(b) Using a CPLD device consisting of several PLDs from Figure 7.38 and assuming you can connect the PLDs in a custom manner, implement the 2-bit carry-ripple adder by drawing Xs on the PLDs.

(c) Compare the size of your PLD and the CPLD by determining the gates required for both designs (make sure you compare the number of gates within the PLD and CPLD and not the number of gates used for your implementation).

SECTION 7.5: IC TECHNOLOGY COMPARISONS

7.37 For each of the system constraints below, choose the most appropriate technology from among FPGA, standard cell, and full-custom IC technologies for implementing a given circuit. Justify your answers.

(a) The system must exist as a physical prototype by next week.

(b) The system should be as small and low-power as possible. Short design time and low cost are *not* priorities.

(c) The system should be reprogrammable even after the final product has been produced.

(d) The system should be as fast as possible and should consume as little power as possible, subject to being completely implemented in just a few months.

(e) Only five copies of the system will be produced and we have no more than $1000 to spend on all the ICs.

7.38 Which of the following implementations are *not* possible? (1) A custom processor on an FPGA. (2) A custom processor on an ASIC. (3) A custom processor on a full-custom IC. (4) A programmable processor on an FPGA. (5) A programmable processor on an ASIC. (6) A programmable processor on a full-custom IC. Explain your answer.

CS224: A8 Materials

8 Programmable Processors

8.1 INTRODUCTION

Digital circuits designed to perform a single processing task, such as a seat belt warning light, a pacemaker, or an FIR filter, are indeed a very common class of digital circuits. We might refer to a circuit performing a single processing task as a ***single-purpose processor***. Single-purpose processors represent a class of digital circuits enabling tremendously fast or power-efficient computation. However, another class of digital circuits, known as programmable processors, is also extremely popular, as well as being more widely known. The programmable processor is largely responsible for the computing revolution that has taken place in the past several decades, leading to what many call the information age. A ***programmable processor***, also known as a ***general-purpose processor***, is a digital circuit whose particular processing task is stored in a memory, rather than being built into the circuit itself. The representation of that processing task in the memory is known as a ***program***. Figure 8.1 illustrates single-purpose versus general-purpose processors. We could create a custom digital circuit for a seat belt warning light system (Chapter 2) or an FIR filter system (Chapter 5), or instead we could program a general-purpose processor circuit to implement those systems.

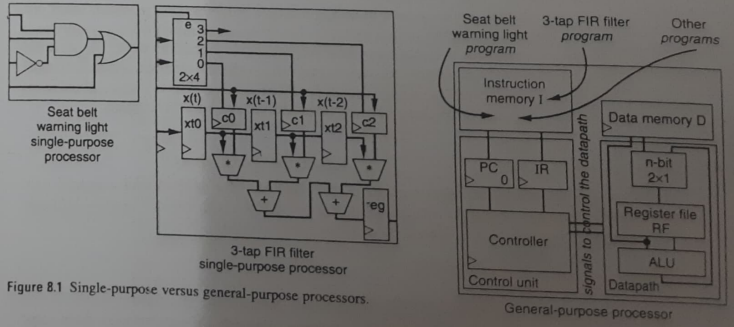

Figure 8.1 Single-purpose versus general-purpose processors.

Some programmable processors, like the well-known Intel Pentium processor or Sun's Sparc processor, are intended for use in desktop computers. Other programmable processors, like ARM, MIPS, 8051, and PIC processors (which are widely known in the design community but less known by the general public), are intended for embedded systems, like cellular telephones, automobiles, video games, or even tennis shoes with blinking lights. Some programmable processors, like the PowerPC, are intended for both desktop and embedded domains.

A benefit of a programmable processor is that its circuit can be mass-produced and then programmed to do almost anything. Thus, the same programmable desktop processor can run Windows 98, Windows XP, Linux, or whatever new operating system program comes about. Likewise, that same processor can run application programs like word processors, spreadsheets, video games, web browsers, etc. Furthermore, the same programmable embedded processor can be used in a cell phone, automobile, video game, or tennis shoe by programming the processor for the desired processing task. Mass-production results in low costs due to amortization of design costs (see "Why such cheap calculators?" in Chapter 4 for a discussion of amortization).

Of course, because programmable processors are mass-produced and then used for a wide variety of applications, there aren't as many unique programmable processor designs as there are single-purpose processor designs. It follows then that there are far fewer programmable processor *designers* than there are single-purpose processor designers. Nevertheless, even though you may never design a programmable processor as part of your job, it is interesting and enlightening to understand how such a programmable processor works. Some people argue that people who understand how a processor works are even better software programmers. And technology trends have led to the situation of designers being able to create semicustom processors ("application-specific" processors) that have just the right architecture for one or a small number of applications, making knowledge of programmable processor designs important. Finally, there are indeed people who do design programmable processor architectures, and you never know if you might end up being one of them.

In this chapter, we show how to design a simple programmable processor using our previously-described digital design methods. Our purpose is mainly to demystify these devices and to provide an intuition of how programmable processors work. We point out that real mass-produced processors are designed using different methods, and their designs can be much more complex than the design described in this chapter—learning about those processors' designs is the subject of many textbooks on computer architecture.

8.2 BASIC ARCHITECTURE

A programmable processor consists of two main parts: a datapath and a control unit. We'll provide a general introduction to those two parts in this section, then we'll provide a more detailed look at those parts in a subsequent section.

Basic Datapath

We can view processing generally as:

- *Loading* data, meaning reading the data on which we wish to work from some input locations,

- *Transforming* that data, meaning performing some computations with that data that result in new data, and
- *Storing* the new data, meaning writing the new data to some output locations.

For example, a seat belt warning system reads bit data from sensors representing whether a seat belt is fastened and whether a person is sitting in a seat, transforms that data by computing a new bit indicating whether to turn on a warning light, and writes that new data to a warning light. An FIR filter reads data representing the most recent set of input signal samples, transforms that data by performing multiplies and adds, and writes new data to an output representing the filtered signal.

