Chapter 4 – Register Transfer and Microoperations

Section 4.1 – Register Transfer Language

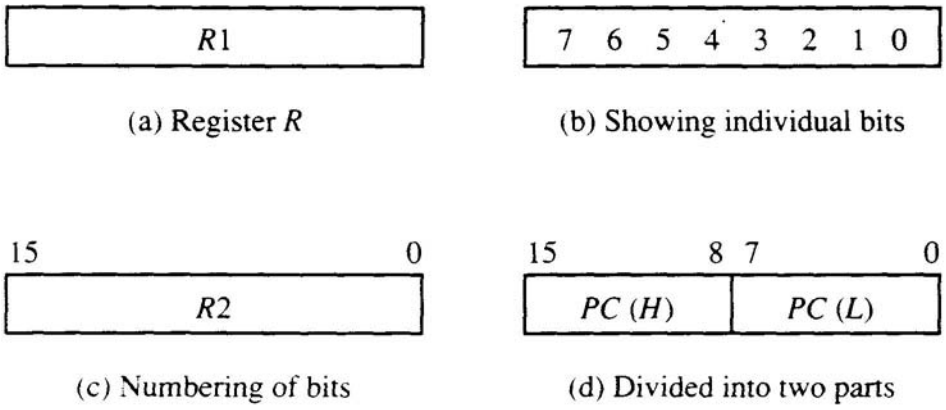
- Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic
- The modules are interconnected with common data and control paths to form a digital computer system
- The operations executed on data stored in registers are called *microoperations*
- A microoperation is an elementary operation performed on the information stored in one or more registers
- Examples are shift, count, clear, and load
- Some of the digital components from before are registers that implement microoperations
- The internal hardware organization of a digital computer is best defined by specifying
    - The set of registers it contains and their functions
    - The sequence of microoperations performed on the binary information stored
    - The control that initiates the sequence of microoperations

- Use symbols, rather than words, to specify the sequence of microoperations
- The symbolic notation used is called a *register transfer language*
- A programming language is a procedure for writing symbols to specify a given computational process
- Define symbols for various types of microoperations and describe associated hardware that can implement the microoperations

Section 4.2 – Register Transfer

- Designate computer registers by capital letters to denote its function
- The register that holds an address for the memory unit is called MAR
- The program counter register is called PC
- IR is the instruction register and R1 is a processor register
- The individual flip-flops in an *n*-bit register are numbered in sequence from 0 to *n*-1
- Refer to Figure 4.1 for the different representations of a register

Figure 4-1 Block diagram of register.



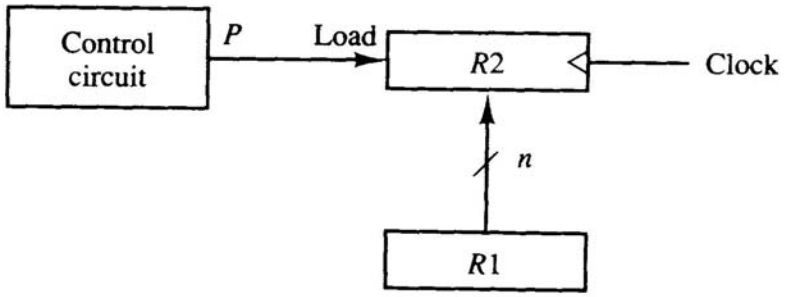
- Designate information transfer from one register to another by
  R2 ← R1

- This statement implies that the hardware is available
  - The outputs of the source must have a path to the inputs of the destination
  - The destination register has a parallel load capability

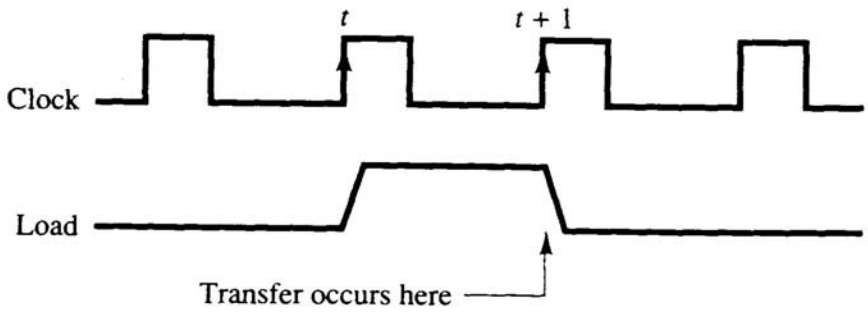
- If the transfer is to occur only under a predetermined control condition, designate it by
  If (P = 1) *then* (R2 ← R1)
  or,
  P: R2 ← R1,
  where P is a control function that can be either 0 or 1

- Every statement written in register transfer notation implies the presence of the required hardware construction

Figure 4-2 Transfer from R1 to R2 when $P = 1$.



(a) Block diagram



(b) Timing diagram

- It is assumed that all transfers occur during a clock edge transition
- All microoperations written on a single line are to be executed at the same time

  T: R2 ← R1, R1 ← R2

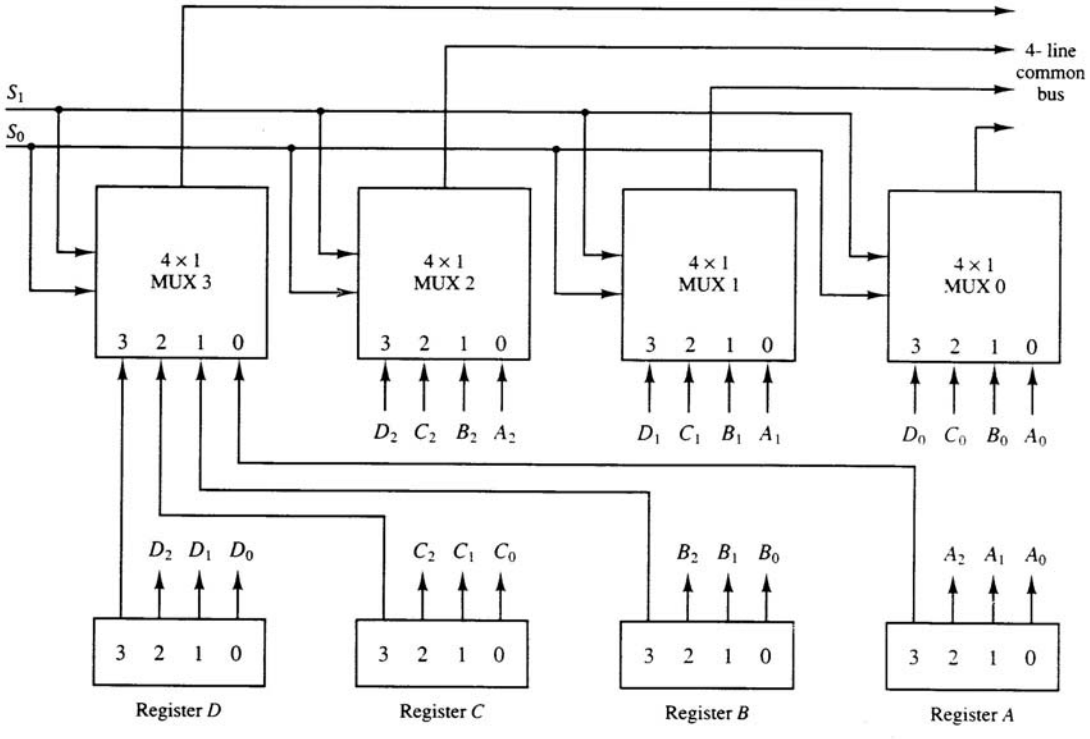
TABLE 4-1 Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*MAR*, *R2*
Parentheses ( )	Denotes a part of a register	*R2*(0–7), *R2*(*L*)
Arrow ←	Denotes transfer of information	*R2* ← *R1*
Comma ,	Separates two microoperations	*R2* ← *R1*, *R1* ← *R2*

Section 4.3 – Bus and Memory Transfers

- Rather than connecting wires between all registers, a common bus is used
- A bus structure consists of a set of common lines, one for each bit of a register
- Control signals determine which register is selected by the bus during each transfer
- Multiplexers can be used to construct a common bus
- Multiplexers select the source register whose binary information is then placed on the bus
- The select lines are connected to the selection inputs of the multiplexers and choose the bits of one register

Figure 4-3 Bus system for four registers.

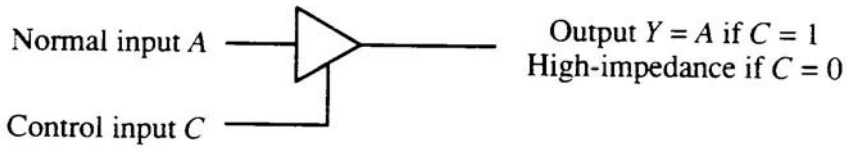


- In general, a bys system will multiplex *k* registers of *n* bits each to produce an *n*-line common bus
- This requires *n* multiplexers – one for each bit
- The size of each multiplexer must be *k* x 1
- The number of select lines required is *log k*
- To transfer information from the bus to a register, the bus lines are connected to the inputs of all destination registers and the corresponding load control line must be activated
- Rather than listing each step as

BUS ← C, R1 ← BUS,
use R1 ← C, since the bus is implied

- Instead of using multiplexers, *three-state gates* can be used to construct the bus system
- A three-state gate is a digital circuit that exhibits three states
- Two of the states are signals equivalent to logic 1 and 0
- The third state is a *high-impedance* state – this behaves like an open circuit, which means the output is disconnected and does not have a logic significance

Figure 4-4 Graphic symbols for three-state buffer.



- The three-state buffer gate has a normal input and a control input which determines the output state
- With control 1, the output equals the normal input
- With control 0, the gate goes to a high-impedance state
- This enables a large number of three-state gate outputs to be connected with wires to form a common bus line without endangering loading effects

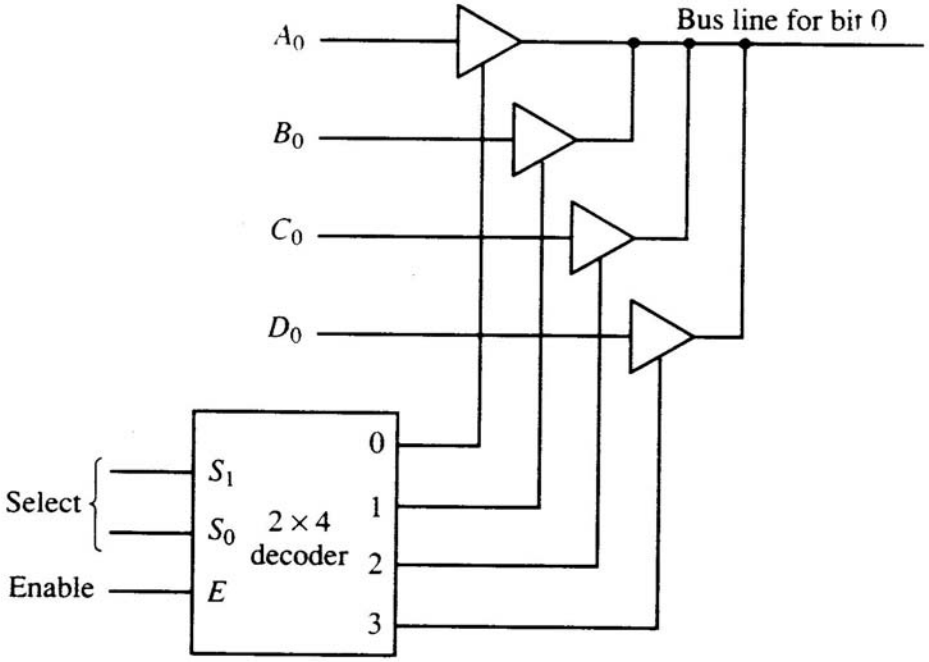


Figure 4-5 Bus line with three state-buffers.

- Decoders are used to ensure that no more than one control input is active at any given time
- This circuit can replace the multiplexer in Figure 4.3
- To construct a common bus for four registers of *n* bits each using three-state buffers, we need *n* circuits with four buffers in each
- Only one decoder is necessary to select between the four registers

- Designate a memory word by the letter M
- It is necessary to specify the address of M when writing memory transfer operations
- Designate the address register by AR and the data register by DR
- The read operation can be stated as:

  Read: DR ← M[AR]
- The write operation can be stated as:

  Write: M[AR] ← R1

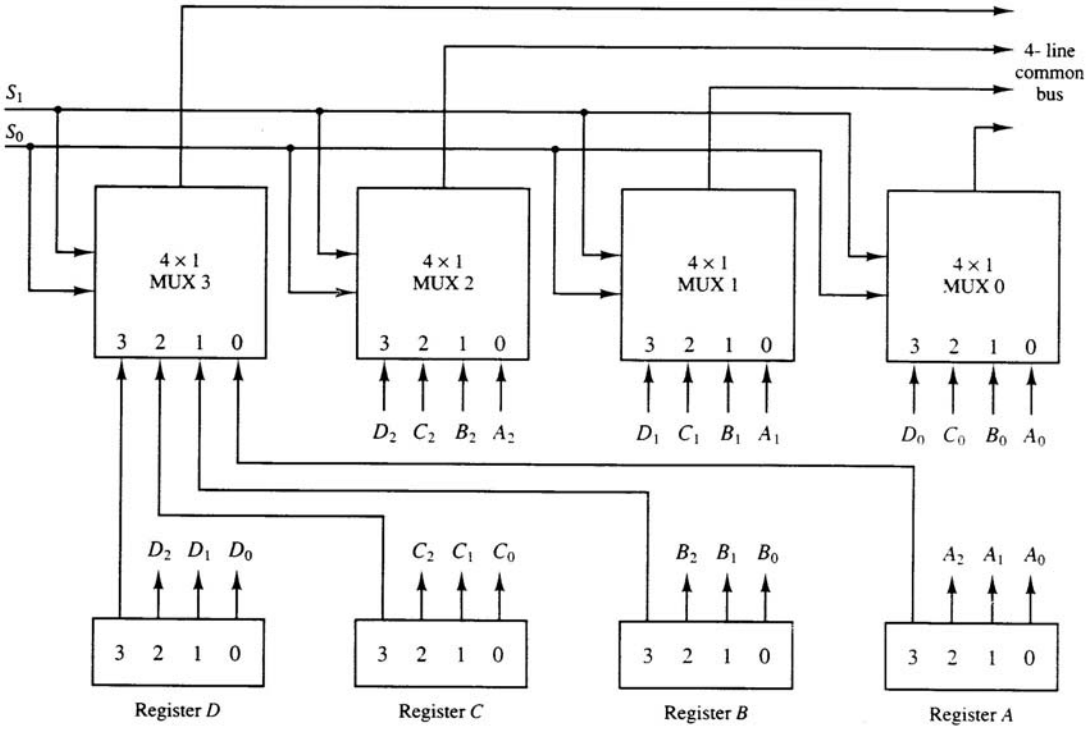
Section 4.4 – Arithmetic Microoperations

- There are four categories of the most common microoperations:
  - Register transfer: transfer binary information from one register to another
  - Arithmetic: perform arithmetic operations on numeric data stored in registers

- Logic: perform bit manipulation operations on non-numeric data stored in registers
- Shift: perform shift operations on data stored in registers

- The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift
- Example of addition: R3 ← R1 +R2
- Subtraction is most often implemented through complementation and addition
- Example of subtraction: R3 ← R1 + $\overline{R2}$ + 1 (strikethrough denotes bar on top – 1's complement of R2)
- Adding 1 to the 1's complement produces the 2's complement
- Adding the contents of R1 to the 2's complement of R2 is equivalent to subtracting

Figure 4-3 Bus system for four registers.



- Multiply and divide are not included as microoperations
- A microoperation is one that can be executed by one clock pulse
- Multiply (divide) is implemented by a sequence of add and shift microoperations (subtract and shift)

- To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the addition
- A full-adder adds two bits and a previous carry

- A *binary adder* is a digital circuit that generates the arithmetic sum of two binary numbers of any length
- A binary added is constructed with full-adder circuits connected in cascade
- An *n*-bit binary adder requires *n* full-adders

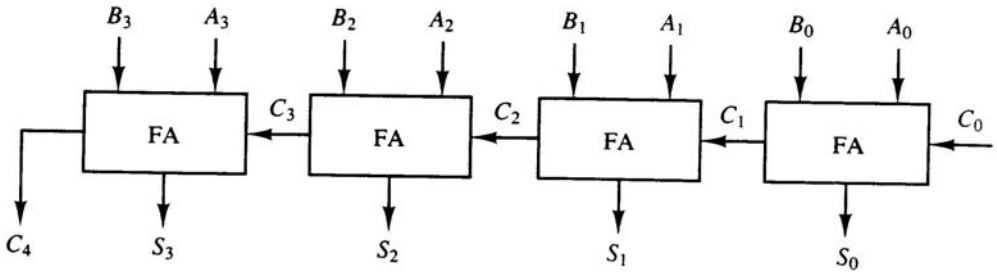


Figure 4-6 4-bit binary adder.

- The subtraction A-B can be carried out by the following steps
  - Take the 1's complement of B (invert each bit)
  - Get the 2's complement by adding 1
  - Add the result to A
- The addition and subtraction operations can be combined into one common circuit by including an XOR gate with each full-adder

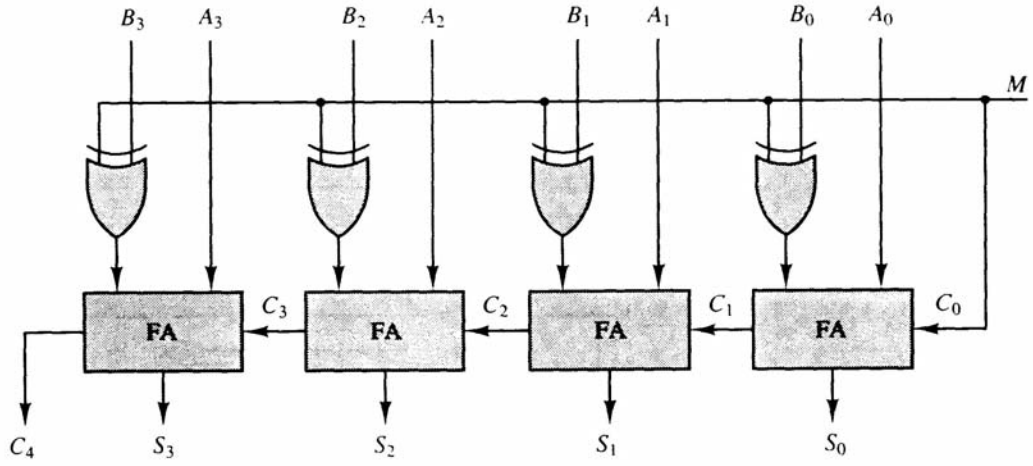


Figure 4-7 4-bit adder-subtractor.

