4. Memory access or R-type instruction completion step

During this step, a load or store instruction accesses memory and an arithmetic-logical instruction writes its result. When a value is retrieved from memory it is stored into the memory data register (MDR), where it must be used on the next clock cycle.

Memory reference:

```
MDR = Memory [ALUOut];
```

or

```
Memory [ALUOut] = B;
```

Operation: If the instruction is a load, a data word is retrieved from memory and is written into the MDR. If the instruction is a store, then the data is written into memory. In either case, the address used is the one computed during the previous step and stored in ALUOut. For a store, the source operand is saved in B. (B is actually read twice, once in step 2 and once in step 3. Luckily, the same value is read both times, since the register number—which is stored in IR and used to read from the register file—does not change.) The signal MemRead (for a load) or MemWrite (for store) will need to be asserted. In addition, for loads and stores, the signal IorD is set to 1 to force the memory address to come from the ALU, rather than the PC. Since MDR is written on every clock cycle, no explicit control signal need be asserted.

Arithmetic-logical instruction (R-type):

```
Reg[IR[15-11]] = ALUOut;
```

Operation: Place the contents of ALUOut, which corresponds to the output of the ALU operation in the previous cycle, into the Result register. The signal RegDst must be set to 1 (to force the rd (bits 15–11) field to be used to select the register file entry to write). RegWrite must be asserted, and MemtoReg must be set to 0 (so that the output of the ALU is written, as opposed to the memory data output).

5. Memory read completion step

During this step, loads complete by writing back the value from memory.

Load:

```
Reg[IR[20-16]]=MDR;
```

Operation: Write the load data, which was stored into MDR in the previous cycle, into the register file. To do this, we set MemtoReg = 1 (to write the result from memory), assert RegWrite (to cause a write), and we make RegDst = 0 to choose the rt (bits 20–16) field as the register number.

<table>
<tr><th>Step name</th><th>Action for R-type instructions</th><th>Action for memory-reference instructions</th><th>Action for branches</th><th>Action for jumps</th></tr>
<tr><td>Instruction fetch</td><td colspan="4">IR = Memory[PC]
PC = PC + 4</td></tr>
<tr><td>Instruction decode/register fetch</td><td colspan="4">A = Reg [IR[25–21]]
B = Reg [IR[20–16]]
ALUOut = PC + (sign-extend (IR[15–0]) << 2)</td></tr>
<tr><td>Execution, address computation, branch/ jump completion</td><td>ALUOut = A op B</td><td>ALUOut = A + sign-extend (IR[15–0])</td><td>if (A == B) then PC = ALUOut</td><td>PC = PC [31–28] || (IR[25–0]<<2)</td></tr>
<tr><td>Memory access or R-type completion</td><td>Reg [IR[15–11]] = ALUOut</td><td>Load: MDR = Memory[ALUOut]
or
Store: Memory [ALUOut] = B</td><td></td><td></td></tr>
<tr><td>Memory read completion</td><td></td><td>Load: Reg[IR[20–16]] = MDR</td><td></td><td></td></tr>
</table>

FIGURE 5.35 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

This five-step sequence is summarized in Figure 5.35. From this sequence we can determine what the control must do on each clock cycle.

Defining the Control

Now that we have determined what the control signals are and when they must be asserted, we can implement the control unit. To design the control unit for the single-cycle datapath, we used a set of truth tables that specified the setting of the control signals based on the instruction class. For the multicycle datapath, the control is more complex because the instruction is executed in a series of steps. The control for the multicycle datapath must specify both the signals to be set in any step and the next step in the sequence.

In this subsection and in section 5.5, we will look at two different techniques to specify the control. The first technique is based on finite state machines that are usually represented graphically. The second technique, called *microprogramming*, uses a programming representation for control. Both of these techniques represent the control in a form that allows the detailed implementation—using gates, ROMs, or PLAs—to be synthesized by a CAD system. In this chapter, we will focus on the design of the control and its representation in these two forms. If you are interested in how these control specifications are

INTEL - 1012

translated into actual hardware, Appendix C continues the development of this chapter, translating the multicycle control unit to a detailed hardware implementation. The key ideas of control can be grasped from this chapter without examining the material in Appendix C. However, if you want to get down to the bits, Appendix C can show you how to do it!

The first method we use to specify the multicycle control is a *finite state machine*. A finite state machine consists of a set of states and directions on how to change states. The directions are defined by a *next-state function*, which maps the current state and the inputs to a new state. When we use a finite state machine for control, each state also specifies a set of outputs that are asserted when the machine is in that state. The implementation of a finite state machine usually assumes that all outputs that are not explicitly asserted are deasserted. The correct operation of the datapath depends on the fact that a signal that is not explicitly asserted is deasserted, rather than acting as a don't care. For example, the RegWrite signal should be asserted only when a register file entry is to be written; when it is not explicitly asserted, it must be deasserted.

Multiplexor controls are slightly different, since they select one of the inputs whether they are 0 or 1. Thus, in the finite state machine, we always specify the setting of all the multiplexor controls that we care about. When we implement the finite state machine with logic, setting a control to 0 may be the default and thus may not require any gates. A simple example of a finite state machine appears in Appendix B, and if you are unfamiliar with the concept of a finite state machine, you may want to examine Appendix B before proceeding.

The finite state control essentially corresponds to the five steps of execution shown on pages 385 through 388; each state in the finite state machine will take 1 clock cycle. The finite state machine will consist of several parts. Since the first two steps of execution are identical for every instruction, the initial two states of the finite state machine will be common for all instructions. Steps 3 through 5 differ, depending on the opcode. After the execution of the last step for a particular instruction class, the finite state machine will return to the initial state to begin fetching the next instruction.

Figure 5.36 shows this abstracted representation of the finite state machine. To fill in the details of the finite state machine, we will first expand the instruction fetch and decode portion, then we will show the states (and actions) for the different instruction classes.

We show the first two states of the finite state machine in Figure 5.37 using a traditional graphic representation. We number the states to simplify the explanation, though the numbers are arbitrary. State 0, corresponding to step 1, is the starting state of the machine.

The signals that are asserted in each state are shown within the circle representing the state. The arcs between states define the next state and are labeled

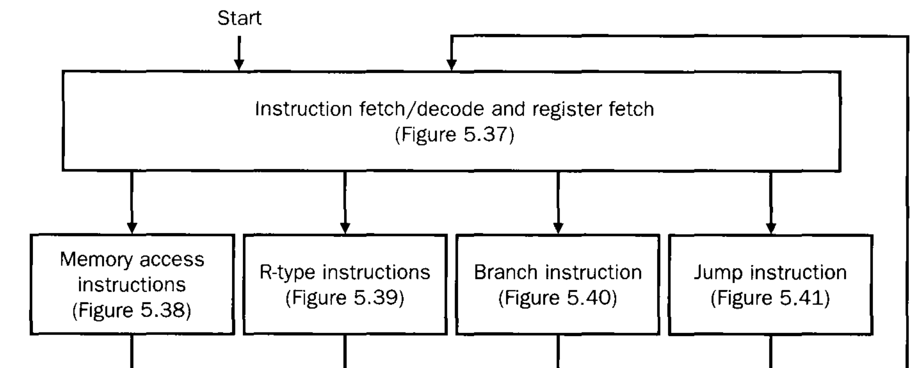


FIGURE 5.36 The high-level view of the finite state machine control. The first steps are independent of the instruction class; then a series of sequences that depend on the instruction opcode are used to complete each instruction class. After completing the actions needed for that instruction class, the control returns to fetch a new instruction. Each box in this figure may represent one to several states. The arc labeled *Start* marks the state in which to begin when the first instruction is to be fetched.

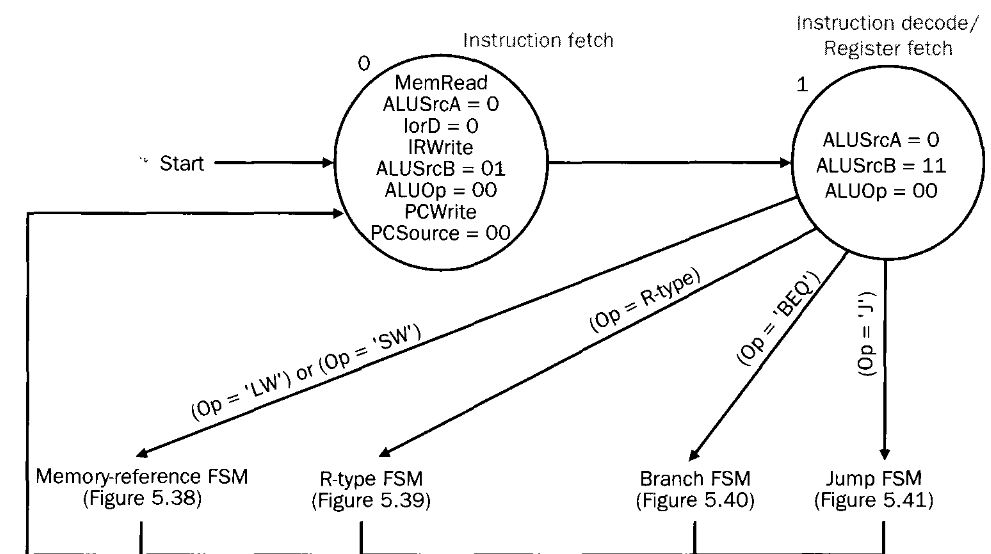


FIGURE 5.37 The instruction fetch and decode portion of every instruction is identical. These states correspond to the top box in the abstract finite state machine in Figure 5.36. In the first state we assert two signals to cause the memory to read an instruction and write it into the Instruction register (MemRead and IRWrite), and we set IorD to 0 to choose the PC as the address source. The signals ALUSrcA, ALUSrcB, ALUOp, PCWrite, and PCSource are set to compute PC + 4 and store it into the PC. (It will also be stored into ALUOut, but never used from there.). In the next state, we compute the branch target address by setting ALUSrcB to 11 (causing the shifted and sign-extended lower 16 bits of the IR to be sent to the ALU), setting ALUSrcA to 0 and ALUOp to 00; we store the result in the ALUOut register, which is written on every cycle. There are four next states that depend on the class of the instruction, which is known during this state. The control unit input, called Op, is used to determine which of these arcs to follow.

INTEL - 1012

with conditions that select a specific next state when multiple next states are possible. After state 1, the signals asserted depend on the class of instruction. Thus, the finite state machine has four arcs exiting state 1, corresponding to the four instruction classes: memory reference, R-type, branch on equal, and jump. This process of branching to different states depending on the instruction is called *decoding*, since the choice of the next state, and hence the actions that follow, depend on the instruction class.

Figure 5.38 shows the portion of the finite state machine needed to implement the memory-reference instructions. For the memory-reference instructions, the first state after fetching the instruction and registers computes the memory address (state 2). To compute the memory address, the ALU input multiplexors must be set so that the first input is the A register, while the second input is the sign-extended displacement field; the result is written into the ALUOut register. After the memory address calculation, the memory should be read or written; this requires two different states. If the instruction opcode is `lw`, then state 3 (corresponding to the step Memory access) does the memory read (MemRead is asserted). The output of the memory is always written into MDR. If it is `sw`, state 5 does a memory write (MemWrite is asserted). In states 3 and 5, the signal IorD is set to 1 to force the memory address to come from the ALU. After performing a write, the instruction `sw` has completed execution, and the next state is state 0. If the instruction is a load, however, another state (state 4) is needed to write the result from the memory into the register file. Setting the multiplexor controls MemtoReg = 1 and RegDst = 0 will send the loaded value in the MDR to be written into the register file, using rt as the register number. After this state, corresponding to the Memory read completion step, the next state is state 0.

To implement the R-type instructions requires two states corresponding to steps 3 (Execute) and 4 (R-type completion). Figure 5.39 shows this two-state portion of the finite state machine. State 6 asserts ALUSrcA and sets the ALUSrcB signals to 00; this forces the two registers that were read from the register file to be used as inputs to the ALU. Setting ALUOp to 10 causes the ALU control unit to use the function field to set the ALU control signals. In state 7, RegWrite is asserted to cause the register file to write, RegDst is asserted to cause the rd field to be used as the register number of the destination, and MemtoReg is deasserted to select ALUOut as the source of the value to write into the register file.

For branches, only a single additional state is necessary, because they complete execution during the third step of instruction execution. During this state, the control signals that cause the ALU to compare the contents of registers A and B must be set, and the signals that cause the PC to be written conditionally with the address in the ALUOut register are also set. To perform the

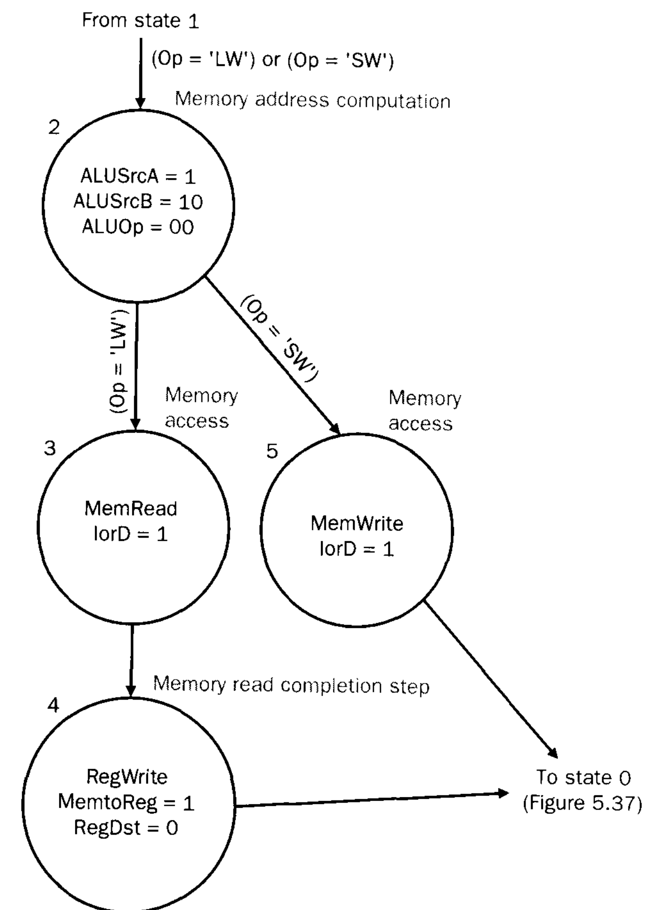


FIGURE 5.38 The finite state machine for controlling memory-reference instructions has four states. These states correspond to the box labeled "Memory access instructions" in Figure 5.36. After performing a memory address calculation, a separate sequence is needed for load and for store. The setting of the control signals ALUSrcA, ALUSrcB, and ALUOp is used to cause the memory address computation in state 2. Loads require an extra state to write the result from the MDR (where the result is written in state 3) into the register file.

comparison requires that we assert ALUSrcA and set ALUSrcB to 00, and set the ALUOp value to 01 (forcing a subtract). (We use only the Zero output of the ALU, not the result of the subtraction.) To control the writing of the PC, we assert PCWriteCond and set PCSource = 01, which will cause the value in the

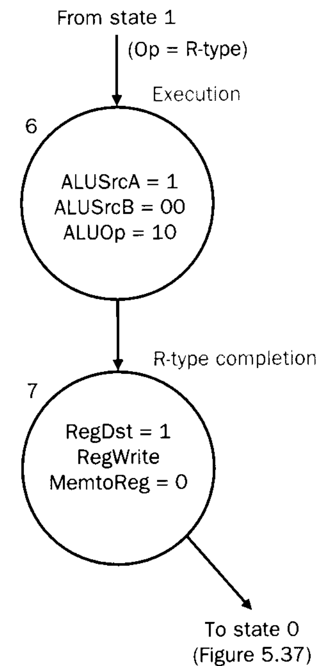


FIGURE 5.39 R-type instructions can be implemented with a simple two-state finite state machine. These states correspond to the box labeled "R-type instructions" in Figure 5.36. The first state causes the ALU operation to occur, while the second state causes the ALU result (which is in ALUOut) to be written in the register file. The three signals asserted during state 7 cause the contents of ALUOut to be written into the register file in the entry specified by the rd field of the Instruction register.

ALUOut register (containing the branch address calculated in state 1, Figure 5.37 on page 391) to be written into the PC if the Zero bit out of the ALU is asserted. Figure 5.40 shows this single state.

The last instruction class is jump; like branch, it requires only a single state (shown in Figure 5.41) to complete its execution. In this state, the signal PCWrite is asserted to cause the PC to be written. By setting PCSource to 10, the value supplied for writing will be the lower 26 bits of the Instruction register with 00_{two} added as the low-order bits concatenated with the upper 4 bits of the PC.

We can now put these pieces of the finite state machine together to form a specification for the control unit, as shown in Figure 5.42. In each state, the signals that are asserted are shown. The next state depends on the opcode bits of the instruction, so we label the arcs with a comparison for the corresponding instruction opcodes.

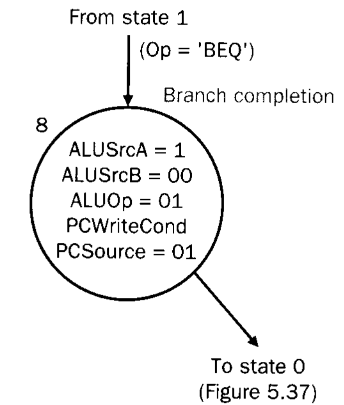


FIGURE 5.40 The branch instruction requires a single state. The first three outputs that are asserted cause the ALU to compare the registers (ALUSrcA, ALUSrcB, and ALUOp), while the signals PCSource and PCWriteCond perform the conditional write if the branch condition is true. Notice that we do not use the value written into ALUOut; instead, we use only the Zero output of the ALU. The branch target address is read from ALUOut, where it was saved at the end of state 1.

