(23CS304) COMPUTER ORGANIZATION & ARCHITECTURE
UNIT-II

MICRO PROGRAMMED CONTROL: Control memory, Address sequencing, micro program example, design of control unit.
CENTRAL PROCESSING UNIT: General Register Organization, STACK organization, Instruction formats, Addressing modes, DATA Transfer and manipulation, Program control. Reduced Instruction set computer.

<u>MICRO PROGRAMMED CONTROL</u>

Introduction

- The function of the control unit in a digital computer is to initiate sequences of microoperations.
- When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired.
- Microprogramming is a second alternative for designing the control unit of a digital computer.
- The principle of microprogramming is an elegant and systematic method for controlling the microoperation sequences in a digital computer.
- In a bus-organized systems, the control signals that specify microoperations are groups of bits that select the paths in multiplexers, decoders, and arithmetic logic units.
- A control unit whose binary control variables are stored in memory is called a microprogrammed control unit.

  <u>Control Memory:</u>
- A memory that is part of a control unit is referred to as a control memory.
    - Each word in control memory contains within it a microinstruction.
    - A sequence of microinstructions constitutes a microprogram.
    - Can be either read-only memory (ROM) or writable control memory (dynamic microprogramming)
- A computer that employs a microprogrammed control unit will have two separate memories:
    - A main memory
    - A control memory
- The general configuration of a microprogrammed control unit is demonstrated as:
    - The control memory is assumed to be a ROM, within which all control information is permanently stored.
    - The control address register specifies the address of the microinstruction.
    - The control data register holds the microinstruction read from memory.
- Thus a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.

Microrogrammed Sequencer

- The next address generator is sometimes called a microprogram sequencer, as it determines the address sequence that is read from control memory.
- Typical functions of a microprogram sequencer are:
    - Incrementing the control address register by one
    - Loading into the control address register an address from control memory
    - Transferring an external address
    - Loading an initial address to start the control operations.

Pipeline Register

- The data register is sometimes called a pipeline register.
- It allows the execution of the microoperations specified by the control word simultaneously with the generation of the next microinstruction.
- This configuration requires a two-phase clock
- The system can operate by applying a single-phase clock to the address register.
- Without the control data register
- Thus, the control word and next-address information are taken directly from the control memory.

Advantages

- The main advantage of the microprogrammed control is the fact that once the hardware configuration is established; there should be no need for further hardware or wiring change.
- Most computers based on the reduced instruction set computer (RISC) architecture concept use hardwired control rather than a control memory with a microprogram. (Why?)

A Microprogram Control Unit that determines the Microinstruction Address to be executed in the next clock cycle

- In-line Sequencing
- Branch
- Conditional Branch
- Subroutine
- Loop
- Instruction OP-code mapping

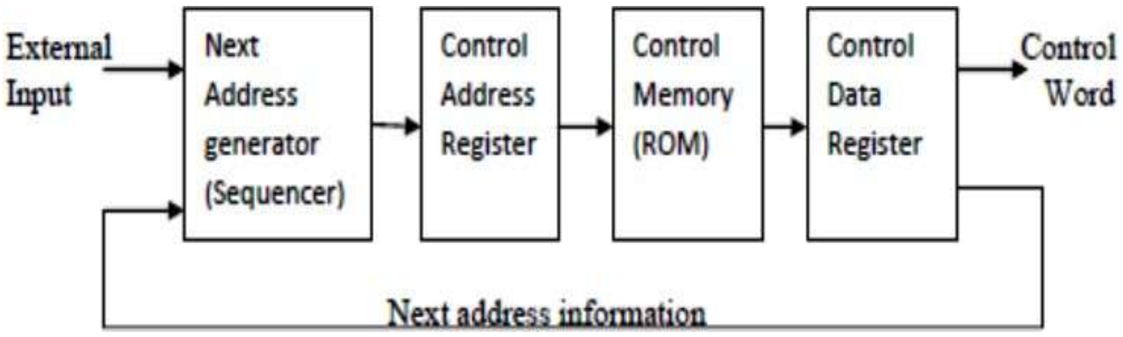


Fig: Microprogrammed Control Unit

<u>Addressing Sequencing:</u>

- Microinstructions are stored in control memory in groups, with each group specifying a routine.
- Each computer instruction has its own microprogram routine in control memory to generate the microoperations that execute the instruction.

  The address sequencing capabilities required in a control memory are:
    - Incrementing of the control address register
    - Unconditional branch or conditional branch, depending on status bit conditions
    - A mapping process from the bits of the instruction to an address for control memory
    - A facility for subroutine call and return
- The below figure shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address.
- The microinstruction in control memory contains
    - a set of bits to initiate microoperations in computer registers
    - Other bits to specify the method by which the next address is obtained

Sequencing Capabilities Required in Control Storage

- Incrementing of the control address register
- Unconditional and conditional branches
- A mapping process from the bits of the machine instruction to an address for control memory
- A facility for subroutine call and return

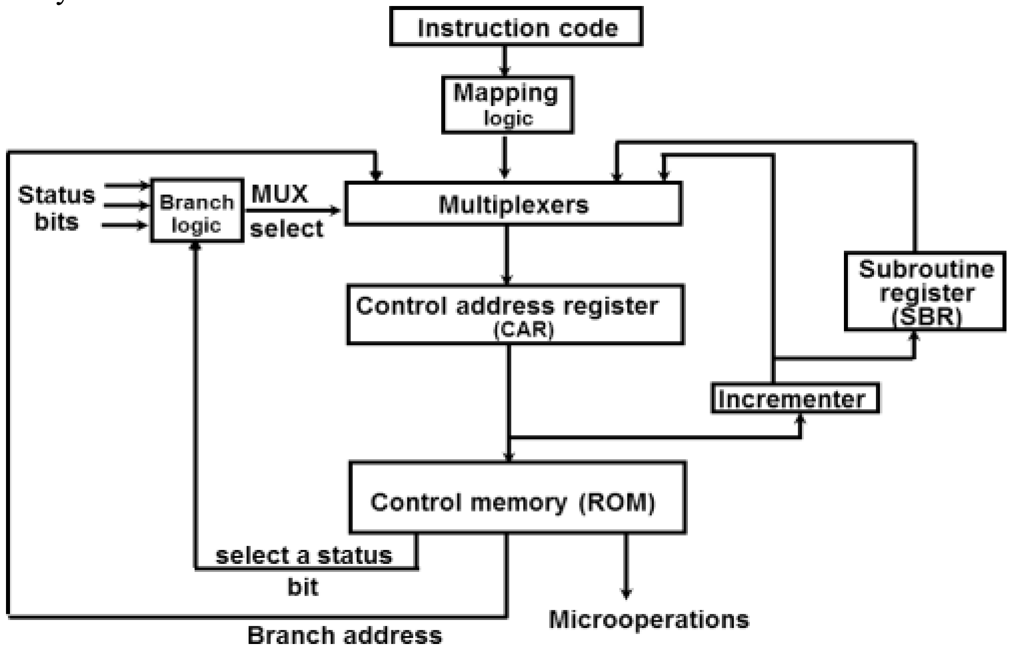


Fig: Selection Address Sequence for Control Memory

Conditional Branching

- The branch logic of Fig. 3-2 provides decision-making capabilities in the control unit.
- The status conditions are special bits in the system that provides parameter information.
  - e.g. the carry-out, the sign bit, the mode bits, and input or output status
- The status bits, together with the field in the microinstruction that specifies a branch address, control the conditional branch decisions generated in the branch logic.
- The branch logic hardware may be implemented by multiplexer.
  - Branch to the indicated address if the condition is met;
  - Otherwise, the address register is incremented.
- An unconditional branch microinstruction can be implemented by loading the branch address from control memory into the control address register.
- If Condition is true, then Branch (address from the next address field of the current microinstruction) else Fall Through
- Conditions to Test: O(overflow), N(negative), Z(zero), C(carry), etc.

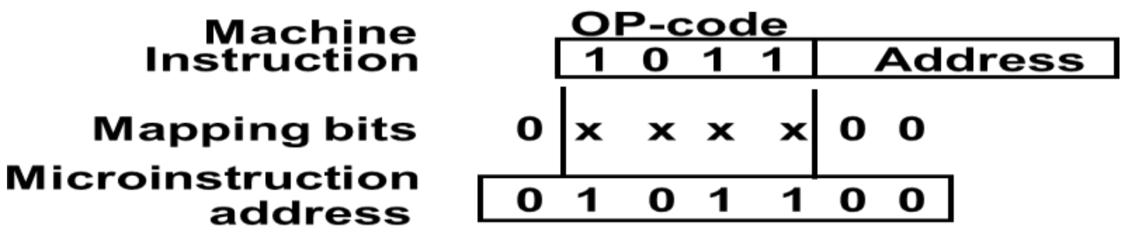
Unconditional Branch

- Fixing the value of one status bit at the input of the multiplexer to 1

Mapping of Instructions

- A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruction is located.
- The status bits for this type of branch are the bits in the operation code part of the instruction.
- One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in Fig. 3-3.
  - Placing a 0 in the most significant bit of the address
  - Transferring the four operation code bits

  - Clearing the two least significant bits of the control address register
- This provides for each computer instruction a microprogram routine with a capacity of four microinstructions.
  - If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111.
  - If it uses fewer than four microinstructions, the unused memory locations would be available for other routines.
- One can extend this concept to a more general mapping rule by using a ROM or programmable logic device (PLD) to specify the mapping function.



