5

Register-Transfer Level (RTL) Design

5.1 INTRODUCTION

In the previous chapters, we’ve defined the combinational and sequential components needed to build digital systems. In this chapter, we’ll learn to build interesting and useful digital systems from those components. In particular, we’ll put together datapath components to build datapaths, and we’ll use controllers to control those datapaths. The combination of a controller and datapath is known as a processor. Some processors, like those in personal computers, are programmable—those processors are the focus of Chapter 8. Other processors are custom-designed for a particular task, and are not programmable—design of such custom processors is the focus of this chapter.

Digital designers today focus largely on designing custom processors, as opposed to designing lower-level digital components. We can define a custom processor as a digital circuit that implements a computer algorithm—a sequence of instructions that carry out a particular task. For example, we can define an algorithm to filter out noise from a digitized stream of audio, and we can then create a processor to implement that algorithm. Another algorithm might encrypt data for secure electronic commerce purposes. An algorithm might compare a fingerprint to a set of 10,000 fingerprints to quickly enable a police officer to determine if someone is a wanted criminal. An image processing algorithm might detect a tank in a large video image. Beamforming, part of the ultrasound machine example in the previous chapter, can be thought of as another algorithm, implemented using the processor design described in that chapter. In fact, several of our examples in the previous chapter, like the above-mirror display, DIP-switch-based calculator, and color space converter, can actually be thought of as very simple processors implementing simple algorithms.

Processors can be designed using different design methods. The most common method in practice today is known as register-transfer level design. Register-transfer level design, or RTL design, actually consists of a wide variety of approaches, but in general, a designer specifies the registers of a design, describes the possible transfers and operations performed on input, output, or register data, and defines the control that specifies when to transfer and operate on data.

Recall the design processes we defined for combinational logic design in Chapter 2, and for sequential logic (controller) design in Chapter 3:
In the combinational logic design process outlined in Table 2.5,
1. The first step was to capture the desired behavior of the combinational logic, with either a truth table or an equation.
2. The remaining steps were to convert the behavior to a circuit.

In the sequential logic (controller) design process in Table 3.2,
1. The first step was to capture the desired behavior of the sequential logic, using a finite-state machine.
2. The remaining steps were to convert the behavior to a circuit.

It should therefore come as no surprise that:

1. The first step of an RTL design method will be to capture the desired behavior of the processor. We'll introduce the concept of a high-level state machine for capturing RTL behavior.
2. The remaining steps will be to convert the behavior to a circuit.

Figure 5.1 illustrates the idea that the design process can be viewed as first capturing behavior and then converting the behavior to structure. That process applies regardless of whether we are performing combinational logic design, sequential logic design, or RTL design.

In this chapter, we will introduce the RTL design process, also known as the RTL design method. As the process is largely creative, we will utilize numerous examples to illustrate the process. We will also introduce several high-level components that are useful during RTL design, including memory components and queue components.

5.2 RTL DESIGN METHOD

RTL design is carried out using a wide variety of methods in practice, but it may be useful to define a general method as in Table 5.1

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>Capture a high-level state machine</td>
</tr>
<tr>
<td>Step 2</td>
<td>Create a datapath</td>
</tr>
<tr>
<td>Step 3</td>
<td>Connect the datapath to a controller</td>
</tr>
<tr>
<td>Step 4</td>
<td>Derive the controller's FSM</td>
</tr>
</tbody>
</table>
5.2 RTL Design Method

A fifth step may be necessary, in which one selects a clock frequency. Designers seeking high performance may choose a clock frequency that is the fastest possible based on the longest register-to-register delay in the final circuit.

Implementing the controller’s FSM as a sequential circuit, as we learned in Chapter 3, would then complete the design.

Notice that the first step captures the desired behavior, while the remaining steps convert that behavior to a circuit.

We’ll first provide a small and simple example as a “preview” of the RTL design method’s steps, before we define each step in more detail.

**Example 5.1** Soda machine dispenser

We are to design a processor for a soda dispenser. A coin detector provides our processor with a 1-bit input $c$ that becomes 1 for one clock cycle when a coin is detected, and an 8-bit input $a$ indicating the coin’s value in cents. Another 8-bit input $s$ indicates the cost of a soda (this cost can be set by the machine owner). Once the processor has seen coins whose value equals or exceeds the cost of a soda, the processor should set an output bit $d$ to 1 for one clock cycle, causing a soda to be dispensed (this machine has only one type of soda). The system does not give change—any excess money is kept. Figure 5.2 provides a block symbol of the system.

**Step 1** of our RTL design method is to capture the desired behavior of the system. Figure 5.3 shows a high-level state machine describing the desired behavior. The first state, Init, sets the output $d$ to 0 and initializes a local register tot to 0. tot will keep track of how many cents the system has seen so far. The state machine then enters state Wait. (Recall from Chapter 3 that a transition with no condition has an implicit “true” condition, and thus transitions on the next rising clock edge.) The FSM stays there as long as no coin is detected and the total cents seen so far is less than the cost of a soda. When a coin is detected, the state machine goes to state Add, which adds the coin’s value to tot, and then returns to state Wait. Once tot is greater than or equal to (in other words, not less than) the cost of a soda, the state machine goes to state Disp, which dispenses a soda by setting $d$ to 1. The state machine then returns to state Init.

**Step 2** is to create a datapath. We’ll need a local register for tot, an adder connected to tot and $a$ to compute $\text{tot} + a$, and a comparator connected to tot and $s$ to compute $\text{tot} < s$. The resulting datapath appears in Figure 5.4.
Step 3 is to connect the datapath to a controller. Figure 5.5 shows the connections. Notice that the controller's inputs and outputs are all just one-bit signals.

Step 4 is to derive the controller's FSM. The FSM has the same states and transitions as the high-level state machine, but utilizes the datapath to perform any data operations. Figure 5.6 shows the FSM for the controller. In the high-level state machine, state Init had a data operation of \( \text{tot} \rightarrow 0 \) (\( \text{tot} \) is 8 bits wide, so that assignment of 0 is not a single-bit operation). We replace that assignment by setting \( \text{tot} \_\text{clr} \rightarrow 1 \), which clears the \( \text{tot} \) register to 0. State Wait's transitions had data operations comparing \( \text{tot} < s \). Now we have a comparator computing that comparison for the controller, so the controller need only look at the result of that comparison in the signal \( \text{tot} \_\text{lt} \_s \). State Add had a data operation of \( \text{tot} = \text{tot} + a \). The datapath computes that addition for the controller using the adder, so the controller merely needs to set \( \text{tot} \_\text{id} \rightarrow 1 \) to cause the addition result to be loaded into the \( \text{tot} \) register.

To complete the design, we would implement the controller's FSM as a state register and combinational logic. Figure 5.7 shows a partial state table for the controller, with the states encoded as Init: 00, Wait: 01, Add: 10, and Disp: 11. To complete the controller design, we would complete the state table, create a 2-bit state register, and create a circuit for each of the five outputs from the table, as discussed in Chapter 3. Appendix C provides details of completing the controller's design. That appendix also traces through the functioning of the controller and datapath with one another.
The previous example gave a preview of the RTL design method. Notice that we started with a high-level state machine, which wasn’t just an FSM because there were local registers declared, and because there were data operations (rather than just Boolean operations) in the states and on the transitions. We then created a datapath to implement those local registers and to carry out the data operations. We further needed a controller to control that datapath. We defined the behavior of that controller to be the same as the behavior of the high-level state machine, except the controller’s FSM used datapath control signals to carry out and evaluate the datapath operations. Finally, we could design the controller using Chapter 3’s controller design process.

We now discuss each RTL design method step in more detail, while illustrating each step with another example.

Step 1—Creating a High-Level State Machine

A high-level state machine is a computation model similar to a finite-state machine, but with additional features that enable the description of computations involving more than just Boolean data.

Recall that a finite-state machine (FSM) consists of inputs, outputs, states, state actions (a mapping of states to output values), and state transitions (a mapping of states and inputs to next states). However, the inputs and outputs of an FSM are limited to Boolean types, actions are limited to Boolean equations, and transition conditions are limited to Boolean expressions. These limitations make specifying of computations involving data cumbersome, other than for just single-bit data.

Figure 5.3 showed a high-level state machine describing the behavior of a soda dispenser processor. Notice that the state machine is not an FSM because of the several reasons highlighted in Figure 5.8. One reason is because the state machine has inputs that are 8-bit types, whereas FSMs only allow inputs and outputs of Boolean types (a single bit each). Another reason is because the state machine declares a local register tot to store intermediate data, whereas FSMs don’t allow local data storage—the only “stored” item in an FSM is the state itself. A third reason is because the state actions and transition conditions involve data operations, like \( \text{tot} = 0 \) (remember that tot is 8-bits wide), \( \text{tot} < s \) (there’s no “<” Boolean operator), and \( \text{tot} = \text{tot} + a \) (where the “+” is addition, not OR, and there’s no addition Boolean operator), whereas an FSM allows only Boolean equations and expressions.

Therefore, a useful form of high-level state machine is an extension of an FSM in which:

- inputs and outputs may involve data types beyond just single bits,
- local registers may be declared (of various data types), and
- actions and conditions may involve general arithmetic equations and expressions, rather than just Boolean equations and expressions.
Such a high-level state machine is not the only possible extension to an FSM. Dozens of varieties of extended FSMs exist. However, we will be utilizing the above-described extended FSM variety throughout this chapter. That particular variety of high-level state machine is sometimes called an **FSM with data**, or **FSMD**.

We will continue to use the following conventions for high-level state machines, which we also used for FSMs:

- Each transition is implicitly ANDed with a rising clock edge.
- Any bit output not explicitly assigned a value in a state is implicitly assigned a 0. Note: this convention does not apply for multibit outputs.

We now provide another example of describing a system using a high-level state machine.

**Example 5.2 Laser-based distance measurer—high-level state machine**

There are countless applications that require one to accurately measure the distance of an object from a known point. For example, road builders need to accurately determine the length of a stretch of road. Map makers need to accurately determine the locations and heights of hills and mountains and the sizes of lakes. A giant crane for constructing skyscraper buildings needs to accurately determine the distance of the sliding crane arm from the base. In all of these applications, stringing out a tape measure to measure the distance is not very practical. A better method involves laser-based distance measurement.

In laser-based distance measurement, a laser is pointed at the object of interest. The laser is briefly turned on, and a timer is started. The laser light, traveling at the speed of light, travels to the object and reflects back. A sensor detects the reflection of the laser light, causing the timer to stop. Knowing the time \( T \) taken by the light to travel to the object and back, and knowing that the speed of light is \( 3 \times 10^8 \) meters/second, we can compute the distance \( D \) easily by the equation: \( 2D = T \) seconds \( \times 3 \times 10^8 \) meters/second. Laser-based distance measurement is illustrated in Figure 5.9.

![Figure 5.9 Laser-based distance measurement.](image)

Let’s design a processor to control the laser and the timer and to compute distances up to 2000 meters. A block diagram of the system is shown in Figure 5.10. The system has a bit input \( B \), which equals 1 when the user presses a button to start the measurement. Another bit input \( S \) comes from the sensor, and is 1 when the reflected laser is detected. A bit output \( L \) controls the laser, turning the laser on when \( L \) is 1. Finally, an \( N \)-bit output \( D \) indicates the distance in binary, in units of meters—we’ll assume a display converts that binary number into a decimal number and displays the results on an LCD for the user to read. \( D \) will have to be at least 11 bits, since 11 bits can represent the numbers 0 to 2047, and we want to measure distances up to 2000 meters. Let’s make \( D \) 16 bits.
Step 1—Create a high-level state machine.
We can describe the overall control of the system using a high-level state machine. To facilitate the creation of the state machine, we enumerate the sequence of events underlying the measurement system:

- The system powers on. Initially, the system's laser is off and the system outputs a distance of 0 meters.
- The system should then wait for the user to initiate measurement by pressing a button, $B$.
- After the button is pressed, the system should turn the laser on. We'll choose to leave the laser on for one clock cycle.
- After the laser is pulsed, the system should wait for the sensor to detect the laser's reflection. Meanwhile, the system should count how much time passes from the time the laser was pulsed until the reflection is sensed.
- After the reflection is detected, the system should use the amount of time passed since the laser was pulsed to compute the distance to the object of interest. The system should then return to waiting for the user to press the button so that a new measurement can be taken.

The above sequence guides our construction of a high-level state machine. We begin with an initial state, which we call $S_0$. $S_0$'s task is to ensure that when our system powers on, it does not output an incorrect distance, and it does not turn the laser on (possibly injuring the unsuspecting user). Specifying this behavior as a high-level state machine is straightforward and seen in Figure 5.11. Notice that the high-level state machine differs from an FSM in that the state's actions use a data type that is larger than one bit (namely, $D$ is 16 bits). However, the high-level state machine itself follows the convention that every transition is implicitly ANDed with a rising clock edge, so the state machine only transitions during clock edges (just like for an FSM).

Note that even though the assignments $L = 0$ and $D = 0$ look the same, the assignment $L = 0$ assigns a 0 bit to the one-bit output $L$, whereas the assignment $D = 0$ assigns the 16-bit binary number 0 (which is actually 0000000000000000) to the 16-bit output $D$. Some other notations distinguish bit assignments from data assignments using different notations, such as enclosing a bit in single quotes. For example, the bit assignment $L = 0$ could be written instead as $L = '0'$.

After initialization, the measurement system waits for the user to press the button $B$, which initiates the measurement process. When the user presses the button, $B$ will equal 1, and the measurement system should proceed to activate the laser. To perform the waiting, we add a state after $S_0$, which we call $S_1$, shown in Figure 5.12. The shown transitions cause the state machine to remain in state $S_1$ while $B = 0$ (meaning $B^*$ is true).
When $B = 1$, the laser should stay on for one cycle. In other words, when $B = 1$, the state machine should transition to a state that turns the laser on, followed by a state that turns the laser off. We'll call the laser-on state $S_2$ and the laser-off state $S_3$. Figure 5.13 shows how $S_2$ and $S_3$ are connected in the high-level state machine.

In state $S_3$, the state machine should wait until the sensor detects the laser's reflection ($S = 1$). The state machine remains in $S_3$ while $S = 0$. As mentioned in the earlier sequence of events, the state machine should meanwhile count the duration between the laser being pulsed and the laser's reflection being sensed. From the discussion of timers in Chapter 4, we know that with a given clock period, we can measure time by counting the number of clock cycles and multiplying that number by the clock period (time = cycles * (1/clock frequency)). Thus, we use a local register, which we'll call $Dctr$, to count clock cycles. The state machine increments $Dctr$ as long as the state machine is waiting for the laser's reflection. (For simplicity, we ignore the possibility that no reflection is ever detected.) We must also initialize $Dctr$ to 0, which we choose to do in state $S_1$. With these modifications, our high-level state machine is seen in Figure 5.14.

Once the reflection is detected ($S = 1$), our high-level state machine should compute the distance $D$ that is being measured. From Figure 5.9, we know that $2D = Tsec * 3 \times 10^8$ m/sec. We also know that the time $T$ in seconds is $Dctr * (1/$clock frequency$)$. To simplify the system's design, let's assume the clock frequency is $3 \times 10^8$ Hz, or $300$ MHz. Since light travels $3 \times 10^8$ meters per second,
each clock cycle would thus correspond to one meter. Thus with a 300 MHz clock, Dctr counts the number of meters that the laser beam traveled from the measurer to the object and back to the measurer. To count just the distance from the measurer to the object, we divide Dctr by 2 (algebraic simplification of the equations in this paragraph verify that D = Dctr/2). We'll perform this calculation in a state we will call S4. Our final high-level state machine is shown in Figure 5.15.

![High-level state machine for measurement system](image)

Figure 5.15 High-level state machine for measurement system: calculating the value of D.

We can summarize the behavior of the high-level state machine in Figure 5.15 as follows:

- **S0** is the initial state. In state S0, the state machine initializes the laser to off by setting L=0 and sets the output D=0 too. The machine then transitions to S1.
- **S1** clears Dctr to 0 and then waits until the button is pressed. When the button is pressed, the machine transitions to state S2.
- **S2** turns on the laser. The machine then transitions to S3.
- **S3** turns off the laser and increments Dctr every clock cycle (with a 300 MHz clock, every cycle corresponds to one meter). The machine stays in S3, incrementing Dctr during each clock cycle, until the reflection is sensed, at which time the machine transitions to state S4.
- **S4** sets the output D to the counted number of cycles divided by two, which corresponds to the measured distance in meters. The machine then returns to state S1, which waits for the button to be pressed again.

A real laser-based distance measurer might use a faster clock frequency in order to measure distance with a greater precision than just 1 meter.