A ***data memory*** holds all the data that a programmable processor can access, as input data or output data—for now, assume the words in that data memory are somehow connected to the outside world (e.g., to the seat belt sensors or to the FIR input and output signals). To process that data, a programmable processor needs to be able to *load* data from data memory into one of several registers (typically a register file) within the processor, needs to be able to feed data from some subset of registers through functional units that can perform all possible *transformation* operations (typically an ALU) we might consider with results stored back into a register, and needs to be able to *store* data from a register back into data memory. Therefore, we see the need for a programmable processor to include the basic circuit shown in Figure 8.2, showing a data memory, register file, and ALU. That circuit is known as the programmable processor's ***datapath***. The basic datapath shown in Figure 8.2 can perform the following possible ***datapath operations*** in a given clock cycle:

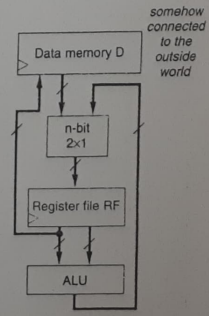

Figure 8.2 Basic datapath of a programmable processor.

- ***Load operation:*** This operation loads (reads) data from any location in the data memory into any register in the register file. A load operation is illustrated in Figure 8.3(a).
- ***ALU operation:*** This operation transforms register data by passing any two registers through the ALU configured for any of the ALU's supported operations, and back into any register of the register file. An ALU operation is illustrated in Figure 8.3(b). Typical ALU operations include addition, subtraction, logical AND, logical OR, etc.
- ***Store operation:*** This operation stores (writes) data from any register in the register file to any data memory location. A store operation is illustrated in Figure 8.3(c).

These possible datapath operations are illustrated in Figure 8.3. Each such operation requires the appropriate setting of the control inputs of the data memory, mux, register file, and ALU—those control inputs will be shown shortly. For now, just familiarize

yourself with the basic datapath's abilities. Notice that the datapath in Figure 8.2 cannot directly operate on data memory locations with the ALU in one clock cycle, because the data must first be read into the register file, which itself requires a clock cycle, before the data can be operated on by the ALU. A datapath that requires all data to first pass through the register file before that data can be transformed by the ALU is known as a ***load-store architecture***.

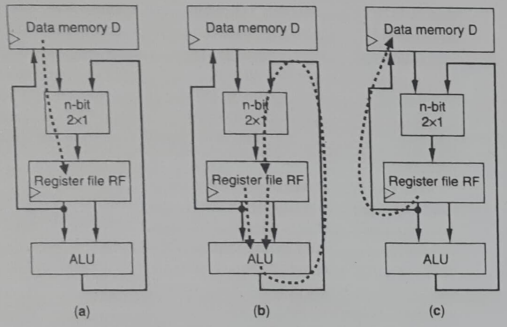

Figure 8.3 Basic datapath operations: (a) load (read), (b) ALU operation (transform), and (c) store (write).

EXAMPLE 8.1 Understanding datapath operations

Which of the following are valid single-clock-cycle datapath operations for the datapath of Figure 8.2?

1. Copy data from a data memory location into a register file location.
2. Read data from two data memory locations into two register file locations.
3. Add data from two data memory locations and store the result in a register file location.
4. Copy data from one register file location to another register file location.
5. Subtract data in a register file location from a data memory location, storing the result in a register file location.

(1) is a valid operation, known as a load operation. (2) is *not* a valid operation. We cannot read more than one data memory location during a datapath operation (for this datapath), and we cannot write to more than one register file location during an operation. (3) is *not* a valid operation. Not only can we not read from two data memory locations during one operation, but we cannot feed the read values directly into the ALU to perform the add—we must first perform operations that read the data items into register file locations. (4) is a valid operation. We can configure the ALU operation to simply pass one of its inputs through to the output (perhaps by adding 0) and store the result in the register file. (5) is *not* a valid operation. We cannot feed a read data memory location directly to the ALU—there is no such connection in the datapath. Values read from data memory must be loaded into the register file first.

Basic Control Unit

Suppose we want to use the basic datapath of Figure 8.2 to perform the simple processing task of adding data memory locations 0 and 1 together, and writing the result in data memory location 9—in other words, we want to compute *D[9] = D[0] + D[1]*. We can achieve this processing task by "instructing" the datapath to perform the following operations:

- *load* datapath memory location 0 to register file register *R0* (i.e., *RF[0] = D[0]*),
- *load* datapath memory location 1 to register file register *R1* (i.e., *RF[1] = D[1]*),
- perform an *ALU* operation that adds *R0* and *R1* and writes the result back into *R2* (i.e., *RF[2] = RF[0] + RF[1]*), and
- *store R2* into data memory location 9 (i.e., *D[9] = RF[2]*).

Note that we could have used any registers in the register file, rather than *R0*, *R1*, and *R2*.

If *D[0]* contained the value 99 (in binary, of course), and *D[1]* contained the value 102, then after carrying out the above operations, *D[9]* would contain 201.

You might think that having to instruct the datapath to perform four distinct operations is a rather cumbersome way of adding two data items. If you could build your own custom digital circuit to implement *D[9] = D[0] + D[1]*, you would likely just feed *D[0]* and *D[1]* through an adder whose output you would connect to *D[9]*, thus avoiding the four operations involving the register file and ALU. We see the basic tradeoff of single-purpose versus programmable processors—programmable processors have the drawback of computation overhead because they have to be general, but they provide the benefits of a mass-produced processor that can be programmed to do almost anything.

Somehow we need to describe the sequence of operations—*RF[0] = D[0]*, then *RF[1]=D[1]*, then *RF[2] = RF[0] + RF[1]*, then *D[9] = RF[2]*—that we desire to execute on the datapath. Such a description of desired processor operations are known as ***instructions***, and a collection of instructions is known as a ***program***. We will store the desired program as words in another memory, called the ***instruction memory***. We'll describe how to represent those instructions later. For now, assume that the four instructions are somehow stored in locations 0, 1, 2, and 3 of the instruction memory *I*, as shown in Figure 8.4.

Now is where the control unit plays a role. The ***control unit*** reads each instruction from instruction memory, and then executes that instruction on the datapath. To execute our simple program, the control unit would begin by performing the following tasks, known as ***stages***, to carry out the first instruction:

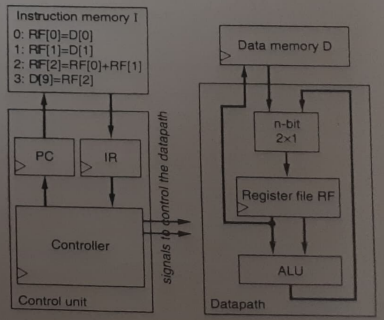

Figure 8.4 The control unit in a programmable processor.

