Chapter 6

Processor Organization and Performance

Objectives

- To introduce processor design issues;
- To describe flow control mechanisms used in RISC and CISC processors;
- To present details on microprogrammed control;
- To discuss performance issues.

This chapter looks at processor design and performance issues. We start our discussion with the number of addresses used in processor instructions. This is an important design characteristic that influences the instruction set design. This section also describes the load/store architecture used by RISC and vector processors.

Flow control deals with branching, procedure calls, and interrupts. It is an important aspect that affects the performance of the overall system. In Section 6.3, we discuss the general principles used to efficiently implement branching and procedure invocation mechanisms. We discuss the interrupt mechanism in Chapter 20. Instruction set design issues are discussed in Section 6.4.

The next section focuses on how the instructions are executed in hardware. A datapath is used to execute instructions. We have already presented an example datapath in Chapter 1. To execute instructions, we need to supply appropriate control signals for the datapath. We can generate these control signals in one of two basic ways: we can design our hardware to directly generate the control signals, or use what is known as the microprogram to issue necessary control signals to the underlying hardware. Typically, RISC processors use the direct hardware execution method, as their instructions are simple. The CISC processors, on the other hand,

depend on microprogrammed control in order simplify the hardware. These details are covered in Section 6.5.

Section 6.6 discusses how the performance of the CPU can be quantified and measured. This section also describes some sample performance benchmarks from the SPEC consortium. The chapter concludes with a summary.

6.1 Introduction

Processor designs can be broadly divided into three types: RISC, CISC, and vector processors. We briefly mentioned CISC and RISC processors in Chapter 1. We give a detailed discussion of the RISC and CISC processors in Chapter 14. Vector processors exploit pipelining to the fullest extent possible. We present details on vector processors in Chapter 8.

This chapter deals with three processor-related topics: instruction set design issues, microprogrammed control, and performance issues. In Chapter 1, we introduced the two main components of the processor: the datapath and control. Although the processors designed in the 1970s consisted of these two components, current processors have many more onchip entities such as caches and pipelined execution units. These features are discussed in other chapters of this book. In this chapter, we mainly focus on the datapath and control. As well, we look at the instruction set architecture and performance issues.

One of the characteristics that influences the ISA is the number of addresses used in the instructions. Since typical operations require two operands, we need three addresses: two source addresses to specify the two input operands and a destination address to indicate where the result should be stored. Most processors specify three addresses. We can reduce the number of addresses to two by using one address to specify a source address as well as the destination address. The Pentium uses two-address format instructions. It is also possible to have instructions that use one or even zero address. The one-address machines are called accumulator machines and the zero-address machines are called stack machines. The relative advantages and drawbacks of these schemes are discussed in Section 6.2.

RISC processors tend to use a special architecture known as the load/store architecture. In this architecture, special load and store instructions are used to move data between the processor's internal registers and memory. All other instructions require the necessary operands to be present in the registers. Vector processors originally used the load/store architecture. We discuss vector processors in Chapter 8. The load/store architecture is described in Section 6.2.6.

Instruction set design involves several other issues. The addressing mode is another important aspect that specifies where the operands are located. CISC processors typically allow a variety of addressing modes, whereas RISC processors support only a couple of addressing modes. The addressing modes and number of addresses directly influence the instruction format. CISC processors use variable-length instructions whereas the RISC processors use fixed-length instructions. The difference is mainly due to the fact that CISC processors use from simple to complex addressing modes. These and other issues such as the instruction and operand types are discussed in Section 6.4.

As mentioned in Section 1.4, the datapath provides the basic hardware to execute instructions. We have given an example datapath on page 16. This datapath uses three internal buses. The performance of the processor depends on the number of internal buses used. To execute instructions on the datapath, we have to provide appropriate control signals. These control signals can be generated by implementing a finite state machine in hardware. Hardware implementation is used for simple, well-structured instructions. RISC processors take this approach. CISC processors use a software approach that uses a microprogram for this purpose. Processors like the Pentium use this approach. We discuss microprogrammed control in detail in Section 6.5.

Performance quantification is very important for both designers and users. Designers need a way to compare performance of various designs in order to select an appropriate design. Users need to know the performance in order to buy the best system that meets their needs. The processor is a critical component that influences overall system performance. We discuss processor performance metrics and standards in Section 6.6.

6.2 Number of Addresses

One of the characteristics of the ISA that shapes the architecture is the number of addresses used in an instruction. Most operations can be divided into binary or unary operations. Binary operations such as addition and multiplication require two input operands whereas the unary operations such as the logical NOT need only a single operand. Most operations produce a single result. There are exceptions, however. For example, the division operation produces two outputs: a quotient and a remainder. Since most operations are binary, we need a total of three addresses: two addresses to specify the two input operands and one to specify where the result should go.

Most recent processors use three addresses. However, it is possible to design systems with two, one, or even zero addresses. In the rest of this section, we give a brief description of these four types of machines. In Section 6.2.5, we discuss their advantages and disadvantages.

6.2.1 Three-Address Machines

In three-address machines, instructions carry all three addresses explicitly. The RISC processors we discuss in Chapters 14 and 15 use three addresses. Table 6.1 gives some sample instructions of a three-address machine.

In these machines, the C statement

```
A = B + C * D - E + F + A
```

is converted to the following code:

```
mult  T,C,D    ; T = C*D
add   T,T,B    ; T = B + C*D
sub   T,T,E    ; T = B + C*D - E
add   T,T,F    ; T = B + C*D - E + F
add   A,T,A    ; A = B + C*D - E + F + A
```

Table 6.1 Sample three-address machine instructions

Instruction	Semantics
`add dest,src1,src2`	Adds the two values at `src1` and `src2` and stores the result in `dest` M(dest) = [src1] + [src2]
`sub dest,src1,src2`	Subtracts the second source operand at `src2` from the first at `src1` and stores the result in `dest` M(dest) = [src1] − [src2]
`mult dest,src1,src2`	Multiplies the two values at `src1` and `src2` and stores the result in `dest` M(dest) = [src1] * [src2]

We use the notation that each variable represents a memory address that stores the value associated with that variable. This translation from symbol name to the memory address is done by using a symbol table. We discuss the function of the symbol table in Section 9.3.3 (see page 330).

As you can see from this code, there is one instruction for each arithmetic operation. Also notice that all instructions, barring the first one, use an address twice. In the middle three instructions, it is the temporary `T` and in the last one, it is `A`. This is the motivation for using two addresses, as we show next.

6.2.2 Two-Address Machines

In two-address machines, one address doubles as a source and destination. Usually, we use `dest` to indicate that the address is used for destination. But you should note that this address also supplies one of the source operands. The Pentium is an example processor that uses two addresses. We discuss Pentium processor details in the next chapter. Table 6.2 gives some sample instructions of a two-address machine.

On these machines, the C statement

```
A = B + C * D - E + F + A
```

is converted to the following code:

```
load  T,C    ; T = C
mult  T,D    ; T = C*D
add   T,B    ; T = B + C*D
sub   T,E    ; T = B + C*D - E
add   T,F    ; T = B + C*D - E + F
add   A,T    ; A = B + C*D - E + F + A
```

Table 6.2 Sample two-address machine instructions

Instruction	Semantics
`load dest,src`	Copies the value at `src` to `dest` M(dest) = [src]
`add dest,src`	Adds the two values at `src` and `dest` and stores the result in `dest` M(dest) = [dest] + [src]
`sub dest,src`	Subtracts the second source operand at `src` from the first at `dest` and stores the result in `dest` M(dest) = [dest] − [src]
`mult dest,src`	Multiplies the two values at `src` and `dest` and stores the result in `dest` M(dest) = [dest] * [src]

Since we use only two addresses, we use a load instruction to first copy the `C` value into a temporary represented by `T`. If you look at these six instructions, you will notice that the operand `T` is common. If we make this our default, then we don't need even two addresses: we can get away with just one address.

6.2.3 One-Address Machines

In the early machines, when memory was expensive and slow, a special set of registers was used to provide an input operand as well as to receive the result from the ALU. Because of this, these registers are called the *accumulators*. In most machines, there is just a single accumulator register. This kind of design, called *accumulator machines*, makes sense if memory is expensive.

In accumulator machines, most operations are performed on the contents of the accumulator and the operand supplied by the instruction. Thus, instructions for these machines need to specify only the address of a single operand. There is no need to store the result in memory: this reduces the need for larger memory as well as speeds up the computation by reducing the number of memory accesses. A few sample accumulator machine instructions are shown in Table 6.3.

In these machines, the C statement

```
A = B + C * D - E + F + A
```

is converted to the following code:

Table 6.3 Sample accumulator machine instructions

Instruction	Semantics
`load addr`	Copies the value at address `addr` into the accumulator accumulator = [`addr`]
`store addr`	Stores the value in the accumulator at the memory address `addr` M(`addr`) = accumulator
`add addr`	Adds the contents of the accumulator and value at address `addr` accumulator = accumulator + [`addr`]
`sub addr`	Subtracts the value at memory address `addr` from the contents of the accumulator accumulator = accumulator − [`addr`]
`mult addr`	Multiplies the contents of the accumulator and value at address `addr` accumulator = accumulator * [`addr`]

```
load    C    ; load C into the accumulator
mult    D    ; accumulator = C*D
add     B    ; accumulator = C*D+B
sub     E    ; accumulator = C*D+B-E
add     F    ; accumulator = C*D+B-E+F
add     A    ; accumulator = C*D+B-E+F+A
store   A    ; store the accumulator contents in A
```

6.2.4 Zero-Address Machines

In zero-address machines, locations of both operands are assumed to be at a default location. These machines use the stack as the source of the input operands and the result goes back into the stack. Stack is a LIFO (last-in-first-out) data structure that all processors support, whether or not they are zero-address machines. As the name implies, the last item placed on the stack is the first item to be taken out of the stack. A good analogy is the stack of trays you find in a cafeteria. We discuss the stack later in this book (see Section 10.1 on page 388).

All operations on this type of machine assume that the required input operands are the top two values on the stack. The result of the operation is placed on top of the stack. Table 6.4 gives some sample instructions for the stack machines.

Notice that the first two instructions are not zero-address instructions. These two are special instructions that use a single address and are used to move data between memory and stack.

Table 6.4 Sample stack machine instructions

Instruction	Semantics
`push addr`	Places the value at address `addr` on top of the stack push([`addr`])
`pop addr`	Stores the top value on the stack at memory address `addr` M(`addr`) = pop
`add`	Adds the top two values on the stack and pushes the result onto the stack push(pop + pop)
`sub`	Subtracts the second top value from the top value of the stack and pushes the result onto the stack push(pop − pop)
`mult`	Multiplies the top two values in the stack and pushes the result onto the stack push(pop * pop)

All other instructions use the zero-address format. Let's see how the stack machine translates the arithmetic expression we have seen in the previous subsections. In these machines, the C statement

```
A = B + C * D - E + F + A
```

is converted to the following code:

```
push  E     ; <E>
push  C     ; <C, E>
push  D     ; <D, C, E>
mult        ; <C*D, E>
push  B     ; <B, C*D, E>
add         ; <B+C*D, E>
sub         ; <B+C*D-E>
push  F     ; <F, B+D*C-E>
add         ; <F+B+D*C-E>
push  A     ; <A, F+B+D*C-E>
add         ; <A+F+B+D*C-E>
pop   A     ; < >
```

On the right, we show the state of the stack after executing each instruction. The top element of the stack is shown on the left. Notice that we pushed `E` early because we need to subtract it from (`B+C*D`).

Stack machines are implemented by making the top portion of the stack internal to the processor. This is referred to as the *stack depth*. The rest of the stack is placed in memory. Thus, to access the top values that are within the stack depth, we do not have to access the memory. Obviously, we get better performance by increasing the stack depth. Examples of stack-oriented machines include the earlier Burroughs B5500 system and the HP3000 from Hewlett–Packard. Most scientific calculators also use stack-based operands. For more details on the HP3000 architecture, see [16].

6.2.5 A Comparison

Each of the four address schemes discussed in the previous subsections has certain advantages. If you count the number of instructions needed to execute our example C statement, you notice that this count increases as we reduce the number of addresses. Let us assume that the number of memory accesses represents our performance metric: the lower the number of memory accesses, the better.

In the three-address machine, each instruction takes four memory accesses: one access to read the instruction itself, two for getting the two input operands, and a final one to write the result back in memory. Since there are five instructions, this machine generates a total of 20 memory accesses.

In the two-address machine, each arithmetic instruction still takes four accesses as in the three-address machine. Remember that we are using one address to double as a source and destination address. Thus, the five arithmetic instructions require 20 memory accesses. In addition, we have the load instruction that requires three accesses. Thus, it takes a total of 23 memory accesses.

The count for the accumulator machine is better as the accumulator is a register and reading or writing to it, therefore, does not require a memory access. In this machine, each instruction requires just two accesses. Since there are seven instructions, this machine generates 14 memory accesses.

Finally, if we assume that the stack depth is sufficiently large so that all our push and pop operations do not exceed this value, the stack machine takes 19 accesses. This count is obtained by noting that each `push` or `pop` instruction takes two memory accesses, whereas the five arithmetic instructions take one memory access each.

This comparison leads us to believe that the accumulator machine is the fastest. The comparison between the accumulator and stack machines is fair because both machines assume the presence of registers. However, we cannot say the same for the other two machines. In particular, in our calculation, we assumed that there are no registers on the three- and two-address machines. If we assume that these two machines have a single register to hold the temporary `T`, the count for the three-address machine comes down to 12 memory accesses. The corresponding number for the two-address machine is 13 memory accesses. As you can see from this simple example, we tend to increase the number of memory accesses as we reduce the number of addresses.

