# Unit # 2 Microprogrammed Control

**Dr. Rajesh Tiwari Professor ( CSE – AIML) CMREC, Hyderabad, Telangana**

**Control Memory**

* The control unit initiates a series of sequential steps of microoperations.
* During any given time, certain microoperations are to be initiated, while others remain idle.
* The control variables at any given time can be represented by a

string of 1's and 0's called a **control word**.

* As such, control words can be programmed to perform various operations on the components of the system.
* A control unit whose binary control variables are stored in memory is called a microprogrammed control unit .
* Each word in control memory contains within it a microinstruction .
* The microinstruction specifies one or more microoperations for the system.
* A sequence of microinstructions constitutes a microprogram.
* Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read- only memory (ROM).
* Dynamic microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk.
* Control units that use dynamic microprogramming employ a writable control memory.
* This type of memory can be used for writing (to change the microprogram) but is used mostly for reading.

#### A memory that is part of a control unit is referred to as a

**control memory.**

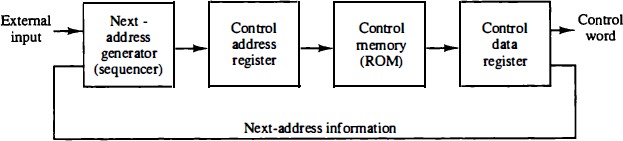
* + A computer that employs a microprogrammed control unit will have two separate memories: a **main memory** and a **control memory**.
  + The main memory is available to the user for storing the programs.
  + The contents of main memory may alter when the data are manipulated and every time that the program is changed.
  + The user's program in main memory consists of machine instructions and data.
  + The control memory holds a fixed microprogram that cannot be altered by

the occasional user.

* + The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations.
  + Each machine instruction initiates a series of microinstructions in control

memory.

* + These microinstructions generate the microoperations to fetch the instruction from main memory; to evaluate the effective address, to execute the operation specified by the instruction, and to return control to the fetch phase in order to repeat the cycle for the next instruction.



**Fig. 2.1: Microprogrammed control organization**

* The general configuration of a microprogrammed control unit is demonstrated in the block diagram of Fig. 2-1.
* The control memory is assumed to be a ROM, within which all control information is permanently stored.
* The control memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory.
* The microinstruction contains a control word that specifies one or more microoperations for the data processor.
* Once these operations are executed, the control must determine the next address.
* The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory.

# Control Memory

* + It is necessary to use some bits of the present microinstruction to control the generation of the address of the next microinstruction.
  + The next address may also be a function of external input

conditions.

* + While the microoperations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction.
  + Thus a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.

## Control Memory

* The next address generator is sometimes called a microprogram **sequencer**, as it determines the address sequence that is read from control memory.
* The address of the next microinstruction can be specified in several ways, depending on the sequencer inputs.
* The control data register holds the present microinstruction while

the next address is computed and read from memory.

* 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.

## Address Sequencing

* + 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 hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another.
  + An initial address is loaded into the control address register when power is turned on in the computer.
  + This address is usually the address of the first microinstruction

that activates the instruction fetch routine.

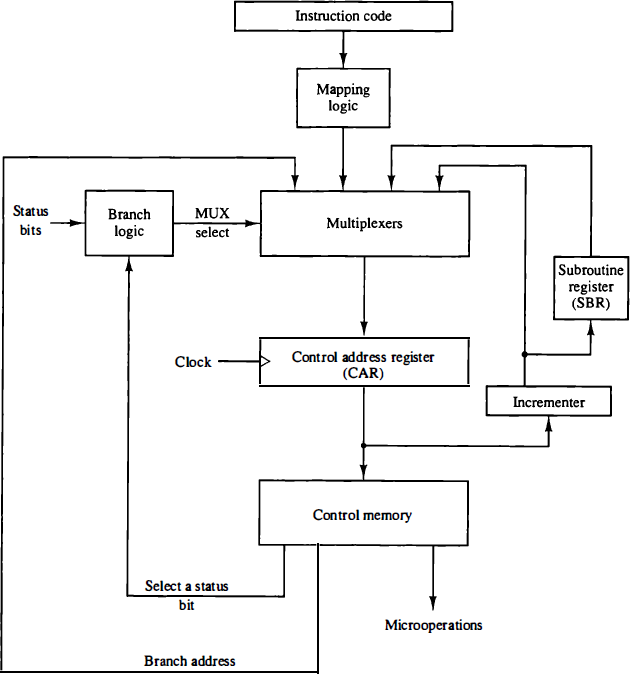
* + The fetch routine may be sequenced by incrementing the control

address register through the rest of its microinstructions.

