

Chapter 4 *Custom single-purpose processors: Hardware*

4.1 Introduction

As mentioned in the previous chapter, a single-purpose processor is a digital system intended to solve a specific computation task. While a manufacturer builds a standard single-purpose processor for use in a variety of applications, we build a custom single-purpose processor to execute a specific task within our embedded system. An embedded system designer choosing to use a custom single-purpose, rather than a general-purpose, processor to implement part of a system's functionality may achieve several benefits, similar to some of those of the previous chapter.

First, performance may be fast, due to fewer clock cycles resulting from a customized datapath, and due to shorter clock cycles resulting from simpler functional units, less multiplexors, or simpler control logic. Second, size may be small, due to a simpler datapath and no program memory. In fact, the processor may be faster and smaller than a standard one implementing the same functionality, since we can optimize the implementation for our particular task.

However, because we probably won't manufacture as many of the custom processor as a standard processor, we may not be able to invest as much NRE, unless the embedded system we are building will be sold in large quantities or does not have tight cost constraints. This fact could actually penalize performance and size.

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

4.2 Combinational logic design

A transistor is the basic electrical component of digital systems. Combinations of transistors form more abstract components called logic gates, which designers primarily 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 (CMOS -- Complementary Metal Oxide Semiconductor) is shown in Figure 4.1(a). The *gate* (not to be confused with logic gate) controls whether or not current flows from the *source* to the *drain*. When a high voltage (typically +5 Volts, which we'll refer to as logic 1) is applied to the gate, the transistor conducts, so current flows. When low voltage (which we'll refer to as logic 0, typically ground, which is drawn as several horizontal lines of decreasing width) is applied to the gate, the transistor does not conduct. We can also build a transistor with the opposite functionality, illustrated in Figure 4.1(b). When logic 0 is applied to the gate, the transistor conducts, and 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 4.1(c). When the input x is logic 0, the top transistor conducts (and the bottom does not), 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 4.1(d). When at least one of the inputs x and y is logic 0, then at least one of the top transistors conducts (and the bottom transistors do not), so logic 1 appears at F . If both inputs are logic 1, then neither of the top transistors conducts, but both of the bottom ones 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 4.1(e). The three circuits shown implement three basic logic gates: an inverter, a NAND gate, and a NOR gate.

Figure 4.1: CMOS transistor implementations of some basic logic gates: (a) nMOS transistor, (b) pMOS transistor, (c) inverter, (d) NAND gate, (e) NOR gate.

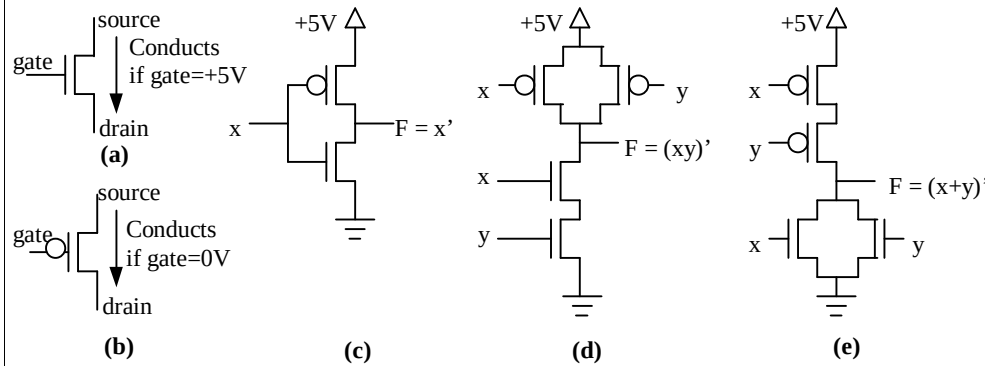
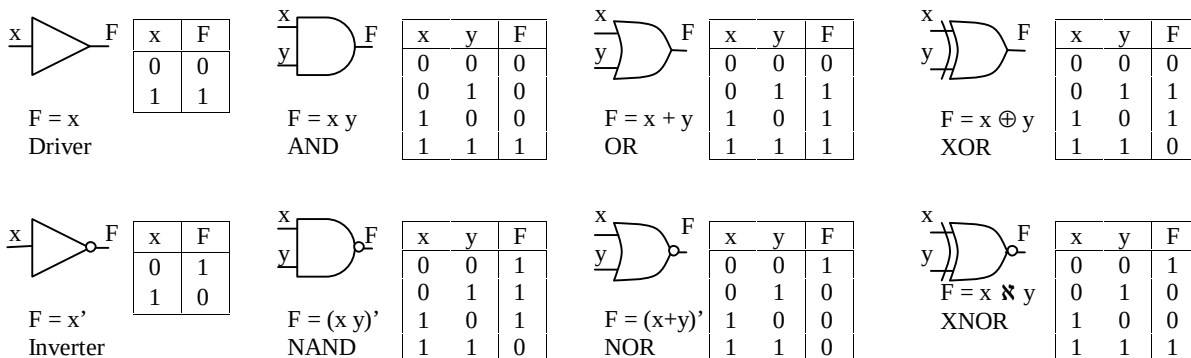


