CHAPTER 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, form a common class of digital circuits. A circuit performing a single processing task is 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 popular. Programmable processors are 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. A designer could create a custom digital circuit for a seat belt warning light system (Chapter 2) or an FIR filter system (Chapter 5), or instead could program a general-purpose processor circuit to implement those systems.

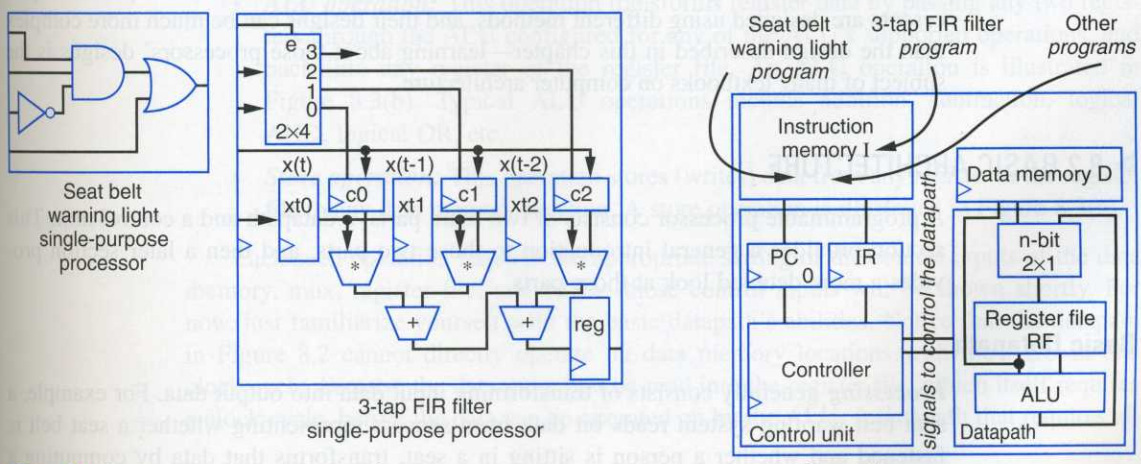


Figure 8.1 Single-purpose versus general-purpose processors.

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 computing 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, a particular programmable desktop processor can run Windows 7, Windows XP, Linux, or some new operating system. That same processor can run application programs like word processors, spreadsheets, video games, and web browsers. Likewise, a particular 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. There are also 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 a job, it is interesting and enlightening to understand how such a programmable processor works. Some people argue that designers who understand how a processor works are 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.

This chapter shows how to design a simple programmable processor using the previously-described digital design methods. The purpose is mainly to demystify these devices and to provide an insight on how programmable processors work. 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. This section provides a general introduction to those two parts, and then a later section provides a more detailed look at those parts.

Basic Datapath

Processing generally consists of transforming input data into output data. 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. The transformations take place inside a processor's ***datapath***, which consists of several parts.

A ***data memory*** contains the programmable processor's input and output data. Components external to the processor, such as sensors or displays, may also access that memory to write or read that data, perhaps through a second memory port (not shown). For example, an FIR filter system may have a component that writes digitized signal values to a particular word in the data memory, which the processor can read. To process that data, a programmable processor needs to be able to *load* data from the 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 *transformation* operations (typically an ALU) with results stored back into a register, and needs to be able to *store* data from a register back into data memory. Therefore, a programmable processor needs the basic circuit shown in Figure 8.2, having a data memory, register file, and ALU, together comprising a datapath. The basic datapath shown in Figure 8.2 can perform the following possible ***datapath operations*** in a given clock cycle:

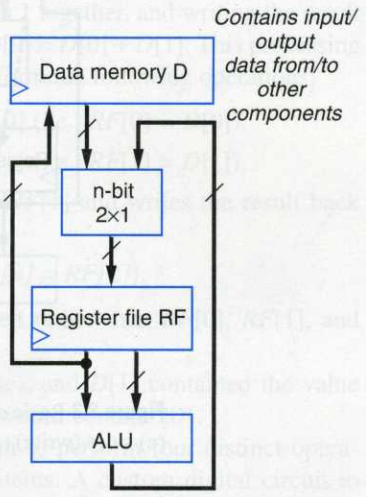


Figure 8.2 Basic datapath of a programmable processor.

- ***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).

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

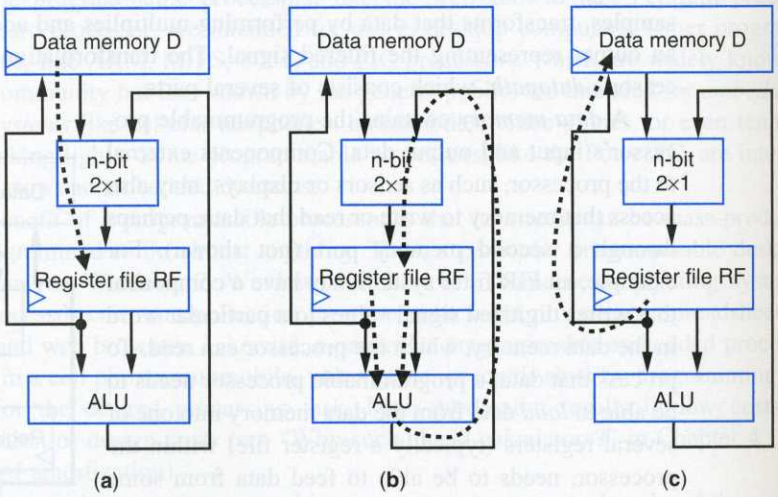


Figure 8.3 Basic datapath operations: (a) load (read), (b) ALU operation (transform), and (c) store (write).

data to first pass through the register file before that data can be transformed by the ALU is known as a ***load-store architecture***.

