multiple LED blinking patterns, each one shown for a couple of seconds before switching to the next one.

4.5 Multiple clocks

The problems associated with interfacing components with different clocks is beyond the scope of this book. We will design circuits where everything runs at the same clock, using counter-based clock rate division where needed. For circuits that generate video output, the base clock rate will be chosen to match the pixel clock rate.

In the second half of the book, we will implement computers based on preexisting designs from the late 1970s. The original machines we will replicate had a multi-clock design by necessity: the video subsystem was required to be simple enough so that it can run at the speed needed to produce a valid video signal, but the CPU was too complex to run at that rate. On a modern FPGA, these simple CPUs can be easily run at the speed prescribed by the video system. By using a single clock domain for the whole computer, we will avoid all the headache associated with domain crossing; but this comes at the price of our computer not behaving exactly the same as the original ones: we will discuss the user-visible implications of this separately for each machine.

4.6 Pushbutton-toggled LED

While blinking is fun, its one drawback is that it doesn't have any inputs. So let's turn back to one of the motivating problems at the start of this chapter: toggling an LED on and off with a pushbutton switch. We can describe this easily using the RTL model: as a register containing the LED state, that at every cycle will either stays its current value or its complement. The trigger to flipping the register is the click of the switch, i.e. if the button is held in the given cycle but wasn't in the previous one.

```
topEntity
    :: "CLK" ::: Clock System
    -> "BTN" ::: Signal System (Active High)
    -> "LED" ::: Signal System (Active High)
topEntity = withResetEnableGen board
  where
    board btn = toActive <$> led
      where
        btn' = fromActive <$> btn
        click = btn' .&&. (not <$> register False btn')
        led = register False $ mux click (not <$> led) led
```

There are two patterns here that are generally useful and will occur in a lot of our other designs; both of these are provided by the Clash Prelude as library functions:

- In the definition of `led`, the pattern of "update a register only if a condition holds" is called `regEn` (for "*reg*ister with a separate update *en*able line), allowing us to write instead:

  ```
  led = regEn False click (not <$> led)
  ```

- In the definition of `click`, this is an instance of *(rising) edge detection*: `click`'s value is `True` iff the underlying `btn'` signal's value is rising, i.e. it is larger than its previous value. For `Bool`, we have `True > False`, and so a transition from `False` to `True` counts as a rising edge. The first argument, `False`, is what initial value to use; its only relevance is for the very first cycle. In this case, its initial value doesn't matter at all: we are processing human input, so no one will notice what happens with button clicks in the very first tens of nanoseconds after turning on the board.

```
click = isRising False btn'
```

We can also abstract a third pattern and add it as a library function: oscillating a `Bool` value on an external signal.

```
oscillateWhen
    :: (HiddenClockResetEnable dom)
    => Bool -> Signal dom Bool -> Signal dom Bool
oscillateWhen init trigger = r
  where
    r = regEn init trigger (not <$> r)
```

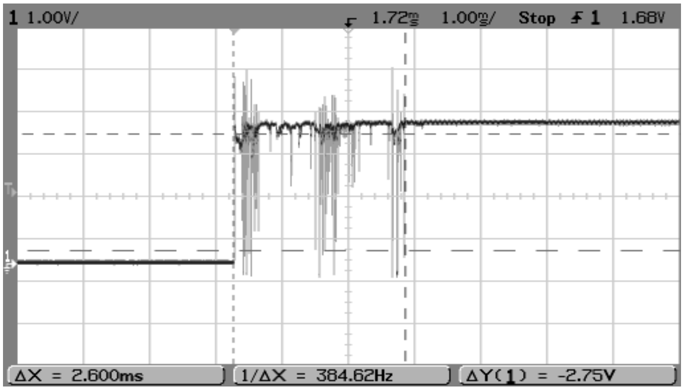
Together, these allow us to rewrite our `topEntity` into the following form:

```
topEntity
    :: "CLK" ::: Clock System
    -> "BTN" ::: Signal System (Active High)
    -> "LED" ::: Signal System (Active High)
topEntity = withResetEnableGen board
  where
    board btn = toActive <$> led
      where
        click = isRising False (fromActive <$> btn)
        led = oscillateWhen False click
```

4.6.1 Debouncing

When trying it out on real hardware, pay close attention to the LED changing as you press the button. Do you notice that sometimes a click doesn't seem to register?

When a real, physical switch is pressed, its output doesn't just go straight from one level to the other. Instead, once it is pushed, it will keep jumping between "open" and "close" for a while before becoming fully closed. This is called *bouncing* and its effect can occur both when pressing and when releasing, but is usually more pronounced when pressing.



The details of how a given switch bounces depends on its physical characteristics, but in general it is going to be a short burst of high-frequency noise starting as the switch is pressed. Since this signal looks very different from the user's intention (a single transition from open to closed), and it is not at all apparent when operating a pushbutton switch that this is happening, devices should take care to remove this effect. In our case, we want the "user-intended" button event to toggle the LED, not these "artificial" events.

Since the bounce noise is very different from the intended signal, we should be able to remove it and get a nice, clean signal instead, with a single sharp edge around presses and releases. There are hardware- and software-based solutions for this. Here, we are going to create a software debouncer in Clash: it will take the `Signal dom Bit` as it comes out of the physical pushbutton, and produce an output `Signal dom Bit` without these high-frequency jumps.

It is not at all impossible that your development board already has hardware debouncing between the pushbutton switches and the FPGA (see the vendor's documentation), and in general, that is a good, useful thing to have. However, it robs us of this opportunity to show a debouncer's effect in the most easily observable way. Debouncing is idempotent[1] so adding the below software debouncer to a circuit running on a board with hardware-debounced buttons will still work correctly.

The idea behind software debouncing is quite simple: we keep track of the input history for a set number of cycles, and let changes through only once the value has stabilized for all of them. Of course, in the real implementation, we don't need to keep around n previous values to compare them with the current input, it is sufficient to keep a single "last seen" value and a counter up to n of cycles since the change.

[1] Modulo lag. At the timescales involved in humans pressing buttons and looking at LEDs, microseconds of lag is not going to be observable.

Because this is the most complicated RTL code we've seen so far, let's go through it in detail. First of all, we will need a way to keep incrementing the "stability counter" capped at maxBound, i.e. we want to compute the *saturating successor*. We can do that easily using succIdx, by mapping the Nothing returned by succIdx maxBound back to maxBound. We'll write the corresponding *saturating predecessor* as well:

```
moreIdx :: (Eq a, Enum a, Bounded a) => a -> a
moreIdx = fromMaybe maxBound . succIdx

lessIdx :: (Eq a, Enum a, Bounded a) => a -> a
lessIdx = fromMaybe minBound . predIdx
```

Now for the main event, starting with the intended type of our debouncing function:

```
debounce
    :: forall ps a dom. (Eq a)
    => SNat ps -> a -> Signal dom a -> Signal dom a
```

As we will see, this is a bit optimistic: the implementation will impose more typeclass constraints; but only things like NFDataX a or KnownNat (ClockDivider dom ps) which are going to be satisfied in any reasonable use case anyway.

The result of debounce ps start this should be the last stable value we've seen of this, i.e. the last value that hasn't changed for ps picoseconds:

```
debounce _ start this = lastStableValue
  where -- Continued below
```

We will store lastStableValue in a register that is only updated whenever the input has stabilized. We will say that a signal is stable if it hasn't changed for the last n cycles; in other words, if it has been true for n cycles that the current value in this is equal to the previous one. If the current and the previous values are not equal, we start looking for a new run of n equal values by resetting the counter to 0:

```
    prev = register x0 this
    counter = register (0 :: Index (ClockDivider dom ps)) $
        mux (this .==. prev) (moreIdx <$> counter) 0
    stable = counter .== maxBound
```

Here, the operator (.==.) is simply an Applicative-lifted version of (==), i.e. it is defined as liftA2 (==) in the Clash prelude. We will also use (==.) and (.==) where the argument on the left- or the right-hand side is a pure value, respectively.

Now we have everything to define `lastStableValue`. It is clear that it should be of the form `regEn start stable _`, but there is one subtlety remaining: when the signal is `stable`, should we update `lastStableValue` from `this`, or from `prev`?

As we have seen in RTL schematics before, the value feeding into combinational circuits from a `register` is the value that is being read, i.e. the value that was written at the end of the previous clock cycle. Since `stable` is defined in terms of `counter`, which is a `register`, this means when a sudden change occurs in the input after a long enough run of stable values, `stable` will still be `True`, since it is seeing `counter` not at value `0` yet, but at `maxBound`. Clearly, this means updating `lastStableValue` from `this` at that point would be incorrect, and we should be using the `prev`ious value, i.e. the input that was used to compute the value of `counter`, and thus, `stable`:

```
    lastStableValue = regEn x0 stable prev
```

Alternatively, we can shave off this extra cycle of lag by using `this` in the definition of `lastStableValue`, if we use the just-now-computed next value of `counter`, bypassing the `register`:

```
debounce _ start this = regEn start stable this
  where
    counter = register (0 :: Index (ClockDivider dom ps)) counterNext
    counterNext = mux (this .==. register start this)
        (moreIdx <$> counter)
        0
    stable = counterNext .== maxBound
```

Let's factor out the "has this signal's value changed" bit into a utility function:

```
changed
    :: (HiddenClockResetEnable dom, Eq a, NFDataX a)
    => a -> Signal dom a -> Signal dom Bool
changed x0 x = x ./=. register x0 x
```

Putting it all together, this gives us the following final versions of `debounce` and our button-toggling circuit, including the full type of `debounce` with all the extra constraints imposed by the use of `register` and `ClockDivider`. The debouncing period of 5 ms was chosen mostly unscientifically, as something in the right order of magnitude for pushbutton switches, but still fast enough to not be a problem with interactive applications such as a video game running at 60 fps, requiring frame-perfect input. Of course, readers should feel free to experiment with different values for a given pushbutton.

```
debounce
    :: forall ps a dom. (Eq a, NFDataX a)
    => (HiddenClockResetEnable dom, KnownNat (ClockDivider dom ps))
    => SNat ps
    -> a -> Signal dom a -> Signal dom a
debounce _ start this = regEn start stable this
  where
    counter = register (0 :: Index (ClockDivider dom ps)) counterNext
    counterNext = mux (changed start this) 0 (moreIdx <$> counter)
    stable = counterNext .== maxBound

topEntity
    :: Clock System
    -> Signal System (Active High)
    -> Signal System (Active High)
topEntity = withResetEnableGen board
  where
    board btn = toActive <$> led
      where
        btn' = debounce (SNat @(Milliseconds 5)) False $
            fromActive <$> btn
        click = isRising False btn'
        led = regEn False click (not <$> led)
```

4.7 Summary

- A shared **clock** provides synchronicity, which in turn allows abstracting away the propagation delay
- The **RTL model** describes stateful circuits as registers, combinational circuits between them, and synchronous feedback through the registers' write inputs
- Clash's **`register`** primitive enables RTL circuit design, with feedback modeled as **`register`-guarded recursion**
- The **`HiddenClockResetEnable`** typeclass clears up the clutter of getting the clock, reset, and enable lines to every `register`
- Once timekeeping enters the picture, we need to make sure **Clash is configured to use the right clock settings** for period calculations
- Clash provides a datatype **`Index n`** with exactly n values, which is useful for counting or indexing. n doesn't need to be a power of 2.

- Just because inputs are digital, doesn't mean they can't have noise: **debouncing** can remove high-frequency components from a signal that we expect to only have low-frequency changes

5 Time-domain Multiplexing

Time is what keeps everything from happening all at once.
— Ray Cummings

Multiplexing is combining multiple signals into one, by dividing the single channel somehow. For example, multiple radio stations can broadcast in the same area at the same time because they are multiplexed in the frequency domain: every radio channel uses a different carrier frequency (sufficiently apart from each other). Radio receivers then can either tune to a specific frequency (which is how "normal" radio receivers work), or decompose the mixed signal in the frequency domain and listen to all radio stations at the same time[1].

In this chapter, we will look into situations where the division is in the time domain: the same signal lines will be used for one purpose for a short while, then for a slightly different purpose for a bit, before moving on to the next one, and so on.

5.1 Does this have anything to do with `mux`?

In the previous chapter, we introduced a combinator for lifting branching into any applicative functor. We named this combinator `mux` (short for *multiplexer*); this is also what it is called in the Clash Prelude:

```
mux :: (Applicative f) => f Bool -> f a -> f a -> f a
```

A natural question at this point is what is the relationship between this `mux` signal combinator and signal multiplexing as discussed in this chapter.

It turns out the two concepts are very tightly related. `mux @(Signal dom)` implements a multiplexer since its result is a single `Signal dom a` that will take its value from one of the two connected inputs. The details of this multiplexing is

[1] see `http://cowlet.org/2014/05/05/listening-to-200-radio-stations-at-once.html` for a detailed explanation of this latter technique

determined by the driver of the `Signal dom Bool` line. In the previous chapter, this selector line was connected to state- and input-dependent logic; but if we connect a simple counter to it, we instantly get a 2-way time-domain multiplexer.

5.2 Seven-segment displays, revisited

Recall that a seven-segment display with n digits has the following $7 + n$ pins:

- 7 segment cathodes/anodes (optionally with 1 extra for the decimal point)
- n common anode/cathode digit selectors

If we want to show different segment patterns for each digit, we seemingly have an impossible task ahead: there are 2^{7n} possible pattern combinations, but we only have $7 + n$ two-state lines, which can't represent more than 2^{7+n} different values.

To the surprise of no one who has read this chapter's title, the solution is time-domain multiplexing. At any given time, we will use the digit selectors to drive a single digit, and set the segment lines to whatever pattern we want to show on that digit. Then, after a while, the selectors and the segments are changed in tandem to the next digit.

The one problem with this scheme is that instead of showing all digits at the same time, the displays will show each digit one by one, jumping around. Or are they? To convince us that we can use this idea for nice, stable display of multiple digits, let's first write a very simple circuit that will keep flashing the seven-segment digits at different speeds.

The idea is to start with a fast clock divider running at 512 Hz, then make a slower clock by dividing further by counting in 8 bits up to a user-supplied number (set with 8 switches). So by setting the counting target to e.g. 0, we get 512 Hz, but setting the target to 3 will give us a division by 4 for 128 Hz. With a target of 255, we get 2 Hz, i.e. each digit flashing for half a second which can easily be seen by the human eye.

We start with a `Bool`-valued signal that goes to `True` with the given period (in picoseconds) or the given rate (in Hertz). This is implemented using Clash's `riseEvery` function, converting the period length to number of cycles:

```
type ClockDivider dom n = n `Div` DomainPeriod dom

risePeriod
    :: forall ps dom. (HiddenClockResetEnable dom, _)
    => SNat ps
    -> Signal dom Bool
risePeriod _ = riseEvery (SNat @(ClockDivider dom ps))
```

```
riseRate
    :: forall rate dom. (HiddenClockResetEnable dom, _)
    => SNat rate
    -> Signal dom Bool
riseRate _ = risePeriod (SNat @(HzToPeriod rate))
```

What remains is to take `i :: Index n` that is the number of the currently selected digit, and turn it into a `Vec n Bool` where only the `i`-th bit is set. This is called *one-hot encoding*, and we do it via `Unsigned n`'s `Bits` instance. Note the use of `reverse` in the definition of `oneHot`, corresponding to the indexing difference between vectors and `Unsigned` numbers.

```
oneHot :: forall n. (KnownNat n) => Index n -> Vec n Bool
oneHot = reverse . bitCoerce . bit @(Unsigned n) . fromIntegral

topEntity
    :: Clock System
    -> Signal System (Vec 8 Bit)
    -> ( Signal System (Vec 4 (Active High))
       , Signal System (Vec 7 (Active Low))
       , Signal System (Active Low)
       )
topEntity = withResetEnableGen board
  where
    board switches =
        ( map toActive <$> anodes
        , map toActive <$> segments
        , toActive <$> dp
        )
      where
        segments = pure $ repeat True
        dp = pure False

        fast = riseRate (SNat @512)

        slow = fast .&&. cnt .==. 0
          where
            speed = bitCoerce <$> switches
            cnt = regEn (0 :: Unsigned 8) fast $
                mux (cnt .>=. speed) 0 (cnt + 1)

        i = regEn 0 slow (nextIdx <$> i)
        anodes = oneHot <$> i
```

As we decrease the counter target, first we see the flashing become faster but then something interesting happens. The actual threshold will depend on individual visual sensitivity, but at least for this author, with a divider of 4, i.e. at 128 Hz, the flashing is still visible (and, in fact, quite distracting), but with a divider of 3 (i.e. by setting the counter target to 2 with the switches), at 170 Hz, the blinking goes away. This is because *persistence of vision* takes over: the flashing gets too fast to see, and the human visual processing system regards it as a continuous image. However, 170 Hz is still not necessarily fast enough: while it might look stable when looked at head-on, the flickering is still noticeable in peripheral vision.

5.2.1 Showing different digits

Now that we have a way of turning on just one digit at a time, by changing the `segments` shown, we can finally display multiple, different digits. As a first version, let's just change our previous circuit slightly:

```
digits = map encodeHexSS (0x1 :> 0x2 :> 0x3 :> 0x4 :> Nil)
segments = (digits !!) <$> i
```

With this small change, the first digit will display `1`, the second `2` and so on. By setting the switches to a counter target of just 3 or less, the blinking and flickering goes away and the display will show `1234` as intended.

Before we move on, let's factor out some reusable components for applicative vector indexing (similar to the various versions of `.==.`), round-robin counting and -multiplexing:

```
(.!!.)
    :: (KnownNat n, Enum i, Applicative f) => f (Vec n a) -> f i -> f a
(.!!.) = liftA2 (!!)

roundRobin
    :: forall n dom a. (KnownNat n, HiddenClockResetEnable dom)
    => Signal dom Bool
    -> (Signal dom (Vec n Bool), Signal dom (Index n))
roundRobin next = (selector, i)
  where
    i = regEn (0 :: Index n) next $ nextIdx <$> i
    selector = bitCoerce . oneHot <$> i
```

```
muxRR
    :: (KnownNat n, HiddenClockResetEnable dom)
    => Signal dom Bool
    -> Signal dom (Vec n a)
    -> (Signal dom (Vec n Bool), Signal dom a)
muxRR tick xs = (selector, current)
  where
    (selector, i) = roundRobin tick
    current = xs .!!. i
```

Armed with these, and fixing the refresh rate to 512 Hz, we can now use 8 input switches to show one full byte as two hexadecimal digits. On an FPGA board with four seven-segment digits, this leaves us with two more digits; let's just leave them empty for now, by lifting `encodeHexSS` over `Maybe`, mapping `Nothing` to no segments.

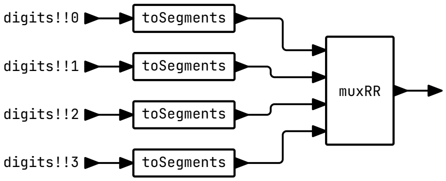
```
board switches = (map toActive <$> anodes, map toActive <$> segments,
    toActive <$> dp)
  where
    digits = (repeat Nothing ++) <$> (map Just . bitCoerce <$> switches)
    toSegments = maybe (repeat False) encodeHexSS

    (anodes, segments) = muxRR (riseRate (SNat @512)) $
        map toSegments <$> digits
    dp = pure False
```

5.2.2 Multiplexing and encoding

There are two ways of combining the hexadecimal encoder `toSegments` and the round-robin multiplexer `muxRR`:

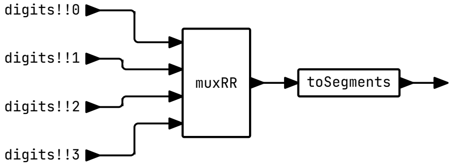
1. In our latest version of `board`, the encoding is done before `muxRR`:

```
(anodes, segments) = muxRR _ (map toSegments <$> digits)
```



2. But there is another way, which is to multiplex the “model”, i.e. the `Maybe (Unsigned 4)` to show, and connect the encoder to the half-byte that is to be shown at the moment:

```
(anodes, digit) = muxRR (riseRate (SNat @512)) digits
segments = toSegments <$> digit
```



These two circuits differ in one important detail: for n half-byte digits, the first one uses n hexadecimal encoders, whereas the second one only uses one: the encoder itself has now become a multiplexed resource. And so, because the second approach leads to a more optimal circuit in terms of FPGA parts usage, we will turn this latter version of the multiplexing seven-segment driver into a reusable component as a higher-order function:

```
data SevenSegment n anodes segments dp = SevenSegment
    { anodes   :: "AN"  ::: Vec n (Active anodes)
    , segments :: "SEG" ::: Vec 7 (Active segments)
    , dp       :: "DP"  ::: Active dp
    }

driveSS
    :: (KnownNat n, HiddenClockResetEnable dom, _)
    => (a -> (Vec 7 Bool, Bool))
    -> Signal dom (Vec n (Maybe a))
    -> Signal dom (SevenSegment n anodes segments dp)
driveSS draw digits = do
    anodes <- map toActive <$> anodes
    segments <- map toActive <$> segments
    dp <- toActive <$> dp
    pure SevenSegment{..}
  where
    (anodes, digit) = muxRR (risePeriod (SNat @(Milliseconds 1))) digits
    (segments, dp) = unbundle $
        maybe (repeat False, False) draw <$> digit
```

Armed with these definitions, we can rewrite our circuit succinctly as:

```
topEntity
    :: "CLK100MHZ" ::: Clock System
    -> "SW"        ::: Signal System (Vec 8 Bit)
    -> "SS"        ::: Signal System (SevenSegment 4 High Low Low)
topEntity = withResetEnableGen board
  where
    board switches = driveSS toSegments digits
      where
        digits = (repeat Nothing ++) . map Just . bitCoerce <$> switches
        toSegments x = (encodeHexSS x, False)
```

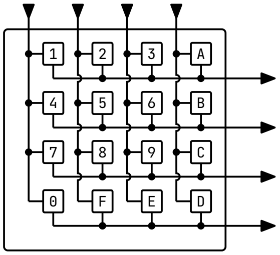
Exercises:

- Given the 8-bit input from the switches, display them on three 7-segment displays in decimal.
- Omit leading zeroes in the decimal version.
- At the press of a pushbutton, start and display a countdown (in seconds).
- Digital stopwatch: one pushbutton to start/stop, one to reset to 0. For extra niceness, compute minutes from the seconds and flash the decimal point between the minutes and the seconds at a half-second interval.

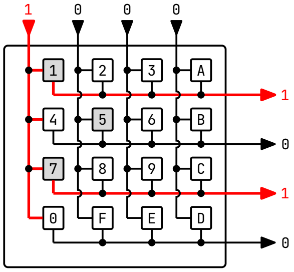
5.3 Keyboard matrix sweeping

Seven-segment display are, of course, output peripherals. In this section, we are going to show an example of using time-domain multiplexing to drive an input peripheral: a 4x4 keypad.

Keypads, and keyboards in general, have the same problem as 7-segment displays when it comes to number of signal lines. A keyboard is simply a collection of pushbuttons, arranged in some user-friendly layout and with keycaps on top. Thus, a naïve design for an $n \times m$-key keyboard would require $n \cdot m$ separate one-bit input signal lines. However, if we arrange the buttons in an $n \times m$ matrix, we can get by with just $n + m$ lines, with either n or m of them being output lines. For example, here is the layout of the keyboard matrix for the 4×4 keypad we are going to be using in this book:



Similarly to the multi-digit seven-segment display, one set of outputs function as the *column selectors*: by setting some `high` and the others `low`, we know that `high` signals on the *row inputs* must come from those selected columns. For example, if we select only the first column (by outputting `<1, 0, 0, 0>`), and observe `<1, 0, 1, 0>` on the rows, we can deduct that the keys `1` and `7` are currently pressed. If `5` is also pressed, it will only be detected when the second column is selected.



It is important to note that while the behavioral diagram above makes it look as if the role of the rows and the columns could be reversed, a real, physical keyboard matrix requires more electrical components to safeguard against accidentally shorting the selector pins, and that can impose directionality. When using a keypad, always check its datasheet to see which axis should be used as a selector, i.e. as outputs from the point of view of the FPGA. In this book, all code is written for keypads that use the **columns** as **active-low** selectors; adapt to your hardware as needed.

5.3.1 Getting the hang of it

Before starting to design anything more complicated, it is perhaps useful to write a simple circuit that drives and displays a keypad, allowing us to experiment with it interactively. To that end, let's hook up some switches and LEDs directly to the 4 columns and 4 rows of our keypad. It is a one-liner circuit but we'll show it here with its full port name annotations to make it clear what gets connected to what:

```
topEntity
    :: "ROWS"     ::: Signal System (Vec 4 Bit)
    -> "SWITCHES" ::: Signal System (Vec 4 Bit)
    -> ( "LEDS"   ::: Signal System (Vec 4 Bit)
       , "COLS"   ::: Signal System (Vec 4 Bit)
       )
topEntity rows switches = (rows, switches)
```

To play around with this toy, connect a 4 × 4 keypad and look at the LEDs while holding down keys on it. By selecting one column at a time with the switches, we can see that the LED state will depend only on keys in that column.

To scan the full keypad, we just need to change the column selector periodically, and record the value read from the rows into a separate register for each column. We can use the round-robin multiplexer function to generate the selectors, and every column's state register only updates itself if it is equal to the currently selected column. We don't really need to sweat the switching rate; in fact, we get away with using a hard-coded rate of 1000 cycles, which is under-specified: its real-time length will depend on the clock rate used.

```
type Matrix rows cols a = Vec rows (Vec cols a)
type KeyStates rows cols = Matrix rows cols Bool

scanKeypad
    :: (KnownNat rows, KnownNat cols, IsActive rowAct, IsActive colAct)
    => (HiddenClockResetEnable dom)
    => Signal dom (Vec rows (Active rowAct))
    -> ( Signal dom (Vec cols (Active colAct))
       , Signal dom (KeyStates rows cols)
       )
scanKeypad rows = (map toActive <$> cols, transpose <$> bundle states)
  where
    (cols, currentCol) = roundRobin nextCol
    nextCol = riseEvery (SNat @1_000)

    states = map colState indicesI

    colState thisCol = regEn (repeat False) (currentCol .== thisCol) $
        map fromActive <$> rows
```

To produce some observable output, we can connect the keypad state straight to some LEDs:

```
topEntity
    :: Clock System
    -> Signal System (Vec 4 (Active Low))
    -> ( Signal System (Vec 16 (Active Low))
      , Signal System (Vec 4 (Active Low))
      )
topEntity = withResetEnableGen board
  where
    board rows = (map toActive <$> leds, cols)
      where
        (cols, keyStates) = scanKeypad rows
        leds = bitCoerce <$> keyStates
```

5.3.2 Debouncing

Let's say we now want to implement the keypad analogue of the LED toggle push-buttons: we want to control the 16 LEDs with the 16 keys, such that each key turns a given LED on or off.

Implementing this without debouncing is not particularly difficult. We'll do it in two steps: first we convert the state matrix into an event matrix, comparing the `current` state to its `delayed` copy, and then we toggle registers (one for each key) on `Pressed` events. The reason for splitting it into two is so we can reuse the first half in later designs.

```
data KeyEvent = Pressed | Released
    deriving (Show, Eq, Generic, NFDataX)

type KeyEvents rows cols = Matrix rows cols (Maybe KeyEvent)

keypadEvents
    :: (KnownNat rows, KnownNat cols, HiddenClockResetEnable dom)
    => Signal dom (KeyStates rows cols)
    -> Signal dom (KeyEvents rows cols)
keypadEvents current = zipWith (zipWith event) <$> delayed <*> current
  where
    delayed = register (repeat $ repeat False) current

    event False True = Just Pressed
    event True False = Just Released
    event _ _ = Nothing
```

```
toggleKeypad
    :: (KnownNat rows, KnownNat cols, HiddenClockResetEnable dom)
    => Signal dom (KeyEvents rows cols)
    -> Signal dom (KeyStates rows cols)
toggleKeypad events = toggles
  where
    clicks = map (map (== Just Pressed)) <$> events
    toggles =
        bundle . map (bundle . map toggleState . unbundle) . unbundle $
        clicks
      where
        toggleState click = let r = regEn False click (not <$> r) in r
```

If we try this out (by setting `leds = bitCoerce <$> toggleKeypad keyState`), we will notice that it bounces, as it can be expected: every key in a keypad is a pushbutton switch on its own, so of course they will behave the same way as standalone buttons.

To fix that, we can't just apply debouncing to the input signal `rows` directly: when we change the column selector `cols`, the state read from `rows` will change very quickly, so a debouncer would lose those changes and every row would read the same. Instead, we should debounce the full keypad state, i.e. store all 16 bits of key state as read, and only update it when all 16 values have been stable for long enough. Also, this author's experiments have shown some noise just after switching columns, so we will ignore `rows` values for 10 cycles: we change `scanKeypad` to update the `colState` registers only when the input is presumed to be `stable`:

```
colState thisCol = regEn (repeat False) (stable .&&. currentCol .==
    thisCol) $ rows
  where
    stable = cnt .== maxBound
    cnt = register (0 :: Index 10) $ mux nextCol 0 (moreIdx <$> cnt)
```

The only remaining change is to debounce the `keyStates` before turning it into events, in the main `board` definition:

```
board rows = (map toActive <$> leds, cols)
  where
    (cols, keyStates) = scanKeypad rows
    keyStates' = debounce (SNat @(Milliseconds 5))
        (repeat $ repeat False)
        keyStates
    ledStates = toggleKeypad . keypadEvents $ keyStates'
    leds = bitCoerce <$> ledStates
```

5.4 Showing keypad input on a seven-segment output

The two peripherals discussed in this chapter are such a good match-up that it would be criminal not to connect them, by making a small device which accepts keypad input and displays the last *n* hexadecimal inputs on an *n*-digit seven-segment display. We will also collect some code along the way that will be very useful for projects in later chapters.

The basic structure of our circuit is captured in the `topEntity` definition. It has ports connecting both to the keypad and to the seven-segment display, and shows how we decompose the design into three parts: `input`, `logic` and `display`.

```
topEntity
    :: "CLK"    ::: Clock System
    -> "ROWS"   ::: Signal System (Vec 4 (Active Low))
    -> ( "SS"   ::: SevenSegment System 4 High Low Low
       , "COLS" ::: Signal System (Vec 4 (Active Low))
       )
topEntity = withResetEnableGen board
  where
    board rows = (display digits, cols)
      where
        (cols, key) = input rows
        digits = logic key
```

5.4.1 Logic

Let's start by implementing the basic logic of our circuit: maintaining a state of user-entered digits. Here, we will describe everything in domain-appropriate types, and leave the I/O conversion to other parts of our circuit. In this case, this means we will assume the input is already a stream of decoded key presses as a `Maybe Hex` where `Nothing` means no new key has just been pressed, and the `Hex` corresponds to the key value, and the output is likewise a `Vec n (Maybe Hex)` with `Nothing` standing for digits not shown.

```
type Hex = Unsigned 4

logic
    :: forall n dom. (KnownNat n, HiddenClockResetEnable dom, _)
    => Signal dom (Maybe Hex) -> Signal dom (Vec n (Maybe Hex))
logic key = digits
  where -- Continued below
```

We store the current digits in a register, and update it on key presses. This conditional update can be written very conveniently with Clash's `regMaybe` combinator, which works a lot like `regEn` but combines the update-enable signal with the new-value signal. The unpacking of the `Maybe` (in the binding of `newDigit`) followed by a subsequent re-packing might seem redundant in `update`, but we only ever want to push a new digit onto the state if there is one available; otherwise, the state would be quickly flushed with all `Nothing`s whenever no keys are pressed.

```
    digits = regMaybe (repeat Nothing) $ update <$> key <*> digits

    update key digits = do
        newDigit <- key
        return $ digits <<+ Just newDigit
```

We are using Clash's `<<+` operator to shift in a new element onto a vector, from the right. This corresponds to our intuition that we want to put new digits to the right-hand side of the display, the same way as a calculator works. We use that orientation here because this way the code matches visually the behavior; but this means (because vectors are indexed from the left) that we need to reorder the displayed digits in `display` below.

5.4.2 Output

The `display` driver is very straightforward because our generic seven-segment function `driveSS` already does the heavy lifting. We simply take `encodeHexSS` and pair it up with an unset decimal point.

```
display
    :: (KnownNat n, HiddenClockResetEnable dom, _)
    => Signal dom (Vec n (Maybe Hex)) -> Signal dom (SevenSegment n _ _
    _)
display = driveSS (\x -> (encodeHexSS x, False))
```

5.4.3 Input

The input to our `logic` circuit is the value of the latest key pressed, so we will need a way to decode the keypad state into that. We have already done half of this in `keypadEvents`; what remains is to map the matrix of `KeyEvent`s onto a single one with the right key value. This is necessarily lossy, as in not all information is kept: if two keys are pressed at the same time, we are going to decode that into just one key press. In other words, one of the two will "win out" over the other one. So let's just take the first `Just` row-by-row, left to right, by folding `mplus` over the `Maybe` key events, at both levels of our `Vec`-in-a-`Vec` nesting. We use Clash's vector-specific `fold` function which exploits associativity to build a shallow tree of multiplexers instead of the deep sequence we would get from `foldr` or `foldl`. Note that `fold` requires a non-empty input vector, hence the `+ 1` in the type:

```
firstJust2D
    :: (KnownNat rows, KnownNat cols)
    => Matrix (rows + 1) (cols + 1) (Maybe a)
    -> Maybe a
firstJust2D = fold mplus . map (fold mplus)
```

Note that it isn't enough to just apply `firstJust2D` on the `KeypadEvents` directly: that would give us a `Maybe KeyEvent`. With that, all we would find out is whether *any* key was pressed or released in the given clock cycle. To find out *which* key it is, we need to apply a keymap, by replacing each `Just Pressed` value with `Just mapping` (and everything else with `Nothing`). The keymap is needed to connect the 4 × 4 layout of the keypad with the intended values of the keys. Hexadecimal keypads have no standard layout, so this needs to be tweaked based on the actual hardware. In this example, we are going to be targeting a keypad with the layout shown earlier.

We can store that layout in a 4×4 matrix, and then `zip` it with the `KeyEvents` to decorate each `KeyEvent` with its value. What's particularly nice about this is that the definition of the `keymap` is visually identical to the intended layout, so it's easy to adapt to other layouts. Again, similar to `firstJust`, the two-level `zipWith (zipWith decode)` structure comes simply from the nested vectors.

```
pressedKeys
    :: Matrix rows cols a
    -> KeyEvents rows cols
    -> Matrix rows cols (Maybe a)
pressedKeys = zipWith (zipWith decode)
  where
    decode mapping (Just Pressed) = Just mapping
    decode _ _ = Nothing

keymap :: Matrix 4 4 Hex
keymap =
    ( 1  :>  2  :>  3  :> 0xa :> Nil) :>
    ( 4  :>  5  :>  6  :> 0xb :> Nil) :>
    ( 7  :>  8  :>  9  :> 0xc :> Nil) :>
    ( 0  :> 0xf :> 0xe :> 0xd :> Nil) :>
    Nil
```

We now have all the parts needed to implement our `input` subsystem. It has two kinds of outputs: the first one is to be connected to the outside world, and is required to drive the peripheral, and the second one is the result of processing the input signal as it arrives from the peripheral into the representation that the `logic` subsystem understands. There's nothing deep going on here, but this structure is nevertheless worth pointing out because it will be a recurring pattern in various peripheral drives throughout this book.

```
input
    :: (HiddenClockResetEnable dom, _)
    => Signal dom (Vec 4 (Active row))
    -> ( Signal dom (Vec 4 (Active col))
       , Signal dom (Maybe Hex)
       )
input = inputKeypad keymap

inputKeypad
    :: (KnownNat rows, KnownNat cols, IsActive rowAct, IsActive colAct)
    => (HiddenClockResetEnable dom)
    => (KnownNat (ClockDivider dom (Milliseconds 5)))
    => Matrix (rows + 1) (cols + 1) a
    -> Signal dom (Vec (rows + 1) (Active rowAct))
    -> ( Signal dom (Vec (cols + 1) (Active colAct))
       , Signal dom (Maybe a)
       )
inputKeypad keymap rows = (cols, pressedKey)
  where
    (cols, keyState) = scanKeypad rows
    events = keypadEvents . debounce (SNat @(Milliseconds 5))
        (repeat . repeat $ False)
        keyState
    pressedKey = firstJust2D . pressedKeys keymap <$> events
```

Exercises:

- Reject non-decimal input, i.e. only accept digits `0` to `9`.

- Map some pushbuttons to memory cells in the following sense: pressing the *i*-th pushbutton once should change the display to show the *i*-th register's value. Entering a new value with the keypad, and pressing the *i*-th pushbutton again should write the new value to the register. Pressing any other pushbutton should change the display yet again, and start editing the newly selected register.

- Change the `B` key's effect to act as a **B**ackspace by deleting the rightmost digit.

5.5 Summary

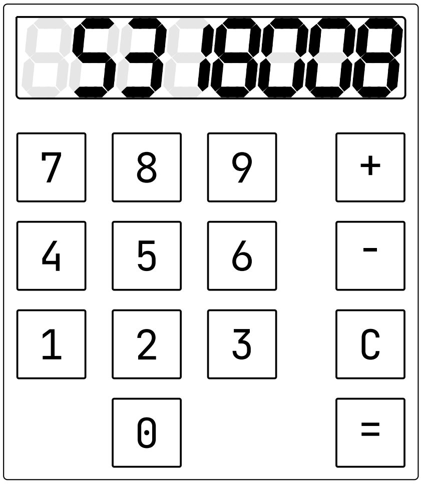
- **Multiplexing** allows using a single resource for multiple purposes. **Time-domain multiplexing** is when the various uses occur one after the other.
- For seven-segment displays and matrix keypads, we can achieve this by **periodically changing the selector** by setting it to the **one-hot encoded** index of the currently selected component (digit/column)
- **Encoding after multiplexing** allows the encoder itself to be a multiplexed, and thus shared, component.
- Transforming all events happening in a single cycle to a **single event simplifies downstream processing** at the cost of **losing simultaneous events**.
- We will design our circuits by **factoring out the logic** parts from the **peripheral drivers**, with the latter taking care of interpreting the signals from the outside world into the types of the domain where the logic lives.

6 Project: Pocket Calculator

In the *Project* chapters, we will build some fun devices out of parts we have already developed. Here we're building a simplified, but working pocket calculator that uses a multi-digit seven-segment display as its output, and a hexadecimal keypad as its input both for the numbers and the operators

6.1 A Minimal Viable Calculator

Everyone knows what a calculator looks like. You have some keys to enter numbers and commands such as + (for "add next number") or = (for "show result"). Results and the currently entered number are shown on a display.



On the most common infix calculators, an arithmetic expression such as 93 + 145 – 53 is entered in the following steps:

1. The first argument, 93 is keyed in (decimal) digit by digit, from left to right, i.e. from most significant to least significant.

2. The operator `+` is chosen by pressing the corresponding key. This moves the currently entered number 93 into the register holding the running value so far and erases the input buffer.
3. The second argument 145 is keyed in.
4. `-` is pressed to choose the next operator. This applies the previously chosen operator (`+` in this case) on the running value and the current input, and erases the input buffer.
5. 53 is keyed in.
6. By pressing `=`, the last selected operator `-` is applied on the running value and current input, and the result is displayed on the screen.

The full sequence of inputs and outputs is as follows (with a typo in step 6, fixed in steps 7 and 8, thrown in for good measure):

Step	Input	Display
0.		0
1.	9	9
2.	3	93
3.	+	93
4.	1	1
5.	4	14
6.	6	146
7.	Backspace	14
8.	5	145
9.	-	238
10.	5	5
11.	3	53
12.	=	185

In our implementation, we are going to make the following choices:

- Output is on a multi-digit seven-segment display, using time-domain multiplexing
- Input is a 4x4 hexadecimal keypad, with the non-decimal digits mapped to the following functions:
  - `A`: **Add**: the next number entered is going to be added to the running value

  - `B`: **Backspace**: removes the right-most digit from the number being entered
  - `C`: **Clear**: resets both the running value and the current number to 0
  - `D`: **Deduct**: the next number entered is going to be subtracted from the running value
  - `E`: **Equals**: shows the running value
  - `F`: **Fails to do anything**

- To keep the scope simple, we are only going to support addition and subtraction. This matches the capabilities of (Turing-complete) CPUs we will implement in later chapters. This also means we don't have to worry about operator precedence.[1]

- All computations will be done in $\mathbb{Z}/10^n\mathbb{Z}$: using n-digit unsigned integers with over/underflow. We will choose n to match the number of seven-segment digits.

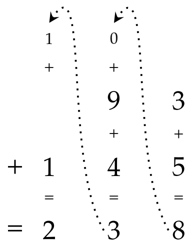
The cornerstone of our implementation is going to be this last point. By doing everything in decimal digits, we will avoid having to convert from/to hexadecimal at the peripherals, at the cost of complicating the arithmetic computations somewhat.

6.2 Binary Coded Decimal arithmetic

The representation we are going to use is called *binary coded decimal*, or *BCD* for short. In this representation, four bits are used to represent one decimal digit, with values greater than 9 considered invalid. The trick is to only keep this invariant once a computation is finished; but internally, digits will temporarily store values greater than 9. This allows us to re-use binary 4-bit arithmetic, which can be very efficiently mapped to FPGA elements, as the basic building block of our arithmetic unit. On the other hand, this representation can easily be connected to peripherals since its unit of display/input is 4 bits, with editing operations such as changing a digit easily expressible as manipulations of vectors of 4-bit elements.

To see how BCD arithmetic works, let's review how two decimal numbers can be added with pen & paper. Let's use the previous example of 93 + 145.

[1] It should be noted that the most common calculators disregard operator precedence anyway, and compute everything as if it was in fully-left-associated parenthesization, unless it's a so-called "scientific" or "engineering" calculator.



We start with the least significant digits, since there we know there is no carry yet.

1. By 3 + 5 = 8, we see that the least significant digit of the result is going to be 8, with no carry.

2. At the next digit, it is 9 + 4 = 13, giving us 3 as the result and, since 13 doesn't fit into a single digit, there is carry.

3. Then we run out of digits in 93, but not in 145 so we pad it to 093, which gives us 0 + 1 + 1 = 2 for the next digit (the extra +1 is for the carry).

4. Now we've run out of digits from both operands, and there is no carry, so we are done: the result is 238.

Now let's change that in two ways:

- The inputs and the output are in a fixed number of digits.
- Numbers are stored in BCD, and all arithmetic is done in base-16.

So now we are trying to compute $0093_{16} \oplus 0145_{16}$. Note that we are using BCD addition $\oplus$, and not normal addition: using normal addition, we would get $0093_{16} + 0145_{16} = 01D8_{16}$ instead of the intended 0238_{16}. To compute $\oplus$, we again have to proceed digit-by-digit starting at the least significant one:

1. Since $3_{16} + 5_{16} = 8_{16}$ which fits into a decimal digit, we have $3_{16} \oplus 5_{16} = 8$ with no carry.

2. $9_{16} + 4_{16} = D_{16}$ which does **not** fit into a decimal digit. We need to regard it as a two-digit result 13 and truncate it to $9_{16} \oplus 4_{16} = D_{16} - 10 = 3$ with carry.

3. $0_{16} + 1_{16} + 1_{16} = 2_{16}$ fits, so we have $0_{16} \oplus 1_{16} + 1 = 2$ with no carry.

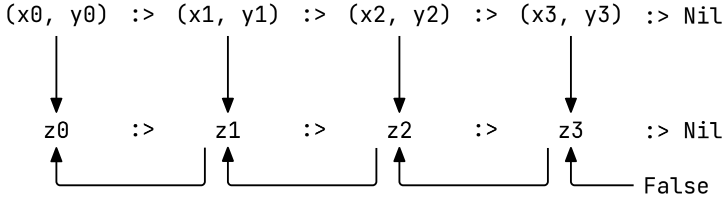
4. We still need to compute $0_{16} \oplus 0_{16} = 0$ because we are operating on a fixed-width representation of four digits.

5. We've run out of digits, giving us the result $0093_{16} \oplus 0145_{16} = 0238_{16}$ which is exactly what we wanted.

For a Haskell implementation of the above, we will use the `Index 10` type to represent (decimal) digits, which allows us to use `bitCoerce` to convert to an `Unsigned 4` hexadecimal digit for the purposes of computing one digit of ⊕.

```
type BCD n = Vec n Digit
type Digit = Index 10

fromDigit :: Digit -> Unsigned 4
fromDigit = bitCoerce
```

We already know roughly how addition should work: we start from the least-significant digits (i.e. on the right), initially with no carry, and keep adding the digits, propagating the carry:


(x0, y0) :> (x1, y1) :> (x2, y2) :> (x3, y3) :> Nil
z0 :> z1 :> z2 :> z3 :> Nil
False

```
addBCD :: BCD n -> BCD n -> BCD n
addBCD xs ys = snd . mapAccumR addDigit False $ zip xs ys
```

But when we get to implementing `addDigit`, we find that it is not as simple as adding them as `Unsigned 4` numbers. For example, suppose we tried to use the following function:

```
addDigitWrong :: Bool -> (Digit, Digit) -> (Bool, Digit)
addDigitWrong c (x, y) = (c', bitCoerce z')
  where
    z = fromDigit x + fromDigit y + if c then 1 else 0
    (c', z') = if z <= 9 then (False, z) else (True, z - 10)
```

While straightforward, this is not correct. The problem comes because adding two decimal digits can lead to a hexadecimal overflow. For example, let's say `c` is `False`, and `x` and `y` are 8 and 9. In this case, their sum is $17 = 11_{16}$, which overflows the 4-bit unsigned value into 1_{16}, leading to the incorrect result of no carry-out and a digit of 1 (instead of the correct result of digit 7 with carry-out).

Instead, we need to do the 4-bit addition while allowing for the possibility of an overflow, and do the decimal truncation and the carry detection appropriately. The largest possible result (with carry-in) is $9 + 9 + 1 = 19 = 13_{16}$ which easily fits into 5 bits, so we can extend each 4-bit `fromDigit` value with an extra 0 bit, and do the addition and the truncation in `Unsigned 5`:

```
addDigit :: Bool -> (Digit, Digit) -> (Bool, Digit)
addDigit c (x, y) = (c', fromIntegral z')
  where
    z :: Unsigned 5
    z = extend (fromDigit x) + extend (fromDigit y) + if c then 1 else 0

    (c', z') = if z <= 9 then (False, z) else (True, z - 10)
```

Or, written a bit more succinctly, using Clash's `add` function instead of `(+)` which is for exactly this use case where the result type is slightly larger than the arguments:

```
z = add (fromDigit x) (fromDigit y) + if c then 1 else 0
```

We can similarly implement BCD subtraction, using `sub` instead of `add`, and passing a `borrow` flag instead of a `carry`:

```
subBCD :: BCD n -> BCD n -> BCD n
subBCD xs ys = snd . mapAccumR subDigit False $ zip xs ys
  where
    subDigit :: Bool -> (Digit, Digit) -> (Bool, Digit)
    subDigit b (x, y) = (b', fromIntegral z')
      where
        z = sub (fromDigit x) (fromDigit y) - if b then 1 else 0

        (b', z') = if z <= 9 then (False, z) else (True, z + 10)
```

6.2.1 Testing

Since these are normal, pure Haskell functions, we can convince ourselves that `addBCD` is correct using e.g. *QuickCheck*, by comparing the result of `addBCD` (converted back to a normal `Integer`) with using `+` on `Integer`s. This is greatly helped by Clash providing `Arbitrary` instances for `Unsigned`/`Signed`, `Index`, and `Vec n a`, among others. Also, since the code below is only used in QuickCheck property testing, and not in actual synthesis, it is not a problem that `Integer` has no fixed size.

```
fromBCD :: BCD n -> Integer
fromBCD = foldl (\x d -> x * 10 + fromIntegral d) 0

infix 4 ~=
(~=) :: forall n. (KnownNat n) => BCD n -> Integer -> Bool
x ~= y = fromBCD x == y `mod` magnitude
  where
    magnitude = 10 ^ natVal (SNat @n)

prop_add :: forall n. (KnownNat n) => BCD n -> BCD n -> Bool
prop_add x y = addBCD x y ~= fromBCD x + fromBCD y

prop_sub :: forall n. (KnownNat n) => BCD n -> BCD n -> Bool
prop_sub x y = subBCD x y ~= fromBCD x - fromBCD y
```

If we try `addBCD` with `addDigitWrong`, QuickCheck makes short work of rejecting it:

```
> quickCheck (prop_add @8)
*** Failed! Falsifiable (after 8 tests and 1 shrink):
<0,1,0,9,7,5,6,0>
<9,6,1,6,3,7,3,5>
> quickCheck (prop_add @1)
*** Failed! Falsifiable (after 9 tests):
<9>
<7>
```

but with the correct implementation of `addDigit`, `addBCD` now passes with any number of digits; for example:

```
> quickCheck (prop_add @8)
+++ OK, passed 100 tests.
> quickCheck (prop_sub @10)
+++ OK, passed 100 tests.
```

6.3 State and state transitions

It is clear from the initial example sequence of input and output that there also needs to be some internal state, storing not only the accumulated value so far, but also the next operator (addition or subtraction) to be applied. Moreover, the current input also needs to be stored somewhere until it is “committed” into the accumulator by pressing a new operator key. Here is the same sequence, with the internal state noted as well:

Step	Input	Display	Value	Next operator
0.		`0`	0	`+`
1.	9	`9`	0	`+`
2.	3	`93`	0	`+`
3.	+	`93`	93	`+`
4.	1	`1`	93	`+`
5.	4	`14`	93	`+`
6.	6	`146`	93	`+`
7.	Backspace	`14`	93	`+`
8.	5	`145`	93	`+`
9.	-	`238`	238	`-`
10.	5	`5`	238	`-`
11.	3	`53`	238	`-`
12.	=	`185`	185	`-`

Note that in steps 0, 3, 9 and 12, there is no currently edited input, and so what is displayed is the accumulated value.

A simple product type can capture the state of our calculator quite easily, parameterized by the number of (BCD) digits to store and do calculations in: `value` is the accumulator, `opBuf` is the buffer holding the next operation to apply, and `inputBuf` is the input buffer. An `inputBuf` value of `Nothing` will be used to denote the state before any input has happened, in which case the accumulator value should be shown instead.

```
data Op
    = Add
    | Subtract
    deriving (Show, Generic, NFDataX)

data St n = MkSt
    { value :: BCD n
    , opBuf :: Op
    , inputBuf :: Maybe (BCD n)
    }
    deriving (Show, Generic, NFDataX)

initSt :: (KnownNat n) => St n
initSt = MkSt{ value = repeat 0, opBuf = Add, inputBuf = Nothing }
```

In any given State, the displayed number will be either value or inputBuf, as a sequence of *n* decimal digits. We'll remove leading zeroes, but if there are no non-zero digits, we still want to display "0" instead of nothing at all:

```
removeLeadingZeroes :: (KnownNat n) => Vec n Digit -> Vec n (Maybe Digit)
removeLeadingZeroes digits = case mapAccumL step False digits of
  (False, _) -> repeat Nothing <<+ Just 0
  (True, digits') -> digits'
 where
  step False 0 = (False, Nothing)
  step _ d = (True, Just d)

displayedDigits :: (KnownNat n) => St n -> Vec n (Maybe Digit)
displayedDigits MkSt{..} = removeLeadingZeroes $ fromMaybe value inputBuf
```

Let's try it out:

```
> removeLeadingZeroes (0 :> 0 :> 1 :> 2 :> Nil)
<Nothing,Nothing,Just 1,Just 2>
```

Looks good, but let's make it easier on the eyes by turning the displayed digits into a String. Similarly to the accessory functions for writing QuickCheck tests, here we don't care about synthesizability: this next function is only going to be useful for testing and simulation. So we can freely use list functions and types of non-fixed size:

```
import Data.Char (intToDigit)
import qualified Data.List as L

prettyDigits :: Vec n (Maybe Digit) -> String
prettyDigits = L.map (maybe ' ' (intToDigit . fromIntegral)) . toList
```

Better?

```
> putStrLn $ prettyDigits $ removeLeadingZeroes (0 :> 0 :> 1 :> 2 :> Nil)
 12
```

Better.

Now that we have a good grasp of the internal state, it is time to think about the state transitions. These transitions will be triggered by commands coming from the user; and we already know what commands we want to support:

```
data Cmd
    = Digit Digit
    | Op Op
    | Backspace
    | Clear
    | Equals
    deriving (Show, Generic, NFDataX)
```

We describe the *action* of these commands on the state in the usual way, as a function that maps the state before the transition to the state after:

```
update :: (KnownNat n) => Cmd -> St n -> St n
```

For `Clear`, we restore the initial state:

```
update Clear _ = initSt
```

For `Digit d`, we can simply shift in `d` from the right to the input buffer, initializing it to `0 :> 0 :> ... :> 0 :> Nil` if needed. For `Backspace`, we can do exactly the same, just shifting in `0` from the left, since this will shift out the rightmost digit into oblivion.

```
update (Digit d) s@MkSt{..} = s
    { inputBuf = Just $ fromMaybe (repeat 0) inputBuf <<+ d }
update Backspace s@MkSt{..} = s
    { inputBuf = Just $ 0 +>> fromMaybe (repeat 0) inputBuf }
```

For `Equals` and `Op`, we need to commit the current input buffer to the accumulator `value`. The only difference between the two is whether the current operator buffer is kept or overwritten by the new `op`:

```
update Equals s@MkSt{..} = compute s
update (Op op) s = (compute s){ opBuf = op }

compute :: (KnownNat n) => St n -> St n
compute s@MkSt{..} = s{ value = newValue, inputBuf = Nothing }
  where
    newValue = case opBuf of
        Add -> maybe value (addBCD value) inputBuf
        Subtract -> maybe value (subBCD value) inputBuf
```

6.4 An interactive software implementation

At this point, we have all the pieces of our calculator that don't directly interact with peripherals:

- The arithmetic functionality is implemented as pure functions, on a representation that is both machine-friendly and easy to manipulate and display.
- The internal state of our circuit-to-be is described as a Haskell datatype which only uses synthesizable field types.
- Input events (user commands) are described as a Haskell sum type, again synthesizable.
- State transition is described as the pure action of input events on the state.

Before moving on to writing the Clash parts that "make it all tick", i.e. the RTL model of the full calculator circuit, let's see how these same parts can be used from straight Haskell to create an interactive software implementation. What makes this straightforward is the fact that all the pieces so far are pure (non-`Signal`) functions, so we can use them just as well from the `Signal` world of circuits as we did from `IO` in a "computer program running on a computer" setting. We will keep coming back to this technique in later designs, since it allows interactive testing of the components doing the heavy lifting, in a familiar Haskell software environment.

For a calculator this simple, the software I/O is going to be simple as well. We'll use the *Terminal* package to implement a text interface without worrying about the platform-specific details of moving around text cursors. This means our `main` will look like this:

```
import Clash.Prelude
import System.Terminal

main :: IO ()
main = withTerminal $ runTerminalT $ do
    putStringLn "Welcome to Calculator."
    runCalculator
```

After a brief welcome message, we get to `runCalculator` which is where the real program will live. For simplicity's sake, we'll just pass around the `State` directly, using `Control.Monad.Extra.loopM` to drive the main loop. In the main loop, we show the output for the current state, wait for an input event, and either exit (if the event is a Ctrl + C interrupt), or keep running with the new state that is the result of processing the event:

```
import Control.Monad.Extra (loopM)

runCalculator = flip loopM (initSt @8) $ \st -> do
    display st
    ev <- awaitEvent
    return $ case ev of
        Left Interrupt -> Right ()
        Right ev -> Left $ processEvent ev st
```

Implementing the output function `display` is trivial now because we already have all the pieces necessary; we just need to wrap it in some *Terminal* commands to keep updating a single line of text:

```
display st = do
    setCursorColumn 0
    putString $ prettyDigits . displayedDigits $ st
    flush
```

To implement `processEvent`, we parse the `Event` argument into a `Cmd`, and return the `updated` state. If the `Event` doesn't correspond to a `Cmd`, we simply keep going with the current state.

```
processEvent ev = maybe id update (eventToCmd ev)
```

There is no finesse to parsing the events: we just need to decide on a mapping of keys to commands. The user-friendly mapping is the one that maps `+` to addition and so on:

```
import Data.Char (digitToInt)

eventToCmd :: Event -> Maybe Cmd
eventToCmd (KeyEvent key mod) | mod == mempty = case key of
    CharKey c | c `elem` ['0'..'9'] -> Just $ Digit . fromIntegral .
    digitToInt $ c
    CharKey '+' -> Just $ Op Add
    CharKey '-' -> Just $ Op Subtract
    CharKey '=' -> Just Equals
    EnterKey -> Just Equals
    BackspaceKey -> Just Backspace
    DeleteKey -> Just Clear
    _ -> Nothing
eventToCmd _ = Nothing
```

But we can also bring our software implementation closer to the planned hardware by mapping `Events` to keys of a hexadecimal keypad, and then mapping those keys to commands. This way, we can reuse the second part in the hardware implementation. But because on a computer keyboard, the layout of the digits `0` to `9` and the letters `A` to `F` is nothing like a 4 × 4 keypad, this version is more suited for testing than actual use.

In `eventToKey`, we can use `digitToInt` on both `0..9` and `a..f` because it can parse not just decimal, but hexadecimal digits. Once we have both `keyToCmd` and `eventToKey`, the definition of `eventToCmd` is a simple composition in the `Maybe` monad.

```
type Hex = Unsigned 4

keyToCmd :: Hex -> Maybe Cmd
keyToCmd n | n <= 9 = Just $ Digit $ bitCoerce n
keyToCmd 0xa = Just $ Op Add
keyToCmd 0xb = Just Backspace
keyToCmd 0xc = Just Clear
keyToCmd 0xd = Just $ Op Subtract
keyToCmd 0xe = Just Equals
keyToCmd _ = Nothing

eventToKey :: Event -> Maybe Hex
eventToKey (KeyEvent key mod) | mod == mempty = case key of
    CharKey c | c `elem` ['0'..'9'] ->
        Just $ fromIntegral . digitToInt $ c
    CharKey c | c `elem` ['a'..'f'] ->
        Just $ fromIntegral . digitToInt $ c
    _ -> Nothing
eventToKey _ = Nothing

eventToCmd :: Event -> Maybe Cmd
eventToCmd = keyToCmd <=< eventToKey
```

The final program isn't the most photogenic, so try it out interactively yourself:

```
Welcome to Calculator.
 5318008
```

6.5 Hooking it up to hardware peripherals

We have put in so much work into our humble little calculator in this and the previous chapter that writing the full hardware implementation is now a breeze:

- All the peripheral drivers are already implemented in the `driveSS` and `inputKeymap` functions we've developed for the keypad example.

- All the parts for the main logic are already implemented in `update` and `displayedDigits`.

The connective tissue between these parts just needs to maintain the state in a register, and connect the decoded inputs and outputs:

```
logic
    :: forall n dom. (KnownNat n, HiddenClockResetEnable dom)
    => Signal dom (Maybe Cmd)
    -> Signal dom (Vec n (Maybe Digit))
logic cmd = displayedDigits <$> s
  where
    s = register initSt (maybe id update <$> cmd <*> s)
```

This `logic` function now lives in the world of `Signals`, but its argument and result types are straightforward enough that we can very easily test it using the Clash simulator. For example, here is the definition of a `Signal` that corresponds to the input events from our $93+145-53$ example, with some cycles with no keypresses thrown in for good measure:

```
testInput :: Signal dom (Maybe Hex)
testInput = fromList . (<> L.repeat Nothing) . stretch $
    [ 9, 3, 0xa
    , 1, 4, 6, 0xb, 5, 0xd
    , 5, 3, 0xe
    ]
  where
    stretch = L.concatMap $ \x -> [Nothing, Just x, Nothing]
```

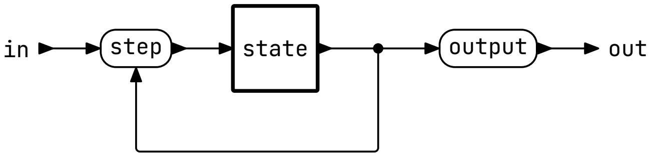
Running `logic testInput` in the Clash simulator for $13 * 3$ cycles with a 4-digit display, we see that the output matches the expected values:

```
> :{
| mapM_ (print . toList . map (maybe ' ' (intToDigit . fromIntegral))) $
    L.map L.head . L.group $
    sampleN @System (13 * 3) $
    logic @4 . fmap (keyToCmd =<<) $ testInput
| :}
```

```
"   0"
"   9"
"  93"
"   1"
"  14"
" 146"
"  14"
" 145"
" 238"
"   5"
"  53"
" 185"
```

6.5.1 Moore machines

Our `logic` circuit has a very particular structure: at every tick of the clock, it takes its current state and computes a new state from it and some input. The output of the circuit is fully determined by the state (by applying `displayedDigits` on it). We can imagine it as a finite-state stream processor that consumes the values of its input signal, updates its internal state, and produces the values of its output signal as shown here:



This kind of stream processor finite state machine is called a *Moore machine*, and we have defined a particular Moore machine for our calculator where the `step` function is `maybe id update` and the `output` function is `displayedDigits`. Clash provides a library function `moore` which implements this pattern, which allows us to avoid the recursion in the definition of `logic`, making it easier to understand:

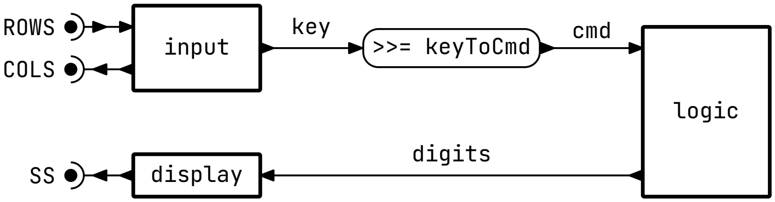
```
moore
    :: (HiddenClockResetEnable dom, NFDataX s)
    => (s -> i -> s) -> (s -> o) -> s -> Signal dom i -> Signal dom o

logic
    :: forall n dom. (KnownNat n, HiddenClockResetEnable dom)
    => Signal dom (Maybe Cmd)
    -> Signal dom (Vec n (Maybe Digit))
logic = moore (flip $ maybe id update) displayedDigits initSt
```

6.5.2 Putting it all together

The main top-level entity has the same definition as the "keypad with seven-segment output" example, just replacing the board `logic` with the above one. For completeness's sake, here it is in full detail:

```
topEntity
    :: "CLK"    ::: Clock System
    -> "ROWS"   ::: Signal System (Vec 4 Bit)
    -> ( "SS"   ::: SevenSegment System 4 High Low Low
       , "COLS" ::: Signal System (Vec 4 (Active Low))
       )
topEntity = withResetEnableGen board
  where
    board rows = (display digits, cols)
      where
        display = driveSS (\x -> (encodeHexSS . bitCoerce $ x, False))
        input = inputKeypad keymap
        digits = logic cmd

        (cols, key) = input rows
        cmd = (keyToCmd =<<) <$> key
```

This corresponds to the following structure:



Exercises:

- Change `E` to mean **Erase**, i.e. to reset the current number to 0 without changing the running value. With this change, it's probably a good idea to also re-map `F` to **Flush**, which does the same as **Equals**.

- Change the software implementation that uses `keyToCmd` to decode keyboard events *positionally* instead of *symbolically*, i.e. instead of mapping `0` to `0`, `1` to `1` etc., choose a 4 × 4 region of the standard QWERTY layout (the slanted square spanning `1` and `V` is an obvious choice). Is that version more or less confusing than the symbolic one? What if you have the hardware keypad in front of you?

- Change the software implementation to use a `State` monad underneath `TerminalT`. Now you can rewrite `runCalculator` to use `Control.Monad.Extra.whileM` instead of `loopM`. Or, go all the way and use `MaybeT (State St)`, exiting with `mzero` and doing everything in a simple `forever` block.

- Change the *hardware* implementation to use a `State` monad through and through: the types change to `update :: Cmd -> State (St n) ()` and `displayedDigits :: State (St n) (Vec n (Maybe Digit))`. At first, this juice might not seem worth the squeeze, since now we have to play `execState` and `evalState` games in `logic`. On the other hand, the changes for the previous exercise become less tedious. In later chapters, we will revisit this idea and build some combinators that will also clean up the new definition of `logic`.

6.6 Summary

- We started with **pure functions** implementing the main pieces of functionality. While writing these, we kept an eye on hardware realizability, but used descriptive types.

- These pure functions can be **unit-tested using standard Haskell techniques** like QuickCheck property testing.

- The **circuit state** and the **state transitions** can also be implemented with no `Signals` in sight. This makes it easy to develop **interactive software frontends** using off-the-shelf I/O libraries like *Terminal* or *SDL2* that allows **interactive integration testing**.

- Once all the pieces are there, the **hardware implementation** becomes a simple matter of taking the right **peripheral drivers** and connecting them to **a state register updating on every input command**. This is an instance of the **Moore machine** pattern for stream processing.

- By separating the peripheral drivers from the main board logic, we can write **integration tests using the Clash simulator** without having to simulate the intricacies of the peripherals themselves.

7 Video Output Using VGA

In this chapter, we implement interfacing with video displays. Video is perhaps the most important and "fun" modality of personal computing – for example, "video game" is a common term for computer games, but no one calls them "joystick games" or "mouse and keyboard games".

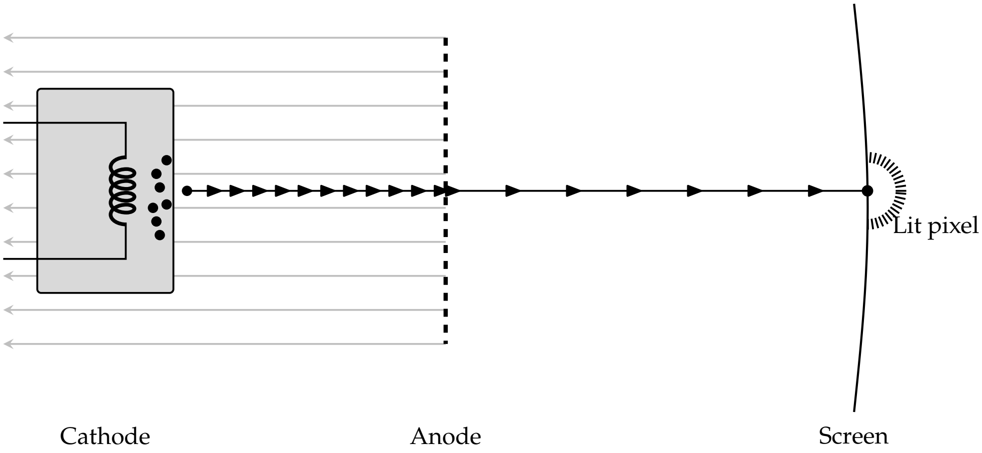
These days, when we think of a screen, we think of displays based on discrete pixels: LCD, plasma displays, or OLEDs. However, the computers we are exploring in this book predate these technologies: the display contemporary to the first home computers was the *cathode ray tube* (or *CRT*). These displays are "dumb" devices: there is a very close correspondence between the input signal and the internal mechanism of turning that signal into a visible picture. For this reason, understanding how CRTs work is crucial in understanding their interface format.

7.1 Basic operation of a CRT display

The goal of a video display is to turn an electronic signal into visible picture.

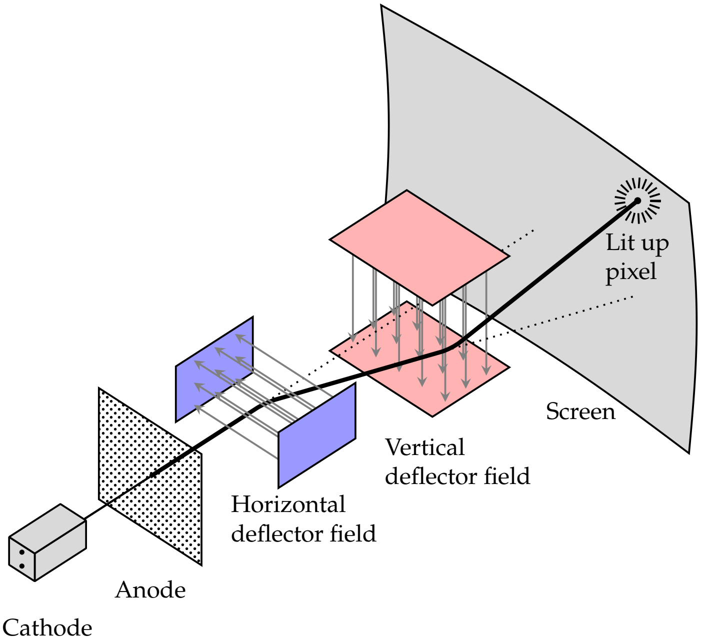
The basic idea behind a cathode-ray tube display is to target a stream of electrons at a screen made of some fluorescent material, i.e. that lights up when charged. Since electrons are statically charged, they can be accelerated by an electric field. This allows us to create an electron beam by taking a source of slow, meandering electrons such as a heated coil (called a *cathode*), and putting a positively charged grid (the *anode*) in front of it. Since electrons are negatively charged, the positive charge of the grid pulls on them, accelerating them in the direction of the grid. This is called an *electron gun*.

Note that in this figure, the arrows denoting the electric fields are, somewhat counter-intuitively, going against the direction of acceleration; similarly, the deflectors in the next figure seem to operate "in reverse". This is because electric fields are traditionally drawn pointing from the positive towards the negative pole, and electrons are negatively charged.



7.1.1 Drawing in black & white

By using an electrostatic or magnetic field perpendicular to the line connecting the electron gun to the screen, we can control its trajectory. By using two fields, we can deflect the beam in the two perpendicular directions, targeting any point of the screen.



As the electron beam leaves a part of the screen, it takes some time for the fluorescent material to lose its charge and stop emitting light. Thus, by quickly moving the beam around while changing its intensity, we can draw an image. This image will be light in the parts hit by the electron beam, and dark everywhere else. The actual color of the light parts depends on the choice of fluorescent material; white, green, and amber have been common.

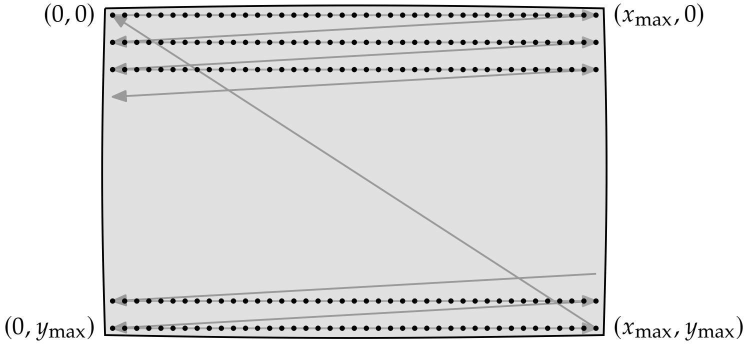
7.1.2 Drawing in monochrome

If we want to generate different shades of the one color, we can simply change the magnitude of the negative voltage in the electron gun, between the cathode and the anode, thereby controlling the electron beam's intensity. A more intense beam causes the targeted part of the fluorescent screen to light up more.

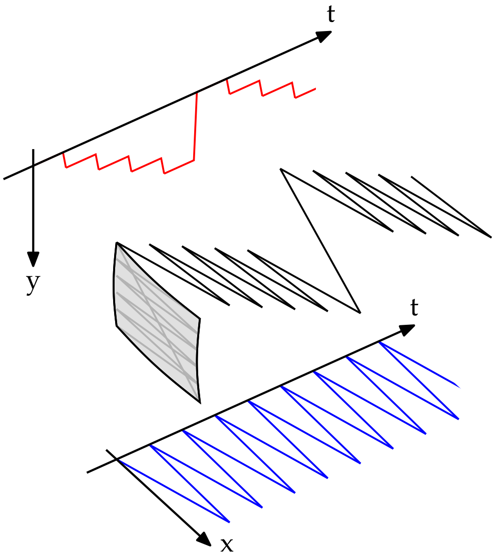
7.1.3 Vector vs. raster displays

The design described so far means we can generate a picture by using a time-varying, three-dimensional signal consisting of the 2D screen coordinates (to be applied to the two perpendicular electrodes or coils) and the intensity. For example, to draw a straight segment between two points on the screen, we can generate a signal that changes x and y from one endpoint to the other, again and again, fast enough that by the time we restart the cycle, the first part of the screen would just start going dark again. This method of driving a CRT is called a *vector display*, and it is how oscilloscopes work. However, apart from some specialized arcade and home gaming computers, this is not how most CRT screens work.

Instead, televisions and most computer screens are *raster displays*: the image is drawn by putting small dots of differing intensity next to each other, filling the whole screen with a two-dimensional matrix of raster dots. Instead of following the geometry of the scene, for every frame displayed, every dot is redrawn one after the other, in a regular zigzag sequence.

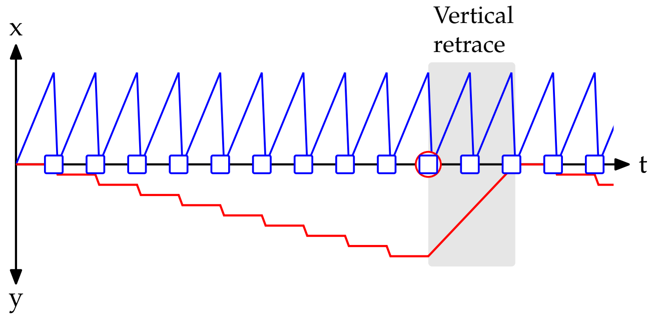


In this setup, the input signal has only one degree of freedom: the intended luminosity of each raster dot, left to right, top to bottom. The rest is handled by circuitry in the display itself: for each line, the horizontal beam deflection grows gradually from minimum to maximum; meanwhile the vertical beam deflection stays constant. Then, the vertical deflection grows by a bit, and the horizontal deflection goes back to its minimum value. Thus, the horizontal deflection is a regular sawtooth pattern and the vertical one is a step function.



Taken this way, we can regard the input signal of a raster display as a serial protocol. Synchronization between the electron beam deflector circuitry and the video signal is achieved by two pulses: one at the end of each raster line, resetting the horizontal sawtooth and incrementing the vertical step function; and one at the end of each frame, resetting the vertical signal. Between these synchronization pulses, the input signal rate is determined by the video resolution and the frame rate; we will look at this in detail when we look at the particulars of the VGA signal standard.

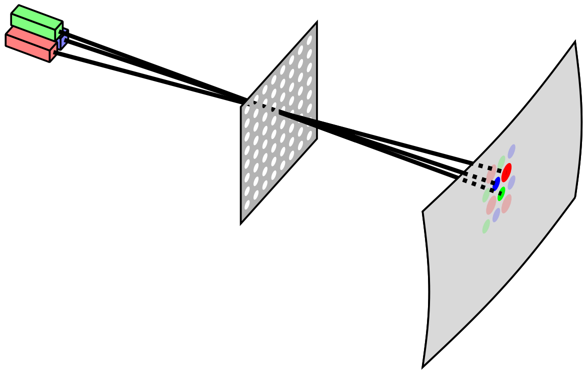
In the following illustration, the blue square marks on the time axis represent the horizontal sync pulses, and the red circle mark shows the timing of the vertical sync pulse. Note that vertical retracing can take several horizontal scan lines' worth of time; in this example, it takes two complete horizontal scan lines for the vertical retrace, during which no picture is generated.



7.1.4 Color video

Because of how the human eye works, it can be tricked into perceiving any color by using just red, green and blue light, and choosing the right ratio of intensity. The details of this are well outside the scope of this book: here, we will just accept that "color video" is simply a combination of red, green, and blue video.

This gives us a simple way of adding color support to a CRT design: we can use three different materials for the screen, put three electron guns next to each other at a slight offset, and use a mask in front of the screens so that for every raster dot, two of the three screens are occluded from each electron beam. Each electron gun has its own anode, so each component's intensity can be controlled separately.



7.2 Video Graphics Array

The *Video Graphics Array* standard, commonly abbreviated VGA, is the most widely used analog video format for computer displays. Originating in 1987, it is more modern than the computers we build in this book. We will use it for video output because it lies in the sweet spot of video signal formats:

- Being an analog standard, VGA is based on the same fundamental theory of operation of CRTs as earlier designs, so "morally" it is not really anachronistic for the 1970s computers we build in this book.
- Unlike analog *television* standards that were designed to accommodate radio transmission, and burdened by layers of backwards compatibility, VGA has separate red/green/blue intensity lines and horizontal/vertical synchronization lines. This makes both generating and interpreting VGA signals very straightforward.
- Also unlike television standards, VGA is versatile in the resolutions and refresh rates it supports.
- Although the modern way of transmitting video is the digital format of HDMI, most contemporary computer displays still support VGA.
- Most hobbyist FPGA development boards either have VGA output connected to a digital to analog converter, or an HDMI serializer that takes a digitized VGA signal and encodes that as an HDMI stream.

7.2.1 Signal lines of VGA

Fundamentally, VGA consists of five signal lines transmitted from the controller to the display:

- Three *color channels* for red, green and blue. These are analog channels, i.e. there is a predetermined voltage range, with ground representing zero intensity (full darkness) in the given channel and +0.7 V representing maximum brightness.[1]
- The *horizontal sync* is a digital signal marking the end of the current raster line. It can be either active high or active low depending on the resolution and refresh rate used.
- The *vertical sync* behaves the same as the horizontal sync, marking the end of the current frame. Its polarity is not necessarily the same as the horizontal sync's.

[1] Depending on the display technology, pixels may always emit some light, i.e. a fully "dark" screen might still be brighter than the screen turned off.

In reality, VGA connectors and cables have more than five lines, since there are extensions to the VGA standard for querying display capabilities such as maximum resolution and refresh rate, which require data transfer in the reverse (display to controller) direction. Most VGA connectors on hobbyist FPGA boards don't connect to these lines, and they are not essential for video generation, so we will not discuss them in this book.

7.2.2 "Digital VGA"

In Clash, and in the logic circuitry of FPGAs in general, we work with digital signals. There is no Clash type for "an analog value between 0 and 0.7 V". Instead, we are going to generate a "digital VGA" signal consisting of multiple bits for each color channel, and using standard logical low/high voltage for the sync lines. Then, some peripheral hardware will convert that multi-bit digital signal to the analog format of VGA, and convert the sync voltage levels to 5V.

Thus, we are going to represent VGA output with a given bit width for the red, green and blue channels using the following datatype:

```
data VGAOut dom r g b = VGAOut
    { vgaHSync :: "HSYNC" ::: Signal dom Bit
    , vgaVSync :: "VSYNC" ::: Signal dom Bit
    , vgaR     :: "RED"   ::: Signal dom (Unsigned r)
    , vgaG     :: "GREEN" ::: Signal dom (Unsigned g)
    , vgaB     :: "BLUE"  ::: Signal dom (Unsigned b)
    }
```

The bit-depth of the color signals depends on the implementation of this conversion. Purpose-built integrated circuits can easily support 8 bits per channel; some hobbyist development boards instead use a resistor ladder for much smaller bit depth, such as 4 bits per channel, or 3 for red and green and 2 for blue for a total of 8.

Home computers of the era we are exploring in this book had much more limited palettes: for example, the Compucolor II from 1977 only supported 8 colors. This only requires 1 bit per channel: each channel is either turned off or on. However, other computers with comparable palette sizes, such as the Commodore 64 with its 16 colors, require finer control of the color lines: for example, the orange color consists of 87% red, 53% blue and 33% green, while the yellow is 93% red, 93% blue and 47% green[2]. The deeper our digital representation is, the better we can

[2] These intensity values are taken from one of several "Commodore 64 palettes" online. In reality, the C-64's video generated an analog TV signal, and the actual colors showing up on the screen depended on many factors, including the television set's characteristics and its brightness and contrast settings. Analog TV doesn't even use the red/green/blue color space to represent color signals.

approximate these colors.

7.2.3 Signal timing

When generating a VGA signal, there are two timing concerns:

- The timing of the synchronization pulses must correspond to a valid VGA mode.
- Between synchronization pulses, we need to keep track of time for addressing purposes.

For the first point, there is a collection of canonical sync patterns corresponding to various resolutions and refresh rates. For example, let's suppose we want to generate a 640 × 480 image, at 60 frames per second. Naïvely, we might think that requires a horizontal sync pulse $480 \cdot 60 = 28,800$ times a second, and a vertical one at every 480^{th} horizontal pulse. Reality is more complicated.

After drawing the visible portion of a line area, there is a blanking period before the next line starts. It is during this time that the (electrostatic or magnetic) field controlling the horizontal deflection is reset and the vertical one changes to the next line. This is why, in our previous timing diagram, the sawtooth pattern of the horizontal deflection had a slight slope on the horizontal retrace side instead of falling completely sharply.

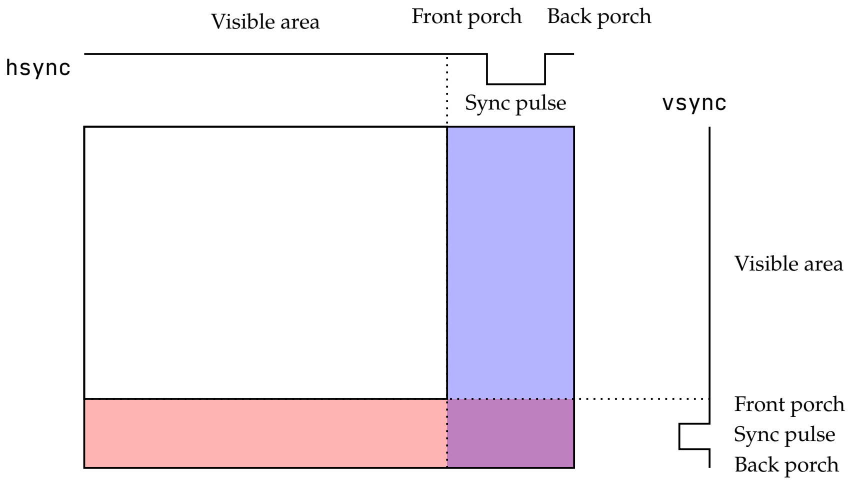
The resetting of the horizontal position is triggered by the horizontal sync pulse, which must have a prescribed length (the *sync pulse width*) and must occur after a certain time has passed since blanking started (the *front porch* length). After the sync pulse ends and the *back porch* period has passed, the blanking period ends and the next visible line starts. The vertical sync signal has similar timing constraints.

Looking at the specification for VGA mode 640 × 480@60, we find the following timing constraints for the horizontal sync signal:

- Horizontal visible area: 25.422 μs
- Horizontal front porch: 0.635 μs
- Horizontal negative pulse width: 3.813 μs
- Horizontal back porch: 1.907 μs
- Total line length: 31.778 μs (i.e. 31.46875 kHz)

And for the vertical one:

- Vertical visible area: 15.253 ms
- Vertical front porch: 349.551 μs
- Vertical negative pulse width: 63.555 μs



- Vertical back porch: 985.104 μs
- Total frame length: 16.651 ms (i.e. 60.054 Hz)

To understand where exactly these seemingly arbitrary numbers come from, we also need to understand how addressing works. On a raster CRT display, between pulses of the horizontal sync lines, the electron beam is gradually sweeping the current line left to right. This behavior is driven autonomously by circuitry in the display unit. Drawing horizontal patterns thus comes down to changing the color channels just at the right time, when the electron beam is at the right place. This requires the controller side to keep accurate time between sync pulses so it can know where the display side is aiming the electron beam. To make this possible, VGA defines the concept of the *pixel clock*, and all signal timing is done in terms of this clock:

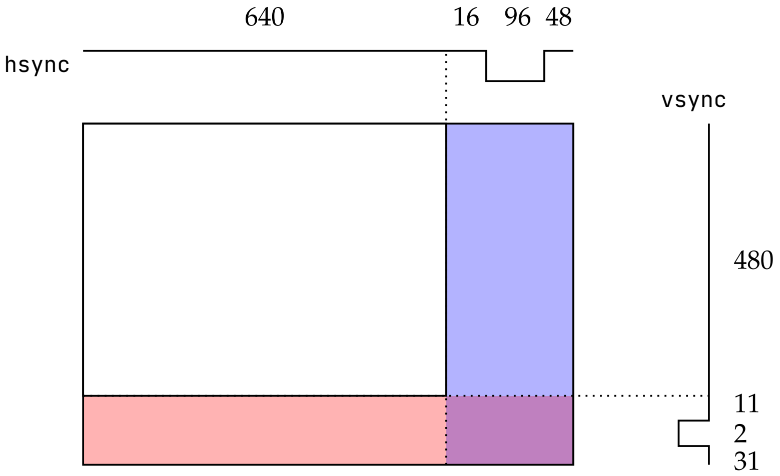
- The horizontal visible area, porch lengths and pulse width are all integer multiples of the pixel clock period.

- While scanning the visible area, the electron beam deflection is affine in the time since the sync pulse; in particular, it is linear in the time since the start of the visible area.

- The vertical visible area, porch lengths and pulse width are all integer multiples of the horizontal line length.

The previous, awkward and arbitrary-looking numbers for 640 × 480@60 can now be expressed much more cleanly:

- Pixel clock: 25.175 MHz
- Horizontal visible area, front porch, negative pulse width and back porch: 640, 16, 96 and 48 pixels
- Vertical visible area, front porch, negative pulse width and back porch: 480, 11, 2, and 31 lines



What this all means is that to generate a valid signal and to know which pixel to draw next, the controller can internally generate a pixel clock signal, and use that to drive various counters. For example, we can generate the correct horizontal sync signal by counting to 640, then to 16, then to 96, then to 48, and starting again; setting the output to high for the first two segments, low for the third one, and high again for the fourth one. While we are in the first segment, the value of the counter is exactly the X coordinate of the pixel just being drawn.

7.3 VGA from Clash

Now that we have an understanding of VGA, let's think about the design of a VGA controller. We want the controller to take care of generating the correct synchronization signals, and provide the coordinates of the currently scanned pixel. This way, we can connect these coordinate signals to other parts of our design that will use that information to compute the current pixel's color.

Thus, we want the driver to give us the following signals for a $w \times h$ resolution.

```
data VGADriver dom w h = VGADriver
    { vgaHSync :: Signal dom Bit
    , vgaVSync :: Signal dom Bit
    , vgaX :: Signal dom (Maybe (Index w))
    , vgaY :: Signal dom (Maybe (Index h))
    }
```

So what API do we need to provide for the driver? As a first approximation, given a timing description for a $w \times h$ resolution VGA mode, we should be able to generate the appropriate signals:

```
vgaDriver
    :: (HiddenClockResetEnable dom, KnownNat w, KnownNat h)
    => VGATimings w h
    -> VGADriver dom w h

vga640x480at60 :: VGATimings 640 480
vga640x480at75 :: VGATimings 640 480
```

However, `vgaDriver` can only meet the VGA mode's timing requirements if it is running at the pixel clock rate. To see why, imagine the example of trying to count at 25.175 MHz using an underlying clock of 32 MHz. The two closest approximations would be to count on one or two cycles, giving 32 or 16 MHz, both very, very far from the target.

Just generating the sync pulses wouldn't be impossible: for example, one horizontal line of 48 + 640 + 16 = 704 high and 96 low cycles (at 25.175 MHz) can be approximated by counting to 895 and 122, respectively, at 32 MHz, which corresponds to an exact match of a pixel clock of 25.172 MHz, i.e. off by less than 5 picoseconds per cycle. However, for the visible 814 cycles of those 1017, we have no good way of keeping count of the X coordinate in a scale from 0 to 639.

Instead, we are going to require the clock domain of the `VGADriver` to match the pixel clock frequency. We do this by including the pixel clock period length as a type-level index on `VGATimings`, and requiring a matching `DomainPeriod`. Now the types differ between the two 640 × 480 resolutions at refresh rates of 60 and the 75:

```
vgaDriver
    :: (HiddenClockResetEnable dom, KnownNat w, KnownNat h)
    => (DomainPeriod dom ~ ps)
    => VGATimings ps w h
    -> VGADriver dom w h

vga640x480at60 :: VGATimings (HzToPeriod 25_175_000) 640 480
vga640x480at75 :: VGATimings (HzToPeriod 31_500_000) 640 480
```

To implement the actual counting in one dimension (horizontal or vertical), we are going to use a sum of `Index` types. This ensures a representation that uses the minimum number of bits, instead of wasting bits on generic `Int` counters.

```
data VGAState visible front pulse back
    = Visible (Index visible)
    | FrontPorch (Index front)
    | SyncPulse (Index pulse)
    | BackPorch (Index back)
    deriving (Show, Generic, NFDataX)
```

However, there is a bit of a tension here between having such a tightly typed representation of `VGAState`, and hiding the front porch, pulse width, and back porch sizes from the type of a VGA mode description. On the other hand, we really don't want to put them into the type of e.g. `vga640x480at60`, since the only thing you should need to know to *use* it is that given a pixel clock of 25.175, it will report coordinates of 640 × 480. We bridge this gap by storing the sync timings *existentially* in the representation of VGA timings:

```
{-# LANGUAGE ExistentialQuantification #-}

data VGATiming (visible :: Nat) = forall front pulse back. VGATiming
    { polarity :: Polarity
    , preWidth :: SNat front
    , pulseWidth :: SNat pulse
    , postWidth :: SNat back
    }
deriving instance Show (VGATiming vis)

data VGATimings (ps :: Nat) (w :: Nat) (h :: Nat) = VGATimings
    { vgaHorizTiming :: VGATiming w
    , vgaVertTiming :: VGATiming h
    }
    deriving (Show)
```

On the term level, we store `SNat` singletons, which have the added benefit that pattern matching on them also gives `KnownNat` instances, which we'll need for implementing the state transition function on `VGAState`. But before that, let's write out our first VGA mode definition in full:

```
-- | VGA 640*480@60Hz, 25.175 MHz pixel clock
vga640x480at60 :: VGATimings (HzToPeriod 25_175_000) 640 480
```

```
vga640x480at60 = VGATimings
    { vgaHorizTiming = VGATiming Low (SNat @16) (SNat @96) (SNat @48)
    , vgaVertTiming  = VGATiming Low (SNat @11) (SNat @2)  (SNat @31)
    }
```

The workhorse of our `vgaDriver` function, then, will be an implementation of a state machine providing the following three observations over a base `VGAState`, for the three critical segments of the timeline: the visible part, the part when the sync pulse should be engaged, and when we're ready for the next cycle. Note how all of these functions have result types that don't refer to the `VGAState` type parameters that are represented existentially in `VGATiming`.

```
visible :: VGAState visible front pulse back -> Maybe (Index visible)
visible (Visible coord) = Just coord
visible _ = Nothing

sync :: VGAState visible front pulse back -> Bool
sync SyncPulse{} = True
sync _ = False

end :: (KnownNat back) => VGAState visible front pulse back -> Bool
end (BackPorch cnt) | cnt == maxBound = True
end _ = False
```

We implement the state transition function `vgaCounter` via a datatype wrapper having, again, an existential type. Note that the front porch, pulse length, and back porch fields of `VGATiming` are working double duty: they pick the right `Index` types for the `VGAState` constructors, while also bringing the necessary `KnownNat` instances in scope for the `succIdx` call inside `count`. The `mkVGACounter` function is used to bolt down the `front`, `pulse`, and `back` existential type variables.

```
type Step a = a -> a

data VGACounter visible =
    forall front pulse back.
    (KnownNat front, KnownNat pulse, KnownNat back)
    => VGACounter (Step (VGAState visible front pulse back))

mkVGACounter
    :: SNat front -> SNat pulse -> SNat back
    -> Step (VGAState visible front pulse back)
    -> VGACounter visible
mkVGACounter SNat SNat SNat = VGACounter
```

```
vgaCounter
    :: (KnownNat visible) => VGATiming visible -> VGACounter visible
vgaCounter (VGATiming _ front@SNat pulse@SNat back@SNat) =
    mkVGACounter front pulse back $ \case
        Visible cnt    -> count Visible    FrontPorch cnt
        FrontPorch cnt -> count FrontPorch SyncPulse  cnt
        SyncPulse cnt  -> count SyncPulse  BackPorch  cnt
        BackPorch cnt  -> count BackPorch  Visible    cnt
  where
    count
        :: (KnownNat n, KnownNat m)
        => (Index n -> a) -> (Index m -> a) -> Index n -> a
    count this next = maybe (next 0) this . succIdx
```

Finally we have the full toolbox to implement `vgaDriver`: horizontally, it is simple register counting by `vgaCounter`; vertically, it is a register of `vgaCounter` that updates only when the horizontal counter is at its end. The correct sync pulse polarity is implemented by using a dynamic version of `toActive` that uses a term-level parameter instead of the type-level tag of `Active`.

```
toActiveDyn :: Polarity -> Bool -> Bit
toActiveDyn High = boolToBit
toActiveDyn Low = complement . boolToBit

vgaDriver
    :: (HiddenClockResetEnable dom, KnownNat w, KnownNat h)
    => (DomainPeriod dom ~ ps)
    => VGATimings ps w h
    -> VGADriver dom w h
vgaDriver VGATimings{..} = case (vgaCounter vgaHorizTiming, vgaCounter
    vgaVertTiming) of
    (VGACounter nextH, VGACounter nextV) -> VGADriver{..}
      where
        stateH = register (Visible 0) $ nextH <$> stateH
        stateV = regEn (Visible 0) endLine $ nextV <$> stateV

        vgaX = visible <$> stateH
        vgaHSync = toActiveDyn (polarity vgaHorizTiming) . sync <$>
    stateH
        endLine = end <$> stateH

        vgaY = visible <$> stateV
        vgaVSync = toActiveDyn (polarity vgaVertTiming) . sync <$> stateV
```

And so this takes care of the correct timing for both synchronization and addressing. However, the VGA standard also requires the color lines to go to zero in the blanking period, to calibrate the baseline of the analog signals. Also, some FPGA dev boards use a VGA-to-HDMI encoder that requires a separate "display enable" signal. For these reasons, we split the `VGADriver` and `VGAOut` structs, add a display enable line, and write a small utility function that ensures blanking in the non-visible area.

```
data VGASync dom = VGASync
    { vgaHSync :: "HSYNC" ::: Signal dom Bit
    , vgaVSync :: "VSYNC" ::: Signal dom Bit
    , vgaDE    :: "DE"    ::: Signal dom Bool
    }

data VGAOut dom r g b = VGAOut
    { vgaSync  :: VGASync dom
    , vgaR     :: "RED"   ::: Signal dom (Unsigned r)
    , vgaG     :: "GREEN" ::: Signal dom (Unsigned g)
    , vgaB     :: "BLUE"  ::: Signal dom (Unsigned b)
    }

data VGADriver dom w h = VGADriver
    { vgaSync :: VGASync dom
    , vgaX :: Signal dom (Maybe (Index w))
    , vgaY :: Signal dom (Maybe (Index h))
    }

vgaOut
    :: (HiddenClockResetEnable dom, KnownNat r, KnownNat g, KnownNat b)
    => VGASync dom
    -> Signal dom (Unsigned r, Unsigned g, Unsigned b)
    -> VGAOut dom r g b
vgaOut vgaSync@VGASync{..} rgb = VGAOut{..}
  where
    (vgaR, vgaG, vgaB) = unbundle $ blank rgb

    blank = mux (not <$> vgaDE) (0, 0, 0)
```

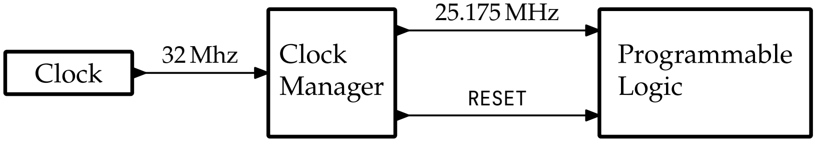
The change to `vgaDriver` is minimal, since `vgaDE` is trivially computable from `vgaX` and `vgaY`:

```
vgaDE = isJust <$> vgaX .&&. isJust <$> vgaY
```

7.3.1 Pixel clock management

The type of `vgaDriver` prescribes a clock domain running exactly at the pixel clock frequency. However, the clock signal is provided by an external source, running at some set rate. What happens if the clock rate is not the same as the pixel clock rate for the desired VGA mode?

The answer to this question is that while yes, the clock signal is external to the logic circuitry programmed from HDLs or Clash, it is still digitally configurable, by putting an extra element called a *clock manager* between the raw, fixed-rate clock source and the clock signal consumed by Clash.



In the above example, the clock manager is configured to convert the raw 32 MHz clock signal into a 25.175 Mhz output clock. As discussed above, this conversion cannot be done with just counters; the clock manager is a black box that couldn't be described using the RTL abstraction. Also, since it can take a couple of input cycles for the clock manager to "lock onto" the right output frequency, an output suitable for use as a reset line is usually provided as well.

Unfortunately, the configuration of clock managers is fully proprietary, differing not only between FPGA vendors but even FPGA product lines from the same vendor; readers will need to look up the details in the reference documentation provided by the vendor of their chosen development board. In this book, all circuits for a given design always run in one single, shared clock domain, assumed to use the right clock rate. For example, we will write the following Clash program to display a black screen in a 640 × 480 VGA mode:

```
createDomain vSystem{vName="Dom25", vPeriod = hzToPeriod 25_175_000}

topEntity
    :: "CLK_25MHZ" ::: Clock Dom25
    -> "RESET"     ::: Reset Dom25
    -> "VGA"       ::: VGAOut Dom25 8 8 8
topEntity = withEnableGen board
  where
    board = vgaOut vgaSync (pure (0, 0, 0))
      where
        VGADriver{..} = vgaDriver vga640x480at60
```

This approach allows us to put the details of the clock manager into external, board-specific HDL files. This HDL file will wrap the Clash-generated HDL with some hand-written code that wires up the reset and clock signals using the clock manager pin naming conventions of the FPGA vendor. The following is an example for a Xilinx 7-series FPGA development board, written in Verilog. All signals except clock and reset are forwarded directly to the instance of the module `topEntity` corresponding to our Clash code. The external 100 MHz clock is used as input to a `ClockWiz25` module, which is a clock manager created with the Xilinx clock wizard, configured for 25.175 MHz output. When the clock manager is locked, we start the Clash circuitry by releasing its reset pin.

```
module Wrapper (
    input        CLK100MHZ,
    output       VGA_HSYNC,
    output       VGA_VSYNC,
    output [7:0] VGA_RED,
    output [7:0] VGA_GREEN,
    output [7:0] VGA_BLUE);

    wire CLK_25MHZ;
    wire CLK_LOCKED;

    ClockWiz25 u_ClockWiz25 (
      .CLKIN_100MHZ(CLK100MHZ),
      .CLKOUT_25MHZ(CLK_25MHZ),
      .LOCKED(CLK_LOCKED));

    topEntity u_topEntity (
      .CLK_25MHZ(CLK_25MHZ),
      .RESET(!CLK_LOCKED),
      .VGA_HSYNC(VGA_HSYNC),
      .VGA_VSYNC(VGA_VSYNC),
      .VGA_RED(VGA_RED),
      .VGA_GREEN(VGA_GREEN),
      .VGA_BLUE(VGA_BLUE));
endmodule
```

We can also use these board-specific wrapper files to convert color depth. For example, suppose we're using a board that uses a small resistor ladder for the digital-to-analog conversion of the color channels, allowing only 3 bits for red and green, and 2 for blue. We can still write our Clash logic in terms of 3×8 bits, keeping it compatible with other boards that use more sophisticated DACs, and then throwing away the lower bits in the board-specific Verilog code, only keeping the highest 3 and 2 bits per channel:

```
module Wrapper (
    input        CLK100MHZ,
    output       VGA_HSYNC,
    output       VGA_VSYNC,
    output [2:0] VGA_RED,
    output [2:0] VGA_GREEN,
    output [1:0] VGA_BLUE);

    wire       CLK_25MHZ;
    wire       CLK_LOCKED;
    wire [7:0] VGA_RED_FULL;
    wire [7:0] VGA_GREEN_FULL;
    wire [7:0] VGA_BLUE_FULL;

    assign VGA_RED = VGA_RED_FULL[7:5];
    assign VGA_GREEN = VGA_GREEN_FULL[7:5];
    assign VGA_BLUE = VGA_BLUE_FULL[7:6];

    ClockWiz25 u_ClockWiz25 (
     .CLKIN_100MHZ(CLK100MHZ),
     .CLKOUT_25MHZ(CLK_25MHZ),
     .LOCKED(CLK_LOCKED));

   topEntity u_topEntity (
     .CLK_25MHZ(CLK_25MHZ),
     .RESET(!CLK_LOCKED),
     .VGA_HSYNC(VGA_HSYNC),
     .VGA_VSYNC(VGA_VSYNC),
     .VGA_RED(VGA_RED_FULL),
     .VGA_GREEN(VGA_GREEN_FULL),
     .VGA_BLUE(VGA_BLUE_FULL));
endmodule
```

7.4 Summary

- **VGA** is a video signal format heavily influenced by the theory of operation of a cathode ray tube. Understanding CRTs helps understanding VGA.

- VGA is a **serial protocol**: a full frame's worth of pixel data is shifted out at a predetermined rate (called the **pixel clock** rate), with synchronization happening on dedicated horizontal and vertical output sync lines.

- Pixel color is represented as **three analog signals**. To work with color in digital computers, we represent them as digital values of a set bit width. This width is ultimately determined by the peripheral hardware that converts the digital signal to analog.

- The key to generating a valid VGA signal is to just **keep time**. Knowing where exactly we are in the frame is what enables both the generation of the sync signals, and knowing the X and Y coordinates of the current pixel.

