http://www.bsccsit.com/

Computer Architecture

CSC. 201

Third Semester

Prepared By: Arjun Singh Saud

Special thanks to Mr. Arjun Singh Saud for providing this valuable note!

Chapter 1
Data representation

Number System

Number of digits used in a number system is called its base or radix. We can categorize number system as below:

- Binary number system
- Octal Number System
- Decimal Number System
- Hexadecimal Number system

Conversion between number systems (do yourself)

Representation of Decimal numbers

We can normally represent decimal numbers in one of following two ways

- By converting into binary
- By using BCD codes

By converting into binary

Advantage

Arithmetic and logical calculation becomes easy. Negative numbers can be represented easily.

Disadvantage

At the time of input conversion from decimal to binary is needed and at the time of output conversion from binary to decimal is needed.

Therefore this approach is useful in the systems where there is much calculation than input/output.

By using BCD codes

Disadvantage

Arithmetic and logical calculation becomes difficult to do. Representation of negative numbers is tricky.

Advantage

At the time of input conversion from decimal to binary and at the time of output conversion from binary to decimal is not needed.

Therefore this approach is useful in the systems where there is much input/output than arithmetic and logical calculation.

Complements

(R-1)'s Complement

(R-1)'s complement of a number N is defined as $(r^n - 1) - N$

Where N is the given number

r is the base of number system

n is the number of digits in the given number

To get the (R-1)'s complement fast, subtract each digit of a number from (R-1)

Example

- 9's complement of 835_{10} is 164_{10}
- 1's complement of 1010_2 is 0101_2(bit by bit complement operation**)**

R's Complement

R's complement of a number N is defined as $r^n - N$

Where N is the given number

r is the base of number system

n is the number of digits in the given number

To get the R's complement fast, add 1 to the low-order digit of its (R-1)'s complement

- 10's complement of 835_{10} is $164_{10} + 1 = 165_{10}$
- 2's complement of 10102 is $0101_2 + 1 = 0110_2$

Representation of Negative numbers

There is only one way of representing positive numbers in computer but we can represent negative numbers in any one of following three ways:

- Signed magnitude representation
- Signed 1's complement representation
- Signed 2's complement representation

Signed magnitude representation

Complement *only* the sign bit

e.g.

+9 ==> 0 001001

-9 ==> 1 001001

Signed 1's complement representation

Complement *all* the bits including sign bit

e.g.

+9 ==> 0 001001

-9 ==> 1 110110

Signed 2's complement representation

Take the 2's complement of the number, *including* its sign bit.

e.g.

+9 ==> 0 001001

-9 ==> 1 110111

Overflow Detection

If we add two n bit numbers, result may be a number with n+1 bit which cannot be stored in n-bit register. This situation is called overflow. We can detect whether there is overflow or not as below:

Case Unsigned numbers

Consider a 4-bit register

Maximum numbers that can be stored $N <= 2^n - 1 = 15$

If there is no end carry => No overflow

e.g.

```
6     0110
9     1001
-----------
15    1111
```

If there is end carry => Overflow.

e.g.

```
9     1001
9     1001
-----------
  (1)0010
```

Overflow

Case Signed Numbers:

Consider a 5-bit register

Maximum and Minimum numbers that can be stored $-2^{n-1} =< N <= +2^{n-1} - 1$

⇨ $-16 =< N <= +15$

To detect the overflow we seed to see two carries. Carry into the sign bit position and carry out of the sign bit position.

If both carries are same => No overflow

```
6    0   0110
9    0   1001
---------------
15   0   1111
```

Here carry in sign bit position $= c_{n-1} = 0$

carry out of sign bit position $= c_n = 0$

$(c_{n-1} \oplus c_n) = 0$ => No overflow

If both carries are different => overflow

```
 9     0 1001
+9     0 1001
-------------
18     1 0010
```

Here carry in sign bit position $= c_{n-1} = 1$

Carry out of sign bit position $= c_n = 0$

$(c_{n-1} \oplus c_n) = 1$ => overflow

Prepared By: Arjun Singh Saud

Floating Point Representation

Floating points are represented in computers as the format given below:

Sign	Exponent	mantissa

Mantissa

Signed fixed point number, either an integer or a fractional number

Exponent

Designates the position of the decimal point

Decimal Value

$$N = m * r^{e}$$

Where m is mantissa
r is base
e is exponent

Example
Consider the number N= 1324.567

Now
m = 0.1324567
e = 4
r = 10

therefore
$N= m * r^{e} = 0.1324567 * 10^{+4}$

Note:
In Floating Point Number representation, only Mantissa (m) and Exponent (e) are explicitly represented. The position of the Radix Point is implied.

Another example
Consider the binary number N=1001.11 (6-bit exponent and 10-bit fractional mantissa)

Now
m= 100111000
e= 0 00100 = +4
r= 2
sign bit = 0

Normalizing Floating point numbers

A number is said to be normalized if most significant position of the mantissa contains a non-zero digit.

e.g.

m= 001001110

e= 0 00100 = +6

r= 2

Above number is not normalized

To normalize the above number we need to remove the leading zeros of mantissa and need to subtract the exponent from the number of zeros that are removed.

i.e.

m= 1001110

e= 0 00100 = +4

Normalization improves the precision of floating point numbers.

ERROR DETECTING CODES

A parity bit(s) is an extra bit that is added with original message to detect error in the message during data transmission. This is a simplest method for error detection**.**

Even Parity

One bit is attached to the information so that the total number of 1 bits is an even number

Message	Parity
1011001	0
1010010	1

Odd Parity

One bit is attached to the information so that the total number of 1 bits is an odd number

Message	Parity
1011001	1
1010010	0

Parity generator

Message (xyz)	Parity bit (odd)
000	1
001	0
010	0
011	1
100	0
101	1
110	1
111	0

Now
$P = \overline{x \oplus y \oplus z}$

Parity Checker:

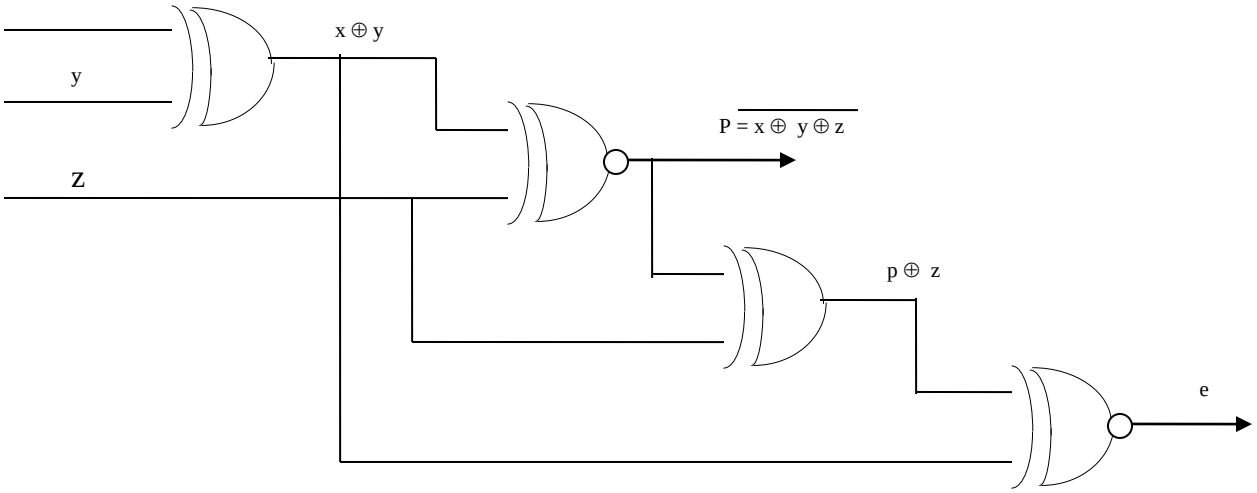
Considers original message as well as parity bit

$e = p \oplus x \oplus y \oplus z$

e= 1 => No. of 1's in pxyz is even => Error in data

e= 0 => No. of 1's in pxyz is odd => Data is error free

Circuit diagram for parity generator and parity checker



Other Decimal Codes

Decimal	BCD(8421)	2421	84-2-1	Excess-3
0	0000	0000	0000	0011
1	0001	0001	0111	0100
2	0010	0010	0110	0101
3	0011	0011	0101	0110
4	0100	0100	0100	0111
5	0101	1011	1011	1000
6	0110	1100	1010	1001
7	0111	1101	1001	1010
8	1000	1110	1000	1011
9	1001	1111	1111	1100

Let d3 d2 d1 d0: symbol in the codes

BCD: d3 x 8 + d2 x 4 + d1 x 2 + d0 x 1 $\Rightarrow$ 8421 code.

2421: d3 x 2 + d2 x 4 + d1 x 2 + d0 x 1

Excess-3: BCD + 3

BCD: It is difficult to obtain the 9's complement.
However, it is easily obtained with the other codes listed above → Self-complementing codes

Gray Codes

Characterized by having their representations of the binary integers differ in only one digit between consecutive integers

Decimal number	Gray				Binary			
	g_3	g_2	g_1	g_0	b_3	b_2	b_1	b_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	1	0	0	1	0
3	0	0	1	0	0	0	1	1
4	0	1	1	0	0	1	0	0
5	0	1	1	1	0	1	0	1
6	0	1	0	1	0	1	1	0
7	0	1	0	0	0	1	1	1
8	1	1	0	0	1	0	0	0
9	1	1	0	1	1	0	0	1
10	1	1	1	1	1	0	1	0
11	1	1	1	0	1	0	1	1
12	1	0	1	0	1	1	0	0
13	1	0	1	1	1	1	0	1
14	1	0	0	1	1	1	1	0
15	1	0	0	0	1	1	1	1

See ASCII code yourself

Chapter 2
Register Transfer and microoperations

Combinational and sequential circuits can be used to create simple digital systems. These are the low-level building blocks of a digital computer. The operations on the data in registers are called microoperations. The functions built into registers are examples of microoperations

- Shift
- Load
- Clear
- Increment

Alternatively we can say that an elementary operation performed during one clock pulse on the information stored in one or more registers is called microopeartion. Register transfer language can be used to describe the (sequence of) microoperations

Register Transfer Language

Registers are designated by capital letters, sometimes followed by numbers (e.g., A, R13, IR). Often the names indicate function:

- MAR memory address register
- PC program counter
- IR instruction register

A register transfer is indicated as "R2 ← R1"

Control Function

Often actions need to only occur if a certain condition is true. In digital systems, this is often done via a control signal, called a control function.
e.g.
P: R2 ← R1

Which means "if P = 1, then load the contents of register R1 into register R2", i.e., if (P = 1 then (R2 ← R1))

If two or more operations are to occur simultaneously, they are separated with commas
e.g.
P: R3 ← R5, MAR ← IR

Microoperations

Computer system microoperations are of four types:

- Register transfer microoperations
- Arithmetic microoperations
- Logic microoperations
- Shift microoperations

Arithmetic microoperations

- The basic arithmetic microoperations are
  - Addition
  - Subtraction
  - Increment
  - Decrement

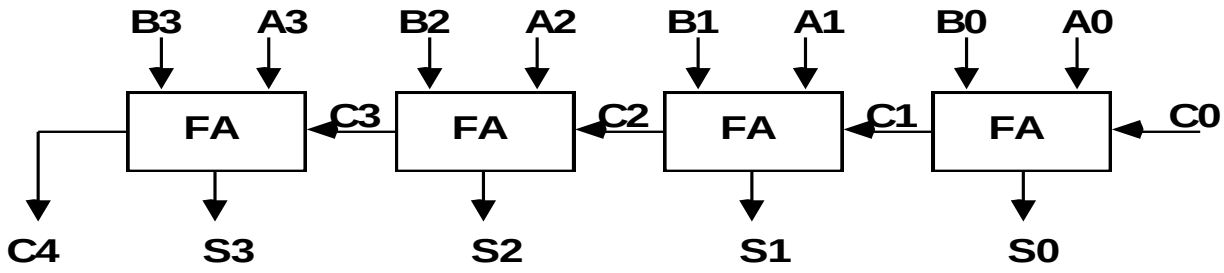
- The additional arithmetic microoperations are
  - Add with carry
  - Subtract with borrow
  - Transfer/Load
  - etc. …

Summary of Typical Arithmetic Micro-Operations

R3 ← R1 + R2	Contents of R1 plus R2 transferred to R3
R3 ← R1 - R2	Contents of R1 minus R2 transferred to R3
R2 ← R2'	Complement the contents of R2
R2 ← R2'+ 1	2's complement the contents of R2 (negate)
R3 ← R1 + R2'+ 1	subtraction
R1 ← R1 + 1	Increment
R1 ← R1 - 1	Decrement

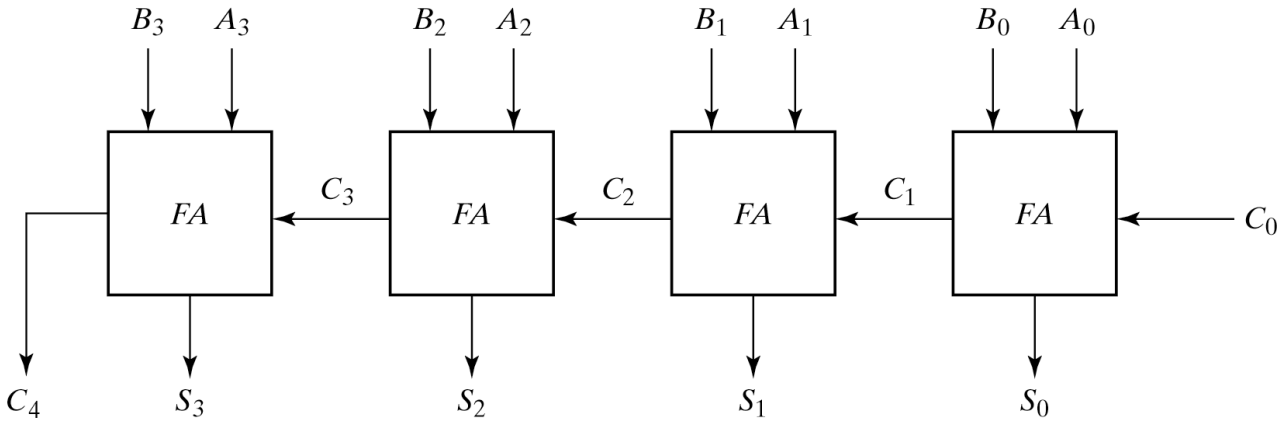
Binary Adder

To perform multibit addition in computer a full adder must be allocated for each bit so that all bits can be added simultaneously. Thus, to add two 4-bit numbers to produce a 4-bit sum (with a possible carry), we need four full adders with carry lines cascaded, as shown in the figure given below. For two 8-bit numbers, we need eight full adders, which can be formed by cascading two of these 4-bit blocks. By extension, two binary numbers of any size may be added in this manner.



Binary Subtractor

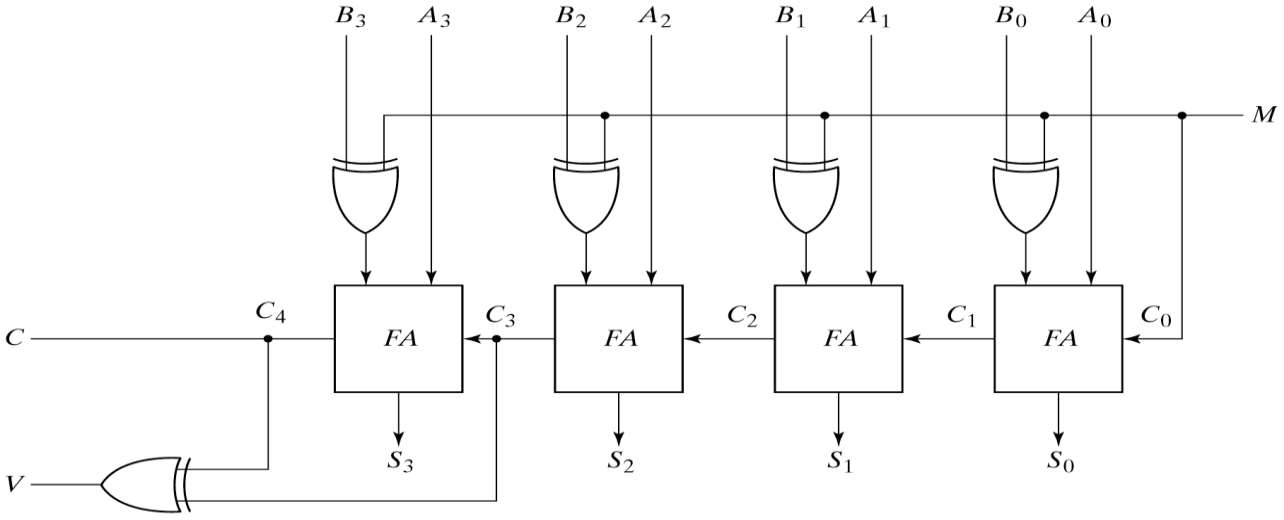
The subtraction A – B can be done by taking the 2's complement of B and adding it to A because A- B = A + (-B). It means if we use the inverters to make 1's complement of B (connecting each Bi to an inverter) and then add 1 to the least significant bit (by setting carry C0 to 1) of binary adder, then we can make a binary subtractor.



Binary Adder Subtractor

The addition and subtraction can be combined into one circuit with one common binary adder. The mode M controls the operation. When M=0 the circuit is an adder when M=1 the circuit is Subtractor. It can be done by using exclusive-OR for each Bi and M. Note that $1 \oplus x = x'$ and $0 \oplus x = x$

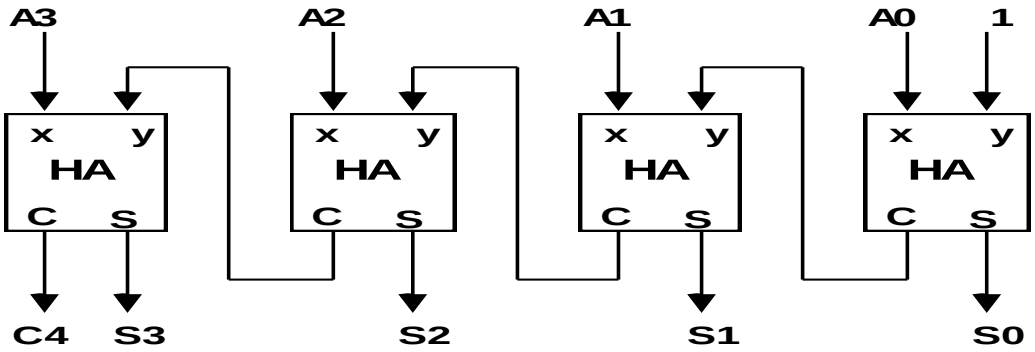
{C = carry bit, V= overflow bit}



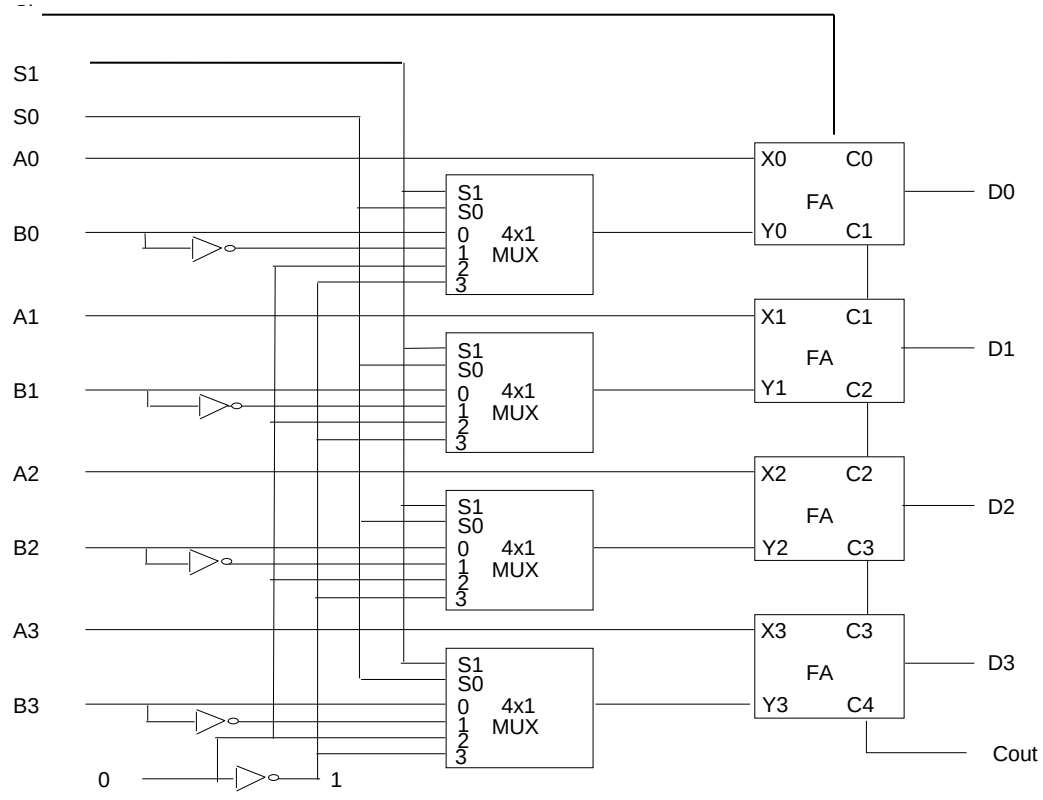
Binary Incrementer

The increment microoperation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 011 1 after it is incremented. This can be accomplished by means of half-adders connected in cascade.

