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.

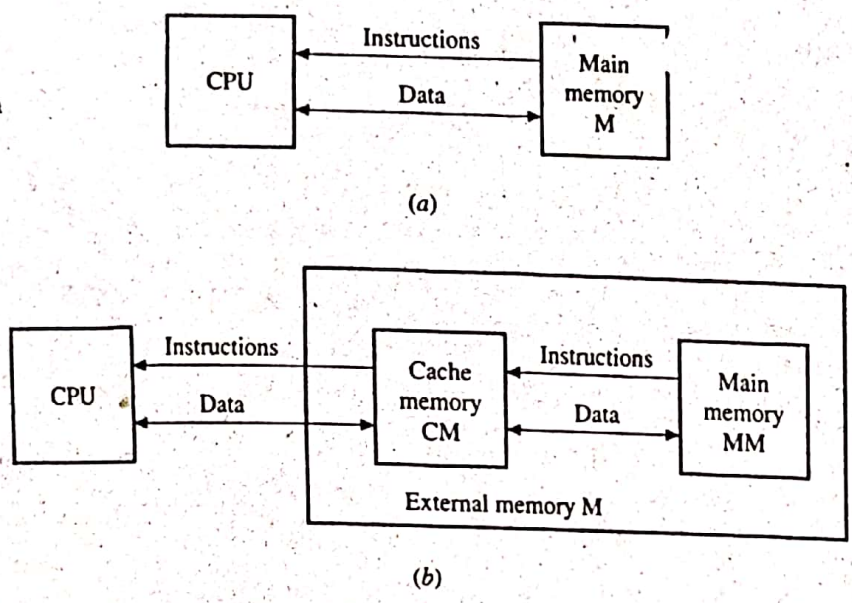


Figure 3.1
Processor-memory communication: (*a*) without a cache and (*b*) with a cache.

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.

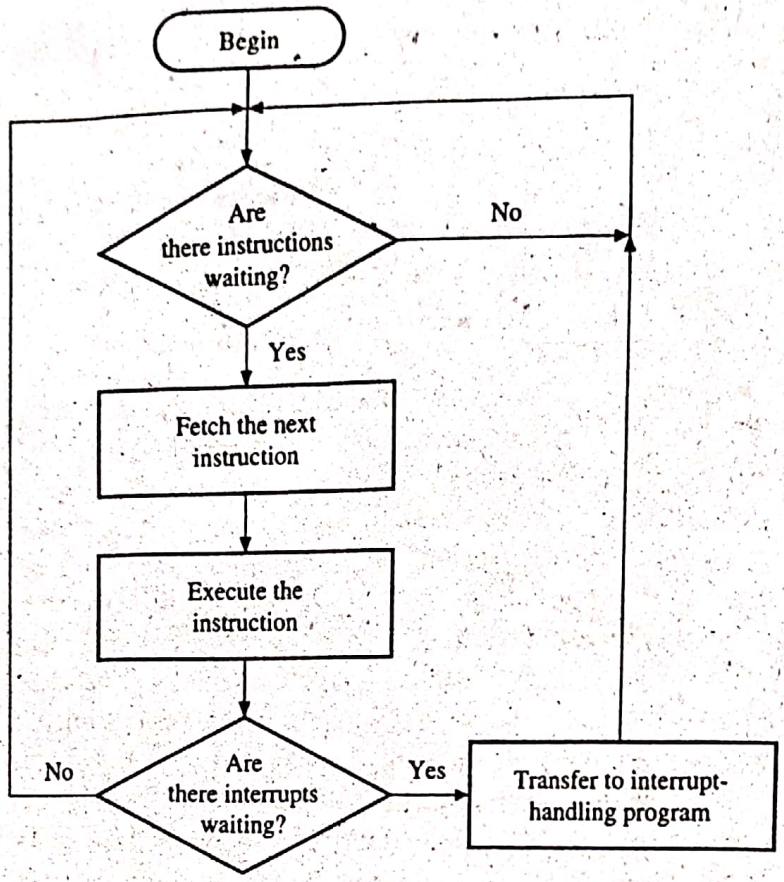


Figure 3.2
Overview of CPU behavior.

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.

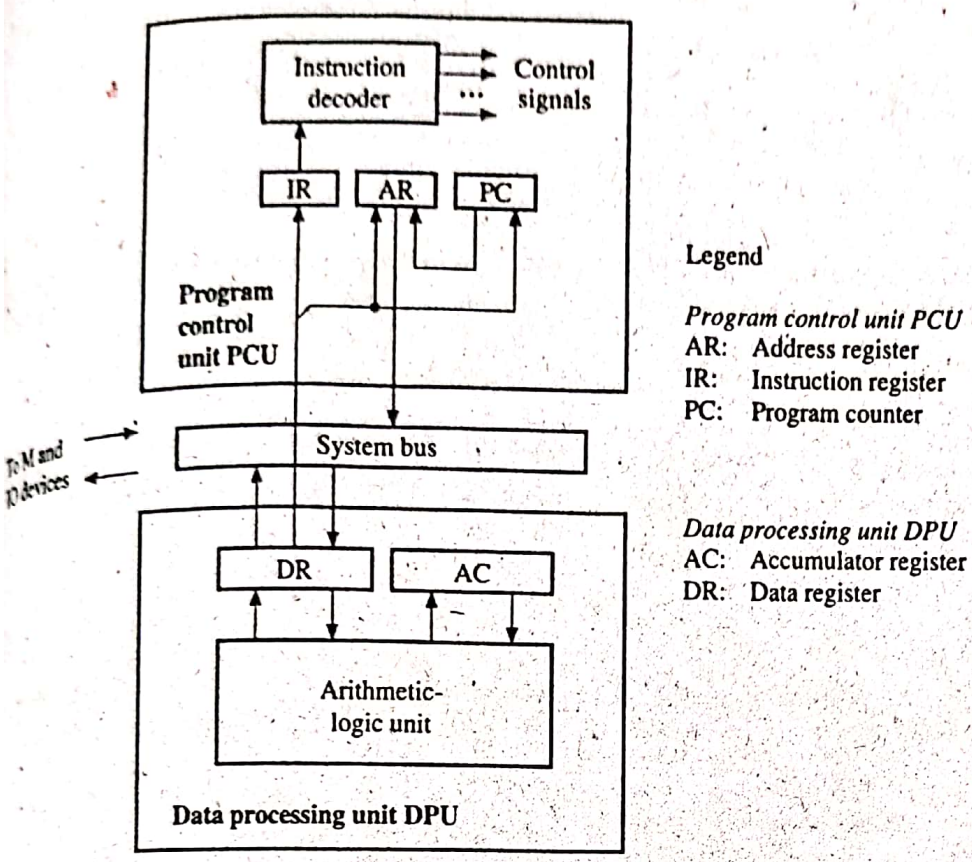


Figure 3.3
A small accumulator-based CPU.

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:

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

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

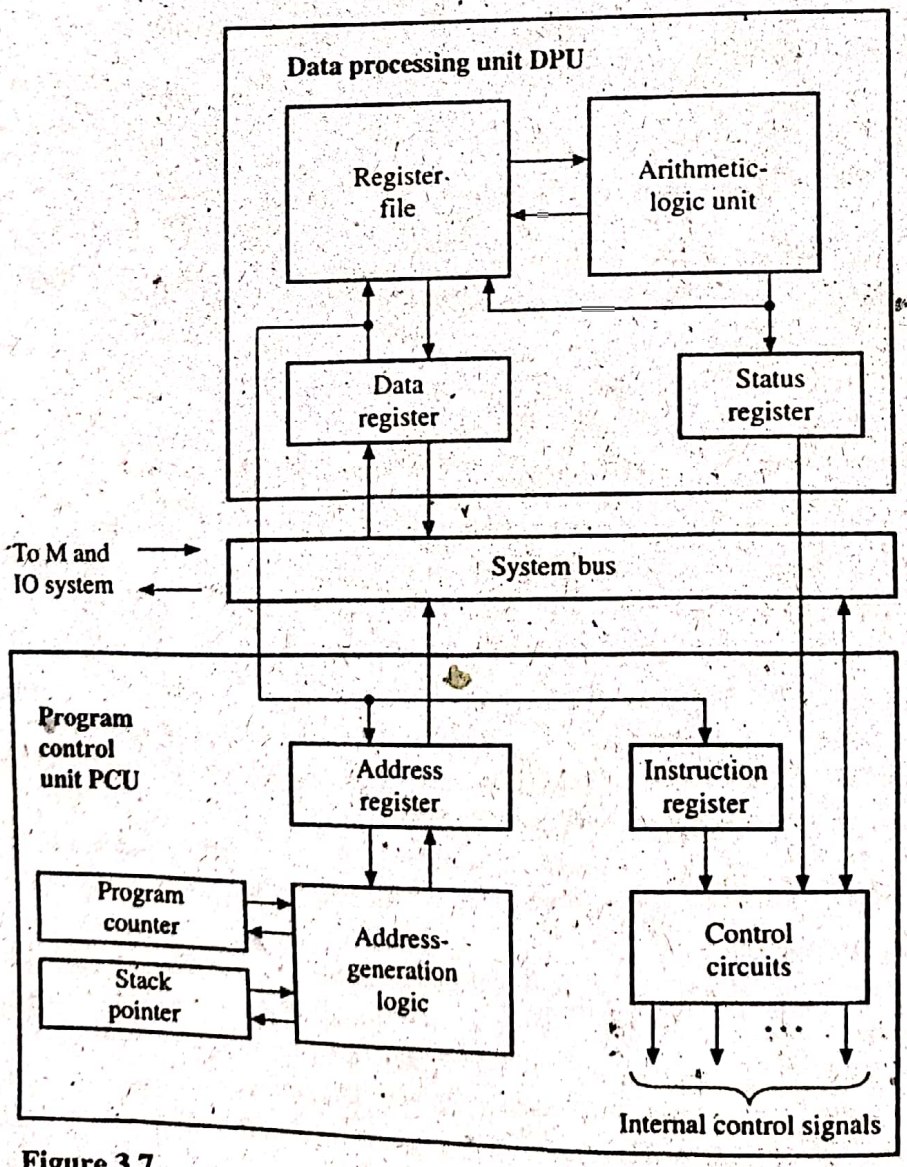


Figure 3.7
A typical CPU with the general register organization.

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

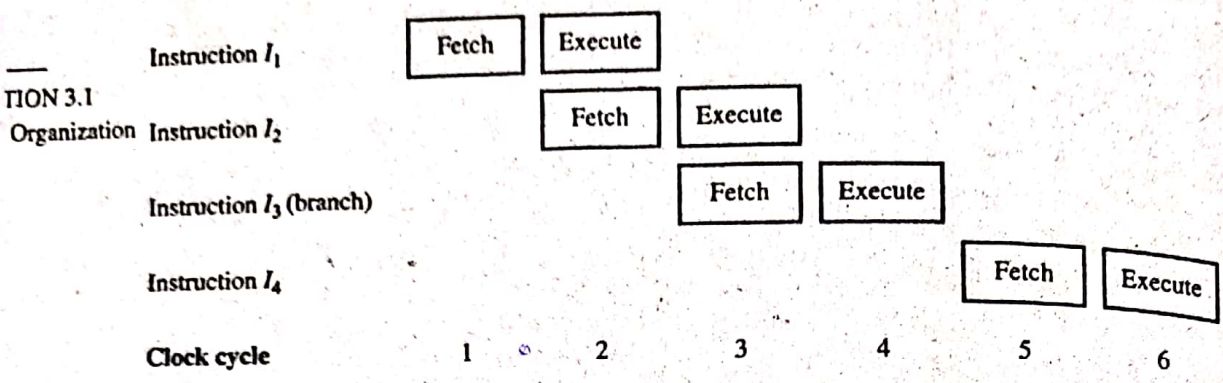


Figure 3.8
Overlapping instructions in a two-stage instruction pipeline.

