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 can not 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

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.

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 $\rightarrow$ 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

ASCII Code

The ASCII code is the standard code commonly used for the transmission of binary information. Each character is represented by a 7-bit code and usually an eighth bit is inserted for parity. The code consists of 128 characters. Ninety-five characters represent *graphic symbols* that include upper- and lowercase letters, numerals zero to nine, punctuation marks, and special symbols. Twenty-three characters represent *format effectors*, which are functional characters for controlling the layout of printing or display devices such as carriage return, line feed, horizontal tabulation, and back space. The other 10 characters are used to direct the data communication flow and report its status.

TABLE 3-4 American Standard Code for Information Interchange (ASCII)

Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	$	010 0100
R	101 0010	*	010 1010
S	101 0011	)	010 1001
T	101 0100	−	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

ERROR DETECTION 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:

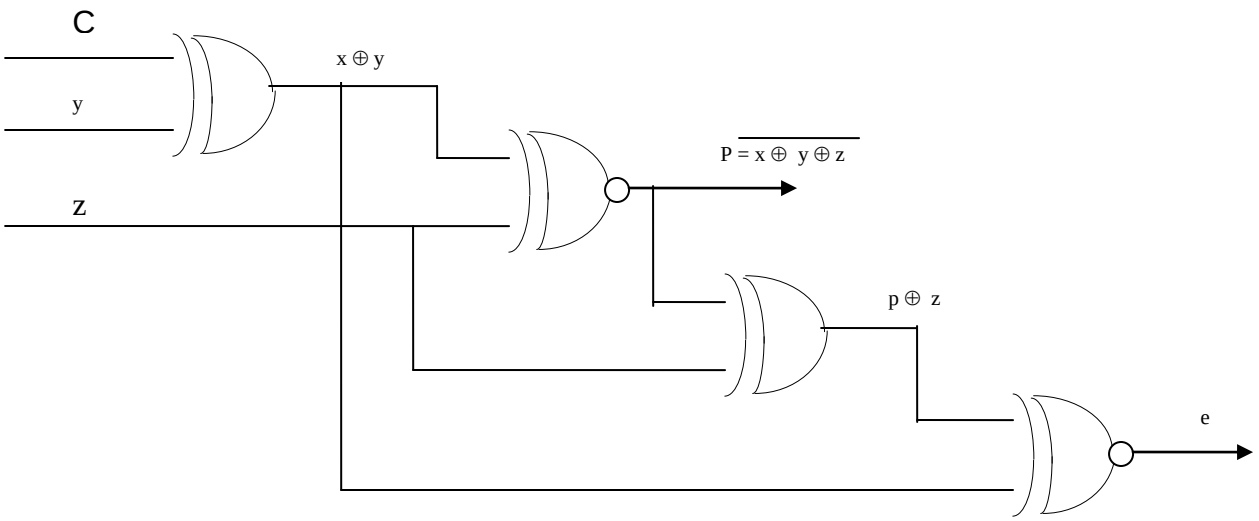
Considers original message as well as parity bit

$e = \overline{p \oplus x \oplus y \oplus z}$

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


C
y
z
x ⊕ y
P = x ⊕ y ⊕ z
p ⊕ z
e

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. …

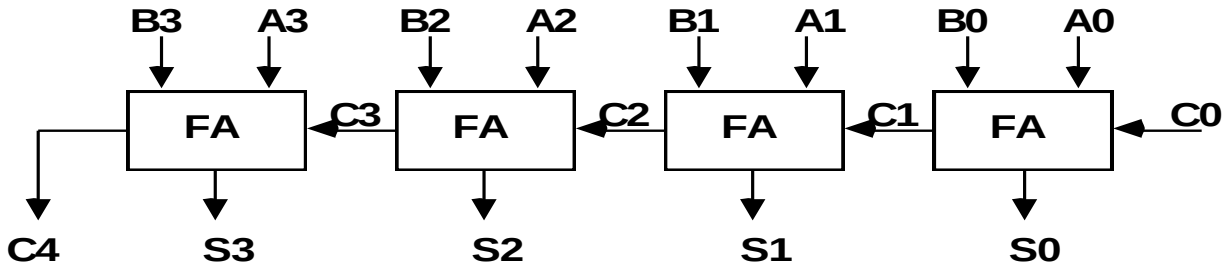
Summary of Typical Arithmetic Micro-Operations

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

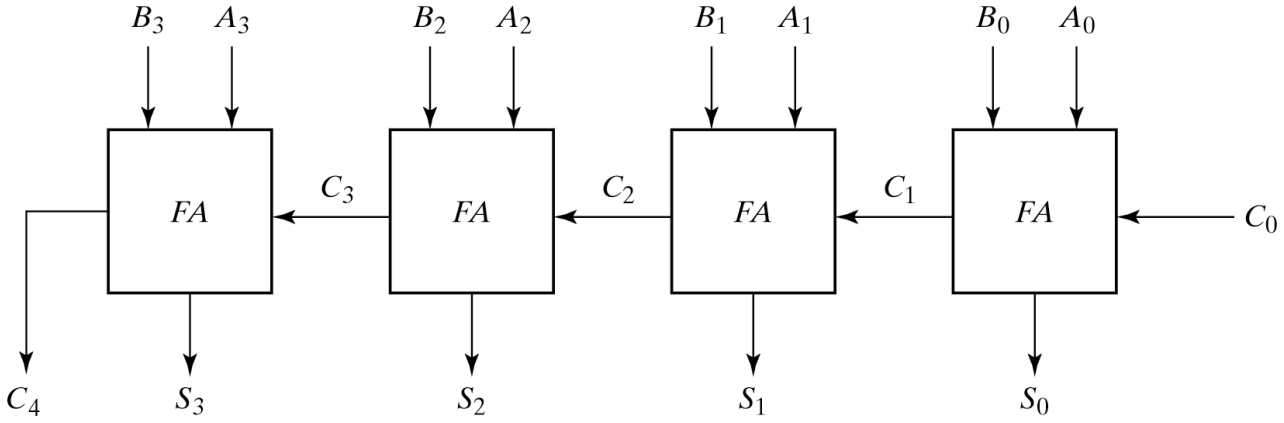
To perform multibit addition in computer a full adder must be allocated for each bit so that all bits can be added simultaneously. Thus, to add two 4-bit numbers to produce a 4-bit sum (with a possible carry), we need four full adders with carry lines cascaded, as shown in the figure given below. For 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

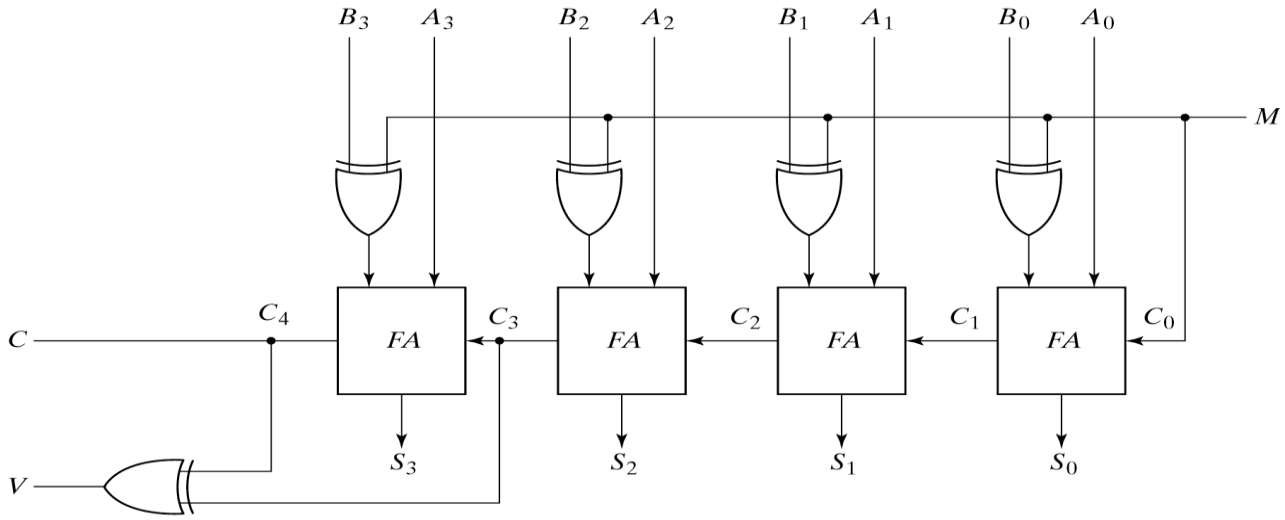
The subtraction A – B can be done by taking the 2's complement of B and adding it to A because A- B = A + (-B). It means if we use the inverters to make 1's complement of B (connecting each Bi to an inverter) and then add 1 to the least significant bit (by setting carry C0 to 1) of binary adder, then 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 don by using exclusive-OR for each Bi and M. Note that $1 \oplus x = x'$ and $0 \oplus x = x$

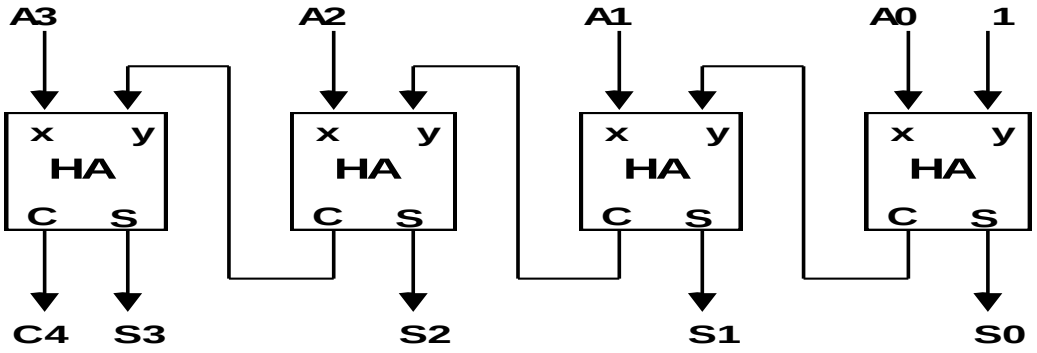
{C = carry bit, V= overflow bit}



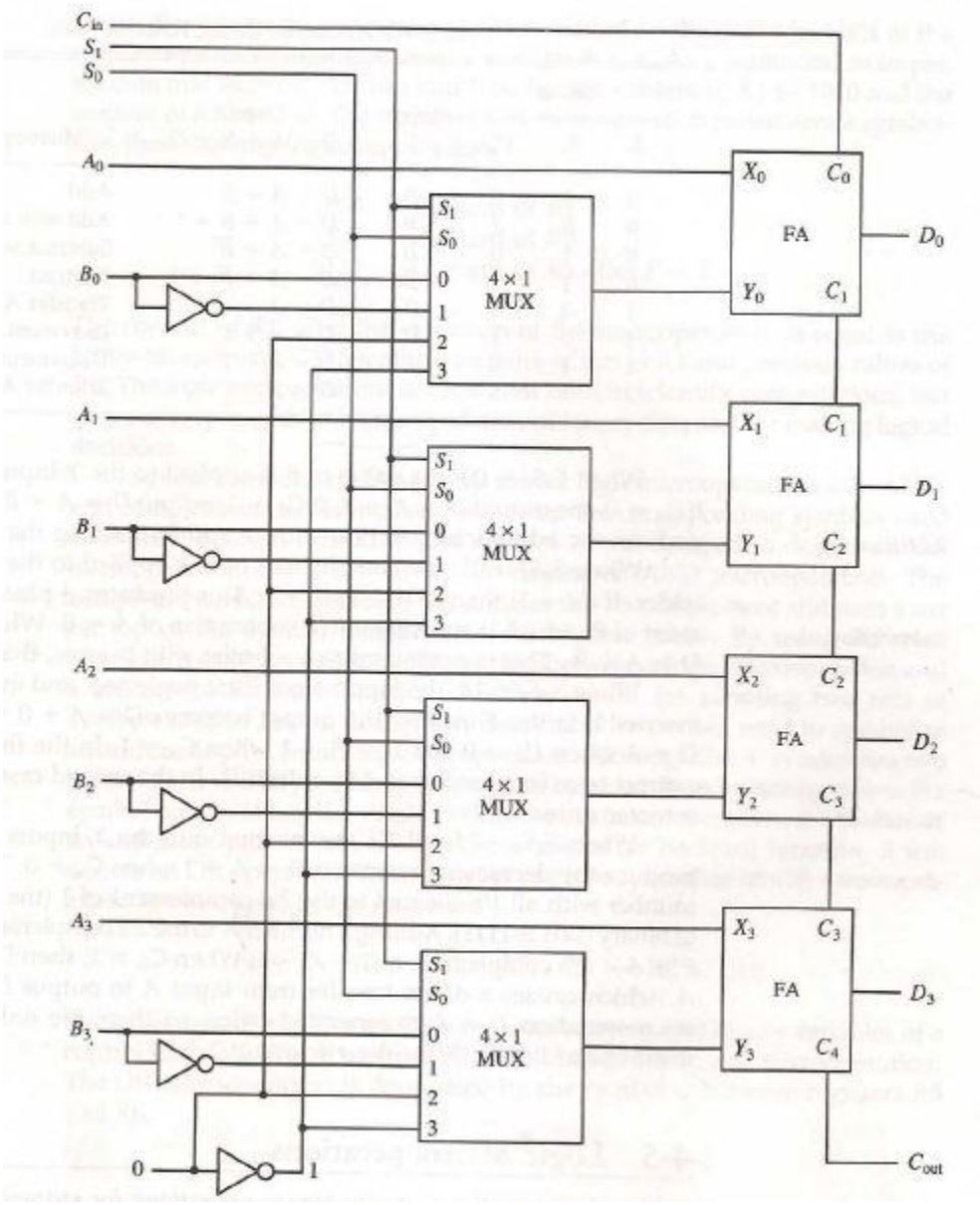
Binary Incrementer

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

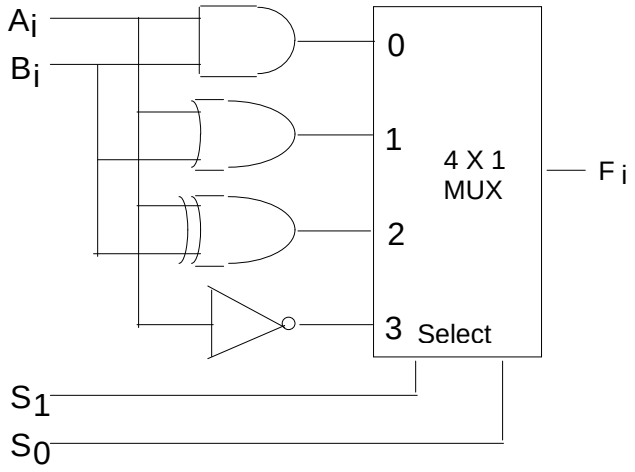
Logic Microoperations

Logic microoperations are bit-wise operations, i.e., they work on the individual bits of data. 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



0

Function table

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

1 1 0 0	A_t	
1 0 1 0	B	
1 1 1 0	A_{t+1}	$(A \leftarrow A + B)$

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

1 1 0 0	A_t	
1 0 1 0	B	
0 1 1 0	A_{t+1}	$(A \leftarrow A \oplus B)$

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

1 1 0 0	A_t	
1 0 1 0	B	
0 1 0 0	A_{t+1}	$(A \leftarrow A \cdot B')$

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

1 1 0 0	A_t	
1 0 1 0	B	
1 0 0 0	A_{t+1}	$(A \leftarrow A \cdot B)$

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

1 1 0 0	A_t	
1 0 1 0	B	
0 1 1 0	A_{t+1}	$(A \leftarrow A \oplus B)$

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)

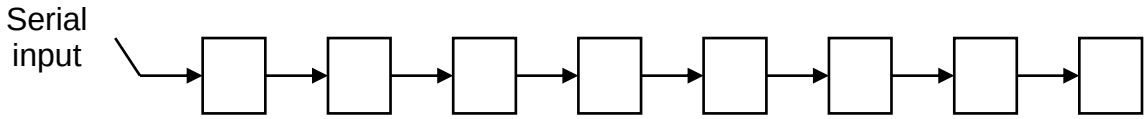
Shift Micro-Operations

There are three types of shifts

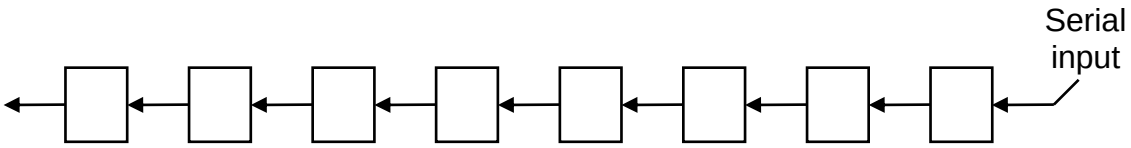
- *Logical shift*
- *Circular shift*

– *Arithmetic shift*

Right shift operation

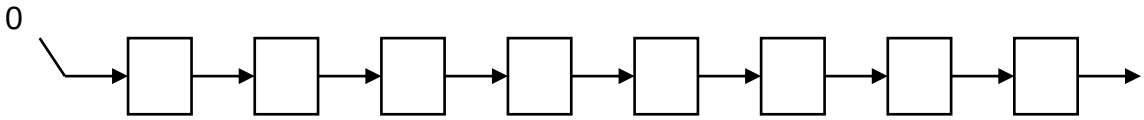


Left shift operation

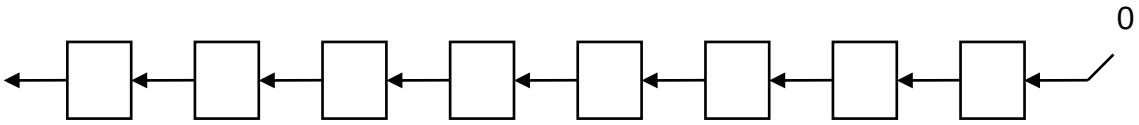


In a logical shift the serial input to the shift is a 0.

Logical right shift operation:



Logical left shift operation



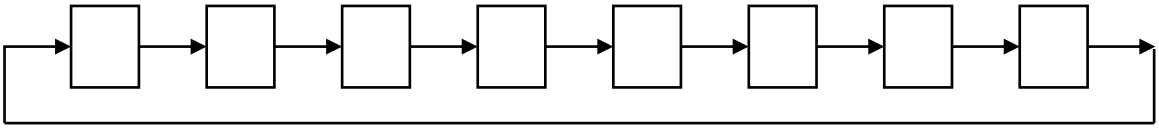
In a Register Transfer Language, the following notation is used

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

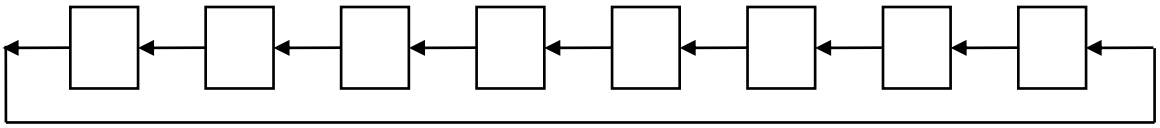
Circular Shift operation

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:



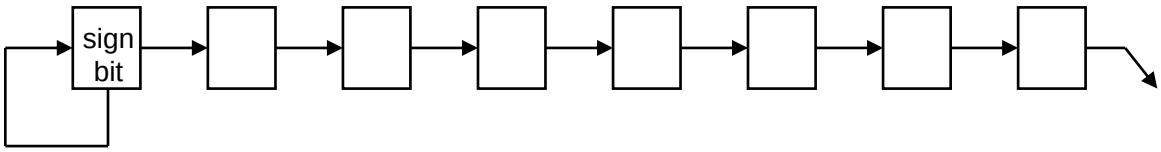
In a RTL, the following notation is used

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

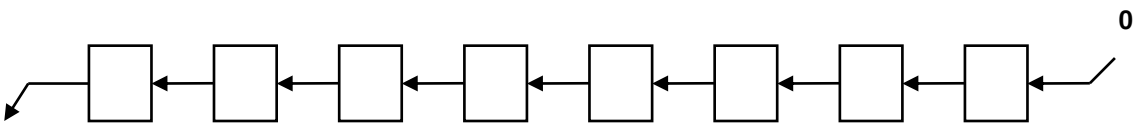
Arithmetic Shift Operation

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

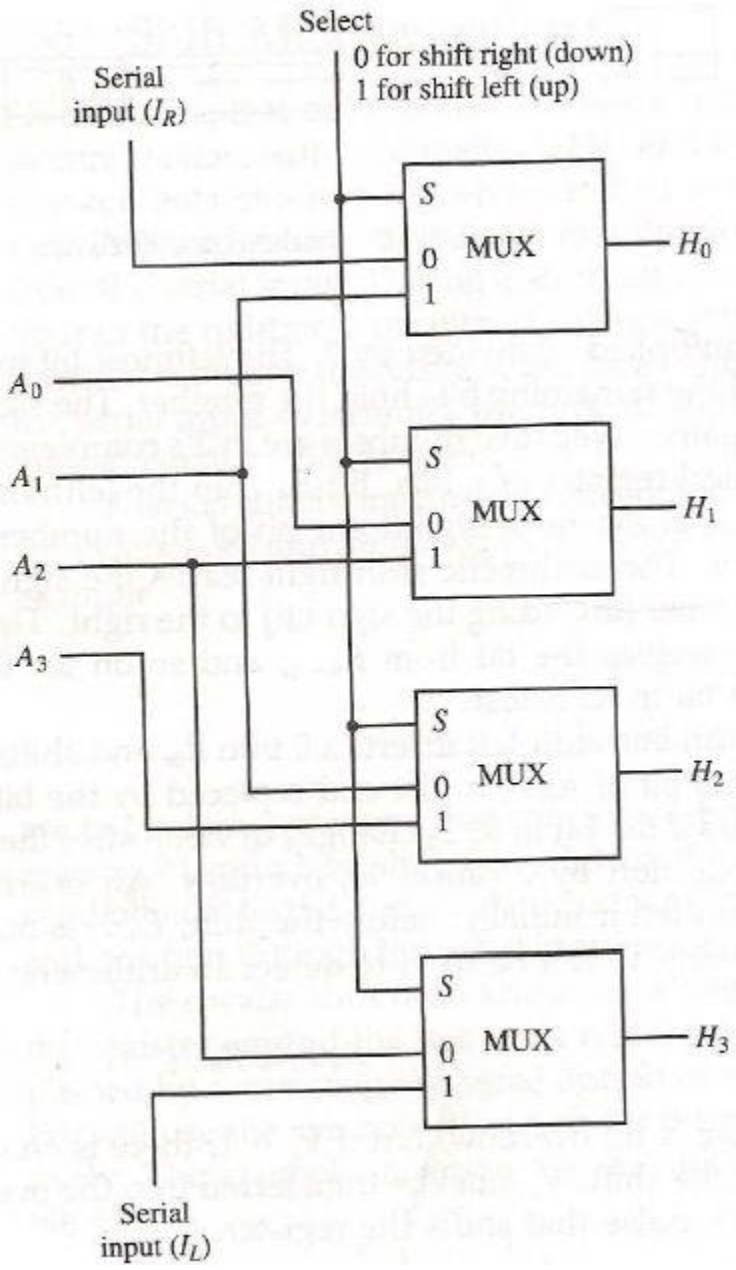


An left arithmetic shift operation must be checked for the overflow

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

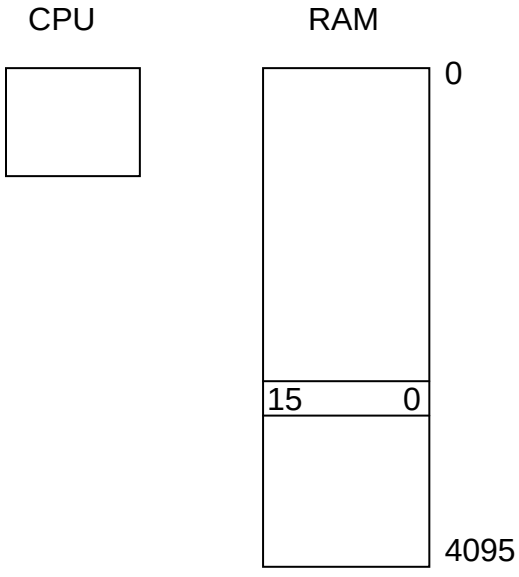
Introduction

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

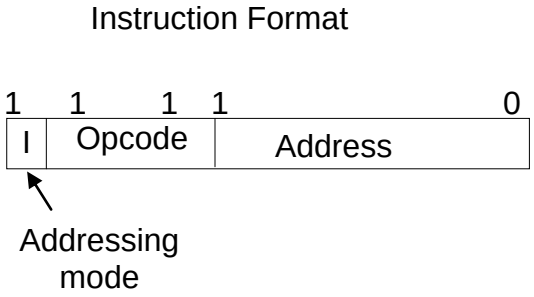
translates the instruction into the sequence of microoperations necessary to implement it

