CHAPTER 3

Processor Basics

This chapter considers the overall design of instruction-set processors as exemplified by the central processing unit (CPU) of a computer. The fundamentals of CPU organization and operation are examined, along with the selection and formats of instruction and data types. Various representative microprocessors of both the RISC and CISC types are presented and discussed.

3.1 CPU ORGANIZATION

We begin by considering the organization of the central processor (microprocessor) of a computer and the methods used to represent the information it is intended to process.

3.1.1 Fundamentals

The primary function of the CPU and other instruction-set processors is to execute sequences of instructions, that is, programs, which are stored in an external main memory. Program execution is therefore carried out as follows:

1. The CPU transfers instructions and, when necessary, their input data (operands) from main memory to registers in the CPU.
2. The CPU executes the instructions in their stored sequence except when the execution sequence is explicitly altered by a branch instruction.
3. When necessary, the CPU transfers output data (results) from the CPU registers to main memory.

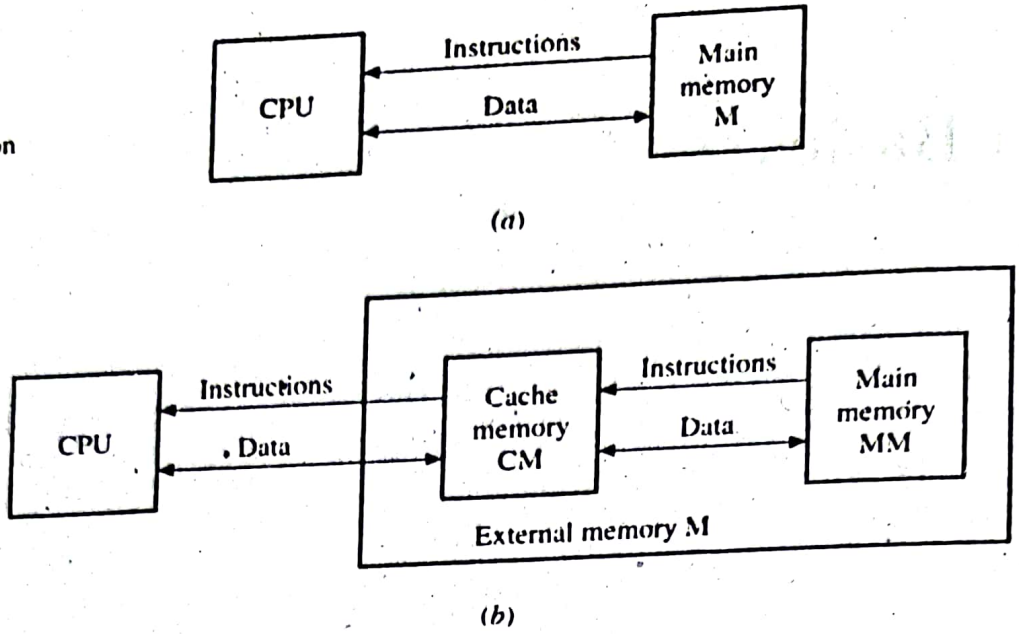


Figure 3.1
Processor-memory communication: (*a*) without a cache and (*b*) with a cache.

Consequently, streams of instructions and data flow between the external memory and the set of registers that forms the CPU's internal memory. The efficient management of these instruction and data streams is a basic function of the CPU.

External communication. If, as in Figure 3.1*a*, no cache memory is present, the CPU communicates directly with the main memory M, which is typically a high-capacity multichip random-access memory (RAM). The CPU is significantly faster than M; that is, it can read from or write to the CPU's registers perhaps 5 to 10 times faster than it can read from or write to M. VLSI technology, especially the single-chip microprocessor, has tended to increase the processor/main-memory speed disparity.

To remedy this situation, many computers have a cache memory CM positioned between the CPU and main memory. The cache CM is smaller and faster than main memory and may reside, wholly or in part, on the same chip as the CPU. It typically permits the CPU to perform a memory load or store operation in a single clock cycle, whereas a memory access that bypasses the cache and is handled by main memory takes many clock cycles. The cache is designed to be transparent to the CPU's instructions, which "see" the cache and main memory as forming a single, seamless memory space consisting of 2^m addressable storage locations M(0), M(1), ..., M(2^m–1). In this chapter we will take this viewpoint and use M to refer to the *external memory*, whether or not a cache is present. A specific memory location in M with address *adr* is referred to as M(*adr*) or simply as *adr*. When necessary, we will use MM to distinguish the main memory from the cache memory CM, as in Figure 3.1*b*. The structure of caches and their interactions with main memory are further studied in Chapter 6.

The CPU communicates with IO devices in much the same way as it communicates with external memory. The IO devices are associated with addressable registers called *IO ports* to which the CPU can store a word (an output operation) or from which it can load a word (an input operation). In some computers there are no IO

instructions per se; all IO data transfers are implemented by memory-referencing instructions, an approach called *memory-mapped IO*. This approach requires that memory locations and IO ports share the same set of addresses, so an address bit pattern that is assigned to memory cannot also be assigned to an IO port, and vice versa. Other computers employ IO instructions that are distinct from memory-referencing instructions. These instructions produce control signals to which IO ports, but not memory locations, respond. This second approach is sometimes called *IO-mapped IO*.

User and supervisor modes. The programs executed by a general-purpose computer fall into two broad groups: user programs and supervisor programs. A *user* or *application program* handles a specific application, such as word processing, of interest to the computer's users. A *supervisor program*, on the other hand, manages various routine aspects of the computer system on behalf of its users; it is typically part of the computer's operating system. Examples of supervisory functions are controlling a graphics interface and transferring data between secondary and main memory. In normal operation the CPU continually switches back and forth between user and supervisor programs. For example, while executing a user program, the need often arises for information that is available only on some hard disk unit in the computer's IO system. This condition causes the supervisor to temporarily suspend execution of the user program, execute a routine that initiates the required IO data-transfer operation, and then resume execution of the user program.

It is generally useful to design a CPU so that it can receive requests for supervisor services directly from secondary memory units and other IO devices. Such a request is called an *interrupt*. In the event of an interrupt, the CPU suspends execution of the program that it is currently executing and transfers to an appropriate interrupt-handling program. As interrupts, particularly from IO devices, require a rapid response from the CPU, it checks frequently for the presence of interrupt requests.

CPU operation. The flowchart in Figure 3.2 summarizes the main functions of a CPU. The sequence of operations performed by the CPU in processing an instruction constitutes an *instruction cycle*. While the details of the instruction cycle vary with the type of instruction, all instructions require two major steps: a *fetch step* during which a new instruction is read from the external memory M and an *execute step* during which the operations specified by the instruction are executed. A check for pending interrupt requests is also usually included in the instruction cycle, as shown in Figure 3.2.

The actions of the CPU during an instruction cycle are defined by a sequence of microoperations, each of which typically involves a register-transfer operation. The time required for the shortest well-defined CPU microoperation is the *CPU cycle time* or *clock period* T_{clock} and is a basic unit of time for measuring CPU actions. Recall that f, the CPU's clock frequency (in MHz) is related to T_{clock} (in μs) by $T_{clock} = 1/f$. As we will see, the number of CPU cycles required to process an instruction varies with the instruction type and the extent to which the processing of individual instructions can be overlapped. For the moment we will assume that each instruction is fetched from M in one CPU clock cycle (this is usually true when M is a cache) and can be executed in another CPU cycle.

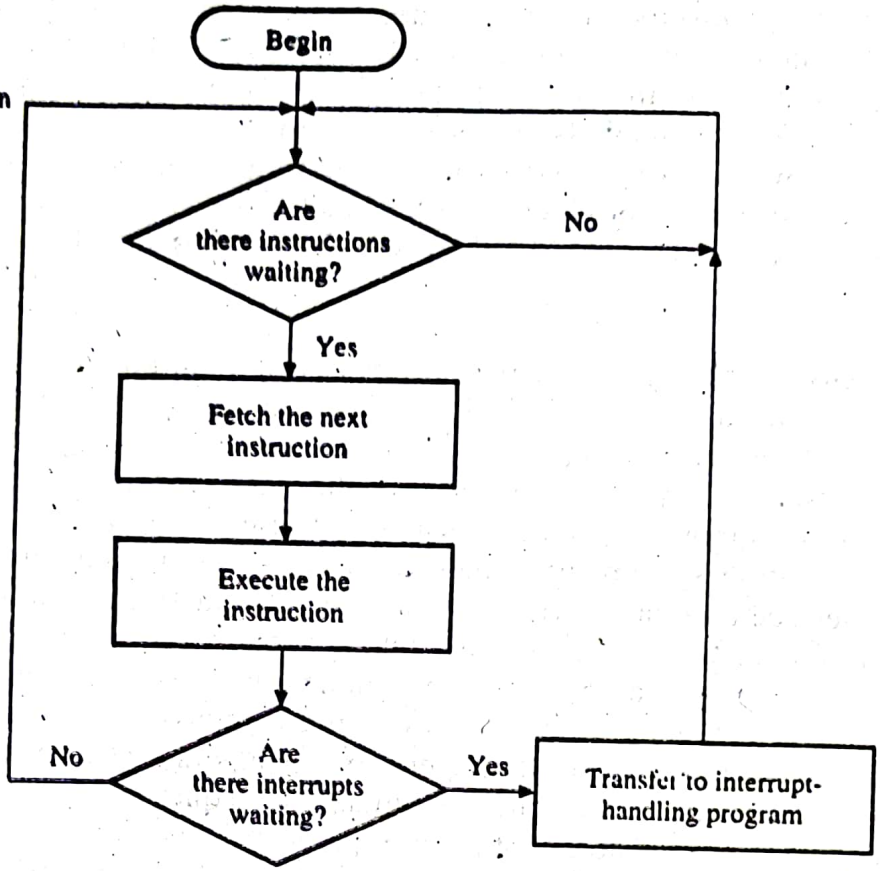


Figure 3.2
Overview of CPU behavior.

Accumulator-based CPU. Despite the improvements in IC technology over the years, CPU design continues to be based on the premise that the CPU should be as fast as the available technology and overall design requirements allow. Since cost generally increases with circuit complexity, the number of components in the CPU must be kept relatively small. The CPU organization proposed by von Neumann and his colleagues for the IAS computer (section 1.2.2) is the basis for most subsequent designs. It comprises a small set of registers and the circuits needed to execute a functionally complete set of instructions. In many early designs, one of the CPU registers, the *accumulator*,[1] played a central role, being used to store an input or output operand (result) in the execution of many instructions.

Figure 3.3 shows at the register level the essential structure of a small accumulator-oriented CPU. This organization is typical of first-generation computers (compare Figure 1.12) and low-cost microcontrollers. Assume for simplicity that instructions and data have some fixed word size n bits and that instructions can be adequately expressed by means of register-transfer operations in our HDL. Instructions are fetched by the program control unit PCU, whose main register is the pro-

[1]The term *accumulator* originally meant a device that combined the functions of number storage and addition. Any quantity transferred to an accumulator was automatically added to its previous contents. *Accumulator* is still often used in this restricted sense.

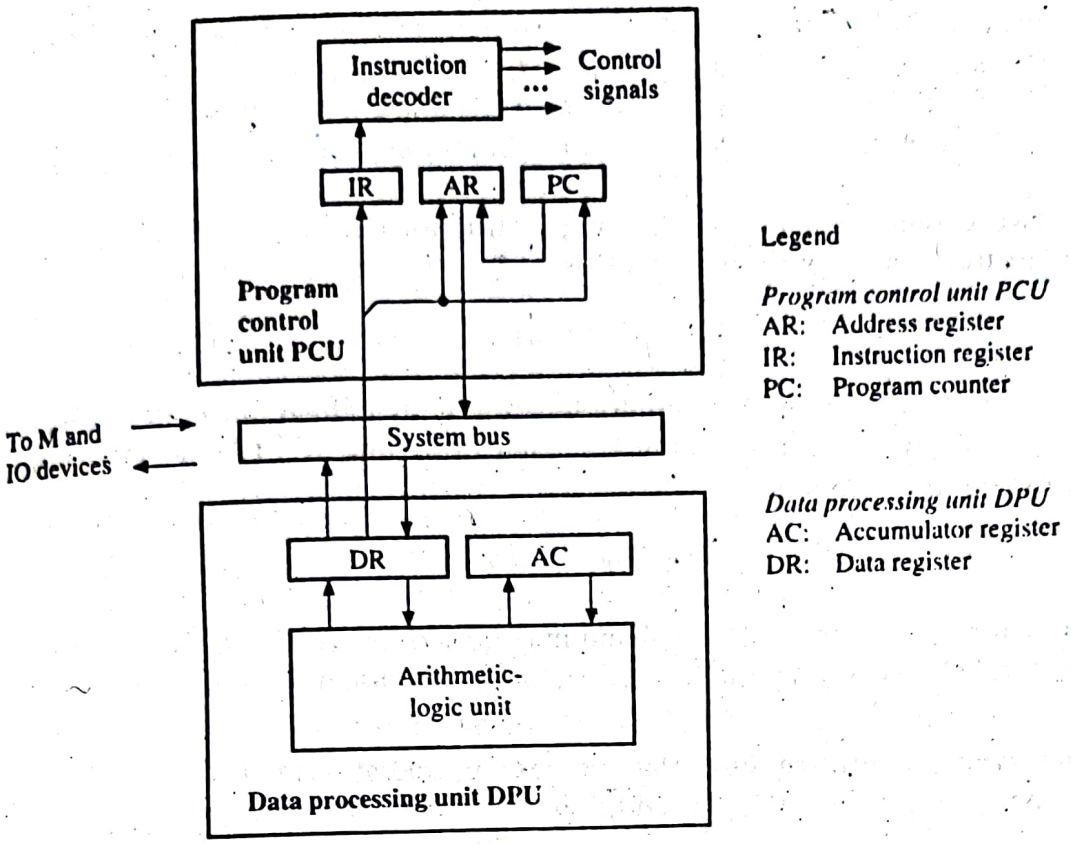


Figure 3.3
A small accumulator-based CPU.

gram counter PC. They are executed in the data processing unit DPU, which contains an n-bit arithmetic-logic unit (ALU) and two data registers AC and DR. Most instructions perform operations of the form

$$X1 := f_i(X1, X2)$$

where X1 and X2 denote a CPU register (AC, DR, or PC) or an external memory location M(*adr*). The operations f_i performed by the ALU are limited to fixed-point (integer) addition and subtraction, shifting, and logical (word-gate) operations.

Some instructions have an operand in an external memory location M(*adr*), and must therefore include the address part *adr*. Memory addresses are stored in two address registers in the PCU: the program counter PC, which stores instruction addresses only, and the general-purpose (data) address register AR. An instruction *I* that refers to a data word in M contains two parts, an opcode *op* and a memory address *adr*, and may be written as $I = op.adr$. Each instruction cycle begins with the instruction fetch operation

$$IR.AR := M(PC); \qquad (3.1)$$

which transfers the instruction word *I* from M to the CPU. The opcode *op* is loaded into the PCU's instruction register IR, and the address *adr* is loaded into address register AR. Hence (3.1) is equivalent to

$$IR := op, AR := adr;$$

Instructions that do not reference M do not use AR; their opcode part specifies the CPU registers to use, as well as the operation f_i to be carried out. Once it has placed the opcode of I in IR, the CPU proceeds to decode and execute it. Note that, at this point, the CPU can increment PC in order to obtain the address of the next instruction.

The two essential memory-addressing instructions are called load and store. The *load* instruction for our sample CPU is

$$AC := M(adr);$$

which transfers a word from the memory location with address *adr* to the accumulator. It is often written in assembly-language programs as LD *adr*. The corresponding *store* instruction is

$$M(adr) := AC;$$

which transfers a word from AC to M and may be written as ST *adr*. Note how the accumulator AC serves as an implicit source or destination register for data words.

Programming considerations. Data-processing operations normally require up to three operands. For example, the addition

$$Z := X + Y \tag{3.2}$$

has three distinct operands X, Y, and Z. The accumulator-based CPU of Figure 3.3 supports only *single-address* instructions, that is, instructions with one explicit memory address. However, AC and DR can serve as *implicit* operand locations so that multioperand operations can be implemented by executing several instructions in sequence. For example, a program to implement (3.2), assuming that X, Y, and Z all refer to data words in M, can take the following form:

HDL format	Assembly-language format	Narrative format (comment)
AC := M(X);	LD X	Load X from M into accumulator AC.
DR := AC;	MOV DR, AC	Move contents of AC to DR.
AC := M(Y);	LD Y	Load Y into accumulator AC.
AC := AC + DR;	ADD	Add DR to AC.
M(Z) := AC;	ST Z	Store contents of AC in M.

The preceding program fragment uses only the load and store instructions to access memory, a feature called *load/store architecture*. It is common (but as we will see, not always desirable) to allow other instructions to specify operands in memory. A CPU like that of Figure 3.3 can be designed to implement memory-referencing instructions of the form

$$AC := f_i(AC, M(adr))$$

whose execution requires two steps: one to move M(*adr*) to or from DR and one to perform the designated operation f_i. With an add instruction of this form, we can reduce the foregoing program from five to three instructions.

HDL Format	Assembly-language format	Narrative format (comment)
AC := M(*X*);	LD *X*	Load *X* from M into accumulator AC.
AC := AC + M(*Y*);	ADD *Y*	Load *Y* into DR and add to AC.
M(*Z*) := AC;	ST *Z*	Store contents of AC in M.

The memory-referencing ADD *Y* instruction can be expected to take longer to execute than the original ADD instruction that references only CPU registers. Memory references also complicate the instruction-decoding logic in the PCU. However, overall execution time should be reduced because we have eliminated an LD and a MOV instruction completely. As we will see later, the cost-performance impact of replacing a simple instruction with a more complex one has subtle implications that lie at the heart of the RISC–CISC debate.

Instruction set. Figure 3.4 gives a possible instruction set for our simple accumulator-based CPU, assuming a load/store architecture. These 10 instructions have the flavor of the instruction sets of some recent RISC machines, which demonstrate that small instruction sets can be both complete and efficient. We are, however, ignoring some important practical implementation issues in the interest of simplicity. We have not, for instance, specified the precise instruction or data formats to be used, and we do not consider such problems as numerical overflow—this condition occurs when an arithmetic instruction produces a result that is too big to fit in its destination register.

Type	Instruction	HDL format	Assembly-language format	Narrative format (comment)
Data transfer	Load	AC := M(*X*)	LD X	Load *X* from M into AC.
	Store	M(*X*) := AC	ST X	Store contents of AC in M as *X*.
	Move register	DR := AC	MOV DR, AC	Copy contents of AC to DR.
	Move register	AC := DR	MOV AC, DR	Copy contents of DR to AC.
Data processing	Add	AC := AC + DR	ADD	Add DR to AC.
	Subtract	AC := AC – DR	SUB	Subtract DR from AC.
	And	AC := AC *and* DR	AND	And bitwise DR to AC.
	Not	AC := *not* AC	NOT	Complement contents of AC
Program control	Branch	PC := M(*adr*)	BRA adr	Jump to instruction with address *adr*.
	Branch zero	if AC = 0 then PC := M(*adr*)	BZ adr	Jump to instruction *adr* if AC = 0.

Figure 3.4
Instruction set for the CPU of Figure 3.3.

The load and store instructions obviously suffice for transferring data between the CPU and main memory. We know from Boolean algebra that the AND and NOT operations are functionally complete, implying that the instruction set enables any logical operation to be programmed. We also know that addition and subtraction suffice for implementing most arithmetic operations. Consider, for example, the arithmetic operation negation, for which many CPUs have a single instruction of the type AC := –AC. We can easily implement negation by a three-instruction sequence as follows:

HDL format	Assembly-language format	Narrative format (comment)
DR := AC;	MOV DR, AC	Copy contents X of AC to DR.
AC := AC – DR;	SUB	Compute AC = $X - X = 0$.
AC := AC – DR;	SUB	Compute AC = $0 - X = -X$.

Figure 3.4 also gives a small set of program control instructions: an unconditional branch instruction BRA and a conditional branch-on-zero instruction BZ that tests the contents of AC. Observe that these instructions load a new address into the program counter PC, thus altering the instruction execution sequence. The BZ instruction allows more powerful program control operations such as procedure call and return to be implemented; it also facilitates complex operations such as multiplication, as we demonstrate in Example 3.1.

EXAMPLE 3.1 A MULTIPLICATION PROGRAM. Suppose we want to use the tiny instruction set of Figure 3.4 to program the multiplication operation

$$AC := AC \times N$$

where the multiplicand is the initial contents of the accumulator AC and the multiplier N is a variable stored in memory. We will assume that the multiplier and multiplicand are both unsigned numbers and that they are sufficiently small that the product will fit in a single word. We can construct the desired program along the following lines. We will execute the basic ADD instruction N times to implement $AC \times N$ in the form AC + AC + ... + AC. We will treat the memory location storing N as a count register and, after each addition step, decrement it by one until it reaches zero. We will test for $N = 0$ by means of the BZ instruction, and so we will have to transfer N to AC in order to perform this test. We will also have to use some memory locations as temporary registers for storing intermediate results and some other quantities, such as the initial value Y of AC. In particular, we will use memory locations *one*, *mult*, *ac*, and *prod* to store the constant 1, N, Y, and the partial product P, respectively. Here *one*, *mult*, *ac*, and *prod* are symbolic names for certain memory addresses that we have arbitrarily assigned. They are translated into numerical memory addresses by an assembler program prior to execution.

An assembly-language program implementing this plan appears in Figure 3.5. Its main body (lines 5 to 17) is traversed N times in the course of a multiplication. At the end the result P is in memory location *prod*. The first two instructions (lines 5 and 6) of the program check the value of N by reading it into AC and testing it with the BZ instruction. If the initial value of N is zero, the program exits immediately with the correct result $P = 0$. If N is nonzero, the instructions in lines 7 to 11 load it from *mult* into AC, subtract one from it, and then return the new, decremented value of N to *mult*. The

Line	Location	Instruction or data	Comment
0	*one*	00 . . . 001	The constant one.
1	*mult*	*N*	The multiplier.
2	*ac*	00 . . . 000	Location for initial value *Y* of AC.
3	*prod*	00 . . . 000	Location for (partial) product *P*.
4		ST *ac*	Save initial value *Y* of AC.
5	*loop*	LD *mult*	Load *N* into AC to test for termination.
6		BZ *exit*	Exit if *N* = 0; otherwise continue.
7		LD *one*	Load 1 into AC.
8		MOV DR, AC	Move 1 from AC to DR.
9		LD *mult*	Load *N* into AC to decrement it.
10		SUB	Subtract 1 from *N*.
11		ST *mult*	Store decremented *N*.
12		LD *ac*	Load initial value *Y* of AC.
13		MOV DR, AC	Move *Y* from AC to DR.
14		LD *prod*	Load current partial product *P*.
15		ADD	Add *Y* to *P*.
16		ST *prod*	Store the new partial product *P*.
17		BRA *loop*	Branch to *loop*.
18	*exit*	. . .	