A = A +1



Binary Arithmetic Circuit



S1	S0	Cin	Y	Output	Microoperation
0	0	0	B	D = A + B	Add
0	0	1	B	D = A + B + 1	Add with carry
0	1	0	B’	D = A + B’	Subtract with borrow
0	1	1	B’	D = A + 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

If s_0 and s_1 both are zero mux selects the input a lebel 0 (i.e. Y= B) Then adder adds A and Y and cin (i.e. A and B and cin)

- ⇨ D = A + B if cin=0
- ⇨ D = A + B+1 if cin=1

If s_0 =0 and s_1=1 mux selects the input at label 1 (i.e. Y= B’) Then adder adds A and Y (i.e. A and B’ and cin)

- ⇨ D = A + B’ if cin=0
- ⇨ D = A + B’ +1 if cin=1

If s_0 =1 and s_1=0 mux selects the input at label 2 (i.e. Y= 0000) Then adder adds A and Y (i.e. A and 0 and cin)

- ⇨ D = A if cin=0
- ⇨ D = A +1 if cin=1

If s_0 =1 and s_1=1mux selects the input at label 3 (i.e. Y= 1111= -1) Then adder adds A and Y (i.e. A and -1 and cin)

- ⇨ D = A - 1 if cin=0
- ⇨ D = A if cin=1

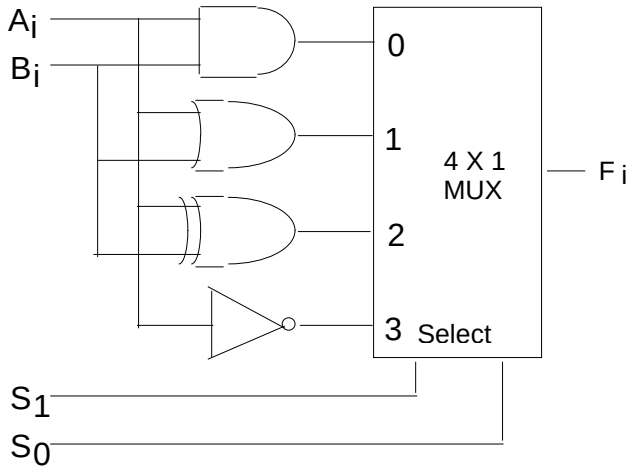
Logic Microoperations

Logic microoperations are bit-wise operations, i.e., they work on the individual bits of data. It is useful for bit manipulations on binary data. Useful for making logical decisions based on the bit value. There are, in principle, 16 different logic functions that can be defined over two binary input variables. However, most systems only implement four of these

- AND ($\wedge$), OR ($\vee$), XOR ($\oplus$), Complement/NOT

The others can be created from combination of these

Hardware Implementation of Logic microoperations



S_1 S_0	Output	μ-operation
0 0	$F = A \wedge B$	AND
0 1	$F = A \vee B$	OR
1 0	$F = A \oplus B$	XOR

Applications Of Logic Microoperations

Logic microoperations can be used to manipulate individual bits or a portions of a word in a register. Consider the data in a register A. In another register, B, is bit data that will be used to modify the contents of A

- Selective-set $A \leftarrow A + B$
- Selective-complement $A \leftarrow A \oplus B$
- Selective-clear $A \leftarrow A \cdot B'$
- Mask (Delete) $A \leftarrow A \cdot B$
- Clear $A \leftarrow A \oplus B$
- Insert $A \leftarrow (A \cdot B) + C$
- Compare $A \leftarrow A \oplus B$

Selective-set

In a selective set operation, the bit pattern in B is used to *set* certain bits in A

$$\begin{array}{ll} 1\ 1\ 0\ 0\ A_t & \\ 1\ 0\ 1\ 0\ B & \\ 1\ 1\ 1\ 0\ A_{t+1} & (A \leftarrow A + B) \end{array}$$

If a bit in B is set to 1, that same position in A gets set to 1, otherwise that bit in A keeps its previous value

Selective-complement

In a selective complement operation, the bit pattern in B is used to *complement* certain bits in A

$$\begin{array}{ll} 1\ 1\ 0\ 0\ A_t & \\ 1\ 0\ 1\ 0\ B & \\ 0\ 1\ 1\ 0\ A_{t+1} & (A \leftarrow A \oplus B) \end{array}$$

If a bit in B is set to 1, that same position in A gets complemented from its original value, otherwise it is unchanged

Selective-clear

n a selective clear operation, the bit pattern in B is used to *clear* certain bits in A

$$\begin{array}{ll} 1\ 1\ 0\ 0\ A_t & \\ 1\ 0\ 1\ 0\ B & \\ 0\ 1\ 0\ 0\ A_{t+1} & (A \leftarrow A \cdot B') \end{array}$$

If a bit in B is set to 1, that same position in A gets set to 0, otherwise it is unchanged

Mask Operation

In a mask operation, the bit pattern in B is used to *clear* certain bits in A

$$\begin{array}{ll} 1\ 1\ 0\ 0\ A_t & \\ 1\ 0\ 1\ 0\ B & \\ 1\ 0\ 0\ 0\ A_{t+1} & (A \leftarrow A \cdot B) \end{array}$$

If a bit in B is set to 0, that same position in A gets set to 0, otherwise it is unchanged

Clear Operation

In a clear operation, if the bits in the same position in A and B are the same, they are cleared in A, otherwise they are set in A

$$\begin{array}{ll} 1\ 1\ 0\ 0\ A_t & \\ 1\ 0\ 1\ 0\ B & \\ 0\ 1\ 1\ 0\ A_{t+1} & (A \leftarrow A \oplus B) \end{array}$$

Insert Operation

An insert operation is used to introduce a specific bit pattern into A register, leaving the other bit positions unchanged.

This is done as

- A mask operation to clear the desired bit positions, followed by
- An OR operation to introduce the new bits into the desired positions
- Example
  - » Suppose you wanted to introduce 1010 into the low order four bits of A: 1101 1000 1011 0001 A (Original)
    1101 1000 1011 1010 A (Desired)

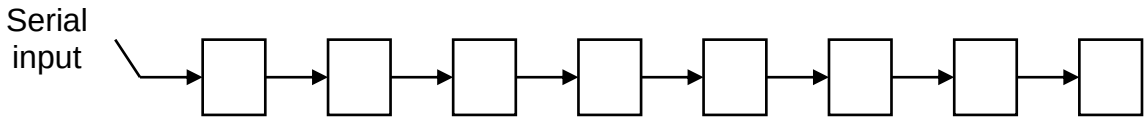
1101 1000 1011 0001	A (Original)
1111 1111 1111 0000	Mask
1101 1000 1011 0000	A (Intermediate)
0000 0000 0000 1010	Added bits
1101 1000 1011 1010	A (Desired)

Shift Micro-Operations

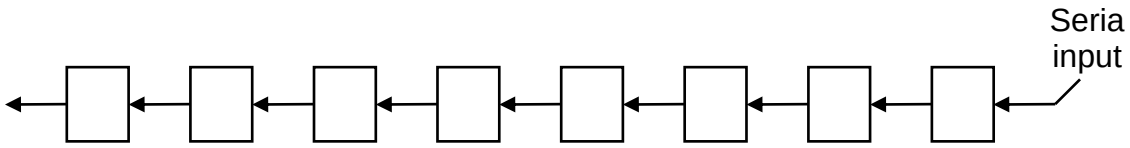
There are three types of shifts

- *Logical shift*
- *Circular shift*
- *Arithmetic shift*

Right shift operation

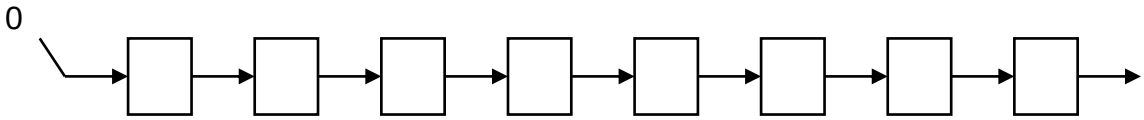


Left shift operation

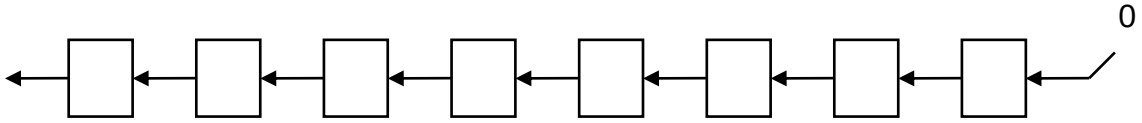


In a logical shift the serial input to the shift is a 0.

Logical right shift operation:



Logical left shift operation



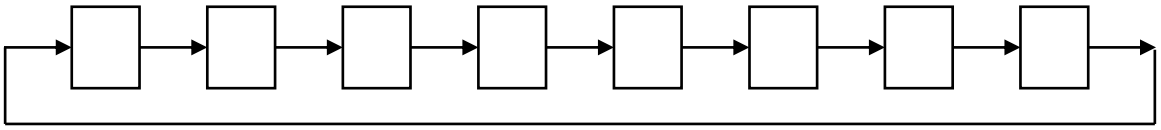
In a Register Transfer Language, the following notation is used

- *shl* for a logical shift left
- *shr* for a logical shift right
- Examples:
  - » R2 ← *shr* R2
  - » R3 ← *shl* R3

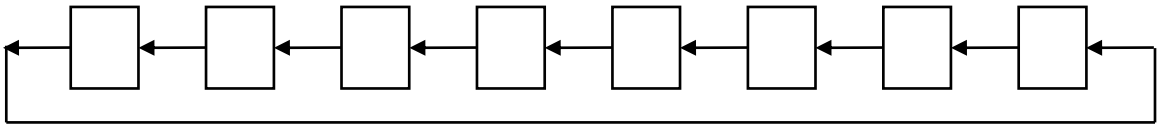
Circular Shift operation

In a circular shift the serial input is the bit that is shifted out of the other end of the register.

Right circular shift operation



Left circular shift operation:



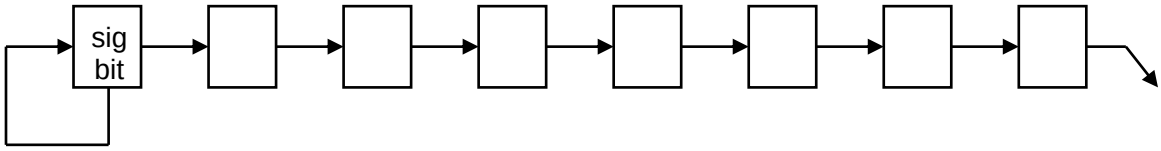
In a RTL, the following notation is used

- *cil* for a circular shift left
- *cir* for a circular shift right
- Examples:
  - » R2 ← *cir* R2
  - » R3 ← *cil* R3
  - »

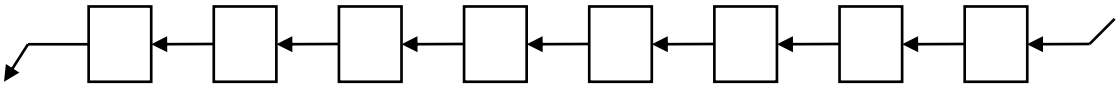
Arithmetic Shift Operation

An arithmetic shift is meant for signed binary numbers (integer). An arithmetic left shift multiplies a signed number by two and an arithmetic right shift divides a signed number by two. The main distinction of an arithmetic shift is that it must keep the sign of the number the same as it performs the multiplication or division

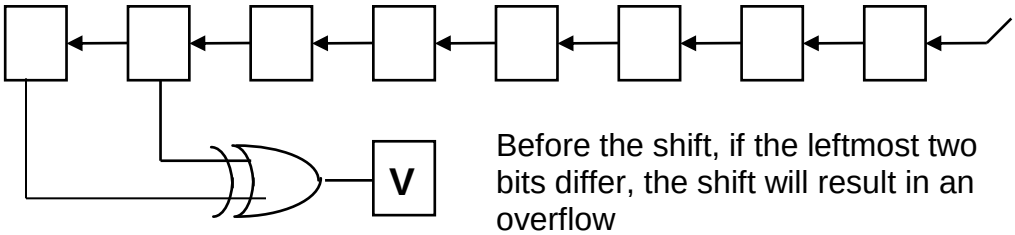
Right arithmetic shift operation:



Left arithmetic shift operation



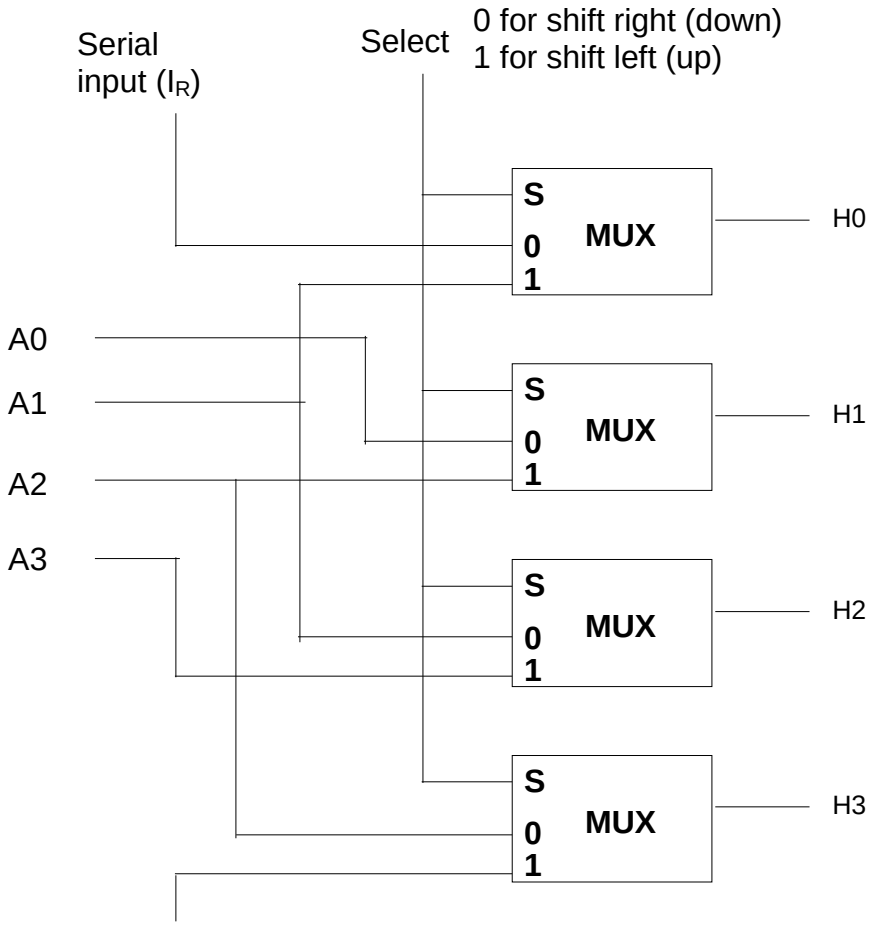
An left arithmetic shift operation must be checked for the overflow



In a RTL, the following notation is used

- *ashl* for an arithmetic shift left
- *ashr* for an arithmetic shift right
- Examples:
  - » R2 ← *ashr* R2
  - » R3 ← *ashl* R3

Hardware Implementation of Shift Microoperations



Chapter 3
Basic Computer Organization and Design

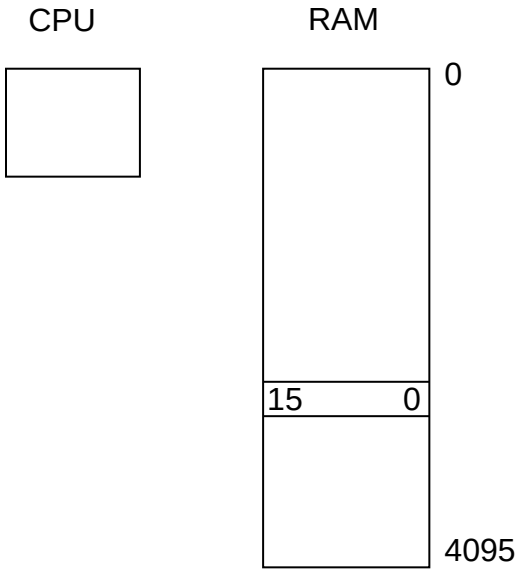
Introduction

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. M. Morris Mano introduces a simple processor model he calls 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



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

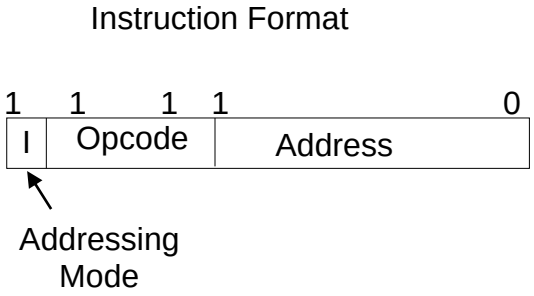
Instruction Format of Basic Computer

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 the memory address that is used by this instruction. 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

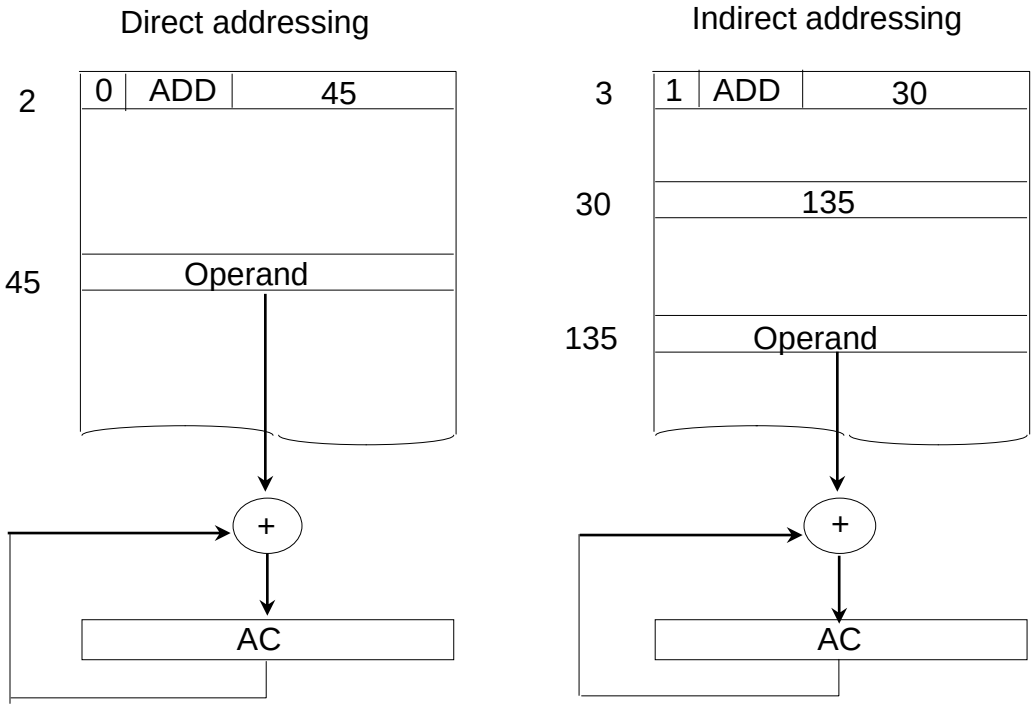


Addressing Modes

The address field of an instruction can represent either

– Direct address: the address operand field is effective address (the address of the operand), or

– Indirect address: the address in operand field contains the memory address where effective address resides.



