CHAPTER EIGHT

Central Processing Unit

IN THIS CHAPTER

8-1 Introduction

CPU

The part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU. The CPU is made up of three major parts, as shown in Fig. 8-1. The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions. The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

The CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer. Computer architecture is sometimes defined as the computer structure and behavior as seen by the programmer that uses machine language instructions. This includes the instruction formats, addressing modes, the instruction set, and the general organization of the CPU registers.

One boundary where the computer designer and the computer programmer see the same machine is the part of the CPU associated with the instruction set. From the designer's point of view, the computer instruction set provides the specifications for the design of the CPU. The design of a CPU is

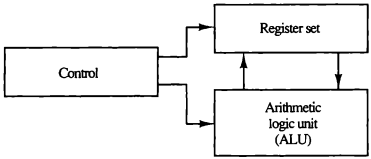


Figure 8-1 Major components of CPU.

a task that in large part involves choosing the hardware for implementing the machine instructions. The user who programs the computer in machine or assembly language must be aware of the register set, the memory structure, the type of data supported by the instructions, and the function that each instruction performs.

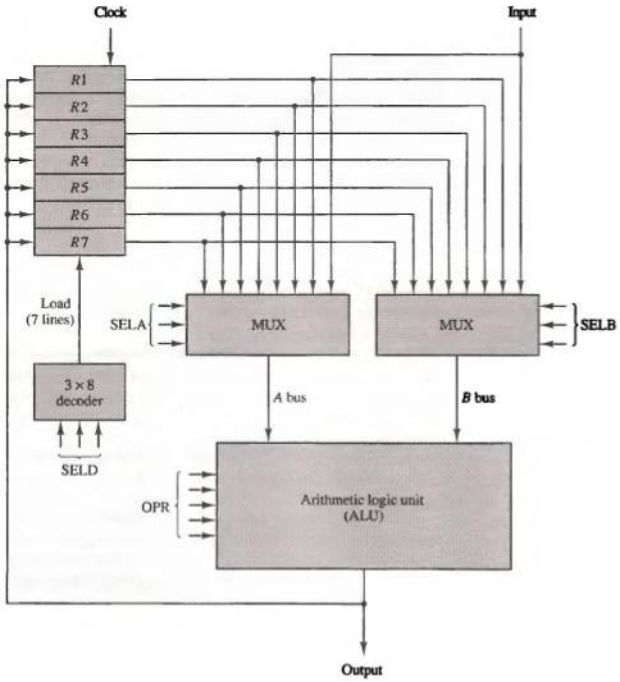
Design examples of simple CPUs are carried out in Chaps. 5 and 7. This chapter describes the organization and architecture of the CPU with an emphasis on the user's view of the computer. We briefly describe how the registers communicate with the ALU through buses and explain the operation of the memory stack. We then present the type of instruction formats available, the addressing modes used to retrieve data from memory, and typical instructions commonly incorporated in computers. The last section presents the concept of reduced instruction set computer (RISC).

8-2 General Register Organization

In the programming examples of Chap. 6, we have shown that memory locations are needed for storing pointers, counters, return addresses, temporary results, and partial products during multiplication. Having to refer to memory locations for such applications is time consuming because memory access is the most time-consuming operation in a computer. It is more convenient and more efficient to store these intermediate values in processor registers. When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfers, but also while performing various microoperations. Hence it is necessary to provide a common unit that can perform all the arithmetic, logic, and shift microoperations in the processor.

bus system

A bus organization for seven CPU registers is shown in Fig. 8-2. The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a



(a) Block diagram

(b) Control word

Figure 8-2 Register set with common ALU.

common arithmetic logic unit (ALU). The operation selected in the ALU determines the arithmetic or logic microoperation that is to be performed. The result of the microoperation is available for output data and also goes into the inputs of all the registers. The register that receives the information from the output bus is selected by a decoder. The decoder activates one of the register load inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.

The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system. For example, to perform the operation

$$R1 \leftarrow R2 + R3$$

the control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of $R2$ into bus A.
2. MUX B selector (SELB): to place the content of $R3$ into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition $A + B$.
4. Decoder destination selector (SELD): to transfer the content of the output bus into $R1$.

The four control selection variables are generated in the control unit and must be available at the beginning of a clock cycle. The data from the two source registers propagate through the gates in the multiplexers and the ALU, to the output bus, and into the inputs of the destination register, all during the clock cycle interval. Then, when the next clock transition occurs, the binary information from the output bus is transferred into $R1$. To achieve a fast response time, the ALU is constructed with high-speed circuits. The buses are implemented with multiplexers or three-state gates, as shown in Sec. 4-3.

Control Word

control word

There are 14 binary selection inputs in the unit, and their combined value specifies a *control word*. The 14-bit control word is defined in Fig. 8-2(b). It consists of four fields. Three fields contain three bits each, and one field has five bits. The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU. The three bits of SELD select a destination register using the decoder and its seven load outputs. The five bits of OPR select one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specify a particular microoperation.

The encoding of the register selections is specified in Table 8-1. The 3-bit

TABLE 8-1 Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	*R1*	*R1*	*R1*
010	*R2*	*R2*	*R2*
011	*R3*	*R3*	*R3*
100	*R4*	*R4*	*R4*
101	*R5*	*R5*	*R5*
110	*R6*	*R6*	*R6*
111	*R7*	*R7*	*R7*

binary code listed in the first column of the table specifies the binary code for each of the three fields. The register selected by fields SELA, SELB, and SELD is the one whose decimal number is equivalent to the binary number in the code. When SELA or SELB is 000, the corresponding multiplexer selects the external input data. When SELD = 000, no destination register is selected but the contents of the output bus are available in the external output.

ALU

The ALU provides arithmetic and logic operations. In addition, the CPU must provide shift operations. The shifter may be placed in the input of the ALU to provide a preshift capability, or at the output of the ALU to provide postshifting capability. In some cases, the shift operations are included with the ALU. An arithmetic logic and shift unit was designed in Sec. 4-7. The function table for this ALU is listed in Table 4-8. The encoding of the ALU operations for the CPU is taken from Sec. 4-7 and is specified in Table 8-2. The OPR field has five bits and each operation is designated with a symbolic name.

TABLE 8-2 Encoding of ALU Operations

OPR Select	Operation	Symbol
00000	Transfer *A*	TSFA
00001	Increment *A*	INCA
00010	Add *A* + *B*	ADD
00101	Subtract *A* − *B*	SUB
00110	Decrement *A*	DECA
01000	AND *A* and *B*	AND
01010	OR *A* and *B*	OR
01100	XOR *A* and *B*	XOR
01110	Complement *A*	COMA
10000	Shift right *A*	SHRA
11000	Shift left *A*	SHLA

Examples of Microoperations

A control word of 14 bits is needed to specify a microoperation in the CPU. The control word for a given microoperation can be derived from the selection variables. For example, the subtract microoperation given by the statement

$$R1 \leftarrow R2 - R3$$

specifies $R2$ for the A input of the ALU, $R3$ for the B input of the ALU, $R1$ for the destination register, and an ALU operation to subtract $A - B$. Thus the control word is specified by the four fields and the corresponding binary value for each field is obtained from the encoding listed in Tables 8-1 and 8-2. The binary control word for the subtract microoperation is 010 011 001 00101 and is obtained as follows:

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

The control word for this microoperation and a few others are listed in Table 8-3.

The increment and transfer microoperations do not use the B input of the ALU. For these cases, the B field is marked with a dash. We assign 000 to any unused field when formulating the binary control word, although any other binary number may be used. To place the content of a register into the output terminals we place the content of the register into the A input of the ALU, but none of the registers are selected to accept the data. The ALU operation TSFA places the data from the register, through the ALU, into the output terminals. The direct transfer from input to output is accomplished with a control word

TABLE 8-3 Examples of Microoperations for the CPU

	Symbolic Designation				
Microoperation	SELA	SELB	SELD	OPR	Control Word
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	—	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	—	R7	TSFA	001 000 111 00000
Output $\leftarrow R2$	R2	—	None	TSFA	010 000 000 00000
Output $\leftarrow$ Input	Input	—	None	TSFA	000 000 000 00000
$R4 \leftarrow$ shl $R4$	R4	—	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

of all 0's (making the *B* field 000). A register can be cleared to 0 with an exclusive-OR operation. This is because $x \oplus x = 0$.

It is apparent from these examples that many other microoperations can be generated in the CPU. The most efficient way to generate control words with a large number of bits is to store them in a memory unit. A memory unit that stores control words is referred to as a control memory. By reading consecutive control words from memory, it is possible to initiate the desired sequence of microoperations for the CPU. This type of control is referred to as microprogrammed control. A microprogrammed control unit is shown in Fig. 7-8. The binary control word for the CPU will come from the outputs of the control memory marked "micro-ops."

8-3 Stack Organization

LIFO

A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

stack pointer

The stack in digital computers is essentially a memory unit with an address register that can count only (after an initial value is loaded into it). The register that holds the address for the stack is called a stack pointer (*SP*) because its value always points at the top item in the stack. Contrary to a stack of trays where the tray itself may be taken out or inserted, the physical registers of a stack are always available for reading or writing. It is the content of the word that is inserted or deleted.

The two operations of a stack are the insertion and deletion of items. The operation of insertion is called *push* (or push-down) because it can be thought of as the result of pushing a new item on top. The operation of deletion is called *pop* (or pop-up) because it can be thought of as the result of removing one item so that the stack pops up. However, nothing is pushed or popped in a computer stack. These operations are simulated by incrementing or decrementing the stack pointer register.

Register Stack

A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure 8-3 shows the organization of a 64-word register stack. The stack pointer register *SP* contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: *A*, *B*, and *C*, in that order. Item *C* is on top of the stack so that the content of *SP* is now 3. To remove the top item, the stack is popped by reading the memory word

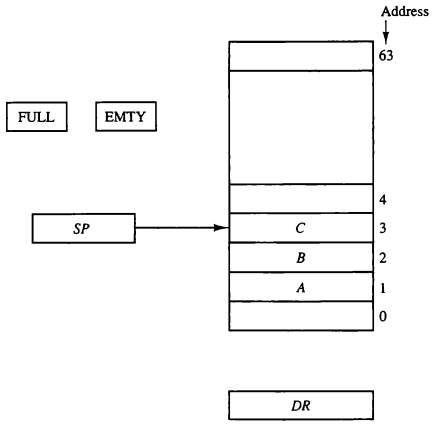


Figure 8-3 Block diagram of a 64-word stack.

at address 3 and decrementing the content of *SP*. Item *B* is now on top of the stack since *SP* holds address 2. To insert a new item, the stack is pushed by incrementing *SP* and writing a word in the next-higher location in the stack. Note that item *C* has been read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place.

In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$. Since *SP* has only six bits, it cannot exceed a number greater than 63 (111111 in binary). When 63 is incremented by 1, the result is 0 since 111111 + 1 = 1000000 in binary, but *SP* can accommodate only the six least significant bits. Similarly, when 000000 is decremented by 1, the result is 111111. The one-bit register *FULL* is set to 1 when the stack is full, and the one-bit register *EMTY* is set to 1 when the stack is empty of items. *DR* is the data register that holds the binary data to be written into or read out of the stack.

Initially, *SP* is cleared to 0, *EMTY* is set to 1, and *FULL* is cleared to 0, so that *SP* points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if *FULL* = 0), a new item is inserted with a push operation. The push operation is implemented with the following sequence of microoperations:

push

$SP \leftarrow SP + 1$	Increment stack pointer
$M[SP] \leftarrow DR$	Write item on top of the stack

If ($SP = 0$) then ($FULL \leftarrow 1$)	Check if stack is full
$EMTY \leftarrow 0$	Mark the stack not empty

The stack pointer is incremented so that it points to the address of the next-higher word. A memory write operation inserts the word from *DR* into the top of the stack. Note that *SP* holds the address of the top of the stack and that *M*[*SP*] denotes the memory word specified by the address presently available in *SP*. The first item stored in the stack is at address 1. The last item is stored at address 0. If *SP* reaches 0, the stack is full of items, so *FULL* is set to 1. This condition is reached if the top item prior to the last push was in location 63 and, after incrementing *SP*, the last item is stored in location 0. Once an item is stored in location 0, there are no more empty registers in the stack. If an item is written in the stack, obviously the stack cannot be empty, so *EMTY* is cleared to 0.

pop

A new item is deleted from the stack if the stack is not empty (if *EMTY* = 0). The pop operation consists of the following sequence of micro-operations:

$DR \leftarrow M[SP]$	Read item from the top of stack
$SP \leftarrow SP - 1$	Decrement stack pointer
If ($SP = 0$) then ($EMTY \leftarrow 1$)	Check if stack is empty
$FULL \leftarrow 0$	Mark the stack not full

The top item is read from the stack into *DR*. The stack pointer is then decremented. If its value reaches zero, the stack is empty, so *EMTY* is set to 1. This condition is reached if the item read was in location 1. Once this item is read out, *SP* is decremented and reaches the value 0, which is the initial value of *SP*. Note that if a pop operation reads the item from location 0 and then *SP* is decremented, *SP* changes to 111111, which is equivalent to decimal 63. In this configuration, the word in address 0 receives the last item in the stack. Note also that an erroneous operation will result if the stack is pushed when *FULL* = 1 or popped when *EMTY* = 1.