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
```

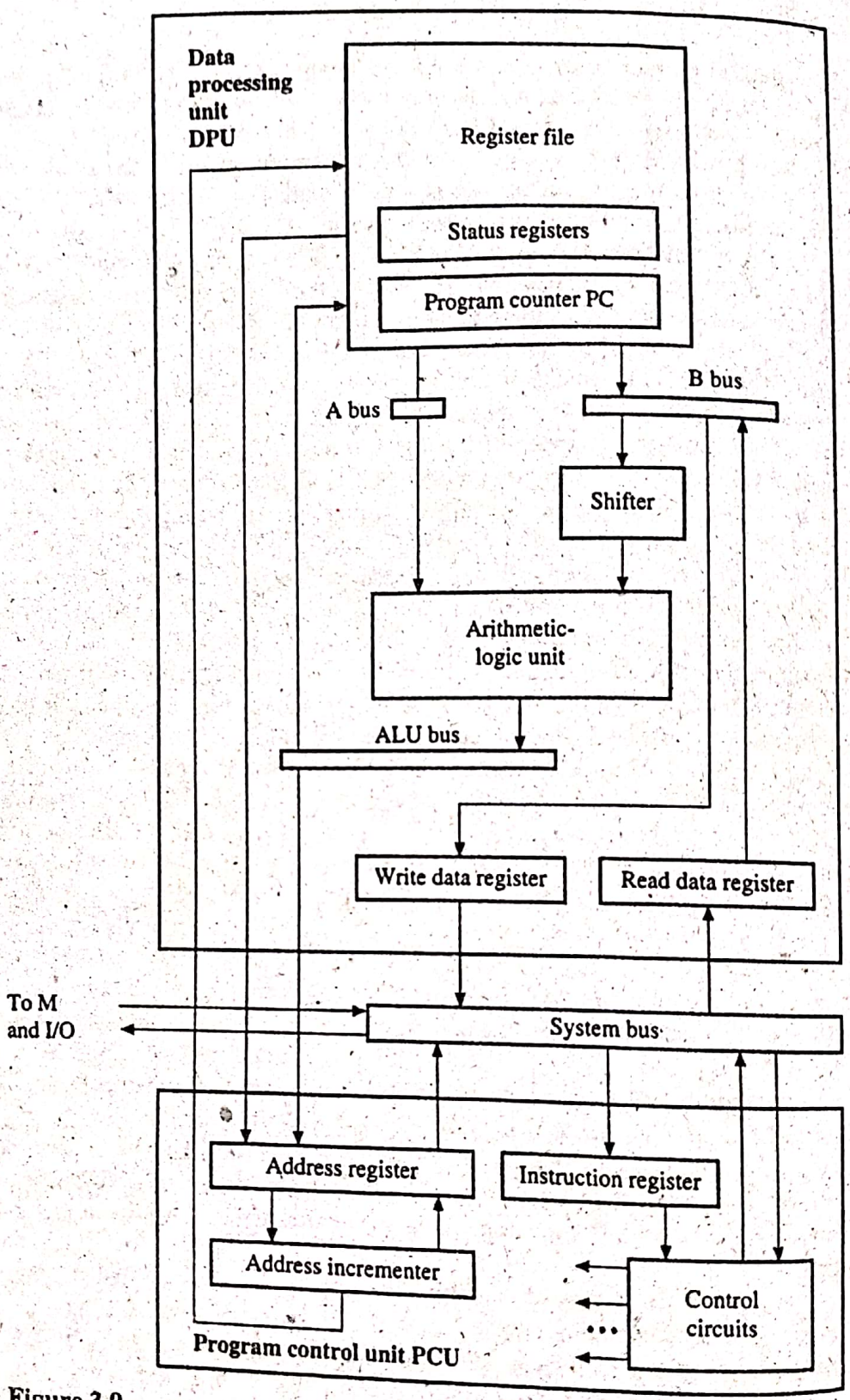


Figure 3.9
Overall organization of the ARM6.

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 R0. 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

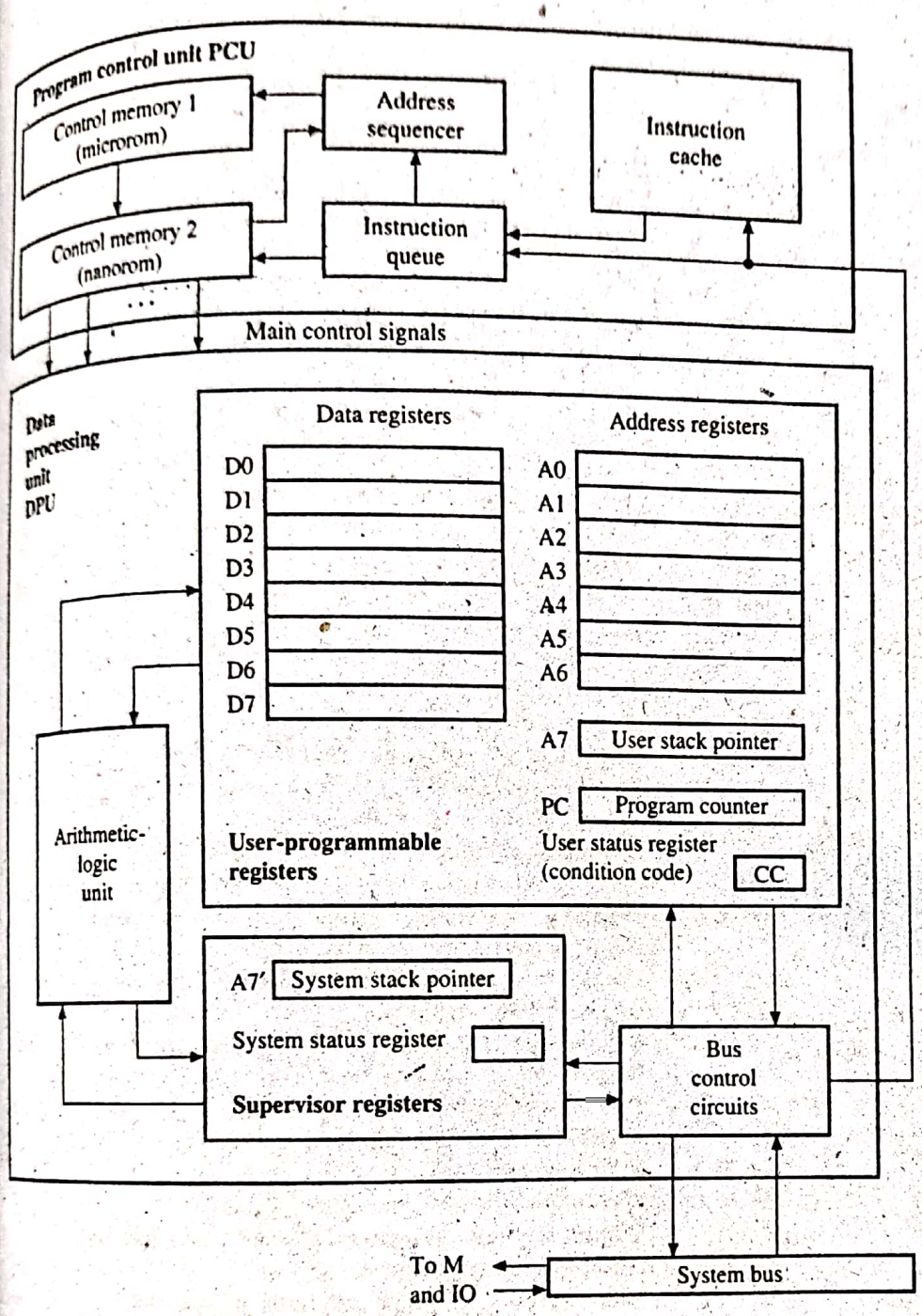


Figure 3.11
Organization of the 68020.

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

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

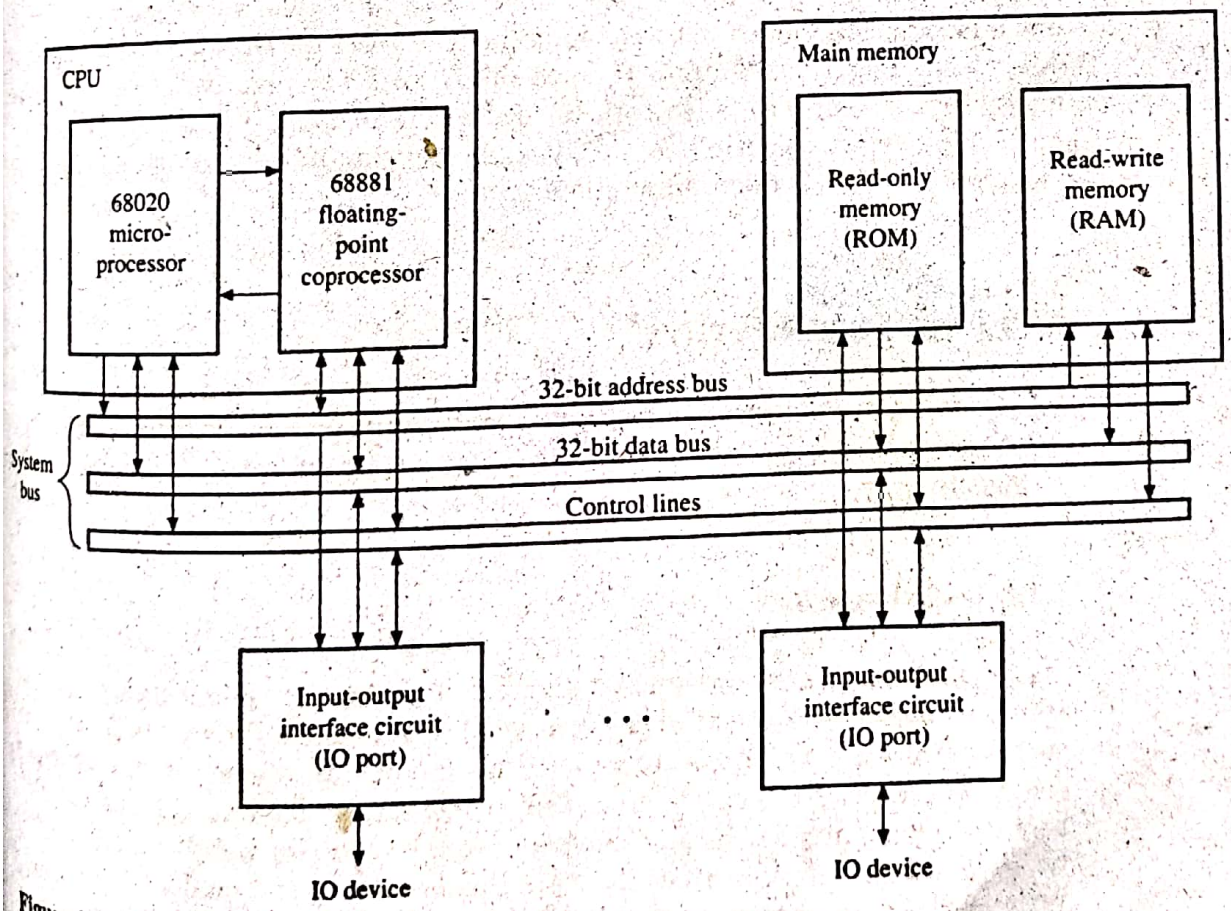


Figure 3.14
68020-based microcomputer with floating-point coprocessor.

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

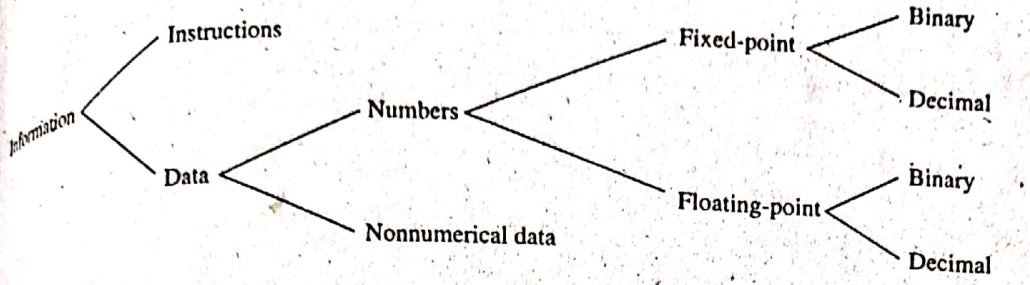


Figure 3.15
The basic information types.

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.

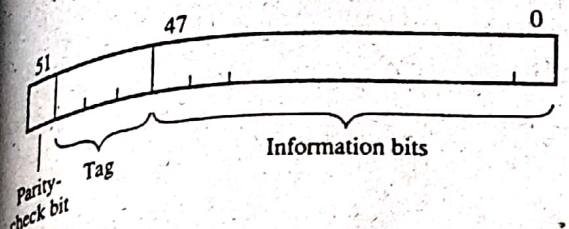
Word length. Information is represented in a digital computer by means of binary words, where a *word* is a unit of information of some fixed length n. An n-bit word allows up to 2^n different items to be represented. For example, with $n = 4$, we can encode the 10 decimal digits as follows:

$$\begin{array}{lllll} 0 = 0000 & 1 = 0001 & 2 = 0010 & 3 = 0011 & 4 = 0100 \\ 5 = 0101 & 6 = 0110 & 7 = 0111 & 8 = 1000 & 9 = 1001 \end{array} \qquad (3.7)$$

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



Tag	Interpretation
000	Single-precision number.
001	Indirect reference word.
010	Double-precision number.
011	Segment descriptor.
100	Step-index control word.
101	Data descriptor.
110	Uninitialized operand.
111	Instruction.

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:

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

TION 3.2 Input
Representation data

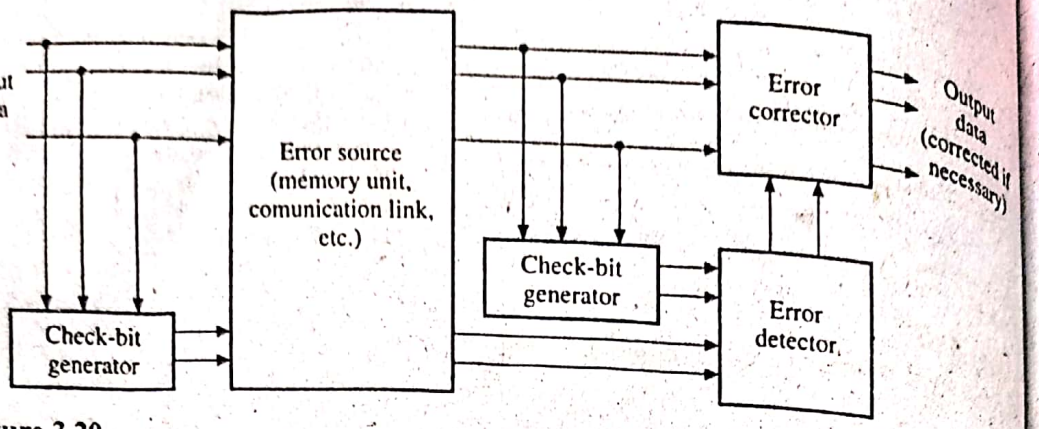


Figure 3.20
Error detection and correction logic.

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

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

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 left-most position, and 0 and 1 are used to denote plus and minus, respectively. This

$$x_{n-1}\underbrace{x_{n-2}x_{n-3}\cdots x_2x_1x_0}_{\text{Magnitude}} \qquad (3.12)$$

Sign

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:

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

Implicit binary point — Implicit binary point

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

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

Implicit binary point — Implicit binary point

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-

Decimal representation	Binary code		
	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

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

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.

	1000 = 5		
+	1100 = 9		
Carry 1 ←	0100		Binary sum
+	0011		Correction
	0111 = 4		Excess-three sum

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 in 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

CHAPTER 4

Datapath Design

An instruction-set processor consists of datapath (data processing) and control units. This chapter addresses the register-level design of the datapath unit, while Chapter 5 covers the control unit. The focus is on the arithmetic algorithms and circuits needed to process numerical data. These circuits are examined first for fixed-point numbers (integers) and then for floating-point numbers. The use of pipelining to speed up data processing is also discussed.

4.1 FIXED-POINT ARITHMETIC

The design of circuits to implement the four basic arithmetic instructions for fixed-point numbers—addition, subtraction, multiplication, and division—is the main topic of this section. It also discusses the implementation of logic instructions and ALU design.

4.1.1 Addition and Subtraction

Add and subtract instructions for fixed-point binary numbers are found in the instruction set of every computer. In smaller machines such as microcontrollers they are the only available arithmetic instructions. As we have seen in earlier chapters, addition and subtraction hardware (Example 2.7) or software (Example 3.1) can be used to implement multiplication and, in fact, any arithmetic operation. Beginning with Charles Babbage, computer designers have devoted considerable effort to the design of high-speed adders and subtracters. As we will see, these basic circuits can be designed in many different ways that involve various trade-offs between operating speed and hardware cost.

Basic adders. First consider the design of a circuit to add two n-bit unsigned binary numbers, a topic discussed in section 2.1.3. The fastest such adder is, in principle, a two-level combinational circuit in which each of the n sum bits is expressed as a (logical) sum of products or product of sums of the n input variables. In practice, such a circuit is feasible for very small values of n only, as it requires $c(n)$ gates with fan-in $f(n)$, where both $c(n)$ and $f(n)$ grow exponentially with n. Practical adders take the form of multilevel combinational or, occasionally, sequential circuits. They sacrifice operating speed for a reduction in circuit complexity as measured by the number and size of the components used. In general, the addition of two n-bit numbers X and Y is performed by subdividing the numbers into stages X_i and Y_i of length n_i, where $n > n_i > 1$. X_i and Y_i are added separately, and the resulting partial sums are combined to form the overall sum. The formation of this sum involves assimilation of carry bits generated by the partial additions.

The sum z_i,c_i of two 1-bit numbers x_i and y_i can be expressed by the *half-adder* logic equations

$$z_i = x_i \oplus y_i$$

$$c_i = x_i y_i$$

where z_i is the sum bit, c_i is the carry-out bit, $\oplus$ denotes EXCLUSIVE-OR, and juxtaposition denotes AND. If we introduce a third input bit c_{i-1} denoting a carry-in signal, we obtain the following *full-adder* equations:

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

$$c_i = x_i y_i + x_i c_{i-1} + y_i c_{i-1} \qquad (4.1)$$

(Note that + denotes logical OR—not plus—here.) A full adder, also called a *1-bit adder,* can be directly implemented from these equations in various ways, as demonstrated by Figure 2.9 (section 2.1.1). Figure 4.1 shows a fast AND-OR realization of a 1-bit adder, along with an appropriate circuit symbol for use in register-level designs.

The least expensive circuit in terms of hardware cost for adding two n-bit binary numbers is a serial adder, the design of which was covered in Example 2.2. A serial adder adds the numbers bit by bit and so requires n clock cycles to compute the complete sum of two n-bit numbers. As Figure 4.2 indicates, a serial adder consists of a full adder realizing Equations (4.1) and a flip-flop to store c_i. One sum bit is generated in each clock cycle; a carry is also computed and stored for use during the next clock cycle. Figure 4.2 presents a high-level view of a serial adder that has a D flip-flop as the carry store. Although this adder is slow, its circuit size is very small and is independent of n.

Circuits that, in one clock cycle, add all bits of two n-bit numbers, as well as an external carry-in signal c_{in}, are called n-bit *parallel* adders or simply n-bit adders. The simplest such adder is formed by connecting n full adders as in Figure 4.3. Each 1-bit adder stage supplies a carry bit to the stage on its left. A 1 appearing on the carry-in line of a 1-bit adder can cause it to generate a 1 on its carry-out line. Hence carry signals propagate through the adder from right to left, giving rise to the name *ripple-carry adder.* In the worst case a carry signal can ripple through all n stages of the adder. The input carry signal c_{in} is normally set to 0 for addition. The maximum signal propagation delay of an n-bit ripple-carry adder, which in synchronous circuit design determines the operating speed, is nd,