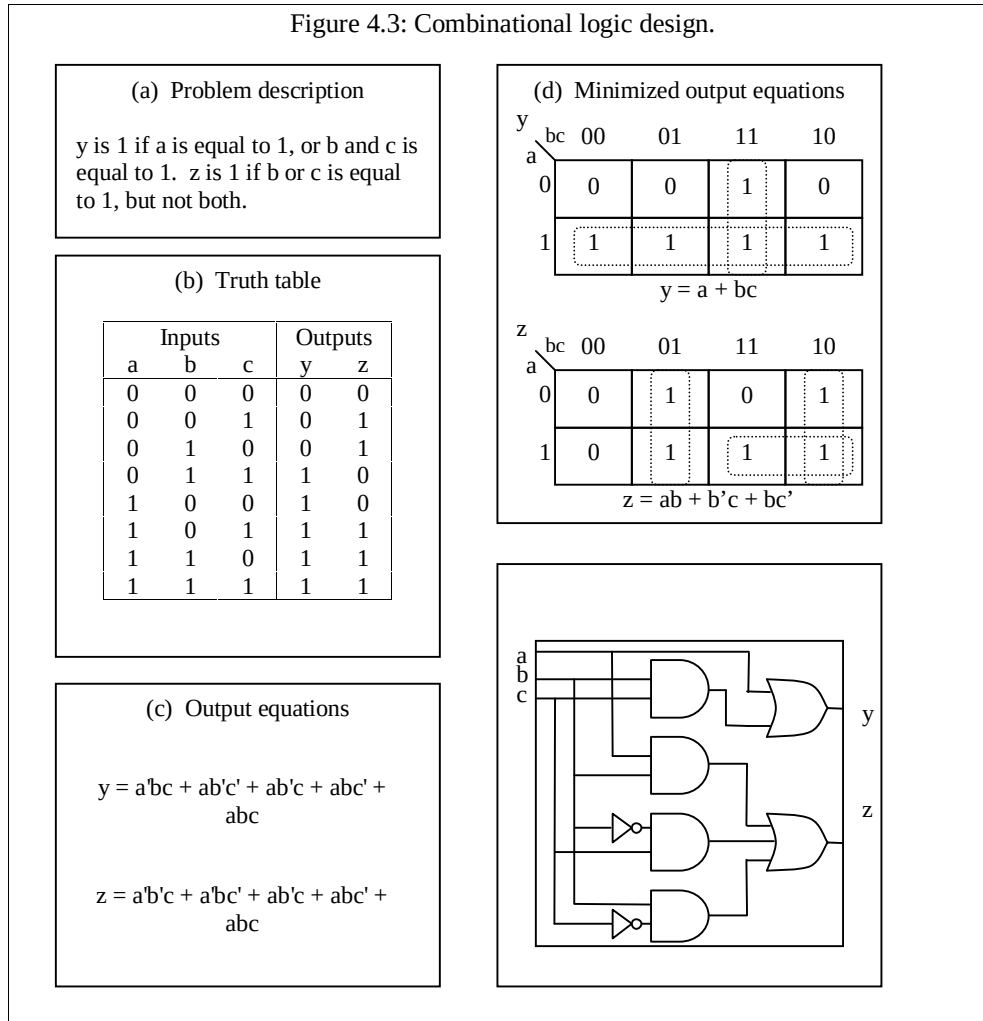
Figure 4.2: Basic logic gates



Digital system designers usually work with logic gates, not transistors. Figure 4.2 describes 8 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 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 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. As you might have noticed from our transistor implementations, the NAND and NOR gates are actually simpler to build than AND and OR gates.

