BASIC COMPUTER ORGANIZATION AND DESIGN

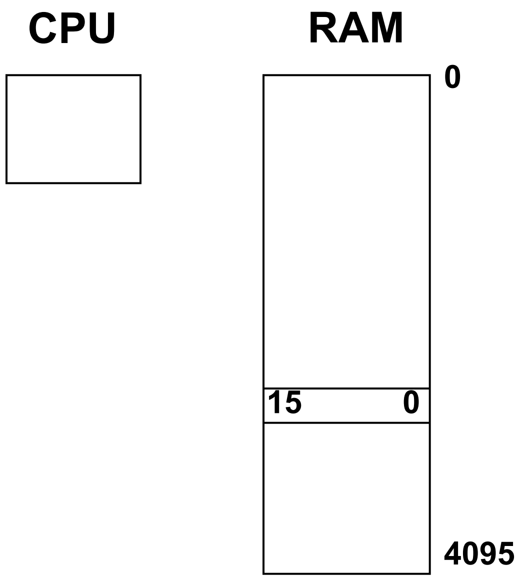
- Instruction Codes
- Computer Registers
- Computer Instructions
- Timing and Control
- Instruction Cycle
- Memory Reference Instructions
- Input-Output and Interrupt
- Complete Computer Description
- Design of Basic Computer
- Design of Accumulator Logic

INTRODUCTION

- **Every different processor type has its own design (different registers, buses, microoperations, machine instructions, etc)**
- **Modern processor is a very complex device**
- **It contains**
  - Many registers
  - Multiple arithmetic units, for both integer and floating point calculations
  - The ability to pipeline several consecutive instructions to speed execution
  - Etc.
- **However, to understand how processors work, we will start with a simplified processor model**
- **This is similar to what real processors were like ~25 years ago**
- **M. Morris Mano introduces a simple processor model he calls the *Basic Computer***
- **We will use this to introduce processor organization and the relationship of the RTL model to the higher level computer processor**

THE BASIC COMPUTER

- The Basic Computer has two components, a processor and memory
- The memory has 4096 words in it
  - 4096 = 2^{12}, so it takes 12 bits to select a word in memory
- Each word is 16 bits long



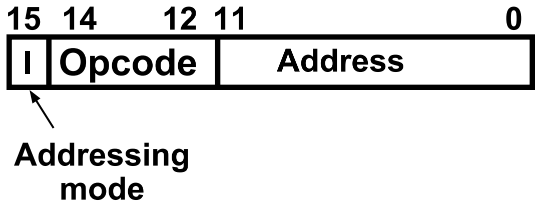
INSTRUCTIONS

- Program
  - A sequence of (machine) instructions
- (Machine) Instruction
  - A group of bits that tell the computer to *perform a specific operation* (a sequence of micro-operation)
- The instructions of a program, along with any needed data are stored in memory
- The CPU reads the next instruction from memory
- It is placed in an *Instruction Register* (IR)
- Control circuitry in control unit then translates the instruction into the sequence of microoperations necessary to implement it

INSTRUCTION FORMAT

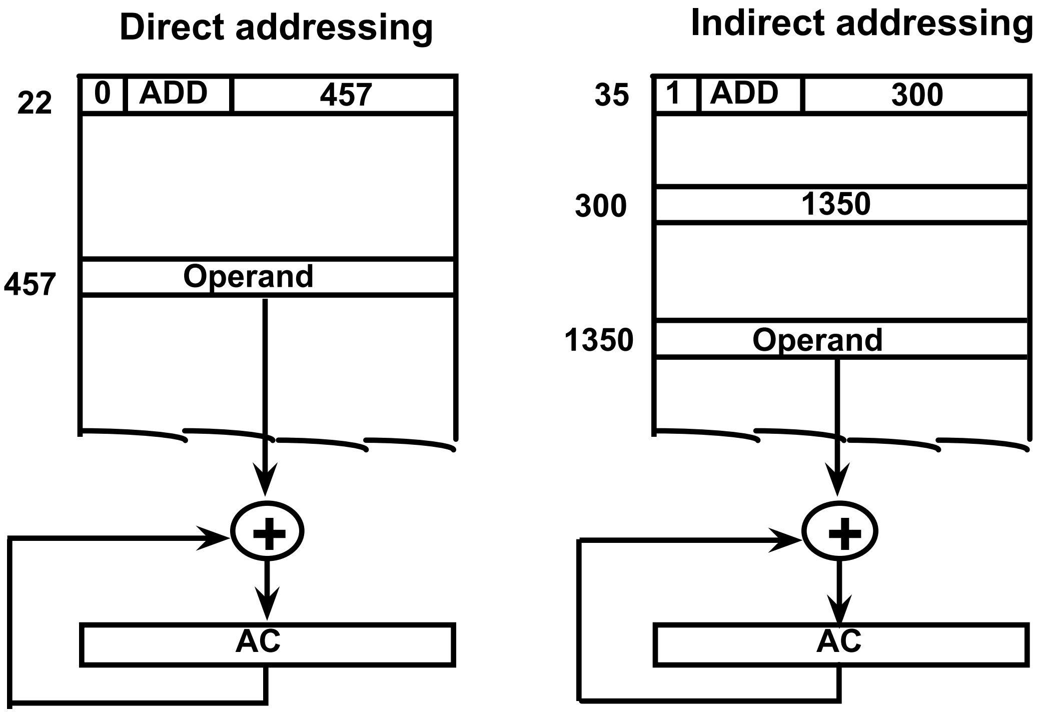
- **A computer instruction is often divided into two parts**
  - **An *opcode* (Operation Code) that specifies the operation for that instruction**
  - **An *address* that specifies the registers and/or locations in memory to use for that operation**
- **In the Basic Computer, since the memory contains 4096 (= 2^{12}) words, we needs 12 bit to specify which memory address this instruction will use**
- **In the Basic Computer, bit 15 of the instruction specifies the *addressing mode* (0: direct addressing, 1: indirect addressing)**
- **Since the memory words, and hence the instructions, are 16 bits long, that leaves 3 bits for the instruction's opcode**

Instruction Format



ADDRESSING MODES

- The address field of an instruction can represent either
  - Direct address: the address in memory of the data to use (the address of the operand), or
  - Indirect address: the address in memory of the address in memory of the data to use



- Effective Address (EA)
  - The address, that can be directly used without modification to access an operand for a computation-type instruction, or as the target address for a branch-type instruction

PROCESSOR REGISTERS

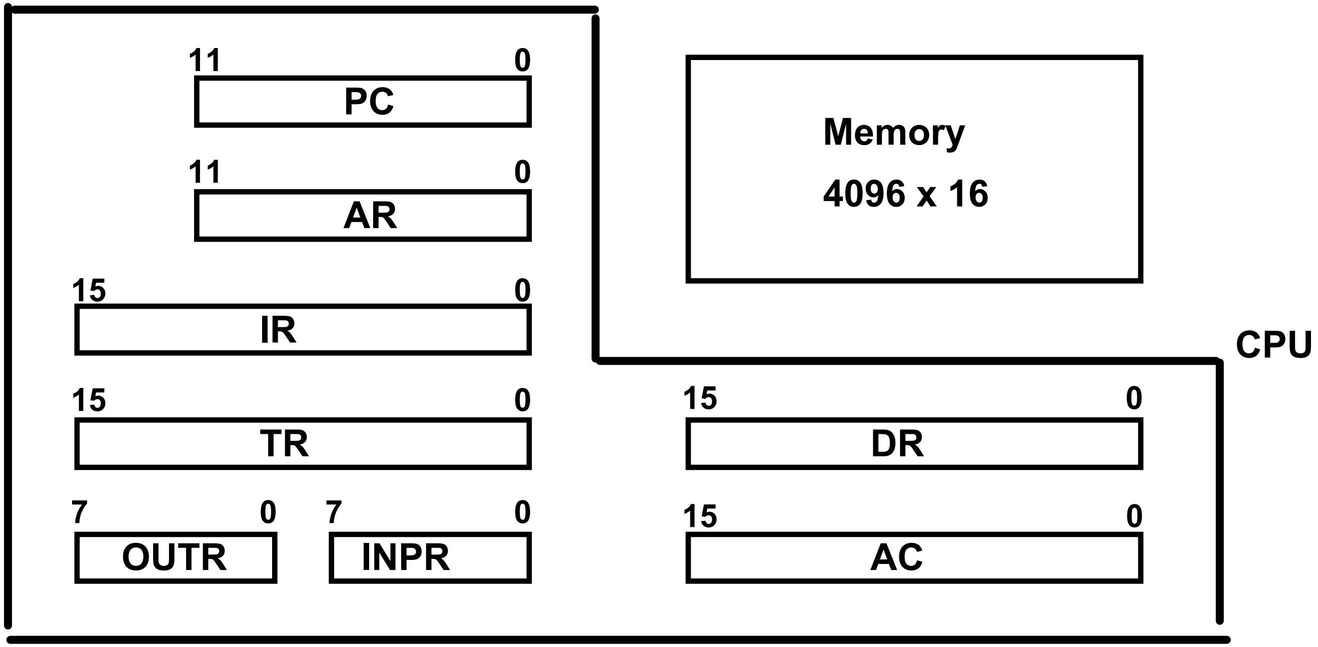
- A processor has many registers to hold instructions, addresses, data, etc
- The processor has a register, the *Program Counter* (PC) that holds the memory address of the next instruction to get
  - Since the memory in the Basic Computer only has 4096 locations, the PC only needs 12 bits
- In a direct or indirect addressing, the processor needs to keep track of what locations in memory it is addressing: The *Address Register* (AR) is used for this
  - The AR is a 12 bit register in the Basic Computer
- When an operand is found, using either direct or indirect addressing, it is placed in the *Data Register* (DR). The processor then uses this value as data for its operation
- The Basic Computer has a single *general purpose register* – the *Accumulator* (AC)

PROCESSOR REGISTERS

- The significance of a general purpose register is that it can be referred to in instructions
  - e.g. load AC with the contents of a specific memory location; store the contents of AC into a specified memory location
- Often a processor will need a scratch register to store intermediate results or other temporary data; in the Basic Computer this is the *Temporary Register* (TR)
- The Basic Computer uses a very simple model of input/output (I/O) operations
  - Input devices are considered to send 8 bits of character data to the processor
  - The processor can send 8 bits of character data to output devices
- The *Input Register* (INPR) holds an 8 bit character gotten from an input device
- The *Output Register* (OUTR) holds an 8 bit character to be send to an output device