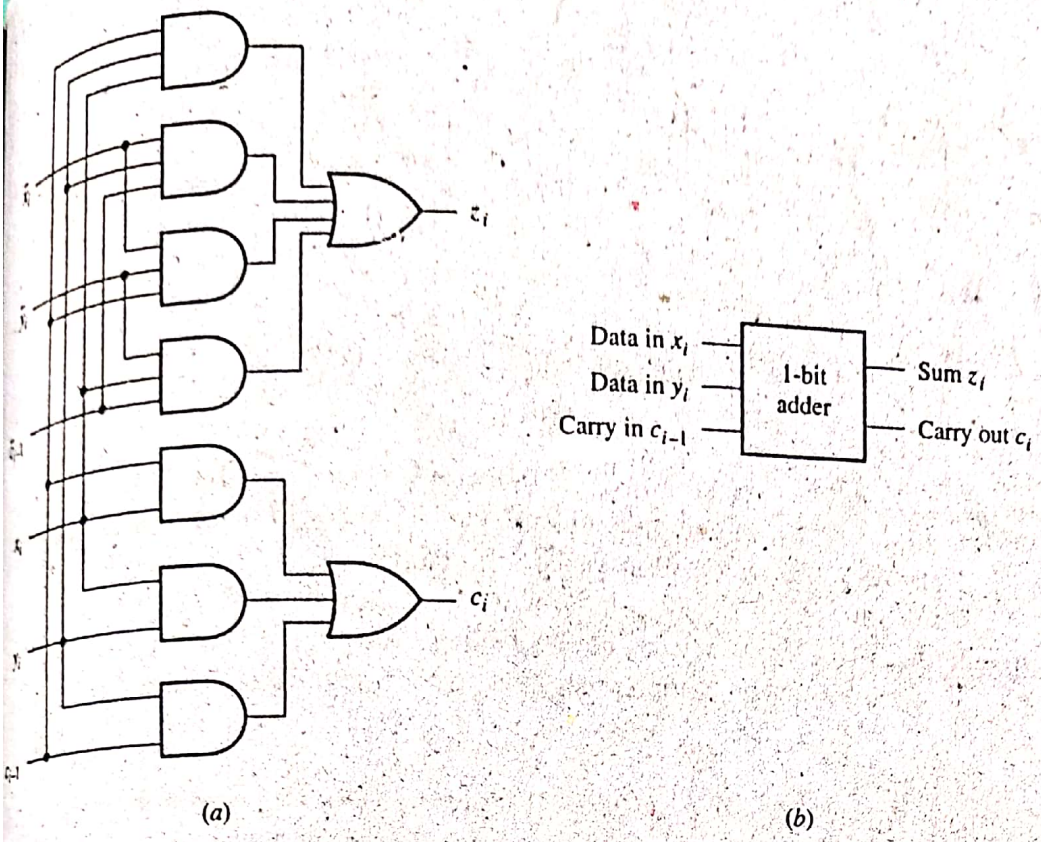


Figure 4.1
A 1-bit (full) adder: (*a*) two-level AND-OR logic circuit and (*b*) symbol.

where d is the delay of a full-adder stage. Unlike a serial adder, the amount of hardware in a ripple-carry adder increases linearly with n, the word size of the numbers being added.

Subtracters. Adders like those of Figures 4.2 and 4.3 operate correctly on both unsigned and positive numbers because the 0 sign bit of a positive number has the same effect as a leading zero in an unsigned number. The best way to add

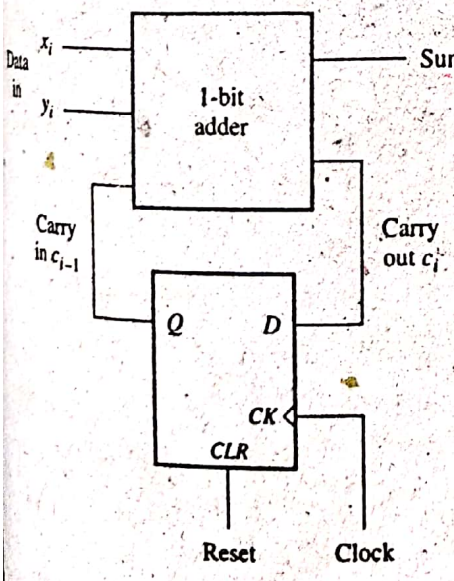


Figure 4.2
A serial binary adder.

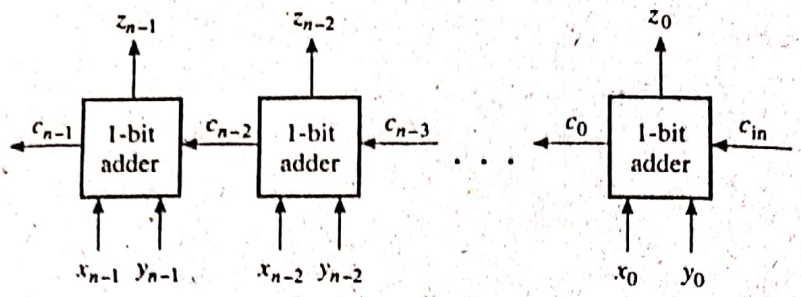


Figure 4.3
An n-bit ripple-carry adder composed of n 1-bit (full) adders.

negative numbers—these have 1 as the sign bit—depends on the number code in use. Adding $-X$ to Y is equivalent to subtracting X from Y, so the ability to add negative numbers implies the ability to do subtraction.

Subtraction is relatively simple with twos-complement code because negation (changing X to $-X$) is very easy to implement. As discussed in section 3.2.2, if $X = x_{n-1}x_{n-2}\ldots x_0$ is a twos-complement integer, then negation is realized by

$$-X = \bar{x}_{n-1}\bar{x}_{n-2}\ldots\bar{x}_0 + 1 \tag{4.2}$$

where + denotes addition modulo 2^n. An efficient way to obtain the ones-complement portion $\bar{X} = \bar{x}_{n-1}\bar{x}_{n-2}\ldots\bar{x}_0$ of $-X$ in (4.2) uses the word-based EXCLUSIVE-OR function $X \oplus s$ with a control variable s. When $s = 0$, $X \oplus s = X$, but when $s = 1$, $X \oplus s = \bar{X}$. Suppose that Y and $X \oplus s$ are now applied to the inputs of an n-bit adder. The addition of 1 required by (4.2) to change $\bar{X}$ to $-X$ can be realized by applying s to the carry input line of the adder. In the resulting circuit shown in Figure 4.4, the control line s selects the addition operation $Y + X$ when $s = 0$ and the subtraction operation $Y - X = Y + \bar{X} + 1$ when $s = 1$. Thus extending a parallel adder to perform twos-complement subtraction as well as addition merely requires connecting n two-input EXCLUSIVE-OR gates to the adder's inputs; these gates are represented by a single n-bit word gate in Figure 4.4.

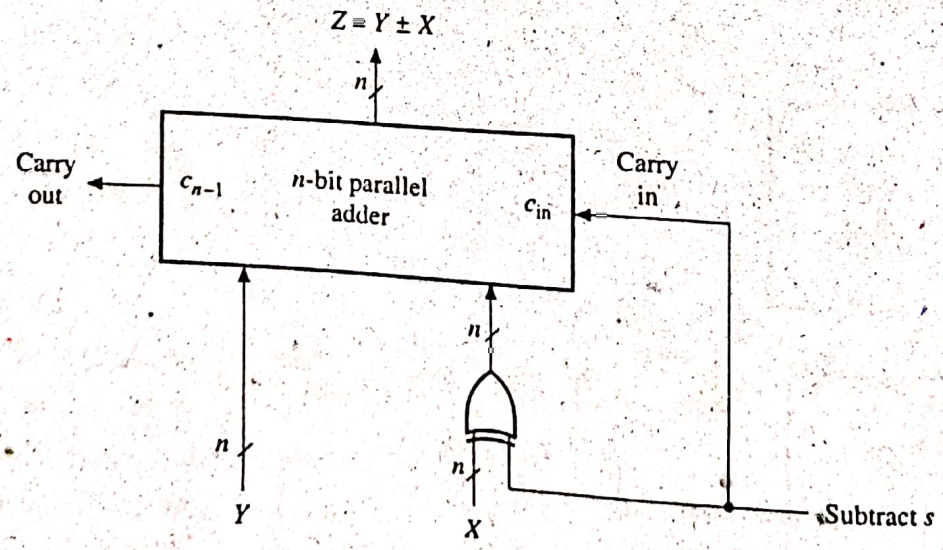


Figure 4.4
An n-bit twos-complement adder-subtracter.

As an example, let $X = 11101011$ and $Y = 00101000$, denoting -21_{10} and 40_{10}, respectively, in twos-complement code. Bit-by-bit addition produces

$$Z = X + Y = 11101011 + 00101000 = 00010011 \tag{4.3}$$

which corresponds to $-21_{10} + 40_{10} = +19_{10}$. (Observe that the output carry $c_{n-1} = 1$ in (4.3) is ignored.) To subtract X from Y, we first compute

$$-X = \bar{1}\ \bar{1}\ \bar{1}\ \bar{0}\ \bar{1}\ \bar{0}\ \bar{1}\ \bar{1} + 1 = 00010101$$

and then the sum

$$Z = (-X) + Y = 00010101 + 00101000 = 00111101$$

which corresponds to $21_{10} + 40_{10} = +61_{10}$.

Subtraction is not so readily implemented in the case of unsigned or sign-magnitude numbers. It is sometimes useful to construct a subtracter for such numbers based on the *full (1-bit) subtracter* function $z_i = y_i - x_i - b_{i-1}$. This operation is defined by the logic equations:

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

$$b_i = x_i\bar{y}_i + x_i b_{i-1} + \bar{y}_i b_{i-1}$$

Here z_i is the difference bit, while b_{i-1} and b_i are the borrow-in and borrow-out bits, respectively. n-bit serial or parallel binary subtracters are constructed in essentially the same way as the corresponding adders with carry signals replaced by borrows. Subtracters are of minor interest compared with adders, because, as we have just seen, an adder suffices for both addition and subtraction when twos-complement number code is used.

Overflow. When the result of an arithmetic operation exceeds the standard word size n, overflow occurs. With n-bit *un*signed numbers, overflow is indicated by an output carry bit $c_{n-1} = 1$. For example, adding the unsigned numbers $X = 11101011 = 235_{10}$ and $Y = 00101010 = 42_{10}$ using an adder like that of Figure 4.3 yields

$$Z = X + Y = 11101011 + 00101010 = 00010101 \tag{4.4}$$

with $c_{n-1} = c_7 = 1$. Now Z corresponds to 21_{10}, which is $235_{10} + 42_{10}$ (modulo 256) and is the result of addition that "wraps around" when the largest number $2^n - 1$, in this case $11111111 = 255_{10}$, is exceeded. On appending c_7 to Z, we get $c_7Z = 100010101 = 277_{10} = 256_{10} + 21_{10}$, which is the sum in ordinary (modulo infinity) arithmetic. Unsigned arithmetic operations are often viewed as modulo-2^n operations only, and overflow is not explicitly detected. This is the case when computing memory addresses in a computer, for instance, where addresses simply wrap around to zero after the highest address is reached.

Overflow is indicated by a flag bit v in operations involving signed numbers; this flag is found in CPU status (condition code) registers. If we reinterpret the numbers in the preceding example as twos-complement rather than as unsigned, then $X = 11101011$ denotes -21_{10}, while $Y = 00101010$ denotes $+42_{10}$. The result Z computed in (4.4) now denotes $+21_{10}$, and the fact that $c_{n-1} = 1$ does not indicate overflow. In fact, we can never have overflow on adding a positive to a negative number. Overflow in modulo-2^n twos-complement addition can only result from adding two positive numbers or two negative numbers. In the first case overflow

is indicated by a carry bit *into* the sign position, that is, by $c_{n-2} = 1$, since this indicates that the magnitude of the sum exceeds the $n - 1$ bits allocated to it. A little thought shows that overflow from adding two negative numbers is indicated by $c_{n-2} = 0$. We can thus conclude (as we did earlier in section 3.2.2) that the overflow condition is specified by the logic expression

$$v = \bar{x}_{n-1}\bar{y}_{n-1}c_{n-2} + x_{n-1}y_{n-1}\bar{c}_{n-2} \tag{4.5}$$

Now c_{n-1}, the carry output signal from the sign position, is defined by $x_{n-1}y_{n-1} + x_{n-1}c_{n-2} + y_{n-1}c_{n-2}$, from which it follows that

$$v = c_{n-1} \oplus c_{n-2} \tag{4.6}$$

Either (4.5) or (4.6) can be used to design overflow detection logic for two's-complement addition or subtraction. Overflow detection in the case of sign-magnitude numbers is similar and is left as an exercise (problem 4.6).

High-speed adders. The general strategy for designing fast adders is to reduce the time required to form carry signals. One approach is to compute the input carry needed by stage i directly from carrylike signals obtained from all the preceding stages $i - 1, i - 2, \ldots, 0$, rather than waiting for normal carries to ripple slowly from stage to stage. Adders that use this principle are called *carry-lookahead adders*. An n-bit carry-lookahead adder is formed from n stages, each of which is basically a full adder modified by replacing its carry output line c_i by two auxiliary signals called g_i and p_i, or *generate* and *propagate*, respectively, which are defined by the following logic equations:

$$g_i = x_i y_i \qquad p_i = x_i + y_i \tag{4.7}$$

The name *generate* comes from the fact that stage i generates a carry of 1 ($c_i = 1$) independent of the value of c_{i-1} if both x_i and y_i are 1; that is, if $x_i y_i = 1$. Stage i propagates c_{i-1}; that is, it makes $c_i = 1$ in response to $c_{i-1} = 1$ if x_i or y_i is 1—in other words, if $x_i + y_i = 1$.