* + At the end of the fetch routine, the instruction is in the instruction register of the computer.
  + *The control memory next must go through the routine that determines the effective address of the operand.*
  + A machine instruction may have bits that specify various addressing modes, such as indirect address and index registers.
  + The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction.
  + *When the effective address computation routine is completed, the address of the operand is available in the memory address register.*
  + The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a mapping process.
  + A mapping procedure is a rule that transforms the instruction code into a control memory address.

## Address Sequencing

* + When the execution of the instruction is completed, control must return to the fetch routine.
  + This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine.
  + 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.



**Fig. 2.2: Selection of address for control memory.**

* + Fig. 2.2 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 and other bits to specify the method by which the next address is obtained.
  + The diagram shows four different paths from which the control address register (CAR) receives the address.
  + The incrementer increments the content of the control address register by one, to select the next microinstruction in sequence.
  + Branching is achieved by specifying the branch address in one of the fields of the microinstruction.
  + Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition.
  + An external address is transferred into control memory via a mapping logic circuit.
  + The return address for a subroutine is stored in a special register whose value is then used when the microprogram wishes to return from the subroutine.

## Conditional Branching

* + The branch logic of Fig. 2.2 provides decision-making capabilities in the control unit.
  + The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions.
  + Information in these bits can be tested and actions initiated based on

their condition: whether their value is 1 or 0.

* + 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 in a variety of ways.

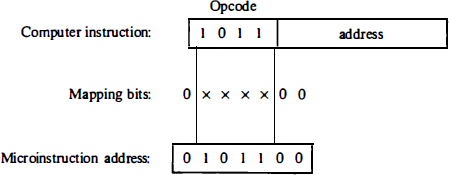
### Mapping of Instruction

* + 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.

* + For example, a computer with a simple instruction format as shown in Fig. 2.3 has an operation code of four bits which can specify up to 16 distinct instructions.
  + Assume further that the control memory has 128 words, requiring an address of seven bits.
  + For each operation code there exists a microprogram routine in control memory that executes the instruction.

## Mapping of Instruction



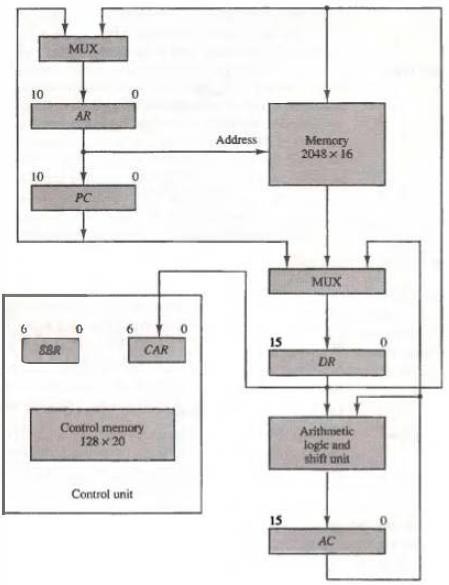
**Fig. 2.3: Mapping from instruction code to microinstruction address**

## Subroutines

* + Subroutines are programs that are used by other routines to accomplish a particular task.
  + A subroutine can be called from any point within the main

body of the microprogram.

* + Frequently, many microprograms contain identical sections of code.
  + Microinstructions can be saved by employing subroutines that use common sections of microcode.
  + When 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 code generation is called **microprogramming.**
  + The block diagram of the computer is shown in Fig. 2.4.
  + It consists of two memory units: a main memory for storing instructions and data, and a control memory for storing the microprogram.
  + Four registers are associated with the processor unit and two with the control unit.
  + The processor registers are program counter PC, address register AR, data register DR, and accumulator register AC.
  + The control unit has a control address register CAR and a subroutine register SBR.