- The increment microoperation adds one to a number in a register
- This can be implemented by using a binary counter – every time the count enable is active, the count is incremented by one
- If the increment is to be performed independent of a particular register, then use half-adders connected in cascade

- An *n*-bit binary incrementer requires *n* half-adders

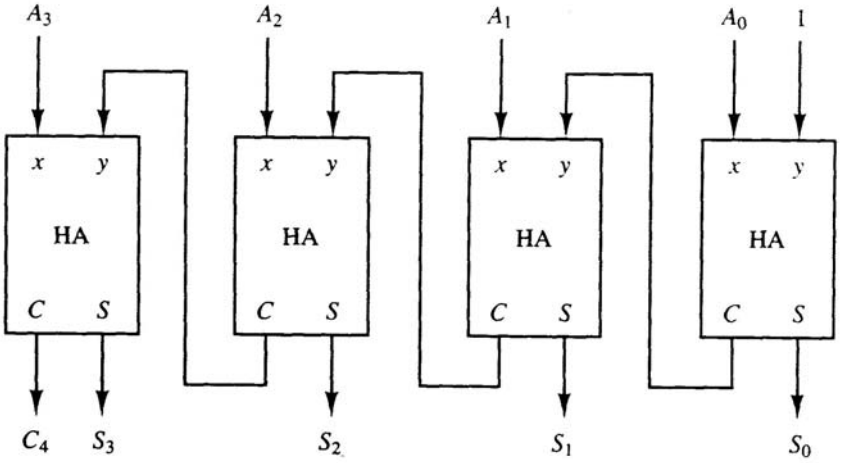


Figure 4-8 4-bit binary incrementer.

- Each of the arithmetic microoperations can be implemented in one composite arithmetic circuit
- The basic component is the parallel adder
- Multiplexers are used to choose between the different operations
- The output of the binary adder is calculated from the following sum:

  $D = A + Y + C_{in}$

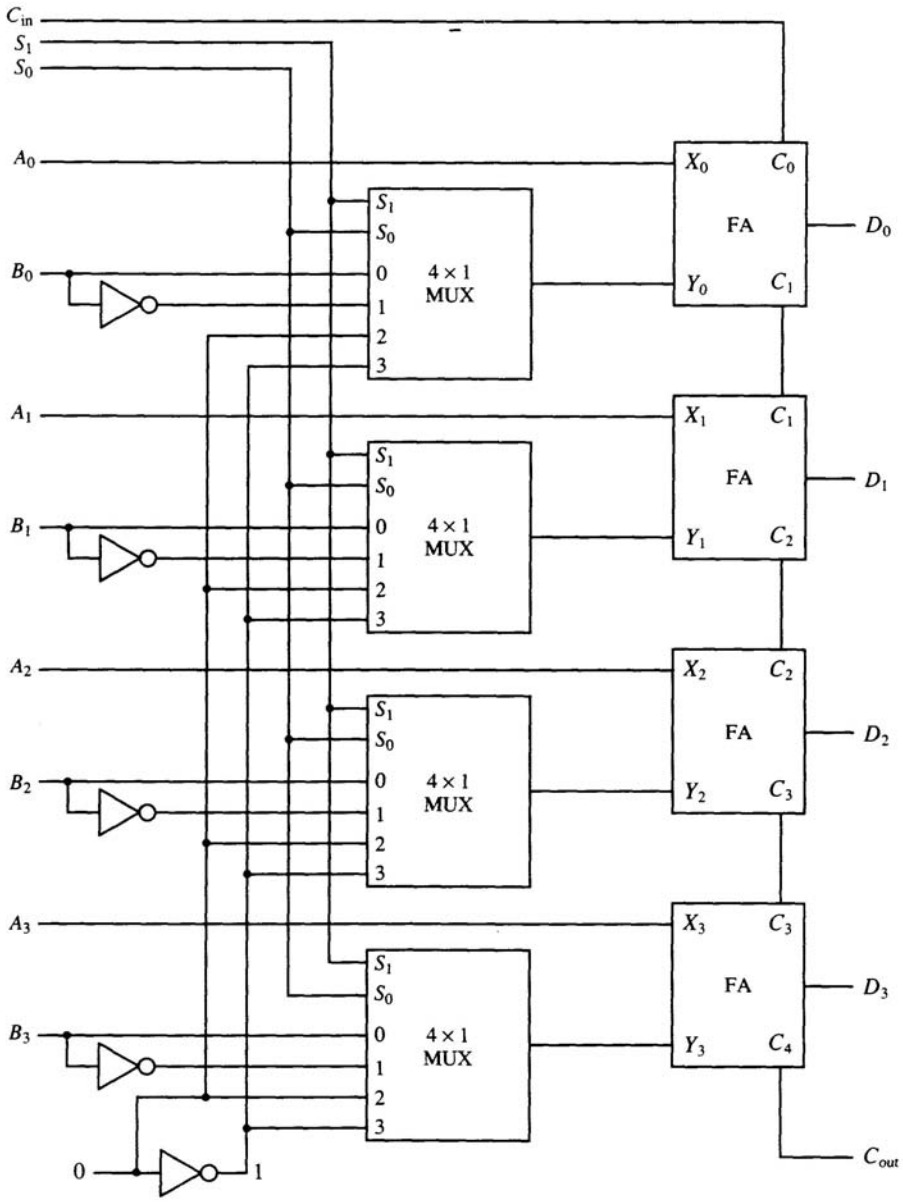


Figure 4-9 4-bit arithmetic circuit.

TABLE 4-4 Arithmetic Circuit Function Table

Select			Input	Output	
S_1	S_0	C_{in}	Y	$D = A + Y + C_{in}$	Microoperation
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	$\overline{B}$	$D = A + \overline{B}$	Subtract with borrow
0	1	1	$\overline{B}$	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

Section 4.5 – Logic Microoperations

- Logic operations specify binary operations for strings of bits stored in registers and treat each bit separately
- Example: the XOR of R1 and R2 is symbolized by

  P: R1 ← R1 ⊕ R2
- Example: R1 = 1010 and R2 = 1100

  1010 Content of R1
  1100 Content of R2
  0110 Content of R1 after P = 1

- Symbols used for logical microoperations:
  - OR: ∨
  - AND: ∧
  - XOR: ⊕

- The + sign has two different meanings: logical OR and summation
- When + is in a microoperation, then summation
- When + is in a control function, then OR
- Example:

  P + Q: R1 ← R2 + R3, R4 ← R5 ∨ R6
- There are 16 different logic operations that can be performed with two binary variables

TABLE 4-5 Truth Tables for 16 Functions of Two Variables

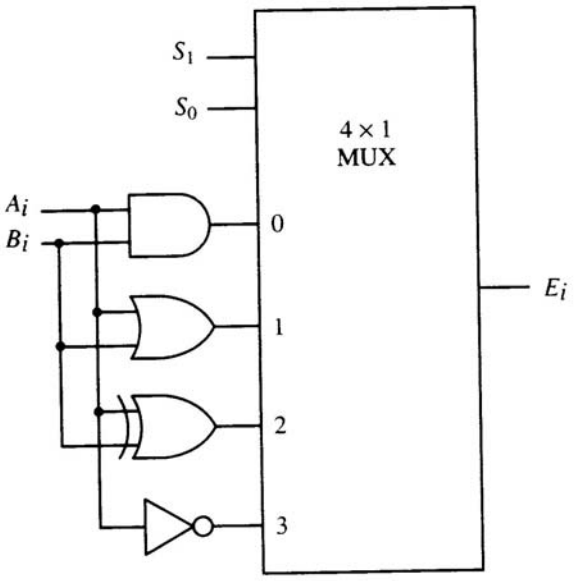
x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

TABLE 4-6 Sixteen Logic Microoperations

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

- The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers
- All 16 microoperations can be derived from using four logic gates

Figure 4-10 One stage of logic circuit.



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Complement

(b) Function table

- Logic microoperations can be used to change bit values, delete a group of bits, or insert new bit values into a register
- The *selective-set* operation sets to 1 the bits in A where there are corresponding 1's in B

  1010 A before
  1100 B (logic operand)
  1110 A after

  $A \leftarrow A \vee B$

- The *selective-complement* operation complements bits in A where there are corresponding 1's in B

  1010 A before
  1100 B (logic operand)
  0110 A after

  $A \leftarrow A \oplus B$

- The *selective-clear* operation clears to 0 the bits in A only where there are corresponding 1's in B

  1010 A before
  1100 B (logic operand)
  0010 A after

$A \leftarrow A \wedge \overline{B}$

- The *mask* operation is similar to the selective-clear operation, except that the bits of A are cleared only where there are corresponding 0's in B

```
1010    A before
1100    B (logic operand)
1000    A after
```

$A \leftarrow A \wedge B$

- The *insert* operation inserts a new value into a group of bits
- This is done by first masking the bits to be replaced and then Oring them with the bits to be inserted

```
0110  1010    A before
0000  1111    B (mask)
0000  1010    A after masking

0000  1010    A before
1001  0000    B (insert)
1001  1010    A after insertion
```

- The *clear* operation compares the bits in A and B and produces an all 0's result if the two number are equal

```
1010    A
1010    B
0000    A ← A ⊕ B
```

Section 4.6 – Shift Microoperations

- Shift microoperations are used for serial transfer of data
- They are also used in conjunction with arithmetic, logic, and other data-processing operations
- There are three types of shifts: logical, circular, and arithmetic
- A *logical shift* is one that transfers 0 through the serial input
- The symbols *shl* and *shr* are for logical shift-left and shift-right by one position

  $R1 \leftarrow shl\ R1$

- The *circular shift* (aka rotate) circulates the bits of the register around the two ends without loss of information
- The symbols *cil* and *cir* are for circular shift left and right

TABLE 4-7 Shift Microoperations

Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register R
$R \leftarrow \text{shr } R$	Shift-right register R
$R \leftarrow \text{cil } R$	Circular shift-left register R
$R \leftarrow \text{cir } R$	Circular shift-right register R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left R
$R \leftarrow \text{ashr } R$	Arithmetic shift-right R

- The *arithmetic shift* shifts a signed binary number to the left or right
- To the left is multiplying by 2, to the right is dividing by 2
- Arithmetic shifts must leave the sign bit unchanged
- A sign reversal occurs if the bit in R_{n-1} changes in value after the shift
- This happens if the multiplication causes an overflow
- An overflow flip-flop V_s can be used to detect the overflow

$$V_s = R_{n-1} \oplus R_{n-2}$$

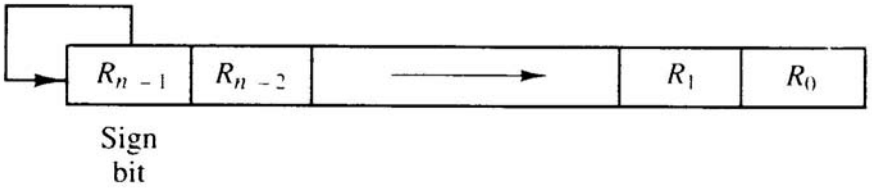
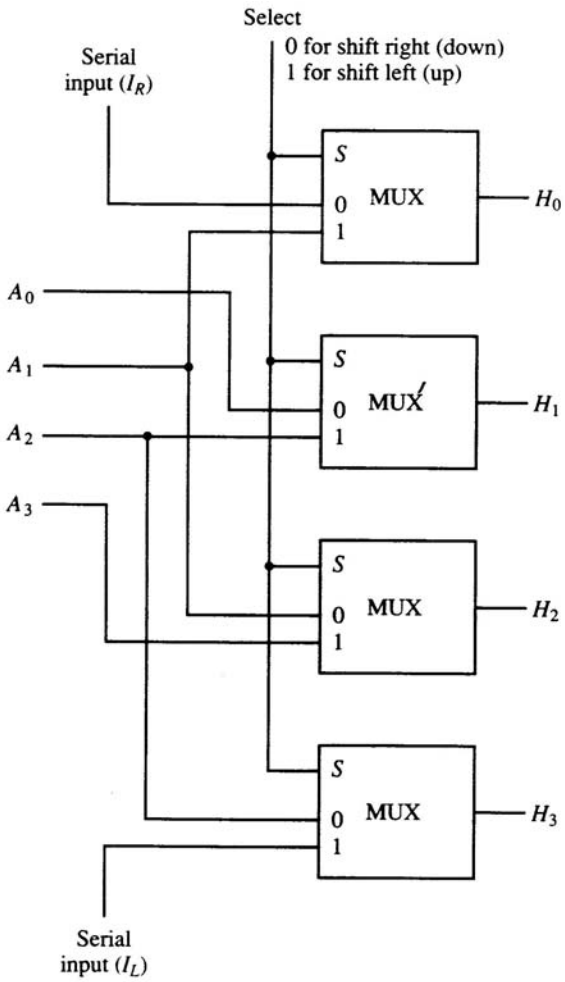


Figure 4-11 Arithmetic shift right.

- A bi-directional shift unit with parallel load could be used to implement this
- Two clock pulses are necessary with this configuration: one to load the value and another to shift
- In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit
- The content of a register to be shifted is first placed onto a common bus and the output is connected to the combinational shifter, the shifted number is then loaded back into the register
- This can be constructed with multiplexers



Function table

Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Figure 4-12 4-bit combinational circuit shifter.

Section 4.7 – Arithmetic Logic Shift Unit

- The *arithmetic logic unit (ALU)* is a common operational unit connected to a number of storage registers
- To perform a microoperation, the contents of specified registers are placed in the inputs of the ALU
- The ALU performs an operation and the result is then transferred to a destination register
- The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period

Figure 4-13 One stage of arithmetic logic shift unit.

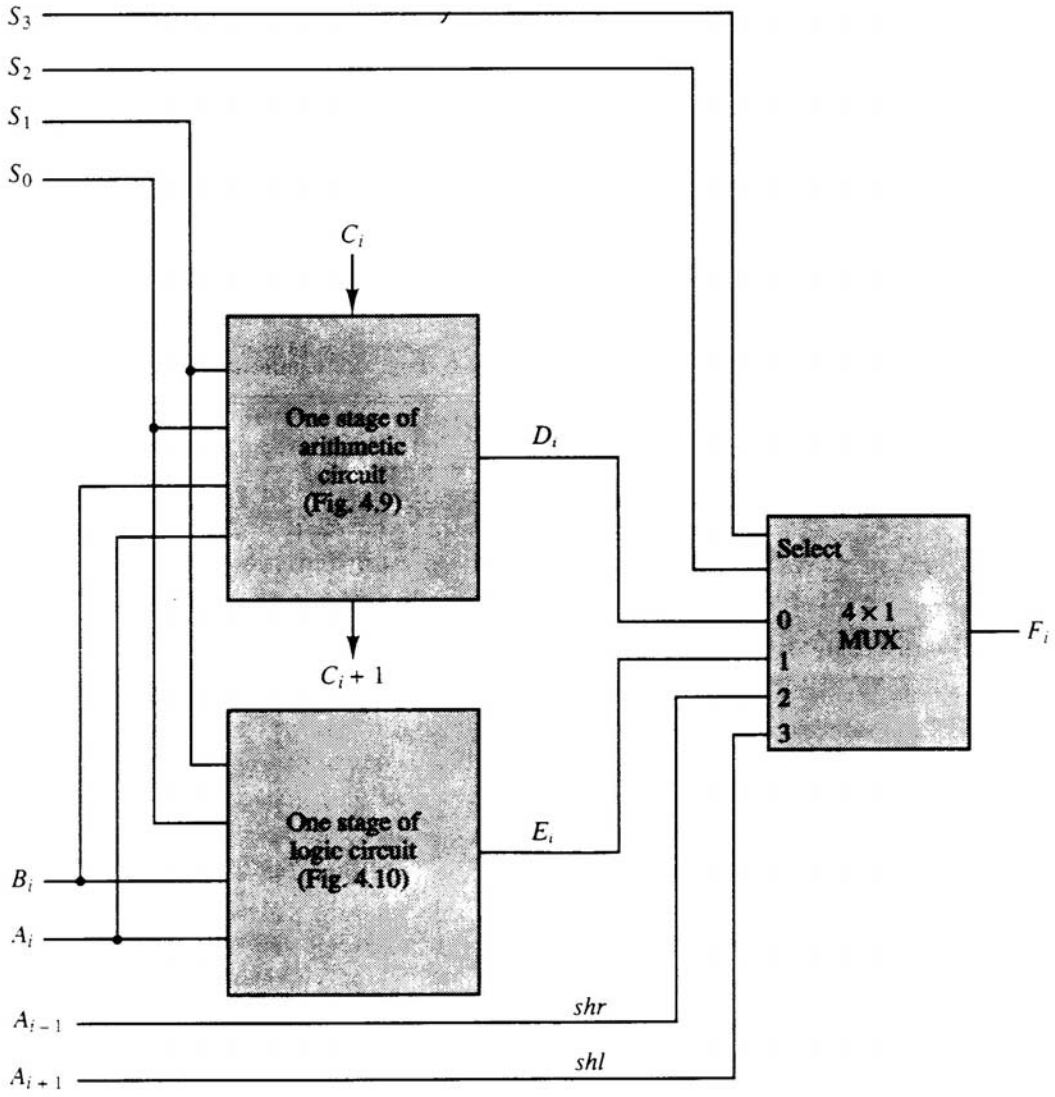


TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	$F = \text{shr } A$	Shift right A into F
1	1	×	×	×	$F = \text{shl } A$	Shift left A into F

5 Computer Organization

Purpose of This Chapter

• To implement a stored program computer which can execute a set of **instructions**.

•(A stored program computer behaves as different machines by loading different programs, i.e., sequences of instructions.)

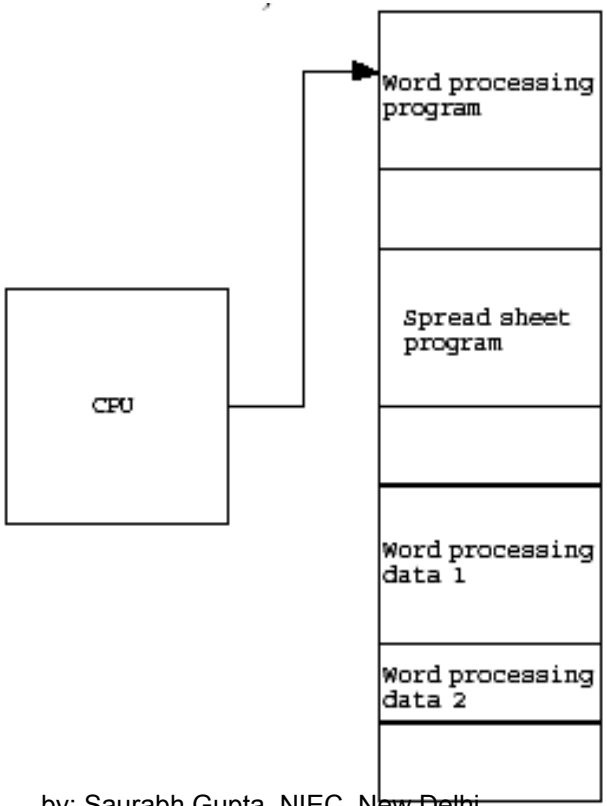


Figure: Stored program computer concept.