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

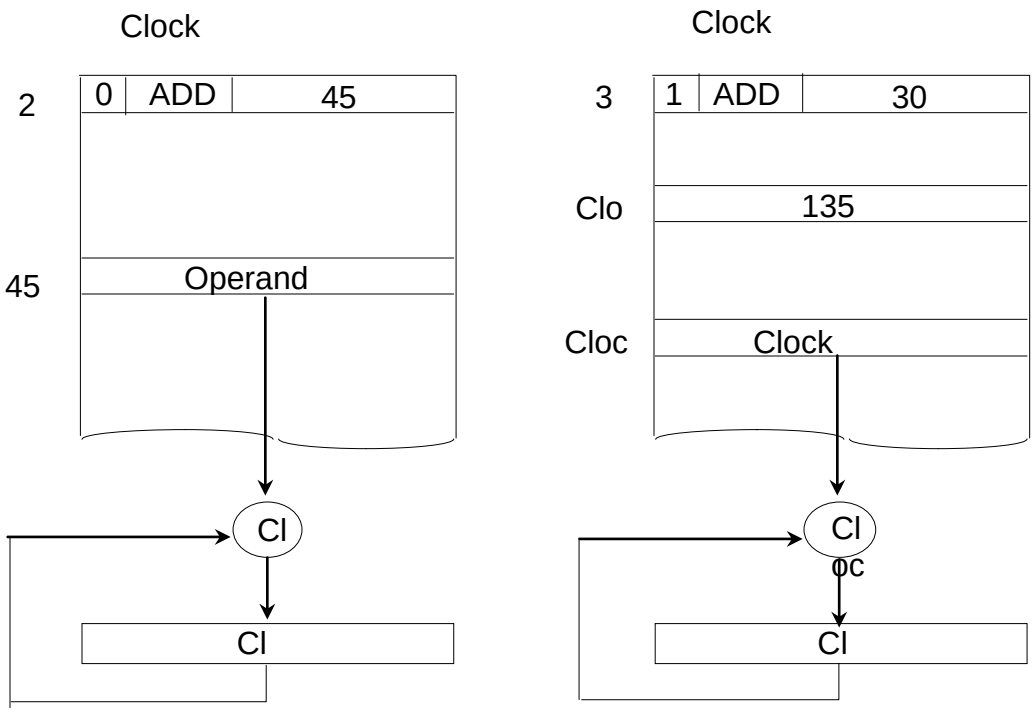
In the Basic Computer, since the memory contains 4096 (= 2^{12}) words, we needs 12 bit to specify the memory address that is used by this instruction. In the Basic Computer, bit 15 of the instruction specifies the *addressing mode* (0: direct addressing, 1: indirect addressing). Since the memory words, 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

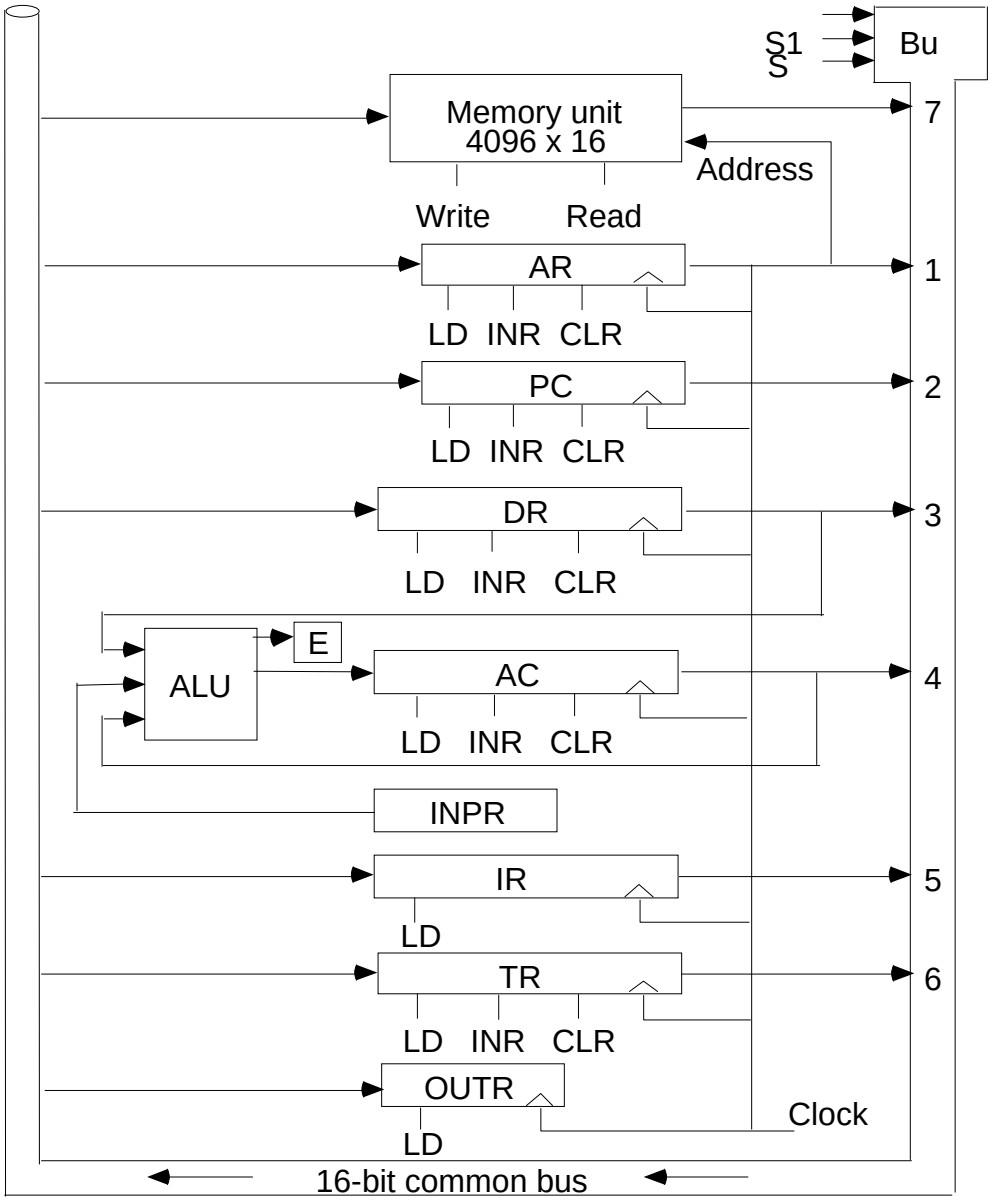
Since the memory in the Basic Computer only has 4096 (=2^{12}) locations, PC and AR only needs 12 bits
Since the word size of Basic Computer only has 16 bit, the DR, AC, IR and TR needs 16 bits.
The Basic Computer uses a very simple model of input/output (I/O) operations

- 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



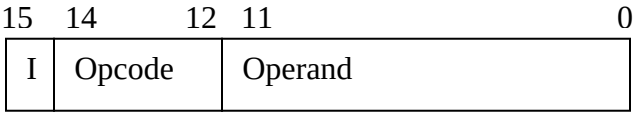
Either one of the registers will have its load signal activated, or the memory will have its read signal activated

- Will determine where the data from the bus gets loaded

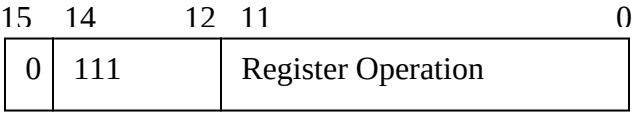
The 12-bit registers, AR and PC, have 0's loaded onto the bus in the high order 4 bit positions. When the 8-bit register OUTR is loaded from the bus, the data comes from the low order 8 bits on the bus

Instruction Formats of Basic Computer

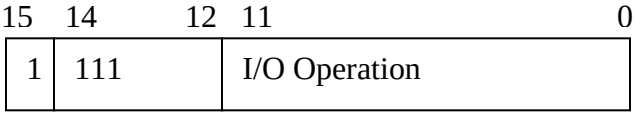
Memory-Reference Instructions (OP-code = 000 ~ 110)



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.

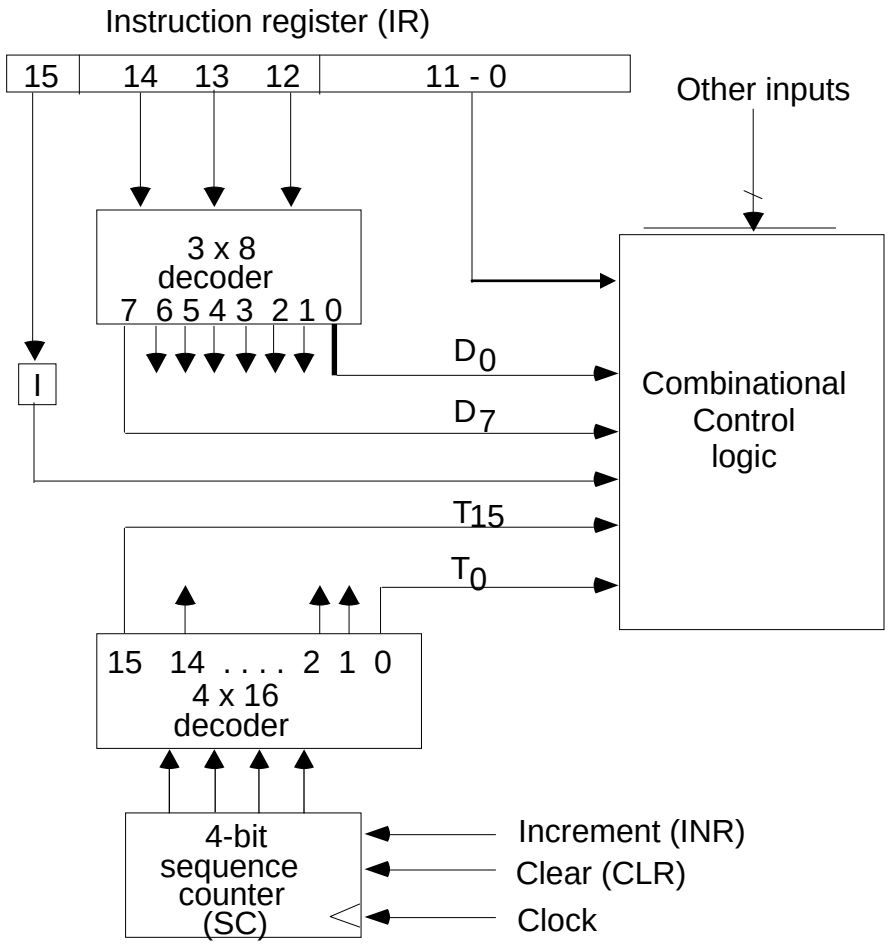
LDA instruction moves data from memory to register and STA instruction moves data from register to memory.
The branch instructions BUN, BSA and ISZ together with skip instruction provide the mechanism of program control and sequencing.
INP instruction is used to read data from input device and OUT instruction 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

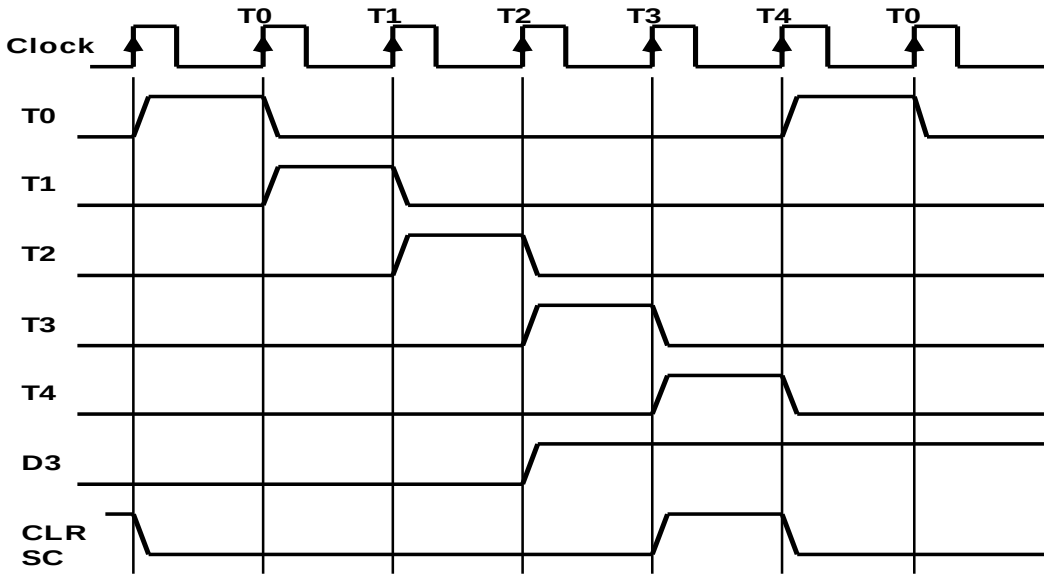
- *Hardwired* Control
  - CU is made up of sequential and combinational circuits to generate the control signals
  - If logic is changed we need to change the whole circuitry
  - Expensive
  - Fast
- *Microprogrammed* Control
  - A control memory on the processor contains microprograms that activate 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 can not 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

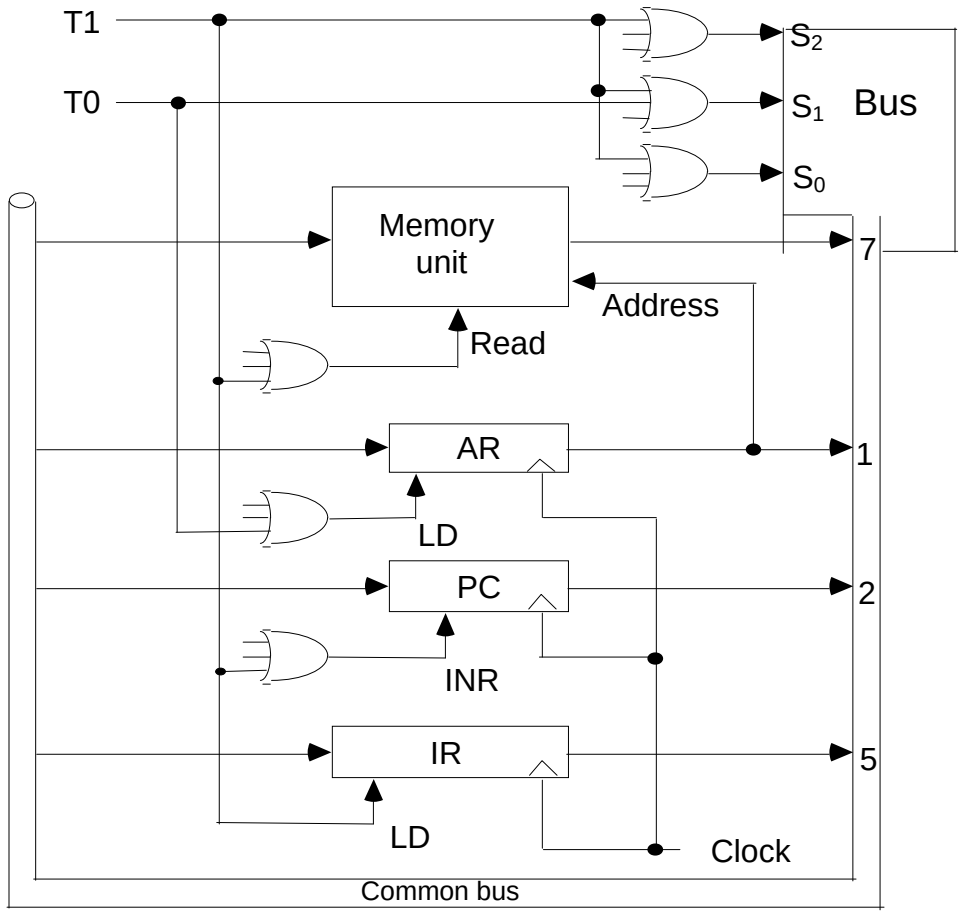
After an instruction is executed, the cycle starts again at step 1, for the next instruction

Fetch and Decode

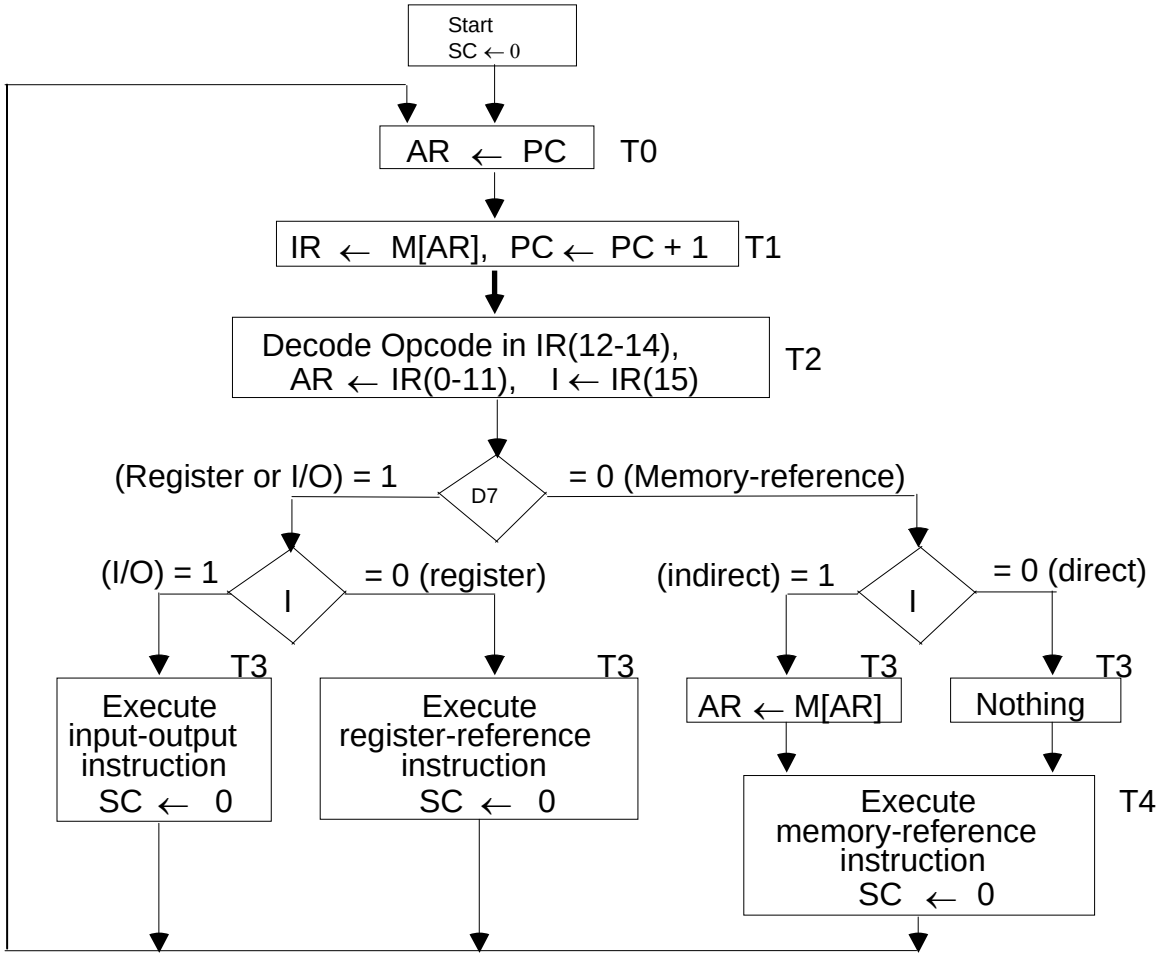
T0: AR ←PC (S0S1S2=010, T0=1)

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
7
Address
Read
AR
1
LD
PC
2
INR
IR
5
LD
Clock
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

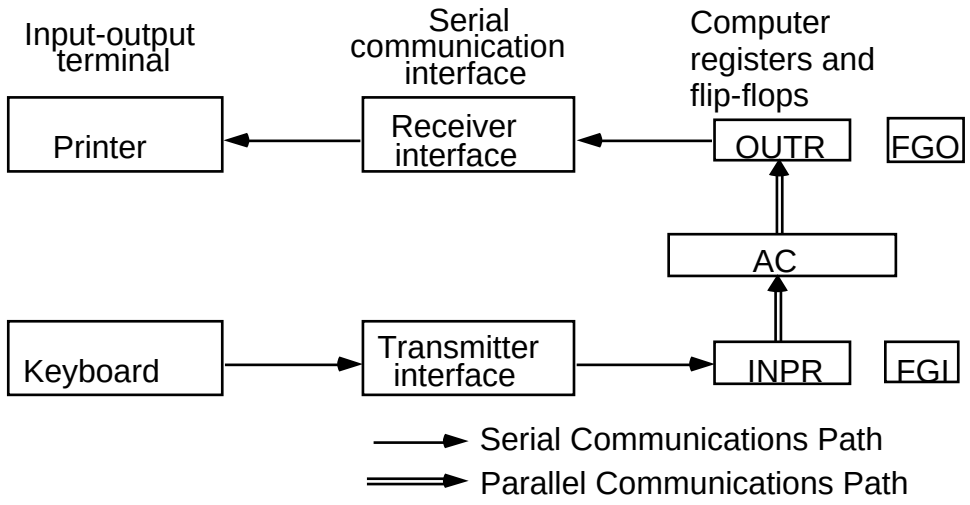
D_6T_4: DR ← M[AR] //Load data into DR

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

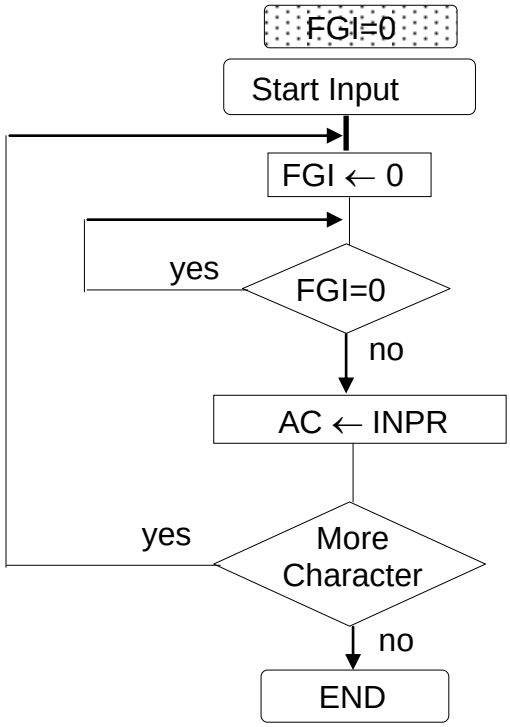
- The serial info. from the keyboard is shifted into INPR
- The serial info. for the printer is stored in the OUTR
- INPR and OUTR communicate with the terminal serially and with the AC in parallel.