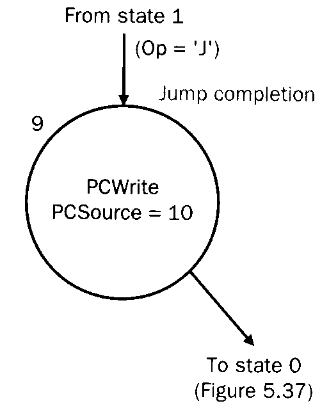


FIGURE 5.41 The jump instruction requires a single state that asserts two control signals to write the PC with the lower 26 bits of the Instruction register shifted left 2 bits and concatenated to the upper 4 bits of the PC of this instruction.

Given this implementation, and the knowledge that each state requires 1 clock cycle, we can find the CPI for a typical instruction mix.

INTEL - 1012

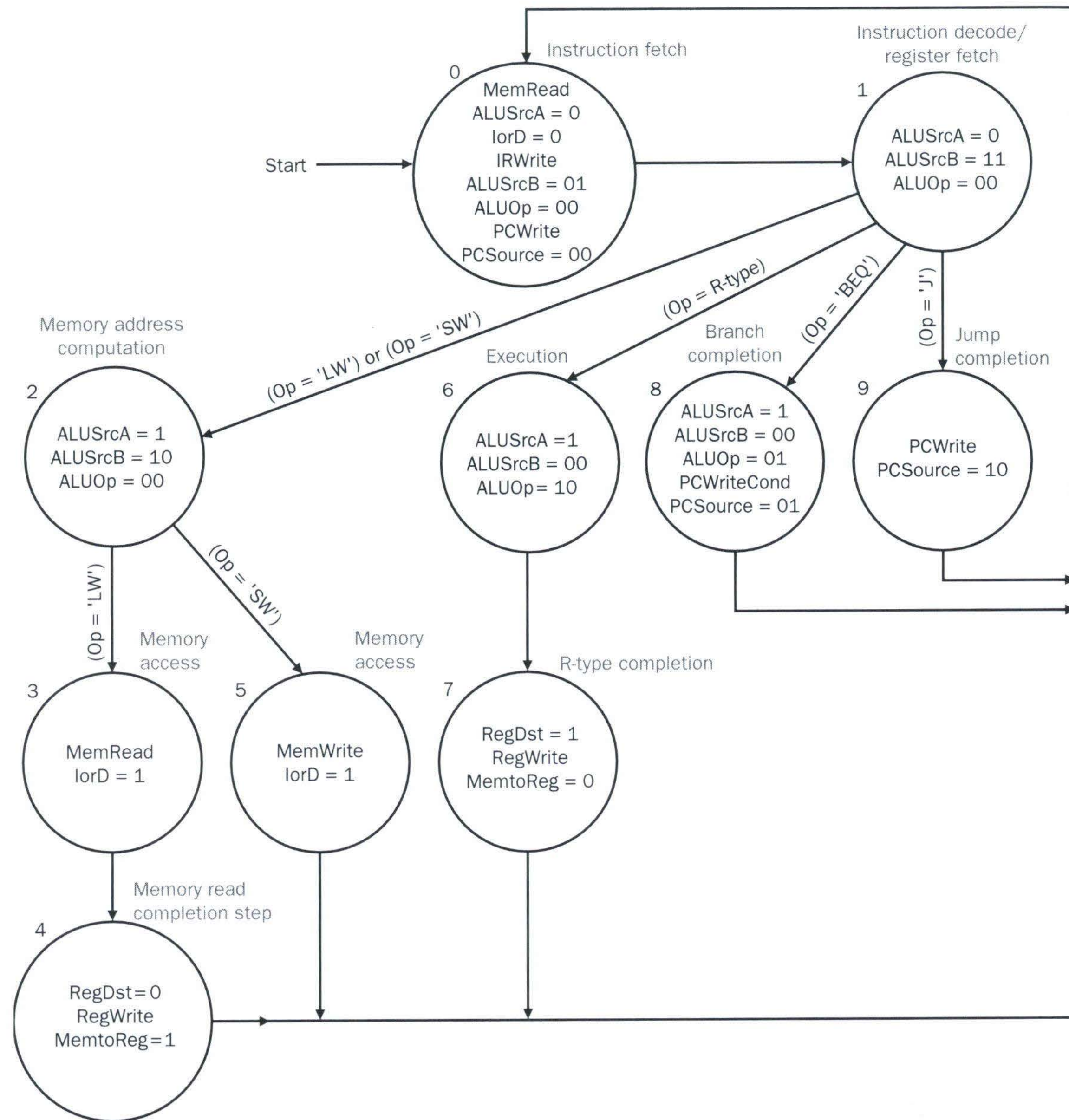


FIGURE 5.42 The complete finite state machine control for the datapath shown in Figure 5.33. The labels on the arcs are conditions that are tested to determine which state is the next state; when the next state is unconditional, no label is given. The labels inside the nodes indicate the output signals asserted during that state; we always specify the setting of a multiplexor control signal if the correct operation requires it. Hence, in some states a multiplexor control will be set to 0. In Appendix C, we examine how to turn this finite state machine into logic equations and look at how to implement those logic equations.

CPI in a Multicycle CPU

Example

Using the control shown in Figure 5.42 and the gcc instruction mix shown in Figure 4.54 on page 311, what is the CPI, assuming that each state requires 1 clock cycle?

Answer

The mix is 23% loads (1% load byte + 1% load halfword + 21% load word), 13% stores (1% store byte + 12% store word), 19% branches (9% BEQ, 8% BNE, 1% BLTZ, 1% BGEZ), 2% jumps (1% jal + 1% jr), and 43% ALU (all the rest of the mix). From Figure 5.42, the number of clock cycles for each instruction class is the following:

- Loads: 5
- Stores: 4
- ALU instructions: 4
- Branches: 3
- Jumps: 3

The CPI is given by the following:

$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction count}} = \frac{\sum \text{Instruction count}_i \times \text{CPI}_i}{\text{Instruction count}}$$

$$= \sum \frac{\text{Instruction count}_i}{\text{Instruction count}} \times \text{CPI}_i$$

The ratio

$$\frac{\text{Instruction count}_i}{\text{Instruction count}}$$

is simply the instruction frequency for the instruction class i. We can therefore substitute to obtain

$$\text{CPI} = 0.23 \times 5 + 0.13 \times 4 + 0.43 \times 4 + 0.19 \times 3 + 0.02 \times 3 = 4.02$$

This CPI is better than the worst-case CPI would have been if all the instructions took the same number of clock cycles (5).

INTEL - 1012

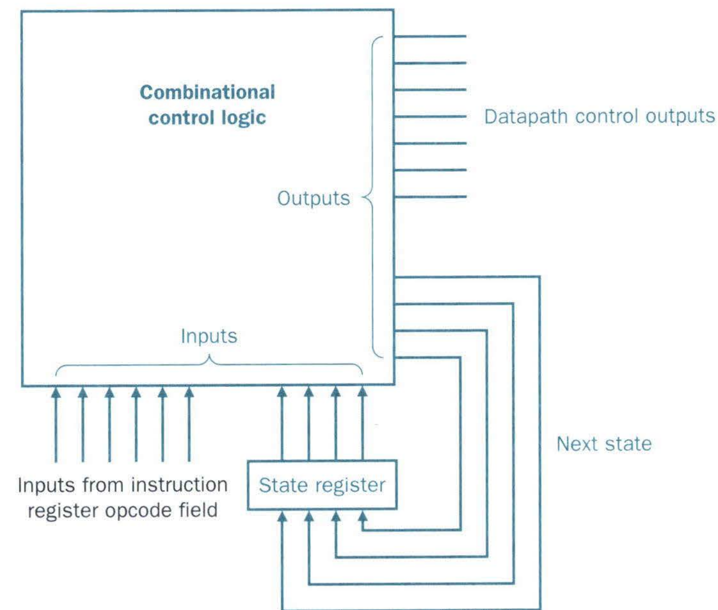


FIGURE 5.43 Finite state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state. The outputs of the combinational logic are the next-state number and the control signals to be asserted for the current state. The inputs to the combinational logic are the current state and any inputs used to determine the next state. In this case, the inputs are the instruction register opcode bits. Notice that in the finite state machine used in this chapter, the outputs depend only on the current state, not on the inputs. The following elaboration explains this in more detail.

A finite state machine can be implemented with a temporary register that holds the current state and a block of combinational logic that determines both the datapath signals to be asserted as well as the next state. Figure 5.43 shows how such an implementation might look. Appendix C describes in detail how the finite state machine is implemented using this structure. In section C.3, the combinational control logic for the finite state machine of Figure 5.42 is implemented both with a ROM (read-only memory) and a PLA (programmable logic array). (Also see Appendix B for a description of these logic elements.) In the next section of this chapter, we consider another way to represent control. Both of these techniques are simply different representations of the same control information.

Elaboration: The style of finite state machine in Figure 5.43 is called a Moore machine, after Edward Moore. Its identifying characteristic is that the output depends only on the current state. For a Moore machine, the box labeled combinational control logic can be split into two pieces. One piece has the control output and only the state input, while the other has only the next-state output.

An alternative style of machine is a Mealy machine, named after George Mealy. The Mealy machine allows both the input and the current state to be used to determine the output. Moore machines have potential implementation advantages in speed and size of the control unit. The speed advantages arise because the control outputs, which are needed early in the clock cycle, do not depend on the inputs, but only on the current state. In Appendix C, when the implementation of this finite state machine is taken down to logic gates, the size advantage can be clearly seen.The potential disadvantage of a Moore machine is that it may require additional states. For example, in situations where there is a one-state difference between two sequences of states, the Mealy machine may unify the states by making the outputs depend on the inputs.

5.5 Microprogramming: Simplifying Control Design

For the control of our simple MIPS subset, a graphical representation of the finite state machine, as in Figure 5.42, is certainly adequate. We can draw such a diagram on a single page and translate it into equations (see Appendix C) without generating too many errors. Consider instead an implementation of the full MIPS instruction set, which contains over 100 instructions (see Appendix A). In one implementation, instructions take from 1 clock cycle to over 20 clock cycles. Clearly, the control function will be much more complex. Or consider an instruction set with more instructions of widely varying classes: The control unit could easily require thousands of states with hundreds of different sequences. For example, the Intel 80x86 instruction set has many more addressing mode combinations, as well as a much larger set of opcodes.

In such cases, specifying the control unit with a graphical representation will be cumbersome, since the finite state machine can contain hundreds to thousands of states, and even more arcs! The graphical representation—although useful for a small finite state machine—will not fit on a page, let alone be understandable, when it becomes very large. Programmers know this phenomenon quite well: As programs become large, additional structuring techniques (for example, procedures and modules) are needed to keep the programs comprehensible. Of course, specifying complex control functions directly as equations, without making any mistakes, becomes essentially impossible.

Can we use some of the ideas from programming to help create a method of specifying the control that will make it easier to understand as well as to design? Suppose we think of the set of control signals that must be asserted in a state as an instruction to be executed by the datapath. To avoid confusing the instructions of the MIPS instruction set with these low-level control instructions, the latter are called *microinstructions*. Each microinstruction defines the

INTEL - 1012

set of datapath control signals that must be asserted in a given state. Executing a microinstruction has the effect of asserting the control signals specified by the microinstruction.

In addition to defining which control signals must be asserted, we must also specify the sequencing—what microinstruction should be executed next? In the finite state machine shown in Figure 5.42 on page 396, the next state is determined in one of two different ways. Sometimes a single next state follows the current state unconditionally. For example, state 1 always follows state 0, and the only way to reach state 1 is via state 0. In other cases, the choice of the next state depends on the input. This is true in state 1, which has four different successor states.

When we write programs, we also have an analogous situation. Sometimes a group of instructions should be executed sequentially, and sometimes we need to branch. In programming, the default is sequential execution, while branching must be indicated explicitly. In describing the control as a program, we also assume that microinstructions written sequentially are executed in sequence, while branching must be indicated explicitly. The default sequencing mechanism can still be implemented using a structure like the one in Figure 5.43 on page 398; however, it is often more efficient to implement the default sequential state using a counter. We will see how such an implementation looks at the end of this section.

Designing the control as a program that implements the machine instructions in terms of simpler microinstructions is called *microprogramming*. The key idea is to represent the asserted values on the control lines symbolically, so that the microprogram is a representation of the microinstructions, just as assembly language is a representation of the machine instructions. In choosing a syntax for an assembly language, we usually represent the machine instructions as a series of fields (opcode, registers, and offset or immediate field); likewise, we will represent a microinstruction syntactically as a sequence of fields whose functions are related.

Defining a Microinstruction Format

The microprogram is a symbolic representation of the control that will be translated by a program to control logic. In this way, we can choose how many fields a microinstruction should have and what control signals are affected by each field. The format of the microinstruction should be chosen so as to simplify the representation, making it easier to write and understand the microprogram. For example, it is useful to have one field that controls the ALU and a set of three fields that determine the two sources for the ALU operation as well as the destination of the ALU result. In addition to readability, we would also like the microprogram format to make it difficult or impossible to write inconsistent microinstructions. A microinstruction is inconsistent if it requires that a given control signal be set to two different values. We will see an example of how this could happen shortly.

To avoid a format that allows inconsistent microinstructions, we can make each field of the microinstruction responsible for specifying a nonoverlapping set of control signals. To choose how to make this partition of the control signals for this implementation into microinstruction fields, it is useful to reexamine two previous figures:

- Figure 5.33, on page 383, which shows all the control signals and how they affect the datapath
- Figure 5.34, on page 384, which shows the function of each datapath control signal

Signals that are never asserted simultaneously may share the same field. Figure 5.44 shows how the microinstruction can be broken into seven fields and defines the general function of each field. The first six fields of the microinstruction control the datapath, while the Sequencing field (the seventh field) specifies how to select the next microinstruction.

Microinstructions are usually placed in a ROM or a PLA (both described in Appendix B and used to implement control in Appendix C), so we can assign addresses to the microinstructions. The addresses are usually given out sequentially, in the same way that we chose sequential numbers for the states in the finite state machine. Three different methods are available to choose the next microinstruction to be executed:

1. Increment the address of the current microinstruction to obtain the address of the next microinstruction. This sequential behavior is indicated in the microprogram by putting `Seq` in the Sequencing field. Since sequential execution of instructions is encountered often, many microprogramming systems make this the default.

Field name	Function of field
ALU control	Specify the operation being done by the ALU during this clock; the result is always written in ALUOut.
SRC1	Specify the source for the first ALU operand.
SRC2	Specify the source for the second ALU operand.
Register control	Specify read or write for the register file, and the source of the value for a write.
Memory	Specify read or write, and the source for the memory. For a read, specify the destination register.
PCWrite control	Specify the writing of the PC.
Sequencing	Specify how to choose the next microinstruction to be executed.

FIGURE 5.44 Each microinstruction contains these seven fields. The values for each field are shown in Figure 5.45.

INTEL - 1012

2. Branch to the microinstruction that begins execution of the next MIPS instruction. We will label this initial microinstruction (corresponding to state 0) as `Fetch` and place the indicator `Fetch` in the Sequencing field to indicate this action.

3. Choose the next microinstruction based on the control unit input. Choosing the next microinstruction on the basis of some input is called a *dispatch*. Dispatch operations are usually implemented by creating a table containing the addresses of the target microinstructions. This table is indexed by the control unit input and may be implemented in a ROM or in a PLA. There are often multiple dispatch tables; for this implementation, we will need two dispatch tables, one to dispatch from state 1 and one to dispatch from state 2. We indicate that the next microinstruction should be chosen by a dispatch operation by placing `Dispatch i`, where `i` is the dispatch table number, in the Sequencing field.

Figure 5.45 gives a description of the values allowed for each field of the microinstruction and the effect of the different field values. Remember that the microprogram is a symbolic representation. This microinstruction format is just one example of many potential formats.

Elaboration: The basic microinstruction format may allow combinations that cannot be supported within the datapath. Typically, a microassembler will perform checks on the microinstruction fields to ensure that such inconsistencies are flagged as errors and corrected. An alternative is to structure the microinstruction format to avoid this, but this might make the microinstruction harder to read. Most microprogramming systems choose readability and require the microcode assembler to detect inconsistencies.

Creating the Microprogram

Now let's create the microprogram for the control unit. We will label the instructions in the microprogram with symbolic labels, which can be used to specify the contents of the dispatch tables (see section C.5 in Appendix C for a discussion of how the dispatch tables are defined and assembled). In writing the microprogram, there are two situations in which we may want to leave a field of the microinstruction blank. When a field that controls a functional unit or that causes state to be written (such as the Memory field or the ALU dest field) is blank, no control signals should be asserted. When a field *only* specifies the control of a multiplexor that determines the input to a functional unit, such as the SRC1 field, leaving it blank means that we do not care about the input to the functional unit (or the output of the multiplexor).

