Designing Digital Circuits
a modern approach

Jonathan Turner

Contents

I First Half **5**

1 Introduction to Designing Digital Circuits **7**
1.1 Getting Started 7
1.2 Gates and Flip Flops 9
1.3 How are Digital Circuits Designed? 10
1.4 Programmable Processors 12
1.5 Prototyping Digital Circuits 15

2 First Steps **17**
2.1 A Simple Binary Calculator 17
2.2 Representing Numbers in Digital Circuits 21
2.3 Logic Equations and Circuits 24

3 Designing Combinational Circuits With VHDL **33**
3.1 The entity and architecture 34
3.2 Signal Assignments 39
3.3 Processes and if-then-else 43

4 Computer-Aided Design **51**
4.1 Overview of CAD Design Flow 51
4.2 Starting a New Project 54
4.3 Simulating a Circuit Module 61
4.4 Preparing to Test on a Prototype Board 66
4.5 Simulating the Prototype Circuit 69

4.6 Testing the Prototype Circuit 70

5 More VHDL Language Features **77**
5.1 Symbolic constants 78
5.2 For and case statements 81
5.3 Synchronous and Asynchronous Assignments 86
5.4 Structural VHDL 89

6 Building Blocks of Digital Circuits **93**
6.1 Logic Gates as Electronic Components 93
6.2 Storage Elements 98
6.3 Larger Building Blocks 100
6.4 Lookup Tables and FPGAs 105

7 Sequential Circuits **109**
7.1 A Fair Arbiter Circuit 110
7.2 Garage Door Opener 118

8 State Machines with Data **127**
8.1 Pulse Counter 127
8.2 Debouncer 134
8.3 Knob Interface 137
8.4 Two Speed Garage Door Opener 141

II Second Half **147**

9 Still More VHDL **149**
9.1 Making Circuit Specifications More Generic 149
9.2 Arrays and Records 152
9.3 Using Assertions to Detect Bugs 155
9.4 VHDL Variables 156
9.5 Functions and Procedures 159

10 Design Studies **163**
10.1 Four-way Max Finder 163
10.2 Binary Input Module 168
10.3 LCD Display Module 172
10.4 Binary Output Module 175

11 Verifying Circuit Operation **179**
11.1 Assertion Checking in Circuit Specifications 180
11.2 Testing Combinational Circuits 181
11.3 Testing State Machines 187
11.4 Testing Larger Circuits 195

12 Continuing Design Studies **197**
12.1 Simple Data Queue 197
12.2 Packet FIFO 202
12.3 Priority Queue 212

13 Small Scale Circuit Optimization **219**
13.1 Algebraic Methods 220
13.2 Algorithmic Methods 224

14 Still More Design Studies **235**
14.1 VGA Display Circuit 235
14.2 Mine Sweeper Game 245

15 Implementing Digital Circuit Elements **265**
15.1 Gates and Transistors 265
15.2 Delays in Circuits 272
15.3 Latches and Flip Flops 275

16 Timing Issues in Digital Circuits **281**
16.1 Flip Flop Timing Parameters 281
16.2 Metastability and Synchronizers 290

III Third Half 295

17 Introduction to Programmable Processors 297
17.1 Overview of the WASHU-2 Processor 297
17.2 Machine Language Programming 302
17.3 Prototyping the WASHU-2 309
17.4 Using Subprograms 312

18 Implementing a Programmable Processor 321
18.1 Overview of the Implementation 321
18.2 Signal Timing 323
18.3 VHDL Implementation 326

19 Supporting Components 337
19.1 Overview of the Complete System 337
19.2 Implementing the Console 341

20 Memory Components 353
20.1 SRAM Organization and Operation 353
20.2 Alternate Memory Organizations 360
20.3 Dynamic RAMs 362

21 Improving Processor Performance 365
21.1 A Brief Look Back at Processor Design 365
21.2 Alternate Instruction Set Architectures 367
21.3 Implementing the WASHU-16 372

22 Improving Processor Performance Even More 387
22.1 Cache Basics 388
22.2 A Cache for the WASHU-2 391
22.3 Beyond Direct-Mapped Caches 394
22.4 Other Ways to Boost Performance 397

23 Making Circuits Faster **399**
23.1 Faster Increment Circuits 399
23.2 Faster Adder Circuits 403
23.3 Other Linear Circuits 405
23.4 Multiplication Circuits 408

24 Producing Better Circuits Using VHDL **413**
24.1 Some Motivating Examples 414
24.2 Estimating Resource Usage 418
24.3 Estimating and Reducing Resource Usage 421

Part I

First Half

Chapter 1

Introduction to Designing Digital Circuits

1.1 Getting Started

This book is all about the design of digital circuits. So what exactly are digital circuits and why should we care about them? Let's start with the second part of that question. Simply put, digital circuits have become a ubiquitous and indispensable part of modern life. They are in our computers, our cell phones, our cars, our televisions, our wrist watches. Almost everywhere you look, you can find digital circuits, and new applications are being developed all the time. Surprisingly, this is a fairly recent phenomenon. In 1960, digital circuits were just beginning to find commercial application and very few people would ever encounter one in their daily lives. By the mid 1970s, hand-held calculators were starting to become popular with scientists, engineers and students, and by the mid 1980s personal computers started to get widespread use. Since then, the growth in the use of digital circuits has been explosive, and today it's hard to imagine living without them.

So how is it that digital circuits have become such a big deal in such a short time? There are two key inventions that have driven the digital revolution. The first was the invention of the transistor in the late 1940s, and the second was the invention of the integrated circuit in the late 1950s.

Now, transistors are the essential building block used to construct digital circuits, and integrated circuit technology is a manufacturing process that allows many transistors to be fabricated at once and wired together to create complex circuits. While early integrated circuits contained just a handful of transistors, advances in the fabrication processes now allow us to produce circuits with billions of transistors on a silicon chip the size of a fingernail.

Now there is another big reason that digital circuits have become so successful, and that brings us to that word "digital". The defining property of a digital circuit is that it uses voltages and currents to represent logical values, commonly denoted as '0' and '1'. Now what's important about this is that because digital circuits represent logical values, it's possible to combine the basic building blocks of a digital circuit using just the rules of logic, and the rules of logic are a whole lot simpler than the laws of physics that ultimately determine how circuits behave. This gives digital circuits a kind of *modularity* that more general analog circuits lack. It is that modularity that allows us to create circuits of mind-boggling complexity that do what we expect them to do, reliably and consistently. Now this is not to say that we can escape the laws of physics entirely. Physical properties do place some constraints on how digital circuit components can be combined and the speed with which they operate. Nonetheless, when designing digital circuits we can largely ignore the underlying physics and focus most of our attention on how to combine components in a way that produces a desired logical behavior.

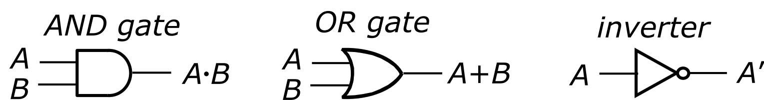
There is still another big reason that digital circuits have become so successful and that is the *programmable processor*, arguably the most important digital circuit of all. What makes it so important is its remarkable flexibility, and the key to that flexibility is *programmability*. While a processor is just a single circuit, it can be programmed to implement a remakable diversity of functions. This programmability means that one device can do many different things. So the same processor can be used in a wrist watch or a calculator, and at different times it can do different things, as the plethora of cell phone apps amply demonstrates. Perhaps the most amazing thing about the progammable processor is that it can be a relatively simple circuit. Programmability does not require a lot of complexity and while modern processors are pretty complex, we'll see that most of that complexity is there

for the sole purpose of improving performance. The essential feature of programmability does not require it.

As you progress through the book, you will learn about the building blocks of digital circuits and how they can be put together to build complex systems. You will learn about how circuits can be constructed efficiently using the rules of logic, and how modern *hardware description languages* can be used to simplify the specification of larger circuits. You will learn how circuit designs are debugged using a circuit simulator and how programmable logic devices can be used to implement the circuits you design. You will see lots of different examples of digital circuits and have the opportunity to develop your own digital design skills. As you move on to the later chapters, you will learn how to implement a programmable processor and how it can be programmed using a simple assembly language. You wil also learn about the factors that limit processor performance, and how processor designers overcome these limitations using a variety of different techniques.

1.2 Gates and Flip Flops

Figure **??** shows the three fundamental components of a digital circuit, the AND *gate*, the OR *gate* and the *inverter*. As its name suggests, the output

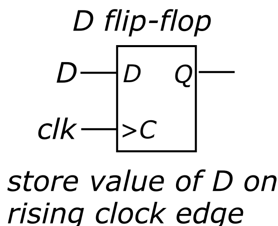


of an AND gate is the logical AND of its two inputs. That is, if both of the inputs are ‘1’, the output is ‘1’; otherwise, the output is zero. We use the symbol ‘·’ to denote the logical AND operation. Similarly, the output of the OR gate is ‘1’ if either of its inputs is ‘1’; otherwise the output is ‘0’. We use the plus symbol ‘+’ to denote the OR operation. Finally, the output of the inverter is just the logical complement of its input, and we will use the prime symbol to denote the logical complement.

Note that the inputs to these devices can change at any time, and as the inputs change, the outputs change in response. For now, we will view

gates as idealized devices that respond instantaneously to changes at their inputs. Later, we'll see see that real physical gate implementations do require a small, but non-zero amount of time to respond to input changes. This has important important practical implications for how we design circuits to obtain the best possible performance.

The figure below shows another key building block of digital circuits, the *flip flop*.



A flip flop is a storage device that holds a single logical value. There are actually several different types of flip flop, but the most common is the simple D flip flop shown here. A D flip flop has two inputs, the *data input* (labelled 'D') and the clock input (labelled '>C'). The output (labelled 'Q') is always equal to the value stored in the flip flop. The stored value changes when the clock input changes from '0' to '1'. More precisely, on each rising clock transition, the value that appears on the D input is stored in the flip flop. That value will remain unchanged until the next rising clock transition. In a later chapter, we will see how flip flops can be implemented using gates, but for now we will simply treat the flip flop as a basic building block. Remarkably, the three gates and the D flip flop are all one needs to construct any digital circuit. As we proceed, we'll see many examples of how they can be combined to produce a variety of interesting circuits.

1.3 How are Digital Circuits Designed?

Ultimately, a digital circuit is just a collection of gates and flip flops that are connected together. So, the essence of the design process is to specify exactly what gates and flip flops should be used and how they should be connected in

order to produce a circuit that behaves in a desired way. One way to specify a circuit is using a *schematic diagram* that shows a collection of components, that are connected to one other with lines drawn on the page. Designing digital circuits using hand-drawn diagrams is an entirely reasonable thing to do for circuits of moderate size, and indeed most digital circuits were designed in exactly this way until well into the 1980s. As the complexity of digital systems has increased, *computer-aided design tools* were developed to reduce the amount of manual effort required to specify circuits and verify that they worked correctly.

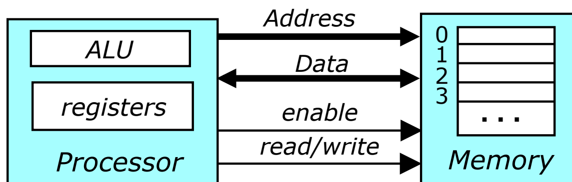
The first computer-aided design tools simply replaced hand-drawn diagrams with diagrams that were drawn using a graphical editor. This did not greatly simplify the designer's job, but did make it possible to automate the later steps in the process required to turn the designer's logic diagram into a collection of transistors on an actual integrated circuit device. A bigger step forward was the introduction of *hardware description languages* that allowed circuit designers to specify circuits using a language similar to the high level programming languages used for computer software. While the net result of the design process was still a circuit composed of gates and flip flops, the use of higher level languages allowed designers to focus most of their attention on the behavior they wanted their circuits to have, and worry less about the details of how the circuit components are connected to each other.

The other big advance in computer-aided design was the development of *circuit simulators*. A simulator is a piece of software that mimics the behavior of a circuit and provides a visual display to allow a designer to observe how internal signals change as the circuit operates. Simulators are powerful tools that allow designers to see the effects of errors in their circuit designs and track those errors back to their underlying causes. They can be used to construct comprehensive test suites, known as *testbenches*, to verify that circuits operate correctly under a wide range of conditions. Thorough simulation-based testing of circuits is extremely important, because the costs of errors in manufactured circuits can be extremely high. One production run of an integrated circuit can cost millions of dollars and delay the launch of a new product by many months, so it is important to get it right the first time. Errors discovered after a product is sold are even more costly, often

requiring product recalls and rebates to unhappy customers.

1.4 Programmable Processors

As mentioned before, the programmable processor is arguably the most important digital circuit of all. in a later chapter, we will introduce a basic programmable processor called the *WashU-2*. Here we give a brief preview to whet your appetite for what's to come. The figure below shows a generic processor and an attached memory device.



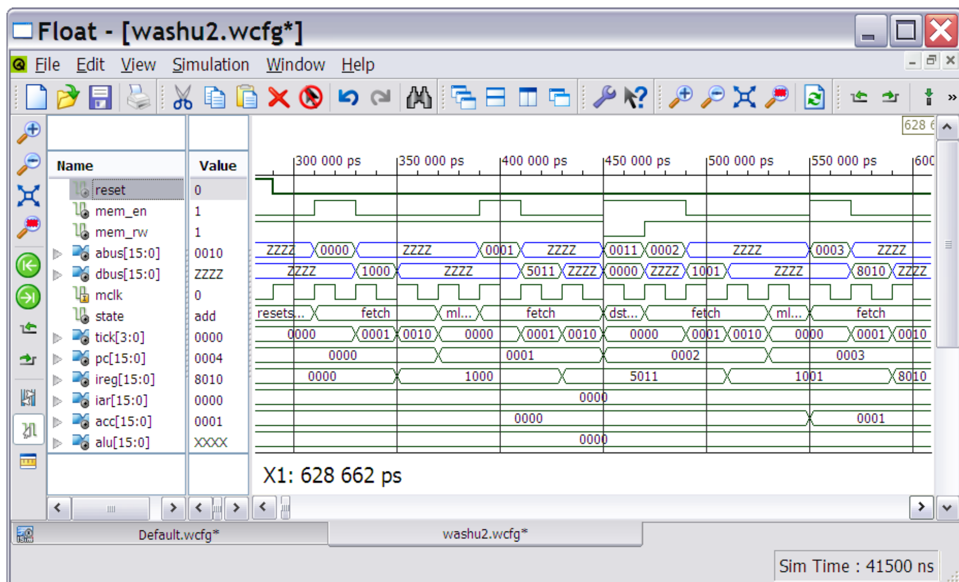
The memory can be viewed as a one-dimensional array of storage locations, each of which is identified by an integer index, called its address. Each location can store a fixed-length integer (in modern processors, they are typically 32 or 64 bits). Memory addresses are also fixed-length integers and the number of memory locations is limited by the number of bits used to specify an address. There are several signals that connect the processor to the memory. To retrieve a value stored in the memory, the processor enables the memory (using the *enable signal*) holds the *read/write* signal high and places the address of the desired memory location on the address lines. The memory device responds by putting the value stored at that location on the data lines. To write a value to the memory, the processor enables the memory, holds the read/write signal low, puts the address of the location to be written on the address lines and the data value to be written on the data lines.

The processor includes some internal storage locations called *registers*. Some registers have special functions, while others are referred to as general-purpose registers. These are typically used to hold intermediate data values

during the course of a longer computation. Modern processors typically have hundreds of registers, but large register sets are not essential. Indeed the WashU-2 has just four registers, only one of which can be viewed as general-purpose. Many processor instructions use a component called the *Arithmetic and Logic Unit* (ALU), which implements a variety of common functions, such as addition, subtraction, logical AND/OR and so forth.

The memory is used to store two types of information: *instructions* and *data*. Data refers to values that a program operates on, and the instructions specify how the data is processed. A typical instruction might add two numbers together, or transfer a value between the processor and the memory. The processor repeatedly performs a basic processing cycle in which it *fetches* an instruction from memory and then *executes* that instruction. Of course each instruction is just a set of bits, but the processor interprets those bits as an instruction specification. So for example, the first few bits of an instruction might specify the type of operation to be performed (addition, subtraction, . . .) while the remaining bits specify the location of any data it requires. This simple repetitive process is at the core of any programmable processor. Because programs can consist of arbitrary instruction sequences, processors can implement an endless variety of different functions. The power of a programmable processor comes from the flexibility that results from this simple repetitive cycle and the tremendous speed with which the cycle can be repeated (billions of times per second in modern computers).

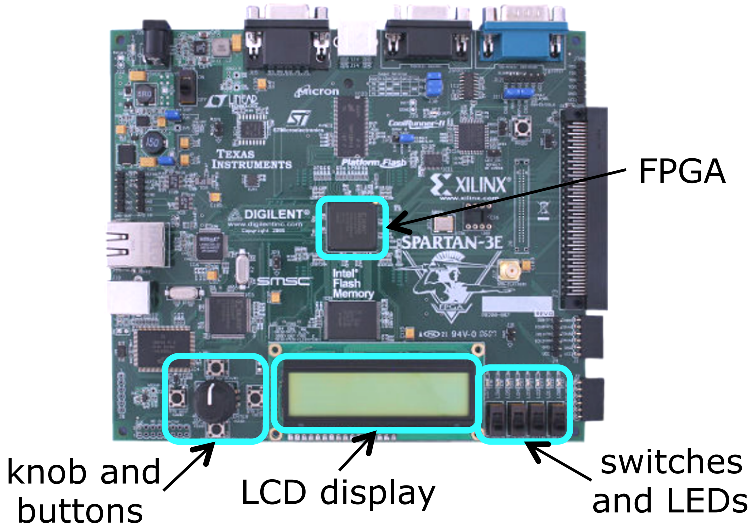
The figure below provides a peek into the internal operation of the WashU-2. Specifically, it shows a *waveform display* from a simulation of a running processor.



Signal names appear in the left part of the window and the main area shows how the values of different signals changes over time. The first few signals are the ones connecting the processor and the memory. By observing these signals, we can see values being transferred between the processor and specific memory locations. For example, the first time the enable signal goes high, the hexadecimal value 1000 is read from location 0, while the third time the enable goes high, the value 0000 is written to location 0011. The signal labeled *state* shows what the processor is doing at each point in time. Note the repeated instruction fetches. Between each pair of fetches an instruction is executed, although in this waveform display only the first two or three letters of the instruction name appear. The four processor registers are the *Program Counter* (PC), the *Instruction Register* (IREG), the *Indirect Address Register* (IAR) and the *Accumulator* (ACC). These appear towards the bottom of the window. When we study the processor in depth, we will see how the registers are used to implement the Washu-2's basic fetch-and-execute cycle.

1.5 Prototyping Digital Circuits

After a digital circuit is designed, it must be manufactured, usually in the form of an integrated circuit. The manufacturing process for modern integrated circuits is a complex and sophisticated one, and it can cost millions of dollars to produce even small quantities of a new circuit. To reduce the likelihood of errors in a manufactured circuit, designers often *prototype* their designs using a special kind of integrated circuit called a *Field Programmable Gate Array* (FPGA). An FPGA is a circuit that can be "programmed" to mimic the behavior of another circuit. Since programming an FPGA incurs no significant cost, testing a new circuit using an FPGA provides an effective way of avoiding more costly mistakes at a later point in the design process. Indeed, in some applications FPGAs are used not just for prototyping, but in the final manufactured products. While FPGAs cannot match the performance and complexity of custom integrated circuits and are not always cost-effective in high volume applications, they can be a good solution in many situations. In this book, we will be use a specific FPGA prototyping board to show how our designs can be implemented and tested. The following figure shows a photograph of such a board.



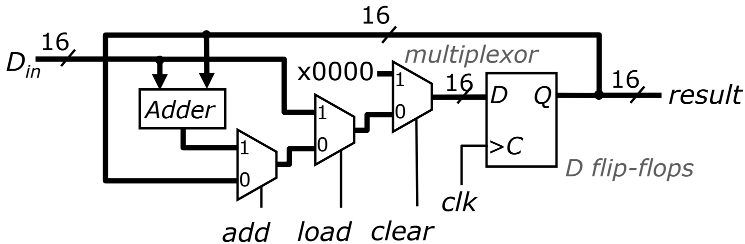
Several of the key features of the board are highlighted in the figure. First notice the LCD display at the bottom. This can be used to display two lines of 16 characters each. When we use the board to prototype the WashU-2, we'll use the display to show values in the processor's registers and a selected memory location. The board also includes several switches and buttons that can be used to interact with whatever circuit is implemented by the board. When using the board to prototype the WashU-2, we'll use these to stop the processor's execution and then *single-step* through one instruction at a time, observing how each instruction changes the values of the registers. We'll also use them to examine any memory location and change the value stored at some location. This provides a crude, but effective way to input small amounts of data to a running program. The board also has a VGA connector that can be used to drive an external display. There are also inputs for a keyboard and mouse.

Chapter 2

First Steps

2.1 A Simple Binary Calculator

Our first digital circuit is a (very) simple binary calculator. You can do three things with it. You can load an input value and store it in the calculator's internal storage register, you can add an input value to the stored value, and you can clear the stored value. Not very exciting perhaps, but it will illustrate many of the key features that can be found in more sophisticated circuits. The following figure is a diagram of the calculator circuit.



This circuit operates on 16 bit values. The label '16' on the heavy lines connecting many of the circuit components indicates that these lines actually represents 16 individual signals. So the *D*-flip flop symbol shown at the right end of the diagram actually represents 16 flip flops, not just one. A

collection of flip flops like this is commonly referred to as a *register*. The three trapezoidal symbols in the figure represent *mulitiplexors*. They each have a control input and two data inputs, labeled '0' and '1'. When the control input of a multiplexor is low, the 0-input is connected to the output on the right. When the control input is high, the 1-input is connected to the output. This allows us to conveniently select one of two different input values. Note that while the data inputs and outputs in this diagram are 16 bits wide, the control input is a single signal. This means that each multiplexor symbol shown in the diagram actually represents 16 multiplexors, all of which are controlled by the same input signal.

The block labeled *adder* has two inputs and a single output which is equal to the sum of the two inputs. Note that both inputs and outputs are 16 bits long, so it's possible for the sum of the input values to be too large to represent in 16 bits. In this case, the high-order bit of the sum is simply lost. This is an example of *arithmetic overflow* which is an unavoidable property of binary arithmetic with fixed-size words.

Note that the adder and multiplexor can both be implemented using nothing but AND gates, OR gates and inverters. While we could show their detailed implementation here, the diagram illustrates a common practice in digital circuit diagrams, in which commonly recurring sub-circuits are represented by labeled blocks. Indeed, the multiplexor is such a commonly recurring building block that it has its own special symbol. Other components are shown like the adder with simply a rectangular block and a label. Now that we've "decoded" the diagram, let's look at it again to understand exactly how it works. First, notice the data inputs at the left and the result output at the right. Next, notice the three control inputs, *add*, *load* and *clear*. To clear the contents of the register, we raise the *clear* input high. This causes the value on the rightmost multiplexor's 1-input to be connected to its output. Since the output connects to the D input of the flip flops, the hexadecimal value 0000 will be loaded into the register on the next rising clock transition. To load a new value into the calculator, we leave the clear input low, while holding the load input high. The high on the load input connects the D_{in} signal to the output of the middle multiplexor, while the low on the clear input connects the output of the middle multiplexor to

the output of the rightmost multiplexor. Consequently, the input value will be loaded into the register on the next rising clock edge. Notice that since the circuit's output is simply the output of the register, the newly loaded value will also appear at the circuit's output. If this output is connected to a display circuit, like the one on our FPGA prototype board, we'll be able to observe that value. Finally, to add a value to a previously stored value, we raise the calculator's add input high, while holding the other two control inputs low. Since the output of the adder is the sum of the input value and the stored value, this causes the sum to be loaded into the register on the next rising clock edge.

Now for a small circuit like this, it's entirely reasonable to specify the circuit using a diagram like this, but for larger circuits, we can save a lot of time and effort by using a hardware description language. In this course, we'll use the language VHDL to specify circuits, so let's take a look at how we can use VHDL to specify the calculator circuit.

```
entity calculator is port (
    clk: in std_logic;
    clear, load, add: in std_logic;  -- operation signals
    dIn: in std_logic_vector(15 downto 0);  -- input data
    result: out std_logic_vector(15 downto 0)); -- output
end calculator;

architecture a1 of calculator is
signal dReg: std_logic_vector(15 downto 0);
begin
    process (clk) begin
        if rising_edge(clk) then
            if clear = '1' then
                dReg <= x"0000";
            elsif load = '1' then
                dReg <= dIn;
            elsif add = '1' then
                dReg <= dReg + dIn;
            end if;
```

```
            end if;
        end process;
        result <= dReg;
    end a1;
```

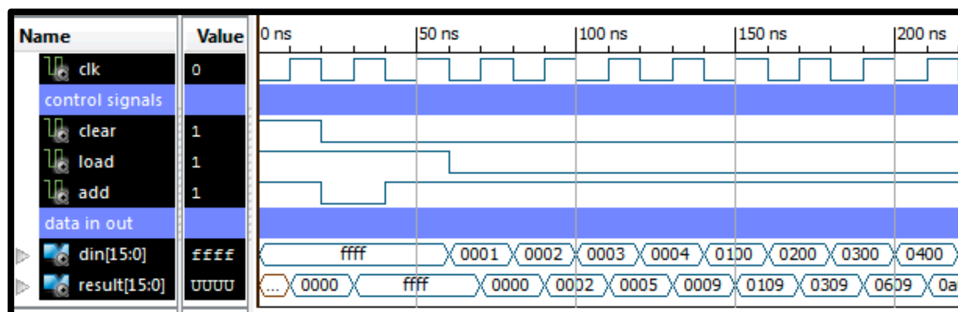
The first six lines of this example form the *entity specification* for the circuit. This part just specifies the inputs and outputs to the circuit, specifically their names and their data types. The single bit signals (clk, clear, load, add) all have type `std_logic`, while the multi-bit signals have type `std_logic_vector`. Note that each individual signal in a `std_logic_vector` can be identified using an integer index, and the *index range* is defined by the phrase "`15 downto 0`".

The remainder of the VHDL code is called the *architecture*. This part specifies the actual internal circuitry. First notice the *signal* definition on the second line of the architecture. This defines a 16 bit internal signal called `dReg`. The third line of the architecture is the start of the "body" of the architecture, and in this case the first thing in the body is the start of a `process` block. Within the process block there is an `if-elsif` block that specifies the value of the `dReg` signal under various conditions. The lines that begin `dReg <= ..` are *signal assignment* statements. Notice that all the code in the process is contained within an initial if-statement that has the condition `rising_edge(clk)`. This specifies that the signal assignements that occur within the scope of this if-statement are to take effect only when the `clk` signal makes a low-to-high transition. In this example, this implies that the `dReg` signal must be the output of a register. The last line of the architecture is a signal assignment that connects the internal signal `dReg` to the output signal `result`. Note that this line is outside of the process. In general, the body of a VHDL architecture may include stand-alone assignments like this, plus more complex blocks of code that appear inside process blocks.

Take a few minutes to carefully compare the VHDL code with the circuit diagram discussed earlier. Make sure you understand how each element of the diagram is reflected in the VHDL. In particular, look closely at the if-elsif block and its correspondence to the sequence of multiplexors.

The figure below shows the waveform display from a simulation of the

calculator circuit.



Observe how the output responds to the various control signals and how the output changes only on the rising edge of the clock signal.

2.2 Representing Numbers in Digital Circuits

As we have seen in the calculator, digital circuits represent numeric values using collections of binary signals. With 16 individual signals, we can represent binary numbers from 0 up to $2^{16}-1$. Just as a quick reminder, numbers are defined in binary notation in much the same way as in decimal notation. For example, the binary notation 110.01 means

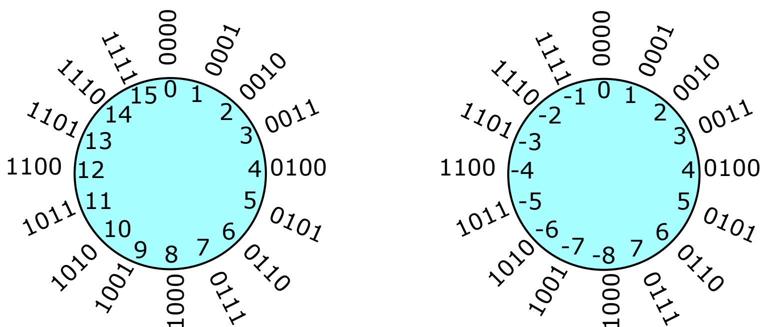
$$1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 4 + 2 + 0 + 0 + 1/4 = 6.25$$

While binary values are convenient for circuits, they are a bit awkward for people. When dealing with binary numbers it's usually more convenient to represent them in *hexadecimal notation* or base 16. Hexadecimal is essentially just a short-hand notation in which groups of 4 bits are represented by one of 16 characters, as shown in the table below.

binary	hex character	binary	hex character
0000	0	1000	8
0001	1	1001	9
0010	2	1010	a
0011	3	1011	b
0100	4	1100	c
0101	5	1101	d
0110	6	1110	e
0111	7	1111	f

So the binary value 1101 0110 0010 is written as d62 in hex, and the hex value ab5 is equivalent to the binary value 1010 1011 0101.

Earlier we mentioned the issue of arithmetic overflow, which can occur when adding two binary values together. This happens because finite circuits cannot implement arbitrary precision arithmetic. Instead, they implement a special kind of arithmetic called *modular arithmetic*, which is defined over a bounded set of values. To understand how this works, it helps to consider a small example. The left side of the figure below shows the 16 four bit binary values arranged in a circular fashion.



The central disk shows the decimal equivalents of these binary values, and we'll refer to this diagram as the *binary number clock* since it kind of resembles an analog clock face. Addition in this binary arithmetic system can be defined as counting our way around the binary number clock. So for example, if we want to add 3 to 6, we start at 0110 on the clock face

and count three positions (in the clockwise direction), which brings us to 1001, which of course corresponds to 9. If we add 3 to 15, the addition is defined in exactly the same way, except that in this case, the result we get is 2, not 18. In a 4 bit modular arithmetic system this is the right result, but of course it does not correspond to what we normally expect. Since the expected answer cannot be represented in our 4 bit number system, this can't really be helped, but it is something that we need to be aware of. Notice that these arithmetic "errors" occur anytime an additon (or subtraction) operation crosses the boundary between 15 and 0.

Now circuits that actually implement arithmetic on binary numbers do not do it by counting around the binary clock face. This is merely a way for us to think about the essential meaning of the arithmetic operations. The circuits that add and subtract binary values are actually based on the same method we all learned in elementary school for doing integer arithmetic. We will defer discussion of exactly how this is done until later in the course.

What about negative numbers? It turns out that there are several different ways that negative numbers can be represented in binary notation, but the one that has become most widely used is called 2s-complement. How ever negative numbers are represented, it's necessary to divide the available bit patterns roughly in half, using some bit patterns for positive numbers and some for negative numbers. The right-hand side of the figure above illustrates a four bit 2s-complement system. Notice that inside the left-half of the "clock-face", we've replaced the positive integers 8 through 15, with the negative integers −8 through −1. Observe that the negative numbers start just to the left of 0 on the clock face and proceed in a *counter-clockwise* fashion. Continuing with the the clock analogy, we can view −1 as representing one minute before the hour, −2 as two minutes before the hour, and so forth.

So how can we do arithmetic in this 2s-complement system? The answer to this is "exactly like we did before." If we want to add 3 to −5, we start at 1011 and count three steps in a clockwise direction. This brings us to 1110, or −2. To add 3 to −1, we start at 1111 and count three steps around, coming to 0010, or 2. Notice that this last example is the one that gave us an error when we were doing an ordinary "unsigned" addition. Here, it produces the correct result. It's still possible to get arithmetic errors in

a 2s-complement system, but in this case the errors occur when we cross the boundary between the positive and negative numbers at the bottom of the clock face. One reason that the 2s-complement system has become the standard way to represent negative numbers is that the same circuits used for unsigned arithmetic, can also be used for signed arithmetic. The only difference in doing arithmetic in these two systems, is deciding when an arithmetic error has occurred.

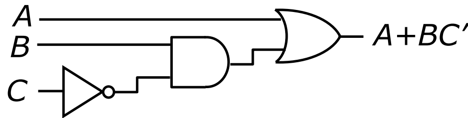
Before we leave the subject of 2s-complement, notice that all the negative numbers start with '1' while the non-negative numbers start with '0'. Because of this, the first bit of a 2s-complement number is referred to as the *sign-bit*. When doing signed arithmetic, there is one extra operation that is needed, the *negation operation*. So, how do we find the negative equivalent of a 2s-complement value? Well one way to do this is to subtract from 0. So for example, we can determine the bit pattern that corresponds to -3 by starting at 0000 on the clock face, and counting three positions in a counter-clockwise direction. It turns out that there is a more direct method for negating a 2s-complement value. We start by finding the rightmost 1 bit, then we flip all the bits to its left. So for example, the negative equivalent of 0001 is 1111 (notice that the negation procedure works in both directions). Similarly, the negative equivalent of 1010 is 0110 (be sure to check this against the clock face).

2.3 Logic Equations and Circuits

As we noted earlier, the defining property of digital circuits is that they represent information in terms of 0s and 1s. We typically interpret these as Boolean logic values, with 0 denoting *false* and 1 denoting *true*. We can then write equations involving involving Boolean variables and use these equations to define circuits. The fundamental operations in Boolean logic are AND, OR and NOT. We will generally write these operations using $\cdot$ for AND, $+$ for OR and $'$ for NOT. So for example, the logical expression $A \cdot (B + C')$ is true whenever $A = 1$ and either $B = 1$ or $C = 0$. We will often omit the $\cdot$ when this can be done with no loss of clarity. So for example, we might write $AB'(C + D)$ in place of $A \cdot B' \cdot (C + D)$. We'll

also use operator precedence to reduce the need for parentheses. Specifically, the NOT operation has the highest precedence followed by AND, then OR. It's worth noting in passing that there are several alternative notations that are commonly used for writing Boolean logic equations. If you happen to encounter one of these alternate notations in another setting, don't let it confuse you. The logical operations are the same, it's only the notation that's different.

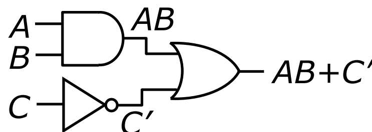
Logic expressions can be used to define digital circuits. So for example, the expression $A + B \cdot C'$ corresponds to the circuit shown below. Notice



that the variables in the logic expression correspond to the circuit's inputs, while the expression itself corresponds to the circuit's output. Also note how each logical operation in the expression corresponds to a gate in the circuit. Given any similar circuit, there is a straightforward procedure to determine the corresponding logic expression. We start by labeling the inputs with distinct symbols, then repeat the following step until the output of the circuit is labeled.

> Select a gate with an unlabeled output, but with all inputs labeled. For an inverter with input label L, the output label is $(L)'$. For an AND gate with input labels L_1 and L_2, the output label is $(L_1) \cdot (L_2)$. For an OR gate with input labels L_1 and L_2, the output label is $(L_1) + (L_2)$. Optionally, remove any redundant parentheses.

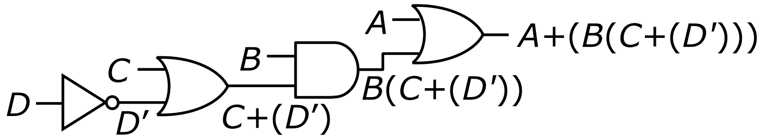
Here's another simple circuit that has been labeled using this procedure.



There is a similar procedure that can be used to derive a circuit from a logic equation. Start by adding parentheses to the expression in a way that is consistent with the original operation ordering, but can be interpreted unambiguously without relying on the precedence rules. Then, create a circuit that contains just an unconnected output wire that is labeled with the fully-parenthesized expression, and repeat the following step.

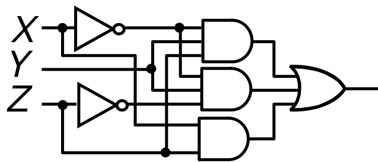
> In the circuit constructed so far, select a labeled signal that is not yet connected to any gate output, where the label contains at least one logical operation. If the label takes the form X', add an inverter to the circuit with its input labeled by X. If the expression takes the form $X_1 \cdot X_2$, add an AND gate with inputs labeled X_1 and X_2. If the expression takes the form $X_1 + X_2$, add an OR gate with inputs labeled X_1 and X_2. Finally, connect the output of the new gate to the selected signal.

So for example, we can construct a circuit for the expression $A + B(C + D')$ by first adding parentheses, giving $A + (B(C + (D')))$, then applying the above procedure. This yields the circuit shown below.



Again, notice how each logical operation in the original expression corresponds to a gate in the resulting circuit. This equivalence between logic expressions and circuits allows us to use the rules of Boolean algebra to create simpler circuits. Let's look at how this works with an example. The circuit shown below corresponds to the logical expression $X'YZ + X'YZ' + XZ$.

Before continuing let's pause for a moment to note that this circuit diagram contains several gates with more than two inputs. For now, we will view gates like this as just a convenient short-hand for an equivalent set of 2-input gates. In general, an AND or OR gate with k inputs can always be replaced by a circuit consisting of $k - 1$ 2-input gates. Two input gates are

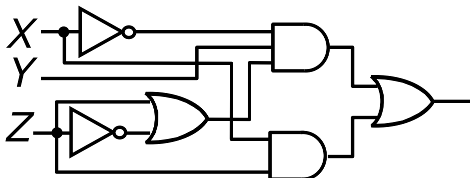


often referred to as *simple gates*. If we made that substitution here, we would get a circuit with nine simple gates.

We can simplify the original expression by factoring out a common subexpression from the first two terms.

$$X'YZ + X'YZ' + XZ = X'Y(Z + Z') + XZ$$

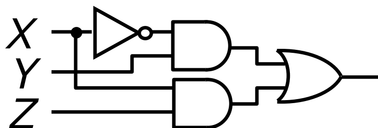
This expression corresponds to the circuit shown below. We can continue to



simplify by noting that $(Z + Z') = 1$, since either Z or Z' must be true.

$$X'Y(Z + Z') + XZ = X'Y + XZ$$

This gives us the following circuit that has just four simple gates, as opposed to the nine we started with.



The rules of Boolean algegra are similar to, but not quite the same as the rules of the familiar algebra used to represent numerical quantities. There is a

collection of basic identities that can be used to simplify Boolean expression. Two of the simplest are these.

$$X + X' = 1 \quad \text{and} \quad X \cdot X' = 0$$

The AND and OR operations are both associative.

$$X + (Y + Z) = (X + Y) + Z \quad \text{and} \quad X(YZ) = (XY)Z$$

Boolean algebra has two distributive laws,

$$X(Y + Z) = XY + XZ \quad \text{and} \quad X + (YZ) = (X + Y)(X + Z)$$

This second one seems a little strange, but there is an easy way to verify that it is true using *truth tables*. For example, here is a truth table for the lefthand side of the second distributive law.

X	Y	Z	YZ	$X + (YZ)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Note that the three variables are shown on the left side of the truth table, and we list all possible combinations of values. The first column on the right side shows the values for the expression YZ, where the entry in each row is determined by the variable values in that row. The last column shows the values of the overall expression. Next, let's construct a truth table for the righthand side of the second distributive law.

X	Y	Z	$X+Y$	$X+Z$	$(X+Y)(X+Z)$
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

Observe that the last column in this truth table has exactly the same values as the last column of the first truth table, so we can conclude that even though it looks a little strange, that second distributive law is in fact true. Note that we can always use truth tables to verify any Boolean equation. This is possible because the number of distinct combinations of input values is always finite. In general, for an equation with k input variables, the truth table will have 2^k rows so the construction of the truth table may not be practical for larger values of k, but for expressions with just a few variables, verification by truth table is a reasonable way to check the correctness of an equation we might not be sure of.

These next two identities are known as DeMorgan's laws.

$$(X+Y)' = X'Y' \quad \text{and} \quad (XY)' = X' + Y'$$

Expressions with lots of NOT operations can be hard to understand. Using DeMorgan's laws, we can rewrite the expressions so that all of the complement operators apply only to input variables. Here are two more handy identities.

$$X + XY = X \quad \text{and} \quad X + X'Y = X + Y$$

We can prove the first one using the first distributive law.

$$X + XY = X \cdot 1 + XY = X(1+Y) = X$$

We can prove the second using the second distributive law.

$$X + X'Y = (X+X')(X+Y) = 1(X+Y) = X+Y$$

Of course all of these identities can be applied to sub-expressions. So for example, the first of DeMorgan's laws yields this.

$$((A+B)+C'D)' = (A+B)'(C'D)'$$

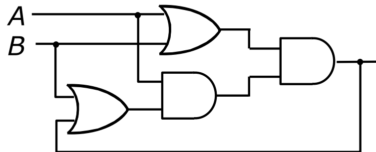
With two more applications of DeMorgan's laws we get the expression on the right below.

$$(A+B)'(C'D)' = A'B'(C+D')$$

Here's an example of applying the identities to simplify a larger expression.

$$\begin{aligned} A(BD'+C'D')+(B+C'+D)' &= A(B+C')D'+B'CD' \\ &= (A(B+C')+B'C)D' \\ &= (A(B'C)'+B'C)D' \\ &= (A+B'C)D' \end{aligned}$$

The circuits that are defined using logic equations belong to a sub-class of digital circuits called *combinational circuits*. A combinatational circuit is one in which the circuit outputs can be written as functions of the circuit inputs. Notice that this property generally does not hold for circuits that contain flip flops (like the calculator). In circuits with flip flops, the outputs usually depend not only on the current inputs, but also on the values stored in the flip flops. Can we tell if a circuit is combinational or not just by looking at it? Generally the answer is yes. A circuit that contains flip flops or other explicit storage elements is almost always not combinational. However, one can also construct circuits that are not combinational without using explicit storage elements. An example is shown below.



If we start with $A = B = 0$ and then make $A = 1$, the output of the circuit will be 0. On the other hand, if we start with $A = B = 1$, then make

$B = 0$, the output will be 1. (Take a minute to convince yourself that these statements are true.) So, in both cases, we end up with $A = 1$ and $B = 0$, but the output has different values. This means that the output is not a simple function of the circuit inputs, hence it is not combinational. The *feedback path* that passes through the leftmost OR gate and the two AND gates is responsible for this behavior. In general, a feedback path in a circuit is a path from the output of some gate G that passes through a sequence of other gates to return to some input of G. Circuits that contain feedback paths are generally not combinational (although there are some exceptions). Indeed, storage elements like flip flops use feedback in order to implement data storage.

Now most circuits we encounter in practice do contain storage elements, hence they are not combinational. On the other hand, if you take out the storage elements, what's left behind is generally a collection of combinational circuits. When we design circuits we're mostly specifying these combinational circuits that link the storage elements to each other and to the inputs and outputs of the overall circuit. For that reason, we'll devote a lot of our time to understanding how to design combinational circuits.

Chapter 3

Designing Combinational Circuits With VHDL

Hardware description languages like VHDL, allow one to specify digital circuits at a higher level of abstraction than the level of gates and individual signals. This allows designers to be more productive and to design more complex systems than would be practical using lower-level design methods. There is a natural analogy between hardware description languages and ordinary programing languages. In both cases, high level specifications are translated into lower level representations. In the case of a programming language, that lower level representation is a sequence of machine instructions. In the case of hardware description languages, it is a circuit consisting of gates and flip flops, connected by wires. In the case of hardware description languages, this translation process is called circuit synthesis, and the progams that do the translation are called synthesizers.

While hardware description languages share some features with programming languages, there are some important differences as well. In a programming language, the language statements translate to machine instructions that are executed one after another, and which modify values stored in a computer's memory. Circuit specifications using hardware description languages define circuits and the behavior of these circuits is fundamentally different from the behavior of sequential programs. In a circuit, there is

no underlying memory that holds the values of program variables. Instead, there are *signals* that correspond to wires in the circuit. Signal values can change in a much more dynamic fashion than variables in a program. In particular, they generally do not change sequentially. Many signals in a circuit may change values at the same time, and this inherent parallelism in circuits can make it more difficult to understand the behavior of a circuit than that of a sequential program. As we proceed with our discussion of VHDL, we'll emphasize how VHDL language statements are translated into circuits. Even though these statements look similar to statements in ordinary programming languages, it's important not to let our experience with sequential programming cloud our understanding of what the VHDL statements mean. Whenever you're reading or writing VHDL code, keep the correspondence between the VHDL and the cicuit clearly in mind. This will help you develop a better understanding of the language and will enable you to more quickly master the art of circuit design using VHDL.

3.1 The entity and architecture

Let's start by discussing the high level structure of a VHDL circuit specification. A VHDL spec consists of a collection of circuit modules, each of which is defined by a block of code called the *entity declaration* and another called the *architecture*. The entity declaration specifies the inputs and outputs to a circuit module, specifically the signal names and their types. Single bit signals typically have type `std_logic` while multi-bit signals have type `std_logic_vector`. An example of an entity declaration is shown below.

```
entity bcdIncrement is
    port (
        a3, a2, a1, a0: in std_logic;
        x3, x2, x1, x0: out std_logic;
        carry: std_logic);
end bcdIncrement;
```

The declaration starts with the `entity` keyword and the name of the module, which in this case is `bcdIncrement`. The declaration ends with the keyword

`end` followed by the name of the module. The bulk of the entity declaration is the *port list* which starts with the keyword `port`, followed by a parenthesized list of port declarations. The declarations for inputs (or outputs) that have identical types may be combined, as shown here. Alternatively, they can be shown separately. Note that in the port list, there is no final semicolon before the closing parenthesis. In this context, the semicolon is used to separate the items in the port list.

Before turning to the architecture for this circuit module, let's talk about what the circuit is supposed to do. This circuit is a *BCD incrementer*. So what does that mean? Well, BCD stands for *binay coded decimal*, which is a way to represent decimal numbers using binary signals. Essentially, each decimal digit is represented by a group of four bits that has the same value (in binary) as that digit. So for example, the digit 5 would be represented by the binary value 0101 and the decimal value 539 would be represented by the 12 bits 0101 0011 1001. BCD is an alternative to representing numbers directly in binary, and there are some applications where it is more convenient than using the standard binary representation. Now a BCD incrementer adds 1 to a BCD value. In this case, the incrementer adds 1 to a single BCD digit and produces the appropriate sum plus a carry signal, if the addition produces a value larger than "1001". The circuit's operation can be specified precisely using the truth-table shown below.

$a_3a_2a_1a_0$	$carry$	$x_3x_2x_1x_0$
0000	0	0001
0001	0	0010
0010	0	0011
0011	0	0100
0100	0	0101
0101	0	0110
0110	0	0111
0111	0	1000
1000	0	1001
1001	1	1010

Notice that the truth table is incomplete. That is, it does not include a row for all 16 possible input values. That's because six of the 16 input combinations are not "legal" in this context, and hence we don't really care what output is produced for those input combinations. We can often take advantage of such *don't care conditions* to obtain simpler circuits than we would if we specified a specific value in advance.

We can use the truth table to derive logic equations for each of the four output signals. Let's start with the easiest one, the *carry* signal. This output is high when the input is "1001", but we can take advantage of the don't care condition to get the simpler equation.

$$carry = a_3 a_0$$

Next, let's turn to the signal x_0. Notice that the values of x_0 are always just the complement of the values of a_0, hence we can write the logic equation

$$x_0 = a_0'$$

for this output. Next, notice that the output x_1 is high whenever the inputs a_1 and a_0 are not equal to each other. Otherwise, x_1 is low. This gives us the equation

$$x_1 = a_1 a_0' + a_1' a_0 = a_1 \oplus a_0$$

The symbol $\oplus$ denotes the *exclusive or* operator. In general, $y \oplus z = 1$ whenever $y = 1$ or $z = 1$, but not both. We can also think of the exclusive-or operator as denoting "not equal", since it is true whenenver $y \neq z$. Turning to the next output signal, note that $x_2 = 1$ when $a_2 = 0$ and $a_1 = a_0 = 1$ or when $a_2 = 1$ and a_1 and a_0 are not both equal to 1. This gives us the equation

$$x_2 = a_2' a_1 a_0 + a_2 (a_1 a_0)' = a_2 \oplus (a_1 a_0)$$

Finally $x_3 = 1$ whenever $a_3 = 1$ or when $a_2 = a_1 = a_0 = 1$. This gives us

$$x_3 = a_3 + a_2 a_1 a_0$$

We note that the expression $a_1 a_0$ appears twice in these equations.

Here is a VHDL architecture based on these equations.

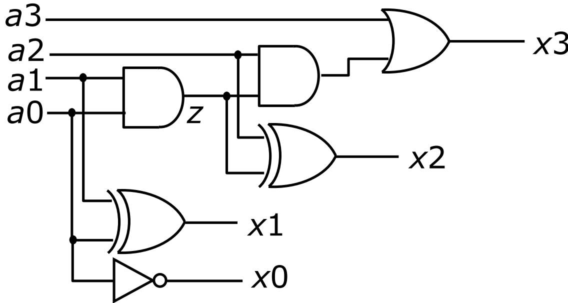
```
architecture a1 of bcdIncrement is
signal z: std_logic;
begin
    carry = a3 and a0;
    z <= a1 and a0;
    x0 <= not a0;
    x1 <= a1 xor a0;
    x2 <= a2 xor z;
    x3 <= a3 or (a2 and z);
end a1;
```

A few comments about the format of the architecture. First, note that it starts with the keyword `architecture` followed by the name of the architecture, the keyword `of` followed by the entity name and the keyword `is`. It may seem redundant to have a separate name for the architecture, but VHDL is defined to allow multiple distinct architectures that share the same interface. While this is a useful feature, we will not use it in this book. Consequently, in all of our examples, the architecture name is just an arbitrary label (such as the `a1` used here), and has no real significance.

The next thing in the architecture is a list of signal declarations (in the example, there's just a single declaration). Later we'll see that this list may also include type declarations, as well as function and procedure definitions. The *body* of the architecture starts with the `begin` keyword. In this case, the body consists of a set of *signal assignments*, where each assignment defines the values of one signal. Note the use of the symbol `<=` to denote signal assignment. This symbol is also used to denote "less-than-or-equal" in other contexts, but here it is being used to mean signal assignment. The right side of the signal assignments uses the keywords `and`, `or`, `xor` and `not` to denote the corresponding logical operations.

It's important to keep in mind that signal assignments are not like assignments in ordinary programming languages. In an ordinary programming language, an assignment means, " evaluate the expression on the right and copy its value into the memory location that corresponds to the variable name on the left." VHDL signal assignments are really just like logic equations specifying the values of the signals, and they effectively define circuits.

In this case, the circuit would look something like this.



The symbol that looks like an OR gate with the extra curve along the left edge is an exclusive-or gate. This arises often enough in real circuits that it has its own symbol.

Because of the nature of VHDL signal assignments, the order in which these assignments are written does not really affect the meaning of code. In the `bcdIncrement` example, we could have written the assignment for `z` *after* all the other assignments, and the specified circuit would be exactly the same. Of course, in an ordinary programming language this would not make sense, since in that case, the assignments to `x3` and `x2` would both contain references to an undefined variable.

Before leaving this example, it's worth noting that while our VHDL architecture for the `bcdIncrement` circuit does what it's supposed to do, the process of coming up with the design was a bit tedious. It turns out, there is a much easier way to specify this circuit, which is shown in the alternate version below.

```
entity bcdIncrement is
    port (
        a: in std_logic_vector(3 downto 0);
        x: out std_logic_vector(3 downto 0);
        carry: out std_logic);
end bcdIncrement;
architecture a1 of bcdIncrement is
```

```
begin
    carry <= a(3) and a(1);
    x <= a + "0001";
end a1;
```

In this case, we've changed the interface to use signal vectors, instead of individual one bit signals. This allows us to specify the `x` output using a single *vector signal assignment* that uses the addition operator. This is much more direct than the original. In general, when using VHDL, it makes sense to take advantage of the higher level of abstraction that the language makes possible. It generally saves time and makes it less likely that we'll make a mistake in our specification.

3.2 Signal Assignments

The signal assignment is the fundamental building block of a VHDL specification. The simplest signal assignments have a single bit signal on the left and a logic expression on the right, constructed using the standard logic operations. We have also seen examples of signal assignments using logic vectors. It's worth taking a few moments to look at these more closely. Suppose we have three signals, declared as follows.

```
signal A: std_logic_vector(3 downto 0);
signal B: std_logic_vector(4 downto 1);
signal C: std_logic_vector(0 to 3);
```

These are all four bit signals, but note that the index ranges are all different. The index ranges affect the way signal assignments are interpreted. Consider the following signal assignments.

```
A <= "1100"; B <= A; C <= A;
```

The first assignment is equivalent ot

```
A(3) <= ‘1’; A(2) <= ‘1’; A(1) <= ‘0’; A(0) <= ‘0’;
```

That is the order in which the bits on the right are assigned to the different bits of A is determined by the order in which the bits appear in the index range of A. We could make this explicit by writing

```
A(3 downto 0) <= "1100";
```

We could also make the other assignments more explicit by writing

```
B(4 downto 1) <= A(3 downto 0);
C(0 to 3) <= A(3 downto 0);
```

Note how the value of A(3) is assigned to B(4) and so forth. Similarly, note that A(3) is assigned to C(0) since the index range for C is the reverse of the index range for A. In general, for any multi-bit signal, we can think of the bits as having a specific left-to-right ordering and this determines the order which bits are assigned. Note that explicit index ranges can be used to specify a subset of bits. For example, we could write

```
B(4 downto 3) <= A(1 downto 0);
B(2 downto 1) <= A(3 downto 2);
```

An alternate way to write this uses the *signal concatenation operator* denoted by '&'.

```
B <= A(1 downto 0) & A(3 downto 2);
```

In this assignment, the right-hand side is a four bit signal obtained by concatentating a pair of two bit signals.

There is still another way to specify logic vectors in expressions. Consider this assignment.

```
A <= (3 => '0', 2 => '0', others => '1');
```

This specifies a bit vector in which the bits with indices 3 and 2 are both set to 0, while all other bits in the vector are set to 1. A common idiom in VHDL is an assignment of this form.

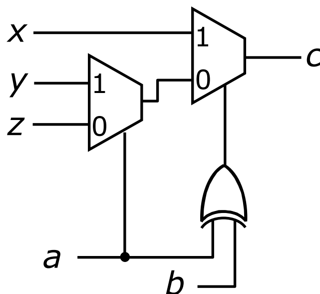
```
A <= (others => '0');
```

This specifies that all bits of `A` should be set to 0, but it has the advantage that it does not depend on the length of `A`. Note that the index range for the signal on the right side of this assignment is inferred from the index range of the signal on the left.

In addition to ordinary signal assignments, VHDL includes a *conditional signal assignment* that often allows us to express the value of a signal in a simpler way than we could otherwise. Here's an example.

```
c <= x when a /= b else
     y when a = '1' else
     z;
```

Here, the *when-clauses* specify a condition under which the specified value is assigned to the signal on the left. The symbol '/=' is the VHDL symbol for "not-equal." This signal assignment can be implemented by the circuit shown below. It's worth noting that a conditional signal assignment can always



be replaced by an ordinary signal assignment. In this case, an equivalent assignment is

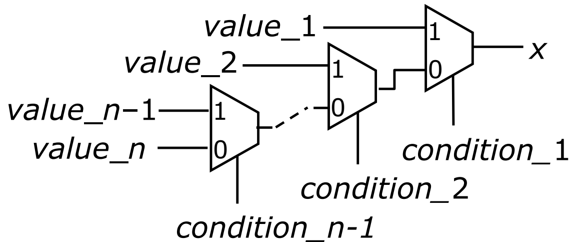
```
c <= ((a /= b) and x) or
     (a and not (a /= b) and y) or
     (not a and not (a /= b) and z);
```

The first version is certainly easier to understand, but it's worth keeping in mind that any circuit that we can define using conditional signal assignments can also be defined using ordinary assignments.

The general form of the conditional signal assignment appears below.

```
x <= value_1 when condition_1 else
     value_2 when condition_2 else
     ...
     value_n;
```

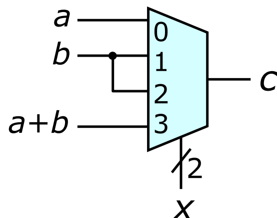
The signal on the left and the values on the right may be signal vectors, but they must all have the same number of bits. The conditions may include expressions defined on signal vectors, but they must evaluate to a single bit value. The circuit that implements the general form of the conditional signal assignment is shown below.



There is still another type of signal assignment in VHDL, called the *selected signal assignment*. Here's an example.

```
with x select
    c <= a when b"00",
         b when b"01" | b"10",
         a+b when others;
```

Here, the signal `x` is used to select one of several alternatives. If `x=b"00"` the value of `a` is assigned to `c`. If `x=b"01"` or `x=b"10"` the value of `b` is assigned to `c`. Otherwise, the sum `a+b` is assigned to `c`. A circuit that implements this selected signal assignment appears below.



The trapezoidal symbol is a 4-input multiplexor. This is similar to the 2-input multiplexor we've already seen. It has a two bit control input and the numerical value of the control input selects one of the data inputs and connects it to the output. In general, a multiplexor with k control inputs can be used to select from among 2^k inputs. Note that the selected signal assignment is essentially a special case of the conditional signal assignment, but the circuit using a single multiplexor is usually faster than the circuit typically used for the conditional signal assignment.

Before leaving the topic of signal assignments we note that the conditional signal assignment and selected signal assignment may only be used outside of a process. Within a process there are other language features that provide the same functionality.

3.3 Processes and if-then-else

So far in this section, we have limited ourselves to VHDL architectures that contain only assignments. The language includes other language constructs such as the *if-then-else* statement. However, these statements can only be used within a *process*. Here is an example of a VHDL module that implements a simple combinational circuit using a process.

```
entity foo is port(
    p, q: in std_logic;
    x, y: out std_logic_vector(3 downto 0));
end foo;
architecture bar of foo is begin
    process (p, q) begin
```

```
            if p /= q then
                x <= "0010"; y <= "1100";
            elsif p = '1' then
                x <= "1101"; y <= p & q & "01";
            else
                x <= "0100"; y <= "10" & q & p;
            end if;
        end process;
    end bar;
```

A process generally contains several signal assignments, and we can view the signals that appear on the left of these assignments as *outputs* of the process, since the process determines their values. In this example, signals `x` and `y` are both process outputs. Now an architecture may include more than one process, but it usually doesn't make sense for a given signal to be controlled by more than one process, since the circuits defined by the different processes could end up producing conflicting values for the same signal. For that reason, we'll generally require that each signal in an architecture be controlled by at most one process. Now architectures may also include signal assignments that fall outside of any process. Each such assignment is considered to be just a simple form of process. So if a signal is controlled by one such assignment, it may not be controlled by another, or by any explicitly-defined process.

To understand how a process defines a circuit, it helps to know that each of the outputs of a process can be viewed independently of the other outputs. So for example, we could re-write the body of the process in the example as follows.

```
if p /= q then x <= "0010";
elsif p = '1' then x <= "1101";
else x <= "0100";
end if;
if p /= q then y <= "1100";
elsif p = '1' then y <= p & q & "01";
else y <= "10" & q & p;
end if;
```

The two if-then-else statements are completely independent of one another and could appear in either order without affecting the meaning of the circuit as a whole. This is an example of the *separation principle*, which states that a process can always be re-written to completely separate the code that defines different output signals. Now an if-then-else that involves only assignments to one signal can be easily re-written as a conditional signal assignment, and so we can view such a statement as defining a circuit of the form that was described earlier for the conditional signal assignment. We can also view a general if-then-else that contains assignments to several different signals as just a collection of conditional signal assignments, one for each signal. Now, a circuit synthesizer will not typically implement such an if-then-else using completely separate sub-circuits. It can typically produce a more efficient circuit by taking advantage of the fact that the same conditions are used in the definition of different signals. Still, the behavior of the circuit must be consistent with the behavior that would be obtained by synthesizing separate circuits for each signal, and that can be helpful when trying to understand the circuit specified by a VHDL process.

The separation principle can help us understand some otherwise confusing situations. Consider the process shown below.

```
process (x) begin
    x <= '0'; y <= x; x <= '1';
end process
```

What value is assigned to signal `y`? Well, if we apply our knowledge of ordinary programming languages, we might say that `y=0`, but that interpretation

is based on a sequential execution model in which assignment statements are executed one after another, with values stored in memory locations. Of course circuits don't work like that. The first assignment to x specifies that it is "wired" to a constant 0 value, while the second specifies that it is wired to a constant 1. Since these are contradictory, VHDL ignores the first and uses the second as the specification for x. Since y is defined in terms of x, it will also have the value 1. Now, if we apply the separation principle to this and just view the assignments to x by themselves, this seems reasonably intuitive. It's only when we insert the assignment to y in between the other two that the situation gets murky.

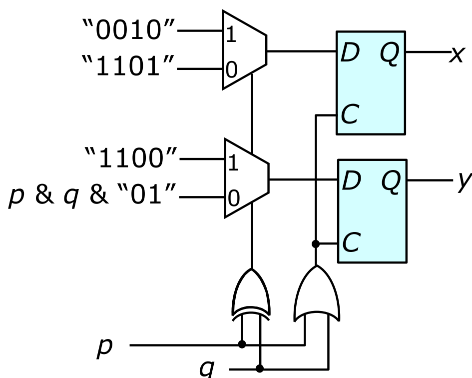
Now you might object that no one would really write a process like the one above, but similar situations can arise in less obvious ways in larger processes. Here is an example that illustrates this.

```
process (p, q) begin
    ...
    if q <= "0100" then p <= '0'; end if;
    ...
    x <= p;
    ...
    if q >= "0100" then p <= '1'; end if;
    ...
end process
```

Here the ellipses are meant to refer to additional, unspecified code. In this example, the value defined for x is usually clear, but what about when q="0100". Once again, we might expect x to be 0, but as in the earlier example, it would actually equal 1. Because the two if-statements have overlapping conditions, the VHDL synthesizer will use the value specified by the second if-statement when q="0100".

Now, let's go back to the example from the start of this section. The process in this example defines a *combinational circuit*. What makes the circuit combinational is that for each of the two output signals, we have specified a value for that signal under every possible input condition. The assignments within the scope of the final else ensure that this is true. What

would happen if we did not include the final else? This would leave the values of x and y undefined whenever p=q=0. The language is defined to interpret this as meaning that the values of x and y should not change in that case, but this means that the circuit must be able to retain their old values whenever p=q=0. Here is a circuit that might be synthesized if the final else were omitted.



The rectangular symbols in this figure denote D-latches. A latch is a storage device that is similar to a flip flop. Indeed later in the course, we will see that latches can be used to implement flip flops. When the C input of a D-latch is high, its D input is connected directly to its output Q. In this case, we say the latch is *transparent* since values on the input appear directly at the output. When the C input drops low, the latch retains its last value. That is, the output no longer follows the input, it just continues to hold the last value it had. In this circuit, the latches are transparent whenever either p or q is high, but when they both drop low, the latch retains its old value implying that signals x and y do not change at that point.

Now there are times when we may want to use latches to store values in a circuit. However, it's very common for designers (especially beginners) to accidentally specify latches in circuits that are supposed to be combinational. This can happen whenever the VHDL code fails to specify values for a signal under all possible input conditions. Such errors can be difficult to track down, because they cause circuits to behave in ways that are very different from

what a designer might expect. Fortunately, there is an easy way to avoid such inferred latches, and that is to always start a combinational process by assigning *default values* to all output signals of the process. Here is an alternate version of our example process that uses default values, while omitting the final else.

```
entity foo is port(
    p, q: in std_logic;
    x, y: out std_logic_vector(3 downto 0));
end foo;
architecture bar of foo is begin
    process (p, q) begin
        x <= "0100"; y <= "10" & q & p;
        if p /= q then
            x <= "0010"; y <= "1100";
        elsif p = '1' then
            x <= "1101"; y <= p & q & "01";
        end if;
    end process;
end bar;
```

Like the original process, this one ensures that the output signals are always defined, since any input condition left "uncovered" by the if-then-else statement is automatically covered by the default assignments. Hence, the resulting circuit is guaranteed to be combinational. Now in this case, there is no compelling reason to prefer the second version over the first. However we'll find that in larger, more complicated processes, it can be easy to accidentally leave some signal only partly specified. The use of default assignments gives us an easy way to avoid such situations, so it's a good idea to make it a habit to include them in your circuit specs.

Before we conclude our discussion of processes, note that the first line of a process includes a list of signals, known as the *sensitivity list*. In the example given earlier, the sensitivity list includes the signals `p` and `q`. The sensitivity list is a somewhat confusing feature of the language. It is essentially used to notify a circuit simulator that it should pay attention to changes in the

values of the listed signals. More precisely, the simulator should re-evaluate the outputs of the process anytime one of the listed signals changes. When a process is being used to define a combinational circuit, any of the input signals to the process should be included in the sensitivity list, since a change in any input signal can affect the output signals. (A signal is considered an *input* to the process if it appears within any expression in the process body. In the example, `p` and `q` are both inputs to the process.)

The sensitivity list does not affect the circuit defined by a VHDL spec, just which signals a circuit simulator responds to. This is unfortunate, since accidental omission of a signal from the sensitivity list can lead to simulated results that do not correspond to the behavior of the circuit defined by the VHDL. Some synthesizers detect such omitted signals and issue warnings about them, but it is ultimately up to the designer to make sure that all required signals appear in the sensitvity list. Failure to do so can lead to frustrating debugging sessions, as one struggles to understand why a specified circuit does not have the expected behavior when it is simulated.

You might wonder why processes are required to include a sensitivity list in the first place. To understand this, you need to know that VHDL was originally designed as a language to *model circuit behavior* rather than to specify actual hardware. Consequently, it's possible to write VHDL code that can be simulated, but cannot be synthesized. In the case of processes containing *non-synthesizable code*, it can be difficult for a simulator to determine which signals have the potential to affect the behavior of a process, so it is left to the designer to specify these signals explicitly.

Chapter 4

Computer-Aided Design

In this chapter, we'll discuss how modern computer-aided design tools can be used in the design of digital circuits. We'll use a particular set of CAD tools to make the discussion concrete, but other tool sets provide similar features and capabilities. The specific CAD tools used here are the ISE design tools provided by Xilinx. Xilinx is a company that makes *Field Programmable Gate Arrays* and they also provide CAD tools to make it easier for their customers to use their products. While Xilinx charges a fee for their more advanced tools, the core tool set is available as a free download (the ISE webpack).

4.1 Overview of CAD Design Flow

The process of designing a circuit using modern CAD tools can be a bit overwhelming at first. A typical set of CAD tools includes lots of different elements which generate a bewildering collection of auxiliary files that are used to store information about the design. In this course, we will keep things relatively simple and focus on just the most essential elements of the process. The diagram in Figure 4.1 provides an overview of a simplified CAD design flow.

At the top of the diagram are the VHDL source files that we will use to specify the digital system we are designing. Usually a design is broken up

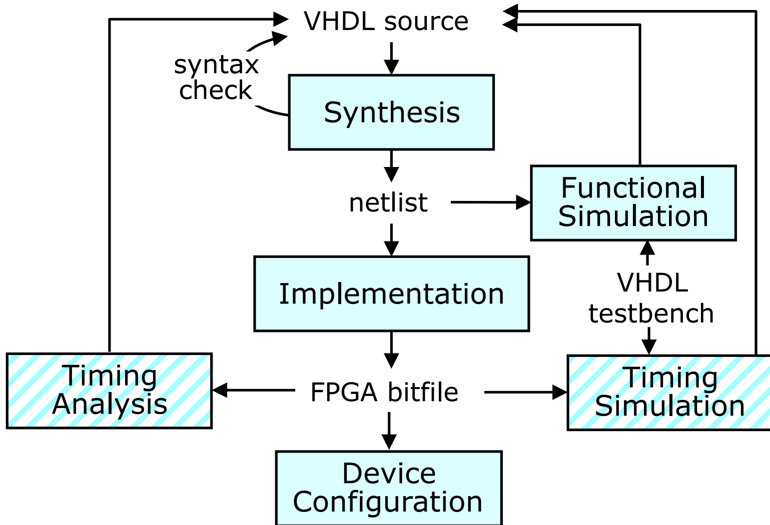


Figure 4.1: Basic CAD Design Flow

into multiple files, typically with a single entity-architecture pair in each file.

The synthesis tool translates VHDL source files into an intermediate form, often referred to as a *netlist*. In the process, it checks that the source files are syntactically correct and generates error messages whenever it finds errors in the source that must be corrected.

The netlist can be used as input to a functional simulation tool. This allows us to verify the logical behavior of our circuit without concerning ourselves with timing or performance issues. Before we can simulate our circuit, we must also provide some test inputs that will put the circuit through its paces and allow us to demonstrate that it operates correctly. We will use a VHDL *testbench* for this purpose. A testbench is just a VHDL source file that, rather than specifying a circuit, defines a sequence of signal values that can be used to test a circuit.

Much of the time spent on a design is devoted to functional simulation. A new circuit design rarely works the first time we attempt to simulate it. By studying the simulation output, we'll find places where the circuit does

not behave as expected. The simulator allows us to observe internal signals within our circuit, and by paying close attention to the relevant signals, we can usually work out exactly where in the source code, a problem originates. This allows us to then determine how the source code needs to change in order to make the circuit behave in the way we intended.

Once a circuit simulates correctly, we can take the next step in the design process, which involves testing the circuit on an actual prototype board. The *implementation* tool converts the netlist into a lower level representation that is appropriate to the specific programmable logic device we are using for our prototype. The resulting *bitfile* can then be used to configure the actual device. If we've done a thorough job of simulating the circuit, this part of the process can be quick and easy. It's also very satisfying to see the prototype circuit operating correctly on a real circuit board. However, if we have not done a thorough job of simulating the circuit, we're likely to find that the prototype circuit does not behave as expected. When this happens, the right thing to do is go back to the functional simulation to figure out what the problem is. It's rarely possible to debug a protoype circuit by experimenting with the actual circuit, because the only things you can observe are the outputs that are available on the prototype board. Because the simulator allows you to see the internal signals as well, it is a much more powerful tool for identifying and fixing mistakes. Don't waste time attempting to debug the physical circuit. If you discover a problem at that phase of the design process, go back to the functional simulation to determine what you did wrong.

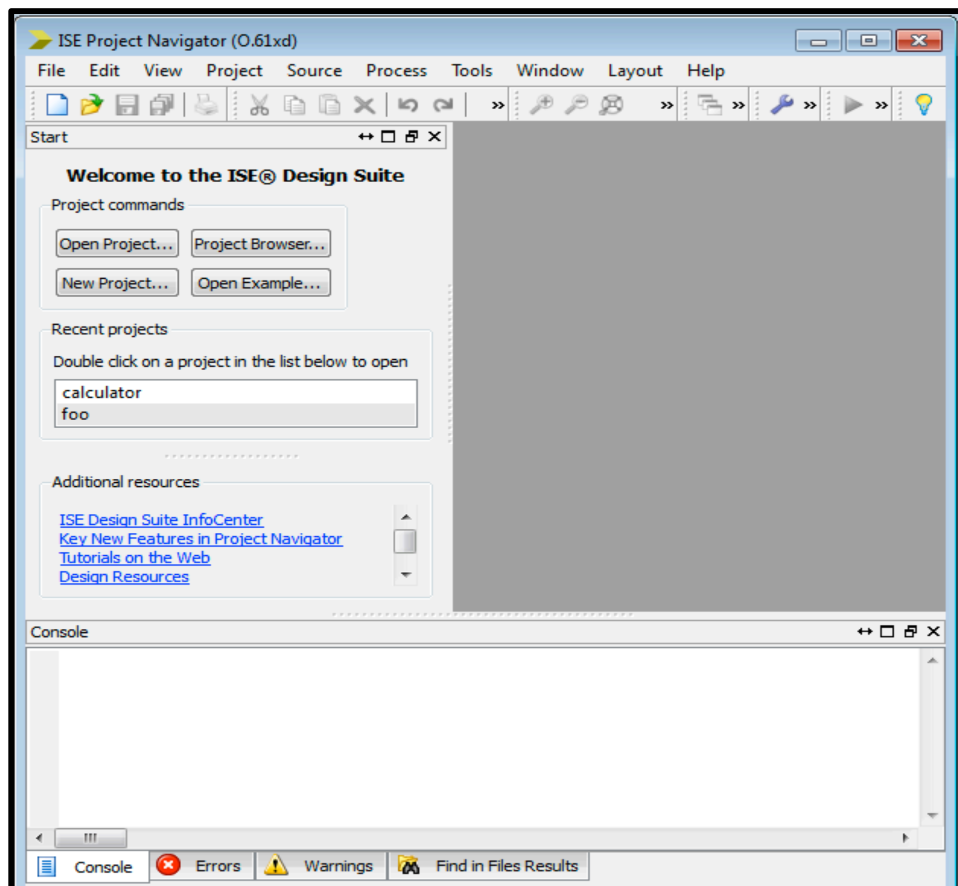
The diagram in Figure 4.1 also has two blocks labeled *timing analysis* and *timing simulation*. In this course, we won't be addressing these aspects of the design flow in any detail, but it is worth mentioning them briefly. When a circuit is mapped onto a specific physical device, the performance of the circuit is affected by many different things, including the locations of various circuit elements within the device, and the lengths of the wires connecting those elements. A timing simulation takes all of these factors into account, in order to produce a more realistic simulation of the circuit's behavior. Such a simulation allows us to observe details that cannot be seen using a functional simulation. We can use this information to identify

possible timing problems that might prevent a circuit from working properly in a real application context.

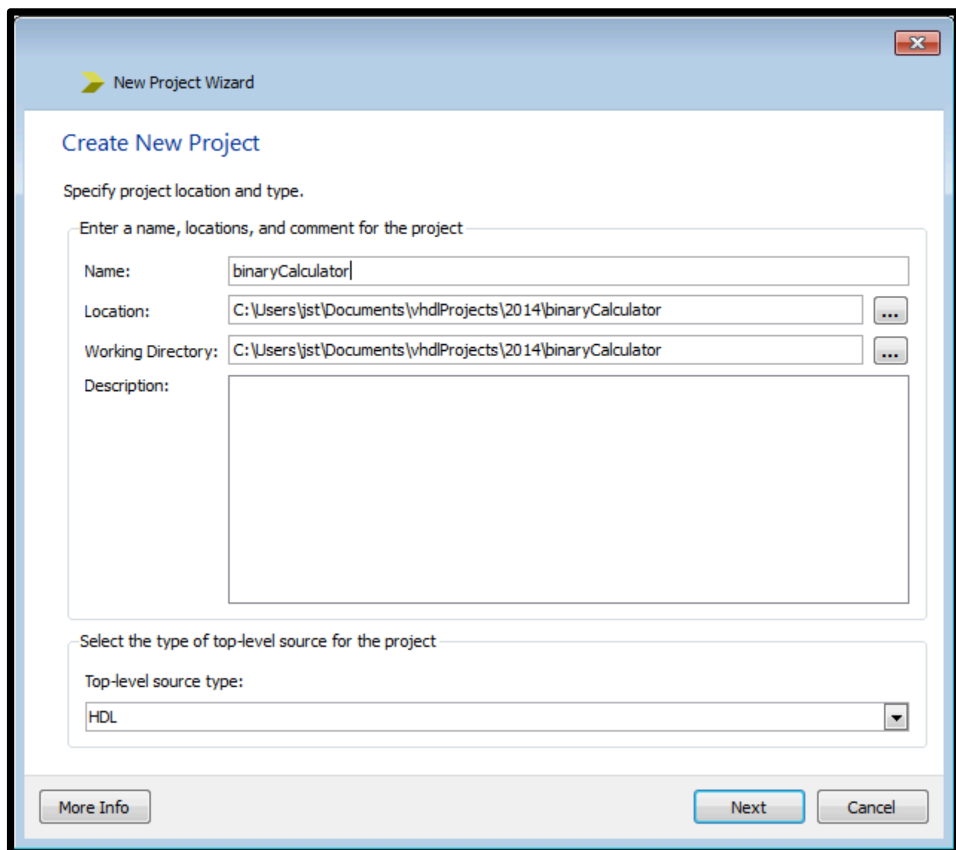
Often, we don't require detailed timing information about our circuit, but we do want to know if the circuit will be able to operate at some target clock frequency. The implementation tool used to produce the bit file actually takes the target clock frequency into account when it makes decisions about how to map circuit elements to specific locations within the device. It will try different options, and analyze their timing characteristics in an effort to find a specific mapping that allows the circuit to operate at the target clock frequency. If the implementation tool is unable to find a mapping that achieves the desired clock frequency, the timing analysis information is reported back to the designer. Changes to the source code may be needed to achieve the desired level of performance.

4.2 Starting a New Project

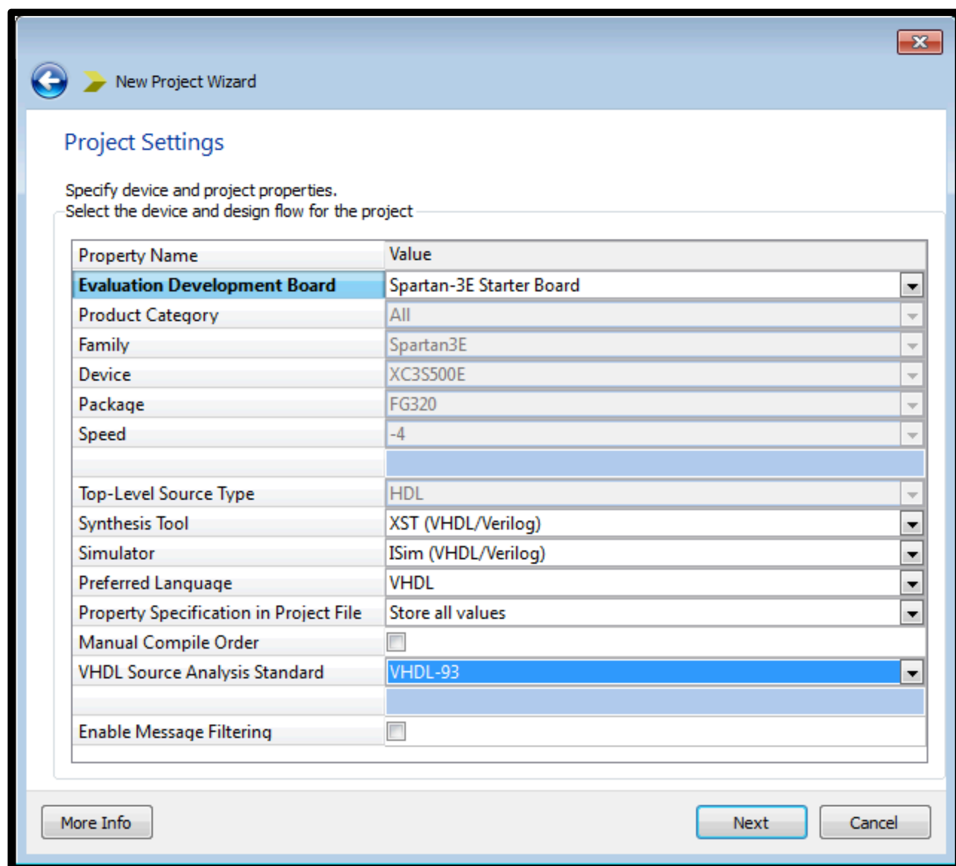
The Xilinx CAD tool set includes an integrated design environment called the Project Navigator. This tool helps organize all the source files needed to define a circuit and the numerous auxiliary files created by various suppporting tools. The main organizational unit in Project Navigator is a *project*, and all the files associated with a project are placed within a common file system folder. To start Project Navigator on a Windows computer, go to the Start Menu, select All Programs, then Xilinx, then ISE Design Tools, then Project Navigator. Your installation may place the tools in a different place in the menu system. If so, you can always search for Project Navigator in the search box of the Start Menu. Once you have started Project Navigator, you should see a window that looks something like this.