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. The datapath does not support reading more than one data memory location during a datapath operation, and it does not support writing to more than one register file location during an operation. (3) is *not* a valid operation. The datapath does not support reading two data memory locations during one operation, and furthermore does not have connections directly from the data memory to the ALU to perform the add. (4) is a valid operation. The ALU can be configured to simply pass one of its inputs through to the output (perhaps by adding 0) and storing the result in the register file. (5) is *not* a valid operation. A read data memory location cannot be fed 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 the basic datapath of Figure 8.2 should perform the simple processing task of adding data memory location 0 and data memory location 1 together, and writing the result in data memory location 9—in other words, computing $D[9] = D[0] + D[1]$. This processing task can be achieved by "instructing" the datapath to perform the following operations:

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

Note that any registers in the register file could be used rather than $RF[0]$, $RF[1]$, and $RF[2]$.

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. A custom digital circuit to implement $D[9] = D[0] + D[1]$ could just feed $D[0]$ and $D[1]$ through an adder whose output connects to $D[9]$, thus avoiding the four operations involving the register file and ALU. This simple example demonstrates 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.

A method is needed 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 should execute on the datapath. Such a description of desired processor operations uses ***instructions***, and a collection of instructions is known as a ***program***. The desired program is stored in another memory called the ***instruction memory***. A later section describes how to represent those instructions. For now, assume that the four instructions are somehow stored in locations 0, 1, 2, and 3 of the instruction memory I shown in Figure 8.4.

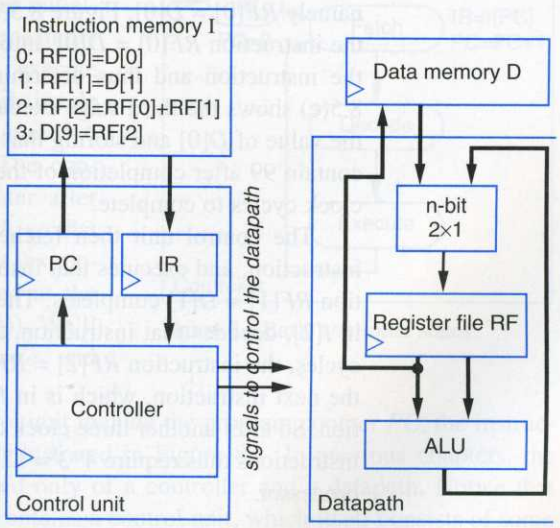


Figure 8.4 The control unit in a programmable processor.

The processor's ***control unit*** reads each instruction from instruction memory, and executes that instruction using the datapath. To execute the above simple program, the control unit repeatedly performs the following tasks, known as ***stages***, each stage requiring one clock cycle.

1. ***Fetch***: The control unit starts by reading ("fetching") the current instruction into a local register called the ***instruction register*** or ***IR***. The current instruction's address in the instruction memory is kept in a register called the ***program counter*** or ***PC***. The first instruction fetched for the above example will be $RF[0] = D[0]$, which will be placed into the IR.
2. ***Decode***: The control unit then determines ("decodes") what operation this instruction is requesting. For the above example, the decode stage will determine that the instruction in the IR is requesting a datapath load operation.
3. ***Execute***: The control unit carries out ("executes") the instruction's requested datapath operation by setting the datapath's control lines appropriately. For the above example's first instruction $RF[0] = D[0]$, the control unit would set the control lines of the datapath to read $D[0]$, set the 2x1 mux in front of the register file to pass the read data, and set the register file to store that data into $RF[0]$.

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

Because the instruction locations are usually in sequence, the PC can be implemented using a simple up-counter to proceed from one instruction to the next instruction of the program—hence the name "program counter." 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 program's first instruction, namely $RF[0] = D[0]$. Figure 8.5(a) shows the first stage fetching $I[0]$'s content, which is the instruction $RF[0] = D[0]$, into the *IR*. Figure 8.5(b) shows the second stage decoding the instruction and thus determining that the instruction is a load instruction. Figure 8.5(c) shows the third stage 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 $RF[0]$ will contain 99 after completion of the execute stage. The first instruction thus required three clock cycles to complete.

The control unit then fetches the next instruction, which is in $I[1]$, decodes that instruction, and executes that instruction. Thus after three more clock cycles, the instruction $RF[1] = D[1]$ completes. The control unit then fetches the next instruction, which is in $I[2]$, decodes that instruction, and executes that instruction. So after three more clock cycles, the instruction $RF[2] = RF[0] + RF[1]$ completes. Finally, the control unit fetches the next instruction, which is in $I[3]$, decodes that instruction, and executes that instruction. So after another three clock cycles, the instruction $D[9] = RF[2]$ completes. The four instructions thus require 4*3 = 12 clock cycles to run to completion on the programmable processor.