- **Effective Address (EA)**
  - The address, where actual data resides is called effective adrress.

Basic Computer Registers

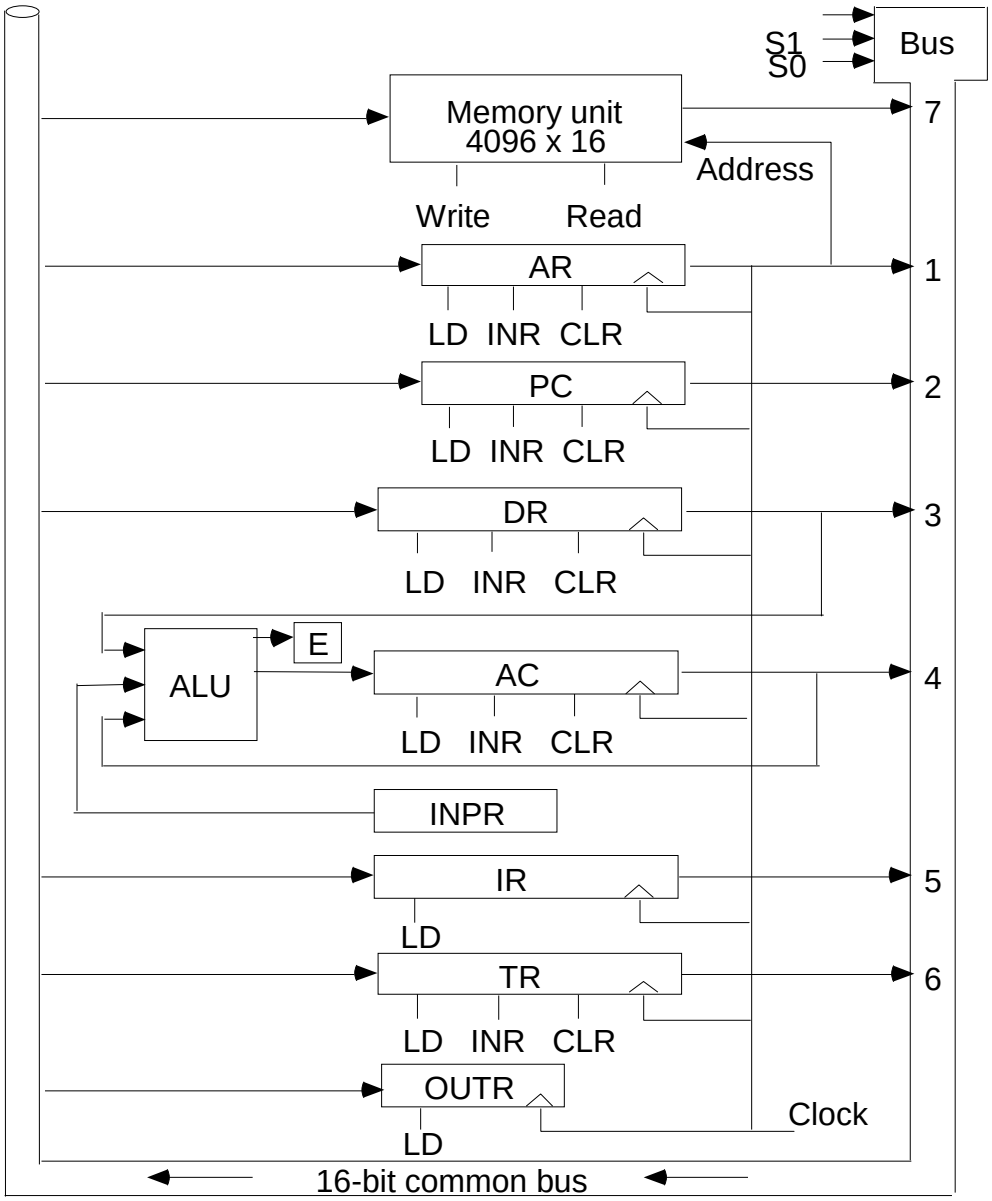
Symbol	Size	Register Name	Description
DR	16	Data Register	Holds memory operand
AR	12	Address Register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction Register	Holds instruction code
PC	12	Program Counter	Holds address of instruction
TR	16	Temporary Register	Holds temporary data
INPR	8	Input Register	Holds input character
OUTR	8	Output Register	Holds output character

Since the memory in the Basic Computer only has 4096 (=2^{12}) locations, PC and AR only needs 12 bits
Since the word size of Basic Computer only has 16 bit, the DR, AC, IR and TR needs 16 bits.
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

Common Bus System of Basic Computer
The registers in the Basic Computer are connected using a bus. This gives a savings in circuitry over complete connections between registers.
Three control lines, S2, S1, and S0 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



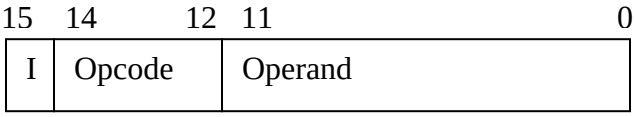
Either one of the registers will have its load signal activated, or the memory will have its read signal activated

- Will determine where the data from the bus gets loaded

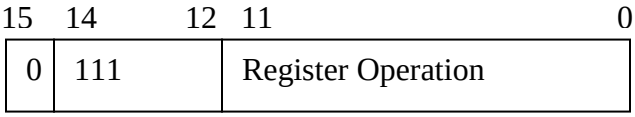
The 12-bit registers, AR and PC, have 0's loaded onto the bus in the high order 4 bit positions. When the 8-bit register OUTR is loaded from the bus, the data comes from the low order 8 bits on the bus

Instruction Formats of Basic Computer

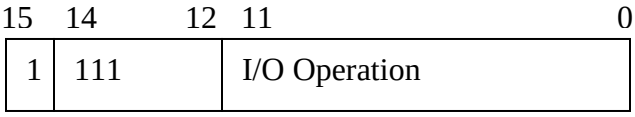
Memory-Reference Instructions (OP-code = 000 ~ 110)



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



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



Instruction Set Completeness

An instruction set is said to be complete if it contains sufficient instructions to perform operations in following categories:

- ✓ Arithmetic, logic, and shift instructions
- ✓ Instructions to transfer data between the main memory and the processor registers
- ✓ Program control and sequencing instructions
- ✓ Instructions to perform Input/Output operations

Instruction set of Basic computer is complete because

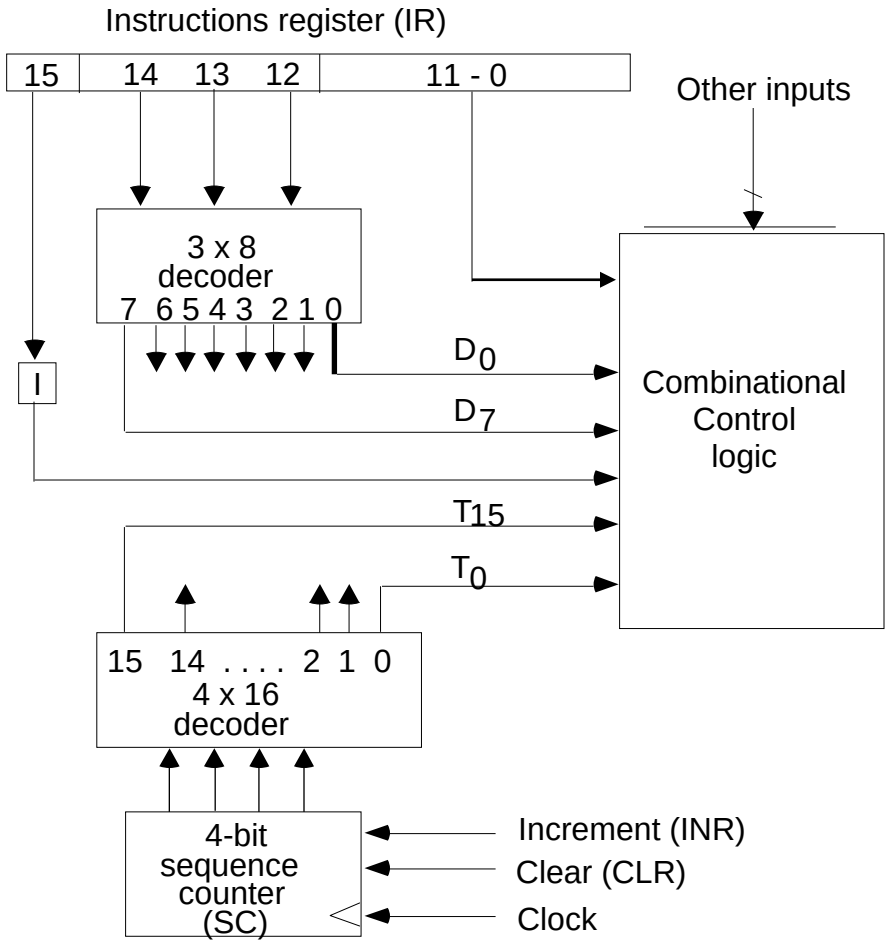
ADD, CMA (complement), INC can be used to perform addition and subtraction and CIR (circular right shift), CIL (circular left shift) instructions can be used to achieve any kind of shift operations. Addition subtraction and shifting can be used together to achieve multiplication and division. AND, CMA and CLA (clear accumulator) can be used to achieve any logical operations.
LDA instruction moves data from memory to register and STA instruction moves data from register to memory.
The branch instructions BUN, BSA and ISZ together with skip instruction provide the mechanism of program control and sequencing.
INP instruction is used to read data from input device and OUT instruction is used to send data from processor to output device.

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

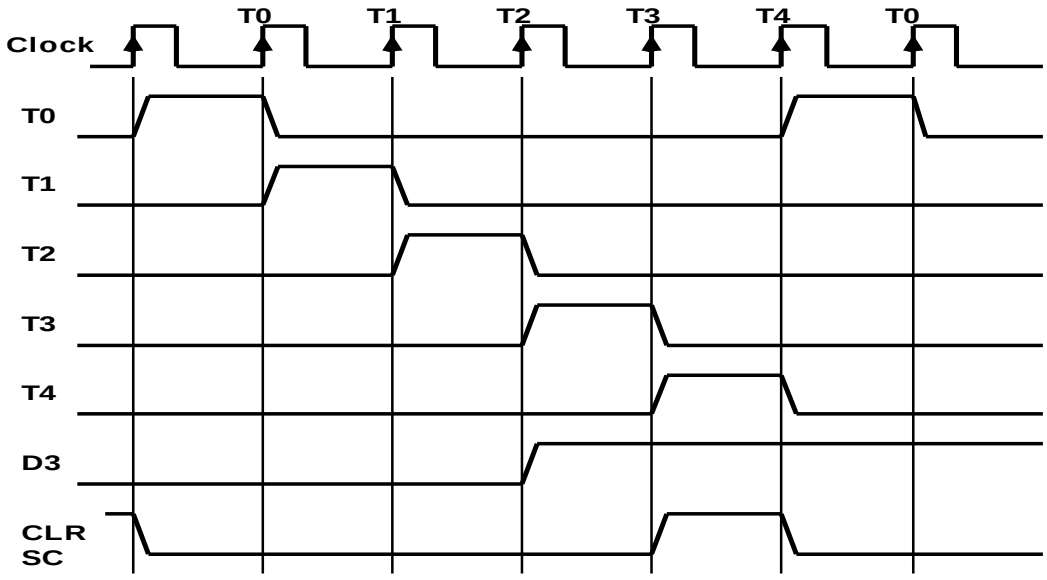
- *Hardwired* Control
  - CU is made up of sequential and combinational circuits to generate the control signals
  - If logic is changed we need to change the whole circuitry
  - Expensive
  - Fast
- *Microprogrammed* Control
  - A control memory on the processor contains microprograms that activate the necessary control signals
  - If logic is changed we only need to change the microprogram
  - Cheap
  - Slow

Hardwired control unit of Basic Computer



Operation code is decoded by 3 x 8 decoder which is used to identify the operation. A 4-bit sequence counter is used to generate timing signals from T_0 to T_{15}. This means instruction cycle of basic computer cannot take more than 16 clock cycles.

Assume: At time T4, SC is cleared to 0 if decoder output D3 is active.
D3T4: SC ← 0
We can show timing control diagram as below:



Instruction Cycle of Basic Computer

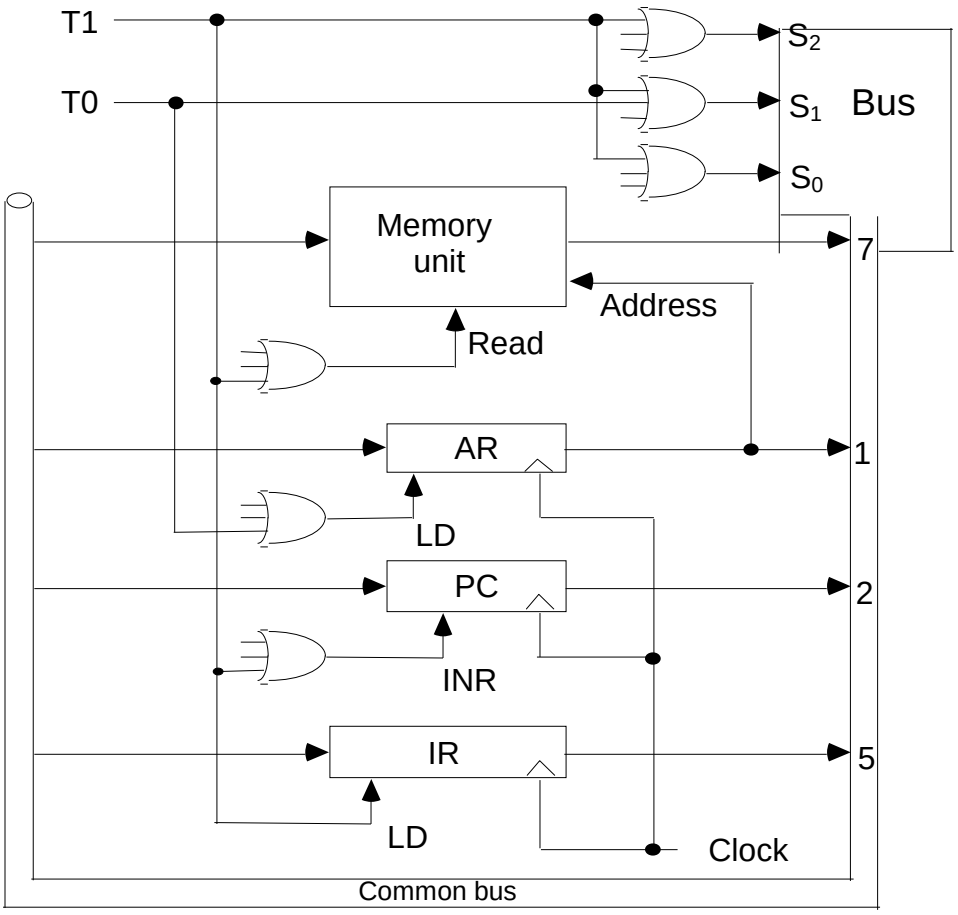
In Basic Computer, a machine instruction is executed in the following cycle:

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

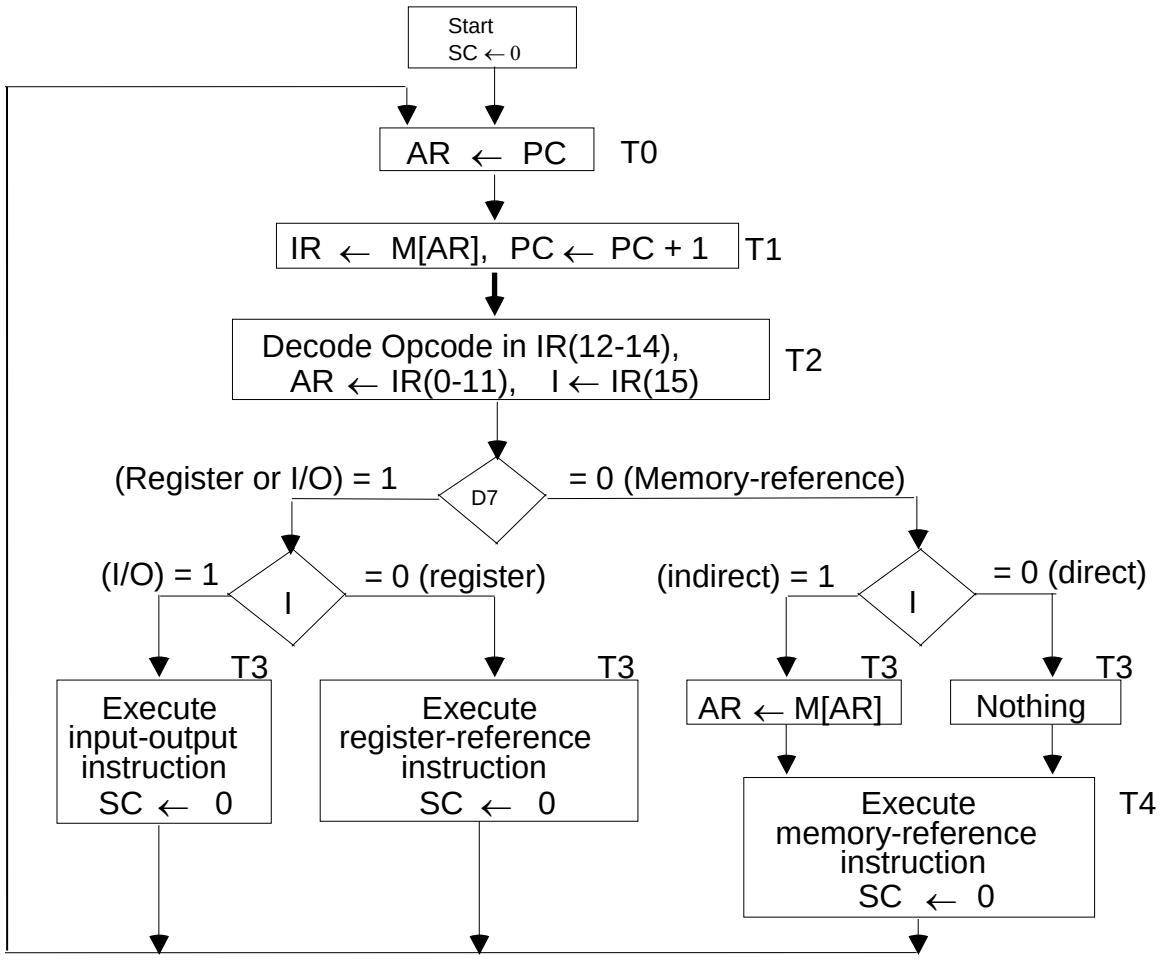
After an instruction is executed, the cycle starts again at step 1, for the next instruction