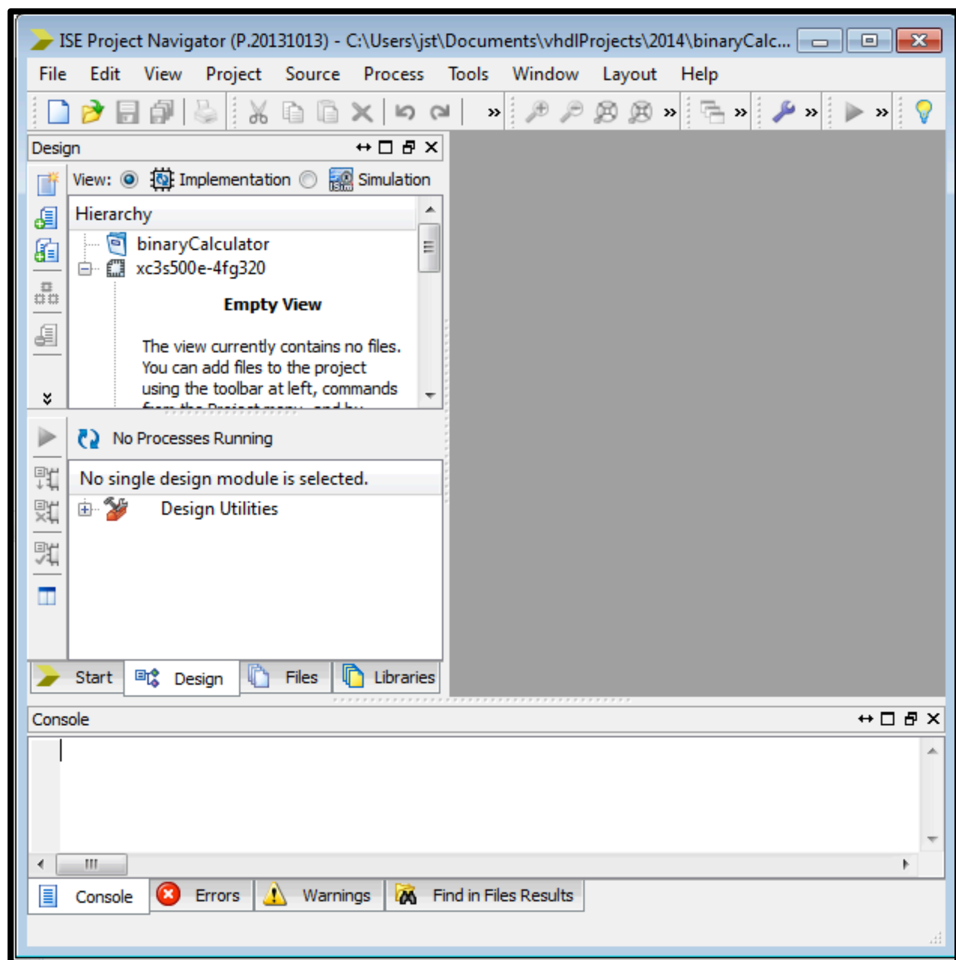
To start a new project, select "New Project" from the File menu. This should bring up the following dialog box.



Type in a name for your project and a location in the file system where you want your project files to be stored. For the purposes of illustration, we'll define a project that implements the binary calculator introduced in Chapter 2. Note that some of the CAD tools cannot handle file system paths that include spaces, so it is best to store your files in a location that is "space-free". One last thing, in the pull-down menu near the bottom of the window, make sure that the selected item is "HDL". After you click the next button, you will get another dialog box which you can use to configure various settings relating to your project.



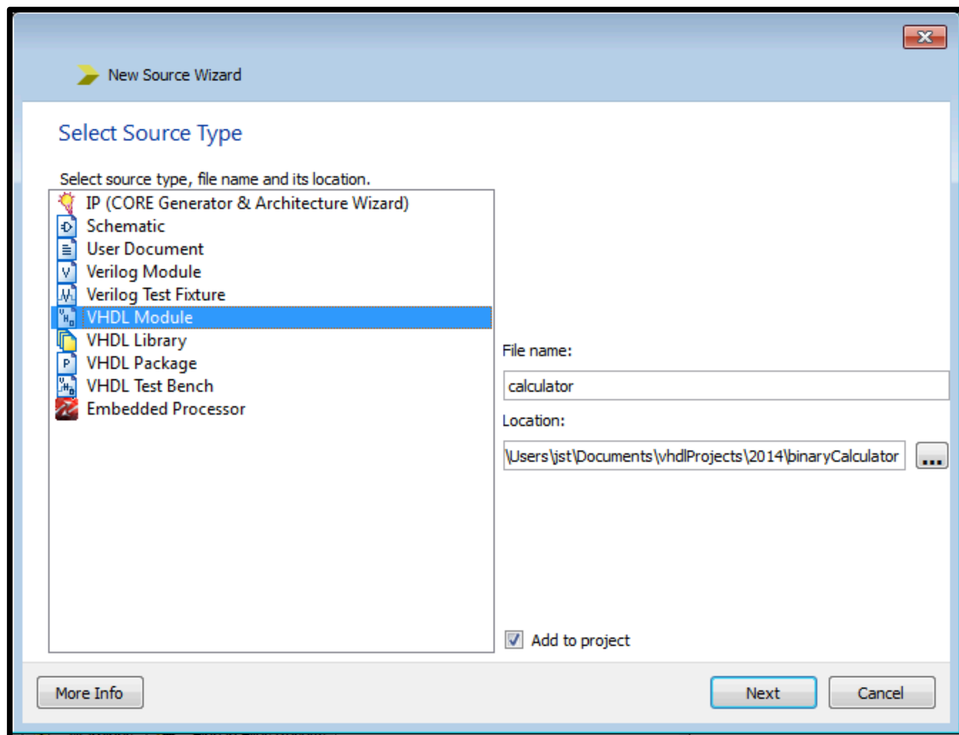
In the setting for "Evaluation Development Board" enter "Spartan 3E Starter Board". This will cause appropriate values to be entered for several other settings related to the FPGA device used on this particular board. Select "XST" as the synthesis tool, ISim for the simulation tool, VHDL as the preferred language and VHDL-93 as the VHDL source analysis standard. When you click "Next", you should get a Project Summary dialog. After clicking the Finish button, your Project Navigator window should look something like this.



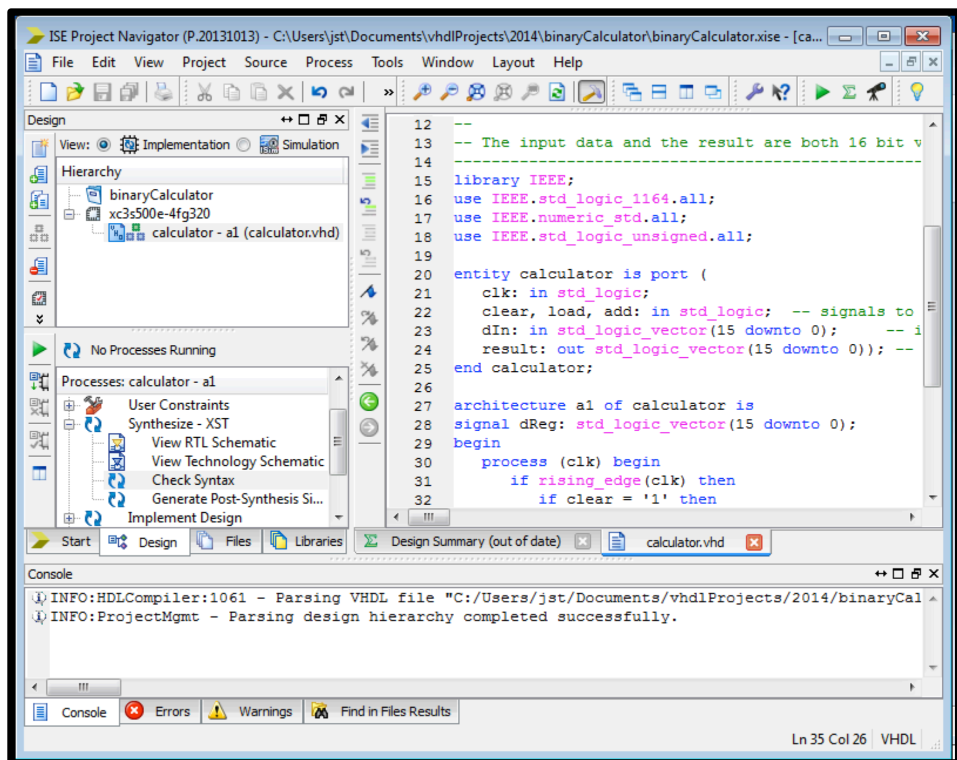
At this point, let's take a moment to observe a few things. First, there is a large gray area at the top right. This area is usually used for entering VHDL source code. We'll get to that shortly. The area to its left is divided into two panels. The top panel typically lists source code files, organized in a hierarchical fashion. The bottom panel provides controls for whatever the current design activity is: typically synthesis or simulation. Across the bottom of the window is a console area used for error messages. Note there

are several tabs at the bottom of this area.

Now, let's go ahead and create our first source file, by selecting "New Source" from the Project menu. This brings up another dialog box. Here,



we select VHDL module and enter a name in the text box at right. Clicking Next takes us to another dialog where we can enter information about the inputs and outputs to a module. It's generally simpler to just skip this and go on to the Summary page. Clicking Finish in the Summary page takes us back to Project Navigator but with the top right area open to the new source code file. After entering the source code for the calculator in the editor window, the Project Navigator looks like this.



It's worth noting that right clicking in the editing window brings up a menu of handy tools, including items to insert a group of lines or add/remove comment marks from a group of lines. Notice that the upper left panel now lists the source file just created. At the top of this panel, there is a place to select one of two differeint "views": Synthesis or Simulation. The choice made here affects what information is displayed in this portion of the window. Double-clicking on the "Check Syntax" item in the bottom panel at left causes the synthesizer to run. Any errors in the VHDL code are reported in the console area below.

4.3 Simulating a Circuit Module

The next step in designing a circuit is to simulate it. In order to do this, we need to create a testbench file. This is done by selecting "New Source" from the Project menu in Project Navigator as we did earlier, but selecting VHDL testbench" rather than "VHDL module". This results in the creation of a new testbench file, which we can edit to test our calculator circuit. Here's what the testbench might look like after editing.

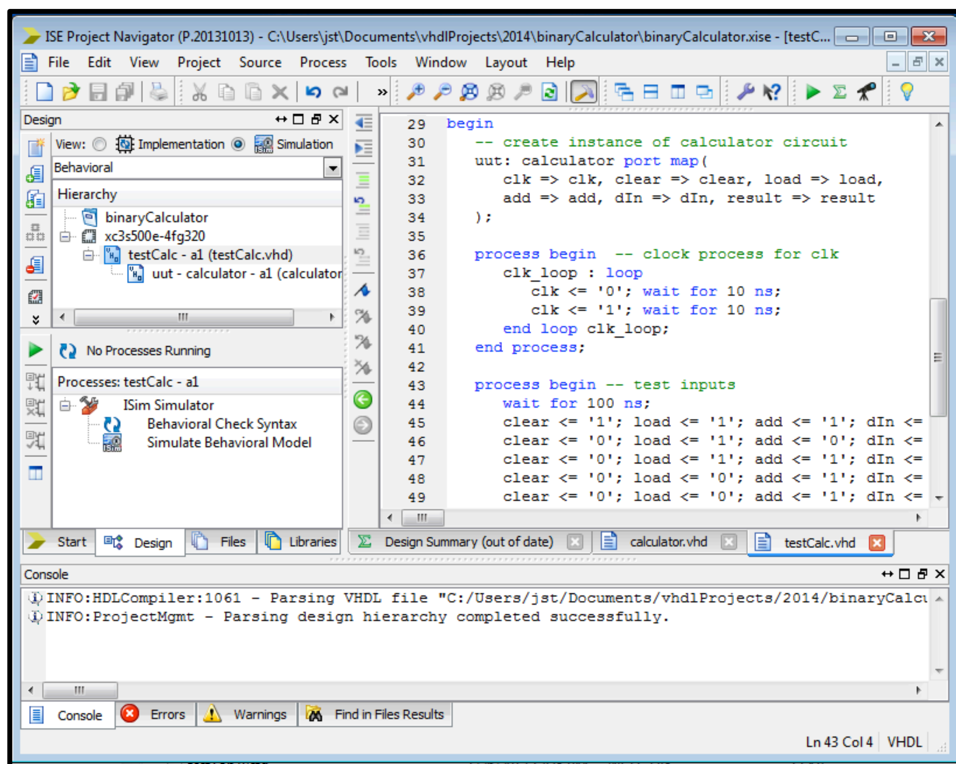
```
entity testCalc is end testCalc;

architecture a1 of testCalc is
component ... end component;
signal clk, clear, load, add: std_logic;
signal dIn, result: std_logic_vector(15 downto 0);
begin
    uut: calculator port map(clk, clear, load, add,
                                  dIn, result);
    process begin  -- clock process
        clk <= '0'; wait for 10 ns;
        clk <= '1'; wait for 10 ns;
    end process;
    process begin
        wait for 100 ns;
        clear <= '1'; load <= '1'; add <= '1';
        dIn <= xffff"; wait for 20 ns;
        clear <= '0'; load <= '1'; add <= '0';
        dIn <= xffff"; wait for 20 ns;
        clear <= '0'; load <= '1'; add <= '1';
        dIn <= xffff; wait for 20 ns;
        clear <= '0'; load <= '0'; add <= '1';
        dIn <= x0001"; wait for 20 ns;
        ...
        clear <= '0'; load <= '0'; add <= '1';
        dIn <= x0500"; wait for 20 ns;
```

```
        wait for 20 ns;

        assert (false) report "Simulation ended normally."
                       severity failure;
    end process;
end;
```

Note that the testbench starts with an entity declaration that has no port clause. This entity has no inputs or outputs, so the port clause is not needed. Note that the testbench includes an instance of the calculator module with the label `uut`. This stands for "unit-under-test" and is the conventional name for the circuit being tested. Next there is a clock process that defines a clock signal with a period of 20 ns. This uses the `wait` statement, which is a non-synthesizable statement that is used in testbenches to control the relative timings of signal transitions. The clock process repeats the two assignments as soln as the simulation is running.

Finally, we have the main process that defines the other input signals to our calculator module. You can think of the main process as operating "in parallel" with the clock process. Note how each group of signal assignments is followed by a wait statement of 20 ns. Since the clock starts out low, and makes low to high transitions at times 10 ns, 30 ns, 50 ns and so forth, the other inputs change when the clock is making its falling transitions. The assert statement at the end of the second process causes the simulation to terminate when it reaches this point. Otherwise, it would simply repeat the test.

Now that we have a testbench, we can actually run the simulator. To do this, selection the "Simulation view" in the Project Navigator. The window should now look something like this.



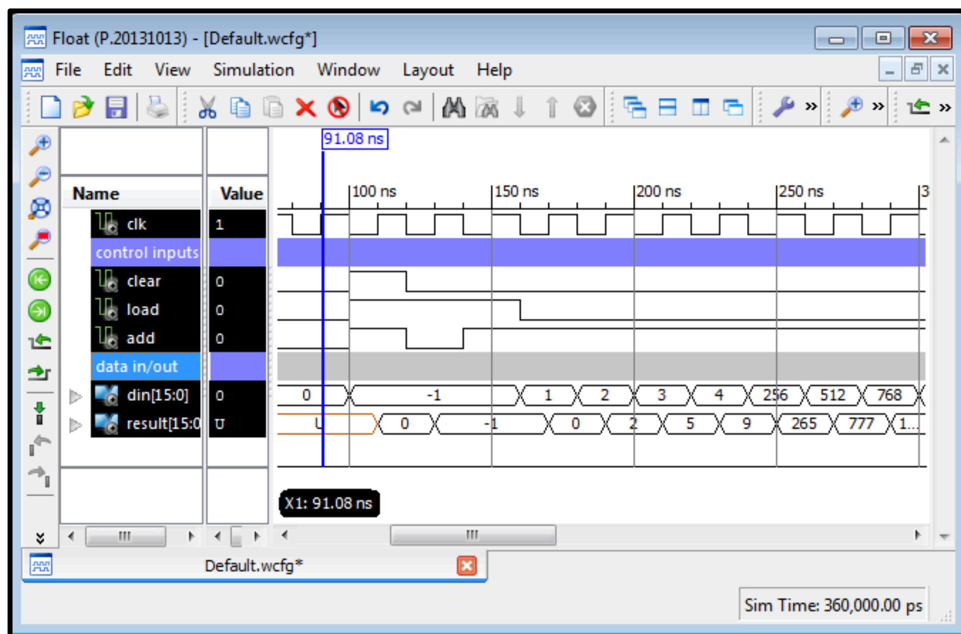
Now, select the testbench file in the top left panel. In the lower panel, you should see an item for the Isim simulator. Expand this if need be and double-click on “Simulate Behavioral Model.” This will bring up the simulation window.

A few things to notice here. First, at the top right, there is a panel containing the "waveform window". This contains the input and output signals for the circuit being simulated. By checking the waveform area we can verify that the outputs produced by the circuit are consistent with what we expect. This window can be separated from the main simulation window, using the "float window" icon that appears just to the left of the icon shaped like a wrench. It is usually more convenient to work with the waveform window in this way, since we can more easily resize it to allow us to closely examine signals of interest.

Before looking in detail at the waveform window, let's examine the other parts of the main simulator window. At the top left there is a panel that includes an item for the testbench, and contained within it, the unit-under-test (that is, the calculator circuit, in this case). When you click on the

"uut" item, the center panel shows a list of all the internal signals associated with the unit-under-test. By right-clicking on one of these signals, we get a context menu that can be used to add that signal to the waveform display. For the calculator, there are no internal signals that we need to add, but in more complex circuits, the ability to add signals to the waveform display is essential for tracking down problems when the circuit is not behaving as we expect it to. At the bottom of the main simulation window, there is a console area, where various informational messages are displayed.

Here is another view of the waveform window, after it has been separated from the simulator window.



We have also made a couple formatting changes to the window. First notice that dividers have been added to the left-hand panel to group related signals together and label them appropriately. Dividers are added by right-clicking in the left-hand panel and selecting the "New Divider" item from the context menu. Another item in the conext menu allows us to change

the radix used to display numeric values. Here, we are showing the input and output signals as signed decimal numbers to make it easier to verify the simulation results.

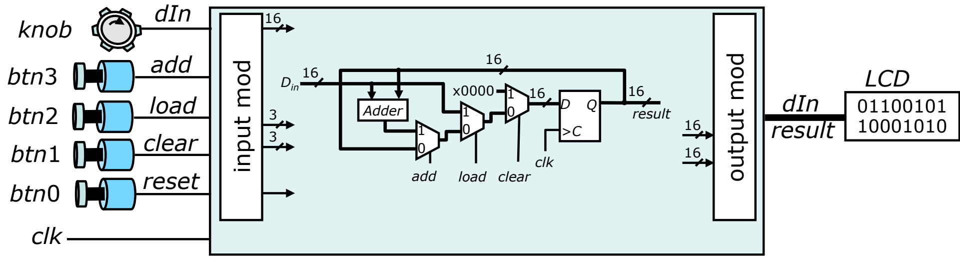
In a larger circuit, we might spend a fair amount of time adding and arranging signals in the waveform window to make it easier to check the results. Since we often have to re-run simulations several times before correcting all the bugs, it makes sense to save the waveform configuration to a file. Selecting "Save As" from the File menu in the waveform window allows us to do just that. Then, on subsequent simulation runs we can start by opening the saved waveform configuration.

A few more things to notice about the waveform window. First, at the top of the icon bar along the left border, there are several zoom controls. Use these to focus in on a specific portion of a simulation that may be of interest. In longer simulations, you'll often find it useful to observe signals that change just occasionally at a large time scale, then zoom in on a small portion of the simulation to observe more detailed behavior. In fact, you can open several different waveform windows at the same time by selecting "New Window" from the Window menu. Each window will appear as a separate tab, allowing you to easily flip back and forth among different views of the simulation.

The simulator provides a variety of other features and you are encouraged to check out the tutorial that can be accessed through the Help menu.

4.4 Preparing to Test on a Prototype Board

Once a circuit has been simulated by itself, the next step is to test it on a prototype board. However, before we can do that, we need to augment the circuit with some additional components so that we can control the inputs to the circuit using the input mechanisms on the prototype board, and observe the results using its LCD display. The basic arrangement is illustrated below.



The prototype board we're using has four buttons and a knob that we can use to control the input signals to the calculator. We'll use the buttons to control the calculator's *clear*, *load* and *add* inputs, while the knob controls the value of a 16 bit register that can be connected to the calculator's data input. The input data and output result can both displayed on the prototype board's LCD display, as binary numbers. To make this work, we need two additional circuit components, an *input module* and an *output module*. We won't discuss the internals of these components, but we will describe how they connect to the calculator.

Let's start with the input module. First however, we have to explain something about buttons. Buttons are mechanical devices with a stationary contact and another that moves when we press down on the button. When these contacts come together, they vibrate causing the resulting electrical signal to switch on and off several times before becoming stable. This phenomenon is called *bouncing*. Now bouncing is a problem because it can cause an internal circuit to behave as though the button has been pressed several times rather than just once. Consequently, the input module includes circuits that *debounce* the buttons. These circuits essentially ignore signal changes that do not remain stable for at least a few milliseconds. So the four external signals, `btn(0)`, `btn(1)`, `btn(2)` and `btn(3)` all have corresponding signals named `dBtn` which stands for debounced button. `DBtn(0)` is actually reserved for a special purpose and given the name `reset`. Pressing the external `btn(0)` will re-initialize all the components of the circuit. The remaining three `dBtn` signals can be connected to other components as appropriate.

Now for some purposes, the `dBtn` signals do not do exactly what we want. In some situations, we want a button press to cause an internal signal to go

high for exactly one clock tick. The `add` input to the calculator provides an example of this. We want a button press to trigger a single addition, but if we connect a `dBtn` signal to the `add` input, each button press would trigger many additions, since the calculator does an addition on every rising clock edge while the `add` signal remains high. For this reason, the input module defines three `pulse signals`, `pulse(1)`, `pulse(2)` and `pulse(3)`. Pressing down on the corresponding external button causes these signals to go high for one clock tick.

Finally, we come to the knob. The input module contains a 16 bit register which is controlled by the knob. Turning the knob clockwise causes the register to increment, turning the knob counter-clockwise causes the register to decrement. This register is connected to an output of the input module, and can be connected to the data input to the calculator.

The output module is simpler to describe. It takes two 16 bit input signals and displays these as binary numbers on the external LCD display. We'll use it to display the data input to the calculator and its result output.

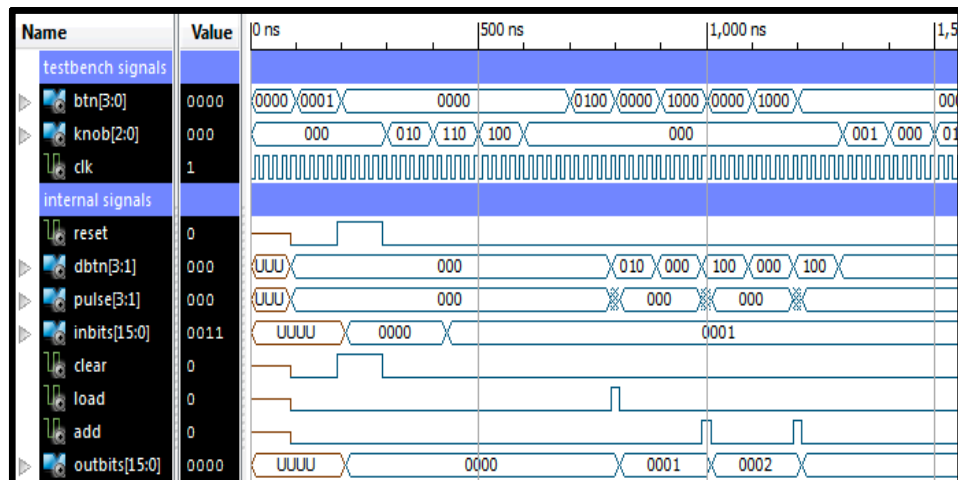
Now, to use the input module and output module with the calculator, we need to define another circuit, which instantiates all three of these circuits and connects them together. The architecture of this top level circuit is shown below.

```
architecture a1 of top is
component calculator ... end component;
component binaryInputModule... end component;
component binaryInputModule... end component;

signal reset, clear, load, add : std_logic;
signal inBits, outBits: std_logic_vector(15 downto 0);
signal dBtn, pulse: std_logic_vector(3 downto 1);

begin
    -- connect the sub-components
    imod: binaryInMod port map(clk,btn,knob,reset,dBtn,
                                           pulse,inBits);
```

```
    calc: calculator port map(clk,clear,load,add,
                              inBits,outBits);
    omod: binaryOutMod port map(clk,reset,inBits,
                                outBits,lcd);

    -- define internal control signals
    clear <= dBtn(1) or reset;
    load <= pulse(2);
    add <= pulse(3);
end a1;
```

4.5 Simulating the Prototype Circuit

Before we can actually test our circuit on the prototype board, it's important to simulate the completed circuit, with all the assembled components. This requires preparing a new testbench that generates the signals for the external buttons and the knob. We'll skip that step here, but we will show part of the results of such a simulation.



The signals for the external buttons and knob appear at the top of the

waveform window. Notice that the debounced buttons, which appear further down the page are delayed versions of the external button signals (only signals `dBtn(3..1)` are shown). Recall from our earlier discussion that the input module debounces the button signals, by suppressing signal transitions that are not stable for at least a few milliseconds. Now our prototype board uses a clock frequency of 50 MHz, so in order to debounce the external signals, we need to delay them by at least 50 thousand clock ticks. This is inconvenient when simulating the circuit, so we have artificially reduced the debouncer delay to just four clock ticks. Finally, note how the `load` and `add` signals appear as one clock tick pulses.

4.6 Testing the Prototype Circuit

Once we are convinced that our prototype circuit works correctly in simulation, we can proceed to the implementation phase. This involves generating a bitfile that can be used to configure the FPGA device on our prototype board. Before we can do this, we need to add a *User Constraint File* to our project. The *ucf* file contains two kinds of information. First, it defines the mapping between the inputs and outputs of our top level circuit with the package pins of the FPGA device we are using. Here is a sample.

```
NET "btn<0>" LOC = V4";
NET "btn<1>" LOC = H14";
...
NET "led<0>" LOC = F12";
NET "led<1>" LOC = E12";
...
NET "clk" LOC = C9";
NET "swt<0>" LOC = L13";
NET "swt<1>" LOC = "L14";
```

The first line associates the top level circuit's `btn(0)` input with the package pin that has the label "V4". The other lines can be interpreted in the same way. The *ucf* file also contains information about the frequency of the clock signal. This takes the form shown below.

```
NET "clk" TNM_NET = "clk";
TIMESPEC "TS_clk" = PERIOD "clk" 20 ns HIGH 50 %;
```

To include a `ucf` file in your project, copy it to the project directory, then select “Add Source” from the Project Navigator’s Project menu. (You can also use Add Source to add VHDL files to your project. This makes it easy to re-use a file from another project.)

The first time we use the implementation tool, there are a couple of steps that are necessary. Start by selecting the “Synthesis View” in Project Navigator and then selecting the top level circuit in the top left panel. Now, right click on the “Synthesis” item in the lower left panel and select the “Process Properties” item. This will bring up the Synthesis Properties window shown below.

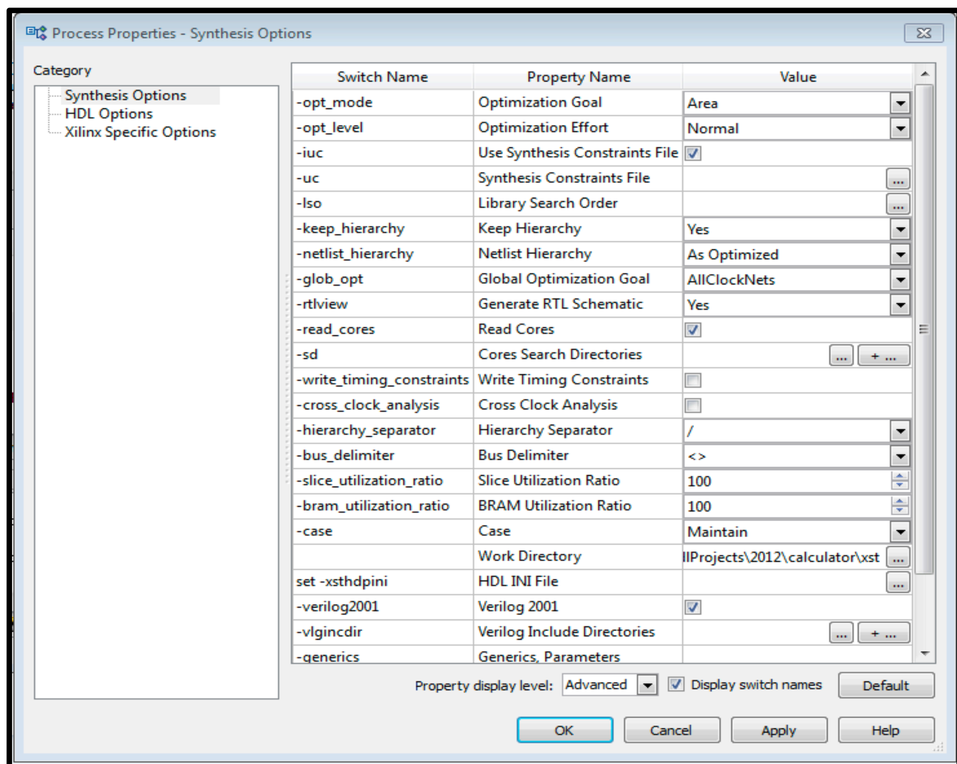
Select the Advanced Display Level and then check that the properties listed match those in the figure. If not, make the necessary adjustments.

Next, right-click on on the “Implementation” item in the lower left panel and select “Process Properties” to bring up the Implementation Properties window.

Select the Advanced Display Level and check the box for the item labelled “Allow Unmatched LOC Constraints”. This tells the implementation tool not to worry if some of the pins on the package are not mapped to inputs and outputs of our top level circuit.

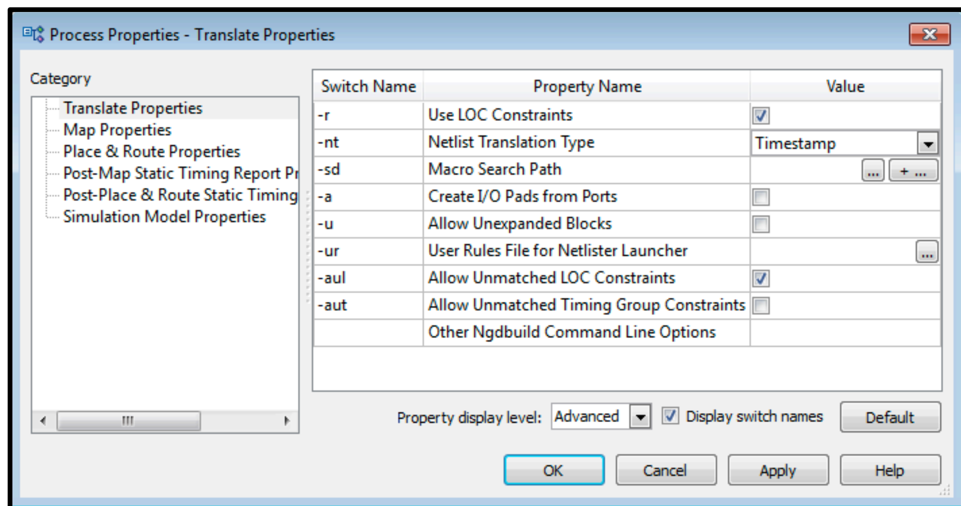
Before we get to the implementation phase, run the synthesis tool by double-clicking on the Synthesis item in the lower left panel. When the synthesis tool runs, it produces a report containing information about the synthesized circuit. It’s worth taking a few minutes to see what’s in this report, as it contains a lot of useful information. You can access it through the “Design Summary” page which appears as a tab in the upper right panel of Project Navigator. One thing you’ll find in the synthesis report is a summary of the basic circuit components used by the synthesizer.

```
Device utilization summary:
---------------------------

Selected Device : 3s500efg320-4
```




```
Number of Slices:                        122  out of   4656
Number of Slice Flip Flops:              117  out of   9312
Number of 4 input LUTs:                  238  out of   9312
   Number used as logic:                 206
   Number used as RAMs:                   32
Number of IOs:                            28
Number of bonded IOBs:                    24  out of    232
Number of GCLKs:                           1  out of     24
```

There are two lines in this section to focus on. First, the number of slice flip flops is reported as 117. The calculator module only accounts for 16 of these.



The remainder are in the input and output modules. The next line reports the number of 4 input Lookup Tables (LUTs). A LUT is a key component in an FPGA device and can implement any four input logic function. By configuring LUTs appropriately and wiring them together, FPGA devices can be used to implement a very wide range of digital systems. Note that the FPGA on the prototype board we're using has 9312 flip flops and 9312 LUTs, so this particular circuit uses a very small fraction of the total resources on the chip.

There is one other part of the synthesis report that is worth taking note of, and that is the section that reports some basic timing estimates for the circuit.

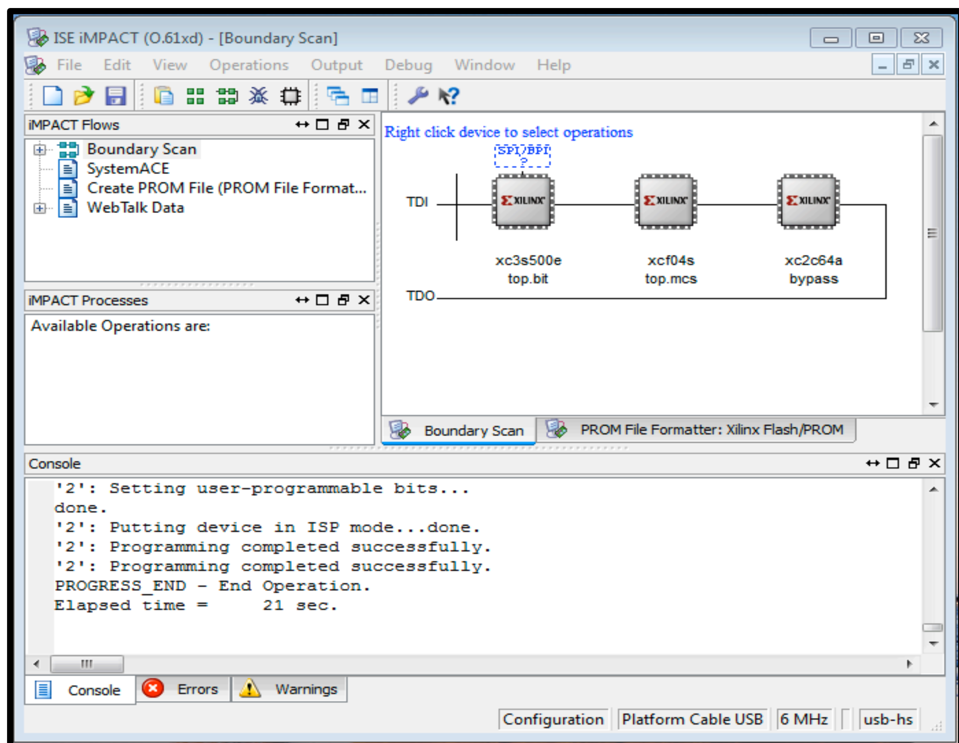
```
Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 9.729ns (Maximum Frequency: 102.785MHz)
   Minimum input arrival time before clock: 5.312ns
   Maximum output required time after clock: 4.450ns
```

```
Maximum combinational path delay: No path found
```

The first line reports a minimum clock period of 9.729 ns for this circuit. This means that the circuit cannot be expected to operate correctly with a smaller clock period. Since the prototype board uses a 50 MHz clock signal (20 ns period), we can be confident that the circuit will work correctly. Note that the timing estimates in the synthesis report should be viewed as rough approximations. Until the implementation phase is complete, we do not really have accurate estimates for all the internal circuit delays, so the synthesis tool has to make educated guesses as to what these delays will be.

Ok, we're finally ready to implement the circuit. To do this, double-click on the "Implement" item in the lower left panel of the Project Navigator window. Next double-click the item that says "Generate Programming File" to create the FPGA bitfile that is used to configure the device. To do the actual testing, you will need a computer with an attached prototype board. If you don't have a board that you can use on the computer where your project files are stored, transfer a copy of the bitfile to a computer that does have a prototype board. Make sure the prototype board is plugged into a USB port, is connected to power and turned on (the power switch is the small slide switch near the power connector).

To load the bitfile on the board, start the Xilinx Impact tool (you may need to do a search in order to find where it's located). When the tool starts up, you will see two dialog boxes; click "No" in the first one and "Cancel" in the second. Then, double-click on the "Boundary Scan" item in the panel at top left. Next, in the large panel at top right, right-click and select "initialize chain". At this point, the window should look like this.



Now right-click on the FPGA icon (it's the leftmost one) and select "Assign New Configuration File" from the context menu. Enter the name of your bit file when prompted. This should cause the file to be downloaded to the FPGA. If all goes well, you should be able to start testing the calculator using the buttons and knobs on the prototype board.

This concludes our introduction to computer-aided design tools. You are encouraged to try things out for yourself. While there are a lot of details to take in, none of it is particularly difficult. Once you've worked through the process a few times, it will start to become familiar to you.

Chapter 5

More VHDL Language Features

In this chapter, we'll describe a number of additional features of the VHDL language, and explain how they can be used to design more complex digital circuits. But first, it's helpful to know a little about how the use of the language has changed since it was first developed. Originally, VHDL was developed as a language for modeling and simulation of circuits, not for circuit synthesis. The idea was that designers would find it useful to model the behavior of hardware systems *before* they developed a detailed design. This modeling process was essentially seen as a precursor to the more detailed design process that would proceed using traditional methods. It was only some years after its initial development that computer-aided design tool vendors recognized that circuits could be synthesized directly from VHDL specifications, if the VHDL code was written so as to avoid *non-synthesizable* aspects of the language.

Because of this history, most books that describe VHDL define the semantics of the language in terms of the required behavior of a circuit simulator. This is unfortunate for those using the language primarily for circuit synthesis, because it leads them to engage in a convoluted thought process about how the sequential behavior of a circuit simulator can be translated into hardware by a circuit synthesizer. In this book, we have made a point to

directly define the language semantics in terms of circuits, because it is more direct and makes it easier to grasp the essential nature of the synthesized circuits.

Next, let's turn to the subject of the *signal* in VHDL. A signal in a circuit specification is an abstraction of a wire, or collection of wires. Now the signal value on a wire is generally determined by the output of some gate or flip flop. So signals change as other signals change, and many such changes can occur at the same time. As we have already noted this is very different from the concept of a variable in a conventional programming language, which is intrinsically connected with values stored in a computer's memory. Coming to grips with this difference between signals and variables is fundamental to understanding the behavior of circuits defined using VHDL. Now, just to add confusion to this issue, VHDL does also include the concept of variables, in addition to signals. The semantics of VHDL variables are different from that of signals, although they are also not really the same as variables in conventional programming languages. Because it can be difficult for beginners to get their heads around the different semantics of these two language constructs, we will largely defer discussion of variables to a later chapter. However, we will encounter a special case later in this chapter, in the context of for-loops.

5.1 Symbolic constants

We're going to turn next to a discussion of *symbolic constants*. Here's an example of a constant definition in VHDL.

```
architecture a1 of constExample is
constant wordSize: integer := 8;
signal x: std_logic_vector(wordSize-1 downto 0);
begin
...
end a1;
```

Here, we are defining `wordSize` to be an integer constant with the value 8. In the next line, we are using `wordSize` in the declaration of a signal vector.

In a large circuit, we might have many signals that must all have the same length, so it makes sense to define them using a symbolic constant, rather than a literal value. This makes it clear to someone reading the code that these signals are all really supposed to have the same length. It also allows us to easily change the lengths of all the signal vectors, by modifying the constant defintion, rather than editing lots of different signal declarations.

Now, in the previous example, the `wordSize` constant can be used anywhere within the architecture where it is declared, but suppose we wanted to use a constant in an entity declaration, or in several different circuit components. To do this, we need to use a *package*. Here is a simple example.

```
package commonConstants is
    constant wordSize: integer := 8;
end package commonConstants;
```

In general, a package contains a collection of definitions that can include constants, user-defined types, functions and procedures (more about these later). Packages are generally defined in a separate file and included in a `library` of related package declarations. In order to use the definitions in a package, we need to include a `use` statement right before any VHDL module that references the definitions in the package. For example,

```
use work.commonConstants.all;
entity ...
architecture ...
```

Here, the `use` statement specifies all definitions in `commonConstants`, which is included in the library `work`. `Work` is a default library that is used to hold packages defined within a given VHDL project.

Most VHDL projects also make use of a standard library defined by the *Institute for Electrical and Electronics Engineers* (IEEE). This can be specified by including the following statements immediately before the declaration of any VHDL module.

```
library ieee;
use ieee.std_logic_1164.ALL;
```

```
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
```

Here, the `library` statement declares the intention to use packages in the `ieee` library, and the subsequent `use` statements incorporate all definitions found in the three specified packages. We'll defer discussion of the contents of these packages.

VHDL allows us to define new signal types. Here's an example of an *enumeration type*.

```
type color is (red, green, blue, black, white);
subtype primary is color range red to blue;
signal c1, c2, c3: color;
signal p: primary;
```

The first line introduces a new type called `color` with five allowed values, each of which is given a symbolic name. The second line defines a `subtype` called `primary` which includes the colors `red`, `green` and `blue`. The last two lines contain signal declarations using the new types. With these definitions, we can write the following code.

```
c1 <= red;
p <= green;
if c2 /= green then
    c3 <= p;
else
    c3 <= blue; p <= blue;
end;
```

Note, the assignment `p <= black` is not allowed, since `black` is not a valid value for signals of type `primary`.

Before moving on to the next topic, it's worth noting that VHDL has an extensive *type system* to enable users to define their own data types. In fact, the types `std_logic` and `std_logic_vector` are examples of types that are not built into the language, but which have been defined using the type system. These type definitions are included in the IEEE library which is

routinely included as part of computer-aided design tool suites. While we're on the subject of `std_logic` it's worth noting that signals of type `std_logic` may have values other than 0 or 1. In fact, there are nine possible values including one that essentially means "undefined". Now these "extra" values are useful in the context of simulation, as they make it easier to identify situations in which a circuit definition fails to explicitly define the value of a signal. We won't discuss them in detail at this point, but it's important to know that there are more possiblities for `std_logic` signals than just 0 or 1.

It's also important to know that the VHDL is a *strongly-typed language*, meaning that signals that are combined in the same expression must have compatible types. This requires that we exercise some care when constructing signal assignments, to ensure that the type-matching rules of the language are satisfied. There are occasions when we will find it necessary to explicitly convert a signal from one type to another. The standard library provides mechanisms for handling such situations when they occur.

5.2 For and case statements

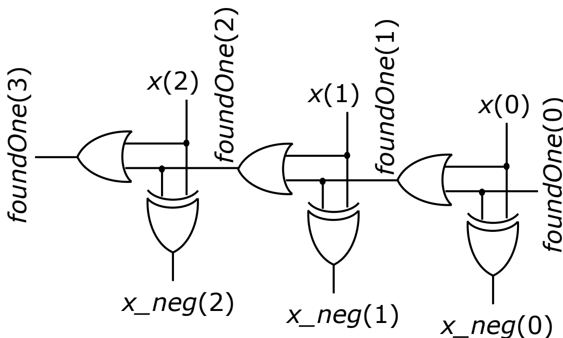
Now let's turn to another type of statement, the *for-loop*. Like conventional programming languages, VHDL includes statements that can be used to iterate through a set of index values. Now in a conventional language, an iteration statement means "repeat the following group of program statements, multiple times in sequence". In VHDL, an iteration statement means "replicate the circuit defined by the following statements". Let's consider an example.

```
-- Negate a 2s-complement input value
-- Output bit i is flipped if input bits 0..i-1
-- include a 1
entity negate is port (
    x : in std_logic_vector(wordSize-1 downto 0);
    x_neg: out std_logic_vector(wordSize-1 downto 0));
end negate;
architecture arch of negate is
```

```
signal foundOne: std_logic_vector(wordSize downto 0);
begin
    process(x, foundOne) begin
        foundOne(0) <= 0;
        for i in 0 to wordSize-1 loop
            x_neg(i) <= foundOne(i) xor x(i);
            foundOne(i+1) <= foundOne(i) or x(i);
        end loop;
    end process;
end arch;
```

The input to this circuit is a 2s-complement number and the output is the negation of the input value. Recall that to negate a 2s-complement number, we first find the rightmost 1, then flip all bits to the left of the rightmost 1.

In the VHDL architecture, the signal vector `foundOne` has a 1 in every bit position to the left of the first 1 in the input and a 0 in all other bit positions. The circuit implemented by the first three iterations of the for-loop is shown below.



Notice how each iteration defines a circuit consisting of a pair of gates and that these gates are linked to each other through the `foundOne` signal vector. Also note that the number of loop iterations is determined by the constant `wordSize`. Since we're using the loop to define multiple copies of a circuit, the number of copies must be a constant. Because the number of

iterations is a constant, we could omit the loop altogether and simply use a series of assignments.

```
process(x, foundOne) begin
    foundOne(0) <= 0;
    x_neg(0) <= foundOne(0) xor x(0);
    foundOne(1) <= foundOne(0) or x(0);
    x_neg(1) <= foundOne(1) xor x(1);
    foundOne(2) <= foundOne(1) or x(1);
    x_neg(2) <= foundOne(2) xor x(2);
    foundOne(3) <= foundOne(2) or x(2);
    ...
end process;
```

We can view the for-loop as just a shorthand way to write these repetitive statements. There is another way we can write this for-loop, using an `exit` statement.

```
process(x) begin
    x_neg <= not x;
    for i in 0 to wordSize-1 loop
        x_neg(i) <= x(i);
        if x(i) then exit; end;
    end loop;
end process;
```

This version does not use the `foundOne` signal In this case, the signal `x_neg` is assigned a default value of *not x*. The body of the loop overrides this default assignment for all bit positions up to the rightmost 1 in `x`. Now, the exit statement can be a little bit confusing, because it seems like it is intrinsically associated with the notion of sequential execution of the loop statements. It's not so easy to see how it defines a circuit. One way to think about this is to view each if-statement as defining an "exit condition" and the other statements in the loop body are only "activated" if the exit condition is not true. Here's a version of the process that illustrates this idea.

```
    process(x) begin
        x_neg <= not x;
        exit(0) <= '1';
        for i in 0 to wordSize-1 loop
            if exit(i) = '0' then
                x_neg(i) <= x(i);
            end;
            exit(i+1) <= x(i) or exit(i);
        end loop;
    end process;
```

The circuit defined by this specification is logically equivalent to the one shown earlier. Before leaving this example, it's worth noting that in this case the loop is simple enough that we can rewrite that architecture without any loop at all.

```
     process(x, foundOne) begin
         foundOne(0) <= 0;
         x_neg <= foundOne xor x;
         foundOne(wordSize downto 1) <=
                 x or foundOne(wordSize-1 downto 0);
     end process;
```

Let's look at another example of a loop. Here, we are using a loop to implement a four digit BCD incrementer.

```
entity bcdIncrement is port(
    a : in std_logic_vector(15 downto 0);
    sum : out std_logic_vector(15 downto 0));
end bcdIncrement;
architecture a1 of bcdIncrement is
signal carry: std_logic_vector(4 downto 0);
begin
    process(a, carry) begin
        carry <= "00001";
```

```
        for i in 0 to 3 loop
            sum(4*i+3 downto 4*i) <= a(4*i+3 downto 4*i)
                                     + ("000" & carry(i));
            if carry(i)='1' and a(4*i+3 downto 4*i)="1001" then
                sum(4*i+3 downto 4*i) <= (others => '0');
                carry(i+1) <= '1';
            end if;
        end loop;
    end process;
end a1;
```

Here, the addition is done on 4 bit segments that correspond to BCD digits. The case that triggers a carry into the next BCD digit is handled by the if-statement. Again, keep in mind that the loop defines repeated copies of a circuit. You can always expand the loop in order to understand exactly what circuit it defines.

Before leaving the for-statement, it's important to note that the loop index in a for-statement is not a signal. That is, there is no wire (or group of wires) in the constructed circuit that corresponds to the loop index. Also note that there is no declaration of the loop index. It is simply defined implicitly by the loop itself, and it is only defined within the scope of the loop. A loop index is a special case of a VHDL *variable*. We'll see other uses of variables in a later chapter.

Next we turn to the *case statement*, which allows one to select among a number of alternatives, based on the value of an expression. Here's an example.

```
q <= '1';    -- provide default value for q
case x+y is
    when "00"   => p <= u; q <= y;
    when "01"   => p <= v; q <= u;
    when "10"   => p <= y xor z;
    when others => p <= '0';
end case;
```

The expression in the case statement determines which of the `when` clauses is active. The expressions in the `when` clauses must either be constants or constant-expressions, and they must all have the same type as the selection expression that follows the `case` keyword.

By the separation principle, we can replace this case statement with two similar case statements, one containing only assignments to `p` and the other containing only assignments to `q`. Now each such case statement is equivalent to a selected signal assignment for that signal. So, each such case statement can be implemented by a circuit similar to the one we saw earlier for the selected signal assignment. A case statement containing assignments to several signals can be implemented using one such circuit for each signal.

In this example, notice that signal `p` appears in all the `when` clauses, but `q` does not. In particular, the case statement defines a value for `q` only when `x+y=00` or `01`. The default assignment to `q` ensures that `q` is well-defined in all other cases.

5.3 Synchronous and Asynchronous Assignments

We have seen how a VHDL process can be used to define *combinational circuits*. What about sequential circuits? Most of the time when designing sequential circuits, we're really interested in clocked sequential circuits. The key to defining clocked sequential circuits in VHDL is the synchronization condition. Here is a small example of a serial parity circuit.

```
entity serialParity is port (
    clk, reset: in std_logic;
    dIn: in std_logic;
    parityOut: out std_logic);
end serialParity;
architecture arch of serialParity is
signal parity: std_logic;
begin
    process (clk) begin
        if rising_edge(clk) then
```

```
            if reset = '1' then
                parity <= '0';
            else
                parity <= dIn xor parity;
            end if;
        end if;
    end process;
    parityOut <= parity;
end arch;
```

The `parityOut` output of this circuit is high, whenever the number of 1s that have been observed on the `dIn` input is odd. The reset signal causes the circuit to "start over".

Now the if-statement in the architecture's process includes the condition `rising_edge(clk)`. This is an example of a *synchronization condition*. This synchronization condition specifies that all assignments that appear within the scope of the if-statement should happen only when the signal `clk` is making a transition from low to high. Now thinking about the circuit defined by this code, the implication is that the internal signal `parity` must be the output of a flip flop. We will use the term *synchronous assignment* to refer to any signal assignment that appears within the scope of a synchronization condition. Other assignments are referred to as *asynchronous assignments*. When using VHDL to define a combinational circuit, all assignments are asynchronous, but a clocked sequential circuit will generally have a combination of both synchronous and asynchronous assignments.

Now, if a specific signal, say `sig`, appears on the left side of a synchronous assignment, then `sig` must be the output of a flip flop. Consequently, it usually does not make sense for `sig` to appear in any asynchronous assignments. So as a general rule, a given signal can appear in synchronous assignments or asynchronous assignments, but not both. Consequently, we can also think of signals as being either synchronous or asynchronous. Note that if a given signal is synchronous, assignments to that signal can only appear within the scope of "compatible" signal conditions. So for example `falling_edge(clk)` and `rising_edge(someOtherClk)` would not be compatible with `rising_edge(clk)`.

Processes that include synchronization conditions often include a single synchronization condition with a scope of that extends over the entire body of the process. The serial parity circuit includes such a process. We refer to such processes as *purely synchronous*. Now this brings us back to the subject of the sensitivity list. In a purely synchronous process, it's only necessary to include the clock signal (the one used in the synchronization condition) in the sensitivity list. Recall that the sensitivity list includes all signals that a circuit simulator should "pay attention to", since when changes to these signals occur, process outputs may change. In a purely synchronous process, all outputs to a circuit change when the clock signal changes, so it's sufficient to include only the clock signal in the sensitivity list.

Now, it's possible for a given process to have both synchronous and asynchronous assignments. Here's an example.

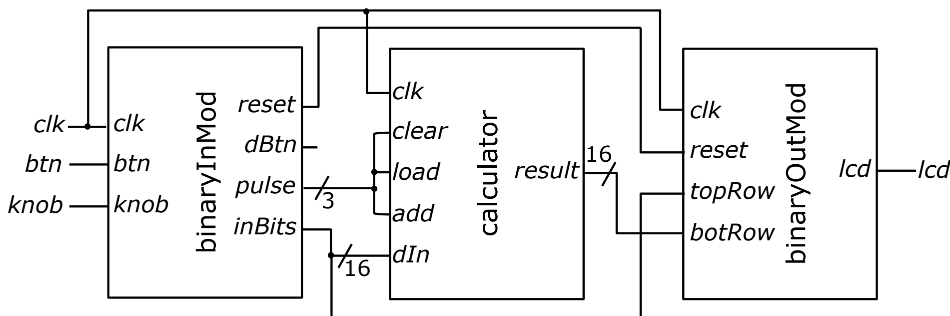
```
process (clk,t,x) begin
    if rising_edge(clk) then
        t <= s;
        if en = '0' then
            s <= '0'; t <= '1';
        else
            s <= x xor s;
        end if;
    end if;
    y <= t and x;
end process;
```

Note that the assignment to y falls outside the scope of the synchronization condition, hence it is an asynchronous assignment. Because this process includes an aynchronous assignment, its sensitivity list must include the sigals that appear on the right-hand side of that assignment. In general all signals that are used within the asynchronous part of such a mixed process must be included in the sensitivity list. Note that we could easily split this process into two separate processes, one for the synchronous signals and one for the asynchronous signals. Indeed, many designers prefer to keep the synchronous and asynchronous parts of their design separate in this way. This

is certainly a legitimate choice, but since mixed processes are permitted by the language, it's important to be aware of the implications of using them.

5.4 Structural VHDL

In an earlier chapter, we described a simple binary calculator circuit. This circuit can be implemented using an FPGA prototype board, and two other circuit components, an input module called `binaryInMod` and an output module called `binaryOutMod`. The circuit formed by combining these components is illustrated below.



VHDL provides language constructs to enable designers to specify groups of components and the signals that connect them. Here is an example of a VHDL module that illustrates how these constructs can be used to define the interconnections shown in the diagram.

```
entity top is port(...); end top;

architecture a1 of top is
component   calculator port( ... ); end component;
component  binaryInMod port( ... ); end component;
component binaryOutMod port( ... ); end component;

signal reset, clear, load, add: std_logic;
signal dBtn, pulse: std_logic_vector(3 downto 1);
```

```
signal inBits, outBits: word;
begin
    imod:  binaryInMod port map(clk,btn,knob,
                                reset,dBtn,pulse,inBits);
    calc:   calculator port map(clk,clear,load,add,
                                inBits,outBits);
    omod: binaryOutMod port map(clk,reset,
                                inBits,outBits,lcd);

    clear <= pulse(1); load <= pulse(2); add <= pulse(3);
end a1;
```

First notice that this code is defining an entity called `top` that contains the other components. The most important part of the architecture of `top` is the series of three *component instantiation statements*. Let's look at the second of these, which is identified by the label `calc`. This statement instantiates a copy of the `calculator` module and includes a `port map` that determines how the interface signals of the `calculator` are connected to local signals defined within `top`. The first signal in the port map is connected to the first signal in the entity specification for the `calculator`, the second is connected to the second signal in the entity specification, and so forth. The other two component instantiation statements are similar.

The port maps in our example associate the local signals within top to the interface signals of the respective components based on the position of the interface signals within the component's entity specifications. We say that this style of port map uses *positional association*. Port maps may also use *named association* in which local signal names are mapped explicitly to the signal names in the entity specification. For example, we could have written the following for the calculator.

```
calc:   calculator port map(clk => clk,
                  dIn => inBits, result => outBits,
                  clear => clear, load => load,
                  add => add);
```

Here, each port is specified by name, where the form is `component_interface_s` `=> local_signal`. When using named association, the order in which the signal mappings are listed does not matter. Although named association is a little more trouble to write, it can make specifications more clear to a reader, in the case of components with a large number of interface signals.

Notice that the architecture starts with a series of *component declarations*, one for each distinct type of component that is used within the architecture. We have abbreviated the component declarations here, but in general, each component declarations contain the same information as the corresponding entity declaration. Even though they are somewhat redundant, the language does require that they be included, in every architecture that uses a given component.

Circuit specifications that use component instantiation statements are said to use *structural* VHDL. This style of circuit specification is mostly used to wire together large circuit blocks, as we have done here. However, structural VHDL can also be used to specify circuits using simple components like gates. When structural VHDL is used in this way, it essentially provides a way to encode a schematic representation of a circuit using text. While there is rarely a good reason for designers to use structural VHDL in this way, it is sometimes used in this way by computer-aided design tools, as an intermediate representation of a circuit that has been synthesized from a higher level specification.

Chapter 6

Building Blocks of Digital Circuits

In this chapter, we're going to look a bit more closely at the basic elements used to implement digital circuits, and we'll see how they can be used to construct larger components that serve as building blocks in larger digital systems.

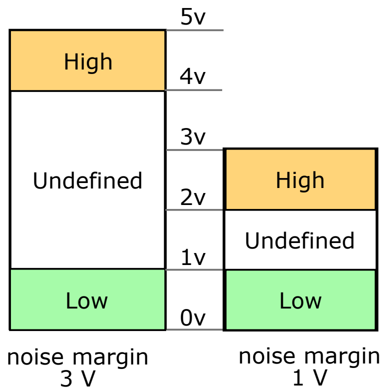
6.1 Logic Gates as Electronic Components

Up to now, we've been viewing gates as idealized logical elements. Of course, in reality, they must implemented by electronic circuits that are constrained by physical properties. Consequently, real digital components only approximate the idealized behavior and it's important to keep this in mind.

Let's start by considering how real digital circuits represent logic values. All electronic circuits involve voltages and currents and these can be used to represent information. Now voltages and currents are continuous, real-valued quantities, not integers and certainly not logical values. Digital circuits use ranges of voltage values to represent the two logic values. So for example, if our digital gates use a power supply voltage of 5 volts, we might define the range of values from 0 to 1 volt as logic 0, and the range from 4 to 5 volts

as logic 1. What about the range between 1 and 4 volts? This is a sort of no-man's land. Voltage values in this range do not correspond to either logic 0 or logic 1, and while our signals must pass through this range when they switch between the low and high ranges, signals do not normally remain in this range for long.

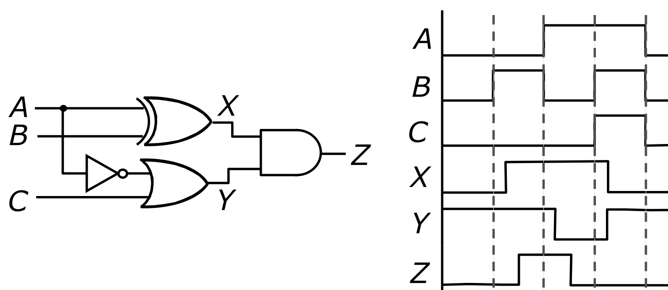
Now, it's important to separate the low range from the high range because real circuits are subject to noise that can distort the voltages and currents in a circuit. If we want our circuits to operate reliably, it's important that noise not cause us to mistakenly interpret a 0 as a 1 or vice versa. The intermediate area between the low and high voltage ranges operates as a buffer that makes it less likely that our circuits will operate incorrectly due to the effects of noise. The size of that intermediate range is referred to as the *noise margin* of the circuit. Now in the 1970s, most digital circuits used a power supply voltage of 5 volts, but as integrated circuit fabrication processes allowed gates to become smaller, it became desirable to reduce the power supply voltage to enable faster operation. That has led to a gradual shrinking of noise margins as illustrated below.



Now, you might ask why is it that digital circuits use just two voltage levels. Why not define three ranges, say 0 to 1 volt, 2 to 3 volts and 4 to 5? This would allow us to construct circuits based on ternary logic (that is base 3). Now, it is possible to do this, and there are some arguments in

favor of using ternary logic, instead of binary. In practice however, ternary logic is rarely used and the main reason is that it's simply more difficult to implement gates that operate reliably using ternary logic. The required circuits are more complex and the available noise margins are smaller. Thus, while it's tempting to think we could construct digital systems that do not represent information in binary, these practical considerations make binary the only sensible choice in most situations.

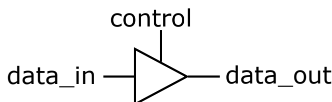
Now because voltages and currents are physical quantities, they cannot change instantaneously from one value to another. It takes time for a signal to transition between the high and low voltage ranges. There are a number of factors that affect the time required, some of which we will study in a later chapter. For now, just keep in mind that signal transitions take time. The effect of this is that signal transitions at a gate input are not immediately reflected at the gate output. The output transition is delayed slightly and in larger circuits, these delays. add up and ultimately determine the performance that the circuit is able to achieve. This is illustrated in the *timing diagram* shown below.



Timing diagrams like this are used to describe the timing relationships among different signals in a circuit. They can also be used to define the *required relationships* among the signals in a circuit we're designing. Most of the time when we draw timing diagrams, we'll draw them as if the gate delays were zero, but it's important to keep in mind that this is just an idealization of the actual circuit behavior.

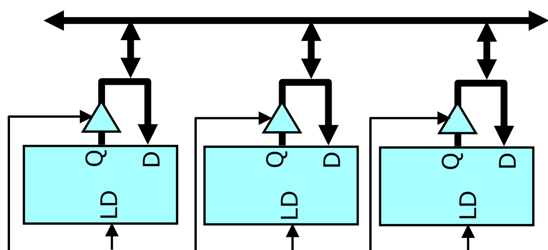
Now it turns out that digital circuits can do some things that fall outside the domain of mathematical logic. Normally, an electronic gate either makes

its output signal have a high voltage or a low voltage. However, there is a third possibility, which is that the gate can simply let the output "float" in a disconnected state. The *tri-state buffer* is a special kind of gate that implements this third option.



When the *control input* of a tri-state buffer is high, the data output is equal to the data input. When the control input is low, the output is disconnected from the input. That means that the voltage level is not being controlled by the gate, and is essentially left to float. When a tri-state buffer leaves its output disconnected, we say that it is in the *high impedance state* and we sometimes say that the output of the gate is equal to a special value Z.

Now, why would we want to have a gate that behaves in this way? Well, we can use such gates to let different parts of a large circuit share a set of set of wires and use them to communicate in a flexible fashion. This is illustrated below.



The diagram shows three registers connected by a shared set of wires called a *bus*. If the tri-state buffers at the output of the first register are enabled (that is, their control inputs are all high), and the tri-state buffers of the other two registers are disabled, the data in the first register will appear on the bus, allowing either of the other registers to store the value,

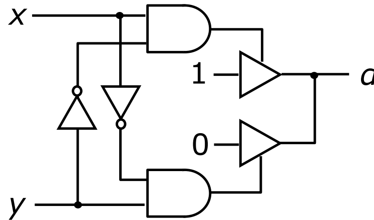
effectively transferring it from one register to another. So long as no two sets of tri-states are enabled at the same time, we can use the shared bus to pass values among the different registers. In larger digital systems, shared buses are often used to pass information among different system components. For this kind of communication scheme to work effectively, we need some way to ensure than no two components attempt to put data on the bus at the same time, and we need to make sure that we only load data from the bus when someone has put data on the bus. If we load a register from a bus that is not being actively controlled by some other component, the voltage values on the bus wires may not correspond to valid logic values.

Here's an example of how we can specify the use of a tri-state buffer in a VHDL circuit specification.

```
entity tristate is port(
    x,y : in std_logic;
    d: out std_logic);
end tristate;

architecture a1 of tristate is
begin
    process (x,y) begin
        if x > y then d <= '1';
        else d <= 'Z';
        end if;
    end process;
    process (x,y) begin
        if x < y then d <= '0';
        else d <= 'Z';
        end if;
    end process;
end a1;
```

First, notice the assignment `d <= 'Z'` that appears in both processes. The value 'Z' is a special value that indicates that the process is placing signal `d` in the high impedance state. To implement this specification, a

circuit synthesizer must instantiate a tri-state buffer for each process, with shared signal `d` as the output of both buffers. Here is a diagram of a circuit that implements this specification.



Notice also that we are violating our usual rule of allowing only one process to control a signal. Signals controlled by tri-states are an exception to this rule, since it is possible for processes to safely share a signal using tri-states. However, it's important to recognize that it's the designers responsibility to ensure that the signal is shared safely. If the two processes attempt to assign complementary values to `d` at the same time, they will effectively create a direct electrical connection between the power supply and ground, leading to excessive current flows that can destroy the physical device.

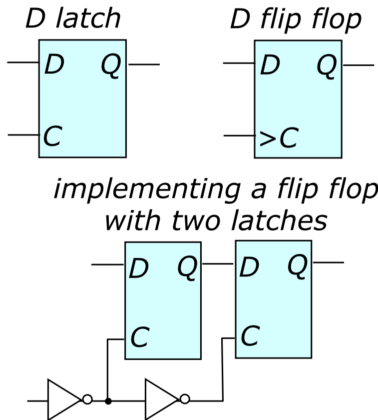
6.2 Storage Elements

In an earlier chapter, we introduced the *D flip flop*, which is the key storage element used to implement digital circuits. The flip flop stores a single bit of information when its clock input makes a transition from low-to-high (or alternatively, high-to-low). A fundamental feature of a flip flop is that it is synchronized to the clock, that is, the value stored (and the flip flop output) change only when there is a clock transition.

Now, while the flip flop is a versatile and ubiquitous storage element, it is not the only kind of storage element used in digital circuits. Another useful element is the *D latch*. Like the flip flop, the latch has a data input and a clock input, but it responds to the clock input in a different way than the flip flop does. Specifically, whenever the clock input of a latch is high, the

value at the input of the latch is propagated directly to the output. We say that the latch is *transparent* in this case. When the clock input of a latch drops, the output holds the value that it had at the time the clock dropped. Changes to the data input that occur while the clock input is low have no effect on the output. We say that the latch is *opaque* in this case.

The symbols for the latch and flip flop are shown below, along with an implementation of a flip flop that uses a pair of latches.



Notice that when the clock input of this circuit is low, the first latch is transparent, while the second is opaque. When the clock input goes from low to high, the first latch becomes opaque, while the second becomes transparent. Because the second latch is transparent only when the first latch is opaque, the output of the second latch can only change when the clock makes its low to high transition.

Now it's important to observe that there this is an ambiguity in our definition of the behavior of both of these storage elements. Start by considering the flip flop. We said that a new value is stored in the flip flop whenever the clock makes a low-to-high transition. But what if that the data input is changing at the same time that the clock is making its transition? What value is stored in the flip flop in that case? Well the answer is that we can't be sure. The flip flop might store a 0, or it might store a 1, or *it might do neither*. If we take a real physical flip flop and change the value on the

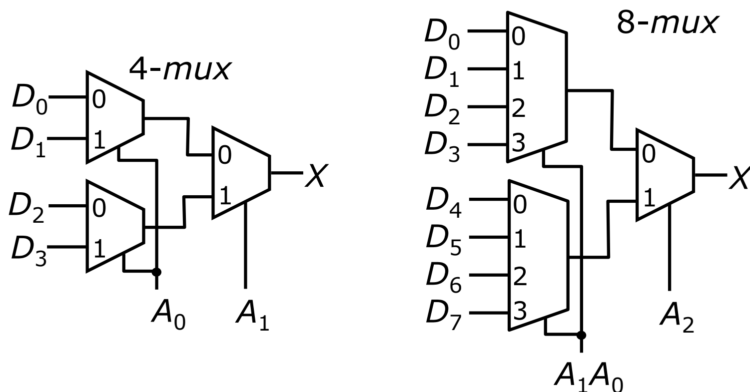
data input at the very instant the clock input is changing, the voltage at the flip flop output may get "stuck" at an intermediate voltage level, or it may oscillate between 0 and 1. When this happens, we say that the flip flop is *metastable*, and it can remain in this intermediate state for an indefinite amount of time. Usually a flip flop will leave the metastable state after just a short interval with the output resolving to either 0 or 1, but there is no way to predict how long this will take.

Metastability is an inherent property of real flip flops and can cause digital circuits to behave erratically. For this reason, when we design digital circuits we take some pains to ensure that the data inputs to flip flops are stable when the clock is changing. In a later chapter, we will discuss a set of timing rules that can be used to ensure that flip flops operate safely, but for now the key point to understand is that correct operation of flip flops depends on data inputs being stable during a clock transition. A similar observation can be made about latches. In the case of a latch, the critical time is when the latch is going from transparent to opaque (that is the clock input is going from high to low). Changing the value on the data input at this instant can cause the latch to become metastable.

6.3 Larger Building Blocks

In this section, we'll look at a series of larger building blocks that can be used to implement digital circuits. Circuit synthesizers often use these building blocks to implement circuits specified in a hardware description language.

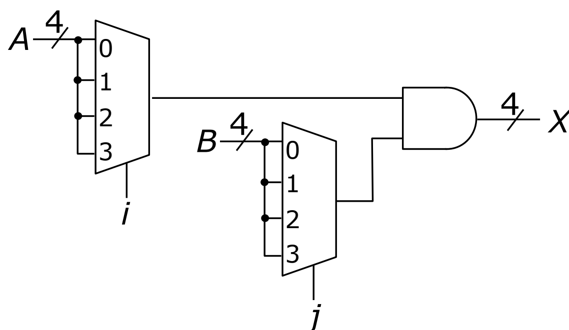
Let's start by considering the multiplexor. In general, a multiplexor is a device with k control inputs, 2^k data inputs and a single output. The data inputs are numbered $0, 1, \ldots, 2^k - 1$ and if the value on the control inputs is the integer i, input i is effectively connected to the output. That is, the data value on input i is propagated to the output. The logic equation defining the output of a 2-to-1 mux is simply $C'D_0 + CD_1$, so we can implement it directly using three gates and an inverter. Larger multiplexors can be implemented in a recursive fashion, using the 2-to-1 mux as a building block, as illustrated below.



Multiplexors can be used to implement selected signal assignments in VHDL. They can also be used to implement signal assignments that operate on individual bits from a logic vector. For example, consider the following assignment, where `A`, `B` and `X` are four bit signal vectors, and `i` and `j` are two bit signal vectors.

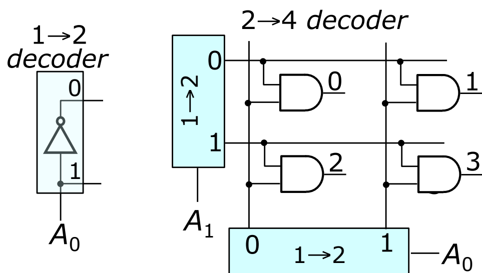
```
X <= A(i) and B(j);
```

This can be implemented by the circuit shown below.



The *decoder* is another building block that can be used to construct larger circuits. A decoder with k inputs has 2^k outputs that are numbered $0, 1, \ldots, 2^k - 1$. When the value on the inputs is the integer i, output i of the

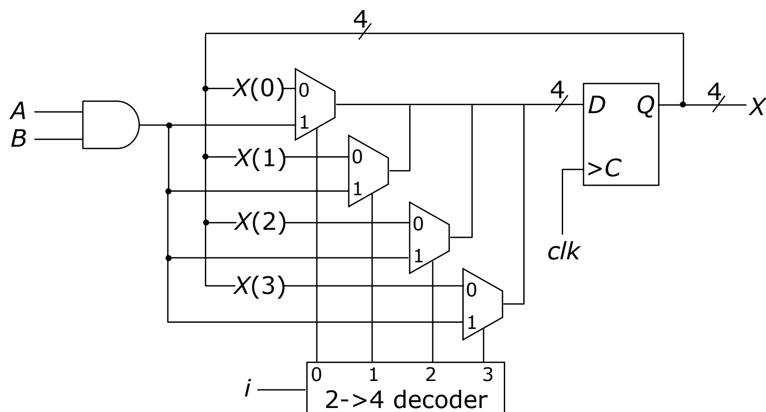
decoder is high, while the other outputs are all low. A 1-to-2 decoder can be implemented with just an inverter, while larger decoders can be constructed from two smaller ones plus an AND gate for each output, as shown below.



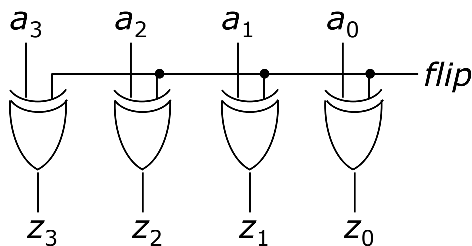
Decoders and multiplexors are closely related to one another. Indeed, one can implement a multiplexor using a decoder. Decoders can also be used to implement VHDL case statements and to control assignment to a selected bit of a value stored in a register. For example, Consider the code fragment below, where `X` is a four bit signal vector and `i` is a two bit signal vector.

```
if rising_edge(clk) then
    X(i) <= A and B;
end if;
```

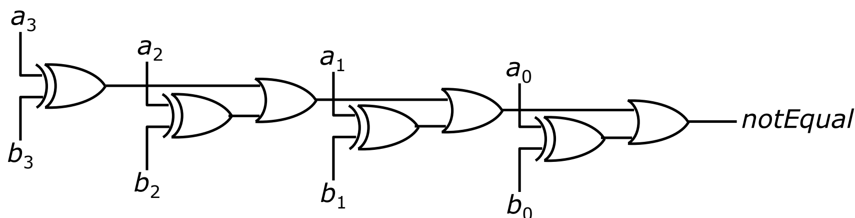
Here's a circuit that uses a decoder together with several multiplexors to implements this.



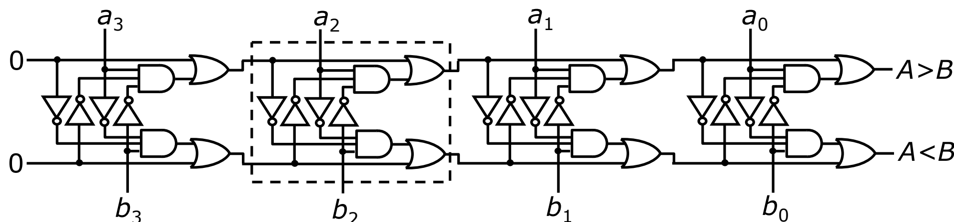
Here's a circuit that uses exclusive-or gates to flip all the bits in a signal vector, based on some condition.



Here's another circuit using exclusive-or gates that compares two values to determine if they are equal or not. Notice how the circuit is constructed from a linear array of repeated sub-circuits.

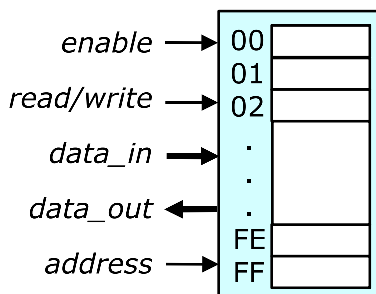


Here's another circuit that is constructed in a simlar way. This one compares two values to determine which is larger numerically.



This circuit compares the bits of the two input values in sequence, starting with the most significant bit. The first bit position where $a(i) \neq b(i)$ determines which of the values is larger.

We conclude this section with a brief discussion of *memory components*. As its name suggests, a memory is a storage element, but unlike a flip flop or latch, a memory component stores multiple bits, typically many thousands. An example of a memory component is shown below.



The memory contains an array of numbered storage locations, called *words*. The integer index that identifies a particular location is called its *address*. This memory component has an *address input*, a *data input* and a *data output*. The address input selects one of the words in the memory. To read from the memory, we raise the *enable* input and the *read/write* input. This causes the value stored at the location specified by the address inputs to be placed on the data output. To write to the memory, we raise the enable

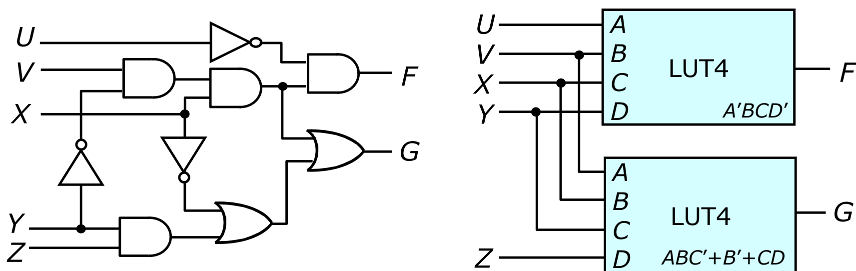
input, while holding the read/write input low. This causes the value on the data inputs to be stored in the location specified by the address inputs.

This particular memory component is asynchronous. That is, it is not synchronized to a clock. Synchronous memory components are also often used. These components have a clock input and all operations on the memory take place on the rising clock edge.

Notice that the interface to the memory constrains the way that it can be accessed. In particular, only one memory operation can take place at one time. Multi-port memory components are able to support more than one simultaneous operation (most often, two), using multiple sets of data, address and control signals.

6.4 Lookup Tables and FPGAs

The *Field Programmable Gate Array* (FPGA) is a configurable logic device that can be used to implement a wide variety of digital circuits. One of the key elements of an FPGA is a configurable logic element called a *Lookup Table* (LUT). A typical LUT has four inputs and a single output and can be used to implement any logic function on four inputs (some newer FPGAs use 6 input LUTs, but we will focus on the 4 input case, to keep things simple). As its name suggests, a LUT4 can be implemented using a small memory, with the LUT inputs serving as the address inputs to the memory. We can think of the LUT as storing the truth table for some logic function. By changing the information stored in the LUT we can change the logic function that it implements. This is the key to using LUTs to implement logic. The figure shown below provides an example of how LUTs can be used to implement logic circuits.



