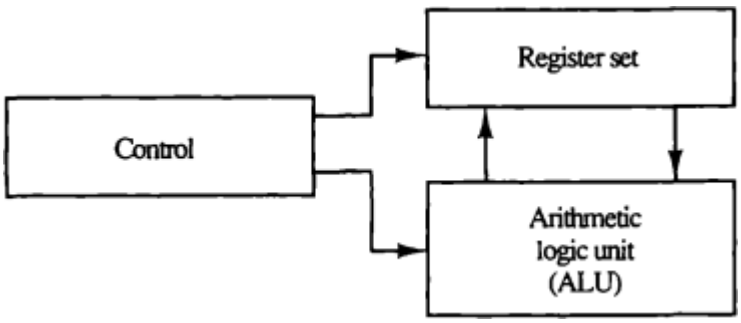
UNIT-III
Central Processing Unit

The part of the computer that performs the bulk of data-processing operations is called the central proofing unit and is refeired to as the CPU. The CPU is made up of three major parts, as shown in Figure. The register set stoics intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) perfoims the required mkraoperations for executing the instructions. The control unit supervises the transfe r 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 induction set provides the specifications for the design of the CPU. The design of a CPU is 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.



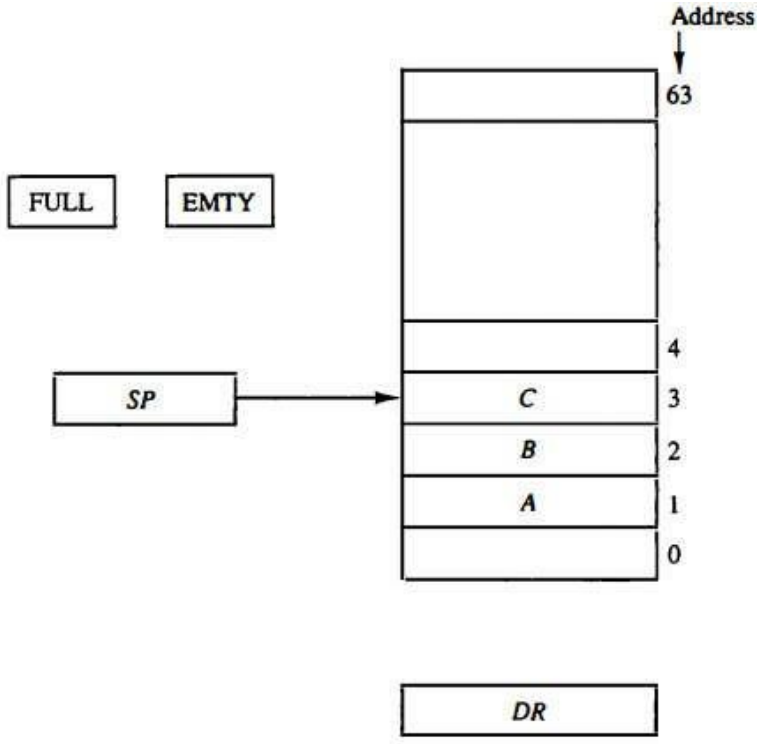
Stack Organization

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. 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 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 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 111 111 + 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 push operation. The push operation is implemented with the following sequence of microoperations:

SP ← SP + 1
M[SP] ← DR
If (SP = 0) then (FULL ← 1)
EMTY ← 0

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. 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 microoperations:

DR ← M[SP]
SP ← SP - 1
If (SP = 0) then (EMTY ← 1)
FULL ← 0

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.

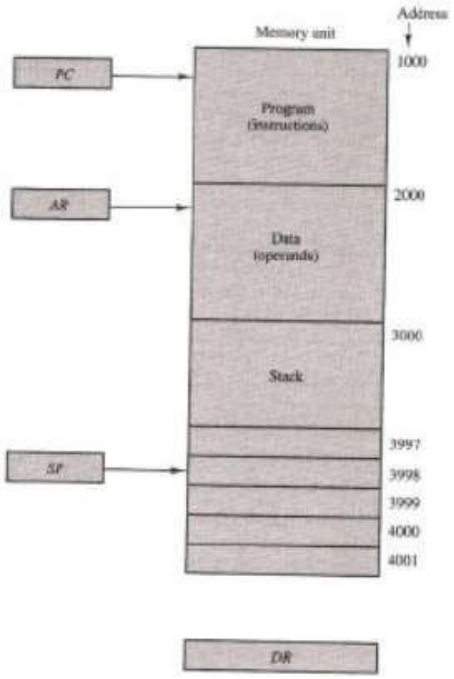
Memory Stack

A stack can exist as a stand-alone unit as in Figure 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 SP points a t the top of the stack. The three registers are eonnecied t o a common address bus, and either one can provide an addins for memory. PC is used during the (etch phase to lead an instruction. AR is used duiing the execute phase to read an opexand. SP is used to push or pop items into or from the stack. As shown in Figure, the initial value of SP is 4001 and the stack grows with decreasing addresses. Thus the fust 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 ← SP - 1
M[SP] ← 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 ← M[SP]
SP← 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. 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 Figure the stack grows by decreasing the memory address. The stack may be constructed to grow by increasing the memory address as in Figure. 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.

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 bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address. 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. 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 ← 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 Figure. 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 Rl, R2, R3 to denote, the operation R1 ← R2 + R3. 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.