**Fig. 2.4: Computer H/W Configuration**

* + Transfer of information among the registers in the processor is done through multiplexers rather than a common bus.
  + DR can receive information from AC, PC, or memory.
  + AR can receive information from PC or DR.
  + PC can receive information only from AR.
  + The arithmetic, logic, and shift unit performs microoperations with data from AC and DR and places the result in AC .

#### Memory receives its address from AR .

* + Input data written to memory come from DR , and data read from

memory can go only to DR .

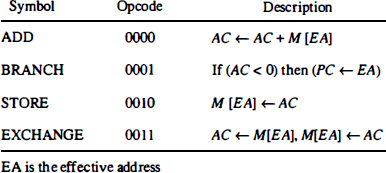
* The computer instruction format is shown in Fig. 2.5(a).
* It consists of three fields: a 1-bit field for indirect addressing symbolized by I, a 4-bit operation code (opcode), and an 11-bit address field.



**Fig. 2.5 (a): Instruction Format**

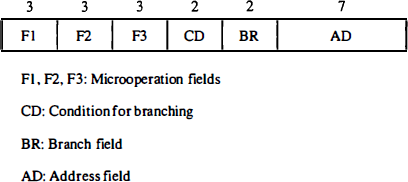
* + Fig. 2.5(b) lists four of the 16 possible memory-reference instructions.
  + The ADD instruction adds the content of the operand found in the effective address to the content of AC .
  + The BRANCH instruction causes a branch to the effective address if the operand in AC is negative.
  + The program proceeds with the next consecutive instruction if AC is not negative.
  + The AC is negative if its sign bit (the bit in the leftmost position of the register) is a 1.
  + The STORE instruction transfers the content of AC into the memory word specified by the effective address.
  + The EXCHANGE instruction swaps the data between AC and the memory word specified by the effective address.

# Computer Instruction



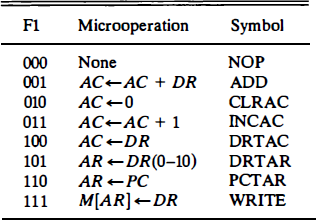
**Fig. 2.5 (b) : Four computer instructions**

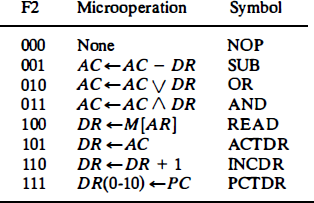
* The microinstruction format for the control memory is shown in Fig. 2.6 .
* The 20 bits of the microinstruction are divided into four functional parts.
* The three fields F1, F2, and F3 specify microoperations for the computer.
* The CD field selects status bit conditions.
* The BR field specifies the type of branch to be used.
* The AD field contains a branch address.
* The address field is seven bits wide, since the control memory has 128 = 27 words.

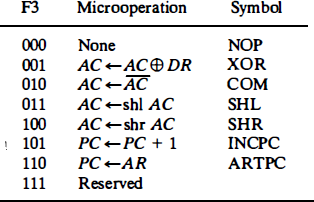


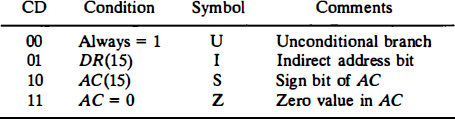
**Fig. 2.6 : Microinstruction code format (20 bits)**

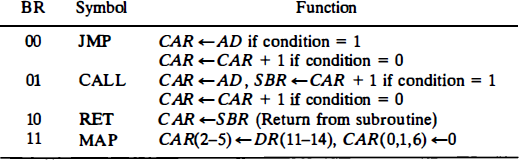
**Symbols and Binary Code for Microinstruction Fields**











* + The microoperations are subdivided into three fields of three bits each.
  + The three bits in each field are encoded to specify seven distinct microoperations as listed in Tables .
  + This gives a total of 21 microoperations.
  + 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.

Example , a microinstruction can specify two simultaneous microoperations from F2 and F3 and none from F1.

DR <- M [AR] with F2 = 100

and PC <- PC + 1 with F3 = 101

* + The nine bits of the microoperation fields will then be 000 100 101.
  + It is important to realize that two or more conflicting microoperations cannot be specified simultaneously.
  + For example, a microoperation field 010 001 000 has no meaning because it specifies the operations to clear AC to 0 and

subtract DR from AC at the same time.

