Computer Architecture

Lecture 1:

Digital logic circuits

The digital computer is a digital system that performs various computational tasks. Digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a bit.

A computer system is sometimes subdivided into two functional entities: hardware and software.

The hardware of the computer consists of all the electronic components and electromechanical devices that comprise the physical entity of the device.

Computer software consists of the instructions and data that the computer manipulates to perform various data-processing tasks. Program A sequence of instructions for the computer is called a program. The data that are manipulated by the program constitute the data base.

Computer organization is concerned with the way the hardware components operate and the way they are connected together to form the computer system. The various components are assumed to be in place and the task is to investigate the organizational structure to verify that the computer parts operate as intended.

Computer design is concerned with the hardware design of the computer. Once the computer specifications are formulated, it is the task of the designer to develop hardware for the system. Computer design is concerned with the determination of what hardware should be used and how the parts should be connected. This aspect of computer hardware is sometimes referred to as computer implementation.

Computer architecture is concerned with the structure and behavior of the computer as seen by the user. It includes the information formats, the instruction set, and techniques for addressing memory. The architectural design of a computer system is concerned with the specifications of the various functional modules, such as processors and memories, and structuring them together into a computer system.

Logic Gates:

Gates are blocks of hardware that produce signals of binary 1 or 0 when input logic requirements are satisfied.

Boolean Algebra:

Boolean algebra is an algebra that deals with binary variables and logic operations. The variables are designated by letters such as A, B, x, and y. The three

Boolean function basic logic operations are AND, OR, and complement. A Boolean function can be expressed algebraically with binary variables, the logic operation symbols, parentheses, and equal sign. For a given value of the variables, the Boolean function can be either 1 or 0. Consider, for example, the Boolean function F = x + y'z

Complement of a Function:

(a)f = ABC + ABC + A'C

(B)F = AB + A'C

Figure 1-6 Two logic diagrams for the same Boolean function.

Complement of a Function

From the general DeMorgan's theorem we can derive a simple procedure for obtaining the complement of an algebraic expression. This is done by changing all OR operations to AND operations and all AND operations to OR operations and then complementing each individual letter variable. As an example, consider the following expression and its complement:

F = AB + CD' + B'D

F' = (A' + B')(C + D)(B + D')

Combinational Circuits

A combinational circuit is a connected arrangement of logic gates with a set of inputs and outputs.

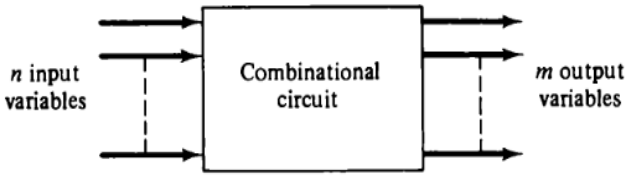


Figure 1-15 Block diagram of a combinational circuit.

Full-Adder:

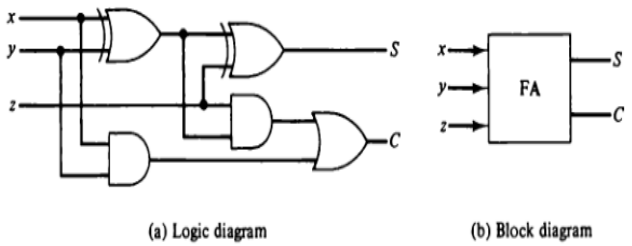
C = xy + (x'y + xy')z

Realizing that x'y + xy' = x(By and including the expression for output S, we obtain the two Boolean expressions for the full-adder:

S=x®y®z

C = xy + (x®y)z

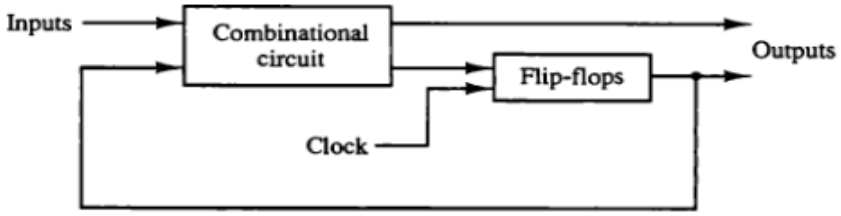
Inputs			Outputs	
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



(a) Logic diagram (b) Block diagram

Sequential Circuits:

A sequential circuit is an interconnection of flip-flops and gates.



Digital Components:

Integrated Circuits

An integrated circuit IC (abbreviated IC) is a small silicon semiconductor crystal, called a chip, containing the electronic components for the digital gates.

Decoders:

A decoder is a combinational circuit that converts binary information from the n coded inputs to a maximum of 2" unique outputs.

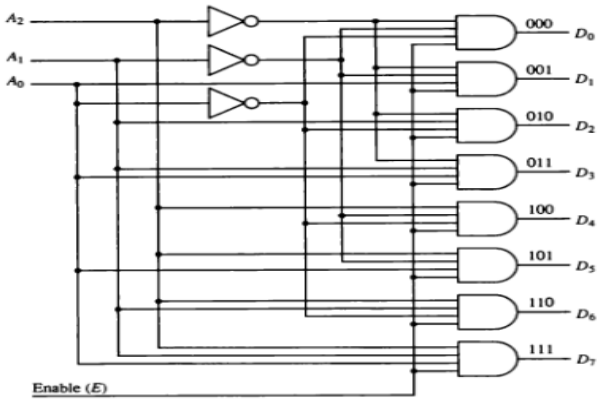


Figure 3-8 decoder.

Encoders:

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2" (or less) input lines and n output lines.

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

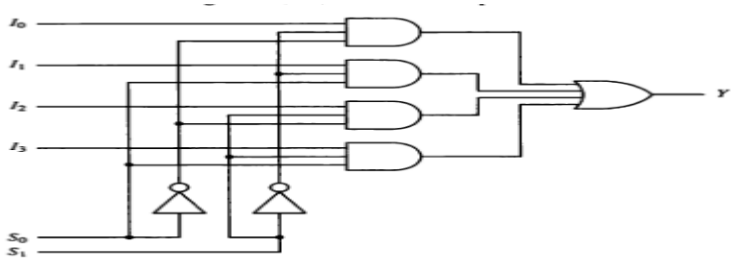
These conditions can be expressed by the following Boolean functions:

A0 = D, + D3 + D5 + D7

A1 = D2 + D3 + D6 + D7

A2 = D4 + D5 + D6 + D7

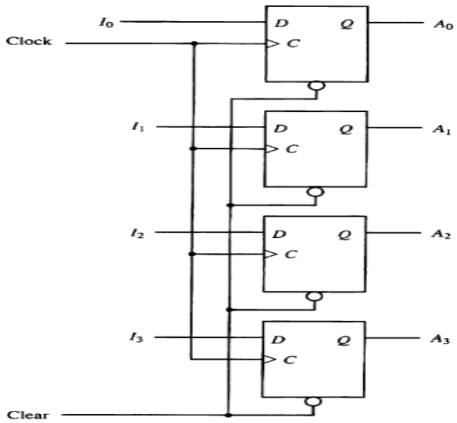
Multiplexer: is a combinational circuit that receives binary information from one of 2" input data lines and directs it to a single output line.



Registers

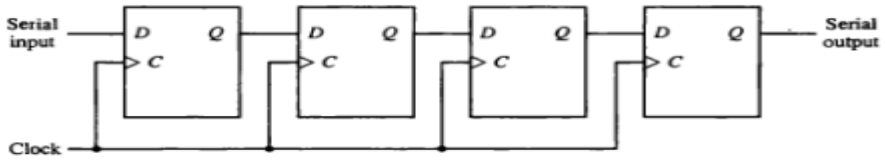
register is a group of flip-flops with each flip-flop capable of storing one bit of information. An n-bit register has a group of n flip-flops and is capable of storing any binary information of n bits.

Ex: 4 bit register



Shift Registers

A register capable of shifting its binary information in one or both directions is called a shift register.



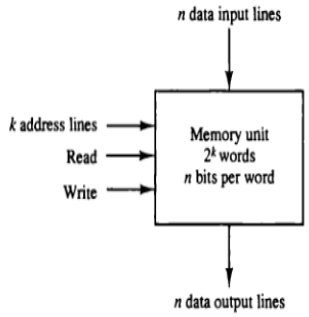
4 bit shift register

Memory Unit

A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of storage. The memory stores binary word information in groups of bits called words.

Random-Access Memory

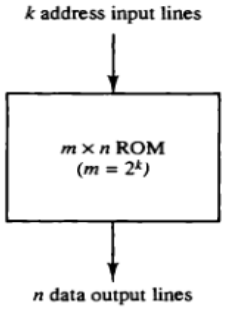
In random-access memory (RAM) the memory cells can be accessed for information transfer from any desired random location.



Block diagram of random access memory (RAM).

Read-Only memory

the name implies, a read-only memory (ROM) is a memory unit that performs the read operation only; it does not have a write capability.



Block diagram of read only memory (ROM).

Lecture 2:

Register Transfer and Microoperations

-Register Transfer Language

Digital system: is an interconnection of digital hardware modules that accomplish a specific information-processing task.

The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic. The various 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.

The internal hardware organization of a digital computer is best defined by specifying:

1. The set of registers it contains and their function.

2. The sequence of microoperations performed on the binary information stored in the registers.

3. The control that initiates the sequence of microoperations.

Registers transfer language:

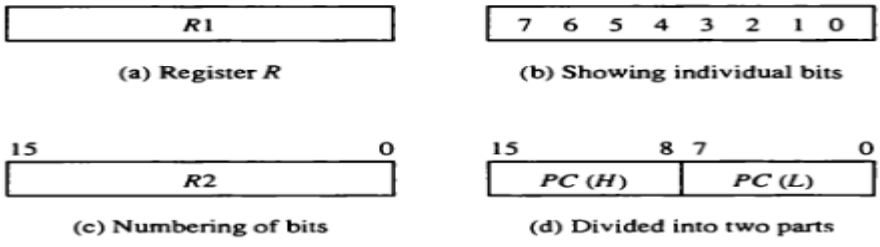
The symbolic notation used to describe the microoperation transfers register transfer among registers is called a register transfer language. The term “registers transfer" implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register.

Register Transfer

Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement R2 ← Rl

denotes a transfer of the content of register Rl into register R2. It designates a replacement of the content of R2 by the content of Rl. By definition, the content of the source register Rl does not change after the transfer.

Block diagram of register.



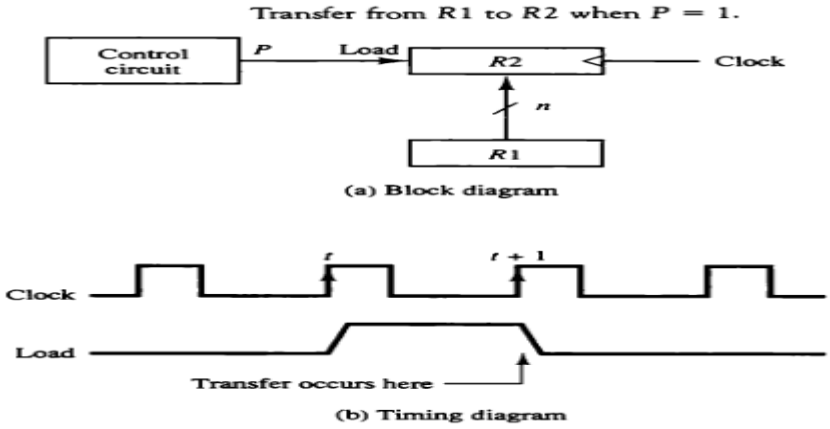
If we want the transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement.

If (P = 1) then (R2 ← Rl)

Control function

It is sometimes convenient to separate the control variables from the register transfer operation by specifying a control function.

P: R2 ←Rl

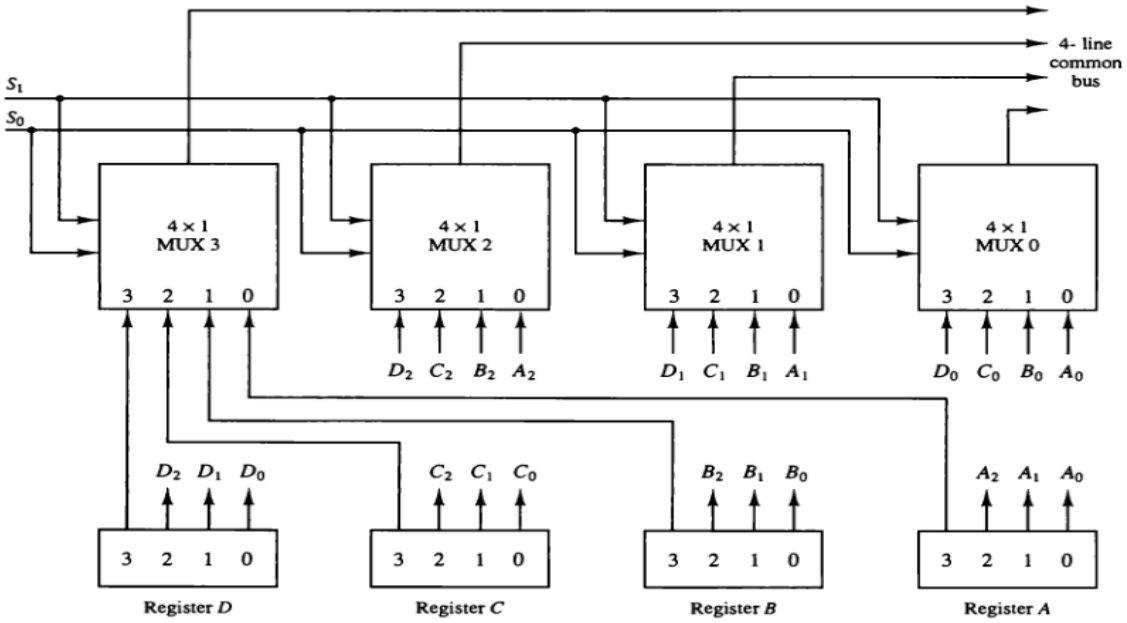


Basic Symbols for Register Transfers

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*

Bus and Memory Transfers

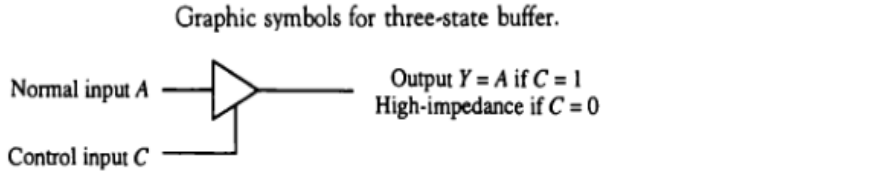
A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between common bus registers in a multiple-register configuration is a common bus system.