Memory Stack

A stack can exist as a stand-alone unit as in Fig. 8-3 or can be implemented in a random-access memory attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. Figure 8-4 shows a portion of computer memory partitioned into three segments: program, data, and stack. The program counter *PC* points at the address of the next instruction in the program. The address register *AR* points at an array of data. The stack pointer

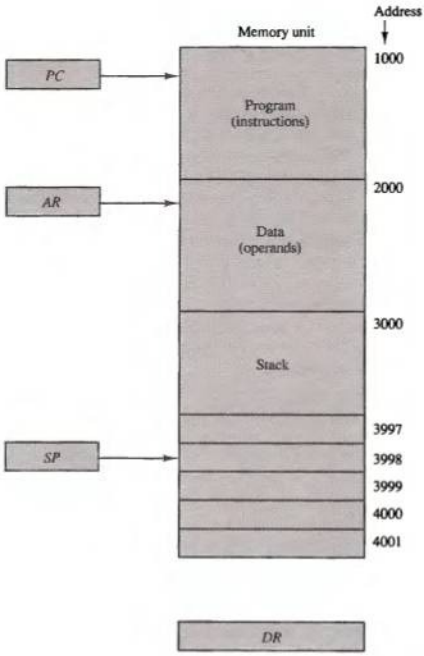


Figure 8-4 Computer memory with program, data, and stack segments.

SP points at the top of the stack. The three registers are connected to a common address bus, and either one can provide an address for memory. *PC* is used during the fetch phase to read an instruction. *AR* is used during the execute phase to read an operand. *SP* is used to push or pop items into or from the stack.

As shown in Fig. 8-4, the initial value of *SP* is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000. No provisions are available for stack limit checks.

We assume that the items in the stack communicate with a data register *DR*. A new item is inserted with the push operation as follows:

$$SP \leftarrow SP - 1$$
$$M[SP] \leftarrow DR$$

The stack pointer is decremented so that it points at the address of the next word. A memory write operation inserts the word from *DR* into the top of the stack. A new item is deleted with a pop operation as follows:

$$DR \leftarrow M[SP]$$
$$SP \leftarrow SP + 1$$

The top item is read from the stack into *DR*. The stack pointer is then incremented to point at the next item in the stack.

stack limits

Most computers do not provide hardware to check for stack overflow (full stack) or underflow (empty stack). The stack limits can be checked by using two processor registers: one to hold the upper limit (3000 in this case), and the other to hold the lower limit (4001 in this case). After a push operation, *SP* is compared with the upper-limit register and after a pop operation, *SP* is compared with the lower-limit register.

The two microoperations needed for either the push or pop are (1) an access to memory through *SP*, and (2) updating *SP*. Which of the two microoperations is done first and whether *SP* is updated by incrementing or decrementing depends on the organization of the stack. In Fig. 8-4 the stack grows by *decreasing* the memory address. The stack may be constructed to grow by *increasing* the memory address as in Fig. 8-3. In such a case, *SP* is incremented for the push operation and decremented for the pop operation. A stack may be constructed so that *SP* points at the next *empty* location above the top of the stack. In this case the sequence of microoperations must be interchanged.

A stack pointer is loaded with an initial value. This initial value must be the bottom address of an assigned stack in memory. Henceforth, *SP* is automatically decremented or incremented with every push or pop operation. The advantage of a memory stack is that the CPU can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

Reverse Polish Notation

A stack organization is very effective for evaluating arithmetic expressions. The common mathematical method of writing arithmetic expressions imposes difficulties when evaluated by a computer. The common arithmetic expressions

are written in *infix notation*, with each operator written *between* the operands. Consider the simple arithmetic expression

$$A * B + C * D$$

The star (denoting multiplication) is placed between two operands A and B or C and D. The plus is between the two products. To evaluate this arithmetic expression it is necessary to compute the product $A * B$, store this product while computing $C * D$, and then sum the two products. From this example we see that to evaluate arithmetic expressions in infix notation it is necessary to scan back and forth along the expression to determine the next operation to be performed.

The Polish mathematician Lukasiewicz showed that arithmetic expressions can be represented in *prefix notation*. This representation, often referred to as *Polish notation*, places the operator before the operands. The *postfix notation*, referred to as *reverse Polish notation* (RPN), places the operator after the operands. The following examples demonstrate the three representations:

RPN

$A + B$	Infix notation
$+AB$	Prefix or Polish notation
$AB+$	Postfix or reverse Polish notation

The reverse Polish notation is in a form suitable for stack manipulation. The expression

$$A * B + C * D$$

is written in reverse Polish notation as

$$AB * CD * +$$

and is evaluated as follows: Scan the expression from left to right. When an operator is reached, perform the operation with the two operands found on the left side of the operator. Remove the two operands and the operator and replace them by the number obtained from the result of the operation. Continue to scan the expression and repeat the procedure for every operator encountered until there are no more operators.

For the expression above we find the operator $*$ after A and B. We perform the operation $A * B$ and replace A, B, and $*$ by the product to obtain

$$(A * B)\,CD * +$$

where $(A * B)$ is a *single* quantity obtained from the product. The next operator

is a * and its previous two operands are C and D, so we perform $C * D$ and obtain an expression with two operands and one operator:

$$(A * B)(C * D) +$$

The next operator is + and the two operands to be added are the two products, so we add the two quantities to obtain the result.

conversion to RPN

The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation. This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations. Consider the expression

$$(A + B) * [C * (D + E) + F]$$

To evaluate the expression we must first perform the arithmetic inside the parentheses $(A + B)$ and $(D + E)$. Next we must calculate the expression inside the square brackets. The multiplication of $C * (D + E)$ must be done prior to the addition of F since multiplication has precedence over addition. The last operation is the multiplication of the two terms between the parentheses and brackets. The expression can be converted to reverse Polish notation, without the use of parentheses, by taking into consideration the operation hierarchy. The converted expression is

$$AB + DE + C * F + *$$

Proceeding from left to right, we first add A and B, then add D and E. At this point we are left with

$$(A + B)(D + E)C * F + *$$

where $(A + B)$ and $(D + E)$ are each a *single* number obtained from the sum. The two operands for the next * are C and $(D+E)$. These two numbers are multiplied and the product added to F. The final * causes the multiplication of the two terms.

Evaluation of Arithmetic Expressions

Reverse Polish notation, combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmetic expressions. This procedure is employed in some electronic calculators and also in some computers. The stack is particularly useful for handling long, complex problems involving chain calculations. It is based on the fact that any arithmetic expression can be expressed in parentheses-free Polish notation.

The procedure consists of first converting the arithmetic expression into its equivalent reverse Polish notation. The operands are pushed into the stack in the order in which they appear. The initiation of an operation depends on whether we have a calculator or a computer. In a calculator, the operators are entered through the keyboard. In a computer, they must be initiated by instructions that contain an operation field (no address field is required). The following microoperations are executed with the stack when an operation is entered in a calculator or issued by the control in a computer: (1) the two topmost operands in the stack are used for the operation, and (2) the stack is popped and the result of the operation replaces the lower operand. By pushing the operands into the stack continuously and performing the operations as defined above, the expression is evaluated in the proper order and the final result remains on top of the stack.

The following numerical example may clarify this procedure. Consider the arithmetic expression

$$(3*4) + (5*6)$$

In reverse Polish notation, it is expressed as

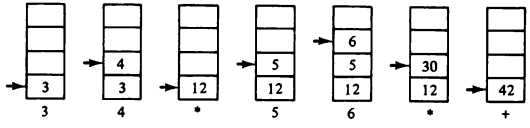
$$34*56*+$$

stack operations

Now consider the stack operations shown in Fig. 8-5. Each box represents one stack operation and the arrow always points to the top of the stack. Scanning the expression from left to right, we encounter two operands. First the number 3 is pushed into the stack, then the number 4. The next symbol is the multiplication operator *. This causes a multiplication of the two topmost items in the stack. The stack is then popped and the product is placed on top of the stack, replacing the two original operands. Next we encounter the two operands 5 and 6, so they are pushed into the stack. The stack operation that results from the next * replaces these two numbers by their product. The last operation causes an arithmetic addition of the two topmost numbers in the stack to produce the final result of 42.

Scientific calculators that employ an internal stack require that the user convert the arithmetic expressions into reverse Polish notation. Computers that use a stack-organized CPU provide a system program to perform the

Figure 8-5 Stack operations to evaluate 3 • 4 + 5 • 6.



conversion for the user. Most compilers, irrespective of their CPU organization, convert all arithmetic expressions into Polish notation anyway because this is the most efficient method for translating arithmetic expressions into machine language instructions. So in essence, a stack-organized CPU may be more efficient in some applications than a CPU without a stack.

8-4 Instruction Formats

The physical and logical structure of computers is normally described in reference manuals provided with the system. Such manuals explain the internal construction of the CPU, including the processor registers available and their logical capabilities. They list all hardware-implemented instructions, specify their binary code format, and provide a precise definition of each instruction. A computer will usually have a variety of instruction code formats. It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction.

The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

1. An operation code field that specifies the operation to be performed.
2. An address field that designates a memory address or a processor register.
3. A mode field that specifies the way the operand or the effective address is determined.

Other special fields are sometimes employed under certain circumstances, as for example a field that gives the number of shifts in a shift-type instruction.

The operation code field of an instruction is a group of bits that define various processor operations, such as add, subtract, complement, and shift. The most common operations available in computer instructions are enumerated and discussed in Sec. 8-6. The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address. The various addressing modes that have been formulated for digital computers are presented in Sec. 8-5. In this section we are concerned with the address field of an instruction format and consider the effect of including multiple address fields in an instruction.

Operations specified by computer instructions are executed on some data stored in memory or processor registers. Operands residing in memory are specified by their memory address. Operands residing in processor registers are specified with a register address. A register address is a binary number of k bits that defines one of 2^k registers in the CPU. Thus a CPU with 16 processor

register address

registers $R0$ through $R15$ will have a register address field of four bits. The binary number 0101, for example, will designate register $R5$.

Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations:

1. Single accumulator organization.
2. General register organization.
3. Stack organization.

An example of an accumulator-type organization is the basic computer presented in Chap. 5. All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field. For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as

```
ADD    X
```

where X is the address of the operand. The ADD instruction in this case results in the operation $AC \leftarrow AC + M[X]$. AC is the accumulator register and $M[X]$ symbolizes the memory word located at address X.

An example of a general register type of organization was presented in Fig. 7-1. The instruction format in this type of computer needs three register address fields. Thus the instruction for an arithmetic addition may be written in an assembly language as

```
ADD    R1, R2, R3
```

to denote the operation $R1 \leftarrow R2 + R3$. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction

```
ADD    R1, R2
```

would denote the operation $R1 \leftarrow R1 + R2$. Only register addresses for $R1$ and $R2$ need be specified in this instruction.

Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction. Thus the instruction

```
MOV    R1, R2
```

denotes the transfer $R1 \leftarrow R2$ (or $R2 \leftarrow R1$, depending on the particular computer). Thus transfer-type instructions need two address fields to specify the source and the destination.

General register-type computers employ two or three address fields in

their instruction format. Each address field may specify a processor register or a memory word. An instruction symbolized by

```
ADD    R1, X
```

would specify the operation $R1 \leftarrow R1 + M[X]$. It has two address fields, one for register $R1$ and the other for the memory address X.

The stack-organized CPU was presented in Fig. 8-4. Computers with stack organization would have PUSH and POP instructions which require an address field. Thus the instruction

```
PUSH   X
```

will push the word at address X to the top of the stack. The stack pointer is updated automatically. Operation-type instructions do not need an address field in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack. The instruction

```
ADD
```

in a stack computer consists of an operation code only with no address field. This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack. There is no need to specify operands with an address field since all operands are implied to be in the stack.

Most computers fall into one of the three types of organizations that have just been described. Some computers combine features from more than one organizational structure. For example, the Intel 8080 microprocessor has seven CPU registers, one of which is an accumulator register. As a consequence, the processor has some of the characteristics of a general register type and some of the characteristics of an accumulator type. All arithmetic and logic instructions, as well as the load and store instructions, use the accumulator register, so these instructions have only one address field. On the other hand, instructions that transfer data among the seven processor registers have a format that contains two register address fields. Moreover, the Intel 8080 processor has a stack pointer and instructions to push and pop from a memory stack. The processor, however, does not have the zero-address-type instructions which are characteristic of a stack-organized CPU.