Field name	Values for field	Function of field with specific value
Label	Any string	Used to specify labels to control microcode sequencing. Labels that end in a 1 or 2 are used for dispatching with a jump table that is indexed based on the opcode. Other labels are used as direct targets in the microinstruction sequencing. Labels do not generate control signals directly but are used to define the contents of dispatch tables and generate control for the Sequencing field.
ALU control	`Add`	Cause the ALU to add.
	`Subt`	Cause the ALU to subtract; this implements the compare for branches.
	`Func code`	Use the instruction's funct field to determine ALU control.
SRC1	`PC`	Use the PC as the first ALU input.
	`A`	Register A is the first ALU input.
SRC2	`B`	Register B is the second ALU input.
	`4`	Use 4 for the second ALU input.
	`Extend`	Use output of the sign extension unit as the second ALU input.
	`Extshft`	Use the output of the shift-by-two unit as the second ALU input.
Register control	`Read`	Read two registers using the rs and rt fields of the IR as the register numbers, putting the data into registers A and B.
	`Write ALU`	Write the register file using the rd field of the IR as the register number and the contents of ALUOut as the data.
	`Write MDR`	Write the register file using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	`Read PC`	Read memory using the PC as address; write result into IR (and the MDR).
	`Read ALU`	Read memory using ALUOut as address; write result into MDR.
	`Write ALU`	Write memory using the ALUOut as address; contents of B as the data.
PCWrite control	`ALU`	Write the output of the ALU into the PC.
	`ALUOut-cond`	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	`Jump address`	Write the PC with the jump address from the instruction.
Sequencing	`Seq`	Choose the next microinstruction sequentially.
	`Fetch`	Go to the first microinstruction to begin a new instruction.
	`Dispatch i`	Dispatch using the ROM specified by i (1 or 2).

FIGURE 5.45 Each field of the microinstruction has a number of values that it can take on. The second column gives the possible values that are legal for the field, and the third column defines the effect of that value. Each field value, other than the label field, is mapped to a particular setting of the datapath control lines; this mapping is described in Appendix C, section C.5. That section also shows how the label field is used to generate the dispatch tables. As we will see, the microcode implementation will differ slightly from the finite state machine control, but only in ways that do not affect instruction semantics.

The easiest way to understand the microprogram is to break it into pieces that deal with each component of instruction execution, just as we did when we designed the finite state machine.

The first component of every instruction execution is to fetch the instructions, decode them, and compute both the sequential PC and branch target PC. These actions correspond directly to the first two steps of execution described on pages 385 through 388. The two microinstructions needed for these first two steps are shown below:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1

To understand what each microinstruction does, it is easiest to look at the effect of a group of fields. In the first microinstruction, the fields asserted and their effects are the following:

Fields	Effect
ALU control, SRC1, SRC2	Compute PC + 4. (The value is also written into ALUOut, though it will never be read from there.)
Memory	Fetch instruction into IR.
PCWrite control	Causes the output of the ALU to be written into the PC.
Sequencing	Go to the next microinstruction.

The label field, containing the label Fetch, will be used in the Sequencing field when the microprogram wants to start the execution of the next instruction.

For the second microinstruction, the operations controlled by the microinstruction are the following:

Fields	Effect
ALU control, SRC1, SRC2	Store PC + sign extension (IR[15–0]) << 2 into ALUOut.
Register control	Use the rs and rt fields to read the registers placing the data in A and B.
Sequencing	Use dispatch table 1 to choose the next microinstruction address.

We can think of the dispatch operation as a *case* or *switch* statement with the opcode field and the dispatch table 1 used to select one of four different microinstruction sequences with one of four different labels (all ending in "1"):

- Mem1 for memory-reference instructions
- Rformat1 for R-type instructions
- BEQ1 for the branch equal instruction
- JUMP1 for the jump instruction

The microprogram for memory-reference instructions has four microinstructions, as shown below. The first instruction does the memory address calculation. A two-instruction sequence is needed to complete a load (memory read followed by register file write), while the store requires only one microinstruction after the memory address calculation:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch

Let's look at the fields of the first microinstruction in this sequence:

Fields	Effect
ALU control, SRC1, SRC2	Compute the memory address: Register (rs) + sign-extend (IR[15–0]), writing the result into ALUOut.
Sequencing	Use the second dispatch table to jump to the microinstruction labeled either LW2 or SW2.

The first microinstruction in the sequence specific to lw is labeled LW2, since it is reached by a dispatch through table 2. This microinstruction has the following effect:

Fields	Effect
Memory	Read memory using the ALUOut as the address and writing the data into the MDR.
Sequencing	Go to the next microinstruction.

The next microinstruction completes execution with a microinstruction that has the following effects:

Fields	Effect
Register control	Write the contents of the MDR into the register file entry specified by rt.
Sequencing	Go to the microinstruction labeled Fetch.

INTEL - 1012

The store microinstruction, labeled SW2, operates similarly to the load microinstruction labeled LW2:

Fields	Effect
Memory	Write memory using contents of ALUOut as the address and the contents of B as the value.
Sequencing	Go to the microinstruction labeled Fetch.

The microprogram sequence for R-type instructions consists of two microinstructions: the first does the ALU operation (and is labeled Rformat1 for dispatch purposes), while the second writes the result into the register file:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch

You might think that because the fields of these two microinstructions do not conflict (i.e., each uses different fields), you could combine them into one. Indeed, microcode optimizers perform such operations when compiling microcode. In this case, however, the result of the ALU instruction is written into the register ALUOut, and the written value cannot be read until the next clock cycle; hence we cannot combine them into one microinstruction. (If you did combine them, you'd end up writing the wrong thing into the register file!) You could try to remove the ALUOut register to allow the two microinstructions to be combined, but this would require lengthening the clock cycle to allow the register file write to occur in the same clock cycle as the ALU operation.

The first microinstruction initiates the ALU operation:

Fields	Effect
ALU control, SRC1, SRC2	The ALU operates on the contents of the A and B registers, using the function field to specify the ALU operation.
Sequencing	Go to the next microinstruction.

The second microinstruction causes the ALU output to be written in the register file:

Fields	Effect
Register control	The value in ALUOut is written into the register file entry specified by the rd field.
Sequencing	Go to the microinstruction labeled Fetch.

Because the immediately previously executed microinstruction computed the branch target address, the microprogram sequence for branch, labeled with BEQ, requires just one microinstruction:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
BEQ1	Subt	A	B			ALUOut-cond	Fetch

The asserted fields of this microinstruction are the following:

Fields	Effect
ALU control, SRC1, SRC2	The ALU subtracts the operands in A and B to generate the Zero output.
PCWrite control	Causes the PC to be written using the value already in ALUOut, if the Zero output of the ALU is true.
Sequencing	Go to the microinstruction labeled Fetch.

The jump microcode sequence also consists of one microinstruction:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
JUMP1						Jump address	Fetch

Only two fields of this microinstruction are asserted:

Fields	Effect
PCWrite control	Causes the PC to be written using the jump target address.
Sequencing	Go to the microinstruction labeled Fetch.

The entire microprogram appears in Figure 5.46. It consists of the 10 microinstructions appearing above. This microprogram matches the 10-state finite state machine we designed earlier, since they were both derived from the same five-step execution sequence for the instructions. In more complex machines, the microprogram sequence might consist of hundreds or thousands of microinstructions and would be the representation of choice for the control. Datapaths of more complex machines typically require additional scratch registers used for holding intermediate results when implementing complex multicycle instructions. Registers A and B are like such scratch registers, but datapaths for more complex instruction sets often have a larger number of such registers

INTEL - 1012

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

FIGURE 5.46 The microprogram for the control unit. Recall that the labels are used to determine the targets for the dispatch operations. `Dispatch 1` does a jump based on the IR to a label ending with a 1, while `Dispatch 2` does a jump based on the IR to a label ending with 2.

with a richer set of interconnections to other datapath elements. These registers are available to the microprogrammer and make the analogy of implementing the control as a programming task even stronger.

Implementing the Microprogram

Translating a microprogram into hardware involves two aspects: deciding how to implement the sequencing function and choosing a method of storing the main control function. The microprogram can be thought of as a text representation of a finite state machine, and implemented in exactly the same way we would implement a finite state machine: using a PLA to encode both the sequencing function as well as the main control (see Figure 5.43 on page 398). Often, however, both the implementation of the sequencing function, as well as the implementation of the main control function, are done differently, especially for large microprograms.

The alternative form of implementation involves storing the control function in a read-only memory (ROM) and implementing the sequencing function separately. Figure 5.47 shows this different way to implement the sequencing function: using an incrementer to choose the next microinstruction. In this type of implementation, the microcode storage would determine the values of the datapath control lines, as well as *how to select* the next state (as opposed to *specifying* the next state, as in our finite state machine implementation). The address select logic would contain the dispatch tables, implemented in ROMs or PLAs, and would, under the control of the address select outputs, determine the next microinstruction to execute. The advantage of this implementation of the sequencing function is that it removes the logic to implement normal

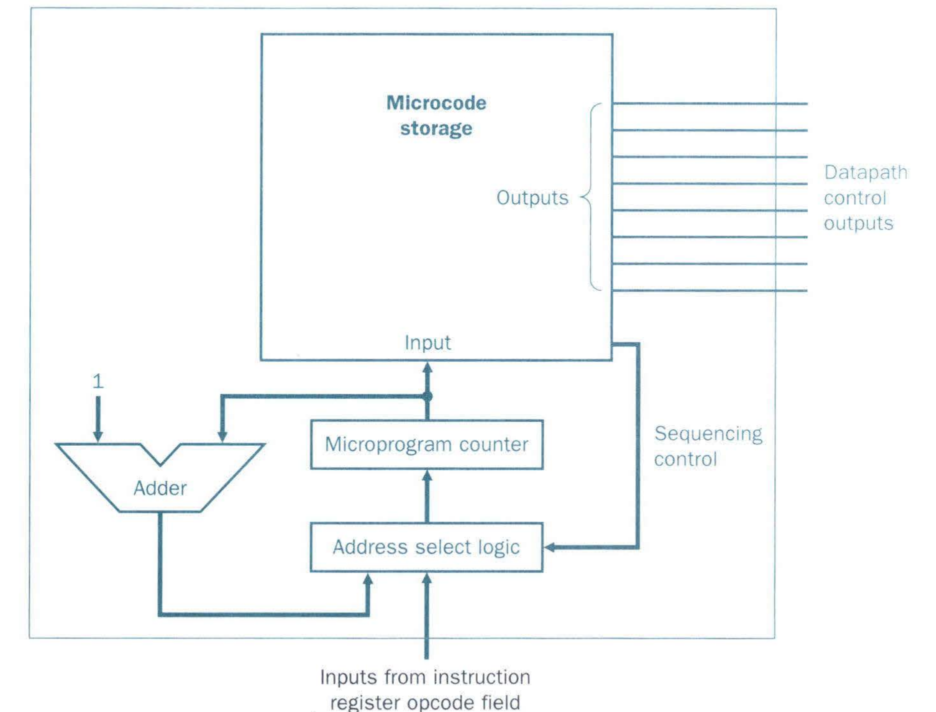


FIGURE 5.47 A typical implementation of a microcode controller would use an explicit incrementer to compute the default sequential next state and would place the microcode in a read-only memory. The microinstructions, used to set the datapath control, are assembled directly from the microprogram. The microprogram counter, which replaces the state register of a finite state machine controller, determines how the next microinstruction is chosen. The address select logic contains the dispatch tables as well as the logic to select from among the alternative next states; the selection of the next microinstruction is controlled by the sequencing control outputs from the control logic. The combination of the current microprogram counter, incrementer, dispatch tables, and address select logic forms a sequencer that selects the next microinstruction. The microcode storage may consist either of read-only memory (ROM) or may be implemented by a PLA. PLAs may be more efficient in VLSI implementations, while ROMs may be easier to change. Further discussions of the advantages of these two alternatives can be found in section 5.9 and in Appendix C.

sequencing of microinstructions, implementing such sequencing with a counter. Thus, in cases where there are long sequences of microinstructions, the explicit sequencer can result in less logic in the microcode controller.

In Figure 5.47, the main control function could be implemented in ROM, rather than implemented in a PLA. With a ROM implementation, the microprogram is assembled and stored in microcode storage and is addressed by the microprogram counter, in much the same way as a normal program is stored in program memory and the next instruction is chosen by the program counter.

INTEL - 1012

This analogy with programming is both the origin of the terminology (microcode, microprogramming, etc.) and the initial method by which microprograms were implemented (see section 5.10).

Although the type of sequencer shown in Figure 5.47 is typically used to implement a microprogram control specification, it can also be used to implement a finite state specification. Section C.4 of Appendix C describes how to generate such a sequencer in more detail. Section C.5 describes how a microprogram can be translated to such an implementation. Similarly, Appendix C shows how the control function can be implemented in either a ROM or a PLA and discusses the trade-offs. In total, Appendix C shows how to go from the symbolic representations of finite state machines or microprograms shown in this chapter to either bits in a memory or entries in a PLA. If you are interested in detailed implementation or the translation process, you may want to proceed to Appendix C.

The choice of which way to represent the control (finite state diagram versus microprogram) and how to implement control (PLA versus ROM and encoded state versus explicit sequencer) are independent decisions, affected by both the structure of the control function and the technology used to implement the control. We return to these issues briefly in section 5.9, but before we do that we need to look at one of the hardest aspects of control: exceptions.

5.6 Exceptions

Control is the most challenging aspect of processor design: it is both the hardest part to get right and the hardest part to make fast. One of the hardest parts of control is implementing *exceptions* and *interrupts*—events other than branches or jumps that change the normal flow of instruction execution. An exception is an unexpected event from within the processor; arithmetic overflow is an example of an exception. An interrupt is an event that also causes an unexpected change in control flow but comes from outside of the processor. Interrupts are used by I/O devices to communicate with the processor, as we will see in Chapter 8.

Many architectures and authors do not distinguish between interrupts and exceptions, often using the older name *interrupt* to refer to both types of events. We follow the MIPS convention, using the term *exception* to refer to *any* unexpected change in control flow without distinguishing whether the cause is internal or external; we use the term *interrupt* only when the event is externally caused. The Intel 80x86 architecture uses the word *interrupt* for all these events, while the PowerPC architecture uses the word *exception* to indicate that an unusual event has occurred and *interrupt* to indicate the change in control flow.

Interrupts were initially created to handle unexpected events like arithmetic overflow and to signal requests for service from I/O devices. The same basic mechanism was extended to handle internally generated exceptions as well. Here are some examples showing whether the situation is generated internally by the processor or externally generated:

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Many of the requirements to support exceptions come from the specific situation that causes an exception to occur. Accordingly, we will return to this topic in Chapter 7, when we discuss memory hierarchies, and in Chapter 8, when we discuss I/O, and we better understand the motivation for additional capabilities in the exception mechanism. In this section, we deal with the control implementation for detecting two types of exceptions that arise from the portions of the instruction set and implementation that we have already discussed.

Detecting exceptional conditions and taking the appropriate action is often on the critical timing path of a machine, which determines the clock cycle time and thus performance. Without proper attention to exceptions during design of the control unit, attempts to add exceptions to a complicated implementation can significantly reduce performance, as well as complicate the task of getting the design correct.

How Exceptions Are Handled

The two types of exceptions that our current implementation can generate are execution of an undefined instruction and an arithmetic overflow. The basic action that the machine must perform when an exception occurs is to save the address of the offending instruction in the exception program counter (EPC) and then transfer control to the operating system at some specified address.

The operating system can then take the appropriate action, which may involve providing some service to the user program, taking some predefined action in response to an overflow, or stopping the execution of the program and reporting an error. After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its execution, using the EPC to determine where to restart the execution of the program. In Chapter 7, we will look more closely at the issue of restarting the execution.

INTEL - 1012

For the operating system to handle the exception, it must know the reason for the exception, in addition to the instruction that caused it. There are two main methods used to communicate the reason for an exception. The method used in the MIPS architecture is to include a status register (called the *Cause register*), which holds a field that indicates the reason for the exception.

A second method is to use *vectored interrupts*. In a vectored interrupt, the address to which control is transferred is determined by the cause of the exception. For example, to accommodate the two exception types listed above, we might define the following:

Exception type	Exception vector address (in hex)
Undefined instruction	$C0\ 00\ 00\ 00_{hex}$
Arithmetic overflow	$C0\ 00\ 00\ 20_{hex}$

The operating system knows the reason for the exception by the address at which it is initiated. The addresses are separated by 32 bytes or 8 instructions, and the operating system must record the reason for the exception and may perform some limited processing in this sequence. When the exception is not vectored, a single entry point for all exceptions can be used, and the operating system decodes the status register to find the cause.

We can perform the processing required for exceptions by adding a few extra registers and control signals to our basic implementation and by slightly extending the finite state machine. Let's assume that we are implementing the exception system used in the MIPS architecture. (Implementing vectored exceptions is no more difficult.) We will need to add two additional registers to the datapath:

- *EPC:* A 32-bit register used to hold the address of the affected instruction. (Such a register is needed even when exceptions are vectored.)
- *Cause:* A register used to record the cause of the exception. In the MIPS architecture, this register is 32 bits, although some bits are currently unused. Assume that the low-order bit of this register encodes the two possible exception sources mentioned above: undefined instruction = 0 and arithmetic overflow = 1.

We will need to add two control signals to cause the EPC and Cause registers to be written; call these *EPCWrite* and *CauseWrite*. In addition, we will need a 1-bit control signal to set the low-order bit of the Cause register appropriately; call this signal *IntCause*. Finally, we will need to be able to write the *exception address*, which is the operating system entry point for exception handling, into the PC; let's assume that this address is $C0000000_{hex}$. Currently, the PC is fed from the output of a three-way multiplexor, which is controlled by the signal PCSource (see Figure 5.33 on page 383). We can change this to a four-way multiplexor, with additional input wired to the constant value $C0000000_{hex}$. Then PCSource can be set to 11_{two} to select this value to be written into the PC.