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 Rl, A, B
ADD R2, C, D
MUL X, Rl, R2
```

It is assumed that the computer has two processor registers, R1 and R2. 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.

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 Rl, A
ADD Rl, B
MOV R2, C
ADD R2, D
MUL Rl, R2
MOV X, Rl
```

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
ADD B
STORE T
LOAD C
ADD D
MUL T
```

STORE X

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
PUSH B
ADD
PUSH C
PUSH D
ADD
MUL
POP X
```

To evaluate arithmetic expressions in a stack computer, i t 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.

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:

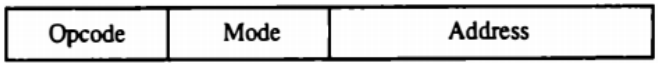
1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
2. To reduce the number of bits in the addressing field of the instruction.

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 program counter (PC) mode field 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.

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. 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. 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-type instruction. It is the address where control branches in response to a branch-type instruction.

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 already explained.

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.

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. 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 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, 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.

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

Table 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, XI 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.

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)+	$AC \leftarrow M[R1], R1 \leftarrow R1 + 1$

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

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. 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 arithmeticinstructions is given in Table. 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 l'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 l's. The add, subtract, multiply, and divide instructions maybe 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. It is not uncommon to find computers with three or more add instructions: one for binary integers, one for floating-point operands, and one for decimal operands.

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

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. The clear instruction causes the specified operand to be replaced by 0's. The complement instruction produces the l'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. 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 b0 = 0 and x bl = x dictate that a binary variable ANDed with a 0 produces a 0; but the variable 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 O' s in a selected portion of an operand. The OR

instruction is used to set a bit or a selected group of bits of an operand. Forany 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 l's in the bit positions that must be set to 1. Similarly, the XOR instruction i s used to selectively complement bits of an operand. This is because of the Boolean relationships x XOR l = x' and x XOR 0 = x. Thus a binary variable is complemented when XORed with a 1 but does not change in value when XORed with a 0.

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

A few other bit manipulation instructions are included in Table. 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 lists four types of 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

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. 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.

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.

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

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 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.

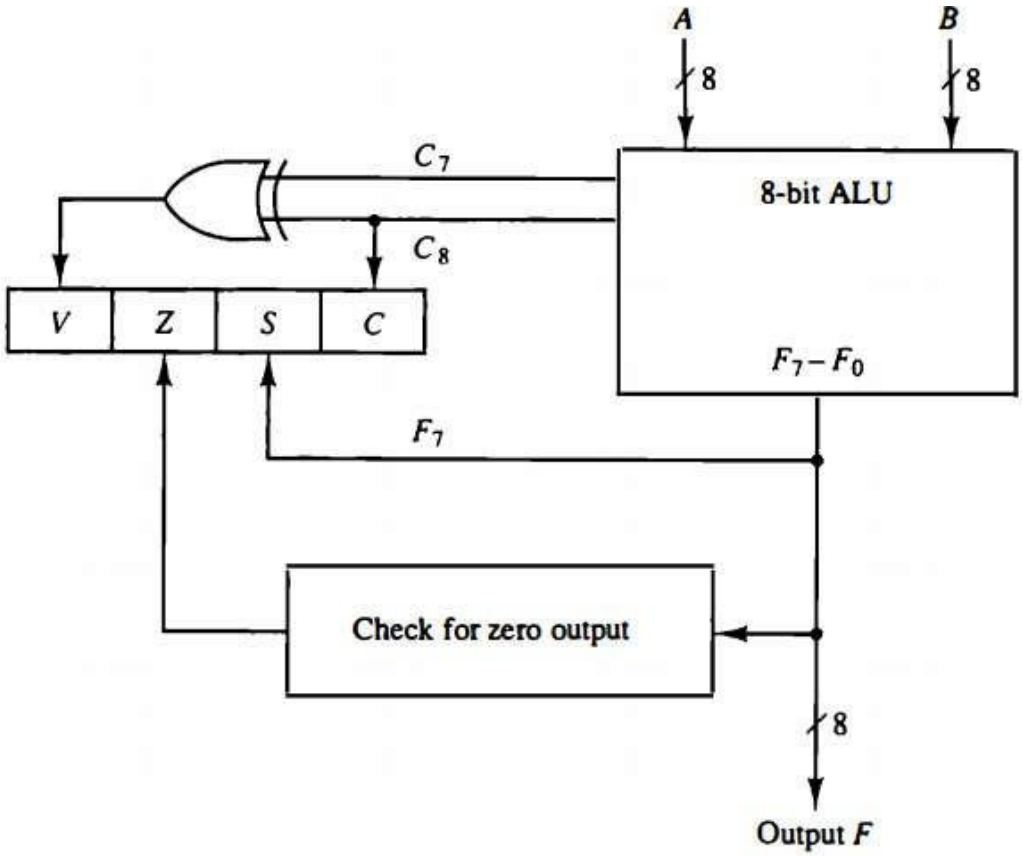
The call and return instructions are used in conjunction with subroutines. Their performance and implementation are discussed later in this section. The compare and test instructions do not change the program sequence directly. They are listed in Table because of their application in setting conditions 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 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 Fy 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 overflow when negative numbers are in 2's complement. 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 XOR 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, if the original content of A must be preserved.

Conditional Branch Instructions

Table 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 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.

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	$A \neq B$

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.

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. Let A = 11110000 and B = 00010100. To perform A - B, the ALU takes the 2's complement of B and adds it to A.

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

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 is not equal to 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 is not equal to 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.

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 ← SP - 1	Decrement stack pointer
M[SP] ← PC	Push content of PC onto the stack
PC ← 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 ← M[SP]	Pop stack and transfer to PC
SP ← 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

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 thelast 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.

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.

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. 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.

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.