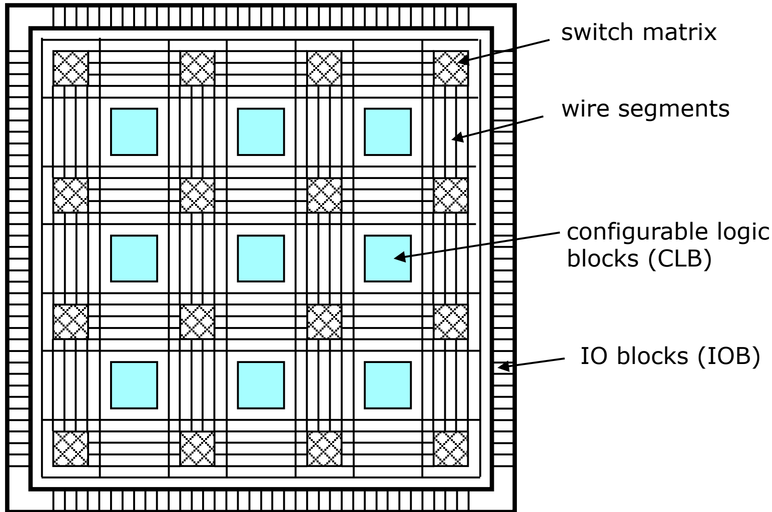
If we view the inputs to the LUT as address inputs with A being the most significant bit, the LUT for output F will have a 1 in bit 6 and 0s in all other bits. The LUT for output G will have 1s in bits 0-3, 7, 8-13 and 15.

Because LUTs are used to implement all the combinational circuits in an FPGA we're often interested in knowing how many LUTs it takes to implement a given component. For many common components it's easy to determine how many LUTs they require. For example, a 2 input mux can be implemented with a single LUT since it has three inputs and a single output. A 4 input mux can be implemented with three LUTs, using the recursive construction discussed earlier.

It's natural to ask if there is a straight-forward method for determining the minimum number of LUTs required to implement a given circuit. Unfortunately, the answer to that question is no, but there are a couple simple rules of thumb that set lower bounds on the number of LUTs required. First, since a LUT has a single output, a circuit with n outputs requires at least n LUTs. Similarly, a circuit with m inputs requires at least $\lceil m/4 \rceil$ LUT4s. We can actually improve on this latter bound by noticing that in a circuit with m inputs, n outputs and k LUT4s, $k-n$ LUT outputs must be connected to LUT inputs. Hence, $m+(k-n) \leq 4k$ or $k \geq \lceil (m-n)/3 \rceil$. Note that these are only lower bounds, and many circuits require far more LUTs than indicated by the bounds. Still, they provide a useful starting point when trying to estimate the number of LUTs that a given function might require.

Now, to enable them to implement a wide range of digital systems, FPGAs contain a number of other components, organized into a fairly elaborate structure. The, highly simplified diagram below, provides a rough indication

of how these devices are organized.

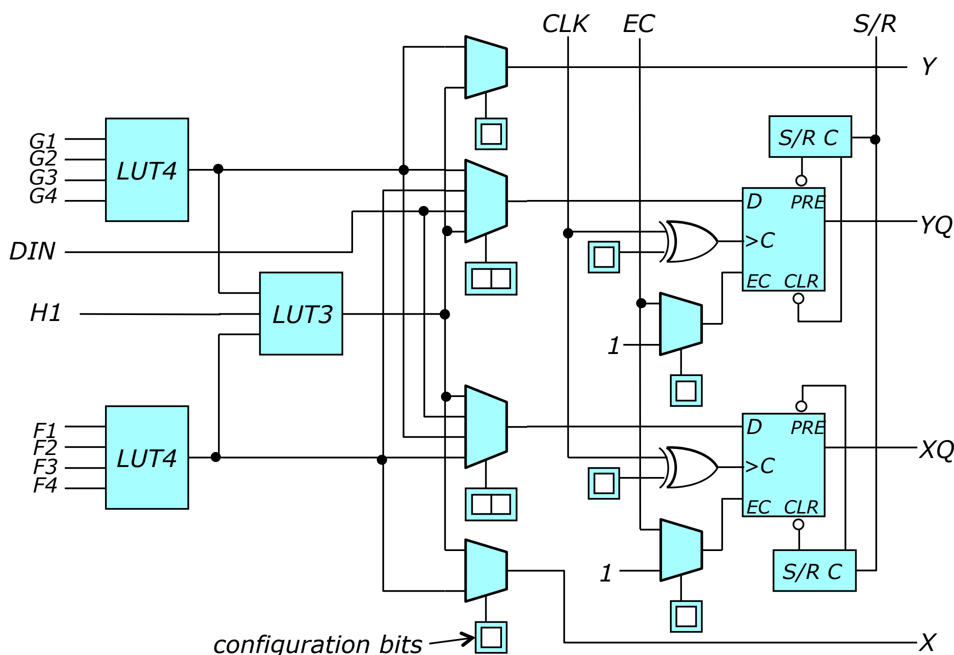


A Xilinx FPGA is organized around a basic building block called a *Configurable Logic Block* (CLB). CLBs can be connected to one another using programmable connections that are implemented by using wire segments that are connected to one another using switch matrix components. We won't discuss the details here, but the ability to connect the CLBs together in a flexible fashion is essential to enabling the FPGA to implement a wide variety of digital systems. The FPGA used in our prototype board has several thousand CLBs and larger devices have hundreds of thousands.

The figure shown below is a simplified diagram of a Xilinx CLB.

At the left edge of the figure are two LUT4 s and a third LUT with three inputs. Near the center of the CLB are four multiplexors that can be configured to propagate the LUT outputs in a variety of ways. Toward the right side of the figure is a pair of *D* flip flops that can store information from either of the LUT4 s, the three input LUT or a direct data input. Configuration bits can be used to determine whether a flip flop operates on the rising or falling clock edges, and how it responds to a global reset signal.

Modern FPGAs include other components, in addition to CLBs. In par-

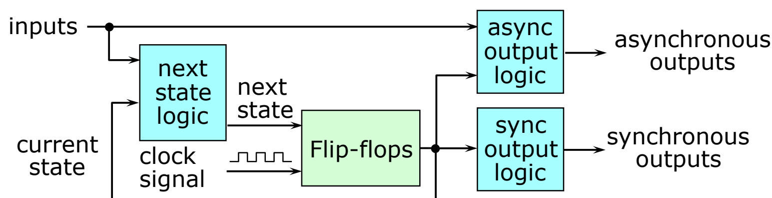


ticular, they are often equipped with memory blocks that can be used for a variety of purposes. For example, the FPGAs on our prototype board have 20 memory blocks, each with approximately 18 Kbits of memory. Larger devices have hundreds of such memory blocks.

Chapter 7

Sequential Circuits

In this chapter, we're going to look more closely at the design of sequential circuits. Earlier, we observed that there are two primary categories of digital circuits: combinational circuits, and sequential circuits. The outputs of a combinational circuit are always a direct function of the current input values; that is the outputs do not depend on past input values. Sequential circuits store data, meaning that their outputs may depend on previous input values, not just current values. In this book, we will focus on *clocked sequential circuits* in which data is stored in flip flops. Here's a diagram that illustrates the general structure of a clocked sequential circuit.



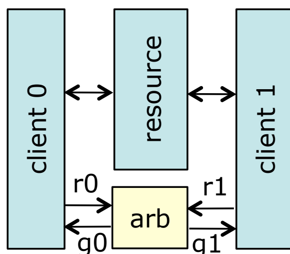
The flip flops at the center of the diagram store one or more bits of information. The outputs of these flip flops are referred to as the *current state* of the circuit. The input signals to the flip flops are referred to as the *next state*.

The block labeled *next state logic* is a combinational circuit that deter-

mines what the next state will be, based on the current input values and the current state. There are two output blocks, which are also combinational circuits. The *asynchronous output logic* block determines the values of those outputs that are functions of both the current inputs and the current state. The *synchronous output logic* block determines the values of those outputs that are functions of the current state alone. The synchronous outputs will change only in response to a clock transition, while the asynchronous outputs can change at any time.

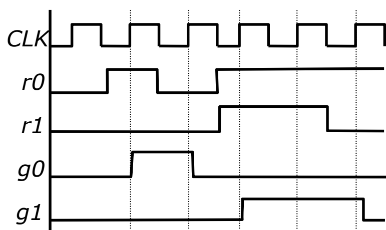
7.1 A Fair Arbiter Circuit

Sequential circuits are often used as control elements in larger digital systems. Sucn controllers are usually referred to as *state machines*. We'll illustrate this by describing an *arbiter*, which is a circuit that controls access to some shared resource, like a memory perhaps. The figure below illustrates a typical application of an arbiter. Two separate parts of a larger digital circuit need



to access the shared resource in the middle of the diagram. We refer to these two parts as *client 0* and *client 1*. The arbiter has two signals connecting it to each client, a *request signal* and a *grant signal*. To request exclusive use of the resource, a client raises its request signal high. The arbiter gives it permission to use the resource by raising its grant signal. The client holds its request signal high while it is using the resource, then drops it low when it is finished. In order to treat the two clients fairly, whenever the arbiter gets simultaneous requests from both clients, it grants access to the client that has been *least recently served*. This behavior is illustrated in the *interface*

timing diagram shown below.

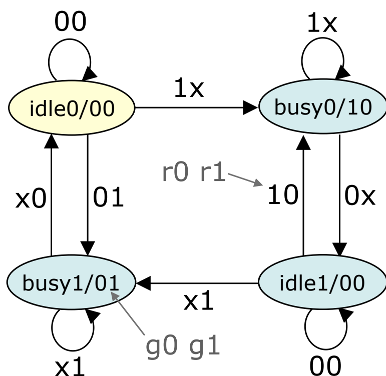


Note that when the two clients make simultaneous requests, client 1 is granted access, since client 0 used the resource most recently.

In order to implement the desired behavior, the arbiter needs to remember a few pieces of information. First, it needs to know if the resource is in use or not, and if it is in use, which client is using it. When the resource is idle, it also must remember which client should go next, in the case of concurrent requests from both clients. We can keep track of this information by defining four *states* for our controller.

- In the *busy0* state, the resource is being used by client 0.
- In the *busy1* state, the resource is being used by client 1.
- In the *idle0* state, the resource is not being and client 0 will get to go next, if simultaneous requests are received from both clients.
- In the *idle1* state, the resource is not being and client 1 will get to go next, if simultaneous requests are received from both clients.

We can specify the behavior of the circuit precisely using a *state transition diagram*, as shown below.



The ovals in the diagrams correspond to the four states of the circuit. The ovals are labeled with the state name and the values of the two output signals. So for example, in the two idle states, the grant outputs are both low, while in each of the two busy states, one grant is high, while the other is low. The arrows connecting the ovals indicate transitions between the states. These are labeled by the input signals, so for example, if circuit is in state *idle0* and the two request inputs have values 0 and 1 respectively, then the circuit will move from the *idle0* state to the *busy1* state. Some of the labels use an x to indicate a don't care condition. So for example, when in state *idle0*, if input *r0* is high, we'll go to the *busy0* state next regardless of the value of input *r1*. This reflects the fact that client 0 gets priority when we're in the *idle0* state. Take a few minutes to study the state diagram and make sure you understand the significance of all the transitions, including the "self-loops". Diagrams like this are a good way to work out the proper relationship among the states in a state machine controller, and are an important first step in the design process for such controllers.

An alternate way to specify the behavior of a state machine is using a *state table*. A state table contains exactly the same information as the state diagram. It just presents it in a different format, and some people find this form easier to work with. Here is the state table for the arbiter circuit.

current state	inputs r0 r1	outputs g0 g1	next state
idle0	00	00	idle0
	1x	00	busy0
	01	00	busy1
busy0	1x	10	busy0
	0x	10	idle1
idle1	00	00	idle1
	x1	00	busy1
	10	00	busy0
busy1	x1	01	busy1
	x0	01	idle0

Given a state diagram or state table, it's easy to write a VHDL specification for the corresponding circuit. Here is the specification for the arbiter.

```
entity fairArbiter is port(
    clk, reset: std_logic;
    request0, request1: in std_logic;
    grant0, grant1: out std_logic);
end fairArbiter;
architecture a1 of fairArbiter is
type stateType is (idle0, idle1, busy0, busy1);
signal state: stateType;
begin
    process(clk) begin
        if rising_edge(clk) then
            -- next state logic
            if reset = '1' then
                state <= idle0;
            else
                case state is
                when idle0 =>
                    if request0 = '1' then
                        state <= busy0;
```

```
                    elsif request1 = '1' then
                        state <= busy1;
                    end if;
                when busy0 =>
                    if request0 = '0' then
                        state <= idle1;
                    end if;
                when idle1 =>
                    if request1 = '1' then
                        state <= busy1;
                    elsif request0 = '1' then
                        state <= busy0;
                    end if;
                when busy1 =>
                    if request1 = '0' then
                        state <= idle0;
                    end if;
                when others => -- nothing else
                end case;
            end if;
        end if;
    end process;
    -- output logic
    grant0 <= '1' when state = busy0 else '0';
    grant1 <= '1' when state = busy1 else '0';
end a1;
```

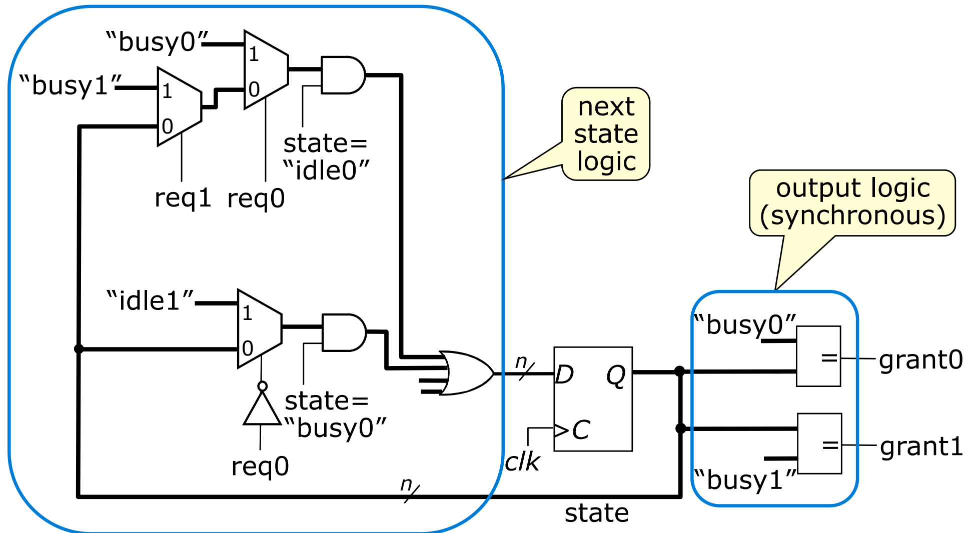
There are several things worth noting about this specification. First observe how an enumeration is used to define a set of symbolic state names. By defining state names in this way, the remainder of the code can use the symbolic names, making the specification easier to understand and get right.

Next, notice that the process is used to define the next state logic. A case statement is a good way to enumerate the various cases. If statements can then be used to determine which state to go to next. Also, note that the VHDL code does not specify the self-loops. The synthesizer will automatically

generate the necessary circuitry to implement the self-loops, but we do not need to show it explicitly in the code.

Finally, note that output logic appears outside the process. This is a very typical way to specify the outputs. Observe that even though the VHDL assignments are not synchronous assignments, the grant signals only change when the clock changes since they are determined entirely by the state signals (which are synchronous). Some circuits have more complex output logic than we have here. In such cases, it may make sense to define a separate combinational process for the output signals. It's also possible to define the output signals in the same process used to define the state signal, but you need to be careful when doing this, to make sure that the resulting output timing is consistent with what's required for your circuit. If you assign values to your outputs within the scope of the synchronization condition, the outputs will not change until the next clock transition. You need to be sure that this is the timing that your circuit requires.

A portion of a circuit that implements this VHDL specification is shown below.

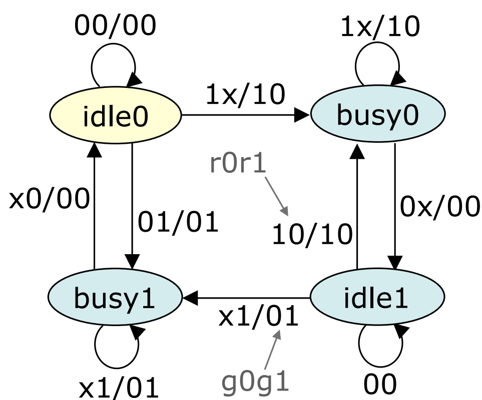


The circuit diagram brings out another issue, which is the question of

how the state signal is actually represented. The diagram uses symbolic state names to keep things simple, but the actual circuit generated by a circuit synthesizer must use some concrete representation of the state using binary signals. In this case, because there are four different states, we need at least two bits to represent the four different possibilities. So for example, we might use the bit pair 00 to represent the *idle0* state, 01 to represent the *idle1* state, 10 to represent the *busy0* state and 11 to represent the *busy1* state. Given these choices, it's straightforward to define logic equations for each of the two state bits, and the circuit synthesizer will do this when it implements the circuit.

There are other ways to represent the state information using binary signals. One common method is so-called one-hot encoding. In this method, a separate bit is used for each state. So for example, we might use the four bits 0001 to represent the *idle0* state, 0010 to represent the *idle1* state, 0100 to represent the *busy0* state and 1000 to represent the *busy1* state. At first glance, this seems like a poor choice, since it uses more bits (and more flip flops) than is strictly necessary. However, state machine implementations that use one-hot encoding generally require simpler next state logic than those that use the more compact binary encoding. This can offset of the cost of the extra flip flops. More importantly, it tends to produce circuits that have better performance, and this is often the more important consideration.

Before leaving the fair arbiter example, let's consider an alternate version in which the grant outputs are not delayed until the next clock tick, but respond immediately to input changes. This allows the clients to start using the shared resource sooner than they can in the original version. The figure below shows a state diagram that describes an arbiter that works in this way.



Note that the format of this state diagram is different from the format of the earlier one. Specifically, the output signals now appear on the arrows denoting the state transitions. This allows us to specify output values that are dependent not only on the current state, but also on the input signals. So for example, when in state *idle0*, the grant outputs are both low if the request inputs are both low, but *g0*=0 and *g1*=1 if *r0*=0 and *r1*=1.

Now, it's easy to get confused about state diagrams like this. When reading (or constructing) such a diagram, remember that the output values labeling an edge are a function of the "current state" and the input values. They are *not* a function of the "next state" (the one the arrow points to). Also, the output values respond immediately respond whenever the inputs change, not on the next clock edge.

We can also specify this state machine using a state table. The first half of the state table for this circuit is shown below.

current state	inputs r0 r1	outputs g0 g1	next state
idle0	00	00	idle0
	1x	10	busy0
	01	01	busy1
busy0	1x	10	busy0
	0x	00	idle1

To implement this version of the state machine, we need to change the output equations. In the VHDL, this means replacing the original assignments to the outputs with the following ones.

```
grant0 <= '1' when (state = idle0 and request0 = '1')
                or (state = idle1 and request0 > request1)
                or (state = busy0 and request0 = '1')
          else '0';
grant1 <= '1' when (state = idle1 and request1 = '1')
                or (state = idle0 and request1 > request0)
                or (state = busy1 and request1 = '1')
           else '0';
```

State machines that have asynchronous output signals (that is outputs that are functions of the state machine inputs, in addition to the current state) are called *Mealy-mode* state machines. State machines that have only synchronous outputs (outputs that are functions of the current state alone) are called *Moore-mode* state machines. Mealy mode state machines can offer a performance advantage in some situations, but they can make it more difficult to achieve a target clock rate. We'll discuss this issue in a later chapter when we study timing issues in digital circuits.

7.2 Garage Door Opener

In this section, we'll look at a slightly larger example of a state machine and we'll use it to explore some other issues that arise in the design of clocked sequential circuits. But first, let's review the process that we used to design the arbiter circuit and see how it illustrates a general design process that can be used for a wide range of different circuits.

At the beginning of the last section, we started with a text description of the aribter and a block diagram showing how it might be used to control access to a shared resource. This is generally a good way to start. Before you can proceed with the design of a circuit, you need to be able to describe exactly what it's supposed to do. Sometimes, you'll be given a precise specification, but often you'll be given only a partial description, and will need

to work out a lot of details for yourself. It's best to start by defining the input signals and output signals, and then describe how the outputs should respond to changes in the inputs. To make this description precise, it's generally a good idea to draw an interface timing diagram (as we did with the arbiter) that shows when outputs should respond to input changes (in particular, should they respond immediately, or only on a clock transition). In this phase of the design proces, you should ask lots of questions about how the circuit should behave in particular cases and make sure you know how to answer those questions.

Once you understand exactly what the circuit is supposed to do, you need to think about what states are needed to allow the circuit to do its job. The key here is to identify those things that the circuit must "remember" in order to work correctly. (For the arbiter, this included whether the resource was busy or idle. When it was busy, we also needed to remember which client was using it. When it was idle, we needed to remember which client should be served next, if both made simultaneous requests.) Once you understand the things that the circuit must be remember, you can proceed is to write down a list of states, and for each one, write a short description of what it means when the circuit is in that state.

At this point, you may be ready to make your first attempt at a state transition diagram. Start with a single state and think about what should happen next for each of the possible input signal combinations. This will allow you to fill in transitions to other states, and you can then proceed to work out the transitions leaving the other states. Do not expect to get this right the first time you try. It can take even experienced designers several iterations before they get a state diagram that completely describes the required circuit behavior. Sometimes you will find that you need additional states you had not anticipated, or that some of the states you had identified are not really necessary. There is some inevitable trial-and-error to this part of the design process, but if you think carefully about what the circuit is supposed to do and work through it systematically, you should be able to arrive at a state diagram that captures the required behavior.

Now once you have a state diagram (or state table, if you prefer that representation), the process of writing the VHDL is very straightforward.

You can use a process to describe the state transitions, as we did with the arbiter. This will typically involve a case statement on the state variable, with if-statements used to define specific transitions from each state. Outputs can be defined with signal assignments outside of the next-state process. Or, you may want to use a separate combinational process to define the output signals. The thing to understand is that once you have the state diagram, the process of writing the VHDL is largely mechanical. It's just a matter of writing code that implements the behavior described by the state diagram.

Let's proceed to consider another example of a state machine, and we'll use it to reinforce what we've said about the general design process. So, this next circuit is a controller for a garage door opener. It will control the motor used to raise or lower the door, in response to a signal from a remote control unit. It will stop the motor if it detects that the door is all the way open or all the way closed (the door is equipped with sensors that detect these conditions) and it will also reverse direction if the door is closing and it detects an obstruction in the path of the door (another sensor detects the presence of obstructions).

So, the circuit has four inputs and two outputs.

- The *openClose* input causes an open door to be closed, or a closed door to be opened. It's activated whenever a user presses the single button on the remote control. This button can also be used to stop a door that is in the process of opening or closing. Pressing the button again will cause the "paused" door to move in the opposite direction.

- The *atTop* input is high whenever the door is fully open.

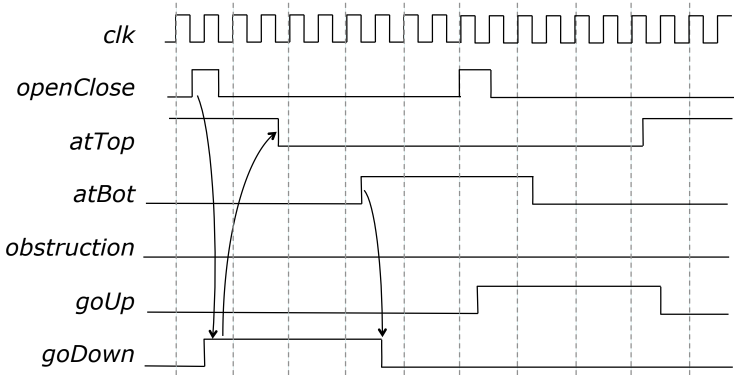
- The *atBot* input is high whenever the door is fully closed.

- The *obstruction* input is high whenever an obstruction has been detected in the door opening.

- The *goUp* output is used by the controller to open the door.

- The *goDown* output is used by the controller to close the door.

Here is a partial interface timing diagram showing a normal cycle of the door closing then opening. We can draw similar diagrams showing cases where the door pauses or reverses due to an obstruction.



In the diagram, the arrows indicate causal relationships. So for example, the *openClose* signal going high causes the *goDown* signal to go high. As an indirect consequence of this, the *atTop* input goes low (after some delay). Later, when the *atBot* input goes high, the *goDown* signal goes low. Of course, in a real implementation of a garage door opener, the door would be moving for 5-10 seconds, or many millions of clock ticks. This timing diagram ignores this to keep things simple.

Now that we have an understanding of the overall behavior of the garage door opener, it's time to identify an appropriate set of states. In this case, the state of the door itself suggests appropriate states for the controller.

- The controller is in the *opened state* when the door is fully open.
- The controller is in the *closed state* when the door is fully closed.
- The controller is in the *opening state* when we're in the process of opening the door.
- The controller is in the *closing state* when we're in the process of closing the door.

- The controller is in the *pauseUp state* when we've stopped the door while opening it.

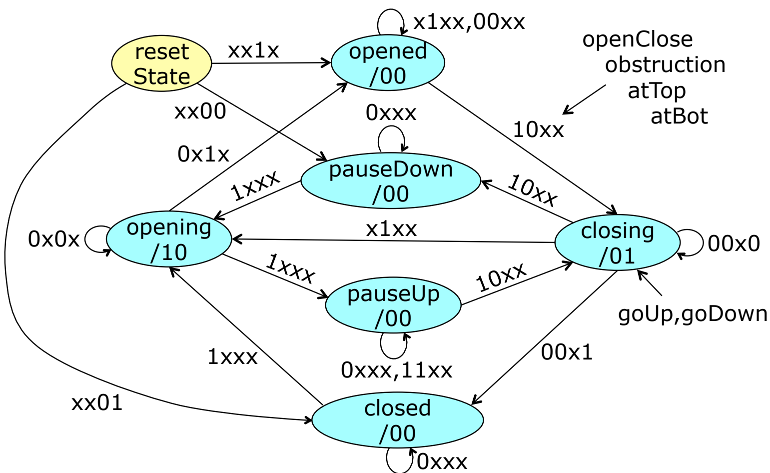
- The controller is in the *pauseDown state* when we've stopped the door while closing it.

The two pause states are used to remember which direction the door was moving at the time the user paused it by pressing the button on the remote control. This allows us to reverse the direction, the next time the user presses the button.

The controller should also have a reset state, which is entered when the power first comes on. Since a power outage might occur at any time, we can't really predict the position of the door when the power comes back on. So the reset state will need to use the sensors to determine if the door is fully open, fully closed or somewhere in between. The controller can then transition to an appropriate state.

Since the garage door opener is not a performance-critical application, it makes sense to use synchronous outputs. The *goUp* signal will be high when we're in the *opening* state. The *goDown* signal will be high when we're in the *closing* state. In all other states, both signals are low.

We're now ready to draw the state transition diagram.



Let's examine a few of the state transitions to confirm our understanding of how the controller should behave. In the *opened* state, we can assume that the *atTop* signal is high and that the *atBot* signal is low. So, we only need to pay attention to the *openClose* input and the *obstruction* input. If *obstruction* is low and *openClose* is high, we'll transition to the *closing* state.

The controller should stay in the *closing* state so long as the *openClose* input, the *obstruction* input and the *atBot* input all remain low. If *obstruction* goes high, we'll switch to the *opening* state, if *openClose* goes high, we'll switch to the *pauseDown* state, and if *atBot* goes high, we'll switch to the *closed* state. The labels on the transitions describe this behavior appropriately. Check the other transitions in the diagram and make sure you understand the reasons for them.

At this point, we can write the VHDL specification of the circuit. We'll start with the entity declaration.

```
entity opener is port (
    clk, reset: in std_logic;
    openClose: in std_logic;    -- signal to open or close door
    obstruction: in std_logic;  -- obstruction detected
    atTop: in std_logic         -- door at top (fully open)
    atBot: in std_logic;        -- door at bottom (closed)
    goUp: out std_logic;        -- raise door
    goDown: out std_logic);     -- lower door
end opener;
```

Next, we define the states.

```
architecture a1 of opener is
type stateType is (opened, closed, opening, closing,
                   pauseUp, pauseDown, resetState);
signal state: stateType;
```

Now, let's move onto the state transition process.

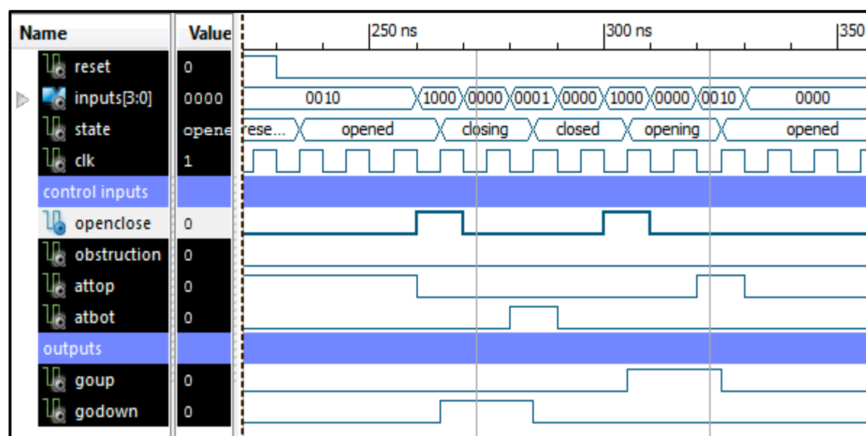
```
begin
    process (clk) begin
```

```
        if rising_edge(clk) then
            if reset = '1' then
                state <= resetState;
            else
                case state is
                when resetState =>
                        if atTop = '1' then state <= opened;
                     elsif atBot = '1' then state <= closed;
                     else state <= pauseDown;
                     end if;
                when opened =>
                    if openClose > obstruction then
                        state <= closing;
                    end if;
                when closing =>
                    if openClose = '0' and obstruction = 0
                       and atBot = '1 then
                        state <= closed;
                    elsif obstruction = '1 then
                        state <= opening;
                    elsif openClose = '1' then
                        state <= pauseDown;
                    end if;
                ...
                when others =>
                end case;
            end if;
        end if;
    end process;
```

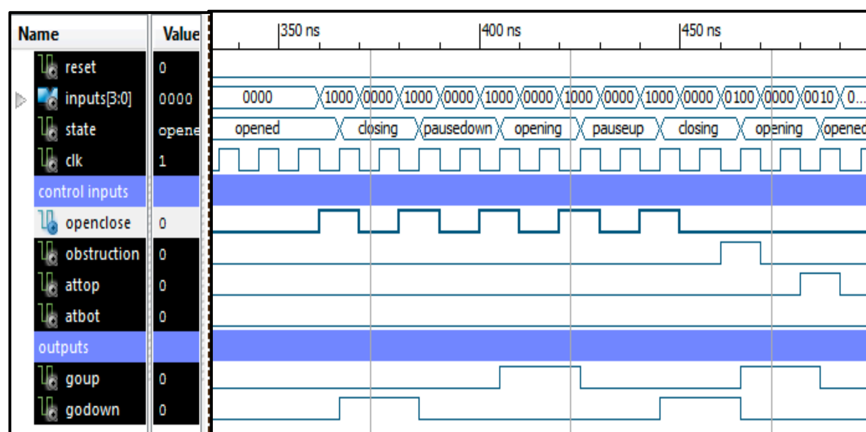
Finally, we have the output signal assignments.

```
    goUp   <= '1' when state = opening else '0';
    goDown <= '1' when state = closing else '0';
end a1;
```

At this point, we need to test the circuit to make sure it does behave as expected. Here is a portion of a simulation showing a normal close/open cycle. The four inputs are shown as a binary vector with the *openClose*



input shown first, followed by *obstruction*, *atTop* and *atBot*. Observe how the transitions occur in response to the input changes. The next segment of the simulation shows the circuit passing through the pause states, and shows its response to an obstruction.



Chapter 8

State Machines with Data

In the last chapter, we introduced the topic of sequential circuits, focusing on simple state machine controllers. In this chapter, we are going to look at more complex controllers that store other data in addition to the basic control state. This other data will be updated in response to input signals and may affect the way the controller behaves. We'll see that these extended state machines can be conviently specified using a slightly higher level version of the state transition diagrams discussed in the last chapter.

8.1 Pulse Counter

We'll start with a simple circuit that detects *pulses* on a data input and raises an event signal after a certain number of pulses have been detected. For the purposes of this circuit, we define a pulse as one or more consecutive clock ticks when the data input is high, preceded and followed by one or more clock ticks when the data input is low. The number of pulses that are to be counted before the event signal is raised is set when the signal is reset. Let's start by defining the interface signals.

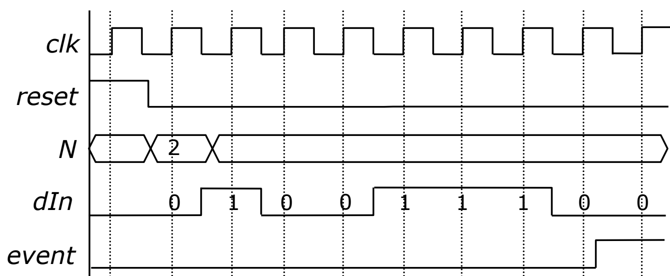
- The *reset* signal intializes the circuit. After reset goes low, the circuit starts detecting and counting pulses.

- The *dIn* input is the data input on which the circuit detects pulses.

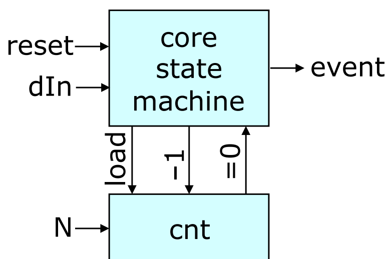
- The count input called N is a four bit signal that specifies the number of pulses to be counted before the *event* output goes high. It must be valid on the first clock tick after reset goes low.

- The *event* output goes high after N pulses have been detected and is cleared by the *reset* signal.

The interface timing diagram illustrates the operation of the circuit.



Note that while this example uses pulses that last for just one clock tick, in general pulses can last for an arbitrary number of clock ticks. We can implement the pulse counter using a circuit that combines a simple state machine with a *counter*, as shown in the block diagram below.



The block labeled *cnt* includes a register and additional circuitry that allows the register to be loaded and decremented (using the two control signals shown). It also has an output that signals when the value in the

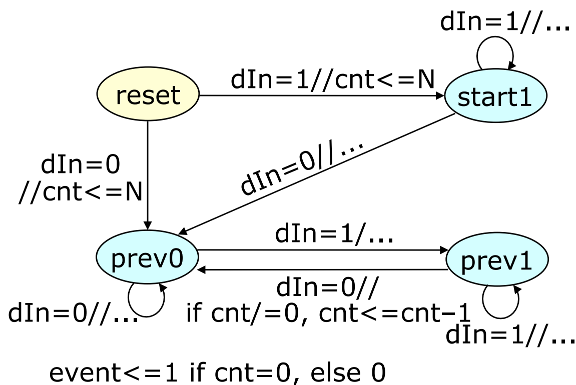
register is zero. This circuit will load the *cnt* register right after reset goes low and will then decrement it whenever it detects the end of a pulse (the data input goes from high to low).

We could write a VHDL specification that includes separate components for the core state machine and counter blocks, but it turns out to be simpler to view this entire circuit as a higher level state machine, in which the state of the core controller and the counter value determine the overall behavior. We can describe such a state machine using a generalized state transition diagram. As in the case of the simple state machines discussed in the last chapter, we need to start by determining what the controller must remember. Then we can define an appropriate set of states.

In the case of the pulse counter, the key thing we must remember is the state of the data input on the previous clock tick. This allows us to detect when a new pulse starts (the data input was low, but now it's high), and when a pulse ends. Whenever a pulse ends, we'll decrement the counter, if it's not already equal to zero. We also have to handle a special case when the data input starts out high. In this case, the first high-to-low transition on the data input does not correspond to the end of a pulse, so we need to detect this case and handle it separately. This suggests the following set of states.

- The *resetState* is entered when reset goes high.
- The *start1* state is entered if the data input is high when the reset input goes low.
- The *prev0* state means that on the previous clock tick the data input was low.
- The *prev1* state means that on the previous clock tick the data input was high, and at some time in the past, the data input was low.

We can use these states to define a generalized state diagram for the pulse counter.



In this diagram, the transitions are labeled not by simple inputs and outputs, but by *conditions* and *actions*, separated by a double forward slash. The conditions refer to both input values and the value of the counter. The actions specify how the counter value changes when state transitions occur. The ellipses are used for those transitions where no action is required other than the state change itself. So for example, if we're in the reset state and the reset input goes low, but the data input is high, the controller will go to the *start1* state and load the counter. It will remain in that state until the data input goes low, when it will transition to the *prev0* state. The counter is decremented on every transition from the *prev1* to the *prev0* state (if it's not already zero). The *event* output is specified at the bottom of the diagram, and is high whenever the counter value is zero. Take a few minutes to examine all the state transitions carefully, and convince yourself that this circuit will in fact implement the required behavior. Note that the reset input is not mentioned in the diagram, but it should be understood that the state machine goes to the reset state whenever the reset input is high, and remains there until the reset input goes low.

Now that we have the state diagram, it's straightforward to write down a VHDL specification for this state machine.

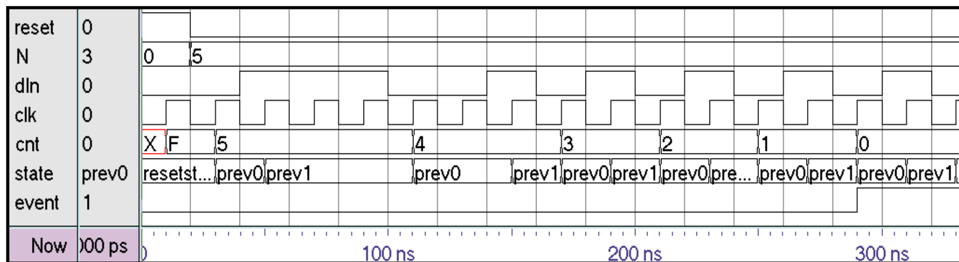
```
entity pulseCount is port(
   clk, reset: in std_logic;
   dIn: in std_logic;
```

```
   N: in std_logic_vector(3 downto 0);
   event: out std_logic);
end pulseCount;
architecture arch of pulseCount is
type stateType is (resetState, start1, prev0, prev1);
signal state: stateType;
signal cnt: std_logic_vector(3 downto 0);
begin
   process (clk) begin
      if rising_edge(clk) then
         if reset = '1' then
            state <= resetState; cnt <= x"F";
         else
            case state is
            when resetState =>
               cnt <= N;
               if dIn = '0' then state <= prev0;
               else state <= start1;
               end if;
            when start1 =>
               if dIn = '0' then state <= prev0; end if;
            when prev0 =>
                if dIn = '1' then state <= prev1; end if;
            when prev1 =>
                if dIn = '0' then
                   state <= prev0;
                   if cnt /= 0 then cnt <= cnt-1; end if;
                end if;
            when others =>
            end case;
         end if;
      end if;
   end process;
   event <= '1' when cnt = x"0" else '0';
```

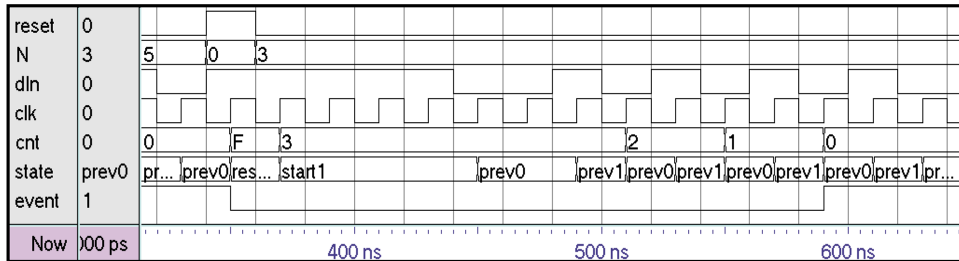
```
end arch;
```

Notice that like with the simple state machines in the previous chapter, we use a process to specify the state transitions. That same process is also used to update the counter. Notice how the conditions in the if-statements correspond directly to the conditions in the state diagram, while the actions in the state diagram are reflected in the the statement lists for those if-statements.

Now that we have a VHDL specification, we can go ahead and simulate it. In this portion of the simulation, the data input is zero initially, so we proceed



directly to the *prev0* state when reset goes low. Note that the counter value is initialized at this point and the counter is decremented on every high to low transition of the data input. We also see that the event output goes high when the counter reaches zero, and additional pulses are ignored at this point. The next part of the simulation shows the case where the data input is high when reset goes low. Notice that the counter is not decremented on



the first high-to-low transition of the data input, in this case.

Before going onto the next topic, let's review the process of designing a generalized state machine controller. As with the simple controllers, we start by writing down a careful description of what the circuit is supposed to do. This will generally include a list of inputs and outputs, and an interface timing diagram that illustrates the operation of the circuit.

The next step is to determine what the circuit must remember in order to implement the required behavior. There are two parts to this. There is the part that must be remembered by the core controller, and the part that is stored in data registers that the controller uses. In our pulse counter example, the *cnt* value was the only additional data required, but in more complex controllers there may be several registers and even large blocks of memory. It's a good idea to write down all the information that must be remembered, including a list of all the states with a brief description for each, plus a list of all the registers with a brief description of what they are used for.

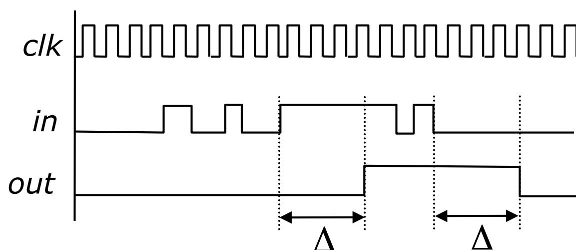
At this point in the process, we can start drawing a state transition diagram, with the transitions labeled by the conditions that trigger the transitions, and the actions that should be taken when those transitions occur. Once again, don't expect to get this right the first time. Check your diagram carefully and make sure that it accurately describes the required circuit behavior. Only after you've done that, should you go onto writing the VHDL. You will generally find that it is easy to write the VHDL once you have a complete state diagram in front of you. This part of the process is largely mechanical, since you are really just translating your state diagram into code.

At this point, all that's left is to test your circuit. Construct your test input carefully to make sure it covers all transitions in your state diagram, and check the simulation results to make sure that they are consistent with the desired behavior. When you find things that are not right, you'll need to figure out what you did wrong. Sometimes the problem is simply that you made a mistake when writing the VHDL code (perhaps you left out a transition, or you made a mistake when writing the code to implement one of the actions). These kinds of mistakes are the easiest to fix. Sometimes you'll find that you made a mistake when developing your state diagram. While many such mistakes are also easy to correct, others may require that

you go "back to the drawing board." Perhaps you forgot to include some state that you needed, or your understanding of how the stored data must be updated is incorrect. When this happens, try to use the results of the failed test to help you get a better understanding of how the circuit is supposed to work. Usually that will give you the insight needed to re-work your design and get back on track.

8.2 Debouncer

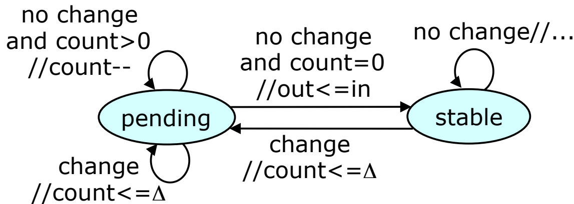
In an earlier chapter, we discussed how the mechanical buttons used in our prototype boards could vibrate when they're pressed or released. These vibrations cause the electrical signal from these buttons to "bounce" up and down a few times before stabilizing to a high or low value. This can cause circuits that take actions when a button is pressed or released to behave erratically. To cope with this, a special circuit called a *debouncer* is used to filter out the bounces, so that the internal signal sees just a single clean transition from low-to-high or high-to-low, for each button press or release. The behavior of a debouncer is illustrated by the following interface timing diagram.



The output signal is a delayed version of the input signal, that filters out changes that do not persist for at least Δ clock ticks, where Δ is a constant that depends on the clock period of the circuit and the typical oscillation time associated with the mechanical vibrations of the buttons. For example, if the clock period is 20 ns and the mechanical vibrations have a period of 1 ms, then Δ should have a value of at least 50,000. The timing diagram

shows a delay of just a few clock ticks, but in practice a much larger value would be used.

So, what is it that this circuit needs to remember. Well clearly it needs a counter, so that it can keep track of how much time has passed since the input last changed. It also needs to remember if it is waiting for the input to stabilize at a new value, or if the current input value has been stable for a long enough time. This suggests a state machine controller with two states *pending* and *stable*, which controls a counter that starts counting whenever the input changes. We'll also need a separate register to store the value of the previous input signal, so that we can easily detect a change in the input. In addition, we'll need a register to hold the output value, so that we can maintain the "old value" while waiting for the input to stabilize This leads to the state diagram shown below.



In this state diagram, we use the shorthand *change* or *no change* to indicate a clock period when the input either has changed or has not. If the old input value is stored in a register called *prevIn*, the *change* condition is simply $in \neq prevIn$. If we're in the stable state and the input changes, the counter is initialized to Δ and we transition to the pending state. While in the pending state, the counter is decremented so long as the input does not change and the counter is greater than zero. If the input does change while the controller is in the pending state, the counter is re-initialized. Once the input stops changing, the counter eventually counts down to zero, triggering a transition to the stable state. Note that on this transition, the input signal is propagated to the output. From the state diagram we can derive the following VHDL spec.

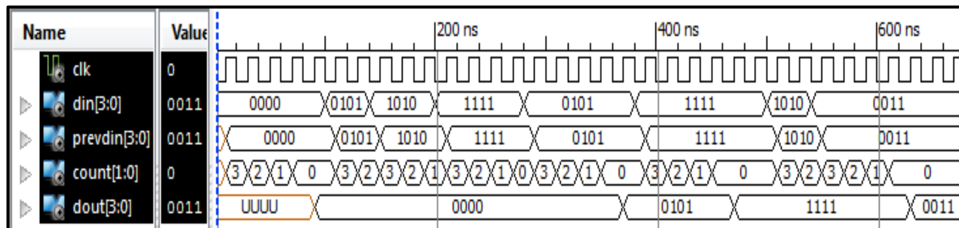
```
entity debouncer is
   generic (width: integer := 8);
   port(clk: in std_logic;
      din: in std_logic_vector(width-1 downto 0);
      dout: out std_logic_vector(width-1 downto 0));
end debouncer;

architecture a1 of debouncer is
signal prevDin: std_logic_vector(width-1 downto 0);
constant debounceBits: integer := 2 + operationMode*14;
signal count: std_logic_vector(debounceBits-1 downto 0);
begin
   process(clk) begin
      if rising_edge(clk) then
         prevDin <= din;
         if prevDin /= din then
            count <= (others => '1');
         elsif count /= (count'range => '0') then
            count <= count - 1;
         else dout <= din;
         end if;
      end if;
   end process;
end a1;
```

First notice that we've defined the debouncer to operate on a signal vector, rather than a single bit signal. Also, we've included a generic constant in the entity declaration. This is used to specify the number of bits in the input and output logic vectors. We've given it a default value of 8, but any time we instantiate the component, we can specify a different width, should we choose to do so.

Next, notice that the architecture defines two registers *prevDin* and *count* that store the previous input value, and the counter value used for filtering out bounces in the input. Also notice that because the data output *dout* is defined within the scope of the process' synchronization condition, a register

will be synthesized to store its value as well. Interestingly, there is no state register in this implementation. While we could have included one, in this case we chose to omit it because the value of the *count* register is all we really need to control the operation of the circuit. Whenever *count*=0 we're in the stable state, otherwise we're in the pending state.

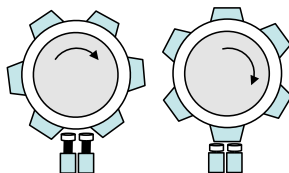
Finally, note that the delay implemented by this circuit is determined by the length of the *count* register. When the constant *operationMode*=1, it has 16 bits, and since the circuit is initialized to the "all-ones" value, it will count down 2^{16} clock ticks giving us a delay of about 1.3 ms when the clock period is 20 ns. If *operationMode*=0, the delay is just four clock ticks. This value is used for convenience when simulating the circuit. A simulation of a four bit version of the debouncer appears below.



Note how the first few changes to the data input do not affect the output. Only those changes that persist for long enough for the counter to reach zero propagate to the output.

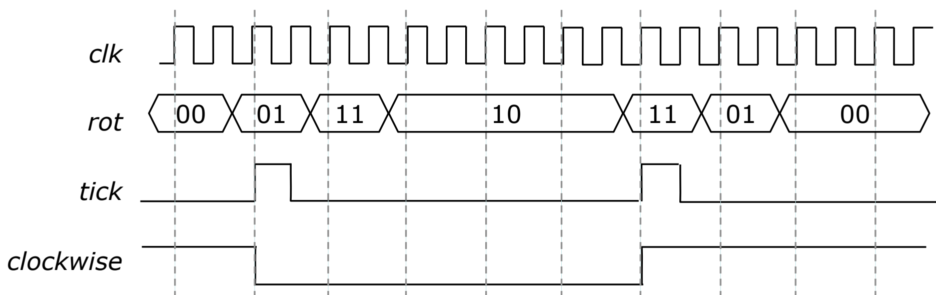
8.3 Knob Interface

In this section, we're going to look at one of the circuits used with the prototype boards we're using for testing our circuits. This circuit implements an interface to the prototype board's knob. When the knob is turned, there are two signals that are activated in a distinct sequence, depending on which direction the knob is turned. We can think of the knob as having a pair of buttons associated with it. As we turn the knob, raised "ridges" on the knob shaft cause first one button to be pressed down, then the other. As the knob continues to turn, the buttons are released.



So, if we turn the knob in one direction we observe the sequence 00, 01, 11, 10, 00, 01, 11, 10 and so forth, while if we turn it in the other direction, we observe the sequence 00, 10, 11, 01, 00, 10, 11, 01. (Note, the actual implementation of the knob does not really involve separate buttons, but it behaves in exactly in the same way that our conceptual knob does, and so this is a useful mental model to use to understand its operation.)

We can use the pattern of signal changes to detect whether the knob is turning in one direction or the other, and use this to increment or decrement a value stored in a register. So for example, we might increment a stored value whenever the two knob signals change from 00 to 01 and we might decrement the stored value whenever the signal values change from 10 to 11. The binary input module used by the calculator uses the knob in just this way. The essential behavior of the knob interface circuit is illustrated in the timing diagram below.



Here, the two knob signals are labeled *rot* (short for rotation) and the transition from 00 to 01 causes the *tick* output to go high for one clock tick and the *clockwise* output to go low and stay low. Transitions from 10 to 11 also cause tick to go high for one clock *tick* and cause *clockwise* to go high.

So what does this circuit need to remember? The main thing it needs to remember is the previous value of the *rot* inputs. By comparing these to the current values, it can make all the decisions required to generate the *tick* output and the *clockwise* output. While we could write a formal state diagram to describe this, it's not really necessary to do so in this case.

However, before we proceed to the VHDL let's add one more feature to the knob interface. Recall that in addition to turning the knob, we can press down on it. So, there is a third signal associated with the knob that works like an ordinary button. In the *binaryInMod* circuit, we used that button to adjust how we incremented the data value controlled by the knob. Initially, turning the knob would increment the value starting with bit 0, but by pressing down on the knob, we could make the *binaryInMod* increment/decrement the data value starting with bit 4 (or 8 or 12). To implement this feature, let's add another output to the knob interface called *delta*; this will be a 16 bit value equal to 2^0 or 2^4 or 2^8 or 2^{12}. Every press on the knob will cause *delta* to go from one of these values to the next. A circuit using the knob interface (like the *binaryInMod*) can then add or subtract *delta* to its stored value whenever the *tick* output of the knob interface is high.

We're now ready to proceed to the VHDL spec for the knob interface.

```
entity knobIntf is port(
   clk, reset: in std_logic;
   knob: in knobSigs;       -- 3 knob signals
   tick: out std_logic;      -- hi for each knob turn
   clockwise: out std_logic;  -- hi for clockwise rotation
   delta: out std_logic_vector(15 downto 0); -- add/sub amt
end knobIntf;

architecture a1 of knobIntf is

signal dbKnob: knobSigs;
signal rot, prevRot: std_logic_vector(1 downto 0);
signal btn, prevBtn: std_logic;
signal diff: std_logic_vector(15 downto 0);
begin
```

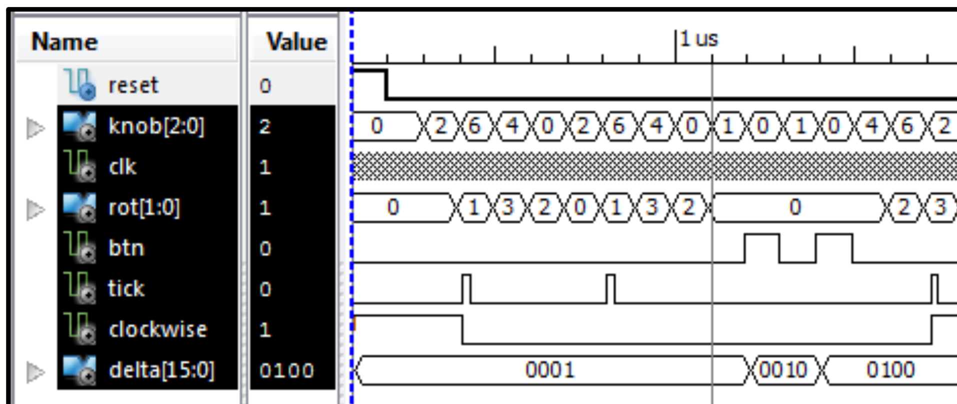
```
   db: debouncer generic map(width => 3)
                 port map(clk, knob, dbKnob);
   rot <= dbKnob(2 downto 1);
   btn <= dbKnob(0);
   delta <= diff;
   process(clk) begin
      if rising_edge(clk) then
         prevRot <= rot;
         prevBtn <= btn;
         tick <= '0';
         if reset = '1' then
            diff <= x"0001";
            clockwise <= '1';
         else
            if prevRot = "00" and rot = "01" then
               tick <= '1'; clockwise <= '0';
            end if;
            if prevRot = "10" and rot = "11" then
               tick <= '1'; clockwise <= '1';
            end if;
            if btn > prevBtn then
               diff <= diff(11 downto 0) &
                     diff(15 downto 12);
            end if;
         end if;
      end if;
   end process;
end a1;
```

The button signals are specified as a three bit value, with the top two bits representing the rotational signals, while bit 0 is the signal that is activated when you press down on the button. Note that all three signals are debounced and then the debounced signals are assigned the local signal names *rot* and *btn*. The signals *prevRot* and *prevBtn* are used to hold the values of *rot* and *btn* from the previous clock tick, so by comparing these to *rot* and

btn, we can detect signal changes and decide how to react.

Notice that the *tick* output is assigned a default value of 0, and is then assigned a value of 1, whenever the rotation signal changes from 00 to 01 or 10 to 11. The clockwise signal changes to 1 whenever the rotation signals change from 00 to 01, and it changes to 0 whenever the rotation signals change from 10 to 11. The internal *diff* signal is initialized to 1 and is rotated to the left by four bit positions whenever a knob press is detected.

A portion of a simulation of the knob interface is shown below.



8.4 Two Speed Garage Door Opener

We'll finish this chapter by extending the garage door opener we discussed in the previous chapter. This new version controls a motor with two operating speeds. When the door first starts moving, the controller sets it to the slower speed, then switches to the faster speed after a couple seconds. Also, when the door is closing, the controller switches to the slower speed when the door is getting close to the bottom of the range.

To implement this new functionality, the controller needs to maintain some additional information. First, it needs to know how much time has passed since the door started moving, so that it can switch to the higher

speed when enough time has passed. Second, it needs to track the position of the door, so that it can decide when to slow it down, as it's closing.

To make things interesting let's say that the controller should be able to support a variety of door heights and should adjust its operation for the height of each specific door it is used with. Also, we'll assume that the speed at which the door travels is known, but we also want to allow for some variation in the speed; say plus or minus 10 percent of the nominal values.

So let's look at how we can extend the VHDL for the controller to satisfy these various requirements. First, here are some new constant, type and signal definitions that we'll use in the architecture.

```
begin entity ... end entity;
architecture a1 of opener is
...
subtype timeVal is unsigned(15 downto 0);   -- up to 65K ms
subtype positionType is signed(9 downto 0); -- up to 511 cm

constant slowPeriod: timeVal := to_unsigned(100,16); -- ms/cm
constant fastPeriod: timeVal := to_unsigned(40,16);  -- ms/cm
constant ticksPerMs: unsigned(15 downto 0)
                := to_unsigned(50000,16);
constant speedUpTime: timeVal :=  to_unsigned(2000,16);
constant slowDownPos: positionType := to_signed(25,10);

signal speed: std_logic;      -- 0 for slow, 1 for fast
signal ticks: unsigned(15 downto 0); -- count clock ticks
signal timeInMotion: timeVal;
signal timeSinceUpdate: timeVal;
signal position: positionType; -- door position (cm from bot)
```

Notice that the `positionType` has been declared as a signed value. We will adapt to the actual door height, by setting the position value to 0 when we detect that the door is at the bottom of its range (based on the sensor). However, if the door does not start in the closed position, we have no way of knowing its initial position. By initializing the position to zero and allowing

it to go negative, we can ensure that the door will be going at the slow speed the first time it comes to a close.

The `slowPeriod` constant specifies the number of milliseconds required for the door to move one centimeter, when going at the slow speed. The `fastPeriod` constant is defined similarly. The `ticksPerMs` constant specfies the number of clock ticks in one millisecond. If the controller uses a 50 MHz clock, this is 50,000. The `speedUpTime` constant specifies how long to operate at the slow speed before speeding up (in ms). The `slowDownPos` constant specifies the position of the door that triggers a switch from fast to slow.

The `ticks` signal is used to keep track of time in milliseconds. The `timeInMotion` signal is the amount of time that has passed since the door last started moving. The `timeSinceUpdate` signal is the amount of time that has passed since the door's position was last incremented or decremented.

Now, let's take a look at the body of the architecture.

```
begin
   process (clk) begin
     if rising_edge(clk) then
       if reset = '1' then
         state <= resetState; speed <= '0';
         ticks <= (others => '0');
         timeInMotion <= (others => '0');
         timeSinceUpdate <= (others => '0');
         position <= (others => '0');
       elsif state = resetState then
            if atTop = '1' then state <= opened;
         elsif atBot = '1' then state <= closed;
         else state <= pauseDown;
         end if;
       else
         -- update time signals
         if ticks = ticksPerMs then
            timeInMotion <= timeInMotion + 1;
            timeSinceUpdate <= timeSinceUpdate + 1;
            ticks <= (others => '0');
```

```
      else
         ticks <= ticks + 1;
      end if;
```

This first part just shows the initialization logic and the advancing of the signals `timeInMotion` and `timeSinceUpdate`. The next section shows how the stored information is used to control the door while it is closing.

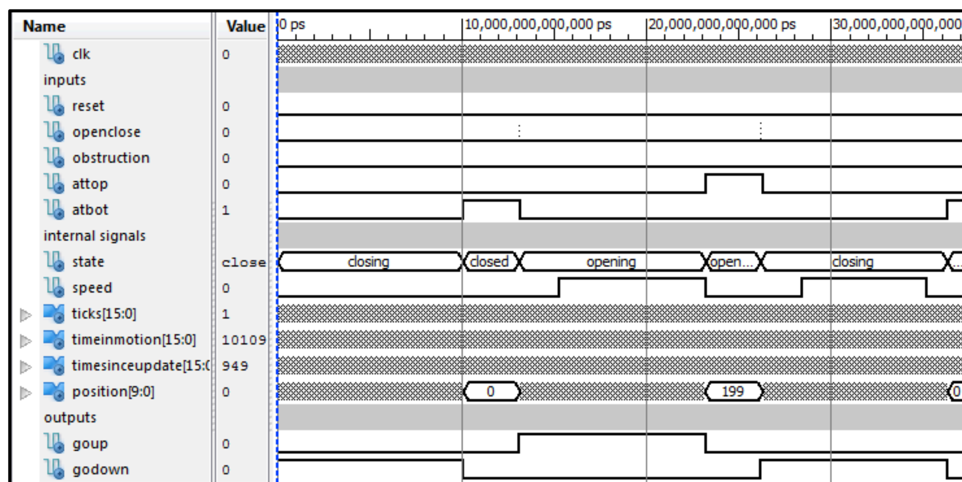
```
-- state transitions
case state is
when opened =>
   speed <= '0';
   if openClose = '1' and obstruction = '0' then
      state <= closing;
      timeInMotion <= (others => '0');
      timeSinceUpdate <= (others => '0');
   end if;
when closing =>
   if obstruction = '1' then
      state <= opening; speed <= '0';
      timeInMotion <= (others=>'0');
      timeSinceUpdate <= (others=>'0');
   elsif openClose = '1' then
      state <= pauseDown;
   elsif atBot='1' then
      state <= closed; position <= (others => '0');
   elsif position <= slowDownPos then
      speed <= '0';
   elsif timeInMotion = speedUpTime then
      speed <= '1';
   end if;
   -- track position of door
   if (speed='0' and timeSinceUpdate=slowPeriod) or
      (speed='1' and timeSinceUpdate=fastPeriod) then
      position<=position-1;
```

```
                timeSinceUpdate<=(others=>'0');
            end if;
        ...
```

When the door starts to close, the `timeInMotion` and `timeSinceUpdate` signals are cleared. While it's closing, an obstruction causes it to reverse direction and switch to the slow speed. The `timeInMotion` and `timeSinceUpdate` signals are also cleared in this case. Note that the speed adjustments ensure that the door will never speedup after it goes below the level specified by *slowDownPos*.

The last few lines of this section decrement the position of the door when it's time to do so. Note that when the door changes direction, there is some error introduced into the door position tracking. However, this should be limited to about 1 cm, which is too small to have any serious impact on the operation of the door.

We'll omit the remaining code for the opener, but it's a worthwhile exercise to fill in the remaining parts. We'll close with a portion of a simulation.



In this simulation, the door starts in the fully open position, then closes. Since the door position was initialized to zero, the position becomes negative

as the door closes. This ensures that the door height signal remains below the *slowDownPos* value, as the door closes the first time, preventing it from speeding up. Once the door closes, the position signal is set to zero, allowing the door position to be tracked as it opens. Notice that the door does speed up after the first two seconds in motion. Also, notice that when it closes the second time, it slows down before becoming fully closed.

Part II

Second Half

Chapter 9

Still More VHDL

In this chapter, we'll complete our discussion of the VHDL lanaguage. While we cover all the language features that are commonly used to define circuits, our treatment of the language is not comprehensive. Once you have mastered the essential elements of the language, you'll be well-prepared to continue your study of the language using other texts.

9.1 Making Circuit Specifications More Generic

In an earlier chapter, we gave an example using a symbolic constant `wordSize` to represent the number of bits in a signal vector. Using symbolic constants in this way helps make circuit specifications less dependent on specific signal lengths. If used consistently, it makes it possible to change the lengths of a set of related signal vectors, if we should decide that our original choice was not ideal. VHDL provides a number of other features that can be used to make circuit specifications more generic. By using these features, we can design circuit components that are more flexible and can be more easily re-used in other contexts or adapted to new requirements.

We'll start by considering *signal attributes*. Let signals `x`, `y` and `z` be defined as follows.

```
constant longSize: integer := 16;
```

```
constant shortSize: integer := 8;
signal x: std_logic_vector(longSize-1 downto 0);
signal y: std_Logic_vector(shortSize-1 downto 0);
signal z: std_Logic_vector(1 to shortSize);
```

Given these definitions, `x'left` is an *attribute* that denotes the leftmost index in the index range for `x`, so in this context `x'left=15`. Similarly `x'right=0`. In general, the ' symbol indicates that what comes next is the name of some signal attribute. Here are some more examples: `z'low=1`, `z'high=8` and `x'length=16`. Using these attributes, we can refer to the sign bit of `x` as `x'high`. Or, we can refer to the high order four bits of `y` as `y(y'high downto y'high-3)`. There is also a `range` attribute which refers to the entire index range of a signal. Using this, we can write things like

```
x <= (y'range => '1', others =>'0');
```

The resulting value of `x` is `x"00ff")`.

Next, we describe another way to make VHDL specs more generic using subtypes. Earlier, we saw an example of subtypes in the context of enumerations. However, subtypes can also be used to assign a name to a `std_logic_vector` or a particular length. Here's an example.

```
subtype word is std_logic_vector(15 downto 0);
signal x, y: word;
```

Here, `word` is a new type, which can be used in signal declarations like any other type. Signal declarations using the new type provide more information to a reader, as well-chosen type names can provide an indication of how the signal is used.

While we're on the subject of types and subtypes, let's take the opportunity to discuss two other types that are provided by the IEEE library, the `signed` and `unsigned` types. These types are very similar to the `std_logic_vector`. The main difference is that they are intended to be used in contexts where the signal represents an integer value. For `signed` signals the signal values are interpreted as being in 2s-complement form, so if `x` is a `signed` signal, the expression in the if-statement

```
if x < 0 then ... end if;
```

will evaluate to true if the sign bit of x is equal to 1. On the other hand, if x is a std_logic_vector or an unsigned signal, the expression will never be true.

Since VHDL is a strongly-typed language, it is sometimes necessary to convert among these types. To convert a signed or unsigned signal x to a std_logic_vector, use std_logic_vector(x). Similarly, to convert a std_logic_vector z to signed use signed(z). To convert z to unsigned use unsigned(z).

This brings us to the subject of integers in VHDL. An integer in VHDL is defined in much the same way as in ordinary programming languages. In particular, VHDL integers take on values that can range from -2^{31} up to $2^{31} - 1$. They are not defined as having a certain number of bits, although they can naturally be represented using 32 bit binary numbers. On the other hand, the std_logic_vector and the signed and unsigned types are defined as one-dimensonal arrays of std_logic values. While they can be manipulated much like normal integers (they can be added together, for example), they are fundamentally different from the integer data type. As a rule, circuits are constructed using signals of type std_logic_vector, signed or unsigned, not integers. While it is possible to define integer signals in a circuit specification, such signals are often not well-supported by CAD tools, so it is best to avoid them.

However, there are some situations in circuit specifications where you do need to deal with integers. In particular, loop indexes and array indexes are defined as having an integer type and this leads to situations where you may need to convert between an integer and one of the other types. To convert a signed or unsigned signal x to an integer, use to_integer(x). To convert an integer z to signed, use to_signed(z,LENGTH) and to convert z to unsigned, use to_unsigned(z,LENGTH). Here, LENGTH is a constant that specifies the number of bits in the new signal. Unfortunately, there are no direct conversions between std_logic_vector and integers, so you need to convert in two steps, through an *unsigned* signal.

Type conversions can be tedious to type, so designers often define more concise versions of their own. We will follow that practice here. We will

use int(x) to convert x to an integer and slv(z,LENGTH) to convert z to a std_logic_vector.

We will complete this section with a discussion of *parameterized components*. Here's an example.

```
entity negate is
    generic ( wordSize => integer := 8 );
    port (
        x: in signed(wordSize-1 downto 0);
        neg_x: out signed(wordSize-1 downto 0));
end negate;
```

The generic clause defines a constant *wordSize* with a default value of 8. WordSize can be used anywhere within the entity or architecture. To instantiate a parameterized component, we use a component instantiation statement that includes a generic map.

```
neg1: negate generic map(wordSize => 16) port map(...);
```

The instantiated component operates on 16 bit words, rather than the default 8. Note, that a circuit may instantiate several negate components, all using different values for the wordSize parameter. The generic clause in the entity declaration may contain several parameter definitions, separated by semi-colons. Similarly, the generic map may specify the values of several parameters.

9.2 Arrays and Records

In this section, we look at how we can define more complex data types. We'll start with arrays. As noted earlier, the std_logic_vector type is actually defined as an array of std_logic elements. Here's an example of an array whose elements have type unsigned.

```
subtype register is unsigned(15 downto 0);
type regsterFileType is array(0 to 7) of register;
signal reg: regsterFileType;
```

VHDL does not allow us to use the **array** directly in a signal declaration, but we can use it to define a new type, registerFileType in this example. Given these definitions, we can write things like this.

```
reg(2) <= reg(1) + reg(4);
reg(3 downto 0) <= reg(7 downto 4);
reg(3)(5) <= '1';
reg(int(x)) <= reg(int(y)) -- int() converts to integer
```

The first line adds the values from two registers and places the sum in a third. The second line copies the values in four registers to four others. The third line sets a single bit in reg(3) and the last one shows how registers can be selected using the values of signals. Before signals can be used to index an array, their values must be converted to integer type.

Now arrays can always be implemented using flip flops, but for large arrays this can become excessively expensive. In some situations, arrays can be implemented using *memory components*, and circuit synthesizers will attempt to do this when it makes sense to do so. In order to synthesize a memory, the use of the array within the VHDL spec must be consistent with the constraints on the memory components that are available to the synthesizer. For example, a basic memory component may have a single address input, a single data input and a single data output. Consequently, it's not possible for two different parts of a circuit to read a value from the memory at the same time. In order for the synthesizer to implement an array using such a memory component, it must be able to determine that there are never any concurrent reads from the memory. Here is an example of a lookup table component that can be synthesized as a memory.

```
entity lookupTable is port(
    clk: in std_logic;
    modTable: in std_logic;
    rowNum: in unsigned(3 downto 0);
    inVal: in std_logic_vector(15 downto 0);
    outVal: out std_logic_vector(15 downto 0));
end lookupTable;
```

```
architecture a1 of lookupTable is
subtype rowType is std_logic_vector(15 downto 0);
type tableType is array(0 to 15) of rowType;
signal table: tableType;
begin
    process(clk) begin
        if rising_edge(clk) then
            outVal <= table(int(rowNum));
            if modTable = '1' then
               table(int(rowNum)) <= inVal;
            end if;
        end if;
    end process;
end a1;
```

In this specification, the table array is accessed in a way that is consistent with the constraints on a typical memory component, allowing a synthesizer to implement the array using a memory.

In some situations, an array is defined to hold a collection of related constant values. Here's an example.

```
subtype asciiChar is std_logic_vector(7 downto 0)
type hex2AsciiMap is array(0 to 15) of asciiChar;
constant hex2Ascii: hex2AsciiMap := (
    x"30", x"31", x"32", x"33",x"34", -- 0-4
    x"35", x"36", x"37",x"38", x"39", -- 5-9
    x"61", x"62",x"63", x"64", x"65", x"66" -- a-f
);
```

This array stores the ASCII character codes for the hex digits 0-9, a-f. So for example, `hex2Ascii(12)=x"63"` which is the ASCII character code for the letter 'c'. Since this has been declared as a constant array, a circuit synthesizer may be able to implement it as a *Read-Only Memory* (ROM), which is generally less expensive than a general memory component.

Next, we turn to the subject of VHDL *records*. While arrays allow us to define collections of items that all have the same type, records allow us to

define collections of dissimilar items. Here's an example of a table whose rows are defined using a record type.

```
type rowType is record
    valid: std_logic;
    key, value: word
end record rowType;
type tableTyp is array(1 to 100) of rowType;
signal table: tableType
```

Given these definitions, we can write things like this.

```
table(2).key <= table(1).value;
if table(0).valid = 1 then
    table(5) <= (1, x"abcd", x"0123");
end if;
```

Notice that the second assignment includes a record specification on the right. This lists the values of all fields of the record in the order they were declared.

9.3 Using Assertions to Detect Bugs

The `assert` statement is a powerful tool for debugging circuit specifications. `Assert` statements can be placed anywhere within an architecture body and are checked during simulation. Here's an example.

```
assert (x'range => '0') <= x and x < n;
```

The simulator will print a message if the signal `x` takes on values outside the range $[0, n-1]$. We can also include a *report clause* and/or a *severity clause*.

```
assert (x'range => '0') <= x and x < n
       report "out of range" severity failure;
```

This will cause the simulator to terminate when it encounters a failure of this assertion. Other possible values are "note", "warning" and "error".

Generally, the simulator will keep running if a failed assertion has a severity of "note" or "warning".

Assertions add no cost to a synthesized circuit, since they are checked only during simulation. It's smart to use them extensively, as it takes very little effort to write them and they can save you many hours of debugging time.

9.4 VHDL Variables

In addition to signals, VHDL supports a similar but different construct called a "variable". Unlike signals, variables do not directly correspond to wires or any other physical element of a synthesized circuit, although they can affect how a circuit is synthesized. The semantics of variable assignment are different from the semantics of signal assignment, so VHDL uses a separate symbol ':=' to represent assignment to a variable. To illustrate the difference, here is a small example of a process that uses a variable to define the values of several signals.

```
process (x)
variable u: unsigned(3 downto 0);
begin
    u := x;
    a <= u;
    u := u + "0001";
    b <= u;
    u := u + "0001";
    c <= u;
end process;
```

Note that the variable `u` is declared as part of the process. Variables may not be declared as part of an architecture. Now the way to think of `u` in this context is as a shorthand for the expressions that appears on the right side of the variable assignments. Each new variable assignment, "re-defines" `u` to represent a different expression. Whenever a variable is used in an

expression, we simply substitute the expression that the variable represents. So, the assignments above are equivalent to the signal assignments

```
a <= x;
b <= x + "0001";
c <= (x + "0001") + "0001";
```

So, if the signal `x` has a value of `"0001"`, then the three outputs will have values of `"0001"`, `"0010"` and `"0011"`. This is what we would expect if we were using an ordinary programming language, in which each variable assignment copied a new value to a memory location. However in the VHDL context, there is no underlying memory that holds variable values. Indeed, there is nothing at all in the synthesized circuit that corresponds to the variable. Still, VHDL variables do affect the values of the signals in a way that is similar to the way variables in ordinary programming languages behave.

Now, most common uses of variable assignment can be understood purely as defining a shorthand for an expression. However, there are some situations that are not as straightforward. Here is an example of such a situation.

```
process (clk)
variable u: unsigned(3 downto 0);
begin
    if rising_edge(clk) then
        if x = "0000" then
            u := "0000";
        end if;
        a <= u;
        u := u + "0001";
        b <= u;
        u := u + "0001";
        c <= u;
    end if;
end process;
```

Notice that the variable `u` is only assigned a value when the signal `x="0000"`. So, it's not entirely clear what value `u` should have in the assignments that

use it. Because u is not defined under all possible conditions, VHDL interprets this as meaning that u retains its value from one clock tick to the next. This requires that the synthesized circuit include a register that can store the value of u. So the process above is equivialent to the process shown below, where u_sig is a new signal.

```
process (clk)
variable u: unsigned(3 downto 0);
begin
    if rising_edge(clk) then
        u := u_sig;
        if x = "0000" then
            u := "0000";
        end if;
        a <= u;
        u := u + "0001";
        b <= u;
        u := u + "0001";
        c <= u;
        u_sig <= u;
    end if;
end process;
```

The assignment to u_sig at the end of the process preserves the variable's value until the next clock tick, where it is used to re-initialize the variable. We can re-write the process without using variables as follows.

```
process (clk)
begin
    if rising_edge(clk) then
        if x = "0000" then
            u_sig <= "0000";
        else
            u_sig <= u_sig + "0010";
        end if;
        a <= u_sig;
```

```
            b <= u_sig + "0001";
            c <= u_sig + "0010";
        end if;
    end process;
```

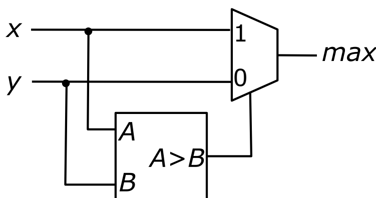
Because the semantics of variable assignment in VHDL can be confusing, it's generally a good idea to use variables sparingly in circuit specifications. Your code will usually be easier to understand and get right, if you use signals whenever you can. There are occasions when a circuit specification can be made simpler by using variables, but in most situations they are are not worth the trouble.

9.5 Functions and Procedures

We have already seen how VHDL components, defined by an entity and architecture, can be used to structure large circuit specifications into smaller parts. VHDL also provides other mechanisms to structure circuit specifications: *functions* and *procedures*. A VHDL function takes one or more arguments and returns a single value. Here's an example of a function that defines a circuit whose output is the maximum of two input values.

```
    function max(x,y: word) return word is
    begin
        if x > y then return x; else return y; end if;
    end function max;
```

When this function is used in a circuit specification, the synthesizer creates a sub-circuit consisting of a multiplexor controlled by a comparison circuit.



Within a function, its arguments are considered to be constants, so they cannot appear on the left side of an assignment.

Given the above function definition, a circuit specification that includes the assignment

```
largest <= max(a,max(b,c));
```

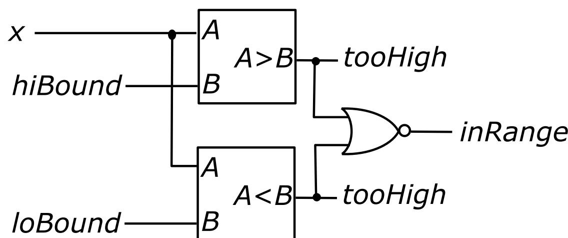
would include two copies of the `max` circuit connected in series.

The VHDL *procedure* can be used to define common circuit elements that have more than one output. Here is an example of a circuit with three outputs that determines if an input value falls within some pre-defined range.

```
procedure inRange(x: in word;
                  signal inRange: out std_logic;
                  signal tooHigh: out std_logic;
                  signal tooLow: out std_logic) is
constant loBound: word := x"0005";
constant hiBound: word := x"0060";
begin
    tooLow <= '0'; inRange <= '0'; tooHigh <= '0';
    if x < loBound then tooLow <= '1';
    elsif x <= hiBound then inRange <= '1';
    else tooHigh <= '1'; end if;
end procedure;
```

Note, the formal parameters to a procedure are designated as either inputs or outputs. Here, the output parameters have been declared to be signals, so that they can be associated with signals in the context where the procedure is used. By default, output parameters are considered variables. In this case, assignments to them must use the variable assignment operator and they must be associated with variables in the context where the procedure is used.

The circuit shown below can be used to implement the procedure.



Local variables may be declared in functions or procedures, but local signals may not be. Also, synchronization conditions may not be used within functions or procedures, although functions and procedures may be used within the scope of a synchronization condition defined as part of a process.

Functions and procedures may be declared either as part of an architecture (before the `begin` that starts the body of the architecture) or as part of a process. In the first case, the function/procedure can be used anywhere within the architecture, in the second case it can be used only within the process. To make a process/procedure available in mutiple circuit components, it must be declared within a package.