Because the PC is incremented during the first cycle of every instruction, we cannot just write the value of the PC into the EPC, since the value in the PC will be the instruction address plus four. However, we can use the ALU to subtract four from the PC and write the output into the EPC. This requires no additional control signals or paths, since we can use the ALU to subtract, and the constant 4 is already a selectable ALU input. The data write port of the EPC, therefore, is connected to the ALU output. Figure 5.48 shows the multicycle datapath with these additions needed for implementing exceptions.

Using the datapath of Figure 5.48, the action to be taken for each different type of exception can be handled in one state apiece. In each case, the state sets the Cause register, computes and saves the original PC into the EPC, and writes the exception address into the PC. Thus, to handle the two exception types we are considering, we will need to add only the two states shown in Figure 5.49.

To connect this finite state machine to the finite state machine of the main control unit, we must determine how to detect exceptions and add arcs that transfer control from the main execution machine to this exception-handling finite state machine.

How Control Checks for Exceptions

Now we have to design a method to detect these exceptions and to transfer control to the appropriate state in the exception states shown in Figure 5.49. Each of the two possible exceptions is detected differently:

- *Undefined instruction*: This is detected when no next state is defined from state 1 for the op value. We handle this exception by defining the next-state value for all op values other than `lw`, `sw`, 0 (R-type), `j`, and `beq` as state 10. We show this by symbolically using *other* to indicate that the op field does not match any of the opcodes that label arcs out of state 1. A modified finite state diagram is shown in Figure 5.50.
- *Arithmetic overflow*: Chapter 4 included logic in the ALU to detect overflow, and a signal called *Overflow* is provided as an output from the ALU. This signal is used in the modified finite state machine to specify an additional possible next state for state 7, as shown in Figure 5.50.

Figure 5.50 represents a complete specification of the control for this MIPS subset with two types of exceptions. Remember that the challenge in designing the control of a real machine is to handle the variety of different interactions between instructions and other exception-causing events in such a way that

INTEL - 1012

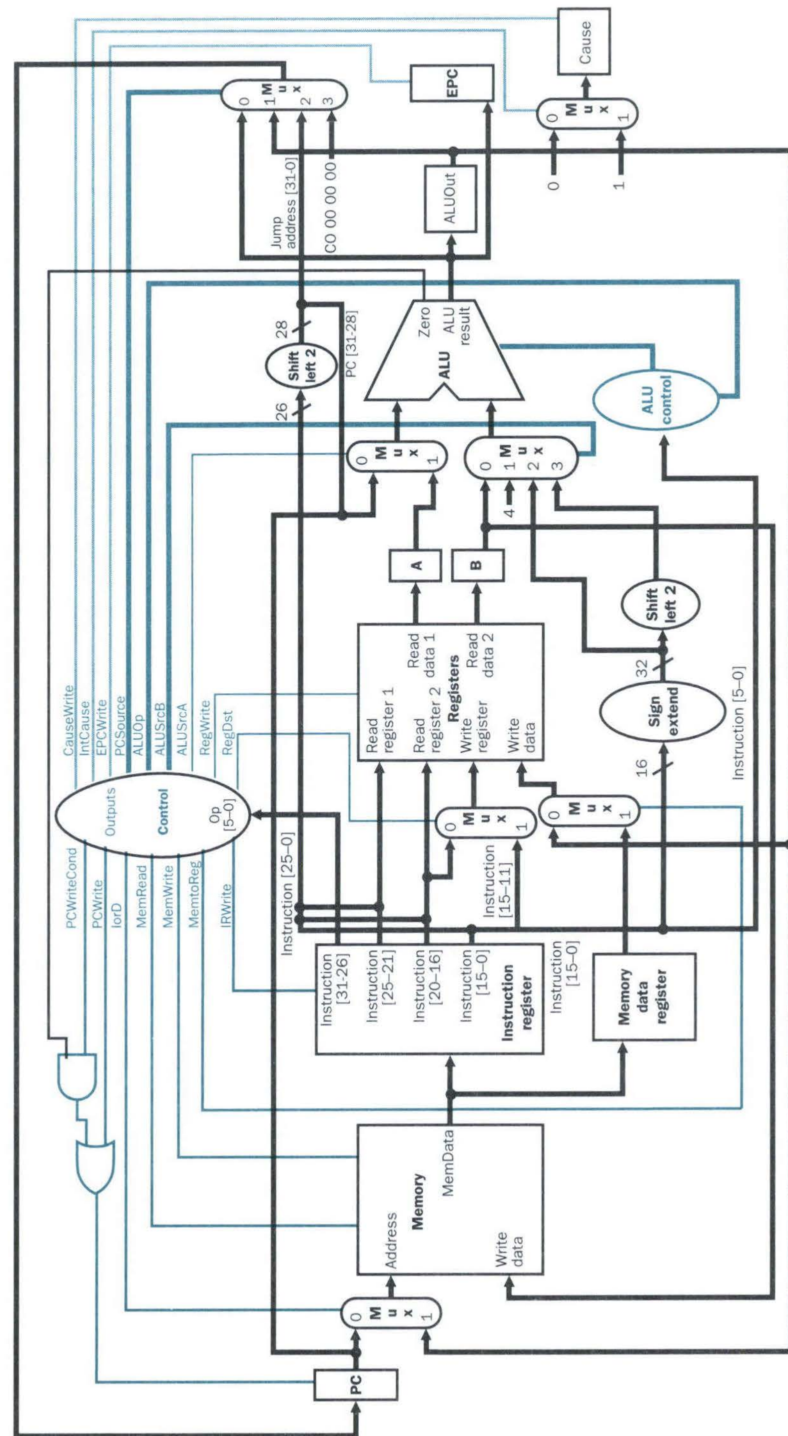


FIGURE 5.48 The multicycle datapath with the addition needed to implement exceptions. The specific additions include the Cause and EPC registers, a multiplexor to control the value sent to the Cause register, an expansion of the multiplexor controlling the value written into the PC, and control lines for the added multiplexor and registers. For simplicity, this figure does not show the ALU overflow signal, which would need to be stored in a one-bit register and delivered as an additional input to the control unit (see Figure 5.50 to see how it is used).

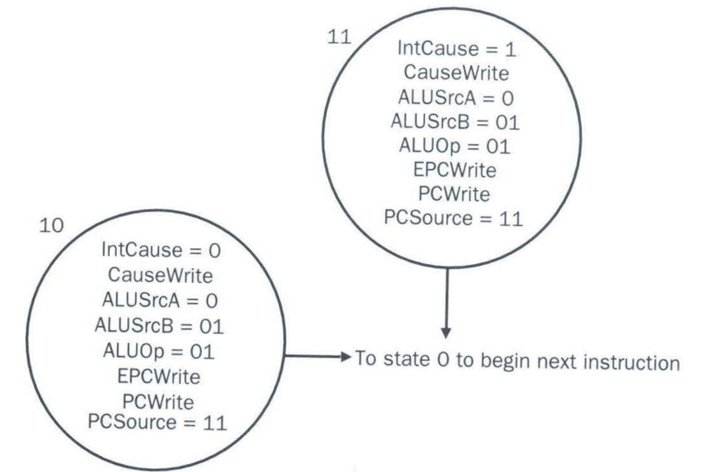


FIGURE 5.49 This pair of states handles the necessary actions for the two different exceptions we are considering. Each state provides control for three actions: setting the Cause register, getting the address of the offending instruction into the EPC, and setting the PC to the exception vector address. Both state 10 and state 11 represent the starting point for an exception. Control is transferred to one of these two states when an exception occurs. After either state 10 or state 11 is completed, control is transferred to state 0, and a new instruction is fetched.

the control logic remains both small and fast. The complex interactions that are possible are what make the control unit the most challenging aspect of hardware design.

Elaboration: If you examine the finite state machine in Figure 5.50 closely, you can see that some problems could occur in the way the exceptions are handled. For example, in the case of arithmetic overflow, the instruction causing the overflow completes writing its result because the overflow branch is in the state when the write completes. However, it's possible that the architecture defines the instruction as having no effect if the instruction causes an exception; this is what the MIPS instruction set architecture specifies. In Chapter 7, we will see that certain classes of exceptions require us to prevent the instruction from changing the machine state, and that this aspect of handling exceptions becomes complex and potentially limits performance.

INTEL - 1012

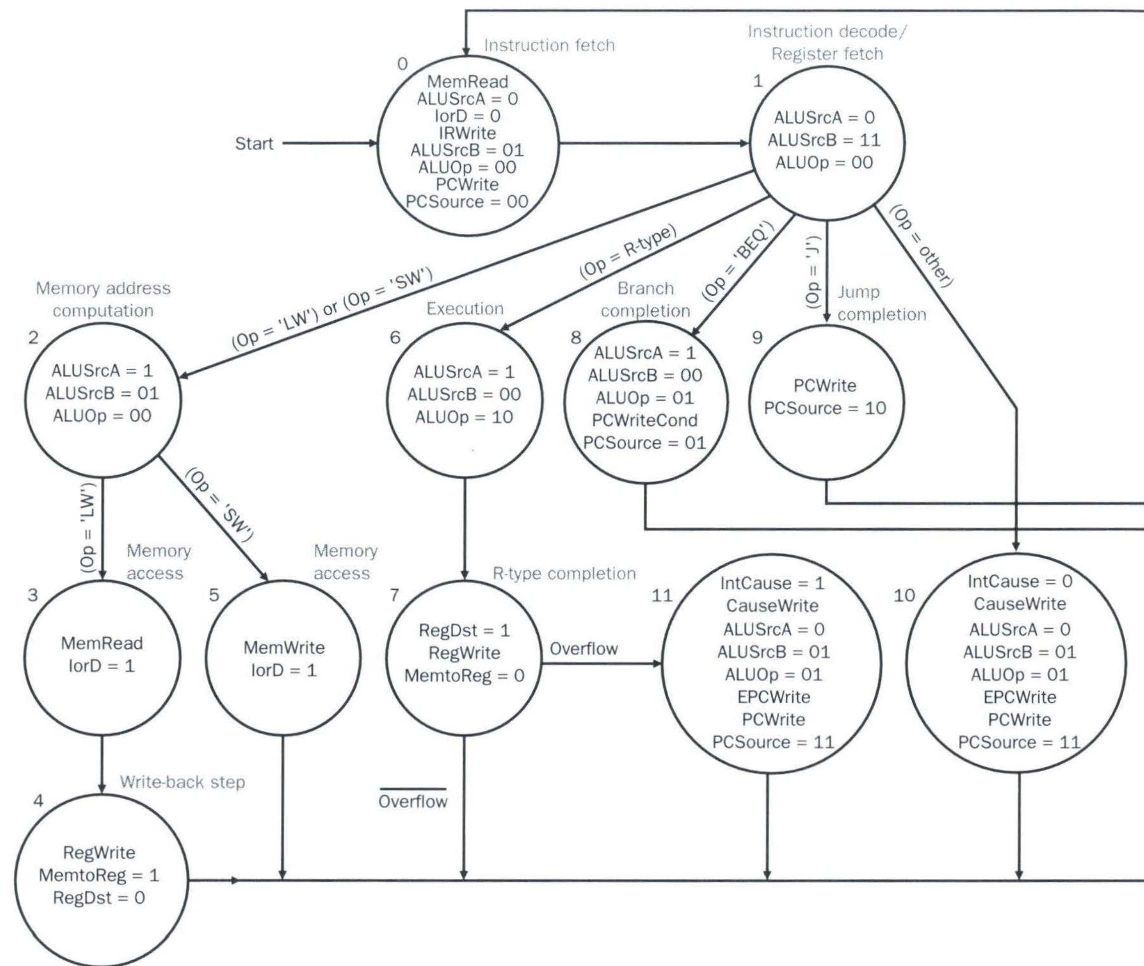


FIGURE 5.50 This shows the finite state machine with the additions to handle exception detection. States 10 and 11 come from Figure 5.49 on page 415. The branch out of state 1 labeled (*Op = other*) indicates the next state when the input does not match the opcode of any of `lw`, `sw`, 0 (R-type), `j`, or `beq`. The branch out of state 7 labeled *Overflow* indicates the action to be taken when the ALU signals an overflow.

5.7 Real Stuff: The Pentium Pro Implementation

The techniques described in this chapter for building datapaths and control units are at the heart of every computer. All recent computers, however, go beyond the techniques of this chapter and use pipelining. *Pipelining*, which is

the subject of the next chapter, improves performance by overlapping the execution of multiple instructions, achieving throughput close to one instruction per clock cycle (like our single-cycle implementation) with a clock cycle time determined by the delay of individual functional units rather than the entire execution path of an instruction (like our multicycle design). The last Intel 80x86 processor without pipelining was the 80386 introduced in 1985; the very first MIPS processor, the R2000, also introduced in 1985, was pipelined.

Recent Intel 80x86 processors (the 80486, Pentium, and Pentium Pro) employ successively more sophisticated pipelining approaches. These processors, however, are still faced with the challenge of implementing control for the complex 80x86 instruction set, described in Chapter 3. The basic functional units and datapaths in use in modern processors, while significantly more complex than those described in this chapter, have the same basic functionality and similar types of control signals. Thus the task of designing a control unit builds on the same principles used in this chapter.

Challenges Implementing More Complex Architectures

Unlike the MIPS architecture, the 80x86 architecture contains instructions that are very complex and can take tens, if not hundreds, of cycles to execute. For example, the string move instruction (`MOVS`) requires calculating and updating two different memory addresses as well as loading and storing a byte of the string. The larger number and greater complexity of addressing modes in the 80x86 architecture complicates implementation of even simple instructions similar to those on MIPS. Fortunately, a multicycle datapath is well structured to adapt to variations in the amount of work required per instruction that are inherent in 80x86 instructions. This adaptability comes from two capabilities:

1. A multicycle datapath allows instructions to take varying numbers of clock cycles. Simple 80x86 instructions that are similar to those in the MIPS architecture can execute in three or four clock cycles, while more complex instructions can take tens of cycles.
2. A multicycle datapath can use the datapath components more than once per instruction. This is critical to handling more complex addressing modes, as well as implementing more complex operations, both of which are present in the 80x86 architecture. Without this capability the datapath would need to be extended to handle the demands of the more complex instructions without reusing components, which would be completely impractical. For example, a single-cycle datapath, which doesn't reuse components, for the 80x86 would require several data memories and a very large number of ALUs.

INTEL - 1012

Using the multicycle datapath and a microprogrammed controller provides a framework for implementing the 80x86 instruction set. The challenging task, however, is creating a high-performance implementation, which requires dealing with the diversity of the requirements arising from different instructions. Simply put, a high-performance implementation needs to ensure that the simple instructions execute quickly, and that the burden of the complexities of the instruction set penalize primarily the complex, less frequently used, instructions.

To accomplish this goal, every Intel implementation of the 80x86 architecture since the 486 has used a combination of hardwired control to handle simple instructions, and microcoded control to handle the more complex instructions. For those instructions that can be executed in a single pass through the datapath (i.e., those with complexity similar to a MIPS instruction), the hardwired control generates the control information and executes the instruction in one pass through the datapath that takes a small number of clock cycles. Those instructions that require multiple datapath passes and complex sequencing are handled by the microcoded controller that takes a larger number of cycles and multiple passes through the datapath to complete the execution of the instruction. The benefit of this approach is that it enables the designer to achieve low cycle counts for the simple instructions without having to build the enormously complex datapath that would be required to handle the full generality of the most complex instructions.

The Structure of the Pentium Pro Implementation

Both the Pentium and Pentium Pro processors are capable of executing more than one instruction per clock, using an advanced pipelining technique, called *superscalar*. We describe how a superscalar processor works in the next chapter. The important thing to understand here is that executing more than one instruction per clock requires duplicating the datapath resources. The simplest way to think about this is that the processor has multiple datapaths, though these are tailored to handle one class of instructions: say, loads and stores, ALU operations, or branches. In this way, the processor is able to execute a load or store in the same clock cycle that it is also executing a branch and an ALU operation. The Pentium allows up to two such instructions to be executed in a clock cycle, while the Pentium Pro allows up to four.

The datapaths of the Pentium Pro actually execute simple microinstructions (or microoperations in Intel terminology), similar to MIPS instructions. These microinstructions are fully self-contained operations that are initially 72 bits wide. The control of datapath to implement these microinstructions is completely hardwired. This last level of control expands up to four 72-bit microinstructions into 120 control lines for the integer datapaths and 285 control lines for the floating-point datapath. This last step of expanding the microinstructions into control lines is very similar to the control generation for the single-cycle datapath or for the ALU control.

These microinstructions are generated from the 80x86 instructions either by hardwired control or by microprogrammed control. For 80x86 instructions that require less than four microinstructions to implement the 80x86 instruction, the 80x86 instruction is directly decoded into one to four microinstructions by a set of PLAs. These PLAs can generate a total of 1200 different microinstructions. If an 80x86 instruction requires more than four microinstructions, the control dispatches to a microcode control store and uses a traditional microcode sequencer to generate a sequence of five or more microinstructions. The microcode ROM provides a total of about 8000 microinstructions, with a number of sequences being shared among 80x86 instructions.