A *combinational* circuit is a digital circuit whose output is purely a function of its current inputs; such a circuit has no memory of past inputs. We can apply a simple technique to design a combinational circuit using our basic logic gates, as illustrated in Figure 4.3. We start with a problem description, which describes the outputs in terms of the inputs. We translate that description to a truth table, with all possible combinations of input values on the left, and desired output values on the right. For each output column, we can derive an output equation, with one term per row. However, we often 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

Figure 4.3: Combinational logic design.

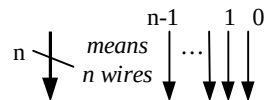
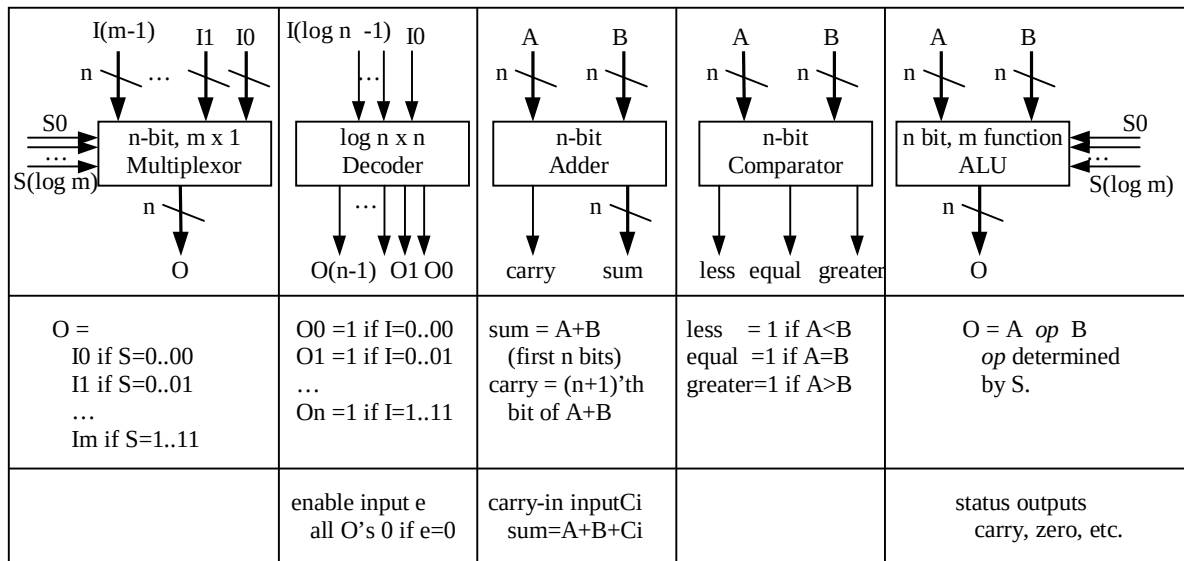


shown in the figure. Once we've obtained the desired output equations (minimized or not), we can draw the circuit diagram.

Although we can design all combinational circuits in the above 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 components that are more abstract than logic gates. Figure 4.4 shows several such combinational components. We now describe each briefly.

A *multiplexor*, sometimes called a selector, allows only one of its data inputs I_m 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 , and we call this an m -by-1 multiplexor (m data inputs, one data output). The binary value of S determines which data input passes through; 00...00 means I_0 may pass, 00...01 means I_1 may pass, 00...10 means I_2 may pass, and so on. For example, an 8x1 multiplexor has 8 data inputs and thus 3 select lines. If those three select lines have values of 110, then I_6 will pass through to the output. So if I_6 is 1, then the output would be 1; if I_6 is 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, consists of n lines. Suppose the previous example used a 4-bit 8x1 multiplexor. Thus, if I_6 is 0110, then the output would be 0110. Note that n does not affect the number of select lines.

Figure 4.4: Combinational components.



