CHAPTER 5

Register-Transfer Level (RTL) Design

5.1 INTRODUCTION

Previous chapters introduced methods to capture behavior and to implement that behavior as a digital circuit. Chapter 2 introduced a method to capture basic combinational behavior using equations or truth tables, and to implement that behavior as a circuit of two or more levels of gates. Chapter 3 introduced a method to capture basic sequential behavior using finite-state machines (FSMs), and to implement that behavior as a circuit having a register and combinational logic, which together are known as a controller. This chapter will focus on capturing even higher-level sequential behavior, using a high-level state machine (HLSM) whose inputs, outputs, state actions, and transition conditions can all involve higher-level data types like binary numbers or integers rather than just the Boolean type used in FSMs. To implement such behavior, this chapter will introduce a method to convert a high-level state machine into a circuit consisting of a controller connected to a datapath, which together are known as a ***processor***. The datapath is composed of datapath components defined in Chapter 4, including registers, adders, comparators, etc., custom connected such that the datapath can perform the specific operations defined by the high-level state machine. The controller sets the datapath's signals in each state such that the state's actions and transition conditions are carried out.

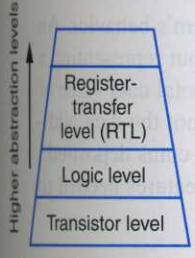


Figure 5.1 Levels of digital design.

The above discussion uses the term "higher-level," which should be explained. Digital designers commonly distinguish among the different levels shown in Figure 5.1. The more complex the building blocks, the higher the level of *abstraction* that the designer deals with. Connecting transistors into circuits to build gates or other components is called ***transistor-level design***. Designing combinational or sequential circuits as in Chapters 2 and 3 involves circuits whose building blocks are primarily logic gates, and is thus called ***logic-level design***. Designing processors involves circuits whose building blocks are registers and other datapath components, and involves transferring data from registers, through other datapath components like adders, and back to registers. Such design is thus called *register-transfer level design* or ***RTL design***—which is the focus of this chapter. In the 1970s and 1980s, most digital design practice occurred at the logic level. Today, most practice is at the register-transfer level. Improving tools continue to move design practice to higher levels. Higher levels deal with fewer and higher-

complexity building blocks, and thus can enable design of higher-complexity circuits with less time and effort.

The term "microprocessor" became popular in the 1980s when programmable processors shrank from occupying many boards or chips down to occupying just a single chip. "Micro" refers to being small.

The name *processor* is best known from its use in the name *microprocessor*. A microprocessor is a *programmable* processor, which is a general predesigned processor created to carry out any desired computation (see Chapter 8). This chapter instead focuses on designing *custom processors*, which are processors each of whose design is specialized to implement one specific computation, like converting a Celsius number to Fahrenheit. As such, custom processors can be extremely small, low-power, and fast compared to programmable processors. Custom processors and programmable processors often coexist in digital systems. For example, a TV set-top box may use a programmable processor to carry out most of the functions related to changing channels, controlling volume, etc., but may use custom processors to very quickly decompress the video data that is streaming into the system and to quickly display that data on the TV screen.

RTL design begins by capturing desired behavior. A formalism for capturing RTL behavior is a high-level state machine.

▶ 5.2 HIGH-LEVEL STATE MACHINES

Some behaviors are too complex to capture using just an equation, truth table, or FSM. Consider capturing the behavior of a custom processor for a soda machine dispenser that dispenses a soda when enough money has been deposited into the machine. A block diagram of the processor system is shown in Figure 5.2. A coin detector provides the 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* indicates the value in cents of the inserted coin, such as 25 cents (`00011001`) or 10 cents (`00001010`). Another 8-bit input *s* indicates the cost of a soda, such as 60 cents (`00111100`), which can be set by the machine owner. When the processor has detected that the total value of deposited coins equals or exceeds the cost of a soda (e.g., 25 + 25 + 10 >= 60), 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, and does not give change). Assume that the value *a* persists until the next coin is deposited, and that many clock cycles (e.g., thousands) occur between successive coins being deposited.

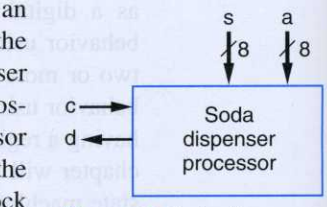


Figure 5.2 Soda dispenser block symbol.

An FSM is not sufficient for capturing the data aspects of this system's behavior. An FSM can only have Boolean (i.e., single-bit) inputs, not an 8-bit data input representing a binary number. An FSM has no convenient way of keeping track of the total data value of coins deposited so far. An FSM can only perform Boolean operations, not the data addition operation (e.g., 25 + 10) required to keep track of the total value of coins deposited.

A ***high-level state machine*** (***HLSM***) extends FSMs with the data features needed to capture more complex behaviors, including:

- *multibit data inputs and outputs* rather than just single bits,
- *local storage*, and
- *arithmetic operations* like add and compare, rather than just Boolean operations.

This chapter will use HLSMs whose local storage is loaded on rising clock edges. Also, each local storage item and multibit input or output is assumed to be unsigned, unless specifically denoted as "signed" (see Section 4.6).

Figure 5.3 shows an HLSM describing the behavior of the soda dispenser processor. The HLSM initially sets output `d` to `0` and sets a local storage item *tot* to 0. The HLSM then waits in state *Wait* to detect a coin being deposited. When detected, the HLSM goes to state *Add*, which adds the coin's value *a* to *tot*, after which the HLSM returns to *Wait*. If *tot*'s value is less than the cost *s* of a soda (*tot* < *s*), the HLSM continues to wait for more coins. Otherwise, the HLSM goes to state *Disp*, which sets `d` to `1` to dispense a soda, after which the HLSM returns to state *Init* to clear *tot* back to 0 and start over again.

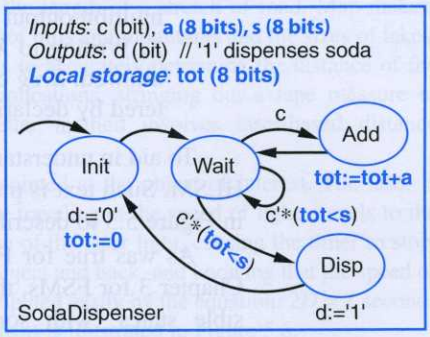


Figure 5.3 Soda dispenser high-level state machine with non-FSM constructs highlighted.

As a reminder, this book usually uses a `Courier` font for names and constants representing a bit, and italics for other names. So "`d`" and "`0`" represent bit values, while "tot" and "0" represent normal data items.

The state machine is *not* an FSM, because of reasons highlighted in the figure. 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 local storage *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 *tot* := 0 (remember that *tot* is 8-bits wide), *tot* < *s*, and *tot* := *tot* + *a* (where the "+" is addition, not OR), whereas an FSM allows only Boolean operations like AND or OR.

This chapter will use the following conventions for HLSMs, 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.

This chapter will also use the following conventions for HLSMs:

- To distinguish between a bit `0` or `1` and a binary number 0 or 1, HLSMs will surround bit values with single quotes as in `'0'` and `'1'`. In Figure 5.3, note that the bit output `d` is assigned the bit `'0'` while the multibit storage *tot* is assigned 0 (without quotes), which is the integer representation of the 8-bit binary number `00000000`. Being 8 bits wide, *tot* could be assigned the number 0, 1, 2, 3, ..., up to 255. In contrast, being a bit, `d` can only be assigned `'0'` or `'1'`. To assign a multibit item with a multibit constant, double quotes will be used, e.g., *tot* = `"00000000"`.
- To avoid confusion between arithmetic comparison and assignment, HLSMs will use "==" for comparison and ":=" for assignment. "=" will not be used for either. In Figure 5.3, note that assigning `d` is written as `d := '0'` rather than as `d = 0`, and assigning *tot* is written as *tot* := 0. If *tot* had to be compared to *s* for equality on a transition, such comparison would be written as *tot* == *s*, making clear that *tot* is

being compared with *s* and not being assigned the value of *s*. Comparing for less than or equal to would be written as *tot* <= *s*.

- Every HLSM multibit output is registered (see Chapter 3). As such, every HLSM multibit output *X* must have a storage item *Xreg* declared that is the same width as *X*. Writing to *X* is accomplished by writing to *Xreg*; writes directly to *X* are not allowed. *Xreg* can be read; *X* cannot. If desired, a single-bit output `B` can be registered by declaring a local storage item `Breg`.

To aid in understanding, an HLSM can include text that describes some aspect of the HLSM. Such text is preceded by "//" and is known as a ***comment***. One comment appears in Figure 5.3 to describe the behavior of output `d`.

As was true for FSMs, capturing behavior as an HLSM can be challenging. As in Chapter 3 for FSMs, the capture process for HLSMs can be aided by first listing all possible states (with some transitions included if helpful), then creating all possible transitions leaving each state, and finally by mentally executing the HLSM and refining it if necessary. The following example illustrates creation of an HLSM for a simple system.

Example 5.1 Cycles-high counter

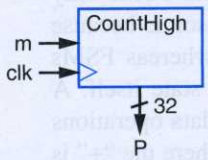


Figure 5.4 Cycles-high counter block diagram.

This example captures an HLSM for a system having a bit input `m` and a 32-bit output `P`. The system should output the total number of clock cycles for which the input `m` is `1`. For example, after powering on the system, if `m` is `0` for 20 cycles, then `1` for 15 cycles, then `0` for 12 cycles, and then `1` for 3 cycles, the system output at that point should be 18 (15 + 3). `P` connects to a display that converts the 32-bit number into a displayed integer. Such a cycles-high counter system might be useful to determine the total time that a car's brakes are applied, that a laser has been turned on, etc.

Figure 5.5(a) shows the first HLSM state *S_Clr* that initializes the output *P* to 0 by setting its storage item *Preg* to 0. *Preg* accomplishes the storage of the cycles-high count, and thus declaring another local storage item is not necessary. Figure 5.5(b) introduces a second state *S_Wt* that waits for `m` to be `1`; transitions are shown for this state. Finally, Figure 5.5(c) introduces a third state *S_Inc* that increments *Preg* once for each clock cycle that `m` is `1`; transitions are also shown for this state. The HLSM now has all possible transitions. Mentally executing the HLSM seems to validate that it correctly captures the desired behavior.

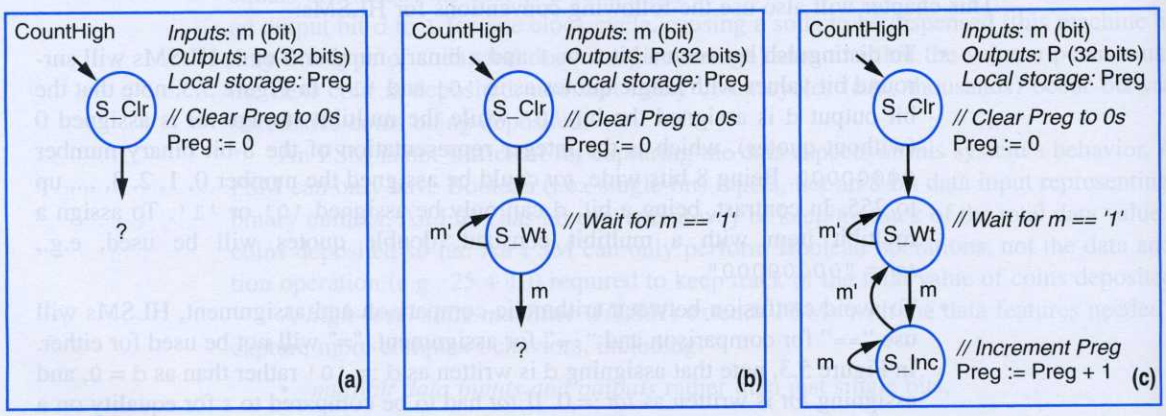


Figure 5.5 HLSM for cycles-high counter: (a) initial state, (b) waiting for `m` to be `1`, (c) incrementing *Preg* when `m` is `1`.

Example 5.2 Laser-based distance measurer

Many applications require accurately measuring 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 skyrise 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 $3x10^8$ meters/second, the distance D can be computed easily by the equation: $2D = T$ seconds * $3x10^8$ meters/second. Laser-based distance measurement is illustrated in Figure 5.6.

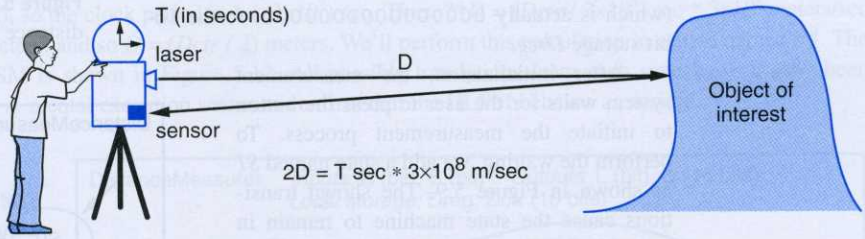


Figure 5.6 Laser-based distance measurement.

This example captures an HLSM to describe the behavior of a processor that controls a laser to compute distances up to 2000 meters. A block diagram of the system is shown in Figure 5.7. 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—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 the system should measure distances up to 2000 meters. To be safe, we'll make D 16 bits.

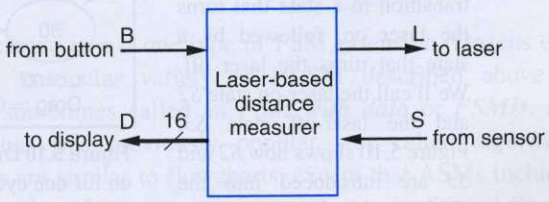


Figure 5.7 Block diagram of the laser-based distance measurer.

To facilitate the creation of the state machine, we first enumerate the sequence of events that would typically occur in 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 waits for the user to press button `B` to initiate measurement.
- 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 turned on, the system should wait for the sensor to detect the laser's reflection. Meanwhile, the system should count how many clock cycles occur from the time the laser was turned on until the reflection is sensed.
- After the reflection is sensed, the system should use the number of cycles that occurred 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 again so that a new measurement can be taken.

The above sequence guides our construction of an HLSM. We begin with an initial state named *S0* as shown in Figure 5.8. *S0's* task is to ensure that when the system powers on, the system does not output an incorrect distance, and the system does not turn the laser on. Note that output *D* is not written directly, but rather the output's local storage *Dreg* is written instead. Recall that the assignment `L := '0'` assigns the bit `0` to the one-bit output `L`, whereas the assignment *Dreg* := 0 assigns the 16-bit binary number 0 (which is actually `0000000000000000`) to the 16-bit storage *Dreg*.

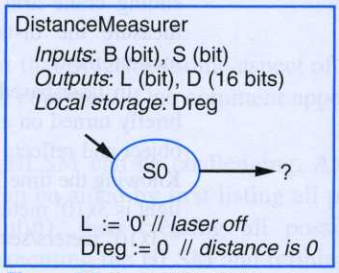


Figure 5.8 Partial HLSM for the distance measurer: Initialization.

After initialization, the measurement system waits for the user to press the button to initiate the measurement process. To perform the waiting, we add a state named *S1* as shown in Figure 5.9. The shown transitions cause the state machine to remain in state *S1* while `B` is `0` (i.e., while `B'` is true). Be aware that the `L` is implicitly assigned `0` in this state, because of the convention that any single-bit output not explicitly assigned in a state is implicitly assigned `0`. On the other hand, *Dreg*, corresponding to a local storage item, is unchanged in this state.

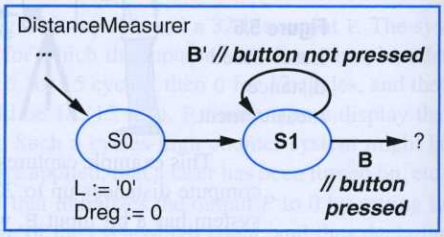


Figure 5.9 Distance measurer HLSM (cont.): Waiting for a button press.

When `B` becomes `1`, the laser should be turned on for one cycle. The HLSM 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 *S2* and the laser-off state *S3*. Figure 5.10 shows how *S2* and *S3* are introduced into the HLSM.

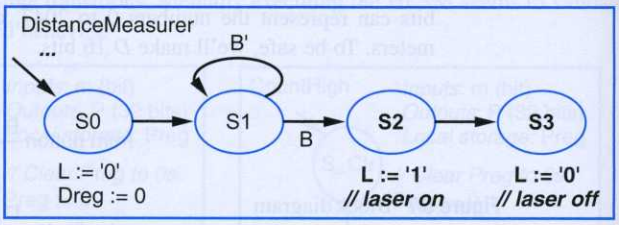


Figure 5.10 Distance measurer HLSM (cont.): Turning the laser on for one cycle.

In state *S3*, the HLSM should wait until the sensor detects the laser's reflection (`S`). The state machine remains in *S3* while `S` is `0`. The HLSM should meanwhile count the clock cycles between the laser being turned on and the laser's reflection being sensed. We can measure time by counting the number of clock cycles and multiplying that number by the clock period (*T* = cycles counted * clock period). Thus, we introduce a *local storage* item named *Dctr* to keep track of the cycles counted. The HLSM increments *Dctr* as long as the HLSM is waiting for the laser's reflection. (For simplicity, we ignore the possibility that no reflection is detected.) The HLSM must also initialize *Dctr* to 0 before

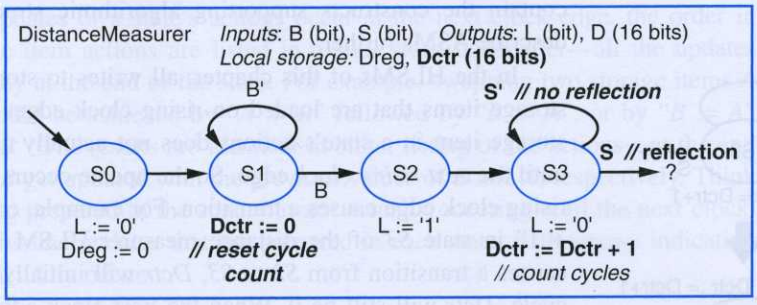


Figure 5.11 Distance measurer HLSM (cont.): Waiting for the laser reflection and counting clock cycles.

counting; such initialization can be added to state *S1*. With these modifications, the HLSM is seen in Figure 5.11.

Once the reflection is detected (`S=1`), the HLSM should compute the distance *D* that is being measured. Figure 5.6 shows that $2*D = T * 3x10^8$ m/sec. We know that the time *T* in seconds is *Dctr* * *clock period*. To simplify the calculation of *D*, assume the clock frequency is $3x10^8$ Hz (which is 300 MHz), so the clock period is $1 / (3x10^8)$ sec. Thus, $2*D = (Dctr / 3x10^8)$ sec $* 3x10^8$ meters/sec = *Dctr* meters, and so *D* = (*Dctr* / 2) meters. We'll perform this calculation in a state named *S4*. The final HLSM is shown in Figure 5.12. All possible transistions from each state have already been included. A mental execution seems to confirm that the HLSM behaves as desired.

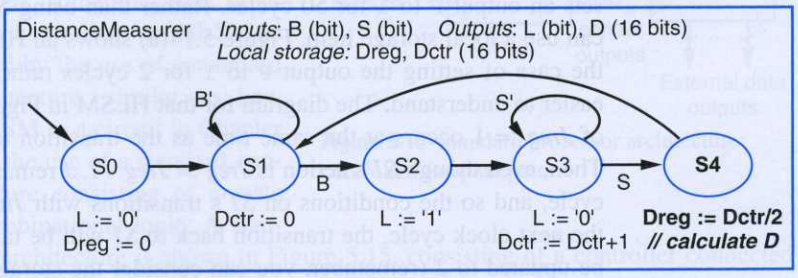


Figure 5.12 Completed HLSM for the distance measurer, including calculation of *D*.

A laser-based distance measurer could use a faster clock frequency to measure distance with a greater precision than 1 meter.

The HLSM described above is just one type of FSM extension. Dozens of extended FSM varieties exist. The particular variety of HLSM described above and used throughout this chapter is sometimes called an ***FSM with data*** or ***FSMD***. A different state machine variation that was previously 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 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, because hardware description languages

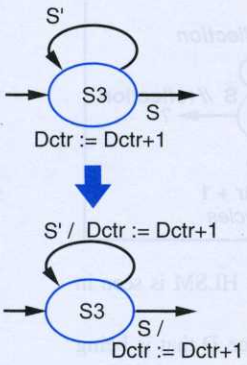


Figure 5.13 A storage update can be thought of as occuring on outgoing transitions.

contain the constructs supporting algorithmic structure, and much more. We do not describe ASMs further.