BASIC COMPUTER REGISTERS

Registers in the Basic Computer



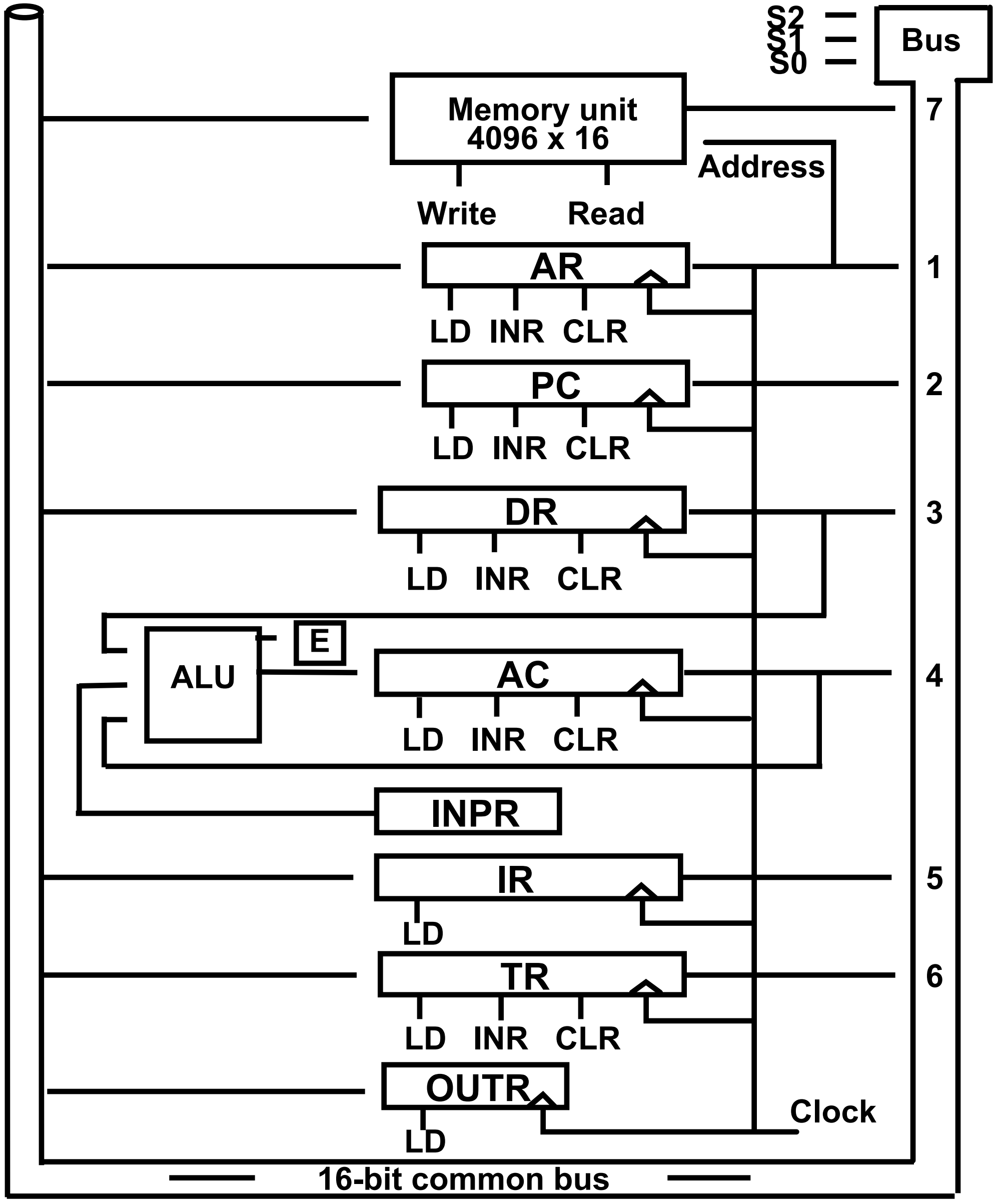
List of BC Registers

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

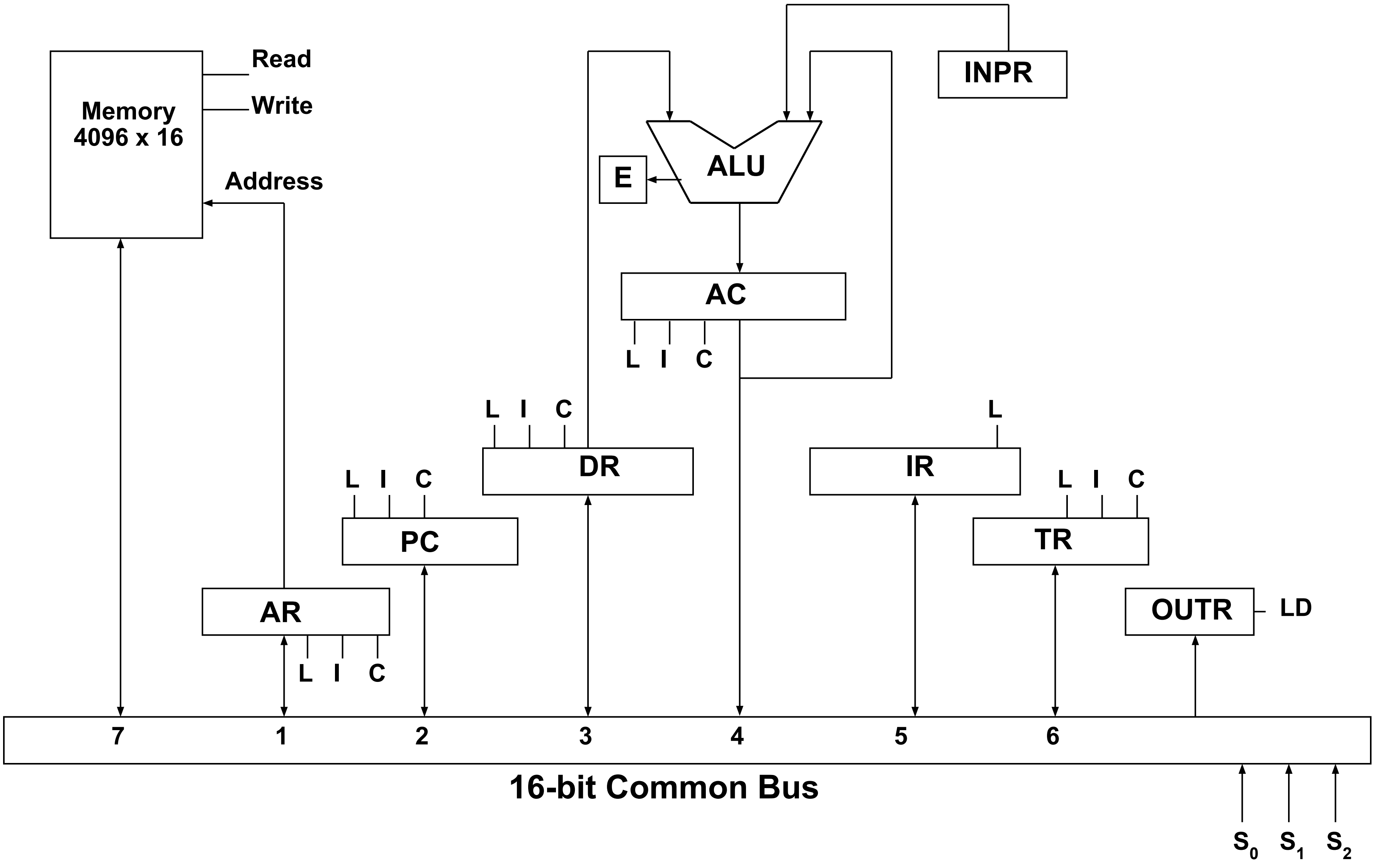
COMMON BUS SYSTEM

- The registers in the Basic Computer are connected using a bus
- This gives a savings in circuitry over complete connections between registers

COMMON BUS SYSTEM



COMMON BUS SYSTEM



COMMON BUS SYSTEM

- Three control lines, S_2, S_1, and S_0 control which register the bus selects as its input

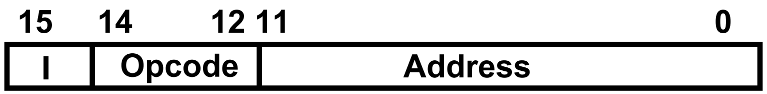
S_2	S_1	S_0	Register
0	0	0	x
0	0	1	AR
0	1	0	PC
0	1	1	DR
1	0	0	AC
1	0	1	IR
1	1	0	TR
1	1	1	Memory

- Either one of the registers will have its load signal activated, or the memory will have its read signal activated
  - Will determine where the data from the bus gets loaded
- The 12-bit registers, AR and PC, have 0's loaded onto the bus in the high order 4 bit positions
- When the 8-bit register OUTR is loaded from the bus, the data comes from the low order 8 bits on the bus

BASIC COMPUTER INSTRUCTIONS

- **Basic Computer Instruction Format**

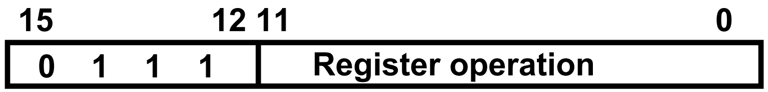
Memory-Reference Instructions (OP-code = 000 ~ 110)



15 14 12 11 0

I Opcode Address

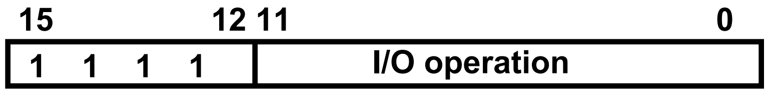
Register-Reference Instructions (OP-code = 111, I = 0)



15 12 11 0

0 1 1 1 Register operation

Input-Output Instructions (OP-code =111, I = 1)



15 12 11 0

1 1 1 1 I/O operation

BASIC COMPUTER INSTRUCTIONS

	Hex Code		
Symbol	I = 0	I = 1	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into 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 instr. if AC is positive
SNA	7008		Skip next instr. if AC is negative
SZA	7004		Skip next instr. if AC is zero
SZE	7002		Skip next instr. if E is zero
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

INSTRUCTION SET COMPLETENESS

A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable.

- Instruction Types

Functional Instructions
- Arithmetic, logic, and shift instructions
- ADD, CMA, INC, CIR, CIL, AND, CLA

Transfer Instructions
- Data transfers between the main memory and the processor registers
- LDA, STA

Control Instructions
- Program sequencing and control
- BUN, BSA, ISZ

Input/Output Instructions
- Input and output
- INP, OUT

CONTROL UNIT

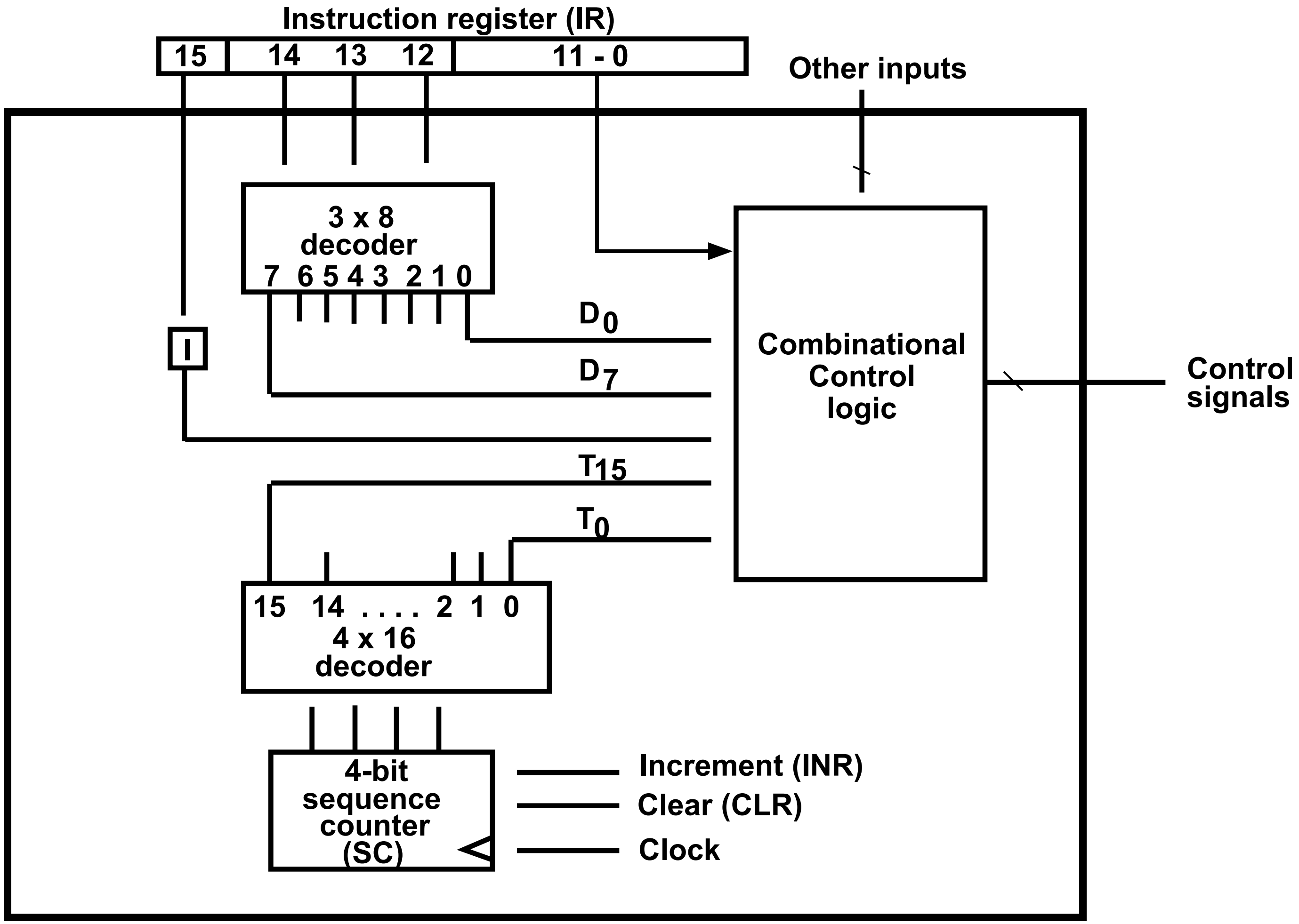
- Control unit (CU) of a processor translates from machine instructions to the control signals for the microoperations that implement them

- Control units are implemented in one of two ways
- *Hardwired* Control
  - CU is made up of sequential and combinational circuits to generate the control signals
- *Microprogrammed* Control
  - A control memory on the processor contains microprograms that activate the necessary control signals

- We will consider a hardwired implementation of the control unit for the Basic Computer