The high-level state machine described above is just one type of FSM variation. A different state machine variation that was previously quite popular was called Algorithmic State Machines, or ASMs. ASMs are similar to flowcharts, except that ASMs include a notion of a clock that enables transitions from one state to another (a traditional flowchart does not have an explicit clock concept). ASMs, like flowcharts, contain more "structure" than a state machine. A state machine can transition from any state to any other state, whereas an ASM restricts transitions in a way that causes the computation to look more like an algorithm—an ordered sequence of instructions. An ASM uses several types of boxes, including state boxes, condition boxes, and output boxes. ASMs typically also allowed local data storage and data operations.

The advent of hardware description languages (see Chapter 9) seems to have largely replaced the use of ASMs, as hardware description languages contain the constructs supporting algorithmic structure, and much more. Thus, we do not describe ASMs further.
Step 2—Creating a Datapath

Given a high-level state machine, we want to create a datapath that can implement all the data storage and computations on non-Boolean data types present in the high-level state machine. Doing so will enable us to then replace the high-level state machine by an FSM that merely controls the datapath. We can decompose the “create a datapath” step into several substeps:

Step 2: Create a datapath

(a) Make all data inputs and outputs to be datapath inputs and outputs.
(b) Implement the data storage by adding a register component into the datapath for every declared register in the high-level state machine. Furthermore, we typically want to add a register component for every data output.
(c) Methodically examine each state and each transition, adding and connecting new datapath components to implement new data computations. We add multiplexors in front of component inputs as they become necessary in order to share a component among multiple signals that use the same component in different states. Sometimes we find that a component already exists (e.g., a register) but that we need to add a new control input to that component (e.g., a clear input on a register to set the register to 0).

A common term used to describe the adding of a component into a design is instantiation. Thus, we say that we “instantiate a new register” rather than we “add a new register.” Using the term “instantiate” rather than “add” helps avoid possible confusion with the use of the term “add” to mean arithmetic addition (e.g., saying “we add two registers” could otherwise be confusing). When we instantiate a new component, we should give that component a name that is unique from any other datapath component name. So if we instantiate a register, we might call it “Register1.” If we instantiate another register, we might call it “Register2.” Actually, we should give meaningful names whenever possible. So we might call one register “TemperatureReg,” and another register “HumidityReg.”

When we instantiate a new component, we may create additional datapath inputs corresponding to the control inputs of the component. For example, instantiating a register will create a new datapath input corresponding to the register’s load and clear control inputs. We should give unique names to each new datapath control input, ideally describing which component the input controls and the control operation performed. For example, if we instantiate a register named Register1, we might then create two new datapath inputs named Register1_load and Register1_clear. Likewise, we may need to utilize control outputs of a component, like the output of a comparator, in which case we should give those outputs unique names too.

**EXAMPLE 5.3** Laser-based distance measurer—Creating a datapath

We now continue Example 5.2 by proceeding to the second step of the RTL design method.

Step 2—Create a datapath

We can follow the substeps of this step to create the datapath shown in Figure 5.17:
Output D is a data output (16 bits), so we make D an output of the datapath, as shown in Figure 5.16(i).

We need a register to implement the 16-bit local register Dreg. Noting that the operations on Dctr are clear (in state S1) and increment (in state S3), we can implement that register by instantiating a 16-bit up-counter, as shown in Figure 5.16(ii). Furthermore, as we want to control when the output D changes (notice that we only change D in state S4), we instantiate a 16-bit register Dreg at the output D, as shown in Figure 5.16(iii). We extend the Dctr counter and Dreg register control signals to be inputs to the datapath, with each signal having a unique name, as in Figure 5.16(iv).

Figure 5.16 Partial datapath for the laser-based distance measureur.

Noting that S3 writes D with Dctr divided by 2, we insert a right shifter between Dctr and Dreg to implement the divide by 2, as shown in Figure 5.17.

Figure 5.17 The datapath for the laser-based distance measurement system.

The resulting datapath in Figure 5.17 is a very simple datapath, but a datapath nonetheless.

The previous example did not require any multiplexors, so we’ll illustrate separately why sometimes multiplexors must be instantiated. Consider the sample high-level state machine portion shown in Figure 5.18(a). Figure 5.18(b) shows the datapath after implementing the actions of state T0. Those actions require an adder, with the $E$ and $F$ registers connected to the A and B inputs of that adder. Figure 5.18(c) shows that datapath after implementing the actions of state T1. That state also requires an adder, but because one already exists in the datapath, we need not instantiate another adder. However, the $R$ and $G$ registers must
connect to the A and B inputs of that adder, but those inputs of the adder already have connections from E and F. We therefore need to instantiate multiplexors, as shown in Figure 5.18(d). Notice that we create unique names for each mux's control input.

**Step 3—Connecting the Datapath to a Controller**

Step 3 of the RTL design method is actually quite straightforward. We simply create a controller block having the system's Boolean inputs and outputs, and we connect the controller block with the datapath control inputs and outputs.

**EXAMPLE 5.4 Laser-based distance measurer—Connecting the datapath to a controller**

Continuing the previous example, we proceed to step 3 of the RTL design method:

**Step 3—Connect the datapath to a controller.**

We connect the datapath to a controller as shown in Figure 5.19. We connect the control inputs and outputs (B, L, and S) to the controller, and the data output (D) to the datapath. We also connect the controller to the datapath control inputs (Dreg_cir, Dreg_ld, Dctr_cir, Dctr_cnt). Normally we don't draw the clock generator block, but we've explicitly shown the clock generator in the figure to make clear that the generator must be exactly 300 MHz.

**Figure 5.19 Controller/datapath (processor) design for the laser-based distance measurer.**
5.2 RTL Design Method

Step 4—Deriving the Controller's FSM

If we created our datapath correctly, deriving an FSM for the controller is straightforward. The FSM will have the same states and transitions as the high-level state machine. We merely define the FSM’s inputs and outputs (all will now be single bits), and replace any data computations in the actions and conditions by the appropriate datapath control signal values. Remember, we created the datapath specifically to carry out those computations, and therefore we should only need to appropriately configure the datapath control signals to implement each particular computation at the right time.

**EXAMPLE 5.5 Laser-based distance measurer—Deriving the controller’s FSM**

We continue the previous example by going to step 4 of the RTL design method.

**Step 4—Derive the controller’s FSM.**

The last step is to design the controller's internals. We can describe the controller's behavior by refining our high-level state machine from Figure 5.15 into an FSM, replacing the “high-level” actions and conditions, like Dctr=0, by actual controller input and output signal assignments and conditions, like Dctr.clr=1, as shown in Figure 5.20. Notice that the FSM does not directly indicate the computations that are happening in the datapath. For example, S4 loads Dreg with Dctr/2, but the FSM itself only shows Dreg’s load signal being activated. Thus, the overall system behavior can be determined from the FSM by looking also at the datapath.

**Inputs:** B, S  **Outputs:** L, Dreg.clr, Dreg.ld, Dctr.clr, Dctr_cnt

![Diagram of FSM]

Figure 5.20 FSM description of the controller for the laser-based distance measurer. The desired action in each state is shown in italics in the bottom row; the corresponding bit signal assignment that achieves that action is shown in bold.

> **HOW DOES IT WORK?—AUTOMOTIVE ADAPTIVE CRUISE CONTROL**

The early 2000s saw the advent of autonomous cruise control systems that not only maintained a particular speed, but also maintained a particular distance from the car in front—thus slowing the automobile down when necessary. Such “adaptive” cruise control thus adapts to changing highway traffic. Adaptive cruise controllers must measure the distance to the car in front. One way to measure that distance uses a laser-based distance measurer, with the laser and sensor placed in the front grille of the car, connected to a circuit and/or microprocessor that computes the distance. The distance is then input to the cruise control system, which determines when to increase or decrease the automobile’s speed.
Recall from Chapter 5 that we typically follow the convention that FSM output signals not explicitly assigned in a state are implicitly assigned 0. Following that convention, the FSM would look as in Figure 5.21. We may still choose to explicitly show the assignment of 0 (e.g., \( L = 0 \) in state \( S3 \)) when that assignment is a key action of a state. The key actions of each state were bolded in Figure 5.20.

**Inputs:** \( B, S \)  **Outputs:** \( L, \text{Dreg}_\text{clr}, \text{Dreg}_d, \text{Dctrl}_\text{clr}, \text{Dctr}_\text{cnt} \)

![FSM Diagram](image)

**Figure 5.21** FSM description of the controller for the laser-based distance measurer, using the convention that FSM outputs not explicitly assigned a value in a state are implicitly assigned 0.

We would complete the design by implementing this FSM, using a 3-bit state register and combinational logic to describe the next state and output logic, as was described in Chapter 3.

### 5.3 RTL Design Examples and Issues

RTL design involves a certain amount of creativity and insight. Thus, a good way to begin to learn RTL design is perhaps through seeing several examples. We now provide additional examples of the RTL design method, through which we also explain some detailed issues.

#### Simple Bus Interface Design Example

**Example 5.6 Simple bus interface**

Processors typically need to transfer data to and from other processors. They typically communicate such data over a bus, to reduce wire congestion problems that might otherwise occur (see Section 4.10). Suppose 16 different processors each has a 32-bit output connected to a single 32-bit bus named \( \text{D} \). Suppose another processor, a master processor, may want to read the output of any of those 16 processors. Let's call those 16 processors **peripherals**, which is a common term for a processor that is auxiliary to a master processor. The master processor outputs a 4-bit address, \( A \), that all the 16 peripherals can read, with each peripheral having its own unique address (0000, or 0001, or 0010, etc.). Because the master processor must always set the address lines to a value, but might not always want to read, the master processor has another output, \( \text{rd} \), that the master processor sets to 1 when reading, and 0 when not reading. So if the master processor wants to read the value in peripheral number five, the master processor would set the address lines \( A \) to 0101, then set \( \text{rd} \) to 1. The master processor would then read the data lines \( \text{D} \) (perhaps storing the read data into a local register), and then return \( \text{rd} \) to 0. Additionally, the value on \( \text{D} \) should not change while the master processor is reading.
A block diagram of the system is shown in Figure 5.22. Such an arrangement is very similar to the arrangement in a desktop computer, where a master processor can read peripheral processor registers—peripherals might include a disk drive, a CD-ROM drive, a keyboard, a modem, etc.

We have just described what is known as a bus protocol. A bus protocol defines a sequence of actions over a set of data, address, and control lines, to carry out a data transfer over those lines from one processor to another.

A bus interface implements a bus protocol for a processor. Let's implement the bus interface for one of the peripheral processors. Figure 5.23 provides a block diagram for a peripheral divided into a main part and a bus interface part. The main part's output 0 is an input to the bus interface. Let's assume the peripheral's own address is another input, called Faddr, to the bus interface. Faddr might come from a DIP switch, or perhaps another register. The bus interface also has inputs and outputs that connect to the bus signals rd, D, and A.

**Step 1** of our RTL design method is to create a high-level state machine. Based on the bus protocol we defined, a peripheral's bus interface part sends data only if the address on input A matches the address on input Faddr AND the processor requests a read by setting rd to 1. While the bus interface waits for an instruction from the master processor to send data, the bus interface should not interfere with what another processor may be writing to the shared data lines, D. Thus, while waiting for a matching address and rd=1, the bus interface should drive 0 with no value (known as high impedance, represented as "Z").

When the bus interface detects a matching address and rd=1, the bus interface should output data from the input 0 (from the main part) to the output 0. However, we must also ensure that 0 does not change while the master processor is reading from the bus interface. We can keep the value on 0 stable by storing Q into a local register Q1. As long as the bus interface is not sending data, the bus interface updates Q1 with the current value of 0. When the bus interface is sending data, the bus interface does not update Q1 and outputs Q1 on 0, causing 0 to not change during a send.

We can see that the bus interface's implementation of the bus protocol can be described by a high-level state machine using two states, shown in Figure 5.24: a state in which the bus interface waits to be able to send data (WaitMyAddress) and a state in which the bus interface sends data (SendData).
Figure 5.25 provides a sample timing diagram of the state machine's behavior. (W stands for state \textit{WaitMyAddress}, SD for \textit{SendData}). As long as the system is in the W state, the system outputs Z on D. When \( r = 1 \) and A=F.addr, the system outputs the contents of Q1 beginning at the next clock cycle's rising edge. The system continues to output Q1 as long as \( r = 1 \). When read returns to 0, the system returns to the \textit{WaitMyAddress} state at the next rising clock edge and hence outputs Z again.

**Step 2** is to create a datapath, as shown on the right in Figure 5.26. The datapath contains a 4-bit equality comparator to compare A and F.addr, a 32-bit register Q1, and a 32-bit wide three-state driver to enable driving of D by nothing or by Q1. A, F.addr, and Q are the datapath's data inputs, and D is the only data output.

**Step 3** is to connect the datapath to a controller, as shown in Figure 5.26. The controller has one external control input, \( r = 1 \), and also gets a control input from the datapath, A.eq.F.addr, indicating whether A equals F.addr. The controller has two control outputs to the datapath, with Q1 ld causing Q1 to be loaded with 0, and D.en controlling the three-state driver.

**Step 4** is to derive the controller's FSM. We simply replace the data operations in the high-level state machine of Figure 5.24 by the appropriate control signals, as shown on the left side of Figure 5.26. We replace A=F.addr by the signal A.eq.F.addr, the actions of D=Z and of D=Q by D.en=0 and D.en=1, and the action of Q1=0 by Q1 ld=1. We would then implement the FSM using a state register (in this case only 1 bit) and combinational logic.

You may have heard of several popular buses, like the PCI (Peripheral Component Interface) bus in a personal computer. Those are the buses that a PC "card" plugs into in a PC, like the card shown in Figure 5.27. You can see on the card the metal pads of the buses—each pad corresponds to one wire of the bus. The bus protocol for PCI is far more complex than the protocol in the above video card.
example. Hundreds of other "standard" bus protocols exist. Designers not needing to interface to other chips often define their own bus protocol for communication.

ALL =’s ARE NOT EQUAL.

Figure 5.24 showed two distinct uses of the "=" symbol. In a state's actions, "=" meant "assign the value of the right side to the left side," e.g., \( D = 01 \). On a transition, "=" meant "the left and right sides are the same," e.g., \( A = F \texttt{addr} \). Be careful not to confuse those two meanings of the "=" symbol. Some languages use different symbols to distinguish the two meanings. For example, the C language uses "=" for "assign" and "==" for "the same," VHDL uses \( =: \) (or "<=") for "assign," and "==" for "the same."

Video Compression—Sum-of-Absolute Differences (SAD) Design Example

EXAMPLE 5.7  Video compression—sum-of-absolute differences

Digitized video is becoming increasingly commonplace, like in the case of the increasingly popular DVD (see Section 6.7 for further information on DVDs). A straightforward digitized video consists of a sequence of digitized pictures, where each picture is known as a frame. However, such digitized video results in huge data files. Each pixel of a frame is stored as several bytes, and let's say a frame contains about a million pixels. Let's assume then that we require about 1 Mbyte per frame, and we play approximately 30 frames per second (a normal rate for a TV), so that's 1 Mbyte/frame × 30 frames/sec = 30 Mbytes/sec. One minute of video would require 60 sec × 30 Mbytes/sec = 1.8 Gbytes, and 60 minutes would require 108 Gbytes. A 2-hour movie would require over 200 Gbytes. That's a lot of data, more than can be downloaded quickly over the Internet, or stored on a DVD, which can only hold between 5 Gbytes and 15 Gbytes. In order to make practical use of digitized video with web pages, digital camcorders, cellular telephones, or even with DVDs, we need to compress those files into much smaller files. A key technique in compressing video is to recognize that successive frames often have much similarity, so instead of sending a sequence of digitized pictures, we can send one digitized picture frame (a "base" frame), followed by data describing just the difference between the base frame and the next frame. We can send just the difference data, before sending another base frame. Such a method results in some loss of quality, but as long as we send base frames frequently enough, the quality may be acceptable.

Of course, if there's a major change from one frame to the next (like for a change of scene, or lots of activity), we can't use the difference method. Video compression devices therefore need to quickly estimate the similarity between two successive digitized frames to determine whether frames can be sent using the difference method. A common way to determine the similarity of two frames is to compute what is known as the sum-of-absolute-differences (SAD). For each pixel in frame 1, we compute the difference between that pixel with the corresponding pixel in frame 2. Each pixel is represented by a number, so difference means the difference in numbers. Suppose we represent a pixel with a byte (real pixels are usually represented by at least three bytes), and we are comparing the pixel at the upper left of frames 1 and 2 in Figure 5.28(a). Say frame 1's upper-left pixel has a value of 255. Frame 2's pixel is clearly the same, so would have a value of 255 also.
A key principle of video compression recognizes that successive frames have much similarity: (a) sending every frame as a distinct digitized picture, (b) instead, sending a base frame and then difference data, from which the original frames can later be reconstructed. If we could do this for 10 frames, (a) would require 1 Mbyte $\times$ 10 = 10 Mbytes, while (b) (compressed) would require only 1 Mbyte $+ 9 \times 0.01$ Mbyte = 1.09 Mbytes, an almost 10x size reduction.