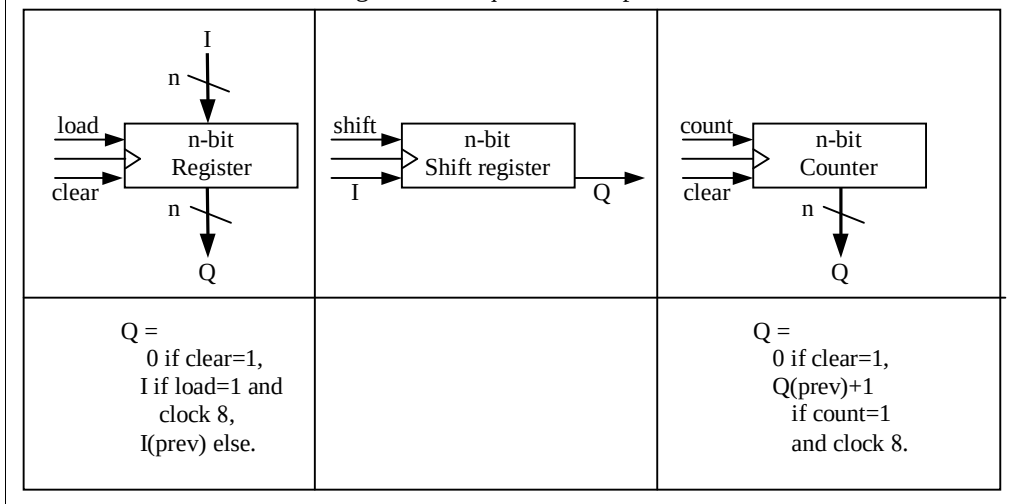
A *decoder* converts its binary input I into a one-hot output O . "One-hot" means that exactly one of the output lines can be 1 at a given time. Thus, if there are n outputs, then there must be $\log_2(n)$ inputs. We call this a $\log_2(n) \times n$ decoder. For example, a 3×8 decoder has 3 inputs and 8 outputs. If the input is 000, then the output O_0 will be 1. If the input is 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 is 1010 and B is 1001, then *sum* would be 0011 and *carry* would be 1.

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 is 1010 and B is 1001, then *less* would be 0, *equal* would be 0, and *greater* would be 1.

An *ALU* (arithmetic-logic unit) 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.

Figure 4.5: Sequential components.



4.3 Sequential logic design

A *sequential circuit* is a digital circuit whose outputs are a function of the current as well as previous input values. In other words, sequential logic possesses memory. One of the most basic sequential circuits is the *flip-flop*. A flip-flop stores a single bit. The simplest type of flip-flop 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 an output *Q*. When *clock* is 0, the value of *D* is ignored; the output *Q* maintains its 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, and *R* for reset. 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 only pay attention to their non-clock inputs when the clock is *rising* from 0 to 1, or alternatively when the clock is *falling* from 1 to 0.

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 4.5 illustrates several sequential components, which we now 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, which we now describe.

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 one-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 the (*n*)th bit is stored in the (*n*-1)th bit, and likewise, until the second bit is stored in the first bit. The first bit is typically shifted out, meaning it appears over an output *Q*.

A *counter* is a register that can also increment (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 the clock edge. A counter often also has a parallel load data input and associated control signal. A common counter feature is both up and down counting (incrementing and decrementing), requiring an additional control input to indicate the count direction.

The control inputs discussed above 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 can be achieved using a straightforward technique, whose steps are illustrated in Figure 4.1. We again start with a problem description. We translate this description to a state diagram. We describe state diagrams 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 AND'ed with a rising (or falling) clock edge. In other words, all inputs are synchronous. State diagrams can also describe asynchronous systems, but we do not cover such systems in this book, since they are not common.