- The pixel clock rate for a given VGA mode (resolution and refresh rate) is standardized. We can convert the native clock rate of our FPGA board using vendor-specific **clock management** parts.

8 Generative Graphics

The previous chapter ended with a way of generating a valid VGA signal, but without any content yet. Now it is time to try our hands at displaying something more interesting than a black screen. Remembering that `vgaDriver` provides the X and Y coordinates of the current pixel, and `vgaOut` takes an RGB triplet, the job here is to compute the color for each pixel based on its coordinates.

8.1 Combinational patterns

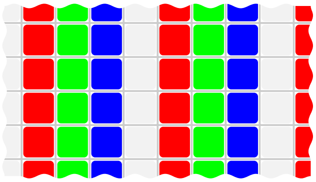
The most fundamental way of doing it is if we simply put a circuit between the coordinates and the color. For example, using a combinational circuit, we can draw red/green/blue/white stripes by looking at the bottom-most two bits of the X coordinate. The logic itself is trivial:

```
rgbwBars
    :: (KnownNat w, KnownNat h, KnownNat r, KnownNat g, KnownNat b)
    => (Index w, Index h)
    -> (Unsigned r, Unsigned g, Unsigned b)
rgbwBars (x, y) = case fromIntegral x :: Unsigned 2 of
    0 -> red
    1 -> green
    2 -> blue
    3 -> white

black = (0, 0, 0)
red = (maxBound, 0, 0)
green = (0, maxBound, 0)
blue = (0, 0, maxBound)
white = (maxBound, maxBound, maxBound)
```

As the very polymorphic type of `rgbwBars` tells us, it can be used to drive a display at any resolution and at any color depth. Let's hook it up to our standard choice of 640 × 480@60 video mode:

```
topEntity
    :: Clock Dom25
    -> Reset Dom25
    -> VGAOut Dom25 8 8 8
topEntity = withEnableGen board
  where
    board = vgaOut vgaSync rgb
      where
        VGADriver{..} = vgaDriver vga640x480at60
        xy = liftA2 (,) <$> vgaX <*> vgaY
        rgb = maybe black rgbwBars <$> xy
```

Here, the type of xy is inferred to be `Signal _ (Maybe (Index 640, Index 480))`, which drives the instantiation of `rgbw` to `(w ~ 640, h ~ 480)`. Similarly, the type of `vgaOut` constrains the color depth to `(r ~ 8, g ~ 8, b ~ 8)`.



8.2 Stateful pattern generators

Our example function `rgbwBars` is nice and simple, but perhaps a bit *too* simple. For example, suppose we wanted to draw just red/green/blue stripes instead of red/green/blue/white. Calculating the modulus of the X coordinate by a power of 2 can be done by simply dropping the lowest bits, but here we would need to calculate it by 3 – not at all easy to do in a binary circuit.

Instead, we can use an RTL circuit to run a *counter* from 0 to 2, in lockstep with the X coordinate. This means our `rgbBars` will need to be a proper signal circuit, not just a pure function:

```
rgbBars1
    :: (KnownNat w, KnownNat h, KnownNat r, KnownNat g, KnownNat b)
    => (HiddenClockResetEnable dom)
    => Signal dom (Index w, Index h)
    -> Signal dom (Unsigned r, Unsigned g, Unsigned b)
```

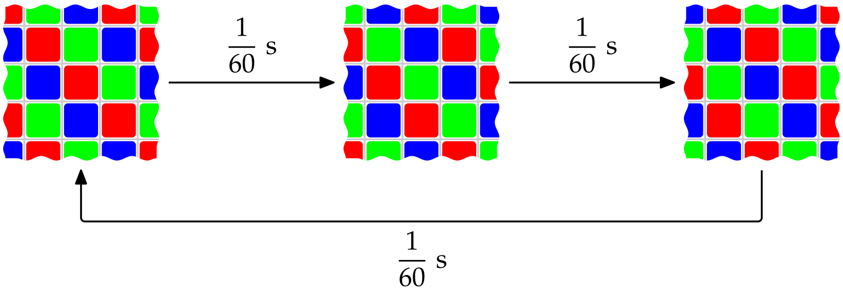
For a first try, we can simply increase an internal `Index 3` counter in every clock cycle, and use that as an index into a lookup table of colors:

```
rgbBars1 xy = colors !!. counter
  where
    counter = register (0 :: Index 3) $ nextIdx <$> counter

    colors = red :> green :> blue :> Nil
```

Hooking it up in `topEntity` is slightly different compared to the purely combinational `rgbBars`: since its input is now a `Signal` of coordinates, we have to always feed it something.

```
-- Inside topEntity
rgb =
    mux (isJust <$> xy) (rgbBars1 (fromMaybe (0,0) <$> xy)) $
    pure black
```

We're ready to try it out. However, once hooked up to a real screen, instead of nice vertical bars, we will see a checkerboard that flickers at 60 Hz:

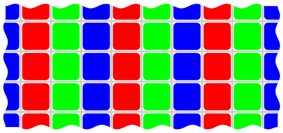


- The whole image is $800 \cdot 524 = 419,200$ clock cycles, not divisible by 3. This means every frame is drawn from a different starting state.

- Each line is 800 clock cycles, which is again not divisible by 3. Thus, every visible line is offset by 2 (the remainder of dividing 800 by 3) compared to the previous line.

We can fix both of these problems by restarting the counter at the start of each visible line. For this, we need to keep track of whether we are scanning the visible area: the corrected version of `rgbBars` gets the original, `Maybe`-wrapped coordinates, and if the X coordinate `isNothing`, resets the counter to 0. The rest of the implementation is unchanged:

```
rgbBars
    :: (KnownNat w, KnownNat h, KnownNat r, KnownNat g, KnownNat b)
    => (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Index w)
    -> Signal dom (Maybe (Index h))
    -> Signal dom (Unsigned r, Unsigned g, Unsigned b)
rgbBars x y = colors .!! counter
  where
    counter = register (0 :: Index 3) $
        mux (isNothing <$> x) (pure 0) (nextIdx <$> counter)

    colors = red :> green :> blue :> Nil
```



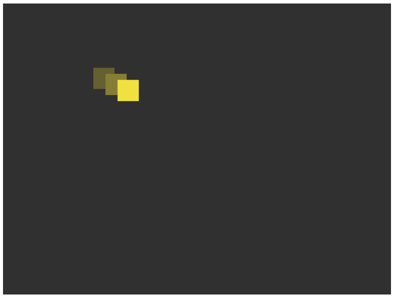
8.3 Animation

We have already, accidentally, implemented animated video in `rgbBars1`: since each frame started from a different state, the generated patterns were different frame by frame. For more controlled animation, we can keep an internal state describing the current frame, and synchronize its transition to the end of the frame.

We can detect the end of the frame simply by the vertical coordinate leaving the visible area. In the following circuit, each frame is rendered in a solid gray color, going from white to black. For simplicity's sake, we will require all three color channels to have the same depth.

```
grayAnim
    :: (KnownNat w, KnownNat h, KnownNat c)
    => (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Index w))
    -> Signal dom (Maybe (Index h))
    -> Signal dom (Unsigned c, Unsigned c, Unsigned c)
grayAnim x y = bundle (brightness, brightness, brightness)
  where
    brightness = regEn 0 endFrame $ nextIdx <$> brightness
    endFrame = isFalling False (isJust <$> y)
```

Our second animated example will show a more involved state transition: we are going to draw a "ball" (more like a square) bouncing around, trapped between the edges of the screen. Not only is this a simplified version of the bouncing DVD logo — a staple of early-2000s video players — but it should also give us inspiration for the next chapter, where we will build a fully playable Pong machine.



As before, our bouncing ball will simply be a pattern generator, i.e. a signal function from coordinates to current color:

```
type BallSize = 35

bouncingBall
    :: (KnownNat w, KnownNat h, KnownNat r, KnownNat g, KnownNat b)
    => ((BallSize + 2) <= w, (BallSize + 1) <= h)
    => (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Index w))
    -> Signal dom (Maybe (Index h))
    -> Signal dom (Unsigned c, Unsigned c, Unsigned c)
bouncingBall vgaX vgaY = draw
  where -- Continued below
```

The extra constraints on the minimal possible screen size are included because we are going to draw our ball as a 35 × 35 square, so we will need at least that much space; also, we will update the ball's position in each frame by a speed of $(2,1)$, and our calculation would break down if there is not enough slack to move the ball by that much. At this point, this might look overly pedantic, since there are no VGA video modes with low enough resolution that this would be a problem. However, later in this chapter we will use this same circuit to demonstrate coordinate transformations which can create "virtual resolutions" internal to our circuit that don't actually exist at the video output.

The key to implementing `bouncingBall` is the bouncing logic, of course. Because all reflecting surfaces (the edges of the screen) are axis-aligned, we can save ourselves a heap of trouble by decomposing the ball's movement into a horizontal component, affected only by the vertical "walls", and a vertical component, bouncing between the horizontal "walls". Thus, our state will be stored in two registers, updated at the end of each frame, each one bouncing between two endpoints:

```
frameEnd = isFalling False (isJust <$> vgaY)

(ballX, speedX) = unbundle $ regEn (0, 2) frameEnd $
    bounceBetween (0, rightWall) <$> bundle (ballX, speedX)
(ballY, speedY) = unbundle $ regEn (0, 1) frameEnd $
    bounceBetween (0, bottomWall) <$> bundle (ballY, speedY)
```

Before we give the definition of `rightWall` and `bottomWall`, we need to think about the type that we want to use for the state. For example, the X coordinate is given to us as a `Maybe (Index w)`, but it would be very painful to model all our ball dynamics using `Index w`; for example, a temporary value like `ballX + speedX` might not even be in bounds. Instead, we will do all `Index n` calculations in `Signed k` for a sufficiently large `k`. What is sufficiently large? Since `Index n` has possible values $0, 1, \ldots, n-1$, it takes up $\lceil \log_2 n \rceil$ bits. However, we want to use a signed representation for a more straightforward implementation of bouncing. Recall that `Signed k` is stored on `k` bits total, for a range of $-2^{k-1}, \ldots, 2^{k-1}-1$, so for a signed representation we need $\lceil \log_2 n \rceil + 1$ bits total:

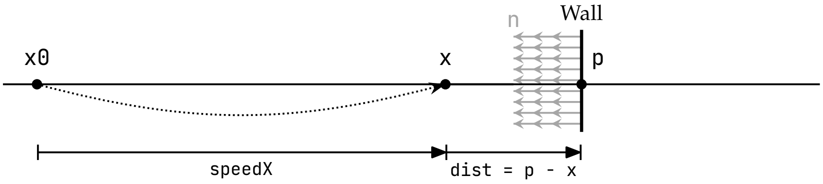
```
maxOf
    :: forall n p. (KnownNat n, 1 <= n)
    => p (Maybe (Index n)) -> Signed (CLog 2 n + 1)
maxOf _ = fromIntegral (maxBound :: Index n)

leftWall = maxOf vgaX - ballSize
bottomWall = maxOf vgaY - ballSize

ballSize :: (Num a) => a
ballSize = snatToNum (SNat @BallSize)
```

Given the current values of `ballX` and `ballY`, drawing is a simple matter of checking if `vgaX` and `vgaY` both fall within `ballSize` of them:

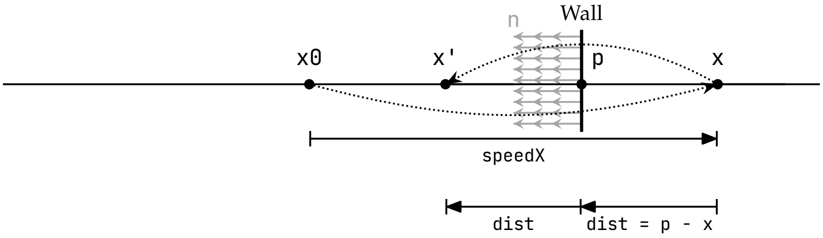
```
draw = mux isBall ballColor backColor

isBall = (near <$> ballX <*> vgaX) .&&. (near <$> ballY <*> vgaY)
  where
    near x0 = maybe False $ \(fromIntegral -> x) ->
        x0 <= x && x < (x0 + ballSize)

ballColor = pure (0xf0, 0xe0, 0x40) -- Yellow
backColor = pure (0x30, 0x30, 0x30) -- Dark gray
```

All that remains is implementing `bounceBetween` itself. There are many ways to do that; here we write a version that is based on multiple one-dimensional reflecting surfaces, which makes it easy to add additional “walls”. We will use this ability in the full-fledged Pong game in the next chapter, to only include the player’s paddle as a vertical reflection surface if it is at the same height as the ball.

Each reflecting surface is characterized with a point and a surface normal (a vector). If the ball is on the “far” side of the surface, then we need to mirror its position along the point. Remembering that we have decomposed our ball’s motion into two one-dimensional components, the normal “vector” is one-dimensional as well, so instead of computing a dot product, we can simply compare signs to determine which side the ball is. If a reflection occurs, we adjust the position and negate the speed.



(1) `x` on the outside of `p` — No reflection



(2) `x` on the inside of `p` — Reflected to `x'`

```
reflect
    :: (Num a, Num a', Ord a, Ord a')
    => (a, a')
    -> (a, a')
    -> (a, a')
reflect (p, n) (x, dx)
    | sameDirection n dist = (p + dist, negate dx)
    | otherwise = (x, dx)
  where
    sameDirection u v = compare 0 u == compare 0 v
    dist = p - x
```

When bouncing between two walls, at each time step we apply the current speed to the position, and then reflect by two surfaces facing each other.

```
move :: (Num a) => (a, a) -> (a, a)
move (x, dx) = (x + dx, dx)

bounceBetween (lo, hi) = reflect (lo, 1) . reflect (hi, -1) . move
```

Exercises:

- Hook up some input switches to `bouncingBall` to independently change the horizontal and vertical speed of the ball. Of course, the direction of motion shouldn't change when the speed changes, only its magnitude.

- Draw a fixed-width border around the screen, and have the ball bounce between them

- Flash the screen for one frame whenever a bounce occurs. This will require changing `reflect` to report whether a reflection has occurred; we will need this functionality for Pong anyway.

8.4 Coordinate transformations

As we have seen, the basic operation of our video circuit is to take the current coordinates of the electron beam as input, and produce the desired color for that pixel as output. Conversely, by putting a coordinate transformer circuit in front, we can transform the output image.

For purely combinational circuits, any coordinate transformation works seamlessly. However, the situation is not that simple for stateful circuits. For example, `rgbBars`, as written, internally keeps a counter that is updated in every clock cycle inside the visible area. We can shift its image by feeding it a `Just` value for only a subset of the real visible area, and it would produce the correct output. However, if we tried to scale its output horizontally by feeding it the same X coordinate multiple times, the counter would be incrementing for each cycle just the same: the result is the same image as without scaling.

If, instead, we change `rgbBars` to only increment its counter when the current X coordinate is different from the previous one, we get a version that can be used in a wider variety of scenarios: any transformation (or composition of transformations) resulting in a monotonically increasing (i.e. left-to-right, top-to-bottom) signal of coordinates, when connected to this new `rgbBars`, would produce an image that is consistent with the transformation.

This scalable version of rgbBars is a bit tricky to get right. First off, we can compare register Nothing x with x to find out if x has changed in the current cycle. We would think that all that remains to do is change register 0 in the definition of counter with regEn 0 newColumn, i.e. something like this:

```
scalableRGBBars1 x y = colors .!! counter
  where
    newColumn = changed Nothing x
    counter = regEn (0 :: Index 3) newColumn $
        mux (isNothing <$> x) (pure 0) (nextIdx <$> counter)
```

However, this version increments counter at the *start* of every "virtual" pixel. For example, if we scale horizontally by 4, i.e. if we use scalableRGBBars1 with an x signal that changes value every 4th cycle, then the value of counter will be as follows:

Cycle	x	newColumn	counter	Output
0	Nothing	False	0	
1	Nothing	False	0	
2	Just 0	True	0	Red
3	Just 0	False	1	Green
4	Just 0	False	1	Green
5	Just 0	False	1	Green
6	Just 1	True	1	Green
7	Just 1	False	2	Blue
8	Just 1	False	2	Blue
9	Just 1	False	2	Blue
10	Just 2	True	2	Blue
11	Just 2	False	0	Red

If we started the register at maxBound instead of 0 when the visible area starts, it would take the first increment to set it to the desired value 0. This means while counter would still be wrong (it would lag one cycle behind the value we'd like), at least its next-value-to-be would be correct:

Cycle	x	newColumn	counter	counterNext
0	Nothing	False	2	2
1	Nothing	False	2	2
2	Just 0	True	2	0
3	Just 0	False	0	0
4	Just 0	False	0	0

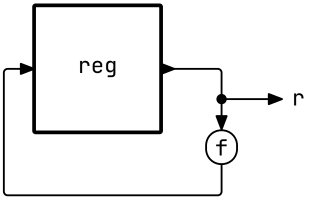
Cycle	x	newColumn	counter	counterNext
5	Just 0	False	0	0
6	Just 1	True	0	1
7	Just 1	False	1	1
8	Just 1	False	1	1
9	Just 1	False	1	1
10	Just 2	True	1	2
11	Just 2	False	2	2

Whenever we have a `register` whose value is lagging one cycle behind, we can solve that by tapping into its new value instead of the value propagated from the previous cycle. In general, this can be done by rewriting code of this form:

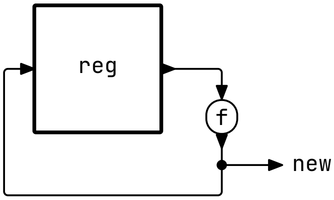
```
someCircuit1 = ... r ...
  where
    r = register x0 $ f r
```

into this:

```
someCircuit2 = ... new ...
  where
    r = register x0 new
    new = f r
```



(1) `someCircuit1`



(2) `someCircuit2`

This transformation, together with starting from `maxBound`, leaves us with the following version. Note that we can't use `regEn` here, because we need the new value `counterNext` to be already gated on `newColumn`. Thus the first `mux` in the definition of `counterNext`.

```
scalableRGBBars x y = colors .!! counterNext
  where
    newColumn = changed Nothing x
    counter = register (0 :: Index 3) counterNext
    counterNext =
        mux (not <$> newColumn) counter $
        mux (isNothing <$> x) (pure maxBound) $
        nextIdx <$> counter
```

All this is not to say that the original `rgbBars` was "wrong". Video pattern generators need to be designed for specific use cases, and if we are building a circuit that will generate video at the native resolution, it is perfectly fine to build a video subsystem that exploits this fact. Moreover, supporting coordinate transformations is not an all-or-nothing deal, but a spectrum – for example, as we have seen, `rgbBars` works as-is for horizontal or vertical translations, but required some changes to work for rescaling. Here, we present these two generally useful coordinate transformation circuits: translation and scaling.

8.4.1 Restricting the visible area

We can restrict the physical visible area of $w \times h$ to a smaller $w' \times h'$ by keeping the values of the virtual X and Y coordinate signals at `Nothing` for some of the time that the real X and Y coordinates are already `Just` values. The generic form of this transformation masks out parts of the visible area on both sides:

```
maskSides
    :: (KnownNat n, KnownNat m, KnownNat k)
    => (HiddenClockResetEnable dom)
    => SNat k
    -> Signal dom (Maybe (Index (k + n + m)))
    -> Signal dom (Maybe (Index n))
maskSides k raw = transformed
  where -- Continued below
```

We implement `maskSides` by starting a `Maybe (Index n)` counter whenever the `raw` input equals `Just k`. To make `maskSides` compose nicely with other transformers, we also implement the same logic as `scalableRGBBars` to only increment the counter whenever the `raw` input `changed`:

```
    changed = register Nothing raw ./=. raw
    started = raw .== Just (snatToNum k)

    r = register Nothing transformed
    transformed =
        mux (not <$> changed) r $
        mux (isNothing <$> raw) (pure Nothing) $
        mux started (pure $ Just 0) $
        (succIdx =<<) <$> r
```

We can use `maskSides` to define simpler combinators that mask out only the `k` pixels in the beginning or the end of the visible area. We write `k + n` and `n + k` in the type signature to be evocative of which side we're masking out:

```
maskStart
    :: forall k n dom. (KnownNat n, KnownNat k)
    => (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Index (k + n)))
    -> Signal dom (Maybe (Index n))
maskStart = maskSides (SNat @k)

maskEnd
    :: forall k n dom. (KnownNat n, KnownNat k)
    => (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Index (n + k)))
    -> Signal dom (Maybe (Index n))
maskEnd = maskSides (SNat @0)
```

With a bit more type-level arithmetic, we can also `center` a smaller image on a larger one: here, the `k ~ ((n0 - n) `Div` 2` constraint is effectively a type-level `let` binding for the size of the padding needed (rounded down). Because of rounding, we don't, in general, have `n0 ~ k + n + k`; instead, we set `m` to be the remaining visible area. The Clash typechecker for arithmetic constraints is powerful enough to solve `n0 ~ (k + n + m)`. In fact, we've already used the solver's power in the definition of `maskSides`, where the size `m` of the trailing side is inferred from knowing `k`, `n` and `k + n + m`. [1]

```
center
    :: forall n n0 k m dom. (KnownNat n, KnownNat n0, KnownNat k,
    KnownNat m)
    => (k ~ ((n0 - n) `Div` 2), n0 ~ (k + n + m))
    => (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Index n0))
    -> Signal dom (Maybe (Index n))
center = maskSides (SNat @k)
```

[1] Clash automatically enables a handful of typechecker plugins implementing painless `KnownNat` propagation and arithmetic solvers for type-level naturals. These plugins can also be used with GHC proper, by adding `GHC.TypeLits.KnownNat.Solver`, `GHC.TypeLits.Extra.Solver`, and `GHC.TypeLits.Normalise` to the list of typechecker plugins loaded.

8.4.2 Scaling

Scaling is very similar to a generalized version of `scalableRGBBars`: we keep an internal counter that is incremented for every changed raw coordinate, and every time we would "start drawing a new red bar", the output coordinate is incremented. We can keep doing this until the output coordinate hits its maximum value, after which we reset it to zero the next time we enter the visible area. We also return the "sub-pixel" coordinate for visible pixels.

```
scale
    :: forall n k dom. (KnownNat n, KnownNat k, 1 <= k)
    => (HiddenClockResetEnable dom)
    => SNat k
    -> Signal dom (Maybe (Index (n * k)))
    -> (Signal dom (Maybe (Index n)), Signal dom (Maybe (Index k)))
scale k raw = (scaledNext, enable (isJust <$> scaledNext) counterNext)
  where
    prev = register Nothing raw
    changed = raw ./=. prev

    counter = register 0 counterNext
    counterNext =
        mux (not <$> changed) counter $
        mux (isNothing <$> prev) (pure 0) $
        nextIdx <$> counter

    scaled = register Nothing scaledNext
    scaledNext =
        mux (not <$> changed) scaled $
        mux (counterNext .== 0) (maybe (Just 0) succIdx <$> scaled) $
        scaled
```

The order of type variables is carefully chosen so that we can write `scale @n` in cases where the input and the output types wouldn't be constrained otherwise; for example, this allows us to write `scale @n (SNat @k) . center` to transform n to $m \geq k * n$ by making each pixel n times larger, and then sufficient padding on both sides.

Exercises:

- Write a 2D version of `rgbBars`, i.e. something that shows 9 different colors in a repeating, 3 × 3 tile. Hint: the vertical divide-by-3 counter should only be incremented at the end of each scanline.

- Combine two or more pattern generators by showing them in different parts of the screen. A simplest version would be showing `rgbwBars` on one half and `rgbBars` on the other half; for a more interesting challenge, try to define multiple window-like rectangles on the screen, each showing a different pattern.

- Change the bouncing ball circuit to internally use a resolution of 300 × 200, and render that to a 640 × 480 VGA mode by scaling up by two and centering horizontally and vertically.

- The above change leaves a border 20 pixels wide on both sides and 40 pixels high on the top and the bottom. Render these areas in some distinctive color, without changing anything in the definition of `bouncingBalls`.

8.5 Animation, differently

One drawback of writing `bouncingBalls` in the above style is that the full circuit is described as a monolithic function mapping signals to signals; as a consequence, it can be arbitrarily stateful. In this section, we rewrite it in a more principled way that will lend itself to high-level simulation.

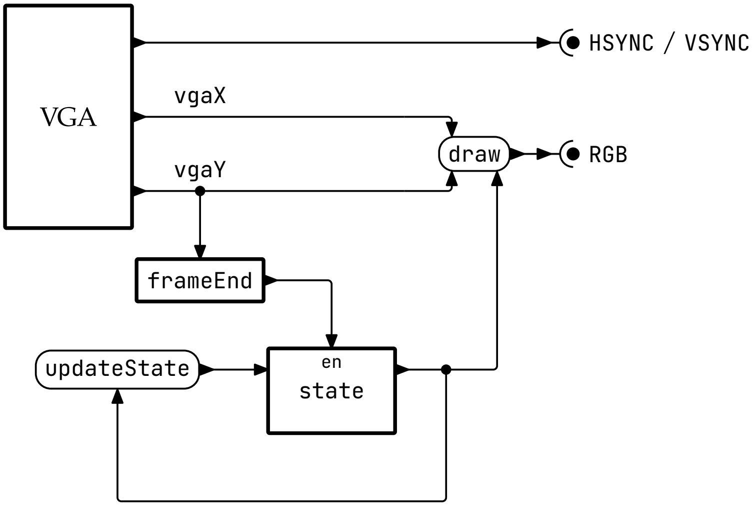
The basic idea behind the restructuring is to split `bouncingBalls` into two functions: a **state transition function** (the "bouncing") and a **drawing function** (the "balls"). Both of these are **pure** functions, with the state explicitly passed between them by the top-level circuit via a register. This is similar to the calculator from chapter 6.

In the calculator circuit, the state transition was triggered by user input. Animation, however, happens in real-time: the ball continues bouncing around on its own. We will use the start of the vertical blanking period to trigger for the state transition once for each frame, 60 times a second. This avoids potential graphical glitches that could occur if, for example, the ball position would be updated just at the same time as we are drawing its current position.

Moreover, although not a concern for our bouncing ball toy, for a more complicated machine it might take several cycles to compute a full state update; by starting just at the beginning of the vertical blanking period, we give us the most cycles possible before starting to draw the next frame.[2] For this reason, arcade machines and home computers generally have some way of signaling from the video subsystem to the CPU when the current frame is finished. In this computer-less circuit, we will simply consume the Y coordinate output of the video controller directly.

[2] Technically, we could start even a bit earlier, when the horizontal blanking period of the last line starts. The reason we aim for the start of the vertical blanking is to make the trigger logic simpler.

Putting it all together, our design will be as follows:



- A VGA controller generates the sync outputs and provides the rest of the circuit with the coordinates of the currently drawn pixel

- The animation state is stored in a register that is updated whenever the currently drawn frame ends.

- The drawing circuit takes the current state and the current video coordinates. By comparing the video coordinate to the ball's current position, it calculates the object at the current coordinate, which determines the color of the currently drawn pixel.

Since we have already written a VGA controller, the only parts we need to write are the following definitions:

- `data St`, the animation state.
- `updateState :: St -> St`, the state transition function.
- `draw :: St -> Index 640 -> Index 480 -> (Unsigned r, Unsigned g, Unsigned b)`, the drawing function.

As with the calculator, none of these parts use Clash `Signals` in their interface. We will exploit this property by assembling the same parts into a software implementation alongside the hardware one.

8.5.1 Animation state

Our only state is the ball position, stored in two pairs of `Coords` to represent the ball's position and speed horizontally and vertically. We generate lenses here which we will use when writing `updateState`. The size of `Coord` is chosen to fit not only `ScreenWidth` and `ScreenHeight`, but also the intermediate calculations during `reflect`.

```
type Coord = Signed 10

data St = MkSt
    { _ballH, _ballV :: (Coord, Coord)
    }
    deriving (Show, Generic, NFDataX)
makeLenses ''St

initState :: St
initState = MkSt
    { _ballH = (10, 2)
    , _ballV = (100, 3)
    }
```

To future-proof our code somewhat, we also create a datatype for the animation's parameters; in this case, the only parameter is the ball size. This will be useful for one of the exercises later on.

```
data Params = MkParams
    { ballSize :: Coord
    }
    deriving (Show, Generic, NFDataX)

defaultParams :: Params
defaultParams = MkParams
    { ballSize = 35
    }
```

Since the bouncing ball is autonomous, there is no user input to `updateState`. We write it using the `State` monad so that we can compose `bounceBetween` for the vertical and the horizontal axis by `zoom`ing on the relevant fields of `St`. This is, arguably, overkill compared to something like `\(St x y) -> St (bounceBetween (0, w) x) (bounceBetween (0, h) y)`. However, when we move on to implementing more complex circuits, like a full-blown Pong game in the next chapter, we will need this flexibility for some of the stateful calculations like detecting the collision between the ball and the paddle. Note that we subtract `ballSize` from the higher boundaries (bottom and right), since the coordinates stored in the state represent the ball's top-left corner.

```
updateState :: Params -> St -> St
updateState params@MkParams{..} = execState $ do
    zoom ballV $ modify $ bounceBetween (0, screenHeight - ballSize)
    zoom ballH $ modify $ bounceBetween (0, screenWidth - ballSize)
```

Although we could make the screen size a parameter, here we go a different route to ensure that the ball stays exactly in the visible area: `screenWidth` and `screenHeight` are reflected from type-level constants which we will also use in the type of `draw`:

```
type ScreenWidth = 640
type ScreenHeight = 480

screenWidth :: Coord
screenWidth = snatToNum (SNat @ScreenWidth)

screenHeight :: Coord
screenHeight = snatToNum (SNat @ScreenHeight)
```

8.5.2 Drawing

We implement drawing by writing a pure function that is only concerned with visible pixels. For the color channels, although the final output is limited in depth by the targeted hardware platform, here we use `Word8` for each channel for 24-bit colors. This allows us to specify the colors we'd like, not just the ones we can have; the latter can be derived trivially in `topEntity` by just truncating the lowermost bits. However, using `Word8` will make simulation performance dramatically better. That is because conversions like `bitCoerce :: Unsigned 8 -> Word8`, while a no-op in a real hardware circuit, involves a significant simulation overhead, so we are better off if we can avoid it three times for each pixel during simulation. By defining `Color` this way, we only have coercions in the other direction, `bitCoerce :: Word8 -> Unsigned 8`, in `topEntity` which is outside the context of our high-level simulation.

```
type Color = (Word8, Word8, Word8)

draw :: Params -> St -> Index ScreenWidth -> Index ScreenHeight -> Color
draw MkParams{..} MkSt{..} ix iy
    | isBall    = yellow
    | otherwise = gray
  where -- Continued below
```

This leaves us with only the problem of calculating if a given (ix, iy) coordinate is within the area where we want the ball to be visible. Later, more complete versions of draw will have exactly the same structure, just with more branches for isWall and isPaddle.

The workhorse function of isBall and similar definitions is determining if the current pixel is within a given axis-aligned rectangle. But first, we convert ix and iy to Coords to be compatible with St's fields.

```
    x = fromIntegral ix
    y = fromIntegral iy

    z `between` (lo, hi) = lo <= z && z <= hi
    rect (x0, y0) (w, h) =
        x `between` (x0, x0 + w) &&
        y `between` (y0, y0 + h)
```

Given these definitions, we can write isBall simply by checking if the current pixel is in a ballSize × ballSize rectangle starting at the ball position:

```
    (ballX, _) = _ballH
    (ballY, _) = _ballV

    isBall = rect (ballX, ballY) (ballSize, ballSize)
```

8.5.3 The top-level circuit

Let's take stock of the components we have written so far:

- data St is our state, which we want to keep in a register.
- updateState is the state transition function, which should be used to update the register at each frameEnd
- draw is the drawing function which calculates the color of the currently rendered, visible pixel

Components we need to assemble it into a full circuit:

- A VGA controller to generate the sync signals and to keep track of which pixel is currently rendered, if any.
- A signal frameEnd :: Signal _ Bool that fires at the end of each frame, to trigger the state register's update. This can be implemented by checking the Y coordinate output of the VGA controller ceasing to be isJust, since that means we have left the last visible line.

With these considerations, and using a 25.175 MHz clock as before for our chosen VGA mode, the full top-level circuit is as follows:

```
createDomain vSystem{vName="Dom25", vPeriod = hzToPeriod 25_175_000}

topEntity
    :: "CLK_25MHZ" ::: Clock Dom25
    -> "RESET"     ::: Reset Dom25
    -> "VGA"       ::: VGAOut Dom25 8 8 8
topEntity = withEnableGen $ vgaOut vgaSync rgb
  where
    VGADriver{..} = vgaDriver vga640x480at60
    frameEnd = isFalling False (isJust <$> vgaY)

    st = regEn initState frameEnd $ updateState defaultParams <$> st

    rgb = fmap (maybe (0, 0, 0) bitCoerce) $
        liftA2 <$> (draw defaultParams <$> st) <*> vgaX <*> vgaY
```

Since all our changes in this section so far were just shuffling around parts of the previous bouncing ball circuit, it shouldn't be a surprise that compiling, synthesizing, and uploading this circuit to an FPGA will result in the same video output as before. So, what was the point?

8.6 High-level simulation with SDL2

Restructuring our circuit into two separate, pure functions, one for the state transition and the other to implement drawing, pays dividends when we want to write a simulator for it. Just like in the calculator project, here we can exploit the structure of our code by assembling `updateState` and `draw` differently, into a sequential, stateful program that does the following:

0. Create a window with a backing texture of size 640 × 480.
1. Poll for any potential user input; exit if the window is closed.
2. Pass the current state and all possible coordinate pairs $(0, 0), (0, 1), \ldots, (639, 479)$ to `draw` and record its output into the texture.
3. Apply `updateState` to the state and the interpretation of the input events.
4. If we have some time left out of the $1/60$ of a second frame time, sleep for the remainder.
5. Repeat from step 1.

We are going to use *SDL2* to take care of the nitty-gritty of opening windows and polling keyboard events in whatever operating system we use. SDL2 is a mature cross-platform library with good Haskell bindings that don't get in the way.

8.6.1 Hello, SDL2!

Before jumping head-first into connecting `draw` and `updateState` to SDL2, let's write a standalone program that uses SDL2 to open a window, draw some pixels, and waits for a keypress before shutting down. This will show us all we need to know about SDL2, and the rest will be up to us.

```haskell
import SDL
import Data.Word

-- These are needed since textures are accessed through pointers
import Foreign.Ptr
import Foreign.Storable
```

We start `main` by initializing SDL and creating a `window`. The window size is scaled from the intended virtual screen size; we use a screen size with a resolution of just 24 × 18 so that we can easily see the result of drawing a single pixel. The window itself will have size 720 × 540.

```haskell
screenSize = V2 24 18
screenScale = 30

main :: IO ()
main = do
    initializeAll
    window <- createWindow "Hello, SDL2!" defaultWindow
    windowSize window $= (screenScale *) <$> screenSize
    withTexture <- setupTexture window

    forever $ withTexture drawHello
  where -- Continued below
```

In `setupTexture`, we attach a `renderer` and a `texture` to the `window`. We want an efficient way of drawing individual pixels, and that is exactly what the texture gives us. Most of the parameters here are not important for us, and we use some sensible default values. As we will see, the `RGB888` texture format is perhaps not exactly what its name suggests, but still the closest to our needs. The `TextureAccessStreaming` argument ensures we can get direct read-write access to the pixel data underlying the texture.

Given this `renderer` and `texture`, we can ask SDL for a raw texture pointer and manipulate pixels through it using the `lockTexture` / `unlockTexture` API. The callback function `drawTo` is given a `Ptr ()` to the beginning of the texture, and an

Int which is the rowstride, i.e. the pointer difference (in bytes) between the location of two pixels that are vertical neighbors.

```
    setupTexture window = do
        renderer <- createRenderer window (-1) defaultRenderer
        texture <- createTexture renderer RGB888 TextureAccessStreaming
    screenSize

        return $ \drawToTexture -> do
            (ptr, stride) <- lockTexture texture Nothing
            drawToTexture ptr (fromIntegral stride)
            unlockTexture texture
            SDL.copy renderer texture Nothing Nothing
            present renderer
```

Now we can get creative in drawHello. This is the point where we need to understand the RGB888 texture format. The name would suggest that it uses three bytes per pixel, storing red, green, blue, then next pixel's red, and so on. Instead, each pixel is stored in *four* bytes, in machine byte order, with the fourth one unused. Accordingly, on a little-endian machine, we can write a version of drawHello that sets a single pixel at $(10, 5)$ to red by accessing each color byte separately in reverse (blue-green-red) order:

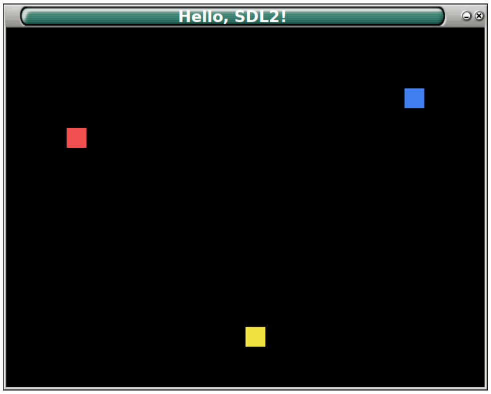
```
drawHello :: Ptr () -> Int -> IO ()
drawHello ptr rowstride = do
    pokeElemOff rowptr (x * 4 + 0) b
    pokeElemOff rowptr (x * 4 + 1) g
    pokeElemOff rowptr (x * 4 + 2) r
  where
    (x, y) = (10, 5)
    (r, g, b) = (0xf0, 0x50, 0x50) :: (Word8, Word8, Word8)
    rowptr = plusPtr ptr $ rowstride * y
```

This works, but it depends on the machine endiannness and in general just *feels awkward*. The reason for that is we are going against the grain here: SDL's intended texture access is via 32-bit values. If we change drawHello to write the color as a single Word32, not only does the code become cleaner, it will also have better performance once we move to changing more pixels than just one.

```
drawHello :: Ptr () -> Int -> IO ()
drawHello ptr rowstride = forM_ points $ \((x, y), rgb) -> do
    let rowptr = plusPtr ptr $ rowstride * y
    pokeElemOff rowptr x (rgb :: Word32)
  where
```

```
    points =
        [ ((3,5),   red)
        , ((12,15), yellow)
        , ((20, 3), blue)
        ]

    red    = 0xf0_50_50
    yellow = 0xf0_e0_40
    blue   = 0x40_80_f0
```

We can now run our program and marvel at the window showing our abstract art:



However, there is no nice way to exit our program; the only thing we can do is kill its process. This is because our main loop runs `forever`, with no way to exit. Instead of running it in `IO`, we will run it in `MaybeT IO`, so that we can exit the `forever` loop at any time by calling `mzero`. Luckily, SDL's API is already polymorphic over the base monad, so we don't have to wrap everything in `liftIO` calls.

Let's replace the main loop with something that only runs until we get an event from SDL that should prompt us to quit:

```
    runMaybeT $ forever $ do
        events <- pollEvents
        keyDown <- getKeyboardState
        let shouldQuit =
                any isWindowCloseEvent events ||
                keyDown ScancodeEscape
        guard $ not shouldQuit
        liftIO $ withTexture drawHello32
```

Here, shouldQuit will be set to True if any of the latest events is a notification that the main window was closed, or if the Esc key is pressed.

```
isWindowCloseEvent ev = case eventPayload ev of
    WindowClosedEvent{} -> True
    _ -> False
```

This concludes our introduction to SDL2: we are now ready to connect our bouncing ball functions to texture drawing routines.

8.6.2 A reusable SDL2 simulator framework

Here we tweak our Hello World example to factor out the parts that will need to depend on the particulars of the simulated design:

- How the internal state is managed
- How input events [Event] and key states Scancode -> Bool are processed
- How the screen texture is updated

We will abstract over internal state management by allowing the simulation to happen in any MonadIO m. Input events and key states are going to be passed as simple function parameters.

For the screen texture update, we want to allow versatility to get good performance based on the access pattern (i.e. we want to expose the underlying texture directly), but also use types to track the screen size to rule out malformed indexing. We achieve this by making an abstract type Rasterizer indexed by the screen dimensions, and providing a library of trusted Rasterizer values for various use cases.

```
newtype Rasterizer (w :: Nat) (h :: Nat) = Rasterizer
    { runRasterizer :: Ptr () -> Int -> IO () }
```

Armed with this type, we can write a function that wraps the simulator in an environment where we keep running it as long as it returns the rasterizer for the current frame. This allows the simulator to decide to quit on its own. The only “free” parameters we have left are the window title and the screen scaling factor, since these are not inferrable from the types.

```
data VideoParams = MkVideoParams
    { windowTitle :: Text
    , screenScale :: CInt
    , screenRefreshRate :: Word32
    }
```

```
withMainWindow
    :: forall w h m. (KnownNat w, KnownNat h, MonadIO m)
    => VideoParams
    -> ([Event] -> (Scancode -> Bool) -> MaybeT m (Rasterizer w h))
    -> m ()
```

The implementation of `withMainWindow` is very similar to the Hello World program; we simply pass to `withTexture` a drawing function that runs the `Rasterizer` returned by the simulation.

```
withMainWindow MkVideoParams{..} runFrame = do
    initializeAll
    window <- createWindow windowTitle defaultWindow
    windowSize window $= fmap (screenScale *) screenSize

    withTexture <- setupTexture window
    let render rasterizer = withTexture $ \ptr rowstride ->
            liftIO $ runRasterizer rasterizer ptr rowstride

    runMaybeT $ forever $ do
        events <- pollEvents
        keyDown <- getKeyboardState
        let windowClosed = any isWindowCloseEvent events
        guard $ not windowClosed
        rasterizer <- runFrame events keyDown
        render rasterizer
    destroyWindow window
  where
    screenSize = V2 (snatToNum (SNat @w)) (snatToNum (SNat @h))

    setupTexture window = ... -- as before
```

Once the main loop finishes (because `windowClosed` or the simulation exits with `mzero`), we clean up the `window` by calling `destroyWindow`; this wasn't needed for our stand-alone Hello World program, since getting out of the the `forever` finishes the whole process anyway, cleaning everything up; but here we are building a library, so we have no control over whether clients will want to do other stuff after `withMainWindow` finishes.

There is nothing yet in our code that would lock it to 60 frames per second (or whatever else the refresh rate of a given circuit is). There's nothing we can easily do about it if our software simulation takes *more* than $1/60^{th}$ of a second, but if `runFrame` and `render`, taken together, take *less* time to run than the intended frame time, we can just sleep for the remaining frame time. To this end, we add the combinator atFrameRate, which record the time (as measured, in milliseconds, by SDL's tick function) before and after the computation for a given frame. waitFrame then converts both the frame rate and the before/after time stamp into microseconds, suitable for threadDelay.

```
atFrameRate :: (MonadIO m) => Int -> m a -> m a
atFrameRate frameRate act = do
    before <- ticks
    x <- act
    after <- ticks
    waitFrame frameRate before after
    return x

waitFrame :: (MonadIO m) => Int -> Word32 -> Word32 -> m ()
waitFrame frameRate before after = when (slack > 0) $ liftIO $
    threadDelay slack
  where
    frameTime = 1_000_000 `div` frameRate
    elapsed = fromIntegral $ 1000 * (after - before)
    slack = frameTime - elapsed
```

Armed with atFrameRate, we simply replace our main loop's forever $ do ... with forever $ atFrameRate screenRefreshRate $ do ....

We conclude our reusable simulator by writing a Rasterizer for combinational pattern generators: we iterate y through all possible values of Index h, calculate the pointer for each row using the row stride, and then poke the result of computing the pattern's color value starting from that pointer, 32-bit value by 32-bit value:

```
{-# INLINE packColor #-}
packColor :: Color -> Word32
packColor (r, g, b) =
    fromIntegral r `shiftL` 16 .|.
    fromIntegral g `shiftL` 8 .|.
    fromIntegral b `shiftL` 0

rasterizePattern
    :: (KnownNat w, KnownNat h)
    => (Index w -> Index h -> Color)
    -> Rasterizer w h
rasterizePattern draw = Rasterizer $ \ptr rowstride -> do
    for_ [minBound..maxBound] $ \y -> do
        let rowPtr = plusPtr ptr $ fromIntegral y * rowstride
        for_ [minBound .. maxBound] $ \x -> do
            pokeElemOff rowPtr (fromIntegral x) (packColor $ draw x y)
```

8.6.3 Let's see some bouncing balls finally!

Now that we've built up the infrastructure, it is time for the payoff: running our circuit design's `updateState` and `draw` functions and seeing their results on our screen in real time.

The idea is to pick `StateT St IO` as the monad we pass to `withMainWindow`. This takes care of holding on to the state from one frame to the next. Since `withMainWindow` wraps it in a `MaybeT`, we also have access to the effect of early termination, which we can use to implement a custom exit command. In this example, we will use the Esc key as an exit trigger.

To get back to vanilla `IO` for `main`, we simply use `evalStateT` to run the `StateT St IO ()` returned by `withMainWindow`. We have arranged the types of `updateState` and `draw` to minimize impedance mismatch with the `StateT` combinators and with `rasterizePattern`:

```
main :: IO ()
main =
    flip evalStateT initState $
    withMainWindow videoParams $ \events keyDown -> do
        guard $ not $ keyDown ScancodeEscape

        modify $ updateState defaultParams
        gets $ rasterizePattern . draw defaultParams
  where
    videoParams = MkVideoParams
        { windowTitle = "Bouncing Ball"
        , screenScale = 2
        , screenRefreshRate = 60
        }
```

And that's it!

Exercises:

- Tweakable parameters. Instead of passing the `defaultParams` to `updateState` and `draw`, change it into a proper signal. Connect some toggle switches as input to change the ball size mid-game.

- Extend the SDL simulator to generate the toggle switch state from keyboard events. For example, hook up the number keys 1 to 8 to each flip one virtual toggle switch.

- Similar to the earlier exercise, change this new version of the bouncing balls circuit to run in a virtual resolution of 300 × 200 pixels, scaled by two and centered. The software simulation should render into a 300 × 200 window, and the hardware circuit will need to handle `Nothing` coordinates by drawing some nice border / background, passing `Just` the valid virtual coordinates to `draw`.

8.7 Summary

- A video pattern generator is a **potentially stateful circuit mapping coordinates to colors**. By default, the video controller's coordinate output is connected directly to the pattern generator's input, but we can put **coordinate transformers** between them.
- Just like in the calculator project, if we structure our design around a **state transition function** and a **pure output function**, this allows us to create a **high-level simulation** by hooking into the "interesting" parts of our design.
- For interactive, real-time designs with video output, such as video games, **sampling input and updating the state once per frame** is a natural and easily-implemented solution.
- The **SDL2** library provides an easy, robust and performant way of rendering pixel-based graphics, which can be used to **simulate video output in real time**.

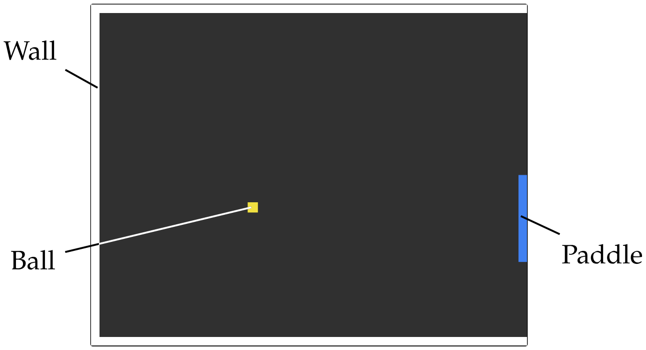
9 Project: Pong

In this chapter, we answer the call of the bouncing ball example circuit from the previous chapter, and build our own version of Pong, one of the earliest video games.

The original Pong didn't run on a computer: instead, its game logic and its video output was all implemented directly as a circuit of discrete components. Our version will also be computer-less; however, rather than building a rat's nest of connected registers, we will apply the same principled design as we did in the Calculator project.

9.1 What is Pong?

Pong is a very minimalistic video game simulation of tennis. Players control paddles on the sides of the screen, by moving them vertically. A ball is bouncing between the paddles and the top and bottom edges of the screen. The aim of the player is to not let the ball go out of bounds on their side. In the original two-player version, there are two paddles, one for each player. Here, we will build a solitaire version first, and leave the two-player version as an exercise. Basically, our game is going to be the squash equivalent to Pong's tennis.

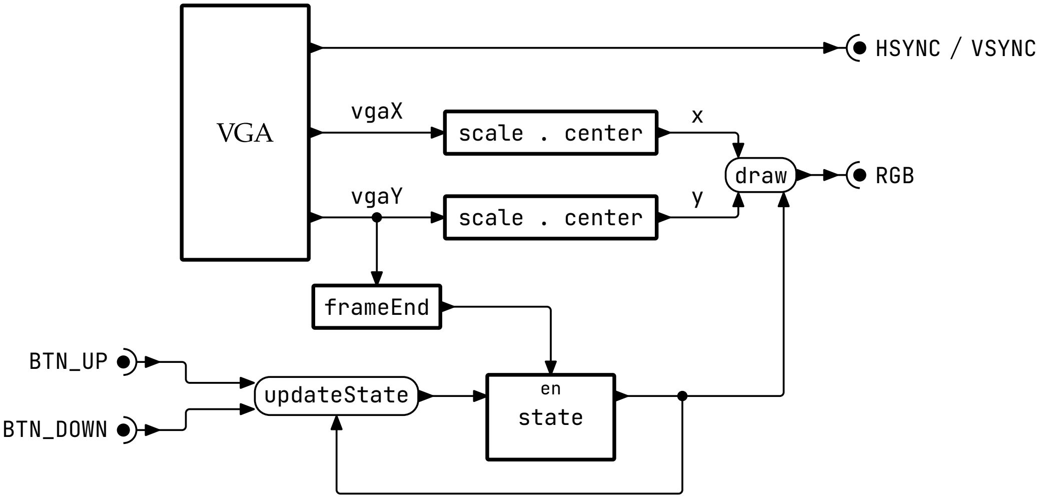


From the outside point of view, Pong is a circuit which outputs a video signal and connects to inputs for paddle control. We will use two pushbuttons for moving the paddle up or down, and generate VGA in 640 × 480@60 mode for the video output. Just for the fun of it, and to give it that nice chunky retro look, the game itself will only use 256 × 200 resolution, which we will scale up by two and centered it on the screen.

9.2 Top-level design

Internally, we will follow the same design as the bouncing ball toy: a register holding the state, a state transition function consuming input, and a drawing function that takes the current state, and turns it into output.

Putting it all together, our design will be as follows:



This is the same design as the bouncing ball one, with an added coordinate transformation to end up with a drawing area of 256 × 200, and input from the outside world in the form of the two pushbuttons controlling the paddles.

With all this groundwork laid, it is time to work out the missing details:

- `data Inputs`, a record that holds all user inputs
- `data St`, the game state
- `updateState :: Inputs -> St -> St`, the state transition function
- `draw :: St -> Index 256 -> Index 200 -> Color`, the drawing function

Once these are filled in, our `topEntity` will mostly match that of the bouncing ball circuit:

```
data Inputs = MkInputs
    { paddleUp :: Bool
    , paddleDown :: Bool
    }

type ScreenWidth = 256
type ScreenHeight = 200

topEntity
    :: "CLK_25MHZ" ::: Clock Dom25
    -> "RESET"     ::: Reset Dom25
    -> "BTN_UP"    ::: Signal Dom25 (Active High)
    -> "BTN_DOWN"  ::: Signal Dom25 (Active High)
    -> "VGA"       ::: VGAOut Dom25 8 8 8
topEntity = withEnableGen board
  where
    board (fmap fromActive -> up) (fmap fromActive -> down) = vgaOut
    vgaSync rgb
      where
        VGADriver{..} = vgaDriver vga640x480at60
        frameEnd = isFalling False (isJust <$> vgaY)

        params = defaultParams
        inputs = MkInputs <$> up <*> down

        st = regEn initState frameEnd $
            updateState params <$> inputs <*> st

        rgb = fmap (maybe (0, 0, 0) bitCoerce) $
            liftA2 <$> (draw params <$> st) <*> x <*> y
          where
            (x, _) = scale (SNat @2) . center $ vgaX
            (y, _) = scale (SNat @2) . center $ vgaY
```

Accordingly, the SDL-based simulator's `main` function also remains mostly the same. We increase the scaling factor to 4, since the virtual screen we are simulating now is only 256 × 200 pixels – remember, the transformation to 640 × 480 takes place only in `topEntity`, as a measure to convert to a standard video format that real-world screens can understand. This conversion is morally no different from converting the `Active High` pushbutton values to a semantic `Bool`, so we put it outside `draw`.

```
main :: IO ()
main =
    flip evalStateT initState $
    withMainWindow videoParams $ \events keyDown -> do
        guard $ not $ keyDown ScancodeEscape

        let params = defaultParams
        modify $ updateState params $ MkInputs
            { paddleUp = keyDown ScancodeUp
            , paddleDown = keyDown ScancodeDown
            }
        gets $ rasterizePattern . draw params
  where
    videoParams = MkVideoParams
        { windowTitle = "Pong"
        , screenScale = 4
        , screenRefreshRate = 60
        }
```

To recap, the signatures of the remaining parts to implement are:

```
data St

updateState :: Params -> Inputs -> St -> St
draw :: Params -> St -> Index ScreenWidth -> Index ScreenHeight -> Color
```

9.3 What is our state?

Compared to just a ball bouncing around in the emptiness of a video screen, a game of Pong is similar in some ways and different in others:

- There is a ball bouncing around in both cases. Our `State` will need to hold the ball's position and speed just like before.
- Pong has another moving part: the paddle. Since it can only be moved vertically, only its Y position needs to be stored.
- We want to draw the ball as before, but of course we also want to draw the paddle (at the right position). To emphasize to the player that it is their responsibility to hit the ball back from the right-hand edge, we will also draw walls around the other three edges of the screen.
- Pong is interactive: the player can move the paddle up or down. We take care of this by adding an extra `Inputs` parameter to `updateState`.

- There is also some complicated interaction between the ball and the paddle. At the minimum, the ball bounces off the paddle when it hits it; but that alone makes for a very boring variant of Pong. We will spice it up a notch by allowing the paddle to nudge the ball vertically, if the paddle itself is moving vertically at the moment of contact.

- Pong is a game with a goal: to avoid the ball leaving the playfield by flying off to the right. We should give some kind of indication of failure when that happens: we will flash the background color in red for one frame.

Based on this analysis, it is clear that `St` should extend the horizontal and vertical speed-and-position of the ball with a vertical paddle position, and a flag denoting if we should draw the background in the given frame in red. `updateFlag` will always clear that flag (unless, of course, the ball is just now leaving the game area); this ensures that it will flash for one frame only.

```
type Coord = Signed 10

data St = MkSt
    { _ballH, _ballV :: (Coord, Coord)
    , _paddleY :: Coord
    , _gameOver :: Bool
    }
    deriving (Show, Generic, NFDataX)
makeLenses ''St

initState :: St
initState = MkSt
    { _ballH = (10, 2)
    , _ballV = (100, 3)
    , _paddleY = 100
    , _gameOver = False
    }
```

The `Params` datatype likewise extends the ball size with new fields for the width of the walls and the size of the paddle. We also need to know how much to move the paddle on each frame if the user is holding one of the input buttons, and how much nudge should be applied to the ball when hitting it with a moving paddle.

```
data Params = MkParams
    { wallSize, ballSize :: Coord
    , paddleHeight, paddleWidth :: Coord
    , paddleSpeed, nudgeSpeed :: Coord
    }
```

```
defaultParams :: Params
defaultParams = MkParams
    { wallSize = 5
    , ballSize = 5
    , paddleHeight = 50
    , paddleWidth = 5
    , paddleSpeed = 3
    , nudgeSpeed = 3
    }
```

9.3.1 `updateState`

If we know what is in our state, we also know how to update it: we just update every component of it, using appropriate helper functions. The devil will be in the details of them, but we can keep this top-level `updateState` simple:

```
updateState :: Params -> Inputs -> St -> St
updateState params inp = execState $ do
    updateBall params inp
    updatePaddle params inp
    checkBounds params
```

Updating the ball is very similar to our previous code: we update its horizontal and vertical position and speed separately. Vertically, there is no extra complication; but horizontally, we need to include the paddle as a reflector only if the ball is (vertically) where the paddle is. Moreover, to implement nudging the ball on a hit with the paddle, we need to detect that collision; so we will extend `reflect` slightly to return an additional `Bool` denoting collisions.

```
reflect
    :: (Num a, Num a', Ord a, Ord a')
    => (a, a')
    -> (a, a')
    -> (Bool, (a, a'))
reflect (p, n) (x, dx)
    | sameDirection n dist = (True, (p + dist, negate dx))
    | otherwise = (False, (x, dx))
  where
    sameDirection u v = compare 0 u == compare 0 v
    dist = p - x
```

We will use `reflect` and `move` in our `State St` monad, so let's use a shorthand for their lifted versions:

```
moveM :: (Num a) => State (a, a) ()
moveM = modify move

reflectM
    :: (Num a, Num a', Ord a, Ord a')
    => (a, a') -> State (a, a') Bool
reflectM = state . reflect
```

This gives us everything to implement `updateBall`:

- `updateVert` simply moves the ball and checks for reflections from the top and bottom walls:

```
updateVert :: Params -> State St ()
updateVert MkParams{..} = void $ do
    zoom ballV $ do
      moveM
      reflectM (wallSize, 1)
      reflectM (screenHeight - wallSize - ballSize, -1)
```

- `updateHoriz` looks at the vertical position to see we are at the height of the paddle, and decides based on that whether to include a second reflector on the right-hand size.

```
updateHoriz :: Params -> State St Bool
updateHoriz MkParams{..} = do
    atPaddle <- do
        y0 <- use paddleY
        (y, _) <- use ballV
        return $ y `between` (y0 - ballSize, y0 + paddleHeight)
    zoom ballH $ do
        moveM
        reflectM (wallSize, 1)
        if not atPaddle then return False
          else reflectM (screenWidth - paddleWidth - ballSize, -1)
```

- `updateBall` itself runs `updateVert` and `updateHoriz`, and changes the vertical ball speed if `updateHoriz` returns `True`, i.e. if there was a collision with the paddle:

```
updateBall :: Params -> Inputs -> State St ()
updateBall params@MkParams{..} MkInputs{..} = do
    updateVert params
    hitPaddle <- updateHoriz params
    when hitPaddle $ ballV._2 += nudge
  where
    nudge | paddleDown = nudgeSpeed
          | paddleUp = negate nudgeSpeed
          | otherwise = 0
```

Compared to the complicated logic of checking for bounces and paddle hits, updating the paddle's state is much simpler: we increase or decrease `paddleY` by `paddleSpeed` depending on which input buttons are held in the given frame, and then ensure it stays in the playfield:

```
updatePaddle :: Params -> Inputs -> State St ()
updatePaddle MkParams{..} MkInputs{..} = do
    when paddleUp $ paddleY -= paddleSpeed
    when paddleDown $ paddleY += paddleSpeed
    paddleY %= clamp (wallSize, screenHeight - (wallSize + paddleHeight))

clamp :: (Ord a) => (a, a) -> a -> a
clamp (lo, hi) = max lo . min hi
```

To detect the ball going out of bounds, we simply check if its X coordinate is larger than the screen width. This way, even after the ball passes the point of no return at the edge of the paddle, we will keep drawing it until it fully leaves the screen – increasing the player's frustration just a bit more.

It is here that we set the `gameOver` field of the state. As mentioned earlier, this field is set to `True` only for the duration of the single frame when the ball actually flies out of bounds – in the next frame, the ball is already reset into the middle of the playfield (keeping its current speed and Y coordinate).

```
checkBounds :: Params -> State St ()
checkBounds MkParams{..} = do
    outOfBounds <- zoom ballH $ gets $ \(x, _) -> x > screenWidth
    gameOver .= outOfBounds
    when outOfBounds resetBall
  where
    resetBall = ballH._1 .= half screenWidth
```

9.4 Drawing

The main structure of `draw` is very similar to the bouncing ball example: we just have more shapes to check against the current raster beam position.

```
draw :: Params -> St -> Index ScreenWidth -> Index ScreenHeight -> Color
draw MkParams{..} MkSt{..} ix iy
    | isWall    = white
    | isPaddle  = blue
    | isBall    = yellow
    | otherwise = if _gameOver then red else gray
  where
    x = fromIntegral ix
    y = fromIntegral iy

    rect (x0, y0) (w, h) =
        x `between` (x0, x0 + w) &&
        y `between` (y0, y0 + h)

    -- Continued below
```

The definitions of the individual shapes are all straightforward: the paddle and the ball are rectangles, and each wall is an even simpler comparison.

```
    isWall = or
        [ x < wallSize
        , y < wallSize
        , y >= screenHeight - wallSize
        ]

    paddleStart = screenWidth - paddleWidth
    isPaddle = rect (paddleStart, _paddleY) (paddleWidth, paddleHeight)

    (ballX, _) = _ballH
    (ballY, _) = _ballV
    isBall = rect (ballX, ballY) (ballSize, ballSize)
```

For completeness's sake, here are some RGB values for these colors that hopefully won't hurt the player's eyes too much:

```
white, blue, yellow, red, gray :: Color
white  = (0xff, 0xff, 0xff)
blue   = (0x40, 0x80, 0xf0)
yellow = (0xf0, 0xe0, 0x40)
red    = (0x80, 0x00, 0x00)
gray   = (0x30, 0x30, 0x30)
```

By plugging these definitions of `St`, `updateState`, and `draw` into our `topEntity` and `main`, we finish our implementation of (solitaire) Pong, including an interactive SDL simulation.

Exercises:

- On each successful hit, flash the part of the physical screen that is outside the playing area of the virtual screen.
- Increase difficulty as the game goes on by decreasing the paddle size on every k^{th} successful hit, up to a reasonable minimum paddle size.
- Draw the ball as a more round shape.
- Keep score. Drawing numerals would be quite hard with what we have so far, but we could e.g. draw a progress bar counting up to 10 misses.
- Two-player mode. This should be quite self-explanatory: hook up two more buttons to move a second, left-hand side paddle up and down. The score could be displayed as a tug-of-war progress bar.
- A fun variant of two-player Pong is to give a third "boost" button to both players, and scale up the ball's speed by two if exactly one of the players is holding their boost button.

9.5 Summary

- Starting with the implementation of the bouncing ball circuit, the only structural change is adding an **input** signal.
- The rest of the changes are in the definition of the **state datatype** and its **transition function**. These are all "normal", pure Haskell parts that we just happen to use in the context of a Clash circuit.

10 Asynchronous Serial Communication

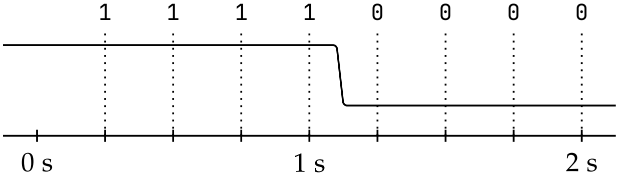
Serial communication is when a one-bit channel is used to transmit data bit by bit. It can be thought of as a kind of temporal n-to-1 multiplexing where n is the number of bits transmitted. In serial communication however, the n is usually not fixed upfront; instead, transmission happens either continuously, or whenever new data is ready to be sent. Its main advantage compared to parallel communication is the lower number of wires needed. In this book, we are going to implement serial communication to interface with other devices that already use some kind of serial protocol, so we don't really have much say in the matter: if we want to talk to those devices, we have to talk serial.

10.1 Synchronicity

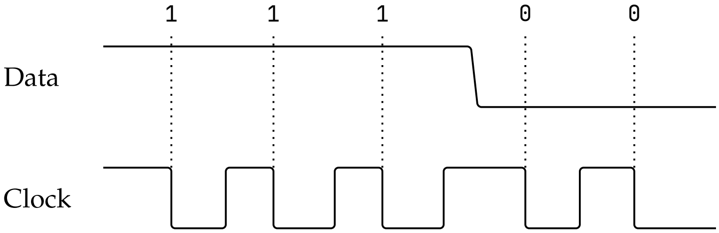
To make serial communication work, the transmitter and the receiver has to have some kind of shared notion of time. To see why, imagine a serial protocol that just says "keep changing the one-bit channel to the next bit of the message". If the receiver side observes the channel's value being high for 1 second and low for 1 second, is that meant to correspond to the bit stream `0b10`, or `0b1100` or `0b111000`, or `0b11111100` because the transmitter decided to take things a bit slower after the first six bits?

The two solutions to this problem is either for the sender and the receiver to agree on a transmission speed (so-called **asynchronous serial communication**), or to add a second clock line that signals when the data line has a new value (**synchronous serial communication**). Depending on the details of the protocol, the clock signal may be generated either by the transmitter or the receiver side.

Let's see how our example of a 1 second high level followed by 1 second low level would be disambiguated into a stream of bits using either an asynchronous or a synchronous protocol. In the first case, let's say the protocol prescribes a transmission period of 250 ms per bit. Then the receiver would know that the full 1 second of a high level would mean 4 high bits:



In the second case, suppose the protocol says to use a falling-edge clock, i.e. the data line is sampled by the receiver at every time the clock line goes from high to low. Then, the following constellation of clock and data signals would correspond to the bit sequence `0b11100`. Note that in this example, the clock signal isn't completely regular; the data line is sampled according to the clock line, not according to the real passage of time.



In this chapter, we will implement the so-called *Universal Asynchronous Serial Communication* protocol, which we will use to communicate with "normal" computers outside the FPGA. Later, in chapter 16, we will implement the synchronous *PS/2* protocol to interface with keyboards.

10.2 Universal Asynchronous Serial Communication

As the "universal" part of its name implies, this is a whole family of serial communication protocols. Our interest in this family is because it provides a straightforward way to communicate with other computers; in particular, the personal computer we're using to develop and compile our Clash code. Even though PCs moved away from the classical RS-232 serial port two decades ago, FPGA development boards usually provide a USB-over-serial interface. This means plugging in the FPGA dev board via USB creates a virtual serial port on the PC side, which can then be used by data transfer programs or interactive serial terminal applications. To put this in concrete terms, the net result of all this is that we will be able to type in a terminal program on our PC and get meaningful replies from our circuit running on the FPGA, printed in the same window.

We are going to do this by implementing a device called a *universal asynchronous receiver-transmitter* (*UART*) which takes care of interfacing between parallel data of some set width and the one-bit serial stream. By using two pins of the FPGA, one for transmission and one for reception, we can implement two-directional full-duplex communication.

10.2.1 Serial data format

Before we can describe the format of the serial data used in UART communication, we first need to enumerate the free parameters that we'll need to refer to in the description. The parameters describing the UART's behavior are:

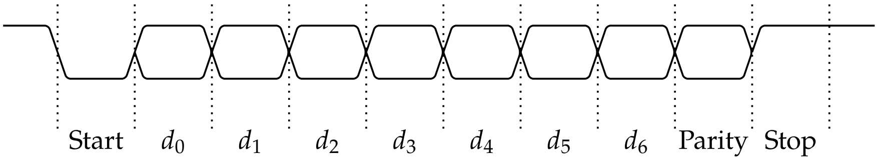
- The data rate (in bits per second). In practice, there are some standard rates (e.g. powers-of-2 multiples of 300 or 600 bps) due to historical reasons. Below where we refer to "one bit's length" it is meant according to this rate.
- The order of the data bits, i.e. least- or most-significant bit first.
- The number of data bits d, between 5 and 9.
- The scheme used for the parity bit, if any.
- The number of stop bits s (at least one).

Of course, both communicating parties need to be configured for the same parameters for successful data transfer.

With these parameters set, the serial format for transmitting the data bits $x_1, x_2, \ldots, x_d$ is as follows:

1. At idle, the line is kept at the high level.
2. When transmission starts, first a *start bit* must be sent. This is done by setting the line to low, and keeping it such for one full bit length.
3. Then, the *d data bits* are sent, by keeping the line low or high for one bit length for each bit.
4. If parity checking is used, the *parity bit* is sent next. Depending on the parity scheme used, the parity bit ensures that the total number of high data and parity bits is even or odd. This is not enough for error correction, but at least makes one-bit error detection possible: for example, if the 8 data bits and the odd parity bit received are `0b111010101`, we know there was an error (caused by e.g. line noise) since an even number of bits are high.
5. The *s stop bits* are sent at high value.

The following figure illustrates the $d = 7, p = 1, s = 1$ configuration:



For simplicity's sake, we will hardcode some of the UART parameters: in particular, we will transmit the least significant bit first, have no parity bits, and use one stop bit. These choices will put us in good company, since these are the defaults used by most PC applications. It also matches what we will need in chapter 18 to implement the virtual floppy drive. Fully universal generalization of the UART implementation could still be useful for other applications, of course, but we leave it as an exercise for the reader.

10.3 Serial Transmitter

Human communication shows us that shouting into the void is easier than listening. In this theme, we will start with implementing the transmitter half of a receiver-transmitter.

For a first version, let's write a serial transmitter that transmits at its driving clock's rate. At any given cycle, the transmitted bit is determined by where we are in the process (e.g. it is `low` for the start bit) and also, of course, by the bit-vector we are transmitting. So the transmitter's internal state will need to know which part of the protocol we are transmitting; and once transmission starts, will need to store the bits to be shifted out:

```
data TxState
    = TxIdle
    | StartBit (BitVector n)
    | DataBit (BitVector n) (Index n)
    | StopBit
    deriving (Show, Eq, Generic, NFDataX)
```

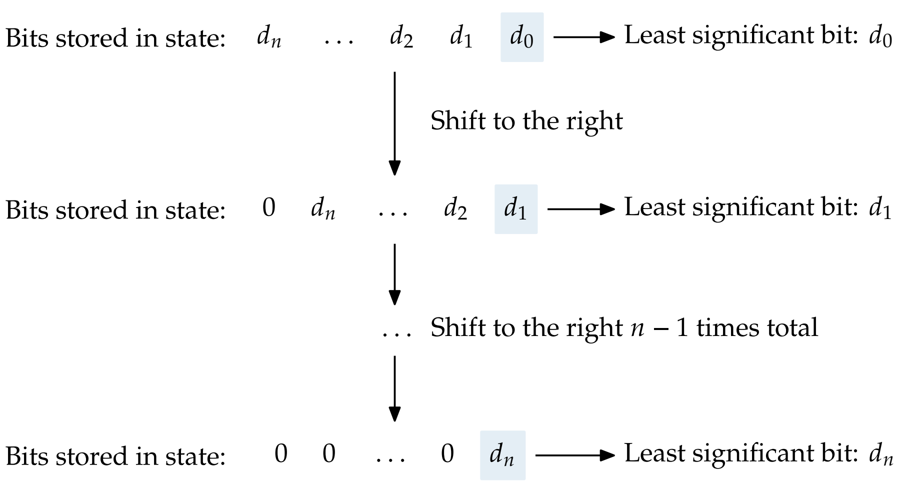
We can approach this with a Moore machine mindset, since we know exactly what the output bit should be in each state:

```
output :: TxState -> Bit
output TxIdle = high
output (StartBit _) = low
output (DataBit xs i) = xs ! i
output StopBit = high
```

This will do the right thing (i.e. least-significant bit first) for `DataBit` if `i` is **decreasing**, since `xs ! (n-1)` is the least significant bit of `xs`, `x ! (n-2)` is the second-least significant bit, and so on. So in our `step` function, we just need to keep decreasing `i` until it gets to the end. Also, we only need to look at the input in the `TxIdle` state: once we get running, the `TxState` stores the data. This makes for a much easier-to-use interface than requiring client code to keep the input constant for the whole duration of the transmission.

```
nextState
    :: (KnownNat n) => TxState n -> Maybe (BitVector n) -> TxState n
nextState TxIdle input = maybe TxIdle StartBit input
nextState (StartBit xs) _ = DataBit xs maxBound
nextState (DataBit xs i) _ = maybe StopBit (DataBit xs) $ predIdx i
nextState StopBit _ = TxIdle
```

This would work, but the vector indexing in the `DataBit xs i` branch of `output` can be rewritten to be much lighter on FPGA parts usage. If we keep shifting `xs` to the right at every `nextStep`, then its least significant bit will go through all bits of the input as we process the `n` states of `DataBit`, and now we don't need indexing. Also, we can get rid of the ugly counting-down in the index, and keep counting up, since its value doesn't matter anymore, just that we stay in `StartBit` for `n` bits.


Bits stored in state: d_n ... d_2 d_1 d_0
Least significant bit: d_0
Shift to the right
Bits stored in state: 0 d_n ... d_2 d_1
Least significant bit: d_1
... Shift to the right $n-1$ times total
Bits stored in state: 0 0 ... 0 d_n
Least significant bit: d_n

The changes required to `output` and `nextState` are tiny. `bvShiftR` shifts to the right by adding the given bit to the left, and returns the just-shifted-out bit along with the new bits.

```
bvShiftR :: (KnownNat n) => Bit -> BitVector n -> (BitVector n, Bit)
bvShiftR x xs = bitCoerce (x, xs)

output (DataBit xs i) = lsb xs

nextState (StartBit xs) _ = DataBit x 0
nextState (DataBit xs i) _ =
    let (xs', _) = bvShiftR 0 xs
    in maybe StopBit (DataBit xs') $ succIdx i
```

If we put `nextState` and `output` together using the `moore` combinator, we get the following serial transmitter (a *UAT*, maybe?):

```
minimalTx
    :: (KnownNat n, HiddenClockResetEnable dom)
    => Signal dom (Maybe (BitVector n)) -> Signal dom Bit
minimalTx = moore nextState output TxIdle
```

But this is simplified to uselessness, so we have to address at least its two main shortcomings first, before building anything around it: we need to slow it down to the transmission rates that are actually going to be supported by anything on the other side, and we need a way to signal the consumption of incoming data.

10.3.1 Taking it slow

One approach to slowing down any Moore machine could be to simply hook up an extra `Bool` signal, and only update the state whenever its value is `True`:

```
mooreSlow
    :: (HiddenClockResetEnable dom, NFDataX s)
    => (s -> i -> s)
    -> (s -> o)
    -> s
    -> Signal dom Bool
    -> Signal dom i
    -> Signal dom o
mooreSlow step output tick input =
    moore step' output $ bundle (tick, input)
  where
    step' s (tick, input) = if tick then step s input else s
```

But this is not quite flexible enough for a UART, since we want to leave the `TxIdle` state as soon as an input is available, and start the clock only then. So instead, we need to keep a separate counter for all the slow states, which we can start at the `TxIdle` to `StartBit` transition; and then at every tick, we maintain this counter to see if we should really go to the next state.

We can make the number of cycles to maintain each output bit fully freely configurable by making it an extra input (alongside the `BitVector n` to shift out). For most applications, we will set it to a fixed number (calculated from the clock rate and a given target rate), but for others, it makes sense to leave it configurable. For example, if we're building a generic serial terminal (with a screen and a keyboard hooked up as peripherals), we would want to enable the user to change the serial communication settings to whatever is used by the computer on the other end.

```
data TxState n
    = TxIdle
    | TxBit Word32 (TxBit n)
    deriving (Show, Eq, Generic, NFDataX)

data TxBit n
    = StartBit (BitVector n)
    | DataBit (BitVector n) (Index n)
    | StopBit
    deriving (Show, Eq, Generic, NFDataX)

nextState
    :: (KnownNat n)
    => TxState n -> Word32 -> Maybe (BitVector n) -> TxState n
nextState TxIdle _ input = maybe TxIdle (TxBit 1 . StartBit) input
nextState (TxBit cnt s) dur _
  | cnt < dur = TxBit (cnt + 1) s
  | otherwise = case s of
      StartBit xs -> TxBit 1 $ DataBit xs 0
      DataBit xs i -> TxBit 1 $
          let (xs', _) = bvShiftR 0 xs
          in maybe StopBit (DataBit xs') $ succIdx i
      StopBit -> TxIdle
```

(`output` doesn't need changing from the previous version, modulo pattern matching on `TxBit _ s` instead of `s` for slow states.)

Then, implementing the slowed-down equivalent of `minimalTx` is a simple matter of shuffling around parameters by judicious use of bundling and currying:

```
slowTx
    :: (KnownNat n, HiddenClockResetEnable dom)
    => Signal dom (Maybe (BitVector n)) -> Signal dom Bit
slowTx = curry $ moore (uncurry . nextState) output TxIdle . bundle
```

10.3.2 Upstream control

To see why we need to do some kind of upstream control, imagine a very simple circuit that keeps sending a predetermined message, byte by byte. If the full message is just one byte `x`, this is easy to do even with `minimalTx`: we can put `Just x` as the input, which will be repeatedly picked up in the `TXState` of `TxIdle`. But if we want to send out a sequence made up of `x` followed by `y`, we need to change the input from `Just x` to `Just y` at the right time (i.e. any time between `minimalTx` entering `StartBit` and leaving `StopBit`). However, our `minimalTx` function doesn't expose this information. So let's change that!

The basic idea is to change the transmitter's interface so that it has two outputs: the `Bit` representing the serial line's value, to be connected to the outside world; and also a `Bool` signaling that the transmitter is ready to send the next message.

```
slowTx
    :: (KnownNat n, HiddenClockResetEnable dom)
    => Signal dom Word32
    -> Signal dom (Maybe (BitVector n))
    -> (Signal dom Bit, Signal dom Bool)
slowTx = curry $ moore (uncurry . nextState) output' TxIdle . bundle
  where
    output' s = (output s, s == TxIdle)
```

10.3.3 Modeling state as `State`

One problem with splitting a stateful circuit's description into `nextState` and `output` is that it makes it very awkward, if not impossible, to compute the circuit's output based on the state *transition*. For example, suppose we want to have another `Bool` output from `slowTx` that signals the consumption of its input, i.e. a transition from `TxIdle` to `StartBit 0`.

The relevance of this problem is that if we compute the output together with the new state, the type of the state transition becomes the following:

```
step :: i -> s -> (s, o)
```

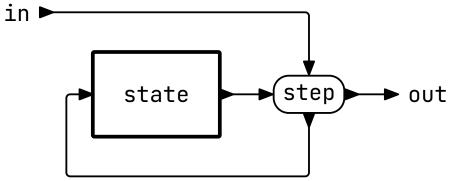
Recall the definition of the `State` monad:

```
newtype State s a = State{ runState :: s -> (a, s) }
```

That's right: the type of `step` is isomorphic to a function from `i` to `State s o`! In other words, we can provide a `State` monad based interface to stateful circuits.

This shouldn't come as a big surprise, since we are describing a stateful computation, so `State` should be the obvious representation, right? However, in chapter 4, when we introduced the whole RTL model, we said that `State` isn't enough to accurately model a circuit with registers, because it would correspond to a non-clocked circuit, with all the problems that entails.

The resolution to this seeming contradiction is that here, we are using `State` to model the *intra-cycle* behavior of our RTL circuit. Under the hood, we will use Clash's `mealy` combinator, which is to *Mealy machines* what `moore` is to Moore machines: lifting a stateful computation into a `Signal` transformer. The difference is exactly what we want: in a Mealy machine, the output is computed as a state transition occurs, i.e. from the previous state and the input, together with the new state:



```
mealy
    :: (HiddenClockResetEnable dom, NFDataX s)
    => (s -> i -> (s, o))
    -> s
    -> Signal dom i
    -> Signal dom o
```

`mealy` internally creates a synchronous RTL circuit with a `register`-guarded recursion to propagate the state from one clock cycle to the next, just like any other RTL circuits we've built so far. But the behavior of the circuit during a *single* cycle *can* be modeled by `State`, by taking its entirety and connecting it between the output and the input of the state register. This means no outside `Signal` can interact with intermediate steps of the `State` circuit, and thus synchronicity is maintained.

This explanation can be captured in code by simply writing the following combinator. It doesn't do anything deep, just gets rid of the `State` newtype wrapper and shuffles some pairs because the `step` argument to `mealy` has the input and the old state, and the output and the new state, in reverse order compared to `runState`:

```
mealyState
   :: (HiddenClockResetEnable dom, NFDataX s)
   => (i -> State s o) -> s -> (Signal dom i -> Signal dom o)
mealyState f = mealy step
  where
    step s x = let (y, s') = runState (f x) s in (s', y)
```

Or, for cases where we want to use bundling to support multiple inputs and outputs (i.e. if we want something like `(Signal dom a, Signal dom b) -> (Signal dom c, Signal dom d)`):

```
mealyStateB
    :: (HiddenClockResetEnable dom, NFDataX s, Bundle i, Bundle o)
    => (i -> State s o)
    -> s
    -> Unbundled dom i
    -> Unbundled dom o
mealyStateB f s0 = unbundle . mealyState f s0 . bundle
```

This finally allows us to write a nicely readable version of our serial transmitter in Mealy style, using the familiar `State` monad. We have a stateful computation of the next `Bit` to be emitted; and we put it in a wrapper to run it `slowly`, by overriding the state change (but keeping the output result) while counting down from one bit's `duration`. Also, we can use a `WriterT` internally to signal readiness in the `TxIdle` state.

```
txStep :: forall n. (KnownNat n)
    => Word32 -> Maybe (BitVector n) -> State (TxState n) (Bit, Bool)
txStep dur input = fmap (fmap getAny) . runWriterT $ get >>= \case
    TxIdle -> do
        tell $ Any True
        traverse_ (goto . StartBit) input
        return high
    TxBit cnt tx -> slowly cnt tx $ case tx of
        StartBit xs -> do
            goto $ DataBit xs 0
            return low
        DataBit xs i -> do
            let (xs', _) = bvShiftR 0 xs
            goto $ maybe StopBit (DataBit xs') $ succIdx i
            return $ lsb xs
        StopBit -> do
            put TxIdle
            return high
```

```
  where
    goto = put . TxBit dur

    slowly cnt tx body
        | cnt > 1 = body <* put (TxBit (cnt - 1) tx)
        | otherwise = body
```

For easier usability, we expose two versions of the serial transmitter: one for dynamic bit durations, and one where the statically known bitrate is converted to the necessary duration based on the main clock rate.

```
serialTxDyn
    :: (KnownNat n, HiddenClockResetEnable dom)
    => Signal dom Word32
    -> Signal dom (Maybe (BitVector n))
    -> (Signal dom Bit, Signal dom Bool)
serialTxDyn dur input = mealyStateB (uncurry txStep) TxIdle (dur, input)

serialTx
    :: forall n rate dom. (HiddenClockResetEnable dom,
       KnownNat n, KnownNat (ClockDivider dom (HzToPeriod rate)))
    => SNat rate
    -> Signal dom (Maybe (BitVector n))
    -> (Signal dom Bit, Signal dom Bool)
serialTx rate = serialTxDyn $ pure dur
  where
    dur = fromIntegral . natVal $
        SNat @(ClockDivider dom (HzToPeriod rate))
```

10.4 Serial Receiver

With all the machinery we've built up for the transmitter, implementing the receiver part is going to be a much more direct affair. The transmitter kept the serial line at the right signal level for the whole bit duration, but the receiver has to decide when to sample it. In our implementation, we are going to keep it simple and do a single-cycle sampling at the exact middle of the period. We're going to do this by basically splitting each `RxBit` state into two: before the sampling, and after. This way, by using a counter equal to half a bit `duration`, we can transition from non-sampled to sampled, and then to the next bit if we do have a sampled bit for this state.

To compute half of bit `dur`, we have to be a bit careful. First of all, we want to use a bit-shift to do the actual division, to get a very simple circuit:

```
half :: (Bits a) => a -> a
half x = x `shiftR` 1
```

Second, we have to make sure that the two halves together add up to a complete duration. To this end, instead of using `half dur` for both halves of the waiting, we will use `half dur` and `dur - half dur`:

```
data RxState n
    = RxIdle
    | RxBit Word32 (Maybe Bit) (RxBit n)
    deriving (Generic, Eq, Show, NFDataX)

data RxBit n
    = StartBit
    | DataBit (BitVector n) (Index n)
    | StopBit (BitVector n)
    deriving (Generic, Eq, Show, NFDataX)

rxStep
    :: (KnownNat n)
    => Word32 -> Bit -> State (RxState n) (Maybe (BitVector n))
rxStep dur input = fmap getLast . execWriterT $ get >>= \case
    RxIdle -> do
        when (input == low) $ waitFor StartBit
    RxBit cnt sample b | cnt > 1 -> do
        put $ RxBit (cnt - 1) sample b
    RxBit _ Nothing b -> do
        consume input b
    RxBit _ (Just sample) rx -> case rx of
        StartBit -> do
            if sample == low then waitFor (DataBit 0 0) else put RxIdle
        DataBit xs i -> do
            let (xs', _) = bvShiftR sample xs
            waitFor $ maybe (StopBit xs') (DataBit xs') $ succIdx i
        StopBit xs -> do
            when (sample == high) $ tell $ pure xs
            put RxIdle
  where
    dur1 = half dur
    dur2 = dur - dur1

    waitFor = put . RxBit dur1 Nothing
    consume input = put . RxBit dur2 (Just input)
```

However, there is a subtle off-by-one error in this implementation. To see why, imagine a transmitter that is sending as fast as it can, leaving no gap between the stop and the start bit. In this case, our implementation would spend one cycle in `RxIdle`, then, detecting the low input, would wait `dur1` cycles to sample the start bit, then spend `dur2` cycles afterwards waiting for the start bit to end; this gives a total of $1 + bd_1 + bd_2 = 1 + bd \neq bd$ cycles. Since the communication is asynchronous, this drift will only accumulate until the sender is a whole bit ahead of the receiver, resulting in faulty transmission.

We fix this by simply waiting for `dur1 - 1` cycles (instead of `dur1`) when entering the first half of the `StartBit` state:

```
    RxIdle -> do
        when (input == low) $
          put $ RxBit (dur1 - 1) Nothing StartBit
```

Now that we have `rxStep`, we can define `serialRxDyn` and `serialRx` exactly as we did for the transmitter:

```
serialRxDyn
    :: (KnownNat n, HiddenClockResetEnable dom)
    => Signal dom Word32
    -> Signal dom Bit
    -> Signal dom (Maybe (BitVector n))
serialRxDyn dur input = mealyStateB (uncurry rxStep) RxIdle (dur, input)

serialRx
    :: forall n rate dom. (HiddenClockResetEnable dom,
       KnownNat n, KnownNat (ClockDivider dom (HzToPeriod rate)))
    => SNat rate
    -> Signal dom Bit
    -> Signal dom (Maybe (BitVector n))
serialRx rate = serialRxDyn $ pure dur
  where
    dur = fromIntegral . natVal $
      SNat @(ClockDivider dom (HzToPeriod rate))
```

This finishes our complete UART implementation.

10.5 Applications

10.5.1 Serial Echo

Now that we have a receiver and a transmitter, a natural idea is to connect the two to form an echo circuit: received bytes are transmitted back as-is. This is a nice way to ensure everything we've written so far works: by uploading this design to our FPGA dev board, we can run a serial terminal program on the computer we use to synthesize the circuit, and start typing away. Most serial terminal programs have the option to show *local echo*, i.e. to print every character as it is sent. Don't forget to turn that **off** for this circuit: the whole point is that we want to get the echo back from the remote (FPGA) side, instead of generating it locally.

Whenever we successfully receive an 8-bit value over the serial line from the UART, we want to queue it for sending. We can do this by putting a buffer between the two that is set from the receiver and cleared by the "ready to send" signal from the transmitter. It is a one-element FIFO that silently drops overflowing input (since we prioritize `r`'s current value over the new `input`):

```
fifo
    :: forall a dom. (NFDataX a, HiddenClockResetEnable dom)
    => Signal dom (Maybe a) -> Signal dom Bool -> Signal dom (Maybe a)
fifo input outReady = r
  where
    r = register Nothing $ mux outReady input (mplus <$> r <*> input)
```

Our Echo circuit can now be defined simply by putting this `fifo` between the de-serialized `input` and the transmitter:

```
topEntity
    :: "CLK" ::: Clock System
    -> "RX"  ::: Signal System Bit
    -> "TX"  ::: Signal System Bit
topEntity = withResetEnableGen board
  where
    board rx = tx
      where
        input = serialRx @8 (SNat @9600) rx
        buf = fifo input txReady
        (tx, txReady) = serialTx @8 (SNat @9600) buf
```

Note that the UART timings depend on the `System` clock domain, which Clash defaults to 100 MHz. If the native clock frequency of the FPGA board used is not 100 MHz, the clock division inside `serialRx` and `serialTx` will give the wrong results. As discussed in chapter 4, the `createDomain` Template Haskell function can be used to define a clock domain with the correct frequency. For example, if our FPGA board uses a 16 MHz clock, we can create a 16 MHz domain and use that in the type of `topEntity`:

```
createDomain vSystem{vName="Dom16", vPeriod = hzToPeriod 16_000_000}

topEntity
    :: "CLK" ::: Clock Dom16
    -> "RX"  ::: Signal Dom16 Bit
    -> "TX"  ::: Signal Dom16 Bit
```

Because the clock divider calculation in the UART is all type-directed, no term-level changes are necessary.

10.5.2 Serial Calculator

We can extend our Calculator from chapter 6 to support serial I/O, while still keeping the existing keypad and seven-segment display support. For input, we can parse incoming bytes into a `Maybe Cmd` very straightforwardly, because each byte arriving through the serial port will correspond directly to zero or one commands. We could use 8-bit numeric values for the various commands, but the code can be made much more readable if instead we do a detour to the `Char` type.

```
pattern ByteChar c <- (chr . fromIntegral -> c) where
  ByteChar = fromIntegral . ord

byteToCmd :: Unsigned 8 -> Maybe Cmd
byteToCmd b@(ByteChar c) | '0' <= c && c <= '9' = Just . Digit $
    fromIntegral $ b - ByteChar '0'
byteToCmd (ByteChar '+') = Just $ Op Add
byteToCmd (ByteChar '-') = Just $ Op Subtract
byteToCmd (ByteChar '=') = Just Equals
byteToCmd (ByteChar '\r') = Just Equals
byteToCmd (ByteChar '\DEL') = Just Clear
byteToCmd (ByteChar '\b') = Just Backspace
byteToCmd _ = Nothing
```

Serial output is a bit more involved, because the n decimal digits have to be sent over a long time, waiting for the first digit to be shifted out, then putting the second one on the transmitter, and so on. Also, if we were to only send out the digits, the output would look something like this (for $n = 4$), if the user types in `123`:

```
   0   1  12 123 123 123
```

To properly replicate the software implementation's UI, we need to clear the screen before each update, instead of accumulating the digits shown so far. Luckily, most serial terminal emulator applications support the escape codes of the ancient VT-100 terminal, which means certain sequences of bytes starting with `0x1b` are interpreted as commands instead of displayed to the user as-is. Two of these commands are `[2J` to clear the screen and `[H` to move the cursor to the home position (i.e. to the upper-left corner). If we prepend these two commands (7 bytes in total) to each update, we will get a nicely updated single-line output.

In `serialDisplay`, we transform the n decimal digits in our input into a vector of $7 + n$ bytes, and keep a `Maybe (Index (7 + n))` that tells us which byte is getting sent right now. Whenever the UART acknowledges a byte, we move on to the next one.

```
serialDisplay
    :: forall n dom. (KnownNat n, HiddenClockResetEnable dom)
    => Signal dom Bool
    -> Signal dom (Vec n (Maybe Digit))
    -> Signal dom (Maybe (Unsigned 8))
serialDisplay ack digits =
    mealyStateB step (Nothing @(Index (7 + n)), repeat 0) (ack, digits)
  where
    step (next, digits) = do
        (i, bs) <- get
        case i of
            Nothing -> do
                put (Just 0, clear ++ map fromDigit digits)
                return Nothing
            Just i -> do
                when next $ put (succIdx i, rotateLeftS bs (SNat @1))
                return . Just $ head bs

    fromDigit :: Maybe Digit -> Unsigned 8
    fromDigit = maybe (ByteChar ' ') $ \n ->
        ByteChar '0' + fromIntegral n

    clear :: Vec 7 (Unsigned 8)
    clear =
        -- clear screen
        0x1b :> ByteChar '[' :> ByteChar '2' :> ByteChar 'J' :>
        -- cursor to home position
        0x1b :> ByteChar '[' :> ByteChar 'H' :>
        Nil
```

If we try this out, depending on the terminal emulator, we might see, at idle, horrible flicker on the serial output. This is because as soon as the last byte (the n'th digit) is sent out, the clear screen command is immediately sent out next, followed by a re-send of the visible digits. One way to avoid this flicker is to only start shifting out (i.e. to only go from the index being `Nothing` to `Just 0`) if the displayed digits have actually changed. Since we continuously rotate the buffer of bytes `bs`, after n rotations it will be back to its original value, ready to be compared to the new one:

```
    case i of
        Nothing -> do
            let bs' = clear ++ map fromDigit digits
            when (bs /= bs') $ put (Just 0, bs')
            return Nothing
```

In the top-level board definition, we can combine the keypad-originating `Maybe Cmd` with the serial-originating one. Which one gets prioritized doesn't really matter, since in any realistic use case, one calculator is not going to be hammered both from the keypad and also over serial at the same time fast enough that there is even any chance for a same-cycle conflict. For the serial output, it is even simpler: we just fan out the `digits` to be displayed to `serialDisplay`, and connect that to a serial transmitter.

```
board rx rows = (tx, display, cols)
  where
    display = driveSS (\x -> (encodeHexSS . bitCoerce $ x, False))
digits
    input = inputKeypad keymap
    digits = logic cmd

    (cols, key) = input rows
    cmdKey = (keyToCmd =<<) <$> key

    -- New part starts here
    (tx, ack) = serialTx (SNat @9600) $
        fmap pack <$> serialDisplay ack digits
    cmdSerial = (byteToCmd . unpack =<<) <$>
        serialRx (SNat @9600) rx
    cmd = mplus <$> cmdKey <*> cmdSerial
```

Exercises:

- Starting from the Echo example, write a ROT-13 circuit that accepts ASCII characters over serial, and sends back only the printable ones (`0x20` to `0x7e`, inclusive), but with letters of the alphabet rotated by 13 places. In other words, `'A'` gets replaced by `'N'` and vice versa, `'B'` with `'O'`, and so on. Note that this mapping is self-inverse, so typing in the output again should give back the original input.

- In the Calculator example, `serialDisplay` uses 7 extra 8-bit bytes worth of registers to store the screen-clearing escape sequence while it is getting shifted out, even though this sequence is constant. Get rid of these 56 registers by changing the internal state of the Mealy machine.
- In the Calculator example, instead of comparing the newly displayed number to the old one, arrange for an extra signal (originating from the logic board) that tells `serialDisplay` when to start sending out a new sequence of bytes. For example, `logic` could return the displayed digits wrapped in a `Maybe`, and only set it to `Just` when they change.
- Apply the same technique to Pong, allowing paddle control over a serial receiver.

10.6 Summary

- In **asynchronous serial communication**, there is no shared clock between the two parties, so both sides do their own timekeeping. In the UART format, the **start bit** signals to the receiver to start its timer, synchronizing the clocks at the start of each byte.
- When the serial receiver has received enough bits to make up a full byte, its `Maybe (BitVector n)` output becomes a `Just` value. We need a similar mechanism of signaling the consumption of the next byte to send in the transmitter; otherwise, the source has no way of pacing itself. This pattern of **upstream control** will come up again in later designs.
- Stateful circuits can be modeled as a **Mealy machine**: a state register coupled with a combinational circuit that computes the next state and the output signal from the old state and the input. This allows turning any `State s a` computation into a circuit.
- With how we structured our calculator, we can easily **add serial communication** as a new UI modality.

11 Programmable Machines

Ever since Church and Turing showed that the lambda calculus and Turing machines are equivalent in the class of computation they can express, a huge number of Turing-complete models of computation has been discovered and invented. From Post's correspondence problem to the collectible card game *Magic the Gathering* to Diophantine equations, there is no shortage of systems that can be used to implement computation including universality, i.e. the ability to input the program to run as part of the input. However, unless one is specifically researching computability, there are three specific models of computation that are widely used:

- **Turing machines** are the gold standard when it comes to concrete calculations involving program size and execution time. If we want to do meta-computational calculations like the Busy Beaver function or upper bounds on Kolmogorov complexity and get actual numbers, Turing machines are the way to go.
- Functional programming as a whole is based on the **lambda calculus**. A lot of programming language theory is interested in various restrictions of the lambda calculus to avoid divergent programs while still allowing expressing a large class of the convergent ones; and a lot of compiler research has gone into efficiently implementing functional languages.
- And then there are **RAM machines**, which are basically modeling how real digital electronic computers work, modulo some mathematical abstractions.

Based on this characterization, it shouldn't come as a surprise that the computers we'll be implementing in this book, whether independent designs or re-implementations of historical computers, are going to be based on the RAM machine model.

11.1 RAM machines

Instead of giving a fully formal definition that nails down all the details of some particular variant of RAM machines, here we are going to give an informal description of the computational model we'll be using. Committing to details wouldn't serve us too well because we are not going to implement *the* RAM machine in hardware; instead, in later chapters we are going to implement various practical designs based on this model.

11.1.1 Definition

In its simplest form, a RAM machine consists of a collection of uniform registers called the *memory*, a list of instructions called the *program*, and a *program counter* indexing into the program.

- The **memory** is a numbered list of registers $m = \langle m_1, \ldots \rangle$. In theoretical models aiming at Turing-completeness, usually a finite number of cells are used, with each one holding a natural number; but the opposite approach (an infinite number of finitely-sized registers) can also work. Machines that are designed to be implemented in the physical world (using e.g. digital electronics) have to use a finite register size and a finite number of registers.

- The **program** is a finite, numbered list of instructions $p = \langle p_1, \ldots, p_k \rangle$. The details of the instruction set is the biggest source of variability between various models all usually called RAM machines. The basic idea behind RAM machines, and what separates them from so-called *register machines*, is that there are instructions that operate via *pointer indirection*: for example, instead of only having instructions that say $m_i \longleftarrow m_i + 1$: "increment by one the value of m_i", we can also have $[m_i] \longleftarrow [m_i] + 1$: "increment by one the value of *the cell whose number* is in m_i".

- The **program counter** is an index into the program, i.e. either a value between 1 and k, or some other value in which case there is no way to keep running the machine: it is in a *halted* state. The program counter is incremented by one in each step of execution for most instructions; but to be Turing-complete, there needs to be at least one instruction that changes the program counter conditionally. For example, we can have a direct branch `if` $m_i = 0$ `then goto` l or an indirect one `if` $[m_i] = 0$ `then goto` l.

Some RAM machine models, in an effort to more closely model real-life hardware, have a separate *accumulator* register a, and require arithmetic calculations and pointer dereferencing to go through that. In this model, there are no operations of the form $m_i \longleftarrow f(m_j)$; instead, there is a pair of load/store instructions $a \longleftarrow m_i$ and $m_i \longleftarrow a$ (and $a \longleftarrow [m_i]$ and $[m_i] \longleftarrow a$ if indirect addressing is supported), and everything else is of the form $a \longleftarrow f(a)$.

This is not a restriction on what computations these machines can express; in fact, there is no fundamental extra power from pointer indirection at all. Just the following three, direct-addressed instructions are already enough for universality:

- $m_i \leftarrow m_i + 1$
- $m_i \leftarrow m_i 1$
- `if` $m_i = 0$ `then goto` l

However, indirect operations can greatly increase efficiency, since a single indirect function application — when simulated using direct operations — would require $O(n)$ instructions, and would run in $O(n)$ steps. In practical terms, this speedup comes “for free”, since real-world memory elements support the random-access indexing operation needed to implement indirect access.

11.1.2 Harvard vs. von Neumann architecture

One detail we have completely skipped over in our definition above is where the program is stored and how it is accessed.

In the simpler case, the program “just is”: it is given as an external list of instructions, and only the program counter is changing as the machine runs our program. This setup, where program is stored somewhere outside the main memory, accessed read-only via the program counter, is called the **Harvard architecture**. This approach is commonly used in microcontrollers with internal flash memory storing the firmware.

In contrast, the **von Neumann architecture** stores its program in main memory, so there is no separate program storage and the program counter is just another memory address. In this setup, instructions need to have an *encoding* that allows using memory cells to represent them. One exciting aspect of this approach is that code becomes data becomes code, opening the door to self-modifying programs. Generic-purpose processors usually go with this architecture.

11.2 Memory

One central feature of all computational models discussed so far is the *memory*: an addressable, mutable container of uniformly typed values. Let’s try writing some Clash code that implements such a container. First of all, let’s think about an interface that would make sense. The inputs are going to be the read and write requests: a read request is simply an address, while a write request (if there is one) is a pair of target address and new value. The output is the result of the read request.