Now the usual equation $c_i = x_i y_i + x_i c_{i-1} + y_i c_{i-1}$, denoting the carry signal c_i to be sent to stage $i + 1$, can be rewritten in terms of g_i and p_i.

$$c_i = g_i + p_i c_{i-1} \tag{4.8}$$

Similarly, c_{i-1} can be expressed in terms of g_{i-1}, p_{i-1}, and c_{i-2}.

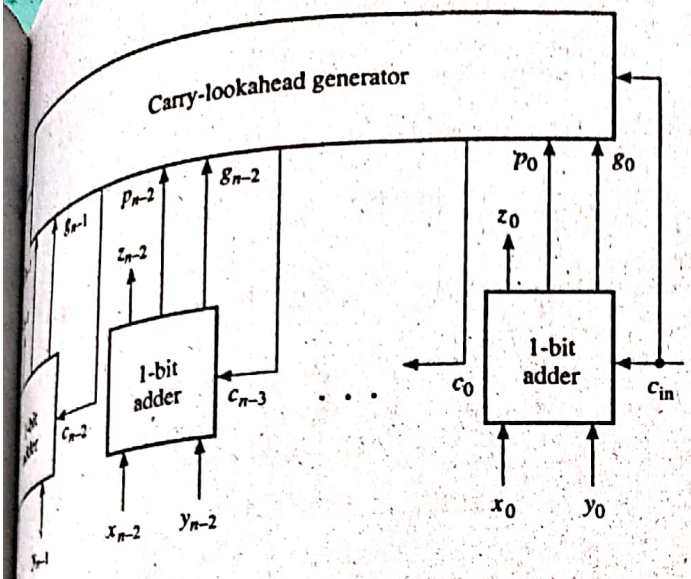
$$c_{i-1} = g_{i-1} + p_{i-1}c_{i-2} \tag{4.9}$$

On substituting (4.9) into (4.8) we obtain

$$c_i = g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-2}$$

Continuing in this way, c_i can be expressed as a sum-of-products function of the p and g outputs of all the preceding stages. For example, the carries in a four-stage carry-lookahead adder are defined as follows:

$$\begin{aligned}
c_0 &= g_0 + p_0 c_{in} \\
c_1 &= g_1 + p_1 g_0 + p_1 p_0 c_{in} \\
c_2 &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_{in} \\
c_3 &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_{in}
\end{aligned} \tag{4.10}$$



4.5
structure of carry-lookahead adder.

4.5 shows the general form of a carry-lookahead adder circuit designed in

gy.
e can further simplify the design by noting that the sum equation for stage i

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

ivalent to

$$z_i = p_i \oplus g_i \oplus c_{i-1} \tag{4.11}$$

ining the pg equations (4.7), the carry-lookahead equations (4.10), and the ified sum equations (4.11) for $0 \le i \le 3$, we obtain the 4-bit carry-lookahead depicted in Figure 4.6. This design is found in practical adders such as the IC [Texas Instruments 1988]. It has four levels of logic gates, so the adder's mum delay is $4d$, where d is the (average) gate delay. This delay is indepen- f the number of inputs n as long as carry generation is defined by two-level s in (4.10). However, the number of gates grows in proportion to n^2 as n ases. In contrast, the number of gates in a two-level adder of the sum-of- cts type grows exponentially with n, while the number of gates in a ripple- adder grows linearly with n. The complexity of the carry-generation logic in rry-lookahead adder, including its gate count, its maximum fan-in, and its mum fan-out, increases steadily with n. Such practical cost considerations in a single carry-lookahead adder module to four or so.

dder expansion. The methods of handling carry signals in the two main com- nal adder designs considered so far, namely, ripple-carry propagation (Fig- 3) and carry-lookahead (Figure 4.5), can be extended to larger adders of the eeded to execute add instructions in, say, a 64-bit computer. If we replace the (full) adder stages in the n-bit ripple-carry design of Figure 4.3 with n k-bit , we obtain an nk-bit adder. Four 4-bit adders such as the 4-bit carry- head circuit of Figure 4.6 can be connected in this way to form the 16-bit appearing in Figure 4.7. This design represents a compromise between a 16- ripple-carry adder, which is cheap but slow, and a single-stage 16-bit

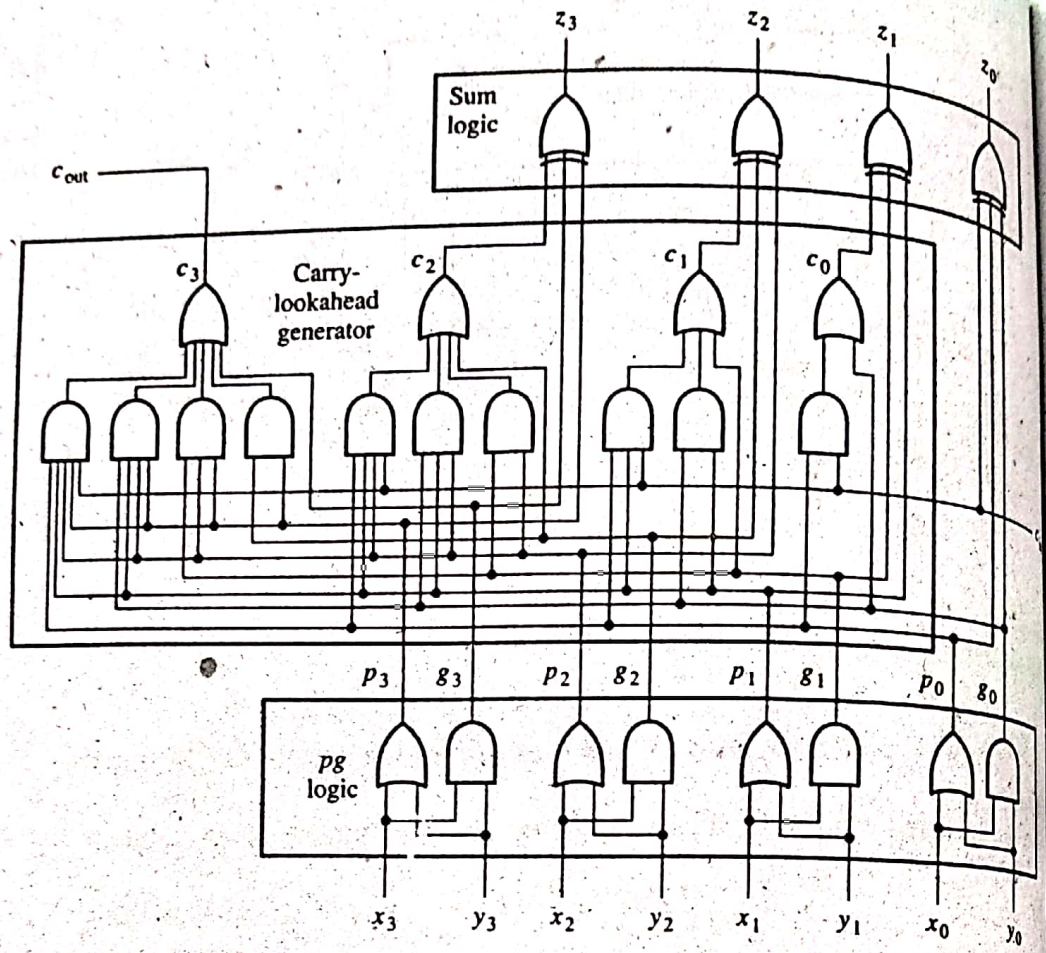


Figure 4.6
A 4-bit carry-lookahead adder.

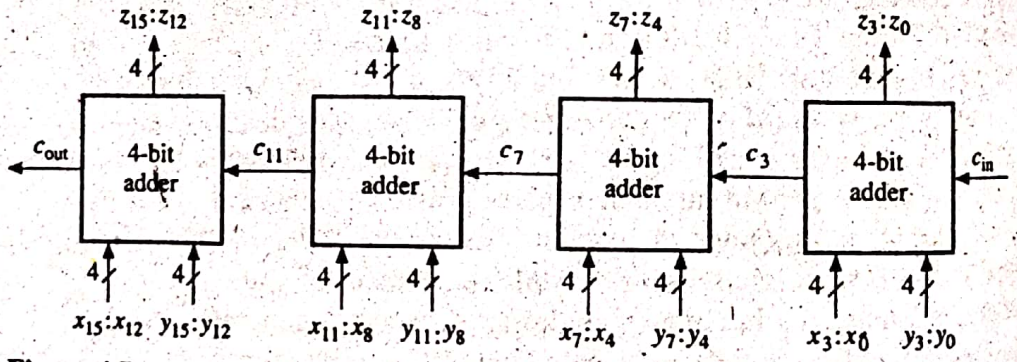


Figure 4.7
A 16-bit adder composed of 4-bit adders linked by ripple-carry propagation.

carry-lookahead adder, which is fast, expensive, and impractical because of the complexity of its carry-generation logic. The circuit of Figure 4.7 effectively combines sets of four $x_i y_i$ inputs into groups that are added via carry lookahead; the results computed by the various groups are then linked via ripple carries.

Comparing Figures 4.3 and 4.7, we see that we have effectively replaced components designed for 1-bit addition with similar but larger components intended for 4-bit addition. If we apply the same principle to the carry-lookahead circuit of Figure 4.5, we get the expanded design of Figure 4.8. Again we are replacing 1-bit

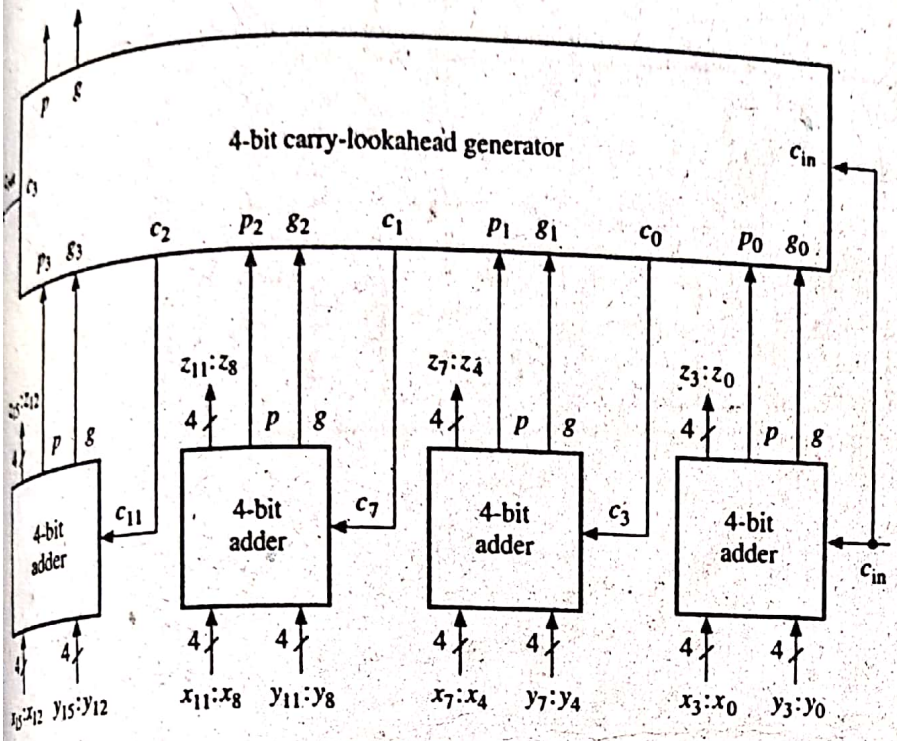


Figure 4.8
A 16-bit adder composed of 4-bit adders linked by carry lookahead.

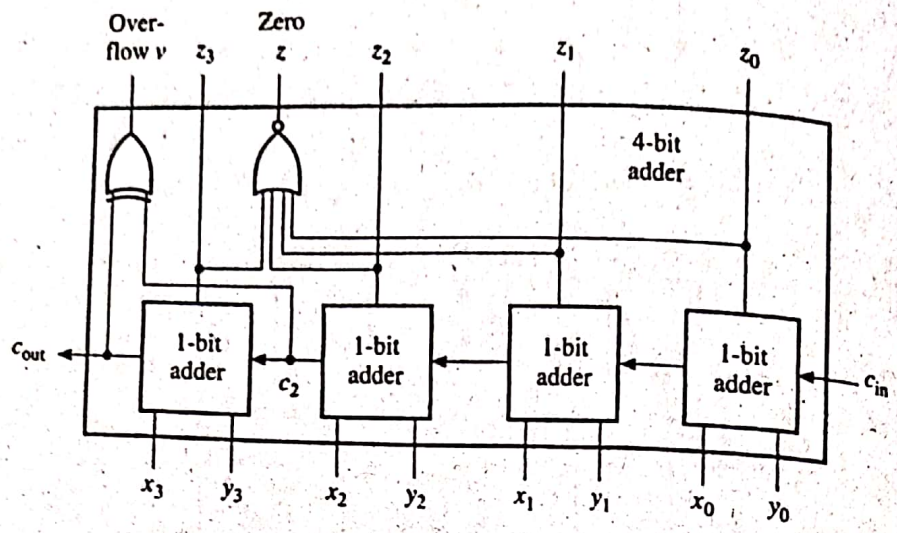
adders with 4-bit adders, but now each adder stage produces a propagate-generate signal pair pg instead of c_{out}, and a carry-lookahead generator converts the four sets of pg signals to the carry inputs required by the four stages. The "group" g and p signals produced by each 4-bit stage are defined by

$$\begin{aligned} g &= x_i y_i + x_{i-1} y_{i-1}(x_i + y_i) + x_{i-2} y_{i-2}(x_i + y_i)(x_{i-1} + y_{i-1}) \\ &\quad + x_{i-3} y_{i-3}(x_i + y_i)(x_{i-1} + y_{i-1})(x_{i-2} + y_{i-2}) \\ p &= (x_i + y_i)(x_{i-1} + y_{i-1})(x_{i-2} + y_{i-2})(x_{i-3} + y_{i-3}) \end{aligned} \tag{4.12}$$

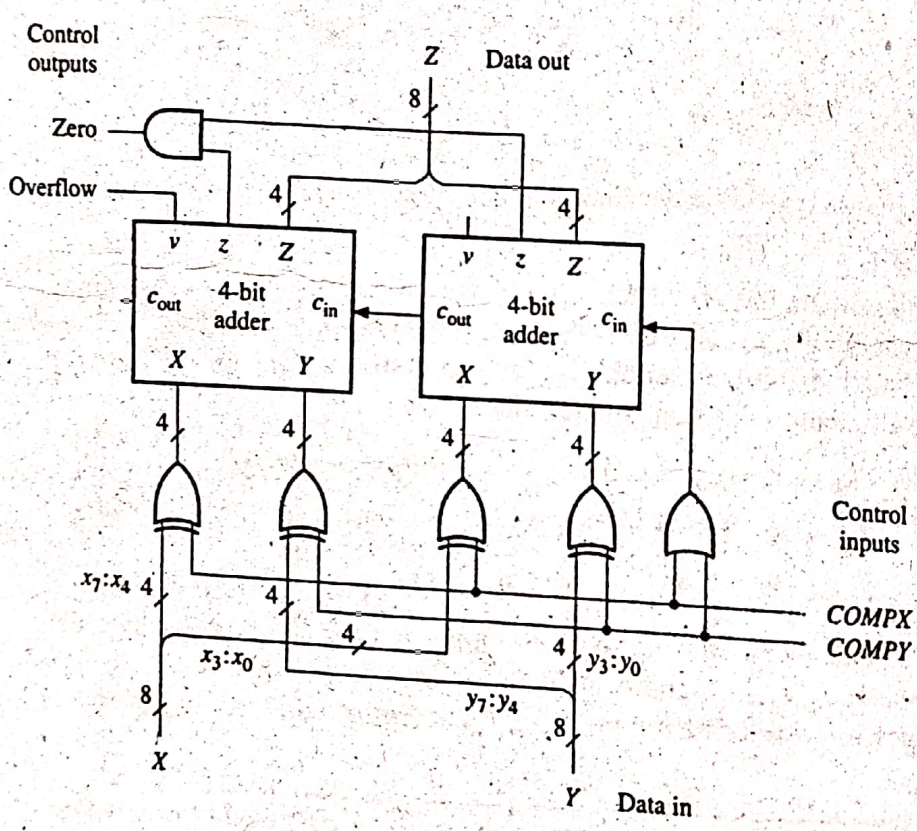
which directly extend (4.7). It is not hard to show that the logic to generate the group carry signals c_{out}, c_{11}, c_7, and c_3 in Figure 4.8 is exactly the same as that of the carry-lookahead generator of Figure 4.6 and is therefore defined by Equations (4.10).

EXAMPLE 4.1 DESIGN OF A COMPLETE TWOS-COMPLEMENT ADDER-SUBTRACTER. To illustrate the preceding concepts, we will design a twos-complement adder-subtracter that computes the three quantities $X + Y$, $X - Y$, and $Y - X$, as well as overflow and zero flags. The design goal is to minimize the number of gates used; operating speed is not of concern. The circuit is required in several versions that handle different data word sizes, including 4, 8, and 16 bits. We will assume that we have standard gate-level and 4-bit register-level components available as building blocks.

The lowest cost adders employ ripple-carry propagation and can easily provide access to the internal signals needed by the flags. Recall that overflow detection uses c_{n-2}, the input carry to the sign position. Zero detection requires access to all the sum outputs and poses no special problems. Figure 4.9*a* shows the logic diagram of an appropriate 4-bit ripple-carry adder. The overflow flag is defined by Equation (4.6) as $v = c_3 \oplus c_2$ and is realized here by an XOR gate. The zero flag is defined by $z =$



(a)



(b)

Figure 4.9
Low-cost addition and subtraction of twos-complement numbers: (*a*) 4-bit adder module and (*b*) 8-bit adder-subtracter.

$\overline{z_3 + z_2 + z_1 + z_0}$ and implemented by a NOR gate. We can use k copies of this adder to produce a $4k$-bit ripple-carry adder in the usual way. The overflow flag for the entire circuit is taken from the v output of the left-most (most significant) stage, while the z outputs of all the stages are ANDed to produce the zero flag.

To extend the adder to an adder-subtracter, the design of Figure 4.4 is a good starting point. It uses an XOR word gate to complement the X input, thereby enabling the circuit to compute $X + Y$ and $Y - X$. To implement the third operation $X - Y$, we could

insert a two-way 4-bit multiplexer into each of the data-in buses so that both X and Y can be applied to each of the adder-subtracter's data inputs. A cheaper solution is to insert a second XOR word gate into the Y bus, enabling Y to be complemented independently. We can then compute $X - Y$ in the form $X + \bar{Y} + 1$.

The complete design of an 8-bit adder-subtracter along the foregoing lines is depicted in Figure 4.9*b*. It contains two 4-bit adders of the type in Figure 4.9*a* linked by their carry lines. Two lines *COMPX* and *COMPY* control the XOR gates that change X and Y to $\bar{X}$ and $\bar{Y}$, respectively. The OR gate sets the adder's carry-in line to 1 during subtraction. A two-input AND gate combines the two z outputs to produce the zero flag, which is 1 if and only if the entire 8-bit result $Z = 0$.