* + Each microoperation in Tables is defined with a register transfer statement and is assigned a symbol for use in a symbolic microprogram.
  + All transfer-type microoperations symbols use five letters.
  + The first two letters designate the source register, the third letter is always a T, and the last two letters designate the destination register.
  + For example, the microoperation that specifies the transfer AC

<- DR (F1 = 100) has the symbol DRTAC, which stands for a transfer from DR to AC .

# Microinstruction Format (CD)

* The CD (condition) field consists of two bits which are encoded to specify four status bit conditions as listed in Table.
* The first condition is always a 1, so that a reference to CD = 00 (or the symbol U) will always find the condition to be true.
* When this condition is used in conjunction with the BR (branch) field, it provides an unconditional branch operation.
* The indirect bit I is available from bit 15 of DR after an instruction

is read from memory.

* The sign bit of AC provides the next status bit.
* The zero value, symbolized by Z, is a binary variable whose value is equal to 1 if all the bits in AC are equal to zero.
* We will use the symbols U, I, S, and Z for the four status bits when we write microprograms in symbolic form.

## Microinstruction Format (BR)

* The BR (branch) field consists of two bits.
* It is used, in conjunction with the address field AD, to choose the address of the next microinstruction.
* As shown in Table, when BR = 00, the control performs a jump (JMP) operation (which is similar to a branch), and when BR = 01, it performs a call to subroutine (CALL) operation.
* The two operations are identical except that a call microinstruction stores the return address in the subroutine register SBR.
* The jump and call operations depend on the value of the CD field.
* If the status bit condition specified in the CD field is equal to 1, the next address in the AD field is transferred to the control address register CAR.
* Otherwise, CAR is incremented by 1 .

**Design of Control Unit**

* + The control memory output of each subfield must be decoded to provide the distinct microoperations.
  + The outputs of the decoders are connected to the appropriate inputs in the processor unit.

**Design of Control Unit**

* Fig. 2.7 shows the three decoders and some of the connections that must be made from their outputs.
* Each of the three fields of the microinstruction presently available in the output

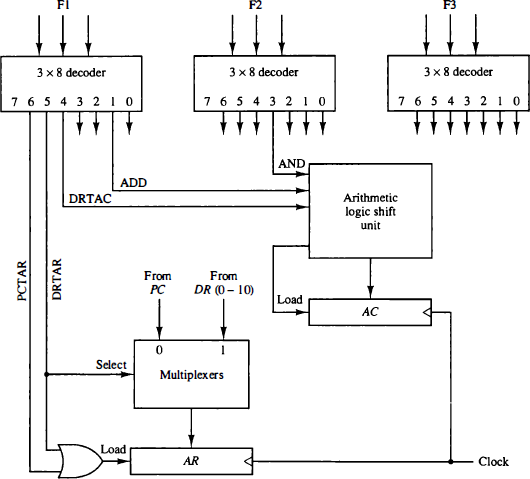
of control memory are decoded with a 3 x 8 decoder to provide eight outputs.

* Each of these outputs must be connected to the proper circuit to initiate the corresponding microoperation as specified in previous Tables.
* For example, when F 1 = 101 (binary 5), the next clock pulse transition transfers the content of DR(0-10) to AR (symbolized by DRTAR).
* Similarly, when F1 = 1 1 0 (binary 6) there is a transfer from PC to AR (symbolized by PCTAR). As shown in Fig. 2.7, outputs 5 and 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 multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive.
* The transfer into AR occurs with a clock pulse transition only when output 5 or

output 6 of the decoder are active.

* The other outputs of the decoders that initiate transfers between registers must be connected in a similar fashion.

**Design of Control Unit**



**Fig. 2.7: Decoding of microoperation fields**

**Design of Control Unit (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.
  + There are large ROM units available in integrated circuit packages, so are general-purpose sequencers suited for the construction of microprogram control units.
  + *The role of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed.*
  + The next-address logic of the sequencer determines the specific address source to be loaded into the control address register.
  + The choice of the address source is guided by the next-address information bits that the sequencer receives from the present microinstruction.
  + Commercial sequencers include within the unit an internal register stack used for temporary storage of addresses during microprogram looping and subroutine calls.
  + Some sequencers provide an output register which can function as the address register for the control memory.
* Block diagram of the microprogram sequencer is shown in Fig. 2.8.