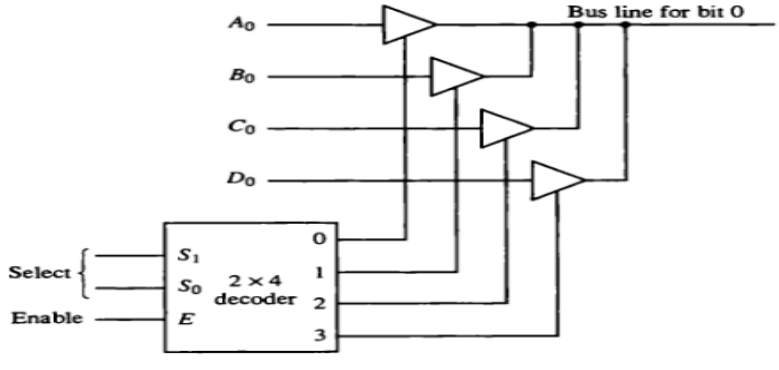


Function Table for Bus

S_1	S_0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

Three-State Bus Buffers:





Bus line with three state-buffers.

Memory Transfer

Read: DR ←M[AR]

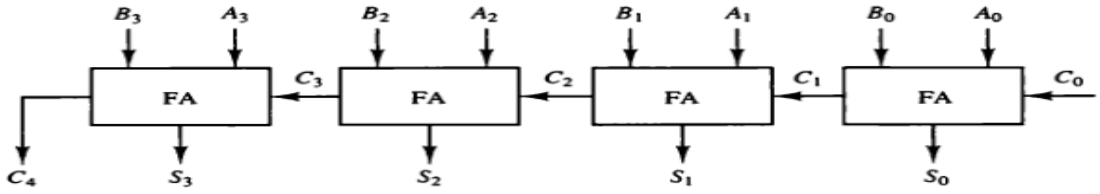
Arithmetic Microoperations

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R3$
$R3 \leftarrow R1 - R2$	Contents of $R1$ minus $R2$ transferred to $R3$
$R2 \leftarrow \overline{R2}$	Complement the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of $R2$ (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus the 2's complement of $R2$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ by one
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ by one

Lecture 3:

Binary Adder

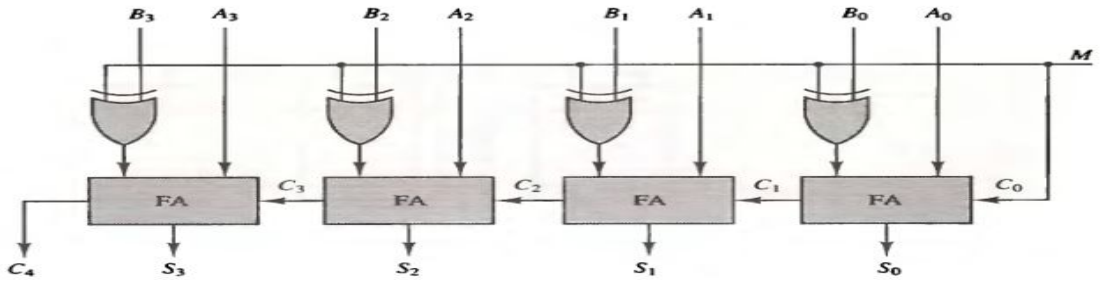
To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the arithmetic addition.



4-bit binary adder.

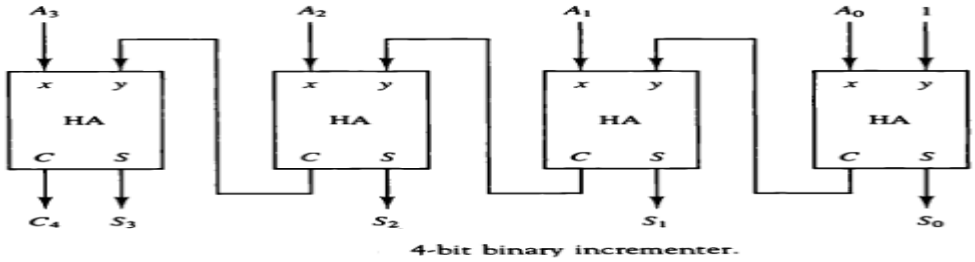
Binary Adder-Subtractor

The subtraction of binary numbers can be done most conveniently by means of complements



4-bit adder-subtractor.

Binary Incrementer

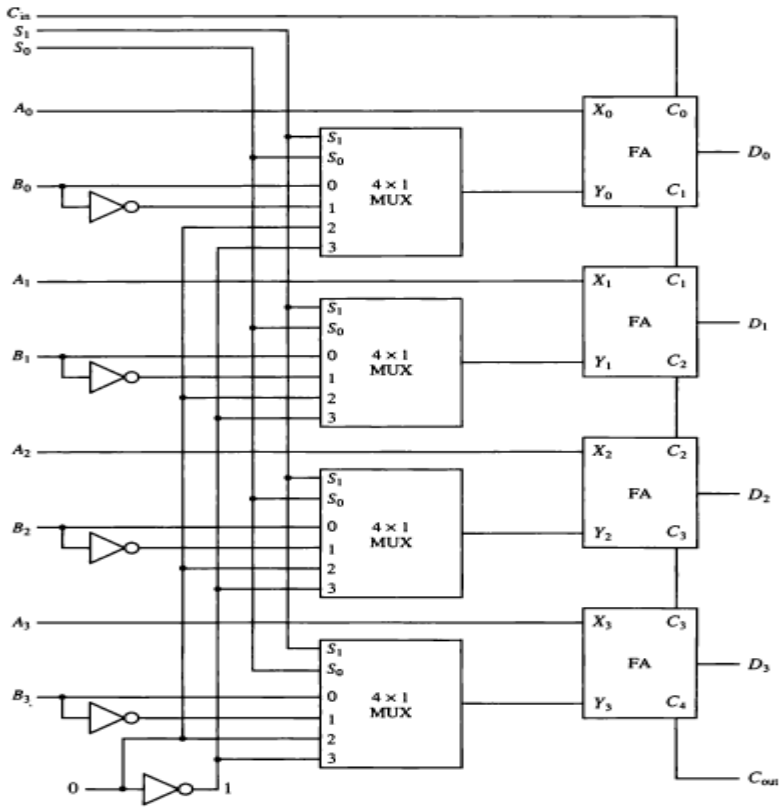


4-bit binary incrementer.

Arithmetic Circuit

The output of the binary adder is calculated from the following arithmetic sum:

D = A + Y + Cin



4-bit arithmetic circuit.

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

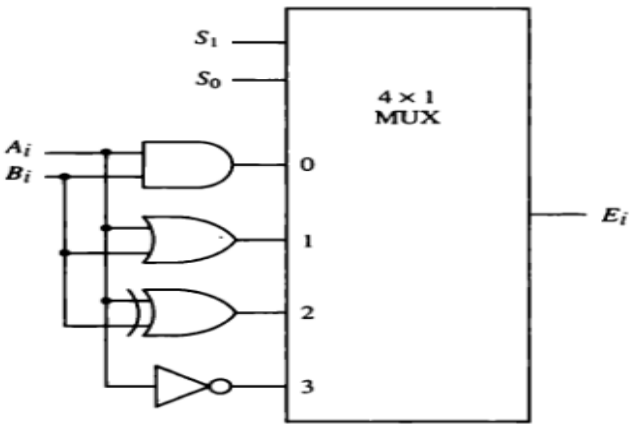
Logic Microoperations

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

Hardware Implementation

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

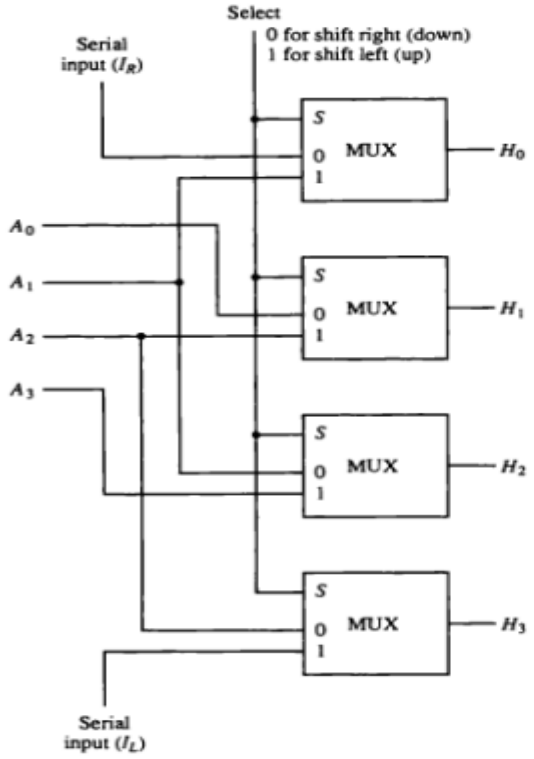
(b) Function table

Shift Microoperations

Shift Microoperations

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

Hardware Implementation

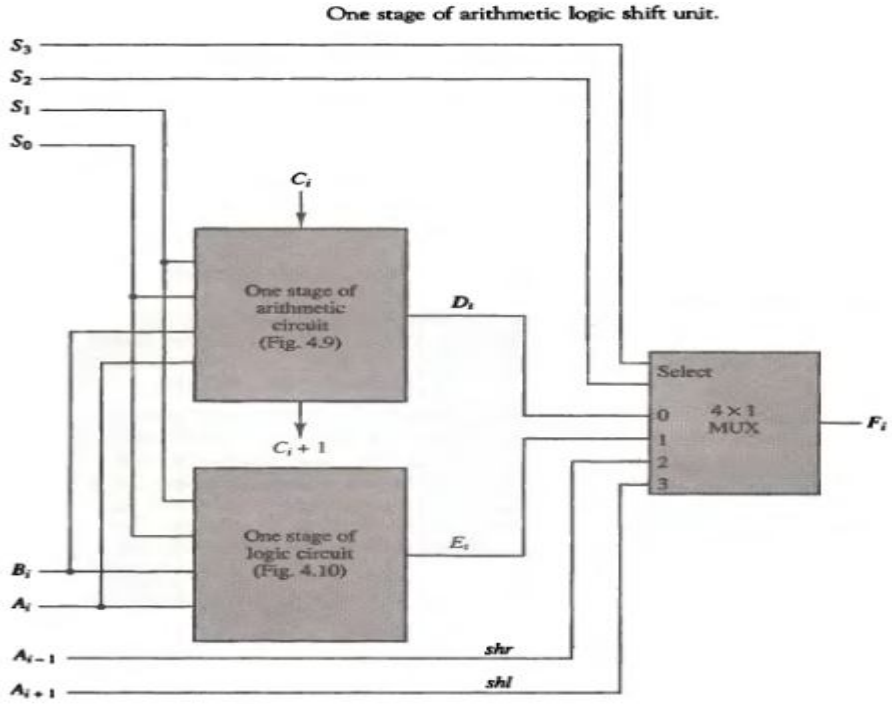


Function table

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

4-bit combinational circuit shifter.

Arithmetic Logic Shift Unit



One stage of arithmetic logic shift unit.