Figure 3.5
A program for the multiplication operation AC := AC × *N*.

main step of adding *Y* to the accumulating partial product, that is, $P := P + Y$, is implemented in straightforward fashion by lines 12 to 16 of the program. Finally, a return is made to *loop* via the unconditional branch BRA (line 17).

This program uses most of the available instruction types and illustrates several weaknesses of an accumulator-based CPU. Because there are only a few data registers in the CPU, a considerable amount of time is spent shuttling the same information back and forth between the CPU and memory. Indeed, most of the instructions in this program are of the data-transfer type (ST, LD, and MOV), which do bookkeeping for the few instructions that actually compute the product *P*. It would both shorten the program and speed up its execution if we could store the quantities 1, *N*, *Y*, and *P* in their own CPU registers, as they are repeatedly required by the CPU.

Program execution. We now examine the execution process for the multiplication program of Figure 3.5. Of course, the program must be translated into executable object code prior to execution, but we can treat the assembly-language program as a symbolic representation of the object code. Recall that we are assuming that every instruction is one word long and can be fetched from M in a single CPU clock cycle. We further assume that every instruction is also executed in a single clock cycle. Hence each instruction requires two CPU clock cycles—one to fetch the instruction from M and one to execute it. At the end of

Clock cycle	Instruction cycle	PC	AR	PCU actions	DPU actions
1	ST *ac*	1004		IR.AR := M(PC), PC := PC + 1	
2			1002		M(AR) := AC
3	LD *mult*	1005		IR.AR := M(PC), PC := PC + 1	
4			1001		AC := M(AR)
5	BZ *exit*	1006		IR.AR := M(PC), PC := PC + 1	
6			1001	Test A; no further action if A ≠ 0	None
7	LD *one*	1007		IR.AR := M(PC), PC := PC + 1	
8			1000		AC := M(AR)
9	MOV DR, AC	1008		IR.AR := M(PC), PC := PC + 1	
10			*dddd*		DR := AC
11	LD *mult*	1009		IR.AR := M(PC), PC := PC + 1	
12			1001		AC := M(AR)
13	SUB	1010		IR.AR := M(PC), PC := PC + 1	
14			*dddd*		AC := AC – DR
15	ST *mult*	1011		IR.AR := M(PC), PC := PC + 1	
16			1001		M(AR) := AC
17	LD *ac*	1012		IR.AR := M(PC), PC := PC + 1	
18			1002		AC := M(AR)
19	MOV DR, AC	1013		IR.AR := M(PC), PC := PC + 1	
20			*dddd*		DR := AC
21	LD *prod*	1014		IR.AR := M(PC), PC := PC + 1	
22			1003		AC := M(AR)
23	ADD	1015		IR.AR := M(PC), PC := PC + 1	
24			*dddd*		AC := AC + DR
25	ST *prod*	1016		IR.AR := M(PC), PC := PC + 1	
26			1003		M(AR) := AC
27	BRA *loop*	1017		IR.AR := M(PC), PC := PC + 1	
28			1005	PC := AR	None
29	LD *mult*	1005		IR.AR := M(PC), PC := PC + 1	
30			1001		AC := M(AR)
31	BZ *exit*	1006		IR.AR := M(PC), PC := PC + 1	
32			1018	Test A: PC := AR if A = 0	None
33		1018		...	

Figure 3.6
Cycle-by-cycle execution trace of the multiplication program of Figure 3.5.

the fetch step, the PCU decodes the instruction's opcode to determine what operation to perform during the execution stage. It can also increment PC in preparation for the next instruction fetch. Recall that an edge-triggered register can be both read from and written into in the same clock cycle so that the new data is ready for use at the beginning of the next clock cycle. Hence every fetch cycle includes the following pair of register-transfer operations:

$$IR.AR := M(PC),\ PC := PC + 1 \qquad (3.3)$$

The subsequent execution cycle depends on the instruction opcode placed in IR.

Figure 3.6 depicts all the main actions taken by the CPU, including the memory addresses it generates, during execution of the program of Figure 3.5. Data of this type is referred to as an *execution trace* and is often obtained by simulation of the target CPU. (In effect, Figure 3.6 is a hand simulation of the multiplication program.) Execution traces are useful for analyzing program behavior and execution speed. In this example the program's data and instructions have been assigned to a consecutive sequence of memory locations 1000, 1001, 1002, . . . , where 1001 is the location named *one* in Figure 3.5. The first executable instruction is ST *ac*, which is in location 1004, so execution begins when PC is set to 1004. Observe how the contents of the program counter PC are incremented steadily until a branch instruction is encountered, at which point the branch address contained in the branch instruction may replace the incremented contents of PC.

3.1.2 Additional Features

Next we examine some more advanced features of CPUs and look at representative commercial microprocessors of the RISC and CISC types.

Architecture extensions. There are many ways in which the basic design of Figure 3.3 can be improved. Most recent CPUs contain the following extensions, which significantly improve their performance and ease of programming.

- *Multipurpose register set for storing data and addresses*: These replace the accumulator AC and the auxiliary registers DR and AR of our basic CPU. The resulting CPU is sometimes said to have the *general register organization* exemplified by the third-generation IBM System/360-370 (Figure 1.17), which has 32 such registers. The set of general registers is now usually referred to as a *register file*.
- *Additional data, instruction, and address types*: Most CPUs have instructions to handle data and addresses with several different word sizes and formats. Although some microprocessors have only add and subtract instructions in the arithmetic category, relatively little extra circuitry is required for (fixed-point) multiply and divide instructions, which simplify many programming tasks. Call and return instructions also simplify program design.
- *Register to indicate computation status*: A *status register* (also called a *condition code* or *flag* register) indicates infrequent or *exceptional conditions* resulting from the instruction execution. Examples are the appearance of an all-zero result or an invalid instruction like divide by zero. A status register can also indicate the user and supervisor states. Conditional branch instructions can test the status register, which simplifies the programming of conditional actions.

- *Program control stack*: Various special registers and instructions facilitate the transfer of control among programs due to procedure calling or external interrupts. Many CPUs use a flexible scheme for program-control transfer, which employs part of the external memory M as a push-down stack (see also Example 1.5). The stack memory is intended for saving key information about an interrupted program via push operations so that the saved information can be retrieved later via pop operations. A CPU address register called a *stack pointer* automatically keeps track of the stack's entry point.

Figure 3.7 shows the organization of a processor with the foregoing features. It has a register file in the DPU for data and/or address storage. The ALU obtains most of its operands from the register file and also stores most of its results there. A status register monitors the output of the ALU and other key points. The principal special-purpose address registers are the program counter and the stack pointer. Special circuits are included for address computation, although the main ALU can also be used for this purpose. The control circuits in the PCU derive their inputs from the instruction register, which stores the opcode of the current instruction, and

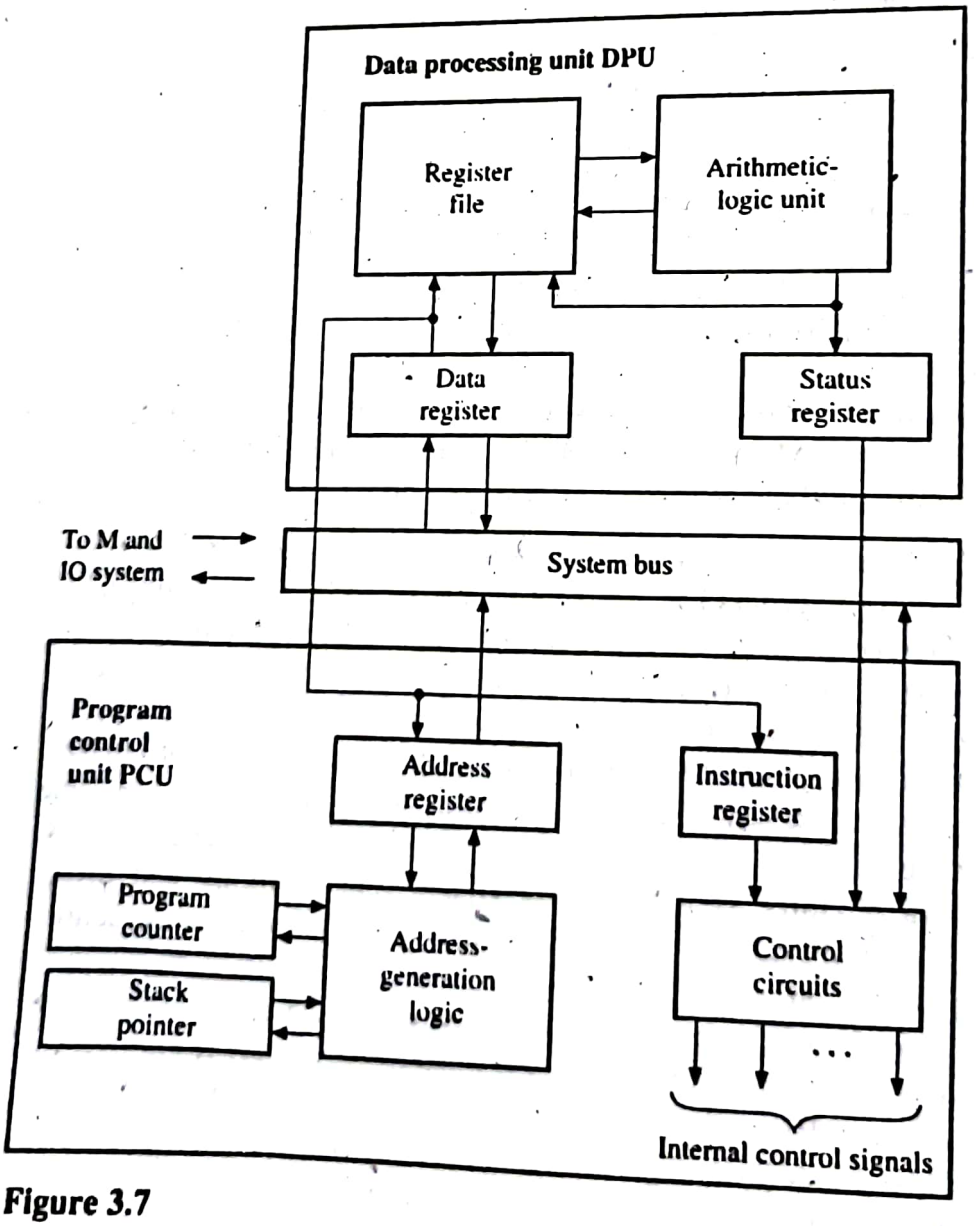


Figure 3.7
A typical CPU with the general register organization.

the status register. Communication with the outside world is via a system bus that transmits address, data, and control information among the CPU, M, and the IO system. Various nonprogrammable "buffer" registers serve as temporary storage points between the system bus and the CPU.

Pipelining. As discussed in Chapter 1, modern CPUs employ a variety of speedup techniques, including cache memories, and several forms of instruction-level parallelism. Such parallelism may be present in the internal organization of the DPU or in the overlapping of the operations carried out by the DPU and PCU. These features add to the CPU's complexity and will be explored in depth later in this book.

The considerable potential for parallel processing at the instruction level is evident even in the simple CPU of Figure 3.3. We see from the execution trace of Figure 3.6 that the main PCU and DPU activities take place in different clock cycles. If these activities do not share a resource such as the system bus, they can be carried out at the same time. In other words, while the current instruction is being executed in the DPU, the next instruction can be fetched by the PCU. For example, the three-instruction negation routine we gave earlier to change AC to –AC would be executed as follows in the style of Figure 3.6:

Clock cycle	Instruction cycle	PC	PCU actions	DPU actions
1	MOV DR, AC	2000	IR.AR := M(PC), PC := PC + 1	
2		2001		DR := AC
3	SUB	2001	IR.AR := M(PC), PC := PC + 1	
4		2002		AC := AC – DR
5	SUB	2002	IR.AR := M(PC), PC := PC + 1	
6		2003		AC := AC – DR

By merging the execution part of each instruction cycle with the fetch part of the following instruction cycle, we can reduce the overall execution time from six clock cycles to four, as shown below. (We use subscripts to distinguish the first and second SUB instructions.)

Clock cycle	Instruction cycle	PC	PCU actions	DPU actions
1	MOV	2000	IR.AR := M(PC), PC := PC + 1	
2	MOV/SUB_1	2001	IR.AR := M(PC), PC := PC + 1	DR := AC
3	SUB_1/SUB_2	2002	IR.AR := M(PC), PC := PC + 1	AC := AC – DR
4	SUB_2	2003		AC := AC – DR

This overlapping of instruction fetching and execution is an example of *instruction pipelining*, which is an important speedup feature of RISC processors. Figure 3.8 illustrates graphically the type of *two-stage* pipelining discussed above. Each instruction can be thought of as passing through two consecutive stages of

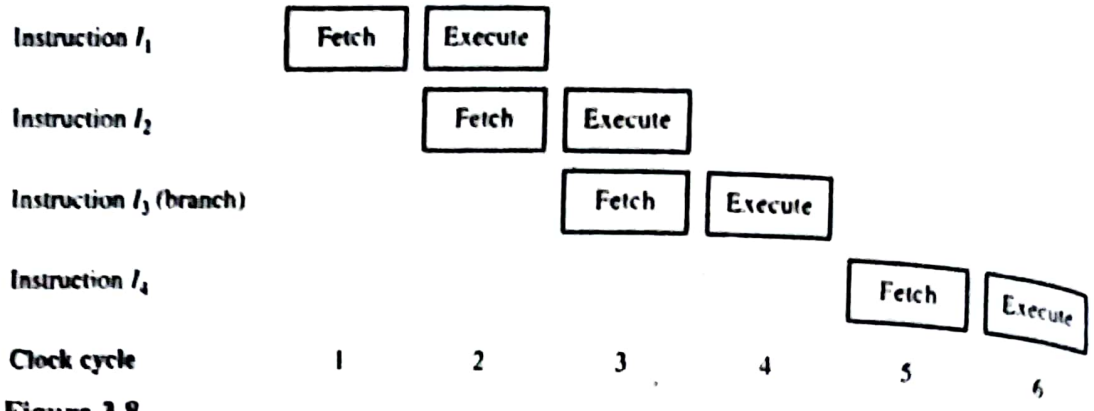


Figure 3.8
Overlapping instructions in a two-stage instruction pipeline.

processing: a fetch stage implemented mainly by the PCU and an execution stage implemented mainly by the DPU. Hence two instructions can be processed simultaneously in every CPU clock cycle, with one completing its fetch phase and the other completing its execute phase. A two-stage pipeline can therefore double the CPU's performance from one instruction every two clock cycles to one instruction every clock cycle.

A problem arises when a branch instruction is encountered, such as the BRA *loop* instruction stored in address (line) 17 of the multiplication program (Figure 3.5). Immediately before this instruction is fetched in some clock cycle i the program counter PC stores the address 17. PC is then incremented to 18 in preparation for clock cycle $i + 1$. Clearly in clock cycle $i + 1$, the CPU should *not* fetch the instruction stored at address 18—that instruction is not even in the multiplication program. In clock cycle $i + 1$, BRA is executed, which causes *loop* = 5 to be loaded into PC, implying that the next instruction should be taken from location 5. The fetching of this instruction can't begin until cycle $i + 2$, however, as illustrated in Figure 3.8 with $i = 4$. It follows that we cannot overlap the branch instruction and the instruction that follows it (I_3 and I_4 in the case of Figure 3.8).

Thus we see that branch instructions reduce the efficiency of instruction pipelining, although we will see later that steps can be taken to reduce this problem. We will also see that instruction processing is usually broken into more than two stages to increase the level of the parallelism attainable.

EXAMPLE 3.2 THE ARM6 MICROPROCESSOR [VAN SOMEREN AND ATACK 1994]. We now examine in some detail the architecture of a microprocessor family that embodies the RISC design philosophy in a relatively direct and elegant form. The ARM has its origins in the Acorn RISC Machine, a microprocessor developed in the United Kingdom in the 1980s to serve as the CPU of a personal computer. Subsequently, the family name was changed—without changing its acronym, however—to Advanced RISC Machine. The ARM family is primarily aimed at low-cost, low-power applications such as portable computers and games. For example, the Newton, a hand-held "personal digital assistant" introduced by Apple Corp. in 1993 employs the ARM6 microprocessor, whose main features are described below.

The ARM6 is a 32-bit processor in that both its data words and its address words are 32 bits (4 bytes) long. It has a load/store architecture, so only its load and store instructions can address external memory M. As in most computers since the IBM System/360, main memory is organized as an array of individually addressable bytes. Thus

the maximum memory size of an ARM6 computer is 2^{32} bytes, also referred to as 4 gigabytes (4G bytes). The ARM6 employs an instruction pipeline to meet the goal of one instruction executed per CPU clock cycle. Note that it shares all these features with a more powerful (and more expensive) RISC microprocessor, the PowerPC (Example 1.7). The ARM6's instruction set is much smaller than the PowerPC's, however—it has no floating-point instructions, for example.

The internal organization of the ARM's CPU is shown in Figure 3.9. It has a 32-bit ALU and a file of 32-bit general-purpose registers. To permit direct interaction between data and control registers, the ARM has the unusual feature of placing its PC and status registers in the register file; conceptually, we will continue to view these registers as part of the PCU. There are several modes of operation, including the normal user and supervisor modes, and four special modes associated with interrupt handling. In user mode the register file appears to contain sixteen 32-bit registers designated R0:R15, where R15 is also the program counter PC, as well as a current program status register designated CPSR. (Additional registers, which we will not discuss here, are used when the CPU is in other operating modes; they are "invisible" in user mode.) The ALU is designed to perform basic arithmetic operations on 32-bit integers. It employs combinational logic for addition and subtraction and a sequential shift-and-add method similar to that described in Example 2.7 for multiplication. A combinational shift circuit is attached to the ALU to support multiplication and other operations. A separate address-incrementer circuit implements address-manipulation operations such as PC := PC + 1 independently of the ALU. Access to external memory M (a cache or main memory) is straightforward. The address of the desired location in M is placed in the PCU's address register. In the case of a store instruction, the data to be stored is also placed in the DPU's write data register. A load instruction causes a data word to be fetched from memory and placed in the read data register. Several internal buses transfer data efficiently among the DPU's registers and data processing circuits.

All ARM6 instructions are 32 bits long, and they have a variety of formats and addressing modes. There are about 25 main instruction types, which are listed in Figure 3.10. (We have omitted block move and coprocessor instructions.) This number is deceptively small, however, as instructions have options that substantially increase the number of operations they can perform. Most instructions can be applied either to 32-bit operands (words) or to 8-bit operands (bytes). Operands and addresses are usually stored in registers that can be referred to by short, 4-bit names, allowing a single ARM6 instruction to specify as many as four operands. The available address space is shared between memory and IO devices (memory-mapped IO). Consequently, the load/store instructions used for CPU-memory transfers are also used for IO operations.

Any instruction can be conditionally executed, meaning that execution may or may not occur depending on the value of designated status bits (flags) in the CPSR. The status flags are set by a previous instruction and include a negative flag N (the previous result *R* computed by the ALU was a negative number), a zero flag Z (*R* was zero), a carry flag C (*R* generated an output carry), and an overflow flag V (*R* generated a sign overflow). Hence every ARM6 instruction is effectively combined with a conditional branch instruction. The basic unconditional move instruction MOV R0, R1 can have any of 15 conditions attached to it to determine if it is to be executed (see problem 3.8). Some examples:

```
MOVCC  R0, R1      ; If flag C = 0, then R0 := R1
MOVCS  R0, R1      ; If flag C = 1, then R0 := R1
MOVHI  R0, R1      ; If flag C = 1 and flag Z = 0, then R0 := R1
```

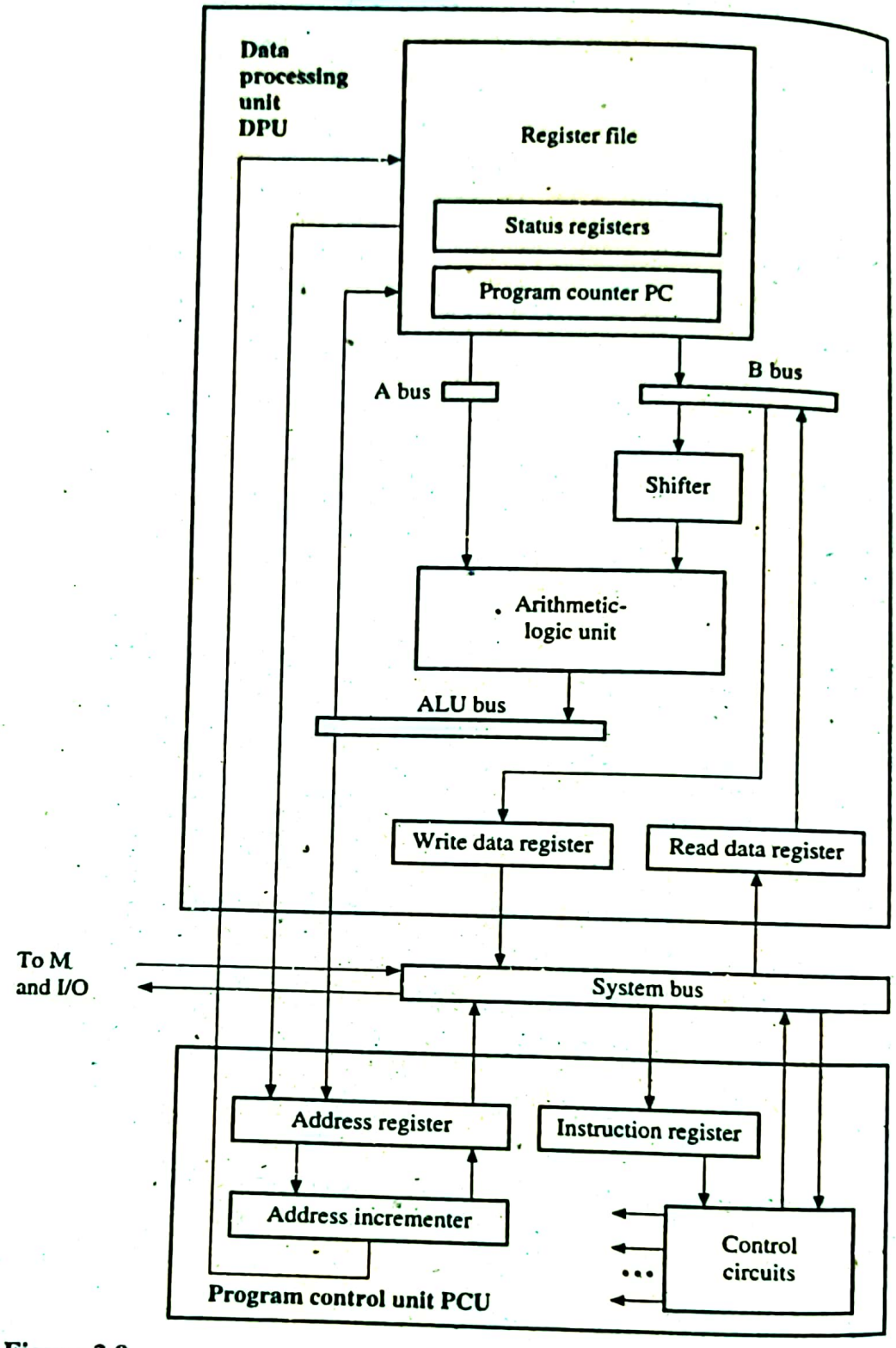


Figure 3.9
Overall organization of the ARM6.