Summarizing the above, our aim is to implement something with the following type:

```
type R dom addr = Signal dom (Index addr)
type W dom addr dat = Signal dom (Maybe (Index addr, dat))
type RAM dom addr dat = R dom addr -> W dom addr dat -> Signal dom dat
```

11.2.1 Implementing our own RAM

Perhaps the most straightforward way of implementing memory is to take the description of "collection of registers" literally: a memory of *n* cells then becomes *n* separate registers, each one initialized to some default value, updating when the write request targets it, and returning its value when the read address matches its index. Then the read output of the whole memory is the read result of the single memory cell that is currently selected.

```
enable :: (Applicative f) => f Bool -> f a -> f (Maybe a)
enable en x = mux en (Just <$> x) (pure Nothing)

ram
    :: (HiddenClockResetEnable dom, KnownNat addr, NFDataX dat)
    => dat
    -> RAM dom addr dat
ram initial rd wr = foldl (liftA2 fromMaybe) (pure undefined) cells
  where
    cell i = enable (rd .== i) r
      where
        r = regMaybe initial (update <$> wr)

        update (Just (addr, val)) | addr == i = Just val
        update _ = Nothing

    cells = map cell indicesI
```

Alternatively, we can turn the design above inside-out, and have a register storing all the cell values in a single, large vector. In this approach, reading becomes a simple matter of `Signal`-lifted indexing, and writing is implemented by replacing the vector with an updated one.[1]

```
ram
    :: (HiddenClockResetEnable dom, KnownNat addr, NFDataX dat)
    => dat
    -> RAM dom addr dat
```

[1] Unfortunately, the type of Clash's `replace` function is very liberal in what types can be used to pick the updated element; hence the need for the type signatures on `update` and `step`.

```
ram initial rd wr = storage .!!. rd
  where
    storage = register (repeat initial) (update <$> wr <*> storage)

    update :: (KnownNat n) => Maybe (Index n, a) -> Vec n a -> Vec n a
    update = maybe id (uncurry replace)
```

11.2.2 The need for Clash memory primitives

If we try using the above two definitions of `ram`, a multitude of problems arise:

- Performance of simulation in Clash is bad (especially for our first version)
- The synthesis tools running after Clash to produce the final FPGA configuration slow down considerably
- The resulting FPGA configuration is very wasteful of hardware parts

This last part is the real kicker: depending on the specifics of the FPGA targeted, even a modest RAM of 1000 bytes can be beyond capacity (in practice, the first, many-register version of `ram` seems to be performing worse in this regard as well). The two approaches presented in the previous section simply won't work for pretty much anything we would like to build.

The reason for all these shortcomings is that the building blocks that are used to implement an RTL design are simply much more versatile than what we need here. A real FPGA hardware has a set amount of circuitry organized in units that each can support an RTL design of some limited number of register size and combinational circuit depth. Larger RTL registers can be split "horizontally" across such units, and deeper combinational circuits can be mapped to multiple units connected serially[2]. But just because the combinational circuitry is "simple", this can't necessarily be exploited to use something smaller than whole units.

Instead, we are going to use the built-in memory primitives of Clash that are then implemented by utilizing specialized memory hardware on the FPGA chip. These parts have a much simpler and more regular structure than the units usable as RTL building blocks, since their circuitry is hardwired for the RAM behavior we tried to model above. The total size of these so-called block RAM parts varies by FPGA chip, but in general they are always orders of magnitude larger than the memory one could possibly implement using general-purpose RTL registers on the same chip. If a 1000 bytes of hand-built RAM would take up all space on a chip, maybe 100 kilobytes would fit in its block RAM, of course also leaving free all the logic parts for the rest of the design.

[2] This increases the propagation delay of the circuit end-to-end, but as we've seen earlier, the RTL model abstracts that away.

In summary, we are always going to use the Clash memory primitives instead of hand-written RTL models. There are multiple functions provided, for read-only vs. mutable memory, for uniformly initialized vs. memory that is initialized to hold the contents of a given file, and for synchronous vs. asynchronous memory.

11.2.3 Synchronous vs. asynchronous memory

Yes, this is the third time in this book that we make a distinction between synchronous and asynchronous designs of something. It seems for something to be a valid area of study in digital electronics, it needs to have its own definition of what to call synchronous versus asynchronous.

In the context of memory design, we call a memory interface *asynchronous* if changing the read address prompts an "immediate" change in the read value. Of course, in a real physical circuit nothing is immediate: what is meant here is that the change occurs as fast as possible, inside the given clock period. In contrast, a *synchronous* RAM design will respond to changes on its address pins at the next clock cycle: the value output by the RAM is what is stored at whatever address was put on its input one cycle ago. Working with the synchronous interface is a bit more tedious than its asynchronous counterpart, but block RAM hardware implements synchronous memory, so that is what we need to use to access it with its benefits: large capacity without using up any parts usable for our circuit's logic.

A further possibility is to use RAM which is much slower than synchronous, i.e. something where there can be quite a large number of cycles of delay between a read request and its result being available. This style of memory is what is used in all modern computers: CPUs have become significantly faster than memory, necessitating multiple levels of caching to get good performance from the complete system. This adds substantial complication to CPU design, and we are not going to address it in this book. Here, we are interested in the decade of classic home computers from the late '70s to the early '80s, when RAM and CPU ran in lockstep.

11.2.4 Clash's memory primitives

Based on what we discussed so far, and planning for implementing computers that originally had separate ROM and uninitialized RAM chips, we will use the following functions from the Clash Prelude:

```
blockRam1
    :: (HiddenClockResetEnable dom, NFDataX a, Enum addr, 1 <= n)
    => ResetStrategy r -> SNat n -> a
    -> Signal dom addr -> Signal dom (Maybe (addr, a)) -> Signal dom a
```

```
romFile
    :: (HiddenClockResetEnable dom, KnownNat n, KnownNat m)
    => SNat n -> FilePath
    -> Signal dom (Unsigned n) -> Signal dom (BitVector m)
```

Clash offers more flexibility than what we will need: `blockRam1` is very liberal in the type used for addressing, accepting any `Enumerable` type; on the other hand, the file-initialized memory primitives insist on exposing their result as a raw `BitVector`, to avoid baking in any convention about how to interpret the initializer file's contents. This initializer file is in a very simple text format, where each line contains *m* number of bits, each written as the character `0` or `1`.

Just to show that we haven't lost anything when moving to using these memory primitives, we can massage `blockRam1` into exactly the same format as our hand-written RAM implementations using mostly just unification, and not initializing the cell values at all:

```
ram
    :: forall addr dat dom.
       ( HiddenClockResetEnable dom
       , KnownNat addr, 1 <= addr, NFDataX dat)
    => RAM dom addr dat
ram = blockRam1 NoClearOnReset (SNat @addr) undefined
```

11.2.5 Memory maps and address decoding

Let's suppose we are building a computer based on the von Neumann architecture: program is stored in main memory, accessed just like any other cells. We can't use ROM as the main memory, because then the CPU is unable to change anything; we might as well not even turn on the machine, since nothing is going to change by letting it run. On the other hand, if we use RAM as the main memory, how do we get the program we want to run onto the CPU? When we turn the computer on, the CPU will start fetching its first instruction from some initial address, but since that address is now pointing to uninitialized RAM, it will start executing garbage.

To get around this, it is quite common on von Neumann architectures to have ROM around the address where the CPU fetches its initial instruction, to provide enough code to bootstrap the computer; and RAM elsewhere. In fact, the von Neumann computers we are going to be building in this book all have extensive ROM to contain the code for whole games and operating systems. But this means we need extra circuitry called an *address decoder* that routes memory data lines according to the value of the memory address lines. A so-called *memory map* describes the assignment of addresses to RAM or ROM. In a physical device, there might be multiple memory chips (for example, implementing 64 kB RAM using four 16 kB RAM chips), requiring further address decoding.

Going by just the definition of a RAM machine, we get a device that keeps reading from and writing to memory, but has no effects observable on the outside. A physical implementation of such a machine would only produce heat. An easy way around this that doesn't require any change to the computational model, is to make memory observable on the outside. Even better, we can avoid the clutter of memory cells used as temporary storage by saying that we are going to connect only certain parts of memory to the outside world.

And so we can make a table of outside-observable memory addresses that assigns an intended meaning to each cell. We can make it even more flexible by not using memory for certain cells at all; instead, we can connect peripheral devices to those addresses. Accessing these cells corresponds to sending to or receiving a message from the given peripheral device. This is called *memory-mapped I/O* and it is a very straightforward way of doing input and output without having to modify the CPU, just by hooking up the peripherals to the address decoder.

Note that the address decoder can easily have access to the CPU's write-enable output as well, allowing read and write requests to the same address to be serviced by different devices, or the same device but with different effect. On some computer designs, a read memory access on a peripheral triggers some side-effect on that device.

Some example ideas for memory-mapped I/O, using peripherals we have already encountered:

- Show status information on LEDs. Simply make a single one-cell-sized register that is connected to the address decoder's data lines and some physical pins that are routed to LEDs.

- Similarly, we can connect an array of two-state switches directly to a single, read-only cell.

- Keyboards can be scanned by assigning one write-only address to the column selector and one read-only address to the row input. The program running on the CPU is responsible for appropriately driving the column selector and interpreting the resulting row.

- A "smarter" keyboard that does its own sweeping and housekeeping and exposes a read-only memory area that contains the last scanned key states.

The address decoder need not be static. In fact, we can even connect it through itself to the CPU, so that writing to a specific address changes the address decoder's behavior. This can be used to implement *memory bank switching*: connecting more memory to the CPU than its address space permits. For example, if we have a 64 kB address space and want to access 64 kB of RAM, but also need 16 kB ROM for booting and OS routines, we can use a dynamic address decoder that starts out with 48 kB RAM and 16 kB ROM mapped, but allows programs to turn off ROM access temporarily by writing to a special memory location, switching to 64 kB RAM accessible.

11.3 CPU

In other chapters, we implement specific pieces of hardware, and then combine them into various devices, either designed from scratch, or aiming to match the behavior of some existing machine. For CPUs, we have to proceed slightly differently, since there is no *the* CPU we can implement with some parameterization: every CPU design is different. Because of this, we are going to sketch here a possible large-scale design for CPU circuitry, but the details are going to be filled in when we create our first concrete CPU in the next chapter.

11.3.1 The basic shape of a CPU

The RAM machine model suggests a design for the CPU that has some internal state (at the very least, the program counter) and communicates with mutable RAM and some store of program (which might be the same as the main RAM, depending on the architecture). To execute one program instruction, the CPU must fetch the next instruction and interpret it; read from and write to RAM as needed; and update its program counter. Since we are aiming to use synchronous memory in lockstep with the CPU, most instructions will require execution spanning multiple clock periods:

1. The next instruction's address has to be placed on the program store's address input.
2. The program store's output now has the next instruction. This needs to be decoded by the CPU to figure out what to do next. Depending on the design, some or all instructions might be wider than the unit of storage in the program store. In this case, it can take several clock periods just to find out what the next instruction is.
3. Some instructions only change the internal state. For example, in an accumulator design, the instruction $a \leftarrow f(a)$ only changes an internal register. These instructions can be done in the same cycle as fetching / decoding. However, instructions that read from RAM will need more cycles. For example, $a \leftarrow m_i$ requires putting i on the address input of RAM, and waiting for the next period to get the result.

4. Generic indirect addressing adds yet more latency. For example, to implement $a \leftarrow [m_i]$, we first request i from RAM as in step 3, but then the result m_i, once available, is fed back as the next address to become available in the subsequent period. This can be avoided if the instruction set only allows for indirect addressing in the form of $[a]$, since a is a register available to the CPU for immediate reading.
5. The story is similar for writing to RAM. Depending on the specifics of our machine, it might be possible to implement direct writes like $m_i \leftarrow a$ in the same cycle as the next instruction fetch (but this can get hairy in von Neumann machines where a given memory write might affect the next instruction to be fetched). But for an indirect write like $[m_i] \leftarrow a$, we need an extra period to first read out m_i, just like for an indirect read.
6. The program counter is updated to point at the next instruction. Since the program counter is stored internally, this doesn't need its own clock cycle; instead, the program store's address input can simply be fed from the updated program counter, thereby equating this last period with the very first period of the next instruction.

So what do we see in this abstract description of the various clock cycles needed for a single instruction?

- Writing it out as a sequence of steps is easy on paper, but in a circuit each clock cycle looks the same, unless we keep track of the various phases of execution. The implicit state of "where are we in the process of executing an instruction" needs to become an explicit state stored in an explicit register, updated in each cycle so that in the next cycle we can dispatch to the correct behavior for that period.

- Some instructions require multiple reads from the program store. For example, if we have instructions of the form $a \leftarrow a + k$, this might be stored in the program store as one word containing the tag for this instruction, and the next one containing the value of k. As we fetch this second word, we still need to remember that we are fetching it as an argument to addition; in general, already-fetched parts of instructions also need to be stored in some internal register.

- The phase-to-phase transitions are driven by the program itself; for example, we go from `Fetch` to `ReadMem` only if the fetched instruction has some memory access components (e.g. a direct read). If the instruction is an accumulator update instead, then we might bump the program counter and go to the next `Fetch` directly.

- There is a lower bound on each instruction's length, based on its memory access pattern and its data dependencies. For example, in $a \leftarrow [m_i]$, we need to wait for m_i to be read out before we can start accessing $[m_i]$.

11.3.2 CPU state

Based on these considerations, we can say that we are going to need to store two kinds of state in our CPU:

- Proper **registers** propagate values between the execution of instructions. This can happen explicitly: for example, if the instruction set has an operation to "increase the accumulator", then the accumulator's initial value is whatever it was at the end of the previous instruction, and its new value is what is accessible to the next instruction. Other registers are accessed implicitly: the program counter is incremented at the end of each (non-branching) instruction, setting the scene for the next instruction to be retrieved. Since registers are observable directly by the behavior of various instructions, they are part of the interface between processor and programmer. Thus, when we are implementing a CPU with the goal of compatibility with a pre-existing machine, the set of registers are decided for us by the original designers.

- Other state pertains to the **phase** of execution we find the CPU in. As we have seen already, we don't expect instructions to finish in a single clock period, so we will need to keep track of where we are in that process. This state is not directly addressable from the program, since programmers work at the level of abstraction provided by the machine instructions. As such, CPU implementors have greater freedom in this part of the designing.

It should be noted that in a complete computer system containing a CPU among other parts, parts of the internal execution phase can still be observed. For example, by looking at the contents of the address bus in each clock period, we can determine which phases correspond to memory access.

In this book, the final chapters will implement computers built around the Intel 8080 processor. However, our implementation will make no reference to the internal details of the original CPU from 1974. Instead, we will be content with just instruction-level compatibility. In other words, the registers will behave according to the original processor's documentation, but there will be no one-to-one correspondence in individual clock cycles between our implementation and the reference chip. In fact, in the first version, instructions will not even have the same clock period length end-to-end.

For the Space Invaders arcade machine, we can observe that this level of fidelity is enough simply by booting up the stock firmware, and trying it out. However, the Compucolor II uses precisely orchestrated cooperation between the firmware running on the CPU and the floppy drive to implement data transfer from and to the floppy disk. This will require us to revisit our original Intel 8080 implementation and change its microcode slightly, to ensure each instruction takes the same number of periods.

Going further, the Compucolor II is a user-programmable computer, so just because we can get to the boot-up prompt, and load some programs from floppies, that doesn't tell us if there are programs out there that exploit the cycle-by-cycle details of instructions to implement some delicate I/O. In particular, the *demo scene* is notorious for squeezing out the last bit of performance from old computer platforms by playing all kinds of tricks including, but not limited to, precisely timed I/O strobing and cycle-counted program design. Programs developed for the Compucolor II with this approach may not work correctly on our implementation.[3]

11.3.3 Phases

How the behavior of the CPU is broken down into cycles depends on the concrete CPU design. In the simplest case, we might be able to execute a full instruction in one cycle; in general, however, the execution can be decomposed into the following steps, where each might require separate (and maybe even multiple) cycles:

- *Fetch*: Arranging for the next instruction to be available to the CPU.
- *Decode*: Looking at the just-fetched opcode, decide on what needs to be done. Depending on the architecture, some instructions might be wider than others, i.e. decoding might result in further fetches.
- *Execute*: The effects of an instruction can be RAM access (reading and/or writing), changes to the CPU's internal state (registers), or I/O via special CPU pins.

The flow of information between phases is more than just which phase we are in: for example, we can only execute an instruction if we know what instruction it is: the result of the decoding phase needs to be passed to the execution phase. Similarly, if we detect during decoding that we need to fetch more, we need to also persist the already-fetched instruction data. Depending on the specifics of our design, this can be done either by persisting the raw memory store contents, or the partial decoding results.

The execution phase can be broken down into further sub-phases for effects that require multiple steps. Consider an `if` $m_i = 0$ `then goto` l conditional jump

[3] This author is not aware of any demo productions for the Compucolor II

instruction: it requires reading from RAM before (potentially) changing the internal program counter register. Like with most things CPU, we will discuss the details of this question in later chapters as we encounter various concrete CPU designs.

11.3.4 Hardwired control vs. microcode

One important split in implementing the details of the execution phase is between hardwired and microcoded control. In the first case, instructions directly map to internal CPU state transitions to execute the desired effects. For example, we might have a datatype for `Phase` that has enough parameters to capture the behavior of all possible instructions, and we pick the right one in the decoding phase. This corresponds to hardwired control because if we look through Clash and the RTL abstraction, the circuit described this way is a bunch of multiplexers, the output of each one feeding into registers directly controlling the various execution functions.

In contrast, a microcoded CPU is one where the problem of implementing instructions is shifted to instead implementing so-called micro-operations. These micro-operations are primitive steps that can be done in a single cycle. Each real instruction is then translated into a sequence of multiple micro-operations. For example, if we have an instruction for $m_j \leftarrow m_i + 1$, we can translate it to the following sequence of micro-operations:

1. Put i onto the address bus
2. Read from the data in bus into the internal (micro-)accumulator
3. Increment the accumulator's value
4. Put j onto the address bus and the accumulator's value on the data out bus
5. Do nothing for one cycle, to allow synchronous RAM to apply the write request

In effect, a microcoded CPU is a shell around a smaller, hardwired CPU with a simpler instruction set, and a table mapping instructions to multiple micro-operations.

In this book, we will encounter both design approaches: we will use hardwired control for the Brainfuck and the CHIP-8 machines, and microcode for the Intel 8080.

11.3.5 Memory access contention

Some hardware designs connect multiple devices to the same memory element. A common example of this is generating video output based on the contents of specific parts of the memory. In this case, both the CPU and the video signal generator need to be able to access the same memory. The question then is, what happens if the CPU and the video subsystem want to read from different addresses at the same time?

We are going to solve this problem by deciding on a priority order: in this example, since we can't suspend the video signal generator (it needs to produce the next pixel's RGB value as the continuously sweeping electron beam passes over it), the CPU will have to wait. The granularity of this conflict resolution depends on the granularity of the memory mapping: if the video memory is implemented as its own RAM element, mapped to some contiguous region of the CPU's address space, then a conflict arises only when, during the video signal generator's access, the CPU's address-out bus contains an address that falls into that given region.

Some designs also allow prioritizing CPU access over the video subsystem, to allow uninterrupted calculations at the cost of visual glitches. In chapter 18, we will see that the Compucolor II allows programmers to pick the priority scheme, by mapping the same video memory to two different address regions: accessing it via one region prioritizes the video system (good for interactive applications), while accessing it via the other one never stalls the CPU, benefiting operations heavy on video memory traffic such as clearing the whole screen.

11.4 Summary

- The **RAM machine** model of computation captures the essence of digital electronic computers.
- The **program** can be stored either in main memory (*von Neumann architecture*) or in dedicated read-only memory (*Harvard architecture*).
- **Memory** requires **dedicated primitives** and **synchronous access** to implement efficiently on real hardware.
- A single **address space** can be used to access **multiple memory and non-memory elements** in a uniform way.
- The **CPU** executes instructions by **reading/writing memory**, **accessing internal registers** and **using I/O pins**. Most operations involve multiple cycles; this needs **internal state** to keep track of the various phases of execution.
- Mapping instructions to execution actions can be done via **hardwired control** or by wrapping a smaller micro-CPU and translating instructions into **microcode** running on that smaller CPU.

12 Brainfuck

Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.

— Alan Perlis

In this chapter, we build our first Turing-complete computer, consisting of a CPU, program ROM, work RAM, a multi-digit seven-segment display for output, and a hex keypad for input. The instruction set of the CPU we will implement is very close to the formal minimalism of the RAM machine design.

12.1 Why Brainfuck

Brainfuck, invented by Urban Müller in 1992, was originally designed as a so-called *Turing tarpit language*: a programming language that is intentionally difficult to program in, but is formally Turing-complete, thus frustrating the programmer because the theory proves any possible program should be expressible. Brainfuck achieves its difficulty by having only eight, very low-level instructions, and using syntax that looks like line noise at first glance. Here is a Brainfuck program that outputs `"hello world"`, hopefully illustrating the reason for the name:

```
+[-[<<[+[--->]-[<<<]]]>>>-]>-.---.>..>.<<<<-.<+.>>>>>.>.<<.<-.
```

As amusing as its frustrating qualities are, we are picking Brainfuck for our first computer for different reasons:

- Its simplistic, low-level design, while difficult for humans to *write* programs, makes it easy to *run* programs once they are written. Although it was originally designed as a programming language, it is simple enough to map directly to a hardware implementation. We could say that a lot of the effort has already been paid by the programmer, so the implementor has less work to do.
- While simple, it's not *too* simple. It does not exactly follow the RAM machine design in one important regard: the `[·]` construct implements structured

programming, suggesting a stack-based implementation that can teach us techniques we'll need later for more complicated processor architectures.

- As the first Turing tarpit language that gained popularity, it has amassed a large library of interesting, dare we say even fun, programs, giving us something to do with our computer once we finish building it.

12.2 Brainfuck as a programming language

Like this whole book, here we start with the assumption that we're more comfortable in the world of software than hardware. For this reason, and also because the original description of Brainfuck was as a programming language to be compiled or interpreted on some existing general-purpose computer, we start by looking at it as a language.

A Brainfuck program is executed in an environment consisting of a one-dimensional array of cells and a single pointer into said array. The size and number of cells varies between implementations; the original implementation used 30,000 8-bit cells, so that's what we'll use as well. Initially, all cells contain 0 and the pointer is pointing at cell #0.

The program itself is a finite sequence built from the following constructs:

- Two instructions `>` and `<` for **incrementing and decrementing the pointer**: the visual mnemonic is to imagine a "head" moving "right" or "left" above the array of cells. It is unspecified what happens if the pointer would "fall off" either end of the array.

- Instructions `+` and `-` for **incrementing and decrementing the pointed cell's content**.

- **I/O instructions** `.` to output the current cell's value, and `,` to read input into the current cell.[1]

- **While-loop** `[·]` that executes the block contents while the current cell's value is not zero (checked at the start of each iteration).

We can represent a Brainfuck instruction using the following straightforward algebraic data type:

[1] The Brainfuck instruction names are what they are, but in this author's opinion, `!` for output and `?` for input would have made for a nicer pair of mnemonics.

```
data BF
    = IncrPtr     | DecrPtr
    | Incr        | Decr
    | Output      | Input
    | While [BF]
    deriving Show
```

We can nail down the operational semantics of Brainfuck by writing a simple interpreter in Haskell. The state consists of a pointer into a list of cells, represented as a zipper `([Cell], [Cell])`. I/O is abstracted into a typeclass to allow both interactive and scripted execution.

```
import Control.Monad.State.Strict
import Control.Monad.Loops (whileM_, whileJust_)
import Control.Monad.Extra (ifM, unlessM)

type Cell = Word8

class (Monad m) => MonadBFIO m where
    doOutput :: Cell -> m ()
    doInput :: m Cell

interp :: (MonadBFIO m) => [BF] -> StateT ([Cell], [Cell]) m ()
interp = mapM_ $ \instr -> case instr of
    IncrPtr -> modify $ \(ls, x:rs) -> (x:ls, rs)
    DecrPtr -> modify $ \(x:ls, rs) -> (ls, x:rs)
    Incr -> modifyCell nextIdx
    Decr -> modifyCell predIdx
    Output -> do
        x <- getCell
        lift $ doOutput x
    Input -> do
        x <- lift doInput
        setCell x
    While prog -> whileM_ ((/= 0) <$> getCell) $ interp prog
  where
    getCell = gets $ \(_, x:_) -> x
    setCell x = modify $ \(ls, _:rs) -> (ls, x:rs)
    modifyCell f = modify $ \(ls, x:rs) -> (ls, f x:rs)
```

Suppose we want to try this interpreter by running it in interactive mode on the Hello World example program above. We can make `IO` an instance of `MonadBFIO` easily:

```
import Data.Char (chr, ord)

instance MonadBFIO IO where
    doOutput = putChar . chr . fromIntegral
    doInput = fromIntegral . ord <$> getChar
```

However, we can't just feed the program into `interp` as-is. That is because `interp` takes the program in our structured format, but `+[-[<<...` is using the concrete syntax of a stream of characters. This means we need to do a bit of *parsing* to go from the 1-dimensional linear sequence of characters to the tree structure of `BF`, ignoring any non-Brainfuck characters on the way:

```
parse :: String -> (String, [BF])
parse [] = ([], [])
parse (c:cs) = case c of
    '>' -> one IncrPtr
    '<' -> one DecrPtr
    '+' -> one Incr
    '-' -> one Decr
    '.' -> one Output
    ',' -> one Input
    '[' -> case parse cs of
        (cs', instrs) -> (While instrs:) <$> parse cs'
    ']' -> (cs, [])
    _ -> parse cs
  where
    one instr = (instr:) <$> parse cs
```

With this parsing function, we can finally run our Hello World example, or any other Brainfuck program specified on the command line, in its full glory, initializing the state to point at the leftmost end of a 30,000-long list of 0-valued cells[2]:

```
hello = "+[-[<<[+[--->]-[<<<]]]>>>-]>-.---.>..>.<<<<-.<+.>>>>>.>.<<.<-."

prepareIO :: IO String
prepareIO = do
    hSetBuffering stdout NoBuffering
    fileName <- listToMaybe <$> getArgs
    maybe (return hello) readFile fileName
```

[2] The first line of `prepareIO` seemingly comes out of nothing. It is needed to turn off output buffering, since Brainfuck programs assume that every cell is output immediately. Our Hello World program, for example, doesn't print a trailing newline, which could cause no output to appear if the output has line buffering.

```
main :: IO ()
main = do
    prog <- snd . parse <$> prepareIO
    evalStateT (interp prog) ([], replicate 30000 0)
```

Exercise:

- What happens if the string passed to `parse` contains unbalanced `[`/`]` pairs? Change the parser to report a meaningful error message in that case.

12.3 Brainfuck as byte code

To move the above software interpreter closer to a potential hardware implementation, we need to get rid of the structured intermediate representation of `[BF]` and process the concrete syntax directly, by accessing it one character at a time.

So how can we handle `[·]` blocks, if our view is limited as if looking through a peephole, showing single `[` and `]` characters? Whenever an opening bracket `[` is encountered, if the current cell's value is 0, we need to jump ahead to the matching `]`. Similarly, when we get to a closing bracket `]`, we need to jump back to its matching `[`. Jumping ahead means scanning forward in the program without executing any instructions, keeping track of nested loops until we see a `]` that doesn't belong to an inner loop. Remembering the past is considerably easier than remembering the future: we can implement jumping back from `]` to its `[` pair by storing, somewhere on the side, the value of the program counter whenever an `[` is encountered.

In concrete terms, this means we need to keep some extra state in the interpreter:

- We need to know the address of the next instruction, i.e. the program counter.
- We need to know which *phase* we are in: are we actually executing the program, or are we skipping a `[·]` block by scanning ahead to find the next matching `]`.
- If scanning ahead, we need to know how many levels of nested `[·]` blocks we have crossed so far. When skipping a block like `[·[·[·]·]·]`, only the third `]` marks the spot when execution needs to resume. We can find this out by incrementing a counter every time an `[` is skipped, and decrementing it for each `]`. The matching `]` is the one that would decrement the counter to zero.
- For jumping backwards, we can avoid a leftward scan if we store the program counter at every `[` before entering the loop body. Of course, because of nesting, just storing the last `[` seen is not enough: we need to store a whole stack of program counter values, one for each currently open `[`.

Based on this analysis, the new state components are as follows:

- `pc :: Int` is the index of the next instruction to execute.
- `stack :: [Int]` holds the indices of the enclosing `[` instructions, which we can consult when an ']' instruction is encountered.
- `data Phase` is the execution phase, i.e. skipping or executing.

Combining it with our previous representation of the memory cells, we get the following record:

```
data Phase
    = Exec
    | Skip Word8

data BFMachine = MkBFMachine
    { pc :: Int
    , stack :: [Int]
    , phase :: Phase
    , cells :: ([Cell], [Cell])
    }

initBFMachine = MkBFMachine
    { pc = 0
    , stack = []
    , phase = Exec
    , cells = ([], replicate 30000 0)
    }
```

The full interpreter fetches the next instruction (addressed by the program counter) and interprets it, and does this in a loop while the program counter is valid:

```
interp :: (MonadBFIO m)
    => (Int -> m (Maybe Char)) -> StateT BFMachine m ()
interp fetch = whileJust_ fetchNext interp1
  where
    fetchNext = do
        pc <- gets pc
        modify $ \st -> st{ pc = pc + 1}
        lift $ fetch pc
```

The interpreter for a single step, in the case of a non-`[` or `]` instruction, is very similar to our previous implementation consuming the program in structured format. The only change for these instructions is to not execute them when skipping ahead. The considerations for the handling of `[` and `]` are:

- When skipping an `[`, it means we are entering a new level of nested `[·]` blocks, and thus we need to increment the skipping depth.
- When skipping a `]`, we decrement the skipping depth (potentially ending the skip-ahead phase).
- When executing an `[`, depending on the current cell's value, either we need to leave a breadcrumb for the matching `]` by pushing the current program counter to the stack, or we need to start skipping.
- When executing a `]`, we need to jump back to its matching `[`. Since all `[` instructions push their address to the stack, we can just pop it back into the program counter.

```
interp1 :: (MonadBFIO m) => Char -> StateT BFMachine m ()
interp1 instr = gets phase >>= \case
    Skip depth -> case instr of
        '[' -> goto $ Skip (depth + 1)
        ']' -> goto $ maybe Exec Skip $ predIdx depth
        _ -> return ()
    Exec -> case instr of
        '>' -> modifyCells $ \(ls, x:rs) -> (x:ls, rs)
        '<' -> modifyCells $ \(x:ls, rs) -> (ls, x:rs)
        '+' -> modifyCell nextIdx
        '-' -> modifyCell predIdx
        '.' -> do
            x <- getCell
            lift $ doOutput x
        ',' -> do
            x <- lift doInput
            setCell x
        '[' -> do
            x <- getCell
            if x /= 0 then pushPC else goto $ Skip 0
        ']' -> popPC
        _ -> return ()
  where
    goto phase = modify $ \st -> st{ phase = phase }

    -- Continued below
```

Now all that remains is to round it off with some utility functions that lift single-cell and cell-zipper modifiers to `BFMachine`. Also, the one bit of subtlety for `push` and `pop` is that by the time we run `push`, we have already incremented the program counter after `fetch`ing the `[` instruction. Accordingly, we need to adjust when we pop it back up, so that we go back to `[` to start the next iteration.

```
    modifyCells f = modify $ \st@MkBFMachine{ cells = cells } ->
        st{ cells = f cells }

    modifyCell f = modifyCells $ \(ls, x:rs) -> (ls, f x:rs)
    setCell = modifyCell . const
    getCell = gets $ \MkBFMachine{ cells = (_, x:_) } -> x

    pushPC = modify $ \st@MkBFMachine{ pc = pc, stack = stack } ->
        st{ stack = pc:stack }
    popPC = modify $ \st@MkBFMachine{ stack = top:stack} ->
        st{ pc = top - 1, stack = stack }
```

We can try our Hello World program by `fetch`ing from a string via an array for efficient indexing:

```
fetchFrom :: (Monad m) => String -> (Int -> m (Maybe Char))
fetchFrom prog = \i -> return $ do
    guard $ inRange (bounds arr) i
    return $ arr!i
  where
    arr = listArray (0, length prog - 1) prog

main :: IO ()
main = do
    prog <- prepareIO
    evalStateT (interp (fetchFrom prog)) initBFMachine
```

Exercise:

- As written, `]` will always pop back to its matching `[` before checking that the current cell is non-zero. Change the handling of `]` to avoid this unnecessary jump.

12.4 Brainfuck with external memory

Next, we are going to focus on the modeling of memory, and using fixed-size types that can be represented in hardware.

In `BFMachine`, we included the array of Brainfuck cells in the machine itself: the list zipper `([Cell], [Cell])` is simply a field of the `BFMachine` record. In a real hardware implementation, we would instead use a fixed-size memory array, implemented as block RAM somewhere outside the processor, with the CPU only storing a pointer to the currently selected cell, as a memory address. We also take this opportunity to use tighter types than just `Int` everywhere for addressing. With these considerations, we can create an interface for external memory as a typeclass similar to the IO situation before, with the same intention: we want to concentrate on the internals of our Brainfuck processing unit, without any regard (for now!) to how memory is actually implemented.[3]

```
type PC = Unsigned 12
type MemSize = 30_000
type Ptr = Index MemSize

class (Monad m) => MonadBFMemory m where
    readProgROM :: PC -> m Word8
    readRAM :: Ptr -> m Cell
    writeRAM :: Ptr -> Cell -> m ()
```

Given the above interface, we can rewrite `interp` to fetch the next instruction by requesting the next character from the program ROM, as addressed by the program counter. We'll use the zero character `'\0'` as a marker that we've run out of program to run.

```
interp :: (MonadBFMemory m, MonadBFIO m) => StateT BFState m ()
interp = whileJust_ fetchNext interp1
  where
    fetchNext = do
        pc <- gets pc
        instr <- lift $ ascii <$> readProgROM pc
        if instr == '\0' then return Nothing else do
            modify $ \st -> st{ pc = pc + 1 }
            return $ Just instr

ascii :: Word8 -> Char
ascii = chr . fromIntegral
```

Pushing on, we also need to get rid of the `[PC]` representation of our skip-backwards stack, instead using a fixed-length vector and a stack pointer pointing to the next element to push to. Although a "real" CPU would store the stack in main memory instead (by designating an area of RAM to be used for this purpose), we will make our lives simpler here by restricting the loop nesting length to the arbitrarily chosen size of 32, and store the tiny stack simply in a CPU register.

[3] Having two monadic interfaces to two abstract effects is, arguably, at least one too many. A nicer way of handling this would be to use an algebraic effects library like *Polysemy*; we skip describing that approach here because our focus is on the eventual hardware implementation.

```
data Stack n a = Stack (Vec n a) (Index n)
    deriving (Show)

push :: (KnownNat n) => a -> Stack n a -> Stack n a
push x (Stack xs i) = Stack (replace i x xs) (i + 1)

pop :: (KnownNat n) => Stack n a -> (a, Stack n a)
pop (Stack xs i) = (xs !! (i - 1), Stack xs (i - 1))
```

Now we have everything to describe the internal state of our Brainfuck interpreter: it is similar to `BFMachine`, but instead of the cells themselves, it only contains a pointer to the current cell, and the stack uses the above fixed-size representation:

```
type StackSize = 32

data Phase
    = Exec
    | Skip (Index StackSize)

data BFState = MkBFState
    { pc :: PC
    , stack :: Stack StackSize PC
    , phase :: Phase
    , ptr :: Ptr
    }

initBFState = MkBFState
    { pc = 0
    , stack = Stack (repeat 0) 0
    , phase = Exec
    , ptr = 0
    }
```

The only changes needed for `interp1` are to manipulate the pointer instead of the zipper in the > and < cases, and to change the definition of the local helpers used to access cells the stack:

```
interp1 :: (MonadBFMemory m, MonadBFIO m) => Char -> StateT BFState m ()
interp1 instr = gets phase >>= \case
    Skip depth -> -- Same as before
    Exec -> case instr of
        '>' -> modifyPtr nextIdx
        '<' -> modifyPtr predIdx
        -- Other cases: same as before
```

```
  where
    goto phase = modify $ \st -> st{ phase = phase }

    modifyPtr f = modify $ \st -> st{ ptr = f (ptr st) }

    modifyCell f = do
        x <- getCell
        setCell (f x)
    getCell = do
        ptr <- gets ptr
        lift $ readRAM ptr
    setCell x = do
        ptr <- gets ptr
        lift $ writeRAM ptr x
    pushPC = do
        pc <- gets pc
        modify $ \st -> st{ stack = push (pc - 1) (stack st) }
    popPC = modify $ \st ->
        let (pc', stack') = pop (stack st)
        in st{ pc = pc', stack = stack' }
```

We round it off by giving an instance of `MonadBFMemory` allowing us to try it out from `main`. We could use `Array` again for the representation, but since one of the goals in the current refinement is to give everything precise sizes, we are instead going to use two vectors for the program ROM and the data RAM. The program ROM has size 2^{12} so that `PC` can be used directly to address it, and is passed around in a `ReaderT`. The data RAM has `MemSize` cells, and since its contents can be changed, it is passed around in a `StateT`.

```
type ProgSize = 2 ^ BitSize PC

type WithROM = ReaderT (Vec ProgSize Word8)

runWithROM :: (Monad m) => Vec ProgSize Word8 -> WithROM m a -> m a
runWithROM rom act = runReaderT act rom

type WithRAM = StateT (Vec MemSize Cell)

runWithRAM :: (Monad m) => WithRAM m a -> m a
runWithRAM act = evalStateT act (repeat 0)
```

We compose `WithROM` and `WithRAM` into a `newtype` so that we can attach a `MonadBFMemory` and a (lifting) `MonadBFIO` instances to it:

```
newtype BFVec m a = BFVec{ unBFVec :: WithROM (WithRAM m) a }
    deriving (Functor, Applicative, Monad)

runBFVec :: (Monad m) => Vec n Word8 -> BFVec m a -> m a
runBFVec prog = runWithRAM . runWithROM prog . unBFVec

instance (Monad m) => MonadBFMemory (BFVec m) where
    readProgROM pc = BFVec $ asks (!!pc)
    readRAM addr = BFVec $ gets (!!addr)
    writeRAM addr x = BFVec $ modify $ replace addr x

instance (MonadBFIO m) => MonadBFIO (BFVec m) where
    doOutput = BFVec . lift . doOutput
    doInput = BFVec . lift $ doInput
```

We can load `hello`, or any other string, into a vector by truncating and padding as necessary.

```
loadVec :: (KnownNat n) => [a] -> a -> Vec n a
loadVec xs x0 = unfoldrI uncons xs
  where
    uncons (x:xs) = (x, xs)
    uncons [] = (x0, [])
```

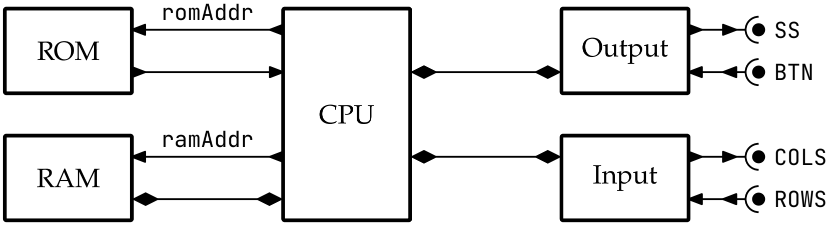
Since `prepareIO` returns the program file's contents as text (in a `String`), we need one more piece before we have everything to call `runBFVec`: a way to turn the loaded, `'\0'`-padded vector of `Chars` into `Word8` bytes. We write this generically over containers because we plan to re-use it later in this chapter for lists instead of vectors.

```
stringToROM :: (Functor f) => f Char -> f Word8
stringToROM = fmap (fromIntegral . ord)

main :: IO ()
main = do
    prog <- prepareIO
    runBFVec (stringToROM $ loadVec hello '\0') $
      evalStateT interp initBFState
```

12.5 A complete Brainfuck computer

Our ultimate aim in this chapter is to build a full Brainfuck computer: something that consumes raw Brainfuck programs as its machine code, and executes them while communicating with the outside world according to the IO instructions of the program. The Harvard architecture is the natural choice for the design, since Brainfuck has no facilities to access the program via memory operations.

The following block diagram shows the components of our computer. For output, we will use a two-digit seven-segment display in hexadecimal mode. Of course, we can't just flash the number for a clock cycle and expect any human to read it. So for this reason, we will require a button press acknowledging any output before continuing with execution. Hence the signal coming from the output module back to the CPU. Similarly, we connect a signal from the CPU to the input module so we can notify the user that we're waiting for their input.



We translate this design to Clash in multiple layers:

- Our `topEntity` connects directly to the keypad and the seven-segment display:

```
topEntity
    :: "CLK"     ::: Clock System
    -> "BTN"     ::: Signal System (Active High)
    -> "ROWS"    ::: Signal System (Vec 4 (Active Low))
    -> ( "SS"    ::: Signal System (SevenSegment 4 Low Low Low)
       , "COLS"  ::: Signal System (Vec 4 (Active Low))
       )
```

- On the next layer, the various peripheral drivers take care of translating to/from the logical representation of IO. This will allow us to easily replace parts or all of the IO peripherals, for example to use a serial line. What remains are the input cell value, the output acknowledgment, the waiting-for-input notification, and the output cell value:

```
logicBoard
    :: (HiddenClockResetEnable dom)
    => Signal dom (Maybe Cell)
    -> Signal dom Bool
    -> (Signal dom Bool, Signal dom (Maybe Cell))
```

- The logic board consists of the RAM, ROM, and the CPU. From the outside, the CPU looks like any other circuit, with some inputs and outputs. We will explore the details of its connections together with its internals.

```
cpu
    :: (HiddenClockResetEnable dom)
    => Signal dom CPUIn
    -> Signal dom CPUOut
```

We will work our way outwards, building the CPU first, then connecting it to the memory, and finally taking care of peripheral IO.

12.6 Brainfuck as machine code

At first glance, it seems the Brainfuck software implementation that uses external memory should work as a hardware circuit as well. In other words, we might expect to be able to write something like `cpu = mealyState interp1 initBFState`, re-using the interpreter directly as hardware. However, some differences remain between our software implementations and real hardware using real memory:

- We used `whileJust_` in the software implementations, but a hardware circuit has no concept of "not running". Instead, we have to explicitly enter a `Halt` state where we don't do anything in subsequent cycles.
- Instead of *doing* IO and memory access as monadic actions, we need to *produce output* and *consume input*. For example, instead of an action to read from RAM, we set some output signals to the address and read the memory value from some input signals.
- We are aiming to use synchronous memory, which complicates operations involving memory access. To read from a certain memory address (either in ROM or RAM), we have to make sure that address was already on the memory's address line in the previous cycle. Writing to RAM only affects reads from the next cycle on.

The first concern's description already contains its solution: we extend the `Phase` datatype with a new constructor for `Halt`. Let's look at the solutions for the other two problems in detail.

12.6.1 Actions as circuit outputs

In software implementations, we were able to affect things outside the interpreter itself, such as memory units or IO, via monadic actions. For example, in the implementation of the . instruction, to output the current cell's value, we were able to write something like the following:

```
x <- getCell
lift $ doOutput x
```

which used the underlying MonadBFIO's implementation of doOutput to write x to the console, or a record of outputs, or whatever else MonadBFIO chooses to do.

A hardware CPU, however, works differently. Its only connection to other components is via its input and output signal lines. Other components, connected to these lines, will take care of turning those signals into something that affects the outside world, for example by changing a seven-segment display to show the cell value as hexadecimal digits. This is evident from the fact that from the outside, a CPU looks just like any other (stateful) circuit:

```
cpu
    :: (HiddenClockResetEnable dom)
    => Signal dom CPUIn
    -> Signal dom CPUOut
```

Let's concentrate on just implementing . for now, without any other effects. In this simplified model, we have no inputs, and a single Maybe Cell output. To have something interesting to show, the internal state will consist of a Cell value that is incremented after each output.

```
data CPUIn = CPUIn{} -- No inputs for now

data CPUOut = CPUOut
    { output :: Maybe Cell
    }

data CPUState = CPUState
    { cell :: Cell
    }
    deriving (Generic, NFDataX)

initState :: CPUState
initState = CPUState
    { cell = 0
    }
```

We have already seen how to write stateful circuits expressed with the State abstraction. Armed with the mealyState combinator, we can approach this problem by writing the CPU as a `CPUIn -> State CPUState CPUOut` function that implements a single-cycle step:

```
type CPU = State CPUState

step :: CPUIn -> CPU CPUOut
step CPUIn{..} = do
    x <- gets cell
    modify $ \s -> s{ cell = x + 1 }
    return $ CPUOut{ output = Just x }

cpu = mealyState step initState
```

This straightforward implementation of `step` looks reasonable, but let's see how well this approach scales with complexity. Let's add a `Bool` input for acknowledging output, e.g. a button the user presses after reading the number shown on the seven-segment display. Now, whenever we output something, we need to start waiting for the button to be pressed. We change `CPUIn` and `CPUState` accordingly:

```
data CPUIn = CPUIn
    { outputAck :: Bool
    }

data Phase
    = Halt
    | Exec
    | WaitOutput
    deriving (Generic, NFDataX)

data CPUState = CPUState
    { phase :: Phase
    , cell :: Cell
    }
    deriving (Generic, NFDataX)

data initState = CPUState
    { phase = Exec
    , cell = 0
    }

goto :: Phase -> CPU ()
goto phase' = modify $ \s -> s{ phase = phase' }
```

The `step` function now becomes more complicated since it needs to handle the two phases differently, and needs to go to the waiting phase after outputting something:

```
step CPUIn{..} = gets phase >>= \case
    Exec -> do
        x <- gets cell
        modify $ \s -> s{ cell = x + 1 }
        goto WaitOutput
        return CPUOut{ output = Just x }
    WaitOutput -> do
        when outputAck $ goto Exec
        return CPUOut{ output = Nothing }
    Halt -> do
        return CPUOut{ output = Nothing }
```

There's quite a lot of detail in this one function. We can factor out the how-to of producing output:

```
outputCell :: Cell -> CPU CPUOut
outputCell x = do
    goto WaitOutput
    return CPUOut{ output = Just x}
```

Leaving us with:

```
step CPUIn{..} = gets phase >>= \case
    Halt -> do
        return CPUOut{ output = Nothing }
    WaitOutput -> do
        when outputAck $ goto Exec
        return CPUOut{ output = Nothing }
    Exec -> do
        x <- gets cell
        modify $ \s -> s{ cell = x + 1 }
        outputCell x
```

But notice that on the `WaitOutput` and `Halt` branches, where we don't want to concern ourselves with the details of output (since we're *not doing any output*), we still have to construct and return a `CPUOut` with the right field values. When we add more fields to `CPUOut` for functionality completely unrelated to outputting cell values, the `WaitOutput` and `Halt` branches will have to take care of all of those new fields.

This is not really equivalent to the software implementation, where the `outputCell x` action took care of emitting *output* without burdening nonoutputting code paths with having to explicitly *not-output*. In more concrete terms, we want to write step as the following:

```
step CPUIn{..} = gets phase >>= \case
    Exec -> do
        x <- gets cell
        modify $ \s -> s{ cell = x + 1 }
        outputCell x
    WaitOutput -> when outputAck $ goto Exec
    Halt -> return ()
```

The key to achieving this is to switch the role of CPUOut from *result* to *output*, by changing the type of step from State CPUState CPUOut to WriterT CPUOut (State CPUState) (). This requires CPUOut to be equipped with a monoid instance, which is easy enough to do:

```
instance Semigroup CPUOut where
    (CPUOut o) <> (CPUOut o') = CPUOut (o `mplus` o')

instance Monoid CPUOut where
    mempty = CPUOut Nothing

type CPU = WriterT CPUOut (State CPUState)

outputCell :: Cell -> CPU ()
outputCell x = do
    tell mempty{ output = Just x }
    goto WaitOutput
```

In fact, by changing CPUOut slightly so that its output field is a monoid itself, we can simplify these instances to be the obvious lifting ones:

```
data CPUOut = CPUOut
    { output :: Last Cell
    }

instance Semigroup CPUOut where
    (CPUOut o) <> (CPUOut o') = CPUOut (o <> o')

instance Monoid CPUOut where
    mempty = CPUOut mempty
```

The change required to `cpu` to run `step` is minuscule, since `execWriterT` has just the right type to turn the `WriterT` shell's output into the inner `State`'s result:

```
cpu = mealyState (execWriterT . step) initState
```

The change to `CPUOut`'s definition does create a bit of a tension between the interface of `cpu` and `step`, since we changed the type of `output` to better suit the latter's implementation instead of the former's intended semantics. In effect, we're leaking an implementation detail in the types. We will revisit this question shortly.

First, however, if we generate field accessor lenses for `CPUOut`, we can provide a combinator that assigns a value to any single output field. Let's use `makeLenses` the usual way on `CPUOut`:

```
data CPUOut = CPUOut
    { _output :: Last Cell
    }
makeLenses ''CPUOut
```

The *Lens* library already has a combinator for the pattern of "`tell`ing a monoid value which is `mempty` with some lens set on it" called `scribe`. This means we can rewrite `outputCell` to:

```
outputCell :: Cell -> CPU ()
outputCell x = do
    scribe output (pure x)
    goto WaitOutput
```

In fact, we will always want to assign `pure` values to these output fields, since the only other possibility would be `mempty` (i.e. not setting the given field at all), which is a no-op. So we finish this section by writing a combinator to assign an output field to a given `pure` value:

```
infix 4 .:=
(.:=)
    :: (Monoid w, Applicative f, MonadWriter w m)
    => Setter' w (f a) -> a -> m ()
fd .:= x = scribe fd (pure x)

outputCell :: Cell -> CPU ()
outputCell x = do
    output .:= x
    goto WaitOutput
```

12.6.2 Reading from synchronous memory

Since we want to use available block RAM elements to implement memory, we will connect synchronous memory to our CPU. This has two consequences for CPU design: read addresses have to be set one cycle in advance, and write requests take one cycle to have an effect.

The first constraint means if we want to implement an instruction that both changes the cell pointer and manipulates the cell data, it will have to take at least two cycles: the first to change the pointer and put the changed value on the address output, and the second to process the input coming from the memory, based on the changed address. The Brainfuck instruction set has no such instructions, which simplifies matters for us: `>` and `<` only change the pointer without manipulating cell values, and the other instructions only operate on the currently pointed cell. This means we can get away with putting an internal pointer register in `CPUState` and connecting it to the address output, and it will always load the right value from memory by the time it is used by a non-pointer-changing instruction.

The same consideration could also apply to the program counter. Similar to the cell pointer, it is true that every time the program counter is changed, fetching the machine code for the new program location is only needed to execute the *next* instruction. So as long as the program counter, as stored in the `CPUState`, is updated correctly before outputting it on the program memory address bus, the next cycle will get from the program ROM the instruction code from that address.

Having convinced ourselves that reading from the program ROM and cell RAM is an easy matter of maintaining the right address registers in `CPUState`, we can try our hand at implementing the instructions that don't modify cell values, i.e. everything except `,` `+` and `-`. Let's start with `<`, `>` and `.` since these don't involve complicated program counter calculations. The new fields added to `CPUIn` feed the instruction (from program ROM) and the memory cell value (from RAM) to the CPU:

```
data CPUIn = CPUIn
    { romRead :: Word8
    , ramRead :: Cell
    , outputAck :: Bool
    }
```

New fields of `CPUState` correspond to the two address registers containing the program counter and the cell pointer; we also switch over to using lenses to access fields, to reduce the noise of all the `modify` calls.

```
data CPUState = CPUState
    { _phase :: Phase
    , _pc :: PC
    , _ptr :: Ptr
    }
    deriving (Generic, NFDataX)
makeLenses ''CPUState

initState = CPUState
    { _phase = Exec
    , _pc = 0
    , _ptr = 0
    }
```

And what about `CPUOut`? Recall that at the topmost level, the end-to-end type of our CPU implementation is to be the following (simplifying the `dom` constraint for brevity):

```
cpu :: Signal dom CPUIn -> Signal dom CPUOut
```

To ensure in `logicBoard` that the `romRead` and `ramRead` fields of `CPUIn` are correctly filled from the program ROM and the cell RAM, respectively, we need `CPUOut` to tell us the right addresses. Combined with the output value coming from the `.` instruction, this leads us to define `CPUOut` as:

```
data CPUOut = CPUOut
    { romAddr :: PC
    , ramAddr :: Ptr
    , output :: Maybe Cell
    }
```

However, we also want to assemble `CPUOut` piecewise, from various monadic actions, which, in the previous section, has led us to writing our CPU in the `WriterT CPUOut (State CPUState)` monad. That would require `CPUOut` to be a monoid; with the semigroup operator corresponding to combining partial outputs. But `romAddr` and `ramAddr` should be filled from the `CPUState` at the end of each cycle; even for the trivial "CPU" of `return ()`, the two addressing fields of `CPUOut` should take the right value.

We solve this problem by making a second datatype `PartialCPUOut`, where each field is combined via `Last`, allowing piecewise assembly inside a `WriterT`. This is more generic than strictly needed for us here, since there will be no situations where we want to write into the `ramAddr'` and `romAddr'` fields: they can always be computed directly from the `CPUState` at the end of any given cycle. We still include them here for flexibility, but also for regularity. This regularity will come useful in the next section, as we will forego the hand-written definition and instances of `PartialCPUOut` and derive them from `CPUOut` itself. But for now, let's keep it simple and write everything manually, accepting the seeming redundancy:

```
data PartialCPUOut = PartialCPUOut
    { _romAddr' :: Last PC
    , _ramAddr' :: Last Ptr
    , _output' :: Last (Maybe Cell)
    }
makeLenses ''PartialCPUOut

instance Semigroup PartialCPUOut where
    (PartialCPUOut pc1 addr1 out1) <> (PartialCPUOut pc2 addr2 out2) =
        PartialCPUOut (pc1 <> pc2) (addr1 <> addr2) (out1 <> out2)

instance Monoid PartialCPUOut where
    mempty = PartialCPUOut mempty mempty mempty
```

We can then take the `PartialCPUOut` output of our `WriterT` and the final `CPUState` of our `State`, and turn it into a full `CPUOut` in two steps: constructing a default `CPUOut`, and `update`ing it:

```
type CPU = WriterT PartialCPUOut (State CPUState)

defaultOutput :: CPUState -> CPUOut
defaultOutput CPUState{..} = CPUOut
    { romAddr = _pc
    , ramAddr = _ptr
    , output = Nothing
    }

update :: CPUOut -> PartialCPUOut -> CPUOut
update CPUOut{..} PartialCPUOut{..} = CPUOut
    { romAddr = fromMaybe romAddr $ getLast romAddr'
    , ramAddr = fromMaybe ramAddr $ getLast ramAddr'
    , output = fromMaybe output $ getLast output'
    }
```

Now we can implement `cpu` by taking the stateful-and-writing computation `step` of our processor, and turning it into a stateful-only function mapping inputs to outputs by applying the edits from the writer on the default output:

```
cpu
    :: (HiddenClockResetEnable dom)
    => Signal dom CPUIn
    -> Signal dom (Pure CPUOut)
cpu = mealyState cpuMachine initState

cpuMachine :: CPUIn -> State CPUState CPUOut
cpuMachine = do
    edits <- execWriterT (step input)
    out0 <- gets defaultOutput
    return $ update out0 edits
```

And finally we get to writing `step`. If we squint hard enough, its structure should look very similar to the software implementations: we pattern match on our internal `phase` and on the next instruction, and change the state and assemble the output accordingly.

```
step :: CPUIn -> CPU ()
step CPUIn{..} = use phase >>= \case
    Exec -> fetch >>= \case
        '>' -> ptr %= nextIdx
        '<' -> ptr %= predIdx
        '.' -> outputCell ramRead
        '\0' -> phase .= Halt
        _ -> return ()
    WaitOutput -> when outputAck $ phase .= Exec
    Halt -> return ()
  where
    fetch = do
        pc += 1
        return $ ascii romRead
```

The next instruction to execute is available on the `romRead` field of the `CPUIn` argument (we arrange for the `romAddr` output to ensure that); instead of pattern-matching on it directly, we access it via `fetch` which increments the program counter (ensuing that it is only incremented in cycles where we actually consume the instruction!) and decodes its ASCII value into a `Char`, for easier pattern matching. At face value, that last step sounds like it could lead to very complicated circuitry; however, GHC and Clash recognizes the immediate pattern matching on the result of `ascii` in `step`, and collapses it all down to a simple byte comparison with `0x3e` (for `'>'`), `0xec` (for `'<'`) and `0x2e` (for `'.'`).

It turns out there is one more complication we need to handle before finishing this section: in the very first cycle, there is no “previous cycle” for the synchronous memory elements to take the address from. This means both the `ramRead` and the romRead fields of CPUIn will be set to undefined values. We can work around this easily by adding one more constructor to Phase to handle initialization on reset.

```
data Phase
    = Init
    | ...

initState = CPUState
    { _phase = Init
    ...
    }

step :: CPUIn -> CPU ()
step CPUIn{..} = use phase >>= \case
    Init -> phase .= Exec
    ...
```

12.6.3 Getting rid of the PartialCPUOut redundancy

Let's repeat the definitions of CPUOUt, PartialCPUOut and the latter's instances, to show them side by side:

```
data CPUOut = CPUOut
    { romAddr :: PC
    , ramAddr :: Ptr
    , output :: Maybe Cell
    }

data PartialCPUOut = PartialCPUOut
    { _romAddr' :: Last PC
    , _ramAddr' :: Last Ptr
    , _output' :: Last (Maybe Cell)
    }
makeLenses ''PartialCPUOut

instance Semigroup PartialCPUOut where
    (PartialCPUOut pc1 addr1 out1) <> (PartialCPUOut pc2 addr2 out2) =
        PartialCPUOut (pc1 <> pc2) (addr1 <> addr2) (out1 <> out2)

instance Monoid PartialCPUOut where
    mempty = PartialCPUOut mempty mempty mempty
```

```
update :: CPUOut -> PartialCPUOut -> CPUOut
update CPUOut{..} PartialCPUOut{..} = CPUOut
    { romAddr = fromMaybe romAddr $ getLast _romAddr'
    , ramAddr = fromMaybe ramAddr $ getLast _ramAddr'
    , output = fromMaybe output $ getLast _output'
    }
```

What is annoying here is that there are no degrees of freedom in these definitions, yet we still needed to write them out:

- `PartialCPUOut` is a record type with exactly the same fields as `CPUOut`, except each field is renamed and their type is wrapped in `Last`.
- The `Semigroup` and `Monoid` instances for `PartialCPUOut` are simply lifting, field-by-field, the `Semigroup` and `Monoid` instances for `Last a`.
- Because `PartialCPUOut` is defined to match `CPUOut` exactly, `update` can only be written in one way, by pairing up the matching fields of the two datatypes.

Not only are we writing code that has no creativity left in it, but also we have to maintain it as `CPUOut` changes; for example, in the next section when we implement the `+`, `-` and `,` instructions, we will extend `CPUOUt` with a new field containing the write to RAM.

Instead, by using the clever library *Barbies*, and writing some generic functions once and for all, we will derive all of the above code just from the definition of `CPUOut`. To give a teaser, by the time we finish this section, we will be able to get rid of the definition of `PartialCPUOut`, its hand-written instances, and even `update`, without having to change `outputCell` and similar output-assigning functions.

The key to automating the definition of `PartialCPUOut` is to think of `CPUOut` as if it was parameterized by a functor that wraps all its field types:

```
data HKDCPUOut f = CPUOut
    { _romAddr :: f PC
    , _ramAddr :: f Ptr
    , _output :: f (Maybe Cell)
    }
makeLenses ''HKDCPUOut
```

Then, we can retrieve `CPUOut` and `PartialCPUOut` by setting `f` to `Identity` and `Last`, respectively. This pattern of datatype is called *higher-kinded data* (since the kind of `HKDCPUOut` is `(Type -> Type) -> Type`), hence the `HKD` in the datatype name `HKDCPUOut`. If we were to write `HKDCPUOut` by hand, one road bump would be that `HKDCPUOut Identity` would only be *isomorphic* to `CPUOut`, in that every field access would still have to go through the Identity / runIdentity wrapper/unwrapper. This is where the Barbies library comes to our help: instead of starting from HKDCPUOut and instantiating it to type CPUOut = HKDCPUOut Identity, we will *start* from CPUOut with its simple, unwrapped field types, and use Template Haskell macros provided by Barbies to turn it into a higher-kinded data type. Then, CPUOut Covered f will wrap every field in f while CPUOut Bare _ recovers the original CPUOut type with no wrapper on the field types. We will use the type variable b to denote these so-called "Barbie types" that can "change their clothes":

```
import Barbies
import Barbies.Bare
import Data.Barbie.TH

-- Definition of CPUOut exactly as before, enclosed in a TH invocation
declareBareB [d|
  data CPUOut = CPUOut
      { _romAddr :: PC
      , _ramAddr :: Ptr
      , _ramWrite :: Maybe Cell
      , _output :: Maybe Cell
      , _inputNeeded :: Bool
      } |]
makeLenses ''CPUOut

type Pure b = b Bare Identity
type Partial b = b Covered Last
```

This single-line definition of Partial CPUOut gives us a type that has the same fields as CPUOut, wrapped in Last; while Pure CPUOut is the same as our original CPUOut type, i.e. where all the fields are unwrapped.

And what about the Semigroup and Monoid instances? The Barbies library contains a simple newtype wrapper Barbie b f which has these instances implemented exactly how we need them – by using the instances of f a, field by field. So in our example, Barbie (CPUOut Covered) Last comes with a Monoid instance, making it ready to be used with the WriterT in CPU. We change the definition of Partial b to use this Barbie-wrapped version, for brevity in other definitions, and make our CPU type an instantiation of the more generic monad for CPU descriptions, which we name CPUM, to be re-used in further chapters:

```
type Partial b = Barbie (b Covered) Last

type CPUM s o = WriterT (Partial o) (State s)
type CPU = CPUM CPUOut CPUState
```

Moreover, the Barbies library allows us to zip the fields of a `b f` and a `b g` with some function `forall a. f a -> g a -> h a` into a `b h`; in particular, that allows us to implement `update` as a function of type `Pure b -> Partial b -> Pure b`. The complete code, while quite *complicated* (especially with all the Barbies-imposed typeclass constraints), doesn't really do anything *deep*. It is perfectly fine to just accept it as-is, shove it in a library, and be done with it:

```
update
    :: (BareB b, ApplicativeB (b Covered))
    => Pure b -> Partial b -> Pure b
update initials edits =
    bstrip $ bzipWith update1 (bcover initials) (getBarbie edits)
  where
    update1 :: Identity a -> Last a -> Identity a
    update1 initial edit = maybe initial Identity (getLast edit)
```

Now we have seen how to get rid of the hand-written `PartialCPUOut` datatype definition, its instance implementations, and `update`. However, there is still one piece missing: how do we change code, such as `outputCell`, to produce a `Partial CPUOut` instead of a `PartialCPUOut`? We change the definition of `(.:=)` to work with any Barbie type, as long as all its fields are monoids, by lifting the field lens to `Barbie` via an isomorphism:

```
infix 4 .:=
(.:=)
    :: (Applicative f, MonadWriter (Barbie b f) m)
    => Setter' (b f) (f a) -> a -> m ()
fd .:= x = scribe (iso getBarbie Barbie . fd) (pure x)
```

12.6.4 Writing to synchronous memory

In this section, we implement the instructions that directly change memory cells: the `+`, `-` and `,` instructions. In our hardware implementation, this means writing to synchronous memory.

The challenge here is that a write request to a `blockRam1` will only influence reads after a full cycle. Suppose we implemented `+` as simply setting an output pin `ramWrite` to `Just (nextIdx ramRead)`: in this case, the instruction immediately afterwards would still see the not-yet-incremented value on next cycle's `ramRead`. Starting with a cell value of 0, the effect of `+-..` would be to print 1 followed by 255, and leave the cell value at 255:

Cycle	Instruction	`ramRead`	`ramWrite`	`output`
0.	+	0	1	-
1.	-	0	255	-
2.	.	1	-	1
3.	.	255	-	255

We could solve this by keeping an internal register in `CPUState` that caches the current cell's value: `+` would not only assign `nextIdx cache` to the `ramWrite` output, but also change the `cache` register. This would require careful invalidation of the cache whenever the pointer changes (i.e. in `<` and `>` instructions).

Here, we go for another solution: we simply give enough time for the memory to finish its write operation, before consuming its read output. We do this by adding a `WaitWrite` phase, and entering it whenever we set `ramWrite`:

```
data Phase
    = WaitWrite
    | ...

declareBareB [d|
  data CPUOut = CPUOut
      { ...
      , _ramWrite :: Maybe Cell
      } |]

writeCell :: Cell -> CPU ()
writeCell x = do
    ramWrite .:= Just x
    phase .= WaitWrite
```

We then handle `WaitWrite` in every execution `step` by going back to `Exec`. We can think of `WaitWrite` as a counter to 1, the delay needed to let the RAM catch up with the writes. The new branches of `step` to implement cell increment and decrement instructions `+` and `-` is as follows:

```
step :: CPUIn -> CPU ()
step CPUIn{..} = use phase >>= \case
    Exec -> fetch >>= \case
        '+' -> writeCell $ nextIdx ramRead
        '-' -> writeCell $ prevIdx ramRead
        ...
    WaitWrite -> phase .= Exec
    ...
```

To implement the , input instruction, we add a new output pin to signal our need for a new input, and then start waiting for a Just value from the peripherals. When it arrives, we proceed as in + and - by setting the memory write pin and entering the WaitWrite phase, as implemented in writeCell:

```
data Phase
    = WaitInput
    | ...

declareBareB [d|
  data CPUOut = CPUOut
      { ...
      , _inputNeeded :: Bool
      } |]

data CPUIn = CPUIn
    { ...
    , input :: Maybe Cell
    }

startInput :: CPU ()
startInput = do
    inputNeeded .:= True
    phase .= WaitInput
```

Accordingly, the new branches of step are as follows:

```
step :: CPUIn -> CPU ()
step CPUIn{..} = use phase >>= \case
    Exec -> fetch >>= \case
        ',' -> startInput
        ...
    WaitInput -> traverse_ writeCell input
    ...
```

12.6.5 Control flow

The only remaining instructions to implement are the looping constructs [and]. Our software bytecode interpreter version already showed what we need to do in terms of program counter maintenance, and the version with external memory also showed how to use a small array of program addresses as a stack. We can import that solution wholesale into the hardware implementation by adding the stack to CPUState, and introducing a new Phase for skipping ahead to a matching]. Because this rounds off our CPU implementation, it is perhaps instructive to include the full code here, not just the new constructors and new branches:

```
data CPUIn = CPUIn
    { romRead :: Word8
    , ramRead :: Cell
    , outputAck :: Bool
    , input :: Maybe Cell
    }

declareBareB [d|
  data CPUOut = CPUOut
      { _romAddr :: PC
      , _ramAddr :: Ptr
      , _ramWrite :: Maybe Cell
      , _output :: Maybe Cell
      , _inputNeeded :: Bool
      } |]
makeLenses ''CPUOut

data Phase
    = Init
    | Exec
    | Skip (Index StackSize)
    | WaitWrite
    | WaitOutput
    | WaitInput
    | Halt
    deriving (Show, Generic, NFDataX)

data CPUState = CPUState
    { _phase :: Phase
    , _pc :: PC
    , _stack :: Stack StackSize PC
    , _ptr :: Ptr
    }
    deriving (Generic, NFDataX)
makeLenses ''CPUState

initCPUState :: CPUState
initCPUState = CPUState
    { _phase = Init
    , _pc = 0
    , _stack = Stack (repeat 0) 0
    , _ptr = 0
    }
```

```
pushPC :: CPU ()
pushPC = do
    pc <- use pc
    stack %= push (pc - 1)

popPC :: CPU ()
popPC = do
    (pc', stack') <- uses stack pop
    pc .= pc'
    stack .= stack'

step :: CPUIn -> CPU ()
step CPUIn{..} = use phase >>= \case
    Halt -> return ()
    Init -> phase .= Exec
    Skip depth -> fetch >>= \case
        '[' -> phase .= Skip (depth + 1)
        ']' -> phase .= maybe Exec Skip (predIdx depth)
        _ -> return ()
    Exec -> fetch >>= \case
        '>' -> ptr %= nextIdx
        '<' -> ptr %= prevIdx
        '+' -> writeCell $ nextIdx ramRead
        '-' -> writeCell $ prevIdx ramRead
        '.' -> outputCell ramRead
        ',' -> startInput
        '[' -> if ramRead /= 0 then pushPC else phase .= Skip 0
        ']' -> popPC
        '\0' -> phase .= Halt
        _ -> return ()
    WaitWrite -> phase .= Exec
    WaitOutput -> when outputAck $ phase .= Exec
    WaitInput -> traverse_ writeCell input
  where
    fetch = do
        pc += 1
        return $ ascii romRead
```

Note that we use `fetch` both in the `Skip` and the `Exec` phases, since in both we want to look at the current instruction and increment the program counter stored in the `pc` register. When `Exec` encounters an end-of-loop `]` instruction, `popPC` changes the `pc` register again. We are changing one register twice in a clock cycle, is that something that actually makes sense in hardware?

The answer is yes, because `step` is not describing a circuit directly. Note the type of `step` – once all the newtype wrappers of `WriterT` and `State` are removed, it is isomorphic to a **pure function** of the following type:

```
step :: CPUIn -> CPUState -> (Partial CPUOut, CPUState)
```

No signals or bona-fide registers anywhere! When `mealyState` then turns it into a signal function of type `Signal dom CPUIn -> Signal dom CPUOut`, everything happening inside `step` will happen in one clock step (as the name implies). If we did something like:

```
foo = do
    reg1 .= x
    ...
    y <- f <$> use reg1
    ...
    reg1 .= y
```

then it corresponds to a circuit where the field `reg1` of the state register is updated directly to `y`, with the value `x` fed to the computation of `f`. In other words, computations involving `use reg1` at various points of `foo` are all compiled into a combinational circuit of just the initial value of `reg1` (and other fields of the state register they might use), and the final value assigned to `reg1` is what gets written back to the state register at the end of the clock cycle.

And so we have our complete `step` function, which, via `cpuMachine` and `mealyState`, yields the full CPU of the following type:

```
cpu
    :: (HiddenClockResetEnable dom)
    => Signal dom CPUIn
    -> Signal dom CPUOut
cpu = mealyState cpuMachine initState
```

12.7 High-level simulation of the CPU

Before we move on to implementing other parts of our complete Brainfuck computer, let's pause and enjoy what we have so far, by putting up enough of a support structures around it that we can simulate the execution of Brainfuck programs.

Our `cpuMachine` function computes the CPU's output from its input (hence the name), with `State CPUState` being its only effect. Notably, it is not a signal function:

```
cpuMachine :: CPUIn -> State CPUState (Pure CPUOut)
```

This means that, similar to previous designs such as the desktop calculator or the game of Pong, we managed to arrange things such that we can interact with it from "normal" (i.e. software-oriented) Haskell. We start with writing the counterpoint of `cpuMachine`: a function that consumes the `CPUOut`, and produces the next cycle's `CPUIn`. This process uses the IO and memory effects from `MonadBFIO` and `MonadBFMemory`, respectively, to implement "the world outside the CPU":

```
world :: (MonadBFMemory m, MonadBFIO m) => Pure CPUOut -> m CPUIn
world CPUOut{..} = do
    romRead <- readProgROM _romAddr
    ramRead <- readRAM _ramAddr
    input <- if _inputNeeded then Just <$> doInput else return Nothing
    traverse_ (writeRAM _ramAddr) _ramWrite
    traverse_ doOutput _output
    outputAck <- return True

    return CPUIn{..}
```

Of particular note here is the ordering of the RAM effects. By running `readRAM` before `writeRAM`, we ensure that our semantics matches that of Clash's `blockRAM1`: write requests from a given cycle only influence reads after the next cycle. The other effects are quite straightforward. Since we apply `doOutput` on `_output` as soon as it is available, there is no need to do any buffering: we can keep `outputAck` at `True` to signal that every output request is immediately processed.

We then connect `cpuMachine` and `world` by combining their effects and also storing the `CPUIn` between cycles in our state:

```
simulateCPU :: (MonadBFMemory m, MonadBFIO m) => StateT (CPUIn,
    CPUState) m ()
simulateCPU = do
    (inp, s) <- get
    let (out, s') = runState (cpuMachine inp) s
    inp' <- lift $ world out
    put (inp', s')
```

Our `main` function then can peel off the effects into `IO` one by one. We already have `runBFVec` to run a `MonadBFMemory`-using computation by using vectors to represent program ROM and cell RAM. The `CPUIn` component of the state can be initialized with some "neutral" field values – we have already seen that in a real circuit, the output of the memory components is undefined in the first cycle, and we have arranged for it not to be a problem by adding the `Init` phase.

```
main :: IO ()
main = do
    prog <- prepareIO

    runBFVec (stringToROM $ loadVec prog '\0') $
        let initInput = CPUIn
                { romRead = 0
                , ramRead = 0
                , outputAck = False
                , input = Nothing
                }
        flip evalStateT (initInput, initCPUState) $ forever simulateCPU
```

Exercise:

- Test the "read-and-write" problem of the `+-..` program with the simulator. Remove the `WaitWrite` phase, and observe the change in output (it might be a good idea for this to change the `MonadBFIO` instance of `IO` to print cell values numerically). What happens if we also change the `world` so that writes happen before reads?

12.8 The logic board

If we imagine a computer built on various modular boards, in this section we turn our attention to the board containing all the logic elements: the CPU, the program ROM, and the cell RAM. This "board" has "connectors" corresponding to input and output from/to the outside world, via peripheral drivers. The two inputs correspond to the `input` and `outputAck` fields of `CPUIn`, while the two outputs are assigned by `inputNeeded` and `output` of `CPUOut`. All other pins of the CPU will be connected to other components on this same board.

```
logicBoard
    :: (HiddenClockResetEnable dom)
    => Signal dom (Maybe Cell)
    -> Signal dom Bool
    -> (Signal dom Bool, Signal dom (Maybe Cell))
```

The CPU consumes the next instruction's machine code from the program ROM, and produces the address of the subsequent instruction's address, so the connection between the CPU and the program ROM is necessarily circular. The same applies to the RAM holding the cell values: the currently selected cell's value is an input to the CPU, and its output contains the next cell's address and, optionally, its write value.

This is similar to circuits involving feedback on a single register: the recursion between the address signal and the data-out signal of the memory primitives is guarded by the memory primitive itself, ensuring well-formedness. We show the basic form of `logicBoard` first, before filling in the details:

```
logicBoard inputValue ack = (_inputNeeded <$> cpuOut, _output <$> cpuOut)
  where
    cpuOut = cpu cpuIn

    romRead = unpack <$> romFilePow2 "hello.rom" romAddr
    ramRead = blockRam1 NoClearOnReset (SNat @30_000) 0 ramAddr ramWrite
```

We use `romFilePow2` (as in, `romFile` with a power-of-2 size) instead of vanilla `romFile` because the former gives us better type inference: with `romFile`, we would have to specify the address bus width of 12 manually.

The `unpack` post-processing in `romRead` is needed because `romFilePow2` compiles to an HDL primitive that initializes the ROM contents from the file `"hello.rom"`, without making any assumptions on its data format. On the Clash side, we interpret the 8-bit values in `"hello.rom"` as `Word8`s, and then, subsequently, as `Char`s courtesy of our `ascii` function. For example, the first couple values for our Hello World program might look like this:

Address	`"hello.rom"`	`unpack @Word8`	`ascii`
0	00101011	0x2b	'+'
1	00001010	0x0a	'\n'
2	01011011	0x5b	'['
3	00001010	0x0a	'\n'
4	00100000	0x20	' '
5	00100000	0x20	' '
6	00101101	0x2d	'-'
7	00001010	0x0a	'\n'
8	00100000	0x20	' '
8	00100000	0x20	' '
10	01011011	0x5b	'['

The full program is the same as before, deliberately exercising our implementation of ignoring non-significant characters (anything other than `><+-.,[]`), by showing structure using whitespace.

```
+
[
  -
  [
    <<
    [ + [--->] - [<<<] ]
  ]
  >>>-
]
>-.---.>..>.<<<<-.<+.>>>>>.>.<<.<-.
```

We turn this into a binary `.rom` file and pass that to `romFile` because that is what downstream synthesis tools understand – the HDL emitted by Clash doesn't contain the data, it only contains the filename `hello.rom`. One benefit of this is that, conditional on our synthesis toolchain supporting this, we can replace the contents of `hello.rom` and avoid re-synthesizing everything else.

We can convert a list of bytes into the file format used by `romFile` by simply going printing each line in binary. The size has to match the ROM size exactly, which we can ensure (if needed) by padding with 0 bytes:

```
binLines :: Maybe Int -> [Word8] -> [String]
binLines size bs = map (printf "%08b") bytes
  where
    bytes = maybe id ensureSize size bs
    ensureSize size bs = take size $ bs <> repeat 0x00
```

This leaves us with the following definitions in `logicBoard` to be written:

- Filling all the fields of `cpuIn`, from `logicBoard`'s arguments and the outputs of the memory elements
- Extracting the right address inputs for the `blockRam1` and the `romFile`

The first part benefits a lot from the synergy of the `ApplicativeDo` and `RecordWildCards` language extensions to connect signals to each field of `CPUIn`:

```
logicBoard inputValue ack = ...
  where
    ...
    cpuIn = do
        romRead <- romRead
        ramRead <- ramRead
        outputAck <- ack
        input <- inputValue
        pure $ CPUIn{..}
```

The memory address and memory write signals are similarly extracted from the fields of `cpuOut`. Note that Clash's `blockRam1` primitive is a true dual-port RAM: the write address (in `ramWrite`) can be specified independently of the read address (in `ramAddr`). Our Brainfuck CPU, however, has a single address output that is used both to read from and to write to memory, so we pair it with the value to be written (if it is a `Just` value):

```
logicBoard inputValue ack = ...
  where
    ...
    romAddr = _romAddr <$> cpuOut
    ramAddr = _ramAddr <$> cpuOut
    ramWrite = packWrite <$> ramAddr <*> (_ramWrite <$> cpuOut)

packWrite :: addr -> Maybe val -> Maybe (addr, val)
packWrite addr val = (addr,) <$> val
```

This pattern of merging an address line (used for both reading and writing) and an optional write line will be a common one in this book, because it matches the interface of the hardware memory elements used in the real-life computers we will be interested in, to the dual-port block RAM components of FPGAs.

12.8.1 Record syntax via Barbies

If we look at the whole of `logicBoard`, one thing to notice is that the majority of its code is just shoveling signals in and out of `CPUIn` and `CPUOut`, compared to the useful part of defining the memory elements and their connections to the CPU pins:

```
logicBoard inputValue ack =
    (_inputNeeded <$> cpuOut, _output <$> cpuOut)
  where
    cpuOut = cpu cpuIn

    romRead = unpack <$> romFilePow2 "hello.bf" romAddr
    ramRead = blockRam1 NoClearOnReset (SNat @30_000) 0 ramAddr ramWrite

    cpuIn = do
        romRead <- romRead
        ramRead <- ramRead
        outputAck <- ack
        input <- inputValue
        pure $ CPUIn{..}
    romAddr = _romAddr <$> cpuOut

    ramAddr = _ramAddr <$> cpuOut
    ramWrite = packWrite <$> ramAddr <*> (_ramWrite <$> cpuOut)
```

This is because `cpuIn` and `cpuOut` are both signals containing records. If, instead, they were records containing signals, we could refer to their fields directly instead of picking them out one by one with `(<$> cpuOut)` and, conversely, assembling them into `cpuIn` by binding each signal's value separately. If we had them as records of signals, we could use the field names directly in conjunction with record wildcard syntax, leading to a much more lightweight implementation:

```
logicBoard input outputAck = (_inputNeeded, _output)
  where
    CPUOut{..} = cpu CPUIn{..}

    romRead = unpack <$> romFilePow2 "hello.bf" _romAddr
    ramRead = blockRam1 NoClearOnReset (SNat @30_000) 0 _ramAddr write
    write = packWrite <$> _ramAddr <*> _ramWrite
```

However, this puts `cpu` into a tight spot: if its inputs and outputs are records of signals, what are the pure record type to use in `cpuMachine` and `step`?

What we are looking for, then, is a way to `bundle` the `CPUIn` argument before passing it to `mealyState`, and then `unbundle` its `CPUOut` result. Before we can start thinking about how to implement these instantiations of `bundle` and `unbundle`, we first need to know what their types should be. We want `bundle` to go from a record where each field's type is wrapped in `Signal dom`, to a `Signal dom` containing a pure record.

Sounds like a job for Barbies: we already know that if `b` is a Barbie-type, we can use `Signal dom (Pure b)` to describe the type of a signal containing a pure record. Similarly, `b Covered (Signal dom)` is the type where every `a`-typed field becomes a `Signal dom a`:

```
type Signals dom b = b Covered (Signal dom)

bbundle :: _ => Signals dom b -> Signal dom (Pure b)
```

The constraint hole in the type of `bbundle` is to emphasize the fact that we haven't figured out yet what exactly we'll require of `b`: that will follow from the implementation. We implement `bbundle` in two steps: first, for any appropriate Barbie-type, we can collect all effects from the fields with the Barbies equivalent of the `Traversable` typeclass's `sequence`, called `bsequence`. We'll use its version `bsequence'` specialized for returning pure (`Identity`-wrapped) fields:[4]

```
bsequence' :: _ => b Covered f -> f (b Covered Identity)
```

Almost there, except we also want to get rid of the `Identity` wrappers in the result type. We can compose this with `bstrip` which removes the cover of a Barbie-type:

```
bstrip :: _ => b Covered Identity -> Pure b
```

Filling in the details of the typeclass constraints, we get the following definition; turns out there is no reason to restrict ourselves to `Signal dom`:

```
bbundle
    :: (Applicative f, BareB b, TraversableB (b Covered))
    => b Covered f
    -> f (Pure b)
bbundle = fmap bstrip . bsequence'
```

To go in the other direction, we need the ability to push the effect of creating a pure value into the individual fields. The opposite of `bsequence'` is the Barbies equivalent of `distribute`, again specialized to `Identity`:

```
bdistribute' :: _ => f (b Covered Identity) -> b Covered f
```

Which we can combine with `bcover` into the following function:

```
bunbundle
    :: (Functor f, BareB b, DistributiveB (b Covered))
    => f (Pure b)
    -> b Covered f
bunbundle = bdistribute' . fmap bcover
```

Of course, we can also package the pair of functions `bbundle` and `bunbundle` into a proper `Bundle` instance:

```
instance
  (BareB b, TraversableB (b Covered), DistributiveB (b Covered)) =>
  Bundle (Pure b) where
```

[4] Strictly speaking, in the code used in this chapter and the remainder of the book, `b Covered` is a Barbie-type, not `b`. We use the Barbies library this way so that we also have access to `b Bare Identity`, where we don't have to bother with the `Identity` wrapper around fields. The Barbies type signatures shown in this chapter have all been specialized for `b Covered`.

```
    type Unbundled dom (Pure b) = Signals dom b
    bundle = bbundle
    unbundle = bunbundle
```

Armed with these definitions, and wrapping the definition of the `CPUIn` datatype in `declareBareB` just like `CPUOut`, we can change `cpu` to expose a `Signals` interface to `logicBoard` (and making sure we use `Pure CPUIn` everywhere else):

```
cpu
    :: (HiddenClockResetEnable dom)
    => Signals dom CPUIn
    -> Signals dom CPUOut
cpu = mealyStateB cpuMachine initState
```

With this definition of `cpu`, we can replace `logicBoard`'s implementation with the more concise one that uses record wildcards.

12.9 Low-level simulation of the logic board

So far, we have always opted for high-level simulation: taking non-signal functions and running them in a software Haskell context. We cannot apply the same approach to `logicBoard`, because it is fundamentally about *connecting signals*. The high-level CPU simulator has already shown how to simulate its main interesting component, the CPU itself; but what if we want to simulate how that CPU is hooked up to the other components?

To motivate this, we can think back to the "read-and-write" problem of the program `+-...` We haven't really *discovered* that problem on our own – instead, we arrived at it from first principles, reasoning about the behavior of Clash's `blockRAM1` primitive. Maybe it even felt like *reading it in a book* written by someone in advance. Sure, the CPU simulator would have shown this problem, but that is only because we took care to use the `MonadBFMemory` primitives in such a way that they model `blockRAM1`. If, instead, we start from the full `logicBoard`, we can observe and test the behaviour of `blockRAM1` itself, in conjunction with the full circuitry connecting `blockRAM1`, `romFilePow2` and the CPU.

There are two problems we need to address to be able to do this:

- Specific to our Brainfuck computer, we need a way to **configure the ROM contents**. Currently, it is hardcoded to take it from the file `hello.rom` which we arrange to contain the Hello World program in the appropriate binary image format. This makes sense for hardware: the contents of ROM chips is set in stone once synthesis finishes. But for the simulation, we'd like the same flexibility as all the previous software implementations and the high-level simulator.

- Generally, have a signal-to-signal function that we would like to simulate *interactively*. For example, one of the arguments to `logicBoard` is a signal containing the latest input from the peripheral controllers; but we need to present previous outputs (from other signals in the result of `logicBoard`).

We solve the first problem in two steps: adding a parameter to `logicBoard` for the filename of the binary image dump of the ROM contents, and then in `main`, creating a temporary file in the right format from the text file input. The change to `logicBoard` is straightforward:

```
logicBoard
    :: (HiddenClockResetEnable dom)
    => FilePath
    -> Signal dom (Maybe Cell)
    -> Signal dom Bool
    -> (Signal dom Bool, Signal dom (Maybe Cell))
logicBoard programFile inputValue ack = ...
  where
    romRead = unpack <$> romFilePow2 programFile romAddr
    ...
```

To actually create `programFile`, we use the *Temporary* library to create-and-open a new file in the host system's directory for temporary files, write the program string to it using `binLines`, and then close the write handle before passing its (randomized) file name to `logicBoard`.

```
import System.IO
import System.IO.Temp

main :: IO ()
main = withSystemTempFile "brainfuck-.rom" $ \romFile romHandle -> do
    prog <- stringToROM <$> prepareIO
    hPutStr romHandle . unlines $
      binLines (Just (snatToNum (SNat @ProgSize))) prog
    hClose romHandle

    let board = logicBoard romFile
    ...
```

Here, `withSystemTempFile "brainfuck-.rom"` generates filenames like `/tmp/brainfuck-26160-0.rom` during runtime, atomically picking a fresh file and opening it. It also takes care of deleting the file on exit. Note the use of `hClose` before handling `romFile` to `logicBoard`: otherwise, Clash will fail trying to simulate the `romFilePow2` inside `board`, since the file it tries to load is locked.

For the second problem, we are looking for a way to connect a signal function of type `Signal dom i -> Signal dom o` to the outside world modeled as `o -> IO i`. In earlier chapters, we used the `sample` interface to Clash's simulator, which turns an output `Signal` into a list of values. To feed the input values into the input signal elementwise, observing the output signal's values in the process, we can use a different interface: `signalAutomaton`.

The idea behind `signalAutomaton` is to turn a `Signal dom i -> Signal dom o` signal function into a stateful function producing an `o` and a new function from the next `i` in one step. The `Automaton` type instantiated by Clash is imported from the *Arrows* package; we reproduce its definition and `signalAutomaton`'s type signature below (with suggestive type variable names) to show the building blocks we are going to use:

```
newtype Automaton arr i o = Automaton (arr i (o, Automaton arr i o))

signalAutomaton
    :: (KnownDomain dom)
    => (HiddenClockResetEnable dom => Signal dom i -> Signal dom o)
    -> Automaton (->) i o
```

What we need to do is wedge our IO function in there somehow: given some initial input `i0`, we want to run the automaton's step function to get the first output `o0`, which we can then pass to the `world` to get the next input `i1`, and so on. To allow stepwise execution, we can keep the current automaton (i.e. the next `step`) in a mutable reference updated at each call. At its simplest version, we could do it like the following:

```
simulateIO
    :: (KnownDomain dom)
    => (HiddenClockResetEnable dom => Signal dom i -> Signal dom o)
    -> i
    -> (o -> IO i)
    -> IO (IO ())
simulateIO circuit input0 world = do
    let Automaton step = signalAutomaton circuit
    ref <- newMVar $ step input0
    return $ do
        (out, Automaton step) <- liftIO $ takeMVar ref
        (input, result) <- world out
        liftIO $ putMVar ref $ step input
```

We can refine this design a bit by allowing simulation in any `MonadIO`, allowing the `world` to be replaced for every simulation step, and getting a stepwise result more informative than `()`:

```
simulateIO
    :: (KnownDomain dom, MonadIO m)
    => (HiddenClockResetEnable dom => Signal dom i -> Signal dom o)
    -> i
    -> IO ((o -> m (i, a)) -> m a)
simulateIO circuit input0 = do
    let Automaton step = signalAutomaton circuit
    ref <- newMVar $ step input0
    return $ \world -> do
        (out, Automaton step) <- liftIO $ takeMVar ref
        (input, result) <- world out
        liftIO $ putMVar ref $ step input
        return result
```

As usual for functions where sometimes the effects are all that matter, we recover a version of `simulateIO` that has no return value:

```
simulateIO_
    :: (KnownDomain dom, MonadIO m)
    => (HiddenClockResetEnable dom => Signal dom i -> Signal dom o)
    -> i
    -> IO ((o -> m i) -> m ())
simulateIO_ circuit input0 = do
    sim <- simulateIO circuit input0
    return $ \world -> do
        void $ sim $ \output -> do
            input <- world output
            return (input, ())
```

We finish our low-level logic board simulator by just running the interactive simulation `forever`. Since the simulated circuit now includes the memory elements, only the IO effects remain to be implemented by `world`. Intuitively, it makes sense: if more parts are inside the boundary of the simulation, the world outside that boundary shrinks by that much.

```
world :: (MonadBFIO m) => (Bool, Maybe Cell) -> m (Maybe Cell, Bool)
world (inputNeeded, output) = do
    traverse_ doOutput output
    input <- if inputNeeded then Just <$> doInput else return Nothing
    return (input, True)
```

```
main :: IO ()
main = withSystemTempFile "brainfuck-.rom" $ \romFile romHandle -> do
    prog <- stringToROM <$> prepareIO
    hPutStr romHandle . unlines $
      binLines (Just (snatToNum (SNat @ProgSize))) prog
    hClose romHandle

    sim <- simulateIO_ @System
           (bundle . uncurry (logicBoard romFile) . unbundle)
           (Nothing, False)
    forever $ sim world
```

12.10 Top-level circuit and peripherals

To finish our Brainfuck computer, we need to implement the peripherals for input and output:

- A pushbutton will serve as an acknowledgment trigger for both inputs and output.
- We'll use a keypad to input 8-bit inputs as two hexadecimal digits; for example, we'll input 32_{16} as 3 2 Button. This also allows for correcting input; for example, the result of 4 3 2 Button is the same.
- A seven-segment display will show either the current output or the input under editing. To avoid confusing with the digits `0` and `1`, we'll display a lowercase `o` and `i` showing output or input mode.

Of course, most of the heavy lifting will be done with functions from earlier chapters: `driveSS` and `encodeHexSS` for the display, and `inputKeypad` for the hexadecimal keypad.

12.10.1 Input driver

The input driver connects to the pushbutton and the keypad, and returns the acknowledgment signal and the input buffer's value. The latter will be connected both to the logic board, for consumption; and the display driver, for feedback while editing.

```
inputs
    :: (HiddenClockResetEnable dom, _)
    => Signal dom (Active btn)
    -> Signal dom (Vec 4 (Active row))
    -> ( Signal dom (Vec 4 (Active col))
       , Signal dom Bool
       , Signal dom Cell
       )
```

Internally, the pushbutton input `btn` is debounced and fed to a rising edge detector so that its single presses can be used as the acknowledgment signal:

```
inputs btn rows = (cols, ack, buffer)
  where
    ack =
        isRising False $
        debounce (SNat @(Milliseconds 5)) False $
        fromActive <$> btn
    ... -- Continued below
```

For the buffer holding the input cell value, we need to keep a two-digit editing area that is updated from keypresses. Also, whenever the user enters a value (by pressing the button that fires the `ack` signal), we want to zero out the buffer so that the next input starts from a clean slate.

```
    (cols, key) = inputKeypad keymap rows
    buffer = bitCoerce <$> shiftInReg zero (enable ack $ pure zero) key
    zero = repeat 0x0
```

Here, we use the helper function `shiftInReg` to collect input events. We implement a shift-in register where we can load in a new vector whole-sale, or push a new element to the rightmost end of the vector. We use `muxA`, an *n*-ary version of `mux` for `Alternatives` such as `Maybe`, to prioritize `load` over `new` elements.

```
shiftInReg
    :: (KnownNat n, NFDataX a, HiddenClockResetEnable dom)
    => Vec n a
    -> Signal dom (Maybe (Vec n a))
    -> Signal dom (Maybe a)
    -> Signal dom (Vec n a)
```

```
shiftInReg initial load new = vec
  where
    vec = regMaybe initial $ muxA [load, shiftIn <$> vec <*> new]
    shiftIn current new = (current <<+) <$> new
```

Since the `Alternative` laws prescribe `(<|>)` to be associative, we can use the `Foldable` typeclass's `foldMap` method instead of restricting left- or right-association of the result with `foldl` or `foldr`. For certain choices of `Foldable`, most importantly for `Vec n`, this can result in more efficient simulation and/or a smaller circuit. The two helper types `Ap` and `Alt` are needed to lift the underlying `Alternative m` instance into a `Monoid (f (m a))`.

```
import qualified Data.Foldable as F
import Data.Monoid (Alt(..), Ap(..))

muxA
    :: (Foldable t, Alternative m, Applicative f)
    => t (f (m a))
    -> f (m a)
muxA = fmap getAlt . getAp . F.foldMap (Ap . fmap Alt)
```

Throughout the rest of the book, we will also make good use of other `Alternative` operators lifted to signals, so we define them here wholesale:

```
infixl 3 .<|>.
(.<|>.) :: (Applicative f, Alternative m) => f (m a) -> f (m a) -> f (m
    a)
(.<|>.) = liftA2 (<|>)

infix 2 .|>., |>., .<|., .<|

(.|>.) :: (Applicative f) => f a -> f (Maybe a) -> f a
(.|>.) = liftA2 fromMaybe

(|>.) :: (Applicative f) => a -> f (Maybe a) -> f a
x |>. fmx = fromMaybe x <$> fmx

(.<|.) :: (Applicative f) => f (Maybe a) -> f a -> f a
(.<|.) = flip (.|>.)

(.<|) :: (Applicative f) => f (Maybe a) -> a -> f a
(.<|) = flip (|>.)
```

Since we are using a hexadecimal keypad to input hexadecimal digits, the keymap is the “identity keymap” that maps every key to its own value:

```
keymap :: Matrix 4 4 (Unsigned 4)
keymap =
    (0x1 :> 0x2 :> 0x3 :> 0xa :> Nil) :>
    (0x4 :> 0x5 :> 0x6 :> 0xb :> Nil) :>
    (0x7 :> 0x8 :> 0x9 :> 0xc :> Nil) :>
    (0x0 :> 0xf :> 0xe :> 0xd :> Nil) :>
    Nil
```

12.10.2 Display driver

The display driver’s job is to show either the current output or the currently edited input on a seven-segment display. We are going to use three seven-segment digits: the first one will show a lowercase `i.` or `o.` to tell the user if it is waiting for input or showing output, and the next two are used for the two digits themselves. Consequently, our display driver will work for any display size that has at least three digits.

```
display
    :: (HiddenClockResetEnable dom, _)
    => Signal dom (Maybe Cell)
    -> Signal dom (Maybe Cell)
    -> Signal dom (SevenSegment (k + 3) anodes segments dp)
display output inbuf = driveSS displaySS (pad <$> chars)
  where
    chars = displayChars <$> output <*> inbuf
    pad xs = repeat Nothing ++ xs
```

The actual rendering is done by `displaySS`. We make an algebraic datatype for the types of seven-segment characters we want to support, and reuse `encodeHexDigitSS` for the case of hexadecimal digits. For other characters, we write a lowercase `i` and `o` by hand, and use the decimal digit separator for the `.`.

```
data SSChar
    = SSHex (Unsigned 4)
    | SSOutput
    | SSInput

ssI :: Vec 7 Bool
ssI = False :> False :> True :> False :> False :> False :> False :> Nil

ssO :: Vec 7 Bool
ssO = False :> False :> True :> True :> True :> False :> True :> Nil
```

```
displaySS :: SSChar -> (Vec 7 Bool, Bool)
displaySS (SSHex digit) = (encodeHexSS digit, False)
displaySS SSOutput = (ssO, True)
displaySS SSInput = (ssI, True)
```

12.10.3 The topEntity

We finish our Brainfuck computer by wiring together the logic board and the peripheral drivers in `topEntity`. Connections to the outside world consist of the acknowledgment button, the row and column pins of the keypad, and the seven-segment display.

```
topEntity
    :: "CLK"     ::: Clock System
    -> "BTN"     ::: Signal System (Active High)
    -> "ROWS"    ::: Signal System (Vec 4 (Active Low))
    -> ( "COLS"  ::: Signal System (Vec 4 (Active Low))
       , "SS"    ::: Signal System (SevenSegment 4 Low Low Low)
       )
topEntity = withResetEnableGen board
  where
    board btn rows = (cols, ss)
      where
       (cols, ack, inBuf) = inputs btn rows
       ... -- Continued below
```

At a high level, the internal connections needed are as follows:

- The input driver connects directly to the input peripherals.
- The logic board consumes the input driver's signals, and its output is fed to the display driver.
- The display driver is connected not only to the logic board, but also to the input driver. This is needed to be able to display the already entered digits of the current input.

The CPU produces output values for a single clock period, allowing peripherals to distinguish between outputting the same value once or several times. However, in our computer design, we want to display the current output until the user presses the acknowledgment button; so we need to use a buffer between the logic board and the display unit. Similarly, the CPU raises the `inputNeeded` signal for one clock period, but we want to display the input prompt until the acknowledgment button is pressed; so we need another buffer.

For the output buffer, we want a register that holds the Just values of the output signal and clears it when ack becomes True. Similarly, for the inputNeeded buffer, we want to hold True values until ack becomes True. We can generalize these two cases, and more, by writing a combinator to integrate any monoidal operation; and then instantiating it for the First and Any monoids. In integrate, we keep an accumulator in a register, and use the mappend of the monoid to refresh it in each period, until we get a request to clear the accumulated value.

```
integrate
    :: (Monoid a, NFDataX a)
    => (HiddenClockResetEnable dom)
    => Signal dom Bool -> Signal dom a -> Signal dom a
integrate clear x = acc
  where
    acc = register mempty $ mux clear x $ mappend <$> acc <*> x
```

Note that in the clear case, we reset to x, not mempty, to avoid missing the value of x while clear is asserted.

Using integrate, we define the two buffers by wrapping and then unwrapping into the right monoid newtypes:

```
outBuf = fmap getFirst . integrate ack . fmap First $ output
inputNeededBuf = fmap getAny . integrate ack . fmap Any $ inputNeeded
```

These two are, of course, fed from the output of logicBoard. The inputs to logicBoard are the output acknowledgment button, and the cell value input submitted by the user. Submission is also triggered by the ack button, so we gate the persistent input buffer inBuf with ack. We do this here instead of inputs because this gives us the flexibility to access inBuf in other contexts as well. By using the buffered value of inputNeeded, we can see when the user is expected to enter a new value; in this case, we can display the input buffer without also submitting it to the logicBoard:

```
(inputNeeded, output) = logicBoard "hello.rom" (enable ack inBuf) ack
ss = display outBuf (enable inputNeededBuf inBuf)
```

Exercises:

- Instead of i. and o., get creative with designing seven-segment fonts, and use three digits to display in and out.
- Implement IO over a serial connection, either replacing the keypad and seven-segment display, or working with it in conjunction.

12.11 Summary

- Brainfuck is a **programming language** that lends itself to a simple **RAM machine-based hardware implementation**.
- We started with an interpreter for a structured programming language, and refined it in multiple steps to a **bytecode interpreter with memory operations externalized**.
- The CPU is implemented using a `State/Writer` monad: the `State` part contains the **internal registers**, while the `Writer` aspect **assembles the output piecewise**. The *Barbies* library enables a seamless way of the latter.
- Memory access can necessitate **waiting phases**.
- We can write **simulations at different levels**: we can run the monadic CPU as a normal (non-`Signal`-level) Haskell function, or we can take the circuit consisting of the processor and the memory elements, and use Clash's `Signal` simulator interactively.
- **Peripherals can be much slower than the processor**, requiring flow control between them. In the Brainfuck machine, the CPU waits for acknowledgment on output, since otherwise the user would have no opportunity to read the seven-segment display.

13 CHIP-8

CHIP-8 is retrocomputing catnip: a virtual machine for video games from 1978, originally targeting a line of home computers sold in kit form. It is widely regarded as an excellent first goal for learning about emulation and virtual machines: on one hand, it is simple enough that naïve implementation approaches can work successfully; on the other hand, it is powerful and expressive enough that there are dozens of video games written for it, and some of those games are even enjoyable! Thus, it provides that ultimate satisfaction of implementing a machine based on a pre-existing specification: when the existing software corpus successfully works on our new implementation.

Although originally a virtual machine, in this chapter we build a hardware implementation: a computer with a CPU that natively uses the CHIP-8 bytecode format as its machine code.

13.1 History

The story of the CHIP-8 starts with the RCA 1802 microprocessor. Launched in 1974 as two chips working in tandem, then unified into a single chip in 1976, it was designed to be inexpensive. The goal of designer Joseph Weisbecker was to put it in homes, in the form of kit computers and video game consoles.

The COSMAC VIP was one such kit computer, designed by Weisbecker himself. Released in 1977, it consisted of an RCA 1802 CPU, an RCA CDP1861 video display chip, a 4×4 keypad, and 2 kB of RAM[1]. Overall, the VIP was a small and simplistic computer compared to its contemporaries like the 1977 "Holy Trinity" of the Commodore PET, the TRS-80 and the Apple II. At that time, "high-level programming" was synonymous with BASIC for home uses; the limitation of the memory size and a keypad instead of a full keyboard meant that it wasn't feasible to support BASIC on the VIP.