Fetch and Decode

T0: AR ←PC (S0S1S2=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)


T1
T0
S_2
S_1
S_0
Bus
Memory unit
Address
Read
AR
LD
PC
INR
IR
LD
Clock
7
1
2
5
Common bus

Flowchart for determining 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 are identified when
- $D_7 = 1$, $I = 0$
- Register Ref. Instr. is specified in b0 ~ b11 of IR
- Execution starts with timing signal T3

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

CLA	rB11:	AC ← 0, SC ← 0
CLE	rB10:	E ← 0, SC ← 0
CMA	rB9:	AC ← AC', SC ← 0
CME	rB8:	E ← E', SC ← 0
CIR	rB7:	AC ← shr AC, AC(15) ← E, E ← AC(0), SC ← 0
CIL	rB6:	AC ← shl AC, AC(0) ← E, E ← AC(15)
INC	rB5:	AC ← AC + 1, SC ← 0
SPA	rB4:	if (AC(15) = 0) then (PC ← PC+1), SC ← 0
SNA	rB3:	if (AC(15) = 1) then (PC ← PC+1), SC ← 0
SZA	rB2:	if (AC = 0) then (PC ← PC+1), SC ← 0
SZE	rB1:	if (E = 0) then (PC ← PC+1), SC ← 0
HLT	rB0:	S ← 0, SC ← 0 (S is a start-stop flip-flop)

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

- Memory cycle is assumed to be short enough to complete in a CPU cycle
- The execution of memory reference instruction starts with T4

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

AND to AC

D_0T_4: DR ← M[AR] //Read operand

D_0T_5: AC ← AC ∧ DR, SC ← 0 //AND with AC

ADD to AC

D_1T_4: DR ← M[AR] //Read operand

D_1T_5: AC ← AC + DR, E ← C_{out}, SC ← 0 //Add to AC and stores carry in E

LDA: Load to AC

D_2T_4: DR ← M[AR] //Read operand

D_2T_5: AC ← DR, SC ← 0 //Load AC with DR

STA: Store AC

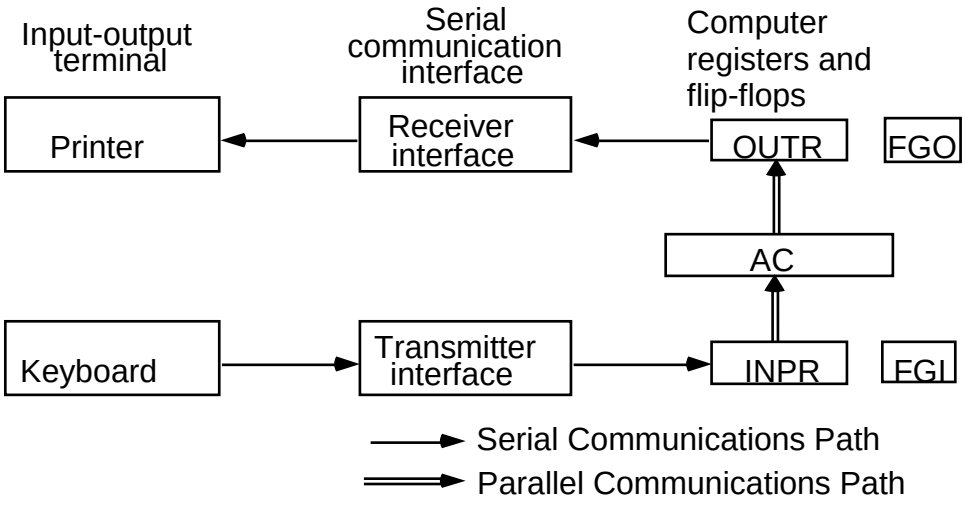
D_3T_4: M[AR] ← AC, SC ← 0 // store data into memory location

BUN: Branch Unconditionally
D_4T_4: PC ← AR, SC ← 0 //Branch to specified address
BSA: Branch and Save Return Address

D_5T_4: M[AR] ← PC, AR ← AR + 1 // save return address and increment AR
D_5T_5: PC ← AR, SC ← 0 // load PC with AR

ISZ: Increment and Skip-if-Zero
D_6T_4: DR ← M[AR] //Load data into DR
D_6T_5: DR ← DR + 1 // Increment the data
D_6T_4: M[AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ← 0
// if DR=0 skip next instruction by incrementing PC

Input-Output Configuration and Interrupt



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

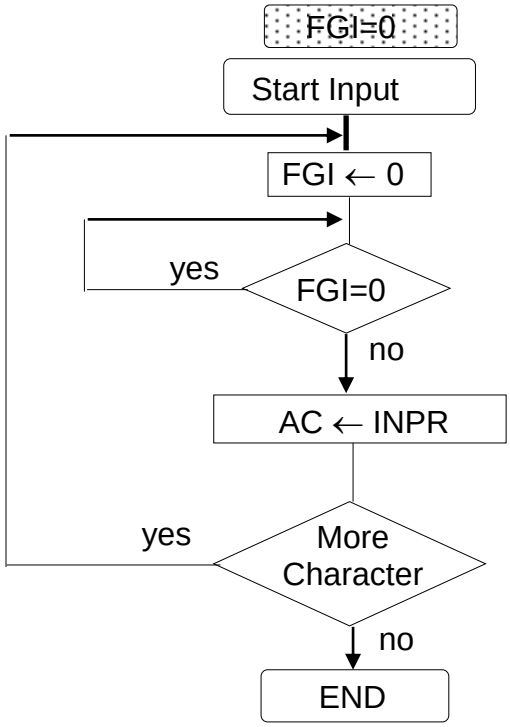
The terminal sends and receives serial information
- The serial info. from the keyboard is shifted into INPR
- The serial info. for the printer is stored in the OUTR
- INPR and OUTR communicate with the terminal serially and with the AC in parallel.

- The flags are needed to *synchronize* the timing difference between I/O device and the computer

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

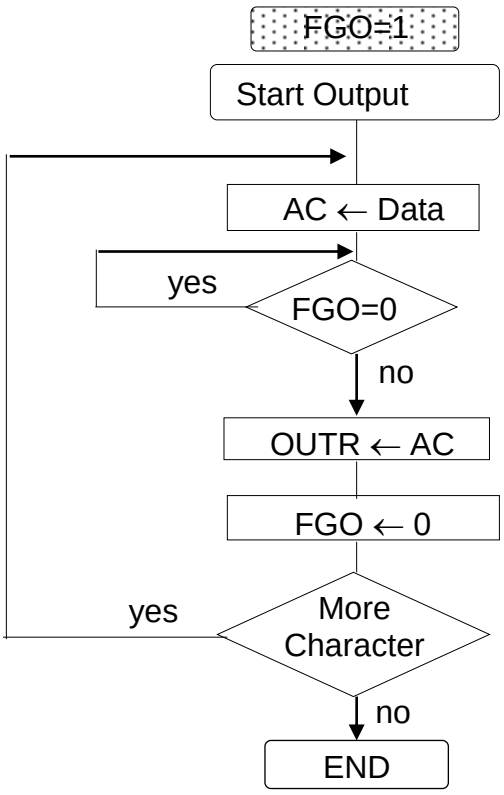
Input Device:
loop: If FGI = 1 goto loop
INPR ← new data, FGI ← 1

Flowchart of CPU operation



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

Flowchart of CPU operation (output)

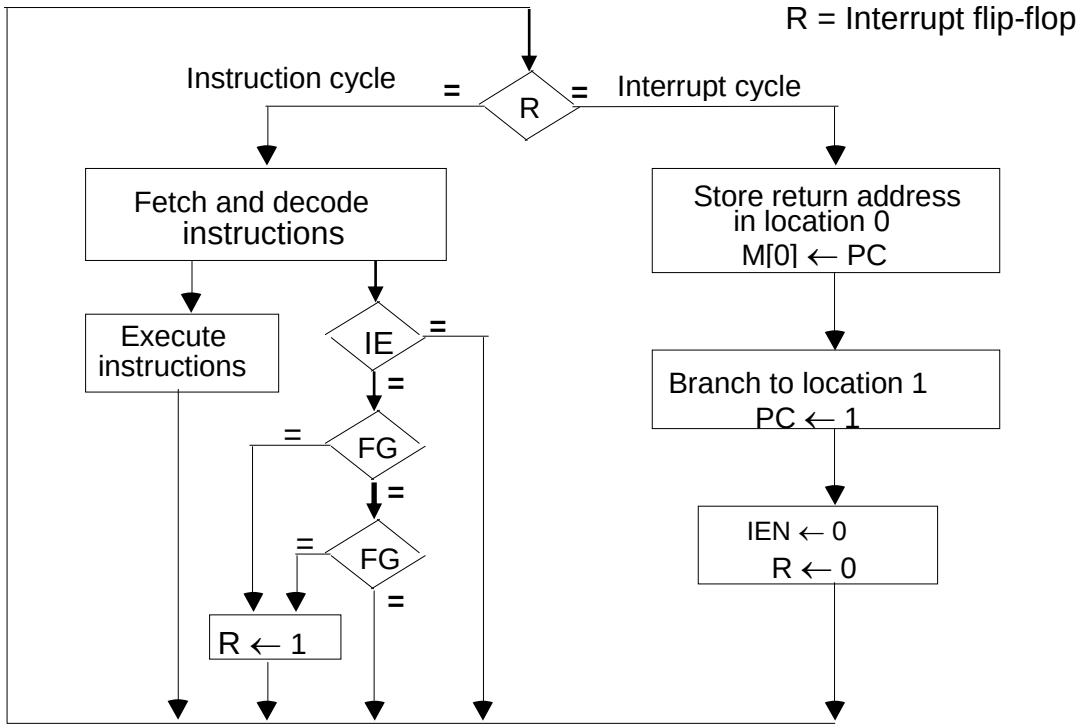


Input Output Instructions

Let
$D_7IT_3 = p$
IR(i) = Bi, i = 6, …, 11

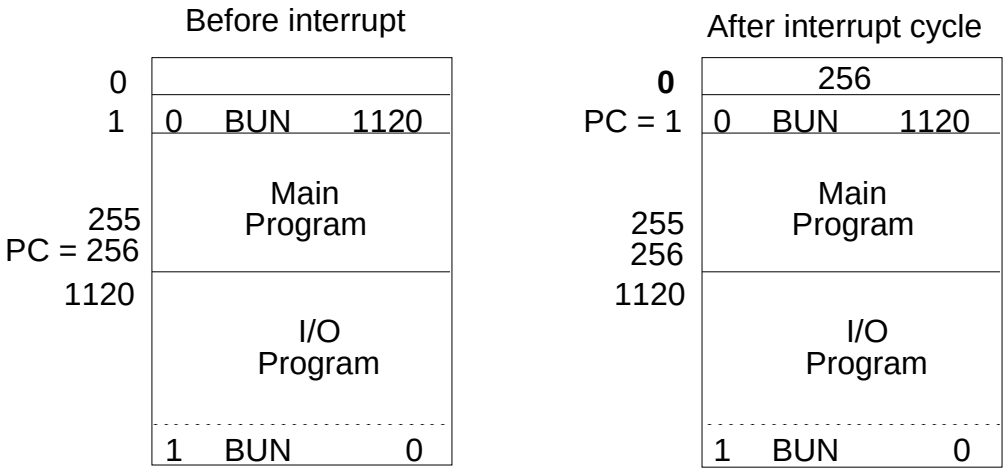
INP	pB_{11}:	AC(0-7) ← INPR, FGI ← 0, SC ← 0	Input char. to AC
OUT	pB_{10}:	OUTR ← AC(0-7), FGO ← 0, SC ← 0	Output char. from AC
SKI	pB_9:	if(FGI = 1) then (PC ← PC + 1), SC ← 0	Skip on input flag
SKO	pB_8:	if(FGO = 1) then (PC ← PC + 1), SC ← 0	Skip on output flag
ION	pB_7:	IEN ← 1, SC ← 0	Interrupt enable on
IOF	pB_6:	IEN ← 0, SC ← 0	Interrupt enable off

Interrupt Cycle

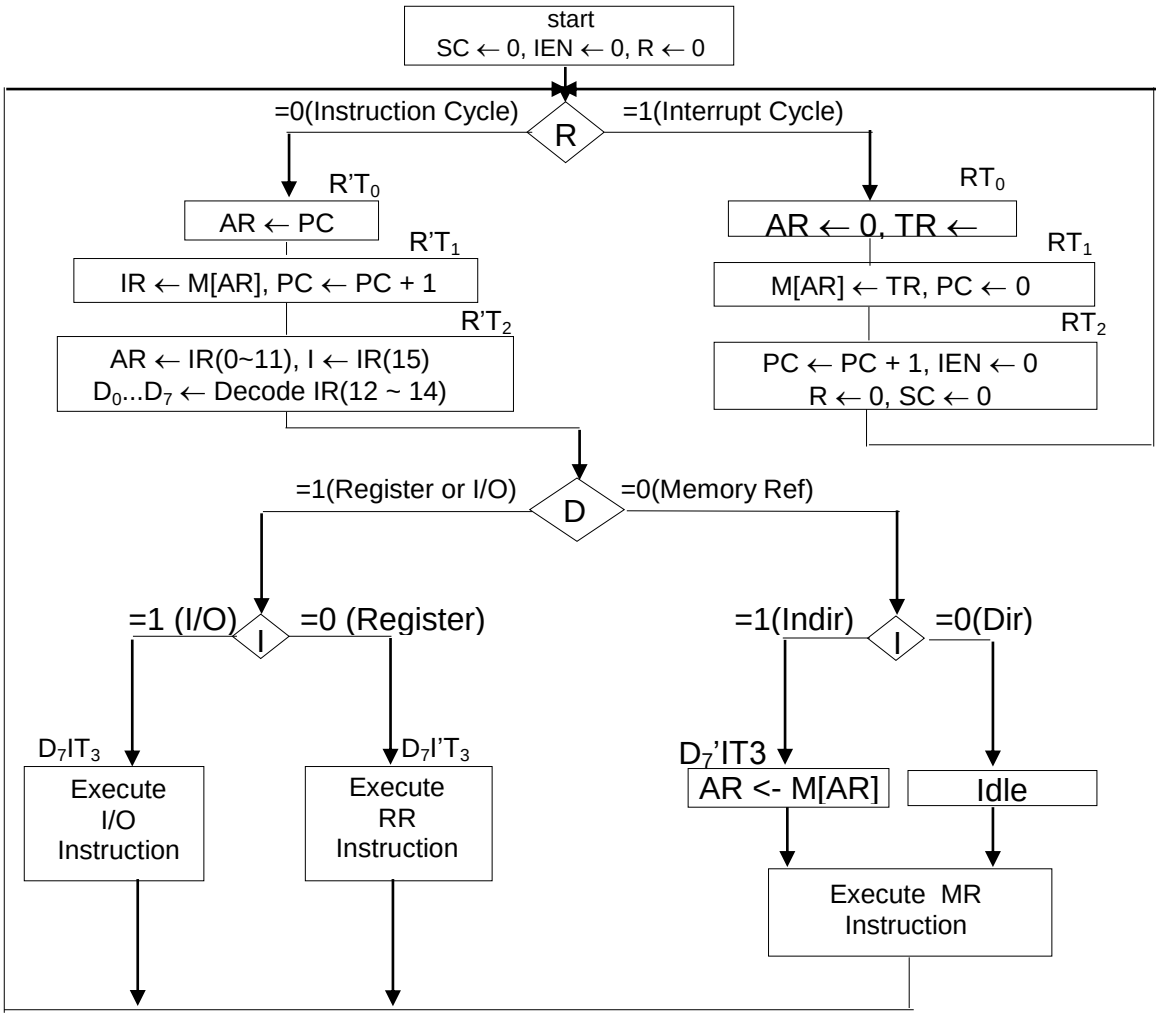


The interrupt cycle is a HW implementation of a branch and save return address operation.

- At the beginning of the 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"



Complete description of Basic Computer



Design of Basic Computer

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
- S2, S1, S0 Controls to select a register for the bus
- AC, and Adder and Logic circuit

Control of AR register

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

$R'T_0$: AR ← PC LD(AR)

$R'T_2$: AR ← IR(0-11) LD(AR)

D'_7IT_3: AR ← M[AR] LD(AR)

RT_0: AR ← 0 CLR(AR)

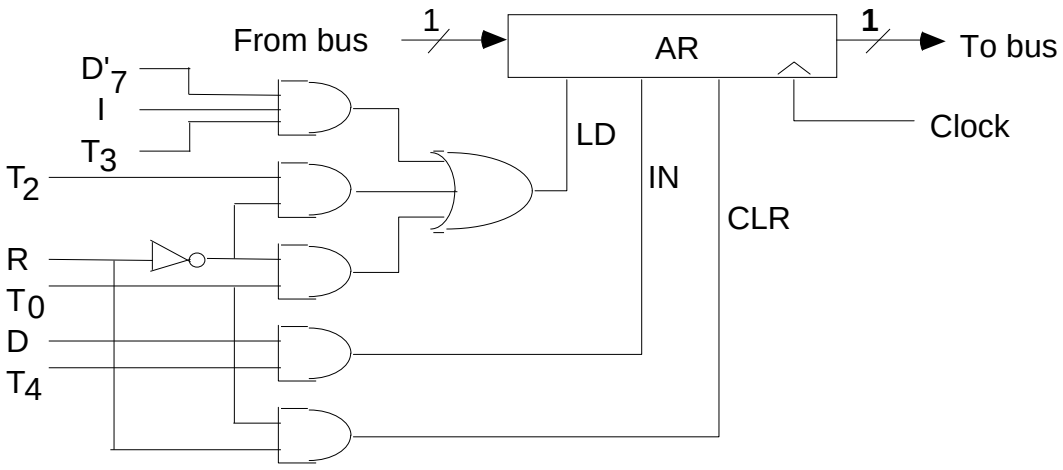
D_5T_4: AR ← AR + 1 INR(AR)

Now,

$LD(AR) = R'T_0 + R'T_2 + D'_7IT_3$

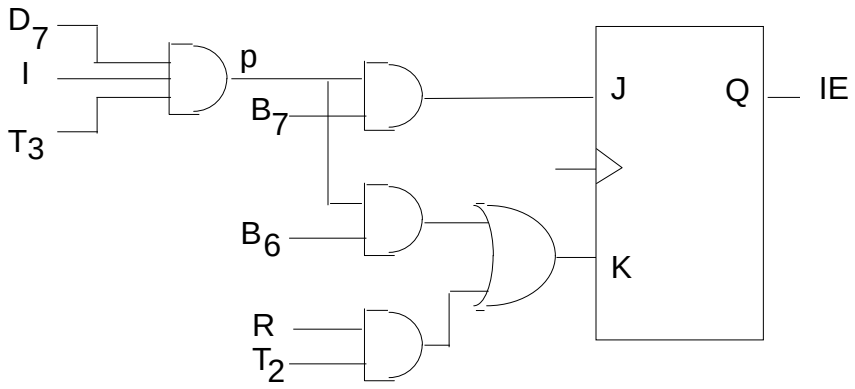
$CLR(AR) = RT_0$

$INR(AR) = D_5T_4$



Control of IEN Flip-Flop

pB_7: IEN ← 1 (I/O Instruction)
pB_6: IEN ← 0 (I/O Instruction)
RT_2: IEN ← 0 (Interrupt)
=>
Set (IEN) = pB_7
Clear(IEN) = $pB_6 + RT_2$
$p = D_7IT_3$ (Input/Output Instruction)



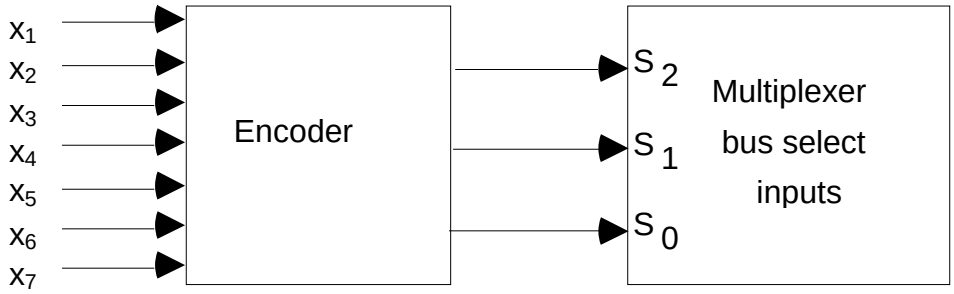
Control of Common Bus

16-bit common bus is controlled by three selection inputs S2, S1 and S0. The decimal number associated with each component of the bus determines the equivalent binary number that must be applied to the selection inputs to select the registers. This is described by the truth table given below:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	S_2	S_1	S_0	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

To find the logic that makes x1=1 we scan and extract all the statements that use AR as source.