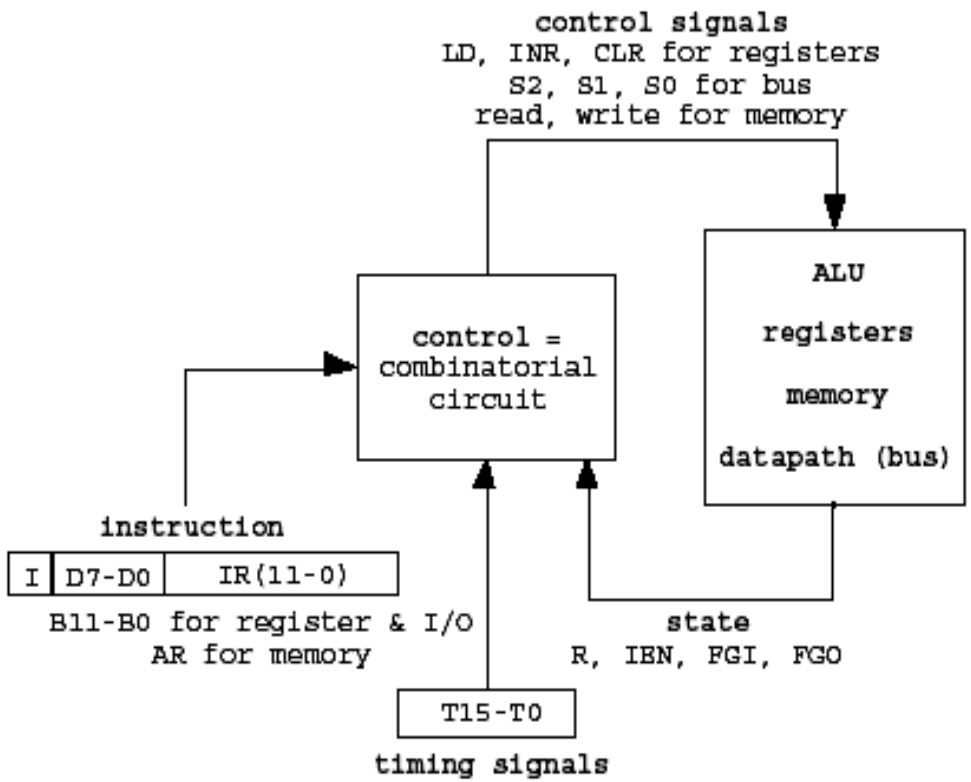


Figure: Stored program computer hardware.

• Computer hardware = registers + ALU + datapath (bus) + control unit.

• The computer goes through **instruction cycles**:

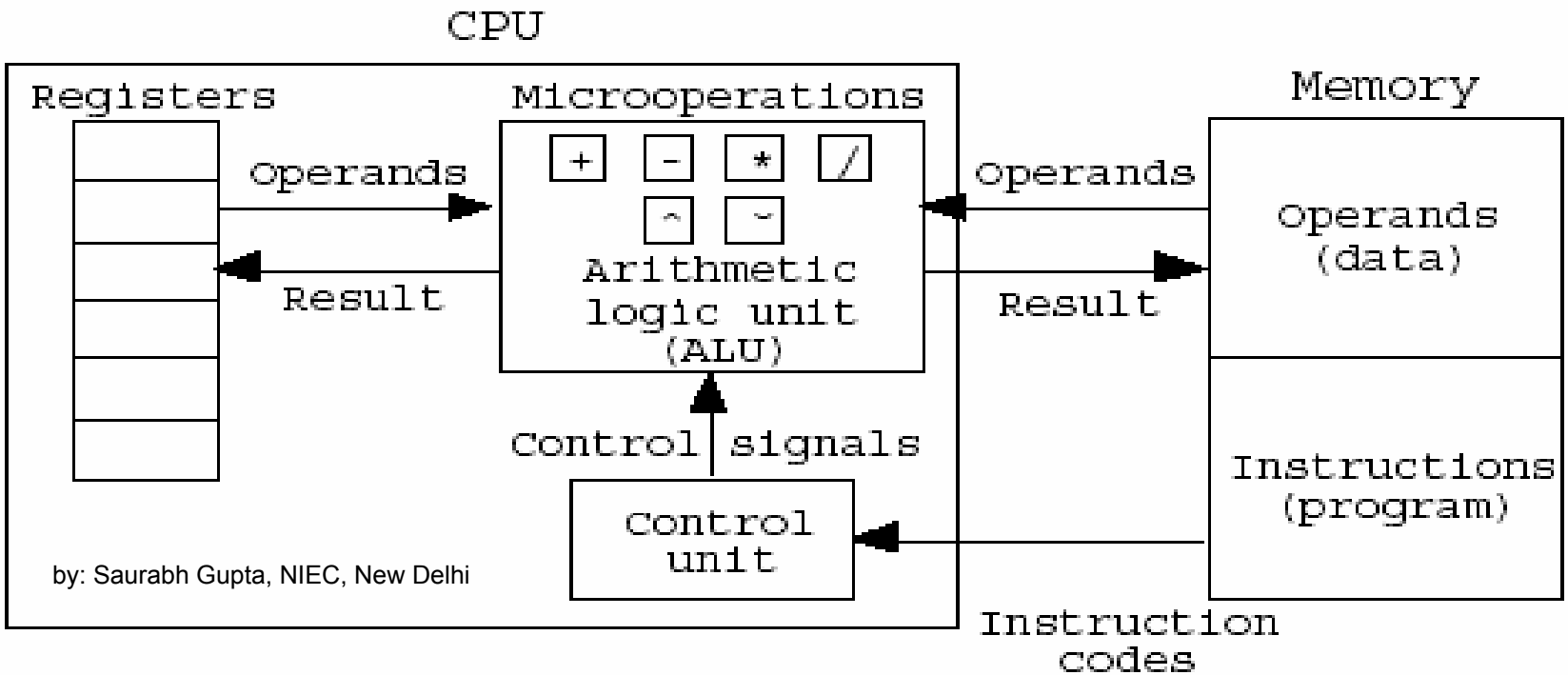
i) Fetch an instruction from memory;

ii) Decode the instruction to a sequence of control signals;

iii) Execute the decoded sequence of microoperations.

• **Control unit**: Instruction → a time sequence of control signals to trigger microoperations.

• Input-output is implemented using an **interrupt cycle**.



5-1. Instruction Codes

- Stored Program Organization : ***Fig. 5-1***
  - The simplest way to organize a computer
    - One processor register : AC(Accumulator)
      - » The operation is performed with the memory operand and the content of AC
    - Instruction code format with two parts : Op. Code + Address
      - » Op. Code : specify 16 possible operations(4 bit)
      - » Address : specify the address of an operand(12 bit)
      - » If an operation in an instruction code does not need an operand from memory, the rest of the bits in the instruction(***address field***) can be used for other purpose
    - Memory : 12 bit = 4096 word(Instruction and Data are stored)
      - » Store each instruction code(***program***) and operand (***data***) in 16-bit memory word

Example
Clear AC, Increment AC, Complement AC, ...

–Addressing Mode

I=0 : Direct,
I=1 : Indirect

- Immediate operand address :
  - –the second part of instruction code(***address field***) specifies ***operand***
- Direct operand address : ***Fig. 5-2(b)***
  - –the second part of instruction code specifies the ***address of operand***
- Indirect operand address : ***Fig. 5-2(c)***
  - –the bits in the second part of the instruction designate an ***address of a memory word in which the address of the operand is found*** (Pointer)
- One bit of the instruction code is used to distinguish between a direct and an indirect address : ***Fig. 5-2(a)***

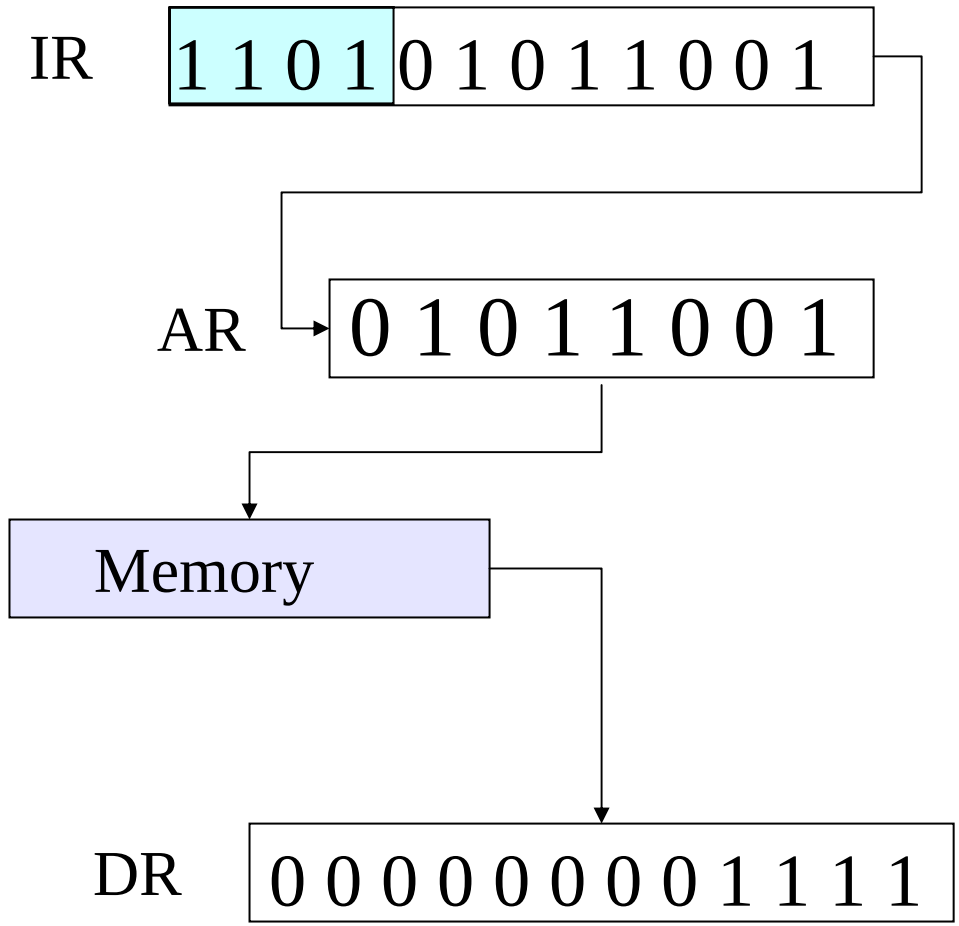
- **Effective address**: Address where an operand is physically located

Direct Addressing

Occurs When the Operand Part Contains the Address of Needed Data.

1. Address part of IR is placed on the bus and loaded back into the AR

2. Address is selected in memory and its Data placed on the bus to be loaded into the Data Register to be used for requested instructions



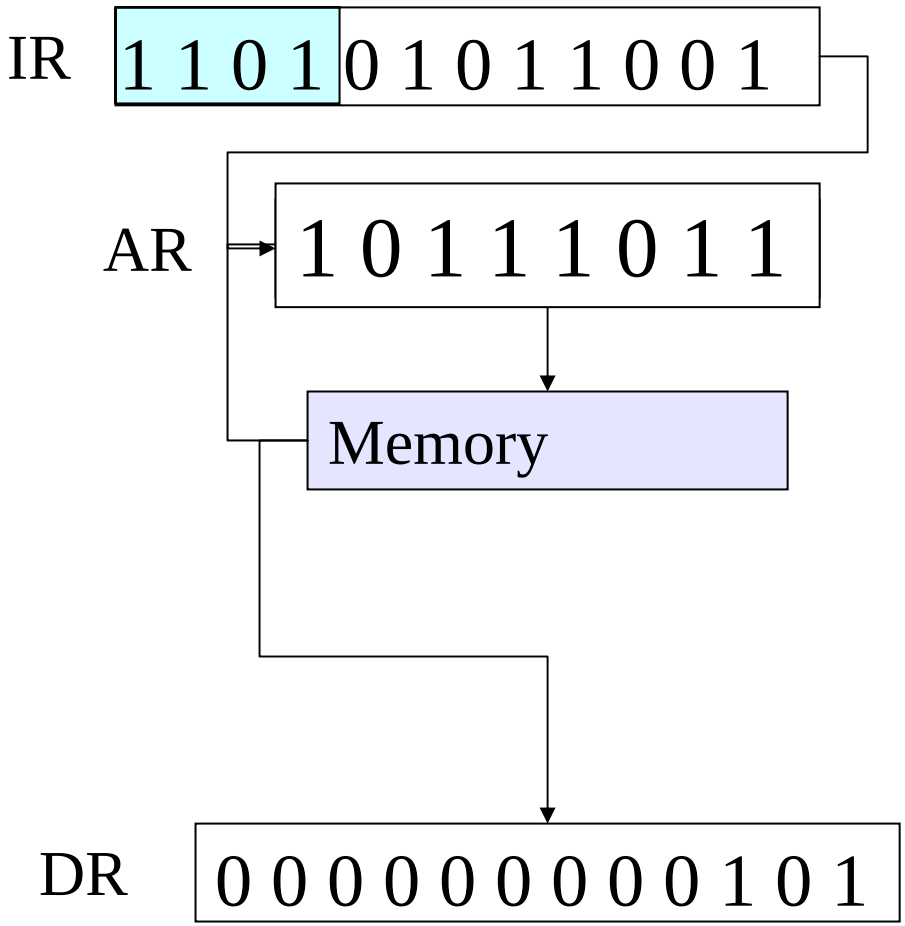
Indirect Addressing

Occurs When the Operand Contains the Address of the Address of Needed Data.

1. Address part of IR is placed on the bus and loaded back into the AR

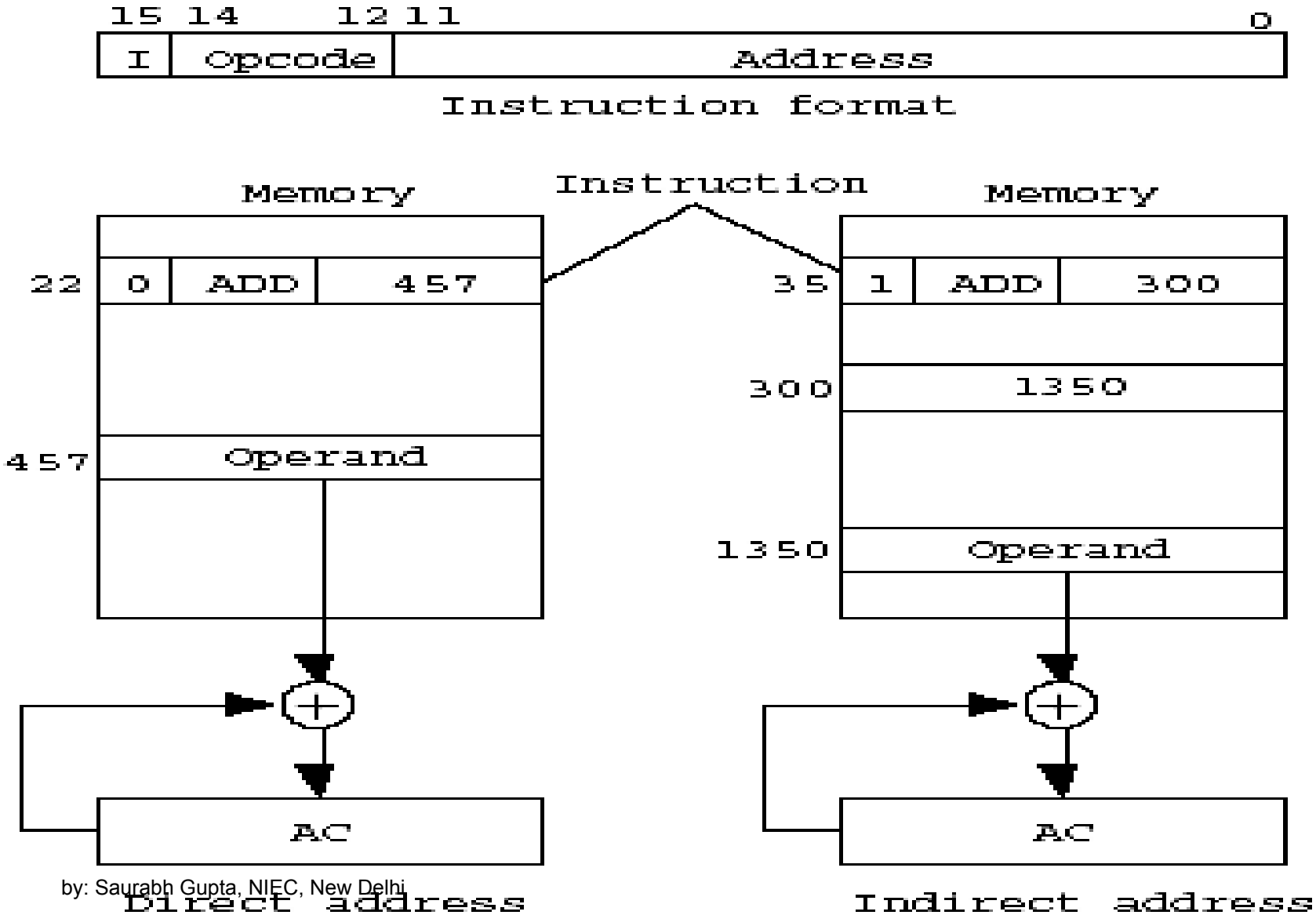
2. Address is selected in memory and placed on the bus to be loaded Back into the AR