An ARM6 instruction can also include a shift or rotation operation that is applied to one of its operands. For instance:

```
MOV     R0, R1, LSL #2          ; R0 := R1 × 4          (3.4)
```

means logically left shift (LSL) the contents of R1 by 2 bits and move the result to RO. This shift is tantamount to multiplying R1 by four before the move.

The opcode suffix S specifies whether or not an instruction affects the status flags. If S is present, appropriate flags are changed; otherwise, the flags are not affected. For example, the ARM6's move instructions affect the N, Z, and C flags, so appending S

Type	Instruction	HDL format	Assembly-language format	Narrative format (comment)
Data transfer	Move register	R3 := R9	MOV R3,R9	Copy contents of register R9 to register R3.
	Move register	R0 := 12	MOV R0,#12	Copy operand (decimal number 12) to register R0.
	Move inverted	R7 := $\overline{R0}$	MVN R7,R0	Copy bitwise inverted contents of R0 to R7
	Load	R5 := M(adr)	LDR R5, adr	Load R5 with contents of memory location adr.
	Store	M(adr) := R8	STR R8,adr	Store contents of R8 in memory location adr.
Data processing	Add	R3 := R5 + 25	ADD R3,R5,#25	Add 25 to R5; place sum in R3.
	Add with carry	R3 := R5 + R6 + C	ADC R3,R5,R6	Add R6 and carry bit C to R5; place sum in R3.
	Subtract	R3 := R5 – 9	SUB R3,R5,#9	Subtract 9 from R5; place difference in R3.
	Subtract with carry	R3 := R5 – 9 – C	SBC R3,R5,#9	Subtract 9 and borrow bit from R5; place difference in R3.
	Reverse subtract	R3 := 9 – R5	RSB R3,R5,#9	Subtract R5 from 9; place difference in R3.
	Reverse subtract with carry	R3 := 9 – R5 – C	RSC R3,R5,#9	Subtract R5 and borrow bit from 9; place difference in R3.
	Multiply	R1 := R3 × R2	MUL R1,R2,R3	Multiply R3 by R2; place result in R1.
	Multiply and add	R1 := (R3 × R2) + R4	MLA R1,R2,R3,R4	Multiply R3 by R2; add R4; place result in R1.
	And	R4 := R11 *and* 25_{16}	AND R4,R11,0x25	Bitwise AND R11 and 25_{16}; place result in R4.
	Or	R4 := R11 *or* 25_{16}	ORR R4,R11,0x25	Bitwise OR R11 and 25_{16}; place result in R4.
	Exclusive-or	R4 := R11 *xor* 25_{16}	EOR R4,R11,0x25	Bitwise XOR R11 and 25_{16}; place result in R4.
	Bit clear	R4 := R11 ∧ $\overline{25}_{16}$	BIC R4,R11,#25	Bitwise invert 25; AND it to R11; place result in R4.
Program control	Branch	PC := PC + *adr*	B *adr*	Jump to designated instruction.
	Branch and link	R14 := PC, PC := PC + *adr*	BL *adr*	Save old PC in "link" register R14; then jump to designated instruction.
	Software interrupt		SWI	Enter supervisor mode.
	Compare	Flags := R1 – 14	CMP r1,#14	Subtract 14 from R1 and set flags.
	Compare inverted	Flags := R1 + 14	CMN r1,#14	Add 14 to R1 and set flags.
	Logical compare	Flags := R1 *xor* 14	TEQ r1,#14	XOR 14 to R1 and set flags.
	Compare inverted	Flags := R1 *or* 14	TST r1,#14	AND 14 to R1 and set flags.

Figure 3.10
Core instruction set of the ARM6.

to, say, MOVCS, yields MOVCSS, which checks the moved data item D. It sets N = 1 (0) if D_{31} = 1 (0), it sets Z = 1 (0) if D is zero (nonzero), and it sets C to the shifter's output value.

Like other RISCs, the ARM6 has an instruction pipeline that permits the various stages of instruction processing to be overlapped. The pipeline has three stages: fetch, decode, and execute; in effect, the ARM6 breaks the first stage of the two-stage pipeline of Figure 3.8 in two. This structure permits the CPU to check every instruction's condition code in stage 2 to determine whether the instruction should be executed in stage 3. Some instructions such as multiply require more than one cycle for execution, but most require only one. Note that inclusion of an operand shift in an instruction as in (3.4) does not require an additional cycle, thanks to the fast (combinational) shifter.

A CISC machine. We turn next to a widely used CPU family, the Motorola 680X0 family, which was introduced in 1979 with the 68000 microprocessor. This example of an older CISC architecture is more streamlined and "RISC-like" than other CISCs. Later members of the family such as the 68060 [Circello et al. 1995] have speedup features such as instruction pipelining, floating-point execution units, and superscalar instruction issue. We examine an intermediate member of the series, the 68020, a 32-bit machine whose design broadly resembles that of a third-generation mainframe computer [Motorola 1989].

The 68020 is a one-chip microprocessor introduced in 1985 to serve as the CPU of a general-purpose computer such as a personal computer or workstation. Figure 3.11 outlines the organization of the 68020. It is designed to handle 32-bit words (termed *long* words in 680X0 literature) efficiently, but instructions are also provided to handle operands of 1, 8, 16, and 64 bits. As in the ARM6, memory addresses are 32 bits long, permitting a total of 2^{32} different memory locations, each storing 1 byte. Memory-mapped IO is also used in the 680X0 series. The data-processing unit has a register file containing sixteen 32-bit registers, half of which are data registers designated D0:D7 and half are address registers designated A0:A7. The ALU can execute a large set of fixed-point (but not floating-point) instructions. Instruction interpretation and other control functions of the CPU are implemented by a microprogrammed control unit.

The 68020 has about 70 distinct instruction types (or around 200 if all opcode variants are distinguished), which are summarized in Figure 3.12. A given instruction such as MOVE can be defined with several different types of operands, and the operands can be addressed in various ways. For example, the following move-register instruction written in 680X0 assembly-language format

```
MOVE.L   D1, A6                    (3.5)
```

causes the entire contents (a long word as indicated by the opcode suffix .L) of data register D1 to be copied to address register A6. In other words, (3.5) implements the register transfer A6 := D1. If .L is replaced by .B, then the resulting instruction

```
MOVE.B   D1, A6
```

causes only the byte stored in the low-order position (bits 0:7) of D1 to be copied to the corresponding part of A6.

Besides the *direct addressing* mode illustrated by the preceding example, the 68020 has several other addressing modes that give the programmer considerable

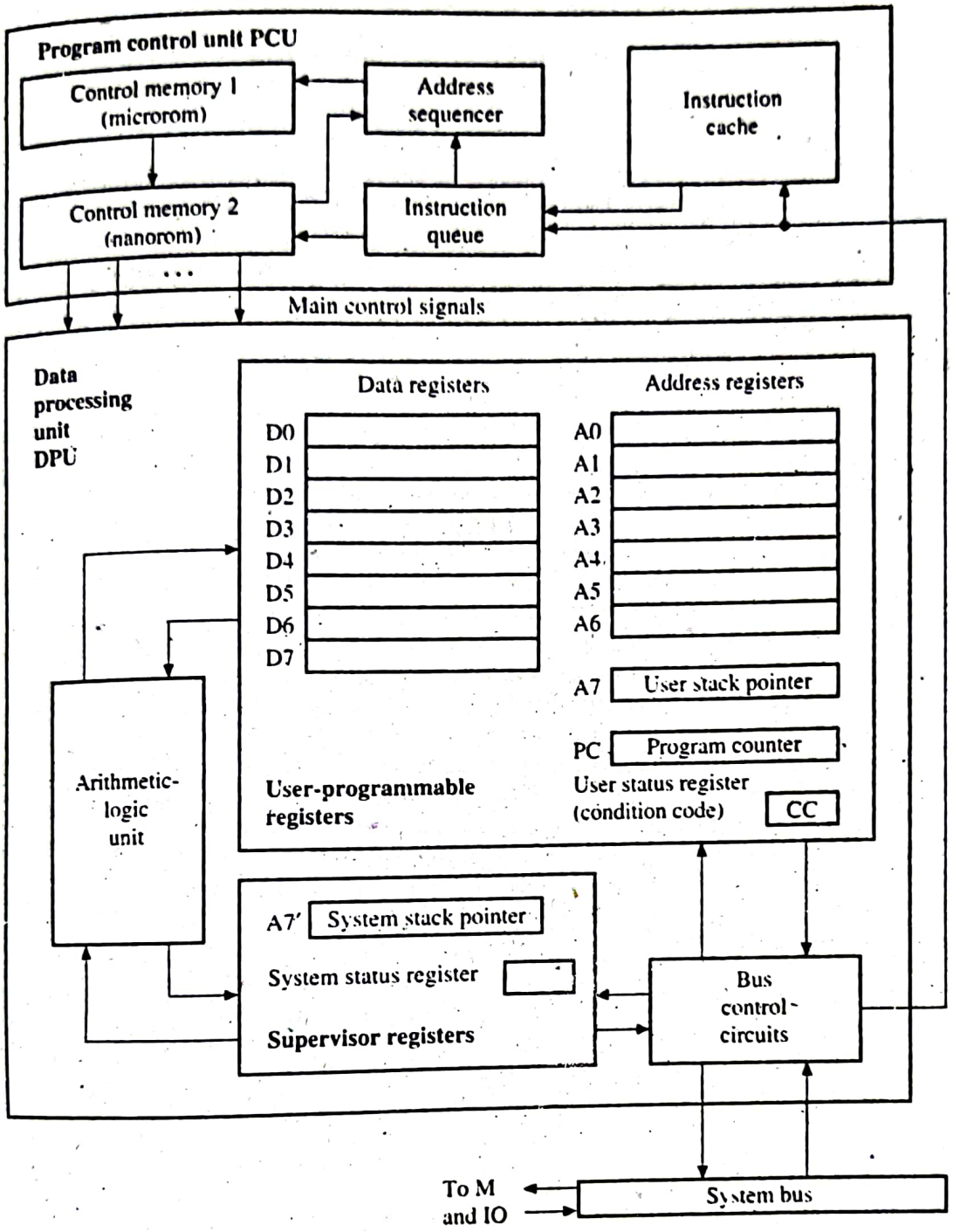


Figure 3.11
Organization of the 68020.

flexibility in accessing data. Most instructions can address memory as well as CPU registers. For example, if (3.5) is replaced by

```
MOVE.L   D1, (A6)                    (3.6)
```

the resulting operation is M(A6) := D1, that is, a store operation with A6 serving as the memory-address register. This is an instance of *indirect addressing*. Note that while (3.5) takes 4 clock cycles to execute, (3.6) takes 12 cycles because of the time required to access external memory. The 68020's data-processing instructions can also access M directly, so the 68020 does *not* have the load/store architecture

Type	Opcode	Description
Data transfer	EXG	Exchange (swap) contents of two registers.
	MOVE	Move (copy) data unchanged from source to destination in CPU or M
	MOVEA	Copy data to address register.
	MOVEC	Copy data to or from control register (privileged instruction).
	MOVEM	Copy multiple data items to or from specified list of registers.
	MOVEP	Copy data between register and alternate bytes of memory.
	MOVEQ	Copy "quick" (8-bit) immediate data to register.
	MOVES	Copy data using address space specified by a control register (privi-leged instruction).
	SWAP	Swap left and right halves of register.
Data processing	ABCD	Add decimal (BCD) numbers with carry (extend) flag.
	ADD	Add binary (twos-complement) numbers.
	ADDA	Add to address register (unsigned binary addition).
	ADDI	Add immediate binary operand.
	ADDQ	Add "quick" (3-bit) immediate binary operand.
	ADDX	Add binary with carry (extension) flag.
	AND.x	Bitwise logical AND (.x = I denotes immediate operand).
	AS.x	Arithmetic left (.x = L) or right (.x = R) shift with extension.
	CLR	Clear operand by resetting all bits to 0.
	DIV.x	Divide signed (.x = S) or unsigned (.x = U) binary numbers.
	EOR.x	Bitwise logical EXCLUSIVE OR (.x = I denotes immediate operand).
	EXT	Extend the sign bit of subword to fill register.
	LS.x	Logical (simple) left (.x = L) or right (.x = R) shift.
	MUL.x	Multiply signed (.x = S) or unsigned (.x = U) binary numbers.
	NBCD	Negate decimal number (subtract with carry from zero).
	NEG	Negate binary number (subtract from zero).
	NEGX	Negate binary number (subtract with carry from zero).
	NOT	Bitwise logical complement.
	OR.x	Bitwise logical OR (.x = I denotes immediate operand).
	PACK*	Convert number from unpacked to packed BCD format.
	RO.x	Rotate (circular shift) left (.x = L) or right (.x = R).
	ROX.x	Rotate left (.x = L) or right (x = R) including the X (extend) flag.
	SBCD	Subtract decimal (BCD) numbers.
	SUB	Subtract binary (twos-complement) numbers.
	SUBA	Subtract from address register (unsigned binary subtraction).
	SUBI	Subtract immediate binary operand.
	SUBQ	Subtract "quick" (3-bit) immediate binary operand.
	SUBX	Subtract binary with borrow (extend) flag.
	UNPK*	Convert number from packed to unpacked BCD format.

Figure 3.12
Instruction set of the 68020.

characteristic of a RISC. For example:

```
ADD   (A0), D0
```

specifies the memory-to-register add operation D0 := M(A0) + D0.

EXAMPLE 3.3 680X0 PROGRAM FOR VECTOR ADDITION. Figure 3.13 gives an example of 680X0 assembly-language code that illustrates several of its basic instruction types and addressing methods. This program adds two 1000-element vectors A and B to produce a third vector C. Each vector is assumed to be a decimal

Type	Opcode	Description
Program control	Bcc	Branch relative to PC if specified condition code *cc* is set.
	Bxxx	Test, modify, and/or transfer (depending on .xxx) a specified bit; set Z flag to indicate old bit value.
	BFxxx*	Test, modify, and/or transfer (depending on .xxx) a specified bit field; set flags to indicate old bit-field value.
	BKPT*	Execute a breakpoint trap (used for debugging).
	BRA	Branch unconditionally relative to PC.
	BSR	Call (branch to) subroutine at address relative to PC; save old PC in stack.
	CALLM*	Call subroutine (program module) saving specified control information in stack.
	CASx*	Compare specified operands and update register.
	CHKx	Check register against specified values (address bounds); trap if bounds are exceeded.
	CMPx	Compare two operand values; set flags based on result; *x* indicates operand type.
	DBcc	Loop instruction: Test condition *cc* and perform no operation if condition is met; otherwise, decrement specified register and branch to specified address.
	ILLEGAL*	Perform trap operation corresponding to an illegal opcode.
	JMP	Branch unconditionally to specified (nonrelative) address.
	JSR	Call (jump to) subroutine at specified (nonrelative) address; save old PC in stack.
	LEA	Compute effective address and load into address register.
	LINK	Allocate local data and parameter region in the stack.
	NOP	No operation (except increment PC); instruction execution continues.
	PEA	Compute effective address and push into stack.
	RTD	Return from subroutine and deallocate stack parameter region.
	RTE	Return from exception (privileged instruction).
	RTM*	Return and restore control (module state) information.
	RTR	Return and restore condition codes.
	RTS	Return from subroutine.
	Scc	Set operand to 1s (0s) if condition code *cc* is true (false).
	STOP	Load status register and halt (privileged instruction).
	TRAP	Begin exception processing at specified address.
	TRAPcc	If condition *cc* is true, then begin exception processing.
	TST	Test an operand by comparing it to zero and setting flags.
	UNLK	Deallocate local data and parameter area in the stack.
External synchronization	cpxxx*	If condition holds, then branch with external coprocessor as specified by .xxx.
	RESET	Reset or restart external device (privileged instruction).
	TAS	Test operand and set one of its bits to 1 using an indivisible memory-access cycle.

*Instruction not in the original 68000 instruction set.

Figure 3.12
(continued).

number composed of 1000 two-digit bytes. Each vector is stored in a fixed block of main memory whose location is known. For example, vector A is stored in memory locations 1001,1002,1003, ...,1999,2000.

The desired addition is accomplished by executing the ABCD (add using the BCD number format) instruction 1000 times. The address registers A0, A1, and A2 are used as pointers to the current 1-byte operands, and they are initialized to the required starting values using the first three MOVE instructions. These instructions use immediate addressing denoted by the prefix # to specify instruction fields that contain actual address values, while a register name such as A0 indicates that the desired operand is

Location	Instruction		Comment
	MOVE.L	#2001, A0	Load address 2001 into register A0 (pointer to vector A).
	MOVE.L	#3001, A1	Load address 3001 into register A1 (pointer to vector B).
	MOVE.L	#4001, A2	Load address 4001 into register A2 (pointer to vector C).
START	ABCD	-(A0), -(A1)	Decrement contents of A0 and A1 by 1, then add M(A0) to M(A1) using 1-byte decimal addition.
	MOVE.B	(A1), -(A2)	Decrement A2 and then store the 1-byte sum M(A1) in location M(A2) of vector C.
TEST	CMPA	#1001, A0	Compare 1001 to address in A0. If equal, set the Z flag (condition code) to 1; otherwise, reset Z to 0.
	BNE	START	Branch to START if Z is not equal to 1.

Figure 3.13
680X0 assembly-language program for vector addition.

the contents of the named register—this is direct addressing. The ABCD and MOVE.B (move byte) instructions use indirect addressing, indicated by parentheses. In this case the data specified by (A0) is the content of the memory location whose address is stored in A0, that is, the data in M(A0). Finally the minus prefix in the operand -(A0) means that A0 is decremented by one before it is used to access main memory, a mode of addressing called *autoindexing*.

The program of Figure 3.13 loads three starting addresses into the selected address registers. Since the ABCD and MOVE.B instructions begin by automatically decrementing these registers, their initial values are made one bigger than the biggest address assigned to the corresponding vector. The ABCD instruction performs the following set of operations:

$$A0 := A0 - 1,\ A1 := A1 - 1;\ M(A1) := M(A1) + M(A0);\ \text{set flags},$$

which are relatively slow because of the memory access required. The MOVE.B instruction implements the memory-to-memory move operation with autoindexing

$$A2 := A2 - 1;\ M(A2) := M(A1);\ \text{set flags}$$

The compare-address instruction CMPA checks for program termination by comparing the current address in A0 to 1001, the lowest address assigned to vector *A*. It actually subtracts its first operand (1001 in this case) from its second and sets the status flags (condition code) based on the result. Hence if A0 > 1001, then A0 − 1001 > 0 and CMPA sets the zero flag Z to 0, indicating a nonzero result. (It also sets various other flags not used by this program). When A0 finally reaches 1001, A0 − 1001 = 0, so CMPA sets Z to 1. Now the last instruction BNE, which stands for branch if not equal to zero, is a conditional branch instruction whose operation is described by

$$\textbf{if } Z \neq 1 \textbf{ then } PC := START$$

It therefore transfers execution back to the ABCD instruction in location START as long as A0 >1001. When A0 finally reaches 1001, Z becomes 1, and PC is incremented normally to exit from the program.

It is interesting to compare this 680X0 program with the similar programs given earlier for the IAS (Figure 1.15) and PowerPC (Figure 1.27) computers.

Coprocessors. The built-in instruction repertoire of the 68020 includes fixed-point multiplication and division and stack-based instructions for transferring control between programs. Hardware-implemented floating-point instructions are not available directly; however, they are provided indirectly by means of an auxiliary IC, the 68881 floating-point coprocessor. (The ARM6 also has provisions for external coprocessors.) In general, a *coprocessor P* is a specialized instruction execution unit that can be coupled to a microprocessor so that instructions to be executed by *P* can be included in programs fetched by the microprocessor. Thus the coprocessor serves as an extension to the microprocessor and forms part of the CPU as indicated in Figure 3.14.

The 68881 (and the similar but faster 68882) contains a set of eight 80-bit registers for storing floating-point numbers of various formats, including 32- and 64-bit numbers conforming to the standard IEEE 754 format (presented later). Additional control registers in the 68881 allow it to communicate with the 68020. A set of coprocessor instructions are defined for the 68020; they contain command fields specifying floating-point operations that the 68881 can execute. When the 68020 fetches and decodes such an instruction, it transfers the command portion to the coprocessor, which then executes it. Further exchanges take place between the main processor and the coprocessor until the coprocessor completes execution of its current operation, at which point the 68020 proceeds to its next instruction. The commands executed by the 68881 include the basic

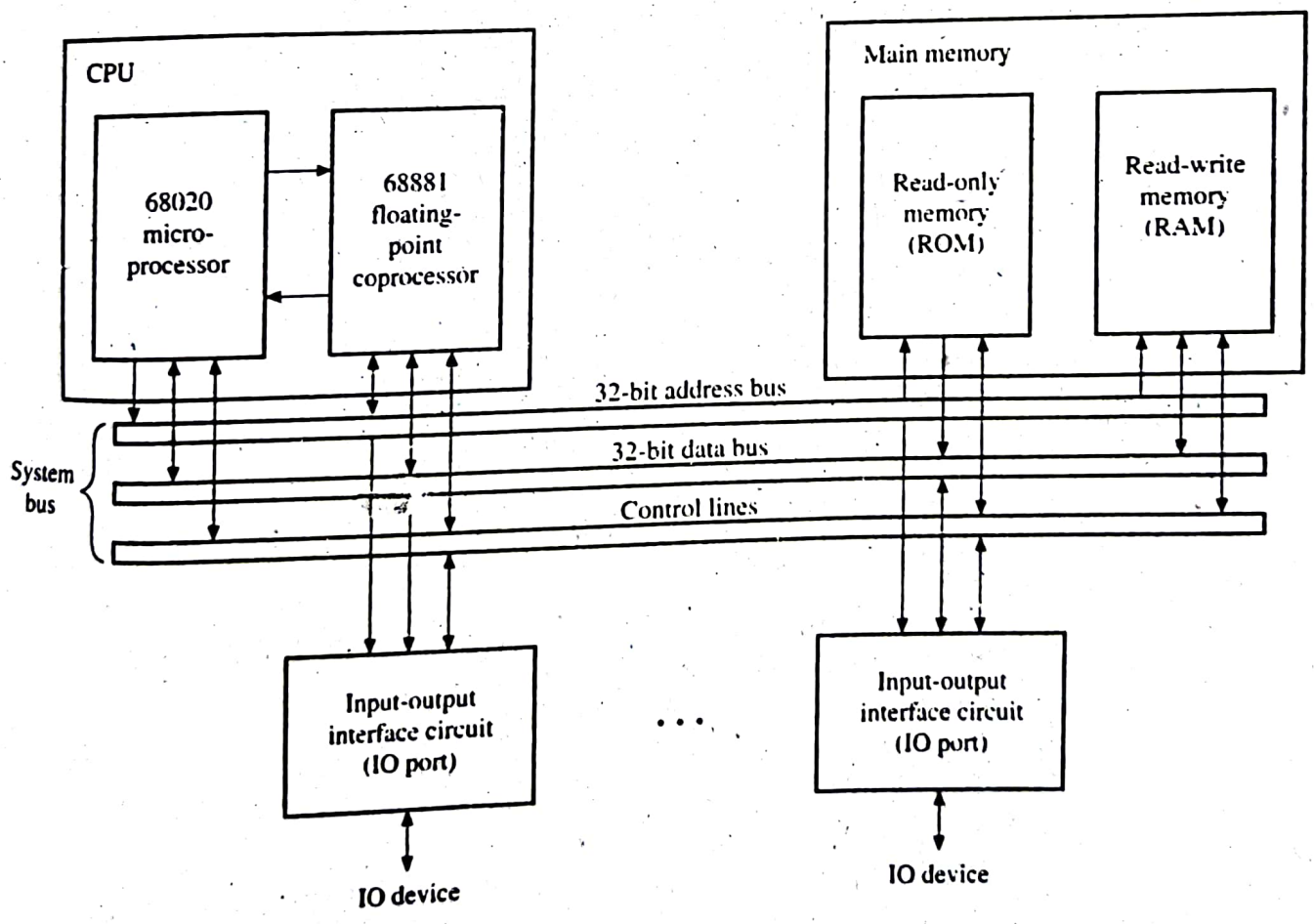


Figure 3.14
68020-based microcomputer with floating-point coprocessor.

arithmetic operations (add, subtract, multiply, and divide), square root, logarithms, and trigonometric functions. Other types of coprocessors may be attached to the 68020 in similar fashion. Later members of the 680X0 family take advantage of advances in VLSI to integrate a floating-point (co)processor into the CPU chip.