To illustrate the influence of the number of addresses on computer programs, we will evaluate the arithmetic statement

$$X = (A + B) * (C + D)$$

using zero, one, two, or three address instructions. We will use the symbols ADD, SUB, MUL, and DIV for the four arithmetic operations; MOV for the transfer-type operation; and LOAD and STORE for transfers to and

from memory and *AC* register. We will assume that the operands are in memory addresses *A*, *B*, *C*, and *D*, and the result must be stored in memory at address *X*.

Three-Address Instructions

Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand. The program in assembly language that evaluates $X = (A + B) * (C + D)$ is shown below, together with comments that explain the register transfer operation of each instruction.

```
ADD   R1, A, B     R1 ← M[A] + M[B]
ADD   R2, C, D     R2 ← M[C] + M[D]
MUL   X, R1, R2    M[X] ← R1 * R2
```

It is assumed that the computer has two processor registers, *R*1 and *R*2. The symbol *M*[*A*] denotes the operand at memory address symbolized by *A*.

The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses. An example of a commercial computer that uses three-address instructions is the Cyber 170. The instruction formats in the Cyber computer are restricted to either three register address fields or two register address fields and one memory address field.

Two-Address Instructions

Two-address instructions are the most common in commercial computers. Here again each address field can specify either a processor register or a memory word. The program to evaluate $X = (A + B) * (C + D)$ is as follows:

```
MOV   R1, A     R1 ← M[A]
ADD   R1, B     R1 ← R1 + M[B]
MOV   R2, C     R2 ← M[C]
ADD   R2, D     R2 ← R2 + M[D]
MUL   R1,R2     R1 ← R1 * R2
MOV   X, R1     M[X] ← R1
```

The MOV instruction moves or transfers the operands to and from memory and processor registers. The first symbol listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.

One-Address Instructions

One-address instructions use an implied accumulator (*AC*) register for all data manipulation. For multiplication and division there is a need for a second register. However, here we will neglect the second register and assume that the *AC* contains the result of all operations. The program to evaluate $X = (A + B) * (C + D)$ is

```
LOAD    A    AC ← M[A]
ADD     B    AC ← AC + M[B]
STORE   T    M[T] ← AC
LOAD    C    AC ← M[C]
ADD     D    AC ← AC + M[D]
MUL     T    AC ← AC * M[T]
STORE   X    M[X] ← AC
```

All operations are done between the *AC* register and a memory operand. *T* is the address of a temporary memory location required for storing the intermediate result.

Zero-Address Instructions

A stack-organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The following program shows how $X = (A + B) * (C + D)$ will be written for a stack-organized computer. (*TOS* stands for top of stack.)

```
PUSH    A    TOS ← A
PUSH    B    TOS ← B
ADD          TOS ← (A + B)
PUSH    C    TOS ← C
PUSH    D    TOS ← D
ADD          TOS ← (C + D)
MUL          TOS ← (C + D) * (A + B)
POP     X    M[X] ← TOS
```

To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation. The name "zero-address" is given to this type of computer because of the absence of an address field in the computational instructions.

RISC Instructions

The advantages of a reduced instruction set computer (RISC) architecture are explained in Sec. 8-8. The instruction set of a typical RISC processor is restricted

to the use of load and store instructions when communicating between memory and CPU. All other instructions are executed within the registers of the CPU without referring to memory. A program for a RISC-type CPU consists of LOAD and STORE instructions that have one memory and one register address, and computational-type instructions that have three addresses with all three specifying processor registers. The following is a program to evaluate $X = (A + B) * (C + D)$.

```
LOAD    R1, A        R1 ← M[A]
LOAD    R2, B        R2 ← M[B]
LOAD    R3, C        R3 ← M[C]
LOAD    R4, D        R4 ← M[D]
ADD     R1, R1, R2   R1 ← R1 + R2
ADD     R3, R3, R2   R3 ← R3 + R4
MUL     R1, R1, R3   R1 ← R1*R3
STORE   X, R1        M[X] ← R1
```

The load instructions transfer the operands from memory to CPU registers. The add and multiply operations are executed with data in the registers without accessing memory. The result of the computations is then stored in memory with a store instruction.

8-5 Addressing Modes

The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words. The way the operands are chosen during program execution is dependent on the addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced. Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:

1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
2. To reduce the number of bits in the addressing field of the instruction.

The availability of the addressing modes gives the experienced assembly language programmer flexibility for writing programs that are more efficient with respect to the number of instructions and execution time.

To understand the various addressing modes to be presented in this section, it is imperative that we understand the basic operation cycle of the

computer. The control unit of a computer is designed to go through an instruction cycle that is divided into three major phases:

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Execute the instruction.

program counter (PC)

There is one register in the computer called the program counter or *PC* that keeps track of the instructions in the program stored in memory. *PC* holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory. The decoding done in step 2 determines the operation to be performed, the addressing mode of the instruction, and the location of the operands. The computer then executes the instruction and returns to step 1 to fetch the next instruction in sequence.

In some computers the addressing mode of the instruction is specified with a distinct binary code, just like the operation code is specified. Other computers use a single binary code that designates both the operation and the mode of the instruction. Instructions may be defined with a variety of addressing modes, and sometimes, two or more addressing modes are combined in one instruction.

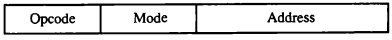
An example of an instruction format with a distinct addressing mode field is shown in Fig. 8-6. The operation code specifies the operation to be performed. The mode field is used to locate the operands needed for the operation. There may or may not be an address field in the instruction. If there is an address field, it may designate a memory address or a processor register. Moreover, as discussed in the preceding section, the instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

mode field

Although most addressing modes modify the address field of the instruction, there are two modes that need no address field at all. These are the implied and immediate modes.

Implied Mode: In this mode the operands are specified implicitly in the definition of the instruction. For example, the instruction "complement accumulator" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, all register reference instructions that use an accumulator are implied-mode instructions.

Figure 8-6 Instruction format with mode field.



Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

Immediate Mode: In this mode the operand is specified in the instruction itself. In other words, an immediate-mode instruction has an operand field rather than an address field. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction. Immediate-mode instructions are useful for initializing registers to a constant value.

It was mentioned previously that the address field of an instruction may specify either a memory word or a processor register. When the address field specifies a processor register, the instruction is said to be in the register mode.

Register Mode: In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction. A k-bit field can specify any one of 2^k registers.

Register Indirect Mode: In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself. Before using a register indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction. A reference to the register is then equivalent to specifying a memory address. The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

Autoincrement or Autodecrement Mode: This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory. When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table. This can be achieved by using the increment or decrement instruction. However, because it is such a common requirement, some computers incorporate a special mode that automatically increments or decrements the content of the register after data access.

The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory. Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated. To differentiate among the various addressing modes it is necessary to distinguish between the address part of the instruction and the effective address used by the control when executing the instruction. The *effective address* is defined to be the memory address obtained from the computation dictated by the given addressing mode. The effective address is the address of the operand in a computational-

effective address

type instruction. It is the address where control branches in response to a branch-type instruction. We have already defined two addressing modes in Chap. 5. They are summarized here for reference.

Direct Address Mode: In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction the address field specifies the actual branch address.

Indirect Address Mode: In this mode the address field of the instruction gives the address where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address. The indirect address mode is also explained in Sec. 5-1 in conjunction with Fig. 5-2.

A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU. The effective address in these modes is obtained from the following computation:

effective address = address part of instruction + content of CPU register

The CPU register used in the computation may be the program counter, an index register, or a base register. In either case we have a different addressing mode which is used for a different application.

Relative Address Mode: In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address. The address part of the instruction is usually a signed number (in 2's complement representation) which can be either positive or negative. When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction. To clarify with an example, assume that the program counter contains the number 825 and the address part of the instruction contains the number 24. The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to 826. The effective address computation for the relative address mode is 826 + 24 = 850. This is 24 memory locations forward from the address of the next instruction. Relative addressing is often used with branch-type instructions when the branch address is in the area surrounding the instruction word itself. It results in a shorter address field in the instruction format since the relative address can be specified with a smaller number of bits compared to the number of bits required to designate the entire memory address.

Indexed Addressing Mode: In this mode the content of an index register is added to the address part of the instruction to obtain the effective address. The

index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the index register. Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands. Note that if an index-type instruction does not include an address field in its format, the instruction converts to the register indirect mode of operation.

Some computers dedicate one CPU register to function solely as an index register. This register is involved implicitly when the index-mode instruction is used. In computers with many processor registers, any one of the CPU registers can contain the index number. In such a case the register must be specified explicitly in a register field within the instruction format.

Base Register Addressing Mode: In this mode the content of a base register is added to the address part of the instruction to obtain the effective address. This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register. The difference between the two modes is in the way they are used rather than in the way that they are computed. An index register is assumed to hold an index number that is relative to the address part of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address. The base register addressing mode is used in computers to facilitate the relocation of programs in memory. When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of instructions must reflect this change of position. With a base register, the displacement values of instructions do not have to change. Only the value of the base register requires updating to reflect the beginning of a new memory segment.

Numerical Example

To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Fig. 8-7. The two-word instruction at address 200 and 201 is a "load to *AC*" instruction with an address field equal to 500. The first word of the instruction specifies the operation code and mode, and the second word specifies the address part. *PC* has the value 200 for fetching this instruction. The content of processor register *R*1 is 400, and the content of an index register *XR* is 100. *AC* receives the operand after the instruction is executed. The figure lists a few pertinent addresses and shows the memory content at each of these addresses.

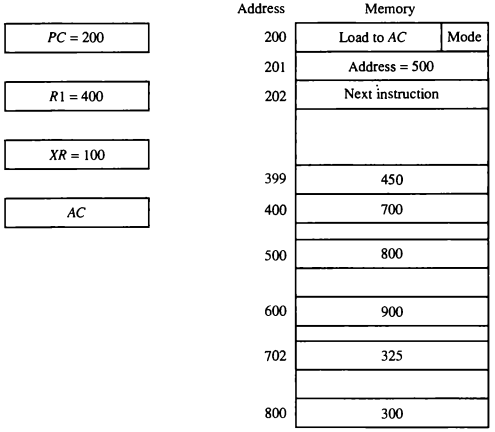


Figure 8-7 Numerical example for addressing modes.

The mode field of the instruction can specify any one of a number of modes. For each possible mode we calculate the effective address and the operand that must be loaded into *AC*. In the direct address mode the effective address is the address part of the instruction 500 and the operand to be loaded into *AC* is 800. In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into *AC*. (The effective address in this case is 201.) In the indirect mode the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300. In the relative mode the effective address is 500 + 202 = 702 and the operand is 325. (Note that the value in *PC* after the fetch phase and during the execute phase is 202.) In the index mode the effective address is *XR* + 500 = 100 + 500 = 600 and the operand is 900. In the register mode the operand is in *R*1 and 400 is loaded into *AC*. (There is no effective address in this case.) In the register indirect mode the effective address is 400, equal to the content of *R*1 and the operand loaded into *AC* is 700. The autoincrement mode is the same as the register indirect mode except that *R*1 is incremented to 401 after the execution of the instruction. The autodecrement mode decrements *R*1 to 399 prior to the execution of the instruction. The operand loaded into *AC* is now 450. Table 8-4 lists the values of the effective address and the operand loaded into *AC* for the nine addressing modes.

TABLE 8-4 Tabular List of Numerical Example

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

8-6 Data Transfer and Manipulation

Computers provide an extensive set of instructions to give the user the flexibility to carry out various computational tasks. The instruction set of different computers differ from each other mostly in the way the operands are determined from the address and mode fields. The actual operations available in the instruction set are not very different from one computer to another. It so happens that the binary code assignments in the operation code field is different in different computers, even for the same operation. It may also happen that the symbolic name given to instructions in the assembly language notation is different in different computers, even for the same instruction. Nevertheless, there is a set of basic operations that most, if not all, computers include in their instruction repertoire. The basic set of operations available in a typical computer is the subject covered in this and the next section.

set of basic operations

Most computer instructions can be classified into three categories:

1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

Data transfer instructions cause transfer of data from one location to another without changing the binary information content. Data manipulation instructions are those that perform arithmetic, logic, and shift operations. Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer. The instruction set of a particular computer determines the register transfer operations and control decisions that are available to the user.

Data Transfer Instructions

Data transfer instructions move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves. Table 8-5 gives a list of eight data transfer instructions used in many computers. Accompanying each instruction is a mnemonic symbol. It must be realized that different computers use different mnemonics for the same instruction name.