1. *Fetch*: The control unit would start by reading *I[0]* into a local register, a task known as ***fetching***. This stage requires one clock cycle.
2. *Decode*: The control unit would then determine what operation this instruction is requesting, a task known as ***decoding***. This stage also requires one clock cycle.
3. *Execute*: Seeing that this instruction requests the datapath operation *RF[0] = D[0]*, the control unit would set the control lines of the datapath to read *D[0]*, pass the read data through the 2x1 mux in front of the register file, and write that data into *R[0]*. The task of carrying out the operation is known as ***executing***. Most operations are datapath operations (such as a load operation, ALU operation, or store operation), but not all operations require the datapath (an example is the jump instruction to be discussed later). This stage requires one clock cycle.

Thus, the basic stages the control unit carries out for that first instruction are: *fetch*, *decode*, and *execute*, requiring three clock cycles to complete just that first instruction.

The local register in which the control unit stores the fetched instruction is known as the ***instruction register***, or ***IR***, as shown in Figure 8.4. Notice that the control unit needs to keep track of the location in instruction memory from which to fetch the next instruction. Since the instruction locations are usually in sequence, we can use a simple up-counter to keep track of the current program instruction—such a counter is known as a ***program counter***, or ***PC*** for short. The processor starts with *PC=0*, so the instruction in *I[0]* represents the first instruction of the program.

Figure 8.5 illustrates the three stages of executing the instruction *RF[0] = D[0]* stored in *I[0]*. Assuming PC was previously initialized to 0, Figure 8.5(a) shows the first

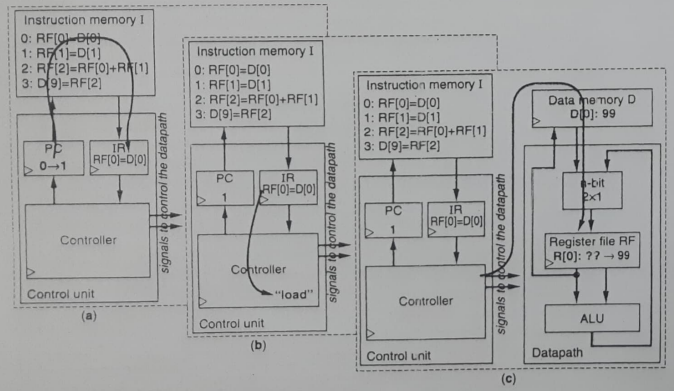

Figure 8.5 Three stages of processing *one* instruction: (a) fetch, (b) decode, (c) execute.

stage fetching *I[0]*'s contents, the instruction *RF[0]=D[0]*, into *IR*. Figure 8.5(b) shows the second state decoding the instruction and thus determining that the instruction is a load instruction. Figure 8.5(c) shows the controller executing the instruction by configuring the datapath to read the value of *D[0]* and storing that value into *RF[0]*. If *D[0]* contained 99, then *R[0]* will contain 99 after completion of the execute stage.

After processing the instruction in *I[0]*, the control unit would fetch the instruction that is in *I[1]*, decode that instruction, and execute that instruction (thus executing *RF[1] = D[1]*), requiring another three cycles. Next, the control unit would fetch the instruction that is in *I[2]*, decode that instruction, and execute that instruction (thus executing *RF[2] = RF[0] + RF[1]*), requiring another three cycles. Finally, the control unit would fetch the instruction that is in *I[3]*, decode that instruction, and execute that instruction (thus executing *D[9] = RF[2]*), requiring another three cycles. The four instructions would require 4*3 = 12 cycles to run to completion on the programmable processor.

The control unit will require a controller, like those described in Chapter 3, that in this case repeatedly performs the fetch, decode, and execute steps (after having initialized *PC* to 0)—note that a controller appears inside the control unit in Figure 8.4. An FSM for that controller appears in Figure 8.6. The controller increments the program counter after fetching each instruction in state *Fetch*, so that the next fetch state will fetch the next instruction (notice that *PC* gets incremented at the end of the fetch stage in Figure 8.5(a)). We'll describe the actions of the *Decode* and *Execute* states later.

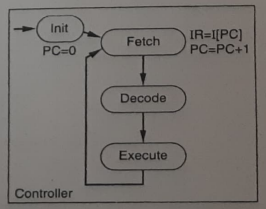

Figure 8.6 Basic controller states.

Thus, the basic parts of the control unit include the program counter *PC*, the instruction register *IR*, and a controller, as illustrated in Figure 8.4. In previous chapters, our nonprogrammable processors consisted only of a controller and a datapath. Notice that the programmable processor instead contains a control unit, which itself consists of some registers and a controller.

To summarize, the control unit processes each instruction in three stages:

1. first *fetching* the instruction by loading the current instruction into *IR* and incrementing the *PC* for the next fetch,
2. next *decoding* the instruction to determine its operation, and
3. finally *executing* the operation by setting the appropriate control lines for the datapath, if applicable. If the operation is a datapath operation, the operation may be one of three possible types:
   (a) *loading* a data memory location into a register file location,
   (b) transforming data using an *ALU* operation on register file locations and writing results back to a register file location, or
   (c) *storing* a register file location into a data memory location.

EXAMPLE 8.2 Creating a simple sequence of instructions

Create a set of instructions for the processor in Figure 8.4 to compute *D[3] = D[0] + D[1] + D[2]*. Each instruction must represent a valid single-clock-cycle datapath operation.

We might start with three operations that read the data memory locations into register file locations:

0. *R[3] = D[0]*
1. *R[4] = D[1]*
2. *R[2] = D[2]*

Note that we intentionally chose arbitrary register locations, to make clear that we can use any registers.

Next, we need to add the three values and store the result in a register file location, say *R[1]*. In other words, we want to perform the following operation: *R[1] = R[2] + R[3] + R[4]*. However, the datapath of Figure 8.4 cannot add three register file locations in a single operation, but rather can only add two locations. Instead, we can describe the desired addition computation by dividing the computation into two datapath operations:

3. *R[1] = R[2] + R[3]*
4. *R[1] = R[1] + R[4]*

Finally, we write the result into *D[3]*:

5. *D[3] = R[1]*

Thus, our program consists of the six instructions appearing above, which we might store in instruction memory locations 0 through 5.