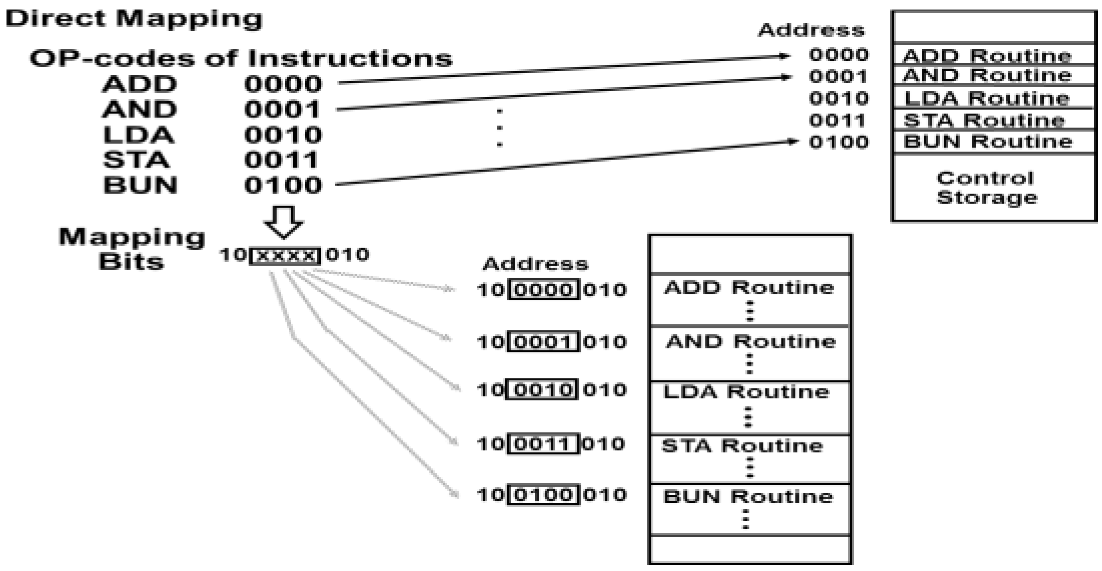


Fig: Direct Mapping

Mapping from the OP-code of an instruction to the address of the Microinstruction which is the starting microinstruction of its execution microprogram.

Subroutine

- Subroutines are programs that are used by other routines to accomplish a particular task.
- Microinstructions can be saved by employing subroutines that use common sections of microcode.
  - e.g. effective address computation
- The subroutine register can then become the source for transferring the address for the return to the main routine.
- The best way to structure a register file that stores addresses for subroutines is to organize the registers in a last-in, first-out (LIFO) stack.

Computer Configuration

- Once the configuration of a computer and its microprogrammed control unit is established, the designer's task is to generate the *microcode* for the control memory.
- This microcode generation is called *microprogramming*.
- The block diagram of the computer is shown in Below Fig.
- Two memory units
  - A main memory for storing instructions and data
  - A control memory for storing the microprogram
- Four registers are associated with the processor unit
  - Program counter *PC*, address register *AR*, data register *DR*, accumulator register *AC*
- The control unit has a control address register *CAR* and a subroutine register *SBR*.
- The control memory and its register are organized as a *microprogrammed control unit.*
- The transfer of information among the registers in the processor is done through *multiplexers rather than a common bus*.

Micro program Example:

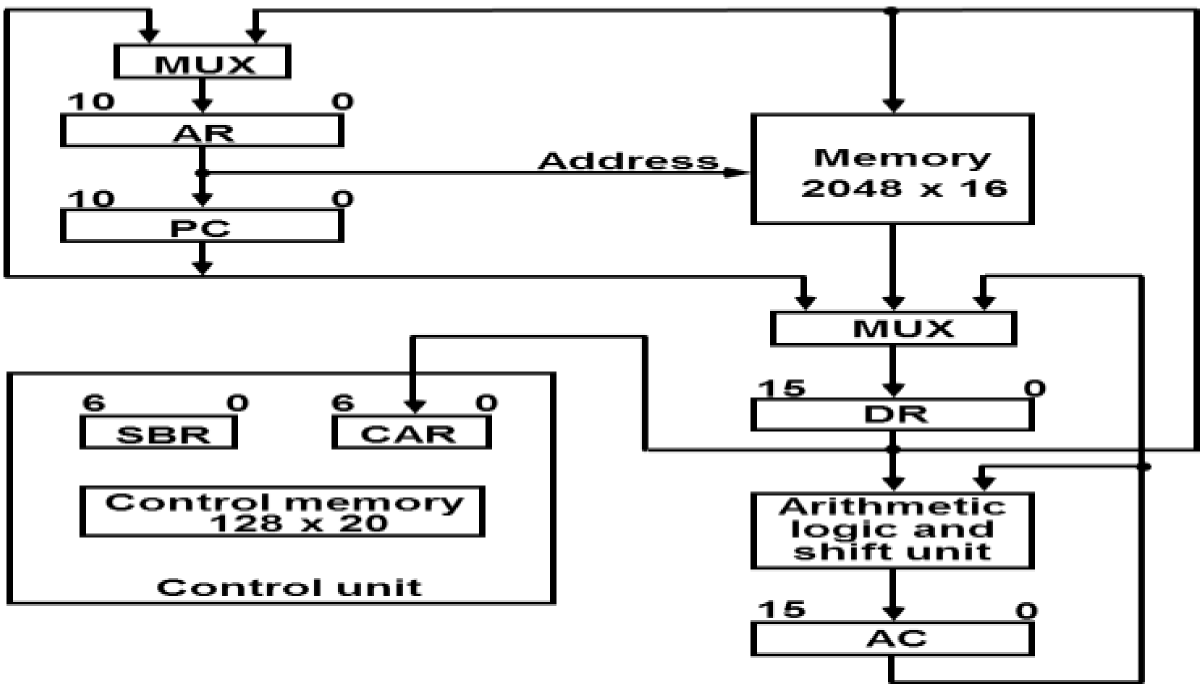


Fig: Computer Hardware Configuration

Microinstruction Format - Computer instruction Format

- The computer instruction format is depicted in Fig. (a).
- It consists of three fields:
    - A 1-bit field for indirect addressing symbolized by *I*
    - A 4-bit operation code (*opcode*)
    - An 11-bit address field
- Fig. (b) lists four of the 16 possible memory-reference instructions.

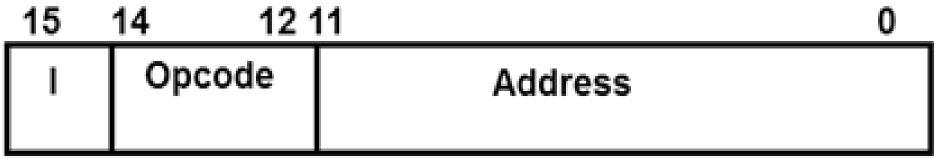
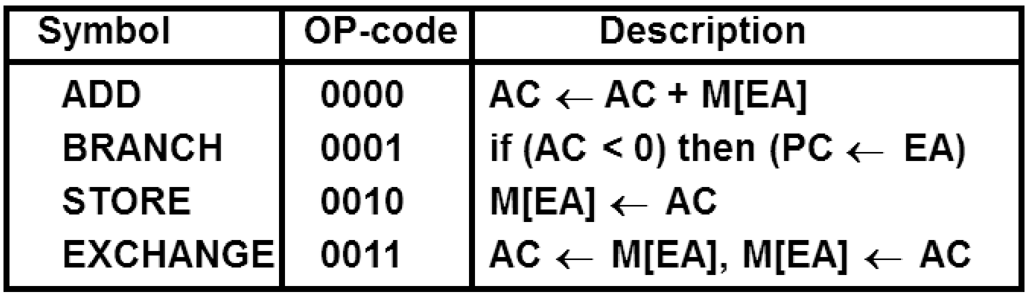


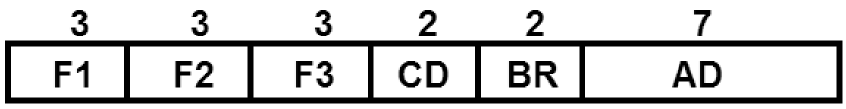
Fig.(a): Instruction Format



Symbol	OP-code	Description
ADD	0000	AC ← AC + M[EA]
BRANCH	0001	if (AC < 0) then (PC ← EA)
STORE	0010	M[EA] ← AC
EXCHANGE	0011	AC ← M[EA], M[EA] ← AC

Fig.(b): Computer Instruction(Four)

- The microinstruction format for the control memory is shown in below Fig.
- The 20 bits of the microinstruction are divided into four functional parts.
    - The three fields F1, F2, and F3 specify *microoperations* for the computer.
    - The CD field selects *status bit conditions*.
    - The BR field specifies *the type of branch*.
    - The AD field contains a *branch address*.



F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

Fig: Micro Instruction Code Format

Microoperations

- The three bits in each field are encoded to specify *seven distinct microoperations* as listed in below Table.
    - No more than *three* microoperations can be chosen for a microinstruction, one from each field.
    - If fewer than three microoperations are used, one or more of the fields will use the binary code 000 for no operation.

- It is important to realize that two or more conflicting microoperations cannot be specified simultaneously. e.g. 010 001 000
- Each microoperation in Table 3-1 is defined with a register transfer statement and is assigned a symbol for use in a **symbolic microprogram**.