The use of simple low-level hardwired control and simple datapaths for handling the microinstructions allows the Pentium Pro to achieve impressive clock rates, similar to those for microprocessors implementing simpler instruction set architectures. Furthermore, the translation process, which combines direct hardwired control for simple instructions with microcoded control for complex instructions, allows the Pentium Pro to execute the simple, high-frequency instructions in the 80x86 instruction set at a high rate, yielding a low, and very competitive, CPI for integer instructions.

5.8 Fallacies and Pitfalls

Pitfall: Implementing a complex instruction with microcode may not be faster than a sequence using simpler instructions.

Most machines with a large and complex instruction set are implemented, at least in part, using a microcode stored in ROM. Surprisingly, on such machines, sequences of individual simpler instructions are sometimes as fast as or even faster than the custom microcode sequence for a particular instruction.

How can this possibly be true? At one time, microcode had the advantage of being fetched from a much faster memory than instructions in the program. Since caches came into use in 1968, microcode no longer has such a consistent edge in fetch time. Microcode does, however, still have the advantage of using internal temporary registers in the computation, which can be helpful on machines with few general-purpose registers. The disadvantage of microcode is that the algorithms must be selected before the machine is announced and can't be changed until the next model of the architecture. The instructions in a program, on the other hand, can utilize improvements in its algorithms at any

INTEL - 1012

time during the life of the machine. Along the same lines, the microcode sequence is probably not optimal for all possible combinations of operands.

One example of such an instruction in the 80x86 implementations is the move string instruction (MOVS) used with a repeat prefix that we discussed in Chapter 3. This instruction is often slower than a loop that moves words at a time, as we saw earlier in the Fallacies and Pitfalls (see page 185).

Another example involves the LOOP instruction, which decrements a register and branches to the specified label if the decremented register is not equal to zero. This instruction is similar to the PowerPC instruction "branch conditional to count register" (bcctr) discussed in Chapter 3. These instructions are designed to be used as the branch at the bottom of loops that have a fixed number of iterations (e.g., many *for* loops). Such an instruction, in addition to packing in some extra work, has benefits in minimizing the potential losses from the branch in pipelined machines (as we will see when we discuss branches in the next chapter).

Unfortunately, on all recent Intel 80x86 implementations, the LOOP instruction is always slower than the macrocode sequence consisting of simpler individual instructions (assuming that the small code size difference is not a factor). Thus, optimizing compilers focusing on speed never generate the LOOP instruction. This, in turn, makes it hard to motivate making LOOP fast in future implementations, since it is so rarely used!

Fallacy: If there is space in the control store, new instructions are free of cost.

One of the benefits of a microprogrammed approach is that control store implemented in ROM is not very expensive, and as transistor budgets grew, extra ROM was practically free. The analogy here is that of building a house and discovering, near completion, that you have enough land and materials left to add a room. This room wouldn't be free, however, since there would be the costs of labor and maintenance for the life of the home. The temptation to add "free" instructions can occur only when the instruction set is not fixed, as is likely to be the case in the first model of a computer. Because upward compatibility of binary programs is a highly desirable feature, all future models of this machine will be forced to include these so-called free instructions, even if space is later at a premium.

During the design of the 80286, many instructions were added to the instruction set. The availability of more silicon resource and the use of microprogrammed implementation made such additions seem painless. Possibly the largest addition was a sophisticated protection mechanism, which is largely unused, but still must be implemented in newer implementations. This addition was motivated by a perceived need for such a mechanism and the desire to enhance microprocessor architectures to provide functionality equal to that of larger computers. Likewise, a number of decimal instructions were added to provide decimal arithmetic on bytes. Such instructions are rarely used today

because using binary arithmetic on 32 bits and converting back and forth to decimal representation is considerably faster. Like the protection mechanisms, the decimal instructions must be implemented in newer processors even if only rarely used.

5.9 Concluding Remarks

As we have seen in this chapter, both the datapath and control for a processor can be designed starting with the instruction set architecture and an understanding of the basic characteristics of the technology. In section 5.2, we saw how the datapath for a MIPS processor could be constructed based on the architecture and the decision to build a single-cycle implementation. Of course, the underlying technology also affects many design decisions by dictating what components can be used in the datapath, as well as whether a single-cycle implementation even makes sense. Along the same lines, in the first portion of section 5.4, we saw how the decision to break the clock cycle into a series of steps led to the revised multicycle datapath. In both cases, the top-level organization—a single-cycle or multicycle machine—together with the instruction set, prescribed many characteristics of the datapath design.

Similarly, the control is largely defined by the instruction set architecture, the organization, and the datapath design. In the single-cycle organization, these three aspects essentially define how the control signals must be set. In the multicycle design, the exact decomposition of the instruction execution into cycles, which is based on the instruction set architecture, together with the datapath, define the requirements on the control.

Control is one of the most challenging aspects of computer design. A major reason is that designing the control requires an understanding of how all the components in the processor operate. To help meet this challenge, we examined two techniques for specifying control: finite state diagrams and microprogramming. These control representations allow us to abstract the specification of the control from the details of how to implement it. Using abstraction in this fashion is the major method we have to cope with the complexity of computer designs.

Once the control has been specified, we can map it to detailed hardware. The exact details of the control implementation will depend on both the structure of the control and on the underlying technology used to implement it. Abstracting the specification of control is also valuable because the decisions of how to implement the control are technology-dependent and likely to change over time.

INTEL - 1012

Trade-offs in Control Approaches

Much has changed since Wilkes [1953] wrote the first paper on microprogramming. The most important changes are the following:

- Control units are implemented as integral parts of the processor, often on the same silicon die. They cannot be changed independent of the rest of the processor. Furthermore, given the right computer-aided design tools, the difficulty of implementing a ROM or a PLA is the same.
- ROM, which was used to hold the microinstructions, is no longer faster than RAM, which holds the machine language program. A PLA implementation of a control function is often much smaller than the ROM implementation, which may have many duplicate or unused entries. If the PLA is smaller, it is usually faster.
- Instruction sets have become much simpler than they were in the 1960s and 1970s, leading to reduced complexity in the control.
- Computer-aided design tools have improved so that control can be specified symbolically and, by using much faster computers, thoroughly simulated before hardware is constructed. This improvement makes it plausible to get the control logic correct without the need for fixes later.

These changes have blurred the distinctions among different implementation choices. Certainly, using an abstract specification of control is helpful. How that control is then implemented depends on its size, the underlying technology, and the available CAD tools.

The Big Picture

Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique. Figure 5.51 shows the variety of methods for specifying the control and moving from the specification to an implementation using some form of structured logic.

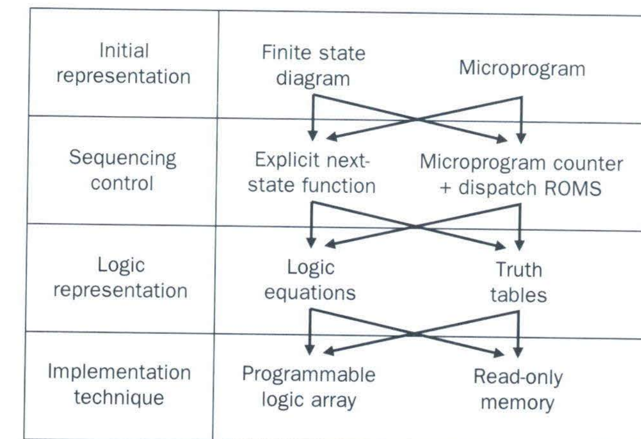


FIGURE 5.51 Alternative methods for specifying and implementing control. The arrows indicate possible design paths: any path from the initial representation to the final implementation technology is viable. Traditionally, "hardwired control" means that the techniques on the left-hand side are used, and "microprogrammed control" means that the techniques on the right-hand side are used.

5.10 Historical Perspective and Further Reading

Maurice Wilkes learned computer design in a summer workshop from Eckert and Mauchly and then went on to build the first full-scale, operational, stored-program computer—the EDSAC. From that experience he realized the difficulty of control. He thought of a more centralized control using a diode matrix and, after visiting the Whirlwind computer in the United States, wrote [Wilkes 1985]:

> *I found that it did indeed have a centralized control based on the use of a matrix of diodes. It was, however, only capable of producing a fixed sequence of eight pulses—a different sequence for each instruction, but nevertheless fixed as far as a particular instruction was concerned. It was not, I think, until I got back to Cambridge that I realized that the solution was to turn the control unit into a computer in miniature by adding a second matrix to determine the flow of control at the microlevel and by providing for conditional micro-instructions.*

Wilkes [1953] was ahead of his time in recognizing that problem. Unfortunately, the solution was also ahead of its time: To provide control, microprogramming relies on fast memory that was not available in the 1950s. Thus Wilkes's ideas remained primarily academic conjecture for a decade, although

INTEL - 1012

he did construct the EDSAC 2 using microprogrammed control in 1958 with ROM made from magnetic cores.

IBM brought microprogramming into the spotlight in 1964 with the IBM 360 family. Before this event, IBM saw itself as a cluster of many small businesses selling different machines with their own price and performance levels, but also with their own instruction sets. (Recall that little programming was done in high-level languages, so that programs written for one IBM machine would not run on another.) Gene Amdahl, one of the chief architects of the IBM 360, said that managers of each subsidiary agreed to the 360 family of computers only because they were convinced that microprogramming made it feasible. To be sure of the viability of microprogramming, the IBM vice president of engineering even visited Wilkes surreptitiously and had a "theoretical" discussion of the pros and cons of microcode. IBM believed that the idea was so important to its plans that it pushed the memory technology inside the company to make microprogramming feasible.

Stewart Tucker of IBM was saddled with the responsibility of porting software from the IBM 7090 to the new IBM 360. Thinking about the possibilities of microcode, he suggested expanding the control store to include simulators, or interpreters, for older machines. Tucker [1967] coined the term *emulation* for this, meaning full simulation at the microprogrammed level. Occasionally, emulation on the 360 was actually faster than on the original hardware.

Once the giant of the industry began using microcode, the rest soon followed. (IBM was over half of the computer industry in 1964, measured in revenue.) One difficulty in adopting microcode was that the necessary memory technology was not widely available, but that was soon solved by semiconductor ROM and later RAM. The microprocessor industry followed the same history, with the limited resources of the earliest chips forcing hardwired control. But as the resources increased, the advantages of simpler design, ease of change, and the ability to use a wide variety of underlying implementations persuaded many to use microprogramming.

In the 1960s and 1970s, microprogramming was one of the most important techniques used in implementing machines. Through most of that period, machines were implemented with discrete components or MSI (medium-scale integration—fewer than 1000 gates per chip), and designers had to choose between two types of implementations: *hardwired control* or *microprogrammed control*. Hardwired control was characterized by finite state machines using an explicit next state and implemented primarily with random logic. In this era, microprogrammed control used microcode to specify control that was then implemented with a microprogram sequencer (a counter) and ROMs. Hardwired control received its name because the control was implemented in hardware and could not be easily changed. Microprograms implemented in ROM were

also called *firmware* because they could be changed somewhat more easily than hardware, but not nearly as easily as software.

The reliance on standard parts of low- to medium-level integration made these two design styles radically different. Microprogrammed approaches were attractive because implementing the control with a large collection of low-density gates was extremely costly. Furthermore, the popularity of relatively complex instruction sets demanded a large control unit, making a ROM-based implementation much more efficient. The hardwired implementations were faster, but too costly for most machines. Furthermore, it was very difficult to get the control correct, and changing ROMs was easier than replacing a random logic control unit. Eventually, microprogrammed control was implemented in RAM, to allow changes late in the design cycle, and even in the field after a machine shipped.

With the increasing popularity of microprogramming came more sophisticated instruction sets. Over the years, most microarchitectures became more and more dedicated to support the intended instruction set, so that reprogramming for a different instruction set failed to offer satisfactory performance. With the passage of time came much larger control stores, and it became possible to consider a machine as elaborate as the VAX with more than 300 different instruction opcodes and more than a dozen memory-addressing modes. The use of RAM to store the microcode also made it possible to debug the microcode and even fix some bugs once machines were in the field. The VAX architecture represented the high-water mark for instruction set architectures based on microcode implementations. Typical implementations of the full VAX instruction set required 400 to 500 Kbits of control store.

The VAX architecture has been laid to rest and replaced by the Alpha architecture. This new architecture is based on the same principles of design used in other RISC architectures, including the MIPS, SPARC, IBM PowerPC, and the HP Precision architecture. With the disappearance of the VAX, traditional microprogramming, in which the control is implemented with one major control store, will largely disappear from conventional microprocessor designs. Even processors such as the Intel Pentium and Pentium Pro are employing large amounts of hardwired control, at least for the central core of the processor.

Of course, control unit design will continue to be a major aspect of all computers, and the best way to specify and implement the control will vary, just as computers will vary, from streamlined RISC architectures with simple control, to special-purpose processors with potentially large amounts of more complex and specialized control. One recent movement in this direction is an announcement by Sun that they will build processors designed to interpret Java. Whether such an approach is competitive with compilation, whether there is a significant market for more specialized processors, and what role microcode will play are questions that will be answered in the next few years.

INTEL - 1012

To Probe Further

Kidder, T. [1981]. *Soul of a New Machine*, Little, Brown, and Co., New York.

Describes the design of the Data General Eclipse series that replaced the first DG machines such as the Nova. Kidder records the intimate interactions among architects, hardware designers, microcoders, and project management.

Levy, H. M., and R. H. Eckhouse, Jr. [1989]. *Computer Programming and Architecture: The VAX*, Second ed., Digital Press, Bedford, MA.

Good description of the VAX architecture and several different microprogrammed implementations.

Patterson, D. A. [1983]. "Microprogramming," *Scientific American* 248:3 (March) 36–43.

Overview of microprogramming concepts.

Tucker, S. G. [1967]. "Microprogram control for the System/360," *IBM Systems J.* 6:4, 222–41.

Describes the microprogrammed control for the 360, the first microprogrammed commercial machine.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA.

Intriguing biography with many stories about industry pioneers and the trials and successes in building early machines.

Wilkes, M. V., and J. B. Stringer [1953]. "Microprogramming and the design of the control circuits in an electronic digital computer," *Proc. Cambridge Philosophical Society* 49:230–38. Also reprinted in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 158–63, 1982, and in "The Genesis of Microprogramming," in *Annals of the History of Computing* 8:116.

These two classic papers describe Wilkes's proposal for microcode.

5.11 Key Terms

This section lists the variety of major new terms introduced in this chapter, which range from elements of the datapath, to clocking methodologies, to control mechanisms, to logic structures used for control. These terms are defined in the Glossary.

branch not taken
branch taken
branch target address
control signal
datapath element
delayed branch
dispatch
don't-care term
exception or interrupt
firmware
hardwired control
macroinstruction
microcode
microinstruction
microprogram
microprogrammed control
multicycle or multiple clock cycle implementation
sign-extend
single-cycle implementation
superscalar
vectored interrupt

5.12 Exercises

5.1 [5] <§5.3> Describe the effect that a single stuck-at-0 fault (i.e., regardless of what it should be, the signal is always 0) would have on the multiplexors in the single-cycle datapath in Figure 5.19 on page 360. Which instructions, if any, would still work? Consider each of the following faults separately: RegDst = 0, ALUSrc = 0, MemtoReg = 0, Zero = 0.

5.2 [5] <§5.3> This exercise is similar to Exercise 5.1, but this time consider stuck-at-1 faults (the signal is always 1).

5.3 [5] <§5.4> This exercise is similar to Exercise 5.1, but this time consider the effect that the stuck-at-0 faults would have on the multiplexors in the multiple-cycle datapath in Figure 5.32 on page 381. Consider each of the following faults: RegDst = 0, MemtoReg = 0, IorD = 0, ALUSrcA = 0.

5.4 [5] <§5.4> This exercise is similar to Exercise 5.3, but this time consider stuck-at-1 faults (the signal is always 1).

WEB ENHANCED

5.5 [15] <§5.3> We wish to add the instruction `addi` (add immediate) to the single-cycle datapath described in this chapter. Add any necessary datapaths and control signals to the single-cycle datapath of Figure 5.19 on page 360 and show the necessary additions to Figure 5.20 on page 361. You can photocopy these figures or download them from *www.mkp.com/cod2e.htm* to make it faster to show the additions.

5.6 [15] <§5.3> This question is similar to Exercise 5.5 except that we wish to add the instruction `jal` (jump and link), which is described in Chapter 3 on page 132. You may find it easier to modify the datapath in Figure 5.29 on page 372.

5.7 [8] <§5.3> This question is similar to Exercise 5.5 except that we wish to add the instruction `bne` (branch if not equal), which is described in Chapter 3.

5.8 [15] <§5.3> This question is similar to Exercise 5.5 except that we wish to add a variant of the `lw` (load word) instruction, which sums two registers to obtain the address of the data to be loaded (see Exercise 4.16) and uses the R-format.

5.9 [5] <§5.3> Explain why it is not possible to modify the single-cycle implementation to implement the swap instruction described in Exercise 4.40 without modifying the register file.

INTEL - 1012

5.10 [5] <§§5.3, 5.4> A friend is proposing that the control signal MemtoReg be eliminated. The multiplexor that has MemtoReg as an input will instead use the control signal MemRead. Will your friend's modification work? Consider both datapaths.

5.11 [10] <§5.3> This exercise is similar to Exercise 5.10 but more general. Determine whether any of the control signals (other than MemtoReg) in the single-cycle implementation can be eliminated and replaced by another existing control signal. Why or why not?