Thus, the difference of these two pixels is 255 - 255 = 0. We might compare the next pixels of both frames in that row, finding the difference to be 0 again. And so on for all the pixels in that row for both frames, as well as the next several rows. However, when we compute the difference of the leftmost pixel of the middle row, where that black circle is located, we see that frame 1's pixel will be black, say with a value of 0. On the other hand, frame 2's corresponding pixel will be white, say with a value of 255. So the difference is 255 - 0 = 255. Likewise, somewhere in the middle of that row, we'll find another difference, this time with frame 1's pixel white (255) and frame 2's pixel black (0)—the difference is again 255 - 0 = 255. Note that we only care about the difference, not which is bigger or smaller, so we are actually looking at the absolute value of the difference between frame 1 and frame 2 pixels. By summing the absolute value of the differences for every pair of pixels, we get a number representing the similarity of the two frames—0 means identical, and bigger numbers means less similar. If the resulting sum is below some threshold (e.g., below 1,000), we might then apply the method of sending the difference data, as in Figure 5.28(b)—we don't explain how to compute the difference data here, as that is beyond the scope of this example. If the sum is above the threshold, then the difference between the blocks is too great, so we might instead send the full digitized frame for frame 2. Thus, video with similarity among frames will achieve a higher compression than video with plenty of differences.

Actually, most video compression methods compute similarity not between two entire frames, but rather between corresponding 16x16 pixel blocks—yet the idea is the same.

Computing the sum of absolute differences is slow in software, so that task may be done using a custom digital circuit, while other tasks may remain in software. For example, you might find an SAD circuit inside a digital camcorder, or inside a cellular telephone that supports video. Let's design such a circuit. A block diagram is shown in Figure 5.29. The circuit's inputs will be a 256-byte memory A, holding the contents of a 16x16 block of pixels of frame 1, and another 256-byte memory B, holding the corresponding block of frame 2. Memories will be discussed in Section 5.6; for now, consider the memory as a register file, and ignore details of the interface to the memories. Another circuit input will be the circuit when to begin computing. An output will present the result after some number of clock cycles.
5.3 RTL Design Examples and Issues

Figure 5.29 Sum-of-absolute-differences (SAD) component: (a) block diagram, and (b) high-level state machine.

**Step 1** of our RTL design method is to create a high-level state machine. We can describe the behavior of the SAD component using the high-level state machine shown in Figure 5.29(a). We declare the inputs, outputs, and local registers \( \text{sum}, i, \) and \( \text{sad\_reg}. \) The \( \text{sum} \) register will hold the running sum of differences; we make this register 32 bits wide. The \( i \) register will be used to index into the current pixel in the block memories; \( i \) will range from 0 to 256, and therefore we’ll make it 9 bits wide. \( \text{sad\_reg} \) will be connected to the output \( \text{sad} \) (it’s good practice to register your data outputs), so will be 32 bits wide, like the \( \text{sad} \) output. The state machine initially waits for the input \( \text{go} \) to become 1. The state machine then initializes registers \( \text{sum} \) and \( i \) to 0. The state machine then enters a loop: if \( i \) is less than 256, the state machine computes the absolute value of the difference of the two blocks’ pixels indexed by \( i \) (the notation \( A[i] \) refers to the data in word \( i \) of memory \( A \)), updates the running sum, increments \( i \), and repeats. Otherwise, the state machine loads \( \text{sad\_reg} \) with the sum, which now represents the final sum, and returns to the first state to wait for the \( \text{go} \) signal to become 1 again.

One point to re-emphasize is that the order of actions in a state does not impact the results, because all those actions occur simultaneously. Thus, for the state inside the loop, arranging the actions as “\( \text{sum} = \text{sum} + \text{abs}(A[i] - B[i]); i = i + 1 \)” or as “\( i = i + 1; \text{sum} = \text{sum} + \text{abs}(A[i] - B[i]) \)” does not impact the results. Either arrangement uses the old value of \( i \).

**Step 2** of our RTL design method is to create a datapath. We see from the high-level state machine that we’ll need a subtractor, an absolute-value component (which we have not designed earlier, but is straightforward to design), an adder, and a comparison of \( i \) to 256. We build the datapath shown in Figure 5.30. The adder will be 32-bits wide, so the 8-bit input coming from the abs component will need to have 0s appended for its high 24 bits.

**Step 3** is to connect the datapath to a controller block, as shown in Figure 5.30. Note that we’ve defined the interface to the \( A \) and \( B \) memories, consisting of a read line, address lines, and data lines. Also note that we haven’t explicitly listed the inputs and outputs of the controller’s FSM, as they can be seen at the periphery of the controller’s block.

**Step 4** is to convert the high-level state machine to an FSM. We show the FSM on the left side of Figure 5.30. For convenience, we’ve shown the original high-level actions (crossed out), and we’ve shown their replacement by the FSM actions.
To complete the design, we would convert the FSM to a controller implementation (a state register and combinational logic), as described in Chapter 3.

**Comparing Software and Custom Circuit Implementations**

In Example 5.7, we said that the output appears after some number of clock cycles. Let's determine exactly how many cycles. After \( g_0 \) becomes 1, our state machine will spend one cycle initializing registers in \( S_1 \), then will spend two cycles in each of the 256 loop iterations (states \( S_2 \) and \( S_3 \)), and finally one more cycle to update the output register in state \( S_4 \), for a total of \( 1 + 2 \times 256 + 1 = 514 \) cycles.

If we executed SAD in software, we would likely need more than two clock cycles per loop iteration. We would need perhaps two cycles to load internal registers, then a cycle for subtract, perhaps two cycles for absolute value, and a cycle for sum, for a total of six cycles per iteration. The custom circuit we built, at two cycles per iteration, is thus about three times faster for computing SAD, assuming equal clock frequencies.

We'll see in Section 6.5 that we could actually build a SAD circuit that is much faster.

**Digital Video—Imagining the Future.**

People seem to have an insatiable appetite for good quality video, and thus much attention is placed on developing fast and/or power-efficient encoders and decoders for digital video devices, like DVD players and recorders, digital video cameras, cell phones supporting digital video, video conferencing units, TVs, TV set-top boxes, etc. It's interesting to think toward the future—assuming video encoding/decoding becomes even more powerful and digital communication speeds increase, we might imagine video displays (with audio) on our walls at home or work that continually display what's happening at another home (perhaps our mom's house) or at a partner office on the other side of the country—like a virtual window to another place. Or we might imagine portable devices that enable us to continually see what someone else wearing a tiny camera—perhaps our child or spouse—sees. Those developments could significantly change our living patterns.
RTL Design Pitfalls and Good Practice

Pitfall: Assuming a Register Is Updated in the State in Which the Register Is Written

Perhaps the most common mistake in creating a high-level state machine is assuming that a register is updated in the state in which the register is written. Such an assumption is incorrect, and can lead to unexpected behavior when the state machine reads the register in the same state, and likewise when the state machine reads the register in a transition condition leaving that state. For example, Figure 5.31(a) shows a simple high-level state machine. Examine the state machine, and then answer the following two questions:

- What will be the value of $\bar{Q}$ after state $A$?
- What will be the final state: $C$ or $D$?

The answers may surprise you. The value of $\bar{Q}$ will not be 99; $\bar{Q}$'s value will actually be unknown. The reason is illustrated by the timing diagram in Figure 5.31(b). State $A$ configures the datapath to load a 99 into $R$ on the next clock edge, and configures the datapath to load the value of register $R$ into register $\bar{Q}$ on the next clock edge. When the next clock edge occurs, both those loads occur simultaneously, $\bar{Q}$ therefore gets whatever value was in $R$ just before the next clock edge, which is unknown.

Furthermore, the final state will not be $D$, but will rather be $C$. The reason is illustrated by the timing diagram in Figure 5.31(b). State $B$ configures the datapath to load 100 into $R$ on the next clock cycle, and configures the controller to load the next state based on the transition condition. $R$ is 99, and therefore the transition condition $R < 100$ is true, meaning the controller will be configured to load state $C$ into the state register, not state $D$. On the next clock edge, $R$ becomes 100, and the next state become $C$.

The key is to always remember that a state's actions configure the datapath and controller such that the next clock edge will load the desired values—but those values don't actually get loaded until that next clock edge. Thus, any expressions in a state's actions or outgoing transition conditions will be using the previous values of registers, not the values being assigned in that state itself. By the same reasoning, all the actions of a state occur simultaneously on the next clock edge, and thus could be written in any order.

Figure 5.31 High-level state machine that behaves different than some people may expect, due to reads of a register in the same state as writes to that register: (a) state machine, (b) timing diagram.
Assuming that the designer actually wants 0 to equal 99 and the final state to be D, then a solution is to add an extra state before reading the value of a register that we assign. Figure 5.32(a) shows a new state machine in which the assignment of \( Q = R \) has been moved to state B, after \( R = 99 \) has taken effect. Furthermore, the state machine has a new state, B2, that simply allows \( R \) to be updated with the new value before we read that value in the transition conditions. The timing diagram in Figure 5.32(b) shows the behavior that the designer expected.

An alternative solution for the transition issue in this case would be to utilize comparison values that take into account that the old value is being used. So instead of comparing \( R \) to 100, the comparisons might instead compare to 99.

Avoiding this pitfall is the reason that we included state S2 in Example 5.7.

**Pitfall: Reading Outputs**

Another common mistake is to create a high-level state machine in which an external output is read in the state machine. Outputs can only be written and cannot be read. For example, Figure 5.33(a) shows an invalid high-level state machine—the read of \( P \) in state \( T \) is not allowed. If you wish to read an output, then create and use a local register. Figure 5.33(b) shows use of a local register \( R \) to avoid reading output \( P \).

**Good Design Practice: Registered Data Outputs**

It's a good idea to always ensure your design has a register at every data output. Doing so prevents those outputs from displaying spurious values. For example, the state machine of Figure 5.33(b) could be implemented as a datapath in which output \( P \) is directly connected to the output of an adder, as shown in Figure 5.34. \( P \) will therefore output spurious values for some time after \( R \) is loaded with \( A \), while the addition is being computed. Furthermore, if \( B \) or \( A \) changes in some other states, \( P \) will also change, but such change is likely not the intended behavior of the state machine—\( P \) should only change when we explicitly assign \( P \) in a state. Another problem is that any processor using the \( P \) output
must take into account the adder when computing the longest register-to-register delays to determine a circuit’s critical path (see Section 5.4).

Therefore, we will follow the design practice of always putting a register directly before the data output, as shown in Figure 5.34(b). Even if we don’t explicitly declare the register as a local register, we always assume it is there in interpreting the high-level state machine, and we always add that register when creating the datapath. Alternatively, we can explicitly declare that register, and then assume that the output is directly connected to that register—this is the approach we took in Example 5.7, in which we declared the register sad_reg. It’s good practice to not read this register; the register’s only purpose is to connect to the output port.

Registering data outputs does have the potential disadvantage of delaying writes to the output port by one cycle, depending on the example.

**Data-Dominated RTL Design**

We can consider RTL designs as falling into one of two categories: control-dominated designs and data-dominated designs.

A *control-dominated design* is a design whose controller contains most of the complexity of the design. When creating such a design, a designer focuses mostly on the design of the controller, meaning design effort goes mostly into defining the state behavior of the system. Once the designer has defined that state behavior, he/she can derive the datapath straightforwardly from that state behavior. A control-dominated design typically responds to external inputs in a precise amount of time, and typically has a simple datapath.

A *data-dominated design* is a design whose datapath contains most of the complexity of the design. When creating such a design, a designer focuses mostly on the design of the datapath, meaning design effort goes mostly into instantiating and interconnecting datapath components. Once the designer has defined the datapath, he/she can define the controller’s state behavior straightforwardly. A data-dominated design typically has a lot of parallelism in its datapath, and the datapath may be large. For a data-dominated design, designers often skip the first step of our RTL design method of Table 5.1.

The laser-based distance measurer example in the previous section was an example of control-dominated design, since the complexity of the design was really in the controller, not the datapath.

The terms “control-dominated” and “data-dominated” are merely descriptive, and can’t be used to strictly categorize designs. Some designs will exhibit properties of both types of designs. It’s like the terms “introvert” and “extrovert” for describing people—while the terms are useful, people can’t be strictly categorized as either introverts or extroverts, since many people are somewhere in between, or exhibit features of both.
categories. The example of the simple bus interface was an example of a design that has a similar amount of control and data design. The video compression SAD circuit, at least the way we designed it, was also a mix of control and data.

RTL design is very much a creative process. Two designers may come up with very different designs for the same system, following perhaps different design methods, with those designs differing in terms of performance, size, and other metrics.

**FIR Filter Design Example**

As our previous examples were either control-dominated or a mix of control and data, we now provide an example of a data-dominated design.

**EXAMPLE 5.8** FIR filter

A digital filter takes a stream of digital inputs and generates a stream of digital outputs with some feature of the input stream removed or modified. Figure 5.35 shows a block diagram of a popular digital filter known as an FIR filter. x and y are N-bits wide each, such as 12 bits each. As a filtering example, consider the following stream of digital temperature values on x coming from a car engine temperature sensor sampled every second: 180, 180, 181, 240, 180, 181. That 240 is probably not an accurate measurement, as a car engine's temperature cannot jump 60 degrees in one second. A digital filter would remove such "noise" from the input stream, generating perhaps an output stream on y like: 180, 180, 181, 181, 180, 181.

An FIR filter (usually pronounced by saying the letters "F" "I" "R"), short for "Finite Impulse Response" filter, is a popular general digital filter design that can be used for a wide variety of filtering goals. Figure 5.35 shows a block diagram of an FIR filter. The basic idea of an FIR filter is simple: the present output is obtained by multiplying the present input value by a constant, and adding that result to the previous input value times a constant, and adding that result to the next earlier input value times a constant, and so on. In a sense, adding to previous values in this manner results in a weighted average. We describe digital filtering and FIR filters in more detail in Section 5.11. For the purpose of this example, we merely need to know that an FIR filter can be described by the following equation:

\[
x(t) = c1 \times x(t) + c2 \times x(t-1) + c3 \times x(t-2)
\]

An FIR filter with three terms, as in the above equation, is known as a 3-tap FIR filter. Real FIR filters typically have many tens of taps—we use only three taps for the purpose of illustration. A filter designer using an FIR filter achieves a particular filtering goal simply by choosing the FIR filter's constants.

We wish to design a circuit to implement an FIR filter. Because the FIR filter equation is just data transformation and no control, let's skip Step 1 of the RTL design method and go straight to Step 2—designing the datapath. We'll need a register for each tap to hold \(x(t)\), \(x(t-1)\), and \(x(t-2)\). On each clock cycle, we'll want to move \(x(t-1)\) to \(x(t-2)\), to move \(x(t)\) to \(x(t-1)\), and to load \(x(t)\) with the present input. We thus start the datapath with three registers, connected as shown in Figure 5.36.
Notice how the data moves to the right on each clock cycle, so that register \( x(t) \) holds the current input sample, \( x(t-1) \) holds the previous input sample, and \( x(t-2) \) holds the sample before the previous one. For the example, we'll assume data is 12 bits wide.

![3-tap FIR filter](image)

Figure 5.36 Beginning to build the datapath for the FIR filter—inserting and connecting the \( x(t) \), \( x(t-1) \), and \( x(t-2) \) registers.

Now we need another register for each tap to hold the constant value \( c0 \), \( c1 \), or \( c2 \)—we'll worry later about how those registers will be loaded. We'll also need a multiplier for each tap, to multiply the tap's \( x \) value by the constant \( c \) value. The datapath with the constant registers and multipliers is shown in Figure 5.37.

![3-tap FIR filter](image)

Figure 5.37 Extending the datapath for the FIR filter—inserting and connecting the \( c0 \), \( c1 \), and \( c2 \) registers, along with the multipliers, for each tap. For simplicity, clock connections are not shown, and all data lines are assumed to be 12 bits wide.

The output \( Y \) is the sum of each tap's product. We can thus insert adders to compute the sum, and we can connect that sum to the output \( Y \), as shown in Figure 5.38.