3. New Address is selected in memory and placed on the bus to be loaded into the DR to use later



by: Saurabh Gupta, NIEC, New Delhi

Direct and Indirect addressing example



by: Saurabh Gupta, NIEC, New Delhi

5-2 Computer Registers

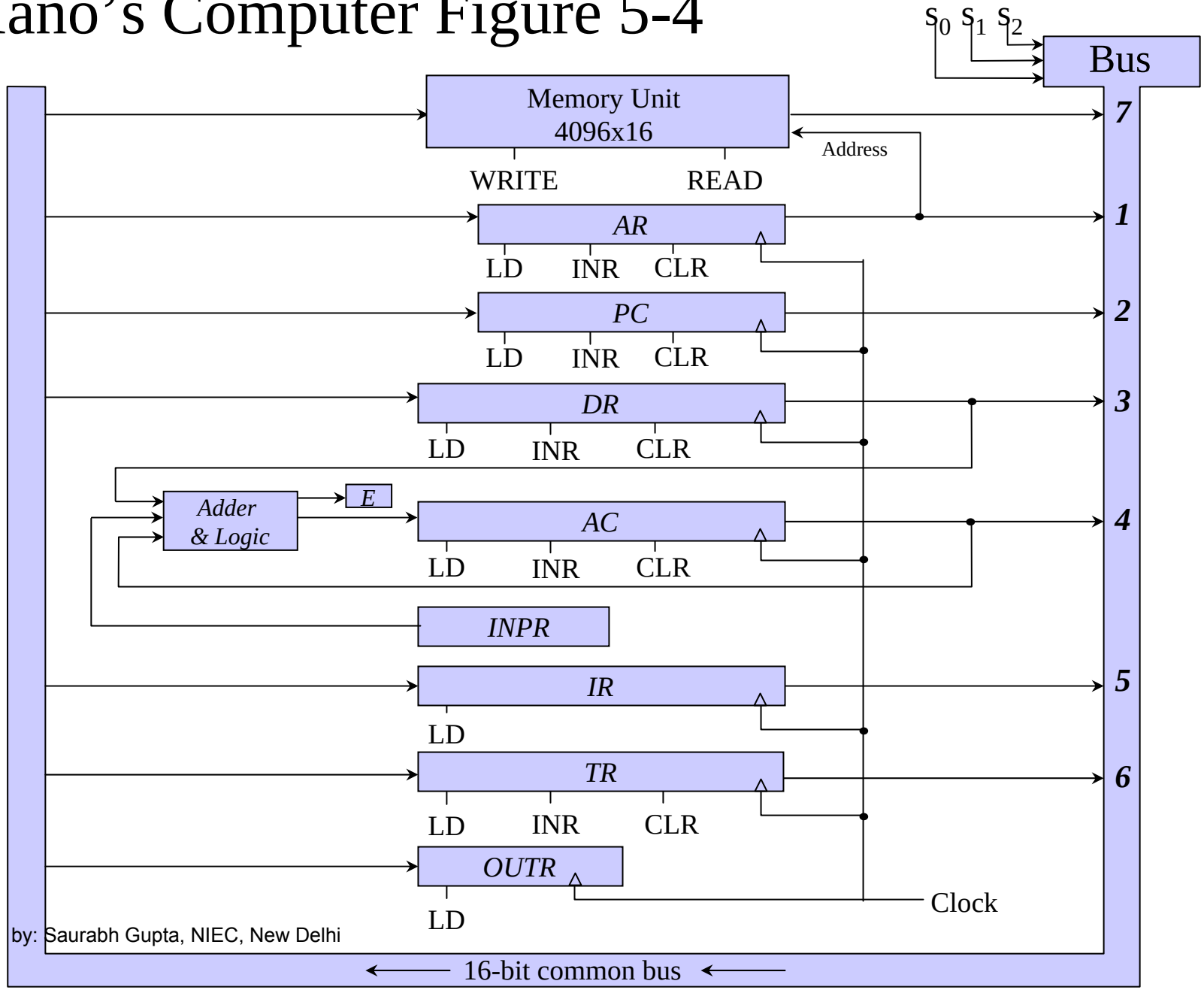
- Data Register(***DR***) : hold the operand(Data) read from memory
  - Accumulator Register(***AC***) : general purpose processing register
  - Instruction Register(***IR***) : hold the instruction read from memory
  - Temporary Register(***TR***) : hold a temporary data during processing
  - Address Register(***AR***) : hold a memory address, 12 bit width
  - Program Counter(***PC***) :
    - »hold the address of the next instruction to be read from memory after the current instruction is executed
    - »Instruction words are read and executed in sequence unless a branch instruction is encountered
    - »A branch instruction calls for a transfer to a nonconsecutive instruction in the program
    - »The address part of a branch instruction is transferred to PC to become the address of the next instruction
    - »To read instruction, memory read cycle is initiated, and PC is incremented by one(next instruction fetch)

- Input Register(***INPR***) : receive an 8-bit character from an input device
- Output Register(***OUTR***) : hold an 8-bit character for an output device

The following registers are used in Mano's example computer.

Register symbol	Number of bits	Register name	Register Function
DR	16	Data register	Holds memory operands
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

Mano's Computer Figure 5-4



Common Bus System

- The basic computer has eight registers, a memory unit, and a control unit.
- Paths must be provided to transfer information from one register to another and between memory and registers
- A more efficient scheme for transferring information in a system with many registers is to use a common bus.
- The connection of the registers and memory of the basic computer to a common bus system : ***Fig. 5-4***
  - The outputs of seven registers and memory are connected to the common bus
  - The specific output is selected by mux(S0, S1, S2) :
    - Memory(7), AR(1), PC(2), DR(3), AC(4), IR(5), TR(6)
    - When LD(Load Input) is enable, the particular register receives the data from the bus
  - Control Input : LD, INC, CLR, Write, Read

COMMON BUS SYSTEM

• **Control variables:** Various control variables are used to select:

i) the paths of information; &

ii) the operation of the registers.

- **Selection variables**: Used to specify a register whose output is connected to the common bus at any given time.
- To select one register out of 8, we need 3 select variables.
- For example, if S2S1S0 = 011, the output of DR is directed to the common bus.

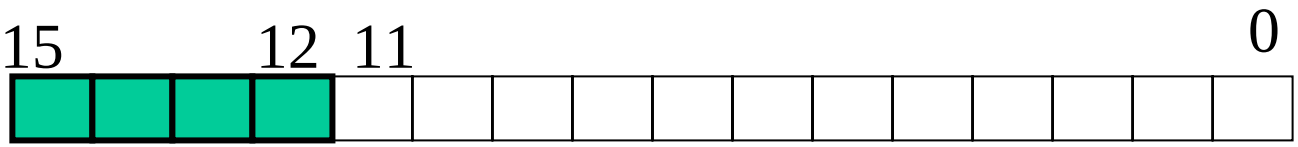
> **Load input (LD)**: Enables the input of a register connected to the common bus. When LD = 1 for a register, the data on the common bus is read into the register during the next clock pulse transition.

> **Increment input (INR)**: Increments the content of a register.

> **Clear input (CLR)**: Clear the content of a register to zero.

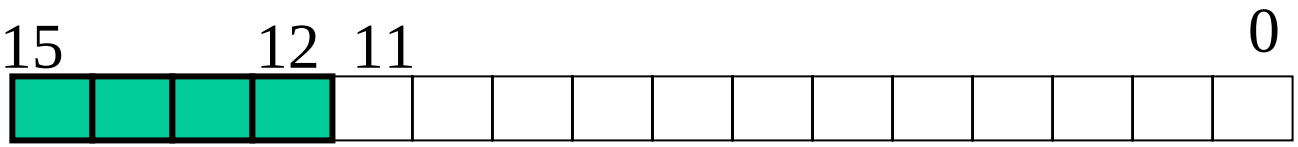
• When the contents of AR or PC (12 bits) are applied to the 16-bit common bus, the four most significant bits are set to zero. When AR or PC receives information from the bus, only the 12 least significant bits are transferred to the register. Both INPR and OUTR use only the 8 least significant bits of the bus.

Mano's Computer: Memory Words



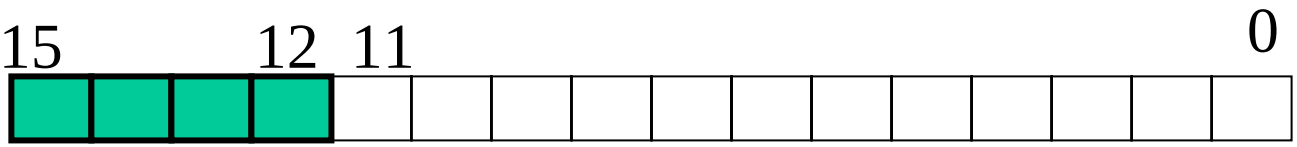
- 4-bit opcode Bits 15-12
- How many possible instructions?
  - $2^4=16$
- This leaves 12 bits for the address
  - How many words of memory?
  - $2^{12} = 2^2 \bullet 2^{10} = 4K = 4096$ 16-bit words

Mano's Computer: Instructions



- $2^4 = 16$ possible instructions
  - Op-code 0111 reserved for register-reference instructions
  - How many possible register-reference instructions?
  - $2^{12} = 2^2 \bullet 2^{10} = 4K = 4096$ possible r-r instructions (only 12 are used)

Mano's Computer: Instructions



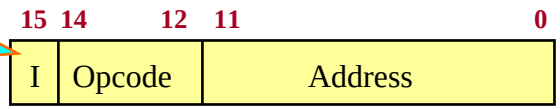
- $2^4 = 16$ possible instructions
  - Op-code 1111 reserved for input/output instructions

- $2^4 = 16$ possible instructions - 0111 (r-r) - 1111 (i/o) = 14 instructions left
  - These are coded as 7 instructions with direct and indirect addressing

5-3. Computer Instruction

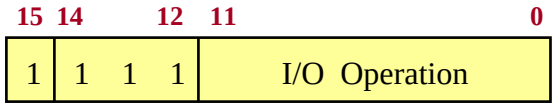
- 3 Instruction Code Formats : ***Fig. 5-5***
  - Memory-reference instruction
    - –Opcode = 000 ~ 110
      - »I=0 : 0xxx ~ 6xxx, I=1: 8xxx ~Exxx

I=0 : Direct,
I=1 : Indirect



      - »Register-reference instruction
    - –7xxx (7800 ~ 7001) : CLA, CMA,

    - –Input-Output instruction
    - –Fxxx(F800 ~ F040) : INP, OUT, ION, SKI,



Symbol	Hex Code I = 0	Hex Code I = 1	Description
AND	0xxx	8xxx	And memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and Save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMS	7200		Complement AC
CME m	7100	e	Comp
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instruction if AC positive
SNA	7008		Skip next instruction if AC negative
SZA	7004		Skip next instruction if AC zero
SZE	7002		Skip next instruction if E is 0
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrup
IOF	F040		Inter

Reading this table:
the presented code is for any instruction that has 16 bits. The xxx represents don't care (any data for the first 12 bits). Example 7002 for is a hexadecimal code equivalent to 0111 0000 0000 0010 Which means B_1 (Bit 1) is set to 1 and the rest of the first 12 bits are set to zeros.

•• Instructions are normally stored in consecutive memory locations and are executed sequentially one at a time.

•The **program counter** (PC) holds the address of the next instruction to be read from memory after the current instruction is executed.

•The PC has 12 bits like the AR.

•The instruction read from memory is placed in the **instruction register** (IR) which has 16 bits corresponding to our instruction code length.

• Most processing takes place in the **accumulator** (AC);

•the **temporary register** (TR) is used for holding temporary data during the processing.

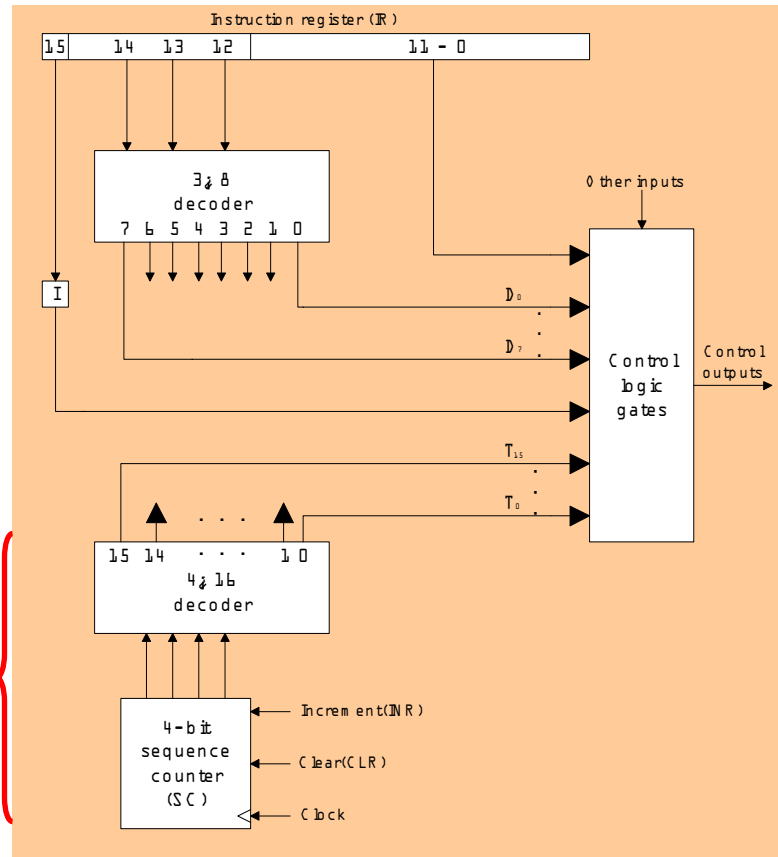
• The **input** (INPR) and **output** (OUTR) **registers** hold a character at a time which is read from an input device or to be printed to an output device, respectively. Using the ASCII code, one character is represented with 8 bits (1 byte).

5-4. Timing and Control

- Microprogrammed Control : *Chap. 7*
  - The control information is stored in a control memory, and the control memory is programmed to initiate the required sequence of microoperations
  - \+ Any required change can be done by updating the microprogram in control memory, - Slow operation

– Control Unit : *Fig. 5-6*

- Control Unit = Control Logic Gate + 3 X 8 Decoder + Instruction Register + Timing Signal
- Timing Signal = 4 X 16 Decoder + 4-bit Sequence Counter
- Example) Control timing : *Fig. 5-7*
  - Sequence Counter is cleared when D_3T_4 =1 : $D_3T_4 : SC \leftarrow 0$
- Memory R/W cycle time > Clock cycle time



by: Saurabh Gupta, NIEC, New Delhi

CONTROL UNIT HARDWARE

• Inputs to the control unit come from IR where an instruction read from the memory unit is stored.

• A hardwired control is implemented in the example computer using:

> A 3 ´ 8 decoder to decode opcode bits 12-14 into signals D0, ..., D7;

> A 4-bit binary **sequence counter** (SC) to count from 0 to 15 to achieve time sequencing;

> A 4 ´ 16 decoder to decode the output of the counter into 16 timing signals, T0, ..., T15

> A flip-flop (I) to store the addressing mode bit in IR;

> A digital circuit with inputs—D0, ..., D7, T0, ..., T15, I, and address bits (11-0) in IR—to generate control outputs supplied to control inputs and select signals of the registers and the bus.

• **Clocking principle**: The binary counter goes through a cycle, 0000 → 0001 → 0010 → ... → 1111 → 0000. Accordingly only one of T0, ..., T15 is 1 at each clock cycle, T0 → T1 → T2 → ... → T15 → T0; all the other timing signals are 0.

• By setting the clear input (CLR) of SC at a clock cycle, say T3, we can achieve a 4-cycle clock: T0 → T1 → T2 → T3 → T0.

5.5 Instruction Cycle

• A computer goes through the following instruction cycle repeatedly:

do

1. Fetch an instruction from memory

2. Decode the instruction

3. Read the effective address from memory if the instruction has an indirect address

4. Execute the instruction until a HALT instruction is encountered

• The fetch & decode phases of the instruction cycle consists of the following microoperations synchronized with the timing signals (clocking principle).

Timing signal	microoperations
T0:	**AR ← PC**
T1:	**IR ← M[AR], PC ← PC + 1**
T2:	**D0, ..., D7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)**

0: Since only AR is connected to the address inputs of memory, the address of instruction is transferred from PC to AR.

. Place the content of PC onto the bus by making the bus selection inputs S2S1S0 = 010.

. Transfer the content of the bus to AR by enabling the LD input to AR

AR ← PC).

1: The instruction read from memory is then placed in the instruction register IR. At the same time, PC is incremented to prepare for the address of the next instruction.

. Enable the read input of the memory.

. Place the content of memory onto the bus by making the bus selection inputs S2S1S0 = 111. (Note that the address lines are always connected to AR, and we have already placed the next instruction address in AR.)

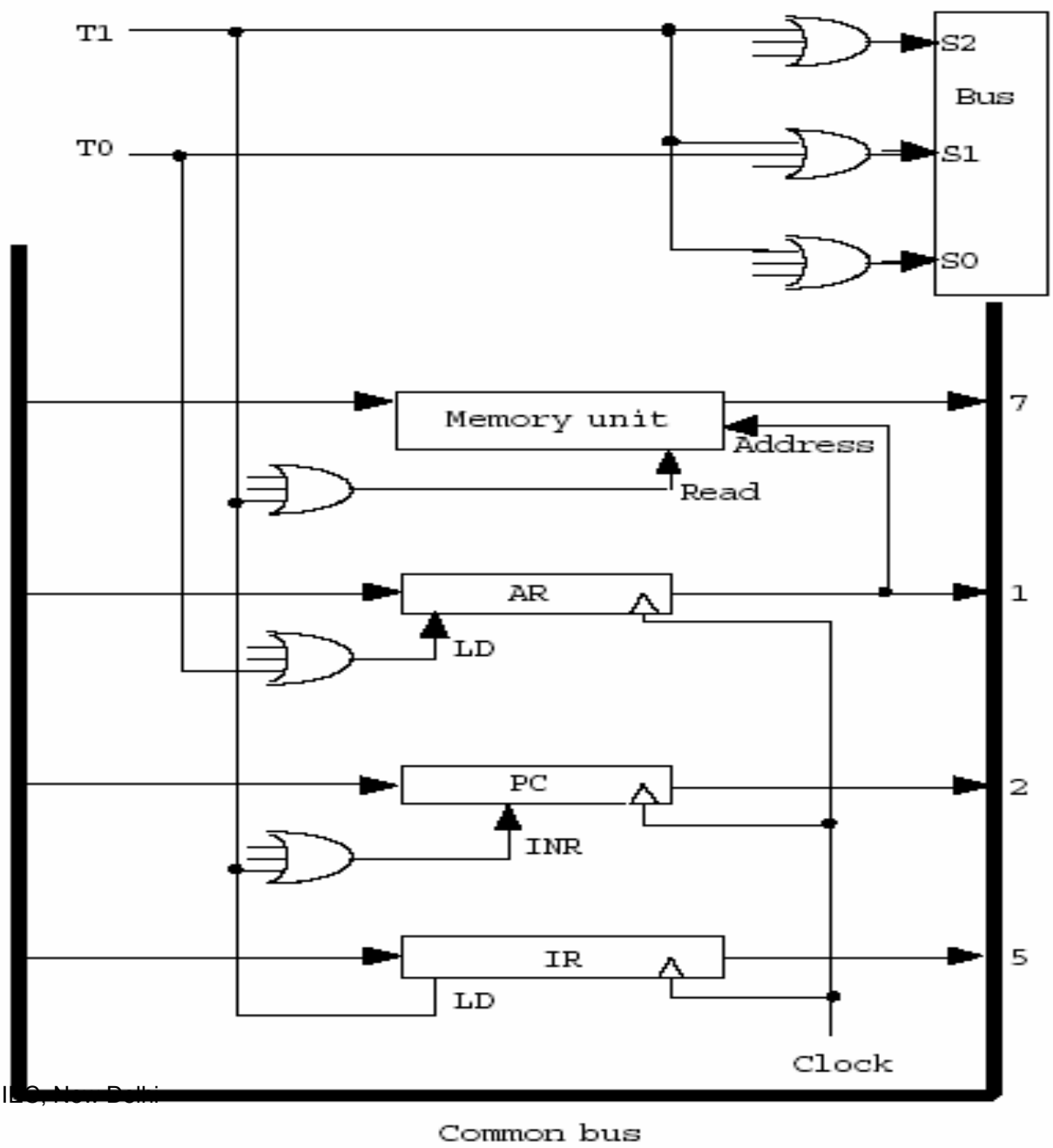
Similar circuits are used to realize the microoperations at T2.