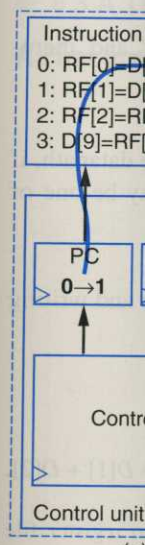


Figure 8.5 Thr

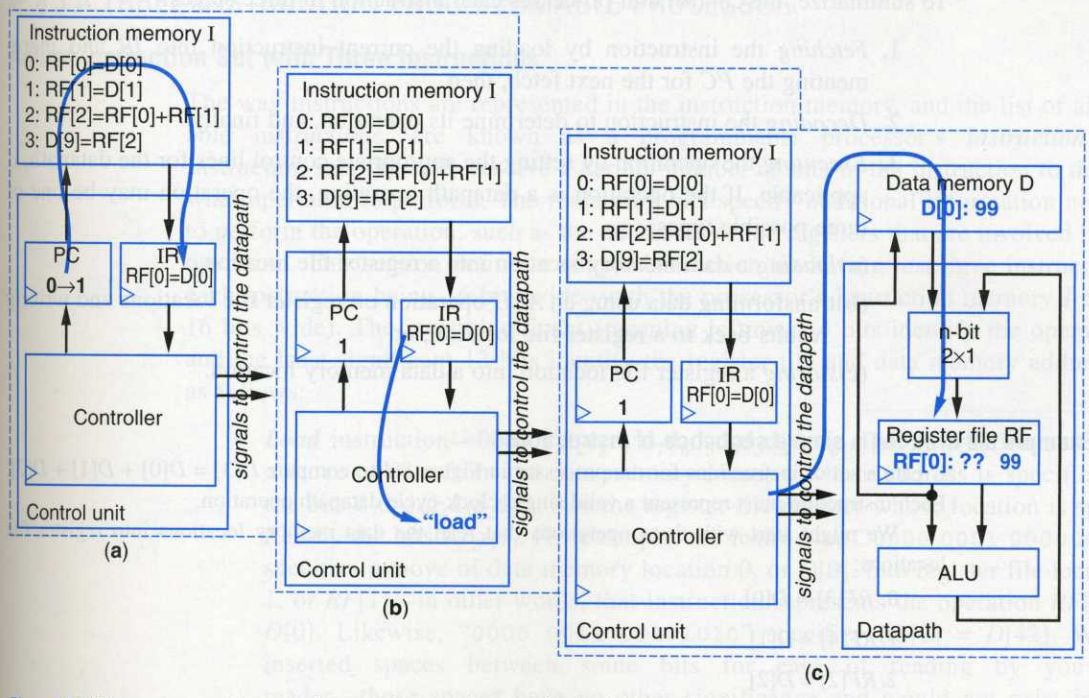


Figure 8.5 Three stages of processing *one* instruction: (a) fetch, (b) decode, (c) execute.

The control unit will require a controller, like those described in Chapter 3. After initializing the *PC* to 0, the controller repeatedly performs the fetch, decode, and execute steps—note that a controller appears inside the control unit in Figure 8.4. Figure 8.6 shows an FSM for that controller. 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.

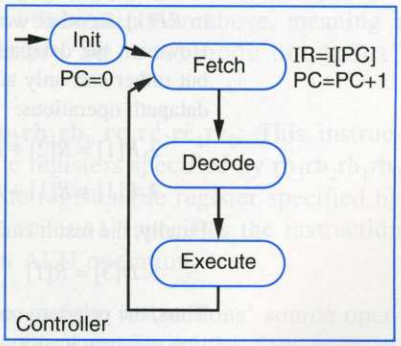


Figure 8.6 Basic controller states.

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, the 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. *Fetching* the instruction by loading the current instruction into *IR* and incrementing the *PC* for the next fetch, then
2. *Decoding* the instruction to determine its operation, and finally
3. *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. $RF[3] = D[0]$
1. $RF[4] = D[1]$
2. $RF[2] = D[2]$

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

Next, the three values need to be added and the result stored in a register file location, such as in $RF[1]$. In other words, the following operation should be performed: $R[1] = R[2] + R[3] + R[4]$. 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, the desired addition computation can split into two datapath operations:

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

Finally, the result can be written into $D[3]$:

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

Thus, the program consists of the six instructions appearing above, which would appear 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 Figure 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 instructions are represented in the instruction memory, and the list of allowable instructions, are known as a programmable processor's ***instruction set***. Instruction sets typically reserve a certain number of bits in the instruction to denote what operation to perform. The remaining bits specify additional information needed to perform the operation, such as the addresses of the registers that are involved in the operation. This section defines a simple instruction set having just three instructions, each instruction being 16 bits wide (with the processor's instruction memory *I* being 16 bits wide). The most significant (meaning leftmost) 4 bits identify the operation, and the least significant 12 bits identify the 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 0001 00000000`" specifies a move of data memory location 0, or *D*[0], into register file location 1, or *RF*[1]—in other words, that instruction represents the operation *RF*[1] = *D*[0]. Likewise, "`0000 0011 00101010`" specifies *RF*[3] = *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 *rb* and *rc* registers without changing them. Note also that the instruction merely contains the addresses of registers (or memory); the registers themselves are in the register file.

Using this instruction set, the desired computation *D*[9] = *D*[0] + *D*[1] can be written as the program 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***. "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.