[1] See the August 1977 issue of *Byte* Magazine for an in-depth contemporary article on the COSMAC VIP, available at https://archive.org/details/byte-magazine-1977-08

Instead, it shipped with CHIP-8: dubbed *An Easy Programming System* by Weisbecker in his article in the December 1978 issue of *Byte* Magazine[2], it was aiming for a sweet spot of working with the limited resources of the COSMAC VIP and similar RCA 1802-based computers, but still providing enough high level constructs to empower hobbyist programmers to create fun games:

> This article describes a hexadecimal interpretive programming system which requires less hardware than high level languages such as BASIC, and which I feel is much easier to use than machine language. In my experience, hexadecimal interpretive programming is ideally suited to real time control, video graphics, games or music synthesis. It can be used with inexpensive computer systems consisting of a hexadecimal keyboard and only 1 K or 2 K of programmable memory. Expensive terminals and large memories aren't required. You can quickly and easily write useful programs that require five to ten times less memory than conventional high level languages without resorting to the tedious complexities of actual machine language.

This article, and the first issue of the COSMAC VIP owners' magazine *VIPER*[3] are both useful sources of information on the CHIP-8.

The second wave of CHIP-8's popularity came in 1990, when Andreas Gustafsson released *CHIP-48*, a CHIP-8 implementation for the HP 48 series of programmable graphing calculators[4]. Understandably, there wasn't much entertainment content for a calculator, so the CHIP-8 game library filled this gap without competition. Later forks of CHIP-48 also extended CHIP-8 with higher resolution graphics modes and new opcodes; here, we will stick to the original CHIP-8 and not cover these extensions.

These days, CHIP-8 is a popular platform as a learning project to understand writing an emulator. The library of games is also growing rapidly, in great part thanks to *Octojam*, an online game jam held annually every October[5]. There is even a higher-level compiler called *Octo*[6] supporting block-structured programming.

13.2 The CHIP-8 computer

Since some of the CHIP-8 opcodes directly interface with peripherals, a complete CHIP-8 computer is more than just a virtual CPU executing one instruction after the

[2] Available at https://archive.org/details/byte-magazine-1978-12-rescan

[3] Available at https://archive.org/details/viper_1_01/

[4] Available to this day from Gustafsson's home page at http://www.gson.org/freeware/

[5] See https://johnearnest.github.io/chip8Archive/ for an archive of Octojam entries

[6] Source code at https://github.com/JohnEarnest/Octo

other. Here, we look at the main features of each component; we will come back to them and discuss them in more detail as we implement them.

13.2.1 Processor

At the center of the CHIP-8 is the CPU. It uses the von Neumann architecture, i.e. the working memory and the program memory is in the same address space. Its design is heavily influenced not just by the underlying RCA 1802, but also by its goal of being an ergonomic platform for program development on the resource-constrained COSMAC VIP.

The CPU has 16 *general-purpose registers*, each one holding 8-bit values; all data arithmetic also operates on 8 bits. The address bus is 12 bits wide; beside the *program counter*, there is a secondary *pointer register* that can be used for indirect (computed) memory addressing. It also has an internal call stack of depth at least 12. This stack is managed internally by the virtual machine to implement the `CALL` and `RET` instructions.

The data bus is 8 bits wide, and instructions have a uniform two-byte length. Although the memory is addressed byte by byte, and instructions are fetched byte by byte, it is more accurate to say that each instruction consists of four half-bytes, or nybbles. This is to make it easier to enter the machine code on the original COSMAC VIP's hexadecimal keypad. For example, the four-nybble instruction `81A4` stands for `ADD V1, VA`: "add the value of register `A` to register `1` and set register `F` to the carry". Here, the general machine code format of the instruction is `8xy4`, where `8` selects arithmetic assignment, x and y are the target and source registers, and the final `4` selects the arithmetic operation (addition with carry-out, in our example).

Other instructions have components that take up multiple nybbles; for example, `6BE8` loads the one-byte constant `0xe8` into register `B` (`LD VB, 0xe8`). Since addresses are 12 bits wide, instructions that contain an address operand use three of the four nybbles to contain said address; for instance, `1DAE` is the jump instruction `JP 0xdae`.

13.2.2 Memory

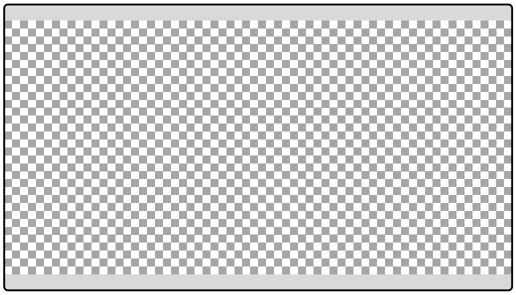
RAM is byte-addressed with the 12-bit address bus of the CPU. This would give us addresses from `0x000` to `0xfff` for a total of 4096 bytes, or 4 kilobytes. However, addresses below `0x200` are reserved for system use; we will use that address range to implement the `HEX` opcode, which loads a memory address that contains hexadecimal font glyphs.

It is worth noting that the original CHIP-8 implementation ran on machines with 1 or 2 kB of physical RAM, and that RAM had to also include the CHIP-8 interpreter itself, plus the video buffer, plus the real stack backing the CHIP-8's virtual in-CPU stack. So contemporary CHIP-8 games were not guaranteed access to RAM in the full address range; instead, programs started at `0x200` and the highest valid RAM address was implementation-dependent.

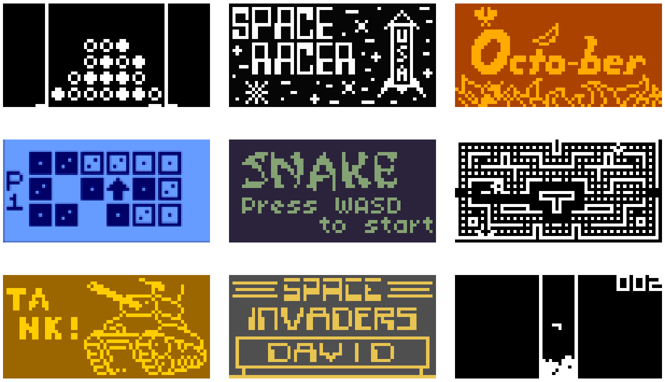
Of course these days, these limitations are no longer relevant, so we will populate the full `0x200..0xfff` range. Also, to simplify things, the video buffer will be stored entirely separate in a different block RAM element.

13.2.3 Graphics

Compared to their more sophisticated and pricier contemporaries, the most stunning user-visible limitation of the COSMAC VIP and similar computers was their graphical capabilities: 64 × 32 black-or-white pixels. Truly just black or white, not even grayscale: each pixel is either black, or white. And so, that is what we get in the CHIP-8 as well:



Creativity thrives under limitations, though, and there are some very clever CHIP-8 games that do a lot with this few pixels, as shown in this montage of various screenshots from https://johnearnest.github.io/chip8Archive/:



The CPU controls the video output by accessing a special piece of memory, the *video buffer*, through the instructions CLS and DRW. CLS stands for "CLear Screen" and changes the whole screen to black. DRW updates a user-specified rectangular area of the screen by combining a pattern stored in RAM with the current screen contents using pixel-by-pixel XOR. We make a quick note here that this means we'll need to *read* the video buffer as part of *writing* it; this point will become important later.

Although strictly speaking not part of the CHIP-8 graphics subsystem, we include here the program-accessible timer here as well. There is a special register in the CPU that is decremented until 0 at 60 Hz, with opcodes to read out and to change its value. Since the video output is also running at 60 frames per second, it makes sense to use the video timing generator to create that 60 Hz signal.

13.2.4 Keypad

The only input peripheral of the CHIP-8 is a 4 × 4 hexadecimal keypad in the following layout:

```
type Key = Nybble

layout :: Matrix 4 4 Key
layout =
    (0x1 :> 0x2 :> 0x3 :> 0xc :> Nil) :>
    (0x4 :> 0x5 :> 0x6 :> 0xd :> Nil) :>
    (0x7 :> 0x8 :> 0x9 :> 0xe :> Nil) :>
    (0xa :> 0x0 :> 0xb :> 0xf :> Nil) :>
    Nil
```

Note that this layout is different from the layout we used in previous chapters like the calculator or the Brainfuck computer. Since we had full control over the layout for those machines, we picked the one that is most common in real physical keypad peripherals that are easy to connect to FPGA development boards. For CHIP-8, however, we are bound to use the specified layout because a lot of games use the keys according to their location; for example, using 4 for moving left and D for moving right.

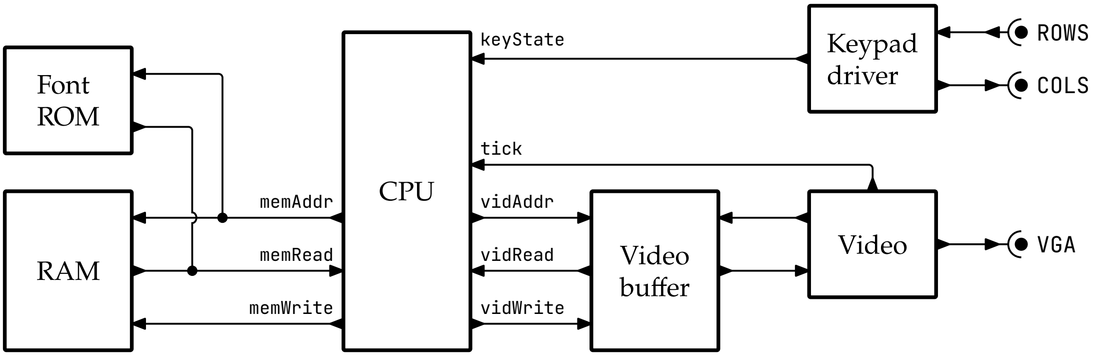
One interesting aspect of the keypad is that there are CHIP-8 opcodes both for checking the immediate state (pressed or released) of individual keys and also for waiting until the next key release. In real hardware, it is usually one or the other: if the keyboard matrix is directly exposed to the CPU's I/O pins, then it can directly check key state but software has to implement some logic to detect events; conversely, if the keyboard is connected via a serial interface (such as PS/2 or USB), the raw inputs are events and it is up to the software to reconstruct the key states at any given point in time.

In our implementation, we will use our keypad matrix sweeper to expose the immediate (debounced) key state to the CPU, and convert that into events when needed with the help of a special 16-bit register to detect state changes from one cycle to the next.

13.2.5 System overview

We can summarize the components and their connections in the diagram below. This diagram is based not just on the CHIP-8 specification, but also on the implementation design choices made so far:

- Font ROM and main RAM connected through address decoding logic
- Separate video buffer with dedicated connection to the CPU
- The keypad driver only provides the key states, not the key events



The CPU pin names used in the diagram correspond to the `CPUIn` / `CPUOut` record field names we will use throughout this chapter.

There is one oddity in this diagram: the video buffer's connections. Our Clash model of (synchronous) RAM allows one read and one write connection, but here we have one write and *two* reads from two different addresses: one from the CPU (driven by `DRW` instructions) and one from the video system (driven by the currently drawn X and Y coordinates). We will explore two solutions to this:

- First, we will **duplicate the video buffer and fan out the writes** to both copies. This way, the contents of both buffers will stay the same (the same writes are applied at the same time to both), but two different cells can be read concurrently by using different read addresses. This is a simple solution that doesn't require reengineering other parts of our design. Its drawback is that it requires twice the amount of block RAM.

- For the second solution, we will **resolve memory access contentions by prioritization**: if, in any given clock cycle, both the CPU and the video system needs access to the video buffer, the video system will "win" and the CPU will have to wait for its turn in another cycle.

The first solution is "local" in the sense that every other component can go on with its life pretending to be the only one accessing the video buffer. In contrast, the second solution requires changes to the CPU (to handle the situation where its video buffer read request isn't fulfilled) and the video system (to minimize video buffer access to avoid starving the CPU). Since we have a total of $64 \times 32 \times 1 = 2048$ bits, or just 256 bytes, of video buffer data, it wouldn't break the bank to store two copies of it. However, since we are here not just to build *a* computer, but also to learn about designing and building computers in general, we will explore the second approach as well, extra complications be damned.

13.3 Instruction set

Now that we have a general idea of the various components and how they fit together, let's look at the CHIP-8 instruction set. This will fill in the details needed to start moving from conceptual ideas to actual Clash code.

There are in total 35 instructions, each encoded in 16 bits. Below, we will use the metavariables x and y for registers (each one nybble), and nn and nnn for two- and three-nybble immediate values. The assembly syntax is based on Werner Stoop's open source CHIP-8 assembler[7], where registers are written with a `V` prefix, so register 0 is written as `V0` and so on.

Below are only 34 descriptions; there is one pseudo-instruction `SYS nnn` (`0nnn`) that jumps to the **host** machine code routine at address nnn. This was the escape hatch to native RCA 1802 code for performance reasons or to access hardware not exposed to the CHIP-8. Of course, modern CHIP-8 games don't use this instruction; like many other emulators, we are going to ignore it because the only way to implement it would be to implement a full RCA 1802 and all the parts of the COSMAC VIP.

13.3.1 Control flow

JP nnn (1nnn)

This is your usual unconditional jump instruction. The three-nybble nnn argument is the absolute address to jump to.

[7] Available from https://github.com/wernsey/chip8/

`JP V0,nnn` (Bnnn)

This variation of `JMP` adds the contents of the register `V0` to the address before jumping to it. This functionality is available for `V0` only: there is no corresponding instruction to take the offset from other registers.

`CALL nnn` (2nnn)

Similar to `JMP`, but pushes the program counter to the internal stack before jumping. It is meant to be used in conjunction with `RET`.

`RET` (00EE)

The pair of `CALL`: pops the program counter from the internal stack, thereby returning from the current subroutine.

`SE Vx,nn` (3xnn) and `SNE Vx, nn` (4xnn)

Skip the next instruction if the value of register *x* is equal (or not equal) to *nn*. There are other versions of these instructions below, for different values to compare against (see below); but in all cases, the only branching available is skipping one instruction.

`SE Vx,Vy` (5xy0) and `SNE Vx, Vy` (9xy0)

Similar to the previously described pair of instructions, these ones skip the next instruction if the values of registers *x* and *y* are equal (or not equal) to each other.

13.3.2 Register manipulation

`LD Vx,nn` (6xnn)

Set the value of the *x* register to the 8-bit value *nn*.

`ADD Vx,nn` (7xnn)

Update the value of the *x* register by adding the 8-bit value *nn*. The result is simply wrapped around in case of overflow.

`MOV Vx,Vy` (8xy0)

Copy the value of the *y* register to the *x* register

`LD I,nnn` (`Annn`)

Set the value of the pointer register (called `I` in mnemonics) to the given 12-bit value.

`ADD I,Vx` (`Fx1E`)

The 8-bit value of the x register is 0-extended to 12 bits, and added to the pointer register.

`LD [I],Vx` (`Fx55`) and `LD Vx, [I]` (`Fx65`)

These two instructions copy multiple values between registers and RAM. The first one, `LD [I] Vx`, writes the values of the first x registers to the memory locations starting at the pointer register (so `V0`'s value is written the address stored in `I`, `V1`'s value is written to the next address and so on). The second one, `LD Vx, [I]`, reads the memory locations starting from the pointer register and sets the first x register accordingly.

In the original CHIP-8 specification, `I` is incremented after each register is saved or restored, so that after these instructions run, the pointer points at the next memory address. However, most second-wave CHIP-8 emulators didn't implement this (leaving the pointer register's value unchanged), leading to some games that outright break if the correct behavior is implemented.

13.3.3 Arithmetic

`OR Vx,Vy` (`8xy1`), `AND Vx,Vy` (`8xy2`), and `XOR Vx,Vy` (`8xy3`)

Set the x register to the result of applying the given binary Boolean function, bitwise, to x and y.

`ADD Vx,Vy` (`8xy4`), `SUB Vx,Vy` (`8xy5`), and `SUBN Vx,Vy` (`8xy7`)

Set the x register to $V_x + V_y$ (`ADD`), $V_x - V_y$ (`SUB`) or $V_y - V_x$ (`SUBN`) using 8-bit arithmetic. The `VF` register is also changed: for `ADD`, it is set to 1 if there is carry (i.e. if the result doesn't fit into 8 bits) and 0 otherwise. For `SUB` and `SUBN`, `VF` is set to 0 if there is borrow (i.e. the result is negative) and 1 otherwise.

`SHR Vx,Vy` (`8xy6`) and `SHL Vx,Vy` (`8xyE`)

Set the x register to the right- or left-shifted value of the y register, and set `VF` to the least (or most) significant bit of y.

13.3.4 Special built-in functions

`RND Vx,nn` (`Cxnn`)

Generate a one-byte random number, and store it in the *x* register, using the one-byte *nn* argument as a bitwise-AND mask. The details of this random number generator is unspecified; we will implement it using a linear-feedback shift register.

`BCD Vx` (`Fx33`)

Store the binary-coded decimal representation of *x*'s value in memory. Since registers are 8-bit wide, its value is between 0 and 255, requiring three decimal digits. These three digits are 0-extended to three full bytes and written to the three memory locations starting at `I`.

13.3.5 Graphics

`CLS` (`00E0`)

Clear the screen, i.e. overwrite the video buffer's contents with all pixels unset.

`DRW Vx,Vy,n` (`Dxyn`)

Draw an $8 \times n$ sprite to the screen. The sprite data is taken from the *n* bytes starting at `I`; each byte describes one 8-pixel row (first byte being the topmost row). The screen position to draw is taken from the contents of the *x* and *y* registers.

The sprite data is then combined with existing video buffer contents via bitwise XOR. As a simple form of collision detection, the `VF` register is updated to 1 if this causes any pixels to change from set to unset, and to 0 otherwise.

The registers *x* and *y* are 8-bit values, but the screen resolution is 64×32. Overhanging *x* and *y* values simply wrap around; however, sprite data doesn't. For example, drawing from $x = 253$ means the sprite starts in column 61 and only the first three columns are drawn.

`HEX Vx` (`Fx29`)

Change the pointer register's value to an implementation-dependent value that can be used as an 8 row tall sprite with `DRW` that is the hexadecimal digit stored in the four lowest bits of register *x*. Since CHIP-8 programs start at address `0x200`, it is customary to use addresses below that to represent these digits.

13.3.6 Peripheral access

`SKP Vx` (`Ex9E`) and `SKNP Vx` (`ExA1`)

Take the lower four bits of register *x*'s value, and skip the next instruction if that key on the keypad is pressed (or not pressed).

`LD Vx,K` (`Fx0A`)

Wait for a key to be pressed, and load its value to register *x*. Although not included in the specification, the original implementation actually waited for the next key **release**, so we'll stick to that.

`LD DT,Vx` (`Fx15`) and `LD Vx,DT` (`Fx07`)

Set the delay timer's value from register *x* and vice versa. The timer counts down to 0 at 60 Hz. Note that polling (with `LD Vx, DT`) is the only way to use the timer, there are no interrupts or callbacks available to run code when 0 is reached.

`LD ST,Vx` (`Fx18`)

Set the sound timer's value from register *x*. The sound timer works exactly like the delay timer, and turns on an external buzzer while its value is non-zero. Because audio is outside of our scope here, we will implement this as a no-op. Not much is lost, because there is no way to change the pitch, let alone the timbre, of the sound: either it is on, or off.

13.3.7 Instruction decoding

We will use an algebraic datatype to represent the result of instruction decoding. The datatype exploits some of the regularity between instructions which will help in the implementation: for example, all binary arithmetic operations can be implemented as simply updating *x* and `VF` with the value of a pure function applied on *x* and *y*'s values; so the decoded representation of all these instructions will be the same, parametrized by the arithmetic function to use. Similarly, we unify the "skip-if" and the "skip-if-not" versions of branching instructions by storing the Boolean value that indicates a skip.

First, we define some type synonyms for one, two, and three-nybble values. As earlier, we pick `Word8` instead of `Unsigned 8` as the representation for bytes; they are the same in the HDL output, but simulation performance is much better for `Word8`.

```
type Nybble = Unsigned 4
type Byte = Word8
type Addr = Unsigned 12
type Reg = Nybble
```

Then, we map each opcode into a constructor of the datatype describing instructions:

```
data Instr
    = Sys Addr                      -- SYS nnn
    | Jump Addr                     -- JP nnn
    | JumpPlusV0 Addr               -- JP nnn, V0
    | Call Addr                     -- CALL nnn
    | Ret                           -- RET
    | SkipEqImmIs Bool Reg Byte     -- S[N]E Vx, nn
    | SkipEqRegIs Bool Reg Reg      -- S[N]E Vx, Vy
    | LoadImm Reg Byte              -- LD Vx, nn
    | AddImm Reg Byte               -- ADD Vx, nn
    | StoreRegs Reg                 -- LD [I], Vx
    | LoadRegs Reg                  -- LD Vx, [I]
    | LoadPtr Addr                  -- LD I, nnn
    | AddPtr Reg                    -- ADD I, Vx
    | Arith Fun Reg Reg             -- LD/ADD/SUB/SUBN/
                                    -- AND/OR/XOR/SHL/SHR
    | Randomize Reg Byte            -- RND Vx, nn
    | StoreBCD Reg                  -- BCD Vx
    | ClearScreen                   -- CLS
    | DrawSprite Reg Reg Nybble     -- DRW Vx, Vy, nn
    | LoadHex Reg                   -- HEX Vx
    | SkipKeyIs Bool Reg            -- SK[N]P Vx
    | WaitKey Reg                   -- LD Vx, K
    | LoadTimer Reg                 -- LD DT, Vx
    | GetTimer Reg                  -- LD Vx, DT
    | LoadSound Reg                 -- LD ST, Vx
    deriving (Show)
```

Note that we mapped all binary register operations to `Arith`, including `MOV Vx, Vy`. After all, it is no different than an arithmetic operation with the identity function.

```
data Fun
    = Mov           -- MOV
    | Or            -- OR
    | And           -- AND
    | XOr           -- XOR
    | Add           -- ADD
    | Subtract      -- SUB
    | ShiftRight    -- SHR
    | SubtractNeg   -- SUBN
    | ShiftLeft     -- SHL
    deriving (Show)
```

We write the instruction decoder as a pure function operating on two bytes. Of course, since we access memory one byte at a time, the CPU will need to do some housekeeping before it can use the decoder. We postpone that problem to the part where we'll be implementing the CPU, and just concentrate on mapping machine code into the `Instr` datatype for now.

We convert the two bytes into four nybbles, but also take the second, third and fourth nybbles into a 12-bit value. This is going to be useful because the machine code for instructions with an immediate address argument (such as `LD I, nnn`) always encode the address operand in the lower three nybbles.

```
decodeInstr :: Byte -> Byte -> Instr
decodeInstr hi lo = case nybbles of
    -- See branches below
    _  -> errorX $ printf "Unknown opcode: %02x %02x" hi lo
  where
    (n1, n2) = bitCoerce hi :: (Nybble, Nybble)
    (n3, n4) = bitCoerce lo :: (Nybble, Nybble)
    nybbles = (n1, n2, n3, n4)
    addr = bitCoerce (n2, n3, n4)
    imm = lo
```

There's no way around it: the branches of the actual decoding is boring code that we have to get through.

```
    (0x0, 0x0, 0xe, 0x0) -> ClearScreen
    (0x0, 0x0, 0xe, 0xe) -> Ret
    (0x0,   _,   _,   _) -> Sys addr
    (0x1,   _,   _,   _) -> Jump addr
    (0x2,   _,   _,   _) -> Call addr
    (0x3,   x,   _,   _) -> SkipEqImmIs True x imm
    (0x4,   x,   _,   _) -> SkipEqImmIs False x imm
    (0x5,   x,   y, 0x0) -> SkipEqRegIs True x y
    (0x6,   x,   _,   _) -> LoadImm x imm
    (0x7,   x,   _,   _) -> AddImm x imm
    (0x8,   x,   y, fun) -> Arith (decodeFun fun) x y
    (0x9,   x,   y, 0x0) -> SkipEqRegIs False x y
    (0xa,   _,   _,   _) -> LoadPtr addr
    (0xb,   _,   _,   _) -> JumpPlusV0 addr
    (0xc,   x,   _,   _) -> Randomize x imm
    (0xd,   x,   y,   n) -> DrawSprite x y n
    (0xe,   x, 0x9, 0xe) -> SkipKeyIs True x
    (0xe,   x, 0xa, 0x1) -> SkipKeyIs False x
    (0xf,   x, 0x0, 0x7) -> GetTimer x
    (0xf,   x, 0x0, 0xa) -> WaitKey x
    (0xf,   x, 0x1, 0x5) -> LoadTimer x
    (0xf,   x, 0x1, 0x8) -> LoadSound x
    (0xf,   x, 0x1, 0xe) -> AddPtr x
    (0xf,   x, 0x2, 0x9) -> LoadHex x
    (0xf,   x, 0x3, 0x3) -> StoreBCD x
    (0xf,   x, 0x5, 0x5) -> StoreRegs x
    (0xf,   x, 0x6, 0x5) -> LoadRegs x
```

The functions available for arithmetic operations are as follows:

```
decodeFun :: Nybble -> Fun
decodeFun 0x0 = Mov
decodeFun 0x1 = Or
decodeFun 0x2 = And
decodeFun 0x3 = XOr
decodeFun 0x4 = Add
decodeFun 0x5 = Subtract
decodeFun 0x6 = ShiftRight
decodeFun 0x7 = SubtractNeg
decodeFun 0xe = ShiftLeft
decodeFun n = errorX $ printf "Unknown arithmetic function: %x" n
```

13.4 Video

We design the video system as a separate component, in the first iteration with its own memory for the video buffer. Its single input is going to be the signal containing video buffer writes, coming from the CPU. The main output is the VGA signal ready to be connected to the outside world. We also provide an auxiliary output of a 60 Hz trigger, simply because the video already runs at 60 frames per second, so we can get that for free.

We want to display 64×32 pixels in a 640×480 VGA mode; we can do that conversion by scaling and centering. If we wanted a full screen image, we could scale horizontally by 10 and vertically by 15, but that would lead to very narrow pixels. Instead, we will scale by 10 in both directions using vertical centering, achieving a letterbox effect.

We have to store $64 \times 32 = 2048$ bits for the pixels, but we need not store each pixel in its own memory cell. If we do, then a single `CLS` instruction will necessarily take 2048 cycles, since it needs to set each bit to 0 separately. Similarly, drawing an $8 \times n$ sprite involves $8 \times n$ roundtrips to the video buffer: both reads and writes are needed to calculate the XOR.

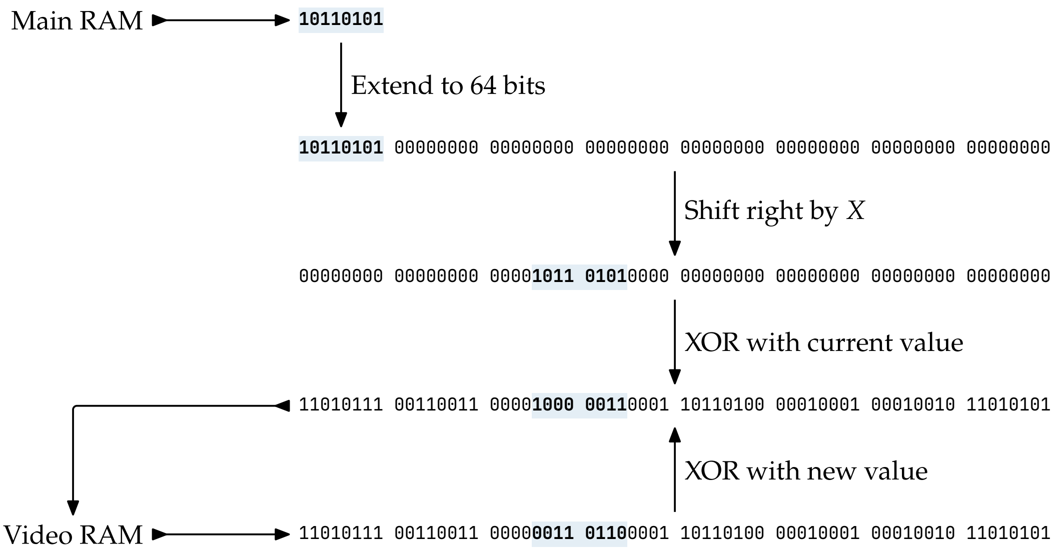
At the other end of the spectrum, we could have a single 2048-bit cell. While this would certainly speed up erasing (a single cycle!), it would be horribly unwieldy to draw onto. By way of an example, suppose we use the `DRW` instruction to draw an 8×9 sprite to $(59, 27)$. After fetching the first byte of sprite data, containing the first row, we would need to modify bits 1787, 1788, 1789, 1790 and 1791 (but not more, since the rest of the sprite is out of bounds horizontally). Then in the next cycle, the next sprite data byte is used to modify bits 1851 to 1855, and so on, until we get to the fifth row of bits 2043 to 2047, and then ignore the remaining four rows since they are out of bounds vertically.

Instead of either extreme, we will go with a memory layout that helps, rather than hinders, the implementation of the `DRW` instruction. Each row is 64 pixels – we can store that as a 64-bit-wide RAM containing 32 cells. The addressing scheme of the video buffer becomes identical to the logical vertical addressing of pixels, which means we can use Y coordinates as-is as addresses. Since sprite data is stored in bytes row by row, it is natural to also update the video buffer row by row; and we can simply stop as soon as an internal counter hits the maximum address to detect vertical overhang.

To draw the sprite to the correct location, we can extend the 8-bit sprite row data into a 64-bit value, and shift it to the right by X. By using shift instead of rotate, we take care of discarding horizontally overhanging pixels. The resulting 64-bit value is then ready to be XOR-combined with the 64 bits currently in the given row's cell.

This layout is also easy to use from the video signal generator, because we can connect the Y coordinate directly to the video buffer's address line. As for horizontal addressing, we can take the $(63 - X)^{th}$ bit of the 64-bit read value (since bits are indexed right to left).

Even better, since the electron beam is scanning the screen line by line, at the start of each line we can read the 64-bit value into a 64-bit register, and keep shifting out its most significant bit. This means we only need to fetch from the video buffer once per line, which will become beneficial when we revisit our design to share the same video RAM between the CPU and the video signal generator.



Drawing a sprite row into video RAM

13.4.1 Interface

Taking all these ideas into account, we are now ready to start writing the video generator component of our CHIP-8 computer. We use a fixed 25 MHz clock for this component since it needs to match the chosen VGA mode's pixel clock frequency. The other components will be, as usual, polymorphic in the clock; by wiring them together in `topEntity`, they will all get instantiated to this 25 MHz clock.

```
-- | 25 MHz clock, needed for the VGA mode we use.
createDomain vSystem{vName="Dom25", vPeriod = hzToPeriod 25_175_000}

type VidX = Unsigned 6
type VidY = Unsigned 5
type VidRow = Word64

video
    :: (HiddenClockResetEnable Dom25)
    => Signal Dom25 (Maybe (VidY, VidRow))
    -> (Signal Dom25 Bool, VGAOut Dom25 8 8 8)
```

The only input to the video subsystem is the signal carrying video buffer writes. Since we have decided to duplicate the video memory for our first version, both this component and the main logic board will maintain their own copies of the video buffer in a small 32 × 64 bit RAM, taking in write requests forked off from the CPU's output. The two outputs of the video circuitry are the 60 Hz tick (to be connected to the CPU) and the VGA output to the outside world.

13.4.2 Implementation

We start with laying the foundations: generating the VGA sync signals, detecting the start of the vertical blanking area, and converting the physical 640 × 480 video coordinates to 64 × 32 by scaling and centering. The only concern of the rest of the circuit will be calculating the `rgb` value for the currently drawn pixel.

```
video write = (frameEnd, vgaOut vgaSync rgb)
  where
    VGADriver{..} = vgaDriver vga640x480at60
    frameEnd = isFalling False (isJust <$> vgaY)

    vgaX' = fst . scale @64 (SNat @10) $ vgaX
    vgaY' = fst . scale @32 (SNat @10) . center $ vgaY
```

Since the CHIP-8 is monochrome, it would make no sense to store a full 24-bit RGB value for each pixel. Instead, we will use a **palette**: a mapping from a small number of color selectors to the definition of those colors. In our case, the palette consists of only two colors, so we will use a `Bit` as the color selector, and choose a nice amber glow for set pixels, to evoke the look of old monochrome displays. Also, since the scaled-up vertical resolution is smaller than the physical one, we have pixels which are "not visible" virtually, but are actually on the screen. We will compute the current `pixel` as a `Maybe Bit` signal, and draw a border in a third color for the `Nothing` case. We could simply draw the border using the unset color (i.e. black), but one advantage of having an explicit border color is as a kind of debugging device, since it allows us to see where both `vgaX'` and `vgaY'` are `Just` values.

```
    rgb = maybe border palette <$> pixel

    border = (0x30, 0x30, 0x50)

    palette 0 = (0x00, 0x00, 0x00)
    palette 1 = (0xff, 0xcc, 0x33)
```

Now we are getting to the meat of it: computing the `Maybe Bit` value of the currently drawn logical `pixel`. We do this by maintaining the remainder of the current `row` as a 64-bit value, shifting it to the left every time the virtual X coordinate changes. From this, we can easily compute the current `pixel` (if visible) by taking the rightmost (i.e. the most significant) bit:

```
    pixel = enable visible $ msb <$> row
    visible = isJust <$> vgaX' .&&. isJust <$> vgaY'
```

This takes care of drawing a single row. Drawing the current row requires `loading` into `row` from memory at each `lineStart`, using the Y coordinate as the `address`:

```
    address = maybe 0 bitCoerce <$> vgaY'
    load = blockRam1 NoClearOnReset (SNat @32) 0 address write

    lineStart = isRising False $ isJust <$> vgaX'
    newX = changed Nothing vgaX'

    row = register 0 $
        mux lineStart load $
        mux newX ((`shiftL` 1) <$> row) $
        row
```

This is not the final version of the video pattern generator: there is a subtle bug lurking in there, and there's also room to improve in preparation of sharing the video buffer with the CPU. But before we move on, let's write a small `topEntity` for testing purposes, that writes a checkerboard pattern into video RAM, drawing one row per second.

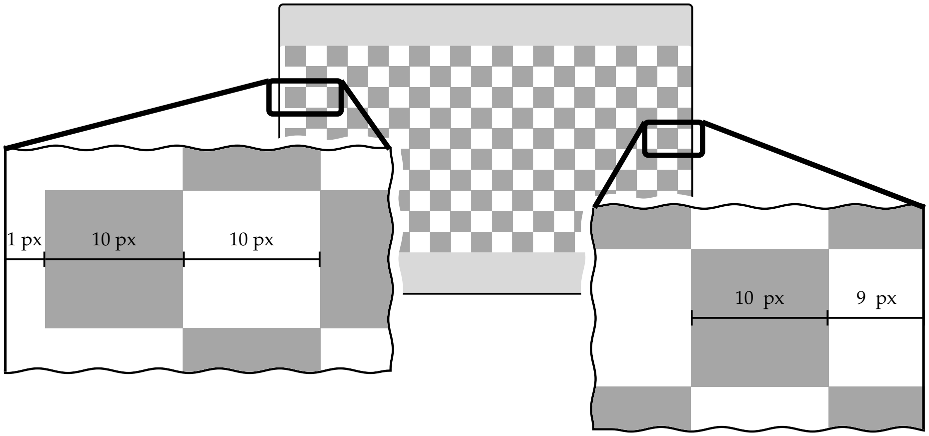
```
topEntity
    :: "CLK_25MHZ" ::: Clock Dom25
    -> "RESET"     ::: Reset Dom25
    -> "VGA"       ::: VGAOut Dom25 8 8 8
topEntity = withEnableGen board
  where
    board = vga
      where
        (frameEnd, vga) = video write
        timer = riseEveryWhen (SNat 60) frameEnd
        ptr = regEn 0 timer $ ptr + 1
        val = regEn pattern timer $ complement <$> val
        write = Just <$> ((,) <$> ptr <*> val)
        pattern = 0xaaaa_aaaa_aaaa_aaaa
```

The 64-bit constant is chosen because the binary representation of `0xaa` is `0b10101010`, so by putting 8 copies of it next to each other, we get a full-screen-width checkerboard pattern. Its complement is, of course, `0b01010101`.

"One row per second" is achieved by counting to 60 frames, using a version of `riseEvery` that takes an explicit trigger signal instead of counting every cycle:

```
riseEveryWhen
    :: forall n dom. (HiddenClockResetEnable dom, KnownNat n)
    => SNat n
    -> Signal dom Bool
    -> Signal dom Bool
riseEveryWhen n trigger = isRising False $ cnt .==. pure maxBound
  where
    cnt = regEn (0 :: Index n) trigger (nextIdx <$> cnt)
```

If we try out this version on real hardware, it is entirely possible that we'll come to the conclusion that everything is all right: after all, the pattern does appear row by row, second by second. However, if we look real closely at the leftmost edge of the screen, we will notice that the first column of physical pixels in the visible area is black, so the first virtual pixel is shifted by one physical pixel. Conversely, the last physical column of the last virtual pixel is cut off: it has to be, since we are trying to draw 10 × 64 pixels into a 640 pixel area but skipping the first column.



(1) Left-side edge
(first pixels of a line)

(2) Right-side edge
(last pixels of a line)

What is going on here?

13.4.3 Timing problems

When it comes to drawing on a screen, time and space is unified like an Einsteinian *gedankenexperiment*. The virtual pixels are off by one column because the correct pixel data is computed one pixel, i.e. one cycle, too late.

We can solve the immediate problem by thinking about the relationship between `row` and `pixel`. Let's look at their definitions side by side:

```
row = register 0 $
    mux lineStart load $
    mux newX ((`shiftL` 1) <$> row) $
    row

pixel = enable visible $ msb <$> row
```

At the start of each line, the *new* value of `row` is the line we `load` from RAM. However, `pixel` is computed from the *old* value, since the `Signal` value of a `register` is defined to be its value as it is and the beginning of the cycle.

We have already seen a solution to a similar problem when writing video coordinate transformers, and indeed we can do the same here: instead of referring to `row` in the definition of `pixel`, we split `row` into a separate signal `row'` holding the new value, and use that both to update `row` and to define `pixel`:

```
row = register 0 row'

row' =
    mux lineStart load $
    mux newX ((`shiftL` 1) <$> row) $
    row

pixel = enable visible $ msb <$> row'
```

And so this version avoids the empty first column: a fact we can either convince ourselves by thinking about it, or observe by testing.

Let's change our implementation a bit, with an eye towards decreasing memory access. Since we plan to eventually share a single video RAM between the video generator and the CPU, one local improvement we can make is to change the `address` to a `Maybe VidY`. The idea here is that with this small change, we can prepare for later versions to service CPU-originating read requests only when the video generator's request `address` is `Nothing`, since we can't ask the CRT's electron beam to take five while the CPU does its thing.

That should be an easy change: we can simply force `address` to be always `Nothing`, except one cycle at the start of each line:

```
address = bitCoerce <$> mux lineStart vgaY' (pure Nothing)
load = blockRam1 NoClearOnReset (SNat @32) 0 (address .<| 0) write
```

This pattern of adding an extra constraint on a `Maybe`-valued signal will come up in other contexts as well, so it makes sense to add a combinator for it:

```
guardA :: (Applicative f, Alternative m) => f Bool -> f (m a) -> f (m a)
guardA en x = mux en x (pure empty)
```

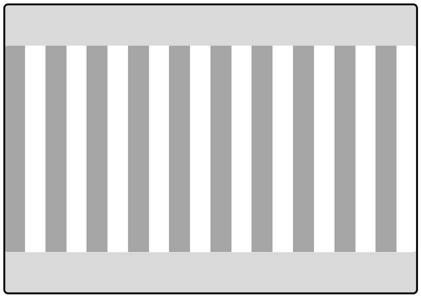
The new version of `address` is thus:

```
    address = bitCoerce <$> guardA lineStart vgaY'
```

This also allows us to change `row'` slightly to avoid duplicating the decision logic to replace its value: we want to `load` from RAM if and only if `address` is set.

```
    row' =
        mux (isJust <$> address) load $
        mux newX ((`shiftL` 1) <$> row) $
        row
```

We synthesize this new version, load it on the development board, hook it up to a screen, and instead of the checkerboard pattern, we see this:

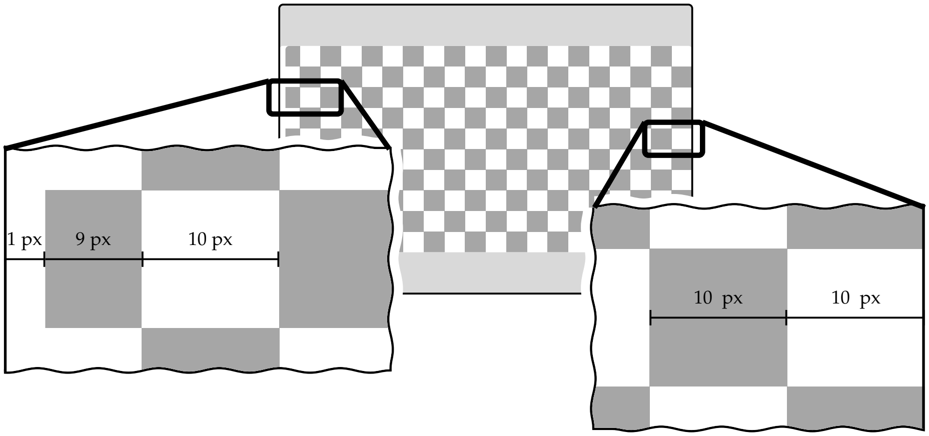


The root of the problem is the read from the `blockRam1` component. Since it is a synchronous RAM component, when the `address` changes, it takes until the next cycle to `load` the new value. By the time we have this new value ready, `address` is already `Nothing`. Conversely, in the cycle when `address` is set, `load` still holds the result of reading from address 0 (since that is the address we fall back to). The end result of this is that whenever `isJust <$> address` occurs, `load` will always contain the value of cell #0, i.e. the first row. That is why instead of the checkerboard pattern, we see just vertical bars: it is the first line, repeated for each virtual Y coordinate.

So if `load` is ready one cycle later than `address`, then we can fix this by replacing `row` with `load` *one cycle after* the `address` has been set:

```
    row' =
        mux (register False $ isJust <$> address) load $
        mux newX ((`shiftL` 1) <$> row) $
        row
```

Let's try this version out:



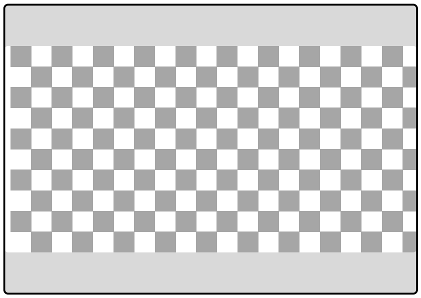
(1) Left-side edge
(first pixels of a line)

(2) Right-side edge
(last pixels of a line)

Oh no, not this again! Are we going in circles? Not exactly. Unlike the previous version with the off-by-one column problem, this new one is not translated by one column – it is missing it. We can see this by measuring the sizes of the virtual pixels: the first one is 9 pixels wide, whereas all the others are 10. And this makes sense: when we changed the definition of `row'`, we have delayed the *start* of each line by one cycle, but the subsequent `newX` trigger will happen just the same as before. Let's delay `newX` as well, if nothing else, then just to get back to a situation we were able to solve once before:

```
    row' =
        mux (register False $ isJust <$> address) load $
        mux (register False newX) ((`shiftL` 1) <$> row) $
        row
```

Just as expected, this puts us firmly back on square one, with the whole image being off by one physical pixel:



But what good is getting back to a previously encountered problem, if our previous solution isn't applicable? And it isn't, because we can't carve out a version of `row'` that wouldn't be late: due to the `blockRam1` delay, the result is simply not ready on time.

One thing we could try to do, of course, is to compute `vgaY'` a bit in advance, and fire `lineStart` early. This is tricky because there is no output of the VGA driver that would tell us "a new line is just about ready to start". Maybe we could make our own "shadow" VGA driver that would count the cycles after sync signal events to figure out how far we are from the next non-blanking portion of the screen?

Instead, we are going to re-frame the problem: *I'm not late, everyone else is too early!* Like a character in a Chaplin movie that sets the factory clock late to mislead the boss, we will time-shift the VGA sync signals, so that the monitor's electron beam will be drawing the pixel that was described in the *previous* cycle's `vgaX` and `vgaY` values.

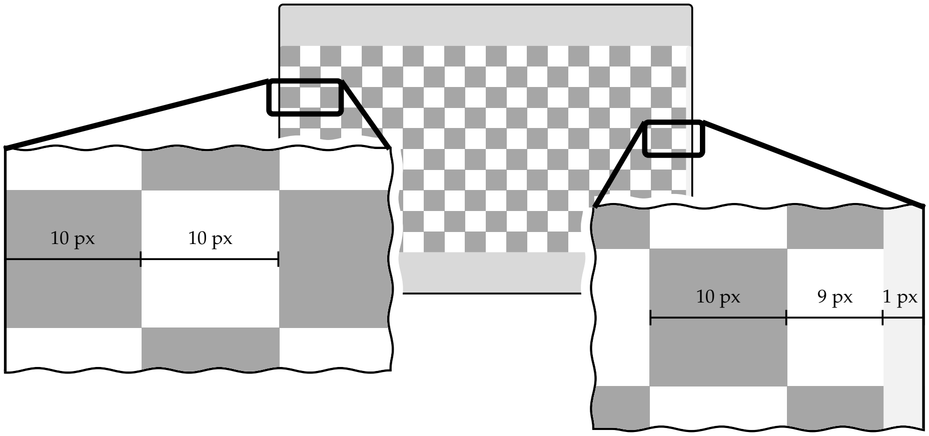
```
video write = (frameEnd, vgaOut (delayVGASync vgaSync) rgb)
```

We shift the sync signals without a worry for their extra value: not that we'd know if it should be `high` or `low`[8], and not that we care either: whatever happens on the first pixel clock cycle, the screen will synchronize to the signals by the end of the first frame in the latest.

```
delayVGASync
    :: (HiddenClockResetEnable dom) => VGASync dom -> VGASync dom
delayVGASync VGASync{..} = VGASync
    { vgaHSync = register undefined vgaHSync
    , vgaVSync = register undefined vgaVSync
    , vgaDE = register False vgaDE
    }
```

[8] If we wanted to extend with the correct values real bad, we could think about tracking the sync polarities in type indices for `VGASync`, but the extra complication is not worth it.

Now with this latest version, the virtual pixels start at the right physical column, the first pixels are the right size... but disaster strikes in the last column, where instead of the rightmost edge of the virtual pixels, we can see the border:



(1) Left-side edge
(first pixels of a line)

(2) Right-side edge
(last pixels of a line)

This game of whack-a-mole is getting out of hand!

13.4.4 Tracking signal delay

It is not *impossible*, but also not *trivial* to figure out what is wrong: the `visible` signal is computed from undelayed inputs, so it now becomes `False` one cycle too early. Accordingly, we fix it by delaying it in the definition of `pixel` so that it is in sync with `row'`:

```
    pixel = enable (register False visible) $ msb <$> row'
```

This finally gives us a video signal generator with the correct output. Let's look at the code in its entirety with all the accumulated changes applied:

```
video
    :: (HiddenClockResetEnable Dom25)
    => Signal Dom25 (Maybe (VidY, VidRow))
    -> (Signal Dom25 Bool, VGAOut Dom25 8 8 8)
video write = (frameEnd, vgaOut (delayVGASync vgaSync) rgb)
  where
    VGADriver{..} = vgaDriver vga640x480at60
    frameEnd = isFalling False (isJust <$> vgaY)
    vgaX' = fst . scale @64 (SNat @10) vgaX
    vgaY' = fst . scale @32 (SNat @10) . center $ vgaY

    rgb = maybe border palette <$> pixel

    border = (0x30, 0x30, 0x50)

    palette 0 = (0x00, 0x00, 0x00)
    palette 1 = (0xff, 0xcc, 0x33)

    pixel = enable (register False visible) $ msb <$> row'
    visible = isJust <$> vgaX' .&&. isJust <$> vgaY'

    address = bitCoerce <$> guardA lineStart vgaY'
    load = blockRam1 NoClearOnReset (SNat @32) 0 (address .<| 0) write

    lineStart = isRising False $ (isJust <$> vgaX')
    newX = changed Nothing vgaX'

    row = register 0 row'
    row' =
        mux (register False $ isJust <$> address) load $
        mux (register False newX) ((`shiftL` 1) <$> row) $
        row
```

We can now get rid of `row'` and use `row` directly, by adding one more cycle of delay to everything else:

```
video
    :: (HiddenClockResetEnable Dom25)
    => Signal Dom25 (Maybe (VidY, VidRow))
    -> (Signal Dom25 Bool, VGAOut Dom25 8 8 8)
video write = (frameEnd, vgaOut (delayVGASync . delayVGASync $ vgaSync)
    rgb)
  where
    pixel = enable (register False . register False $ visible) $ msb <$>
    row

    row = register 0 $
        mux (register False $ isJust <$> address) load $
        mux (register False newX) ((`shiftL` 1) <$> row) $
        row

    -- Rest unchanged
```

Looking at the final code, there are two problems with our approach so far:

- There is no way we could have written the final code on our first try. We had to keep synthesizing onto hardware and looking at the resulting image, then thinking a lot to come up with an explanation.
- It is very fragile: if we change anything, we have to do the above process again.

The alternative is to let the computer do what it does best: compute. In this case, computing the signal delays and letting us know if things are not in sync.

For this use case, Clash's standard library provides the `DSignal` type for *delayed signals*, or, more accurately, *signals with tracked delay*. The type `DSignal dom d a` is the same as `Signal dom a`, but marked to be delayed by `d` clock cycles, compared to the baseline of `DSignal dom 0 a`. It is still an applicative functor, so combining signals with the same delay is just as straightforward as non-delayed signals. `DSignal` only *tracks* delay, it doesn't model the source of delay or what delay even means; that is up to us as users.

For our current problem, we will use `DSignal dom d a` to mean that something is computed based on the pixel coordinates `d` cycles ago. We will have two sources of delay: the block RAM read, and the shift register containing the current row. For the first one, we can create a function that transforms non-delay-tracked RAM elements into delay-tracked ones:

```
delayedRam
    :: (HiddenClockResetEnable dom)
    => (forall dom'. (HiddenClockResetEnable dom')
        => Signal dom' addr
        -> Signal dom' wr
        -> Signal dom' a)
    -> DSignal dom d addr
    -> DSignal dom d wr
    -> DSignal dom (d + 1) a
delayedRam syncRam addr write =
    unsafeFromSignal $ syncRam (toSignal addr) (toSignal write)
```

Here, we use the Clash primitives `toSignal` and `unsafeFromSignal` to convert from/to `DSignal`. `toSignal` simply forgets the delay information, turning any `DSignal dom d a` into a `Signal dom a`. Its dual is `fromSignal`, which creates a `DSignal dom 0 a`. Here, we use `unsafeFromSignal` because we want to state, by fiat, that the resulting signal has a non-zero delay. It is unsafe in the sense that it is up to us, the programmer, to ensure that the delay in the type corresponds to the signal delay of the underlying circuit. In this case, since we know that the various synchronous RAM primitives like `blockRam1` produce a result one cycle after changing the address line, we use `unsafeFromSignal` and set the correct delay in the type annotation to one more than the input's delay. We use a rank-2 type for the RAM primitive to make sure we aren't passed a circuit which has other signal dependencies that could influence the delay.

For registers, we can make a function that takes care of tying the knot, by computing a d-delayed value from the current value (and possibly other d-delayed signals), and using it as the new value that will be available from the next cycle, delayed by $d + 1$ cycles in total:

```
delayedRegister
    :: (NFDataX a, HiddenClockResetEnable dom)
    => a
    -> (DSignal dom d a -> DSignal dom d a)
    -> DSignal dom (d + 1) a
delayedRegister initial feedback = r
  where
    r = unsafeFromSignal $ register initial $ toSignal new
    old = antiDelay (SNat @1) r
    new = feedback old
```

Here, `antiDelay` is another unsafe primitive operation on `DSignal`: this one changes the delay tag from `DSignal dom (d + k)` to `DSignal dom d`. Just like `toSignal`, `fromSignal` or `unsafeFromSignal`, it has no operational effect; the only change is in the type-level delay index. Here, we use it because our register's value `r` is delayed by $d + 1$ according to the type signature of `delayedRegister`, but the whole point of tying the knot ourselves instead of outside is to apply the `feedback` function on the start-of-cycle value of `r`.

Another handy utility function is one that lifts a non-delaying signal function to the world of `DSignal`; we will need this when using the signal functions `isRising` and `changed` with `DSignal`-typed arguments.

```
liftD
    :: (HiddenClockResetEnable dom)
    => (forall dom'. (HiddenClockResetEnable dom') =>
          Signal dom' a -> Signal dom' b)
    -> DSignal dom d a -> DSignal dom d b
liftD f = unsafeFromSignal . f . toSignal
```

Armed with these definitions, the plan is to rewrite the video circuit to use `DSignals`. To the outside world, it should still look like a circuit taking a `Signal` input and producing VGA output (note that the type signature below is unchanged); but internally, it will use the type-level delay information to offset the VGA sync signals by just the right number of cycles so that they match up with the RGB signals. So the type signature of `video` will not change, but instead of using `delayVGASync` some arbitrary number of times, we make a version that is type-directed by the delay tag:

```
delayVGA
    :: (KnownNat d, KnownNat r, KnownNat g, KnownNat b)
    => (HiddenClockResetEnable dom)
    => VGASync dom
    -> DSignal dom d (Unsigned r, Unsigned g, Unsigned b)
    -> VGAOut dom r g b

video
    :: (HiddenClockResetEnable Dom25)
    => Signal Dom25 (Maybe (VidY, VidRow))
    -> (Signal Dom25 Bool, VGAOut Dom25 8 8 8)
video write = (frameEnd, delayVGA vgaSync rgb)
  where
    -- See other changes below
```

The key to `delayVGA` is that it takes a delay-tracked RGB signal, and produces a full `VGAOut` which has the sync and the RGB signals matched, and then packaged up. If we didn't do the packaging, and instead wrote something like this:

```
delayVGA
    :: (HiddenClockResetEnable dom)
    => VGASync dom
    -> DSignal dom d rgb
    -> VGASync dom
```

This API would run the risk of passing a different RGB signal to `delayVGA` as to the eventual `vgaOut` call, i.e. something like the following where `rgb1` and `rgb2` are unrelated and could have different delays:

```
vga = vgaOut (delayVGA vgaSync rgb1) (toSignal rgb2)
```

The implementation of `delayVGA` is quite straightforward: we transform the sync signals by delaying them the appropriate amount, and package them up with `toSignal rgb` which removes the delay tag from `rgb`:

```
delayVGA VGASync{..} rgb = vgaOut vgaSync' (toSignal rgb)
  where
    vgaSync' = VGASync
        { vgaHSync = matchDelay rgb undefined vgaHSync
        , vgaVSync = matchDelay rgb undefined vgaVSync
        , vgaDE = matchDelay rgb False vgaDE
        }
```

The workhorse function of `delayVGA` is `matchDelay` which takes a reference `DSignal` whose contents is irrelevant, and prepends the right amount of initial values `x0`. We do this with the Clash primitive `delayI`, which translates a `DSignal dom d0` to a `DSignal dom (d0 + k)`: operationally, `delayI x0` is equivalent to `register x0` iterated k times. In our case, we will start with the result of `fromSignal`, so `d` = 0; the type of our reference delayed signal forces $k = d$ via `(ref *>)`.

```
delayI :: _ => a -> DSignal dom d a -> DSignal dom (d + k) a

matchDelay
    :: (KnownNat d, NFDataX a, HiddenClockResetEnable dom)
    => DSignal dom d any
    -> a
    -> Signal dom a
    -> Signal dom a
matchDelay ref x0 = toSignal . (ref *>) . delayI x0 . fromSignal
```

Let's evaluate what all this extra machinery gets us, by rewriting our video circuit to use `DSignal`, but without applying any of the hard-earned delay consistency bug fixes.

```
video write = (frameEnd, delayVGA vgaSync rgb)
  where
    VGADriver{..} = vgaDriver vga640x480at60
    frameEnd = isFalling False (isJust <$> vgaY)

    vgaX' = fromSignal $ fst . scale @64 (SNat @10) $ vgaX
    vgaY' = fromSignal $ fst . scale @32 (SNat @10) . center $ vgaY

    rgb = maybe border palette <$> pixel

    border = (0x30, 0x30, 0x50)

    palette 0 = (0x00, 0x00, 0x00)
    palette 1 = (0xff, 0xcc, 0x33)

    pixel = enable visible $ msb <$> row
    visible = isJust <$> vgaX' .&&. isJust <$> vgaY'

    address = bitCoerce <$> guardA lineStart vgaY'
    load = delayedRam (blockRam1 NoClearOnReset (SNat @32) 0)
        (address .<| 0) (fromSignal write)
    lineStart = liftD (isRising False) $ (isJust <$> vgaX')
    newX = liftD (changed Nothing) vgaX'

    row = delayedRegister 0 $ \row ->
        mux (isJust <$> address) load $
        mux newX ((`shiftL` 1) <$> row) $
        row
```

Compared to the non-`DSignal` version, the changes are:

- `vgaX'` and `vgaY'` are converted using `fromSignal`. This is what kickstarts the whole refactoring: since these signals are now delay-tagged, everything else up to `pixel` is now delay-tagged as well.
- We use `liftD` to lift into `DSignal` the stateful signal functions `isRising` and `changed`. We use `liftD` for these because both of these return results that are valid in the same cycle as their inputs (for example, `isRising` is `True` in the same cycle as when its argument signal first becomes `True`).
- The video buffer access and the `row` register are the two sources of signal delay we are modeling, so these use `unsafeFromSignal` under the hood.

If we try to compile this version, we get a type error:

```
CHIP8/Video.hs:56:34: error:
    • Couldn't match type ‘’1 with ‘’0
      Expected type: DSignal Dom25 0 VidRow
        Actual type: DSignal Dom25 (0 + 1) VidRow
    • In the second argument of ‘’mux, namely ‘’load
      In the expression: mux (isJust <$> address) load
```

And boom goes the dynamite: the type checker has identified the bug that we try to load the new row value one cycle too late compared to the condition of `isJust <$> address`. In the non-`DSignal` version, this compiled, and we had a nasty timing bug on our hands, without any indication of where to find it and then how to fix it. But here, the type checker helpfully points out that we need to shift `isJust <$> address` by one cycle so that its value will be consistent with when `load` is ready.

This allows us to fix this problem by changing the definition of `row`:

```
    row = delayedRegister 0 $ \row ->
        mux (delayI False $ isJust <$> address) load $
        mux newX ((`shiftL` 1) <$> row) $
        row
```

Further type errors gradually guide us first towards adding a `delayI` call to the other condition in `row`'s definition (since `newX` is not delayed but `row` is, since it is loaded from synchronous RAM), and then to the discrepancy between `visible` (with zero delay) and `msb <$> row` (with delay of two cycles). Fixing these, we get a version that typechecks, and works exactly as we'd like. In the following listing of the final code, * marks the changes prompted by the type checker:

```
video write = (frameEnd, delayVGA vgaSync rgb)
  where
    VGADriver{..} = vgaDriver vga640x480at60
    frameEnd = isFalling False (isJust <$> vgaY)

    vgaX' = fromSignal $ fst . scale @64 (SNat @10) $ vgaX
    vgaY' = fromSignal $ fst . scale @32 (SNat @10) . center $ vgaY

    rgb = maybe border palette <$> pixel

    border = (0x30, 0x30, 0x50)

    palette 0 = (0x00, 0x00, 0x00)
    palette 1 = (0xff, 0xcc, 0x33)

    pixel = enable (delayI False visible) $ msb <$> row  -- *
    visible = isJust <$> vgaX' .&&. isJust <$> vgaY'

    address = bitCoerce <$> guardA lineStart vgaY'
    load = delayedRam (blockRam1 NoClearOnReset (SNat @32) 0)
        (address .<| 0) (fromSignal write)

    lineStart = liftD (isRising False) $ (isJust <$> vgaX')
    newX = liftD (changed Nothing) vgaX'

    row = delayedRegister 0 $ \row ->
        mux (delayI False $ isJust <$> address) load $   -- *
        mux (delayI False newX) ((`shiftL` 1) <$> row) $ -- *
        row
```

13.5 CPU

We will use the same monadic approach to writing our CHIP-8 processor that we developed for the Brainfuck CPU. We start by defining the interface according to the system overview section:

```
type KeypadState = Vec 16 Bool

declareBareB [d|
  data CPUIn = CPUIn
      { memRead :: Byte
      , vidRead :: VidRow
      , keyState :: KeypadState
      , tick :: Bool
      } |]

declareBareB [d|
  data CPUOut = CPUOut
      { _memAddr :: Addr
      , _memWrite :: Maybe Byte
      , _vidAddr :: VidY
      , _vidWrite :: Maybe VidRow
      } |]
makeLenses ''CPUOut
```

We don't yet have a clear picture of what the CPU registers are going to be, but even a cursory reading of the instruction set suggests that at the very least we are going to need a 12-bit program counter, 16 general purpose registers, one 12-bit pointer register, and a stack of addresses. The original specification says the `CALL` / `RET` stack to have enough space to store 12 entries, but because many later implementations had more stack space, we will double that to ensure compatibility with more modern programs.

```
data CPUState = CPUState
    { _pc, _ptr :: !Addr
    , _registers :: !(Vec 16 Byte)
    , _stack :: !(Stack 24 Addr)
    }
    deriving (Show, Generic, NFDataX)

initState :: CPUState
initState = CPUState
    { _pc = 0x200
    , _ptr = 0x000
    , _registers = repeat 0
    , _stack = Stack (repeat 0) 0
    }
```

We are using strict fields in `CPUState` to improve simulation performance and avoid inconsistent speed when taking one CPU step; they make no difference when synthesizing.

Because the CHIP-8 is a von Neumann architecture machine, it is natural to assign by default the program counter to the memory address:

```
defaultOut :: CPUState -> Pure CPUOut
defaultOut CPUState{..} = CPUOut
  { _memAddr = _pc
  , _memWrite = Nothing
  , _vidAddr = 0
  , _vidWrite = Nothing
  }
```

One big difference compared to the Brainfuck CPU is that the CHIP-8 has instructions that are larger than the smallest addressable memory: addresses are per byte, but instructions take up 16 bits. This means we will have to fetch instructions in two cycles: first the first (high) byte, then the second (low) byte. So let's add a `Phase` ADT to keep track of what we are supposed to be doing in a given cycle. Just like `CPUState`, the below is not the final definition; we will add more phases as needed, while implementing the various instructions. We'll also add an `Init` phase because we remember from the Brainfuck CPU that we need to set the address output in a full cycle before fetching the first valid byte.

```
data Phase
    = Init
    | Fetch
    | Exec Byte
    deriving (Show, Generic, NFDataX)

data CPUState = CPUState
    { _phase :: !Phase
    ...
    }

initState = CPUState
    { phase = Init
    ...
    }
```

We only store one `Byte` (the high one) in `Exec` because the low one will be just ready on the data bus. So now we know roughly the shape of implementing a single cycle of our CPU, going from `Init` to `Fetch`, from `Fetch` to `Exec`, and then from `Exec` back to `Fetch` for the next instruction:

```
type CPU = CPUM CPUState CPUOut

step :: Pure CPUIn -> CPU ()
step CPUIn{..} = do
    use phase >>= \case
        Init -> phase .= Fetch
        Fetch -> do
            pc += 1
            phase .= Exec memRead
        Exec hi -> do
            let lo = memRead
            pc += 1
            phase .= Fetch
            case decodeInstr hi lo of
                -- TODO: Implement instructions
                instr -> errorX $ show instr
        -- Other phases to be added as needed
  where
    -- Various utility definitions described below
```

13.5.1 Simple register-changing operations

Now that we have a structure, we can start filling it in for the implementation of the simpler instructions, by adding branches to the pattern matching on the decoded instruction. Perhaps the simplest of them all is `Jump`, which overwrites the program counter as its only effect:

```
Jump addr -> do
    pc .= addr
```

And that's all we need to do, because by default, we go to the `Fetch` phase next (since we set `phase` to `Fetch` before branching on the decoded instruction). Since we use the value of the `pc` register as the default memory address output, by changing `pc` here we ensure that the `Fetch` in the next cycle will get the opcode from the correct address.

Let's fill in everything that is as straightforward as `Jump`. For example, `LoadPtr` is exactly the same as a jump, the only difference is the target register being `ptr`, not `pc`:

```
LoadPtr addr -> do
    ptr .= addr
```

A lot of the instructions index into the 16-element general purpose register array. Let's add two helper functions for that, and then we can implement everything involving direct register access.

```
setReg :: Reg -> Byte -> CPU ()
setReg reg val = registers %= replace reg val

getReg :: Reg -> CPU Byte
getReg reg = uses registers (!! reg)
```

In the rest of this chapter, in code we will refer to the register names as `vx, vy :: Reg` and their current values as `x, y :: Byte`.

```
LoadImm vx imm -> do
    setReg vx imm
AddImm vx imm -> do
    x <- getReg vx
    setReg regX (x + imm)

JumpPlusV0 addr -> do
    offset <- getReg 0
    pc .= addr + fromIntegral offset
AddPtr vx -> do
    x <- getReg vx
    ptr += fromIntegral x
```

To implement the conditional branching instructions, we make the observation that since one instruction is 2 bytes, and we have already added 1 to `pc` before the branch, we can skip an instruction by simply adding 2 more to the program counter:

```
skip :: CPU ()
skip = pc += 2
```

We have picked the instruction representation so that both `SkipEqImmIs b` and `SkipEqRegIs b` are supposed to skip if the result of the given comparison is `b`:

```
SkipEqImmIs b vx imm -> do
    x <- getReg vx
    when ((x == imm) == b) skip
SkipEqRegIs b vx vy -> do
    x <- getReg vx
    y <- getReg vy
    when ((x == y) == b) skip
```

Two instructions Ret and Call require access to the internal stack, by respectively pushing and popping the current program counter. The code is simplified a lot by reusing the Stack implementation from our Brainfuck CPU:

```
Ret -> do
    (pc', stack') <- pop <$> use stack
    stack .= stack'
    pc .= pc'
Call addr -> do
    pc0 <- use pc
    stack %= push pc0
    pc .= addr
```

The remaining instructions can be classified into the following groups:

- Various 8-bit arithmetic functions via Arith that take arguments from registers and write the result back into a register.
- Instructions that deal directly with peripherals and special-purpose registers: WaitKey and SkipKeyIs for the keypad, Randomize for the built-in pseudo-random number generator, and GetTimer, LoadTimer and LoadSound for the timers.
- Graphic instructions: ClearScreen, DrawSprite, LoadHex.
- Instructions that deal with memory access: StoreRegs, LoadRegs and StoreBCD. The latter might seem like an arithmetic instruction, but recall that instead of writing its result to a register (where it wouldn't fit), it stores it directly in three consecutive bytes at the address stored in the ptr register.

Actually, there is one more instruction that we missed: Sys. However, as explained earlier, we are not going to support it:

```
Sys n -> do
    errorX "Unimplemented: Sys"
```

13.5.2 Arithmetic instructions

In our representation of instructions, we use a single Arith constructor for all of the bitwise logic instructions OR, AND, and XOR, the arithmetic instructions ADD, SUB, and SUBN, and the shifting operations SHR and SHL. This is because the effect of all of these instructions is the same: take the current value of the two specified registers, apply some binary function on that, and then write the result back into the first register; additionally, the non-bitwise instructions also update the value of the VF register. Because the effects are the same, it makes sense to handle them in one single branch.

This unification of all Boolean logic and arithmetic instructions is quite usual in hardware CPUs as well, because it allows using one set of components for all the non-function-specific housekeeping. In our Clash code, this will manifest itself as a single `step` branch for the `Arith` case, leaving the actual computation to a separate pure function:

```
Arith fun fx vy -> do
    x <- getReg vx
    y <- getReg vy
    let (flag, x') = alu fun x y
    setReg vx x'
    traverse_ setFlag flag
```

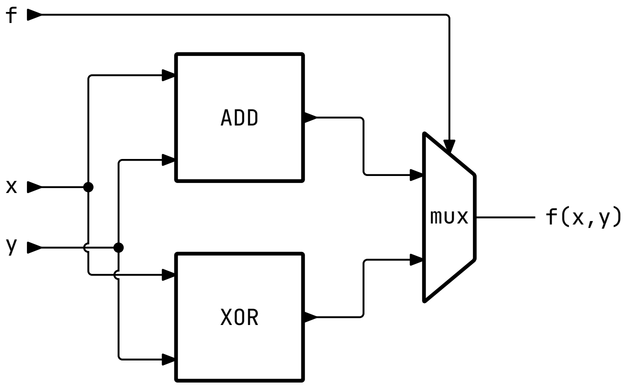
This uses a utility function to write a one-bit flag result to the register `VF`:

```
setFlag :: Bit -> CPU ()
setFlag = setReg 0xf . fromIntegral
```

The computation itself takes place in a pure function that returns the new value of the first register, and, if the given function changes `VF`, the flag:

```
alu :: Fun -> Byte -> Byte -> (Maybe Bit, Byte)
```

The manifestation of that pure function in a hardware CPU is traditionally called the *Arithmetic Logic Unit*, or ALU for short (hence the name `alu` in our code). It has inputs for the operands (usually two), which fan out to multiple combinational circuits each implementing a different function. Then, the results of each circuit are fed into a multiplexer which selects the one based on the selected function. The following diagram shows only two functions (addition and bitwise `XOR`) to avoid clutter:



For CHIP-8 specifically, we have nine arithmetic functions (the nine constructors of `Fun`), and they fall into two categories:

- For `OR`, `AND`, and `XOR`, a binary operator on bytes (i.e. a function `Byte -> Byte -> Byte`) is used to compute the result, and there is no flag to write to `VF`. As a somewhat degenerate case, `MOV` also fits this category, it just ignores its first operand.
- For `ADD`, `SUB` and `SUBN`, we have operators from 8 to 9 bits (i.e. functions `Unsigned 8 -> Unsigned 8 -> Unsigned 9`), with the topmost bit acting as the flag output. It is easy to see why this is the case for addition and subtraction: the carry from an 8-bit addition is the same as the topmost bit of an extending addition (and similarly for the borrow when subtracting). Note that the specification of `SUB` and `SUBN` calls for `VF` to be set to the *complement* of borrow; we implement this by XOR'ing the result with `0x100`.
- For `SHL` and `SHR`, if we implement them as a 9-bit *rotation* instead of *shift* (after 0-extending the 8-bit operand), the bit that is just getting shifted out from the lower 8 bits will rotate exactly into the topmost bit.

The implementation follows this categorization, lifting functions producing 8- or 9-bit sized results into the full type of `alu`:

```haskell
alu :: Fun -> Byte -> Byte -> (Maybe Bit, Byte)
alu fun = case fun of
    Mov         -> noFlag (\x y -> y)
    Or          -> noFlag (.|.)
    And         -> noFlag (.&.)
    XOr         -> noFlag xor
    Add         -> withFlag add
    Subtract    -> withFlag sub'
    SubtractNeg -> withFlag (flip sub')
    ShiftRight  -> withFlag (\_ y -> extend y `rotateR` 1)
    ShiftLeft   -> withFlag (\_ y -> extend y `rotateL` 1)
  where
    sub' x y = sub x y `xor` 0x100

    noFlag
        :: (Byte -> Byte -> Byte)
        -> (Byte -> Byte -> (Maybe Bit, Byte))
    noFlag f x y = (Nothing, f x y)
    withFlag
        :: (Unsigned 8 -> Unsigned 8 -> Unsigned 9)
        -> (Byte -> Byte -> (Maybe Bit, Byte))
    withFlag f x y = (Just c, z)
      where
        (c, z) = bitCoerce (f (bitCoerce x) (bitCoerce y))
```

Here we use `add` and `sub`, which are Clash-provided extending arithmetic functions, i.e. they operate on *n*-bit inputs and provide *n + 1*-bit results.

13.5.3 Keypad access

The two instructions for interfacing with the keypad are `SkipKeyIs` and `WaitKey`. Since the first one depends on the immediate state of the keys (which is available in the `keyState` input) and doesn't suspend execution, its implementation is going to be more straightforward.

The register operand of `SkipKeyIs b` contains the key whose pressed state we are interested in. Since registers contain 8 bits, but we only have 16 keys, we convert from `Byte` to `Key` by taking its lower 4 bits; fortunately, that is exactly how `fromIntegral` works for `Unsigned` numbers.

```
SkipKeyIs b vx -> do
    key <- fromIntegral <$> getReg vx
    let isPressed = keyState !! key
    when (isPressed == b) skip
```

Implementing `WaitKey` takes a bit more effort, since it blocks execution until an external event (a key release) happens. To suspend execution of further instructions, we add a new `Phase` which will only go back to `Fetch` when the key release is detected. To implement the actual detection, we need to remember the target register, and the previous keypad state so that we have something to compare against.

```
data Phase
    = WaitKeyRelease Reg KeypadState
    | ...
```

The branch for `WaitKey` now becomes trivial: we take the choice of `vx` and the current key state, and enter the `WaitKeyRelease` phase:

```
WaitKey vx -> do
    phase .= WaitKeyRelease vx keyState
```

The real work happens in the new branch for handling the `WaitKeyRelease` phase. By comparing the stored previous key state to the current key state, we can find the just now released key, if any, and write its value into the target register `vx`.

```
WaitKeyRelease vx prevState -> do
    case keyRelease prevState keyState of
        Just key -> do
            setReg vx $ fromIntegral key
            phase .= Fetch
        Nothing -> do
            phase .= WaitKeyRelease vx keyState
```

Note that when `keyRelease` finds `Nothing`, we pass the new key state to the next cycle; this is so that we can properly detect the release of keys that weren't yet pressed at the beginning of the `WaitKey` instruction. As for `keyRelease` itself, it simply looks for the index of the first key which was pressed `before` but isn't `now`:

```
keyRelease :: KeypadState -> KeypadState -> Maybe Key
keyRelease prev new = bitCoerce <$> findIndex released (zip prev new)
  where
    released (before, now) = before && not now
```

13.5.4 Special-purpose registers

There are some instructions that operate on registers that have a life of their own: the two timer registers change value once every `tick`, and reading twice from the random number source should give a different value.

We implement the delay register as a simple 8-bit register, and decrement it by one on every `tick` regardless of the execution phase:

```
data CPUState = CPUState
    { _timer :: !Byte
    ...
    }

initState :: CPUState
initState = CPUState
    { _timer = 0
    ...
    }

step :: Pure CPUIn -> CPU ()
step CPUIn{..} = do
    when tick $ timer %= fromMaybe 0 . predIdx
    use phase >>= \case ...
```

Then, the implementation of the instructions `GetTimer` and `LoadTimer` is trivial:

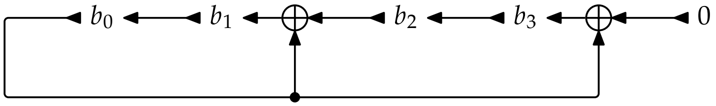
```
GetTimer vx -> do
    setReg vx =<< use timer
LoadTimer vx -> do
    x <- getReg vx
    timer .= x
```

We could implement the sound timer similarly, but instead just stub it out:

```
LoadSound regX -> do
    return ()
```

For the `Randomize` instruction, we require a source of pseudo-random numbers, of at least 8 bits. We are going to implement this using a so-called *linear-feedback shift register*, or *LFSR* for short. An LFSR is a shift register where the bit shifted out is fed back into the rest of the bits. This way, the register can keep operating without any further inputs, producing a new value in every cycle. The feedback structure makes the evolution of its value seem chaotic, which is why it's a good source of pseudo-randomness. On the other hand, since it's just a shift with some extra wires and XOR gates, it allows for efficient implementation in hardware.

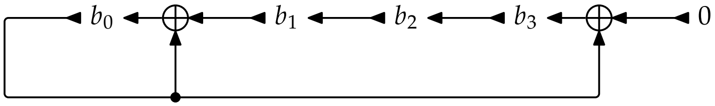
Let's look at an example of a 4-bit LFSR. We will think of the state as a vector of bits, so we'll use indexing from left to right:



Here, in every cycle all the bits shift to the left, and b_0 is fed back to positions 1 and 3 via XOR (which is just addition in $\mathbb{Z}_2$). So after one cycle, the new value becomes $\langle b_1, b_0 \oplus b_2, b_3, b_0 \oplus 0 \rangle$. Let's say we start from the state filled with all 1 bits; the evolution of the bit-vector will be as follows:

Step	State
1.	$\langle 1,1,1,1 \rangle$
2.	$\langle 1,0,1,1 \rangle$
3.	$\langle 1,0,1,1 \rangle$
4.	$\langle 0,0,1,1 \rangle$
5.	$\langle 0,1,1,0 \rangle$
6.	$\langle 1,1,0,0 \rangle$
7.	$\langle 1,1,0,1 \rangle$
8.	$\langle 1,1,1,1 \rangle$

It took 7 steps to get back to our initial state; since there are 16 possible 4-bit vectors, this means we have only covered roughly half of the state space. By carefully choosing the feedback structure, we can ensure that an n-bit LFSR will produce $2^n - 1$ different bit patterns before repeating; this is the best we can hope for, since the state containing all zeroes will always be a fixed point. One example of such a maximal LFSR for 4 bits, is the following structure:



Why this scheme produces 15 different values before repeating if the other one only produced 7 is a deep topic best approached via finite field theory; as such, we will omit the theoretical details here and just take maximal LFSR schemes from the literature.

We implement a generic linear-feedback shift register as a pure function over bit vectors, parameterized by a vector of coefficients describing where the shifted-out bit should be fed back:

```
lfsr
    :: (KnownNat n)
    => Vec (1 + n) Bit -> Vec (1 + n) Bit -> Vec (1 + n) Bit
lfsr coeffs (b0 :> bs) = zipWith xor (bs :< 0) feedback
  where
    feedback = fmap (b0 *) coeffs
```

For example, we can describe our first 4-bit LFSR as `lfsr (0 :> 1 :> 0 :> 1 :> Nil)` and the second one as `lfsr (1 :> 0 :> 0 :> 1 :> Nil)`. For the CHIP-8, we need 8-bit output, so we'll need an LFSR of at least 8 bits. We're going to go with a maximal 9-bit LFSR so that by taking its lowest 8 bits, we can cover the full set of possible 8-bit values, including 0.

We can look up a maximal 9-bit LFSR in polynomial notation in Wikipedia to find $x^9 + x^5 + 1$, which in our representation stands for `0 :> 0 :> 0 :> 1 :> 0 :> 0 :> 0 :> 0 :> 1 :> Nil` (since the x^9 term stands for the shift-out itself):

```
lfsr9 :: Unsigned 9 -> Unsigned 9
lfsr9 = bitCoerce . lfsr (unpack 0b0_0010_0001) . bitCoerce
```

Does it really work? Let's generate 511 iterations in the REPL and check that it's maximal, and its lower 8 bits cover all possible bytes:

```
> let values = toList $ iterate (SNat @512) lfsr9 1
> L.length . L.nub . L.sort $ values
511

> let bytes = L.map fromIntegral values :: [Byte]

> L.nub (L.sort bytes) == [minBound..maxBound]
True
```

So this looks good. We extend `CPUState` to contain a 9-bit register which we will update at every `step`:

```
data CPUState = CPUState
    { _randomState :: !(Unsigned 9)
    ...
    }

initState :: CPUState
initState = CPUState
    { _timer = 0x100
    ...
    }

step :: Pure CPUIn -> CPU ()
step CPUIn{..} = do
    randomState %= lfsr9
    ...
```

And when executing a `Randomize` instruction, we just read out the value of its lower 8 bits, via the given mask, for storage into `vx`:

```
Randomize vx mask -> do
    rnd <- fromIntegral <$> use randomState
    setReg vx $ rnd .&. mask
```

13.5.5 Graphics

The first graphical instruction we implement is `ClearScreen`, which simply writes the 64-bit value 0 to all 32 video buffer cells. Since we can only write to one cell per cycle, we add another `Phase` that goes through all 32 video addresses:

```
data Phase
    = ClearVideoBuf VidY
    | ...
```

The implementation of `ClearScreen` simply starts the process by going to this new phase:

```
ClearScreen -> do
    phase .= ClearVideoBuf 0
```

For the implementation of the behavior in this new phase, we add the utility function `writeVid` which sets both the address and the write-out lines going to the video buffer. We will reuse it shortly.

```
ClearVideoBuf y -> do
    writeVid y 0
    phase .= maybe Fetch ClearVideoBuf (succIdx y)
  where
    writeVid y val = do
        vidAddr .:= y
        vidWrite .:= Just val
```

For `DrawSprite`, we need a similar approach to draw all rows of the sprite, but with a bit more variability, since we only want to change the rows from V_y to $V_y + h$. Also, since the newly drawn sprite is combined with the existing screen contents (using XOR), it is not enough to just write out new values: we need to read the existing values first. This requires a two-step process, where in the first cycle we set the video address to the intended target, then in the next one combine the video read with the sprite data to compute the video write output to that same address. We will orchestrate that through two new phases, going from `DrawRead` to `DrawWrite` to the next row's `DrawRead`, until we run out of sprite data or screen space:

```
data Phase
    = DrawRead VidX VidY Nybble Nybble
    | DrawWrite VidX VidY Nybble Nybble
    | ...
```

We kickstart this process in the handling of the `DrawSprite` instruction by going to the `DrawRead x y height 0` phase. We also reset the flag register V_F to 0, to be set to 1 whenever we will encounter a collision during `DrawWrite`:

```
DrawSprite vx vy height -> do
    x <- fromIntegral <$> getReg vx
    y <- fromIntegral <$> getReg vy
    setFlag 0
    phase .= DrawRead x y height 0
```

In the `DrawRead` phase, we set up both address lines for `DrawWrite`: from the main memory, we need to read the sprite data (starting at the pointer), and from the video memory, we read the current row:

```
DrawRead x y height row -> do
   spriteAddr <- use ptr
   memAddr .:= spriteAddr + extend row
   vidAddr .:= y + extend row
   phase .= DrawWrite x y height row
```

In `DrawWrite`, we first check if there's anything left to do: if we have drawn enough rows, or if we'd need to draw out of screen, then we simply go back to the `Fetch` phase. Otherwise, the previous `DrawRead` phase has ensured we have the to-be-drawn sprite data in `memRead` (as an 8-bit value) and the current video contents for the target line (i.e. the background) in `vidRead`. As discussed earlier, we implement the horizontal translation to `x` by zero-extending to 64 bits and then right-shifting the sprite data. Then we combine it with the background, bitwise, in two ways: with XOR to compute the new screen pattern, and with AND to check for collisions.

```
DrawWrite x y height row -> do
   let finished = row == height
       outOfBounds = msb (add y row) == 1
   if finished || outOfBounds then phase .= Fetch else do
       let bg = vidRead
           sprite = bitCoerce (memRead, repeat low)
           sprite' = sprite `shiftR` fromIntegral x
           pattern = bg `xor` sprite'
           collision = (bg .&. sprite') /= 0
       when collision $ setFlag 1
       writeVid (y + extend row) pattern
       phase .= DrawRead x y height (row + 1)
```

The only remaining graphics-related instruction is `LoadHex vx`, which should set up the pointer register such that a subsequent `DrawSprite` instruction can be used to draw a 4×5 hexadecimal glyph showing the lower four bits of V_x. We implement this by using the reserved address space below `0x200`, and putting the glyph for the numeral n to address $8 * n$. This way, we can convert a 4-bit value into a glyph starting address by simply shifting it to the left by 3:

```
LoadHex vx -> do
    x <- getReg vx
    ptr .= toHex (fromIntegral x)
  where
    toHex :: Nybble -> Addr
    toHex x = extend x `shiftL` 3
```

Of course, for this to work, we will have to remember to put the right sprite data at the right addresses when we get to creating and hooking up the font ROM. To this effect, we create a vector containing the sprite data for all digits:

```
hexDigits :: Vec (16 * 8) Byte
```

By specification, each digit should be 4 pixels wide and 5 pixels tall. We can flex our creative muscles on this one, or just look up dumps of the original font online[9] and `pad` each 5-byte sprite to 8 bytes for easier addressing:

```
hexDigits = concat . map pad $
    (0xf0 :> 0x90 :> 0x90 :> 0x90 :> 0xf0 :> Nil) :>
    (0x20 :> 0x60 :> 0x20 :> 0x20 :> 0x70 :> Nil) :>
    (0xf0 :> 0x10 :> 0xf0 :> 0x80 :> 0xf0 :> Nil) :>
    (0xf0 :> 0x10 :> 0xf0 :> 0x10 :> 0xf0 :> Nil) :>
    (0x90 :> 0x90 :> 0xf0 :> 0x10 :> 0x10 :> Nil) :>
    (0xf0 :> 0x80 :> 0xf0 :> 0x10 :> 0xf0 :> Nil) :>
    (0xf0 :> 0x80 :> 0xf0 :> 0x90 :> 0xf0 :> Nil) :>
    (0xf0 :> 0x10 :> 0x20 :> 0x40 :> 0x40 :> Nil) :>
    (0xf0 :> 0x90 :> 0xf0 :> 0x90 :> 0xf0 :> Nil) :>
    (0xf0 :> 0x90 :> 0xf0 :> 0x10 :> 0xf0 :> Nil) :>
    (0xf0 :> 0x90 :> 0xf0 :> 0x90 :> 0x90 :> Nil) :>
    (0xe0 :> 0x90 :> 0xe0 :> 0x90 :> 0xe0 :> Nil) :>
    (0xf0 :> 0x80 :> 0x80 :> 0x80 :> 0xf0 :> Nil) :>
    (0xe0 :> 0x90 :> 0x90 :> 0x90 :> 0xe0 :> Nil) :>
    (0xf0 :> 0x80 :> 0xf0 :> 0x80 :> 0xf0 :> Nil) :>
    (0xf0 :> 0x80 :> 0xf0 :> 0x80 :> 0x80 :> Nil) :>
    Nil
  where
    pad = (++ repeat 0)
```

13.5.6 Memory access

The remaining three CHIP-8 instructions are the only ways in which RAM can be directly accessed. All three can access multiple addresses from a single instruction, so they necessarily take up multiple cycles and so will go through their own `Phase`s.

Let's start with `StoreBCD`. The real work is in computing the three BCD digits of an 8-bit value. We can take the so-called *shift-and-add-3* algorithm from the literature: alternating a shift-left and an add-3 step on a buffer that is initialized with the input in its lowest n bits, and finishes with the BCD digits in the highest $k * 4$ bits, where k is the number of decimal digits:

[9] One such source is at https://github.com/mattmikolay/chip-8/wiki/CHIP%E2%80%908-Technical-Reference#fonts

```
type BCDSize n = CLog 10 (2 ^ n)

bitwise
    :: (BitPack a)
    => (BitVector (BitSize a) -> BitVector (BitSize a))
    -> (a -> a)
bitwise f = unpack . f . pack

toBCD
    :: forall n. (KnownNat n)
    => Unsigned n
    -> Vec (BCDSize n) (Unsigned 4)
toBCD = fst . last . iterate (SNat @(n + 1)) (shift . add) . init
  where
    init x = (repeat 0, x)

    shift = bitwise (`shiftL` 1)

    add (digits, buf) = (map add3 digits, buf)
      where
        add3 d = if d >= 5 then d + 3 else d
```

Why and how this works (i.e. why exactly we have to add 3 to all intermediate digits at least 5) is beyond the scope of this book; but at least we can test it exhaustively on 8-bit input using the Clash simulator, with the help of the `toBCD` function from the *Calculator* project:

```
> let roundtrip = fromIntegral . fromBCD . toBCD
> :{
| let prop_roundtrip :: (KnownNat n) => Unsigned n -> Bool
|     prop_roundtrip x = x == roundtrip x
| :}
> all (prop_roundtrip @8) [minBound..maxBound]
True
```

For larger sizes, we can use QuickCheck:

```
> import Test.QuickCheck
> quickCheck (prop_roundtrip @64)
+++ OK, passed 100 tests.
```

Now that we have a way of computing `toBCD @8` that we are confident in, we just need to write the three elements of the resulting vector of nybbles into RAM. We write this is similar to the drawing instructions: a new Phase goes through three cycles, writing one cell at each step:

```
data Phase = WriteBCD Reg (Index 3) | ...

WriteBCD vx k ->
    x <- getReg vx
    addr <- uses ptr (+ fromIntegral k)
    writeMem addr $ toBCD' x !! k
    phase .= maybe Init (WriteBCD vx) (succIdx k)
  where
    toBCD' = fmap fromIntegral . toBCD . bitCoerce
```

The extra noise of toBCD' is merely because we operate on Byte (i.e. Word8) values, but toBCD goes from Unsigned 8 to Unsigned 4 values. Note that when we are done (i.e. when we run out of k), we don't go to the next Fetch phase directly; instead, we take a one-cycle detour via Init. This is because Fetch only works correctly if it is receiving on memRead the next instruction's high byte, which requires the value of the pc register to be put on the address line previously. However, to write the last decimal digit to memory, in the WriteBCD vx 2 state we set the memory address line to $I+2$, so we need the extra cycle provided by Init to arrange for the program counter's value to appear on the address bus before the Fetch.

Then the implementation of StoreBCD just starts this process by entering the WriteBCD phase:

```
StoreBCD vx -> do
    phase .= WriteBCD vx 0
```

The implementation of StoreRegs is along the same way: we go through the registers from 0 to the last one requested, write each one to the right memory address, and go back to Init when done:

```
data Phase = WriteRegs Reg Reg | ...

WriteRegs reg end -> do
    addr <- uses ptr (+ fromIntegral reg)
    writeMem addr =<< getReg reg
    phase .= if reg == end then Init else WriteRegs (reg + 1) end

StoreRegs vx -> do
    phase .= WriteRegs 0 vx
```

For LoadRegs, we proceed similarly. The one difference is that we need to always be one step ahead of ourselves: when we get to loading the value of register V_4, we take its value from the memory read line, which means the memory address line should have been set to $I + 4$ in the previous cycle. Accordingly, in *this* cycle we should set the address line to $I + 5$ in preparation for the next cycle. This also means that when we are loading the last register's value, we can just go straight to `Fetch` without setting the address line (which means it will get the value of the program counter, as default), instead of taking a detour via `Init`.

```
data Phase = ReadRegs Reg Reg | ...

ReadRegs reg end -> do
    setReg reg memRead
    if reg == end then phase .= Fetch else do
        let reg' = reg + 1
        addr <- uses ptr (+ fromIntegral reg')
        memAddr .:= addr
        phase .= ReadRegs reg' end
```

We jumpstart `ReadRegs` by setting the address line to I, the memory address corresponding to the source of loading V_0:

```
LoadRegs vx -> do
    addr <- use ptr
    memAddr .:= addr
    phase .= ReadRegs 0 vx
```

13.5.7 All CPU registers

Because we have developed `CPUState` and `Phase` piecewise, it is worthwhile to include their definitions in full in one place:

```
data Phase
    = Init
    | Fetch
    | Exec Byte
    | WaitKeyRelease Reg KeypadState
    | ClearVideoBuf VidY
    | DrawRead VidX VidY Nybble Nybble
    | DrawWrite VidX VidY Nybble Nybble
    | WriteBCD Reg (Index 3)
    | WriteRegs Reg Reg
    | ReadRegs Reg Reg
    deriving (Show, Generic, NFDataX)
```

```
data CPUState = CPUState
    { _pc, _ptr :: !Addr
    , _registers :: !(Vec 16 Byte)
    , _stack :: !(Stack 24 Addr)
    , _phase :: !Phase
    , _timer :: !Byte
    , _randomState :: !(Unsigned 9)
    }
    deriving (Show, Generic, NFDataX)
makeLenses ''CPUState
```

13.6 Simulation, take 1

Before we move on to the connective tissue filling in the blanks between the CPU and the video system, it is worthwhile to see our CPU in action by hooking it up to a high-level simulator in the same vein as what we did for the Brainfuck machine: a Haskell program that uses the CPU implementation as a pure `State` computation, which we connect to an SDL-based frontend.

We start with the implementation of the dual of the CPU: a function that maps one step's `CPUOut` to the next step's `CPUIn`. We will use mutable, unboxed arrays to store the contents of the RAM and the video buffer, so our function itself will also live in `IO`. The other parameters of the function are the keypad state (mapped to SDL keyboard inputs) and the once-per-frame timer tick:

```
world
    :: IOUArray Addr Word8
    -> IOUArray VidY Word64
    -> KeypadState
    -> Bool
    -> Pure CPUOut
    -> IO (Pure CPUIn)
world ram vid keyState tick CPUOut{..} = do
    memRead <- readArray ram _memAddr
    vidRead <- readArray vid _vidAddr

    traverse_ (writeArray ram _memAddr) _memWrite
    traverse_ (writeArray vid _vidAddr) _vidWrite

    return CPUIn{..}
```

To prepare the keypad state, we need a mapping of host keyboard keys to keypad locations. We will then compose it with the CHIP-8 keypad `layout` that we have already defined, to map keyboard keys to hexadecimal values:

```
keyboardLayout :: Matrix 4 4 Scancode
keyboardLayout =
    (Scancode1 :> Scancode2 :> Scancode3 :> Scancode4 :> Nil) :>
    (ScancodeQ :> ScancodeW :> ScancodeE :> ScancodeR :> Nil) :>
    (ScancodeA :> ScancodeS :> ScancodeD :> ScancodeF :> Nil) :>
    (ScancodeZ :> ScancodeX :> ScancodeC :> ScancodeV :> Nil) :>
    Nil
```

We then turn that into a 16-element vector of SDL scan codes by applying the CHIP-8 layout as a permutation to the flattened list of codes, using the `scatter` vector function from the Clash Prelude:

```
keyboardMap :: Vec 16 Scancode
keyboardMap = scatter (repeat ScancodeUnknown)
    (concat layout)
    (concat keyboardLayout)
```

We will use an `IOUArray` to hold the video buffer's contents, and `freeze` it into an immutable array to make a pattern rasterizer that reads the right bit from the right row. Not the fastest way to go, but not slow enough to matter compared to the CPU simulation itself.

```
rasterizeVideoBuf
    :: (MonadIO m) => IOUArray VidY Word64 -> m (Rasterizer 64 32)
rasterizeVideoBuf vid = do
    vidArr <- liftIO $ freeze vid
    return $ rasterizePattern $ \x y ->
      let fg = (0xe7, 0xc2, 0x51)
          bg = (0x50, 0x50, 0x50)
          row = vidArr ! bitCoerce y
      in if testBit row (fromIntegral (maxBound - x)) then fg else bg
```

The rest of the simulator is straightforward:

```
cpuMachine :: Pure CPUIn -> State CPUState (Pure CPUOut)
cpuMachine = runCPU defaultOut . step

videoParams = MkVideoParams
    { windowTitle = "CHIP-8"
    , screenScale = 20
    , screenRefreshRate = 60
    }
```

```
main :: IO ()
main = do
    fileName <- listToMaybe <$> getArgs
    fileName <- return $ fromMaybe (error "no image file name") fileName
    img <- BS.readFile fileName
```

1. We make an `IOUArray` for the main RAM, initializing it starting from `0x000` with the font data, then from `0x200` with the contents of the desired CHIP-8 game data.

```
    ram <- do
        ram <- newArray (minBound, maxBound) 0
        zipWithM_ (writeArray ram) [0x000..] (toList hexDigits)
        zipWithM_ (writeArray ram) [0x200..] (BS.unpack img)
        return ram
    vid <- newArray (minBound, maxBound) 0
```

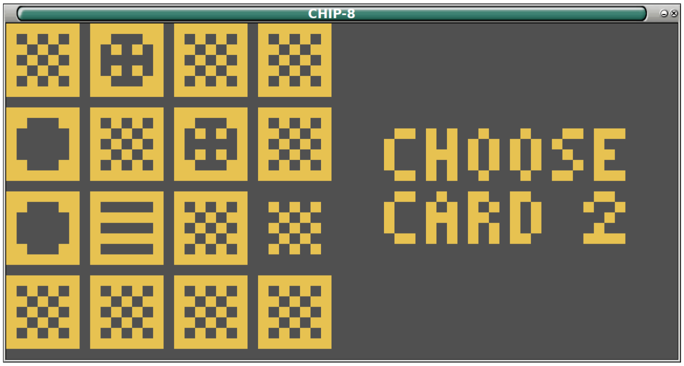
2. Our simulation's state consists of the next cycle's CPU input and the CPU registers.

```
    let initInput = CPUIn
            { memRead = 0
            , vidRead = 0
            , tick = False
            , keyState = repeat False
            }
    flip evalStateT (initInput, initState) $
      withMainWindow videoParams $ \events keyDown -> do
        guard $ not $ keyDown ScancodeEscape
        let keyState = fmap keyDown keyboardMap
```

3. In each frame, we run the CPU for multiple steps. The first simulation step will receive `True` as the timer tick input, and subsequent steps for the same frame will receive `False`.

```
        let sim tick = do
                (inp, s) <- get
                let (out, s') = runState (cpuMachine inp) s
                inp' <- liftIO $ world ram vid keyState tick out
                put (inp', s')
        sim True
        replicateM_ 5000 $ sim False

        rasterizeVideoBuf vid
```

We can run it with existing CHIP-8 games, old and new. This screenshot is showing David Winter's game *Hidden*:



13.7 The complete machine

With the major parts ready, it is time to turn our attention to integration. We take an approach similar to the Brainfuck computer:

- The logic board will consist of the CPU and its supporting memory components: the ROM containing the hexadecimal glyphs, the main RAM, and the CPU's copy of the video RAM.

- The top-level circuit connects the logic board to the peripherals via the video signal generator and the keypad scanner.

13.7.1 Memory address decoding

As far as the CPU is concerned, it is accessing a single unified 12-bit address space via the `memAddr`, `memRead` and `memWrite` pins. However, the first 512 bytes of that memory is reserved for implementation-specific use: we want to use it for font data, so we will put a 512-byte ROM there (and ignore writes to that area), and only use a $4096 - 512 = 3584$-byte RAM for the rest.

To connect multiple memory elements to the same address and data lines, we need to decide, based on the address line's value, which memory's read value should be put on the data line. Similarly, write requests should only be applied to the single right memory component.

For a machine as simple as the CHIP-8, writing the address decoding logic by hand isn't too bad: our only two components are the font ROM initialized using our vector of hexadecimal glyphs, and the main RAM initialized with the contents of a CHIP-8 game. We can read the type of logicBoard from the system overview diagram at this chapter's beginning, by circumscribing the CPU and the memory elements:

```
logicBoard
    :: (HiddenClockResetEnable dom)
    => FilePath
    -> Signal dom Bool
    -> Signal dom KeypadState
    -> Signal dom (Maybe (VidY, VidRow))
```

Inside logicBoard, after we instantiate our CPU, and create the video RAM, we can start making sense of its _memAddr output. Computing either the font ROM address (if it is below 0x200), or the main RAM address, is a matter of a simple comparison:

```
logicBoard programFile tick keyState = vidOut
  where
    CPUOut{..} = cpu CPUIn{..}
    vidOut = packWrite <$> _vidAddr <*> _vidWrite

    vidRead = blockRam1 NoClearOnReset (SNat @32) 0 _vidAddr vidOut

    fontAddr = enable (_memAddr .< 0x200) _memAddr
    ramAddr = enable (0x200 <=. _memAddr) (_memAddr - 0x200)

    -- Continued below
```

Note that we are deliberately *not* defining ramAddr as "whenever fontAddr is Nothing" or vice versa: this is to hint at the shape of this code if we had more than two components connected to the address bus.

Given these decoded addresses, we can connect them to their respective memory elements. Note that fontAddr and ramAddr contain Maybe Addr values, with Nothing denoting that that element is not the selected one. But all the memory primitives like blockRamFile require an input signal that contains an address, always, so how are we going to handle the Nothings?

Recall that these synchronous memory elements produce output delayed by one clock period. Let's say _memAddr's value was 0x012 in a given period. That means fontAddr will have value Just 0x012, while ramAddr is Nothing. If we connect fromJustX <$> fontAddr to the font ROM's address bus[10] (and, similarly,

[10] fromJustX is a Clash-specific version of fromJust that has better simulation properties

`fromJustX <$> ramAddr` to the main RAM), then in the next period, we will have a valid read result in `font`, and an undefined value in `ram`:

```
    font = rom (hexDigits ++ repeat 0 :: Vec 0x200 Byte) $
        fromJustX <$> fontAddr
    ram = packRam (blockRamFile (SNat @(0x1000 - 0x200)) programFile)
        (fromJustX <$> ramAddr)
        (liftA2 (,) <$> ramAddr <*> _memWrite)
```

When we then multiplex these two reads into the single relevant one, we use `font` exactly when `fontAddr` was valid *in the previous period*, and `ram` when `ramAddr` was valid. This way, the undefined read from the undefined result of `fromJustX Nothing` is never connected to `ramRead`.

```
    memRead = muxA
        [ enable (register False $ isJust <$> fontAddr) font
        , enable (register False $ isJust <$> ramAddr) ram
        ] .<|
        0
```

Similar to the `romFile` variants we've used earlier, the `blockRamFile` family of Clash primitives creates `BitVector`-containing memory, hence we need to `unpack` the read values, and `pack` the write requests, in the definition of our `ram`:

```
packRam
    :: (BitPack d) => RAM dom a (BitVector (BitSize d)) -> RAM dom a d
packRam ram addr = fmap unpack . ram addr . fmap (second pack <$>)
```

13.7.2 The top-level circuit

Not much work remains: we know the intended type of `topEntity` simply from the peripherals of the complete CHIP-8 machine: the 4 × 4 keypad and the VGA screen:

```
topEntity
    :: "CLK_25MHZ" ::: Clock Dom25
    -> "RESET"     ::: Reset Dom25
    -> "ROWS"      ::: Signal Dom25 (Vec 4 (Active Low))
    -> ( "COLS"    ::: Signal Dom25 (Vec 4 (Active Low))
       , "VGA"     ::: VGAOut Dom25 8 8 8
       )
```

Implementation-wise, the interfaces of `scanKeypad`, `video` and `logicBoard` pretty much prescribe the definition of `topEntity`. The 4×4 state of the keypad matrix scanner is flattened into a 16-element vector using exactly the same approach we have used in the simulator: by applying the keypad-to-value `layout` as a permutation.

```
topEntity = withEnableGen board
  where
    board rows = (cols, vga)
      where
        (cols, keypadState) = scanKeypad rows
        keyState =
            debounce (SNat @(Milliseconds 5)) (repeat False) $
            fmap (scatter (repeat False) (concat layout)) $
            concat <$> keypadState

        (frameEnd, vga) = video vidWrite
        vidWrite = logicBoard "image.bin" frameEnd keyState
```

At this point, we have built a complete CHIP-8 machine that can run existing CHIP-8 software. However, there is one improvement that is very instructive in understanding how real computers of the era worked: avoiding the duplication of video RAM. Since the change will be somewhat involved, we first write a `Signal`-level simulation of the complete `logicBoard`, to help with testing.