Other design features. Like the IBM System/360-370 and the ARM6, the CPU has a supervisor state intended for operating system use and a user state for application programs. As Figures 3.11 and 3.12 indicate, certain "privileged" control registers and instructions can be used only in the supervisor state. User and supervisory programs are thus clearly separated—for example, they employ different stack pointers—thereby improving system security. 680X0-based computers are also designed to allow easy implementation of *virtual memory*, whereby the operating system makes the main memory appear larger to user programs than it really is. Hardware support for virtual memory is provided by the 68851 memory management unit (MMU), another 680X0 coprocessor.

Provided they meet certain independence conditions, up to three 68020 instructions can be processed simultaneously in pipeline fashion. This pipelining is complicated by the fact that instruction lengths and execution times vary, a problem that RISCs try to eliminate. Another speedup feature found in the 68020 is a small instruction-only cache (I-cache). The 68020 prefetches instructions from main memory while the system bus is idle; the instructions can subsequently be read much more quickly from the on-chip cache than from the off-chip main memory. An unusual feature of the 68020 noted in Figure 3.11 is its use of two levels of microprogramming to implement the CPU's control logic. For the manufacturer, this feature increases design flexibility while reducing IC area compared with conventional (one-level) microprogrammed control.

3.2 DATA REPRESENTATION

The basic items of information handled by a computer are instructions and data. We now examine the methods used to represent such information, focusing on the formats for numerical data.

3.2.1 Basic Formats

Figure 3.15 shows the fundamental division of information into instructions (operation or control words) and data (operands). Data can be further subdivided into numerical and nonnumerical. In view of the importance of numerical computation, computer designs have paid a great deal of attention to the representation of numbers. Two main number formats have evolved: fixed-point and floating-point. The binary fixed-point format takes the form $b_A b_B b_C \ldots b_K$, where each b_i is 0 or 1 and a binary point is present in some fixed but implicit position. A floating-point number, on the other hand, consists of a pair of fixed-point numbers M,E, which denote the number $M \times B^E$, where B is a predetermined base. The many formats used to encode fixed-point and floating-point numbers will be examined later in

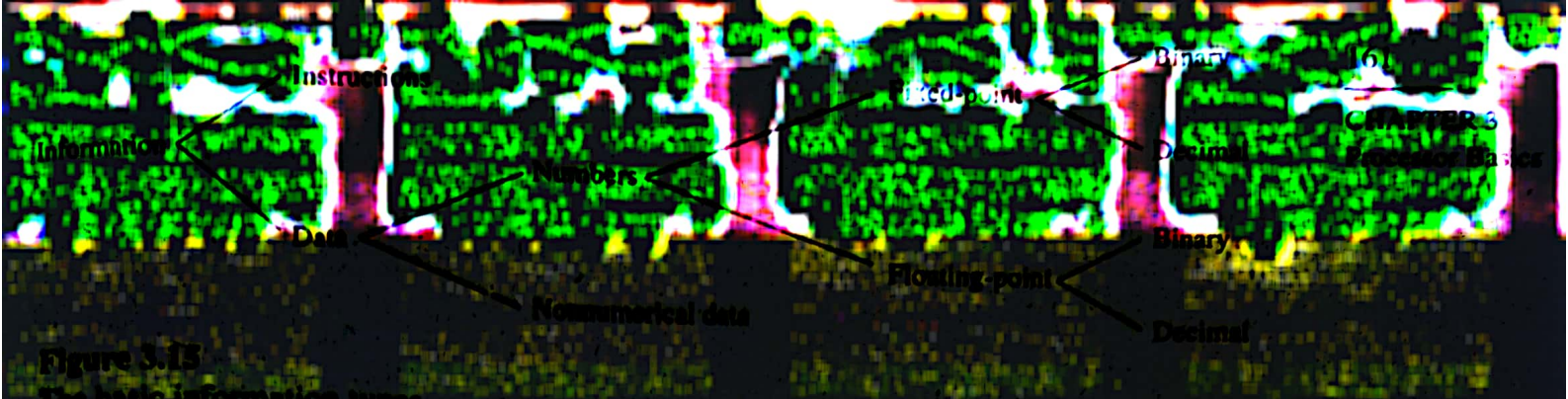


Figure 3.15
The basic information types.

the chapter. Nonnumerical data usually take the form of variable-length character strings encoded in one of several standard codes, such as ASCII (American Standards Committee on Information Exchange) code.

Word length. Information is represented in a digital computer by means of binary words, where a *word* is a unit of information of some fixed length n. An n-bit word allows up to 2^n different items to be represented. For example, with $n = 4$, we can encode the 10 decimal digits as follows:

$$0 = 0000 \quad 1 = 0001 \quad 2 = 0010 \quad 3 = 0011 \quad 4 = 0100$$
$$5 = 0101 \quad 6 = 0110 \quad 7 = 0111 \quad 8 = 1000 \quad 9 = 1001 \qquad (3.7)$$

To encode alphanumeric symbols or *characters*, 8-bit words called *bytes* are commonly used. As well as being able to encode all the standard keyboard symbols, a byte allows efficient representation of decimal numbers that are encoded in binary according to (3.7). A byte can store two decimal digits with no wasted space. Most computers have the 8-bit byte as the smallest addressable unit of information in their main memories. The CPU also has a standard word size for the data it processes. Word size is typically a multiple of 8, common CPU word sizes being 8, 16, 32, and 64 bits.

No single word length is suitable for representing every kind of information encountered in a typical computer. Even within a single domain such as a computer's instruction set, we often find several different word sizes. For example, instructions such as load and store that reference memory need long address fields. Instructions whose operands are all in the CPU need not contain memory addresses and so can be shorter. The precision of a number word is determined by its length; it is common therefore to have numbers of various sizes. Figure 3.16 gives a sampling of data sizes used by the Motorola 680X0. As here, the term *word* is often restricted to mean a 32-bit (4 byte) word. (680X0 literature refers to 32-bit words with the nonstandard term *long word*.) Fixed-point numbers come in lengths of 1, 2, 4, or more bytes. Floating-point numbers also come in several lengths, the shortest (single precision) number being one word (32 bits) long.

The circuits of a CPU must be carefully designed to permit various information formats to coexist smoothly. For example, if instruction length varies, as is the case in many CISC microprocessors, the program control unit must be designed to determine an instruction's length from its opcode and to fetch a variable number of instruction bytes from memory. It must also increment the program counter by a

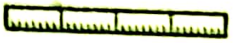
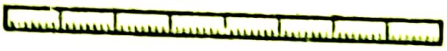
Bits	Name	Illustration	Typical uses
1	Bit		Status flag. Logic variable.
8	Byte		Smallest addressable memory item. Binary-coded decimal digit pair.
16	Halfword		Short fixed-point number. Short address (offset). Short instruction.
32	Word		Fixed- or floating-point number. Memory address. Instruction.
64	Double word		Long instruction. Double-precision floating-point number.

Figure 3.16
Some information formats of the Motorola 680X0 microprocessor series.

variable amount to obtain the address of the next consecutive instruction. Thus while the ARM6 has instructions of length 4 bytes only, the 68020's instructions range in length from 2 to 10 bytes.

Instruction sets commonly have features to make it easy to apply instructions to nonstandard-length operands. An example is the add-with-carry (ADC) instruction and its counterpart subtract with carry, which enable add and subtract instructions to apply to long fixed-point numbers by adding them in short segments and propagating carries from segment to segment. Suppose, for example, that we want to add two unsigned 64-bit (double word) binary integers A and B using the ARM6 instruction set (Figure 3.10), which is designed to add 32-bit words. Let A be placed in registers R0 and R1, with the right (least significant) half of A in R0. Similarly, let B be placed in registers R2 and R3, with its right half in R2. We first apply the ADD instruction with inputs R0 and R2 and place the resulting sum in R4. We also instruct ADD to activate the status flags, which requires an *S* suffix to the ARM6 opcode, changing it to ADDS. (In most other computers the flags are set automatically by all data-processing instructions.) ADDS results in the carry flag C assuming the value of the carry-out bit produced by the addition R0 + R2. Then we apply the ADC (add with carry) instruction with inputs R1 and R3 to compute the sum R1 + R3. In the following ARM6 code, the final sum $A + B$ is placed in R4 and R5.

HDL format	ARM6 assembly-language format		Narrative format (comment)
C.R4 := R0 + R2	ADDS	R4,R0,R2	Add right words and store carry signal C.
R5 : = R1 + R3 + C	ADC	R5,R1,R3	Add left words plus C.

Storage order. A small but important aspect of data representation is the way in which the bits of a word are indexed. We will usually follow the convention illustrated in Figure 3.17, where the right-most bit is assigned the index 0 and the bits are labeled in increasing order from right to left. The advantage of this convention is that when the word is interpreted as an unsigned binary integer, the low-order indexes correspond to the numerically less significant bits and the high-order indexes correspond to the numerically more significant bits. Similarly, we label the

significant byte $B_{i,0}$ on the right and assign it the lowest index 0. Now the entire sequence can be rewritten as

$$W_0, W_1, \ldots, W_m = B_{0,3}, B_{0,2}, B_{0,1}, B_{0,0}, B_{1,3}, B_{1,2}, B_{1,1}, B_{1,0}, \ldots, B_{m,3}, B_{m,2}, B_{m,1}, B_{m,0} \quad (3.8)$$

Suppose we store these $4(m + 1)$ bytes in M using the "natural" order defined by (3.8); that is, we assign a sequence of increasing memory addresses

$$adr_0, adr_1, adr_2, adr_3, \ldots, adr_{4m+3}, adr_{4m+4}$$

to the bytes as listed in (3.8). This storage sequence, which is illustrated in Figure 3.18*a*, is a byte-storage convention called *big-endian*.[2] It is so named because the most significant (biggest) byte $B_{i,3}$ of word W_i is assigned the lowest address and the least significant byte $B_{i,0}$ is assigned the highest address. In other words, the big-endian scheme assigns the highest address to byte 0. The alternative byte-storage scheme called *little-endian* assigns the lowest address to byte 0. This corresponds to

$$W_0, W_1, \ldots, W_m = B_{0,0}, B_{0,1}, B_{0,2}, B_{0,3}, B_{1,0}, B_{1,1}, B_{1,2}, B_{1,3}, \ldots, B_{m,0}, B_{m,1}, B_{m,2}, B_{m,3}$$

and is illustrated by Figure 3.18*b*.

Interestingly, computer manufacturers have never agreed on this issue, so both the big-endian and little-endian conventions are in widespread use. For example, the Motorola 680X0 uses the big-endian method, whereas the Intel 80X86 series is little-endian. Some computers including the ARM family can switch between the two endian conventions.

Tags. In the von Neumann computer, instruction and data words are stored together in main memory and are indistinguishable from one another—this is the classic "stored program" concept. An item plucked at random from memory cannot be identified as an instruction or data. Different data types such as fixed-point and floating-point numbers also cannot be distinguished by inspection. A word's type is determined by the way a processor interprets it. In principle, the same word can be treated as an instruction and data at different times, for example, the word 24 in

[2] The allusion is to an argument appearing in *Gulliver's Travels* on whether an egg should be opened at its big or little end (Cohen, 1981).

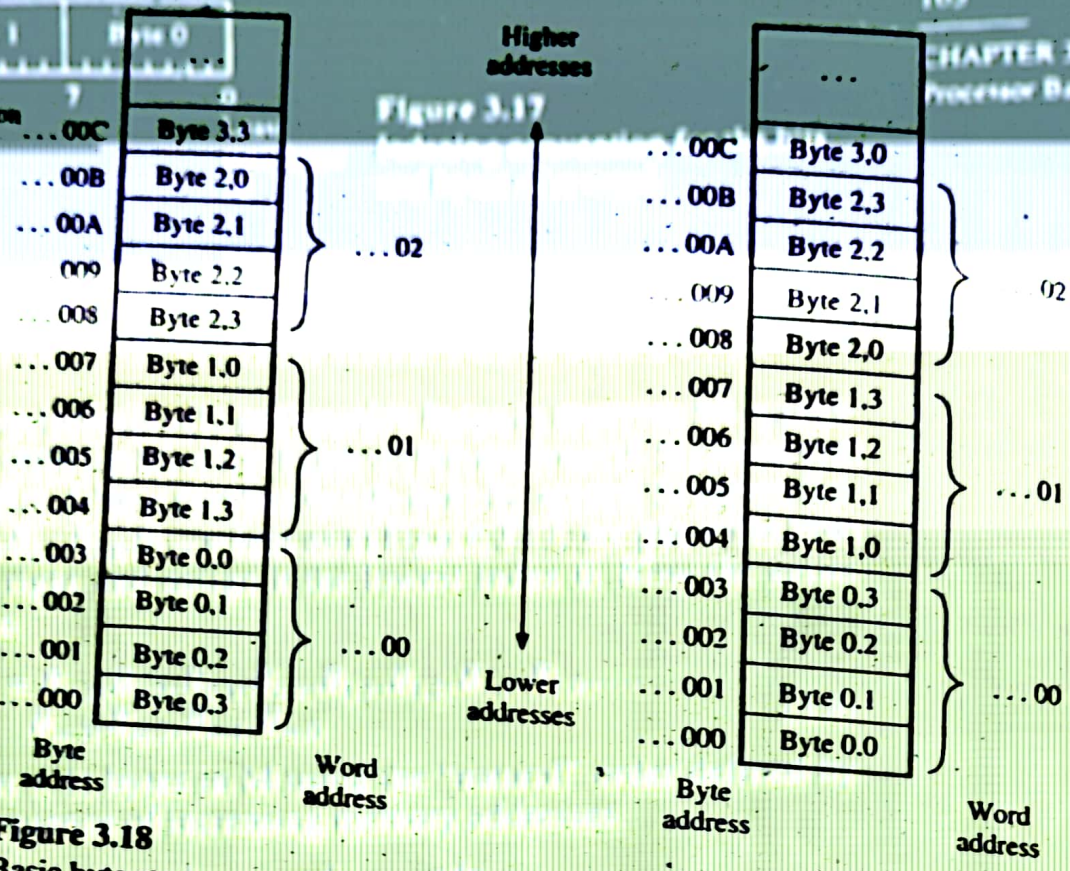


Figure 3.18
Basic byte storage methods: (*a*) big-endian and (*b*) little-endian.

the instruction sequence

```
X := X + Y;
go to X;
```

It is the programmer's (and compiler's) responsibility to ensure that data are not interpreted as instructions, and vice versa. A reason for this deliberate indistinguishability of data and instructions can be seen in the design of the IAS computer (section 1.2.2). The IAS's address-modify instructions alter stored instructions in main memory. The ability to modify instructions in this way—in effect, treating them as data—is useful when processing indexed variables, as illustrated in Example 1.4. However, this type of instruction modification in memory became obsolete with the introduction of address-indexing hardware.

A few computer designers have argued that the major information types should be assigned formats that identify them [Feustel 1973; Myers 1982]. This can be done by associating with each information word a group of bits, called a *tag*, that identifies the word's type. The tag may be considered as a physical implementation of the **type** declaration found in some high-level programming languages. One of the earliest machines to use tags was the 1960s-vintage Burroughs B6500/7500 series, which employed a 3-bit tag field in every word so that eight word types could be distinguished. The 52-bit word format of the B6500/7500 and the interpretation of its tag appear in Figure 3.19.

Tagging simplifies instruction specification. In conventional, nontagged computers, an instruction's opcode must explicitly or implicitly specify the type of data on which it operates. The PCU must know the operand types in order to route them

Figure 3.19
Tagged-word format of the Burroughs B6500/7500 series.

to the proper arithmetic circuits and registers. It is therefore necessary to provide distinct instructions for each data type; for example, add binary word, add binary half-word, add BCD word, add floating-point word, and add floating-point double word. If, on the other hand, tags distinguish the operand types, then a single ADD opcode suffices for all cases. The processor merely has to inspect an operand's tag to determine its type. Furthermore, tag inspection permits the hardware to check for software errors, such as an attempt to add operands whose types are incompatible. Tags have a serious cost disadvantage, however. They increase memory size and add to the system hardware costs without increasing computing performance. This fact has severely restricted the use of tagged architectures.

Error detection and correction. Various factors like manufacturing defects and environmental effects cause errors in computation. Such errors frequently appear when information is being transmitted between two relatively distant points within a computer or is being stored in a memory unit. "Noise" in the communication link can corrupt a bit x that is being sent from A to B so that B receives $\bar{x}$ instead of x. To guard against errors of this type, the information can be encoded so that special logic circuits can detect, and possibly even correct, the errors.

A general way to detect or correct errors is to append special check bits to every word. One popular technique employs a single check bit c_0 called a *parity bit*. The parity bit is appended to an n-bit word $X = (x_0, x_1, \ldots, x_{n-1})$ to form the $(n+1)$-bit word $X^* = (x_0, x_1, \ldots, x_{n-1}, c_0)$; see Figure 3.19. Bit c_0 is assigned the value 0 or 1 that makes the number of ones in X^* even, in the case of *even-parity* codes, or odd, in the case of *odd-parity* codes. In the even-parity case, c_0 is defined by the logic equation

$$c_0 = x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1} \tag{3.9}$$

where $\oplus$ denotes EXCLUSIVE-OR, while in the odd-parity case

$$\bar{c}_0 = x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1}$$

Suppose that the information X is to be transmitted from A to B. The value of c_0 is generated at the source point A using, say, (3.9), and X^* is sent to B. Let B receive the word $X'^* = (x'_0, x'_1, \ldots, x'_{n-1}, c'_0)$. B then determines the parity of the received word by recomputing the parity bit according to (3.9) thus:

$$c'_0 = x'_0 \oplus x'_1 \oplus \ldots \oplus x'_{n-1}$$

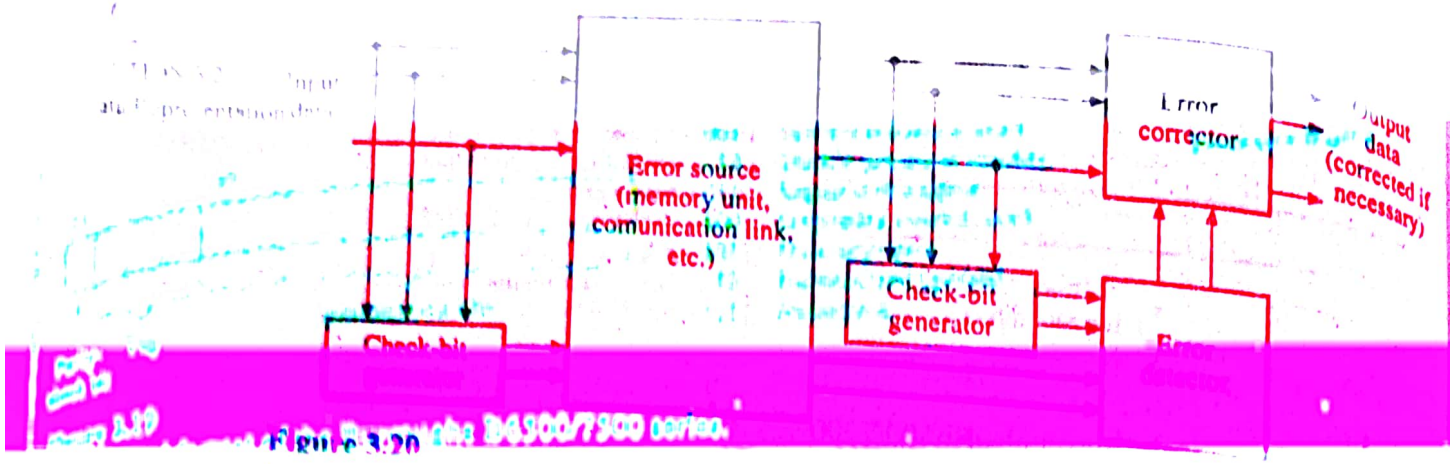


Figure 3.20 Error detection and correction logic

The received parity bit c'_0 and the reconstituted parity bit c^*_0 are then compared. If $c'_0 \neq c^*_0$, the received information contains an error. In particular, if exactly 1 bit of X^* has been inverted during the transmission process (a single-bit error), then $c'_0 \neq c^*_0$. If $c'_0 = c^*_0$, it can be concluded that no single-bit error occurred, but the possibility of multiple-bit errors is not ruled out. For example, if a 0 changes to 1 and a 1 changes to 0 (a double error), then the parity of X' is the same as that of X^* and the error will go undetected. The parity bit c_0 therefore provides *single-error detection*. It does not detect all multiple errors, much less provide any information about the location of the erroneous bits.