F1	Microoperation	Symbol
000	None	NOP
001	AC ← AC + DR	ADD
010	AC ← 0	CLRAC
011	AC ← AC + 1	INCAC
100	AC ← DR	DRTAC
101	AR ← DR(0-10)	DRTAR
110	AR ← PC	PCTAR
111	M[AR] ← DR	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	AC ← AC - DR	SUB
010	AC ← AC ∨ DR	OR
011	AC ← AC ∧ DR	AND
100	DR ← M[AR]	READ
101	DR ← AC	ACTDR
110	DR ← DR + 1	INCDR
111	DR(0-10) ← PC	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	AC ← AC ⊕ DR	XOR
010	AC ← AC'	COM
011	AC ← shl AC	SHL
100	AC ← shr AC	SHR
101	PC ← PC + 1	INCPC
110	PC ← AR	ARTPC
111	Reserved	

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

BR	Symbol	Function
00	JMP	CAR ← AD if condition = 1
		CAR ← CAR + 1 if condition = 0
01	CALL	CAR ← AD, SBR ← CAR + 1 if condition = 1
		CAR ← CAR + 1 if condition = 0
10	RET	CAR ← SBR (Return from subroutine)
11	MAP	CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

Fig: Symbols and Binary codes for Microinstruction fields

- The CD field consists of two bits which are encoded to specify four status bitconditions as listed in above Table.
- The BR field consists of two bits. It is used, in conjunction with the address field AD, to choose the address of the next microinstruction.
    - The *jump and call* operations depend on the value of the CD field.
    - The two operations are identical except that a call microinstruction stores the *return address* in the subroutine register SBR.
    - Note that the last two conditions in the BR field are *independent of* the values in the CD and AD fields.

Symbolic Microinstructions

- The symbols defined in Table 3-1 cab be used to specify microinstructions in symbolic form.
- Symbols are used in microinstructions as in assembly language
- The simplest and most straightforward way to formulate an assembly language fora microprogram is to define symbols for each field of the microinstruction and to give users the capability for defining their own symbolic addresses.
- A symbolic microprogram can be translated into its binary equivalent by a microprogram assembler.

Sample Format

- Five fields: ***label; micro-ops; CD; BR; AD***
- The label field: may be empty or it may specify a symbolic address terminated with a colon
- The microoperations field: of one, two, or three symbols separated by commas , the NOP symbol is used when the microinstruction has no microoperations
- The CD field: one of the letters {U, I, S, Z} can be chosen where
    - U: Unconditional Branch
    - I: Indirect address bit
    - S: Sign of AC
    - Z: Zero value in AC
- The BR field: contains one of the four symbols {JMP, CALL, RET, MAP}
- The AD field: specifies a value for the address field of the microinstruction with one of {Symbolic address, NEXT, empty}
    - When the BR field contains a RET or MAP symbol, the AD field is left empty

Fetch Subroutine

During FETCH, Read an instruction from memory and decode the instruction and update PC.

- The first 64 words are to be occupied by the routines for the 16 instructions.
- The last 64 words may be used for any other purpose.
    - A convenient starting location for the fetch routine is address 64.
- The three microinstructions that constitute the fetch routine have been listed in three different representations.
    - The register transfer representation:

$AR \leftarrow PC$

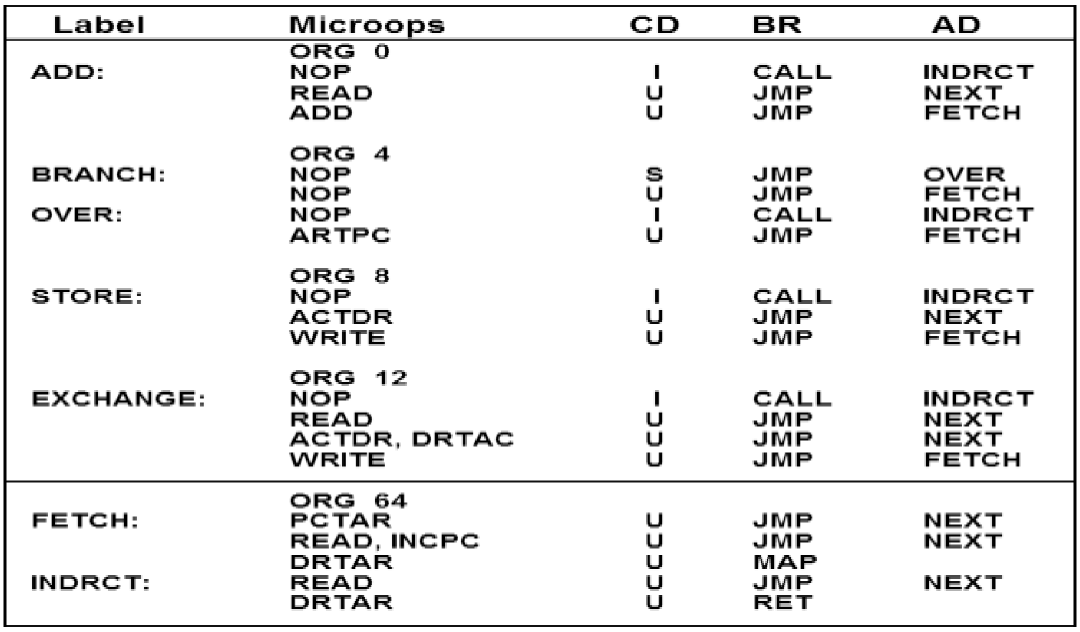
$DR \leftarrow M[AR], PC \leftarrow PC + 1$

$AR \leftarrow DR(0\text{-}10), CAR(2\text{-}5) \leftarrow DR(11\text{-}14), CAR(0,1,6) \leftarrow 0$

    - The symbolic representation:

```
          ORG 64
FETCH:    PCTAR          U  JMP  NEXT
          READ, INCPC    U  JMP  NEXT
          DRTAR          U  MAP
```

The binary representation:



Label	Microops	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

Symbolic Microprogram

- Control Storage: 128 20-bit words
- The first 64 words: Routines for the 16 machine instructions 0, 4, 8, …, 60 gives four words in control memory for each routine.
- The last 64 words: Used for other purpose (e.g., fetch routine and other subroutines)
- The execution of the third (MAP) microinstruction in the fetch routine results in a branch to address 0xxxx00, were xxxx are the four bits of the operation code. e.g. ADD is 0000
- In each routine we must provide microinstructions for evaluating the effective address and for executing the instruction.
- The indirect address mode is associated with all memory-reference instructions.
- A saving in the number of control memory words may be achieved if the microinstructions for the indirect address are stored as a subroutine.
- This subroutine, INDRCT, is located right after the fetch routine, as shown in Table.
- Mapping: OP-code XXXX into 0XXXX00, the first address for the 16 routines are 0(0 0000 00), 4(0 0001 00), 8, 12, 16, 20, ..., 60
- To see how the transfer and return from the indirect subroutine occurs:
  - MAP microinstruction caused a branch to address 0
  - The first microinstruction in the ADD routine calls subroutine INDRCT when *I*=1

- The return address is stored in the subroutine register *SBR*.
- The INDRCT subroutine has two microinstructions:

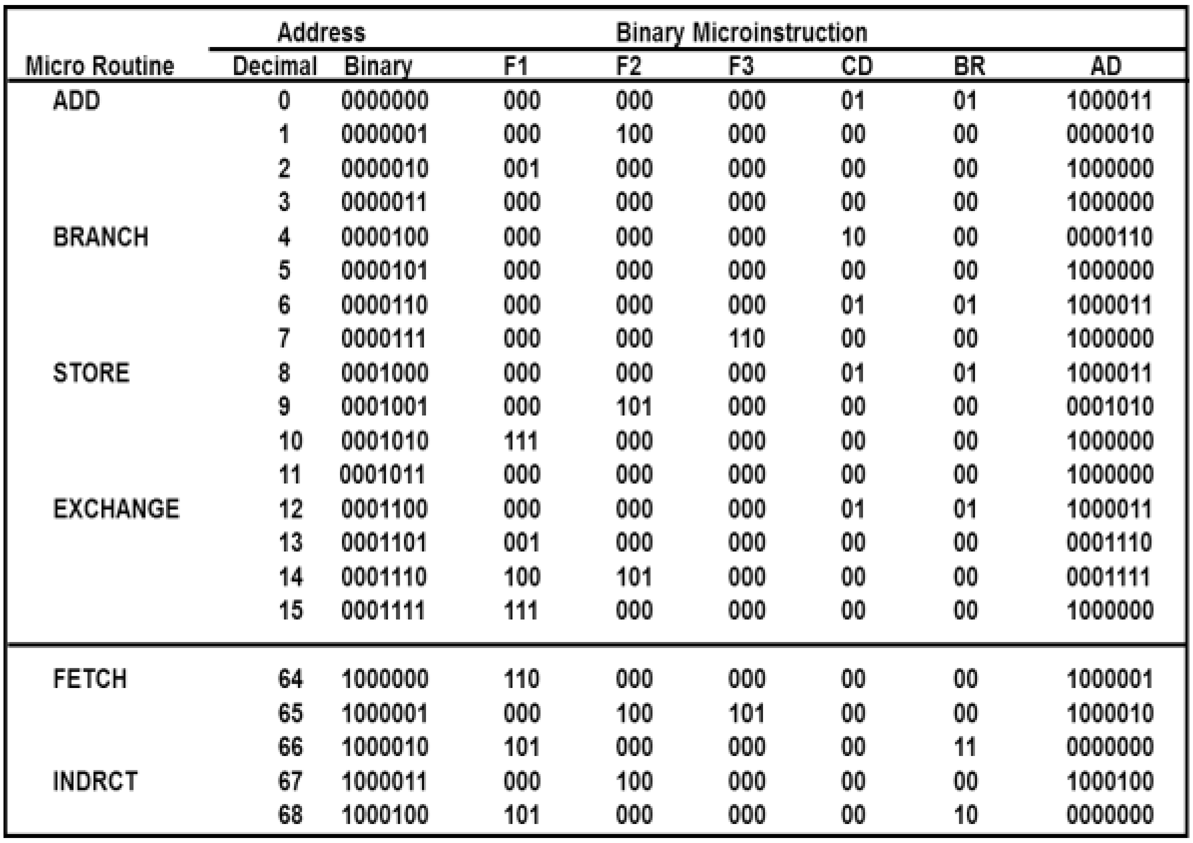
INDRCT: READ U JMP NEXT
DRTAR U RET

- Therefore, the memory has to be accessed to get the effective address, which is then transferred to *AR*.
- The execution of the ADD instruction is carried out by the microinstructions at addresses 1 and 2
- The first microinstruction reads the operand from memory into DR.