We have completed the heart of the FIR filter datapath design. We now need to provide a method for a user to load values into the constant registers \( c0 \), \( c1 \), and \( c2 \). Let's create another input \( C \) to the filter, a load line \( C1 \), and a 2-bit address \( Ca1 \) and \( Ca0 \), that the filter user can use to load a particular constant register. \( Ca1Ca0=00 \) indicates that register \( c0 \) should be loaded, \( 01 \) indicates that \( c1 \) should be loaded, and \( 10 \) indicates that \( c2 \) should be loaded. Loading of the value on input \( C \) into the appropriate register occurs on a clock edge only when \( C1=1 \). We can straightforwardly design the circuit for such loading using a decoder, as shown in Figure 5.39. The address lines \( Ca1 \) and \( Ca0 \) feed into a 2x4 decoder, thus enabling the appropriate register (note that address \( 11 \) is unused). The load input \( C1 \) is connected to the decoder's enable input. Note that we've also added a register at the \( Y \) output, which is generally good design practice, since such a register ensures the output doesn't fluctuate as intermediate products and sums are computed, and reduces the likelihood of the user accidentally extending the critical path by connecting \( Y \) through a lot of combinational logic before loading \( Y \) into a register.
Figure 5.38 Computing the output $Y$ in the FIR filter as the sum of the tap products (all data lines are assumed to be 12 bits wide).

Figure 5.39 Finalizing the FIR filter datapath with circuitry for loading the constant registers. We've also added a register on the $Y$ output, which is good design practice. The critical path—the longest register-to-register delay—is shown as a dotted line.

Our RTL design method involves two steps after designing the datapath to complete the controller. However, this particular design does not require a controller, nor even a simple one! This example is indeed an extreme example of a data-dominated design.

Comparing Software and Custom Circuit Implementations
It is interesting to compare the performance of the hardware implementation of a 3-tap FIR filter with a software implementation. The critical path goes from the $x[t]$ and $c$ registers, through one multiplier, and through two adders, before reaching the $Y$ register $Y_{reg}$. 
5.4 Determining Clock Frequency

RTL design produces a processor, consisting of a datapath and a controller. Inside the datapath and controller are registers, and registers require a clock signal. A clock signal must have a particular frequency. The frequency will determine how fast the system will execute its specified task. Obviously, a lower frequency will result in slower execution, while a higher frequency will result in a faster execution. Conversely stated, a larger period is slower, while a smaller period is faster.
Designers of digital circuits often (but not always) want their systems to execute as fast as possible. However, a designer cannot choose an arbitrarily high clock frequency (meaning an arbitrarily small period). Consider, for example, the simple circuit in Figure 5.40, in which registers \( a \) and \( b \) feed through an adder into register \( c \). The adder has a delay of 2 ns, meaning that when the adder's inputs change, the adder's outputs will not be stable until after 2 ns—before 2 ns, the adder's outputs will have spurious values (see Section 4.3 for a description of spurious values appearing at an adder's outputs). If the designer chooses a clock period of 10 ns, the circuit should work fine. Shortening the period to 5 ns will speed the execution. But shortening the period to 1 ns will result in incorrect circuit behavior. One clock cycle might load new values into registers \( a \) and \( b \). The next clock cycle will load register \( c \) 1 ns later (as well as \( a \) and \( b \)), but the output of the adder won't be stable until 2 ns have passed. The value loaded into register \( c \) will thus be some spurious value that has no useful meaning, and will not be the sum of \( a \) and \( b \).

Thus, a designer must be careful not to set the clock frequency too high. To determine the highest possible frequency, a designer must analyze the entire circuit, and find the longest path delay from any register to any other register, or from any circuit input to any register. The longest register-to-register or input-to-register delay in a circuit is known as the circuit's critical path. Designers then choose a clock whose period is longer than the circuit's critical path.

Figure 5.41 illustrates a circuit with at least four possible paths from any register to any other register:

- One path starts at register \( a \), goes through the adder, and ends at register \( c \). This path's delay is 2 ns.
- Another path starts at register \( a \), goes through the adder, through the multiplier, and ends at register \( d \). This path's delay is 2 ns + 5 ns = 7 ns.
- Another path starts at register \( b \), goes through the adder, through the multiplier, and ends at register \( d \). This path's delay is also 2 ns + 5 ns = 7 ns.
- The last path starts at register \( b \), goes through the multiplier, and ends at register \( d \). This path's delay is 5 ns.

The longest path is thus 7 ns (there are actually two such paths). Thus, the clock period must be at least 7 ns.
The above analysis assumes that the only delay between registers is caused by logic delays. In reality, wires also have a delay. In the 1980s and 1990s, the delay of logic dominated over the delay of wires—wire delays were often negligible. But in modern chip technologies, the delay of wires may equal or even exceed the delay of logic, and thus wire delays cannot be ignored. Wire delays add to a path’s length just as logic delays do. Figure 5.42 illustrates a path length calculation with wire delays included.

Furthermore, the above analysis does not consider setup times for the registers. Recall from Section 3.5 that flip-flop inputs (and hence register inputs) must be stable for a specified amount of time before a clock edge. The setup time adds to the path length.

Even considering wire delays and setup times, designers typically choose a clock period that is still longer than the critical path by an amount depending on how conservative the designer wants to be with respect to ensuring the circuit works under a variety of operating conditions. Certain conditions can change the delay of circuit components, conditions like very high temperature, very low temperature, vibration, age, etc. Generally, the longer the period beyond the critical path, the more conservative the design. For example, we might determine that the critical path is 7 ns, but we might choose a clock period of 10 ns, or even 15 ns, the latter being quite conservative.

If low power is a design goal, then a designer might choose an even longer period, such as 100 ns, to reduce circuit power. Why reducing the clock frequency reduces power will be discussed in Section 6.6.

When analyzing a processor (controller and datapath) to find the critical path, a designer must be aware that register-to-register paths exist not just within the datapath (Figure 5.43(a)), but also within the controller (Figure 5.43(b)), between the controller and datapath (Figure 5.43(c)), and even between the processor and external components.

The number of possible paths in a circuit can be quite large. Consider a circuit with $N$ registers that has paths from every register to every other register. Then there are $N^2-N$, or $N^2$ possible register-to-register paths. For example, if $N$ is 3 and the three registers are named A, B, and C.

> **CONSERVATIVE CHIP MAKERS, AND PC OVERCLOCKING.**

Chip makers usually publish their chips’ maximum clocking frequency somewhat lower than the real maximum—perhaps 10%, 20%, or even 30% lower. Such conservatism reduces the chances that the chip will fail in unanticipated situations, such as extremes of hot or cold weather, or slight variations in the chip manufacturing process. Many personal computer enthusiasts have taken advantage of such conservatism by “overclocking” their PCs, meaning to set the clock frequency higher than a chip’s published maximum, by changing the PC’s BIOS (basic input/output system) settings. Numerous websites post statistics on the successes and failures of people trying to overclock nearly every PC processor—it seems the norm is about 10%-40% higher than the published maximum. Now, I don’t recommend overclocking (for one, you may damage the microprocessor due to overheating), but it’s interesting to see the common presence of conservative design.
5.5 BEHAVIORAL-LEVEL DESIGN: C TO GATES (OPTIONAL)

As transistors per chip continue to increase and hence designers build more complex digital systems that use those additional transistors, digital system behavior becomes increasingly difficult to understand. Frequently, a designer building a new digital system finds it useful to first describe the desired system behavior using a programming language, like C, C++, or Java, in order to first get the desired behavior correct. (Alternatively, the designer may use the high-level programming constructs in a hardware description language, like VHDL or Verilog, to first get the desired behavior correct.) Then, the designer converts that programming language description to an RTL design, by following the RTL design method that usually starts with a high-level state machine RTL description. Converting a system's programming language description to an RTL description is known as behavioral-level design. We'll introduce behavioral-level design using an example.

EXAMPLE 5.9 Sum-of-absolute-differences in C for video compression

Recall Example 5.7, in which we created a sum-of-absolute-differences component. In that example, we started with a high-level state machine—but that state machine wasn't very easy to understand. We can more easily describe the computation of the sum of absolute differences using C code, as shown in Figure 5.44.
int SAD (byte A [256], byte B [256]) // not quite C syntax
{
  uint sum; short uint i;
  sum = 0;
  i = 0;
  while (i < 256) {
    sum = sum + abs (A[i] - B[i]);
    i = i + 1;
  }
  return (sum);
}

Figure 5.44 C program description of a sum of absolute differences computation—the C program may be easier to develop and easier to understand than a state machine.

That code is much easier to understand for most people than the high-level state machine in Figure 5.29. Thus, for some designs, C code (or something similar) is the most natural starting point.

To begin the RTL design method, we could convert this code to a high-level state machine, like that in Figure 5.29, and then proceed to complete the RTL design method and hence design the circuit.

It is instructive to define a structured method for converting C code to a high-level state machine. Defining such a method makes clear to us that C code can be automatically compiled to either software on a programmable processor, or to a custom digital circuit. We point out that most designers that start with C code and then continue with RTL design do not necessarily follow a particular method in performing such conversion. However, automated tools do follow a method having some similarities to the one we now describe. We also point out that the conversion method will sometimes result in “extra” states that you might notice could be combined with other states—these extra states would be combined by a later optimization step, though we’ll combine some of them as we follow the method.

We consider three types of statements in C code—assignment statements, while loops, and condition statements (if-then-else)—and provide high level state machine templates for each such statement.

An assignment statement in C translates simply into a state in a state machine, with the state’s actions carrying out the assignment, as shown in Figure 5.45.

An if-then statement in C translates into a state that checks the condition of the if statement, and branches to the states for the then part if the condition is true, otherwise branching past those states to an end state, as shown in Figure 5.46.

We can translate an if-then-else statement in C into a similar state machine with a state that checks the condition of the if statement, but
this time branching to states for the else part if the if condition is false, as shown in Figure 5.47.

The else part commonly contains another if statement as C programmers may have multiple else if parts in a region of code.

Finally, a while loop statement in C translates into states similar to an if-then statement, except that after executing the while’s statements, if the while condition is true, the state machine branches back to the condition check, rather than to the end state, as shown in Figure 5.48. Only when the condition is false can we reach the end state.

Given these simple templates, we can convert a wide variety of C programs to high-level state machines, from which we already know how to create circuit designs following our RTL design method.

**EXAMPLE 5.10 Converting an if-then-else statement to a state machine**

We are given the C-like code shown in Figure 5.49(a), which computes the maximum of two data inputs X and Y. We can translate that code to a state machine by first translating the if-then-else statement to states using the method of Figure 5.47, as shown in Figure 5.49(b). We then translate the then statements to states, and then the else statements, yielding the final state machine in Figure 5.49(c).

```
inputs: uint X, Y
outputs: uint Max
if (X > Y)
    Max = X;
else
    Max = Y;
```

![Diagram of state machines](image)

**Figure 5.49** Behavioral-level design starting from C code: (a) C code for computing the max of two numbers, (b) translating the if-then-else statement to a high-level state machine, (c) translating the then and else statements to states. From the state machine in (c), we could use our RTL design method to complete the design. Note: max can be implemented more efficiently; we use max here to provide an easy-to-understand example.
EXAMPLE 5.11 SAD C code to high-level state machine conversion

We wish to convert the C program description of the sum-of-absolute differences example of Example 5.9 to a high-level state machine. The code is shown in Figure 5.50(a), written as an infinite loop rather than a procedure call, and using an input "go" to indicate when the system should compute the SAD. The "while (1)" statement, after some optimization, translates just to a transition from the last state back to the first state, so we'll hold off on adding that transition until we have formed the rest of the state machine. We begin with the statement "while (!go)", which based on the template approach translates to the states shown in Figure 5.50(b). Since the loop has no statements

```
uint sum; short int i;

while (1) {
    while (!go);
    sum = 0;
    i = 0;
    while (i < 256) {
        sum = sum + abs(A[i] - B[i]);
        i = i + 1;
    }
    sad = sum;
}
```

Figure 5.50 Behavioral-level design of the sum-of-absolute difference code: (a) original C code, written as an infinite loop, (b) translating the statement "while (!go)" to a state machine, (c) simplified states for "while (!go)", and states for the assignment statements that follow, (d) merging the two assignment states into one, (e) inserting the template for the next while loop, (f) inserting the states for that while loop, merging two assignment statements into one, (g) the final high-level state machine, with the "while (1)" included by transitioning from the last state back to the first state, and with obviously unnecessary states removed.
in the loop body, we can simplify the loop's states as shown in Figure 5.50(c). Figure 5.50(c) also shows the states for the next two statements, which are assignment statements. Since those two assignments could be done simultaneously, we merge the two states into one, as shown in Figure 5.50(d). We then translate the next while loop, using the while loop template, to the states shown in Figure 5.50(e). We fill in the states for the while loop's statements in Figure 5.50(f), merging the two assignment statement states into one state since the assignments can be done simultaneously. Figure 5.50(f) also shows the state for the last statement of the C code, which assigns \texttt{sad=sum}. Finally, we eliminate obviously unnecessary empty states, and add a transition from the last state to the first state to account for the entire code being enclosed in a "while (1)" loop.

Notice the similarity between our final high-level state machine in Figure 5.50(g) and the high-level state machine we designed from scratch in Figure 5.29.

We will need to map the C data types to bits at some point. For example, the C code declares \texttt{i} to be a short unsigned integer, which means 16 bits. So we could declare \texttt{i} to be 16 bits in the high-level state machine. Or, knowing the range of \texttt{i} to be 0 to 256, we could instead define \texttt{i} to be 9 bits (C doesn't have a 9-bit wide data type).

We could then proceed to design a controller and datapath from this state machine, as was done in Figure 5.30. Thus, we can translate C code to gates, using a straightforward automatable method.

Through the previous examples, you have seen how C code can be converted to a custom digital circuit using methods that are fully automatable.

General C code can contain additional types of statements, some of which can be easily translated to states. For example, a for loop can be translated to states by first transforming the for loop into a while loop. A switch statement can be translated by first translating the switch statement to if-then-else statements.

Some C constructs pose problems for converting to a circuit, though. For example, pointers and recursion are not easy to translate. Thus, tools that automate behavioral design from C code typically impose restrictions on the allowable C code that can be handled by the tool. Such restrictions are known as subsetting the language.

While we have emphasized C code in this section, obviously any similar language, such as C++, Java, VHDL, Verilog, etc., can be converted to custom digital circuits—with appropriate language subsetting.

5.6 MEMORY COMPONENTS

Register-transfer level design involves instantiating and connecting datapath components to form datapaths, controlled by controllers. RTL design often utilizes some additional components outside the datapath and controller.

One such component is a memory. An \textit{MxN memory} is a memory component able to store \( M \) data items of \( N \) bits each. Each data item in a memory is known as a \textit{word}. Figure 5.51 depicts the storage available in an \( MxN \) memory.

We can generally categorize memories into two groups: RAM memory, which can be written to and read from, and ROM memory, which can only be read from. However, as we shall see, the distinction between the two categories is blurring due to new technologies.

In the early day if you had to store, you magnetic tape to spin the tape write onto the location. If the 1 and you wanted to s
Random Access Memory (RAM)

A RAM is logically the same as a register file (see Section 4.10)—both components are memories whose words (each of which can be thought of as a register) can be individually read and written using address inputs. The differences between a RAM and a register file are:

- The size of $M$—We typically refer to smaller memories (from 4 to 512 or perhaps even 1024 words or so) as register files, and larger memories as RAMs.
- The bit storage implementation—For large numbers of words, a compact implementation becomes increasingly important. Thus, a RAM typically uses a very compact implementation for bit storage, rather than using a flip-flop.
- The memory’s physical shape—For large numbers of words, the physical shape of the memory's implementation becomes important. A tall rectangular shape will have some short wires and some long wires, whereas a square shape will have all medium length wires. A RAM therefore typically has a square shape, to reduce the memory's critical path. Reads are performed by first reading out an entire row of words, and then selecting the appropriate word (column) out of that row.

There's no clear-cut border between what defines a register file and what defines a RAM. Smaller memories (typically) tend to be called register files, and larger memories tend to be called RAMs. But you'll often see the terms used quite interchangeably.

A typical RAM is single-ported. Some RAMs are dual-ported. Adding more ports to RAMs is much less common than to register files, because a RAM's larger size makes the delay and size overhead of extra ports much more costly. Nevertheless, conceptually, a RAM can have an arbitrary number of read ports and write ports, just like a register file.

Figure 5.52 shows a block diagram for a 1024x32 single-port RAM ($M = 1024$, $N = 32$). `data` is a 32-bit wide set of data lines that can serve either as input lines during writes or as output lines during reads. `addr` is a 10-bit input serving as the address lines during reads or writes. `rw` is a 1-bit control input that indicates whether the present operation should be a read or a write (e.g., `rw = 0` means read, `rw = 1` means write). `en` is a 1-bit control input that enables the RAM for reading or writing—if we don't want to read nor write during a particular clock cycle, we set `en` to 0 to prevent a read or write (regardless of the value of `rw`).