EXAMPLE 8.3 Evaluating the time to carry out a program

Determine the number of clock cycles required for the processor of Figure 8.4 to execute the six-instruction program of Example 8.2.

The processor requires 3 cycles to process each instruction: 1 cycle to fetch the instruction, 1 to decode the fetched instruction, and 1 to execute the instruction. At 3 cycles per instruction, the total cycles for 6 instructions is: 6 instr * 3 cycles/instr = 18 cycles.

8.3 A THREE-INSTRUCTION PROGRAMMABLE PROCESSOR

A First Instruction Set with Three Instructions

The way we represent instructions in the instruction memory, and the list of allowable instructions, are known as a programmable processor's ***instruction set***. Let's assume that a processor uses 16-bit instructions, and that the instruction memory *I* is 16-bits wide. Instruction sets typically reserve a certain number of bits in the instruction to denote what operation to perform. The remaining bits specify any additional information needed to perform the operation, such as the source or destination registers. We define a simple, three-instruction set, with the most significant (meaning leftmost) 4 bits identifying the appropriate operation and the least significant 12 bits containing register file and data memory addresses, as follows:

- *Load* instruction—**0000 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$**: This instruction specifies a move of data from the data memory location whose address is specified by the bits $d_7d_6d_5d_4d_3d_2d_1d_0$ into the register file register whose location is specified by the bits $r_3r_2r_1r_0$. For example, the instruction "0000 0000 00000000" specifies a move of data memory location 0, or *D[0]*, into register file location 0, or *RF[0]*—in other words, that instruction represents the operation *RF[0]=D[0]*. Likewise, "0000 0001 00101010" specifies *RF[1]=D[42]*. We've inserted spaces between some bits for ease of reading by you the reader—those spaces have no other significance and would not exist in the instruction memory.
- *Store* instruction—**0001 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$**: This instruction specifies a move of data in the opposite direction as the instruction above, meaning a move from the register file to the data memory. So "0001 0000 00001001" specifies *D[9]=RF[0]*.
- *Add* instruction—**0010 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**: This instruction specifies an addition of two register file registers specified by $rb_3rb_2rb_1rb_0$ and $rc_3rc_2rc_1rc_0$, with the result stored in the register file register specified by $ra_3ra_2ra_1ra_0$. For example, "0010 0010 0000 0001" specifies the instruction *RF[2]=RF[0]+RF[1]*. Note that *add* is an ALU operation.

None of these instructions modifies the contents of the instructions' source operands. In other words, the *load* instruction copies the contents of the data memory location to the specified register, but leaves the data memory location itself unchanged. Likewise, the *store* instruction copies the specified register to data memory, but leaves the register's contents unchanged. The *add* instruction reads its *b* and *c* registers without changing them.

Using this instruction set, we would describe our earlier program that computes *D[9]=D[0]+D[1]* as shown in Figure 8.7.

Notice that the first four bits of each instruction are a binary code that indicates the instruction's operation. Those bits are known as the instruction's *operation code*, or ***opcode*** for short. "0000" means a move from data memory to register file, "0001" means a move from register file to data memory, and "0010" means an add of two registers, based on the instruction set defined in the bulleted list above. The remaining bits of the instruction represent ***operands***, which indicate what data to operate on.

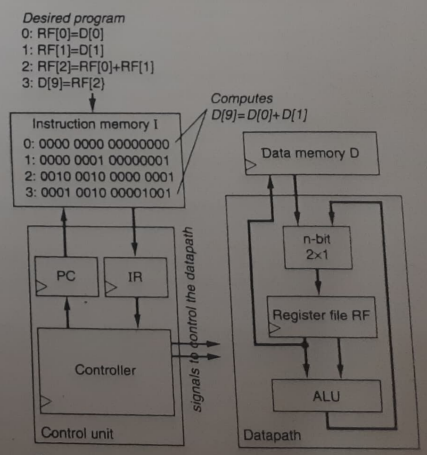

Figure 8.7 A program that computes D[9]=D[0]+D[1], using a given instruction set. We've inserted spaces between the instruction memory's bits for readability only—those spaces don't exist in the memory.

We could write a different program using the same three-instruction instruction set. For example, we could write a program that computes *D[5] = D[5] + D[6] + D[7]*. We must perform that computation using instructions chosen from the three-instruction instruction set. We might write the program as shown in Figure 8.8. The number before the colon represents the instruction's address in the instruction memory *I*. The text following the two forward slashes (*//*) represent comments, and are not part of the instructions.

```
0: 0000 0000 00000101 // RF[0] = D[5]
1: 0000 0001 00000110 // RF[1] = D[6]
2: 0000 0010 00000111 // RF[2] = D[7]
3: 0010 0000 0000 0001 // RF[0] = RF[0] + RF[1]
                       // which is D[5]+D[6]
4: 0010 0000 0000 0010 // RF[0] = RF[0] + RF[2]
                       // now D[5]+D[6]+D[7]
5: 0001 0000 00000101 // D[5] = RF[0]
```

Figure 8.8 A program to compute *D[5]=D[5]+D[6]+D[7]* using the three-instruction instruction set.

Note how that program ultimately computes the desired sum. This might be the first time that you have had to think of computations in terms of low-level programmable processor instructions. Thinking in terms of such register-level operations can be difficult at first, but becomes easier as you see and develop more programs at that level.

Machine Code versus Assembly Code

As you have seen, the instructions of a program exist in instruction memory as 0s and 1s. A program represented as 0s and 1s is known as ***machine code***. Writing and reading programs represented as 0s and 1s are tasks that humans are not particularly good at. We humans can't understand those 0s and 1s easily, and thus will likely make plenty of mistakes when writing such programs. Thus, early computer programmers developed a tool, known as an ***assembler*** (which itself is just another program), to help humans write other programs. An assembler allows us to write instructions using ***mnemonics***, or symbols, that the assembler automatically translates to machine code. Thus, an assembler may tell us that we can write instructions from our three-instruction instruction set using the following mnemonics:

- *Load* instruction—**MOV Ra, d**: specifies the operation *RF[a]=D[d]*. *a* must be 0, 1, ... or 15—so *R0* means *RF[0]*, *R1* means *RF[1]*, etc. *d* must be 0, 1, ..., 255.
- *Store* instruction—**MOV d, Ra**: specifies the operation *D[d]=RF[a]*.
- *Add* instruction—**ADD Ra, Rb, Rc**: specifies the operation *RF[a]=RF[b]+RF[c]*.