An example of a package containing a function declaration appears below.

```
package commonDefs is
    function max(x,y: word) return word;
end package commonDefs;

package body commonDefs is
    function max(x,y: word) return word is
    begin
        if x > y then return x; else return y; end if;
    end function max;
end package body commonDefs;
```

The package declaration has two parts. The *package header* includes constant declarations, type declarations and function/procedure declarations. The *package body* includes the actual definitions of functions and procedures, previously declared in the package header.

Chapter 10

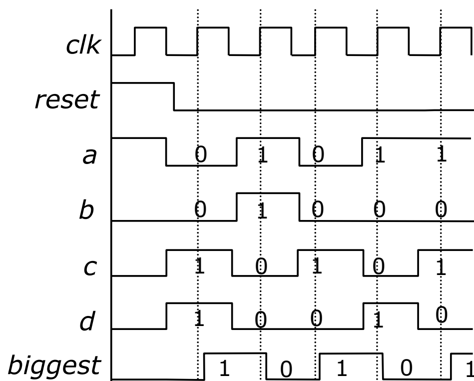
Design Studies

This chapter is the first of three that present the design of a variety of different digital circuits. The objective of these design studies is to help you build experience and develop your own design skills.

10.1 Four-way Max Finder

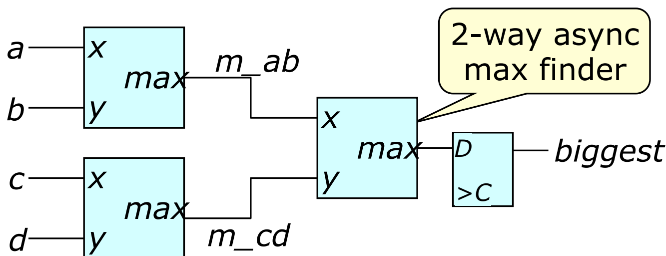
We'll start with a circuit that finds the maximum of four input values that are presented one bit at a time, starting with the high-order bit. Circuits that work this way are referred to as *bit serial* circuits and are useful in contexts where data is communicated bit-by-bit in order to minimize the number of wires used to carry the data. It also provides a nice example of building large components from smaller ones.

The circuit has a reset input and four data inputs. After the reset goes low, new data bits appear on every clock tick oneach of the four data inputs. The most-significant-bits arrive first. The circuit's output signal is the largest of the four input values. So for example, if the input values are "01001", "00100", "01100" and "00111", the output value would be "01100". The interface timing diagram shown below illustrates the operation of the circuit.



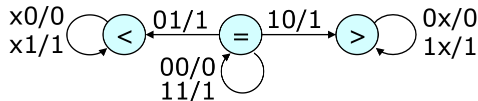
Notice that in the first clock tick following reset, we already know that neither of the first two inputs can have the maximum value, since they both have a most-significant-bit equal to 0, while the other two inputs have a most-significant-bit equal to 1. By the third clock tick, it's clear that input *c* has the largest value. Note that the output *biggest* is synchronous with the clock. Also, observe the bits of *biggest* can be output before we know which of the inputs has the largest value. At each point in time, one or more of the inputs is a "contender" for the largest value. All of the contenders have the same sequence of high-order bits, so no matter which of them turns out to have the largest value, those high-order bits will be the same.

Now, we could design a state machine to directly find the largest of the four input values, but we're going to take a different approach, by breaking the problem into three instances of a smaller problem. This is illustrated in the block diagram shown below.



This circuit contains three copies of a 2-way max-finder circuit. These each propagate the larger of two values, presented bit-serially. To avoid adding extra delay to the output of the 4-way circuit, these are designed as Mealy-mode state machines. That is, they have an asynchronous output that responds directly to changes in the input signal. To provide a synchronous output for the overall circuit, we add a single D flip-flop after the last max-finder.

So now, all we have to do is design a 2-way max finder. A simplified state machine for such a circuit is shown below. The reset input, puts the circuit



in the state labeled '='. After reset goes low, the circuit starts comparing its two data inputs and it remains in the equal state, so long as the data inputs are equal to each other. When the two inputs differ, the circuit switches to either than less-than state, or the greater-than state, depending on which of the two inputs has the larger value. Once this transition has occurred, the circuit stays in that state and simply propagates the bits from the input that has the larger value.

Now at this point, we could write down the VHDL for a 2-way max finder component, then write the VHDL for a 4-way component that instantiates three copies of the 4-way component and connects them together. However, we're going to look at an alternative approach to implementing the larger component, using state registers and a pair of functions to define the individual 2-way max finders. We'll start with the entity declaration and the definition of the state type.

```
entity max4 is port (
    clk, reset : in std_logic;
    a,b,c,d : in std_logic;
    biggest : out std_logic);
end max4;

architecture arch of max4 is
```

```
-- states for 2-way max-finders
type stateType is (eq, lt, gt);
```

Now, let's take a look at the next state function for the 2-way max finders.

```
-- next state function for 2-way max-finders
function nextState(state:stateType; x,y:std_logic)
return stateType is begin
    if    state = eq and x > y then return gt;
    elsif state = eq and x < y then return lt;
    else return state;
    end if;
end function nextState;
```

Note that the state values returned exactly reflect the transitions in the state diagram. Here's the output function.

```
-- output function for 2-way max-finders
function maxBit(state: stateType; x,y: std_logic)
return std_logic is begin
    if state = gt or (state = eq and x > y) then return x;
    else return y;
    end if;
end function maxBit;
```

This returns the bit from the input that has the largest value, based on the given state. Now, let's look at how these functions can be used to define multiple state machines.

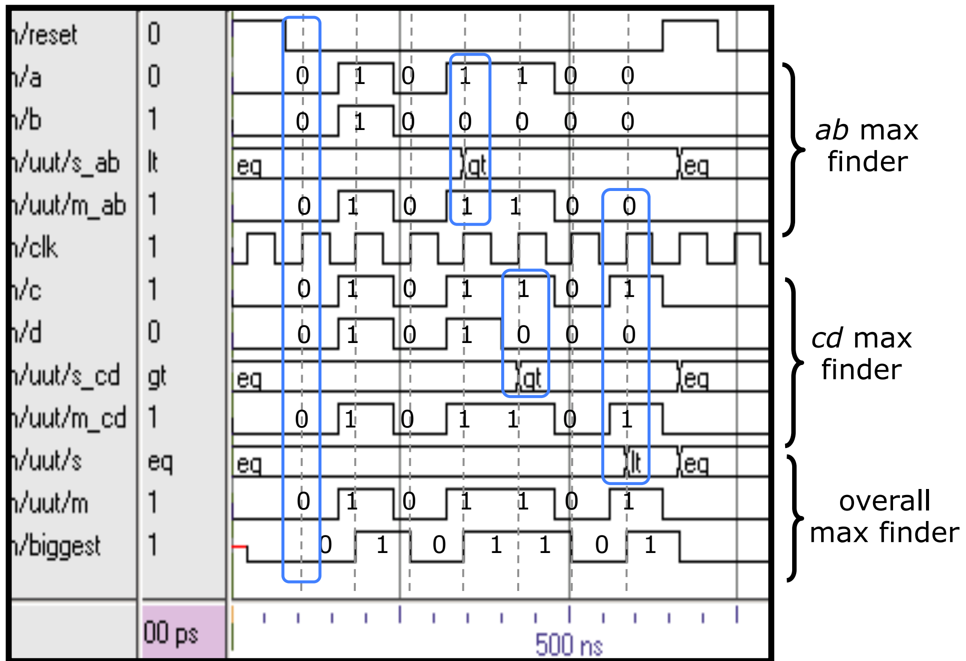
```
signal s_ab, s_cd, s: stateType;
signal m_ab, m_cd, m: std_logic;
begin
    m_ab <= maxBit(s_ab,a,b);
    m_cd <= maxBit(s_cd,c,d);
    m <= maxBit(s,m_ab,m_cd);
    process(clk) begin
        if rising_edge(clk) then
```

```
            assert m >= a or s_ab = lt or s = lt;
            assert m >= b or s_ab = gt or s = lt;
            assert m >= c or s_cd = lt or s = gt;
            assert m >= d or s_cd = gt or s = gt;
            if reset = '1' then
                s_ab <= eq; s_cd <= eq; s <= eq;
                biggest <= '0';
            else
                s_ab <= nextState(s_ab,a,b);
                s_cd <= nextState(s_cd,c,d);
                s <= nextState(s,m_ab,m_cd);
                biggest <= m;
            end if;
        end if;
    end process;
end arch;
```

The first line in this section defines three state registers, one for the 2-way max-finder that determines the larger of the first two inputs, one for the max-finder that determines the larger of the next two and one for the max-finder that determines the overall maximum. The second line defines three intermediate signals: `m_ab` is the larger of the first two inputs, `m_cd` is the larger of the next two and `m` is the overall maximum. The first three lines of the architecture define the output signals from each of the three state machines using the `maxBit` function defined earlier. The state registers are updated by the process, using the `nextState` function.

The four `assert` statements provide a check that the overall output is correct. The first one asserts that either the current `m`-bit is greater than or equal to the current `a`-bit, or the `ab` state machine has already determined that `a<b` or the overall state machine has determined that `m_ab<m_cd`. The other four are defined similarly.

Let's finish with a simulation of the max finder.



Note that the first group of signals relate to the max-finder for the first two inputs, the next group relates to the max-finder for the next pair of inputs and the last group relates to the max-finder for the overall circuit. Observe how the values of inputs a and c are propagated by the first two max-finders, while the larger of these two (c) is propagated by the third one.

10.2 Binary Input Module

Next, we're going to look at the *binary input module* used with our prototype board to translate the button and knob signals into internal signals that can be used with a variety of other circuit modules. We've already looked at the *debouncer* and *knob interface*, which are used by the binary input module, but now we're going to bring it all together.

The binary input module does two things. First, it debounces the button

inputs and provides both the debounced versions of buttons 1-3 and pulse versions of buttons 1-3 for use by other circuit modules. (Button 0 is used to generate a reset signal.) Second, it provides a set of 16 output bits that are controlled by the knob, using the knob interface circuit.

Let's start by looking at the entity declaration and various internal signals of the input module.

```
entity binaryInMod is port(
    clk: in std_logic;
    -- inputs from actual buttons and knob
    btn: in buttons; knob: in knobSigs;
    -- outputs produced using inputs
    resetOut: out std_logic;
    dBtn: out std_logic_vector(3 downto 1);
    pulse: out std_logic_vector(3 downto 1);
    inBits: out word);
end binaryInMod;

architecture a1 of binaryInMod is
component debouncer ... end component;
component knobIntf ... end component;

signal dbb, dbb_prev: buttons;   -- debounced buttons, old vals
signal reset: std_logic;         -- reset from btn(0)
signal tick, clockwise: std_logic; -- from knob interface
signal delta: word;              -- from knob interface
signal bits: word;               -- internal register
```

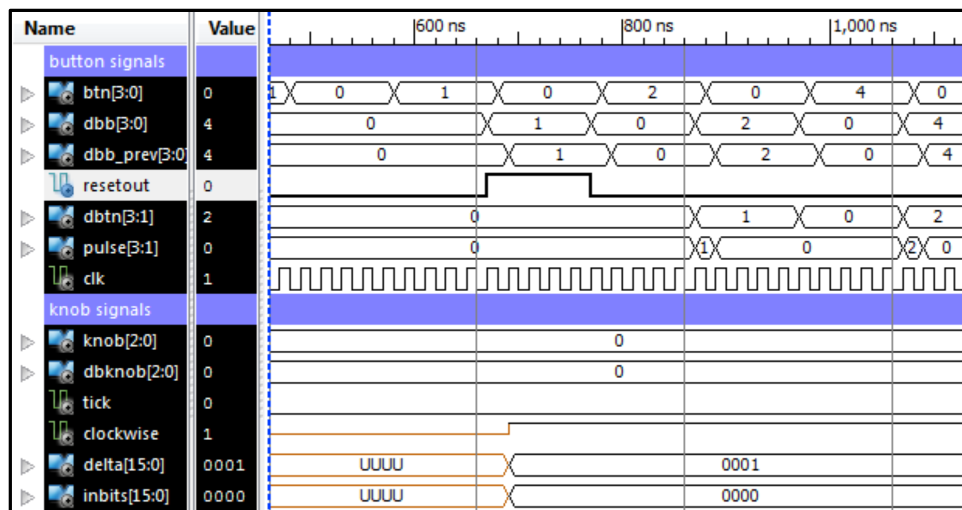
Next, let's look at the body of the architecture

```
begin
    -- debounce the buttons
    db: debouncer generic map(width => 4)
                  port map(clk, btn, dbb);
    dBtn <= dbb(3 downto 1);
```

```
    reset <= dbb(0); resetOut <= reset;

    ki: knobIntf port map(clk, reset, knob,
                          tick, clockwise, delta);

    -- define pulse and inBits
    process (clk) begin
        if rising_edge(clk) then
            dbb_prev <= dbb; -- previous debounced buttons
            if reset = '1' then bits <= (others => '0');
            elsif tick = '1' then
                if clockwise = '1' then
                    bits <= bits + delta;
                else
                    bits <= bits - delta;
                end if;
            end if;
        end if;
    end process;
    pulse <= dbb(3 downto 1) and (not dbb_prev(3 downto 1));
    inBits <= bits;
end a1;
```

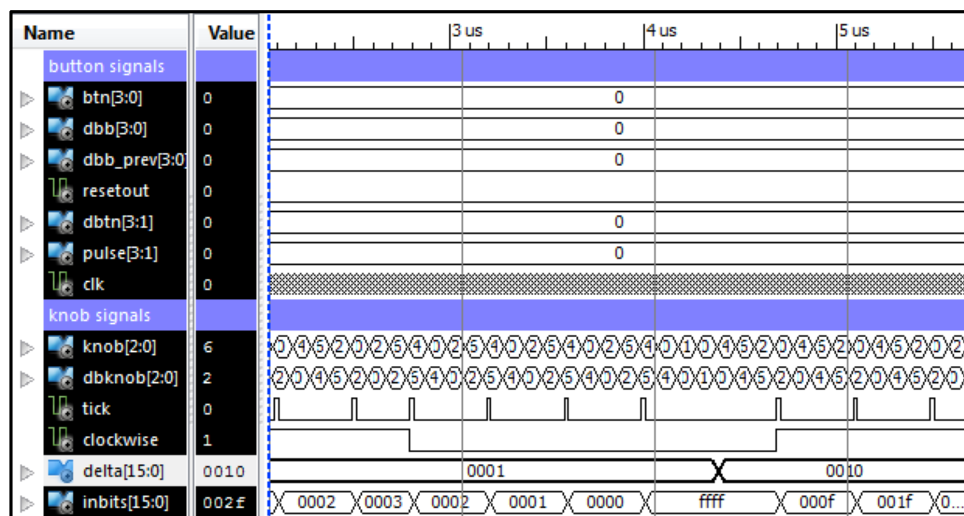
Most of the action here is in the process, which does two things. First, it saves the current debounced button values in the `dbb_prev` register. Second, it adds or subtracts `delta` from the internal `bits` register whenever `tick` goes high. The assignment to `pulse` right after the process produces a one clock-tick wide pulse whenever a debounced button goes from low to high.

The figure below shows a portion of a simulation of the input module.



Notice that the first transition of the buttons is not reflected in the debounced buttons, as it does not last long enough to propagate through the debouncer. The remaining changes to the buttons are reflected in the debounced version after a delay. At times 870 ns and 1070 ns, we can observe two one tick pulses.

The next segment of the simulation output shows how the outputs from the knob interface affect the `inBits` signal.



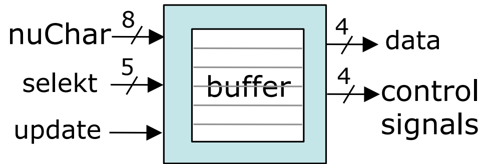
Note how the change to **delta** near the end of this segment affects the way the inBits signal changes.

10.3 LCD Display Module

In this section, we're going to look at a component that is used to provide a higher level interface to the LCD display on our prototype boards. The actual physical device on the board has its own embedded controller and communicates with circuits on the FPGA using a four bit wide data interface and four additional control signals. The device defines a fairly elaborate set of commands that can be used to transfer information and control how it is displayed. While this results in a flexible and fairly feature-rich display device, it makes it cumbersome to use in more typical applications. For this reason, it's useful to have an interface circuit that hides the messy details of the device's interface behind a much simpler, higher level interface.

So, what form should this higher level interface take? Perhaps the simplest approach is to number each of the 32 character positions on the LCD display with an index in 0-31, then design the interface to allow a "client circuit" to write data to any of these character positions by specifying its

index. This leads to the circuit in the diagram below.



The *nuChar* input is an eight bit ASCII character code for a character to be displayed. The *selekt* input specifies the character position on the display where *nuChar* should be written and the *update* signal is asserted by the client whenever it's time to write a new character to the display. This interface allows a client to control the display in a very simple way, without having to be aware of all the low level details required to control the physical device. Interface circuits like this are common in larger systems, where it's frequently useful to hide the details of a lower level interface from clients that are better served by something simple.

The output side of the circuit has the four data signals and four control signals used to control the actual physical device. Internally, the circuit has a 32 byte character buffer in which values written by the client are stored. The circuit periodically transfers each character in this buffer to the physical display device using the output signals on the right. We are not going to describe the details of this portion of the circuit, as they are very dependent on the specifics of the display device. (If you are interested, the reference manual for the prototype board includes a section that describes the display device in some detail.) Instead, we are going to limit ourselves to a description of the "client side" of the interface circuit, since this will give you a more informed understanding of how it can be used.

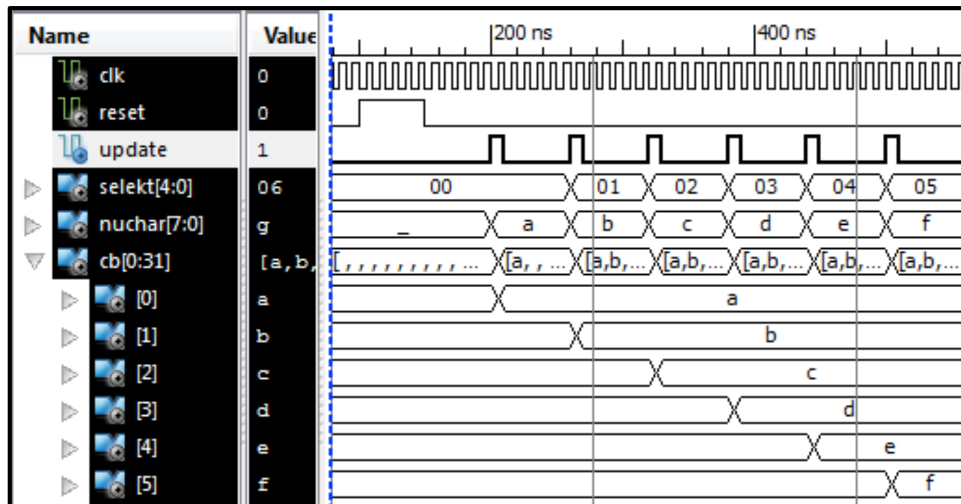
Here is the relevant portion of the architecture for the `lcdDisplay` interface circuit.

```
architecture a1 of lcdDisplay is
type char_buf is array(0 to 31) of byte;
signal cb: char_buf := (others => x"20"); -- space char
...
```

```
begin
    ...
    -- update character buffer when update is asserted
    process(clk) begin
        if rising_edge(clk) then
            if reset = '0' and update = '1' then
                cb(int(selekt)) <= nuChar;
            end if;
        end if;
    end process;
    ...
```

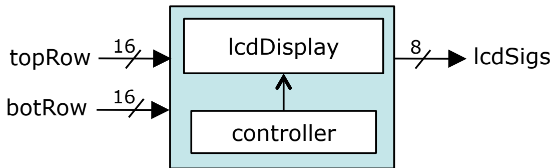
The first couple lines simply define the character buffer as an array of 32 bytes. The process simply writes to the selected byte in the buffer whenever `reset` is low and `update` is asserted.

The figure below shows the initial section of a simulation, where a series of characters are written to the LCD display module. Note that the first six bytes of the character buffer are shown.



10.4 Binary Output Module

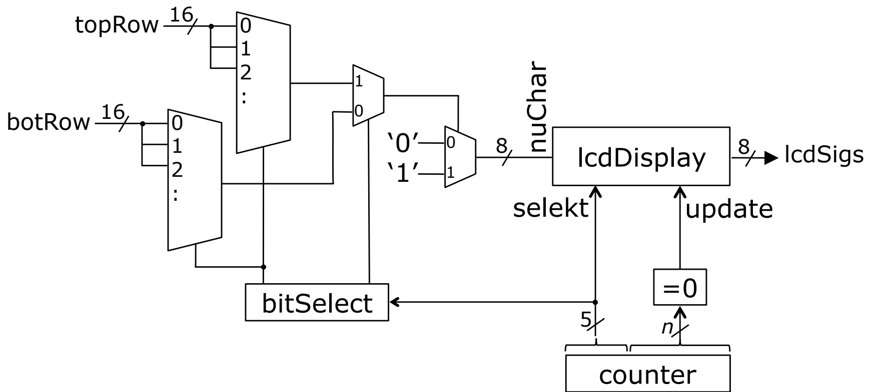
In this section, we'll look at how the binary output module that we've used with the prototype board is implemented using the LCD display module. Recall, that the binary ouput module has two 16 bit inputs, `topRow` and `botRow`. The bits from these inputs are displayed as ASCII characters on the external display device. The figure below shows the high level view of the binary output module.



The LCD display module is used transfer characters to the external display device. The output module's controller simply has to write the appropriate ASCII characters to the display module's internal buffer using the provided interface signals. The controller examines the bits from the two input values one at a time and writes the ASCII character for each bit to the display module. It cycles through all the bits repeatedly, using a counter to control the process.

To complete the design, we need to decide how often the input characters should be written to the display. It turns out that if we write to the display at too high a rate, we can cause excessive contention for the display module's internal buffer. This can cause the external display to flicker. There is also no reason to write to the display at a high rate, since it's going to be viewed by human beings, who cannot react to rapid changes anyway. If we update the display every 20 ms or so, it should be fast enough to appear instantaneous, while minimizing contention for the display module's buffer. Since the prototype board uses a clock with a period of 20 ns, a counter length of 20 bits will give us an update period of about 20 ms.

The figure below shows a suitable circuit for the the output module.



Observe that the *selekt* input of the display module comes from the high-order five bits of the counter. The *update* signal is asserted whenever the low-order bits of the counter are all zero. The mulitplexors at left select a specific bit from one of the two data inputs. The *bit select* block determines which bit is selected. It must account for the fact that the input bits are numbered 15 downto 0, while the indexes of the LCD display positions start with 0 in the top left and increase along the top row, then continue on the next. The *nuChar* input to the display module is either the ASCII character code for 0 or the character code for 1. Which code is used is determined by the selected bit from the two data inputs.

Ok, so let's see how this is expressed using VHDL.

```
entity binaryOutMod is port(
    clk, reset: in std_logic;
    topRow, botRow: in word;
    lcd: out lcdSigs);
end entity binaryOutMod;

architecture a1 of binaryOutMod is
component lcdDisplay ... end component;
-- counter for controlling when to update lcdDisplay
constant CNTR_LENGTH: integer := 6+14*operationMode;
signal counter: std_logic_vector(CNTR_LENGTH-1 downto 0);
```

```
signal lowBits: std_logic_vector(CNTR_LENGTH-6 downto 0);
-- signals for controlling lcdDisplay
signal update: std_logic;
signal selekt: std_logic_vector(4 downto 0);
signal nuChar: std_logic_vector(7 downto 0);
begin
    disp:    lcdDisplay port map(clk, reset, update,
                                 selekt, nuchar, lcd);

    selekt <= counter(CNTR_LENGTH-1 downto CNTR_LENGTH-5);
    lowBits <= counter(CNTR_LENGTH-6 downto 0);
    update <= '1' when lowBits = (lowBits'range => '0') else
              '0';
    nuchar <= x"30" when
        (selekt(4) = '0' and
            topRow((wordSize-1)-int(selekt(3 downto 0))) = '0')
        or (selekt(4) = '1' and
            botRow((wordSize-1)-int(selekt(3 downto 0))) = '0')
        else x"31";

    process(clk) begin
        if rising_edge(clk) then
            if reset = '1' then counter <= (others => '0');
            else counter <= counter + 1;
            end if;
        end if;
    end process;
end a1;
```

The `selekt`, `update` and `nuChar` signals are defined near the center of the architecture body. Note that the assignment for `nuChar` specifies all the bit selection circuitry shown in the diagram using the multiplexors and the *bit select* block. The process simply increments the counter whenever reset is low.

A portion of a simulation of the output module appears below.

Notice that the display module's internal buffer is the last waveform in the window. It shows the ASCII characters for all of the `topRow` bits and the first several bits from `botRow`.

Chapter 11

Verifying Circuit Operation

In this chapter, we're going to focus on methods for effective testing of digital circuits. Testing is a very important part of the circuit design process and companies that produce digital circuit products invest a major part of their development budgets on testing and verification. Indeed, in many projects there are more engineers assigned to the testing team than to the design team.

What makes testing so important is that errors in digital circuits can be very expensive. It can cost millions of dollars to manufactur a custom integrated circuit. If an error is found after the first batch of chips has been produced, a costly "re-spin" may be required to correct the problem. Moreover, the time required to produce a new batch of chips may delay a product release, costing a company far more money in lost sales, and potentially allowing a competitor to gain a significant market advantage. If a design error is discovered only after a product is shipped, the impact can be far worse, requiring a costly product recall and damaging a company's reputation.

There are two major approaches to ensuring the correctness of digital circuits. Testing circuits via simulation is the most widely used method, but for some projects, formal verification methods are used to prove that a circuit is logically correct. In this book, we'll focus on testing, but it's important to keep in mind that more comprehensive methods are being developed and

may be more widely used at some point in the future.

11.1 Assertion Checking in Circuit Specifications

One very powerful tool for discovering errors in a circuits involves the systematic use of `assert` statements in circuit specifications. Assertions can be used to test that the inputs to a module, function or procedure do not violate its input assumptions. For example, here is a procedure that updates an entry in a table with 24 entries, and uses an assert statement to verify that its five bit argument falls within the index range for the table.

```
procedure update(i: in unsigned(4 downto 0);
                 v: in unsigned(7 downto 0)) is begin
    assert 0 <= to_integer(i) and to_integer(i) <= 23;
    if table(to_integer(i)).valid = 1 then
         table(to_integer(i)).value <= v;
    end if;
end procedure update;
```

More complex modules will typically have more elaborate assertions. Note that an `assert` statement may invoke a function that carries out more complex checking. Also keep in mind that assertions have no impact on circuit performance as they are checked by the simulator but ignored by a circuit synthesizer. So, it makes sense to use them extensively.

Here's an example of an assertion being used to check that the outputs of a circuit satisfy a basic consistency condition.

```
procedure minMax(a, b: in  signed(7 downto 0);
                 x, y: out signed(7 downto 0))
is begin
    if a < b then x <= a; y <= b;
    else x <= b; y <= a;
    end if;
    assert x <= y;
end procedure minMax;
```

While the assertion is hardly necessary in such a tiny circuit, it does illustrate an important point. We can often include a useful check on a circuit's operation without having to completely repeat the circuit's computation. Of course the assertion shown above is not enough to catch all possible errors in this circuit, but even assertions that just perform consistency checks like this can be effective at detecting errors that might otherwise be overlooked.

11.2 Testing Combinational Circuits

Let's start our study of testing by looking at how to test combinational circuits effectively. For small circuits, it's often possible to do exhaustive testing, where every output of the circuit is tested for every possible input combination. If the number of inputs is very small (say 6 or less), it's reasonable to check all test cases manually, but as the number of inputs gets larger, it's worthwhile to automate the testing process to allow the computer to check the results. To illustrate this, consider the following circuit.

```
entity negate is
    generic(size: integer := 4);
    port(
        c: in std_logic;  -- negate when c = high
        dIn : in signed(3 downto 0);
        dOut : out signed(3 downto 0));
end negate;
architecture a1 of negate is
begin
    dOut <= (dIn'range => 0) - dIn when c = '0' else dIn;
end a1;
```

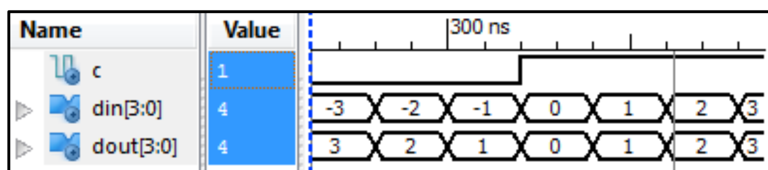
This circuit is supposed to negate its input value (that is, take the 2s-complement) when its control input is high. Otherwise, the data output should equal the input. Notice however, that the actual implementation negates the input when the control input is low, rather than when it is high.

Now let's look at a testbench that checks this circuit.

```
...
use work.txt_util.all;
...
entity testNegate is end testNegate;
architecture ... begin
    uut: negate generic map (size => 4) port map(c, din, dout);
    process begin
        c <= '0';
        for i in -8 to 7 loop
            din <= to_signed(i,4); wait for pause;
            assert dout = din
            report "error when c=0 i=" & str(i);
        end loop;
        c <= '1';
        for i in -8 to 7 loop ... end loop;
        ...
```

First note the `use` statement at the top. This references a package of useful utility functions for dealing with text strings. This package is not part of the IEEE library and so most be included in the working library. Here, we are using it so that we can use the function `str()` to convert an integer to a string representing the value of that integer. Since the circuit has been instantiated with four bit inputs and outputs, the for loops iterate over all the possible values of the data input. Note that the `assert` statement in the first loop is used to check that the output is correct in the case when `c=0`. The second loop covers the case when `c=1`, athough the body of that loop has been omitted.

Here's a portion of the output from a simulation run, which shows the incorrect result.



and here is a view of the simulation console where the assertion reports appear when assertions are violated.

ISim (P.20131013)
File Edit View Simulation Window Layout Help
Instances a... Objects
Simulation Objects for :tb
Instance and Proces
testnegate
Object Name
c
Instanc...
Console

```
This is a Lite version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
at 40 ns: Error: error when c=0 i=-7
at 60 ns: Error: error when c=0 i=-6
at 80 ns: Error: error when c=0 i=-5
at 100 ns: Error: error when c=0 i=-4
at 120 ns: Error: error when c=0 i=-3
at 140 ns: Error: error when c=0 i=-2
```

Console Compilation Log Breakpoints Find in Files

Observe that because the simulator is checking the results, we don't really

need to check the waveform window that closely (although it's still a good idea to do so, the first few times you simulate your circuit, in order to catch errors in your assertions).

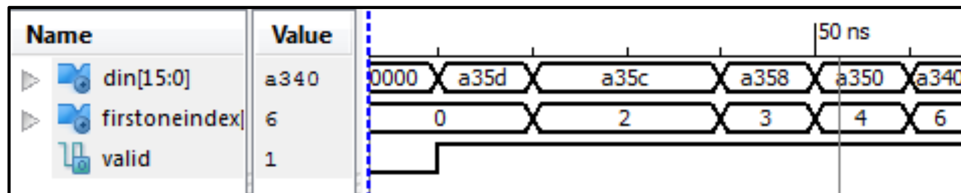
Also note that even though the testbench only checks a four bit version of the circuit, we can be confident (once the error is corrected), that larger versions will work correctly as well, since the circuit architecture is completely independent of the specific signal length used.

Here's part of a testbench for a *findFirstOne* circuit, which outputs the index of the first 1-bit in its data input (starting from the least-significant bit).

```
architecture a1 of testFindFirstOne is begin ...
    uut: findFirstOne port map (dIn, firstOneIndex, valid);
    ...
    process
        -- return copy of x but with bits 0..i set to 0
        function mask(x: word; i: integer) return word ...
    begin
        din <= x"0000"; wait for 10 ns;
        for i in 0 to wordSize-1 loop
            dIn <= mask(x"a35d",i); wait for 10 ns;
            for j in 0 to int(firstOneIndex) - 1 loop
                assert dIn(j) = '0' report "detected error ...
            end loop;
            assert (valid='0' and dIn = (dIn'range => '0')) or
                   (valid='1' and dIn(int(firstOneIndex))='1')
                     report "detected error when i=" & str(i);
        end loop;
        ...
```

This testbench is not testing the circuit exhaustively. Instead it takes a fixed input value (`a35d` in this example) and successively "turns-off" lower order bits, leading to increasing values of `firstOneIndex`. The inner loop checks that all output bits with index values less than `firstOneIndex` are zero and the final assertion checks that the bit specified by `firstOneIndex` is in fact

equal to 1. A portion of the simulation output from this circuit appears below.



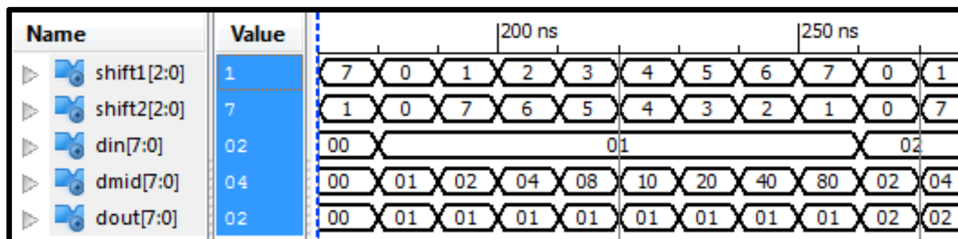
Let's look at a slightly bigger example. A barrel shifter is a circuit with a data input, a data output and a *shift* input. The data output is a rotated version of the data input, where the amount of rotation is determined by the specified shift amount. For example, if the value of the shift input is 2, bit 5 of the output is equal to bit 3 of the input. All other bits of the outputs are mapped similarly, with the highest bit positions from the input "wrapping" around into the lowest bit positions of the output.

Consider an eight bit barrel shifter with a three bit shift input, and suppose we want to test it exhaustively. There are more than two thousand possible input combinations, so we clearly need some way of automating the testing, so that we do not have to check the results manually. One way to simplify the checking is to include two barrel shifters in our test, rather than just one. The output of the first shifter is connected to the input of the second. If we make the sum of the two shift inputs equal to eight, then the output of the second barrel shifter should always be equal to the input of the first, a condition that's easy to check. Here is a portion of a testbench based on this idea.

```
architecture ... begin
   -- instantiate two shifters
   uut1: barrelShift port map (dIn,  shift1, dMid);
   uut2: barrelShift port map (dMid, shift2, dOut);
   ...
   process begin
      wait for 100 ns;
      for i in 0 to 255 loop
```

```
        dIn <= to_unsigned(i,8);
        shift1 <= "000"; shift2 <= "000"; wait for pause;
        assert dIn = dMid
        report "error detected when i=" & str(i) &
                " and shift1=" & str(to_integer(shift1));
        for j in 1 to 7 loop
            shift1 <= to_unsigned(j,3); wait for 1 ps;
            shift2 <= "000" - shift1; wait for pause;
            assert dIn = dOut
            report "error detected when i=" & str(i) &
                   " and shift1=" & str(to_integer(shift1));
        end loop;
    end loop;
...
```

Notice that in the inner loop, there is a 1 picosecond `wait` right after the assignment to `shift1`. This is required to ensure that in the subsequent assignment to `shift2` the simulator uses the new value of `shift1` rather than the previous value. The figure below shows a portion of the simulation of the barrel shifter.

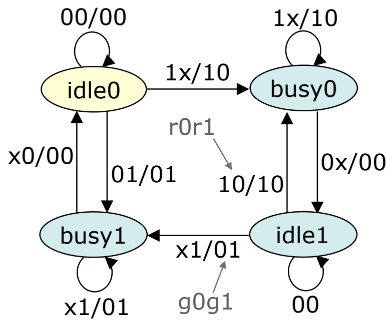


Notice how the output of the second shifter (`dOut`) is equal to the input of the first. You may wonder why the simulation displays the `dOut` signal as changing every 10 ns while `dIn=1`, even though its value remains equal to 1. This is just an artifact of the 1 ps delay in the testbench mentioned earlier.

11.3 Testing State Machines

Testing state machines is generally more complicated than testing combinational circuits, since we have to take into account the state of the circuit, including both the control state and whatever data may be stored in registers. It's also difficult to provide very specific guidelines, because state machines can differ very widely, making it necessary to adapt the testing strategy to the individual situation. Still, there are some general methods that can be widely applied.

Let's start by considering a simple state machine with no data beyond the control state. Specifically, let's consider the asynchronous version (Mealy-mode) of the aribter circuit that was introduced in an earlier chapter. The state transition diagram for that circuit is shown below.



Small state machines like this can be tested exhaustively, using the state diagram. The objective is to find a set of inputs that traverse every transition in the state diagram. For transitions that are labeled by inputs with don't-care conditions, a complete test will follow the transition multiple times, one for each possible input condition. For example, the following input sequence will test all the transitions in the arbiter state diagram.

00, 10, 10, 11, 00, 10, 00, 00, 01, 01, 11, 00, 01, 00, 11, 00, 11, 00

Follow the sequence through the diagram to make sure you understand how this works. To verify that the circuit is correct, the testbench must verify

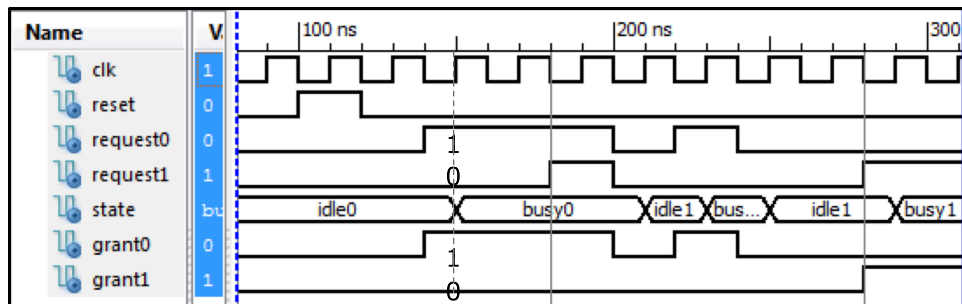
that the output values match the state diagram. In this case, the sequence of output values is

00, 10, 10, 10, 00, 10, 00, 00, 01, 01, 01, 00, 01, 00, 10, 00, 01, 00

Verify this for yourself, using the state diagram. Here is the architecture of a testbench that applies the input sequence above and checks it against the expected output sequence.

```
architecture a1 of testArbiter is ...
type testData is record
    requests: std_logic_vector(0 to 1);
    grants: std_logic_vector(0 to 1);
end;
type testVec is array(natural range <>) of testData;
constant tv: testVec := (
    ("00", "00"), ("10", "10"), ("10", "10"), ...);
begin ...
    process
        procedure doTest(td: testData) is
        variable r0, r1, g0, g1: std_logic;
        begin
            r0 := td.requests(0); r1 := td.requests(1);
            g0 := td.grants(0);   g1 := td.grants(1);
            request0 <= r0; request1 <= r1; wait for 1 ps;
            assert grant0 = g0 and grant1 = g1
            report "error detected when r0,r1=" & str(r0) ...
            wait for clk_period;
        end;
   begin
      wait for 100 ns;
        reset <= '1'; wait for clk_period; reset <= '0';
        for i in tv'range loop doTest(tv(i)); end loop;
        ...
end; end;
```

This testbench illustrates a general technique that can make it easier to specify the input sequence for a circuit and the expected output sequence. The `testData` record includes a pair of request inputs and the corresponding pair of expected outputs. The `testVec` array contains the test sequence, with each record in the array defining one step in the overall test sequence. The `doTest` procedure carries out a single step in the test sequence, assigning the specified test inputs to the circuit and comparing the actual circuit outputs to the expected output values. An `assert` statement does the actual checking and displays a suitable error message when a violation is detected. The body of the process resets the circuit and applies the entire test sequence by iterating through the `testVec` array and invoking `doTest` on each record. A portion of the output of this simulation appears below. The input and output values at time 150 ns have been highlighted.



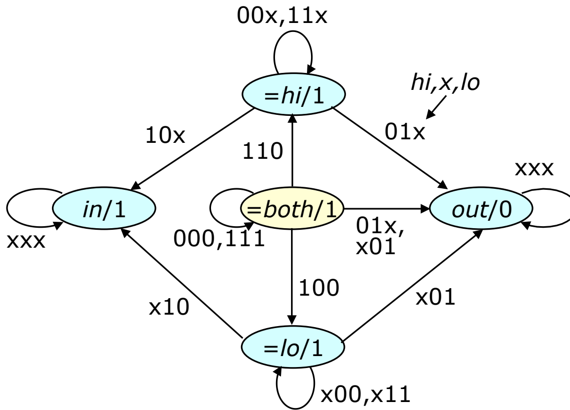
There are a couple things worth noting at this point. First, the test only applies the reset signal once. This is arguably a reasonable thing to do, since the standard implementation of reset in VHDL generally ignores the state and all input conditions, so it's not something that we're likely to get wrong. Still, if we really wanted to be thorough, we could generalize the testbench to enable checking that the circuit always resets correctly. There is a bigger issue, which is that while our test does allow us to verify that the circuit implemented by the VHDL correctly reflects the state diagram, it does nothing to help us identify possible errors in the state diagram itself. Moreover, in many situations, we may not have access to a state diagram of the circuit we're testing. In this case, we must be prepared to construct

our test sequence based purely on the high level specification of the circuit itself. For the arbiter, it's not too difficult to construct a comprehensive test sequence, even without knowledge of the state diagram, but in other situations it can be challenging to develop an effective test based only on the specification.

Next, let's look at an example of a testbench for a circuit that operates on bit-serial input data. This circuit, called `inRange`, determines if one of its bit-serial input sequences lies within the range of values defined by its other two inputs. Specifically, the inputs are `hi, x` and `lo` and the circuit's `inRange` output should be high when the numerical value of the bit sequence for `x` lies between the values for `hi` and `lo`. The three input values are presented one bit at a time, with the most significant bit first. Consider an example in which the bits of `hi` are 01110, the bits of `x` are 01011 and the bits of `lo` are 01001. Now the bits for the three inputs are all received one bit at a time. So, on the first clock tick following reset, the bits 000 will be received, on the second clock tick, the bits 111 will be received, and so forth.

The `inRange` output will always start out high, and will remain high if `x` falls between `hi` and `lo`. However, if `x` does not fall between `hi` and `lo`, `inRange` will eventually go low. When it goes low depends on when the circuit is first able to detect that `x` is outside the range. For example if the bits of `hi` are 011001, the bits of `x` are 011010 and the bits of `lo` are 010011, the circuit will not be able to detect the fact that `x` is outside the range until the fifth clock tick, since up until that point, the bits of `x` that have been received are the same as the bits of `hi` that have been received.

The state diagram shown below can be used to implement the `inRange` circuit.



The circuit starts in the center state (labelled =*both*) when the reset signal goes low. It remains in this state so long as the three input bits are all equal to each other. The significance of the state name is that, based on the bits received so far, `x` is equal to both `hi` and `lo`. If, while we are this state, the next input bit of `x` is either greater than the next input bit of `hi` or smaller than the next input bit of `lo`, the circuit transitions to the *out* state, meaning that `x` is outside the range. If, on the other hand, the next input bit of `x` is equal to the next bit of `hi`, but greater than the next input bit of `lo`, the circuit switches to the =*hi* state, signifying that based on the bits of received so far, `x=hi>lo`. Examine the remaining transitions and make sure you understand why each transition makes sense. Also, note that the `inRange` output remains high in all states except the *out* state.

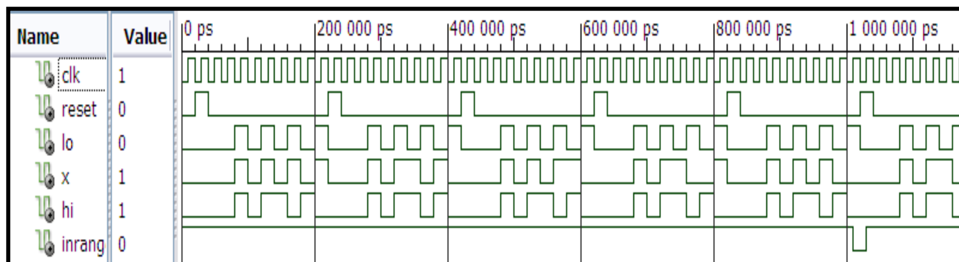
Now, let's look at how we can setup a testbench to conveniently test this state machine.

```
architecture a1 of testInRange is
...
type testData is record
lo, x, hi, result: std_logic_vector(7 downto 0);
end record inVec;
type testVec is array(natural range <>) of testData;
constant tv: testVec := (
```

```
    (x"55", x"55", x"5a", x"ff"),
    (x"55", x"54", x"5a", x"fe"), ...);
begin
    process begin
        wait for pause;
        for i in tv'low to tv'high loop
            reset <= '1'; wait for pause;
            reset <= '0'; wait for pause;
            for j in 7 downto 0 loop
                lo <= tv(i).lo(j);
                x <= tv(i).x(j);
                hi <= tv(i).hi(j);
                wait for pause;
                assert (inRangeOut=tv(i).result(j)) report ...
            end loop;
            lo <= '0'; x <= '0'; hi <= '0';
        end loop;
        ...
```

As in the previous example, we have defined a `testVec` array which consists of records containing the test data. In this case however, each record contains logic vectors that represet all the bit sequences for `hi`, `x` and `lo`, plus the bit sequence for the `inRange` output. The outer loop iterates through the entries in the `testVec` array, while the inner loop iterates through the sequences of input bits, and applies those bits to the circuit inputs. It then checks the output and reports any discrepancy between the actual output bits and expected output.

A portion of the the output of the resulting simulation appears below.



We'll finish up this section with another example, this time using the pulseCount circuit introduced in an earlier chapter. Recall that this circuit loads a count value right after reset goes low and then observes its data input looking for complete "pulses". When the number of pulses observed is equal to the count, it raises an event output. It uses an internal counter to track the number of pulses seen so far.

When testing state machines with internal data, it can be useful to have the testbench check the internal data values, in addition to the circuit outputs. Unfortunately, VHDL does not provide a mechanism to allow the testbench to directly refer to an internal signal of a circuit being tested. However, we can get the same effect by adding auxiliary outputs to the VHDL specification for use when testing. These extra outputs can be "commented-out" of the source file when a circuit has been fully tested on its own and is ready to be integrated into whatever larger system it may be part of.

We'll use the pulseCount circuit to demonstrate this idea. We first modify the entity declaration.

```
entity pulseCount is port(
    cntOut: out nibble; -- comment out, when not testing
    clk, reset: in std_logic;
    N: in nibble;
    dIn: in std_logic;
    eventOut: out std_logic);
end pulseCount;
```

We also need to add the following assignment to the architecture.

```
cntOut <= cnt;
```

We can now verify the internal register value in addition to the event output.

```
begin
    uut: pulseCount port map(counter,clk,reset,N,dIn,eventOut);
    ...
    process
    procedure doTest(count: std_logic_vector(3 downto 0);
                     dInBits: std_logic_vector(0 to 19);
                     evBits: std_logic_vector(0 to 19);
                     cntVals: std_logic_vector(79 downto 0)) is
    begin
        reset <= '1'; wait for pause; reset <= '0';
        N <= count; wait for clk_period; N <= (others=>'0');
        for i in 0 to 19 loop
            dIn <= dInBits(i); wait for clk_period;
            assert eventOut = evBits(i) and
                counter = cntVals(4*(19-i)+3 downto 4*(19-i))
            report "error detected at step " & str(i) ...
        end loop;
    end;
    begin
        wait for 100 ns;
        doTest(x"0", "01110010100110111011",
                     "11111111111111111111",
                    x"00000000000000000000");
        doTest(x"1", "01110010100110111011",
                     "00001111111111111111",
                    x"11110000000000000000");
        doTest(x"1", "11110010100110111011",
                     "00000001111111111111",
                    x"11111110000000000000");
        doTest(x"2", "01110010100110111011",
                     "00000001111111111111",
                    x"22221110000000000000");
        doTest(x"5", "01110010100110111011",
```

```
                    "00000000000000000111",
                   x"55554443322221111000");
        doTest(x"5", "11110010100110111011",
                    "00000000000000000000",
                   x"55555554433332222111");
        ...
```

Observe that binary format is used to represent the input data and expected event output, while hex is used for the expected count values. A portion of the simulation output appears below.

11.4 Testing Larger Circuits

As digital systems get larger and more complicated, testing them becomes increasingly challenging. Perhaps the most important tool for meeting this challenge is the systematic application of *unit testing*. That is, testing each individual design unit on its own, before proceeding to testing larger-scale subsystems. This has some important advantages. First, it's easier to find errors when testing an individual design unit than when testing a larger subsystem. Because a testbench only has access to the inputs and outputs of the top-level circuit, it cannot automatically check the correctness of values of internal signals. Consequently, design errors in internal modules may be difficult to check automatically at the subsystem level. (We can still manually check things by adding internal signals to the waveform window, and carefully checking the circuit's operation, but this can be time-consuming, and it's easy

to miss things when we are relying on manual checking alone.) Second, once the smaller components of a larger system have been tested in isolation, the higher level testing can focus on interactions among the components rather than their individual operation. This greatly reduces the scope of what needs to be tested at the system level. Third, in most professional design settings, design components are re-used in different projects and adapted for different purposes over time. A component that has been carefully tested as an independent unit is more valuable, because it allows others to have greater confidence in its correctness, when they seek to re-use it.

Now, even at the level of individual design units, testing can be become challenging as the design units get larger. State machines that incorporate large amounts of internal data can be tricky to test completely, because it's simply not practical to test them exhaustively, as we can for smalller circuits. Instead, one needs to focus on key relationships among the state machine's internal values. For example, check that the circuit operates correctly when when two data values are equal, or one is slighly larger, or smaller than the other. Look for "corner-cases" where input values reach extremes of their allowed ranges. In general look for unusual cases that might have been overlooked in the design process. Experience has shown that design errors are most often found when the circuit's input is atypical, in one way or another.

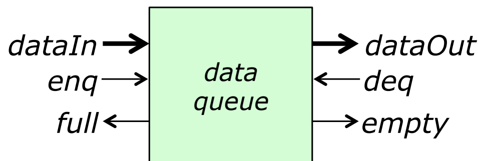
Chapter 12

Continuing Design Studies

In this chapter, we'll be looking at circuits that implement different kinds of queues. The first is a simple data queue, designed to allow two different parts of a circuit to pass data values in an asynchronous fashion. The second is a packet FIFO that operates on larger multi-word blocks of information. The third is a priority queue that maintains a set of (key, value) pairs and supports efficient access to the stored value with the smallest key.

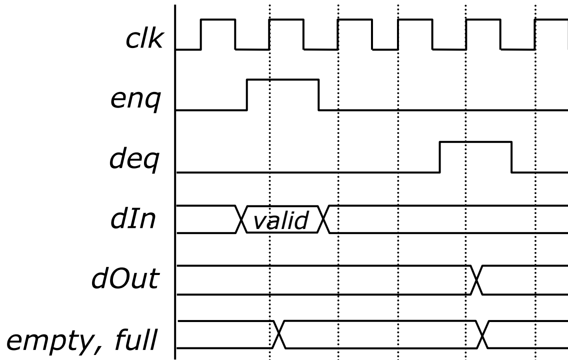
12.1 Simple Data Queue

We'll start with a circuit that implements a simple data queue, illustrated in the figure below.

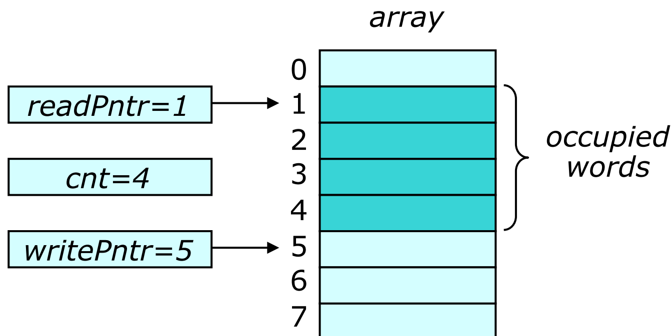


This component can be used to let two parts of a larger circuit pass data in an asynchronous fashion. One interface allows a *producer* to add data to the end of the queue, while the other allows a *consumer* to retrieve data from the front of the queue. These operations are requested using control inputs *enq*

and *deq*. Outputs *empty* and *full* indicate the status of the queue. Whenever the queue is not empty, the data output is the value of the first item in the queue. Note that since the producer and consumer operate independently of each other, the circuit must support simultaneous enqueue and dequeue operations. The interface timing diagram below illustrates the typical operation.



There are several ways that a queue can be implemented, but perhaps the most straightforward uses an array of storage locations, implemented as a memory. A *readPointer* identifies the storage location where the "first" item in the queue can be found, while a separate *writePointer* identifies the location where the next arriving value can be written to the array. A *counter* keeps track of the number of array locations currently in use. This approach is illustrated in the figure below.



The write pointer is advanced as new values are added to the queue, eventually wrapping around to the first position in the array. Similarly, the read pointer is advanced as values are removed from the queue. The first part of a VHDL specification for the queue appears below.

```
entity queue is port (
    clk, reset: in std_logic;
    -- producer interface
    dataIn: in word;
    enq: in std_logic;
    full: out std_logic;
    -- consumer interface
    deq: in std_logic;
    dataOut: out word;
    empty: out std_logic);
end queue;

architecture arch of queue is
-- data storage array
constant qSize: integer := 16;
type qStoreTyp is array(0 to qSize-1) of word;
signal qStore: qStoreTyp;

-- pointers and counter
```

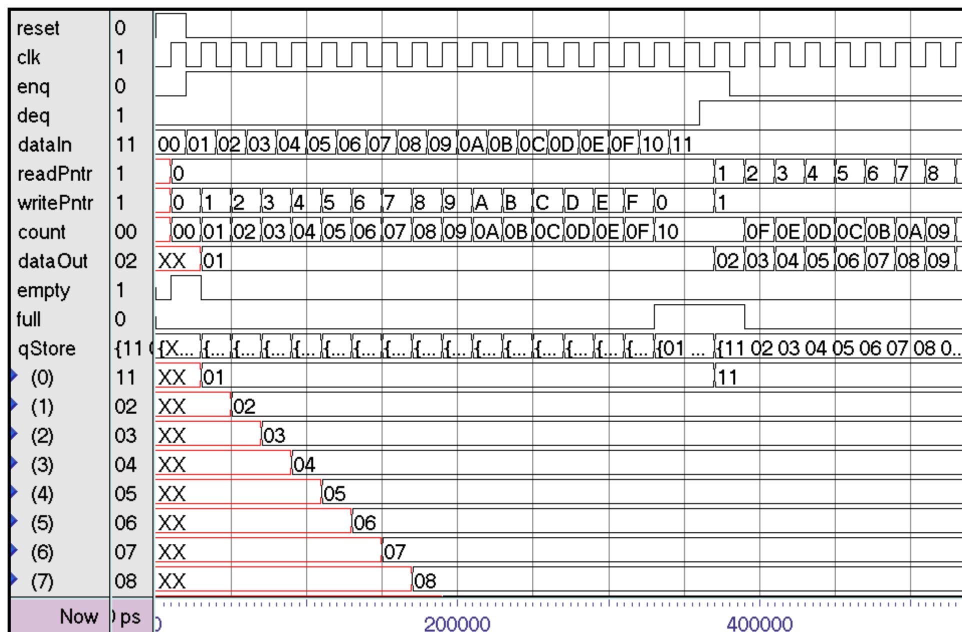
```
signal readPntr, writePntr: std_logic_vector(3 downto 0);
signal count: std_logic_vector(4 downto 0);
```

Notice that the read and write pointers are dimensioned to match the size of the data storage array, while the counter has one extra bit, so that it can represent both the number of values in a full queue and the number in an empty queue. Here is the body of the architecture.

```
begin
    process (clk) begin
        if rising_edge(clk) then
            if reset = '1' then
                readPntr <= (others => '0');
                writePntr <= (others => '0');
                count <= (others => '0');
            else
                -- simultaneous enqueue/dequeue operations
                if enq = '1' and deq = '1' then
                    if count = 0 then -- ignore deq when empty
                        qStore(int(writePntr)) <= dataIn;
                        writePntr <= writePntr + 1;
                        count <= count + 1;
                    else
                        qStore(int(writePntr)) <= dataIn;
                        readPntr <= readPntr + 1;
                        writePntr <= writePntr + 1;
                    end if;
                -- simple enqueue
                elsif enq = '1' and count /= qSize then
                    qStore(int(writePntr)) <= dataIn;
                    writePntr <= writePntr + 1;
                    count <= count + 1;
                -- simple dequeue
                elsif deq = '1' and count /= 0 then
                    readPntr <= readPntr + 1;
```

```
                    count <= count - 1;
                end if;
            end if;
        end if;
    end process;
    -- synchronous outputs
    dataOut <= qStore(int(readPntr));
    empty <= '1' when count = 0 else '0';
    full <= '1' when count = qSize else '0';
end arch;
```

Observe that enqueue operations on full queues are ignored, while dequeue operations on empty queues are ignored. A portion of a simulation is shown below. Note the queue contents in the bottom part of the waveform display and the pointer and counter values near the top.



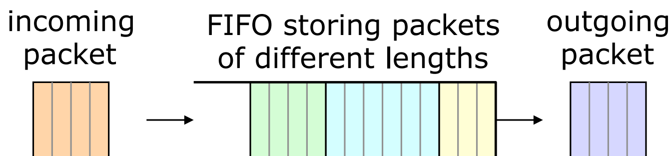
We chose to use a counter to keep track of the number of items in the

queue. This makes it easy to determine when the queue is empty or full. An alternative approach replaces the counter with a single flip flop which is high when the queue is full. Notice that whenever the queue is either empty or full, the read and write pointers are equal to each other. The full flip flop can be set to 1 whenever an enqueue operation makes the write pointer equal to the read pointer.

Still another variant on the queue eliminates the full flip flop. Instead, it simply reduces the number of items that can be stored in the queue. In this version, we consider the queue to be full whenever the value of the read pointer is one larger than the write pointer.

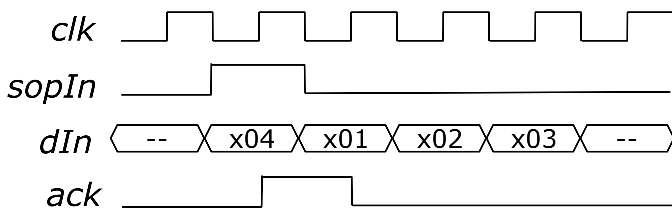
12.2 Packet FIFO

A packet is a block of data consisting of multiple words, with a pre-defined format. It can be used by a producer/consumer pair to pass larger blocks of information than can be conveniently transferred in a single word. This is illustrated in the figure below.



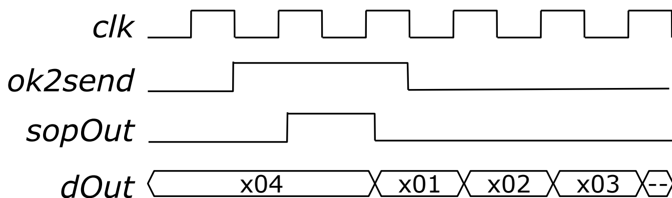
In this example, we will assume that the words are four bits wide and that the first word of every packet contains its length (that is, the number of words in the packet). We'll constrain the length to be at least three words and at most six and we'll raise an error signal if the number words in a packet falls outside this range. Because the producer cannot know in advance if its next packet will fit in the FIFO, we allow the producer to simply send the packet to the FIFO component, and have the FIFO raise an *acknowledgement* signal if it has room to store the entire packet. The producer uses a *start-of-packet* signal to identify the first word of a packet. This interface is illustrated below.

The acknowledgment signal is raised right after the first word of the packet



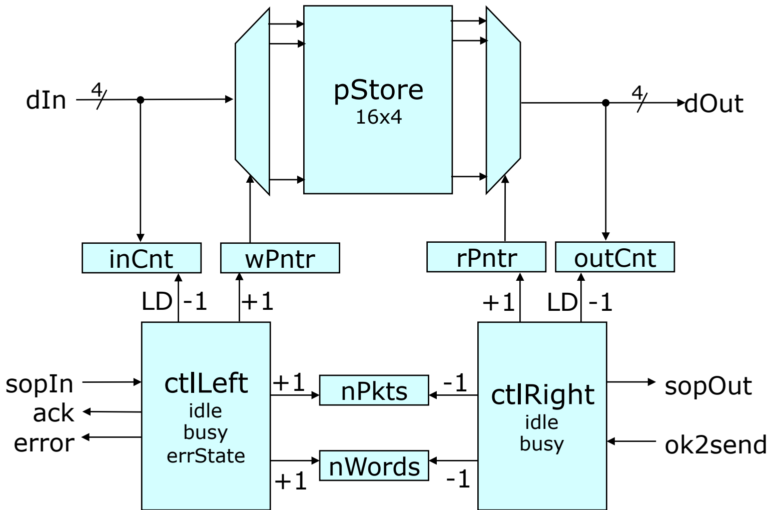
arrives, if there is room for the packet in the FIFO. If there is not room, it remains low. The producer is free to re-try sending a packet until the FIFO is able to accept it.

The consumer-side interface has a similar character. The consumer has an *ok2send* signal that it uses to tell the FIFO when it is ready to receive a packet. The FIFO sends its first packet whenever the consumer is ready and there is a packet to send. It identifies the first word of the packet being sent using its own start-of-packet signal. This is illustrated below.



As with the simple data queue, we can store the packets in a memory, using a read pointer and a write pointer. However, the FIFO circuit needs to keep track of both the number of packets that are present in the FIFO and the number of words that are in use. It also has to use the length of an arriving packet to determine how many incoming words to store. Similarly, it must use the length of a departing packet to determine when the last word of the packet has been sent.

Designing a control circuit that simultaneously manages the arriving and departing packets can be tricky. Here, we'll use another approach using two separate controllers, one to handle incoming packets and other to handle outgoing packets. This is shown in the following block diagram.

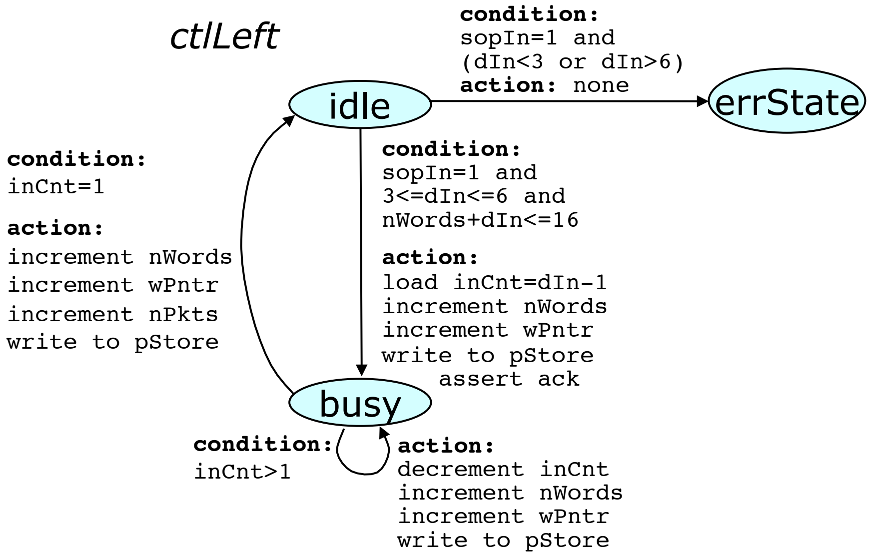


The top of the diagram shows a packet storage block, implemented using a memory. The write pointer controls where each word of an arriving packet is placed in the memory, while an input counter keeps track of the number of words left to be read. It is loaded from the data input when the first word of the packet arrives. The write pointer and input counter are controlled by the "left controller", which manages the producer-side interface.

The consumer-side interface is controlled by the "right-controller" which controls the read pointer and an output counter that keeps track of the number of words remaining in the outgoing packet.

The two controllers share two counters, one which keeps track of the total number of words in use in the memory, and another that keeps track of the number of packets. These are incremented by the left controller and decremented by the right controller.

A state diagram for the left controller is shown below.



When the controller is in the *idle* state, it is just waiting for a new packet to arrive. In this state, if the *sopIn* signal goes high, the controller switches to the *busy* state or the *error* state, based on whether the length of the arriving packet is in the allowed range or not. A transition to the busy state will only occur if the packet length is in range and if the packet will fit in the memory. (The diagram omits the self-loop for the idle state, since no action is required in this case.) If the controller does transition to the busy state, it loads the input counter, writes the first word of the packet to the memory, advances the write pointer and increments the count of the number of words in use. It also raises the acknowledgment signal to let the producer know that the packet can be accepted. The controller remains in the busy state as the packet arrives, then transitions to the idle state when the last word arrives. The right-hand controller is similar.

The first part of a VHDL spec for the packet FIFO appears below.

```
entity pFIFO is port(
```

```
    clk, reset: in std_logic;
    -- left interface
    dIn: in word;
    sopIn: in std_logic;
    ack, errSig: out std_logic;
    -- right interface
    dOut: out word;
    sopOut: out std_logic;
    ok2send: in std_logic);
end pFIFO;

architecture arch1 of pFIFO is
constant psSiz : integer := 16;
subtype register is std_logic_vector(4 downto 0);
-- number of words left to arrive/leave
signal inCnt, outCnt: word;
-- pointer to next word to write/read
signal wPntr, rPntr: register;
-- number of words/packets stored
signal nWords, nPkts: register;
-- Packet store
type pStoreType is array(0 to psSiz-1) of word;
signal pStore: pStoreType;
- state machines controlling input and output
type stateType is (idle, busy, errState);
signal leftCtl, rightCtl: stateType;
-- auxiliary signals
signal inRange, validArrival, enoughRoom: std_logic;
signal nWordsPlus, nWordsMinus: std_logic;
signal nPktsPlus, nPktsMinus: std_logic;
```

The left and right controllers each have their own state variable. We'll describe the various auxiliary next.

```
begin
```

```
-- inRange is high if an arriving packet has a legal length
-- is only used when sopIn is high
inRange <= '1' when dIn >= x"3" and dIn <= x"6" else '0';
-- enoughRoom is high if we have room for an arriving packet
enoughRoom <= '1' when (('0' & dIn) + nWords) <= psSiz
              else '0';
-- validArival is high whenever an arriving packet can
-- be accepted
validArrival <= '1' when leftCtl = idle and sopIn = '1' and
                           inRange = '1' and enoughRoom = '1
                else '0';
```

The **inRange** signal checks that the value on the data inputs is within the allowed range for a packet length. It is only used when the **sopIn** input is high. The **enoughRoom** signal is used to determine if an arriving packet will fit in the available space. The **validArrival** signal is high on the first byte of a packet that can be accepted by the FIFO.

Now that we have these preliminaries out of the way, let's take a look at the process that implements the left controller.

```
process (clk) begin -- process for left controller
    if rising_edge(clk) then
        ack <= '0';  -- default value
        if reset = '1' then
            leftCtl <= idle;
            inCnt <= (others => '0');
            wPntr <= (others => '0');
        else
            if leftCtl = idle then
                if validArrival = '1' then
                    pStore(int(wPntr)) <= dIn;
                    inCnt <= dIn - 1; wPntr <= wPntr + 1;
                    ack <= '1'; leftCtl <= busy;
                elsif sopIn = '1' and inRange = '0' then
                   leftCtl <= errState;
```

```
                end if;
            elsif leftCtl = busy then
                pStore(int(wPntr)) <= dIn;
                inCnt <= inCnt - 1; wPntr <= wPntr + 1;
                if inCnt = 1 then leftCtl <= idle; end if;
            end if;
        end if;
    end if;
end process;
```

Note that the process implements the state transitions specified in the diagram, it writes arriving data to the packet store and updates the input counter and write pointer as required. It also raises the acknowledgement signal when an arriving packt can be accepted. However, the process does not increment the `nWords` and `nPkts` registers. This is done by a separate process using the `nWordsPlus` and `nPktsPlus` signals defined below.

```
-- outputs of left controller
nWordsPlus <= '1' when leftCtl = busy or validArrival = '1'
              else '0';
nPktsPlus <= '1' when leftCtl = busy and inCnt = 1 else '0';
errSig <= '1' when leftCtl = errState else '0';
```

The right controller is similar to the left.

```
process (clk) begin -- process for right controller
    if rising_edge(clk) then
        sopOut <= '0';
        if reset = '1' then
            rightCtl <= idle;
            outCnt <= (others => '0');
            rPntr <= (others => '0');
        else
            if rightCtl = idle then
                if ok2send = '1' and nPkts > 0 then
                    outCnt <= pStore(int(rPntr));
```

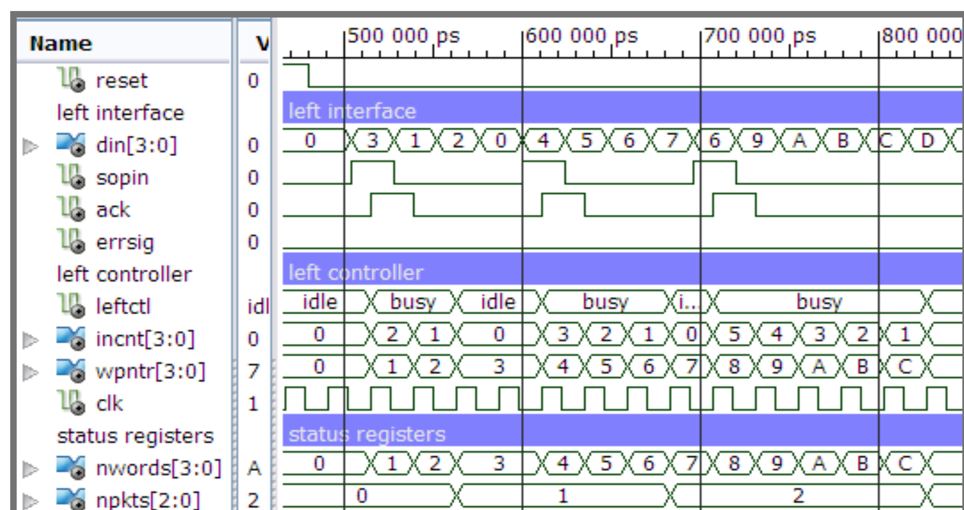
```
                    rightCtl <= busy; sopOut <= '1';
                end if;
            elsif rightCtl = busy then
                outCnt <= outCnt - 1;
                rPntr <= rPntr + 1;
                if outCnt = 1 then
                    rightCtl <= idle;
                end if;
            end if;
        end if;
    end if;
end process;
-- outputs of right controller
nPktsMinus <= '1' when rightCtl = busy and outCnt = 1
              else '0';
nWordsMinus <= '1' when rightCtl /= idle else '0';
dOut <= pStore(int(rPntr));
```

Finally, here is the process that maintains the nWords and nPkts registers.

```
-- process for updating nWords and nPkts
process(clk) begin
    if rising_edge(clk) then
        if reset = '1' then
            nWords <= (others => '0');
            nPkts <= (others => '0');
        else
            if nWordsPlus > nWordsMinus then
                nWords <= nWords + 1;
            elsif nWordsPlus < nWordsMinus then
                nWords <= nWords - 1;
            end if;
            if nPktsPlus > nPktsMinus then
                nPkts <= nPkts + 1;
            elsif nPktsPlus < nPktsMinus then
```

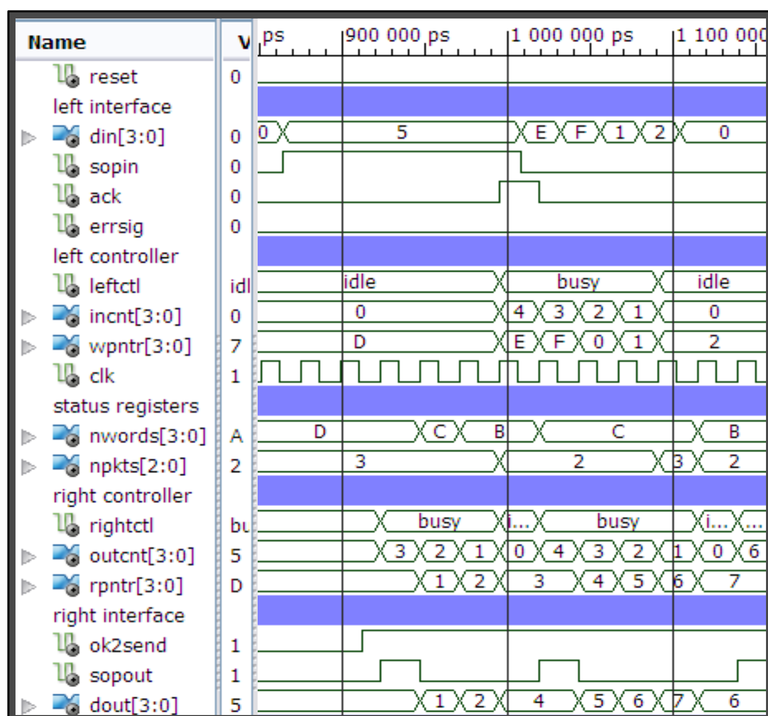
```
                    nPkts <= nPkts - 1;
                end if;
            end if;
        end if;
    end process;
end arch1;
```

The figure below shows a portion of a simulation, showing the arrival of several packets.



Note that the `sopIn` signal goes high on the first word of each arriving packet, and that an acknowledgement is generated immediately by the controller. Note how the input counter and write pointer are updated as the packets arrive. Also observe the updating of the `nWords` and `nPkts` registers.

The next figure shows a portion of the simulation where packet arrivals and departures overlap.



Notice how the next arriving packet is delayed because the FIFO does not yet have room to accommodate the incoming packet. Once packets begin to leave the FIFO, more space becomes available allowing new packets to arrive. Observe that the `nWords` and `nPkts` signals do not much change much during this period, reflecting the overlapping arrivals and departures.

12.3 Priority Queue

In this section, we'll look at a *priority queue*. This is a data structure that stores (key,value) pairs and provides efficient access to the pair with the smallest key. We'll look at a hardware implementation of a priority queue that can perform insertions and deletions in constant time. The priority queue is organized around a fairly simple idea that is illustrated in the fol-

lowing figure.

7,8	4,2	6,17	-	-
2,6	3,1	5,13	8,2	-

In this diagram, each of the occupied cells stores a (key,value) pair. Unoccupied cells are marked with a dash. The cells are partially ordered to support efficient operations on the entire array. Specifically, the pairs in the bottom row are ordered by their keys, with the smallest key appearing in the bottom-left cell. Each column is also ordered, with the smaller key appearing in the bottom row. The unoccupied cells are always at the right, with at most one column having a single unoccupied cell. When a column has one unoccupied cell, that cell is in the top row. If the cells satisfy these constraints, the bottom left cell is guaranteed to have the smallest key in the array.

This arrangement supports efficient insertion of new pairs and efficient extraction of the pair with the smallest key. The figure below illustrates an insertion operation, which proceeds in two steps.

shift top row right

1,11	7,8	4,2	6,17	-
2,6	3,1	5,13	8,2	-

compare & swap in each column

2,6	7,8	5,13	8,2	-
1,11	3,1	4,2	6,17	-

In the first step, the new (key,value) pair is inserted into the top left cell, with all others pairs in the top row shifting to the right, to make room. In the second step, the keys of the pairs in each column are compared to one another and if the top pair has a smaller key than the bottom pair, then the pairs swap positions. In the example, all but one of the pairs gets swapped.

It's also easy to remove the pair with the smallest key. We simply shift the bottom row to the left, then compare and swap pairs within each column.

From a hardware perspective, what's appealing about this approach is that it can be implemented by an array of independent components that communicate only with neighboring components. This allows the components in an integrated circuit chip to be arranged in the same way as the cells in the diagram. The locality of the communication among components leads to a compact and efficient circuit.

Here is the first part of a VHDL specification of the priority queue.

```
entity priQueue is port(
    clk, reset: in std_logic;
    insert, delete: in std_logic;      -- control signals
    key, value: in word;               -- incoming pair
    smallKey, smallValue: out word;    -- outgoing pair
    busy, empty, full: out std_logic); -- status signals
end priQueue;

architecture arch1 of priQueue is
constant rowSize: integer := 4;
-- array of cells
type pair is record
    valid: std_logic;
    key, value: word;
end record pair;
type rowTyp is array(0 to rowSize-1) of pair;
signal top, bot: rowTyp;
-- state type
type state_type is (ready, compareAndSwap);
signal state: state_type;
```

The information associated with each cell is defined as a record containing a `valid` bit plus the `key` and `value`. The two rows are defined as separate arrays to facilitate independent operations.

The next portion the architecture shows the circuit initialization and the first step in the insertion operation.

```
begin
    process(clk) begin
        if rising_edge(clk) then
            if reset = '1' then
                for i in 0 to rowSize-1 loop
                    top(i).valid <= '0'; bot(i).valid <= '0';
                end loop;
                state <= ready;
            elsif state = ready and insert = '1' then
                if top(rowSize-1).valid /= '1' then
                    top(1 to rowSize-1) <= top(0 to rowSize-2);
                    top(0) <= ('1',key,value);
                    state <= compareAndSwap;
                end if;
```

The two assignments to `top` shift the new pair into the top row. The next portion of the specification shows the first part of the delete operation plus the processing of the compare and swap step.