TIMING AND CONTROL

Control unit of Basic Computer



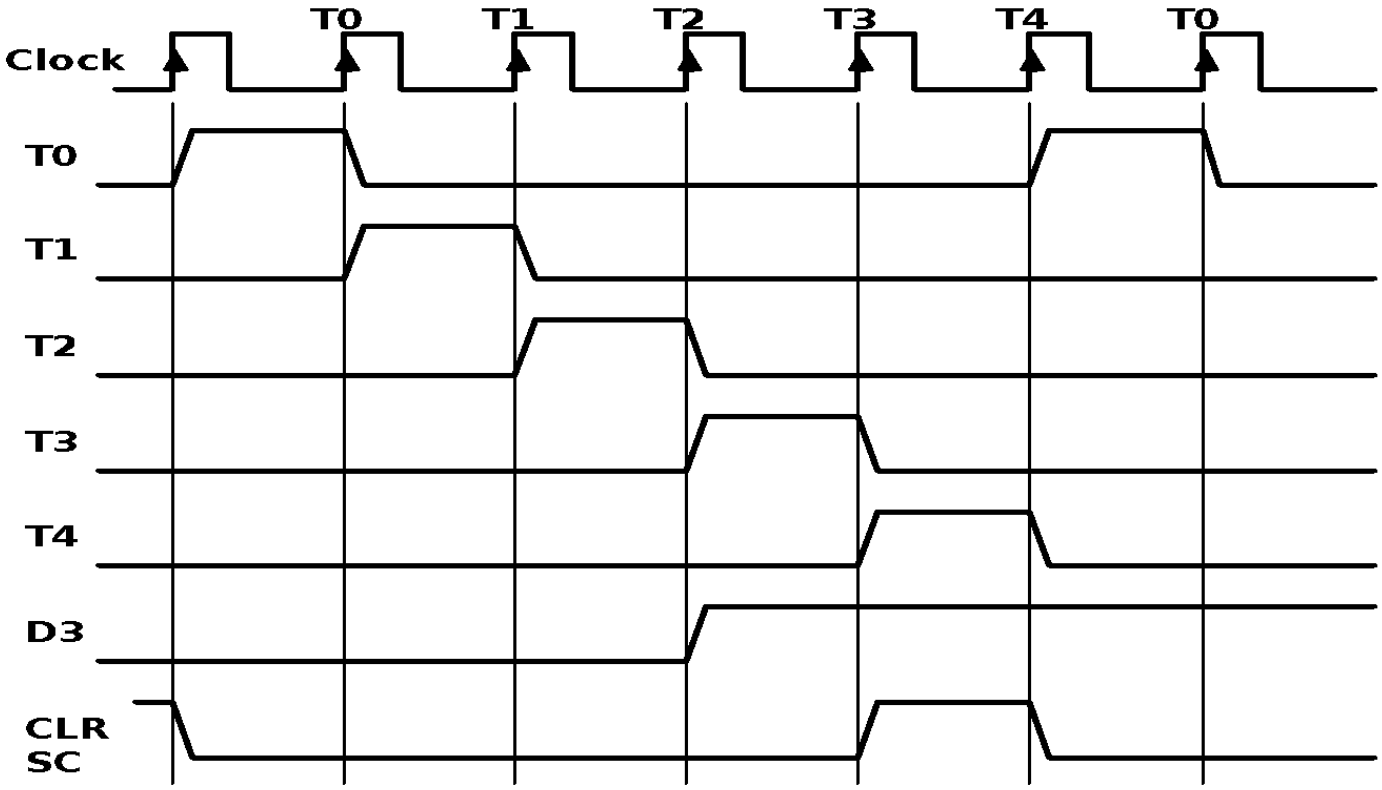
TIMING SIGNALS

- Generated by 4-bit sequence counter and 4×16 decoder
- The SC can be incremented or cleared.

- Example: $T_0, T_1, T_2, T_3, T_4, T_0, T_1, \ldots$

  Assume: At time T_4, SC is cleared to 0 if decoder output D3 is active.

$$D_3T_4: SC \leftarrow 0$$



INSTRUCTION CYCLE

- **In Basic Computer, a machine instruction is executed in the following cycle:**
  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**

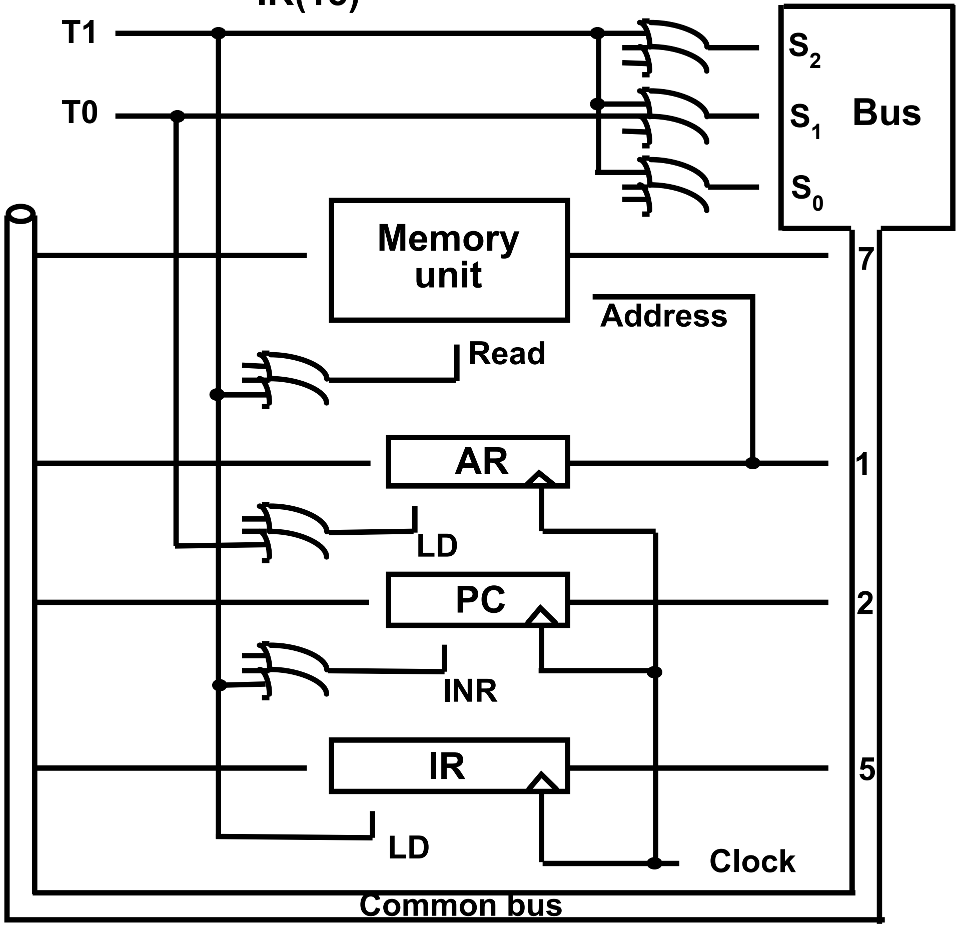
- **After an instruction is executed, the cycle starts again at step 1, for the next instruction**

- ***Note*: Every different processor has its own (different) instruction cycle**

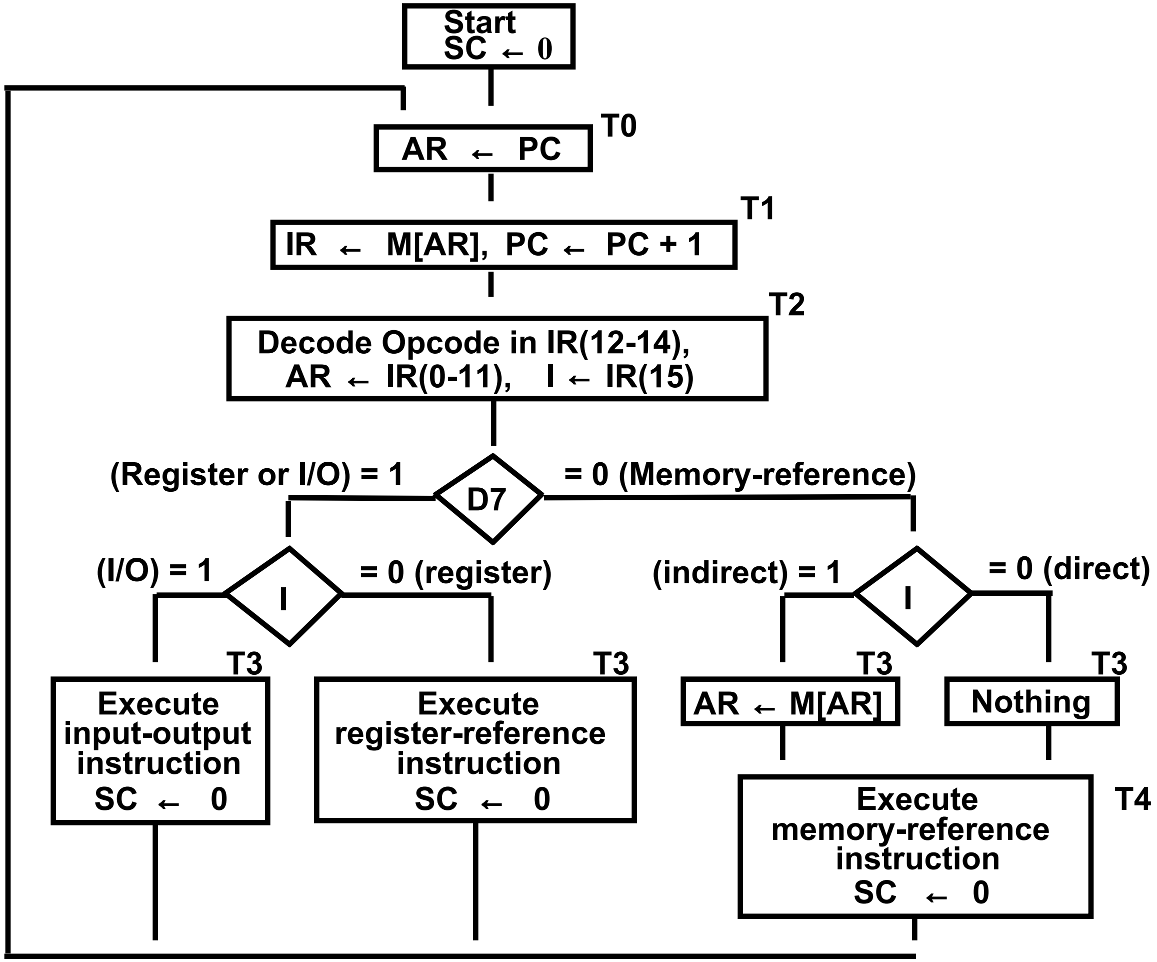
FETCH and DECODE

- **Fetch and Decode**

T0: AR ← PC ($S_0S_1S_2$=010, T0=1)
T1: IR ← M [AR], PC ← PC + 1 (S0S1S2=111, T1=1)
T2: D0, . . . , D7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)



DETERMINE THE TYPE OF INSTRUCTION



D'_7IT_3: AR ← M[AR]
$D'_7I'T_3$: Nothing
$D_7I'T_3$: Execute a register-reference instr.
D_7IT_3: Execute an input-output instr.

REGISTER REFERENCE INSTRUCTIONS

Register Reference Instructions are identified when

- $D_7 = 1$, $I = 0$
- Register Ref. Instr. is specified in $b_0 \sim b_{11}$ of IR
- Execution starts with timing signal T_3

$r = D_7 I'T_3$ => Register Reference Instruction
$B_i = IR(i)$, i=0,1,2,...,11

r:		$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow AC'$
CME	rB_8:	$E \leftarrow E'$
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:	$S \leftarrow 0$ (S is a start-stop flip-flop)

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 $PC \leftarrow PC+1$

- The effective address of the instruction is in AR and was placed there during timing signal T_2 when I = 0, or during timing signal T_3 when I = 1
- Memory cycle is assumed to be short enough to complete in a CPU cycle
- The execution of MR instruction starts with T_4

AND to AC

D_0T_4: $DR \leftarrow M[AR]$ Read operand

D_0T_5: $AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$ AND with AC

ADD to AC

D_1T_4: $DR \leftarrow M[AR]$ Read operand

D_1T_5: $AC \leftarrow AC + DR$, $E \leftarrow C_{out}$, $SC \leftarrow 0$ Add to AC and store carry in E

MEMORY REFERENCE INSTRUCTIONS

LDA: Load to AC

D_2T_4: $DR \leftarrow M[AR]$

D_2T_5: $AC \leftarrow DR, SC \leftarrow 0$

STA: Store AC

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

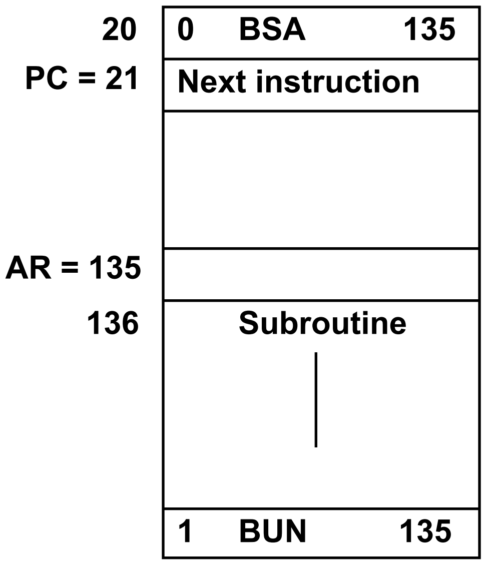
BUN: Branch Unconditionally

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

BSA: Branch and Save Return Address

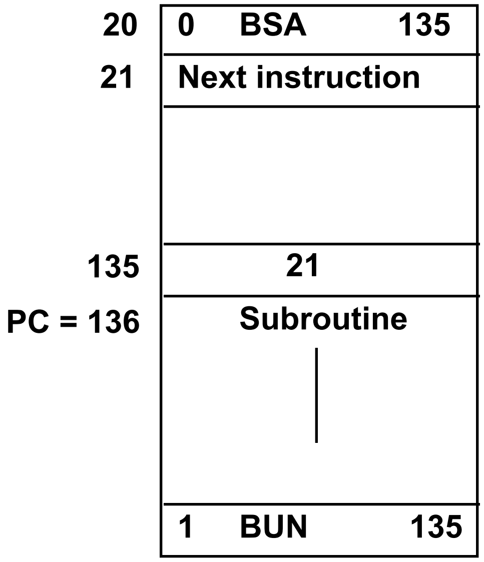
$M[AR] \leftarrow PC, PC \leftarrow AR + 1$