D_4T_4: PC ← AR
D_5T_5: PC ← AR
=>
$x_1 = D_4T_4 + D_5T_5$



Control of AC Register

All the statements that change the content of AC

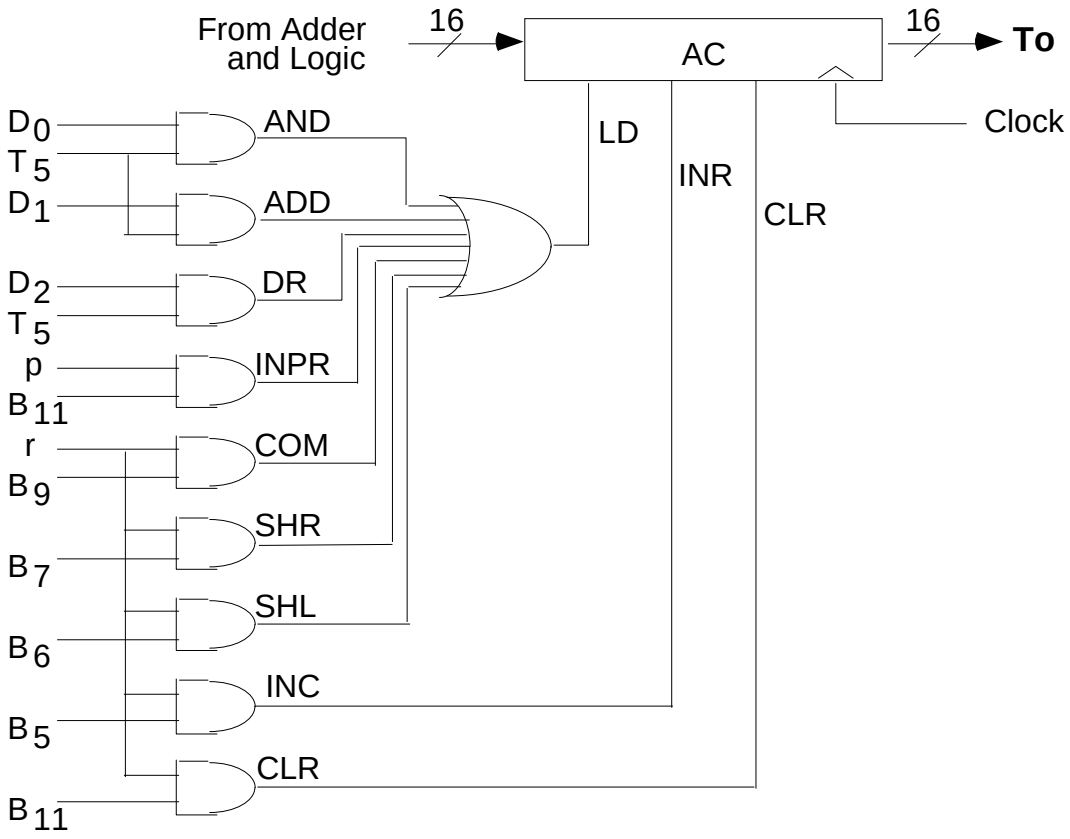
D_0T_5:	AC ← AC ∧ DR	AND with DR
D_1T_5:	AC ← AC + DR	Add with DR
D_2T_5:	AC ← DR	Transfer from DR
pB_{11}:	AC(0-7) ← INPR	Transfer from INPR
rB_9:	AC ← AC′	Complement
rB_7 :	AC ← shr AC, AC(15) ← E	Shift right
rB_6 :	AC ← shl AC, AC(0) ← E	Shift left
rB_{11} :	AC ← 0	Clear
rB_5 :	AC ← AC + 1	Increment

=>

$LD(AC) = D_0T_5 + D_1T_5 + D_2T_5 + pB_{11} + rB_9 + rB_7 + rB_6$

$CLR(AC) = rB_{11}$

$INR(AC) = rB_5$



Chapter 4
Microprogrammed control

Terminologies

Microprogram
- ✓ Program stored in memory that generates all the control signals required to execute the instruction set correctly
- ✓ Consists of microinstructions

Microinstruction
- ✓ Contains a control word and a sequencing word
- ✓ Control Word – contains all the control information required for one clock cycle
- ✓ Sequencing Word - Contains information needed to decide the next microinstruction address

Control Memory(Control Storage: CS)
- ✓ Storage in the microprogrammed control unit to store the microprogram

Writeable Control Memory(Writeable Control Storage:WCS)
- ✓ CS whose contents can be modified:
  -> Microprogram can be changed
  -> Instruction set can be changed or modified

Dynamic Microprogramming
- ✓ Computer system whose control unit is implemented with a microprogram in WCS.
- ✓ Microprogram can be changed by a systems programmer or a user

Control Address Register:
- ✓ Control address register contains address of microinstruction

Control Data Register:
- ✓ Control data register contains microinstruction

Sequencer:
- ✓ The device or program that generates address of next microinstruction to be executed is called sequencer.

Address Sequencing

Process of finding address of next micro-instruction to be executed is called address sequencing. Address sequencer must have capabilities of finding address of next micro-instruction in following situations:
- ✓ In-line Sequencing
- ✓ Unconditional Branch
- ✓ Conditional Branch
- ✓ Subroutine call and return
- ✓ Looping
- ✓ Mapping from instruction op-code to address in control memory.

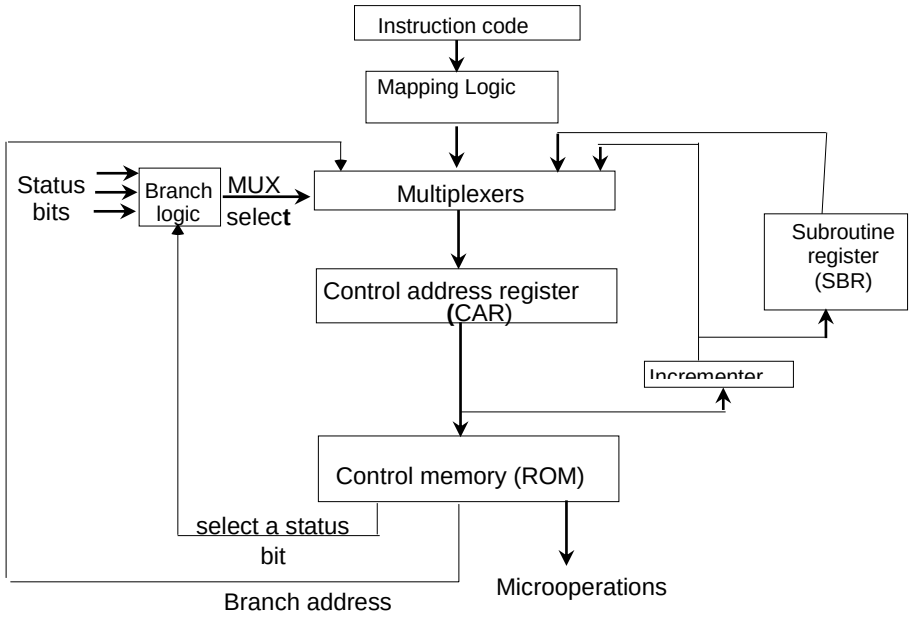


Fig: Block diagram of address sequencer.

- ✓ Control address register receives address of next micro instruction from different sources.
- ✓ Incrementer simply increments the address by one
- ✓ In case of branching branch address is specified in one of the field of microinstruction.
- ✓ In case of subroutine call return address is stored in the register SBR which is used when returning from called subroutine.

Conditional Branch

If Condition is true, set the appropriate field of status register to 1. Conditions are tested for O (overflow), N (negative), Z (zero), C (carry), etc.
Then test the value of that field if the value is 1 take branch address from the next address field of the current microinstruction)
Otherwise simple increment the address.

Unconditional Branch

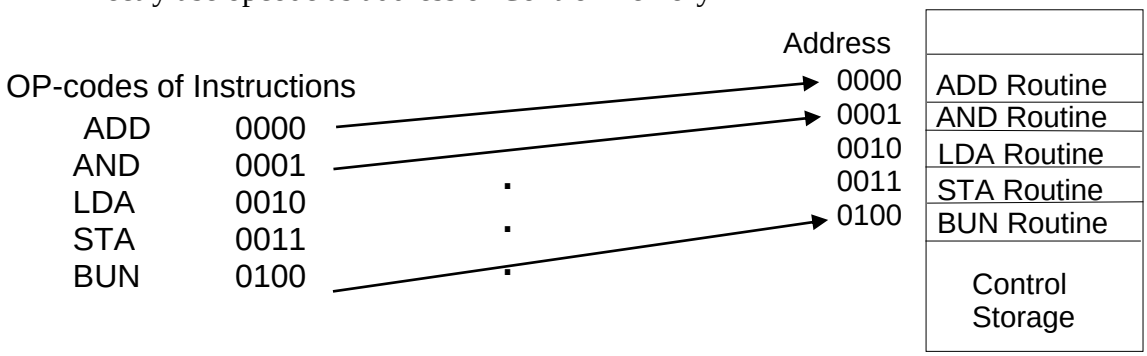
Fix the value of one status bit at the input of the multiplexer to 1. So that always branching is done

Mapping:

Mapping from the OP-code of an instruction to the address of the Microinstruction which is the starting microinstruction of its subroutine in memory

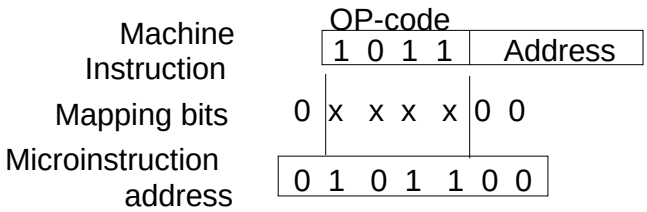
Direct mapping:

Directly use opcode as address of Control memory



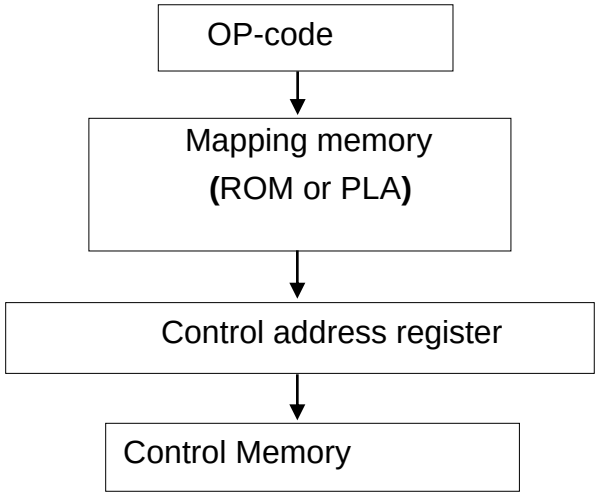
Another approach of mapping:

Modify Opcode to use it as an address of control memory

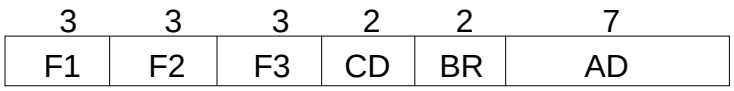


Mapping function implemented by ROM or PLA

Use opcode as address of ROM where address of control memory is stored and than use that address as an address of control memory.



Microinstruction Format



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

Description of CD

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

Description of BR

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

Symbolic Microinstruction

Symbols are used in microinstructions as in assembly language. A symbolic mcroprogram can be translated into its binary equivalent y a microprogram assembler.

Format of Microinstruction:

Contains five fields: label; micro-ops; CD; BR; AD

Label: may be empty or may specify a symbolic address terminated with a colon
Micro-ops: consists of one, two, or three symbols separated by commas

CD: one of {U, I, S, Z}, where U: Unconditional Branch
I: Indirect address bit
S: Sign of AC
Z: Zero value in AC

BR: one of {JMP, CALL, RET, MAP}

AD: one of {Symbolic address, NEXT, empty (in case of MAP and RET)}

Symbolic Microprogram (example)

Sequence of microoperations in the fetch cycle:

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

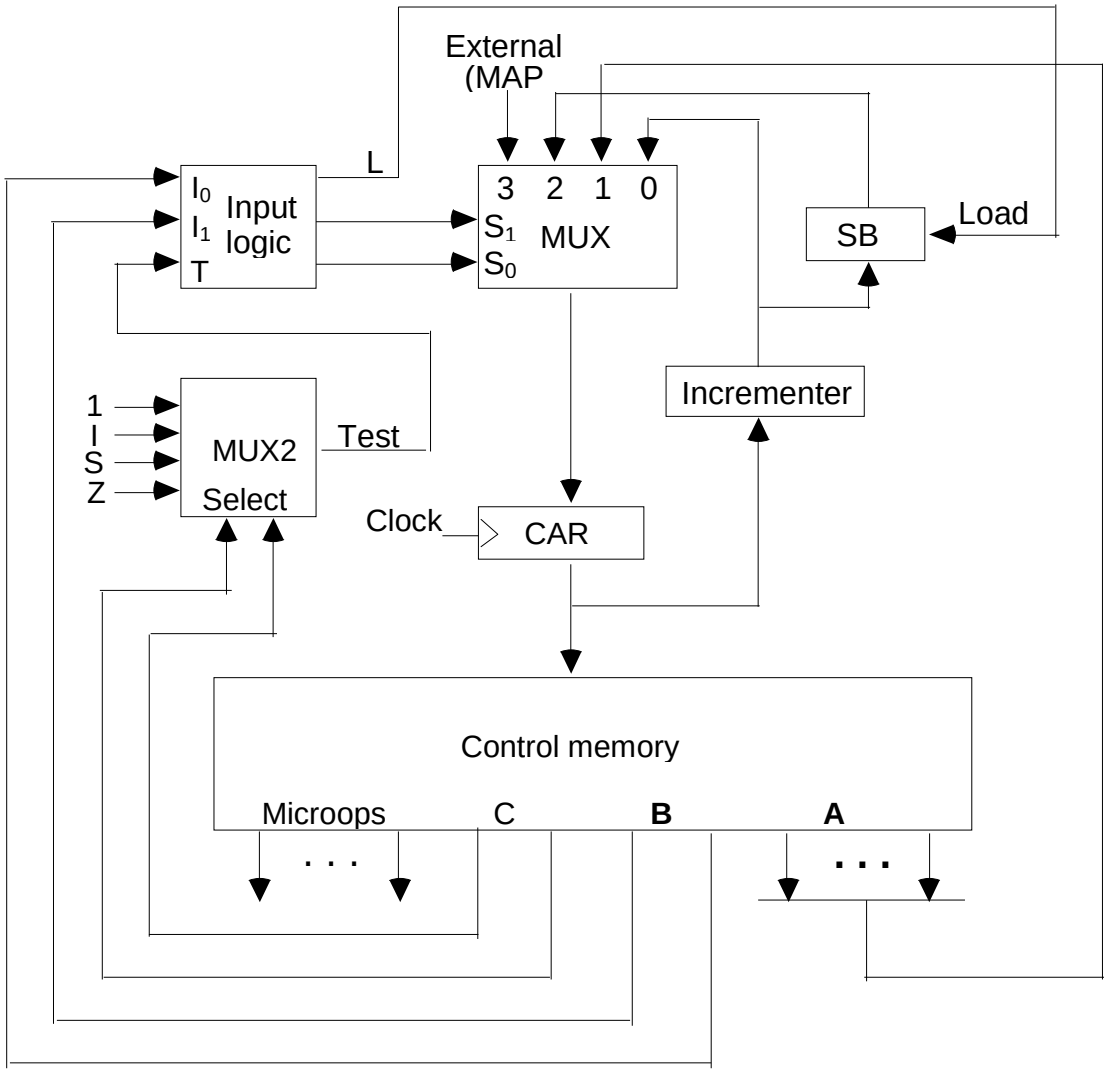
Symbolic microprogram for the fetch cycle:

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

Binary equivalents translated by an assembler

Binary Address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	00000

Microprogram Sequencer



MUX-1 selects an address from one of four sources and routes it into a CAR

- In-Line Sequencing → CAR + 1
- Branch, Subroutine Call → Take address from AD field
- Return from Subroutine → Output of SBR
- New Machine instruction → MAP