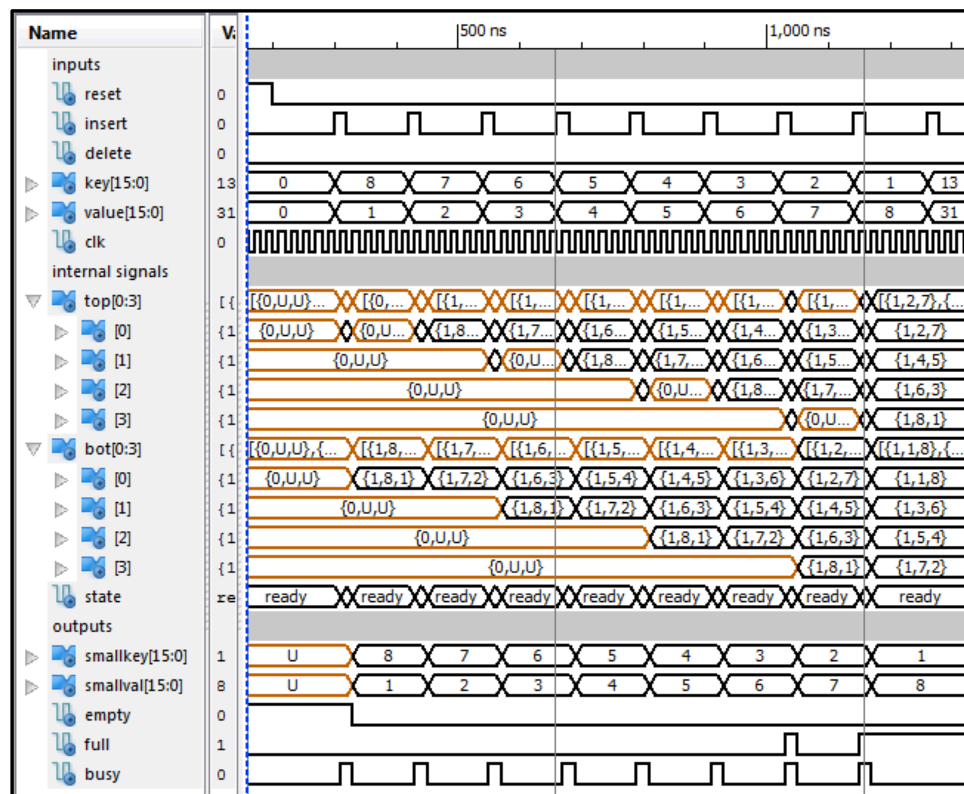
```
            elsif state = ready and delete = '1' then
                if bot(0).valid /= '0' then
                    bot(0 to rowSize-2) <= bot(1 to rowSize-1);
                    bot(rowSize-1).valid <= '0';
                    state <= compareAndSwap;
                end if;
            elsif state = compareAndSwap then
                for i in 0 to rowSize-1 loop
                    if top(i).valid = '1' and
                        (bot(i).valid = '0 or
```

```
                        top(i).key < bot(i).key) then
                       bot(i) <= top(i); top(i) <= bot(i);
                   end if;
               end loop;
               state <= ready;
           end if;
       end if;
   end process;
```

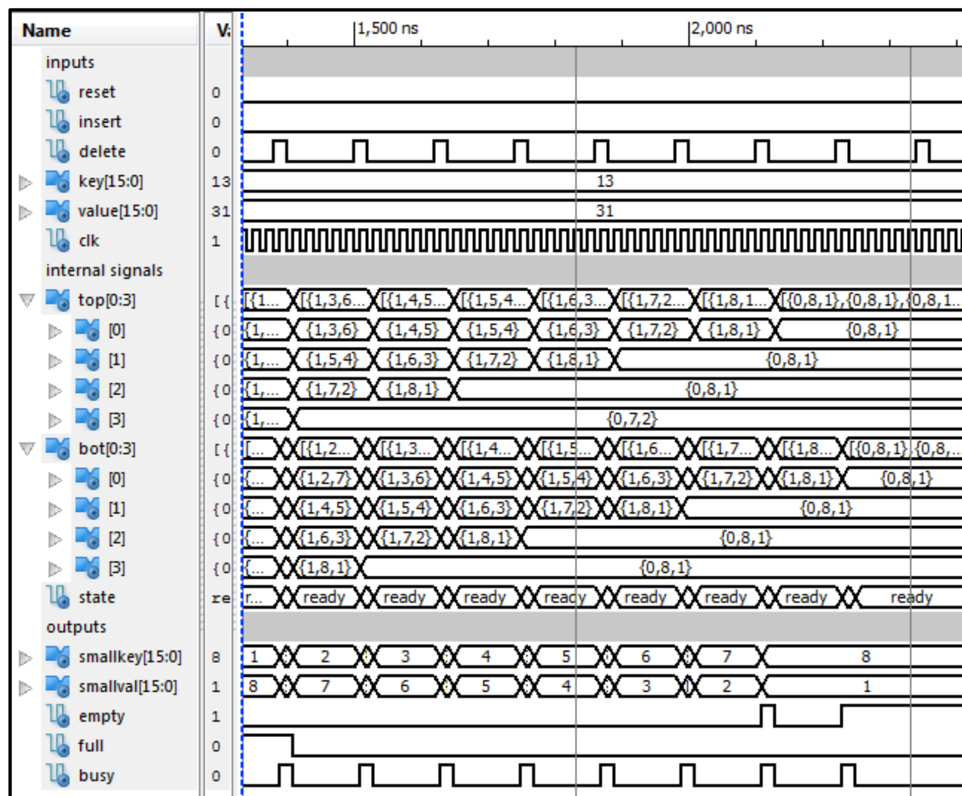
Note that this priority queue does not support simultaneous insert and delete operations. The final portion of the VHDL spec defines the output signals.

```
    smallKey <= bot(0).key; smallValue <= bot(0).value;
    empty <= not bot(0).valid;
    full <= top(rowSize-1).valid;
    busy <= '1' when state = compareAndSwap else '0';
end arch1;
```

We'll finish off this chapter by reviewing a simulation of the priority queue. This first portion shows a series of insertion operations.



Observe how the data in the top and bottom rows is adjusted as new (key,values) pairs arrive. The next section shows a series of deletion operations.



Once again, observe how the top and bottom rows are adjusted as values are removed from the array.

Chapter 13

Small Scale Circuit Optimization

The logic elements used to construct digital circuits (for example, gates or LUTs) represent a limited resource. The number of logic elements on a single chip directly affects its physical size and hence its manufacturing cost. Also, as the size of a chip grows, the manufacturing yield (the percentage of chips that have no manufacturing defects) can drop, further increasing costs. Consequently, we are usually interested in designing circuits that provide the required functionality using as few logic elements as possible.

In the 1970s, digital circuits were largely designed by hand, leading to the development of circuit optimization methods that could be easily applied by human designers. Today, computer-aided design tools do most of the heavy-lifting, and are generally pretty good at doing small-scale circuit optimization. Still, there are occasions when it is useful for designers to be able to optimize small circuits for themselves. An understanding of these techniques also provides some insight into the kinds of decisions that CAD tools must make when they perform automatic circuit optimization. So, in this chapter, we will look at some basic methods for optimizing small circuits. Later in the book, we will look at how the larger-scale decisions made by human designers can effect the cost of circuits, and we'll explore how to make better choices that lead to more efficient circuits.

Our focus in this chapter will be on optimizing combinational circuits, since this is where there is generally the greatest opportunity to reduce circuit complexity. Recally that a typical sequential circuit consists of a set of flip flops defining its state and three combinational circuits, one defining the signals leading to flip flop inputs, one defining the asynchronous output signals and a third defining the synchronous output signals. It is these combinational circuit blocks that are generally the most fruitful targets for optimization.

We will explore two broad classes of methods in this chapter, *algebraic methods* that involve applying the rules of Boolean algebra to obtain simpler logic expressions, and *algorithmic methods* that are largely mechanical and can be easily carried out by computer software.

13.1 Algebraic Methods

In Chapter 2, we explained the correspondence between combinational circuits and logic expressions in Boolean algebra. Any logic expression can be directly converted to a combinational circuit using simple AND gates, OR gates and inverters, and there is a direct correspondence between the gates in the circuit and the operators in the logic expression. Hence, the simplest logic expression for a given function yields the simplest circuit for that function.

To take advantage of this, we need to know how to apply the rules of Boolean algebra to obtain simpler logic equations. We discussed some of the basic rules in Chapter 2. Here is a table that includes all the basic identities that can be used to simplify logic expressions.

1. $X + 0 = X$	2. $X \cdot 1 = X$
3. $X + 1 = 1$	4. $X \cdot 0 = 0$
5. $X + X = X$	6. $X \cdot X = X$
7. $X + X' = 1$	8. $X \cdot X' = 0$
9. $(X')' = X$	
10. $X + Y = Y + X$	11. $X \cdot Y = Y \cdot X$
12. $X + (Y + Z) = (X + Y) + Z$	13. $X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z)$
14. $X(Y + Z) = X \cdot Y + X \cdot Z$	15. $X + (Y \cdot Z) = (X + Y) \cdot (X + Z)$
16. $(X + Y)' = X' \cdot Y'$	17. $(X \cdot Y)' = X' + Y'$

Identities 10 and 11 are commutative laws, while 12 and 13 are associative laws. These should all be familiar from ordinary algebra. Rules 14 and 15 are distributive laws and while the first of these corresponds to the distributive law of ordinary algebra, the second has no counterpart in ordinary algebra. The last two are referred to as DeMorgan's rules and are useful for simplifying expressions containing complement operators. DeMorgan's rules can be extended to handle larger expressions. For example

$$(X + Y + Z)' = ((X + Y) + Z)' = (X + Y)' \cdot Z' \;= X' \cdot Y' \cdot Z'$$

More generally,

$$(X_1 + X_2 + \cdots + X_n)' = X_1' \cdot X_2' \cdots X_n'$$

Similarly

$$(X_1 \cdot X_2 \cdots X_n)' = X_1' + X_2' + \cdots + X_n'$$

There are a couple other handy identities that are worth committing to memory.

$$X + X'Y = X + Y \qquad X + XY = X$$

Note that all of these identities can be applied to complete sub-expressions. So for example

$$(A + B) + A'B'C = (A + B) + (A + B)'C = A + B + C$$

Just as with ordinary algebra, simplifying logic expressions generally involves a certain amount of trial-and-error. If you know the basic identities well,

you can use this knowledge to recognize patterns in expressions to which the identities can be applied.

If you examine the table of identities, you will notice that there is a striking resemblance between the entries in the left column, with the corresponding entries in the right column. This is no accident, but the consequence of a general principle of Boolean algebra known as the *duality principle*. In general, one can obtain the *dual* of any logic expression by replacing AND operations with ORs, replacing ORs with ANDs, and interchanging all 0s and 1s. So for example, the dual of $A + B \cdot C' + 0$ is $A \cdot (B + C') \cdot 1$. When constructing the dual, be careful to preserve the order in which operations are performed. This often requires adding parentheses.

Referring to the table of identities, notice that in each row with two identities, the left sides of the two equations are duals of each other, as are the right sides. Now the duality principle states that if two expressions E_1 and E_2 are equivalent, then their duals are also. That is

$$E_1 = E_2 \Leftrightarrow dual(E_1) = dual(E_2)$$

Consequently, all the equations on the right side of the table follow directly from the equations on the left side.

The duality principle can be useful when attempting to prove that two expressions are equivalent, since sometimes it may be easier to prove the equivalence of the duals than the original expressions. We can also use it to derive new identities. For example, by duality

$$A + A \cdot B = A \Rightarrow A \cdot (A + B) = A$$

and

$$A + A' \cdot B = A + B \Rightarrow A \cdot (A' + B) = A \cdot B$$

We can also use duality to simplify the complementing of large expressions. For example, suppose we wanted to complement the expresson $X \cdot (Y' \cdot Z' + Y \cdot Z)$. We could do this by applying DeMorgan's rule repeatedly, but there is a simpler method using duality. We simply complement all the literals in the expression and then take the dual (interchange ANDs and ORs, interchange 0s and 1s). So for example, complementing the literals in $X \cdot (Y' \cdot Z' + Y \cdot Z)$

gives us $X' \cdot (Y \cdot Z + Y' \cdot Z')$ and taking the dual yields $X' + (Y + Z) \cdot (Y' + Z')$. Thus,

$$(X \cdot (Y' \cdot Z' + Y \cdot Z))' = X' + (Y + Z) \cdot (Y' + Z')$$

We'll finish section by deriving a useful property of Boolean expresssions known as the *consensus theorem*.

$$X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$$

We can prove this by applying the basic identites. In the first step, we apply identities 2 and 7, giving us

$$X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z + (X + X')Y \cdot Z$$

Next we apply identity 14 to get

$$X \cdot Y + X' \cdot Z + (X \cdot Y \cdot Z + X' \cdot Y \cdot Z)$$

Continuing, we apply identites 2, 11 and 14 to get

$$X \cdot Y \cdot (1 + Z) + X' \cdot Z \cdot (Y + 1)$$

and finally, we apply identites 3 and 2 to get

$$X \cdot Y + X' \cdot Z$$

To better understand this argument, notice that what we're doing in the first two steps is splitting the expression $Y \cdot Z$ into two parts $X \cdot Y \cdot Z$ and $X' \cdot Y \cdot Z$). The first part is true only when $X \cdot Y$ is true, while the second part is true only when $X' \cdot Z$ is true. This allows us to discard each of the two parts, since they are redundant.

Of course, we can apply the duality principle to obtain the following variant of the consensus theorem.

$$(X + Y) \cdot (X' + Z) \cdot (Y + Z) = (X + Y) \cdot (X' + Z)$$

13.2 Algorithmic Methods

Algebraic methods are very powerful tools for simplifying logic expressions. With practice, one can become very adept at using them to simplify even very complicated expressions. Computer software can also perform symbolic manipulation of logic expressions. In this section, we'll look at another approach that is often used to simplify expressions in an essentially mechanical fashion. We'll see that for expressions with a small number of variables, we can apply this method by hand. Computer software can also use it for simplifying expressions with a moderate number of variables (say 10-15).

Before we can describe the method in detail, we need to introduce some terminology. Expressions like $AB + C'D + BCD'$ and $ABC + D + BC'$ are examples of *sum-of-products* expressions. More generally, a sum-of-products expression is any expression of the form

$$pterm_1 + pterm_2 + \cdots + pterm_n$$

where *pterm* stands for *product-term* and is a sub-expression of the form

$$X_1 \cdot X_2 \cdots X_k$$

Here, X is either the name of one of the variables in the expression or the complement of one of the variables. The method we will introduce allows one to find the simplest sum-of-products expression for a given expression.

A *minterm* is a product term that includes every variable in the expression. So for example, in the sum-of-products expression $X'Y'Z+X'Z+XY+XYZ$ the first and last product terms are minterms, while the others are not. Any sum-of-products expression can be re-written as a *sum of minterms* by expanding each of the product terms that is not already a minterm into a set of minterms. So for example, in $X'Y'Z + X'Z + XY + XYZ$ the term $X'Z$ can be expanded into the pair of minterms $X'Y'Z$ and $X'YZ$ and the pair XY can be expanded into minterms XYZ' and XYZ. Eliminating redundant minterms, we get the sum-of-minterms expression $X'Y'Z+X'YZ+XYZ' + XYZ$

We can also identify the minterms for an expression by constructing a truth table and finding the rows in the truth table where the expression is equal to 1. Here is a truth table for $X'Y'Z + X'Z + XY + XYZ$.

XYZ	F
000	0
001	1
010	0
011	1
100	0
101	0
110	1
111	1

Note that the first 1 in the table corresponds to the minterm $X'Y'Z$. The other 1s in the table match the remaining minterms. We can assign numbers to minterms, according to their row in a truth table. This gives us a convenient way to specify a logic expression, by listing its minterms by number. So for example, since $X'Y'Z + X'Z + XY + XYZ$ consists of minterms with numbers 1, 3, 6 and 7, it can be specified using the notation

$$\sum_m (1, 3, 6, 7)$$

which is read as "the sum of minterms 1, 3, 6 and 7".

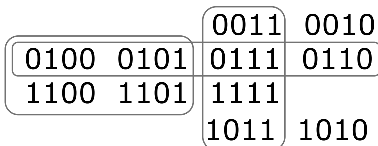
It's often convenient to write a minterm in a short-hand form using 0s and 1s. So for example $X'Y'Z$ can be abbreviated as 001. There is also a corresponding short-hand for product terms that are not minterms; $X'Z$ can be written 0x1 to indicate that the product term is true when $X = 0$ and $Z = 1$, but it doesn't matter what value Y has.

Ok, so now let's proceed to describe the general minimization procedure for sum-of-products expressions. We start by identifying all the minterms in the expression. If we want to simplify the expression, $ABD + A'B + BC'D' + B'CD + B'CD'$, we first identify the minterms 0010, 0011, 0100, 0101, 0110, 0111, 1010, 1100, 1011, 1101 and 1111, or more concisely $\sum_m(2, 3, 4, 5, 6, 7, 10, 1$ The diagram below shows these minterms arranged on the page so that minterms that are adjacent to each other differ in exactly one bit position.

```
                0011  0010
0100  0101  0111  0110
1100  1101  1111
                1011  1010
```

Notice that if you take an adjacent pair, they can be viewed as a product term. For example, the adjacent pair 0100 and 0101 corresponds to the product term 010x. Similarly, 0101 and 1101 corresponds to the product term x101. Similarly, any 2×2 grouping of minterms can be viewed as a product term with two don't cares. For example 0101, 0111, 1101 and 1111 corresponds to the product term x1x1.

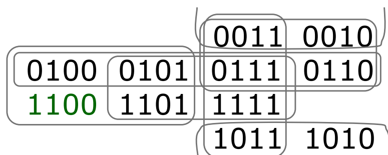
We are now ready for the second step in the minimization procedure, which is to identify the set of *maximal product terms* for the given expression. A maximal product term is one that has the largest possible number of don't care conditions. So, x10x is a maximal product term for our example because if we change either the 0 or the 1 to an x, we will get a product term that includes minterms that are not in our original list. Several maximal product terms are indicated in the diagram below.



Note that one of the maximal product terms corresponds to a 1×4 sub-array of the minterm array, while another corresponds to a 4×1 sub-array. In general, maximal product terms correspond to rectangular subarrays with 2, 4 or 8 cells.

The complete list of maximal product terms is 01xx, x10x, x1x1, xx11, 0x1x, x01x. Observe that most of these consist of contiguous sub-arrays containing four minterms. The product term x01x appears to be an exception. This product term corresponds to the two minterms in the top row together with the two in the bottom row. It turns out that we can think of the top and bottom rows as being adjacent to one another (that is neighboring minterms

in these rows differ in just one variable), so the set of four minterms corresponding to x01x do form a 2×2 subarray. Note that the leftmost and rightmost columns can also be considered adjacent to one another. The diagram below shows all the maximal product terms.



The next step is to identify those product terms that are *essential* to any final solution. We say that a product term is essential if it is the only one covering some minterm. Referring to the diagram, we can see that minterm 1100 is covered by just one product term x10x, so x10x qualifies as essential. Similarly, 1010 is covered by just one product term, x01x, so it is also essential.

Notice that the two essential product terms cover eight of the eleven minterms in the array of minterms. The last step is to select one or more additional product terms that cover the remaining minterms. While up to this point, the process has been completely mechanical and deterministic, now we have to make some choices. There is usually more than one way to cover the remaining minterms. In general, we want to use as few additional product terms as possible, and we prefer to use "larger" product terms rather than smaller ones. In this case, one good choice is the two product terms 01xx and xx11. This gives us a final set of four product terms x10x, x01x, 01xx and xx11. The corresponding sum-of-products expression is

$$BC' + B'C + A'B + CD$$

Note that the minimum sum-of-products expression may not be the simplest expression overall. In this example, we could go one step further by factoring the first and third terms, and the second and fourth. This gives

$$B(A' + C') + C(B' + D)$$

This expression can be implemented using two AND gates, three OR gates and three inverters.

Let's summarize the general method.

1. List all the minterms.
2. Combine minterms to obtain a list of maximal product terms.
3. Identify the essential product terms.
4. Select additional product terms to cover the minterms that have not been covered already.

One way to proceed in the last step is to use a *greedy* strategy. Specifically, we add product terms one at a time, and at each step we select a new product term that covers the largest possible number of not-yet-covered minterms. While this approach is not guaranteed to produce the best possible solution, in practice it generally produces results that are at least very close to the best possible.

Now, there is a nice graphical tool called a *Karnaugh map* that can simplify the process we just went through. We've essentially introduced the Karnaugh map in our presentation of the general algorithm, but we'll now present it a bit more directly. A Karnaugh map for a function of four variables is a square array of 16 cells, each corresponding to a possible minterm. Consider first the left side of the figure below.

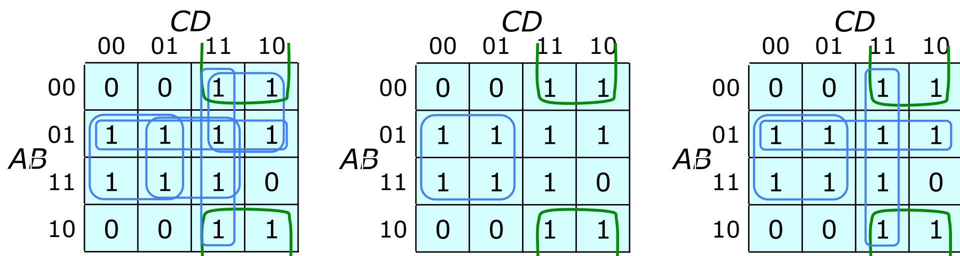
AB \ CD	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

AB \ CD	00	01	11	10
00	0	0	1	1
01	1	1	1	1
11	1	1	1	0
10	0	0	1	1

The labels along the side of the main array list values of two variables, A and B, while the labels along the top list values of two more variables, C and D. Each cell in the array corresponds to a minterm and the numeric values

of these minterms are shown within each cell. The number for a given cell can be obtained by taking the label for the row and the label for the column and interpreting it as a binary number. So for example, the cell labeled 13 is in the row labeled 11 and the column labeled 01. Since 1101 is the binary representation of 13, this cell corresponds to minterm 13. Notice that the labels on the side and top are not listed in numerically increasing order. This is intentional and necessary. By ordering the labels in this way, we obtain an array in which adjacent cells correspond to minterms that differ in exactly one position.

When we use a Karnaugh map to simplify an expression, we fill in the cells with 1s and 0s, placing a 1 in each cell corresponding to a minterm of the expression we are simplifying. The right side of the figure above shows the minterms for the function we just simplified in our previous example. When using a Karnaugh map, we effectively perform the first step in the procedure by filling in the cells. To perform the second step, we identify maximal product terms by circling rectangular sub-arrays that have either 2, 4 or 8 cells, that all contain 1s. These sub-arrays may wrap around from the top row to the bottom or from the left colum to the right. Note that the number of cells must be a power of two, since the number of minterms in any product term is a power of two. This step is shown in the left side of the figure below.



The middle part of the figure shows the selection of essential product terms, while the right side shows a complete set that covers all the 1s in the map. With a little practice, it becomes fairly easy to do steps 2-4 all at once.

Let's try another example. The expression $A(BC + B'C') + B'C + BC'$

is a function of three variables and can be represented by a Karnaugh map with two rows and four columns. To make it easier to fill in the 1s in the map, it helps to put the expression in sum of products form: $ABC + AB'C' + B'C + BC'$.

A \ BC	00	01	11	10
00	0	1	0	1
01	1	1	1	1

In this case, there are just three maximal product terms and they are easy to identify in the map. This gives us the expression $A + B'C + BC'$. which can also be written $A + (B \oplus C)$.

Let's do another four variable example. Consider the function $ABC' + A'CD' + ABC + AB'C'D' + A'BC' + AB'C$. Filling in the 1s in the Karnaugh map gives us the configuration shown below.

AB \ CD	00	01	11	10
00	0	0	0	1
01	1	1	0	1
11	1	1	1	1
10	1	0	1	1

The circled product terms give us the sum-of-products expression $BC' + CD' + AC + AD'$. We can factor this to get $BC' + CD' + A(C + D')$.

We can also use a Karnaugh map to get a *product-of-sums* expression for a function F. To do this, we cover the 0s in the Karnaugh map instead of the 1s. This gives us a sum-of-products expression for F'. By complementing this expression, we get a product-of-sums expression for F. Here's the map

showing the 0-covering for the previous example.

AB \ CD	00	01	11	10
00	0	0	0	1
01	1	1	0	1
11	1	1	1	1
10	1	0	1	1

The resulting expression for F' is $A'B'C' + B'C'D + A'CD$. Complementing this, gives us $F = (A+B+C)(B+C+D')(A+C'+D')$. In this situation, the product-of-sums expression leads to a slightly more expensive circuit than the sum-of-products. However, in other situations, the product-of-sums may be simpler.

Karnaugh maps can also be used to simplify functions with don't care conditions. Recall that a don't care condition is when we don't really care about certain combinations of input variables, typically because in a given application context that combination of inputs will never arise. We can take advantage of don't care conditions to obtain simpler circuits. The way we do this using a Karnaugh map is that we mark the cells in the map that correspond to don't care conditions with an x. When selecting maximal product terms, we are free to choose whether to cover the x's or not. Generally, it makes sense to cover an x's if you can obtain a larger product term (that is, a larger rectangle) by including it. The figure below shows two alternative coverings of a given function.

AB \ CD	00	01	11	10
00	0	1	0	0
01	x	x	x	1
11	1	1	1	x
10	x	0	1	1

AB \ CD	00	01	11	10
00	0	1	0	0
01	x	x	x	1
11	1	1	1	x
10	x	0	1	1

The one on the left yields the expression $A'C'D + B + AC$, while the one on the right yields $A'B'C'D + ABC' + BC + AC$.

We close this section by calling your attention to a computer program that implements the circuit minimization method we have discussed in this section. It can be found on the course web site and consists of two Java classes, *Simplify.java* and a support class *Pterm.java*. To compile them, type

```
javac Pterm.java
javac Simplify.java
```

in a shell window on a computer with an installed Java compiler. To run *Simplify*, type

```
java Simplify 3 0 1 3 5 7
```

or something similar. Here, the first argument to *Simplify* is the number of variables in the expression to be simplified, and the remaining arguments are the minterm numbers. The resulting output from *Simplify* in this case is

```
00x xx1
```

which represents the product terms in the resulting sum-of-products expression. The command

```
java Simplify 4 0 1 5 7 10 14 15
```

yields the output

```
000x 1x10 x111 01x1
```

You are encouraged to try out *Simplify* and run it on several different logic functions of three or four variables. Check the results using a Karnaugh map. If you are interested in getting a deeper understanding of circuit optimization, here are a couple programming exercises that might interest you.

1. *Simplify* currently requires a list of minterms. Modify it so that it will also accept product terms as command-line arguments (where a product term is a bit string with n bit positions and at least one x). This provides a more convenient way to express functions with more than four of five variables.

2. The current version of *Simplify* does not handle don't care conditions in the input. Modify it to accept a command line argument consisting of a single letter X. Additional arguments after the X, specify minterms that correspond to don't care conditions. Modify the optimization procedure to take advantage of the don't cares.

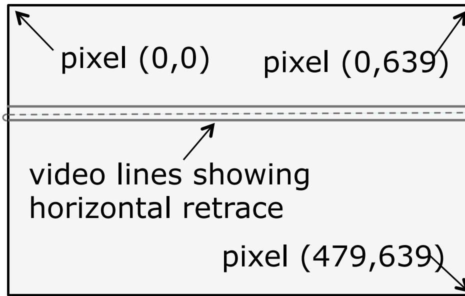
Chapter 14

Still More Design Studies

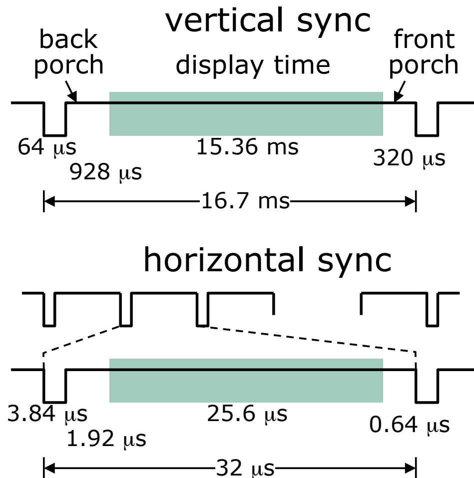
In this chapter, we'll be looking at two circuits, one that provides an interface to a VGA video display, and another that implements the popular minesweeper game, using the VGA display to show the game board.

14.1 VGA Display Circuit

Before we get into the details of the circuit, we need some background about video displays and how video information is generally transferred from a computer to a display using the Video Graphics Array (VGA) standard. The VGA standard was developed for analog Cathode Ray Tube CRT displays in the late 1980s, and even though modern displays no longer use CRT technology, the VGA standard continues to be widely used. CRT displays used an electron beam to activate pixels on the inside surface of a glass display tube. The beam was scanned horizontally across the screen for each row of pixels in the display, with a short *retrace interval* between each row to give the electron beam time to scan back across the screen to the start of the next row of pixels. The standard VGA display has 640 pixels in each display row and 480 visible display lines, giving a total of 307,200 pixels. The figure below illustrates the basic operation.



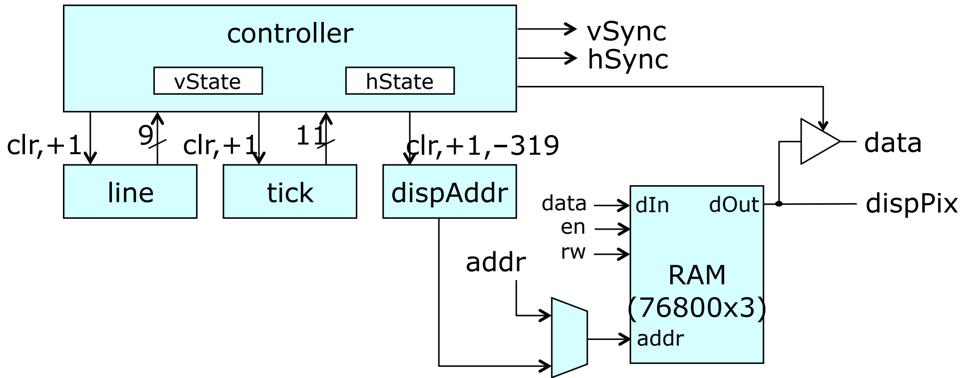
The scanning process is controlled using two synchronization signals: the vertical sync signal consists of a periodic pulse that occurs about every 16.7 ms to give a 60 Hz video update rate for the display. The horizontal sync signal consists of a similar pulse, but with a period of 32 μs. The next figure gives a more detailed view of the two timing signals.



The time periods immediately before and after the sync pulses are referred to as the *front porch* and *back porch* respectively.

Now, a video display interface circuit contains a memory, called a *display buffer*, that holds the information that is to be transferred to the video display. The VGA interface on our prototype board can only display eight

different colors for each display pixel, so the display buffer requires 3 bits per pixel, or $3 \times 640 \times 480$ bits altogether. Unfortunately, the FPGA our prototype board does not have this much memory, so we will implement a half-resolution display with 320×240 pixels. We will still use the entire video display, but each pixel stored in the display buffer will correspond to a 2×2 block of pixels on the actual display. A block diagram of the display interface appears below.

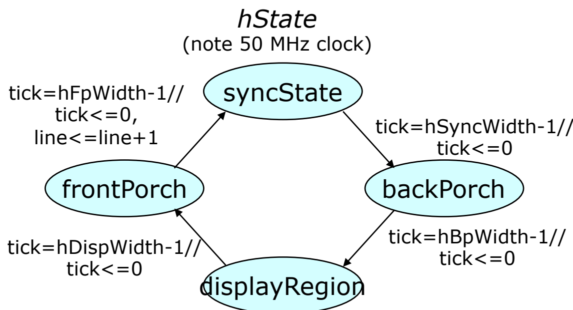


The display buffer is shown at the bottom right. A client circuit can write data to the display buffer using the provided memory control signals. The controller generates the horizontal and vertical sync signals and scans the entire display buffer every 16.7 ms, transferring vido data to the display. It implements two state machines, a vertical state machine with states *syncState*, *backPorch*, *displayRegion* and *frontPortch* and a similar horizontal state machine. It uses two counters to generate the timing signals. The *line* counter keeps track of the current video line, while the *tick* counter keeps track of the current clock tick. These counters are both reset to zero when their state machines enter a new state.

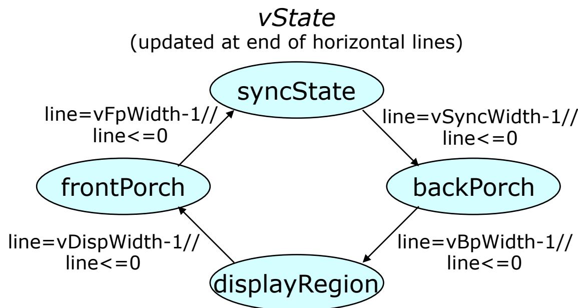
The *dispAddr* register is initialized to zero at the start of a vertical scan (during the vertical sync interval) and is incremented when the horizontal and vertical state machines are both in the *displayRegion* state. Now, to understand the way that *dispAddr* must be updated, we need to think carefully about a couple details. First, the FPGA on our prototype boards uses

a 50 MHz clock, so it has a clock period of 20 ns. This is half the nominal 40 ns required for each display pixel. This implies that the display address should be incremented on every other clock tick, when the state machines are in the *displayRegion* state. Now, since we're implementing a half resolution display, even that is not quite right. Each pixel in the display buffer is used for a 2×2 block of display pixels. This means that during a horizontal scan, we should only increment *dispAddr* after every *four* clock ticks. Moreover, when we get to the end of a line, there are two possible cases, for how we update *dispAddr*. If we're at the end of an even-numbered line, we should reset *dispAddr* to the value it had at the start of the current line, since we need to display the same set of pixels again on the next line. This can be done by subtracting 319 from *dispAddr* (there are 640 display pixels per line, but only 320 pixel values stored in the display buffer). If we're at the end of an odd-numbered line, we can simply increment *dispAddr*.

The transition diagrams for the horizontal state machine is shown below.



The transitions are triggered when the *tick* counter reaches values defined by the constants *hSyncWidth*, *hBpWidth*, *hDispWidth* and *hFpWidth* which define the number of clock ticks in the sync pulse, the back porch, the display region and the front porch. These constants are defined relative to a 50 MHz clock. On the transition from the front porch to the sync state, the vertical *line* counter is incremented. The vertical state machine is also processed on these transitions, using the state transitions defined below.



The structure is similar to the horizontal state machine, but here the transitions occur only at the ends of horizontal lines.

With this background, we're now ready to look at the VHDL specification of the circuit. Let's start with the entity declaration.

```
entity vgaDisplay is port (
    clk, reset: in std_logic;
    -- client-side interface
    en, rw: in std_logic;
    addr: in dbAdr; data: inout pixel;
    -- video outputs
    hSync, vSync: out std_logic;
    dispPix: out pixel);
end vgaDisplay;
```

The client-side interface includes a memory enable and read/write signal, as well as an address input and a bi-directional data signal that can be used to write to or read from the display buffer. The video-side interface consists of just the two sync signals and a three bit display pixel.

The architecture header defines all the data needed for the VGA display circuit.

```
architecture a1 of vgaDisplay is
-- display buffer and related signals
type dbType is array(0 to 240*320) of pixel;
signal dBuf: dbType;
```

```
signal dispAddr: dbAdr; signal dispData: pixel;
-- signals and constants for generating video timing
type stateType is (syncState, frontPorch,
                   displayRegion, backPorch);
signal hState, vState: stateType;
-- horizontal clock tick counter, vertical line counter
signal tick: unsigned(10 downto 0);
signal line: unsigned( 9 downto 0);
-- Constants defining horizontal timing, in 50 MHz clock ticks
constant hSyncWidth: hIndex := to_unsigned( 192,tick'length);..
-- Constants defining vertical timing, in horizontal lines
constant vsyncWidth: vIndex := to_unsigned(  2,line'length);..
```

The *dbufAddr* signal is the actual address input to the display buffer. It can come either from the client interface or the *dispAddr* register used by the display controller. The display buffer memory is synchronous, which means that values read from the memory appear one clock tick after the memory read is requested.

The first part of the architecture body defines the display buffer memory and controls the read and write operations on the memory.

```
begin
    -- display buffer process - dual port memory
    process (clk) begin
        if rising_edge(clk) then
            data <= (others => 'Z');
            if en = '1' then
                if rw = '0' then
                    dBuf(int(addr)) <= data;
                else
                    data <= dBuf(int(addr));
                end if;
            end if;
            dispData <= dBuf(int(dispAddr));
        end if;
```

```
end process;
dispPix <= dispData when vState = displayRegion
                     and hState = displayRegion
           else (others => '0');
```

Notice that there are two different signals used to address the memory. This causes the synthesizer to infer a dual-port memory, with two separate address and data signals. FPGAs are typically capable of synthesizing either single-port or dual-port memory blocks. In this situation, it makes sense to use a dual-port memory so that memory accesses by the client do not interfere with the transfer of pixel data to the display.

The next process implements the controller that generates the display timing signals and the *dispAddr* signal. We start with the horizontal timing.

```
-- generate display timing signals and display address
process (clk) begin
    if rising_edge(clk) then
        if reset = '1' then
            vState <= syncState; hState <= syncState;
            tick <= (others => '0');
            line <= (others => '0');
            dispAddr <= (others => '0');
        else
            -- generate horizontal timing signals
            tick <= tick + 1; -- increment by default
            case hState is
            when syncState =>
                if tick = hSyncWidth-1 then
                    hState <= backPorch;
                    tick <= (others => '0');
                end if;
            when backPorch => ...
            when displayRegion =>
                if tick = hDispWidth-1 then
```

```
                hState <= frontPorch;
                tick <= (others => '0');
            end if;
            if vState = displayRegion then
                if tick(1 downto 0) = "11" then
                    dispAddr <= dispAddr+1;
                end if;
                if tick = hDispWidth-1 and
                   line(0) = '0' then
                    dispAddr <= dispAddr
                        - to_unsigned(319,
                          dispAddr'length);
                end if;
            end if;
```

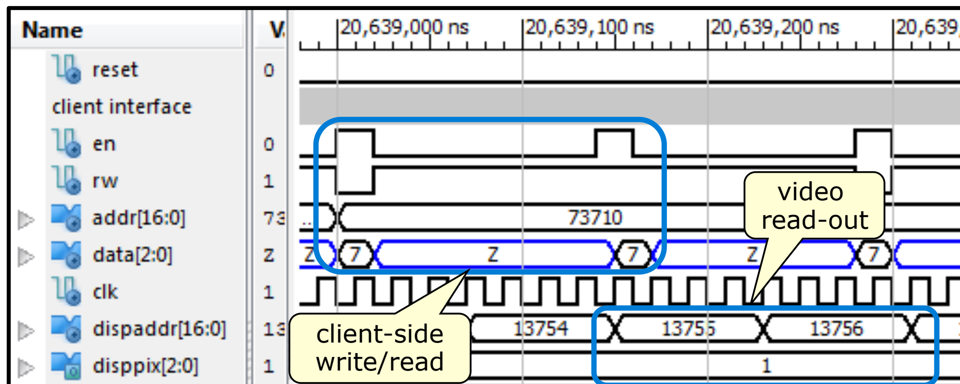
The code for the *syncState* is typical of the timing logic for all of the states. For brevity, we've skipped this code in other cases. Note the updating of the *dispAddr* signal when the horizontal state machine is in the *displayRegion* state. By default, *dispAddr* is incremented on every fourth clock tick (when the two low-order bits of *tick* are equal to "11"), but at the end of every even-numbered line, 319 is subtracted to reset it to the start of the same line.

The final section of code handles the vertical timing at the end of the front porch.

```
        when frontPorch =>
            if tick = hFpWidth-1 then
                hState <= syncState;
                tick <= (others => '0');
                -- generate vertical timing signals
                line <= line + 1;
                case vState is
                when syncState =>
                    dispAddr <= (others => '0');
                    if line = vSyncWidth-1 then
```

```
                                vState <= backPorch;
                                line <= (others=>'0')
                            end if;
                        when backPorch => ..
                        when displayRegion => ..
                        when frontPorch => ..
                        end case;
                    end if;
                end case;
            end if;
        end if;
    end process;
    hSync <= '0' when hState = syncState else '1';
    vSync <= '0' when vState = syncState else '1';
end a1;
```

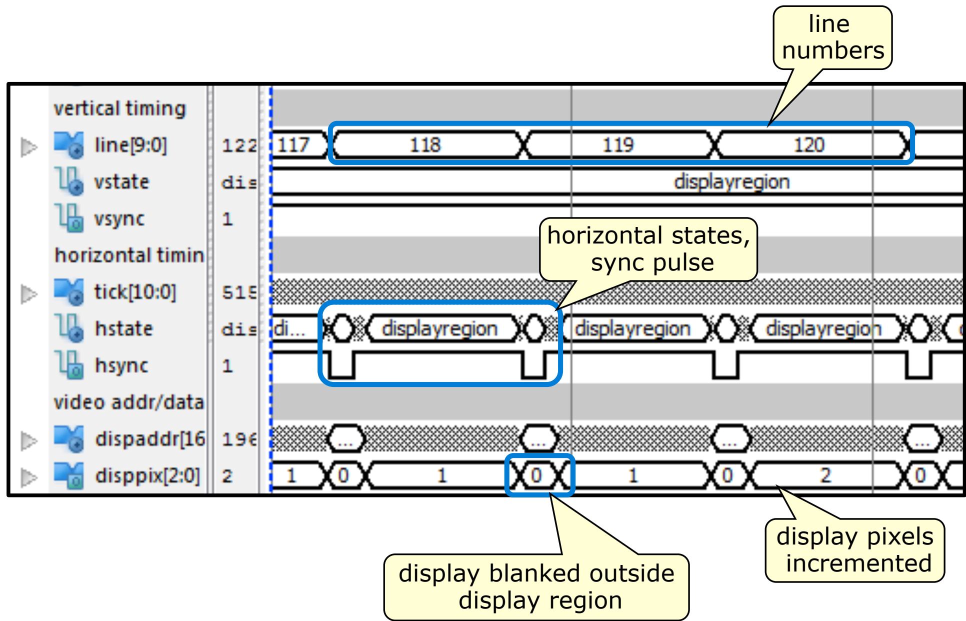
Let's finish up this section by verifying the circuit operation using simulation. The figure below highlights the operation of the display buffer memory.



Note that client-side reads and writes take place indendently of the reads used to retrieve the pixel data so that it can be sent to the video display.

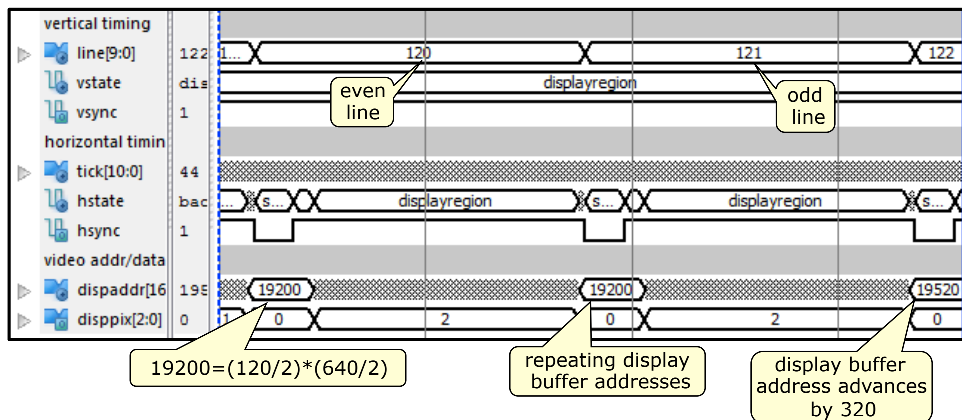
The next portion of the simulation highlights some of the signals gener-

ated by the controller.



Notice how the line numbers are incremented at the end of the horizontal front porch interval. In this simulation, the display contains bands of differing color. In the bottom row, we can observe the color of the displayed pixels changing from one line to the next.

The next part of the simulation shows the updating of the display addresses.



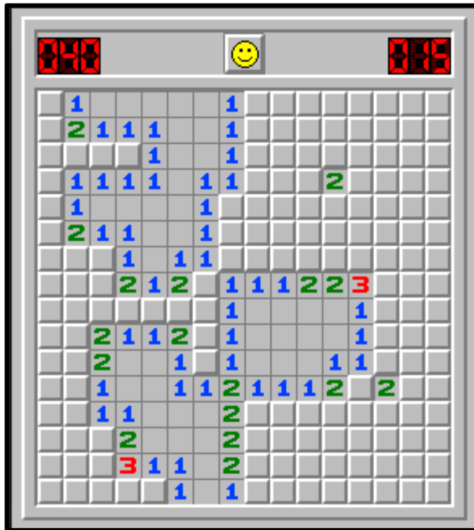
Notice how the display address at the start of the line can be calculated from the line number. Also observe how at the start of an odd line, we use the same address as on the previous even line, while at the end of an odd line, the display address moves onto the next line.

14.2 Mine Sweeper Game

Mine sweeper is a classic video game in which players attempt to locate and avoid "mines" in a rectangular array of squares, while "clearing" all unmined squares. Unmined squares that are adjacent to squares containing mines are labeled with the number of mines in all of their neighboring squares.

In this section, we'll design a version of the mineSweeper game that can be implemented on our prototype board, using the board's VGA output to display the array of squares. To accommodate the limitations of the prototype board, we'll limit ourselves to a game board with a 10×14 array of squares. If each square is 20×20 pixels in size, our game board will occupy the center of our half resolution display, leaving a 20 pixel border on all four sides.

We'll use the knob to control an x register and a y register that determine the position of the "current square" on the board. This square will be highlighted visually on the video display. We'll use a button to uncover



the current square, and a second button to "flag" (or un-flag) the current square, so that we can mark it to avoid accidentally stepping on squares that we know contain mines. A third button will be used to start a new game, where the difficulty of the game is specified using the switches on the board.

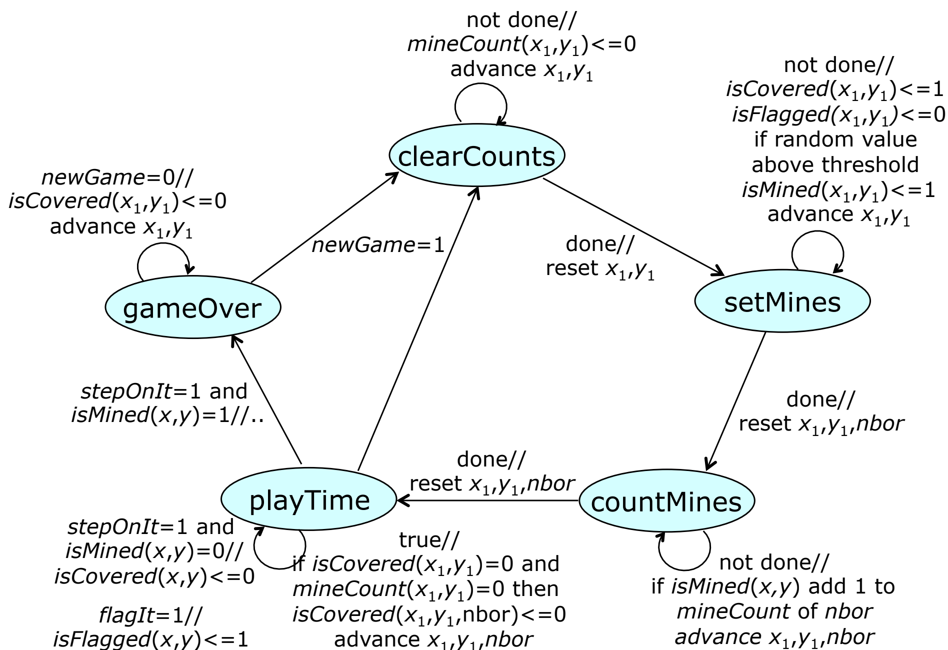
To represent the game state, we'll define three arrays of bits. *IsMined*(x, y) will be true whenever square (x, y) contains a mine, *isCovered*(x, y) will be true whenever square (x, y) is covered and *isFlagged*(x, y) will be true whenever square (x, y) is marked with a flag. In addition, we'll find it useful to have a *mineCount* array that specfies the number of mines adjacent to each square in the array.

There are several things that our circuit needs to do. First, it must initialize the board state at the start of a game. This will involve clearing the mine counts from the previous game, randomly placing mines on the board and setting the *isMined* bits appropriately. It must also set all the *isCovered* bits and clearing the *isFlagged* bits. Finally, it must compute the mine counts. This can be done by scanning the array to find each mined square and then incrementing the mine count values for all neighboring squares.

While the game is being played, the circuit must respond to user input.

If the user requests that the current square be uncovered, the circuit must update the *isCovered* bit, and end the game if the square contains a mine. If the user requests that the current square be flagged, the circuit must update the *isFlagged* bit. In addition, the circuit should continuously uncover squares that are adjacent to existing uncovered squares with a *mineCount* value of 0. While users could clear such squares manually, it's convenient to have such obviously unmined squares cleared automatically, and this is a standard feature of typical implementations of the game.

With these preliminaries out of the way, we can proceed to a state diagram of the mine sweeper circuit.



We'll start with the *clearCounts* state, which is entered when *reset* goes low. In this state, the *mineCount* values are cleared, one at a time. Signals x_1 and y_1 are used to iterate through the game board, so that all *mineCount* values get cleared. In the state diagram, the notation *advance* x_1, y_1 refers

to this adjustment of these index signals. Similarly, the word *done* is used to refer to the point when x_1 and y_1 have progressed through all entries in the array.

In the *setMines* state, the circuit iterates through the game board another time, this time setting the *isCovered* bits, clearing the *isFlagged* bits and conditionally setting the *isMined* bits. In the *countMines* state, the circuit iterates through the board yet another time, but in this case, for each value of x_1 and y_1, it also iterates through the neighboring squares, incrementing their *mineCount* values if square x_1, y_1 contains a mine.

In the *playTime* state, the circuit responds to the user's input. If the *stepOnIt* input is high, the circuit uncovers the square specified by the inputs x and y and ends the game if the square contains a mine. If the *flagIt* input is high, the circuit either flags or unflags square x, y. In addition to responding to user input, the circuit iterates continuously through the game board, looking for uncovered squares that have no neighbors that are mined. It uncovers all neighbors of such squares. In the *gameOver* state, the circuit simply uncovers all squares.

So, let's turn to the actual VHDL circuit specification.

```
entity mineSweeper is port (
    clk, reset: in std_logic;
    -- inputs from user
    xIn, yIn: in nibble;
    newGame, markIt, stepOnIt: in std_logic;
    level: in std_logic_vector(2 downto 0);
    -- video signals
    hSync, vSync: out std_logic; dispVal: out pixel);
end mineSweeper;
```

The `xIn` and `yIn` signals specify the current square on the game board. The `newGame`, `markIt` and `stepOnIt` signals allow the user to control the game. The three bit `level` signal controls the density of mines on the board, with larger values producing more mines.

The next section shows the states and arrays that define the game board.

```
architecture a1 of mineSweeper is
```

```
-- state of game controller
type stateType is (clearCounts,setMines,countMines,
                   playTime,gameOver);
signal state: stateType;
-- arrays of bits that define state of squares on game board
-- range extends beyond gameboard to eliminate boundary cases
type boardBits is array(0 to 15) of std_logic_vector(0 to 11);
signal   isMined: boardBits := (others => (0 to 11 => '0'));
signal isCovered: boardBits := (others => (0 to 11 => '1'));
signal isFlagged: boardBits := (others => (0 to 11 => '0'));
-- mineCount(x)(y)=# of mines in squares that are neighbors of (x
type countColumn is array(0 to 11) of unsigned(2 downto 0);
type countArray is array(0 to 15) of countColumn;
signal mineCount: countArray := (others => (0 to 11 => o"0"));
```

Note that the first array index specifies the x coordinate (or column), while the second specifies the y-coordinate (row). Also, notice that the arrays are all have two extra rows and columns. This makes the code that accesses the arrays somewhat simpler, as it does not need to handle lots of special "boundary cases". So the normal x-coordinate range is 1..14, while the y-coordinate range is 1..10.

Note that the `mineCount` values are just three bits. Strictly speaking, we should use four bit values, since a square could have mines in eight neighboring squares. We're ignoring this case, in order to reduce the circuit complexity enough to allow it to fit in the FPGA used by the prototype board.

Now, we're going to skip ahead to the first part of the main process.

```
begin
    if rising_edge(clk) then
        if reset = '1' then
            -- set initial pseudo-random value
            randBits <= x"357d";
            state <= clearCounts;
            x1 <= x"0"; y1 <= x"0"; nbor <= x"0";
```

```
elsif newGame = '1' then
    state <= clearCounts;
    x1 <= x"0"; y1 <= x"0"; nbor <= x"0
else
    case state is
    when clearCounts =>
        mineCount(int(x1))(int(y1)) <= o"0";
        if x1 /= x"f" then
            x1 <= x1 + 1;
        elsif y1 /= x"b" then
            x1 <= x"0"; y1 <= y1 + 1;
        else
            x1 <= x"1"; y1 <= x"1";
            state <= setMines;
        end if;
    when setMines =>
        -- place mines at random and "cover" them
        randBits <= random(randBits);
        if randBits < ("0" & level & x"000") then
            setMine(x1,y1,'1');
        else setMine(x1,y1,'0');
        end if;
        setCover(x1,y1,'1'); setFlag(x1,y1,'0');
        -- move onto next square
        advance(x1,y1);
        if lastSquare(x1,y1) then
            state <= countMines;
        end if;
```

The code for the *clearCounts* state illustrates the process of iterating through an array, so that elements can be cleared one by one. Note that here, the circuit iterates through the extended range of values (x from 0 to 15, y from 0 to 11).

In the code for the *setMines* we do the same thing, but this time we use a procedure called `advance` which modifies its arguments in much the same

way, but using only the index values that fall within the strict limits of the game board (1 to 14, 1 to 10). The *lastSquare* function returns true when its arguments refer to the last square on the game board. The *setMine*, *setCover* and *setFlag* procedures are simple convenience procedures for setting values in the corresponding bit arrays. The *random* function implements a simple pseudo-random number generator. If the resulting value is less than a threshold determined by the *level* input, the circuit places a mine on the specified square.

The next section shows the code for the *countMines* state.

```
when countMines =>
    addToMineCount(x1,y1,nbor);
    advance(x1,y1,nbor);
    if lastSquare(x1,y1,nbor) then
        state <= playtime;
    end if;
```

The *addToMineCount* procedure increments the specified neighbor of the specified square. Neighbors are identified by small integers, with 0 referring to the neighbor immediately above the designated square, 1 referring to the neighbor above and to its right and successive values continuing around the square in a clockwise fashion. The version of the *advance* procedure used here increments the neighbor index in addition to the x and y coordinates. The version of the *lastSquare* procedure used here returns true if the arguments refer to the last neighbor of the last square.

The next section covers the remaining states.

```
when playTime =>
    if markIt = '1' then
        -- mark/unmark cell if it's covered
        if covered(x,y) then
            if flagged(x,y) then
                setFlag(x,y,'0');
            else
                setFlag(x,y,'1');
```

```
                    end if;
                end if;
            elsif stepOnIt = '1' then
                if covered(x,y) then
                    if mined(x,y) then
                        state <= gameOver;
                    else
                        setCover(x,y,'0');
                    end if;
                end if;
            end if;
            clearNeighbors(x1,y1,nbor);
            advance(x1,y1,nbor);
        when gameOver =>
            setCover(x1,y1,'0'); advance(x1,y1);
        when others =>
        end case;
    end if;
  end if;
end process;
```

The `x` and `y` signals used here are derived from the `xIn` and `yIn` inputs. Their values are guaranteed to be within the legal range of index values for the game board. The convenience functions *covered*, *flagged* and *mined* test the specified bits of the corresponding arrays. They allow us to write

```
if mined(x,y) then ...
```

instead of

```
if isMined(int(x))(int(y)) = '1' then ...
```

which is less cumbersome and makes the intended meaning more clear. The *clearNeighbors* procedure uncovers the specified neighbor of the designated square, if that square is uncovered and has 0 neighbors that are mined. In the *gameOver* state, the circuit iterates through all the squares and uncovers them.

It's worth taking a moment to observe that whenever the circuit iterates through all squares and neighbors on the board, it takes $14 \times 10 \times 8 = 1120$ clock ticks or just over 22 microseconds. There is no reason that the updating has to occur this frequently, but there is also no strong reason to slow down the update rate.

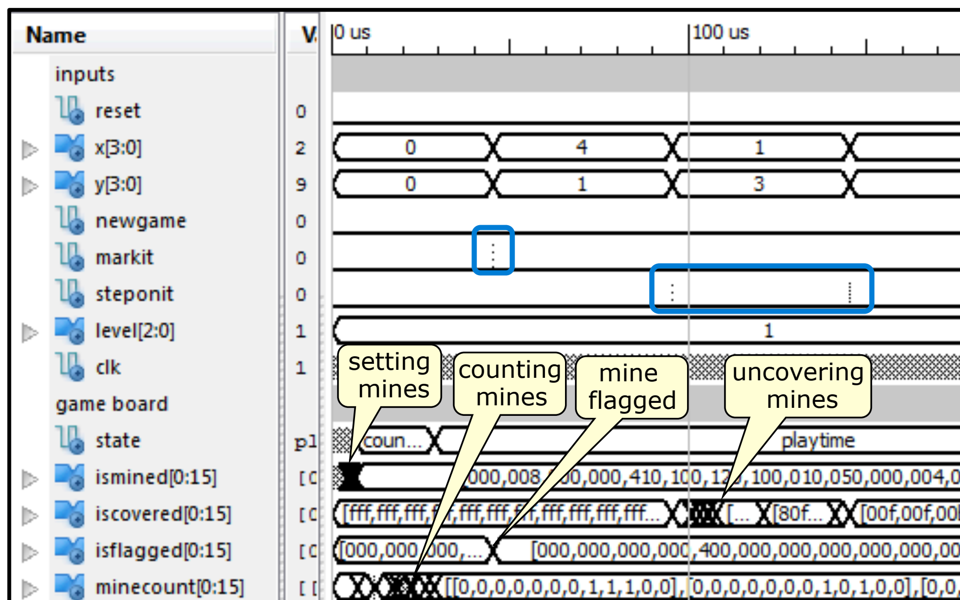
This VHDL spec uses quite a few special functions and procedures. We won't detail them all, but here's a small sample.

```
-- Return true if square contains a mine, else false
begin impure function mined(x,y:nibble) return boolean is begin
    if isMined(int(x))(int(y)) = '1' then return true;
    else return false;
    end if;
end;
-- Return the number of mines contained in neighboring squares
impure function numMines(x,y: nibble) return unsigned is begin
    return mineCount(int(x))(int(y));
end;
-- Advance x and y to the next square on board.
-- Wrap around at end.
procedure advance(signal x,y: inout nibble) is begin
       if x /= x"e" then x <= x + 1;
    elsif y /= x"a" then x <= x"1"; y <= y + 1;
    else                 x <= x"1"; y <= x"1";
    end if;
end;
-- Return next pseudo-random value following r
function random(r: word) return word is begin
    return (r(5) xor r(3) xor r(2) xor r(0)) & r(15 downto 1);
end;
```

The *random* function uses a simple linear feedback shift register to generate a pseudo-random sequence.

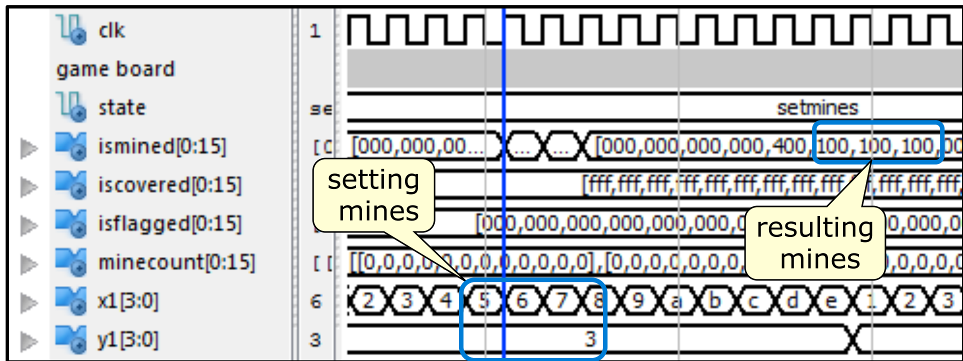
Let's move on to examine a simulation of the *mineSweeper* circuit. We'll start with a very coarse time-scale, so that we can observe some of the large-

scale features of the game.



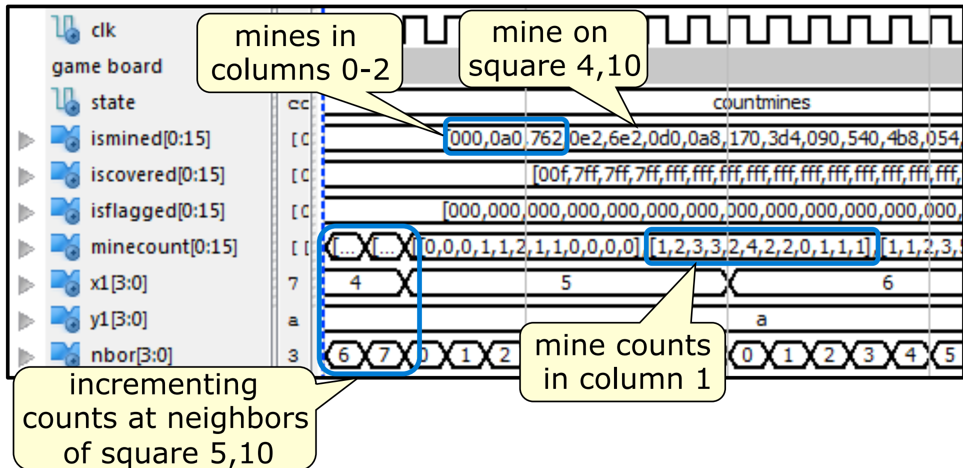
Notice how the *isMined* array is changing at the very start of the simulation, as the board is initialized. Shortly after this point, the *mineCount* array is modified as the circuit counts the mines adjacent to each square. The next feature we can observe is a flag being set, leading to a change in bit (4,1) of the *isFlagged* array. Next, we see the *stepOnIt* signal going high two times, resulting in a number of mines being uncovered.

Let's take a closer look at some of these features.



Here, we are highlighting an interval where mines are set on squares (5,3), (6,3) and (7,3). Note how these bits of the *isMined* array are getting set as a result.

Now let's look at the mine counting process. At the lower left, we can see



that *mineCounts* is changing as we iterate through the last two neighbors of square (4,10). Since the circuit should only increment the mine counts of squares that neighbor a mine, we should check that square (4,10) does indeed contain a mine. We can see that it does by looking at the *isMined* signal. We can verify the mine count values for column 1 of the mine count

array, by comparing the mine counts against the mines in columns 0, 1 and 2.

Next, let's look at the uncovering of "safe squares".

The changes we see to the *isCovered* array occur after the *stepOnIt* signal goes high at square (1,3). By looking at the *isMined* array, we can determine which squares should be uncovered and verify that the results shown at the right end of the *isCovered* waveform are correct. It's worth noting that it can take multiple passes through the game board to uncover all the squares that should be uncovered as a consequence of a single *stepOnIt* event. Here, we are showing the state after the last square gets uncovered. (Thought exercise: how many passes through the game board are needed to clear the squares after a single *stepOnIt* event, in the worst-case?)

At this point, we're essentially done with all the circuitry needed to implement the core game logic, but we still need to design the circuit that displays the game state on an external VGA monitor. We'll use our existing *vgaDisplay* component, but we need to augment this with some additional circuitry to copy pre-defined patterns to designated locations in its display buffer. We'll build a separate component to copy a single specified pattern to the display. The code segment below shows how that component is used.

```
-- process to iterate through cells for copying to display
 -- advance (x2,y2) through range of values periodically
```

```
    process(clk) begin
        if rising_edge(clk) then
            if reset = '1' then
                timer <= (others => '0');
                x2 <= x"1"; y2 <= x"1";
            else
                if timer = (timer'range => '0') then
                    advance(x2,y2);
                end if;
                timer <= timer + 1;
            end if;
        end if;
    end process;

    -- copy pattern for x2,y2 to its position on display
    cp: copyPattern port map(clk,reset,startCopy,highlight,pat
                             x2,y2,busy,hSync,vSync,dispVal);
    startCopy <= '1' when timer = (timer'range=>'0') else '0';
    highlight <= '1' when x2 = x and y2 = y else '0';
    pat <= pattern(x2,y2);
```

The *copyPattern* component does the heavy lifting of copying a specified pattern to the display buffer. The process simply advances the signals x_2, y_2 through all the coordinates on the board, under the control of a timer that limits the update rate. A new copy operation is initiated every time the timer is equal to zero, and the *highlight* signal is raised whenever the x_2, y_2 coordinates match the player's current square. This signal directs the *copyPattern* block to modify the appearance of the pattern for this square, so that players can identify their current location on the board by looking at the video display.

The *pat* signal is determined by the *pattern* function, which returns an integer index for the pattern that should be displayed for square x_1, y_2.

```
-- Return the appropriate pattern number for square x,y
impure function pattern(x,y: nibble) return nibble is begin
```

```
    if (not covered(x,y)) and (not mined(x,y)) then
        return "0" & std_logic_vector(numMines(x,y));
    elsif covered(x,y) and (not flagged(x,y)) then
        return x"9";
    elsif covered(x,y) and flagged(x,y) then
        return x"a";
    elsif (not covered(x,y)) and mined(x,y) then
        return x"b";
    else
        return x"0";
    end if;
end;
```

Pattern number 0 depicts an uncovered blank square. For i in the range 1-8, pattern number i depicts an uncovered square labeled by a mine count. Pattern number 9 depicts a covered square, pattern number 10 depicts a covered square with a flag and pattern number 11 depicts an uncovered square with a mine.

So all that remains is the *copyPattern* component. Here is the entity declaration.

```
entity copyPattern is port(
    clk, reset : in  std_logic;
    -- client side interface
    start, highlight: in std_logic;
    pattern: in nibble;
    x, y: in nibble;
    busy: out std_logic;
    -- interface to external display
    hSync, vSync: out std_logic;
    dispVal: out pixel);
end copyPattern;
```

Next, we have the portion of the architecture header that defines the graphical patterns.

```
architecture a1 of copyPattern is
component vgaDisplay ...end component;
subtype patRow is std_logic_vector(3*20-1 downto 0);
type patArray is array(0 to 12*20-1) of patRow;
constant patMem: patArray := (
    -- uncovered, blank square
    o"22222222222222222220",
    ...
    o"22222222222222222220",
    o"00000000000000000000",

    -- uncovered square labeled 1
    o"22222222222222222220",
    o"22222222222222222220",
    o"22222222227222222220",
    o"22222222277222222220",
    o"22222222777222222220",
    o"22222227777222222220",
    o"22222222277222222220",
    o"22222222277222222220",
    o"22222222277222222220",
    o"22222222277222222220",
    o"22222222277222222220",
    o"22222222277222222220",
    o"22222222277222222220",
    o"22222222277222222220",
    o"22222222277222222220",
    o"22222227777772222220",
    o"22222227777772222220",
    o"22222222222222222220",
    o"22222222222222222220",
    o"00000000000000000000",

    -- uncovered square labeled 2
```

```
...
-- uncovered square labeled 3
...
);
```

Each pattern is a 20×20 block of pixels. They are all stored in a constant array, where each row of the array holds 60 bits, enough for 20 pixels. So each block of 20 consecutive rows defines one pattern. Since we have a total of 13 distinct patterns, the array contains 13×20 rows. Only one of the patterns (the one for an uncovered square containing a 1) is shown explicitly. Note that octal strings are used to represent the pixel data, so each octal digit corresponds to one pixel. The pixel value 2 designates green, while 0 designates black and 7 designates white, Consequently, uncovered squares have a green background with a black border along the bottom and right edges, while the numeric labels are shown in white.

Let's move onto the body of the architecture.

```
vga: vgaDisplay port map(clk,reset,en,rw,dispAdr,curPix,
                         hSync,vSync,dispVal);
curPix <= patPix when hiLite = '0' else not patPix;
process(clk)
... -- function and procedure definitions
begin
    if rising_edge(clk) then
        en <= '0'; rw <= '0'; -- default values
        if reset = '1' then
            tick <= (others => '1'); hiLite <= '0';
        else
            tick <= tick + 1; -- increment by default
            if start='1' and tick=(tick'range=>'1') then
                initAdr(x, y, pattern,
                        dispAdr, patAdr, patOffset);
                hiLite <= highLight;
            elsif tick < to_unsigned(4*400,11) then
                -- an update is in progress
```

```
                    -- each step involves copying a pixel from
                    -- the pattern to the display buffer;
                    -- we allow four clock ticks per step
                    if tick(1 downto 0) = "00" then
                        -- first read from pattern memory
                        patPix <= unsigned(
                            patMem(int(patAdr))(
                                   int(patOffset) downto
                                         int(patOffset-2)));
                        -- write display buf during next tick
                        en <= '1';
                    elsif tick(1 downto 0) = "11" then
                        advanceAdr(dispAdr, patAdr, patOffset);
                    end if;
                else -- returns circuit to "ready" state
                    tick <= (others => '1');
                end if;
            end if;
        end if;
    end process;
```

The *vgaDisplay* module is instantiated at the top of this section. Its *dispAdr* input specifies the display buffer address where the next pixel is to be written. The *curPix* input specifies the pixel data to be written. Observe that the bits of *curPix* are inverted for highlighted squares to give them a distinctive appearance.

The timing for the main process is controlled by the `tick` signal. This is set to all 1s when the circuit is idle, and is incremented on each clock tick while it is performing a copy operation. At the start of the copy operation, the display address *dispAdr* is initialized along with the signals *patAdr* and *patOffset* that specify a row in the pattern memory and the offset of a pixel within that row. This initialization is done by the *initAdr* procedure. The value of the *highlight* input is also saved in a register at this time. Each step in a copy operation involves reading one pixel from the pattern memory and writing that pixel to the display buffer. The circuit allows four clock

ticks for each such step. When the low-order two bits of *tick* are equal to zero, it reads a pixel from the pattern memory and sets the display buffer enable high, so that during the next clock tick, the value read from the pixel memory is written to the display buffer. When the low order two bits of *tick* are both high, the *advancdAdr* procedure adjusts the display buffer address, the pattern address and the pattern offset, as needed to handle the next pixel.

The *initAdr* procedure appears below.

```
-- initialize address signals used to access the
-- pattern memory and the display buffer
procedure initAdr(x, y, pat: in nibble;
                  signal dAdr: out dbAdr;
                  signal pAdr, pOffset: out byte) is
variable row, col: unsigned(2*dAdr'length-1 downto 0);
begin
    -- first display address of square x,y is 20*320*y+20*x
    -- since 320 pixels per display row and each
    -- pattern extends over 20 rows
    row := to_unsigned(20*320,dAdr'length)
                   * pad(unsigned(y),dAdr'length);
    col := to_unsigned(20,dAdr'length)
                   * pad(unsigned(x),dAdr'length);
    dAdr <= row(dAdr'high downto 0)
            + col(dAdr'high downto 0);
    -- first pattern address is 20*(the pattern number)
    -- offset starts at 59 and goes down to 2
    pAdr <= pad(slv(20,8) * pad(pattern,8),8);
    pOffset <= slv(59,8);
end;
```

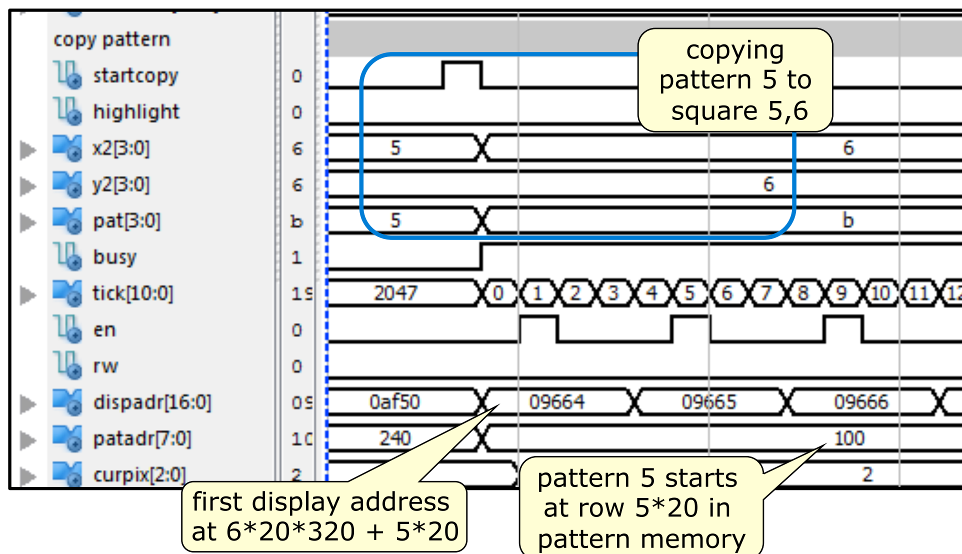
The *row* and *col* variables are defined to break the calculation of the initial display buffer address into smaller parts. The *pad* function used here, is used to adjust the length of a given signal. In the assignments to *row* and *col*, it creates signals with the same values as *x* and *y* but enough bits to be

compatible with a display buffer address. The VHDL multiplication operator produces a product that has twice as many bits as its two operands. This is why *row* and *col* have been defined to have 2**dAdr'length* bits. The assignment to *dAdr* discards the high-order bits of *row* and *col* before adding them together. The calculation of the pattern address is similar, although a bit simpler. Notice here that we are using *pad* to truncate the signal produced by the multiplication operation.

Finally, we have the *advanceAdr* procedure, used to update the display address plus the pattern address and offset signals.

```
-- Advance address signals used to access
-- the pattern memory and display buffer
procedure advanceAdr(signal dAdr: inout dbAdr;
                     signal pAdr, pOffset: inout byte) is
begin
    if pOffset = 2 then -- reached end of pattern row
        pAdr <= pAdr + 1; pOffset <= slv(59,8);
        dAdr <= dAdr + to_unsigned(320-19,dAdr'length);
    else -- continue along the row
        pOffset <= pOffset - 3; dAdr <= dAdr + 1;
    end if;
end;
```

All that's left is to verify the operation of the *copyPattern* block from the simulation.



This shows the start of a copy operation that copies pattern number 5 to the position on the display used for square 5,6. The first display address is $6 \times 20 \times 320 + 5 \times 20 = 38,500$ or 09664 in hex. The first pattern address is $5 \times 20 = 100$

Chapter 15

Implementing Digital Circuit Elements

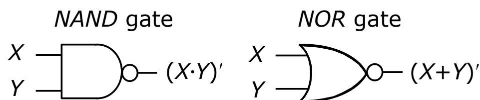
In this chapter, we're going to take a closer look at the basic elements used to build digital circuits. We'll discuss how they can be implemented using lower level components (transistors) and how the underlying implementation affects delays that occur in real circuits.

15.1 Gates and Transistors

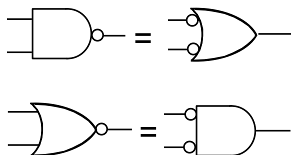
Digital circuits can be implemented using a variety of specific circuit technologies. By far the most commonly used technology is CMOS, which stands for *complementary metal oxide semiconductor*. In this section, we're going to look at how gates are implemented in CMOS circuits using more fundamental electronic components called transistors. We will find that in CMOS, the standard AND gate actually requires more transistors than the so-called NAND gate, which is logically, an AND gate with an inverted output. So, before we talk about transistors, it's worth taking a little time to discuss NAND gates and NOR gates.

The figure below shows the symbols for the NAND and NOR gates

and the logic functions that they define. The inversion bubbles on the

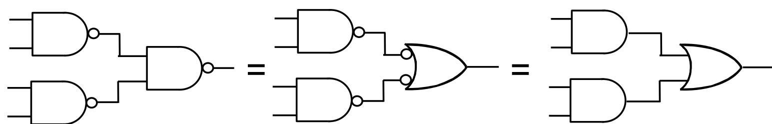


outputs of these symbols just indicate that the output signals are complements of what we would expect for the conventional gates. We can always construct a standard AND gate by connecting the output of a NAND gate to an inverter, but since NAND gates use fewer transistors than AND gates, it can be more efficient to construct circuits using NAND (and NOR) gates than the conventional gates. Unfortunately, circuit diagrams containing NANDs and NORs can be difficult to understand Fortunately, there are a couple of handy tricks that can be used to make it a bit easier. The key observation is contained in the figure below.



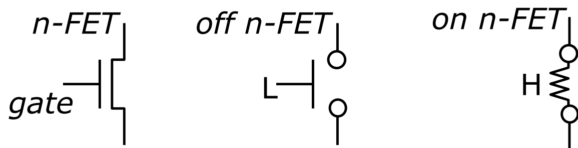
This diagram shows two equivalent ways to draw a NAND gate in a circuit (and the same for a NOR gate). We can draw a NAND in the conventional way, as an AND with an inverted output, or we can draw it as an OR with inversion bubbles on both inputs. The equivalence of these two representations follows directly from DeMorgan's law: $(X \cdot Y)' = X' + Y'$.

Now, these alternate representations of the NAND and NOR gates allow us to draw a circuit containing NANDs and NORs in a way that's easier to understand. For example, consider the circuit on the left side of the diagram below.



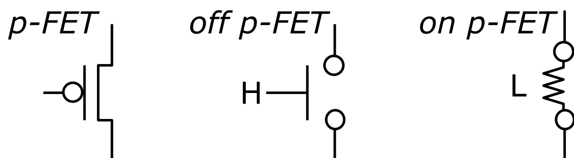
In the middle diagram, we have re-drawn the rightmost NAND gate in its alternate form. Now, since the wires connecting the gates have inversion bubbles on both ends, they are equivalent to wires with no-inversion bubbles. That is, the circuit on the left is logically equivalent to the one on the right. By using the alternate forms for NAND and NOR gates, we can make circuit diagrams using them, much easier to understand.

Now, let's turn to the subject of transistors. CMOS circuit technology uses a particular type of transistor called the *Field Effect Transistor* or FET. There are two types of FETs, *n-type* and *p-type*, of more concisely *n*-FET and *p*-FET. The left part of the diagram below shows the symbol for an *n*-FET.



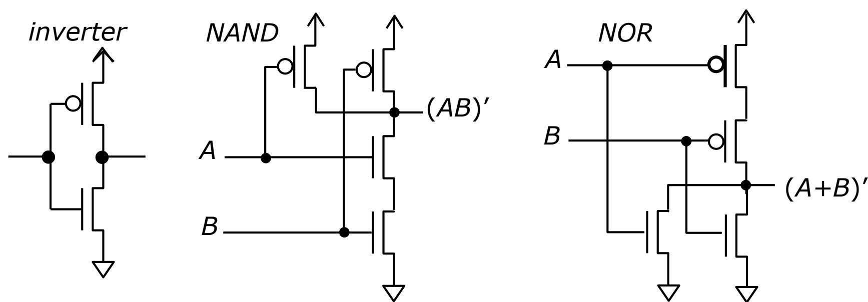
An *n*-FET has three terminals. The control terminal on the left is referred to as the *gate* of the transistor (note we are using the word "gate" here in a different way, than we have up to now), while the other two terminals are referred as the *source* and the *drain*. When the voltage on the gate input is low, the transistor acts like an open switch, that prevents current from flowing between the source and the drain. This is illustrated by the middle part of the diagram. On the other hand, when the voltage on the gate input is high, the transistor acts like a closed switch that connects the source and drain together, although with some resistance to current flow, as indicated by the resistor symbol in the right part of the diagram. (Readers who have studied transistors in electronics courses will recognize that this discussion over-simplifies the operation of the transistor, but for our purposes, this simplified description will suffice.)

The *p*-FET is similar to the *n*-FET and is illustrated in the diagram below.



Note that the *p*-FET symbol includes an inversion bubble on its gate input. This indicates that the the *p*-FET turns on when the voltage on the gate is *low* and turns off when the voltage on the gate is high. That is, the *p*-FET behaves in a way that is complementary to the *n*-FET.

n-FETs and *p*-FETs are usually used in pairs, with one transistor in each pair in the on-state whenever the other is in the off-state. This is illustrated in the figure below, which shows the typical CMOS implementation of an inverter, a NAND gate and a NOR gate.