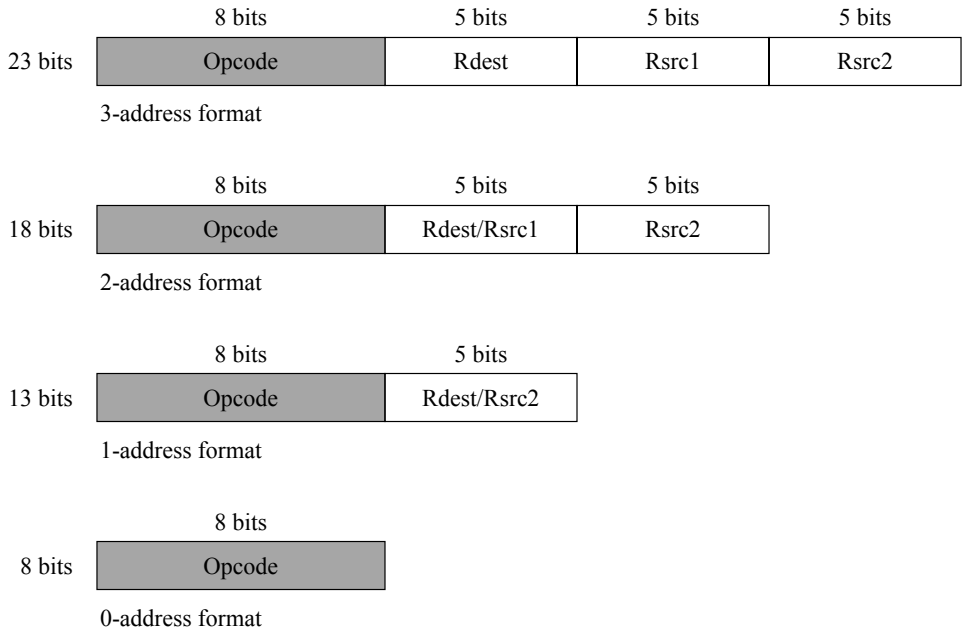

Figure 6.1 Instruction sizes for the four formats: This format assumes that the operands are located in registers.

There are still problems with this comparison. The reason is that we have not taken the size of the instructions into account. Since the stack machine instructions do not need to specify the operand addresses, each instruction takes fewer bits to encode than an instruction in the three-address machine. Of course, the difference between the two depends on several factors including how the addresses are specified and whether we allow registers to hold the operands. We discuss these issues shortly.

Figure 6.1 shows the size of the instructions when the operands are available in the registers. This example assumes that the processor has 32 registers like the MIPS processor and the opcode takes 8 bits. The instruction size varies from 23 bits to 8 bits.

In practice, most systems use a combination of these address schemes. This is obvious from our stack machine. Even though the stack machine is a zero-address machine, it uses load and store instructions that specify an address. Some processors impose restrictions on where the operands can be located. For example, the Pentium allows only one of the two operands to be located in memory. Part V provides details on the Pentium instruction set.

RISC processors take the Pentium's restriction further by allowing most operations to work on the operands located in the processor registers. These processors provide special instructions to move data between the registers and memory. This architecture is called the load/store architecture, which is discussed next.

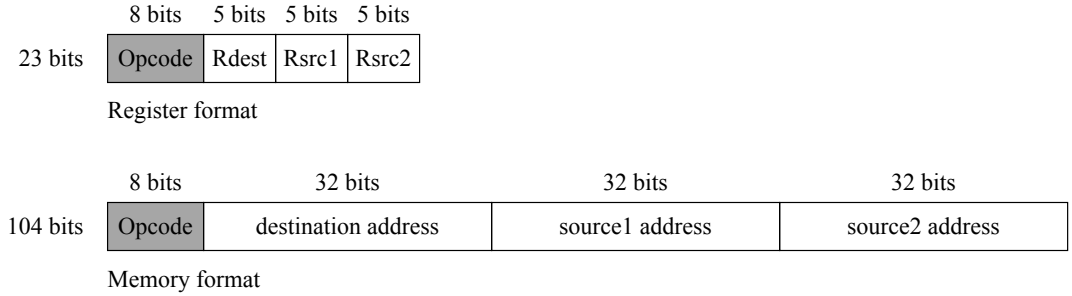

Figure 6.2 A comparison of the instruction size when the operands are in registers versus memory.

6.2.6 The Load/Store Architecture

In the load/store architecture, instructions operate on values stored in internal processor registers. Only load and store instructions move data between the registers and memory. RISC machines as well as vector processors use this architecture, which reduces the size of the instruction substantially. If we assume that addresses are 32 bits long, an instruction with all three operands in memory requires 104 bits whereas the register-based operands require instructions to be 23 bits, as shown in Figure 6.2.

We discuss RISC processors in more detail in Part VI. We look at the vector processors in Chapter 8. Table 6.5 gives some sample instructions for the load/store machines.

In these machines, the C statement

```
A = B + C * D - E + F + A
```

is converted to the following code:

```
load   R1,B      ; load B
load   R2,C      ; load C
load   R3,D      ; load D
load   R4,E      ; load E
load   R5,F      ; load F
load   R6,A      ; load A
mult   R2,R2,R3  ; R2 = C*D
add    R2,R2,R1  ; R2 = B + C*D
sub    R2,R2,R4  ; R2 = B + C*D - E
add    R2,R2,R5  ; R2 = B + C*D - E + F
add    R2,R2,R6  ; R2 = B + C*D - E + F + A
store  A,R2      ; store the result in A
```

Each load and store instruction takes two memory accesses: one to fetch the instruction and the other to access the data value. The arithmetic instructions need just one memory access to fetch the instruction, as the operands are in registers. Thus, this code takes 19 memory accesses.

Table 6.5 Sample load/store machine instructions

Instruction	Semantics
`load Rd,addr`	Loads the `Rd` register with the value at address `addr` Rd = [addr]
`store addr,Rs`	Stores the value in `Rs` register at address `addr` (addr) = Rs
`add Rd,Rs1,Rs2`	Adds the two values in `Rs1` and `Rs2` registers and places the result in `Rd` register Rd = Rs1 + Rs2
`sub Rd,Rs1,Rs2`	Subtracts the value in `Rs2` from that in `Rs1` and places the result in `Rd` register Rd = Rs1 − Rs2
`mult Rd,Rs1,Rs2`	Multiplies the two values in `Rs1` and `Rs2` and places the result in `Rd` register Rd = Rs1 * Rs2

Note that the elapsed execution time is not directly proportional to the number of memory accesses. Overlapped execution reduces the execution time for some processors. In particular, RISC processors facilitate this overlapped execution because of their load/store architecture. We give more details in Chapter 8.

In the RISC code, we assumed that we have six registers to load the values. However, you don't need this many registers. For example, once the value in R3 is used, we can reuse this register. Typically, RISC machines tend to have many more registers than CISC machines. For example, the MIPS processor has 32 registers and the Intel Itanium processor has 128 registers. Both are RISC processors and are covered in Part VI.

6.2.7 Processor Registers

Processors have a number of registers to hold data, instructions, and state information. We can classify the processors based on the structure of these registers and how the processor uses them. Typically, we can divide the registers into general-purpose or special-purpose registers. Special-purpose registers can be further divided into those that are accessible to the user programs and those reserved for the system use. The available technology largely determines the structure and function of the register set.

The number of addresses used in instructions partly influences the number of data registers and their use. For example, stack machines do not require any data registers. However, as noted, part of the stack is kept internal to the processor. This part of the stack serves the same purpose

that registers do. In three- and two-address machines, there is no need for the internal data registers. However, as we have demonstrated before, having some internal registers improves performance by cutting down the number of memory accesses. The RISC machines typically have a large number of registers.

Some processors maintain a few special-purpose registers. For example, the Pentium uses a couple of registers to implement the processor stack. Processors also have several registers reserved for the instruction execution unit. Typically, there is an instruction register that holds the current instruction and a program counter that points to the next instruction to be executed.

Throughout the book we present details on several processors. In total we describe five processors—the Pentium, MIPS, PowerPC, Itanium, and SPARC. Of these five processors, only the Pentium belongs to the CISC category. The rest are RISC processors. Our selection of processors reflects the dominance of the RISC designs in newer processors and the market domination of the Pentium. Pentium processor details are given in the next chapter. Part VI and Appendix H give details on the RISC processors.

6.3 Flow of Control

Program execution, by default, proceeds sequentially. This default behavior is due to the semantics associated with the execution cycle described in Section 1.5. The program counter (PC) register plays an important role in managing the control flow. At a simple level, the PC can be thought of as pointing to the next instruction. The processor fetches the instruction at the address pointed to by the PC (see Figure 1.9 on page 17). When an instruction is fetched, the PC is incremented to point to the next instruction. If we assume that each instruction takes exactly four bytes as in MIPS and SPARC processors, the PC is automatically incremented by four after each instruction fetch. This leads to the default sequential execution pattern. However, sometimes we want to alter this default execution flow. In high-level languages, we use control structures such as `if-then-else` and `while` statements to alter the execution behavior based on some run-time conditions. Similarly, the procedure call is another way we alter the sequential execution. In this section, we describe how processors support flow control. We look at both branch and procedure calls next. Interrupt is another mechanism to alter flow control, which is discussed in Chapter 20.

6.3.1 Branching

Branching is implemented by means of a branch instruction. This instruction carries the address of the target instruction explicitly. Branch instruction in processors such as the Pentium is also called the jump instruction. We consider two types of branches: unconditional and conditional. In both cases, the transfer control mechanism remains the same as that shown in Figure 6.3a.

Unconditional Branch

The simplest of the branch instructions is the *unconditional branch*, which transfers control to the specified target. Here is an example branch instruction:

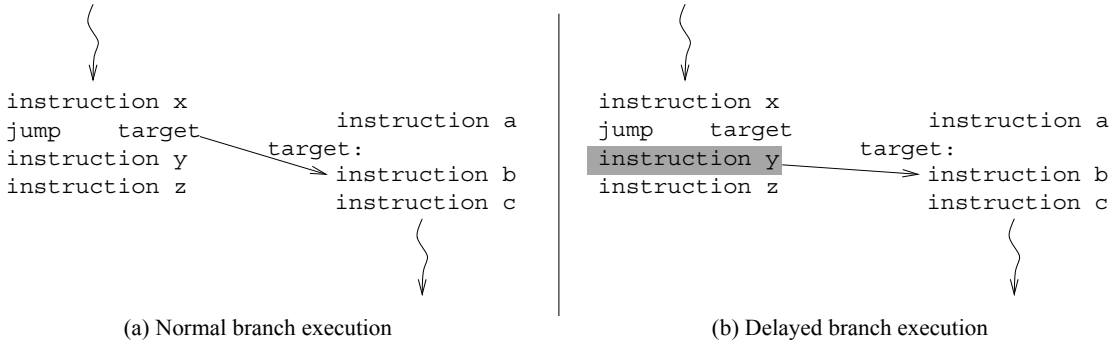

Figure 6.3 Control flow in branching.

```
branch    target
```

Specification of the target address can be done in one of two ways: absolute address or PC-relative address. In the former, the actual address of the target instruction is given. In the PC-relative method, the target address is specified relative to the PC contents. Most processors support absolute address for unconditional branches. Others support both formats. For example, MIPS processors support absolute address-based branch by

```
j    target
```

and PC-relative unconditional branch by

```
b    target
```

In fact, the last instruction is an assembly language instruction, although the processor only supports the `j` instruction.

The PowerPC allows each branch instruction to use either an absolute or a PC-relative address. The instruction encoding has a bit—called the absolute address (AA) bit—to indicate the type of address. As shown on page 589, if AA = 1, the absolute address is assumed; otherwise, the PC-relative address is used.

If the absolute address is used, the processor transfers control by simply loading the specified target address into the PC register. If PC-relative addressing is used, the specified target address is added to the PC contents, and the result is placed in the PC. In either case, since the PC indicates the next instruction address, the processor will fetch the instruction at the intended target address.

The main advantage of using the PC-relative address is that we can move the code from one block of memory to another without changing the target addresses. This type of code is called *relocatable code*. Relocatable code is not possible with absolute addresses.

Conditional Branch

In conditional branches, the jump is taken only if a specified condition is satisfied. For example, we may want to take a branch if the two values are equal. Such conditional branches are handled in one of two basic ways:

- *Set-Then-Jump:* In this design, testing for the condition and branching are separated. To achieve communication between these two instructions, a condition code register is used. The Pentium follows this design, which uses a flags register to record the result of the test condition. It uses a compare (`cmp`) instruction to test the condition. This instruction sets the various flag bits to indicate the relationship between the two compared values. For our example, we are interested in the zero bit. This bit is set if the two values are the same. Then we can use the conditional jump instruction that jumps to the target location if the zero bit is set. The following code fragment, which compares the values in registers AX and BX, should clarify this sequence:

```
        cmp  AX,BX    ;compare the two values in AX and BX
        je   target   ;if equal, transfer control to target
        sub  AX,BX    ;if not, this instruction is executed
          . . .
target:
        add  AX,BX    ;control is transferred here if AX = BX
          . . .
```

  The `je` (jump if equal) instruction transfers control to `target` only if the two values in registers AX and BX are equal. More details on the Pentium jump instructions are presented in Part V.

- *Test-and-Jump:* Most processors combine the testing and branching into a single instruction. We use the MIPS processor to illustrate the principle involved in this strategy. The MIPS provides several branch instructions that test and branch (for a quick peek, see Table 15.9 on page 633). The one that we are interested in here is the branch on equal instruction shown below:

```
beq    Rsrc1,Rsrc2,target
```

  This conditional branch instruction tests the contents of the two registers `Rsrc1` and `Rsrc2` for equality and transfers control to `target` if equal. If we assume that the numbers to be compared are in register `t0` and `t1`, we can write the branch instruction as

```
beq    $t1,$t0,target
```

  This single instruction replaces the two-instruction `cmp/je` sequence used by the Pentium.

Some processors maintain registers to record the condition of the arithmetic and logical operations. These are called *condition code registers.* These registers keep a record of the

status of the last arithmetic/logical operation. For example, when we add two 32-bit integers, it is possible that the sum might require more than 32 bits. This is the overflow condition that the system should record. Normally, a bit in the condition code register is set to indicate this overflow condition. The MIPS, for example, does not use condition registers. Instead, it uses exceptions to flag the overflow condition. On the other hand, the Pentium, PowerPC, and SPARC processors use condition registers. In the Pentium, the flags register records this information. In the PowerPC, this information is maintained by the XER register. SPARC processors use a condition code register.

Some instruction sets provide branches based on comparisons to zero. Some example processors that provide this type of branch instructions include the MIPS and SPARC processors.

Highly pipelined RISC processors support what is known as delayed branch execution. To see the difference between the delayed and normal branch execution, let us look at the normal branch execution shown in Figure 6.3*a*. When the branch instruction is executed, control is transferred to the target immediately. The Pentium, for example, uses this type of branching.

In delayed branch execution, control is transferred to the target after executing the instruction that follows the branch instruction. For example, in Figure 6.3*b*, before the control is transferred, the instruction `instruction y` (shown shaded) is executed. This instruction slot is called the *delay slot*. For example, the SPARC uses delayed branch execution. In fact, it also uses delayed execution for procedure calls. Why does this help? The reason is that by the time the processor decodes the branch instruction, the next instruction is already fetched. Thus, instead of throwing it away, we improve efficiency by executing it. This strategy requires reordering of some instructions. In Appendix H, which gives the SPARC processor details, we give examples of how it affects the programs.