 WHY IS IT CALLED "RANDOM ACCESS" MEMORY?

In the early days of digital design, RAMs did not exist. If you had information you wanted your digital circuit to store, you stored it on a magnetic drum, or a magnetic tape. Tape drives (and drum drives too) had to spin the tape to get the head, which could read or write onto the tape, above the desired memory location. If the head was currently above location 900, and you wanted to write to location 999, the tape would have to spin past 901, 902, ..., 998, until location 999 was under the head. In other words, the tape was accessed sequentially. When RAM was first released, its most appealing feature was that any "random" address could be accessed in the same amount of time as any other address—regardless of the previously read address. That's because there is no "head" used to access a RAM, and no spinning of tapes or drums. Thus, the term "random access" memory was used, and has stuck to this day.
Figure 5.53 shows the logical internal structure of an MxN RAM. By “logical” structure, we mean that we can think of the structure being implemented in that way, although a real physical implementation may possess a different actual structure. (As an analogy, a logical structure of a telephone includes a microphone and a speaker connected to a phone line, although real physical telephones vary tremendously in their implementations, including handheld devices, headsets, wireless connections, built-in answering machines, etc.) The main part of the RAM structure is the grid of bit storage blocks, also known as cells. A collection of N cells forms a word, and there are M words. The address inputs feed into a decoder, each output of which enables all the cells in one word corresponding to the present address values. The enable input en can disable the decoder and prevent any word from being enabled. The read/write control input rw also connects to every cell to control whether the cell will be written with wdata, or read out to rdata. The data lines are connected through one word’s cell to the next word’s cell, so each cell must be designed to only output its contents when enabled and thus output nothing when disabled, to avoid interfering with another cell’s output.

![Diagram of RAM structure](image)

Figure 5.53 Logical internal structure of a RAM.

Notice that the RAM in Figure 5.53 has the same inputs and outputs as the RAM block diagram in Figure 5.52, except that the RAM in Figure 5.53 has separate write and read data lines whereas Figure 5.52 has a single set of data lines (a single port). Figure 5.54 shows how the separate lines might be combined inside a RAM having just a single set of data lines.

![Diagram of RAM data input/output](image)

Figure 5.54 RAM data input/output for a single port.

**Bit Storage in a RAM**

Compared to a register file, the key feature of RAM is its compactness. Recall from Chapter 3 that we implemented a bit storage block using a D flip-flop. Because RAMs store large numbers of bits, RAMs utilize a bit storage block that is more compact than a flip-flop. We thus discuss briefly the internal design of the bit storage blocks inside two.
popular types of RAM—static RAM and dynamic RAM. However, be forewarned that the internal design of those blocks involves electronics issues beyond the scope of this book, and instead is within the scope of textbooks on VLSI or advanced digital design. Fortunately, a RAM component hides the complexity of its internal electronics by using a memory controller, and thus a digital designer’s interaction with a RAM remains as discussed in the previous section.

**Static RAM**

Static RAM uses a bit storage block involving two inverters connected in a loop, as shown in Figure 5.55. A bit d will go through the bottom inverter to become d’, then back through the top inverter to become d again—thus, the bit is stored in the inverter loop. Notice that this bit storage block has an extra line, data’, passing through it, compared with the “logical” RAM structure in Figure 5.53.

To write a bit into this inverter loop, we set the data line to the value of the desired bit, and data’ to the complement. So to store a 1, the memory controller sets data=1 and data’=0, as shown in Figure 5.56. (To store a 0, the controller would have set data=0 and data’=1.) The controller then sets enable=1, which turns on both shown transistors. The data and data’ values thus appear in the inverter loop as shown (overwriting whatever value was there before). Fully understanding why this circuit works involves electrical details beyond the scope of this discussion.

Reading the stored bit can be done by first setting the data and data’ lines both to 1 (an act known as precharging), and then by setting enable to 1. One of the enabled transistors will have a 0 at one end, causing the precharged 1 on the data or data’ to drop to a voltage slightly less than a regular logic 1. Both the data and data’ lines connect to a special circuit called a sense amplifier that detects whether the voltage on data is slightly higher than data’, meaning logic 1 is stored, or whether the voltage on data’ is slightly higher than on data, meaning logic 0 is stored. Again, details of the electronics are beyond the scope of this discussion.

Notice that the bit storage block of Figure 5.57 utilizes six transistors—two inside each of the two inverters, and two transistors outside the inverters. Six transistors are fewer than needed inside a D flip-flop. A tradeoff is that special circuitry must be used to read a bit stored in this bit storage block, whereas a D flip-flop outputs regular logic values directly. Such special circuitry slows the access time of the stored bits.
RAM based on a six-transistor bit storage block, or similar such block, is known as a static RAM, or SRAM. A static RAM maintains the stored bit as long as power is supplied to the transistors. Except, of course, when the block is being written, the stored bit does not change—it is static (not changing).

**Dynamic RAM**

An alternative popular bit storage block used in RAM has only a single transistor per block. Such a block utilizes a (relatively large) capacitor at the output of the transistor, as shown in Figure 5.57(a).

Writing can occur when enable is 1: data = 1 will charge the top plate of the capacitor to 1, while data = 0 will make it 0. When enable is returned to 0, a 1 on the top plate will begin to discharge across to the bottom plate of the capacitor on to ground (Why? Because that’s what a capacitor does.) However, the capacitor is intentionally designed to be relatively large, so that the discharge takes a long time, during which time the bit is effectively stored in the capacitor. Figure 5.58(b) provides a timing diagram illustrating the charge and discharge of the capacitor.

Reading can be done by first setting data to a voltage midway between 0 and 1, and then setting enable to 1. The value stored in the capacitor will alter the voltage on the data line, and that altered voltage can be sensed by special circuits connected to the data line that amplify the sensed value to either a logic 1 or a logic 0.

It turns out that reading the charge stored in the capacitor discharges the capacitor. Thus, the RAM must immediately write the read bit back to the bit storage block after reading the block. The RAM must contain a memory controller that automatically performs such a write back.

Because a bit stored in the capacitor gradually discharges to ground, the RAM must refresh every bit storage block before the bits completely discharge and hence the stored bit is lost. To refresh a bit storage block, the RAM must read the block and then write the read bit back to the block. Such refreshing may be done every few microseconds. The RAM must include a built-in memory controller that automatically performs these refreshes.

Note that the RAM may be busy refreshing itself at a time that we wish to read the RAM. Furthermore, every read must be followed by an automatic write. Thus, RAM based on one-transistor plus capacitor technology may be slower to access.

---

**DRAM chips first appeared in the early 1970s, and could hold only a few thousand bits. Modern DRAMs can hold many billions of bits.**
Because the stored bit changes (discharges) even when power is supplied and we are not writing the bit storage block, RAM based on the one transistor plus capacitor bit storage block is known as dynamic RAM, or DRAM.

Compared to SRAM, DRAM is even more compact, requiring only one transistor per bit storage block rather than six transistors. The tradeoff is that DRAM requires refreshing, which ultimately slows the access time. Another tradeoff, not alluded to above, is that creating the relatively large capacitor in a DRAM requires a special chip fabrication process, and thus incorporating DRAM with regular logic can be costly. In the 1990s, incorporating DRAM with regular logic on the same chip was nearly unheard of. Technology advancements, however, have led to DRAM and logic appearing on the same chip in more and more cases.

Figure 5.59 graphically depicts the compactness advantages of SRAM over register files, and DRAM over SRAM, for storing the same number of bits.

Using a RAM

Figure 5.60 shows timing diagrams describing how to write and read the RAM of Figure 5.52. The timing diagram in Figure 5.60 shows how to write a 9 and a 13 into locations 500 and 999 during clock edges 1 and 2, respectively. The next cycle shows how to read location 9 of the RAM, by setting \( \text{addr}=9 \), \( \text{data}=Z \), and \( \text{rw}=0 \) (meaning read). Shortly after \( \text{rw} \) becomes 0, \( \text{data} \) becomes 500 (the value we had previously stored in location 9). Notice that we had to disable our writing of \( \text{data} \) first (by setting it to \( Z \)), so as not to interfere with the data being read from the RAM. Also notice that this RAM’s read functionality is asynchronous.

Figure 5.59 Depiction of compactness benefits of SRAM and DRAM (not to scale).

The delay between our setting the \( \text{rw} \) line to read and the read data stabilizing at the \( \text{data} \) output is known as the RAM’s access time or read time.

We now provide an example of using a RAM in an RTL design.
EXAMPLE 5.12  Digital sound recorder using a RAM

Let's design a system that can record sound, and can play back that recorded sound. Such a recorder is found in various toys, in telephone answering machines, in cell phone outgoing announcements, and numerous other devices. We'll need an analog-to-digital converter to digitize the sound, a RAM to store the digitized sound, a digital-to-analog converter to output the digitized sound, and a processor to control both converters and the RAM. Figure 5.61 shows a block diagram of the system.

![Block Diagram of Digital Sound Recorder System]

Figure 5.61  Utilizing a RAM in a digital sound recorder system.

To store digitized sound, the processor block can implement the high-level state machine segment shown in Figure 5.62. The machine first initializes its internal address counter \( a \) to 0 in state \( S \). Next, in state \( T \), the machine loads a value into the analog-to-digital converter to cause a new analog sample to be digitized, and sets the three-state buffer to pass that digitized value to the RAM’s data lines. That state also sets the RAM address to the counter \( a \)'s value, and sets the control lines to enable writing. The machine then transitions to state \( U \), whose transitions check the value of \( a \) against 4095. That state also increments \( a \). (Remember that the transitions from \( U \) will use the old value, not the incremented value, of \( a \). Thus, the transitions compare with 4095, not 4096.) The machine returns to state \( P \) and hence continues writing samples in sequential memory addresses as long as the memory is not yet filled (\( a < 4095 \)). Notice that the comparison is with 4095, not 4096. This is because the action in state X of \( a = a + 1 \) does not occur until the next clock edge, so the comparison of \( a < 4095 \) on state X's outgoing transition uses the old value of \( a \), not the incremented value (See Section 5.3 discussion of common pitfalls.)

![State Machine for Storing Digitized Sound in RAM]

Figure 5.62  State machine for storing digitized sound in RAM.

To playback the stored digitized sound, the processor block can implement the high-level state machine segment shown in Figure 5.63. After initializing the counter \( a \) in state \( V \), the machine enters state \( W \). State \( W \)
disables the three-state buffer, to avoid interfering with the RAM’s output data that will appear during RAM reads. That state also sets the RAM address lines, and sets the RAM control lines to enable reading. Read data will thus appear on data lines. The next state X loads a value into the digital-to-analog converter, to convert the data just read from RAM to the analog signal. That state also increments the counter a. The machine returns to state W to continue reading, until the entire memory has been read.

**Read-Only Memory (ROM)**

A Read-Only Memory (ROM) is a memory that can be read from, but not written to. Because of being read only, the bit-storage mechanism in a ROM can be made to have several advantages over a RAM, including:

- **Compactness**—a ROM’s bit storage may be even smaller than a RAM’s.
- **Nonvolatility**—A ROM’s bit storage maintains its contents even after the power supply to the ROM is shut off—when turned back on, the ROM’s contents can be read again. In contrast, a RAM loses its contents when power is shut off. A memory that loses its contents when power is shut off is known as volatile, while a memory that maintains its contents without power is known as nonvolatile.
- **Speed**—A ROM may be faster to read than a RAM, especially than a DRAM.
- **Low-power**—A ROM does not consume power to maintain its contents, in contrast to a RAM. Thus, a ROM consumes less power than a RAM.

Therefore, when the data stored in a memory will not change, we might choose to store that data in a ROM to gain the above advantages.

Figure 5.64 shows a block symbol of a 1024x32 ROM. The logical internal structure of an MN ROM is shown in Figure 5.65. Notice that the internal structure is very similar to the internal structure of a RAM shown in Figure 5.53. Bit storage blocks forming a word are enabled by a decoder output, with the decoder input being the address. However, because a ROM can only be read and cannot be written, there is no need for a write input control to specify read versus write, nor for write inputs to provide data being written. Also, because no synchronous writes occur in a ROM, the ROM does not have a clock input. In fact, not only is a ROM an asynchronous component, but in fact a ROM can be thought of as a combinational component (when we only read from the ROM; we’ll see variations later).

Some readers might at this point be wondering how we write the initial contents of a ROM that we then can only read. After all, if we can’t write the contents of a ROM at all, then the ROM is really of no use to us. Obviously, there must be a way to write the contents of a ROM, but in ROM terminology, the writing of the initial contents of a ROM is known as **ROM programming**. ROM types differ in their bit storage block implementations, which in turn causes differences in the methods used for ROM programming. We now describe several popular bit storage block implementations for ROM.
ROM Types

Mask-programmed ROM

Figure 5.66 illustrates the bit storage cell for a mask-programmed ROM. A mask-programmed ROM has its contents programmed when the chip is manufactured, by directly wiring 1s to cells that should store a 1, and 0s to cells that should store a 0. Recall that a "1" is actually a higher-than-zero voltage coming from one of several power input pins to a chip—thus, wiring a 1 means wiring the power input pin directly to the cell. Likewise, wiring a 0 to a cell means wiring the ground pin directly to the cell. Be aware that Figure 5.66 presents a logical view of a mask-programmed ROM cell—the actual physical design of such cells may be somewhat different—for example, a common design strings several vertical cells together to form a large NOR-like logic gate. We leave details for more advanced textbooks on CMOS circuit design.

Wires are placed onto chips during manufacturing by using a combination of light-sensitive chemicals and light passed through lenses and "masks" that block the light from reaching regions of the chemicals. (See Chapter 7 for further details.) Hence the term "mask" in mask-programmed ROM.

Mask-programmed ROM has the best compactness of any ROM type, but the contents of the ROM must be known during chip manufacturing. This ROM type is best suited for high-volume well-established products in which compactness or very low cost is critical, and in which programming of the ROM will never be done after the ROM's chip is manufactured.
Fuse-Based Programmable ROM—One-Time Programmable (OTP) ROM

Figure 5.67 illustrates the bit storage cell for a fuse-based ROM. A **fuse-based ROM** uses a fuse in each cell. A fuse is an electrical component that initially conducts from one end to the other just like a wire, but whose connection from one end to the other can be destroyed ("blown") by passing a higher-than-normal current through the fuse. A blown fuse does not conduct and is instead an open circuit (no connection). In the figure, the cell on the left has its fuse intact, so when the cell is enabled, a 1 appears on the data line. The cell on the right has its fuse blown, so when the cell is enabled, nothing appears on the data line (special electronics will be necessary to convert that nothing to a logic 0).

A fuse-based ROM is manufactured with all fuses intact, so the initially stored contents are all 1s. A user of this ROM can program the contents by connecting the ROM to a special device, known as a **programmer**, that provides higher than normal currents to only those fuses in cells that should store 0s. Because a user can program the contents of this ROM, the ROM is known as a programmable ROM, or **PROM**.

A blown fuse cannot be changed back to its initial conducting form. Thus, a fuse-based ROM can only be programmed once. Fuse-based ROM are therefore also known as **one-time programmable (OTP)** ROM.

Erasable PROM—EPROM

Figure 5.68 depicts a logical view of an erasable PROM cell. An **erasable PROM** or **EPROM**, cell uses a special type of transistor, having what is known as a floating gate, in each cell. The details of a floating gate transistor are beyond the scope of this section, but briefly—a floating gate transistor has a special gate in which electrons can be "trapped." A transistor with electrons trapped in its gate stays in the nonconducting situation, and thus is programmed to store a 0. Otherwise, the cell is considered to store a 1. Special electronic circuitry converts sensed currents on the data lines as logic 1 or 0.

An EPROM cell initially has no electrons trapped in any floating gate transistors, so the initially stored contents are all 1s. A programmer device applies higher-than-normal voltages to those transistors in cells that should store 0s. That high voltage causes electrons to **tunnel** through a small insulator into the floating gate region. When the voltage is removed, the electrons do not have enough energy to tunnel back, and thus are trapped as shown in the right cell of Figure 5.68.
The electrons can be freed by exposing the electrons to ultraviolet (UV) light of a particular wavelength. The UV light energizes the electrons such that they tunnel back through the small insulator, thus escaping the floating gate region. Exposing an EPROM chip to UV light therefore "erases" all the stored 0s, restoring the chip to having all 1s as contents, after which the EPROM can be programmed again. Hence the term "erasable" PROM. Such a chip can typically be erased and reprogrammed about ten thousand times or more, and can retain its contents without power for ten years or more. Because a chip usually appears inside a black package that doesn't pass light, a chip with an EPROM requires a window in that package through which UV light can pass, as shown in Figure 5.69.