•

* The control memory is included in the diagram to show the

interaction between the sequencer and the memory attached to it.

* There are two multiplexers in the circuit.
* The first multiplexer selects an address from one of four sources and routes it into a control address register CAR .
* The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit.
* The output from CAR provides the address for the control memory.
* The content of CAR is incremented and applied to one of the multiplexer inputs and to the subroutine register SBR .
* The other three inputs to multiplexer number 1 come from the address field of the present microinstruction, from the output of SBR, and from an external source that maps the instruction.
* 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 number of subroutines can be active at the same time.
* A push and pop operation, in conjunction with a stack pointer, stores and

retrieves the return address during the call and return microinstructions.



**Fig. 2.8: Microprogram sequencer for a control memory**

* + The CD (condition) field of the microinstruction selectgs one of the status bits in the second multiplexer.
  + If the bit selected is equal to 1, the T (test) variable is equal to

1; otherwise, it is equal to 0.

* + The T value together with the two bits from the BR (branch) field go to an input logic circuit.
  + 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.
  + With three inputs, the sequencer can provide up to eight address sequencing operations.
  + The input logic circuit in Fig. 2.8 has three inputs, I0, I1, and T, and three outputs, S0, S1, and L.
  + Variables So and S1, select one of the source addresses for CAR .
  + Variable L enables the load input in SBR .
  + The binary values of the two selection variables determine the path in the multiplexer.
  + For example, with S1 So = 10, multiplexer input number 2 is selected and establishes a transfer path from SBR to CAR.
  + Each of the four inputs as well as the output of MUX 1 contains a 7-bit address.

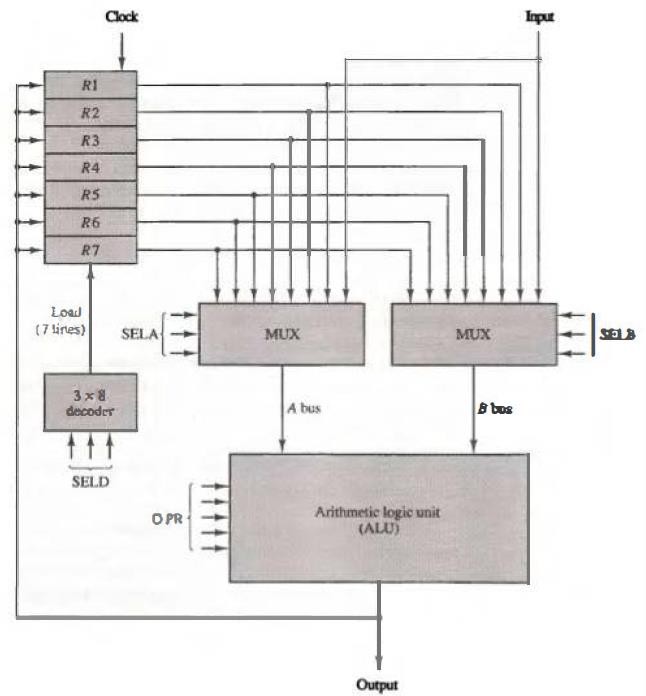
# Central Processing Unit

## General Register Organization

* + A bus organization for seven CPU registers is shown in Fig. 2.9.
  + 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

microoperation that is to be performed.

* + The result of the microoperation is available for output data and also goes into the inputs of all the registers.
  + The register that receives the information from the output bus is selected by a decoder.
  + The decoder activates one of the register load inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.



**(b) : Control word**

* + 1. **: Block diagram**

**Fig. 2.9: Register set with common ALU**

* 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

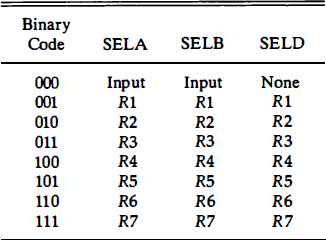
R 1 <--R2 + R3

* the control must provide binary selection variables to the following

selector inputs:

* + MUX A selector (SELA): to place the content of R2 into bus A .
  + MUX B selector (SELB): to place the content o f R 3 into bus B .
  + ALU operation selector (OPR): to provide the arithmetic addition A + B .
  + Decoder destination selector (SELD): t o transfer the content of the output bus into R 1 .
* There are 14 binary selection inputs in the unit, and their combined value specifies a **control word**.
* The 14-bit control word is defined in Fig. 2.9 (b).
* It consists of four fields.
  + Three fields contain three bits each, and one field has five bits.
  + The three bits of SELA select a source register for the A input of the ALU.
  + The three bits of SELB select a register for the B input of the ALU.
  + The three bits of SELD select a destination register using the decoder and its seven load outputs.
  + The five bits of OPR select one of the operations in the ALU.
* The 14-bit control word when applied to the selection inputs specify a particular microoperation.
* The encoding of the register selections is specified in next Table.
* The 3-bit binary code listed in the first column of the table specifies the binary code for each of the three fields.
* The register selected by fields SELA, SELB, and SELD is the one whose decimal number is equivalent to the binary number in the code.
* When SELA or SELB is 000, the corresponding multiplexer selects the external input data.
* When SELD = 000, no destination register is selected but the contents of the output bus are available in the external output.

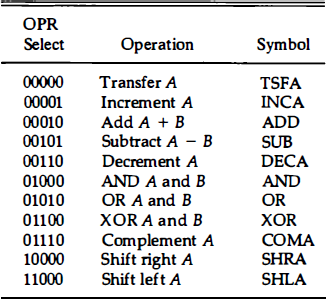
**Table : Encoding of Register Selection Fields**



* + The ALU provides arithmetic and logic operations.
  + In addition, the CPU must provide shift operations.
  + The shifter may be placed in the input of the ALU to provide a preshift capability, or at the output of the ALU to provide postshifting capability.
  + In some cases, the shift operations are included with the ALU.
  + The encoding of the ALU operations for the CPU is shown in next Table.
  + The OPR field has five bits and each operation is designated with a

symbolic name.

**Table: Encoding of ALU Operations**



#### Examples of Microoperations

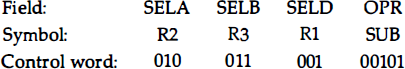
* A control word of 14 bits is needed to specify a microoperation in the CPU.
* The control word for a given microoperation can be derived

from the selection variables.

* For example, the subtract microoperation given by the statement

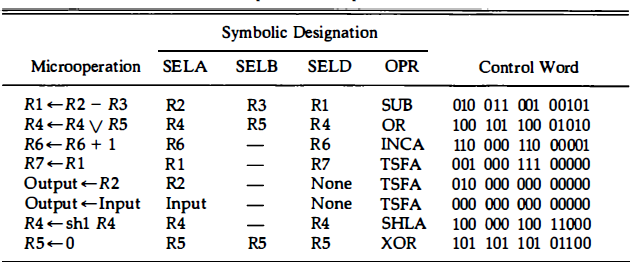
R 1 <- R 2 - R3

* 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.
  + Thus the control word is specified by the four fields and the corresponding binary value for each field is obtained.
  + The binary control word for the subtract microoperation is 010 011 001 00101 and is obtained as follows:



* + The control word for this microoperation and a few others are listed in next Table.

**TABLE: Examples of Microoperations for the CPU**



**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.

* Fig. 2.10 shows the organization of a 64-word register stack.
* The stack pointer register SP contains a binary number whose value is

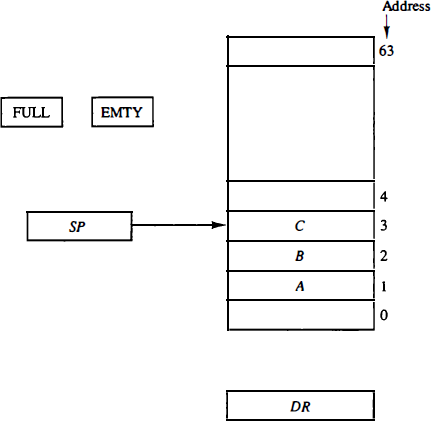
equal to the address of the word that is currently on top of the stack.

* 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 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.

### Register Stack



**Figure 2.10: Block diagram of a 64-word stack**

* + The format of an instruction is usually expresed 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:
* An operation code field that specifies the operation to be performed.
* An address field that designates a memory address or a processor register.
* A mode field that specifies the way the operand or the effective address is determined.
  + 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 2k registers in the CPU.