In the HLSMs of this chapter, all writes to storage items in a state's actions are to storage items that are loaded on rising clock edges only. As such, writing a value to a storage item in a state's actions does not actually cause the storage item to be updated until the *next rising clock* edge. So the update occurs at the *end* of the state when the next rising clock edge causes a transition. For example, consider the state action "*Dctr* := *Dctr* + 1" in state *S3* of the distance measurer HLSM in Figure 5.12. When a clock edge causes a transition from *S2* to *S3*, *Dctr* will initially be 0. While in *S3* during that clock cycle, *Dctr* will still be 0. When the *next* clock edge arrives, *Dctr* will be updated to 1 (i.e., to *Dctr* + 1, meaning 0 + 1 = 1), and the HLSM will transition back to *S3* if `S` is `0`. *Dctr* will remain 1 until the next clock edge, when it will become *Dctr* + 1 = 1 + 1 = 2. A simple way to visualize a state's local storage updates is to consider those updates as occuring on each outgoing transition instead of in the state, as in Figure 5.13. Because local storage item updates occur at the end of a state when the next rising clock edge occurs, and because transitions are taken on that same clock edge, then a transition whose condition uses the storage item uses the non-updated value. A common mistake is to assume the transition uses the updated value.

For example, suppose a system waits for an input control bit `B` to become `1` and then sets an output `P` to `1` for 50 cycles. Rather than using 50 states to set `P` high, an HLSM can use a local storage item. Figure 5.14(a) shows an HLSM using local storage *Jreg*, for the case of setting the output `P` to `1` for 2 cycles rather than 50 so that the example is easier to understand. The diagram for that HLSM in Figure 5.14(b) shows that *S0*'s action of *Jreg* := 1 occurs at the same time as the transition to *S1* during the next clock edge. Then, even though *S1*'s action is *Jreg* := *Jreg* +1, *J* remains 1 for the duration of the clock cycle, and so the conditions on *S1*'s transitions with *Jreg*<2 will be comparing 1<2. On the next clock cycle, the transition back to *S1* will be taken and *Jreg* will simultaneously be updated to 2 (remember, you can consider the storage update as occuring on the outgoing transitions). During that second cycle in *S1*, the transition comparison will be 2<2, so on the next clock edge the transition to *S0* will be taken and *Jreg* will simultaneously be updated with 3 (that value of 3 will never be used, and at the end of state *S0, Jreg* will become 1 again).

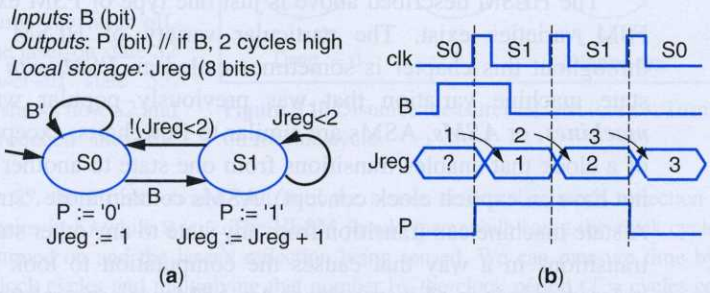


Figure 5.14 HLSM clocked storage update behavior: (a) HLSM with storage *Jreg*, (b) *S0*'s setting *Jreg* to *1* doesn't occur until the next clock edge, which is also when *S1* is entered. *S1*'s setting of *Jreg*:=*Jreg*+1 will set *Jreg* to 2, but not until the next edge, meaning *Jreg*<2 is false for the first *S1* state.

Because the updates of a state's actions occur at the next clock edge, the order in which local storage item actions are listed in a state does not matter—all the updates occur simultaneously at the end of the state. For example, swapping two storage items *A* and *B* in a state could be achieved by "*A* := *B*" followed by "*B* := *A*", or by "*B* := *A*" followed by "*A* := *B*." The results are identical in either listing of the actions—at the end of the state, *A* and *B* get updated with the previous values of *B* and *A*, respectively. Think of the updates being prepared in the state, but not actually occuring until the next clock. Figure 5.14(b) shows the updates that are prepared in each state, with arrows indicating when the updates actually occur.

5.3 RTL DESIGN PROCESS

RTL design follows a two-step process, as was the case for combinational design in Chapter 2 and for sequential design in Chapter 3. The first step is to *capture* the desired behavior, and the second step is to *convert* that behavior as a circuit. This chapter captures behavior using an HLSM; designers sometimes use other formalisms too. Converting an HLSM to a circuit is aided by the use of a standard processor architecture, similar to how converting an FSM to a circuit in Chapter 3 was aided by the use of a standard controller architecture consisting of a state register and combinational logic. A standard processor architecture is shown in Figure 5.15, consisting of a controller connected with a datapath. The datapath will have the ability to carry out each particular data operation present in the HLSM, by having the necessary datapath components (e.g., if the HLSM requires an addition and a comparison, then the datapath will include an adder and a comparator). The controller will set the control input signals of the various datapath components (e.g., the load control input of a register) to carry out the specific actions of each particular state and to transition to appropriate next states based on the control output signals of those datapath components.

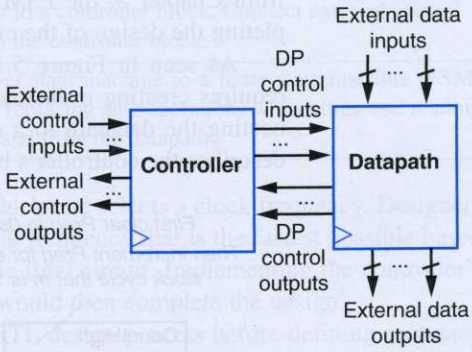


Figure 5.15 Standard processor architecture: controller/datapath pair.

To help understand the controller/datapath pair in the standard processor architecture, consider the cycles-high counter from Example 5.1, which is revisited in Figure 5.16. Figure 5.16(a) shows the system block diagram and describes the system's desired behavior. As this chapter's convention is to always register data outputs, the figure already shows a register *Preg* connected to data output *P*. Figure 5.16(b) shows the desired behavior captured as an HLSM created in the earlier example. The behavior requires that the computation "*Preg* := *Preg* + 1" be performed. As such, Figure 5.16(c) shows an adder whose two inputs are *Preg* and a constant 1, and whose output is connected to *Preg*. Thus, computing "*Preg* := *Preg* + 1" can be achieved simply by setting *Preg*'s `ld` control input to `1`. Furthermore, the required computation "*Preg* := 0" can be achieved by setting *Preg*'s `clr` control input to `1`. Thus, the circuit of the adder and reg-

ister form a datapath that is capable of performing the data computation required for the system's behavior.

What is still needed is a component, denoted as "?" in the figure, to control that datapath to carry out the right computation at the right time to achieve the desired cycles-high count behavior. That component is a controller as shown in Figure 5.16(d). The controller's behavior is described as an FSM (as in Chapter 3) and is similar to the HLSM's behavior in Figure 5.16(b), except that each desired data operation is replaced by control actions that use the datapath to carry out the desired data operation. Rather than "*Preg* := 0," the controller's FSM executes the action "`Preg_clr = 1`," which clears the register. Rather than "*Preg* := *Preg* + 1," the controller's FSM executes the action "`Preg_ld = 1`," which loads *Preg* with *Preg* + 1 because that is how the datapath is set up. The controller thus uses the datapath to implement the overall desired system behavior. Using methods from Chapter 3, the FSM in Figure 5.16(d) could be converted to a circuit, thus completing the design of the cycles-high count processor circuit.

As seen in Figure 5.16, converting an HLSM to a controller and datapath circuit requires creating a datapath capable of carrying out the required data operations, connecting the datapath to a controller block, and converting the HLSM into an FSM that describes the controller's behavior.

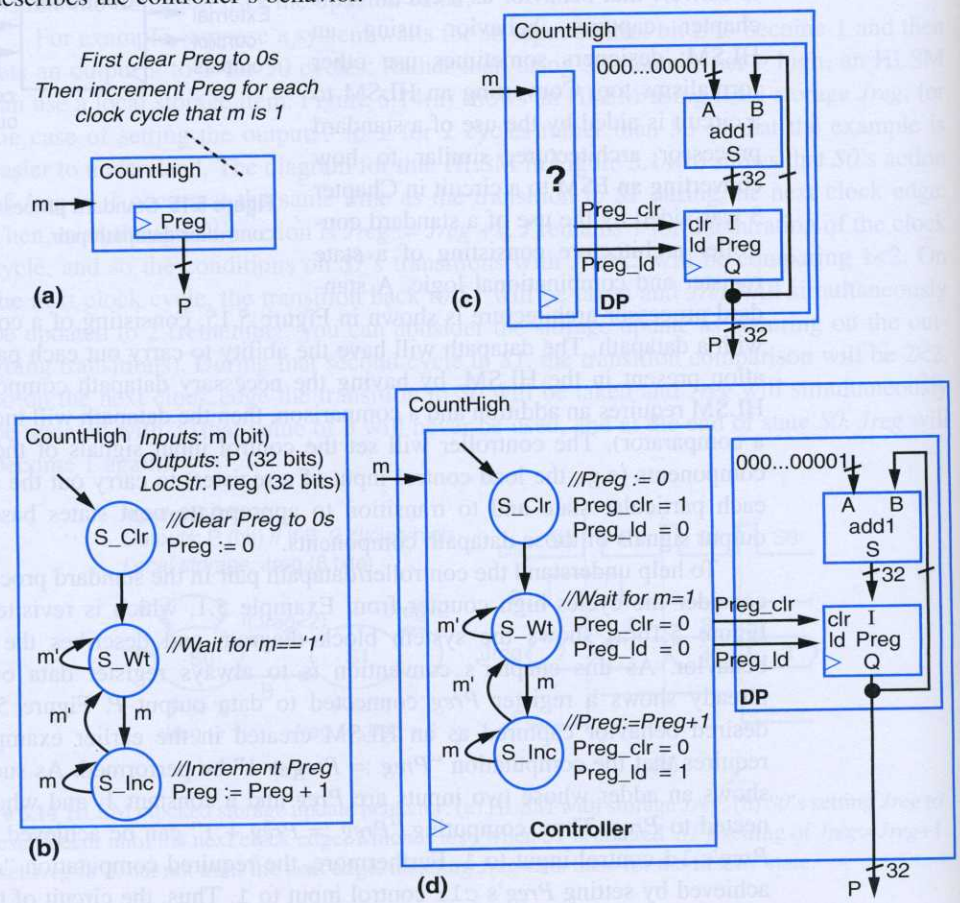


Figure 5.16
Example requiring a controller and datapath pair: (a) desired behavior, (b) HLSM description of desired behavior (*LocStr* means Local storage), (c) datapath with potential to implement behavior, but requiring time-varying setting of the datapath's control signals, (d) FSM showing datapath control signal values that should be set in each state to achieve behavior.

Thus, an RTL design process consisting of first capturing desired behavior and then converting the behavior to a circuit is summarized in Table 5.1.

Table 5.1 RTL design method.

		Step	Description
Step 1: Capture behavior		*Capture a high-level state machine*	Describe the system's desired behavior as a high-level state machine. The state machine consists of states and transitions. The state machine is "high-level" because the transition conditions and the state actions are more than just Boolean operations on single-bit inputs and outputs.
Step 2: Convert to circuit	**2A**	*Create a datapath*	Create a datapath to carry out the data operations of the high-level state machine.
	2B	*Connect the datapath to a controller*	Connect the datapath to a controller block. Connect external control inputs and outputs to the controller block.
	2C	*Derive the controller's FSM*	Convert the high-level state machine to a finite-state machine (FSM) for the controller, by replacing data operations with setting and reading of control signals to and from the datapath.

Another substep 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 in Chapter 3 would then complete the design.

We'll provide a simple example of the RTL design process before defining each step in more detail.

Example 5.3 Soda dispenser

This example completes the design of the soda dispenser whose HLSM was captured in Figure 5.3, which thus completed **Step 1** of the RTL design process.

Step 2A is to create a datapath. The datapath needs a register for the storage item *tot*, an adder connected to *tot* and *a* to compute *tot* + *a*, and a comparator connected to *tot* and *s* to compute *tot*<*s*. The resulting datapath appears in Figure 5.17.

Step 2B is to connect the datapath to a controller, as in Figure 5.18. Notice that the controller's inputs and outputs are all just one-bit signals.

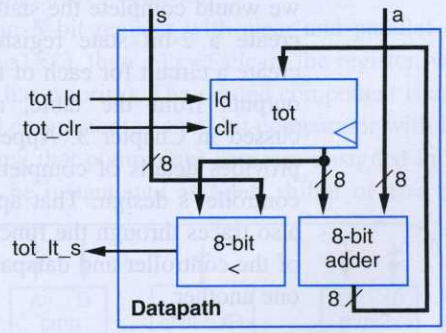


Figure 5.17 Soda dispenser datapath.

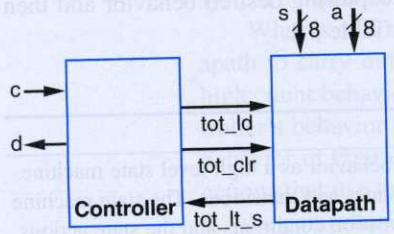


Figure 5.18 Soda dispenser controller/ datapath pair.

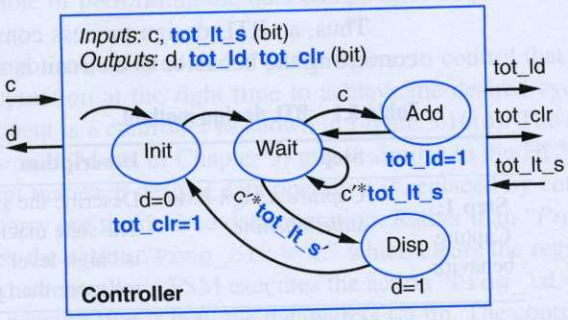


Figure 5.19 Soda dispenser controller FSM.

While an HLSM diagram uses ":=" and "==" to distinguish assignment and equality, and uses "'1'" and "1" to distinguish a bit 1 from an integer 1, an FSM diagram has no need for such distinctions (equality is never checked, and everything is a bit) and thus for FSM diagrams we still use the style defined in Chapter 3.

Step 2C is to derive the controller's FSM from the HLSM. The FSM has the same states and transitions as the HLSM, but utilizes the datapath to perform any data operations. Figure 5.19 shows the FSM for the controller. In the HLSM, state *Init* had a data operation of "*tot* := 0" (*tot* is 8 bits wide, so that assignment of 0 is a data operation). We replace that assignment by setting `tot_clr=1`, which clears the *tot* register to 0. State *Wait*'s transitions had data operations comparing "*tot* < *s*." Now that a comparator exists to compute that comparison for the controller, then the controller need only look at the result of that comparison by using the signal `tot_lt_s`. State *Add* had a data operation of *tot* = *tot* + *a*. The datapath computes that addition for the controller using the adder, so the controller merely needs to set `tot_ld=1` to cause the addition result to be loaded into the *tot* register.

To complete the design, we would implement the controller's FSM as a state register and combinational logic. Figure 5.20 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.

	s1	s0	c	tot_lt_s	n1	n0	d	tot_ld	tot_clr
Init	0	0	0	0	0	1	0	0	1
	0	0	0	1	0	1	0	0	1
	0	0	1	0	0	1	0	0	1
	0	0	1	1	0	1	0	0	1
Wait	0	1	0	0	1	1	0	0	0
	0	1	0	1	0	1	0	0	0
	0	1	1	0	1	0	0	0	0
	0	1	1	1	1	0	0	0	0
Add	1	0	0	0	0	1	0	1	0
			• • •				• • •		
Disp	1	1	0	0	0	0	1	0	0
			• • •				• • •		

Figure 5.20 Soda dispenser controller state table (partial).

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

Step 2A—Creating a Datapath using Components from a Library

Given a high-level state machine, the RTL design process requires creating a datapath that can implement all the data storage and computations on data items present in the HLSM. Doing so will enable replacing the HLSM by an FSM that merely controls the datapath. The task of "creating a datapath" can be decomposed into several subtasks.

Step 2A: Create a datapath

(a) Make all *data* inputs and outputs to be datapath inputs and outputs.

(b) Instantiate a register component into the datapath for every declared local storage item in the HLSM.

(c) Methodically examine each state action and each transition condition for data computations. Instantiate and connect datapath components to implement each data computation. Instantiate multiplexors in front of component inputs when muxes become necessary to share a component input among computations in different states.

Instantiate means to add a new component into a circuit. 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). An instantiated component is called an ***instance***. A new component instance should be given a name that is unique from any other datapath component instance's name. So a new register instance might be named "*Reg1*." Another register instantiated later might be named "*Reg2*." Actually, meaningful names should be used whenever possible. One register might be "*TemperatureReg*" and another register "*HumidityReg*."

A set of known components is needed to indicate what types of components can be instantiated. Such a set of allowable components is called a ***library***. Chapter 4 described the design and behavior of several datapath components. Figure 5.21 shows a small library consisting of several such components, indicating the inputs and outputs of each component and a brief summary of each component's behavior.

The first component in the library is an *N*-bit register with clear and parallel load functions. If the clock is rising (denoted as `clk^`), then `clr=1` clears the register, while `ld=1` loads the register (if both are `1`, clear has priority). The second component is an *N*-bit adder (without a carry output). The third component is an *N*-bit comparator with less-than, equals, and greater-than control outputs; that comparator assumes unsigned inputs. The fourth component is a shifter that can be instantiated as a left shifter or as a right

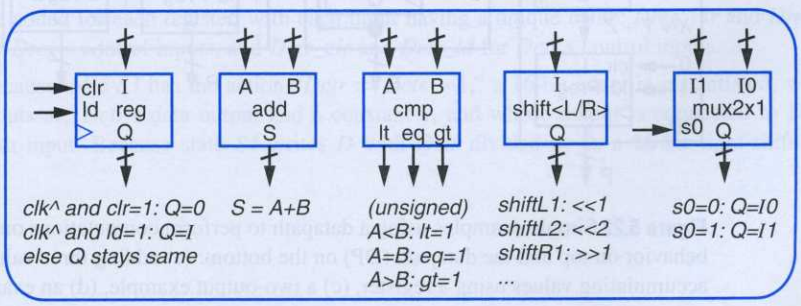


Figure 5.21 A basic datapath component library.

shifter, and that can be instantiated to shift a fixed number of places. The fifth component is an *N*-bit 2x1 multiplexor. Each component can be instantiated as any bitwidth *N*; e.g., an 8-bit register or a 32-bit register can be instantiated. Note that the concept of a library of datapath components differs from the familiar concept of a library as a collection of books. In a library of books, checking out a book means removing the book from the library, meaning nobody else can then check out that same book; only one instance of the book exists. A library of datapath components, in contrast, is more like a library of electronic books that can be printed out. Instantiating a component means creating a *new* occurrence of that component, much like one might print out a new copy of an electronic book—instantiating the component or printing a new copy of a book does not change the library of components or of electronic books.

Figure 5.22 provides several examples of converting (always occuring) actions into datapaths. Figure 5.22(a) indicates that the datapath should always compute "*Preg* := *X* + *Y* + *Z*." The corresponding datapath uses two adder instances and a register instance, connected such that the input to the register is *X* + *Y* + *Z*. The register's load control input is hardwired to `1`, meaning the register will be loaded on every clock cycle (for these examples, the actions always occur on every clock cycle so the control inputs are hardwired to constants; for HLSMs, the control inputs will be set on a state-by-state basis depending on the actions of the state).

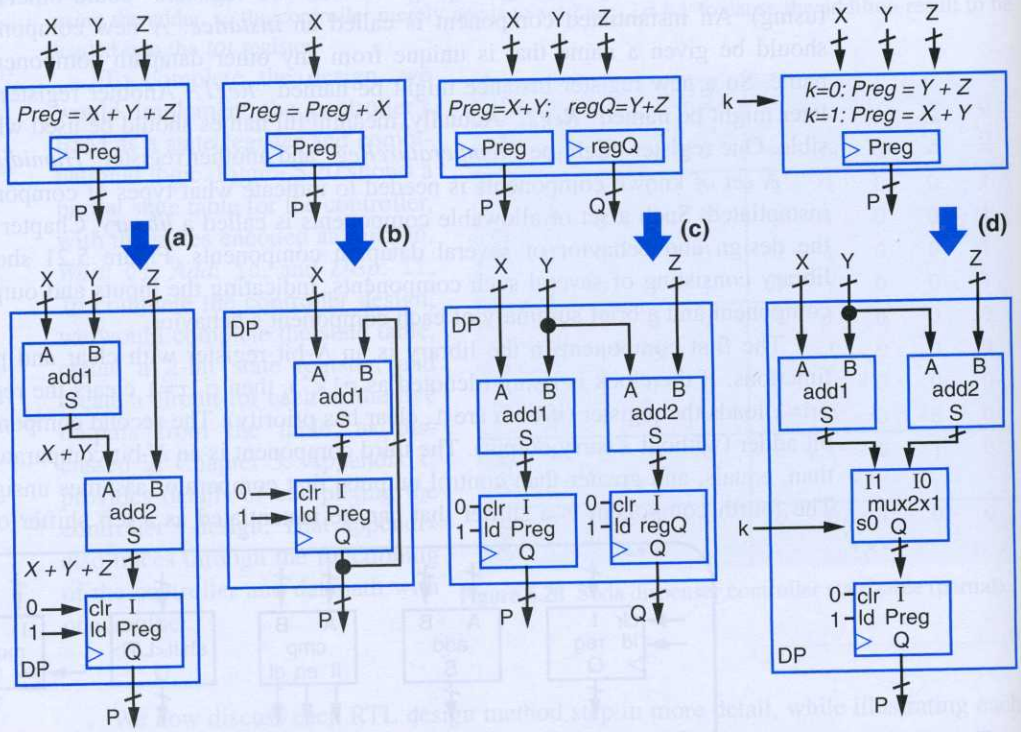


Figure 5.22 Simple examples using a datapath to perform computations on input data, with the desired behavior on top and the datapath (DP) on the bottom: (a) adding three values using two adders, (b) accumulating values using a register, (c) a two-output example, (d) an example requiring a mux.