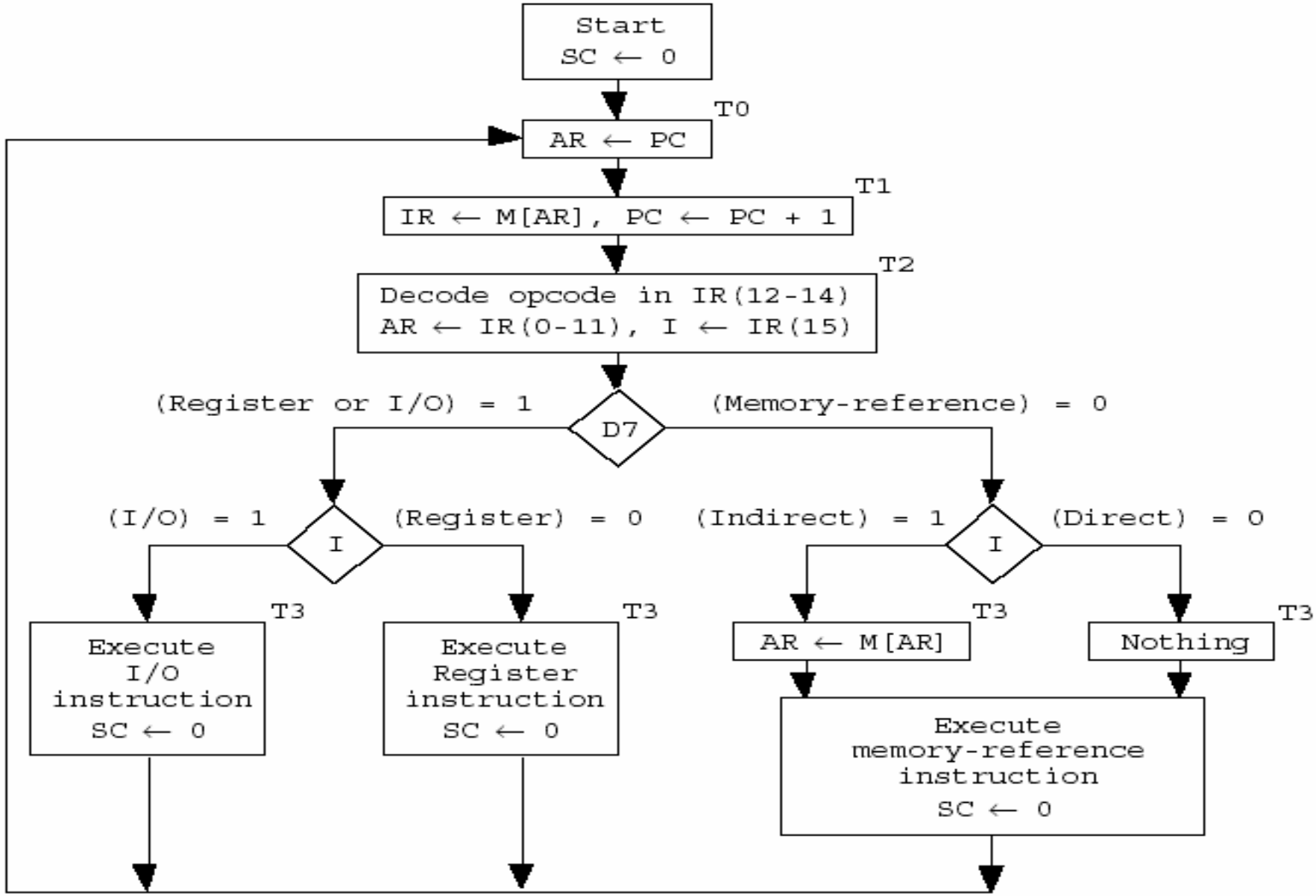
• At T3, microoperations which take place depend on the type of instruction. The four different paths are symbolized as follows, where the control functions must be connected to the proper inputs to activate the desired microoperations.

Control function	Microoperation
D7'IT3:	**AR ← M[AR], indirect memory transfer**
D7'I'T3:	**Nothing, direct memory transfer**
D7I'T3:	**Execute a register-reference instruction**
D7IT3:	**Execute an I/O instruction**

When D7'T3 = 1 (At T3 & IR(12-14) ≠ 111), the execution of memory-reference instructions takes place with the next timing variable T4.

Figure: Control circuit for instruction fetch. This is a part of the control circuit and demonstrates the kind of wiring needed.





by: Saurabh Gupta, NIEC, New Delhi

Figure: Flowchart for fetch & decode phases.

REGISTER-REFERENCE INSTRUCTIONS

• The 12 register-reference instructions are recognized by I = 0 and D7 = 1 (IR(12-14) = 111). Each operation is designated by the presence of 1 in one of the bits in IR(0-11). Therefore D7I'T3 ≡ r = 1 is common to all register-transfer instructions.

Symbol	Control	Microoperations	Description
	r	SC ← 0 (Common to all, done in 1 cycle)	Clear SC
CLA	rB_{11}	AC ← 0	Clear AC
CLE	rB_{10}	E ← 0	Clear E
CMA	rB_9	AC ← $\overline{AC}$	Complement AC
CME	rB_8	E ← $\overline{E}$	Complement E
CIR	rB_7	AC ← shr AC, AC(15) ← E, E ← AC(0)	Circular right
CIL	rB_6	AC ← shl AC, AC(0) ← E, E ← AC(15)	Circular left
INC	rB_5	AC ← AC + 1	Increment AC
SPA	rB_4	If AC(15)=0 then PC ← PC + 1	Skip if positive
SNA	rB_3	If AC(15)=1 then PC ← PC + 1	Skip if negative
SZA	rB_2	If AC=0 then PC ← PC + 1	Skip if AC zero
SZE	rB_1	If E=0 then PC ← PC + 1	Skip if E zero
HLT	rB_0	S ← 0 (S is a start-stop flip-flop)	Halt computer

5.6 Memory Reference Instructions

• Opcode (000 - 110) or the decoded output Di (i = 0, ..., 6) are used to select one memory-reference operation out of 7.

Symbol	Operation decoder	Symbolic description
AND	D_0	AC ← AC ^ M[AR]
ADD	D_1	AC ← AC + M[AR], E ← C_{out}
LDA	D_2	AC ← M[AR]
STA	D_3	M[AR] ← AC
BUN	D_4	PC ← AR
BSA	D_5	M[AR] ← PC, PC ← AR + 1
ISZ	D_6	M[AR] ← M[AR] + 1, If M[AR] + 1 = 0 then PC ← PC + 1

5-6. Memory Reference Instruction

- STA : memory write

$$D_3T_4 : M[AR] \leftarrow AC, SC \leftarrow 0$$

- BUN : branch unconditionally

$$D_4T_4 : PC \leftarrow AR, SC \leftarrow 0$$

- BSA : branch and save return address

$$D_5T_4 : M[AR] \leftarrow PC, AR \leftarrow AR + 1$$
$$D_5T_5 : PC \leftarrow AR, SC \leftarrow 0$$

  - Return Address : save return address (135 ← 21)
  - Subroutine Call : *Fig. 5-10*

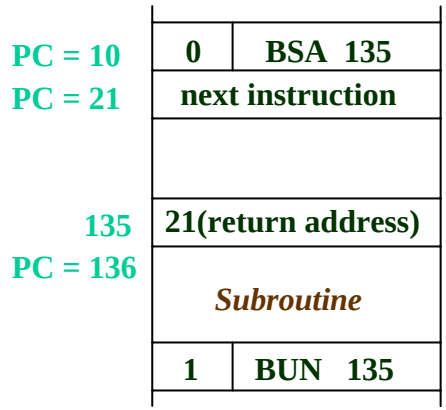
$$D_5T_4 : M[135] \leftarrow 21(PC), 136(AR) \leftarrow 135 + 1$$
$$D_5T_5 : 136(PC) \leftarrow 136(AR), SC \leftarrow 0$$

- ISZ : increment and skip if zero

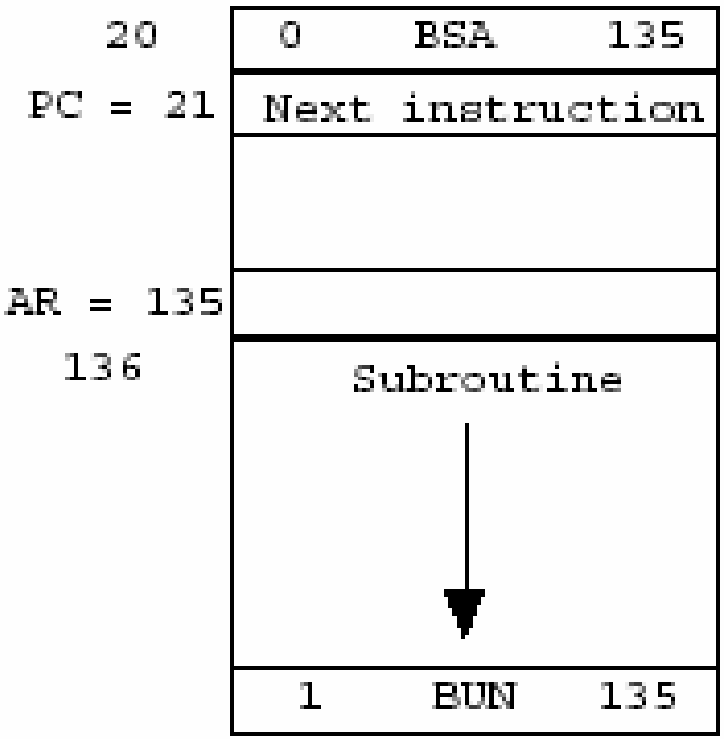
$$D_6T_4 : DR \leftarrow M[AR]$$
$$D_6T_5 : DR \leftarrow DR + 1$$
$$D_6T_6 : M[AR] \leftarrow DR, \textit{if}\ (DR = 0)\ \textit{then}\ (PC \leftarrow PC + 1), SC \leftarrow 0$$

- Control Flowchart : ***Fig. 5-11***
  - Flowchart for the 7 memory reference instruction
    - The longest instruction : ISZ(T6)
    - 3 bit Sequence Counter

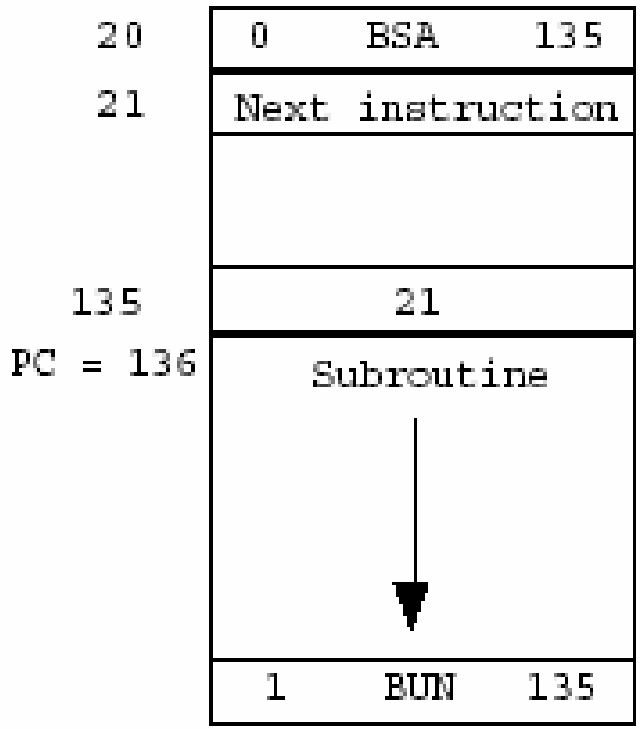
Fig. 5-10 Example of BSA



Branch and Save Address (BSA)



Memory, PC, and AR at time T_4



Memory and PC after BSA execution

by: Saurabh Gupta, NIEC, New Delhi

Subroutine implementation using BSA.

5-7. Input-Output and Interrupt

- 5-7 Input-Output and Interrupt
  - Input-Output Configuration : ***Fig. 5-12***
    - Input Register(***INPR***), Output Register(***OUTR***)
      - These two registers communicate with a communication interface serially and with the AC in parallel
      - Each quantity of information has eight bits of an alphanumeric code
    - Input Flag(***FGI***), Output Flag(***FGO***)
      - FGI : ***set*** when INPR is ready, ***clear*** when INPR is empty
      - FGO : ***set*** when operation is completed, ***clear*** when output device is in the process of printing

1 : Ready
0 : Not ready

  - Input-Output Instruction : *Tab. 5-5*
    - $p = D_7IT_3$
    - $IR(i) = B_i \leftarrow IR(6 - 11)$

Address

    - ***B_6 - B_{11}*** : **6 I/O Instruction**
  - Program Interrupt
    - I/O Transfer Modes
      - 1) Programmed I/O, 2) Interrupt-initiated I/O, 3) DMA, 4) IOP
      - 2) Interrupt-initiated I/O (FGI FGO 1 Int.)
      - Maskable Interrupt (ION IOF Int. mask)

- Interrupt Cycle : *Fig. 5-13*
  - During the execute phase, IEN is checked by the control
    - » IEN = 0 : the programmer does not want to use the interrupt, so control continues with the next instruction cycle
    - » IEN = 1 : the control circuit checks the flag bit, If either flag set to 1, R (R is the interrupt flip flop) is set to 1
  - At the end of the execute phase, control checks the value of R
    - » R = 0 : instruction cycle
    - » R = 1 : Interrupt cycle
- Demonstration of the interrupt cycle : ***Fig. 5-14***
  - The memory location at address 0 as the place for storing the return address
  - Interrupt Branch to memory location 1
  - Interrupt cycle IEN=0 (*ISR Interrupt ION*)
- The condition for R = 1

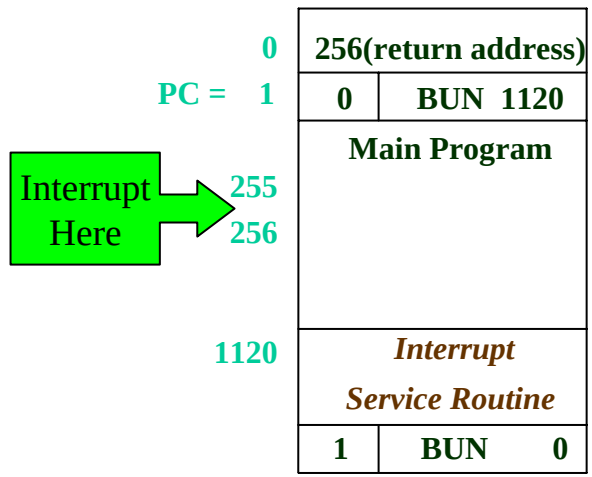
$$T_0' T_1' T_2' (IEN)(FGI + FGO) : R \leftarrow 1$$

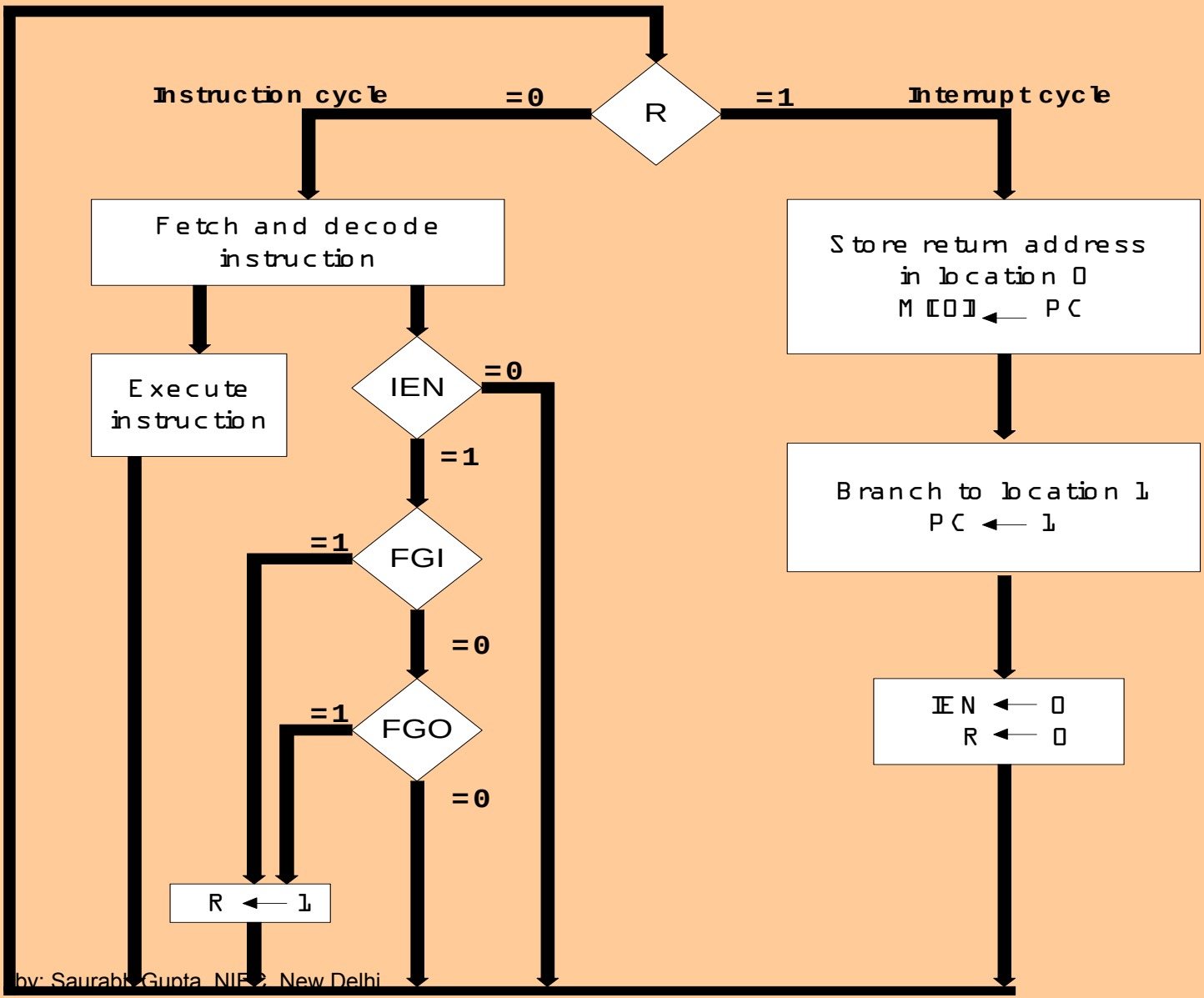
- Modified Fetch Phase
  - Modified Fetch and Decode Phase

Save Return Address(PC) at 0

Jump to 1(PC=1)

$$RT_0 : AR \leftarrow 0, TR \leftarrow PC$$
$$RT_1 : M[AR] \leftarrow TR, PC \leftarrow 0$$
$$RT_2 : PC \leftarrow PC + 1, IEN \leftarrow 0, R \leftarrow 0, SC \leftarrow 0$$




Instruction cycle
=0
R
=1
Interrupt cycle
Fetch and decode instruction
Store return address in location 0
M[0] ← PC
Execute instruction
IEN
=0
=1
Branch to location 1
PC ← 1
=1
FGI
=0
IEN ← 0
R ← 0
=1
FGO
=0
R ← 1