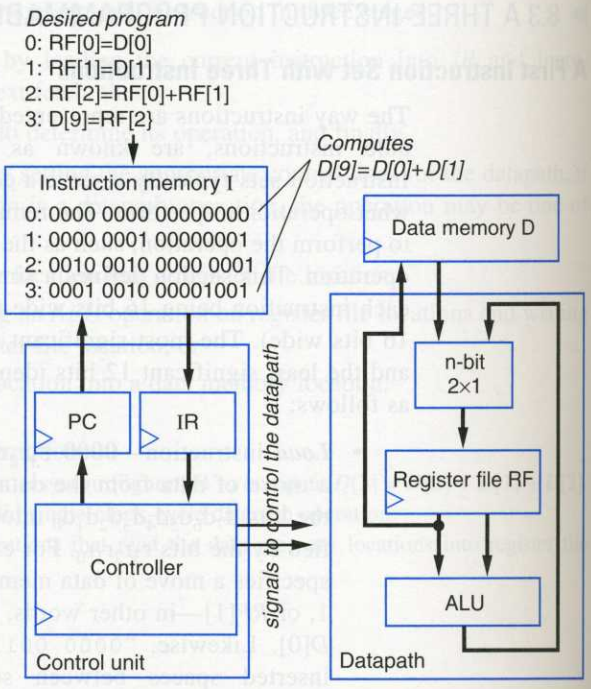


Figure 8.7 A program that computes $D[9] = D[0] + D[1]$ using a given instruction set. The spaces between the instruction memory's bits are for readability only—those spaces don't exist in the memory.

The above-defined three-instruction instruction set can be used to write programs to perform other computations, such as $D[5] = D[5] + D[6] + D[7]$. Such a program is 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 (//) represents a comment, and is not part of an instruction. Examining the program instruction by instruction reveals that the program computes the desired result.