Table: Symbolic Microprogram for Control Memory

Binary Microprogram

- The symbolic microprogram must be translated to binary either by means of an assembler program or by the user if the microprogram is simple.
- The equivalent binary form of the microprogram is listed in Table.
- Even though address 3 is not used, some binary value, e.g. all 0's, must be specified for each word in control memory.
- However, if some unforeseen error occurs, or if a noise signal sets CAR to the value of 3, it will be wise to jump to address 64.



	Address		Binary Microinstruction					
Micro Routine	Decimal	Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
FETCH	64	1000000	110	000	000	00	00	1000001
	65	1000001	000	100	101	00	00	1000010
	66	1000010	101	000	000	00	11	0000000
INDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	101	000	000	00	10	0000000

Table: Binary Microprogram for Control Memory

---

2. **Hardwired Implementation**

- In this implementation, CU is essentially a combinational circuit. Its i/p signals are transformed into set of o/p logic signal which are control signals.
- Control unit inputs
- Flags and control bus
    - Each bit means something
- Instruction register
    - Op-code causes different control signals for each different instruction
    - Unique logic for each op-code
    - Decoder takes encoded input and produces single output
    - Each decoder i/p will activate a single unique o/p
- Clock
    - Repetitive sequence of pulses
    - Useful for measuring duration of micro-ops
    - Must be long enough to allow signal propagation along data paths and through processor circuitry
    - Different control signals at different times within instruction cycle
    - Need a counter as i/p to control unit with different control signals being used for t1, t2 etc.
    - At end of instruction cycle, counter is re-initialized

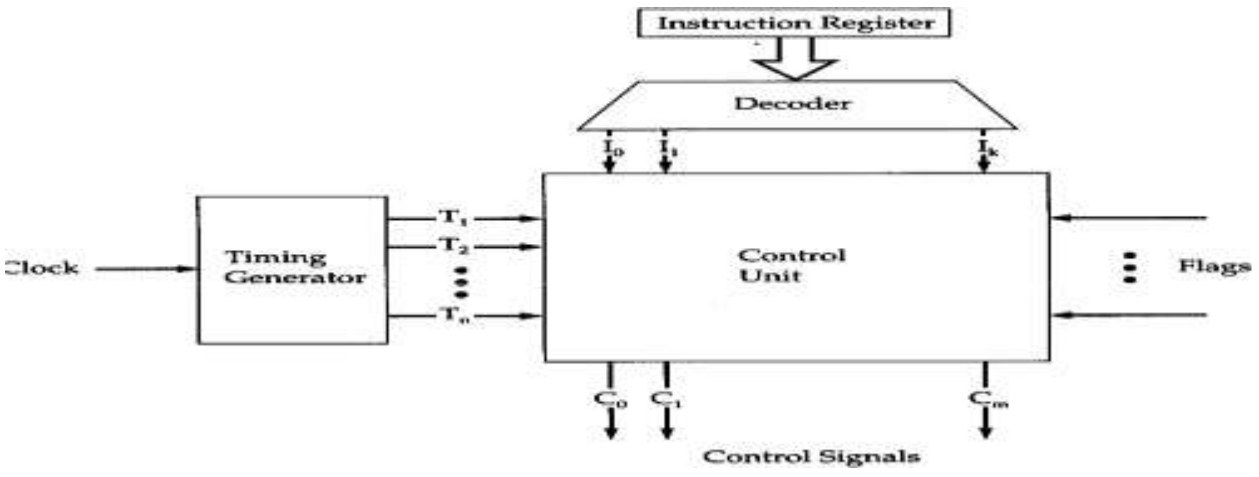


Fig : Control Unit With Decoded Input

Implementation

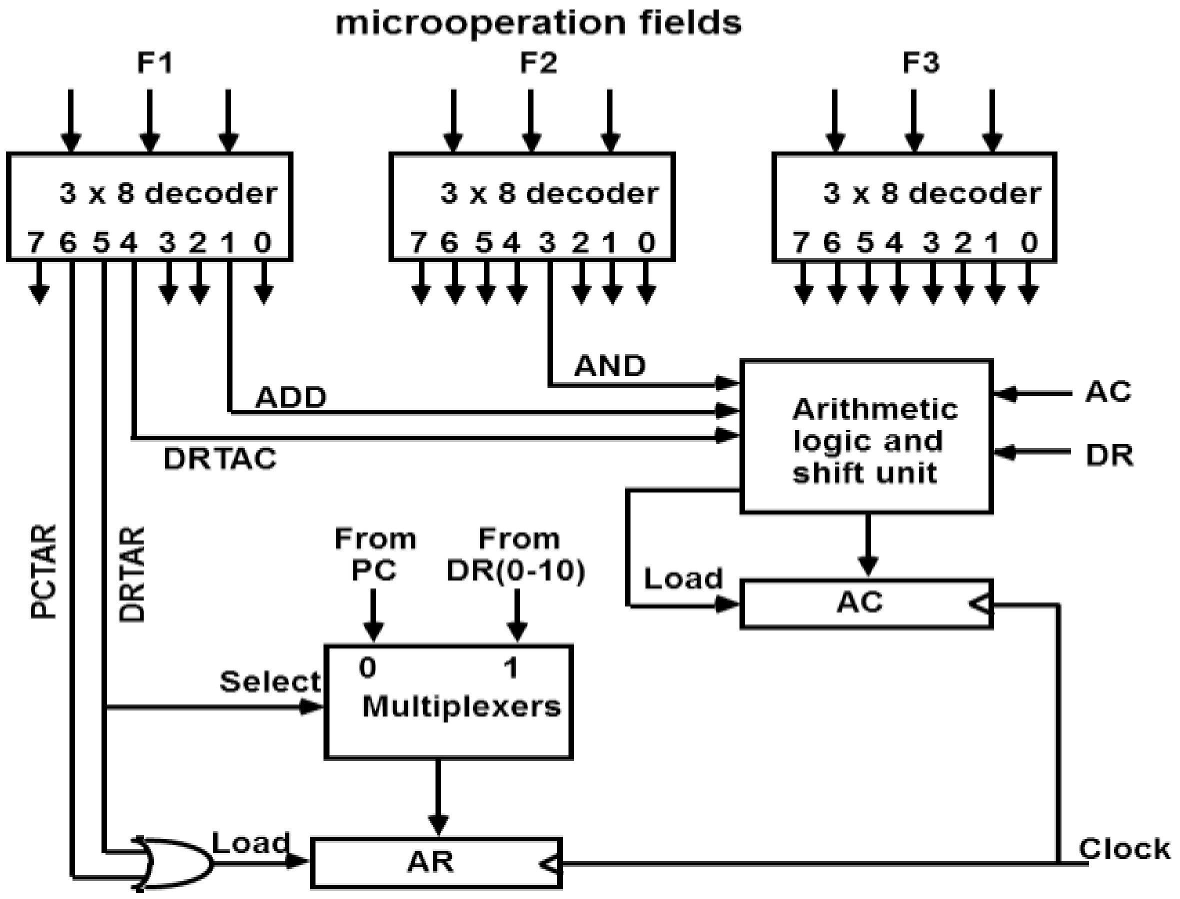
- For each control signal, a Boolean expression of that signal as a function of the inputs is derived
- With that the combinatorial circuit is realized as control unit.

Problems with hardwired designs

- Complex sequencing & micro-operation logic
- Difficult to design and test
- Inflexible design
- Difficult to add new instructions

Design of Control Unit:

- The bits of the microinstruction are usually divided into fields, with each field defining a distinct, separate function.
- The various fields encountered in instruction formats provide:
  - Control bits to initiate microoperations in the system
  - Special bits to specify the way that the next address is to be evaluated
  - An address field for branching
- The number of control bits that initiate microoperations can be reduced by grouping *mutually exclusive* variables into fields by encoding the *k* bits in each field to provide 2*k* microoperations.
- Each field requires a decoder to produce the corresponding control signals.
  - Reduces the size of the microinstruction bits
  - Requires additional hardware external to the control memory
  - Increases the delay time of the control signals
- Below Fig.shows the three decoders and some of the connections that must be made from their outputs.
- Outputs 5 or 6 of decoder F1 are connected to the load input of AR so that when either one of these outputs is active; information from the multiplexers is transferred to AR.
- The transfer into *AR* occurs with a clock pulse transition only when output 5 (from DR (0-10) to AR i.e. DRTAR) or output 6 (from PC to AR i.e. PCTAR) of the decoder are active.
- The arithmetic logic shift unit can be designed instead of using gates to generate the control signals; it comes from the outputs of the decoders.