Three of the four signal combinations on *COMPX* and *COMPY* control lines implement the desired three arithmetic functions. The fourth combination 11 implements the sum $\bar{X} + \bar{Y} + 1$, which is an arithmetic function implemented by our design that has no obvious uses.[1] Such superfluous functions are common in the design of data processing circuits.

4.1.2 Multiplication

Fixed-point multiplication requires substantially more hardware than fixed-point addition and, as a result, is not included in the instruction sets of some smaller processors. Multiplication is usually implemented by some form of repeated addition. A simple but slow method to compute $X \times Y$ is to add the multiplicand Y to itself X times, where X is the multiplier. (A version of this technique using counters is discussed in problem 2.4.) Often multiplication is implemented by multiplying Y by X k bits at a time and adding the resulting terms. Figure 4.10 shows this process for unsigned binary numbers in pencil-and-paper calculations with $k = 1$. The main operations involved are shifting and addition. The algorithm of Figure 4.10 is inefficient in that the 1-bit products $x_j 2^i Y$ must be stored until the final addition step is completed. In machine implementations it is desirable to add each $x_j 2^i Y$ term as it is generated to the sum of the preceding terms to form a number P_{i+1} called a *partial product*. Figure 4.11 shows the calculation in Figure 4.10 implemented in this way. The computation involved in processing one multiplier bit x_j can be described by a register-transfer statement of the form

$$P_{i+1} := P_i + x_j 2^i Y \tag{4.13}$$

1010	Multiplicand Y
1101	Multiplier $X = x_3x_2x_1x_0$
1010	x_0Y
0000	$x_1 2Y$
1010	$x_2 2^2 Y$
1010	$x_3 2^3 Y$
10000010	Product $P = \sum_{j=0}^{3} x_j 2^j Y$

Figure 4.10
Typical pencil-and-paper method for multiplication of unsigned binary numbers.

[1] On the other hand, it has been observed, that "there is no feature of a machine, however pathological, which cannot be exploited by a programmer." [Kampe 1960].

1010	Multiplicand Y
1101	Multiplier $X = x_3x_2x_1x_0$
00000000	$P_0 = 0$
1010	x_0Y
00001010	$P_1 = P_0 + x_0Y$
0000	$x_1 2Y$
00001010	$P_2 = P_1 + x_1 2Y$
1010	$x_2 2^2 Y$
00110010	$P_3 = P_2 + x_2 2^2 Y$
1010	$x_3 2^3 Y$
10000010	$P_4 = P_3 + x_3 2^3 Y = P$

Figure 4.11
The multiplication of Figure 4.10 modified for machine implementation.

where 2^iY is equivalent to Y shifted i positions to the left. In the version of this multiplication algorithm presented in Example 2.7 (section 2.2.3), P_i is shifted with respect to a fixed multiplicand Y so that (4.13) is replaced by the equivalent two operations

$$P_i := P_i + x_jY; \quad P_{i+1} := 2^{-j}P_i; \tag{4.}$$

The multiplication of sign-magnitude numbers requires a straightforward extension of the unsigned case discussed above. The magnitude part of the product $P = X \times Y$ is computed by the unsigned shift-and-add multiplication algorithm, and the sign p_s of P is computed separately from the signs of X and Y as follows: $p_s = x_s \oplus y_s$. The implementation of sign-magnitude multiplication using this sequence method is covered in Example 2.7.

Twos-complement multipliers. The multiplication of twos-complement numbers presents some difficulties in the case of negative operands. For example, when a negative P_i is right-shifted as in (4.14), leading 1s rather than leading 0s must be introduced at the left end of the number. More seriously, the multiplication process must treat positive and negative operands differently.

A conceptually simple approach to twos-complement multiplication is to negate all negative operands at the beginning, perform unsigned multiplication on the resulting (positive) numbers, and then negate the result if necessary. Twos-complement negation for an integer $X = x_{n-1}x_{n-2}x_{n-3}\ldots x_1x_0$ is specified by

$$-X = \bar{x}_{n-1}\bar{x}_{n-2}\bar{x}_{n-3}\ldots\bar{x}_1\bar{x}_0 + 000\ldots01 \quad \text{(modulo } 2^n\text{)} \tag{4.}$$

and can easily be implemented by an adder and an EXCLUSIVE-OR word gate as shown in Figure 4.4. However, up to four extra clock cycles are needed to negate X and Y and the double-length product P. Several faster schemes have been proposed to handle negative operands. Since these hinge on certain properties of the two's-complement representation, we consider the latter first.

Clearly $\bar{x}_i = 1 - x_i$ (modulo 2), so we can rewrite (4.15) as follows:

$$-X = 111\ldots11 - x_{n-1}x_{n-2}x_{n-3}\ldots x_1x_0 + 000\ldots01 \quad \text{(modulo } 2^n\text{)} \tag{4.}$$

Since $2^n = 111\ldots11 + 000\ldots01$, this equation is equivalent to $-X = 2^n - X$, which, incidentally, indicates the origin of the term twos-complement. Now if X is positive

$(x_{n-1} = 0)$, we can express its value as

$$X = \sum_{i=0}^{n-2} 2^i x_i \tag{4.17}$$

If X is negative ($x_{n-1} = 1$), then (4.17) does not hold. However, we can rewrite (4.16) as

$$\begin{aligned} -X &= 111\ldots11 - (0x_{n-2}x_{n-3}\ldots x_1x_0 + 100\ldots00) + 000\ldots01 \\ &= 2^{n-1} - x_{n-2}x_{n-3}\ldots x_1x_0 \end{aligned} \tag{4.18}$$

because $2^{n-1} = 111\ldots11 - 100\ldots00 + 000\ldots0$. Hence for negative X,

$$\begin{aligned} X &= -2^{n-1} + x_{n-2}x_{n-3}\ldots x_1x_0 \\ &= -2^{n-1} + \sum_{i=0}^{n-2} 2^i x_i \end{aligned} \tag{4.19}$$

Finally, we combine (4.17) and (4.19) into a single formula

$$X = -2^{n-1}x_{n-1} + \sum_{i=0}^{n-2} 2^i x_i \tag{4.20}$$

which is valid for both positive and negative n-bit integers. For example, suppose that $n = 6$ and $X = 101101$. Evaluating X according to (4.20) yields

$$\begin{aligned} X &= -2^5 \times 1 + 2^4 \times 0 + 2^3 \times 1 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 \\ &= -32 + 8 + 4 + 1 = -19 \end{aligned}$$

Equation (4.20) implies that we can treat bits $x_{n-2}x_{n-3}\ldots x_1x_0$ of a negative twos-complement integer in the same way as the corresponding (magnitude) bits of a positive number; each bit x_i has the positive weight 2^i. Weight $+2^{n-1}$ is assigned to the sign bit x_{n-1} of a positive number; however, since $x_{n-1} = 0$, its contribution to the number is zero. In the negative case, the sign x_{n-1} is assigned weight -2^{n-1}; this adds -2^{n-1} to the number, ensuring that it is negative.

If $X = x_{n-1}x_{n-2}\ldots x_1x_0$ is a twos-complement fraction instead of an integer, then the negation formula (4.15) remains valid, but because bit i now has weight 2^{i-n+1} instead of 2^i, Equation (4.20) is replaced by

$$X = -2^0 x_{n-1} + \sum_{i=0}^{n-2} 2^{i-n+1} x_i \tag{4.21}$$

In effect we have multiplied (4.20) by the scaling factor $2^{-(n-1)}$. For example, let $n = 4$ and $X = 1011$, which represents the fraction -0.625_{10}. Application of (4.21) yields

$$\begin{aligned} X &= -2^0 \times 1 + 2^{-1} \times 0 + 2^{-2} \times 1 + 2^{-3} \times 1 \\ &= -1.000 + 0.250 + 0.125 = -0.625 \end{aligned}$$

Suppose that X is the multiplier operand in a shift-and-add multiplication algorithm to compute $P = X \times Y$ for twos-complement numbers. Equations (4.20) and

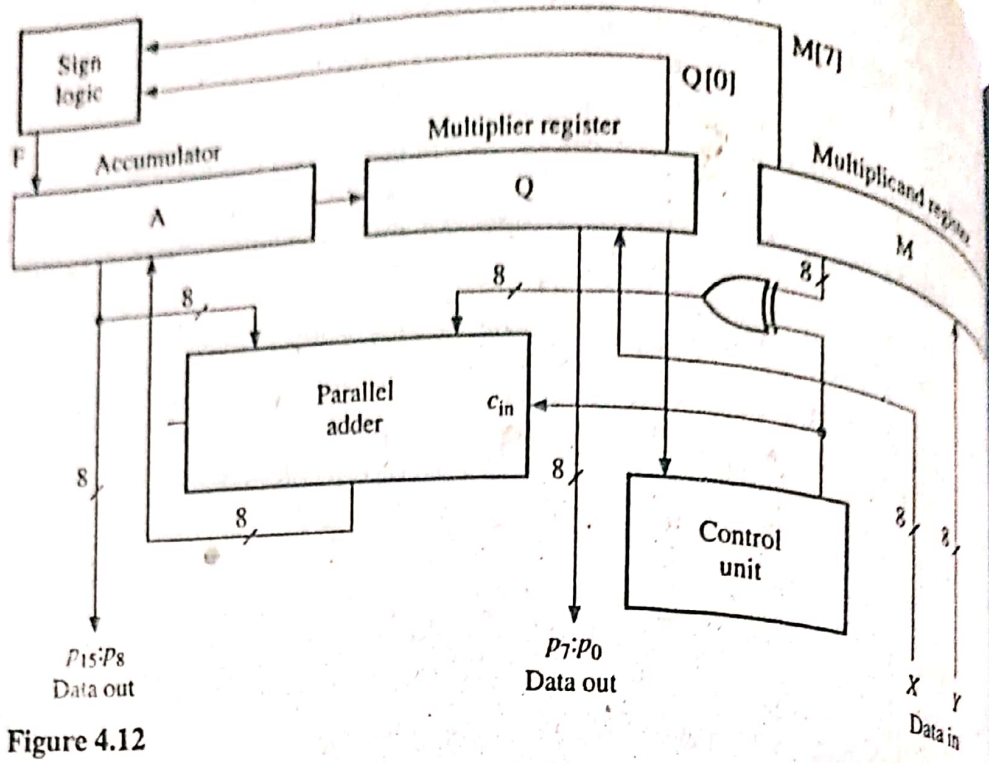


Figure 4.12
The datapath of the twos-complement multiplier.

(4.21) suggest that we can use an unsigned multiplication technique like the one illustrated in Figures 4.12 and 4.13 with one change: When multiplying by the sign bit, perform subtraction rather than addition in the final step if a minus sign x_{n-1} is encountered. This observation is the basis of a twos-complement multiplication algorithm developed by James E. Robertson, which has been widely used in computer design [Robertson 1955; Cavanagh 1984]. We now show one way to adapt the circuit developed in Example 2.7 for sign-magnitude multiplication to deal with the twos-complement case.