The *load* instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator. The *store* instruction designates a transfer from a processor register into memory. The *move* instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers between CPU registers and memory or between two memory words. The *exchange* instruction swaps information between two registers or a register and a memory word. The *input* and *output* instructions transfer data among processor registers and input or output terminals. The *push* and *pop* instructions transfer data between processor registers and a memory stack.

It must be realized that the instructions listed in Table 8-5, as well as in subsequent tables in this section, are often associated with a variety of addressing modes. Some assembly language conventions modify the mnemonic symbol to differentiate between the different addressing modes. For example, the mnemonic for *load immediate* becomes LDI. Other assembly language conventions use a special character to designate the addressing mode. For example, the immediate mode is recognized from a pound sign # placed before the operand. In any case, the important thing is to realize that each instruction can occur with a variety of addressing modes. As an example, consider the *load to accumulator* instruction when used with eight different addressing modes.

TABLE 8-5 Typical Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

TABLE 8-6 Eight Addressing Modes for the Load Instruction

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1], R1 \leftarrow R1 + 1$

Table 8-6 shows the recommended assembly language convention and the actual transfer accomplished in each case. *ADR* stands for an address, *NBR* is a number or operand, *X* is an index register, *R*1 is a processor register, and *AC* is the accumulator register. The @ character symbolizes an indirect address. The $ character before an address makes the address relative to the program counter *PC*. The # character precedes the operand in an immediate-mode instruction. An indexed mode instruction is recognized by a register that is placed in parentheses after the symbolic address. The register mode is symbolized by giving the name of a processor register. In the register indirect mode, the name of the register that holds the memory address is enclosed in parentheses. The autoincrement mode is distinguished from the register indirect mode by placing a plus after the parenthesized register. The autodecrement mode would use a minus instead. To be able to write assembly language programs for a computer, it is necessary to know the type of instructions available and also to be familiar with the addressing modes used in the particular computer.

Data Manipulation Instructions

Data manipulation instructions perform operations on data and provide the computational capabilities for the computer. The data manipulation instructions in a typical computer are usually divided into three basic types:

1. Arithmetic instructions
2. Logical and bit manipulation instructions
3. Shift instructions

A list of data manipulation instructions will look very much like the list of microoperations given in Chap. 4. It must be realized, however, that each instruction when executed in the computer must go through the fetch phase

to read its binary code value from memory. The operands must also be brought into processor registers according to the rules of the instruction addressing mode. The last step is to execute the instruction in the processor. This last step is implemented by means of microoperations as explained in Chap. 4 or through an ALU and shifter as shown in Fig. 8-2. Some of the arithmetic instructions need special circuits for their implementation.

Arithmetic Instructions

The four basic arithmetic operations are addition, subtraction, multiplication, and division. Most computers provide instructions for all four operations. Some small computers have only addition and possibly subtraction instructions. The multiplication and division must then be generated by means of software subroutines. The four basic arithmetic operations are sufficient for formulating solutions to scientific problems when expressed in terms of numerical analysis methods.

A list of typical arithmetic instructions is given in Table 8-7. The increment instruction adds 1 to the value stored in a register or memory word. One common characteristic of the increment operations when executed in processor registers is that a binary number of all 1's when incremented produces a result of all 0's. The decrement instruction subtracts 1 from a value stored in a register or memory word. A number with all 0's, when decremented, produces a number with all 1's.

data type

The add, subtract, multiply, and divide instructions may be available for different types of data. The data type assumed to be in processor registers during the execution of these arithmetic operations is included in the definition of the operation code. An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-precision data. The various data types are presented in Chap. 3.

It is not uncommon to find computers with three or more add instruc-

TABLE 8-7 Typical Arithmetic Instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

tions: one for binary integers, one for floating-point operands, and one for decimal operands. The mnemonics for three add instructions that specify different data types are shown below.

```
ADDI    Add two binary integer numbers
ADDF    Add two floating-point numbers
ADDD    Add two decimal numbers in BCD
```

Algorithms for integer, floating-point, and decimal arithmetic operations are developed in Chap. 10.

The number of bits in any register is of finite length and therefore the results of arithmetic operations are of finite precision. Some computers provide hardware double-precision operations where the length of each operand is taken to be the length of two memory words. Most small computers provide special instructions to facilitate double-precision arithmetic. A special carry flip-flop is used to store the carry from an operation. The instruction "add with carry" performs the addition on two operands plus the value of the carry from the previous computation. Similarly, the "subtract with borrow" instruction subtracts two words and a borrow which may have resulted from a previous subtract operation. The negate instruction forms the 2's complement of a number, effectively reversing the sign of an integer when represented in the signed-2's complement form.

Logical and Bit Manipulation Instructions

Logical instructions perform binary operations on strings of bits stored in registers. They are useful for manipulating individual bits or a group of bits that represent binary-coded information. The logical instructions consider each bit of the operand separately and treat it as a Boolean variable. By proper application of the logical instructions it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in registers or memory words.

Some typical logical and bit manipulation instructions are listed in Table 8-8. The clear instruction causes the specified operand to be replaced by 0's. The complement instruction produces the 1's complement by inverting all the bits of the operand. The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operands. Although they perform Boolean operations, when used in computer instructions, the logical instructions should be considered as performing bit manipulation operations. There are three bit manipulation operations possible: a selected bit can be cleared to 0, or can be set to 1, or can be complemented. The three logical instructions are usually applied to do just that.

clear selected bits

The AND instruction is used to clear a bit or a selected group of bits of an operand. For any Boolean variable x, the relationships $x \cdot 0 = 0$ and $x \cdot 1 = x$ dictate that a binary variable ANDed with a 0 produces a 0; but the variable

TABLE 8-8 Typical Logical and Bit Manipulation Instructions

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

does not change in value when ANDed with a 1. Therefore, the AND instruction can be used to clear bits of an operand selectively by ANDing the operand with another operand that has 0's in the bit positions that must be cleared. The AND instruction is also called a *mask* because it masks or inserts 0's in a selected portion of an operand.

set selected bits

The OR instruction is used to set a bit or a selected group of bits of an operand. For any Boolean variable x, the relationships $x + 1 = 1$ and $x + 0 = x$ dictate that a binary variable ORed with a 1 produces a 1; but the variable does not change when ORed with a 0. Therefore, the OR instruction can be used to selectively set bits of an operand by ORing it with another operand with 1's in the bit positions that must be set to 1.

complement selected bits

Similarly, the XOR instruction is used to selectively complement bits of an operand. This is because of the Boolean relationships $x \oplus 1 = x'$ and $x \oplus 0 = x$. Thus a binary variable is complemented when XORed with a 1 but does not change in value when XORed with a 0. Numerical examples showing the three logic operations are given in Sec. 4-5.

A few other bit manipulation instructions are included in Table 8-8. Individual bits such as a carry can be cleared, set, or complemented with appropriate instructions. Another example is a flip-flop that controls the interrupt facility and is either enabled or disabled by means of bit manipulation instructions.

Shift Instructions

Instructions to shift the content of an operand are quite useful and are often provided in several variations. Shifts are operations in which the bits of a word are moved to the left or right. The bit shifted in at the end of the word determines the type of shift used. Shift instructions may specify either logical

shifts, arithmetic shifts, or rotate-type operations. In either case the shift may be to the right or to the left.

Table 8-9 lists four types of shift instructions. The logical shift inserts 0 to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left. Arithmetic shifts usually conform with the rules for signed-2's complement numbers. These rules are given in Sec. 4-6. The arithmetic shift-right instruction must preserve the sign bit in the leftmost position. The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged. This is a shift-right operation with the end bit remaining the same. The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-left instruction. For this reason many computers do not provide a distinct arithmetic shift-left instruction when the logical shift-left instruction is already available.

The rotate instructions produce a circular shift. Bits shifted out at one end of the word are not lost as in a logical shift but are circulated back into the other end. The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated. Thus a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shifts the entire register to the left.

Some computers have a multiple-field format for the shift instructions. One field contains the operation code and the others specify the type of shift and the number of times that an operand is to be shifted. A possible instruction code format of a shift instruction may include five fields as follows:

```
OP    REG    TYPE    RL    COUNT
```

Here OP is the operation code field; REG is a register address that specifies the location of the operand; TYPE is a 2-bit field specifying the four different types of shifts; RL is a 1-bit field specifying a shift right or left; and COUNT is a k-bit field specifying up to $2^k - 1$ shifts. With such a format, it is possible to specify the type of shift, the direction, and the number of shifts, all in one instruction.

TABLE 8-9 Typical Shift Instructions

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

8-7 Program Control

Instructions are always stored in successive memory locations. When processed in the CPU, the instructions are fetched from consecutive memory locations and executed. Each time an instruction is fetched from memory, the program counter is incremented so that it contains the address of the next instruction in sequence. After the execution of a data transfer or data manipulation instruction, control returns to the fetch cycle with the program counter containing the address of the instruction next in sequence. On the other hand, a program control type of instruction, when executed, may change the address value in the program counter and cause the flow of control to be altered. In other words, program control instructions specify conditions for altering the content of the program counter, while data transfer and manipulation instructions specify conditions for data-processing operations. The change in value of the program counter as a result of the execution of a program control instruction causes a break in the sequence of instruction execution. This is an important feature in digital computers, as it provides control over the flow of program execution and a capability for branching to different program segments.

Some typical program control instructions are listed in Table 8-10. The branch and jump instructions are used interchangeably to mean the same thing, but sometimes they are used to denote different addressing modes. The branch is usually a one-address instruction. It is written in assembly language as BR ADR, where ADR is a symbolic name for an address. When executed, the branch instruction causes a transfer of the value of ADR into the program counter. Since the program counter contains the address of the instruction to be executed, the next instruction will come from location ADR.

Branch and jump instructions may be conditional or unconditional. An unconditional branch instruction causes a branch to the specified address without any conditions. The conditional branch instruction specifies a condition such as branch if positive or branch if zero. If the condition is met, the program counter is loaded with the branch address and the next instruction is taken

TABLE 8-10 Typical Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

from this address. If the condition is not met, the program counter is not changed and the next instruction is taken from the next location in sequence.

The skip instruction does not need an address field and is therefore a zero-address instruction. A conditional skip instruction will skip the next instruction if the condition is met. This is accomplished by incrementing the program counter during the execute phase in addition to its being incremented during the fetch phase. If the condition is not met, control proceeds with the next instruction in sequence where the programmer inserts an unconditional branch instruction. Thus a skip-branch pair of instructions causes a branch if the condition is not met, while a single conditional branch instruction causes a branch if the condition is met.

The call and return instructions are used in conjunction with subroutines. Their performance and implementation are discussed later in this section. The compare and test instructions do not change the program sequence directly. They are listed in Table 8-10 because of their application in setting conditions for subsequent conditional branch instructions. The compare instruction performs a subtraction between two operands, but the result of the operation is not retained. However, certain status bit conditions are set as a result of the operation. Similarly, the test instruction performs the logical AND of two operands and updates certain status bits without retaining the result or changing the operands. The status bits of interest are the carry bit, the sign bit, a zero indication, and an overflow condition. The generation of these status bits will be discussed first and then we will show how they are used in conditional branch instructions.

Status Bit Conditions

It is sometimes convenient to supplement the ALU circuit in the CPU with a status register where status bit conditions can be stored for further analysis. Status bits are also called *condition-code* bits or *flag* bits. Figure 8-8 shows the block diagram of an 8-bit ALU with a 4-bit status register. The four status bits are symbolized by C, S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.

1. Bit C (carry) is set to 1 if the end carry C_8 is 1. It is cleared to 0 if the carry is 0.
2. Bit S (sign) is set to 1 if the highest-order bit F_7 is 1. It is set to 0 if the bit is 0.
3. Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. It is cleared to 0 otherwise. In other words, $Z = 1$ if the output is zero and $Z = 0$ if the output is not zero.
4. Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries is equal to 1, and cleared to 0 otherwise. This is the condition for an

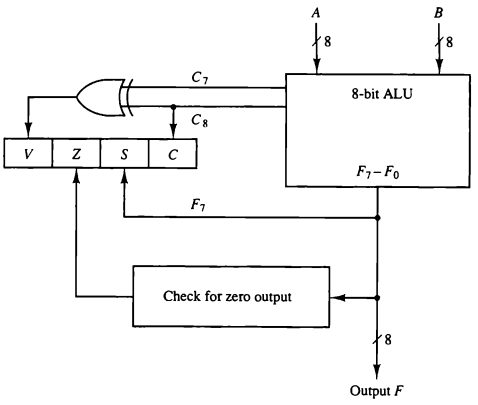


Figure 8-8 Status register bits.

overflow when negative numbers are in 2's complement (see Sec. 3-3). For the 8-bit ALU, $V = 1$ if the output is greater than +127 or less than −128.