▶ COMPUTERS WITH BLINKING LIGHTS.

Big computers shown in the movies often have many rows of small blinking lights. In the early days of computing, computer programmers programmed using machine code, and they entered that code into the instruction memory by flipping switches up and down to represent 0s and 1s. To enable debugging of the program, as well as to show the computed data, those early computers used rows of lights—on lights meant 1s, off lights meant 0s. Today, nobody in their right mind would try writing or debugging a program by using machine code. So computers today look like big boxes—with no rows of lights. But big plain boxes don't make for interesting backgrounds in movies, so movie makers continue to use movie props with lots of blinking lights to represent computers—lights that are useless, but entertaining.

▶ "BOOTING" A COMPUTER.

Turning on a personal computer causes the operating system to load, a process known as "booting" the computer. The computer executes instructions beginning at address 0, which usually has an instruction that jumps to a built-in small program that loads the operating system (the small program is often called the basic input/output system, or BIOS). Most computing dictionaries state that the term "boot" originates from the popular expression "to pull oneself up by one's bootstraps," which means to pick yourself up without any help, though obviously you can't do this by grabbing onto your own bootstraps and pulling—hence the cleverness of the expression. Since the computer loads its own operating system, the computer is in a sense picking itself up without any help. The term bootstrap eventually got shortened to boot. A colleague of mine who has been around computing a long time claims a different origin. One way of loading a program into the instruction memory of early computers was to create a ribbon with rows of holes. Each row might have enough room for say 16 holes, thus each row would represent a 16-bit machine instruction—a hole meant a 0, no hole a 1 (or vice versa). A programmer would punch holes in the ribbon to store the program on the ribbon (using a special hole-punching machine), and then feed the ribbon into a computer's ribbon reader, which would read the rows of 0s and 1s and load those 0s and 1s into the computer's instruction memory. Those ribbons might have been several feet long, and looked a lot like the straps of a boot, hence the term bootstrap, shortened to boot. Whichever is the actual origin, we can be fairly sure the term "boot" comes from the bootstraps on the boots we wear on our feet.

Using those mnemonics, we could rewrite the program *D[9]=D[0]+D[1]* as follows:

```
0: MOV R0, 0
1: MOV R1, 1
2: ADD R2, R0, R1
3: MOV 9, R2
```

That program is much easier to understand than the 0s and 1s in Figure 8.7. A program written using mnemonics that will be translated to machine code by an assembler is known as ***assembly code***. Hardly anybody writes machine code directly these days. An assembler would automatically translate the above assembly program to the machine code shown in Figure 8.7.

You might be wondering how the assembler can distinguish between the load and store instructions above, when the mnemonic for both instructions is the same—"MOV." The assembler distinguishes those two types of instructions by looking at the first character after the mnemonic "MOV"—if the first character is an "R," then that operand is a register, and thus that instruction must be a load instruction.

Control Unit and Datapath for the Three-Instruction Processor

From the definition of the three-instruction instruction set and an understanding of the basic control unit and datapath architecture of a programmable processor as shown in Figure 8.4, we can design a complete digital circuit for a three-instruction programmable processor. The design process is actually very similar to the RTL design process of Chapter 5.

We begin with a high-level state machine description of the system, shown in Figure 8.9. Assume that *op* is shorthand for *IR[15..12]*, meaning the leftmost four bits of the instruction register. Likewise, assume that *ra* is shorthand for *IR[11..8]*, *rb* is shorthand for *IR[7..4]*, *rc* is shorthand for *IR[3..0]*, and *d* is shorthand for *IR[7..0]*.

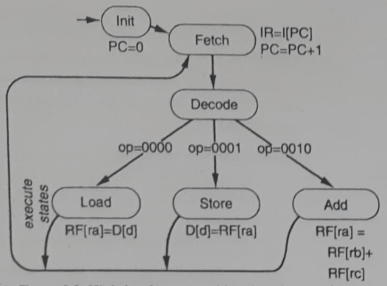

Figure 8.9 High-level state machine description of a three-instruction programmable processor.

Recall that the next step in the RTL design process was to create the datapath. We already created the datapath in Figure 8.4, which we refine to show every control signal from the controller, as shown in Figure 8.10. The refined datapath has control signals for each read and write port of the register file (see Chapter 4 for information on register files). The register file has 16 registers because the instructions have only 4 bits with which to address registers. The datapath has a control signal to the ALU called `alu_s0`—we'll assume the simple ALU adds its inputs when `alu_s0=1`, and just passes input *A* when `alu_s0=0`. The datapath has a select line for the

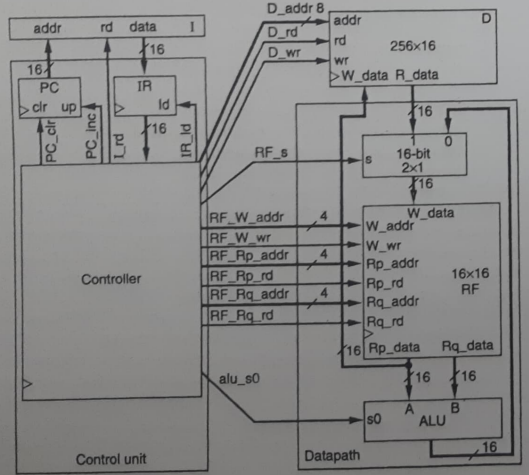

Figure 8.10 Refined datapath and control unit for the three-instruction processor.

2x1 mux in front of the register file's write data port. Finally, we have also included the control signals for the data memory, which we assume has a single address port, and can thus support only a read or a write, but not both simultaneously. The data memory has 256 words, since the instruction only has 8 bits with which to address the data memory.

The datapath is now able to carry out all of the load/store operations and arithmetic operations that we need for the high-level state machine from Figure 8.9. Thus, we can proceed to the third step of the RTL design process of connecting the datapath with a controller. Figure 8.10 shows those connections, as well as the connections of the controller to the *PC* and *IR* registers in the control unit, and to the instruction memory *I*.