- 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

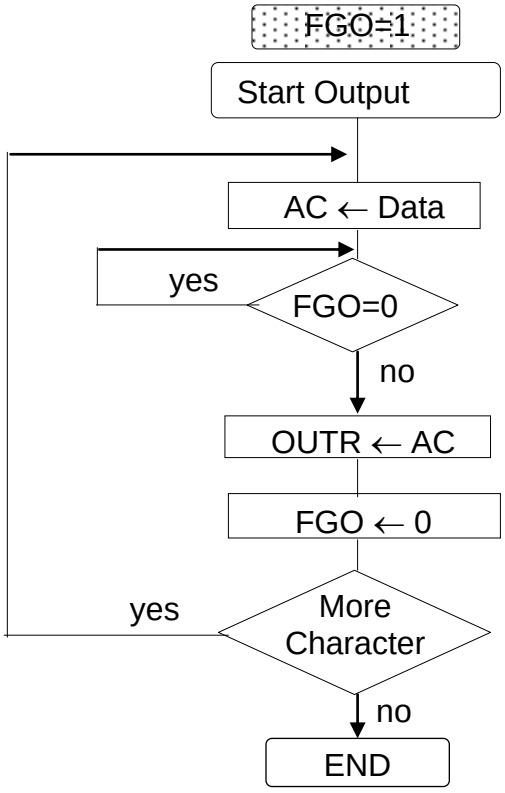
Input Device:
loop: If FGI = 1 goto loop
INPR ← new data, FGI ← 1

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

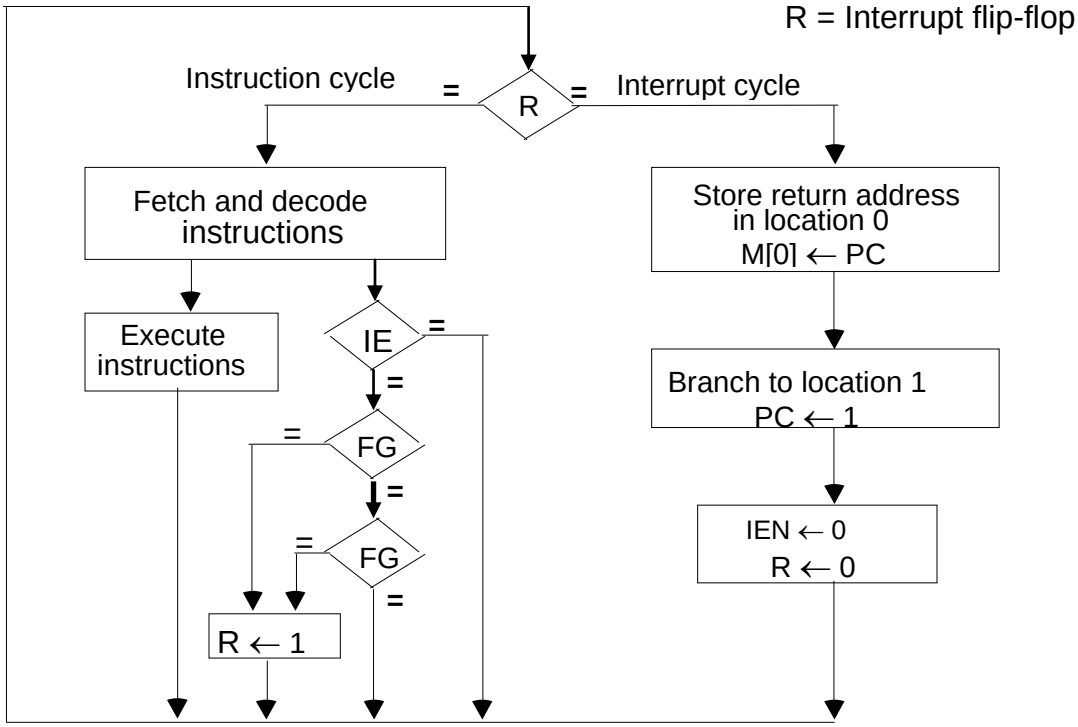
Let

$D_7IT_3 = p$

IR(i) = Bi, i = 6, …, 11

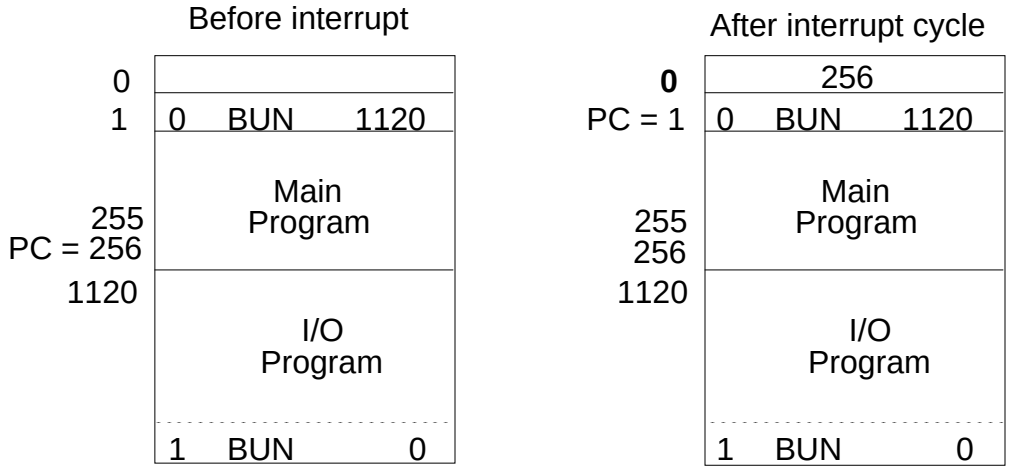
INP	pB_{11}:	AC(0-7) ← INPR, FGI ← 0, SC ← 0	Input char. to AC
OUT	pB_{10}:	OUTR ← AC(0-7), FGO ← 0, SC ← 0	Output char. from AC
SKI	pB_9:	if(FGI = 1) then (PC ← PC + 1), SC ← 0	Skip on input flag
SKO	pB_8:	if(FGO = 1) then (PC ← PC + 1), SC ← 0	Skip on output flag
ION	pB_7:	IEN ← 1, SC ← 0	Interrupt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

Interrupt Cycle

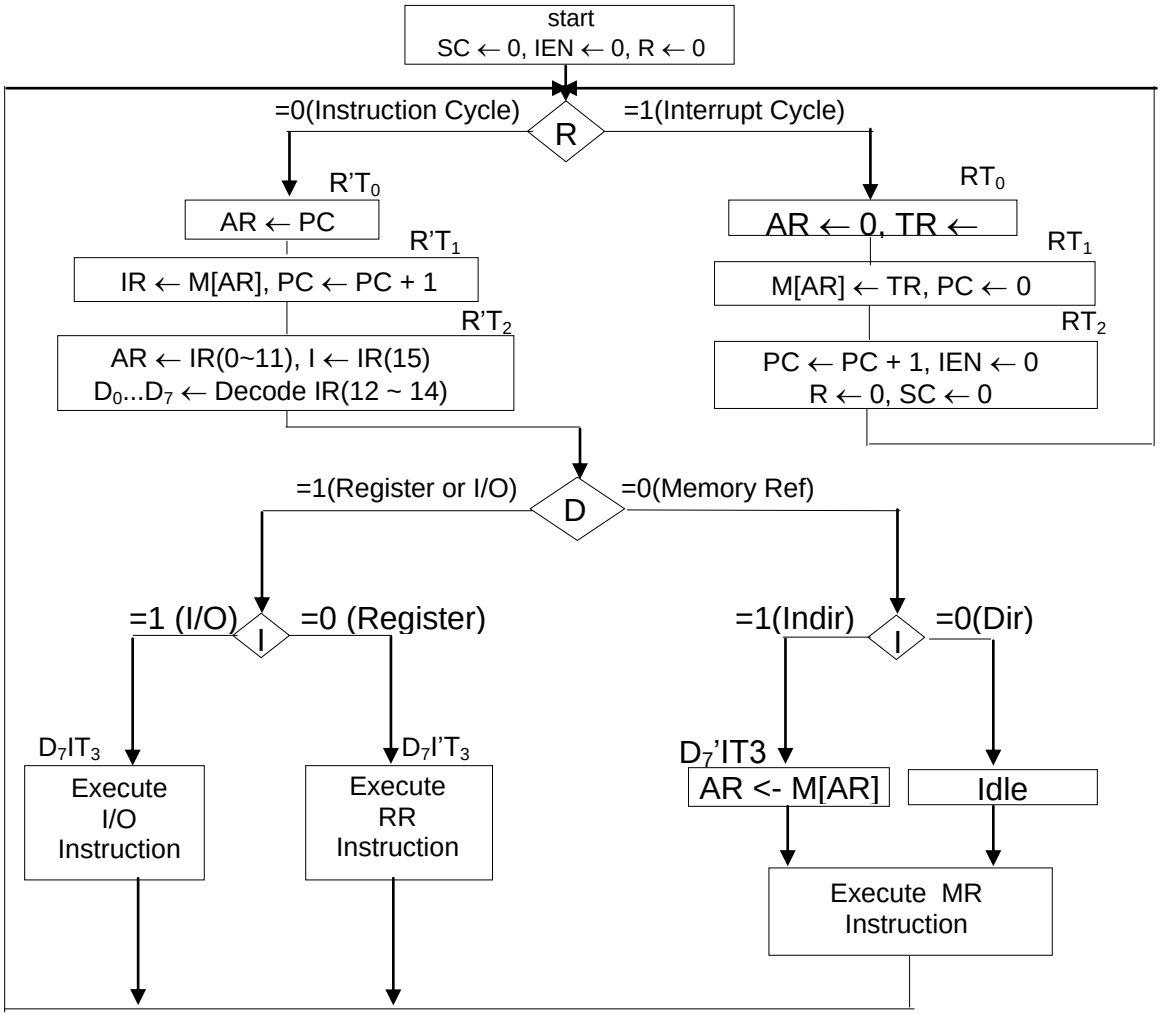


The interrupt cycle is a HW implementation of a branch and save return address operation.

- 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

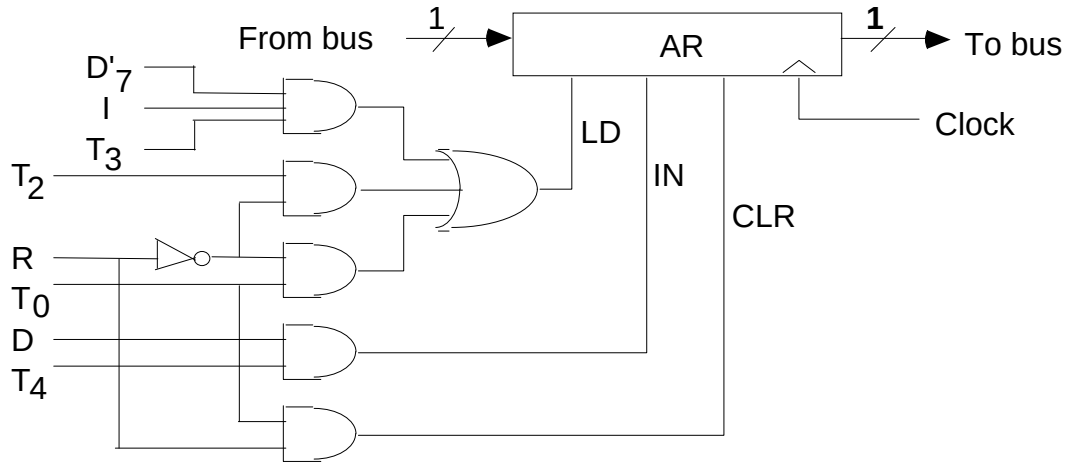
Control of AR register

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,

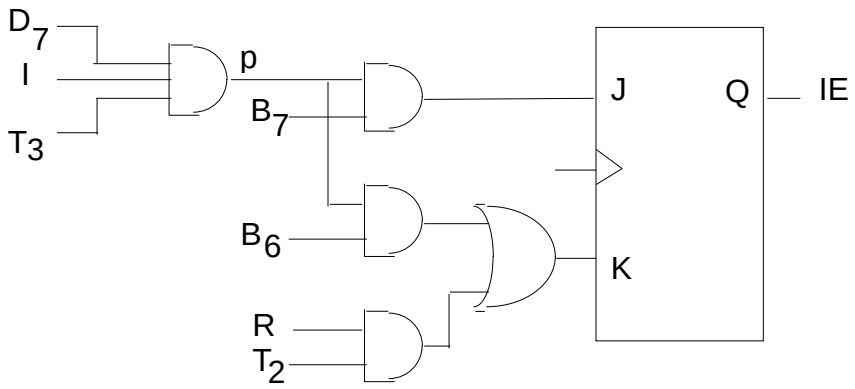
$LD(AR) = R'T_0 + R'T_2 + D'_7IT_3$
$CLR(AR) = RT_0$
$INR(AR) = D_5T_4$



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)



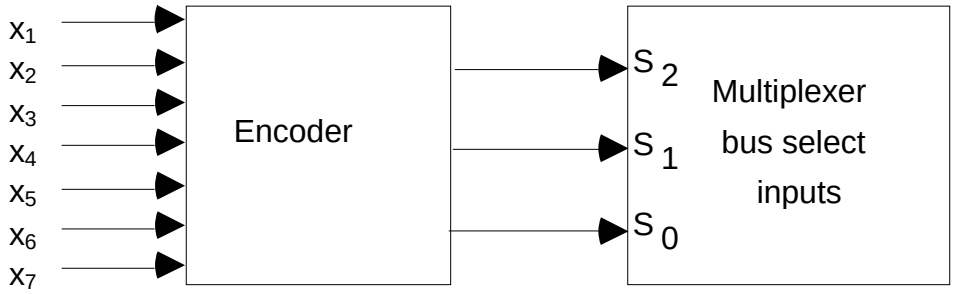
Control of Common Bus

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

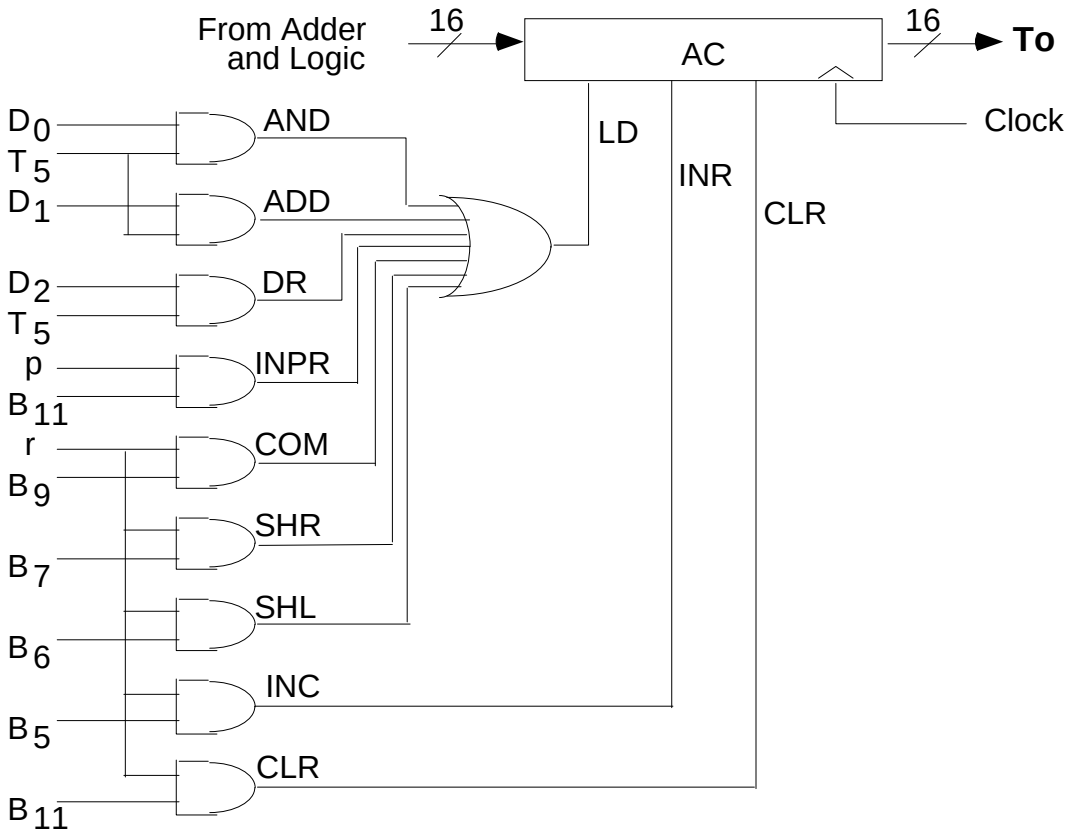
D_0T_5:	$AC \leftarrow AC \wedge DR$	AND with DR
D_1T_5:	$AC \leftarrow AC + DR$	Add with DR
D_2T_5:	$AC \leftarrow DR$	Transfer from DR
pB_{11}:	$AC(0\text{-}7) \leftarrow INPR$	Transfer from INPR
rB_9:	$AC \leftarrow AC'$	Complement
rB_7 :	$AC \leftarrow shr\ AC, AC(15) \leftarrow E$	Shift right
rB_6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

=>

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

$CLR(AC) = rB_{11}$

$INR(AC) = rB_5$


From Adder
and Logic
16
AC
16
To
Clock
LD
INR
CLR
D0
T5
AND
D1
ADD
D2
T5
DR
p
B11
INPR
r
B9
COM
B7
SHR
B6
SHL
B5
INC
B11
CLR

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.

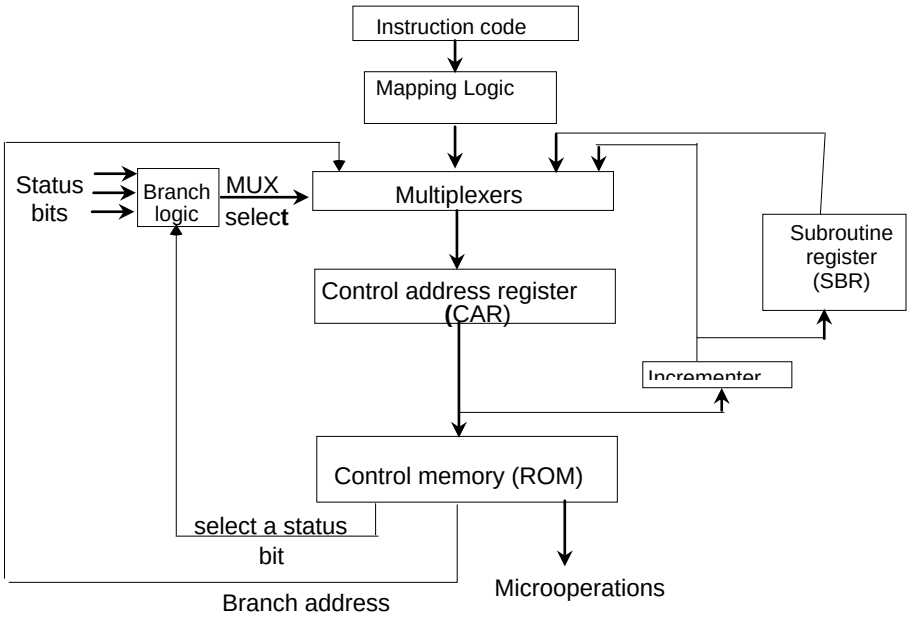


Fig: Block diagram of address sequencer.