The status bits can be checked after an ALU operation to determine certain relationships that exist between the values of A and B. If bit V is set after the addition of two signed numbers, it indicates an overflow condition. If Z is set after an exclusive-OR operation, it indicates that $A = B$. This is so because $x \oplus x = 0$, and the exclusive-OR of two equal operands gives an all-0's result which sets the Z bit. A single bit in A can be checked to determine if it is 0 or 1 by masking all bits except the bit in question and then checking the Z status bit. For example, let $A = 101x1100$, where x is the bit to be checked. The AND operation of A with $B = 00010000$ produces a result $000x0000$. If $x = 0$, the Z status bit is set, but if $x = 1$, the Z bit is cleared since the result is not zero. The AND operation can be generated with the TEST instruction listed in Table 8-10 if the original content of A must be preserved.

Conditional Branch Instructions

Table 8-11 gives a list of the most common branch instructions. Each mnemonic is constructed with the letter B (for branch) and an abbreviation of the condition name. When the opposite condition state is used, the letter N (for no) is

TABLE 8-11 Conditional Branch Instructions

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
	Unsigned compare conditions ($A - B$)	
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
	Signed compare conditions ($A - B$)	
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

inserted to define the 0 state. Thus BC is Branch on Carry, and BNC is Branch on No Carry. If the stated condition is true, program control is transferred to the address specified by the instruction. If not, control continues with the instruction that follows. The conditional instructions can be associated also with the jump, skip, call, or return type of program control instructions.

The zero status bit is used for testing if the result of an ALU operation is equal to zero or not. The carry bit is used to check if there is a carry out of the most significant bit position of the ALU. It is also used in conjunction with the rotate instructions to check the bit shifted from the end position of a register into the carry position. The sign bit reflects the state of the most significant bit of the output from the ALU. $S = 0$ denotes a positive sign and $S = 1$, a negative sign. Therefore, a branch on plus checks for a sign bit of 0 and a branch on minus checks for a sign bit of 1. It must be realized, however, that these two conditional branch instructions can be used to check the value of the most significant bit whether it represents a sign or not. The overflow bit is used in conjunction with arithmetic operations done on signed numbers in 2's complement representation.

As stated previously, the compare instruction performs a subtraction of two operands, say $A - B$. The result of the operation is not transferred into a destination register, but the status bits are affected. The status register provides information about the relative magnitude of A and B. Some computers provide conditional branch instructions that can be applied right after the execution of a compare instruction. The specific conditions to be tested depend on whether the two numbers A and B are considered to be unsigned or signed numbers. Table 8-11 gives a list of such conditional branch instructions. Note that we use the words higher and lower to denote the relations between unsigned numbers, and greater and less than for signed numbers. The relative magnitude shown under the tested condition column in the table seems to be the same for unsigned and signed numbers. However, this is not the case since each must be considered separately as explained in the following numerical example.

numerical example

Consider an 8-bit ALU as shown in Fig. 8-8. The largest unsigned number that can be accommodated in 8 bits is 255. The range of signed numbers is between +127 and −128. The subtraction of two numbers is the same whether they are unsigned or in signed-2's complement representation (see Chap. 3). Let $A = 11110000$ and $B = 00010100$. To perform $A - B$, the ALU takes the 2's complement of B and adds it to A.

$$\begin{array}{rll} A: & 11110000 & \\ \overline{B} + 1: & +11101100 & \\ A - B: & 11011100 & C = 1 \quad S = 1 \quad V = 0 \quad Z = 0 \end{array}$$

The compare instruction updates the status bits as shown. $C = 1$ because there is a carry out of the last stage. $S = 1$ because the leftmost bit is 1. $V = 0$ because the last two carries are both equal to 1, and $Z = 0$ because the result is not equal to 0.

If we assume unsigned numbers, the decimal equivalent of A is 240 and that of B is 20. The subtraction in decimal is $240 - 20 = 220$. The binary result 11011100 is indeed the equivalent of decimal 220. Since $240 > 20$, we have that $A > B$ and $A \neq B$. These two relations can also be derived from the fact that status bit C is equal to 1 and bit Z is equal to 0. The instructions that will cause a branch after this comparison are BHI (branch if higher), BHE (branch if higher or equal), and BNE (branch if not equal).

If we assume signed numbers, the decimal equivalent of A is −16. This is because the sign of A is negative and 11110000 is the 2's complement of 00010000, which is the decimal equivalent of +16. The decimal equivalent of B is +20. The subtraction in decimal is $(-16) - (+20) = -36$. The binary result 11011100 (the 2's complement of 00100100) is indeed the equivalent of decimal −36. Since $(-16) < (+20)$ we have that $A < B$ and $A \neq B$. These two relations can also be derived from the fact that status bits $S = 1$ (negative), $V = 0$ (no overflow), and $Z = 0$ (not zero). The instructions that will cause a branch after this comparison are BLT (branch if less than), BLE (branch if less or equal), and BNE (branch if not equal).

It should be noted that the instruction BNE and BNZ (branch if not zero) are identical. Similarly, the two instructions BE (branch if equal) and BZ (branch if zero) are also identical. Each is repeated three times in Table 8-11 for the purpose of clarity and completeness.

It should be obvious from the example that the relative magnitude of two unsigned numbers can be determined (after a compare instruction) from the values of status bits C and Z (see Prob. 8-26). The relative magnitude of two signed numbers can be determined from the values of S, V, and Z (see Prob. 8-27).

Some computers consider the C bit to be a borrow bit after a subtraction operation $A - B$. A borrow does not occur if $A \geq B$, but a bit must be borrowed from the next most significant position if $A < B$. The condition for a borrow is the complement of the carry obtained when the subtraction is done by taking the 2's complement of B. For this reason, a processor that considers the C bit to be a borrow after a subtraction will complement the C bit after adding the 2's complement of the subtrahend and denote this bit a borrow.

Subroutine Call and Return

A subroutine is a self-contained sequence of instructions that performs a given computational task. During the execution of a program, a subroutine may be called to perform its function many times at various points in the main program. Each time a subroutine is called, a branch is executed to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is made back to the main program.

The instruction that transfers program control to a subroutine is known by different names. The most common names used are *call subroutine*, *jump to subroutine*, *branch to subroutine*, or *branch and save address*. A call subroutine instruction consists of an operation code together with an address that specifies the beginning of the subroutine. The instruction is executed by performing two operations: (1) the address of the next instruction available in the program counter (the return address) is stored in a temporary location so the subroutine knows where to return, and (2) control is transferred to the beginning of the subroutine. The last instruction of every subroutine, commonly called *return from subroutine*, transfers the return address from the temporary location into the program counter. This results in a transfer of program control to the instruction whose address was originally stored in the temporary location.

Different computers use a different temporary location for storing the return address. Some store the return address in the first memory location of the subroutine, some store it in a fixed location in memory, some store it in a processor register, and some store it in a memory stack. The most efficient way is to store the return address in a memory stack. The advantage of using a stack for the return address is that when a succession of subroutines is called, the sequential return addresses can be pushed into the stack. The return from

subroutine instruction causes the stack to pop and the contents of the top of the stack are transferred to the program counter. In this way, the return is always to the program that last called a subroutine. A subroutine call is implemented with the following microoperations:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Push content of PC onto the stack
$PC \leftarrow$ effective address	Transfer control to the subroutine

If another subroutine is called by the current subroutine, the new return address is pushed into the stack, and so on. The instruction that returns from the last subroutine is implemented by the microoperations:

$PC \leftarrow M[SP]$	Pop stack and transfer to PC
$SP \leftarrow SP + 1$	Increment stack pointer

By using a subroutine stack, all return addresses are automatically stored by the hardware in one unit. The programmer does not have to be concerned or remember where the return address was stored.

A *recursive subroutine* is a subroutine that calls itself. If only one register or memory location is used to store the return address, and the recursive subroutine calls itself, it destroys the previous return address. This is undesirable because vital information is destroyed. This problem can be solved if different storage locations are employed for each use of the subroutine while another lighter-level use is still active. When a stack is used, each return address can be pushed into the stack without destroying any previous values. This solves the problem of recursive subroutines because the next subroutine to exit is always the last subroutine that was called.

Program Interrupt

The concept of program interrupt is used to handle a variety of problems that arise out of normal program sequence. Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request. Control returns to the original program after the service program is executed.

The interrupt procedure is, in principle, quite similar to a subroutine call except for three variations: (1) The interrupt is usually initiated by an internal or external signal rather than from the execution of an instruction (except for software interrupt as explained later); (2) the address of the interrupt service program is determined by the hardware rather than from the address field of an instruction; and (3) an interrupt procedure usually stores all the information

necessary to define the state of the CPU rather than storing only the program counter. These three procedural concepts are clarified further below.

After a program has been interrupted and the service routine been executed, the CPU must return to exactly the same state that it was when the interrupt occurred. Only if this happens will the interrupted program be able to resume exactly as if nothing had happened. The state of the CPU at the end of the execute cycle (when the interrupt is recognized) is determined from:

1. The content of the program counter
2. The content of all processor registers
3. The content of certain status conditions

program status word

The collection of all status bit conditions in the CPU is sometimes called a *program status word* or PSW. The PSW is stored in a separate hardware register and contains the status information that characterizes the state of the CPU. Typically, it includes the status bits from the last ALU operation and it specifies the interrupts that are allowed to occur and whether the CPU is operating in a supervisor or user mode. Many computers have a resident operating system that controls and supervises all other programs in the computer. When the CPU is executing a program that is part of the operating system, it is said to be in the supervisor or system mode. Certain instructions are privileged and can be executed in this mode only. The CPU is normally in the user mode when executing user programs. The mode that the CPU is operating at any given time is determined from special status bits in the PSW.

supervisor mode

Some computers store only the program counter when responding to an interrupt. The service program must then include instructions to store status and register content before these resources are used. Only a few computers store both program counter and all status and register content in response to an interrupt. Most computers just store the program counter and the PSW. In some cases, there exist two sets of processor registers within the computer, one for each CPU mode. In this way, when the program switches from the user to the supervisor mode (or vice versa) in response to an interrupt, it is not necessary to store the contents of processor registers as each mode uses its own set of registers.

The hardware procedure for processing an interrupt is very similar to the execution of a subroutine call instruction. The state of the CPU is pushed into a memory stack and the beginning address of the service routine is transferred to the program counter. The beginning address of the service routine is determined by the hardware rather than the address field of an instruction. Some computers assign one memory location where interrupts are always transferred. The service routine must then determine what caused the interrupt and proceed to service it. Some computers assign a memory location for each possible interrupt. Sometimes, the hardware interrupt provides its own address that directs the CPU to the desired service routine. In any case, the CPU

must possess some form of hardware procedure for selecting a branch address for servicing the interrupt.

The CPU does not respond to an interrupt until the end of an instruction execution. Just before going to the next fetch phase, control checks for any interrupt signals. If an interrupt is pending, control goes to a hardware interrupt cycle. During this cycle, the contents of PC and PSW are pushed onto the stack. The branch address for the particular interrupt is then transferred to PC and a new PSW is loaded into the status register. The service program can now be executed starting from the branch address and having a CPU mode as specified in the new PSW.

The last instruction in the service program is a *return from interrupt* instruction. When this instruction is executed, the stack is popped to retrieve the old PSW and the return address. The PSW is transferred to the status register and the return address to the program counter. Thus the CPU state is restored and the original program can continue executing.

Types of Interrupts

There are three major types of interrupts that cause a break in the normal execution of a program. They can be classified as:

1. External interrupts
2. Internal interrupts
3. Software interrupts

External interrupts come from input–output (I/O) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source. Examples that cause external interrupts are I/O device requesting transfer of data, I/O device finished transfer of data, elapsed time of an event, or power failure. Timeout interrupt may result from a program that is in an endless loop and thus exceeded its time allocation. Power failure interrupt may have as its service routine a program that transfers the complete state of the CPU into a nondestructive memory in the few milliseconds before power ceases.

Internal interrupts arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called *traps*. Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation. These error conditions usually occur as a result of a premature termination of the instruction execution. The service program that processes the internal interrupt determines the corrective measure to be taken.

The difference between internal and external interrupts is that the internal interrupt is initiated by some exceptional condition caused by the program itself rather than by an external event. Internal interrupts are synchronous with

the program while external interrupts are asynchronous. If the program is rerun, the internal interrupts will occur in the same place each time. External interrupts depend on external conditions that are independent of the program being executed at the time.

software interrupt

External and internal interrupts are initiated from signals that occur in the hardware of the CPU. A software interrupt is initiated by executing an instruction. Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program. The most common use of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a CPU user mode to the supervisor mode. Certain operations in the computer may be assigned to the supervisor mode only, as for example, a complex input or output transfer procedure. A program written by a user must run in the user mode. When an input or output transfer is required, the supervisor mode is requested by means of a supervisor call instruction. This instruction causes a software interrupt that stores the old CPU state and brings in a new PSW that belongs to the supervisor mode. The calling program must pass information to the operating system in order to specify the particular task requested.