5.12 [15] <§5.3> Consider the following idea: Let's modify the instruction set architecture and remove the ability to specify an offset for memory access instructions. Specifically, all load-store instructions with nonzero offsets would become pseudoinstructions and would be implemented using two instructions. For example:

```
addi   $at, $t1, 104   # add the offset to a temporary
lw     $t0, $at        # new way of doing lw $t0, 104 ($t1)
```

What changes would you make to the single-cycle datapath and control if this simplified architecture were to be used?

5.13 [10] <§5.3> {Ex. 5.12} If the modifications described in Exercise 5.12 are implemented, there are some definite trade-offs with regard to performance. Specifically, the cycle time may be affected, and all load-store instructions with nonzero offsets would now require an extra `addi` instruction (a good compiler might find ways to reduce the need for extra `addi` instructions, but you can ignore this). If there are too many load-store instructions with nonzero offsets, it is likely that the modification would not improve performance. Assuming delays as specified on page 373, what is the highest percentage of load-store instructions with offsets that could be tolerated (i.e., that would still result in the modification having a positive impact on performance)?

5.14 [10] <§5.3> In estimating the performance of the single-cycle implementation, we assumed that only the major functional units had any delay (i.e., the delay of the multiplexors, control unit, PC access, sign extension unit, and wires was considered to be negligible). Assume that we change the delays specified on page 373 such that we use a different type of adder for simple addition:

- ALU: 2 ns
- adder for PC + 4: X ns
- adder for branch address computation: Y ns

a. What would the cycle time be if $X = 3$ and $Y = 3$?

b. What would the cycle time be if $X = 5$ and $Y = 5$?

c. What would the cycle time be if $X = 1$ and $Y = 8$?

5.15 [15] <§5.4> We wish to add the instruction `addi` (add immediate) to the multicycle datapath described in this chapter. This instruction is described in Chapter 3 on page 145. Add any necessary datapaths and control signals to the multicycle datapath of Figure 5.33 on page 383 and show the necessary modifications to the finite state machine of Figure 5.42 on page 396. You may find it helpful to examine the execution steps shown on pages 385 through 388 and consider the steps that will need to be performed to execute the new instruction. You can photocopy existing figures or download figures from *www.mkp.com/cod2e.htm* to make it easier to show your modifications. Try to find a solution that minimizes the number of clock cycles required for the new instruction. Please explicitly state how many cycles it takes to execute the new instruction on your modified datapath and finite state machine.

5.16 [5] <§§5.5, 5.8> {Ex. 5.15} Write the microcode sequences for the `addi` instruction. If you need to make any changes to the microinstruction format or field contents, indicate how the new format and fields will set the control outputs.

5.17 [15] <§5.4> This question is similar to Exercise 5.15 except that we wish to add the instruction `jal` (jump and link), which is described in Chapter 3.

5.18 [15] <§5.4> This question is similar to Exercise 5.15 except that we wish to add the `swap` instruction described in Exercise 4.40. Do not modify the register file. Since the instruction format for `swap` has not yet been defined, you are free to define it however you wish.

5.19 [15] <§5.4> This question is similar to Exercise 5.15 except that we wish to add a new instruction, `wai` (where am I), which puts the instruction's location (the value of the PC when the instruction was fetched) into a register specified by the rt field of the machine language instruction. Assume that the datapath hasn't changed and that, as usual, the clock cycle is too short to allow an ALU operation and a register file access in a single clock cycle if one of them is dependent on the results of the other.

5.20 [15] <§5.4> This question is similar to Exercise 5.15 except that we wish to add a new instruction, `jm` (jump memory). Its instruction format is similar to that of `load word` except that the rt field is not used because the data loaded from memory is put in the PC instead of the target register.

5.21 [20] <§5.4> This question is similar to Exercise 5.15 except that we wish to add support for four-operand arithmetic instructions such as `add3`, which adds three numbers together instead of two:

```
add3 $t5, $t6, $t7, $t8   # $t5 = $t6 + $t7 + $t8
```

INTEL - 1012

Assume that the instruction set is modified by introducing a new instruction format similar to the R-format except that bits [0–4] are used to specify the additional register (we still use rs, rt, and rd) and of course a new opcode is used. Your solution should not rely on adding additional read ports to the register file, nor should a new ALU be used.

5.22 [10] <§5.4> Show how the jump register instruction (described on pages 129 and A-65) can be implemented simply by making changes to the finite state machine of Figure 5.42 on page 396. (It may help you to remember that `$0` = `$zero` = 0.)

5.23 [15] <§5.4> Consider a change to the multiple-cycle implementation that alters the register file so that it has only one read port. Describe (via a diagram) any additional changes that will need to be made to the datapath in order to support this modification. Modify the finite state machine to indicate how the instructions will work, given your new datapath.

5.24 [15] <§§5.1–5.4> For this problem, use the gcc data from Figure 4.54 on page 309. Assume that there are three machines:

- M1: The multicycle datapath of Chapter 5 with a 500-MHz clock.
- M2: A machine like the multicycle datapath of Chapter 5, except that register updates are done in the same clock cycle as a memory read or ALU operation. Thus, in Figure 5.42 on page 396, states 6 and 7 and states 3 and 4 are combined. This machine has a 400-MHz clock, since the register update increases the length of the critical path.
- M3: A machine like M2, except that effective address calculations are done in the same clock cycle as a memory access. Thus, states 2, 3, and 4 can be combined, as can 2 and 5, as well as 6 and 7. This machine has a 250-MHz clock because of the long cycle created by combining address calculation and memory access.

Find out which machine is fastest. Are there instruction mixes that would make another machine faster, and if so, what are they?

5.25 [20] <§5.4> Your friends at C^3 (Creative Computer Corporation) have determined that the critical path that sets the clock cycle length of the multicycle datapath is memory access for loads and stores (*not* for instructions). This has caused their newest implementation of the MIPS 30000 to run at a clock rate of 500 MHz rather than the target clock rate of 750 MHz. However, Clara at C^3 has a solution. If all the cycles that access memory are broken into two clock cycles, then the machine can run at its target clock rate. Using the gcc mixes shown in Chapter 4 (Figure 4.54 on page 309), determine how much faster the machine with the two-cycle memory accesses is compared with the 500-MHz machine with single-cycle memory access. Assume that all jumps and branches take the same number of cycles and that the set instructions and arithmetic immediate instructions are implemented as R-type instructions.

5.26 [20] <§5.4> Suppose there were a MIPS instruction, called `bcp`, that copied a block of words from one address to another. Assume that this instruction requires that the starting address of the source block is in register `$t1` and the destination address is in `$t2`, and that the number of words to copy is in `$t3` (which is ≥ 0). Furthermore, assume that the values of these registers as well as register `$t4` can be destroyed in executing this instruction (so that the registers can be used as temporaries to execute the instruction).

Write the MIPS assembly language program to implement block copy. How many instructions will be executed to perform a 100-word block copy? Using the CPI of the instructions in the multicycle implementation, how many cycles are needed for the 100-word block copy?

5.27 [30] <§5.5> {Ex 5.26} Microcode has been used to add more powerful instructions to an instruction set; let's explore the potential benefits of this approach. Devise a strategy for implementing the `bcp` instruction described in Exercise 5.26 using the multicycle datapath and microcode. You will probably need to make some changes to the datapath in order to efficiently implement the `bcp` instruction. Provide a description of your proposed changes and describe how the `bcp` instruction will work. Are there any advantages that can be obtained by adding internal registers to the datapath to help support the `bcp` instruction? Estimate the improvement in performance that you can achieve by implementing the instruction in hardware (as opposed to the software solution you obtained in Exercise 5.26) and explain where the performance increase comes from.

5.28 [30] <§5.5> {Ex. 5.27} Using the strategy you developed in Exercise 5.27, modify the MIPS microinstruction format described in Figure 5.45 on page 403 and provide the complete microprogram for the `bcp` instruction. Describe in detail how you extended the microcode so as to support the creation of more complex control structures (such as a loop) within the microcode. Has support for the `bcp` instruction changed the size of the microcode? Will other instructions besides `bcp` be affected by the change in the microinstruction format?

5.29 [15] <§5.6> We wish to add the instruction `rfe` (return from exception) to the multicycle datapath described in this chapter. A primary task of the `rfe` instruction is to copy the contents of the EPC to the PC (the exception mechanisms require several additional capabilities that we will discuss in Chapter 7). Add any necessary datapaths and control signals to the multicycle datapath of Figure 5.48 on page 414 and show the necessary modifications to the finite state machine of Figures 5.49 and 5.50 on pages 415 and 416. You can photocopy the figures or download them from *www.mkp.com/cod2e.htm* to make it easier to show your modifications.

INTEL - 1012

5.30 [1 week] <§§5.2, 5.3> Using a hardware simulation language such as Verilog, implement a functional simulator for the single-cycle version. Build your simulator using an existing library of parts, if such a library is available. If the parts contain timing information, determine what the cycle time of your implementation will be.

5.31 [1 week] <§§5.2, 5.4, 5.5> Using a hardware simulation language such as Verilog, implement a functional simulator for the multicycle version of the design. Build your simulator using an existing library of parts, if such a library is available. If the parts contain timing information, determine what the cycle time of your implementation will be.

5.32 [2–3 months] <§§5.1–5.3> Using standard parts, build a machine that implements the single-cycle machine in this chapter.

5.33 [2–3 months] <§§5.1–5.8> Using standard parts, build a machine that implements the multicycle machine in this chapter.

5.34 [Discussion] <§§5.5, 5.8, 5.9> Hypothesis: If the first implementation of an architecture uses microprogramming, it affects the instruction set architecture. Why might this be true? Can you find an architecture that will probably always use microcode? Why? Which machines will never use microcode? Why? What control implementation do you think the architect had in mind when designing the instruction set architecture?

5.35 [Discussion] <§§5.5, 5.10> Wilkes invented microprogramming in large part to simplify construction of control. Since 1980, there has been an explosion of computer-aided design software whose goal is also to simplify construction of control. This has made control design much easier. Can you find evidence, based either on the tools or on real designs, that supports or refutes this hypothesis?

5.36 [Discussion] <§5.10> The MIPS instructions and the MIPS microinstructions have many similarities. What would make it difficult for a compiler to produce MIPS microcode rather than macrocode? What changes to the microarchitecture would make the microcode more useful for this application?

INTEL - 1012

8

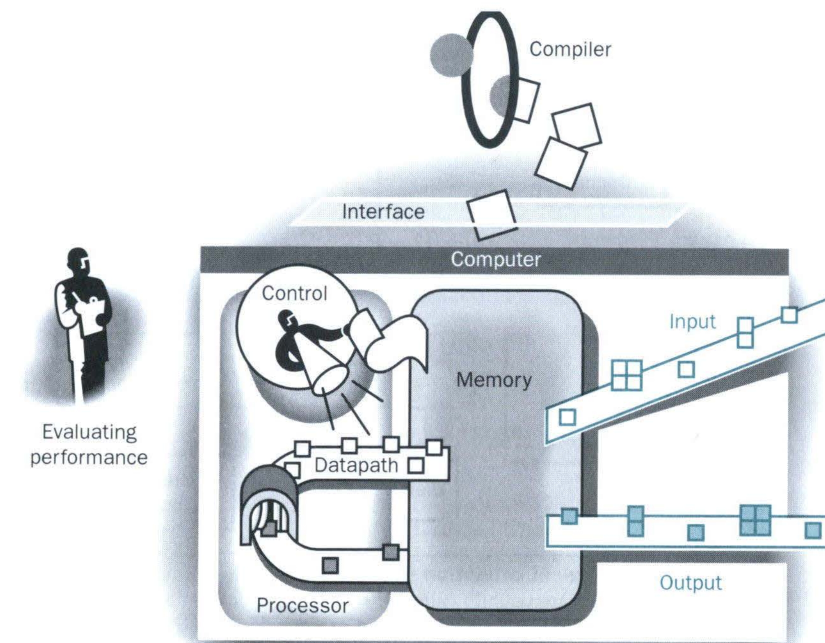
Interfacing Processors and Peripherals

I/O certainly has been lagging in the last decade.

Seymour Cray
Public lecture, 1976

8.1 **Introduction** 638
8.2 **I/O Performance Measures: Some Examples from Disk and File Systems** 641
8.3 **Types and Characteristics of I/O Devices** 644
8.4 **Buses: Connecting I/O Devices to Processor and Memory** 655
8.5 **Interfacing I/O Devices to the Memory, Processor, and Operating System** 673
8.6 **Designing an I/O System** 684
8.7 **Real Stuff: A Typical Desktop I/O System** 687
8.8 **Fallacies and Pitfalls** 688
8.9 **Concluding Remarks** 690
8.10 **Historical Perspective and Further Reading** 694
8.11 **Key Terms** 700
8.12 **Exercises** 700

The Five Classic Components of a Computer



8.1 Introduction

As in processors, many of the characteristics of input/output (I/O) systems are driven by technology. For example, the properties of disk drives affect how the disks should be connected to the processor, as well as how the operating system interacts with the disks. I/O systems, however, differ from processors in several important ways. Although processor designers often focus primarily on performance, designers of I/O systems must consider issues such as expandability and resilience in the face of failure as much as they consider performance. Second, performance in an I/O system is a more complex characteristic than for a processor. For example, with some devices we may care primarily about access latency, while with others throughput is crucial. Furthermore, performance depends on many aspects of the system: the device characteristics, the connection between the device and the rest of the system, the memory hierarchy, and the operating system. Figure 8.1 shows the structure of a system with its I/O. All of the components, from the individual I/O devices to the processor to the system software, will affect the performance of tasks that include I/O.

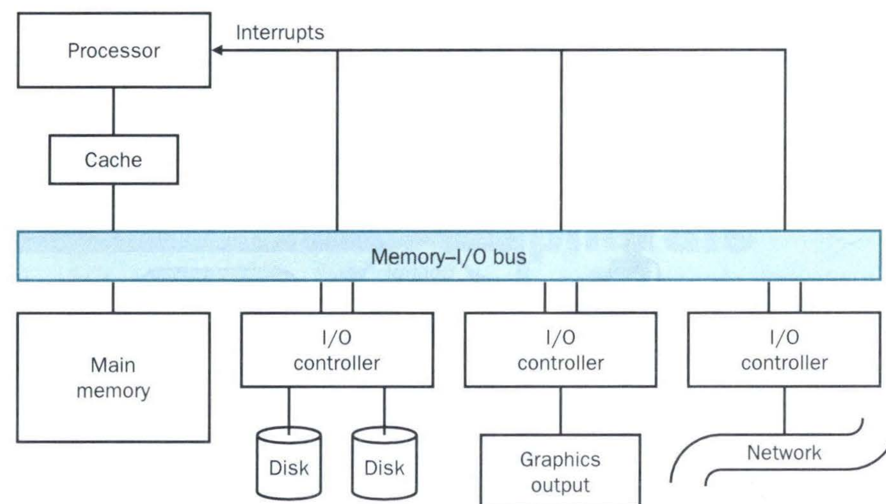


FIGURE 8.1 Typical collection of I/O devices. The connections between the I/O devices, processor, and memory are usually called *buses*. Communication among the devices and the processor use both protocols on the bus and interrupts, as we will see in this chapter.

The difficulties in assessing and designing I/O systems have often relegated I/O to second-class status. Research focuses on processor design; companies present performance using primarily processor-oriented measures; courses in every aspect of computing, from programming to computer architecture, often ignore I/O or give it scanty coverage; and textbooks leave the subject to near the end, making it easier for students and instructors to skip it!

This situation doesn't make sense: imagine how you'd like to use a computer without I/O! Furthermore, in an era when machines, from low-end PCs to the fastest mainframes, and even supercomputers, are being built from the same basic microprocessor technology, I/O capability is often one of the most distinctive features of the machines. Lastly, as the importance of networking and the information infrastructure grows, I/O will play an increasingly important role. Remember that machines interact with people through I/O.

If these concerns are still not convincing, our discussion of Amdahl's law in Chapter 2 should remind us that ignoring I/O is dangerous. A simple example demonstrates this.

Impact of I/O on System Performance

Example

Suppose we have a benchmark that executes in 100 seconds of elapsed time, where 90 seconds is CPU time and the rest is I/O time. If CPU time improves by 50% per year for the next five years but I/O time doesn't improve, how much faster will our program run at the end of five years?

Answer

We know that

$$\begin{aligned}\text{Elapsed time} &= \text{CPU time} + \text{I/O time}\\ 100 &= 90 + \text{I/O time}\\ \text{I/O time} &= 10 \text{ seconds}\end{aligned}$$

The new CPU times and the resulting elapsed times are computed in the following table:

After *n* years	CPU time	I/O time	Elapsed time	% I/O time
0	90 seconds	10 seconds	100 seconds	10%
1	$\frac{90}{1.5}$ = 60 seconds	10 seconds	70 seconds	14%
2	$\frac{60}{1.5}$ = 40 seconds	10 seconds	50 seconds	20%
3	$\frac{40}{1.5}$ = 27 seconds	10 seconds	37 seconds	27%
4	$\frac{27}{1.5}$ = 18 seconds	10 seconds	28 seconds	36%
5	$\frac{18}{1.5}$ = 12 seconds	10 seconds	22 seconds	45%

The improvement in CPU performance over five years is

$$\frac{90}{12} = 7.5$$

However, the improvement in elapsed time is only

$$\frac{100}{22} = 4.5$$

and the I/O time has increased from 10% to 45% of the elapsed time.