- ✓ 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

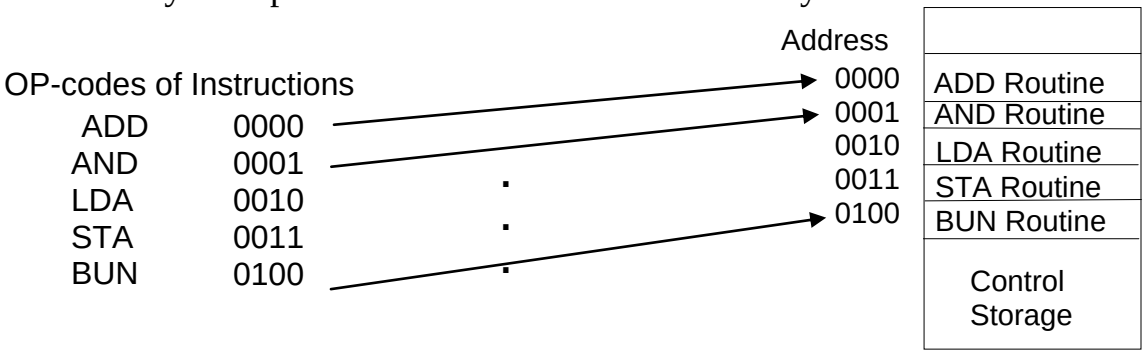
Fix the value of one status bit at the input of the multiplexer to 1. So that always branching is done

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

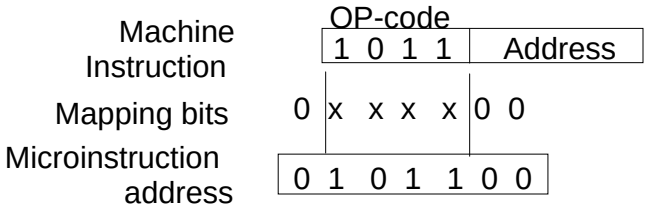
Direct mapping:

Directly use opcode as address of Control memory



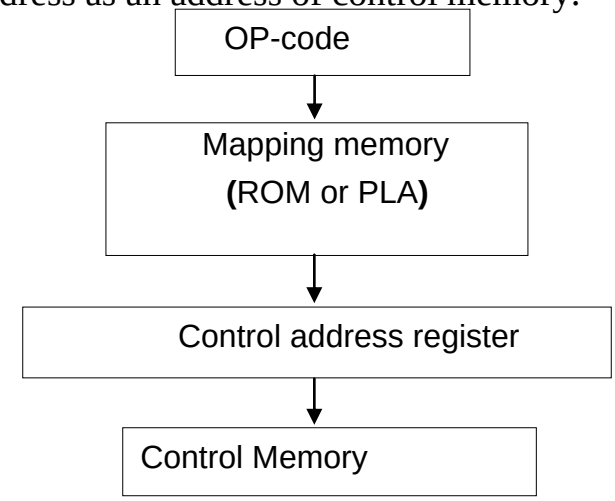
Another approach of mapping:

Modify Opcode to use it as an address of control memory

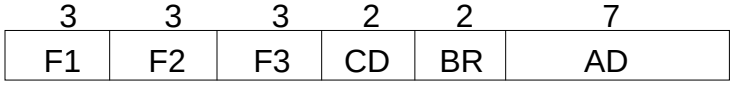


Mapping function implemented by ROM or PLA

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

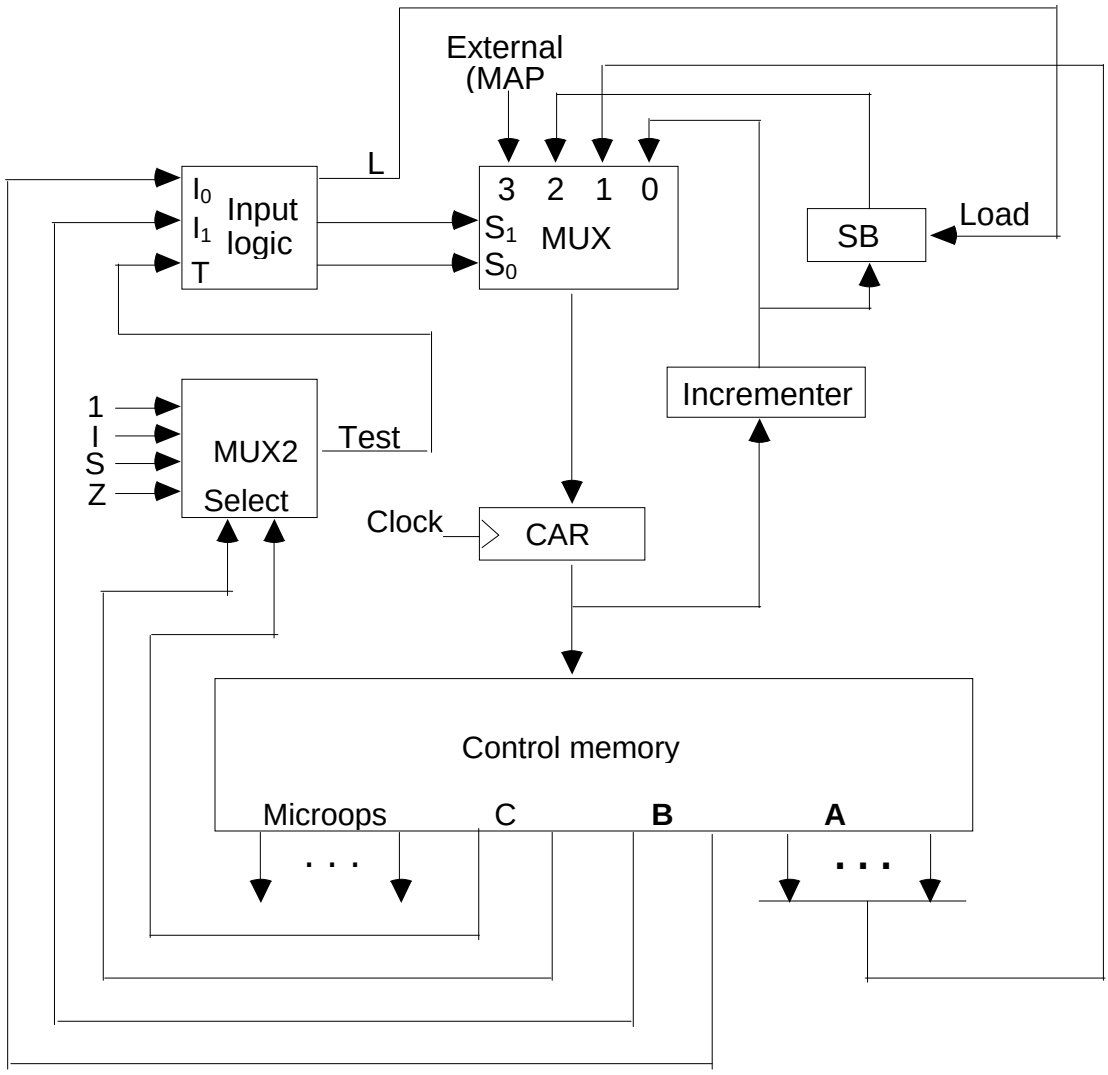
Symbolic microprogram for the fetch cycle:

```
        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	0000000

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

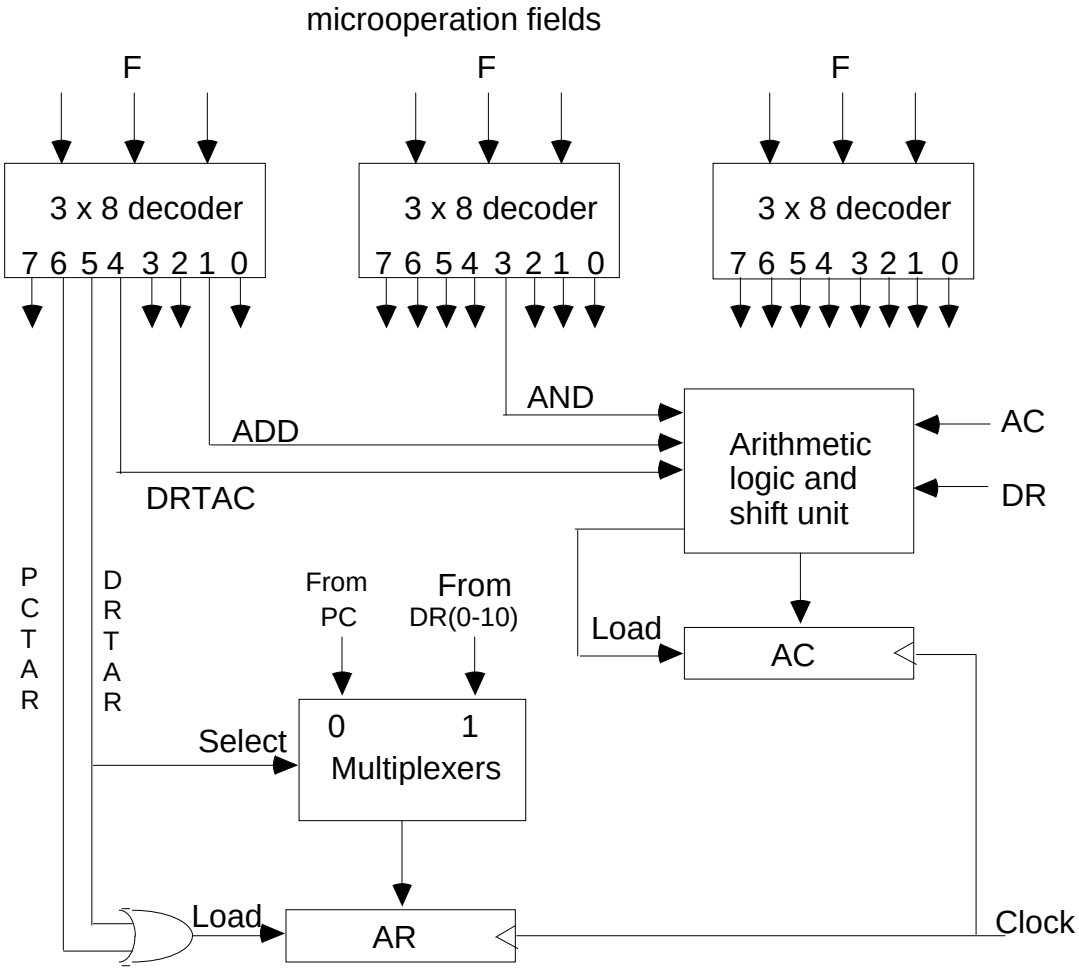
I_0I_1T	Meaning	Source of Address	S_1S_0	L
000	In-Line	CAR+1	00	0
001	JMP	CS(AD)	10	0
010	In-Line	CAR+1	00	0
011	CALL	CS(AD) and SBR <- CAR+1	10	1
10x	RET	SBR	01	0
11x	MAP	DR(11-14)	11	0

$S_0 = I_0$

$S_1 = I_0I_1 + I0'T$

$L = I_0'I_1T$

F Field Decoding



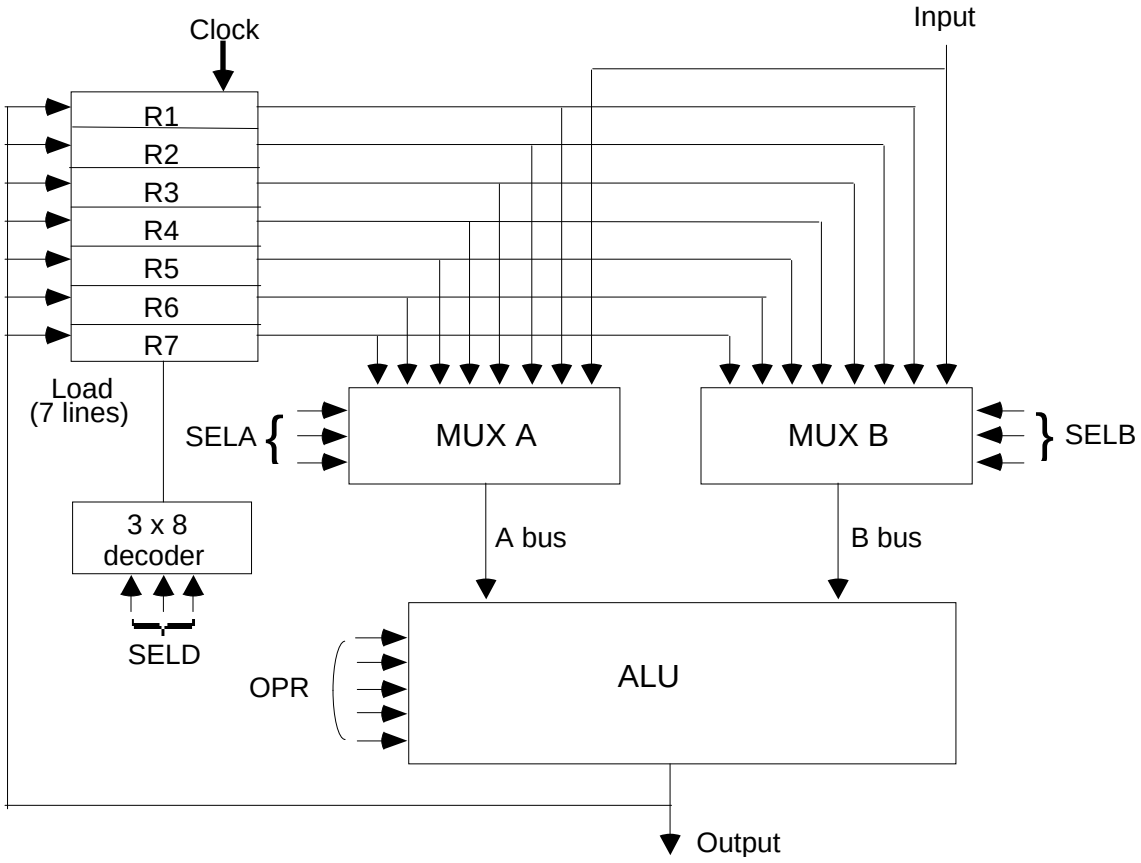
Since there are three microoperation fields we need 3 decoders. Only some of the outputs of decoders are shown to be connected to their output. Each of the output of the decoders must be connected to the proper circuit to initiate the corresponding microoperation. For example when F1=101 the next clock 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).

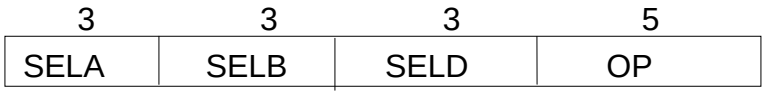
Example: R1 ← R2 + R3

- 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

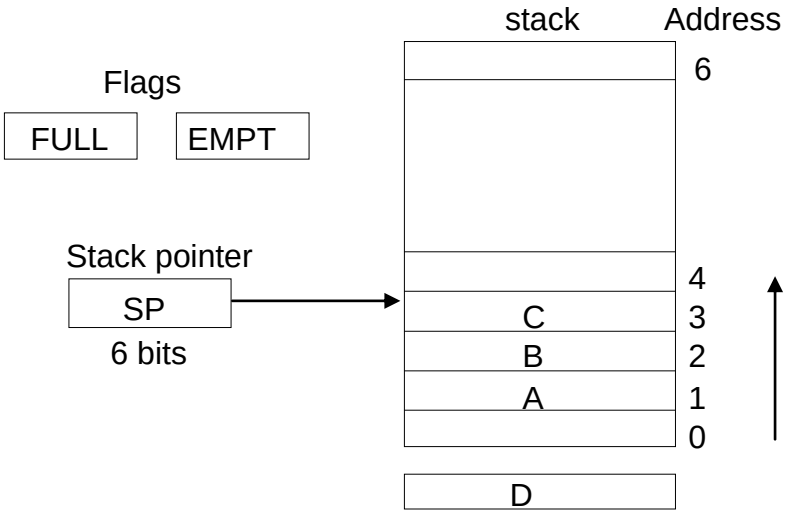


Examples of Microoperations for CPU

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

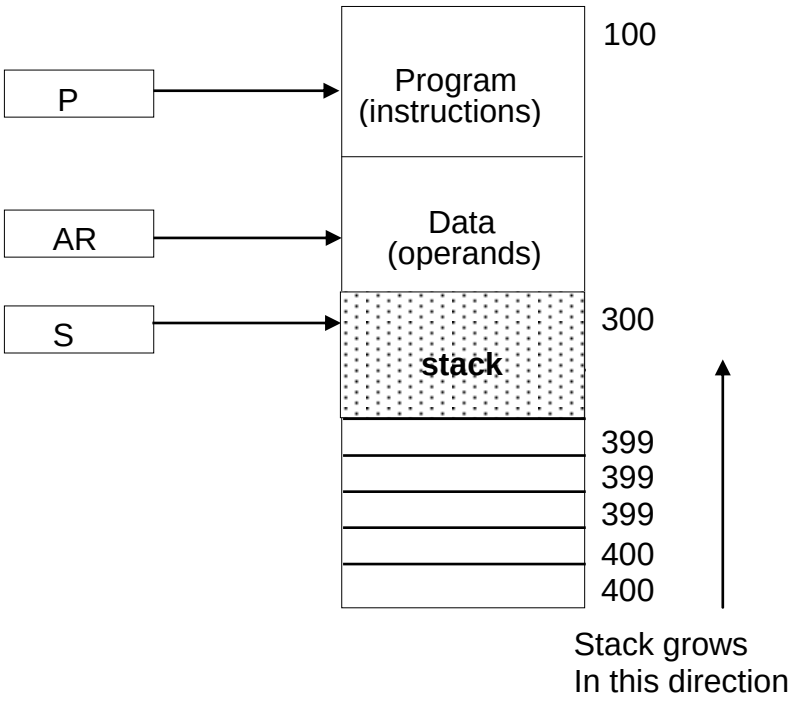
It is the collection of finite number of registers. Stack pointer (SP) points to the register that is currently at the top of stack.



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

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

  ```
  PUSH X          // 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

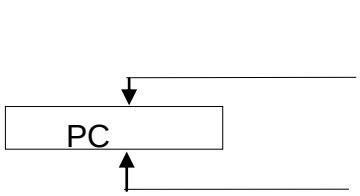
  - Logical and bit manipulation instructions
    Some examples:

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

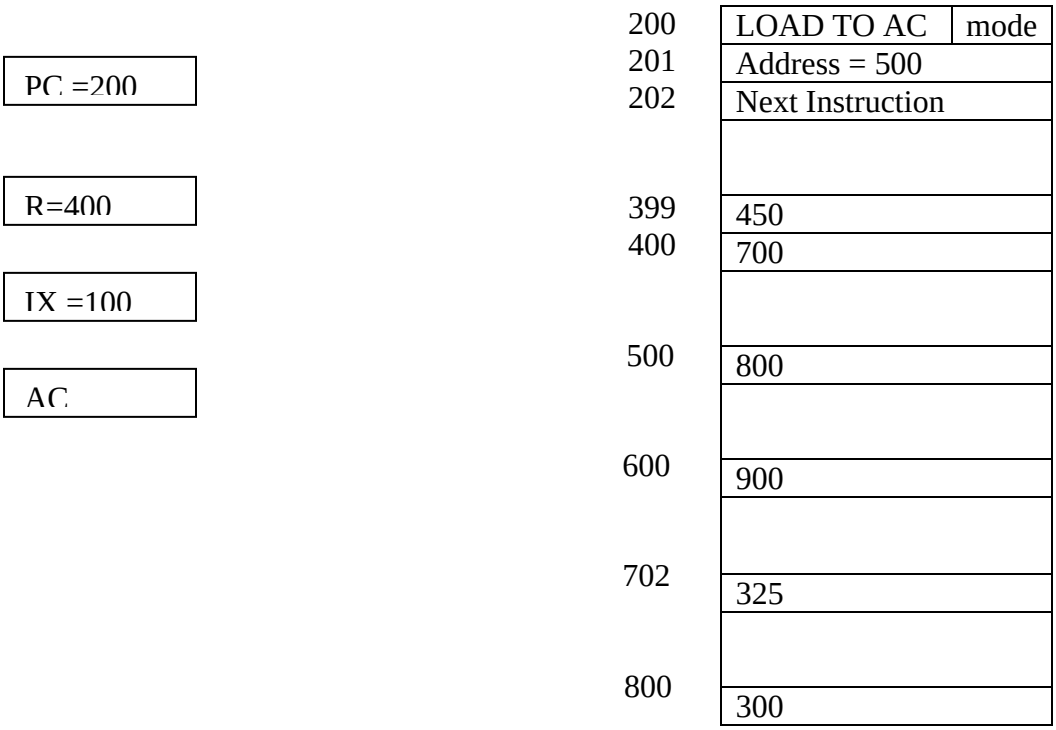
- **Register Indirect Mode**
  Instruction specifies a register which contains the memory address of the operand
  – Saving instruction bits since register address is shorter than the memory address
  – Slower to acquire an operand than both the register addressing or memory addressing
  – EA= content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

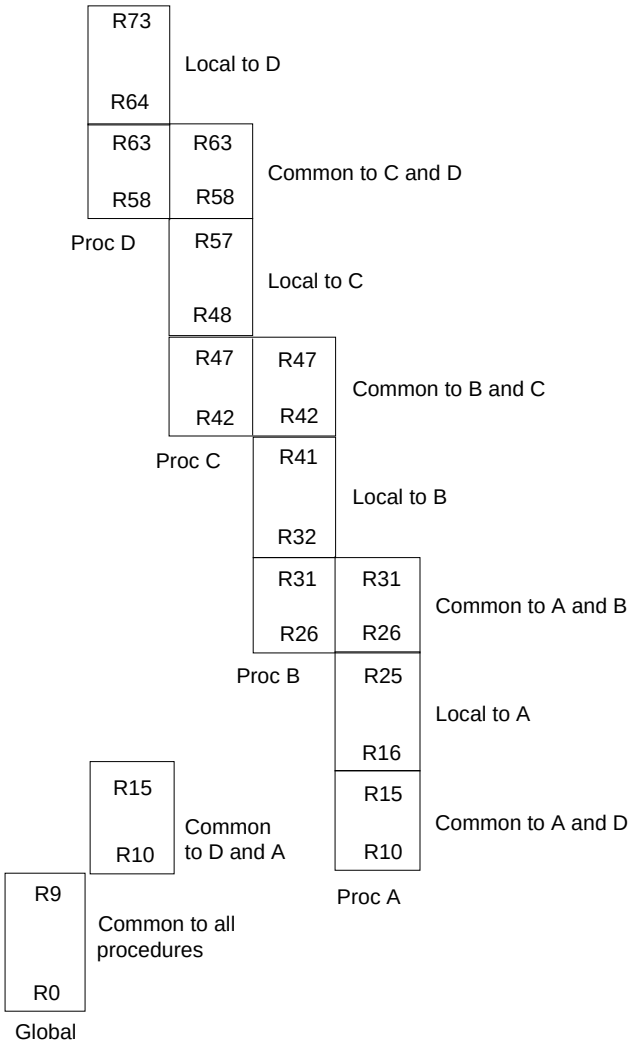
Reduced Instruction Set Computer (RISC)

