CHAPTER 2: *Custom Single-Purpose Processors: Hardware*

2.1 Introduction

A *processor* is a digital circuit designed to perform computation tasks. A processor consists of a datapath capable of storing and manipulating data and a controller capable of moving data through the datapath. A general-purpose processor is designed such that it can carry out a wide variety of computation tasks, which are described by a set of programmer-provided instructions. In contrast, a single-purpose processor is designed specifically to carry out a particular computation task. While some tasks are so common that we can purchase standard single-purpose processors to implement those tasks, others are unique to a particular embedded system. Such custom tasks may be best implemented using custom single-purpose processors that we design ourselves.

An embedded system designer may obtain several benefits by choosing to use a custom single-purpose processor rather than a general-purpose processor to implement a computation task.

First, performance may be faster, due to fewer clock cycles resulting from a customized datapath, and due to shorter clock cycles resulting from simpler functional units, fewer multiplexors, or simpler controller logic. Second, size may be smaller, due to a simpler

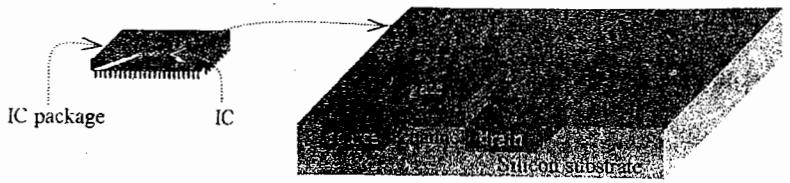


Figure 2.1: A simplified view of a CMOS transistor on silicon.

datapath and no program memory. Third, power consumption may be less, due to more efficient computation.

However, cost could be higher because of high NRE costs. Since we may not be able to afford to invest as much NRE cost as can designers of a mass-produced general-purpose processor, performance and size could actually be worse. Time-to-market may be longer, and flexibility reduced, compared to general-purpose processors.

In this chapter, we describe basic techniques for designing custom processors. We start with a review of combinational and sequential design, and we describe methods for converting programs to custom single-purpose processors.

2.2 Combinational Logic

Transistors and Logic Gates

A transistor is the basic electrical component in digital systems. Combinations of transistors form more abstract components called logic gates, which designers use when building digital systems. Thus, we begin with a short description of transistors before discussing logic design.

A transistor acts as a simple on/off switch. One type of transistor, complementary metal oxide semiconductor (CMOS), is shown in Figure 2.1. Figure 2.2(a) shows the schematic of a transistor. The *gate*, not to be confused with logic gate, controls whether or not current flows from the *source* to the *drain*. We can apply either low or high voltage levels to the gate. The high level may be, for example, +3 or +5 volts, which we'll refer to as logic 1. The low voltage is typically ground, drawn as several horizontal lines of decreasing width, which we'll refer to as logic 0. When logic 1 is applied to the gate, the transistor conducts and so current flows. When logic 0 is applied to the gate, the transistor does not conduct. We can also build a transistor with the opposite functionality, illustrated in Figure 2.2(b). When logic 0 is applied to the gate, the transistor conducts. When logic 1 is applied, the transistor does not conduct.

Given these two basic transistors, we can easily build a circuit whose output inverts its gate input, as shown in Figure 2.2(c). When the input x is logic 0, the top transistor conducts and the bottom transistor does not conduct, so logic 1 appears at the output F. We can also easily build a circuit whose output is logic 1 when at least one of its inputs is logic 0, as shown in Figure 2.2(d). When at least one of the inputs x and y is logic 0, then at least one of

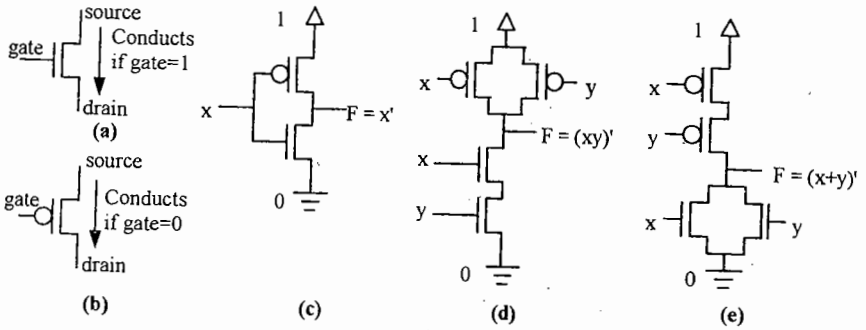


Figure 2.2: CMOS transistor implementations of some basic logic gates: (a) nMOS transistor, (b) pMOS transistor, (c) inverter, (d) NAND gate, (e) NOR gate.

the top transistors conducts and the bottom transistors do not conduct, so logic 1 appears at F. If both inputs are logic 1, then neither of the top transistors conducts, but both of the bottom transistors do, so logic 0 appears at F. Likewise, we can easily build a circuit whose output is logic 1 when both of its inputs are logic 0, as illustrated in Figure 2.2(e). The three circuits shown implement three basic logic gates: an inverter, a NAND gate, and a NOR gate.

Digital system designers usually work at the abstraction level of logic gates rather than transistors. Figure 2.3 describes eight basic logic gates. Each gate is represented symbolically, with a Boolean equation, and with a truth table. The truth table has inputs on the left and an output on the right. The AND gate outputs 1 if and only if both inputs are 1. The OR gate outputs 1 if and only if at least one of the inputs is 1. The XOR (exclusive-OR) gate outputs 1

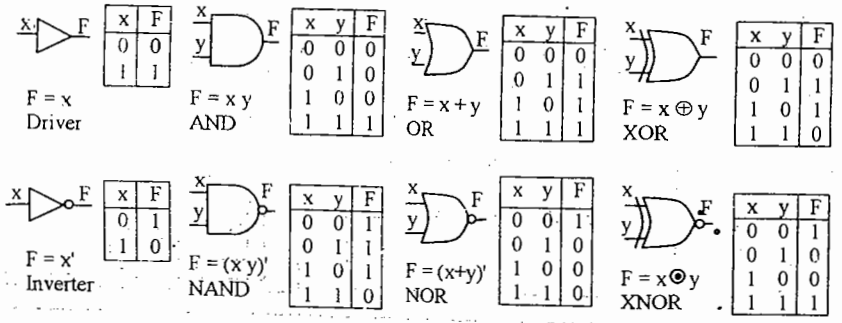


Figure 2.3: Basic logic gates.

if and only if exactly one of its two inputs is 1. The NAND, NOR, and XNOR gates output the complement of AND, OR, and XOR, respectively

Even though AND and OR gates are easier to comprehend logically, NAND and NOR gates are more commonly used, and those are the gates we built using transistors in Figure 2.2. The NAND could easily be changed to AND by changing the 1 on the top to 0 and the 0 on the bottom to 1; the NOR could be changed to OR similarly. But it turns out that pMOS transistors don't conduct 0s very well, though they do fine conducting 1s, for reasons beyond this book's scope. Likewise, nMOS transistors don't conduct 1s very well, though they do fine conducting 0s. Hence, NANDs and NORs prevail.

Basic Combinational Logic Design

A *combinational* circuit is a digital circuit whose output is purely a function of its present

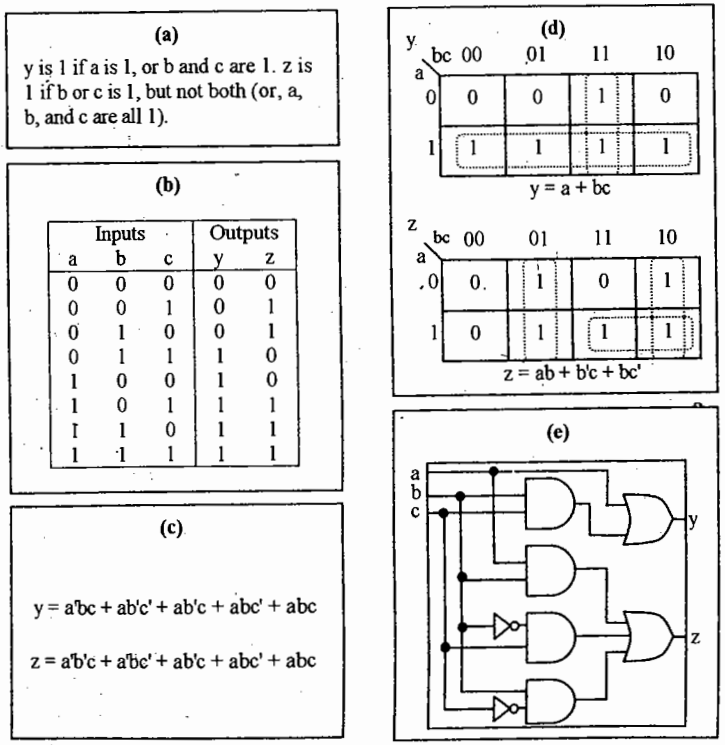


Figure 2.4: Combinational logic design.: (a) problem description, (b) truth table. (c) output equations, (d) minimized output equations, (e) final circuit.

