Subject title: Allied 4: COMPUTER SYSTEM ARCHITECTURE

YEAR 2018-19 ONWARDS SEMESTER: IV

SUBJECT CODE: 18BIT44A

UNIT III: Central processing unit: general register organization – stack organization – instruction formats – addressing modes – data transfer and manipulation – programmed control – reduced instruction set computer – CISC.

Central processing unit:

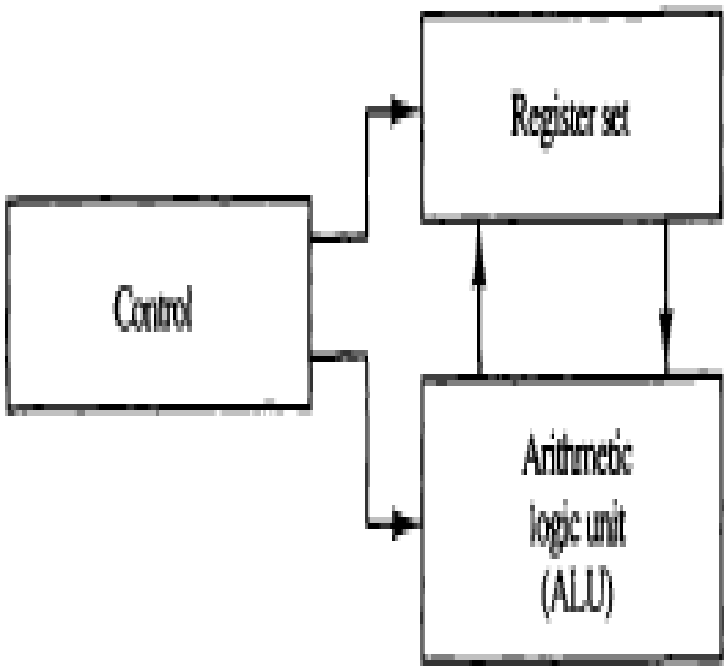
INTRODUCTION

The part of the computer that performs the bull< 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. The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required micro operations 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.

Components of CPU



General register organization

A bus organization for seven CPU registers is shown in Fig. 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 common arithmetic logic unit (ALU). The operation selected in the ALU determines the arithmetic or logic micro operation that is to be performed. The result of the micro operation 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

R 1 <--R2 + R3

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

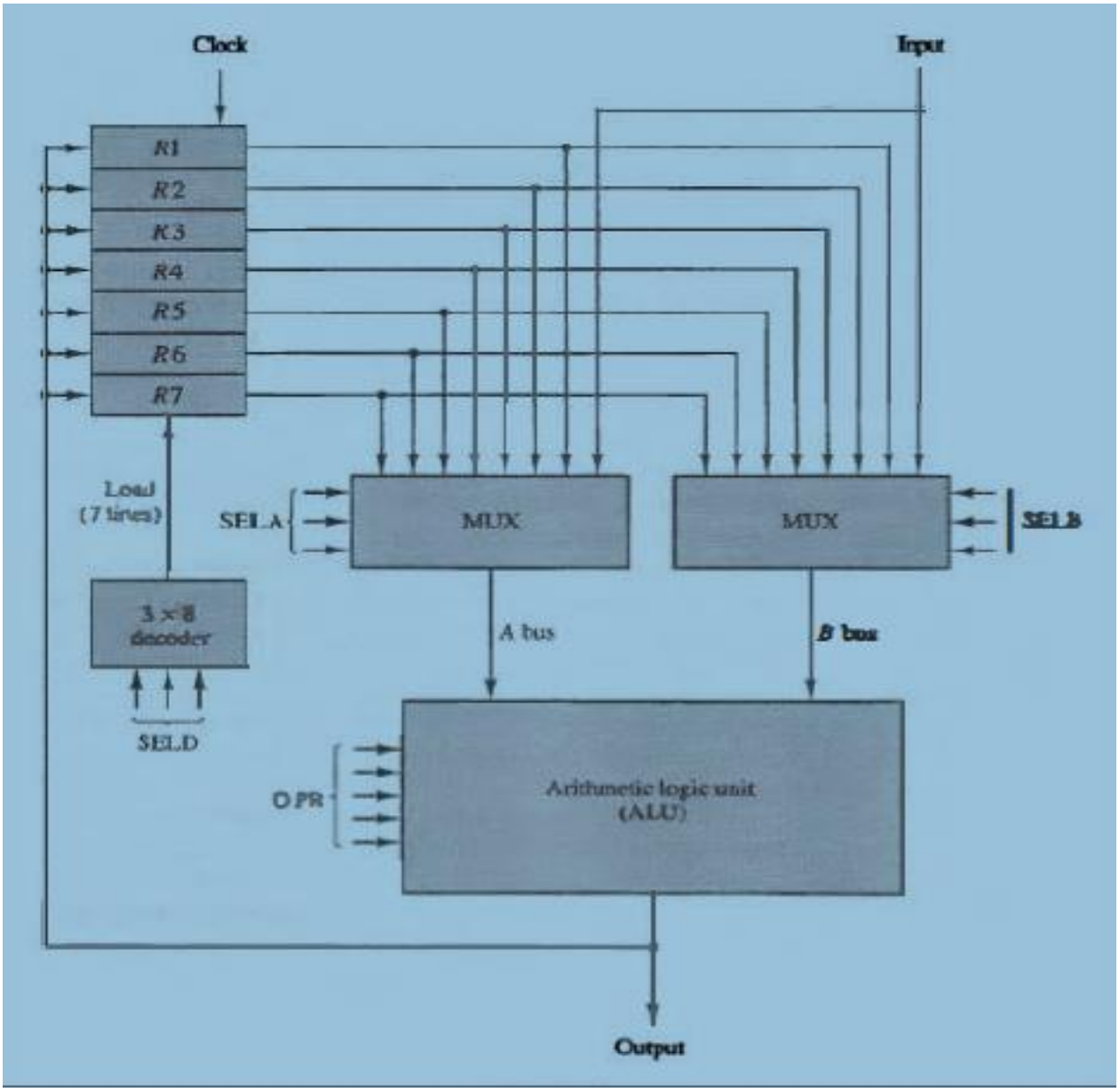
1. MUX A selector (SELA): to place the content of R2 into bus A .