Let's start with the inverter. Notice that when the inverter input is high, the *n*-FET will turn on, while the *p*-FET will turn off. This means that the output is essentially connected to the ground voltage, which forces the output voltage to be low. Similarly, when the inverter input is low, the *n*-FET will turn off and the *p*-FET will turn on, connecting the inverter output to the power supply voltage and forcing the output voltage high. So we can see that this CMOS circuit does behave like a logical inverter.

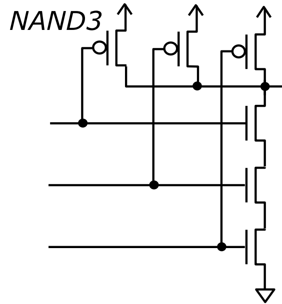
The NAND gate is a bit more complicated. First, let's consider the case when the two inputs are both high. This turns on both of the *n*-FETs, while

it turns off both of the *p*-FETs. As a result, the output is pulled low, as we expect for a logical NAND gate. On the other hand, if either (or both) of the inputs are high, then one (or both) of the *n*-FETs will be turned off and one (or both) of the *p*-FETs will be turned on. As a result, the output will be connected to the power supply voltage, and the output will be high. Again, just what we expect for a logical NAND gate. Note how the *p*-FETs in the NAND gate are connected in parallel to each other, while the *n*-FETs are connected in series. This choice is essential to ensuring that the circuit functions as a NAND gate.

The NOR gate is similar to the NAND. In this case, the *p*-FETs are connected in series, while the *n*-FETs are connected in parallel. Note that when either input is high, the output is pulled low, but if both inputs are low, the output is pulled high.

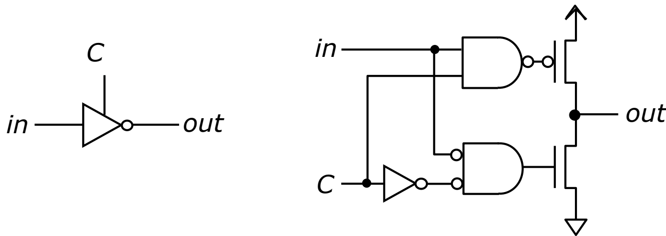
It's natural to ask if we could directly implement AND and OR gates using transistors. For example, why not just take the NOR circuit but use *n*-FETs in the top part of the diagram and *p*-FETs in the bottom part. This would appear to directly implement the desired AND functionality and would save us the mental hassle of dealing with gates that have inverted outputs. Unfortunately, more subtle aspects of the *n*-FET and *p*-FET components prevent us from building reliable gates in this way. Because of the underlying electrical properties of the transistors, *n*-FETs generally are only used to connect gate outputs to the ground voltage, while *p*-FETs are only used to connect gate outputs to the power supply. Consequently, it's common practice to refer to the *n*-FETs in a gate as *pulldown transistors* and to refer to the *p*-FETs as *pullups*.

We can generalize the circuits for the NAND and NOR to produce gates with more than two inputs, as illustrated below.



Here, we simply extended the previous circuit, by adding another *p*-FET in parallel with the original two, and another *n*-FET in series with the original three. Note, we could also implement a three input NAND using a NAND, a NOR and an inverter, but this circuit would require ten transistors, while the circuit shown above uses just six.

Now that we know about transistors, we can take a closer look at the tristate buffer, to better understand what it means when a tristate buffer is in the high impedance state. The figure below shows an implementation of a tristate buffer using a NAND gate, a NOR gate, an inverter and a pair of transistors.

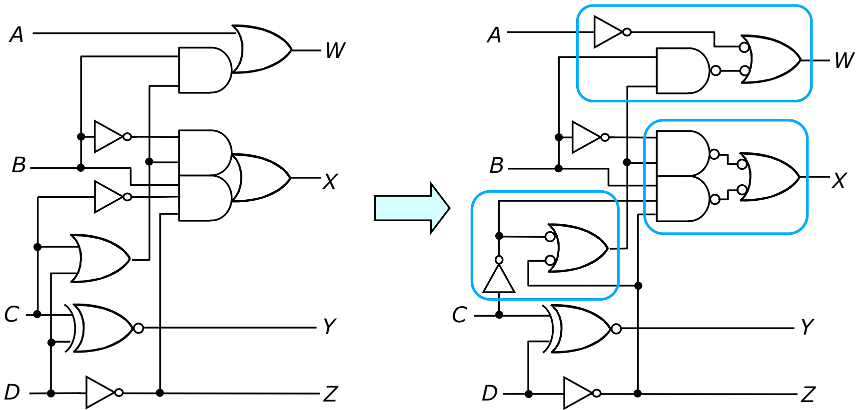


When the control input is high, both gates respond to changes in the data input. Specifically, when the data input is low, the pulldown is turned on and the pullup is turned off (making the output low). Similarly, when the data input is high, the pullup is turned on and the pulldown is turned off (making the output high). In both cases, the data output is equal to the data input.

When the control intput is low, both the pullup and pulldown are turned

off, effectively disconnecting the tristate buffer from the output wire. This is really what is what we mean, when we say that a tristate buffer is in the "high impedance" state. If several tristate buffers have their outputs connected to the same wire, they can take turns using the wire, so long as at most one has its control input high, at any one time. The others, by disconnecting from the output wire, allow the active tristate to have complete control over whether the output wire is high or low.

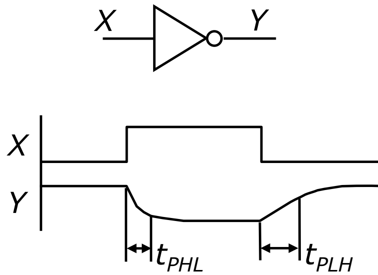
Let's wrap up this section by looking at an example of how we can reduce the number of transistors used by a circuit by replacing conventional gates with NANDs and NORs. The lefthand diagram below shows a combinational circuit with four inputs and four outputs. The righthand diagram shows an equivalent circuit in which NANDs and NORs have been substituted for some of the conventional gates.



If AND gates are implemented using a NAND followed by an inverter (as is required, in CMOS) and the OR gates are implemented similarly, the lefthand circuit requires 60 transistors, while the righthand circuit requires just 50, a significant reduction. Fortunately, with modern CAD tools, it's rarely necessary for us to perform such optimizations manually. Circut synthesizers take the relative costs of different types of gates into account, and attempt to select components that minimize the overall circuit cost.

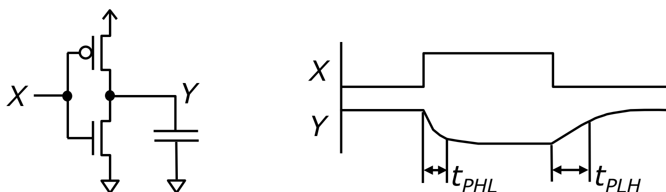
15.2 Delays in Circuits

Up to now, we've largely treated gates as ideal logic devices with zero delay. That is, when a change at a gate input causes an output to change, that output change occurs instantaneously. In reality, physical implementations of gates do not work that way. To change the output of a gate, we actually need to change the voltage at the output of the circuit that implements the gate, and changing a voltage takes some time. In human terms, the magnitude of the resulting circuit delays is infitesimal, but in the world of integrated circuits these delays are what ultimately limit the performance of our computers, cell phones and other devices. The figure shown below illustrates the delays involved.

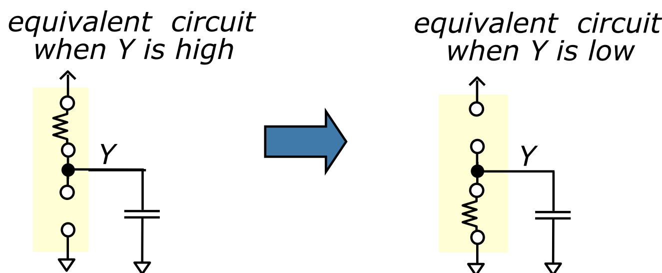


When the inverter input X goes high, the output Y changes from high to low, but the signal takes some time to move from the high voltage to the low voltage. We refer to this delay as the *propagation delay* for the gate, and it's typically measured from the time the input changes until the output gets "most of the way" to its ultimate voltage level. Because n-FETs and p-FETs have some different electrical characteristics, the propagation delays are usually different for rising transitions and falling transitions. The notation t_{PHL} denotes the propagation delay for a high-to-low transition (at the gate output), while t_{PLH} denotes the propagation delay for a low-to-high transition.

To understand why these delays occur, let's a closer look at the operation of the inverter, using the diagram below.
Here, we've shown the circuit that implements the inverter. Let's assume



that the output of the inverter is connected to the input of some other gate, which is not shown. From an electrical perspective, the input of the other gate operates like a small capacitor. Before the inverter input X goes high, the inverter output Y is high, and the capacitor stores a positive charge. When X goes high, the n-FET in the inverter turns on, while the p-FET turns off. This allows the positive charge on the capacitor to flow through the n-FET to ground. This transfer of charge from the capacitor allows the output voltage to drop. This is illustrated in the diagram below which shows the equivalent circuit representation of the inverter, both before and after the input changes.



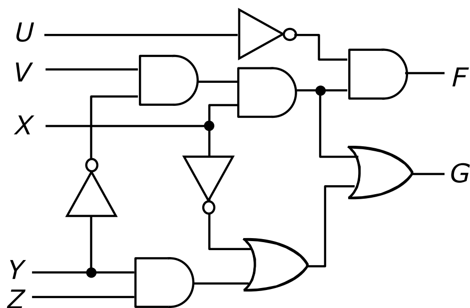
Observe that because an on-transistor has some resistance to current flow, there is a limit to the rate at which the charge on the capacitor can be transferred through the transistor. This is what ultimately causes the delay in the change to the output voltage.

We can now make some other observations based on what we have learned. First, notice that a NAND gate has its two pulldown transistors arranged in series, and since their resistances add together when they are both in the on state, it can take longer for the output of a NAND gate to go from high-to-

low than it does for it to go from low-to-high. Similarly, for a NOR gate, a low-to-high transition can take longer than a high-to-low transition. Also, because a three input NAND has three pulldowns in series, its high-to-low transitions will take longer than those for a two input NAND.

The time it takes for signal transitions to occur also depends on the number of other gates that a given gate's output is connected to. If the output of an inverter is connected to several other gates, those other gates each add to the effective capacitance at the inverter's output. Since a larger capacitance can store more charge, it takes more time to transfer the charge from the capacitance through the inverter's pulldown (or pullup) when the inverter output changes. The number of other gates that a given gate output connects to is referred to as the *fanout* of that gate. In general, the larger the fanout, the slower the signal transitions.

What does this all mean for the performance of larger circuits constructed using gates? Let's consider the example circuit shown below.



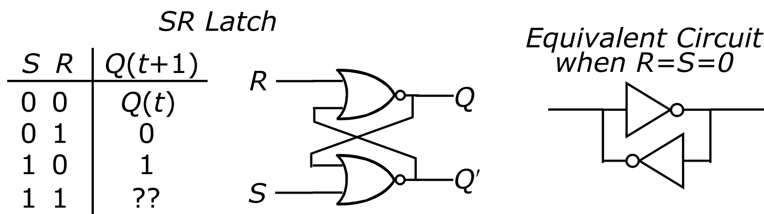
Each gate in this circuit contributes to the delay experienced by signals passing from inputs to outputs. So, for example, a signal passing from input U to output F passes through two gates, each contributing some delay. A signal from input Z, going to output G passes through three gates, so we can expect a somewhat larger delay along this circuit path. In general, the more gates that a signal must pass through to get from an input to an output, the larger the delay will be. If we assume that each gate contributes the same amount of delay, then we can estimate the delay for a circuit simply by counting the number of gates along all paths from circuit inputs to circuit outputs. For

the example circuit, we can observe that the *worst-case path* (from input Y to output F) passes through four gates. We often use the worst-case delay through a circuit as a general way to characterize the circuit's performance.

To accurately determine delays in a circuit, it's necessary to consider the electrical details of all the individual gates, as well as the fanouts of each gate within the larger circuit. Fortunately, modern CAD tools can automate the process of estimating circuit delays, using fairly detailed models of individual circuit elements. Still, even though delay estimation is something that we're rarely called upon to do manually, it is important to understand how these delays arise and the factors that influence them.

15.3 Latches and Flip Flops

Recall that in an earlier chapter, we discussed how a D flip flop could be implemented using simpler storage elements called D latches. In this section, we're going to look at how latches are implemented using gates. We're going to start with a particularly simple type of latch called the *set-reset latch* or more concisely, the SR-latch. As the name implies, it has two inputs, a *set* input S, and a *reset* input, R. When S is high and R is low, the latch output becomes high. When R is high and S is low, the latch output becomes low. When both inputs are low, the output of the latch does not change. If both inputs are high, the output of the latch is not defined, although circuits implementing a latch do generally have a well-defined behavior in this case. The figure below shows a table summarizing the behavior of the latch and an implementation using a pair of NOR gates.



S R	Q(t+1)
0 0	Q(t)
0 1	0
1 0	1
1 1	??

In the table at the left, the notation $Q(t)$ refers to the output of the latch

at time t. So, when we say that $Q(t+1) = Q(t)$ when $S = R = 0$, we're really just saying that in this case, the latch remains in whatever state it was in before both inputs became low. Notice that the latch has been defined with two outputs Q and Q', where Q is what we normally think of as the state of the latch and Q' is simply the complement of Q. Since we get the complement "for free" in this circuit, it's natural to make it available as a second output of the latch. Also, notice that since the latch is constructed using gates, the latch outputs do not change instantaneously when the inputs do. The propagation delay of a latch is defined as the time between an input change and the corresponding output change. As with gates, there may be different propagation delays for high-to-low vs low-to-high transitions.

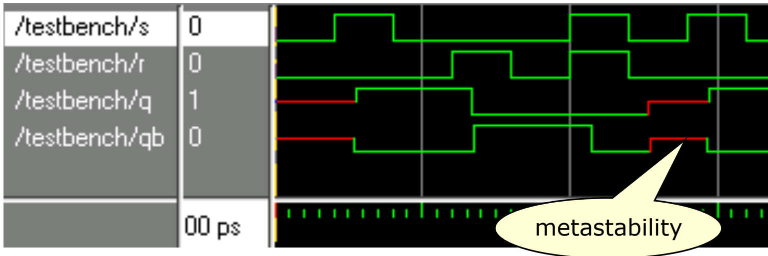
The righthand part of the figure shows an equivalent circuit for the latch when the two inputs are both low. Notice that this circuit consists of two inverters connected to form a cycle. The outputs of the two inverters are stable and can store a data value indefinitely.

Now, let's look at what happens when the S input of the latch goes high, while the R input remains low, the output of the bottom NOR gate will go low, causing the output of the top gate to go high. This makes the top input of the bottom gate high. Once that happens, we can drop the S input low again, and the latch will remain set. Similarly, raising R high while keeping S low, causes the latch to reset.

Now, what happens if we raise both the S and R inputs high at the same time. We said earlier that the latch outputs are not defined in this case. That is, the logical specification of the latch does not account for this situation, since it is not consistent with the intended usage of the latch. Still, there is nothing to stop us from applying these inputs to the actual circuit. When we do so, we observe that both NOR gate outputs go low, leading to the contradictory situation that Q and Q' have the same value. This makes it a little unclear what the state of the latch is at this point, so for that reason alone, it makes sense to avoid the situation where S and R are both high at the same time.

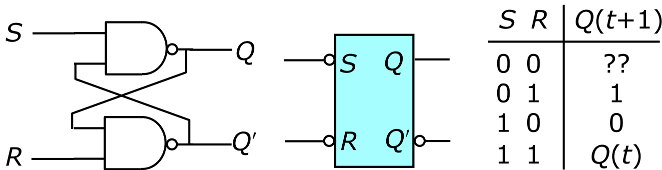
It turns out, there is another reason to avoid the condition $S = R = 1$. Consider what happens if the circuit is in this state and we simultaneously change S and R from high to low, meaning that all gate inputs are low. This

will cause the outputs of both NOR gates to change from low to high, after a short delay. Once the gate outputs become high, both NOR inputs have a high input. This causes the gate outputs to change again, from high to low. But this leads back to the situation where all gate inputs are low, triggering yet another transition from low to high. This process can continue indefinitely and is an example of *metastability*. In general, we say that a latch or flip flop is metastable if it is stuck between the 0 and 1 states. Metastability can cause voltages in a circuit to oscillate between high and low, or to remain stuck at an intermediate voltage that is neither low, nor high. The timing simulation below shows an example of an SR-latch becoming metastable.

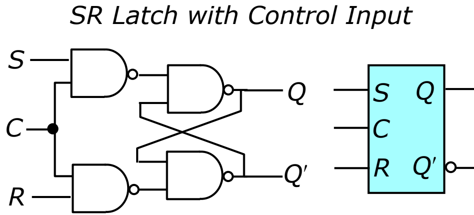


Metastability is an inherent property of storage elements in digital circuits and can cause circuits to behave in unpredictable ways. We can usually avoid metastability by avoiding the conditions that cause circuits to become metastable. We'll see in the next chapter there are some situations where we cannot always avoid metastability. Fortunately, in those situations where we cannot avoid it, we can usually mitigate its effects.

Now, it turns out that an SR-latch can be implemented with NAND gates instead of NOR gates, as illustrated in the diagram below.

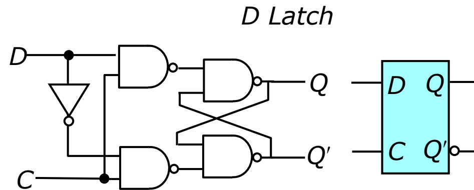


This circuit behaves slightly differently from the NOR-based latch, since the latch sets when the S input is low, rather than when it is high (similarly for the R input). Latches are often equipped with control inputs. In the varant shown below, the state of the latch can only be changed when the control input is high.



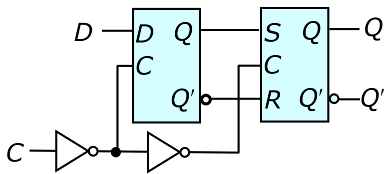
Also, note that the additional NAND gates in this circuit effectively invert the S and R inputs when $C = 1$, so this latch will be set when the S input and C input are both high.

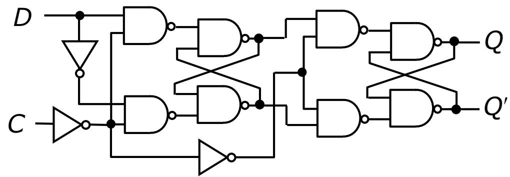
Now, this last version of the SR-latch can be turned into an implementation of a D-latch by adding an inverter, as shown below.



Like the SR-latch, the D-latch can become metastable under certain conditions. In this case, metastability can occur if the D input of the latch is changing at the same time the the control input is going from high to low. As discussed in an earlier chapter, we can use a pair of D-latches to implement a D-flip flop.

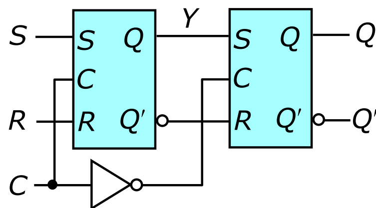
When the flip flop's control input is low, the first latch is transparent, while the second is opaque. When the flip flop's clock input (C) goes high, the first latch becomes opaque, meaning that its output cannot change, while the second latch becomes transparent, reflecting the value in the first latch at the



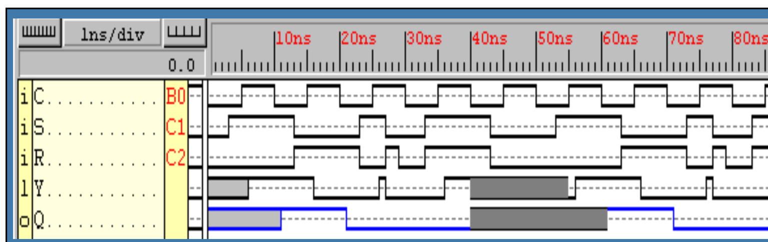


time the clock input went high. The propagation delay of a flip flop is the time beteen the clock input going from low-to-high and any corresponding change at the output. Because the latches from which they are constructed can become metastable, flip flops can also.

We're going to finish up this section with a brief description of several different types of flip flops. While these other types are not often used in circuits constructed using modern CAD tools, they do sometimes appear in circuits, so it's a good idea to be aware of them and understand how they work. We'll start with the SR-master-slave flip flop. This flip flop can be implemented using a pair of SR-latches, as shown below.

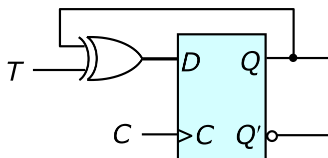


When the control input is high, the first latch can be set or reset, using the S and R inputs. When the control input goes low, the first latch can no longer change, and the value it stores gets propagated to the second latch. Note that this flip flop is not edge-triggered. Changes to the S and R inputs that occur while the control input is high can affect the output, as illustrated in the simulation shown below.



Observe that this simulation also shows the flip flop becoming metastable. The figure below shows a *toggle flip flop*.

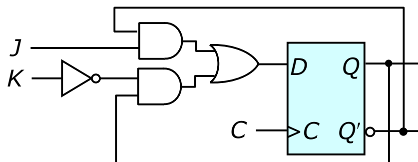
T	$Q(t+1)$
0	$Q(t)$
1	$Q'(t)$



This is an edge-triggered flip flop with a "toggle" input called T. If T is high when the clock goes from low-to-high, the flip flop toggles from one state to the other. The right part of the diagram shows an implementation that uses a D-flip flop and an exclusive-or gate.

Our final flip flop is a clocked JK-flip flop, which combines aspects of the SR and toggle flip flops.

J	K	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$Q'(t)$



If the J input of this flip flop is high on a rising clock edge, and the K input is low, the flip flop will be set (that is, the output will become high). Similarly, if the K input is high and the J input low on a rising clock edge, the flip flop will be reset. If both are high, when the clock goes high, the state of the flip flop toggles. An implementation using a D-flip flop and some additional gates is shown in the right part of the diagram.

Chapter 16

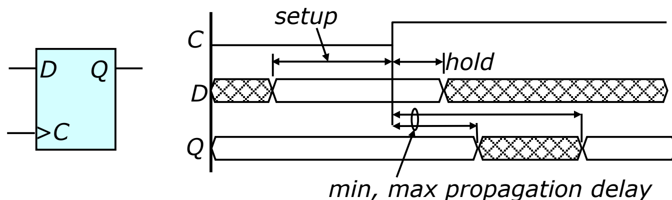
Timing Issues in Digital Circuits

In earlier chapters, we have mentioned that if the input to a D-flip flop is changing at the same time that the clock is going from low to high, the flip flop can become metastable, causing the overall circuit behavior to be unpredictable. To ensure reliable operation, it's important to make sure that flip flop inputs are stable when the clock is changing. In this chapter, we will look more closely at this issue and learn how we can structure our circuits to ensure reliable operation, and how CAD tools support this.

16.1 Flip Flop Timing Parameters

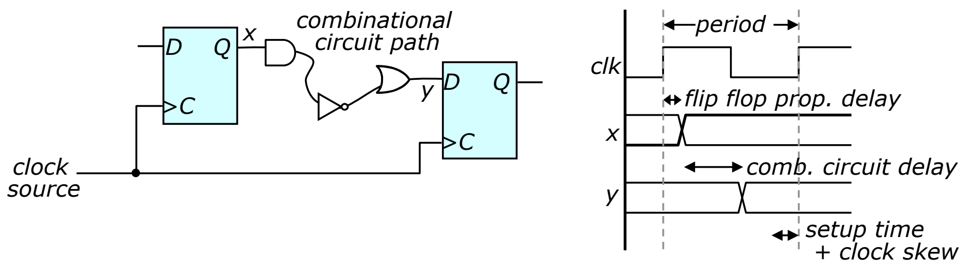
Let's start by looking more closely at the operation of the D-flip flop. We've said that in order to avoid metastability, the D input to the flip flop must be stable when the clock is changing. To make that statement more precise, we want to define a time interval around the time the clock makes its low-to-high transition when the flip flop input must not change. This *stable interval* is defined by two parameters. The *setup time* of the flip flop is the time from the start of the stable interval to the rising clock edge, while the *hold time* is the time from the rising clock edge to the end of the stable period. These

definitions are illustrated in the diagram shown below.



In this diagram, the unshaded portion of the waveform for the D input is the stable period. To ensure safe operation, the signal should not change during this time interval. It is safe for the input to change during the shaded portion of the waveform. The shaded part of the waveform for output Q is the period when the output can change. It is bounded by the minimum and maximum propagation delay of the flip flop.

The timing constraints on the D-input have some implications for the flip flop operation. Consider the following diagram, which shows two flip flops connected by a circuit path that passes through several gates.



Suppose the clock makes a low-to-high transition, causing the output of the first flip flop to change after a short delay. As the signal propagates through the connecting gates to the second flip flop, it will be delayed further. If the total delay along this path is too long, it's possible that the input of the second flip flop will be changing when the *next* rising clock edge occurs. To ensure that this does not happen, we require that the following inequality be

satisfied.

$$\begin{aligned}(\textit{clock period}) \;\geq\; & (\textit{max FF propagation delay}) \\ & + (\textit{max combinational circuit delay}) \\ & + (\textit{FF setup time}) + (\textit{max clock skew})\end{aligned}$$

This is known as the *setup time condition*. Here, clock skew is a parameter that specifies the maximum time difference between the arrival of a clock edge at two different flip flips. In integrated circuits, the circuitry that delivers the clock to different flip flops is designed to keep this difference as small as possible. Still, there is always some non-zero difference in the clock arrival times and this must be taken into account.

Note that the setup time condition essentially imposes a lower bound on the clock period. Modern CAD tools can check all paths from the output of one flip flop to the input of another, and use this to compute a minimum clock period for the overall circuit. In the Xilinx design tools, this minimum clock period is reported in the *synthesis report* that is produced as part of the synthesis phase of the circuit implementation process. An example of the relevant section of the synthesis report is shown below.

```
==================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 4.227ns (frequency: 236.560MHz)
  Total number of paths / destination ports: 45 / 5
------------------------------------------------------------------
Delay:               4.227ns (Levels of Logic = 3)
  Source:            state_FFd1 (FF)
  Destination:       cnt_2 (FF)
  Source Clock:      clk rising
  Destination Clock: clk rising

  Data Path: state_FFd1 to cnt_2
                           Gate     Net
    Cell:in->out   fanout Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDR:C->Q         9   0.626   1.125  state_FFd1 (state_FFd1)
     LUT2:I1->O       1   0.479   0.740  _mux0001<2>20_SW0 (N123)
```

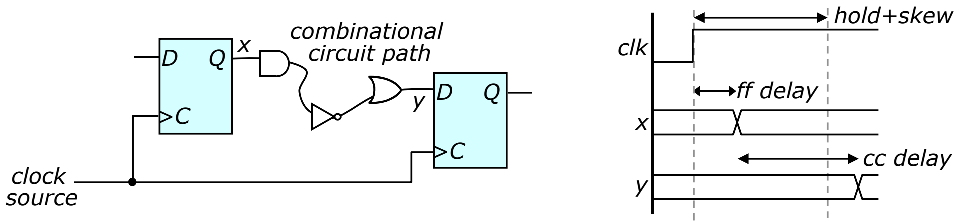
```
     LUT4_L:I2->LO     1   0.479   0.123  _mux0001<2>24_SW0 (N119)
     LUT4:I3->O        1   0.479   0.000  _mux0001<2>43 (_mux0001<2>)
     FDS:D                 0.176          cnt_2
    ----------------------------------------
    Total                  4.227ns (2.239ns logic, 1.988ns route)
                                   (53.0% logic, 47.0% route)

=====================================================================
```

The first few lines summarize the overall results of the timing analyis. They show a minimum clock period of 4.227 ns, resulting in a maximum clock frequency of 236.56 MHz. The rest of the text describes a particular circuit path that has the worst-case delay of 4.227 ns. It identifies the source and destination flip flops, and the sequence of components that the circuit passes through as it goes from the source flip flop to the destination. The per component delay is divided into two parts, an intrinsic "gate delay" and a "net delay" that is related to the length of the connecting wires and the fanout of the network (recall that larger fanout implies larger capacitance, which in turn leads to higher delays).

Often, when designing a circuit, we have some specific clock period that we're trying to achieve, so we canot simply increase the clock period to ensure that the setup time condition is always satisfied. During the "place-and-route" phase of the circuit implementation process, modern CAD tools attempt to place components on the chip, and route the connecting wires so as to avoid violations of the setup time condition. However, in some situations, the tools may be unable to find any combination of component placement and routing that satisfies all setup time conditions at the same time. In this situation, circuit designers must either settle for a longer clock period than they would like (reducing the performance of the system), or restructure the circuit in some way to eliminate the setup time violations without increasing the clock period.

There is a second timing condition that circuits must also satisfy, to ensure that flip flop inputs are only changing when it's safe to do so. Again, let's consider the path from one flip flop to another. Note that if the sum of the flip flop propagation delay and the combinational circuit delay is too

small, it's possible for the input of the second flip flop to change during the time interval when it's required to be stable. This is illustrated below

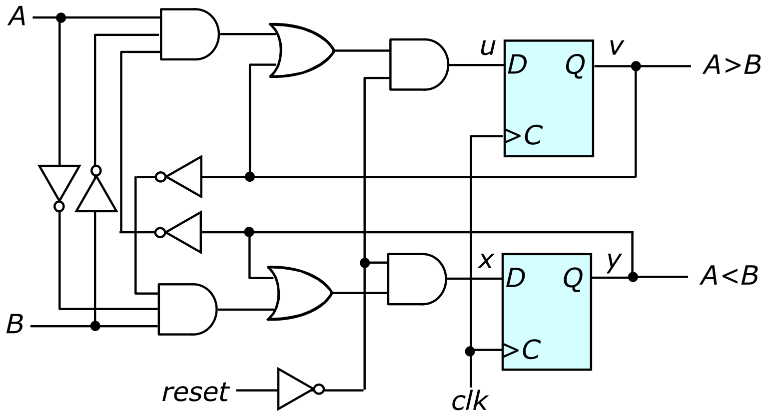


The *hold time condition* requires that

$$\begin{aligned}(\mathit{hold\ time}) + (\mathit{max\ clock\ skew}) \quad \leq \quad & (\mathit{min\ FF\ propagation\ delay}) \\ & + (\mathit{min\ combinational\ circuit\ delay})\end{aligned}$$

Note that if a manufactured integrated circuit violates the setup time condition, we can still use the circuit by operating it with a reduced clock period. Violations of the hold time condition are more serious, in that they can prevent a circuit from working reliably at any clock speed. Fortunately, in many contexts hold time violations are unlikely to occur, because the sum of the flip flop hold times and the clock skew are usually smaller than the minimum flip flop propagation delay. So, even when the combinational circuit delay is zero, hold time violations will not occur.

Next, let's consider an example of a simple circuit and look at how we can manually check the that the setup and hold time conditions are satisfied. The diagram below shows a state machine that implements a bit-serial comparison circuit.



After reset goes low, the circuit compares inputs A and B, one bit at a time. The most-significant bits arrive first, so the first time the circuit detects a difference between the two inputs, it can tell whether $A > B$ or $A < B$. The inputs and outputs of the two flip flops have been labeled u, v and x, y for reference. Let's suppose that inverters have a propagation delay between 1 ns and 2 ns, that AND gates and OR gates have a propagation delay that is between 2 ns and 4 ns, and that flip flops have a propagation delay between 3 ns and 8 ns. Let's also suppose that the flip flop setup time is 2 ns, that the hold time is 1 ns and that the clock skew is .5 ns. (Note, these are not realistic values; they have been chosen simply to keep the arithmetic simple. On the other hand, the variation in values does reflect the charactistics of real circuits. The delays in physical components can vary considerably, due to variations in the manufacturing process and the operating conditions for the circuit.)

Let's start by checking the hold time condition from the output of the first flip flop to the input of the second flip flop (so from v to x).

$$\begin{aligned}(\textit{hold time}) + (\textit{max clock skew}) &\leq (\textit{min FF propagation delay}) \\ &\quad + (\textit{min combinational circuit delay}) \\ 1 + .5 = 1.5 &\leq 3 + 1 + 3 \times 2 = 10\end{aligned}$$

We can see that the inequality is not violated in this case, so the hold-time

condition is satisfied for the path from v to x. Since the path from y to u is symmetric with the v-to-x path, we can conclude that it also satisfies the hold-time condition. Next, let's consider the path from v to u. This passes through two gates.

$$\begin{aligned}(\textit{hold time}) + (\textit{max clock skew}) &\le (\textit{min FF propagation delay}) \\ &\quad + (\textit{min combinational circuit delay}) \\ 1 + .5 = 1.5 &\le 3 + 2 \times 2 = 7\end{aligned}$$

This also satisfies the hold-time condition. Actually, in this case we can drop the clock skew from the hold-time condition since we're going from the output of a flip flop back to its own input, the clock skew is zero in this case. Finally, we note that the y-to-x path is symmetric with the v-to-u path, so it also satisfies the hold time condition.

Next, let's look at the setup time condition for the path from v to x.

$$\begin{aligned}(\textit{clock period}) &\ge (\textit{max FF propagation delay}) \\ &\quad + (\textit{max combinational circuit delay}) \\ &\quad + (\textit{FF setup time}) + (\textit{max clock skew}) \\ (\textit{clock period}) &\ge 8 + (2 + 3 \times 4) + 2 + .5 = 24.5\end{aligned}$$

The setup time condition implies a minimum clock period of 24.5 ns. Next, let's check the path from v to u.

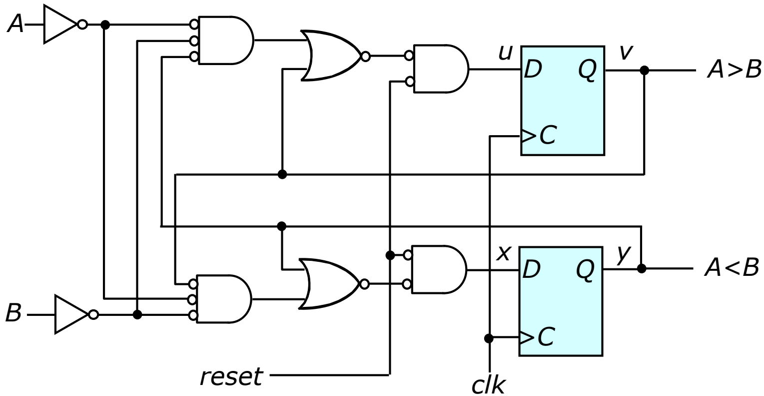
$$\begin{aligned}(\textit{clock period}) &\ge (\textit{max FF propagation delay}) \\ &\quad + (\textit{max combinational circuit delay}) \\ &\quad + (\textit{FF setup time}) + (\textit{max clock skew}) \\ (\textit{clock period}) &\ge 8 + (2 \times 4) + 2 + .5 = 18.5\end{aligned}$$

We can actually drop the clock skew from the condition in this case also, so the minimum clock period implied by this path is 18 ns. Since the other two paths are symmetric with the first two, the worst-case clock period is 23.5 ns, giving a maximum clock frequency of 42.6 MHz.

Suppose we have a circuit with a hold time violation. Is there anything we can do about it? The answer is yes, we simply add delay. Suppose that in the

previous example, the hold time parameter for the flip flops was 8 ns instead of 1. This would lead to hold time violations on the u-to-v path and the x-to-y path. We can eliminate the violation by inserting a pair of inverters into each of these paths, right before the OR gate inputs. This increases the minimum path delay by 2 ns, eliminating the hold-time violation.

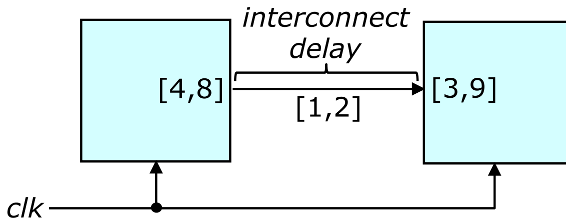
Now, let's return to the original example, but let's suppose that the 23.5 ns clock period is not acceptable for the application in which we're using the circuit. Is there anything we can do to reduce the maximum circuit delay, in order to improve the performance? In this case, we can replace the AND and OR gates with NOR gates as illustrated below.



(Note, some of the NOR gates are drawn in their alternate form, to emphasize the similarity with the original circuit.) Because NOR gates are faster than the ANDs and ORs they're replacing, we get an overall reduction in the maximum propagation delay. Moreover, this version does not require inverters in the v-to-x and y-to-u paths, further reducing the delay on those paths. In a later chapter, we'll take a more systematic look at how we can reduce delays in combinational circuits in order to obtain better performance. This often involves more fundamental restructuring of the circuit, to obtain an equivalent circuit with lower delay and hence higher performance.

Modern CAD tools can perform the required timing analysis for all pairs of flip flops within a single integrated circuit or FPGA. However, what happens

when we are building a larger system using multiple integrated circuits or FPGAs? In this context, we may not have detailed information about the circuits within the components we're using, but we still need to be able to verify that the setup and hold conditions will be satisified within the larger system. To do this, we need to extend our timing analysis across all the paths that pass between components. In order to do this, we need some information about the delays within the components we're using. Consider the example shown below.



The output from the lefthand component has been labeled with a pair of numbers [4,8]. These represent the minimum and maximum delays in the circuit from the time the clock changes, to the time when a change appears at the output (note, that in this example we are considering a signal that only changes following a clock transition). These delay numbers include the flip flop propagation delay in addition to the delay on the circuit path from the flip flop to the output of the component.

The input of the righthand component is labeled similarly, with the minimum and maximum delays from that input to a flip flop within the circuit. The interconnect between the circuit components can also contribute significantly to the delay, and is labeled with pair of numbers, indicating minimum and maximum delays.

Using the numbers shown in the diagram, the minimum delay from a flip flop in the lefthand component to a flip flop in the righthand component is 8 ns, while the maximum delay is 19 ns. If the flip flop setup time is 2 ns and the clock skew is 1 ns, the minimum clock period implied by this connection is 22 ns. This connection will not cause a hold time violation, so long as long as the flip flop hold time is at most 7 ns.

CAD tools can provide the information needed to check that setup and hold conditions are satisfied when combining components to build larger systems. A section of a synthesis report from the Xilinx tools is shown below

```
===================================================================
Timing constraint: Default OFFSET IN BEFORE for Clock 'clk'
  Total number of paths / destination ports: 17 / 12
-------------------------------------------------------------------
Offset:                4.356ns (Levels of Logic = 4)
  Source:              dIn (PAD)
  Destination:         cnt_2 (FF)
  Destination Clock:   clk rising

  Data Path: dIn to cnt_2
                          Gate    Net
  Cell:in->out   fanout Delay  Delay  Logical Name (Net Name)
  --------------------------------------------------------
  IBUF:I->O          7   0.715  1.201  dIn_IBUF (dIn_IBUF)
  LUT4:I0->O         1   0.479  0.704  _mux0001<2>33 (_mux0001<2>_map1
  LUT4_L:I3->LO      1   0.479  0.123  _mux0001<2>24_SW0 (N119)
  LUT4:I3->O         1   0.479  0.000  _mux0001<2>43 (_mux0001<2>)
  FDS:D                  0.176         cnt_2
  --------------------------------------------------------
  Total                  4.356ns (2.328ns logic, 2.028ns route)
                                 (53.4% logic, 46.6% route)

====================================================================
```

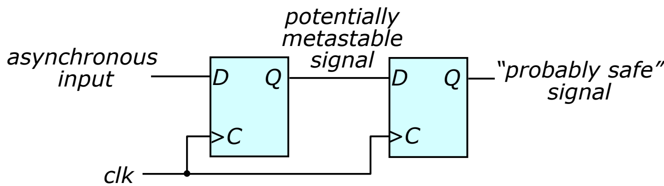
This shows the maximum delay from an input to a flip flop. The synthesis report also provides information about output delays.

16.2 Metastability and Synchronizers

By carefully adhering to the timing rules discussed in the last section, we can usually ensure that the inputs to flip flops change only when it is safe for them to do so. However, there are some situations when the timing of input signals is outside our control as circuit designers. For example, when someone presses a button on the prototype board, the timing of that signal

transition is inherently unpredictable, so there is always some chance that this will cause the D input of some flip flop to change at the same time as a rising clock edge, potentially causing the flip flop to become metastable.

So what can we do about this? The key to addressing this problem is to use a special circuit called a *synchronizer* at each input that can change asynchronously. Now a synchronizer is not a panacea, in that it cannot prevent metastability failures 100% of the time. However, synchronizers can be designed to make metastability failures extremely rare. To see how this is done, let's take a look at a basic sychrnonizer.



This circuit consists of two flip flops connected in series. The input to the first flip flop is connected to the input that has the potential to change asynchronously. The output of the second flip flop connects to the rest of our circuit. So long as the second flip flop does not become metastable, the remainder of our circuit can be protected from metastability failures. We say that the synchronizer *fails* if the second flip flop does become metastable.

Now, let's suppose that the asynchronous input does change at the same time as a rising clock transition This can cause the first flip flop to become metastable. Now, the second flip flop won't be immediately affected by this, but if the first flip flop is still metastable at the time of the next rising clock edge, the second flip flop can also become metastable, leading to a failure of the synchronizer.

Now, what makes the synchronizer effective is that when a flip flop becomes metastable, it rarely stays metastable for very long. We cannot put any definite upper bound on how long it will be metastable, but the more time passes, the less likely it is that the flip flop will remain metastable. So, one way to reduce the probability of a synchronizer failure is for the synchronizer to use a clock signal with a relatively long period. While this can be

effective, we probably don't want to make the synchronizer clock period too long, as this may have a negative impact on system performance. So, it's important to be able estimate the frequency with which synchronizer failures can occur, so that we can engineer our systems to make them extremely rare.

So how likely is it that an asynchronous signal transition leads to a synchronizer failure. Well, there are two things that must happen for a synchronizer to fail. First, the signal must change, just when the clock is making its rising transition. Let T_0 be the length of the critical time interval during which signal changes lead to metastability. If T is the clock period, then the probability that a random signal change will trigger metastability is T_0/T.

Now, for a synchronizer to fail, the first flip flop must not only become metastable, it must stay metastable for nearly a full clock period. Experimental measurements of flip flops have shown that the amount of time that they remain metastable follows an exponential distribution. More precisely, the probability that a metastable flip flop is still metastable after T time units is less than $e^{-T/\tau}$ where $e \approx 2.71828$ is the base of the natural logarithm and τ is a parameter of the flip flop. Hence, the probability that a random signal transition causes a synchronizer failure is

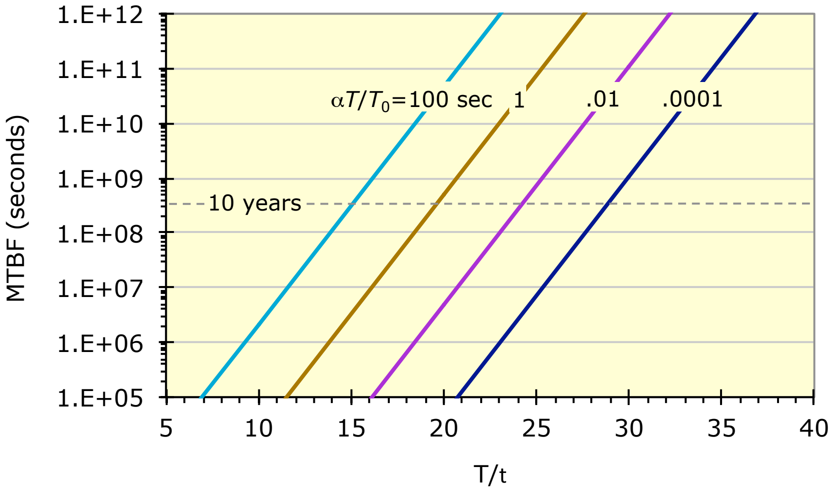
$$(T_0/T)e^{-T/\tau}$$

Now the time between synchronizer failures depends not only on this probability, but on how often asynchronous signal transitions occur. If α is the average amount of time between consecutive asynchronous signal transitions, then the *mean time between failures* for the synchronizer is

$$\alpha/((T_0/T)e^{-T/\tau}) = (\alpha T/T_0)e^{T/\tau}$$

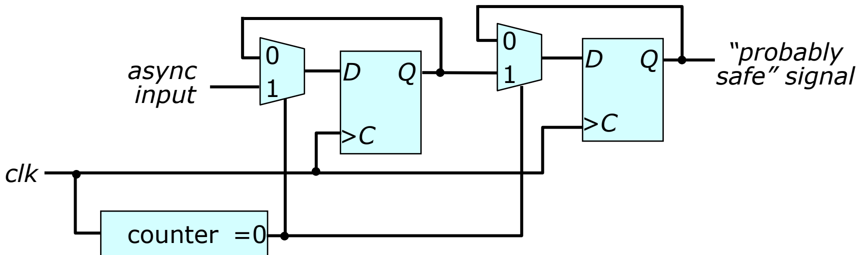
So for example, if $T = 50$ ns, $\alpha = 1$ ms, $\tau = T_0 = 1$ ns then the mean time between failures is approximately 8 trillion years. However, note that this time is very sensitive to the paramater choices, especially the value of T. In particular, if $T = 10$ ns, the mean time between failures drops to just 220 seconds.

The chart shown below shows the mean time to failure under a variety of conditions.



Observe that whenever T/τ increases by 2, the MTBF increases by about an order of magnitude. The dashed line indicates an MTBF of 10 years. Points at the top end of the range represent MTBF values of more than 10 thousand years.

We can make a synhronizer as reliable as we like by increasing time between the instants when the asynchronous input is sampled. Here is a circuit that illustrates this.



By increasing the number of bits in the counter, we increase the effective period of the synchronizer without changing the period of the clock used

by the rest of the system. The drawback of this approach is that we delay processing of the asynchronous input and we may even fail to observe some signal transitions. However in many situations, there is a known lower bound on the time between signal transitions. So long as the time between sampling intervals is smaller than the minimum time between signal transitions, we can avoid missing any transitions.

Part III

Third Half

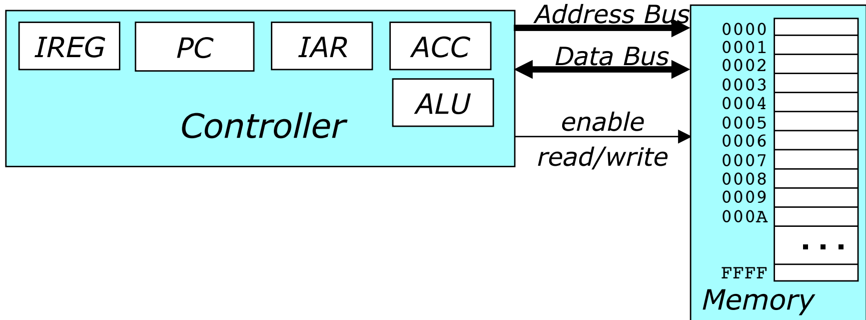
Chapter 17

Introduction to Programmable Processors

As mentioned in Chapter 1, one of the most important digital circuits is the *programmable processor*. These are the circuits that drive our laptops and cell phones, as well as many modern appliances, and even our cars. What makes processors so important is their remarkable flexibility, and the key to that flexibility is *programmability*. While a processor is just a single circuit, it can be programmed to implement a remakable diversity of functions. This programmability means that one device can do many different things. What we will discover in the next couple chapters is that as powerful as they are, processors can be implemented using fairly modest circuits. While it is certainly true that the modern processors at the heart of our laptops and cell phones are pretty complex, most of that complexity is there for the sole purpose of improving performance. The essential feature of programmability does not require it.

17.1 Overview of the WASHU-2 Processor

We will begin our study of processors with an example of a simple processor, called the WASHU-2, which is illustrated in the diagram below.



There are two main components to this system, the processor itself, and the attached memory which stores data used by the processor and *instructions* that determine what the processor does. The two are connected by a 16 bit wide address bus and a 16 bit wide, bidirectional data bus. There are also two control signals, *enable* and *read/write*.

The processor contains four registers. The *Instruction Register* (IREG) is used to hold a copy of the instruction currently being executed. The *Program Counter* (PC) holds the address of the current instruction. The *Accumulator* (ACC) is used to hold intermediate results that are computed by the processor. The *Indirect Address Register* (IAR) is used when processing indirect address instructions, which we'll describe shortly. The *Arithmetic/Logic Unit* (ALU) implements the core arithmetic functions (addition, negation, logical AND, and so forth); in the WASHU-2, it is a fairly simple combinational circuit. Finally, there is a controller that coordinates the activities of the other elements of the processor and controls the transfer of data between the processor and the memory.

Now that we've covered the components that make up our programmable processor, we can look at what it actually does. The central activity of the processor is to retrive instructions from memory and carry out the steps required by each instruction. This process is referred to as the *fetch-and-execute cycle* and the processor repeats it over and over again as it executes programs. In the *fetch* portion of each cycle, the processor retrieves a word from memory, using the value in the PC as the memory address. The value

returned by the memory is stored in the IREG. Now this value is just a 16 bit number, but the processor *interprets* it as an instruction. In the *execute* part of the instruction, the processor does whatever the current instruction directs it to do. For example, it might retrieve some data from memory and place it in the accumulator, or it might change the value of the PC so as to change which instruction is handled next. By default, the processor proceeds from one instruction to the one at the next location in memory. The processor implements this by simply incrementing the program counter.

Before we move on to describe the WASHU-2's instructions, we need to introduce a simple property of 2s-complement binary numbers. Suppose, we have a four bit binary number that we want to convert to an eight bit binary number. If the four bit number is 0101, it's obvious what we should do, just add 0s on the left, giving us 00000101. But what if the original number is negative, like 1010, for example. It turns out that to produce an eight bit number with the same magnitude, all we need to do is add 1s to the left, giving us 11111010. You can check this by taking the 2s complement of 1010 and 11111010. So in general, if you want to convert a four bit number to an eight bit number, you just copy the *sign bit* into the new positions. This process is called *sign extension* and can be used to convert any binary number to a longer format without changing its magnitude. Several of the WASHU-2 instructions use sign extension, as we'll see shortly.

The WASHU-2 has just 14 instructions, so we'll describe each of them in detail. The first instruction is the *Halt* instruction, which has the numerical value x0000. It simply causes the processor to stop processing instructions, and is summarized below.

```
0000  halt - halt execution
```

The next instruction is a little more interesting. It is the *negate* instruction and has the numerical value x0001. To execute the negate instruction, the processor simply replaces the value in the accumulator by its negation.

```
0001   negate - ACC = -AC
```

The next instruction is the *branch* instruction, which adds a specified offset to the value in the PC in order to change which instruction gets executed

next. The first two hex digits of this instruction are 01 and the remaining digits are treated as an eight bit signed number. This number is extended to 16 bits (using sign-extension) and then added to the PC.

```
01xx   branch - PC = PC + ssxx
```

In the instruction summary, the value ssxx refers to the sign-extended value derived from the low order eight bits of the instruction word. There are several *conditional branch* instructions that are similar to the (unconditional) branch.

```
02xx   branch-if-zero - if ACC==0, PC = PC + ssxx
03xx   branch-if-positive - if ACC>0, PC = PC + ssxx
04xx   branch-if-negative - if ACC<0, PC = PC + ssxx
```

Notice that these branch instructions all have a limitation, which is that they can only adjust the PC value by about 128 in either direction. More precisely, we can only jump ahead by 127 instructions, or jump back 128. There is one more branch instruction, called the *indirect branch*, that does not have this limitation.

```
05xx   indirect branch - PC = M[PC + ssxx]
```

Here, the notation `M[PC + ssxx]` refers to the value stored in the memory at the address equal to `PC + ssxx`. Notice that since the value stored at `M[PC + ssxx]` can be any 16 bit number, we can use the indirect branch to jump to any location in memory.

Our next instruction loads a constant into the accumulator.

```
1xxx   constant load - ACC = sxxx
```

Here again, we are using sign-extension, this time to convert the low-order 12 bits of the instruction word into a 16 bit signed value with the same magnitude. The *direct load* instruction loads a value into the accumulator from a specified memory location.

```
2xxx   direct load - ACC = M[pxxx]
```

The notation M[pxxx] refers to the value in the memory location specified by pxxx, where the xxx comes from the instruction word and the p is the high-order four bits of the PC. This makes it easy to access memory locations that are in the same part of memory as the current instruction. The *indirect load* instruction allows us to access a memory location through a *pointer*.

```
3xxx   indirect load - ACC = M[M[pxxx]]
```

The *direct store* and *indirect store* instructions are similar to the load instructions. They just transfer data in the opposite direction.

```
5xxx   direct store - M[pxxx] = ACC
6xxx   indirect store - M[M[pxxx]] = ACC
```

There are just two more instructions. The *add* instruction adds the value in a memory location to the value currently in the accumulator. The *and* instruction performs a bit-wise logical AND of the value in a specified memory location with the value in the accumulator.

```
8xxx   add - ACC = ACC + M[pxxx]
cxxx   and - ACC = ACC and M[pxxx]
```

Summarizing, we have

```
0000  halt - halt execution
0001   negate - ACC = -AC

01xx   branch - PC = PC + ssxx
02xx   branch-if-zero - if ACC==0, PC = PC + ssxx
03xx   branch-if-positive - if ACC>0, PC = PC + ssxx
04xx   branch-if-negative - if ACC<0, PC = PC + ssxx
05xx   indirect branch - PC = M[PC + ssxx]

1xxx   constant load - ACC = sxxx
2xxx   direct load - ACC = M[pxxx]
3xxx   indirect load - ACC = M[M[pxxx]]
```

```
5xxx   direct load - M[pxxx] = ACC
6xxx   indirect load - M[M[pxxx]] = ACC

8xxx   add - ACC = ACC + M[pxxx]
cxxx   and - ACC = ACC and M[pxxx]
```

To run a program on the WASHU-2, all we need to do is load the instructions into memory, starting at location 0, then reset the processor. This will cause it to begin the fetch-and-execute cycle, retrieving instructions based on the value of the PC and carrying out the steps involved in each instruction, one after another.

Note that the basic fetch-and-execute cycle is the one essential feature of a programmable processor. Instructions are nothing but numbers stored in a computer's memory that are interpreted in a special way. There is an endless variety of instruction sequences we can specify, and this is what allows us to program the processor to do so many different things. Even though the WASHU-2 is a relatively simple processor, it is fundamentally no different from the processors that run our laptops and cell phones. All execute instructions stored in memory in a similar fashion. The main difference is that modern processors have larger memories and can execute instructions at faster rates.

17.2 Machine Language Programming

Now that we have seen the instruction set used by the WASHU-2, let's take a look at some programs. We'll start with a simple program that adds a sequence of numbers and stores the sum in a specified location in memory. However, before we do that, we need to say a little bit about how a user interacts with a program on the WASHU-2.

Most "real computers" have input and output devices, like a keyboard, a mouse and a video display that we can use to interact with running programs. Since we'll be implementing the WASHU-2 on our prototype boards, we need to interact with the processor using the limited mechanisms that our boards provide. In the next section, we'll see how we can use the prototype board's knob, buttons and LCD display to read values stored in memory and

change the values in selected memory locations. We'll use this mechanism to enter data for use by running programs and to retrieve the results of their computations.

The first program we'll implement is equivalent to the program shown below, written in C-style notation.

```
sum = 0;
while (true) {
    inVal = -1;
    while (inVal == -1) {} // wait for user to change inVal
    if (inVal = 0) break;
    sum += inVal;
}
```

In this program, `inVal` and `sum` refer to specific locations in the WASHU-2's memory. Specifically, `inVal` is the location where a user enters input data, and `sum` is the location where the processor places its result. The program starts by setting `inVal` to -1 and waits for the user to set it to some other value (this does imply that the user cannot enter -1 as an input value). If the value entered by the user is 0, the program interprets that as a signal to terminate the program. To implement this using the WASHU-2's machine instructions, we'll go through the C program, one statement at a time, replacing each with an equivalent sequence of WASHU-2 instructions. The initialization of *sum* is easy.

```
0000  1000  (ACC = 0)                    -- sum = 0;
0001  5011  (M[0011] = ACC)
```

The left column indicates the address at which the instruction is stored, while the next column is the instruction itself. The text in parentheses is just a reminder of what the instruction does, while the comment at right shows the C-statement that is implemented by this sequence of instructions. Observe that the first instruction just loads a constant 0 into the accumulator, while the next instruction stores this in memory location `0011`, which is the location that the program uses to store its result.

The next two instructions shows the start of the outer loop.

```
0002  1FFF  (ACC= 1)                  -- while (true) {
0003  5010  (M[0010] = ACC)            --     inVal = -1;
```

Since there is no condition to test for the loop, we don't need any instructions here to implement the start of the loop. The immediate load instruction loads -1 into the accumulator (observe the use of sign-extension) and the next instruction stores this in location `0010`, which is the location used by the program for input data from the user.

Next, we have the inner loop.

```
0004  1001  (ACC = 1)                  --     while(inVal==-1)
0005  8010  (ACC = ACC+M[0010])        --         {}
0006  02FE  (if ACC=0 go back 2)
```

The first instruction loads 1 into the accumulator, while the second adds the current value of `inVal` to the accumulator. If the user has not changed the value of `inVal`, this will make the value in the accumulator 0. If it is, the conditional branch instruction at location 0006 subtracts 2 from the program counter (again, notice the use of sign-extension), causing the program to return to location 0004 for the next instruction. When the user does eventually change `inVal`, the program continues to the next instruction in sequence.

```
0007  2010  (ACC = M[0010])            --      if(inVal==0)
0008  0204  (if ACC=0 go ahead 4)      --          break;
```

Here, the first instruction loads `inVal` into the accumulator. If it's zero, we jump ahead four instructions in order to break out of the loop. The remainder of the program is

```
0009  8011  (ACC = ACC + M[0011])      --      sum += inVal
000A  5011  (M[0011] = ACC)
000B  01F7  (go back 9)                -- }
000C  0000  (halt)                     -- halt
```

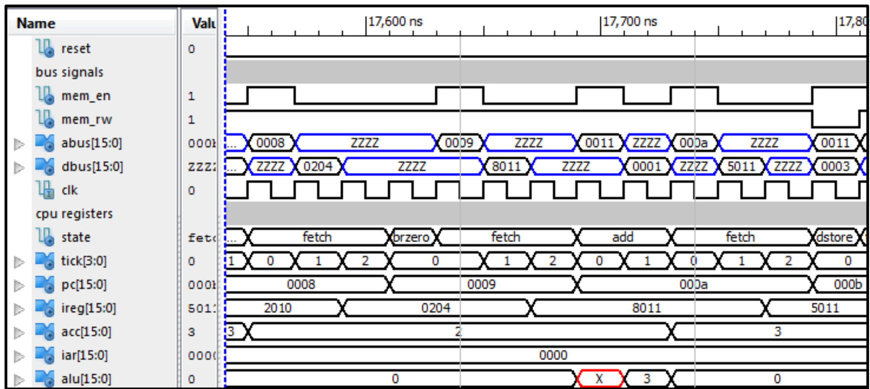
The first instruction adds the current `sum` to the value in the accumulator (which is inVal) and the next stores the result back to `sum`. The branch

instruction at location 000B goes back 9 locations to location 0002, which is where the outer loop started.

Putting it altogether, we have

```
0000  1000  (ACC = 0)                  -- sum = 0;
0001  5011  (M[0011] = ACC)
0002  1FFF  (ACC= -1)                  -- while (true) {
0003  5010  (M[0010] = ACC)            --     inVal = -1;
0004  1001  (ACC = 1)                  --     while(inVal==-1)
0005  8010  (ACC = ACC+M[0010])        --         {}
0006  02FE  (if ACC=0 go back 2)
0007  2010  (ACC = M[0010])            --     if (inVal==0)
0008  0204  (if ACC=0 go ahead 4)      --         break;
0009  8011  (ACC = ACC + M[0011])      --     sum += inVal;
000A  5011  (M[0011] = ACC)
000B  01F7  (go back 9)                -- }
000C  0000  (halt)                     -- halt
. . .
0010  1234                             -- input value
0011  5678                             -- sum
```

We can use the simulator to observe the program running. To do this, we need to initialize the processor's memory so that it contains the program when the simulator starts up. This is done by modifying the VHDL for the memory component so that the memory is initialized to contain the instructions listed in the program. We then simulate the processor using a testbench that resets the processor, then let's it run. Here is a small portion of a simulation of the sum program.



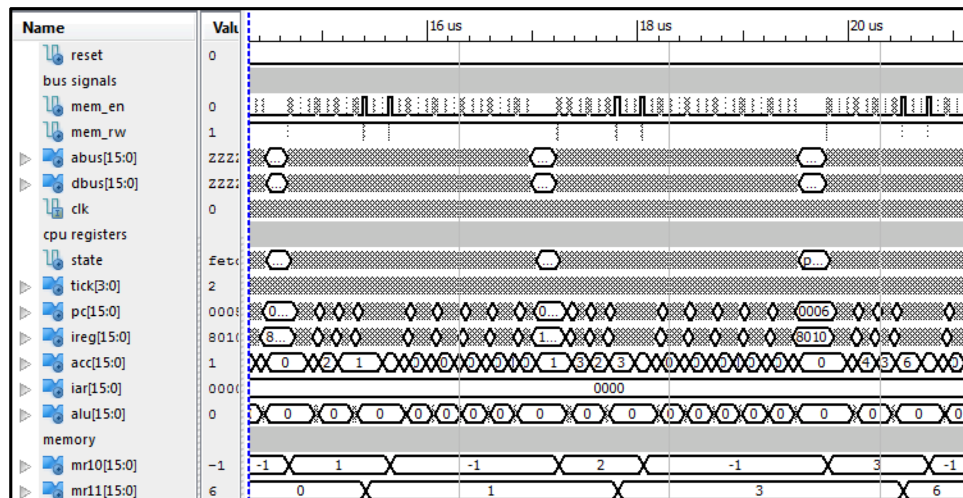
First, note the signals listed in the waveform display. It starts with a *reset* followed by the memory signals. The processor *state* waveform gives an indication of what the processor is doing at each point in time and the tick register identifies the individual clock tick within each processor state. The register values and ALU output are shown at the bottom.

This segment of the simulation starts with an instruction fetch from memory location 0008. We can observe the returned instruction (0204) on the data bus, and we can see it being stored in the IREG at the end of tick 1 of the fetch. We can also observe the PC being incremented at the end of the fetch. Instruction 0204 is a *branch-if-zero* instruction, but since the value in the accumulator is not 0, the processor continues to the next instrruction in sequence.

The next fetch retrieves the instruction 8011 from location 0009 (observe the changes to the address and data signals) and stores the instruction in the instruction register at the end of tick 1 of the fetch. Once again, the PC is incremented at the end of the fetch. Because 8011 is an *add* instruction, the processor initiates another read to memory location 0011. The value returned on the data bus is 0001. This is added to the ACC causing its value to increase from 2 to 3. The final instruction in this segment of the simulation is a direct store to location 0011. Observe the read/write signal going low, and the value 0003 being sent across the data bus to the memory.

The figure below shows a view of the processor at a much longer time-

scale.



At this time-scale, we cannot see the details of the instructions being executed, but we can observe changes to the memory locations used by the program for input and output. For the purposes of this simulation, the memory has been augmented with a pair of registers that are updated whenever locations 0010 and 0011 are modified. The last two lines of the waveform window show the values of these registers. We can observe that `M[0010]` is changed to 1 at the start of this portion of the simulation. This causes the running program to add 1 to the value of `M[0011]` and we can observe this in the waveform display. The program then changes `M[0010]` back to -1 and waits for it to change again. When the user sets `M[0010]` to 2, the program responds by adding this to the value in `M[0011]` and so forth.

Writing programs in machine language quickly becomes tedious. Higher level languages, like C or Java, allow us to write programs in a more convenient notation, while compilers handle the tedium of translating those programs into machine instructions. In the case of the WASHU-2, we do not have the luxury of a compiler, but we do have a simple *assembler* that makes the job somewhat easier. Here is an assembly language version of the machine language program we just looked at.

```
        location 0
        cLoad 0          -- sum = 0;
        dStore sum
loop:   cLoad -1         -- while (true) {
        dStore inVal     --     inVal = -1;
        cLoad 1          --     while (inVal == -1) {}
        add inVal
        brZero -2
        dLoad inVal      --     if (inVal == 0) break;
        brZero end
        add sum          --     sum += inVal
        dStore sum
        branch loop      -- }
end:    halt             -- halt
        location 0010
inVal:  01234            -- input value
sum:    05678            -- sum of input values
```

The very first line of this program is an *assembler directive* that instructs the assembler to assign the instruction on the next line to location 0000 in the WASHU-2's memory. It will assign subsequent instructions to 0001, 0002, 0003 and so forth. The first instruction is a constant load, designated by the mnemonic `cLoad`. The next instruction, a direct store, uses the mnemonic *dStore*.

Notice that we are not required to specify the location of the target of the store. Instead, we just use a name, which the assembler associates with the target location. A name in an assembly language program is defined by entering the name at the start of a line, followed by a colon. This associates the name with the memory location for that line. So for example, the name `loop` corresponds to location 0002, the name `end` corresponds to location 000c, `inVal` corresponds to location 0010 and `sum` to location 0011.

Note that while most branches in our program use labels, there is one that uses a numerical offset to branch back 2 instructions. Also notice that data values can be entered directly into memory as we have done here for `inVal` and `sum`. The leading 0 in 01234 tells the assembler to interpret the

number as a hex value. If no leading 0 is used, the value is interpreted as decimal.

Our assembler is a fairly simple Java program. To compile and run it type

```
javac Assembler.java
java Assembler sourceFile
```

If we do this for our little sum program, we get the output shown below.

```
16#0000# => x"1000",        -- sum = 0;
16#0001# => x"5011",
16#0002# => x"1fff",        -- while (true) {
16#0003# => x"5010",        --     inVal = -1;
16#0004# => x"1001",        --     while (inVal == -1) {}
16#0005# => x"8010",
16#0006# => x"02fe",
16#0007# => x"2010",        --     if (inVal == 0) break;
16#0008# => x"0204",
16#0009# => x"8011",        --     sum += inVal
16#000a# => x"5011",
16#000b# => x"01f7",        -- }
16#000c# => x"0000",        -- halt
16#0010# => x"1234",        -- input value
16#0011# => x"5678",        -- sum of input values
```

Each line of output takes the form of a VHDL initializer. If we insert these lines into the memory initialization section of the WASHU-2's memory component, the processor will begin executing the program whenever it is reset.

17.3 Prototyping the washu-2

To use any computer, we need to be able to interact with it. We will use a very simple mechanism to interact with the WASHU-2. First, we'll use the LCD display to show the information in the processor's registers, and to

examine the processor's memory. Here's an example of what we might see on the LCD display while running a program.

```
5010  ffff  000a
0003  0000  5011
```

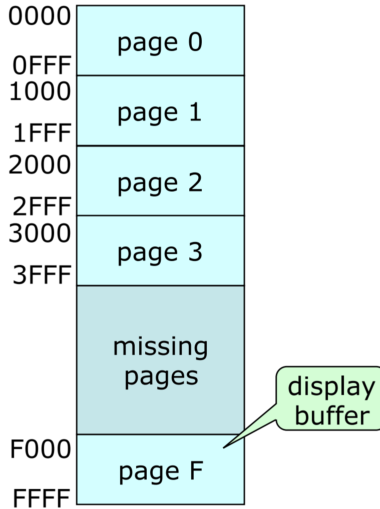
The top left number is the value in the IREG. In this case, it shows a direct store instruction. Just below it is the PC value. The next column shows the values in the accumulator and the IAR. The rightmost column shows the values of two additional registers called *snoopAdr* and *snoopData* that are used to examine and modify the contents of the memory.

We can use the knob to change the value of either *snoopAdr* or *snoopData*. Specifically, when switch number 3 is in the down position, then turning the knob changes the value in the *snoopAdr* register. This allows us to examine successive memory locations, simply by turning the knob. When switch number 3 is in the up position, turning the knob controls the value in *snoopData*. Pressing button number 1 causes the value in the *snoopData* register to be written to the location specified by the *snoopAdr* register. This allows us to enter data for use by a running program, and to observe the results of its computation.

The buttons on the prototype board are also used to control the *single step* feature of the WASHU-2 processor. By pressing button number 3, while the processor is running, we can pause the processor, allowing us to examine the values in the processor's registers (while it is running, these values are changing all the time, making it difficult to discern their values on the LCD display). By pressing button number 3 again, we can cause the processor to execute a single instruction before pausing again. Pressing button 2, causes it to go back to normal operation.

In principle, the WASHU-2 processor can handle 65,536 memory locations. Unfortunately, our prototype board has only enough memory for about 20,000 16 bit words. We'd like to use part of this memory for a display buffer, so we can drive an external video monitor. In order to preserve most of the memory for programs and data, we'll use a quarter resolution display buffer to store 160×120 pixels.

The figure below shows how the memory is organized for the prototype WASHU-2.



The memory address space has been divided into 16 logical *pages*, each consisting of 4096 words. The main memory component provides the storage for the first four pages, and a separate *vgaDisplay* component provides storage for the last page, which is used as a display buffer. There is no memory to support pages 5 through E. The *vgaDisplay* component is connected to the same memory signals as the main memory component. The main memory component responds to accesses that specify addresses in the first four pages, while the *vgaDisplay* responds to accesses that specify addresses in the last page.

Now, this *vgaDisplay* component has a slightly different client-side interface than the one we looked at in an earlier chapter. That component allowed the client to read and write individual pixels. The component we'll use here provides a client-side interface that uses 16 bit words, since that is what the WASHU-2 processor uses. Each word stores five pixel values, and the high-order bit is ignored. Programs that need to modify individual pixels in the display buffer will typically need to read a 16 bit word, modify a specific

pixel within the word, then write that word back to the display buffer.

17.4 Using Subprograms

From the early days of computing, programmers have found it useful to decompose large programs into smaller, re-usable parts. In some programming languages, these are referred to as subroutines, in others as procedures or functions. Here, we'll use the generic term subprogram to refer to a part of larger program that is designed to be "called" from other parts of the program and to return one or more results. We'll start with the example of a subprogram that implements multiplication. Since the WASHU-2 has no multiplication instruction, this will come in handy when we write programs that require multiplication.

Our multiplication subprogram is based on the long multiplication algorithm we all learned in elementary school, but adapted to binary numbers. It can be written in C-style notation as follows.

```
int mult(int a, int b) {
    prod = 0; // initialize product
    mask = 1; // mask to select bits of multiplier
    while (mask != 0) {
        if ((b & mask) != 0) // add next partial product
            prod += a;
        a <<= 1;             // shift multiplicand
        mask <<= 1;          // shift mask bit
    }
    return prod;
}
```

Let's look at an example using 6 bit arithetic, with `a=000110` and `b=000101`. At the start of the first iteration we have

```
prod = 000000  a = 000110  mask = 000001
```

At the end of the first iteration, we have

```
prod = 000110  a = 001100  mask = 000010
```

At the end of the second iteration, we have

```
prod = 000110  a = 011000  mask = 000100
```

At the end of the third, we have

```
prod = 011110  a = 110000  mask = 001000
```

The remaining iterations produce no change in `prod`, so the final result is 011110 or 30, which is the product of 6 and 5.

To implement a subprogram in assembly language, we need some convention about how a calling program passes arguments to a subprogram, and how the subprogram returns results. We will use a very simple convention, in which the arguments and return values are stored in memory locations just before the first instruction in the subprogram. Here are the first few lines of our multiplication subprogram.

```
           location 0100        -- int mult(int a, int b)
mult_a:    0                    -- first argument
mult_b:    0                    -- second argument
mult_prod: 0                    -- return value
mult_ret:  0                    -- return address
```

These lines simply label the locations where the arguments and return value will be stored. The last line defines the location of the *return address* in the calling program. This is the location in the calling program that the subroutine should branch to when it has completed its computation. To use this subprogram, a calling program will copy the arugments to the locations `mult_a` and `mult_b` and the return address to `mult_ret`. When the subprogram completes, it will leave the product in `mult_prod` and then branch to the location specified by the return address. The calling program can then copy the result from `mult_prod`.

This segment of code highlights an issue that arises with assembly language programs. With a simple assembler like ours, all the names used in the program must be unique. This makes things complicated when attempting

to write subprograms, since we want them to be re-usable in a variety of contexts. We have resolved this issue by preceding each name used by the subprogram with a unique prefix `mult_`. This will be sufficient for our purposes, but it is worth noting that more sophisticated assemblers do provide mechanisms to support more flexible and convenient use of names.

Let's move onto the actual instructions that define our subprogram. The first few just initialize `mult_prod` and `mult_mask` and start the main loop.

```
mult:       cLoad 0                 -- prod = 0;
            dStore mult_prod
            cLoad 1                 -- mask = 1;
            dStore mult_mask
mult_loop:  dLoad mult_mask         -- while (mask != 0) {
            brZero mult_end
```

The next few instructions add the next partial product to `mult_prod`, when the current bit of the multiplier is 1.

```
            dLoad mult_b            --     if ((b&mask) != 0)
            and mult_mask
            brZero 4
            dLoad mult_prod         --          prod += a;
            add mult_a
            dStore mult_prod
```

The remaining instructions finish off the loop and return to the calling program.

```
            dLoad mult_a            --     a <<= 1;
            add mult_a
            dStore mult_a
            dLoad mult_mask         --     mask <<= 1;
            add mult_mask
            dStore mult_mask
            branch mult_loop        -- }
mult_end:   iBranch mult_ret        -- return prod;
mult_mask:  0                       -- mask
```

Note the use of the indirect branch to return to the calling program. Putting it altogether, we have

```
            location 0100           -- int mult(int a, int b)
mult_a:     0                       -- first argument
mult_b:     0                       -- second argument
mult_prod:  0                       -- return value
mult_ret:   0                       -- return address
mult:       cLoad 0                 -- prod = 0;
            dStore mult_prod
            cLoad 1                 -- mask = 1;
            dStore mult_mask
mult_loop:  dLoad mult_mask         -- while (mask != 0) {
            brZero mult_end
            dLoad mult_b            --     if ((b & mask) != 0)
            and mult_mask
            brZero 4
            dLoad mult_prod         --          prod += a;
            add mult_a
            dStore mult_prod
            dLoad mult_a            --     a <<= 1;
            add mult_a
            dStore mult_a
            dLoad mult_mask         --     mask <<= 1;
            add mult_mask
            dStore mult_mask
            branch mult_loop        -- }
mult_end:   iBranch mult_ret        -- return prod;
mult_mask:  0                       -- mask
```

Next, let's take a look at another subprogram that converts a string of ASCII characters representing a decimal number into binary. Here's a C-style version.

```
int asc2bin(char* p, int n) {
    num = 0;
```

```
    while (n != 0) {
        num *= 10;
        num += (*p - '0');
        p++;
    n--;
    }
    return num;
}
```

Here, p points to the first in a sequence of memory locations that are assumed to contain ASCII character codes corresponding to decimal digits. The second argument n indicates the number of ASCII characters in the string of digits. The program considers each digit in turn, starting with the most significant digit. We subtract the ASCII code for 0 from each digit, in order to get the actual value of that digit.

We'll implement the multiplication on the fourth line of the subprogram using our multiply subprogram. Here is the first part of the program.

```
            location 0200       -- asc2bin(char* p,int n){
a2b_p:      0                   -- first argument
a2b_n:      0                   -- second argument
a2b_num:    0                   -- return value
a2b_ret:    0
asc2bin:    cLoad 0             -- num = 0;
            dStore a2b_num
a2b_loop:   dLoad a2b_n         -- while (n != 0) {
            brZero a2b_end
```

As before, we store the arguments, return value and return address right before the first instruction. The first few instructions just initialize num and start the loop. Here's the part that calls the multiply subroutine

```
            cLoad 10            --       num *= 10;
            dStore mult_a
            dLoad a2b_num
            dStore mult_b
```

```
            cLoad a2b_retLoc
            dStore mult_ret
            iBranch 1
            mult
a2b_retLoc: dLoad mult_prod
            dStore a2b_num
```

Note how this uses the locations defined earlier in the `mult` subroutine. The second constant load instruction loads the constant `a2b_retLoc`. The assembler interprets this as the address associated with the label. The second line from the end contains just the label used for the first instruction in the `mult` subroutine. The assembler interprets a line like this as specifying that the memory location corresponding to this line should contain a copy of the address associated with the label `mult`. Consequently, the indirect branch instruction just before this line, will transfer control to the first line of the subroutine. The next part of the program adds the next decimal digit into num.

```
cLoad 030           --          num += (*p - 0);
negate
dStore a2b_temp
iLoad a2b_p
add a2b_temp
add a2b_num
dStore a2b_num
```

Note the use of the indirect load instruction, to retrieve the next character. The remainder of the program follows

```
cLoad 1             --          p++;
add a2b_p
dStore a2b_p
cLoad -1            --          n--;
add a2b_n;
dStore a2b_n;
branch a2b_loop     -- }
```

```
a2b_end:    iBranch a2b_ret       -- return num;
a2b_temp:   0                     -- temporary storage
```

Putting it altogether, we have

```
            location 0200         -- int asc2bin(char* p,int n
a2b_p:      0                     -- first argument
a2b_n:      0                     -- second argument
a2b_num:    0                     -- return value
a2b_ret:    0
asc2bin:    cLoad 0               -- num = 0;
            dStore a2b_num
a2b_loop:   dLoad a2b_n           -- while (n != 0) {
            brZero a2b_end
            cLoad 10              --       num *= 10;
            dStore mult_a
            dLoad a2b_num
            dStore mult_b
            cLoad a2b_retLoc
            dStore mult_ret
            iBranch 1
            mult
a2b_retLoc: dLoad mult_prod
            dStore a2b_num
            cLoad 30              --        num += (*p  0);
            negate
            dStore a2b_temp
            iLoad a2b_p
            add a2b_temp
            add a2b_num
            dStore a2b_num
            cLoad 1               --        p++;
            add a2b_p
            dStore a2b_p
            cLoad 1              --       n--;
```

```
            add a2b_n;
            dStore a2b_n;
            branch a2b_loop     -- }
a2b_end:    iBranch a2b_ret     -- return num;
a2b_temp:   0                   -- temporary storage
```

At this point, it's interesting to look back and reflect on the processor's role in the execution of these programs. No matter how big or complicated the programs get, the processor's role remains the same. It simply fetches instructions from memory, carries out the steps required for each instruction, then does it again, and again and again. It's kind of remarkable that this relatively simple process provides the foundation on which all modern software systems rest.

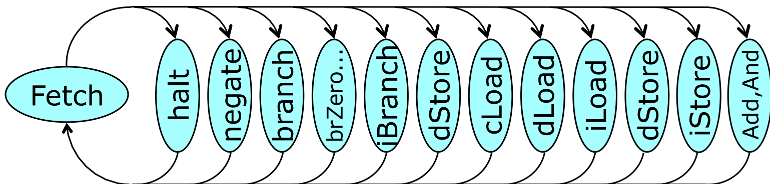
Chapter 18

Implementing a Programmable Processor

In the last chapter, we discussed the operation of the WASHU-2 processor, largely from the programmer viewpoint. In this chapter, we will take a detailed look at the circuit used to implement it.