**EEPROM and Flash Memory**

An *electrically erasable PROM*, or *EEPROM*, utilizes the EPROM programming method of using high voltage to trap electrons in a floating gate transistor. However, unlike an EPROM that requires UV light to free the electrons and hence erase the PROM, an EEPROM uses another high voltage to free the electrons. EEPROMs thus avoid the need for placing the chip under UV light.

Because EEPROMs use voltages for erasing, those voltages can be applied to specific cells only. Thus, while EPROMs must be erased in their entirety, EEPROMs can be erased one word at a time. Thus, we can erase and reprogram certain words in an EEPROM without changing the contents of other words.

Some EEPROMs require a special programmer device for programming. However, most modern EEPROMs do not require special voltages to be applied to the pins, and also include internal memory controllers that manage the programming process. Thus, we can reprogram an EEPROM's contents (or part of its contents) without ever removing the chip from the system that the EEPROM serves—such an EEPROM is known as being *in-system programmable*. Most such devices can therefore be read and written in a manner very similar to a RAM.

Figure 5.70 shows a block diagram of an EEPROM. Notice that the data lines are bidirectional, just as was the case for RAM. The EEPROM has a control input `write=0` indicates a read operation (when `en=1`), while `write=1` indicates that the data on the data lines should be programmed into the word at the address specified by the address lines. Programming a word into an EEPROM takes time, though, perhaps several, dozens, hundreds, or even thousands of clock cycles. Therefore, EEPROMs may have a control output `busy` to indicate that programming is not yet complete. While the device is busy, the EEPROM user should not try writing to a different word. Fortunately, most EEPROMs will load
Because this doll's message will never change, we might choose to use a mask-programmed ROM or an OTP ROM. We might utilize OTP ROM during prototyping or during initial sales of the doll, and then produce mask programmed ROM versions during high-volume production of the doll.

**EXAMPLE 5.14 Digital telephone answering machine using a flash memory**

We are to design the outgoing announcement part of a telephone answering machine (e.g., "We're not home right now, leave a message."). That announcement should be stored digitally, should be recordable by the machine owner any number of times, and should be saved even if power is removed from the answering machine. Recording begins immediately after the owner presses a record button, which sets a signal rec to 1. Because we must be able to record the announcement, we cannot use a mask-programmed ROM or OTP ROM. Because removing power should not cause the announcement to be lost, we cannot use a RAM. Thus, we might choose an EEPROM or a flash memory. We'll use a flash memory, as shown in Figure 5.73. Notice that the flash memory has the same interface as a RAM, except that the flash memory has an extra input named erase. erase on this particular flash memory clears the contents of the entire flash. While the flash memory is erasing itself, the flash sets an output busy to 1, during which time we cannot write to the flash memory.

![Diagram](image)

**Figure 5.73 Utilizing a flash memory in a digital answering machine.**

Figure 5.74 shows a high-level state machine segment for recording the announcement. The state machine segment begins when the record button is pressed. State S activates the erase of the flash memory (er = 1), and then state T waits for the erasing to complete (bu = 1). Such erasing should occur in just a few milliseconds, so we shouldn't miss any of the spoken announcement. The state machine then transitions to state U, which copies a digitized sample from the analog-to-digital converter to the flash memory, writing to the current address a. State U also increments a. The next state (V) checks to see if the memory is filled with samples by checking if a < 4096, returning to state U until the memory is filled.

![State machine diagram](image)

**Figure 5.74 State machine for storing digitized sound in a flash memory.**
Notice that, unlike Examples 5.12 and 5.13, this state machine increments a before the state that checks for the last address (state V), so V's transitions use 4096, not 4095. We show this version just for variety. The version in Example 5.12 may be slightly better because that version requires that a, and the comparator, only be 12 bits wide (to represent 0 to 4095) rather than 13 bits wide (to represent 0 to 4096).

This state machine assumes that writes to the flash occur in one clock cycle. Some flash memories require more time for writes, asserting their busy output until the write has completed. For such a flash, we would need to add a state between states U and V, similar to the state T between S and U.

To prevent missing sound samples while waiting, we might want to first save the entire sound sample in a 4096x16 RAM, and then copy the entire RAM contents to the flash.

The Blurring of the Distinction between RAM and ROM

Notice that EEPROM and flash ROM blur the distinction between RAM and ROM. Many modern EEPROM devices are writable just like a RAM, having nearly the same interface, with the only difference being longer write times to an EEPROM than to a RAM. However, the difference between those times is shrinking each year.

Further blurring the distinction are nonvolatile RAM (NVRAM) devices, which are RAM devices that retain their contents even without power. Unlike ROM, NVRAM write times are just as fast as regular RAM—typically one clock cycle. One type of NVRAM simply includes an SRAM with a built-in battery, with the battery able to supply power to the SRAM for perhaps ten years or more. Another type of NVRAM includes both an SRAM and an EEPROM—the NVRAM controller automatically backs up the SRAM's contents into the EEPROM, typically just at the time when power is being removed. Furthermore, extensive research and development into new bit storage technologies are leading to NVRAMs that are even closer to RAM in terms of performance and density while being nonvolatile. One such technology is known as MAGRAM, short for magnetic RAM, which uses magnetism to store charge, having access times similar to DRAM, but without the need for refreshing, and with nonvolatility.

Thus, digital designers have a tremendous variety of memory types available to them, with those types differing in their cost, performance, size, nonvolatility, ease-of-use, write time, duration of data retention, and other factors.

5.7 QUEUES (FIFOs)

Sometimes our data storage needs specifically that we read items in the same order that we wrote them, and that reading removes the item from the list. For example, a busy restaurant may maintain a waiting list of customers—the host writes customer names to the rear of the list, but when a table becomes available, the host reads the next customer's name from the front of the list and removes that name from the list. Thus, the first customer written to the list is the first customer read from the list. A queue is

```
   back
    ↓
[        ]
    ↑
   front
```

write items to back of the queue

read (and remove) items from front of the queue

Figure 5.75 Conceptual view of a queue.
a list that is written at the rear of the list but read from the beginning of the list, with a read also removing the read item from the list, as illustrated in Figure 5.75. The common term for a queue in American English is a "line"—for example, you stand in a line at the grocery store, with people entering the rear of the line, and being served from the front of the line. In British English, the word queue is used directly in everyday language (which sometimes confuses Americans who visit other English-speaking countries). Because the first item written into the list will be the first item read out of the list, a queue is known as being first-in first-out (FIFO). As such, sometimes queues are called FIFO queues, although that term is redundant because a queue is by definition first-in first-out. The term FIFO itself is often used to refer to a queue. The term buffer is also sometimes used. A write to a queue is sometimes called a push or enqueue, and a read is sometimes called pop or dequeue.

We can implement a queue using a memory—either a register file or a RAM, depending on the queue size needed. When using a memory, the front and rear will move to different memory locations as the queue is written and read, as illustrated in Figure 5.76. The figure shows an initially empty eight-word queue with front and rear both set to memory address 0. The first action on the queue is a write of item A, which goes to the rear (address 0), and the rear increments to address 1. The next action is a write of item B, which goes to the rear (address 1), and the rear increments to 2. The next action is a read, which comes from the front (address 0) and thus reads out item A, and the front increments to 1.

Subsequent reads and writes continue likewise, except that when the rear or front reaches 7, its next value should be 0, not 8. In other words, the memory can be thought of as a circle, as shown in Figure 5.77.

Two conditions of a queue are of interest:

- **Empty**: there are no items in the queue. This condition can be detected as \( \text{front} = \text{rear} \), as seen in the topmost queue of Figure 5.76.
- **Full**: there is no more room to add items to the queue, meaning there are \( N \) items in a queue of size \( N \). This comes about when the rear wraps around and catches back up to the front, meaning \( \text{front} = \text{rear} \).

![Figure 5.76 Writing and reading a queue implemented in a memory causes the front \( f \) and rear \( r \) to move.](image)

![Figure 5.77 Implementing a queue in a memory treats the memory as a circle.](image)
Unfortunately, notice that the conditions detecting the queue being empty and the queue being full are the same—the front address equals the rear address. One way to tell the two conditions apart is to keep track of whether a write or a read preceded the front and rear addresses becoming equal.

In many uses of a queue, the circuit writing the queue operates independently from the circuit reading the queue. Thus, a queue implemented with a memory may use a two-port memory having separate read and write ports.

We can implement an 8-word queue using an 8-word two-port register file and additional components, as depicted in Figure 5.78. A 3-bit up-counter maintains the front address, while another 3-bit up-counter maintains the rear address. Notice that these counters will naturally wrap around from 7 to 0, or from 0 to 7, as desired when treating the memory as a circle. An equality comparator detects whether the front counter equals the rear counter. A controller writes the write data to the register file and increments the rear counter during a write, reads the read data from the register file and increments the front counter during a read, and determines whether the queue is full or empty based on the equality comparison as well as whether the previous operation was a write or a read. We omit further description of the queue's controller, but it can be built by starting with an FSM.

A user of the queue should never read an empty queue or write a full queue—depending on the controller design, such an action might just be ignored or might put the queue into a misleading internal state (e.g., the front and rear addresses may cross over).

Most queues come with one or more additional control outputs that indicate whether the queue is half full, or perhaps 80% full.

Queues are commonplace in digital systems. Some examples include:

- A computer keyboard writes the pressed keys into a queue and meanwhile requests that the computer read the queued keys. You might at some time have typed faster than your computer was reading the keys, in which case your additional keystrokes were ignored—and you may have even heard beeps each time you pressed additional keys, indicating the queue was full.
- A digital video camera may write recently captured video frames into a queue, and concurrently may read those frames from the queue, compress them, and store them on tape or another medium.
- A computer printer may store print jobs in a queue while those jobs are waiting to be printed.
EXAMPLE 5.15 Using a queue

Show the internal state of a 8-word queue, and popped data values, after each of the following sequences of pushes and pops, assuming an initially empty queue:

1. Push 9, 5, 8, 5, 7, 2, and 3.
2. Pop
3. Push 6
4. Push 3
5. Push 4
6. Pop

Figure 5.79 shows the queue's internal states. After the first sequence of seven pushes (step 1), we see that the rear address points to address 7. The pop (step 2) reads from the front address of 0, returning data of 9. The front address increments to 1. Note that although the queue might still contain the value of 9 in address 0, that 9 is no longer accessible during proper queue operation, and thus is essentially gone. The push of 6 (step 3) increments the rear address, which wraps around from 7 to 0.

The push of 3 (step 4) increments the rear address to 1, which now equals the front address, meaning the queue is now full. If a pop were to occur now, it would read the value 5. But instead, a push of 4 occurs (step 5)—this push should not have been performed, because the queue is full. Thus, this push puts the queue into an erroneous state, and we cannot predict the behavior of any subsequent pushes or pops.

A queue could of course come with some error-tolerance behavior built in, perhaps ignoring pushes when full, or perhaps returning some particular value (like 0) if popped when empty.
5.8 HIERARCHY—A KEY DESIGN CONCEPT

Managing Complexity

Throughout this book, we have been utilizing a powerful design concept known as hierarchy. Hierarchy in general is defined as an organization with a few “things” at the top, and each thing possibly consisting of several other things. Perhaps the most widely known type hierarchy involves a country. At the top is a country, which consists of many states or provinces, each of which in turn consists of many cities. A hierarchy involving a country, provinces, and cities is shown in Figure 5.80. That figure shows all three levels of the hierarchy—country, provinces, and cities.

Figure 5.81 shows the same country, but this time showing only the top two levels of hierarchy—countries and provinces. Indeed, most maps of a country only show these top two levels (possibly showing key cities in each province/state, but certainly not all the cities)—showing all the cities also makes the map far too detailed and cluttered. A map of a province/state, however, might then show all the cities within that state. Thus, we see that hierarchy plays an important role in understanding countries (or at least their maps).

Likewise, hierarchy plays an important role in digital design. In Chapter 2, we introduced the most fundamental component in digital systems—the transistor. In Chapters 2 and 3, we introduced several basic components composed from transistors, like AND gates, OR gates, and NOT gates, and then some slightly more complex components composed from gates: multiplexers, decoders, flip-flops, etc. In Chapter 4, we composed the basic components into a higher level of components, datapath components, like registers, adders, ALUs, multipliers, etc. In Chapter 5, we introduced components composed of datapath components, including controllers, datapaths, processors (made up of controllers and datapaths), memories, and queues.

Use of hierarchy enables us to manage complex designs. Imagine trying to comprehend the design of Figure 5.30 at the level of logic gates—that design likely consists of several thousand logic gates. Humans can't comprehend several thousand things at once. But they can comprehend a few dozen things. As the number of things grows beyond a few dozen, we therefore group those things into a new thing, to manage the complexity. However, hierarchy alone is not sufficient—we must also associate an understandable meaning to the higher-level things we create, a task known as abstraction.
Abstraction

Hierarchy may not only involve grouping things into a larger thing, but may also involve associating a higher-level behavior to that larger thing. So when we grouped transistors to form an AND gate, we didn't just say that an AND gate was a group of transistors—rather, we associated a specific behavior with the AND gate, with that behavior describing the behavior of the group of transistors in an easily understandable way. Likewise, when we grouped logic gates into a 32-bit adder, we didn't just say that an adder was a group of logic gates—rather, we associated a specific understandable behavior with the adder: A 32-bit adder adds two 32-bit numbers.

Associating higher-level behavior with a component to hide the complex inner details of that component is a process known as abstraction.

Abstraction frees a designer from having to remember, or even understand, the low-level details of a component. Knowing that an adder adds two numbers, a designer can use an adder in a design. The designer need not worry about whether the adder internally is implemented using a carry-ripple design, or using some complicated design that is perhaps faster but larger. Instead, the designer just needs to know the delay of the adder and the size of the adder, which are further abstractions.

Composing a Larger Component from Smaller Versions of the Same Component

A common design task is to create a larger version of a component from smaller versions of the same component. For example, suppose you have 3-input AND gates available to you, but you need a 9-input AND gate. You could compose several 3-input AND gates to form a 9-input AND gate, as shown in Figure 5.82. You could compose OR gates into a larger OR gate, and XOR gates into larger XOR gates, similarly. Some compositions might require more than two levels—composing an 8-bit AND from 2-input ANDs requires four 2-input ANDs in the first level, two 2-input ANDs in the second level, and a 2-input AND in the third level. Some compositions might end up with extra inputs that must be hardwired to 0 or 1—an 8-input AND built from 3-input ANDs would look similar to Figure 5.82, but with the bottom input of the bottom AND gate hardwired to 1. After trying a few examples of composing AND gates into larger ones, you can come up with a general rule to compose any size AND gates into a larger gate: fill the first level with (the largest available) AND gates until the sum of their inputs equal the desired number of inputs, then fill the second level similarly (feeding first level outputs to the second level gates), until a level has just one gate (that's the last level). Connect any unused AND gate inputs to 1. Composing NAND, NOR, or XNOR gates into larger gates of the same kind would require a few more gates to maintain the same behavior.

Multiplexers can also be composed together to form a larger multiplexer. For example, suppose you had 4x1 and 2x1 muxes available, but you needed an 8x1 mux. You could compose the smaller muxes into an 8x1 mux as shown in Figure 5.83. Notice that
s2 selects among group i0-i3 and i4-i7, while s1 and s0 select one input from the group. You can check that select line values pass the appropriate input through, for example, s2s1s0 = 000 passes i0, s2s1s0 = 100 passes i4, and s2s1s0 = 111 passes i7.

One particularly commonly occurring composition problem is that of creating a larger memory from smaller ones. The larger memory may have wider words, may have more words, or both.

For example, suppose you have available a large number of 1024x8 ROMs, but you want a 1024x32 ROM. Composing the smaller ROMs into the larger one is straightforward, and shown in Figure 5.84. We'll need four 1024x8 ROMs to obtain 32 bits per word. We connect the 10 address inputs to all four ROMs. Likewise, we connect the enable input to all four ROMs. We group the four 8-bit outputs into our desired 32-bit output. Thus, each ROM stores one byte of the 32-bit word. Reading a location, say location 99, results in four simultaneous reads, of the byte at location 99 of each ROM.

As another example using ROM, suppose you again have 1024x8 ROMs available, but this time you need a 2048x8 ROM. So you have an extra address line because you have twice as many words to address. Figure 5.85 shows how to use two 1024x8 ROMs to create a 2048x8 ROM. The top ROM represents the top half of the memory (1024 words), and the bottom ROM the bottom half (1024 words). We use the 11th address line (a10) to enable either the top ROM or the bottom ROM—the other 10 bits represent the offset into the ROM. That 11th bit feeds into a 1x2 decoder, whose outputs feed into the ROM enables. Figure 5.86 uses a table of addresses to show how the 11th bit selects among the two smaller ROMs.
Actually, we could use any bit to select between the top ROM and bottom ROM. Designers commonly use the lowest-order bit (a0) to select. The top ROM would thus represent all even-addressed words, the bottom ROM all odd-addressed words.