inputs. Such a circuit has no memory of past inputs. We can use a simple technique to design a combinational circuit from our basic logic gates, as illustrated in Figure 2.4. We start with a problem description, which describes the outputs in terms of the inputs, as in Figure 2.4(a). We translate that description to a truth table, with all possible combinations of input values on the left and desired output values for each combination on the right, as in Figure 2.4(b). For each output column, we can derive an output equation, with one equation term per row, as in Figure 2.4(c). We can then translate these equations to a circuit diagram. However, we usually want to minimize the logic gates in the circuit. We can minimize the output equations by algebraically manipulating the equations. Alternatively, we can use Karnaugh maps, as shown in Figure 2.4(d). Once we've obtained the desired output equations, we can draw the circuit diagram, as shown in Figure 2.4(e).

RT-Level Combinational Components

Although we can design all combinational circuits in this manner, large circuits would be very complex to design. For example, a circuit with 16 inputs would have 2^{16}, or 64K, rows in its truth table. One way to reduce the complexity is to use combinational components that are more powerful than logic gates. Figure 2.5 shows several such combinational components, often called *register–transfer*, or RT, level components. We now describe each briefly.

A *multiplexor*, sometimes called a *selector*, allows only one of its data inputs *Im* to pass through to the output *O*. Thus, a multiplexor acts much like a railroad switch, allowing only one of multiple input tracks to connect to a single output track. If there are *m* data inputs, then there are $\log_2(m)$ select lines *S*. We call this an *m*-by-1 multiplexor, meaning *m* data inputs, and 1 data output. The binary value of *S* determines which data input passes through; 00...00 means *I*0 passes through, 00...01 means *I*1 passes through, 00...10 means *I*2 passes through, and so on. For example, an 8×1 multiplexor has eight data inputs and thus three select lines. If those three select lines have values of 110, then *I*6 will pass through to the output. So if *I*6 were 1, then the output would be 1; if *I*6 were 0, then the output would be 0. We commonly use a more complex device called an *n*-bit multiplexor, in which each data input as well as the output consist of *n* lines. Suppose the previous example used a 4-bit 8×1 multiplexor. Thus, if *I*6 were 1110, then the output would be 1110. Note that *n* is independent of the number of select lines.

Another combinational component is a decoder. A *decoder* converts its binary input *I* into a one-hot output *O*. "One-hot" means that exactly one of the output lines can be 1 at a given time. Thus, if there are *n* outputs, then there must be $\log_2(n)$ inputs. We call this a $\log_2(n)\times n$ decoder. For example, a 3×8 decoder has three inputs and eight outputs. If the input were 000, then the output *O*0 would be 1 and all other outputs would be 0. If the input were 001, then the output *O*1 would be 1, and so on. A common feature on a decoder is an extra input called *enable*. When enable is 0, all outputs are 0. When enable is 1, the decoder functions as before.

An *adder* adds two *n*-bit binary inputs *A* and *B*, generating an *n*-bit output sum along with an output carry. For example, a 4-bit adder would have a 4-bit *A* input, a 4-bit *B* input, a 4-bit *sum* output, and a 1-bit *carry* output. If *A* were 1010 and *B* were 1001, then *sum* would be 0011 and *carry* would be 1. An adder often comes with a carry input also, so that such adders can be cascaded to create larger adders.

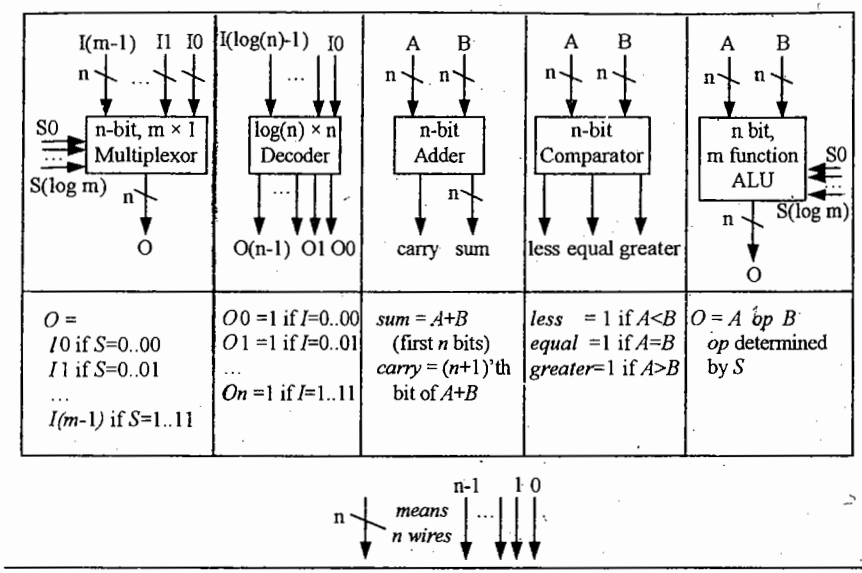


Figure 2.5: Combinational components.

A *comparator* compares two n-bit binary inputs A and B, generating outputs that indicate whether A is less than, equal to, or greater than B. If A were 1010 and B were 1001, then *less* would be 0, *equal* would be 0, and *greater* would be 1.

An *arithmetic-logic unit* (ALU) can perform a variety of arithmetic and logic functions on its n-bit inputs A and B. The select lines S choose the current function; if there are m possible functions, then there must be at least $\log_2(m)$ select lines. Common functions include addition, subtraction, AND, and OR.

Another common RT-level component, not shown in the figure, is a shifter. An n-bit input I can be shifted left or right and then output to an output O. For example, a 4-bit shifter with an input 1010 would output 0101 when shifting right one position. Shifters usually come with an additional input indicating what value should be shifted in and an additional output indicating the value of the bit being shifted out.

2.3 Sequential Logic

Flip-Flops

A *sequential circuit* is a digital circuit whose outputs are a function of the present as well as previous input values. In other words, sequential logic possesses memory. One of the most

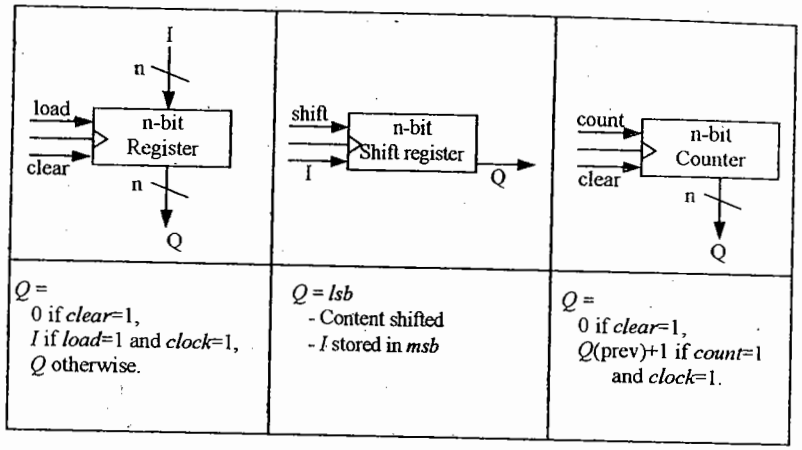


Figure 2.6: Sequential components.

basic sequential circuits is a *flip-flop*. A flip-flop stores a single bit. The simplest type is the D flip-flop. It has two inputs: D and *clock*. When *clock* is 1, the value of D is stored in the flip-flop, and that value appears at the output Q. When *clock* is 0, the value of D is ignored, and the output Q continues to reflect the stored value. Another type of flip-flop is the SR flip-flop, which has three inputs: S, R, and *clock*. When *clock* is 0, the previously stored bit is maintained and appears at output Q. When *clock* is 1, the inputs S and R are examined. If S is 1, a 1 is stored. If R is 1, a 0 is stored. If both are 0, there's no change. If both are 1, behavior is undefined. Thus, S stands for set to 1, and R for reset to 0. Another flip-flop type is a JK flip-flop, which is the same as an SR flip-flop except that when both J and K are 1, the stored bit toggles from 1 to 0 or 0 to 1. To prevent unexpected behavior from signal glitches, flip-flops are typically designed to be *edge-triggered*, meaning they examine their non-clock inputs when *clock* is rising from 0 to 1, or alternatively when *clock* is falling from 1 to 0.