Function Table for Arithmetic Logic Shift Unit

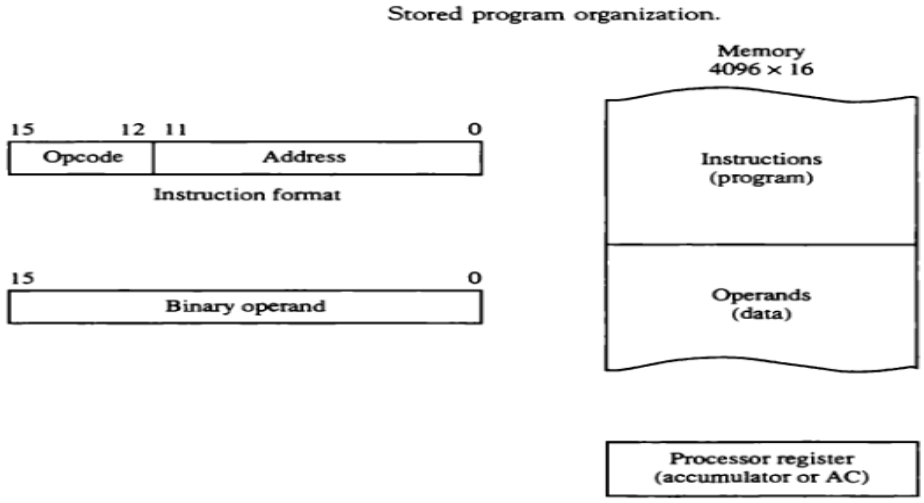
Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
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	×	×	×	$F =$ shr A	Shift right A into F
1	1	×	×	×	$F =$ shl A	Shift left A into F

Lecture 4:

Basic Computer Organization and Design

Instruction code is a group of bits that instruct the computer to perform a specific operation. It is usually divided into parts, each having its own particular interpretation. The most basic part of an instruction code is its operation code operation part.

Stored Program Organization

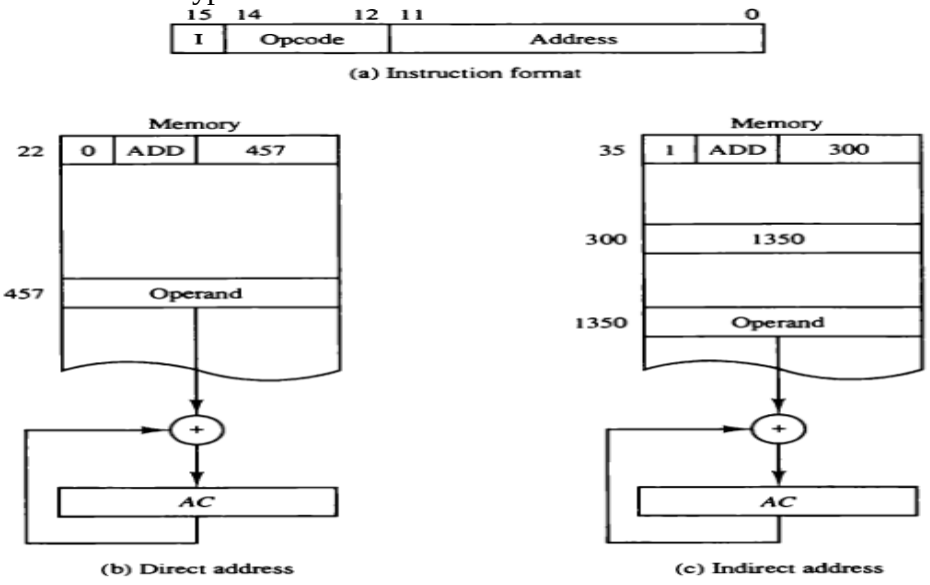


Stored program organization.

-Computers that have a single-processor register usually assign to it the name accumulator (AC) and label it AC.

Indirect Address:It is sometimes convenient to use the address bits of an instruction code not as an address but as the actual operand. When the second part of an instruction code specifies an operand, the instruction is said to have an immediate operand.

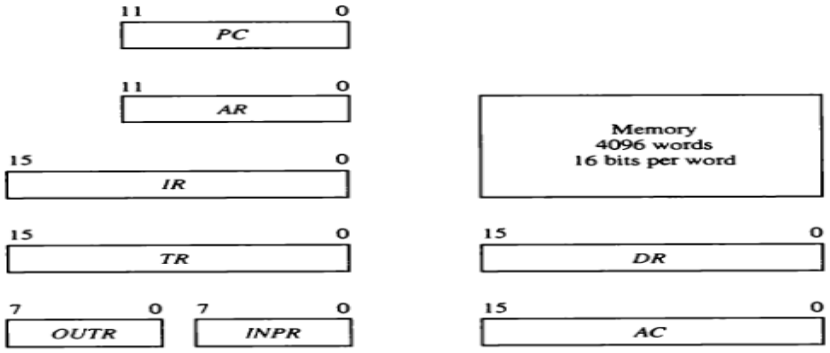
The **effective address** to be the address of the operand in a computation-type instruction or the target address in a branch-type instruction.



Demonstration of direct and indirect address.

Computer Registers

-Computer instructions are normally stored in consecutive memory locations and are executed sequentially one at a time.



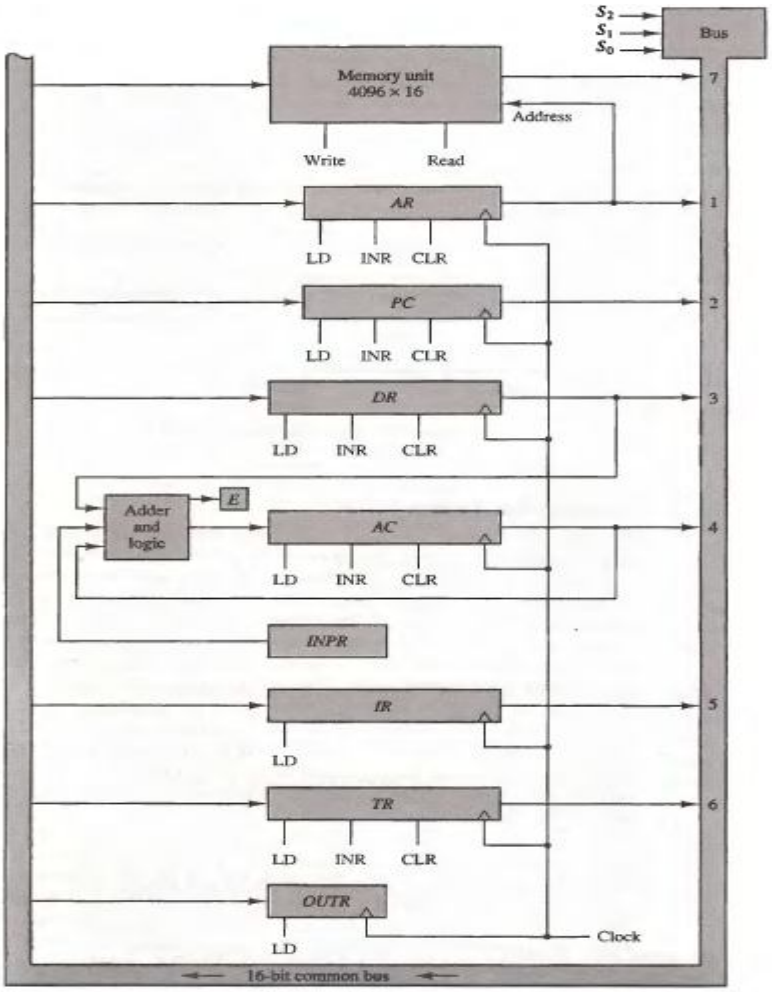
Basic computer registers and memory.

List of Registers for the Basic Computer

Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
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

Common Bus System

Paths must be provided to transfer information from one register to another and between memory and registers. The number of wires will be excessive if connections are made between the outputs of each register and the inputs of the other registers.



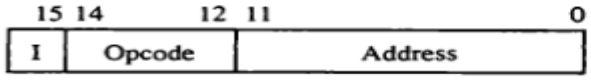
Basic computer registers connected to a common bus.

- The input data and output data of the memory are connected to the common bus, but the memory address is connected to AR.
- DR←AC and AC ←DR
- can be executed at the same time. This can be done by placing the content of AC on the bus (with S2SiS0 = 100), enabling the LD (load) input of DR, transferring the content of DR through the adder and logic circuit into AC, and enabling the LD (load) input of AC, all during the same clock cycle. The two transfers occur upon the arrival of the clock pulse transition at the end of the clock cycle.

Computer Instructions

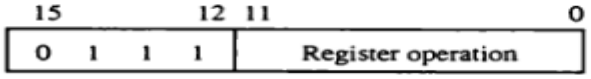
The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered. A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode J.

Basic computer instruction formats.



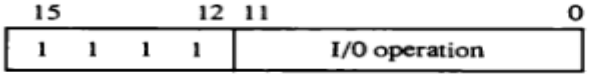
(Opcode = 000 through 110)

(a) Memory – reference instruction



(Opcode = 111, $I = 0$)

(b) Register – reference instruction



(Opcode = 111, $I = 1$)

(c) Input – output instruction

Basic Computer Instructions

Symbol	Hexadecimal code $I = 0$	Hexadecimal 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
CMA	7200		Complement AC
CME	7100		Complement E
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		Interrupt on
IOF	F040		Interrupt off

Lecture 5:

Instruction Set Completeness

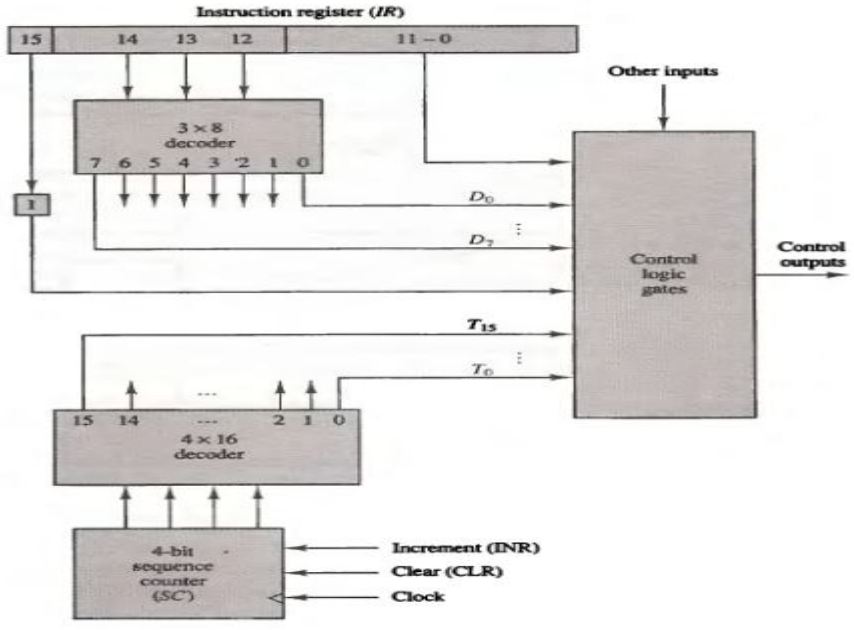
The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

1. Arithmetic, logical, and shift instructions
2. Instructions for moving information to and from memory and processor registers
3. Program control instructions together with instructions that check status conditions
4. Input and output instructions

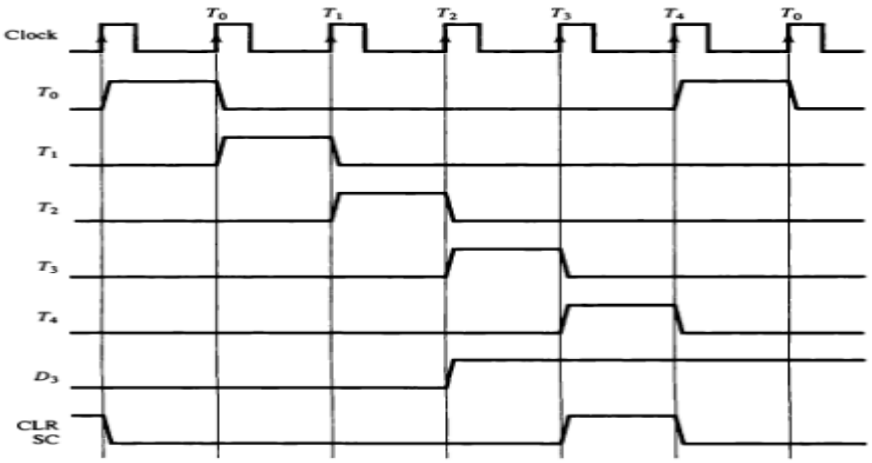
Timing and Control

The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by a control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

- In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation.
- In the microprogrammed control, any required control changes or modifications can be done by updating the microprogram in control memory.
- The sequence counter SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 x 16 decoder.



Control unit of basic computer.



Example of control timing signals.

Lecture 6:

Instruction Cycle:

• The steps that the control unit carries out in executing aprogram are:

(1) Fetch the next instruction to be executed from memory.
(2) Decode the opcode.
(3) Read operand(s) from main memory, if any.
(4) Execute the instruction and store results, if any.
(5) Go to step 1.