How we should assess I/O performance often depends on the application. In some environments, we may care primarily about system throughput. In these cases, I/O bandwidth will be most important. Even I/O bandwidth can be measured in two different ways:

1. How much data can we move through the system in a certain time?
2. How many I/O operations can we do per unit of time?

Which measurement is best may depend on the environment. For example, in many supercomputer applications, most I/O requests are for long streams of data, and transfer bandwidth is the important characteristic. In another environment, we may wish to process a large number of small, unrelated accesses to an I/O device. An example of such an environment might be a tax-processing office of the National Income Tax Service (NITS). NITS mostly cares about processing a large number of forms in a given time; each tax form is stored separately and is fairly small. A system oriented toward large file transfer may be satisfactory, but an I/O system that can support the simultaneous transfer of many small files may be cheaper and faster for processing millions of tax forms.

In other applications, we care primarily about response time, which you will recall is the total elapsed time to accomplish a particular task. If the I/O requests are extremely large, response time will depend heavily on bandwidth, but in many environments most accesses will be small, and the I/O system with the lowest latency per access will deliver the best response time. On single-user machines such as workstations and personal computers, response time is the key performance characteristic.

A large number of applications, especially in the vast commercial market for computing, require both high throughput and short response times. Examples include automatic teller machines (ATMs), airline reservation systems, order entry and inventory tracking systems, file servers, and machines for timesharing. In such environments, we care about both how long each task takes *and* how many tasks we can process in a second. The number of ATM requests you can process per hour doesn't matter if each one takes 15 minutes—you won't have any customers left! Similarly, if you can process each ATM request quickly but can only handle a small number of requests at once, you won't be able to support many ATMs, or the cost of the computer per ATM will be very high.

If I/O is truly important, how should we compare I/O systems? This is a complex question because I/O performance depends on many aspects of the system and different applications stress different aspects of the I/O system. Furthermore, a design can make complex trade-offs between response time and throughput, making it impossible to measure just one aspect in isolation. For example, response time is generally minimized by handling a request as early as possible, while greater throughput can be achieved if we try to handle related requests together. Accordingly, we may increase throughput on a disk by grouping requests that access locations that are close together. Such a policy will increase the response time for some requests, probably leading to a larger variation in response time. Although throughput will be higher, some benchmarks constrain the maximum response time to any request, making such optimizations potentially problematic.

Before discussing the aspects of I/O devices and how they are connected, let's look briefly at some performance measures for I/O systems.

8.2 I/O Performance Measures: Some Examples from Disk and File Systems

Assessment of an I/O system must take into account a variety of factors. Performance is one of these, and in this section, we give some examples of measurements proposed for determining the performance of disk systems. These benchmarks are affected by a variety of system features, including the disk technology, how disks are connected, the memory system, the processor, and the file system provided by the operating system. Overall, the state of

benchmarking on the I/O side of computer systems remains quite primitive compared with the extensive activity lately seen in benchmarking processor systems. Perhaps this situation will change as designers realize the importance of I/O and the inadequacy of our techniques to evaluate it.

Before we discuss these benchmarks, we need to address a confusing point about terminology and units. The performance of I/O systems depends on the rate at which the system transfers data. The transfer rate depends on the clock rate, which is typically given in MHz $=10^6$ cycles per second. The transfer rate is usually quoted in MB/sec. In I/O systems, MBs are measured using base 10 (i.e., 1 MB $= 10^6 =$ 1,000,000 bytes), unlike main memory where base 2 is used (i.e., 1 MB $= 2^{20} =$ 1,048,576). In addition to adding confusion, this difference introduces the need to convert between base 10 (1K = 1000) and base 2 (1K = 1024) because many I/O accesses are for data blocks that have a size that is a power of two. Rather than complicate all our examples by accurately converting one of the two measurements, we make note of this distinction and the fact that treating the two measures as if the units were identical introduces a small error. We illustrate this error in section 8.8.

Supercomputer I/O Benchmarks

Supercomputer I/O is dominated by accesses to large files on magnetic disks. Many supercomputer installations run batch jobs, each of which may last for hours. In these situations, I/O consists of one large read followed by writes to snapshot the state of the computation should the computer crash. As a result, supercomputer I/O in many cases consists more of output than input. The overriding supercomputer I/O measure is data throughput: the number of bytes per second that can be transferred between a supercomputer's main memory and disks during large transfers.

Transaction Processing I/O Benchmarks

Transaction processing (TP) applications involve both a response time requirement and a performance measurement based on throughput. Furthermore, most of the I/O accesses are small. Because of this, TP applications are chiefly concerned with *I/O rate*, measured as the number of disk accesses per second, as opposed to *data rate*, measured as bytes of data per second. TP applications generally involve changes to a large database, with the system meeting some response time requirements as well as gracefully handling certain types of failures. These applications are extremely critical and cost-sensitive. For example, banks normally use TP systems because they are concerned about a range of characteristics. These include making sure transactions aren't lost, handling transactions quickly, and minimizing the cost of processing each transaction. Although reliability in the face of failure is an absolute requirement in such systems, both response time and throughput are critical to building cost-effective systems.

A number of transaction processing benchmarks have been developed. The best-known set of benchmarks is a series developed by the Transaction Processing Council (TPC). The most recent versions of these benchmarks are TPC-C and TPC-D, both of which involve processing of queries against a database. TPC-C involves light- and medium-weight queries based on an order-entry environment, but also typical of the type of transactions needed in a reservation system or online banking system. TPC-D involves complex queries typical of decision support applications.

TPC-C is significantly more sophisticated than the earlier TPC-A and TPC-B benchmarks. It involves nine different types of database records, five different types of transactions, and a model of transaction requests meant to simulate real users generating transactions at terminals. The benchmark specification, including the reporting rules, is 128 pages long! Performance on TPC-C is measured in transactions per minute or second (TPM or TPS) and encompasses a complete system measurement including disk I/O, terminal I/O, and computation. An extensive description of the TPC organization and benchmarks is available via the TPC link at *www.mkp.com/books_catalog/cod/links.htm.*

File System I/O Benchmarks

File systems, which are stored on disks, have a different access pattern. For example, measurements of Unix file systems in an engineering environment have found that 80% of accesses are to files of less than 10 KB and that 90% of all file accesses are to data with sequential addresses on the disk. Furthermore, 67% of the accesses were reads, 27% were writes, and 6% were read-modify-write accesses, which read data, modify it, and then rewrite the same location. Such measurements have led to the creation of synthetic file system benchmarks. One of the most popular of such benchmarks has five phases, using 70 files with a total size of 200 KB:

- *MakeDir*: Constructs a directory subtree that is identical in structure to the given directory subtree
- *Copy*: Copies every file from the source subtree to the target subtree
- *ScanDir*: Recursively traverses a directory subtree and examines the status of every file in it
- *ReadAll*: Scans every byte of every file in a subtree once
- *Make*: Compiles and links all the files in a subtree

As we will see in section 8.6, the design of an I/O system involves knowing what the workload is.

INTEL - 1012

8.3 Types and Characteristics of I/O Devices

I/O devices are incredibly diverse. Three characteristics are useful in organizing this wide variety:

- *Behavior*: Input (read once), output (write only, cannot be read), or storage (can be reread and usually rewritten).
- *Partner*: Either a human or a machine is at the other end of the I/O device, either feeding data on input or reading data on output.
- *Data rate*: The peak rate at which data can be transferred between the I/O device and the main memory or processor. It is useful to know what maximum demand the device may generate.

For example, a keyboard is an *input* device used by a *human* with a *peak data rate* of about 10 bytes per second. Figure 8.2 shows some of the I/O devices connected to computers.

In Chapter 1, we briefly discussed four important and characteristic I/O devices: mice, graphics displays, disks, and networks. We use mice, disks, and networks as examples to illustrate how I/O devices interface to processors and memories, but before we do that it will be useful to discuss these devices in more detail than in Chapter 1.

Device	Behavior	Partner	Data rate (KB/sec)
Keyboard	input	human	0.01
Mouse	input	human	0.02
Voice input	input	human	0.02
Scanner	input	human	400.00
Voice output	output	human	0.60
Line printer	output	human	1.00
Laser printer	output	human	200.00
Graphics display	output	human	60,000.00
Modem	input or output	machine	2.00–8.00
Network/LAN	input or output	machine	500.00–6000.00
Floppy disk	storage	machine	100.00
Optical disk	storage	machine	1000.00
Magnetic tape	storage	machine	2000.00
Magnetic disk	storage	machine	2000.00–10,000.00

FIGURE 8.2 The diversity of I/O devices. I/O devices can be distinguished by whether they serve as input, output, or storage devices; their communication partner (people or other computers); and their peak communication rates. The data rates span six orders of magnitude. Note that a network can be an input or an output device, but cannot be used for storage. Disk sizes, as well as transfer rates for devices, are always quoted in base 10, so that 1 MB = 1,000,000 bytes, and 10 Mbit/sec = 10,000,000 bits/sec.

Mouse

The interface between a mouse and a system can take one of two forms: the mouse either generates a series of pulses when it is moved (using the LED and detector described in Chapter 1 to generate the pulses), or it increments and decrements counters. Figure 8.3 shows how the counters change when the mouse is moved and describes how the interface would operate if it generated pulses instead. The processor can periodically read these counters, or count up the pulses, and determine how far the mouse has moved since it was last examined. The system then moves the cursor on the screen appropriately. This motion appears smooth because the rate at which you can move the mouse is slow compared with the rate at which the processor can read the mouse status and move the cursor on the screen.

Most mice also include one or more buttons, and the system must be able to detect when a button is depressed. By monitoring the status of the button, the system can also differentiate between clicking the button and holding it down. Of course, the mapping between the counters and the button position and what happens on the screen is totally controlled by software. That's why, for example, the rate at which the mouse moves across the screen and the rate at which single and double clicks are recognized can usually be set by the user. Similarly, software interpretation of the mouse position means that the cursor doesn't jump completely off the screen when the mouse is moved a long distance in one direction. This method of having the system monitor the status of

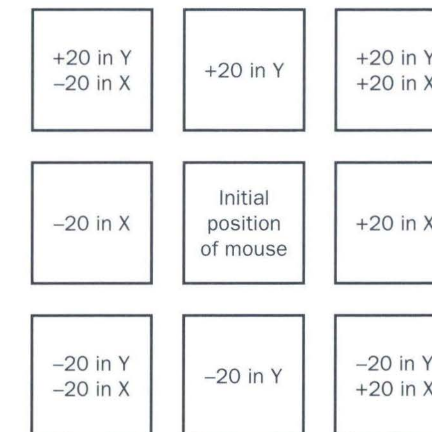


FIGURE 8.3 Moving the mouse in the horizontal direction or vertical direction causes the X or Y counter, respectively, to increment or decrement. Moving it along a diagonal causes both counters to change. Since the ball doesn't move when the mouse is not contacting the surface, it may be picked up and moved without changing the counters. When the mouse uses pulses to communicate its movement, there are four types of pulses: +X, –X, +Y, and –Y. Rather than generate a change in the counter value, the mouse generates the appropriate number of pulses on each of the four pulse signal lines. The value 20 is an arbitrary count that measures how far the mouse has moved.

INTEL - 1012

the mouse by reading signals from it is a common way to interface lower-performance devices to machines; it is called *polling*, and we'll revisit it in section 8.5.

Magnetic Disks

As mentioned in Chapter 1, there are two major types of magnetic disks: floppy disks and hard disks. Both types of disks rely on a rotating platter coated with a magnetic surface and use a moveable read/write head to access the disk. Disk storage is *nonvolatile*, meaning that the data remains even when power is removed. Because the platters in a hard disk are metal (or, recently, glass), they have several significant advantages over floppy disks:

- The hard disk can be larger because it is rigid.
- The hard disk has higher density because it can be controlled more precisely.
- The hard disk has a higher data rate because it spins faster.
- Hard disks can incorporate more than one platter.

For the rest of this section, we will focus on hard disks, and we use the term *magnetic disk* to mean hard disk.

A magnetic disk consists of a collection of platters (1–15), each of which has two recordable disk surfaces, as shown in Figure 8.4. The stack of platters is rotated at 3600 to 7200 RPM and has a diameter from just over an inch to just over 8 inches. Each disk surface is divided into concentric circles, called *tracks*. There are typically 1000 to 5000 tracks per surface. Each track is in turn divided into *sectors* that contain the information; each track may have 64 to 200 sectors. Originally, the sector was the smallest unit that could be read or written. With the introduction of Logical Block Access (LBA), disk drives became addressed by blocks, and a block became the minimum accessible unit. In 1997, blocks were typically 512 bytes in size. The sequence recorded on the magnetic media is a sector number, a gap, the information for that sector including error correction code (see Appendix B, page B-34), a gap, the sector number of the next sector, and so on. Originally, all tracks had the same number of sectors and hence the same number of bits, but with the introduction of Zone Bit Recording, (ZBR) in the early 1990s, disk drives changed to a varying number of sectors (and hence bits) per track, instead keeping the spacing between bits constant. ZBR increases the number of bits on the outer tracks and thus increases the drive capacity.

As we saw in Chapter 1, to read and write information the read/write heads must be moved so that they are over the correct location. The disk heads for each surface are connected together and move in conjunction, so that every head is over the same track of every surface. The term *cylinder* is used to refer to all the tracks under the heads at a given point on all surfaces.

To access data, the operating system must direct the disk through a three-stage process. The first step is to position the head over the proper track. This operation is called a *seek*, and the time to move the head to the desired track is called the *seek time*.

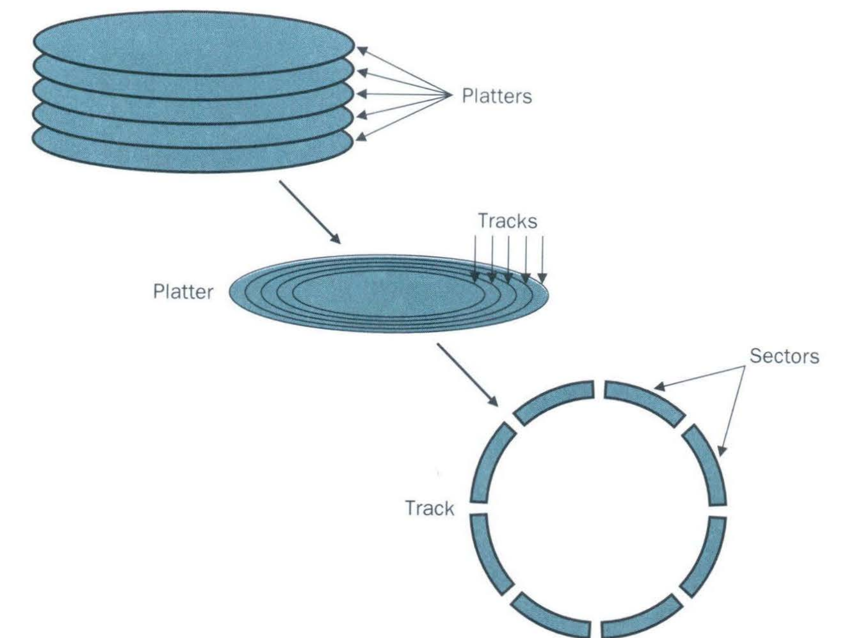


FIGURE 8.4 Disks are organized into platters, tracks, and sectors. Both sides of a platter are coated so that information can be stored on both surfaces. Floppy disks have the same organization, but consist of only one platter.

Disk manufacturers report minimum seek time, maximum seek time, and average seek time in their manuals. The first two are easy to measure, but the average is open to wide interpretation because it depends on the seek distance. The industry has decided to calculate average seek time as the sum of the time for all possible seeks divided by the number of possible seeks. Average seek times are usually advertised as 8 ms to 20 ms, but, depending on the application and scheduling of disk requests, the actual average seek time may be only 25% to 33% of the advertised number, because of locality of disk references. This locality arises both because of successive access to the same file and because the operating system tries to schedule such access together.

Once the head has reached the correct track, we must wait for the desired sector to rotate under the read/write head. This time is called the *rotational latency* or *rotational delay*. The average latency to the desired information is halfway around the disk. Because the disks rotate at 3600 RPM to 7200 RPM, the average rotational latency is between

$$\text{Average rotational latency} = \frac{0.5 \text{ rotation}}{3600 \text{ RPM}} = \frac{0.5 \text{ rotation}}{3600 \text{ RPM}/\left(60 \frac{\text{seconds}}{\text{minute}}\right)}$$

$$= 0.0083 \text{ seconds} = 8.3 \text{ ms}$$

INTEL - 1012

and

$$\text{Average rotational latency} = \frac{0.5 \text{ rotation}}{7200 \text{ RPM}} = \frac{0.5 \text{ rotation}}{7200 \text{ RPM}/\left(60 \frac{\text{seconds}}{\text{minute}}\right)}$$

$$= 0.0042 \text{ seconds} = 4.2 \text{ ms}$$

Smaller diameter disks are attractive because they can spin at higher rates without excessive power consumption, thereby reducing rotational latency.

The last component of a disk access, *transfer time,* is the time to transfer a block of bits. The transfer time is a function of the sector size, the rotation speed, and the recording density of a track. Transfer rates in 1997 are between 2 and 15 MB/sec. The one complication is that most midrange and high-end disks have a built-in cache that stores sectors as they are passed over; transfer rates from the cache are typically higher, and may be up to 40 MB/sec in 1997. Today, most disk transfers are multiple sectors in length.