13.8 Simulation, take 2

The first simulator in this chapter took the CPU `step` function and ran it as a Haskell computation. Here, we instead take the full `logicBoard` as a `Signal`-to-`Signal` stateful circuit, and use the `simulateIO` interface to the Clash signal simulator.

The changes compared to the high-level CPU simulator are quite localized: since the main RAM and the CPU's copy of the video RAM are both inside the `logicBoard`, they don't need to be manually simulated in `world`. Note that we still need a copy of the video RAM in the simulator for rendering purposes; but that copy is write-only as far as `logicBoard` is concerned.

```
world
    :: IOUArray VidY Word64
    -> Maybe (VidY, VidRow)
    -> IO ()
world vid vidWrite = for_ vidWrite $ \(addr, row) -> do
    writeArray vid addr row
```

The `main` function then becomes an amalgamation of the ideas we've seen in the Brainfuck `Signal`-level simulator and the CHIP-8 high-level simulator:

1. We create a temporary file containing the initial memory contents in the file format used by `blockRamFile`, and initialize the simulator.

```
main :: IO ()
main = withSystemTempFile "chip8-.bin" $ \romFile romHandle -> do
    fileName <- listToMaybe <$> getArgs
    fileName <- return $ fromMaybe (error "no image file name") fileName
    img <- BS.readFile fileName
    hPutStr romHandle $ unlines $
        binLines (Just (0x1000 - 0x200)) (BS.unpack img)
    hClose romHandle

    sim <- simulateIO_ @System
           (uncurry (logicBoard romFile) . unbundle)
           (False, repeat False)
```

2. We create the array that will hold the simulator’s copy of the video buffer.

```
    vid <- newArray (minBound, maxBound) 0
```

3. We use SDL, this time without any additional state since the circuit state is now handled internally by `simulateIO_`. In each frame, we run the simulation for 1,001 cycles: once with a timer tick, and 1,000 times without.

```
    withMainWindow videoParams $ \events keyDown -> do
        guard $ not $ keyDown ScancodeEscape

        let keyState = fmap keyDown keyboardMap

        let step tick = do
                sim $ \vidWrite -> do
                    liftIO $ world vid vidWrite
                    return (tick, keyState)
        step True
        replicateM_ 1000 $ step False

        rasterizeVideoBuf vid
```

13.9 Memory contention

In our CHIP-8 computer, we have a piece of memory (the video buffer) that needs to be accessed by two components: the CPU and the video signal generator. This means we can get into a situation where two components want to read from two different addresses at the same time. So far, we have worked around this problem by giving each component its own copy of the video buffer that they can address to their heart's content. However, this is a cop-out: this approach simply doesn't scale beyond toy examples.

For example, on the COSMAC VIP, the total RAM was 1 or 2 kB depending on the model. For the 64 × 32 = 2048 pixels, we need 256 bytes, which is one-fourth (or one-eighth) of total memory, so including it twice would be quite a significant cost increase. And from our contemporary point of view, block RAM can be quite a limited resource depending on the FPGA board used: we can come up against this problem quite easily on larger designs.

Of course, in more complicated machines, there can be more components than just the CPU and the video generator that need access to shared memory. This situation in general is called *memory contention*, because there's a disagreement between various parties on what memory cell should be addressed.

In this section, we are going to remove the duplication of video memory, and instead put a requirement on the CPU to wait for its turn in case of a conflict. We are prioritizing the video system's access because the electron beam on the CRT waits for no one: if we don't have the `row` register ready to shift out the next pixel's color, we are going to get a visual artifact.

13.9.1 Waiting for `vidRead`

Before we decide on the details of scheduling, the first change we do is on the CPU side: we wrap the `vidRead` input in a `Maybe` and then start chasing down the type errors.

```
declareBareB [d|
  data CPUIn = CPUIn
      { memRead :: Byte
      , vidRead :: Maybe VidRow
      , keyState :: KeypadState
      , tick :: Bool
      } |]
```

The only knock-on effect of this *inside* the CPU is that in the `DrawWrite` phase, we might not have read the current pixel values (i.e. the background) yet. We will simply stay put in that case, retrying in the next cycle.

When the data to draw on is available from `vidRead` as `Just bg`, i.e. contingent on a successful read from the video buffer, the implementation matches the previous version (using `bg` from the pattern match instead of aliasing `vidRead`). The pseudo-random number generator and the delay timer are updated independently.

```
step :: Pure CPUIn -> CPU ()
step CPUIn{..} = do
    randomState %= lfsr9
    when tick $ timer %= fromMaybe 0 . predIdx

    use phase >>= \case        -- Other branches unchanged
        DrawWrite x y height row -> case vidRead of
            Nothing -> do
                spriteAddr <- use ptr
                memAddr .:= spriteAddr + extend row
                vidAddr .:= y + extend row
            Just bg -> do
                let finished = row == height
                    outOfBounds = msb (add y row) == 1
                if finished || outOfBounds then phase .= Fetch else do
                    let sprite = bitCoerce (memRead, repeat low)
                        sprite' = sprite `shiftR` fromIntegral x
                        pattern = bg `xor` sprite'
                        collision = (bg .&. sprite') /= 0
                    when collision $ setFlag 1
                    writeVid (y + extend row) pattern
                    phase .= DrawRead x y height (row + 1)
```

The one subtlety comes in the `Nothing` branch inside `DrawWrite`, where we do the same setup of `memAddr` and `vidAddr` that we have previously done in the `DrawRead` phase. This is needed because if the CPU doesn't get access to RAM in the given cycle, we need to be set up correctly to retry in the next cycle. For example, if we didn't set `memAddr` here, the next cycle's `DrawWrite` would see the memory read result from the program counter because of the default values of the `CPUOut` fields.

13.9.2 Removing the duplicate video buffer

The next change is in `logicBoard`: we remove its copy of the video buffer, and instead connect the CPU's video address and data lines directly to the rest of the circuit.

```
logicBoard
    :: (HiddenClockResetEnable dom)
    => FilePath
    -> Signal dom Bool
    -> Signal dom KeypadState
    -> Signal dom (Maybe VidRow)     -- New input
    -> ( Signal dom VidY             -- New output
       , Signal dom (Maybe VidRow)
       )
logicBoard programFile tick keyState vidRead = (_vidAddr, _vidWrite)
```

We will connect these new signals to the video system, where the rubber hits the road and the actual memory contention resolution will happen. The changes to the video board's interface is as expected: the write request is unbundled into separate address line (for CPU-originating reads) and data line parameters, and the memory read results become a new output signal:

```
video
    :: (HiddenClockResetEnable Dom25)
    => Signal Dom25 VidY               -- Newly unbundled
    -> Signal Dom25 (Maybe VidRow)
    -> ( Signal Dom25 Bool
       , Signal Dom25 (Maybe VidRow)   -- New output
       , VGAOut Dom25 8 8 8
       )
```

Before we move on the changes in `video`'s definition, for completeness's sake we round off the refactoring by changing `topEntity` to connect the new `logicBoard` and `video` signals:

```
topEntity = withEnableGen board
  where
    board rows = (cols, vga)
      where
        (frameEnd, vidRead, vga) = video vidAddr vidWrite
        (vidAddr, vidWrite) = logicBoard "image.bin" frameEnd keyState
    vidRead
        -- Rest unchanged
```

At this point, it is probably a good idea to also update our logic board-level simulator, to help with debugging potential issues in the CPU changes. For example, this makes it easy to try out what would happen if we didn't propagate the correct `memAddr` and `vidAddr` values in `DrawWrite` when stalled. Most of the simulator changes are straightforward by just following the type errors arising from the `logicError` change. Because the video buffer is not write-only anymore, we return the video read result from `world`:

```
world
    :: IOUArray VidY Word64
    -> VidY
    -> Maybe VidRow
    -> IO VidRow
world vid vidAddr vidWrite = do
    traverse_ (writeArray vid vidAddr) vidWrite
    readArray vid vidAddr
```

To make the types work out, we could change the definition of `step` inside `main` to simply always have a video memory read result available:

```
let step tick = sim $ \(vidAddr, vidWrite) -> do
        vidRead <- liftIO $ world vid vidAddr vidWrite
        return (tick, keyState, Just vidRead)
step True
replicateM_ 1000 $ step False
```

But that wouldn't really show us how the CPU behaves when stalled. Instead, we will allow video memory reads only in some awkward schedule; for example like this:

```
let step i = sim $ \(vidAddr, vidWrite) -> do
        let tick = i == 0
            allowVideoAccess = (i `mod` 23) `elem` [1, 8, 13]
        vidRead <- liftIO $ world vid vidAddr vidWrite
        return (tick, keyState, vidRead <$ guard allowVideoAccess)
mapM_ step [0..1000]
```

The choice of only serving the CPU's video reads on every first, eighth and thirteenth cycle of a repeating 23-cycle pattern is completely arbitrary and hopefully irregular enough that if the simulation shows the CPU working correctly, then whatever other scheme we end up with will also work.

13.9.3 Resolving memory contention

Now we get to the meat of this section: changing the implementation of the video signal generator to serve the CPU's video buffer read requests.

Our goal is to make sure we have the 64 bits of the next line available for loading into the `row` register whenever we need it. This means the single address signal connected to the block RAM element must take the value of the next line rendered. At other times, when the video rendering doesn't need memory access, we can instead route the CPU's `vidAddr` line to that same address signal. Similarly, the single read-out line coming from the block RAM needs to be used both as the new `row` value to `load`, and also as the `vidRead` input to the CPU.

Let's repeat the relevant parts of the definition of `video`, to see what exactly needs to be done:

```
video
    :: (HiddenClockResetEnable Dom25)
    => Signal Dom25 (Maybe (VidY, VidRow))
    -> ( Signal Dom25 Bool
       , VGAOut Dom25 8 8 8
       )
video write = (frameEnd, delayVGA vgaSync rgb)
  where
    lineStart = liftD (isRising False) $ (isJust <$> vgaX')

    address = bitCoerce <$> guardA lineStart vgaY'
    load = delayedRam (blockRam1 NoClearOnReset (SNat @32) 0)
        (address .<| 0) (fromSignal write)

    row = delayedRegister 0 $ \row ->
        mux (delayI False $ isJust <$> address) load $
        mux (delayI False newX) ((`shiftL` 1) <$> row) $
        row

    -- Other local definitions omitted
```

Now we have two possible addresses to choose from: to avoid confusion, we will call the one coming from the CPU `cpuAddr`, and the one needed for rasterization `vgaAddr`. Since `video` internally uses `DSignals` to track signal delay due to synchronous RAM, we start by converting `cpuAddr` and `write` to `DSignals` as well:

```
video
    :: (HiddenClockResetEnable Dom25)
    => Signal Dom25 VidY
    -> Signal Dom25 (Maybe VidRow)
    -> ( Signal Dom25 Bool
       , Signal Dom25 (Maybe VidRow)
       , VGAOut Dom25 8 8 8
       )
video (fromSignal -> cpuAddr) (fromSignal -> write) =
    (frameEnd, toSignal cpuRead, delayVGA vgaSync rgb)
  where
    vgaAddr = bitCoerce <$> guardA lineStart vgaY'
    -- Continued below
```

One simple design is to always set `address` to `vgaAddr` in the vertical visible region, and only use `cpuAddr` during the vertical blanking period. This is not the most efficient approach: we only really look at the value of `load` once per (physical) line, everything else is handled by the 64-bit shift register containing the current `row`, so there are lots of cycles during the visible period that could let the CPU access the video buffer. Regardless, we implement this approach first, because it is a design that was used by a lot of the computers of the era we are targeting; its other benefit is the easy predictability. A common design for games running on such architectures is to do as much computation as possible during the visible area, then switch over to drawing during the vertical blanking.

As an aside, there are also retro-architectures where the opposite makes sense: if the video system allows direct real-time manipulation of the currently drawn pixel, then we can let precisely timed code run during the visible period to draw the screen, and only do non-graphics-related computation during blanking. The Atari 2600 console is the most famous poster-child of this "racing the beam" approach, but it is also an important consideration for programs that go "beyond" what the hardware is "supposed to be" capable of, such as displaying more than 8 sprites at the same time on the screen on a Commodore 64 by reprogramming the video chip's registers while the screen is drawn.

We give the CPU access to the video RAM during the blanking period by connecting `vgaAddr` to the block RAM if `vgaY` is set, and to `cpuAddr` otherwise:

```
vblank = fromSignal $ isNothing <$> vgaY
address = mux vblank cpuAddr (vgaAddr .<| 0)

load = delayedRam (blockRam1 NoClearOnReset (SNat @32) 0)
    address (packWrite <$> cpuAddr <*> write)
```

Accordingly, we connect `load` to the `cpuRead` output if it corresponds to a CPU-originating address:

```
cpuRead = enable (delayI False vblank) load
```

Note that there is still a little bit of cheating going on here: we lean heavily on the fact that the Clash block RAM primitives are *dual port*. This means that we are able to write to the RAM while reading from it, even at an address unrelated to the read address. This is why we can connect `packWrite <$> cpuAddr <*> write` unconditionally to the write port of the video buffer; this also means the CPU doesn't have to include special logic to wait for writes to take effect. The real hardware RAM chips used in the COSMAC VIP, or any other contemporary home computer, were *single port*, allowing for either reading or writing in a given cycle, but not both at the same time.

Of course, even with dual port RAM, if there are multiple components that could potentially write to the same memory element, we need to implement write arbitration similar to the read side. In this book, the computers all follow this pattern of the CPU being the only component writing to video RAM; because write contention can be solved similarly, and because modern FPGA development boards all contain dual-port block RAM, we will stick to exploiting this shortcut instead of complicating the circuits with write-acknowledgment signals.

Before we move on to improve this design by letting the CPU access the memory anytime the video generator doesn't need it, let's measure the throughput we can get with this current approach. To do this, we will use the Clash simulator to run the video signal generator for a full frame, and count the number of cycles where the video buffer read output is available.

We can get one frame's worth of output by running between two consecutive `frameEnd` signals:

```
> let (frameEnd, cpuRead, _) = video (pure 0) (pure Nothing)
> let sim = sample $ bundle (frameEnd, cpuRead)
> import qualified Data.List as L
> let dropFrame = dropWhile (not . fst)
> let takeFrame (x:xs) = x : takeWhile (not . fst) xs
> let frame = takeFrame . dropFrame $ sim
> L.length frame
419200
```

As a sanity check, this frame size matches what we can calculate from the timing specification for the 60 Hz 640×480 VGA mode we use: including the front and back porches and the sync spike, horizontally we have $640+16+96+48 = 800$ cycles and vertically $480 + 11 + 2 + 31 = 524$, for a total of $800 \times 524 = 419200$ cycles per frame.

To calculate the throughput of reading from the video buffer, we simply count the `Just` values of `cpuRead`:

```
> L.length $ L.filter (isJust . snd) frame
35200
```

So this gives us a baseline figure of 35,200 video buffer access cycles per frame. Now we will change the video signal generator to allow video RAM reads in every cycle where there is no need to load a new value into the shift register. The change is actually quite small: we have already arranged for `vgaAddr` to be a `Just` value only when we need it, so instead of `vblank`, we can simply use its `Just`-ness for the arbitration:

```
addr = vgaAddr .<|. cpuAddr
cpuRead = enable (delayI False $ isNothing <$> vgaAddr) load
```

With this small change, we have dramatically increased the throughput and latency of CPU to video RAM access:

```
> L.length $ L.filter (isJust . snd) frame
418880
```

Exercises:

- Some of the second-wave CHIP-8 implementations differed from the original COSMAC VIP CHIP-8 in some details, and because of their widespread use, there are lots of CHIP-8 games out there that only work with these changes. Add switches that change the behavior of the following, historically problematic instructions:

  - Change the `ShiftLeft` and `ShiftRight` ALU operations to ignore V_y and use V_x both as input and output.
  - Change `ReadRegs` and `WriteRegs` to adhere to the original CHIP-8 spec and increment the pointer register's value.

  The switches can be either synthesis-time flags, or real physical switches that allow the user to fine-tune the CHIP-8 CPU's behavior for any given game.

- The shift-and-add-3 combinational circuit of `toBCD` is very deep: it involves eight shifts, and three conditional 4-bit additions per shift. This can be a problem (especially if scaled up to larger than 8 bits), because the depth of the circuit puts a limit on the maximum clock speed possible. Instead, we can add a new CPU `Phase` that runs a single `stepBCD` for eight cycles.

- The total size of `CPUState` includes the register to hold the `Phase`, and the largest `Phase` constructors are `DrawRead` and `DrawWrite`. We can reduce their size somewhat by storing the *remaining* number of rows and stopping when it hits 0. Requires a bit more calculations upfront to implement proper vertical clipping. Another small gain to be had is noticing that the X and Y coordinates are 6 and 5 bits, respectively, but the choice of register for each is only 4.

- In the video signal generator, after shifting out the 64 bits that make up one physical line, the value of the `row` buffer becomes 0. If we use `rotateL` instead of `shiftL`, we get back the original value after 64 rotations, meaning we are ready to draw the same line again. Use this to decrease the video signal generator's memory access further, so that we fetch from the video buffer only once per virtual line instead of physical line. This should give us 419,168 cycles of CPU memory access per frame. Hint: as written, `vgaX'` changes 65 times per line, including the final transition from `Just 63` to `Nothing`, so we need to be a bit more careful.

- Although we have completely skipped over sound, the CHIP-8's audio capabilities are so simple that we can easily implement them with an external active buzzer, i.e. a component that makes a buzzing noise whenever voltage is applied. Add a one-bit "sound enable" output controlled by the `LoadSound` instruction, and connect it to the buzzer.

- As written, the block RAM for main memory is initialized with a single game's data chosen at synthesis time. Since each game takes up a maximum of 3584 bytes, depending on the block RAM capacity of the targeted FPGA, we might be able to store tens or even hundreds of games. Implement a simple game selection mechanism where some input (for example, four switches for 16 games) allows the user to select between different block RAMs, each initialized with a different game's data.

13.10 Summary

- The CHIP-8 was **originally a virtual machine** running on very resource-constrained platforms, even for its time. In this chapter, we implemented it in hardware, starting by analyzing the instruction set, and figuring out the parts that are needed to support them.

- It is a **von Neumann machine** using 16-bit instructions stored in 8-bit-addressable memory. This requires a **separate CPU phase to fetch** the first byte vs. executing an instruction once we have both bytes.

- In previous chapters, video output was generated by computing each pixel's color from some internal state. In this chapter, we have moved on to **memory-backed graphics** instead: each pixel's intended color is stored in its own memory cell, as an index into a palette (of just two colors, in the case of CHIP-8).

- The synchronous memory used for the video buffer introduces delay, which caused us problems in making sure the VGA color lines were in sync with the CRT beam's position. Clash's `DSignal` type helps by **tracking signal delay statically**. We have engineered the video system to take care of synchronization internally, presenting to the rest of the system the full, consistent VGA signal.

- Since the CPU and the video signal generator both needs access to the video buffer, we needed to solve **memory contention** by **prioritizing the video system**. Fine-grained scheduling leads to lower latency and higher throughput for the CPU's video buffer access; the computers of the time usually used a much simpler arbitration scheme where **CPU access is only permitted during vertical blanking**.

14 Address decoding and memory maps

One important component of our CHIP-8 was the logic to decode the CPU's address output according to the following, very simple, memory map:

- 512 bytes of (font) ROM starting at address `0x000`
- 3584 bytes of RAM starting at address `0x200`

Let's recall the code we wrote that implements the above logic:

```
fontAddr = enable (_memAddr .< 0x200) _memAddr
ramAddr = enable (0x200 <=. _memAddr) (_memAddr - 0x200)

font = rom (hexDigits ++ repeat 0 :: Vec 0x200 Byte)
        (fromJustX <$> fontAddr)
ram = packRam (blockRamFile (SNat @(0x1000 - 0x200)) programFile)
        (fromJustX <$> ramAddr)
        (liftA2 (,) <$> ramAddr <*> _memWrite)

memRead = muxA
    [ enable (register False $ isJust <$> fontAddr) font
    , enable (register False $ isJust <$> ramAddr) ram
    ] <|. 0
```

The topic of this chapter is developing abstractions that allow us to rewrite the above code into a form that is much closer to the above textual description. These abstractions will help in further chapters as memory maps become more complex.

14.1 Room for improvement

Dedicating a whole chapter to this problem only makes sense if we are not satisfied with the hand-written code. So what are its problems that we hope to improve?

First of all, there is the disconnect between the three parts of our code: matching the address against subintervals, using each matched sub-address to create a memory component, and then using multiplexing the read results into a single `memRead`. We could have easily accidentally used `isJust <$> fontAddr` to select `ram`.

Second is just how verbose it is, when the actual information to characterize the whole memory map is just the sizes and types of components, and their place in the address space. Some of this verbosity is local, like how we check that `_memAddr` is at least `0x200` before subtracting `0x200` from it to calculate its intra-component offset; and some of this is global: compare the whole code to the two bullet points of text preceding it.

A much nicer approach would be taking a declarative description of the memory map, and generating everything else from it. Maybe something like the following:

```
memRead = fromMaybe 0 <$> memoryMap _memAddr _memWrite
    [ from 0x000 $ romFromVec (hexDigits ++ repeat 0 :: Vec 0x200 Byte)
    , from 0x200 $ ramFromFile (SNat @(0x1000 - 0x200)) programFile
    ]
```

Note that while we have to give the size (as a static parameter) for the RAM, we can infer the size of the ROM from its contents.

While this is about as concise as it could be (it corresponds directly to our two original bullet points), it overfits to the CHIP-8 example in some important regards:

- In some designs, the same memory component is connected to multiple parts of the address space. For example, imagine a 16-bit address space, with the same 4 kB RAM mapped both from `0x7000` and `0xf000`. This **sharing** is observably different from having two 4 kB RAM components, since the value written to `0x789a` is then also available for reading from `0xf89a`.

- Components can be shared between the CPU and other elements, leading to potential **access contention**. The CHIP-8 provided an example of this when accessing the video RAM, but since it has dedicated lines to it, this concern didn't affect the address decoder. In systems where the video RAM (or other shared memory) is mapped to the normal memory address space, the address decoder has to make sure the read-data-ready signal is correctly routed back.

- Just because a component communicates with the CPU via its address and data bus, doesn't necessarily mean it is a memory element: it could be a memory-mapped I/O peripheral. And the whole point of a peripheral is to interface with the wide world outside the CPU; so these components will have some **backpane connections** that need to refer to parts of our circuit that have nothing to do with address decoding.

Our design will be driven by the first requirement; we will come back to the other two later in this chapter, but their implementation will involve only fairly localized changes. We address sharing by assigning *identity* to components: when a component is declared, we get a handle pointing to it; that handle is then used when defining its place in the address space:

```
memRead = memoryMap_ _memAddr _memWrite $ do
    font <- romFromVec (hexDigits ++ repeat 0 :: Vec 0x200 Byte)
    ram <- ramFromFile (SNat @(0x1000 - 0x200)) programFile

    from 0x000 $ connect font
    from 0x200 $ connect ram
```

The point, of course, is that this allows us to write our motivating example as the following:

```
dataIn = memoryMap_ addr dataOut $ do
    ram <- ram0 (SNat @0x1000) -- 0-initialized RAM

    from 0x7000 $ connect ram
    from 0xf000 $ connect ram
```

And why the name `memoryMap_`, with an underscore tacked on? That is to mark it as "`memoryMap` without any backpane connections", similar to how `traverse_` is "`traverse` without any result". We will stick to these backpane-less versions of all our combinators until later in the chapter when we get to implementing backpane connections.

14.2 A whirlwind intro to Template Haskell

Before we move on to the implementation of `memoryMap_`, there is one more piece of the puzzle we have to discuss: Template Haskell. As much as it would be desirable to be able to write the code from the previous examples, our final version will instead look like the following:

```
memRead = $(memoryMap_ @Addr [|_memAddr|] [|_memWrite|] $ do
    font <- romFromVec (SNat @0x0200)
                [|hexDigits ++ repeat 0 :: Vec 0x200 Byte|]
    ram <- ramFromFile (SNat @(0x1000 - 0x200)) [|programFile|]

    from 0x000 $ connect font
    from 0x200 $ connect ram)
```

It turns out, as of the writing of this book, the Clash compiler is unable to handle complicated, recursive definitions of signals. We don't have this problem in other parts of this book, because all our other complex code lives *inside* the signal processing functions; but here, we are not defining the combinational part of a circuit (inside a Mealy machine or otherwise), we are defining the circuit itself.[1] Instead of generating `Signal` *values*, we are forced to generate the *code* of the circuit we want – and this is where Template Haskell comes into play.

For readers who haven't worked with Template Haskell before, the expected reactions are "ugh, what's with these weird `[| |]` brackets?!" and "how is that `$(memoryMap_ ...)$` even valid Haskell syntax?!". In this section, we very briefly review the small subset of Template Haskell that we will use in this chapter.

14.2.1 Expressions

Haskell expressions are represented by the type `Language.Haskell.TH.Exp`[2]. As one would expect, its constructors correspond to the term formers of Haskell; for example, if we have `fun :: Exp` and `arg :: Exp`, then `AppE fun arg :: Exp` corresponds to the Haskell term that is the application of the `fun` expression to the `arg` expression. In this chapter, we will never take expressions apart, and will only build them using `VarE`, `ListE` and `LetE`; as their names suggest, these correspond to variable occurrences, list literals, and `let` expressions, respectively. Everything else will be user-supplied, or static templates.

14.2.2 Splicing and quoting

The `$(...)` syntax instructs the compiler to splice the results of an `Exp`-valued computation into our source code. So when we write `memRead = $(memoryMap_ foo bar)`, the expectation is that `memoryMap_` is a function that, after application, results in the `Exp` that represents the term `muxA [...]`. The basic operation on `Exp`, of course, is to make larger expressions from smaller ones, according to the term formers of Haskell.

Note that we've been describing the process in terms of "computations resulting in an `Exp` result"; that is because the Template Haskell macros like `memoryMap_` are all run in a monad called `Q` (for "code quotation"). In our code, we are going to use only a single effect provided by `Q`, and that is generating fresh, unique variable names with the `newName` function. In its full generality, `Q` provides a lot of other facilities for compile-time inspection, but we are not going to discuss those. Since everything is in the `Q` monad, the `Language.Haskell.TH` module also exports handy

[1] See https://github.com/clash-lang/clash-compiler/issues/1536 for the very long thread in the Clash bug tracker for details.

[2] We will omit the `Language.Haskell.TH` namespace qualifier for the rest of this chapter, and assume that the Template Haskell support library has been imported unqualified.

type synonyms for quoted representations, for example `ExpQ` for `Q Exp`; similarly, we have `varE :: Name -> ExpQ` beside the raw constructor `VarE :: Name -> Exp`.

Pairing up with the yin that is `$(...)` is the yang of the quoting bracket `[| ... |]`. As the former takes a computation and splices its result into the code, the latter takes a Haskell term and gives back its `Exp` representation. In the example we've seen, `programFile` is the Haskell term referring to the parameter of `logicBoard`, and `[|programFile|]` is an expression of type `ExpQ` that returns the `Exp` representation of it.

Here is an example of Haskell code that uses both `$(...)` and `[| ... |]`, which comes from the code we will write later in this chapter, using another variable `maskeds :: [ExpQ]`:

```
masked = [| muxA $(listE maskeds) |]
```

If we have a program that processes the memory map description and generates the list of expressions `maskeds`:

```
maskeds =
    [ [| enable (register False $ isJust <$> $addr) $rd |]
    | (addr, rd) <- components
    ]
```

then the expression generated by `out` will be exactly what we wrote by hand initially.

14.2.3 Declarations

Our original hand-written code for the CHIP-8 is not a single definition of `memRead`; instead, it is a whole set of variable definitions, allowing us to use `ramAddr` both as the address line going into the `blockRamFile` as well as its multiplex selector. These definitions, and other possible declarations, are represented in Template Haskell as the `Dec` (and corresponding `DecQ`) type. Declarations are quoted using the syntax `[d| ... |]`. A single such quotation can contain a whole list of declarations, so its type is `Q [Dec]`. The only declarations we'll need here are variable definitions, which are written as `[d| $(varP v) = $someExpr |]`, where `v` is the newly bound variable name (as created by `newName`), `varP` creates a variable pattern, and `someExpr` is, of course, the right-hand side of the definition.

The following example code assembles the CHIP-8 memory map from fragments containing the address decoding logic, the component creation, and driving the final read data multiplexer. Its aim is to illustrate all the Template Haskell concepts introduced here, without doing any complicated processing yet.

```
chip8Example :: ExpQ -> ExpQ -> ExpQ     -- (1)
chip8Example addr wr = do
    -- (2)
    addrs@[addr1, addr2] <- replicateM 2 $ newName "addr"
    let addrDecs =
          [d| $(varP addr1) = enable ($addr .< 0x200) $addr
              $(varP addr2) = enable (0x200 <=. $addr) ($addr - 0x200)
          |]

    -- (3)
    rds@[rd1, rd2] <- replicateM 2 $ newName "rd"
    let rdDecs =
          [d| $(varP rd1) = rom (hexDigits ++ repeat 0 :: Vec 0x200 Byte)
                  (fromJustX <$> $(varE addr1))
              $(varP rd2) = packRam
                  (blockRamFile (SNat @(0x1000 - 0x200)) programFile)
                  (fromJustX <$> $(varE addr2))
                  (liftA2 (,) <$> $(varE addr2) <*> $wr)
          |]

    -- (4)
    let maskeds =
          [ [| enable selected $(varE rd) |]
          | (addr, rd) <- L.zip addrs rds
          , let selected = register False $ isJust <$> $(varE addr)
          ]

    -- (5)
    decs <- addrDecs <> rdDecs
    letE (pure <$> decs) [| muxA $(listE maskeds) |]
```

1. We write our macro as a function from the expressions holding the address and write signals (`_memAddr` and `_memWrite` in the CHIP8 example). It might be cleaner to use the type `Exp -> Exp -> Q Exp`, but at the use site in `logicBoard`, we will want to pass `[| ... |]`-quoted expressions as the arguments, and those have type `ExpQ`.

2. We create variable bindings for the addresses restricted for individual components. For simplicity's sake, in this example code we create both names `addr1` and `addr2` upfront.

3. We do the same for the read results of individual components. Note that the definition of the components refers to `addr1` and `addr2`. In our real implemen-

tation, the tricky part will be figuring out which restricted addresses to use for which component. Here, it's all hardcoded.

4. We create the masked versions of the component read-outs uniformly, by applying `enable (register False $ isJust <$> addr)`.

5. The total set of declarations is the union of the addresses and the component reads. The type of `[d| ... |]` is `Q [Dec]`, not `[Dec]` (since it can contain other, effectful splices), or `Q Dec` (since one `[d| ... |]` quotation can contain multiple declarations). However, `letE` expects a `[Q Dec]` as its first argument. We bridge this impedance mismatch by first binding to `decs` (so at this point we have `decs :: [Dec]`), and then turning each `Dec` element into a `Q Dec` by mapping `pure` over the list.

We can see what the generated code is from GHCi, by using `runQ` and then pretty-printing the result with Template Haskell's `ppr` function:

```
Main> fmap ppr $ runQ $ chip8Example  [|_memAddr|] [|_memWrite|]
```

Looking at the output, we can confirm that it matches our expectation in recovering the original, hand-written CHIP-8 memory map:

```
let {addr_0 = enable (_memAddr .< 512) _memAddr;
     addr_1 = enable (512 <=. _memAddr) (_memAddr - 512);
     rd_2 = rom (hexDigits ++ repeat 0 :: Vec 512 Byte)
         (fromJustX <$> addr_0);
     rd_3 = packRam (blockRamFile (SNat @(4096 - 512)) programFile)
         (fromJustX <$> addr_1) ((liftA2 (,) <$> addr_1) <*> _memWrite)}
 in muxA [enable (register False $ (isJust <$> addr_0)) rd_2,
          enable (register False $ (isJust <$> addr_1)) rd_3]
```

(The actual output of `ppr` is less readable, because every bound identifier is printed with is fully qualified name. So `<$>` is rendered as `Data.Functor.<$>`, and so on. We are omitting these qualifiers here for the sake of readability).

Looks great, so let's try actually using it in `logicBoard`, by replacing our hand-written definition of `memRead` with `$(chip8Example [|_memAddr|] [|_memWrite|]`. One constraint is that `chip8Example` has to be defined in a different module than where we want to splice its result; if we define it right next to `logicBoard` in the same module, we get the following GHC error message:

> ***GHC stage restriction***: `chip8Example` *is used in a top-level splice, quasi-quote, or annotation, and must be imported, not defined locally.*

Once moved somewhere else, we can indeed replace the hand-written definition. This concludes our small Template Haskell tutorial. The rest of this chapter is about using this facility to implement a nice declarative way of describing memory maps; in other words, it concerns itself with compiling the user code into the right Template Haskell fragments.

14.2.4 But what about Typed Template Haskell?

One shortcoming of using Template Haskell is that splicing a given, well-typed `ExpQ` into a bigger expression may lead to a non-well-typed expression because the larger context requires a different type. For a very banal example, imagine passing `[|True|]` as the first argument to `chip8Example`: on its own, it is the quotation of a valid Haskell expression, but it leads to the following output:

```
addr_0 = enable (True .< 512) True;
```

GHC typechecks the output of Template Haskell, so this will be detected; but because it is the generated code that has a type error, the error will necessarily be reported in terms of the generated code. And depending on how complicated the expression generator program is, it might not be obvious which argument to the generator is causing the problem. This difficulty increases even more when debugging the generator itself.

GHC's answer to these concerns is *Typed* Template Haskell, which enables `chip8Example` to be typed as:

```
chip8Example
    :: TExpQ (Signal dom addr)
    -> TExpQ (Signal dom dat)
    -> TExpQ (Signal dom dat)
```

In Typed Template Haskell, the typed quotation `[|| True ||]` has type `TExpQ Bool`, so it cannot be passed as an argument in the `TExpQ (Signal dom addr)` position.

Unfortunately, the rest of this chapter will not use Typed Template Haskell, for two reasons:

- The address comparison needs `(Num addr, Ord addr)`, while wrapping the per-component addresses in `register False` requires an `(NfDataX addr, HiddenClockResetEnable dom)` constraint; thus, the full type of `chip8Example` would need to be:

```
chip8Example
    :: (HiddenClockResetEnable dom, NFDataX addr, Num addr, Ord
    addr)
    => TExpQ (Signal dom addr)
    -> TExpQ (Signal dom dat)
    -> TExpQ (Signal dom dat)
```

Unfortunately, at the time of writing this book, handling typeclass constraints in Typed Template Haskell is an open problem.[3]

- Typed Template Haskell is an “all or nothing” affair: typed quotations are only valid in typed splices, and vice versa. So even for the bits of our library that don’t need typeclass constraints, we can’t use Typed Template Haskell and then integrate nicely with the rest of the library, since the top-level splice will be necessarily an untyped one.

14.3 A memory map DSL

The API we will create for defining memory maps is going to be monadic, to make it easy to carry the handles of the created component to the points in the code where they are connected to the memory address space:

```
do
    bootRom <- romFromFile (SNat @0x0800) [|"bootRom.bin"|]
    ram <- ram0 (SNat @0x1000)

    from 0x0000 $ connect bootRom
    from 0x1000 $ connect ram
    from 0x2000 $ connect ram
```

Here, `bootRom` and `ram` are bound to the results of the two component builders; the `do`-notation makes it seamless to pass them to later `connect` calls.

The above code also illustrates the three moving parts that need to come together to make everything work out:

1. When a component is created, we need to remember the Clash code we will need to generate (for example, a call to `romFile`), and we need to provide a handle that identifies that particular component at later `connect` calls.

2. `from` restricts all connections in its body to be active only when the current address matches the given interval. For example, in the

[3] See https://stackoverflow.com/a/65406350/477476

`from 0x0000 $ connect bootRom` line, the `bootRom` is only connected to the address range `0x0000` to `0x07ff`. Where does the upper bound come from? Note that we passed a (type-level) size argument to `romFromFile`. We will use this to mark the `bootRom` handle as being addressed by a signal of type `Index 0x0800`; and this means when we can use this information in the type of `connect bootRom`, propagating the type information to its encompassing `from` (or any other address matcher), which can use that to calculate the full target address range.

3. When we `connect` a component, we add the restricted address to the set of addresses that are connected to the given component. In other words, there is always a "current most restricted address" for every point of our memory map specification, and a given component's address input will be collected from these.

Below is the code we intend to generate from this example. Note that each `from` call corresponds to the definition of a new, `Maybe`-valued signal; these signals are collected per component, and then also used in the final output multiplexer. Of course, we could have a `connect` call in the top level as well, so it is simpler to just always suppose the type of the address is a `Maybe`-valued signal.

```
let rd1 = fmap unpack $ romFilePow2 "bootRom.bin"
        (bitCoerce . fromJustX <$> compAddr1)
    rd2 = blockRam1 NoClearOnReset (SNat @0x1000) 0
        (fromJustX <$> compAddr2)
        (liftA2 (,) <$> compAddr2 <*> wr)

    addr1 = (from' @(Index 0x0800) 0x0000 =<<) <$> addr
    addr2 = (from' @(Index 0x1000) 0x1000 =<<) <$> addr
    addr3 = (from' @(Index 0x1000) 0x2000 =<<) <$> addr
    compAddr1 = muxA [addr1]
    compAddr2 = muxA [addr2, addr3]
in
    muxA
        [ enable (register False $ isJust <$> addr1) rd1
        , enable (register False $ isJust <$> addr2) rd2
        , enable (register False $ isJust <$> addr3) rd2
        ]
```

Here, `from'` is the term-level workhorse function for matching continuous address ranges, going from a larger space `addr` to `addr'`, if the argument matches the given range:

```
from'
    :: forall addr' addr. (Integral addr, Ord addr, Integral addr',
    Bounded addr')
    => addr -> addr -> Maybe addr'
from' base addr = do
    guard $ addr >= base
    let offset = addr - base
    guard $ offset <= fromIntegral (maxBound :: addr')
    return (fromIntegral offset)
```

Note that in this latest example, we create three matched addresses, and then need to combine the second and the third one to get the effective address of the second component. In the `from 0x1000 $ connect ram` line, we want to emit the definition of `addr2`, and also, behind the scenes, remember that `addr2` is one of the possible address lines of the component identified by the handle `ram`; this is so that later on, when we get to generating each component's address signal, we can look up all the address variables that go into `compAddr2`.

14.3.1 Implementation

The two main types that we will build our API around is `Addressing`, a monad in which we describe addressing schemes; and `Handle`, which is created by declaring memory components, and can be `connect`ed to address sub-ranges. Both `Addressing` and `Handle` are parameterized by the address sub-range type, which, as we have seen, is used by the interaction of `from` and `connect`. Without knowing yet the representation of `Addressing` and `Handle`, looking at our latest example, we can come up with the following desired API:

```
type Addressing addr a
instance Monad (Addressing addr)

type Handle addr

romFromFile :: SNat n -> ExpQ -> Addressing addr (Handle (Index n))
ram0 :: SNat n -> Addressing addr (Handle (Index n))
from :: addr -> Addressing addr' a -> Addressing addr a
connect :: Handle addr -> Addressing addr ()
```

In `Addressing`, we need to collect three kinds of output: standalone declarations for things like `rd` and `addr`, a list of addresses per component for `compAddr`, and a single global list of the masked component outputs to be put in the final `muxA`. This suggests using a `WriterT` over the `Q` monad (we need the latter to come up with the fresh names for `rd1`, `rd2` etc.). We also need access to the write signal (for components like `ram0`), and the current restricted address signal. We can put these two in a `ReaderT`, giving the following definition for `Addressing`:

```
newtype Addressing addr a = Addressing
  { runAddressing ::
      ReaderT (Addr, Wr) (WriterT (DecsQ, Connections, [Component]) Q) a
  }
  deriving newtype (Functor, Applicative, Monad)
```

The `addr` type parameter is merely a phantom parameter because we can't use Typed Template Haskell; but at least we can use some type synonyms (all defined as `ExpQ`) just to keep track of the role of each generated fragment:

```
type Addr = ExpQ        -- Signal dom (Maybe addr)
type Wr = ExpQ          -- Signal dom (Maybe dat)
type Component = ExpQ   -- Signal dom (Maybe dat)
```

For `Connections`, we need an associative data structure that collects the per-component addresses in such a way that its `Monoid` instance combines results rather than overwriting them, as a `Map Name [Addr]` would do. Note that the `Monoid` instance of `[Addr]` provides exactly the right operator for combining sub-results; this suggests using a `MonoidalMap` (from the *Monoidal Containers* package):

```
import Data.Map.Monoidal as Map

type Connections = MonoidalMap Name [Addr]
```

Note that our code will crucially depend on the `Addressing` newtype *only* inheriting the `Monad` instance from its underlying representation, *not* the `MonadWriter` instance. The fact that, under the hood, we are building up three collections is an implementation detail, and we will write our combinators to populate the collections properly; but we don't want user code to break because, for example, an out-of-scope name makes its way into the list of `Components`.

To see how this representation of `Addressing` works, we can write the compiler that takes an addressing scheme, the input signals for the address and the data-out, and generates a single Template Haskell expression corresponding to the result of reading from the whole address space:

```
compile
    :: Addressing addr ()
    -> Addr
    -> Wr
    -> ExpQ
```

```
compile addressing addr wr = do
    -- (1)
    (decs, conns, outs) <- execWriterT $
        runReaderT (runAddressing addressing) (addr, wr)

    -- (2)
    let compAddrs = [ [d| $(varP nm) = muxA $(listE addrs) |]
                    | (nm, addrs) <- Map.toList conns
                    ]

    -- (3)
    decs <- mconcat (decs:compAddrs)
    letE (pure <$> decs) [| muxA $(listE outs) |]
```

1. We peel off all the layers of `addressing`, running it with the given address and write signals. Note that we represent addresses as `Maybe`-valued signals (where a `Nothing` value means the address doesn't match the current address sub-range), and of course the original address is always inside the outermost (full) address range; we will take care of this in `memoryMap_`.

2. For each component's connections, we generate its corresponding `compAddr` definition by taking the combination of all its incoming addresses.

3. The rest of the code should be familiar from our Template Haskell orientation example: we flatten the list of declarations, create our final output signal by multiplexing all outputs, and put everything under a `let`.

The public entry point `memoryMap_` calls `compile` with two locally bound variables for `addr` and `wr`; the purpose of this is to avoid generating large and redundant code when the user supplies a complicated expression for any of the two arguments:

```
type RawAddr = ExpQ -- Signal dom addr

memoryMap_
    :: RawAddr
    -> Wr
    -> Addressing addr ()
    -> ExpQ
memoryMap_ addr wr addressing =
    [| let addr' = Just <$> $addr; wr' = $wr
       in $(compile addressing [| addr' |] [| wr' |])
    |]
```

We also have everything to implement `from`. In fact, `from` is just a special case of address matching; in later chapters, we will add other matchers built in top of `matchAddr` as the need arises.

```
matchAddr
    :: ExpQ {- addr -> Maybe addr' -}
    -> Addressing addr' a
    -> Addressing addr a
matchAddr match body = Addressing $ do
    nm <- lift . lift $ newName "addr"
    let addr' = varE nm
    ReaderT $ \(addr, wr) -> do
        let dec = [d| $(varP nm) = ($match =<<) <$> $addr |]
        runReaderT
          (tell (dec, mempty, mempty) >> runAddressing body)
          (addr', wr)
```

We use `newName` (from the underlying `Q` monad) to create a fresh name, and want to bind it in a declaration to the result of applying the given matcher function to the current address signal. The only complication comes from the fact that we are changing types from `Addressing addr` to `Addressing addr'`. Hence, we can't run `body` using just `local`; instead, we need to peel off the `ReaderT` layer, emit the definition of `nm`, and then re-enter `ReaderT` with the address signal corresponding to the result of the match.

Note that the bulk of `matchAddr` is written in `ReaderT _ (WriterT _ Q)`, not in `Addressing`; we merely wrap it into the latter at the outermost layer. This pattern will repeat itself in all our combinator definitions, and the reason for it is that `Addressing` is our public interface, providing no direct way to accessing `MonadReader`, `MonadWriter` or `Q` effects. In the implementation, we can do all these by relying on the representation of `Addressing`, and using the newtype constructor of `Addressing` to shrink-wrap it before delivery. User code has no access to the newtype constructor unless we export it, ensuring this safety barrier.

Given `matchAddr`, we write `from` by using `from'` as the matcher. We need to splice both the result type and the start address into the matcher fragment; however, we have these not as quoted expressions, but as bona fide, compile-time values. Luckily, Template Haskell provides the `lift` function for reifying values into expressions; and for reifying types, we have `liftTypeQ` from the *lift-type* package. The extra typeclass constraints on `from` come from the constraints of `lift` and `liftTypeQ`.[4]

[4] GHC arranges behind the scenes for `Typeable` to always hold for any concrete type; however, on the term level, only first-order values can be unquoted, so the `Lift` constraint is a real restriction on the types of addresses we can work with.

```
import LiftType
import Type.Reflection (Typeable)
import qualified Language.Haskell.TH.Syntax as TH

from
    :: forall addr' addr a. (Typeable addr', Lift addr)
    => (Integral addr, Ord addr, Integral addr', Bounded addr')
    => addr
    -> Addressing addr' a
    -> Addressing addr a
from base = matchAddr [| from' @($(liftTypeQ @addr')) $(TH.lift base) |]
```

In fact, GHC can insert the `TH.lift` calls for us, whenever we refer from a fragment to something in the scope outside the fragment, so we can rewrite the above in more compact form as:

```
from base = matchAddr [| from' @($(liftTypeQ @addr')) base |]
```

14.3.2 Representing components

Next, we think about how to represent component handles. In this section, we will concentrate on a single memory component type: block RAM initialized from a file. We use this because it illustrates both the handling of compile-time arguments (to pass the RAM contents file name) and writable components.

```
ramFromFile
    :: SNat n
    -> ExpQ {- FilePath -}
    -> Addressing addr (Handle (Index n))
```

Note that the file path is passed as a quoted expression instead of a compile-time `FilePath` value. If we passed a value, and used `TH.lift` to reify that, it would mean that the filename is determined at compile time. However, we have seen in the Brainfuck and CHIP-8 chapters that it can be useful for high-level simulation if we can take the filename as a runtime argument set from the command line. We wouldn't want to recompile our simulator every time we want to test with a different memory image! We solve this by passing a fragment instead of a value, since that fragment can then be just a variable.

Another thing worth highlighting is that when a component is created, its address type (`Index n` in this case) has no relation to the current address space `addr`. They will need to be the same when we get to `connecting` the component, but there can be any number of `matchAddr` calls between the two ensuring this holds. Sharing components in the address map only makes sense if the same component can be connected to multiple address sub-spaces.

Inside `ramFromFile`, we need to emit the definition of the value read from the component, as in the previous examples:

```
    rd2 = blockRamFile size fileName
        (fromJustX <$> compAddr2)
        (liftA2 (,) <$> compAddr2 <*> wr)
```

We don't know yet what the definition of `compAddr2` will be: that will only be available once we have collected all later `connect` calls. But that's not a problem, as long as we have the *name* `compAddr2` at hand. When we make a new component, we will generate both the `rd` and the `compAddr` names, but only define the former. The latter will be defined in `compile`, from the collected address signals. Of course, for the signals to go to the right place, we need to remember (in the `Handle`) the generated name. Similarly, even though we define `rd2` here, we record it in the `Handle` so that `connect` can emit its masked version as one of the alternatives of the final `muxA`:

```
data Handle addr = Handle Name Name

ramFromFile size fileName = Addressing $ do
    rd <- lift . lift $ newName "rd"
    addr <- lift . lift $ newName "compAddr"
    (_, wr) <- ask
    let decs = [d| $(varP rd) = packRam (blockRamFile size $fileName)
                       (fromJustX <$> $(varE addr))
                       (liftA2 (,) <$> $(varE addr) <*> $wr)
               |]
    tell (decs, Map.singleton addr mempty, mempty)
    return $ Handle rd addr
```

In the type of `connect`, we require the `Handle` address type to match the current address type, thereby ensuring that the current `addr` can be passed as one possible address input during `compile`:

```
connect
    :: Handle addr
    -> Addressing addr ()
connect (Handle rd compAddr) = Addressing $ do
    (addr, _) <- ask
    let masked = [| enable (delay False $ isJust <$> $addr) $(varE rd) |]
    tell (mempty, Map.singleton compAddr [addr], [masked])
```

This short definition packs quite a punch: first, it creates an expression fragment that corresponds to taking the given component's read-out value *if that component was addressed through this particular address range*; then, it adds the current address to the component's list of addresses (recall that we are collecting results into a `MonoidalMap`, so multiple `connect` calls at different sites can build up the list keyed to a given `compAddr`); and finally, the masked read-out value is added to the list of all read-outs that goes into the output multiplexer.

14.3.3 More components

We can implement `romFromFile` similar to `ramFromFile`:

```
romFromFile
    :: SNat n
    -> ExpQ {- FileName -}
    -> Addressing addr (Handle (Index n))
romFromFile size fileName = Addressing $ do
    rd <- lift . lift $ newName "rd"
    addr <- lift . lift $ newName "compAddr"
    let decs = [d| $(varP rd) = fmap unpack $ romFilePow2 $fileName
                       (bitCoerce . fromJustX <$> $addr)
               |]
    tell (decs, Map.singleton addr mempty, mempty)
    return $ Handle rd addr
```

In fact, it is so similar that it begs to be unified. The general form of a read-write component (with no backpane signals, hence the underscore suffix of the name) is one that given the effective component-specific address and write signals, returns the code fragment that creates the read-out value:

```
readWrite_
    :: (Addr -> Wr -> ExpQ)
    -> Addressing addr (Handle addr')
readWrite_ component = Addressing $ do
    rd <- lift . lift $ newName "rd"
    addr <- lift . lift $ newName "compAddr"
    (_, wr) <- ask
    let decs = [d| $(varP rd) = $(component (varE addr) wr) |]
    tell (decs, Map.singleton addr mempty, mempty)
    return $ Handle rd addr
```

We can recover `ramFromFile` and `romFromFile` in a straightforward way:

```
romFromFile size fileName = readWrite_ $ \addr _wr ->
    [| fmap unpack $ romFilePow2 $fileName
           (bitCoerce . fromJustX <$> $addr)
    |]

ramFromFile size fileName = readWrite_ $ \addr wr ->
    [| packRam (blockRamFile size $fileName)
           (fromJustX <$> $addr)
           (liftA2 (,) <$> $addr <*> $wr)
    |]
```

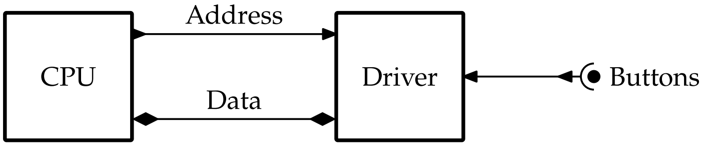
And for good measure, we will add two more: `romFromVec` and `ram0`, the latter standing for "`0`-initialized RAM". For the former, the same consideration applies to the vector of elements as for the filename argument of `ramFromFile` and `romFromFile`: by taking a fragment instead of a compile-time value, we allow for simulations to calculate the ROM contents at runtime.

```
romFromVec
    :: SNat n
    -> ExpQ {- Vec n dat -}
    -> Addressing addr (Handle (Index n))
romFromVec size xs = readWrite_ $ \addr _wr ->
    [| rom $xs (bitCoerce . fromJustX <$> $addr) |]

ram0
    :: SNat n
    -> Addressing addr (Handle (Index n))
ram0 size = readWrite_ $ \addr wr ->
    [| blockRam1 NoClearOnReset size 0
        (fromJustX <$> $addr)
        (liftA2 (,) <$> $addr <*> $wr)
    |]
```

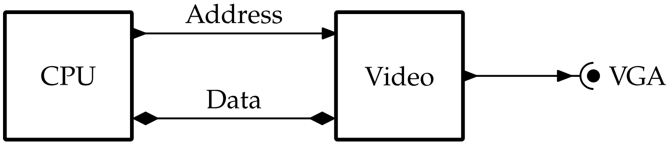
14.4 Backpane connections

Next, we extend our library with support for components with backpane connections. What we mean by that is a component that has pins on two sides: the CPU-facing side contains the address and data pins, and its backside contains pins facing the outside world.

For a concrete example, imagine a peripheral adapter connected to sixteen push-buttons that can be toggled by button presses. In this case, the adapter would implement debouncing each button, maintain its own internal 16-bit toggle state, and present it as two bytes mapped to two particular read-only memory addresses:



Another example would be if the CHIP-8's video buffer was mapped to the main address space. If we abstract the complete video system as a single component, its backpane connection is the VGA signal itself:



We can describe an addressing scheme that includes the first example with the tools we already have:

```
toggleBtnDriver
    :: (HiddenClockResetEnable dom)
    => Signal dom (BitVector 16)
    -> Signal dom (Maybe (Index 2))
    -> Signal dom (Unsigned 8)

toggleBtnExample :: Addressing (Unsigned 16) ()
toggleBtnExample = do
    ram <- ram0 (SNat @0x1000)
    btns <- readWrite_ @(Index 2) $ \addr _ ->
        [| toggleBtnDriver btns addr |]

    from 0x0000 $ connect ram
    from 0xf000 $ connect btns
    from 0xf800 $ connect btns
```

This works because like any other circuit, `toggleBtnDriver` consumes its backpane input signal as an argument; to instantiate it, we simply apply it on `btns` alongside the collected component-specific address `addr`.

The second example, however, requires some changes to our infrastructure. The reason for this is that these backpane outputs must have a way of getting out of the `memoryMap`. We can't directly bind them to return values of `readWrite`, since readWrite runs while we build our internal Addressing representation, whereas the signals to be returned will only exist once compile (as called by memoryMap) finishes assembling its fragment.

Before we actually solve this, let's look at what the end result should be: one possible version of the code to generate. Continuing the second example above, suppose we have the following video driver component (omitting constraints on dom for simplicity):

```
video
    :: Signal dom (Maybe (Index 4096))
    -> Signal dom (Maybe (Unsigned 8))
    -> (Signal dom (Unsigned 8), VGAOut dom 8 8 8)
```

This gives us 4096 individually addressable bytes of video RAM, in a format that almost looks like normal RAM. The difference becomes clear if we write out the code that we would generate if video were passed to readWrite_:

```
let rd1 = video compAddr1 wr
    addr1 = ...
    compAddr1 = muxA [addr1]
in
    muxA
        [ enable (register False $ isJust <$> addr1) rd1
        ]
```

This is clearly wrong: instead of the read-out signal from the video RAM, rd1 is now bound to the *pair* of signals coming out of video; moreover, the final result of this expression is only whatever byte is read out from the video RAM, with no way to refer to the VGA backpane output.

What we want to do is to be able to write code like the following:

```
(addrOut, dataOut) = cpu (dataIn .|> 0)

(dataIn, vgaOut) = $(memoryMap [|addrOut|] [|dataOut|] $ do
    (vid, vga) <- readWrite @(Index 4096) $ \addr wr ->
        [| video $addr $wr |]
    from 0xc000 $ connect vid
    return vga)
```

And have it generate code that binds dataIn to what comes out of the memory map, and vgaOut to the backpane of the vid component:

```
(dataIn, vgaOut) =
    let addr = Just <$> addrOut
        wr = dataOut
        (rd1, vga) = video compAddr1 dataOut
        addr1 = (from' @(Index 4096) 0 =<<) <$> addr
        compAddr1 = muxA [addr1]
    in ( muxA
            [ enable (register False $ isJust <$> addr1) rd1
            ] .<| 0
       , vga
       )
```

As we can see, the differences compared to our previous version are minor, but important:

- For each component (`video` in this case), its result is a pair consisting of the read output signal and the backpane connections.
- When a component is created, its result pair is bound to its `rd` variable (as before) and a new variable holding the backpane outputs.
- The final expression generated is similarly a pair of the `muxA` output and the result of the `Addressing` description.

The first two of these changes are straightforward; the real question is how to tie the code generated for the component `video` to the `return vga` statement at the end of our memory map description. The solution is similar to the representation of `Handle`: whenever a component with a backpane is created, its declaration will bind a fresh name, and that name is stored in the `Result` returned. Then, in `compile`, we take the expression from the `Result` and splice it into the second coordinate of a pair:

```
data Result = Result ExpQ

compile
    :: Addressing addr Result
    -> Addr
    -> Wr
    -> ExpQ
compile addressing addr wr = do
    (Result x, conns, outs) <- execWriterT $
        runReaderT (runAddressing addressing) (addr, wr)
    ...
    letE (pure <$> decs) [| muxA $(listE outs), $x |]
```

Indeed, this allows us to write a version of `readWrite` for components with backpane outputs, like our `video` example:

```
readWrite
    :: (Addr -> Wr -> Component)
    -> Addressing addr (Handle addr', Result)
readWrite component = Addressing $ do
    ...
    result <- lift . lift $ newName "result"
    let decs = [d| ($(varP rd), $(varP result)) =
                       $(component (varE addr) wr)
               |]
    tell (decs, Map.singleton addr mempty, mempty)
    return (Handle rd addr, Result (varE result))
```

And this degenerates to `readWrite_` nicely, as expected, by using a single `()` value as the backpane, and dropping the `Result`:

```
readWrite_
    :: (Addr -> Wr -> Dat)
    -> Addressing addr (Handle addr')
readWrite_ component = fmap fst $ readWrite $ \addr wr ->
    [| ($(component addr wr), ()) |]
```

However, `compile` as written can only handle the case of exactly one `Result`. What if we want to return none, do we need an almost duplicate implementation for `compile_`? And what if we have multiple components with backpane outputs, how would we return two or more `Results` from one `memoryMap`?

We can generalize in the number of `Results` by making a typeclass for `Backpane` results that is closed under products. This works out because Template Haskell fragments themselves are closed under products, in the obvious way: a pair of fragments can be turned into a fragment that is the `(,)` constructor applied on both fragments.

```
class Backpane a where
    backpane :: a -> ExpQ

instance Backpane () where
    backpane () = [|()|]

instance Backpane Result where
    backpane (Result e) = e
```

```haskell
instance (Backpane a1, Backpane a2) => Backpane (a1, a2) where
    backpane (x1, x2) = [| ($(backpane x1), $(backpane x2)) |]

instance (Backpane a1, Backpane a2, Backpane a3) =>
  Backpane (a1, a2, a3) where
    backpane (x1, x2, x3) =
        [| ($(backpane x1), $(backpane x2), $(backpane x3)) |]
```

It's truly unfortunate for situations like this that Haskell's tuple types are not inductively defined; we have to handle two-, three-, etc. tuples in as many instances. In this book, we won't need more than three-tuples; of course, the instances for larger tuples follow naturally.

We can now rewrite `compile` to work with any `Backpane` result:

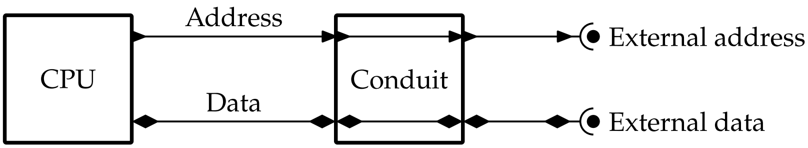
```haskell
compile
    :: (Backpane a)
    => Addressing addr a
    -> Addr
    -> Wr
    -> ExpQ
compile addressing addr wr = do
    ...
    letE (pure <$> decs) [| muxA $(listE outs), $(backpane x) |]
```

Giving us the public entry points `memoryMap` and `memoryMap_` as before:

```haskell
memoryMap
    :: forall addr a. (Backpane a)
    => Addr
    -> Wr
    -> Addressing addr a
    -> ExpQ
memoryMap addr wr addressing =
    [| let addr' = $addr; wr' = $wr
        in $(compile addressing [| addr' |] [| wr' |])
    |]

memoryMap_
    :: forall addr dat. ()
    => Addr
    -> Wr
    -> Addressing addr ()
    -> ExpQ
memoryMap_ addr wr addressing = [| fst $(memoryMap addr wr addressing) |]
```

This completes the implementation of backpane outputs. Before we move on to the next section, however, it is worth revisiting the `video` example from earlier. Suppose we are building a computer with a memory-mapped video buffer, and we would like to write a high-level simulator that interprets video RAM writes by drawing into an SDL texture. In the CHIP-8, we implemented exactly this, and it crucially depended on splitting the video system off the main logic board. But if we were writing our memory map as in the example, that means the video system is part of the main logic board, its connections tangled up with other internal signals.

Instead, we can add a new kind of component that is not really a component, merely a way of hooking an external read signal into part of our address space, by forwarding the address and write signals directly into the backpane:



The implementation is straightforward: we use `readWrite` with a component that simply returns the external read signal, and the current address and write signals.

```
conduit
    :: forall addr' addr. ()
    => ExpQ
    -> Addressing addr (Handle addr', Result, Result)
conduit rdExt = do
    (h, Result x) <- readWrite $ \addr wr -> [| ($rdExt, ($addr, $wr)) |]
    return (h, Result [| fst $x |], Result [| snd $x |])
```

We can rewrite our previous example using `conduit` to expose the video RAM address and write signals to the rest of the circuit, or to the simulator:

```
(addrOut, dataOut) = cpu (dataIn .|> 0)
(dataIn, (vidAddr, vidWrite)) = $(memoryMap [|addrOut|] [|dataOut|] $ do
    (vid, vidAddr, vidWrite) <- conduit @(Index 4096) [|vidRead|]
    from 0xf000 $ connect vid
    return (vidAddr, vidWrite))
```

14.5 Access contention

As we have seen in the previous chapter, when a memory element is shared between the CPU and some peripherals, sometimes memory access requests from the CPU cannot be fulfilled immediately. In a complex address space, at any given moment some components might be preempted while others are available for immediate access.

In the CHIP-8 chapter, we used a simple way of handling this situation by using `Maybe` to model availability. We can use the same approach here. Let's take our latest example and build a bit more code around it, to show the exact types of `vidAddr`, `vidWrite` and `vidRead`:

```
boardWithExternalVideo
    :: Signal dom (Maybe (Unsigned 8))
    -> (Signal dom (Index 4096), Signal dom (Maybe (Unsigned 8)))
boardWithExternalVideo = (vidAddr, vidWrite)
  where
    (addrOut, dataOut) = cpu (dataIn .|> Just 0)
    (dataIn, (vidAddr, vidWrite)) = $(memoryMap [|addrOut|] [|dataOut|]
    $ do
        (vid, vidAddr, vidWrite) <- conduit @(Index 4096) [|vidRead|]
        from 0xf000 $ connect vid
        return (vidAddr, vidWrite))
```

This works, because `memoryMap` is agnostic about the type on the data bus, as long as it is consistent between the components. However, what happens if we now want to add a piece of RAM to the start of the address space?

```
    (dataIn, (vidAddr, vidWrite)) = $(memoryMap [|addrOut|] [|dataOut|]
    $ do
        ram <- ram0 (SNat @0x8000)
        (vid, vidAddr, vidWrite) <- conduit @(Index 4096) [|vidRead|]

        from 0x0000 $ connect ram
        from 0xf000 $ connect vid
        return (vidAddr, vidWrite))
```

The problem here, is that we don't want to store `Maybe` values in the RAM – instead, since this RAM is an internal component with no other connections, and thus, no way of having contention, we want to wrap the values read from it in a `Just` constructor.

Since a `Handle` is represented by the names of its read-out and address variables, we can make a transformed `Handle` by creating a new variable for the former:

```
mapH :: ExpQ -> Handle addr' -> Addressing addr (Handle addr')
mapH f (Handle rd compAddr) = Addressing $ do
    rd' <- lift . lift $ newName "rd"
    tell ([d| $(varP rd') = $f <$> $(varE rd)|], mempty, mempty)
    return $ Handle rd' compAddr
```

This allows us to change our example so that the value read from `ram0` is always `Just`-wrapped:

```
    (dataIn, (vidAddr, vidWrite)) = $(memoryMap [|addrOut|] [|dataOut|]
    $ do
        ram <- mapH [|Just|] =<< ram0 (SNat @0x8000)
        ...
```

And what about the other direction, i.e. the CPU explicitly saying "I have no need for memory access at the moment"? Again, we can model that as a `Maybe` – this time on the address line. A `Nothing` address will simply not match any components:

```
matchJust
    :: Addressing addr a
    -> Addressing (Maybe addr) a
matchJust = matchAddr [| id |]
```

Note that in both cases of wrapping the address signal and the read-out signal in an extra layer of `Maybe`, the internal representation of address sub-spaces and read results is already a `Maybe`, but the two layers mean completely different things:

- A `Nothing` address, internally, means that the current address, whatever it is, does not match the given address sub-space. A `Just Nothing` address, on the other hand, means the current address sub-space is of type `Maybe a`, and the current address is in that subspace.
- A `Nothing` read-out value means no components match the current address. Real hardware usually has no way of detecting this – basically, some circuit-dependent bogus value will always be on the data bus. We can model that easily by applying `fromMaybe dummyValue` on the output of the memory mapper.
- A `Just Nothing` read-out value means access to the addressed component was pre-empted. This is a situation the CPU needs to handle. In a real hardware, a separate signal line tells the CPU to wait for the memory to be ready. Of course, this separate line is not raised if no component is selected. Due to this, if we are modeling CPU stalling with `Maybe`, we should make sure the default `dummyValue` is `Just` some value.

Overall, the details of how memory access contention is modeled depends on the design of the whole system. Instead of building a specific solution into the `memoryMap` DSL, we have seen how the `mapH` combinator can be used to implement specific designs.

14.6 Summary

- When multiple components are attached to the same address and data bus, there is some mapping of address sub-spaces to components.
- Since implementing such mappings is highly regular, we can librarize it. Due to technical issues ruling out some of the alternatives, our weapon of choice here is Template Haskell.
- We created a monadic API where the effects are declaring components, entering address sub-spaces, and connecting components. Because one component can appear under multiple address ranges, observable sharing is necessary.
- In this chapter, we added code for creating RAM and ROM components, and mapping them to contiguous address ranges. We can also add an escape hatch for hooking external components to the address map with the `conduit` combinator. In later chapters, we will expand this toolkit as needed.

15 Intel 8080

In this chapter, we build a CPU that is machine code compatible with the Intel 8080 microprocessor. In the next two chapters, we will use it to build replicas of two very different machines that were both originally using the real Intel 8080: the Space Invaders arcade video game machine, and the Compucolor II home computer.

15.1 History

The 8080 debuted in 1974, as the second eight-bit microprocessor product of Intel. For a brief time, it was one of the focal points of the personal computer revolution, with the Altair 8800 kit computer spawning multiple lines of computers based on its S-100 bus, running the CP/M operating system. However, in 1976 the Zilog Z80 arrived: created by a group mostly spun off from Intel, it was binary backwards-compatible with the 8080, with more registers, more instructions, easier-to-interface interrupt system and power requirements, and just plain faster. It didn't take long for the Z80 (and the MOS 6502, coming from a different lineage) to overtake the 8080 in the home microcomputer market.

So why build an Intel 8080 instead of a Zilog Z80? While the Z80 ultimately won out over the 8080, there are still important retrocomputers based on it. And since a reductive view of the Z80 is that it is two copies of the 8080 register set, with some new instructions and new interrupt logic, we can do more depth and less breadth in this chapter, and leave the expansion into a full Z80 as an exercise for the reader.

15.2 Veracity

When working on the CHIP-8, we started with an overview of the logical parts of the machine; however, since the original only existed virtually, these parts were mostly demarcated by our need to decompose the problem of describing a complete machine into more manageable parts.

Not so with the Intel 8080. The original was a physical standalone piece of hardware in a 40-pin DIP package. Designing and building a real computer using the 8080 means creating a circuit that has 40 connectors in the right layout communicating with the CPU. It is important to be upfront about our intentions here: the aim in this chapter is not to produce a drop-in replacement that could then be synthesized onto an FPGA with 40 GPIO ports and then hooked into an existing physical computer.

Instead, we will build a CPU that is binary compatible with the 8080 machine code instruction set, and that has comparable I/O capabilities that we will be able to take 8080-based machines and replicate them to be firmware- and software-compatible with the originals. Crucially, we want to build not just the CPU, but the whole machines in Clash; and so for input and output types, we will use whatever is convenient in Clash, and not necessarily in hardware.

Another question is timing accuracy. Even with a pin-compatible drop-in replacement, it is not necessarily a given that everything will happen with the same timings as a real Intel 8080. On the other hand, it is possible, and makes sense for certain applications, to build a non-drop-in-compatible implementation that still adheres to the original timings.

Timing accuracy can also mean different things in different contexts. At the lowest level, remembering that the clock is just an abstraction, we have parameters like how many nanoseconds the outputs lag behind the clock edges. In a fully synchronous circuit, like everything we build in Clash, we will not care about this level at all.

One level up, we have clock cycle-tied timings: the externally observable behavior of the CPU at every rising and falling clock edge. For example, fetching and executing a given instruction might take 6 clock cycles, with the address bus showing the program counter by the falling edge of the first cycle and keeping it like that for the next 2 cycles, reading the data bus contents at the rising edge of the second cycle, then changing the address bus and the data bus, and asserting the write-enable output, by the third falling edge. If we simplify the handling of rising and falling clock edges to just “inputs at start of a clock cycle” and “outputs at end of a clock cycle”, this is the level of accuracy that is within reach from Clash.

Then one level further up, we have instruction execution length, as measured in cycles. This means the same program will run for the same time on our replica as it does on a real CPU. This level of accuracy is commonly achieved by even naïve software emulator implementations.

So what level of timing accuracy is desirable?

On one hand, the more the better: a more accurate replica implementation not only means we can connect more original peripherals, but also that software written for the original machine, that was written to exploit the low-level details of the hardware, will run correctly. At the extreme end, we have software produced by the demo scene: programs that are written with the explicit purpose of squeezing the last possible drops of performance from a given limited hardware platform, by using in-depth knowledge of the target computer often surpassing even the original designers.

On the other hand, the more timing leeway we leave for ourselves, the easier our task becomes. As we will see in chapter 17, the Space Invaders arcade box runs fixed software that is written to use interrupts to do all time-dependent computations such as animations and updating the game state. For such a computer, it makes no observable difference even if the CPU takes longer or shorter to execute instructions, as long as it is "fast enough" to finish one interrupt-triggered procedure by the time the next interrupt comes. To use an analogy, the Pong game from chapter 9 runs the same if the clock speed is changed from 25 MHz to 40 MHz to produce 800 × 600 video instead of 640 × 480, because the game is updated 60 times per second in both cases. On the other hand, as we will see, the Compucolor II's disk drive is carefully tuned to the CPU speed, using cycle-counted busy loops in the disk accessing functions.

In this chapter, we will not care about timing accuracy at all. Our instructions, by and large, will run in fewer cycles than the original, so we could improve that by padding the microcode. Cycle accuracy should also be achievable by reorganizing the microcode slightly. However, there is still the elephant in the room: to avoid dealing with multiple clock domains, we run everything, including the CPU, at the pixel clock's rate. So while the real Intel 8080 could only run at about 2 MHz (depending on the model), in the Space Invaders machine we will run our core at 25 MHz to produce video at 640 × 480 resolution at 60 frames per second. We will return to this question in the Compucolor II chapter.

15.3 Interface

The real Intel 8080 has 40 pins. Four of them provide the chip with power, and two are clock signals, leaving 34 pins to our interest:

- `RESET` (active low) changes the program counter's value to 0x0000 and starts a new fetch, but otherwise keeps internal register values.

- Memory I/O consists of 16 `ADDRESS` and 8 `DATA` lines with an active-low `WR` to indicate memory writes. There is also a `DBIN` pin to signal memory reads.

- A pair of input `HOLD` and output `HLDA` pins for memory bus control. If `HOLD` is high, the CPU disconnects from the address and data buses to allow peripherals to do direct memory access. `HLDA` is the acknowledgment signal from the CPU that it has done so.

- A pair of input READY and output WAIT pins for memory access preemption: as long as READY is held low, the processor stalls on memory access, and this stalling is indicated by WAIT going high.

- The INT input triggers an interrupt. Since interrupts are the big new feature of the Intel 8080 compared to previous chapters, we will go into more detail on it below. The INTE output pin is high if the CPU has interrupts enabled, i.e. if it is accepting interrupts.

- The remaining single SYNC pin is used to expose a so-called *status word* on the data bus. When SYNC is high, the output on the DATA bus is not a byte to be written to memory; instead, each of its output bits takes on a meaning of its own.

To understand some of these status word bits, we need to do a bit of a detour into the concept of the confusingly named "machine cycles" as used by the Intel 8080 documentation. A "machine cycle" is not a single clock cycle; rather, it is multiple clock cycles that correspond to a certain phase of executing a given instruction. For example, the WO bit is low if the "current machine cycle" is writing to memory; but since the status word shares pins with the data bus, the data bus cannot also contain the byte to be written on the same clock cycle. The actual write (together with setting WR) will happen in a later clock cycle that still belongs to the current "machine cycle".

Keeping this in mind, the meaning of the various bits of the status word are as follows:

- INP and OUT: Instead of memory, an I/O peripheral should be used for the read (on DBIN) or write (on WR) in the current machine cycle. The 8080 can address 256 I/O ports by putting the port number instead of a memory address on both the low and the high 8 bits of the address bus. We will look into I/O ports in more detail below.

- WO: The current machine cycle is writing to memory to an I/O port.

- MEMR: The current machine cycle is reading from memory.

- INTA: Acknowledgement of the INT interrupt request. As we will see, this is used for much more than a simple notification.

- HLTA: The CPU has executed a HALT instruction and is now in the halt state.

- STACK: The address bus contains the stack pointer's value.

- M1: The current machine cycle is the fetching of the first byte of the next instruction.

For the interface of our Clash implementation, we make several simplifications:

- Clash has no notion of bidirectional signals: we will represent the data bus as a separate input and output signal, just like in previous chapters. In our model, the concept of detaching from the address and data buses doesn't make sense, so we omit the `HOLD` and `HLDA` signals completely. Direct memory access by peripherals such as video signal generators will simply preempt the CPU's memory access.

- We omit pins for observing the internal state, i.e. the `STACK`, `M1`, `WO`, `WAIT` and `INTE` pins. Adding these can be done as an exercise, but none of the peripherals we implement will need them. We do keep the `HLTA` output, so we can write tests that run the CPU until it halts.

- For the first version, we ignore `READY`. Later in this chapter, we will use a `Maybe Value` for the type of the data-in bus, using `Nothing` to denote preemption as in the CHIP-8. We also omit `MEMR`, instead deciding to always interpret the address output as a read request.

- Instead of having the data output do double duty on memory/port writes and the status word, we have a dedicated output signal for `INTA`, and merge the functionality of `INP` and `OUT` into a port-select bit of the address bus.

In summary, our simplified and consolidated interface is described by the following Clash data types:

```
type Value = Unsigned 8
type Port = Unsigned 8
type Addr = Unsigned 16

declareBareB [d|
  data CPUIn = CPUIn
    { dataIn :: Value
    , interruptRequest :: Bool
    } |]

declareBareB [d|
  data CPUOut = CPUOut
      { _addrOut :: Either Port Addr
      , _dataOut :: Maybe Value
      , _interruptAck :: Bool
      , _halted :: Bool
      } |]
makeLenses ''CPUOut
```

15.3.1 Interrupts

On the processors we have implemented so far — the Brainfuck CPU and the CHIP-8 — the only way of reacting to the outside world was to run a program that actively keeps looking for some input signal, for example by executing a `LD Vx, K` instruction on the CHIP-8. In contrast, **interrupts** allow peripherals to initiate a response from the CPU. Their name comes from the fact that they interrupt normal execution: the CPU is executing some program that is, for example, doing some arithmetic computation, and then suddenly, in response to an interrupt request, the program counter gets replaced by the address of the interrupt handler. It is then the interrupt handler's responsibility to handle the event that caused the interrupt, and then resume normal operation. The latter is made possible by the processor pushing the original program counter value to the stack before running the interrupt handler, thereby allowing a `RET` instruction to resume the original program.

If there is only one interrupt input pin, how does the handler know what event to react to? On the 8080, the answer is... quite complicated. When reading the abstract description of interrupts above, we might think that the interrupt handler has some fixed known address, maybe via an extra pointer indirection, and there's some logic inside the CPU to push the PC and jump to that handler. Not so on the Intel 8080!

If the CPU is in a state accepting interrupts (enabled by the `EI` and disabled by the `DI` instruction), and an interrupt request is received, then the current instruction finishes execution, and afterwards, as `SYNC` is set high, further interrupts are disabled, and the `INTA` bit of the status word is set to high. Afterwards, execution proceeds normally, except the program counter is not incremented.

Wait, what? How does that correspond to handling the interrupt request at all?! Well, it turns out circuitry outside the processor is supposed to detect when `INTA` is set, and sneakily replace the next byte input to the CPU with the opcode of an instruction to run the handler. This way, the CPU will fetch and execute that byte instead of the next normal instruction's first byte. Since the program counter is not incremented on an interrupt, after executing that single extra one-byte instruction, execution will proceed normally.

That is, if the byte squeezed in by the interrupt circuitry really is a full one-byte instruction! For example, a `CALL` instruction does pretty much what we'd want to start handling an interrupt request: it pushes the PC on the stack, before jumping to a subroutine. However, a `CALL` takes three bytes: the first byte is 0xcd, the machine code for `CALL`, and the next two bytes make up the 16-bit subroutine address. Clearly, it doesn't work to put the byte value `0xcd` on the data bus on `INTA`: the actual subroutine address would be taken from the next two bytes of the original program where it got interrupted: a completely random address.

The Intel 8080 solves this problem of its own making by having eight dedicated, one-byte `RST` instructions. The instruction `RST i` behaves the same as `CALL (8 * i)` (and, in contrast to its name, doesn't reset any internal state of the CPU at all), thus allowing eight different 8-byte interrupt handlers to be installed at addresses `0x0000`, `0x0008`, ... `0x0040`. Of course, interrupt handler code can also be longer than 8 bytes, simply by `JMP`ing to more code before `RET`urning to the normal operation.

To the surprise of no one, the Zilog Z80, designed to be "a better 8080", was released with a much more sensible interrupt protocol. While keeping the option to use 8080 mode for backwards compatibility, it also has a much simpler mode requiring no peripheral support at all (with a fixed interrupt handler address at 0x38), and a much more versatile mode where the byte read on `INTA` is taken as the lower 8 bits of an address containing a pointer to the interrupt handler, and a Z80-specific internal register provides the higher 8 bits.

In our implementation, we will stick to the 8080 protocol, requiring peripheral circuitry to provide the one-byte interrupt instruction. In our computers, we will use the following interrupt generator to turn events into interrupts:

```
type Interrupt = Unsigned 3

rst :: Interrupt -> Value
rst i = bitCoerce (0b11 :: Unsigned 2, i, 0b111 :: Unsigned 3)

interruptor
    :: (HiddenClockResetEnable dom)
    => Signal dom (Maybe Interrupt)
    -> Signal dom Bool
    -> (Signal dom Bool, Signal dom (Maybe Value))
interruptor irq ack = mealyStateB irqManager Nothing (irq, ack)
  where
    irqManager (irq, ack) = case irq of
        _ | ack -> do
            req <- gets $ fromMaybe 0
            put Nothing
            return (False, Just $ rst req)
        Just req -> do
            put $ Just req
            return (True, Nothing)
        Nothing -> do
            return (False, Nothing)
```

The interruptor takes 3-bit interrupt requests, and maintains a stack of size one. If a new interrupt comes in, the first output signal fires; this should be connected to the CPU's `interruptRequest` pin. When an `ack` is received from the CPU, the latest interrupt number is put on the output in the form of a `RST` instruction.

15.3.2 I/O ports

Besides addressing 65536 bytes of RAM, the Intel 8080 can also address 256 I/O ports. As we have seen, the CPU can let the outside world know via the status word that the next read or write is addressing an I/O port; in our version of the interface, we make this distinction explicit by making the type of the `addrOut` field `Either Port Addr`.

Although the interpretation of the port vs. address distinction is up to the components connected to the address pins, the intended meaning of these ports is to be connected to peripherals. This way, we can use 256 single-byte lanes to control peripherals without using up address space for memory mapping. In terms of programming, there is a dedicated pair of instructions `IN` and `OUT` to read from and write to a given I/O port.

An important consideration is that peripherals can react to being addressed, even when reading from them. We will see an example of this in chapter 18: the Compucolor II uses the TMS 5501 I/O controller. Reading from port #2 of this chip returns the highest-priority pending peripheral event. However, a side effect of this read is clearing said event, so the next read will return the next-highest-priority pending event, and so on.

15.4 Instruction set architecture

Similar to the CHIP-8, we start our study of the 8080's instruction set by looking at the registers available to the programmer. Afterwards, we will look at the machine instructions, skimming through ones that bring nothing new to the table compared to the CHIP-8, and only detailing the fundamentally new ones.

15.4.1 Registers

The Intel 8080 works with 8-bit data registers and 16-bit addressing. There are seven **general-purpose registers** named `A`, `B`, `C`, `D`, `E`, `H` and `L`. `A` is the accumulator: binary arithmetic operations such as addition always use `A` both as the first operand and also to store the result. The other six registers are grouped into three pairs `BC`, `DE` and `HL`; these pairs are used by certain pointer arithmetic instructions.

Although `BC` and `DE` can be used for some indirect memory operations, they are more useful for temporary operands during 16-bit address calculation: most of the memory access has to go through `HL`. For example, there are instructions to load and store the value of `A` from/to the address stored in `BC` or `DE`; but if we want to target any other register, the address has to come from `HL`.

The results of certain operations can be used in conditional jump instructions. For example, we can add a register's value to `A`, and then jump to a given address only if this addition overflows, i.e. if the result doesn't fit into the 8 result bits. This can be done via the five available **status flags**, which are updated by certain instructions:

- The *Zero* (`Z`) flag indicates if the latest arithmetic result was 0.
- The *Sign* (`S`) flag indicates if the latest arithmetic result has a high most significant bit.
- The *Parity* (`P`) flag is the complement of the number of high bits in the latest result. In other words, it is set if there is an even number of high bits in the result, and cleared otherwise.
- The *Carry* (`C`) flag indicates a carry (on addition) or a borrow (on subtraction).
- The *Auxillary Carry* (`AC`) flag indicates a carry (or borrow) from the low four bits to the high four bits.

The first four flags can be used for conditional jumps, calls and returns. The only use of the `AC` flag, on the other hand, is to convert hexadecimal values to BCD using the *Decimal Adjust Accumulator* (`DAA`, `0x27`) instruction: when `DAA` is used after an addition, it rearranges the value of the `A` register to fall within the `0x00`...`0x99` range, as a binary-coded decimal value.

The five flags together can also be thought of as an eighth 8-bit register, paired with the `A` register into a pair called `AF`. The reason for this is that the `PUSH` and `POP` instructions have four variations each: three of them push/pop a register pair (`BC`, `DE` or `HL`) on the stack, and the fourth one pushes `A` and the status flags. Since the contents of the stack can be accessed using normal memory operations, this also makes it possible to observe the layout of these five flags. Indeed, they are mapped to 8 bits in the following way:

7	6	5	4	3	2	1	0
`S`	`Z`	`0`	`AC`	`0`	`P`	`1`	`C`

Here, 0 and 1 stand for unused bits with the given value. There is no storage backing these bits: when the flag register is read or written wholesale (with the `PUSH AF` and `POP AF` instructions, respectively), the value of these bits always remain the fixed value.

To make instruction decoding easier, we use 3-bit numbers to represent registers, using the same encoding as the machine code of the instructions. Similarly, each flag is represented as its bit index into the flags register.

```
type Reg = Index 8

pattern RA, RFlags, RB, RC, RD, RE, RH, RL :: Reg
pattern RA = 7
pattern RFlags = 6
pattern RB = 0
pattern RC = 1
pattern RD = 2
pattern RE = 3
pattern RH = 4
pattern RL = 5

type Flag = Index (BitSize Value)

pattern FS, FZ, FAC, FP, FC :: Flag
pattern FS = 7
pattern FZ = 6
pattern FAC = 4
pattern FP = 2
pattern FC = 0
```

Although the primary mechanism by which the status flags are changes are the various arithmetic instructions, there is a pair of special instructions to change the `C` flag directly: `STC` sets it, and `CMC` flips it. So while there is no single instruction to *clear* `C`, a sequence of `STC; CMC` will do the trick.

15.4.2 Control flow

The internal 16-bit **program counter** is initialized to address `0x0000` at reset. Normally, it is incremented each time the next byte of an instruction is fetched. We can change its value with the `JMP` family of instructions: these all take a 16-bit direct argument for the intended jump target. There is one unconditional variant (machine code 0xc3) and eight conditional ones, two for each of the `Z`, `S`, `P` and `C` flags. The original Intel 8080 documentation has a separate mnemonic for each of these instructions (for example, `JNZ` for *jump if non-zero*, i.e. jump if the `Z` flag is cleared); we will use a more uniform decoded representation of `JMPIf Cond`:

```
data Cond = Cond Flag Bool
    deriving (Eq, Ord, Show, Generic, NFDataX)
```

This is slightly more generic than the real Intel 8080, since this allows branching on the `AC` flag's state; this means the instruction decoding will not be surjective, and also slightly wasteful on decoded instruction size. Since our aim here is not to implement the smallest 8080 core, we will make this and similar concessions to readability and regularity.

Before we move on, there are two instructions that affect control flow in two degenerate ways: `NOP`, which is the no-operation instruction, and thus doesn't change anything at all; and `HLT`, which halts all execution until a reset or interrupt arrives from the outside.

15.4.3 Stack

Unlike the CHIP-8, there is no internal stack in the Intel 8080. Instead, the 16-bit **stack pointer** (register `SP`) can be used to access any area of the main memory via the stack operations.

Similar to the CHIP-8's `CALL` and `RET` instructions, the Intel 8080 also has stack-based control flow operations. Just like direct jumps, these also come in nine variations: one unconditional and eight conditional ones, which again we will uniformly name `CALLIf` and `RETIf`. The `CALL` family puts the program counter's value at `SP` - 1 and `SP` - 2 and decrements `SP` by two; correspondingly, `RET` restores the program counter from `SP` and `SP` + 1 before incrementing it by two. In other words, the stack grows downwards and points at the next address to pop from.

We have already encountered the eight `RST` instructions: `RST i` behaves exactly like `CALL (8 * i)`, but only takes up one byte instead of three.

We can store and retrieve more from the stack than just the program counter, by using the `PUSH` and `POP` family of instructions. These are the only instructions that allow the `AF` register pair as an operand. The relevance of this is that this makes the representation of the flags register observable: after, e.g., a sequence of `PUSH AF` followed by `POP BC`, the `C` register now contains the eight bits of the flags register.

15.4.4 Inter-register traffic

This group of instructions moves data from one register (or pair of registers) to another. The most straightforward of these is the `MOV r1, r2` instruction, which simply changes the value of the r_1 register to the current value of the r_2 register.

`XCHG` swaps (*exchanges*) the values of the register pair `HL` and `DE`, i.e. `H` and `D` swap values, and so do `L` and `E`. There is no other register pair configuration for this instruction (e.g. to swap `BC` with `HL`).

For calculated or indirect jumps, we can use the `PCHL` instruction, which replaces the program counter with the `HL` register pair's value, using, unsurprisingly, the `H` register's value as the high byte and `L` as the low byte. Similarly, `SPHL` changes the stack pointer to the value of `HL`.

15.4.5 Immediate loads and memory access

The instruction `MVI r, imm` loads the 8-bit immediate value imm into the register r. Its two-byte sibling is `LXI rr, imm2` which loads the two-byte $imm2$ value into the register pair rr, which can also be `SP`.

There are versions of `MOV` and `MVI` where either the source or the target is `(HL)` instead of a register. These instructions load from, or write to, the single byte of memory addressed by the `HL` register pair. If we want to use `BC` or `DE` instead, we can use `LDAX BC` / `LDAX DE`, which is a version of `MOV` specialized to use the `A` register as the target. Similarly, `STAX BC` / `STAX DE` is a specialized version of `MOV` with the `A` register as the source.

In our representation, we unify all these not-fully-orthogonal instructions into a single `MOV` instruction where the left-hand side can be either a register, or the address stored in a register pair, and the right-hand side can be an immediate value (to be fetched from the next byte of the program), or a left-hand side:

```
data RegPair
    = Regs Reg Reg
    | SP
    deriving (Eq, Ord, Show, Generic, NFDataX)

pattern RAF, RBC, RDE, RHL :: RegPair
pattern RAF = Regs RA RFlags
pattern RBC = Regs RB RC
pattern RDE = Regs RD RE
pattern RHL = Regs RH RL

data LHS
    = Reg Reg
    | Addr RegPair
    deriving (Eq, Ord, Show, Generic, NFDataX)

data RHS
    = Imm
    | LHS LHS
    deriving (Eq, Ord, Show, Generic, NFDataX)
```

As explained earlier, beside 16-bit-addressable memory, there are also 256 bidirectional I/O ports available. The two instructions to access these are `IN port` and `OUT port`, both containing the port number as an immediate argument. Note that there is no register argument: `IN` always reads into `A`, and `OUT` always writes from `A`.

The immediate version of `LDAX` is `LDA addr`, which takes the 16-bit immediate argument as an address, and loads the value of `A` from the one byte there. Similarly, STA addr stores A at the given constant address. LHLD addr and SHLD addr similarly loads and stores HL at the given address, with H at the given address, and L at addr + 1.

In this category of instructions, the odd one out is XTHL, which exchanges HL with the top of the stack: L is swapped with the byte at (SP), and H is swapped with (SP + 1).

15.4.6 Arithmetic

The 8-bit arithmetic instructions ADD/ADC, SUB/SBB, AND, ORA, and XOR behave exactly as one would expect them: their first argument and their target is always A, whereas the second argument can be any left-hand side (i.e. another register, or the memory cell addressed by HL). These instructions also update all five status flags based on their result. The difference between ADD and ADC (*ADd with Carry-in*) is that the latter uses the C flag as an additional input for the initial carry. Similarly, SBB uses C as the initial borrow. This way, ADC and SBB can be used to implement multi-byte addition or subtraction by first clearing the C flag, then going from lowest to highest bytes.

The DAA instruction can also be modeled as an arithmetic instruction: just like the others in this group, it uses the A register's value and modifies all the status flags.

We will use the following datatype to represent these arithmetic functions, unifying ADD with ADC, and SUB with SBB, by storing a separate "use initial value of C" flag:

```
data ALU = ADD Bool | SUB Bool | AND | OR | XOR | BCD
    deriving (Eq, Ord, Show, Generic, NFDataX)
```

CMP is a special version of SUB that doesn't actually change A. The point of this instruction is that the flags are still updated, allowing for comparison with the other argument via the Z (for equality), C (for unsigned comparision) and S (for signed comparison) flags.

The CMA instruction replaces A with its bitwise complement, without updating any of the status flags.

The bit-rotating instructions RLC and RRC, and the bit-shifting instructions RAL and RAR operate in-place on the A register, and can only shift/rotate by one bit at a time. We will represent these as a choice of direction (Left or Right), and then a choice of Shift or Rotate:

```
data ShiftRotate = Shift | Rotate
    deriving (Eq, Ord, Show, Generic, NFDataX)
```

The one-byte increment and decrement instructions `INR` and `DCR` change their single LHS target in-place, updating all flags except `C`. Their two-byte counterpart is `INX` / `DCX`, operating on a register pair or `SP`.

There is also a special two-byte arithmetic instruction `DAD`, which adds a register pair (or `SP`) to `HL`, updating the `C` flag in the process.

15.4.7 Interrupt masking

The only remaining instructions are `DI` (for *Disable Interrupts*) and `EI` (for *Enable Interrupts*). Using `DI`, we can temporarily disable the servicing of interrupts. There is no queuing of interrupt requests in the CPU: requests received after `DI` are simply dropped.

As part of the interruption process, the CPU disables interrupts before the interrupt handler starts running. This is useful because it avoids the interrupt trashing that would happen if there was a new interrupt interrupting the interrupt handler (try saying that out loud!). It is the interrupt handler's responsibility to re-allow interrupts with `EI` at the right point.

15.5 Instruction decoding

Each 8080 instruction is encoded in 1 to 3 bytes. For example, `MOV B, D` is one byte (0x42); `OUT 0x64` is two bytes (0xd3 0x64), and `JMP 0x6543` is three bytes (0xc3, 0x65, 0x43). This means we need to do some decoding on the first byte to figure out how many more bytes we need to fetch for further decoding. We will make our life considerably simpler by writing a decoder that only looks at the first byte, and then letting the actual execution of the instruction handle any further fetching, if applicable.

We have made some considerable unification of the instruction set, at the cost of some "impossible" instructions like `MOV (Addr RBC) (RHS (Addr RDE)`, which would be a memory-to-memory transfer. The decoder will simply never emit such instructions in its output; however, as we will later see, we can easily make the execution unit generic enough that such impossible instructions are handled without hiccup. This is not to say that these non-orthogonality restrictions don't make sense in the context of the original Intel 8080 – our resource constraints are simply so far beyond the design considerations of the seventies that we can afford to be graceful instead of stingy.

```
data Instr
    = MOV LHS RHS
    | LXI RegPair
    | LDA
    | STA
    | LHLD
    | SHLD
    | XCHG
    | ALU ALU RHS
    | CMP RHS
    | SHROT (Either ShiftRotate ShiftRotate)
    | INR LHS
    | DCR LHS
    | INX RegPair
    | DCX RegPair
    | DAD RegPair
    | CMA
    | CMC
    | STC
    | JMP
    | JMPIf Cond
    | CALL
    | CALLIf Cond
    | RET
    | RETIf Cond
    | RST (Unsigned 3)
    | PCHL
    | PUSH RegPair
    | POP RegPair
    | XTHL
    | SPHL
    | IN
    | OUT
    | INT Bool
    | HLT
    | NOP
    deriving (Eq, Ord, Show, Generic, NFDataX)
```

We recover the specialized original instructions as pattern synonyms. The use of these specialized versions is to match the instruction mnemonics used in the original Intel 8080 documentation. This way, we can read our instruction decoder side-by-side with the reference documentation.

```
pattern MVI lhs = MOV lhs Imm

pattern ADD rhs = ALU (Add False) rhs
pattern ADC rhs = ALU (Add True) rhs
pattern SUB rhs = ALU (Sub False) rhs
pattern SBC rhs = ALU (Sub True) rhs
pattern AND rhs = ALU And rhs
pattern ORA rhs = ALU Or rhs
pattern XOR rhs = ALU XOr rhs
pattern DAA = ALU BCD (LHS (Reg RA))

pattern LDAX rr = MOV (Reg RA) (LHS (Addr rr))
pattern STAX rr = MOV (Addr rr) (LHS (Reg RA))

pattern RLC = SHROT (Left Rotate)
pattern RRC = SHROT (Right Rotate)
pattern RAL = SHROT (Left Shift)
pattern RAR = SHROT (Right Shift)

pattern DI = INT False
pattern EI = INT True
```

At this point, we could implement the instruction decoder as a simple 256-way branch, and call it a day:

```
decodeInstr :: Value -> Instr
decodeInstr 0x00 = NOP
decodeInstr 0x01 = LXI RBC
decodeInstr 0x02 = STAX RBC
...
```

Instead, we will write the decoder in a fourth of the size by exploiting the structure of the opcodes, along the same way we did in the CHIP-8 CPU when we split the two bytes into four nybbles. For this, we will need some of the branches to match on some bits only; for example, the bit pattern `11...010` corresponds to conditional jumps, with the "wildcards bits" from 5th down to 3rd encoding the condition itself (which status flag, and if the jump should be taken if the flag is set, or if it is cleared). Clash provides the Template Haskell macro `bitPattern` to write such matches on some subset of bits; this macro generates bit comparison operations that are efficient both for synthesis and simulation.

```
decodeInstr :: Value -> Instr
decodeInstr b = case b of
    $(bitPattern "01110110") -> HLT
    $(bitPattern "01......") -> MOV dest src
    $(bitPattern "00...110") -> MVI dest
    $(bitPattern "00..0001") -> LXI rr

    $(bitPattern "00111010") -> LDA
    $(bitPattern "00110010") -> STA
    $(bitPattern "00101010") -> LHLD
    $(bitPattern "00100010") -> SHLD
    $(bitPattern "00..1010") -> LDAX rr
    $(bitPattern "00..0010") -> STAX rr
    $(bitPattern "11101011") -> XCHG
    $(bitPattern "10000...") -> ADD src
    $(bitPattern "11000110") -> ADD Imm
    $(bitPattern "10001...") -> ADC src
    $(bitPattern "11001110") -> ADC Imm

    $(bitPattern "10010...") -> SUB src
    $(bitPattern "11010110") -> SUB Imm
    $(bitPattern "10011...") -> SBC src
    $(bitPattern "11011110") -> SBC Imm

    $(bitPattern "10100...") -> AND src
    $(bitPattern "11100110") -> AND Imm

    $(bitPattern "10101...") -> XOR src
    $(bitPattern "11101110") -> XOR Imm

    $(bitPattern "10110...") -> ORA src
    $(bitPattern "11110110") -> ORA Imm

    $(bitPattern "10111...") -> CMP src
    $(bitPattern "11111110") -> CMP Imm

    $(bitPattern "00...100") -> INR dest
    $(bitPattern "00...101") -> DCR dest
    $(bitPattern "00..0011") -> INX rr
    $(bitPattern "00..1011") -> DCX rr

    $(bitPattern "00..1001") -> DAD rr
    $(bitPattern "00100111") -> DAA
    $(bitPattern "00000111") -> RLC
    $(bitPattern "00001111") -> RRC
    $(bitPattern "00010111") -> RAL
    $(bitPattern "00011111") -> RAR

    $(bitPattern "00101111") -> CMA
    $(bitPattern "00111111") -> CMC
    $(bitPattern "00110111") -> STC

    $(bitPattern "11000011") -> JMP
    $(bitPattern "11...010") -> JMPIf cond
    $(bitPattern "11001101") -> CALL
    $(bitPattern "11...100") -> CALLIf cond
    $(bitPattern "11001001") -> RET
    $(bitPattern "11...000") -> RETIf cond

    $(bitPattern "11011011") -> IN
    $(bitPattern "11010011") -> OUT

    $(bitPattern "11101001") -> PCHL
    $(bitPattern "11..0101") -> PUSH rr'
    $(bitPattern "11..0001") -> POP rr'
    $(bitPattern "11100011") -> XTHL
    $(bitPattern "11111001") -> SPHL

    $(bitPattern "11111011") -> EI
    $(bitPattern "11110011") -> DI
    $(bitPattern "11...111") -> RST (bitCoerce op1)
    $(bitPattern "00000000") -> NOP
    _ -> NOP
  where
    op1 = slice (SNat @5) (SNat @3) b
    op2 = slice (SNat @2) (SNat @0) b

    dest = decodeLHS op1
    src = LHS $ decodeLHS op2

    rr = decodeRR $ slice (SNat @5) (SNat @4) b
    rr' = pushPopRR rr
    cond = decodeCond op1
```

Note that the bit pattern for `HLT`, `01_110_110`, overlaps with the bit patterns of `MOV`. Since `0b110` is the encoding of `(HL)`, i.e. memory addressing via the pointer in `HL`, the corresponding `MOV` would be the memory-to-memory transfer instruction

MOV (HL), (HL) which as we mentioned, is not a valid 8080 instruction.