-

Fig : Decoding of microoperation fields

Microprogram Sequencer

- The basic components of a microprogrammed control unit are the *control memory* and *the circuits that select the next address*.
- The address selection part is called a *microprogram sequencer*.
- A microprogram sequencer can be constructed with *digital functions* to suit a particular application.
- To guarantee a wide range of acceptability, an *integrated circuit sequencer* must provide an internal organization that can be adapted to a wide range of application.
- The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed.
- The block diagram of the microprogram sequencer is shown in below Fig.
- The control memory is included to show the interaction between the sequencer and the attached to it.
- There are two multiplexers in the circuit; first multiplexer selects an address from one of the four sources and routes to CAR, second multiplexer tests the value of the selected status bit and result is applied to an input logic circuit.
- The output from CAR provides the address for control memory, contents of CAR incremented and applied to one of the multiplexers input and to the SBR.
- Although the diagram shows a *single subroutine register,* a typical sequencer will have a *register stack* about four to eight levels deep. In this way, a push, pop operation and stack pointer operates for subroutine call and return instructions.
- The CD (Condition) field of the microinstruction selects one of the status bits in the second multiplexer.
- The Test variable (either 1 or 0) i.e. T value together with the two bits from the BR (Branch) field go to an input logic circuit.
- The input logic circuit determines the type of the operation.

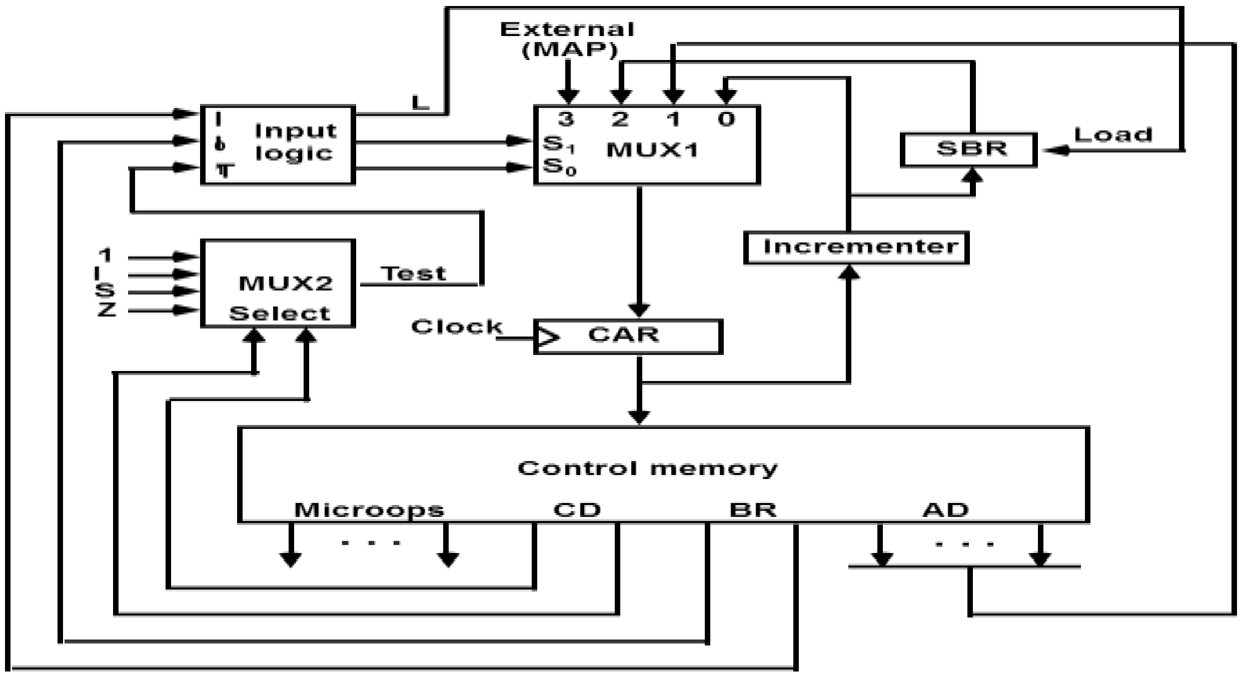


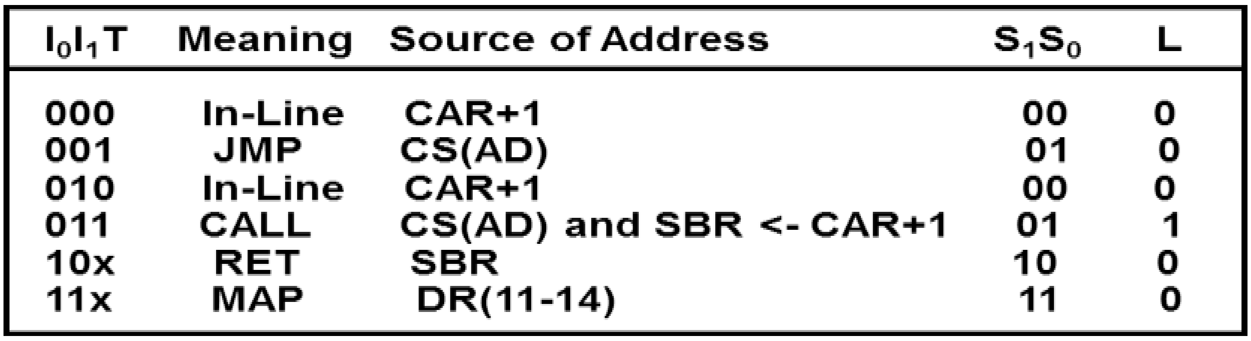
Fig: Microprogram Sequencer for a Control Memory

Design of Input Logic

- The input logic in a particular sequencer will determine the type of operations that are available in the unit.
- Typical sequencer operations are: *increment, branch or jump, call and return from subroutine, load an external address, push or pop the stack, and other address sequencing operations*.
- Based on the function listed in each entry was defined in previous Table, the truth table for the input logic circuit is shown in below Table.
- Therefore, the simplified Boolean functions for the input logic circuit can be given as:

  $S_0 = I_0$

  $S_1 = I_0 I_1 + I_0'T$



I_0I_1T	Meaning	Source of Address	S_1S_0	L
000	In-Line	CAR+1	00	0
001	JMP	CS(AD)	01	0
010	In-Line	CAR+1	00	0
011	CALL	CS(AD) and SBR <- CAR+1	01	1
10x	RET	SBR	10	0
11x	MAP	DR(11-14)	11	0

Table: Input logic truth table for microprogram sequencer

- The bit values for S1 and S0 are determined from the stated function and the path in the multiplexer that establishes the required transfer.
- Note that the incrementer circuit in the sequencer diagram is not a counter constructed with flip-flops but rather a combinational circuit constructed with gates.

CENTRAL PROCESSING UNIT:

Major Components of CPU:

1) Storage components- Registers, Flip-flops.
2) Execution components- ALU.
3) Transfer components- BUS.
4) Control components- Control Unit.

General Register Organization:—

A set of flip-flops forms a register,A register is a unique high-speed storage area in the CPU.
The number of registers in a processor unit may vary from just one processor register to as many as 64 registers or more.

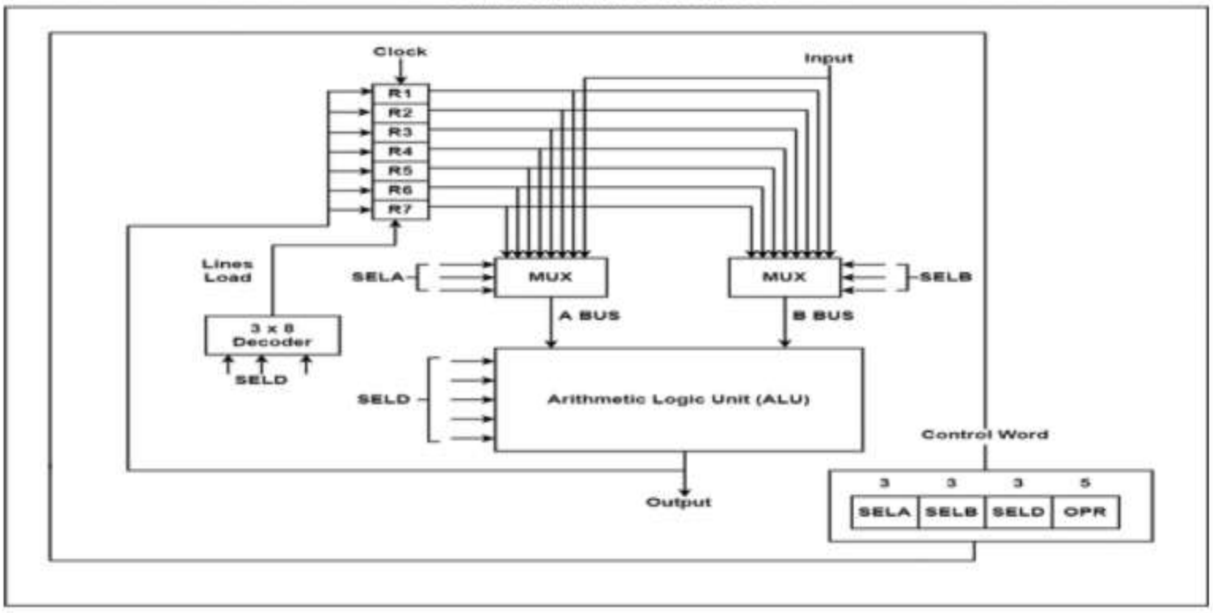
1. One of the CPU registers is called as an accumulator AC or 'A' register. It is the main operand register of the ALU.
2. The data register (DR) acts as a buffer between the CPU and main memory. It is used as an input operand register with the accumulator.
3. The instruction register (IR) holds the opcode of the current instruction.
4. The address register (AR) holds the address of the memory in which the operand resides.
5. The program counter (PC) holds the address of the next instruction to be fetched for execution.