Figure 5.22(b) indicates that the desired datapath behavior is adding the input *X* to register *Preg* on every clock cycle; the corresponding datapath thus connects *X* and the output of *Preg* to an adder, and connects the adder's output to *Preg*. Figure 5.22(c) indicates that the desired datapath behavior involves two distinct additions "*X+Y*" and "*Y+Z*" being computed for two distinct data outputs on each clock cycle; the datapath uses two adders and the shown connections. Finally, Figure 5.22(d) lists the desired behavior as loading *Preg* with *Y+Z* when the datapath control input `k` is `0`, and instead loading *Preg* with *X+Y* when `k` is `1`; the datapath uses two adders and uses a 2x1 mux to route the appropriate adder output to *Preg*'s input depending on the value of `k`. (In fact, an alternative datapath could use just one adder, with a mux in front of each adder input to route the appropriate inputs to the adder depending on `k`.)

When instantiating a new component for an HLSM's datapath, datapath control inputs must be introduced corresponding to the component's control inputs. For example, instantiating a register requires introducing two datapath control inputs corresponding to the register's load and clear control inputs. Unique names should be given to each new datapath control input, ideally describing which component the input controls and the control operation performed. For example, if a register named *Reg1* is instantiated, two new datapath control inputs must be added, possibly being named *Reg1_ld* and *Reg1_clr*. Likewise, control outputs of a component may be needed by the controller, like the output of a comparator, in which case those datapath control outputs should be given meaningful unique names too.

Example 5.4 Laser-based distance measurer—Creating a datapath

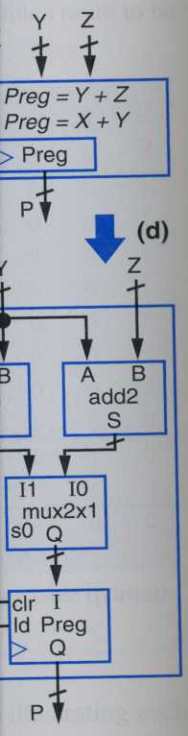
We continue Example 5.2 by first creating a datapath for the HLSM of Figure 5.12.

Step 2A—Create a datapath. We follow the subtasks of the create-a-datapath task to create the datapath shown in Figure 5.23:

(a) Output *D* is a data output (16 bits), so *D* becomes an output of the datapath, as shown in Figure 5.23.

(b) By convention of this chapter, every data output is registered, and thus a 16-bit register *Dreg* was declared as local storage. *Dreg* is now instantiated and connected to output *D*. A 16-bit register is also instantiated for the 16-bit local storage *Dctr*. Datapath control inputs are added for each register, with each input having a unique name: *Dreg_clr* and *Dreg_ld* for *Dreg*'s control inputs, and *Dctr_clr* and *Dctr_ld* for *Dctr*'s control inputs.

(c) Because state *S3* has the action "*Dctr* := *Dctr* + 1," a 16-bit adder is instantiated, whose inputs are *Dctr*'s data output and a constant 1, and whose output is connected to *Dctr*'s data input. Because state *S4* writes *D* with *Dctr* divided by 2, a 16-bit right-shift-by-1



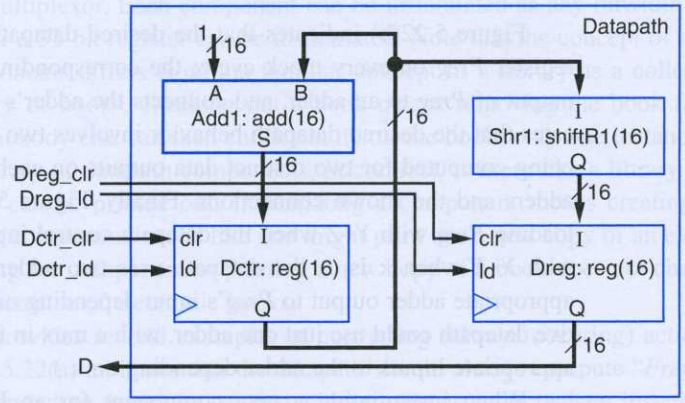


Figure 5.23 Datapath for the laser-based distance measurer.

component (which achieves division by 2 as discussed in Chapter 4) is instantiated between *Dctr* and *Dreg* to implement the divide by 2.

Step 2B—Connecting the Datapath to a Controller

Step 2B of the RTL design process is straightforward. This step simply involves creating a controller component having the system's control inputs and outputs, and then connecting the controller component with the datapath control inputs and outputs.

Example 5.5 Laser-based distance measurer—Connecting the datapath to a controller

Continuing the RTL design process for the previous example proceeds as follows.

Step 2B—Connect the datapath to a controller. We connect the datapath to a controller as shown in Figure 5.24. 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_clr*, *Dreg_ld*, *Dctr_clr*, and *Dctr_ld*. Normally we don't draw the clock generator component, but we've explicitly shown the clock generator in the figure to make clear that the generator must be exactly 300 MHz for this particular circuit.

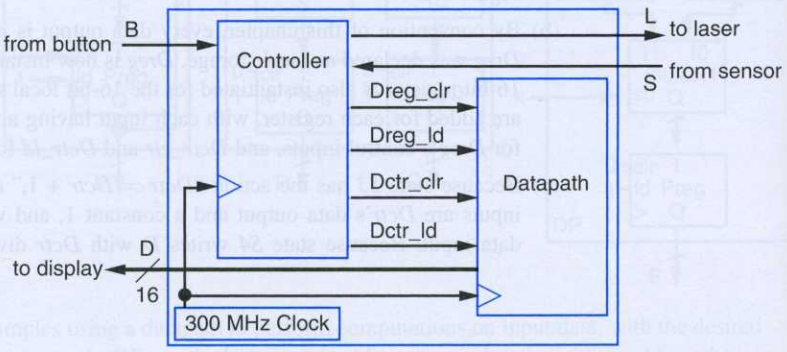


Figure 5.24 Controller/datapath (processor) design for the laser-based distance measurer.

Step 2C—Deriving the Controller's FSM

If the datapath was created correctly, then deriving an FSM for the controller is straightforward. The FSM will have the same states and transitions as the HLSM. We merely define the controllers inputs and outputs to be the FSM's inputs and outputs (all will now be single bits), and replace any data computations in the HLSM's actions and conditions by the appropriate datapath control signal values. Remember, the datapath was created 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 5.6 Laser-based distance measurer—Deriving the controller's FSM

Continuing the RTL design process for the previous example proceeds as follows.

Step 4—Derive the controller's FSM. This step defines the behavior of the controller. The controller's behavior is defined by converting the HLSM from Figure 5.12 into an FSM, replacing the data operations, like "*Dctr* := 0," by controller input and output signal assignments and conditions, like "*Dctr_clr* = 1," as shown in Figure 5.25. 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 by looking at both the controller's FSM and the datapath.

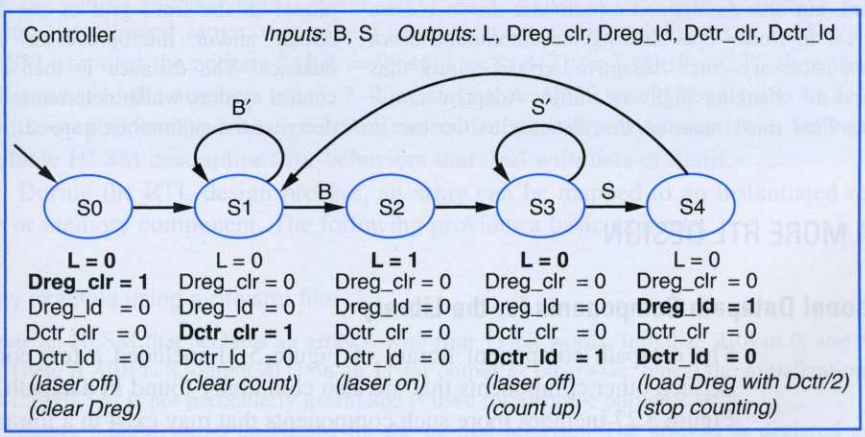


Figure 5.25 FSM description of the controller for the laser-based distance measurer. The desired action in each state is shown in italics in the bottom row; the corresponding bit signal assignment that achieves that action is shown in bold.

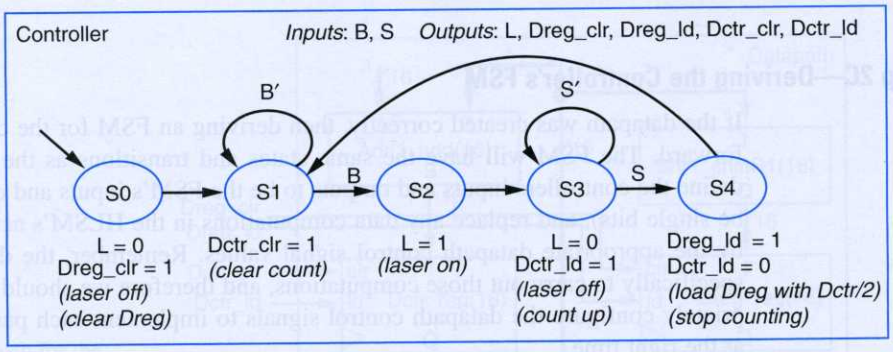


Figure 5.26 FSM description of the controller for the laser-based distance measurer, using the convention that FSM outputs not explicitly assigned a value in a state are implicitly assigned `0`.

Recall from Chapter 3 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 5.26. We may still choose to explictly 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 5.25.

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.

▶ HOW DOES IT WORK?—AUTOMOTIVE ADAPTIVE CRUISE CONTROL

The early 2000s saw the advent of automobile 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 grill 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.

▶ 5.4 MORE RTL DESIGN

Additional Datapath Components for the Library

The datapath component library of Figure 5.21 included a few components. Chapter 4 defined other components that are also commonly found in datapath component libraries. Figure 5.27 includes more such components that may exist in a library. One component is a subtractor; this particular subtractor used signed input and output numbers (a subtractor dealing with unsigned numbers is also possible). Another component is a multiplier; this multiplier deals with unsigned numbers (a multiplier for signed numbers is also possible). An absolute value component (designed in a Chapter 4 exercise) uses a signed number as input and outputs the number's magnitude as an unsigned number. An up-counter component is shown with synchronous clear and increment inputs; clear has priority. A down-

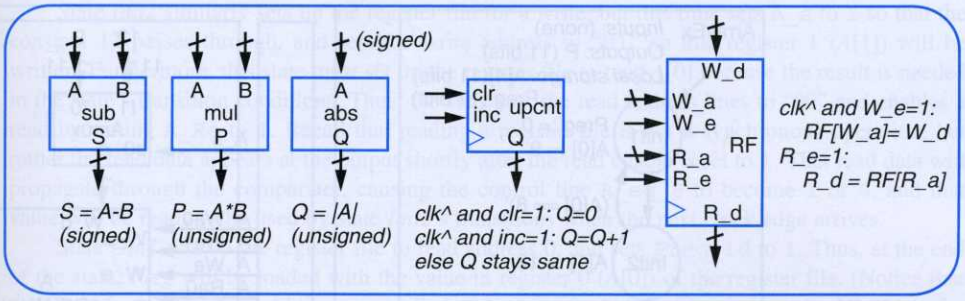


Figure 5.27
More datapath components for the library of Figure 5.21.

counter component could be similarly included. Each component can be instantiated with an arbitrary width of N. The last component is a register file with one write port and one read port. The component can be instantiated with a width of N and with a number of words M; the width of the address inputs W_a and R_a will be $\log_2(M)$, while W_d and R_d will have widths of N.

RTL Design Involving Register Files or Memories

RTL designs commonly involve register file or memory components. Register files were introduced in Chapter 4. Memory will be discussed in Section 5.7; for now, consider a memory to be a register file with just one port. Register file and memory components are especially useful for storing arrays.

An ***array*** in an HLSM is an ordered list of items, such as a list named A of four 8-bit numbers. Such a list might be defined in an HLSM as "Local storage: A[4](8-bit)." List items can be accessed using the notation "$A[i]$" where i is known as the ***index***. The indices start with 0. For example, A[0] reads item 0, and A[3] reads item 3 (which is actually the fourth and last item), while "A[1] := 8" writes 8 into item 1. Indices must be within the allowed range; e.g., A[–1] or A[4] are not allowed for array A above. If an HLSM executes the actions "A[0] := 9; A[1] := 8; A[2] := 7; A[3] := 22", then the array will consist of the numbers <9, 8, 7, 22>, and "X := A[1]" would set X to 8. An HLSM's inputs, outputs, or local storage items can be declared as arrays, which can lead to more readable HLSM descriptions for behaviors that deal with lists of items.

During the RTL design process, an array can be mapped to an instantiated register file or memory component. The following provides a basic example.

Example 5.7 Array example using a register file

Create an HLSM that declares an array A with four 11-bit words. Initialize A[0] to 9, and A[1] to 12. Then, if A[0] is 8, output A[1] on an 11-bit output P; otherwise, repeat the initialization. Note that this HLSM is not particularly useful and is used for example purposes only.

Figure 5.28(a) shows an HLSM for the desired behavior. The array A is declared as local storage, and the array's items such as A[0] and A[1] can then be read and written like other local storage items. Note that the first state initializes the output *Preg*—good practice involves always initializing outputs.

Figure 5.28(b) shows a datapath for the HLSM. The datapath has a register file of the same dimensions as the array. It has 11-bit constants 12 (which is actually `00000001100`), 9, and 8. Because the register file write input W_d at one time should get 9 and at another time should get 12,

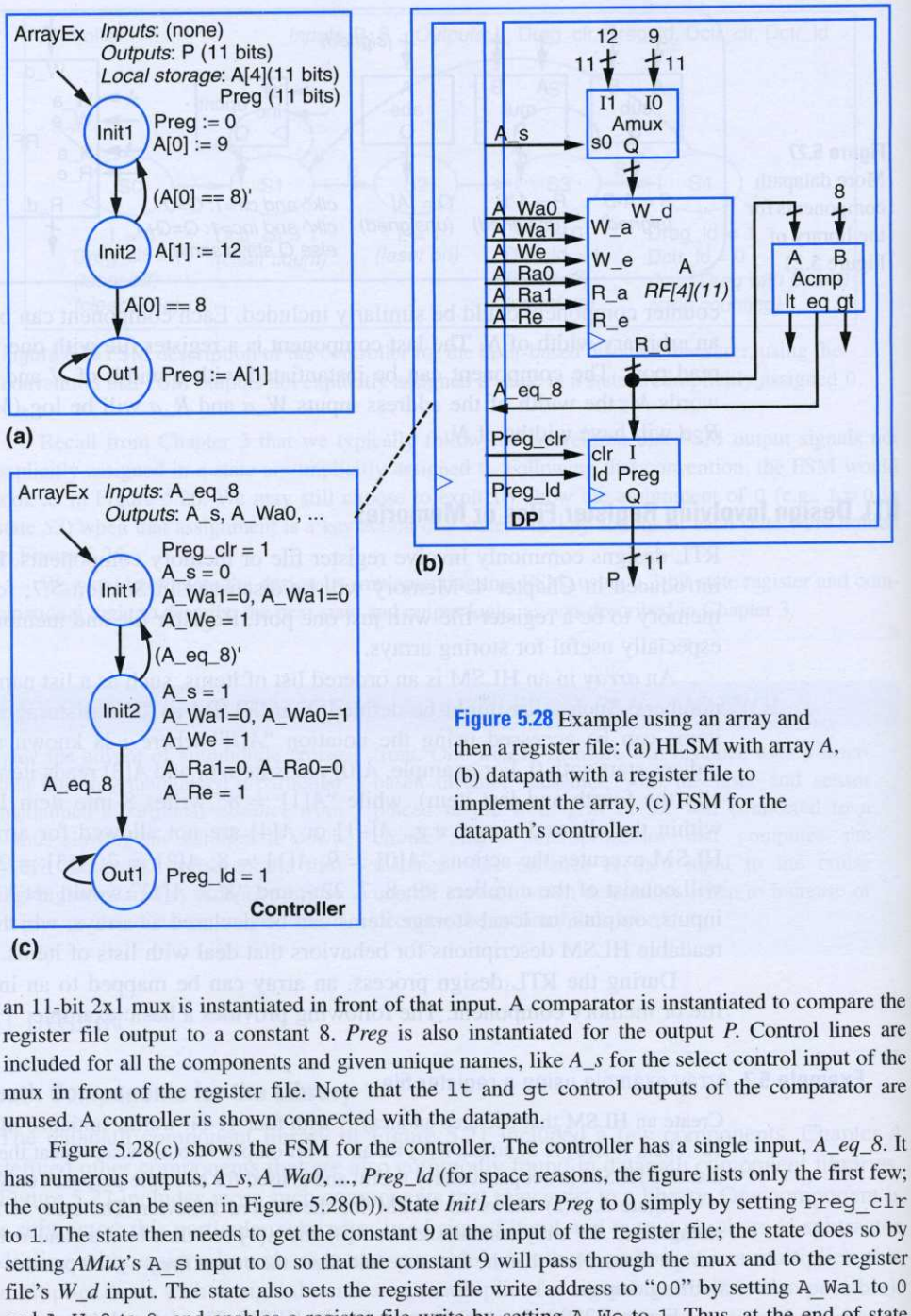


Figure 5.28 Example using an array and then a register file: (a) HLSM with array *A*, (b) datapath with a register file to implement the array, (c) FSM for the datapath's controller.

an 11-bit 2x1 mux is instantiated in front of that input. A comparator is instantiated to compare the register file output to a constant 8. *Preg* is also instantiated for the output *P*. Control lines are included for all the components and given unique names, like *A_s* for the select control input of the mux in front of the register file. Note that the `lt` and `gt` control outputs of the comparator are unused. A controller is shown connected with the datapath.

Figure 5.28(c) shows the FSM for the controller. The controller has a single input, *A_eq_8*. It has numerous outputs, *A_s*, *A_Wa0*, ..., *Preg_ld* (for space reasons, the figure lists only the first few; the outputs can be seen in Figure 5.28(b)). State *Init1* clears *Preg* to 0 simply by setting `Preg_clr` to `1`. The state then needs to get the constant 9 at the input of the register file; the state does so by setting *AMux*'s `A_s` input to `0` so that the constant 9 will pass through the mux and to the register file's *W_d* input. The state also sets the register file write address to "`00`" by setting `A_Wa1` to `0` and `A_Wa0` to `0`, and enables a register file write by setting `A_We` to `1`. Thus, at the end of state *Init1*, register 0 (which corresponds to *A*[0]) inside the register file will be loaded with the constant 9, and *Preg* will be cleared to 0.

State *Init2* similarly sets up the register file for a write, but this time sets `A_s` to `1` so that the constant 12 passes through, and sets the write address to "`01`" so that register 1 (*A*[1]) will be written. Furthermore, this state must set up the register file to read *A*[0] because the result is needed in the state's transition conditions. Thus, the state sets the read address lines to "00" and enables a read by setting `A_Re` to `1`. Recall that reading a register file is not a synchronous operation, but rather the read data appears at the output shortly after the read enable is set to `1`. That read data will propagate through the comparator, causing the control line `A_eq_8` to become `1` or `0`, and that value will be ready to be used by state *Init2*'s transitions when the next clock edge arrives.

State *Out1* sets up the register file to read address 0, and sets `Preg_ld` to `1`. Thus, at the end of the state, *Preg* will be loaded with the value in register 0 (*A*[0]) of the register file. (Notice that the HLSM as defined would never actually reach state *Out1* during execution; the HLSM is for example purposes only.)