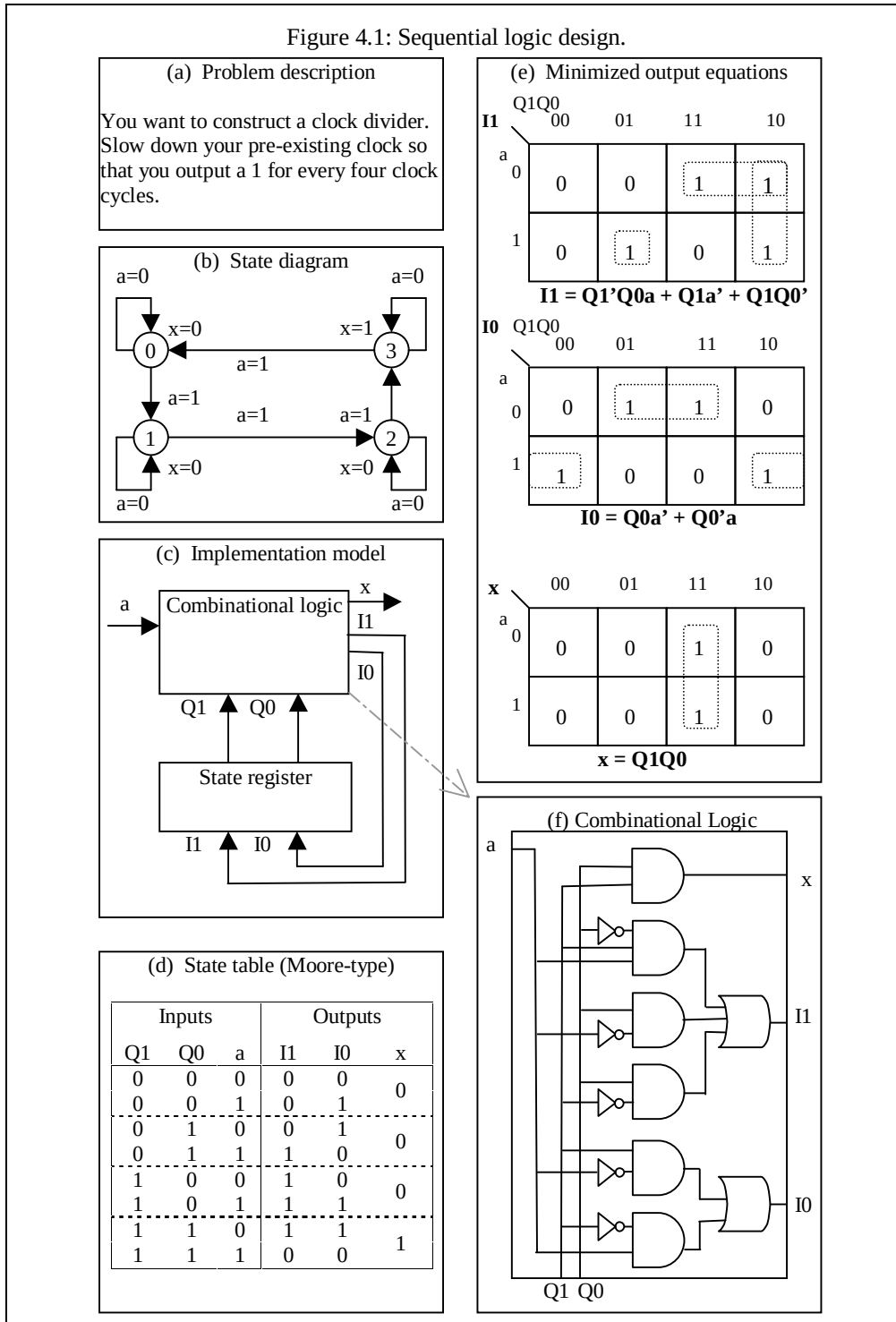
We will implement this state diagram using a register to store the current state, and combinational logic to generate the output values and the next state. We assign each state with a unique binary value, and we then create a truth table for the combinational logic. The inputs for the combinational logic 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, and then drawing the combinational logic circuit.

4.4 Custom single-purpose processor design

We can apply the above combinational and sequential logic design techniques to build datapath components and controllers. Therefore, we have nearly all the knowledge we need to build a custom single-purpose processor for a given program, since a processor consists of a controller and a datapath. We now describe a technique for building such a processor.