Memory, PC, AR at time T4



Memory

Memory, PC after execution



Memory

MEMORY REFERENCE INSTRUCTIONS

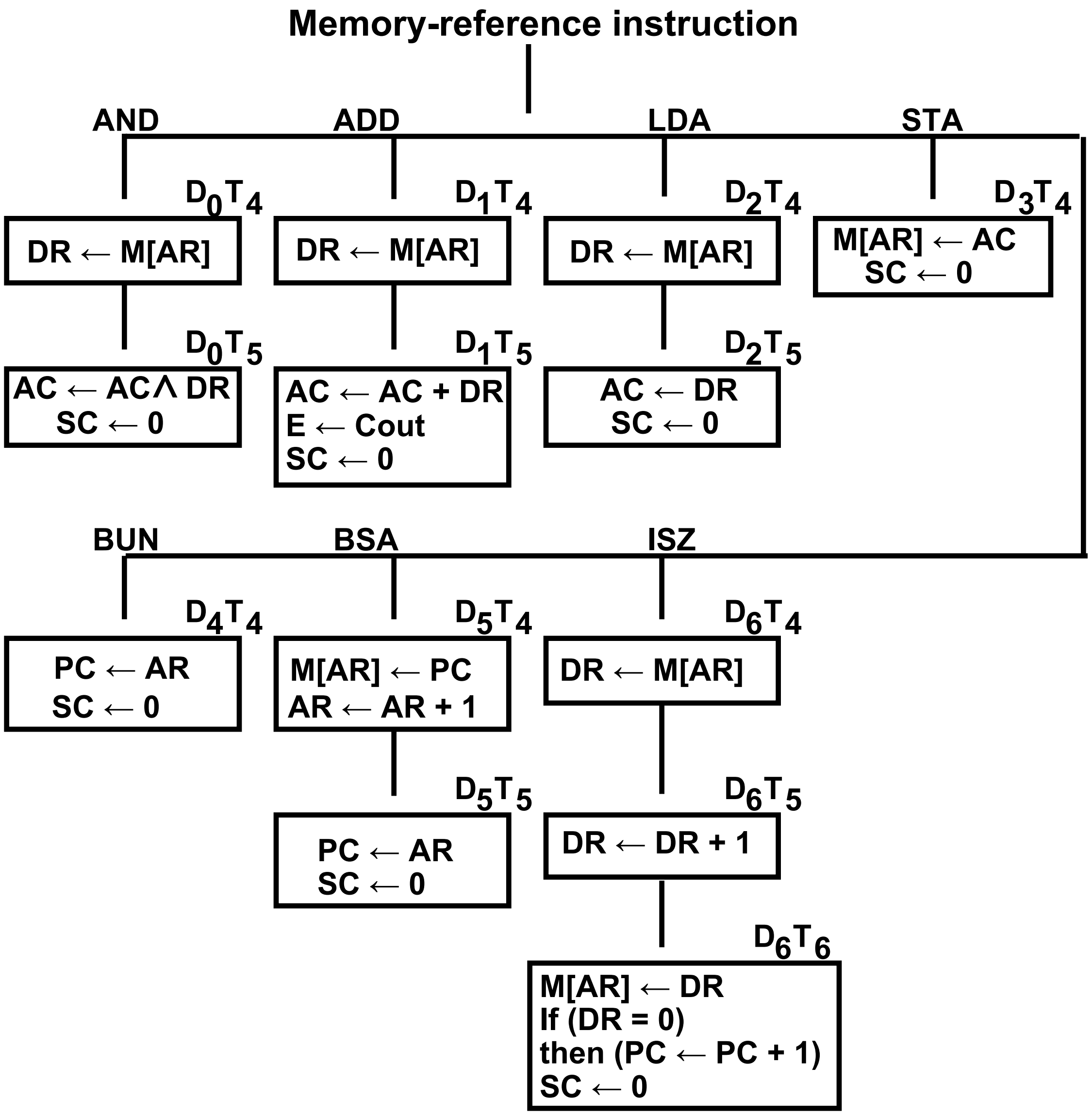
BSA:

D_5T_4: M[AR] ← PC, AR ← AR + 1
D_5T_5: PC ← AR, SC ← 0

ISZ: Increment and Skip-if-Zero

D_6T_4: DR ← M[AR]
D_6T_5: DR ← DR + 1
D_6T_4: M[AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ← 0

FLOWCHART FOR MEMORY REFERENCE INSTRUCTIONS



Memory-reference instruction

AND

D_0T_4

DR ← M[AR]

D_0T_5

AC ← AC∧ DR
SC ← 0

ADD

D_1T_4

DR ← M[AR]

D_1T_5

AC ← AC + DR
E ← Cout
SC ← 0

LDA

D_2T_4

DR ← M[AR]

D_2T_5

AC ← DR
SC ← 0

STA

D_3T_4

M[AR] ← AC
SC ← 0

BUN

D_4T_4

PC ← AR
SC ← 0

BSA

D_5T_4

M[AR] ← PC
AR ← AR + 1

D_5T_5

PC ← AR
SC ← 0

ISZ

D_6T_4

DR ← M[AR]

D_6T_5

DR ← DR + 1

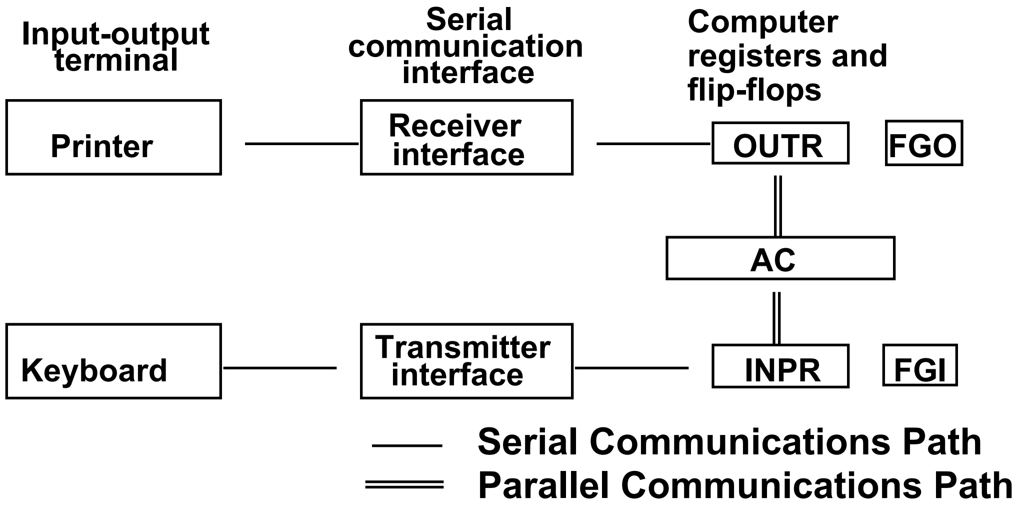
D_6T_6

M[AR] ← DR
If (DR = 0)
then (PC ← PC + 1)
SC ← 0

INPUT-OUTPUT AND INTERRUPT

A Terminal with a keyboard and a Printer

- Input-Output Configuration


Input-output terminal
Serial communication interface
Computer registers and flip-flops
Printer
Receiver interface
OUTR
FGO
AC
Keyboard
Transmitter interface
INPR
FGI
Serial Communications Path
Parallel Communications Path

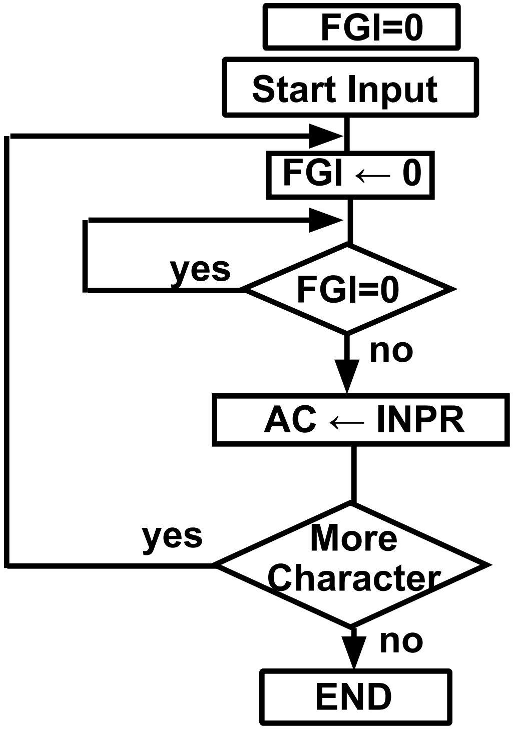
INPR	Input register - 8 bits
OUTR	Output register - 8 bits
FGI	Input flag - 1 bit
FGO	Output flag - 1 bit
IEN	Interrupt enable - 1 bit

- The terminal sends and receives serial information
- The serial info. from the keyboard is shifted into INPR
- The serial info. for the printer is stored in the OUTR
- INPR and OUTR communicate with the terminal serially and with the AC in parallel.
- The flags are needed to *synchronize* the timing difference between I/O device and the computer

PROGRAM CONTROLLED DATA TRANSFER

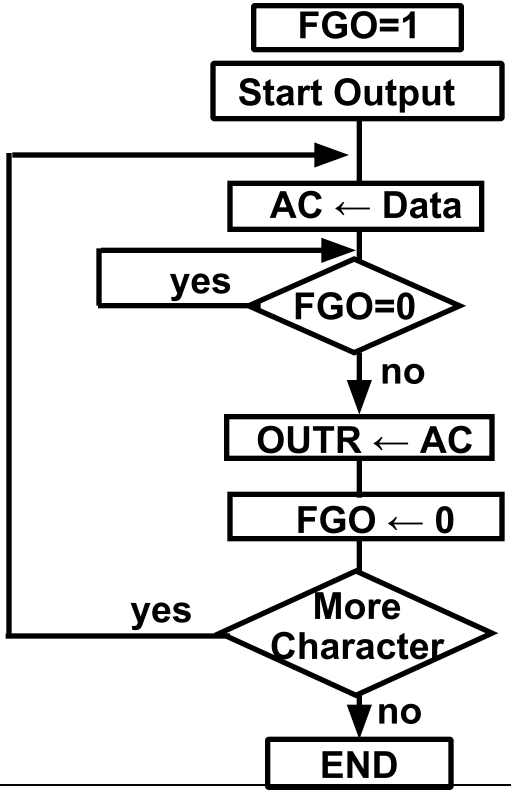
-- CPU --

```
--
/* Input */        /* Initially FGI = 0 */
   loop:  If FGI = 0 goto loop
               AC ←  INPR,  FGI ← 0

/* Output */        /* Initially FGO = 1 */
   loop:  If FGO = 0 goto loop
               OUTR ←  AC,  FGO ← 0
```



-- I/O Device

```
loop: If FGI = 1 goto loop
         INPR ←  new data, FGI ← 1

loop: If FGO = 1 goto loop
         consume OUTR, FGO ← 1
```



INPUT-OUTPUT INSTRUCTIONS

$D_7IT_3 = p$

$IR(i) = B_i$, $i = 6, \ldots, 11$

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0\text{-}7) \leftarrow INPR, FGI \leftarrow 0$	Input char. to AC
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{-}7), FGO \leftarrow 0$	Output char. from AC
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

PROGRAM-CONTROLLED INPUT/OUTPUT

- Program-controlled I/O
  - Continuous CPU involvement
    I/O takes valuable CPU time
  - CPU slowed down to I/O speed
  - Simple
  - Least hardware

Input

```
LOOP,   SKI   DEV
        BUN   LOOP
        INP   DEV
```

Output

```
LOOP,   LDA   DATA
LOP,    SKO   DEV
        BUN   LOP
        OUT   DEV
```

INTERRUPT INITIATED INPUT/OUTPUT

- Open communication only when some data has to be passed --> *interrupt*.

- The I/O interface, instead of the CPU, monitors the I/O device.

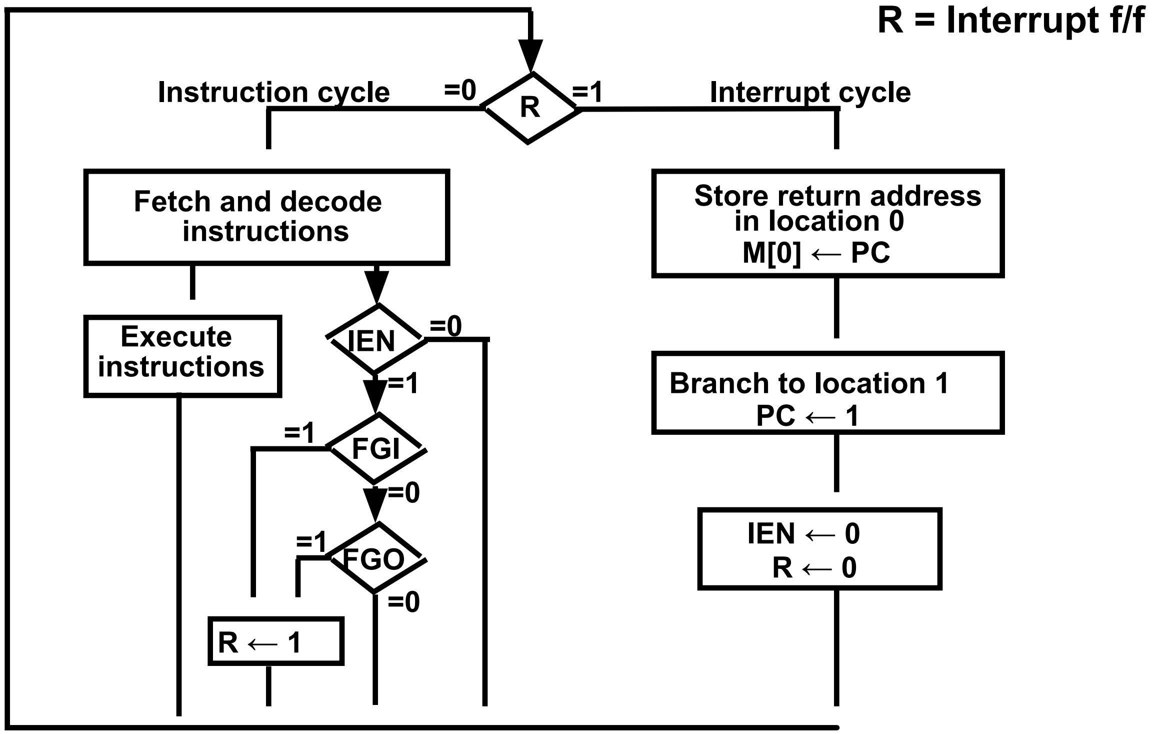
- When the interface founds that the I/O device is ready for data transfer, it generates an interrupt request to the CPU

- Upon detecting an interrupt, the CPU stops momentarily the task it is doing, branches to the service routine to process the data transfer, and then returns to the task it was performing.