Finally, since only one ROM will be active at any time, we can tie together the output data lines to form our 8-bit output, as shown in Figure 5.85.

As a final example using ROM, suppose you needed a 4096x32 ROM, but had only 1024x8 ROMs available. In this situation, we need both to create more words, and wider words. The approach is straightforward: first, create a 4096x8 ROM by using 4 ROMs one on top of the other and by feeding the top two address lines to a 2x4 decoder to select the appropriate ROM, and then second, widen the ROM by adding 3 more ROMs to each row.

Most of the datapath components we introduced in Chapter 4 can be composed into larger versions of the same type of component.

5.9 RTL Design Optimizations and Tradeoffs (See Section 6.5)

Previous sections in this chapter described how to perform register-transfer level design to create processors consisting of a controller and a datapath. This section, which physically appears in the book as Section 6.5, describes how to create processors that are better optimized, or that trade off one feature for another (e.g., size for performance). One use of this book covers such RTL optimizations and tradeoffs immediately after introducing RTL design, meaning now. Another use introduces them later.
5.10 RTL DESIGN USING HARDWARE DESCRIPTION LANGUAGES (SEE SECTION 9.5)

This section, which physically appears in the book as Section 9.5, describes use of HDLs during RTL design. One use of this book describes such HDL use immediately after introducing RTL design (meaning now). Another use describes use of HDLs later.

5.11 PRODUCT PROFILE: CELL PHONE

A cell phone, short for cellular telephone and also known as a mobile phone, is a portable wireless telephone that can be used to make phone calls while moving about a city. Cell phones have made it possible to communicate with distant people nearly anytime and anywhere. Before cell phones, most telephones were tied to physical places like a home or an office. Some cities supported a radio-based mobile telephone system using a powerful central antenna somewhere in the city, perhaps atop a tall building. Because radio frequencies are scarce and thus carefully doled out by governments, such a radio telephone system could only use perhaps tens or a hundred different radio frequencies, and thus could not support large numbers of users. Those few users therefore paid a very high fee for the service, limiting such mobile telephone use to a few wealthy individuals and to key government officials. Those users had to be within a certain radius of the main antenna, measured in tens of miles, to receive service, and that service typically didn’t work in another city.

Cells and Basestations

Cell phone popularity exploded in the 1990s, growing from a few million users to hundreds of millions of users in that decade (even though the first cell phone call was made way back in 1973, by Martin Cooper of Motorola, the inventor of the cell phone), and today it is hard for many people to remember life before cell phones. The basic technical idea behind cell phones divides a city into numerous smaller regions, known as cells (hence the term “cell phone”). Figure 5.87 shows a city divided into three cells. A typical city might actually be divided into dozens, hundreds, or even thousands of cells. Each cell has its own radio antenna and equipment in the center, known as a basestation. Each basestation can use dozens or hundreds of different radio frequencies. Each basestation antenna only needs to transmit radio signals powerful enough to reach the basestation’s cell area. Thus, nonadjacent cells can actually reuse the same frequencies, so the limited number of radio frequencies allowed for mobile phones
can be thus shared by more than one phone at one time. Hence, far more users can be supported, leading to reduced costs per user. Figure 5.87 illustrates that phone1 in cell A can use the same radio frequency as phone2 in cell C, because the radio signals from cell A don’t reach cell C. Supporting more users means greatly reduced cost per user, and more basestations means service in more areas than just major cities.

Figure 5.88(a) shows a typical basestation antenna. The basestation's equipment may be in a small building or commonly in a small box near the base of the antenna. The antenna shown actually supports antennas from two different cellular service providers—one set on top, one set just under, on the same pole. Land for the poles is expensive, which is why providers share, or sometimes find existing tall structures on which to mount the antennas, like buildings, park light posts, and other interesting places (e.g., Figure 5.88(b)). Some providers try to disguise their antennas to make them more soothing to the eye, as in Figure 5.88(c)—the entire tree in the picture is artificial.

All the basestations of a service provider connect to a central switching office of a city. The switching office not only links the cellular phone system to the regular “hardline” phone system, but also assigns phone calls to specific radio frequencies, and handles switching among cells of a phone moving between cells.

**How Cellular Phone Calls Work**

Suppose you are holding phone1 in cell A of Figure 5.87. When you turn on the cell phone, the phone listens for a signal from a basestation on a control frequency, which is a special radio frequency used for communicating commands (rather than voice data) between the basestation and cell phone. If the phone finds no such signal, the phone reports a “No Service” error. If the phone finds the signal from basestation A, the phone then transmits its own identification (ID) number to basestation A. Every cell phone has its own unique ID number. (Actually, there is a nonvolatile memory card inside each phone that has that ID number—a phone user can potentially switch cards among phones, or have multiple cards for the same phone, switching cards to change phone numbers.) Basestation A communicates this ID number to the central switching office’s computer, and thus the service provider computer database now records that your phone is in cell A. Your phone intermittently sends a control signal to remind the switching office of the phone’s presence.

If somebody then calls your cell phone’s number, the call may come in over the regular phone system, which goes to the switching office. The switching office computer database
indicates that your phone is in cell A. In one type of cell phone technology, the switching office computer assigns a specific radio frequency supported by basestation A to the call. Actually, the computer assigns two frequencies, one for talking, one for listening, so that talking and listening can occur simultaneously on a cell phone—let’s call that frequency pair a channel. The computer then tells your phone to carry out the call over the assigned channel, and your phone rings. Of course, it could happen that there are so many phones already involved with calls in cell A that basestation A has no available frequencies—in that case, the caller may hear a message indicating that user is unavailable.

Placing a call proceeds similarly, but your cell phone initiates the call, ultimately resulting in assigned radio frequencies again (or a “system busy” message if no frequencies are presently available).

Suppose that your phone is presently carrying out a call with basestation A, and that you are moving through cell A toward cell B in Figure 5.87. Basestation A will see your signal weakening, while basestation B will see your signal strengthening, and the two basestations transmit this information to the switching office. At some point, the switching office computer will decide to switch your call from basestation A to basestation B. The computer assigns a new channel for the call in cell B (remember: adjacent cells use different sets of frequencies to avoid interference), and sends your phone a command (through basestation A, of course) to switch to a new channel. Your phone switches to the new channel and thus begins communicating with basestation B. Such switching may occur dozens of times while a car drives through a city during a phone call, and is transparent to the phone user. Sometimes the switching fails, perhaps if the new cell has no available frequencies, resulting in a “dropped” call.

Inside a Cell Phone

Basic Components
A cell phone requires sophisticated digital circuitry to carry out calls. Figure 5.39 shows the insides of a typical basic cell phone. The printed-circuit boards include several chips implementing digital circuits. One of those circuits performs analog-to-digital conversion.

![Figure 5.39 Inside a cell phone: (a) handset, (b) battery and SIM card on left, keypad and display in center, digital circuitry on a printed-circuit board on right, (c) the two sides of the printed-circuit board, showing several digital chip packages mounted on the board.](image-url)
of a voice (or other sound) to a signal stream of 0s and 1s, and another performs digital-to-analog conversion of a received digital stream back to an analog signal. Some of the circuits, typically software on a microprocessor, execute tasks that manage the various features of the phone, such as the menu system, address book, games, etc. Note that any data that you save on your cell phone (e.g., an address book, customized ring tones, game high score information, etc.) will likely be stored on a flash memory, whose nonvolatility ensures the data stays saved in memory even if the battery dies or is removed. Another important task involves responding to commands from the switching office. Another task carried out by the digital circuits is filtering. One type of filtering removes the carrier radio signal from the incoming radio frequency. Another type of filtering removes noise from the digitized audio stream coming from the microphone, before transmitting that stream on the outgoing radio frequency. Let’s examine filtering in more detail.

Filtering, and FIR Filters
Filtering is perhaps the most common task performed in digital signal processing. Digital signal processing operates on a stream of digital data that comes from digitizing an input signal, such as an audio, video, or radio signal. Such streams of data are found in countless electronic devices, such as CD players, cell phones, heart monitors, ultrasound machines, radios, engine controllers, etc. Filtering a data stream is the task of removing particular aspects of the input signal, and outputting a new signal without those aspects.

A common filtering goal is to remove noise from a signal. You’ve certainly heard noise in audio signals—it’s that hissing sound that’s so annoying on your stereo, cell phone, or cordless phone. You’ve also likely adjusted a filter to reduce that noise, when you adjusted the “treble” control of your stereo (though that filter may have been implemented using analog methods rather than digital). Noise can appear in any type of signal, not just audio. Noise might come from an imperfect transmitting device, an imperfect listening device (e.g., a cheap microphone), background noise (e.g., freeway sounds coming into your cell phone), electrical interference from other electric appliances, etc. Noise typically appears in a signal as random jumps from a smooth signal.

Another common filtering goal is to remove a carrier frequency from a signal. A carrier frequency is a signal added to a main signal for the purpose of transmitting that main signal. For example, a radio station might broadcast a radio signal at 102.7 MHz, 102.7 MHz is the carrier frequency. The carrier signal may be a sine wave of a particular frequency (e.g., 102.7 MHz) that is added to the main signal, where the main signal is the music signal itself. A receiving device locks onto the carrier frequency, and then filters out the carrier signal, leaving the main signal.

An FIR filter (usually pronounced by saying the letters “F” “I” “R”), short for “Finite Impulse Response,” is a very general filter design that can be used for a huge variety of filtering goals. The basic idea of an FIR filter is very simple: multiply the present input value by a constant, and add that result to the previous input value times a constant, and so on. A designer using an FIR filter achieves a particular filtering goal simply by choosing the FIR filter’s constants.

Mathematically, an FIR filter can be described as follows:

\[ y(t) = c_0 x(t) + c_1 x(t-1) + c_2 x(t-2) + c_3 x(t-3) + c_4 x(t-4) + \ldots \]

\( t \) is the present time step, \( x \) is the input signal, and \( y \) is the output signal. Each term (e.g., \( c_0 x(t) \)) is called a tap. So the above equation represents a 5-tap FIR filter.
Let's see some examples of the versatility of an FIR filter. Assume we have a 5-tap FIR filter. For starters, to simply pass a signal through the filter unchanged, we set $c_0$ to 1, and we set $c_1=c_2=c_3=c_4=0$. To amplify an input signal, we can set $c_0$ to a number larger than 1, perhaps setting $c_0$ to 2. To create a smoothing filter that outputs the average of the present value and the past four input values, we can simply set all the constants to equivalent values that add to 1, namely, $c_1=c_2=c_3=c_4=c_5=0.2$. The results of such a filter applied to a noisy input signal are shown in Figure 5.90. To smooth and amplify, we can set all constants $c$ to equivalent values that add to something greater than 1, for example, $c_1=c_2=c_3=c_4=c_5=1$, resulting in 5x amplification. To create a smoothing filter that only includes the previous two rather than four input values, we simply set $c_3$ and $c_4$ to 0. We see that we can build all the above different filters just by changing the constant values of an FIR filter. The FIR filter is indeed quite versatile.

![Graph of a 5-tap FIR filter with $c_0=c_1=c_2=c_3=c_4=0.2$ applied to a noisy signal. The original signal is a sine wave. The noisy signal has random jumps. The FIR output (fir_avg_out) is much smoother than the noisy signal, approaching the original signal. Notice that the FIR output is slightly shifted to the right, meaning the output is slightly delayed in time (probably a tiny fraction of a second delayed). Such slight shifting is usually not important to a particular application.]

That versatility extends even further. We can actually filter out a carrier frequency using an FIR filter, by setting the coefficients to different values, carefully chosen to filter out a particular frequency. Figure 5.91 shows a main signal, $in_1$, that we want to transmit. We can add that to a carrier signal, $in_2$, to obtain the composite signal, $in_{total}$. The signal $in_{total}$ is the signal that would be the signal that is transmitted by a radio station, for example, with $in_1$ being the signal of the music, and $in_2$ the carrier frequency.

Now say a stereo receiver receives that composite signal, and needs to filter out the carrier signal, so the music signal can be sent to the stereo speakers. To determine how to filter out the carrier signal, look carefully at the samples (the small filled squares in Figure 5.91) of that carrier signal. Notice that the sampling rate is such that if we take any sample, and add it to a sample from three time steps back, we get 0. That's because for a positive point, three samples earlier was a positive point of the same magnitude. For a negative point, three samples earlier was a negative point of the same magnitude. And for a zero point, three samples earlier was also a zero point. Likewise, adding a carrier signal
Figure 5.91 Adding a main signal, \( in1 \), to a carrier signal, \( in2 \), resulting in a composite signal \( in\text{\_total} \).

A sample to a sample three steps later also adds to zero. So to filter out the carrier signal, we can add each sample to a sample three time steps back. Or we can add each sample to \( 1/2 \) times a sample three steps back, plus \( 1/2 \) times a sample three steps ahead. We can achieve this using a 7-tap FIR filter with the following seven coefficients: 0.5, 0, 0, 1, 0, 0, 0.5. Since that sums to 2, we can scale the coefficients to add to 1, as follows: 0.25, 0, 0, 0.5, 0, 0, 0.25. Applying such a 7-tap FIR filter to the composite signal results in the FIR output shown in Figure 5.92. The main signal is restored. We should point out that we chose the main signal such that this example would come out very nicely—other signals might not be restored so perfectly. But the example demonstrates the basic idea.

Figure 5.92 Filtering out the carrier signal using a 7-tap FIR filter with constants 0.25, 0, 0, 0.5, 0, 0, 0.25. The slight delay in the output signal typically poses no problem.

While 5-tap and 7-tap FIR filters can certainly be found in practice, many FIR filters may contain tens or hundreds of taps. FIR filters can certainly be implemented using software (and often are), but many applications require that the hundreds of multiplications and additions for every sample be executed faster than is possible in software, leading to custom digital circuit implementations. Example 5.8 illustrated the design of a circuit for an FIR filter.

Many types of filters exist other than FIR filters. Digital signal filtering is part of a larger field known as digital signal processing, or DSP. DSP has a rich mathematical foundation and is a field of study in itself. Advanced filtering methods are what make cell phone conversations as clear as they are today.
5.12 CHAPTER SUMMARY

In this chapter, we described (Section 5.1) that much digital design today involves designing processor-level components, and that design is done at what is called the register-transfer level (RTL). We introduced (Section 5.2) a four-step RTL design method for converting RTL behavior to a processor implementation, with that implementation consisting of a datapath controlled by a controller. The RTL design method made use of the datapath components defined in Chapter 4, and the controller design process defined in Chapter 3, which built on the combinational design process of Chapter 2. We provided several examples of RTL design (Section 5.3), while pointing out several pitfalls and good design practices, and discussing the characteristics of control- versus data-dominated designs. We discussed (Section 5.4) how to set a circuit’s clock frequency based on the circuit’s critical path. We demonstrated (Section 5.5) how a sequential program, like a C program, could conceptually be converted to gates using some straightforward transformations that transform the C into RTL behavior, which as we know can then be converted to gates using the four-step RTL design method. That demonstration should make it clear that a digital system’s functionality can be implemented as either software on a microprocessor or as a custom digital circuit (or even as both). The differences among software and custom circuit implementations are not related to what each can implement—they can both implement any functionality. The differences are instead related to design metrics like system performance, power consumption, size, cost, design time, and so on. Modern digital designers must therefore be comfortable migrating functionality between software on a microprocessor and custom digital circuits, in order to obtain the best overall implementation with respect to constraints on design metrics. We introduced (Section 5.6) several memory components commonly used in RTL design, including RAM and ROM components. We also introduced (Section 5.7) a queue component that can be useful during RTL design. We took a moment to discuss (Section 5.8) a general technique that we’ve been using throughout the book, hierarchy, which helps a designer to manage complexity.

In Chapters 1 through 5, we have emphasized straightforward design methods for increasingly complex systems, but we have not emphasized how to design those systems well. Improving on our designs will be the focus of the next chapter.

5.13 EXERCISES

Any problems noted with an asterisk (*) represent especially challenging problems.

SECTION 5.2: RTL DESIGN METHOD

5.1 (a) Create a high-level state machine that describes the following system behavior. The system has an 8-bit input A, a single-bit input d, and a 32-bit output S. On every clock cycle, if d = 1, the system should add A to a running sum and output that sum on S. If d = 0, the system should instead subtract. Ignore issues of overflow and underflow. Don’t forget to include an initialization state. Hint: Declare and use an internal register to keep the sum.
(b) Add a 1-bit input rst to the system. When rst = 1, the system should clear its sum back to 0.