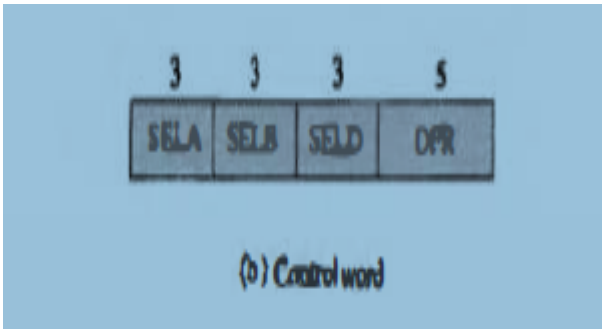
2 . MUX B selector (SELB): to place the content o f R 3 into bus B .

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 R 1. To achieve a fast response time, the ALU is constructed with high-speed circuits.





(b) Control word

Control Word

There are 14 binary selection inputs in the unit, and their combined value specifies a control word.

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

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 pre shift capability, or at the output of the

ALU to provide post shifting capability. In some cases, the shift operations are included with the ALU.

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 Micro operations

R 1 <- R 2 - 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.

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

Microoperation	Symbolic Designation 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

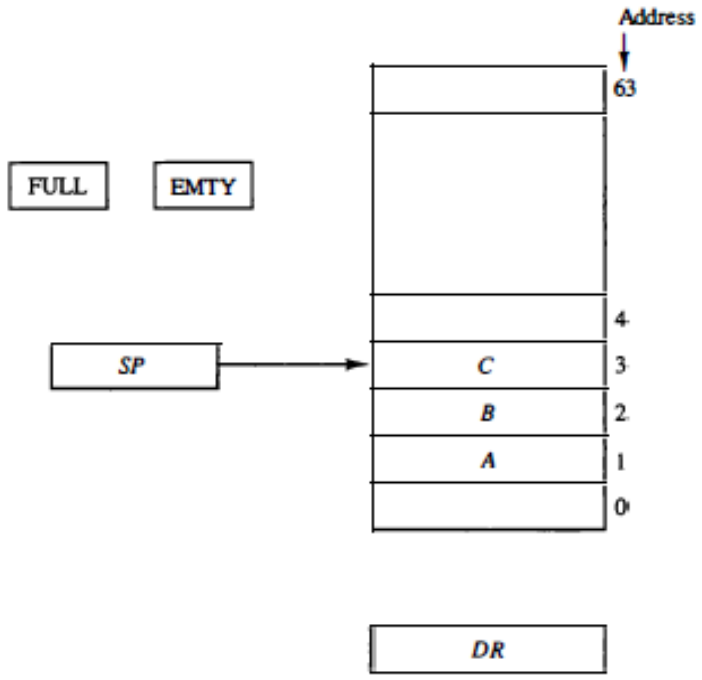
Stack organization

A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (UFO) 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 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.

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.



64 word stack

In a 64-word stack, the stack pointer contains 6 bits because 26 = 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 1 1 1 1 1 1 . 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 micro operations;

SP <- SP + 1
M [SP] <- DR
If (SP = 0) then (FULL <--1)
EMTY <--0
Check if stack is full
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.

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

Read item from the top of stack
Decrement stack pointer
Check if stack is empty
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 .

Instruction Formats

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.

Most computers fall into one of three types of CPU organizations:

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

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.

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 R2. The symbol M [A] denotes the operand at memory address symbolized by A .

The advantage o f the three-address format i s that i t 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.

Addressing Modes

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 .

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.

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. 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.

Data Transfer and Manipulation

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.

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

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

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

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.

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

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.

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

Program Control

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

Reduced Instruction Set Computer (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 micro programmed 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.

CISC Characteristics

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.