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

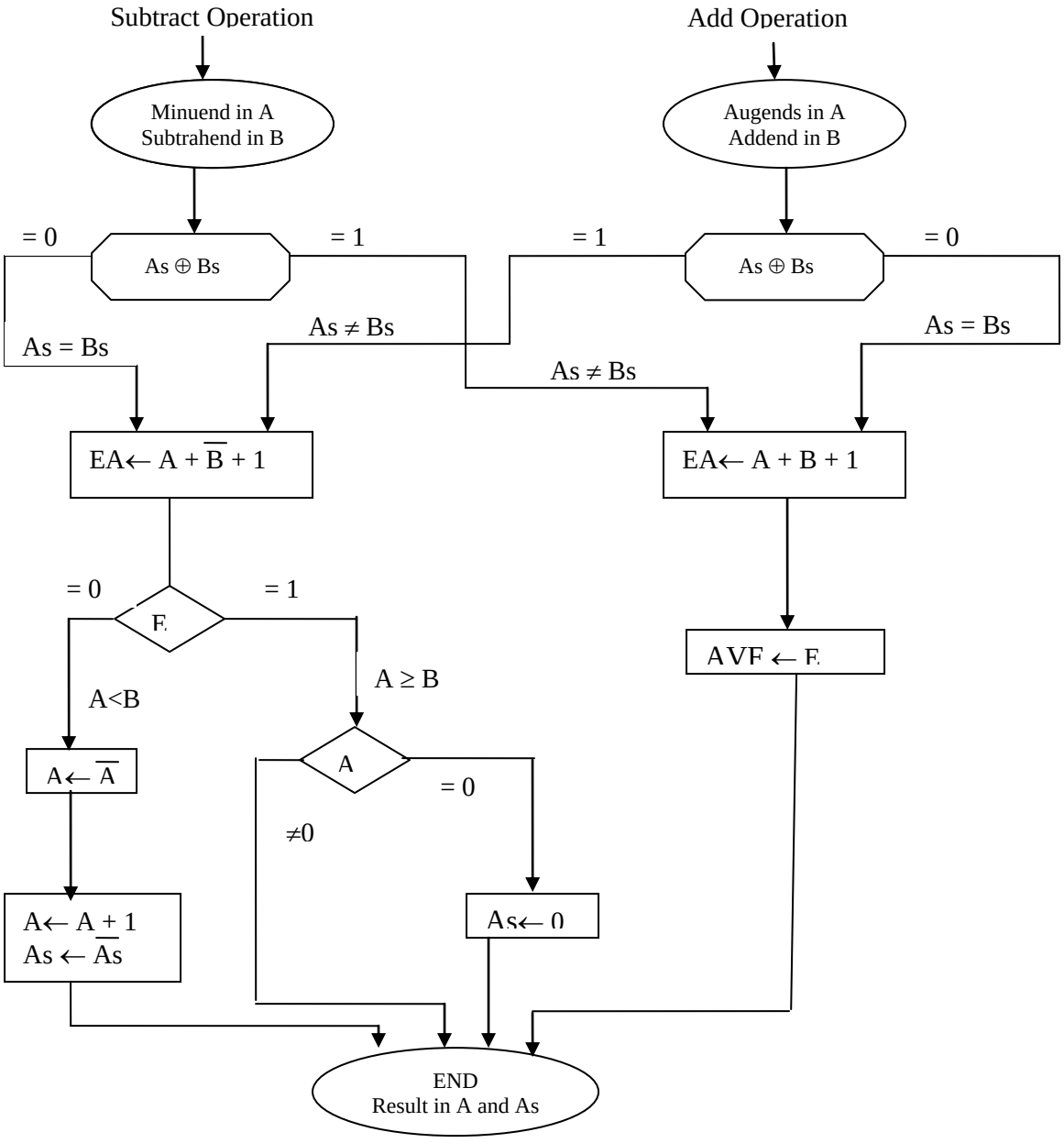
Chapter 6
Computer Arithmetic

Addition and Subtraction

Introduction

There are three ways of representing negative fixed-point binary numbers: signed-magnitude, signed-i's complement, or signed-2's complement. Most computers use the signed-2's complement representation when performing arithmetic operations with integers. For floating-point operations, most computers use the signed-magnitude representation for the mantissa. In this section we develop the addition and subtraction algorithms for data represented in signed-magnitude and again for data represented in signed-2's complement. It is important to realize that the adopted representation for negative numbers refers to the representation of numbers in the registers before and after the execution of the arithmetic operation. It does not mean that complement arithmetic may not be used in an intermediate step. For example, it is convenient to employ complement arithmetic when performing a subtraction operation with numbers in signed-magnitude representation. As long as the initial minuend and subtrahend, as well as the final difference, are in signed-magnitude form the fact that complements have been used in an intermediate step does not alter the fact that the representation is in signed-magnitude.

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)

Hardware Implementation

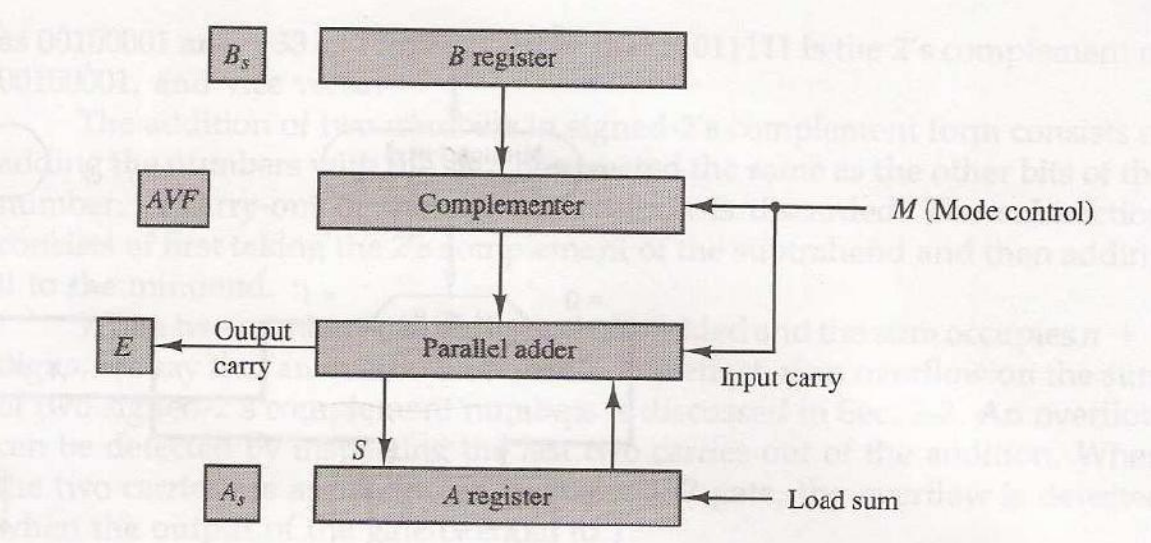


Fig. Hardware for signed magnitude addition and subtraction

To implement the two arithmetic operations with hardware, it is first necessary that the two numbers be stored in registers. Let A and B be two registers that hold the magnitudes of the numbers, and A_5 and B_5 be two flip-flops that hold the corresponding signs. The result of the operation may be transferred to a third register: however, a saving is achieved if the result is transferred into A and A_5. Thus A and A_5 together form an accumulator register.

Consider now the hardware implementation of the algorithms above. First, a parallel-adder is needed to perform the microoperation A + B. Second, a comparator circuit is needed to establish if A > B, A = B, or A < B. Third, two parallel-subtractor circuits are needed to perform the microoperations A - B and B - A. The sign relationship can be determined from an exclusive-OR gate with A_5 and B_5 as inputs.

This procedure requires a magnitude comparator, an adder, and two subtractors. However, a different procedure can be found that requires less equipment. First, we know that subtraction can be accomplished by means of complement and add. Second, the result of a comparison can be determined from the end carry after the subtraction. Careful investigation of the

alternatives reveals that the use of 2's complement for subtraction and comparison is an efficient procedure that requires only an adder and a complementer.

Figure above shows a block diagram of the hardware for implementing the addition and subtraction operations. It consists of registers A and B and sign flip-flops A_5 and B_5. Subtraction is done by adding A to the 2's complement of B. The output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitudes of the two numbers. The add-overflow flip-flop *AVF* holds the overflow bit when A and B are added. The A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

The addition of A plus B is done through the parallel adder. The S (sum) output of the adder is applied to the input of the A register. The complementer provides an output of B or the complement of B depending on the state of the mode control M. The complementer consists of exclusive-OR gates and the parallel adder consists of full-adder circuits as shown in Fig. 4-7 in Chap. 4. The M signal is also applied to the input carry of the adder. When M = 0, the output of B is transferred to the adder, the input carry is 0, and the output of the adder is equal to the sum A+B. When M = 1, the l's complement of B is applied to the adder, the input carry is 1, and output S = A + B + 1. This is equal to A plus the 2's complement of B, which is equivalent to the subtraction A - B.

Hardware Algorithm

The flowchart for the hardware algorithm is presented in Fig. 10-2. The two signs A, and B, are compared by an exclusive-OR gate. If the output of the gate is 0, the signs are identical; if it is 1, the signs are different. For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added. The magnitudes are added with a microoperation EA ←A + B, where EA is a register that com-bines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip-flop AVF. The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complement of B . No overflow can occur if the numbers are subtracted so AVF is cleared to 0. A 1

in E indicates that A > B and the number in A is the correct result. If this number is zero, the sign A must be made positive to avoid a negative zero. A 0 in E indicates that A < B. For this case it is necessary to take the 2's complement of the value in A. This operation can be done with one microoperation A (—A + 1. However, we assume that the A register has circuits for microoperations complement and increment, so the 2's complement is obtained from these two microoperations. In other paths of the flowchart, the sign of the result is the same as the sign of A, so no change in A, is required. However, when A < B, the sign of the result is the comple-ment of the original sign of A . It is then necessary to complement A, to obtain the correct sign. The final result is found in register A and its sign in A_s. The value in AVF provides an overflow indication. The final value of E is immaterial.

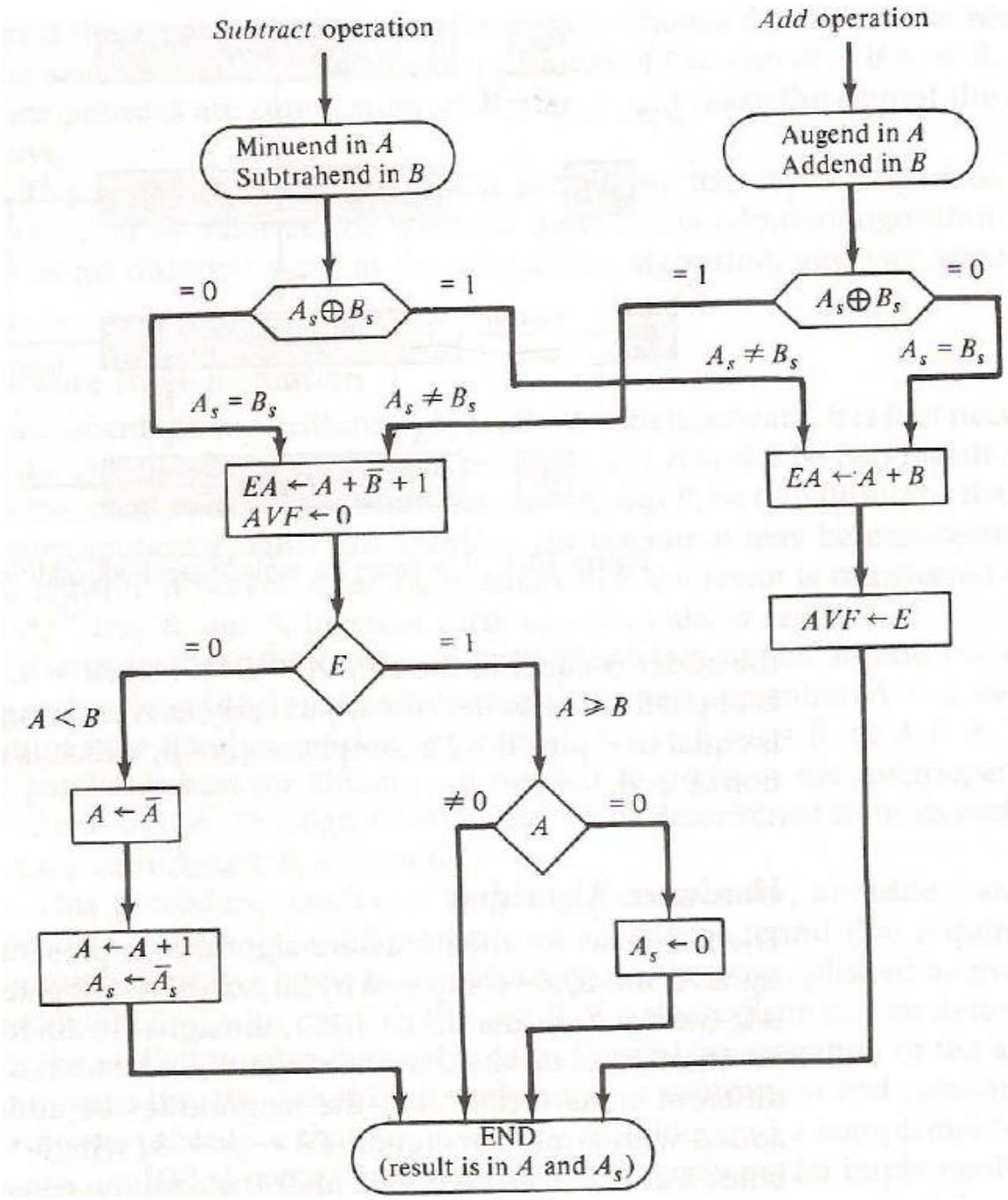
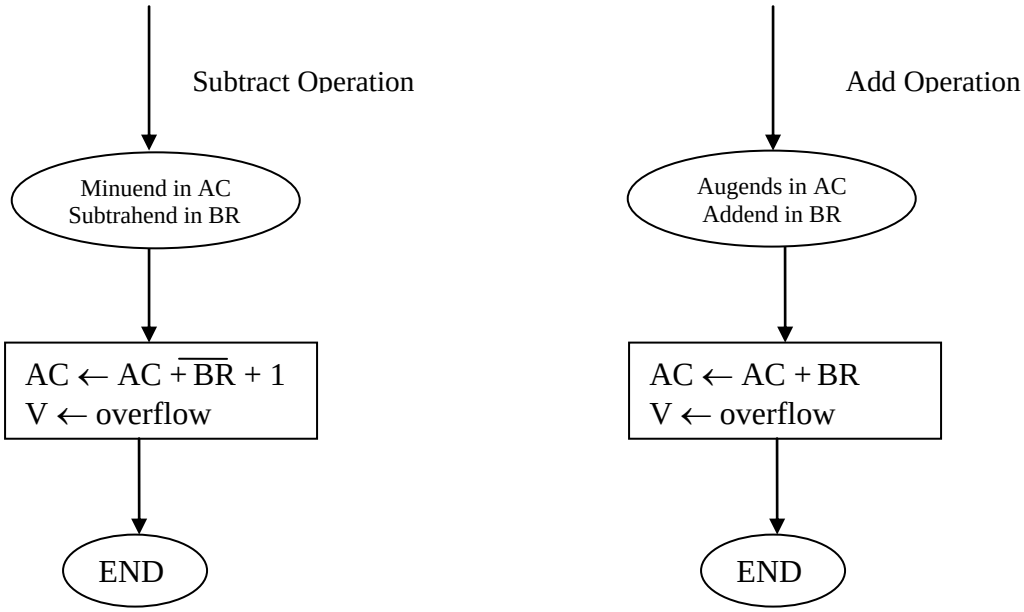


Fig. Flowchart for add and subtract operations

Addition and Subtraction with Signed 2's Complement Data



Example:

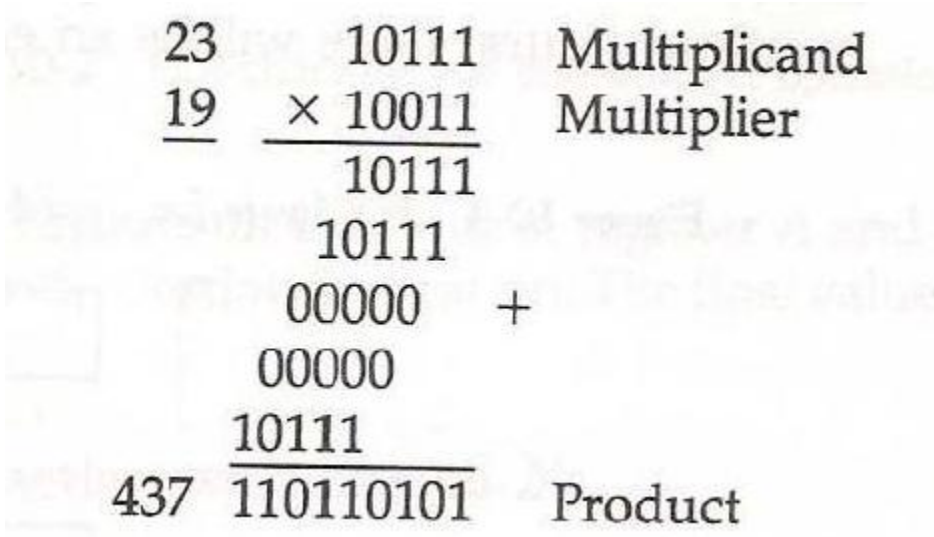
33 + (-35)

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

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

Multiplication

Multiplication of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive shift and adds operations. This process is best illustrated with a numerical example.



The process consists of looking at successive bits of the multiplier, least significant bit first. If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product.

Hardware Implementation for signed magnitude data

When multiplication is implemented in a digital computer, it is convenient to change the process slightly. First, instead of providing registers to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers and successively accumulate the partial products in a register. Second, instead of shifting the multiplicand to the left, the partial product is shifted to the right, which results in leaving the partial product and the multiplicand in the required relative positions. Third, when the corresponding bit of the multiplier is 0, there is no need to add all zeros to the partial product since it will not alter its value.

The hardware for multiplication consists of the equipment shown in Fig. 10-1 plus two more registers. These registers together with registers A and B are shown in Fig. 10-5. The multiplier is stored in the Q register and its sign in Qs. The sequence counter SC is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When the content of the counter reaches zero, the product is formed and the process stops.

Initially, the multiplicand is in register B and the multiplier in Q. The sum of A and B forms a partial product which is transferred to the EA register. Both partial product and multiplier are shifted to the right. This shift will be denoted by the statement shr EAQ to designate the right shift depicted in Fig. 10-5. The least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A, and 0 is shifted into E. After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right. In this manner, the rightmost flip-flop in register Q, designated by Q_n, will hold the bit of the multiplier, which must be inspected next.

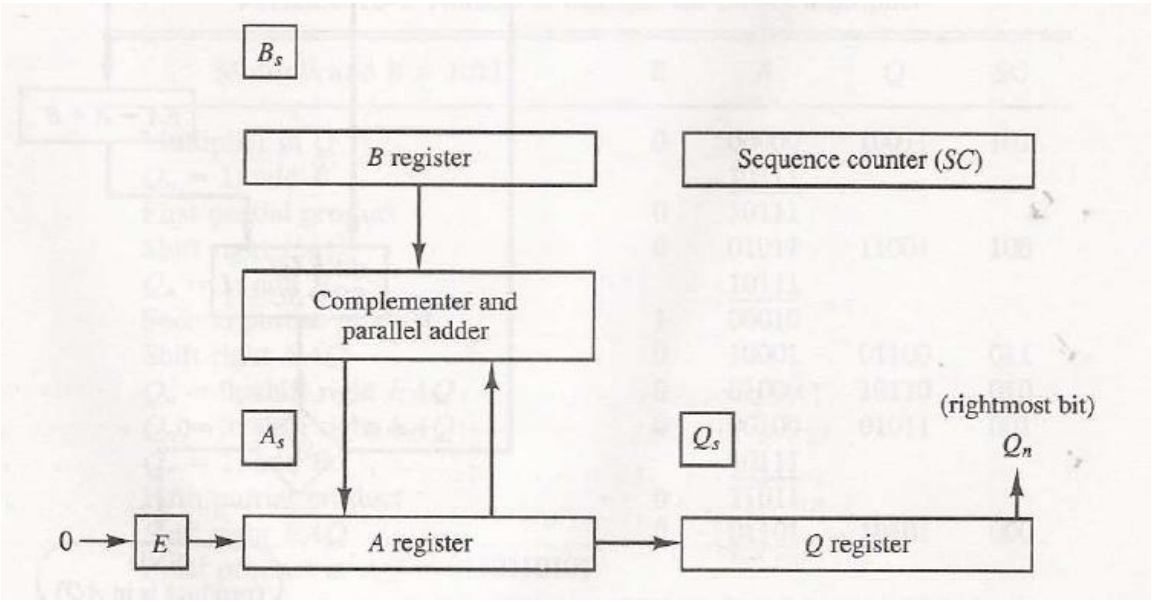


Fig. Hardware for Multiply Operation

Hardware Algorithm

Figure below is a flowchart of the hardware multiply algorithm. Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B_s and Q_s, respectively. The signs are compared, and both A and Q are set to correspond to the sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to a number equal to the number of bits of the multiplier. We are assuming here that operands are transferred to registers from a memory unit that has words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n - 1 bits.