* + Thus a CPU with 16 processor registers R0 through R15 will have a register address field of four bits.
  + The binary number 0101, for example, will designate register R5.

# Instruction Formats

* + 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:
* Single accumulator organization.
* General register organization.
* Stack organization.
  + An example of an accumulator-type organization is presented as:
  + 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 are:
  + The instruction format in this type of computer needs three register address fields.
  + Thus the instruction for an arithmetic addition may be written in an assembly language as

ADD R1 , R2 , R3

* + to denote the operation R 1 <--- R2 + R 3 .
  + The number o f address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers.
  + Thus the instruction

ADD R1 , R2

* + would denote the operation R 1 <--- R 1 + R2.
  + Only register addresses for R 1 and R2 need be specified in this instruction.

# Instruction Formats

* + The stack-organized CPU is as:
  + 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.

### Instruction Type

* Three-Address Instructions
* Two-Address Instructions
* One-Address Instructions
* Zero-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)

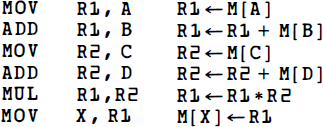
\*(C + D) is shown below, together with comments that explain the register transfer operation of each instruction.

|  |  |  |
| --- | --- | --- |
| 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, 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:



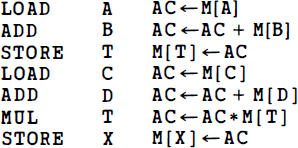
* + The MOV instruction moves or transfers the operands to and from memory and processor registers.
  + The first symbol listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.

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.

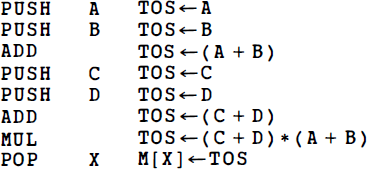
* However, here we will neglect the second register and assume that the AC contains the result of all operations.
* The program to evaluate X = (A + B) \* (C + D) is:



* + 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.)



* + 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 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:
    - 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.
    - To reduce the number of bits in the addressing field of the instruction.

# Addressing Modes

* + The control unit of a computer is designed to go through an instruction cycle that is divided into three major phases:
    - Fetch the instruction from memory.
    - Decode the instruction.
    - Execute the instruction.

## Addressing Modes

* 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 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.
  + Most addressing modes **modify the address field of the instruction**, there are two modes that need no address field at all.
  + These are :
    - Implied modes and
    - Immediate modes.
* **Implied Mode:** Here 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.
* 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:** Here 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.

* **Register Mode:** Here 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 2k registers.
* **Register Indirect Mode:** Here 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.
* Few **computers incorporate a special mode that automatically** increments or decrements the content of the register after data access.
  + **Direct Address Mode:** Here 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:** Here 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.
* 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:** Here *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.
* 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.
  + **Indexed Addressing Mode:** Here *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.
  + If an indextype instruction does not include an address field in its format, the instruction converts to the register indirect mode of operation.
  + **Base Register Addressing Mode:** Here 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.

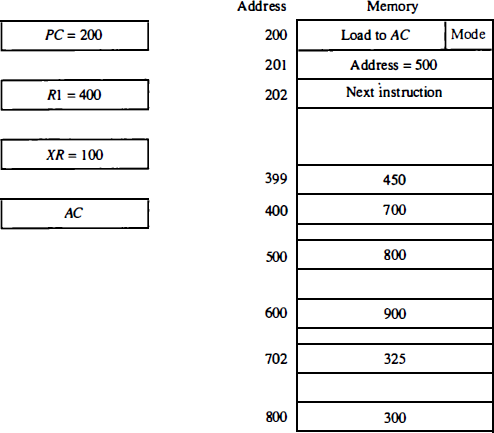
* + The base register addressing mode is used in computers to facilitate the relocation of programs in memory.
  + When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of instructions must reflect this change of position.
  + 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

# Addressing Modes

* + - Fig. 2.11 show the effect of the addressing modes on the instruction defined.
    - The two-word instruction at address 200 and 201 is a "load to

AC" instruction with an address field equal to 500.