Additional addressable registers can be provided for storing operands and address. This can be viewed as replacing the single accumulator by a set of registers. If the registers are used for many purposes, the resulting computer is said to have general register organization. In the case of processor registers, a register is selected by the multiplexers that form the buses.

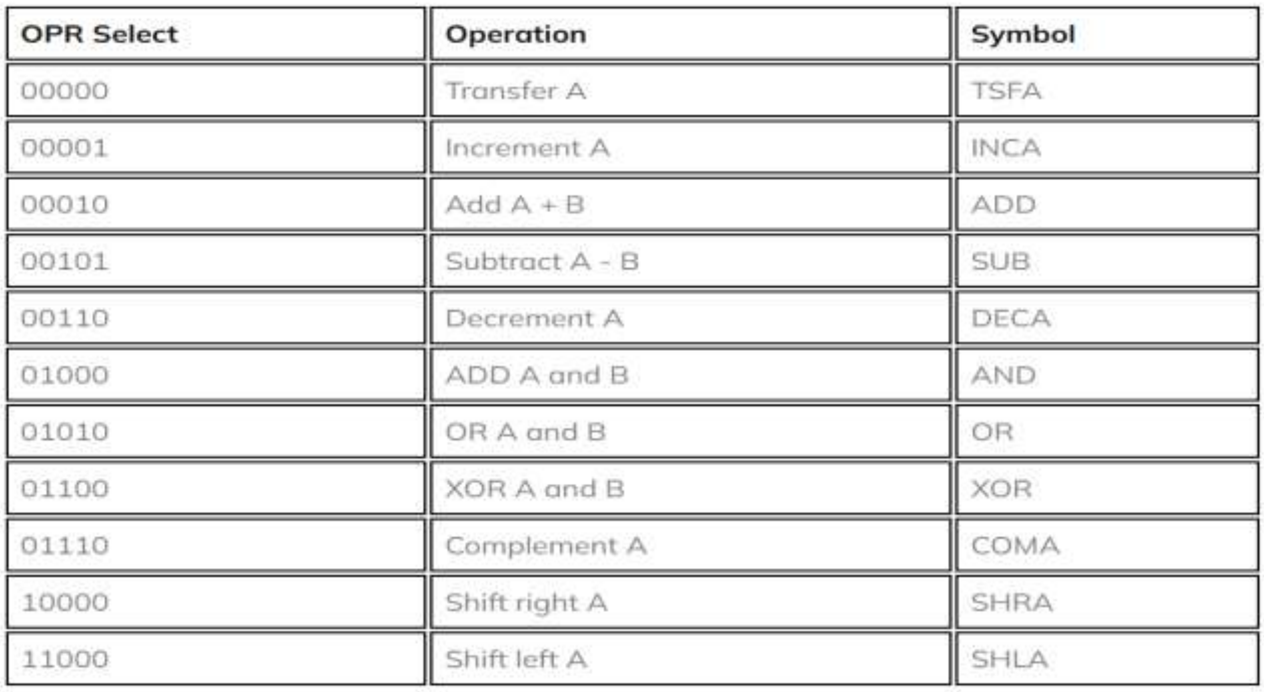
A Bus organization for seven CPU registers: —

The output of each register is connected to true multiplexer (mux) to form the two buses A & B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses forms the input to a common 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 and also goes into the inputs of 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 both 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 systems.

General Organization of Registers



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	ADD 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

R1 <= R2 + R3

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

(2) MUX B selection (SELB): to place the content of R3 into bus B

(3) ALU operation selection (OPR): to provide the arithmetic addition (A + B)

(4) Decoder destination selection (SELD): to transfer the content of the output bus into R1

These form the 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 multiplexer and the ALU, to the output bus, and into the of the destination registers, all during the clock cycle intervals.

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 register that holds the address for the stack is called a stack pointer (SP) because its value always points at the top item in stack.

The two operation of stack are the insertion and deletion of items.

1. The operation of insertion is called PUSH.
2. The operation of deletion is called POP

Register Stack:

Below 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. In a 64-word stack, the stack pointer contains 6 bits because 2^6=64.

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.

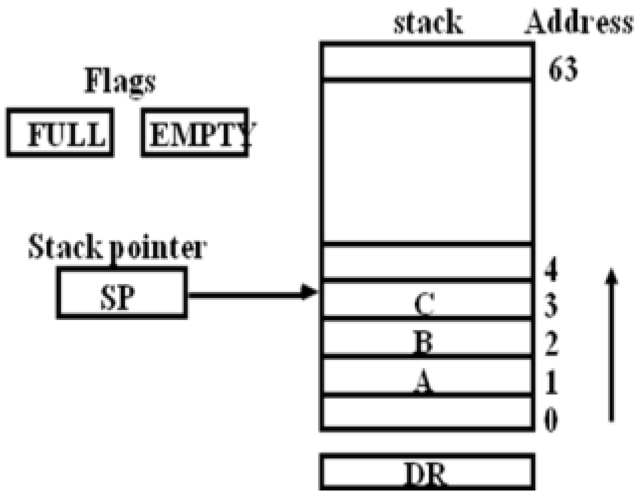


Fig: Block diagram of Stack

The **push** operation is implemented with the following sequence of micro-operation.

SP ←SP + 1 (Increment stack pointer)
M(SP) ← DR (Write item on top of the stack)
if (sp=0) then (Full ← 1) (Check if stack is full)
Emty ← 0 (Marked the stack not empty)

The **pop** operation is implemented with the following sequence of micro-operation.

DR← M[SP] Read item from the top of stack
SP ← SP-1 Decrement stack Pointer
if(SP=0) then (Emty ← 1) Check if stack is empty
FULL ← 0 Mark the stack not full

Memory Stack Organization:

Memory with Program, Data, and Stack Segment as showed below:

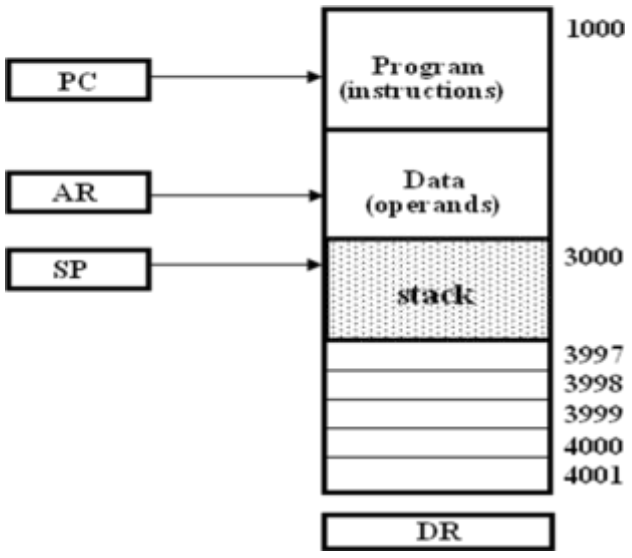


Fig: Computer memory with Program, data, and Stack

A portion of memory is used as a stack with a processor register as a stack pointer

PUSH: SP ← SP - 1
M[SP] ← DR

POP: DR ←M[SP]
SP ← SP + 1

Most computers do not provide hardware to check stack overflow (full stack) or underflow (empty stack).

Reverse Polish Notation (Postfix Notation):

Arithmetic Expressions: A + B

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

Any arithmetic expression can be expressed in parenthesis-free Polish notation, including reverse Polish notation.

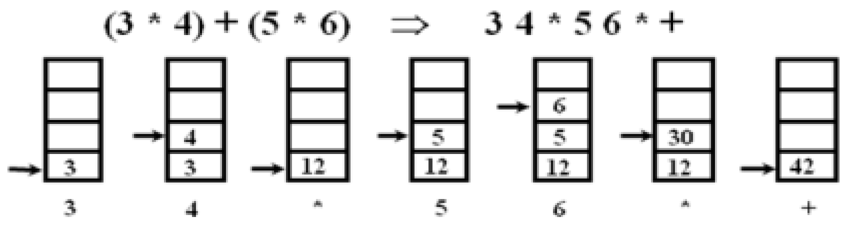


Fig: Reverse Polish Notation

Instruction Formats

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 operands or the effective address is determined.

Four Types of Address Instructions

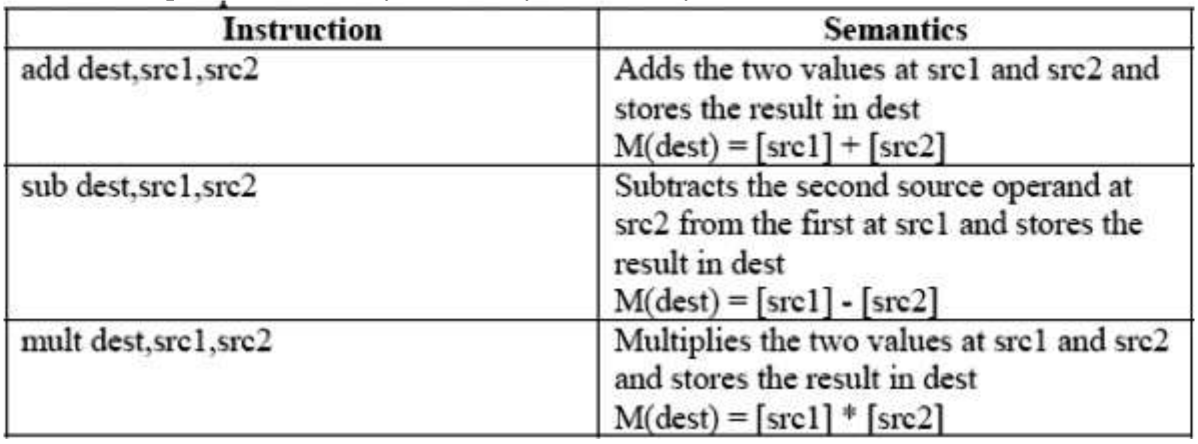
The division operation produces two outputs: a quotient and a remainder. Since most operations are binary, we need a total of three addresses: two addresses to specify the two input operands and one to specify where the result should go.