Fetch and Decode:

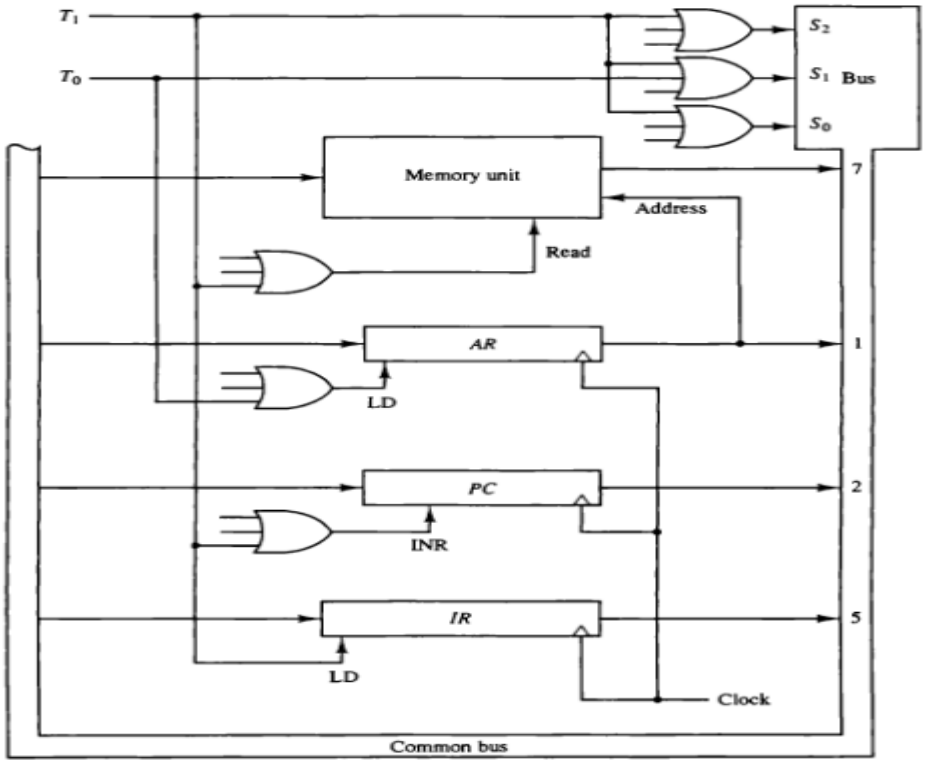
Initially, the program counter PC is loaded with the address of the first instruction in the program. The sequence counter SC is cleared to 0, providing a decoded timing signal T0. After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T0, Tu T2, and so on.

T0: AR←PC

T1: IR←M[AR], PC←PC + 1

T2: D0/..., D7←Decode IR(12-14), AR←IR(O-ll), I←1R(15)

Since only AR is connected to the address inputs of memory, it is necessary to transfer the address from PC to AR during the clock transition associated with timing signal T0.



Register transfers for the fetch phase.

- To provide the data path for the transfer of PC to AR we must apply timing signal T0 to achieve the following connection:
  1. Place the content of PC onto the bus by making the bus selection inputs S2SiS0 equal to 010.
  2. Transfer the content of the bus to AR by enabling the LD input of AR.
  The next clock transition initiates the transfer from PC to AR since T0 = 1.
  T1: IR←M[AR], PC←PC +1
- The microoperation for the indirect address condition can be symbolized by the register transfer statement AR←M[AR].
- The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T3. This can be symbolized as follows:
  D7'IT3: AR←M[AR]
  D7'I'T3: Nothing
  D7I'T3: Execute a register-reference instruction
  D7IT3: Execute an input-output instruction

Lecture 7:

Register-Reference Instructions

Register-reference instructions are recognized by the control when D7 = 1 and 1 = 0. These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions. These 12 bits are available in IR(O-ll). They were also transferred to AR during time T2.

Execution of Register-Reference Instructions

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0–11)$ that specifies the operation]

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

Memory -Reference Instructions

The effective address of the instruction is in the address register AR and was placed there during timing signal T2 when I = 0, or during timing signal T3 when I=1.

Memory-Reference Instructions

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

AND to AC

This is an instruction that performs the AND logic operation on pairs of bits in AC and the memory word specified by the effective address.

D0T4: DR←M[AR]

D0T5: AC←AC +DR, SC←O

ADD to AC

This instruction adds the content of the memory word specified by the effective address to the value of AC. The sum is transferred into AC and the output carry Cout is transferred to the £ (extended accumulator) flip-flop.

D1T4: DR ←M[AR]

D1Ts: AC ← AC + DR, E ←Cout, SC ← 0

LDA: Load to AC

This instruction transfers the memory word specified by the effective address to AC. The microoperations needed to execute this instruction are

D2T4: DR ←M[AR]

D2T5: AC ← DR, SC ← 0

STA: Store AC

This instruction stores the content of AC into the memory word specified by the effective address.

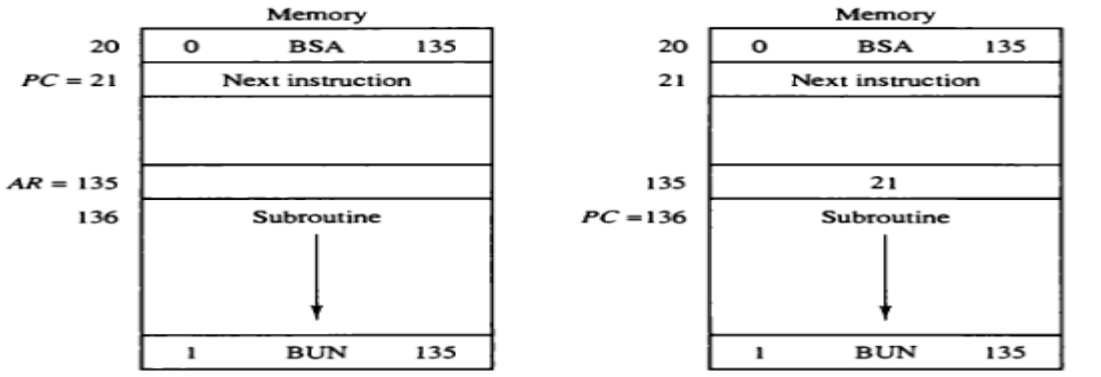
D3T4: M[AR] ← AC, SC ←0

BUN: Branch Unconditionally

D4T4: PC ←AR, SC ←0

BSA: Branch and Save Return Address

Example of BSA instruction execution.



(a) Memory, *PC*, and *AR* at time T_4 (b) Memory and *PC* after execution

D5T4: M[AR] ←PC, AR ← AR + 1

D5T5: PC ←AR, SC ← 0

ISZ: Increment and Skip if Zero

D6T4: DR ←M[AR]

D6T5: DR ← DR + 1

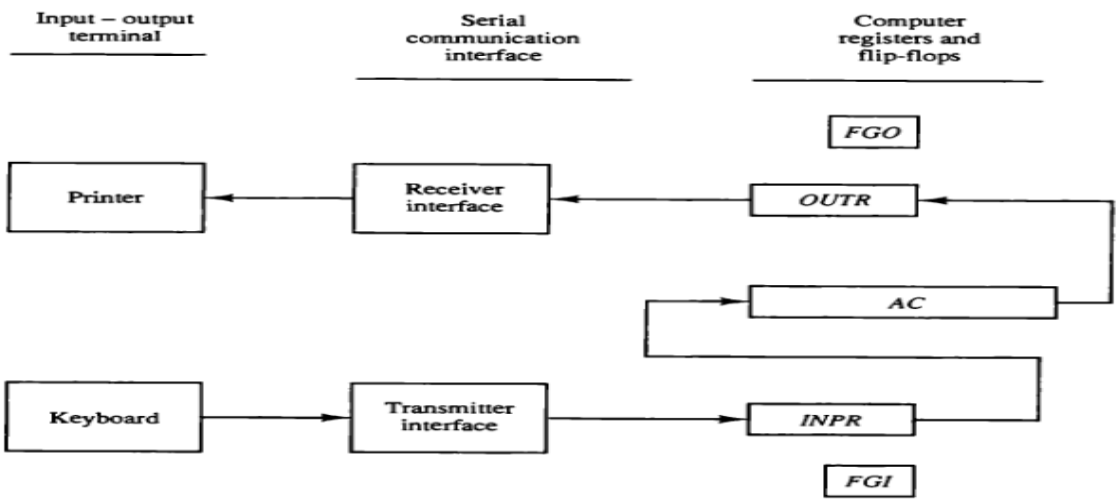
D6T6: M[AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ←0

Input—Output and Interrupt

- A computer can serve no useful purpose unless it communicates with the external environment. Instructions and data stored in memory must come from some input device.

Input-Output Configuration

Input-output configuration.



Input-Output Instructions

Input-Output Instructions

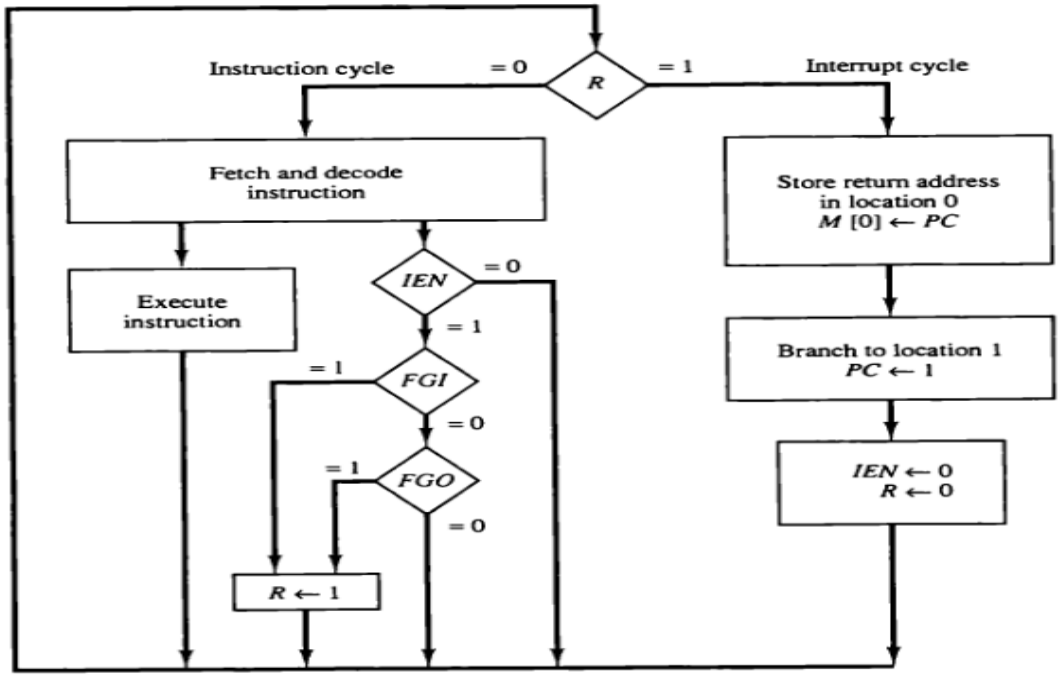
$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR$, $FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7)$, $FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

Lecture 8:

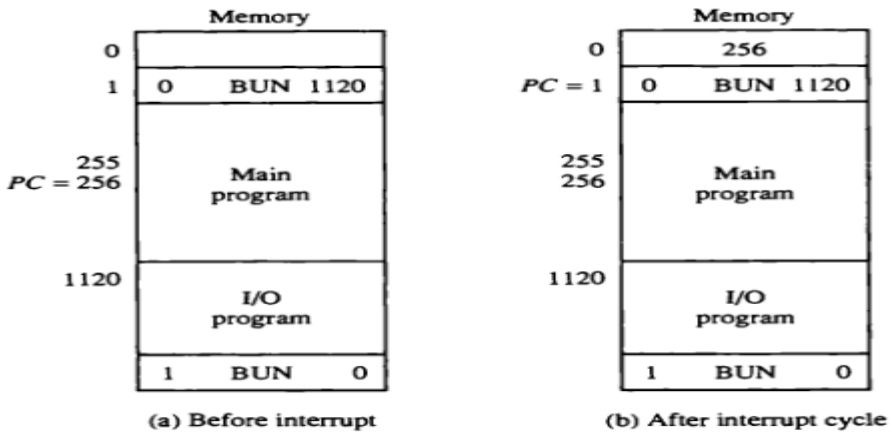
Program Interrupt

- The process of communication just described is referred to as programmed control transfer. The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer. The difference of information flow rate between the computer and that of the input-output device makes this type of transfer inefficient.
- To see why this is inefficient, consider a computer that can go through an instruction cycle in 1 (is. Assume that the input-output device can transfer information at a maximum rate of 10 characters per second. This is equivalent to one character every 100,000 μs.
- An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer. In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility. While the computer is running a program, it does not check the flags.



Flowchart for interrupt cycle.

Demonstration of the interrupt cycle.