RT-Level Sequential Components

Just as we used more abstract combinational components to implement complex combinational systems, we also use more abstract sequential components for complex sequential systems. Figure 2.6 illustrates several sequential components, which we describe.

A *register* stores n bits from its n-bit data input I, with those stored bits appearing at its output Q. A register usually has at least two control inputs, *clock* and *load*. For a rising-edge-triggered register, the inputs I are only stored when *load* is 1 and *clock* is rising from 0 to 1. The clock input is usually drawn as a small triangle, as shown in the figure. Another common register control input is *clear*, which resets all bits to 0, regardless of the value of I. Because all n bits of the register can be stored in parallel, we often refer to this type of register as a *parallel-load* register, to distinguish it from a shift register.

A *shift register* stores *n* bits, but these bits cannot be stored in parallel. Instead, they must be shifted into the register serially, meaning one bit per clock edge. A shift register has a 1-bit data input *I*. and at least two control inputs *clock* and *shift*. When *clock* is rising and *shift* is 1, the value of *I* is stored in the *n*th bit, while simultaneously the *n*th bit is stored in the (*n* – 1)th bit, the (*n* – 1)th bit is stored in the (*n* – 2)th bit, and so on, down to the second bit being stored in the first bit. The first bit is typically shifted out, appearing over an output *Q*.

A *counter* is a register that can also increment, meaning add binary 1, to its stored binary value. In its simplest form, a counter has a clear input, which resets all stored bits to 0, and a count input, which enables incrementing on each clock edge. A counter often also has a parallel load data input and associated load control signal. A common counter feature is both up and down counting or incrementing and decrementing, requiring an additional control input to indicate the count direction.

These control inputs can be either synchronous or asynchronous. A *synchronous* input's value only has an effect during a clock edge. An asynchronous input's value affects the circuit independent of the clock. Typically, clear control lines are asynchronous.

Sequential Logic Design

Sequential logic design can be achieved using a straightforward technique, whose steps are illustrated in Figure 2.7. We again start with a problem description, shown in Figure 2.7(a). We translate this description to a state diagram, also called a finite state machine (FSM), as in Figure 2.7(b). We describe FSMs further in a later chapter. Briefly, each state represents the current "mode" of the circuit, serving as the circuit's memory of past input values. The desired output values are listed next to each state. The input conditions that cause a transition from one state to another are shown next to each arc. Each arc condition is implicitly "ANDed" with a rising (or falling) clock edge. In other words, all inputs are synchronous. All inputs and outputs must be Boolean, and all operations must be Boolean operations. FSMs can also describe asynchronous systems, but we do not cover such systems in this book, since they are not very common.

We will implement this FSM using a register to store the current state, and combinational logic to generate the output values and the next state, as shown in Figure 2.7(c). We assign to each state a unique binary value, and we then create a truth table for the combinational logic, as in Figure 2.7(d). The inputs for the combinational logic are the state bits coming from the state register, and the external inputs, so we list all combinations of these inputs on the left side of the table. The outputs for the combinational logic are the state bits to be loaded into the register on the next clock edge (the next state), and the external output values, so we list desired values of these outputs for each input combination on the right side of the table. Because we used a state diagram for which outputs were a function of the current state only, and not of the inputs, we list an external output value only for each possible state, ignoring the external input values. Now that we have a truth table, we proceed with combinational logic design as described earlier, by generating minimized output equations as shown in Figure 2.7(e), and then drawing the combinational logic circuit as in Figure 2.7(f). As you can see, sequential logic design is very much like combinational logic design, as long as we draw the state table in such a way that it can be used as a combinational logic truth table also.

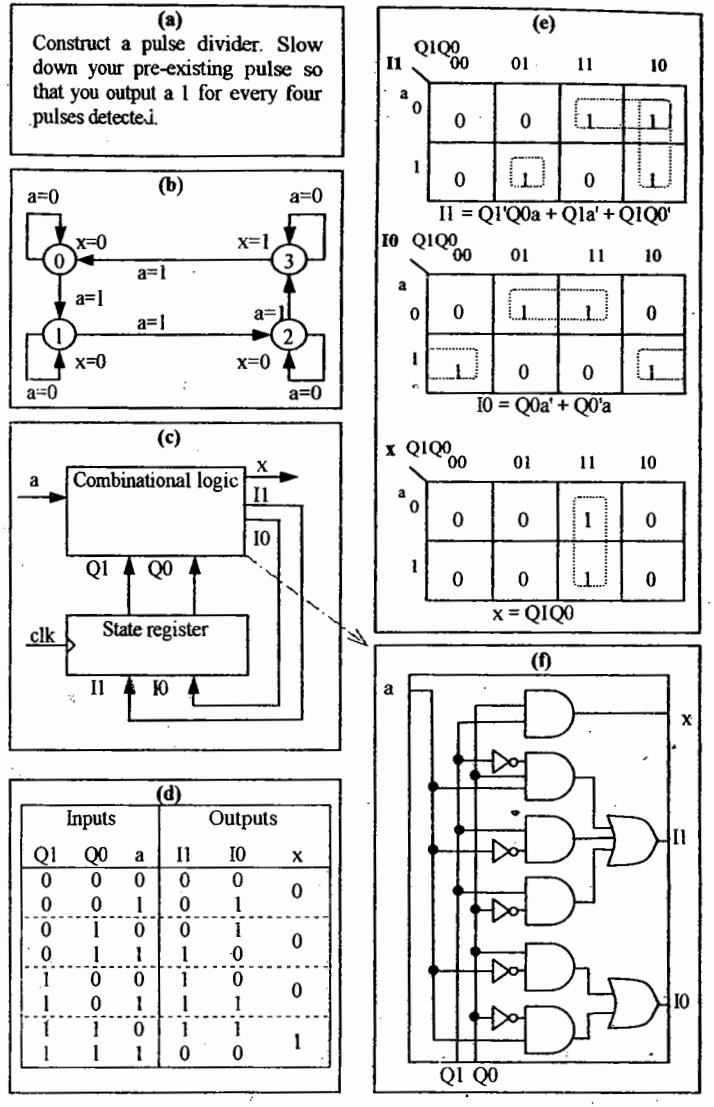


Figure 2.7: Sequential logic design: (a) problem description, (b) state diagram, (c) implementation model, (d) state table (Moore-type), (e) minimized output equations, (f) combinational logic.

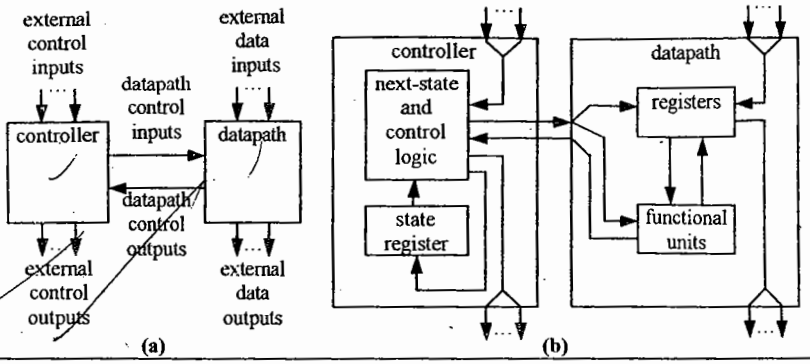


Figure 2.8: A basic processor: (a) controller and datapath, (b) a view inside the controller and datapath.