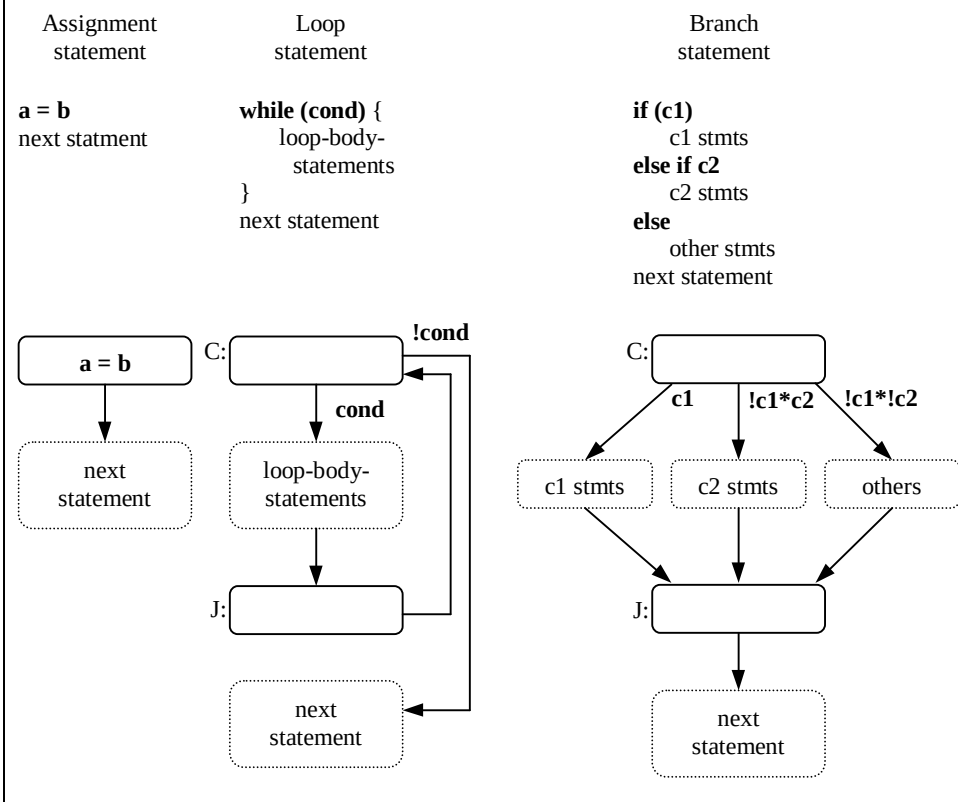
We begin with a sequential program we must implement. Figure 4.3 provides an example based on computing a greatest common divisor (GCD). Figure 4.3(a) shows a black-box diagram of the desired system, having x_i and y_i data inputs and a data output d_i . The system's functionality is straightforward: 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 4.3(b) provides a simple program with this functionality. The reader might trace this program's execution on the above examples to verify that the program does indeed compute the GCD.

Figure 4.1: Sequential logic design.



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 arithmetics expressions, and these expressions may use external inputs and outputs or variables. In contrast, our earlier state diagrams only included boolean expressions, and these expressions could only use external inputs and outputs, not

Figure 4.2: Templates for creating a state diagram from a program.



variables. Thus, these more complex state diagram looks like a sequential program in which statements have been scheduled into states.

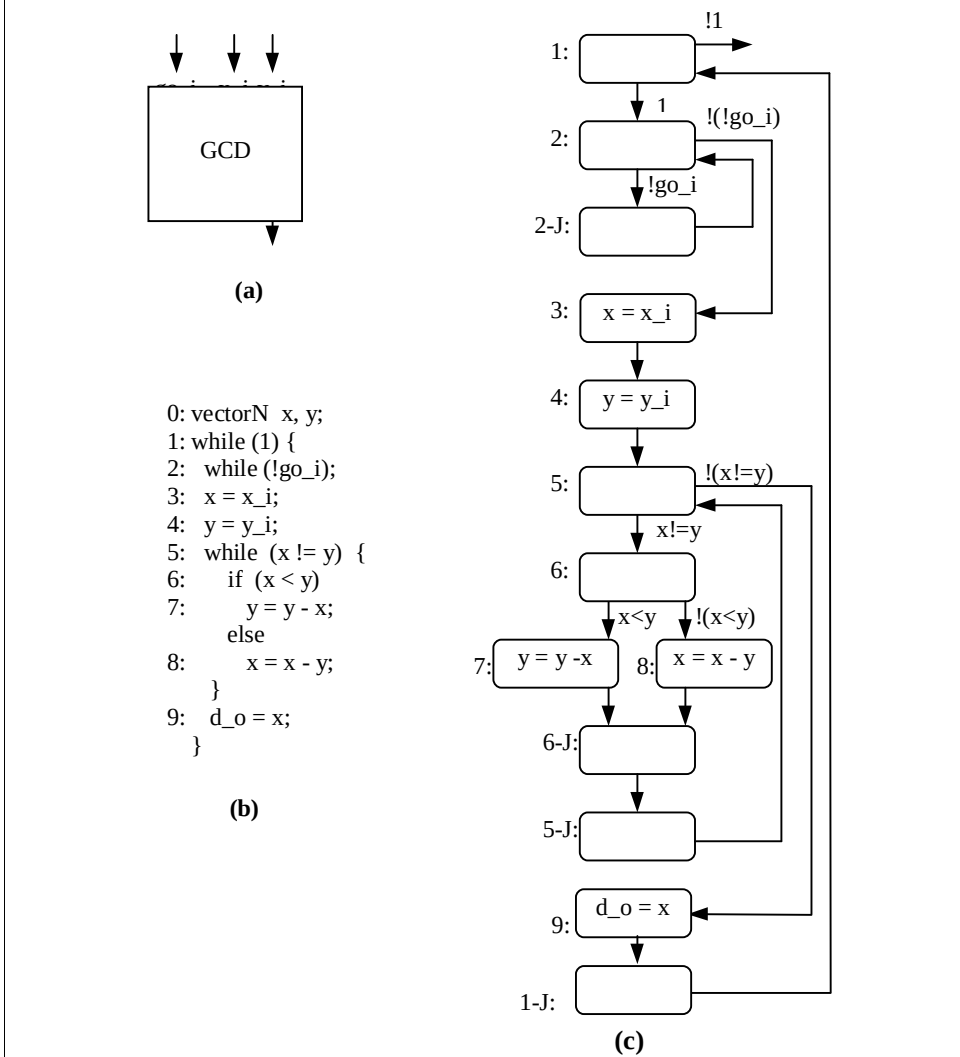
We can use templates to convert a program to a state diagram, as illustrated in Figure 4.2. First, we classify each statement as an assignment statement, loop statement, or branch (if-then-else or case) statement. For an assignment statement, we create a state with that statement as its action. We add an arc from this state to the state for the next statement, whatever type it may be. For a loop statement, we create a condition state *C* and a join state *J*, both with no actions. 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. 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 AND'ed with the second branches condition from the condition state to the branches 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 complex state diagram of Figure 4.3(c).