***a.* Three-Address Instructions:**

In three-address machines, instructions carry all three addresses explicitly. The RISC processors use three addresses. Table X1 gives some sample instructions of a three-address machine. In these machines, the C statement X = (A + B) ∗(C + D) is converted to the following code:

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, R1 and R2. The symbol M[A] denotes the operand at memory address symbolized by A.



Instruction	Semantics
add dest,src1,src2	Adds the two values at src1 and src2 and stores the result in dest M(dest) = [src1] + [src2]
sub dest,src1,src2	Subtracts the second source operand at src2 from the first at src1 and stores the result in dest M(dest) = [src1] - [src2]
mult dest,src1,src2	Multiplies the two values at src1 and src2 and stores the result in dest M(dest) = [src1] * [src2]

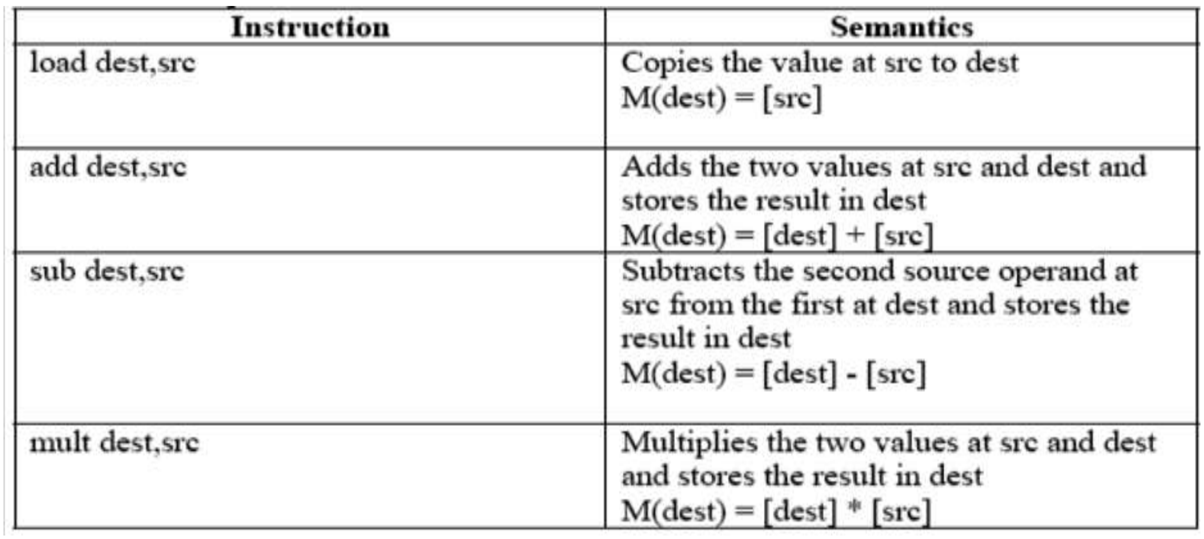
The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions.

The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

b.* 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



Instruction	Semantics
load dest,src	Copies the value at src to dest M(dest) = [src]
add dest,src	Adds the two values at src and dest and stores the result in dest M(dest) = [dest] + [src]
sub dest,src	Subtracts the second source operand at src from the first at dest and stores the result in dest M(dest) = [dest] - [src]
mult dest,src	Multiplies the two values at src and dest and stores the result in dest M(dest) = [dest] * [src]

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.

***c.* 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 and assume that the AC contains the result of tall operations. The program to evaluate X = (A + B) ∗(C + D) is

LOAD A AC ← M [A]
ADD B AC ← A [C] + 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.

***d.* 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

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 in 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.

3 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.

Addressing modes are as:

1. **Implied Mode**

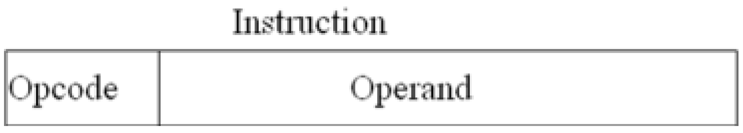
Address of the operands are specified implicitly in the definition of the instruction

- No need to specify address in the instruction
- EA = AC, or EA = Stack[SP**],** **EA: Effective Address.**

2. **Immediate Mode**

Instead of specifying the address of the operand, operand itself is specified

- No need to specify address in the instruction
- However, operand itself needs to be specified
- Sometimes, require more bits than the address
- Fast to acquire an operand



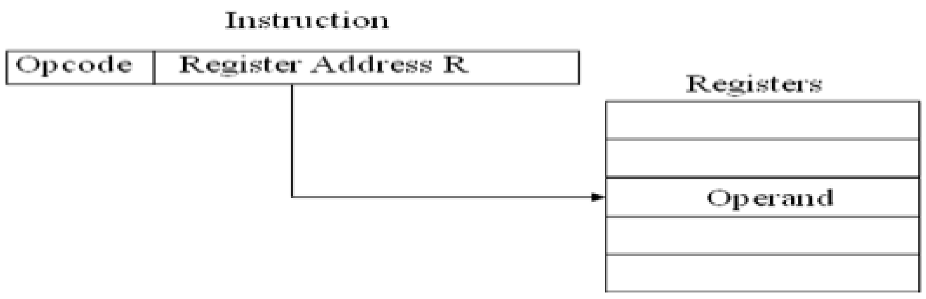
EA=Not defined.

3. **Register Mode**

Address specified in the instruction is the register address.

- Designated operand need to be in a register
- Shorter address than the memory address
- Saving address field in the instruction
- Faster to acquire an operand than the memory addressing

EA = IR(R) (IR(R): Register field of IR)

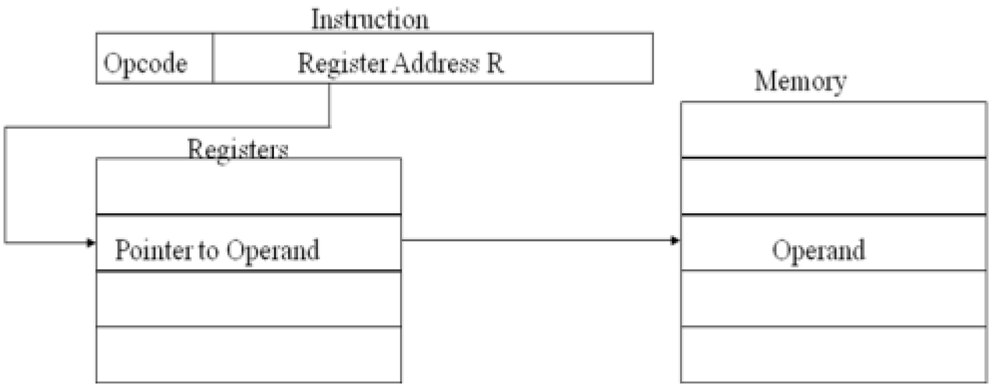


----

4. **Register Indirect Mode**

Instruction specifies a register which contains the memory address of the operand

- Saving instruction bits since register address is shorter than the memory address
- Slower to acquire an operand than both the register addressing or memory addressing
- EA = [IR(R)] ([x]: Content of x)



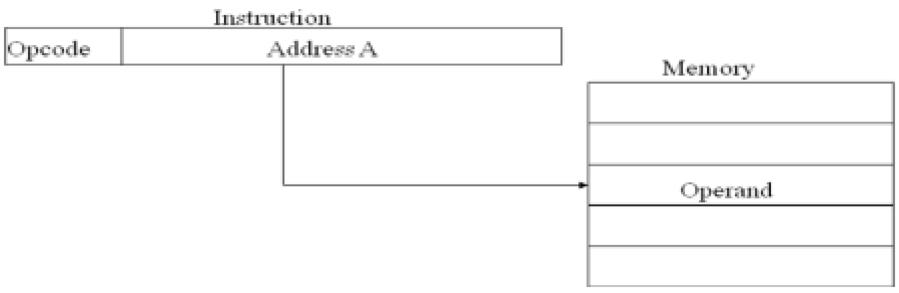
5. **Auto-increment or Auto-decrement features:**

Same as the Register Indirect, but, when the address in the register is used to access memory, the value in the register is incremented or decremented by 1 (after or before the execution of the instruction).