We could have created the HLSM with the first state having both the actions *A*[0] := 9 and *A*[1] := 12. However, during implementation as a controller and datapath, we would have noticed that the only register file available has only one write port, and thus we would have had to introduce a second initialization state so that each state has no more than one write to the register file.

The previous example demonstrated basic use of a register file. The following example provides a more interesting use that illustrates the benefit of being able to index an array. The example also uses register files that exist external to the processor being designed.

Example 5.8 Video compression—sum of absolute differences

After a 2004 natural disaster in Indonesia, a TV news reporter broadcast from the scene by "camera phone." The video was smooth as long as the scene wasn't changing significantly. When the scene changed (like panning across the landscape), the video became very jerky—the camera phone had to transmit complete pictures rather than just differences, resulting in fewer frames transmitted over the limited bandwidth of the camera phone.

Digitized video is becoming increasingly commonplace, like in the case of DVDs (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 a frame may contain about a million pixels. Assume then that each frame requires about 1 Mbyte, and video is played at 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 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 for numerous frames, 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 is much 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***, pronounced "ess-aye-dee"*)*. For each pixel in frame 1, SAD involves computing difference between that pixel and the corresponding pixel in frame 2. Each pixel is represented by a number, so difference means the difference in numbers. Suppose a pixel is represented with a byte (real pixels are usually represented

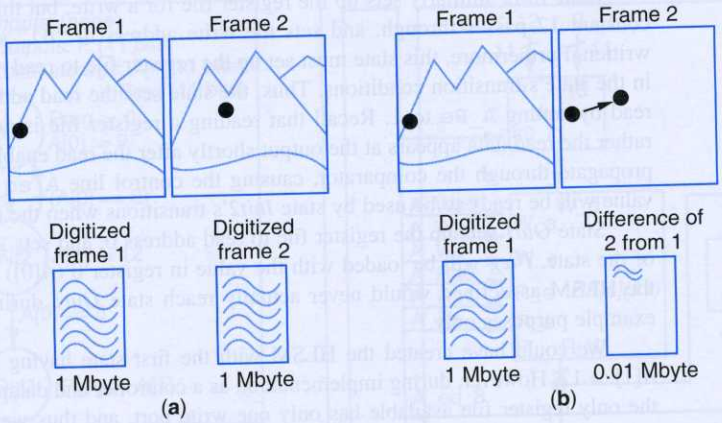


Figure 5.29 A key principle of video compression recognizes that successive frames have much similarity: (a) sending every frame as a distinct digitized picture, (b) instead, sending a base frame and then difference data, from which the original frames can later be reconstructed. If we could do this for 10 frames, (a) would require 1 Mbyte * 10 = 10 Mbytes, while (b) (compressed) would require only 1 Mbyte + 9 * 0.01 Mbyte = 1.09 Mbytes, an almost 10x size reduction.

by at least three bytes), and the pixels at the upper left of frames 1 and 2 in Figure 5.29(a) are being compared. 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. Thus, the difference of these two pixels is 255 – 255 = 0. SAD would 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 computing 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 only the difference matters to SAD, 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. Summing the absolute value of the differences for every pair of pixels results in a number that represents 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), the video compression method might apply the method of sending the difference data, as in Figure 5.29(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 the compression method might instead send the full digitized frame for frame 2. Thus, video with similarity among frames will achieve a higher compression than video with many 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 on a microprocessor, so that task may be done using a custom digital circuit, while other tasks may remain on a microprocessor. 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 5.30(a). The circuit's inputs will be a 256-byte register file *A*, holding the contents of a 16x16 block of pixels of frame 1, and another 256-byte register file *B*, holding the corresponding block of frame 2. Another

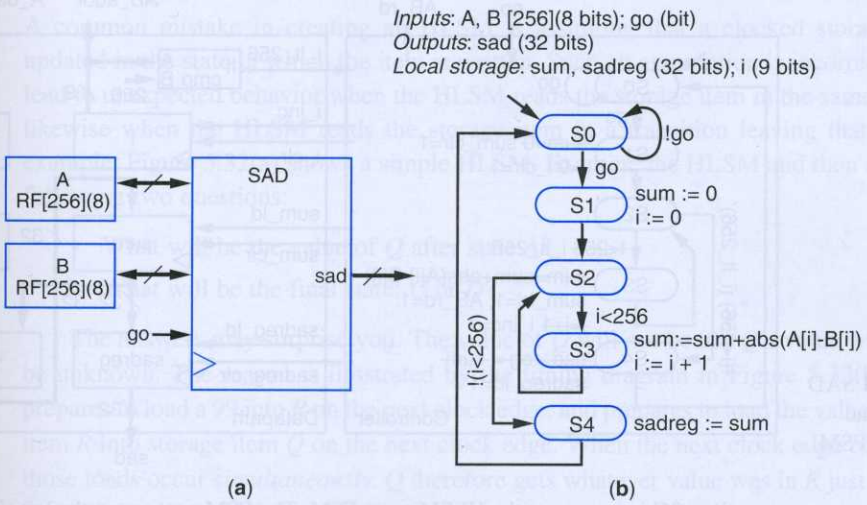


Figure 5.30 Sum-of-absolute-differences (SAD) component: (a) block diagram, and (b) HLSM.

circuit input `go` tells the circuit when to begin computing. An output *sad* will present the result after some number of clock cycles.

Step 1 of the RTL design process is to capture behavior with an HLSM. We can describe the behavior of the SAD component using the HLSM shown in Figure 5.30(b). We declare the inputs, outputs, and local registers *sum* and *i*. The *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. We also must declare the 32-bit storage item *sadreg* that registers the *sad* output. The HLSM initially waits for the input `go` to become `1`. The HLSM then initializes registers *sum* and *i* to 0. The HLSM next enters a loop: if *i* is less than 256, the HLSM computes the absolute value of the difference of the two blocks' pixels indexed by *i* (the notation *A[i]* refers to register *i* of register file *A*), updates the running sum, increments *i*, and repeats. Otherwise if *i* is not less than 256, the HLSM loads *sadreg* with the sum, which now represents the final sum, and returns to the first state to wait for the go signal to become `1` again. (The reader may notice that HLSM did not follow good design practice by not initializing *sadreg*; a better HLSM would include a state before S0 that initializes *sadreg* to 0).

We re-emphasize that the order of storage update actions in a state does not impact the results, because all those actions occur simultaneously at the end of the state. For the state inside the loop, arranging the actions as "*sum := sum + abs(A[i]-B[i]); i := i + 1*" or as "*i := i + 1; sum := sum + abs(A[i]-B[i])*" does not impact the results. Either arrangement uses the old value of *i*.

Step 2A of the RTL design process is to create a datapath. We see from the HLSM that we need a subtractor, an absolute-value component, an adder, and a comparison of `i` to 256. We build the datapath shown in Figure 5.31. The adder will be 32 bits wide, so the 8-bit input coming from the *abs* component will need to have `0`s appended as its high 24 bits. We have also introduced interfaces to the external register files. Data inputs *A_data* and *B_data* come from each register file. Because both register files always have the same address read at the same time, we use data output *AB_addr* as the address for both register files. We also use a control output `AB_rd` as the read enable for both register files.

Step 2B is to connect the datapath to a controller block, as shown in Figure 5.31.

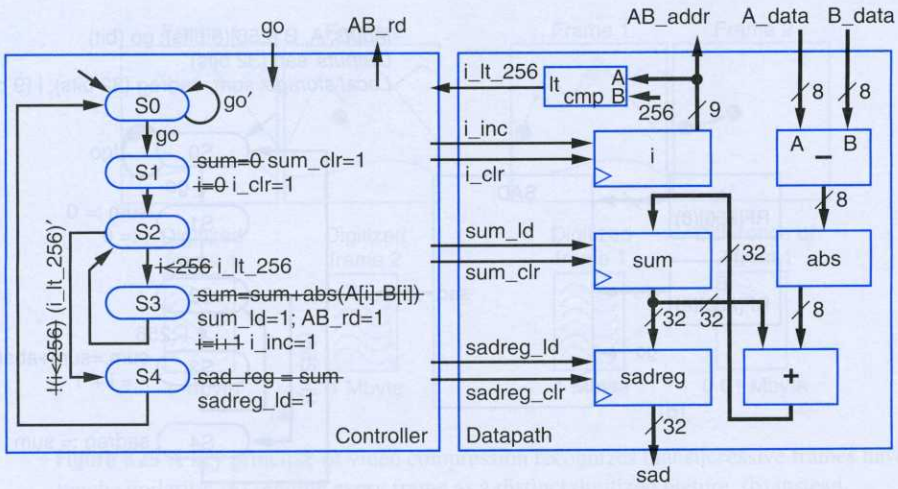


Figure 5.31 SAD datapath and controller FSM.

Step 2C is to convert the HLSM to an FSM. The FSM appears on the left side of Figure 5.31. For convenience, the FSM shows the original high-level actions (crossed out), and the 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.

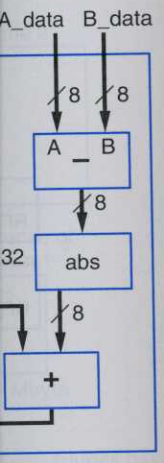
Comparing Microprocessor and Custom Circuit Implementations

Example 5.8 stated that output *sad* will present the result some number of clock cycles after `go` becomes 1. Let's determine that number of cycles. After `go` becomes `1`, the HLSM will spend one cycle initializing registers in *S1*, then will spend two cycles in each of the 256 loop iterations (states *S2* and *S3*), and finally one more cycle to update the output register in state *S4*, for a total of 1 + 2*256 + 1 = 514 cycles.

If the SAD algorithm ran on a microprocessor, the algorithm would likely need more than two clock cycles per loop iteration. It would need 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 built in the above example, at two cycles per iteration, is thus about three times faster for computing SAD, assuming equal clock frequencies. Section 6.5 will show how to 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 items could significantly change our living patterns.



left side of Figure 5.31.
t), and the replacement

ementation (a state reg-

ns
mber of clock cycles
r go becomes 1, the
nd two cycles in each
e cycle to update the

ould likely need more
to load internal regis-
, and a cycle for sum,
above example, at two
SAD, assuming equal
t that is *much* faster.

displays (with audio)
at continually display
e (perhaps our mom's
the other side of the
another place. Or we
s that enable us to
else wearing a tiny
spouse—sees. Those
ur living patterns.

RTL Design Pitfall Involving Storage Updates

A common mistake in creating an HLSM is assuming that a clocked storage item is updated in the state in which the item is written. Such an assumption is incorrect, and can lead to unexpected behavior when the HLSM reads the storage item in the same state, and likewise when the HLSM reads the storage item in a transition leaving that state. For example, Figure 5.32(a) shows a simple HLSM. Examine the HLSM and then answer the following two questions:

- What will be the value of Q after state A?
- What will be the final state: C or D?

The answers may surprise you. The value of Q will not be 99; Q's value will actually be unknown. The reason is illustrated by the timing diagram in Figure 5.32(b). State A prepares to load a 99 into R on the next clock edge, and prepares to load the value of storage item R into storage item Q on the next clock edge. When the next clock edge occurs, both those loads occur *simultaneously*. Q therefore gets whatever value was in R just before the next clock edge, which is unknown to us.

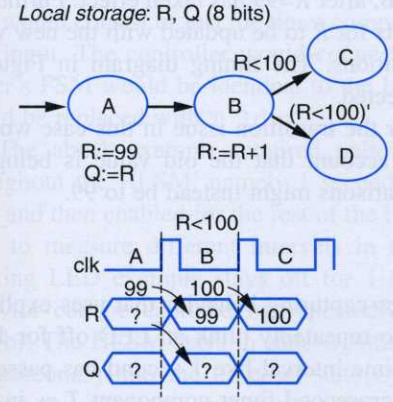


Figure 5.32 High-level state machine that behaves differently than some people may expect, due to reads of a clocked storage item in the same state as writes to that item: (a) HLSM, (b) timing diagram.

Furthermore, the final state will not be D, but will rather be C. The reason is illustrated by the timing diagram in Figure 5.32(b). State B prepares to load 100 into R on the next clock cycle, and prepares to load the next state based on the transition condition. R is 99, and therefore the transition condition $R<100$ is true, meaning the HLSM is prepared to make state C the next state, not state D. On the next clock edge, R becomes 100, and the next state becomes C.

The key is to always remember that *a state's actions* ***prepare*** *the values that will occur on the next rising clock edge—but those values* ***don't actually get loaded into storage items*** *until that next clock edge*. Thus, any expressions in a state's actions or outgoing transition conditions will be using the previous values of storage items, 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 listed in any order.

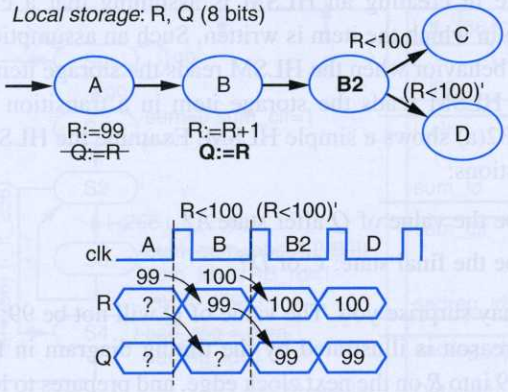


Figure 5.33 HLSM that ensures storage item reads occur at least one state after writes: (a) HLSM, (b) timing diagram.

Assuming that the designer actually wants Q to equal 99 and the final state to be D, then a solution is to insure that local storage writes occur at least one state before reads that rely on those writes. Figure 5.33(a) shows a new HLSM in which the assignment of Q:=R has been moved to state B, after R=99 has taken effect. Furthermore, the HLSM has a new state $B2$ that simply waits for R to be updated with the new value before that value is read in the transition conditions. The timing diagram in Figure 5.33(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 be to 99.

RTL Design Involving a Timer

RTL design commonly requires capturing behavior that uses explicit time intervals. For example, a design may have to repeatedly blink an LED off for 1 second and on for 1 second. Determining when a time interval like 1 second has passed can be achieved by pre-instantiating a 32-bit 1-microsecond timer component T as in Figure 5.34(a). Then, an HLSM as shown in Figure 5.34(b) can declare a special item T as a timer, which can be loaded like a local storage item but which also has an enable input `T_en` and a rollover output `T_Q` implicitly associated with the item. The HLSM can make use of the declared timer to detect 1-second intervals. The HLSM loads 1 second (1,000,000 microseconds) into the timer in state *Init* by writing T := 1000000. The HLSM then enables the timer in state *Off* by setting `T_en` to `1`, and also sets output `L` to `0` to turn the LED off. The HLSM waits in that state until the timer's `T_Q` output becomes `1`, which will happen after 1 second. The HLSM then transitions to state *On*, which turns on the LED, and stays in that state until `T_Q` becomes `1` again, which will happen after another 1 second. The HLSM returns to state *Off* , staying there for 1 second again, and so on. (Note that this HLSM

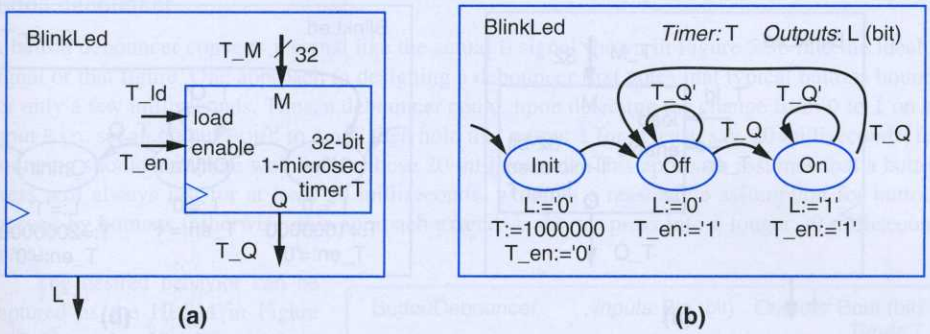


Figure 5.34 Blinking LED example: (a) pre-instantiated timer, (b) HLSM making use of the timer to turn the LED off for 1 second (1,000,000 microseconds), then off for 1 second, and repeating.

assumes that the timer output `T_Q` stays `1` for only one clock cycle, which indeed is how the timer compnent was designed in Chapter 4.)

The HLSM can be straighforwardly converted to a controller and datapath. The datapath would consist of just the timer component with the constant "1000000" at the timer's data input. The controller would connect with the timer's control signals, and the controller's FSM would be identical to the HLSM except that the assignment $T := 1000000$ would be replaced with `T_ld = 1`.

The above example required only one time interval to be repeatedly measured throughout the HLSM, namely 1 second; as such, the timer component was initialized once and then enabled for the rest of the HLSM's execution. However, a timer can also be used to measure different intervals in the same HLSM. For example, an alternative blinking LED example stays off for 1 second but then stays on for 2 seconds. Such behavior can be captured by repeatedly re-initializing the timer as shown in Figure 5.35(b). The HLSM stays in state *OffWait* for 1 second, by loading 1 second (1,000,000 microseconds) into the timer in state *OffInit*, and then by enabling the timer in state *OffWait* and staying in that state until the timer's output `T_Q` becomes `1`. Likewise, state *OnInit* loads 2 seconds into the timer, and state *OnWait* enables the timer and waits for the timer's output to become `1`. Converting this HLSM to a processor would thus have the timer *T* in the datapath. *T_M* would have a 2x1 mux to route either 1000000 or 2000000 to the timer's data input, thus completing the datapath. The controller would connect with the timer and mux control signals. Converting the HLSM to an FSM for the controller would consist merely of replacing the *T* assignments by the appropriate 2x1 mux select line assignment and by setting `T_ld` to `1`. (Notice that each initialization state adds an extra clock cycle, which is likely not a problem for a blinking LED system but would need to be compensated for if a precise 3 second blinking period was required.) An alternative solution could use two timers, one to compute the 1-second interval, and the other for the 2-second interval.

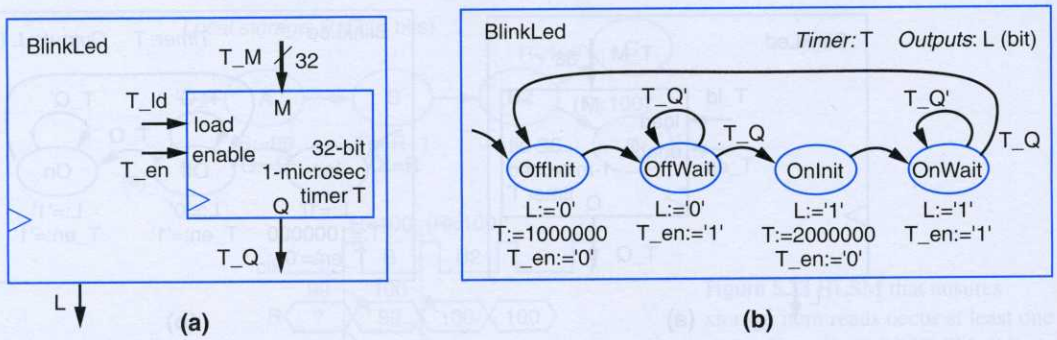


Figure 5.35 Blinking LED example: (a) pre-instantiated timer, (b) HLSM making use of the timer to turn the LED on for 1 second (1,000,000 microseconds), then off for 2 seconds, and repeating.

Button Debouncing

Button debouncing is a typical task in RTL design that also illustrates the use of a timer. A button is an common input device to a digital circuit. Ideally, a button outputs `1` when pressed, and outputs `0` when not pressed. Actually, though, a real button may output some spurious `0`s just after being pressed. The reason is because the button is a mechanical device, and pressing the button down down results in a small amount of ***bounce*** as the button settles in the down position. An analogy is dropping a hard ball such as a billiard ball or a bowling ball onto the floor—the ball will bounce slightly before coming to rest onto the floor. The bounce of a button is illustrated in Figure 5.36. Instead of `B` becoming `1` and staying `1` when initially pressed down, `B` changes to `1`, then briefly changes back to `0`, then to `1`, then `0` again, and finally settles at `1`. Typical buttons may exhibit bounce for a few milliseconds after being pressed down. Bounce is a problem because the bouncing signal appears to be several distinct rapid button presses rather than just one press—the actual `B` signal in Figure 5.36 suggests that the button was pressed three times. A simple solution is to create a circuit that converts the actual `B` signal into the ideal `B` signal, a process known as ***button debouncing***.