The detailed control of the disk and the transfer between the disk and the memory is usually handled by a *disk controller.* The controller adds the final component of disk access time, *controller time,* which is the overhead the controller imposes in performing an I/O access. The average time to perform an I/O operation will consist of these four times plus any wait time incurred because other processes are using the disk.

Disk Read Time

Example

What is the average time to read or write a 512-byte sector for a typical disk rotating at 5400 RPM? The advertised average seek time is 12 ms, the transfer rate is 5 MB/sec, and the controller overhead is 2 ms. Assume that the disk is idle so that there is no waiting time.

Answer

Average disk access time is equal to average seek time + average rotational delay + transfer time + controller overhead. Using the advertised average seek time, the answer is

$$12 \text{ ms} + 5.6 \text{ ms} + \frac{0.5 \text{ KB}}{5 \text{ MB/sec}} + 2 \text{ ms} = 12 + 5.6 + 0.1 + 2 = 19.7 \text{ ms}$$

If the measured average seek time is 25% of the advertised average time, the answer is

$$3 \text{ ms} + 5.6 \text{ ms} + 0.1 \text{ ms} + 2 \text{ ms} = 10.7 \text{ ms}$$

Notice that when we consider average measured seek time, as opposed to average advertised seek time, the rotational latency can be the largest component of the access time.

Disk densities have continued to increase for more than 40 years. The impact of this compounded improvement in density and the reduction in physical size of a disk drive has been amazing, as Figure 8.5 shows. The aims of different disk designers have led to a wide variety of drives being available at any particular time. Figure 8.6 shows the characteristics of three different

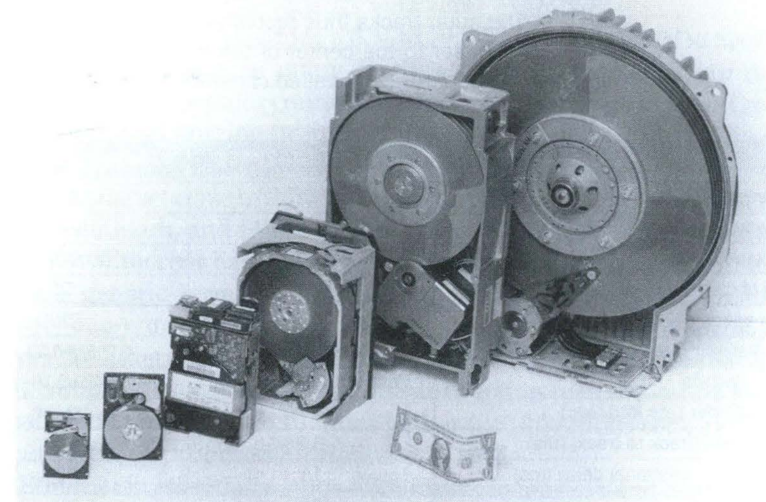


FIGURE 8.5 Six magnetic disks, varying in diameter from 14 inches down to 1.8 inches. These disks were introduced over more than a decade ago and hence are not intended to be representative of the best 1998 capacity of disks of these diameters. This photograph does, however, accurately portray their relative physical sizes. The widest disk is the DEC R81, containing four 14-inch diameter platters and storing 456 MB. It was manufactured in 1985. The 8-inch diameter disk comes from Fujitsu, and this 1984 disk stores 130 MB on six platters. The Micropolis RD53 has five 5.25-inch platters and stores 85 MB. The IBM 0361 also has five platters, but these are just 3.5 inches in diameter. This 1988 disk holds 320 MB. In 1997, the most dense 3.5-inch disk has 10 platters and holds 9.1 GB in the same space, yielding an increase in density of about 30 times! The Conner CP 2045 has two 2.5-inch platters containing 40 MB, and was made in 1990. The smallest disk in this photograph is the Integral 1820. This single 1.8-inch platter contains 20 MB and was made in 1992. Photo by Peg Skorpinski.

INTEL - 1012

magnetic disks from a single manufacturer. Large-diameter drives have many more megabytes to amortize the cost of electronics, so the traditional wisdom was that they had the lowest cost per megabyte. But this advantage is offset for the small drives by the much higher sales volume, which lowers manufacturing costs: in 1997, disks cost between $0.10 and $0.20 per megabyte, almost independent of width. The smaller drives also have advantages in power and volume per byte, as Figure 8.6 shows.

Elaboration: Many recent disks have included caches directly in the disk. Such caches allow for fast access to data that was recently read between transfers requested by the CPU. Of course, such capabilities complicate the measurement of disk performance and increase the importance of workload choice. The 5.25-inch Seagate drive shown in Figure 8.6 comes with an integrated cache.

Elaboration: Each track has the same number of bits, and the outer tracks are longer. The outer tracks thus record information at a lower density per inch of track than do tracks closer to the center of the disk. Recording more sectors on the outer tracks than on the inner tracks, called *constant bit density*, is becoming more widespread with the advent of intelligent interface standards such as SCSI (see section 8.4). The rate at which an inch of track moves under the head varies: it is faster on the outer tracks. Accordingly, if the number of bits per inch is constant, the rate at which bits must be read or written varies, and the electronics must accommodate this factor when constant bit density is used.

Characteristics	Seagate ST423451	Seagate ST19171	Seagate ST92255
Disk diameter (inches)	5.25	3.50	2.50
Formatted data capacity (MB)	23,200	9100	2250
MTBF (hours)	500,000	1,000,000	300,000
Number of disk surfaces	28	20	10
Rotation speed (RPM)	5400	7200	4500
Internal transfer rate (Mbits/sec)	86–124	80–124	up to 60.8
External interface	Fast SCSI-2 (8–16 bit)	Fast SCSI-2 (8–16 bit)	Fast ATA
External transfer rate (MB/sec)	20–40	20–40	up to 16.6
Minimum seek (track to track) (ms)	0.9	0.6	4
Average seek + rotational delay (ms)	11	9	14
Power/box (watts)	26	13	2.6
MB/watt	892	700	865
Volume (cu. in.)	322	37	8
MB/cu. in.	72	246	273

FIGURE 8.6 Characteristics of three magnetic disks by a single manufacturer. These disks represent the maximum density of the 1997 Seagate product family at each size. The disks shown here either interface to SCSI, a standard I/O bus that we discuss on page 672, or ATA, a standard disk interface for PCs. Compared to the disks shown in the table that appeared in the first edition of this book in 1994, the disks shown above have 25–40 times the MB/watt and 90–450 times the MB/cu. ft.! MTBF stands for mean time before failures—a standard measurement of reliability. The two larger disks contain sector caches that store the contents of sectors as they are passed over. The internal transfer rate is that rate at which bits are read from the disk surface, while the external transfer rate includes that rate at which a sector in the cache that is requested can be transferred. See the link to Seagate at *www.mkp.com/books_catalog/cod/links.htm* for more information on these drives, as well as some information on modern disk technology.

Networks

Networks are the major medium used to communicate between computers. Key characteristics of typical networks include the following:

- *Distance:* 0.01 to 10,000 kilometers
- *Speed:* 0.001 MB/sec to 100 MB/sec
- *Topology:* Bus, ring, star, tree
- *Shared lines:* None (point-to-point) or shared (multidrop)

We'll illustrate these characteristics with three examples.

The RS232 standard provides a 0.3- to 19.2-Kbit/sec *terminal network.* A central computer connects to many terminals over slow but cheap dedicated wires. These point-to-point connections form a star from the central computer, with each terminal ranging from 10 to 100 meters in distance from the computer.

The *local area network* (LAN) is what is commonly meant today when people mention a network, and Ethernet is what most people mean when they mention a LAN. (Ethernet has in fact become such a common term that it is often used as a generic term for LAN.) The basic Ethernet is essentially a 10-Mbit/sec, one-wire bus that has no central control. Messages, or *packets,* are sent over the Ethernet in blocks that vary from 64 bytes to 1518 bytes. Recently, several companies have developed a faster version (usually called Fast Ethernet) that offers rates that are 10 times higher (i.e., 100 Mbit/sec), and a Gigabit Ethernet has been proposed for delivery in 1998.

An Ethernet is essentially a bus with multiple masters and a scheme for determining who gets bus control; we'll discuss how the distributed control is implemented in the exercises. Because the Ethernet is a bus, only one sender can be transmitting at any time; this limits the bandwidth. In practice, this is not usually a problem because the utilization is fairly low. Of course, some LANs become overloaded through poor capacity planning, and response time and throughput can degrade rapidly at higher utilization.

One way in which the limits of the original bus-oriented Ethernet have been overcome is through switched networks. A *switched network* is one in which switches are introduced to reduce the number of hosts per Ethernet segment. In the limit, there is only one host per segment and that host is directly connected to a switch. Switched networks are common in long-haul networks, the next

INTEL - 1012

topic, but such networks have recently been popular in local area applications as the use of higher-performance machines and multimedia data has put significant strains on shared Ethernets.

Long-haul networks cover distances of 10 to 10,000 kilometers. The first and most famous long-haul network was the ARPANET (named after its funding agency, the Advanced Research Projects Agency of the U.S. government). It transferred data at 56 Kbits/sec and used point-to-point dedicated lines leased from telephone companies. The host computer talked to an *interface message processor* (IMP), which communicated over the telephone lines. The IMP took information and broke it into 1-Kbit packets, which could take separate paths to the destination node. At each hop, a packet was stored (for recovery in case of failure) and then forwarded to the proper IMP according to the address in the packet. The destination IMP reassembled the packets into a message and then gave it to the host. Most networks today use this *packet-switched* approach, in which packets are individually routed from source to destination.

The ARPANET was the precursor of the Internet. The key to interconnecting different networks was standardizing on a single protocol family, TCP/IP (Transmission Control Protocol/Internet Protocol). The IP portion of the protocol provides for addressing between two hosts on the Internet, but does not guarantee reliable delivery. TCP provides a protocol that can guarantee that all packets are received and that the packets have no transmission errors. These two protocols work together to form a *protocol stack*, where TCP packets are encapsulated in IP packets. The standardization of the TCP/IP packet format is what allows the different hosts and network to communicate.

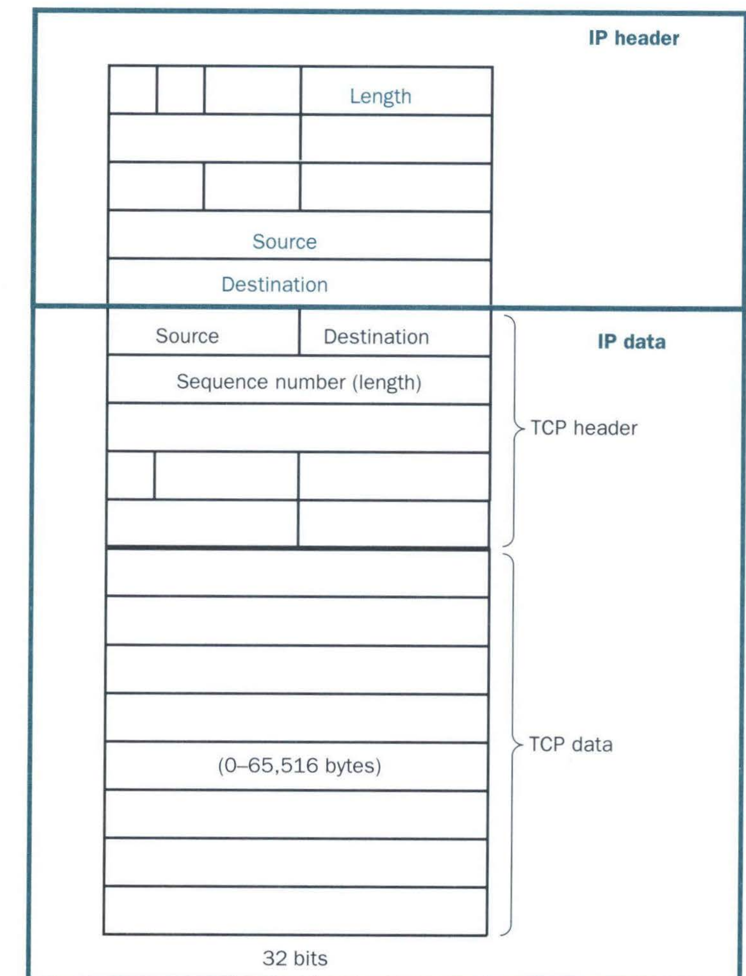
The bandwidths of networks are probably growing faster than the bandwidth of any other type of device at present. High-speed networks using copper and coaxial cable offer 100 Mbit/sec bandwidths, while optical fiber offers bandwidths up to 1 Gbit/sec. In the future, it appears that Internet-like technologies may be extended up to the 1-Gbit/sec range. These super high-speed networks are likely to be switched rather than using shared links.

Another leader among the emerging network technologies is ATM (Asynchronous Transfer Method). ATM is a scalable network technology (from 155 Mbits/sec to 2.5 Gbits/sec) that originated in long-haul networks switching both voice and data. It is already being deployed in backbone switching applications and, together with Fast Ethernet approaches, is a contender for future desktop connectivity.

The challenge in putting these networks into use lies primarily in building systems that can efficiently interface to these media and sustain these bandwidths between two programs that want to communicate. Meeting this challenge requires that all the pieces of the I/O system, from the operating system to the memory system to the bus to the device interface, be able to accommodate these bandwidths. This is truly a top-to-bottom systems challenge.

Hardware Software Interface

To allow communication across multiple networks with different characteristics, TCP/IP defines a standard packet format. An IP packet, which contains Internet addressing information, encapsulates a TCP packet that contains both address information interpreted by the host and the data being communicated. The IP header specifies that the length of the IP data (from 1 to 65,536 bytes). Since the TCP header uses 20 bytes of the IP data, the maximum size of the TCP data packet is 65, 536 – 20 = 65,516 bytes.



INTEL - 1012

To see the importance of looking at performance from top to bottom, including both hardware and software, consider the following example.

Performance of Two Networks

Example

Consider the following measurements made on a pair of SPARCstation 10s running Solaris 2.3, connected to two different types of networks, and using TCP/IP for communication:

Characteristic	Ethernet	ATM
Bandwidth from node to network	1.125 MB/sec	10 MB/sec
Interconnect latency	15 µs	50 µs
HW latency to/from network	6 µs	6 µs
SW overhead sending to network	200 µs	207 µs
SW overhead receiving from network	241 µs	360 µs

Find the host-to-host latency for a 250-byte message using each network.

Answer

We can estimate the time required as the sum of the fixed latencies plus the time to transmit the message. The time to transmit the message is simply the message length divided by the bandwidth of the network.

The transmission times are

$$\text{Transmission time}_{\text{Ethernet}} = \frac{250 \text{ bytes}}{1.125 \times 10^6 \text{ bytes/sec}} = 222\ \mu\text{s}$$

$$\text{Transmission time}_{\text{ATM}} = \frac{250 \text{ bytes}}{10 \times 10^6 \text{ bytes/sec}} = 25\ \mu\text{s}$$

So the transmission time for the ATM network is about a factor of nine lower.

The total latency to send and receive the packet is the sum of the transmission time and the hardware and software overheads:

$$\text{Total time}_{\text{Ethernet}} = 15 + 6 + 200 + 241 + 222 = 684\ \mu\text{s}$$

$$\text{Total time}_{\text{ATM}} = 50 + 6 + 207 + 360 + 25 = 648\ \mu\text{s}$$

The end-to-end latency of the Ethrnet is only about 1.06 times higher, even though the transmission time is almost 9 times higher!

8.4 Buses: Connecting I/O Devices to Processor and Memory

In a computer system, the various subsystems must have interfaces to one another. For example, the memory and processor need to communicate, as do the processor and the I/O devices. This is commonly done with a *bus*. A bus is a shared communication link, which uses one set of wires to connect multiple subsystems. The two major advantages of the bus organization are versatility and low cost. By defining a single connection scheme, new devices can easily be added, and peripherals can even be moved between computer systems that use the same kind of bus. Furthermore, buses are cost-effective because a single set of wires is shared in multiple ways.

The major disadvantage of a bus is that it creates a communication bottleneck, possibly limiting the maximum I/O throughput. When I/O must pass through a single bus, the bandwidth of that bus limits the maximum I/O throughput. In commercial systems, where I/O is very frequent, and in supercomputers, where the I/O rates must be very high because the processor performance is high, designing a bus system capable of meeting the demands of the processor as well as connecting large numbers of I/O devices to the machine presents a major challenge.

One reason bus design is so difficult is that the maximum bus speed is largely limited by physical factors: the length of the bus and the number of devices. These physical limits prevent us from running the bus arbitrarily fast. Within these limits, there are a variety of techniques we can use to increase the performance of the bus; however, these techniques may adversely affect other performance metrics. For example, to obtain fast response time for I/O operations, we must minimize the time to perform a bus access by streamlining the communication path. On the other hand, to sustain high I/O data rates, we must maximize the bus bandwidth. The bus bandwidth can be increased by using more buffering and by communicating larger blocks of data, both of which increase the delay to complete the bus access! Clearly, these two goals, fast bus accesses and high bandwidth, can lead to conflicting design requirements. Finally, the need to support a range of devices with widely varying latencies and data transfer rates also makes bus design challenging.

A bus generally contains a set of control lines and a set of data lines. The control lines are used to signal requests and acknowledgments, and to indicate what type of information is on the data lines. The data lines of the bus carry information between the source and the destination. This information may consist of data, complex commands, or addresses. For example, if a disk wants to write some data into memory from a disk sector, the data lines will be used to indicate the address in memory in which to place the data as well as to carry

INTEL - 1012