The parity-checking concept can be extended to the detection of multiple errors or to the location of single or multiple errors. These goals are achieved by providing additional parity bits, each of which checks the parity of some subset of the bits in the word X^*. By appropriately overlapping these subsets, the correctness of every bit can be determined. Suppose, for instance, that we can deduce from the parity checks the identity of the bit x_i responsible for a single-bit error. It is then a simple matter to introduce logic circuits to replace x_i by $\bar{x}_i$, thus providing *single-error correction*. Let c be the number of check bits required to achieve single-error correction with n-bit data words. Clearly the check bits have 2^c patterns that must distinguish between $n + c$ possible error locations and the single error-free case. Hence c must satisfy the inequality

$$2^c \geq n + c + 1 \tag{3.10}$$

For $n = 16$, (3.10) implies that $c \geq 5$, while for $n = 32$ we have $c \geq 6$. A variety of practical single-error-correcting parity-check codes meet the lower bound on c implied by (3.10) [Siewiorek and Swarz 1992]. Some of these codes can also detect double errors and so are called *single-error-correcting double-error-detecting* (SECDED) codes. As the main memories of computers have increased in storage capacity and decreased in physical size, they have become more prone to transient failures that are often correctable via SECDED codes. Figure 3.20 shows the structure of a typical error detection and correction scheme used with a computer's main memory.

3.2.2 Fixed-Point Numbers

In selecting a number representation to be used in a computer, the following factors should be taken into account:

- The number types to be represented; for example, integers or real numbers.
- The range of values (number magnitudes) likely to be encountered.
- The precision of the numbers, which refers to the maximum accuracy of the representation.
- The cost of the hardware required to store and process the numbers.

The two principal number formats are fixed-point and floating-point. Fixed-point formats allow a limited range of values and have relatively simple hardware requirements. Floating-point numbers, on the other hand, allow a much larger range of values but require either costly processing hardware or lengthy software implementations.

Binary numbers. The fixed-point format is derived directly from the ordinary (decimal) representation of a number as a sequence of digits separated by a decimal point. The digits to the left of the decimal point represent an integer; the digits to the right represent a fraction. This is *positional notation* in which each digit has a fixed *weight* according to its position relative to the decimal point. If $i > 1$, the ith digit to the left (right) of the decimal point has weight 10^{i-1} (10^{-i}). Thus the five-digit decimal number 192.73 is equivalent to

$$1 \times 10^2 + 9 \times 10^1 + 2 \times 10^0 + 7 \times 10^{-1} + 3 \times 10^{-2}$$

More generally, we can assign weights of the form r^i, where r is the *base* or *radix* of the number system, to each digit.

The most fundamental number representation used in computers employs a base-two positional notation. A binary word of the form

$$b_N \ldots b_3 b_2 b_1 b_0 \cdot b_{-1} b_{-2} b_{-3} b_{-4} \ldots b_M \tag{3.11}$$

represents the number

$$\sum_{i=M}^{N} b_i 2^i$$

When unclear from the context, the base r being used will be indicated by appending r as a subscript to the number. Thus 1010_2 denotes the binary equivalent of the decimal number 10_{10}, whereas 10_2 denotes 2_{10}. The format of (3.11) is an example of a fixed-point binary number and is used to denote unsigned numbers. Several distinct methods used for representing signed (positive and negative) numbers are discussed below.

Suppose that an n-bit word is to contain a signed binary number. One bit is reserved to represent the sign of the number, while the remaining bits indicate its magnitude. To permit uniform processing of all n bits, the sign is placed in the leftmost position, and 0 and 1 are used to denote plus and minus, respectively. This

leads to the format

$$\underbrace{x_{n-1}}_{\text{Sign}}\ \underbrace{x_{n-2}x_{n-3}\cdots x_2x_1x_0}_{\text{Magnitude}} \tag{3.12}$$

The precision allowed by this format is $n-1$ bits, which is equivalent to $(n-1)\log_2 10$ decimal digits. The binary point is not explicitly represented; instead, it is implicitly assigned to some fixed location in the word. The binary point's position is not very important from the point of view of design. In many situations the numbers being processed are integers, so the binary point is assumed to lie immediately to the right of the least significant bit x_0. Monetary quantities are often expressed as integers; for instance, \$54.30 might be expressed as 5430 cents. Using an n-bit integer format, we can represent all integers N with magnitude $|N|$ in the range $0 \le |N| \le 2^n - 1$. The other most widely used fixed-point format treats (3.12) as a fraction with the binary point lying between x_{n-1} and x_{n-2}. The fraction format denotes numbers with magnitudes in the range $0 \le |N| \le 1 - 2^{-n}$.

Signed numbers. Suppose that both positive and negative binary numbers are to be represented by an n-bit word $X = x_{n-1}x_{n-2}x_{n-3}\ldots x_2x_1x_0$. The standard format for positive numbers is given by (3.12) with a sign bit of 0 on the left and the magnitude to the right in the usual positional notation. This means that each magnitude bit x_i, $0 \le i \le n-2$, has a fixed weight of the form 2^{k+i}, where k depends on the position of the binary point. A natural way to represent negative numbers is to employ the same positional notation for the magnitude and simply change the sign bit x_{n-1} to 1 to indicate minus. Thus with $n = 8$, $+75 = 01001011$, while $-75 = 11001011$. This number code is called *sign magnitude*. Note that humans normally use decimal versions of sign-magnitude code. Nevertheless, operations like subtraction are costly to implement by logic circuits when sign-magnitude codes are used. However, multiplication and division of sign-magnitude numbers is almost as easy as the corresponding operation for unsigned numbers, as Example 2.7 (section 2.3.3) shows.

Several number codes have been devised that use the same representation for positive numbers as the sign-magnitude code but represent negative numbers in different ways. For example, in the *ones-complement* code, $-X$ is denoted by $\bar{X}$, the bitwise logical complement of X. In this code we again have $+75 = 01001011$, but now $-75 = 10110100$. In the *twos-complement* code, $-X$ is formed by adding 1 to the least significant bit of $\bar{X}$ and ignoring any carry bit generated from the most significant (sign) position. If $X = x_{n-1}x_{n-2}\ldots x_0$ is an n-bit binary fraction, $-X$ can be expressed as follows:

$$-X = \underbrace{\bar{x}_{n-1}.\bar{x}_{n-2}\bar{x}_{n-3}\cdots\bar{x}_1\bar{x}_0}_{\text{Implicit binary point}} + \underbrace{0.00\ldots01}_{\text{Implicit binary point}} \quad \text{(modulo 2)} \tag{3.13}$$

where the use of modulo-2 addition corresponds to ignoring carries from the sign position. If X is an integer, then (3.13) becomes

$$-X = \underbrace{\bar{x}_{n-1}\bar{x}_{n-2}\bar{x}_{n-3}\cdots\bar{x}_1\bar{x}_0.}_{\text{Implicit binary point}} + \underbrace{000\ldots01.}_{\text{Implicit binary point}} \quad \text{(modulo } 2^n\text{)} \tag{3.14}$$

For example, in twos-complement code +75 = 01001011 and −75 = 10110101. Note that in both complement codes x_{n-1} retains its role as the sign bit, but the remaining bits no longer form a simple positional code when the number is negative.

The primary advantage of the complement codes is that subtraction can be performed by logical complementation and addition only. Consider the twos-complement code. To subtract X from Y, just add $-X$ to Y, where $-X$ is obtained by logical complementation and addition of a 1 bit, as in (3.13) and (3.14). As we will see later, the sign bits do not require special treatment; consequently, twos-complement addition and subtraction can be implemented by a simple adder designed for unsigned numbers. Multiplication and division are more difficult to implement if twos-complement code is used instead of sign magnitude. The addition of ones-complement numbers is complicated by the fact that a carry bit from the most significant magnitude bit x_{n-2} must be added to the least significant bit position x_0. Otherwise ones-complement codes are quite similar to twos-complement codes and so will not be considered further.

Figure 3.21 illustrates how integers are represented using all three codes when $n = 4$. These codes are all referred to as *binary* codes to distinguish them from the so-called decimal codes discussed below. Observe that in all cases, 0000 represents zero. Only in the case of twos-complement code, however, is the nega-

	Binary code		
Decimal representation	Sign magnitude	Ones complement	Twos complement
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
−0	1000	1111	0000
−1	1001	1110	1111
−2	1010	1101	1110
−3	1011	1100	1101
−4	1100	1011	1100
−5	1101	1010	1011
−6	1110	1001	1010
−7	1111	1000	1001

Figure 3.21
Comparison of three 4-bit codes for signed binary numbers.

tive (numerical complement) of 0000 also 0000. This unique representation of zero is a significant advantage, for example, in implementing instructions like BNE in Figure 3.13 that test for zero. Consequently, twos-complement code is by far the most popular code for representing signed binary numbers in computers.

Exceptional conditions. If the result of an arithmetic operation involving n-bit numbers is too large (small) to be represented by n bits, *overflow (underflow)* is said to occur. It is generally necessary to detect overflow and underflow, since they may indicate bad data or a programming error. Consider, for example, the addition operation

$$z_{n-1}z_{n-2}\ldots z_0 := x_{n-1}x_{n-2}\ldots x_0 + y_{n-1}y_{n-2}\ldots y_0$$

using n-bit twos-complement operands. Assume that bitwise addition is performed with a carry bit c_i generated by the addition of x_i, y_i, and c_{i-1}. The output bits z_i and c_i can be computed according to the full-adder logic equations

$$z_i = x_i \oplus y_i \oplus c_{i-1}$$

$$c_i = x_i y_i + x_i c_{i-1} + y_i c_{i-1}$$

Let v be a binary variable indicating overflow when $v = 1$. Figure 3.22 shows how the sign bit z_{n-1} and v are determined as functions of the sign bits x_{n-1}, y_{n-1} and the carry bit c_{n-2}. The overflow indicator v is therefore defined by the logic equation

$$v = \bar{x}_{n-1}\bar{y}_{n-1}c_{n-2} + x_{n-1}y_{n-1}\bar{c}_{n-2}$$

If the combinations $(x_{n-1},y_{n-1},c_{n-2}) = (0,0,1)$ and $(1,1,0)$, which make $v = 1$, are removed from the truth table of Figure 3.22, then z_{n-1} is defined correctly for all the remaining combinations by the equation

$$z_{n-1} = x_{n-1} \oplus y_{n-1} \oplus c_{n-2}$$

Consequently, during twos-complement addition the sign bits of the operands can be treated in the same way as the remaining (magnitude) bits.

A related issue in computer arithmetic is *round-off error*, which results from the fact that every number must be represented by a limited number of bits. An

Inputs			Outputs	
x_{n-1}	y_{n-1}	c_{n-2}	z_{n-1}	v
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	0

Figure 3.22
Computation of the sign bit z_{n-1} and the overflow indicator v in twos-complement addition.

operation involving n-bit numbers frequently produces a result of more than n bits. For example, the product of two n-bit numbers contains up to $2n$ bits, all but n of which must normally be discarded. Retaining the n most significant bits of the result without modification is called *truncation*. Clearly the resulting number is in error by the amount of the discarded digits. This error can be reduced by a process called *rounding*. One way of rounding is to add $r^j/2$ to the number before truncation, where r^j is the weight of the least significant retained digit. For instance, to round 0.346712 to three decimal places, add 0.0005 to obtain 0.347212 and then take the three most significant digits 0.347. Simple truncation yields the less accurate value 0.346. Successive computations can cause round-off errors to build up unless countermeasures are taken. The number formats provided in a computer should have sufficient precision that round-off errors are of no consequence to most users. It is also desirable to provide facilities for performing arithmetic to a higher degree of precision if required. Such high precision is usually achieved by using several words to represent a single number and writing special subroutines to perform multiword, or *multiple-precision*, arithmetic.

Decimal numbers. Since humans use decimal arithmetic, numbers being entered into a computer must first be converted from decimal to some binary representation. Similarly, binary-to-decimal conversion is a normal part of the computer's output processes. In certain applications the number of decimal-binary conversions forms a large fraction of the total number of elementary operations performed by the computer. In such cases, number conversion should be carried out rapidly. The various binary number codes discussed above do not lend themselves to rapid conversion. For example, converting an unsigned binary number $x_{n-1}x_{n-2}...x_0$ to decimal requires a polynomial of the form

$$\sum_{i=0}^{n-1} x_i 2^{k+i}$$

to be evaluated.

Several number codes exist that facilitate rapid binary-decimal conversion by encoding each decimal digit separately by a sequence of bits. Codes of this kind are called *decimal codes*. The most widely used decimal code is the BCD (*binary-coded decimal*) code. In BCD format each digit d_i of a decimal number is denoted by its 4-bit equivalent $b_{i,3}b_{i,2}b_{i,1}b_{i,0}$ in standard binary form, as in (3.7). Thus the BCD number representing 971 is 100101110001. BCD is a weighted (positional) number code, since $b_{i,j}$ has the weight 10^i2^j. Signed BCD numbers employ decimal versions of the sign-magnitude or complement formats. The 8-bit ASCII code represents the 10 decimal digits by a 4-bit BCD field; the remaining 4 bits of the ASCII code word have no numerical significance.

Two other decimal codes of moderate importance are shown in Figure 3.23. The *excess-three* code can be formed by adding 0011_2 to the corresponding BCD number—hence its name. The advantage of the excess-three code is that it may be processed using the same logic used for binary codes. If two excess-three numbers are added like binary numbers, the required decimal carry is automatically generated from the high-order bits. The sum must be corrected by adding +3. For

Decimal digit	Decimal code			
	BCD	ASCII	Excess-three	Two-out-of-five
0	0000	0011 0000	0011	11000
1	0001	0011 0001	0100	00011
2	0010	0011 0010	0101	00101
3	0011	0011 0011	0110	00110
4	0100	0011 0100	0111	01001
5	0101	0011 0101	1000	01010
6	0110	0011 0110	1001	01100
7	0111	0011 0111	1010	10001
8	1000	0011 1000	1011	10010
9	1001	0011 1001	1100	10100

Figure 3.23
Some important decimal number codes.

example, consider the addition 5 + 9 = 14 using excess-three code.

$$\begin{array}{rrll} & & 1000 = 5 & \\ & + & \underline{1100} = 9 & \\ \text{Carry } 1 \leftarrow & & 0100 & \text{Binary sum} \\ & + & \underline{0011} & \text{Correction} \\ & & 0111 = 4 & \text{Excess-three sum} \end{array}$$

Binary addition of the BCD representations of 5 and 9 results in 1110 and no carry generation. (The binary sum of two BCD numbers can also be corrected to give the proper BCD sum as described later.) Some arithmetic operations are difficult to implement using excess-three code, mainly because it is a *nonweighted* code; that is, each bit position in an excess-three number does not have a fixed weight.

The final decimal code illustrated by Figure 3.23 is the *two-out-of-five* code. Each decimal digit is represented by a 5-bit sequence containing two 1s and three 0s; there are exactly 10 distinct sequences of this type. The particular merit of the two-out-of-five code is that it is single-error detecting, since changing any one bit results in a sequence that does not correspond to a valid code word. Its drawbacks are that it is a nonweighted code and uses 5 rather than 4 bits per decimal digit.

The main advantage of the decimal codes is ease of conversion between the internal computer representation that allows only the symbols 0, 1 and external representations using the 10 decimal symbols 0, 1, 2,..., 9. Decimal codes have two disadvantages.

1. They use more bits to represent a number than the binary codes. Decimal codes therefore require more memory space. An n-bit word can represent 2^n numbers using binary codes; approximately $10^{n/4} = 2^{0.830n}$ numbers can be represented if a 4-bit decimal code such as BCD or excess-three is used.
2. The circuitry required to perform arithmetic using decimal operands is more complex than that needed for binary arithmetic. For example, in adding BCD

numbers bit by bit, a uniform method of propagating carries between adjacent positions is not possible, since the weights of adjacent bits do not differ by a constant factor.

Hexadecimal numbers. One or two other numerical codes are encountered in the design or use of computers. Of particular importance is *hexadecimal* (hex) code, which is characterized by a base $r = 16$ and the use of 16 digits, consisting of the decimal digits 0,1,...,9 augmented by the six digits A,B,C,D,E, and F, which have the numerical values 10, 11, 12, 13, 14, and 15, respectively. The unsigned hexadecimal integer 2FA0C has the interpretation

$$\begin{aligned} &2 \times 16^4 + F \times 16^3 + A \times 16^2 + 0 \times 16^1 + C \times 16^0 \\ &\quad = 2 \times 65{,}536 + 15 \times 4{,}096 + 10 \times 256 + 0 \times 16 + 12 \times 1 \\ &\quad = 195{,}084 \end{aligned}$$

Hence $2FA0C_{16} = 195{,}084_{10}$.

Hexadecimal code is useful for representing long binary numbers, a consequence of the fact that the base 16 is a power of two. A hexadecimal number is converted to binary simply by replacing each hex digit by the equivalent 4-bit binary form. For example, we can convert $2FA0C_{16}$ to binary by replacing the first digit 2 by 0010, the second digit F by 1111, the third digit A by 1010, and so on, yielding

$$2FA0C_{16} = 00101111101000001100_2$$

Conversely, we can convert a binary number to hex form by replacing each four-digit group by the corresponding hex digit. Clearly hexadecimal-binary number conversion is very similar to BCD-binary conversion. By treating any binary word as an unsigned integer, we can easily convert the word to hex form as indicated above. Hex code provides a very convenient shorthand for binary information.

3.2.3 Floating-Point Numbers

The range of numbers that can be represented by a fixed-point number code is insufficient for many applications, particularly scientific computations where very large and very small numbers are encountered. Scientific notation permits us to represent such numbers using relatively few digits. For example, it is easier to write a quintillion as

$$1.0 \times 10^{18} \tag{3.15}$$

than as the 19-bit, fixed-point integer 1 000 000 000 000 000 000. The floating-point codes used in computers are binary (or binary-coded) versions of (3.15).

Basic formats. Three numbers are associated with a floating-point number: a *mantissa M*, an *exponent E*, and a *base B*. The mantissa M is also referred to as the *significand* or *fraction* in the literature. These three components together represent the real number $M \times B^E$. For example, in (3.15) 1.0 is the mantissa, 18 is the exponent, and 10 is the base. For machine implementation the mantissa and exponent are encoded as fixed-point numbers with a base r that is usually 2 or 10. The base B

is also r, or some power of r, for reasons that will become obvious. Since B is a constant, it need not be included in the number code; it is simply built into the circuits that process the numbers. A floating-point number is therefore stored as a word (M,E) consisting of a pair of signed fixed-point numbers: a mantissa M, which is usually a fraction or an integer, and an exponent E, which is an integer. The number of digits in M determines the precision of (M,E); B and E determine its range. With a word size of n bits, 2^n is the most real numbers that (M,E) can represent. Increasing B increases the range of the representable real numbers but results in a sparser distribution of numbers over that range.

As a small example, suppose that M and E are both 3-bit, sign-magnitude integers and $B = 2$. Then M and E can each assume the values ±0, ±1, ±2, and ±3. All binary words of the form $(M,E) = (x00, xxx)$ represent zero, where x denotes either 0 or 1. The smallest nonzero positive number is (001,111), denoting $1 \times 2^{-3} = 0.125$; (101,111) denotes –0.125. The largest representable positive number is (011,011), which denotes $3 \times 2^3 = 24$, while (111,011) denotes the largest negative number –24. Observe that the left-most bit, which is the sign of the mantissa, is also the sign of the floating-point number. Figure 3.24 illustrates the real numbers representable by this 6-bit, floating-point format. As the figure shows, they are sparsely and nonuniformly distributed over the range ±24.

The floating-point representation of most real numbers is only approximate. For instance, the 6-bit format of Figure 3.24 cannot represent the number 1.25; it is approximated by (011,101), representing 1.5, or by either (001,000) or (001,100), representing 1.0. Moreover, the results of most calculations with floating-point arithmetic only approximate the correct result. For example, in the system of Figure 3.24, the exact result (18) of the addition (011,001) + (011,010), which implements 6 + 12, is not representable. The closest representable number, that is, the best approximation to 18, is (010,011) = 16. Overflow occurs in this small system when a result's magnitude exceeds 24, and underflow occurs when a nonzero result has a magnitude less than 0.125. In practice, floating-point numbers must have long mantissas (at least 20 bits), and the results of floating-point operations must be carefully rounded off to minimize the errors inherent in floating-point representation. It is common practice for floating-point processing circuits to include a few extra mantissa digits termed *guard digits* to reduce approximation errors; the guard digits are removed automatically from the final results.

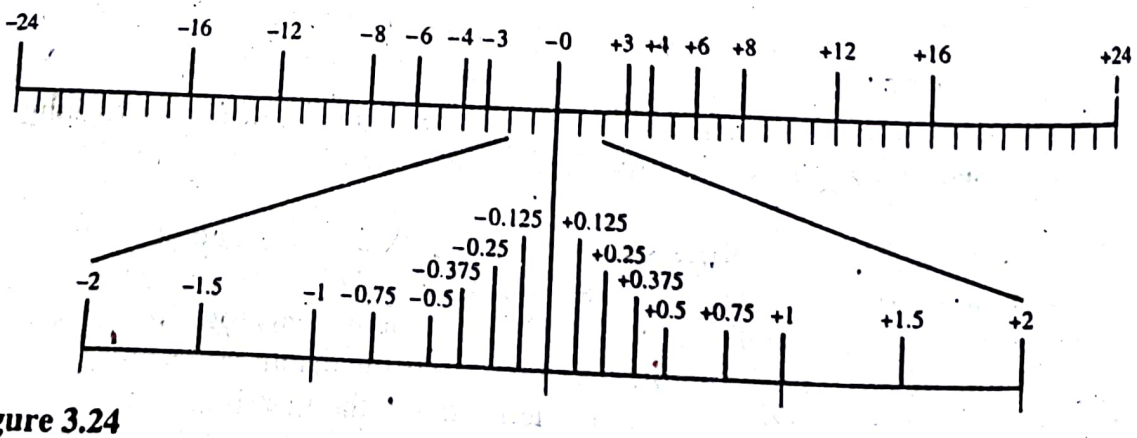


Figure 3.24
The real numbers representable by a hypothetical 6-bit, floating-point format.

Normalization and biasing. Floating-point representation is redundant in the sense that the same number can be represented in more than one way. For example, 1.0×10^{18}, 0.1×10^{19}, 1000000×10^{12}, and 0.000001×10^{24} are possible representations of a quintillion. It is generally desirable to have a unique or *normal* form for each representable number in a floating-point system. Consider the common case where the mantissa is a sign-magnitude fraction and a base of r is used. The mantissa is said to be *normalized* if the digit to the right of the radix point is not zero, that is, no leading zeros appear in the magnitude part of the number. Thus, for example, 0.1×10^{19} is the unique normal form of a quintillion using base 10, a decimal mantissa, and a decimal exponent. A binary fraction in twos-complement code is normalized when the sign bit differs from the bit to its right. This implies that no leading 1s appear in the magnitude part of negative numbers. Normalization restricts the magnitude $|M|$ of a fractional binary mantissa to the range

$$1/2 \leq |M| < 1$$

Normal forms can be defined similarly for other floating-point codes. An unnormalized number is normalized by shifting the mantissa to the right or left and appropriately incrementing or decrementing the exponent to compensate for the mantissa shift.