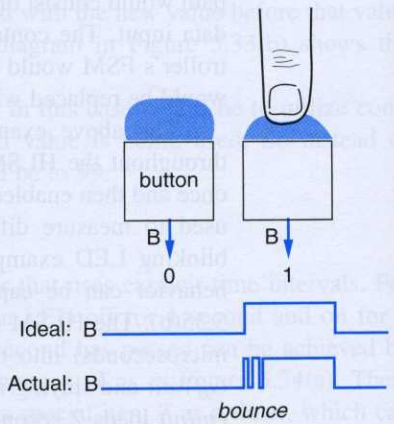
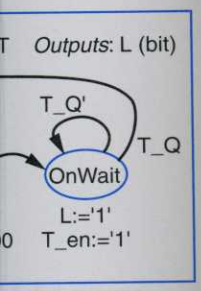
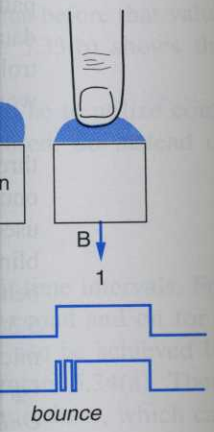


Figure 5.36 Button bounce.



the timer to turn the



bounce.

ges to 1, then briefly
. Typical buttons may
Bounce is a problem
ton presses rather than
he button was pressed
he actual B signal into

Example 5.9 Button debouncer

A button debouncer converts a signal like the actual B signal shown in Figure 5.36 into the ideal B signal of that figure. One approach to designing a debouncer first notes that typical buttons bounce for only a few milliseconds. Thus, a debouncer could, upon detecting the change from 0 to 1 on an input Bin, set an output Bout to 1 and then hold that output 1 for at least, say, 20 milliseconds; the bouncing should complete well before those 20 milliseconds. This approach assumes that a button press will always last for at least 20 milliseconds, which is a reasonable assumption for buttons pressed by humans (otherwise, this approach extends a shorter press into a longer 20-millisecond press).

The desired behavior can be captured as the HLSM in Figure 5.37. Measuring 20 milliseconds in the HLSM is achieved by pre-instantiating a 32-bit microsecond timer component, and then using that component in the HLSM. State *Init* loads the timer component with the value 20 milliseconds (20,000 microseconds). State *WaitBin* waits for the button input Bin to become 1, after which state *Wait20* sets Bout to 1, enables the timer, and waits until the timer output T_Q becomes 1, thus waiting for 20 milliseconds regardless of the value of Bin. After the 20 milliseconds have passed, the HLSM enters state *WhileBin*, which continues to set Bout to 1 as long as Bin is still 1. When Bin returns to 0 (meaning the button is released), the HLSM starts over again.

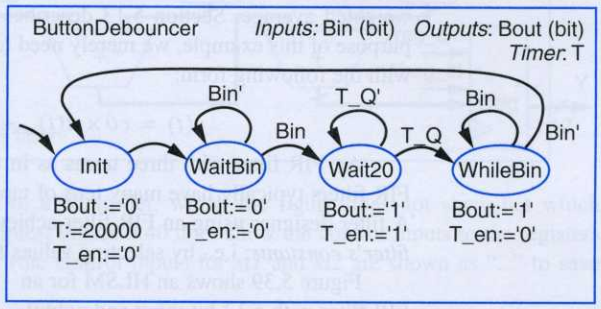


Figure 5.37 Button debouncer.

A Data-Dominated RTL Design Example

Some systems have an extensive datapath and a relatively simple controller. Such a system is known as ***data-dominated system***. In contrast, a system with a simple datapath and relatively complex controller is known as a ***control-dominated system***. A data-dominated system may have only a few states, and in fact may have only one state. Within those few states, however, may be extensive computations. Nevertheless, the RTL design process can proceed as before. The following example illustrates the design of a data-dominated system.

Example 5.10 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. A ***stream*** is a sequence of values separated by a fixed amount of time. Figure 5.38 shows a block diagram of a popular digital filter known as an FIR filter. Input X and output Y are N bits wide each, such as 12 bits each. As a filtering example,

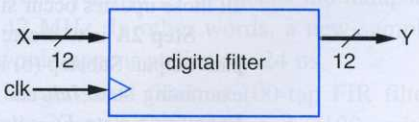


Figure 5.38 FIR filter block diagram.

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 finite impulse response filter, or FIR filter (commonly pronounced by saying the letters "F" "I" "R"), is a popular general digital filter design that can be used for a variety of filtering goals. Figure 5.38 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. Section 5.13 describes digital filtering and FIR filters in more detail. For the purpose of this example, we merely need to know that an FIR filter can be described by an equation with the following form:

$$y(t) = c0 \times x(t) + c1 \times x(t-1) + c2 \times x(t-2)$$

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*; i.e., by selecting values for *c0*, *c1*, *c2*, etc.

Figure 5.39 shows an HLSM for an FIR filter with a 12-bit input and output, thus fulfilling **Step 1** of the RTL design process. Note the simplicitly of the HLSM, which has only two states, and which actually spends all its time in just the second state *FC* (standing for "FIR Compute"). In addition to declaring the input *X* and output *Y*, the HLSM declares 12-bit local storage items *xt0*, *xt1*, and *xt2* for the three most recent input values; *xt0* will be for the current value, *xt1* for the value from time *t-1* (the previous clock cycle), and *xt2* for the value from time *t-2*. It also declares 12-bit local storage items for the three filter constants *c0*, *c1*, and *c2*, and for the storage item *Yreg* associated with output *Y*. State *Init* initializes the output by setting the output's storage item *Yreg* to 0. The state also initializes the *xt* registers to 0s. The state sets the *c* registers to hold the constants for this particular FIR filter, which for this filter are 3, 2, and 2. State *FC* computes the FIR equation for the current values in the *xt* registers and sets *Yreg* to the computed value. That state also updates the *xt* registers, setting *xt0* to the current value on input *X*, *xt1* to *xt0*, and *xt2* to *xt1*. Recall that all those updates occur at the end of the state when the next rising clock edge arrives, and that all those updates occur simultaneously.

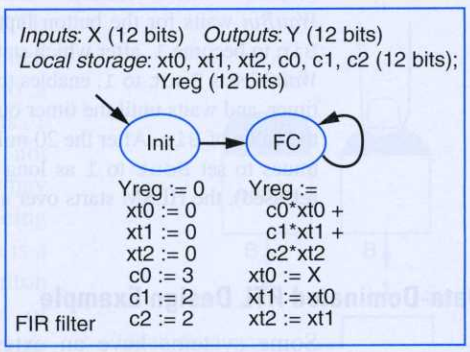


Figure 5.39 FIR filter HLSM.

Step 2A is to create a datapath. Substep (a) involves making *X* a datapath input and *Y* a datapath output. Substep (b) involves instantiating the seven local storage registers. Substep (c) involves examining state *Init* and then instantiating constants 3, 2, and 2 at the inputs of the *c* registers. Examining state *FC* reveals the need to instantiate 3 multipliers and two adders, and to connect them as shown in the figure. That state also requires that the inputs of each *xt* register be connected as shown.

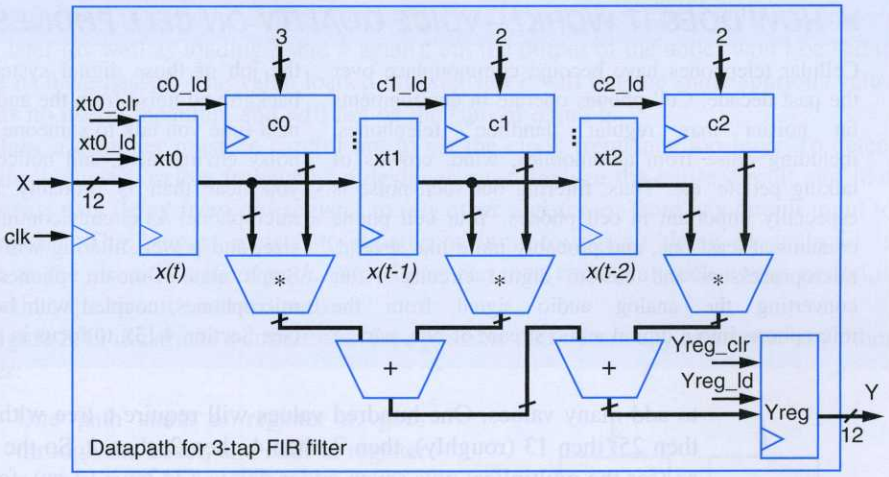


Figure 5.40 FIR filter datapath.

Step 2B connects the datapath to a controller, which the figure does not show but which follows similarly from previous examples. Figure 5.40 does show the control inputs to the registers that will be needed by the controller (the control inputs for *xt1* and *xt2* are shown as "..." to save space in the figure).

Finally, **Step 2C** would convert the HLSM of Figure 5.39 into an FSM for the controller. State *Init* would set the clear line for *Yreg* and each *xt* reg to `1`, and would set the load line for each *c* register to `1`. State *FC* would simply set the load line for *Yreg* and for each *xt* reg to `1`.

Commonly an FIR filter should sample the input at a specified rate, such as once every 10 microseconds. A timer can be used for this purpose. The HLSM would be extended to configure, enable, and then monitor the timer as in earlier examples.

Comparing Microprocessor and Custom Circuit Implementations

It is interesting to compare the performance of the circuit implementation of a 3-tap FIR filter with a microprocessor implementation. The datapath's critical path goes from the *xt* and *c* registers, through one multiplier, and through two adders, before reaching *Yreg*. For the circuit implementation, assume that the adder has a 2 ns delay. Also assume that chaining the adders together results in the delays adding, so that two adders chained together have a delay of 4 ns (detailed analysis of the internal gates of the adders could show the delay to actually be slightly less). Assume the multiplier has a 20 ns delay. Then the critical path, or longest register-to-register delay (to be discussed further in Section 5.5) would be from *c0* to *Yreg*, going through the multiplier and two adders as shown in Figure 5.40. That path's length would be 20 + 4 = 24 ns. Note that the path from *c1* to *Yreg* would be equally long, but not longer. A critical path of 24 ns means the datapath could be clocked at a frequency of 1 / 24 ns = 42 MHz. In other words, a new sample could appear at *X* every 24 ns, and new outputs would appear at *Y* every 24 ns.

Now consider the circuit performance of a larger-sized filter: a 100-tap FIR filter rather than a 3-tap filter. Assume that 100 multipliers are available—then the 100 multiplications could occur simultaneously in 100 multipliers just as the 3 multiplications occurred simultaneously. Thus, the main performance difference is that the circuit must add 100 values rather than just 3. Recall from Section 4.13 that an adder tree is a fast way

▶ HOW DOES IT WORK?—VOICE QUALITY ON CELL PHONES.

Cellular telephones have become commonplace over the past decade. Cell phones operate in environments far noisier than regular "landline" telephones, including noise from automobiles, wind, crowds of talking people, etc. Thus, filtering out such noise is especially important in cell phones. Your cell phone contains at least one, and probably more like several, microprocessors and custom digital circuits. After converting the analog audio signal from the microphone into a digital audio stream of bits, part of the job of those digital systems is to filter out the background noise from the audio signal. Pay attention next time you talk to someone using a cell phone in a noisy environment, and notice how much less noise you hear than is probably actually heard by the microphone. As circuits continue to improve in speed, size, and power, filtering will likely improve further. Some state-of-the-art phones may even use two microphones, coupled with beamforming techniques (see Section 4.13), to focus in on a user's voice.

to add many values. One hundred values will require a tree with 7 levels—50 additions, then 25, then 13 (roughly), then 7, then 4, then 2, then 1. So the total delay would be 20 ns (for the multiplier) plus seven adder-delays (7*2ns = 14 ns), for a total delay of 34 ns.

For a microprocessor implementation, assume 10 ns per instruction. Assume that each multiplication or addition would require two instructions. A 100-tap filter would need approximately 100 multiplications and 100 additions, so the total time would be (100 multiplications * 2 instr/mult + 100 additions * 2 instr/add) * 10 ns per instruction = 4000 ns. In other words, the circuit implementation would be over 100 times faster (4000 ns / 34 ns) than the microprocessor implementation. A circuit could therefore process 100 times more data than a microprocessor implementation, resulting in better filtering.

▶ 5.5 DETERMINING CLOCK FREQUENCY

RTL design produces a processor circuit consisting of a datapath and a controller. Inside the datapath and controller are registers that require a clock signal. A clock signal has a particular ***clock frequency***, which is the number of clock cycles per second, also known as Hertz (Hz). The frequency impacts how fast the circuit executes 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 clock period (the duration of a clock cycle, which is the inverse of frequency) is slower, while a smaller period is faster.

Designers of digital circuits often want their circuits 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 5.41, 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). 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

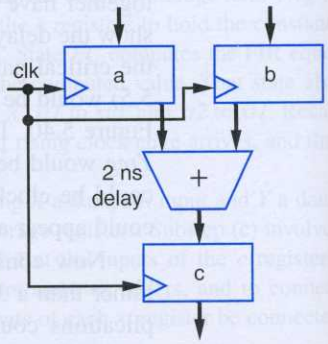


Figure 5.41 Longest path is 2 ns.

cycle might load new values into registers *a* and *b*. The next clock cycle will load register *c* 1 ns later (as well as loading *a* and *b* again), 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.*** A designer can then choose a clock whose period is *longer* than the circuit's critical path.

Figure 5.42 illustrates a circuit with 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*. That path's delay is 2 ns.
- Another path starts at register *a*, goes through the adder, then through the multiplier, and ends at register *d*. That 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*. That 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*. That path's delay is 5 ns.

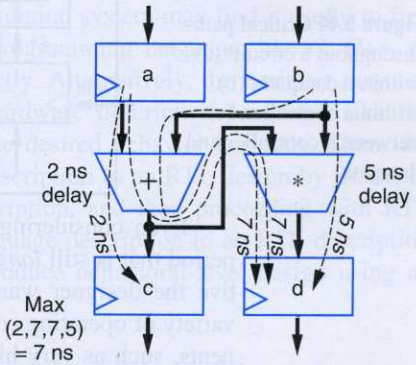


Figure 5.42 Determining the critical path.

The longest path is thus 7 ns (there are two paths with that delay). 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 5.43 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 also adds to the path length.

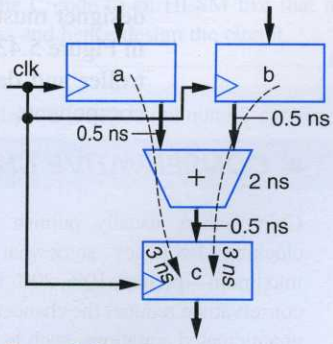


Figure 5.43 Longest path is 3 ns, considering wire delays.

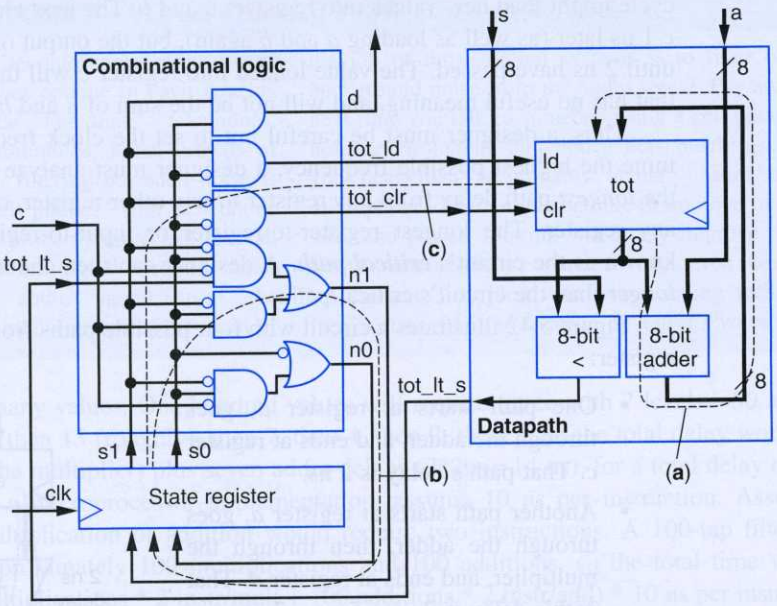


Figure 5.44 Critical paths throughout a circuit: (a) within a datapath, (b) within a controller, (c) between a controller and datapath.

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 that the circuit works under a variety of operating conditions. Certain conditions can change the delay of circuit components, such as very high temperature, very low temperature, age, electrical interference, etc. Generally, the longer the period beyond the critical path, the more conservative the design. For example, a designer might determine that the critical path is 7 ns, but might choose a clock period of 10 ns or even 15 ns, the latter being very conservative.

If desiring low power, a designer might choose an even lower frequency to reduce circuit power. Section 6.6 describes why reducing the clock frequency reduces power.

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 as in Figure 5.43(a), but also within the controller as in Figure 5.43(b), and between the controller and datapath as in Figure 5.43(c), and even between the processor and external components.

▶ 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 setting 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. We 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.

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 $N*N$, or N^2 possible register-to-register paths. For example, if N is 3 and the three registers are named A, B, and C, then the possible paths are: A—>A, A—>B, A—>C, B—>A, B—>B, B—>C, C—>A, C—>B, C—>C, for 3*3 = 9 possible paths. For N=50, there may be up to 2500 possible paths. Because of the large number of possible paths, automated tools can be of great assistance. ***Timing analysis*** tools automatically analyze all paths to determine the longest path, and may also ensure that setup and hold times are satisfied throughout the circuit.

5.6 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 harder to understand. A designer building a new digital system may find it useful to first describe the desired system behavior using a programming language, like C, C++, or Java, in order to capture desired behavior correctly. Alternatively, the designer may use the high-level programming constructs in a hardware description language, like the VHDL or Verilog languages, to first capture the desired behavior correctly. Then, the designer converts that programming language description to an RTL design by first converting the description to an HLSM RTL description, and then proceeding with RTL design. 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 5.11 Sum of absolute differences in C for video compression

Recall Example 5.8, which created a sum-of-absolute-differences component. That example started with an HLSM—but that HLSM 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 5.45.

That code is much easier to understand for most people than the HLSM in Figure 5.30. Thus, for some designs, C code (or something similar) is the most natural starting point.