MUX-2 Controls the condition and branching as below

Input Logic

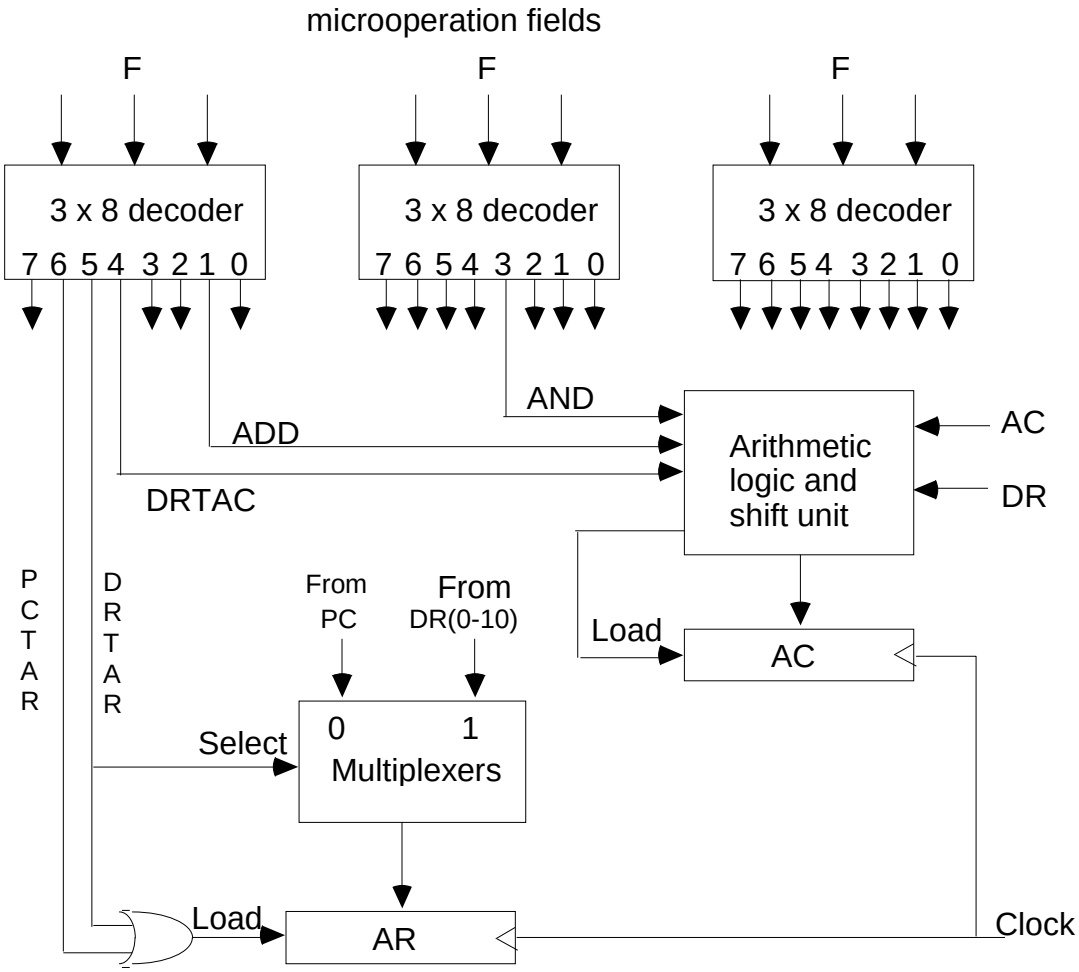
I_0I_1T	Meaning	Source of Address	S_1S_0	L
000	In-Line	CAR+1	00	0
001	JMP	CS (AD)	10	0
010	In-Line	CAR+1	00	0
011	CALL	CS (AD) and SBR <- CAR+1	10	1
10x	RET	SBR	01	0
11x	MAP	DR (11-14)	11	0

$S_0 = I_0$

$S_1 = I_0I_1 + I0'T$

$L = I_0'I_1T$

F Field Decoding

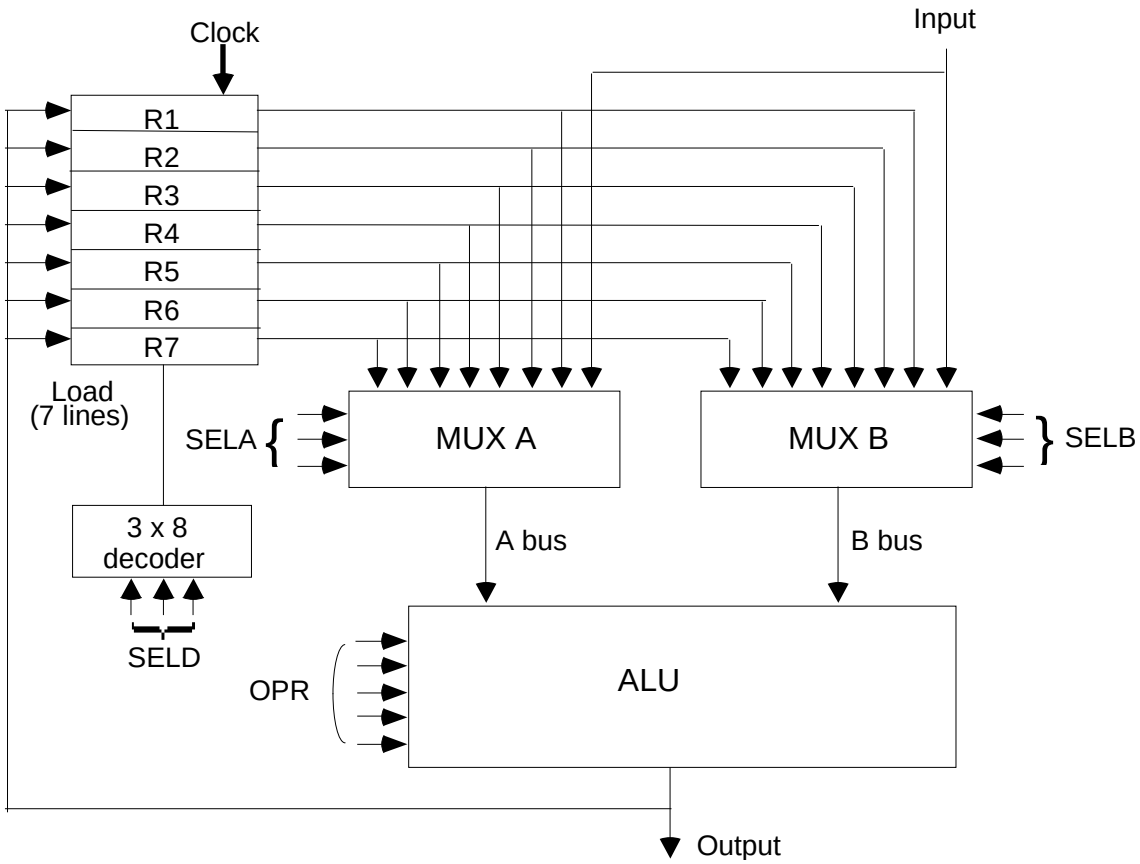


Since there are three microoperation fields we need 3 decoders. Only some of the outputs of decoders are shown to be connected to their output. Each of the output of the decoders must be connected to the proper circuit to initiate the corresponding microoperation. For example when F1=101 the next clock pulse transition transfers content of DR(0-10) to AR (Symbolized by DRTAC). Similarly other operations are also performed.

Chapter 5
CENTRAL PROCESSING UNIT

Bus System and CPU

A bus organization for 7 CPU registers can be shown as below:



All registers are connected to two multiplexers (MUXA and MUXB) that select the registers for bus A and bus B. Registers selected by multiplexers are sent to ALU. Another selector (OPR) connected to ALU selects the operation for the ALU. Output produced by CPU is stored in some register and the destination register for storing the result is activated by the destination decoder (SELD).

Example: R1 ← R2 + R3

- MUX A selector (SELA): BUS A ← R2
- MUX B selector (SELB): BUS B ← R3
- ALU operation selector (OPR): ALU to ADD
- Decoder destination selector (SELD): R1 ← Out Bus

Control word

Combination of all selection bits of a unit is called control word. Control Word for above CPU is as below

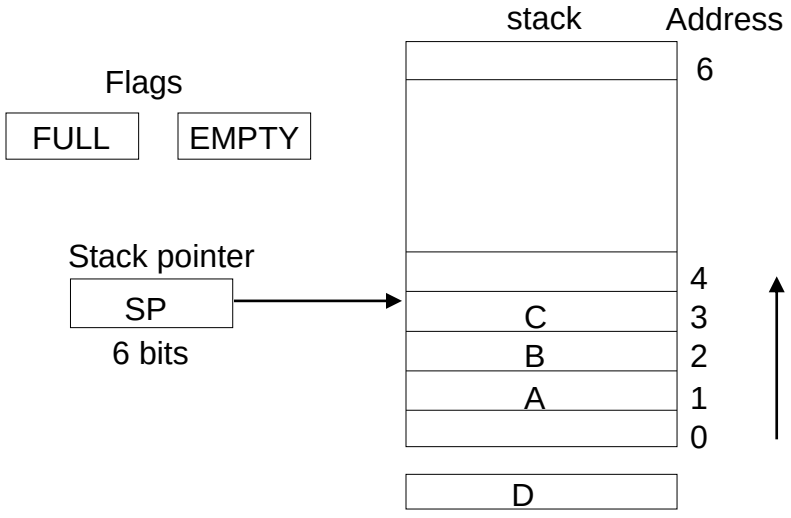
3	3	3	5
SELA	SELB	SELD	OP

Examples of Microoperations for CPU

Microoperation	Symbolic Designation SELA	SELB	SELD	OPR	Control Word
R1 ← R2 – R3	R2	R3	R1	SUB	010 011 001 00101
R4 ← R4 ∨ R5	R4	R5	R4	OR	100 101 100 01010
R6 ← R6 + 1	R6	-	R6	INCA	110 000 110 00001
R7 ← R1	R1	-	R7	TSFA	001 000 111 00000
Output ← R2	R2	-	None	TSFA	010 000 000 00000
Output ← Input	Input	-	None	TSFA	000 000 000 00000
R4 ← shl R4	R4	-	R4	SHLA	100 000 100 11000
R5 ← 0	R5	R5	R5	XOR	101 101 101 01100

Register Stack

It is the collection of finite number of registers. Stack pointer (SP) points to the register that is currently at the top of stack.

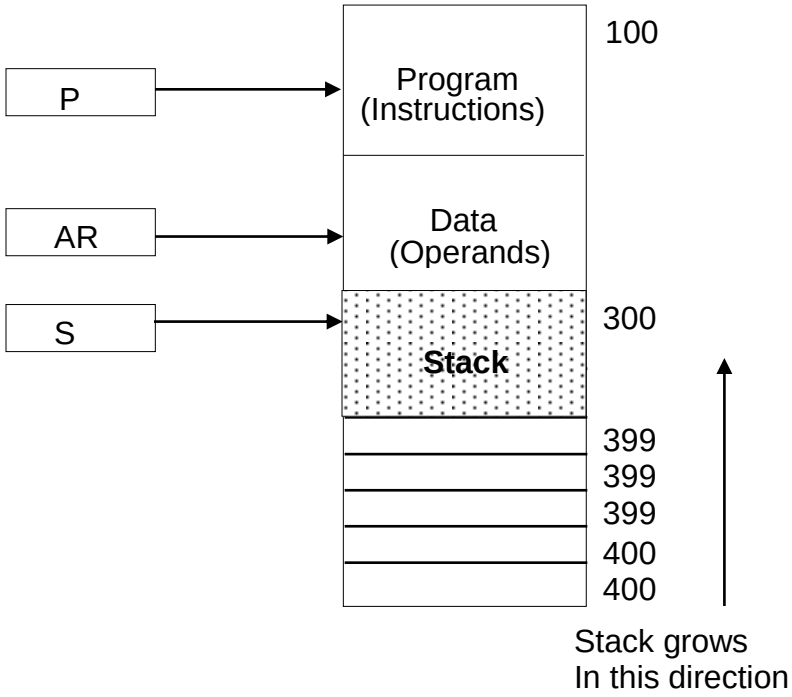


/* Initially, SP = 0, EMPTY = 1(true), FULL = 0(false) */

Push
SP ← SP + 1
M[SP] ← DR
If (SP = 0) then (FULL ← 1)
EMPTY ← 0

Pop
DR ← M[SP]
SP ← SP – 1
If (SP = 0) then (EMPTY ← 1)
FULL ← 0

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



PUSH:
SP ← SP - 1
M[SP] ← DR

POP:
DR ← M[SP]
SP ← SP + 1

PROCESSOR ORGANIZATION

In general, most processors are organized in one of 3 ways

- Single register (Accumulator) organization
  - » Basic Computer is a good example
  - » Accumulator is the only general purpose register
  - » Uses implied accumulator register for all operations

  E.g.

```
ADD  X          // AC ← AC + M[X]
LDA  Y          // AC ← M[Y]
```

- General register organization
  - » Used by most modern computer processors
  - » Any of the registers can be used as the source or destination for computer operations

  e.g.

```
ADD  R1, R2, R3     // R1 ← R2 + R3
ADD  R1, R2         // R1 ← R1 + R2
MOV  R1, R2         // R1 ← R2
ADD  R1, X          // R1 ← R1 + M[X]
```

- Stack organization
  - » All operations are done with the stack
  - » For example, an OR instruction will pop the two top elements from the stack, do a logical OR on them, and push the result on the stack

  e.g.

```
PUSHX           // TOS ← M[X]
ADD             // TOS=TOP(S) + TOP(S)
```

Types of instruction:

The number of address fields in the instruction format depends on the internal organization of CPU. On the basis of no. of address field we can categorize the instruction as below:

- **Three-Address Instructions**

  Program to evaluate X = (A + B) * (C + D):

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

  Results in short programs
  Instruction becomes long (many bits)

- **Two-Address Instructions**

  Program to evaluate X = (A + B) * (C + D) :

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

  Tries to minimize the size of instruction
  Size of program is relative larger.

- **One-Address Instructions**

  Use an implied AC register for all data manipulation

  Program to evaluate X = (A + B) * (C + D):

```
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
```

  Memory access is only limited to load and store
  Large program size

- **Zero-Address Instructions**

  Can be found in a stack-organized computer
  Program to evaluate X = (A + B) * (C + D):

```
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
```

On the basis of type of operation performed by instruction we can categorize instructions as below:

- **Data transfer instructions**
  Used for transferring data from memory to register, register to memory, register to register, memory to memory, input device to register and from register to output device.

Load	LD	→ Transfers data from memory to CPU
Store	ST	→ Transfers data from CPU to memory
Move	MOV	→ Transfers data from memory to memory
Input	IN	→ transfers data from input device to register

- **Data manipulation instructions**
  Used for performing any type of calculations on data.
  Data manipulation instructions can be further divided into three types:
  - Arithmetic instructions
    Examples

Operation	**Symbol**
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB

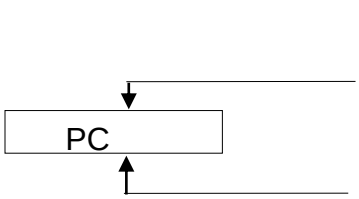
  - Logical and bit manipulation instructions
    Some examples:

Operation	**Symbol**
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR

  - Shift instructions
    Some examples:

Operation	**symbol**
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL

- **Program Control Instructions**
  Used for controlling the execution flow of programs



Some examples:

Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RTN

Addressing Modes

Specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.
We use variety of addressing modes:

- To give programming flexibility to the user
- To use the bits in the address field of the instruction efficiently

Types of addressing modes:

- **Implied Mode**
  Address of the operands are specified implicitly in the definition of the instruction
  - No need to specify address in the instruction
  - Examples from Basic Computer CLA, CME, INP
    ADD X;
    PUSH Y;

- **Immediate Mode**
  Instead of specifying the address of the operand, operand itself is specified in the instruction.
  - No need to specify address in the instruction
  - However, operand itself needs to be specified
  - Sometimes, require more bits than the address
  - Fast to acquire an operand

- **Register Mode**
  Address specified in the instruction is the address of a register
  - Designated operand need to be in a register
  - Shorter address than the memory address

- Saving address field in the instruction
- Faster to acquire an operand than the memory addressing

- **Register Indirect Mode**
  Instruction specifies a register which contains the memory address of the operand
  – Saving instruction bits since register address is shorter than the memory address
  – Slower to acquire an operand than both the register addressing or memory addressing
  – EA= content of R.

- **Autoincrement or Autodecrement Mode**
  – When the address in the register is used to access memory, the value in the register is incremented or decremented by 1 automatically. i.e in case of register indirect mode.

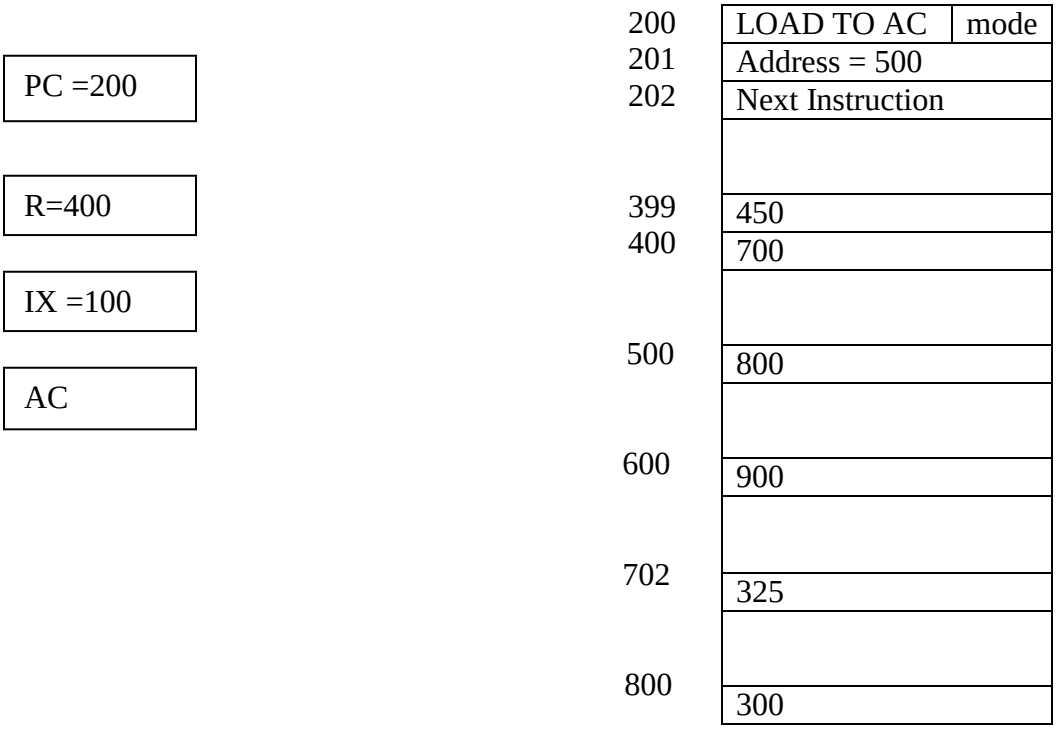
- **Direct Address Mode**
  Instruction specifies the memory address which can be used directly to access the Memory
  – Faster than the other memory addressing modes
  – Too many bits are needed to specify the address for a large physical memory Space
  – EA= IR(address)

- **Indirect Addressing Mode**
  – The address field of an instruction specifies the address of a memory location that contains the address of the operand
  – When the abbreviated address is used large physical memory can be addressed with a relatively small number of bits
  – Slow to acquire an operand because of an additional memory access
  – EA= M[IR (address)]

- **Relative Addressing Modes**
  The Address fields of an instruction specifies the part of the address which can be used along with a designated register to calculate the address of the operand
  - Address field of the instruction is short
  - Large physical memory can be accessed with a small number of address bits

3 different Relative Addressing Modes:
* PC Relative Addressing Mode
  - EA = PC + IR(address)
* Indexed Addressing Mode
  - EA = IX + IR(address) { IX is index register }
* Base Register Addressing Mode
  - EA = BAR + IR(address)

Addressing modes (Example)



Direct address 500 // AC ← M[500]
Value = 800
Immediate operand // AC ← 500
Value = 500
Indirect address 500 // AC ← M[M[500]]
Value = 300
Relative address 500 // AC ← M[PC+500]
Value = 325
Indexed address 500 // AC ← (IX+500)
Value = 900
Register 500 // AC ← R1
400
Register indirect 500 // AC ← M[R1]
Value = 700
Autoincrement 500 // AC ← (R1)
Value = 700
Autodecrement 399 /* AC ← -(R) */

RISC and CISC

Complex Instruction Set Computer (CISC):

Computers with many instructions and addressing modes came to be known as Complex Instruction Set Computers (CISC).One goal for CISC machines was to have a machine language instruction to match each high-level language statement type so that job of compiler writer becomes easy. Characteristics of CISC computers are:

- The large number of instructions and addressing modes led CISC machines to have variable length instruction formats
- Multiple operand instructions could specify different addressing modes for each operand
- Variable length instructions greatly complicate the fetch and decode problem for a processor
- They have instructions that act directly on memory addresses due to which multiple memory cycle are needed for executing instructions.
- Microprogrammed control is used rather than hardwired

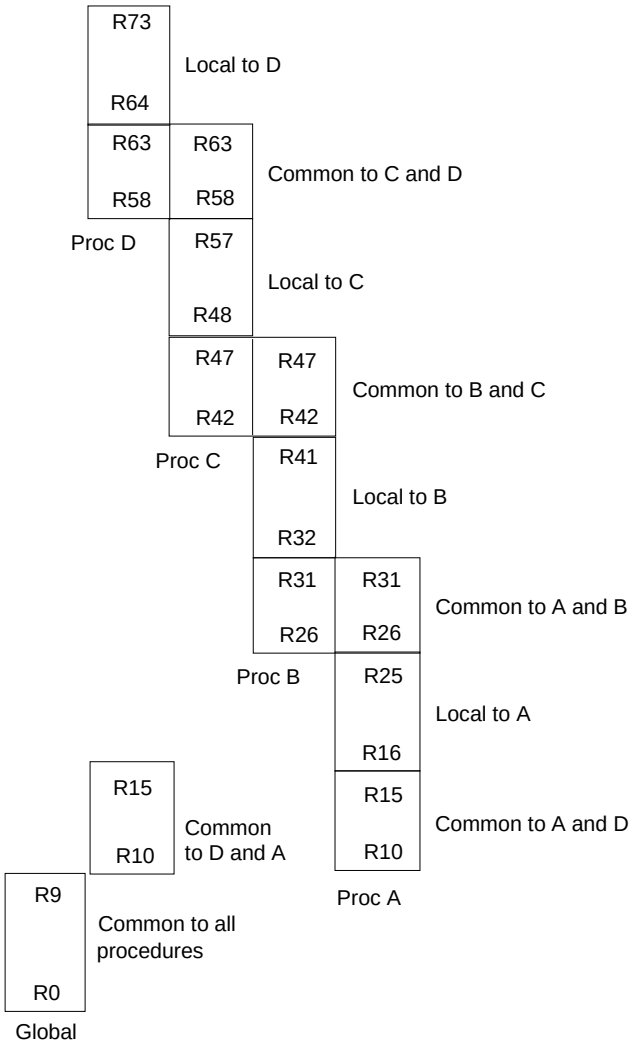
Reduced Instruction Set Computer (RISC)

Reduced Instruction Set Computers (RISC) were proposed as an alternative The underlying idea behind RISC processors is to simplify the instruction set and reduce instruction execution time. Characteristics of RISC are:

- Few instructions
- Few addressing modes
- Only load and store instructions access memory
- All other operations are done using on-processor registers
- Fixed length instructions
- Single cycle execution of instructions
- The control unit is hardwired, not microprogrammed

Register Overlapped Windows:

The procedure (function) call/return is the most time-consuming operations in typical HLL programs. The depth of procedure activation is within a relatively narrow range. If we use multiple small sets of registers (windows), each assigned to a different procedure, a procedure call automatically switches the CPU to use a different window of registers, rather than saving registers in memory. Windows for adjacent procedures are overlapped to allow parameter passing.



There are three classes of registers:

- Global Registers
  - » Available to all functions
- Window local registers
  - » Variables local to the function
- Window shared registers
  - » Permit data to be shared without actually needing to copy it

Only one register window is active at a time. The active register window is indicated by a pointer. When a function is called, a new register window is activated. This is done by incrementing the pointer. When a function calls a new function, the high numbered registers of the calling function window are shared with the called function as the low numbered registers in its register window. This way the caller's high and the called function's low registers overlap and can be used to pass parameters and results

The advantage of overlapped register windows is that the processor does not have to push registers on a stack to save values and to pass parameters when there is a function call.

This saves

- Accesses to memory to access the stack.
- The cost of copying the register contents at all

And, since function calls and returns are so common, this results in a significant savings relative to a stack-based approach

Chapter 7
Input-Output Organization

I/O subsystem

The input-output subsystem (also referred as I/O) proves an efficient mode of communication between the central system and outside environment. Data and programs must be entered into the computer memory for processing and result of processing must be must be recorded or displayed for the user

Peripheral devices

Any input/output devices connected to the computer are called peripheral devices.

Input Devices

- Keyboard
- Card Reader
- Digitizer
- Screen Input Devices
- Touch Screen
- Light Pen
- Mouse
-

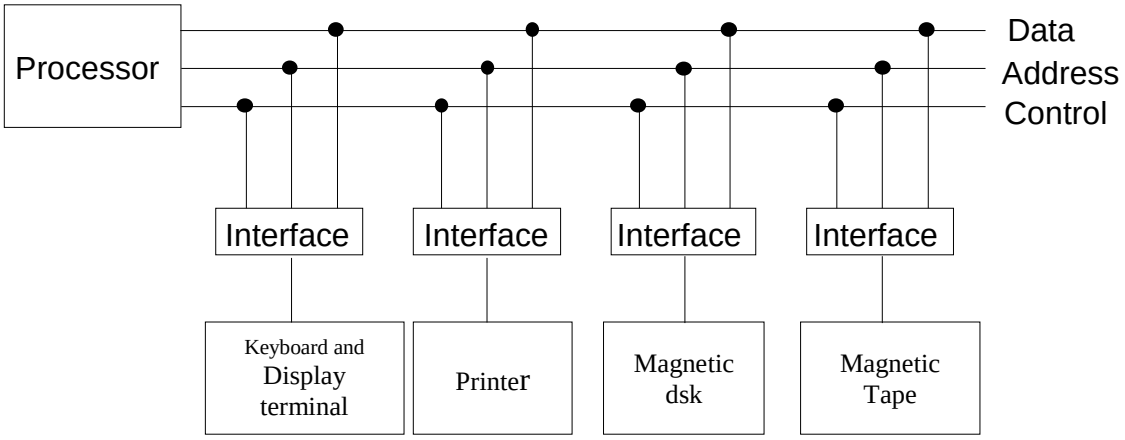
Output Devices

- CRT
- Printer (Impact, Ink Jet, Laser, Dot Matrix)
- Plotter
- Voice input devices

Input-Output Interface

- Provides a method for transferring information between internal storage (such as memory and CPU registers) and external I/O devices
- Resolves the *differences* between the computer and peripheral devices
  - Peripherals - Electromechanical Devices, CPU or Memory - Electronic Device
  - Data Transfer Rate
    - » Peripherals - Usually slower
    - » CPU or Memory - Usually faster than peripherals
      - Some kinds of Synchronization mechanism may be needed
  - Unit of Information
    - » Peripherals – Byte, Block, …
    - » CPU or Memory – Word
  - Data representations may differ

I/O bus and interface modules



I/O bus from the processor is connected to all peripheral interfaces. To communicate with a particular device, the processor places a device address on the address lines. Each peripheral has an interface module associated with its interface. Functions of an interface are as below:

- Decodes the device address (device code)
- Decodes the I/O commands (operation or function code)
- Provides signals for the peripheral controller
- Synchronizes the data flow and
- Supervises the transfer rate between peripheral and CPU or Memory

Types of I/O command

Control command:-Issued to activate the peripheral and to inform it what to do?
Status command:-Used to check the various status conditions of the interface before a transfer is initiated
Data input command:-Cause the interface to read the data from the peripheral and place it into the interface buffer.
Data output command:-Causes the interface to read the data from the bus and save it into the interface buffer

I/O bus and memory bus

Memory bus is used for information transfers between CPU and the MM (main memory). I/O bus is for information transfers between CPU and I/O devices through their I/O interface

Physical Organizations

Many computers use a common single bus system for both memory and I/O interface units

- Use one common bus but separate control lines for each function
- Use one common bus with common control lines for both functions

Some computer systems use two separate buses,

- One to communicate with memory and the other with I/O interfaces

I/O Bus

Communication between CPU and all interface units is via a common I/O bus. An interface connected to a peripheral device may have a number of data registers, a control register, and a status register. A command is passed to the peripheral by sending to the appropriate interface register

Isolated vs. Memory mapped I/O

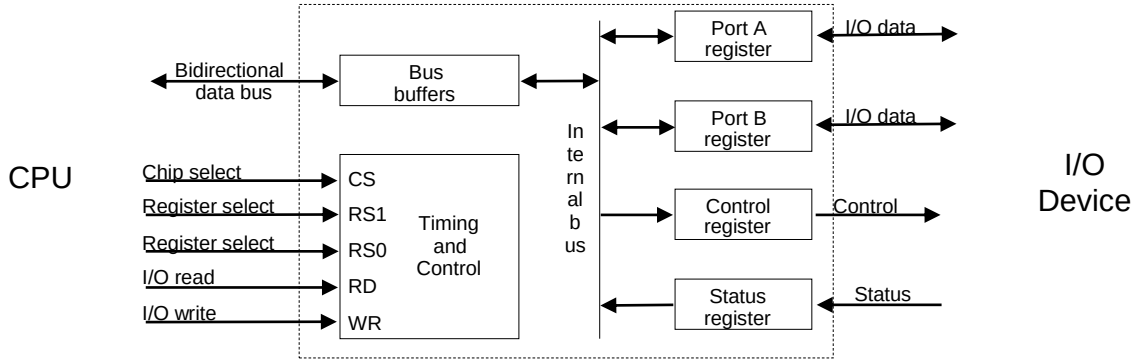
Isolated I/O

- Separate I/O read/write control lines in addition to memory read/write control lines
- Separate (isolated) memory and I/O address spaces
- Distinct input and output instructions

Memory-mapped I/O

- A single set of read/write control lines (no distinction between memory and I/O transfer)
- Memory and I/O addresses share the common address space
  - ➔ reduces memory address range available
- No specific input or output instruction
  - ➔ The same memory reference instructions can be used for I/O transfers
- Considerable flexibility in handling I/O operations

IO Interface Unit



CS	RS1	RS0	Register selected
0	x	x	None - data bus in high-impedence
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

Interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. Control lines I/O read and write are used to specify the input and output respectively. Bidirectional lines represent both data in and out from the CPU. Information in each port can be assigned a meaning

depending on the mode of operation of the I/O device: Port A = Data; Port B =Command; Port C = Status. CPU initializes (loads) each port by transferring a byte to the Control Register. CPU can define the mode of operation of each port

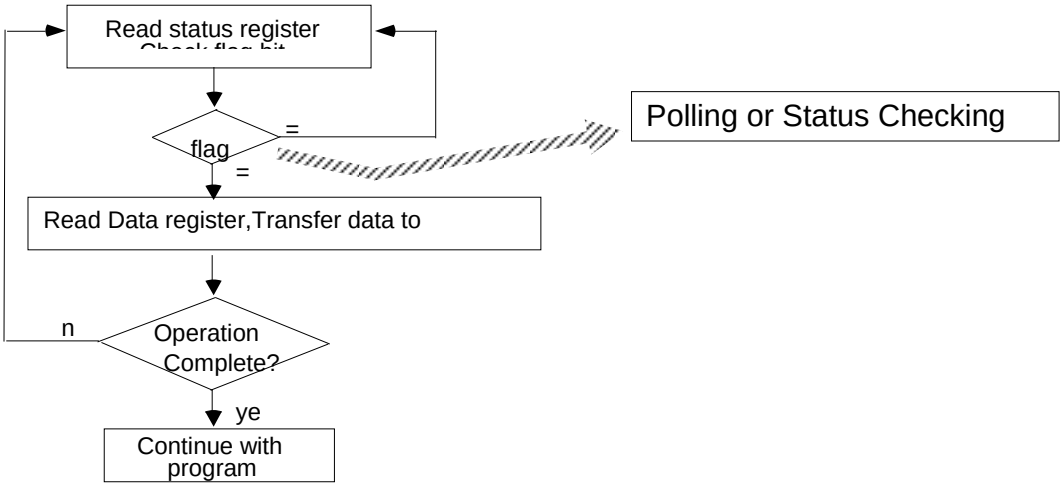
Modes of I/O transfer

Three different Data Transfer Modes between the central computer (CPU or Memory) and peripherals;

- Program-Controlled I/O
- Interrupt-Initiated I/O
- Direct Memory Access (DMA)

Program-Controlled I/O (Input Dev to CPU)

- Continuous CPU involvement
- CPU slowed down to I/O speed
- Simple
- Least hardware



Interrupt Initiated I/O

- Polling takes valuable CPU time
- Open communication only when some data has to be passed -> *Interrupt*.
- I/O interface, instead of the CPU, monitors the I/O device
- When the interface determines that the I/O device is ready for data transfer, it generates an *Interrupt Request* to the CPU
- Upon detecting an interrupt, 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

DMA (Direct Memory Access)

Large blocks of data transferred at a high speed to or from high speed devices, magnetic drums, disks, tapes, etc.

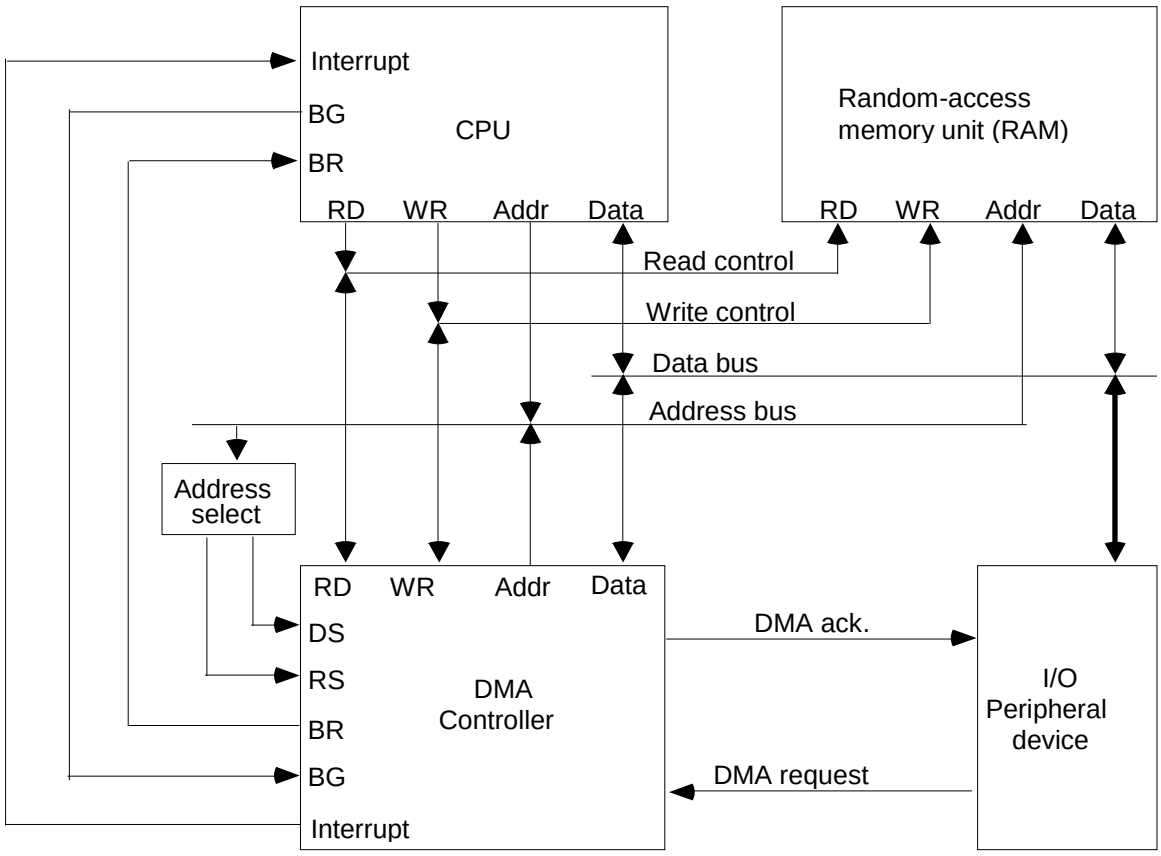
- DMA controller (Interface) provides I/O transfer of data directly to and from the memory and the I/O device
- CPU initializes the DMA controller by sending a memory address and the number of words to be transferred

- Actual transfer of data is done directly between the device and memory through DMA controller freeing CPU for other tasks
- DMA works by stealing the CPU cycles

Cycle Stealing:

- While DMA I/O takes place, CPU is also executing instructions
- DMA Controller and CPU both access Memory which causes memory Access Conflict
- Memory Bus Controller is responsible for coordinating the activities of all devices requesting memory access by using priority schemes
- Memory accesses by CPU and DMA Controller are interwoven; with the top priority given to DMA Controller which is called cycle Stealing
- CPU is usually much faster than I/O(DMA), thus CPU uses the most of the memory cycles
- DMA Controller steals the memory cycles from CPU for those stolen cycles, CPU remains idle
- For those slow CPU, DMA Controller may steal most of the memory cycles which may cause CPU remain idle long time

DMA Transfer



CPU executes instruction to

Load Memory Address Register
Load Word Counter
Load Function(Read or Write) to be performed
Issue a GO command

Upon receiving a GO Command DMA performs I/O operation as follows independently from CPU

Input

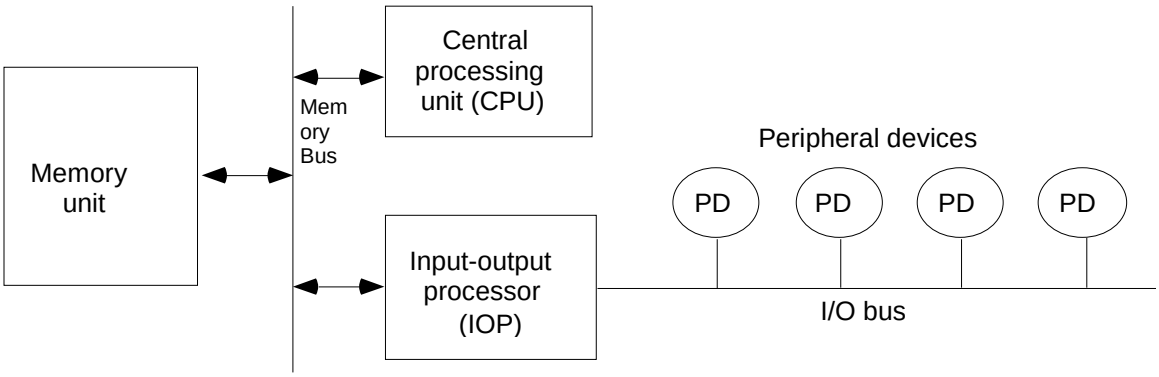
- Send read control signal to Input Device
- DMA controller collects the input from input device byte by bye and assembles the byte into a word until word is full
- Send write control signal to memory
- Increment address register (Address Reg <- Address Reg +1)
- Decrement word count (WC <- WC – 1)
- If WC = 0, then Interrupt to acknowledge done, else repeat same process