* IEN (Interrupt-enable flip-flop)

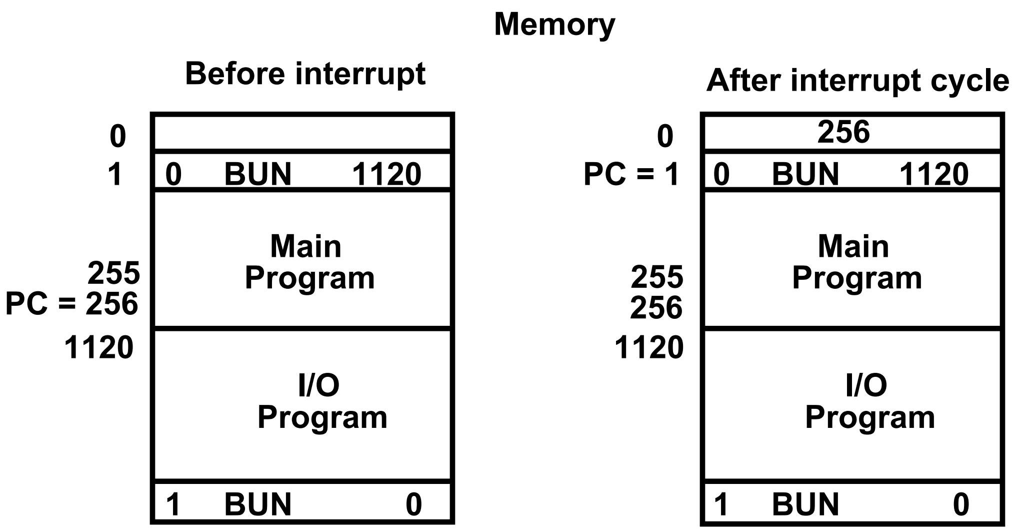
- can be set and cleared by instructions
- when cleared, the computer cannot be interrupted

FLOWCHART FOR INTERRUPT CYCLE



- The interrupt cycle is a HW implementation of a branch and save return address operation.
- At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1.
- At memory address 1, the programmer must store a branch instruction that sends the control to an interrupt service routine
- The instruction that returns the control to the original program is "indirect BUN 0"

REGISTER TRANSFER OPERATIONS IN INTERRUPT CYCLE



Register Transfer Statements for Interrupt Cycle

- R F/F $\leftarrow 1$ if $IEN\,(FGI + FGO)T_0'T_1'T_2'$

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

- The fetch and decode phases of the instruction cycle must be modified □Replace T_0, T_1, T_2 with $R'T_0$, $R'T_1$, $R'T_2$
- The interrupt cycle :

RT_0: $AR \leftarrow 0$, $TR \leftarrow PC$

RT_1: $M[AR] \leftarrow TR$, $PC \leftarrow 0$

RT_2: $PC \leftarrow PC + 1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$

FURTHER QUESTIONS ON INTERRUPT

How can the CPU recognize the device requesting an interrupt ?

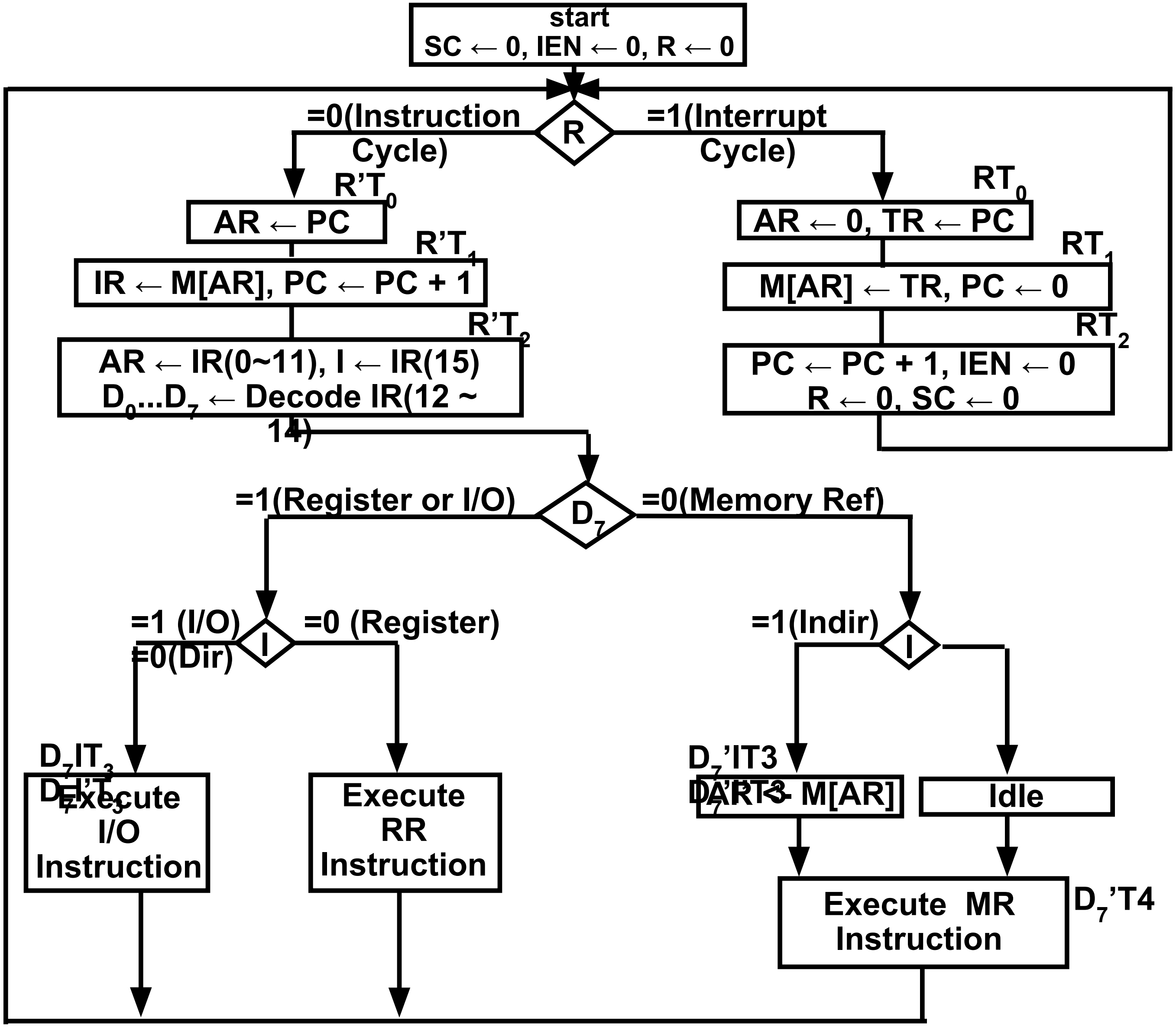
Since different devices are likely to require different interrupt service routines, how can the CPU obtain the starting address of the appropriate routine in each case ?

Should any device be allowed to interrupt the CPU while another interrupt is being serviced ?

How can the situation be handled when two or more interrupt requests occur simultaneously ?

COMPLETE COMPUTER DESCRIPTION

Flowchart of Operations



start

SC ← 0, IEN ← 0, R ← 0

R

=0(Instruction Cycle)

=1(Interrupt Cycle)

$R'T_0$

AR ← PC

$R'T_1$

IR ← M[AR], PC ← PC + 1

$R'T_2$

AR ← IR(0~11), I ← IR(15)

$D_0...D_7$ ← Decode IR(12 ~ 14)

RT_0

AR ← 0, TR ← PC

RT_1

M[AR] ← TR, PC ← 0

RT_2

PC ← PC + 1, IEN ← 0

R ← 0, SC ← 0

D_7

=1(Register or I/O)

=0(Memory Ref)

I

=1 (I/O)

=0 (Register)

I

=1(Indir)

=0(Dir)

D_7IT_3

$D_7I'T_3$

Execute I/O Instruction

Execute RR Instruction

D_7'IT3

D_7'I'T3

AR <- M[AR]

Idle

Execute MR Instruction

D_7'T4

COMPLETE COMPUTER DESCRIPTION

Microoperations

Fetch	$R'T_0$:	AR ← PC
	$R'T_1$:	IR ← M[AR], PC ← PC + 1
Decode	$R'T_2$:	D0, ..., D7 ← Decode IR(12 ~ 14), AR ← IR(0 ~ 11), I ← IR(15)
Indirect	$D_7'IT_3$:	AR ← M[AR]
Interrupt		
	$T_0'T_1'T_2'(IEN)(FGI + FGO)$:	R ← 1
	RT_0:	AR ← 0, TR ← PC
	RT_1:	M[AR] ← TR, PC ← 0
	RT_2:	PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0
Memory-Reference		
AND	D_0T_4:	DR ← M[AR]
	D_0T_5:	AC ← AC $\wedge$ DR, SC ← 0
ADD	D_1T_4:	DR ← M[AR]
	D_1T_5:	AC ← AC + DR, E ← C_{out}, SC ← 0
LDA	D_2T_4:	DR ← M[AR]
	D_2T_5:	AC ← DR, SC ← 0
STA	D_3T_4:	M[AR] ← AC, SC ← 0
BUN	D_4T_4:	PC ← AR, SC ← 0
BSA	D_5T_4:	M[AR] ← PC, AR ← AR + 1
	D_5T_5:	PC ← AR, SC ← 0
ISZ	D_6T_4:	DR ← M[AR]
	D_6T_5:	DR ← DR + 1

	D_6T_6:	M[AR] ← DR, if(DR=0) then (PC ← PC + 1),
		SC ← 0

COMPLETE COMPUTER DESCRIPTION

Microoperations

Register-Reference		
	$D_7I'T_3 = r$	(Common to all register-reference instr)
	$IR(i) = B_i$	(i = 0,1,2, ..., 11)
	r:	SC ← 0
CLA	rB_{11}:	AC ← 0
CLE	rB_{10}:	E ← 0
CMA	rB_9:	AC ← AC′
CME	rB_8:	E ← E′
CIR	rB_7:	AC ← shr AC, AC(15) ← E, E ← AC(0)
CIL	rB_6:	AC ← shl AC, AC(0) ← E, E ← AC(15)
INC	rB_5:	AC ← AC + 1
SPA	rB_4:	If(AC(15) =0) then (PC ← PC + 1)
SNA	rB_3:	If(AC(15) =1) then (PC ← PC + 1)
SZA	rB_2:	If(AC = 0) then (PC ← PC + 1)
SZE	rB_1:	If(E=0) then (PC ← PC + 1)
HLT	rB_0:	S ← 0
Input-Output	$D_7IT_3 = p$	(Common to all input-output
	$IR(i) = B_i$	instructions)
	p:	(i = 6,7,8,9,10,11)
INP	pB_{11}:	SC ← 0
OUT	pB_{10}:	AC(0-7) ← INPR, FGI ← 0
SKI	pB_9:	OUTR ← AC(0-7), FGO ← 0
SKO	pB_8:	If(FGI=1) then (PC ← PC + 1)
ION	pB_7:	If(FGO=1) then (PC ← PC + 1)
IOF	pB_6:	IEN ← 1
		IEN ← 0

DESIGN OF BASIC COMPUTER(BC)

Hardware Components of BC

A memory unit: 4096 x 16.
Registers:

AR, PC, DR, AC, IR, TR, OUTR, INPR, and SC

Flip-Flops(Status):

I, S, E, R, IEN, FGI, and FGO

Decoders: a 3x8 Opcode decoder

a 4x16 timing decoder

Common bus: 16 bits

Control logic gates:

Adder and Logic circuit: Connected to AC

Control Logic Gates

- Input Controls of the nine registers
- Read and Write Controls of memory
- Set, Clear, or Complement Controls of the flip-flops
- S_2, S_1, S_0 Controls to select a register for the bus
- AC, and Adder and Logic circuit