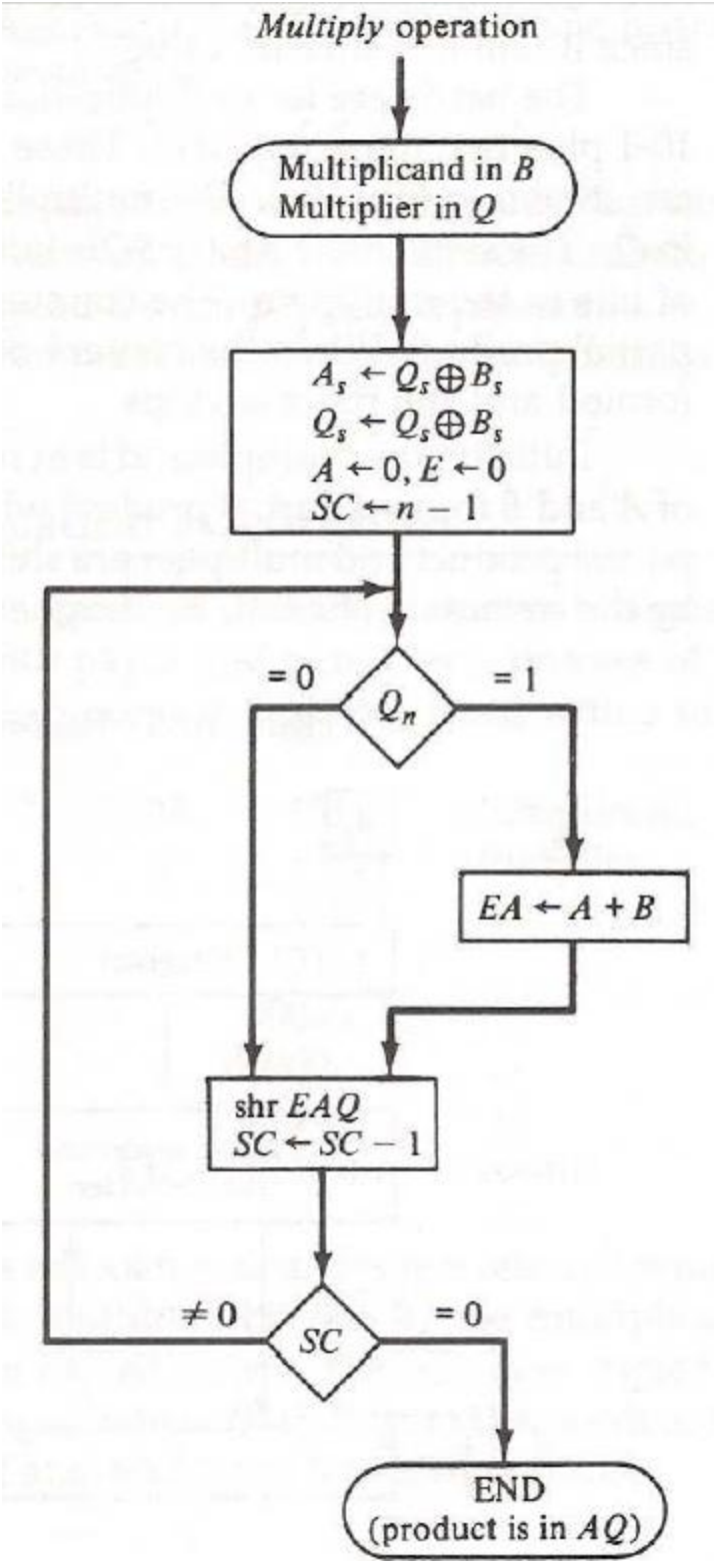


Fig. Flowchart for multiply operation

After the initialization, the low-order bit of the multiplier in Qin is tested. If it is a 1, the multiplicand in B is added to the present partial product in A. If it is a 0, nothing is done. Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. The process stops when SC = 0. Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

The previous numerical example is repeated in Table 10-2 to clarify the hardware multiplication process. The procedure follows the steps outlined in the flowchart.

Booth Multiplication Algorithm

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

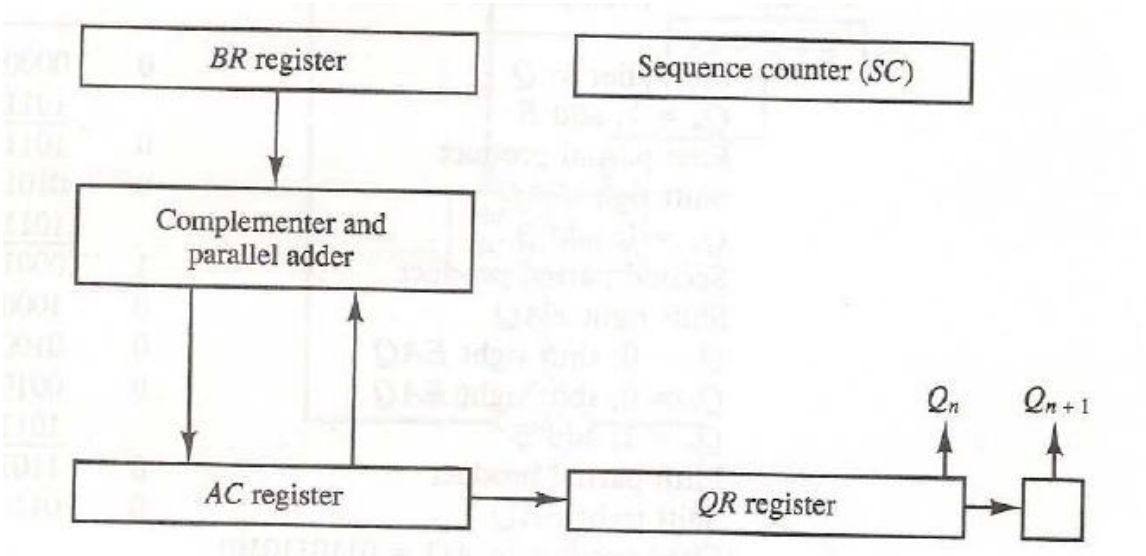
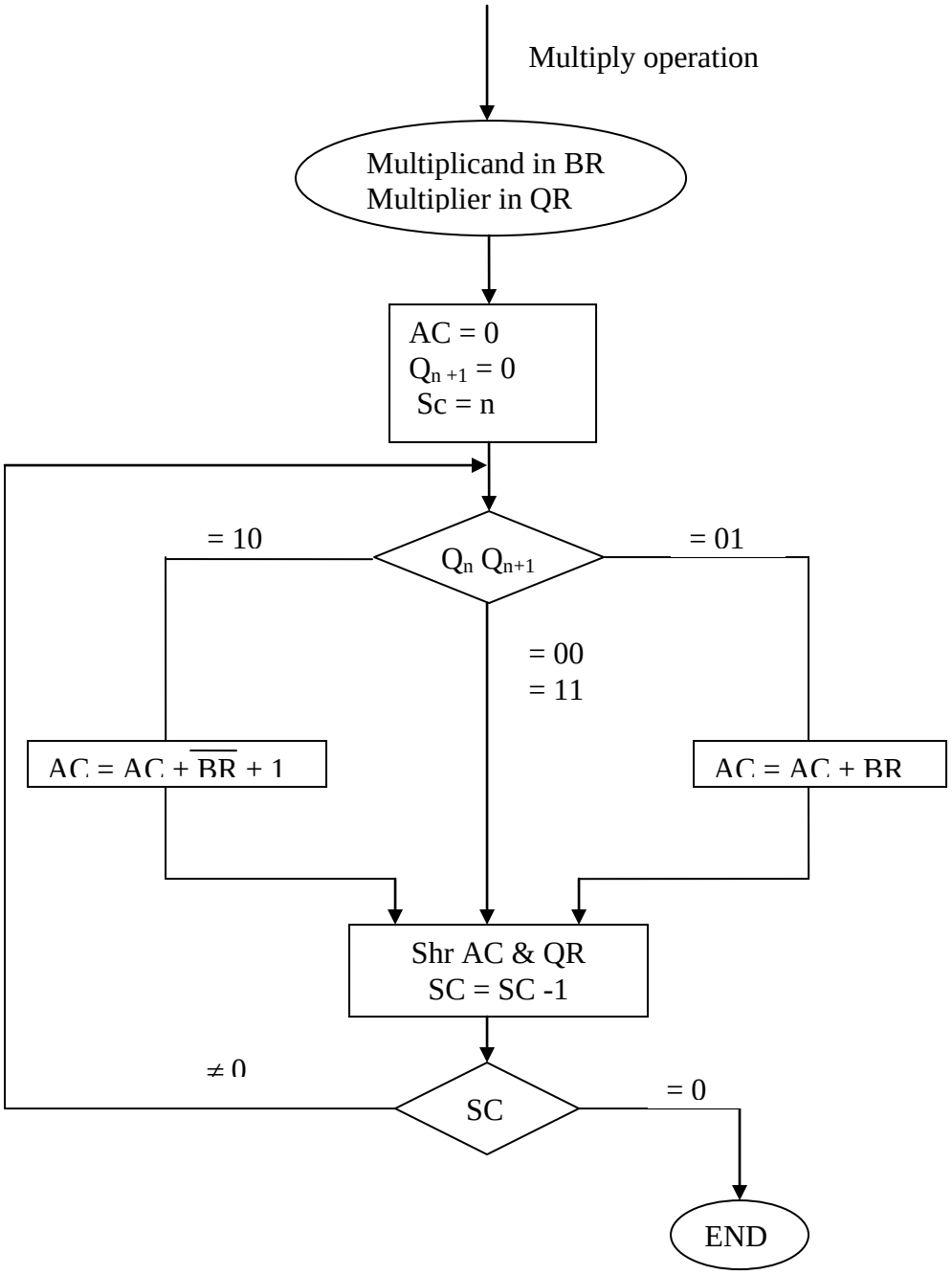


Fig. Hardware for Booth Algorithm


Multiply operation
Multiplicand in BR
Multiplier in QR
AC = 0
Q_{n+1} = 0
Sc = n
= 10
Q_n Q_{n+1}
= 01
= 00
= 11
AC = AC + $\overline{BR}$ + 1
AC = AC + BR
Shr AC & QR
SC = SC -1
≠ 0
SC
= 0
END

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

Array Multiplier

Checking the bits of the multiplier one at a time and forming partial products is a sequential operation that requires a sequence of add and shift microoperations. The multiplication of two binary numbers can be done with one micro-operation by means of a combinational circuit that forms the product bits all at once.

Example of Multiplication with Booth Algorithm

$Q_n Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000

This is a fast way of multiplying two numbers since all it takes is the time for the signals to propagate through the gates that form the multiplication array. However, an array multiplier requires a large number of gates, and for this reason it was not economical until the development of integrated circuits.

To see how an array multiplier can be implemented with a combinational circuit, consider the multiplication of two 2-bit numbers as shown in Fig. 10-9. The multiplicand bits are b1 and bo, the multiplier bits are al and ao, and the product is c3 c2 c1 co. The first partial product is formed by multiplying ao by bi bo. The multiplication of two bits such as ao and bo produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation and can be implemented with an AND gate. As shown in the diagram, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying a1 by b, bo and is shifted one position to the left. The two partial products are added with two half-adder (HA) circuits. Usually, there are more bits in the partial products and it will be necessary to use full-adders to produce the sum. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.

A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary

output in each level of AND gatesis added in parallel with the partial product of the previous level to form a new partial product. The last level produces the product. For j multiplier bits and k multiplicand bits we need j x k AND gates and (j — 1) k-bit adders to produce a product of j + k bits. As a second example, consider a multiplier circuit that multiplies a binary number of four bits with a number of three bits. Let the multiplicand be represented by b_3 b_2 b_i b_o and the multiplier by a_2 a_l a_0. Since k = 4 and j = 3, we need 12 AND gates and two 4-bit adders to produce a product of seven bits. The logic diagram of the multiplier is shown in Figure above.

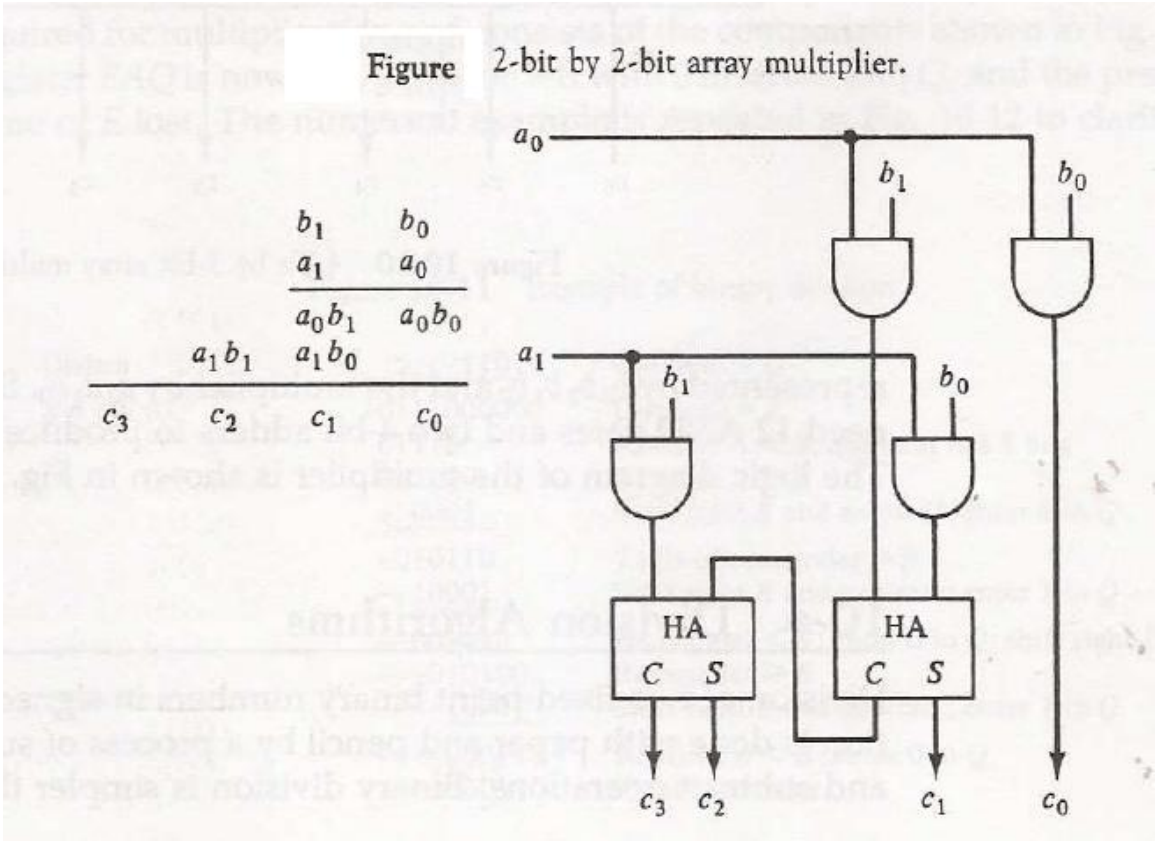


Figure 2-bit by 2-bit array multiplier.

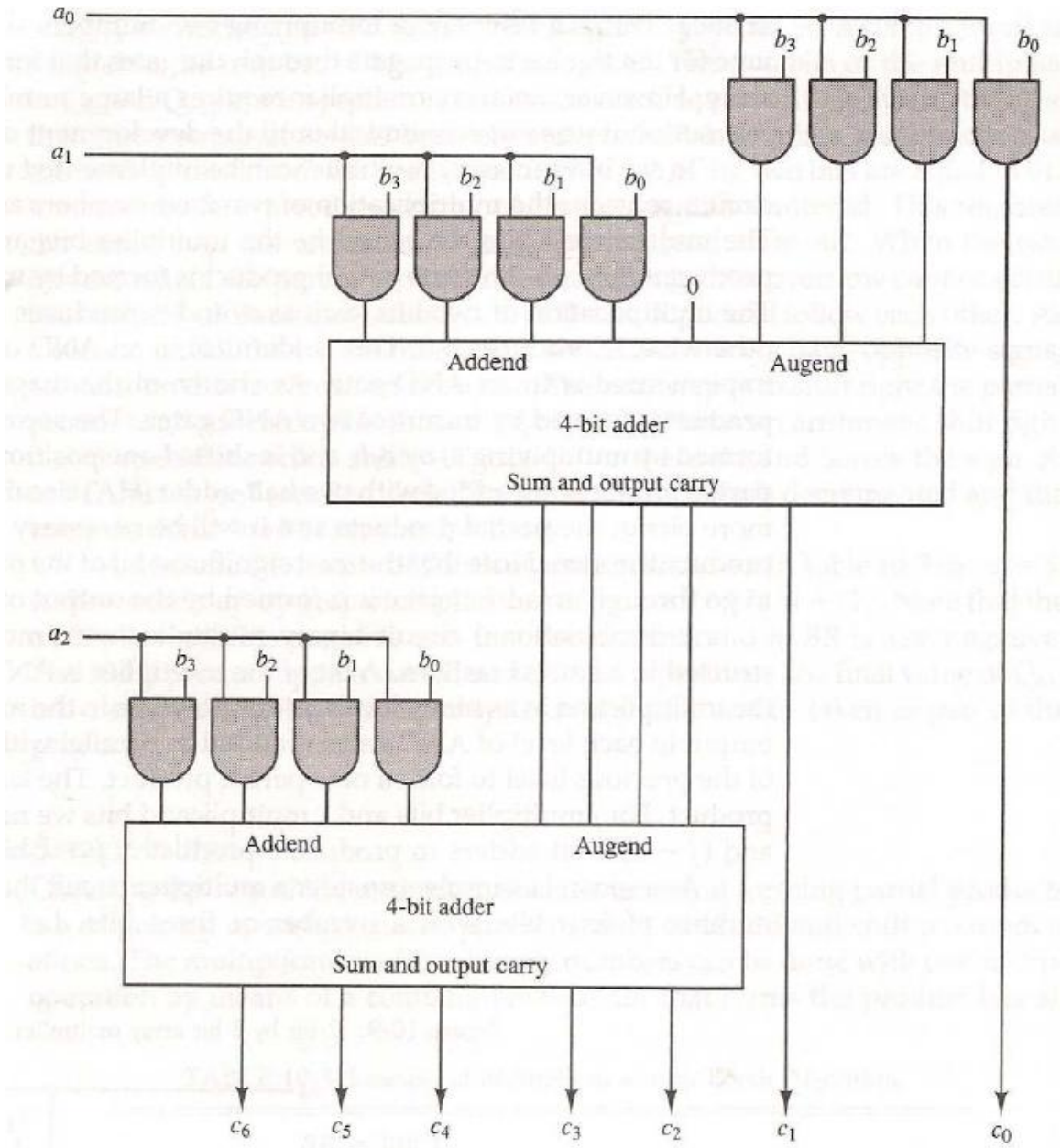


Figure 4-bit by 3-bit array multiplier.

The logic diagram of the multiplier is shown in Figure above.

Division

Division of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive compare, shift, and subtract operations. Binary division is simpler than decimal division because the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor. The division process is illustrated by a numerical example in Fig. 1041. The divisor B consists of five bits and the dividend A, of ten bits. The five most significant bits of the dividend are compared with the divisor. Since the 5-bit number is smaller than B, we try again by taking the six most significant bits of A and compare this number with B. The 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. The divisor is then shifted once to the right and subtracted from the dividend. The difference is called a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. The process is continued by comparing a partial remainder with the divisor. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Note that the result gives both a quotient and a remainder.

Hardware Implementation for Signed-Magnitude Data

When the division is implemented in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, the dividend, or partial remainder, is shifted to the left, thus leaving the two numbers in the required relative position. Subtraction may be achieved by adding A to the 2's complement of B. The information about the relative magnitudes is then available from the end-carry. The hardware for implementing the division operation is identical to that required for multiplication. Register *EAQ* is now shifted to the left with 0 inserted into Q_n and the previous value of E lost. The numerical example is repeated in Fig. 10-12 to clarify the proposed division process.

Figure 10-11 Example of binary division.

Divisor:		Quotient = Q
B = 10001	11010	
	)0111000000	Dividend = A
	01110	5 bits of $A < B$, quotient has 5 bits
	011100	6 bits of $A \geq B$
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder $\geq B$
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder $< B$; enter 0 in Q; shift right B
	---010100	Remainder $\geq B$
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder $< B$; enter 0 in Q
	-----00110	Final remainder

Divisor B = 10001, $\overline{B} + 1$ = 01111

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
E = 1	1	01011		
Set Q_n = 1	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
E = 1	1	00101		
Set Q_n = 1	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
E = 0; leave Q_n = 0	0	11001	00110	
Add B		10001		
Restore remainder	1	01010		2
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
E = 1	1	00011		
Set Q_n = 1	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
E = 0; leave Q_n = 0	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Figure 10-12 Example of binary division with digital hardware.

The divisor is stored in the B register and the double-length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value. The information about the relative magnitude is available in E. If E = 1, it signifies that A B. A quotient bit 1 is inserted into Q, and the partial remainder is shifted to the left to repeat the process. If E = 0, it signifies that A < B so the quotient in Q. remains a 0 (inserted during the shift). The value of B is then added to restore the partial remainder in A to its previous value. The partial remainder is shifted to the left and the process is repeated again until all five quotient bits are formed. Note that while the partial remainder is shifted left, the quotient bits are shifted also and after five shifts, the quotient is in Q and the final remainder is in A. Before showing the algorithm in flowchart form, we have to consider the sign of the result and a possible overflow condition. The sign of the quotient is determined from the signs of the dividend and the divisor. If the two signs are alike, the sign of the quotient is plus. If they are unlike, the sign is minus. The sign of the remainder is the same as the sign of the dividend.

Overflow

The division operation may result in a quotient with an overflow. This is not a problem when working with paper and pencil but is critical when the operation is implemented with hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length. To see this, consider a system that has 5-bit registers. We use one register to hold the divisor and two registers to hold the dividend. From the example of Fig. 10-11 we note that the quotient will consist of six bits if the five most significant bits of the dividend constitute a number greater than the divisor. The quotient is to be stored in a standard 5-bit register, so the overflow bit will require one more flip-flop for storing the sixth bit. This divide-overflow condition must be avoided in normal computer operations because the entire quotient will be too long for transfer into a memory unit that has words of standard length, that is, the same as the length of registers. Provisions to ensure that this condition is detected must be included in either the hardware or the software of the computer, or in a combination of the two.

When the dividend is twice as long as the divisor, the condition for overflow can be stated as follows: A divide-overflow condition occurs if the high-order half bits of the dividend constitute a number greater than or equal to the divisor. Another problem associated with division is the fact that a

division by zero must be avoided. The divide-overflow condition takes care of this condition as well. This occurs because any dividend will be greater than or equal to a divisor which is equal to zero. Overflow condition is usually detected when a special flip-flop is set. We will call it a divide-overflow flip-flop and label it DVF.

The occurrence of a divide overflow can be handled in a variety of ways. In some computers it is the responsibility of the programmers to check if DVF is set after each divide instruction. They then can branch to a subroutine that takes a corrective measure such as renting the data to avoid overflow. In some older computers, the occurrence of a divide overflow stopped the computer and this condition was referred to as a divide stop. Stopping the operation of the computer is not recommended because it is time consuming. The procedure in most computers is to provide an interrupt request when DVF is set. The interrupt causes the computer to suspend the current program and branch to a service routine to take a corrective measure. The most common corrective measure is to remove the program and type an error message explaining the reason why the program could not be completed. It is then the responsibility of the user who wrote the program to rescale the data or take any other corrective measure. The best way to avoid a divide overflow is to use floating-point data.