18.1 Overview of the Implementation

Let's start with a closer look at the WASHU-2's fetch-and-execute cycle. This can be conveniently summarized in the state diagram shown below.

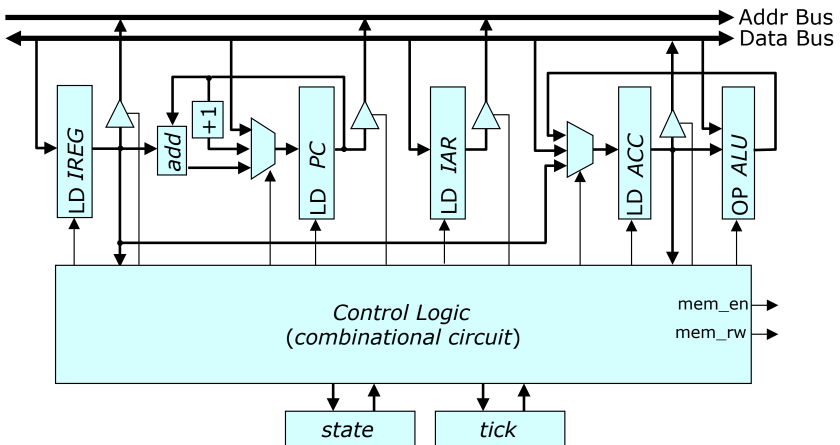


At the left end of the diagram is the processor's *fetch* state, and for each instruction there is a corresponding state that appears to the right (a few have been combined to simplify the diagram). While it is in the fetch state, the processor determines which instruction is to be executed next and switches

to the state for that instruction. Each of the instruction states returns to the fetch state once the instruction has been completed.

Most of the processor states require several clock ticks, in order to carry out all the steps that their instructions require. A separate *tick* register keeps track of the current clock tick within the current state, and the processor uses this to control the timing of the signal transitions.

The next figure is a detailed block diagram of the WASHU-2 processor, showing all the registers, their connections to the address and data bus and the controller.



Starting at the left end of the diagram, we have the IREG. Note that the input to the IREG is connected to the data bus, so that instructions retrieved from memory can be loaded into the register. The output side of the IREG also connects to the data bus through a set of tristate buffers.

The PC appears next. Note that its input can come from several sources. First, it can be loaded from the data bus (this is used during indirect branch instructions), second, it can be incremented to go to the next instruction (the most common case) and third its current value can be added to the sign-extended value obtained from the low-order eight bits of the IREG. The PC's output can be connected to the address bus through a set of tristate buffers. The IAR's connections to the bus are relatively simple. It can be loaded from

the data bus and its value placed on the address bus. The accumulator can obtain a new value from the output of the ALU, from the data bus or from the low-order 12 bits of the IREG. Its output can be connected to the data bus through a set of tristate buffers. The ALU is a combinational circuit that implements the various arithmetic and logic operations. It's *operation* input selects which operation is selected. For example, it may negate the value received from the accumulator, or add the value in the accumulator to the value on the data bus. The state and tick registers are shown at the bottom of the diagram. The remaining circuitry (labeled controller) is just a large combinational circuit.

The block diagram shows all the signals needed by the controller to coordinate the activities of the other components. To illustrate how these are used, consider what happens during the fetch portion of the fetch-and-execute cycle. To retrieve the next instruction from memory, the controller first enables the tristate buffers at the output of the PC, while raising the memory enable and read/write signals. This causes the memory to return the data value stored at the address specified by the PC. The controller causes this data value to be stored in the IREG by raising the load input of the IREG at the appropriate time. At the end of the fetch, it increments the PC, by asserting its load input, while selecting the appropriate input to the multiplexor.

18.2 Signal Timing

In this section, we're going to look closely at each of the processor states, identify the sequence of steps that must take place for each state and derive appopriate signal timings for each of these steps.

We'll start with the *fetch*, which was discussed briefly at the end of the last section. Here is a list of the steps involved in the fetch.

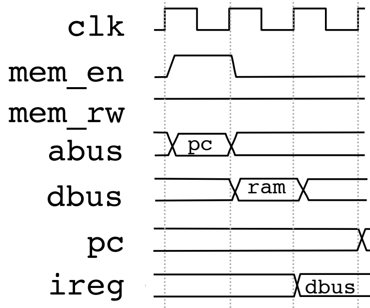
- *Initiate memory read.* This inolves enabling the tristate buffers at the output of the PC, while raising the memory enable and read/write signals.

- *Complete memory read.* This involves turning off the memory enable,

disabling the PC's tristates and loading the returned data value in the IREG.

- *Increment the* PC*.* Requires enabling the PC's load input while selecting the increment input of the PC's multiplexor.

The timing diagram shown below indicates exactly when each of these steps is done.



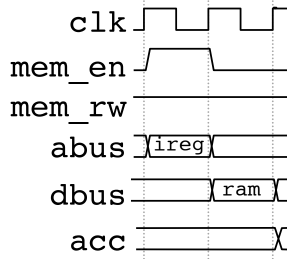
The memory read occurs when the *tick* register is zero. The data value is returned by the memory when *tick=1* and is stored in the IREG at the end tick 1. The label on the waveform for the address bus indicates that the value on the bus is coming from the PC. Similarly, the label on the data bus indicates that the value comes the RAM and the label on the IREG indicates that the value comes from the data bus.

Let's turn next to the timing for a direct load instruction. Like the fetch, this involves a memory read.

- *Initiate memory read.* Enable the tristate buffers for the low 12 bits of the IREG and the high four bits of the PC while raising the memory enable and read/write signals.

- *Complete memory read.* Turn off the memory enable, disable the tristates and load the returned data value in the accumulator.

Here's the signal timing needed to implement these steps.
The label on the address bus only mentions the IREG but as noted above, the



high order four bits of the address come from the PC. The *add* instruction and *and* instruction have exactly the same timing as the direct load, so we do not show them separately. The only difference with these instructions is that the controller must select the appropriate ALU operation and load the accumulator from the ALU, rather than directly from the data bus.

Next, let's turn to the indirect load instruction. Here are the steps required

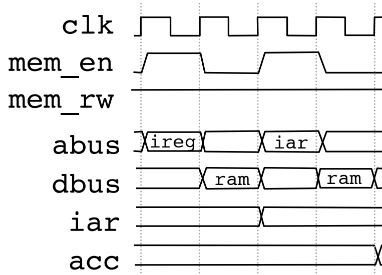
- *Initiate first memory read.* Enable the tristate buffers for the low 12 bits of the IREG and the high four bits of the PC while raising the memory enable and read/write signals.

- *Complete first memory read.* Turn off the memory enable, disable the tristates and load the returned data value in the IAR.

- *Initiate second memory read.* Enable the tristate buffers for the IAR while raising the memory enable and read/write signals.

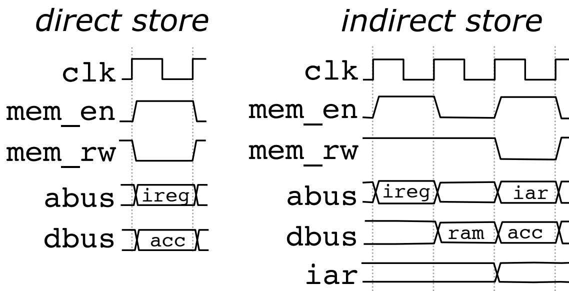
- *Complete first memory read.* Turn off the memory enable, disable the tristates and load the returned data value in the ACC.

Here is the timing used to carry out these steps.



This starts out like the direct load, but the result returned by the first memory read is stored in the indirect address register, rather than the accumulator. The second memory read, uses the IAR to supply the address. The result of this second read is then stored in the accumulator.

Now, let's look at the timing for the store instructions.



Note that the direct store requires just a single clock tick, as we do not have to wait an extra tick in order to get a reply from the memory. The indirect store starts out just like the indirect load, but finishes much like the direct store.

18.3 VHDL Implementation

Now that we've worked out the detailed signal timing, all that's left to do is look at how the implementation can be done using VHDL. We'll start with

the entity declaration.

```
entity cpu is port (
        clk, reset: in  std_logic;
        -- memory signals
        en, rw: out std_logic;
        aBus: out address; dBus: inout word;
        -- console interface signals
        pause: in std_logic;
        regSelect: in std_logic_vector(1 downto 0);
        dispReg: out word);
end cpu;
```

The memory signals constitute the main part of the processor's interface with the rest of the system. However, there are several other signals used by a *console* component that we'll describe in the next chapter. The console provides the "user-interface" to the computer. That is, it supports the display of processor registers on the LCD display and the mechanisms for single-stepping the processor and examining/modifying the contents of memory locations. The *pause* signal causes the processor to suspend execution after completing the current instruction. The *regSelect* signal is used to select one of the four processor registers. The *dispReg* signal is used to pass the value of the selected register to the console, so that it can be displayed.

The next section defines the processor state, the tick register and the signals for the registers and ALU.

```
architecture cpuArch of cpu is
type state_type is (
            resetState, pauseState, fetch,
            halt, negate,
            branch, brZero, brPos, brNeg, brInd,
            cLoad, dLoad, iLoad,
            dStore, iStore,
            add, andd
);
```

```
signal state: state_type;
signal tick: std_logic_vector(3 downto 0);

signal pc:   address; -- program counter
signal iReg: word;    -- instruction register
signal iar:  address; -- indirect address register
signal acc:  word;    -- accumulator
signal alu:  word;    -- alu output
```

Next, we have several auxiliary signals, that are used internally.

```
-- address of instruction being executed
signal this: address;
-- address used for direct load, store, add, andd, ...
signal opAdr: address;
-- target for branch instruction
signal target: word;
begin
    opAdr <= this(15 downto 12) & ireg(11 downto 0);
    target <= this + ((15 downto  8 => ireg(7))
                      & ireg( 7 downto 0));
```

The signal called *this* is the address of the instruction currently being executed, or that has just completed execution. This may seem superfluous, since the PC contains the address of the current instruction. However, since the PC is incremented at the end of the fetch state, its value is generally "off-by-one" during the execution phase of the instruction. The *opAdr* signal is formed from the high four bits of *this* and the low-order 12 bits of IREG. This is the address used by several instructions, including *direct load*, *direct store* and *add*. The *target* signal is used by the branch instructions. It is formed by sign-extending the low eight bits of the IREG and then adding this quantity to *this*.

Moving on, we hve the following assignments.

```
    -- connect selected register to console
    with regSelect select
```

```
        dispReg <= iReg  when "00",
                   this  when "01", -- this instead of pc
                   acc   when "10",
                   iar   when others;
    -- select alu operation based on state
    alu <=     (not acc) + x"0001" when state = negate else
                acc + dbus          when state = add    else
                acc and dbus        when state = andd   else
                (alu'range => '0');
```

The *dispReg* output selects one of the four processor registers based on the value of *regSelect*. Note that in place of the PC, it returns the value of *this*. When the processor suspends operation in response to the *pause* input, it does so after the instruction has completed execution, when the value in the PC no longer corresponds to the current instruction.. At this point in the fetch-and-execute cycle, the IREG shows the current instruction and *this* shows its address.

The ALU is defined by the selected assignment statement at the end of this segment. Because the WASHU-2 has such a limited instruction set, there is not much to the ALU in this case. In processors with more extensive instruction sets, the ALU forms a much more substantial part of the processor.

The remainder of the WASHU-2 specification consists of two processes. The first is a synchronous process that controls the processor's registers, including *state* and *tick*. The second is a combinational process that controls the memory signals, using *state* and *tick*. We'll start with the synchronous process, specifically the part that handles the decoding of instructions.

```
    -- synchronous process controlling state, tick and processor
    process (clk)

    function decode(instr: word) return state_type is begin
        -- Instruction decoding.
        case instr(15 downto 12) is
        when x"0" =>
            case instr(11 downto 8) is
```

```
        when x"0" =>
            if instr(11 downto 0) = x"000" then
                return halt;
            elsif instr(11 downto 0) = x"001" then
                return negate;
            else
                return halt;
            end if;
        when x"1" =>     return branch;
        when x"2" =>     return brZero;
        when x"3" =>     return brPos;
        when x"4" =>     return brNeg;
        when x"5" =>     return brInd;
        when others => return halt;
        end case;
    when x"1" =>     return cLoad;
    when x"2" =>     return dLoad;
    when x"3" =>     return iLoad;
    when x"5" =>     return dStore;
    when x"6" =>     return iStore;
    when x"8" =>     return add;
    when x"c" =>     return andd;
    when others => return halt;
    end case;
end function decode;
```

The instruction decoding is implemented by the *decode* function, which simply returns the state for the instruction specified by its argument. Observe how the case statements directly correspond to the numerical instruction encodings discussed in the previous chapter.

Next, we hae a simple utility procedure that is invoked at the end of every instruction.

```
procedure wrapup is begin
    -- Do this at end of every instruction
```

```
        if pause = '1' then
            state <= pauseState;
        else
            state <= fetch; tick <= x"0";
        end if;
    end procedure wrapup;
```

Note that it enters the pause state when the console raises the pause input. The processor always completes the current instruction before reacting to the pause input, so the console is expected to hold it high long enough for any in-progress instruction to complete. Now, let's move onto the body of the process.

```
    begin
          if rising_edge(clk) then
              if reset = '1' then
                state <= resetState; tick <= x"0";
                pc <= (others => '0');   this <= (others => '0');
                iReg <= (others => '0'); acc <= (others => '0');
                iar <= (others => '0');
            else
                tick <= tick + 1; -- advance time by default
                if state = resetState then
                    state <= fetch; tick <= x"0";
                elsif state = pauseState then
                    if pause = '0' then
                        state <= fetch; tick <= x"0";
                    end if;
                elsif state = fetch then
                    if tick = x"1" then
                        iReg <= dBus;
                    elsif tick = x"2" then
                        state <= decode(iReg); tick <= x"0";
                        this <= pc; pc <= pc + 1;
                    end if;
```

When *reset* is dropped, the processor enters a transient *resetState* that simply initializes the *state* and *tick* registers. Whenever the processor is in the *pause* state, it checks to see if *pause* has dropped low, and if so, moves onto the *fetch* state. When in the fetch state, the processor performs a memory read. This is controlled by the asynchronous process responsible for the memory, which we'll get to shortly. The memory returns a value on the data bus during clock tick 1, and this value is stored in the IREG at the end of clock tick 1. At the end of clock tick 2, the *state* register is updated, and *tick* is set to zero, to take the processor into the execution phase. In addition, the program counter is updated along with *this*.

Next, we have the code that implements the execution phase of of the branch instructions.

```
else
    case state is
    -- branch instructions
    when branch =>
        pc <= target; wrapup;
    when brZero =>
        if acc = x"0000" then
            pc <= target;
        end if;
        wrapup;
    when brPos =>
        if acc(15)='0' and acc /= x"0000" then
            pc <= target;
        end if;
        wrapup;
    when brNeg =>
        if acc(15) = '1' then
            pc <= target;
        end if;
        wrapup;
    when brInd =>
        if tick = x"1" then
```

```
            pc <= dBus; wrapup;
        end if;
```

Most of these instructions take just a single clock tick. The one exception is the indirect branch, which must wait for the memory to return the address of the branch destination during tick 1. Again, note that the asynchronous process is responsible for initiating the memory operation. Here, we simply store the returned value in the PC. Next, we have the memory load instructions

```
    -- load instructions
    when cload =>
        acc <= (15 downto 12 => ireg(11))
                & ireg(11 downto 0);
        wrapup;
    when dload =>
        if tick = x"1" then
            acc <= dBus; wrapup;
        end if;
    when iload =>
        if tick = x"1" then
            iar <= dBus;
        elsif tick = x"3" then
            acc <= dBus; wrapup;
        end if;
```

Observe that at the end of tick 1 of the indirect load, the processor stores the value returned by the memory in the IAR, while at the end of tick 3, it stores the value returned by the second memory read in the ACC. Finally, we have the store instructions, followed by the arithmetic and logic instructions

```
    -- store instructions
    when dstore => wrapup;
    when istore =>
        if tick = x"1" then iar <= dBus;
        elsif tick = x"2" then wrapup;
        end if;
```

```
                    -- arithmetic and logic instructions
                    when negate => acc <= alu; wrapup;
                    when add | andd =>
                        if tick = x"1" then
                            acc <= alu; wrapup;
                        end if;
                    when others => state <= halt;
                    end case;
                end if;
            end if;
          end if;
    end process;
```

Note that the *add* and *and* instructions can share a case since they differ only in the operation used by the ALU.

Next, let's look at the asynchronous process that controls the memory.

```
    -- Memory control process (combinational)
    process (ireg,pc,iar,acc,this,opAdr,state,tick) begin
        -- default values for memory control signals
        en <= '0'; rw <= '1';
        aBus <= (others => 'Z'); dBus <= (others => 'Z');
        case state is
        when fetch =>
            if tick = x"0" then
                en <= '1'; aBus <= pc;
            end if;
        when brInd =>
            if tick = x"0" then
                en <= '1'; aBus <= target;
            end if;
```

The default assignments leave the memory disabled. During tick 0 of the fetch state, the memory enable is raised high, while the value in the PC is placed on the address bus. (Note, that the read/write signal is high by default.) During tick 0 of the indirect branch, the memory enable is raised

high again, while the *target* signal is placed on the address bus. The next segment handles the load and arithmetic instructions.

```
        when dLoad | add | andd =>
            if tick = x"0" then
                en <= '1'; aBus <= opAdr;
            end if;
        when iLoad =>
            if tick = x"0" then
                en <= '1'; aBus <= opAdr;
            elsif tick = x"2" then
                en <= '1'; aBus <= iar;
            end if;
```

Observe that two memory reads take place during the indirect load. The first uses the *opAdr* signal as the memory address, while the second uses the IAR. Finally, we have the memory operations for the two store instructions.

```
        when dStore =>
            if tick = x"0" then
                en <= '1'; rw <= '0';
                aBus <= opAdr; dBus <= acc;
            end if;
        when iStore =>
            if tick = x"0" then
                en <= '1'; aBus <= opAdr;
            elsif tick = x"2" then
                en <= '1'; rw <= '0';
                aBus <= iar; dBus <= acc;
            end if;
        when others =>
        end case;
    end process;
end cpuArch;
```

Note that these instructions both drop the read/write signal low in order to store the value in the accumulator to memory.

That wraps up our implementation of the WASHU-2 processor. It's worth noting that there is not really that much to it. The entire processor specification takes less than 250 lines of VHDL, and the circuit implemented by the spec uses just 106 flip flops and 387 LUTs. This is about 1% of the flip flops on the prototype board's FPGA and about 4% of the LUTs. Now, it must be acknowledged that the performance of this processor is relatively limited. It can execute only about 10 million instructions per second, while the processors in our laptops can execute over a billion instructions per second. Nonetheless, it exhibits the same kind of flexibility as more sophisticated processors, because it shares with them, the key attribute of programmability.

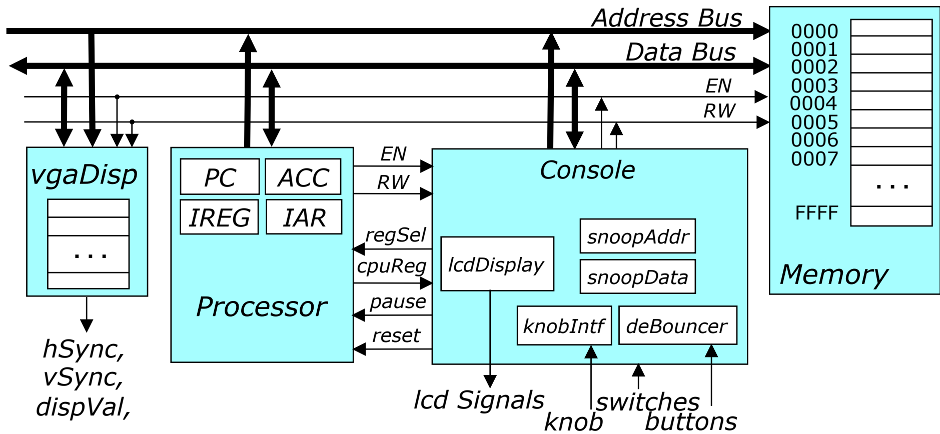
Chapter 19

Supporting Components

In this chapter, we're going to finish up our discussion of the WASHU-2 processor with a look at the supporting components that allow us to interact with running programs, when the processor is implemented on our prototype boards.

19.1 Overview of the Complete System

Let's start with a block diagram that shows all the major system components.



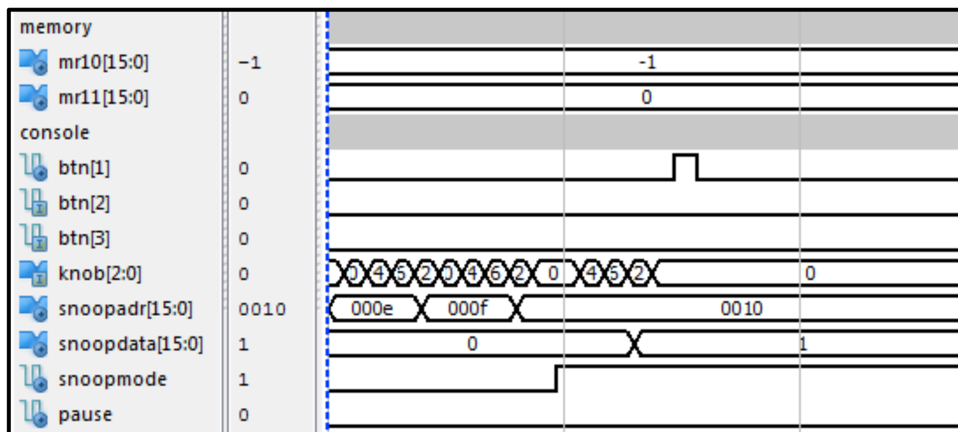
At the left end of the diagram, we have a *vgaDisplay* component similar to the circuit presented in an earlier chapter. The main difference is that this version contains a display buffer with 16 bit words, each containing five pixels. It shares the bus with the main memory and responds to memory read/writes for addresses that are greater than or equal to `f000`. The main memory component handles reads/writes for addresses up to `3fff`. The remaining memory addresses are ignored, since the prototype boards do not not have enough memory to support the full range of addresses. Programs running on the processor can display information on an external monitor simply by writing the appropriate pixel values into the *vgaDisplay* component's display buffer. The display resolution is one quarter the standard VGA resolution, so there is a total of 120 lines, where each line has 160 pixels. Since there are five pixels per word, each line corresponds to 32 words in the display buffer.

The *Console* component allows us to single-step the processor, to observe the values in the processor registers or selected memory locations. It also allows us to write data into memory, providing a crude, but effective mechanism for supplying data to a running program. The prototype board's buttons, knob and switches are used to control the interaction with the processor. The LCD display is used to show the registers and memory contents.

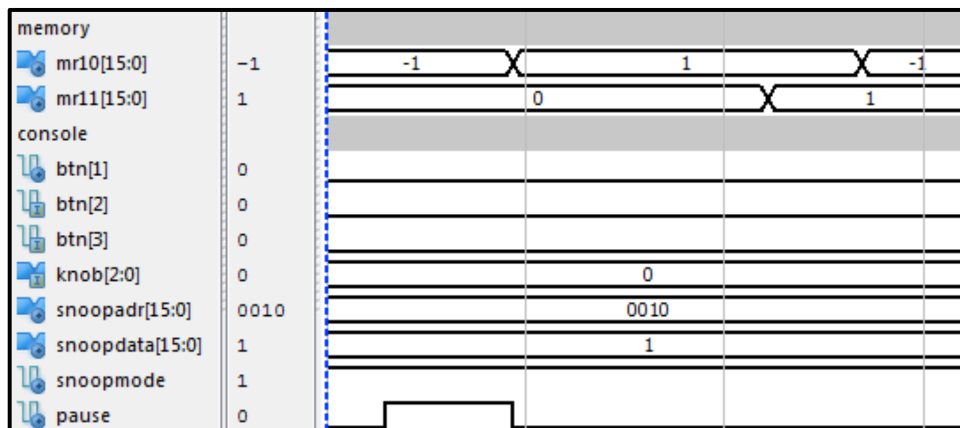
Notice that the processor's memory *enable* and *read/write* signals actually go to the console. Most of the time, the console simply propagates these

signals, but in order to implement the memory snooping functions, the console must be able to take control of the memory away from the processor. Routing these signals through the console allows it to do that. The console uses the *pause* signal to suspend the normal processor operation when in single-step mode. This is also used to interrupt the processor in order to perform the memory snooping functions. The *regSelect* and *cpuReg* signals are used by the console to obtain the current values of the the processor's registers, so that they can be shown on the LCD display.

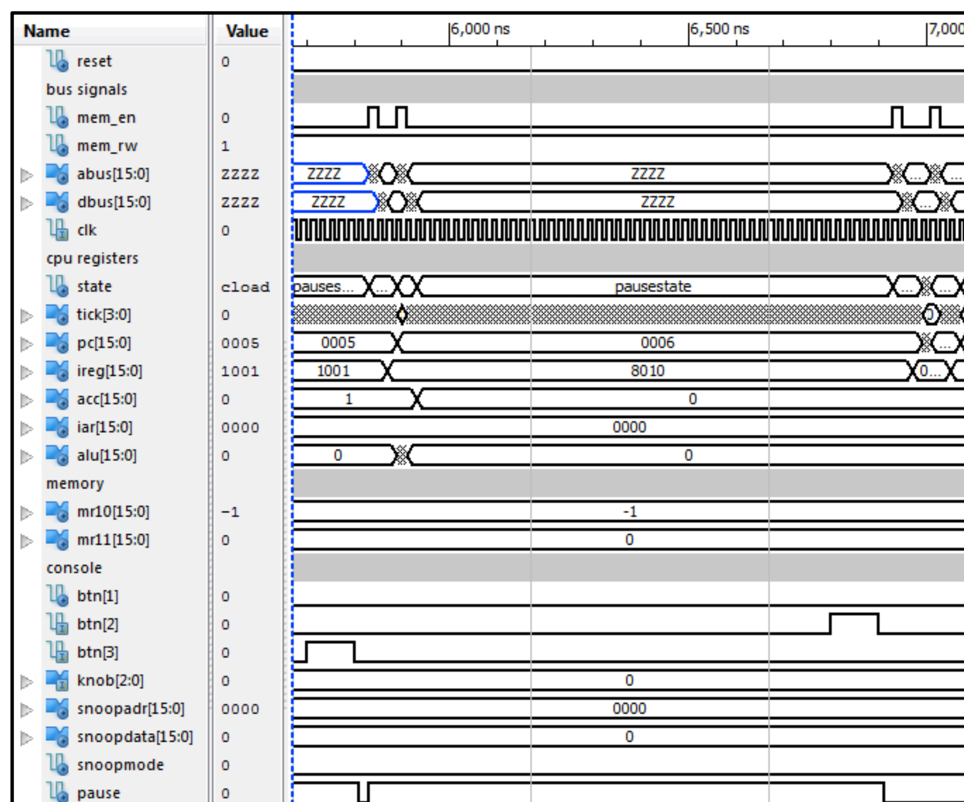
Before proceeding to the implementation of the console, let's take a look at a simulation that demonstrates its various functions. This first segment shows how turning the knob controls the *snoopAdr* and *snoopData* registers.



Note where the *snoopMode* changes from 0 to 1, causing the role of the knob to change. Also, observe the button press about halfway through the segment. This triggers a memory write, which we see in the next segment, shown below.



Note that the pause signal goes high at the start of this segment. At the end of the pause interval, the value in the *snoopData* register is written to memory location 0010. The subsequent changes to memory locations 0010 and 0011 are caused by the running program, which adds the value in location 0010 to the value in location 0011, then changes the value in 0010 back to -1. The next segment demonstrates the single-step feature.



At the start of this segment, the processor is paused. Then, button 3 is pressed, causing the pause signal to drop for one clock tick, allowing the processor to start the next instruction. Once started, it continues until the instruction has completed, then pauses again. Near the end of this segment, button 2 is pressed, causing the pause signal to drop and normal operation to resume.

19.2 Implementing the Console

In this section, we'll go through the details of the console implementation. Let's start with the entity declaration.

```
entity console is port(
    clk: in std_logic;
    resetOut: out std_logic;
    -- inputs from user
    btn: in buttons;
    knob: in knobSigs;
    swt: in switches;
    -- memory bus signals
    memEnOut, memRwOut: out std_logic;
    aBus: out word;
    dBus: inout word;
    -- processor interface signals
    memEnIn, memRwIn: in std_logic;
    pause: out std_logic;
    regSelect: out std_logic_vector(1 downto 0);
    cpuReg: in word;
    -- signals for controlling LCD display
    lcd: out lcdSigs);
end console;
```

The *resetOut* signal is just a debounced version of button 0. Next, we have component declarations, and a number of signal declarations.

```
architecture a1 of console is
component debouncer ..  end component;
component knobIntf .. end component;
component lcdDisplay .. end component;

signal dBtn, prevDBtn: buttons;
signal reset: std_logic;

-- single step control signal
signal singleStep: std_logic;

-- local signals for controlling memory
```

```
signal memEn, memRw: std_logic;

-- signals for controlling snooping
signal snoopAdr: address; signal snoopData: word;
signal snoopCnt: std_logic_vector(6*operationMode + 9 downto 0);
signal snoopMode, snoopTime, writeReq: std_logic;

-- signals for controlling input from knob, buttons
signal tick, clockwise : std_logic;
signal delta: word;
```

The *singleStep* signal is high whenever the processor is in single step mode. During periods when snooping operations are being performed, the internal memory control signals *memEn* and *memRw* determine the values of the output signals *memEnOut* and *memRwOut*.

The snooping functions are controlled using the *snoopAdr* and *snoopData* registers, plus a counter called *snoopCnt*. The *snoopMode* signal is controlled by switch number 3. When it is low, the console is in "snoop out" mode, meaning that the *snoopData* register shows the value in the memory location specified by *snoopAdr*. In this mode, the knob controls the value of *snoopAdr*. When *snoopMode* is high, we're in "snoop in" mode, meaning that the value in *snoopData* is written to memory whenever the user presses button number 1. In this mode, the knob controls the value of *snoopData*.

The *snoopTime* signal goes high periodically for an interval lasting 16 clock ticks. During this interval, the processor *pause* signal is high, causing the processor to suspend execution after it completes the instruction currently in progress. At the end of the *snoopTime* interval, the console either loads the *snoopData* register from the memory location specified by *snoopAdr* or it writes the value currently in *snoopData* to that memory location. The *writeReq* signal is raised when the user presses button 1 on the prototype board, to request that the value in *snoopData* be written to memory. The memory write does not actually take place until the next *snoopTime* interval. The *tick*, *clockwise* and *delta* signals come from an internal instance of the knob interface component we have used before. The next few signals are used to control the updating of the LCD display.

```
-- counter for controlling when to update lcdDisplay
constant CNTR_LENGTH: integer := 8 + operationMode*12;
signal lcdCounter: std_logic_vector(CNTR_LENGTH-1 downto 0);
signal lowBits: std_logic_vector(CNTR_LENGTH-6 downto 0);

-- signals for controlling lcdDisplay
signal update: std_logic;
signal selekt: std_logic_vector(4 downto 0);
signal nuChar: std_logic_vector(7 downto 0);

type hex2asciiMap is array(0 to 15) of character;
constant hex2ascii: hex2asciiMap :=
    ( 0 => '0',  1 => '1',  2 => '2',  3 => '3',  4 => '4',
      5 => '5',  6 => '6',  7 => '7',  8 => '8',  9 => '9',
     10 => 'a', 11 => 'b', 12 => 'c', 13 => 'd', 14 => 'e',
     15 => 'f');
```

The console uses an instance of the *lcdDisplay* module we've used previously, so these signals should look familiar. Next, let's move onto the body of the architecture.

```
begin
    -- connect all the sub-components
    db:   debouncer generic map(width => 4)
                    port map(clk, btn, dBtn);
    kint: knobIntf port map(clk, reset, knob,
                            tick, clockwise, delta);
    disp: lcdDisplay port map(clk, reset, update,
                              selekt, nuchar, lcd);
    reset <= dBtn(0); resetOut <= reset;

    -- process for controlling single step operation
    process(clk) begin
        if rising_edge(clk) then
            prevDBtn <= dBtn;
```

```
            if reset = '1' then
                singleStep <= '0';
            else
                if dBtn(3) > prevDBtn(3) then
                    singleStep <= not singleStep;
                elsif dBtn(3) = '1' then
                    singleStep <= '1';
                elsif dBtn(2) > prevDBtn(2) then
                    singleStep <= '0';
                end if;
            end if;
        end if;
    end process;
```

The single step process determines the value of the *singleStep* signal, based on button pushes from the user. Button 3 is pressed either to enter single-step mode, or to execute one instruction at a time. Whenever the processor is not in single step mode, pressing the button causes it to enter single step mode. If the processor is already in single step mode when the button is pressed, it will leave single step mode for a single clock tick before returning to single step mode. This allows the processor to start the next instruction, and once it starts, it will continue until that instruction has been completed. Whenever button 2 is pressed, the processor leaves single step mode.

Next we have assignments that control several different signals.

```
    memEnOut <= memEnIn or memEn;
    memRwOut <= memRwIn and memRw;
    snoopTime <= '1' when snoopCnt(snoopCnt'high downto 4) =
                                (snoopCnt'high downto 4 => '1')
                     else '0';
    pause <= singleStep or snoopTime;
    snoopMode <= swt(3); -- when low, we're in "snoop out" mode
                         -- when high, we're in "snoop in" mode
```

The memory control outputs are a combination of the memory control signals received from the processor and the internal memory control signals used

when performing snooping functions. Either the processor or the console can raise the *memEnOut* signal. Similarly, either can pull the *memRwOut* signal low. The console never attempts to use the memory while the processor is executing instructions, so most of the time the signals from the processor are simply propagated through to the memory.

The *snoopTime* signal is asserted periodically, based on the value of *snoopCnt* and remains high for 16 clock ticks at a time. The *pause* signal is raised high whenever we're in single step mode or the *snoopTime* signal has been raised.

Next, we'll look at the implementation of the snooping functions. There are two processes that control snooping. The first is a synchronous process that controls the snooping registers, plus the *writeReq* signal. The second is a combinatorial process that defines the memory control signals.

```
-- process that controls snoop registers and writeReq
process(clk) begin
    if rising_edge(clk) then
        if reset = '1' then
            snoopAdr <= (others => '0');
            snoopData <= (others => '0');
            snoopCnt <= (others => '0');
            writeReq <= '0';
        else
            snoopCnt <= snoopCnt + 1;
            -- raise writeReq when button pushed
            if dBtn(1) > prevDBtn(1) and snoopMode = '1'
            then
                writeReq <= '1';
            end if;
            -- load snoopData at end of snoopTime period
            if snoopTime = '1' and snoopMode = '0' then
                if snoopCnt(3 downto 0) = x"d" then
                    snoopData <= dBus;
                end if;
            end if;
```

```
                if writeReq = '1' and snoopTime = '1' and
                       snoopCnt(3 downto 0) = x"f" then
                    writeReq <= '0';
                end if;
                -- update snoopAdr or snoopData from knob
                if tick = '1' then
                    if snoopMode = '0' then
                        if clockwise = '1' then
                            snoopAdr <= snoopAdr + delta;
                        else
                            snoopAdr <= snoopAdr - delta;
                        end if;
                    else
                        if clockwise = '1' then
                            snoopData <= snoopData + delta;
                        else
                            snoopData <= snoopData - delta;
                        end if;
                    end if;
                end if;
            end if;
        end if;
    end process;
```

Note that the *writeReq* signal is raised high when a button is pushed and *snoopMode* is high, meaning that the console is in “snoop in” mode. Next, observe that when the console is in “snoop out” mode, the value on the data bus is stored in *snoopData* near the end of the *snoopTime* interval. The actual memory read is controlled by a separate process that we’ll get to shortly. Also, observe that the *writeReq* signal is cleared at the end of the *snoopTime* period. The last part of this process simply updates either *snoopAdr* or *snoopData*, depending on the value of *snoopMode*.

Next, we have the process that actually generates the memory signals.

```
    -- process controlling memory signals for snooping
```

```
process (snoopTime, snoopCnt, snoopData, snoopAdr) begin
    memEn <= '0'; memRw <= '1';
    aBus <= (others => 'Z'); dBus <= (others => 'Z');
    if snoopTime = '1' then
        -- allow time for in-progress instruction to finish
        if snoopCnt(3 downto 0) = x"c" then
            memEn <= '1'; aBus <= snoopAdr;
        elsif writeReq = '1' and
                snoopCnt(3 downto 0) = x"f" then
            memEn <= '1'; memRw <= '0';
            aBus <= snoopAdr; dBus <= snoopData;
        end if;
    end if;
end process;
```

Observe that memory operations are performed only near the end of the *snoopTime* interval, after the processor has had enough time to complete its current instruction. The read operation is always performed, but as we saw earlier, the value returned by the memory is only stored in the *snoopData* register when *snoopMode* is low. If a memory write has been requested, it is performed during the last clock tick of the *snoopTime* interval.

Next, we have the part of the console that controls the LCD display.

```
-- process to increment lcdCounter
process(clk) begin
    if rising_edge(clk) then
        if reset = '1' then lcdCounter <= (others => '0');
        else lcdCounter <= lcdCounter + 1;
        end if;
    end if;
end process;

-- update LCD display to show cpu and snoop registers
-- first row:  ireg acc snoopAdr
-- second row:  pc  iar snoopData
```

```
lowBits <= lcdCounter(CNTR_LENGTH-6 downto 0);
update <= '1' when lowBits = (lowBits'range => '0')
           else '0';
selekt <= lcdCounter(CNTR_LENGTH-1 downto CNTR_LENGTH-5);

regSelect <= "00" when selekt <= slv(4,5) else
             "10" when selekt <= slv(10,5) else
             "01" when selekt <= slv(20,5) else
             "11";
```

The *selekt* signal selects one of the character positions on the LCD display. The circuit displays the IREG in the first four positions of the first row (*selekt* in the range 0 to 3); it displays the accumulator in the middle four positions of the first row, (*selekt* in the range 6 to 9); it displays the PC in the first four positions of the second row, (*selekt* in the range 16 to 19); and finally, it displays the IAR in the middle four positions of the second row, (*selekt* in the range 22 to 25). The *regSelect* signal chooses the appropriate processor register for each range of *selekt* values.

The next process determines the value of the *nuChar* signal. This specifies the character to be written on the LCD display at the position specified by *selekt*. During each clock tick, it selects a hex digit from one of the processor registers, or from *snoopAdr* or *snoopData*. The selected hex digit is converted to the corresponding ASCII character and this character is then converted to type *byte* (the *c2b* function does this trivial type conversion).

```
process (cpuReg, snoopAdr, snoopData, selekt) begin
    case selekt is
    -- high nibble of processor registers
    when "00000" | "00110" | "10000" | "10110" =>
        nuChar <= c2b(hex2Ascii(int(cpuReg(15 downto 12))));
    -- second nibble of processor registers
    when "00001" | "00111" | "10001" | "10111" =>
        nuChar <= c2b(hex2Ascii(int(cpuReg(11 downto 8))));
    -- third nibble of processor registers
    when "00010" | "01000" | "10010" | "11000" =>
```

```
            nuChar <= c2b(hex2Ascii(int(cpuReg(7 downto 4))));
        -- low nibble of processor registers
        when "00011" | "01001" | "10011" | "11001" =>
            nuChar <= c2b(hex2Ascii(int(cpuReg(3 downto 0))));

        -- nibbles of snoopAdr register
        when "01100" => nuChar <= c2b(hex2Ascii(int(
                                     snoopAdr(15 downto 12))));
        when "01101" => nuChar <= c2b(hex2Ascii(int(
                                     snoopAdr(11 downto 8))));
        when "01110" => nuChar <= c2b(hex2Ascii(int(
                                     snoopAdr(7 downto 4))));
        when "01111" => nuChar <= c2b(hex2Ascii(int(
                                     snoopAdr(3 downto 0))));
        -- nibbles of snoopData register
        when "11100" => nuChar <= c2b(hex2Ascii(int(
                                     snoopData(15 downto 12))));
        when "11101" => nuChar <= c2b(hex2Ascii(int(
                                     snoopData(11 downto 8))));
        when "11110" => nuChar <= c2b(hex2Ascii(int(
                                     snoopData(7 downto 4))));
        when "11111" => nuChar <= c2b(hex2Ascii(int(
                                     snoopData(3 downto 0))));
        -- space characters everywhere else
        when others => nuChar <= c2b(' ');
        end case;
    end process;
end a1;
```

This brings us to the end of our presentation of the WASHU-2 processor and its supporting components. It's interesting to note that the amount of VHDL code needed to specify the console is actually a bit larger than the amount that was needed to specify the processor. The amount of actual hardware needed is roughly comparable. While the processor requires 106 flip flops plus 387 LUTs, the console uses 123 flip flops and 362 LUTs. The complete

system, including *vgaDisplay* and the main memory component uses 278 flip flops and 935 LUTs.

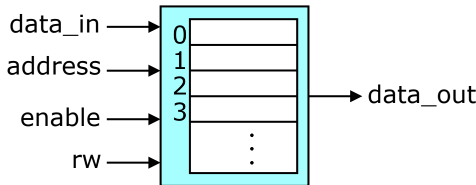
Chapter 20

Memory Components

In this chapter, we're going to take a closer look at how memory components are implemented, and the factors that limit their performance. In the next two chapters, we'll see how the performance of memory systems affects the performance of programmable processors.

20.1 SRAM Organization and Operation

Let's start with a reminder of how a memory component operates. The figure below shows a typical *asynchronous* memory component.

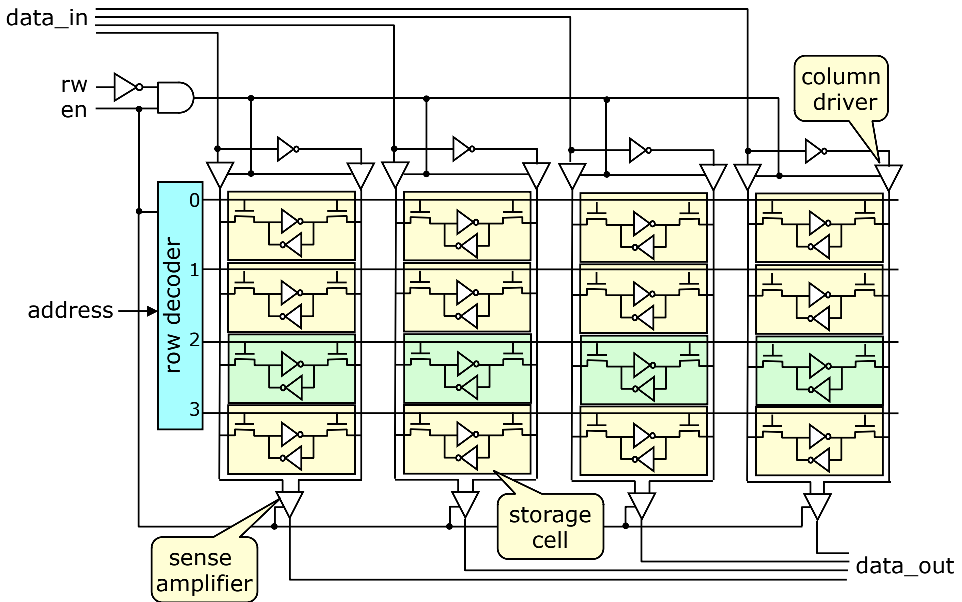


Conceptually, the memory is just an array of numbered storage locations, each capable of storing a *word* of some fixed size. The numbers used to identify the storage locations are referred to as the addresses of those locations. This type of memory is referred to as a *Random Access Memory* (RAM) because we can access different memory locations in arbitrary order with no

impact on performance.

The RAM component has an *enable* input, a *read/write* input, an *address* input and separate *data* input and output signals. To perform a memory read operation, we provide an address, then raise the enable and read/write signals high. The component then responds with the value stored at the specified location. To perform a memory write, we provide an address and data value, raise the enable signal and then drop the read/write signal low. This causes the memory component to store the specified value in the specified location.

The figure below shows an implementation of a small *static* RAM(SRAM).



This particular SRAM stores four words, each of which is four bits long. At the center of the diagram is an array of storage cells, each of which stores a single bit of data. Each storage cell contains a pair of inverters and is accessed using a pair of transistors. When the transistors are turned off, the cell is isolated, and the inverter outputs have complementary values. Either the top inverter's output is 0 and the bottom inverter's output is

1, or the top inverter's output is 1 and the bottom inverter's output is 0. These two possibilities are referred to as the *storage states* of the memory cell. Notice that each memory cell requires a total of six transistors. We could also implement a storage cell using an SR-latch with a control input. Such a circuit can be implemented with four NAND gates, giving a total of 16 transistors. Thus, the SRAM cell uses less than half the number of transistors as a latch-based storage cell.

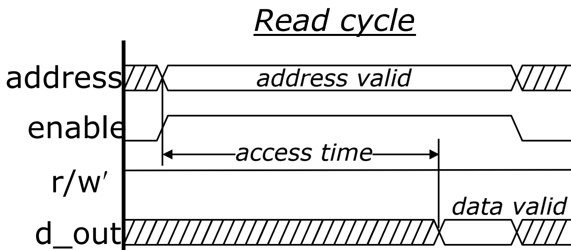
Now, it's important to keep in mind that typical memory components have much larger storage arrays than the one shown here. It's common to have hundreds of rows and columns a single storage array. We're showing a very small example to illustrate the essential features, but keep in mind that in real memory components, the scale is very different.

To read a word from the memory, we supply an address and raise the enable and read/write signals high. This causes one of the outputs of the row decoder to go high, which causes the access transistors in the storage cells in one row to turn on. When the transistors turn on, the inverters in the storage cell are connected to the vertical data lines in each of the four columns. Because the inverters in each cell have complementary output values, one of the two data lines in each column will go high, while the other will go low. Now, because a typical storage array has hundreds of rows and columns, there is a large capacitance associated with the vertical data lines. Recall that large capacitances lead to relatively slow signal transitions. To make matters worse, the transistors used in the storage cells are kept as small as possible to maximize storage density, and small transistors have relatively high resistance, making the signal transitions even slower. To speed up the read-out process, there is a *sense amplifier* at the bottom of each column, which amplifies the difference in voltage on the two data lines. This makes it unnecessary to wait until the inverters in the storage cell are able to drive the voltage on the vertical lines all the way high or low. In this example, the sense amplifiers have tristate outputs which are controlled by the memory enable input.

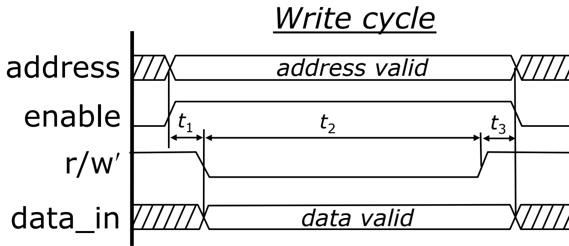
To write a word to memory, we provide address and data, raise the enable and then drop the read/write signal low. This allows the input data to flow through the tristate column drivers to the vertical data lines in each

column. What happens when the column drivers for a given column are changing the value stored in the selected cell within that column? When the access transistors for that cell are turned on, the column drivers and the inverters within the cell have their outputs connected together, through the access transistors. Normally, we would not want to have two different gates simultaneously trying to drive the same wire, but this is an exception to the usual rule. The transistors in the column drivers are designed to be much more powerful than the transistors in the inverters (essentially this means that the on-resistance of the column drivers' transistors are much smaller than the on-resistance of the transistors in the inverters). This allows the column drivers to quickly force the pair of inverters to switch from one storage state to the other.

Now, let's take a closer look at the operation of the memory, starting with the read operation.



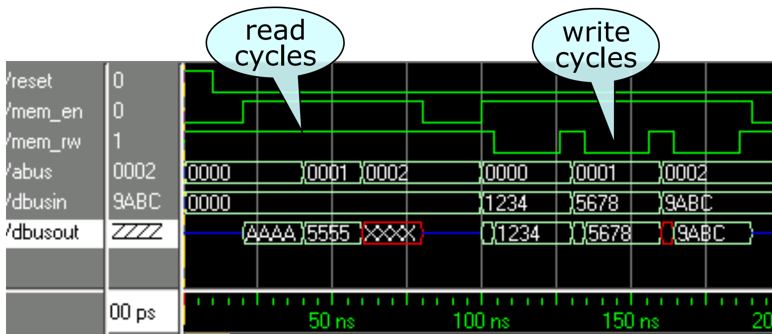
Note that there is a delay between the time the enable signal goes high and the time that the data is available at the output. This delay is a combination of the combinational circuit delay in the row decoder and the time required for the inverters in the selected storage cells to drive the voltages on the vertical data lines far enough apart so that the outputs of the sense amplifiers are all the way high or all the way low. This interval is called the *access time* of the memory. Any circuit that uses the memory must wait until this much time has passed before attempting to use the output of the memory. Also note that the address inputs must remain stable while the memory access is taking place. Now, let's take a look at the write operation.



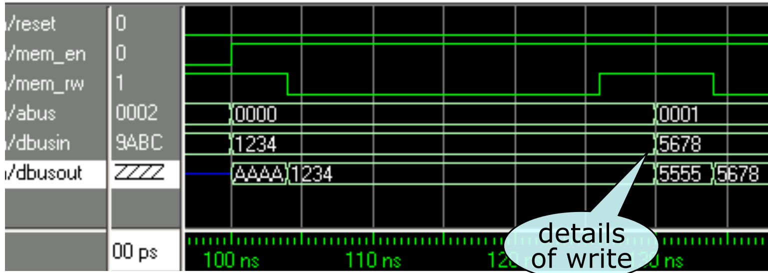
The timing for the write is a little more complicated than the timing for the read. Here, it's important that the address is valid and the enable is high before the read/write signal goes low. Also, the read/write signal must go high again before the address lines can change and the enable drops low. The time periods labeled t_1, t_2 and t_3 are the minimum allowed times between the indicated signal transitions. If these constraints are not adhered to, it's possible that a memory write may affect some memory location other than the one that we're trying to change. The sum of the three time intervals is called the *memory cycle time*.

Together, the access time and the cycle time characterize the performance of the memory component. The access time imposes a limit on how often we can read from the memory. The cycle time imposes a limit on how often we can write.

Next, let's look at a simulation of the operation of an asynchronous RAM. We'll start with a functional simulation that does not include circuit delays.

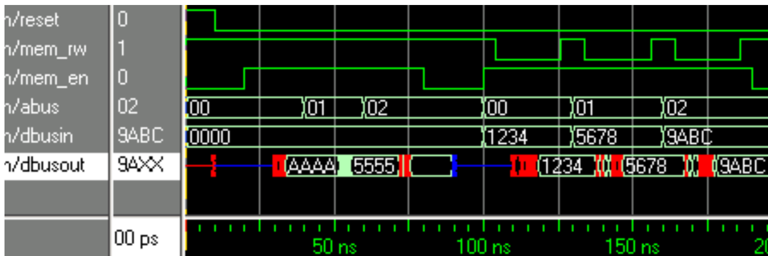


Let's start with the read cycles. When the enable first goes high, the address starts out at 0000 and then changes to 0001. The data output returns stored values AAAA and 5555. When the adrress changes to 0002 the output becomes XXXX because this memory location was not initialized in the simulation. Notice that during the write cycles, the address is only changed when the read/write signal is high. Here's a closer look at the first write cycle.

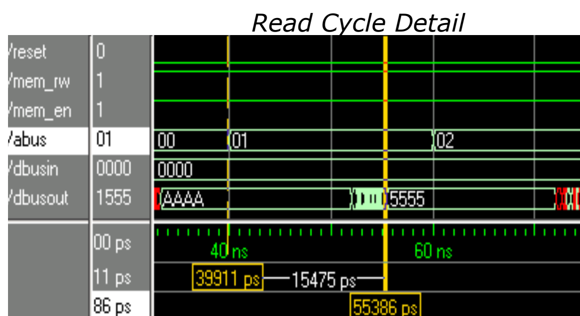


We see that before the read/write signal goes low, the data output shows the value currently stored in the memory. When the read/write signal drops, the output signal follows the input. The new value is retained when the read/write signal goes high again.

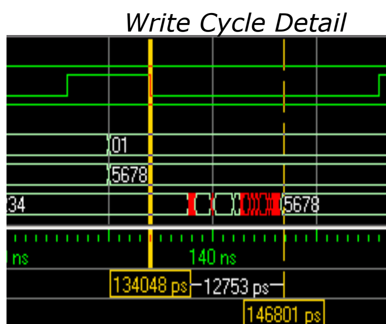
To get a more realistic view of the operation of the memory, let's look at a timing simulation.



In the read cycles, notice that there is now a delay from the time the enable goes high or the address changes until the output becomes stable. A similar observation applies to the write operations. Here's a closer look at the read.



For the second read, we observe a delay of about 15.5 ns from the time the address changes to the time that the output becomes stable. This implies that the access time for this memory must be at least 15.5 ns. Here's a closer look at the write.

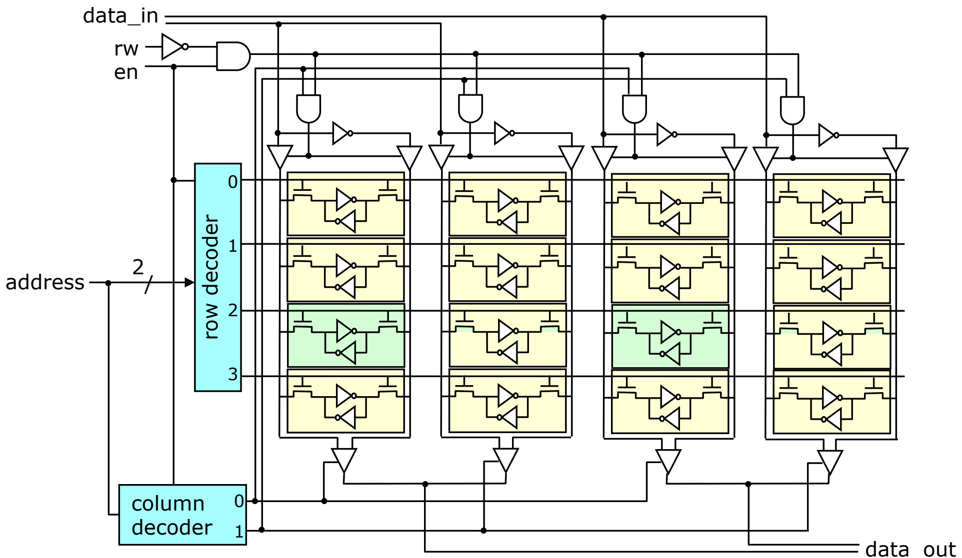


Here, we observe a delay of about 12.7 ns from the time the read/write signal goes low until the data output reflects the input. This implies that the memory cycle time must be at least this long.

Before leaving this section, we note that modern memory components are often designed to operate synchronously, rather than asynchronously. A synchronous memory uses a clock, and stores the provided address and data in an internal register at the start of a memory operation. During a memory read, the output from the memory array is stored in an output register on the next clock tick.

20.2 Alternate Memory Organizations

In the last section, we noted that the storage arrays in real memory components may have hundreds of rows and columns. If we simply scaled up the memory organization used in the last section, this would mean that our memory component would have hundreds of data inputs and outputs. This is usually not very practical, so real memory arrays are typically structured so that each row in the array contains bits belonging to multiple words. For example, a memory with a 256×256 array of storage cells might store 8 words of 32 bits apiece in each row (or 32 words of 8 bits, or 16 words of 16 bits, and so forth). The figure below illustrates how this is done. It shows a memory that stores eight words of two bits each.



Because the memory stores eight words, it requires a total of three address bits. Two of these are used by the row decoder, as before. The third bit goes to the column decoder at bottom left. The outputs of the column decoder are used to enable the sense amplifiers in alternate columns, when reading. They are also used to enable the column drivers in alternate columns when

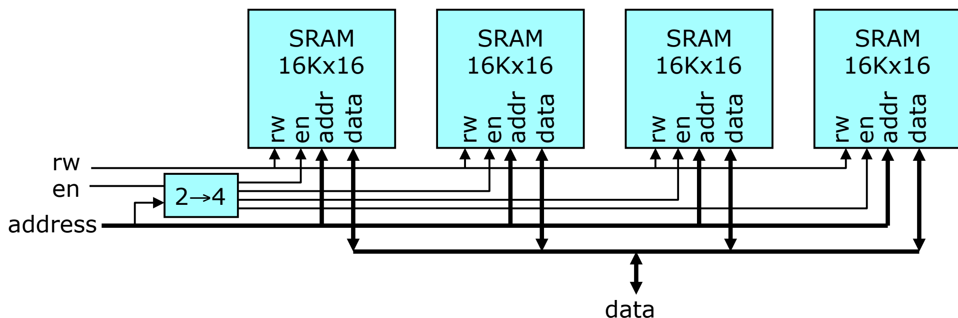
writing.

Let's consider a larger example, to get a more complete understanding of how such memory components are constructed. Let's suppose we want to implement a memory that stores 128 bytes, that is, the "words" in the memory are each eight bits long. Since the total amount of memory required is $128 \times 8 = 1024$, we can implement this using a square storage array with 32 rows and 32 columns ($32 \times 32 = 1024$). Now, because we have $128 = 2^7$ words, we need 7 address bits. Since each row in the array contains 32 bits, each will store four words. Consequently, the row decoder will use the first five address bits and the column decoder will use the remaining two.

We can generalize this discussion to handle SRAMs of arbitrary size. Let's suppose we want to implement a memory with n words of w bits apiece. For simplicity, we will assume that both n and w are powers of 2 and that the total number of bits $N = nw$ is an even power of 2, say $N = 2^{2k}$, where k is an integer. The number of rows and columns are both equal to 2^k and so k address bits are needed by the row decoder. Since the total number of address bits is $\log_2 n$, the number of bits used by the column decoder is $(\log_2 n) - k$. The number of words per row is $2^k/w$. In our previous example, $n = 128$, $w = 8$, $N = 1024$ and $k = 5$, and the row decoder used $k = 5$ address bits and the column decoder $(\log_2 128) - 5) = 2$. The number of words per row is $2^5/8 = 4$.

So far, our example memory components have used separate data inputs and outputs. For single port memory components, it is more typical to use bidirectional data lines, in order to reduce the number of IO signals required. This is straightforward to implement, requiring only that the output data paths be controlled by tristate buffers.

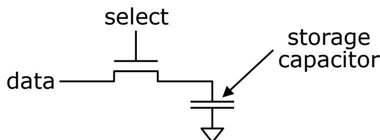
Several of the circuits we have studied use memory blocks that are built into the FPGA on our prototype board. These FPGAs are equipped with 20 configurable *block rams*. These can be configured for a variety of different word sizes, making them flexible enough to use in wide range of applications. They can also be combined to form larger memory blocks. The figure below illustrates how a larger memory can be constructed using smaller memory components as building blocks.



In this example, we are using 16K × 16 building blocks to construct a 64K × 16 memory. The decoder at the left uses two of the 16 address bits, while the remaining 14 go to the memory components. The outputs of the decoder go to the enable inputs of the memory components, so only one of them will be enabled at any one time. There are a variety of other ways we could construct this memory. For example, if we had memory blocks that were 64K × 4, we could use each memory block to provide four bits for each word of the 64K × 16 memory being constructed. This version is a little simpler, as it does not require an external decoder.

20.3 Dynamic RAMs

We noted earlier that the static RAM storage cell uses just six transistors, as opposed to 16 for a storage cell constructed from an SR-latch. *Dynamic RAMs* reduce the storage cell to just a single transistor, plus a capacitor, as shown in the diagram below.



Many copies of this storage cell can be assembled into storage arrays much like those discussed earlier. The horizontal bit lines control the selection

transistors in the storage cells and the vertical data lines connect to the lefthand terminal of the transistor. Unlike in the SRAM case, a DRAM storage array has just one vertical data line per column and one column driver. To store data into a particular storage cell, we turn on the selection transistor (by providing the appropriate address bits to the row decoder) and enable the column driver for the column containing the storage cell (by providing the appriate address bits to the column decoder). This will either charge the cell's storage capacitor (storing a "1") or discharge it (storing a "0").

Now the readout process for DRAMs is a little more complicated than for SRAMs. The first step in the process is to *precharge* the vertical data lines in the storage array to an intermediate voltage V_{ref} that is between ground and the power supply voltage. We then supply an address, which turns on the selection transistors in the desired row of the storage array. This causes one storage cell in each column to be connected to the vertical data line for that column. If the storage cell stores a "1", the positive charge on the storage capacitor will flow out onto the vertical data line, raising the voltage level on the data line. If the storage cell stores a "0", charge will flow into the storage cell's storage capacitor from the vertical data line, lowering the voltage on the data line. Now recall that the data lines in a storage array have an associated capacitance of their own. In large arrays, this capacitiance is much larger than the capacitance of the storage cells. Consequently, when we connect the storage array to the vertical data line (by turning on its selection transistor), the voltage on the data line changes by a very small amount. The sense amplifier for the column compares the voltage on the data line to V_{ref} and amplifies this voltage difference to produce the output bit for that column.

Now, an unfortunate side-effect of the read-out process is that in order to read the stored bit, we have to change the amount of charge on the storage capacitor, effectively destroying the stored value. In order to maintain the data following a read operation, it's necessary to write the data back to the storage array after reading it. This is done automatically by DRAM components, but it does have a direct impact on their performance. This property of DRAM arrays is referred to as *destructive read-out*.

Now, DRAMs have another unfortunate property, which is that the information stored in a DRAM array can "decay" over time. This happens

because real-world capacitors cannot maintain a charge for an indefinite period. Small leakage currents within a capacitor causes the charge on the capacitor to dissipate after a period of time, effectively erasing any stored information. To prevent information from being lost, DRAM arrays must be *refreshed* periodically. This means that the information stored in each row must be read out (before the charge on the storage capacitors has dissipated), and rewritten (restoring the full charge on the capacitors). Modern DRAM components do this refresh automatically during periods of inactivity.

We can now explain where the adjectives *static* and *dynamic* come from. A dynamic memory is one in which the stored information can decay over time, while a static memory is one in which the stored data remains stable so long as the circuit is powered on. This may seem a little counter-intuitive as we normally think of "dynamic" as a positive attribute. Here it is not.

Before wrapping up this chapter, let's briefly consider the relative advantages and disadvantages of SRAM and DRAM. The big advantage of DRAM is that because it has such a small storage cell, it's possible to build DRAM memory chips that have much larger storage capacities than can be done using SRAM. For this reason, DRAM is generally used for the main memory in modern computer systems. The main drawback of DRAM is that it takes considerably more time to access data stored in a DRAM than it does to access data stored in an SRAM, and this limits the system performance that can be achieved in systems that use DRAM. In a later chapter, we'll see that while modern computer systems use DRAM as their primary storage mechanism, they also use smaller SRAMs to boost overall system performance.

Chapter 21

Improving Processor Performance

In this chapter, we're going to examine the factors that limit the performance of programmable processors and look at some of the ways that processor designers have tried to overcome these limits.

21.1 A Brief Look Back at Processor Design

In the early days of computing, programmable processors were necessarily fairly simple, because the capabilities of the available circuit technology made it expensive to implement processors that required large numbers of logic gates. Processors of that period often had quite limited instructions sets that were not much more sophisticated than that of the WASHU-2.

As technology improved through the sixties and seventies, it became possible to implement processors with more elaborate instruction sets. This led to better performance, since a single instruction could often do the work of several instructions in earlier computers. Instructions for integer multiplication and division, while consuming a significant number of logic gates, replaced time-consuming arithmetic subroutines, significantly speeding up numerical calculations. The later introduction of hardware floating point

units had an even more dramatic impact on scientific computations. This period also saw the introduction of processors with growing numbers of general purpose registers, allowing programmers to eliminate many of the load and store operations requred in processors with a single accumulator, like the WASHU-2.

Throughout the 1970s and 1980s a major focus of processor designers was improving performance through the design of more effective *instruction set architectures*. In large part this process was driven by the rapid improvements in digital circuit technology. As it become practical to integrate large numbers of gates on a single chip, designers looked for ways to use the new resources to improve performance. Adding new instructions was often seen as the best way to take advantage of the growing capabilities. This quickly led to instruction set architectures with hundreds of instructions, some of which served highly specialized purposes.

In the 1980s there was a growing recognition that the expansion of instruction sets had gotten out of hand. The vast majority of software was now being written in high level languages rather than assembly language, and compiler writers found it difficult to take advantage of many of the more specialized instructions. This led to the introduction of so-called *Reduced Instruction Set Computers* (RISC). Proponents of the RISC approach argued that instructions should be included in instruction sets only if this could be justified by better performance when executing real software, or at least benchmarks that reflected the characteristics of real software. They argued that excess instructions do impose a cost, because they consume chip area that might better be used for some other purpose and they tend to slow down the achievable clock rate, which affects the performance of all instructions. During this period, processor design became much more sophisticated, with alternate architectures evaluated in a rigorous way through extensive simulations.

There was another issue that emerged as a major concern for processor designers in this time period, and that was the growing gap between the time it took to execute an instruction and the time it took to retrieve a value from memory. In the 1970s, processors might execute one instruction every microsecond and the time needed to get a value from memory was roughly

comparable. As processor clock rates increased throughout the 1980s and 1990s, the situation changed dramatically. Modern processors can execute instructions in less than one nanosecond, but the time needed to retrieve data from main memory can exceed 100 ns. While memory technology has improved dramatically since the 1970s, most of that improvement has been in the form of higher memory density. Memory speeds have also improved, but nearly so much as processor speeds.

This has created a very different kind of performance challenge for processor designers, and the principal way they have found to address it is to avoid main memory accesses whenever possible. One way to avoid memory accesses is to increase the number of processor registers, and indeed modern processors have hundreds of general purpose registers. This eliminates most memory accesses for program variables, because those variables can be kept in registers. A more sophisticated way to avoid main memory accesses is to equip processors with small on-chip memories called *caches*. Because caches are small and close to the processor, they can be much faster than the main processor memory which is implemented using separate memory components. Caches are used to hold copies of data from selected memory locations, allowing programs to avoid main memory accesses whenever a copy is available in the cache. It turns out that this technique is remarkably effective, for the vast majority of real software.

In the remainder of this chapter, we will focus on how processor performance can be improved by expanding instruction set architectures. In the next chapter, we'll discuss the role of caches in some detail, and we'll touch on some of the other ways that processor designers have sought to improve performance.

21.2 Alternate Instruction Set Architectures

A processor's *instruction set architecture* is essentially the set of processor features that directly affect the way programs are written. This includes the instructions themselves, but also the number and specific characteristics of the processor's registers.

In a processor with multiple general purpose registers, the instruction

set architecture must provide a way to specify which register (or registers) is to be used to perform a given operation. Processors are often designed to make it easy for programs to execute many instructions using only values stored in registers. This requires instructions whose operands are obtained from registers and whose result is stored in a register. Indeed, in most modern processors, the only instructions that access memory are load and store operations. All others operate only on values in registers.

Modern instruction set architectures generally have instructions to handle all the standard arithmetic and logical operations (addition, subtraction, multiplication, division, and, or, xor, shift, etc), since these operations are used frequently by most software. Frequently, each instruction will come in several variants to handle data of different sizes (8 bits, 16, 32 and 64). Many (but not all) also include hardware to perform floating point operations.

Since many programs process data using pointers, processors commonly include instructions to facilitate the efficient use of pointers. This may include incrementing or decrementing a pointer as part of an instruction that accesses data through a pointer.

Let's consider an architecture that illustrates some of these features. The WASHU-16 is a processor with 16 general-purpose registers, which take the place of the single accumulator in the WASHU-2. Like the WASHU-2, it is a 16 bit processor, but unlike the WASHU-2 it includes some instructions that require multiple words. Multi-word instructions require extra processing steps, but they do give processor designers a lot more flexibility than architectures in which all instructions must fit in a single word.

Let's start by looking at the constant load instruction.

```
0tdd        constant load. R[t]=ssdd (sign-extended)
```

The constant load instruction is identified by a 0 in the first hex digit. The second hex digit (indicated by the 't' in the summary) specifies one of the 16 general purpose registers (the *target*) of the load, while the last two hex digits (indicated by "dd") specify a constant value. This constant is sign-extended, then stored in the specified register.

Next, let's consider the direct load.

```
1txx aaaa  direct load. R[t]=M[aaaa]
```

This is our first two word instruction. It is identified by a 1 in the first hex digit. The second digit specifies the target register, as with the constant load, and the second word of the instruction specifies the memory address of the value to be loaded. Note that the last two hex digits of the first word are not used.

Next, we have an *indexed load* instruction, which is used to access data using a pointer. This takes the place of the indirect load in the WASHU-2.

```
2tsx        indexed load. R[t] = M[R[s]+x]
```

The instruction is identified by the 2 in the first hex digit. The next two digits specify the target and source registers respectively. In this case, the source register usually contains the address of the value to be loaded. To make the instruction a little more flexible, the last hex digit is added to `R[s]` to obtain the memory address. This is a handy feature for languages that use a *stack* to store local variables and arguments to sub-programs. A typical implementation of such languages uses two general-purpose registers, a *stack pointer* and a *frame pointer*. The frame pointer points to the first stack location used by the subprogram at the top of the stack. The local variables of that subprogram can then be accessed by adding small constant offsets to the frame pointer.

Because pointers are often used to access memory locations one after another in sequence, the WASHU-16 includes indexed load instructions that also increment or decrement the pointer register.

```
3tsx        indexed load, increment. R[t]=M[R[s]+x]; R[s]++;
4tsx        indexed load, decrement. R[s]--; R[t]=M[R[s]+x];
```

In the WASHU-2, we must use three instructions to advance a pointer, after accessing memory using it. Here, the increment requires no extra instructions. Note that the decrement version does the decrementing before the memory access. This is designed to facilitate use of the register as a stack pointer. Often processors have instructions with both pre-increment, post-increment and pre-decrement, post-decrement variants. Here, we are limiting ourselves to two variants, for simplicity.

The store instructions are directly comparable to the load instructions.

```
5sxx aaaa  direct store. M[aaaa]=R[s]
6tsx       indexed store. M[R[t]+x]=R[s]
7tsx       indexed store, increment. M[R[t]+x]=R[s]; R[t]++;
8tsx       indexed store, decrement. R[t]--; M[R[t]+x]=R[s];
```

Next, let's look at the arithmetic instructions.

```
90ts       copy. R[t]=R[s]
91ts       add. R[t]=R[t]+R[s]
92ts       subtract. R[t]=R[t]-R[s]
93ts       negate. R[t]=-R[s]
```

Note, that these are all register-to-register instructions, so they do not require any memory accesses. With 16 registers at our disposal, many subprograms can store all of their local variables in registers. When we compare this to the WASHU-2, it's easy to see how this would significantly cut down on the number of memory accesses required. Also observe that there are 12 unused instruction codes starting with 9, so it is straightforward to extend this set to include additional instructions.

Here are the logical instructions

```
A0ts       and. R[t]=R[t] and R[s]
A1ts       or. R[t]=R[t] or R[s]
A2ts       exclusive-or. R[t]=R[t] xor R[s]
```

Once again, there is ample room in the instruction coding for additional instructions.

Finally, let's examine the branch instructions.

```
C0xx tttt branch. PC=tttt
C1tt      relative branch. PC=PC+sstt (sign-extended add)
Dstt      relative branch on zero. if R[s]=0, PC=PC+sstt
Estt      relative branch on plus. if R[s]>0, PC=PC+sstt
Fstt      relative branch on minus. if R[s]<0, PC=PC+sstt
```

There are two unconditional branches. The first uses a second word to supply the target address, while the second adds an offset to the PC, as in the

WASHU-2. As with the WASHU-2, there are three additional branch instructions, but these all specify a specific register.

Putting it altogether, we have

```
0tdd      constant load. R[t]=ssdd (sign-extended)
1txx aaaa direct load. R[t]=M[aaaa]
2tsx      indexed load. R[t]=M[R[s]+x]
3tsx      indexed load, increment. R[t]=M[R[s]+x]; R[s]++;
4tsx      indexed load, decrement. R[s]--; R[t]=M[R[s]+x];
5sxx aaaa direct store. M[aaaa]=R[s]
6tsx      indexed store. M[R[t]+x]=R[s]
7tsx      indexed store, increment. M[R[t]+x]=R[s]; R[t]++;
8tsx      indexed store, decrement. R[t]--; M[R[t]+x]=R[s];

90ts      copy. R[t]=R[s]
91ts      add. R[t]=R[t]+R[s]
92ts      subtract. R[t]=R[t]-R[s]
93ts      negate. R[t]=-R[s]

A0ts      and. R[t]=R[t] and R[s]
A1ts      or. R[t]=R[t] or R[s]
A2ts      exclusive-or. R[t]=R[t] xor R[s]

B000      Halt.
C0xx tttt branch. PC=tttt
C1tt      relative branch. PC=PC+sstt (sign-extended add)
Dstt      relative branch on zero. if R[s]=0, PC=PC+sstt
Estt      relative branch on plus. if R[s]>0, PC=PC+sstt
Fstt      relative branch on minus. if R[s]<0, PC=PC+sstt
```

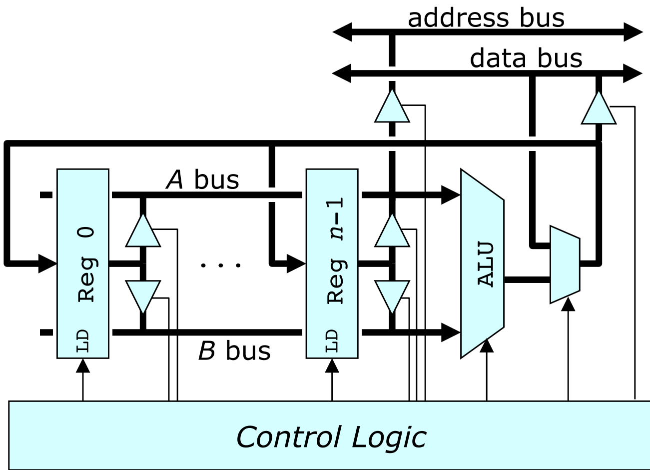
The WASHU-16 has 22 instructions. This is still a fairly small instruction set, but does give a sense of how more complete instruction sets can improve the performance of running programs. For example, consider a subprogram with three arguments and five local variables that includes a loop that is repeated many times. If we keep all the subprogram arguments and variables

in registers, we can run this entire subprogram without ever loading data from memory or writing data to memory (with the exception of initializing the registers at the start of the subprogram, and writing the final results to memory, at the end). This means that for most of the time the program is running, the only memory accesses will be those that happen during instruction fetches. This gives us a major improvement in performance.

Now, the new instruction set architecture has a smaller impact on the performance of the fetches. It does reduce the number of instruction fetches required to some extent, since a single instruction can often do the work of several WASHU-2 instructions. Still, every fetch does require a memory access. In the next chapter, we'll see how we can avoid most of these accesses using a cache.

21.3 Implementing the WASHU-16

In this section, we'll look at how we can implement the WASHU-16. We'll see that while there are some significant differences from the WASHU-2, there are also many similarities. Perhaps the biggest difference with the WASHU-16 is its use of multiple registers. These can be organized in a *register file*, along with the ALU, as shown below.



The register file includes two internal buses, labeled A and B that connect the registers to the ALU. By enabling the appropriate tristate buffers at the register ouputs, the controller can connect any pair of registers to the ALU inputs. The output of the ALU can be loaded into any register, making it straightforward for the controller to implement any of the arithmetic/logic operations. The tristate buffers connecting the A bus to the address bus, allow the contents of any register to be used as a memory address. The mux and tristate at the right end of the diagram allow any register to be loaded from memory and allow the value in any register to be stored to memory.

Now, let's consider how we can implement the WASHU-16 processor using VHDL.

```
entity cpu is port (
        clk, reset: in  std_logic;
        en, rw: out std_logic;
        aBus: out address;
        dBus: inout word);
end cpu;

architecture cpuArch of cpu is
type state_type is (
```

```
            reset_state, fetch,
            cload, dload, xload, xloadI, xloadD,
            dstore, xstore, xstoreI, xstoreD,
            copy, add, sub, neg, andd, orr, xorr,
            halt,
            branch, rbranch, rbranchZ, rbranchP, rbranchM
);
signal state: state_type;
signal tick: std_logic_vector(3 downto 0);
```

To simplify things a little bit, we are not including console signals, nor implementing the *pause* feature. As with the WASHU-2, we define a state for each instruction and use a *tick* register to keep track of the current time step with each state.

Here are the processor registers.

```
signal pc:     address; -- program counter
signal iReg:   word;    -- instruction register
signal maReg:  address; -- memory address register

signal this:    address; -- address of current instruction

-- register file
type regFile is array(0 to 15) of  word;
signal reg: regFile;
```

In addition to register file, we have introduced a new *memory address register*, which is used during direct load and store instructions to hold the address of the location being accessed.

Next we have three auxiliary registers.

```
-- index of source and target registers
signal target: std_logic_vector(3 downto 0);
signal source: std_logic_vector(3 downto 0);
-- target used for branch instructions
signal branchTarget: address;
```

```
begin
    branchTarget <= this + ((15 downto 8 => ireg(7))
                                 & ireg(7 downto 0));
```

Recall that most instructions involve a target register, a source register, or both. The *target* and *source* signals are the indices that identify those registers. The *branchTarget* register is the target used by the branch instructions. It is obtained by adding the address of the current instruction to the low-order eight bits of the instruction register (after sign-extension).

Next, let's turn to the instruction decoding.

```
    -- main process that updates all processor registers
    process (clk)

    -- instruction decoding, sets state, source, target
    procedure decode is begin
        -- default assignments to target and source
        target <= ireg(11 downto 8);
        source <= ireg(7 downto 4);
        -- Instruction decoding.
        case ireg(15 downto 12) is
        when x"0" => state <= cload;
        when x"1" => state <= dload;
        when x"2" => state <= xload;
        when x"3" => state <= xloadI;
        when x"4" => state <= xloadD;
        when x"5" => state <= dstore;
                     source <= ireg(11 downto 8);
        when x"6" => state <= xstore;
        when x"7" => state <= xstoreI;
        when x"8" => state <= xstoreD;
        when x"9" => case ireg(11 downto 8) is
                          when x"0" => state <= copy;
                          when x"1" => state <= add;
```

```
                    when x"2" => state <= sub;
                    when x"3" => state <= neg;
                    when others => state <=halt;
                    end case;
                    target <= ireg(7 downto 4);
                    source <= ireg(3 downto 0);
    when x"a" => case ireg(11 downto 8) is
                    when x"0" => state <= andd;
                    when x"1" => state <= orr;
                    when x"2" => state <= xorr;
                    when others => state<=halt;
                    end case;
                    target <= ireg(7 downto 4);
                    source <= ireg(3 downto 0);
    when x"b" => state <= halt;
    when x"c" => if ireg(11 downto 8) = x"0" then
                    state <= branch;
                 elsif ireg(11 downto 8) = x"1" then
                    state <= rbranch;
                 else
                    state <=halt;
                 end if;
    when x"d" => state <= rbranchZ;
                 source <= ireg(11 downto 8);
    when x"e" => state <= rbranchP;
                 source <= ireg(11 downto 8);
    when x"f" => state <= rbranchM;
                 source <= ireg(11 downto 8);
    when others => state <= halt;
    end case;
end procedure decode;
procedure wrapup is begin
    -- Do this at end of every instruction
    state <= fetch; tick <= x"0";
```

```
    end procedure wrapup;
```

Unlike the WASHU-2, here we are using a procedure to implemement the decoding operation, rather than a function. This is so that we can assign source and target values, in addition to the processor state. The *wrapup* procedure serves the same purpose as in the WASHU-2.