6.3.2 Procedure Calls

The use of procedures facilitates modular programming. Procedure calls are slightly different from the branches. Branches are one-way jumps: once the control has been transferred to the target location, computation proceeds from that location, as shown in Figure 6.3. In procedure calls, we have to return control to the calling program after executing the procedure. Control is returned to the instruction following the call instruction as shown in Figure 6.4.

From Figures 6.3 and 6.4, you will notice that the branches and procedure calls are similar in their initial control transfer. For procedure calls, we need to return to the instruction following the procedure call. This return requires two pieces of information:

- *End of Procedure:* We have to indicate the end of the procedure so that the control can be returned. This is normally done by a special return instruction. For example, the Pentium uses `ret` and the MIPS uses the `jr` instruction to return from a procedure. We do the same in high-level languages as well. For example, in C, we use the `return` statement to indicate an end of procedure execution. High-level languages allow a default fall-through mechanism. That is, if we don't explicitly specify the end of a procedure, control is returned at the end of the block.
- *Return Address:* How does the processor know where to return after completing a proce-

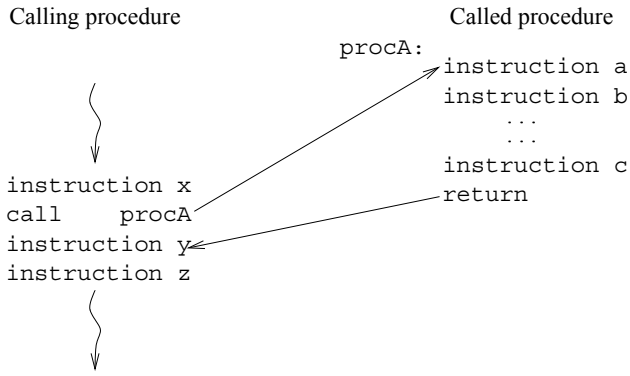

Figure 6.4 Control flow in procedure calls.

dure? This piece of information is normally stored when the procedure is called. Thus, when a procedure is called, it not only modifies the PC as in a branch instruction, but also stores the return address. Where does it store the return address? Two main places are used: a special register or the stack. In processors that use a register to store the return address, some use a special dedicated register, whereas others allow any register to be used for this purpose. The actual return address stored depends on the processor. Some processors such as the SPARC store the address of the `call` instruction itself. Others such as the MIPS and the Pentium store the address of the instruction *following* the `call` instruction.

The Pentium uses the stack to store the return address. Thus, each procedure call involves pushing the return address onto the stack before control is transferred to the procedure code. The return instruction retrieves this value from the stack to send the control back to the instruction following the procedure call. A more detailed description of the procedure call mechanism is found in Chapter 10.

MIPS processors allow any general-purpose register to store the return address. The return statement can specify this register. The format of the return statement is

```
jr    $ra
```

where `ra` is the register that contains the return address. The PowerPC, on the other hand, has a dedicated register, called the link register (LR), to store the return address. Both the MIPS and the PowerPC use a modified branch to implement a procedure call. The advantage of these processors is that simple procedure calls do not have to access memory. In Appendix H, we describe the procedure call mechanism used by the SPARC processor.

Most RISC processors that support delayed branching also support delayed procedure calls. As in the branch instructions, control is transferred to the target after executing the instruction that follows the call (see Figure 6.5). Thus, after the procedure is done, control should be

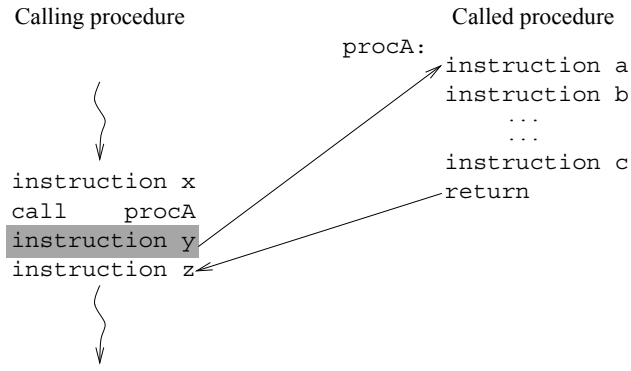

Figure 6.5 Control flow in delayed procedure calls.

returned to the instruction after the delay slot (to `instruction z` in the figure). We show some SPARC examples of this in Appendix H.

Parameter Passing

The general architecture dictates how parameters are passed on to the procedures. There are two basic techniques: register-based or stack-based. In the first method, parameters are placed in processor registers and the called procedure will read the parameter values from these registers. In the stack-based method, parameters are pushed onto the stack and the called procedure would have to pop them off the stack.

The advantage of the register method is that it is faster than the stack method. However, because of the limited number of registers, it imposes a limit on the number of parameters. Furthermore, recursive procedures cannot use the register-based mechanism. Because RISC processors tend to have more registers, register-based parameter passing is used in PowerPC and MIPS processors. The Pentium, due to the small number of registers, tends to use the stack for parameter passing. We describe these two parameter passing mechanisms in detail in Chapter 10.

Recent processors use a register window mechanism that allows a more flexible parameter passing. The SPARC and Intel Itanium processors use this parameter passing mechanism. We describe this method in detail in Chapter 14 and Appendix H.

6.4 Instruction Set Design Issues

There are several design issues that influence the instruction set of a processor. We have already discussed one issue, the number of addresses used in an instruction. Recent processors, except for the Pentium, use three-address instructions. The Pentium, as mentioned, uses the two-address format. In this section, we discuss some other design issues.

6.4.1 Operand Types

Processor instructions typically support only the basic data types. These include characters, integers, and floating-point numbers. Since most memories are byte addressable, representing characters does not require special treatment. Recall that in a byte-addressable memory, the smallest memory unit we can address, and therefore access, is one byte. We can, however, use multiple bytes to represent larger operands. Processors provide instructions to load various operand sizes. Often, the same instruction is used to load operands of different sizes. For example, the Pentium instruction

```
mov     AL,address      /* Loads an 8-bit value */
```

loads the AL register with an 8-bit value from memory at `address`. The same instruction can also be used to load 16- and 32-bit values as shown in the following two Pentium instructions.

```
mov     AX,address      /* Loads a 16-bit value */
mov     EAX,address     /* Loads a 32-bit value */
```

In these instructions, the size of the operand is indirectly given by the size of the register used. The AL, AX, and EAX are 8-, 16-, and 32-bit registers, respectively. In those instructions that do not use a register, we can use size specifiers. We show examples of this in Section 9.5.1 on page 339. This type of specification is typical for the CISC processors.

RISC processors specify the operand size in their load and store operations. Note that only the load and store instructions move data between memory and registers. All other instructions operate on register-wide data. Below we give some sample MIPS load instructions:

```
lb      Rdest,address       /* Loads a byte */
lh      Rdest,address       /* Loads a halfword (16 bits) */
lw      Rdest,address       /* Loads a word (32 bits) */
ld      Rdest,address       /* Loads a doubleword (64 bits) */
```

The last instruction is available only on 64-bit processors. In general, when the size of the data moved is smaller than the destination register, it is sign-extended to the size of `Rdest`. There are separate instructions to handle unsigned values. For unsigned numbers, we use `lbu` and `lhu` instead of `lb` and `lh`, respectively.

Similar instructions are available for store operations. In store operations, the size is reduced to fit the target memory size. For example, storing a byte from a 32-bit register causes only the lower byte to be stored at the specified address. SPARC processors also use a similar set of instructions.

So far we have seen operations on operands located either in registers or in memory. In most instructions, we can also use constants. These constants are called immediate values because these values are available immediately as they are encoded as part of the instruction. In RISC processors, instructions excluding the load and store use registers only; any nonregister value is treated as a constant. In most assembly languages, a special notation is used to indicate registers. For example, in MIPS assembly language, the instruction

```
add     $t0,$t0,-32     /* t0 = t0 - 32 */
```

subtracts 32 from the `t0` register and places the result back in the `t0` register. Notice the special notation to represent registers. But there is no special notation for constants. Pentium assemblers also use a similar strategy. Some assemblers, however, use the "#" sign to indicate a constant.

6.4.2 Addressing Modes

Addressing mode refers to how the operands are specified. As we have seen in the last section, operands can be in one of three places: in a register, in memory, or part of the instruction as a constant. Specifying a constant as an operand is called the *immediate addressing mode*. Similarly, specifying an operand that is in a register is called the *register addressing mode*. All processors support these two addressing modes.

The difference between the RISC and CISC processors is in how they specify the operands in memory. RISC processors follow the load/store architecture. Instructions other than load and store expect their operands in registers or specified as constants. Thus, these instructions use register and immediate addressing modes. Memory-based operands are used only in the load and store instructions. In contrast, CISC processors allow memory-based operands for all instructions. In general, CISC processors support a large variety of addressing modes. RISC processors, on the other hand, support only a few, often just two, addressing modes in their load/store instructions. Most RISC processors support the following two addressing modes to specify the memory-based operands:

- The address of the memory operand is computed by adding the contents of a register and a constant. If this constant is zero, the contents of the register are treated as the operand address. In this mode, the memory address is computed as

  Address = Register + constant.

- The address of the memory operand is computed by adding the contents of two registers. If one of the register contents is zero, this addressing mode becomes the same as the one above with zero constant. In this mode, the memory address is computed as

  Address = Register + Register.

Among the RISC processors we discuss, the Itanium provides slightly different addressing modes. It uses the computed address to update the contents of the register. For example, in the first addressing mode, the register contents are replaced by the value obtained by adding the constant to the contents of the register.

The Pentium provides a variety of addressing modes. The main motivation for this is the desire to support high-level language data structures. For example, one of the Pentium's addressing modes can be used to access elements of a two-dimensional array. We discuss the addressing modes of the Pentium in Chapter 11.

6.4.3 Instruction Types

Instruction sets provide different types of instructions. We describe some of these instruction types here.

Data Movement Instructions: All instruction sets support data movement instructions. The type of instructions supported depends on the architecture. We can divide these instructions into two groups: instructions that facilitate movement of data between memory and registers and between registers. Some instruction sets have special data movement instructions. For example, the Pentium has special instructions such as `push` and `pop` to move data to and from the stack.

In RISC processors, data movement between memory and registers is restricted to load and store instructions. Some RISC processors do not provide any explicit instructions to move data between registers. This data transfer is accomplished indirectly. For example, we can use the `add` instruction

```
add    Rdest,Rsrc,0 /* Rdest= Rsrc + 0 */
```

to copy contents of `Rsrc` to `Rdest`. The Pentium provides an explicit `mov` instruction to copy data. The instruction

```
mov    dest,src
```

copies the contents of `src` to `dest`. The `src` and `dest` can be either registers or memory. In addition, `src` can be a constant. The only restriction is that both `src` and `dest` cannot be located in memory. Thus, we can use the `mov` instruction to transfer data between registers as well as between memory and registers.

Arithmetic and Logical Instructions: Arithmetic instructions support floating-point as well as integer operations. Most processors provide instructions to perform the four basic arithmetic operations: addition, subtraction, multiplication, and division. Since the 2s complement number system is used, addition and subtraction operations do not need separate instructions for unsigned and signed integers. However, the other two arithmetic operations need separate instructions for signed and unsigned numbers.

Some processors do not provide division instructions, whereas others support only partially. What do we mean by partially? Remember that the division operation produces two outputs: a quotient and a remainder. We say that the division operation is fully supported if the division instruction produces both results. For example, the Pentium and MIPS provide full division support. On the other hand, the SPARC and PowerPC only provide the quotient, and the Itanium does not support the division instruction at all.

Logical instructions provide the basic bit-wise logical operations. Processors typically provide logical `and` and `or` operations. Other logical operations including the `not` and `xor` operations are supported by most processors.

Most of these instructions set the condition code bits, either by default or when explicitly instructed. The common condition code bits, which record the status of the most recent operation, are

S — Sign bit (0 = positive, 1 = negative);
Z — Zero bit (0 = nonzero value, 1 = zero value);
O — Overflow bit (0 = no overflow, 1 = overflow);
C — Carry bit (0 = no carry, 1 = carry).

The sign bit is updated to indicate whether the result is positive or negative. Since the most significant bit indicates the sign, the S bit is a copy of the sign bit of the result of the last operation. The zero bit indicates whether the last operation produced a zero or nonzero result. This bit is useful in comparing two values. For example, the Pentium instructions

```
cmp     count,25      /* compare count to 25 */
je      target        /* if equal, jump to target*/
```

compare the value of `count` to 25 and set the condition code bits. The jump instruction checks the zero bit and jumps to `target` if the zero bit is set (i.e., Z = 1). Note that the `cmp` instruction actually subtracts 25 from `count` and sets the Z bit if the result is zero.

The overflow bit records the overflow condition when the operands are signed numbers. The carry bit is set if there is a carry out of the most significant bit. The carry bit indicates an overflow when the operands are unsigned numbers.

In the Pentium, the condition code bits are set by default. In other processors, two versions of arithmetic and logical instructions are provided. For example, in the SPARC processor, `ADD` does not update the condition codes, whereas the `ADDcc` instruction updates the condition codes.

Flow Control and I/O Instructions: The flow control instructions include the branch and procedure calls discussed before. Since we have already discussed these instructions, we do not describe them. Interrupt is another flow control mechanism that is discussed in Chapter 20.

The type of input/output instructions provided by processors varies widely from processor to processor. The main characteristic that influences the I/O instructions is whether the processor supports isolated or memory-mapped I/O. Recall that isolated I/O requires special I/O instructions whereas memory-mapped I/O can use the data movement instructions to move data to or from the I/O devices (see Section 1.7 on page 27).

Most processors support memory-mapped I/O. The Pentium is an example of a processor that supports isolated I/O. Thus, it provides separate instructions to perform input and output. The `in` instruction can be used to read a value from an I/O port into a register. For example, the instruction

```
in    AX,io_port
```

reads a 16-bit value from the specified I/O port. Similarly, the `out` instruction

```
out    io_port,AX
```

writes the 16-bit value in the AX register to the specified I/O port. More details on the Pentium I/O instructions are given in Chapter 19.