Hardware Algorithm

The hardware divide algorithm is shown in the flowchart below. The dividend is in *A* and Q and the divisor in *B*. The sign of the result is transferred into Q_s to be part of the quotient. A constant is set into the sequence counter *SC* to specify the number of bits in the quotient. As in multiplication, we assume that operands are transferred to registers from a memory unit that has words of *n* bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of *n*-1 bits.
A divide-overflow condition is tested by subtracting the divisor in *B* from half of the bits of the dividend stored in *A*. If $A \geq B$, the divide-overflow flip-flop *DVF* is set and the operation is terminated prematurely. If $A < B$, no divide overflow occurs so the value of the dividend is restored by adding *B to A*.

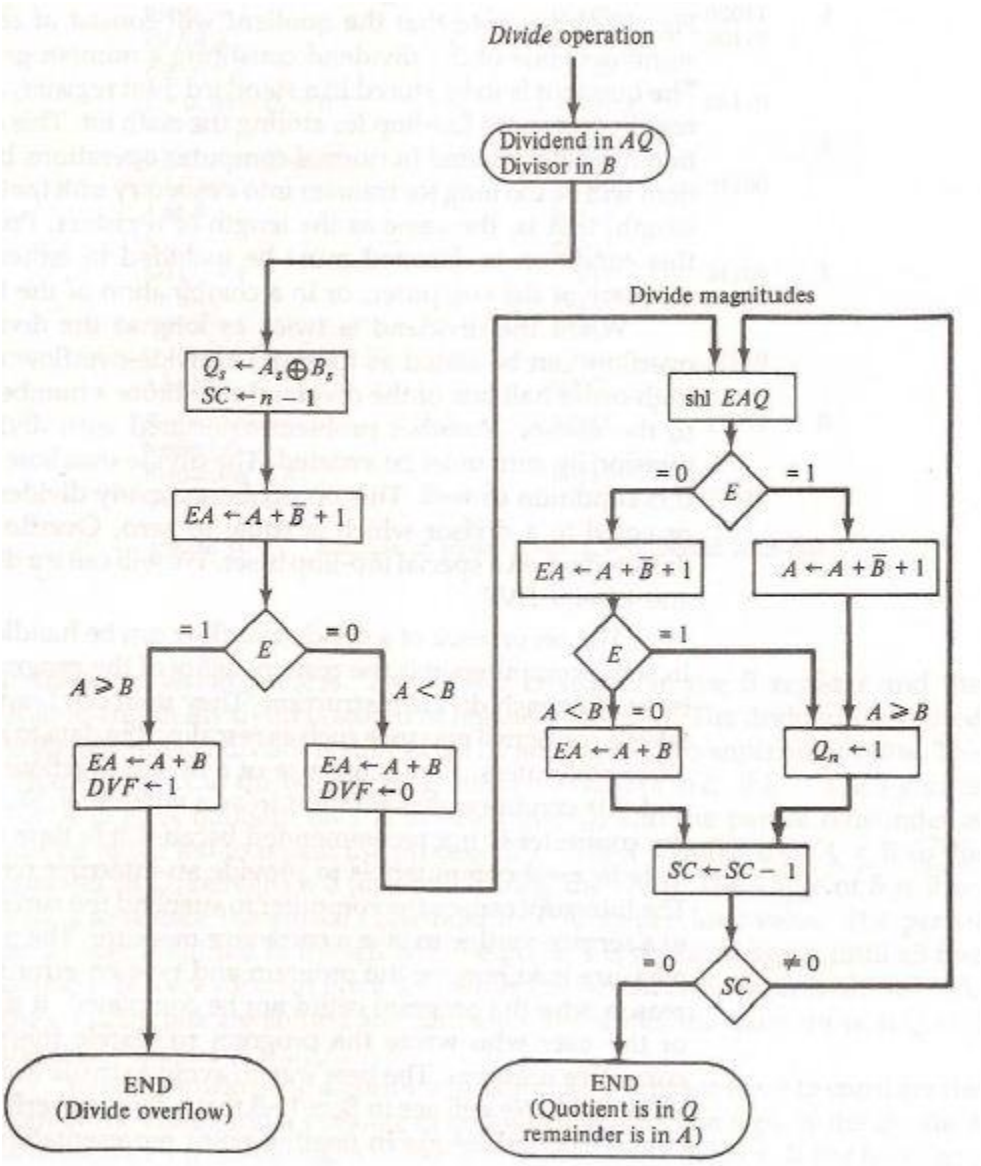


Fig. Flowchart for divide operation

The division of the magnitudes starts by shifting the dividend in *AQ* to the left with the high-order bit shifted into *E.* If the bit shifted into E is 1, we know that EA > *B* because *EA* consists of a 1 followed by *n* -1 bits while *B* consists of only *n* - 1 bits. In this case, *B* must be subtracted from *LA* and 1 inserted into Q_n for the quotient bit. Since register *A* is missing the high-order bit of the dividend (which is in E*)*, its value is *EA* - 2^{n-1}. Adding to this value the 2's complement of *B* results in

$$(EA - 2^{n-1}) + (2^{n-1} - B) = EA - B$$

The carry from this addition is not transferred to E if we want E to remain a 1.

If the shift-left operation inserts a 0 into E, the divisor is subtracted by adding its 2's complement value and the carry is transferred into E. If E = 1, it signifies that A≥B; therefore, Q_n is set to 1. If E = 0, it signifies that A < B and the original number is restored by adding B to A. In the latter case we leave a 0 in Q_n (0 was inserted during the shift). This process is repeated again with register A holding the partial remain-der. After n - 1 times, the quotient magnitude is formed in register Q_s and the remainder is found in register A. The quotient sign is in Q, and the sign of the remainder in A_s is the same as the original sign of the dividend.

Restoring Method

The hardware method just described is called the restoring method. The reason for this name is that the partial remainder is restored by adding the divisor to the negative difference.

Comparison and Non-Restoring Method

Two other methods are available for dividing numbers, the comparison method and the non-restoring method. In the comparison method A and B are compared prior to the subtraction operation. Then if A ≥ B, B is subtracted from A. If A < B nothing is done. The partial remainder is shifted left and the numbers are compared again. The comparison can be determined prior to the subtraction by inspecting the end-carry out of the parallel-adder prior to its transfer to register E. In the non-restoring method, B is not added if the difference is negative but instead, the negative difference is shifted left

and then B is added. To see why this is possible consider the case when A < B. From the flowchart in Fig. 9-11 we note that the operations performed are A - B + B; that is, B is subtracted and then added to restore A. The next time around the loop, this number is shifted left (or multiplied by 2) and B subtracted again. This gives 2(A - B + B) - B = 2A - B. This result is obtained in the non-restoring method by leaving A - B as is. The next time around the loop, the number is shifted left and B added to give 2(A - B) + B = 14 - B, which is the same as before. Thus, in the non-restoring method, B is subtracted if the previous value of Q,, was a 1, but B is added if the previous value of Q_n was a 0 and no restoring of the partial remainder is required. This process saves the step of adding the divisor if A is less than B, but it requires special control logic to remember the previous result. The first time the dividend is shifted, B must be subtracted. Also, if the last bit of the quotient is 0, the partial remainder must be restored to obtain the correct final remainder.

Chapter 7
Input-Output Organization

Introduction to Peripheral Devices

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

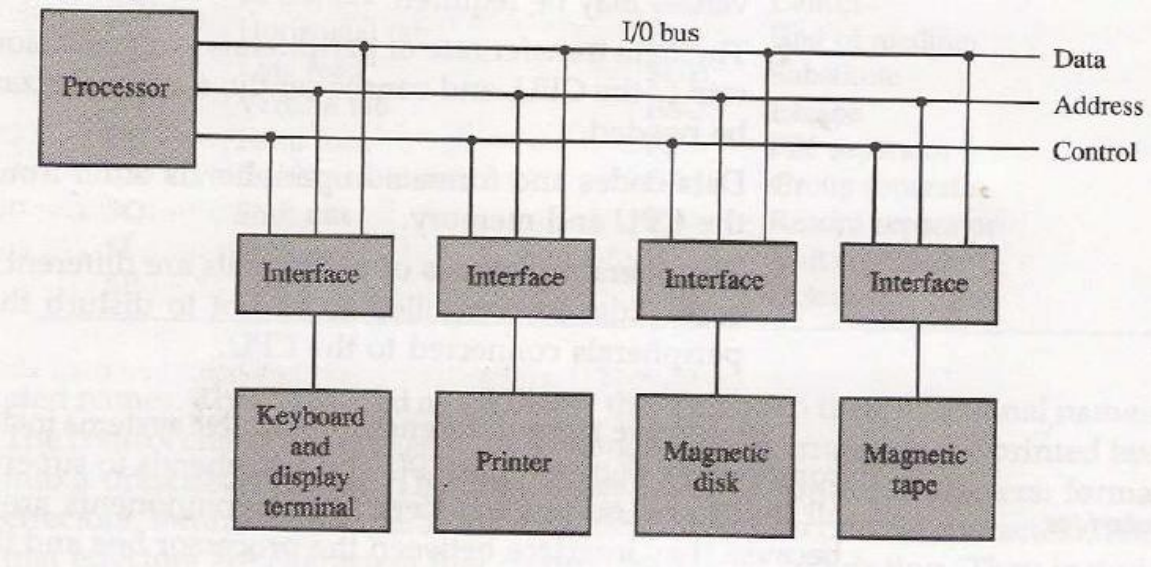


Fig. Connection of I/O bus to input-output devices.

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

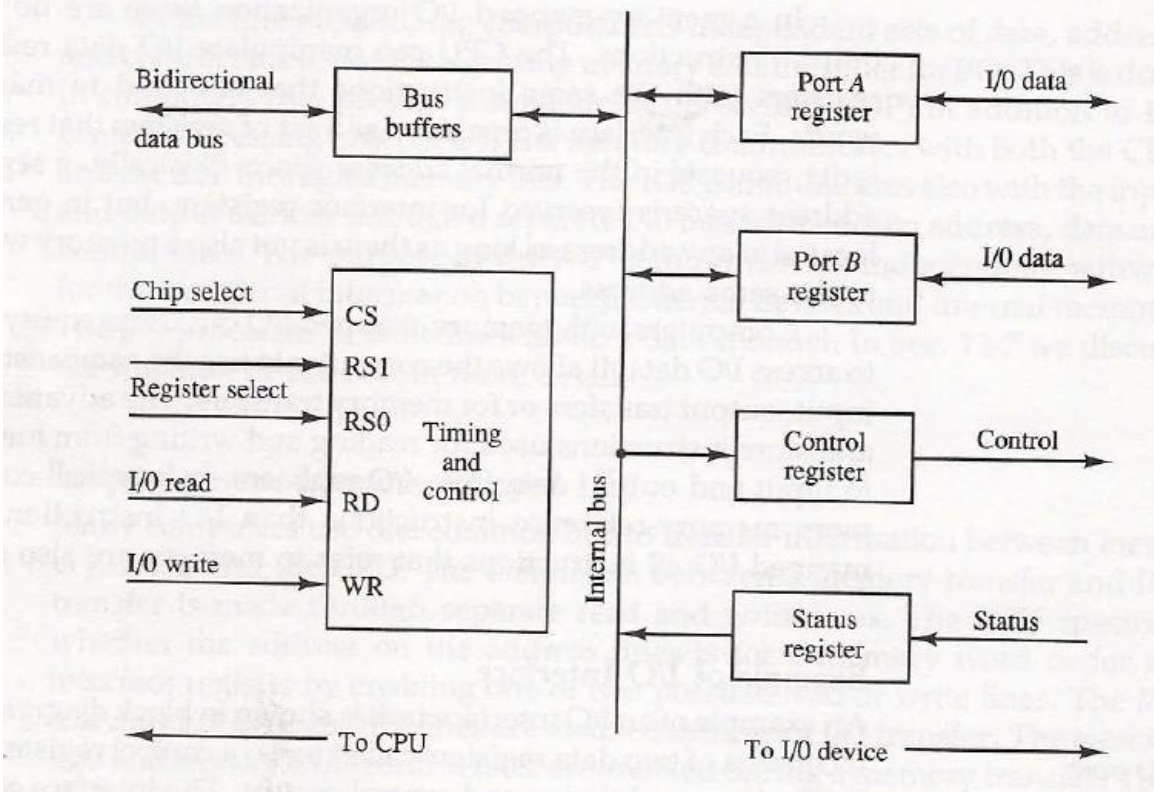
Isolated vs. Memory mapped I/O
Isolated I/O

- 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

I/O Interface Unit



CS	RS1	RS0	Register selected
0	×	×	None: data bus in high–impedance
1	0	0	Port *A* register
1	0	1	Port *B* register
1	1	0	Control register
1	1	1	Status register

Figure 11-2 Example of I/O interface unit.

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.

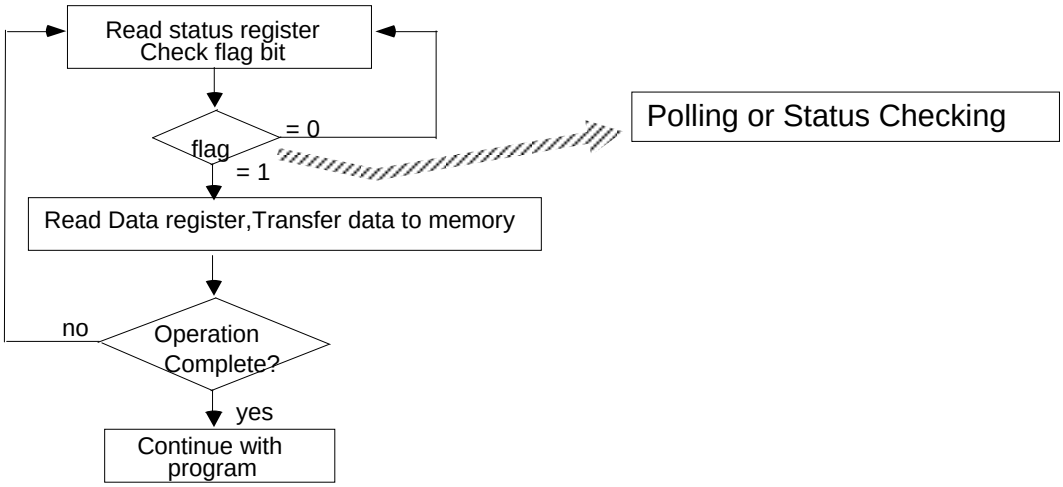
Direct Memory Access (DMA)

Types of I/O

- 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)

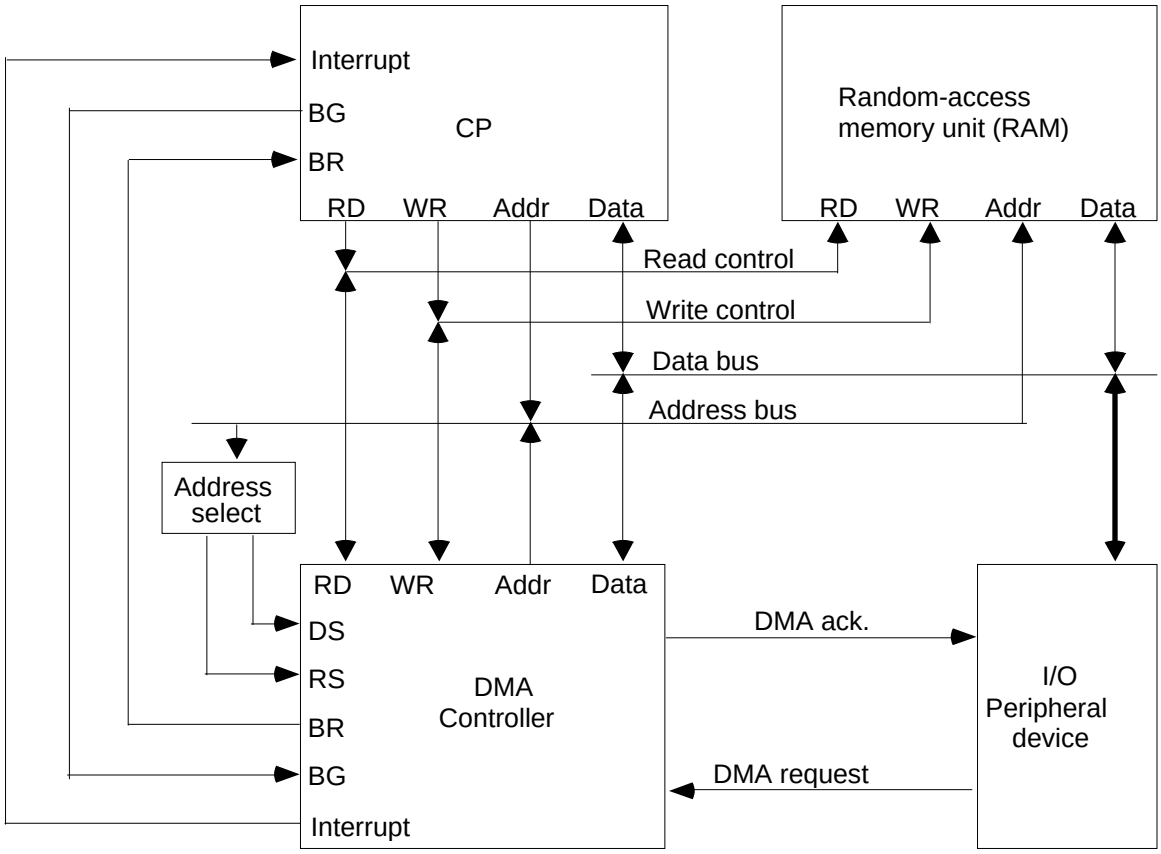
Large blocks of data transferred at a high speed to or from high speed devices, magnetic drums, disks, tapes, etc.

- 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

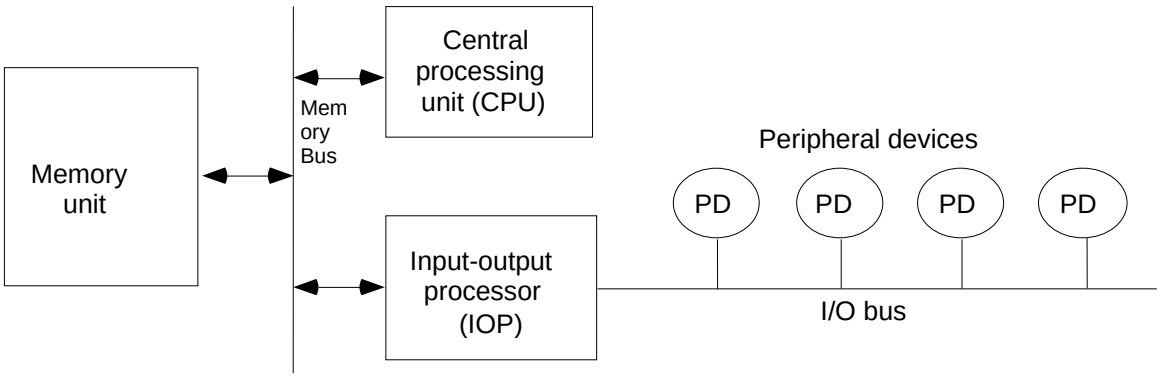
- Send write control signal to memory
- Increment address register (Address Reg <- Address Reg +1)
- Decrement word count (WC <- WC – 1)
- If WC = 0, then Interrupt to acknowledge done, else repeat same process

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)

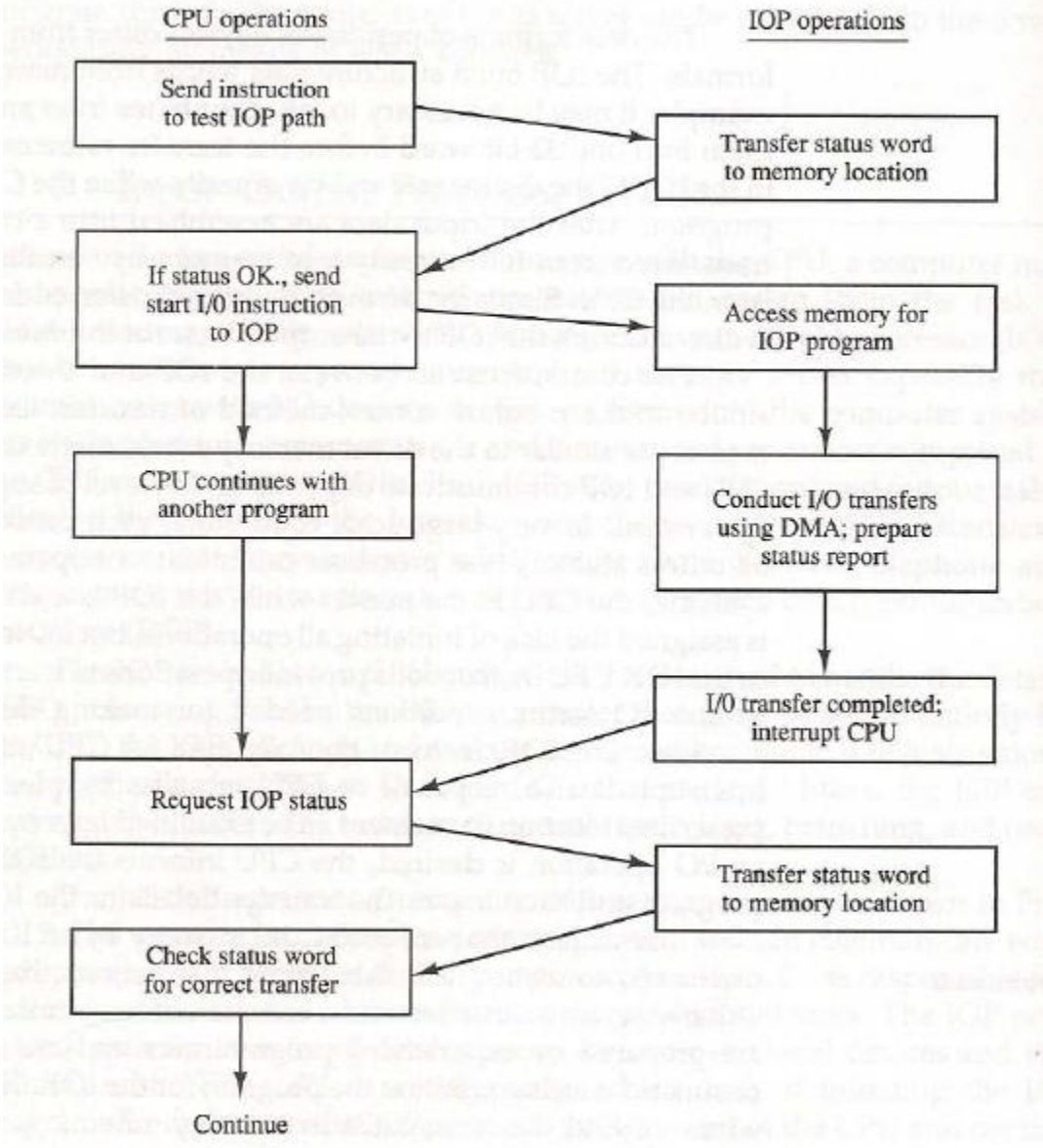
Processor with direct memory access capability that communicates with I/O devices is called I/O processor (channel). Channel accesses memory by cycle stealing. Channel can execute a channel program stored in the main memory. CPU initiates the channel by executing a channel I/O class instruction and once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



CPU-IOP Communication

The communication between CPU and IOP may take different forms, depending on the particular computer considered. In most cases the memory unit acts as a message center where each processor leaves information for the other. To appreciate the operation of a typical IOP, we will illustrate by a specific example the method by which the CPU and IOP communicate. This is a simplified example that omits many operating details in order to provide an overview of basic concepts.