The last step of the RTL design process is to derive the controller's FSM. We can do this straightforwardly by replacing the high-level actions of the state machine in Figure 8.9 by Boolean operations on the controller's input and output lines, as shown in Figure 8.11. (Remember that *op*, *d*, *ra*, *rb*, and *rc* are shorthand notations for *IR[15..12]*, *IR[7..0]*, *IR[11..8]*, *IR[7..4]*, and *IR[3..0]*, respectively.) We could then finish the controller's design by converting the FSM to a state register and combinational logic, using the methods from Chapter 3.

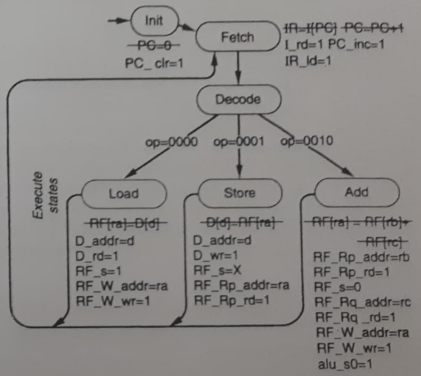

Figure 8.11 FSM for the three-instruction processor's controller.

We would have thus designed a programmable processor.

Let's trace through the controller's FSM behavior to see how a program would execute on the three-instruction processor. As a reminder, remember that we follow the FSM conventions that all transitions are implicitly ANDed with a rising clock edge, and that any control signal not explicitly assigned a value in a state is implicitly assigned a 0.

- The FSM initially starts in state *Init*, which sets `PC_clr=1`, causing the *PC* register to be cleared to 0.
- The FSM on the next clock cycle enters the *Fetch* state, in which the FSM reads the instruction memory at address 0 (because *PC* is 0) and loads the read value into *IR*—that read value will be the instruction that was stored in *I[0]*. At the same time, the FSM increments the *PC*'s value.
- The FSM on the next clock cycle enters the *Decode* state, which has no actions but which branches on the next clock cycle to one of three states, *Load*, *Store*, or *Add*, depending on the values of the highest four bits of the *IR* register (the current instruction's opcode).

- In the *Load* state, the FSM sets the data memory address lines to the low eight bits of the *IR* and sets the data memory read enable to 1, sets the 2x1 mux's select line to pass the data memory output to the register file, and sets the register file write address to *IR[11..8]* and the write enable to 1, causing whatever gets read from the data memory to be loaded into the appropriate register in the register file.
- Likewise, the *Store* and *Add* states set the control lines as needed for the store and add operations.
- Finally, the FSM returns to the *Fetch* state, and begins fetching the next instruction.

Notice that because the *Store* state does not write to the register file, then the value of the register file's mux select lines don't matter, so we've assigned signal RF_s=X in that state, meaning the signal's value does not matter. Using such don't care values (see Section 6.2) can help us to minimize logic in the controller.

You may wonder why the *Decode* state is necessary when that state contains no actions—could we not have just had *Decode*'s transitions originate instead from state *Fetch*? Recall from Section 5.3 that register updates listed in a state do not actually occur until the next clock edge, meaning that transitions originating from a state use the previous register values. Thus, we could not have originated *Decode*'s transitions from the *Fetch* state, because those transitions would have been using the old opcode in the instruction register *IR*, not the new value read during the *Fetch* state.

8.4 A SIX-INSTRUCTION PROGRAMMABLE PROCESSOR

Extending the Instruction Set

Clearly, having only a three-instruction instruction set limits the behavior of the programs that we can write. All we can do with those instructions is add numbers. A real programmable processor will support many more instructions, perhaps 100 or more, so that a wider variety of programs can be written.

Let's extend our programmable processor's instruction set with a few more instructions, in order to give you a slightly better idea of how a programmable processor with a full instruction set would look.

We'll begin by introducing an instruction able to load a constant value into a register file register. For example, suppose we wanted to compute *RF[0] = RF[1] + 5*. The 5 is a constant. A ***constant*** is a value that is part of our program, not something to be found in data memory. We need an instruction that allows us to load a constant into a register, after which we could add that register to *RF[1]* using the ADD instruction. Thus, we introduce a new instruction with the following machine and assembly code representations:

- ***Load-constant*** instruction—**0011 $r_3r_2r_1r_0$ $c_7c_6c_5c_4c_3c_2c_1c_0$**: specifies that the binary number represented by the bits $c_7c_6c_5c_4c_3c_2c_1c_0$ should be loaded into the register specified by $r_3r_2r_1r_0$. The binary number being loaded is known as a *constant*. The mnemonic for this instruction is:

  MOV Ra, #c—specifies the operation *RF[a]=c*

a can be 0, 1, ..., or 15. Assuming two's complement representation (see Section 4.8), *c* can be −128, −127, ..., 0, ..., 126, 127. The "#" enables the assembler to distinguish this instruction from a regular load instruction.

We continue by introducing an instruction for performing subtraction of two registers, similar to addition of two registers, having the following machine and assembly code representations:

- ***Subtract*** instruction—**0100 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**: specifies subtraction of two register file registers specified by $rb_3rb_2rb_1rb_0$ and $rc_3rc_2rc_1rc_0$, with the result stored in the register file register specified by $ra_3ra_2ra_1ra_0$. For example, "0100 0010 0000 0001" specifies the instruction *RF[2]=RF[0] − RF[1]*. The mnemonic for this instruction is:

  SUB Ra, Rb, Rc—specifies the operation *RF[a]=RF[b] − RF[c]*

Let's also introduce an instruction that allows us to jump to other parts of a program:

- ***Jump-if-zero*** instruction—**0101 $ra_3ra_2ra_1ra_0$ $o_7o_6o_5o_4o_3o_2o_1o_0$**: specifies that if the contents of the register specified by $ra_3ra_2ra_1ra_0$ is 0, we should load the *PC* with the current value of *PC* plus $o_7o_6o_5o_4o_3o_2o_1o_0$, which is an 8-bit number in two's complement form representing a positive or negative offset amount. The mnemonic is:

  JMPZ Ra, offset—specifies the operation *PC = PC + offset* if *RF[a]* is 0.

By using two's complement for the jump offset, which allows representation of positive or negative numbers, the program can jump backwards in the program, thus implementing a loop. With an 8-bit offset, the instruction can specify a jump forward by 127 addresses, or backward by 128 addresses (-128 to +127).