In decodeInstr we use Clash's slice function to retrieve a BitVector from a wider value, by specifying the most and least significant bit's positions. We use this to extract the three-bit operands op1 and op2 and the two-bit register pair rr.The helper functions decodeLHS, decodeRR and decodeCond implement the decoding of these BitVectors. In decodeLHS, our internal register numbering scheme pays dividends, because we can just bitCoerce the three-bit encoding:

```
decodeLHS :: BitVector 3 -> LHS
decodeLHS 0b110 = Addr RHL
decodeLHS reg   = Reg (bitCoerce reg)

decodeRR :: BitVector 2 -> RegPair
decodeRR 0b00 = RBC
decodeRR 0b01 = RDE
decodeRR 0b10 = RHL
decodeRR 0b11 = SP

decodeCond :: BitVector 3 -> Cond
decodeCond cond = Cond flag b
  where
    (flag0, b) = bitCoerce cond :: (BitVector 2, Bool)
    flag = case flag0 of
        0b00 -> FZ
        0b01 -> FC
        0b10 -> FP
        0b11 -> FS
```

We also need an alternate version of the register pair rr for the PUSH and POP instructions: other instructions operating on register pairs can use either BC, DE, HL or SP, but these two use AF instead of SP.

```
pushPopRR :: RegPair -> RegPair
pushPopRR SP = RAF
pushPopRR rr = rr
```

15.6 Microcoded implementation

The largest difference between the CHIP-8 and the Intel 8080 is that the latter has much more varied control requirements. On the CHIP-8, all instructions were represented on two bytes, so we could implement a single unified way of fetching before execution. Except for a handful of exceptions, most instructions could also be executed in a single clock cycle. On the 8080, after the first byte of an instruction is fetched and decoded, we might find that we need to fetch more bytes. And the execution itself might be a trivial matter of changing internal state (like `EI`), or it could be as complex as `SHLD addr`, which involves a total of four roundtrips to memory, not counting the initial opcode fetch: two more fetches to get the target address, and then two writes to `(addr)` and `(addr + 1)`.

We will manage this complexity by opting for a microcoded implementation instead of hardwired control. We will write microcode in terms of micro-instructions that are simple enough that a micro-CPU can execute them in one clock cycle. We're really putting the *micro* in *microprocessor* here! Moreover, the mapping of each real 8080 instruction into a sequence of micro-instructions exposes more of the regularity and redundancy between instructions.

It should be stressed that on most microprocessors, the microcode is an implementation detail that is hidden from the programmer. The most famous retrocomputing exception is the Xerox Alto, where programmers could define new instructions and remap opcodes by writing custom microcode. In the rest of this chapter, the microcode we write for our 8080 has nothing to do whatsoever with the internals of the real physical hardware implementation from Intel. Both the design of the individual micro-instructions and the mappings from instructions to micro-operations that we present here are custom, and are optimized for readability instead of more real-world concerns like component count or circuit depth.

So what should our microcode look like? We start by looking at all the instructions and searching for patterns. First of all, we see that all instructions are either 1, 2 or 3 bytes long; moreover, in all 3-byte instructions, the extra two bytes are interpreted as a single 16-bit word: mostly as an address (in e.g. `LDA`), but sometimes as a value to be written into a register pair.

To keep things simple, we will use two separate micro-architectural registers: an 8-bit buffer and a 16-bit one. All transfer from and to memory goes through the 8-bit buffer; for example, we implement the instruction `MVI A, imm` in two steps: first fetching the immediate value into the buffer, then moving from the buffer into register `A`. Arithmetic calculations also operate on the buffer; for binary functions like `ADD`, we will take the other operand from the `A` register.

The primary use of the 16-bit buffer is for memory addressing. Looking at the entirety of the instruction set, we can see that we have four types of memory addressing:

- From the program counter `PC` (for every instruction just to fetch it; but also for the immediate operand, if any)
- From the stack pointer `SP`
- From an immediate address argument
- From an immediate port argument

- Via a register pair

Similarly to using the single value buffer as the hub of memory transfer, we will use the 16-bit buffer as the addressing hub. To implement something like `LDAX (BC)`, we first move the value of the register pair `BC` into the address buffer, then load the value buffer from the address in the address buffer, before finally overwriting the `A` register's value from the value buffer.

By making `PC` and `SP`-based addressing first class, we can make the incrementing of the program counter, and the incrementing (on pop) / decrementing (on push) of the stack pointer part of the addressing:

```
data InAddr
    = FromPtr
    | FromPort
    | IncrPC
    | IncrSP
    deriving (Enum, Bounded, Eq, Show)

data OutAddr
    = ToPtr
    | ToPort
    | DecrSP
    deriving (Enum, Bounded, Eq, Show)
```

One question we have side-stepped so far is how we are going to get data into and out of the address buffer, if only the 8-bit value buffer is connected to the data-in and data-out buses. Our solution to this is based on the observation that for all instructions, if one byte of the address buffer is read or updated, then the other byte will also follow shortly, during the execution of the same instruction. So for the address → value transfer, it is enough to come up with a pair of functions f, g that decomposes a 16-bit input into an 8-bit extract and a new 16-bit value, such that the extracts give the two bytes of the input: $256 * f(x) + f(g(x))) = x$, and g is an involution: $g(g(x)) = x$. Together, these two conditions ensure that we can extract the two bytes of x one by one, in two steps without any extra state, by writing $g(x)$ back into x. We will use an 8-bit rotation as a function with this property, and name it *twisting*:

```
twist :: Addr -> (Value, Addr)
twist x = (hi, lohi)
  where
    (hi, lo) = bitCoerce x :: (Value, Value)
    lohi = bitCoerce (lo, hi)
```

We can use it not only to extract values byte-by-byte:

```
twistFrom :: (MonadState s m) => Lens' s Addr -> m Value
twistFrom l = do
    (v, addr') <- twist <$> use l
    l .= addr'
    return v
```

But also to move bytes in the other direction:

```
twistTo :: (MonadState s m) => Lens' s Addr -> Value -> m ()
twistTo l x = do
    (y, _) <- twist <$> use l
    l .= bitCoerce (x, y)
```

Before we dive into further details, let's illustrate the ideas so far by giving the microcode for `SHLD`, a complex instruction, in an informal, but step-by-step manner:

```
SHLD:
    1. Load data buffer from: address stored in PC, increasing it
       Twist data buffer into address buffer
       Write data buffer to: nowhere

    2. Load data buffer from: address stored in PC, increasing it
       Twist data buffer into address buffer
       Write data buffer to: nowhere

    3. Load data buffer from: nowhere
       Overwrite data buffer from L
       Write data buffer to: address stored in address buffer

    4. Load data buffer from: nowhere
       Increment address buffer, not updating C flag
       Write data buffer to: nowhere

    5. Load data buffer from: nowhere
       Overwrite data buffer from H
       Write data buffer to: address stored in address buffer
```

As the above pseudo-code shows, the microcode is very regular compared to the instruction set. Each micro-step is specified in three parts: data in, internal state transition, and data out.

15.6.1 Representing microcode

If each micro-step of the microcode consists of a read, a micro-instruction, and a write, can't we just represent it as a list of 3-tuples?

```
type Microcode inAddr instr outAddr =
    [(Maybe inAddr, instr, Maybe outAddr)]
```

As the old joke goes, it only takes three steps to put an elephant in the fridge (1. open the fridge, 2. put in the elephant, 3. close the fridge), but it takes four to put in a giraffe (we have to take out the elephant first). We have a similar potential problem with this microcode representation.

We haven't enumerated all possible micro-instructions yet, but for this example, we only need two, to overwrite the value buffer from a register, and vice versa. Let's look at two micro-steps in isolation. The first one pushes the value of the register `A` onto the stack:

```
step1 = (Nothing, Get RA, Just DecrSP)
```

The second one fetches a byte from the program, and writes its value into the `B` register:

```
step2 = (Just IncrPC, Set RB, Nothing)
```

Now what happens if we try to do `step` and `step2` straight after each other?

```
steps = [step1, step2]
```

When executing `steps`, according to `step1`, after overwriting the value buffer with register `A`'s value, we should set the address bus to the value of `SP` (decrementing it in the process), and the data-out bus to `Just` the value of the value buffer. According to `step2`, on the other hand, *before* overwriting the value of register `B` from the value buffer, the latter should be loaded from memory, using the program counter (incrementing it in the process). But if we are using synchronous RAM, that means the value of `PC` needs to be put on the address bus in the *previous* cycle. So the question is, *what should be on the address bus and the data-out bus after the first cycle?* The elephant wants to eat its write-to-`SP` cake, but the giraffe is intent on having its read-from-`PC` one.

It looks like the addressing conflict is so severe, it has even led to a total confusion of metaphors!

We could get out of this conundrum by taking a more operational view instead. When we start executing microcode, we might need to do a read before the first step;

and then in each step, we execute the instruction and set the address bus based on either the current instruction's write address, or the next instruction's read address:

```
type Microcode inAddr instr outAddr =
    (Maybe inAddr, [(instr, Wedge outAddr inAddr)])
```

Here we use the *wedge sum* of `outAddr` and `inAddr`, which is isomorphic to `Maybe (Either outAddr inAddr)`, with the idea being that if we have `Here outAddr` it means the current step wants to write, and a `There inAddr` comes from the next step.

However, this representation makes it harder to make microcode out of smaller parts, even though *conceptually* we still want to think of each step as a triple of read-before, do-during and write-after. We will use this representation as a normal form during execution, but we want to keep a concatenable representation for the description.

We solve this by using a nonempty list datatype that is indexed by the addressing at its two ends. If we have a microcode fragment `frag1 :: Steps True False`, its type tells us that on the front end, the first step has a read requirement (the `True` index value), and on the back end, the last step has no write requirement (the `False` index value). So if we have another fragment `frag2 :: Steps False True`, we should be able to create their concatenation `frag2 >++> frag1 :: Steps False False`, but the type checker should reject `frag1 >++> frag2` since that would lead to a conflict in the middle.

To achieve this, we first of all need to track statically if the read and write addresses are set or not. We create an indexed version of the `Maybe` datatype, where the boolean index tells the `isJust`-ness of the value:

```
data IMaybe (isJust :: Bool) a where
    INothing :: IMaybe False a
    IJust :: a -> IMaybe True a
deriving instance (Show a) => Show (IMaybe isJust a)

fromIMaybe :: IMaybe free a -> Maybe a
fromIMaybe INothing = Nothing
fromIMaybe (IJust x) = Just x
```

So given the `isJust`-ness of a step's postamble and the `isJust`-ness of the next step's preamble, when are they compatible? Always, unless both are `True`. We provide a custom `TypeError` for this case, to improve the error message on conflict.

```
class Impossible where
    impossible :: a

type family Compat post1 pre2 where
    Compat True True =
        ( TypeError (Text "Conflict between postamble and next preamble")
        , Impossible
        )
    Compat post1 pre2 = ()
```

One `Step` is still a triple of values, but it presents the `isJust`-ness of its addressing coordinates as indices:

```
data Step pre a post hasPre hasPost where
    Step
        :: IMaybe hasPre pre
        -> a
        -> IMaybe hasPost post
        -> Step pre a post hasPre hasPost
```

Multiple `Steps` are non-empty cons-lists where the head `Step` is `Compatible` with the tail `Steps`. Also, since we will need to store the microcode in hardware, and index it with a micro-program-counter in the CPU state, we will eventually need to convert it into a vector, not a list, of `(instr, Wedge outAddr inAddr)` pairs. For this reason, we also track the length of `Steps` in the `n` index.

```
data Steps pre a post (n :: Nat) hasPre hasPost where
    One
        :: Step pre a post hasPre hasPost
        -> Steps pre a post 1 hasPre hasPost
    More
        :: (Compat hasPost1 hasPre2)
        => Step pre a post hasPre1 hasPost1
        -> Steps pre a post n hasPre2 hasPost2
        -> Steps pre a post (1 + n) hasPre1 hasPost2
```

This representation allows easy concatenation if the two ends are `Compatible`; in fact, this concatenation operation is the reason we are using this representation. We also provide a function so that users can create singleton `Steps` without the hassle of the `Step` constructor:

```
infixr 5 >++>
(>++>)
    :: (Compat hasPost1 hasPre2)
    => Steps pre a post n hasPre1 hasPost1
    -> Steps pre a post k hasPre2 hasPost2
    -> Steps pre a post (n + k) hasPre1 hasPost2
One x >++> ys = More x ys
More x xs >++> ys = More x $ xs >++> ys

step
    :: IMaybe hasPre pre
    -> a
    -> IMaybe hasPost post
    -> Steps pre a post 1 hasPre hasPost
step hasPre x hasPost = One $ Step hasPre x hasPost
```

All that remains is normalizing into individual steps with at most a single addressing directive between each pair. Note how `go` rules out the case where both the current postamble and the next preamble is an `IJust` value: in that case, `hasPost1` and `hasPre2` are both `True`, and the typeclass application `Compat True True` in the type of the `More` constructor reduces to `Impossible`.

```
stepsOf
    :: Steps pre a post n hasPre hasPost
    -> (Maybe pre, Vec n (a, Wedge post pre))
stepsOf xs = case go xs of (hasPre, ys) -> (fromIMaybe hasPre, ys)
  where
    go :: Steps pre a post n hasPre hasPost
       -> (IMaybe hasPre pre, Vec n (a, Wedge post pre))
    go (One (Step pre x post)) =
        (pre, singleton (x, wedgeLeft $ fromIMaybe post))
    go (More (Step pre x post) xs) =
        let (pre', ys) = go xs
            combined = case (post, pre') of
                (INothing, pre) -> wedgeRight $ fromIMaybe pre
                (post, INothing) -> wedgeLeft $ fromIMaybe post
                (IJust _, IJust _) -> impossible
        in (pre, (x, combined) :> ys)
```

15.6.2 Microcode interpreter

Now that we have a plan for writing down the microcode for each instruction, we should also think about how we are going to execute said microcode. The eventual goal, of course, is to create a CPU that is similar to our previous Brainfuck and CHIP-8 processors, using the microcode as its machine code. However, this means we have several, intertwined tasks ahead of us: creating the CPU, designing the details of the microcode, and using said microcode to implement each 8080 instruction. We can simplify this considerably by regarding the microcode as a programming language, and writing a direct software interpreter alongside designing the microcode, instead of jumping straight into hardware design.

Even better, if we write this interpreter polymorphically, we can reuse it in the eventual hardware CPU implementation. We instantiate it with a simple software base monad for rapid microcode development, and then use it as a building block in the hardware implementation which contains extra logic to drive the execution, clock cycle by clock cycle.

Even though we don't yet know what the micro-instructions are going to be, we can start writing the interface of our microcode interpreter. The main operations needed are executing a single micro-instruction, and computing (with potential register-changing side-effects, to implement e.g. `IncrPC`) the address bus value for an input or an output address. We use the `_` context in the type signatures to denote that we don't know yet what exactly the constraint on `m` is going to be:

```
uexec :: _ => MicroInstr -> m ()
inAddr :: _ => InAddr -> m (Either Port Addr)
outAddr :: _ => OutAddr -> m (Either Port Addr)
```

15.7 Micro-architecture & micro-instructions

We have already described the basis of the micro-architecture as one 8-bit value buffer and one 16-bit address buffer. Along the program counter, the stack pointer, and the eight 8080 registers, we also include a flag for interrupt masking. This is purely to implement the `EI` and `DI` instructions: since the whole point of interrupts is to be triggered outside the normal execution of (micro-) instructions, interrupt handling is otherwise not going to be part of the microcode interpreter.

```
data MicroState = MicroState
    { _pc, _sp :: Addr
    , _registers :: Vec 8 Value
    , _allowInterrupts :: Bool
    , _valueBuf :: Value
    , _addrBuf :: Addr
    }
    deriving (Show, Generic, NFDataX)
makeLenses ''MicroState
```

```
mkMicroState :: Addr -> MicroState
mkMicroState pc0 = MicroState{..}
  where
    _pc = pc0
    _sp = 0
    _allowInterrupts = False
    _registers = repeat 0x00
    _valueBuf = 0
    _addrBuf = 0
```

Armed with this definition, we can make a function giving us a `Lens` from `MicroState` to a given `Register`. We special-case loads from `RFlags` to set the three dummy bits to their correct values:

```
reg :: Reg -> Lens' MicroState Value
reg r = registers . lens (fixup . (!! r)) (\s v -> replace r v s)
  where
    fixup = case r of
        RFlags -> (`clearBit` 5) . (`clearBit` 3) . (`setBit` 1)
        _ -> id
```

We also have enough now to implement the two addressing functions `inAddr` and `outAddr`:

```
inAddr :: (MonadState MicroState m) => InAddr -> m (Either Port Addr)
inAddr FromPtr = Right <$> use addrBuf
inAddr FromPort = do
    (port, _) <- twist <$> use addrBuf
    return $ Left port
inAddr IncrPC = Right <$> (use pc <* (pc += 1))
inAddr IncrSP = Right <$> (use sp <* (sp += 1))

outAddr :: (MonadState MicroState m) => OutAddr -> m (Either Port Addr)
outAddr ToPtr = Right <$> use addrBuf
outAddr ToPort = do
    (port, _) <- twist <$> use addrBuf
    return $ Left port
outAddr DecrSP = Right <$> ((sp -= 1) *> use sp)
```

Note that in `FromPort` and `ToPort` we complicate our life by using the low byte of the address buffer instead of the value buffer, which is already eight bits wide. This is because we want to use the value buffer for the value read from / written to the port; it also nicely matches the behavior of `FromPtr` / `ToPtr` regarding the role of the microarchitectural registers.

15.7.1 Hello Micro-World!

Since we have already spelled out the steps of SHLD in prose in the previous section, let's start our design of the MicroInstr datatype by writing out the microcode for SHLD in Clash. Since our design takes five steps, we will put it in a MicroSteps 5 pre post, where MicroSteps is Steps instantiated to our needs, and we will figure out pre and post as we go.

```
type MicroSteps = Steps InAddr MicroInstr OutAddr

shld :: MicroSteps 5 _ _
shld =
    _fetchToAddr   >++>
    _fetchToAddr   >++>
    _writeToPtr RL >++>
    _incrementAddr >++>
    _writeToPtr RH
```

We next decompose each of these five steps into a read/do/write tuple. This also specifies the two ends of shdl as True and True, since the first step reads from memory, and the last step writes to it:

```
shld :: MicroSteps 5 True True
shdl =
    step (IJust IncrPC) ToAddrBuf       INothing      >++>
    step (IJust IncrPC) ToAddrBuf       INothing      >++>
    step INothing       (ToReg RL)      (IJust ToPtr) >++>
    step INothing       (Compute2 Inc2) INothing      >++>
    step INothing       (ToReg RH)      (IJust ToPtr)
```

Since all memory-accessing operations go through the value buffer, here we see our first two micro-instructions: ToAddrBuf copies (or, more precisely, twists) the value buffer's contents into the address buffer, and ToReg which does the same to a general-purpose register.

```
data MicroInstr
    = ToAddrBuf
    | ToReg Reg

uexec ToAddrBuf = twistTo addrBuf =<< use valueBuf
uexec (ToReg r) = reg r <~ use valueBuf
```

To represent the incrementing of the address buffer in step four, it pays to think ahead a bit. There are four instructions involving 16-bit increment/decrement: INX and DCX operates on register pairs, and LHLD and SHLD writes to two neighboring addresses. We can implement all these instructions by loading the right value into the address buffer, using a 16-bit ALU on that value, and writing the result back. We call the corresponding micro-instruction Compute2 to denote that it works on a 2-byte value (in contrast to Compute and ComputeSR which do arithmetic and shift/rotate operations on single bytes, and Compute0 for one-bit (flag) operations). We only have two 16-bit arithmetic functions: increment (for INX, LHLD and SHLD), and decrement (for DCX).

```
data ALU2 = Inc | Dec

data MicroInstr
    = Compute2 ALU2
    | ...

uexec (Compute2 fun) = do
    addrBuf %= case fun of
        Inc -> (+ 1)
        Dec -> subtract 1
```

Keen readers might remember the DAD instruction, which also needs 16-bit arithmetic. However, because DAD behaves more like ADD in terms of handling the carry flag, we will avoid adding a third Add case to ALU2 and instead implement it using the 8-bit ALU, in two rounds.

15.7.2 Handling different MicroSteps types

Using only the micro-instructions added so far, we can already implement several instructions. LHLD is just SHDL with memory traffic going the other way:

```
lhld :: MicroSteps 5 True False
lhld =
    step (IJust IncrPC)  ToAddrBuf      INothing >++>
    step (IJust IncrPC)  ToAddrBuf      INothing >++>
    step (IJust FromPtr) (ToReg RL)     INothing >++>
    step INothing        (Compute2 Inc) INothing >++>
    step (IJust FromPtr) (ToReg RH)     INothing
```

The first two micro-steps of shld and lhld are the same; together, these steps implement fetching a 2-byte immediate argument from the program into the address buffer. We will use this pattern for all instructions with this format, so let’s factor it out:

```
imm2 :: MicroSteps 2 True False
imm2 =
    step (IJust IncrPC) ToAddrBuf INothing >++>
    step (IJust IncrPC) ToAddrBuf INothing

shld :: MicroSteps 5 True True
shdl =
    imm2                                        >++>
    step INothing (ToReg RL)      (IJust ToPtr) >++>
    step INothing (Compute2 Inc2) INothing      >++>
    step INothing (ToReg RH)      (IJust ToPtr)

lhld :: MicroSteps 5 True False
lhld =
    imm2                                          >++>
    step (IJust FromPtr) (ToReg RL)      INothing >++>
    step INothing        (Compute2 Inc2) INothing >++>
    step (IJust FromPtr) (ToReg RH)      INothing
```

In the same vein, `LDA` is similar to `LHLD`, targeting `RA` only:

```
lda :: MicroSteps 3 True False
lda =
    imm2 >++>
    step (IJust FromPtr) (ToReg RA) INothing
```

Note however that the types of `shld`, `lhld` and `lda` are all different from each other: only the first writes to memory in its last step, and the microcode for `lda` is shorter than the others. If our ultimate goal is to write a complete mapping from 8080 instructions to micro-programs, what type are we going to give to that mapping function?

One solution to this problem comes from the fact that unlike `imm2`, the code of `shld` and `lhld` doesn't need to be composed with other code fragments; they are complete. This means we can convert them to the normalized representation micro-program by micro-program, where these type differences have dissolved. To get around the size problem, we also pad them with a micro-NOP up to a fixed size. We just pick 10 for this size for now, hoping that it will be enough for all instructions; once we're done with writing all microcode, we can of course revisit this choice.

```
type MicroLen = 10
type MicroOp = (MicroInstr, Wedge OutAddr InAddr)
type Microcode = (Maybe InAddr, Vec MicroLen MicroOp)
```

```
padded
    :: (KnownNat k, KnownNat n, ((n + k) ~ MicroLen))
    => MicroSteps n pre post
    -> Microcode
padded ops = (first, uops ++ uNOPs)
  where
    (first, uops) = stepsOf ops
    uNOPs = repeat (uNOP, Nowhere)

microcode :: Instr -> Microcode
microcode SHLD = padded shld
microcode LHLD = padded lhld
microcode LDA = padded lda
```

As for `uNOP`, the microcode instruction that doesn't do anything, we could add a bespoke micro-instruction; however, we can use any instruction that has no effect outside the microarchitectural registers. `ToReg` wouldn't be a good choice, since it changes a "real" (i.e. non-microarchitectural) register. Instead, we can pick `ToAddrBuf`: instruction microcode can't make any assumptions about the value of the address buffer before the first micro-instruction, so changing its value at the end of microcode doesn't matter.

```
uNOP :: MicroInstr
uNOP = ToAddrBuf
```

This also gives us an implementation of `NOP`: there's no way to create an empty `Steps` for it, but we can use a single `uNOP` step:

```
microcode NOP = padded $ step INothing uNOP INothing
```

15.7.3 Memory and register transfers

We have already seen how to implement `LDA` by setting the address lines from the address buffer, and transferring the value buffer's value to the `A` register. To go in the other direction for `STA`, we just need the opposite of `ToReg`:

```
data MicroInstr
    = FromReg Reg
    | ...

uexec (FromReg r) = valueBuf <~ use (reg r)
```

```
microcode STA = padded $
    imm2 >++>
    step INothing (FromReg RA) (IJust ToPtr)
```

We can also implement `IN` and `OUT` similar to `LDA` and `STA`; the only difference is that the port number is stored in a single byte.

```
microcode IN = padded $
    step (IJust IncrPC)   ToAddrBuf    INothing >++>
    step (IJust FromPort) (ToReg RA)   INothing
microcode OUT = padded $
    step (IJust IncrPC)   ToAddrBuf    INothing >++>
    step INothing         (FromReg RA) (IJust ToPort)
```

Note that we only twist into the address buffer once; however, this matches the `FromPort` and `ToPort` cases of `inAddr` and `outAddr` which take the 8-bit port number from the correct half of the address buffer.

Beside `LDA` and `STA`, the other generic-purpose memory-transfer instructions are the variants of `MOV` that use memory (via `HL`) as either the source or the destination argument, i.e. the ones that are aliased to `LDAX` and `STAX`. Since these take the memory address from a register pair, we need a 2-byte version of `FromReg` targeting the address buffer; otherwise, their implementation is straightforward.

```
data MicroInstr
    = FromReg2 RegPair
    | ...

microcode (LDAX rr) = padded $
    step INothing        (FromReg2 rr) INothing >++>
    step (IJust FromPtr) (ToReg RA)    INothing
microcode (STAX rr) = padded $
    step INothing (FromReg RA)  INothing      >++>
    step INothing (FromReg2 rr) (IJust ToPtr)
```

We want to implement `FromReg2` similar to `FromReg`, by using a lens over `MicroState`. We have to be a bit careful here; lenses in general are not closed over products[1]. In our case, however, any time a register pair occurs in the result of the instruction decoder, the two components of the pair are always different. This together with the definition of `reg` ensures the orthogonality property required to make `unsafePairL` well-behaved.

[1] See e.g. https://stackoverflow.com/a/36521209/477476 for discussion containing counterexamples.

```
unsafePairL :: Lens' s a -> Lens' s b -> Lens' s (a, b)
unsafePairL l1 l2 = lens
    (view l1 &&& view l2)
    (\s (x,y) -> set l1 x . set l2 y $ s)

regPair :: RegPair -> Lens' MicroState Addr
regPair (Regs r1 r2) =
    unsafePairL (reg r1) (reg r2) . iso bitCoerce bitCoerce
regPair SP = sp

uexec (FromReg2 rr) = addrBuf <~ use (regPair rr)
```

Of course, just to get the value of a `regPair`, we don't really need the full generality of a lens; also, it is always a safe operation. We create a lens instead because other instructions will need a way to transfer from the address buffer to register pairs as well.

A special case of memory access is loading from program code, i.e. immediate loads. Let's implement `MVI`: we read from `IncrPC` and either put the result directly into a register, or write it to `ToPtr` after loading the address buffer from the given register pair:

```
microcode (MVI (Reg r)) = padded $
    step (IJust IncrPC) (ToReg r) INothing
microcode (MVI (Addr rr)) = padded $
    step (IJust IncrPC) (FromReg2 rr) (IJust ToPtr)
```

The third kind of instruction belonging to this group is register-to-register transfers, like `MOV A, B`. The implementation, again, is straightforward:

```
microcode (MOV (Reg r1) (LHS (Reg r2))) = padded $
    step INothing (FromReg r2) INothing >++>
    step INothing (ToReg r1) INothing
```

At this point, we should note that we have made `MOV` generic enough to subsume `LDAX`, `STAX` and `MVI`, but here we have seemingly given up on this generality by writing their microcode separately. Instead, let's write a version of `MOV`'s microcode that covers all combinations of destination and source arguments:

```
microcode (MOV (Reg r) src) = case src of
    Imm -> padded $
        step (IJust IncrPC) (ToReg r) INothing
    LHS (Reg r') -> padded $
        step INothing (FromReg r') INothing >++>
        step INothing (ToReg r)    INothing
    LHS (Addr rr) -> padded $
        step INothing        (FromReg2 rr) INothing >++>
        step (IJust FromPtr) (ToReg r)     INothing

microcode (MOV (Addr rr) src) = case src of
    Imm -> padded $
        step (IJust IncrPC) (FromReg2 rr) (IJust ToPtr)
    LHS (Reg r) -> padded $
        step INothing (FromReg r)   INothing      >++>
        step INothing (FromReg2 rr) (IJust ToPtr)
    LHS (Addr rr') -> padded $
        step INothing        (FromReg2 rr') INothing      >++>
        step (IJust FromPtr) (FromReg2 rr)  (IJust ToPtr)
```

Some of the immediate load and register transfer instructions target register pairs instead of single registers: these are `LXI` (the register pair version of `MVI`),`PCHL`, `SPHL`, and `XCHG`. In fact, we can regard `JMP` as an immediate load into `PC`.

We can use the 16-bit address buffer for these operations by adding the `ToReg2` pair of `FromReg2`, and `Jump` for the special case

```
data MicroInstr
    = ToReg2 RegPair
    | Jump
    | ...

uexec (ToReg2 rr) = regPair rr <~ use addrBuf
uexec Jump = pc <~ use addrBuf
```

With these additions, the microcode for the register pair instructions is straightforward:

```
microcode (LXI rr) = padded $
    imm2 >++>
    step INothing (ToReg2 rr)    INothing
microcode JMP =
    imm2 >++>
    step INothing Jump INothing
microcode PCHL = padded $
    step INothing (FromReg2 RHL) INothing >++>
    step INothing Jump           INothing
microcode SPHL = padded $
    step INothing (FromReg2 RHL) INothing >++>
    step INothing (ToReg2 SP)    INothing
```

However, we get into a pickle once we get to XCHG. Recall that XCHG is supposed to *swap* the values of HL and DE, but we don't want to have any direct connection between these four registers: everything should go via the single address buffer. We solve this by using a swapping version of FromReg2 / ToReg2:

```
data MicroInstr
    = SwapReg2 RegPair
    | ...

uexec (SwapReg2 rr) = swap addrBuf (regPair rr)

swap :: (MonadState s m) => Lens' s a -> Lens' s a -> m ()
swap lx ly = do
    x <- use lx
    y <- use ly
    lx .= y
    ly .= x

microcode XCHG = padded $
    step INothing (FromReg2 RHL) INothing >++>
    step INothing (SwapReg2 RDE) INothing >++>
    step INothing (SwapReg2 RHL) INothing
```

In fact, we don't even need ToReg2 if we have SwapReg2: the only difference between the two micro-instructions is the value of the address buffer after execution, and the microcode of different instructions isn't allowed to make any assumptions about the values of the micro-architectural registers, so there is no possible interference. For this reason, we remove ToReg2 from MicroInstr and simply replace its occurrences with SwapReg2 in LXI, PCHL and SPHL.

15.7.4 Arithmetic

Because we have already added Compute2 for LHLD and SHLD, the 16-bit arithmetic instructions INX and DCX are a breeze to implement: they both just load the given register pair into the address buffer, apply the given two-byte function, and swap back the result:

```
microcode (DCX rr) = padded $
    step INothing (FromReg2 rr)  INothing >++>
    step INothing (Compute2 Dec) INothing >++>
    step INothing (SwapReg2 rr)  INothing
```

```
microcode (INX rr) = padded $
    step INothing (FromReg2 rr)  INothing >++>
    step INothing (Compute2 Inc) INothing >++>
    step INothing (SwapReg2 rr)  INothing
```

The corresponding 8-bit arithmetic operations are `INR` and `DCR`. We could implement these with a `Compute2`-like dedicated instruction for 8-bit incrementing/decrementing; instead, we fold them into the general 8-bit ALU that powers `ADD` and `XOR` and all the others. This makes sense not just because it is more economical on the number of kinds of micro-instructions, but also because all the general arithmetic instructions, and `INR` and `DCR` as well, change the various status flags in the same way. This means that the only difference between `ADD B` and `INR B` is that the former uses the `A` register as the first operand, while the latter uses the constant 1. We can always use the value buffer as the second operand, by loading the right register value beforehand.

Actually, if we re-read the description of `INR` and compare it to `ADD`, it turns out there is another difference: `INR` doesn't update the `C` flag, but `ADD` does. Similarly, we can think of `CMA` as subtraction from the constant 255, without updating any of the flags. We could add five parameters to `Compute` to set which status flags should be updated; however, looking through all the valid instructions, we can see that `Z`, `S` and `P` are always either all updated or neither of them are.

```
data MicroInstr
    = Compute ALUArg ALU UpdateZSP UpdateAC UpdateC
    | ...

data ALUArg
    = RegA
    | Const01
    | ConstFF

data UpdateZSP = SetZSP | KeepZSP deriving (Eq)
data UpdateAC = SetAC | KeepAC deriving (Eq)
data UpdateC = SetC | KeepC deriving (Eq)
```

Before implementing this new micro-instruction, let's add the microcode for all the arithmetic instructions that we can now write with `Compute`. The main ones, of course, are `ALU` and `CMP`, which have the same structure: get the right-hand side argument into the value buffer, apply `Compute`, then use the result by optionally writing it back into register `A`:

```
alu
    :: ALU
    -> RHS
    -> MicroInstr
    -> Microcode
alu fun rhs writeback = case rhs of
    Imm -> padded $
        step (IJust IncrPC) (Compute RegA fun SetZSP SetAC SetC)
                                                        INothing >++>
        step INothing       writeback                   INothing
    LHS (Reg r) -> padded $
        step INothing (FromReg r)                       INothing >++>
        step INothing (Compute RegA fun SetZSP SetAC SetC)
                                                        INothing >++>
        step INothing writeback                         INothing
    LHS (Addr rr) -> padded $
        step INothing        (FromReg2 rr)              INothing >++>
        step (IJust FromPtr) (Compute RegA fun SetZSP SetAC SetC)
                                                        INothing >++>
        step INothing        writeback                  INothing
```

We can think of CMP as a special version of SUB with a no-op writeback:

```
microcode (ALU fun src) = alu fun src (ToReg RA)
microcode (CMP src) = alu (Sub False) src uNOP
```

The other arithmetic functions can be written in a more ad-hoc way, but still centered on Compute to implement the actual arithmetics:

```
microcode (INR (Addr rr)) = padded $
    step INothing        (FromReg2 rr)                  INothing >++>
    step (IJust FromPtr) (Compute Const01 (Add False) SetZSP SetAC KeepC)
                                                        (IJust ToPtr)
microcode (INR (Reg r)) = padded $
    step INothing (FromReg r)                           INothing >++>
    step INothing (Compute Const01 (Add False) SetZSP SetAC KeepC)
                                                        INothing >++>
    step INothing (ToReg r)                             INothing

microcode (DCR (Addr rr)) = padded $
    step INothing        (FromReg2 rr)                  INothing >++>
    step (IJust FromPtr) (Compute ConstFF (Add False) SetZSP SetAC KeepC)
                                                        (IJust ToPtr)
```

```
microcode (DCR (Reg r)) = padded $
    step INothing (FromReg r)                           INothing >++>
    step INothing (Compute ConstFF (Add False) SetZSP SetAC KeepC)
                                                        INothing >++>
    step INothing (ToReg r)                             INothing

microcode CMA = padded $
    step INothing (FromReg RA)                          INothing >++>
    step INothing (Compute ConstFF (Sub False) KeepZSP KeepAC KeepC)
                                                        INothing >++>
    step INothing (ToReg RA)                            INothing
```

Quite good mileage from just one new micro-instructions, especially if we consider that with our generalizations, the single `ALU` opcode stands for all of `ADD`, `ADC`, `SUB`, `SBC`, `AND`, `ORA`, `XOR`, and `DAA`!

Its handler in `uexec` is quite straightforward: taking a page from the CHIP-8 CPU, we leave all the heavy lifting of `Compute` to the pure function `binALU`. This leaves `Compute` with only the task of providing the right first argument, and updating the flags.

```
uexec (Compute arg fun updateAC updateC) = do
    x <- case arg of
        RegA -> use (reg RA)
        Const01 -> pure 0x01
        ConstFF -> pure 0xff
    ac <- use (flag FAC)
    c <- use (flag FC)
    y <- use valueBuf
    let (ac', c', y') = binALU fun x (ac, c, y)
    when (updateZSP == SetZSP) $ do
       flag FZ .= (y' == 0)
       flag FS .= y' `testBit` 7
       flag FP .= even (popCount y')
    when (updateAC == SetAC) $ flag FAC .= ac'
    when (updateC == SetC) $ flag FC .= c'
    valueBuf .= y'
```

We promised an implementation of `DAD` using the 8-bit ALU; it is also an example of using `Compute` without `SetZSP`.

Recall that `DAD` adds the given register pair's value (as a 16-bit unsigned integer) to `HL`. So far, we can only add `A` and the constant values 1 and 255 to the value buffer, so let's extend `ALUArg` with a new constructor that takes the low byte of the address buffer as the other operand:

```
data ALUArg
    = AddrLo
    | ...

uexec (Compute arg fun updateZSP updateAC updateC) = do
    x <- case arg of
        RegA -> use (reg RA)
        AddrLo -> truncateB <$> use addrBuf
        Const01 -> pure 0x01
        ConstFF -> pure 0xff
    -- Rest unchaged
```

We can then do 16-bit addition simply as two 8-bit additions, with the carry propagated via the `C` flag. Since the specification of `DAD` says that `C` should be updated, it is not a problem to overwrite its value during the first addition, since the second addition will set it to its correct value anyway. The `ToAddrBuf` instructions after each addition `twist` the current byte (first the low one, then the high one) back into the address buffer.

```
microcode (DAD rr) = padded $
    step INothing (FromReg2 rr)                     INothing >++>
    step INothing (FromReg RL)                      INothing >++>
    step INothing (Compute AddrLo (Add False) KeepZSP KeepAC SetC)
                                                    INothing >++>
    step INothing ToAddrBuf                         INothing >++>
    step INothing (FromReg RH)                      INothing >++>
    step INothing (Compute AddrLo (Add True) KeepZSP KeepAC SetC)
                                                    INothing >++>
    step INothing ToAddrBuf                         INothing >++>
    step INothing (SwapReg2 RHL)                    INothing
```

For shifts and rotations, we follow the same pattern: the `ComputeSR` micro-instruction is implemented in terms of a pure `shiftRotateALU` function.

```
data MicroInstr
    = ComputeSR (Either ShiftRotate ShiftRotate)
    | ...

microcode (SHROT sr) = padded $
    step INothing (FromReg RA)   INothing >++>
    step INothing (ComputeSR sr) INothing >++>
    step INothing (ToReg RA)     INothing
```

```
uexec (ComputeSR sr) = do
    c <- use (flag FC)
    x <- use $ reg RA
    let (c', x') = shiftRotateALU sr (c, x)
    flag FC .= c'
    valueBuf .= x'
```

The actual implementation of binALU and shiftRotateALU is straightforward; for Add and Sub, computing the AC flag requires going 4 bits at a time. Similarly, ensuring a correct (two-digit) binary-coded decimal value requires fixing both nybbles separately.

```
binALU :: ALU -> Value -> (Bool, Bool, Value) -> (Bool, Bool, Value)
binALU fun x (a, c, y) = case fun of
    Add c0 -> addC (c0 && c)
    Sub c0 -> subC (c0 && c)
    And    -> (testBit (x .|. y) 4, False, x .&. y)
    Or     -> (False, False, x .|. y)
    XOr    -> (False, False, x `xor` y)
    BCD    -> bcd
  where
    (xh, xl) = nybbles x
    (yh, yl) = nybbles y

    addC c = (a', c', byte (zh, zl))
      where
        (a', zl) = bitCoerce $ (xl `add` yl) + if c then 1 else 0
        (c', zh) = bitCoerce $ (xh `add` yh) + if a' then 1 else 0

    subC c = (a', c', byte (zh, zl))
      where
        (a', zl) = bitCoerce $ (xl `sub` yl) - if c then 1 else 0
        (c', zh) = bitCoerce $ (xh `sub` yh) - if a' then 1 else 0

    bcd = (a', c', z)
      where
        (a', w) = bitCoerce $ add y $
            if yl > 9 || a then (0x06 :: Value) else 0
        (wh, _) = nybbles w
        (c', z) = bitCoerce $ add w $
            if wh > 9 || c then (0x60 :: Value) else 0
```

```
nybbles :: Value -> (Unsigned 4, Unsigned 4)
nybbles = bitCoerce

byte :: (Unsigned 4, Unsigned 4) -> Value
byte = bitCoerce

shiftRotateALU
    :: Either ShiftRotate ShiftRotate
    -> (Bool, Value)
    -> (Bool, Value)
shiftRotateALU fun (c, x) = case fun of
    Left sr ->
        (b7, bitCoerce (mid, b0, case sr of Shift -> c; Rotate -> b7))
    Right sr ->
        (b0, bitCoerce (case sr of Shift -> c; Rotate -> b0, b7, mid))
  where
    (b7, mid, b0) = bitCoerce x :: (Bool, Unsigned 6, Bool)
```

15.7.5 Conditional execution

The conditional instructions `JMPIf`, `CALLIf` and `RETIf` need part of the microcode to only be executed if certain status flags have certain values. Recall the microcode for an unconditional `JMP`:

```
microcode JMP = padded $
    imm2 >++>
    step INothing Jump INothing
```

To change it to a conditional jump, we add a check after the `imm2` fetch but before the `Jump`. And what do we need to be able to check for? We need to support all conditions of type `Cond`, which is what we have in `JMPIf`, `CALLIf` and `RETIf`:

```
data MicroInstr
    = When Cond | ...

evalCond :: (MonadState MicroState m) => Cond -> m Bool
evalCond (Cond flg target) = uses (flag flg) (== target)
```

We create the lens for a given `flag` from the `RFlags` register, by taking its `flg`-th `Bit`, and converting it from/to `Bool`

```
bitL :: (BitPack a, Enum i) => i -> Lens' a Bit
bitL i = lens (!i) (flip $ replaceBit i)
```

```
flag :: Flag -> Lens' MicroState Bool
flag flg = reg RFlags . bitL flg . iso bitToBool boolToBit
```

The microcode for `JMPIf`, then, loads the target program counter into the address buffer, and then bails out if the `condition` doesn't hold:

```
microcode (JMPIf cond) = padded $
    imm2 >++>
    step INothing (When cond) INothing >++>
    step INothing Jump        INothing
```

We will handle `When` micro-instructions by wrapping our underlying monad `m` in `ExceptT FlowControl`. The name `ExceptT` is unfortunate, but its exception-like semantics are a good match for our use case: if, during microcode execution, the interpreter encounters a request to go to the next instruction or the halt state, it will need to terminate the current instruction.

```
data FlowControl = GotoNext | GotoHalt

uexec (When cond) = do
    passed <- evalCond cond
    unless passed $ throwError GotoNext
```

Similarly, for `HALT`, we just need a micro-instruction that throws `GotoHalt`:

```
data MicroInstr
    = Halt | ...

uexec Halt = throwError GotoHalt

microcode HLT = padded $ step INothing Halt INothing
```

15.7.6 Stack operations

For `CALL` and `RET`, we need a way of pushing and popping the program counter. We can pop any two-byte value by reading from `IncrSP` into the value buffer twice and `twisting` it into the address buffer, just like how `imm2` reads from `IncrPC`:

```
pop2 :: MicroSteps 2 True False
pop2 =
    step (IJust IncrSP) ToAddrBuf INothing >++>
    step (IJust IncrSP) ToAddrBuf INothing
```

This gives us a straightforward implementation of `RET` and `RETIf`:

```
popPC = pop2 >++> step INothing Jump INothing

microcode RET = padded $
    popPC
microcode (RETIf cond) = padded $
    step INothing (When cond) INothing >++>
    popPC
```

Can we do the same for `pushPC`, by first moving the value of the program counter into the address buffer by `push2`, and then `twisting` it into the value buffer as we write to `DecrSP`? The problem with this approach becomes apparent when we write out the full the microcode for `CALL`:

```
push2 =
    step INothing FromAddrBuf (IJust DecrSP) >++>
    step INothing FromAddrBuf (IJust DecrSP)

pushPC = step INothing FromPCToAddrBuf INothing >++> push2

microcode CALL = padded $
    imm2   >++>
    pushPC >++>
    step INothing Jump INothing
microcode (CALLIf cond) = padded $
    imm2                                >++>
    step INothing (When cond) INothing >++>
    pushPC                              >++>
    step INothing Jump               INothing
```

Here, `imm2` fetches the two-byte call target address into the address buffer, which we use at the end when `Jump` copies it into the program counter. We can't swap the order of `imm2` and `pushPC`, because the program counter value to push needs to point to the start of the next instruction, i.e. *after* the two-byte immediate argument has been fetched. So `pushPC` cannot be allowed to overwrite the address buffer. Instead, we add a `FromPC` micro-instruction which uses the value buffer directly, thereby keeping the address buffer unchanged:

```
data MicroInstr
    = FromPC | ...

uexec FromPC = valueBuf <~ twistFrom pc
```

```
pushPC :: MicroSteps 2 False True
pushPC =
    step INothing FromPC (IJust DecrSP) >++>
    step INothing FromPC (IJust DecrSP)
```

RST is just half a CALL: instead of fetching the call target from the program, the target address is encoded in the instruction itself. The Rst micro-instruction decodes the reset index into a full 16-bit address via a left-shift.

```
data MicroInstr
    = Rst (Unsigned 3) | ...

uexec (Rst rst) = pc .= extend rst `shiftL` 3

microcode (RST irq) = padded $
    pushPC >++>
    step INothing (Rst irq) INothing
```

Accessing the stack as data isn't different from accessing it for control: the only difference between PUSH and CALL, and RET and POP, is where the value comes from/goes to:

```
microcode (PUSH rr) = padded $
    step INothing (FromReg2 rr) INothing >++>
    push2
microcode (POP rr) = padded $
    pop2 >++>
    step INothing (SwapReg2 rr) INothing
```

And then we have XTHL, which swaps the top of the stack with the HL register pair. Because we already have SwapReg2, we can write its microcode to match this explanation exactly: pop, swap, and push back.

```
microcode XTHL = padded $
    pop2                                  >++>
    step INothing (SwapReg2 RHL) INothing >++>
    push2
```

15.7.7 Internal settings

This final catch-all category covers CMC, STC, and EI/DI: all these are instructions with no connection to the outside world, just changing various status bits. Our life here would be much simpler if the interrupt masking bit was part of the flag register, but alas, it isn't. So we need *two* bespoke micro-instructions to transform individual flags and the interrupt enable bit.

```
data MicroInstr
    = Compute0 Flag ALU0
    | SetInt Bool
    | ...

data ALU0
    = Complement
    | ConstTrue

microcode CMC = padded $ step INothing (Compute0 FC Complement) INothing
microcode STC = padded $ step INothing (Compute0 FC ConstTrue)  INothing
microcode (INT b) = padded $ step INothing (SetInt b) INothing

uexec (Compute0 flg fun) = flag flg %= case fun of
    ConstTrue -> const True
    Complement -> complement
uexec (SetInt b) = allowInterrupts .= b
```

As this concludes the microcode for all instructions, we can also retroactively change `MicroLen` to just 8, the length of the longest microcode from `DAD`.

15.8 A direct software implementation

Now that we have a complete mapping of Intel 8080 instructions to microcode, and a complete implementation of all micro-instructions in terms of some underlying monad `m`, we turn our attention to providing such an `m` for a software implementation.

First, recall the full type of `uexec`, including the constraint imposed on `m`:

```
uexec
    :: (MonadState MicroState m)
    => MicroInstr
    -> ExceptT FlowControl m ()
```

The natural choice of `m` would be `State MicroState`, but that leaves no room to implement memory and port access; instead, we create a record type to describe the model of the outside world, and carry that around.

```
data World m = World
    { readMem :: Addr -> m Value
    , writeMem :: Addr -> Value -> m ()
    , inPort :: Port -> m Value
    , outPort :: Port -> Value -> m Value
    }

newtype SoftCPU m a = SoftCPU
    { unSoftCPU :: ReaderT (World m) (StateT MicroState m) a }
    deriving newtype
      (Functor, Applicative, Monad,
       MonadReader (World m), MonadState MicroState)

instance MonadTrans SoftCPU where
    lift = SoftCPU . lift . lift
```

The big simplification compared to the eventual hardware implementation is that access to the outside world (via ports and memory addresses) is executed *directly*, inline with the execution of the microcode. We write a function `nextInstr` that runs the full microcode for the next instruction, and `runSoftCPU` which runs all the way until the CPU encounters a `HLT` instruction:

```
nextInstr :: (Monad m) => SoftCPU m Bool
nextInstr = do
    instr <- decodeInstr <$> fetchByte
    exec instr

runSoftCPU :: (Monad m) => World m -> MicroState -> m ()
runSoftCPU w s0 =
    (evalStateT `flip` s0) . (runReaderT `flip` w) .
    unSoftCPU $
    whileM nextInstr
```

We can fetch the next instruction via the `readMem` operation provided by the `World`:

```
fetchByte :: (Monad m) => SoftCPU m Value
fetchByte = do
    addr <- use pc <* (pc += 1)
    peekByte addr
peekByte :: (Monad m) => Addr -> SoftCPU m Value
peekByte addr = do
    readMem <- asks readMem
    lift $ readMem addr
```

Of course, `peekByte` is useful for more than just fetching instructions: we will also use it to implement the `addressing` request of each micro-step, if it is a memory read. In other cases, we proceed similarly, just using different fields of `World`.

```
addressing :: (Monad m) => Wedge OutAddr InAddr -> SoftCPU m ()
addressing = bitraverse_ (doWrite <=< outAddr) (doRead <=< inAddr)

doWrite :: (Monad m) => Either Port Addr -> SoftCPU m ()
doWrite target = either writePort poke target =<< use valueBuf
  where
    writePort port value = do
        write <- asks outPort
        lift $ void $ write port value

    poke addr value = do
        writeMem <- asks writeMem
        lift $ writeMem addr value

doRead :: (Monad m) => Either Port Addr -> SoftCPU m ()
doRead target = valueBuf <~ either readPort peekByte target
  where
    readPort port = do
        read <- asks inPort
        lift $ read port
```

Finally we have everything to write the instruction executor. Given a decoded instruction, we look up its microcode, execute its first addressing operation (which is always a read, so we lift it to a read-or-write via `wedgeRight`), then run each micro-step in turn. The `Bool` returned by `exec` is whether the CPU should keep running, or it executed a halt instruction; this latter signals to `runSoftCPU` to finish its execution loop.

```
exec :: (Monad m) => Instr -> SoftCPU m Bool
exec instr = do
    let (setup, usteps) = microcode instr
    addressing $ wedgeRight setup
    ex <- runExceptT $ mapM_ ustep usteps
    return $ ex /= Left GotoHalt
```

```
ustep :: (Monad m) => MicroOp -> ExceptT FlowControl (SoftCPU m) ()
ustep (effect, post) = do
    uexec effect
    lift $ addressing post
```

15.8.1 Zero-cost padding

Our naïve implementation of padding means that no matter how many useful steps it takes to implement a given instruction, we always take the full `MicroLen` cycles. The only exception to this is the `When` micro-instruction, which can cause an early exit via `GotoNext`. But isn't an early exit what we want at the end of the useful part of the microcode as well?

If we mark each `MicroOp` with a flag telling if we should continue afterwards, we can use that to exit `exec` without going through all the remaining `uNOP` steps:

```
type MicroOp = ((MicroInstr, Wedge OutAddr InAddr), Bool)

padded
    :: (KnownNat k, KnownNat n, ((n + 1) + k) ~ MicroLen)
    => MicroSteps (n + 1) pre post
    -> Microcode
padded ops = (first, withCont uops ++ uNOPs)
  where
    (first, uops) = stepsOf ops
    uNOPs = repeat ((nop, Nowhere), False)

withCont :: (KnownNat n) => Vec (n + 1) a -> Vec (n + 1) (a, Bool)
withCont xs = zip xs $ repeat True :< False
```

The change in `padded`'s type reflects that we can't mark the end of an empty `MicroSteps`; however, recall that we already have no way of constructing a value of type `MicroSteps 0 pre post`, since the smallest constructor corresponds to a singleton vector. The new type of `padded` (coming from the type of `withCont`) simply expresses that this property of `MicroSteps` is exploited here.

The inspiration for this improvement came from the handling of a `When` where the condition doesn't hold; it only makes sense that we can handle the last micro-instruction in `ustep` by signaling the same `GotoNext` exception:

```
ustep :: (Monad m) => MicroOp -> ExceptT FlowControl (SoftCPU m) ()
ustep ((effect, post), cont) = do
    uexec effect
    lift $ addressing post
    unless cont $ throwError GotoNext
```

15.8.2 Interrupts

Putting together the instruction decoder `decodeInstr`, the microcode table `microcode`, the addressing interpreters `inAddr` and `outAddr`, the microcode interpreter `uexec`, and `SoftCPU` as the main controller, we have everything for straight-line execution of Intel 8080 programs: at the start of `runSoftCPU`, the first instruction is fetched from the initial value of the program counter, and subsequent instructions are read and executed, interleaved with memory and port access, one after the other.

However, the Intel 8080 also provides a way of *interrupting* this regular sequence of instruction execution. As we have seen, the interrupt request pin can be used to instructs the CPU to fetch the first byte of the next instruction without using the program counter, instead relying on some external mechanism to put the right value on the data bus. For the software implementation, we keep things simple by putting the instruction itself in the interrupt request:

```
interrupt :: (Monad m) => Instr -> SoftCPU m ()
interrupt instr = whenM (use allowInterrupts) $ do
    allowInterrupts .= False
    exec instr
```

We can combine regular `nextInstr` for normal operation and `interrupt` for external triggers by running the former in a loop, and calling the latter whenever we want to simulate an event. We will see an example of this in the software implementation of the Space Invaders arcade machine, where the video subsystem raises interrupts on certain raster lines.

15.8.3 Testing

The big benefit of having a software model of the CPU is that it allows high-level testing of the components that are going to be shared with the hardware implementation. Concretely, we can use the software CPU as a test bench for the microcode. We started with the specification of the various 8080 instructions and turned them into microcode; before thinking too much about how to implement that microcode in hardware, it would be nice to have some assurance that we're targeting *correct* microcode.

There are several test suites one can find for the 8080, most of them contemporary to the CPU itself. Here, we will look at two tests in particular: *Preliminary Z80 tests*, which was released as open source in 1994 by Frank D. Cringle and modified to target the Intel 8080 by Ian Bartholomew; and the *Microcosm Associates 8080/8085 CPU Diagnostic* from 1980 by Kelly Smith and containing updates by Mike Douglas, *donated to the SIG/M CP/M users' group* according do its header.

The interface to both of these tests is the same: they are meant to run on the CP/M operating system, print some success / failure messages, and exit. Does that mean we need to build a full computer that can run CP/M just to be able to test our CPU? Fortunately not: we can implement the OS routine (at address `0x0005`) that originally prints to the screen by writing to an output port. Similarly, we can use a single `HLT` instruction to "exit" the current "process" at `0x0000`. This gives us a simple way to observe the test results by just listening to writes to the output port and the CPU's halt status.

The only other CP/M convention we need to be aware of is that user programs are loaded to and run from address 0x0100 onwards. So given a binary image file containing one of the two test suites, we can load it into an array representing the 8080 memory by prepending our "mini CP/M" prelude:

```
load :: FilePath -> IO (IOArray Addr Value)
load file = do
    bs <- fmap bitCoerce . BS.unpack <$> BS.readFile file
    arr <- newArray (minBound, maxBound) 0x00
    zipWithM_ (writeArray arr) [0x0000..] prelude
    zipWithM_ (writeArray arr) [0x0100..] bs
    return arr
```

Then we can run it by implementing `readMem` and `writeMem` using the array, and printing in `outPort`. The initial CPU state sets the program counter to 0x0100, which is where the tests start.

```
run :: IOArray Addr Value -> IO ()
run arr = runSoftCPU (mkWorld arr) (mkState 0x0100)

mkWorld :: IOArray Addr Value -> World IO
mkWorld arr = World{..}
  where
    readMem addr = readArray arr addr
    writeMem addr val = writeArray arr addr val
    inPort _ = return 0xff
    outPort _port value = do
        putChar . chr . fromIntegral $ value
        return 0xff
```

Note that `outPort` doesn't actually check the port number – our `prelude` only uses a single port (port 0), so we just connect the same functionality to all of them.

Here, we write `main` to simply run each test suite in turn, and it is up to the user to verify their output. Integrating into a testing framework is beyond our scope, and is thus left as an exercise for the reader; in the code accompanying this book, the Intel 8080 implementation uses the *Tasty* library for this purpose.

```
main :: IO ()
main = do
    hSetBuffering stdout NoBuffering

    for_ files $ \file -> do
        putStrLn $ "Running test " <> takeBaseName file <> ":"
        run =<< load file
        putStrLn ""
  where
    files = ["TST8080.COM", "8080PRE.COM"]
```

The only missing part is writing the `prelude`; finally we get to program our own CPU! We need two entry points for these tests: `0x0000` finishes the test, and `0x0005` prints a message. For the first one, we implement `0x0000` as printing a courtesy newline before stopping the CPU:

```
prelude :: [Value]
prelude = mconcat [exit, message]
  where
    exit =
      [ [ 0x3e, 0x0a ]        -- 0x0000: exit:    MVI A, 0x0a
      , [ 0xd3, 0x00 ]        -- 0x0002:          OUT 0
      , [ 0x76 ]              -- 0x0004:          HLT
      ]
```

Printing a message turns out to be a bit trickier, because CP/M actually maps several different functionalities to entry address `0x0005`, dispatched on the `C` register's value. If `C` is equal to 2, the single ASCII character in the `E` register is printed; if `C` is 9, a sequence of characters starting at `DE` is printed, terminated by a `'$'` character. Because these are the only two modes used in our test suites, instead of multi-way switching, we simply check `C` against 2, and then do one or the other.

```
    message =
      [ [ 0x3e, 0x02 ]        -- 0x0005: message: MVI A, 0x02
      , [ 0xb9 ]              -- 0x0007:          CMP C
      , [ 0xc2, 0x0f, 0x00 ]  -- 0x0008:          JNZ 0x000f
      , [ 0x7b ]              -- 0x000B: putChr:  MOV A, E
      , [ 0xd3, 0x00 ]        -- 0x000C:          OUT 0
      , [ 0xc9 ]              -- 0x000E:          RET
      , [ 0x0e, 0x24 ]        -- 0x000F: putStr:  MVI C, '$'
      , [ 0x1a ]              -- 0x0011: loop:    LDAX DE
      , [ 0xb9 ]              -- 0x0012:          CMP C
      , [ 0xc2, 0x17, 0x00 ]  -- 0x0013:          JNZ next
      , [ 0xc9 ]              -- 0x0016:          RET
      , [ 0xd3, 0x00 ]        -- 0x0017: next:    OUT 0
      , [ 0x13 ]              -- 0x0019:          INX DE
      , [ 0xc3, 0x11, 0x00 ]  -- 0x001a:          JMP loop
```

If everything works fine, we should see the following output:

```
Running test TST8080:
MICROCOSM ASSOCIATES 8080/8085 CPU DIAGNOSTIC
 VERSION 1.0  (C) 1980

 CPU IS OPERATIONAL

Running test 8080PRE:
8080 Preliminary tests complete
```

Just to see what a positive result look like, let's break the microcode a bit. Let's say we forget to write the result back to the register pair in DCX:

```
microcode (DCX rr) = padded $
    step INothing (FromReg2 rr)  INothing >++>
    step INothing (Compute2 Dec) INothing
```

This is caught by the Microcosm test:

```
Running test TST8080:
MICROCOSM ASSOCIATES 8080/8085 CPU DIAGNOSTIC
 VERSION 1.0  (C) 1980

 CPU HAS FAILED!    ERROR EXIT=04EE
```

For this test, the exit location is always the address of the instruction just *after* the error; looking at 0x04ee in the assembly source of TST8080, we see the following code testing the DCX instruction:

```
04E5   0B          DCX B
04E6   1B          DCX D
04E7   2B          DCX H
04E8   3E 12       MVI A,012H
04EA   B8          CMP B
04EB   C4 A006     CNZ CPUER   ;TEST "DCX" B
```

The so-called "preliminary" test doesn't catch this error; for another example, we can try breaking `PCHL` by omitting the step that does the actual jump:

```
Running test 8080PRE:
02f7
```

The output is less elaborate than the one from the Microcosm test, but it does lead us to the assembly code just before `0x2f7`:

```
                   ; test indirect jumps
02F0   21 02F7     LXI H,lab7
02F3   E9          PCHL
02F4   CD 0352     CALL error
```

15.9 The complete CPU

We structure the hardware implementation of the CPU the same way as our earlier processors: we create a datatype to control the various execution phases and use that in the state.

```
data Phase
    = Init
    | Halted
    -- More to be added
    deriving (Show, Generic, NFDataX)

data CPUState = CPUState
    { _phase :: Phase
    , _microState :: MicroState
    }
    deriving (Show, Generic, NFDataX)
makeLenses ''CPUState

initState :: Addr -> CPUState
initState pc0 = CPUState
    { _phase = Init
    , _microState = mkMicroState pc0
    }

type CPU = CPUM CPUState CPUOut
```

We implement the default output values by wiring the address buffer to the address line, and checking the phase for the `halted` output.

```
defaultOut :: CPUState -> Pure CPUOut
defaultOut CPUState{_microState = MicroState{..}, ..} = CPUOut{..}
  where
    _addrOut = Right _addrBuf
    _dataOut = Nothing
    _interruptAck = False
    _halted = case _phase of
        Halted -> True
        _ -> False
```

For the implementation of the actual CPU state transition function, let's ignore interrupts first. We extend the `Phase` datatype with two more constructors: we enter `Fetching` with the next instruction's machine code available on the `dataIn` input, so we can use that as an index into a ROM containing the microcode; we then go to the `Executing` phase, which goes through the microcode step by step.

```
data Phase
    = Fetching
    | Executing Value (Index MicroLen)
    | ...
```

As a first approximation, the four constructors of `Phase` can be handled as such:

- In `Init`, we start fetching the next instruction, by setting the address bus to the program counter's value and going to the `Fetching` phase.
- In `Fetching`, the instruction machine code is read from the data bus, the program counter is incremented, and we go to the `Executing` phase for the first micro-step.
- In `Executing instr i`, the *i*-th micro-step of the given instruction is executed.
- In the `Halted` phase, we simply remain in that phase without any other state change.

```
cpu :: Pure CPUIn -> CPUM CPUState CPUOut ()
cpu CPUIn{..} = use phase >>= \case
    Init -> fetchNext
    Fetching -> do
        let instr = dataIn
        microState.pc += 1
        phase .= Executing instr 0
    Executing instr i -> exec instr i
    Halted -> return ()
```

```
fetchNext :: CPU ()
fetchNext = do
    addr <- use (microState.pc)
    addrOut .:= Right addr
    phase .= Fetching
```

For the actual execution, the microcode is retrieved from ROM and executed using the `uexec` function we have already implemented. Then, depending on the result, either we go to the next step (`Executing instr (i + 1)`), to the next instruction via `fetchNext`, or the `Halted` phase.

```
exec :: Value -> Index MicroLen -> CPU ()
exec instr i = do
    let ((uop, post), cont) = microcodeFor instr !! i
    runExceptT (zoom microState $ uexec uop) >>= \case
        Left GotoNext -> do
            fetchNext
        Left GotoHalt -> do
            phase .= Halted
        Right () -> do
            maybe fetchNext (assign phase . Executing instr) $ succIdx i
```

The implementation of `microcodeFor` uses Template Haskell to evaluate `microcode` at all possible 8-bit machine code bytes. The Template Haskell excursion ensures that Clash puts only the flattened bit-level data of the microcode in the generated Verilog, instead of complicated expressions to compute it that can trip up some downstream synthesis tools.

```
microcodeFor :: Value -> MicroOps
microcodeFor = asyncRom
    $(TH.lift $ map (snd . microcode . decodeInstr . bitCoerce) $
    indicesI @256)
```

While the above is enough scaffolding for the execution of the micro-instructions, it doesn't solve the problem of I/O: note that both the initial read address and the `after`-instruction read/write address from the microcode is ignored. So what does it look like to handle these requests? Writes are easy: we can simply set the address line output to the `outAddr`-computed value, and put `valueBuf` on `dataOut`. However, reads are more involved: after setting the address bus, the result comes back from RAM at the next cycle. So for example, in the `Fetching` phase, we have no way of handling the full extent of an initial read request; the most we can do is set the address bus, and then add a flag for the `Executing` phase to store the `dataIn` input in the value buffer:

```
data Phase
    = Executing Value (Index MicroLen) | ...
```

We write the `addressing` implementation to return whether it is a read request, so that both the `Fetching` and `Executing` phases know what to pass to the next `Executing` phase:

```
addressing :: Wedge OutAddr InAddr -> CPU Bool
addressing Nowhere = return False
addressing (Here write) = do
    doWrite =<< zoom microState (outAddr write)
    return False
addressing (There read) = do
    doRead =<< zoom microState (inAddr read)
    return True

doWrite :: Either Port Addr -> CPU ()
doWrite target = do
    addrOut .:= target
    value <- use $ microState.valueBuf
    dataOut .:= Just value

doRead :: Either Port Addr -> CPU ()
doRead addr = addrOut .:= addr
```

The changes to `cpu` and `exec` are fairly small: just applying `addressing` on the initial setup and the after-instruction address request, and passing its result on.

```
cpu :: Pure CPUIn -> CPUM CPUState CPUOut ()
cpu CPUIn{..} = use phase >>= \case
    Fetching -> do
        let instr = dataIn
        microState.pc += 1
        let setup = setupFor instr
        load <- addressing (wedgeRight setup)
        phase .= Executing load instr 0
    Executing load instr i -> do
        when load $ microState.valueBuf .= dataIn
        exec instr i
    ...
```

```
exec :: Value -> Index MicroLen -> CPU ()
exec instr i = do
    let ((uop, post), cont) = microcodeFor instr !! i
    runExceptT (zoom microState $ uexec uop) >>= \case
        Right () -> do
            load <- addressing post
            maybe fetchNext (assign phase . Executing load instr) $
    succIdx i
        ...
```

The code providing the lookup table for the initial `Setup` of each instruction is similar to the microcode table, we just take the `fst` component of each microcode entry instead of the `snd`:

```
setupFor :: Value -> Setup
setupFor = asyncRom
    $(TH.lift $ map (fst . microcode . decodeInstr . bitCoerce) $
    indicesI @256)
```

Note that we need to keep `microcodeFor` and `setupFor` separate. If instead, we had `microcodeAndSetupFor :: Value -> (Setup, MicroOps)`, Clash would generate two copies of the same ROM, since it is accessed from two separate places (the two branches of the switch on `phase`), with two different addresses.

One more optimization we can do is to take the `cont` flag into account: since a `False` value means there is no point in continuing execution (since all that remains is padding), we can go to `fetchNext` at that point:

```
exec :: Value -> Index MicroLen -> CPU ()
exec instr i = do
    let ((uop, post), cont) = microcodeFor instr !! i
    runExceptT (zoom microState $ uexec uop) >>= \case
        Right () -> do
            load <- addressing post
            let i' = guard cont *> succIdx i
            maybe fetchNext (assign phase . Executing load instr) i'
        ...
```

Note, however, that mixing `addressing` and `fetchNext` in the same cycle can be bad news, since both may want to set the address bus. This bug is not introduced by the optimization that takes `cont` into account; rather, it was there in our previous version as well, but would only cause problems if we had any instruction that has no padding (i.e. is 8 micro-steps long) and finishes with a memory write. But now that we have a way of going to the next instruction earlier, it is important to ensure that the last real (i.e. non-padding) instruction has no write:

```
padded
    :: (KnownNat k, KnownNat n, ((n + 1) + k) ~ MicroLen)
    => MicroSteps (n + 1) pre False
    -> Microcode
```

This type-level change ensures this property holds without the need for any term-level checks. And in fact, when we try to compile our code after this check, we will find that several instructions (among them `push2`) require an extra `step INothing uNOP INothing` step just to avoid finishing on a step that writes to memory – these would cause faulty behavior if we didn't change `padded` to catch them statically.

15.9.1 Running the test suite

We can use similar scaffolding as the CHIP-8 to run the test suite on our CPU using high-level simulation. We start with an implementation of the `world` that consumes the CPU output to produce the next cycle's input:

```
world :: (Monad m) => World m -> Pure CPUOut -> m (Pure CPUIn)
world World{..} CPUOut{..} = do
    dataIn <- case _addrOut of
        Left port ->
            maybe (inPort port) (outPort port) _dataOut
        Right addr -> do
            x <- readMem addr
            traverse_ (writeMem addr) _dataOut
            return x
    interruptRequest <- pure False
    return CPUIn{..}
```

Then, we run the CPU and the world in tandem, feeding one into the other. As with the software implementation, `sim` returns a flag telling the rest of the simulation if the CPU is still running, i.e. not halted.

```
sim :: (Monad m) => World m -> StateT (Pure CPUIn, CPUState) m Bool
sim w = do
    (inp, s) <- get
    let (out, s') = runState (runCPU defaultOut $ cpu inp) s
    inp' <- lift $ world w out
    put (inp', s')
    return $ not $ _halted out
```

```
initInput :: Pure CPUIn
initInput = CPUIn
    { dataIn = 0x00
    , interruptRequest = False
    }

run :: IOArray Addr Value -> IO ()
run arr =
    evalStateT `flip` (initInput, initState 0x0100) $
    whileM $ sim (mkWorld arr)
```

The rest of the test running framework can be reused as-is from the software implementation.

15.9.2 Interrupt handling

The above takes care of straight-line execution, but what about interrupts? The two components of interrupt handling are *recording* the interrupt event, and then *reacting* to it. These two are separate because interrupt requests can come in at any time, including in the middle of executing an instruction; but reacting to them will have to wait until just before fetching the next instruction.

We record interrupt requests in a simple flag which is updated in every cycle in `cpu` independently of other, phase-specific state transitions. Of course, interrupt requests should only be latched when not masked, i.e. when `allowInterrupts` is set.

```
data CPUState = CPUState
    { _interrupted :: Bool
    ...
    }

latchInterrupt :: Pure CPUIn -> CPU Bool
latchInterrupt CPUIn{..} = do
    allowed <- use (microState.allowInterrupts)
    when (interruptRequest && allowed) $ interrupted .= True
    use interrupted
```

To accept an interrupt, we need to do a special kind of `Fetching`: instead of setting the address bus from the program counter, we need to raise the `interruptAck` output, and then wait one more cycle for the interrupt instruction to be available. Interrupts also enable coming back from the `Halted` state, which gives programmers a handy way to sleep until the next interrupt occurs.

```
cpu :: Pure CPUIn -> CPUM CPUState CPUOut ()
cpu inp@CPUIn{..} = do
    interrupted <- latchInterrupt inp
    use phase >>= \case
        Fetching False | interrupted -> do
            acceptInterrupt
            phase .= Fetching True
        Fetching interrupting -> do
            let instr = dataIn
            unless interrupting $ microState.pc += 1
            ...
        Halted -> when interrupted $ do
            acceptInterrupt
            phase .= Fetching True
        ...
```

When an interrupt is accepted, we also disable further interrupts until an `EI` instruction enables them back:

```
acceptInterrupt :: CPU ()
acceptInterrupt = do
    microState.allowInterrupts .= False
    interrupted .= False
    interruptAck .:= True
```

Although not needed for the two test suites we've been using, it can be instructive, even useful, to also update our high-level simulation framework to support interrupts. We store an extra `IRQ` component in the state of the `world`:

```
data IRQ
    = NewIRQ Value
    | QueuedIRQ Value

world
    :: (Monad m)
    => World m
    -> Pure CPUOut
    -> StateT (Maybe IRQ) m (Pure CPUIn)
```

Externally, a new interrupt can be raised by changing the state to `Just (NewIRQ instr)`. We leave the policy decision of what to do if there is already a queued interrupt to the rest of the simulation code. Internally, memory read requests are overridden with the instruction code from the `QueuedIRQ` by adding a new branch to the `dataIn` computation. The two new functions taking care of interrupt state management are `getInterrupt` and `newInterrupt`.

```
world World{..} CPUOut{..} = do
    dataIn <- case _addrOut of
        Right addr | _interruptAck ->
            getInterrupt
        ...
    interruptRequest <- newInterrupt
    return CPUIn{..}
  where
    getInterrupt = get >>= \case
        Just (QueuedIRQ instr) -> do
            put Nothing
            return instr
        _ -> return 0x00

    newInterrupt = get >>= \case
        Just (NewIRQ instr) -> do
            put $ Just $ QueuedIRQ instr
            return True
        _ -> return False
```

We can get rid of this new `StateT` layer in our test bench runner by starting from no queued interrupts, and never adding any:

```
run :: IOArray Addr Value -> IO ()
run arr =
    evalStateT `flip` Nothing $
    evalStateT `flip` (initInput, initState 0x0100) $
    whileM $ sim (mkWorld arr)
```

15.9.3 Stalling on memory contention

As we have seen in the CHIP-8 chapter, when peripherals are connected to the same RAM elements as the CPU, those peripherals may need to pre-empt the processor's memory access. The real Intel 8080 has the `READY` and `WAIT` pins for this purpose: when a peripheral needs access to RAM, the `READY` input is pulled low, which stalls the CPU on its next read from the data bus, signaled to the outside world by the `WAIT` output going high.

In our implementation, we leave the `WAIT` output to an exercise, and bundle `READY` with `dataIn`: if the memory is not ready for reading, we simply put `Nothing` on the data input pins of the CPU. This ensures we don't accidentally forget to implement stalling and mistake the memory output from an address requested by a peripheral with the intended read value.

```
data CPUIn = CPUIn
  { dataIn :: Maybe Value
  ...
  }
```

There are two points in our code where we access `dataIn`: during `Fetching` to get the machine code of the next instruction, and when `Executing` a microcode step that was preceded by a read request. In both of these cases, if `dataIn` contains `Nothing`, there is no meaningful way to apply the rest of the state changes in the given period, and we need to retry in the next one. We can do this easily by wrapping `CPUM` inside the `MaybeT` monad transformer, and treating `mzero` as a retry request.

```
type CPU = MaybeT (CPUM CPUState CPUOut)

cpu :: Pure CPUIn -> CPUM CPUState CPUOut ()
cpu inp@CPUIn{..} = void . runMaybeT $ do
    ...
```

We get rid of the `MaybeT` layer at the top of `cpu`, which ensures that the rest of it will only be executed up to the first `mzero`. Do we need to go through the rest of our code and `lift` everything? No, because `uexec` and `addressing` are polymorphic, and `MaybeT` has lifting instances of `MonadState`! All we need to do is to change direct accesses of `dataIn` with an effectful way of reading a byte, and use `readByte` both when `Fetching` and when `Executing`:

```
readByte :: Pure CPUIn -> CPU Value
readByte CPUIn{..} = maybe retry consume dataIn
  where
    retry = mzero
    consume x = return x
```

If we change our simulation-based test to always put `Just` values on the data input, it will seem as if we are done: just change `world` to reflect that the `World` can fail to provide read values; that is, that the underlying monad is wrapped in a `MaybeT`.

```
world
    :: (Monad m)
    => World (MaybeT m)
    -> Pure CPUOut
    -> StateT (Maybe IRQ) m (Pure CPUIn)
```

```
world World{..} CPUOut{..} = do
    dataIn <- case _addrOut of
        Left port -> lift . runMaybeT $
            maybe (inPort port) (outPort port) _dataOut
        Right addr | _interruptAck ->
            getInterrupt
        Right addr -> lift . runMaybeT $ do
            x <- readMem addr
            traverse_ (writeMem addr) _dataOut
            return x
    ...
```

With this change, we can run our test benches and see that they still work. However, this is misleading, because we haven't really exercised the new functionality: we run the test with the CPU always getting read results immediately. To expand the scope of our testing, we can extend the test bench program with a supply of `Bool` values to tell if memory should be ready in a given period. There are many ways to do this; here, we present one way, using the *monad-supply* package. Note the use of `cycle` to ensure an infinite list of Booleans, avoiding `evalSupplyT` to error out because of running out of items.

```
import Control.Monad.Supply

run :: [Bool] -> IOArray Addr Value -> IO ()
run accessPattern arr =
    evalStateT `flip` Nothing $
    evalStateT `flip` (initInput, initState 0x0100) $
    evalSupplyT `flip` cycle accessPattern $
    whileM $ do
        memReady <- supply
        lift $ sim (mkWorld memReady)
```

The only change to `mkWorld` here is the addition of a new `Bool` parameter, which we can use in `readMem` to decide if the result should be held back:

```
mkWorld :: Bool -> IOArray Addr Value -> World (MaybeT IO)
mkWorld memReady arr = World{..}
  where                   -- writeMem, inPort and outPort unchanged
    readMem addr = do
        guard memReady
        liftIO $ readArray arr addr
```

If we try running the tests with different memory accessPatterns from the imaginary peripherals, taking care to include at least some True cycles where the CPU gets a chance to use the RAM, we quickly run into trouble: either the tests hang, or print nonsense, or just produce failures.

To see what is going wrong, we should look at how the address bus changes whenever readByte fails with a retry. In the cycle before the first retry, the address bus is set directly to the desired address in doRead, for example by an IncrSP addressing mode. Then, in the first cycle, if the RAM was busy serving some other peripheral's request, readByte fails, leading to another readByte in the next cycle. However, since the address bus isn't restored, it will use whatever value is specified in defaultOut; in our case, that is to use the contents of the addrBuf. But that will not give the same result as IncrSP did in the previous cycle – and so the next cycle (unless it fails again) will read from the wrong address. Neither would it be a good idea to re-evaluate the last addressing mode, since that would result in SP being increased twice (and the second try would already be using the once-increased value).

The solution to this problem is to remember the last read address and the last value to be written until the read succeeds, and put that on the address bus for subsequent cycles. We put an address and a write latch in the CPU state, and use their value as the default for addrOut and dataOut:

```
data CPUState = CPUState
    { _addrLatch :: Either Port Addr
    , _dataOutLatch :: Maybe Value
    ...
    }

defaultOut :: CPUState -> Pure CPUOut
defaultOut CPUState{_microState = MicroState{..}, ..} = CPUOut{..}
  where
    _addrOut = fromMaybe (Right 0x0000) _addrLatch
    _dataOut = _dataOutLatch
    ...
```

In fact, we can do one better, by changing the type of addrOut in CPUOut to a Maybe; this allows circuits to implement peripherals that have lower priority access to RAM than the CPU.

```
data CPUOut = CPUOut
    { _addrOut :: Maybe (Either Port Addr)
    ...
    }

defaultOut :: CPUState -> Pure CPUOut
defaultOut CPUState{_microState = MicroState{..}, ..} = CPUOut{..}
  where
    _addrOut = _addrLatch
    ...
```

Now all that remains to be done is to update `addrLatch` and `dataOutLatch` both on read/write requests and successful reads. If there is no pending read/write request, we still return the `dataIn` value (if available); this is for the case where the data bus is populated by an external actor in the case of an interrupt. The `fromJustX` used here is a simulation-friendly version of `fromJust`.

```
readByte :: Pure CPUIn -> CPU Value
readByte CPUIn{..} = do
    pending <- isJust <$> use addrLatch
    if pending then maybe retry consume dataIn
        else return $ fromJustX dataIn
  where
    retry = mzero
    consume x = do
        addrLatch .= Nothing
        dataOutLatch .= Nothing
        return x

doRead :: Either Port Addr -> CPU ()
doRead addr = addrLatch .= Just addr

doWrite :: Either Port Addr -> CPU ()
doWrite target = do
    addrLatch .= Just target
    value <- use $ microState.valueBuf
    dataOutLatch .= Just value
```

The new structure of `cpu` is as follows: after latching interrupt requests (since that can happen while waiting for memory), we use `readByte` to ensure any pending read/write requests are fulfilled. Then, the result of `readByte` is used in all branches where normally, `dataIn` would be accessed (in the lines marked with `(*)` below):

```
cpu :: Pure CPUIn -> CPUM CPUState CPUOut ()
cpu inp@CPUIn{..} = void . runMaybeT $ do
    interrupted <- latchInterrupt inp
    dataRead <- readByte inp                             -- (*)

    use phase >>= \case
        Init -> do
            fetchNext
        Fetching False | interrupted -> do
            acceptInterrupt
            phase .= Fetching True
        Fetching interrupting -> do
            let instr = dataRead                         -- (*)
            unless interrupting $ microState.pc += 1
            let setup = setupFor instr
            load <- addressing (wedgeRight setup)
            phase .= Executing load instr 0
        Executing load instr i -> do
            when load $ microState.valueBuf .= dataRead  -- (*)
            exec instr i
        Halted -> when interrupted $ do
            acceptInterrupt
            phase .= Fetching True
```

With these changes applied, both test suites should pass with any access pattern that has at least some `True` values.

All that remains is exporting the complete CPU as a Clash signal function for our further chapters:

```
intel8080From
    :: (HiddenClockResetEnable dom)
    => Addr
    -> Signals dom CPUIn
    -> Signals dom CPUOut
intel8080From startAddr =
    mealyCPU (initState startAddr) defaultOut cpu

intel8080
    :: (HiddenClockResetEnable dom)
    => Signals dom CPUIn
    -> Signals dom CPUOut
intel8080 = intel8080From 0x0000
```

Exercises

- Add various status outputs from the real 8080's `SYNC` functionality, such as `WAIT`
- The implementation of addressing by `IncrPC`, `IncrSP` or `DecrSP` contains its own 16-bit incrementer/decrementer. Change the microcode to make use of the `Compute2` instruction instead, thereby sharing a single 16-bit ALU both for addressing and arithmetic.
- Implement `Compute2` via two rounds of 8-bit `Compute`, reusing a single 8-bit ALU
- The only use of `SwapReg2` that exploits its swapping behavior (i.e. that couldn't be implemented with a much simpler `ToReg2`) is in `XTHL` and `XCHG`. By using both the value buffer and the address buffer as temporary storage, it is possible to implement `XTHL` and `XCHG` using `ToReg2`, thereby obliviating the need for `SwapReg2`.
- Instead of microcode instructions that operate directly on register pairs, explore implementing everything in terms of single (8-bit) registers only.

15.10 Summary

- The Intel 8080 was a **real, physically existing CPU**. Some of its design decisions stem from the constraints of physical IC design, such as saving on the number of pins.
- Instead of following the hardware interface directly, we are **capturing the intended meaning of its pins**: on one hand, there is no constraint on space, so we can avoid multiple-duty pins like the various status bits on the data bus; on the other hand, we package up related pins, such as packing `READY` and the data-in bus into a single `Maybe Value` input.
- It is a **von Neumann machine** using instructions taking up between one and three bytes. **The first byte determines the instruction width**, so we implement generic fetching for the first byte, and fold the rest of the fetching into the execution of the instruction.
- Control flow can be changed externally using **interrupts**. The 8080's interrupt design in particular requires quite a lot of external machinery; we implement that outside our CPU core so that different computers using the same 8080 implementation can use different interrupting schemes, just like the real hardware.

- Our implementation is **microcoded**: each 8080 instruction is mapped to a sequence of simpler, more regular micro-instructions that can be executed in a single clock period. By **embedding the microcode into Haskell**, we get to use Haskell's tools for abstraction to reuse microcode fragments, and can use the host type system to enforce some microcode invariants.

16 Project: TinyBASIC

Compared to the Brainfuck computer and the CHIP-8 machine, the end result of the Intel 8080 chapter is less satisfying: we have built a processor, but without a full computer around it, there is no way to just jump in and start playing around with it. In this chapter, we design and build a very simple machine that boots straight into a BASIC interpreter, just like early home computers. The goal is not just the gratification of turning on a self-built computer and programming it. Instead, this chapter is also a stepping stone on our way to the two other, Intel 8080-based computers in the last chapters of this book: the Space Invaders arcade machine and the Compucolor 2 home computer.

16.1 What is Tiny BASIC?

Tiny BASIC originally started in 1975 as a community project to create a BASIC interpreter for the Altair 8800 kit computer, organized and released in a way that we would recognize today as an open source project. It quickly spread to other platforms, covering the big three home computer CPUs of Intel 8080, Motorola 6800 and MOS 6502. As its name suggests, one common thread among Tiny BASIC versions was its aim to be small and efficient enough to run on these early home computers with 8-bit CPUs running at single-digit MHz clock speed and very limited RAM size.

Because of its licensing, community development, and distribution primarily via printed newsletters, it is difficult to pinpoint *the* Tiny BASIC. Here, we will use the so-called *Palo Alto Tiny BASIC 2.0* by Li-Chen Wang[1]. This Intel 8080 version fits into 2 kB and is configured to be used with a total of 6 kB of RAM. In the name of simplicity, and because the Altair 8800 kit didn't come with keyboard input or video output as standard, all I/O is done via a serial console, accessed by the CPU through port operations. The actual UART functionality is implemented by the Motorola 6850 *Asynchronous Communications Interface Adapter*.

[1] Available online from https://www.autometer.de/unix4fun/z80pack/ftp/altair/tinybasic-2.0.prn

In this chapter, we will create two 8080-based computers that boot into Tiny BASIC. The first one only uses parts we have already developed, and provides a user experience similar to the Altair 8800 of the mid-seventies: the computer exposes a serial port, and a serial terminal is required to interact with it. The second version reimagines Tiny BASIC as a late-seventies home computer: a keyboard and a screen can be connected directly to the board. This will require us to write a PS/2 keyboard interface implementation and a video driver that displays text.

16.2 Asynchronous Communications Interface Adapter

Tiny BASIC itself is written with the assumption that an ACIA chip is connected to the CPU via ports 0x10 and 0x11. This chip, controlled via `IN` and `OUT` instructions in the program, is basically a UART implementation with a CPU-friendly interface consisting of two register addresses.

The real Motorola 6850 used in the Altair 8800 provides more functionality and configurability than what is used by Tiny BASIC. In particular, we will not implement error detection and interrupting, and hard-code the communication protocol of 9600 bits per second with 8 data bits and 1 stop bit. This leaves us with very little to worry about:

- Register 0 is the control/status register. Since the control register is used for configurability, we will simply ignore writes. Reading from this register returns a byte with bit 0 containing output state (high if ready to output, low if busy) and bit 1 containing input state (whether the next byte is ready for reading). We will not implement the functionality of the other bits and keep them at 0.

- Register 1 is the data register. Writing to it when the transmitter is not busy starts a new transmission; reading returns the last byte received, and clears the read-ready flag.

As we can see, reading from register 1 has a side-effect. This is an important difference between peripherals and memory, and we will represent port reads/writes with a custom `PortCommand` data type:

```
data PortCommand port a
    = ReadPort port
    | WritePort port a
    deriving (Generic, NFDataX, Show)
```

This way, a value of type `Maybe (PortCommand port a)` packs together both the selection of the given peripheral (in its `Just`-ness) and, when selected, the port command.

Since we already have a standalone UART implementation, and because we want to provide the same ACIA-compatible interface even in the second version of our Tiny BASIC computer that uses a keyboard and video output directly, no serial communication involved, we implement the ACIA without the actual serialization/deserialization built in. In other words, beside the `Maybe (PortCommand (Unsigned 1) (Unsigned 8))` input from the CPU and the `Maybe (Unsigned 8)` data output, the communication with the outside world is also in terms of a parallel `Unsigned 8` type.

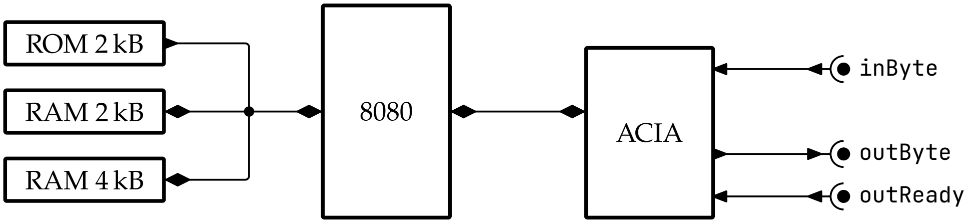
```
acia
    :: (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Unsigned 8))
    -> Signal dom Bool
    -> Signal dom (Maybe (PortCommand (Unsigned 1) (Unsigned 8)))
    -> ( Signal dom (Unsigned 8)
       , Signal dom (Maybe (Unsigned 8))
       )
```

Internally, the only state is the last-read value, stored when the input byte is `Just` a new value and cleared when the data register is read. The status register is computed from the receiving buffer and the transmitter's readiness signal; for this simplified implementation, all other flags are reported as `False`. The output is emitted when the data register is written; we use a `WriterT (First (Unsigned 8))` effect to do this "on the side" while computing the port read value.

```
acia inByte outReady cmd =
    mealyStateB step Nothing (inByte, outReady, cmd)
  where
    step (inByte, outReady, cmd) =
        fmap (second getFirst) . runWriterT $ do
            traverse (put . Just) inByte
            fmap fromJustX $ for cmd $ \case
                -- Data register
                ReadPort 0x1 -> do
                    queued <- get <* put Nothing
                    return $ fromMaybe 0x00 queued
                WritePort 0x1 x -> do
                    tell $ pure x
                    return 0x00
                -- Control register
                ReadPort 0x0 -> do
                    inReady <- isJust <$> get
                    return $ bitCoerce $
                        False :>
                        False :>
                        False :>
                        False :>
                        False :>
                        False :>
                        outReady :>
                        inReady :>
                        Nil
                WritePort 0x0 x -> do
                    return 0x00
```

16.3 The core logic board

We can share the logic board design between both versions of our Tiny BASIC computer. It consists of the 8080 core, 2 kB of ROM containing the Tiny BASIC software, 6 kB of RAM, and an ACIA peripheral controller. The only signal going in and coming out is via the ACIA.



```
logicBoard
    :: (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Unsigned 8))
    -> Signal dom Bool
    -> Signal dom (Maybe (Unsigned 8))
```

The details of the memory layout come from assumptions made by the Tiny BASIC version we use:

- The two registers of the ACIA are accessed through port numbers `0x10` and `0x11`.

- Since a real hardware Intel 8080 always starts execution at address 0x0000, we have to put ROM there. Since Palo Alto Tiny BASIC v2.0 is 1900 bytes, we round it up to 2 kB, giving us ROM from `0x0000` to `0x07ff`.
- Starting at `0x0800`, various states of the BASIC interpreter are stored before the actual program text. We need RAM starting at this address.
- Initialization code sets the stack pointer to `0x2000`. Recalling that the stack grows towards `0x0000`, that means we need RAM at least up to that address.

Putting it all together, we need 6 kB of RAM from `0x0800` to `0x1fff`. We split it into two parts for easier address decoding: 2 kB from `0x0800` to `0x0fff`, and 4 kB from `0x1000` to `0x1fff`.

To decode the `Either Port Address` coming from the CPU, we need to extend the `memoryMap` facility with a pair of new address matchers:

```
matchLeft
    :: Addressing addr1 a
    -> Addressing (Either addr1 addr2) a
matchLeft = matchAddr [| either Just (const Nothing) |]

matchRight
    :: Addressing addr2 a
    -> Addressing (Either addr1 addr2) a
matchRight = matchAddr [| either (const Nothing) Just |]
```

The other extension is for attaching peripheral endpoints. The two key differences between memory and an I/O `port` are that port reads can have side-effects, and that a port may have backpane signals going to the rest of the circuit (or to the outside world):

```
type Port dom addr dat a =
    Signal dom (Maybe (PortCommand addr dat)) ->
    (Signal dom (Maybe dat), a)

port
    :: forall addr' a addr. ()
    => ExpQ {- Port dom addr dat a -}
    -> Addressing addr (Handle addr', Result)
```

The implementation of `port` differs from `ram0` etc. only in that we don't assume peripherals to be synchronous, and so we do our own delaying. This gives a unified interface for the CPU: port access is exposed with the same semantics as synchronous memory, with read results appearing on the data bus at the next clock cycle.

```
port mkPort = readWrite $ \addr wr ->
  [| let cmd = cmdFromAddr <$> $addr <*> $wr
         (read, x) = $mkPort cmdk
     in (register undefined read, x) |]

cmdFromAddr :: Maybe addr -> Maybe a -> Maybe (PortCommand addr a)
cmdFromAddr (Just addr) Nothing = Just $ ReadPort addr
cmdFromAddr (Just addr) (Just w) = Just $ WritePort addr w
cmdFromAddr _ _ = Nothing
```

We are now ready to describe the full logic board in a very straightforward way: all connections to the CPU are handled by the `memoryMap` and there are no sources of interrupts.

```
logicBoard inByte outReady = outByte
  where
    CPUOut{..} = intel8080 CPUIn{..}

    interruptRequest = pure False

    dataIn = Just 0 |>. dataIn'
    (dataIn', outByte) =
      $(memoryMap [|Right 0 .|> _addrOut|] [|_dataOut|] $ do
        rom <- romFromFile (SNat @0x0800) [|"tinybasic.bin"|]
        ram <- ram0 (SNat @0x1800)
        (acia, outByte) <- port @(Unsigned 1) [|acia inByte outReady|]

        matchLeft @(Unsigned 8) $ do
            from 0x10 $ connect acia
        matchRight @(Unsigned 16) $ do
            from 0x0000 $ connect rom
            from 0x0800 $ connect ram

        return outByte)
```

In this simple computer, there are no shared resources where memory access contention could occur. Although our Intel 8080 core is written in such a way that memory access requests are explicitly signaled and peripherals can stall the CPU (which is just a fancy way of saying the `_addrOut` and `dataIn` field's types are wrapped in `Maybe`), here we don't use any of these features and write our memory map in terms of non-`Maybe` addresses and read-out values.

16.3.1 Testing

Since `logicBoard` contains the full Tiny BASIC computer apart from the details of connecting its input and output to the outside world, it is just the right unit to build an end-to-end testbench around it. We return to the Terminal package from chapter 6 to sample keypresses at every simulated clock cycle, feed that as input to `logicBoard`, and print its output by interpreting bytes as (printable) ASCII characters.

Unfortunately, Terminal doesn't provide a direct way to check for available events without waiting for any to occur, but we can use STM's `MonadPlus` instance to achieve it via the `awaitWith` interface, creating a non-blocking version of `awaitEvent`:

```
sampleEvent
    :: (MonadInput m)
    => m (Maybe (Either Interrupt Event))
sampleEvent = awaitWith $ \int ev -> msum
    [ Just . Left <$> int
    , Just . Right <$> ev
    , return Nothing
    ]
```

Here, the third branch doesn't block on either `int` or `ev` and immediately returns `Nothing` if neither of the preceding two is ready to return with an interrupt or an input event.

We turn `sampleEvent`'s output into a byte to feed into `logicBoard` by taking the ASCII value of characters, using the carriage return `'\r'` for the ENTER key, and ignoring everything else:

```
sampleKey :: (MonadInput m) => m (Maybe (Unsigned 8))
sampleKey = fromEvent <$> sampleEvent
  where
    fromEvent (Just (Right (KeyEvent key mods)))
        | CharKey c <- key, mods == mempty =
            Just . fromIntegral . ord $ c
        | EnterKey <- key, mods == mempty =
            Just . fromIntegral . ord $ '\r'
    fromEvent _ = Nothing
```

Printing a byte is a much simpler affair: `'\r'` is interpreted as a newline, printable characters are printed, and everything else is (again) ignored:

```
printByte :: (MonadPrinter m) => Unsigned 8 -> m ()
printByte val = case chr . fromIntegral $ val of
    '\r' -> putStringLn ""
    c | isPrint c -> putChar c
    _ -> return ()
```

Now that we have `sampleKey` to provide input, and `printByte` to consume output, we can hook these up to `logicBoard` using the Clash simulator. Since `printByte` works synchronously, printing then and there, we can keep the `outReady` input of `logicBoard` at `True`.

```
main :: IO ()
main = do
    sim <- simulateIO_ @System (uncurry logicBoard . unbundle)
        (Nothing, True)

    withTerminal $ runTerminalT $ forever $ sim $ \outByte -> do
        traverse_ printByte outByte
        inByte <- sampleKey
        return (inByte, True)
```

Running this testbench not only gives us assurance that our Tiny BASIC computer works, but also gives us our first user-programmable computer! Unlike the Brainfuck machine and CHIP-8, which required their program to be burnt into the FPGA configuration in the form of memory initializers, we can just type in programs in a simplified dialect of BASIC.

Note that Tiny BASIC only accepts keywords in uppercase, so e.g. `print "Hello"` is not syntactically valid, only `PRINT "Hello"`. Here's a sample interaction session:

```
TINY BASIC

OK
>10 FOR I=1 TO 10
>20 FOR J=1 TO I
>30 PRINT " ",
>40 NEXT J
>50 print "Hello from Clash"
>60 NEXT I
>LIST
  10 FOR I=1 TO 10
  20 FOR J=1 TO I
  30 PRINT " ",
  40 NEXT J
  50 print "Hello from Clash"
  60 NEXT I
```

```
OK
>run
WHAT?

OK
>RUN
WHAT?
  50 print "Hello from Clash"

OK
>50 PRINT "Hello from Clash"
>RUN
 Hello from Clash
  Hello from Clash
   Hello from Clash
    Hello from Clash
     Hello from Clash
      Hello from Clash
       Hello from Clash
        Hello from Clash
         Hello from Clash
          Hello from Clash

OK
```

Exercises

- One very annoying property of running Tiny BASIC via our test bench is that there is no way to exit the simulator, other than killing the process externally. Improve `sampleKey` so that an interrupt exits the simulation.

- Interrupts in Terminal correspond to Ctrl + C events from the user. However, Tiny BASIC is capable of handling Ctrl + C on its own, by breaking the execution of the current BASIC program and putting us back at the prompt, with the program intact in memory. Change `sampleKey` so that Terminal interrupts are turned into byte 0x03 (ASCII *end of text* marker), and use Ctrl + D (which is a `CharKey` event with `ctrlKey` set in the `mods`) to exit the simulation instead.

- Increase test coverage by simulating `outReady` as sometimes `False`. Similar to memory contention, we can use any (potentially randomly generated) scheme. A realistic one would be one that e.g. sets `outReady` to `False` for a certain number of cycles after a `printByte`.

16.4 Version 1: serial I/O

Connecting the `logicBoard` to the outside world via a serial UART is just three lines of code: one to instantiate the `logicBoard`, one to connect the serial receiver's output, and one to connect the serial transmitter's input. The rest is just the naming of the pins.

```
topEntity
    :: "CLK"   ::: Clock System
    -> "RESET" ::: Reset System
    -> "RX"    ::: Signal System Bit
    -> "TX"    ::: Signal System Bit
topEntity = withEnableGen board
  where
    board rx = tx
      where
        outByte = logicBoard inByte outReady
        inByte = fmap unpack <$> serialRx (SNat @9600) rx
        (tx, outReady) = serialTx (SNat @9600) (fmap pack <$> outByte)
```

Recall that Clash uses a 100 MHz clock domain as the default `System` one; hence, the type of `topEntity` needs to be adapted on FPGA boards with a different built-in clock.

16.5 PS/2 keyboard interface

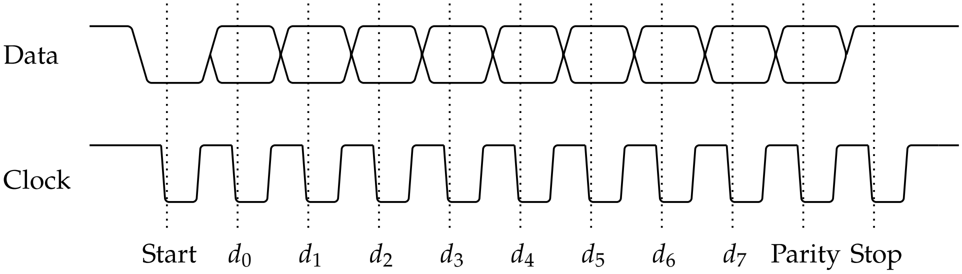
In the remainder of this chapter, we build a second version of our Tiny BASIC computer that interfaces directly with a keyboard and a screen, instead of requiring a serial terminal. First, we take care of keyboard input via the so-called PS/2 keyboard interface.

The name PS/2 comes from IBM's *Personal System/2*, a personal computer first released in 1987. It used a 6-pin connector for mouse and keyboard that became the standard de facto connector for these devices until USB took over. At first glance, it seems like a weird choice to pick PS/2 for interfacing with keyboards in this book: it is more modern than the actual computers we are building, but not modern enough to be around anymore: contemporary keyboards only have a USB connector.

However, for keyboard and mice, the PS/2 interface sits in the same sweet spot as VGA for video: it is simple enough to implement in our own hardware, and there is a huge stock of PS/2 and combo USB-PS/2 keyboards out there. Even better, some FPGA development boards contain a chip that translates USB to PS/2: the physical connector on the board is a USB socket accepting any modern USB keyboard, but the FPGA pins are connected to a PS/2 reinterpretation of the incoming data.

16.5.1 Synchronous serial communication

The PS/2 keyboard and mouse interface is a synchronous serial protocol. *Synchronous* here means that unlike in a UART, there is a separate, explicit clock line; all data line values are to be sampled on the falling edge of the clock line.



The full PS/2 protocol is bidirectional: the computer can send commands to the peripheral, setting the state of LEDs found on Num Lock and similar keys. Here, we only look at communication from the keyboard or mouse to the computer. In this case, the clock and the data lines are both driven by the peripheral. At the lowest level, sampling a PS/2 port simply means looking at the data line at the appropriate time:

```
data PS2 dom = PS2
    { ps2Clk  :: "CLK"  ::: Signal dom Bit
    , ps2Data :: "DATA" ::: Signal dom Bit
    }

samplePS2
    :: (HiddenClockResetEnable dom)
    => PS2 dom
    -> Signal dom (Maybe Bit)
samplePS2 PS2{..} = enable (isFalling low ps2Clk) ps2Data
```

On some development boards, this author has had problems with noise without some debouncing applied. Since the clock is specified to run between 10 KHz and 16.7 KHz, we can safely debounce at 1 μs, effectively creating a low-pass filter:

```
samplePS2
    :: forall dom. (HiddenClockResetEnable dom)
    => (KnownNat (ClockDivider dom (Microseconds 1)))
    => PS2 dom -> Signal dom (Maybe Bit)
samplePS2 PS2{..} =
    enable (isFalling low (lowpass ps2Clk)) (lowpass ps2Data)
  where
```

```
    lowpass :: Signal dom Bit -> Signal dom Bit
    lowpass = debounce (SNat @(Microseconds 1)) low
```

The rest of it comes down to making sense of this stream of sampled bits. In this chapter, we only look at keyboards; mice use the same serial protocol with different higher-level meaning of packets.