8-8 Reduced Instruction Set Computer (RISC)

An important aspect of computer architecture is the design of the instruction set for the processor. The instruction set chosen for a particular computer determines the way that machine language programs are constructed. Early computers had small and simple instruction sets, forced mainly by the need to minimize the hardware used to implement them. As digital hardware became cheaper with the advent of integrated circuits, computer instructions tended to increase both in number and complexity. Many computers have instruction sets that include more than 100 and sometimes even more than 200 instructions. These computers also employ a variety of data types and a large number of addressing modes. The trend into computer hardware complexity was influenced by various factors, such as upgrading existing models to provide more customer applications, adding instructions that facilitate the translation from high-level language into machine language programs, and striving to develop machines that move functions from software implementation into hardware implementation. A computer with a large number of instructions is classified as a *complex instruction set computer*, abbreviated CISC.

CISC

RISC

In the early 1980s, a number of computer designers recommended that computers use fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often. This type of computer is classified as a *reduced instruction set computer* or RISC. In

this section we introduce the major characteristics of CISC and RISC architectures and then present the instruction set and instruction format of a RISC processor.

CISC Characteristics

The design of an instruction set for a computer must take into consideration not only machine language constructs, but also the requirements imposed on the use of high-level programming languages. The translation from high-level to machine language programs is done by means of a compiler program. One reason for the trend to provide a complex instruction set is the desire to simplify the compilation and improve the overall computer performance. The task of a compiler is to generate a sequence of machine instructions for each high-level language statement. The task is simplified if there are machine instructions that implement the statements directly. The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement that is written in a high-level language. Examples of CISC architectures are the Digital Equipment Corporation VAX computer and the IBM 370 computer.

Another characteristic of CISC architecture is the incorporation of variable-length instruction formats. Instructions that require register operands may be only two bytes in length, but instructions that need two memory addresses may need five bytes to include the entire instruction code. If the computer has 32-bit words (four bytes), the first instruction occupies half a word, while the second instruction needs one word in addition to one byte in the next word. Packing variable instruction formats in a fixed-length memory word requires special decoding circuits that count bytes within words and frame the instructions according to their byte length.

The instructions in a typical CISC processor provide direct manipulation of operands residing in memory. For example, an ADD instruction may specify one operand in memory through index addressing and a second operand in memory through a direct addressing. Another memory location may be included in the instruction to store the sum. This requires three memory references during execution of the instruction. Although CISC processors have instructions that use only processor registers, the availability of other modes of operations tend to simplify high-level language compilation. However, as more instructions and addressing modes are incorporated into a computer, the more hardware logic is needed to implement and support them, and this may cause the computations to slow down. In summary, the major characteristics of CISC architecture are:

1. A large number of instructions—typically from 100 to 250 instructions
2. Some instructions that perform specialized tasks and are used infrequently

3. A large variety of addressing modes—typically from 5 to 20 different modes
4. Variable-length instruction formats
5. Instructions that manipulate operands in memory

RISC Characteristics

The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer. The major characteristics of a RISC processor are:

1. Relatively few instructions
2. Relatively few addressing modes
3. Memory access limited to load and store instructions
4. All operations done within the registers of the CPU
5. Fixed-length, easily decoded instruction format
6. Single-cycle instruction execution
7. Hardwired rather than microprogrammed control

The small set of instructions of a typical RISC processor consists mostly of register-to-register operations, with only simple load and store operations for memory access. Thus each operand is brought into a processor register with a load instruction. All computations are done among the data stored in processor registers. Results are transferred to memory by means of store instructions. This architectural feature simplifies the instruction set and encourages the optimization of register manipulation. The use of only a few addressing modes results from the fact that almost all instructions have simple register addressing. Other addressing modes may be included, such as immediate operands and relative mode.

By using a relatively simple instruction format, the instruction length can be fixed and aligned on word boundaries. An important aspect of RISC instruction format is that it is easy to decode. Thus the operation code and register fields of the instruction code can be accessed simultaneously by the control. By simplifying the instructions and their format, it is possible to simplify the control logic. For faster operations, a hardwired control is preferable over a microprogrammed control. An example of hardwired control is presented in Chap. 5 in conjunction with the control unit of the basic computer. Examples of microprogrammed control are presented in Chap. 7.

A characteristic of RISC processors is their ability to execute one instruction per clock cycle. This is done by overlapping the fetch, decode, and execute phases of two or three instructions by using a procedure referred to as pipelining. A load or store instruction may require two clock cycles because access to

memory takes more time than register operations. Efficient pipelining, as well as a few other characteristics, are sometimes attributed to RISC, although they may exist in non-RISC architectures as well. Other characteristics attributed to RISC architecture are:

1. A relatively large number of registers in the processor unit
2. Use of overlapped register windows to speed-up procedure call and return
3. Efficient instruction pipeline
4. Compiler support for efficient translation of high-level language programs into machine language programs

A large number of registers is useful for storing intermediate results and for optimizing operand references. The advantage of register storage as opposed to memory storage is that registers can transfer information to other registers much faster than the transfer of information to and from memory. Thus register-to-memory operations can be minimized by keeping the most frequent accessed operands in registers. Studies that show improved performance for RISC architecture do not differentiate between the effects of the reduced instruction set and the effects of a large register file. For this reason a large number of registers in the processing unit are sometimes associated with RISC processors. The use of overlapped register windows when transferring program control after a procedure call is explained below. Instruction pipeline in RISC is presented in Sec. 9-5 after we explain the concept of pipelining.

pipelining

Overlapped Register Windows

Procedure call and return occurs quite often in high-level programming languages. When translated into machine language, a procedure call produces a sequence of instructions that save register values, pass parameters needed for the procedure, and then calls a subroutine to execute the body of the procedure. After a procedure return, the program restores the old register values, passes results to the calling program, and returns from the subroutine. Saving and restoring registers and passing of parameters and results involve time-consuming operations. Some computers provide multiple-register banks, and each procedure is allocated its own bank of registers. This eliminates the need for saving and restoring register values. Some computers use the memory stack to store the parameters that are needed by the procedure, but this requires a memory access every time the stack is accessed.

A characteristic of some RISC processors is their use of *overlapped register windows* to provide the passing of parameters and avoid the need for saving and restoring register values. Each procedure call results in the allocation of

a new window consisting of a set of registers from the register file for use by the new procedure. Each procedure call activates a new register window by incrementing a pointer, while the return statement decrements the pointer and causes the activation of the previous window. Windows for adjacent procedures have overlapping registers that are shared to provide the passing of parameters and results.

The concept of overlapped register windows is illustrated in Fig. 8-9. The system has a total of 74 registers. Registers $R0$ through $R9$ are global registers that hold parameters shared by all procedures. The other 64 registers are divided into four windows to accommodate procedures A, B, C, and D. Each register window consists of 10 local registers and two sets of six registers common to adjacent windows. Local registers are used for local variables. Common registers are used for exchange of parameters and results between adjacent procedures. The common overlapped registers permit parameters to be passed without the actual movement of data. Only one register window is activated at any given time with a pointer indicating the active window. Each procedure call activates a new register window by incrementing the pointer. The high registers of the calling procedure overlap the low registers of the called procedure, and therefore the parameters automatically transfer from calling to called procedure.

As an example, suppose that procedure A calls procedure B. Registers $R26$ through $R31$ are common to both procedures, and therefore procedure A stores the parameters for procedure B in these registers. Procedure B uses local registers $R32$ through $R41$ for local variable storage. If procedure B calls procedure C, it will pass the parameters through registers $R42$ through $R47$. When procedure B is ready to return at the end of its computation, the program stores results of the computation in registers $R26$ through $R31$ and transfers back to the register window of procedure A. Note that registers $R10$ through $R15$ are common to procedures A and D because the four windows have a circular organization with A being adjacent to D.

As mentioned previously, the 10 global registers $R0$ through $R9$ are available to all procedures. Each procedure in Fig. 8-9 has available a total of 32 registers while it is active. This includes 10 global registers, 10 local registers, six low overlapping registers, and six high overlapping registers. Other fixed-size register window schemes are possible, and each may differ in the size of the register window and the size of the total register file. In general, the organization of register windows will have the following relationships:

number of global registers = G

number of local registers in each window = L

number of registers common to two windows = C

number of windows = W

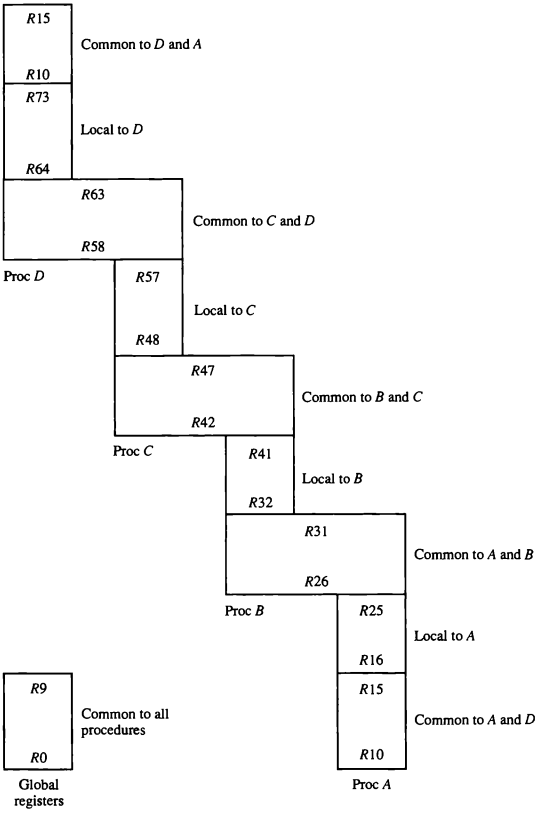


Figure 8-9 Overlapped register windows.

The number of registers available for each window is calculated as follows:

$$\text{window size} = L + 2C + G$$

The total number of registers needed in the processor is

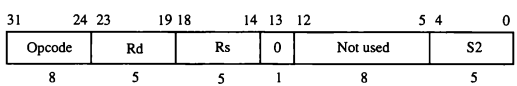
$$\text{register file} = (L + C)W + G$$

In the example of Fig. 8-9 we have $G = 10$, $L = 10$, $C = 6$, and $W = 4$. The window size is $10 + 12 + 10 = 32$ registers, and the register file consists of $(10 + 6) \times 4 + 10 = 74$ registers.

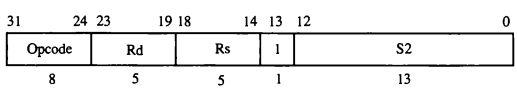
Berkeley RISC I

One of the first projects intended to show the advantages of RISC architecture was conducted at the University of California, Berkeley. The Berkeley RISC I is a 32-bit integrated circuit CPU. It supports 32-bit addresses and either 8-, 16-, or 32-bit data. It has a 32-bit instruction format and a total of 31 instructions. There are three basic addressing modes: register addressing, immediate operand, and relative to *PC* addressing for branch instructions. It has a register file of 138 registers arranged into 10 global registers and 8 windows of 32 registers in each. The 32 registers in each window have an organization similar to the one shown in Fig. 8-9. Since only one set of 32 registers in a window is

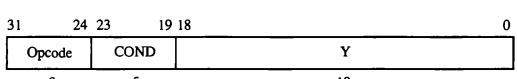
Figure 8-10 Berkeley RISC I instruction formats.



(a) Register mode: (S2 specifies a register)



(b) Register–immediate mode: (S2 specifies an operand)



(c) *PC* relative mode:

accessed at any given time, the instruction format can specify a processor register with a register field of five bits.

Figure 8-10 shows the 32-bit instruction formats used for register-to-register instructions and memory access instructions. Seven of the bits in the operation code specify an operation, and the eighth bit indicates whether to update the status bits after an ALU operation. For register-to-register instructions, the 5-bit Rd field selects one of the 32 registers as a destination for the result of the operation. The operation is performed with the data specified in fields Rs and S2. Rs is one of the source registers. If bit 13 of the instruction is 0, the low-order 5 bits of S2 specify another source register. If bit 13 of the instruction is 1, S2 specifies a sign-extended 13-bit constant. Thus the instruction has a three-address format, but the second source may be either a register or an immediate operand. Memory access instructions use Rs to specify a 32-bit address in a register and S2 to specify an offset. Register *R*0 contains all 0's, so it can be used in any field to specify a zero quantity. The third instruction format combines the last three fields to form a 19-bit relative address Y and is used primarily with the jump and call instructions. The COND field replaces the Rd field for jump instructions and is used to specify one of 16 possible branch conditions.