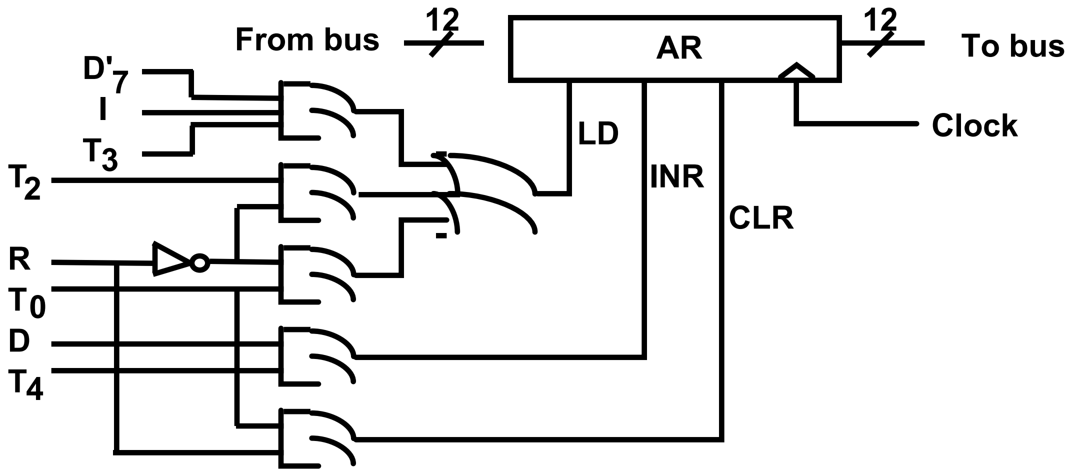
CONTROL OF REGISTERS AND MEMORY

Address Register; AR

Scan all of the register transfer statements that change the content of AR:

$R'T_0$:	$AR \leftarrow PC$	LD(AR)
$R'T_2$:	$AR \leftarrow IR(0\text{-}11)$	LD(AR)
D'_7IT_3:	$AR \leftarrow M[AR]$	LD(AR)
RT_0:	$AR \leftarrow 0$	CLR(AR)
D_5T_4:	$AR \leftarrow AR + 1$	INR(AR)

$$LD(AR) = R'T_0 + R'T_2 + D'_7IT_3$$
$$CLR(AR) = RT_0$$
$$INR(AR) = D_5T_4$$

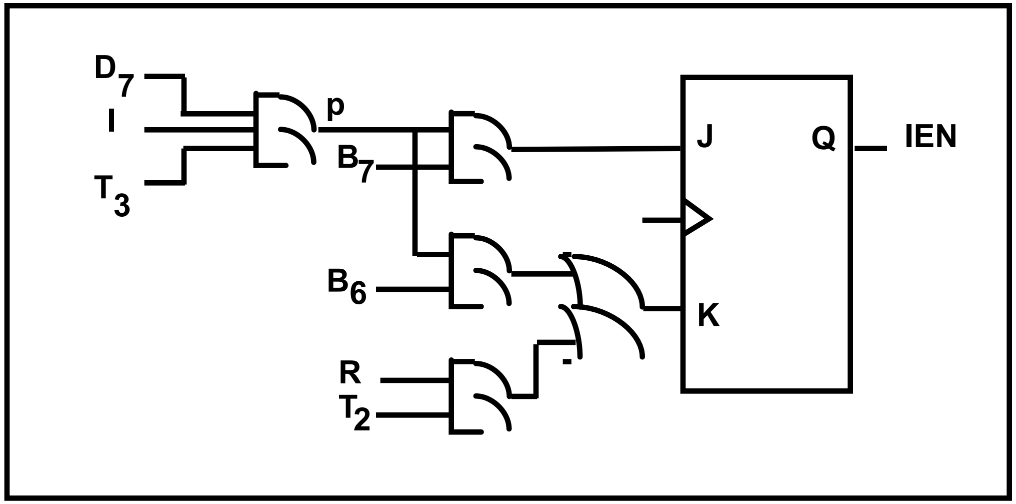


CONTROL OF FLAGS

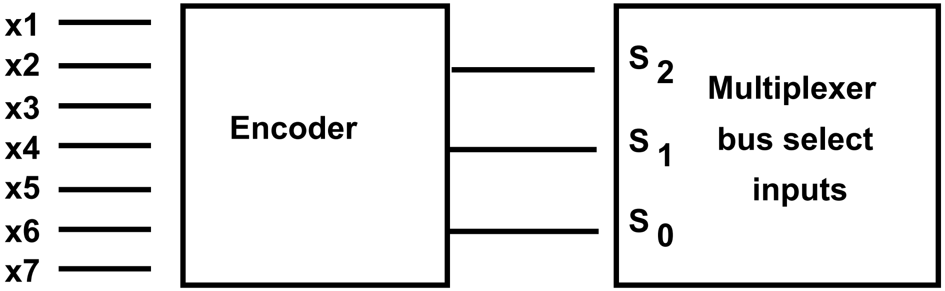
IEN: Interrupt Enable Flag

pB_7: IEN ← 1 (I/O Instruction)
pB_6: IEN ← 0 (I/O Instruction)
RT_2: IEN ← 0 (Interrupt)

$p = D_7IT_3$ (Input/Output Instruction)



CONTROL OF COMMON BUS



x1	x2	x3	x4	x5	x6	x7	S2	S1	S0	selected register
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

For AR

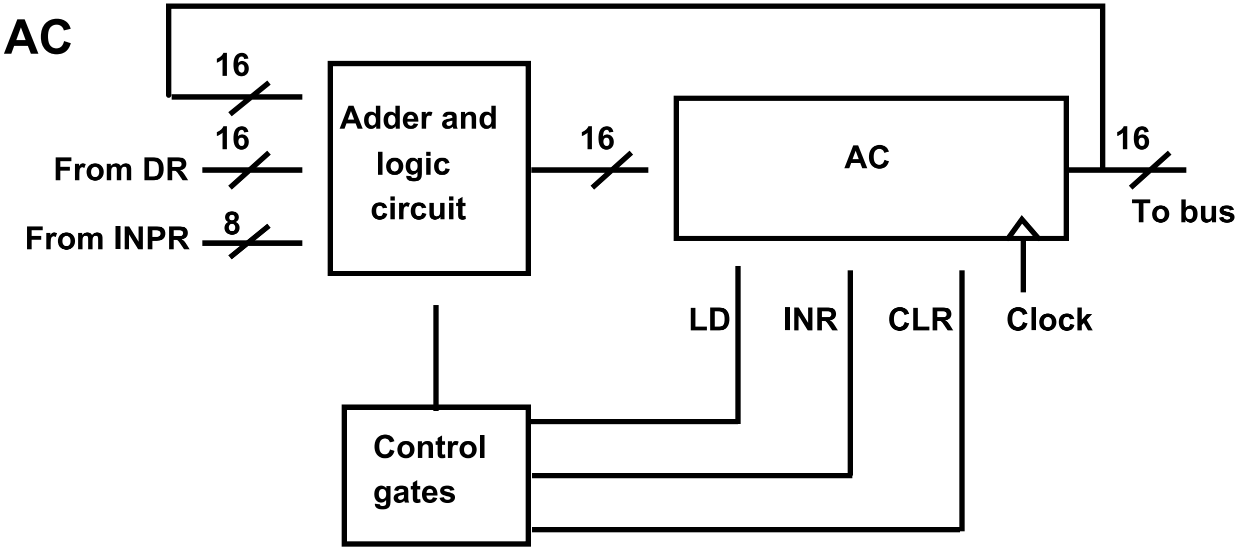
D_4T_4: PC ← AR
D_5T_5: PC ← AR

⇩

$x1 = D_4T_4 + D_5T_5$

DESIGN OF ACCUMULATOR LOGIC

Circuits associated with AC

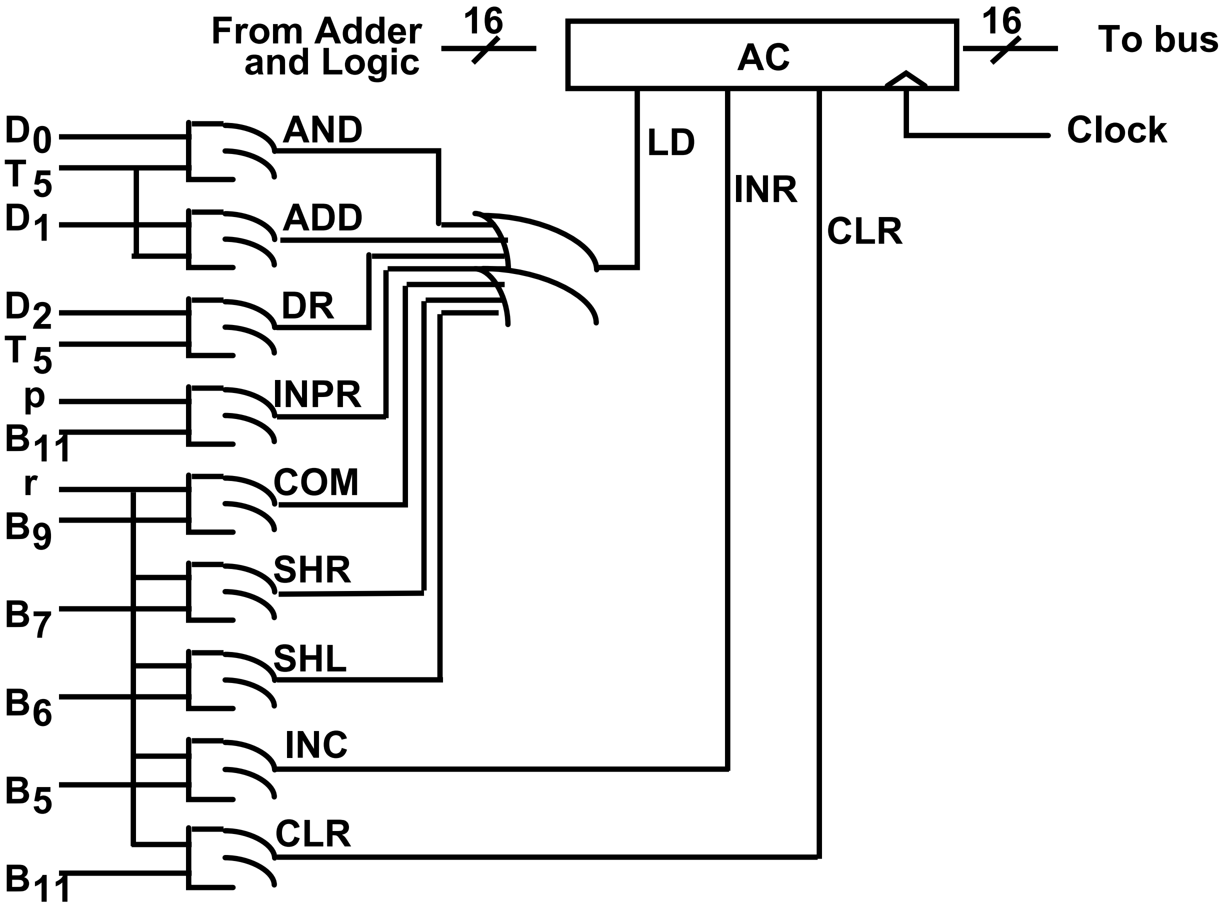


All the statements that change the content of AC

D_0T_5:	AC ← AC ∧ DR	AND with DR
D_1T_5:	AC ← AC + DR	Add with DR
D_2T_5:	AC ← DR	Transfer from DR
pB_{11}:	AC(0-7) ← INPR	Transfer from
INPR		
rB_9:	AC ← AC′	Complement
rB_7 :	AC ← shr AC, AC(15) ← E	Shift right
rB_6 :	AC ← shl AC, AC(0) ← E	Shift left
rB_{11} :	AC ← 0	Clear
rB_5 :	AC ← AC + 1	Increment

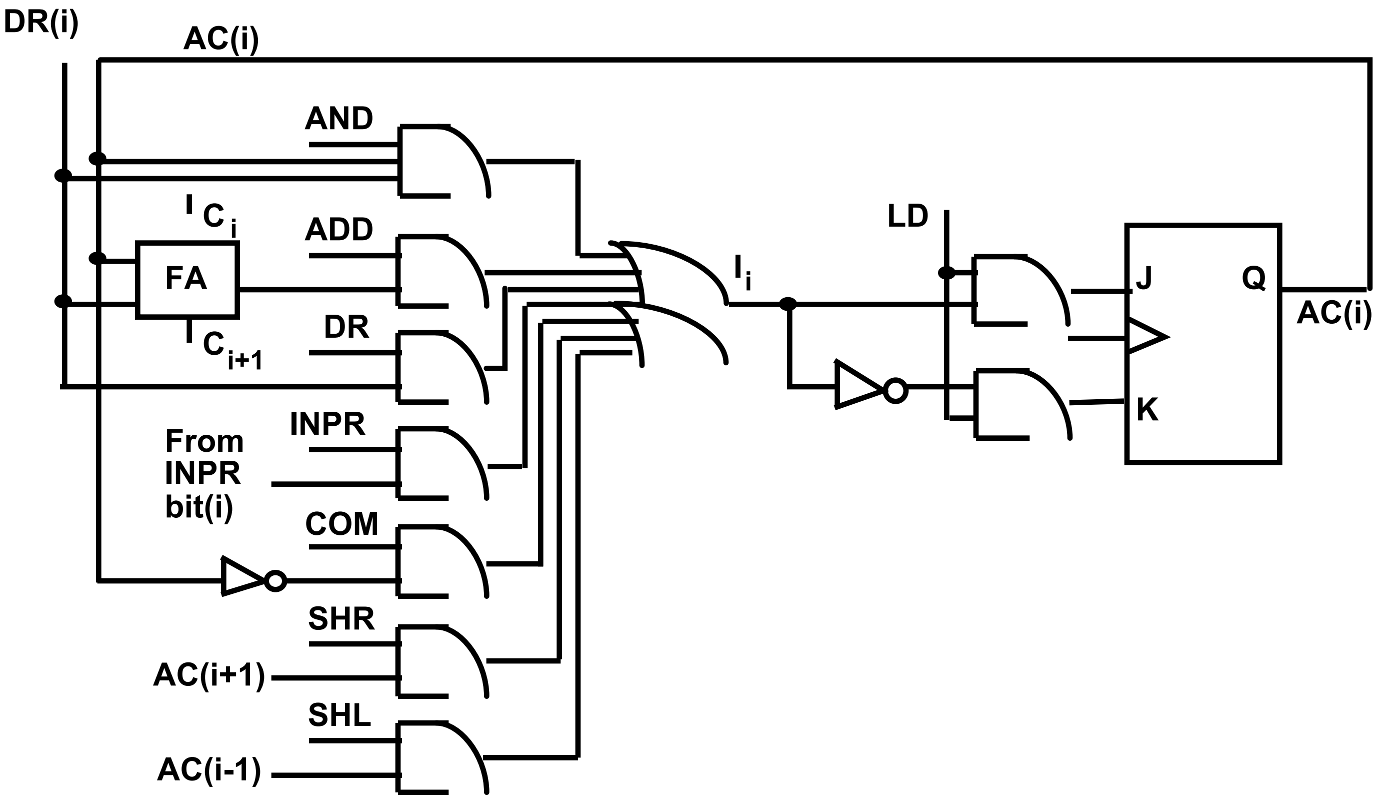
CONTROL OF AC REGISTER

Gate structures for controlling the LD, INR, and CLR of AC



ALU (ADDER AND LOGIC CIRCUIT)