The representation of zero poses some special problems. The mantissa must, of course, be zero, but the exponent can have any value, since $0 \times B^E = 0$ for all values of E. Often in attempting to compute zero, round-off errors result in a mantissa that is nearly, but not exactly, zero. For the entire floating-point number to be close to zero, its exponent must be a very large negative number $-K$. This requirement suggests that the exponent used for representing zero should be the negative number with the largest magnitude that can be contained in the exponent field of the number format. If k bits are allowed for the exponent including its sign, then 2^k exponent bit patterns are available to represent signed integers, which can range either from -2^{k-1} to $2^{k-1} - 1$ or from $-2^{k-1} + 1$ to 2^{k-1}, so that K is 2^{k-1} or $2^{k-1} - 1$.

A second complication arises from the desirability of representing zero by a sequence of 0-bits only. This convention gives zero the same representation in both fixed- and floating-point formats, which facilitates the implementation of instructions that test for zero. These considerations suggest that floating-point exponents should be encoded in excess-K code similar to the excess-three code of Figure 3.23, where the exponent field E contains an integer that is the desired exponent value plus K. The quantity K is called the *bias*, and an exponent encoded in this way is called a *biased exponent* or *characteristic*. Figure 3.25 shows the possible values of an 8-bit exponent with bias 127 and 128.

Standards. Until the 1980s floating-point number formats varied from one computer family to the next, making it difficult to transport programs between different computers without encountering small but significant differences in such areas as round-off errors. To deal with this problem, the Institute of Electrical and Electronics Engineers (IEEE) sponsored a standard format for 32-bit and larger floating-point numbers, known as the IEEE 754 standard [IEEE 1985], which has been widely adopted by computer manufacturers. Besides specifying the permissible formats for M, E, and B, the IEEE standard prescribes methods for handling round-off errors, overflow, underflow, and other exceptional conditions.

Exponent bit pattern E	Unsigned value	Number represented	
		Bias = 127	Bias = 128
111 . . . 11	255	+128	+127
111 . . . 10	254	+127	+126
. . .		. . .	. . .
100 . . . 01	129	+2	+1
100 . . . 00	128	+1	0
011 . . . 11	127	0	−1
011 . . . 10	126	−1	−2
. . .		. . .	. . .
000 . . . 01	1	−126	−127
000 . . . 00	0	−127	−128

Figure 3.25
Eight-bit biased exponents with bias = 127 (excess-127 code) and bias = 128 (excess-128 code).

EXAMPLE 3.4 THE IEEE 754 FLOATING-POINT NUMBER FORMAT [IEEE 1985; GOLDBERG 1991]. This standard format for 32-bit numbers is illustrated in Figure 3.26. It comprises a 23-bit mantissa field M, an 8-bit exponent field E, and a sign bit S. The base B is two. As in all signed binary number formats, both fixed-point and floating-point, S occupies the left-most bit position. M is a fraction that with S forms a sign-magnitude binary number. For the reasons discussed earlier, floating-point numbers are usually normalized, meaning that the magnitude field should contain no insignificant leading bits. Hence the magnitude part of a normalized sign-magnitude number always has 1 as its most significant digit. There is no need to actually store this leading 1 in floating-point numbers, since it can always be inserted by the arithmetic circuits that process the numbers. Consequently, in the IEEE 754 format the complete mantissa (called the *significand* in the standard) is actually $1.M$, where the 1 to the left of the binary point is an implicit or *hidden* leading bit that is not stored with the number. Use of the hidden 1 means that the precision of a normalized number is effectively increased by 1 bit. The exponent representation is the 8-bit excess-127 code of Figure 3.25; hence the actual exponent value is computed as $E - 127$. The base B of the floating-point number is 2, so that a 1-bit left (right) shift of M corresponds to incrementing (decrementing) E by one.

Consequently, a 32-bit floating-point number conforming to the IEEE 754 standard represents the real number N given by the formula

$$N = (-1)^S 2^{E-127}(1.M) \tag{3.16}$$

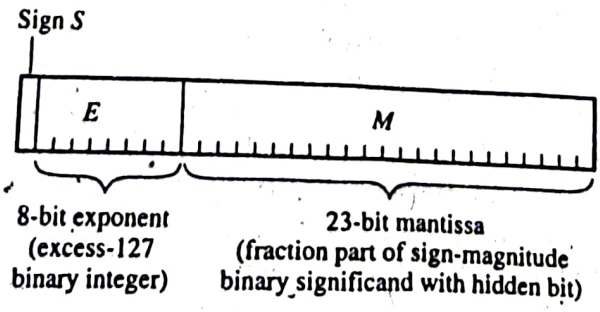


Figure 3.26
IEEE 754 standard 32-bit floating-point number format.

provided $0 < E < 255$. For example, the number $N = -1.5$ is represented by

1 01111111 10000000000000000000000

where $S = 1$, $E = 127$, and $M = 0.5$, since from (3.16) we have $N = (-1)^1 2^{127-127}(1.5) = -1.5$. Nonzero floating-point numbers in this format have magnitudes ranging from $2^{-126}(1.0)$ to $2^{+127}(2 - 2^{-23})$, that is, from 1.18×10^{-38} to 3.40×10^{38} approximately. In contrast, 32-bit, fixed-point binary formats for integers can only represent nonzero numbers with magnitudes from 1 to $2^{31} - 1$ (approximately 2.15×10^9). The 64-bit version of the IEEE 754 standard is a straightforward extension of the 32-bit case. It employs an 11-bit exponent E and a 52-bit mantissa M and defines the number

$$N = (-1)^S 2^{E-1023}(1.M) \tag{3.17}$$

where $0 < E < 2047$.

The IEEE floating-point standard addresses a number of subtle problems encountered in floating-point arithmetic. Well-defined formats are specified for the results of overflow, underflow, and other exceptional conditions, which often yield unpredictable and unusable numbers in computers employing other floating-point formats. The IEEE standard's exception formats are intended to set flags in the host processor, which subsequent instructions can use for error control, in many cases with little or no loss of accuracy. If the result of a floating-point operation is not a valid floating-point number, then a special code referred to as *not a number* (NaN) is used. Examples of operations that result in NaNs are dividing zero by zero and taking the square root of a negative number. NaN formats are identified in the standard by $M \neq 0$, and $E = 255$ (32-bit format) or $E = 2047$ (64-bit format).

When overflow occurs, meaning that a number has been produced whose magnitude is too big to represent by the usual format, the result is referred to as *infinity*, or ∞, and is identified by $M = 0$, and $E = 255$ (32-bit format) or $E = 2047$ (64-bit format). The 754 standard stipulates that operations using the floating-point infinities $\pm\infty$ should follow certain properties of infinity in real-number theory, such as $-\infty + N = \infty$ and $-\infty < N < +\infty$ for any finite N. If underflow occurs, implying that a result is nonzero, but too small to represent as a normalized number, it is encoded in a *denormalized*[3] form characterized by $E = 0$ and a significand $0.M$ having a leading 0 instead of the usual leading 1. Denormalization reduces the effect of underflow to a systematic loss of precision equivalent to a small round-off error. Finally, floating-point zero is identified by an all-0 exponent and significand, but the sign S may be 0 or 1. Note that as the tiny denormalized numbers are diminished, they eventually reach zero.

In summary, the number N represented by a 32-bit IEEE-standard, floating-point number has the following set of interpretations.

If $E = 255$ and $M \neq 0$, then $N =$ NaN.
If $E = 255$ and $M = 0$, then $N = (-1)^S\infty$.
If $0 < E < 255$, then $N = (-1)^S 2^{E-127}(1.M)$.
If $E = 0$ and $M \neq 0$, then $N = (-1)^S 2^{E-126}(0.M)$.
If $E = 0$ and $M = 0$, then $N = (-1)^S 0$.

The interpretation of 64-bit and larger floating-point numbers is similar.

[3] The term *unnormalized* applies to numbers with any value of E and a leading 0 instead of a leading 1 associated with their mantissas. Such numbers are encountered only as intermediate results during floating-point computations and are not relevant to the standard.

Typical of other floating-point number formats still in use is that of the IBM System/360-370. It consists of a sign bit S, a 7-bit exponent field E, and a mantissa field M containing 24, 56, or 112 bits. M is treated as a fraction, which with S forms a sign-magnitude number; there is no hidden leading 1. E is an integer in excess-64 code, corresponding to an exponent bias of 64. Unlike the IEEE 754 format where the base B of the representation is two, the System/360-370 has $B = 16$. Consequently, M is interpreted as a hexadecimal (base 16) number with every hexadecimal digit corresponding to 4 bits, and the exponent is treated as a power of 16. The value of a floating-point number in the normalized System/360-370 format is therefore given by

$$N = (-1)^S 16^{E-64}(0.M)$$

where M is a 6-, 14-, or 28-digit hexadecimal number. For example, the number 0.125×16^5 is encoded as

0 1000101 00100000...0000

Note that the left-most four bits 0010 of the mantissa represent the nonzero hexadecimal digit 2; hence the above number is normalized. The number zero is always represented by the all-0 word, making the floating-point representation of zero identical to the System/360-370 fixed-point (twos-complement) representation. There are no equivalents of the IEEE 754 standard's NaN, infinity, and denormalized formats. While most floating-point instructions are performed with automatic normalization of the results, a few may be specified without normalization, thus providing some of the advantages of denormalization. Due to the larger value of B being used, the System/360-370 32-bit format can represent numbers with magnitudes ranging from 5.40×10^{-79} to 7.24×10^{75} approximately.

.3.3 INSTRUCTION SETS

Next we turn to the representation, selection, and application of instruction sets. This topic embraces opcode and operand formats, the design of the instruction types to include in a processor's instruction set, and the use of instructions in executable programs.

3.3.1 Instruction Formats

The purpose of an instruction is to specify both an operation to be carried out by a CPU or other processor and the set of operands or data to be used in the operation. The operands include the input data or arguments of the operation and the results that are produced.

Introduction. Most instructions specify a register-transfer operation of the form

$$X_1 := op(X_1, X_2, \ldots, X_n)$$

which applies the operation *op* to n operands $X_1, X_2, \ldots, X_n$, where n ranges from zero to four or so. We can write the same instruction in the assembly-language notation

$$op \quad X_1, X_2, \ldots, X_n \tag{3.18}$$

which defines the operation and its operands by specific "fields" within the instruction word (3.18). The operation *op* is specified by a field called the *opcode* (*op*eration *code*). The n $X_1, X_2, \ldots, X_n$ fields are referred to as *addresses*. An address X_i typically names a register or a memory location that stores an operand value. In some instances X_i itself *is* the desired value, in which case it is called an *immediate* address.

To reduce instruction size and thereby reduce program storage space, it is common to specify only $m < n$ operands explicitly in the instruction; the remaining operands are implicit. The explicit address fields refer to general-purpose CPU registers or memory locations, while the implicit ones refer to special-purpose registers. If m is the normal maximum number of explicit main-memory addresses allowed in any processor instruction, the processor is called an *m-address machine*. Implicit input operands must be placed in known locations before the instruction that refers to them is executed.

Inside the computer, instructions are stored as binary words. There can be several different sizes and formats, depending on the instruction type. RISCs tend to have few instruction formats, while CISCs tend to have many to accommodate more opcode types and operand addressing methods. The Motorola 680X0 (Example 3.3) is a CISC microprocessor series with many different instruction formats and sizes, a sampling of which appear in Figure 3.27 [Motorola 1989]. Instruction length in the 680X0 varies from 2 to 10 bytes. The 2-byte opcode field of the 680X0 is often used to hold one or two 3-bit register addresses, blurring the distinction between opcode and operand.

In the 680X0 family, simple instructions are assigned short formats. For example, the add-register instruction

```
ADD.L    D1,D2                    (3.19)
```

denotes register-to-register addition of 32-bit (long word) operands, that is,

```
D2 := D2 + D1
```

where D1 and D2 are two of the 680X0's data registers (Figure 3.11). This instruction fits in the third 2-byte format F3 of Figure 3.27, which accommodates two register-address fields. A variant of the same two-address instruction can also refer to an operand in memory:

```
ADD.L    ADR1, D2                 (3.20)
```

This instruction specifies the memory-to-register addition operation

```
D2 := D2 + M(ADR1)
```

and so combines the load and add operations. It uses the 6-byte format F6 to contain the 4-byte immediate address field ADR1. It also requires a memory access to obtain one of its input operands, the 4-byte long word with start address ADR1. Note that the binary (machine language) opcodes corresponding to (3.19) and (3.20) have to be different to distinguish their operand types.

The longest (10 byte) format F8 of the 680X0 is employed by such memory-to-memory move instructions as

```
MOVE.B   ADR1,ADR2
```

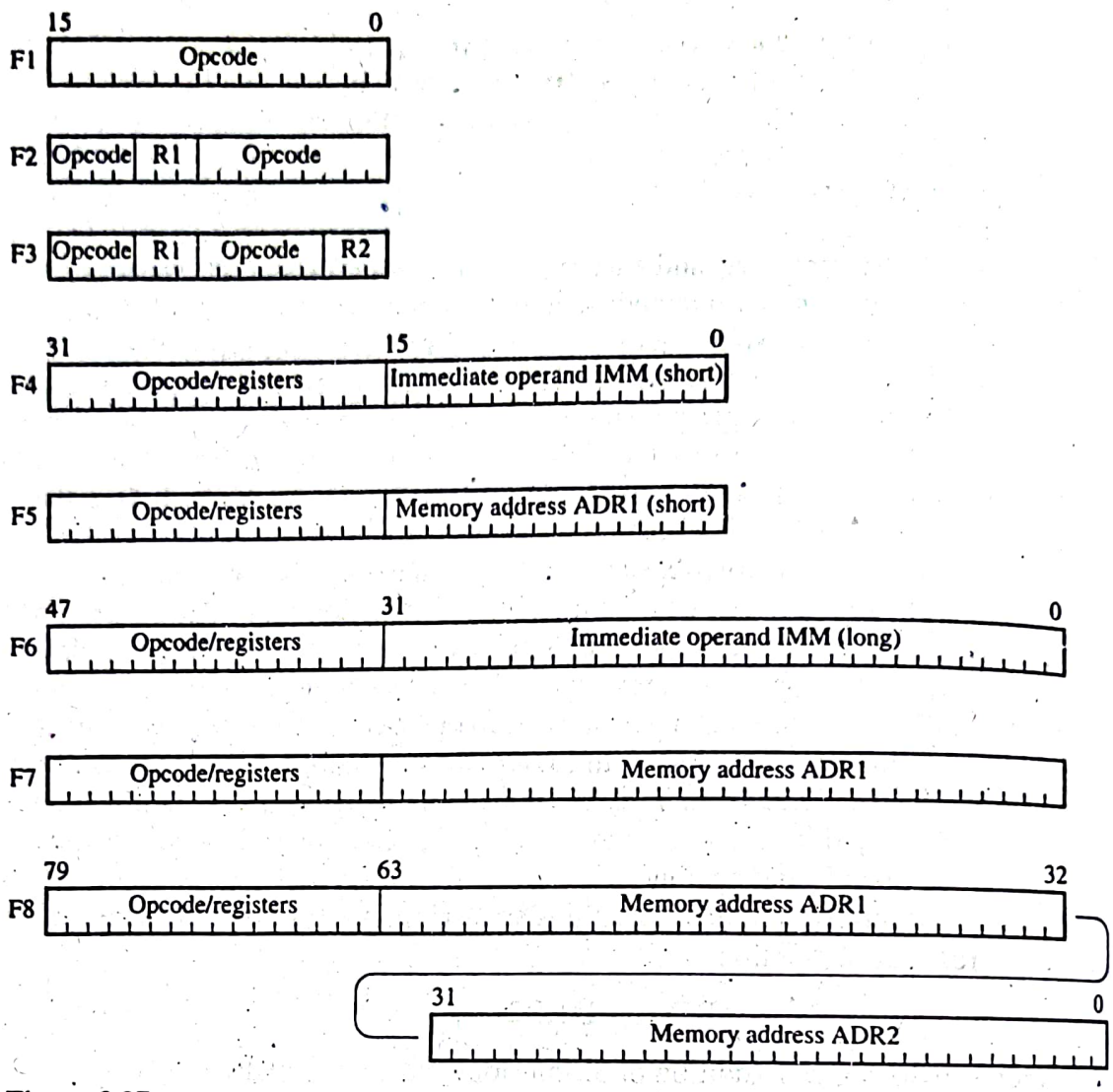


Figure 3.27
A selection of instruction formats of the Motorola 680X0.

which copies (via the CPU) the byte stored in memory location ADR1 to memory location ADR2, that is,

$$M(ADR2) := M(ADR1)$$

RISC formats. The instruction formats of the 680X0 accommodate a wide variety of operations and addressing modes. They also try to reduce object-program size by encoding the more common instructions in short formats and the less frequent and more complex instructions in longer formats. Since such instructions are often primitives in high-level programming languages, they serve to reduce both program length and what has been called the *semantic gap* between the user and the computer languages.

Complex instructions lead to several difficulties, which RISCs with their smaller and streamlined instruction sets attempt to minimize.

- The many instruction types and formats of a CISC complicate the program-control unit that decodes instruction opcodes and issues the control signals that

govern their execution. The 68020 employs a large, two-level microprogrammed PCU (Figure 3.11), whereas the ARM6 has a smaller hardwired circuit as its PCU.

- Fast, single-cycle instruction execution is harder to achieve with a complex instruction set, and it is more difficult for a compiler to optimize object-code performance.

A typical RISC employs instructions of fixed length. Memory addressing is restricted to load and store instructions, so the operands of most instructions are register addresses, which are short and easy to accommodate in a one-word format. Figure 3.28 shows the single 32-bit format used by instructions in the RISC 1 computer, a prototype RISC machine designed by David A. Patterson and his colleagues at the University of California, Berkeley, around 1980 [Patterson and Séquin 1982]. Most of the 31 instruction types defined for the RISC 1 perform register-to-register operations of the form

$$\text{Rd} := \text{F}(\text{Rs},\text{S2}) \tag{3.21}$$

where Rd is the destination register, Rs is the first source register, and the rightmost 5 bits of S2 define a second-source register. If bit 13 of the instruction is set to one, then S2 is interpreted as an immediate address, that is, as a 13-bit constant. The instructions of the ARM6 microprocessor (Example 3.2), like those of the RISC 1, are all 32 bits long, but they come in a large and CISC-like number of formats [Furber 1989].

Operand extension. A CPU is designed primarily to process data words and addresses of one specific length—a 32-bit word in the case of the ARM6 and RISC 1—although some instructions handle longer or shorter operands. Numerical operands can be unsigned binary number words, such as memory addresses, or signed data words that employ twos-complement code. (Recall from section 3.2.1 that the same arithmetic circuits can be used with unsigned and twos-complement numbers.) Instructions often contain operand fields that are shorter than the standard word size, for example, the 13-bit immediate address field S2 in the RISC 1 format of Figure 3.28. This problem is unavoidable in RISC instruction sets where the instruction length and the standard word size are the same. Consequently, a systematic method is needed to extend short operand values to full-size, signed or unsigned numbers.

When a short m-bit, twos-complement number is used in an n-bit arithmetic operation where $n > m$, a technique called *sign extension* is employed. This techniques replicates the left-most bit s of the short operand N, which corresponds to its sign bit, $n - m$ times and attaches $s^{n-m} = ss\ldots s$ to the left side of N. Sign extension

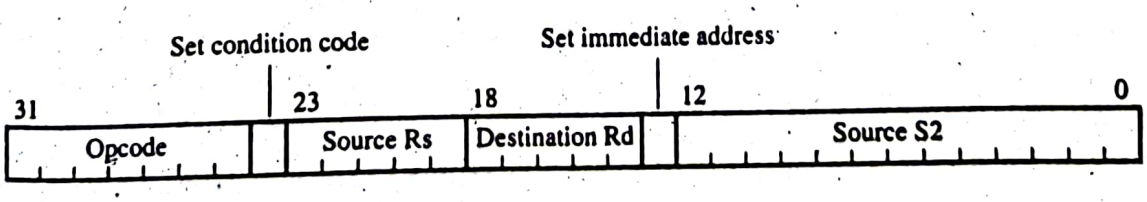


Figure 3.28
Instruction format of the Berkeley RISC 1.

$$N = 10101\ 01010101 \tag{3.22}$$

in the S2 field of Figure 3.28 to the 32-bit word

$$N_{\text{sign-extended}} = 11111111\ 11111111\ 11110101\ 01010101 \tag{3.23}$$

In this case $s = 1$ and $n - m = 19$. If s were 0, then sign extension would precede N by 19 leading 0s. The point of sign extension is that it does not change the numerical value of a twos-complement number. For instance, both (3.22) and (3.23) represent the *same* negative integer, namely, $-2{,}646_{10}$, in twos-complement code, as can readily be verified. Sign extension maintains a number's correct sign and magnitude because it introduces only numerically insignificant leading 0s (positive numbers) or insignificant leading 1s (negative numbers). If N is to be treated as an unsigned binary number, then it is always extended by leading 0s, independent of the value of s. This technique has been called *zero extension*. Applying zero extension to (3.22) yields

$$N_{\text{zero-extended}} = 00000000\ 00000000\ 00010101\ 01010101$$

Next we ask: How is an n-bit memory address, which is a long (typically 32-bit) unsigned integer, constructed from a short m-bit address field, when $n > m$? Zero extension alone is sometimes used for this purpose, but it does not allow the m-bit address to refer to all 2^n possible addresses. The usual solution found in CISCs as well as in RISCs is to treat a short memory address as a modifier, or *offset*, which is added (in zero-extended form) to a full-length memory address stored in a designated CPU register, called a *base register*. The RISC 1 uses its Rs register for this purpose, with S2 serving as the offset. The following store-byte instruction

$$\text{STB}\quad \text{Rs,Rd(S2)} \tag{3.24}$$

is designed to copy the byte from the right end of register Rs to the memory location whose address is $\text{Rd} + \text{S2}_{\text{zero-extended}}$. In practice, sign extension is often implicit and $\text{Rd} + \text{S2}_{\text{zero-extended}}$ is written simply as Rd +S2. Hence (3.24) is equivalent to

$$\text{M(Rd + S2)} := \text{Rs[24:31]}$$

The final memory address Rd + S2 is an example of an *effective address*. As we will see shortly in our discussion of addressing modes, many other techniques are employed for constructing effective addresses.

EXAMPLE 3.5 INSTRUCTION FORMATS OF THE MIPS RX000 SERIES [KANE AND HEINRICH 1992]. MIPS Computer Systems (now a division of Silicon Graphics) introduced the MIPS RX000 series of microprocessors in 1986. The first members of the series, the MIPS R2000 and R3000, are 32-bit machines that have most of the classic RISC features: a streamlined instruction set, a load/store architecture, and an instruction pipeline to support a performance target of one instruction completed every clock cycle. Later RX000 machines, such as the R10000 announced in 1994, add various extensions to the "MIPS I" architecture implemented in the R2000 and R3000; we will confine our discussion to the MIPS I case.