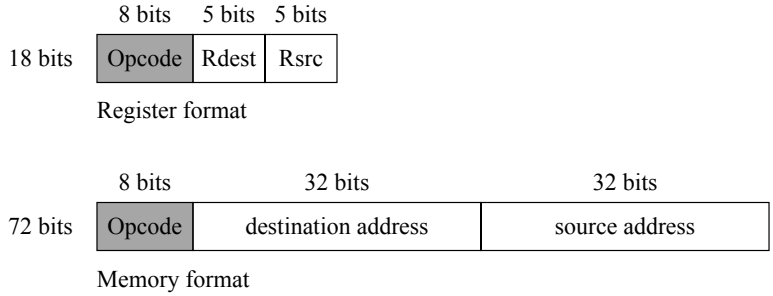

Figure 6.6 Instruction size depends on whether the operands are in registers or memory.

6.4.4 Instruction Formats

Processors use two types of basic instruction format: fixed-length or variable-length instructions. In the fixed-length encoding, all (or most) instructions use the same size instructions. In the latter encoding, the length of the instructions varies quite a bit. Typically, RISC processors use fixed-length instructions, and the CISC designs use variable-length instructions.

All 32-bit RISC processors discussed in this book use instructions that are 32-bits wide. Some examples are the SPARC, MIPS, and PowerPC processors. The Intel Itanium, which is a 64-bit processor, uses fixed-length, 41-bit wide instructions. We discuss the instruction encoding schemes of all these processors throughout the book.

The size of the instruction depends on the number of addresses and whether these addresses identify registers or memory locations. Figure 6.1 shows how the size of the instruction varies with the number of addresses when all operands are located in registers. This format assumes that eight bits are reserved for the operation code (opcode). Thus we can have 256 different instructions. Each operand address is five bits long, which means we can have 32 registers. This is the case in processors like the MIPS. The Itanium, for example, uses seven bits as it has 128 registers.

As you can see from this figure, using fewer addresses reduces the length of the instruction. The size of the instruction also depends on whether the operands are in memory or in registers. As mentioned before, RISC processors keep their operands in registers. In CISC processors like the Pentium, operands can be in memory. If we use 32-bit memory addresses for each of the two addresses, we would need 72 bits for each instruction (see Figure 6.6) whereas the register-based instruction requires only 18 bits. For this and other efficiency reasons, the Pentium does not permit both addresses to be memory addresses. It allows at most one address to be a memory address.

The Pentium, which is a CISC processor, encodes instructions that vary from one byte to several bytes. Part of the reason for using variable length instructions is that CISC processors tend to provide complex addressing modes. For example, in the Pentium, if we use register-based operands, we need just 3 bits to identify a register. On the other hand, if we use a memory-based operand, we need up to 32 bits. In addition, if we use an immediate operand,

we need a further 32 bits to encode this value into the instruction. Thus, an instruction that uses a memory address and an immediate operand needs 8 bytes just for these two components. You can realize from this description that providing flexibility in specifying an operand leads to dramatic variations in instruction sizes.

The opcode is typically partitioned into two fields: one identifies the major operation type, and the other defines the exact operation within that group. For example, the major operation could be a branch operation, and the exact operation could be "branch on equal." These points become clearer as we describe the instruction formats of various processors in later chapters.

6.5 Microprogrammed Control

In the last section, we discussed several issues in designing a processor's instruction set. Let us now focus on how these instructions are executed in the hardware. The basic hardware is the datapath discussed in Chapter 1 (e.g., see page 16). Before proceeding further, you need to understand the digital logic material presented in Chapters 2 and 3.

We start this section with an overview of how the hardware executes the processor's instructions. To facilitate our description, let's look at the simple datapath shown in Figure 6.7. This datapath uses a single bus to interconnect the various components. For the sake of concreteness, let us assume the following:

- The *A bus*, all registers, and the system data and address buses are all 32 bits wide,
- There are 32 general-purpose registers `G0` to `G31`,
- The ALU can operate on 32-bit operands.

Since we are using only a single bus, we need two temporary holding registers: registers A and C. Register A holds the A operand required by the ALU. The output of the ALU is stored in register C. If you have read the material presented on digital logic design in Part II, you will see that the implementation of these registers is straightforward. A sample design is shown in Figure 6.8. A set of 32 D flip-flops is used to latch the A operand for the ALU. As shown in this figure, we use the control input `Ain` to clock in the data. The output of the A register is always available to the A input of the ALU. A similar implementation for the C register uses `Cin` as the clock input signal to store the ALU output. The output of this register is fed to the A bus only if the control signal `Cout` is activated. Later on we show how these control signals are used to execute processor instructions.

The memory interface uses the four shaded registers shown in Figure 6.7. These registers interface to the data and address buses on the one side and to the A bus on the other. Figure 6.9 shows how these registers are interfaced to these buses. Details about the use of these registers and the required control signals are discussed next.

- *PC Register:* This is the program counter register we have discussed before. It contains the address of the next instruction to be executed. In our datapath, we assume that we can place the PC contents on the system address bus. This register can also place its contents

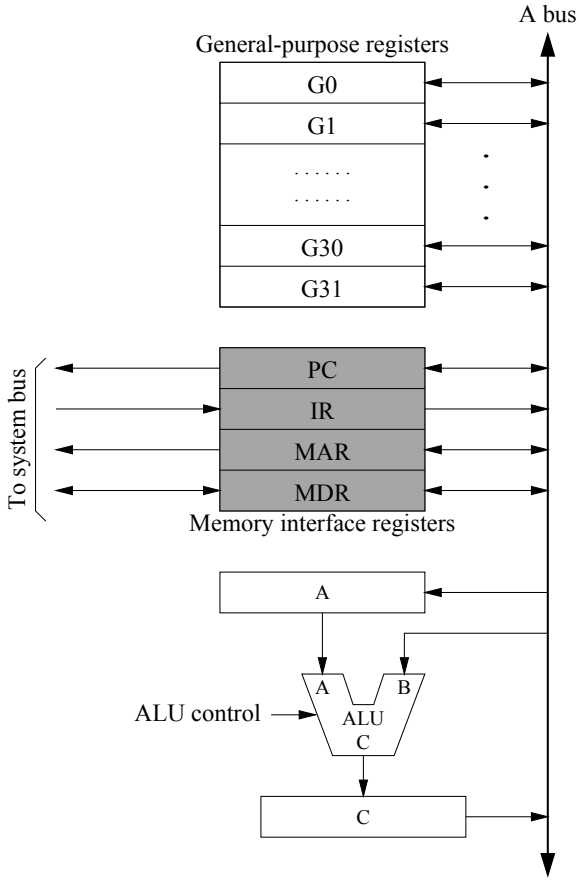

Figure 6.7 An example 1-bus datapath.

on the A bus; in addition, we can write into this register from the A bus. We use `PCin` to load the contents of the A bus into the PC register. The contents of the PC register can be placed on the system address bus and A bus, simultaneously if required. The two control signals, `PCbout` and `PCout`, independently control this operation.

- *IR Register:* The instruction register holds the instruction to be executed. The IR register receives the instruction from the system data bus. Because of its simple interface, we just need `IRbin` and `IRout` control signals.
- *MAR Register:* The memory address register is used to hold the address of an operand stored in memory. This is used in addressing modes that allow an operand to be located in memory. This register interface is similar to that of the PC register. It uses three control signals as does the PC register: `MARbout`, `MARin`, and `MARout`.

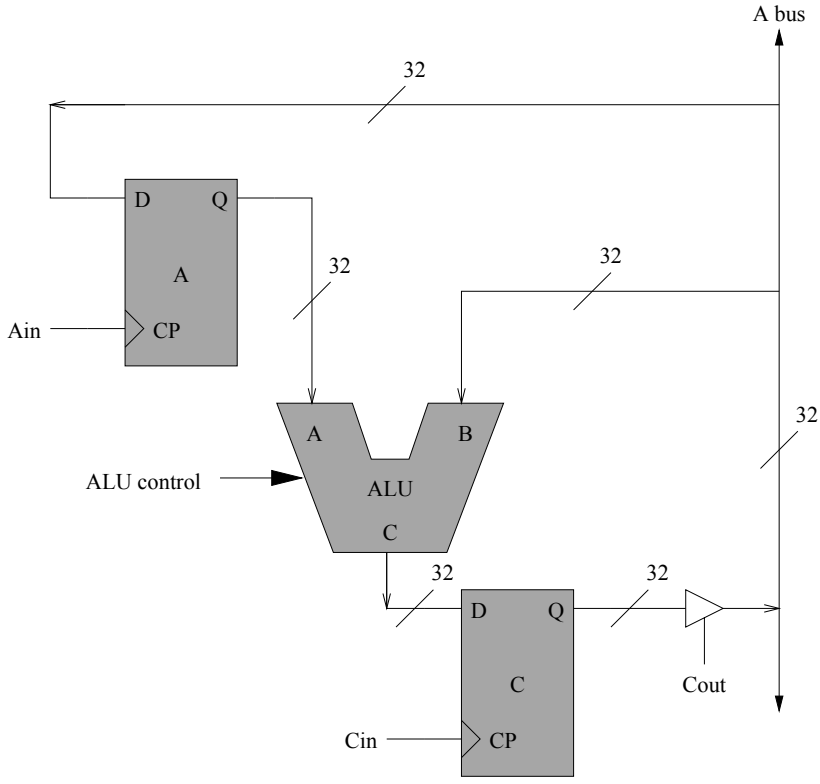

Figure 6.8 ALU circuit details: All datapaths are assumed to be 32 bits wide.

- *MDR Register:* The memory data register is used to hold the operand read from memory. The address of the operand is in MAR. This register provides a bidirectional interface to both the system data bus and the A bus. Thus, we need the four control signals shown in Figure 6.9.

The general-purpose registers interface to the A bus only. Each of the 32 registers has two control signals, `Gxin` and `Gxout`, where `GX` is in the range `G0` to `G31`.

So how do we use this hardware to execute processor instructions? Let us consider the `add` instruction

```
add     Rd,Rs1,Rs2
```

to illustrate the control signals needed to execute instructions. This instruction adds the contents of general-purpose registers `Rs1` and `Rs2` and stores the result in `Rd`. Suppose we want to add the contents of registers 5 and 7 and place the result in register 9. That is, we want to execute the following instruction:

```
add     %G9,%G5,%G7
```

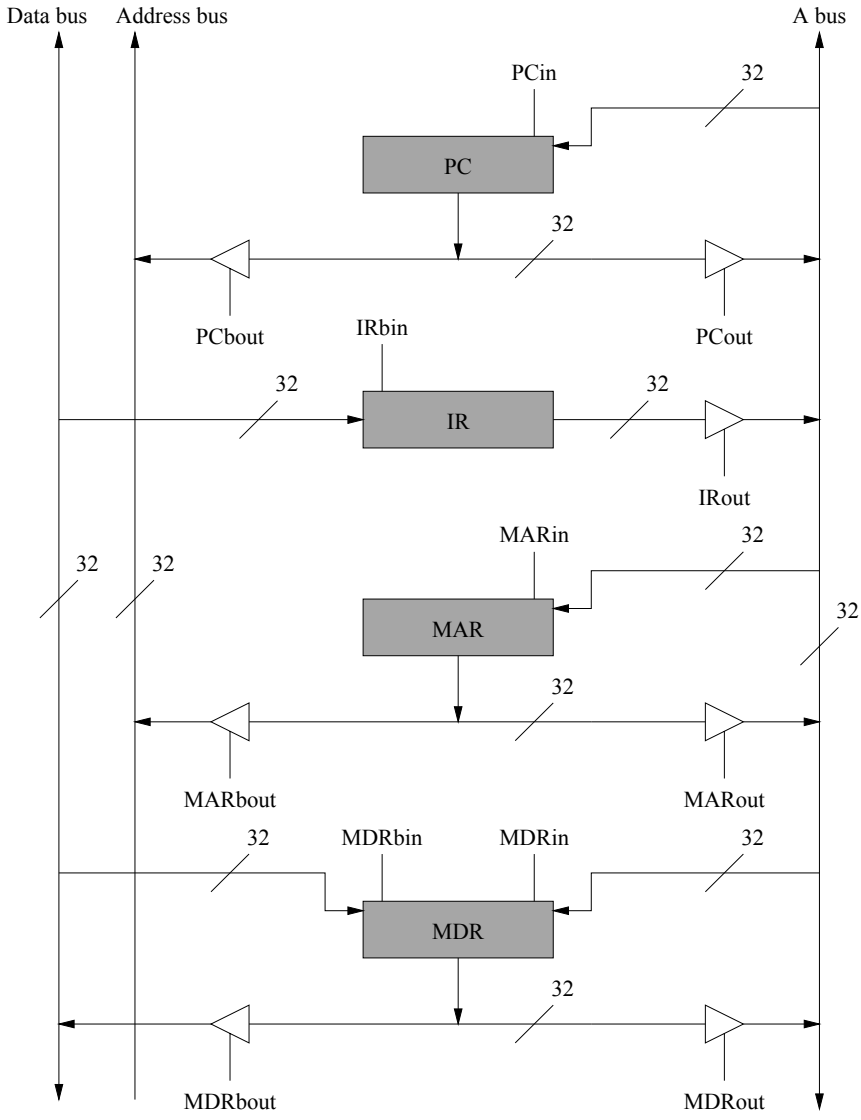

Figure 6.9 Implementation details of the memory interface.

Since we have only a single bus, we have to latch the contents of one of the two source registers in our temporary holding register A. Then we can move the contents of the other register on to the A bus to add the two values. We outline the sequence of steps involved in performing this addition:

1. We assert the `G5out` signal to place the contents of the general-purpose register G5 on the A bus. Simultaneously, we assert the `Ain` signal to latch the A bus contents. Asserting these two control signals simultaneously transfers contents of the G5 register to the A register via the A bus.
2. We now have to place the contents of G7 on the A bus. We do this by asserting the `G7out` control signal. Since the output of the A register is always available to the A input of the ALU, we can now instruct the ALU to perform the addition by specifying appropriate function control. The ALU output is latched into the C register by asserting the `Cin` signal.
3. The final step is to write the value in the C register to G9. This transfer is achieved by asserting `Cout` and `G9in` simultaneously. This step completes the addition operation.

Ideally, the time required to perform each of these steps should be the same. This defines our cycle time. Although the actions taken in Steps 1 and 3 are similar (assume that each step takes one cycle), the second step might require more time. It depends on the time needed by the ALU to perform the addition operation. If this time is more than that required for the other two steps, we can add more cycles to the second step. This is similar to the "wait" cycles inserted into a memory read or write operation (see Section 5.3.2 on page 154).