Table 8.1 summarizes the six-instruction instruction set. A programmable processor typically comes with a databook that lists the processor's instructions, and the meaning of each instruction, using a format similar to the format of Table 8.1. Typical programmable processors have dozens, even hundreds, of instructions.

TABLE 8.1 Six-instruction instruction set.

Instruction	Meaning
MOV Ra, d	RF[a] = D[d]
MOV d, Ra	D[d] = RF[a]
ADD Ra, Rb, Rc	RF[a] = RF[b]+RF[c]
MOV Ra, #C	RF[a] = C
SUB Ra, Rb, Rc	RF[a] = RF[b]-RF[c]
JMPZ Ra, offset	PC=PC+offset if RF[a]=0

Extending the Control Unit and Datapath

The three new instructions require some extensions to our control unit and datapath of Figure 8.10, with those extensions shown in Figure 8.12. First, the *load constant* instruction requires that the register file be able to load data from *IR[7..0]*, in addition to data from data memory or the ALU output. Thus, we widen the register file's multiplexer from 2x1 to 3x1, add another mux control signal, and also create a new signal coming from the controller labeled *RF_W_data*, which will connect with *IR[7..0]*—these changes are highlighted by the dashed circle labeled "*1*" in Figure 8.12. Second, the subtract

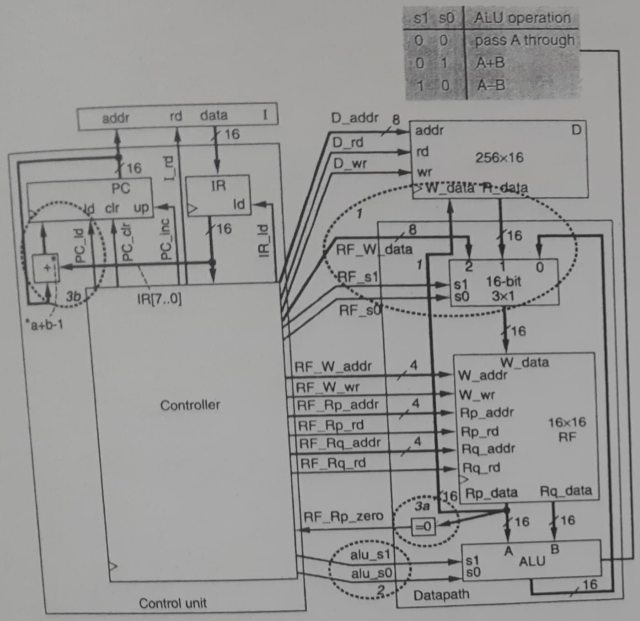

Figure 8.12 Control unit and datapath for the six-instruction processor.

instruction requires that we use an ALU capable of subtraction, so we add another ALU control signal—highlighted by the dashed circle labeled "2" in the figure. Third, the jump-if-zero instruction requires that we be able to detect if a register is zero, and that we be able to add *IR[7..0]* to the *PC*. Thus, we insert a datapath component to detect if the register file's *Rp* read port is all zeros (that component would just be a NOR gate), labeled as dashed-circle "*3a*" in the figure. We also upgrade the *PC* register so it can be loaded with *PC* plus *IR[7..0]*, labeled as "*3b*" in the figure. The adder used for this also subtracts 1 from the sum, to compensate for the fact that the *Fetch* state already added 1 to the *PC*.

We also need to extend the FSM for the controller within the control unit to handle the three additional instructions. Figure 8.13 shows the extended FSM. The *Init* and *Fetch* states stay the same. We added three new transitions from the *Decode* state for the three new instruction opcodes. We made a minor revision to the *Load*, *Store*, and *Add* states' actions (the new actions are italicized) since the register file mux has a mux with two select lines instead of just one. Likewise, we revised the *Add* state actions to configure the ALU with two control lines instead of one. We added four new states, *Load-constant*, *Subtract*,

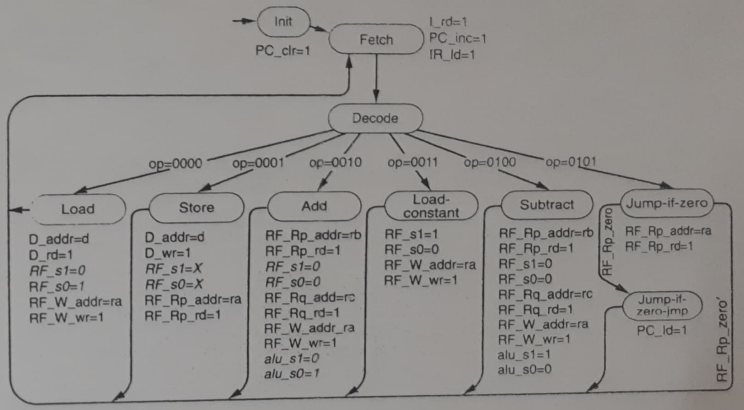

Figure 8.13 Control unit and datapath for the six-instruction processor.

Jump-if-zero, and *Jump-if-zero-jmp*, for the three new instructions. The new instruction states perform the following functions on the datapath:

- In the *Load-constant* state, we configure the register file mux to pass the `RF_W_data` signal, and we configure the register file to write to the address specified by *ra* (which is *IR[11..8]*).
- In the *Subtract* state, we perform the same actions as in the *Add* state, except that we configure the ALU for subtraction instead of addition.
- In the *Jump-if-zero* state, we configure the register file to read the register specified by *ra* onto read port *Rp*. If the value of the read register *Rp* is all 0s, `RF_Rp_zero` will become 1 (and 0 otherwise). Thus, we include two transitions from the *Jump-if-zero* state. One transition will be taken if `RF_Rp_zero` is 0, meaning the read register was not all 0s—that transition takes the FSM back to the *Fetch* state, meaning no actual jump occurs. The other transition will be taken if `RF_Rp_zero` is 1, meaning the read register was all 0s. That transition goes to another state, *Jump-if-zero-jmp*, which should actually carry out the jump. That state carries out the jump simply by setting the load line of the *PC*.

Notice that with the addition of a *Jump-if-zero* instruction, the processor may take up to four cycles to complete an instruction. Namely, when the *ra* register of a *Jump-if-zero* instruction is all 0s, then an extra state is needed to load the *PC* with the address of the instruction to which to jump.