5.2 Create a high-level state machine for a simple data encryption/decryption device. If a bit-input b is 1, the device stores the data from a 32-bit input I as what is known as an offset value. If b is 0 and another bit-input c is 1, then the device “encrypts” its input I by adding the stored offset value to I, and outputs this encrypted value over a 32-bit output J. If instead another
bit-input d is 1, the device should “decrypt” the data on 1 by subtracting the offset value before outputting the decrypted value over d. Be sure to explicitly handle all possible combinations of the three input bits.

5.3 Create a high-level state machine for a digital bath-water controller. The system has a 3-bit input ratio indicating the desired ratio of cold water to hot water, and a bit input on indicating that the water should flow. The system has two 4-bit outputs hflow and cflow, controlling the hot water flow rate and the cold water flow rate. The sum of these two rates should always equal 16. Your high-level state machine should determine the output values for hflow and cflow such that the ratio of hot water to cold water is as close as possible to the desired ratio, while the total flow is always 16. Hint: As there are only 8 possible ratios, a reasonable solution may use one state for each ratio.

5.4 Create a high-level state machine that initializes a 10x32 register tile’s contents to all 0s, beginning the initialization when an input rst is 1.

5.5 (a) Create a high-level state machine that adds each register of one 128x8 register tile to the corresponding registers of another 128x8 register tile, storing the results in a third 128x8 register tile. The system should only begin the addition when a bit-input add is 1, and should not perform the addition again until it has finished adding (only adding again if add is 1).

(b) Extend this system to either add or subtract, using an additional bit-input op, where op = 1 means add, and op = 0 means subtract.

5.6 Design a high-level state machine for a 4-bit up-counter with count control input cnt, count clear input clr, and a terminal count output tc. Use the RTL design method of Table 5.1 to convert the high-level state machine to a controller and a datapath. Use a register and incrementer in the datapath, not a counter itself. Design the controller down to a state register and logic gates.

5.7 Compare the up-counter designed in Exercise 5.6 with the up-counter design shown in Figure 4.48.

5.8 Create a datapath for the high-level state machine in Figure 5.93.

5.9 Starting with the soda machine dispenser design described in Example 5.1, create a block diagram and high-level state machine for a soda machine dispenser that has a choice of two soda types, and that also provides change to the consumer. A coin detector provides the circuit with a 1-bit input c that becomes 1 for one clock cycle when a coin is detected, and an 8-bit input a indicating the coin’s value in cents. Two 8-bit inputs s1 and s2 indicate the cost of the two soda choices. The user’s soda selection is controlled by two buttons b1 and b2 that when pushed will output 1 for one clock cycle. If the user has inserted enough change for their selection, the circuit should set either output bit d1 or d2 to 1 for one clock cycle, causing the selected soda to be dispensed. The soda dispenser circuit should also set an output bit e to 1 for one clock cycle if change is required, and should output the amount of change required using an 8-bit output cca. Use the RTL design method shown in Table 5.1 to convert the high-level state machine to a controller and a datapath. Design the datapath to structure, but design the controller to the point of an FSM only, as was done in Figure 5.26.
(a) Use the RTL design method of Table 5.1 to convert the high-level state machine in Figure 5.94 to a controller and a datapath. Design the datapath to structure, but design the controller to an FSM only, as was done in Figure 5.26.

(b) *Design the controller's FSM down to structure.

5.11 Create an FSM that interfaces with the datapath in Figure 5.95. The FSM should use the datapath to compute the average value of the 16 32-bit elements of any Array A. Array A is stored in memory, with the first element at address 25, the second at address 26, and so on. Assume that putting a new value onto the address lines M_addr causes the memory to almost immediately output the read data on the M_data lines. Ignore the possibility of overflow.

```
Figure 5.95 Datapath for computing the average of 16 elements of an array.
```

5.12 Using the RTL design method shown in Table 5.1, create an RTL design of a reaction timer circuit that measures the time elapsed between the illumination of a light and the pressing of a button by a user. The reaction timer has three inputs, a clock input clk, a reset input rst, and a button input B, and three outputs, a light enable output len, a 10-bit reaction time output rtim, and a slow output indicating the user was not fast enough. The reaction timer works as follows: On reset, the reaction timer waits for 10 seconds before illuminating the light by setting len to 1. The reaction timer then measures the length of time in milliseconds before the user presses the button B, outputting the time as a 12-bit binary number on rtim. If the user did not press the button within 2 seconds (2000 milliseconds), the reaction timer will set the output slow to 1 and output 2000 on rtim. Assume your clock input has a frequency of 1 kHz. Hint: This is a control-dominated RTL design problem. Design the datapath to structure, but design the controller to an FSM only, as was done in Figure 5.26.
5.13 Use the RTL design method shown in Table 5.1 to convert the high-level state machine in Figure 5.74 to a controller and a datapath. Design the datapath to structure, but design the controller to the point of an FSM only, as was done in Figure 5.26.

SECTION 5.3: RTL DESIGN EXAMPLES AND ISSUES
For the following problems, design the datapath to structure, but design the controller to an FSM only, as done in Figure 5.26.

5.14 Using the RTL design method shown in Table 5.1, create an RTL design that computes the sum of all positive numbers within a 512-word register tile \( A \) consisting of 32-bit numbers stored in two’s complement form.

5.15 Using the RTL design method shown in Table 5.1, create an RTL design that computes the sum of all positive numbers from a set of 16 separate 32-bit registers storing numbers in two’s complement form. Make the design as fast as possible by performing as many computations concurrently as possible. Hint: This is a data-dominated design.

5.16 Using the RTL design method shown in Table 5.1, create an RTL design that outputs the maximum value found within a register tile \( A \) consisting of 64 32-bit numbers.

5.17 Using the RTL design method shown in Table 5.1, create an RTL design that outputs a warning signal whenever the average temperature over the past four samples exceeds a user-defined value. The circuit has a 32-bit input \( CT \) indicating the current temperature reading, a 32-bit input \( WT \) indicating the user-specified temperature at which the warning should be enabled, and a button input \( clr \) that will disable the warning. When the average temperature exceeds the user-specified warning level, the circuit should assert the output \( W \) to enable the warning. The warning output should remain high until the \( clr \) button is pressed. Hint: You can use a right shift to implement the divide within your datapath.

5.18 Using the RTL design method shown in Table 5.1, create an RTL design for a digital filter that outputs the average of the current 32-bit input and the previous 32-bit sample. Hint: You can use a right shift to implement the divide within your datapath.

SECTION 5.4: DETERMINING CLOCK FREQUENCY
5.19 Assuming an inverter has a delay of 1 ns, all other gates have a delay of 2 ns, and wires have a delay of 1 ns, determine the critical path for the full-adder circuit shown in Figure 4.31.

5.20 Assuming an inverter has a delay of 1 ns, all other gates have a delay of 2 ns, and wires have a delay of 1 ns, determine the critical path for the 3x8 decoder of Figure 2.50.

5.21 Assuming an inverter has a delay of 1 ns, all other gates have a delay of 2 ns, and wires have a delay of 1 ns, determine the critical path for a 4x1 multiplexer.

5.22 Assuming an inverter has a delay of 1 ns, and all other gates have a delay of 2 ns, determine the critical path for an 8-bit carry-ripple adder:
   (a) assuming wires have no delay,
   (b) assuming wires have a delay of 1 ns.

5.23 (a) Convert the laser-based distance measurer’s FSM, shown in Figure 5.21, to a state register and logic.
   (b) Assuming all gates have a delay of 2 ns and the 16-bit up-counter has a delay of 5 ns, and wires have no delay, determine the critical path for the laser-based distance measurer.
   (c) Calculate the corresponding maximum clock frequency for the circuit.

SECTION 5.5: BEHAVIORAL-LEVEL DESIGN: C TO GATES (OPTIONAL)
5.24 Convert the following C-like code, which calculates the greatest common divisor (GCD) of the two 8-bit numbers \( a \) and \( b \), into a high-level state machine.
Inputs: byte a, byte b, bit go
Outputs: byte gcd, bit done

GCD:
while(!l) {
    while(!go):
        done = 0;
        while ( a != b ) {
            if( a > b ) {
                a = a - b;
            } else {
                b = b - a;
            }
        }
        gcd = a;
        done = 1;
    }

5.25 Use the RTL design method shown in Table 5.1 to convert the high-level state machine you created in Exercise 5.24 to a controller and a datapath. Design the datapath to structure, but design the controller to the point of an FSM only.

5.26 Convert the following C-like code, which calculates the maximum difference between any two numbers within an array A consisting of 256 8-bit values, into a high-level state machine.

Inputs: byte a[256], bit go
Outputs: byte max_diff, bit done

MAX_DIFF:
while(!l) {
    while(!go):
        done = 0;
        i = 0;
        max = 0;
        min = 255; // largest 8-bit value
        while( i < 256 ) {
            if( a[i] < min ) {
                min = a[i];
            } else if( a[i] > max ) {
                max = a[i];
            }
            i = i + 1;
        }
        max_diff = max - min;
        done = 1;
    }
5.27 Use the RTL design method shown in Table 5.1 to convert the high-level state machine you created in Exercise 5.26 to a controller and a datapath. Design the datapath to structure, but design the controller to the point of an FSM only.

5.28 Convert the following C-like code, which calculates the number of times the value \( b \) is found within an array \( A \) consisting of 256 8-bit values, into a high-level state machine.

```c
Inputs: byte a[256], byte b, bit done
Outputs: byte freq, bit done

// FREQUENCY:
while(1) {
    while(!done);
    done = 0;
    i = 0;
    freq = 0;
    while( i < 256 ) {
      if( a[i] == b ) {
        freq = freq + 1;
      }
    }
    done = 1;
}
```

5.29 Use the RTL design method shown in Table 5.1 to convert the high-level state machine you created in Exercise 5.28 to a controller and a datapath. Design the datapath to structure, but design the controller to the point of an FSM only.

5.30 Develop a template for converting a do{}while loop of the following form to a high-level state machine.

```c
do {} {
  // do while statements
  while (cond);
}
```

5.31 Convert the while( a != b ) loop within the C code description of Exercise 24 into a do{}while loop as described in Exercise 5.30. Using the do{}while loop template you created in Exercise 5.30, convert the revised C code into a high-level state machine. Use the RTL design method shown in Table 5.1 to convert the high-level state machine you created in the previous problem to a controller and a datapath. Design the datapath to structure, but design the controller to the point of an FSM only.

5.32 Develop a template for converting a for() loop of the following form to a high-level state machine.

```c
for(i=start; i<end; i++) {
  // for statements
}
```

5.33 Convert the while( a != b ) loop within the C code description of Exercise 5.24 to a for() loop as described in Exercise 5.32. Using the for() loop template you created in
Exercise 5.32, convert the revised C code into a high-level state machine. Use the RTL design method shown in Table 5.1 to convert the high-level state machine you created in the previous problem to a controller and a datapath. Design the datapath to structure, but design the controller to the point of an FSM only.

5.34 *Convert the while (i < 256) loop within the C code description of Exercise 5.26 to a for() loop as described in Exercise 5.32. Using the for() loop template you created in Exercise 5.32, convert the revised C-like code into a high-level state machine. Use the RTL design method shown in Table 5.1 to convert the high-level state machine you created in the previous problem to a controller and a datapath. Design the datapath to structure, but design the controller to the point of an FSM only.*

5.35 Compare the time required to execute the following computation using a custom circuit versus using software. Assume a gate has a delay of 1 ns. Assume a microprocessor executes one instruction every 5 ns. Assume that n=10 and m=5. Estimates are acceptable; you need not design the circuit, or determine exactly how many software instructions will execute.

```c
for (i = 0; i < n, i++) { 
  s = 0;
  for (j = 0; j < m, j++) { 
    s = s + c[i]*x[i + j];
  } 
  y[i] = s;
}
```

SECTION 5.6: MEMORY COMPONENTS

5.36 Calculate the approximate number of DRAM bit storage cells that will fit on an IC with a capacity of 10 million transistors.

5.37 Calculate the approximate number of SRAM bit storage cells that will fit on an IC with a capacity of 10 million transistors.

5.38 Summarize the main differences between DRAM and SRAM memories.

5.39 Draw a complete logic internal structure for a 4x2 DRAM (four words, 2 bits each), clearly labeling all internal components and connections.

5.40 Draw a complete logic internal structure for a 4x2 SRAM (four words, 2 bits each), clearly labeling all internal components and connections.

5.41 *Design an SRAM memory cell with a reset input that when enabled will set the memory cell’s contents to 0.*

SECTION : READ-ONLY MEMORY (ROM)

5.42 Summarize the main differences between EPROM and EEPROM memories.

5.43 Summarize the main differences between EEPROM and flash memories.

SECTION 5.7: QUEUES (FIFOS)

5.44 For an 8-word queue, show the queue’s internal state and provide the value of popped data for the following sequences of pushes and pops: (1) push A, B, C, D, E, (2) pop, (3) pop, (4) push U, V, W, X, Y, (5) pop, (6) push Z, (7) pop, (8) pop, (9) pop.

5.45 Create an FSM describing the queue controller of Figure 5.78. Pay careful attention to correctly setting the full and empty outputs.
Create an FSM describing the queue controller of Figure 5.78, but with error-preventing behavior that ignores any pushes when the queue is full, and ignores pops of an empty queue (outputting 0).

SECTION 5.8: HIERARCHY—A KEY DESIGN CONCEPT

Compose a 20-input AND gate from 2-input AND gates.

Compose a 16x1 mux from 2x1 muxes.

Compose a 4x16 decoder with enable from 2x4 decoders with enable.

Compose a 1024x8 RAM using only 512x8 RAMs.

Compose a 512x8 RAM using only 512x4 RAMs.

Compose a 1024x8 ROM using only 512x4 ROMs.

Compose a 2048x8 ROM using only 256x8 ROMs.

Compose a 1024x16 RAM using only 512x8 RAMs.

Compose a 1024x12 RAM using 512x8 and 512x4 RAMs.

Compose a 640x12 RAM using only 128x4 RAMs.

Write a program that takes a parameter N, and automatically builds an N-input AND gate from 2-input AND gates. Your program merely need indicate how many 2-input AND gates exist in each level, from which we could easily determine the connections.

task was to help sit built. For over 10 generations of net and switch ATM packets. The chips are complex and take together almost per telecommunication devices becomes un

When asked what Kai says, “More z matters more than (requires the ability to picture), to design the silicon in the it interplay more and particular area required as well. Also, each example, verification ability, while bring a analyzer—good...
Chi-Kai started college as an engineering major, and became a Computer Science major due to his developing interests in algorithms and in networks. After graduating, he worked for a Silicon Valley startup company that made chips for computer networking. His first task was to help simulate those chips before the chips were built. For over 10 years now, he has worked on multiple generations of networking devices that buffer, schedule, and switch ATM network cells and Internet Protocol packets. “The chips required to implement networking devices are complex components that must all work together almost perfectly to provide the building blocks of telecommunication and data networks. Each generation of devices becomes successively more complex.”

When asked what skills are necessary for his job, Chi-Kai says “More and more, breadth of one’s skill set matters more than depth. Being an effective chip engineer requires the ability to understand chip architecture (the big picture), to design logic, to verify logic, and to bring up the silicon in the lab. All these parts of the design cycle interplay more and more. To be truly effective at one particular area requires hands-on knowledge of the others as well. Also, each requires very different skills. For example, verification requires good software programming ability, while bring up requires knowing how to use a logic analyzer—good hardware skills.”

High-end chips, like those involved in networking, are quite costly, and require careful design. “The software design process and the chip design process are fundamentally different. Software can afford to have bugs because patches can be applied. Silicon is a different story. The one time expenses to spin a chip are on the order of $500,000. If there is a show-stopping bug, you may need to spend another $500,000. This constraint means the verification approach taken is quite different—effectively: there can be no bugs.” At the same time, these chips must be designed quickly to beat competitors to the market, making the job “extremely challenging and exciting.”

One of the biggest surprises Chi-Kai encountered in his job is the “incredible importance of good communication skills.” Chi-Kai has worked in teams ranging from 10 people to 30 people, and some chips require teams of over 100 people. “Technically outstanding engineers are useless unless they know how to collaborate with others and disseminate their knowledge. Chips are only getting more complex—individual blocks of code in a given chip have the same complexity as an entire chip only a few years ago. To architect, design, and implement logic in hardware requires the ability to convey complexity.” Furthermore, Chi-Kai points out that “just like any social entity, there are politics involved. For example, people are worried about aspirations for promotion, financial gain, and job security. In this greater context, the team still must work together to deliver a chip.” So, contrary to the conceptions many people have of engineers, engineers must have excellent people skills, in addition to strong technical skills. Engineering is a social discipline.