In our description, we conveniently skipped one important question: How does the processor know that it has to perform the addition operation? This information is obtained from the opcode field of the instruction.

Now we show how instructions are fetched from memory. Instruction fetch involves placing the PC contents on the system address bus and, after waiting for the memory to place the data on the system data bus, reading the data into the IR register. We have to also update the PC to point to the next instruction. We assume that each instruction is 32 bits wide. Updating the PC means adding 4 to the PC contents. As in the `add` instruction execution, we detail the sequence of steps involved below:

1. Assert `PCbout` to place the PC contents on the system address bus. Since we have to update the PC contents to point to the next instruction, we use the services of the ALU to do this. Therefore, we simultaneously pass the PC contents to the ALU via the A bus by asserting the `PCout` signal. The ALU is asked to perform the `add4` operation on its B input. The `add4` is a unary operator that adds 4 to the input. As in the `add` instruction execution, the ALU output is latched into the C register by asserting the `Cin` signal.
2. We wait one clock cycle to give time for the memory to retrieve the instruction. We read this instruction during the next clock cycle. During this cycle, we also load the updated PC value by copying it from the C register. This transfer requires the `Cout` and `PCin` signals.
3. Let us assume that the memory is able to place the data on the system data bus by this clock cycle. All we have to do now is to copy the data into the IR register. We can easily accomplish this by asserting the `IRbin` signal. This completes the instruction fetch operation.

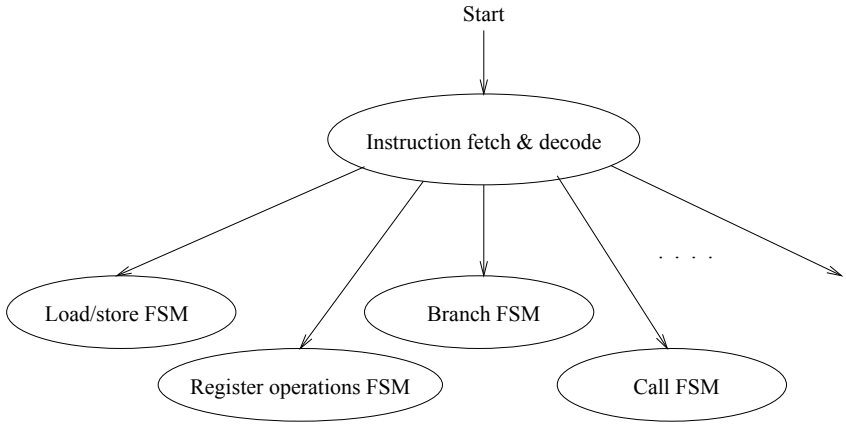

Figure 6.10 A high-level FSM for instruction execution.

The instruction in the IR register is decoded to find the operation to be performed (e.g., `add`). The opcode field specifies the type of operation to be done. If, for example, the operation is addition, we have to identify the source and destination registers and generate control signals to perform the operation as explained before. The behavior of the fetch-decode-execute cycle can be expressed by using a finite state machine. Recall that we have used finite state machines in Chapter 4 to design digital logic circuits.

Figure 6.10 shows a high-level FSM for implementing the instruction execution cycle. The first step is common to all instructions: the instruction must be fetched and decoded. After decoding, the opcode identifies the group to which the instruction belongs. In this figure, we have used typical instruction groups found on RISC machines. We have shown four example instruction groups:

1. *Load/Store Instructions:* These instructions move data between registers and memory. All other instructions operate on the data located in the registers. The FSM associated with this group of instructions will further distinguish the various types of load and store instructions.
2. *Register Instructions:* Instructions in this group include the arithmetic and logical instructions. All required operands are assumed to be in the processor's internal registers. The FSM for this group will generate the necessary control signals depending on the actual instruction (such as `add`, which we have seen before).
3. *Branch Instructions:* These instructions alter the flow control of a program. The target of the branch can be specified directly as a constant in the instruction or indirectly through a register (see our discussion in Section 6.3.1). The branch FSM distinguishes among the different types of branches and generates appropriate control signals.
4. *Call Instructions:* The last group we have shown is the procedure call instructions. As mentioned in our discussion in Section 6.3.2, call instructions are related to the branch

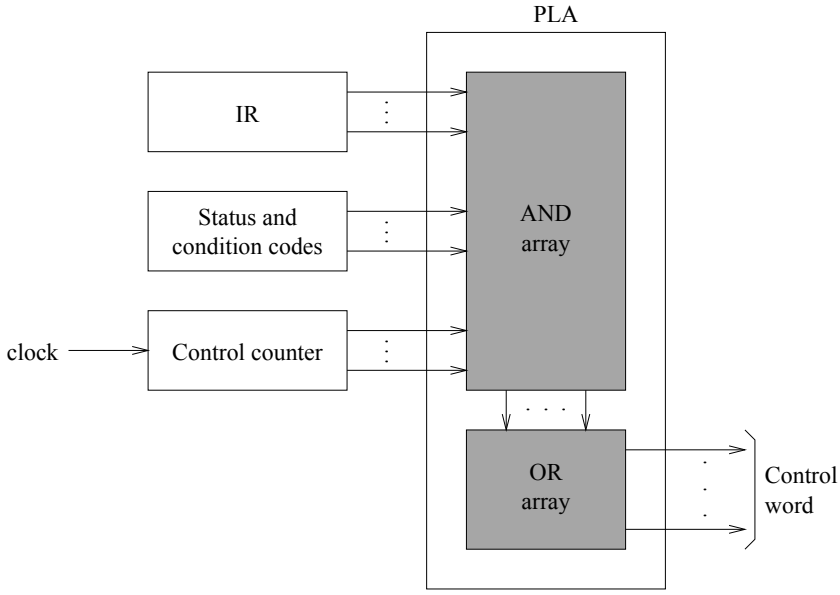

Figure 6.11 Hardware implementation of the controller.

group but are more complicated as they have to return control after completing the procedure.

Depending on the processor instruction set, more instruction groups can be added. Since we have covered FSMs in detail in Chapter 4, we do not discuss this topic in any more detail. Instead, we look at hardware- and software-based implementations of such an FSM.

6.5.1 Hardware Implementation

From our discussion, it is clear that we can implement instructions by generating appropriate control signals. The required control signals are described in the corresponding FSM. As described in Chapter 4, we can implement the FSM in hardware. Figure 6.11 shows an example implementation using a PLA.

The input to the PLA consists of three groups of signals. We need to feed the opcode so that the circuit generates appropriate control signals for that instruction. This input comes from the opcode field of the IR register.

The next group is the status and condition codes. This input is required for instructions such as conditional branches. For example, branch on equal `beq` requires the zero flag input to decide whether to take the branch.

The third input is driven by the clock input. The control counter keeps track of the steps involved in executing an instruction. For example, in executing the `add` instruction, we identified three steps. We can use this counter to specify the control signals that should be generated during each step.

If the instruction set is simple, hardware implementation is preferred. This is typically the case for RISC processors. However, for complex instruction sets, this implementation is not preferred. Instead, a program is used to generate the control signals. This is the approach taken by CISC processors. We describe this approach next.

6.5.2 Software Implementation

The hardware approach is complex and expensive to implement for CISC machines. This was particularly true in the 1960s and 1970s. Furthermore, hardware implementation is very rigid. To avoid these problems, Wilkes and Stinger [40] proposed a software approach. If we look closely, the FSM specifies the control signals that should be generated during each step. To see what we mean, let's rewrite the instruction fetch and add instruction control sequences.

Instruction	Step	Control signals
Instruction fetch	S1	`PCbout: read: PCout: ALU=add4: Cin;`
	S2	`read: Cout: PCin;`
	S3	`read: IRbin;`
	S4	Decodes the instruction and jumps to the appropriate execution routine
`add %G9,%G5,%G7`	S1	`G5out: Ain;`
	S2	`G7out: ALU=add: Cin;`
	S3	`Cout: G9in: end;`

If we assume that each step can be executed in one cycle, we need three cycles to fetch the instruction and at least one cycle to decode the instruction. Another three cycles are needed to execute the `add` instruction. All signals in a single step can be asserted simultaneously. We separate the signals by a colon (:) and use a semicolon (;) to indicate the end of a step. Most of the signals are from the datapath shown in Figure 6.7, but there are some new signals that need explanation. The `read` signal is used to generate the system control bus read signal. As we have seen in Chapter 5, this signal initiates a memory read cycle.

In instruction fetch, we use the ALU function `add4` to update the PC contents. This ALU function adds 4 to the B input. In the `add` instruction, we use `GXout` and `GXin` to control output and input to the general-purpose register `GX`. The `end` signal indicates that the instruction execution has been completed and we should initiate another instruction fetch cycle.

To illustrate the use of the MAR and MDR registers, let us see how the instruction

```
add     %G9,[%G5],%G7
```

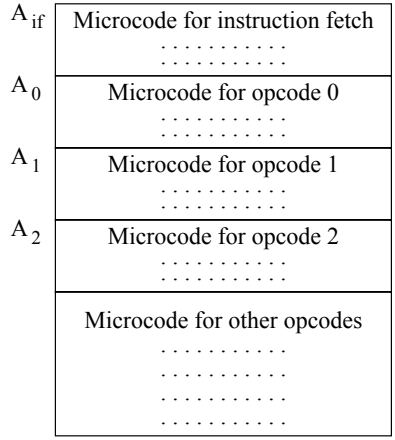

Figure 6.12 A simple microcode organization.

is implemented. This instruction uses register indirect addressing to specify one of the operands, and is very similar to the previous `add` instruction except that one of the operands is in memory. The general-purpose register G5 gives the operand address. To execute this instruction, we need to get this operand from memory. To do this, we place the contents of G5 in MAR and initiate a memory read cycle by placing the address in MAR on the system address bus. After a cycle, the operand from the memory is placed in MDR. From then on, we go through the same sequence of steps as in the previous `add` instruction, as shown below:

Instruction	Step	Control signals
`add %G9,[%G5],%G7`	S1	`G5out: MARin: MARbout: read;`
	S2	`read;`
	S3	`read: MDRbin: MDRout: Ain;`
	S4	`G7out: ALU=add: Cin;`
	S5	`Cout: G9in: end;`

These examples suggest an alternative way of generating the control signals. Suppose that we encode the signals for each step as a codeword. Then we can store these codewords as a program just as with machine language instructions. Each such codeword is referred to as a *microinstruction* and the sequence of codewords for an instruction constitutes a *microroutine*. We can write a *microprogram* that implements the FSM we talked about before.

A straightforward way of structuring the microprogram is shown in Figure 6.12, which shows a linear organization. The instruction fetch microroutine is shown first in this micropro-

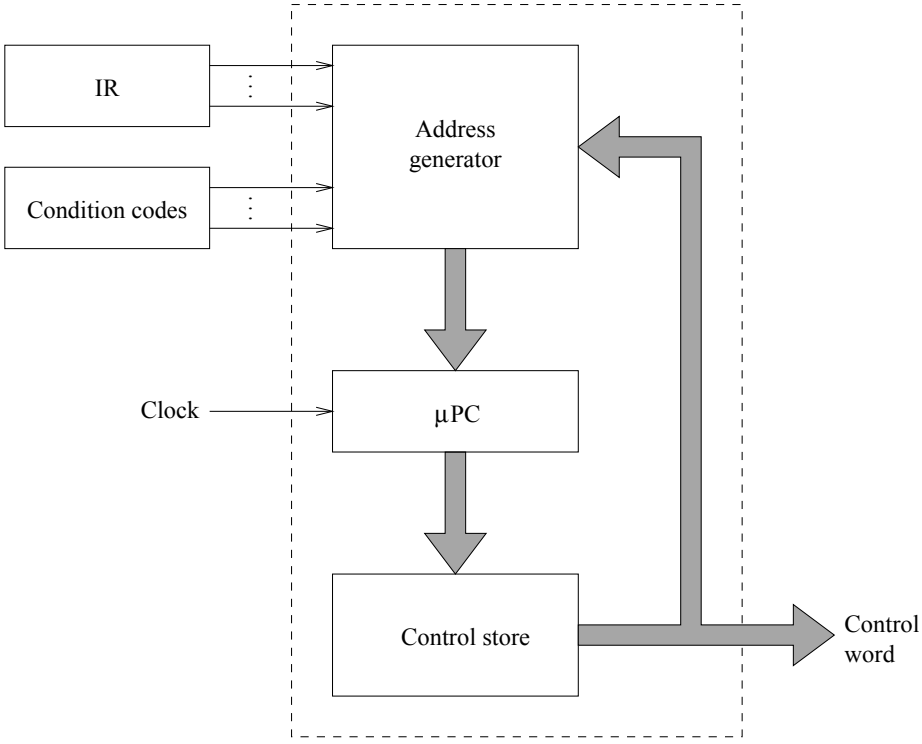

Figure 6.13 Microprogramming uses a control store to control the actions.

gram. After the instruction has been decoded, it jumps to the appropriate microroutine based on the opcode. The execution of the microcode is sequential. When the `end` signal is encountered, the instruction fetch routine is executed.

A microcontroller that executes this microprogram is shown in Figure 6.13. The microprogram is stored in the control store. The microprogram counter (μPC) is similar to the program counter we have for machine language programs. Like the PC, μPC specifies the codeword that should be executed next. The address generation circuit is used to initiate the starting address (i.e., the address of the instruction fetch microroutine) and to implement microprogram jumps. For example, at the end of executing an instruction, the `end` signal causes initiation of the instruction fetch. If we assume that the instruction fetch microroutine is at address 0, the `end` signal can be used to clear the address register to initiate an instruction fetch cycle.

The address generation circuit is also useful to generate the appropriate address depending on the opcode from the IR register and conditional branch type of instructions by taking the condition code inputs. The clock input steps the μPC through the microprogram.

The microprogram organization shown in Figure 6.12 makes the microprogram unnecessarily long as common parts of the code are replicated due to its linear organization. An efficient

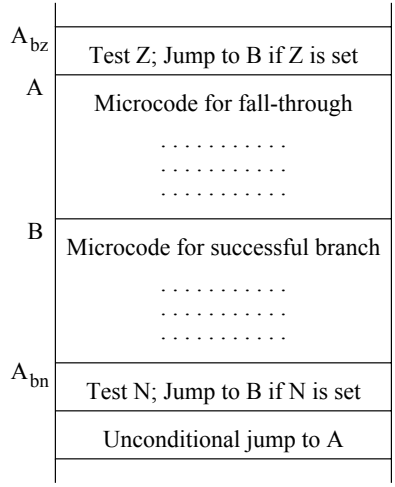

Figure 6.14 Microcode organization to allow conditional branches in the microprogram.

way of organizing the microprogram is shown in Figure 6.14. In this organization, as in the programs we write, we can keep only one copy of the common microcode. To use this organization, however, we have to augment each microinstruction with the address of the next microinstruction. Thus, our control word gets longer than in the other organization. Since we do not replicate the common code, we end up saving space in the control store.