Machine Code versus Assembly Code

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 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 programs. An assembler allows us

```
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.

to write instructions using ***mnemonics***, or symbols, that the assembler automatically translates to machine code. Thus, an assembler may support the 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 $RF[a] = RF[b] + RF[c]$.

Using those mnemonics, the program in Figure 8.7 for $D[9] = D[0] + D[1]$ could be rewritten 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 `0`s and `1`s 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 mnemonics for both instructions are the same—"MOV." The assembler distinguishes those two types of instructions by examining 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, a complete digital circuit for a three-instruction programmable processor can be designed. The design process is similar to the RTL design process of Chapter 5.

▶ 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 `0`s and `1`s. To enable debugging of the program, as well as to show the computed data, those early computers used rows of lights—on lights meant `1`s, off lights meant `0`s. 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.

The process begins 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 left-most four bits of the instruction register. Likewise, *ra* means *IR*[11..8], *rb* means *IR*[7..4], *rc* means *IR*[3..0], and *d* means *IR*[7..0].

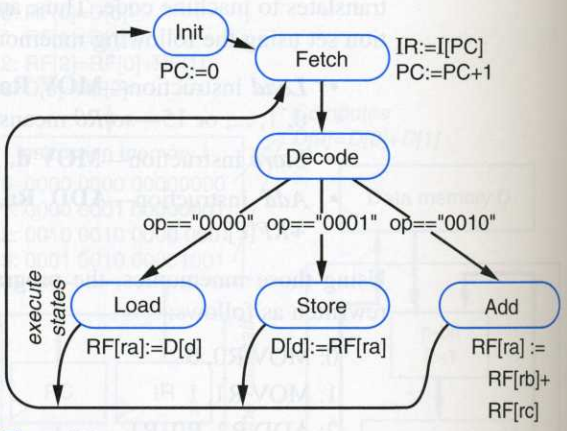


Figure 8.9 High-level state machine description of a three-instruction programmable processor.

The next step in the RTL design process is to create the datapath. The datapath was already created in Figure 8.4, which is refined in Figure 8.10 to show every control signal from the controller. The refined datapath has control signals for each read and write port of the register file. 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`—assume that 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 2x1 mux in front of the register file's write data port. Finally, control signals are

▶ "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 "to 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 `0`s and `1`s and load those `0`s and `1`s 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.

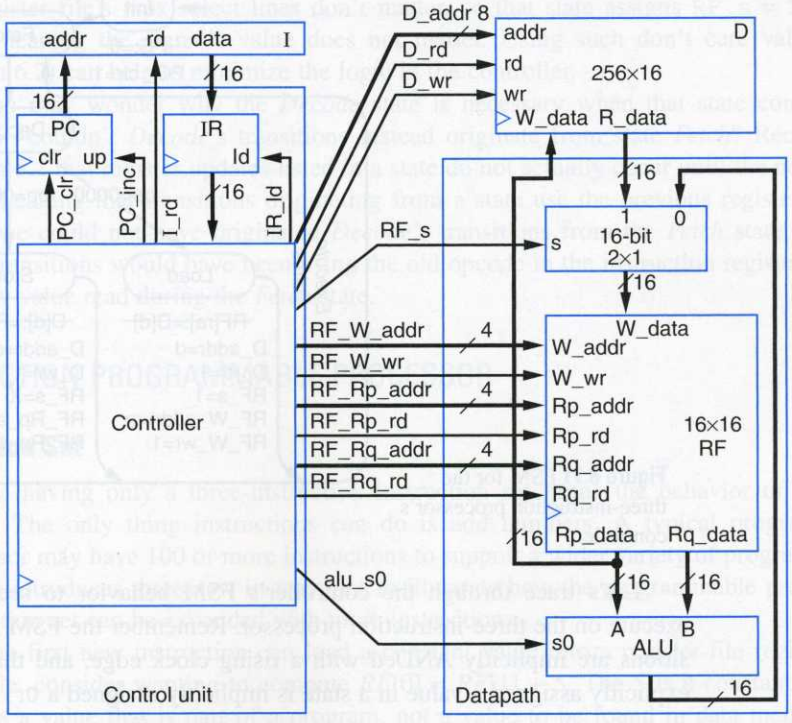


Figure 8.10 Refined datapath and control unit for the three-instruction processor.

included for the data memory, which 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 needed for the HLSM from Figure 8.9. The RTL design process proceeds by 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. The FSM is straightforwardly achieved by replacing the high-level actions of the HLSM 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. The controller's design could be completed by converting the FSM to a state register and combinational logic using the methods from Chapter 3.

The resulting design represents a simple but complete programmable processor.

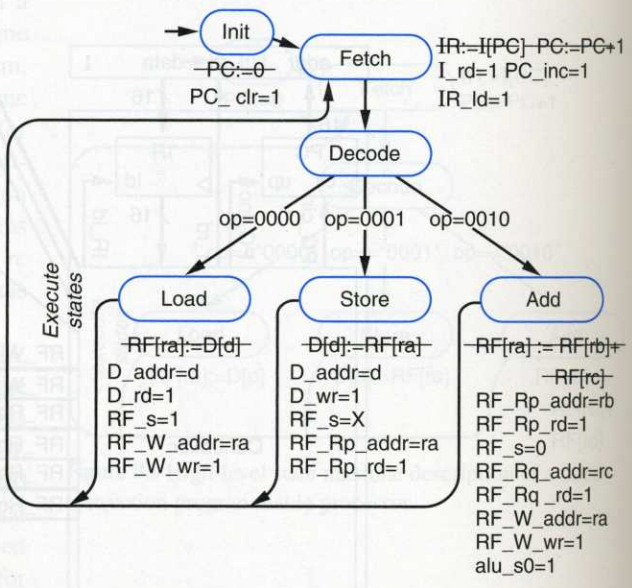


Figure 8.11 FSM for the three-instruction processor's controller.

Let's trace through the controller's FSM behavior to see how a program would execute on the three-instruction processor. Remember 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 *I* at address 0 (because *PC* is 0) and loads the read value into *IR*—that read value will be the instruction that was 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 (i.e., depending on 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 *ra* (which is *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 that state assigns `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 to minimize the logic in the controller.

You may wonder why the *Decode* state is necessary when that state contains no actions—couldn't *Decode*'s transitions instead originate from state *Fetch*? Recall from Section 5.2 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. The only thing instructions can do is add numbers. A typical programmable processor may have 100 or more instructions to support a wider variety of programs. This section introduces three new instructions to illustrate how the programmable processor's instruction set can be expanded with more instructions.