8.5 EXAMPLE ASSEMBLY AND MACHINE PROGRAMS

Using the six-instruction instruction set of the previous section, we now provide an example of assembly-language programming using the six-instruction processor to perform a particular task, and we show how the assembly code would be converted to machine code by an assembler. An assembler would make use of the table shown in Table 8.2, which maps instructions to opcodes.

TABLE 8.2 Instruction opcodes.

Instruction	Opcode
MOV Ra, d	0000
MOV d, Ra	0001
ADD Ra, Rb, Rc	0010
MOV Ra, #C	0011
SUB Ra, Rb, Rc	0100
JMPZ Ra, offset	0101

EXAMPLE 8.4 Assembly and machine programs for a simple program

Write a program that counts the number of words that are not equal to 0 in data memory locations 4 and 5, and that stores the result in data memory location 9. Thus, the possible results that would be stored in location 9 are zero, one, or two.

Using the instruction set of Table 8.2, we can write an assembly program as shown in Figure 8.14(a). The program maintains the count in register *R0*, which the program initializes to 0. The program may need to add 1 to this register later, so the program loads the value 1 into register *R1*. The program next loads data memory location 4 into register *R2*. The program then jumps to the instruction labeled as "lab1" if the value of *R2* is zero. If *R2* is not zero, the program will execute an add instruction that adds one to register *R0*, and will then proceed to the instruction labeled "lab1" since that instruction is the next instruction. The instruction labeled "lab1" loads data

```
    MOV R0, #0; // initialize result to 0          0011 0000 00000000
    MOV R1, #1; // constant 1 for incrementing result  0011 0001 00000001
    MOV R2, 4; // get data memory location 4        0000 0010 00000100
    JMPZ R2, lab1; // if zero, skip next instruction  0101 0010 00000010
    ADD R0, R0, R1; // not zero, so increment result  0010 0000 0000 0001
lab1:MOV R2, 5; // get data memory location 5        0000 0010 00000101
    JMPZ R2, lab2; // if zero, skip next instruction  0101 0010 00000010
    ADD R0, R0, R1; //not zero, so increment result  0010 0000 0000 0001
lab2:MOV 9, R0; // store result in data memory location 9  0001 0000 00001001
```

(a) (b)

Figure 8.14 A program to count the number of nonzero numbers in *D[4]* and *D[5]*, storing the result in *D[9]*: (a) assembly code, and (b) corresponding machine code generated by an assembler. The spaces in the machine code's 16-bit instructions are there for your convenience as you read this book; actual machine code has no such spaces.

memory location 5 into register *R2*. The program jumps to the instruction labeled "lab2" if *R2* is zero. If *R2* is not zero, the program executes an add instruction that adds one to register *R0*, and then proceeds to the next instruction, which is the instruction labeled "lab2." That instruction stores the contents of register *R0* to data memory location 9.

In writing the assembly program, we arbitrarily chose the registers that we used to store the result, the constant 1, and the data memory location copy. We could have used any registers for those purposes. For example, we could have used register *R7* to hold the result, meaning all occurrences of *R0* in the code would instead have been *R7*. Furthermore, in writing the assembly program, we arbitrarily chose the labels "lab1" and "lab2." We could have picked other names for those labels, such as "skip1" and "done," or "Fred" and "George." It's best, though, to use descriptive labels that help people reading the assembly code to understand the program.

An assembler would automatically convert the assembly code to the machine code shown in Figure 8.14(b). For each assembly instruction, the assembler determines the specific instruction type by looking at the mnemonic as well as the operands if necessary, and then outputs the appropriate opcode bits (four bits) for that instruction type, as defined in Table 8.2. For example, the assembler would look at the first instruction "MOV R0, #0" and thus know from the first three letters "MOV" that this is one of the data movement instructions; the assembler would look at the operands, and seeing "R0" would know this is either a regular load or a load-constant instruction; finally, the assembler would see the "#" and conclude this is a load-constant instruction, thus outputting the opcode "0011" for a load-constant instruction, as shown in the first machine instruction of the figure.

The assembler converts the operands to bits also, converting "R0" of the first instruction to "0000," and "#0" to "00000000," as shown in the first machine instruction of the figure.

The JMPZ instruction requires some extra handling. The assembler recognizes this as a *Jump-if-zero* instruction and thus outputs the opcode "0101." The assembler converts the first operand, "R2," to "0010." The assembler then reaches the second operand, "lab1," and does not know what bits to output, since the assembler doesn't yet know the address of the instruction labeled "lab1," as the assembler hasn't reached that instruction yet in the program. To solve this problem, many assemblers actually make *two passes* over the assembly code: during the first pass, the assembler creates a table of all labels and their addresses, and then on the second pass the assembler outputs machine code. Such an assembler would therefore know during the second pass that the instruction labeled "lab1" is at an address two addresses beyond the first JMPZ instruction—specifically, that the "lab1" instruction is at address 5, while the JMPZ instruction is at address 3 (assuming that the first instruction is at address 0, not 1). Thus, the assembler would output an offset of 2 to jump forward 2 addresses. Notice that the labels "lab1" and "lab2" do not appear in the machine code—they are merely a convenience construct that the assembler provides for the assembly-language programmer.

8.6 FURTHER EXTENSIONS TO THE PROGRAMMABLE PROCESSOR

Instruction Set Extensions

Extending the instruction set with further instructions would require similar types of extensions and modifications to the control unit, datapath, and FSM. A programmable processor might contain dozens more ***data movement instructions***, which move data between data memory and the register file, or between registers. For example, a processor might have instructions for copying the contents of one register to another (e.g., MOV R0, R1, which would copy *R1*'s contents into *R0*), and would carry out that instruction using a state that reads the source register, passes the read value through the ALU unchanged, and writes the ALU output to the destination register. As another example, a processor might have instructions that would use the contents of a register as the address from which to read data memory, known as *indirect* addressing.

A programmable processor would also contain dozens of ***arithmetic/logic instructions***, which perform arithmetic and logic operations on registers in the register file. For example, a processor might include not just add and subtract instructions, but also increment, complement, decrement, AND, OR, XOR, shift left, shift right, and other instructions that could be carried out by an ALU.