Microinstruction Format

Each microinstruction consists of the control signals needed to execute that instruction on the datapath. Let us consider the single-bus datapath shown in Figure 6.7. The microinstruction format for this datapath is shown in Figure 6.15.

The first group of 12 signals comes from the control signals shown in Figure 6.9. These signals control the memory interface. The next three bits control the A and C latches (see Figure 6.8). The general-purpose registers are controlled by the 64 signals: two for each register. We are assuming that the ALU can perform eight functions: `add`, `add4`, `sub`, `BtoC`, `and`, `or`, `shl`, and `shr`. These functions are self-explanatory except for the following:

- The `add4` function is used to update the PC contents. We have seen an example usage of this function in the instruction fetch microroutine.
- The `BtoC` function copies the B input to the C output. This function, for example, is useful for moving data from one register to another. However, in our single-bus datapath, we can do this transfer without involving the ALU. For example, to copy contents of G5 to G6, we use the following microinstruction:

```
G5out: G6in;
```

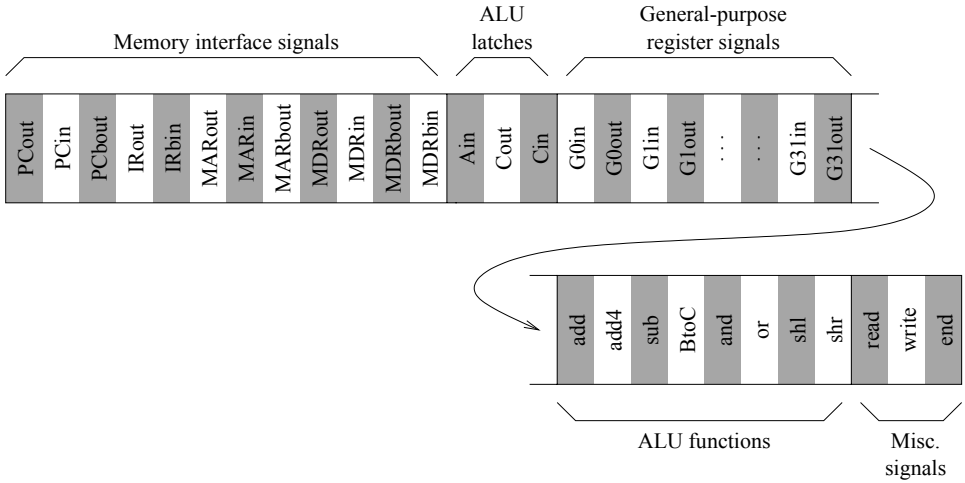

Figure 6.15 A simple microinstruction format for the datapath shown in Figure 6.7. In this organization, there is one bit for each signal. This microcode organization is called the horizontal organization.

As we show next, depending on the encoding scheme used for the microinstructions, we may not be able to specify both `G5out` and `G6in` in the same codeword. Furthermore, as we show later, in 2- and 3-bus systems, such a transfer will have to go through the ALU. In that case, we need a function to pass one of the ALU inputs to the output.

- The `shl` and `shr` functions shift left and right by one bit position, respectively.

These are some of the typical instructions provided by processors. Deriving microinstructions in the format shown in Figure 6.15 is straightforward. For example, the codeword for

```
G5out: G6in;
```

consists of `G5out` = 1, `G6in` = 1, and all the other bits are zero.

The advantage of this microinstruction format is that it allows specification of many actions in a single instruction. For example, we can copy the contents of the G0 register to registers G2 through G5 in one step, as shown below:

```
G0out: G2in: G3in: G4in: G5in: end;
```

The main problem with this format is the size of the microinstruction. In our example, we need 90 bits for each codeword. This encoding scheme follows the *horizontal organization*. Clearly, horizontal organization does not require any decoding of the information contained in the microinstruction. These bits can be used directly to generate control signals.

We can reduce the codeword size by encoding the information in the microinstruction. For example, instead of using 64 bits to control the 32 general-purpose registers, we could use a 5-bit register number and a single bit to indicate in or out control. This type of encoding

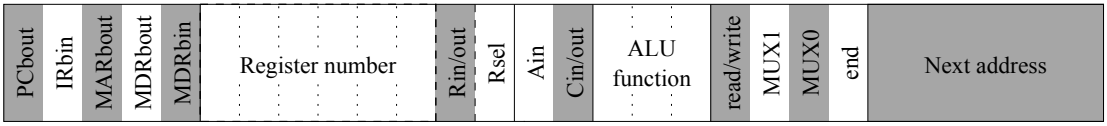

Figure 6.16 A vertical microcode organization.

scheme is called the *vertical organization*. Vertical organization can specify only a few functions compared to horizontal organization. The microinstruction format, shown in Figure 6.16, is organized along these lines. As you can see from this figure, the size of the codeword is substantially smaller: we need only 20 bits (excluding the *Next address* portion and the two MUX signals). Of course, there is no free lunch. This format needs additional decoders and execution takes more cycles. Since we can specify only one of the 32 general-purpose registers, even a simple register-to-register copy takes two steps. For example, to copy the contents of G0 to G2, we use the following microinstruction sequence:

```
G0out: ALU=BtoC: Cin;
Cout: G2in: end;
```

To further demonstrate the disadvantages of the vertical microcode organization, let's look at the copying example discussed before. To copy G0 to G2 through G5, we need several cycles:

```
G0out: ALU=BtoC: Cin;
Cout: G2in;
Cout: G3in;
Cout: G4in;
Cout: G5in: end;
```

This code also shows the need for the `BtoC` ALU function.

The semantics of the microinstruction format shown in Figure 6.16 requires some explanation. The encoding scheme uses six bits for the register number. The reason is that we have 32 general-purpose register and four memory interface registers. When the most significant bit of the "Register number" field is 0, the remaining five bits are taken as the register number of a general-purpose register. When this bit is 1, each of the remaining four bits is used to indicate one of the four memory interface registers—PC, IR, MAR, and MDR, as shown below:

Register number field	Register specified
`0xxxxx`	General-purpose register `Gxxxxx`
`100001`	PC register
`100010`	IR register
`100100`	MAR register
`101000`	MDR register

The first five bits are used to generate memory control signals for the four memory interface registers. The Rin/out signal specifies the in (1) or out (0) direction of data movement on the A bus. These actions take place only when the register selection signal Rsel is asserted. When this bit is 0, no register is selected. For example, we set Rsel to 0 if we want to feed the C register back to the A register. The read and write actions are combined into a single control bit. Like the Rin/out bit, the Cin/out bit is used to control data movement to the C register. For the A register, we just have a single Ain control bit.

The ALU functions are encoded using three bits as follows:

ALU function field	Function specified
000	add
001	add4
010	sub
011	BtoC
100	and
101	or
110	shl
111	shr

Deriving microinstructions in the vertical format is simple. For example, the microinstruction

```
G0out: ALU=BtoC: Cin;
```

can be encoded as

Register number = 000000
Rin/out = 0
ALU function = 011
Cin/out = 1

All other fields in the instruction are zero.

The register number of a general-purpose register comes from two main sources:

- The machine instruction in IR specifies the registers that should be used in the current instruction. In our case, we need three register specifications: two source registers Rs1 and Rs2 and a destination register Rd.
- The microinstruction can also specify one of these registers using the register number field.

To select a particular register, we can use a multiplexer/decoder circuit such as the one shown in Figure 6.17. The microinstruction controls the multiplexer function using the MUX1 and MUX0

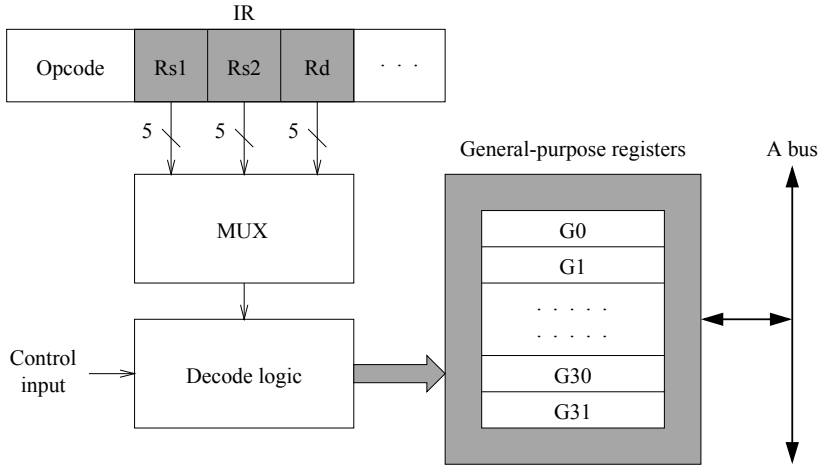

Figure 6.17 An example general register control circuit.

control bits. The decoder circuit takes the 5-bit multiplexer output and `Rin/out` and `Rsel` control inputs from the microinstruction to enable and select a register. If the `Rsel` signal is inactive, no G register is selected.

We mentioned before that, to provide complete support for microinstruction branching, we need to add the next microinstruction address. The format shown in Figure 6.16 includes this field. The microcontroller shown in Figure 6.18 shows how the vertically organized microcode is executed.

The microinstruction register (μIR) holds the microinstruction. Since we are using the vertical organization, we need to decode the microinstruction to generate the control signals. The μPC provides the address of the microinstruction to the control store. The μPC can be loaded from either the `Next address` field or from the start address generator. The start address generator outputs the appropriate microroutine address depending on the opcode and control codes.

The designer needs to weigh the pros and cons of the horizontal and vertical microprogram organizations. For example, to provide improved performance, the horizontal scheme is preferred, as it does not impose any restrictions on the concurrent use of the recourses in the datapath. But the microprogram tends to be large, and the hardware cost increases. If the designer wants a cheaper version, vertical organization may be used. The downside is the reduced performance due to the restrictions imposed by the vertical scheme.

Datapaths with More Buses

In our discussions so far, we have used a single-bus datapath. The use of single bus forces us to multiplex the bus to transfer operands, which takes more time to execute instructions. Figure 6.19 shows a 2-bus datapath with one input bus (A bus) and one output bus (C bus).

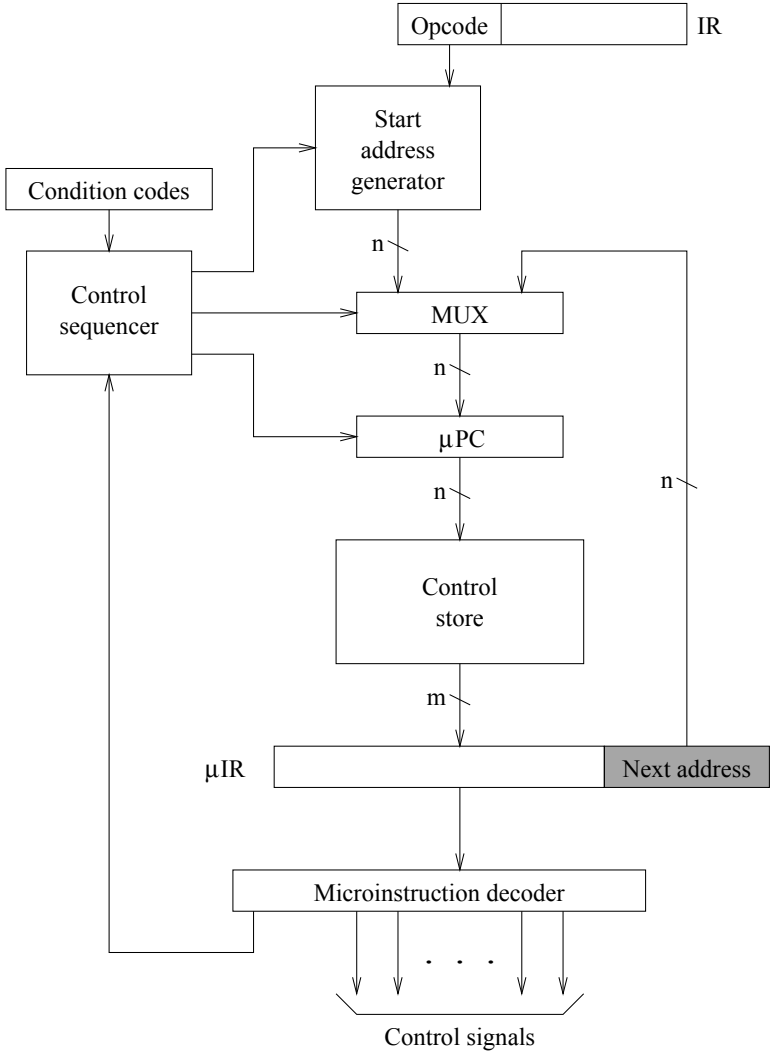

Figure 6.18 Microprogramming uses a control store to control the actions.

Since we have a separate output bus, we do not need the C register to capture the ALU output.

The availability of two buses reduces the time needed to execute instructions. To see the impact of two buses, let us implement the `add` instruction:

```
add     %G9,%G5,%G7
```

The microroutine for this instruction is shown below:

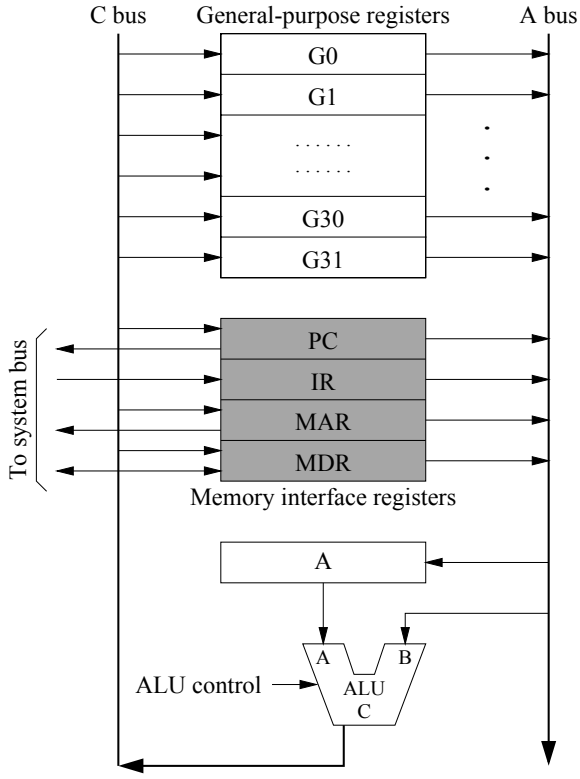

Figure 6.19 An example 2-bus datapath.