The RX000 is noteworthy for its simple and regular instruction formats, which we now examine in detail. As seen from Figure 3.29, all the RX000 instructions are one word (32 bits) in length and contain a 6-bit opcode in a fixed position. The remaining

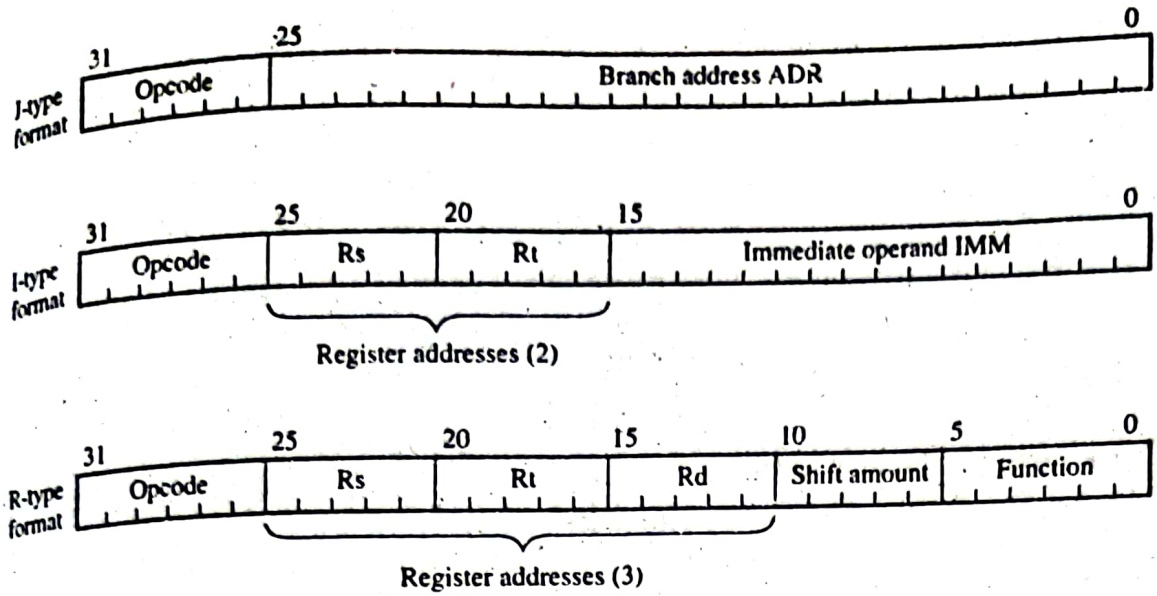


Figure 3.29
Instruction formats of the MIPS RX000.

26 bits are used in various ways, depending on the instruction type. Any operands included in the instruction must be less than a full word in length, so some way is needed to extend them to a full-size memory address or a twos-complement number.

In the case of a J-type (jump or branch) instruction, the 26 operand bits form a memory address ADR, which is the target or branch address. For example, a simple unconditional branch instruction has the J-type format

$$\text{J ADR} \tag{3.25}$$

meaning **go to** ADR. Since RX000 memory addresses are 32 bits long, the PCU must extend the 26-bit address field ADR in (3.25) to 32 bits. This is done automatically by the following two-step process:

```
Temp := PC[31:28].ADR.00;
PC := Temp;
```

First the four high-order bits from the program counter PC are placed in front of ADR and 00 is appended to it. Then the resulting 32-bit word is made the new contents of PC.

The above address-extension method confines the possible branch addresses to a 2^{26}-word region of memory space near the location of the current branch instruction. However, this is not as restrictive as it might appear. First of all, recall that a 32-bit memory address refers to just one byte. Only 2^{30} instructions can be placed in a 2^{32}-byte memory, so only 30 bits are really needed to locate an instruction. The RX000 and similar machines always assign instructions to memory word locations with addresses that end in 00; that is, all instructions are aligned with the natural word boundaries in M. Moreover, while the 26-bit address field ADR is still 4 bits short of 30, the size of the accessible region for branching ($2^{26} \approx 6.71 \times 10^7$ different addresses) is more than adequate for most programming purposes—and can be increased by software means, if necessary.

The other two formats shown in Figure 3.29 specify register addresses using either two or three 5-bit fields. The RX000 has $2^5 = 32$ general-purpose registers in its register file, so register addresses can be fully specified with no difficulty. The second

(I type) format is used by ALU-immediate instructions such as

ADDI Rs,Rt,IMM

which adds the contents of the instruction's immediate address field, that is, bits 15:0 of the instruction, to the contents of register Rs and places the result in register Rt. To convert the immediate operand IMM from 16 to 32 bits, it is sign-extended to 32 bits by duplicating its left-most bit to obtain bits 31:16.

The third (R type) format of the RX000 is used by data-processing instructions that have a natural three-address format to define operations of the form $X_1 := op(X_2, X_3)$. For instance, the add-register instruction

ADD Rd,Rs,Rt

performs the 32-bit addition

Rd := Rs + Rt

using the contents of the named registers. Since the register addresses occupy only 15 bits of the instruction format, the remaining 11 bits are used in various ways to increase (and complicate) the range of operations that can be performed. In effect, they serve as extensions to the opcode. For example, there are six shift-register instructions, all of which use instruction bits 10:6 to specify the amount by which the target register's contents are to be shifted. The shift-left logical instruction

SLL Rd,Rt,Shamt

shifts the contents of register Rt left by Shamt (shift amount) bits; it inserts 0s in the vacated positions on the right and places the result in Rd. In other words,

$$Rt := Rd[31-Shamt:0].0^{Shamt}$$

where 0^k denotes a string of k 0s.

For load and store instructions, the RX000 uses the typical RISC technique of providing a short address in the instruction, which serves as an offset to a full-length address stored in a CPU register. The I-type format of Figure 3.29 is used for load and store instructions. In this case Rs serves as the base register, and Rt serves as the data source (for store) or destination (for load). The instruction that loads a word into the CPU has the assembly-language format

LW Rt, IMM(Rs)

which causes the 16-bit immediate address IMM, that is, the offset, to be sign-extended to 32 bits and added to the contents of Rs to form the effective address. This address is then used to read a word of data from M into register Rt. In HDL terms

Rt := M(Rs + IMM)

Addressing modes. The purpose of an address field is to point to the current value $V(X)$ of some operand X used by an instruction. This value can be specified in various ways, which are termed *addressing modes*. The addressing mode of X affects the following issues:

- The speed with which $V(X)$ can be accessed by the CPU.
- The ease with which $V(X)$ can be specified and altered.

Access speed is influenced by the physical location of $V(X)$—normally the CPU or the external memory M. Operand values located in CPU registers, such as the

general-register file and the program counter PC, can be accessed faster than operands in M. It is therefore usual to favor instructions that address CPU registers, both in the design of instruction sets and in their use in computer programs. An operand's accessibility is also affected by the directness of its addressing mode: The address field X itself can be $V(X)$, it can specify directly the location of $V(X)$, or it can identify a location that specifies directly the location of $V(X)$. We can thus distinguish the number of *levels of indirection* associated with an address. The advantage of indirection, as we will see, is increased programming flexibility. We can achieve further flexibility by providing addresses that are automatically altered or *indexed*, for example, to step through an array of consecutive addresses.

If the value $V(X)$ of the target operand is contained in the address field itself, then X is called an *immediate operand* and the corresponding addressing mode is *immediate addressing*. By implication X is a constant, since it is very undesirable to modify instruction fields during execution.[4] More often than not, X is a variable in the usual mathematical sense, and the corresponding address field identifies the storage location that contains the required value $V(X)$. Thus X corresponds to a variable, and its value $V(X)$ can be varied without modifying the instruction address field. Operand specification of this type is called *direct addressing*.

The addressing modes of the operands appearing in a machine-language instruction, which can vary from operand to operand, are defined in the instruction's opcode. Some assembly languages allow addressing modes to be similarly defined by distinct opcodes. For example, the assembly language of the Intel 8085 series has the opcode MOV (move) to specify data transfers involving direct addressing only. Therefore, the register-to-register transfer $A := B$, for instance, is specified by

```
MOV  A,B                                   (3.26)
```

The A and B operands of (3.26) are considered to be directly addressed, since the contents of the named registers are the desired operand values. In contrast, to specify the operation A := 99, where 99 is an immediate operand, the 8085 instruction

```
MVI  A,99                                  (3.27)
```

with the opcode MVI (*move immediate*) must be used. Note that (3.27) uses both the direct and immediate addressing modes.

Most assembly languages take a different approach by specifying the addressing modes in the operand fields. For example, the Motorola 680X0 equivalents of (3.26) and (3.27), with D1 = A and D2 = B are

```
MOVE  D2,D1
```

and

```
MOVE  #99,D1                               (3.28)
```

respectively. (Note that the Motorola operand order is reversed with respect to the Intel convention.) In (3.28) the prefix # indicates that the immediate addressing mode is to be used for the operand in question. Deleting the # from (3.28) causes

[4]Self-modifying programs like the IAS code shown in Figure 1.15 (section 1.2.2) reflect the inadequacy of the addressing modes available in the earliest computers.

the first operand to refer to the data in memory location 99, that is, M(99), which would be an instance of direct memory addressing.

It is sometimes useful to change the location (as opposed to the value) of X without changing the address fields of any instructions that refer to X. This may be accomplished by *indirect addressing*, whereby the instruction contains the address W of a storage location, which in turn contains the address X of the desired operand value $V(X)$. By changing the contents of W, the address of the operand value required by the instruction is effectively changed. While direct addressing requires only one fetch operation to obtain an operand value, indirect addressing requires two. Figure 3.30 illustrates these different ways of specifying operands in the case of three load instructions that transfer the number 999 to the CPU register AC.

The ability to use all addressing modes in a uniform and consistent way with all opcodes of an instruction set or assembly language is a desirable feature termed *orthogonality*. Orthogonal instruction sets simplify programming both by reducing the number of distinct opcodes needed and by simplifying the rules for operand address specification. Many CISC computers like the 680X0 have little orthogonality, since processor costs can be reduced (at the expense of programming costs) by restricting instructions to a few frequently used addressing modes that vary from instruction to instruction.

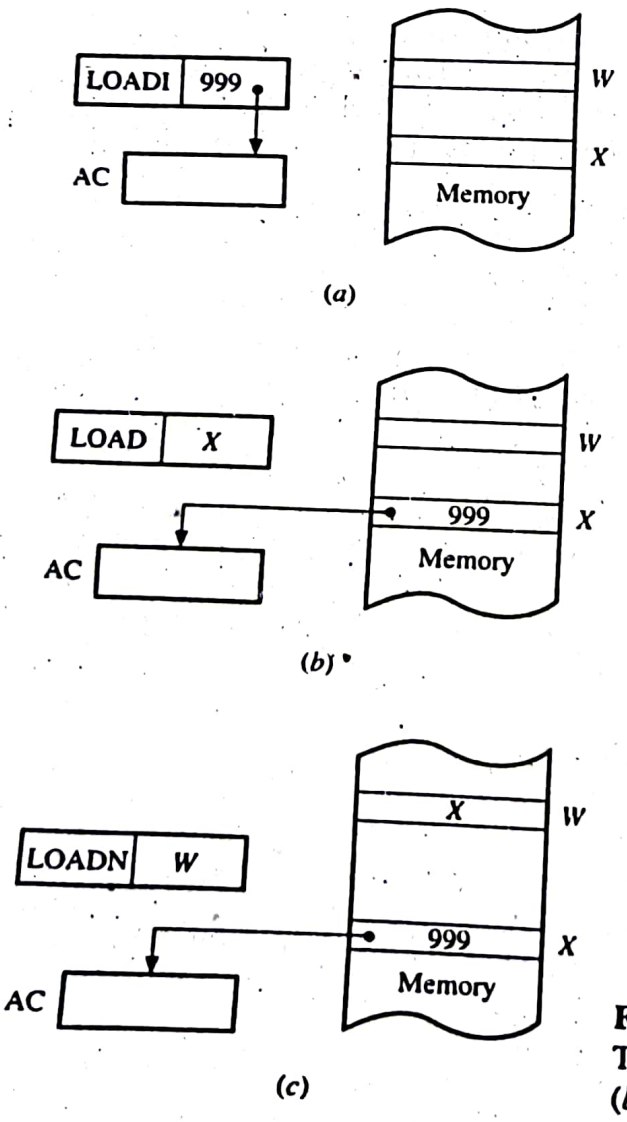


Figure 3.30
Three basic addressing modes: (*a*) immediate; (*b*) direct; (*c*) indirect.

Relative addressing. *Absolute addressing*, conceptually the simplest mode of direct address formation, requires the complete operand address to appear in the instruction operand field. This address is used without modification (except, perhaps, zero or sign extension in the case of a short address field) to access the desired data item. Frequently, only partial addressing information is included in the instruction, so the CPU must construct the complete (absolute) address. One of the commonest address construction techniques is *relative addressing*, in which the operand field contains a relative address, also called an *offset* or *displacement D*. The instruction also implicitly or explicitly identifies other storage locations R_1, $R_2,\ldots,R_k$ (usually CPU registers) containing additional addressing information. The *effective address A* of an operand is then some function $f(D,R_1,R_2,\ldots,R_k)$. In most cases of interest, each operand is associated with a single address register R from a set of general-purpose address registers, and A is computed by adding D to the contents of R, that is,

$$A := R + D$$

R may also be a special-purpose address register such as the program counter PC.

There are several reasons for using relative addressing.

1. Since all the address information need not be included in the instructions, instruction length is reduced.
2. By changing the contents of R, the processor can change the absolute addresses referred to by a block of instructions B. This address modification permits the processor to move (relocate) the entire block B from one region of main memory to another without invalidating the addresses in B. When used in this way, R may be referred to as a *base register* and its contents as a *base address*.
3. R can be used for storing indexes to facilitate the processing of indexed data. In this role R is called an *index register*. The indexed items $X(0)$, $X(1),\ldots,X(k)$ are stored in consecutive addresses in memory. The instruction-address field D contains the address of the first item $X(0)$, while the index register R contains the index i. The address of item $X(i)$ is $D + R$. By changing the contents of the index register, a single instruction can be made to refer to any item $X(i)$ in the given data list.

The main drawbacks of relative addressing are the extra logic circuits and processing time needed to compute addresses.

So far we have assumed that each operand is a single memory word and can therefore be specified by a single address. If an instruction must process variable-length data consisting of many words, each operand specification is divided into two parts: an address field that points to the location of the first word of the operand and a length field L that indicates the number of words in the operand. The CPU automatically increments the instruction address field as successive words of the operand are accessed. The access is complete when L words have been accessed.

Indexed items are frequently accessed sequentially so that a reference to $X(k)$ stored in memory location A is immediately followed by a reference to $X(k + 1)$ or $X(k-1)$ stored in location $A + 1$ or $A - 1$, respectively. To facilitate stepping through a sequence of items in this manner, addressing modes that automatically increment or decrement an address can be defined; the resulting address-modification process

is called *autoindexing*. In the case of the Motorola 680X0 series [Motorola 1989], the address field –(A3) appearing in an assembly-language instruction indicates that the contents of the designated address register A3 should be decremented automatically before the instruction is executed; this process is called *predecrementing*. Similarly, (A3)+ specifies that A3 should be incremented automatically after the current instruction has been executed (*postincrementing*). In each case the amount of the address increment or decrement is the length in bytes of the indexed operands.

Most processors have only a few, simple addressing modes for CPU registers, principally direct and immediate addressing. Immediate addresses represent data values that come with the instruction fetch and are placed in the instruction register IR. In *register direct addressing*, the address (name) *R* of the register containing the desired value *V*(*R*) appears in the instruction. The Motorola 680X0 instruction

MOVE #99,D1

which means "move the constant 99 to data register D1," uses immediate addressing for 99 and register direct (or simply direct) addressing for D1.

The term *register indirect addressing* refers to indirect addressing with a register *R* name in the address field. It is often used to access memory, in which case *R* becomes a memory address register. For example,

MOVE.B (A0), D1

uses parentheses to indicate that (A0) is an indirect address involving the 680X0's A0 addresss register. This move-byte instruction—the opcodes's .B suffix specifies a 1-byte operand—corresponds to

D1[7:0] := M(A0)

and copies the byte addressed by A0 into the low-order byte position of data register D1. (The other three bytes of D1 are unchanged.) An extension of this addressing mode is *register indirect with offset*, which can also be viewed as a type of base or indexed addressing. This mode is the only memory addressing mode employed by the MIPS RX000 series (Example 3.5). The RX000's store-word instruction, for example, is written as

SW Rt, OFFSET(Rs) (3.29)

where Rs is the base register and OFFSET is a number acting as an (immediate) offset operand. Instruction (3.29) is equivalent to the HDL statement

M(Rs + OFFSET) := Rt

where the offset is sign-extended before adding it to Rs to obtain the effective address Rs + OFFSET. The PowerPC has two addressing modes: register indirect with offset as described above (but called *register indirect with immediate index*) and a second mode (called *register indirect with index*) in which the effective address is Rs + Ri, where Ri is a register name.

The Motorola 680X0, like other CISC-style architectures, has many addressing modes, including the following: immediate, register direct, register indirect, register indirect with postincrement, register indirect with predecrement, register indirect with offset, register indirect with index, absolute short, absolute long, PC

with offset, and PC with index. Its autoindexing features are illustrated in the following example.

EXAMPLE 3.6 STACK CONTROL IN THE MOTOROLA 680X0 [GILL, CORWIN AND LOGAR 1987; MOTOROLA 1989]. A *stack* is a sequence of storage locations that are accessible from only one end referred to as the *top of the stack*. A write operation addressed to a stack, termed a *push* operation, stores a new item at the top of the stack, while a read operation, termed a *pop* operation, removes the item stored at the top of the stack. Push or pop changes the position of the stack top by an amount that depends on the length of the operand pushed or popped. A stack is controlled by an address register called the *stack pointer* SP. This register stores the address of the last operand placed in the stack; that address is automatically adjusted after a push or pop operation so that SP contains the address of the new stack top.

Some computers—the Intel 80X86, for example—have special instructions and hardware for handling stacks that are intended as communication areas for program-control instructions like call and return. A few early computers such as the Burroughs B6500/7500 even employed stacks in place of general-register files; see Example 1.5 (section 1.2.3). The Motorola 680X0 has no explicit hardware for stack support, but, as we now show, its various addressing modes make it easy to treat any contiguous region of its external memory M as a stack.

Suppose that the programmer designates the address register A2 of the 680X0 to be a stack pointer and that the stack grows toward the low addresses of M. To push the contents of a data register, say, D6, into the stack requires the single instruction

$$\text{MOVE.L} \quad \text{D6}, -(\text{A2}) \tag{3.30}$$

The input operand is the 4-byte contents of D6, which is directly addressed in (3.30), while the output operand, which is the new contents of the top of the stack, is designated by –(A2), which denotes indirect addressing with predecrementing using address register A2. This push instruction is equivalent to the following HDL operations:

$$\text{A2} := \text{A2} - 4; \quad \text{M(A2)} := \text{D6};$$

Figure 3.31 shows the state of the affected parts of the CPU and M immediately before (Figure 3.31*a*) and immediately after (Figure 3.31*b*) execution of instruction (3.30). Observe how the data bytes are stored in M according to the big-endian convention.

It is easily seen that the pop instruction corresponding to (3.30) is

$$\text{MOVE.L} \quad (\text{A2})+, \text{D6} \tag{3.31}$$

which is equivalent to

$$\text{D6} := \text{M(A2)}; \quad \text{A2} := \text{A2} + 4;$$

In this case the operand (A2)+ employs the register indirect with postincrement addressing mode.

Number of addresses. Some computers, notably CISCs like the 680X0, have instructions of several different lengths containing various numbers of addresses. A source of controversy in the early days was the question of how many explicit operand addresses to include in instructions. Clearly the fewer the addresses, the shorter the instruction format needed. However, limiting the number of addresses also limits the range of operations that an instruction can perform. Roughly speaking, fewer addresses mean more primitive instructions and therefore longer

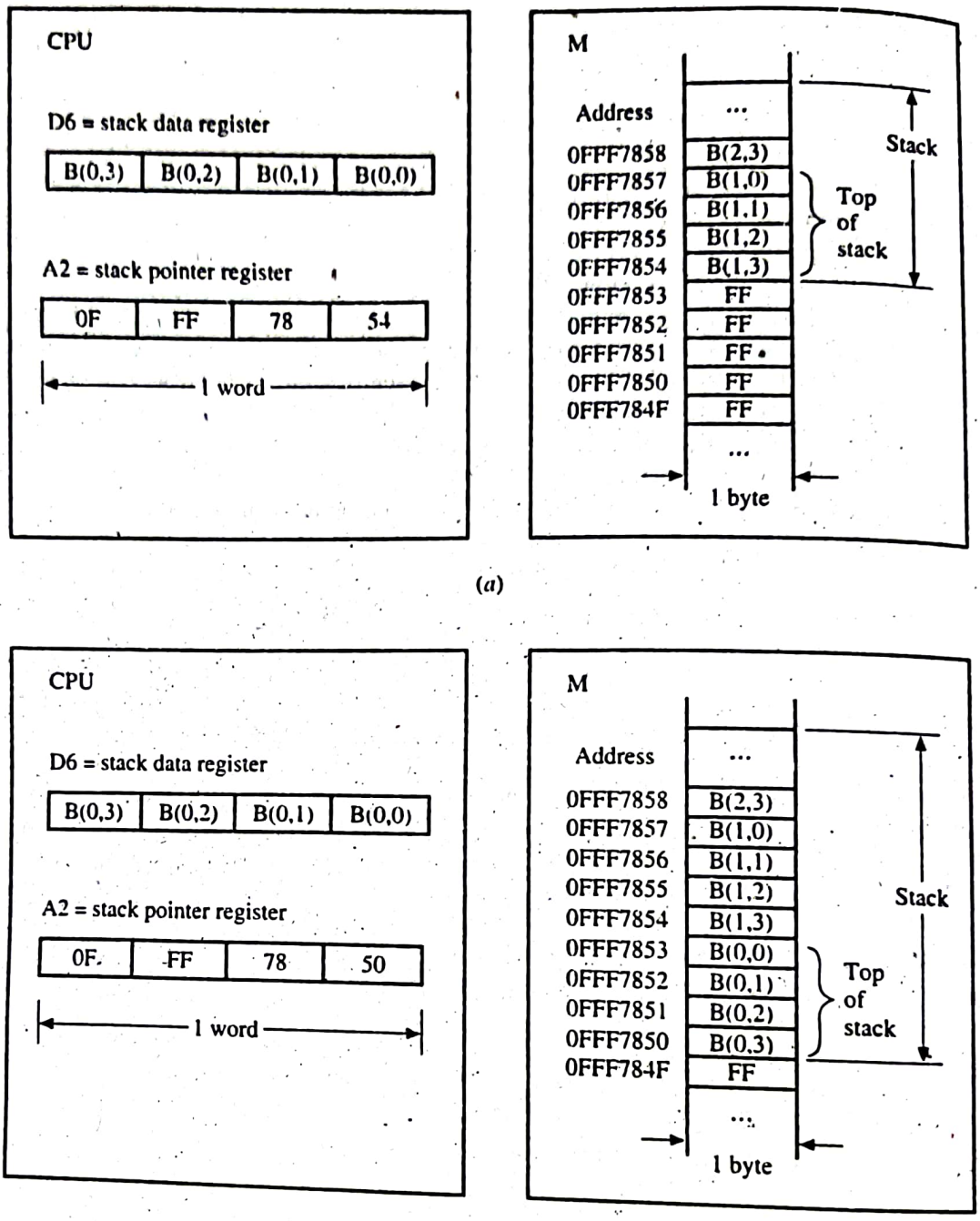


Figure 3.31
State of the Motorola 680X0 (*a*) immediately before and (*b*) immediately after execution of the push instruction MOVE.L D6,–(A2).

programs to perform a given task. While the storage requirements of shorter instructions and longer programs tend to balance, larger programs require longer execution times. On the other hand, long instructions with multiple addresses require more complex decoding and processing circuits. RISC instructions, with the exception of load and store, contain short register addresses only, so two or three addresses can be accommodated within a short and fixed-length instruction word.