Interrupt Cycle

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

Can be done with the following sequence of microoperations:

RT0: AR←O, TR←PC

RT1: M[AR]←TR, PC ←0

RT2:PC←PC + 1, IEN←0, R←0, SC←0

Design of Basic Computer

The basic computer consists of the following hardware components:

1. A memory unit with 4096 words of 16 bits each

2. Nine registers: AR, PC, DR, AC, IR, TR, OUTR, INPR, and SC

3. Seven flip-flops: /, S, E, R, IEN, FGI, and FGO

4. Two decoders: a 3 x 8 operation decoder and a 4 x 16 timing decoder

5. A 16-bit common bus

6. Control logic gates

7. Adder and logic circuit connected to the input of AC

Control Functions and Microoperations for the Basic Computer

Fetch	$R'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14)$, $AR \leftarrow IR(0\text{–}11)$, $I \leftarrow IR(15)$
Indirect	D_7IT_3:	$AR \leftarrow M[AR]$
Interrupt:		
$T_0'T_1'T_2'(IEN)(FGI + FGO)$:		$R \leftarrow 1$
	RT_0:	$AR \leftarrow 0$, $TR \leftarrow PC$
	RT_1:	$M[AR] \leftarrow TR$, $PC \leftarrow 0$
	RT_2:	$PC \leftarrow PC + 1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$
Memory-reference:		
AND	D_0T_4:	$DR \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR$, $E \leftarrow C_{out}$, $SC \leftarrow 0$
LDA	D_2T_4:	$DR \leftarrow M[AR]$
	D_2T_5:	$AC \leftarrow DR$, $SC \leftarrow 0$
STA	D_3T_4:	$M[AR] \leftarrow AC$, $SC \leftarrow 0$
BUN	D_4T_4:	$PC \leftarrow AR$, $SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC$, $AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR$, $SC \leftarrow 0$
ISZ	D_6T_4:	$DR \leftarrow M[AR]$
	D_6T_5:	$DR \leftarrow DR + 1$
	D_6T_6:	$M[AR] \leftarrow DR$, if $(DR = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$
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, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow$ shl AC, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$
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}:	$AC(0\text{–}7) \leftarrow INPR$, $FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{–}7)$, $FGO \leftarrow 0$
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:	$IEN \leftarrow 0$

Control Logic Gates

- The inputs to this circuit come from the two decoders, the J flip-flop, and bits 0 through 11 of IR. The other inputs to the control logic are: AC bits 0 through 15 to check if AC = 0 and to detect the sign bit in AC(15); DR bits 0 through 15 to check if DR = 0; and the values of the seven flip-flops.
- The outputs of the control logic circuit are:
  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 flip-flops
  4. Signals for S2, Si, and S0 to select a register for the bus
  5. Signals to control the AC adder and logic circuit

Control of Registers and Memory

- The control inputs of the registers are LD (load), INR (increment), and CLR (clear).
- Suppose that we want to derive the gate structure associated with the control inputs of AR. We scan Table 5-6 to find all the statements that change the content of AR:

R'T0: AR←PC

R'T2: AR←IR(O-ll)

D7'IT3: AR←M[AR]

RT0: AR←O

D5T4: AR ←AR + 1

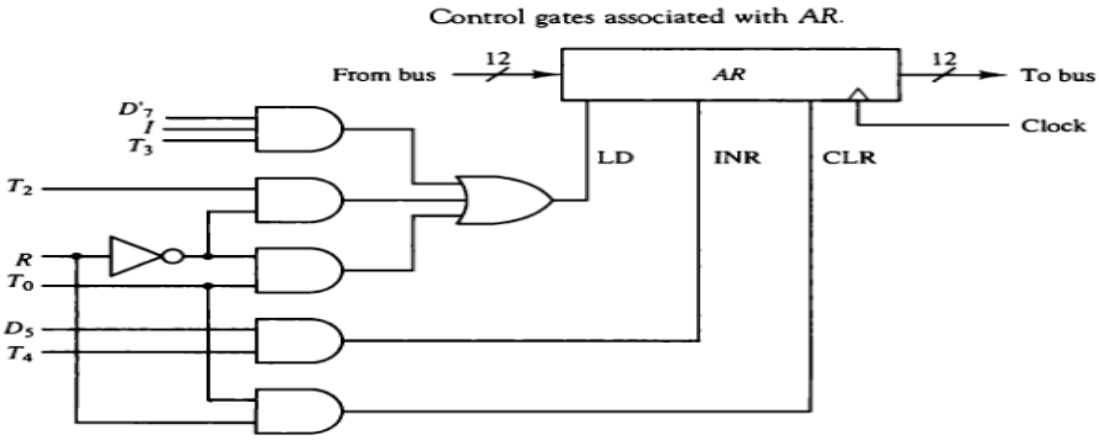
The control functions can be combined into three Boolean expressions as follows:

LD(AR) = R'T0 + R'T2 + D7'TT3

CLR(AR) = RT0

INR(AR) = D5T4

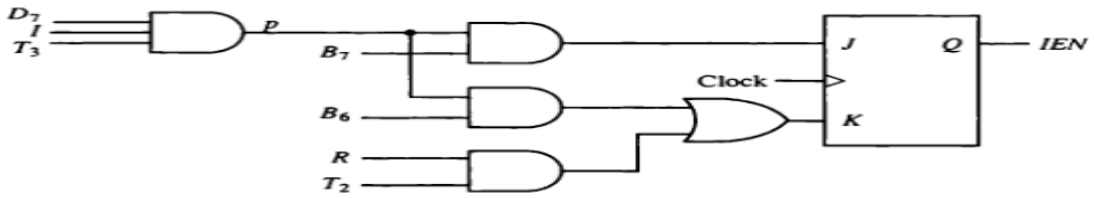
- In a similar fashion we can derive the control gates for the other registers as well as the logic needed to control the read and write inputs of memory.
  Read = R'T1+ D7'IT3 + (D0 + D1 + D2 + D6)T4
- The output of the logic gates that implement the Boolean expression above must be connected to the read input of memory.



Control of Single Flip-flops

- The control gates for the seven flip-flops can be determined in a similar manner.
- EXAMPLE: IEN may change as a result of the two instructions ION and IOF.
  pB7: IEN←1
  pB6: lEN←O
  where p = D7IT3 and B7 and B6 are bits 7 and 6 of IR, respectively. Moreover, at the end of the interrupt cycle IEN is cleared to 0.
  RT2: IEN←0

Control inputs for IEN.



Control of Common Bus

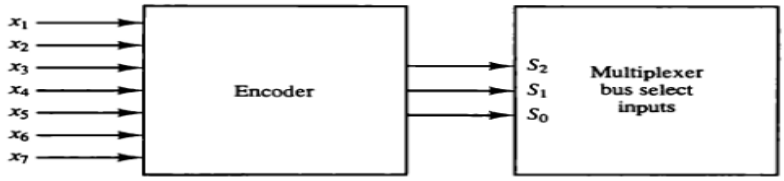
TABLE Encoder for Bus Selection Circuit

Inputs							Outputs			Register selected for bus
x_1	x_2	x_3	x_4	x_5	x_6	x_7	S_2	S_1	S_0	
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

So = X1 + x3 + x5 + x7
Si = x2 + x3 + x6 + x7
S2 = Xi + x5 + X6, + x7

- x7 = R'T1 + D7'IT3 + (D0 + D1 + D2 + D6)T4

Encoder for bus selection inputs.



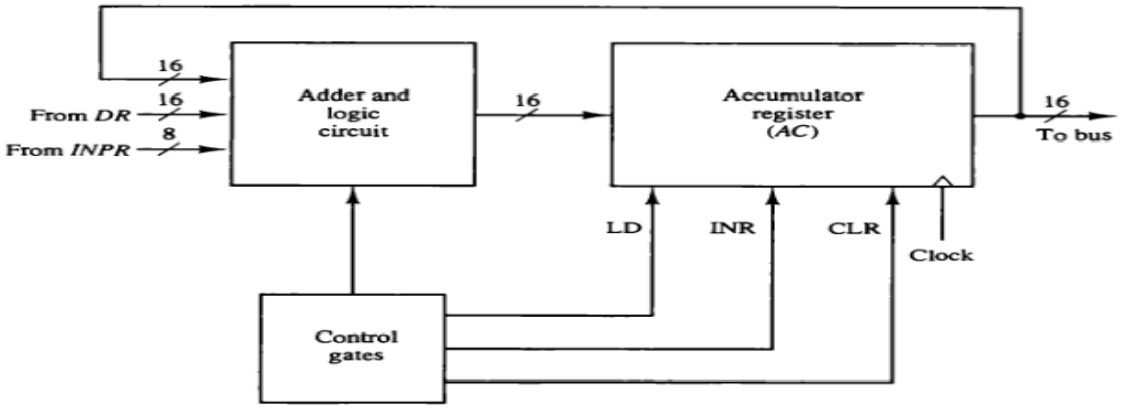
Design of Accumulator Logic

- In order to design the logic associated with AC, it is necessary to go over the register transfer statements and extract all the statements that change the content of AC.

D_0T_5:	$AC \leftarrow AC \wedge DR$	AND with *DR*
D_1T_5:	$AC \leftarrow AC + DR$	Add with *DR*
D_2T_5:	$AC \leftarrow DR$	Transfer from *DR*
pB_{11}:	$AC(0\text{–}7) \leftarrow INPR$	Transfer from *INPR*
rB_9:	$AC \leftarrow \overline{AC}$	Complement
rB_7:	$AC \leftarrow \text{shr } AC, \quad AC(15) \leftarrow E$	Shift right
rB_6:	$AC \leftarrow \text{shl } AC, \quad AC(0) \leftarrow E$	Shift left
rB_{11}:	$AC \leftarrow 0$	Clear
rB_5:	$AC \leftarrow AC + 1$	Increment

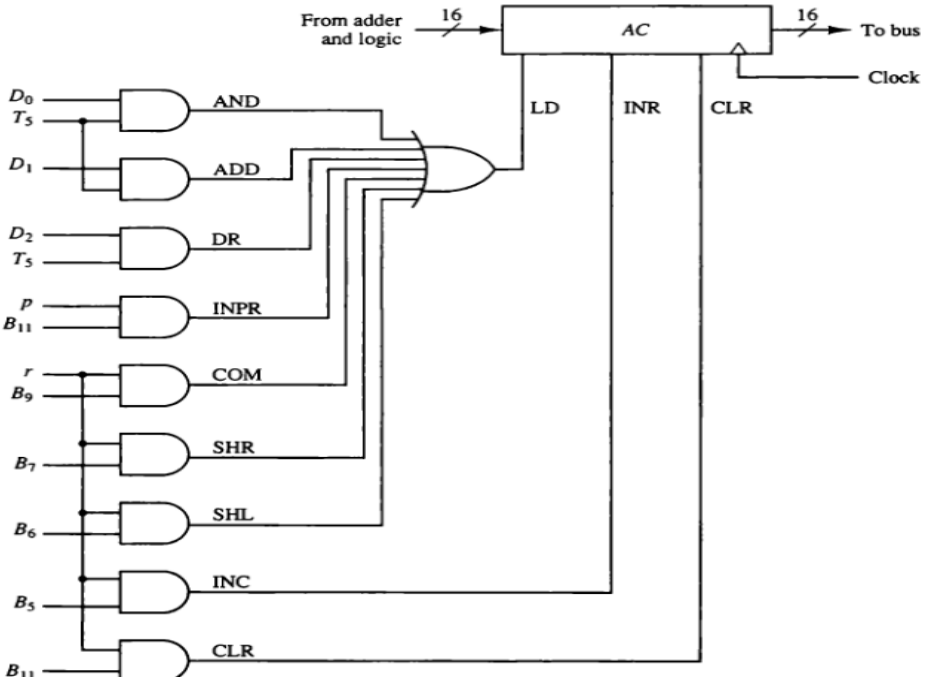
-
- From this list we can derive the control logic gates and the adder and logic circuit.

Circuits associated with AC.



Control of AC Register

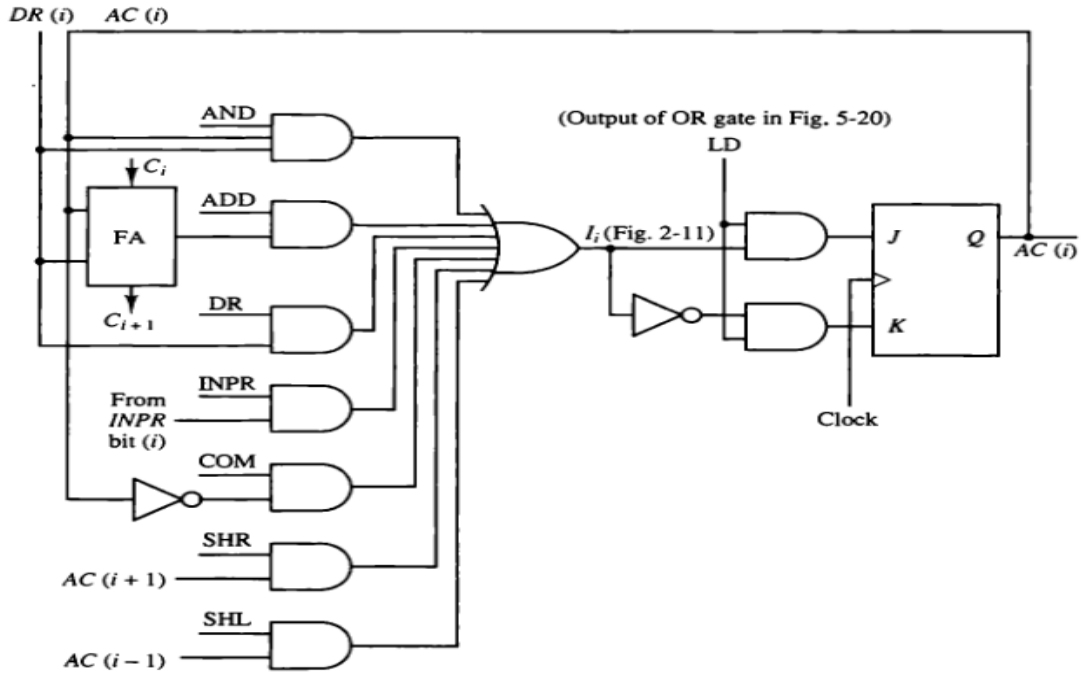
Gate structure for controlling the LD, INR, and CLR of AC.



Adder and logic circuit

- The adder and logic circuit can be subdivided into 16 stages, with each stage corresponding to one bit of AC.
- One stage of the adder and logic circuit consists of seven AND gates, one OR gate and a full-adder (FA)

One stage of adder and logic circuit.