Mano's Computer: RTL

TABLE 5-6 Control Functions and Microoperations for the Basic Computer

Fetch	$R'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR],\quad PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14),$
		$AR \leftarrow IR(0\text{–}11),\quad I \leftarrow IR(15)$
Indirect	D_7IT_3:	$AR \leftarrow M[AR]$
Interrupt:		
$T_0T_1T_2(IEN)(FGI + FGO)$:		$R \leftarrow 1$
	RT_0:	$AR \leftarrow 0,\quad TR \leftarrow PC$
	RT_1:	$M[AR] \leftarrow TR,\quad PC \leftarrow 0$
	RT_2:	$PC \leftarrow PC + 1,\quad IEN \leftarrow 0,\quad R \leftarrow 0,\quad SC \leftarrow 0$
Memory-reference:		
AND	D_0T_4:	$DR \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR,\quad SC \leftarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR,\quad E \leftarrow C_{out},\quad SC \leftarrow 0$
LDA	D_2T_4:	$DR \leftarrow M[AR]$
	D_2T_5:	$AC \leftarrow DR,\quad SC \leftarrow 0$
STA	D_3T_4:	$M[AR] \leftarrow AC,\quad SC \leftarrow 0$
BUN	D_4T_4:	$PC \leftarrow AR,\quad SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC,\quad AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR,\quad SC \leftarrow 0$
ISZ	D_6T_4:	$DR \leftarrow M[AR]$
	D_6T_5:	$DR \leftarrow DR + 1$
	D_6T_6:	$M[AR] \leftarrow DR,\quad$ if $(DR = 0)$ then $(PC \leftarrow PC + 1),\quad SC \leftarrow 0$
Register-reference:		
		$D_7I'T_3 = r$ (common to all register-reference instructions)
		$IR(i) = B_i\ (i = 0, 1, 2, \ldots, 11)$
	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow \overline{AC}$
CME	rB_8:	$E \leftarrow \overline{E}$
CIR	rB_7:	$AC \leftarrow$ shr $AC,\quad AC(15) \leftarrow E,\quad E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow$ shl $AC,\quad AC(0) \leftarrow E,\quad E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$
HLT	rB_0:	$S \leftarrow 0$
Input-output:		
		$D_7IT_3 = p$ (common to all input–output instructions)
		$IR(i) = B_i\ (i = 6, 7, 8, 9, 10, 11)$
	p:	$SC \leftarrow 0$
INP	pB_{11}:	$AC(0\text{–}7) \leftarrow INPR,\quad FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{–}7),\quad FGO \leftarrow 0$
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$
ION	pB_7:	$IEN \leftarrow 1$
IOF	pB_6:	$IEN \leftarrow 0$

5-8. Complete Computer Description

- 5-8 Complete Computer Description
  - The final flowchart of the instruction cycle : Fig. 5-15
  - The control function and microoperation : Table. 5-6
- 5-9 Design of Basic Computer
  - The basic computer consists of the following hardware components
    - 1. A memory unit with 4096 words of 16bits
    - 2. Nine registers : AR, PC, DR, AC, IR, TR, OUTR, INPR, and SC(***Fig. 2-11***)
    - 3. Seven F/Fs : I, S, E, R, IEN, FGI, and FGO
    - 4. Two decoders in control unit : 3 x 8 operation decoder, 4 x 16 timing decoder(*Fig. 5-6*)
    - 5. A 16-bit common bus(***Fig. 5-4***)
    - ***6. Control Logic Gates*** : *Fig. 5-6*
    - ***7. Adder and Logic circuit connected to the AC input***
  - Control Logic Gates
    - 1. Signals to control the inputs of the nine registers
    - 2. Signals to control the read and write inputs of memory
    - 3. Signals to set, clear, or complement the F/Fs
    - 4. Signals for $S_2\ S_1\ S_0$ to select a register for the bus
    - 5. Signals to control the AC adder and logic circuit

Since memory is 4K in size, it requires 12 address bits. Each word of memory contains 16 bits of data. Similarly, the program counter (PC) is also 12 bits wide. Each data word is 16 bits wide. The Data Register (DR) must also be 16 bits wide, since it receives data from and sends data to memory. The accumulator (AC) acts on 16 bits of data. The Instruction Register (IR) receives instruction codes from memory which are 16 bits wide.

Basic Computer specification

- **4K x 16 RAM**
- **12-bit AR, PC**
- **16-bit DR, AC, IR, TR**
- **8-bit INPR, OUTR**
- **3-bit SC**
- **1-bit E, I, IEN, FGI, FGO**

TR is a temporary register. Only the CPU can cause it to be accessed. The programmer cannot directly manipulate the contents of TR. Most CPU's have one or more temporary registers which it uses to perform instructions. The input and output registers (INPR and OUTR) are 8 bits wide each. For this CPU, I/O instructions only transfer 8 bits of data at a time. The 3-bit sequence counter (SC) is used to generate the correct timing (T) states. Other 1-bit registers are the carry out (E), the indirect register (I), the interrupt enable (IEN) and the input and output flags (FGI and FGO).

•The control unit must make sure that at most one register (or memory unit) places data onto the bus at one time.

•The memory unit is external to the CPU. It always receives its address from the address register (AR) and makes its data available to the CPU bus. It receives data from the CPU bus as well.

•Read and write signals are supplied by the control unit.

•The address registers, program counter (PC) and data register (DR) each load data onto and receive data from the system bus. Each has a load, increment and clear signal derived from the control unit. These signals are synchronous; each register combines these signals with the system clock to activate the proper function.

•Since AR and PC are only 12-bits each, they use the low order 12 bits of the bus.

•The accumulator makes its data available on the bus but does not receive data from the bus.

• it receives data from ALU (Adder and Logic) only.

•To load data into AC, place it onto the bus via DR and pass it directly through the ALU.

•Note that E, the 1-bit carry flag, also receives its data from the ALU.

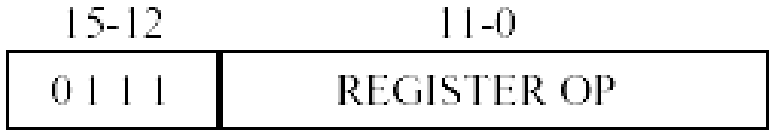
•The input register, INPR, receives data from an external input port and makes it available only to AC.

•The output register makes its data available to the output port using specific hardware.

•The instruction register, IR, can only be loaded; it cannot be incremented nor cleared. Its output is used to generate D_i's and T_i's control signals.

•TR is a temporary register. The CPU uses this register to store intermediate results of operations. It is not accessible by the external programs. It is loaded, incremented and cleared like the other registers.

Register-reference instructions:



- Register reference instructions are those which access data and manipulate the contents of registers.
- They do not access memory.
- These instructions are executed in one clock cycle.
- Each register reference instruction is performed in a single clock cycle.
- Each instruction manipulates the contents of a register within the CPU, so the relatively time consuming accesses to memory are avoided.
- There are 12 register reference instructions overall, each of which is encoded by one of the 12 low order bits of the instruction code.

I/O instructions:

- This class of instructions accesses I/O devices.
- The instructions in this class also enable and disable interrupts. Since this computer only allows for a single input device and a single output device, no address information is needed.
- The input/output instructions are performed in a single clock cycle.
- Note that there are no instructions to set FGI or FGO to 1.
- These flags are set by external hardware when input data is ready or output data is requested. When the CPU performs the proper input or output instruction (INP or OUT), it resets the flag to allow for future I/O data transfers.

Control signals

- T0, T1, ... T6 : Timing signals
- D0, D1, ... D7 : Decoded instruction
- I: Indirect bit
- R: Interrupt cycle bit

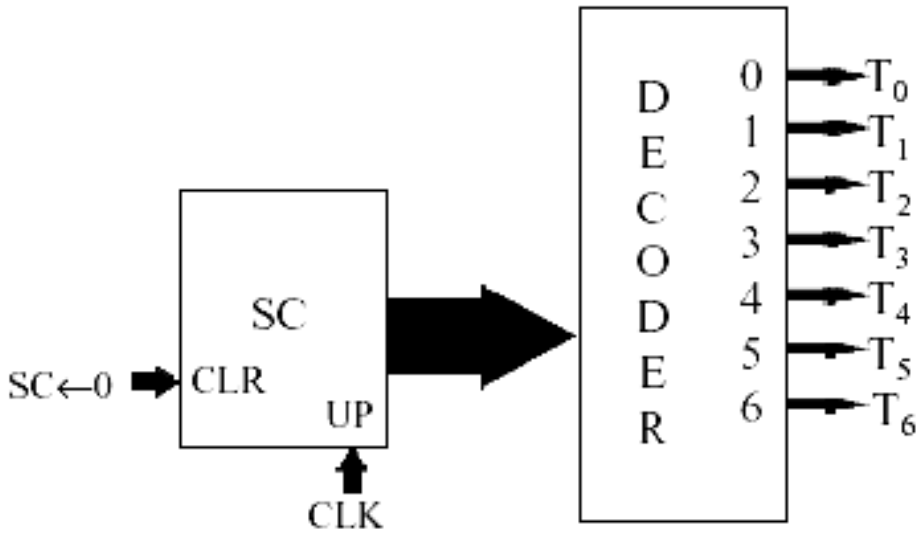
•The T signals occur in sequence and are never skipped over. The only two options during a T-state are to proceed to the next T-state or to return to T state 0.

•The D signals decode the instruction and are used to select the correct execute routine.

•I is used to select the indirect routine and also to select the correct execute routine for non-memory reference instructions.

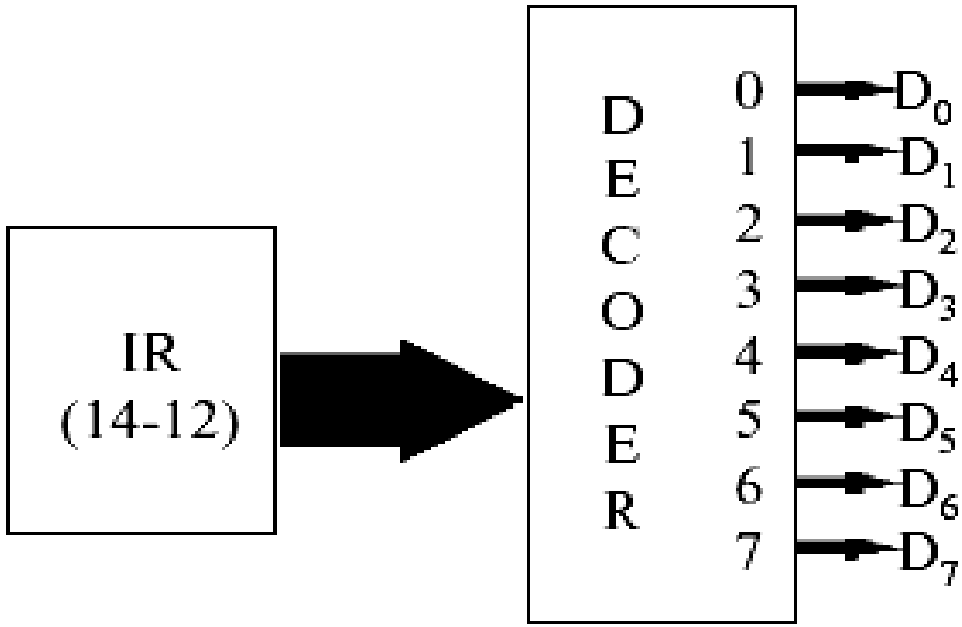
•R is used for interrupt processing and will be explained later.

Control signals



This circuit generates the T signals. The sequence counter, SC, is incremented once per clock cycle. Its outputs are fed into a 3-8 decoder which generates the T signals. Whenever a micro-operation sets SC to zero, it resets the counter, causing T0 to be activated during the next clock cycle.

Control signals



The D signals are generated in a similar way as the T signals.

For the D signals the source is IR(14-12) instead of SC.

Also note that IR won't change during the instruction execution.

Fetch and indirect cycles

Example: Fetch from 100: AND I 500;

M[500] = 234

T0: AR ← 100

T1: IR ← 8500, PC ← 101

T2: AR ← 500, I ← 1, D0 ← 1

D7'IT3: AR ← M[500](11-0) = 234

In this example, the instruction AND I 500 is fetched from memory location 100.

•During T0, the address (100) is loaded into AR.

•During T1, the instruction code is loaded into IR and PC is incremented.

•In T2, the address portion of this instruction, 500, is loaded into AR. The indirect register gets 1, the value of the indirect bit of the instruction. Since bits 14-12 are 000, D0 is activated by the decoder. These tell us that we have an indirect AND instruction.

•In T3, D7 is '0' and I is '1', the address portion of the instruction is not the address of the operand. It is the address of a memory location which contains address of actual operand. Look in memory to get the actual address, 234, which is loaded into AR.

AND execute cycle

AND:

D0T4: DR ← M[AR]

D0T5: AC ← AC ^ DR, SC ← 0

Example: AND 500: AC = 31, M[500] = 25

D0T4: DR ← 25

D0T5: AC ← 31 ^ 25 = 21, SC ← 0

In this and all examples, all data is given in hex. Here, the instruction cycle has fetched the AND instruction, determined that this execute routine must perform and load address 500 into the AR.

•In T4, the data is read from memory and loaded into the DR, 25 in this case. Next,

•in T5, it is logically ANDed with the current contents of the accumulator, 31 here, and the result is stored back into the accumulator. Setting SC to zero returns to the fetch routine to access the next instruction.

ADD execute cycle

ADD:

D1T4: DR ← M[AR]

D1T5: AC ← AC + DR, SC ← 0

Example: ADD 500: AC = 31, M[500] = 25

D1T4: DR ← 25

D1T5: AC ← 31 + 25 = 56, SC ← 0

The ADD operation proceeds similarly to the AND operation. The only difference is that once the operand is loaded from memory it is arithmetically added to the current contents of the accumulator.

LDA execute cycle

LDA:

D2T4: DR ← M[AR]

D2T5: AC ← DR, SC ← 0

Example: LDA 500: M[500] = 25

D2T4: DR ← 25

D2T5: AC ← 25, SC ← 0

As in the previous instructions, the CPU reads the data from memory into DR during T4. In the following cycle, this data is copied into the accumulator.

Since the accumulator only receives data from the adder and logic section, the data from DR is passed into this unit and then passed through it unchanged.

STA execute cycle

STA:

D3T4: M[AR] ← AC, SC ← 0

Example: STA 500: AC = 31, M[500] = 25

D3T4: M[500] ← 31, SC ← 0

The STA instruction is much more straightforward than the LDA instruction.

Since the address is already available from AR to the memory unit, we simply move data directly from the accumulator to the memory unit in a single clock cycle.

BUN execute cycle

BUN:

D4T4: PC ← AR, SC ← 0

Example: BUN 500

D4T4: PC ← 500, SC ← 0

The BUN instruction implements a jump by loading the new address directly from AR into the PC. Unlike many of the other memory reference instructions, BUN receives its data as part of the original instruction and does not require a secondary memory access.

BSA execute cycle

BSA:

D5T4: M[AR] ← PC, AR ← AR+1

D5T5: PC ← AR, SC ← 0

The BSA instruction implements a subroutine call.

•A BSA for address X stores the return address at location X.

•Note that PC was incremented as part of the opcode fetch and thus contains the return address. AR contains X.

•During T4, AR is incremented to X+1, since this is the start of the actual subroutine code.

•T5 loads the value X+1 into the program counter and returns to the fetch routine. Note that this computer cannot implement recursion. If a subroutine called itself, it would overwrite the original return address and would be caught in the subroutine forever! We return from a subroutine by using a BUN I X instruction.

Subroutine call using BSA

Example: 100: BSA 200

D5T4: M[AR] ← PC, AR ← AR+1

M[200] ← 101, AR ← 201

D5T5: PC ← AR, SC ← 0

PC ← 201, SC ← 0

- During T4, the return address, 101, is loaded into memory location 200 and AR is set to 201. This value is the location of the first instruction of the subroutine.
- During T5 it is loaded into the program counter.
- The computer will next fetch the instruction at this location.

Subroutine return using BUN I

Example: 205: BUN I 200

M[200] = 101

D7I'T3: AR ← M[AR](11-0)

AR ← M[200](11-0) = 101

D4T4: PC ← AR, SC ← 0

PC ← 101, SC ← 0

- In this example we perform a return from a previous subroutine.
- After executing a few instructions which comprise the subroutine, we reach the BUN I 200 instruction.
- During the indirect cycle, we go to location 200 to get the actual address we want, in this case 101.
- During T4, we load this value into the program counter, affecting the jump.

ISZ execute cycle

D6T4: DR ← M[AR]

D6T5: DR ← DR+1

D6T6: M[AR] ← DR, SC ← 0,

if (DR=0) then PC ← PC+1

The ISZ instruction is used for program loops. The negative of the count value is stored in some memory location, say X. At the end of the loop, we place the instruction ISZ X. During T4, the Basic Computer copies the contents of memory location X into the data register. This value is incremented during T5, and written back into memory during T6. (AR still contains the memory address at this point.) Also during T6, this value, still available in DR, is check to see if it is zero. If so, PC is incremented, in effect skipping the next instruction.

Loop control using ISZ

Example:100: ISZ 200 M[200] = 55

D6T4: DR ← M[AR] (DR ← 55)

D6T5: DR ← DR+1 (DR ← 56)

D6T6: M[AR] ← DR, SC ← 0,

if (DR=0) then PC ← PC+1

(M[200] ← 56, SC ← 0)

In this example, memory location 200 contains 55, which is loaded into the data register during T4. It is incremented to 56 during T5 and stored back into memory location 200. Since it is not zero, we do not increment the PC.

by: Saurabh Gupta, NIEC, New Delhi

Loops using ISZ

X: *Start of loop*

.

.

ISZ 200

BUN X

Continue on...

Here is an example of how to use the ISZ instruction in a program loop. The loop starts at some location X and does its work. Then we perform the ISZ instruction, which increments the loop counter. If it is not zero, it does not skip the next instruction. It executes that instruction, which branches back to the beginning of the loop. If it is zero, it skips the BUN X instruction, exiting the loop.

Register-reference execute cycles

r = D7I'T3 r: SC ← 0

(CLA) rIR11: AC ← 0

(CLE) rIR10: E ← 0

(CMA) rIR9: AC ← AC

(CME) rIR8: E ← E

(CIR) rIR7: EAC ← cir(EAC)

(CIL) rIR6: EAC ← cil(EAC)

There are 12 register reference instructions, each activated by one of the 12 low order bits of the instruction register. Each register reference instruction is executed in a single clock cycle.

•r: SC <-- 0. This means that SC <-- 0 whenever r=1, regardless of the values of IR(11-0). In short, this is equivalent to adding the micro-operation SC <-- 0 to each of these instructions individually. The CLA and CLE instructions clear AC and E. CMA and CME perform complement. CIR and CIL perform circular right and left shifts

Register-reference execute cycles

r = D7I'T3 r: SC ← 0

(INC) rIR5: EAC ← AC+1

(SPA) rIR4: IF AC(15)=0 THEN PC ← PC+1

(SNA) rIR3: IF AC(15)=1 THEN PC ← PC+1

(SZA) rIR2: IF AC=0 THEN PC ← PC+1

(SZE) rIR1: IF E=0 THEN PC ← PC+1

(HLT) rIR0:HALT

The INC instruction increments AC, storing the result in register pair E/AC. The next four instructions skip an instruction in the program if AC is positive (SPA), AC is negative (SNA), AC is zero (SZA) or E is zero (SZE). Note that SPA actually skips an instruction if AC is not negative, since it also skips an instruction if AC is zero. The HLT instruction shuts down the computer.