Most instructions require no more than three distinct operands. For example, the fundamental arithmetic operations—addition, subtraction, multiplication, and division—require three operands: two input operands and one output operand. A three-address instruction can therefore specify all needed operands. For example, the three-address add instruction

ADD Z, X, Y

means add the contents of memory locations X and Y and place the result in location Z; that is, $Z := X + Y$. A one-address add instruction has the format

ADD X

The unspecified operands are assumed to be stored in fixed locations such as the accumulator AC, in which case the instruction specifies the operation AC := AC + X. In the case of a two-address instruction, the accumulator is used to store the result (the sum) only.

ADD X, Y

has the typical interpretation AC := $X + Y$. Another possibility is to use one address, say, X, to store both the addend X and the sum as follows: $X := X + Y$. In the latter case the addition operation destroys the X operand. Figures 3.32*a*, *b*, and *c* show how processors that employ one-address, two-address, and three-address instructions, respectively, might implement the operation

$$X := A \times B + C \times C \tag{3.32}$$

where the four operands A, B, C, and X are assumed to be stored in external memory.

A few computers have been designed so that most instructions contain no explicit addresses; they can be called *zero-address* machines; see also Example 1.5. Addresses are eliminated by storing operands in a push-down stack. All operands used by a zero-address instruction are required to be in the top locations in the stack. For example, the addition $X + Y$ is invoked by an instruction such as

ADD

that causes the top two operands, which should be X and Y, to be removed from the stack and added. The resulting sum $X + Y$ is then placed at the top of the stack. A stack pointer automatically keeps track of the stack top. Push and pop instructions are needed to transfer data to and from the stack. PUSH X causes the contents of X to be placed at the top of the stack. POP X causes the top word in the stack to be transferred to location X. Note that PUSH and POP are not themselves zero-address instructions; as implemented by (3.30) and (3.31), for instance, they are two-address instructions. Figure 3.33 shows how a program for (3.32) might be constructed for a zero-address, stack machine.

3.3.2 Instruction Types

We now turn to the question: What types of instructions should be included in a general-purpose processor's instruction set? We are concerned with the instructions

Instruction	Comments
LOAD A	Transfer A to accumulator AC.
MULTIPLY B	AC := AC × B
STORE T	Transfer AC to memory location T.
LOAD C	Transfer C to accumulator AC.
MULTIPLY C	AC := AC × C
ADD T	AC := AC + T
STORE X	Transfer result to memory location X.

(*a*) One-address machine

Instruction	Comments
MOVE T,A	T := A
MULTIPLY T,B	T := T × B
MOVE X,C	X := C
MULTIPLY X,C	X := X × C
ADD X,T	X := X + T

(*b*) Two-address machine

Instruction	Comments
MULTIPLY T,A,B	T := A × B
MULTIPLY X,C,C	X := C × C
ADD X,X,T	X := X + T

(*c*) Three-address machine

Figure 3.32
Programs to execute the operation $X := A \times B + C \times C$ in one-address, two-address, and three-address processors.

that are in the processor's machine language. All processors have a well-defined machine language, and some implement a lower-level "micromachine" language specified by microinstructions. A typical machine instruction defines one or two register transfer (micro) operations, and a sequence of such instructions is needed to implement a statement in a high-level programming language such as C. Because of the complexity of the operations, data types, and syntax of high-level languages, few attempts have been made to construct computers whose machine language directly corresponds to a high-level language. As noted earlier, there is a semantic gap between problem-specification languages and the machine instruction set that implements them, a gap that language-translation programs such as compilers and assemblers must bridge.

The requirements to be satisfied by an instruction set can be stated in the following general, but rather imprecise, terms:

Instruction	Comments
PUSH A	Transfer A to top of stack.
PUSH B	Transfer B to top of stack.
MULTIPLY	Remove A,B from stack and replace by $A \times B$.
PUSH C	Transfer C to top of stack.
PUSH C	Transfer second copy of C to top of stack.
MULTIPLY	Remove C,C from stack and replace by $C \times C$.
ADD	Remove $C \times C$, $A \times B$ from stack and replace by their sum.
POP X	Transfer result from top of stack to X.

Figure 3.33
Program to execute $X := A \times B + C \times C$ in a zero-address, stack processor.

- It should be *complete* in the sense that we should be able to construct a machine-language program to evaluate any function that is computable using a reasonable amount of memory space.
- It should be *efficient* in that frequently required functions can be performed rapidly using relatively few instructions.
- It should be *regular* in that the instruction set should contain expected opcodes and addressing modes; for example, if there is a left shift, there should be a right shift.
- To reduce both hardware and software design costs, the instructions may be required to be *compatible* with those of existing machines—previous members of the same computer family, for instance.

Because of the wide variation in CPU architectures between different computer families, standard machine or assembly languages do not exist. There are, nevertheless, broad similarities between all instruction sets, which go back to the IAS computer and other early machines.

Completeness. A function $f(x)$ is said to be *computable* if it can be evaluated in a finite number of steps by a Turing machine (see section 1.1.1). While real computers differ from Turing machines in having only a finite amount of memory, they can, in practice, evaluate any computable function to a reasonable degree of approximation. When viewed as instruction-set processors, Turing machines have a very simple instruction set. In our discussion of Turing machines, we defined four instruction types: write, move tape one square to the left, move tape one square to the right, and halt, all of which are conditional on the control processor's state. It follows that complete instruction sets can be constructed for finite-state machines using equally simple instruction types. In fact, computers have been proposed that employ only a single type of instruction; see problem 3.44. While very small instruction sets require simple, and therefore inexpensive, logic circuits to implement them, they lead to excessively complex programs. There is therefore a fundamental trade-off between processor simplicity and programming complexity.

Instructions are conveniently divided into the following five types:

1. *Data-transfer* instructions, which copy information from one location to another either in the processor's internal register set or in the external main memory.
2. *Arithmetic* instructions, which perform operations on numerical data.
3. *Logical* instructions, which include Boolean and other nonnumerical operations.
4. *Program-control* instructions, such as branch instructions, which change the sequence in which programs are executed.
5. *Input-output* (IO) instructions, which cause information to be transferred between the processor or its main memory and external IO devices.

These types are not mutually exclusive. For example, the arithmetic instruction $A := B + C$ implements the data transfer $A := B$ when C is set to zero.

Figure 3.34 lists representative instructions from the five types defined above, which have been culled from the instruction sets of various computers. The data-transfer instructions, particularly load and store, are the most frequently used instructions in computer programs, despite the fact that they involve no explicit computation. The arithmetic instructions cover a wide range of operations and are sometimes used as a rough measure of the complexity of an instruction set. The logical instructions include the word-based Boolean operations, as well as operations that have no obvious numerical interpretation. The major branch instructions are jump (un)conditionally and the call and return instructions used for subroutine linkage. The simplest IO instructions are data-transfer instructions addressed to IO ports, which transfer one or more words between an IO port and either the CPU or M. If the CPU delegates control of IO operations to an IO processor (IOP), the CPU needs instructions that enable it to supervise the execution of IO programs by the IOP. Instructions that are specific to particular IO devices, such as REWIND TAPE, PRINT LINE, and SCAN KEYBOARD, are treated as data by the CPU and IOP and are interpreted as instructions only by the IO devices to which they are transferred.

The completeness of an instruction set can be demonstrated informally by showing that it can program certain key operations in each of the five instruction groups. It must be possible to transfer a word between the processor and any memory location. It must be possible to add two numbers, so an add instruction is included in most instruction sets. Other arithmetic operations can readily be programmed using addition. As noted in section 3.2.2, subtraction of twos-complement numbers requires addition and logical complementation (NOT) only. More complex arithmetic operations such as multiplication, division, and exponentiation can be programmed using addition, subtraction, and shifting, as in Example 2.7. If a logically complete set of Boolean operations such as {AND,NOT} is in the instruction set, then any other Boolean operation can be programmed. Branching requires at least one conditional branch instruction that tests some stored quantity and alters the instruction execution sequence based on the test outcome. An unconditional branch can easily be realized by a conditional branch instruction.

RISC versus CISC. While an instruction set that is limited to two or three instructions is impractical, there is no agreement about the appropriate size or membership of a general-purpose instruction set. Early computers like the IAS had a small and simple instruction set forced by the need to minimize the amount of

Type	Operation name(s)	Description
Data transfer	MOVE	Copy word or block from source to destination.
	LOAD	Copy word from memory to processor register.
	STORE	Copy word from processor register to memory.
	SWAP (EXCHANGE)	Swap contents of source and destination.
	CLEAR	Transfer word of 0s to destination.
	SET	Transfer word of 1s to destination.
	PUSH	Transfer word from source to top of stack.
	POP	Transfer word from top of stack to destination.
Arithmetic	ADD	Compute sum of two operands.
	ADD WITH CARRY	Compute sum of two operands and a carry bit.
	SUBTRACT	Compute difference of two operands.
	MULTIPLY	Compute product of two operands.
	DIVIDE	Compute quotient (and remainder) of two operands.
	MULITPLY AND ADD	Compute product of two operands; add it to a third operand.
	ABSOLUTE	Replace operand by its absolute value.
	NEGATE	Change sign of operand.
	INCREMENT	Add 1 to operand.
	DECREMENT	Subtract 1 from operand.
	ARITHMETIC SHIFT	Shift operand left (right) with sign extension.
Logical	AND	Perform the specified logical operation bitwise.
	OR	
	NOT	
	EXCLUSIVE-OR	
	LOGICAL SHIFT	Shift operand left (right) introducing 0s at end.
	ROTATE	Left- (right-) shift operand around closed path.
	CONVERT (EDIT)	Change data format, for example, from binary to decimal.
Program control	JUMP (BRANCH)	Unconditional transfer; load PC with specified address.
	JUMP CONDITIONAL	Test specified conditions; if true, load PC with specified address.
	JUMP TO SUBROUTINE (BRANCH-AND-LINK)	Place current program control information including PC in known location, for example, top of stack; jump to specified address.
	RETURN	Restore current program control information including PC from known location, for example, from top of stack.
	EXECUTE	Fetch operand from specified location and execute as instruction; note that PC is not modified.
	SKIP CONDITIONAL	Test specified condition; if true, increment PC to skip next instruction.
	TRAP (SOFTWARE INTERRUPT)	Enter supervisor mode.
	TEST	Test specified condition; set flag(s) based on outcome.
	COMPARE	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome.

Figure 3.34
List of common instruction types.

Type	Operation name(s)	Description
Program control	SET CONTROL VARIABLES	Large class of instructions to set controls for protection purposes, interrupt handling, timer control, and so forth (often privileged).
	WAIT (HOLD)	Stop program execution; test a specified condition continuously; when the condition is satisified, resume instruction execution.
	NO OPERATION	No operation is performed, but program execution continues.
Input-output	INPUT (READ)	Copy data from specified IO port to destination, for example, output contents of a memory location or processor register.
	OUTPUT (WRITE)	Copy data from specified source to IO port.
	START IO	Transfer instuctions to IOP to initiate an IO operation.
	TEST IO	Transfer status information from IO system to specified destination.
	HALT IO	Transfer instructions to IOP to terminate an IO operation.

Figure 3.34
(continued).

CPU hardware. These instruction sets included only the most frequently used operations such as load a register from memory, store a result in memory, and add two fixed-point numbers. As hardware became cheaper, instructions tended to increase both in number and complexity so that by 1980 a typical computer had dozens of instruction types, with versions to handle several data types and addressing modes. These large instruction sets contain infrequently used but hard-to-program operations like floating-point divide. Since such operations are primitives in programming languages, they serve to reduce the semantic gap between the user's language and the computer's. However, complex instructions lead to a number of complications in both hardware and software design, which we now consider.

Suppose that a particular operation F can be implemented either by a single complex instruction I_F or by a multiinstruction routine P_F composed of simple instructions. Execution of P_F will generally be slower than execution of I_F because the processor must spend more time fetching the instructions of P_F and, depending on the nature of F, handling the intermediate data that links the instructions. A further drawback of P_F is that it occupies more memory space than I_F occupies. An obvious disadvantage of I_F is that it adds to the complexity of a processor's control unit, thereby increasing both the size of the processor and the time required to design it.

Clearly a program involving F is simplified by using I_F in place of P_F. When the program is written in a high-level language, however, as most programs are, the execution speedup that justifies a complex instruction like I_F may not be fully realizable. A compiler will typically translate F into the corresponding machine instruction I_F, if available, which uses fixed CPU registers and has a fixed execution time. On the other hand, if I_F is not available, an efficient or *optimizing* compiler may be able to generate object code Q_F corresponding to P_F that exploits information known at compilation time to reduce F's execution time. Suppose, for instance, that F is fixed-point multiplication and is implemented by both I_F and Q_F via a shift-and-add algorithm of the kind described in Example 2.7. If one of F's

operands is a small constant or zero, then the compiler can easily generate a shorter form of P_F that is faster than the generic n-step multiply instruction I_F. The speed gap between I_F and P_F can also be narrowed by designing the small instruction set required for P_F to reduce the instruction fetch and execute cycle times as far as possible, preferably to one CPU clock cycle each. Another speed advantage of P_F over I_F is that P_F can be interrupted in midoperation at an appropriate instruction boundary, whereas I_F must proceed to termination before the CPU can respond to an interrupt.

Motivated by considerations of the foregoing sort, a number of computer designers advocated machines with relatively small and simple instruction sets, which have been dubbed *RISCs* for reduced instruction-set computers. *RISC* architecture is contrasted with the complex instruction-set computer (*CISC*) architecture found in most pre-1980 designs such as the IBM System/360-370 and the Motorola 680X0. The major attributes of RISCs have been defined as follows [Colwell et al. 1985]:

- Relatively few instruction types and addressing modes.
- Fixed and easily decoded instruction formats.
- Fast, single-cycle instruction execution.
- Hardwired rather than microprogrammed control.
- Memory access limited mainly to load and store instructions.
- Use of compilers to optimize object-code performance.

Several of these RISC attributes are closely related. For example, the small size and regularity of the instruction set simplifies the design of a hardwired program control unit, which in turn facilitates the achievement of fast single-cycle execution. The stress placed on efficient compilation requires the machine architects and compiler writers to cooperate closely in the design process.

RISC architectures restrict the instructions that access memory to load and store. Consequently, most RISC instructions involve only register-to-register operations that are internal to the CPU. To support them, a larger-than-usual number of registers may be placed in the CPU. This design facilitates single-cycle execution and minimizes the CPU cycle time. Pipelining the instruction execution process also supports single-cycle execution. Since complex instructions are not in the instruction set, they must be implemented by multiinstruction routines, which prompts the attention to efficient compilation. Machine code compiled for a RISC computer is likely to have more instructions than the corresponding CISC code but can execute more efficiently, especially if only fixed-point (integer) instructions are involved. However, if the frequency of complex operations is high, then the performance of the CISC machine may be better than that of the RISC machine.

EXAMPLE 3.7 INSTRUCTION SET OF THE MIPS RX000 [KANE AND HEINRICH 1992]. The RX000 microprocessor series and its instruction formats were introduced in Example 3.5 (section 3.3.1). A microprocessor in this family is implemented by a single IC and has the major components indicated in Figure 3.35. These include a file of 32 general-purpose 32-bit registers and the processing logic to perform the basic fixed-point ALU functions: add, subtract, multiply, divide and logical operations using 32-bit operands. Numerical operands are treated as unsigned or signed integers in twos-complement code. One register R0 in the register file permanently stores the constant zero. Some special-purpose arithmetic circuits perform address computation. The

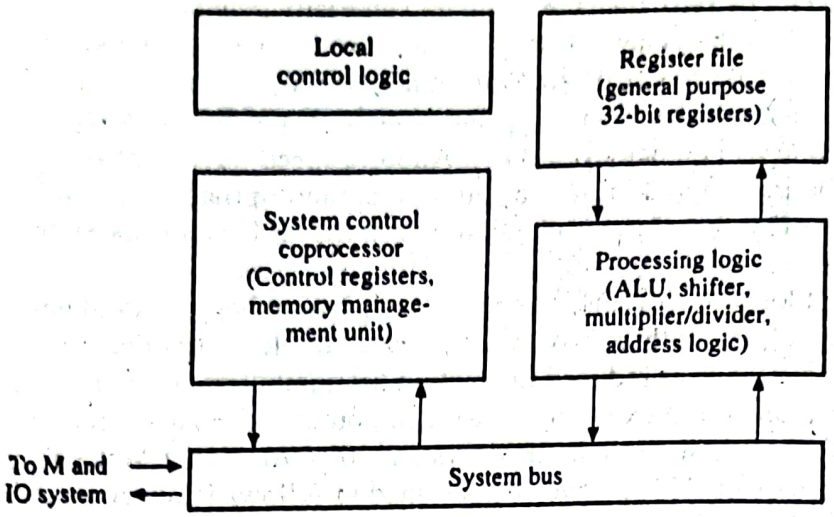


Figure 3.35
Overall organization of the MIPS RX000.

overall organization of the RX000 E-unit is similar to that of the ARM6 (Figure 3.9). As in the ARM6 case, the E-unit of the RX000 is pipelined to support the goal of executing instructions at a peak rate of one instruction per clock cycle. Floating-point operations meeting the requirements of the IEEE 754 standard are supported by an on-chip or off-chip floating-point unit (FPU).

In addition to the control logic needed for instruction execution, the RX000 contains a unit referred to as the *system control coprocessor* whose functions include communication with external memory (caches and main memory) and the automatic address translation logic needed to support a virtual memory system. The virtual memory feature uncouples the address space seen by the programmer from the computer's physical address space, making it possible, for example, to run a large program in a small amount of physical memory. The system control coprocessor is essentially invisible to the applications programmer. The RX000 can have several additional coprocessors implemented on additional ICs.

We now consider in detail the RX000's basic (MIPS I) instruction set, which is summarized in Figure 3.36. There are 74 types, divided almost equally between data-transfer, data-processing, and program-control instructions. All are 32 bits long and use one of the I, J, and R formats illustrated in Figure 3.29. The smallest addressable item in external memory M is, as usual, an 8-bit byte, which requires a 32-bit address to specify its location. Smaller address fields such as the 26-bit branch address field of J-type instructions are automatically extended to 32 bits before loading into the program counter PC. Note that to increment PC to point to the next sequential instruction of a program requires the step PC := PC + 4. The 16-bit (half-word) IMM field of I-type instructions serves either as an immediate data operand or else as an address offset. In either case it is also extended to 32 bits either by zero extension or by sign extension. During initialization, the microprocessor can be reset to store data according to either the big-endian or the little-endian convention.

Following the basic RISC philosophy, communication between the CPU and external memory M is via load and store instructions only, using the I-type format (Figure 3.29). The RX000 has instructions to load and store data in bytes and half-words (2 bytes), as well as full, 4-byte words. If a byte or half-word is to be loaded into a CPU register, then the loaded item is expanded to a full word by sign extension, unless the "unsigned" version of the load instruction is specified, in which case zero extension is

Type	Instruction	Assembly-language format	Narrative format (comment)
Data transfer	Load byte	LB Rt,Source	Load register Rt with sign-extended memory byte.
	Load byte unsigned	LBU Rt,Source	Load register Rt with zero-extended memory half-word.
	Load half-word	LH Rt,Source	Load register Rt with sign-extended memory half-word.
	Load half-word unsigned	LHU Rt,Source	Load register Rt with zero-extended memory half-word.
	Load word	LW Rt,Source	Load register Rt with memory word.
	Load word left	LWL Rt,Source	Load left side of register Rt with 1 to 3 memory bytes.
	Load word right	LWR Rt,Source	Load right side of register Rt with 1 to 3 memory bytes.
	Store byte	SB Rt,Dest	Store least significant byte of register Rt in memory.
	Store half-word	SH Rt,Dest	Store least significant half-word of register Rt in memory.
	Store word	SW Rt,Dest	Store register Rt in memory.
	Store word left	SWL Rt,Dest	Store left 1 to 3 bytes of register Rt in memory.
	Store word right	SWR Rt,Dest	Store right 1 to 3 bytes of register Rt in memory.
	Load upper immediate	LUI Rt,IMM	Move immediate operand IMM.0^{16} into register Rt.
	(Four special register-move instructions for use with multiplication and division)		
	(Eight special data-transfer instructions for use with coprocessors, including the system control coprocessor)		
Data processing	Add	ADD Rd,Rs,Rt	Add Rs to Rt; put result in Rd (trap on overflow).
	Add unsigned	ADDU Rd,Rs,Rt	Add Rs to Rt; put result in Rd.
	Add immediate	ADDI Rt,Rs,IMM	Add sign-extended IMM to Rs; put result in Rt (trap on overflow).
	Add immediate unsigned	ADDIU Rt,Rs,IMM	Add sign-extended IMM to Rs; put result in Rt.
	Subtract	SUB Rd,Rs,Rt	Subtract Rt from Rs; put result in Rd (trap on overflow).
	Subtract unsigned	SUBU Rd,Rs,Rt	Subtract Rt from Rs; put result in Rd.
	AND	AND Rd,Rs,Rt	Bitwise AND Rt and Rs; put result in Rd.
	AND immediate	ANDI Rt,Rs,IMM	Bitwise AND zero-extended IMM and Rs; put results in Rt.
	NOR	NOR Rd,Rs,Rt	Bitwise NOR Rt and Rs; put result in Rd.
	OR	OR Rd,Rs,Rt	Bitwise OR Rt and Rs; put result in Rd.
	OR immediate	ORI Rt,Rs,IMM	Bitwise OR zero-extended IMM and Rs; put result in Rt.
	XOR	XOR Rd,Rs,Rt	Bitwise XOR Rt and Rs; put result in Rd.

Figure 3.36
Instruction set of the MIPS RX000.