One of the biggest difficulties for people new to implementing FSMs is understanding FSM and controller timing. Consider the situation of being in state 0 in Figure 2.7(b). This means that the output x is being assigned 0, and the FSM is waiting for the next clock pulse to come along. When the new pulse comes, we'll transition to either state 0 or 1, depending on input a. From the implementation model perspective of Figure 2.7(c), in state 0, the state register has 0s in it, and these 0s are trickling through the combinational logic, eventually producing a stable 0 on output x, and the next state signals $I0$ and $I1$ are being produced as a function of the state register outputs and input a. Input a needs to be stable before the next clock pulse comes along, so that the next state signals are stable. When the next pulse does come along, the state register will be loaded with either 00 or 01. Assume 01, meaning we will now be in state 1. Then this 01 will trickle through the combinational logic, causing output x to be 0. And so on. Notice that the actions of a state occur slightly after a clock pulse causes us to enter that state.

Notice that there is a fundamental assumption being made here regarding the clock frequency, namely, that the clock frequency is fast enough to detect events on input a. In other words, input a must be held at its value long enough so that the next clock pulse will detect it. If input a switches from 0 to 1 and back to 0, all in between two clock pulses, then the switch to 1 would never be detected. Yet the clock frequency must be slow enough to allow outputs to stabilize after being generated by the combinational logic. We recommend that one study the relationship between the FSM and the implementation model for a while, until one is comfortable with this relationship.

2.4 Custom Single-Purpose Processor Design

We now have the knowledge needed to build a basic processor. A basic processor consists of a controller and a datapath, as illustrated in Figure 2.8. The *datapath* stores and manipulates a

system's data. Examples of data in an embedded system include binary numbers representing external conditions like temperature or speed, characters to be displayed on a screen, or a digitized photographic image to be stored and compressed. The datapath contains register units, functional units, and connection units like wires and multiplexors. The datapath can be configured to read data from particular registers, feed that data through functional units configured to carry out particular operations like add or shift, and store the operation results back into particular registers. A *controller* carries out such configuration of the datapath. It sets the datapath control inputs, like register load and multiplexor select signals, of the register units, functional units, and connection units to obtain the desired configuration at a particular time. It monitors external control inputs as well as datapath control outputs, known as status signals, coming from functional units, and it sets external control outputs as well.

We can apply the combinational and sequential logic design techniques described earlier to build a controller and a datapath. Therefore, we now describe a technique to convert a computation task into a custom single-purpose processor consisting of a controller and a datapath.

We begin with a sequential program describing the computation task that we wish to implement. Figure 2.9 provides an example task of computing a greatest common divisor (GCD). Figure 2.9(a) shows a black-box diagram of the desired system, having x_i and y_i data inputs and a data output d_o. The system's functionality is straightforward: the output should represent the GCD of the inputs. Thus, if the inputs are 12 and 8, the output should be 4. If the inputs are 13 and 5, the output should be 1. Figure 2.9(b) provides a simple program with this functionality. The reader might trace this program's execution on these examples to verify that the program does indeed compute the GCD.

To begin building our single-purpose processor implementing the GCD program, we first convert our program into a complex state diagram, in which states and arcs may include arithmetic expressions, and those expressions may use external inputs and outputs as well as variables. In contrast, our earlier state diagrams included only Boolean expressions, and those expressions could use only external inputs and outputs, not variables. This more complex state diagram is essentially a sequential program in which statements have been scheduled into states. We'll refer to a complex state diagram as a *finite state machine with data* (FSMD).

We can use templates to convert a program to an FSMD, as illustrated in Figure 2.10. First, we classify each statement as an assignment statement, loop statement, or branch (if-then-else or case) statement. For an assignment statement, we create a single state with that statement as its action, and we add an arc from this state to the first state of the next statement, as shown in Figure 2.10(a). For a loop statement, we create a condition state C and a join state J, both with no actions, as shown in Figure 2.10(b). We add an arc with the loop's condition from the condition state to the first statement in the loop body. We add a second arc with the complement of the loop's condition from the condition state to the next statement after the loop body. We also add an arc from the join state back to the condition state. For a branch statement, we create a condition state C and a join state J, both with no actions, as shown in Figure 2.10(c). We add an arc with the first branch's condition from the condition state to the branch's first statement. We add another arc with the complement of the first branch's condition ANDed with the second branch's condition from the condition state to the branch's

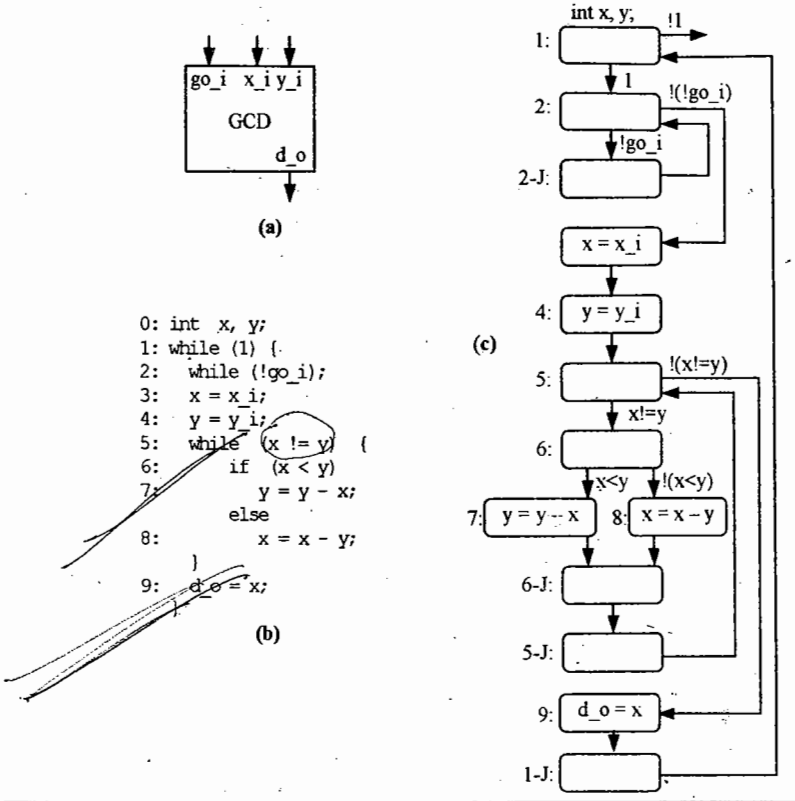


Figure 2.9: Example program – Greatest Common Divisor (GCD): (a) black-box view, (b) desired functionality, (c) state diagram.

first statement. We repeat this for each branch. Finally, we connect the arc leaving the last statement of each branch to the join state, and we add an arc from this state to the next statement's state.

Using this template approach, we convert our GCD program to the FSMD of Figure 2.9(c). Notice that variables are being assigned in some of the states, such as the action $x = x - y$, which also includes an arithmetic operation. Again, variables and arithmetic operations/conditions are what make FSMDs more powerful than FSMs.

We are now well on our way to designing a custom single-purpose processor that executes the GCD program. Our next step is to divide the functionality into a datapath part

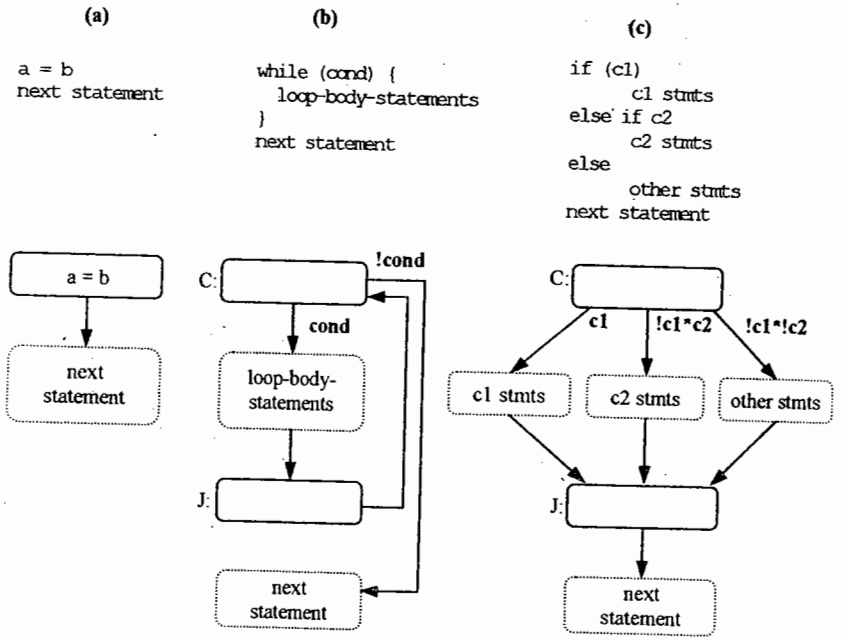


Figure 2.10: Templates for creating a state diagram from program statements: (a) assignment, (b) loop, (c) branch.

and a controller part, as shown in Figure 2.11. The datapath part should consist of an interconnection of combinational and sequential components. The controller part should consist of a pure FSM (i.e., one containing only Boolean actions and conditions.)