* + - The first word of the instruction specifies the operation code and mode, and the second word specifies the address part.
    - PC has the value 200 for fetching this instruction.
    - The content of processor register R 1 is 400, and the content of an index register XR is 100.
    - AC receives the operand after the instruction is executed.



**Fig. 2.11: Numerical example for addressing modes**

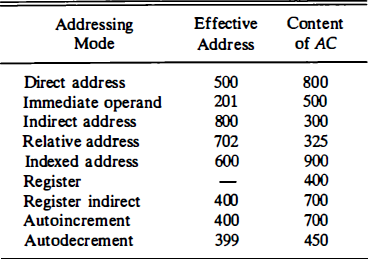
* + - The mode field of the instruction can specify any one of a number of modes.
    - For each possible mode we calculate the effective address and the operand that must be loaded into AC .
    - In the direct address mode the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 800.
    - In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC . (The effective address in this case is 201 . )
    - In the indirect mode the effective address is stored in memory at address 500.
    - Therefore, the effective address is 800 and the operand is 300.
* In the relative mode the effective address is 500 + 202 = 702 and the operand is 325. ( The value in PC after the fetch phase and during the execute phase is 202)
* In the index mode the effective address is XR + 500 = 100 + 500 =

600 and the operand is 900.

* In the register mode the operand is in R1 and 400 is loaded into AC

. (There is no effective address in this case).

* In the register indirect mode the effective address is 400, equal to the content of R1 and the operand loaded into AC is 700.
* The autoincrement mode is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction.
* The autodecrement mode decrements R1 to 399 prior to the execution of the instruction.
* The operand loaded into AC is now 450.
* Below Table lists the values of the effective address and the operand loaded into AC for the nine addressing modes.



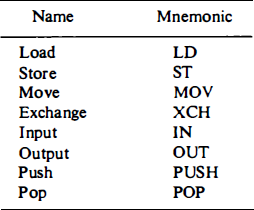
### Data Transfer and Manipulation

* + Most computer instructions can be classified into three categories:
    - Data transfer instructions
    - Data manipulation instructions
    - 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.
  + Next Table gives a list of eight data transfer instructions used

in many computers.



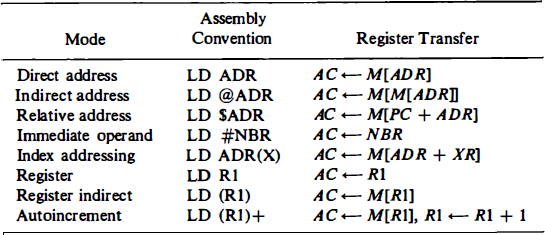
* + The load instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator.
  + The store instruction designates a transfer from a processor register into memory.
  + The move instruction has been used in computers with multiple CPU registers to 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.
  + Next 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, Rl 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.



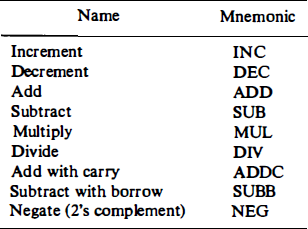
## Data Manipulation Instructions

* + 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:
    - Arithmetic instructions
    - Logical and bit manipulation instructions
    - Shift instructions
  + Each instruction when executed in the computer must go through the fetch phase to read its binary code value from memory.
  + The operands must also be brought into processor registers according to the rules of the instruction addressing mode.
  + The last step is to execute the instruction in the processor.
  + This last step is implemented by means of microoperations or through an ALU and shifter as shown in Fig. 2.9.

# Data Manipulation Instructions

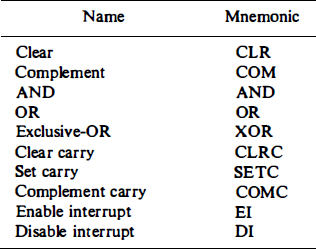
#### Arithmetic Instructions

* + - A list of typical arithmetic instructions is given in next Table.



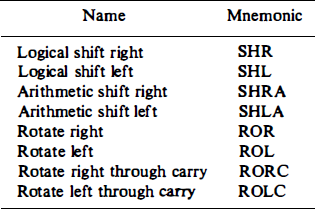
**Data Manipulation Instructions**

* + **Logical and Bit Manipulation Instructions**



# Data Manipulation Instructions

* + Shift Instructions



**Program Control**