I/O configuration

- 8-bit input register INPR
- 8-bit output register OUTR
- 1-bit input flag FGI
- 1-bit input flag FGO
- 1-bit interrupt enable IEN

The Basic Computer has one 8-bit input port and one 8-bit output port. Each port interface is modeled as an 8-bit register which can send data to or receive data from AC(7-0). Whenever input data is to be made available, the external input port writes the data to INPR and sets FGI to 1. When the output port requests data, it sets FGO to 1. As will be shown shortly, the FGI and FGO flags are used to trigger interrupts (if interrupts are enabled by the IEN flag).

I/O execute cycles

p = D7IT3 p: SC ← 0

(INP) pIR11: AC(7-0) ← INPR, FGI ← 0

(OUT) pIR10: OUTR ← AC(7-0), FG0 ← 0

(SKI) pIR9: IF FGI = 1 THEN PC ← PC+1

(SKO) pIR8: IF FGO = 1 THEN PC ← PC+1

(ION) pIR7: IEN ← 1

(IOF) pIR6: IEN ← 0

Once data is made available to the CPU, it can be read in using the INP instruction. Note that this not only reads the data into the accumulator, but also resets FGI to zero. This tells the input port that it may send more data. In a similar manner, the OUT instruction writes data to OUTR and resets FGO to zero, notifying the output port that data is available. The SKI and SKO instructions skip an instruction if there is a pending input or output request. This is useful in determining the I/O request which caused an interrupt to occur. ION and IOF enable and disable interrupts. Interrupts will be explained more fully shortly.

Input operation

- **Input device makes data available and sets FGI=1.**
- ***If interrupt is enabled*, Basic Computer calls interrupt routine at location 0, which disables further interrupts.**
- **Interrupt routine reads in and processes data, re-enables interrupts and returns. Reading in data resets FGI to zero.**

In the Basic Computer, I/O requests are processed as interrupts. This process is followed for input requests. The input will only be processed if interrupts are enabled. It will be ignored, but will remain pending, if interrupts are disabled.

Output operation

- **Output device requests data and sets FGO=1.**
- ***If interrupt is enabled*, Basic Computer calls interrupt routine at location 0, which disables further interrupts.**
- **Interrupt routine processes and outputs data, re-enables interrupts and returns. Writing out data resets FGO to zero.**

Outputs are handled similarly to inputs. Note that both input and output interrupts call an interrupt service routine at location 0. There is only one routine for both input and output, so it must distinguish between the two. This is where the SKI and SKO instructions become useful.

Interrupt processing

- **An interrupt occurs if the interrupt is enabled (IEN = 1) AND an interrupt is pending (FGI or FGO = 1).**
- **Before processing the interrupt, *complete the current instruction!!!***
- **Call the interrupt service routine at address 0 and disable interrupts.**

It is of the utmost importance to complete the current instruction, otherwise the CPU will not perform properly.

The interrupt service routine is called by the CPU in a manner similar to the execution of the BSA instruction.

Interrupt cycle

Activating an interrupt request:

T0' T1' T2'(IEN)(FGI + FGO): R ← 1

Interrupt cycle:

RT0: AR ← 0, TR ← PC

RT1: M[AR] ← TR, PC ← 0

RT2: PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0

An interrupt is asserted by setting R to 1. This occurs when interrupts are enabled (IEN) and there is either an input or output request (FGI+FGO). We must also have completed the current fetch cycle (T0'T1'T2').

When we look at the code to implement the interrupt cycle, we see why we must wait until after T2 to set R to 1. If we set R to 1 during T0, for example, the next micro-instruction would be RT1, right in the middle of the interrupt cycle. Since we want to either perform an entire opcode fetch or an entire interrupt cycle, we don't set R until after T2.

The interrupt cycle acts like a BSA 0 instruction.

•During T0 we write a 0 into AR and copy the contents of PC, the return address, to TR.

•We then store the return address to location 0 and clear the program counter during T1.

•In T2, we increment PC to 1, clear the interrupt enable, set R to zero (because we've finished the interrupt cycle) and clear SC to bring us back to T0.

Note that IEN is set to 0. This disables further interrupts. If another interrupt occurred while one was being serviced, the second interrupt would write its return address into location 0, overwriting the interrupt return address of the original routine. Then it would not be possible to return to the program properly.

Modified fetch cycle

R'T0: AR ← PC

R'T1: IR ← M[AR], PC ← PC+1

R'T2: AR ← IR(11-0), I ← IR15,

D0, D1, ... D7 ← Decode IR(14-12)

This is exactly the same as before, but R' insures that no interrupts must be serviced.

5-9 Design of Basic Computer

- Two basic things are needed: data paths and control signals
- A hardwired-control implementation: Stitch together the individual pieces of the data path.
- The microoperation table provides sufficient information to implement the circuits for control (wiring various gates).

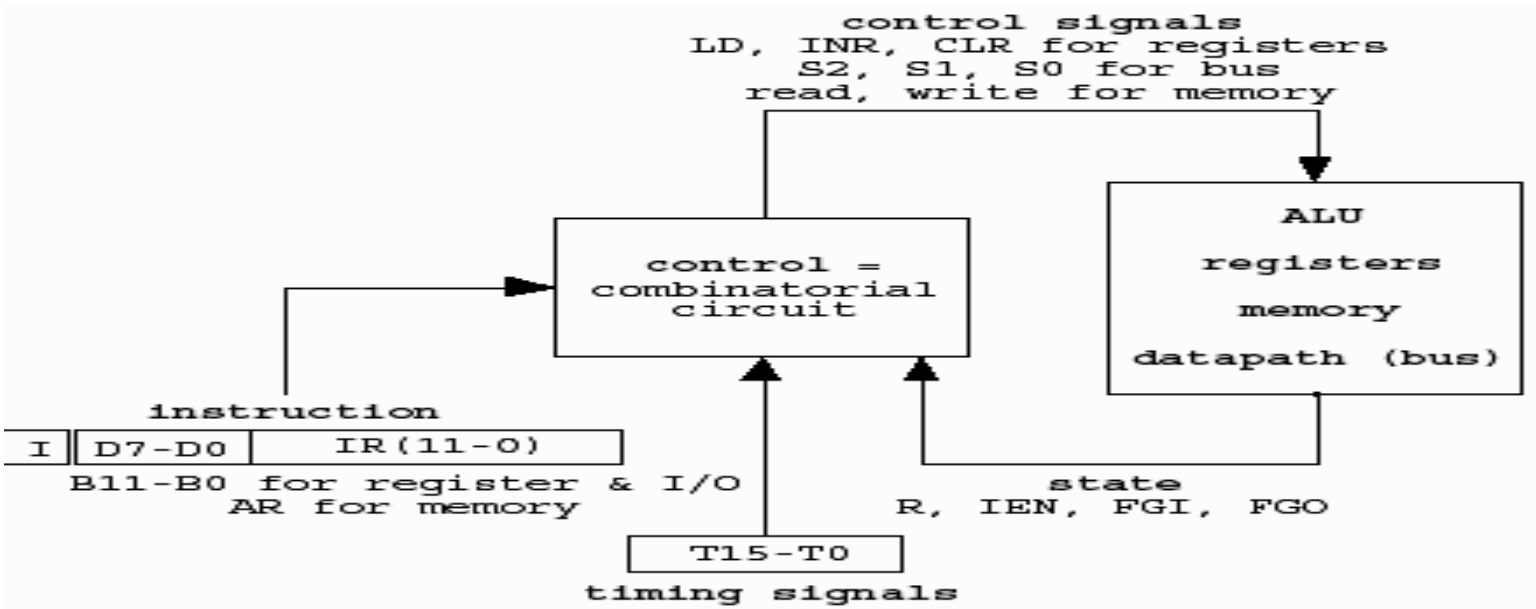


Figure: Where the control resides in the computer.

Input:

1. D0 - D7: Decoded IR(14-12)
2. T0 - T15 : Timing signals
3. I: Indirect signal
4. IR(0-11)

Output:

1. Control inputs of the nine registers, AR, PC, DR, AC, IR, TR, OUTR, INPR, SC
2. Read and write inputs of memory
3. Signals to set, clear, or complement the flip-flops, IEN, R, etc.
4. Select signals, S2, S1, S0, for common bus
5. Control signals to the AC adder and logic circuit

CONTROL OF REGISTERS AND MEMORY

Systematic Design Procedure

1. For a given register, scan the table of microoperations in the previous slides to find all the statements involving that gate.
2. Translate the associated control functions to Boolean functions.
3. Convert the Boolean expressions into logic gates.

Example: Control of AR

1. The following is the summary of the register transfers associated with the address register.

R'T0: AR ← PC load

R'T2: AR ← IR(0-11) load

D7'IT3: AR ← M[AR] load

RT0: AR ← 0 clear

D5T4: AR ← AR + 1 increment

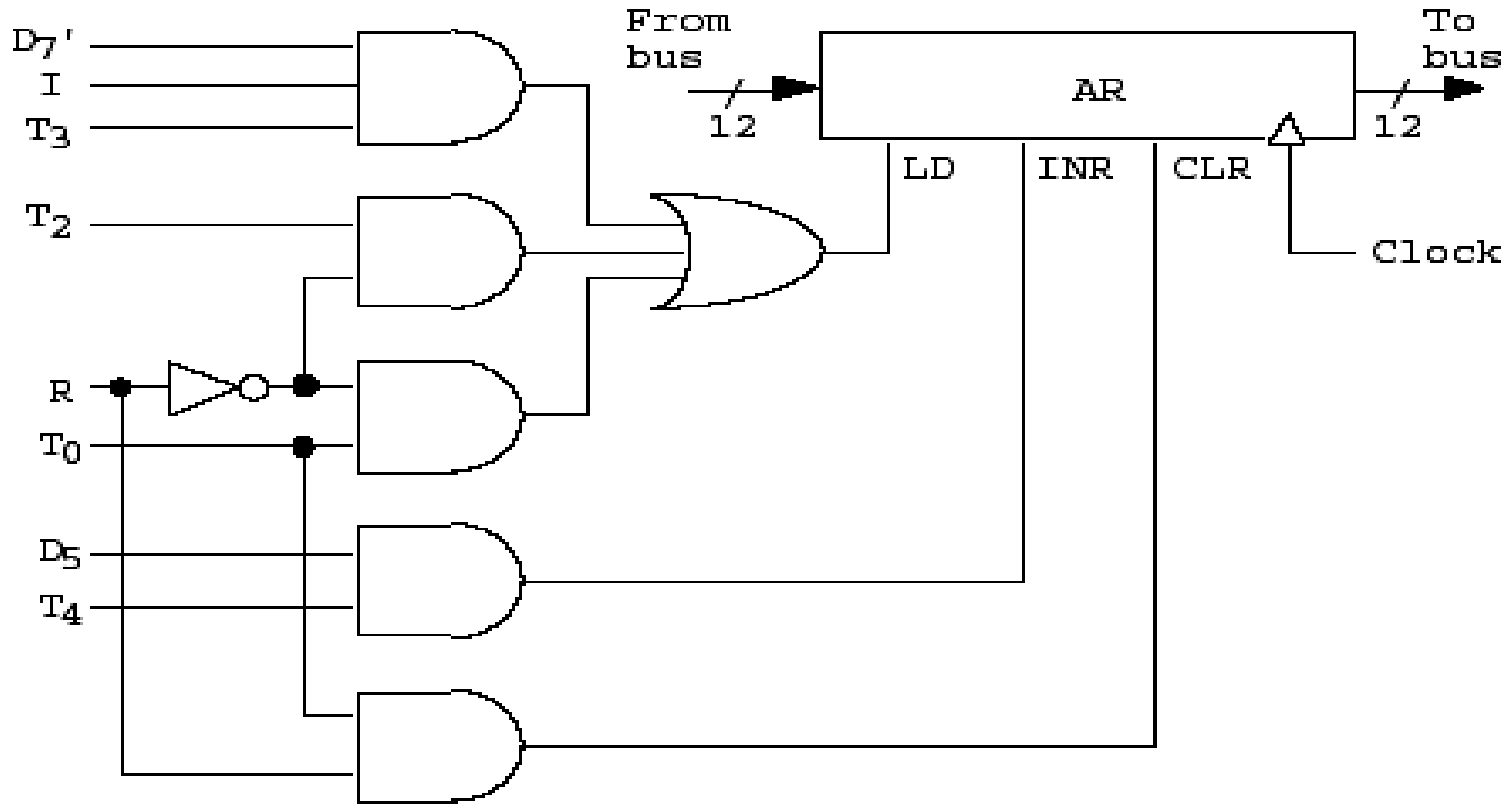
2. The control functions can be combined into the following Boolean expressions.

LD(AR) = R'T0 + R'T2 + D7'IT3

CLR(AR) = RT0

INR(AR) = D5T4

3. The previous Boolean expressions can be converted to the following logic gates.



• In a similar fashion, the control gates for the other registers and memory can be derived. For example, the logic gates associated with the read input of memory is derived by scanning the microoperation table to find the statements that specify a read operation. The read operation is recognized from the symbol ← M[AR].

Read = R'T1 + D7'IT3 + (D0 + D1 + D2 + D3)T4

5-9 Design of Basic Computer

- Register Control : AR
  - Control inputs of AR : ***LD, INR, CLR***
  - ***Find all the statements that change the AR in Table. 5-6***
  - Control functions

$AR \leftarrow ?$

$R'T_0 : AR \leftarrow PC$

$R'T_1 : AR \leftarrow IR(0-11)$

$D_7'IT_3 : AR \leftarrow M[AR]$

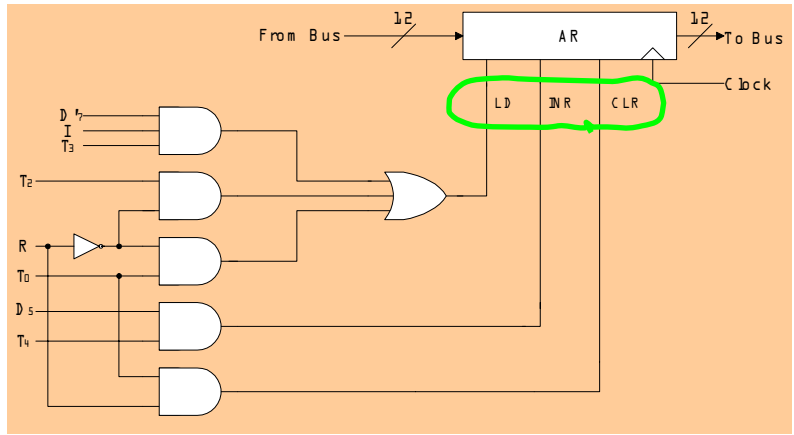
$RT_0 : AR \leftarrow 0$

$D_5T_4 : AR \leftarrow AR + 1$

$LD(AR) = R'T_0 + R'T_1 + D_7'IT_3$

$CLR(AR) = RT_0$

$INR(AR) = D_5T_4$



- Memory Control : READ
  - Control inputs of Memory : ***READ, WRITE***
  - Find all the statements that specify a ***read operation*** in Table. 5-6
  - Control function

$M[AR] \leftarrow ?$

$? \leftarrow M[AR]$

$READ = R'T_1 + D_7'IT_3 + (D_0 + D_1 + D_2 + D_3)T_4$

- F/F Control : IEN
  - Control functions

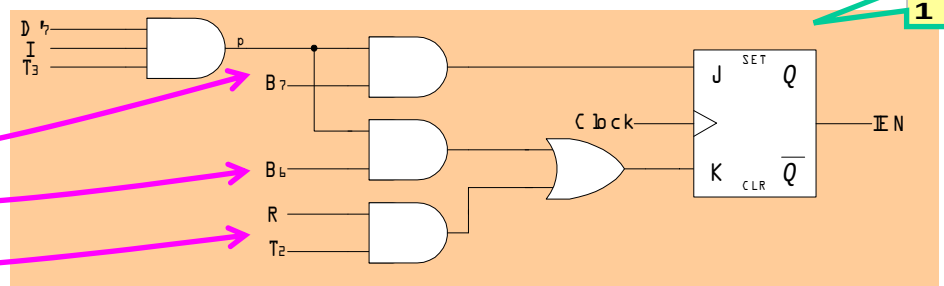
$IEN \leftarrow ?$

$pB_7 : IEN \leftarrow 1$

$pB_6 : IEN \leftarrow 0$

$RT_2 : IEN \leftarrow 0$

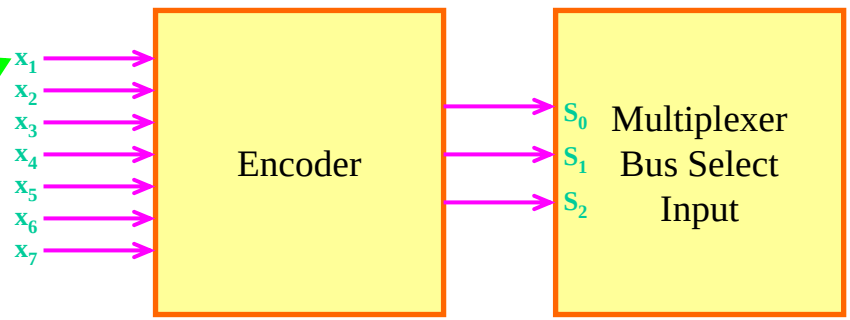
J	K	Q(t+1)
0	1	0
1	0	1



5-9. Design of Basic Computer

- Bus Control
  - Encoder for Bus Selection : *Table. 5-7*
    - $S_0 = x_1 + x_3 + x_5 + x_7$
    - $S_1 = x_2 + x_3 + x_6 + x_7$
    - $S_0 = x_4 + x_5 + x_5 + x_7$
  - $x_1 = 1$: $Bus \leftarrow AR = Find\ ? \leftarrow AR$
    - $D_4T_4 : PC \leftarrow AR$
      $D_5T_5 : PC \leftarrow AR$
    - Control Function : $x_1 = D_4T_4 + D_5T_5$
  - $x_2 = 1$: $Bus \leftarrow PC = Find\ ? \leftarrow PC$
    - “
      $Bus \leftarrow Memory = Find\ ? \leftarrow M[AR]$
      “
  - $x_7 = 1$: $x_7 = R'T_1 + D_7'IT_3 + (D_0 + D_1 + D_2 + D_3)T_4$
    - Same as Memory Read
    - Control Function :
    -