We construct the datapath through a four-step process:

1. First, we create a register for any declared variable. In the example, the variables are x and y. We treat an output port as an implicit variable, so we create a register d and connect it to the output port. We also draw the input and output ports. Figure 2.11(b) shows these three registers as light-gray rectangles.
2. Second, we create a functional unit for each arithmetic operation in the state diagram. In the example, there are two subtractions, one comparison for less than, and one comparison for inequality, yielding two subtractors and two comparators, shown as white rectangles in Figure 2.11(b).
3. Third, we connect the ports, registers, and functional units. For each write to a variable in the state diagram, we draw a connection from the write's source to the variable's register. A source may be an input port, a functional unit, or another register. For each arithmetic and logical operation, we connect sources to an input of the operation's corresponding functional unit. When more than one source is

connected to a register, we add an appropriately sized multiplexor, shown as dark-gray rectangles in Figure 2.11(b).

4. Finally, we create a unique identifier for each control input and output of the datapath components. Examples in the figure include *x_sel* and *x_neq_y*.

Now that we have a complete datapath, we can modify our FSMD of Figure 2.9(c) into the FSM of Figure 2.11(a) representing our controller. The FSM has the same states and transitions as the FSMD. However, we replace complex actions and conditions by Boolean ones, making use of our datapath. We replace every variable write by actions that set the

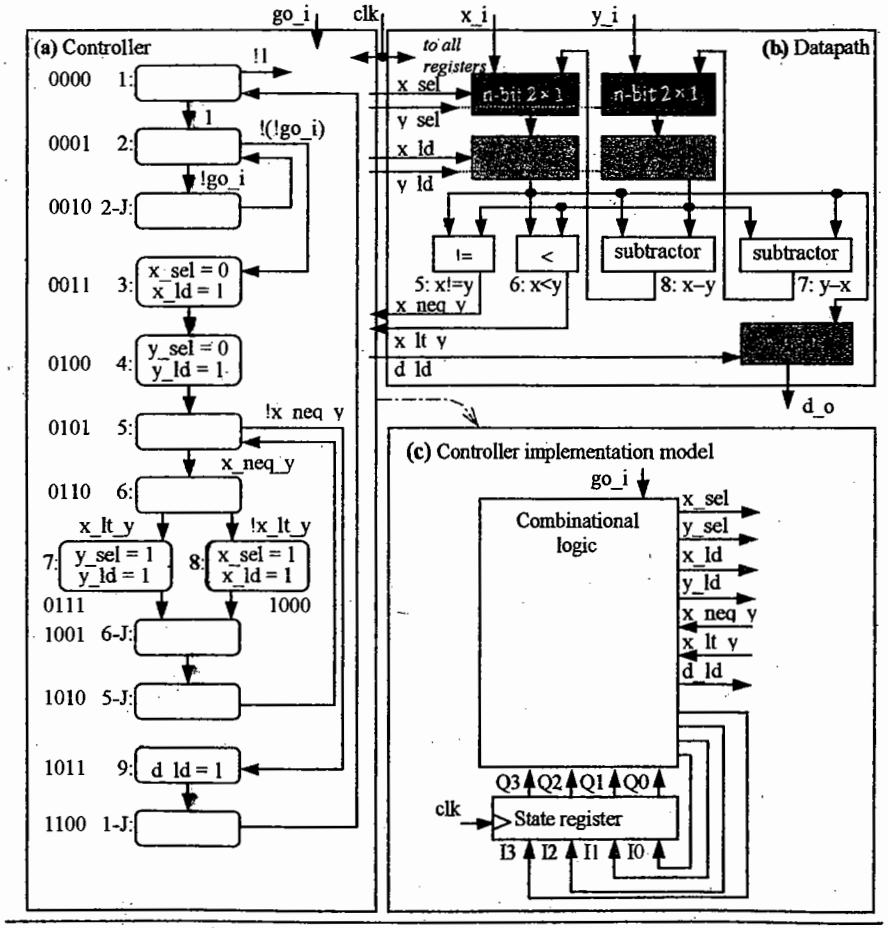


Figure 2.11: Example program — Greatest Common Divisor (GCD): (a) controller, (b) datapath, (c) controller model.

State table

Inputs							Outputs								
Q3	Q2	Q1	Q0	x_neq_y	x_lt_y	go_i	I3	I2	I1	I0	x_sel	y_sel	x_ld	y_ld	d_ld
0	0	0	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	0	1	*	*	0	0	0	1	0	X	X	0	0	0
0	0	0	1	*	*	1	0	0	1	1	X	X	0	0	0
0	0	1	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	1	1	*	*	*	0	1	0	0	0	X	1	0	0
0	1	0	0	*	*	*	0	1	0	1	X	0	0	1	0
0	1	0	1	0	*	*	1	0	1	1	X	X	0	0	0
0	1	0	1	1	*	*	0	1	1	0	X	X	0	0	0
0	1	1	0	*	0	*	1	0	0	0	X	X	0	0	0
0	1	1	0	*	1	*	0	1	1	1	X	X	0	0	0
0	1	1	1	*	*	*	1	0	0	1	X	1	0	1	0
1	0	0	0	*	*	*	1	0	0	1	1	X	1	0	0
1	0	0	1	*	*	*	1	0	1	0	X	X	0	0	0
1	0	1	0	*	*	*	0	1	0	1	X	X	0	0	0
1	0	1	1	*	*	*	1	1	0	0	X	X	0	0	1
1	1	0	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	0	1	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	1	*	*	*	0	0	0	0	X	X	0	0	0

* - indicates all possible com[illegible]itions of 0s and 1s
X - indicates don't cares

Figure 2.12: State table for the GCD example.

select signals of the multiplexor in front of the variable's register such that the write's source passes through, and we assert the load signal of that register. We replace every logical operation in a condition by the corresponding functional unit control output. In this FSM, any signal not explicitly assigned in a state is implicitly assigned a 0. For example, *x_ld* is implicitly assigned 0 in every state except for 3 and 8, where *x_ld* is explicitly assigned 1.

We can then complete the controller design by implementing the FSM using our sequential design technique described earlier and illustrated in Figure 2.4. Figure 2.11(c) shows the controller implementation model. Figure 2.12 shows a state table for the controller. Note that there are seven inputs to the controller, resulting in 128 rows for the state table. We reduced rows in the state table of the figure by using * for some input combinations, but we can still see that optimizing the design using hand techniques could be quite tedious. For this reason, computer-aided design (CAD) tools that automate both the combinational and the sequential logic design can be very helpful; we'll introduce some CAD tools in the last chapter. CAD tools that automatically generate digital gates from sequential programs, FSMDs, FSMs, or logic equations are known as synthesis tools.

Also, note that we could perform significant amounts of optimization to both the datapath and the controller. For example, we could merge functional units in the datapath, resulting in fewer units at the expense of more multiplexors. We could also merge a number of states into a single state, reducing the size of the controller. Interested readers might examine the textbook by Gajski referred to at the end of this chapter for an introduction to these optimizations.

Note that we could alternatively implement the GCD program by programming a general-purpose processor, thus eliminating the need for this design process, but possibly yielding a slower and bigger design.

Finally, we once again discuss timing, this time for FSMDs rather than FSMs. When in a particular state, all actions internal to that state are considered to be concurrent to one another. Those actions are very different from a sequential program, in which statements are executed in sequence. So, if $x = 0$ before entering a state A in an FSMD, and state A's actions are "$x = x + 1$" and "$y = x$," then y will equal 0, not 1, after exiting state A. This concurrency of actions also implies that the order in which we write the actions in the state does not matter.

Furthermore, note that actions consisting of writes to variables do not actually update those variables until the next clock pulse, because those variables are implemented as registers. However, arcs leaving a state may use those variables in their conditions. Thus, an arc leaving state A, but using variable x, is using the old value of x, 0 in our example in the previous paragraph. Assuming an outgoing arc is using the new value assigned in the arc's source state is by far the most common mistake that people make when creating FSMDs. If we wish to assign a value to variable x and then branch to different states depending on that value, then we must insert an additional state before branching.