Instruction	Step	Control signals
`add %G9,%G5,%G7`	S1	`G5out: Ain;`
	S2	`G7out: ALU=add: G9in;`

Compared to the single-bus datapath, we reduce the number of steps by one. How do we execute this instruction on the 3-bus datapath shown on page 16? We just need one step to execute this instruction:

Instruction	Step	Control signals
`add %G9,%G5,%G7`	S1	`G5outA: G7outB: ALU=add: G9in;`

Notice that the 3-bus data path does not require the A and C registers. However, since the registers are connected to both A and B buses, we have to specify which bus should receive the register output. We use the notations `G?outA` and `G?outB` to indicate that register `G?` output should be placed on the A bus and B bus, respectively.

6.6 Performance

Measuring performance of a computer system is a complex task. It is easy to define a set of very complex performance metrics, but such metrics are not useful in practice. In addition, complex metrics make performance measurement difficult. What we need is a simple yet representative metric that captures the capabilities of the computer system for our usage. The keyword is *our usage*, which means we want a metric that takes into account the kind of applications we run on our system. For example, if we plan to run scientific applications involving lots of floating-point calculations, there is no point in knowing how many integer calculations a given machine can perform. Similarly, if our application almost always uses character manipulation, we don't find much use in knowing how many integer and floating-point calculations per second the system can do. Thus, it is important to take the expected mix of applications, also called the *workload*, into account and derive metrics that make sense for the target user group.

A workload typically consists of a suite of representative programs, which can be executed to measure the time. If this suite of applications represents the target user application mix reasonably, then we can compare the performance of different systems by comparing the execution times for this particular workload. Obviously, if machine X executes the workload in 300 seconds and machine Y takes 330 seconds, we say that machine X is better *for this workload*. You should note that, if we change the workload, it is quite possible that machine Y performs better than machine X for the new workload. The point to take away from this discussion is that the workload is important in comparing the performance of different machines.

If your company is evaluating the performance of two computer systems from competing manufacturers, you can run some typical programs from your application mix and compare the execution times. However, not every company is going spend time and effort in evaluating the machine they intend to buy. Furthermore, such a strategy does not always work. For example, if you are a designer working on a new processor, it is not feasible to run programs during the initial stages of the design process. Furthermore, there may not be an optimizing compiler available for you to run the programs. Thus, we need some general-purpose metrics that give us an idea of the relative performance of various systems.

We already mentioned that the workload is important. This implies that we should not define a single metric and use it for all purposes. Standard bodies define a set of *benchmark* programs that approximate the intended real-world applications. Benchmarks can be *real* programs taken from sample applications or *synthetic*. In synthetic benchmarks, artificial programs are created to exercise the system in a specific way. For example, the Whetstones and Dhrystones benchmarks, described later, are examples of synthetic benchmarks. In Section 6.6.4, we describe some real benchmarks from SPEC.

6.6.1 Performance Metrics

Computer system performance can be measured by several performance metrics. The metrics we use depend on the purpose as well as the component of the system in which we are interested. For example, if you are interested in the network component, we can use network bandwidth, which tells us the number of bits it can transmit per second. Two common metrics are used for almost all components: *response time* and *throughput*. Response time expresses the time needed to execute a task. For example, on a network, we may be interested in message delivery time. In this context, message delivery time represents the response time.

Throughput refers to the rate of flow. Looking at the network example again, throughput of the network represents the number of messages delivered per second. In this section, in order to limit the scope of our discussion, we focus on the processor.

For processors, response time represents the time it takes for a job to complete its execution. Response time includes the time to preprocess the job, any waiting time if the processor is busy with other jobs, and the actual execution time.

As you can see from this discussion, the response time metric is something in which a user is interested. When we say time, we usually mean the wall clock time, the amount of time the user had to wait to finish the job. This time consists of the actual CPU time spent on the job and waiting time that includes the time to access the disk and execute other jobs.

Throughput expresses the system capacity. For example, we say, "The system can execute 100 transactions per second," to express its capacity. Throughput is a system metric, whereas the response time is a single job metric. As a result of this orientation, users are interested in minimizing response times for their jobs, whereas a system administrator strives to get higher throughput from the system. Often, these two goals are conflicting. To minimize response time to your job, you don't want the system to run any other job, which wastes system resources. To increase throughput, you want to run a certain number of jobs concurrently so that the system resources are well utilized. A compromise is usually struck between these conflicting objectives. In Section 6.6.4, we show that both response time and throughput are used to characterize a system.

MIPS and MFLOPS are sometimes used as performance metrics. MIPS stands for millions of instructions per second. Although it is a simple metric, it is practically useless to express the performance of a system. Since instructions vary widely among the processors, a simple instruction execution rate will not tell us anything about the system. For example, complex instructions take more clocks than simple instructions. Thus, a complex instruction rate will be lower than that for simple instructions. The MIPS metric does not capture the actual work done by these instructions. MIPS is perhaps useful in comparing various versions of processors derived from the same instruction set.

MFLOPS is another popular metric often used in the scientific computing area. MFLOPS stands for millions of floating-point operations per second. This is a far better metric than MIPS as it captures the number of operations in which the user is interested. This measure also takes various system overheads to read operands, store results, and loop testing. We later look at more useful metrics.

6.6.2 Execution Time Calculation

The time required to execute a program represents an intrinsic measure of a processor's capability. Execution time depends on the following three factors:

- *Instruction Count (IC):* We need to know the number of instructions required to execute the program. Obviously, the more instructions a program takes, the more time it needs to execute the program. If all instructions of a processor take more or less the same amount of time to execute, a simple instruction count is sufficient. If, on the other hand, instruction execution times vary widely as in CISC processors, we need to get an effective instruction count.
- *Clocks per Instruction (CPI):* This represents time in terms of CPU clocks required for an average instruction. In RISC processors most instructions take the same number of clocks. In CISC processors, however, the clock count depends on the instruction type. In such cases, we can take the average value. We describe later a number of ways one can compute averages.
- *Clock Period (T):* Clock period is defined as the time taken by a single clock cycle.

Given these three factors, we can estimate the execution time of a program as

$$\text{Execution time} = IC \times CPI \times T\,. \tag{6.1}$$

We can then define performance of a system as

$$\text{Performance} = \frac{1}{\text{Execution time}}\,. \tag{6.2}$$

These three factors provide us with an understanding of the impact of various improvements on the performance. For example, we double the performance by increasing the clock rate from 500 MHz to 1 GHz, which reduces the clock period from 2 to 1 ns. In reality, application performance is dependent on many other factors including the number of other programs running on the system, the performance of the cache subsystem, and I/O subsystem latencies.

6.6.3 Means of Performance

We often want a single summarizing metric to get an idea of performance, even though we may conduct several experiments. Once the appropriate workload has been identified and the performance metric has been selected, we need to find a method to get a value for the performance metric. There are several ways of obtaining such a metric. We start with the simplest of all, the *arithmetic mean*. Suppose you run two programs to evaluate a system. If the individual execution times are 100 seconds (for Program 1) and 80 seconds (for Program 2), we compute the arithmetic mean as

$$\text{Mean execution time} = \frac{100 + 80}{2} = 90 \text{ seconds}.$$

In general, the arithmetic mean of n numbers $a_1, a_2, \ldots, a_n$ is computed as

$$\text{Arithmetic mean} = \frac{1}{n}\sum_{i=1}^{n} a_i\,,$$

where $\sum_{i=1}^{n} a_i = a_1 + a_2 + \cdots + a_n$.

There is one implicit assumption in our arithmetic mean calculation of the two programs: We assume that both programs appear equally likely in the target workload. Now suppose we know that Program 2 appears three times more often than Program 1. What would be the summary execution time that reflects this reality? Of course, we want to give three times more weight to the execution time of Program 2. That is, the mean is computed as

$$\text{Mean execution time} = \frac{1 \times 100 + 3 \times 80}{4} = 85 \text{ seconds.}$$

This is called the *weighted arithmetic mean*. This computation assigns a weight for each value. This fact becomes clear if we rewrite the previous expression as

$$\text{Mean execution time} = \frac{1}{4} \times 100 + \frac{3}{4} \times 80 = 85 \text{ seconds.}$$

This expression clearly shows that Program 1 execution time is given a weight of 1/4 and the other program 3/4. The general formula is

$$\text{Weighted mean execution time} = \sum_{i=1}^{n} w_i \times a_i\,,$$

where w_i is the weight expressed as a fraction. In our example, Program 1 has a weight of 25% and Program 2 has 75%. We express these weights as $w_1 = 0.25$ and $w_2 = 0.75$. Note that all weights should add up to 1. That is, $\sum_{i=1}^{n} w_i = 1$. The normal arithmetic mean is a special case of the weighted arithmetic mean with equal weights.

The weighted arithmetic mean is fine for metrics such as the response time to look at the performance of a single system. When comparing relative performance of two systems, it does cause problems. Let's assume that the response time of each machine is expressed relative to a reference machine. For example, most performance metrics from SPEC are expressed as a ratio relative to a reference machine. Table 6.6 shows an example to demonstrate the problems in using the arithmetic means. It lists the execution times of two programs on two machines (A and B) and a reference machine (REF). The first two columns under "Normalized values" give response time values normalized to the reference machine. That is, these values are obtained by dividing the response time of machines A and B by the corresponding response times for the reference machine.

When we use the arithmetic mean, we get 30.25 and 36 for machines A and B, respectively. The next column, labeled "Ratio," gives the ratio of B over A (i.e., 36/30.25) as 1.19. When we compute the corresponding ratio using the normalized values, we get 1.16. Clearly, there is a mismatch between the two values.

Table 6.6 Arithmetic versus geometric mean

	Response time on machine				Normalized values		
	REF	A	B	Ratio	A	B	Ratio
Program 1	10	11	12		1.1	1.2	
Program 2	40	49.5	60		1.24	1.5	
Arithmetic mean		30.25	36	1.19	1.17	1.35	1.16
Geometric mean		23.33	26.83	1.15	1.167	1.342	1.15

This is where the geometric mean is useful. The geometric mean of n numbers $a_1, a_2, \ldots, a_n$ is defined as

$$\text{Geometric mean} = \sqrt[n]{\prod_{i=1}^{n} a_i} \qquad \text{or} \qquad \left(\prod_{i=1}^{n} a_i\right)^{1/n},$$

where $\prod_{i=1}^{n} a_i = a_1 \times a_2 \times \cdots \times a_n$. When we use the geometric mean, we get a matching value of 1.15 for the two ratios computed from the normalized and the original values. This is because the geometric mean has the property

$$\frac{\text{Geometric mean}(a_i)}{\text{Geometric mean}(b_i)} = \text{Geometric mean}\left(\frac{a_i}{b_i}\right).$$

Analogous to the weighted arithmetic mean, we can also define the weighed geometric mean as

$$\text{Weighted geometric mean} = \prod_{i=1}^{n} {a_i}^{w_i},$$

where w_i is the weight as defined in the weighted arithmetic mean. The geometric mean can be used to maintain consistency in summarizing normalized results. Unfortunately, geometric means do not predict execution times. To see this, consider the execution times of two machines, A and B, shown in Table 6.7. The arithmetic mean says that Machine A is about three times faster than Machine B. On the other hand, the geometric mean suggests that both machines perform the same. Why? The geometric mean keeps track of the performance ratio. Since Program 1 runs 10 times faster on Machine A and Program B runs 10 times faster on Machine B, by using the geometric mean we erroneously conclude that the average performance of the two programs is the same.

The geometric mean, however, is useful when our metric is a ratio, like the throughput. For example, the SPECviewperf benchmark from SPEC, which measures the 3D rendering performance of systems running under OpenGL, uses the weighted arithmetic mean [34]. This benchmark uses a throughputlike measure (frames/second) as the unit.

Table 6.7 An example to demonstrate the drawback of the arithmetic mean

	Response time on machine	
	A	B
Program 1	20	200
Program 2	50	5
Arithmetic mean	35	102.5
Geometric mean	31.62	31.62

6.6.4 The SPEC Benchmarks

We mentioned two types of benchmarks: synthetic and real. Synthetic benchmarks are programs specifically written for performance testing. Whetstone and Dhrystone benchmark programs are two example synthetic benchmarks. The *Whetstones* benchmark, named after the Whetstone Algol compiler, was developed in the mid-1970s to measure floating-point performance. The performance is expressed in MWIPS, millions of Whetstone instructions per second. The *Dhrystone* benchmark was developed in 1984 to measure integer performance. Both these benchmarks are small programs. A drawback with these benchmarks is that they encouraged excessive optimization by compilers to distort the performance results.

As computer systems become more complex, we need to measure performance of various components for different types of applications. The Standard Performance Evaluation Corporation (SPEC) was formed as a nonprofit consortium consisting of computer vendors, system integrators, universities, research organizations, publishers, and consultants. The objective is to provide benchmarks to measure performance of components as well as the system as a whole for multiple operating systems and environments. These benchmarks would be based on real-world applications. To give you an idea of the types of benchmarks provided, we describe some sample benchmarks next.

SPEC CPU2000

This benchmark is used for measuring the processor performance, memory, and compiler. The previous version, CPU95, was retired at the end of June, 2000. For this benchmark, applications are classified as "integer" if they spend less than 1% of their time performing floating-point calculations. This definition covers most nonscientific applications such as compilers, utilities, and simulators [34, 18]. SPEC CPU2000 consists of 26 applications that span four languages: C, C++, FORTRAN 77, and FORTRAN 90. SPEC CPU2000 consists of integer and floating-point components.