```
2Cmultiplier    (in: INBUS; out: OUTBUS);
                register A[7:0], M[7:0], Q[7:0], COUNT[2:0], F;
                bus INBUS[7:0], OUTBUS[7:0];
BEGIN:          A := 0, COUNT := 0, F := 0,
INPUT:          M := INBUS;
                Q := INBUS;
ADD:            A[7:0] := A[7:0] + M[7:0] × Q[0],
                F := (M[7] and Q[0]) or F;
RSHIFT:         A[7] := F, A[6:0].Q := A.Q[7:1], COUNT := COUNT + 1;
TEST:           if COUNT ≠ 7 then go to ADD;
SUBTRACT:       A[7:0] := A[7:0] – M[7:0] × Q[0], Q[0] := 0;
OUTPUT:         OUTBUS := Q;
                OUTBUS := A;
end 2Cmultiplier;
```

Figure 4.13
HDL description of the multiplier for 8-bit twos-complement fractions.

EXAMPLE 4.2 DESIGN OF A MULTIPLIER FOR TWOS-COMPLEMENT FRACTIONS. Consider again the task of multiplying two 8-bit binary fractions $X = x_7x_6x_5x_4x_3x_2x_1x_0$ and $Y = y_7y_6y_5y_4y_3y_2y_1y_0$ to form the product $P = Y \times X$, this time using twos-complement code. (Example 2.7 analyzed this problem for the sign-magnitude case.) Assume that the multiplier will have a register-level structure similar to that in Figure 2.41, with registers A, M, and Q storing the various operands and A.Q forming a right-shift register. Since sign bits will be included in additions and subtractions, we need an 8-bit adder-subtracter, rather than the 7-bit magnitude-only adder used in the earlier design. Figure 4.12 shows the datapath of the proposed design at the register level.

To develop the required twos-complement multiplication algorithm for this machine, we consider the four possible cases determined by the signs of X and Y.

1. $x_7 = y_7 = 0$; that is, both X and Y are positive. The computation in this case is effectively unsigned multiplication with the product P computed in a series of add-and-shift steps of the form

$$P_i := P_i + x_iY; \quad P_{i+1} := 2^{-1}P_i;$$

All partial products P_i are nonnegative, so leading 0s are introduced into A during the right-shift operation indicated by the factor 2^{-1}.

2. $x_7 = 0$, $y_7 = 1$; that is, X is positive and Y is negative. The partial product P_i will be zero, and leading 0s should be shifted into A as before, until the first 1 in X is encountered. Multiplication of Y by this 1 and addition of the result to A causes P_i to become negative, from which point on leading 1s rather than 0s must be shifted into A. These rules ensure that a right shift corresponds to division by 2 in twos-complement code.

3. $x_7 = 1$, $y_7 = 0$; that is, X is negative and Y is positive. This follows case 1 for the first seven add-and-shift steps yielding the partial product

$$P_7 = \sum_{i=0}^{6} 2^{i-7}x_iY$$

For the final step, often referred to as a correction step, the subtraction $P := P_7 - Y$ is performed. The result P is then given by

$$P = -Y + \sum_{i=0}^{6} 2^{i-7}x_iY = \left(-x_7 + \sum_{i=0}^{6} 2^{i-7}x_i\right)Y$$

which is $X \times Y$ by (4.21).

4. $x_7 = y_7 = 1$; that is, both X and Y are negative. The procedure used here follows case 2, with leading 0s (1s) being introduced into the accumulator whenever its contents are zero (negative). The correction (subtraction) step of case 3 is also performed, which ensures that the final product in A.Q is nonnegative.

Each addition/subtraction step can be performed in the usual twos-complement fashion by treating the sign bits like any other and ignoring overflow. Care is needed in the shift step to ensure that the correct new value is placed in the accumulator's sign position A[7]. This value must be a leading 0 if the current partial product in A.Q is positive or zero, and 1 if it is negative. We introduce a flip-flop F to control the values assigned to A[7]. F is initially set to 0, and is subsequently defined by

$$\text{F} := (y_7 \textit{ and } x_i) \textit{ or } \text{F}$$

Step	Action	F	Accumulator A	Register Q
	Initialize registers	0	00000000	10110011 = multiplier X
			11010101	= multiplicand Y
0		1	11010101	10110011
1	Add M to A	1	11101010	11011001
	Right-shift F.A.Q		11010101	
		1	10111111	11011001
2	Add M to A	1	11011111	11101100
	Right-shift F.A.Q		00000000	
		1	11011111	11101100
3	Add zero to A	1	11101111	11110110
	Right-shift F.A.Q		00000000	
		1	11101111	11110110
4	Add zero to A	1	11110111	11111011
	Right-shift F.A.Q		11010101	
		1	11001100	11111011
5	Add M to A	1	11100110	01111101
	Right-shift F.A.Q		11010101	
		1	10111011	01111101
6	Add M to A	1	11011101	10111110
	Right-shift F.A.Q		00000000	
		1	11011101	10111110
7	Add zero to A	1	11101110	11011111
	Right-shift F.A.Q		11010101	
		1	00011001	11011111
8	Subtract M from A		00011001	11011110 = product P
	Set Q[0] to 0			

Figure 4.14
Illustration of the Robertson multiplication algorithm for twos-complement fractions.

Here y_7 is the sign of the multiplicand stored in M[7], and x_i is the current multiplie being tested in Q[0]. Thus F is set to 1 if Y is negative and at least one nonzero encountered. Once set to 1, it remains at that value. A negative Y and a positive or ative X therefore produce a series of negative partial products. This situation is expected, since bits x_6:x_0 of the multiplier X are always treated as if they were posi A positive Y, or $X = 0$, causes F to remain permanently at 0. Note that the sign p_{15} product P requires no separate computational step. As in Example 2.7, the least sig cant bit p_0 of P is set to 0 to make the result exactly 16 bits long.

Figure 4.13 presents an HDL description of the twos-complement multiplic algorithm, which summarizes the foregoing analysis; compare the corresponding magnitude algorithm in Figure 2.39. An application of the present algorithm to the $X = 10110011$ and $Y = 11010101$ appears in Figure 4.14. The sign bit x_7 of the plier X is underlined to show its passage through Q. Observe how F becomes 1 in 1, when the negative multiplicand is first added to the accumulator. F continues to ply leading 1s to the A register until step 8. Then because $Q[7] = x_7 = 1$, a subtracti performed that produces the proper sign $p_{15} = 0$ in A(0). Setting $Q[0] = p_0$ to 0 c pletes the multiplication process.

Booth's algorithm. Another interesting and widely used scheme for tw complement multiplication was proposed by Andrew D. Booth in the 19

Booth 1951]. Like Robertson's method in Example 4.2, Booth's algorithm employs both addition and subtraction, but it treats positive and negative operands uniformly—no special actions are required for negative numbers. Booth's algorithm can also be readily extended in various ways to speed up the multiplication process; see problems 4.16 and 4.17. A version of this algorithm implements the MC68000's multiply instruction.

The multiplication algorithms we have considered so far involve scanning the multiplier X from right to left and using the value of the current multiplier bit x_i to determine which of the following operations to perform: add the multiplicand Y, subtract Y, or add zero, that is, no operation. In Booth's approach two adjacent bits $x_i x_{i-1}$ are examined in each step. If $x_i x_{i-1} = 01$, then Y is added to the current partial product P_i, while if $x_i x_{i-1} = 10$, Y is subtracted from P_i. If $x_i x_{i-1} = 00$ or 11, then neither addition or subtraction is performed; only the subsequent right shift of P_i takes place. Thus Booth's algorithm effectively skips over runs of 1s and runs of 0s that it encounters in X. This skipping reduces the average number of add-subtract steps and allows faster multipliers to be designed, although at the expense of more complex timing and control circuitry.

The validity of Booth's method can be seen as follows. Suppose that X is a positive integer and contains a subsequence X^* consisting of a run of k 1s flanked by two 0s.

$$\begin{aligned} X^* &= x_i x_{i-1} x_{i-2} \ldots x_{i-k+1} x_{i-k} x_{i-k-1} \\ &= 0\,1\,1 \ldots 1\,1\,0 \end{aligned}$$

In a direct add-and-shift multiplication algorithm such as Robertson's, Y is multiplied by each bit of X^* in sequence and the results are summed so that X^*'s contribution to the product $P = X \times Y$ is

$$\sum_{j=i-k}^{i-1} 2^j Y \tag{4.22}$$

Now when Booth's algorithm is applied to X^*, it performs an addition when it encounters $x_i x_{i-1} = 01$, which contributes $2^i Y$ to P. It performs a subtraction at $x_{i-k} x_{i-k-1} = 10$, which contributes $-2^{i-k} Y$ to P. Thus the net contribution of X^* to the product P in this case is

$$\begin{aligned} 2^i Y - 2^{i-k} Y &= 2^{i-k} Y (2^k - 1) Y \\ &= 2^{i-k} \sum_{m=0}^{k-1} 2^m Y \\ &= \sum_{m=0}^{k-1} 2^{m+i-k} Y \end{aligned} \tag{4.23}$$

Suppose the index m is replaced by $j = m + i - k$. Then the upper and lower limits of the summation in (4.23) change from $k - 1$ and 0 to $i - 1$ and $i - k$, respectively, implying that (4.22) and (4.23) are, in fact, the same. It follows that Booth's algorithm correctly computes the contribution of X^*, and hence of the entire multiplier X, to the product P. Equation (4.20) implies that the contribution of a negative X^*