2.5 RT-Level Custom Single-Purpose Processor Design

Section 2.4 described a basic technique for converting a sequential program into a custom single-purpose processor, by first converting the program to an FSMD using the provided templates for each language construct, splitting the FSMD into a simple FSM controlling a datapath, and performing sequential logic design on the FSM. However, in many cases, we prefer not to start with a program, but instead directly with an FSMD. The reason is that often the cycle-by-cycle timing of a system is central to the design, but programming languages don't typically support cycle-by-cycle description. FSMDs, in contrast, make cycle-by-cycle timing explicit.

For example, consider the design problem in Figure 2.13(a). We want one device (the sender) to send an 8-bit number to another device (the receiver). The problem is that while the receiver can receive all 8 bits at once, the sender sends 4 bits at a time; first it sends the low-order 4 bits, then the high-order 4 bits. So we need to design a bridge that will enable to two devices to communicate.

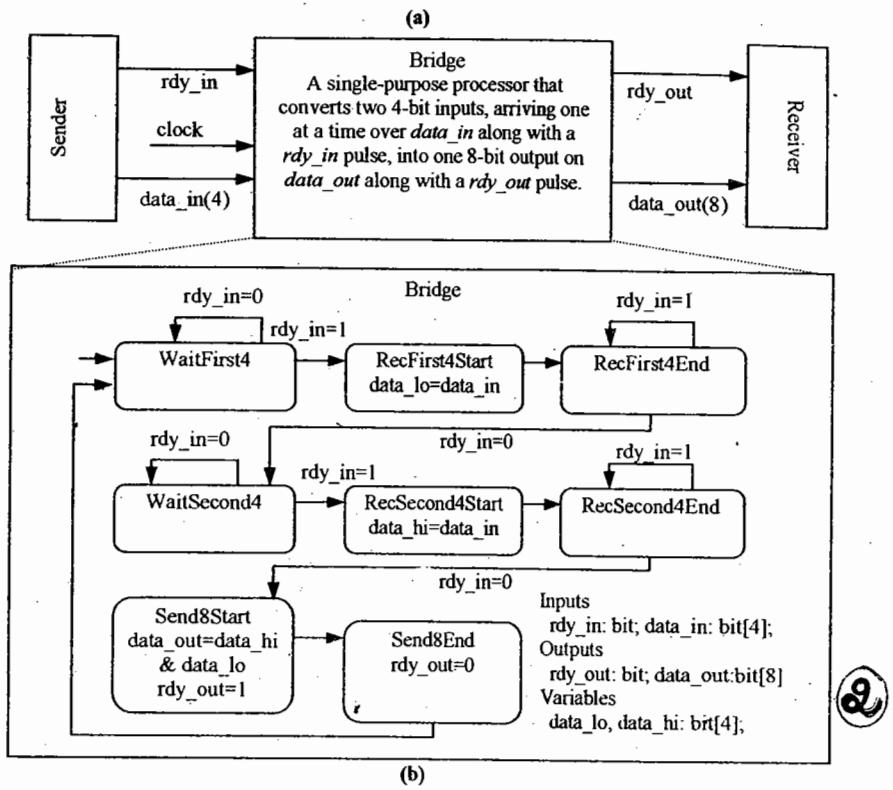


Figure 2.13: RT-level custom single-purpose processor design example: (a) problem specification, (b) FSMD.

Different designers might attack this problem at different levels of abstraction. One designer might start thinking in terms of registers, multiplexors, and flip-flops. Another might try to describe the bridge as a sequential program. But perhaps the most natural level is to describe the bridge as an FSMD, as shown in Figure 2.13(b). We begin by creating a state *WaitFirst4* that waits for the first 4 bits, whose presence on *data_in* will be indicated by a pulse on *rdy_in*. Once the pulse is detected, we transition to a state *RecFirst4Start* that saves the contents of *data_in* in a variable called *data_lo*. We then wait for the pulse on *rdy_in* to end, and then wait for the other 4 bits, indicated by a second pulse on *rdy_in*. We save the contents of *data_in* in a variable called *data_hi*. After waiting for the second pulse on *rdy_in* to end, we write the full 8 bits of data to the output *data_out*, and we pulse *rdy_out*. We

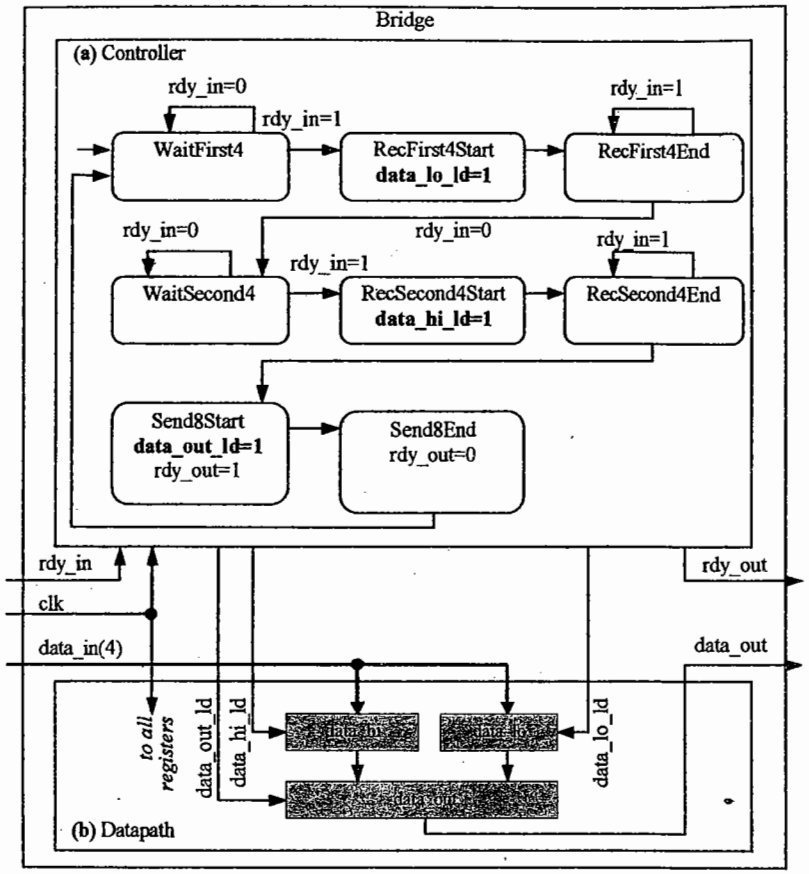


Figure 2.14: RT-level custom single-purpose processor design example continued: (a) controller, (b) datapath.

assume we are building a synchronous circuit, so the bridge has a clock input — in our FSMD, every transition is implicitly ANDed with the clock.

We apply the same methods as before to convert this FSMD to a controller and a datapath implementation, as illustrated in Figure 2.14. We build a datapath, shown in Figure 2.14(b), using the four-step process outlined before. We add registers for *data_hi* and *data_lo*, as well as for the output *data_out*. We don't add any functional units since there are no arithmetic operations. We connect the registers according to the assignments in the FSMD; no multiplexors are necessary. We create unique identifiers for the register control signals. Having completed the datapath, we convert the FSMD into an FSM that uses the datapath, as

shown in Figure 2.14(a). This conversion requires only three simple changes, as shown in bold in the figure. Having obtained the FSM, we can convert the FSM into a state-register and combinational logic using the same technique as in Figure 2.7; we omit this conversion here.

This example demonstrates how a problem that consists mostly of waiting for or making changes on signals, rather than consisting mostly of performing computations on data, might most easily be described as an FSMD. The FSMD would be even more appropriate if specific numbers of clock cycles were specified (e.g., the input pulse would be held high exactly two cycles and the output pulse would have to be held high for three cycles). On the other hand, if a problem consists mostly of an algorithm with lots of computations, the detailed timing of which are not especially important, such as the GCD computation in the earlier example, then a program might be the best starting point.

The FSMD level is often referred to as the register-transfer (RT) level, since an FSMD describes in each state which registers should have their data transferred to which other registers, with that data possibly being transformed along the way. The RT-level is probably the most common starting point for custom single-purpose processor design today.