To begin the RTL design process, a designer can convert the C code to an HLSM like that in Figure 5.30, and then proceed to complete the RTL design process and hence design the circuit.

```
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 5.45 C program description of a sum-of-absolute-differences computation—the C program may be easier to develop and easier to understand than an HLSM.

It is instructive to define a structured method for converting C code to an HLSM. Defining such a method makes it clear that C code can be *automatically translated* to either software on a programmable processor (such translation known as *compilation*), or *to a custom digital circuit* (such translation known as *synthesis*). 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 described below.

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 the method proceeds.

Consider three basic types of statements in C code—assignment statements, while loops, and condition statements (if-then-else). Equivalent HLSM templates exist for each such statement.

An assignment statement in C translates into one HLSM state. The state's actions execute the assignment as in Figure 5.46.

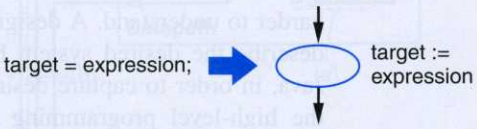


Figure 5.46 HLSM template for assignment statement.

An *if-then* statement in C translates into the HLSM structure of Figure 5.47. A state checks the condition of the *if* statement and has two transitions, one for the statement's condition being true, and the other for false. The true one points to the states for the *then* part of the statement. The false one points past those states to an end state.

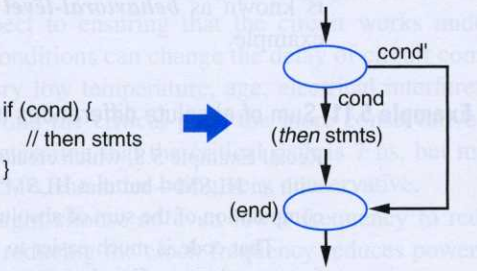


Figure 5.47 Template for *if-then* statement.

An *if-then-else* statement in C similarly translates to a state that checks the condition of the *if* statement, but this time pointing to states for the *else* part if the condition is false, as shown in Figure 5.48.

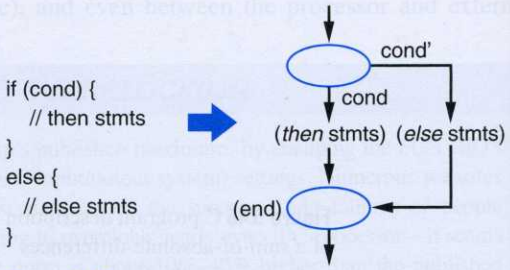


Figure 5.48 Template for *if-then-else* statement.

The *else* part commonly contains another *if* statement because C programmers may have multiple *else if* parts in a region of code. That if statement is translated as described earlier.

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 state rather than to the end state. The template appears in Figure 5.49. Only when the condition is false is the end state reached.

Given these simple templates, a wide variety of C programs can be converted to HLSMs, and the RTL design process can then be used to convert those HLSMs to digital circuits.

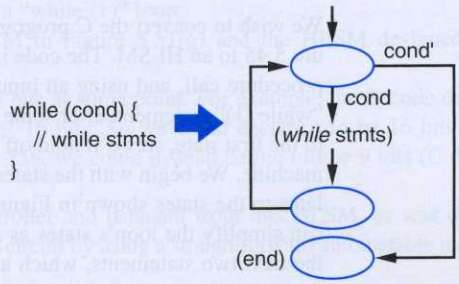


Figure 5.49 Template for *while* loop statement.

Example 5.12 Converting an if-then-else statement to a state machine

We are given the C-like code shown in Figure 5.50(a), which computes the maximum of two unsigned data inputs *X* and *Y*. We can translate that code to an HLSM by first translating the *if-then-else* statement to states using the method of Figure 5.48, as shown in Figure 5.50(b). We then translate the *then* statements to states, and then the *else* statements, yielding the final state machine in Figure 5.50(c).

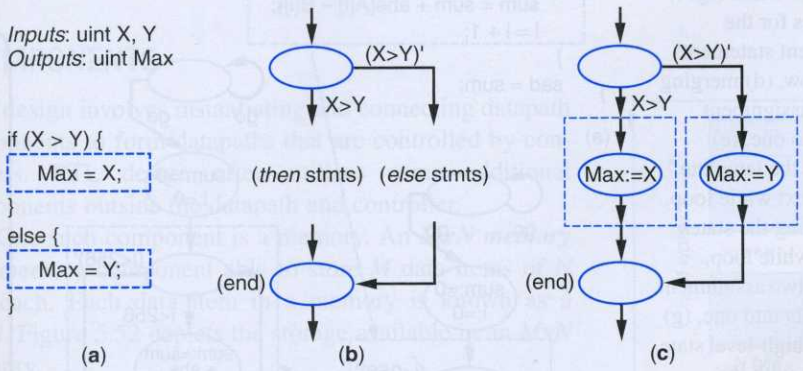


Figure 5.50 Behavioral-level design starting from C code: (a) C code for computing the max of two numbers, (b) translating the if-then-else statement to a high-level state machine, (c) translating the *then* and *else* statements to states. From the state machine in (c), we could use our RTL design method to complete the design. Note: max can be implemented more efficiently; we use max here to provide an easy-to-understand example.

Example 5.13 SAD C code to high-level state machine conversion

We wish to convert the C program description of the sum-of-absolute-differences behavior in Figure 5.45 to an HLSM. The code is shown in Figure 5.51(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 5.51(b). Since the loop has no statements in the loop body, we can simplify the loop's states as shown in Figure 5.51(c). Figure 5.51(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 5.51(d). We then translate the next *while* loop, using the *while* loop template, to the states shown in Figure 5.51(e). We fill in the states for the *while* loop's statements in Figure 5.51(f), merging the two assignment statement states into one state since the assignments can be done simultaneously. Figure 5.51(f) also

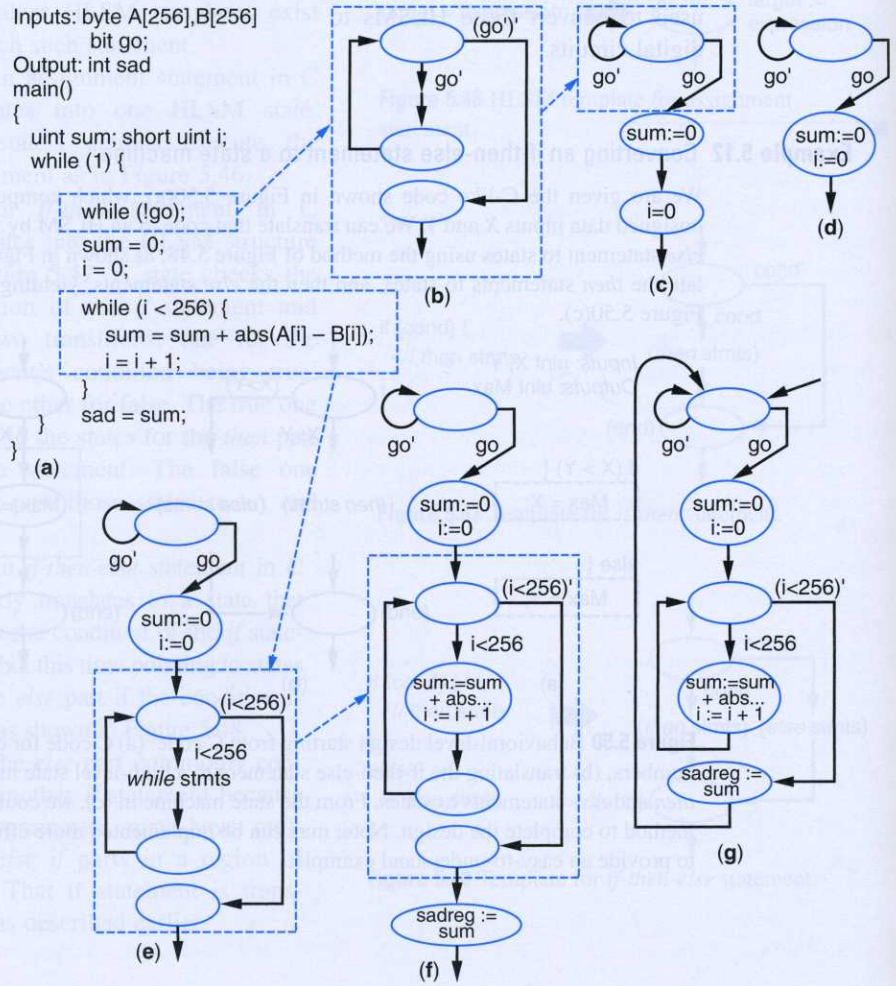


Figure 5.51 Behavioral-level design of the sum-of-absolute-differences code: (a) original C code, written as an infinite loop, (b) translating the statement "while (!go);" to a state machine, (c) simplified states for "while (!go);" and states for the assignment statements that follow, (d) merging the 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.

shows the state for the last statement of the C code, which assigns *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 the HLSM in Figure 5.51(g) and the HLSM designed from scratch in Figure 5.30.

We will need to map the C data types to bits at some point. For example, the C code declares *i* to be a short unsigned integer, which means 16 bits. So we could declare *i* to be 16 bits in the HLSM. Or, knowing the range of *i* to be 0 to 256, we could instead define *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 HLSM, as was done in Figure 5.31. Thus, we can translate C code to a circuit by using a straightforward automatable method.

The previous example shows 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.

5.7 MEMORY COMPONENTS

RTL design involves instantiating and connecting datapath components to form datapaths that are controlled by controllers. RTL design often utilizes some additional components outside the datapath and controller.

One such component is a memory. An ***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 ***word***. Figure 5.52 depicts the storage available in an *MxN* memory.

Memory can be categorized into two groups: RAM memory, which can be written to and read from, and ROM memory, which can only be read from. However, as shall be discussed, the distinction between the two categories is blurring due to new technologies.

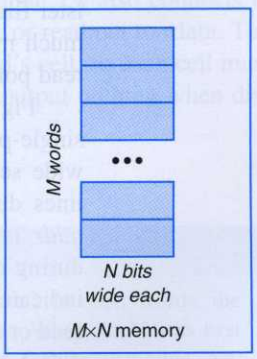


Figure 5.52 Logical view of a memory.

Random Access Memory (RAM)

A random-access memory (RAM) is logically the same as a register file (see Section 4.10)—both are memory components 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 that will be described below, rather than using a faster but larger 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 from the RAM, and then selecting the appropriate word (column) out of that row.

There is 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, with that port having both read and write capability (one at a time). Some RAMs are dual-ported. In contrast, register files are almost never single-ported. Furthermore, RAMs with more ports are much less common than for 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 5.53 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 or write during a particular clock cycle, we set `en` to `0` to prevent a read or write (regardless of the value of `rw`).

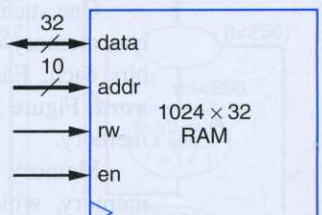


Figure 5.53 1024x32 RAM block symbol.

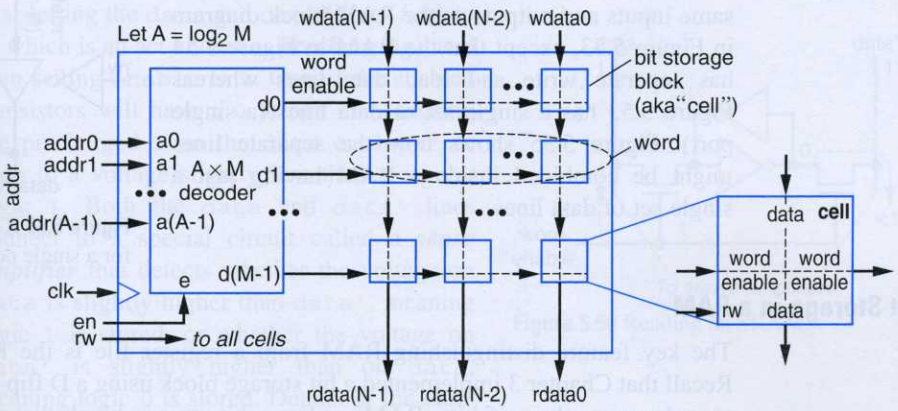


Figure 5.54 Logical internal structure of a RAM.

Figure 5.54 shows the logical internal structure of an *M*x*N* RAM. "Logical" structure means 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.

▶ 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.

Notice that the RAM in Figure 5.54 has the same inputs and outputs as the RAM block diagram in Figure 5.53, except that the RAM in Figure 5.54 has separate write and read data lines whereas Figure 5.53 has a single set of data lines (a single port). Figure 5.55 shows how the separate lines might be combined inside a RAM having just a single set of data lines.

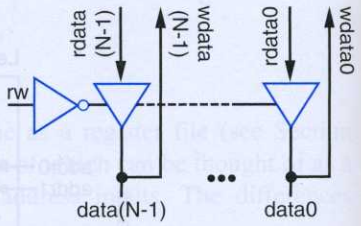


Figure 5.55 RAM data input/output for a single port.

Bit Storage in a RAM

The key feature distinguishing RAM from a register file is the RAM's compactness. Recall that Chapter 3 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, but slower, than a flip-flop. This section briefly describes 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 (SRAM) uses a bit storage block having two inverters connected in a loop as shown in Figure 5.56. A bit `d` will pass through the bottom inverter to become `d'`, then 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 5.54.

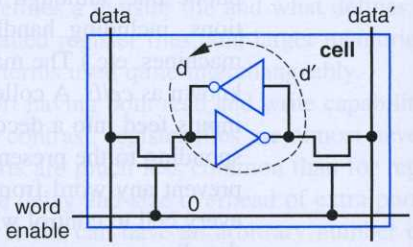


Figure 5.56 SRAM cell.

Writing a bit into this inverter loop is accomplished by setting the `data` line to the value of the desired bit, and `data'` to the complement. To store a `1`, the memory controller sets `data` to `1` and `data'` to `0` as in Figure 5.57. To store a `0`, the controller sets `data` to `0` and `data'` to `1`. The controller then sets `enable` to `1` so both shown transistors will conduct. The `data` and `data'` values will appear in the inverter loop as shown, overwriting any previous value.

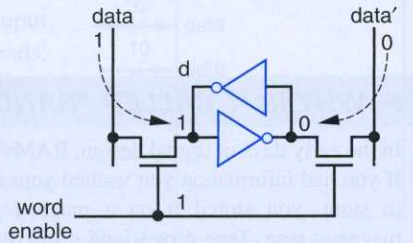


Figure 5.57 Writing a `1` to an SRAM cell.

Reading the stored bit can be done by first setting the `data` and `data'` lines *both* to `1`, which is an act known as ***precharging***), and then 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. Details of the electronics are beyond our scope.

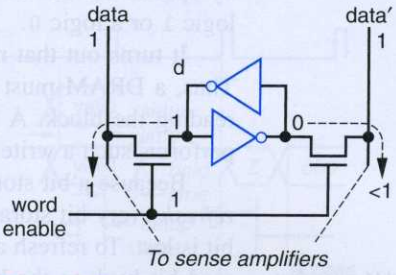


Figure 5.58 Reading an SRAM.

Notice that the bit storage block of Figure 5.58 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.

SRAM maintains the stored bit as long as power is supplied to the transistors. The stored bit (except when written) does *not change*—it is *static* (not changing).

Dynamic RAM

An alternative bit storage block used in RAM and popular for its compactness 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 5.59(a). The block is known as dynamic RAM (DRAM) because the stored bit *changes* as will be seen—the bit is *dynamic* (changing).

Writing can occur when `enable` is `1`. When enabled, `data=1` will charge the top plate of the capacitor to a `1`, while `data=0` will make the plate `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 and on to ground. Such discharging is the nature of a capacitor. However, the capacitor is intentionally designed to be relatively large, so that the discharge takes a long time, during which time the bit `d` is effectively considered as stored in the capacitor. Figure 5.59(b) provides a timing diagram illustrating the charge and discharge of the capacitor.

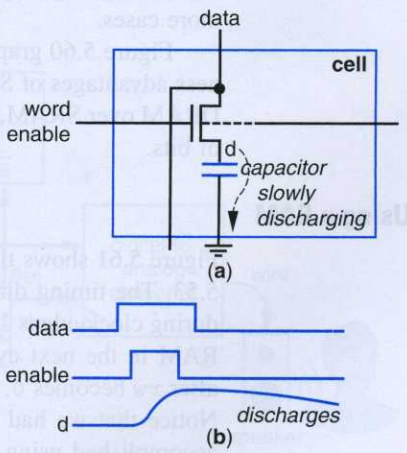


Figure 5.59 DRAM bit storage (a) bit storage block, (b) discharge.

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, a DRAM must immediately write the read bit back to the bit storage block after reading the block. A DRAM therefore contains 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.

DRAM chips first appeared in the early 1970s, and could hold only a few thousand bits. Modern DRAMs can hold many billions *of bits.*

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.

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 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 cases.

Figure 5.60 graphically depicts the compactness advantages of SRAM over register files, and DRAM over SRAM, for storing the *same* number of bits.

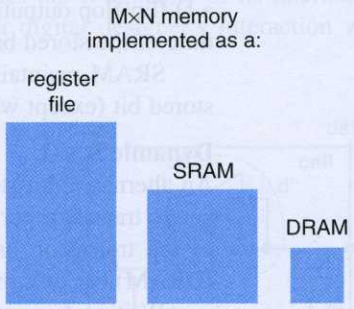


Figure 5.60 Depiction of compactness benefits of SRAM and DRAM (not to scale).

Using a RAM

Figure 5.61 shows timing diagrams describing how to write and read the RAM of Figure 5.53. The timing diagram shows how to write a 9 and a 13 into locations 500 and 999 during clock edges 1 and 2, respectively. The diagram shows how to read location 9 of the RAM in the next cycle, by setting *addr*=9, *data*=*Z*, and `rw=0` (meaning read). Shortly after `rw` becomes `0`, *data* becomes 500 (the value we had previously stored in location 9). Notice that we had to first disable the setting of *data* by setting it to *Z* (which can be accomplished using a three-state buffer) so as not to interfere with the data being read from the RAM. Also notice that this RAM's read functionality is asynchronous.

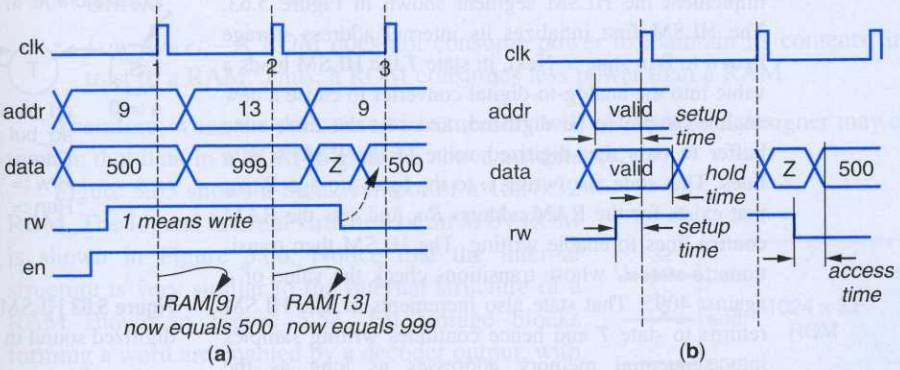


Figure 5.61 Reading and writing a RAM: (a) timing diagrams, (b) setup, hold, and access times.

The delay between our setting the `rw` line to read and the read data stabilizing at the *data* output is known as the RAM's ***access time*** or ***read time***.

The next example uses a RAM during RTL design.

Example 5.14 Digital sound recorder using a RAM

This example designs a system that can record sound digitally and that 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. An analog-to-digital converter is needed to digitize the sound, a RAM to store the digitized sound, a digital-to-analog converter to output the digitized sound, a three-state buffer to disable the data line going into the RAM, and a processor (to be designed) to control both converters and the RAM. Figure 5.62 shows a block diagram of the system.

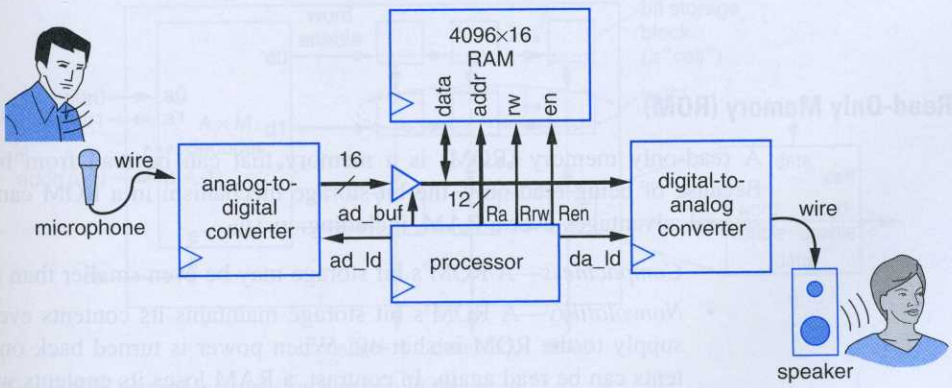


Figure 5.62 Utilizing a RAM in a digital sound recorder system.

To store digitized sound, the processor block can implement the HLSM segment shown in Figure 5.63. The HLSM first intializes its internal address storage item *a* to 0 in state *S*. Next, in state *T* the HLSM 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 writes *a* to the local storage *Rareg* that exists for the RAM address *Ra*, and sets the RAM control lines to enable writing. The HLSM then transitions to state *U* whose transitions check the value of *a* against 4095. That state also increments *a*. The HLSM returns to state *T* and hence continues writing samples into sequential memory addresses as long as the memory is not yet filled, meaning as long as $a < 4095$. Notice that the comparison is with 4095 rather than with 4096. This is because the action in state *U* of $a := a + 1$ does not cause an update until the next clock edge, so the comparison $a < 4095$ on state *U*'s outgoing transition uses the old value of *a*, not the incremented value (see Section 5.4 for further discussion).

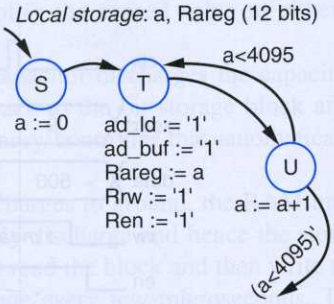


Figure 5.63 HLSM for storing digitized sound in RAM.

To playback the stored digitized sound, the processor can implement the HLSM segment shown in Figure 5.64. After initializing the local storage item *a* in state *V*, the HLSM 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. State *W* also sets the RAM address lines, and sets the RAM control lines to enable reading. The read data will thus appear on the *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 *a*. The HLSM returns to state *W* to continue reading, until the entire memory has been read.