6. **Direct Address Mode**

Instruction specifies the memory address which can be used directly to the physical memory

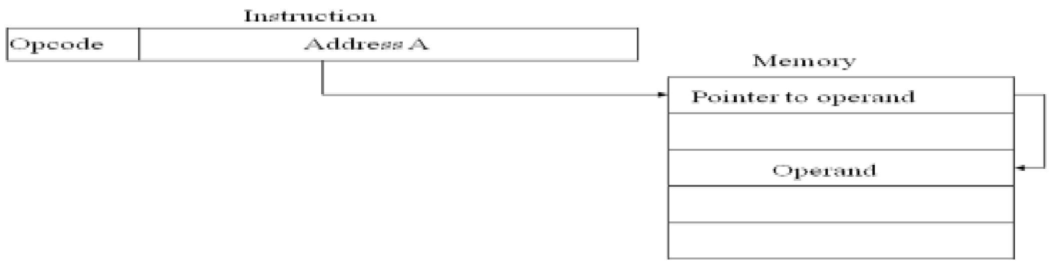
- Faster than the other memory addressing modes
- Too many bits are needed to specify the address for a large physical memory space
- EA = IR(address), (IR(address): address field of IR)



7. **Indirect Addressing Mode**

The address field of an instruction specifies the address of a memory location that contains the address of the operand

- When the abbreviated address is used, large physical memory can be addressed with a relatively small number of bits
- Slow to acquire an operand because of an additional memory access
- EA = M[IR(address)]



8. **Displacement Addressing Mode**

The Address fields of an instruction specifies the part of the address (abbreviated address) which can be used along with a designated register to calculate the address of the operand

a) PC Relative Addressing Mode(R = PC)

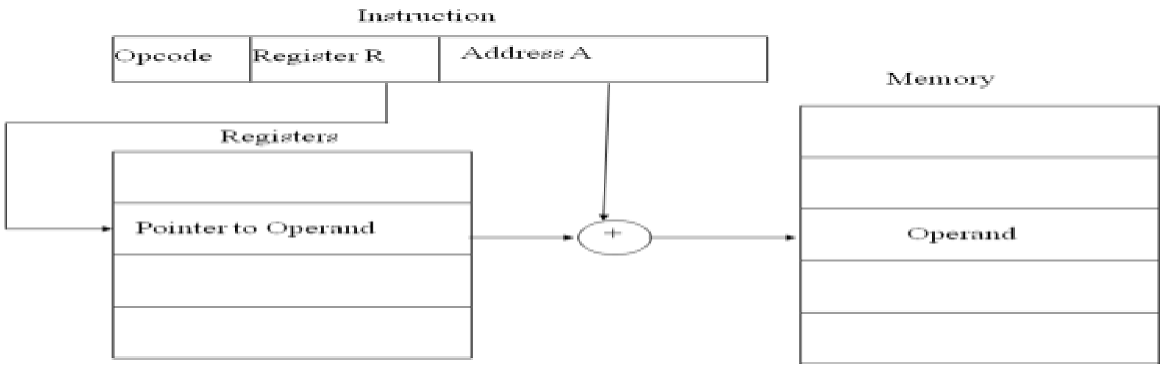
- EA = PC + IR(address)
  - Address field of the instruction is short
  - Large physical memory can be accessed with a small number of address bits

b) Indexed Addressing Mode

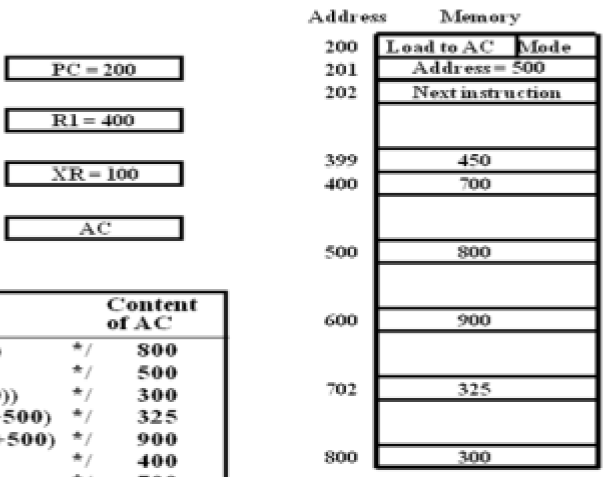
- XR : Index Register:
  - EA = XR + IR(address)

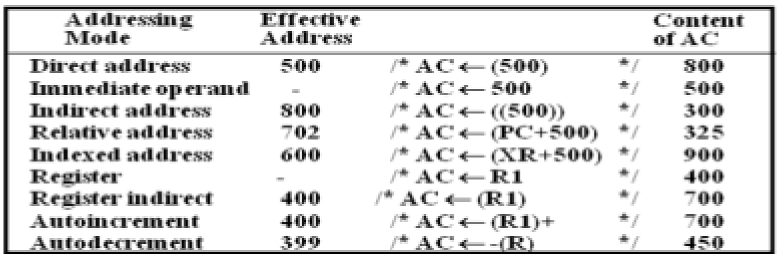
c) Base Register Addressing Mode

BAR: Base Address Register:
- EA = BAR + IR(address)



Numerical Example:





Addressing Mode	Effective Address			Content of AC
Direct address	500	/* AC ← (500)	*/	800
Immediate operand	-	/* AC ← 500	*/	500
Indirect address	800	/* AC ← ((500))	*/	300
Relative address	702	/* AC ← (PC+500)	*/	325
Indexed address	600	/* AC ← (XR+500)	*/	900
Register	-	/* AC ← R1	*/	400
Register indirect	400	/* AC ← (R1)	*/	700
Autoincrement	400	/* AC ← (R1)+	*/	700
Autodecrement	399	/* AC ← -(R)	*/	450

Data Transfer & Manipulation

Computer provides an extensive set of instructions to give the user the flexibility to carryout various computational tasks. Most computer instruction can be classified into three categories.

(1) **Data transfer instruction**
(2) **Data manipulation instruction**
(3) **Program control instruction**

Data transfer instruction cause transferred data from one location to another without changing the binary instruction 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.

(1) **Data Transfer Instruction**

Data transfer instruction move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processes registers, between processes register & input or output, and between processes register themselves

(Typical data transfer instruction)

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

Data Manipulation Instruction

It performs operations on data and provides the computational capabilities for the computer. The data manipulation instructions in a typical computer are usually divided into three basic types.

(a) Arithmetic Instruction
(b) Logical bit manipulation Instruction
(c) Shift Instruction.

_(a)_Arithmetic Instruction

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	Add
Subtract	Sub
Multiply	MUL
Divide	DIV
Add with Carry	ADDC
Subtract with Basses	SUBB
Negate (2's Complement)	NEG

(b) Logical & Bit Manipulation Instruction

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-Or	XOR
Clear Carry	CLRC
Set Carry	SETC
Complement Carry	COMC
Enable Interrupt	ET
Disable Interrupt	OI

(c) Shift Instruction

Instructions to shift the content of an operand are quite useful and one often provided in several variations. Shifts are operation in which the bits of a word are moved to the left or right. The bit-shifted in at the and of the word determines the type of shift used. Shift instruction may specify either logical shift, arithmetic shifts, or rotate type shifts.

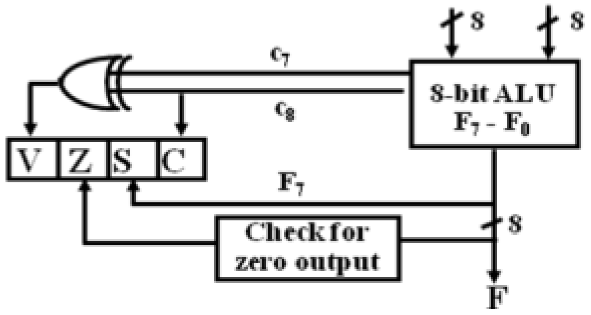
Name	Mnemonic
Logical Shift right	SHR
Logical Shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate mgmt through carry	RORC
Rotate left through carry	ROLC

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

Status Bit Conditions

It is sometimes convenient to supplement the ALU circuit in the CPU with a status register where status b it conditions can be stored for further analysis. Status bits are also called condition-code bits or flag bits. 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 cany C8 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? 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.

Conditional Branch Instructions

Some computers consider the C bit to be a borrow bit after a subtraction operation A — B. A Borrow does not occur if A ^ 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.
- 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.
- 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
(2) Control is transferred to the beginning of the subroutine. 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.

- A subroutine call is implemented with the following micro operations:

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 Micro operations:

PC←M [SP] Pop stack and transfer to PC

SP ←SP + 1 Increment stack pointer

Program Interrupt:

- 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.

(3) An interrupt procedure usually stores all the information

- 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.

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.

- 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.

- 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.

RISC (Reduced Instruction Set Computers)

It is a CPU design strategy based on the insight that simplified (as opposed to complex) instructions can provide higher performance if this simplicity enables much faster execution of each instruction. A computer based on this strategy is a **reduced instruction set computer** (also **RISC**). There are many proposals for precise definitions, but the term is slowly being replaced by the more descriptive **load-store architecture**. Well known RISC families include, and SPARC. Some aspects attributed to the first RISC-*labeled* designs around 1975 include the observations that the memory-restricted compilers of the time were often unable to take advantage of features intended to facilitate *manual* assembly coding, and that complex addressing modes take many cycles to perform due to the required additional memory accesses. It was argued that such functions would be better performed by sequences of simpler instructions if this could yield implementations small enough to leave room for many registers, reducing the number of slow memory accesses. In these simple designs, most instructions are of uniform length and similar structure, arithmetic operations are restricted to CPU registers and only separate *load* and *store* instructions access memory. These properties enable a better balancing of pipeline stages than before, making RISC pipelines significantly more efficient and allowing.

Typical characteristics of RISC:

- Relatively few instructions
- Relatively few addressing modes
- Memory access limited to load and store instructions
- All operations done within the registers of the CPU
- Fixed-length, easily decoded instruction format
- Single-cycle instruction format
- Hardwired rather than microprogrammed control

More RISC characteristics:

- A relatively large numbers of registers in the processor unit.
- Efficient instruction pipeline
- Compiler support: provides efficient translation of high-level language programs into machine language programs.

Advantages of RISC:

- - VLSI Realization
- - Computing Speed
- - Design Costs and Reliability
- - High Level Language Suppor