Some custom single-purpose processors do not manipulate much data. These processors consist primarily of a controller, with perhaps no datapath or a trivial one with just a couple registers or counters, as in our bridge example of Figure 2.14. Likewise, other custom single-purpose processors do not exhibit much control. These processors consist primarily of a datapath configured to do one or a few things repeatedly, with no controller or a trivial one with just a couple flip-flops and gates. Nevertheless, we can still think of these circuits as processors.

2.6 Optimizing Custom Single-Purpose Processors

You may have noticed in the GCD example of Figure 2.11 that we ignored several opportunities to simplify the resulting design. For example, the FSM had several states that obviously do nothing and could have been removed. Likewise, the datapath has two adders whereas one would have been sufficient. We intentionally did not perform such optimizations so as not to detract from the basic idea that programs can be converted to custom single-purpose processors through a series of straightforward steps. However, when we really design such processors, we will usually also want to optimize them whenever possible. Optimization is the task of making design metric values the best possible. Optimization is an extensive subject, and we do not intend to cover it in depth here. Instead, we point out some simple optimizations that can be applied, and refer the reader to textbooks on the subject.

Optimizing the Original Program

Let us start with optimizing the initial program, such as the GCD program in Figure 2.9. At this level, we can analyze the number of computations and size of variables that are required by the algorithm. In other words, we can analyze the algorithm in terms of time complexity and space complexity. We can try to develop alternative algorithms that are more efficient. In

the GCD example, if we assume we can make use of a modulo operation %, we could write an algorithm that would use fewer steps. In particular, we could use the following algorithm:

```
int  x, y, r;
while (1) {
   while (!go_i);
   if (x_i >= y_i)    {x=x_i; y=y_i;}
   else {x=y_i; y=x_i;} // x must be the larger number
   while  (y != 0)  {
      r = x % y;
      x = y;
      y = r;
   }
   d_o = x;
}
```

Let us compare this second algorithm with the earlier one when computing the GCD of 42 and 8. The earlier algorithm would step through its inner loop with x and y values as follows: (42,8), (34,8), (26,8), (18,8), (10,8), (2,8), (2,6), (2,4), (2,2), thus outputting 2. The second algorithm would step through its inner loop with x and y values as follows: (42,8), (8,2), (2,0), thus outputting 2. The second algorithm is far more efficient in terms of time. Analysis of algorithms and their efficient design is a widely researched area. The choice of algorithm can have perhaps the biggest impact on the efficiency of the designed processor.

Optimizing the FSMD

Once an algorithm is settled upon, we convert the program describing that algorithm to an FSMD. Use of the template-based method introduced in this chapter will result in a rather inefficient FSMD. In particular, many states in the resulting FSMD could likely be merged into fewer states.

Scheduling is the task of assigning operations from the original program to states in an FSMD. The scheduling obtained using the template-based method can be improved. Consider the original FSMD for the GCD, which is redrawn in Figure 2.15(a). State 1 is clearly not necessary since its outgoing transitions have constant values. States 2 and 2-J can be merged into a single state since there are no loop operations in between them. States 3 and 4 can be merged since they perform assignment operations that are independent of one another. States 5 and 6 can be merged. States 6-J and 5-J can be eliminated, with the transitions from states 7 and 8 pointing directly to state 5. Likewise, state 1-J can be eliminated. The resulting reduced FSMD is shown in Figure 2.15(b). We reduced the FSMD from thirteen states to only six states. Be careful, though, to avoid the common mistake of assuming that a variable assigned in a state can have the newly assigned value read on an outgoing arc of that state!

The original FSMD could also have had too few states to be efficient in terms of hardware size. Suppose a particular program statement had the operation $a = b * c * d * e$. Generating a single state for this operation will require us to use three multipliers in our datapath. However, multipliers are expensive, and thus we might instead want to break this

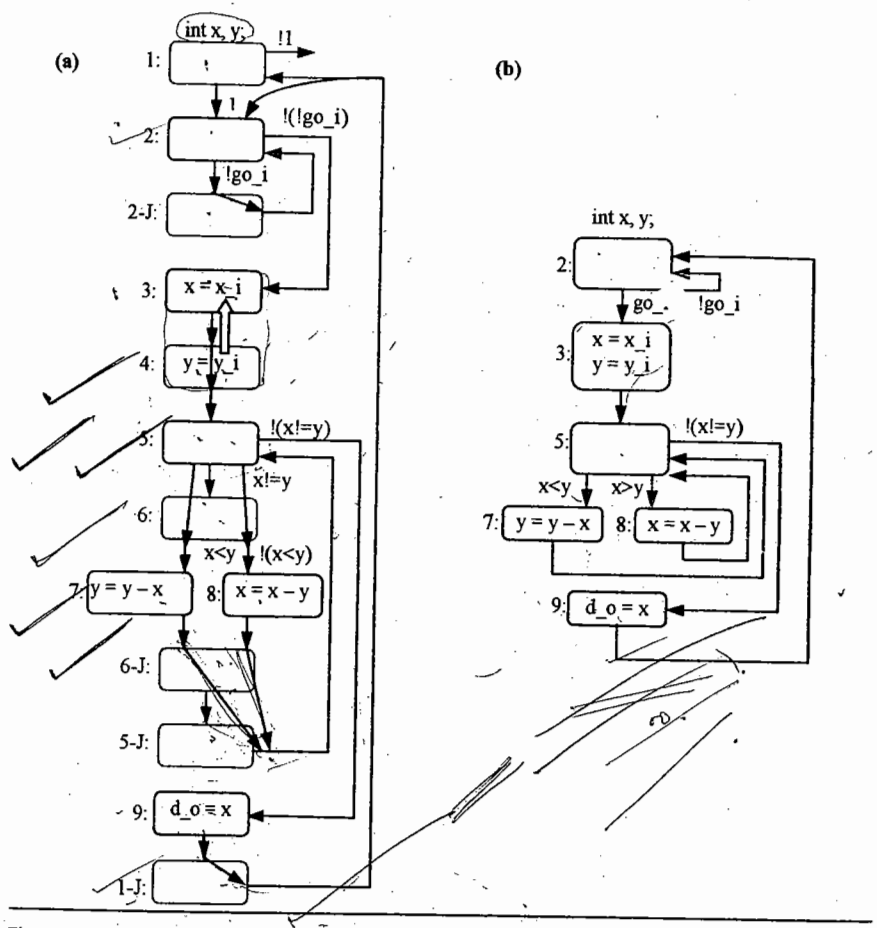


Figure 2.15: Optimizing the FSMD for the GCD example: (a) original FSMD and optimizations, and (b) optimized FSMD.

operation down into smaller operations, like $t1 = b * c$, $t2 = d * e$, and $a = t1 * t2$, with each smaller operation having its own state. Thus, only one multiplier would be needed in the datapath, since the three multiplications could share multiplier; sharing will be discussed in the next section.

In this scenario, we assumed that the timing of output operations could be changed. For example, the reduced FSMD will generate the GCD output in fewer clock cycles than the original FSMD. In many cases, changing the timing is not acceptable. For example, in our earlier clock divider example, changing the timing clearly would not be acceptable, since we

intended for the cycle-by-cycle behavior of the original FSM to be preserved during design. Thus, when optimizing the FSMD, a design must be aware of whether output timing may or may not be modified.

Optimizing the Datapath

In our four-step datapath approach, we created a unique functional unit for every arithmetic operation in the FSMD. However, such a one-to-one-mapping is often not necessary. Many arithmetic operations in the FSMD can share a single functional unit if that functional unit supports those operations, and those operations occur in different states. In the GCD example, states 7 and 8 both performed subtractions. In the datapath of Figure 2.11, each subtraction got its own subtractor. Instead, we could use a single subtractor and use multiplexors to choose whether the subtractor inputs are x and y, or instead y and x.

Furthermore, we often have a number of different RT components from which we can build our datapath. For example, we have fast and slow adders available. We may have multifunction components, like ALUs, also. *Allocation* is the task of choosing which RT components to use in the datapath. *Binding* is the task of mapping operations from the FSMD to allocated components.

Scheduling, allocation, and binding are highly interdependent. A given schedule will affect the range of possible allocations, for example. An allocation will affect the range of possible schedules. And so on. Thus, we sometimes want to consider these tasks simultaneously.

Optimizing the FSM

Designing a sequential circuit to implement an FSM also provides some opportunities for optimization, namely, state encoding and state minimization.