16.5.2 Bytes from bits

The next level of PS/2 is decoding the stream of bits into bytes. The situation here is very similar to a UART: pulling `low` indicates the start of a new byte, which is transmitted as 8 data bits, followed by a parity bit and a `high` stop bit. The parity bit is a so-called *odd* parity, meaning it is low if the data bits have high parity, and vice versa.

Note how much simpler the code is compared to the UART, since the explicit clock line allows us to fully decouple the sampling from the decoder state transitions. The `decoder` runs in a `WriterT` that produces a one-byte output at the end of a successful transmission.

```
data PS2State
    = Idle
    | Bit (BitVector 8) (Index 8)
    | Parity (BitVector 8)
    | Stop (Maybe (Unsigned 8))
    deriving (Show, Eq, Generic, NFDataX)

decoder :: Bit -> WriterT (Last (Unsigned 8)) (State PS2State) ()
decoder x = get >>= \case
    Idle -> do
        when (x == low) $ put $ Bit 0 0
    Bit xs i -> do
        let (xs', _) = bvShiftR x xs
        put $ maybe (Parity xs') (Bit xs') $ succIdx i
    Parity xs -> do
        put $ Stop $ unpack xs <$ guard (parity xs /= x)
    Stop b -> do
        when (x == high) $ tell . Last $ b
        put Idle
```

We compute `parity` efficiently by a binary tree of `xor` gates built by `fold`ing over the representation bit `Vector`:

```
parity :: forall a n. (BitPack a, BitSize a ~ (n + 1)) => a -> Bit
parity = fold xor . bitCoerce @_ @(Vec (BitSize a) Bit)
```

And finally, as usual, we turn the `decoder` into a circuit using `mealyState`, consuming the `Maybe Bits` coming out of the sampler:

```
decodePS2
    :: (HiddenClockResetEnable dom)
    => Signal dom (Maybe Bit)
    -> Signal dom (Maybe (Unsigned 8))
decodePS2 = mealyState sampleDecoder Idle
  where
    sampleDecoder = fmap getLast . execWriterT . traverse_ decoder
```

16.5.3 Packets from bytes

A standard keyboard has 104 keys[2], which can be crammed into 7 bits, leaving the topmost bit to distinguish between press and release events, right?

While a tempting idea, this is not at all how PS/2 works. The actual keyboard events, as transmitted from the keyboard to the computer, are represented as packets (called *scan codes*) containing one or more bytes. These bytes together encode the event type (pressed or released) and the key code of the key. The simplest case is the press of a so-called *non-extended* key: these are one byte only, containing the key code itself. Key releases are represented by prepending a `0xf0` byte to the key code. To support more than 255 different keys (`0xf0` being taken up for key releases), there are also *extended* key codes that use the marker `0xe0` as the first byte.

So putting it all together, the first byte of a PS/2 scan code can either signal an extended key code, a key release, or it can be the key code of a key press itself. If it is an extended code, the next byte might still not be the key code itself, since it itself can be a key release marker. Annoyingly, the release of a key with an extended scan code is represented as `0xe0` followed by `0xf0` followed by the rest of the code, instead of the much friendlier `0xf0` followed by `0xe0`; this means we can't just look at the first byte to distinguish between presses and releases.

Here, we represent extended or non-extended key codes in a unified way as a 9-bit number. Decoded PS/2 bytes are parsed into a scan code by starting in the `Init` state, and using a separate `Extended` state to remember if we should set the topmost bit of the key code once the full scan code is ready to be assembled.

[2] 105 for some language layouts

```
data KeyEvent = KeyPress | KeyRelease
    deriving (Generic, Eq, Show, NFDataX)

type KeyCode = Unsigned 9

data ScanCode = ScanCode KeyEvent KeyCode
    deriving (Generic, Eq, Show, NFDataX)

data ScanState
    = Init
    | Extended
    | Code KeyEvent Bit
    deriving (Show, Generic, NFDataX)

parser :: Unsigned 8 -> WriterT (Last ScanCode) (State ScanState) ()
parser raw = get >>= \case
    Init
        | raw == 0xe0 -> put $ Extended
        | raw == 0xf0 -> put $ Code KeyRelease 0
        | otherwise   -> finish KeyPress 0
    Extended
        | raw == 0xf0 -> put $ Code KeyRelease 1
        | otherwise   -> finish KeyPress 1
    Code ev ext       -> finish ev ext
  where
    finish ev ext = do
        tell $ pure $ ScanCode ev $ bitCoerce (ext, raw)
        put Init

parseScanCode
    :: (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Unsigned 8))
    -> Signal dom (Maybe ScanCode)
parseScanCode = mealyState byteParser Init
  where
    byteParser = fmap getLast . execWriterT . traverse_ parser
```

In simpler cases, we only care about key press events as they come in:

```
keyPress :: ScanCode -> Maybe KeyCode
keyPress (ScanCode KeyPress kc) = Just kc
keyPress _ = Nothing
```

16.5.4 Key codes

Interpreting the 9-bit key codes in a `ScanCode` is not a straightforward manner. Computers are expected to apply their own *keymap* to these codes: the same key code might mean `Z` with a QWERTY layout, or `Q` with Dvorak. For the Tiny BASIC, we will use a hardcoded key map that corresponds to QWERTY: each "normal" (non-modifier) key is mapped to its lowercase ASCII code. This way, we can feed the keyboard driver's output directly into the `logicBoard`.

```
charMap :: KeyCode -> Maybe Char
charMap 0x05a = Just '\r'
charMap 0x029 = Just ' '
charMap 0x04c = Just ';'
charMap 0x052 = Just '\''
charMap 0x054 = Just '['
charMap 0x05b = Just ']'
charMap 0x05d = Just '\\'
charMap 0x04a = Just '/'
charMap 0x041 = Just ','
charMap 0x049 = Just '.'
charMap 0x04e = Just '-'
charMap 0x055 = Just '='
charMap 0x045 = Just '0'
charMap 0x016 = Just '1'
charMap 0x01e = Just '2'
charMap 0x026 = Just '3'
charMap 0x025 = Just '4'
charMap 0x02e = Just '5'
charMap 0x036 = Just '6'
charMap 0x03d = Just '7'
charMap 0x03e = Just '8'
charMap 0x046 = Just '9'
charMap 0x01c = Just 'a'
charMap 0x032 = Just 'b'
charMap 0x021 = Just 'c'
charMap 0x023 = Just 'd'
charMap 0x024 = Just 'e'
charMap 0x02b = Just 'f'
charMap 0x034 = Just 'g'
charMap 0x033 = Just 'h'
charMap 0x043 = Just 'i'
charMap 0x03b = Just 'j'
charMap 0x042 = Just 'k'
charMap 0x04b = Just 'l'
charMap 0x03a = Just 'm'
charMap 0x031 = Just 'n'
charMap 0x044 = Just 'o'
charMap 0x04d = Just 'p'
charMap 0x015 = Just 'q'
charMap 0x02d = Just 'r'
charMap 0x01b = Just 's'
charMap 0x02c = Just 't'
charMap 0x03c = Just 'u'
charMap 0x02a = Just 'v'
charMap 0x01d = Just 'w'
charMap 0x022 = Just 'x'
charMap 0x035 = Just 'y'
charMap 0x01a = Just 'z'
charMap _ = Nothing

asciiMap :: KeyCode -> Maybe (Unsigned 7)
asciiMap = fmap (fromIntegral . ord) . charMap
```

This mapping is not complete, but should be enough for our purposes. Another important class of keys are the modifiers, such as Shift or Alt. The PS/2 protocol signals these keys as pressed or released individually, and it is up to the host to maintain their state and to assign any meaning to them. In our case, we want to care about Shift (for obvious reasons) and Ctrl, to detect Ctrl + C and pass it to the rest of the circuit as an ASCII EOT. For these cases, it is useful to keep the state of a given key:

```
keyState
    :: (HiddenClockResetEnable dom)
    => KeyCode
    -> Signal dom (Maybe ScanCode)
    -> Signal dom Bool
keyState target = regMaybe False . fmap fromScanCode
  where
    fromScanCode sc = do
        ScanCode ev kc <- sc
        guard $ kc == target
        return $ ev == KeyPress
```

16.5.5 The complete keyboard driver

The keyboard driver for our Tiny BASIC computer consumes the raw PS/2 bit stream, and produces ASCII codes extended to full bytes, ready to be connected to the ACIA. The Ctrl modifier is detected by watching for key codes 0x014 and 0x114; there are two of them because the PS/2 protocol assigns a different key code to modifiers on the left and the right side of the keyboard. Similarly, left Shift has key code 0x012 while the right one is 0x059.

```
keyboard
    :: (HiddenClockResetEnable dom,
        KnownNat (ClockDivider dom (Microseconds 1)))
    => PS2 dom
    -> Signal dom (Maybe (Unsigned 8))
keyboard ps2 = fmap extend <$> (toChar <$> shift <*> ctrl <*> sc)
  where
    sc = parseScanCode . decodePS2 . samplePS2 $ ps2

    shift = keyState 0x012 sc .||. keyState 0x059 sc
    ctrl = keyState 0x014 sc .||. keyState 0x114 sc

    toChar shift ctrl sc = case asciiMap =<< keyPress =<< sc of
        Just 0x63 | ctrl -> Just 0x03 -- Ctrl-C
        Just c | not ctrl -> Just $ shiftASCII shift c
        _ -> Nothing
```

Since Tiny BASIC only accepts keywords in uppercase, we write `shiftASCII` to emulate the behavior of the Caps Lock key: for characters above 0x40, unshifted input produces uppercase, and shifted input produces lowercase. The meaning of “uppercase” and “lowercase” is also simplified to a single bit flip. This causes, for example, the double quote " to be mapped to Shift + 2 instead of the more conventional Shift + ' . Supporting the standard US keyboard layout would require an unwieldy lookup table instead of the simple bit-twirling logic below.

```
shiftASCII :: Bool -> Unsigned 7 -> Unsigned 7
shiftASCII shift c
  | c > 0x40 = if shift then c `setBit` 5 else c `clearBit` 5
  | c > 0x20 = if shift then c `clearBit` 4 else c
  | otherwise = c
```

16.6 Textual video

Showing text in video involves picking the right glyphs from some font and rendering them in the correct positions. These days, computers have no problem with complicated text layouts composited with other, non-textual images; but the home computers of the late seventies and eighties had a much simpler notion of textual video. In these systems, text was rendered in some monospaced font at a fixed regular grid. For example, the first version of the Commodore PET, released in 1977, used a text resolution of 40 × 25 characters with a single built-in font.

We can think of these regular grid-based text modes as a special case of tiled graphics. Instead of containing pixel data directly, each cell in the video RAM stores the character at the given position, and a separate font memory is consulted to find the glyph of that character. The font memory can be ROM or RAM depending on whether user-configurable fonts are supported.

The steps to calculate the value of a pixel at (x, y) can thus be described in the following way, handwaving away the details for now:

1. The *screen coordinate* (x, y) is converted to a *character coordinate* and a *glyph coordinate*.

2. The character coordinate is used as an address to load the *character* from video RAM.

3. The character and the glyph coordinate together is used as an address to load the right pixel of the *glyph* from the font.

4. The pixel data is turned into an RGB color. The details of this step depend on the specifics of the video system; in this chapter, we will use fixed-color monochrome video. In chapter 18, we will build a computer where each character cell has an individually configurable background and foreground color.

Note the data dependency between the first memory access (from video RAM) and the second memory addressing (to the font memory). Tracking the signal delay in the type system, as we have done in chapter 13, is going to be even more important here; otherwise, we get weird, hard-to-debug rendering artifacts of the "why is every fifth column off by 2 pixels" kind.

Since our Tiny BASIC computer has no way to readback from the text shown on screen, our video controller in this chapter will provide a write-only interface to video RAM:

```
video
    :: (HiddenClockResetEnable Dom25)
    => Signal Dom25 (Maybe (TextAddr, Unsigned 8))
    -> VGAOut Dom25 8 8 8
video (fromSignal -> w) = delayVGA vgaSync rgb
  where
    VGADriver{..} = vgaDriver vga640x480at60
    -- Continued below
```

A separate component will take care of turning the stream of output characters coming from the ACIA into video RAM writes.

16.6.1 Character addressing

Turning the raw (x, y) video coordinate into a character coordinate is a straightforward matter of scaling; for example, if the font size is 8×8 pixels, we can use `scale` on both coordinates to convert (x, y) into $(\lfloor x/8 \rfloor, \lfloor y/8 \rfloor)$.

```
  (charX, glyphX) = scale @TextWidth  (SNat @FontWidth)  . center $ vgaX
  (charY, glyphY) = scale @TextHeight (SNat @FontHeight) . center $ vgaY
```

However, we then need to transform that character position `(charX, charY)` to a linear address into the video memory.

The easy way out is to make either the horizontal or the vertical textual resolution a power of 2. For example, suppose we decide on a resolution of 50×32 characters. We can then turn any text coordinate (of type `(Index 50, Index 32)`) into a linear address (of type `Unsigned 11`) by putting the five bits of the vertical address into the lower bits:

```
addr :: (Index 50, Index 32) -> Unsigned 11
addr = bitCoerce
```

Similarly, if the horizontal resolution is a power of 2, we can just flip the representation:

```
addr :: (Index 64, Index 40) -> Unsigned 12
addr (x, y) = bitCoerce (y, x)
```

This second variant has the added benefit that consecutive characters are stored at consecutive addresses. So if the rest of the computer wants to display a string, only the starting address needs any coordinate computation; the rest of it is a simple manner of iterative incrementing.

However, if we want to create computers based on existing designs, the text resolution is not a free variable. A Commodore PET needs exactly 40×25 characters on the screen; moreover, those 1000 characters are stored in a single, contiguous, 1000-byte area of RAM. To *address* this problem head-on, we will deliberately make things awkward in this chapter, and use a text resolution of 72×50 with a linear address space of $72 \cdot 50 = 3600$ bytes:

```
type TextWidth = 72
type TextHeight = 50
type TextSize = TextWidth * TextHeight
type TextAddr = Index TextSize
```

In this setup, computing the address of the character that is at the currently drawn raster dot can be by keeping a *line address* (the base address of the current line) and a *column offset*. The line address is incremented whenever the character Y coordinate changes, and the column offset is incremented every time the character X coordinate changes. The character address is simply the sum of the two.

```
    (newLine, lineAddr) = addressBy (snatToNum (SNat @TextWidth)) charY
    (newChar, charOffset) = addressBy 1 charX

    charAddr = lineAddr + charOffset
```

The reason to keep two counters instead of just one, is because the line address stays the same for multiple raster lines, as many as the height of the font. For example, if the font is 8 pixels tall, the column offset needs to sweep the full interval of $0, 1, \ldots, 71$ a total of eight times without the line address changing. We can see the implementation of `addressBy` takes care to reset the address component whenever the underlying coordinate is zero; this is precisely to avoid the character offset "spilling over" into the next line by taking on a value larger than `TextWidth`.

```
addressBy
    :: (NFDataX coord, NFDataX addr, Num coord, Eq coord, Num addr)
    => (HiddenClockResetEnable dom)
    => addr
    -> Signal dom (Maybe coord)
    -> (DSignal dom 0 Bool, DSignal dom 1 addr)
addressBy stride coord = (new, addr)
  where
    start = fromSignal $ coord .== Just 0
    new = fromSignal $ changed Nothing coord
    addr = delayedRegister 0 $ \addr ->
        mux (delayI False start) 0 $
        mux new (addr + pure stride) $
        addr
```

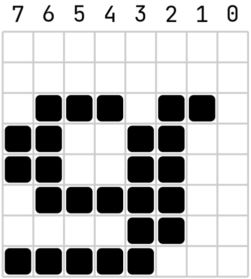
Given the character address, we can then use that to load the currently displayed character from the video RAM. Because the CPU has no read access to the video RAM, there is no room for access contention: even though we will only use the loaded value when `newChar` holds, we might as well read from `charAddr` in every cycle.

```
    charLoad = delayedRam (blockRam1 NoClearOnReset (SNat @TextSize) 0)
        charAddr (delayI Nothing w)
```

16.6.2 Glyph addressing

The character loaded from video RAM is an index into the font memory; in our case, the font ROM. The font ROM contains a glyph of each character. Each glyph is a monochrome mask of a set size; for example, using Marcel Sondaar's Public Domain 8 × 8 font from https://github.com/dhepper/font8x8, this is what the lowercase letter "g" looks like:

		7654 3210
0	0x00	0b0000_0000
1	0x00	0b0000_0000
2	0x76	0b0111_0110
3	0x99	0b1100_1100
4	0x99	0b1100_1100
5	0x7c	0b0111_1100
6	0x0c	0b0000_1100
7	0xf8	0b1111_1000



On any given raster line, we will need to render a single row of this image before switching over to the same row of the next character's glyph. Therefore, it makes sense to make the row be the unit of storage for the glyph data. In other words, the font ROM will contain eight bytes for each glyph; for example, the bytes `0x00 0x00 0x76 0x99 0x99 0x7c 0x0c 0xf8` for the character "g". To get the n-th row of the glyph for character c, we just look up the byte at address $8c + n$ in a piece of ROM. The ROM is loaded with the contents of a font file in the above format: eight bytes per glyph, row by row.

```
type FontWidth = 8
type FontHeight = 8

fontRom
    :: (HiddenClockResetEnable dom)
    => DSignal dom n (Unsigned 8)
    -> DSignal dom n (Index FontHeight)
    -> DSignal dom (n + 1) (Unsigned FontWidth)
fontRom char row = delayedRom (fmap unpack . romFilePow2 "font.bin") $
    bitCoerce <$> D.bundle (char, row)
```

The current glyph row is loaded from the above `fontRom` for each `newChar` by taking the glyph Y coordinate from the vertical coordinate scaler. We then shift out the current row into the current pixel; this is similar to the CHIP-8 video controller from chapter 13. The only reason to start with the highest bit is so that the glyphs visually match their bit pattern; this is also how fonts were stored in binary format on most home computers. Of course, if the font data is flipped, we also need to flip the shift direction.

```
    glyphLoad = fontRom charLoad (delayI Nothing (fromSignal glyphY) .<|
    0)
    glyphRow = delayedRegister 0x00 $ \glyphRow ->
      mux (delayI False newChar) glyphLoad $
      (`shiftL` 1) <$> glyphRow

    visible = fromSignal $ isJust <$> charX .&&. isJust <$> charY
    pixel = enable (delayI False visible) $ msb <$> glyphRow
```

While the above works for native resolutions, what happens if we want to scale up, for example, the PET's 40 × 25 text screen with an 8 × 8 font to the standard VGA resolution of 640 × 480? Each glyph pixel now corresponds to a 2 × 2 rectangle of physical pixels; and that means we should only shift `glyphRow` whenever the logical X coordinate (i.e. `glyphX`) changes. This is exactly the same situation we have encountered with the CHIP-8 machine, which scaled its 64×32 internal frame buffer by 10.

```
    newCol = fromSignal $ changed Nothing glyphX
    glyphRow = delayedRegister 0x00 $ \glyphRow ->
      mux (delayI False newChar) glyphLoad $
      mux (delayI False newCol) ((`shiftL` 1) <$> glyphRow) $
      glyphRow
```

We lift this pattern of "left-shifter with reload" for reuse in further chapters. For versatility, we change it to avoid the extra one-cycle delay via the usual trick of returning the `next` value of the register instead of the `register` itself:

```
shifterL
    :: (BitPack a, HiddenClockResetEnable dom)
    => Signal dom (Maybe a)
    -> Signal dom Bool
    -> Signal dom Bit
shifterL load tick = msb <$> next
  where
    r = register 0 next
    next = muxA
        [ fmap pack <$> load
        , enable tick $ (`shiftL` 1) <$> r
        ] .|>.
        r
```

We can rewrite the definition of `pixel` using `shifterL` by making sure `glyphLoad` is only `enabled` when a new glyph row should be loaded:

```
    glyphLoad = enable (delayI False newChar) $
        fontRom charLoad (delayI Nothing (fromSignal glyphY) .<| 0)
    newCol = delayI False $ fromSignal $ changed Nothing glyphX
    pixel = enable (delayI False visible) $
        liftD2 shifterL glyphLoad newCol
```

Here, `liftD2` is a binary verison of `liftD` written in exactly the same way, just with two arguments:

```
liftD2
    :: (HiddenClockResetEnable dom)
    => (forall dom'. (HiddenClockResetEnable dom') => Signal dom' a ->
    Signal dom' b -> Signal dom' c)
    -> DSignal dom d a -> DSignal dom d b -> DSignal dom d c
liftD2 f x y = unsafeFromSignal $ f (toSignal x) (toSignal y)
```

Finally, it is easy to see that this same approach would work for user-editable fonts as well. In the most versatile setup, both the character RAM and the font RAM can be parts of a single main RAM accessible to other parts of the computer (most notably, the CPU) as well. Character and glyph addresses are resolved by adding the base addresses of the two memory areas to the addresses computed from the raster position. The only constraint is that the font must not be wider than one memory cell. If, for example, we have `FontWidth ~ 10` and main memory containing `Unsigned 8` values (i.e. bytes), then loading glyph rows becomes more complicated than presented here.

16.6.3 Rendering into colors

Nothing new here: we use the same palette-based logic as in the similarly monochrome CHIP-8. For variety's sake, we will use a green foreground color, reminiscent of old phosphorous screens:

```
    rgb = maybe frame palette <$> pixel

    frame = (0x30, 0x30, 0x30)
    palette 0 = (0x00, 0x00, 0x00)
    palette 1 = (0x33, 0xff, 0x33)
```

16.7 Screen editing

Our `video` controller is capable of rendering a full screen's worth of text, based on previous writes of characters to screen positions represented by video RAM addresses. In a "normal" computer, this is usually sufficient, since any software running on the CPU can do its own internal housekeeping to calculate the correct coordinates when wanting to print text. This software can either be part of the user's program, or provided via operating system routines.

Our Tiny BASIC computer, however, has no program running on it other than the Tiny BASIC interpreter itself. And since said interpreter was originally written to do serial I/O via the ACIA chip, it has no functionality to maintain a cursor position so that subsequent characters can be printed left to right, top to bottom.

In this section, we develop a hardware solution to this problem: a screen editor circuit that consumes the output serially, and converts it to a sequence of video RAM writes.

At its simplest, we can exploit that video RAM addresses are linear, and use a mere counter:

```
screenEditor
    :: (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Unsigned 8))
    -> Signal dom (Maybe (TextAddr, Unsigned 8))
screenEditor chr = packWrite <$> addr <*> chr
  where
    addr = regEn 0 (isJust <$> chr) $ nextIdx <$> addr
```

However, this has serious limitations when it comes to handling multiple lines of output. First, let's add support for consuming newline characters. When the `chr` is a newline (ASCII `\n` 0x0a), the subsequent character should be printed to the first column of the next row of text. We can use a trick similar to the video controller to achieve this, by storing the current line's base address and the current column's address offset separately:

```
screenEditor = mealyState putChar (0, 0)
  where
    base `offsetBy` x = base + fromIntegral x

    stride = snatToNum (SNat @TextWidth)
    nextLine = satAdd SatWrap stride

    putChar chr = do
        (base, x) <- get
        case chr of
            Nothing -> do
                return Nothing
            Just 0x0a -> do
                put (nextLine base, 0)
                return Nothing
            Just chr -> do
                put (base, nextIdx x)
                return $ Just (base `offsetBy` x, chr)
```

At this point we are also set up to implement line-wrapping: whenever `x` would overflow, we increment the `base` address to point to the next line instead:

```
Just chr -> do
    put $ maybe (nextLine base, 0) (base,) $ succIdx x
    return $ Just (base `offsetBy` x, chr)
```

The other big missing feature of our screen editor is handling running out of vertical real estate. After 50 lines of output, where should the next line be put? With our implementation of `nextLine`, the current position jumps from the last line back to the first one. This isn't too bad on its own (we will see an alternative suggestion as an exercise), but if the newly printed lines are shorter than the previous ones, the rest of the old first line will remain as garbage to the right of the fifty-first line. For example, if the first line printed was `Clash is awesome`, the fiftieth was `Look at the crap that most`, a reasonable fifty-first line is `HDLs are`, but it would lead to the very misleading `HDLs are awesome` line being shown at the top of the screen.

We are going to improve this by *clearing the next line* when going to a new line. To clear a line, we want to emit a full line's worth of space (ASCII 0x20) characters. Of course, this is going to take some time – 72 cycles, to be exact; during which the screen editor is unable to process character-printing requests. Luckily, because our Tiny BASIC computer was designed around serial communication, which is even slower (a *lot* slower), we already have a way of doing flow control: the ACIA has the "output ready" pin for exactly this reason.

We extend the type of `screenEditor` to expose a readiness signal in the same format as our UART: a second output signal of type `Bool`.

```
screenEditor
    :: (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Unsigned 8))
    -> (Signal dom (Maybe (TextAddr, Unsigned 8)), Signal dom Bool)
```

Our screen editor can now be in one of two states: either accepting requests and printing them to the next position of the current line, or busy clearing the next line.

```
data EditorState
    = Ready TextAddr (Index TextWidth)
    | Clear TextAddr (Index TextWidth)
    deriving (Generic, NFDataX)
```

Not much changes in `screenEditor`'s implementation beside the state type. The `Clear` state just iterates through all values of `x` before returning to the `Ready` state; because we use `Clear` to prepare the next line for the output that is coming, when done, we simply set the next output position back to the beginning of the just-cleared line.

```
screenEditor = mealyStateB step (Ready 0 0)
  where
    base `offsetBy` x = base + fromIntegral x

    stride = snatToNum (SNat @TextWidth)
    nextLine = satAdd SatWrap stride

    step chr = do
        write <- putChar chr
        ready <- gets $ \case
            Ready _ _ -> True
            Clear _ _ -> False
        return (write, ready)

    putChar chr = get >>= \case
        Clear base x -> do
            put $ maybe (Ready base 0) (Clear base) $ succIdx x
            return $ Just (base `offsetBy` x, 0x20)
        Ready base x -> case chr of
            Nothing -> do
                return Nothing
            Just 0x0a -> do
                put $ Clear (nextLine base) 0
                return Nothing
            Just chr -> do
                put $ maybe (Clear (nextLine base) 0) (Ready base) $
                    succIdx x
                return $ Just (base `offsetBy` x, chr)
```

16.8 Version 2: Keyboard and video

The `keyboard` driver, the `video` driver and the `screenEditor`, while doing quite a lot internally, expose the same one-character-at-a-time interface than a UART.

Let's look at the two versions side by side. First, we have the serial one from before, repeated here for easy comparison:

```
topEntity
    :: "CLK"   ::: Clock System
    -> "RESET" ::: Reset System
    -> "RX"    ::: Signal System Bit
    -> "TX"    ::: Signal System Bit
topEntity = withEnableGen board
  where
    board rx = tx
      where
        outByte = logicBoard inByte outReady
        inByte = fmap unpack <$> serialRx (SNat @9600) rx
        (tx, outReady) = serialTx (SNat @9600) (fmap pack <$> outByte)
```

And then the new version, where `inByte` is fed from the result of decoding the PS/2 keyboard input, and `outByte` and `outReady` are handled by our `screenEditor`:

```
topEntity
    :: "CLK_25MHZ" ::: Clock Dom25
    -> "RESET"     ::: Reset Dom25
    -> "PS2"       ::: PS2 Dom25
    -> "VGA"       ::: VGAOut Dom25 8 8 8
topEntity = withEnableGen board
  where
    board ps2 = vga
      where
        outByte = logicBoard inByte outReady
        inByte = keyboard ps2
        (vidWrite, outReady) = screenEditor outByte
        vga = video vidWrite
```

And here we have, in just a handful of lines of code, an interactive home computer of our own design, running a BASIC interpreter just like the real machines of the late seventies and early eighties.

Exercises

- Extend the ACIA implementation to provide the full functionality of the Motorola 6850; full datasheets for the chip can be found online. The main features to add are configurable bit rates and more extensive error reporting. At startup, Tiny BASIC writes 0x15 into the control register of the ACIA, which corresponds to 8 data bits, no parity bit, and one stop bit, with the serial clock computed from the master clock by dividing by 16. For our Clash implementation, it makes more practical sense to instead have three hardcoded common bit rates (e.g. 2400, 9600, and 115,200), so that the main clock period can be changed independently.
- Hook up the PS/2 keyboard driver to the designs from previous chapters. It can be used with the CHIP-8 by using some 4 × 4 area of keys; it can be used with the Brainfuck computer by listening for hexadecimal digits. Similarly, the calculator extends naturally to allowing keyboard input. As an extra exercise, instead of *replacing* the existing keypad or serial input, add the PS/2 input as an *alternative*, handling inputs from all peripherals at the same time.
- Implement a cursor to show the position of the next character. This requires extending the interface of the video controller, to allow other parts of the circuit to provide a *cursor position* signal. The screen editor is in a perfect place to provide that signal. For extra credit, make the cursor blink.
- Instead of wrap-around, implement vertical screen scrolling when the cursor gets to the last line of the screen. One way of implementing this, that is easy to do in hardware, is to add a vertical offset input signal to the video controller, and use that when computing the address of the currently drawn character. Then, the screen editor can maintain the vertical offset so that it starts at zero, and remains zero until the last line is used up. When we get to the last line, the vertical offset increases by one, causing everything on the screen to shift up by one line, putting the previously first line as the new last one.

16.9 Summary

- Tiny BASIC was originally designed to run on very small and simple computers, requiring only a single peripheral chip beside the CPU. Our first version recreated exactly this minimal setup.

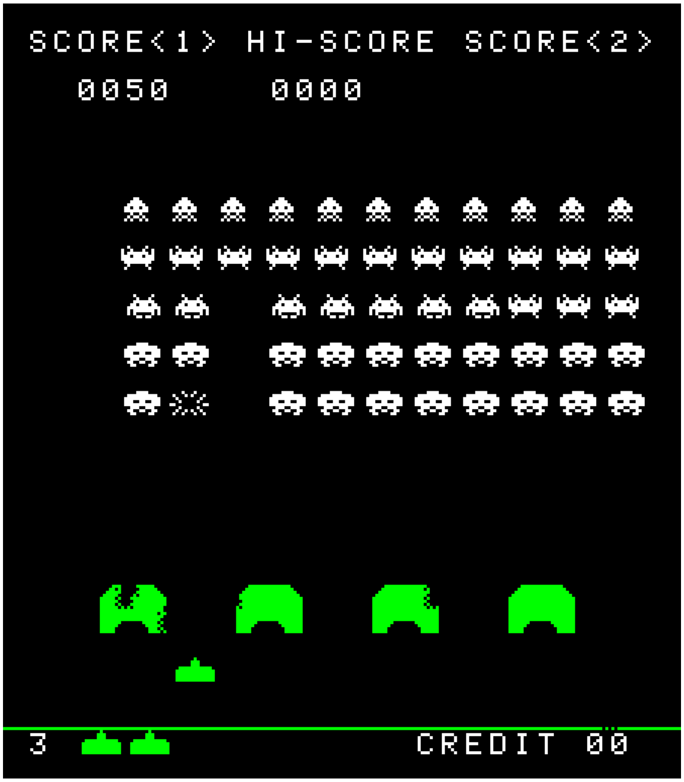
- By decoupling the serialization/deserialization from flow control, we were able to reuse the same ACIA implementation with other input/output modalities; in this case, we replaced serial input with a keyboard and serial output with video.

- PS/2 is a synchronous serial protocol which is very easy to implement in hardware, and gives access to a wide range of off-the-shelf keyboards.

- To implement textual video, we do two levels of indirection: the video memory contains the characters in each cell, and the font memory contains the shape of each character; the former is an index into the latter. Other data can also be stored per cell; for example we could have two bytes for each text position, one containing the character and the other containing the foreground and background color.

17 Space Invaders

In this chapter, we create a functional replica of Taito's trailblazing arcade machine Space Invaders. Released in 1978 to huge commercial success all over the world, it pushed video gaming to the mainstream, and established a lot of the language of arcade games, especially for the *shoot-'em-up* genre. Space Invaders is a great choice for us not only for its historical relevance, but also because at its heart lies an Intel 8080 microprocessor.



17.1 The design of Space Invaders

Looking at the Space Invaders arcade game as a physical object, it comes in two formats:

- A roughly human-sized upright cabinet with a CRT screen in portrait orientation and some buttons in front of it

- A so-called "cocktail table" layout with the screen in the middle, and buttons on both sides.

It might seem like the screen on the upright cabinet version produces a color picture, but in reality, it is the same black-and-white CRT as the cocktail table, with transparent colored overlays at the bottom (green) and the top (red).

Beside the buttons to start a one- or two-player game, and the left/right/fire buttons, there are three other peripherals on the machine that aren't immediately obvious: there is a set of 8 DIP switches to control game settings such as difficulty or scoring; there is a tilt sensor to detect physical abuse; and finally, but most importantly to the arcade owners, there is the coin slot.

Already we can see that an arcade machine is designed very differently than a generic-purpose microcontroller. The whole machine only runs a single game, and if Space Invaders requires the player to move their ship left/right and fire, then three buttons of input per player are enough. Similarly, screen orientation and the color overlays pasted on only make sense for the layout of this one game.

Looking inside, we find the Intel 8080 connected to 8 kB of ROM holding the game code and 1 kB of generic-purpose RAM. The video system uses a framebuffer to produce its 256×224 output image. Since the image is black & white, one bit per pixel is enough to store the whole frame; putting a run of eight pixels in a byte, this means a total of 7 kilobytes of video RAM.

Note that the video resolution is given as 256×224, not 224×256 as one would expect for a portrait-oriented screen. That is because Taito, of course, didn't manufacture a custom portrait CRT; instead, a standard 4:3 CRT was put in the cabinet rotated -90 degrees. The Space Invaders firmware simply produces the image rotated at 90 degrees, so that combined with the physical rotation of the screen, the resulting image is correct.

The input peripherals (the buttons and the DIP switches) and the sound system are connected to the CPU via its I/O ports. Audio is generated by a circuit of discrete analog components; this part of the machine is outside the scope of this book.

There is also a *barrel shifter* connected to the CPU. Of all the components of the Space Invaders hardware, this is the most specialized to the game at hand. Since the video RAM is accessed in units of 8 bits, to draw a pattern (such as an Invader) anywhere on the screen which is not exactly on an 8 × 8 grid, we need to shift the bits of pattern data by the right amount of offset. However, the Intel 8080 only has instructions to shift by one bit at a time; and to update a full screen's worth of Space Invaders, that would be a lot of shifts in a lot of small loops. The Space Invaders designers solved this by adding an external chip that reads both the bits to shift and the shift amount as its two inputs, and produces the shifted bits in one cycle.

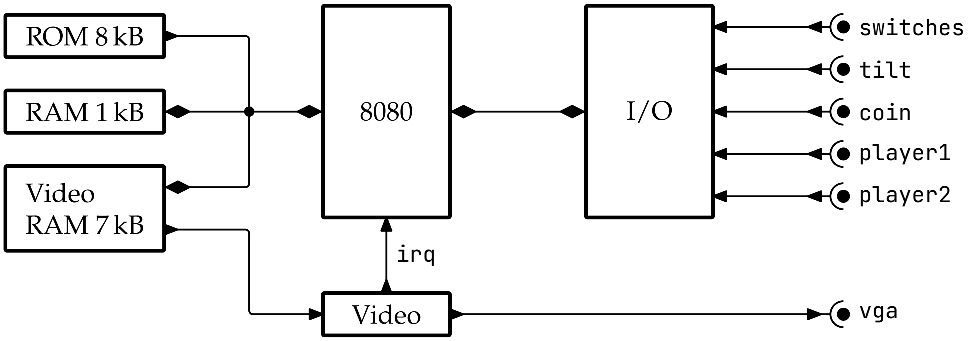
17.2 How it fits together

Now that we have a high-level understanding of the components of the machine, it is time to dive into the specifics. The Computer Archeology website's Space Invaders page at `https://www.computerarcheology.com/Arcade/SpaceInvaders/` has been an invaluable resource for this chapter. Beside detailed descriptions of the memory and I/O maps, it also contains a complete, commented disassembly of the original Space Invaders firmware. Since our Intel 8080 core is supposed to be compatible with the original, if we hook up the right peripherals, we should be able to use the original firmware as-is, treated as a black box. Nevertheless, the disassembly can still be an interesting read on software development from a very different era with very different constraints.

The only hardware difference between the upright cabinet and the cocktail table versions is that beside the coin slot and the buttons to start a one- or two-player game, the cabinet only has one set of player input buttons, while the table has two. This difference is papered over by simply connecting the buttons of the cabinet to both the player 1 and player 2 input lines. In terms of software, the cocktail table version includes code to rotate the screen by 180 degrees between the two players sitting at opposite ends of the table. This is not configurable: the two versions of the firmware simply shipped as different ROM images.

The input peripherals, the sound circuit and the barrel shifter are connected to the CPU via I/O ports. The assignment to port numbers is very ad-hoc: for example, reading from port 2 accesses some of the DIP switches and player two's button state; but writing to port 2 sets the shift amount (as a 3-bit number) of the barrel shifter. The simplest way to deal with this is to just put everything inside one dedicated "peripheral chip" that can map `ReadPort` and `WritePort` requests to different functionality without any regard to consistency between reads and writes to the same port. We will also fold the full barrel shifter implementation into this circuit.

Thus we arrive at the following high-level design:



Communication between the CPU and the video system is via two channels:

- The video RAM is shared between the CPU (with read and write access) and the video system (with read-only access). Similar to the previous designs, the video system has priority over the CPU.
- The video system generates an interrupt for the CPU when it gets to raster lines 96 and 224. This allowed the programmers to update the the game state and the screen, synchronized to the frame rate, without having to count cycles.

This video-originating interrupt is a godsend for us: while the original hardware runs the 8080 at 2 MHz, we can run our CPU at the VGA pixel clock speed of 25 MHz, with our own microcode that doesn't match the timing characteristics of the original 8080, and still get correct game behavior. Compared to the real hardware, our CPU simply wastes more cycles waiting for the next interrupt request.

17.3 Peripherals

We present all peripheral input and the barrel shifter to the CPU as a single, `PortCommand`-controlled component. The only peripheral output would be audio, which is outside the scope of this book.

```
declareBareB [d|
  data Player = MkPlayer
    { pLeft, pRight, pShoot, pStart :: Bool
    }
    deriving (Generic, NFDataX) |]
```

```
peripherals
    :: forall dom. HiddenClockResetEnable dom
    => Signal dom (BitVector 8)
    -> Signal dom Bool
    -> Signal dom Bool
    -> Signal dom (Pure Player)
    -> Signal dom (Pure Player)
    -> Signal dom (Maybe (PortCommand (Index 7) (Unsigned 8)))
    -> Signal dom (Unsigned 8)
```

The total of 18 input bits are mapped to three bytes (read via the first three ports), in a fairly ad-hoc way, with some unused bits set to 0, some to 1:

```
portBytes
    :: BitVector 8
    -> Bool
    -> Bool
    -> Pure Player
    -> Pure Player
    -> (Unsigned 8, Unsigned 8, Unsigned 8)
portBytes sws tilt coin p1 p2 =
    (bitCoerce inp0, bitCoerce inp1, bitCoerce inp2)
  where
    inp0 = (low,   joy p1, high,  high,      high,      sws!4)
    inp1 = (low,   joy p1, high,  pStart p1, pStart p2, coin)
    inp2 = (sws!7, joy p2, sws!6, tilt,      sws!5,     sws!3)

    joy MkPlayer{..} = (pRight, pLeft, pShoot)
```

Writing to port 2 or 4, or reading from port 3, accesses the barrel shifter. The state of our circuit is the `Index 8` of the shift amount and the `BitVector 16` with the shifted 8-bit value. When a new 8-bit value is written to port 4, it is shifted in from the right:

```
startAt
    :: Index 8
    -> BitVector 16
    -> Unsigned 8
startAt offset value = result
  where
    (result, _) = unpack shiftedValue :: (Unsigned 8, Unsigned 8)
    shiftedValue = value `rotateL` fromIntegral offset
```

```
shiftIn
    :: forall n. (KnownNat n)
    => Unsigned n
    -> BitVector (2 * n)
    -> BitVector (2 * n)
shiftIn new old = pack (new, old1)
  where
    (old1, _old2) = unpack old :: (Unsigned n, Unsigned n)
```

The complete implementation of `peripherals` dispatches to the appropriate `portBytes` component, updates the barrel shifter state, or simply returns a zero result:

```
peripherals sws tilt coin p1 p2 cmd =
    mealyStateB (uncurry step) (0, 0) (inputs, cmd)
  where
    inputs = portBytes <$> sws <*> tilt <*> coin <*> p1 <*> p2

    step (inp0, inp1, inp2) cmd = fmap fromJustX $ for cmd $ \case
        ReadPort 0 -> do
            return inp0
        ReadPort 1 -> do
            return inp1
        ReadPort 2 -> do
            return inp2
        ReadPort 3 -> do
            gets $ uncurry startAt
        WritePort 2 x -> do
            _1 .= fromIntegral x
            return 0x00
        WritePort 4 x -> do
            _2 %= shiftIn x
            return 0x00
        _ -> return 0x00
```

17.4 Video

On the original hardware, video output is monochrome 256 × 224 at 60Hz. The straightforward way for us to implement this is to generate a 512 × 448 image (scaled up by two), and center it on a 640 × 480 VGA mode. Of course, if we connect it to a screen, it will display the picture rotated by 90 degrees clockwise; to replicate the original Space Invaders design, we will need to physically rotate the screen into portrait orientation.

It is tempting to instead do the rotation in the video generator: after all, a 90-degree rotation is a simple matter of changing the order in which the video RAM is accessed[1]. However, the video system also acts as a timing source for the CPU, with interrupts generated on lines 96 and 224. The software is well within its rights to use this information to, for example, update only the upper (already drawn) half of the screen in the interrupt handler for line 96, without causing any graphical glitches. But if we change the video system to rotate the screen by 90 degrees, there is no single "line 96" anymore: what used to be a single horizontal line, is now spread out over 256 raster lines.

The interface presented by the video driver to the rest of the system consists of three parts:

- The VGA signal going to the outside world, as always.
- Address and write inputs and read output for accessing the video RAM. These read/write requests are to be served at a lower priority than the internal read requests of the video system.
- An end-of-line signal to be used in the interrupt generator. We will generate a single-period spike at the end of each line containing the line number; so instead of "line 96", it will be "the end of line 95".

```
type VidX = 256
type BufX = VidX `Div` 8
type VidY = 224
type BufX = VidY
type VidSize = BufX * BufY
type VidAddr = Index VidSize

video
    :: (HiddenClockResetEnable Dom25)
    => Signal Dom25 (Maybe VidAddr)
    -> Signal Dom25 (Maybe (Unsigned 8))
    -> ( VGAOut Dom25 8 8 8
       , Signal Dom25 (Maybe (Unsigned 8))
       , Signal Dom25 (Maybe (Index VidY))
       )
```

Similar to previous video drivers, we will use `DSignals` internally to match the VGA sync signals with the delay caused by accessing the video RAM. The external

[1] Or at least, it would be, if we didn't store a whole strip of eight pixels in one byte. In the natural orientation, we can fetch one byte from video RAM and then shift out its bits for 8 consecutive pixels. With rotation applied, pixels next to each other each come from different bytes; this would complicate the video driver, and also give less time for the CPU to access video RAM.

address and write request arguments are lifted directly into the `DSignal` world via `unsafeFromSignal`. On the output side, the current line number is synchronized to the delay of the rest of the video driver.

```
video (unsafeFromSignal -> extAddr) (unsafeFromSignal -> extWrite) =
    ( delayVGA vgaSync rgb
    , toSignal extRead
    , matchDelay rgb Nothing line
    )
  where
    -- Continued below
```

To calculate the video buffer address, we start with a 640×480 physical coordinate, and apply scaling and centering to calculate the following derived coordinates:

- `(bufX, bufY)` is a coordinate in the 32×224 space of video RAM addresses. Because the rows are 32 bytes apart, we can calculate the video RAM address simply by taking the X coordinate as the bottom five bits and the Y coordinate as the rest:

  ```
  toVidAddr :: Index BufX -> Index BufY -> VidAddr
  toVidAddr x y = bitCoerce (y, x)
  ```

- `pixX` is the X coordinate of the current pixel in the current 8-pixel strip.
- `scanline` is the index of the current scan line for the current logical line. Since we are scaling vertically by 2, this will keep changing between the values 0 and 1. Together with `bufY`, we use this to determine the end of a logical line and emit the line number.

We start the implementation of `video` with the definitions corresponding to the above, moving everything into `DSignal`s:

```
    VGADriver{..} = vgaDriver vga640x480at60

    (fromSignal -> bufX, fromSignal -> pixX) =
        scale (SNat @8) . fst .
        scale (SNat @2) . center $
        vgaX
    (fromSignal -> bufY, fromSignal -> scanline) =
        scale (SNat @2) . center $
        vgaY

    bufAddr = liftA2 toVidAddr <$> bufX <*> bufY
```

```
    lineEnd =
        liftD (isFalling False) (isJust <$> bufX) .&&.
        scanline .== Just maxBound
    line = guardA lineEnd bufY
```

The actual pixel values will come from an 8-bit `block` that is updated from the video buffer whenever `bufAddr` changes, and is shifted when `pixX` changes. For now, let's just suppose that we already have a way of getting the internal video buffer read result `intRead` from the internal address `intAddr`; we will revisit this point shortly.

```
    intAddr = guardA (liftD (changed Nothing) bufAddr) bufAddr

    intRead = -- See next section
    extRead = -- See next section

    newPix = delayI False $ liftD (changed Nothing) pixX
    visible = delayI False $ isJust <$> bufAddr
    pixel = enable visible $ liftD2 shifterR intRead newPix
```

Note that we shift `block` to the *right* on every new pixel, and read from its *least* significant bit, unlike we did in the earlier chapters of the CHIP-8 and in the text-mode video driver of our own design. This is simply how the video data is stored on the Space Invaders machine; of course, we have to match the original layout, because the software is written with this assumption as it writes full bytes to the video memory. The implementation of `shifterR` is the same as `shifterL`, with `msb` replaced with `lsb`, and `shiftL` replaced with `shiftR`:

```
shifterR
    :: (BitPack a, HiddenClockResetEnable dom)
    => Signal dom (Maybe a)
    -> Signal dom Bool
    -> Signal dom Bit
shifterR load tick = lsb <$> next
  where
    r = register 0 next

    next = muxA
        [ fmap pack <$> load
        , enable tick $ (`shiftR` 1) <$> r
        ] .|>.
        r
```

At this point, we are at the victory lap: `pixel` indexes into a `palette`, as usual.

```
    rgb = maybe border palette <$> pixel

    border = (0x30, 0x30, 0x30)
    palette 0 = (0x00, 0x00, 0x00)
    palette 1 = (0xff, 0xff, 0xff)
```

17.4.1 Shared memory

Previously, we have encountered several situations where a piece of RAM is shared between the video signal generator and some other system:

- In the TinyBASIC computer, the screen editor takes the incoming stream of characters and turns them into video text buffer writes. Reading is only done by the video system.
- In the CHIP-8, the CPU both reads from and writes to video RAM. We multiplexed reads by prioritizing the video system; but writes always went through directly to video RAM, exploiting the dual-port nature of FPGA block RAM primitives.

Here, we are finally in a good situation to handle shared memory to its fullest extent in a more principled way. The key to this is that our 8080 core, just like the real thing, is prepared for be preempted in its memory access, both for reading and writing. This gives us the opportunity to move completely to a single-port RAM model.

But if the underlying block RAM primitives remain dual port, what is the point of adding a more complicated version of the contention resolution logic? One advantage is that it simply models better the constraints that had to be handled by designs of the era we are interested in. The other is that it generalizes to more than one client with write access.

It is instructive to start from a version of `intRead` / `extRead` that is maximally naïve: using two copies of video RAM, connecting the same write requests to both.

```
wr = liftA2 (,) <$> extAddr <*> extWrite
vidRam = delayedRam (blockRam1 NoClearOnReset (SNat @VidSize) 0)
intRead = Just <$> vidRam (intAddr .<| 0) wr
extRead = Just <$> vidRam (extAddr .<| 0) wr
```

Here, of course, both read results are valid in all cycles; we have no contention, and there is always a read address available (the real address, or 0 otherwise).

The first improvement we can do is to make `intRead` and `extRead` only contain `Just` a value in this cycle if its corresponding address was set in the previous cycle.

```
intRead = enable (delayI False $ isJust <$> intAddr) $
    vidRam (intAddr .<| 0) wr
extRead = enable (delayI False $ isJust <$> extAddr) $
    vidRam (extAddr .<| 0) wr
```

The next step is to remove the redundancy of two copies of `vidRam`. We do this the same way we did it in the CHIP-8: by computing the effective address `addr`, and only enabling `intRead` and `extRead` if the address is theirs.

```
isInt = isJust <$> intAddr
isExt = not <$> isInt .&&. isJust <$> extAddr

addr =
    mux isInt intAddr $
    mux isExt extAddr $
    pure Nothing

read = vidRam (addr .<| 0) wr
intRead = enable (delayI False isInt) read
extRead = enable (delayI False isExt) read
```

This is a deliberately long-winded way of writing the same code as in the CHIP-8, without any benefit so far. However, we can use the information in `isInt` and `isExt` to multiplex the writes as well as the reads.

First of all, to keep ourselves honest, let's change `vidRam` so that it enforces a single-port protocol:

```
singlePort
    :: (Applicative f)
    => (f addr -> f (Maybe (addr, wr)) -> r)
    -> (f addr -> f (Maybe wr) -> r)
singlePort mem addr wr = mem addr (packWrite <$> addr <*> wr)

vidRam = singlePort $ delayedRam $
    blockRam1 NoClearOnReset (SNat @VidSize) 0
```

Now, we can't set the write address independently of the read address. Accordingly, the external write value should only be used when the address is the external one. We could write this as `guardA isExt extWrite`; we opt for the more verbose version to suggest its generalization to multiple write sources.

```
wr = mux isInt (pure Nothing) $
    mux isExt extWrite $
    pure Nothing
```

We implement the generalization of this schema that works for an arbitrary number of requests (passed in priority order) and for any single-port component, be it ROM or RAM:

```
sharedDelayed
    :: (KnownNat k, HiddenClockResetEnable dom)
    => (DSignal dom d (Maybe req) -> DSignal dom (d + k) a)
    -> Vec (n + 1) (DSignal dom d (Maybe req))
    -> Vec (n + 1) (DSignal dom (d + k) (Maybe a))
```

In the ROM case, `req` can be the address directly. In the RAM case, we will choose `req` to be the pair of an `addr` and a write request `Maybe wr`:

```
sharedDelayedRW
    :: (KnownNat k, KnownNat n, HiddenClockResetEnable dom)
    => (DSignal dom d addr ->
        DSignal dom d (Maybe wr) ->
        DSignal dom (d + k) a)
    -> Vec (n + 1) (DSignal dom d (Maybe (addr, Maybe wr)))
    -> Vec (n + 1) (DSignal dom (d + k) (Maybe a))
sharedDelayedRW ram =
    sharedDelayed $ uncurry ram . D.unbundle . (.<| (undefined, Nothing))
```

In the implementation of `sharedDelayed`, we first go through the collection of `requests`, and only enable later ones if all the earlier ones contain `Nothing`. This ensures at most one address at a time is `Just` – the effective `address` is then this single address, and its position also indicates which of the output `reads` should be populated from the actual, memory-originating `read` value.

```
sharedDelayed mem reqs = reads
  where
    addrs = snd $ mapAccumL step (pure True) reqs
      where
        step en addr = (en .&&. isNothing <$> addr, guardA en addr)

    addr = muxA addrs

    read = mem addr
    selectedBy addr = enable (delayI False $ isJust <$>addr)
    reads = map (\addr -> selectedBy addr read) addrs
```

Using this infrastructure, we can now implement the shared video memory in a clean and high-level way:

```
intRead :> extRead :> Nil = sharedDelayedRW ram $
    noWrite intAddr :>
    extAddr `withWrite` extWrite :>
    Nil
  where
    ram = singlePort $ delayedRam (blockRam1 NoClearOnReset (SNat
    @VidSize) 0)
```

This uses the following two utility functions to convert address/write pairs into `sharedDelayed`'s format:

```
withWrite
    :: (Applicative f)
    => f (Maybe addr)
    -> f (Maybe wr)
    -> f (Maybe (addr, Maybe wr))
withWrite = liftA2 $ \addr wr -> (,wr) <$> addr

noWrite
    :: (Applicative f)
    => f (Maybe addr)
    -> f (Maybe (addr, Maybe wr))
noWrite addr = addr `withWrite` pure Nothing
```

17.5 Logic board

We have all the components from our initial schematics, so it is time to put them all together:

- I/O ports `0` to `7` connect to the peripheral driver.
- ROM of 8 kB starts at address `0x0000`.
- The same 1 kB RAM is connected from `0x2000` and `0x4000`. This is called *mirroring*: basically, only the lower 10 bits of the address matter, as long as the top 6 ones match. If we write to `0x2345`, afterwards we can read the same value from `0x4345`.
- The 7 kB address range starting at `0x2400` is connected to the video system.

Furthermore, the currently drawn line is turned into an interrupt request: at the end of lines 95 and 223, we generate a `RST 1` and `RST 2` interrupt, respectively. We have already written the `interruptor` to implement the 8080 interrupt protocol; we can just feed it the reset vector numbers.

```
mainBoard
    :: (HiddenClockResetEnable dom)
    => Signal dom (BitVector 8)
    -> Signal dom Bool
    -> Signal dom Bool
    -> Signal dom (Pure Player)
    -> Signal dom (Pure Player)
    -> Signal dom (Maybe (Unsigned 8))
    -> Signal dom (Maybe (Index VidY))
    -> (Signal dom (Maybe VidAddr), Signal dom (Maybe (Unsigned 8)))
mainBoard sws tilt coin p1 p2 vidRead line = (vidAddr, vidWrite)
  where
    CPUOut{..} = intel8080 CPUIn{..}

    (interruptRequest, rst) =
        interruptor irq (delay False _interruptAck)
    irq =
        mux (line .== Just 95) (pure $ Just 1) $
        mux (line .== Just 223) (pure $ Just 2) $
        pure Nothing

    dataIn = Just 0 |>. dataIn'
    (dataIn', (vidAddr, vidWrite)) =
        $(memoryMap [|_addrOut|] [|_dataOut|] $ do
            rom <- mapH [|Just|] =<<
                romFromFile (SNat @0x2000) [|"SpaceInvaders.bin"|]
            ram <- mapH [|Just|] =<< ram0 (SNat @0x0400)
            io <- mapH [|Just|] =<<
                port_ @(Index 7) [|peripherals sws tilt coin p1 p2|]
            (vid, vidAddr, vidWrite) <- conduit @VidAddr [|vidRead|]

            override [|fmap Just <$> rst|]

            matchJust $ do
                matchLeft @(Unsigned 8) $ do
                    from 0x00 $ connect io
                matchRight @(Unsigned 16) $ do
                    from 0x0000 $ connect rom
                    from 0x2000 $ connect ram
                    from 0x2400 $ connect vid
                    from 0x4000 $ connect ram

            return (vidAddr, vidWrite))
```

The new memory map description primitives used here are:

- `port_`, which is exactly what it says on the tin: a version of `port` without backpane connections:

```
type Port_ dom addr dat =
    Signal dom (Maybe (PortCommand addr dat)) -> Signal dom dat

port_
    :: forall addr' addr. ()
    => ExpQ {- Port_ dom addr dat -}
    -> Addressing addr (Handle addr')
port_ mkPort = readWrite_ $ \addr wr ->
  [| let read = $mkPort $ portFromAddr $addr $wr
     in delay undefined read
  |]
```

- `override`, which takes a `Maybe dat`-valued signal, and replaces the read value with it when it is a `Just`. We use it to feed the `RST` machine code created by the `interruptor` to the CPU.

```
override
    :: ExpQ
    -> Addressing addr ()
override sig = Addressing $ do
    rd <- lift . lift $ newName "rd"
    let decs = [d| $(varP rd) = $sig |]
    tell (decs, mempty, [varE rd])
```

Unlike in the previous chapter, here we need to handle memory access contention. All internal, unshared memory components return a result immediately in the next cycle, which is why they are used via `mapH [|Just|]`. The value from `vidRead`, however, can be `Nothing` when the video controller is fetching the next 8-pixel block.

The top-level circuit then needs to connect `mainBoard` to `video`, and hook up all the external inputs. On an FPGA development board with five directional pushbuttons, we have just enough inputs for one player and the coin deposit. To still support two players, we take a page from the original upright arcade machine, and connect the same buttons to both `p1` and `p2` inputs.

```
topEntity
    :: "CLK_25MHZ" ::: Clock Dom25
    -> "RESET"     ::: Reset Dom25
    -> "SWITCHES"  ::: Signal Dom25 (BitVector 8)
    -> "BTN"       ::: ( "CENTER" ::: Signal Dom25 (Active High)
                       , "UP"     ::: Signal Dom25 (Active High)
                       , "DOWN"   ::: Signal Dom25 (Active High)
                       , "LEFT"   ::: Signal Dom25 (Active High)
                       , "RIGHT"  ::: Signal Dom25 (Active High)
                       )
    -> "VGA"       ::: VGAOut Dom25 8 8 8
topEntity = withEnableGen board
  where
    board sws (c, u, d, l, r) = vga
      where
        tilt = pure False
        coin = fromActive <$> c

        p1 = MkPlayer
            { pLeft  = fromActive <$> l
            , pRight = fromActive <$> r
            , pShoot = fromActive <$> u
            , pStart = fromActive <$> d
            }

        p2 = p1
            { pStart = fromActive <$> l .&&. fromActive <$> r
            }

        (vga, vidRead, line) = video vidAddr vidWrite
        (vidAddr, vidWrite) = mainBoard
            sws tilt coin
            (bundle p1)
            (bundle p2)
            vidRead
            line
```

17.6 Simulation

For simulation purposes, we take the `logicBoard`, and interpret its video write outputs to produce the screen image. Each write goes straight to `varr`, a mutable array of bytes, so that we can later retrieve it into `vidRead`.

```
video
    :: IOArray VidAddr (Unsigned 8)
    -> Maybe VidAddr
    -> Maybe (Unsigned 8)
    -> IO (Maybe (Unsigned 8))
video varr vidAddr vidWrite = for vidAddr $ \addr -> do
    vidRead <- readArray varr addr
    traverse_ (writeArray varr addr) vidWrite
    return vidRead
```

Similar to the CHIP-8, we also write an accompanying function that turns the contents of that array into a pattern by taking apart the eight bits per byte into eight pixels.

```
rasterizeVideoBuf
    :: (MonadIO m)
    => IOArray VidAddr (Unsigned 8)
    -> m (Rasterizer VidX VidY)
rasterizeVideoBuf varr = do
    arr <- liftIO $ freeze varr
    return $ rasterizePattern $ \x y ->
      let (addr, i) = toAddr x y
          block = arr ! addr
      in if testBit block (fromIntegral i) then fg else bg
  where
    toAddr x y = (addr, i)
      where
        (x0, i) = bitCoerce x :: (Index BufX, Index 8)
        addr = bitCoerce (y, x0)

    fg = (0xff, 0xff, 0xff)
    bg = (0x00, 0x00, 0x00)
```

In fact, we can do one better: by simply changing the mapping of the pattern rasterizer's coordinates, we can draw the image with a -90 degree rotation applied, so that it is displayed right-side up. First, we change the rasterizer size in the return type of `rasterizeVideoBuf`:

```
rasterizeVideoBuf
    :: (MonadIO m)
    => IOArray VidAddr (Unsigned 8)
    -> m (Rasterizer VidY VidX)
```

Then, the only term-level change needed is in the computation of the `address`:

```
      let (addr, i) = toAddr (maxBound - y) x
```

The rest of the simulator is a straightforward matter of mapping the SDL keyboard state to Space Invaders input signals. The same keys are mapped to player 1 and player 2's inputs, except for the buttons that start a new one- or two-player game.

```
inputs
    :: (Scancode -> Bool)
    -> (BitVector 8, Bool, Bool, Pure Player, Pure Player)
inputs keyDown = (sws, tilt, coin, p1, p2)
  where
    sws = 0b0000_0000
    tilt = False
    coin = keyDown ScancodeC

    p1 = MkPlayer
        { pLeft = keyDown ScancodeLeft
        , pRight = keyDown ScancodeRight
        , pShoot = keyDown ScancodeLCtrl
        , pStart = keyDown ScancodeReturn
        }

    p2 = p1
        { pStart = keyDown ScancodeLShift
        }
```

In terms of timing, we need to run the simulation of the main board for multiple cycles per frame, making sure interrupts are generated by injecting the correct `line` values. In a real Space Invaders arcade machine running the CPU at 2 MHz, there would be 33 thousand cycles per video frame. In our implementation, since the whole circuit runs at 25 MHz, we would need to run 418 thousand cycles in simulation per frame.

However, the CPU would mostly spend its time busy-waiting for the next interrupt, and we would get horrible simulation performance. Instead, we can run the main board for just 5,000 cycles between interrupts (for a total of 10,000 cycles per frame), and still have everything working. This 5,000 figure is determined empirically; going too much below it means the CPU doesn't get enough time to finish all game state updates due to one interrupt before the next one arrives.

```
main :: IO ()
main = do
    varr <- newArray (minBound, maxBound) 0

    let p0 = MkPlayer False False False False
    sim <- simulateIO_ @System
        (bundle . uncurryN mainBoard . unbundle)
        (0x00, False, False, p0, p0, Nothing, Nothing)

    withMainWindow videoParams $ \events keyDown -> do
        guard $ not $ keyDown ScancodeEscape

        let (sws, tilt, coin, p1, p2) = inputs keyDown
        liftIO $ do
            let run line = sim $ uncurryN $ \ vidAddr vidWrite -> do
                    vidRead <- video varr vidAddr vidWrite
                    return (sws, tilt, coin, p1, p2, vidRead, line)
            replicateM_ 5000 $ run Nothing
            run (Just 95)
            replicateM_ 5000 $ run Nothing
            run (Just maxBound)
        rasterizeVideoBuf varr
  where
    videoParams = MkVideoParams
        { windowTitle = "Space Invaders"
        , screenScale = 4
        , screenRefreshRate = 60
        }
```

Exercises

- Implement screen rotation in the video driver. But wait, haven't we concluded that this is not a good idea because of interrupt timing? We can work around that by completely detaching the rotation from the initial rasterization. The idea is to draw on a virtual 256×224 screen (in reality, a memory buffer) from the video system, in the natural orientation, raising interrupts on lines 96 and 224. A second, real VGA signal generator reads from this buffer and rasterizes its contents in the rotated orientation.

- Implement a virtual simulation of the colored transparent overlays. In the upright cabinet version of Space Invaders, strips of translucent decal was affixed to the screen: red at the top (where the bonus UFO flies across every now and then), and green at the bottom (where the cover buildings are and where the remaining lives are shown). We can implement this by changing the color palette based on the bufX and bufY coordinates.

- Implement PS/2 keyboard input by using keyState of appropriately chosen keys, connected to coin and p1 and p2's signals. Similarly, allow toggling the switches by pressing F1 to F8 .
- Although fully gratuitous, adventurous spirits might want to look into interfacing with accelerometers to implement the tilt input.

17.7 Summary

The great thing about this chapter is how little new code we needed to write to get a playable Space Invaders machine!

- The Space Invaders firmware does all **time-sensitive operations tied to video interrupts**, which freed us from worrying not just about precise timing issues due to differences between our 8080 microcode and a real Intel 8080, but even allowed us to use the VGA pixel clock as our CPU clock, even though that is more than ten times faster than the original Space Invaders hardware's CPU.
- We handle **shared memory** by ordering the address/write inputs by priority, and letting only the highest one go through. Read results are then routed to the single corresponding output signal.
- **External memory-mapped peripherals** can be included in our address decoder infrastructure by exposing a version of the address and the write lines that are restricted to the given address sub-range, as implemented by the conduit combinator.

18 Compucolor II

In this final chapter, our capstone project takes everything we have learned so far, and integrates the components we have developed to produce a replica of the Compucolor II home computer from 1977.

The reason for choosing this machine from the myriad of late-seventies and early-eighties home computers is that it is one of the earlier, cheaper, simpler designs, and at its heart is the same Intel 8080 CPU that we have already examined in detail. On the other hand, it is recognizably a *home computer* fit for consumer use, not a clunky kit computer: it features a keyboard for input, color graphics on a video screen for output, a floppy drive for storage, and a built-in BASIC interpreter for user interaction.

The treasure trove of information on all things Compucolor is the *Compucolor II Tribute* at `https://compucolor.org/`. Beside technical documentation and a software emulator, this website also contains hundreds of downloadable floppy disk images for use once we get our machine working.

18.1 Design

After Intelligent Systems Corporation's first two products, the *Intercolor 8001* and the *Compucolor 8001* terminals, their third product aimed to be a standalone computer that is cheap enough to target the home user market. Its distinguishing features were to be the included floppy drive, the color display supporting both text and graphics, and, staying close to its terminal ancestors, a serial port. The electronics, the floppy drive, and the 13″ color CRT screen was all built into one cabinet, with a separate QWERTY keyboard that came in three different sizes.

Owing to its design goal of good value at low cost, the complete computer is made up of a very small number of components:

- At the core, we find the Intel 8080 CPU with 16 kB of ROM, and 8, 16, or 32 kB of RAM depending on the model.

- The CRT is a standard color TV with the tuner removed. Because the TV was originally designed for the US market, it is refreshed at 60 Hz.
- The video signal is generated by the CRT 5027 video controller, originally designed for text only. As we will see, the graphics mode was achieved with a cleverly designed character set.
- Everything else was handled by a single TMS 5501 I/O controller: keyboard scanning, timers, interrupt generation, and serial communication.

Notably missing from the above list is the floppy drive controller. That is because, in the name of cost-cutting, the Compucolor II floppy drive had no controller logic at all: the stepper motor moving the head between tracks is connected to three bits of the TMS 5501's parallel data port (the same port used for keyboard scanning), and the head itself is connected directly to the serial line. Everything else, most importantly finding the start of sectors, is handled in software. As we will see, this design decision places very stringent timing requirements on our implementation to keep compatibility.

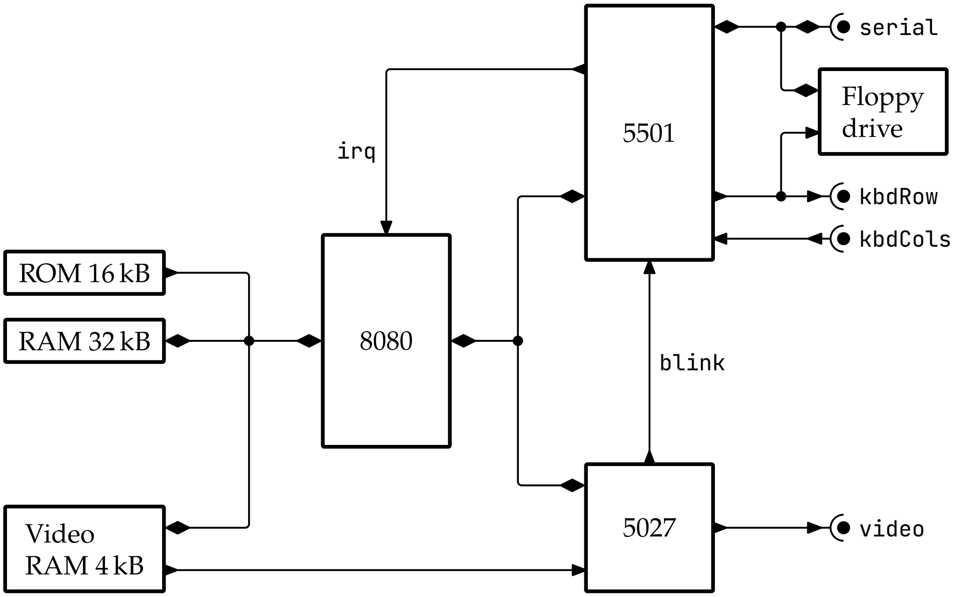
Software-wise, the 16 kB ROM boots directly to a BASIC interpreter; the included *File Control System* can be accessed by pressing Esc followed by the D key. FCS has simple commands for file operations like listing the directory of a disk, or copying files.

The connections between the components are as we would expect them: the TMS 5501 and the control ports of the CRT 5027 are exposed to the CPU as I/O ports, while the actual video contents is stored in 4 kB of RAM shared between the video driver and the CPU. This video RAM is mapped both from `0x6000` and from `0x7000`, with an important distinction: when accessed from `0x7000`, the video signal generator takes precedence, only allowing CPU access during the vertical blanking period; however, when accessed from `0x6000`, the CPU gets priority, at the cost of visual artefacts.

The complexity of this schematic suggests we have our work cut out for us: the components we have to implement are the TMS 5501 and the CRT 5027 peripherals and a virtual floppy drive that can be controlled by the same TMS 5501 operations as the real one.

We could just go through this list, implement each element according to its specification from the original documentation, and then assemble the full computer and hope for the best. Instead, we will chart a way that starts with a minimal viable Compucolor II and iteratively adds more and more components and features until we arrive at our intended fidelity:

1. We start with just the CPU and the memory elements, interpreting video memory writes in simulation as textual output. This will allow us to boot the



original ROM and get all the way to the BASIC prompt, albeit without a way to input anything yet.

2. Next, we will extend this simulator to produce color output and use the original font ROM, by rasterizing into an SDL surface. The purpose of this step is to give a simple software environment to experiment with the intended meaning of the video memory contents.

3. Using our experience with text-based video generators, and taking the software implementation of the previous step as a guideline, we implement the video signal generator part of the CRT 5027, followed by its CPU-controlled, higher-level features.

4. The previous step will leave us with a tantalizing blinking cursor at the BASIC prompt, so the obvious next step is focusing on the parts of the TMS 5501 needed to get keyboard input working: the parallel I/O port, timers, and interrupt generation. At the end of this step, we will be able to type in and run BASIC programs.

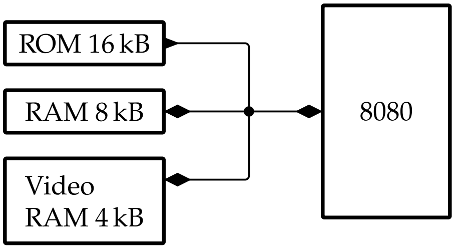
5. As any Haskell programmer would know, *typing* is cool! But *typing in* programs is not so much. We want to load programs from the Compucolor II's existing software library. This will involve extending the TMS 5501 for serial communication, implementing a virtual floppy drive with a virtual magnetic

head sliding over a virtually rotating virtual disk, and a detour into improving the accuracy of our Intel 8080 implementation.

6. The finishing touch will be slowing down the CPU so that games written with the assumption of a 2 MHz processor will not be unplayable with our 40 MHz clock.

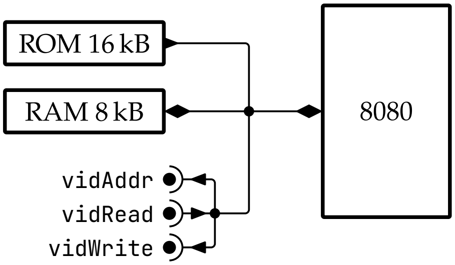
18.2 A Minimal Viable Compucolor II

The simplest possible machine that can still boot the original Compucolor II ROM is a computer with only the CPU, ROM, and RAM. To emphasize that this is a minimal configuration, we have also put only 8 kB of RAM instead of the maximum 32 kB, corresponding to the cheapest model available at Compucolor II's premier.



One problem with this much minimalism is that it also cuts away the *fun* part, i.e. seeing the machine in motion. To see why, just note that this is a completely closed system, with no input/output. From the outside, the only thing this computer would produce is heat.

We work around this problem by *removing even more*: similar to our Space Invaders simulator, we externalize the video RAM so that we can interpret its contents in software:



The corresponding code is a straightforward analogue of the Space Invaders main board without input peripherals and interrupt generators:

```
type VidAddr = Index 4096

mainBoard
    :: (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Unsigned 8))
    -> ( Signal dom (Maybe VidAddr)
       , Signal dom (Maybe (Unsigned 8))
       )
mainBoard vidRead = (vidAddr, vidWrite)
  where
    CPUOut{..} = intel8080 CPUIn{..}
    interruptRequest = pure False

    dataIn = Just 0 |>. dataIn'
    (dataIn', (vidAddr, vidWrite)) =
        $(memoryMap [|_addrOut|] [|_dataOut|] $ do
            rom <- mapH [|Just|] =<<
                romFromFile (SNat @0x4000) [|"compucolor.bin"|]
            ram <- mapH [|Just|] =<< ram0 (SNat @0x2000)
            (vid, vidAddr, vidWrite) <- conduit @VidAddr [|vidRead|]

            matchJust $ matchRight @(Unsinged 16) $ do
                from 0x0000 $ connect rom
                from 0x6000 $ connect vid
                from 0x7000 $ connect vid
                from 0x8000 $ connect ram

            return (vidAddr, vidWrite))
```

We have mapped the 4 kB video RAM both from `0x6000` and `0x7000`, but there is no way to distinguish between the two access paths yet. For the high-level simulation interpreting the contents of the video memory, the distinction doesn't matter, since all read and write access can go through immediately. Of course, when we get to implementing a real hardware video signal generator, we will have to return to this point.

But enough worrying about the future, let's take what we have and boot it up! Once again, we turn to the *Terminal* package to take care of single character positioning. As usual, we create an `IOArray` to store the video memory contents; then, once every couple of cycles, we update our output based on the array. We pick the number of cycles, 20,000, mostly arbitrarily: too small a number would mean we waste a lot of time updating the screen without giving the CPU the opportunity to do anything; too large, and the simulator would be non-responsive.

```
mkSim :: IO (IOArray VidAddr (Unsigned 8), IO ())
mkSim = do
    vidRam <- newArray (minBound, maxBound) 0x00
    sim <- simulateIO_ @System (bundle . mainBoard) Nothing
    let step = do
            sim $ \(vidAddr, vidWrite) -> for vidAddr $ \addr -> do
                x <- readArray vidRam addr
                traverse_ (writeArray vidRam addr) vidWrite
                return x
    return (vidRam, step)

main :: IO ()
main = do
    (vidRam, step) <- mkSim

    withTerminal $ runTerminalT $ forever $ do
        eraseInDisplay EraseAll
        hideCursor
        setAutoWrap False

        liftIO $ replicateM_ 20000 step
        putScreen vidRam
```

The split between `main` and `mkSim` is in preparation for the later, non-text-based simulators. For the text-based simulator, the heavy lifting will happen in `putScreen`. Of course, first we must understand how the Compucolor II uses its video memory.

The screen is rendered as a 64 × 32 grid of characters, each character drawn with a 6 × 8 resolution font, for a total pixel resolution of 384 × 256. Each character is stored as two bytes: one for the character itself, and another one for its attributes, such as blinking, or the foreground and background color.

One innovation of the Compucolor II was emulating a 128 × 128 graphics mode with the same text mode video circuitry. This is achieved by using one bit of the attribute byte to switch between a normal text font, and a special font ROM that maps each 8-bit character to a 2 × 4 bitmap, stretched to 6 × 8. For now, we will just assume that every character is in text mode. This assumption holds for the boot-up screen, and since we don't have inputs hooked up yet, we can't proceed from here yet anyway.

The 64 × 32 grid gives a total of 2048 characters. Using two bytes for each character neatly explains the video memory size instead of the previous seemingly magic number:

```
type TextWidth = 64
type TextHeight = 32
type TextSize = TextWidth * TextHeight
type VidSize = TextSize * 2
type VidAddr = Index VidSize
```

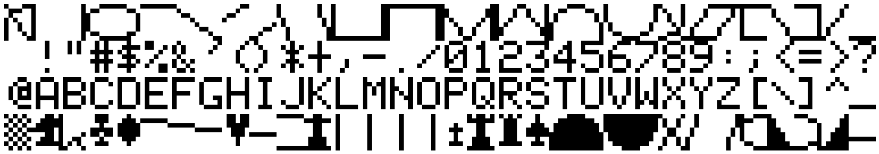
Now we are ready to start writing `putScreen`. To put the contents of the video RAM on the text screen, we go through each line, move the terminal cursor to the start of the line, and then write each character with `putCharAt`, passing it the coordinates:

```
putScreen
    :: forall m. (MonadIO m, MonadScreen m)
    => IOArray VidAddr (Unsigned 8)
    -> m ()
putScreen vidRam = do
    for_ [minBound..maxBound] $ \y -> do
        setCursorPosition $ Position (fromIntegral y) 0
        for_ [minBound..maxBound] $ \x -> do
            putCharAt x y
    flush
  where
    -- putCharAt defined below
```

The character to be displayed at (x, y) is stored in the byte `b` stored at the even address $2 \cdot (y \cdot w + x)$, truncated to 7 bits. Instead of calculating the address arithmetically, we do it via `bitCoerce`, in preparation for the eventual hardware implementation:

```
    putCharAt :: Index TextWidth -> Index TextHeight -> m ()
    putCharAt x y = do
        let addr = bitCoerce (y, x, (0 :: Unsigned 1))
        b <- liftIO $ readArray vidRam addr
        let (isTall, c) = bitCoerce b
        putCharCC $ if tall && odd y then 0x20 else c

    putCharCC :: Unsigned 7 -> m ()
    putCharCC = putChar . chr . fromIntegral
```

The lucky break with `putCharCC` is that the Compucolor, internally, used an ASCII-compatible text encoding for punctuation, numbers, and uppercase letters, so we can write `putCharCC` with just a bit of type scaffolding around `putChar`. The rest of the 7-bit ASCII range, most notably, what should be the lowercase letters, are mapped to various arcs and diagonals for drawing, as well as card and chess symbols to be used by games:



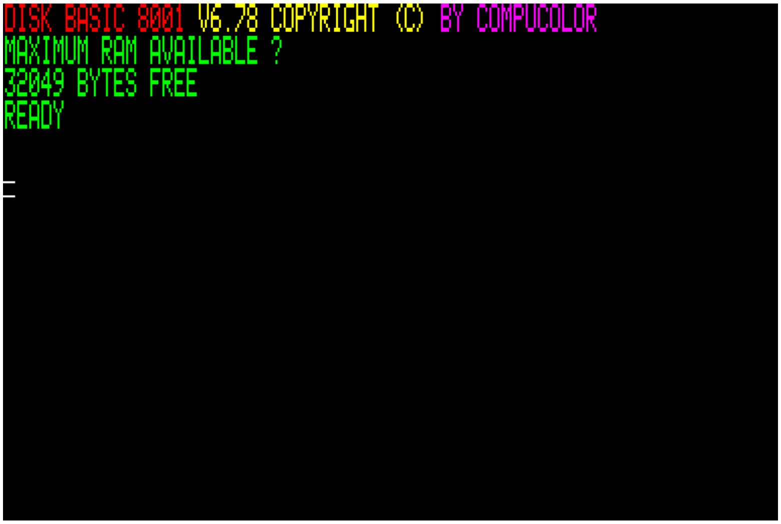
And there we have it: our simplified Compucolor II is ready to be turned on. Running the simulator, first we see some garbage on the screen, corresponding to the `0x00` characters in the empty video RAM. These characters are then quickly overwritten with `0x20` (i.e. space) characters by the initialization routines of the Compucolor II ROM. Depending on the performance of the computer running the simulation, in 10-20 seconds we are finally greeted with the following output:

```
DISK BASIC 8001 V6.78 COPYRIGHT (C) BY COMPUCOLOR
DISK BASIC 8001 V6.78 COPYRIGHT (C) BY COMPUCOLOR
MAXIMUM RAM AVAILABLE ?
MAXIMUM RAM AVAILABLE ?
7473 BYTES FREE
7473 BYTES FREE
READY
READY
```

18.2.1 What's what's with with the the stutter stutter?

The lights are on, someone is definitely at home! But why the weird duplication of each line? According to our addressing scheme, one character below another is 128 cells away; why are we reading the same byte from the video RAM for these two addresses?

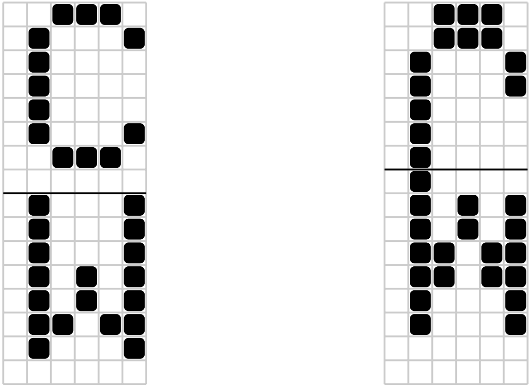
The answer lies in the topmost bit of each character that we have so carelessly discarded. This bit denotes the so-called *tall mode*: a character that should be rendered at double height. Let's look at a screenshot of the Compucolor right after booting:



```
DISK BASIC 8001 V6.78 COPYRIGHT (C) BY COMPUCOLOR
MAXIMUM RAM AVAILABLE ?
32049 BYTES FREE
READY
```

Measuring the glyph sizes against the visible area, we can see that the character resolution comes out to 64 × 16, i.e. vertically just half of what it should be. Each character has its highest bit set, turning on tall mode.

Rasterization still works in single-character blocks of 6 × 8 pixels, so each tall character needs to be made up of a top 6 × 8 block, and a bottom 6 × 8 block. On even rows, the top half of the given character is rendered, duplicating each line; on odd rows, the bottom half. It is the responsibility of the software to ensure that the top and the bottom halves match; for example, it is perfectly possible to put the characters `C` at $(0, 0)$ and `W` at $(0, 1)$. If we then turn on the tall bit on both of them, we get the top half of the `C` stretched vertically, on top of the bottom half of `W`, also stretched. The next picture shows these two rendering modes side by side, with the tall bit turned on on the right-hand side.



We have no good way of replicating that behavior until we go beyond text terminal output; for now, we will simply skip tall characters on odd lines:

```
    putCharAt x y = do
        let addr = bitCoerce (y, x, (0 :: Unsigned 1))
        b <- liftIO $ readArray vidRam addr
        let (isTall, c) = bitCoerce b
        putCharCC $ if isTall && odd y then 0x20 else c
```

18.2.2 Putting the Color in Compucolor

Customers would have rightly felt that they only got half their money's worth if the Compucolor would be a Computer with no Color. Indeed, six bits of the attribute byte specify the background and foreground color of each character block.

The change to `putCharAt` is to simply read both the character byte `b0` (from `addr`) and the attribute byte `b1` (from `addr + 1`), then take the latter apart. The eight bits of the attribute byte are as follows:

- The topmost bit selects "plot mode", which is the name used in the Compucolor documentation for the 128 × 128 graphics mode. For now, we will assume all blocks are in character mode.
- The next bit turns on blinking. Blinking blocks switch between normal foreground and black foreground once every 16 frames. Just like plot mode, we leave this to later, when we use SDL for output.
- Bits 5 to 3 specify the background color, in BGR order.
- Bits 2 to 0 is the foreground color.

```
attributes
    :: Unsigned 8 -> (Bool, Bool, (Bit, Bit, Bit), (Bit, Bit, Bit))
attributes = bitCoerce
```

Although not relevant for now, since we skip the bottom half of tall characters, it should be noted that just like character bytes, attribute bytes can also differ between the top and the bottom half of a tall character: we can combine the top half of a red on black `A` and the bottom half of a yellow on blue blinking `Y`.

```
    putCharAt :: Index TextWidth -> Index TextHeight -> m ()
    putCharAt x y = do
        let addr = bitCoerce (y, x, (0 :: Unsigned 1))
        b0 <- liftIO $ readArray vidRam addr
        b1 <- liftIO $ readArray vidRam (addr + 1)
        let (isTall, c) = bitCoerce b0
            (isPlot, blink, back, fore) = attributes b1
        setAttribute $ background $ bright $ toColor back
        setAttribute $ foreground $ bright $ toColor fore
        putCharCC $ if isTall && odd y then 0x20 else c
```

We use Terminal's `bright` combinator on both the background and the foreground color as a stylistic choice, to get a kind of washed-out look.

Because text terminals don't usually give the full freedom of arbitrary colors, the Terminal library exposes a finite set of predefined colors that we have to use. This makes `toColor` a big enumeration of all eight possible colors instead of computing the red, green, and blue channels separately:

```
    toColor :: (Bit, Bit, Bit) -> Color m
    toColor (0,0,0) = black
    toColor (0,0,1) = red
    toColor (0,1,0) = green
    toColor (0,1,1) = yellow
    toColor (1,0,0) = blue
    toColor (1,0,1) = magenta
    toColor (1,1,0) = cyan
    toColor (1,1,1) = white
```

This concludes our text terminal-based simulation: it is time to aim higher!

18.3 Detailed rendering with SDL

The purpose of this section is to familiarize ourselves further with the video system, including the details of plot mode. Our `main` function replaces Terminal-based I/O with SDL. We also need to load the Compucolor's original font image into an immutable array, which we will use in `renderScreen` to draw each (non-plot) block.

```
main :: IO ()
main = do
    fontBS <- fmap bitCoerce . BS.unpack <$> BS.readFile "font.img"
    fontArr <- newArray @IOArray (minBound, maxBound) 0
    zipWithM_ (writeArray fontArr) [0..] bs
    fontRom <- freeze fontArr

    (vidRam, step) <- mkSim
    withMainWindow videoParams $ \events keyDown -> do
        guard $ not $ keyDown ScancodeEscape

        liftIO $ replicateM_ 20000 step
        lift $ renderScreen fontRom vidRam

videoParams :: VideoParams
videoParams = MkVideoParams
    { windowTitle = "Compucolor II"
    , screenScale = 4
    , screenRefreshRate = 60
    }
```

In `renderScreen`, we freeze the `vidRam`'s contents, and create a rasterizer sized $(64 \cdot 6) \times (32 \cdot 8)$. The first half of the rasterizer is a direct analogue of the text-based one:

```
type FontWidth = 6
type FontHeight = 8

renderScreen
    :: Array (Index 1024) (BitVector 8)
    -> IOArray VidAddr (Unsigned 8)
    -> IO (Rasterizer (TextWidth * FontWidth) (TextHeight * FontHeight))
renderScreen fontRom vidRam = do
    vidRam <- freeze vidRam
    return $ rasterizePattern $ \x y ->
      let (x1, x0) = divI (SNat @FontWidth) x
          (y1, y0) = divI (SNat @FontHeight) y

          charAddr = bitCoerce (y1, x1, (0 :: Unsigned 1))
          char = vidRam ! charAddr
          attr = vidRam ! (charAddr + 1)
          (isTall, c) = bitCoerce char :: (Bool, Unsigned 7)
          (isPlot, blink, back, fore) = attributes attr

          pixel = _ -- see below
      in fromBGR $ if pixel then fore else back
```

Here, `divI` does division in the `Index` type, acting as the “backwards” software equivalent of the “forwards” counter used inside the `scale` pattern combinator:

```
divI
    :: (KnownNat n, KnownNat k, 1 <= k, n ~ ((n `Div` k) * k))
    => SNat k
    -> Index n
    -> (Index (n `Div` k), Index k)
divI k@SNat x = (fromIntegral x1, fromInegral x0)
  where
    (x1, x0) = x `quotRem` snatToNum k
```

`fromBGR` transforms our three-bit BGR value into the RGB format required by `rasterizePattern`, by `stretch`ing each color component into the full range of its result. We write it polymorphically over the result type's color components, so that it can be reused for the hardware implementation.

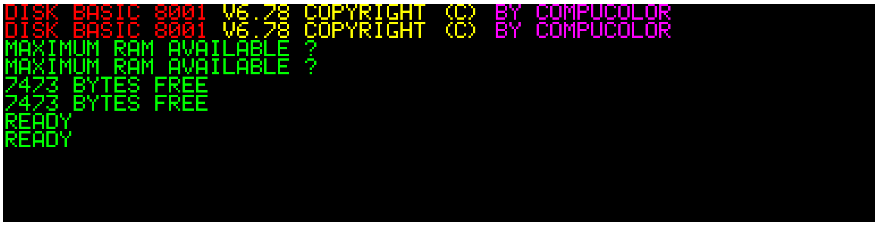
```
fromBGR
    :: (Bounded r, Bounded g, Bounded b)
    => (Bit, Bit, Bit) -> (r, g, b)
fromBGR (b, g, r) = (stretch r, stretch g, stretch b)
  where
    stretch 0 = minBound
    stretch 1 = maxBound
```

All that remains is calculating whether the `pixel` at (x, y) should be set (i.e. in the foreground color), or unset (i.e. in the background color). Without worrying about the `isTall` and `isPlot` flags, our first version simply looks up the current sub-row in the `fontRom`, and selects the bit corresponding sub-column.

The font ROM contains the y_0-th row of character c at address $8c + y_0$, which is of course ripe for a `bitCoerce`-based implementation. This gives us 8 bits per line, of which only the top 6 bits are used, most significant bit first. The implementation is supposed to evoke `shifterL`, suggesting a shift register-based hardware implementation later:

```
pixel = fontPixel

fontPixel = bitToBool $ msb $ glyphRow `shiftL` fromIntegral x0
  where
    glyphAddr = bitCoerce (c, y0)
    glyphRow = fontRom ! glyphAddr
```

This definition of `pixel` gets us to feature parity with the text-based simulator, with the added benefit of using the real Compucolor font set to render each glyph:



```
DISK BASIC 8001 V6.78 COPYRIGHT (C) BY COMPUCOLOR
DISK BASIC 8001 V6.78 COPYRIGHT (C) BY COMPUCOLOR
MAXIMUM RAM AVAILABLE ?
MAXIMUM RAM AVAILABLE ?
7473 BYTES FREE
7473 BYTES FREE
READY
READY
```

Now it's time to move beyond, by implementing the `isTall` and `isPlot` flags. The former is a simple matter of coordinate transformation: a tall character on line `y0` is rendered as if it was on line `half y0` (for the top half) or `half y0 + 4` (for the bottom half).

```haskell
fontPixel = bitToBool $ msb $ glyphRow `shiftL` fromIntegral x0
  where
    y0'
      | isTall = half y0 + if odd y1 then 4 else 0
      | otherwise = y0
    glyphAddr = bitCoerce (c, y0')
    glyphRow = fontRom ! glyphAddr
```

We can also do this without arithmetic, with mere bit manipulation: observe that `half y0` is just `y0` with its lowest bit dropped, and the addition corresponds to adding a new topmost bit:

```haskell
toTall :: Index TextHeight -> Index FontHeight -> Index FontHeight
toTall y1 y0 = bitCoerce (lsb y1, halfIndex y0)

fontPixel = bitToBool $ msb $ glyphRow `shiftL` fromIntegral x0
  where
    y0' = if isTall then toTall y1 y0 else y0
    glyphAddr = bitCoerce (c, y0')
    glyphRow = fontRom ! glyphAddr
```

This requires `halfIndex`, a function that drops the lowest bit of an `Index`. All the work goes into the type signature; of course, when used with concrete types, these constraints are solved for us behind the scenes.

```haskell
halfIndex
    :: (KnownNat n, 1 <= (2 * n), (CLog 2 (2 * n)) ~ (CLog 2 n + 1))
    => Index (2 * n)
    -> Index n
halfIndex = fst . bitCoerce @_ @(_, Bit)
```

To implement `isPlot`, we have to first understand the layout of the bits in `char`: the eight bits `76543210` are interpreted as the following 6 × 8 pattern:

```
000444
000444
111555
111555
222666
222666
333777
333777
```

Of course, this is just the following 2 × 4 pattern scaled up:

```
04
15
26
37
```

We can handle this “arithmetically”, as follows:

```
pixel = if isPlot then plotPixel else fontPixel

plotPixel = testBit char (fromIntegral idx)
  where
    idx = half y0 + if x0 < 3 then 0 else 4
```

However, if implemented in hardware, this would require a barrel shifter to dynamically select the `idx`-th bit of `char`. An alternative implementation that lends itself more to hardware, is to instead create a small ROM containing the mapping of bytes to 2 × 4 patterns, and then stretch that vertically and horizontally to the full 6 × 8 blocks. In fact, we can factor those 2 × 4 patterns into two 1 × 4 patterns, since the left-hand side only depends on the lower four bits, while the right-hand side only uses the upper half. `splitChar` will be used instead of straight `bitCoerce` to aid type inference.

```
splitChar :: Unsigned 8 -> (Index 16, Index 16)
splitChar = bitCoerce

stretchRow :: Bit -> Bit -> BitVector 8
stretchRow b1 b2 = bitCoerce $
    replicate (SNat @3) b1 ++
    replicate (SNat @3) b2 ++
    repeat 0
```

The idea is that to draw a given plot character `c`, we first split it into two halves `hi` and `lo`, then look up the bit for the current row in each, and then stretch the two bits to a full 8-bit row in the same format as `glyphRow`.

So what are the four rows corresponding to a plot half-byte? Looking at the left half of our diagram from earlier, the half-byte $b_3b_2b_1b_0$ should be rendered with b_0 on the first row, b_1 on the second, and so on (before stretching). But that is just the bits of the half-byte itself, in reverse order!

```
rowsOf :: Index 16 -> Vec 4 Bit
rowsOf = reverse . bitCoerce
```

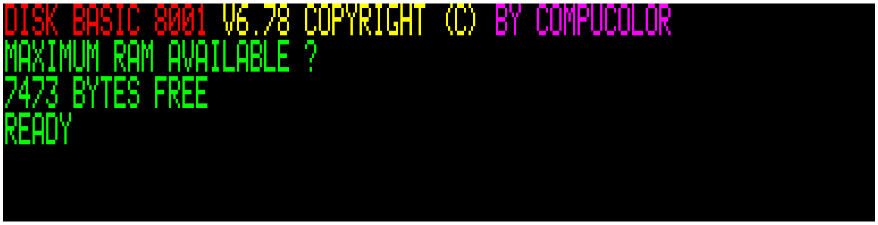
We can populate our full plotting ROM by putting each half-byte's pattern one after the other, with an appropriate addressing scheme. Note the careful choice of `Index` vs. `Unsigned` types here: `rowsOf` works on `Index` so that it can be applied on `indicesI` in the definition of `plots`; while `plotAddr` returns the encoded ROM address as an `Unsigned 6` because later, in the hardware implementation, that will allow us to connect it directly to the address port of `rom`.

```
plots :: Vec (16 * 4) Bit
plots = concatMap rowsOf indicesI

plotAddr :: Index 16 -> Index 4 -> Unsigned 6
plotAddr halfChar row = bitCoerce (halfChar, row)
```

After this detour that will pay dividends in the hardware video generator, let's get back to the problem at hand and define `pixel` using a single shift register shared between the `fontBlock` and the `plotBlock`:

```
pixel = bitToBool $ msb $ block `shiftL` fromIntegral x0
block = if isPlot then plotBlock else fontBlock

plotBlock = stretchRow b1 b2
  where
    (char2, char1) = splitChar char
    b1 = plots !! plotAddr char1 (halfIndex y0)
    b2 = plots !! plotAddr char2 (halfIndex y0)

fontBlock = fontRom ! glyphAddr
  where
    y0' = if isTall then toTall y1 y0 else y0
    glyphAddr = bitCoerce (c, y0')
```

Put together, this version implements all the "static" features of the video system: glyph rendering, color support, and plot characters:



```
DISK BASIC 8001 V6.78 COPYRIGHT (C) BY COMPUCOLOR
MAXIMUM RAM AVAILABLE ?
7473 BYTES FREE
READY
```

The missing dynamic features are those controlled by the CPU through the use of the CRT 5027's ports: vertical scrolling and cursor placement. We will now take the understanding of the Compucolor's video system that we earned by writing this software model, move on to a hardware implementation, and add these features afterwards.

18.4 Video hardware

The first decision we have to make is what VGA mode to use for the video output. On the real Compucolor, have $64 \times 6 = 384$ pixels horizontally and $32 \cdot 8 = 256$ pixels vertically, refreshed at 60 Hz. Scaling it up by two gives us 768×512 which fits neatly into the 800×600 VGA mode. Looking up the timing specification of the 800×600@60 mode, we get the following numbers and sync levels:

```
-- | VGA 800x600@60Hz, 40 MHz pixel clock
vga800x600at60 :: VGATimings (HzToPeriod 40_000_000) 800 600
vga800x600at60 = VGATimings
    { vgaHorizTiming = VGATiming High (SNat @40) (SNat @128) (SNat @88)
    , vgaVertTiming  = VGATiming High (SNat @1)  (SNat @4)   (SNat @23)
    }
```

This means we'll need a 40 MHz clock. As before, we will use a single clock domain for our whole computer, so the CPU and everything else will also run at 40 MHz in `Dom40`:

```
-- | 40 MHz clock, needed for the VGA mode we use.
createDomain vSystem{vName="Dom40", vPeriod = hzToPeriod 40_000_000}
```

18.4.1 Back to the basics

We start with taking the output-only `mainBoard`, and implementing our first minimal renderer in hardware. This gives us a `topEntity` that we have seen a million times already:

```
topEntity
    :: "CLK_40MHZ" ::: Clock Dom40
    -> "RESET"     ::: Reset Dom40
    -> "VGA"       ::: VGAOut Dom40 8 8 8
topEntity = withEnableGen board
  where
    board = vga
      where
        (vidAddr, vidWrite) = mainBoard vidRead
        (vga, vidRead) = video vidAddr vidWrite
```

The type of the `video` signal generator also follows the previously established patterns:

```
video
    :: (HiddenClockResetEnable Dom40)
    => Signal Dom40 (Maybe VidAddr)
    -> Signal Dom40 (Maybe (Unsigned 8))
    -> ( VGAOut Dom40 8 8 8
       , Signal Dom40 (Maybe (Unsigned 8))
       )
```

In the previous section, the simplest possible software renderer was one that assumed all characters should be rasterized using the font ROM, in black and white, without tall mode. But that sounds a lot like the Tiny BASIC video generator! The only difference is that the video RAM is now shared with the rest of the system, requiring access arbitration.

```
video (unsafeFromSignal -> extAddr) (unsafeFromSignal -> extWrite) =
    ( delayVGA vgaSync rgb
    , toSignal extRead
    )
  where -- Continued below
```

We start with generating the 800 × 600 VGA timers, and transforming that into $(64, 6) \times (32, 8)$ coordinates via scaling and centering:

```
VGADriver{..} = vgaDriver vga800x600at60
(fromSignal -> x1, fromSignal -> x0) =
    scale (SNat @FontWidth) .
    fst . scale (SNat @2) .
    center $
    vgaX
(fromSignal -> y1, fromSignal -> y0) =
    scale (SNat @FontHeight) .
    fst . scale (SNat @2) .
    center $
    vgaY
```

Here, we use the names `x1` and `x0` consistently with the software implementation, to make it easier to see the correspondence. Our goal at this point is to compute the current pixel's `rgb` value, which, for now, we can do if we know whether the current `pixel` should be set or not. Alternatively, outside the visible area, we render a `border`.

```
isBorder = not <$> (isJust <$> x1 .&&. isJust <$> y1)

rgb = mux (delayI True isBorder) (pure border) $
    palette !!. pixel
  where
    border = (0x30, 0x30, 0x30)

    palette =
        (0x00, 0x00, 0x00) :>
        (0xff, 0xff, 0xff) :>
        Nil
```

As before, we will use a left-shifter to compute the current `pixel`, loading the next `block` whenever we get to a new character; otherwise, we shift to the next pixel when a new logical column comes up:

```
newChar = liftD (changed Nothing) x1
newCol = liftD (changed Nothing) x0

pixel = liftD2 shifterL block (delayI False newCol)
```

The next `block` to load can be computed quite easily: `x1` and `y1` give us the address of the new character, which we can use to index into the video RAM. In this version, we give priority to reads originating from the video driver, as before:

```
    charAddr = guardA newChar $ liftA2 toAddr <$> x1 <*> y1
      where
        toAddr :: Index TextWidth -> Index TextHeight -> VidAddr
        toAddr x1 y1 = bitCoerce (y1, x1, (0 :: Unsigned 1))

    charRead :> extRead :> Nil = sharedDelayedRW ram $
        noWrite charAddr :>
        extAddr `withWrite` extWrite :>
        Nil
      where
        ram = singlePort $ delayedRam $
            blockRam1 NoClearOnReset (SNat @VidSize) 0
```

The lower 7 bits of `charRead` form one part of the address into the font; the rest comes from the current intra-character row, i.e. `y0`.