-

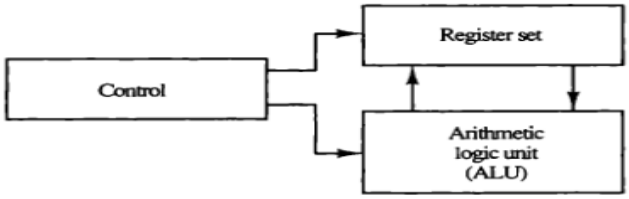
LECTURE 9:

Central Processing Unit

CPU:The part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU.

The CPU is made up of three major parts:

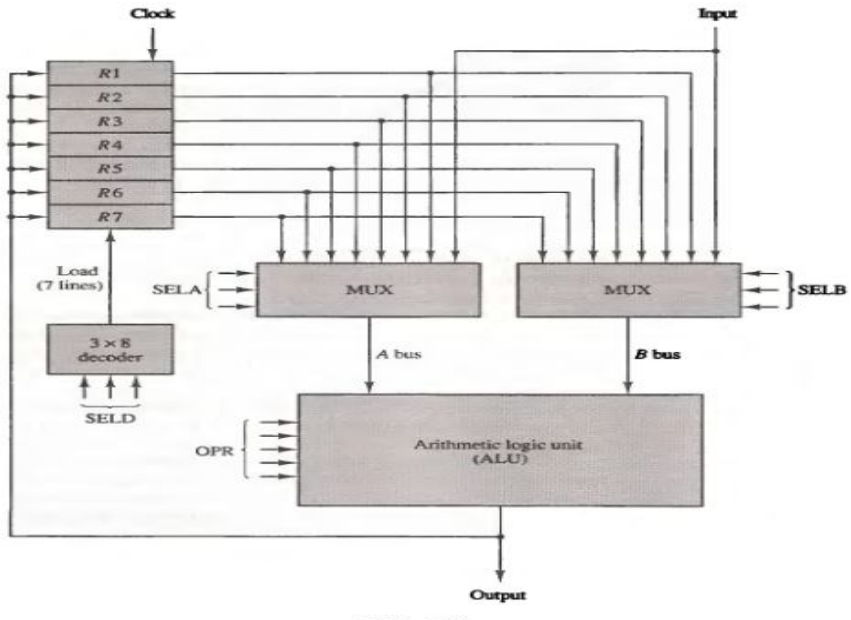
1- The register set stores intermediate data used during the execution of the instructions.
2- The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions.
3- The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.



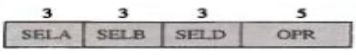
Major components of CPU.

General Register Organization

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.



(a) Block diagram



For example, to perform the operation R1←R2 + R3

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

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

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

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

4. Decoder destination selector (SELD): to transfer the content of the output bus into Rl.

Control Word

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

The encoding of the register selections is specified in following 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.

TABLE Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	*R1*	*R1*	*R1*
010	*R2*	*R2*	*R2*
011	*R3*	*R3*	*R3*
100	*R4*	*R4*	*R4*
101	*R5*	*R5*	*R5*
110	*R6*	*R6*	*R6*
111	*R7*	*R7*	*R7*

- 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.

Encoding of ALU Operations

OPR Select	Operation	Symbol
00000	Transfer *A*	TSFA
00001	Increment *A*	INCA
00010	Add *A* + *B*	ADD
00101	Subtract *A* − *B*	SUB
00110	Decrement *A*	DECA
01000	AND *A* and *B*	AND
01010	OR *A* and *B*	OR
01100	XOR *A* and *B*	XOR
01110	Complement *A*	COMA
10000	Shift right *A*	SHRA
11000	Shift left *A*	SHLA

Lecture10:

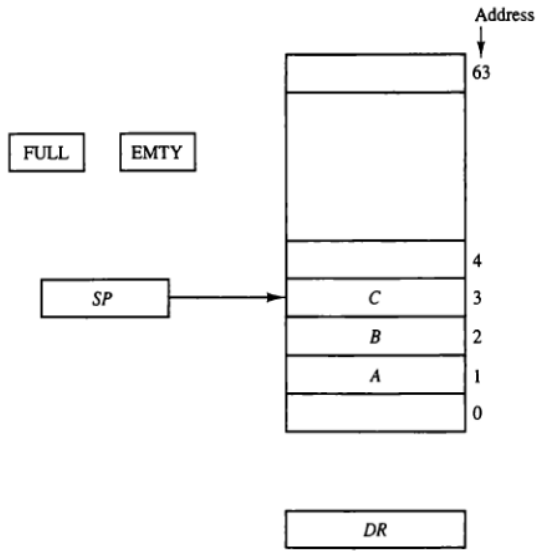
Stack Organization

A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

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

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. Following figure shows the organization of a 64-word register stack.



Block diagram of a 64-word stack.

- Initially, SP is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if FULL = 0), a new item is inserted with a push operation. The push operation is implemented with the following sequence of microoperations:
  SP←SP + 1 Increment stack pointer
  M[SP] ← DR Write item on top of the stack

  If (SP = 0) then (FULL ←1) Check if stack is full

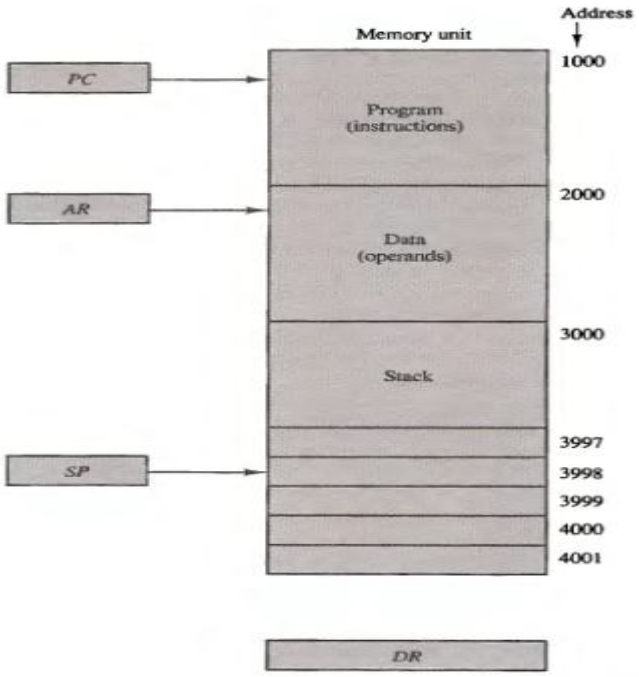
  EMTY ←0 Mark the stack not empty

- A new item is deleted from the stack if the stack is not empty (if pop EMTY = 0). The pop operation consists of the following sequence of micro- operations:

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

Memory Stack

Following Figure shows a portion of computer memory partitioned into three segments: program, data, and stack. The program counter PC points at the address of the next instruction in the program. The address register AR points at an array of data.



Computer memory with program, data, and stack segments.

We assume that the items in the stack communicate with a data register DR. A new item is inserted with the push operation as follows:

SP←SP- 1

M[SP)←DR

The stack pointer is decremented so that it points at the address of the next word. A memory write operation inserts the word from DR into the top of the stack. A new item is deleted with a pop operation as follows:

DR←M[SP]

SP←SP + 1

The top item is read from the stack into DR. The stack pointer is then incremented to point at the next item in the stack.

- The advantage of a memory stack is that the CPU can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

Reverse Polish Notation

- stack organization is very effective for evaluating arithmetic expressions. The common mathematical method of writing arithmetic expressions imposes difficulties when evaluated by a computer. The common arithmetic expressions are written in infix notation, with each operator written between the operands. Consider the simple arithmetic expression A*B + C*D
- The Polish mathematician Lukasiewicz showed that arithmetic expressions can be represented in prefix notation. This representation, often referred to as Polish notation, places the operator before the operands. The postfix notation, referred to as reverse Polish notation (RPN), places the operator after the operands. The following examples demonstrate the three representations:

  A + B Infix notation

  +AB Prefix or Polish notation

  AB+ Postfix or reverse Polish notation

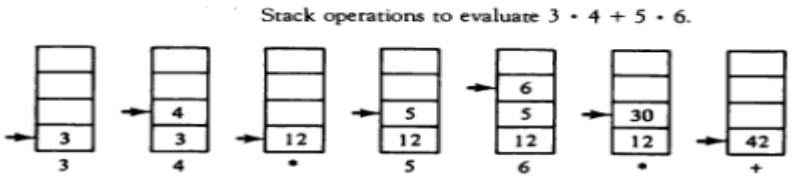
- The reverse Polish notation is in a form suitable for stack manipulation.
- The expression A*B + C*D is written in reverse Polish notation as AB*CD*+

Evaluation of Arithmetic Expressions

- Reverse Polish notation, combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmetic expressions. This procedure is employed in some electronic calculators and also in some computers.
- The stack is particularly useful for handling long, complex problems involving chain calculations. It is based on the fact that any arithmetic expression can be expressed in parentheses-free Polish notation.
- The following numerical example may clarify this procedure. Consider the arithmetic expression (3*4) + (5*6)

In reverse Polish notation, it is expressed as 34*56* +

- Scientific calculators that employ an internal stack require that the user convert the arithmetic expressions into reverse Polish notation. Computers that use a stack-organized CPU provide a system program to perform the conversion for the user.



Instruction Formats

The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

1. An operation code field that specifies the operation to be performed.

2. An address field that designates a memory address or a processor register.

3. A mode field that specifies the way the operand or the effective address is determined.

- Operands residing in processor registers are specified with a **register address**.

- Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations:

1. Single accumulator organization.

2. General register organization.

3. Stack organization.

-- Three*Address Instructions

Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand. The program in assembly language that evaluates X = (A + B) * (C + D) is shown below, together with comments that explain the register transfer operation of each instruction.

ADD Rl, A, B R1←M[A]+M[B]

ADD R2, C, D R2←M[C]+M[D]

MUL X, Rl, R2 M[X]←R1*R2

from memory and AC register. We will assume that the operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X.

Three*Address Instructions

Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand. The program in assembly language that evaluates X = (A + B) * (C + D) is shown below, together with comments that explain the register transfer operation of each instruction.

ADD Rl, A, B R1^M[A]+M[B]

ADD RE, C, D RE^M[C]+M[D]

MUL X, Rl, RE M[X]^R1*RE

It is assumed that the computer has two processor registers, Rl 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.

An example of a commercial computer that uses three-address instructions is the Cyber 170. The instruction formats in the Cyber computer are restricted to either three register address fields or two register address fields and one memory address field.

---Two'Address Instructions

Two-address instructions are the most common in commercial computers.

Here again each address field can specify either a processor register or a memory word. The program to evaluate X = (A + B) * (C + D) is as follows:

MOV Rl, A	R1←M[A]
ADD Rl, B	Rl←Rl + M[B]
MOV R2, C	R2←M[C]
ADD R2,D	R2←R2 + M[D]
MUL Rl ,R2	R1←R1*R2
MOV X, R1	M[X]←R1

--One'Address Instructions

One-address instructions use an implied accumulator (AC) register for all data manipulation. For multiplication and division there is a need for a second register. However, here we will neglect the second register and assume that the AC contains the result of all operations. The program to evaluate

X = (A + B)*(C + D)is

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

--Zero'Address Instructions

A stack-organized computer does not use an address held for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address held to specify the operand that communicates with the stack. The following program shows how X = (A + B) * (C + D) will be written for a stack-organized computer. (TOS stands for top of stack.)

```
PUSH    A    TOS←A
PUSH    B    TOS←B
ADD          TOS←(A + B)
PUSH    C    TOS←C
PUSH    D    TOS←D
ADD          TOS←(C + D)
MUL          TOS←(C + D)*(A + B)
POP     X    M[X]←TOS
```

--RISC Instructions

The instruction set of a typical RISC processor is restricted to the use of load and store instructions when communicating between memory and CPU. All other instructions are executed within the registers of the CPU without referring to memory.

The following is a program to evaluate X = (A + B)*(C + D).

```
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, R2   R3←R3 + R4
MUL     R1, R1, R3   R1←R1*R3
STORE   X, R1        M[X]←R1
```

Lecture11:

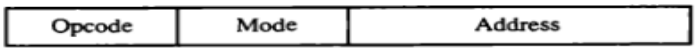
Addressing Modes

The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words.

Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:

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

Instruction format with mode field.



- The control unit of a computer is designed to go through an instruction cycle that is divided into three major phases:

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Execute the instruction.

- There is one register in the computer called the program counter or PC that keeps track of the instructions in the program stored in memory. PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory.
- The **mode field** is used to locate the operands needed for the operation.
- **Implied Mode**: In this mode the operands are specified implicitly in the definition of the instruction. For example, the instruction "complement accumulator" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, all register reference instructions that use an accumulator are implied-mode instructions.
- Immediate Mode: In this mode the operand is specified in the instruction itself.

Register Mode: In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction. A k-bit field can specify any one of 2* registers.

Register Indirect Mode: In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory.

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.

The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode.

Direct Address Mode: In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction.

Indirect Address Mode: In this mode the address field of the instruction gives the address where the effective address is stored in memory.

--- A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU. The effective address in these modes is obtained from the following computation: effective address = address part of instruction + content of CPU register

Relative Address Mode: In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.

Indexed Addressing Mode: In this mode the content of an index register is added to the address part of the instruction to obtain the effective address. The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory.