The 31 instructions of RISC I are listed in Table 8-12. They have been grouped into three categories. Data manipulation instructions perform arithmetic, logic, and shift operations. The symbols under the opcode and operands columns are used when writing assembly language programs. The register transfer and description columns explain the instruction in register transfer notation and in words. Note that all instructions have three operands. The second source S2 can be either a register or an immediate operand, symbolized by the number sign #. Consider, for example, the ADD instruction and how it can be used to perform a variety of operations.

```
ADD R22,R21,R23      R23←R22 + R21
ADD R22,#150,R23     R23←R22 + 150
ADD R0,R21,R22       R22←R21  (Move)
ADD R0,#150,R22      R22←150  (Load immediate)
ADD R22,#1,R22       R22←R22 + 1  (Increment)
```

By using register *R*0, which always contains 0's, it is possible to transfer the contents of one register or a constant into another register. The increment operation is accomplished by adding a constant 1 to a register.

The data transfer instructions consist of six load instructions, three store instructions, and two instructions that transfer the program status word PSW. The register that holds PSW contains the status of the CPU and includes the program counter, the status bits from the ALU, pointers used in conjunction with the register windows, and other information that determines the state of the CPU.

TABLE 8-12 Instruction Set of Berkeley RISC I

Opcode	Operands	Register Transfer	Description
Data manipulation instructions			
ADD	Rs,S2,Rd	$Rd \leftarrow Rs + S2$	Integer add
ADDC	Rs,S2,Rd	$Rd \leftarrow Rs + S2 +$ carry	Add with carry
SUB	Rs,S2,Rd	$Rd \leftarrow Rs - S2$	Integer subtract
SUBC	Rs,S2,Rd	$Rd \leftarrow Rs - S2 -$ carry	Subtract with carry
SUBR	Rs,S2,Rd	$Rd \leftarrow S2 - Rs$	Subtract reverse
SUBCR	Rs,S2,Rd	$Rd \leftarrow S2 - Rs -$ carry	Subtract with carry
AND	Rs,S2,Rd	$Rd \leftarrow Rs \wedge S2$	AND
OR	Rs,S2,Rd	$Rd \leftarrow Rs \vee S2$	OR
XOR	Rs,S2,Rd	$Rd \leftarrow Rs \oplus S2$	Exclusive-OR
SLL	Rs,S2,Rd	$Rd \leftarrow Rs$ shifted by $S2$	Shift-left
SRL	Rs,S2,Rd	$Rd \leftarrow Rs$ shifted by $S2$	Shift-right logical
SRA	Rs,S2,Rd	$Rd \leftarrow Rs$ shifted by $S2$	Shift-right arithmetic
Data transfer instructions			
LDL	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load long
LDSU	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load short unsigned
LDSS	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load short signed
LDBU	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load byte unsigned
LDBS	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load byte signed
LDHI	Rd,Y	$Rd \leftarrow Y$	Load immediate high
STL	Rd,(Rs)S2	$M[Rs + S2] \leftarrow Rd$	Store long
STS	Rd,(Rs)S2	$M[Rs + S2] \leftarrow Rd$	Store short
STB	Rd,(Rs)S2	$M[Rs + S2] \leftarrow Rd$	Store byte
GETPSW	Rd	$Rd \leftarrow PSW$	Load status word
PUTPSW	Rd	$PSW \leftarrow Rd$	Set status word
Program control instructions			
JMP	COND, S2(Rs)	$PC \leftarrow Rs + S2$	Conditional jump
JMPR	COND,Y	$PC \leftarrow PC + Y$	Jump relative
CALL	Rd,S2(Rs)	$Rd \leftarrow PC$ $PC \leftarrow Rs + S2$ $CWP \leftarrow CWP - 1$	Call subroutine and change window
CALLR	Rd,Y	$Rd \leftarrow PC$ $PC \leftarrow PC + Y$ $CWP \leftarrow$CWP $- 1$	Call relative and change window
RET	Rd,S2	$PC \leftarrow Rd + S2$ $CWP - CWP + 1$	Return and change window
CALLINT	Rd	$Rd \leftarrow PC$ $CWP \leftarrow CWP - 1$	Disable interrupts
RETINT	Rd,S2	$PC \leftarrow Rd + S2$ $CWP \leftarrow CWP + 1$	Enable interrupts
GTLPC	Rd	$Rd \leftarrow PC$	Get last PC

The load and store instructions move data between a register and memory. The load instructions accommodate signed or unsigned data of eight bits (byte) or 16 bits (short word). The long-word instructions operate on 32-bit data. Although there appears to be a register plus displacement addressing mode in data transfer instructions, register indirect addressing and direct addressing is also possible. The following are examples of load long instructions with different addressing modes.

```
LDL (R22)#150,R5     R5←M[R22] + 150
LDL (R22)#0,R5       R5←M[R22]
LDL (R0)#500,R5      R5←M[500]
```

The effective address in the first instruction is evaluated from the contents of register *R*22 plus a displacement of 150. The second instruction uses a 0 displacement, which reduces it to a register indirect mode. The third instruction uses all 0's from register *R*0 to produce a direct address type of instruction.

The program control instructions operate with the program counter *PC* to control the program sequence. There are two jump and two call instructions. One uses an index plus displacement addressing; the second uses a relative to *PC* mode with the 19-bit Y value as the relative address. The call and return instructions use a 3-bit *CWP* (current window pointer) register which points to the currently active register window. Every time the program calls a new procedure, *CWP* is decremented by one to point to the next-lower register window. After a return instruction *CWP* is incremented by one to return to the previous register window.

PROBLEMS

8-1. A bus-organized CPU similar to Fig. 8-2 has 16 registers with 32 bits in each, an ALU, and a destination decoder.

- **a.** How many multiplexers are there in the *A* bus, and what is the size of each multiplexer?
- **b.** How many selection inputs are needed for MUX A and MUX B?
- **c.** How many inputs and outputs are there in the decoder?
- **d.** How many inputs and outputs are there in the ALU for data, including input and output carries?
- **e.** Formulate a control word for the system assuming that the ALU has 35 operations.

8-2. The bus system of Fig. 8-2 has the following propagation delay times: 30 ns for the signals to propagate through the multiplexers, 80 ns to perform the ADD operation in the ALU, 20 ns delay in the destination decoder, and 10 ns to clock the data into the destination register. What is the minimum cycle time that can be used for the clock?

havioral characteristics of the computer system rather than its operational and structural interconnections. One type of parallel processing that does not fit Flynn's classification is pipelining. The only two categories used from this classification are SIMD array processors discussed in Sec. 9-7, and MIMD multiprocessors presented in Chap. 13.

In this chapter we consider parallel processing under the following main topics:

1. Pipeline processing
2. Vector processing
3. Array processors

Pipeline processing is an implementation technique where arithmetic suboperations or the phases of a computer instruction cycle overlap in execution. Vector processing deals with computations involving large vectors and matrices. Array processors perform computations on large arrays of data.

9-2 Pipelining

Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments. The name "pipeline" implies a flow of information analogous to an industrial assembly line. It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time. The overlapping of computation is made possible by associating a register with each segment in the pipeline. The registers provide isolation between each segment so that each can operate on distinct data simultaneously.

Perhaps the simplest way of viewing the pipeline structure is to imagine that each segment consists of an input register followed by a combinational circuit. The register holds the data and the combinational circuit performs the suboperation in the particular segment. The output of the combinational circuit in a given segment is applied to the input register of the next segment. A clock is applied to all registers after enough time has elapsed to perform all segment activity. In this way the information flows through the pipeline one step at a time.

an example

The pipeline organization will be demonstrated by means of a simple

example. Suppose that we want to perform the combined multiply and add operations with a stream of numbers.

$$A_i * B_i + C_i \qquad \text{for } i = 1, 2, 3, \ldots, 7$$

Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig. 9-2. $R1$ through $R5$ are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The suboperations performed in each segment of the pipeline are as follows:

$R1 \leftarrow A_i, \quad R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2, \quad R4 \leftarrow C_i$	Multiply and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table 9-1. The first clock pulse transfers A_1 and B_1 into $R1$ and

Figure 9-2 Example of pipeline processing.

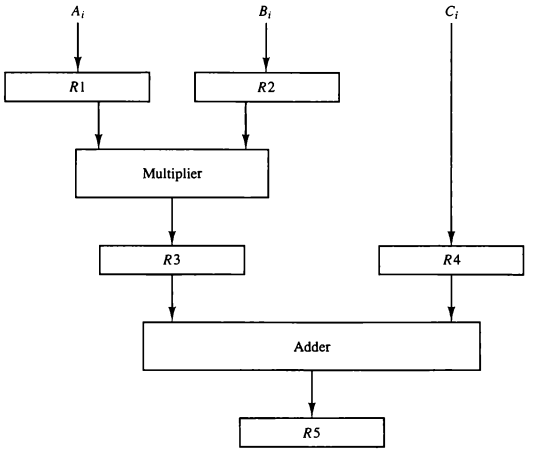


TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

$R2$. The second clock pulse transfers the product of $R1$ and $R2$ into $R3$ and C_1 into $R4$. The same clock pulse transfers A_2 and B_2 into $R1$ and $R2$. The third clock pulse operates on all three segments simultaneously. It places A_3 and B_3 into $R1$ and $R2$, transfers the product of $R1$ and $R2$ into $R3$, transfers C_2 into $R4$, and places the sum of $R3$ and $R4$ into $R5$. It takes three clock pulses to fill up the pipe and retrieve the first output from $R5$. From there on, each clock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

General Considerations

Any operation that can be decomposed into a sequence of suboperations of about the same complexity can be implemented by a pipeline processor. The technique is efficient for those applications that need to repeat the same task many times with different sets of data. The general structure of a four-segment pipeline is illustrated in Fig. 9-3. The operands pass through all four segments in a fixed sequence. Each segment consists of a combinational circuit S_i that performs a suboperation over the data stream flowing through the pipe. The segments are separated by registers R_i that hold the intermediate results between the stages. Information flows between adjacent stages under the control of a common clock applied to all the registers simultaneously. We define a *task* as the total operation performed going through all the segments in the pipeline.

task

space-time diagram

The behavior of a pipeline can be illustrated with a *space-time* diagram. This is a diagram that shows the segment utilization as a function of time. The space-time diagram of a four-segment pipeline is demonstrated in Fig. 9-4. The horizontal axis displays the time in clock cycles and the vertical axis gives the

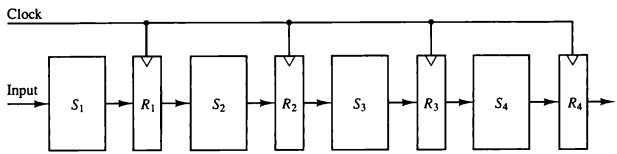


Figure 9-3 Four-segment pipeline.

segment number. The diagram shows six tasks T_1 through T_6 executed in four segments. Initially, task T_1 is handled by segment 1. After the first clock, segment 2 is busy with T_1, while segment 1 is busy with task T_2. Continuing in this manner, the first task T_1 is completed after the fourth clock cycle. From then on, the pipe completes a task every clock cycle. No matter how many segments there are in the system, once the pipeline is full, it takes only one clock period to obtain an output.

Now consider the case where a k-segment pipeline with a clock cycle time t_p is used to execute n tasks. The first task T_1 requires a time equal to kt_p to complete its operation since there are k segments in the pipe. The remaining $n - 1$ tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to $(n - 1)t_p$. Therefore, to complete n tasks using a k-segment pipeline requires $k + (n - 1)$ clock cycles. For example, the diagram of Fig. 9-4 shows four segments and six tasks. The time required to complete all the operations is $4 + (6 - 1) = 9$ clock cycles, as indicated in the diagram.

Next consider a nonpipeline unit that performs the same operation and takes a time equal to t_n to complete each task. The total time required for n tasks is nt_n. The speedup of a pipeline processing over an equivalent nonpipeline processing is defined by the ratio

speedup

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

Figure 9-4 Space-time diagram for pipeline.

Segment:	1	2	3	4	5	6	7	8	9	→ Clock cycles
1	T_1	T_2	T_3	T_4	T_5	T_6				
2		T_1	T_2	T_3	T_4	T_5	T_6			
3			T_1	T_2	T_3	T_4	T_5	T_6		
4				T_1	T_2	T_3	T_4	T_5	T_6	

As the number of tasks increases, n becomes much larger than $k - 1$, and $k + n - 1$ approaches the value of n. Under this condition, the speedup becomes

$$S = \frac{t_n}{t_p}$$

If we assume that the time it takes to process a task is the same in the pipeline and nonpipeline circuits, we will have $t_n = kt_p$. Including this assumption, the speedup reduces to

$$S = \frac{kt_p}{t_p} = k$$

This shows that the theoretical maximum speedup that a pipeline can provide is k, where k is the number of segments in the pipeline.