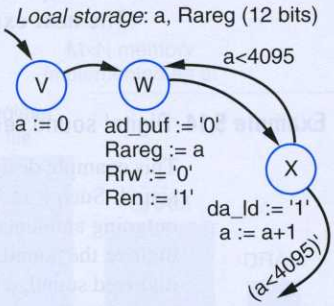


Figure 5.64 HLSM for playing sound from RAM.

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 that of a RAM.
- *Nonvolatility*—A ROM's bit storage maintains its contents even after the power supply to the ROM is shut off. When power is 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 compared to 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, a designer may choose to store that data in a ROM to gain the above advantages.

Figure 5.65 shows a block symbol of a 1024x32 ROM. The logical internal structure of an *M*x*N* ROM is shown in Figure 5.66. Notice that the internal structure is very similar to the internal structure of a RAM shown in Figure 5.54. 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 an `rw` input control to specify read versus write, nor for *wdata* 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).

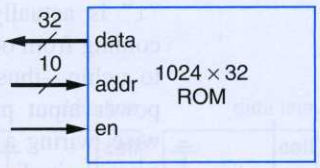


Figure 5.65 1024x32 ROM block symbol.

Some readers might at this point be wondering how a designer can write the initial contents of a ROM that will later be read. After all, if a designer can't write the contents of a ROM at all, then the ROM is really of no use. 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.

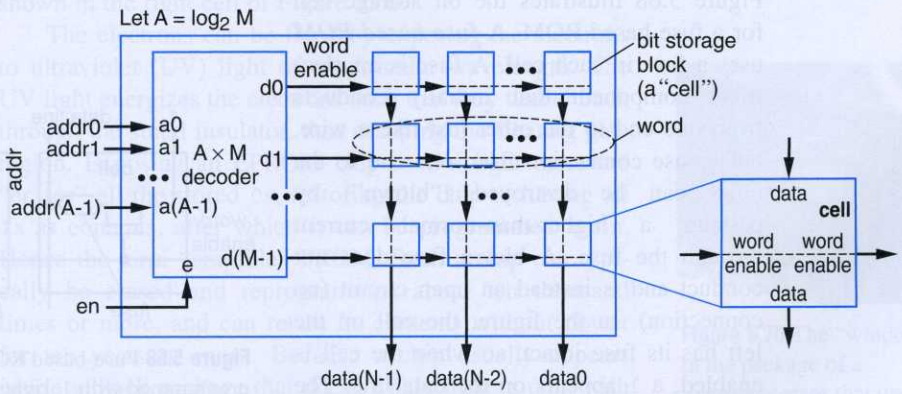


Figure 5.66 Logical internal structure of a ROM.

ROM Types

Mask-programmed ROM

Figure 5.67 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* `1`s to cells that should store a `1`, and `0`s 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 5.67 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.

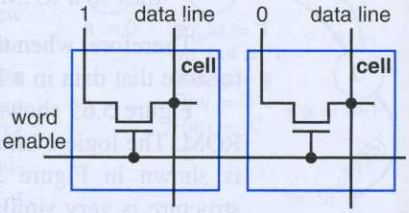


Figure 5.67 Mask-programmed ROM cells: left cell programmed with 1, right cell with 0.

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 5.68 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`).

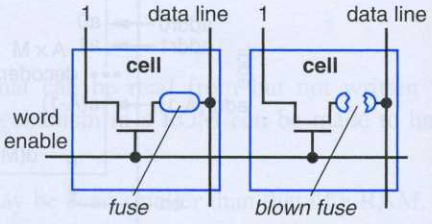


Figure 5.68 Fuse-based ROM cells: left cell programmed with 1, right cell with 0.

A fuse-based ROM is manufactured with all fuses intact, so the initially stored contents are all `1`s. 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 `0`s. 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 5.69 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`.

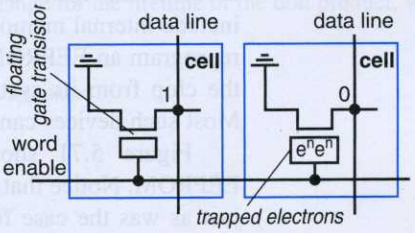


Figure 5.69 EPROM cells: left cell programmed with `1`, right cell with `0`.

An EPROM cell initially has no electrons trapped in any floating gate transistors, so the initially stored contents are all `1`s. A programmer device applies higher-than-normal voltages to those transistors in cells that should store `0`s. 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 5.69.

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 `0`s, restoring the chip to having all `1`s as contents, after which it 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 5.70.

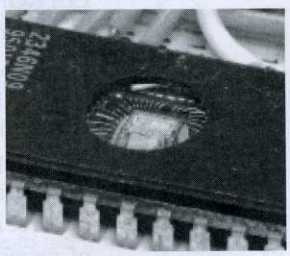


Figure 5.70 The "window" in the package of a microprocessor that uses an EPROM to store programs.

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, thus avoiding 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, but 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 device's contents (or part of its contents) without ever removing the chip from its system—such a device 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 5.71 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`. `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, a circuit that writes to the EEPROM should not try writing to a different word; that write will likely be ignored. Most EEPROMs will load the data to be programmed and the address into internal registers, freeing the circuit that is writing the EEPROM from having to hold these values constant during programming.

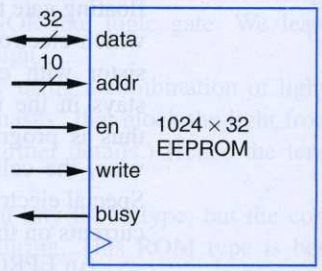


Figure 5.71 1024x32 EEPROM block symbol.

Modern EEPROMs can be programmed hundreds of thousands to millions of times or more, and can retain their contents for several decades to one hundred years or more without power.

While erasing one word at a time is fine for some applications that utilize EEPROM, other applications need to erase large blocks of memory quickly—for example, a digital camera application would need to erase a block of memory corresponding to an entire picture. ***Flash memory*** is a type of EEPROM in which all the words within a large block of memory can be erased very quickly, typically simultaneously, rather than one word at a time. A flash memory may be completely erased by setting an `erase` control input to `1`. Many flash memories also allow only a specific region, known as a *block* or *sector*, to be erased while other regions are left untouched.

Using a ROM

Below are examples of using a ROM during RTL design.

Example 5.15 Talking doll using a ROM

We wish to design a doll that speaks the message "Nice to meet you" whenever the doll's right arm is moved. A block diagram of the system is shown in Figure 5.72. A vibration sensor in the doll's right arm has an output v that is 1 when vibration is sensed. A processor detects the vibration and should then output a digitized version of the "Nice to meet you" message to a digital-to-analog converter attached to a speaker. The "Nice to meet you" message will be the prerecorded voice of a professional actress. Because that message will not change for the lifetime of the doll product, we can store that message in a ROM.

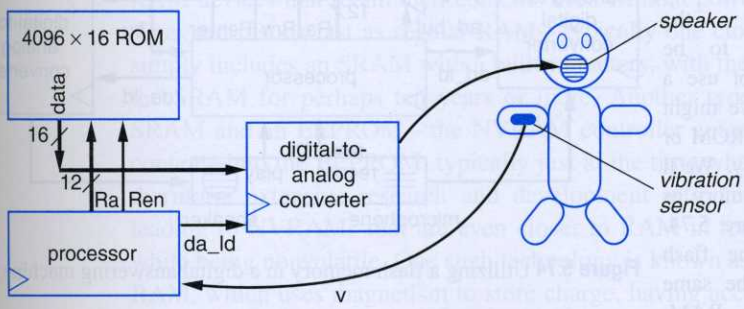


Figure 5.72 Utilizing a ROM in a talking doll system.

Figure 5.73 shows an HLSM segment that plays the message after detecting vibration. The machine starts in state *S*, initializing the ROM address counter *a* to *0*, and waiting for vibration to be sensed. When vibration is sensed, the machine proceeds to state *T*, which reads the current ROM location. The machine moves on to state *U*, which loads the digital-to-analog converter with the read value from ROM, increments *a*, and proceeds back to *T* as long as *a* hasn't reached 4095 (remember that the transition from *U* uses the value of *a* before the increment, so should compare to 4095, not to 4096).

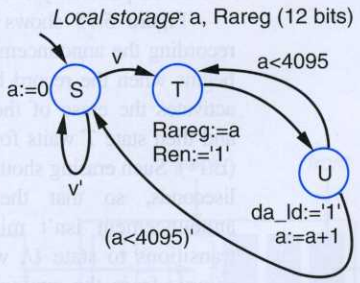


Figure 5.73 HLSM for reading the ROM.

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 5.16 Digital telephone answering machine using a flash memory

This example designs 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 thus 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 5.74. Notice that the flash memory has the same interface as a RAM, except that the flash memory has an extra input named `erase`, which 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.

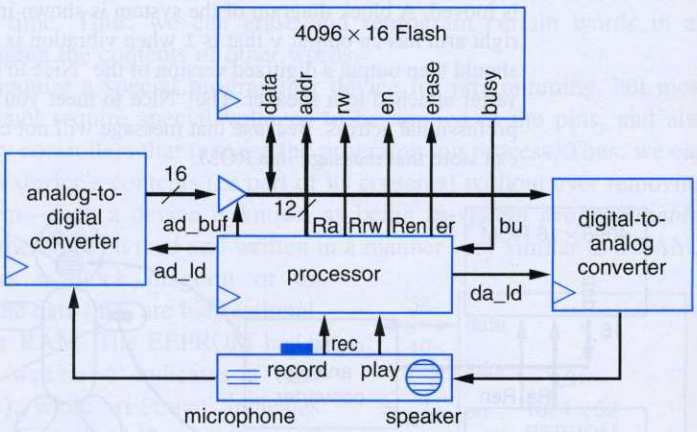


Figure 5.74 Utilizing a flash memory in a digital answering machine.

Figure 5.75 shows an HLSM segment for recording the announcement. The HLSM 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'`). Such erasing should occur in just a few milliseconds, so that the start of the spoken announcement isn't missed. The HLSM then transitions to state *U*, which copies a digitized sample from the analog-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.

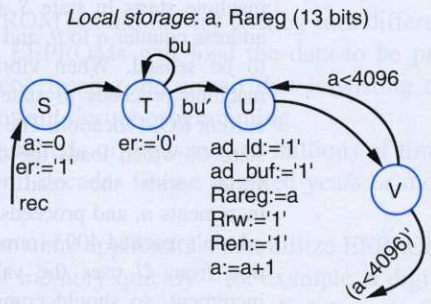


Figure 5.75 HLSM for storing digitized sound in a flash memory.

Notice that, unlike previous examples, this HLSM 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 earlier examples may be slightly better because they require 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 HLSM 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.

5.8 QUEUES (FIFOs)

Sometimes a designer's data storage needs specifically require reading items in the same order that the items were written. 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 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 5.76. 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

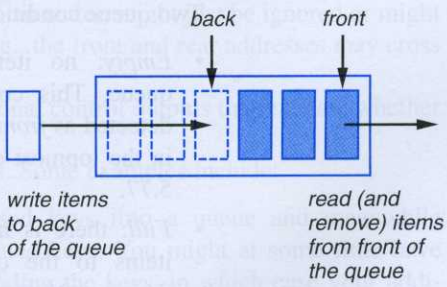


Figure 5.76 Conceptual view of a queue.

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***.

A queue can be implemented 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 5.77. The figure shows an initially empty eight-word queue with front and rear both set to memory address 0. The first sample 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 sample 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.

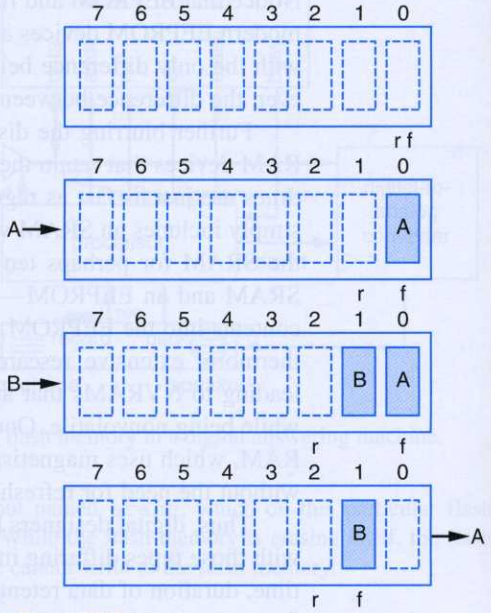


Figure 5.77 Writing and reading a queue implemented in a memory causes the front (*f*) and rear (*r*) to move.

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 5.78.

Two queue conditions of interest:

- *Empty*: no items are in the queue. This condition can be detected as *front* = *rear*, as seen in the topmost queue of Figure 5.77.
- *Full*: there is no 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 *front* = *rear*.

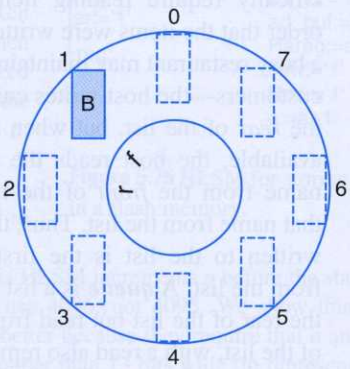


Figure 5.78 Implementing a queue in a memory treats the memory as a circle.

Unfortunately, notice that the conditions for 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.

An 8-word queue can be implemented using an 8-word two-port register file and additional components, as depicted in Figure 5.79. 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.

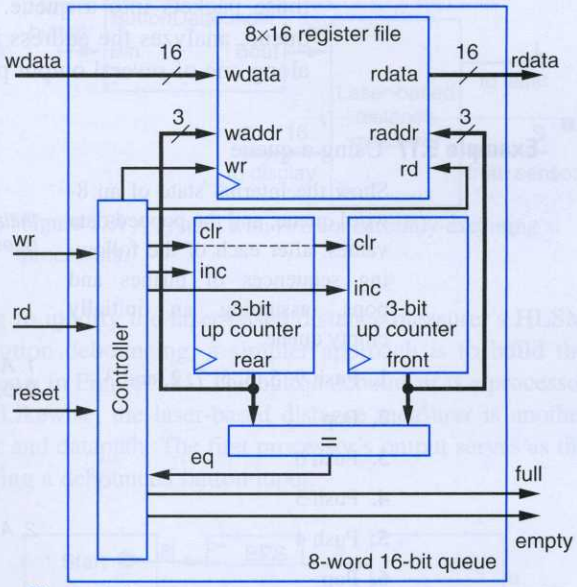


Figure 5.79 Architecture of an 8-word 16-bit queue.

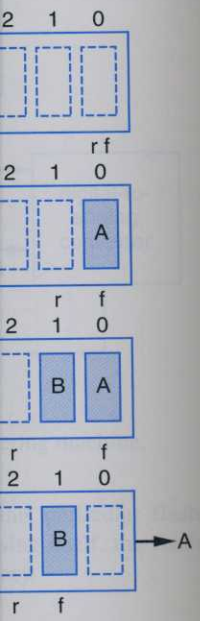
A circuit that uses a queue should never read an empty queue or write a full queue—depending on the controller design, such an action might just be ignored or might put the queue into a misleading internal state (e.g., the front and rear addresses may cross over).

Many 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.

y in everyday lan-
eaking countries).
the list, a queue is
are called ***FIFO***
n first-in first-out.
is also sometimes
read is sometimes



ng a queue
uses the front (*f*)

6 **Figure 5.78**
Implementing a queue in a memory treats the memory as a circle.

- A computer printer may store print jobs in a queue while those jobs are waiting to be printed.
- A modem stores incoming data in a queue and requests a computer to read that data. Likewise, the modem writes outgoing data received from the computer into a queue and then sends that data out over the modem's outgoing medium.
- A computer network router receives data packets from an input port and writes those packets into a queue. Meanwhile, the router reads the packets from the queue, analyzes the address information in the packet, and then sends the packet along one of several output ports.

Example 5.17 Using a queue

Show the internal state of an 8-word queue, and the 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

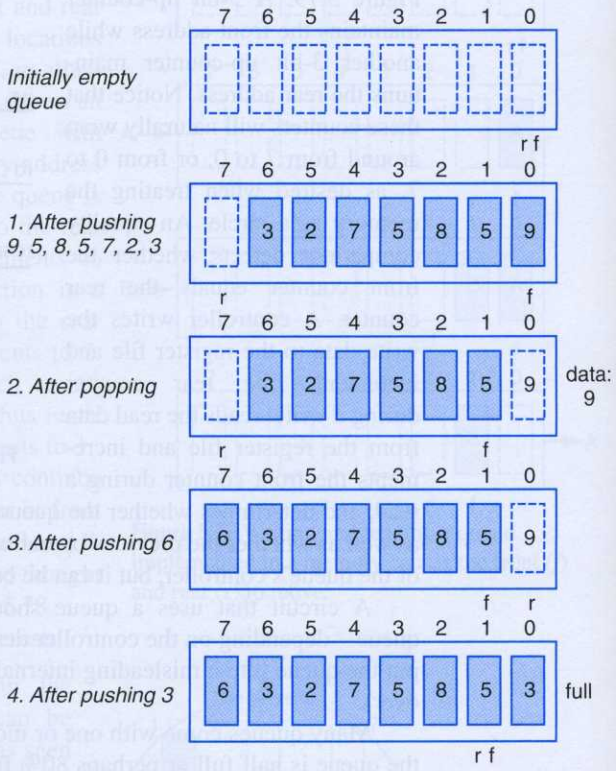


Figure 5.80 Example pushes and pops of a queue.

Figure 5.80 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.

▶ 5.9 MULTIPLE PROCESSORS

RTL design can be aided by capturing behavior using multiple concurrently-executing HLSMs and converting those HLSMs to multiple processors. For example, consider the laser-based distance measurer from Example 5.13. The system has an input `B` from a button. Section 5.4 explained that button inputs may need to be debounced. Rather than attempting to modify the laser-based distance measurer's HLSM of Figure 5.12 to also perform button debouncing, a simpler approach is to build the system using two processors as shown in Figure 5.81. The button debouncer is a processor having a controller and datapath. Likewise, the laser-based distance measurer is another processor having its own controller and datapath. The first processor's output serves as the second processor's input, representing a debounced button input.

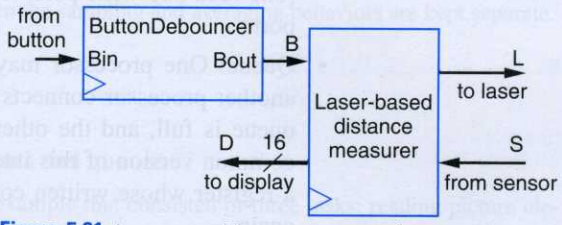


Figure 5.81 A system with two concurrently-executing processors.

As another example, recall the code detector of Example 3.6. The example assumed that the inputs from the buttons were each synchronized to the clock such that each unique press would result in the corresponding signal being `1` for exactly one clock cycle. Such synchronization was achieved by designing a button press synchronizer in Example 3.9. The system could then be built as shown in Figure 5.82, with a button press synchronizer (BPS) processor instantiated for each input. The system thus consists of 5 BPS processors and one code detector processor (each of those processors happen to consist of just a controller and no datapath, but each is a processor nevertheless).

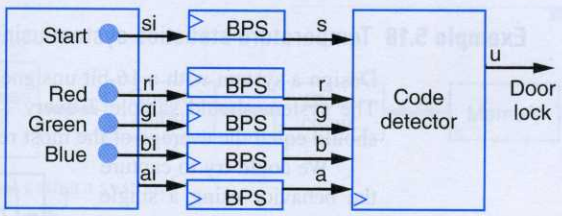


Figure 5.82 A system with multiple concurrently-executing processors.

Many different interfaces between processors are possible through the use of global items. A signal, register, or other component that is not inside a processor is called a ***global*** signal, a global register, or global component, respectively. Possible interfaces between processors include:

- Control signal: The above examples represent the simplest interface between processors involving one processor writing a control output that another processor reads. The processors are said to share a global control signal.

- Data signal: Another interface involves a processor writing a data output that the other processor reads. The processors share a global data signal.
- Register: One processor may write to a global register that another processor reads.
- Register file: One processor may connect to the write port of a global two-port (one-read one-write port) register file while another processor connects to the read port.
- Queue: One processor may connect to the write lines of a global queue, while another processor connects to the read lines. A processor should not write if the queue is full, and the other processor should not read if the queue is empty. A common version of this interface uses a one-word queue, effectively representing a register whose written contents must be read before the register can be written again.

Even though the multiple processors (and global components) execute concurrently, they all use the same clock signal, and thus the processors are ***synchronized***. Such synchronization avoids metastability issues in the interfaces between the processors (unsynchronized processors are also possible but are not considered in this book).

Note that if a multiple processor system has an unregistered output of one processor connecting to an input of another processor, the system's critical path could be from a register of one processor to a register of another processor.

Example 5.18 Temperature statistics system using multiple processors

Design a system with a 16-bit unsigned input *T* from a temperature sensor, and a 16-bit output *A*. The system should sample *T* every 1 second. Output *A* should be computed every minute and should equal the average of the most recent 64 samples.