Next, let's look at the initialization of the processor and the handling of the fetch state.

```
    begin
          if rising_edge(clk) then
              if reset = '1' then
                state <= reset_state; tick <= x"0";
                pc <= (others => '0'); iReg <= (others => '0');
                maReg <= (others => '0');
                target <= (others => '0');
                source <= (others => '0');
                for i in 0 to 15 loop
                    reg(i) <= (others => '0');
                end loop;
            else
                tick <= tick + 1; -- advance time by default
                case state is
                when reset_state => wrapup;
                when fetch =>
                    case tick is
                    when x"1" => ireg <= dBus;
                    when x"2" =>
                        if ireg(15 downto 12) /= x"1" and
                           ireg(15 downto 12) /= x"5" and
                           ireg(15 downto 8) /= x"c0" then
                            decode; tick <= x"0";
                        end if;
                        this <= pc; pc <= pc + 1;
                    when x"4" =>
```

```
            maReg <= dBus;
            decode; tick <= x"0";
            pc <= pc + 1;
        when others =>
        end case;
```

Observe that for most instructions, the instruction fetch terminates at the end of tick 2. The three exceptions are the direct load, the direct store and the (non-relative) branch instruction. For these, the fetch is extended for two more ticks to allow the address to be read from the second word of the instruction.

Now, let's turn to some of the instructions.

```
when halt => tick <= x"0"; -- do nothing

-- load instructions
when cload =>
    reg(int(target)) <= (15 downto 8=>ireg(7))
                        & ireg(7 downto 0);
    wrapup;
when dload =>
    if tick = x"1" then
        reg(int(target)) <= dBus; wrapup;
    end if;
when xload =>
    if tick = x"1" then
        reg(int(target)) <= dBus; wrapup;
    end if;
when xloadI =>
    if tick = x"1" then
        reg(int(target)) <= dBus;
        reg(int(source)) <= reg(int(source))+1;
        wrapup;
    end if;
when xloadD =>
```

```
        if tick = x"1" then
            reg(int(target)) <= dBus; wrapup;
            reg(int(source)) <= reg(int(source))-1;
        end if;
```

Note that for the indexed load instructions, the target register is loaded from the memory location specified by the source register. The store instructions are similar, although somewhat simpler.

```
    -- store instructions
    when dstore | xstore => wrapup;
    when xstoreI =>
        reg(int(target)) <= reg(int(target))+1;
        wrapup;
    when xstoreD =>
        reg(int(target)) <= reg(int(target))-1;
        wrapup;
```

The register-to-register instructions can be implemented very simply, as shown below.

```
    -- register-to-register instructions
    when copy =>
        reg(int(target)) <= reg(int(source));
        wrapup;
    when add =>
        reg(int(target)) <=
            reg(int(target)) + reg(int(source));
        wrapup;
    when sub =>
        reg(int(target)) <=
            reg(int(target)) - reg(int(source));
        wrapup;
    when neg =>
        reg(int(target)) <=
```

```
                (not reg(int(source))+1;
            wrapup;
        when andd =>
            reg(int(target)) <=
                reg(int(target)) and reg(int(source));
            wrapup;
        when orr =>
            reg(int(target)) <=
                reg(int(target)) or reg(int(source));
            wrapup;
        when xorr =>
            reg(int(target)) <=
                reg(int(target)) xor reg(int(source));
            wrapup;
```

Finally, we have the branch instructions.

```
        -- branch instructions
        when branch => pc <= maReg; wrapup;
        when rbranch => pc <= branchTarget; wrapup;
        when rbranchZ =>
            if reg(int(source)) = x"0000" then
                pc <= branchTarget;
            end if;
            wrapup;
        when rbranchP =>
            if reg(int(source))(15) = '0' then
                pc <= branchTarget;
            end if;
            wrapup;
        when rbranchM =>
            if reg(int(source))(15) = '1' then
                pc <= branchTarget;
            end if;
            wrapup;
```

```
                when others => state <= halt;
                end case;
            end if;
        end if;
    end process;
```

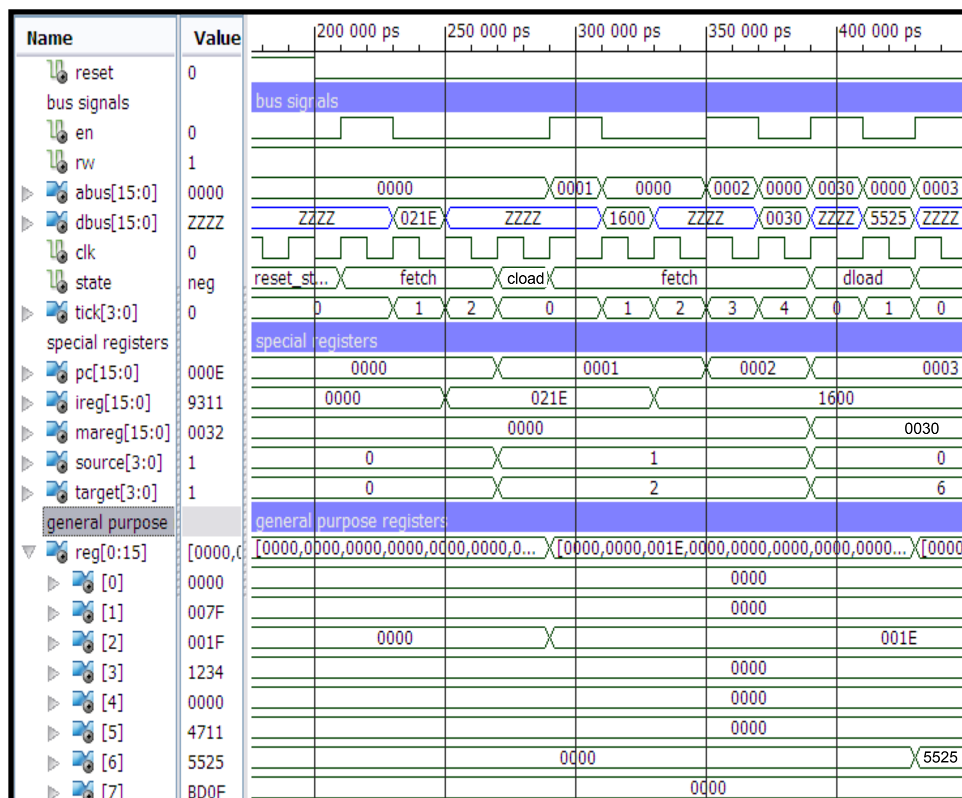
As with the WASHU-2, a separate combinatorial process controls the memory signals.

```
    -- process controlling memory signals
    process(ireg,pc,maReg,reg,state,tick) begin
        en <= '0'; rw <= '1';
        aBus <= (others => '0'); dBus <= (others => 'Z');
        case state is
        when fetch =>
            if tick = x"0" or tick = x"3" then
                en <= '1'; aBus <= pc;
            end if;
        when dload =>
            if tick = x"0" then
                en <= '1'; aBus <= maReg;
            end if;
        when xload | xloadI =>
            if tick = x"0" then
                en <= '1';
                aBus <= reg(int(source)) + ireg(3 downto 0);
            end if;
        when xloadD =>
            if tick = x"0" then
                en <= '1';
                aBus <= (reg(int(source))-1) + ireg(3 downto 0);
            end if;
        when dstore =>
            en <= '1'; rw <= '0';
            aBus <= maReg; dBus <= reg(int(source));
```

```
        when xstore | xstoreI =>
            en <= '1'; rw <= '0';
            aBus <= reg(int(target)) + ireg(3 downto 0);
            dBus <= reg(int(source));
        when xstoreD =>
            en <= '1'; rw <= '0';
            aBus <= (reg(int(target)) - 1) + ireg(3 downto 0);
            dBus <= reg(int(source));
        when others =>
        end case;
    end process;
end cpuArch;
```

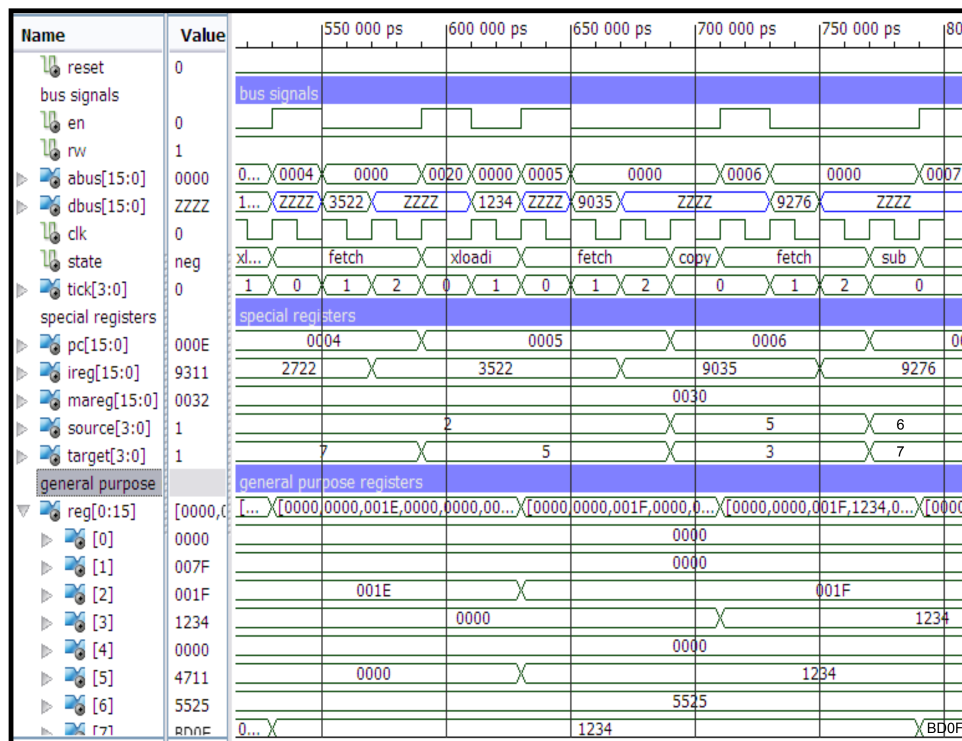
Note that the indexed load with decrement does not use the source register directly, to specify the address. Instead it first subtracts 1, since we need to perform the memory read during tick 0, when the source register has not yet been decremented. The indexed store with decrement is handled similarly.

Let's finish up this chapter by taking a look at a simualation showing the execution of some of these instructions. First, here is a short segment showing a constant load, followed by a direct load.



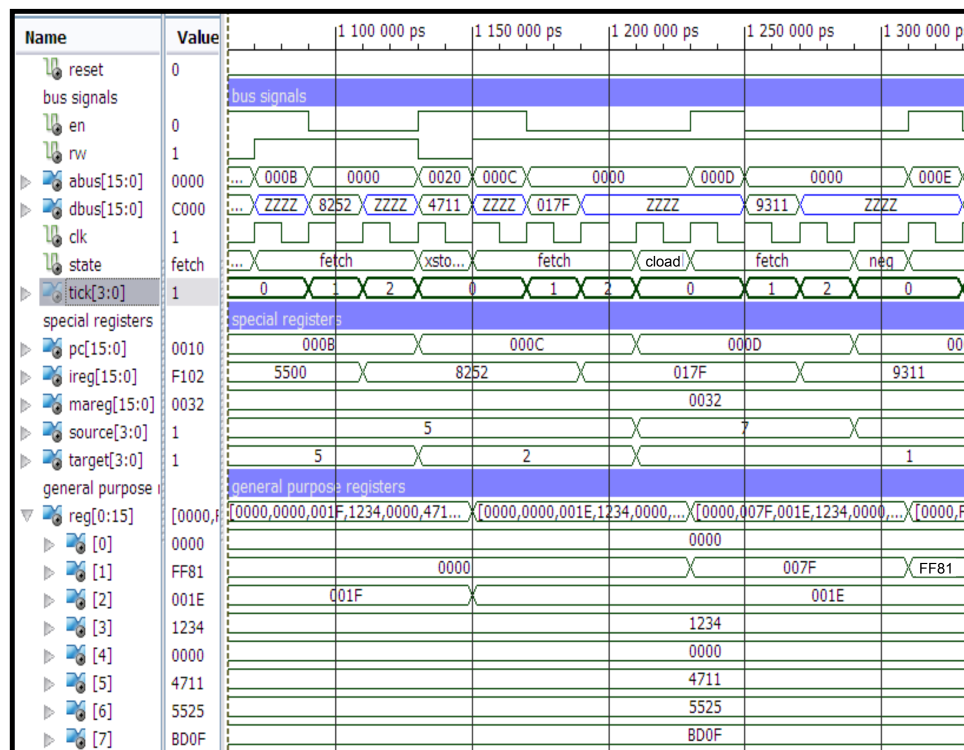
Note that the first eight of the general-purpose registers are shown at the bottom of the waveform windows, with the more specialized registers, appearing above. Observe that the first instruction (021E) changes the value of $R[2]$ (the target) to 001E. Notice that the second instruction (1600) has a five tick fetch state, during which the program counter is incremented twice. During the last two ticks of the fetch, the processor retrieves the address at location 2 and stores it in the memory address register. The value of the memory address register is then used as the address during the execution phase of the instruction, resulting in the value 5525 being loaded into $R[6]$

Next, we have an indexed load with increment, a copy and a subtract instruction.



The indexed load (3522) adds 2 to $R[2]$ and uses the sum as the memory address. The value returned from memory is loaded into $R[6]$, and $R[2]$ is then incremented. The copy instruction (9035) copies the value in $R[5]$ to register $R[3]$ while the subtract instruction (9276) subtracts the value in $R[6]$ from $R[7]$.

Finally, we have an indexed store with decrement, a constant load and a negate instruction.



The indexed store (8252) with decrement adds 2 to $R[2] - 1$ and uses this as the address for the store. It then decrements $R[2]$. The constant load (017f) loads 007F into $R[1]$ and the negate instruction (9311) negates the value in $R[1]$.

These examples illustrate the basic operation of the WASHU-16 processor. Note that while the specifics are different, the overall operation is really very similar to that of the WASHU-2. The main additional source of complexity is the 16 general purpose registers. Of course, this is also what gives the WASHU-16, its performance advantage over the WASHU-2.

Chapter 22

Improving Processor Performance Even More

In the last chapter, we concentrated on how we could improve processor performance through more efficient instruction set architectures. In this chapter, we're going to focus on how caches can be used to improve processor performance.

A cache is a small fast memory used to hold copies of memory words that have been recently used. Before a processor initiates a memory read operation, it first checks to see if the desired word is in the cache. If it is, the processor uses the cached copy, otherwise it goes ahead and does a memory read. Now caches can make a big difference in modern processors, because in a processor with a 1-2 GHz clock, it can take 100 clock cycles or more to retrieve data from main memory, but just a few cycles to retrieve data from cache. Caches tend to be very effective because programs have a tendency to re-use the same memory locations many times within a short time period. For example, consider the memory locations that hold the instructions that implement a loop in a program. The same instructions are used every time the program goes through the loop, so if these instructions are saved in a cache, the first time through the loop, they will be available in the cache for all the remaining iterations.

22.1 Cache Basics

There are several different ways to implement a cache. The most general is the *fully associative cache*, which is implemented using a special type of memory called *content-addressable memory* CAM. A CAM is essentially a hardware implementation of the software data structure known as a *map*. The words in a CAM are divided into two parts, a *key* and a *value*. To lookup an entry in a CAM, we provide a *query*, which is compared against all the keys stored in the CAM. The value associated with the first entry for which the key field is equal to the query is returned as the output of the lookup operation. When a CAM is used to implement a cache, the key field contains a main memory address and the value is the contents of the memory word at that address. So, we can think of the cache as storing a set of (address, value) pairs, and defining a mapping from an address to the value stored at that address.

The figure below shows an example of a cache implemented using a CAM.

query key=3456
result=341c

v	*key*	*value*
0	0123	9876
1	1234	3412
1	1235	6431
1	3456	341c
0	213b	bcde
1	abcd	9090

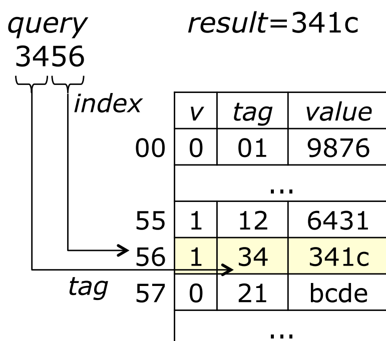
valid bit

Note that each entry has an *address field*, a *value field* and also a *valid bit*. Entries for which the valid bit is zero are ignored during lookup operations. The cache entries tell us what values are stored at certain memory locations; for the example, we can infer that M[1234]=3412 and M[abcd]=9090. When we perform a lookup, the query is compared against the stored address fields of all entries for which the valid bit is set. In the example, the query of 3456

matches the entry (3456, 341c), so the value 341c is returned as the result of the lookup.

Now, suppose the processor executes a load instruction for memory location 5412. Since there is no cache entry with an address field equal to 5412, the processor will have to retrieve the word from main memory. When it does so, it will make a new entry in the cache, using one of the unused words. What happens if we retrieve a new word from memory, when the cache is already full (all valid bits are set)? In this case, we generally want to *replace* one of the existing cache entries with the new one. The ideal choice is the *least-recently used* entry, since experience with real programs has shown that this one is less likely to be needed in the future, than those entries that have been accessed recently.

Now, one problem with fully associative caches, is that CAM's are relatively expensive (compared to SRAMs) because they require comparison logic in every word, to compare the query to the stored addresses. This has led to alternative cache architectures that use SRAM in place of the CAM. The simplest of these alternatives is the *direct-mapped cache*, illustrated in the figure below.



First note that the numbers at the last edge of the memory represent the memory addresses for the SRAM used to implement the cache. Also, observe that the SRAM has a *tag* field in place of the address field used in the fully associative cache. The entry for a given main memory address is stored in the SRAM at the location specified by its low-order bits. The high order bits

of the main memory address are stored in the tag field of the entry. So in the example shown above, we can infer that M[1255]=6431 and M[3456]=341c.

When we do a lookup in a direct-mapped cache, we use the low-order bits of the query key (the *index*) as the SRAM address. The high order bits of the query are then compared to the stored tag bits in the entry selected by the index. If they match and the valid bit is set, then the value field represents the contents of the memory location that we're interested in. Otherwise, we'll need to do a main memory read, updating the cache when the contents of the memory word is returned. Note that unlike in the fully-associative cache, here we have no choice about where to store the new entry. It must be placed in the location specified by its low-order bits. Because the entry for a given main memory address can only appear in one location, we do not need to search the entire memory to find it; we just need to check one location.

How do evaluate the performance of alternative cache designs? The crucial criterion is how often we find what we're looking for in the cache. This is generally referred to as the *hit rate*. Note that in order for any cache to have a high hit rate, it's necessary that the same instructions are executed repeatedly. Thus programs with loops generally have better cache performance than programs that have long sections of "straight-line code".

Given a direct-mapped cache and a fully-associative cache with the same number of entries, the fully-associative cache will usually a higher hit rate, since it has no restriction on where it must store a particular (address, value) pair. It is limited only by the total number of entries. On the other hand, the fully-associative cache consumes considerably more area on an integrated circuit chip than a direct-mapped cache with the same number of entries. If we reduce the number of entries in the fully-associative cache to the point where the area consumed is equal to that of the direct-mapped cache, we often find that the direct-mapped cache performs better.

Direct-mapped caches are most commonly used as instruction caches; that is, they are used to hold instructions to be executed, rather than data. Let's consider an example, to get a better understanding of how it is that this very simplistic caching strategy can deliver good performance. Suppose we have a subprogram consisting of 200 instructions. Note that since the in-

structions in the subprogram are stored in consecutive memory locations, the low order bits of the instruction addresses will be different. More precisely, for any two instructions in the subprogram the low-order eight bits of the addresses of the two instructions will be different. Consequently, if we have a direct-mapped cache that can hold 256 instructions, all the subprogram's instructions can be present in the cache at the same time. So, when the subprogram begins execution, no instruction will have to be loaded into the cache more than once. Now, if the subprogram contains a loop that is executed many times, we get to re-use those loop instructions from cache many times, avoiding the memory reads that would normally occur during an instruction fetch. Note that a fully-associative cache with the same number of entries will perform no better in this situation, but will consume considerably more chip area.

The behavior of this subprogram is an example of a more general property called *locality of reference*. Essentially this just means that instruction addresses that are encountered during the execution of real programs are highly correlated. More precisely, if two instructions are executed close to each other in time, it's likely that those two instructions are close to each other in memory. This in turn implies that their addresses differ in the low order bits, and that property is what allows a direct-mapped cache to be effective at reducing the number of main memory accesses.

22.2 A Cache for the WASHU-2

Let's take a look at how we can apply these ideas to the WASHU-2. In particular, let's add a direct-mapped cache to the WASHU-2 processor and see how it improves the performance of running programs. To keep things simple, we'll use a fairly small cache with just 64 entries (by comparison, the instruction caches in a typical laptop can hold thousands of instructions). Here are the declarations needed to implement the cache.

```
type icacheEntry is record
    valid: std_logic;
    tag: std_logic_vector(9 downto 0);
```

```
    value: word;
end record;
type icacheType is array(0 to 63) of icacheEntry;
signal icache: icacheType;
signal icacheIndex: std_logic_vector(5 downto 0);
signal icacheOut: icacheEntry;
begin
    ...
    icacheIndex <= pc(5 downto 0);
    icacheOut <= icache(int(icacheIndex));
```

Note that each cache entry has a valid bit, a ten bit tag and a 16 bit memory word. Since the cache has 64 entries, the index is the low-order six bits of the main memory address, and the tag is the remaining ten bits. Since we are implementing an instruction cache, the cache is only used during the instruction fetch, and the index bits are always the low-order six bits of the PC. The *icacheOut* signal is the value of the cache entry with the specified index.

Here is the code that implements the fetch.

```
elsif state = fetch then
    if tick = x"0" then
        -- check cache for data
        if icacheOut.valid = '1' and
           icacheOut.tag = pc(15 downto 6)
        then
            state <= decode(icacheOut.value);
            iReg <= icacheOut.value;
            this <= pc; pc <= pc+1;
            tick <= x"0";
        end if;
    elsif tick = x"1" then
        iReg <= dBus;
        icache(int(icacheIndex)) <=
                    ('1,pc(15 downto 6),dBus);
```

```
                elsif tick = x"2" then
                    state <= decode(ireg);
                    dpc <= pc; pc <= pc + 1; tick <= x"0";
                end if;
```

Observe that the cached entry is checked during tick 0 of the fetch, and if the entry is valid and has a matching tag, the processor proceeds directly to the execution phase, skipping the memory read. If the cached entry is either not valid, or has a tag that does not match the PC, the memory read takes place, and the value returned from memory is stored in the IREG and used to update the cache entry, at the end of tick 1.

Here is the relevant section of the process that controls the memory.

```
process (ireg,pc,iar,acc,pcTop,state,tick) begin
-- Memory control section (combinational)
  ...
    case state is
    when fetch =>
        if tick = x"0" and
           (icacheOut.valid = '0' or
            icacheOut.tag /= pc(15 downto 6)) then
             en <= '1'; aBus <= pc;
        end if;
```

This code allows the memory read to proceed only if the the (address, value) pair we're looking for is not in the cache.

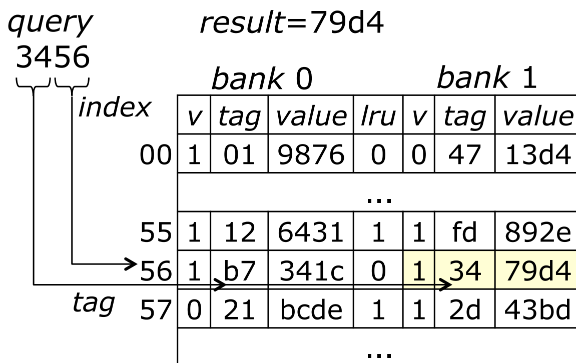
So, how big a difference will this make in the execution of a real program? Consider the multiply subprogram from Chapter 17. This has a loop that contains 15 instructions and is executed 16 times. So, there are 240 instruction fetches performed during that loop. With no cache, these fetches take three clock ticks each, for a total of 720 clock ticks. With the cache, the number of clock ticks required by all the fetches is $3 \times 15 + 225 = 270$, a savings of 450. This is a significant improvement, but it's actually a much smaller improvement than we would see on a modern processor that uses an external DRAM for its main memory. In that case, the time to fetch an

instruction from main memory might be 100 ns, while the time to retrieve the instruction from cache could be just a couple ns. So a cache hit could give us a 50× speedup, rather than the 3× we get with the WASHU-2.

22.3 Beyond Direct-Mapped Caches

Direct-mapped caches are very effective for reducing the number of memory accesses when used as instruction caches. They are less effective when used to store data values used by a program, because data access patterns are not as regular as instruction fetches. So, if direct-mapped caches do not work well for data, do we have to fall back on fully associative caches? It turns out that there is an intermediate strategy that allows us to get most of the benefits of a fully associative cache, but at a cost which is not much higher than that of a direct-mapped cache.

A *k-way set associative cache* is implemented using k "banks" of SRAM, where k is a small integer, typically in the range of $4 - 8$. Each bank can be viewed as a direct-mapped cache, but they are operated in parallel, and each main memory word can be stored in any of the k banks. An example of a 2-way cache is shown below.



As with the direct-mapped cache, we place entries in the set-associative cache based on the low-order bits of their main-memory addresses. The index determines the row in the cache where the entry will be stored. However,

because we can place a given entry in either bank, we have somewhat more flexibility than we do in a direct-mapped cache. As with the direct-mapped cache, we can use the index and tag bits of the entries to infer the contents of main memory locations. So, in the example, we can infer that M[1255]=6431 and M[2d57]=43bd.

When we do a lookup in this cache, we read the entries in both banks and compare the stored tag values against the tag of the query. If we find a matching tag, then the value field for that entry is returned. If neither matches, we must retrieve the required value from memory. Each row in the cache also has an *lru* bit that specifies which bank has the least-recently-used entry. In a row with two valid entries, this tells us which entry should be replaced when a new entry is being added to the cache. (For $k > 2$, we'll need to maintain more than one bit of information to keep track of the least-recently-used entry in the row.)

To compare the performance of a set-associative cache to a direct-mapped cache in a fair way, we need to assume the same number of entries in both. So, a direct-mapped cache will have k times as many rows as the comparable k-way cache. The advantage of the k-way cache comes from the fact that each main memory word can be stored in any of the k banks. This makes it less likely that a cache entry will be evicted from the cache by the entry for some other memory location that happens to have the same index. Note that if we let k get large, the set-associative cache starts to resemble the fully-associative cache. Indeed if k is equal to the number of cache entries, then the two are equivalent.

Before wrapping up our discussion of caches, we should note that caches used in computer systems usually do not store individual words, but larger blocks of data, referred to as *cache lines*. Typical cache line sizes range from 32 to 128 bytes. The use of larger cache lines requires an adjustment in how we define the index and tag. Consider an example of a computer system that uses byte addresses and has a cache with a line size of 64 bytes. Observe that for a give cache line, the bytes in the cache line will have addresses in which the low order six bits range from 0 to 63. So effectively, the low-order six bits of the main memory address identify a particular byte within a cache line, while the higher-order bits of the address can be viewed as the address of the

cache line itself. The index and tag are defined relative to this "cache-line address." That is, the low-order six bits of the main memory address are not included in the index.

The reason that computer systems use larger cache lines rather than individual words is that it leads to better performance. There are two reasons for this. First, it turns out that we can retrieve an entire cache line from memory in about the same amount of time as we can retrieve a single word, so there is little extra cost involved. Second, for typical memory access patterns, the words in a cache line are often needed by the processor at around the same time. The most striking example of this is when a program is executing straight-line code. When we fetch the first instruction in a cache line, we must wait for the cache line to be retrieved from memory, but for the remaining instructions, there is no memory access required. Those instructions are already in the cache.

There are a couple other variations on how caches can be designed that are worth mentioning. The first concerns the *replacement policy*, which determines which entry in a k-way cache is replaced when a new entry must be added to a given row. The most natural policy is the least-recently-used policy, which we have already mentioned. Unfortunately, a straightforward implementation requires $\log_2 k$ bits per entry, and can slow down the operation of the cache. For this reason, caches are often designed using a *pseudo-LRU* policy that requires a single bit per entry and performs nearly as well as a strict LRU policy.

The second variation concerns when data is written to main-memory. In a *write-through cache*, whenever a store instruction is executed, the stored value is written to both the cache and main memory. In a *write-back cache*, the stored value is written only to the cache when the store instruction is executed. If at some later point in time, that value must be evicted from the cache, it is then written to main memory. Write-back caches can significantly reduce the amount of write traffic to main memory, especially in situations where stored values are written multiple times in succession.

22.4 Other Ways to Boost Performance

Processor designers have devised a number of other ways to improve the performance or programmable processors. One of them is *pipelining*. This is essentially a form of parallel computing, in which the processor works on multiple instructions at the same time. The operation of a pipeline is similar to a manufacturing assembly line, in which a product moves through a sequence of station, with each station responsible for a particular small part of the overall assembly process. In an instruction pipeline, we have a series of *pipeline stages* that each perform a small part of the processing of the instructions that flow through them. When a pipeline is operating at peak efficiency, it will complete the execution of some instruction on every clock tick, even though it may take ten or more clock ticks for a single instruction to work its way through all the pipeline stages.

While the basic concept of an instruction pipeline is straight-forward, the actual implementation can be very challenging, since often a given instruction cannot be carried out until the result from some previous instruction is known. A variety of techniques are used keep the pipeline operating efficiently, even when instructions depend on each other in this way.

Conditional branches in programs pose a significant challenge to the design of pipelines. When we come to a conditional branch in a program, there are two possibilities for the instruction that is to executed next. To keep the pipeline busy, we want to start work on the next instruction immediately, but we cannot know for sure which instruction to start working on until after the conditonal branch instruction completes. Processors typically resolve this conflict by trying to predict which branch direction is most likely. If the prediction is correct, the processor can just keep on feeding instructions into the pipeline. If it is not, the processor must backup to the branch point, discarding the instructions that were started since the branch point.

It turns out that the vast majority of branch instructions in real programs are highly predictable. For example, the branch instruction at the top of a loop is resolved the same way on all but the last iteration of the loop. Similarly, branches associated with tests for error conditions are almost always resolved the same way. To exploit this predictability, processors incorporate

branch prediction mechanisms. One simple branch predictor uses a single bit for each branch instruction. Every time a branch instruction is executed, its prediction bit is set based on which way the branch went. When we need to predict the direction of a given instruction, we base our prediction based on which way the branch went the last time. For branches that almost always go the same way, this simple mechanism can be very effective. More elaborate techniques can be used in situations where the branch direction is not so consistent.

We'll close this chapter with a brief discussion of *multi-core processors*. A multi-core processor is a single integrated circuit that contains several distinct processor cores, each capable of executing a separate program. Modern server chips now typically have eight or more cores, laptop chips frequently have four and even cell phones are now being equipped with dual core processors. The different cores on a single chip generally share main memory. This allows programs running on different cores to communicate with each other through values saved in the shared memory.

Each core in a multi-core chip maintains its own L1 cache, while L2 caches may be either shared or kept separate. When different cores maintain their own caches, values stored in main memory can also be stored in multiple caches. This creates the potential for the various copies to become inconsistent. Multi-core processors implement explicit *cache-consistency* mechanisms to ensure that copies are maintained in a way that at least appears consistent to software.

Microprocessor manufacturers began producing multi-core processor chips in the late 1990s when they could no longer boost single-core performance by increasing the clock frequency. While multi-core processors have allowed manufacturers to continue to improve the aggregate performance of their products, they do create challenges for software designers. In order to use the new chips to boost the performance of single applications, software designers must incorporate parallelism into their programs. Correctly coordinating the activity of multiple threads of execution can be very challenging, and getting the best possible performance from such programs can be even more difficult.

Chapter 23

Making Circuits Faster

When designing circuits, we often find ourselves facing two competing sets of demands. On the one hand, we would like to make our circuits as small as possible, in order to fit within the available resources of an available FPGA, for example. On the other hand, we would also like our circuits to be as fast as possible, in order to achieve a target clock frequency needed to meet the system's performance objective. Often, these two things go together (small circuits are often fast circuits), but in some cases we have to trade-off circuit cost against performance.

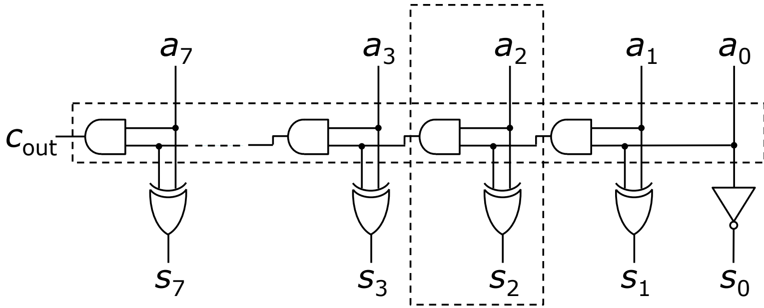
In this chapter, we are going to look at some ways we can improve the performance of circuits. We will focus on arithmetic circuits, both because they are often critical to system performance, and because there are some well-developed techniques that have been developed for arithmetic circuits but that can also be applied in other contexts.

23.1 Faster Increment Circuits

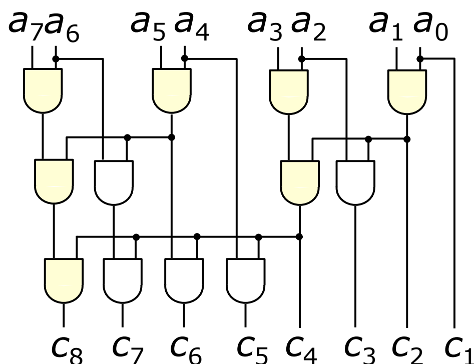
We'll start with what the simplest of all arithmetic circuits, the incrementer. Consider what happens when we add 1 to a binary number.

$$10101111 + 1 = 10110000$$

Notice how the rightmost bits of the the input value (up through the first zero) are flipped in the result. The circuit shown below is based on this observation.

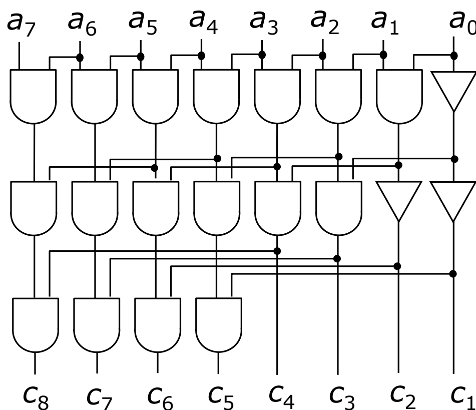


In each bit position, we flip the "current" input bit if all input bits to the right of the current position are equal to 1. This is a simple and elegant circuit, but it does have one drawback. The number of gates on the circuit path from the rightmost input bit to the leftmost output bit grows directly with the number of bits. This can limit the performance when operating on large data values. Now there's an obvious solution to this problem; instead of using a chain of AND gates, why not use a tree-structured circuit in which the circuit depth grows as the logarithm of the number of inputs, rather than linearly? Here's a circuit that implements the carry logic, based on this approach.



Here c_i represents the carry into bit position i. The shaded gates highlight a binary tree that produces the c_8. The carrys for the other bit positions are "piggy-backed" onto this tree, adding as few additional gates as needed to generate the remaining carry signals. A 64 bit version of this circuit has a maximum path length of 6, while the carry logic for the original ripple-carry incrementer has a maximum path length of 63. Unfortunately, this version is still not ideal, because it includes some gates with a large fanouts. Since the maximum fanout grows linearly with the number of bits, the circuit delay will also grow linearly.

Here's another circuit that is based on the same basic idea, but does so in a way that keeps the fanout small.



Note that in this circuit, no gate has a fanout larger than two, and the maximum path length is three. For an n bit version of the circuit, the maximum path length is $\log_2 n$, while the fanout remains 2. To understand fully how the circuit works, define $A(i,j) = a_i \cdot a_{i-1} \cdots a_j$ and note that for $i \geq k > j$,

$$A(i,j) = A(i,k)A(k-1,j)$$

In the diagram, the first row of AND gates implements the function $A(i,i-1)$ for all values of $i > 0$. The second row implements the function $A(i,i-3)$ using the equation

$$A(i,i-3) = A(i,i-1) \cdot A(i-2,i-3)$$

The third row implemements $A(i,i-7)$ using

$$A(i,i-7) = A(i,i-3) \cdot A(i-4,i-7)$$

In a larger circuit, the k-th row implements $A(i,i-2^k-1)$ using

$$A(i,i-(2^k-1)) = A(i,i-(2^{k-1}-1)) \cdot A(i-2^{k-1},i-(2^k-1))$$

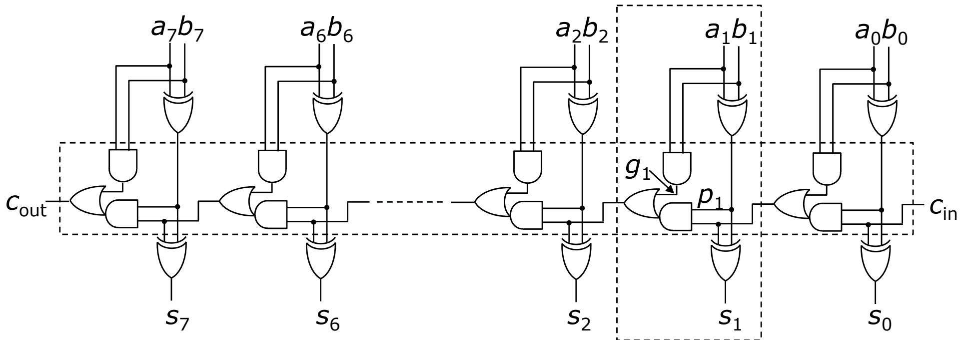
The method used for the carry-logic is referred to as *carry-lookahead*. In the next section, we'll see how it can be applied to general adders.

23.2 Faster Adder Circuits

In this section, we're going to look at how to build fast addition circuits, but let's start by reviewing the binary version of the standard longg addition algorithm that we learned in grade school. An example of binary addition is shown below.

111110100	carry-in
010011010	addend
001101011	augend
100000101	sum

The addition proceeds from right to left. Let a_i and b_i be the two input bits for position i and let c_i be the the carry into position i. The sum bit $s_i = a_i \oplus b_i \oplus c_i$ where $\oplus$ denotes the exlusive-or operation. The carry out of position i is given by, $c_{i+1} = a_i b_i + a_i c_i + b_i c_i$. This is equal to $a_i b_i + (a_i \oplus b_i) c_i$. These observations lead to the circuit shown below.



The long horizontal box highlights the carry logic. Notice that the maximum path length through the carry logic for an n bit adder is $2n$. We'd like to speed this up using the carry-lookahead technique that we used for the increment circuit, but here things are a bit more complicated, so we need to proceed in a systematic way.

There are two basic ideas that play a central role in the lookahead adder. The first is *carry generation*. We say that bit position i *generates* a carry if $a_i = b_i = 1$. If this condition is true, we know there will be a carry out of

position i, even if there is no carry into position i. The second idea is *carry propagation*. We say that bit position i *propagates* a carry if $a_i \neq b_i$, since when this is true, $c_{i+1} = c_i$. The circuit diagram above identifies signals $g_1 = a_1 b_1$ and $p_1 = a_1 \oplus b_1$. In general, $g_i = a_i b_i$ and $p_i = a_i \oplus b_i$.

We can generalize the idea of carry propagation to groups of bits. Define the function $P(i, j) = 1$ if and only if $p_i = p_{i-1} \cdots = p_j = 1$. Thus, whenever $P(i, j) = 1$, $c_{i+1} = c_j$. This implies that for $i \geq k > j$

$$P(i, j) = P(i, k)P(k-1, j)$$

We can use this to generate a family of propagate signals just as we did for the carry logic in the increment circuit.

We can also generalize the idea of carry generation to groups of bits. Define $G(i, j) = 1$ if bit positions i downto j *generate* a carry. That is, $c_{i+1} = 1$ even if $c_j = 0$. Note that $c_{i+1} = G(i, 0)$, so we can obtain our required carry signals using the appropriate generate signals. We'll do this in a recursive fashion. First, observe that

$$G(i, i-1) = g_i + g_{i-1} p_i$$

That is, the two bit group $(i, i-1)$ generates a carry if position i does, or position $i-1$ generates a carry and position i propagates the carry. Similarly,

$$G(i, i-3) = G(i, i-1) + G(i-2, i-3)P(i, i-1)$$

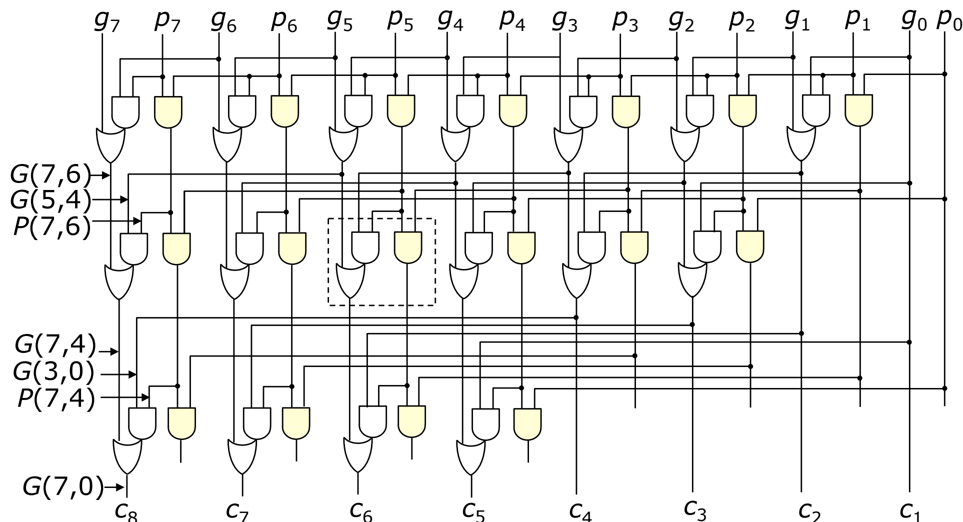
and

$$G(i, i-7) = G(i, i-3) + G(i-4, i-7)P(i, i-3)$$

More generally, for $i \geq k > j$

$$G(i, j) = G(i, k) + G(k-1, j)P(i, i-k)$$

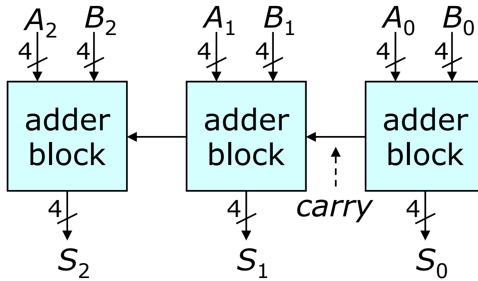
These equations lead directly to the circuit shown below.



The shaded circuits implement the propagate signals, while the remainder implement the generate signals. Here, the maximum path length through the carry logic is $2\log_2 n$, as opposed to $2n$ for the ripple-carry adder. So for $n = 64$, we can expect the lookahead adder to be ten times faster than the ripple-carry adder.

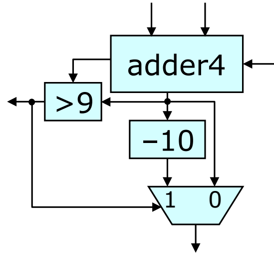
23.3 Other Linear Circuits

The carry lookahead idea can also be applied to circuits that use other other number bases. The figure below shows a BCD adder.



Here, each adder block adds a pair of BCD digits and produces a BCD digit as the result. The carry that goes between adder blocks represents the carry in the base-10 addition algorithm. The performance characteristics of this adder are similar to those of the binary adder, and we can apply the lookahead technique in exactly the same was as we have done for the binary adder.

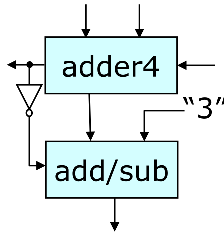
The adder block itself can be implemented as shown below.



When the four bit binary adder produces a result that is greater than nine, a carry-out is generated, and the sum is corrected by subtracting 10. Note that this requires a comparison circuit to the carry chain, increasing the carry delay. Also note that the comparison circuit must use the four sum bits plus the carry-out, since in some cases the incoming digits can add to a value that is too large to fit in four bits.

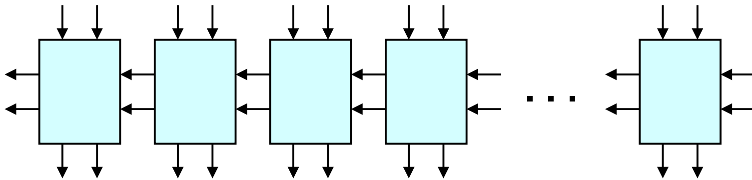
There is alternative version of the BCD adder based on a different assignment of bit patterns to the decimal digits. The *excess-three* representation adds three to the usual binary equivalents. So for example, the decimal digit

2 is represented by the bit pattern 0101, and the digit 8 is represented by 1011. It turns out that this approach produces a more symmetric encoding of the decimal digits, which leads to simpler and faster circuits. Here is an excess-three adder block that demonstrates this.



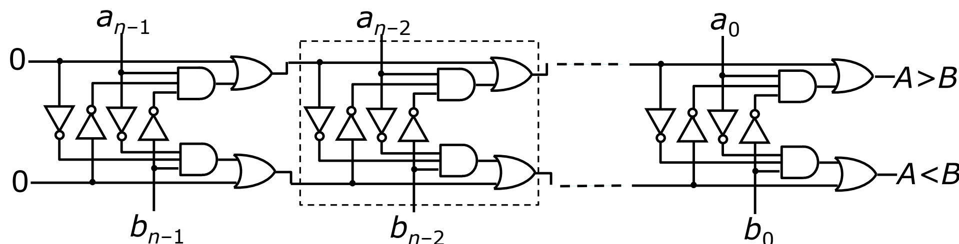
In the excess three circuit, a carry is generated when the binary adder overflows, producing a carry out of the high order bit. (For example, if we add the decimal digits 6+4, the binary adder is actually adding 1001+ 0111 giving 0000 with a carry out of the high-order bit.) When no carry is generated, we correct the result by subtracting 3. This converts the excess-6 sum produced the adder to an excess-3 value. When a carry is generated, we correct the sum by adding 3. This is equivalent to sutracting 10 and converting from excess-6 to excess-3. This circuit is a little simpler than the conventional *bcd* adder block, but more important, it has a much smaller carry delay.

The ripple-carry increment, binary adder and BCD adder circuits are all examples of a general linear circuit structure, in which a basic building block is organized in an linear array with connecting sign providing "intermediate results" of one form or another. A general pattern for such circuits is illustrated below.



Some other circuits that have the same property are subtractors, twos-

complement circuits, max/min circuits and comparators. The figure below shows a standard inequality comparator. In this case the signals flow from high-order bits to low-order bits, but the basic structure is the same. Also observe that because of the linear structure, we can expect performance to degrade as number of bits gets large. However, like with the increment and adder circuits, we can apply the lookahead idea to obtain much faster versions.



Linear circuit structures are particularly common among basic arithmetic circuits, but can also appear in more complicated circuits. For example, consider a "tally" circuit that takes an n bit input vector and for each bit position produces a tally $T(i)$ equal to the number of ones in positions $i, \ldots, 0$. This can be implemented as a linear array of adders. Moreover, we can speed it up using a lookahead circuit in which the basic elements are also adders.

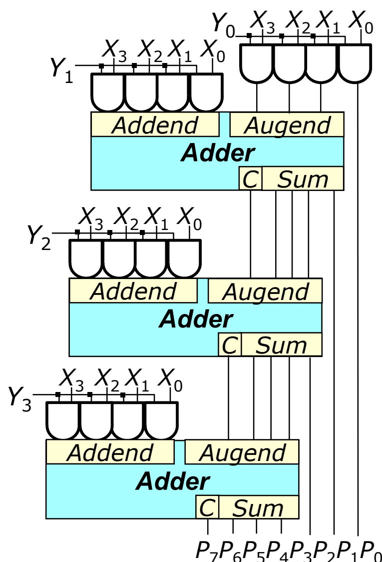
23.4 Multiplication Circuits

We'll finish this chapter by consider circuits for integer multiplication. First, let's review the binary version of the long multiplication algorithm that we

all learned in grade school

1010	multiplicand
1101	multiplier
1010	partial products
0000	
1010	
1010	
10000010	product

As in the decimal multiplication algorithm, we multiply the bits of the multiplier with the multiplicand to produce the partial products. Since each bit is either 0 or 1, the partial products are either 0000 or the multiplicand (in this case 1010). Adding the partial products together produces the overall sum. Here is a circuit that implements this algorithm.

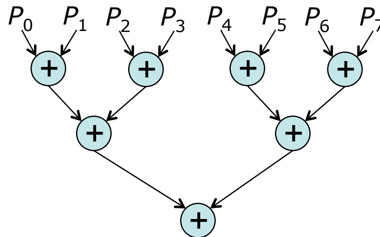


Here, the adder outputs labeled C are the carry out of the high-order bits.

What can we say about the performance of this circuit? Well, at first glance it would appear that the maximum delay through the circuit would

be about $n-1$ times larger than that for a binary adder. So, if ripple-carry adders were used, the maximum path length would be contain about $2n^2$ gates. It turns out that this is not the case. If you look at the diagram above, and think about the structure of the ripple-carry adder, you'll realize that the worst-case path starts at the top right of the circuit and ends at the bottom left. The circuit path always proceed from top-to-bottom and from right-to-left. There are actually many worst-case paths, but they all share this basic property. Consequently, the worst-case path passes through $2(n-1)$ exclusive-or gates, $2(n-1)$ AND gates and $2(n-1)$ OR gates. If we assume that the delay through an exclusive-or gates is about twice as large as the delay through an AND gate or OR gate, the overall delay is roughly four times larger than the delay through a single adder, not $n-1$ times larger.

What happens if we substitute lookahead adders for the ripple-carry adders? In this case, it turns out that the adder delays do all accumulate, giving us a worst-case path length containing $2(n-1)$ exclusive-or gates, $(n-1)\log_2 n$ AND gates and $(n-1)\log_2 n$ OR gates. Consequently, this circuit is actually *slower* than the multiplier that uses ripple-carry adders. However, we're not done yet. We can get better performance if we replace the linear array of adders in the original circuit with an *adder tree*, as shown below.



Here, the partial products are fed into the top row of adders, and the overall sum of the partial products is produced by the adder in the bottom row. When using lookahead adders, this circuit has a worst-case path that contains $2\log_2 n$ exclusive-or gates, $(\log_2 n)^2$ AND gates and $(\log_2 n)^2$ OR gates. For $n=64$, this is 12 exclusive-or gates, 36 AND gates and 36 OR gates. This compares to 126 of each type, in the case of the original multiplier.

What if we use ripple-carry adders in an adder-tree? In this case we get $2 \log_2 n$ exclusive-or gates, $2(n-1) \log_2 n$ AND gates and $2(n-1) \log_2 n$ OR gates in the worst-case path. That turns out to be worse than the original version.

Chapter 24

Producing Better Circuits Using VHDL

In previous chapters, we've discussed how we can reduce the number of gates needed to implement certain logic functions. However, there is a snag when trying to apply these lessons in the context of modern CAD tools, where we do not have direct control over the circuits that a synthesizer produces. While it is possible to write VHDL code that specifies exactly what gates to use and how to connect them, writing VHDL in this way largely throws away the advantages of using a higher level language in thie first place.

In this chapter, we'll discuss how the way we write our VHDL code affects the resources (flip flops and LUTs, when using FPGAs) used by the circuits produced by circuit synthesizers. By getting a better understanding of the capabilities and limitations of synthesizers, we can learn to write code in a way that makes it more likely that the synthesizer will produce a high quality result.

At the outset, it's worth mentioning that resource usage is not the only criteria that is important when designing circuits using VHDL. Often we may be more concerned with completing a design quickly, so as to deliver a product in a timely way. In this context, we may not have the time to explore a variety of different architectures. On the other hand, there are also situations when it may be crucial to reduce the resources used by a given

circuit, in order to ensure that it fits within the constraints imposed by the FPGA or ASIC being used to implement the overall system. In any case, it's always worthwhile to maintain some basic awareness of the resources used by our circuits, so that we can make an informed decision about whether we need to consider more efficient alternatives, or not.

24.1 Some Motivating Examples

The way in which we write our VHDL circuit specifications can have a big effect on the quality of the circuit produced by a synthesizer. Let's look at a very simple example that illustrates this. The following code fragment implements the core of a very simple arithmetic and logic unit (ALU), with 16 bit data inputs A and B, a 16 bit data output X and a two bit control input C.

```
with C select
X <= B             when 00,
     (not B) + 1   when 01,
     A+B           when 10,
     A-B;
```

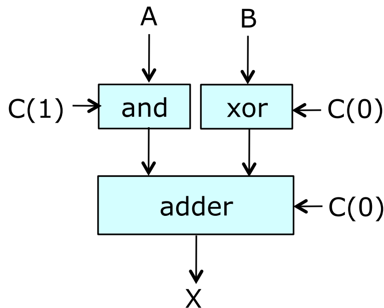
The expression `(not B) + 1` produces the 2s-complement of B. The "default" implementation of this circuit uses a 4:1 multiplexor, an adder, a subtracter and a 2s-complement circuit. If we were to construct this circuit using only four input LUTs, we would get 3 LUTs per bit for the multiplexor, 2 LUTs per bit for the adder, another 2 LUTs for the subtracter and another 2 LUTs per bit for the 2s-complement circuit. This gives us a total of 10 LUTs per bit, or 160 altogether. In reality, the synthesizer is able to do much better than this for a couple reasons. First, it is able to recognize that the adder, subtracter and 2s-complement circuit are never used at the same time, so that it can combine these functions. Second, the FPGA we are using has other "hidden" resources in addition to the LUTs and flip flops. These are not general purpose resources, but are designed to assist in the synthesis of certain commonly occuring logic functions (like addition and subtraction).

Consequently, the synthesizer is able to produce a circuit using 49 LUTs, or just over three per bit.

This is not too bad, but it turns out, we can do better. Here is an alternate code fragment that implements the same ALU.

```
A1 <= A and (15 downto 0 => C(1));
B1 <= B xor (15 downto 0 => C(0));
X <= A1 + B1 when C(0) = '0'
     else A1 + B1 + 1;
```

This version was written to implement the circuit shown below.



The adder combines the intermediate signals $A1$ and $B1$. The $C(1)$ input is and-ed with the bits of A, making $A1 = 0$ whenever $C(1) = 0$. The $C(0)$ input is connected to the carry input of the adder and to a set of exclusive-or gates. When it is high, the bits of B are flipped and 1 is added to the result, effectively negating B. So this circuit, either adds A or 0 to B of $-B$, which produces exactly the same result as the original circuit. The payoff is that this circuit can be synthesized using just 16 LUTs, a three-to-one reduction.

Next, let's consider a version of the fair arbiter circuit we discussed earlier in the book. This version is designed to handle four clients and maintains an internal list of the clients to determine which client should get to go next, when two or more simultaneous requests are received. In this case, the first client in the list that is making a request gets to go first, and after being served, it is moved to the end of the list. Here is a table summarizing

the resources used by six different VHDL specifications for this circuit (all produced by different designers).

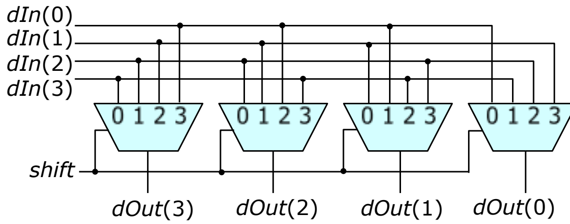
design	flip flops	LUTs
0	19	35
1	27	127
2	28	108
3	19	69
4	17	71
5	27	78

Note the wide variation in the resource usage, especially in the LUT counts. The most expensive design uses more than three times the number of LUTs as the least-expensive design. This kind of variation is not all that unusual, even among experienced designers.

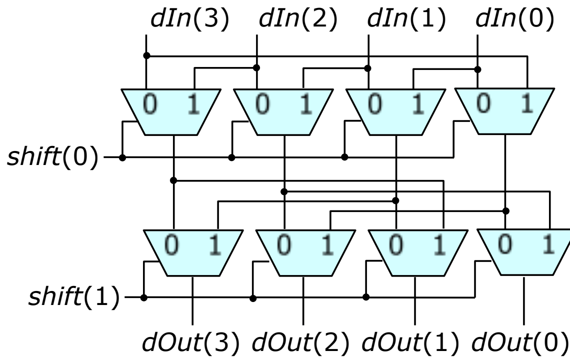
Here's another example that illustrates two fundamentally different architectures for the same function. The code fragment below implements a simple *barrel shifter*, which rotates its input to the left by a specified amount.

```
process (dIn, shift)
begin
   for i in 0 to n-1 loop
      dOut(i) <= dIn(i-shift);
   end loop;
end process;
```

Here, n is the width of the data input and output and *shift* is an input signal that controls the amount by which the individual bits are rotated. At first glance, it would appear that this circuit requires n subtraction circuits plus n multiplexors, each with n data inputs. It turns out that because the subtraction involves the loop index, the synthesizer is able to implement it using the circuit shown below (for the case of $n = 4$).



Observe how the inputs to the different muliplexors are rotated relative to one another. This allows the circuit to be implemented without any explicit subtraction circuits. Now unfortunately, the circuit still requires n multiplexors with n inputs each, and since an n input multiplexor requires about $n/2$ LUTs, this circuit requires about $n^2/2$ LUTs, altogether. For $n = 64$, that's 2048 LUTs, which is more than 20% of the total number in the FPGA used on our prototype board. It turns out that there is an alternative design that uses far fewer LUTs.



Note that the first row of multlexors rotates the input signals by one position when *shift*(0)=1. Similarly, the second row rotates the signals by two positions when when *shift*(1)=1. For larger versions of this circuit, row k rotates the inputs by 2^k positions when *shift*(k)=1. This produces exactly the same result as the original version but uses only $n \log_2 n$ LUTs, which is 384 when $n = 64$, about one fifth the number used by the first design. Here is a VHDL specification for the second version of the barrel shifter.

```
process (dIn, shift)
variable t: unsigned(n-1 downto 0);
begin
   t := dIn;
   for j in 0 to lgN-1 loop
      if shift(j) = '1' then
         t := t(n-(1+2**j) downto 0) & t(n-1 downto n-2**j);
      end if;
   end loop;
   dOut <= temp;
end process;
```

The key point of this example is that there are often fundamentally different implementations of a given logic function, and the resources used by these different implementations can vary widely. Recognizing the existence of a fundamentally different architcture is not something that a circuit synthesizer can do. It is up to a human designer to recognize when the resource usage is excessive, and look for more efficient alternatives.

24.2 Estimating Resource Usage

In order to determine if the resources used for a given circuit are reasonable, we need to have at least a rough idea of what a reasonable resource usage would be. There are three categories of resources we are generally concerned with when designing with FPGAs: flip flops, LUTs and memory.

Flip flops are generally the easiest resource to estimate, since most flip flops are consumed by registers that we specify in a direct way. Any signal that is assigned within the scope of a synchronization condition must be implemented using flip flops, so we can easily account for the number used. There are a few things that can complicate this process. First, when the encoding of state signals used in state machines is left up to the synthesizer, we may not be sure how many flip flops are used. Most often, the synthesizer will use a one-hot encoding, leading to one flip flop per state. Second, synthesizers sometimes replicate flip flops that have large fanouts, in order

to improve performance. Finally, synthesizers will sometimes optimize out flip flops whose outputs are never actually used in the circuit. In all three cases, the synthesis report will include messages that allow you to determine exactly what it did, so if the reported flip flop counts do not match what you expect, you can usually resolve the discrepancy fairly easily by examining the synthesis report.

The second category of resource is LUTs. It is often difficult to account for LUTs precisely, but there is one easy thing to check to get a sense of whether the LUT count is reasonable or not. Since FPGAs are equipped with equal numbers of LUTs and flip flops, any circuit that uses many more LUTs than it does flip flops is likely to end up wasting a significant fraction of the available resources. So, if your circuit uses an appropriate number of flip flops and has a LUT count that is roughly comparable to the number of flip flops used, you can be reasonably confident that your design is well-balanced and is probably not using more LUTs than it should. Now, just because a design uses more LUTs than flip flops does not necessarily mean that its resource usage is excessive, but it does mean that we may need to look at it more closely to make sure we understand how LUTs are being used.

To make it easier to estimate the LUTs used by a given VHDL specification, it's helpful to know how many LUTs are used by common building blocks. The table below lists resource counts for a variety of building blocks that are used by circuit synthesizers. Here, n is the width of the data handled by the component.

component	no constant input	one constant input
incrementer/adder	n	n
select bit	$n/2$	0
modify selected bit	$n+n/4$	1
decoder	$n+n/8$	0
equality compare	$n/2$	$n/4$
less-than compare	n	$n/2$
shift/rotate	$n\log_2 n$	0
loadable register	0	-
loadable counter	n	-

Let's start with the middle column, which gives the LUT counts for the general case when all inputs are signals whose values can change as the circuit operates. As mentioned earlier, FPGAs include certain hidden resources that enable commonly occuring functions to be implemented more efficiently. In particular, there are special circuits to implement carry logic in ripple-carry adders and similar circuits. These allow adder circuits to be implemented with just one LUT per bit, rather than the two LUTs that would be required if we had to implement the carry logic explicitly with LUTs. Similarly, there are "helper circuits" that allow an n input multiplexor to be implemented using about $n/2$ LUTs, rather than the $n-1$ we would use if we built such a mux using 2-to-1 muxes arranged in a tree. This allows us to select one bit from an n bit vector using $n/2$ LUTs. To change a selected bit of an n bit vector, we need $n + n/4$ LUTs, making assignment to an element of a vector a relatively expensive operation. Comparing two n bit signals for equality takes $n/2$ LUTs, but comparing them to see if one is less than the other requires n.

The right hand column of the table gives the LUT counts when one of the inputs to the component has a constant value. So to select a bit from a vector, when the index of the selected bit is a constant requires no LUTs in the synthesized circuit. Similarly, when comparing an input signal to a constant, we can cut the number of LUTs in half, compared to the general case.

By examining the VHDL for a circuit, we can often identify where the synthesizer will need to instantiate various components, such as adders, counters and comparison circuits. We can then use the information in the table to quickly estimate the LUTs used by these components. In circuits that process parallel data with word widths of 16 or more bits, these elements typically account for most of the resources used.

We can also use this information to try to identify places where we can reduce the resources used. For example, we might have a circuit with three 32 bit adders, that are always used at different times. It may make sense in this situation to use one adder for all three additions, rather than three separate adders.

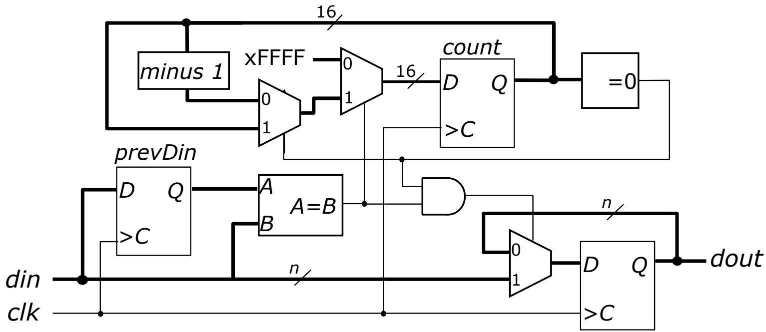
The third resource that we must account for in our circuits is memory.

FPGAs are often equipped with configurable SRAM blocks. In the FPGAs used on our prototype boards, these *block* RAMs have 18 Kbits and can be configured in a variety of ways. LUTs can also be used to implement memory blocks, and if we only need a small amount of memory (say a few hundred bits), it makes more sense to implement the memory using LUTs than using a block RAM. The number of LUTs needed to implement an n bit memory is $n/16$, so for example, a 64×8 bit memory can be implemented using 32 LUTs. Note that memory provides a much more efficient way of storing large amounts of data than flip flops. In most situations, when we have lots of data to store, it makes sense to store it in memory if we can.

24.3 Estimating and Reducing Resource Usage

In this section, we'll look at several examples to see how we can estimate the resources required by a given circuit. We will also compare our own estimates to the information found in the synthesis report, to get a better understanding of the synthesized circuits, and to identify opportunities to reduce the resource usage.

Let's start with a familiar circuit, the debouncer that we've used to debounce the buttons and knobs on the prototype boards. Recall that the VHDL specification of this circuit include a generic width parameter, so we can instantiate debouncers of various sizes. The debouncer has an internal counter that it uses to delay the propagation of the input signals until they've had a chance to "settle". It also stores the input signals on each clock tick so that it can compare the previous value to the current value. The figure below shows a circuit diagram, based on the VHDL specification.



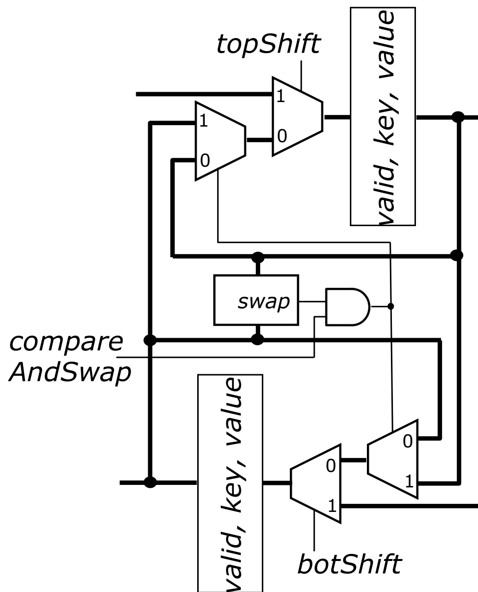
Let's start by accounting for the number of flip flops. The counter is 16 bits wide and the *prevDin* and *dOut* registers are both n bits wide, where n is the generic width. So for example, if $n = 8$, we have a total of 32 flip flops in this circuit. This is exactly what the synthesis report gives as the flip flop count.

To estimate the LUTs used, we need to identify the various elements of the circuit. The top section of the diagram implements the counter plus an "equal-0" comparator. The counter can be implemented using one LUT per bit, while the equality counter requires one LUT per group of four bits, giving us 16+4=20 LUTs for this part of the circuit. The general equality comparator requires $n/2$ LUTs, so for $n = 8$ that's 4 more. This gives us a total of 24 LUTs, while the synthesis report gives a total of 27. This is a small enough discrepancy that we won't worry about it. The objective is not to account for every single LUT but to confirm that we have a reasonably accurate understanding of how the resources are being used.

Since the number of LUTs used by this circuit is smaller than the number of flip flops, we can reasonably conclude that this circuit is about as efficient as we can reasonably make it. However, there is still an opportunity to reduce overall resource usage, with respect to the debouncer circuit. Recall that in our design of the binary input module, we used one debouncer for the button signals, while the knob interface circuit used a separate debouncer for the knob signals. Each of these debouncers has its own 16 bit counter, which is clearly somewhat redundant. If we had designed the circuit using a single 7 bit wide debouncer, instead of two smaller ones, we could have saved

16 flip flops and 20 LUTs, cutting the overall resource usage for debouncing by nearly a factor of 2.

Next, let's look at priority queue circuit, that we discussed in Chapter 12. Recal that this circuit consists of an array of cells with two rows and k columns. Each cell stores a (*key, value*) pair, including a *valid* bit used to identify the cells that are "in-use". To insert a new entry, we shift the top row to the left, then do a "compare-and-swap" operation in each column. To remove an entry, we shift the bottom row to the right, then do a compare-and-swap. The diagram below shows a typical column in the array.



The block labeled *swap* consists of a less-than comparator, plus a small amount of additional logic to compare the valid bits. If the *key* and *value* are each n bits wide, one column in the array contains $4n+2$ flip flops. Since each register can be loaded from two different sources, we need one LUT per register bit, plus n LUTs for the less-than comparison. This gives us a total LUT count of $5n+2$ or 82 when $n = 16$. The synthesis report for this circuit gives 91 LUTs per column, which is within about 10% of our estimate.

In this case, the LUT count is about 40% larger than the flip flop count for the case of $n = 16$. This is reasonably close, and since there is no obvious way to reduce the number of LUTs used by the column circuit, we can conclude that the circuit is probably about as efficient as we can make it.

Still, it is worth asking if the overall approach used by the priority queue could be improved upon. If we wanted to instantiate a priority queue that could hold say 1000 key/value pairs where the key and value are both 16 bits, this circuit would require 33,000 flip flops, far more than we can accommodate on the FPGAs used on on our prototype boards. However, we could store this information in just two of the 20 block RAMs that the FPGAs contain. Is there a way to use this memory to store the (key,value) pairs, while still supporting fast insertion and deletion? It turns out that there are ways to do this, based on software algorithms for priority queues. While these approaches do require more time to perform an insertion or deletion, they can accommodate much larger sets of (key,value) pairs. The bottom line is that the given circuit is appropriate in applications where performance is critical and the number of pairs is not too large. If we can tolerate slower performance and need to handle hundreds or thousands of pairs, an alternate approach may be called for.

Next let's turn to a larger example, the WASHU-2 processor. This circuit is large enough that in order to estimate the resource use it helps to build a spreadsheet listing the key elements of the circuit and an estimate of the resources they use. Here is such a spreadsheet for the WASHU-2.

	flip flops	LUTs
PC	16	32
IREG	16	0
IAR	16	0
ACC	16	32
this	16	16
ALU	16	80
aBus	0	32
dBus	0	0
target	0	16
decoder	0	40
state	17	17
tick	4	4
dispReg	0	32
if-conditions	0	25
total	101	326
actual/area	100	361
actual/speed	106	389

The first five lines list the flip flops and LUTs used in connection with the main processor registers. The PC requires 32 LUTs, since it can be incremented and can be loaded from either the IREG or the data bus. Similarly, the ACC requires 32 since it can be loaded either from the IREG (during constant load instructions), the data bus or the ALU. The IREG and IAR do not require any extra LUTs because they are both loaded only from the data bus. The ALU implements addition, negation and logical-AND. These each require one LUT per bit, plus we need two more LUTs per bit to select from among four possible values. This leads to our estimate of 80 LUTs altogether. The address bus signal uses no flip flops, but does require two LUTs per bit to select from among three possible sources (the IREG ,PCand the IAR). The branch target signal requires one LUT per bit, since it requires an adder to add the PC to the offset in the IREG. The instruction decoding function requires about 40 LUTs to implement the conditions implied by its case statements, and to construct its 17 bit result. The *state* signal is implemented using a one-hot

encoding and since there are 17 states, it requires 17 flip flops. We need one LUT per flip flop to handle the updating of the state. The *dispReg* signal is the output that sends a selected register to the console. It requires a 4:1 multiplexor with 2 LUTs per bit. The line labelled *if-conditions* is an estimate of the number of LUTs used by for the conditions in all of the if statements in the two processes. Most of these conditions require a single LUT.

The total from our estimates is 101 flip flops and 326 LUTs. The circuit was synthesized using two different optimization criteria (these are specified in the process properties of Project Navigator). When optimized for area, the synthesized circuit had 100 flip flops and 361 LUTs. When optimized for speed, it had 106 flip flops and 389 LUTs. These are both reasonably consistent with the estimates, allowing us to be comfortable with our understanding of how the resources are being used. We can check our understanding by examing the high level synthesis section of the synthesis report.

```
Synthesizing <cpu>...
Found finite state machine for signal <state>.
-------------------------------------------------------------
| States          | 17                                      |
| Transitions     | 54  ...                                 |
-------------------------------------------------------------
Found 16-bit tristate buffer for signal <aBus>.
Found 16-bit tristate buffer for signal <dBus>.
Found 16-bit 4-to-1 multiplexer for signal <dispReg>.
Found 16-bit register for signal <acc>.
Found 16-bit adder for signal <alu$addsub00>.
Found 16-bit register for signal <iar>.
Found 16-bit register for signal <iReg>.
Found 16-bit register for signal <pc>.
Found 16-bit adder for signal <pc$addsub00> created: line 146.
Found 16-bit adder for signal <target>.
Found 16-bit register for signal <this>.
Found 4-bit register for signal <tick>.
Found 4-bit adder for signal <tick$add00> created: line 134.
Summary: inferred  1 Finite State Machine(s).
```

```
inferred  84 D-type flip-flop(s).
inferred   4 Adder/Subtractor(s).
inferred  16 Multiplexer(s).
inferred  32 Tristate(s).
```

This identifies elements like state machines, registers and adders, and summarizes the overall resources used. In a circuit that instantiates additional sub-components, a separate section like this appears for each sub-component.

We'll finish this chapter with a similar analysis of the resources used by the WASHU-2's console circuit. As before, we'll start with a spreadsheet.

	flip flops	LUTs
debouncer	24	22
knob interface	43	30
lcd display	36	120
snoopAdr	16	32
snoopData	16	48
snoopCount	16	16
lcdCounter	20	20
regSelect	0	6
nuChar	0	44
hex2Ascii	0	8
prevDBtn	4	0
singletons	2	10
total	177	356
actual/area	160	389

Here, the first three lines are for the three sub-components of the console. The values shown came from synthesizing the sub-components separately. The *snoopAdr* signal is a 16 bit register along with an adder/subtracter, that is used to adjust the value as the knob turns. We estimate 2 LUTs per bit for this. For the *snoopData* signal we estimate 3 LUTs per bit, since in this case, we must also be able to load a new value from the data bus. The *snoopCount* is a 16 bit counter, requiring 1 LUT per bit. The *lcdCounter* is a 20 bit counter, also requiring 1 LUT per bit. The *regSelect* is a two bit signal specified by the following conditional signal assignment.

```
regSelect <= "00" when selekt <= slv(4,5) else
             "10" when selekt <= slv(10,5) else
             "01" when selekt <= slv(20,5) else
             "11";
```

The three less-or-equal compaators each require 2 LUTs, leading to our estimat of 6 LUTs for this signal. The *nuChar* signal is eight bits wide and is derived from one of 12 different four bit signals, selected by a large case statement. We estimate 5 LUTs per bit in this case, giving a total of 40. The *prevDbtn* signal is just a delayed copy of the debounced button signals, used to detect button presses. The line labeled *singletons* is for the two one bit registers, *singleStep* and *writeReq*.

When we compare our estimates to the values in the synthesis report, there is one surprise. Our estimate of the flip flop count is too large by 16, which is a fairly large discrepancy when it comes to flip flops. If we look more closely at the synthesis report, we can find the explanation in the following message.

```
INFO:Xst:2146 - In block <console>, Counter <lcdCounter>
<snoopCnt> are equivalent, XST will keep only <lcdCounter>.
```

What the synthesizer has noticed is that both of these counters are initialized to zero and incremented on every clock tick. While *lcdCounter* has four more bits that *snoopCount*, its low order 16 bits are always exactly the same as *snoopCount*. Hence, it can omit *snoopCount* and use the low order bits of *lcdCounter* in its place. This is how it comes up with 16 fewer flip flops than we got from our own analysis.

There is actually one other surprise in the synthesis report, but this one is not so pleasant. The suprise is contained in the following lines.

```
Found 16x8-bit ROM for signal <nuChar$rom0000> at line 261.
Found 16x8-bit ROM for signal <nuChar$rom0001> at line 264.
...
Found 16x8-bit ROM for signal <nuChar$rom0011> at line 282.
```

Now the circuit does contain the constant array *hex2Ascii* that we expect to be implemented as a 16×8 ROM. This ROM can be implemented using

8 LUTs, but the synthesis report lists 12 copies of this circuit for a total of 96 LUTs. Why should this be happening? The culprit turns out to be this process.

```
process (cpuReg, snoopAdr, snoopData, selekt) begin
   case selekt is
   when "00000" | "00110" | "10000" | "10110" =>
      nuChar <= c2b(hex2Ascii(int(cpuReg(15 downto 12))));
   when "00001" | "00111" | "10001" | "10111" =>
      nuChar <= c2b(hex2Ascii(int(cpuReg(11 downto 8))));
   ...
   when "01100=>nuChar<=c2b(hex2Ascii(
                               int(snoopAdr(15 downto 12))))
   when "01101=>nuChar<=c2b(hex2Ascii(
                               int(snoopAdr(11 downto 8))));
   ...
```

The individual cases are never needed at the same time, but the synthesizer is not able to figure that out by itself. Consequently, it has created separate copies of the *hex2Ascii* ROM for each of the 12 distinct cases. We can get a more efficient circuit by re-writing this process as shown below.

```
process (cpuReg, snoopAdr, snoopData, selekt) begin
   case selekt is
   when "00000" | "00110" | "10000" | "10110" =>
      showDigit <= cpuReg(15 downto 12);
   when "00001" | "00111" | "10001" | "10111" =>
      showDigit <= cpuReg(11 downto 8);
   when "00010" | "01000" | "10010" | "11000" =>
      showDigit <= cpuReg(7 downto 4);
   when "00011" | "01001" | "10011" | "11001" =>
      showDigit <= cpuReg(3 downto 0);
   when "01100" => showDigit <= snoopAdr(15 downto 12);
   when "01101" => showDigit <= snoopAdr(11 downto 8);
   when "01110" => showDigit <= snoopAdr(7 downto 4);
   when "01111" => showDigit <= snoopAdr(3 downto 0);
```

```
      when "11100" => showDigit <= snoopData(15 downto 12);
      when "11101" => showDigit <= snoopData(11 downto 8);
      when "11110" => showDigit <= snoopData(7 downto 4);
      when "11111" => showDigit <= snoopData(3 downto 0);
      when others => showDigit <= x"0;
      end case;
   end process;
   nuChar <= x"20" when selekt(3 downto 1)="010"
                     or selekt(3 downto 1)=101
            else c2b(hex2Ascii(int(showDigit)));
```

Here, the process defines the new *showDigit* signal, which is the four bit digit that should be displayed on the LCD display. The final assignment to *nuChar* uses *showDigit* as the argument to *hex2Ascii* eliminating the redundant copies. Synthesizing the circuit with this change leads to a savings of 58 LUTs, a reduction of about 15%.