```
BoothMult       (in: INBUS; out: OUTBUS);
                register A[7:0], M[7:0], Q[7:–1], COUNT[2:0],
                bus INBUS[7:0], OUTBUS[7:0];
                A := 0, COUNT := 0,
BEGIN:
INPUT:          M := INBUS;
                Q[7:0] := INBUS, Q[–1] := 0;
SCAN:           if Q[1] Q[0] = 01 then A[7:0] := A[7:0] + M[7:0], go to
                TEST;
                  else if Q[1] Q[0] = 10 then A[7:0] := A[7:0] – M[7:0];
TEST:           if COUNT = 7 then go to OUTPUT,
RSHIFT:         A[7] := A[7], A[6:0].Q = A.Q[7:0],
INCREMENT:      COUNT := COUNT + 1, go to SCAN;
OUTPUT:         OUTBUS := A, Q[0] := 0;
                OUTBUS := Q[7:0];
end BoothMult;
```

Figure 4.15
HDL description of an 8-bit multiplier implementing the basic Booth algorithm.

to P can also be expressed in the formats of (4.20) and (4.23); a similar argume demonstrates the correctness of the algorithm for negative multipliers. The arg ment for fractions is essentially the same as that for integers.

The twos-complement multiplication circuit of Figure 4.12 can easily be mod ified to implement Booth's algorithm. Figure 4.15 describes a straightforwa implementation of the Booth algorithm using the above approach with $n = 8$ and circuit based on Figure 4.12. An extra flip-flop Q[–1] is appended to the right en of the multiplier register Q, and the sign logic for A is reduced to the simple sig extension A[7] := A[7]. In each step the two adjacent bits Q[0]Q[–1] of Q ar examined, instead of Q[0] alone as in Robertson's algorithm, to decide the opera tion (add Y, subtract Y, or no operation) to be performed in that step. For compari son with Robertson's method in Figure 4.13, the operands are assumed to b fractions. The application of this algorithm to the example solved by Robertson method in Figure 4.14 appears in Figure 4.16, where the bits stored in Q[0]Q[–1] i each step are underlined.

Combinational array multipliers. Advances in VLSI technology have made possible to build combinational circuits that perform $n \times n$-bit multiplication fo fairly large values of n. An example is the Integrated Device Technology IDT721CL multiplier chip, which can multiply two 16-bit numbers in 16 *ns* [Inte grated Device Technology 1995]. These multipliers resemble the n-step sequential multipliers discussed above but have roughly n times more logic to allow the prod uct to be computed in one step instead of in n steps. They are composed of arrays of simple combinational elements, each of which implements an add/subtract-and shift operation for small slices of the multiplication operands.

Suppose that two binary numbers $X = x_{n-1}x_{n-2}\ldots x_1x_0$ and $Y = y_{n-1}y_{n-2}\ldots y_1y$ are to be multiplied. For simplicity, assume that X and Y are unsigned integers. Th product $P = X \times Y$ can therefore be expressed as

Action	Accumulator	Register Q
Initialize registers	00000000	10110011 = multiplier X
Set Q[−1] to 0	00000000	101100110
	11010101	= mulitplicand Y = M
Subtract M from A	00101011	101100110
Right-shift A.Q	00010101	110110011
Skip add/subtract	00010101	110110011
Right-shift A.Q	00001010	111011001
	11010101	
Add M to A	11011111	111011001
Right-shift A.Q	11101111	111101100
Skip add/subtract	11101111	111101100
Right-shift A.Q	11110111	111110110
	11010101	
Subtract M from A	00100010	111110110
Right-shift A.Q	00010001	011111011
Skip add/subtract	00010001	011111011
Right-shift A.Q	00001000	101111101
	11010101	
Add M to A	11011101	101111101
Right-shift A.Q	11101110	110111110
	11010101	
Subtract M from A	00011001	110111110
Set Q[0] to 0	00011001	110111100 = product P

ure 4.16
ration of the Booth multiplication algorithm.

$$P = \sum_{i=0}^{n-1} 2^i x_i Y \tag{4.24}$$

esponding to the bit-by-bit multiplication style of Figure 4.10. Now (4.24) can written as

$$P = \sum_{i=0}^{n-1} 2^i \left(\sum_{j=0}^{n-1} x_i y_j 2^j \right) \tag{4.25}$$

h of the n^2 1-bit product terms $x_i y_j$ appearing in (4.25) can be computed by a -input AND gate—observe that the arithmetic and logical products coincide e 1-bit case. Hence an $n \times n$ array of two-input ANDs of the type shown in ure 4.17 can compute all the $x_i y_j$ terms simultaneously. The terms are ined according to (4.25) by an array of $n(n-1)$ 1-bit full adders as shown in ure 4.18; this circuit is a kind of two-dimensional ripple-carry adder. The s implied by the 2^i and 2^j factors in (4.25) are implemented by the spatial dis- ement of the adders along the x and y dimensions. Note the similarities ween the circuit of Figure 4.17 and the multiplication examples of Figures 0 and 4.11.

The AND and add functions of the array multiplier can be combined into a single component (cell) as illustrated in Figure 4.19. This cell realizes the arithmetic expression

$$c_{out}s = a \textit{ plus } b \textit{ plus } xy \tag{4.}$$

An $n \times n$-bit multiplier can be built using n^2 copies of this cell as the sole component, although, as in Figure 4.18, some cells on the periphery of the array have inputs set to 0 or 1, effectively reducing their operation from (4.26) a *plus* b *plus* xy to a *plus* b (a half adder). The multiplication time for this multiplier is determined by the worst-case carry propagation and, ignoring the differences between internal and peripheral cells, is $(2n-1)D$, where D is the delay of the basic cell.

Multiplication algorithms for twos-complement numbers, such as Robertson's and Booth's, can also be realized by arrays of combinational cells as the following example shows.

EXAMPLE 4.3 ARRAY IMPLEMENTATION OF THE BOOTH MULTIPLICATION ALGORITHM [KOREN 1993]. Implementing the Booth method by a combinational array requires a multifunction cell capable of addition, subtraction, and operation (skip). Such a cell B is shown in Figure 4.20*a*. Its various functions are selected by a pair of control lines H and D as indicated. It is easily seen that the required functions of B are defined by the following logic equations.

$$z = a \oplus bH \oplus cH$$

$$c_{out} = (a \oplus D)(b + c) + bc$$

When $HD = 10$, these equations reduce to the usual full-adder equations (4.1); when $HD = 11$, they reduce to the corresponding full-subtracter equations

$$z = a \oplus \bar{b} \oplus c$$

$$c_{out} = \bar{a}b + \bar{a}c + bc$$

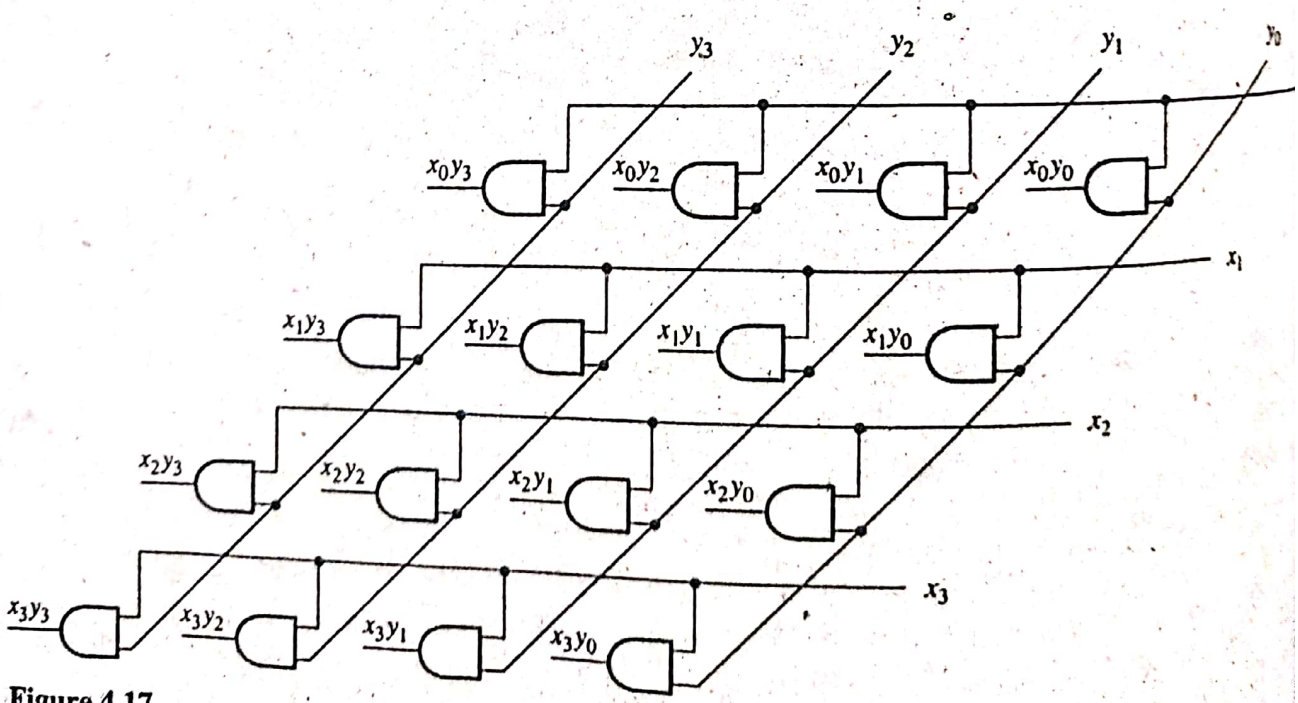


Figure 4.17
AND array for 4 × 4-bit unsigned multiplication.

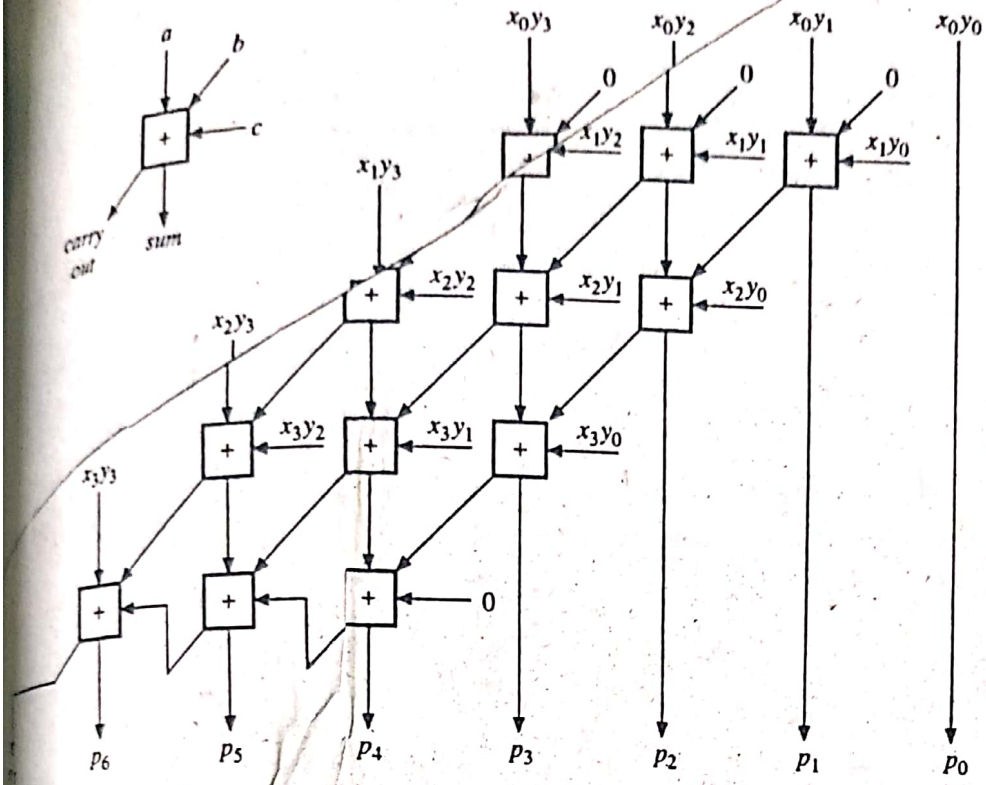


Figure 4.18
Full-adder array for 4 × 4-bit unsigned multiplication.

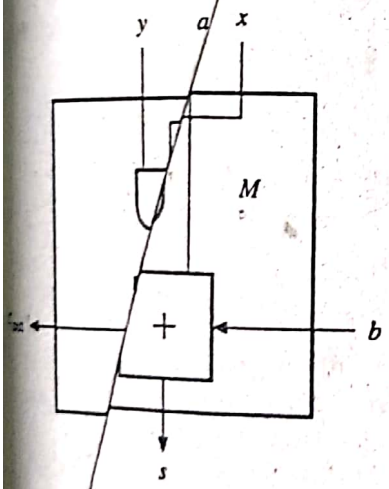
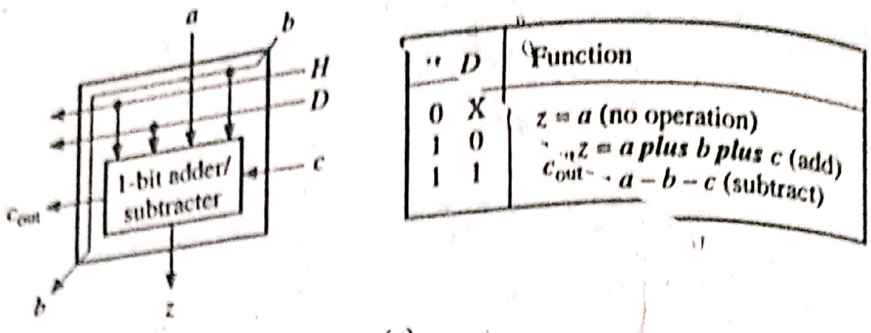


Figure 4.19
Cell *M* for an unsigned array multiplier.

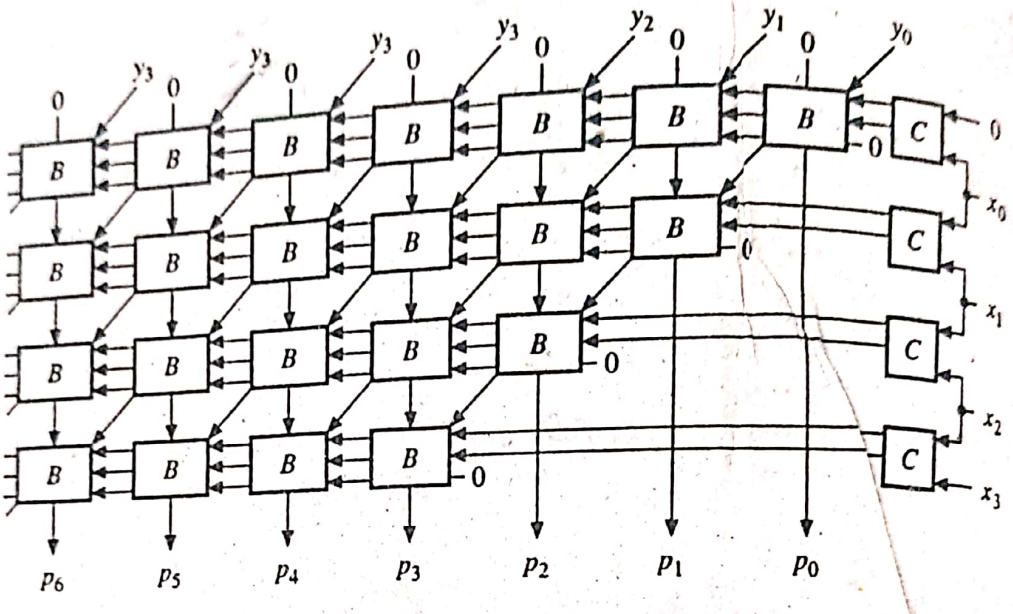
in which c and c_{out} assume the roles of borrow-in and borrow-out, respectively. When $H = z$ becomes a, and the carry lines play no role in the final result.

An n-bit multiplier is constructed from $n^2 + n(n-1)/2$ copies of the B cell connected as shown in Figure 4.20*b*. The extra cells at the top left change the array's shape from the parallelogram of Figure 4.18 to a trapezium and are employed to sign-extend the multiplicand Y for addition and subtraction. Note how the diagonal lines marked b deliver the sign-extended Y directly to every row of B cells. When Y is positive, it is sign-extended by leading 0s; this is implicit in the array of Figure 4.18. In the present case, when Y is negative, it must be explicitly sign-extended by leading 1s.

The operation to be performed by each row i of B cells is decided by bits $x_i x_{i-1}$ of the operand X. To allow each possible $x_i x_{i-1}$ pair to control row operations, we



(*a*)



(*b*)

Figure 4.20
Combinational array implementing Booth's algorithm: (*a*) main cell *B* and (*b*) array multiplier for 4 × 4-bit numbers.

introduce a second cell type denoted *C* in Figure 4.20*b* to generate the control inp signal *H* and *D* required by the *B* cells. Cell *C* compares x_i with x_{i-1} and generat the values of *HD* required by Figure 4.20*a*; these values are as follows:

$$H = \overline{x_i \oplus x_{i-1}}$$
$$D = x_i \bar{x}_{i-1}$$

4.1.3 Division

In fixed-point division two numbers, a divisor *V* and a dividend *D*, are given. T object is to compute a third number *Q*, the quotient, such that $Q \times V$ equals or very close to *D*. For example, if unsigned integer formats are being used, *Q* is com puted so that

$$D = Q \times V + R$$

R, the remainder, is required to be less than V, that is, $0 \le R < V$. We can then

$$D/V = Q + R/V \tag{4.27}$$

R/V is a small quantity representing the error in using Q alone to represent this error is zero if $R = 0$.

Preliminaries. The relationship $D \approx Q \times V$ suggests that a close correspondence exists between division and multiplication, specifically the dividend, quotient and divisor correspond to the product, multiplicand, and multiplier, respectively. This correspondence means that similar algorithms and circuits can be used for multiplication and division. In multiplication the shifted multiplier is added to the multiplicand to form the product. In division the shifted divisor is subtracted from the dividend to form the quotient. Just as multiplication ends with a double-length product, division often begins with a double-length dividend. Despite these similarities, division is a more difficult operation than multiplication because to determine a particular quotient bit q_i, we have to answer the question: How many multiples is the divisor V of the current partial dividend D_i? This question is typically answered by trial and error: Multiply V by a trial value for q_i, subtract the result from D_i, and check the value of the remainder. Note too that the next quotient bit q_{i+1} cannot be determined until q_i is known. Thus division has an element of uncertainty not found in multiplication.

One of the simpler binary division methods is a sequential digit-by-digit algorithm similar to that used in pencil-and-paper methods with decimal numbers. Figure 4.21 illustrates this approach for a 3-bit divisor $V = 101$ and a 6-bit dividend $D = 100110$. The dividend is scanned from left to right, and the quotient is computed bit by bit. In each step divisor V is compared to the current partial dividend D_i, referred to here as the partial remainder R_i.[2] The current quotient bit q_i is either 0 or 1, and is determined by comparing V with R_i; this comparison is the crucial part of division. Note that decimal division is harder than binary in this

0111	Quotient $Q = q_3q_2q_1q_0$
$V = 101$ | 100110	Dividend $D = R_0$
000	q_3V
100110	R_1
101	$q_2 2^{-1} V$
10010	R_2
101	$q_1 2^{-2} V$
1000	R_3
101	$q_0 2^{-3} V$
011	R_4 = remainder R

Figure 4.21
Sequential pencil-and-paper method for division of unsigned numbers.

[2] We use the terms *partial dividend* and *partial remainder* interchangeably because the remainder from step i is used as the dividend in step $i + 1$.

regard because q_i must be selected from 10 possible digit values instead of two. If the numbers appearing in the division calculation of Figure 4.21 are unsigned binary integers of length six, then (4.27) becomes

$$100110./000101. = 000111. + 000011./000101.$$

corresponding to the decimal division 38/5 = 7 + 3/5. If the numbers are unsigned 6-bit fractions, then Figure 4.21 is interpreted as

$$.100110/.101000 = .111000 + .000011/.101000$$

corresponding to .59375/.625 = .875 + .046875/.625.

In integer arithmetic Q and R are always integers of the standard word size. If fraction formats are used, however, the number of bits of Q is not necessarily bounded. For example, .2000/.3000 = .66666..., a repeating fraction. It is necessary, therefore, to limit the number of quotient bits generated by the division process. Division of .2000 by .3000 might be required to yield a four-digit quotient with truncation or rounding determining the final digit of Q. Several other difficulties occur in division. If D is too large relative to V, then Q will not fit in the standard word size, resulting in *quotient overflow*. For instance, the four-digit fraction division .2000/.0100 produces a nonfraction six-digit result 20.0000. When V is 0, the quotient Q is treated as undefined or infinity and a *divide-by-zero error* is said to occur. Special circuits are employed to check for, and flag, quotient overflow and zero divisors before division begins.

Basic algorithms. Suppose that the divisor V and dividend D are unsigned integers and the quotient $Q = q_{n-1}q_{n-2}q_{n-3}\ldots$ is to be computed one bit at a time. At each step i, $2^{-i}V$, which represents the divisor shifted i bits to the right, is compared with the current partial remainder R_i. The quotient bit q_i is set to 1 (0) if $2^{-i}V$ is (greater) than R_i, and a new partial remainder R_{i+1} is computed according to the relation

$$R_{i+1} := R_i - q_i 2^{-i} V \tag{4.28}$$

In machine implementations it is more convenient to shift the partial remainder to the left relative to a fixed divisor, in which case (4.28) is replaced by

$$R_{i+1} := 2R_i - q_i V$$

Figure 4.22 shows the calculation of Figure 4.21 modified in this way. The final partial remainder R_4 is now the overall remainder R shifted three bits to the left, so that $R = 2^{-3}R_4$.

As observed above, the central problem in division is finding the quotient digit q_i. If radix-r numbers are being represented, then q_i must be chosen from among r possible values. When $r = 2$, q_i can be generated by comparing V and $2R_i$ in the ith step, as is done implicitly in Figure 4.22. If $V > 2R_i$, then $q_i = 0$; otherwise, $q_i = 1$. If V is long, a combinational magnitude comparator circuit may be impractical, in which case q_i is usually determined by subtracting V from $2R_i$ and examining the sign of $2R_i - V$. If $2R_i - V$ is negative, $q_i = 0$; otherwise, $q_i = 1$.

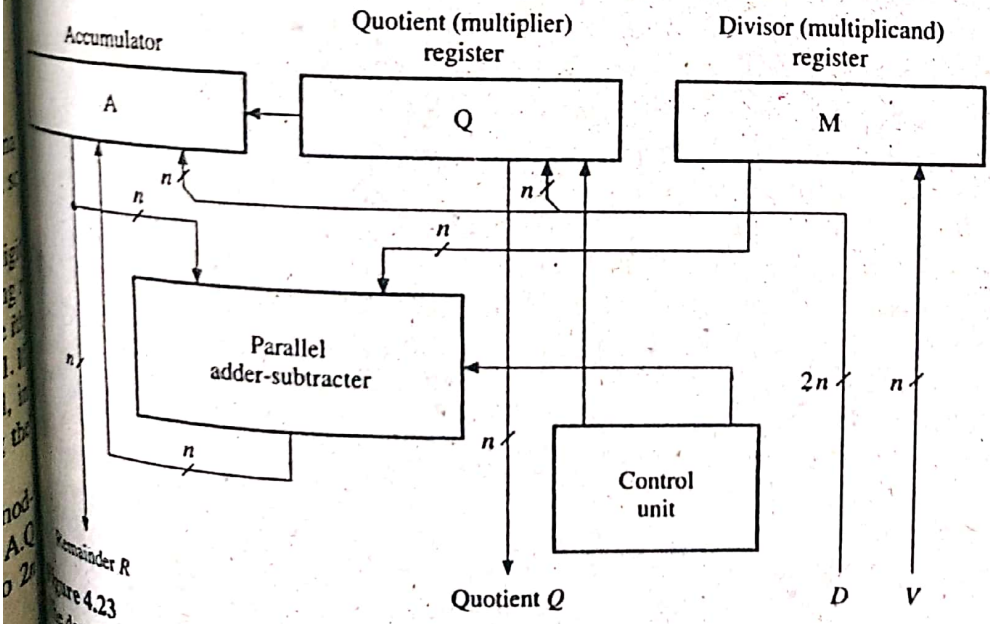
The circuit used for multiplication in Example 4.2 (Figure 4.12) is easily modified to perform division, as shown in Figure 4.23. The $2n$-bit shift register stores the partial remainders. Initially the dividend (which can contain up to

		Quotient Q
100110	Dividend $D = 2R_0$	
000	q_3V	0
100110	R_1	
1001100	$2R_1$	
101	q_2V	01
100100	R_2	
1001000	$2R_2$	
101	q_1V	011
100000	R_3	
1000000	$2R_3$	
101	q_0V	0111
011000	$R_4 = 2^3R$	

ure 4.22
e division of Figure 4.21 modified for machine implementation.

its) is placed in A.Q. The divisor V is placed in the M register where it remains roughout the division process. In each step A.Q is shifted to the left. The positions vacated at the right-most end of the Q register can be used to store the quotient bits as they are generated. When the division process terminates, Q contains the quotient, while A contains the (shifted) remainder.

As noted already, the quotient bit q_i can be determined by a trial subtraction of the form $2R_i - V$. This subtraction also yields the new partial remainder R_{i+1} when $2R_i - V$ is positive; that is, when $q_i = 1$. Clearly, the process of determining q_i and R_{i+1} can be integrated. Two major division algorithms are distinguished by the way they combine the computation of q_i and R_{i+1}. If $q_i = 0$, then the result of the trial



ure 4.23
e datapath of a sequential n-bit binary divider.

subtraction is $2R_i - V$; however, the required new partial remainder R_{i+1} is $2R_i$. The partial remainder R_{i+1} can be obtained by adding V back to the result of the first subtraction. This straightforward technique is called *restoring division*. In every step the operation

$$R_{i+1} := 2R_i - V \tag{4.29}$$

is performed. When the result of the subtraction is negative, a restoring addition is performed as follows:

$$R_{i+1} := R_{i+1} + V$$

If the probability of $q_i = 1$ is 1/2, then this algorithm requires n subtractions and an average of $n/2$ additions.

The restoration step of the preceding algorithm is eliminated in a slightly different technique called *nonrestoring division*. This method is based on the observation that a restoration of the form

$$R_i := R_i + V \tag{4.30}$$

is followed in the next step by the subtraction (4.29). Operations (4.29) and (4.30) can be merged into the single operation

$$R_{i+1} := 2R_i + V \tag{4.31}$$

Thus when $q_i = 1$, which is indicated by a positive value of R_i, R_{i+1} is computed using (4.29). When $q_i = 0$, R_{i+1} is computed using (4.31). The calculation of each quotient bit involves either an addition or a subtraction, but not both. Nonrestoring division therefore requires n additions or subtractions, whereas restoring division requires an average of $3n/2$ additions and subtractions.

Figure 4.24 presents a nonrestoring division algorithm designed for the circuit of Figure 4.23 with unsigned integers. The divisor V and quotient Q are n bits long (with leading 0s if necessary), while the dividend D is up to $2n - 1$ bits long, which is the maximum length of the product of two n-bit integers. The flip-flop S is appended to the accumulator A to record the sign of the result of an addition or subtraction and to determine the quotient bit. Each new quotient bit is placed in $Q[0]$, and the final values of the quotient Q and the remainder R are in the Q and A registers, respectively. An application of this algorithm when $n = 4$ appears in Figure 4.25 with $D = 1100001_2 = 97_{10}$ and $V = 1010_2 = 10_{10}$.

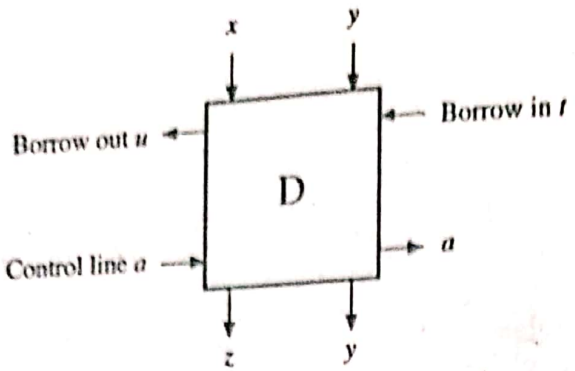
The restoring and nonrestoring division techniques can be extended to signed numbers in much the same way as multiplication. Sign-magnitude numbers present few difficulties; the magnitudes of the quotient and remainder can be computed as in the unsigned number case, while their signs are determined separately. As remarked in [Cavanagh 1984], there are no simple division algorithms for handling negative numbers directly in twos-complement code because of the difficulty of selecting the quotient bits so that the quotient has the correct positive or negative representation. The most direct approach to signed division is to negate any negative operands, perform division on the resulting positive numbers, and then negate the results, as needed. A fast division algorithm for twos-complement numbers based on the nonrestoring approach was devised independently in 1958 by Daggett, W. Sweeney, James E. Robertson, and Keith D. Tocher and is called the *SRT* method in their honor; see [Cavanagh 1984; Koren 1993] for details.