Base Register Addressing Mode: In this mode the content of a base register is added to the address part of the instruction to obtain the effective address. This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register.

Lecture12:

Data Transfer and Manipulation

The instruction set of different computers differ from each other mostly in the way the operands are determined from the address and mode fields.

Most computer instructions can be classified into three categories:

1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

Data transfer instructions cause transfer of data from one location to another without changing the binary information content.

Data manipulation instructions are those that perform arithmetic, logic, and shift operations.

Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer.

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.

Typical Data Transfer Instructions

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

Data Manipulation Instructions

- Data manipulation instructions perform operations on data and provide the computational capabilities for the computer. The data manipulation instructions in a typical computer are usually divided into three basic types:

1. Arithmetic instructions
2. Logical and bit manipulation instructions
3. Shift instructions

Arithmetic Instructions

The four basic arithmetic operations are addition, subtraction, multiplication, and division. Most computers provide instructions for all four operations.

Typical Arithmetic Instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

Logical and Bit Manipulation Instructions

Logical instructions perform binary operations on strings of bits stored in registers. They are useful for manipulating individual bits or a group of bits that represent binary-coded information.

Typical Logical and Bit Manipulation Instructions

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

Shift Instructions

Instructions to shift the content of an operand are quite useful and are often provided in several variations.

Typical Shift Instructions

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

Program Control

Instructions are always stored in successive memory locations. When processed in the CPU, the instructions are fetched from consecutive memory locations and executed.

Typical Program Control Instructions

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

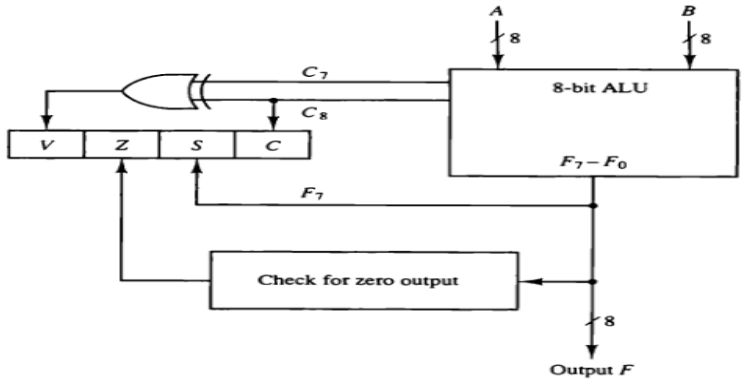
Status Bit Conditions

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

The four status bits are symbolized by C, S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.

1. Bit C (carry) is set to 1 if the end cany C8 is 1. It is cleared to 0 if the cany isO.
2. Bit S (sign) is set to 1 if the highest-order bit F? is 1. It is set to 0 if the bit is 0.
3. Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. It is cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.
4. Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries is equal to 1, and cleared to 0 otherwise. This is the condition for an overflow when negative numbers are in 2's complement.

For the 8-bit ALU, V = 1 if the output is greater than +127 or less than -128.



Status register bits.

Conditional Branch Instructions

Conditional Branch Instructions

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
	Unsigned compare conditions ($A - B$)	
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
	Signed compare conditions ($A - B$)	
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

- Some computers consider the C bit to be a borrow bit after a subtraction operation A — B. A borrow does not occur if A ^ B, but a bit must be borrowed from the next most significant position if A < B. The condition for a borrow is the complement of the carry obtained when the subtraction is done by taking the 2's complement of B. For this reason, a processor that considers the C bit to be a borrow after a subtraction will complement the C bit after adding the 2's complement of the subtrahend and denote this bit a borrow.

Subroutine Call and Return

- A subroutine is a self-contained sequence of instructions that performs a given computational task.
- The instruction that transfers program control to a subroutine is known by different names. The most common names used are call subroutine, jump to subroutine, branch to subroutine, or branch and save address.
- The instruction is executed by performing two operations:
  (1) the address of the next instruction available in the program counter (the return address) is stored in a temporary location so the subroutine knows where to return
  (2) control is transferred to the beginning of the subroutine.

Different computers use a different temporary location for storing the return address.

- Some store the return address in the first memory location of the subroutine, some store it in a fixed location in memory, some store it in a processor register, and some store it in a memory stack. The most efficient way is to store the return address in a memory stack. The advantage of

using a stack for the return address is that when a succession of subroutines is called, the sequential return addresses can be pushed into the stack. The return from subroutine instruction causes the stack to pop and the contents of the top of the stack are transferred to the program counter.

- A subroutine call is implemented with the following microoperations:

SP ←SP - 1	Decrement stack pointer
M[SP] ←PC	Push content of PC onto the stack
PC ← effective address	Transfer control to the subroutine

- If another subroutine is called by the current subroutine, the new return address is pushed into the stack, and so on. The instruction that returns from the last subroutine is implemented by the microoperations:

PC←M[SP]	Pop stack and transfer to PC
SP ←SP + 1	Increment stack pointer

Lecture13:

Program Interrupt

- Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request. Control returns to the original program after the service program is executed.
- The interrupt procedure is, in principle, quite similar to a subroutine call except for three variations:
  (1) The interrupt is usually initiated by an internal or external signal rather than from the execution of an instruction (except for software interrupt as explained later);
  (2) the address of the interrupt service program is determined by the hardware rather than from the address field of an instruction; and
  (3) an interrupt procedure usually stores all the information

- The state of the CPU at the end of the execute cycle (when the interrupt is recognized) is determined from:

1. The content of the program counter
2. The content of all processor registers
3. The content of certain status conditions

- **program status word** The collection of all status bit conditions in the CPU is sometimes called a program status word or PSW. The PSW is stored in a separate hardware register and contains the status information that characterizes the state of the CPU.

Types of Interrupts

- There are three major types of interrupts that cause a break in the normal execution of a program. They can be classified as:

1. External interrupts
2. Internal interrupts
3. Software interrupts

- External interrupts come from input-output (I/O) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source.
- Internal interrupts arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called traps. Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation.
- A software interrupt is initiated by executing an instruction. Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program.

Reduced Instruction Set Computer (RISC)

- An important aspect of computer architecture is the design of the instruction set for the processor. The instruction set chosen for a particular computer determines the way that machine language programs are constructed.
- A computer with a large number of instructions is classified as a complex instruction set computer, abbreviated CISC.
- In the early 1980s, a number of computer designers recommended that
- Computers use fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often. This type of computer is classified as a reduced instruction set computer or RISC.

CISC Characteristics

- The design of an instruction set for a computer must take into consideration not only machine language constructs, but also the requirements imposed on the use of high-level programming languages.
- the major characteristics of CISC architecture are:
  1. A large number of instructions—typically from 100 to 250 instructions
  2, Some instructions that perform specialized tasks and are used infrequently.
  3. A large variety of addressing modes—typically from 5 to 20 different modes
  4. Variable-length instruction formats
  5. Instructions that manipulate operands in memory

RISC Characteristics

- The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer. The major characteristics of a RISC processor are:

1. Relatively few instructions
2. Relatively few addressing modes
3. Memory access limited to load and store instructions
4. All operations done within the registers of the CPU
5. Fixed-length, easily decoded instruction format
6. Single-cycle instruction execution
7. Hardwired rather than microprogrammed control

- Other characteristics attributed to RISC architecture are:

1. A relatively large number of registers in the processor unit
2. Use of overlapped register windows to speed-up procedure call and return
3. Efficient instruction pipeline
4. Compiler support for efficient translation of high-level language programs into machine language programs.

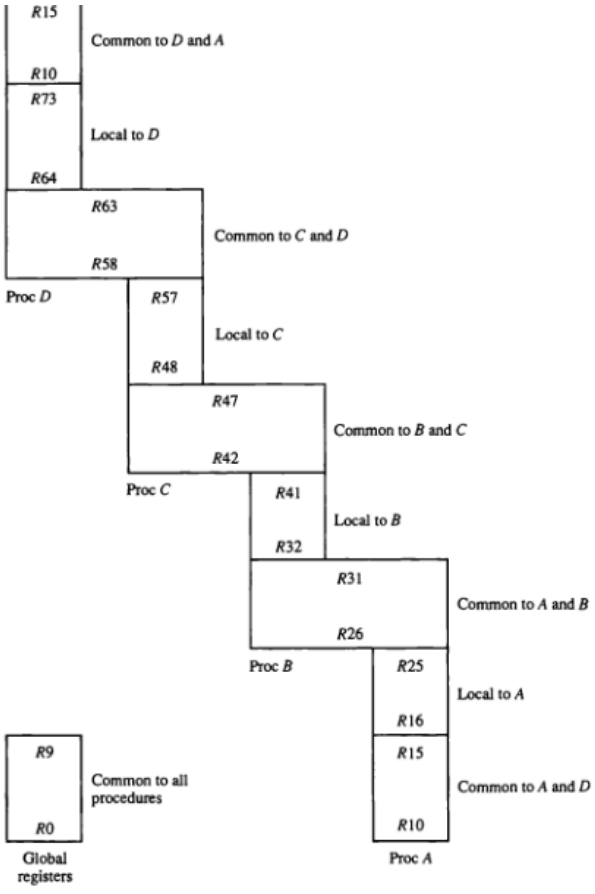
Overlapped Register Windows

- Procedure call and return occurs quite often in high-level programming languages. When translated into machine language, a procedure call produces a sequence of instructions that save

register values, pass parameters needed for the procedure, and then calls a subroutine to execute the body of the procedure.

- A characteristic of some RISC processors is their use of overlapped register windows to provide the passing of parameters and avoid the need for saving and restoring register values.
- The system has a total of 74 registers. Registers RO through R9 are global registers that hold parameters shared by all procedures. The other 64 registers are divided into four windows to accommodate procedures A,B,C, and D. Each register window consists of 10 local registers and two sets of six registers common to adjacent windows. Local registers are used for local variables.
- In general, the organization of register windows will have the following relationships:

number of global registers = G
number of local registers in each window = L
number of registers common to two windows = C
number of windows = W



Overlapped register windows.

- The number of registers available for each window is calculated as follows:
  window size = L + 2C + G
  The total number of registers needed in the processor is register file = (L + C)W + G
  In the example of Fig above we have G = 10, L = 10, C = 6, and W = 4. The window size is 10 + 12 + 10 = 32 registers, and the register file consists of (10 + 6) x 4 + 10 = 74 registers.

Lecture14:

Microprogrammed Control***(Control Unit)***

Control Memory

Control Unit

Initiate sequences of microoperations

Control signal *(that specify microoperations)* in a bus-organized systemgroups of bits that select the paths in multiplexers, decoders, and arithmetic logic units

Two major types of Control Unit

- Hardwired Control : The control logic is implemented with gates, F/Fs, decoders, and other digital circuits
- Fast operation, - Wiring change(if the design has to be modified)
- Microprogrammed Control :

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 Word

The control variables at any given time can be represented by a string of 1's and 0's.

Microprogrammed Control Unit

A control unit whose binary control variables are stored in memory (*control memory*).

- Microinstruction : *Control Word in Control Memory*
  The microinstruction specifies one or more microoperations
- Microprogram :A sequence of microinstruction

Dynamic microprogramming : *Control Memory* = RAM

- RAM can be used for writing (*to change a writable control memory*)
- Microprogram is loaded initially from an auxiliary memory such as a magnetic disk

Static microprogramming : *Control Memory* = ROM

- Control words in ROM are made permanent during the hardware production.

Microprogrammed control Organization :

1) Control Memory

- A memory is part of a control unit : *Microprogram*
- Computer Memory (*employs a microprogrammed control unit*)
  Main Memory : for storing user program (*Machine instruction/data*)
  Control Memory : for storing microprogram (*Microinstruction*)

2) Control Address Register
   Specify the address of the microinstruction

3) Sequencer (= *Next Address Generator*)
   Determine the address sequence that is read from control memory
   Next address of the next microinstruction can be specified several way depending on the sequencer input

4) Control Data Register (= *Pipeline Register*)

Hold the microinstruction read from control memory

Allows the execution of the microoperations specified by the control word ***simultaneously*** with the generation of the next microinstruction

- RISC Architecture Concept

RISC(Reduced Instruction Set Computer) system use hardwired control rather than microprogrammed control

- Address Sequencing
- Address Sequencing = Sequencer : Next Address Generator

  Selection of address for control memory
- Routine

  Microinstruction are stored in control memory *in groups*
- Mapping

  Instruction Code Address in control memory(*where routine is located*)

- Address Sequencing Capabilities : ***control memory address***

1) Incrementing of the control address register

2) Unconditional branch or conditional branch, depending on status bit conditions

3) Mapping process (*bits of the instruction address for control memory*)

4) A facility for subroutine return

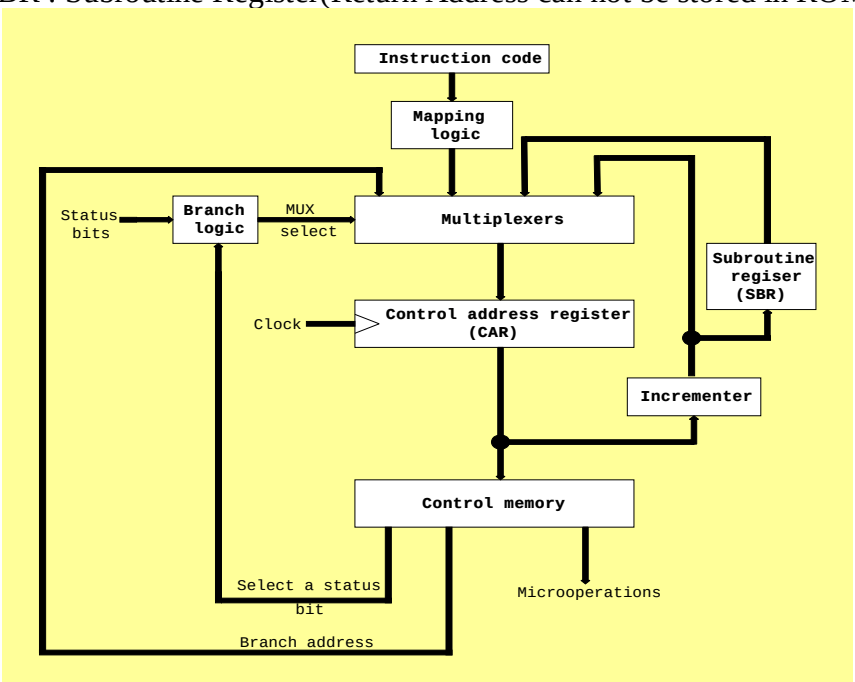
- Selection of address for control memory :

Multiplexer

① CAR Increment
② JMP/CALL
③ Mapping
④ Subroutine Return

CAR : Control Address Register. CAR receive the address from 4different paths

1) Incrementer
2) Branch address from control memory
3) Mapping Logic
4) SBR : Subroutine Register ,SBR : Subroutine Register(Return Address can not be stored in ROM)



◆ Conditional Branching

Status Bits

» Control the conditional branch decisions generated in the ***Branch Logic***

Branch Logic

» Test the specified condition and Branch to the indicated address if the condition is met ; otherwise, the control address register is just incremented.