State encoding is the task of assigning a unique bit pattern to each state in an FSM. Any assignment in which the encodings are unique will work properly, but the size of the state register as well as the size of the combinational logic may differ for different encodings. For example, four states A, B, C, and D can be encoded as 00, 01, 10, and 11, respectively. Alternatively, those states can be encoded as 11, 10, 00, and 01, respectively. In fact, for an FSM with n states where n is a power of 2, there are $n!$ possible encodings. We can see this easily if we treat encoding as an ordering problem — we order the states and assign a straightforward binary encoding, starting with 00...00 for the first state, 00...01 for the second state, and so on. There are $n!$ possible orderings of n items, and thus $n!$ possible encodings. $n!$ is a very large number for large n, and thus checking each encoding to determine which yields the most efficient controller is a hard problem. Even more encodings are possible, since we can use more than $\log_2(n)$ bits to encode n states, up to n bits to achieve a one-hot encoding. CAD tools are therefore a great aid in searching for the best encoding.

State minimization is the task of merging equivalent states into a single state. Two states are equivalent if, for all possible input combinations, those two states generate the same outputs and transition to the same next state. Such states are clearly equivalent, since merging them will yield exactly the same output behavior.

The state merging that we did when optimizing our FSMD was not the same as state minimization as defined here. The reason is that our state merging in the FSMD actually changed the output behavior, in particular the output timing, of the FSMD. Typically, by the time we arrive at an FSM, we assume output timing cannot be changed. State minimization does not change the output behavior in any way.

2.7 Summary

Designing a custom single-purpose processor for a given program requires an understanding of various aspects of digital design. Design of a circuit to implement Boolean functions requires combinational design, which consists of building a truth table with all possible inputs and desired outputs, optimizing the output functions, and drawing a circuit. Design of a circuit to implement a state diagram requires sequential design, which consists of drawing an implementation model with a state register and a combinational logic block, assigning a binary encoding to each state, drawing a state table with inputs and outputs, and repeating our combinational design process for this table. Finally, design of a single-purpose processor circuit to implement a program requires us to first schedule the program's statements into a complex state diagram, construct a datapath from the diagram, create a new state diagram that replaces complex actions and conditions by datapath control operations, and then design a controller circuit for the new state diagram using sequential design. The register-transfer level is the most common starting point of design today. Much optimization can be performed at each level of design, but such optimization is hard, so CAD tools would be a great designer's aid.

2.8 References and Further Reading

- De Micheli, Giovanni, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994. Covers synthesis techniques from sequential programs down to gates.
- Gajski, Daniel D., *Principles of Digital Design*. Englewood Cliffs, NJ: Prentice-Hall, 1997. Describes combinational and sequential logic design, with a focus on optimization techniques, CAD, and higher levels of design.
- Gajski, Daniel D., Nikil Dutt, Allen Wu, and Steve Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Norwell, MA: Kluwer Academic Publishers, 1992. Emphasizes optimizations when converting sequential programs to a custom single-purpose processor.
- Katz, Randy, *Contemporary Logic Design*. Redwood City, CA: Benjamin/Cummings, 1994. Describes combinational and sequential logic design, with a focus on logic and sequential optimization and CAD.

2.9 Exercises

2.1 What is a single-purpose processor? What are the benefits of choosing a single-purpose processor over a general-purpose processor?

2.2 How do nMOS and pMOS transistors differ?

2.3 Build a 3-input NAND gate using a minimum number of CMOS transistors.

2.4 Build a 3-input NOR gate using a minimum number of CMOS transistors.

2.5 Build a 2-input AND gate using a minimum number of CMOS transistors.

2.6 Build a 2-input OR gate using a minimum number of CMOS transistors.

2.7 Explain why NAND and NOR gates are more common than AND and OR gates.

2.8 Distinguish between a combinational circuit and a sequential circuit.

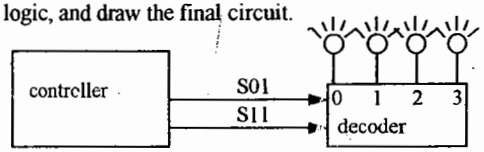
2.9 Design a 2-bit comparator (compares two 2-bit words) with a single output "less-than," using the combinational design technique described in the chapter. Start from a truth table, use K-maps to minimize logic, and draw the final circuit.

2.10 Design a 3×8 decoder. Start from a truth table, use K-maps to minimize logic and draw the final circuit.

2.11 Describe what is meant by edge-triggered and explain why it is used.

2.12 Design a 3-bit counter that counts the following sequence: 1, 2, 4, 5, 7, 1, 2, etc. This counter has an output "odd" whose value is 1 when the current count value is odd. Use the sequential design technique of the chapter. Start from a state diagram, draw the state table, minimize the logic, and draw the final circuit.

2.13 Four lights are connected to a decoder. Build a circuit that will blink the lights in the following order: 0, 2, 1, 3, 0, 2, ... Start from a state diagram, draw the state table, minimize the logic, and draw the final circuit.



2.14 Design a soda machine controller, given that a soda costs 75 cents and your machine accepts quarters only. Draw a black-box view, come up with a state diagram and state table, minimize the logic, and then draw the final circuit.

2.15 What is the difference between a synchronous and an asynchronous circuit?

2.16 Determine whether the following are synchronous or asynchronous: (a) multiplexor, (b) register, (c) decoder.

2.17 What is the purpose of the datapath? of the controller?

2.18 Compare the GCD custom-processor implementation to a software implementation (a) Compare the performance. Assume a 100-ns clock for the microcontroller, and a 20-ns clock for the custom processor. Assume the microcontroller uses two operand instructions, and each instruction requires four clock cycles. Estimates for the microcontroller are fine. (b) Estimate the number of gates for the custom design, and compare this to 10,000 gates for a simple 8-bit microcontroller. (c) Compare the custom GCD with the GCD running on a 300-MHz processor with 2-operand instructions and one clock cycle per instruction (advanced processors use parallelism to meet or exceed one cycle per instruction). (d) Compare the estimated gates with 200,000 gates, a typical number of gates for a modern 32-bit processor.

2.19 Design a single-purpose processor that outputs Fibonacci numbers up to n places. Start with a function computing the desired result, translate it into a state diagram, and sketch a probable datapath.

2.20 Design a circuit that does the matrix multiplication of matrices A and B. Matrix A is 3×2 and matrix B is 2×3. The multiplication works as follows:

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}} \bullet \underset{B}{\begin{bmatrix} g & h & i \\ j & k & l \end{bmatrix}} = \underset{C}{\begin{bmatrix} a*g + b*j & a*h + b*k & a*i + b*l \\ c*g + d*j & c*h + d*k & c*i + d*l \\ e*g + f*j & e*h + f*k & e*i + f*l \end{bmatrix}}.$$

2.21 An algorithm for matrix multiplication, assuming that we have one adder and one multiplier, follows. (a) Convert the matrix multiplication algorithm into a state diagram using the template provided in Figure 2.10. (b) Rewrite the matrix multiplication algorithm given the assumption that we have three adders and six multipliers. (c) If each multiplication takes two cycles to compute and each addition takes one cycle compute, how many cycles does it take to complete the matrix multiplication given one adder and one multiplier? Three adders and six multipliers? Nine adders and 18 multipliers? (d) If each an adder requires 10 transistors to implement and each multiplier requires 100 transistors to implement, what is the total number of transistor needed to implement the matrix multiplication circuit using one adder and one multiplier? Three adders and six multipliers? Nine adders and 18 multipliers? (e) Plot your results from parts (c) and (d) into a graph with latency along the x-axis and size along the y-axis.

```
main( ) {
        int A[3][2] = { {1, 2}, {3,4}, {5,6} };
        int B[2][3] = { {7, 8, 9}, {10, 11, 12} };
        int C[3][3];
        int i, j, k;

for (i=0; i < 3; i++){
        for ( j=0; j < 3; j++){
                C[i][j]=0;
                for (k=0; k < 2; k++){
                        C[i][j] += A[i][k] * B[k][j];
                }
        }
}
```

2.22 A subway has an embedded system controlling the turnstile, which releases when two tokens are deposited. (a) Draw the FSMD state diagram for this system. (b) Separate the FSMD into an FSM+D. (c) Derive the FSM logic using truth tables and K-maps to minimize logic. (d) Draw your FSM and datapath connections.