To clarify the meaning of the speedup ratio, consider the following numerical example. Let the time it takes to process a suboperation in each segment be equal to $t_p = 20$ ns. Assume that the pipeline has $k = 4$ segments and executes $n = 100$ tasks in sequence. The pipeline system will take $(k + n - 1)t_p = (4 + 99) \times 20 = 2060$ ns to complete. Assuming that $t_n = kt_p = 4 \times 20 = 80$ ns, a nonpipeline system requires $nkt_p = 100 \times 80 = 8000$ ns to complete the 100 tasks. The speedup ratio is equal to 8000/2060 = 3.88. As the number of tasks increases, the speedup will approach 4, which is equal to the number of segments in the pipeline. If we assume that $t_n = 60$ ns, the speedup becomes 60/20 = 3.

To duplicate the theoretical speed advantage of a pipeline process by means of multiple functional units, it is necessary to construct k identical units that will be operating in parallel. The implication is that a k-segment pipeline processor can be expected to equal the performance of k copies of an equivalent nonpipeline circuit under equal operating conditions. This is illustrated in Fig. 9-5, where four identical circuits are connected in parallel. Each P circuit performs the same task of an equivalent pipeline circuit. Instead of operating with the input data in sequence as in a pipeline, the parallel circuits accept four input data items simultaneously and perform four tasks at the same time. As far as the speed of operation is concerned, this is equivalent to a four segment pipeline. Note that the four-unit circuit of Fig. 9-5 constitutes a single-instruction multiple-data (SIMD) organization since the same instruction is used to operate on multiple data in parallel.

There are various reasons why the pipeline cannot operate at its maximum theoretical rate. Different segments may take different times to complete their suboperation. The clock cycle must be chosen to equal the time delay of the segment with the maximum propagation time. This causes all other segments to waste time while waiting for the next clock. Moreover, it is not always

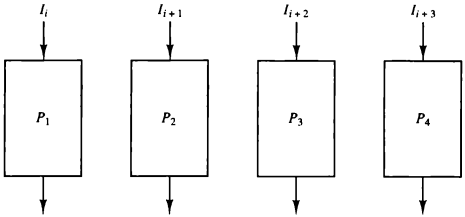


Figure 9-5 Multiple functional units in parallel.

correct to assume that a nonpipe circuit has the same time delay as that of an equivalent pipeline circuit. Many of the intermediate registers will not be needed in a single-unit circuit, which can usually be constructed entirely as a combinational circuit. Nevertheless, the pipeline technique provides a faster operation over a purely serial sequence even though the maximum theoretical speed is never fully achieved.

There are two areas of computer design where the pipeline organization is applicable. An *arithmetic pipeline* divides an arithmetic operation into suboperations for execution in the pipeline segments. An *instruction pipeline* operates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle. The two types of pipelines are explained in the following sections.

9-3 Arithmetic Pipeline

Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems. A pipeline multiplier is essentially an array multiplier as described in Fig. 10-10, with special adders designed to minimize the carry propagation time through the partial products. Floating-point operations are easily decomposed into suboperations as demonstrated in Sec. 10-5. We will now show an example of a pipeline unit for floating-point addition and subtraction.

The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.

$$X = A \times 2^a$$
$$Y = B \times 2^b$$

The sum is adjusted by normalizing the result so that it has a fraction with a nonzero first digit. This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain the normalized sum.

$$Z = 0.10324 \times 10^4$$

The comparator, shifter, adder-subtractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits. Suppose that the time delays of the four segments are $t_1 = 60$ ns, $t_2 = 70$ ns, $t_3 = 100$ ns, $t_4 = 80$ ns, and the interface registers have a delay of $t_r = 10$ ns. The clock cycle is chosen to be $t_p = t_3 + t_r = 110$ ns. An equivalent nonpipeline floating-point adder-subtractor will have a delay time $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320$ ns. In this case the pipelined adder has a speedup of 320/110 = 2.9 over the nonpipelined adder.

9-4 Instruction Pipeline

Pipeline processing can occur not only in the data stream but in the instruction stream as well. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and execute phases to overlap and perform simultaneous operations. One possible digression associated with such a scheme is that an instruction may cause a branch out of sequence. In that case the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.

Consider a computer with an instruction fetch unit and an instruction execution unit designed to provide a two-segment pipeline. The instruction fetch segment can be implemented by means of a first-in, first-out (FIFO) buffer. This is a type of unit that forms a queue rather than a stack. Whenever the execution unit is not using memory, the control increments the program counter and uses its address value to read consecutive instructions from memory. The instructions are inserted into the FIFO buffer so that they can be executed on a first-in, first-out basis. Thus an instruction stream can be placed in a queue, waiting for decoding and processing by the execution segment. The instruction stream queuing mechanism provides an efficient way for reducing the average access time to memory for reading instructions. Whenever there is space in the FIFO buffer, the control unit initiates the next instruction fetch phase. The buffer acts as a queue from which control then extracts the instructions for the execution unit.

instruction cycle

Computers with complex instructions require other phases in addition to the fetch and execute to process an instruction completely. In the most general case, the computer needs to process each instruction with the following sequence of steps.

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

There are certain difficulties that will prevent the instruction pipeline from operating at its maximum rate. Different segments may take different times to operate on the incoming information. Some segments are skipped for certain operations. For example, a register mode instruction does not need an effective address calculation. Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory. Memory access conflicts are sometimes resolved by using two memory buses for accessing instructions and data in separate modules. In this way, an instruction word and a data word can be read simultaneously from two different modules.

The design of an instruction pipeline will be most efficient if the instruction cycle is divided into segments of equal duration. The time that each step takes to fulfill its function depends on the instruction and the way it is executed.

Example: Four-Segment Instruction Pipeline

Assume that the decoding of the instruction can be combined with the calculation of the effective address into one segment. Assume further that most of the instructions place the result into a processor register so that the instruction execution and storing of the result can be combined into one segment. This reduces the instruction pipeline into four segments.

Figure 9-7 shows how the instruction cycle in the CPU can be processed with a four-segment pipeline. While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3. The effective address may be calculated in a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in an instruction FIFO. Thus up to four suboperations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.

Once in a while, an instruction in the sequence may be a program control type that causes a branch out of normal sequence. In that case the pending operations in the last two segments are completed and all information stored in the instruction buffer is deleted. The pipeline then restarts from the new address stored in the program counter. Similarly, an interrupt request, when acknowledged, will cause the pipeline to empty and start again from a new address value.

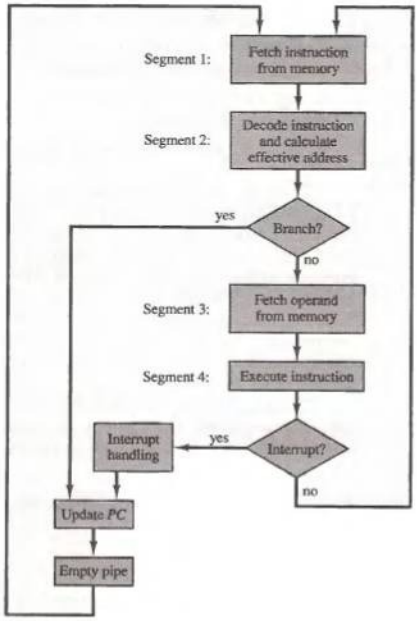


Figure 9-7 Four-segment CPU pipeline.

Figure 9-8 shows the operation of the instruction pipeline. The time in the horizontal axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.

1. FI is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

It is assumed that the processor has separate instruction and data memories so that the operation in FI and FO can proceed at the same time. In the absence

	Step:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction:	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	–	–	FI	DA	FO	EX			
	5					–	–	–	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

Figure 9-8 Timing of instruction pipeline.

of a branch instruction, each segment operates on different instructions. Thus, in step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI.

Assume now that instruction 3 is a branch instruction. As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6. If the branch is taken, a new instruction is fetched in step 7. If the branch is not taken, the instruction fetched previously in step 4 can be used. The pipeline then continues until a new branch instruction is encountered.

Another delay may occur in the pipeline if the EX segment needs to store the result of the operation in the data memory while the FO segment needs to fetch an operand. In that case, segment FO must wait until segment EX has finished its operation.

pipeline conflicts

In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

1. *Resource conflicts* caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.
2. *Data dependency* conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
3. *Branch difficulties* arise from branch and other instructions that change the value of *PC*.

Data Dependency

A difficulty that may caused a degradation of performance in an instruction pipeline is due to possible collision of data or address. A collision occurs when

an instruction cannot proceed because previous instructions did not complete certain operations. A data dependency occurs when an instruction needs data that are not yet available. For example, an instruction in the FO segment may need to fetch an operand that is being generated at the same time by the previous instruction in segment EX. Therefore, the second instruction must wait for data to become available by the first instruction. Similarly, an address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available. For example, an instruction with register indirect mode cannot proceed to fetch the operand if the previous instruction is loading the address into the register. Therefore, the operand access to memory must be delayed until the required address is available. Pipelined computers deal with such conflicts between data dependencies in a variety of ways.

hardware interlocks

The most straightforward method is to insert *hardware interlocks*. An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline. Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict. This approach maintains the program sequence by using hardware to insert the required delays.

operand forwarding

Another technique called *operand forwarding* uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments. For example, instead of transferring an ALU result into a destination register, the hardware checks the destination operand, and if it is needed as a source in the next instruction, it passes the result directly into the ALU input, bypassing the register file. This method requires additional hardware paths through multiplexers as well as the circuit that detects the conflict.

A procedure employed in some computers is to give the responsibility for solving data conflicts problems to the compiler that translates the high-level programming language into a machine language program. The compiler for such computers is designed to detect a data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting no-operation instructions. This method is referred to as *delayed load*. An example of delayed load is presented in the next section.

delayed load

Handling of Branch Instructions

One of the major problems in operating an instruction pipeline is the occurrence of branch instructions. A branch instruction can be conditional or unconditional. An unconditional branch always alters the sequential program flow by loading the program counter with the target address. In a conditional branch, the control selects the target instruction if the condition is satisfied or the next sequential instruction if the condition is not satisfied. As mentioned previously, the branch instruction breaks the normal sequence of the instruction stream, causing difficulties in the operation of the instruction pipeline.

Pipelined computers employ various hardware techniques to minimize the performance degradation caused by instruction branching.

prefetch target instruction

One way of handling a conditional branch is to prefetch the target instruction in addition to the instruction following the branch. Both are saved until the branch is executed. If the branch condition is successful, the pipeline continues from the branch target instruction. An extension of this procedure is to continue fetching instructions from both places until the branch decision is made. At that time control chooses the instruction stream of the correct program flow.

branch target buffer

Another possibility is the use of a *branch target buffer* or BTB. The BTB is an associative memory (see Sec. 12-4) included in the fetch segment of the pipeline. Each entry in the BTB consists of the address of a previously executed branch instruction and the target instruction for that branch. It also stores the next few instructions after the branch target instruction. When the pipeline decodes a branch instruction, it searches the associative memory BTB for the address of the instruction. If it is in the BTB, the instruction is available directly and prefetch continues from the new path. If the instruction is not in the BTB, the pipeline shifts to a new instruction stream and stores the target instruction in the BTB. The advantage of this scheme is that branch instructions that have occurred previously are readily available in the pipeline without interruption.

loop buffer

A variation of the BTB is the *loop buffer*. This is a small very high speed register file maintained by the instruction fetch segment of the pipeline. When a program loop is detected in the program, it is stored in the loop buffer in its entirety, including all branches. The program loop can be executed directly without having to access memory until the loop mode is removed by the final branching out.

branch prediction

Another procedure that some computers use is *branch prediction*. A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed. The pipeline then begins prefetching the instruction stream from the predicted path. A correct prediction eliminates the wasted time caused by branch penalties.

delayed branch

A procedure employed in most RISC processors is the *delayed branch*. In this procedure, the compiler detects the branch instructions and rearranges the machine language code sequence by inserting useful instructions that keep the pipeline operating without interruptions. An example of delayed branch is the insertion of a no-operation instruction after a branch instruction. This causes the computer to fetch the target instruction during the execution of the no-operation instruction, allowing a continuous flow of the pipeline. An example of delayed branch is presented in the next section.

9-5 RISC Pipeline

The reduced instruction set computer (RISC) was introduced in Sec. 8-8. Among the characteristics attributed to RISC is its ability to use an efficient instruction pipeline. The simplicity of the instruction set can be utilized to