The first new instruction can load a constant value into a register-file register. For example, consider wanting to compute $RF[0] = RF[1] + 5$. The 5 is a constant. A ***constant*** is a value that is part of a program, not a value to be found in data memory. An instruction is thus needed that can load a constant into a register, e.g., to support an instruction like $RF[2] = 5$. A new instruction can be introduced 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*. For example, "`0011 0010 00000101`" specifies the instruction $RF[2] = 5$. 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 4.6), c can be –128, –127, ..., 0, ..., 126, 127. The "#" is a special symbol that enables the assembler to distinguish this instruction from a regular load instruction.

Another new instruction performs 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 $RF[a] = RF[b] - RF[c]$

A third new instruction allows the program 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 $ra_3ra_2ra_1ra_0$ are 0, the *PC* should be loaded 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. For example, "0101 0011 11111110" specifies that if the value in *RF*[3] is 0, then the *PC* should be set to *PC* – 2. The mnemonic is

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

Negative jump offsets are commonly used to implement a loop. The 8-bit offset can specify an offset 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 or hundreds of instructions.

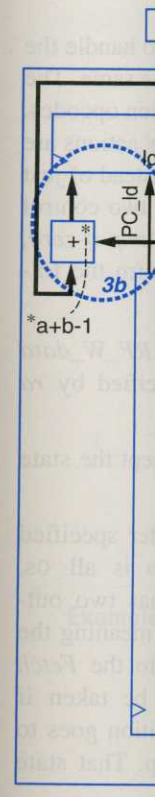
TABLE 8.1 Six-instruction instruction set.

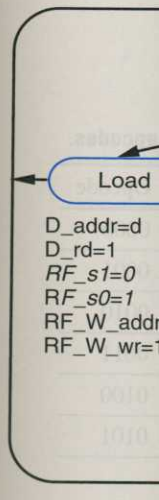
Instruction	Meaning
MOV Ra, d	RF[a] = D[d]
MOV d, Ra	D[d] = RF[a]
ADD Ra, Rb, Rc	RF[a] = RF[b]+RF[c]
MOV Ra, #C	RF[a] = C
SUB Ra, Rb, Rc	RF[a] = RF[b]-RF[c]
JMPZ Ra, offset	PC=PC+offset if RF[a]=0

Extending the Control Unit and Datapath

The three new instructions require some extensions, shown in Figure 8.12, to the control unit and datapath of Figure 8.10. 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 from the ALU output. Thus, the register file's multiplexer is widened from 2x1 to 3x1, another mux control signal is added, and a new signal coming from the controller labeled *RF_W_data* is added, which will connect with *IR*[7...0]. These changes are highlighted by the dashed circle labeled "*1*" in Figure 8.12.

Second, the subtract instruction requires using an ALU capable of subtraction, so another ALU control signal is added, highlighted by the dashed circle labeled "*2*" in the figure. Third, the jump-if-zero instruction requires that the ability to detect if a register is zero, and the ability to add *IR*[7...0] to the *PC*. Thus, a datapath component is inserted 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. The *PC* register is upgraded 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*.





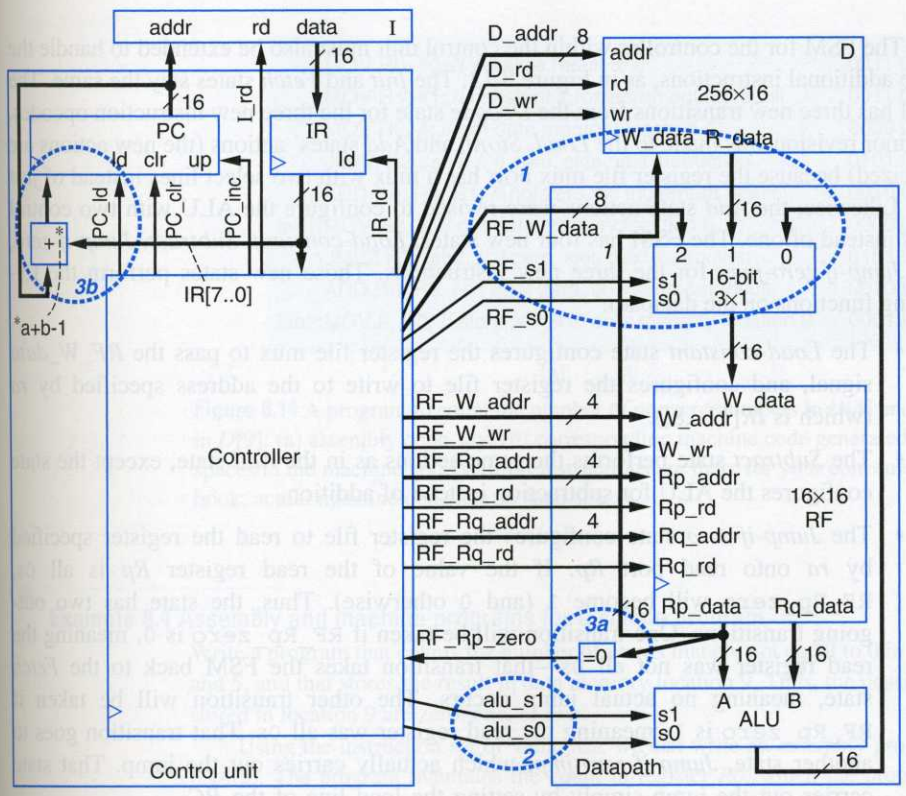


Figure 8.12 Control unit and datapath for the six-instruction processor.

s1	s0	ALU operation
0	0	pass A through
0	1	A+B
1	0	A–B

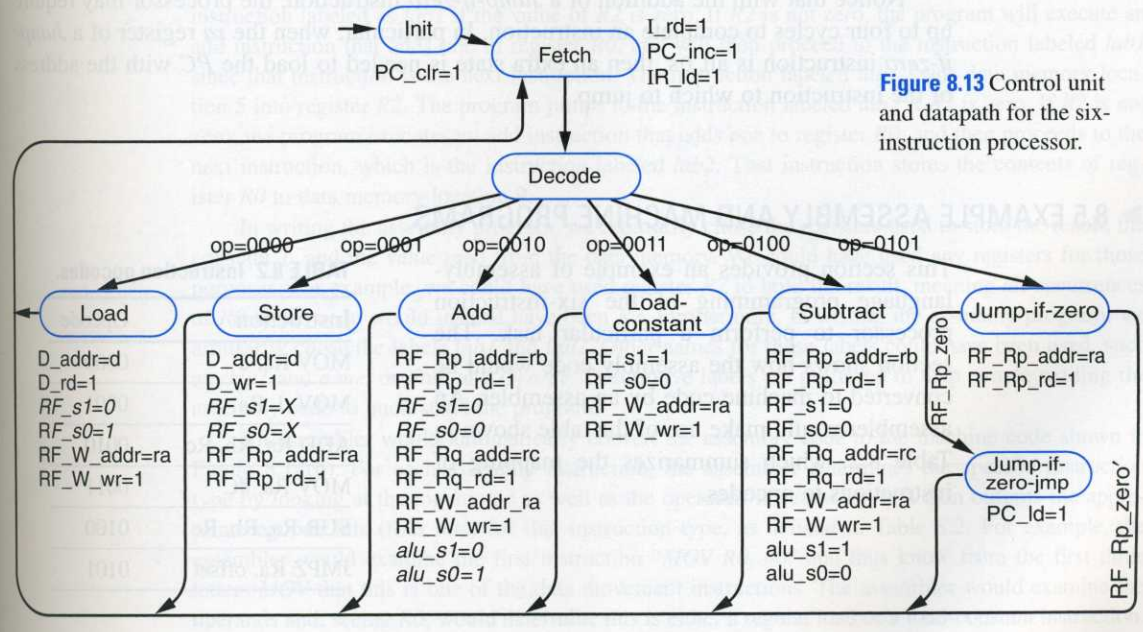


Figure 8.13 Control unit and datapath for the six-instruction processor.

The FSM for the controller within the control unit must also be extended to handle the three additional instructions, as in Figure 8.13. The *Init* and *Fetch* states stay the same. The FSM has three new transitions from the *Decode* state for the three new instruction opcodes. A minor revision was made to the *Load*, *Store*, and *Add* states' actions (the new actions are italicized) because the register file mux now has a mux with two select lines instead of just one. Likewise, the *Add* state actions were revised to configure the ALU with two control lines instead of one. The FSM has four new states, *Load-constant*, *Subtract*, *Jump-if-zero*, and *Jump-if-zero-jmp*, for the three new instructions. Those new states perform the following functions on the datapath:

- The *Load-constant* state configures the register file mux to pass the *RF_W_data* signal, and configures the register file to write to the address specified by *ra* (which is *IR*[11...8]).
- The *Subtract* state performs the same actions as in the *Add* state, except the state configures the ALU for subtraction instead of addition.
- The *Jump-if-zero* state configures the register file to read the register specified by *ra* onto read port *Rp*. If the value of the read register *Rp* is all `0`s, `RF_Rp_zero` will become `1` (and `0` otherwise). Thus, the state has two outgoing transitions. One transition will be taken if `RF_Rp_zero` is `0`, meaning the read register was not all `0`s—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 `0`s. That transition goes to another state, *Jump-if-zero-jmp*, which actually carries 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 require up to four cycles to complete an instruction. In particular, when the *ra* register of a *Jump-if-zero* instruction is all `0`s, 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