```
NRdivider   (in: INBUS; out: OUTBUS);
            register S, A[n–1:0], M[n–1:0], Q[n–1:0], COUNT[⌈log2n⌉:0];
            bus INBUS[n–1:0], OUTBUS[n–1:0];
            COUNT := 0, S := 0,
BEGIN:      A := INBUS; {Input the left half of the dividend D}
INPUT:      Q := INBUS; {Input the right half of the dividend D}
            M := INBUS; {Input the divisor V}
SUBTRACT:   S.A := S.A – M; {S is the sign of the result}
TEST:       if S = 0 then
              begin Q[0] := 1;
              if COUNT = n – 1 then go to CORRECTION; else
                begin COUNT := COUNT + 1, S.A.Q[n–1:1] := A.Q; end
              S.A := S.A – M, go to TEST; end
            else {if S = 1}
              begin Q[0] := 0;
              if COUNT = n – 1 then go to CORRECTION; else
                begin COUNT := COUNT + 1, S.A.Q[n–1:1] := A.Q; end
            S.A := S.A + M, go to TEST; end
CORRECTION: if S = 1 then S.A := S.A + M;
OUTPUT:     OUTBUS := Q; {Output the quotient Q}
            OUTBUS := A; {Output the remainder R}
end NRdivider;
```

Figure 4.24
Nonrestoring division algorithm for unsigned integers.

Step	Action	S	A	Q
0	Initialize registers	0	1100	0010 = dividend D
1			1010	= divisor V = M
	Subtract M from A	0	0010	0010
	Reset Q[0]	0	0010	0011
	Left shift S.A.Q	0	0100	0110
2			1010	
	Subtract M from A	1	1010	0110
	Set Q[0]	1	1010	0110
	Left shift S.A.Q	1	0100	1100
3			1010	
	Add M to A	1	1110	1100
	Set Q[0]	1	1110	1100
	Left shift S.A.Q	1	1101	1000
4			1010	
	Add M to A	0	0111	1000
	Reset Q[3]	0	0111	1001
				1001 = quotient Q
			0111	= remainder R

Figure 4.25
Illustration of the nonrestoring division algorithm for unsigned integers.



a	Function
0	*uz = x minus y minus t*
1	*z = x*

Figure 4.26
A cell D for array implementation of restoring division.

Combinational array dividers. Combinational array circuits can be used for division as well as for multiplication. Figure 4.26 shows a cell D suitable for implementing a version of the restoring division algorithm. This cell is basically a full subtracter with t and u being the borrow-in and borrow-out bits, respectively. The main output z is controlled by input a. When $a = 1$, z is the difference bit defined by the arithmetic equation

$$z = x \textit{ minus } y \textit{ minus } t$$

When $a = 0$, $z = x$. Thus the behavior of the cell D is given by the logic equations

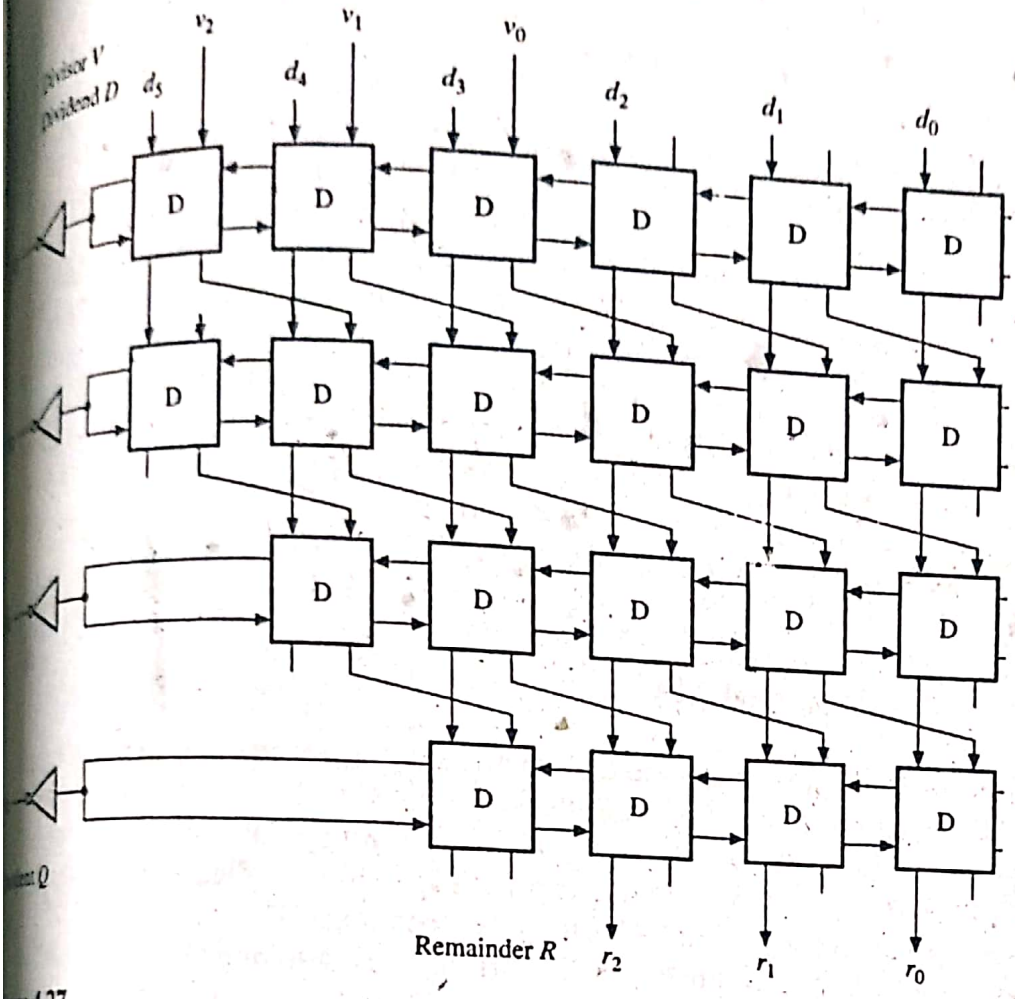
$$z = x \oplus \bar{a}(y \oplus t)$$

$$u = \bar{x}y + \bar{x}t + yt$$

Figure 4.27 shows an array of D cells to divide 3-bit unsigned integers and generate a 4-bit quotient. Each row of the array subtracts the divisor V from the shifted partial remainder $2R_i$ generated by the row above. The sign of the result, and therefore of the quotient bit, is indicated by the borrow-out signal from the left-most cell in the row. This signal u_i is connected to the control inputs a of all cells in the same row. If $u_i = 0$, then the output from the row is $2R_i - V$ and $q_i = \bar{u}_i = 1$. If $u_i = 1$, then the output from the row is restored to $2R_i$, and again $q_i = \bar{u}_i = 0$. Thus the output of each row is initially $2R_i - V$, but it is restored to $2R_i$ when required. Restoration is achieved by overriding the subtraction performed by the row rather than by explicitly adding back the divisor.

Let d and d' be the carry (borrow) propagation and restore times of a cell, respectively. Let the divisor and dividend be n bits long. Each row of the divider array functions as an n-bit ripple-borrow subtracter, so the maximum time required to compute one quotient bit is $nd + d'$. The time required to compute an m-bit quotient and the corresponding remainder is therefore $m(nd + d')$, and the number of cells needed is $m(n + 1) - 1$.

Division by repeated multiplication. In systems containing a high-speed multiplier, division can be performed efficiently and at low cost using repeated multiplication. In each iteration a factor F_i is generated and used to multiply both the divisor V and the dividend D. Therefore



ure 4.27
ider array for 3-bit unsigned numbers using the cell D of Figure 4.26.

$$Q = \frac{D \times F_0 \times F_1 \times F_2 \times \ldots}{V \times F_0 \times F_1 \times F_2 \times \ldots}$$

chosen so that the sequence $V \times F_0 \times F_1 \times F_2 \ldots$ converges rapidly toward . Hence $D \times F_0 \times F_1 \times F_2 \ldots$ must converge toward the desired quotient.
The convergence of the method depends on the selection of the F_i's. For simply, assume that D and V are positive normalized fractions so that $V = 1 - x$, ere $x < 1$. Set $F_0 = 1 + x$. We can now write

$$V \times F_0 = (1 - x)(1 + x) = 1 - x^2$$

ly $V \times F_0$ is closer to one than to V. Next set $F_1 = 1 + x^2$. Hence

$$V \times F_0 \times F_1 = (1 - x^2)(1 + x^2) = 1 - x^4$$

so on. Let V_i denote $V \times F_0 \times F_1 \times \ldots \times F_i$. The multiplication factor at each e is computed as $F_i = 2 - V_{i-1}$, which is simply the twos-complement of V_{i-1}.
ce

$$F_i = 1 + x^{2^i} \text{ and } V_i = 1 - x^{2^{i+1}}$$

increases, V_i converges quickly toward one. The process terminates when $V_i =$ ...11, the number closest to one for the given word size.

4.2 ARITHMETIC-LOGIC UNITS

The various circuits used to execute data-processing instructions are usually combined in a single circuit called an arithmetic-logic unit or ALU. The complexity of an ALU is determined by the way in which its arithmetic instructions are realized. Simple ALUs that perform fixed-point addition and subtraction, as well as word-based logical operations, can be realized by combinational circuits. ALUs that perform multiplication and division can be constructed around the circuitry developed for these operations in the preceding section. Much more extensive processing and control logic is necessary to implement floating-point arithmetic hardware, as we will see later. Some processors having fixed-point ALUs employ special-purpose auxiliary units called arithmetic (co)processors to perform floating-point and other complex numerical functions.

4.2.1 Combinational ALUs

The simplest ALUs combine the functions of a twos-complement adder-subtracter with those of a circuit that generates word-based logic functions of the form, for example, AND, XOR, and NOT. They can thus implement most of a computer's fixed-point data-processing instructions. Figure 4.28 outlines an ALU that has separate subunits for logical and arithmetic operations. The particular class of operation (logical and arithmetic) to be performed is determined by a "mode" control line M attached to a two-way multiplexer that channels the required result

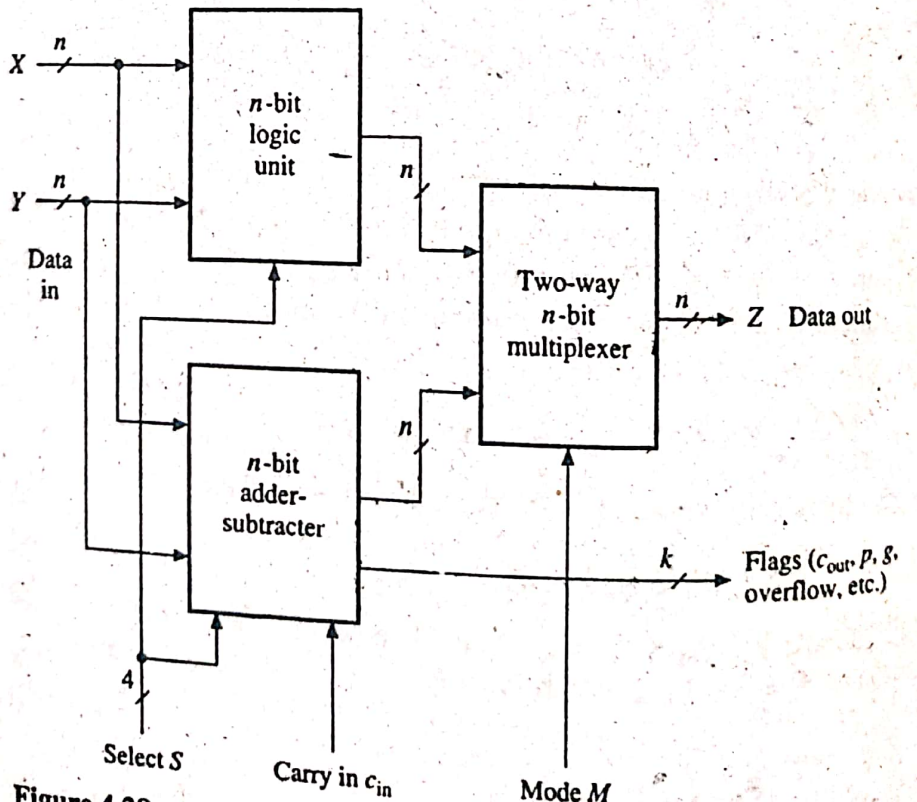


Figure 4.28
A basic n-bit arithmetic-logic unit (ALU).