We could try to capture the behavior using a single HLSM, but that approach may lead to a complicated HLSM. Instead, we'll use two HLSMs that will be interfaced using a one-write-port one-read-port 64x16 register file. The first HLSM, *Tsample*, will write the sampled value into successive register file addresses. The second HLSM, *Avg*, will compute the average of the register file's contents. Figure 5.83 shows a block diagram of the system.

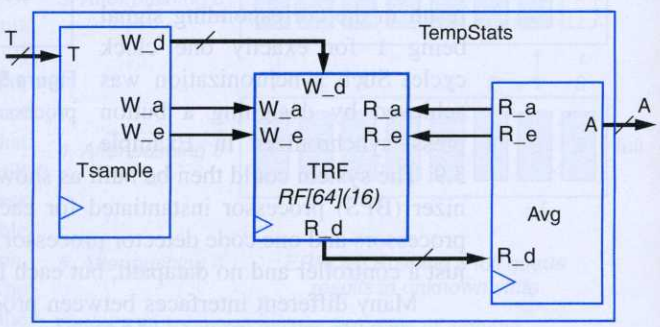


Figure 5.83 A temperature statistics system.

The procedure of defining the HLSMs would follow from previous examples and is not shown. There would be three top-level objects: a 64x16 register file *TRF*, an HLSM *Tsample*, and an HLSM *Avg*. HLSM *Tsample* would declare a timer and a 6-bit local storage item *addr*, initialize the timer with 1 second, and then write input *T* to *TRF*[*addr*] every 1 second and increment *addr* (note that *addr* would wrap around from 63 to 0). HLSM *Avg* would declare a timer and a local storage

item *sum*, initialize the timer with 60 seconds, and then compute the average every 60 seconds. Computing the average would be achieved by reading each word 0 to 63 of the register file TRF one at a time, adding each to *sum*. To avoid overflow, *sum* might be made 32 bits wide. Then, the HLSM would divide the sum by 64 to compute the average, and would output the result on *A*. Recall that a one-read-port one-write-port register file can support both a read and a write operation in the same clock cycle, which is what allows the two HLSMs to access the register file independently.

Notice how simple the design is when the sampling and averaging behaviors are kept separate.

Example 5.19 Digital camera with multiple processors and queues

Section 1.3 introduced a digital camera example that consisted of three tasks: reading picture elements from an image sensor, compressing those elements, and storing the results into a memory. Figure 5.84 shows processors for the three tasks along with interfaces between them. Consider the interface between the *Read* and *Compress* processors. Assume *Compress* sometimes runs faster and sometimes runs slower depending on what item is being compressed. A queue is a good interface for such a situation. In this case, an 8-bit 8-word queue is used. When *Read* runs faster than *Compress*, *Read* can push more items onto the queue (until the queue is full, at which point *Read* must wait until the queue is not full). When *Compress* runs faster than *Read*, *Compress* can pop more items (until the queue is empty, at which point *Compress* must wait for the queue to be not empty). The queue thus improves overall performance, while also ensuring that items are accessed by *Compress* before being overwritten by *Read*. Similarly, a queue exists between *Compress* and *Store*.

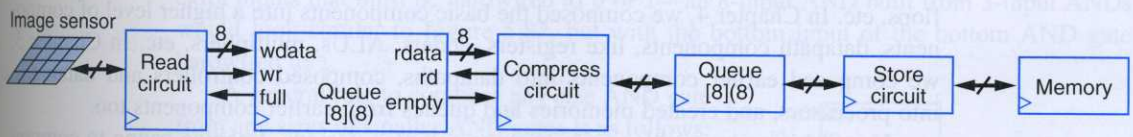


Figure 5.84 Three processor circuits and two queues in a digital camera system.

5.10 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 example of a hierarchy is 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 three-level hierarchy involving a country, provinces, and cities is shown in Figure 5.85.

Figure 5.86 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 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).

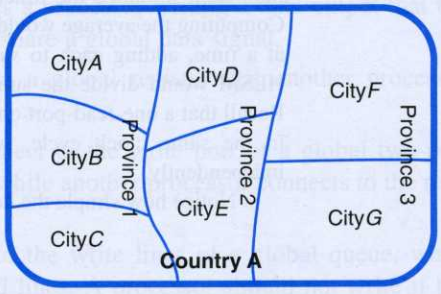


Figure 5.85 Three-level hierarchy example: a country, made up of provinces, each made up of cities.

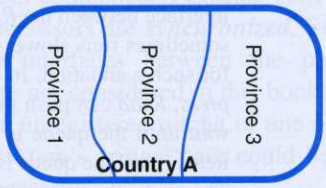


Figure 5.86 Hierarchy showing just the top two levels.

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 composed earlier components into datapaths, composed controllers and datapaths, into processors, and created memories and queues from earlier components too.

Use of hierarchy enables us to manage complex designs. Imagine trying to comprehend the design of Figure 5.40 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 *compose* a larger version of a component from smaller versions of the same component. For example, suppose 3-input AND gates are available in a library, but a 9-input AND gate is needed in a design. A designer can compose several 3-input AND gates to form a 9-input AND gate as shown in Figure 5.87. A designer 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 5.87, but with the bottom input of the bottom AND gate hardwired to `1`.

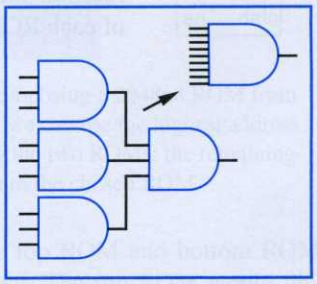


Figure 5.87 Composing a 9-input AND gate from 3-input AND gates.

A general rule to compose any size AND gate from any sizes of smaller AND gates is as follows: fill the first level with (the largest available) AND gates until the sum of the number of inputs equals the desired number of inputs, then fill the second level similarly (feeding first-level outputs to the second-level gates), until the last level has just one gate. 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 4x1 and 2x1 muxes were available in a library, but an 8x1 mux was needed. A designer could compose the smaller muxes into an 8x1 mux as shown in Figure 5.88. Notice that `s2` selects among group `i0-i3` and `i4-i7`, while `s1` and `s0` select one input from the group. The select line

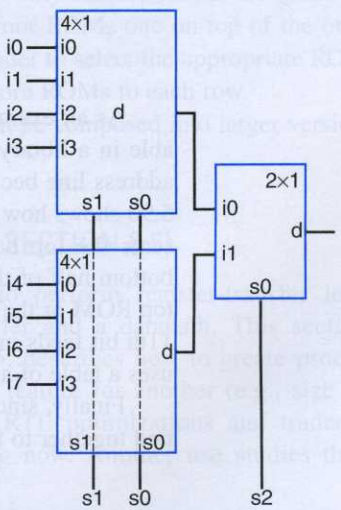


Figure 5.88 An 8x1 mux composed from 4x1 and 2x1 muxes.

values pass the appropriate input through to the output. For example, `s2s1s0=000` passes `i0` through, `s2s1s0=100` passes `i4`, and `s2s1s0=111` passes `i7`.

A common 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.

As an example of needing wider words, suppose 1024x8 ROMs are available in a library, but a 1024x32 ROM is needed. Composing the smaller ROMs into the larger one is straightforward, as shown in Figure 5.89. Four 1024x8 ROMs are needed to obtain 32 bits per word. The 10 address inputs are connected to all four ROMs. Likewise, the enable input is connected to all four ROMs. The four 8-bit outputs are grouped into the desired 32-bit output. Thus, each ROM stores one byte of the 32-bit word. Reading a location such as location 99 results in four simultaneous reads of the byte at location 99 of each ROM.

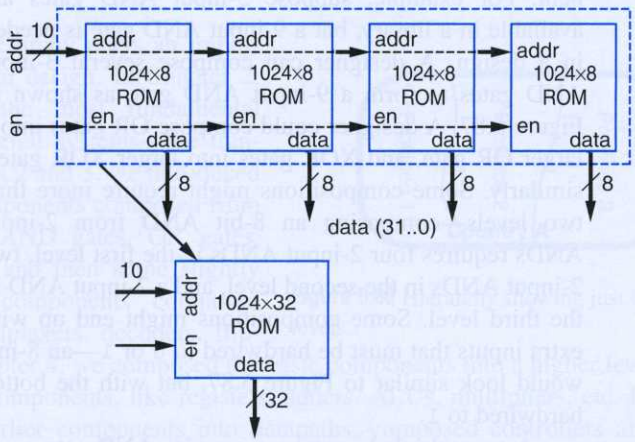


Figure 5.89 Composing a 1024x32 ROM from 1024x8 ROMs.

As an example of needing more words, suppose again that 1024x8 ROMs are available in a library, but this time a 2048x8 ROM is needed. The larger ROM has an extra address line because it has twice as many words to address as the smaller ROMs. Figure 5.90 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 represents the bottom half of the memory (1024 words). The 11th address line (`a10`) enables 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 5.91 uses a table of addresses to show how the 11th bit selects among the two smaller ROMs.

Finally, since only one ROM will be active at any time, the output data lines can be tied together to form the 8-bit output, as shown in Figure 5.90.

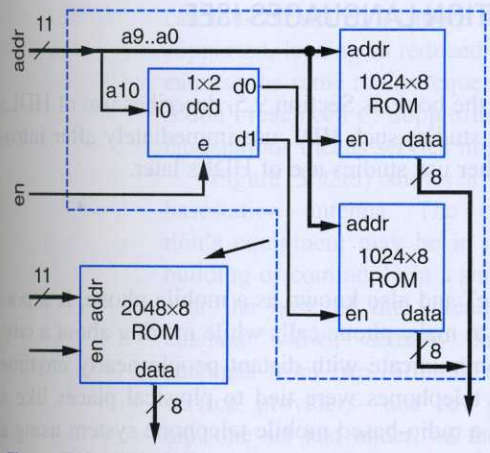


Figure 5.90 Composing a 2048x8 ROM from 1024x8 ROMs.

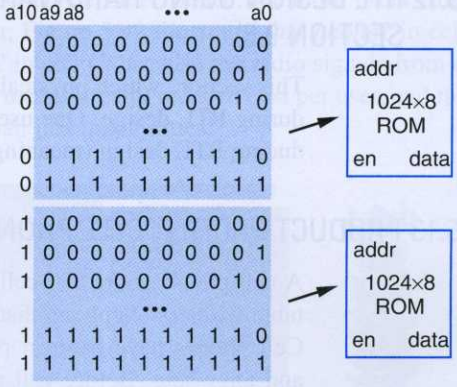


Figure 5.91 When composing a 2048x8 ROM from two 1024x8 ROMs, we can use the highest address bit to choose among the two ROMs; the remaining address bits offset into the chosen ROM.

Note that any bit could be used to select between the top ROM and bottom ROM. Designers sometimes use the lowest-order bit (`a0`) to select. The top ROM would thus represent all even addresses, and the bottom ROM would represent all odd addresses.

If the desired ROM is four times larger than the available ROM, then two address lines would select from four ROMs via a 2x4 decoder. If the desired ROM is eight times larger, then three address lines and a 3x8 decoder would be used. Other sizes follow similarly.

The approaches for creating a ROM with wider words and with more words can be used together. Suppose a 4096x32 ROM is needed, but only 1024x8 ROMs are available. A designer can first create a 4096x8 ROM by using four ROMs one on top of the other and by feeding the top two address lines to a 2x4 decoder to select the appropriate ROM. The designer can then widen the ROM by adding 3 more ROMs to each row.

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

▶ 5.11 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 studies such RTL optimizations and tradeoffs immediately after introducing RTL design, meaning now. Another use studies them later.

▶ 5.12 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 studies such HDL use immediately after introducing RTL design (meaning now). Another use studies use of HDLs later.

▶ 5.13 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 5.92 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

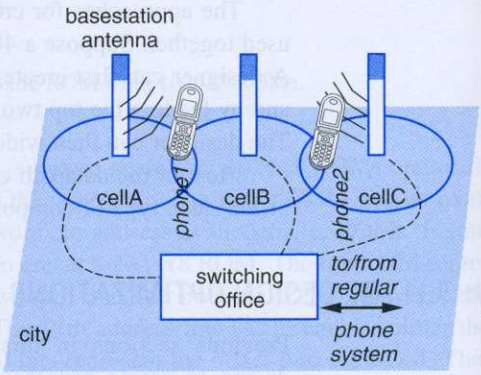


Figure 5.92 *Phone1* in cell *A* can use the same radio frequency as *phone2* in cell *C*, increasing the number of possible mobile phone users in a city.

can thus be shared by more than one phone at one time. Hence, far more users can be supported, leading to reduced costs per user. Figure 5.92 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 5.93(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 the 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 5.93(b)). Some providers try to disguise their antennas to make them more soothing to the eye, as in Figure 5.93(c)—the entire tree in the picture is artificial.

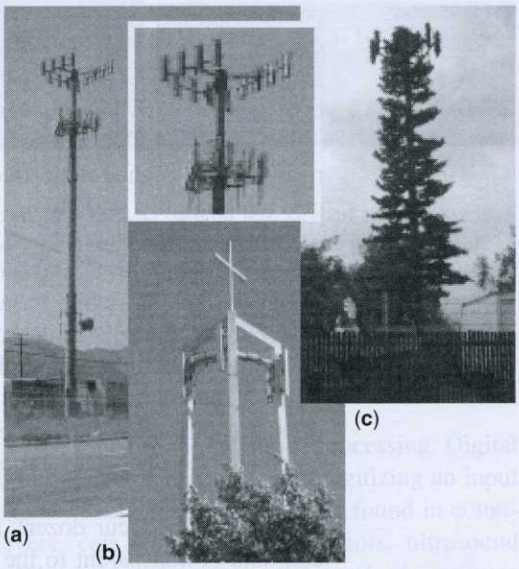


Figure 5.93 Basestations found in various locations.

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 "landline" 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 5.91. 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 5.92. 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 5.94 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 of a voice (or other sound) to a signal stream of `0`s and `1`s, 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

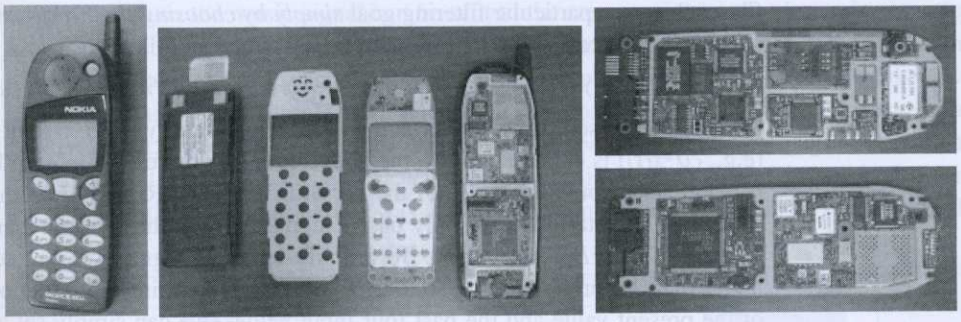


Figure 5.94 Inside a cell phone: (a) handset, (b) battery and ID card on left, keypad and display in center, digital circuitry on a printed-circuit board on right, (c) the two sides of the printed-circuit board, showing several digital chip packages mounted on the board.

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 on to 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 add that

result to the next-earlier 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:

$$y(t) = c0 \times x(t) + c1 \times x(t-1) + c2 \times x(t-2) + c3 \times x(t-3) + c4 \times x(t-4) + \ldots$$

t is the present time step. x is the input signal, and y is the output signal. Each term (e.g., *c0*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 *c0* to 1, and we set *c1=c2=c3=c4*=0. To amplify an input signal, we can set *c0* to a number larger than 1, perhaps setting *c0* 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, *c1=c2=c3=c4=c5*=0.2. The results of such a filter applied to a noisy input signal are shown in Figure 5.95. To smooth and amplify, we can set all constants to equivalent values that add to something greater than 1, for example, *c1=c2=c3=c4=c5*=1, resulting in 5x amplification. To create a smoothing filter that only includes the previous two rather than four input values, we simply set *c3* and *c4* 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.

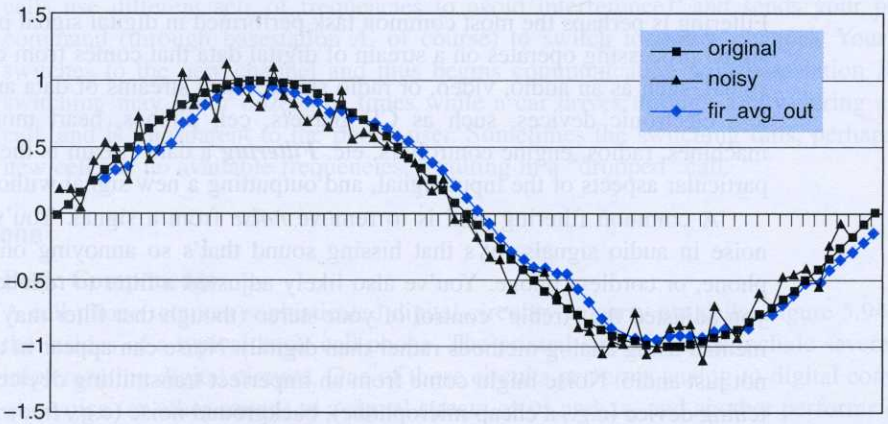


Figure 5.95 Results of a 5-tap FIR filter with *c0=c1=c2=c3=c4*=0.2 applied to a noisy signal. The original signal is a sine wave. The noisy signal has random jumps . The FIR output (fir_avg_out) is much smoother than the noisy signal, approaching the original signal. Notice that the FIR output is slightly 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 5.96 shows a main signal, *in1*, that we want to transmit. We can add that to a carrier signal, *in2*, 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 *in1* being the signal of the music, and *in2* 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 5.96) 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 negative point of the same magnitude. For a negative point, three samples earlier was a positive point of the same magnitude. And for a zero point, three samples earlier was also a zero point. Likewise, adding a carrier signal

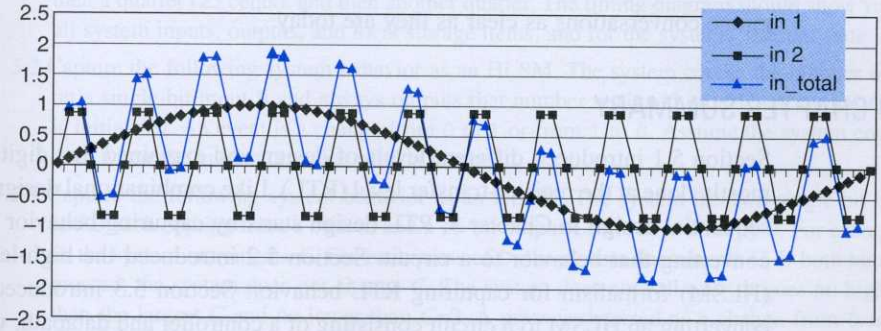


Figure 5.96 Adding a main signal *in1* to a carrier signal *in2*, resulting in a composite signal *in_total*.

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 5.97. 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.

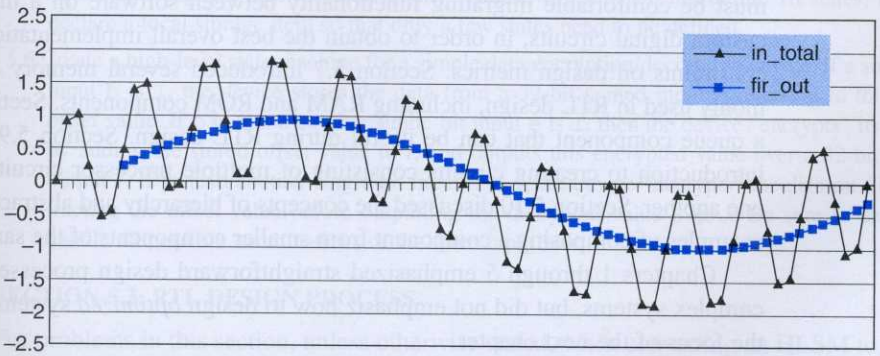


Figure 5.97 Filtering out the carrier signal using a 7-tap FIR filter with constants 0.25, 0, 0, 0.5, 0, 0, 0.25. The slight delay in the output signal typically poses no problem.

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 5.10 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.

5.14 CHAPTER SUMMARY

Section 5.1 introduced different levels of design and explained that digital design today is mostly done at the register-transfer level (RTL). Like combinational design in Chapter 2 and sequential design in Chapter 3, RTL design starts by capturing behavior and then involves converting that behavior to a circuit. Section 5.2 introduced the high-level state machine (HLSM) formalism for capturing RTL behavior. Section 5.3 introduced a procedure for converting an HLSM to a circuit consisting of a controller and datapath, which together are known as a processor. The datapaths used components defined in Chapter 4 capable of executing the HLSM's data operations, and the controller was built using methods from Chapter 3. Section 5.4 showed how RTL design could make use of register files and of timers, and also showed a data-oriented example. Section 5.5 showed how to set a circuit's clock frequency based on the circuit's critical path. Section 5.6 demonstrated how a sequential program like a C program can be converted to gates using straightforward transformations that convert the C program into RTL behavior. Such conversion makes 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 between microprocessor and custom circuit implementations relate to design metrics like system performance, power consumption, size, cost, and design time. Modern digital designers must 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. Section 5.7 introduced several memory components commonly used in RTL design, including RAM and ROM components. Section 5.8 introduced a queue component that can be useful during RTL design. Section 5.9 provided a basic introduction to creating circuits consisting of multiple processor circuits interacting with one another. Section 5.10 discussed the concepts of hierarchy and abstraction, and provided examples of composing a component from smaller components of the same kind.

Chapters 1 through 5 emphasized straightforward design processes for increasingly complex systems, but did not emphasiz how to design *optimized* systems. Optimization is the focus of the next chapter.