This section provides an example of assembly-language programming of the six-instruction processor to perform a particular task. The section shows 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 summarizes the mapping of 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

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.

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 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 used to store the result, the constant 1, and the value read from the data memory. 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*. Other names for those labels could have been used, such as *skip1* and *done*, or *Fred* and *George*. Descriptive labels are preferred to 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 examine 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 examine the operands and, seeing *R0*, would determine this is either a regular load or a load-constant instruction.

Finally, the assembler would detect 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 Figure 8.14(b).

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 Figure 8.14(b).

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* because 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 to the instruction labeled *lab1*. 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*** that 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*** that 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.

A programmable processor would furthermore contain several ***flow-of-control instructions*** that determine the next value of the *PC*. For example, a processor might include not just a jump-if-zero instruction, but also a jump-if-not-zero, an unconditional jump, an indirect jump, and perhaps even jump-if-negative and similar such instructions. Furthermore, a processor may include instructions that can jump farther than just a small offset from the current *PC*, and perhaps even to an absolute address rather than an offset address.

Input/Output Extensions

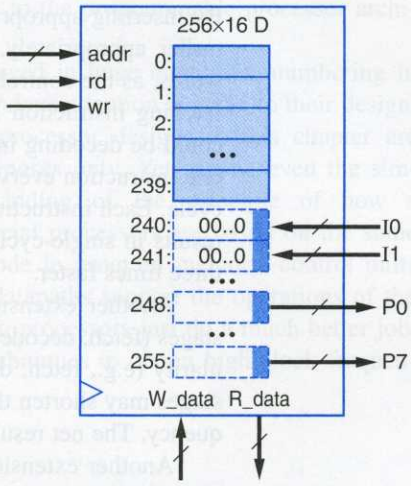


Figure 8.15 Connecting to external pins.

Section 1.3 introduced a basic microprocessor having eight inputs *I0*, *I1*, ..., *I7*, and eight outputs *P0*, *P1*, ..., *P7*. The basic programmable processor of Figure 8.12 can be extended to implement such external inputs and outputs. One method for such an extension utilizes a specially designed data memory. In that data memory, the last 16 words of the memory are replaced by direct connections to the input and output pins, as illustrated in Figure 8.15. The data memory stores locations 0 through 239 in a normal RAM. Location 240, however, is actually a special word whose high 15 bits are all `0`s, and whose lowest bit comes from a flip-flop loaded every cycle with the value on external input pin *I0*. Thus, reading location 240 will result in either `00...01` (integer 1), or `00...00` (integer 0), depending on the value appearing at *I0*. Likewise, location 241 is connected to pin *I1*, location 242 to *I2*, and so on, with location 247 connected to *I7*. Locations 248 through 255 are connected to pins *P0* through *P7*, except the pins are connected to those locations' flip-flop outputs rather than inputs. For example, writing to location 255 writes the flip-flop with either `0` or `1` (only the low-order bit matters during the write), and that flip-flop drives external output pin *P7*. This approach of accessing inputs and outputs as if they were data memory locations is known as ***memory-mapped I/O***. Thus, an assembly-language programmer can read or write a microprocessor's external data pins simply by reading or writing particular data memory locations.

Example 8.5 Motion-in-the-dark detector in assembly language

Section 1.3 included an example, illustrated in Figure 1.24, that utilized a microprocessor to implement a motion-in-the-dark detector. That section utilized C code to compute the expression `P0 = I0 && !I1`. This example shows the underlying assembly code that implements that C expression. Assuming that the microprocessor's external pins *I0...I7* and *P0...P7* are mapped to data memory locations as in Figure 8.15, the expression can be programmed in assembly as follows:

```
0: MOV R0, 240     // move D[240], which is the value at pin I0, into R0
1: MOV R1, 241     // move D[241], which is that value at pin I1, into R1
2: NOT R1, R1      // compute !I1, assuming existence of a complement instruction
3: AND R0, R0, R1  // compute I0 && !I1, assuming an AND instruction
4: MOV 248, R0     // move result to D[248], which is pin P0
```

Performance Extensions

One difference between real processors and the basic processor architecture in this chapter is that many real processors are pipelined (see Section 6.5). The basic three-instruction processor utilized a controller with three stages: *fetch*, *decode*, and *execute*.

By inserting appropriate pipeline registers throughout the design and modifying the controller appropriately, the fetch, decode, and execute stages can be pipelined. In other words, as the control unit decodes instruction 1, the control unit could be simultaneously fetching instruction 2. Next, as the control unit executes instruction 1, the control unit could be decoding instruction 2, and fetching instruction 3. Thus, rather than processing one instruction every 3 cycles, the control unit could be processing one instruction every cycle. Each instruction still takes 3 cycles to process (3-cycle latency), but the pipelining results in single-cycle throughput. The net result would be that programs would execute three times faster.

Another extension involves creating deeper pipelines. Thus, rather than just three stages (fetch, decode, execute), the stages might be divided into stages of even finer granularity (e.g., fetch, decode, read operands, execute, store results). Creating finer grained stages may shorten the longest register-to-register delay, which enables a faster clock frequency. The net result would again be faster program execution.

Another extension involves having multiple ALUs in the datapath. The control unit may then perform multiple ALU operations simultaneously in the datapath. One form of this extension involves a processor whose instruction set uses instructions with multiple opcodes and associated operands in a single instruction, known as a ***very large instruction word (VLIW)*** processor. Another form uses a processor with a control unit that reads in multiple instructions simultaneously and then assigns those instructions to execute simultaneously on available ALUs, known as a ***superscalar*** processor. A high-end desktop processor may support perhaps 5 simultaneous instructions, with perhaps 10 stages of pipelining. Thus, at any moment, such a processor may be in the middle of processing 5*10 = 50 different instructions. Needless to say, modern processor architectures can become quite complex.

This chapter described the basic idea of how a programmable processor's design works and how the design could be extended to support a fuller instruction set. We leave the role of describing a complete processor, as well as modern processor design techniques for improved performance (such as pipelining, caching, etc.), to textbooks on computer architecture.

▶ 8.7 CHAPTER SUMMARY

Section 8.1 stated that programmable processors are widely used for implementing a system's desired functionality, due in part to their easy availability and short design time (design consists of writing programs). Section 8.2 provided the basic architecture of a programmable processor, consisting of a general-purpose datapath having a register file and ALU; a control unit having a controller, *PC*, and *IR*; and memories for storing the program and the data. The control unit would fetch the next instruction from program memory, decode the instruction, and then execute the instruction by configuring the datapath to carry out the instruction's specified operation. Section 8.3 designed a simple three-instruction programmable processor and showed how a program would be represented as `0`s and `1`s (machine code) in the processor's program memory. Section 8.4 designed a six-instruction processor and discussed how further extensions could be made to add more instructions and hence achieve a more complete processor architecture. Section 8.5 provided an example of assembly and machine code for the six-instruction

processor. Section 8.6 discussed a few extensions to the programmable processor architecture such as memory-mapped I/O.

Programmable processors are typically produced in huge quantities, numbering in the tens of millions or even billions, and so tremendous attention is given to their design. Readers should realize that the programmable processor designs in this chapter are extremely simplistic and used for illustration purposes only. Yet, seeing even the simplistic designs hopefully provides an understanding of the principle of how a programmable processor works. Modern commercial processors are based on the same principles—instructions are stored as machine code in program memory, control units fetch, decode, and execute the instructions, and datapaths support the operations of the instructions using register files and ALUs. Modern processors just do a much better job, using concurrency, pipelining, and many other techniques to obtain high clock frequencies and fast program execution.

8.8 EXERCISES

SECTION 8.2: BASIC ARCHITECTURE

8.1 If a processor's program counter is 20 bits wide, up to how many words can the processor's instruction memory hold (ignoring any special tricks to expand the instruction memory size)?

8.2 Which of the following are legal single-cycle datapath operations for the datapath in Figure 8.2? Explain your answer.
(a) Copy data from a memory location into another memory location.
(b) Copy two register locations into two memory locations.
(c) Add data from a register file location and a memory location, storing the result in a memory location.

8.3 Which of the following are legal single-cycle datapath operations for the datapath in Figure 8.2? Explain your answer.
(a) Copy data from a register file location into a memory location.
(b) Subtract data from two memory locations and store the result in another memory location.
(c) Add data from a register file location and a memory location, storing the result in the same memory location.

8.4 Assume we are using a dual-port memory from which we can read two locations simultaneously. Modify the datapath of the programmable processor of Figure 8.2 to support an instruction that performs an ALU operation on any two memory locations and stores the result in a register file location. Trace through the execution of this operation, as illustrated in Figure 8.3.

8.5 Determine the operations required to instruct the datapath of Figure 8.2 to perform the operation: $D[8] = (D[4] + D[5]) - D[7]$, where D represents the data memory.

SECTION 8.3: A THREE-INSTRUCTION PROGRAMMABLE PROCESSOR

8.6 If a processor's instruction has 4 bits for the opcode, how many possible instructions can the processor support?

8.7 What does the following assembly program, which uses the three-instruction instruction set of this chapter, compute? *MOV R5, 19; ADD R5, R5, R5; MOV 20, R5.*