Status Bit Test 와 Branch Logic

» 4 X 1 Mux 와 Input Logic

◆ Mapping of Instruction :

4 bit Opcode = specify up to 16 distinct instruction

Mapping Process : Converts *the 4-bit Opcode* to *a 7-bit control memory address*

» 1) Place a "0" in the most significant bit of the address

» 2) Transfer 4-bit Operation code bits

» 3) Clear the two least significant bits of the CAR (*Microinstruction)*

Mapping Function : Implemented by *Mapping ROM* or *PLD*

Control Memory Size : 128 words (= 2^7)

◆ Subroutine

Subroutines are programs that are used by other routines

- Subroutine can be called from any point within the main body of the microprogram

Microinstructions can be saved by subroutines that use common section of microcode

) Memory Reference, Operand, Effective Address Subroutine

Subroutine은 ORG 64, 즉 1000000 - 1111111에 위치(*Routine은 0000000 - 0111111*)

Subroutine must have a provision for

» storing the return address during a subroutine call

» restoring the address during a subroutine return

Last-In First Out(LIFO) Register Stack

Lecture15:

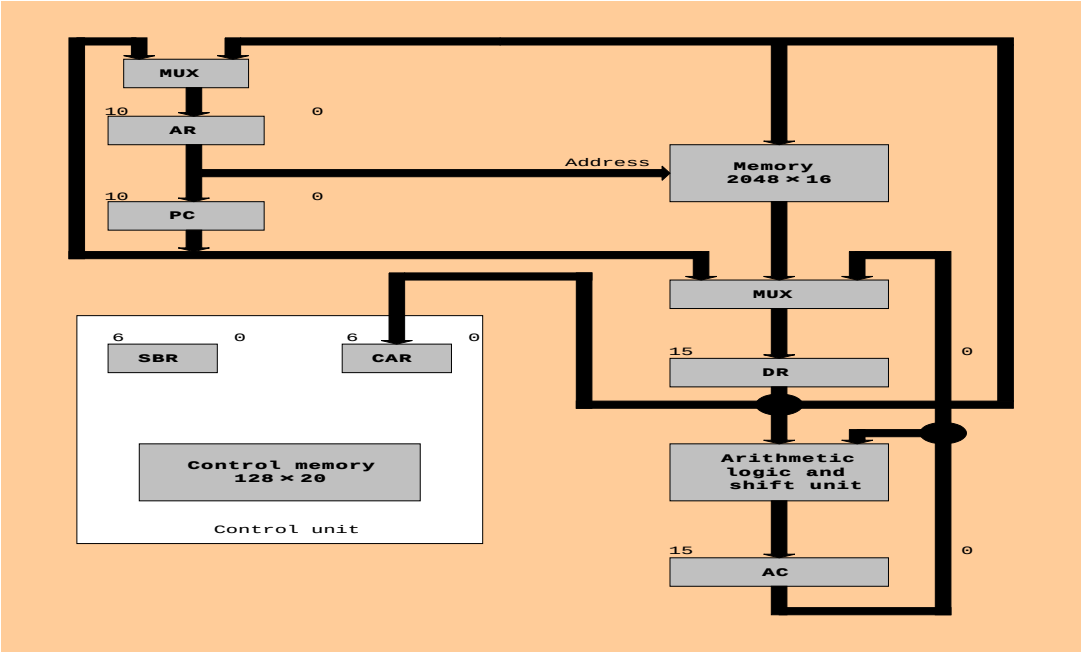
Microprogram Example

- Computer Configuration :
  - 2 Memory : Main memory(*instruction*/*data*), Control memory(*microprogram*)
    - » Data written to memory come from DR, and Data read from memory can go only to DR
  - 4 CPU Register and ALU : DR, AR, PC, AC, ALU
    - » DR can receive information from AC, PC, or Memory (*selected by MUX*)
    - » AR can receive information from PC or DR (*selected by MUX*)
    - » PC can receive information only from AR
    - » ALU performs microoperation with data from AC and DR
  - 2 Control Unit Register : SBR, CAR

- Instruction Format
  - Instruction Format :
    - » I : 1 bit for indirect addressing
    - » Opcode : 4 bit operation code
    - » Address : 11 bit address for system memory
  - Computer Instruction :
- Microinstruction Format :
  - 3 bit Microoperation Fields : F1, F2, F3
    - » 총 21개 Microoperation :
    - » 3 microoperation
      - 3 , 000(no operation)
    - » two or more conflicting microoperations can not be specified simultaneously**)**010 001 000

Clear AC to 0 and subtract DR from AC at the same time

- » Symbol **DRTAC**(F1 = 100)
  - stand for a transfer from DR to AC (*T* = *to*)



2 bit Condition Fields: CD

- » 00 : Unconditional branch, **U**
- » 01 : Indirect address bit, **I** = DR(15)
- » 10 : Sign bit of AC, **S** = AC(15)
- » 11 : Zero value in AC, **Z** = AC = 0

00 bit Branch Fields : BR

- 01 **JMP** ,Condition = 0 : Condition = 1 :
- 01 : **CALL** ,Condition = 0 : Condition = 1 :
- 10 : **RET**
- 11 **MAP**

7 bit Address Fields : AD(128 word : 128 X 20 bit)

◆ Symbolic Microinstruction

① Label Field : Terminated with a colon (**:**)

② Microoperation Field : one, two, or three symbols, separated by commas

③ CD Field : U, I, S, or Z

④ BR Field : JMP, CALL, RET, or MAP

⑤ AD Field

a. Symbolic Address : Label (= Address)

b. Symbol "NEXT" : next address

c. Symbol "RET" or "MAP" : AD field = 0000000

ORG : Pseudoinstruction(*define the origin, or first address of routine)*

◆ Fetch (**Sub**)Routine

l Memory Map(*128 words*) :

- » Address 0 to 63 : Routines for the 16 instruction(*4 instruction*)
- » Address 64 to 127 : Any other purpose(*Subroutines : FETCH, INDRCT*)

Microinstruction for FETCH Subroutine

$$AR \leftarrow PC$$
$$DR \leftarrow M[AR],\ PC \leftarrow PC + 1$$
$$AR \leftarrow DR(0-10),\ CAR(2-5) \leftarrow DR(11-14),\ CAR(0, 1, 6) \leftarrow 0$$

Label	Microoperation	CD	BR	AD
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	0

Design of Control Unit

◆ Decoding of Microinstruction Fields :

F1, F2, and F3 of Microinstruction are decoded with a 3 x 8 decoder

Output of decoder must be connected to the proper circuit to initiate the corresponding microoperation

F1 = 101 (5) : **DRTAR**

F1 = 110 (6) : **PCTAR**

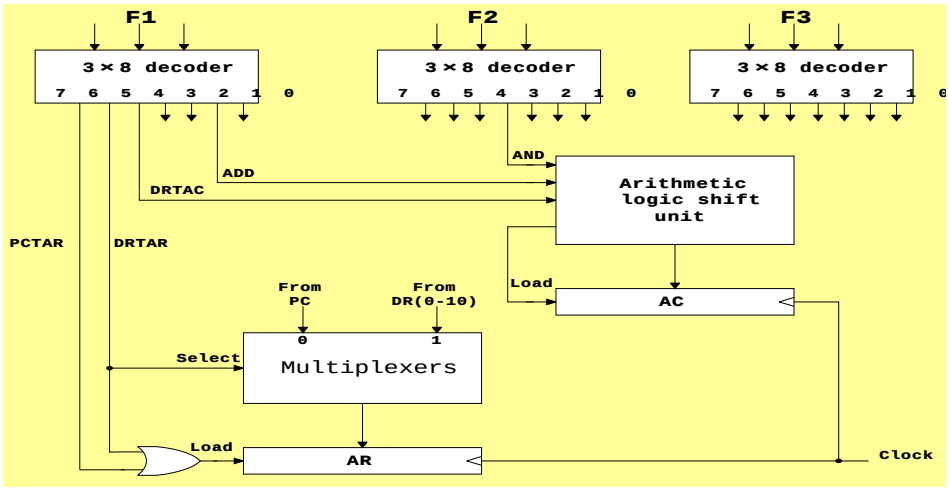
Output 5 and 6 of decoder F1 are connected to the load input of AR (*two input of OR gate*)

Multiplexer select the data from DR when output 5 is active
Multiplexer select the data from AC when output 5 is inactive

Arithmetic Logic Shift Unit

Control signal of ALU in *hardwired control*

Control signal will be now come from the *output of the decoders* associated with the AND, ADD, and DRTAC.



Microprogram Sequencer :

Microprogram Sequencer select the next address for control memory

MUX 1 Select an address source and route to CAR

JMP 와 CAL

JMP : AD가 MUX 1의 2CAR

CALL : AD가 MUX 1의 2 CAR,CAR + 1(Return Address) 이 LOAD SBR.

MUX 2

- » Test a status bit and the result of the test is applied to an input logic circuit
- » One of 4 Status bit is selected by Condition bit (**CD**)

Design of Input Logic Circuit

Select one of the source address(**S_0, S_1**) for CAR

Enable the load input(**L**) in SBR

Input Logic Truth Table :

Input :

I_0, I_1 from Branch bit (**BR**)

T from MUX 2 (**T**)

Output :

MUX 1 Select signal (**S_0, S_1**)

$S1 = I_1I_0' + I_1I_0 = I_1(I_0' + I_0) = I_1$

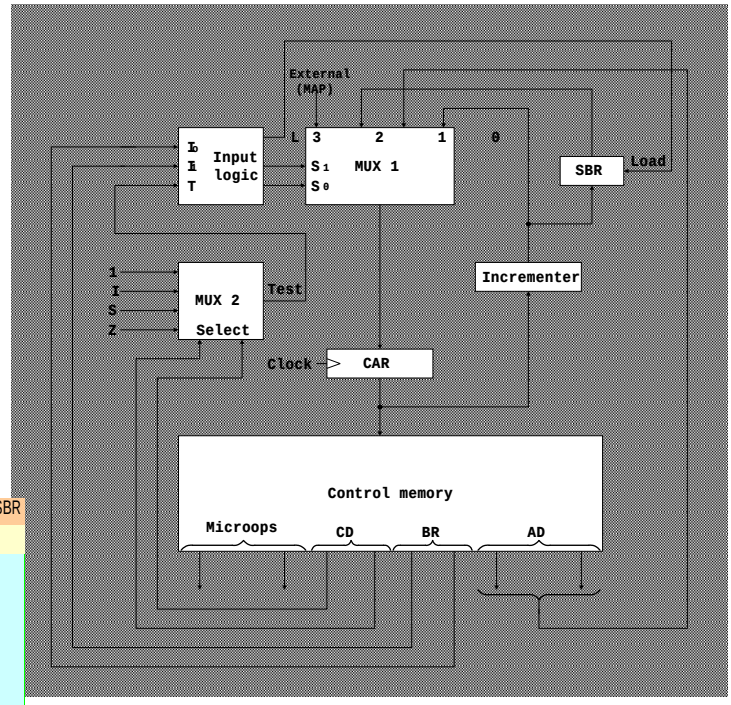
$S0 = I_1'I_0'T + I_1'I_0T + I_1I_0$

$= I_1'T(I_0' + I_0) + I_1I_0$

$= I_1'T + I_1I_0$

SBR Load signal (**L**)

$L = I_1'I_0T$



BR Field		Input			MUX 1		Load SBR
		I1	I0	T	S1	S0	L
0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	1	0	1	0	0	0	0
0	1	0	1	1	0	1	1
1	0	1	0	x	1	0	0
1	1	1	1	x	1	1	0