Output

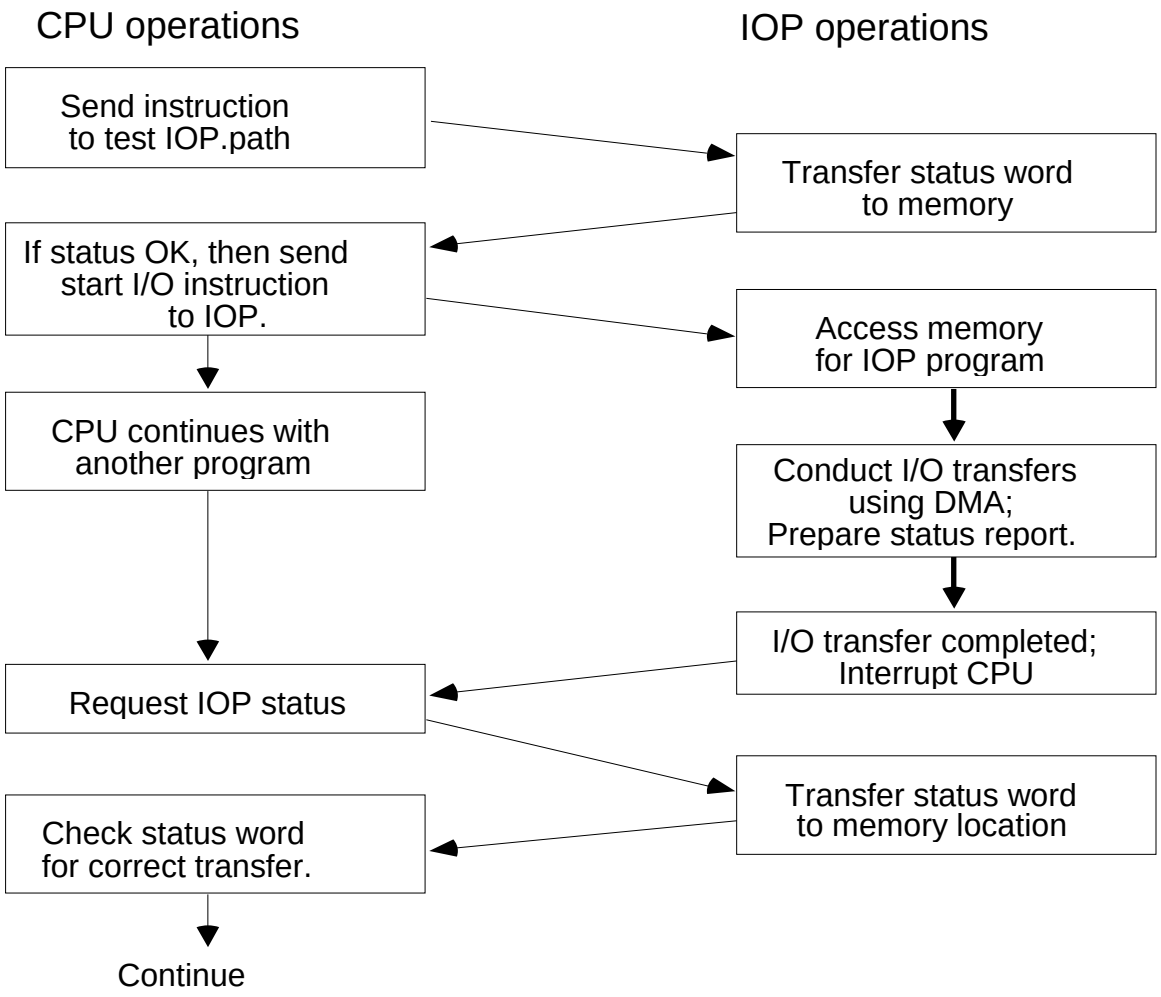
- Send read control signal to memory
- Read data from memory
- Increment address register (Address Reg <- Address Reg +1)
- Decrement word count (WC <- WC – 1)
- Disassemble the word
- Transfer data to the output device byte by byte
- If WC = 0, then Interrupt to acknowledge done, else repeat same process

I/O Processor (I/O Channel)

Processor with direct memory access capability that communicates with I/O devices is called I/O processor (channel). Channel accesses memory by cycle stealing. Channel can execute a channel program stored in the main memory. CPU initiates the channel by executing a channel I/O class instruction and once initiated, channel operates independently of the CPU.



Channel- CPU communication



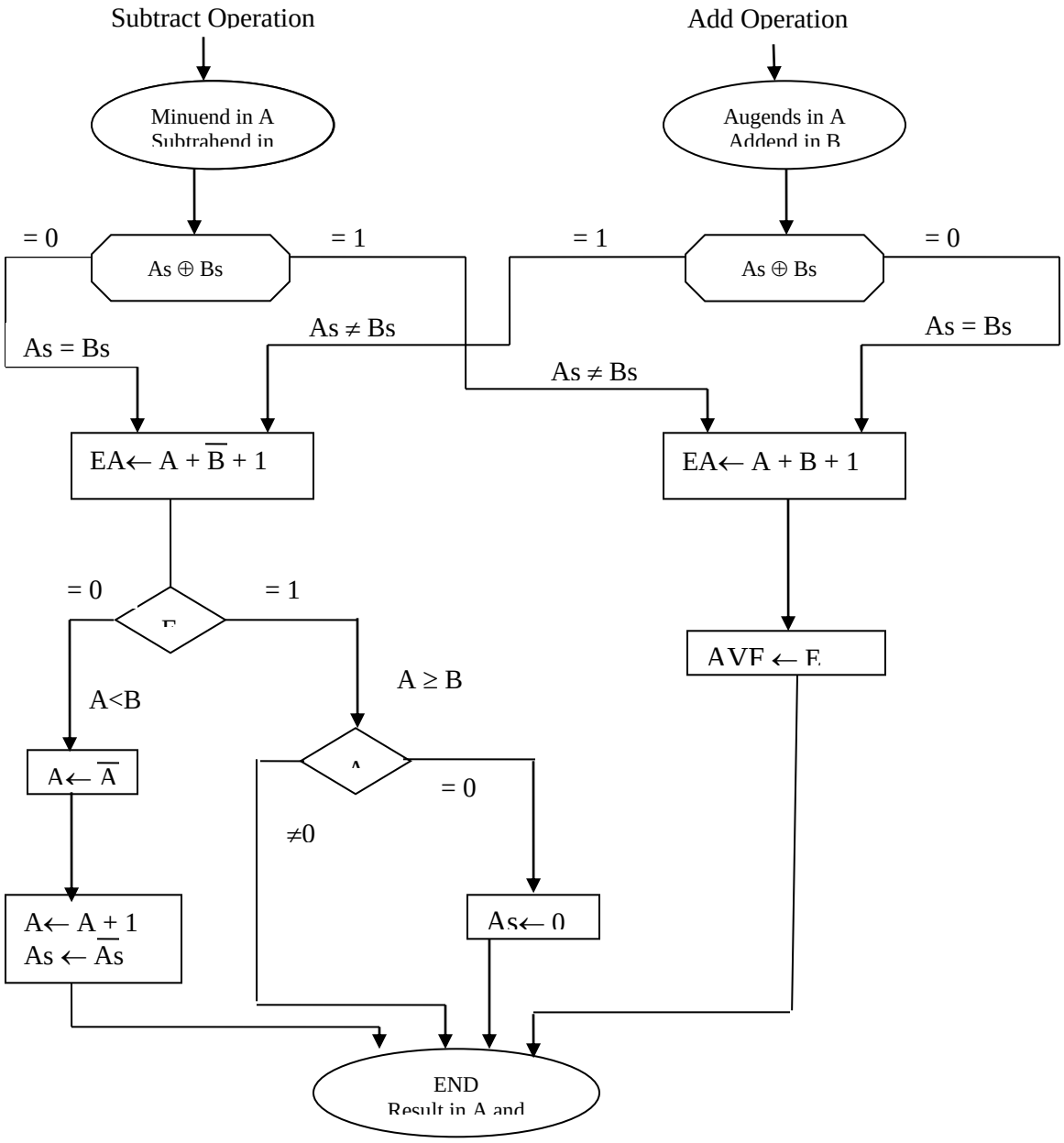
Data Communication Processor

Read your self

Chapter 8

Computer Arithmetic

Addition and Subtraction with Signed Magnitude Data



AVF: Addition overflow Flip-flop

Perform 45 + (-23)

Operation is add

45 = 00101101
-23 = 10010111

As = 0 A=0101101
Bs = 1 B=0010111

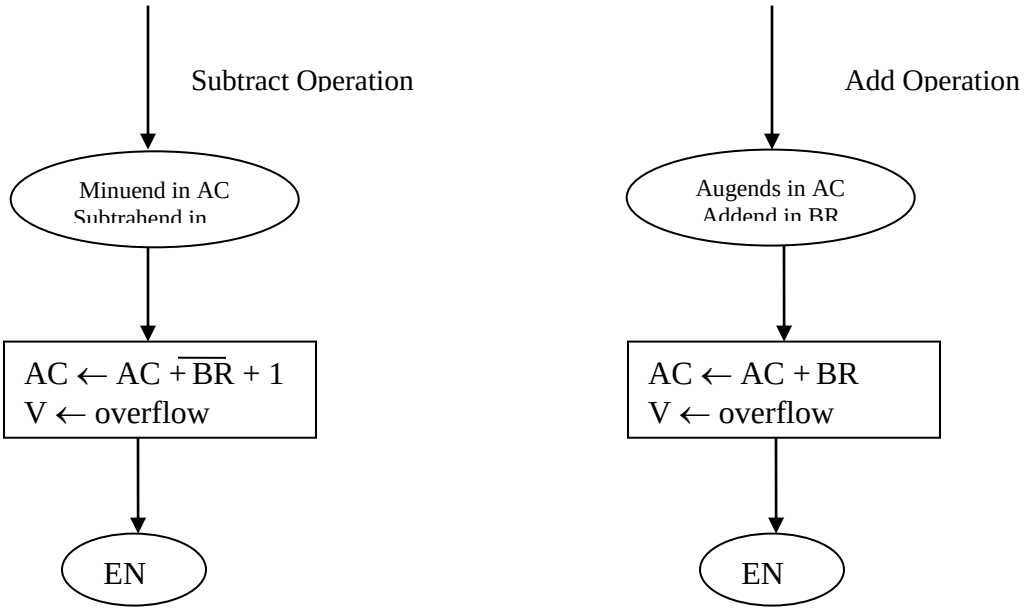
As ⊕ Bs =1

EA=A + B' + 1 = 0101101 + 1101000 +1 = 10010110
AVF=0
=> E=1 A= 0010110

Result is AsA= 0 0010110

Exercise
Perform
(-65) + (50), (-30) + (-12), (20) + (34), (40) – (60), (-20) – (50)

Addition and Subtraction with Signed 2's Complement Data



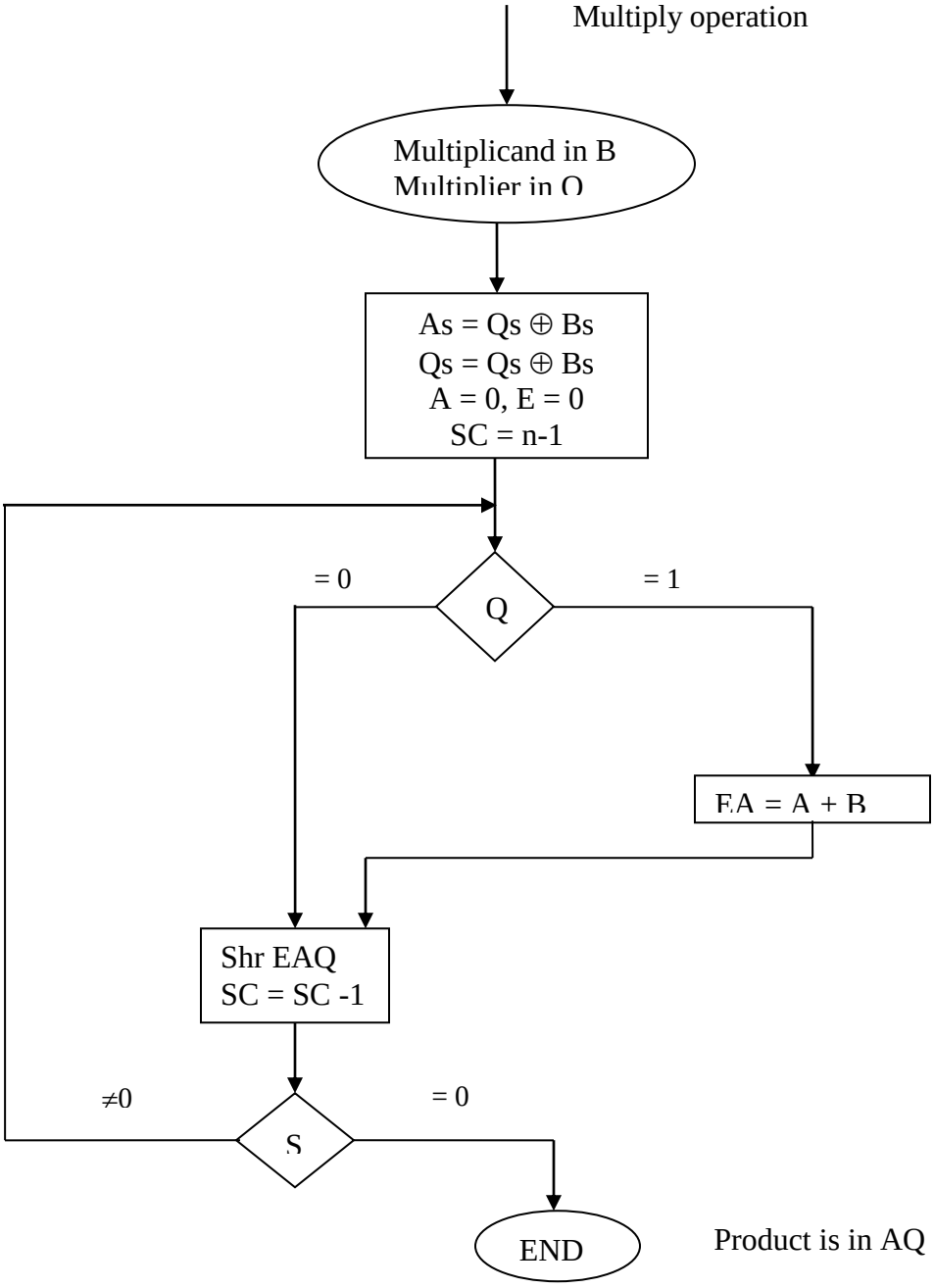
Example:

33 + (-35)

AC = 33 = 00100001
BR = -35 = 2's complement of 35 = 11011101

AC + BR = 11111110 = -2 which is the result

Multiplication with Signed Magnitude Data



Example:

Perform 5 x – 4

Bs B = 5 = 0 0101
Qs Q = - 4 = 1 0100

As = 1
Qs = 1

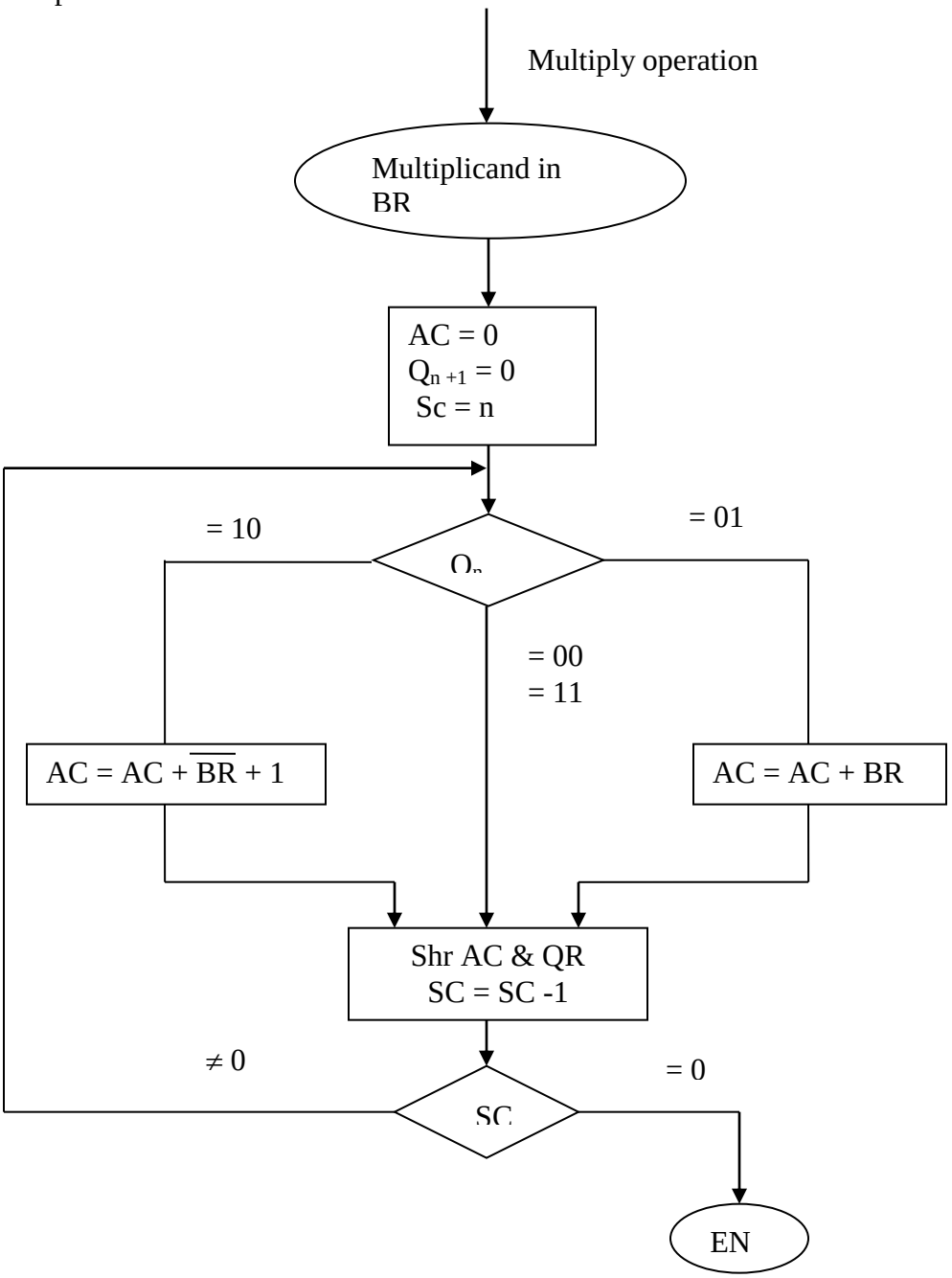
	E	A	Q	SC
	0	0000	0100	4
Step1				
Qn =0				
Shr	0	0000	0010	3
Step2				
Qn =0				
Shr	0	0000	0001	2
Step 3				
Qn=1	0	0000	0001	
		0101		
	0	0101	0001	
Shr	0	0010	1000	1
Step 4				
Qn = 0				
Shr	0	0001	0100	0

As AQ = 0001 0100 = - 20

Question:
Perform the operation 4 x 3, 6 x -7, -9 x 8 by using 5-bit representation

Booth Multiplication Algorithm

Booth multiplication algorithm is used to multiply the numbers represented in signed 2's complement form.



BR = 10111

$\overline{BR}$ + 1 = 01001

	Q_n	Q_{n+1}	AC	QR	Q_{n+1}	SC
	1	0	00000	10011	0	5
Step 1						
Subtract BR			01001	10011		
AShr			00100	11001	1	4
Step 2						
	1	1	00100	11001	1	4
AShr	1	1	00010	01100	1	3
Step 3	0	1	00010	01100	1	3
Add BR	0	1	+10111			
			11001	01100	1	3
Shr			11100	10110	1	2
Step 4						
	0	0	11100	10110	0	2
AShr	0	0	11110	01011	0	1
Step 5						
	1	0	11110	01011	0	1
Subtract BR			01001			
			00111	01011	0	1
AShr			00011	10101	0	0

Terminate: Result in AC & QR = 117

]

Division of Signed Magnitude Data