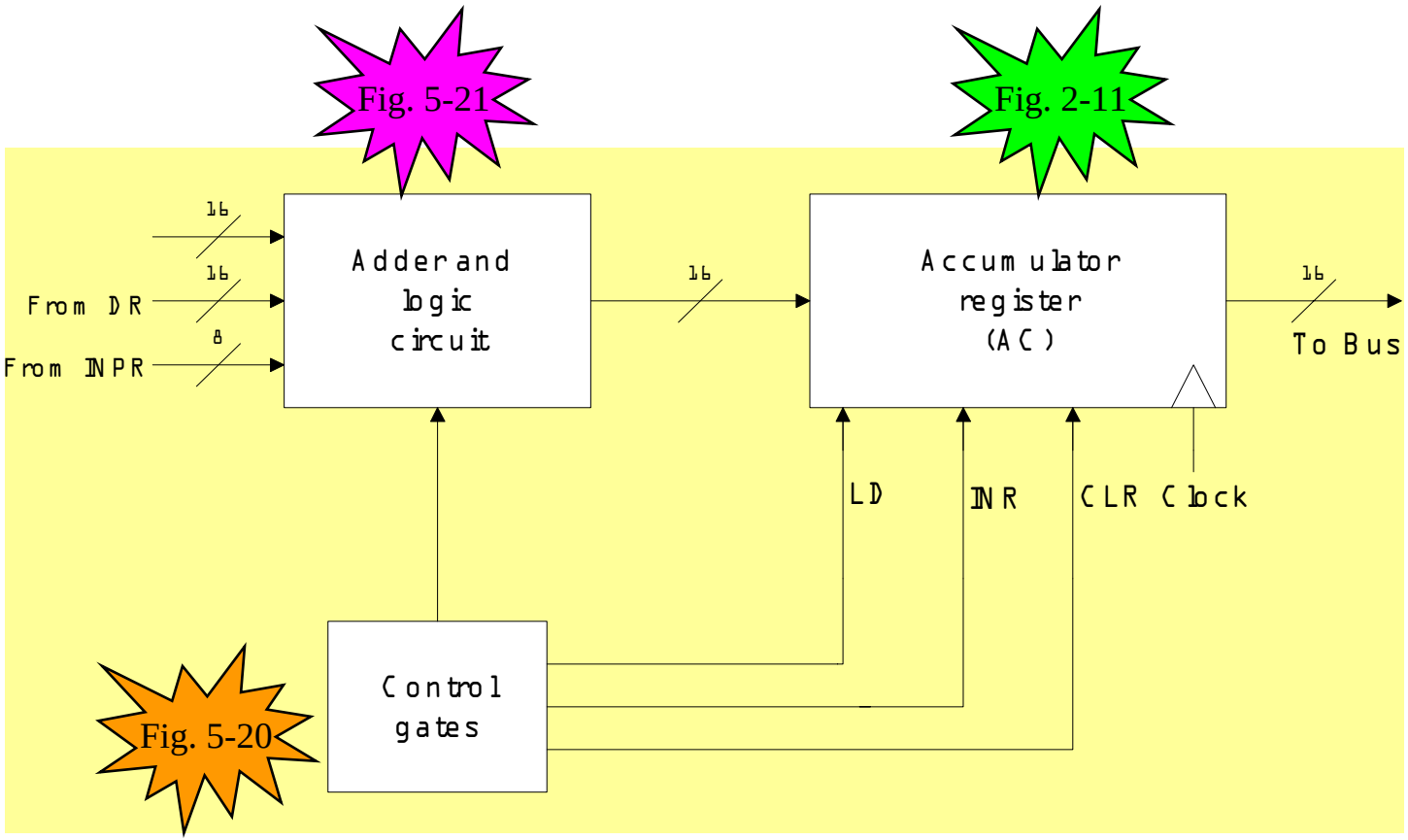
x1 = 1 corresponds to the bus connection of AR as a source



Inputs							Outputs			Register selected for bus
x1	x2	x3	x4	x5	x6	x7	S2	S1	S0	
0	0	0	0	0	0	0	0	0	0	None
1	0	0	0	0	0	0	0	0	1	AR
0	1	0	0	0	0	0	0	1	0	PC
0	0	1	0	0	0	0	0	1	1	DR
0	0	0	1	0	0	0	1	0	0	AC
0	0	0	0	1	0	0	1	0	1	IR
0	0	0	0	0	1	0	1	1	0	TR
0	0	0	0	0	0	1	1	1	1	Memory

5-10. Design of Accumulator Logic

- 5-10 Design of Accumulator Logic
  - Circuits associated with AC : ***Fig. 5-19***



5-10. Design of Accumulator Logic

Control of AC : ***Fig. 5-20***

- Find the statement that change the AC : $AC \leftarrow ?$

$D_0T_5 : AC \leftarrow AC \wedge DR$

$D_1T_5 : AC \leftarrow AC + DR$

$D_2T_5 : AC \leftarrow DR$

$pB_{11} : AC(0-7) \leftarrow INPR$

$rB_9 : AC \leftarrow \overline{AC}$

$rB_7 : AC \leftarrow shr\ AC, AC(15) \leftarrow E$

$rB_6 : AC \leftarrow shr\ AC, AC(0) \leftarrow E$

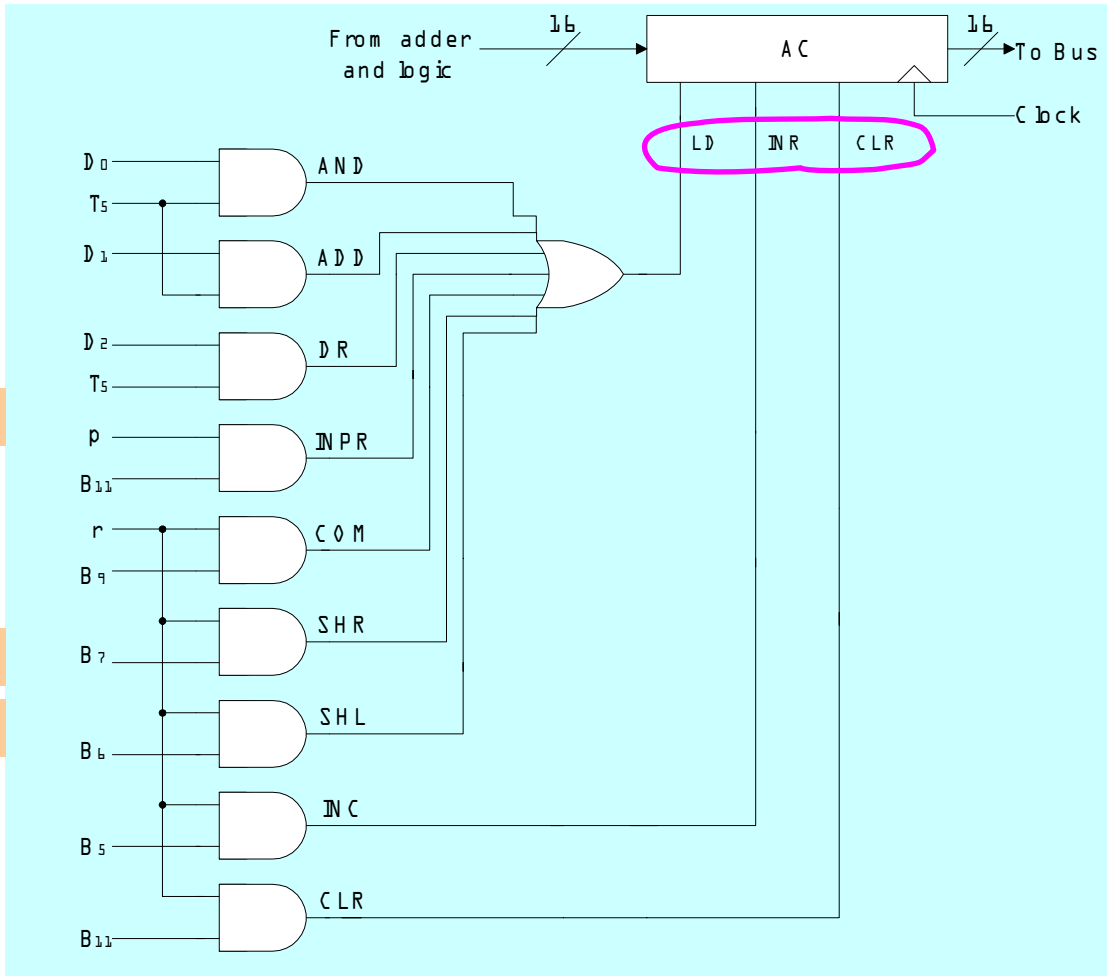
$rB_{11} : AC \leftarrow 0$

$rB_5 : AC \leftarrow AC + 1$

LD

CLR

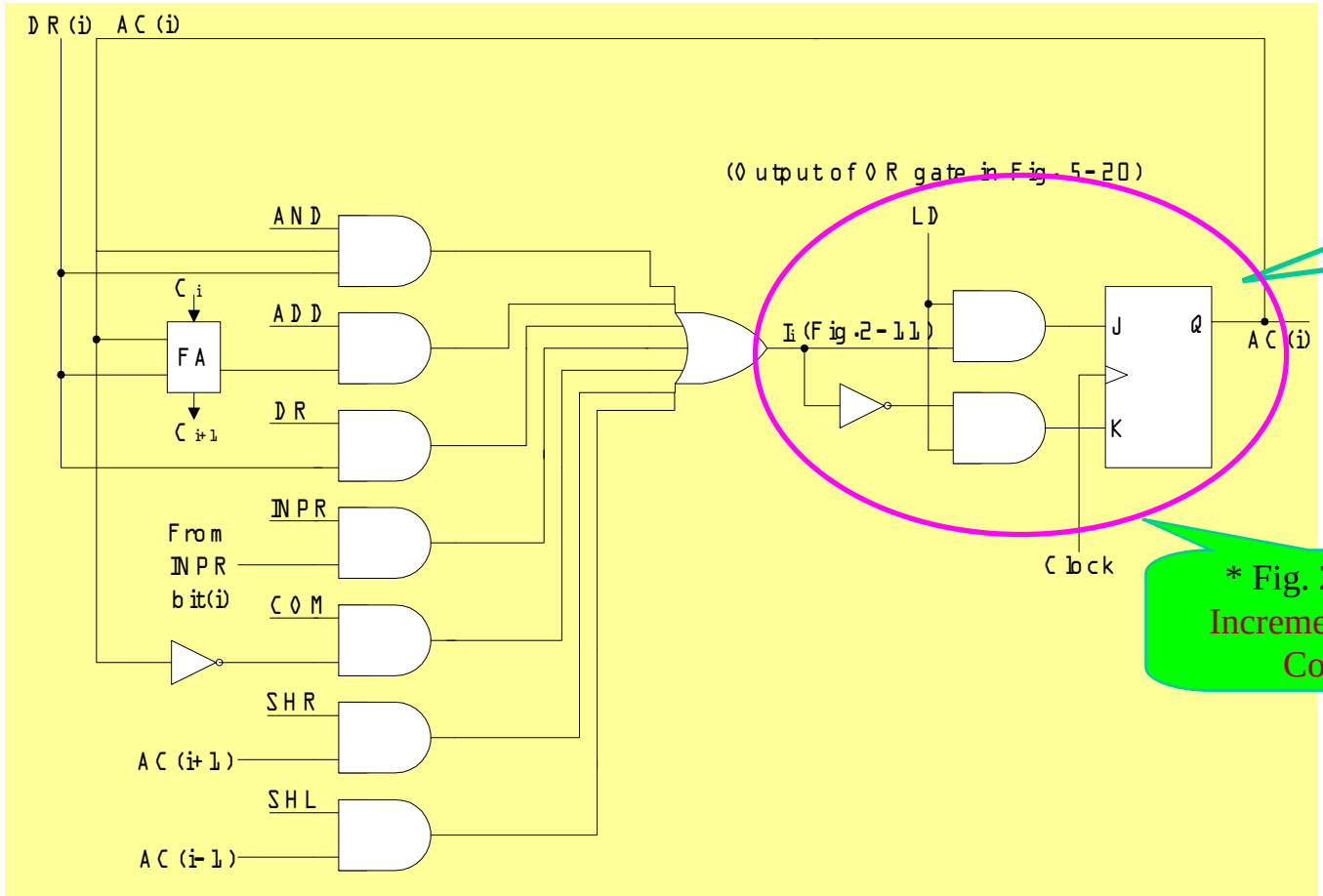
INR



by: Saurabh Gupta, NIEC, New Delhi

5-10. Design of Accumulator Logic

– Adder and Logic Circuit : ***Fig. 5-21 (16 bit)***



Summary of Mano's Computer

Integration !

- Fig. 5-4 : Common Bus *(p.130)*
- Fig. 2-11 : Register *(p. 59)*
- Fig. 5-6 : Control Unit *(p. 137)*
- Fig. 5-16, 17,18 : Control Logic Gates *(p.161- 163)*
  - Fig. 5-4 Component Control Input
  - Register, Memory, F/Fs, Bus Selection
- Fig. 5-20 : AC control *(p.165)*
- Fig. 5-21 : Adder and Logic *(p.166)*

by: Saurabh Gupta, NIEC, New Delhi

Chapter 7

Section 7.1 – Control Memory

- The control unit in a digital computer initiates sequences of microoperations
- The complexity of the digital system is derived form the number of sequences that are performed
- When the control signals are generated by hardware, it is *hardwired*
- In a bus-oriented system, the control signals that specify microoperations are groups of bits that select the paths in multiplexers, decoders, and ALUs.

- The control unit initiates a series of sequential steps of microoperations
- The control variables can be represented by a string of 1's and 0's called a *control word*
- A *microprogrammed control unit* is a control unit whose binary control variables are stored in memory
- A sequence of microinstructions constitutes a *microprogram*
- The control memory can be a read-only memory
- *Dynamic* microprogramming permits a microprogram to be loaded and uses a writable control memory

- A computer with a microprogrammed control unit will have two separate memories: a main memory and a control memory
- The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations
- These microinstructions generate the microoperations to:
    - fetch the instruction from main memory
    - evaluate the effective address
    - execute the operation
    - return control to the fetch phase for the next instruction

- The control memory address register specifies the address of the microinstruction
- 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

- The location for the next microinstruction may, or may not be the next in sequence
- Some bits of the present microinstruction 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 (sequencer) and then transferred into the CAR to read the next microinstructions
- Typical functions of a sequencer are:

- incrementing the CAR by one
- loading into the CAR and address from control memory
- transferring an external address
- loading an initial address to start the control operations

- A clock is applied to the CAR and the control word and next-address information are taken directly from the control memory
- The address value is the input for the ROM and the control work is the output
- No read signal is required for the ROM as in a RAM

- The main advantage of the microprogrammed control is that once the hardware configuration is established, there should be no need for h/w or wiring changes
- To establish a different control sequence, specify a different set of microinstructions for control memory

Section 7.2 – Address Sequencing

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

- Steps the control must undergo during the execution of a single computer instruction:
  - Load an initial address into the CAR when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine – IR holds instruction
  - The control memory then goes through the routine to determine the effective address of the operand – AR holds operand address
  - The next step is to generate the microoperations that execute the instruction by considering the opcode and applying a *mapping*
  - After execution, control must return to the fetch routine by executing an unconditional branch

- 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

- Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition

- 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 i/o status conditions
- The status bits, together with the field in the microinstruction that specifies a branch address, control the branch logic
- The branch logic tests the condition, if met then branches, otherwise, increments the CAR
- If there are 8 status bit conditions, then 3 bits in the microinstruction are used to specify the condition and provide the selection variables for the multiplexer
- For unconditional branching, fix the value of one status bit to be one load the branch address from control memory into the CAR

- A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine is located
- The status bits for this type of branch are the bits in the opcode
- Assume an opcode of four bits and a control memory of 128 locations
- The mapping process converts the 4-bit opcode to a 7-bit address for control memory
- This provides for each computer instruction a microprogram routine with a capacity of four microinstructions

- Subroutines are programs that are used by other routines to accomplish a particular task and can be called from any point within the main body of the microprogram
- Frequently many microprograms contain identical section of code
- Microinstructions can be saved by employing subroutines that use common sections of microcode
- Microprograms that use subroutines must have a provisions for storing the return address during a subroutine call and restoring the address during a subroutine return
- A subroutine register is used as the source and destination for the addresses

Section 8.4 – Instruction Formats

- It is the function of the control unit within the CPU to interpret each instruction code
- The bits of the instruction are divided into groups called fields
- The most common fields are:
  - Operation code
  - Address field – memory address or a processor register
  - Mode field – specifies the way the operand or effective address is determined
- A register address is a binary number of *k* bits that defines one of 2^k registers in the CPU

- The instructions may have 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 the three following organizations:
  - Single accumulator organization
  - General register organization
  - Stack organization

- Single accumulator org. uses one address field
  ADD X : AC ← AC + M[X]
- The general register org. uses three address fields
  ADD R1, R2, R3: R1 ← R2 + R3
- Can use two rather than three fields if the destination is assumed to be one of the source registers
- Stack org. would require one address field for PUSH/POP operations and none for operation-type instructions
  PUSH X
  ADD
- Some computers combine features from more than one organizational structure

Example: X = (A+B) * (C + D)

Three-address instructions:

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

Two-address instructions:

MOV	R1, A	R1 ← M[A]
ADD	R1, B	R1 ← R1 + M[B]
MOV	R2, C	R2 ← M[C]

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

One-address instructions:

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

Zero-address instructions:

```
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
```

RISC instructions:

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

Section 8.5 – Addressing Modes

- The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced
- The decoding step in the instruction cycle determines the operation to be performed, the addressing mode of the instruction, and the location of the operands
- Two addressing modes require no address fields – the implied mode and immediate mode
- Implied mode: the operands are specified implicitly in the definition of the instruction – complement accumulator or zero-address instructions

- Immediate mode: the operand is specified in the instruction
- Register mode: the operands are in registers
- Register indirect mode: the instruction specifies a register that contains the address of the operand
- Autoincrement or autodecrement mode: similar to the register indirect mode
- Direct address mode: the operand is located at the specified address given
- Indirect address mode: the address specifies the effective address of the operand
- Relative address mode: the effective address is the summation of the address field and the content of the PC
- Indexed addressing mode: the effective address is the summation of an index register and the address field
- Base register address mode: the effective address is the summation of a base register and the address field

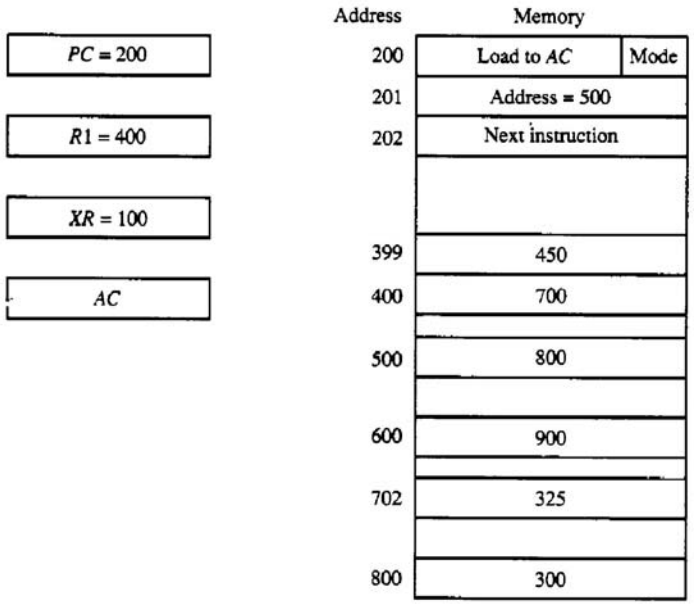


Figure 8-7 Numerical example for addressing modes.