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 and a controller part, as shown in Figure 4.4. The datapath part should consist of an interconnection of combinational and sequential components. The controller part should consist of a basic state diagram, i.e., one containing only boolean actions and conditions.

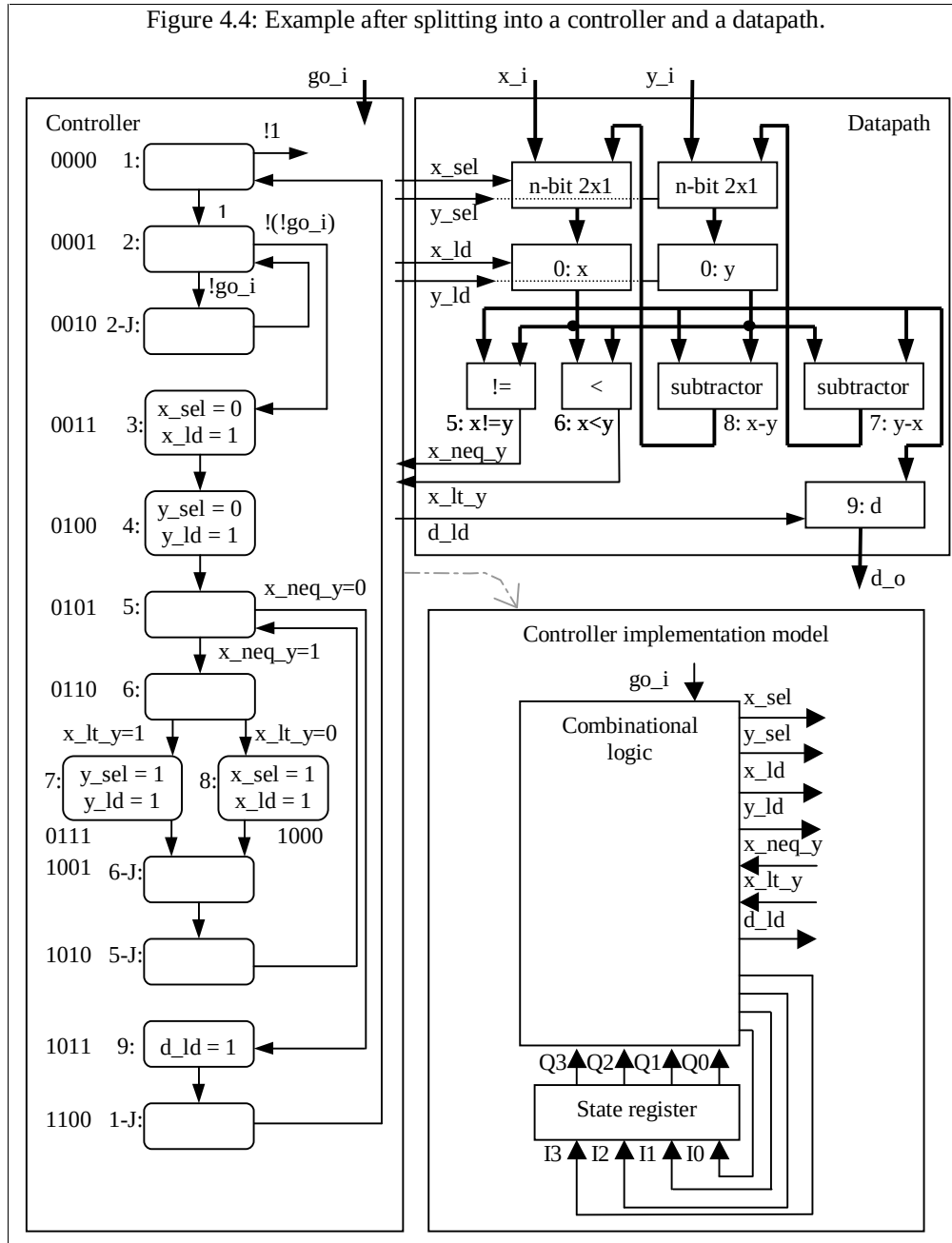
We construct the datapath through a four-step process:

Figure 4.3: Example program -- Greatest Common Divisor (GCD): (a) black-box view, (b) desired functionality, (c) state diagram



1. First, we create a register for any declared variable. In the example, these are x and y . We treat an output port as having an implicit variable, so we create a register d and connect it to the output port. We also draw the input and output ports.
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, as shown in the figure.
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 (an input port, a functional unit, or another register) to the variable's 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.
4. Finally, we create a unique identifier for each control input and output of the datapath components.

Figure 4.4: Example after splitting into a controller and a datapath.



Now that we have a complete datapath, we can build a state diagram for our controller. The state diagram has the same structure as the complex state diagram. 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 select signals of the multiplexor in front of the variable's register's 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.

We can then complete the controller design by implementing the state diagram using our sequential design technique described earlier. Figure 4.4 shows the controller implementation model, and Figure 4.5 shows a state table.

Note that there are 7 inputs to the controller, resulting in 128 rows for the table. We reduced rows in the state table by using don't cares for some input combinations, but we

Figure 4.5: State table for the GCD example.

							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	1	0	X	X	0	0	0
0	0	0	1	*	*	*	1	0	0	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
1	1	1	1	*	*	*	0	0	0	0	X	X	0	0	0

* - indicates all possible combinations of 0's and 1's
X - indicates don't cares

can still see that optimizing the design using hand techniques could be quite tedious. For this reason, computer-aided design (CAD) tools that automate the combinational as well as sequential logic design can be very helpful; we'll introduce such CAD tools in a later chapter.

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 states in the datapath. Such optimizations will be discussed in a later chapter.

Remember that we could alternatively implement the GCD program by programming a microcontroller, thus eliminating the need for this design process, but possibly yielding a slower and bigger design.

4.5 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, 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. Because processors can be complex, CAD tools would be a great designer's aid.

4.6 References and further reading

Gajski, Daniel D. *Principles of Digital Design*. New Jersey: Prentice-Hall, 1997. ISBN 0-13-301144-5. Describes combinational and sequential logic design, with a focus on optimization techniques, CAD, and higher-levels of design.

Katz, Randy. *Contemporary Logic Design*. Redwood City, California: Benjamin/Cummings, 1994. ISBN 0-8053-2703-7. Describes combinational and sequential logic design, with a focus on logic and sequential optimization and CAD.

4.7 Exercises

1. Build a 3-input NAND gate using a minimum number of CMOS transistors.
2. 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.
3. Design a 3-bit counter that counts the following sequence: 1, 2, 4, 5, 7, 1, 2, This counter has an output "odd" that is one 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 logic, and draw the final circuit.
4. 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 4 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 1 clock cycle per instruction (advanced processors use parallelism to meet or exceed 1 cycle per instruction). (d) Compare the estimated gates with 200,000 gates, a typical number of gates for a modern 32-bit processor.
5. Design a custom single-purpose processor implementing the following program, using the technique of the chapter. Start with a complex state diagram, construct a datapath and a simplified state diagram, and draw the truth table for the controller, but do not complete the design for the controller beyond the truth table.

```
input_port U;
int V;
for (int i=0; i<32; i++)
    V = V + U*V;
```