Table 6.8 SPEC CINT2000 integer benchmarks

Benchmark	Language	Description
164.gzip	C	Compression (A GNU data compression program)
175.vpr	C	Integrated circuit computer-aided design program (It performs field-programmable gate arrays (FPGA) circuit placement and routing.)
176.gcc	C	Compiler (GNU C compiler)
181.mcf	C	Combinatorial optimization program (It performs single-depot vehicle scheduling in public mass transportation.)
186.crafty	C	Game-playing program (chess)
197.parser	C	Word-processing program (a syntactic parser of English, which has a dictionary of about 60,000 word forms)
252.eon	C++	Computer visualization program (a probabilistic raytracer)
253.perlbmk	C	PERL programming language (The reference workload consists of four scripts.)
254.gap	C	Group theory (an interpreter used to implement a language and library designed mostly for computing in groups)
255.vortex	C	Object-oriented database (a single-user object-oriented database transaction benchmark)
256.bzip2	C	Compression (This is based on Julian Seward's bzip2 version 0.1.)
300.twolf	C	Place and route simulator (a placement and global routing package used for creating the microchip lithography artwork)

CINT2000: This is an integer benchmark to measure the performance for integer operations. This benchmark consists of the 12 applications shown in Table 6.8.

CFP2000: This is a floating-point benchmark that measures the performance for floating-point operations. It consists of the 14 applications shown in Table 6.9. As you can see from this list, these applications are all derived mainly from a scientific computation workload.

Performance is expressed relative to a reference machine, which is a 300 MHz Sun Ultra 5. This machine gets a score of 100. Integer and floating-point performance of various Pentium III and 4 processors is shown in Figure 6.20.

Table 6.9 SPEC CFP2000 floating-point benchmarks

Benchmark	Language	Description
168.wupwise	FORTRAN 77	Physics/quantum chromodynamics
171.swim	FORTRAN 77	Shallow water modeling
172.mgrid	FORTRAN 77	Multigrid solver (3D potential field)
173.applu	FORTRAN 77	Parabolic/elliptic partial differential equations
177.mesa	C	3D graphics library
178.galgel	FORTRAN 90	Computational fluid dynamics
179.art	C	Image recognition/neural networks
183.equake	C	Seismic wave propagation simulation
187.facerec	FORTRAN 90	Image processing (face recognition)
188.ammp	C	Computational chemistry
189.lucas	FORTRAN 90	Number theory/primality testing
191.fma3d	FORTRAN 90	Finite-element crash simulation
200.sixtrack	FORTRAN 77	High energy nuclear physics accelerator design
301.apsi	FORTRAN 77	Meteorology (pollutant distribution)

SPECmail2001

This is a standardized mail server benchmark designed to measure a system's ability to service email requests. It was developed by mail server vendors and research organizations to enable performance evaluation of systems supporting the Post Office Protocol (POP3) and Simple Mail Transfer Protocol (SMTP). This benchmark uses both throughput and response time to characterize a mail server system with realistic network connections, disk storage, and client workloads. The benchmark focuses on the ISPs with 10,000 to 1,000,000 users. It can also be used by vendors to test and finetune products under development.

Results from SPECmail2001 are based on a messages-per-minute rating that indicates the load the mail server can sustain with a reasonable quality of service. For example, Mirapoint MSR 2.8 has a SPECMail2001 rating of 2000 messages/minute. It uses a single 400 MHz Pentium II processor with 32 KB of primary cache (16 KB of instruction cache and 16 KB data cache) and 512 KB of secondary cache.

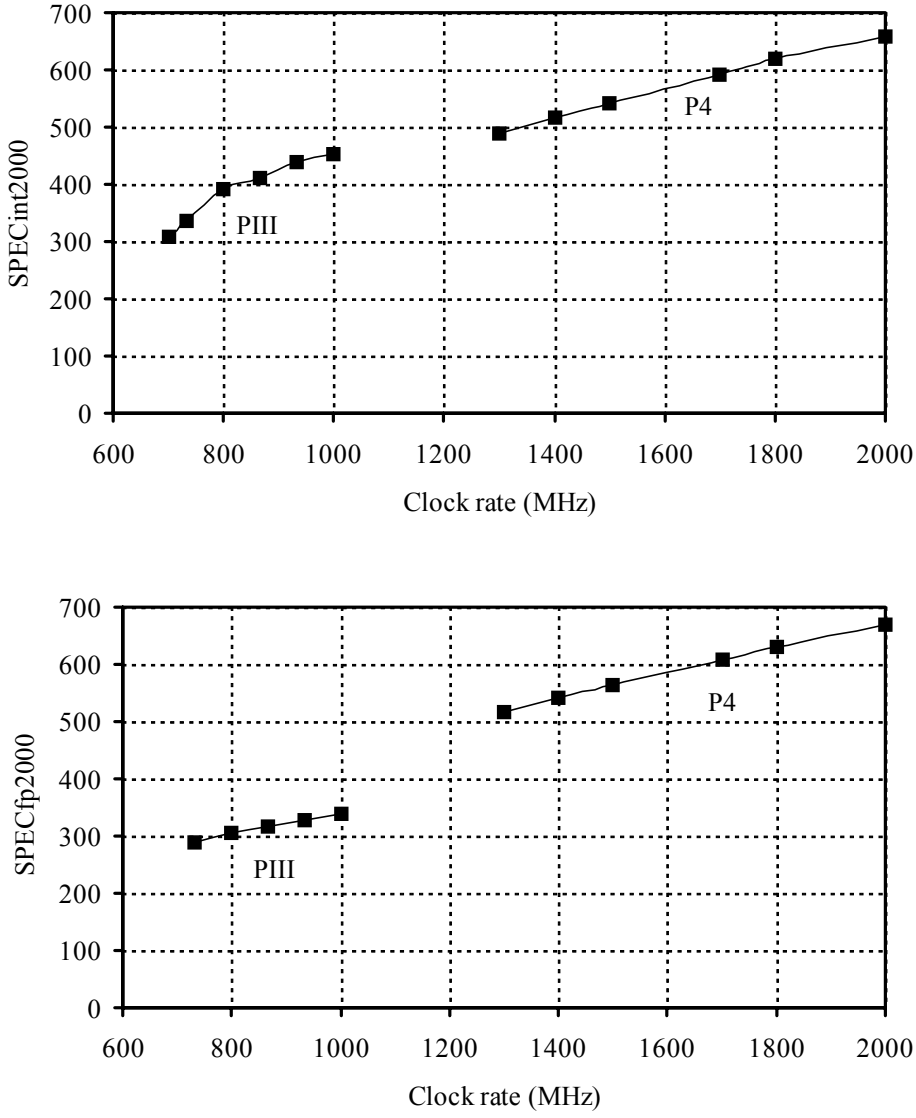

Figure 6.20 SPEC CPU2000 scores for Pentium III and 4 processors.

SPECMail2001 also specifies a response time limit for various actions such as SMTP connect and POP delete, as shown in Table 6.10. The percentage compliance rating of MSR 2.8 is also shown in this table.

Table 6.10 SPECMail2000 results for Mirapoint MSR 2.8 mail server system

Function	Response time limit (seconds)	Required percentage compliance (%)	Percentage Compliance (%) 80%	100%	120%
SMTP Connect	5	> 95	100.00	100.00	98.02
SMTP Data	5	> 95	100.00	100.00	100.00
POP Connect	5	> 95	100.00	100.00	100.00
POP Status	5	> 95	100.00	100.00	100.00
POP Retrieve	5	> 95	100.00	100.00	100.00
POP Delete	5	> 95	100.00	100.00	100.00
Delivery Time	60	> 95	99.14	99.01	95.60
Error Rate	N/A	< 1	0.15	0.14	1.13

SPECweb99

SPECweb99 is a benchmark used to measure the performance of HTTP servers. It measures a server's ability to handle HTTP GET requests from a number of external client drivers. The metric used is the number of simultaneous connections that conform to the specified bit rate limits. Each test is repeated three times for a reported result. The SPECweb99 metric is the median result for the three iterations.

As an example of the SPECweb99 benchmark, we present the values for the Sun Fire 4810 that runs the iPlanet Web Server 6.0. This server, which uses 12 750 MHz UltraSPARC III with 96 KB of primary cache (32 KB instruction cache and 64 KB data cache) and 8 MB of secondary cache, has a SPECweb99 rating of 8739 simultaneous connections. It uses 12 gigabit Ethernets to support the Web activities. Table 6.11 gives the three iteration results for the throughput and response time.

SPECjvm98

SPECjvm98 is the Java Virtual Machine benchmark suite that allows users to evaluate performance of the JVM client platform. This benchmark evaluates performance of both hardware and software components. It measures the efficiency of software components such as the JVM and the just-in-time (JIT) compiler. It also takes into account hardware components including the performance of the CPU for integer and floating-point operations, cache, and memory.

The SPECjvm98 benchmark suite consists of eight different applications. Five of these are either real applications or derived from real applications. Each test measures the time it takes to

Table 6.11 Sun Fire SPECWeb99 results

Iteration	Conforming connections	Conformance (%)	Throughput (operations/sec)	Response time (msec)
1	8749	100.0	24414.4	358.2
2	8739	99.9	24188.4	361.5
3	8739	99.9	24184.7	361.6

load the program, verify the class files, compile on the fly if a JIT compiler is used, and execute the test. Each test is run several times and a geometric mean is used to compute a composite score for all tests. Test scores are normalized against a reference machine: a midrange IBM PowerPC 604 with a 133 MHz processor.

6.7 Summary

When designing a processor, several design choices will have to be made. These choices are dictated by the available technology as well as the requirements of the target user group. Processor designers will have to make compromises in order to come up with the best design. This chapter looked at some of the important design issues involved in such an endeavor. Other design issues are covered in the rest of the book.

Here we looked at how the processor design at the ISA level gets affected by various design choices. We stated that the number of addresses in an instruction is one of the choices that can have an impact on the instruction set design. It is possible to have zero-, one-, two-, or three-address instruction sets; however, most recent processors use the three-address format. The Pentium, on the other hand, uses the two-address format.

The addressing mode is another characteristic that affects the instruction set. RISC processors tend to use the load/store architecture and use simple addressing modes. Often, these processors support just two addressing modes. CISC processors such as the Pentium provide a wide variety of addressing modes.

Both of these choices—number of addresses and the complexity of addressing modes—affect the instruction format. RISC processors use fixed-length instructions because they use the load/store architecture and support simple addressing modes. CISC processors use variable-length instructions to accommodate various complex addressing modes.

We also looked at how the instructions are executed in the underlying hardware. The hardware consists of a datapath with one, two, or three internal buses. We have seen the tradeoffs associated with the three types of datapaths. For simple instruction sets, typically used by RISC processors, necessary control signals for the datapath can be generated by the hardware. For complex instruction sets used by CISC processors, a software-based approach called microprogram control is used. We have discussed in detail how the microprogrammed control works.

In the last section, we covered processor performance. We introduced the concept of clocks per instruction and how it can be used to estimate the execution time of a program. We provided information on quantifying the performance of processors. Synthetic benchmarks tend to be exploited to produce skewed performance results. The recent trend is to use real application-based benchmarks to evaluate performance. Furthermore, benchmarks are specialized to the target application. For example, there is a benchmark for mail servers, another for web servers, and so on. To give you a concrete idea, we have presented several example benchmarks proposed by the SPEC consortium.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- 0-address machines
- 1-address machines
- 2-address machines
- 3-address machines
- Absolute address
- Accumulator machines
- Addressing modes
- Arithmetic instructions
- Conditional branch
- Data movement instructions
- Delayed procedure call
- End of procedure
- Flow control instructions
- Immediate addressing mode
- Input/output instructions
- Instruction format
- Instruction set design issues
- Instruction types
- Isolated I/O
- Load/store architecture
- Load/store instructions
- Logical instructions
- Memory-mapped I/O
- Microcontroller
- Microprogrammed control
- Number of addresses
- Opcode
- Operand types
- Parameter passing
- PC-relative
- Procedure call
- Processor registers
- Register addressing mode
- Return address
- Stack depth
- Stack machines
- Unconditional branch
- Wait cycles

6.8 Exercises

6–1 We have discussed instructions with zero to three addresses. Discuss why modern RISC processors use the three-address format.

6–2 The Pentium processor uses the two-address format. Do you support the decision made by the Pentium designers in this regard? Justify your answer.

6–3 Discuss the advantages of the load/store architecture. Focus your discussion on why current RISC processors use this architecture.

6–4 In Section 6.2.5, we have stated that 19 memory accesses are required to execute the example expression under the assumption that the push and pop operations do not require any memory accesses. Calculate the number of memory references required if the stack depth is zero (i.e., all push/pop operations require memory access).

6–5 RISC processors tend to have a large number of registers compared to CISC processors. Explain why.

6–6 What is the difference between normal and delayed branch execution? Why do some processors use delayed branch execution?

6–7 Conditional branching can be done in one of two basic ways: *set-then-jump* or *test-and-jump*. Discuss the advantages and disadvantages of these two methods.

6–8 During a procedure invocation, the return address must be saved in order to send the control back to the calling program. Most RISC processors store the return address in a register, whereas the Pentium uses the stack to store the return address. Discuss the pros and cons of these two schemes.

6–9 Explain why RISC processors tend to use fixed-length instructions whereas the CISC processors such as the Pentium do not.

6–10 We stated that the Pentium does not allow both operands to be located in memory. Explain the rationale for this.

6–11 In the single-bus datapath shown in Figure 6.7 (page 220), both the PC and IR registers are connected to the system bus. This allows the PC register to place the address on the system bus and the IR register to receive the instruction from the system bus. Suppose that these two registers are connected only to the A bus as are the general-purpose registers. Describe the steps involved in placing the PC address on the system bus. Also explain how the IR register will receive the instruction. What impact would this modification have on the processor performance?

6–12 Suppose we want to implement the instruction

```
shl4    %G7,%G5
```

on the single-bus datapath shown in Figure 6.7. This instruction shifts the contents of `G5` by four bit positions and stores the result in `G7`. Show how this instruction is implemented using the table format we used for the `add` instruction.

6–13 Our example ALU does not have a multiply by 10 function. Show how we can implement the following instruction:

```
mul10   %G7,%G5
```

This instruction multiplies the contents of `G5` by 10 and places the result in `G7`. Assume that there will be no overflow. How many cycles do you need to implement this instruction?

6–14 On the 2-bus datapath shown in Figure 6.19 (page 235) implement the data movement instruction

```
mov     %G7,%G5
```

to copy contents of `G5` to `G7`.

6–15 What is wrong with performance metrics like MIPS? What are the circumstances in which they are useful to compare the performance of processors?

6–16 What are real and synthetic benchmarks? Why is there a preference for the use of real benchmarks in current standards?

6–17 What are the major problems with synthetic benchmarks such as Whetstones?

6–18 What is the need for having so many specialized benchmarks such as SPECmail2001 and SPECweb99?