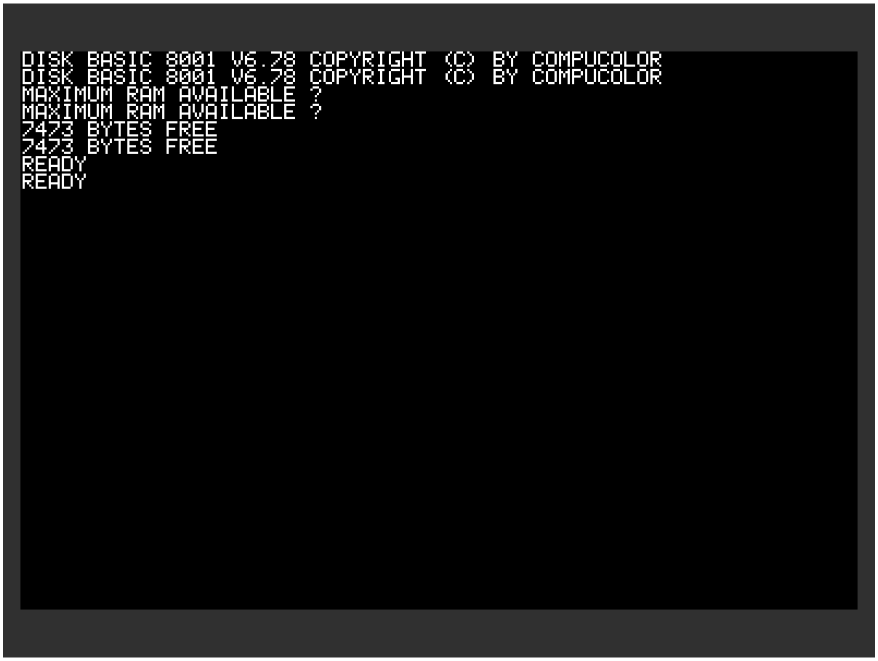
```
    fontAddr = fmap resize <$> charRead
    block = enable (delayI False newChar) $
        fontRom (0 |>. fontAddr) (0 |>. delayI Nothing y0)
```

The change in `fontRom` is that instead of an `Array`, we put the glyph definitions into a small ROM, justifying the choice of name:

```
fontRom
    :: (HiddenClockResetEnable dom)
    => DSignal dom n (Unsigned 7)
    -> DSignal dom n (Index FontHeight)
    -> DSignal dom (n + 1) (BitVector 8)
fontRom char row = delayedRom (romFilePow2 "font.bin") $
    bitCoerce <$> D.bundle (char, row)
```

As we can see, the complete code is a mix between the Tiny BASIC text-mode video generator and the way video memory is shared in Space Invaders.

Of course, since we have only done the bare minimum to get something on the screen, we also have the same problems as some of our earlier implementations: namely, output is in black & white, with tall characters rendered twice:



```
DISK BASIC 8001 V6.78 COPYRIGHT (C) BY COMPUCOLOR
DISK BASIC 8001 V6.78 COPYRIGHT (C) BY COMPUCOLOR
MAXIMUM RAM AVAILABLE ?
MAXIMUM RAM AVAILABLE ?
2473 BYTES FREE
2473 BYTES FREE
READY
READY
```

What we *can't* see, though, is another missing feature that we should address before moving on: allowing the CPU to access the video memory "urgently".

18.4.2 Accessing memory, fast and slow

In general, video signal generators need priority access to video RAM because the CRT waits for no one: even if the next pixel's color is not ready in time, the electron beam will keep sweeping. However, this isn't necessarily the end of the world: it just means parts of the screen, those corresponding to when the video system was "too late", will appear garbled.

The Compucolor II allows the programmer to make this decision: when accessing the video memory from the CPU, is it more important to get it done quickly and keep running the program, even at the cost of potential visual artefacts? If the i^{th} video RAM cell is accessed via the address $6000_{16} + i$, the read/write is instantaneous. If, on the other hand, the address $7000_{16} + i$ is used, the CPU will wait for the next horizontal blanking period (i.e. the time between the end of the current raster line's visible portion and the start of the next one).

To implement this, first of all we need to know which address region was used to access the video RAM. We `tag` each region with a Boolean flag marking urgency, by changing the lines of `mainBoard` marked with `(*)` below:

```
mainBoard
    :: (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Unsigned 8))
    -> ( Signal dom (Maybe (Bool, VidAddr))  -- (*)
       , Signal dom (Maybe (Unsigned 8))
       )
mainBoard vidRead = (vidAddr, vidWrite)
  where
    CPUOut{..} = intel8080 CPUIn{..}

    interruptRequest = pure False

    dataIn = Just 0 |>. dataIn'
    (dataIn, (vidAddr, vidWrite)) =
        $(memoryMap [|_addrOut|] [|_dataOut|] $ do
            rom <- mapH [|Just|] =<<
                romFromFile (SNat @0x4000) [|"compucolor.bin"|]
            ram <- mapH [|Just|] =<< ram0 (SNat @0x2000)
            (vid, vidAddr, vidWrite) <-
                conduit @(Bool, VidAddr) [|vidRead|] -- (*)

            matchJust $ matchRight @(Unsinged 16) $ do
                from 0x0000 $ connect rom
                from 0x6000 $ tag True $ connect vid   -- (*)
                from 0x7000 $ tag False $ connect vid  -- (*)
                from 0x8000 $ connect ram

            return (vidAddr, vidWrite))
```

We can `tag` an address with extra data by writing a new matcher that always succeeds:

```
tag
    :: (Lift addr')
    => addr'
    -> Addressing (addr', addr) a
    -> Addressing addr a
tag t = matchAddr [| \addr -> Just (t, addr) |]
```

The corresponding change in the `video` generator is in how the `extAddr` is mixed with the `charAddr`. We `schedule` the external address based on urgency and availability, where the latter is defined by `hblank`:

```
hblank = isNothing <$> x1

extAddr' = schedule <$> hblank <*> extAddr
  where
    schedule hblank extAddr = do
        (urgent, addr) <- extAddr
        guard $ urgent || hblank
        return addr
```

Since it is now `schedule`'s sole responsibility to decide if the address from the external (i.e. CPU) source should get priority, we also need to swap the order of `extAddr'` and `charAddr` as passed to `sharedDelayedRW`; otherwise, `extAddr'` would have no chance to preempt `charAddr`.

```
extRead :> charRead :> Nil = sharedDelayedRW ram $
    extAddr' `withWrite` extWrite :>
    noWrite charAddr :>
    Nil
```

18.4.3 Tall mode

Implementing tall mode is a straightforward change to the second argument passed to `fontRom`. Instead of discarding the topmost bit of `charRead`, we use that to dispatch between `y0` and `toTall y1 y0`. The only further complication is handling the `Maybe`-ness of `y0` and `y1` outside the visible area:

```
char = charRead .<| 0
(isTall, fontAddr) = D.unbundle $ bitCoerce <$> char

y0' = mux isTall tall short .<| 0
  where
    short = delayI Nothing y0
    tall = delayI Nothing $ liftA2 toTall <$> y1 <*> y0

-- Use y0' instead of y0
block = enable (delayI False newChar) $ fontRom fontAddr y0'
```

18.4.4 Attributes and plot mode

The big complication in supporting attributes is making them available at the same time as `char`, even though we can only access one byte at a time from the video RAM. Some possible solutions to this would be:

- Have two copies of the video RAM, fan out writes, and read from both `charAddr` and `charAddr + 1` at the same time. While this would be a horrible waste of resources, it is worth keeping in mind as it can give us a hint to the eventual solution we will settle on.

- Change the video RAM layout so that each cell contains 16 bits, i.e. a full character-attribute pair. This would make it easy on the internal memory access, but since the CPU-originating external access is single-byte-based, we would need some extra logic (and cycle!) to implement the writing logic of reading out both bytes and then changing just one of them before writing it back.

- Schedule character and attribute byte access so that they happen in sequence. Each line of a character is drawn for 6 pixels over 12 cycles (because of scaling), so we could easily fit 3 cycles of memory access into that:

  1. Read `char` from `charAddr`
  2. Read `fontBlock` from `fontRom` and `plotBlock` from `plotRom` using `char` as the address, and at the same time, read `attr` from `charAddr + 1`
  3. Based on the contents of `attr`, load `block` from either `fontBlock` or `plotBlock`

  The downside of this approach is the engineering complexity: it is hard to keep everything in order when there is no single definite delay we can assign to the result of reading from the video RAM. To see why, consider that `char` has delay 1, but `attr` would have delay 2, even though both are the results of reading from the same RAM.

- If two copies of the video RAM is wasteful, what about two half-copies? This idea is based on the observation that if we went with our first idea, the video signal generator would always read from even addresses from one of the copies, and from the odd addresses for the other.

So in this last approach, which is what we are going with, we have 2 kB of character RAM and 2 kB of attribute RAM, addressed separately. For CPU-originating read- or write-requests, we can look at the bottom bit of the address to determine which of the two RAM to route it to. Of course, this kind of unbraiding an address generalizes to any number of lower bits to dispatch on:

```
unbraid
    :: (KnownNat n, KnownNat k, _)
    => Maybe (Index (n * 2 ^ k))
    -> Vec (2 ^ k) (Maybe (Index n))
```

```
unbraid Nothing = repeat Nothing
unbraid (Just addr) = map (\k -> addr' <$ guard (sel == k)) indicesI
  where
    (addr', sel) = bitCoerce addr
```

We use `PartialTypeSignatures` here to omit a lot of arithmetic tautologies (such as `(CLog 2 (2 ^ k)) ~ k`); as usual, these constraints are trivially proven behind the scenes for concrete monomorphic applications, so we will never have to interact with them.

Now we can `unbraid` the scheduled `extAddr'` into two external addresses, and pair each with *the same* internal address: namely, all but the lowest bits of what used to be `charAddr`.

```
    extAddr1 :> extAddr2 :> Nil = D.unbundle $ unbraid <$> extAddr'

    intAddr = guardA newChar $ liftA2 toAddr <$> x1 <*> y1
      where
        toAddr :: Index TextWidth -> Index TextHeight -> Index TextSize
        toAddr x1 y1 = bitCoerce (y1, x1)
```

This works because the lowest bit, corresponding to the difference between character and attribute data, will now manifest itself in which of the two video memory units we read from:

```
    videoMem extAddr = sharedDelayedRW ram $
        extAddr `withWrite` extWrite :>
        noWrite intAddr :>
        Nil
      where
        ram = singlePort $ delayedRam $
            blockRam1 NoClearOnReset (SNat @TextSize) 0

    extRead1 :> charRead :> Nil = videoMem extAddr1
    extRead2 :> attrRead :> Nil = videoMem extAddr2
    extRead = extRead1 .<|>. extRead2
```

The seeming prioritization of `extRead1` over `extRead2` is, of course, purely arbitrary: due to their construction by unbraiding, at any given cycle, at most of one of `extAddr1` and `extAddr2` (and correspondingly, `extRead1` and `extRead2`) will take on a `Just` value.

The next step is to make sense of the attribute byte. Right after loading, we immediately latch it so that its value is available through the rendering of the given block:

```
    attr = delayedRegister 0 (.|>. attrRead)
    (isPlot, blink, back, fore) = D.unbundle $ attributes <$> attr
```

Now that we know what the `background` and `foreground` colors should be for a given block, we can make use of that information when computing the RGB value for the current `pixel`. One neat way of formulating it is to say that there is a *per-character palette* of just two colors, but that palette changes dynamically, i.e. is a `DSignal` itself:

```
    palette = D.bundle $ back :> fore :> Nil
    border = pure (0x30, 0x30, 0x30)

    rgb = mux (delayI True isBorder) border $
        fromBGR <$> (palette .!!. pixel)
```

The rest of the changes follow the software implementation directly. To support the `isTall` attribute, we compute `y0'` using `toTall`:

```
    y0' = mux isTall tall short .<| 0
      where
        short = delayI Nothing y0
        tall = delayI Nothing $ liftA2 toTall <$> y1 <*> y0
```

We have already seen how plot mode can be implemented with the help of two small ROMs. Here, we set the next `block` to the result of reading either from the `plotRom` or the `fontRom`:

```
    block = enable (delayI False newChar) $
        mux (delayI False isPlot)
          (plotRom plotAddr (delayI Nothing y0 .<| 0))
          (fontRom fontAddr y0')
```

The lookup function `plotRom` takes the character byte apart into two halves, uses each half as addresses to two copies of the same ROM (filled with `plots`), and stretches the resulting bits into a full 8-pixel horizontal character block.

```
plotRom
    :: (HiddenClockResetEnable dom)
    => DSignal dom n (Unsigned 8)
    -> DSignal dom n (Index 8)
    -> DSignal dom (n + 1) (BitVector 8)
plotRom char row = stretchRow <$> b1 <*> b2
  where
```

```
    (hi, lo) = D.unbundle . fmap splitChar $ char
    row' = halfIndex <$> row

    b1 = delayedRom (rom plots) $ plotAddr <$> lo <*> row'
    b2 = delayedRom (rom plots) $ plotAddr <$> hi <*> row'
```

This brings us to feature parity with the SDL renderer.

18.4.5 Port-based control via the CRT 5027

Our implementation so far only takes into account the video RAM's contents. To change what's on the screen, the CPU simply writes to the appropriate regions of memory.

However, the CRT 5027 also provides a higher-level interface to changing the video result, via port-mapped I/O commands. On a real CRT 5027, there are two categories of these commands: changing the fundamental parameters of the video generator, and controlling a small set of dynamic transformations that can be composed with the video RAM-based output: showing a text cursor, and vertical scrolling.

We are only going to look at the latter group of operations. The former is only useful insofar as the same CRT 5027 component can be used in multiple computers without any hardware changes; however, in a real Compucolor II, part of the initialization sequence reads out a small 32-byte PROM connected to I/O ports `0x20` to `0x30` and feeds it to the various ports of the CRT 5027 to set up the $(64 \times 6) \times (32 \times 8)$ video mode with the correct timing settings for the built-in CRT. Afterwards, changing these values was not supported; as page 110 of the *Compucolor II Programming and Reference Manual* succinctly puts: "Warning: Do not output any values to the SMC 5027 CRT chip".

There are the following six "benign" I/O ports controlling the cursor and scrolling:

- `0x6` sets the vertical offset as an integer number of (non-tall) character lines, modulo 32. Specifically, it sets the line number of the last displayed line, so a value of 29 means the screen is scrolled down 2 lines (in a wrap-around fashion): the first visible line is line 30, followed by line 31, then 0, 1, all the way to 29. Note that the choice of top vs. bottom half for tall characters is not affected by scrolling.

- `0xb` scrolls the screen up by one line. This, of course, is just a more convenient way of incrementing the scroll offset by one.

- `0xc` and `0xd` set the horizontal and vertical (character) coordinates of the hardware cursor, respectively. The hardware cursor is drawn by overlaying two white, blinking horizontal bars on the given character.
- `0x8` and `0x9` retrieve the horizontal and vertical coordinates of the hardware cursor. These use different port numbers from `0xc` and `0xd` because the CRT 5027 has no read/write selection: reading and writing functionality need to be on separate ports.

In our implementation, we are going to add the high-level control interface of the CRT 5027 as a separate component that communicates with the video signal generator:

```
crt5027
    :: (HiddenClockResetEnable dom)
    => Signal dom Bool
    -> Signal dom (Maybe (PortCommand (Index 16) (Unsigned 8)))
    -> ( Signal dom (Maybe (Unsigned 8))
       , Signals dom CRTOutput
       )
crt5027 frameEnd cmd = (dataOut, crtOut)
  where -- Continued below, after the datatype definitions
```

Beside the port command-based CPU interface, there is one extra `Bool` input and a record of extra outputs. The extra input comes from the video generator and signals the end of each (visible) frame; this is used to blink the cursor every 16 frames (roughly four times a second). The extra outputs contain the current cursor position and the current scroll offset:

```
declareBareB [d|
  data CRTOutput = MkCRTOutput
      { cursor :: Maybe (Unsigned 8, Unsigned 8)
      , scrollOffset :: Index TextHeight
      } |]
```

In `crt5027` we need to keep track of the current values of the signals to generate, and change them according to the port writes, as necessary:

```
data S = MkS
    { _cursorX :: Unsigned 8
    , _cursorY :: Unsigned 8
    , _lastLine :: Index TextHeight
    } deriving (Show, Generic, NFDataX)
makeLenses ''S
```

```
initS :: S
initS = MkS
    { _cursorX = 0
    , _cursorY = 0
    , _lastLine = 0
    }
```

We use raw, 8-bit bytes to store the cursor's X and Y position. This allows the cursor to be outside the visible area, which is an easy way to turn it off. Also, this makes it possible to roundtrip the coordinates when set via ports `0xc/0xd` and then read out over `0x8/0x9`.

Internally, there isn't much to do in `crt5027`, since the bulk of the work will be in the changes to the `video` generator to interpret these new signals. The `frameEnd` signal drives an oscillator, which is used to mask out the cursor position to facilitate its blinking.

```
crt5027 frameEnd cmd = (dataOut, crtOut)
  where
    blink = riseEveryWhen (SNat @16) frameEnd
    blinkState = oscillateWhen True blink

    (dataOut, unbundle -> crtOut) =
        mealyStateB step initS (cmd, blinkState)
```

To take one `step`, we `execute` the new port command (if any), and do a bit of post-processing on the state, to make the video generator's life easier:

- The cursor coordinates are `Just` values only on frames where the cursor should be visible

- The last displayed line index is turned into a scroll offset, i.e. 0 for no scrolling, 1 to scroll up by one line, and so on. This means we will be able to compute the visible Y coordinate from the real Y coordinate with a simple (wrapping-over) addition.

```
    step (cmd, blinkState) = do
        dataOut <- traverse exec cmd
        scrollOffset <- nextIdx <$> use lastLine
        x <- use cursorX
        y <- use cursorY
        let cursor = (x, y) <$ guard blinkState
        return (dataOut, MkCRTOutput{..})
```

We keep `exec` simple by only implementing the ports that are useful for the Compucolor II:

```
type Ctl = State S

exec :: PortCommand (Index 16) (Unsigned 8) -> Ctl (Unsigned 8)
exec cmd = case cmd of
    ReadPort 0x8 -> use cursorX
    ReadPort 0x9 -> use cursorY
    WritePort port val -> (*> return 0x00) $ case port of
        0x6 -> lastLine .= bitCoerce (resize val)
        0xb -> lastLine %= nextIdx
        0xc -> cursorX .= val
        0xd -> cursorY .= val
        _ -> return ()
```

Next, we need to hook up these new signals to the video driver. We change its type to add a new parameter corresponding to `CRTOutput`, and a new output to signal the end of a frame:

```
video
    :: (HiddenClockResetEnable Dom40)
    => Signals Dom40 CRTOutput
    -> Signal Dom40 (Maybe (Bool, VidAddr))
    -> Signal Dom40 (Maybe (Unsigned 8))
    -> ( VGAOut Dom40 8 8 8
       , Signal Dom40 Bool
       , Signal Dom40 (Maybe (Unsigned 8))
       )
video
  MkCRTOutput{..}
  (unsafeFromSignal -> extAddr)
  (unsafeFromSignal -> extWrite) =
    ( delayVGA vgaSync rgb
    , toSignal $ delayI False frameEnd <* rgb
    , toSignal extRead
    )
  where
```

We signal `frameEnd` whenever we enter the vertical blanking period:

```
    vblank = isNothing <$> y1
    frameEnd = liftD (isRising False) vblank
```

Now for the new features. Scrolling is a simple matter of transforming y1 according to the scroll offset:

```
    (fromSignal -> rawY1, fromSignal -> y0) =
        scale (SNat @FontHeight) .
        fst . scale (SNat @2) .
        center $
        vgaY
    y1 = scroll <$> fromSignal scrollOffset <*> rawY1
```

Applying the scroll offset means adding the offset, wrapping over the range. For example, if the screen is scrolled down by two lines (i.e. scrollOffset's value is 30), when the electron ray is scanning the first line (rawY1 is 0), we need to read from video RAM as if we were scanning line 30; then, when rawY1 is 1, we pretend that it is 31; and in the next row of characters, we add 30 to 2, wrapping over to 0.

```
scroll :: (SaturatingNum a) => a -> Maybe a -> Maybe a
scroll offset x = satAdd SatWrap offset <$> x
```

The only place in the video driver where we have to look through the abstraction of y1 and use the rawY1 value is when computing y0' in tall mode: the Compucolor computes line parity from the original, unscrolled Y coordinate.

```
    y0' = mux isTall tall short .<| 0
      where
        short = delayI Nothing y0
        tall = delayI Nothing $ liftA2 toTall <$> rawY1 <*> y0
```

The cursor is rendered by drawing the top and bottom lines of the cursor position in white. Blinking is achieved by flipping between Just the coordinates, and Nothing on frames where the cursor shouldn't be drawn. We can also piggy-back on this to implement the blink attribute: on the Compucolor, characters that have the blink attribute set are rendered with a black foreground on frames where the cursor is visible.

```
    (isBlinked, isCursor) = D.unbundle $ do
        x1 <- fromIntegral <$> (delayI Nothing x1 .<| 0)
        y1 <- fromIntegral <$> (delayI Nothing y1 .<| 0)
        y0 <- delayI Nothing y0 .<| 0
        cursor <- delayI Nothing $ fromSignal cursor
        blink <- blink
        pure $
            let atCursor = cursor == Just (x1, y1)
                cursorRow = any (y0 ==) [minBound, maxBound]
            in (blink && isJust cursor, atCursor && cursorRow)
```

We determine that the currently drawn pixel belongs to the cursor if it is in the same block as the cursor (the `fromIntegral` calls converts `x1` and `y1` to full 8-bit bytes for the comparison), and we are drawing either the top or the bottom row.

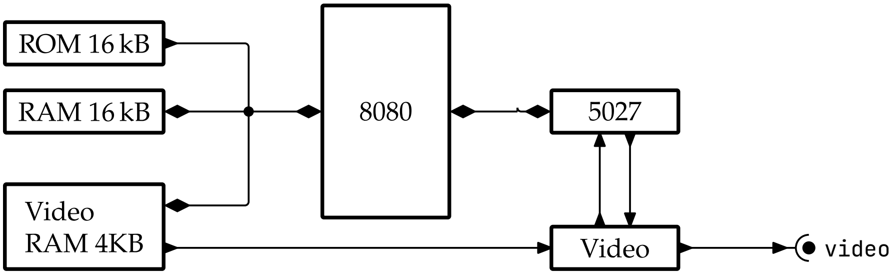
We can then use these two predicates to drive rendering as follows. Characters that are blinked should be drawn with a black foreground, so we simply replace `fore` with `black` in the palette. Similarly, inside the cursor, the whole palette is replaced with just white.

```
    black = (0, 0, 0)
    white = (1, 1, 1)

    palette = mux isCursor (pure $ repeat white) $ D.bundle $
        back :>
        mux isBlinked (pure black) fore :>
        Nil
```

We now need to hook up the CRT 5027 module to the CPU, and connect its output to the video signal generator. On the Compucolor II, the 16 ports of the CRT 5027 are mapped to the CPU's ports twice, starting at `0x60` and `0x70`. We extend our `mainBoard` accordingly, tunneling the `frameEnd` input and the `crtOut` output to outside the `mainBoard`.

```
mainBoard
    :: (HiddenClockResetEnable dom)
    => Signal dom Bool
    -> Signal dom (Maybe (Unsigned 8))
    -> ( Signals dom CRTOutput
       , Signal dom (Maybe VidAddr)
       , Signal dom (Maybe (Unsigned 8))
       )
mainBoard frameEnd vidRead = (crtOut, vidAddr, vidWrite)
  where
    CPUOut{..} = intel8080 CPUIn{..}
    interruptRequest = pure False

    dataIn = Just 0 |>. dataIn'
    (dataIn', ((vidAddr, vidWrite), crtOut)) =
        $(memoryMap [|_addrOut|] [|_dataOut|] $ do
            rom <- mapH [|Just|] =<<
                romFromFile (SNat @0x4000) [|"compucolor.bin"|]
            ram <- mapH [|Just|] =<< ram0 (SNat @0x4000)
            (vid, vidAddr, vidWrite) <-
                conduit @(Bool, VidAddr) [|vidRead|]
            (crt, crtOut) <- port @(Index 16) [| crt5027 frameEnd |]
            matchJust $ do
                matchLeft @(Unsigned 8) $ do
                    from 0x60 $ connect crt
                    from 0x70 $ connect crt

                matchRight @(Unsigned 16) $ do
                    from 0x0000 $ connect rom
                    from 0x6000 $ tag True $ connect vid
                    from 0x7000 $ tag False $ connect vid
                    from 0x8000 $ connect ram

            return ((vidAddr, vidWrite), crtOut)
```

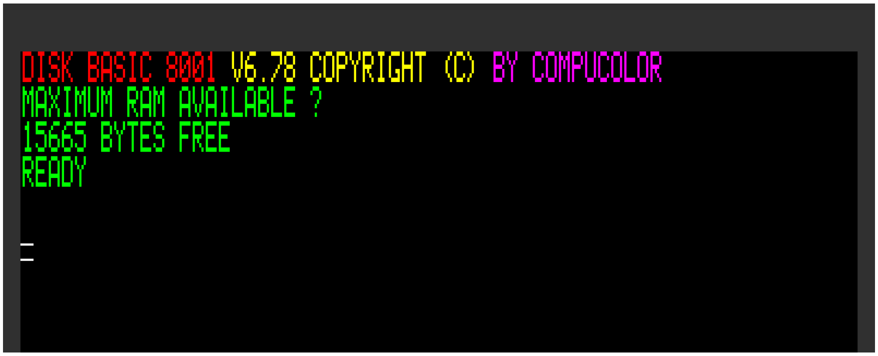
Our current design corresponds to the following schematic. Note that in our implementation, `crt5027` only takes care of interfacing with the CPU; the actual video signal generation is separate:



We omit the changes to the simulators, since they are trivial: the `frameEnd` input doesn't matter if the `crtOut` output is ignored anyway. The change to `topEntity` is also a straightforward matter of piping:

```
topEntity
    :: "CLK_40MHZ" ::: Clock Dom40
    -> "RESET"     ::: Reset Dom40
    -> "VGA"       ::: VGAOut Dom40 8 8 8
topEntity = withEnableGen board
  where
    board = vga
      where
        (crtOut, vidAddr, vidWrite) = mainBoard frameEnd vidRead
        (vga, frameEnd, vidRead) = video crtOut vidAddr vidWrite
```

We have also sneakily increased the RAM size in `mainBoard` from 8 kB to 16 kB, to mark the milestone of finishing all video-related features:



```
DISK BASIC 8001 V6.78 COPYRIGHT (C) BY COMPUCOLOR
MAXIMUM RAM AVAILABLE ?
15665 BYTES FREE
READY
```

18.5 TMS 5501

If the Intel 8080 is the brain of the Compucolor II, then the TMS 5501 is its brainstem and spinal cord all in one. This versatile I/O controller packs a ton of features into a single IC, controlled via 16 ports:

- First of all, it provides an **interrupt source**, including an implementation of the 8080 interrupt song and dance routine. Since there are eight possible `RST` instructions, the 5501 can map eight different functionality to different interrupts. All interrupts can also be individually **masked**, i.e. turned on or off. A lot of the other features listed below are (optionally) exposed to the CPU via interrupts.
- As an alternative to interrupts, the TMS 5501 also supports **polling**. In this configuration, whenever convenient, the CPU can read from a port to query for any pending interrupts.
- **Parallel 8-bit I/O**: separate input and output pins allow easy interfacing with peripherals via port commands. While the input is directly exposed to the CPU via a port read, the outputs are latched and inverted.
- Five 8-bit **timers** running at 64 microseconds. Five dedicated ports allow setting the timer values, but there is no way for the CPU to query the remaining countdown value. Instead, each timer gets its own interrupt index which is fired when the timer runs out.
- **Serial I/O** via a 9600 bits per second UART. Two separate interrupts signal the transmission buffer becoming empty and the receiver buffer filling, alleviating the CPU from busy-waiting.

- Two external low-to-high **interrupt triggers**: one of them a dedicated *sensor* pin, the other the highest bit of the parallel input.
- A special **testing mode** speeds up all clocks (timers and the UART) by 8: timers run at 8 microsecond resolution, and the UART data transfer rate goes up to 76,800 bits per second. Originally, this was intended for diagnostic / factory testing purposes, but as we will see, the Compucolor II uses this feature to good effect when interfacing with the floppy drive.

Those who are keeping track would notice that we ended up with nine interrupt sources, even though the 8080 can only distinguish between eight. The resolution of this is that interrupt 7 can be configured to either correspond to timer 4 or the low-to-high transition of bit 7 of the parallel input pins. The complete assignment of interrupt sources to interrupt numbers is as follows:

0. Timer 0
1. Timer 1
2. Sensor pin low-to-high
3. Timer 2
4. UART receiver buffer filled
5. UART transmission buffer emptied
6. Timer 3
7. Timer 4 or MSB of parallel input low-to-high

This also corresponds to a priority ordering, with lower indices taking higher priority.

18.5.1 Towards keyboard input

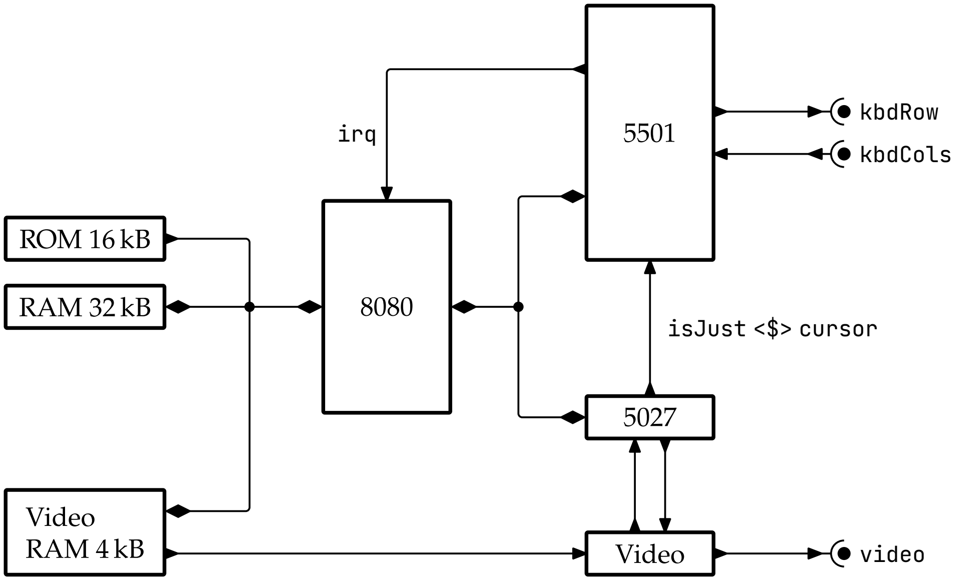
The list of TMS 5501 functionality may look a bit daunting at first, but the good news is that most of them only interact with each other via the interrupt register: we can chip away at it feature by feature until we have everything we need. In this section, we will concentrate on the components required to get keyboard input working. So how does keyboard input work on the Compucolor II?

As we have seen when interfacing with the 4×4 keypad, a pin-frugal way of reading lots of pushbutton states is to organize them in a matrix of rows and columns, and then use temporal multiplexing to read out each row one by one. This is exactly how the full keyboard is read on the Compucolor: the parallel output pins of the TMS 5501 act as a row selector, and the parallel input pins read back the columns of the given row. The whole process is driven by an interrupt handler that is triggered by timer 2.

In our implementation, we will interpret the PS/2 keyboard events to build a virtual keyboard matrix, and then connect that to the TMS 5501. As we have seen, the following 5501 features will be needed for this step:

- Parallel I/O
- Timers
- Interrupt generation

When we finish this section, the resulting computer will look like this:



18.5.2 Generating interrupts

Let's start with interrupt generation, since that functionality will tie everything else together. The external interface of this minimal TMS 5501 consists of the interrupt acknowledgment and the port command as inputs, and the result of the port command and the interrupt signals as output:

```
declareBareB [d|
  data TMSInput = MkTMSInput
      { ack :: Bool
      } |]
```

```
declareBareB [d|
  data TMSOutput = MkTMSOutput
      { interruptRequest :: Bool
      , rst :: Maybe Value
      } |]

tms5501
    :: (HiddenClockResetEnable dom)
    => Signals dom TMSInput
    -> Signal dom (Maybe (PortCommand (Index 16) Value))
    -> ( Signal dom (Maybe Value)
       , Signals dom TMSOutput
       )
```

Planning ahead for the final, feature-complete version, we split the TMS 5501 into two parts: the *controller* and the *UART*. This will make it easier to reuse our existing UART implementation when we get to that point. We delay the interrupt outputs by one period to avoid a cycle between the CPU changing the interrupt settings (by emitting `PortCommands`) and the TMS 5501 interrupting the CPU based on these interrupt settings.

```
tms5501 MkTMSInput{..} cmd = (dataOut, out)
  where
    out = MkTMSOutput{..}

    (dataOut, unbundle -> MkCtlOutput{..}) = mealyStateB
        (uncurryN controller)
        initCtlS
        (bundle MkCtlInput{..}, cmd)
    interruptRequest = register False irq
    rst = register Nothing int
```

All of the controller is going to live in the state machine created by `mealyStateB`. In this initial version, the state only consists of the currently active interrupts and the interrupt mask. As we add new features, we will keep extending this data type.

```
data CtlS = MkCtlS
    { _intBuf :: BitVector 8
    , _intMask :: BitVector 8
    }
    deriving (Show, Generic, NFDataX)
makeLenses ''CtlS
```

Initially, no interrupts are active and all are disabled:

```
initCtlS :: CtlS
initCtlS = MkCtlS
    { _intBuf = 0x00
    , _intMask = 0x00
    }
```

The controller itself has some inputs and outputs, and reacts to `PortCommand`s:

```
declareBareB [d|
  data CtlInput = MkCtlInput
      { ack :: Bool
      } |]

declareBareB [d|
  data CtlOutput = MkCtlOutput
      { irq :: Bool
      , int :: Maybe Value
      } |]

controller
    :: Pure CtlInput
    -> Maybe (PortCommand Port Value)
    -> State CtlS (Maybe Value, Pure CtlOutput)
```

Inside the controller, we have to respond either to an interrupt acknowledgment from the CPU, or to an incoming port command. We also have to check for pending interrupts and notify the CPU accordingly:

```
controller MkCtlInput{..} cmd = do
    (int, dataOut) <- do
        if ack then do                                  -- (1)
            opcode <- clearPending
            return (Just opcode, Nothing)
          else do                                       -- (2)
            dataOut <- traverse exec cmd
            return (Nothing, dataOut)

    irq <- isJust <$> getPending                        -- (3)

    return (dataOut, MkOutput{..})
```

1. Whenever the CPU sends an interrupt acknowledgment, we reply with the appropriate `RST` machine code and clear the pending interrupt:

```
clearPending :: Ctl Value
clearPending = do
    pending <- getPending
    traverse clearInt pending
    return $ toRST pending
```

First, we determine the pending interrupt index by looking for the highest priority enabled value in the interrupt buffer:

```
getPending :: State CtlS (Maybe Interrupt)
getPending = do
    masked <- maskBy <$> use intMask <*> use intBuf
    return $ if masked == 0 then Nothing
        else Just . fromIntegral $ countTrailingZeros masked
  where
    maskBy mask = (mask .&.)
```

If there is no pending interrupt, the TMS 5507 data sheet specifies that we should return the interrupt with the lowest priority, i.e. 7:

```
toRST :: Maybe Interrupt -> Value
toRST = rst . fromMaybe 7
```

We also remove the pending interrupt from the queue by clearing its bit in `intBuf`:

```
setInt :: Interrupt -> State CtlS ()
setInt i = intBuf %= (`setBit` fromIntegral i)

clearInt :: Interrupt -> State CtlS ()
clearInt i = intBuf %= (`clearBit` fromIntegral i)
```

2. During normal (non-interrupt-handling) operation, port commands are executed by the following function, which will gain a lot more branches over the course of this chapter:

```
exec :: PortCommand Port Value -> State CtlS Value
exec cmd = case cmd of
    ReadPort _ -> return 0x00

    WritePort port x -> (*> return 0x00) $ case port of
        0x8 -> intMask .= bitCoerce x
        _ -> return ()
```

3. After handling the inputs for the current cycle, we determine if any interrupts are still, or newly, pending. This is after handling the interrupt acknowledgment, since that could have cleared the previously pending interrupt.

One reasonable question is why there needs to be two separate outputs for `dataOut` and `int`. Indeed, in a real TMS 5501 chip, there is only one set of eight pins connected to the data bus; and even in our implementation, at most one of `dataOut` and `int` will contain a `Just` value in any given cycle. However, we need to route the `int` output straight to the CPU, bypassing any other address decoding, whenever `ack` fires; and the simplest way of doing this is by adding a separate output that can be used directly in the memory map in an `override`.

This gives us a good frame to hang the rest of the TMS 5501 functionality on. For starters, we can implement polling mode. In this mode, the TMS 5501 doesn't respond to the `ack` input; instead, the CPU can get the pending interrupt (in `RST` form) by reading from port 2. Whether an interrupt is pending or not is available in the status byte on port 3. The mode itself is controlled via what the TMS 5501 data sheet calls a "discrete command": each bit of the byte written to port 4 has a different effect, with bit 3 turning `ack` responses on/off.

The first change is to store the current setting in the controller's state:

```
data CtlS = MkCtlS
    { _enableAck :: Bool
    ...
    }

initCtlS = MkCtlS
    { _enableAck = True
    ...
    }
```

Then, we change the `controller` to take this new flag into account:

```
controller MkCtlInput{..} cmd = do
    (int, dataOut) <- do
        shouldRespond <- use enableAck
        if ack && shouldRespond then do
            ...
```

The rest of the changes add new branches to `exec`:

- A read from port 2 is the polling itself: the pending interrupt is cleared and put on the data bus. We have already implemented this logic for handling `ack`s, so we can reuse `clearPending`:

  ```
  ReadPort 0x2 -> clearPending
  ```

- Reading port 3 puts the status byte on the data bus:

  ```
  ReadPort 0x3 -> getStatus
  ```

  Most of the eight bits of the status byte contain information about the UART, but bit 5 shows if there are any pending interrupts. Until we implement the UART integration (which we don't need just to get the keyboard working), we fill most bits with dummy values:

  ```
  getStatus :: State CtlS Value
  getStatus = do
      intPending <- isJust <$> getPending
      return $ bitCoerce $
        False :>
        False :>
        intPending :>
        True :>
        True :>
        False :>
        False :>
        False :>
        Nil
  ```

  The dummy bits are chosen to correspond to the "happy path" of serial communication: no errors, and no need to wait for anything because the transmitter buffer is empty and the receiver buffer is full.

- Each bit of the value written to port 4 controls a different internal flag of the TMS 5501:

  ```
  WritePort 0x4 x -> execDiscrete x
  ```

  For now, we implement bits 0 and 3. Bit 0 signals a reset: all timers are turned off, and all pending interrupts are cleared, except for interrupt 4, which is set. Bit 3 changes between interrupting and polling mode.

  ```
  execDiscrete :: Value -> Ctl ()
  execDiscrete cmd = do
      when (cmd `testBit` 0) $ do
          intBuf .= 0b0001_0000
          timers .= repeat 0
      enableAck .= cmd `testBit` 3
  ```

At this point, we have the final shape of the `controller`: we just need to add the rest of the functionality to the appropriate places.

18.5.3 Parallel I/O

The parallel I/O capabilities of the TMS 5501 consists of eight separate input and output lines that can be read and latched via port commands; the rising edge of bit 7 can also be used as an interrupt event. We start by extending the interface of `tms5501`:

```
data TMSInput = MkTMSInput
    { parallelIn :: BitVector 8
    ...
    }

data TMSOutput = MkTMSOutput
    { parallelOut :: BitVector 8
    ...
    }
```

If we added the same fields to `CtlInput` only, we would need to make the detection of the rising edge of `msb <$> parallelIn` part of the stateful controller. However, said detection can be implemented much more straightforward using the signal function `isRising`, so we do that outside `controller`, and pass its result in a separate `inputTrigger` field:

```
tms5501 MkTMSInput{..} cmd = (dataOut, out)
  where
    inputTrigger = isRising low $ msb <$> parallelIn
    ... -- Rest unchanged

data CtlInput = MkCtlInput
    { parallelIn :: BitVector 8
    , inputTrigger :: Bool
    ...
    }

data CtlOutput = MkCtlOutput
    { parallelOut :: BitVector 8
    ...
    }
```

The parallel input is exposed directly via port 1, and the output is set via a complementing latch from port 7. To implement `ReadPort 0x1`, extend `exec` slightly, by giving it access to the `CtlInput`:

```
exec :: Pure Input -> PortCommand Port Value -> Ctl Value
exec inp@MkInput{..} cmd = case cmd of
    ReadPort 0x1 -> return $ unpack parallelIn
    WritePort port x -> (*> return 0x00) $ case port of
        0x7 -> parallelBuf .= pack x
```

Responding to the `inputTrigger` is user-configurable, via bit 2 of the discrete command byte:

```
data CtlS = MkCtlS
    { _enableInputTrigger :: Bool
    ...
    }

execDiscrete cmd = do
    enableInputTrigger .= cmd `testBit` 2
    ...
```

Inside the `controller` itself, we now have our first interrupt source of many to come. We also compute the parallel output by taking the `complement` of its buffer.

```
controller inp@MkCtlInput{..} cmd = do
    inputTriggerEnabled <- use enableInputTrigger
    when (inputTrigger && inputTriggerEnabled) $ setInt 7

    ...

    parallelOut <- complement <$> use parallelBuf
    return (dataOut, MkOutput{..})
```

At this point, we should have a good idea of how the keyboard sampling will work: the CPU first selects a given row by writing to port 7, which is connected to the keyboard matrix. This value is kept by the latch, so when port 1 is read next, the output is still what was written to port 7 previously, closing the circuit from the parallel output, via the keyboard matrix, to the parallel input. After reading from port 1, the CPU can change the output to select the next row of keys.

Since the switches under the keys are internally arranged in 16 rows, we can do a full scan of the keyboard state by doing this 16 times. Keypresses can be detected in software by comparing the newly discovered key states with a previously stored state.

Of course, the CPU can't keep doing this keyboard sampling in a loop: that would leave no time to do anything useful, such as actually running programs.

What is missing is a way to periodically trigger this keyboard sampling routine, *interrupting* the normal execution of programs.

18.5.4 Countdown timers

The TMS 5501 provides five timers for this and similar applications. Each timer counts down at 64 microseconds and triggers a timer-specific interrupt when it reaches zero. So for example, if the CPU wants to do something after 10 milliseconds, it can set timer 1 to 157, put that something in the interrupt handler for `RST 0`, and keep running normally. To get periodic behavior, the interrupt handler can reset the timer value to 157, ensuring it will be called again.

Internally, the state of each timer is an 8-bit unsigned number:

```
data CtlS = MkCtlS
    { _timers :: Vec 5 (Unsigned 8)
    ...
    }
```

In a situation similar to `inputTrigger`, creating the 64 microsecond `tick` signal is easier outside the `controller`, using `risePeriod`:

```
tms5501 MkTMSInput{..} cmd = (dataOut, out)
  where
    tick = risePeriod (SNat @(Microseconds 64))
    ... -- Rest unchanged

data CtlInput = MkCtlInput
    { tick :: Bool
    ...
    }
```

The workhorse function of timer handling is `setTimer`, which replaces a given timer's current countdown value. According to the TMS 5501 documentation, setting a timer to zero should trigger an immediate interrupt; this allows us the share the implementation of `setTimer` between reacting to port writes and counting down on a tick.

```
setTimer :: Index 5 -> Unsigned 8 -> State CtlS ()
setTimer i newCount = do
    timers %= replace i newCount
    when (newCount == 0) $ do
        enableTimer4 <- not <$> use enableInputTrigger
        traverse_ setInt $ case i of
```

```
            0 -> Just 0
            1 -> Just 1
            2 -> Just 3
            3 -> Just 6
            4 -> guard (enableTimer4) *> Just 7
```

Note that although the interrupt from timer 4 is enabled only when the input trigger is disabled, the timer value itself is still always updated.

Inside the `controller`, each timer is updated on each `tick`. If a given timer already contains the value 0, we don't update it (`setTimer` is not called), since doing so would keep triggering them.

```
controller inp@MkCtlInput{..} cmd = do
    when tick countdown
    ...

countdown :: State CtlS ()
countdown = for_ indicesI $ \i -> do
    count <- uses timers (!! i)
    traverse_ (setTimer i) (predIdx count)
```

The other source of timer changes are, of course, writes to the appropriate ports:

```
exec :: Pure Input -> PortCommand Port Value -> State CtlS Value
exec inp@MkInput{..} cmd = case cmd of
    WritePort port x -> (*> return 0x00) $ case port of
        0x9 -> setTimer 0 x
        0xa -> setTimer 1 x
        0xb -> setTimer 2 x
        0xc -> setTimer 3 x
        0xd -> setTimer 4 x
```

At this point, we have implemented all parts of the TMS 5501 that are used by the conjunction of hardware and software in the Compucolor II to implement keyboard output. All we need is to translate PS/2 keyboard events to key matrix states. Before we do that, though, we extend our TMS 5501 implementation with two more features that logically fit into this section on timers.

First, we implement the TMS 5501's so-called *test mode*. In this mode, all timers (including the UART speed) is sped up by eight. We can easily implement this by starting from an 8 microsecond "fast" timer, and computing the 64 microsecond "slow" timer by taking every 8th tick of the former. We cut the cycle between the controller's input and output with a well-placed `register`, as usual.

```
tms5501 MkTMSInput{..} cmd = (dataOut, out)
  where
    fastTick = risePeriod (SNat @(Microseconds 8))
    slowTick = riseEveryWhen (SNat @8) fastTick
    tick = mux (register False fast) fastTick slowTick
    ...
```

We select between the two with a new piece of controller state that can be changed via bit 4 of the discrete command byte:

```
data CtlS = MkCtlS
    { _testingMode :: Bool
    ...
    }

data CtlOutput = MkCtlOutput
    { fast :: Bool
    ...
    }

execDiscrete cmd = do
    testingMode .= cmd `testBit` 4
    ...

controller MkCtlInput{..} cmd = do
    ...
    fast <- use testingMode
    return (dataOut, MkOutput{..})
```

The second change is even simpler: implementing interrupt source 2, the so-called *sensor*. Sensing behaves like `inputTrigger` without the `enableInputTrigger` flag: whenever a rising edge occurs on this line, interrupt 2 is triggered.

```
data TMSInput = MkTMSInput
    { sensor :: Bit
    ...
    }

tms5501 MkTMSInput{..} cmd = (dataOut, out)
  where
    sensorTrigger = isRising low sensor
    ...
```

```
data CtlInput = MkCtlInput
    { sensorTrigger :: Bool
    ...
    }

controller inp@MkCtlInput{..} cmd = do
    when sensorTrigger $ setInt 2
    ...
```

This was a very straightforward addition, but what does it have to do with timers? The answer is not in the TMS 5501 as such, but in how the Compucolor II uses it to keep time. The memory locations `0x81B9` to `0x81BB` contain the time in seconds, minutes and hours since the machine was turned on. The slowest possible timer from the TMS 5501 would run at $255 \cdot 64\ \mu\text{s} = 16.3$ ms; having an interrupt handler that runs almost 62 times per second just to update an internal counter that tells if a whole second has passed yet would be very wasteful. Instead, in the Compucolor II, the `sensor` input of the TMS 5501 is fed from the cursor state of the CRT 5027: the cursor state changes once every 16 frames, so the rising edge detection turns it into a spike that fires every 32 frames. Since the video system is running at 60 frames per second, this gives a 1.875 Hz timer, i.e. roughly two interrupts for each update of the second timer. The Compucolor II ROM includes code that spreads the difference between 1.875 Hz and 2 Hz so that the second counter remains mostly accurate over longer times.

The only missing feature from our TMS 5502 implementation is serial communication via UART. Because its main use in the Compucolor II is to access the floppy drive, we postpone worrying about that until the floppy drive section of the chapter.

18.5.5 Integration

Pretending that we have a real keyboard matrix to connect to the parallel input and output pins, we can now change `mainBoard` to hook up the TMS 5501:

```
mainBoard
    :: (HiddenClockResetEnable dom, _)               -- (1)
    => Signal dom (BitVector 8)                      -- (2)
    -> Signal dom Bool
    -> Signal dom (Maybe (Unsigned 8))
    -> ( Signal dom (BitVector 8)                    -- (2)
       , Signals dom CRT5027.Output
       , Signal dom (Maybe (Bool, VidAddr))
       , Signal dom (Maybe (Unsigned 8))
       )
```

```
mainBoard kbdCols frameEnd vidRead =
    (kbdRow, crtOut, vidAddr, vidWrite)
  where
    CPUOut{..} = intel8080 CPUIn{..}

    tmsIn = MkTMSInput                              -- (3)
        { parallelIn = kbdCols
        , sensor = boolToBit . isJust <$> cursor
        , ack = _interruptAck                       -- (4)
        }

    kbdRow = parallelOut                            -- (3)

    dataIn = Just 0 |>. dataIn'
    (dataIn', ((vidAddr, vidWrite), crtOut@MkCRTOutput{..},
    MkTMSOutput{..})) =
        $(memoryMap [|_addrOut|] [|_dataOut|] $ do
            rom <- mapH [|Just|] =<<
                romFromFile (SNat @0x4000) [|"compucolor.bin"|]
            ram <- mapH [|Just|] =<< ram0 (SNat @0x8000)
            (vid, vidAddr, vidWrite) <-
                conduit @(Bool, VidAddr) [|vidRead|]

            (tms, tmsOut) <- port @(Index 16) [| tms5501 tmsIn |]
                                                    -- (5)
            (crt, crtOut) <- port @(Index 16) [| crt5027 frameEnd |]

            override [|rst|]                        -- (4)

            matchJust $ do
                matchLeft @(Unsigned 8) $ do
                    from 0x00 $ connect tms         -- (5)
                    from 0x10 $ connect tms
                    from 0x60 $ connect crt
                    from 0x70 $ connect crt

                matchRight @(Unsinged 16) $ do
                    from 0x0000 $ connect rom
                    from 0x6000 $ tag True $ connect vid
                    from 0x7000 $ tag False $ connect vid
                    from 0x8000 $ connect ram

            return ((vidAddr, vidWrite), crtOut, tmsOut))
```

1. Because of the timer `ticks`, we have accumulated some extra constraints in the type of `tms5501`. We can let the typechecker figure them out (they are trivially satisfiable anyway) by using the wildcard type `_` in the type signature; or we can fill it in manually as `KnownNat (DomainPeriod dom), 1 <= DomainPeriod dom)`.
2. We receive a new `BitVector 8` input which contains the columns of the currently selected keyboard row. The row selection is returned by the new `BitVector 8` output component.
3. The inputs and outputs of the TMS 5501 are connected as previously explained: the keyboard matrix is connected to the parallel pins, and the blink state from the CRT 5027 is used as the edge triggered interrupt source.
4. Since the TMS 5501 acts as our interrupt manager, we connect the CPU's `_interruptAck` pin, and let the TMS 5501 output `override` the data bus contents with its `rst` output.
5. We instantiate the `tms5501` component and connect its 16 ports to two address ranges: from `0x00` and from `0x10` onwards.

18.6 Keyboard

Although even modern keyboards, internally, are wired up in a matrix configuration, this is not directly exposed through their interface. Moreover, even if we gutted a keyboard and hooked its lines directly to our FPGA board, we would still have the problem of the layout not matching that of the Compucolor II.

Instead, we will adapt our PS/2 keyboard driver from earlier, interpreting its events by maintaining an internal virtual keyboard matrix:

```
keyboard
    :: forall dom. (HiddenClockResetEnable dom)
    => Signal dom (Maybe ScanCode)
    -> Signal dom (BitVector 8)
    -> Signal dom (BitVector 8)
```

Reading the Compucolor II documentation, we find that the keyboard matrix is organized into 16 rows, with 8 keys (i.e. columns) per row. Interestingly, the mapping is not based on the physical layout of the keyboard: for example, the second column starts with `@` and then continues with all letters from `A` to `O`; then the third column continues from `P` to `Z`, followed by some punctuation characters. Because of this, it makes more sense to write the mapping of PS/2 scancodes to keyboard matrix positions in column-major order:

```
keymap :: Matrix 8 16 KeyCode
keymap =
    (0x045 :> 0x016 :> 0x01e :> 0x026 :> 0x025 :> 0x02e :> 0x036 :>
    0x03d :>
     0x03e :> 0x046 :> 0x04c :> 0x052 :> 0x041 :> 0x04e :> 0x049 :>
    0x14a :>
     Nil) :>
    (0x000 :> 0x01c :> 0x032 :> 0x021 :> 0x023 :> 0x024 :> 0x02b :>
    0x034 :>
     0x033 :> 0x043 :> 0x03b :> 0x042 :> 0x04b :> 0x03a :> 0x031 :>
    0x044 :>
     Nil) :>
    (0x04d :> 0x015 :> 0x02d :> 0x01b :> 0x02c :> 0x03c :> 0x02a :>
    0x01d :>
     0x022 :> 0x035 :> 0x01a :> 0x054 :> 0x05b :> 0x058 :> 0x05d :>
    0x04e :>
     Nil) :>
    repeat 0x000 :>
    (0x066 :> 0x000 :> 0x000 :> 0x000 :> 0x000 :> 0x000 :> 0x000 :>
    0x000 :>
     0x16c :> 0x00d :> 0x172 :> 0x000 :> 0x000 :> 0x05a :> 0x000 :>
    0x000 :>
     Nil) :>
    (0x000 :> 0x000 :> 0x000 :> 0x000 :> 0x000 :> 0x000 :> 0x000 :>
    0x000 :>
     0x000 :> 0x174 :> 0x16b :> 0x076 :> 0x175 :> 0x000 :> 0x000 :>
    0x000 :>
     Nil) :>
    (0x029 :> 0x000 :> 0x000 :> 0x000 :> 0x000 :> 0x000 :> 0x000 :>
    0x000 :>
     0x000 :> 0x000 :> 0x000 :> 0x000 :> 0x000 :> 0x000 :> 0x000 :>
    0x000 :>
     Nil) :>
    repeat 0x000 :>
    Nil
```

This is not a complete mapping (all those `0x000` cells are unmapped), and modern keyboards don't even have the same keys as the Compucolor II; for example, the larger Compucolor II keyboard models have function keys going all the way to F15, and some dedicated command keys for Save or Load. Our mapping is meant to be a close enough approximation; for example, because the Compucolor II has no Backspace key, we map that to the Break key which is in the upper right corner.

At every cycle, `keyboard` has to return the `BitVector 8` that contains each key's state in the row selected by the input signal. But this is fundamentally different from how the PS/2 protocol works, which sends events whenever something happens (a key is pressed or released), and nothing in between. We translate between the two by computing the `keyState` of every scan code in the keyboard layout (one `Bool` per key), and grouping them into `BitVector 8` rows:

```
keyRow
    :: (HiddenClockResetEnable dom)
    => Vec 8 KeyCode -> Signal dom (BitVector 8)
keyRow = fmap (pack . reverse) . bundle . map (\kc -> keyState kc sc)
```

We `reverse` the `Vec 8 Bool` before turning it into a `BitVector 8` so that the elements of the `keymap` can be written in lowest to highest bits in the same order as in the original Compucolor II technical documentation.

Now that we know how to create the register for a single row, we can easily translate all rows:

```
keyboardState
    :: (HiddenClockResetEnable dom)
    => Signal dom (Maybe ScanCode)
    -> Vec 16 (Signal dom (BitVector 8))
keyboardState sc = map keyRow $ transpose keymap
```

Just like the `reverse` in `keyRow`, the `transpose` here is only to enable a more natural way of laying out the `keymap`.

Now we return to the `keyboard` driver itself. The 16 rows of the keyboard matrix are addressed by the lower 4 bits of the selector; we can make a skeleton keyboard driver by using `bundle` to turn the `keyboardState` into a `Vec`-valued `Signal`, and index into it (using `.!!.`) by the truncated `selector`. The role of the delay introduced by `register 0` below is, of course, to avoid cycles between the TMS 5501 and the keyboard interface. We apply `complement` because the original hardware used an active-low representation for the key matrix, so this is what the Compucolor firmware expects as well.

```
keyboard sc selector = register 0 $ complement <$> cols
  where
    keys = bundle $ keyboardState sc
    row = resize @BitVector @8 @4 <$> selector
    cols = keys .!!. row
```

However, this is not quite the full story of the keyboard driver. Although the Compucolor II keyboard has three modifier keys (Shift , Ctrl , and Rpt) and one Caps Lock key, none of these keys show up in the `keymap`. What gives?

The answer lies in the four, seemingly unused top bits of the `selector`. If bit 7 is high, the `cols` of the currently selected row is returned as discussed. However, if bit 7 of the `selector` is low, the top four bits of the return value is replaced with the state of the modifier keys. Of course, this means there is no way to sample a full key row and the modifiers at the same time; basically, we can think of it as having 17 rows of keys instead of 16, with the last one being somewhat special.

To support these modifier keys, we change `keyboard` slightly, by returning `cols` with the `mods` state included, if requested:

```
keyboard sc selector = register 0 . fmap complement $
    mux includeMods (withMods <$> mods <*> cols) cols
  where
    -- Continued below
```

The modifiers should be included if bit 7 of the selector is not set, by re-packing the `mods` and the truncated `cols`:

```
    includeMods = not . (`testBit` 7) <$> selector

    withMods :: BitVector 4 -> BitVector 8 -> BitVector 8
    withMods mods cols = pack (mods, resize cols)
```

To compute the state of the modifier keys, we can use `keyState` again, mapping the Alt keys to Rpt. To implement Caps Lock, we have to proceed differently. In the original Compucolor keyboard, the Caps Lock key had an actual hardware locking mechanism which held the key down when pressed, until the next press. On a PS/2 keyboard, however, the Caps Lock key is just another key with key press and release events; we need to use `oscillateWhen` to toggle the key state on each `keyPress`.

```
    mods = fmap pack . bundle . reverse $
        ctrl :> shift :> rpt :> capsLock :> Nil
      where
        key = (keyPress =<<) <$> sc

        ctrl = keyState 0x014 sc .||. keyState 0x114 sc
        shift = keyState 0x012 sc .||. keyState 0x059 sc
        rpt = keyState 0x011 sc .||. keyState 0x111 sc -- left/right Alt
        capsLock = oscillateWhen True $ key .== Just 0x058
```

The initial state of `capsLock` is `True` because the Compucolor II maps its various symbols and drawing characters to the unshifted keys. By starting with `capsLock` on, we ensure that we can start typing normally straight away when our machine starts up.

With most of the TMS 5501 implemented, and with our virtual keyboard driver ready to provide the keyboard matrix, it is time to put it all together and, for the first time, boot up our machine in a state where we can actually interact with it. Since we have already integrated `tms5501` into `mainBoard`, the only change we need to do is to add a PS/2 input to `topEntity`, decode that into scan codes, and feed that into our virtual keyboard matrix:

```
topEntity
    :: "CLK_40MHZ" ::: Clock Dom40
    -> "RESET"     ::: Reset Dom40
    -> "PS2"       ::: PS2 Dom40
    -> "VGA"       ::: VGAOut Dom40 8 8 8
topEntity = withEnableGen board
  where
    board ps2 = vga
      where
        scanCode = parseScanCode . decodePS2 . samplePS2 $ ps2
        kbdCols = keyboard scanCode kbdRow

        (kbdRow, crtOut, vidAddr, vidWrite) =
            mainBoard kbdCols frameEnd vidRead
        (vga, frameEnd, vidRead) = video crtOut vidAddr vidWrite
```

At this point, we have a fully functional home computer that we can boot up to BASIC to type in and run programs. It would be downright criminal to pass up the opportunity to type in the Compucolor II version of the one-liner maze generator:

```
10 PLOT 30, 96 + (28 + INT(RND(1) + 0.5) * 2) : GOTO 10
```

`PLOT 30` turns on special character support: subsequent `PLOT` commands have 96 subtracted from the character values before being written to the video memory. This is needed because parameters to `PLOT` that are below 32 are interpreted as various control codes. Since `RND(1)` returns a random number in the interval $[0, 1)$, the value of `INT(RND(1) + 0.5))` corresponds to a coin flip. The value of the expression `28 + INT(RND(1) + 0.5) * 2)`, then, is either 28 or 30, chosen randomly. 28 and 30 are, of course, the codes of the two diagonal lines in the Compucolor II character set.

18.7 Floppy drive

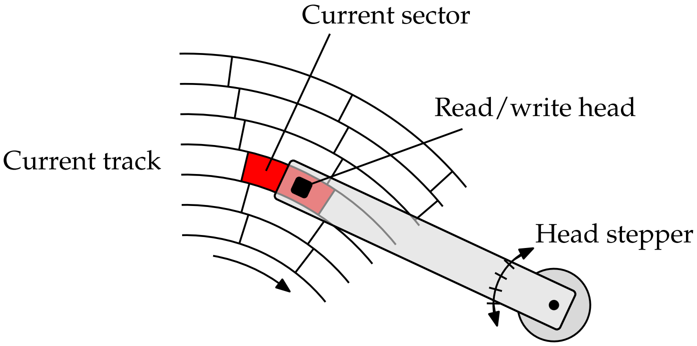
These days, the predominant data storage methods in the home computing sphere are solid state, and content is mostly delivered online. In some storage applications where performance is less critical, like home backup solutions, magnetic hard disks are still used due to the price premium of solid state drives. The Bluray drives of home consoles are the last holdouts of optical disk technology.

When the Compucolor II debuted in 1977, the storage landscape was very different. For home consumers, the two viable options were cassette tapes and floppy disks. Both of these technologies are based on modulating a magnetic field; tapes were the cheap, slow, and cumbersome option, while floppy disk drives were more expensive, but faster and more convenient. The designers of the Compucolor II went for the floppy option, but as we will see shortly, implemented it in the cheapest way possible.

18.7.1 How floppy drives and floppy disks work

A floppy disk is a thin slice of a cylinder whose surface can be magnetized, housed in some protective enclosure. Data is stored on the disk by changing the magnetic field at various points of the cylinder. The disk is read and written by a floppy drive, which contains an electromagnetic head that can be used to detect or change the magnetic field of the disk.

At the extreme, we could store a single bit on a whole disk side by using a head that has the same size and shape as the disk itself. That would be useless, of course, so instead, a much smaller head is used, and multiple bits are stored at different locations of the disk. The drive spins the disk at constant angular velocity, and the head can be moved in and out in the radial direction. Any single head position corresponds to a given ring of the disk; these rings are called *tracks*. As the disk spins, the head sweeps over the full contents of the track. To change tracks, a *stepper motor* is used: this kind of motor does a predetermined amount of rotation per input pulse, allowing for movement in discrete increments.



Note that while it is easy to know which track is currently under the head (since the head position is controlled), it is not at all straightforward to know, at a glance, where we are on the given track. The canonical solution to this problem is to divide each track into *sectors*, with each sector header containing the sector index. This reduces the problem of finding the next sector's start: once we know that, we can read off the sector index from the next bits. Different floppy disk formats used different ways of marking sector starts; some used tiny holes in the disk material and a light sensor in the drive. As we will see, the Compucolor II simply used a special bit pattern to mark the start of each sector.

18.7.2 The Compucolor II floppy drive

Based on the description of floppy disks and drives, we can conclude that a floppy drive needs the following components:

- An electromagnetic read/write head
- A motor to spin the disk
- A stepper motor to move the head in and out, track by track
- Controller circuitry that manages the stepper motor and locates tracks and sectors

The designers of the Compucolor II decided to skimp on the last component: **there is no dedicated floppy drive circuitry in the Compucolor II**. Instead, a combination of finely tuned software and the TMS 5501's parallel and serial I/O capabilities are used in concert to achieve the bare minimum of something that could be sold as a floppy drive:

- Bit 4 of the TMS 5501 parallel output port selects the floppy drive (active low). This bit controls the disk-spinning motor.

- The lowest 3 bits of the parallel output are connected to the stepper motor. The stepper motor has three leads, and the magnetized core's orientation can be set by pulling one of the leads low and two of them high. To rotate the motor in a given direction, the one low lead is changed in the right order.

- Bit 3 of the parallel output is the write-enable line.

- The head is directly connected to the serial input and output pins of the TMS 5501, gated by the select and write-enable bits of the parallel output.

So what are the implications of this design? First of all, since the UART output is directly stored on disk, that means each 8-bit byte is stored as a minimum of 10 bits with the start and stop bit. As the disk keeps rotating under the head, any pause between writes results in a string of superfluous high bits.

We can also compute the maximum possible capacity of a track, based on the rotational speed of the floppy and the UART's data rate. On the Compucolor II, the floppy disk rotates at 300 RPM, i.e. 0.2 seconds for a full rotation. By default, the TMS 5501 runs at 9600 bits per second; this would give us 1920 bits per track, or just 78,720 bits for a full 42-track disk. At 10 bits per byte, even before any sector markers, this comes to 7872 bytes: not enough to store even the smallest Compucolor model's 8 kB of RAM.

Luckily, the TMS 5501's test mode speeds up by 8× not just the timers, but the UART as well. The data rate goes up to 76800 bits per second, giving 15360 bits per track, for a total of 629,760 bits per disk. The start of each sector is marked by a long pause before its first start bit, so the final capacity of the disk is 51200 bytes: much less than other contemporary floppy systems, but at least large enough to hold data to fill even the largest Compucolor II memory configuration of 32 kB.

18.7.3 Virtual floppy disks

Our virtual floppy drive has the same connections as the real one: as inputs, a `selector` that turns on disk spinning, and a 3-bit stepper motor `phase`; and as output, the serial data read from the disk:

```
floppyDrive
    :: (HiddenClockResetEnable dom, _)
    => Signal dom Bool
    -> Signal dom (BitVector 3)
    -> Signal dom Bit
floppyDrive sel phase = mux sel rd (pure 1)
  where
    -- Continued below
```

This is for a read-only floppy drive; adding writing capability will be one of this chapter's exercises.

First, let's define some (type-level) constants that we'll use throughout this module:

```
type SlowRate = 9600
type FastRate = 8 * SlowRate

type TrackSize = 15360
type TrackCount = 41
type DiskSize = TrackCount * TrackSize
```

As a real floppy disk spins at constant angular velocity, the head scans the bits of the current track, starting at a random position, going sequentially and eventually getting back to the starting position. For the virtual version, we want to store the whole disk's contents in a piece of block RAM, which needs linear addressing. We can compute the current head `address` as the sum of a `base`, corresponding to the current track, and an `offset` that "rotates at 300 RPM", i.e. is stored as an `Index TrackSize` and updated using `nextIdx`:

```
rd' = blockRamFile (SNat @DiskSize) "disk.bin") (pure Nothing)
rd = unpack <$> rd'

tick = riseRate (SNat @FastRate)

offset = regEn (0 :: Index TrackSize) tick $ nextIdx <$> offset
addr = base + (fromIntegral <$> offset)
```

The definition of `base` is comparatively more complex, because it needs to react to changes in the stepper motor `phase`. Whenever the disk is powered on (i.e. when it is `selected` for operation), the `phase` is connected to electromagnets in the stepper motor. Rotation, and hence, moving the head in or our one track, is achieved by changing the `phase`.

```
trackSize = snatToNum (SNat @TrackSize)

base = regEn (0 :: Index DiskSize) sel $ do
    phase0 <- regEn 0 sel phase
    phase <- phase
    base <- base
    pure $
        let stepOut = satSub SatBound base trackSize
            stepIn = satAdd SatBound base trackSize
        in case (phase0, phase) of
            (0b110, 0b011) -> stepOut
            (0b011, 0b101) -> stepOut
            (0b101, 0b110) -> stepOut

            (0b110, 0b101) -> stepIn
            (0b101, 0b011) -> stepIn
            (0b011, 0b110) -> stepIn

            _ -> base
```

Note the use of `satAdd` / `satSub` in `SatBound` mode when stepping the head in or out by one track. This corresponds to stoppers on either side of the head that prevent it from going off the disk. In fact, the Compucolor II firmware depends on these stoppers to pick the right track on bootup: since the head is left in whatever position it was before turning the machine off, there is no telling which track it is currently on. By moving the head outwards 42 times, eventually the outer railing is hit, ensuring that it is now at a known track (i.e. track 0).

18.7.4 Serial receiver

To make sense of the raw bit stream coming from the floppy driver head, it is connected to the serial receiver of the TMS 5501. In this section, we re-use and extend our UART code to implement this feature; the extensions are needed because the TMS 5501 exposes details of the `RxState` to the CPU via the status byte.

The meaning of the serial receiver-related bits of the status byte are as follows:

- Bit 0 signals the detection of a *framing error*. Since stop bits are always high, if the receiver sees a low bit in the `StopBit` state, it must mean that it mis-detected the start of the transmission.
- Bit 1 signals *overruns*, i.e. if the CPU was not reading out the received bytes in time before the next byte arrived.
- Bit 2 is the raw serial input.
- Bit 3 means the receiver buffer is ready to use, i.e. it contains the contents of a freshly received byte.
- Bits 6 and 7 tell, respectively, that the first data bit, and the stop bit, has been received.

Note that bits 0, 6 and 7 are fundamentally different from bits 1 and 3. While the correct values of the former three bits can be maintained in the serial receiver on its own, the latter require tracking when the byte value is read out via `ReadPort 0x0`, and so its implementation will belong in the controller.

We start writing the receiver by defining a datatype for the various flags required by the status bit. The state of the receiver consists of the (8-bit) `RxState` itself, and the flags that are set and cleared as the `RxState` evolves.

```
data RxFlags = MkRxFlags
    { _rxStart :: Bool
    , _rxData :: Bool
    , _rxFrameError :: Bool
    }
    deriving (Show, Generic, NFDataX)
makeLenses ''RxFlags
```

```
data RxS = MkRxS
    { _rxState :: RxState 8
    , _rxFlags :: RxFlags
    }
    deriving (Show, Generic, NFDataX)
makeLenses ''RxS

initRxS :: RxS
initRxS = MkRxS
    { _rxState = RxIdle
    , _rxFlags = MkRxFlags False False False
    }
```

The meaning of the various `RxFlags` fields is given by the following stateful function:

```
updateRxFlags :: RxState n -> State RxFlags ()
updateRxFlags rxState = case rxState of
    RxBit _ (Just 1) StartBit{} -> do
        rxStart .= True
    RxBit _ (Just _) DataBit{} -> do
        rxData .= True
    RxBit _ (Just sample) StopBit{} -> do
        rxStart .= False
        rxData .= False
        rxFrameError .= (sample /= high)
    _ -> return ()
```

The external interface to the receiver feeds the serial input to `rxStep`, and updates the status flags via `updateRxFlags`. Moreover, when a reset command is issued via bit 0 of the discrete command byte, the UART is also affected. Signaling this requires an extra `Bool` parameter, which we'll fill by changing the `controller` shortly.

```
uartRx
    :: forall period. (KnownNat period, 1 <= period)
    => SNat period
    -> Bit
    -> Bool
    -> State RxS (Maybe (Unsigned 8), RxFlags)
```

```
uartRx period serialIn reset = do
    rxResult <-
        if reset then do
            rxFlags .= RxFlags False False False
            rxState .= RxIdle
            return Nothing
        else do
            rxResult <- zoom rxState $ rxStep bitDuration serialIn
            rxState <- use rxState
            zoom rxFlags $ updateRxFlags rxState
            return rxResult
    rxFlags <- use rxFlags
    return (unpack <$> rxResult, rxFlags)
  where
    bitDuration = snatToNum $ SNat @(HzToPeriod FastRate `Div` period)
```

18.7.5 Extending the TMS 5501 controller

In the controller, we need to expose the `rxResult` via `ReadPort` and the `rxFlags` via the status byte. We also need to implement the overrun and the ready bits of the status byte, and raise interrupt 4 when a new byte is received.

We start by extending the state and the input and output interfaces of `controller`. We can't use a single `Maybe (Unsigned 8)` for the `rxBuf` to package the last received value together with the `rxReady` flag, because reading out the contents of `rxBuf` only resets the flag, but keeps the actual value for subsequent repeated `ReadPort` requests.

```
data CtlS = MkCtlS
    { _rxBuf :: Unsigned 8
    , _rxReady :: Bool
    , _rxOverrun :: Bool
    ...
    }

data CtlInput = MkCtlInput
    { serialIn :: Bit
    , rxResult :: Maybe (Unsigned 8)
    , rxFlags :: RxFlags
    ...
    }
```

```
data CtlOutput = MkCtlOutput
    { rxReset :: Bool
    ...
    }
```

In the main `controller` function, we react to a new `rxResult` value by overwriting the receiver buffer, marking it as ready, and notifying the CPU via an interrupt:

```
controller inp@MkCtlInput{..} cmd = do
    for_ rxResult $ \x -> do
        whenM (use rxReady) $ rxOverrun .= True
        setInt 4
        rxReady .= True
        rxBuf .= x
    ...
```

We also change `getStatus`, passing it the `CtlInput` so that it can fish out the `RxFlag` bits from it. This is also where `rxOverrun` is reset. This fills in most of the dummy bits in `getStatus`. The only remaining one is bit 4, which signals that the serial transmitter is ready to send. Because we are only implementing the receiver half of the UART, we keep this at `True` so that programs won't get stuck waiting for a never-ending transmission to finish.

```
getStatus MkCtlInput{..} = do
    intPending <- isJust <$> getPending
    rxReady <- use rxReady
    rxOverrun <- use rxOverrun <* (rxOverrun .= False)

    return $ bitCoerce $
      rxFlags ^. rxStart :>
      rxFlags ^. rxData :>
      intPending :>
      True :>
      rxReady :>
      bitToBool serialIn :>
      rxOverrun :>
      rxFlags ^. rxFrameError :>
      Nil
```

The received byte itself is accessed via port 0. We clear `rxReady` so that the subsequent reception won't cause a false positive overrun, and also to let the CPU know (in bit 3 of the status byte) that no new byte has been received since the last time the buffer was read out.

```
exec inp@MkInput{..} cmd = case cmd of
    ReadPort 0x0 -> do
        rxReady .= False
        use rxBuf
```

One more thing remains: setting the `rxReset` output from `execDiscrete`. The problem is that `execDiscrete` is quite far from the point where the output is computed, which is in the top-level `controller` function. A `WriterT Any` monad transformer gives us a convenient way of bridging this distance: we change the type of `execDiscrete` (and `exec`) to run in `WriterT Any Ctl`, we emit an `Any True` value from `reset`, and we collect its output in the `controller`. Because this is the last time we'll touch these functions, it is worthwhile to look at their full definition, with the changes marked with `(*)`. First, `execDiscrete`:

```
execDiscrete :: Value -> WriterT Any Ctl ()
execDiscrete cmd = do
    when (cmd `testBit` 0) reset
    enableInputTrigger .= cmd `testBit` 2
    enableAck .= cmd `testBit` 3
    testingMode .= cmd `testBit` 4
  where
    reset = do
        intBuf .= 0b0001_0000
        timers .= repeat 0

        tell $ Any True                                -- (*)
        rxReady .= False                               -- (*)
        rxOverrun .= False                             -- (*)
```

Then, the complete `controller`, with all the interrupt sources we have built up over the last several sections:

```
controller inp@MkCtlInput{..} cmd = do
    when sensorTrigger $ setInt 2
    inputTriggerEnabled <- use enableInputTrigger
    when (inputTrigger && inputTriggerEnabled) $ setInt 7
    when tick countdown

    for_ rxResult $ \x -> do
        whenM (use rxReady) $ rxOverrun .= True
        setInt 4
        rxReady .= True
        rxBuf .= x
    ((int, dataOut), Any rxReset) <- runWriterT $ do    -- (*)
        shouldRespond <- use enableAck
        if ack && shouldRespond then do
            int <- clearPending
            return (Just int, Nothing)
          else do
            dataOut <- traverse (exec inp) cmd
            return (Nothing, dataOut)

    irq <- isJust <$> getPending
    parallelOut <- complement <$> use parallelBuf
    fast <- use testingMode
    return (dataOut, MkOutput{..})
```

This takes care of the implementation, now let's integrate it into `tms5501`, and, ultimately, `mainBoard`. From the outside, the only new pin of `tms5501` is the serial input pin, which we add to `TMSInput`:

```
data TMSInput = MkTMSInput
    { serialIn :: Bit
    ...
    }
```

Internally, we can use record wildcards to take care of most of the wiring between the `controller` and `uartRx`:

```
tms5501 MkInput{..} cmd = (dataOut, out)
  where
    ...
    (rxResult, rxFlags) = mealyStateB
        (uncurryN $ uartRx (SNat @(DomainPeriod dom)))
        initRxS
        (serialIn, register False rxReset)
```

In a fully generic implementation, we would also need to pass the `fast` setting to `uartRx`, and then switch between `SlowRate` and `FastRate` when computing the `bitDuration`, similar to how we switch between `slowTick` and `fastTick`. Because here we only implement the serial functionality to support the floppy drive, we can get away with hardcoding fast mode.

In `mainBoard`, we hook up the floppy drive's output to the serial input of the TMS 5501. The floppy drive is controlled by certain lines of the parallel output: a low bit 4 selects the floppy drive (and spins up the disk), while bits 2 to 0 drive the stepper motor moving the head across tracks:

```
mainBoard kbdCols frameEnd vidRead = (kbdRow, crtOut, vidAddr, vidWrite)
  where
    tmsIn = MkTMSInput
        { serialIn = floppyOut
        ...
        }

    floppyOut = register 1 $ floppyDrive sel phase
      where
        sel = not . (`testBit` 4) <$> parallelOut
        phase = slice (SNat @2) (SNat @0) <$> parallelOut
```

18.7.6 Preparing disk images

At this point, the natural progression of this chapter would be to try out our new code by adding one of the floppy images from http://www.compucolor.org/ and loading it into our machine. However, before we can do that, we need to convert these image files into flat bit sequences that we can use with the `blockRamFile` in `floppyDrive`.

Each floppy image comes in a single file with the `.ccvf` file extension. This is a simple textual file format, with the first line specifying the file format, followed by some human-readable labels, then each track is stored as a list of hexadecimal characters:

```
Compucolor Virtual Floppy Disk Image
Label Hangman (8k) 990003
Label 1. Hangman.  Classic word game.  Improves vocabulary and spelling
Label skills.  If you miss, the Compucolor II will hang you right on
Label the screen.
Label 2. Math Tutor.  Improves your mathematics skills.  Presents
Label problems in addition, subtraction, multiplication, and
Label division.  Five levels of difficulty.
Label 3. Two To Ten.  Compucolor II deals the cards.  Try to reach a
Label sum total including the "mystery number" hinted at by the
Label computer.
Track 0
0000000000000000000000000000000000000000000000000000000000000000
...
```

Since each hexadecimal character encodes 4 bits, a full track of 15360 bits takes up 3840 characters; the lines are also usually wrapped at column 64. We can parse this file using a lightweight applicative regular expression parser:

```
import Text.Regex.Applicative
import Text.Regex.Applicative.Common
import Data.Filtrable (filter)

type Track = Vec TrackSize Bit
type Disk = Vec TrackCount Track

diskLines :: Disk -> [String]
diskLines = toList . map show . concat

parseDisk :: String -> Disk
parseDisk = fromMaybe (error "CCVF parsing failed") . match disk
  where
    disk = magic *> many label *> tracks
    magic = string "Compucolor Virtual Floppy Disk Image" *> eol
    label = string "Label " *> few anySym <* eol

    tracks = for indicesI $ \i -> trackHeader i *> track

    trackHeader i = string "Track " *> filter (== i) decimal *> eol
    track = bits <$> (sequenceA . repeat $ byte <* many eol)

    eol = sym '\n'

    byte = ((++#) @4 @4) <$> hexDigit <*> hexDigit

    bits :: Vec n (BitVector 8) -> Vec (n * 8) Bit
    bits = concatMap (reverse . bv2v)
```

The only unexpected complication is having to parse the input as bytes (i.e. two hexadecimal digits at a time) instead of just 4-bit values. The reason for this is that the CCVF file format packs each 8 bits of the 15360 bit track into a byte starting at bit 0. In other words, the bit stream `abcdefghjiklmnop` is stored as two bytes `hgfedcba` and `ponmlji`. This requires `reverse`-ing in our `bits` function; but that, in turn, requires seeing all 8 bits together (otherwise we'd get `dcba hgfe lkji ponm`), hence the need for the two-digit parser `byte`.

Note how the `Track` and `Disk` types drive the parsing in the definitions of `tracks` and `track`: in both cases, the sub-parsers for one track and one hexadecimal digit are required to match exactly the right times for the resulting vector sizes to line up. So we automatically get a parse failure for e.g. disk images with a missing track. Furthermore, we check track numbers in when parsing track headers, to ensure they are in the right order.

Using the above function in a program with a thin layer of I/O scaffolding, we can take a CCVF file and prepare a `disk.bin` to be used in `floppyDrive`. Uploading our latest version onto our FPGA development board, the quickest way to smoke-testing the floppy drive is to use the `dir` command of the FCS: the key sequence Escape followed by D gets us to the `FCS>` prompt, where we can issue the `DIR` command.

We can try that out, and then instead of getting a list of file names and sizes, we get. . .

```
ESKF CD00:00 0000
FCS ERROR - EDIR
```

What is going on here?

18.7.7 Timing

The problem is the mismatch between the CPU's clock and the speed of the UART and the floppy drive. We have calculated the latter starting from the 9,600 bits per second serial speed, and then multiplied it with 8 to account for the testing mode; this gives a bit rate of 76,800 bits per second, matching the bit rate of the real Compucolor II's floppy drive. When that bit rate is converted into the duration of a single bit, as measured in clock cycles, we are using our 40 MHz clock as the reference. Thus, the `bitDuration` computed in `uartRx` is $40{,}000{,}000 \div 76{,}800 = 520$ cycles.

However, on a real Compucolor II, the clock runs at 2 MHz; the same calculation gives us $2{,}000{,}000 \div 76{,}800 = 26$ cycles as the bit duration. And of course the original Compucolor firmware was written with the assumption that 26 cycles equal one bit. In particular, when looking for the start of a sector, the CPU polls the TMS 5501 in a tight loop to measure the gap before a valid start bit. But if the UART and the floppy is presenting the same bit for 520 cycles, a run of e.g. 100 gap bits will take $520 \cdot 100 = 52{,}000$ cycles, which is a lot more than what the firmware expects ($26 \cdot 100 = 2{,}600$) , and so it gives up waiting for the first valid start bit, signaling a disk I/O error.

Solving this problem requires changes at two different levels. On one hand, the obvious solution to the 20-time speed difference is to just multiply `FastRate` with 20. This will result in a 26 cycle bit length both in the UART and the floppy drive. The second layer is a bit more complicated and involves the timing of individual Intel 8080 instructions. With `FastRate` boosted 20-fold, those 100 gap bits will be read in 2,600 cycles, while the CPU runs the part of the Compucolor II system software that was written to take roughly 2,600 cycles, so everything should match up – but if our CPU is faster than a real Intel 8080, and some of the instructions of that loop finish in less cycles, the loop overall will finish before the gap is over, leading the firmware down the path of a missing sector header.

If the difference is small, the problems caused can be quite subtle. For example, if we compile our Compucolor II with `FastRate` set to `SlowRate * 8 * 20`, and load one of the disk images that contain more than five files, the directory listing will look something like this:

```
DIRECTORY CD0: HANGMAN    06

ATR  NAME TYPE VR SBLK SIZE LBC LADR SADR

 03 MENU  .BAS;01 0006 0004 3B  829A 8455
 03 HANGMN.BAS;01 000A 002B 38  829A 9A79
 03 HARD  .LIB;01 0035 001B 80  00D8 0010
 03 EASY  .LIB;01 0050 001B 80  00D8 0010
 03 MATHTU.BAS;01 006B 001E 56  829A 9230
_ 03 TWO10 .BAS;01 0089 0032 7F  829A 9DF7
 01 <FREE SPACE>  00BB 00D5
```

The extra underscore-like character after the fifth directory entry is because the directory takes up two (consecutive) sectors, and when looking for the second sector's start, our CPU gets ahead of itself just enough to enable the "print disk output to screen" interrupt-driven callback function a bit too early, before the useful bytes of the directory entry are read. Debugging problems like this, even with the help of a simulator, can be a daunting task.

We could play whack-a-mole with this and similar bugs by altering the bitrate of the floppy drive; perhaps using 25 cycles per bit, instead of 26, would account for the instruction length difference, and everything would line up again? What we have here is software that was written for a particular hardware, using all available knowledge about that hardware without any abstractions. The only way to correctly run this software on our Intel 8080 look-alike is to increase the fidelity of our implementation; i.e. *make it look more like the real thing*. And so we face the reality of the situation head on and change our 8080 core so that every instruction takes the same number of cycles as the real CPU.

18.8 Cycle-count accuracy

This is not as herculean a task as it might first seem. Instruction timings for the Intel 8080 are well documented in the original data sheets, and our microcoded implementation allows for very simple padding to make instructions take longer.

The longest Intel 8080 instruction is `XTHL`, taking 18 cycles total. That includes one cycle for fetching the opcode `0xE3`, which is before our microcode kicks in; to make our code line up better with the documentation, we will store the desired lengths as one more than the number of micro-instructions. Recall our microcode for XTHL:

```
microcode XTHL = padded $
    pop2                                 >++>
    step INothing (SwapReg2 RHL) INothing >++>
    push2
```

pop2 and push2 both have length 2, giving our XTHL a total of 5 cycles. Then, because we use a uniform microcode length for each instruction, we pad it with ToAddrBuf micro-instructions into a full micro-program of length 8. If, instead, we padded it to 17, without the early exit after the fifth step, we could easily achieve a 17-cycle instruction time without any further changes to the microcode.

```
type MicroLen = 17

padded
    :: forall k n p pre post. (KnownNat k, KnownNat p)
    => ((n + 1 + p + k) ~ MicroLen)
    => SNat (n + 1 + p + 1)
    -> MicroSteps (n + 1) pre False
    -> Microcode
padded SNat ops = (first, withCont (usteps ++ uNOPs1) ++ uNOPs2)
  where
    (first, uops) = stepsOf ops
    uNOPs1 = replicate (SNat @p) (uNOP, Nowhere)
    uNOPs2 = repeat ((uNOP, Nowhere), False)
```

This function replaces our earlier padded function which only added the second set of uNOPs. The change to most microcode is minor; here are two examples, XTHL from earlier and XCHG:

```
microcode XTHL = padded (SNat @17) $
    pop2                                 >++>
    step INothing (SwapReg2 RHL) INothing >++>
    push2
microcode XCHG = padded (SNat @3) $
    step INothing (FromReg2 RHL) INothing >++>
    step INothing (SwapReg2 RDE) INothing >++>
    step INothing (SwapReg2 RHL) INothing
```

Here, XCHG already takes 3 cycles before padding, so padded (SNat @3) simply adds 14 uNOP micro-instructions. We still use padded to ensure the correct length even if we change the microcode later.

We can implement length padding for most of the instructions in a similar way, just by looking up their intended length in the Intel documentation, and using `padded` to transform their microcode. The only tricky cases are the following:

- The conditional instructions `CALLIf` and `RETIf` take different number of cycles depending on the checked condition's value. For example, a `RETIf` should be either 5 or 11 cycles (including the initial fetch). Our non-timing-accurate implementation is 2 or 5 cycles:

```
microcode (RETIf cond) = padded $
    step INothing (When $ Just cond) INothing >++>
    popPC
```

  If we padded it at the end with our new `padded` function, it would only apply to the case when the condition holds. Instead, we need a way of adding `padding` to the beginning of a microcode, which we can achieve by reflecting the desired `MicroSteps` length into a term-level singleton using `toUNat`:

```
padding :: SNat (p + 1) -> MicroSteps (p + 1) False False
padding = go . toUNat
  where
    go :: UNat (p + 1) -> MicroSteps (p + 1) False False
    go (USucc UZero) = step INothing nop INothing
    go (USucc m@(USucc _)) = step INothing nop INothing >++> go m
```

  If we now add 3 cycles of `padding` before checking the condition, it will come out to a total of 5 (including the fetch and checking the condition), as desired:

```
microcode (RETIf cond) = padded (SNat @11) $
    padding (SNat @3)                         >++>
    step INothing (When $ Just cond) INothing >++>
    popPC
```

  Similarly, for `CALLIf`, which should take 11 or 17 cycles:

```
microcode (CALLIf cond) = padded (SNat @17) $
  padding (SNat @7)                  >++>
  imm2                               >++>
  step INothing (When cond) INothing >++>
  pushPC                             >++>
  step INothing Jump        INothing
```

- What about the third conditional instruction, `JMPIf`? Unfortunately, `JMPIf` should take 10 cycles regardless of the condition holding or not. To see why this is a problem, let's try to apply our previous technique to `JMPIf`:

```
microcode (JMPIf cond) = padded (SNat @10) $
    padding (SNat @5)                 >++>     -- (*)
    imm2                              >++>
    step INothing (When cond) INothing >++>
    step INothing Jump        INothing
```

  What should the `padding` amount be on the marked line? If we use 5, that means the un-taken branch takes $1 + 5 + 2 + 1 = 9$ cycles, while the taken one takes $1 + 5 + 2 + 1 + 1 = 10$ cycles. If we use 6, that fixes the early-exit case, but now the case where the branch is taken is too slow. There's just no winning here, because the `Jump` after the `When` will take one more cycle anyway.

  We can work around this by spending an extra cycle after a `When` that evaluates to false. This requires decreasing the `padding` in `RETIf` and `CALLIf`, but these are the only instructions that use `When`. An easy way of wasting an extra cycle before fetching the next instruction is to go to `Init` instead of `Fetching`, since `Init` will, on the next cycle, transition to the `Fetching` state straight away. We can do all this in `exec`, where `GotoNext` is handled:

```
exec :: Value -> Index MicroLen -> CPU ()
exec instr i = do
    let (uop, after) = microcodeFor instr !! i
    runExceptT (zoom microState $ uexec uop) >>= \case
        Left GotoNext -> do
            phase .= Init
        -- Other parts unchanged
```

- For the arithmetic instructions `ALU` and `CMP`, the cycle count differs based on the location of the source and destination arguments. For example, on a real Intel 8080, `ADD r imm` takes 7 cycles, but `ADD r1 r2` takes 4; it makes sense that there would be a difference, since the former needs to do an extra fetch to get the immediate argument's value, whereas the latter has it on the ready in the register `r2`. However, for any chosen source and destination mode, `ALU` and `CMP` has the same length; this means we can just annotate the `padding` calls in `alu` with the right numbers, and it will give us the correct timing for both `ALU` and `CMP`:

```
alu
    :: ALU
    -> RHS
    -> MicroInstr
    -> Microcode
alu fun rhs writeback = case rhs of
    Imm -> padded (SNat @7) $
        step (IJust IncrPC) (Compute RegA fun SetZSP SetAC SetC)
                                                        INothing >++>
        step INothing      writeback                    INothing
    LHS (Reg r) -> padded (SNat @4) $
        step INothing (FromReg r)                       INothing >++>
        step INothing (Compute RegA fun SetZSP SetAC SetC)
                                                        INothing >++>
        step INothing writeback                         INothing
    LHS (Addr rr) -> padded (SNat @7) $
        step INothing        (FromReg2 rr)              INothing >++>
        step (IJust FromPtr) (Compute RegA fun SetZSP SetAC SetC)
                                                        INothing >++>
        step INothing        writeback                  INothing
```

With the padded values filled in throughout the branches of microcode, we have no more work left: padded will insert just the right amount of uNOP micro-steps before the given microcode's continuation flag is cleared. No changes are required on the CPU itself, or even uexec.

18.9 Slowing down the CPU

The original Compucolor II runs at 2 MHz, whereas our machine is locked to the 800 × 600@60 VGA video mode's pixel clock of 40 MHz. Interactive applications (most importantly, games) that use busy waiting instead of timer interrupts will run 20 times faster. Now that we have a way of loading existing software from disks, this problem is immediately apparent with almost any game we care to try out.

Luckily, we don't need to complicate our lives with multiple clock domains to solve this – we can just change our Intel 8080 core to add a pause pin which, when set, inhibits state transitions in the given cycle. Then we can set pause to True in 19 out of every 20 cycles to get the effective CPU speed divided by 20.

The change to our 8080 interface is minor: just a new field in CPUIn:

```
data CPUIn = CPUIn
    { pause :: Bool
    ... }
```

Inside the `cpu` itself, we want to exit prematurely if `pause` is set, i.e. we want to put a `guard $ not pause` somewhere. But where? Let's recall the definition of `cpu`, with the details of the phase-specific execution omitted:

```
cpu :: Pure CPUIn -> CPUM CPUState CPUOut ()
cpu inp@CPUIn{..} = void . runMaybeT $ do
    interrupted <- latchInterrupt inp
    dataRead <- readByte inp

    use phase >>= \case
        ...
```

Peripherals that run at the native 40 MHz clock can issue interrupts at any time, so we need to `latchInterrupt` even when the CPU is otherwise paused. Similarly, memory elements usually (unless the access is contended) respond in the cycle immediately following the read request. With our planned scheme of pausing for 19 cycles out of 20, this means it is almost guaranteed that read results will come back while paused.

So, we should check for `pause` after the data-in bus is sampled, and we're done, right?

```
cpu :: Pure CPUIn -> CPUM CPUState CPUOut ()
cpu inp@CPUIn{..} = void . runMaybeT $ do
    interrupted <- latchInterrupt inp
    dataRead <- readByte inp

    guard $ not pause

    use phase >>= \case
        ...
```

Turns out it's not that easy: the return value of `readByte` isn't used for anything in the paused case, so the read result is lost by the time `pause` is unset and we're ready to make progress. Clearly, we need a way to latch read bytes the same way we latch interrupts, accumulating while paused. To do this, we add a new bit of state that is the data-in pair of `dataOutLatch`:

```
data CPUState = CPUState
    { _dataInLatch :: Maybe Value
    ...
    }
```

```
initState pc0 = CPUState
    { _dataInLatch = Nothing
    ...
    }
```

This latch is updated from the current `dataIn` in `readByte`, basically implementing a one-element FIFO on memory reads. For reference, the non-latching version of `readByte` was as follows:

```
readByte :: Pure CPUIn -> CPU Value
readByte CPUIn{..} = do
    pending <- isJust <$> use addrLatch
    if pending then maybe retry consume dataIn
        else return $ fromJustX dataIn
  where
    retry = mzero
    consume x = do
        addrLatch .= Nothing
        dataOutLatch .= Nothing
        return x
```

In the new version, we still want to `retry` immediately if there's a `pending` memory request and nothing has come back yet; but if we do have a `dataIn` value, we want to store it in `dataInLatch` (if the latter is still empty):

```
readByte CPUIn{..} = do
    pending <- isJust <$> use addrLatch
    current <- if pending then Just <$> maybe retry consume dataIn
        else return dataIn

    latched <- use dataInLatch
    let latest = current <|> latched
    dataInLatch .= latest

    return $ fromJustX latest
```

Of course, we also need to clear `dataInLatch` when the CPU is not paused, leading to the following final version of `cpu`:

```
cpu :: Pure CPUIn -> CPUM CPUState CPUOut ()
cpu inp@CPUIn{..} = void . runMaybeT $ do
    interrupted <- latchInterrupt inp
    dataRead <- readByte inp
    guard $ not pause
    dataInLatch .= Nothing

    use phase >>= \case
        ...
```

Jumping back into the Compucolor II itself, we can define the `pause` signal in `mainBoard` using an inverted `riseEvery`, effectively turning it into a `fallEvery` function:

```
mainBoard kbdCols frameEnd vidRead = (kbdRow, crtOut, vidAddr, vidWrite)
  where
    pause = not <$> riseEvery (SNat @20)
    ...
```

With this change, we also need to re-define `FastRate` as `SlowRate * 8` instead of the hacked `SlowRate * 8 * 20`, otherwise the sector start detection would fail for the same reason as before: with the CPU running at 2 MHz, the firmware's assumption of each serial bit taking 26 CPU cycles can only be met if the bits actually take $26 \cdot 20 = 520$ real cycles.

18.10 Our complete computer

With the floppy drive and CPU timing issues out of the way, our machine is complete and it is time to take stock. Our Compucolor II is faithful to the original in many important ways:

- The CPU is compatible and accurate to instruction cycle count with the Intel 8080.
- All functionality of the TMS 5501 that is used by the Compucolor II is implemented.
- The contents of the video buffer is interpreted correctly, including plot mode and the values of the attribute bytes.
- Our CRT 5027 implements the dynamic video features used by Compucolor II programs.
- The virtual floppy drive can be used to load programs from the library of pre-existing Compucolor II software.

However, there is still some room for improvement:

- Just like its predecessors the Intercolor 8001 and the Compucolor 8001, the original Compucolor II could also be used as a serial terminal, using the

TMS 5501's non-testing mode to implement 9600 bps UART communication. To support this, we need a way to switch from the 9600 × 8 × 20 bps rate of the floppy drive.

- Our video signal generator uses a fixed 64 × 32 character mode with 6 × 8 fonts; whereas the real CRT 5027 supports a wide range of character counts and glyph sizes. Although on the Compucolor II, these settings cannot be changed without potentially damaging the screen circuit, a fully reusable CRT 5027 component would need to be more flexible.

We leave these, and some smaller, more self-contained improvement ideas as exercises to the reader.

Exercises

- Although glyphs are 6 pixels wide, we use 8-bit ROM for the font and the plot ROM. Change these to only store and return a `BitVector 6`. Note that this will also require extending `binLines` to prepare ROM image files with non-byte-sized contents.

- Running the CPU at 2 MHz is accurate to the Compucolor II's speed, but for computationally intensive programs, it might be better to flip back to the native 40 MHz. For example, some games that generate randomized maps might take tens of seconds to initialize each time the game is restarted. Add a hardware "turbo" switch that temporarily turns off the `pause` signal. For bonus points, change the TMS 5501 and the floppy drive to also be aware of this setting, allowing disk I/O to work both in turbo and in normal mode.

- Implement writing to floppy disks. This requires adding a serial transmitter to the TMS 5501, and using its output to overwrite the current bit in the floppy drive when bit 3 of the TMS 5501's parallel output is low.

- For virtually removable disks, store multiple images in multiple block RAMs, and use some buttons on the FPGA board to switch between them.

- A more ambitious project is to implement physically removable media for the floppy drive using an SD card.

- With all instructions stored as 17 micro-steps, our microcode is now more wasteful than ever. Redesign the microcode ROM representation to use a linked list, which means not only do we not need to store the trailing `uNOP`s, we can even unify common microcode suffixes across instructions by using the same next address at multiple points.

18.11 Summary

- Reimplementing a general-purpose computer requires more fidelity than implementing an abstract design like the CHIP-8 or a computer designed for a narrow use case like Space Invaders, because of the open world nature of existing software that was written with the fine details of the hardware in mind.

- In particular, we've seen an example where the cycle count of each CPU instruction needs to match the original hardware implementation, to get some timing-sensitive parts of the firmware to work.

- We could reuse our Intel 8080 implementation (after improving its timing fidelity), our VGA timer, and our UART; but some parts, most notably the TMS 5501, are a collection of ad-hoc functionality that required specific code.

- Virtual disk storage can be implemented with a piece of memory with a pointer: the geometry of the disk is implemented by cyclically scanning the pointer over an address range; the disk spin speed corresponds to the rate of advancing the pointer; and head servo commands can be interpreted by changing the current address range.

Parting words

The 1970s started with the introduction of cheap, integrated circuit-based pocket calculators and ended with home computers that made it affordable for individuals to own a complete computer system. On the way there, video games and arcade machines changed the entertainment landscape for generations.

Our journey has taken us through roughly the same stops, hitting the same story beats; but this version is a modernized retelling of this story, in a (literally) more modern language.

Digital electronics hardware has advanced in leaps and bounds since the seventies. With this advancement came more complexity, and less overall visibility for users and developers into what exactly goes on inside their computer. However, these advancements are also making it possible for the individual hobbyist to recreate these machines of the past on a sub-$100 FPGA development board, achieving the deepest level of understanding a computer possible, short of worrying about the quantum mechanical properties of individual transistors.

The progress in hardware has also been instrumental in unlocking the full potential of higher-level programming languages like Haskell. 1985's Miranda couldn't possibly have run on the Commodore 64 released in 1983; in contrast, by the time the Haskell 98 standard was finalized, the performance of the widespread i386-based personal computers of the late nineties were wholly adequate to run Haskell 98 implementations like GHC 4.02.

At the time of writing this, the seventies ended 41 years ago, just as this author was born. Maybe there will never be again a generation that just stumbles onto the scene of computers and grows up not just *with* them, but *in tandem* with them. However, hopefully this book helps mitigating that somewhat by explaining the history of this era with hardware descriptions that are readable, high level, executable, and synthesizable at the same time — putting the *fun* of *functional programming* to use in the domain of hardware.

Index

Haskell has become the functional programming language of choice for many developers due to its excellent tools for abstraction and principled program design. The open source Clash hardware description language now brings these features to FPGA development.

Retrocomputing with Clash takes the experienced Haskell programmer on a journey into the world of hardware design with Clash. Our approach is based on using Haskell to its fullest potential, using abstractions like monads and lenses in building a library of reusable components.

But that wouldn't put the *fun* in *functional programming*! And so we put these components to good use in implementing various retro-computing devices:

- Pocket calculator
- Pong
- A simple, but Turing-complete computer that uses Brainfuck as its machine code
- An implementation of the CHIP-8 virtual computer specification
- Intel 8080 CPU
- Space Invaders arcade machine
- Compucolor II, a home computer from 1977 complete with keyboard, color, video, and a floppy drive

> *I absolutely love the very Haskell approach to circuit design in this book, as opposed to my own write-Verilog-in-Haskell style. It leverages Haskell's type system in a very natural way to protect against many traps we as circuit designers often fall into.*
> *The book clearly demonstrates the benefits of using a modern programming language for circuit design, where it builds reusable functionality and components at a far finer granularity than what I'm used to in traditional hardware description languages.*
>
> – Christiaan Baaij,
> Clash lead developer, QBayLogic co-founder

> *This is the book for functional programmers looking to get into FPGAs and digital logic design. Learn Clash, the "I cant believe it's not Haskell!" hardware description language, while indulging in nostalgia for the 1980s. Take a joyride through a variety of hands-on projects, including Pong, Space Invaders, and the Compucolor II, a personal computer based on the Intel 8080. Recommended.*
>
> — Miëtek Bak,
> mathematician

https://unsafePerform.IO/retroclash/