One stage of Adder and Logic circuit



PROGRAMMING THE BASIC COMPUTER

Introduction

Machine Language

Assembly Language

Assembler

Program Loops

Programming Arithmetic and Logic Operations

Subroutines

Input-Output Programming

INTRODUCTION

Those concerned with computer architecture should have a knowledge of both hardware and software because the two branches influence each other.

Instruction Set of the *Basic Computer*

Symbol	*Hexa code*	*Description*
AND	0 or 8	AND M to AC
ADD	1 or 9	Add M to AC, carry to E
LDA	2 or A	Load AC from M
STA	3 or B	Store AC in M
BUN	4 or C	Branch unconditionally to m
BSA	5 or D	Save return address in m and branch to m+1
ISZ	6 or E	Increment M and skip if zero
CLA	7800	Clear AC
CLE	7400	Clear E
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	Circulate right E and AC
CIL	7040	Circulate left E and AC
INC	7020	Increment AC, carry to E
SPA	7010	Skip if AC is positive
SNA	7008	Skip if AC is negative
SZA	7004	Skip if AC is zero
SZE	7002	Skip if E is zero
HLT	7001	Halt computer
INP	F800	Input information and clear flag
OUT	F400	Output information and clear flag
SKI	F200	Skip if input flag is on
SKO	F100	Skip if output flag is on
ION	F080	Turn interrupt on
IOF	F040	Turn interrupt off

m: effective address
M: memory word (operand)
found at m

MACHINE LANGUAGE

- **Program**

  A list of instructions or statements for directing the computer to perform a required data processing task

- **Various types of programming languages**

  - Hierarchy of programming languages

  - **Machine-language**

    - Binary code

    - Octal or hexadecimal code

  - **Assembly-language (Assembler)**

    - Symbolic code

  - **High-level language (Compiler)**

COMPARISON OF PROGRAMMING LANGUAGES

- **Binary Program to Add Two Numbers**

Location	Instruction Code
0	0010 0000 0000 0100
1	0001 0000 0000 0101
10	0011 0000 0000 0110
11	0111 0000 0000 0001
100	0000 0000 0101 0011
101	1111 1111 1110 1001
110	0000 0000 0000 0000

- **Hexa program**

Location	Instruction
000	2004
001	1005
002	3006
003	7001
004	0053
005	FFE9
006	0000

- **Program with Symbolic OP-Code**

Location	Instruction		Comments
000	LDA	004	Load 1st operand into AC
001	ADD	005	Add 2nd operand to AC
002	STA	006	Store sum in location 006
003	HLT		Halt computer
004	0053		1st operand
005	FFE9		2nd operand (negative)
006	0000		Store sum here

- **Assembly-Language Program**

```
      ORG    0     /Origin of program is location 0
      LDA    A     /Load  operand from location A
      ADD    B     /Add  operand from location B
      STA    C     /Store sum in location C
      HLT          /Halt computer
A,    DEC    83    /Decimal operand
B,    DEC    -23   /Decimal operand
C,    DEC    0     /Sum stored in location C
      END          /End of symbolic program
```

- **Fortran Program**

```
INTEGER  A, B, C
DATA  A,83 / B,-23
C = A + B
END
```

ASSEMBLY LANGUAGE

Syntax of the BC assembly language

Each line is arranged in three columns called fields

Label field

- May be empty or may specify a symbolic address consists of up to 3 characters
- Terminated by a comma

Instruction field

- Specifies a machine or a pseudo instruction
- May specify one of
  * Memory reference instr. (MRI)

    MRI consists of two or three symbols separated by spaces.

    ADD OPR (direct address MRI)

    ADD PTR I (indirect address MRI)
  * Register reference or input-output instr.

    Non-MRI does not have an address part
  * Pseudo instr. with or without an operand

    Symbolic address used in the instruction field must be defined somewhere as a label

Comment field

- May be empty or may include a comment

PSEUDO-INSTRUCTIONS

ORG N

Hexadecimal number N is the memory loc. for the instruction or operand listed in the following line

END

Denotes the end of symbolic program

DEC N

Signed decimal number N to be converted to the binary

HEX N

Hexadecimal number N to be converted to the binary

Example: Assembly language program to subtract two numbers

```
          ORG  100          / Origin of program is location 100
          LDA  SUB          / Load subtrahend to AC
          CMA               / Complement AC
          INC               / Increment AC
          ADD  MIN          / Add minuend to AC
          STA  DIF          / Store difference
          HLT               / Halt computer
MIN,      DEC  83           / Minuend
SUB,      DEC  -23          / Subtrahend
DIF,      HEX  0            / Difference stored here
          END               / End of symbolic program
```

TRANSLATION TO BINARY

Hexadecimal Code		Symbolic Program	
Location	Content		
			ORG 100
100	2107		LDA SUB
101	7200		CMA
102	7020		INC
103	1106		ADD MIN
104	3108		STA DIF
105	7001		HLT
106	0053	MIN,	DEC 83
107	FFE9	SUB,	DEC -23
108	0000	DIF,	HEX 0
			END

ASSEMBLER - FIRST PASS -

Assembler

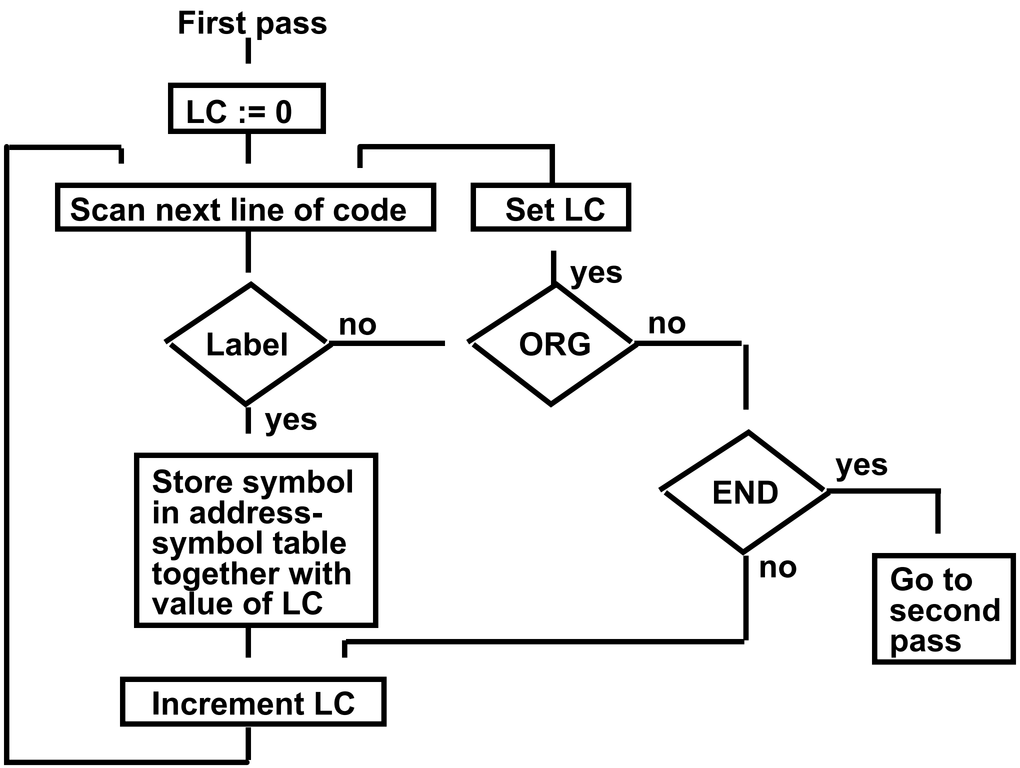
Source Program - Symbolic Assembly Language Program
Object Program - Binary Machine Language Program

Two pass assembler

1st pass: generates a table that correlates all user defined (address) symbols with their binary equivalent value

2nd pass: binary translation

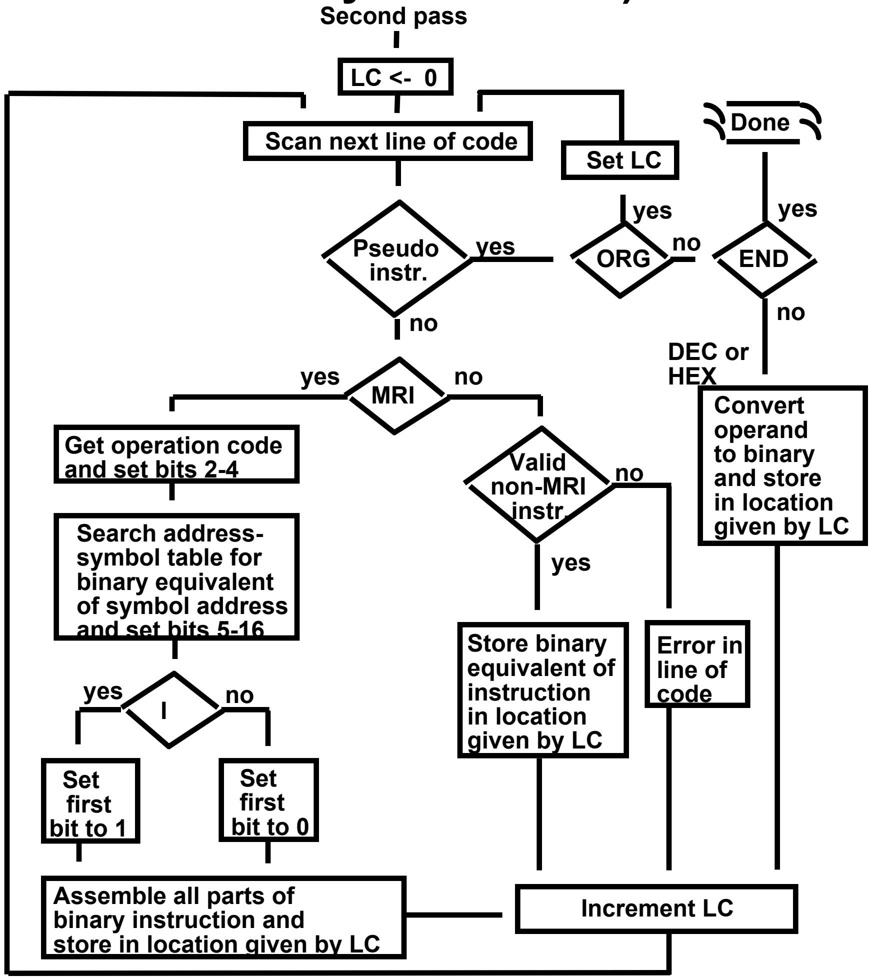
First pass



ASSEMBLER - SECOND PASS -

Second Pass

Machine instructions are translated by means of table-lookup procedures;
(1. Pseudo-Instruction Table, 2. MRI Table, 3. Non-MRI Table
4. Address Symbol Table)


Second pass
LC <- 0
Scan next line of code
Set LC
Done
yes
yes
Pseudo instr.
yes
ORG
no
END
no
no
DEC or HEX
yes
MRI
no
Convert operand to binary and store in location given by LC
Get operation code and set bits 2-4
Valid non-MRI instr.
no
Search address-symbol table for binary equivalent of symbol address and set bits 5-16
yes
yes
I
no
Store binary equivalent of instruction in location given by LC
Error in line of code
Set first bit to 1
Set first bit to 0
Assemble all parts of binary instruction and store in location given by LC
Increment LC

PROGRAM LOOPS

Loop: A sequence of instructions that are executed many times,
each with a different set of data

Fortran program to add 100 numbers:

```
        DIMENSION  A(100)
        INTEGER  SUM,  A
        SUM = 0
        DO  3  J = 1,  100
   3    SUM = SUM + A(J)
```

Assembly-language program to add 100 numbers:

```
            ORG  100            / Origin of program is HEX 100
            LDA  ADS            / Load first address of operand
            STA  PTR            / Store in pointer
            LDA  NBR            / Load -100
            STA  CTR            / Store in counter
            CLA                 / Clear AC
LOP,        ADD  PTR  I         / Add an operand to AC
            ISZ  PTR            / Increment pointer
            ISZ  CTR            / Increment counter
            BUN  LOP            / Repeat loop again
            STA  SUM            / Store sum
            HLT                 / Halt
ADS,        HEX  150            / First address of operands
PTR,        HEX  0              / Reserved for a pointer
NBR,        DEC  -100           / Initial value for a counter
CTR,        HEX  0              / Reserved for a counter
SUM,        HEX  0              / Sum is stored here
            ORG  150            / Origin of operands is HEX 150
            DEC  75             / First operand
               :
               :

            DEC  23             / Last operand
            END                 / End of symbolic program
```

PROGRAMMING ARITHMETIC AND LOGIC OPERATIONS

Implementation of Arithmetic and Logic Operations

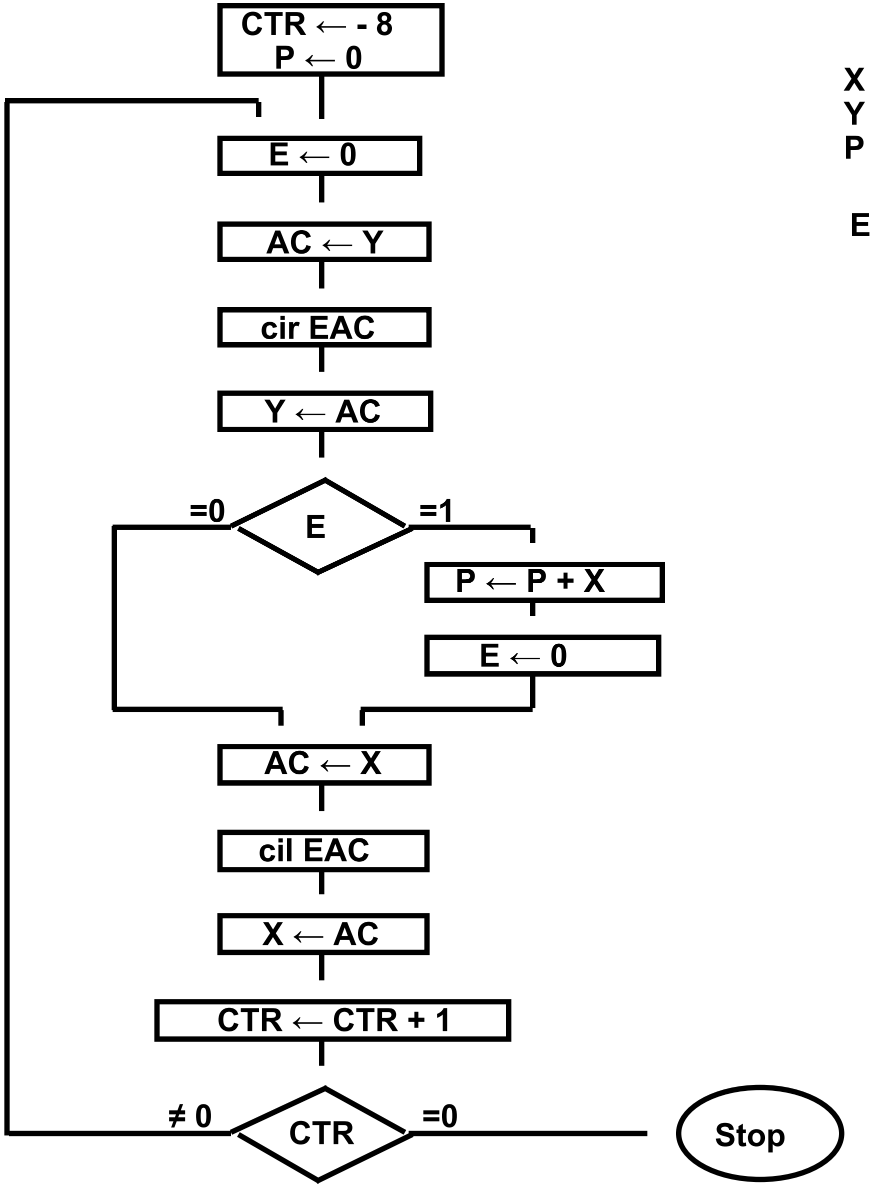
- Software Implementation
  - Implementation of an operation with a program using machine instruction set
  - Usually when the operation is not included in the instruction set

- Hardware Implementation
  - Implementation of an operation in a computer with one machine instruction

Software Implementation example:

* Multiplication
- For simplicity, unsigned positive numbers
- 8-bit numbers -> 16-bit product

FLOWCHART OF A PROGRAM - Multiplication -



X holds the multiplicand
Y holds the multiplier
P holds the product

Example with four significant digits

		P
X =	0000 1111	
Y =	0000 1011	0000 0000
	0000 1111	0000 1111
	0001 1110	0010 1101
	0000 0000	0010 1101
	0111 1000	1010 0101
	1010 0101	

ASSEMBLY LANGUAGE PROGRAM   - Multiplication -

```
        ORG  100
LOP,    CLE            / Clear E
        LDA  Y         / Load multiplier
        CIR            / Transfer multiplier bit to E
        STA  Y         / Store shifted multiplier
        SZE            / Check if bit is zero
        BUN  ONE       / Bit is one; goto ONE
        BUN  ZRO       / Bit is zero; goto ZRO
ONE,    LDA  X         / Load multiplicand
        ADD  P         / Add to partial product
        STA  P         / Store partial product
        CLE            / Clear E
ZRO,    LDA  X         / Load multiplicand
        CIL            / Shift left
        STA  X         / Store shifted multiplicand
        ISZ  CTR       / Increment counter
        BUN  LOP       / Counter not zero; repeat loop
        HLT            / Counter is zero; halt
CTR,    DEC  -8        / This location serves as a counter
X,      HEX  000F      / Multiplicand stored here
Y,      HEX  000B      / Multiplier stored here
P,      HEX  0         / Product formed here
        END
```

ASSEMBLY LANGUAGE PROGRAM
- Double Precision Addition -

```
LDA    AL          / Load A low
ADD    BL          / Add B low, carry in E
STA    CL          / Store in C low
CLA                / Clear AC
CIL                / Circulate to bring carry into AC(16)
ADD    AH          / Add A high and carry
ADD    BH          / Add B high
STA    CH          / Store in C high
HLT
```

ASSEMBLY LANGUAGE PROGRAM
- Logic and Shift Operations -

- **Logic operations**

  - BC instructions : AND, CMA, CLA

  - Program for OR operation

```
LDA    A              / Load 1st operand
CMA                   / Complement to get A'
STA    TMP            / Store in a temporary location
LDA    B              / Load 2nd operand B
CMA                   / Complement to get B'
AND    TMP            / AND with A' to get A' AND B'
CMA                   / Complement again to get A OR B
```

- **Shift operations - BC has *Circular Shift* only**

  - Logical shift-right operation

```
CLE
CIR
```

  - Logical shift-left operation

```
CLE
CIL
```

  - Arithmetic right-shift operation

```
CLE     / Clear E to 0
SPA     / Skip if AC is positive
CME     / AC is negative
CIR     / Circulate E and AC
```

SUBROUTINES

Subroutine

- A set of common instructions that can be used in a program many times.
- Subroutine *linkage* : a procedure for branching to a subroutine and returning to the main program

Example

```
Loc.
100             ORG  100        / Main program
101             LDA  X          / Load X
102             BSA  SH4        / Branch to subroutine
103             STA  X          / Store shifted number
104             LDA  Y          / Load Y
105             BSA  SH4        / Branch to subroutine again
106             STA  Y          / Store shifted number
107             HLT
108     X,      HEX  1234
        Y,      HEX  4321
                                / Subroutine to shift left 4 times
109     SH4,    HEX  0          / Store return address here
10A             CIL             / Circulate left once
10B             CIL
10C             CIL
10D             CIL             / Circulate left fourth time
10E             AND  MSK        / Set AC(13-16) to zero
10F             BUN  SH4  I     / Return to main program
110     MSK,    HEX  FFF0       / Mask operand
                END
```

SUBROUTINE PARAMETERS AND DATA LINKAGE

Linkage of Parameters and Data between the Main Program and a Subroutine

- **via Registers**
    - **via Memory locations**
- **....**

Example: Subroutine performing *LOGICAL OR operation*; Need two parameters

```
Loc.
              ORG  200
200           LDA  X          / Load 1st operand into AC
201           BSA  OR         / Branch to subroutine OR
202           HEX  3AF6       / 2nd operand stored here
203           STA  Y          / Subroutine returns here
204           HLT
205   X,      HEX  7B95       / 1st operand stored here
206   Y,      HEX  0          / Result stored here
207   OR,     HEX  0          / Subroutine OR
208           CMA             / Complement 1st operand
209           STA  TMP        / Store in temporary location
20A           LDA  OR  I      / Load 2nd operand
20B           CMA             / Complement 2nd operand
20C           AND  TMP        / AND complemented 1st operand
20D           CMA             / Complement again to get OR
20E           ISZ  OR         / Increment return address
20F           BUN  OR  I      / Return to main program
210   TMP,    HEX  0          / Temporary storage
              END
```

SUBROUTINE - Moving a Block of Data -

```
                             / Main program
         BSA  MVE            / Branch to subroutine
         HEX  100            / 1st address of source data
         HEX  200            / 1st address of destination data
         DEC  -16            / Number of items to move
         HLT
MVE,     HEX  0              / Subroutine MVE
         LDA  MVE  I         / Bring address of source
         STA  PT1            / Store in 1st pointer
         ISZ  MVE            / Increment return address
         LDA  MVE  I         / Bring address of destination
         STA  PT2            / Store in 2nd pointer
         ISZ  MVE            / Increment return address
         LDA  MVE  I         / Bring number of items
         STA  CTR            / Store in counter
         ISZ  MVE            / Increment return address
LOP,     LDA  PT1  I         / Load source item
         STA  PT2  I         / Store in destination
         ISZ  PT1            / Increment source pointer
         ISZ  PT2            / Increment destination pointer
         ISZ  CTR            / Increment counter
         BUN  LOP            / Repeat 16 times
         BUN  MVE  I         / Return to main program
PT1,     --
PT2,     --
CTR,     --
```

- **Fortran subroutine**

```
   SUBROUTINE  MVE (SOURCE, DEST, N)
   DIMENSION  SOURCE(N), DEST(N)
   DO  20  I = 1, N
20 DEST(I) = SOURCE(I)
   RETURN
   END
```

INPUT OUTPUT PROGRAM

Program to Input one Character(Byte)

```
CIF,    SKI            / Check input flag
        BUN  CIF       / Flag=0, branch to check again
        INP            / Flag=1, input character
        OUT            / Display to ensure correctness
        STA  CHR       / Store character
        HLT
CHR,    --             / Store character here
```

Program to Output a Character

```
        LDA  CHR       / Load character into AC
COF,    SKO            / Check output flag
        BUN  COF       / Flag=0, branch to check again
        OUT            / Flag=1, output character
        HLT
CHR,    HEX  0057      / Character is "W"
```

CHARACTER MANIPULATION

Subroutine to Input 2 Characters and pack into a word

```
IN2,    --              / Subroutine entry
FST,    SKI
        BUN  FST
        INP             / Input 1st character
        OUT
        BSA  SH4        / Logical Shift left 4 bits
        BSA  SH4        / 4 more bits
SCD,    SKI
        BUN  SCD
        INP             / Input 2nd character
        OUT
        BUN  IN2  I     / Return
```

PROGRAM INTERRUPT

Tasks of Interrupt Service Routine

- Save the Status of CPU
  Contents of processor registers and Flags

- Identify the source of Interrupt
  Check which flag is set

- Service the device whose flag is set
  (Input Output Subroutine)

- Restore contents of processor registers and flags

- Turn the interrupt facility on

- Return to the running program
  Load PC of the interrupted program

INTERRUPT SERVICE ROUTINE

```
Loc.
  0     ZRO,    -               / Return address stored here
  1             BUN  SRV        / Branch to service routine
100             CLA             / Portion of running program
101             ION             / Turn on interrupt facility
102             LDA  X
103             ADD  Y          / Interrupt occurs here
104             STA  Z          / Program returns here after interrupt

                                / Interrupt service routine
200     SRV,    STA  SAC        / Store content of AC
                CIR             / Move E into AC(1)
                STA  SE         / Store content of E
                SKI             / Check input flag
                BUN  NXT        / Flag is off, check next flag
                INP             / Flag is on, input character
                OUT             / Print character
                STA  PT1  I     / Store it in input buffer
                ISZ  PT1        / Increment input pointer
        NXT,    SKO             / Check output flag
                BUN  EXT        / Flag is off, exit
                LDA  PT2  I     / Load character from output buffer
                OUT             / Output character
                ISZ  PT2        / Increment output pointer
        EXT,    LDA  SE         / Restore value of AC(1)
                CIL             / Shift it to E
                LDA  SAC        / Restore content of AC
                ION             / Turn interrupt on
                BUN  ZRO  I     / Return to running program
        SAC,    -               / AC is stored here
        SE,     -               / E is stored here
        PT1,    -               / Pointer of input buffer
        PT2,    -               / Pointer of output buffer
```