Figure 11-20 CPU-IOP communication.



The sequence of operations may be carried out as shown in the flowchart of Fig. 11-20. The CPU sends an instruction to test the IOP Path. The IOP responds by inserting a status word in memory for the CPU to check. The bits of the status word indicate the condition of the IOP and I/0 devices, such as TOP overload condition, device busy with another transfer, or device ready for I/O transfer. The CPU refers to the status word in memory to decide what to do next. If all is in order, the CPU sends the instruction to start I/O transfer. The memory address received with this instruction tells the IOP where to find its program.

The CPU can now continue with another program while the IOP is busy with the I/O program. Both programs refer to memory by means of DMA transfer. When the IOP terminates the execution of its program, it sends an interrupt request to the CPU. The CPU responds to the interrupt by issuing an instruction to read the status from the IOP. The IOP responds by placing the contents of its status report into a specified memory location. The status word indicates whether the transfer has been completed or if any errors occurred during the transfer. From inspection of the bits in the status word, the CPU determines if the I/O operation was completed satisfactorily without errors.

The IOP takes care of all data transfers between several 110 units and the memory while the CPU is processing another program. The IOP and CPU are competing for the use of memory, so the number of devices that can be in operation is limited by the access time of the memory. It is not possible to saturate the memory by I/O devices in most systems, as the speed of most devices is much slower than the CPU. However, some very fast units, such as magnetic disks can use an appreciable number of the available memory cycles. In that case, the speed of the CPU may deteriorate because it will often have to wait for the IOP to conduct memory transfers.

Serial and Parallel Communication

Serial Communication

It is the process of sending data one bit at a time, sequentially, over a communication channel or computer bus. This is in contrast to parallel communication, where several bits are sent as a whole, on a link with several parallel channels.

Serial communication is used for all long-haul communication and most computer networks, where the cost of cable and synchronization difficulties makes parallel communication impractical. Serial computer buses are becoming more common even at shorter distances, as improved signal integrity and transmission speeds in newer serial technologies have begun to outweigh the parallel bus's advantage of simplicity (no need for serializer and deserializer, or SerDes) and to outstrip its disadvantages (clock skew, interconnect density). The migration from PCI to PCI Express is an example.

Parallel Communication

It is a method of conveying multiple binary digits (bits) simultaneously. It contrasts with serial communication, which conveys only a single bit at a time; this distinction is one way of characterizing a communications link.

Parallel communication implies more than one such conductor. For example, an 8-bit parallel channel will convey eight bits (or a byte) simultaneously, whereas a serial channel would convey those same bits sequentially, one at a time. If both channels operated at the same clock speed, the parallel channel would be eight times faster. A parallel channel may have additional conductors for other signals, such as a clock signal to pace the flow of data, a signal to control the direction of data flow, and handshaking signals.

Data Communication Processor

A data communication Processor is an I/O processor that distributes and collects data from many remote terminals connected through telephone and

other communication lines. It is a specialized I/O processor designed to communicate directly with data communication networks. A communication network may consist of any of a wide variety of devices, such as printers, interactive display devices, digital sensors, or a remote computing facility. With the use of a data communication processor, the computer can service fragments of each network demand in an interspersed manner and thus have the apparent behavior of serving many users at once. In this way the computer is able to operate efficiently in a time-sharing environment.

The most striking difference between an I/O processor and a data communication processor is in the way the processor communicates with the I/O devices. An I/O processor communicates with the Peripherals through a common I/O bus that is comprised of many data and control lines. All peripherals share the common bus and use it to transfer information to and from the I/O processor. A data communication processor communicates with each terminal through a single pair of wires. Both data and control information are trans-character received. Another procedure used in asynchronous terminals involving a human operator is to *echo* the character. The character transmitted from the keyboard to the computer is recognized by the processor and retransmitted to the terminal printer. The operator would realize that an error occurred during transmission if the character printed is not the same as the character whose key he has struck.

Modes of Data Transfer

Data can be transmitted between two points in three different modes: simplex, half-duplex, or full-duplex.

Simplex

A simplex line carries information in one direction only. This mode is seldom used in data communication because the receiver cannot communicate with the transmitter to indicate the occurrence of errors. Examples of simplex transmission are radio and television broadcasting.

Half-Duplex

A half-duplex transmission system is one that is capable of transmitting in both directions but data can be transmitted in only one direction at a time. A pair of wires is needed for this mode. A common situation is for one modem to ad as the transmitter and the other as the receiver. When transmission in one direction is completed, the role of the modems is reversed to enable transmission in the reverse direction. The time required to switch a half-duplex line from one direction to the other is called the turnaround time.

Full-Duplex

A full-duplex transmission can send and receive data in both directions simultaneously. This can be achieved by means of a four-wire link, with a different pair of wires dedicated to each direction of transmission. Alternatively, a two-wire circuit can support full-duplex communication if the frequency spectrum is subdivided into two non-overlapping frequency bands to create separate receive and transmit channels in the same physical pair of wires.

Protocol

The communication lines, modems, and other equipment used in the transmission of information between two or more stations are called a data link. The orderly transfer of information in a data link is accomplished by means of a *protocol*. A data link control protocol is a set of rules that are followed by interconnecting computers and terminals to ensure the orderly transfer of information. The purpose of data link processor is to establish and terminate a connection between two stations, to identify the sender and receiver, to ensure that all messages are passed correctly without errors, and to handle all control functions involved in a sequence of data transfers. Protocols are divided into two major categories according to the message-framing technique used. These are character-oriented protocol and bit-oriented protocol.

Character-Oriented Protocol

The character-oriented protocol is based on the binary code of a character set. The code most commonly used is ASCII (American Standard Code for Information Interchange). It is a 7-bit code with an eighth bit used for parity. The code has 128 characters, of which 95 are graphic characters and 33 are control characters. The graphic characters include the upper- and lowercase letters, the ten numerals, and a variety of special symbols. The control characters are used for the purpose of routing data, arranging the test in a desired format, and for the layout of the printed page. The characters that control the transmission are called communication control characters. These characters are listed in Table 11-4. Each character has a 7-bit code and is referred to by a three-letter symbol. The role of each character in the control of data transmission is stated briefly in the function column of the table.

The SYN character serves as synchronizing agent between the transmitter and receiver. When the 7-bit ASCII code is used with an odd-parity bit in the most significant position, the assigned SYN character has the 8-bit code 00010110 which has the property that, upon circular shifting, it repeats itself only after a full 8-bit cycle. When the transmitter starts sending 8-bit characters, it sends a few characters first and then sends the actual message. The initial continuous string of bits accepted by the receiver is checked for a SYN character. In other words, with each clock pulse, the receiver checks the last eight bits received. Of they do not match the bits of the SYN character, the receiver accepts the next bit, rejects the previous high-order bit

TABLE 11-4 ASCII Communication Control Characters

Code	Symbol	Meaning	Function
0010110	SYN	Synchronous idle	Establishes synchronism
0000001	SOH	Start of heading	Heading of block message
0000010	STX	Start of text	Precedes block of text
0000011	ETX	End of text	Terminates block of text
0000100	EOT	End of transmission	Concludes transmission
0000110	ACK	Acknowledge	Affirmative acknowledgement
0010101	NAK	Negative acknowledge	Negative acknowledgement
0000101	ENQ	Inquiry	Inquire if terminal is on
0010111	ETB	End of transmission block	End of block of data
0010000	DLE	Data link escape	Special control character

and again checks the last eight bits received for a SYN character. This is repeated after each clock pulse and bit received until a SYN character is recognized. Once a SYN character is detected, the receiver has framed a character. From here on the receiver counts every eight bits and accepts them as a single character. Usually, the receiver checks two consecutive SYN characters to remove any doubt that the first did not occur as a result of a noise signal on the line Moreover, when the transmitter is idle and does not have any message characters to send, it sends a continuous string of SYN characters. The receiver recognizes these characters as a condition for synchronizing the line and goes into a synchronous idle state. In this state, the two units maintain bit and character synchronism even though no meaningful information is communicated.

Messages are transmitted through the data link with an established format consisting of a header field, a text field, and an error-checking field. A typical message format for a character-oriented protocol is shown in Fig. 11-25. The two SYN characters assure proper synchronization at the start of the message. Following the SYN characters is the header, which starts with an SOH (start of heading) character. The header consists of address and control information. The STX character terminates the header and signifies the beginning of the text transmission. The text portion of the message is variable in length and may contain any ASCII characters except the communication control characters. The text field is terminated with the ETX character. The last field is a block check character (BCC) used for error checking. It is usually either a longitudinal redundancy check (LRC) or a cyclic redundancy check (CRC).

The receiver accepts the message and calculates its own BCC. If the BCC transmitted does not agree with the BCC calculated by the receiver, the receiver responds with a negative acknowledge (NAK) character. The message is then retransmitted and checked again. Retransmission will be typically attempted several times before it is assumed that the line is faulty. When the transmitted BCC matches the one calculated by the receiver, the response is a positive acknowledgment using the ACK character.

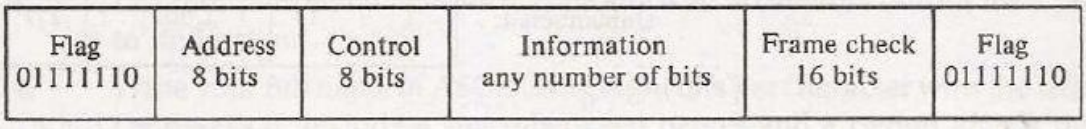
Bit-Oriented Protocol

The bit-oriented protocol does not use characters in its control field and is independent of any particular code. It allows the transmission of serial bit stream of any length without the implication of character boundaries. Messages are organized in a specific format called a frame. In addition to the information field, a frame contains address, control, and error-checking fields. The frame boundaries are determined from a special 8-bit number called a flag. Examples of bit-oriented protocols are SDLC (synchronous data link control) used by IBM, HDLC (high-level data link control) adopted by the International Standards Organization, and ADCCP (advanced data communication control procedure) adopted by the American National Standards Institute.

Any data communication link involves at least two participating stations. The station that has responsibility for the data link and issues the commands to control the link is called the primary station. The other station is a secondary station. Bit-oriented protocols assume the presence of one primary station and one or more secondary stations. All communication on the data link is from the primary station to one or more secondary stations or from a secondary station to the primary station.

The frame format for the bit-oriented protocol is shown in Fig. 11-26.

Figure 11-26 Frame format for bit-oriented protocol.



A frame starts with the 8-bit flag 01111110 followed by an address and control sequence. The information field is not restricted in format or content and can be of any length. The frame check field is a CRC (cyclic redundancy check) sequence used for detecting errors in transmission. The ending flag indicates to the receiving station that the 16 bits just received constitute the CRC bits. The ending frame can be followed by another frame, another flag, or a sequence of consecutive l's. When two frames follow each other, the intervening flag is simultaneously the ending flag of the first frame and the beginning flag of the next frame. If no information is exchanged, the transmitter sends a series of flags to keep the line in the active state. The line

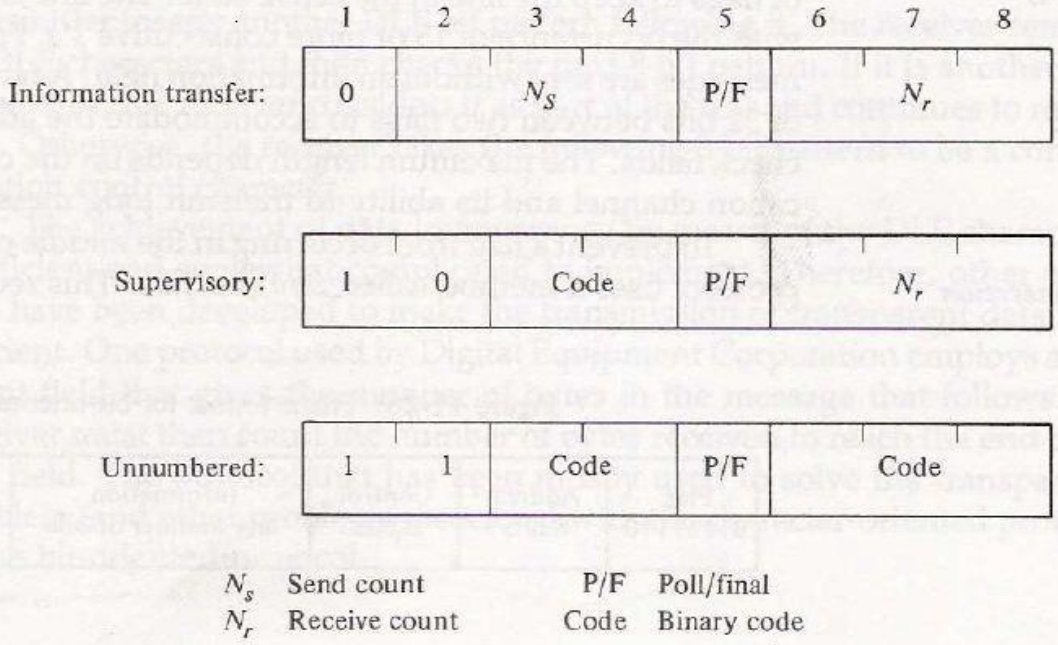
is said to be in the idle state with the occurrence of 15 or more consecutive l's. Frames with certain control messages are sent without an information field. A frame must have a minimum of 32 bits between two flags to accommodate the address, control, and frame check fields. The maximum length depends on the condition of the communication channel and its ability to transmit long messages error-free.

To prevent a flag from occurring in the middle of a frame, the bit-oriented protocol uses a method called zero insertion. This requires that a 0 be inserted by the transmitting station after any succession of five continuous 1's. The receiver always removes a 0 that follows a succession of five 1's. Thus the bit pattern 0111111 is transmitted as 01111101 and restored by the receiver to its original value by removal of the 0 following the five 1's. As a consequence, no pattern of 01111110 is ever transmitted between the beginning and ending flags.

Following the flag is the address field, which is used by the primary station to designate the secondary station address. When a secondary station transmits a frame, the address tells the primary station which secondary station originated the frame. An address field of eight bits can specify up to 256 addresses. Some bit-oriented protocols permit the use of an extended address field. To do this, the least significant bit of an address byte is set to 0 if another address byte follows. A 1 in the least significant bit of a byte is used to recognize the last address byte.

Following the address field is the control field. The control field comes in three different formats, as shown in Fig. 11-27.

Figure 11-27 Control field format in bit-oriented protocol.



The information transfer format is used for ordinary data transmission. Each frame transmitted in this format contains send and receive counts. A station that transmits sequenced frames counts and numbers each frame. This count is given by the send count N_s. A station receiving sequenced frames counts each error-free frame that it receives. This count is given by the receive count N_r. The N_r count advances when a frame is checked and found to be without errors. The receiver confirms accepted numbered information frames by returning its N_r count to the transmitting station.

The P/F bit is used by the primary station to poll a secondary station to request that it initiate transmission. It is used by the secondary station to indicate the final transmitted frame. Thus the P/F field is called P (poll) when the primary station is transmitting but is designated as F (final) when a secondary station is transmitting. Each frame sent to the secondary station from the primary station has a P bit set to 0. When the primary station is finished and ready for the secondary station to respond, the P bit is set to 1. The secondary station then responds with a number of frames in which the F bit is set to 0. When the secondary station sends the last frame, it sets the F bit to 1. Therefore, the P/F bit is used to determine when data transmission from a station is finished.

The supervisory format of the control field is recognized from the first two bits being 1 and 0. The next two bits indicate the type of command. This follows by a P/F bit and a receive sequence frame count. The frames of the supervisory format do not carry an information field. They are used to assist in the transfer of information in that they confirm the acceptance of preceding frames carrying information, convey ready-or busy conditions, and report frame numbering errors.

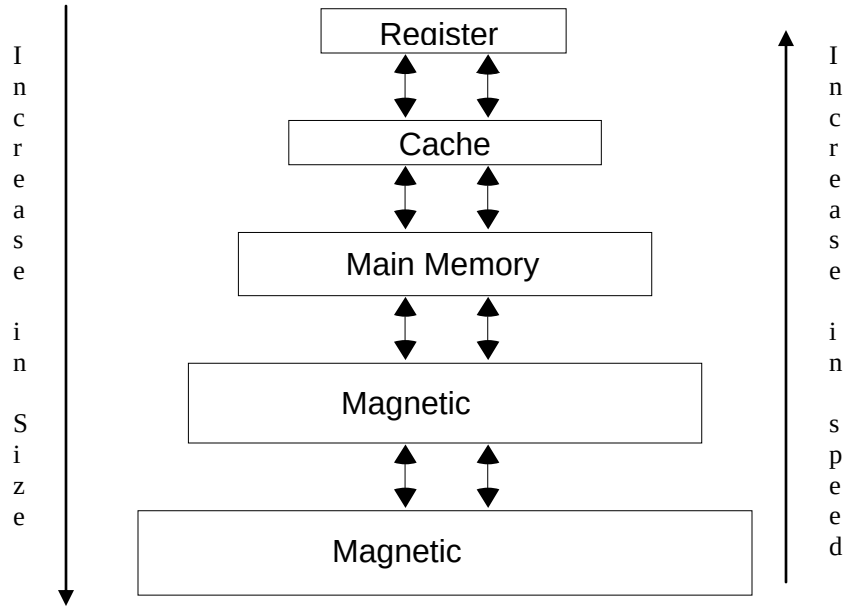
The unnumbered format is recognized from the first two bits being 11. The five code bits available in this format can specify up to 32 commands and responses. The primary station uses the control field to specify a command for a secondary station. The secondary station uses the control field to transmit a response to the primary station. Unnumbered-format frames are employed for initialization of link functions, reporting procedural errors, placing stations in a disconnected mode, and other data link control operations.

Chapter 8

Memory Organization

Hierarchy of Memory System

Main goal of memory Hierarchy is to obtain the highest possible access speed while minimizing the total cost of the memory system



Types of Memory

On the basis of access memory can be categorized in three types

Sequential access: - If we want to read a data from some particular position all the data between current head position and position of the data need to be read. Transfer time heavily depends upon location of the data

Direct Access: - If we want to read a data from some particular position all the data between current head position and position of the data need not to be read but head position must move from current position to position of the data (without reading) Transfer time still depends upon location of the data

Random Access:- There is no concept of head position.. Same time is needed to read the data from any location.

Memory Hierarchy

- Registers
  - In CPU
- Internal or Main memory
  - May include one or more levels of cache
  - "RAM"
- External memory
  - Backing store

Primary and Secondary Memory

Main Memory

The memory which is used by the CPU to during program execution is called main memory. It directly connected with CPU.

RAM, ROM and Cache memory are main memories.

Cache Memory

Cache is a fast small capacity memory that should hold that information which is most likely to be accessed

Locality of Reference

- The references to memory at any given time interval tends to be confined within localized areas. This characteristic of program is called locality of reference.

- **Temporal Locality**

  The information which is used currently is likely to be in use in near future (e.g. Reuse of information in loops)

- **Spatial Locality**

  If a word is accessed, adjacent (near) words are likely accessed soon (e.g. Related data items (arrays) are usually stored together; instructions are executed sequentially)

  The property of Locality of Reference makes the Cache memory systems work

Performance of cache

- All the memory accesses are directed first to cache
- If the word is in cache (cache hit); Access cache to provide it to CPU.
- If the word is not in cache (cache miss); Bring a block (or a line) including that word to replace a block now in Cache

Hit Ratio - % of memory accesses satisfied by Cache memory system
Te: Effective memory access time in Cache memory system
Tc: Cache access time
Tm: Main memory access time

$$Te = h\,Tc + (1 - h)\,Tm$$

Example: Tc = 0.4 μs, Tm = 1.2μs, h = 0.85

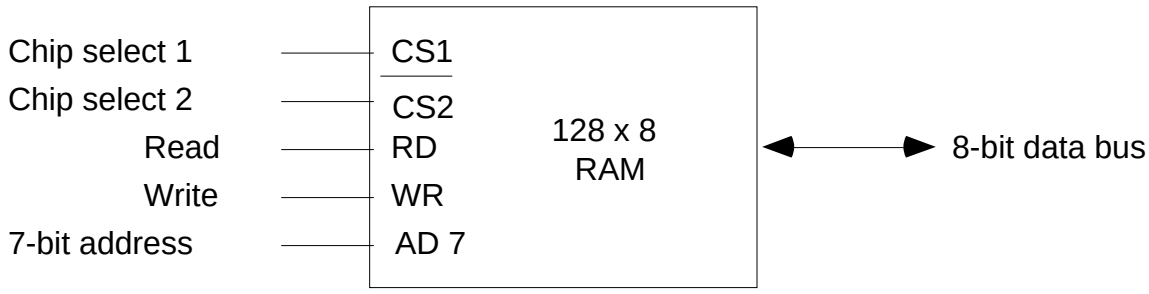
$$Te = 0.85 \times 0.4 + (1 - 0.85) * 1.2 = 0.54\mu s$$

Bootstrap Loader:

Bootstrap loader is a program that is stored in ROM and is used to start the loading of OS from hard disk to RAM when power is turned on in a computer. When a computer is turned on hardware of the computer sets PC (program counter) to first instruction of bootstrap loader so that execution of bootstrap loader begins when power is turned on.

RAM and ROM Chips

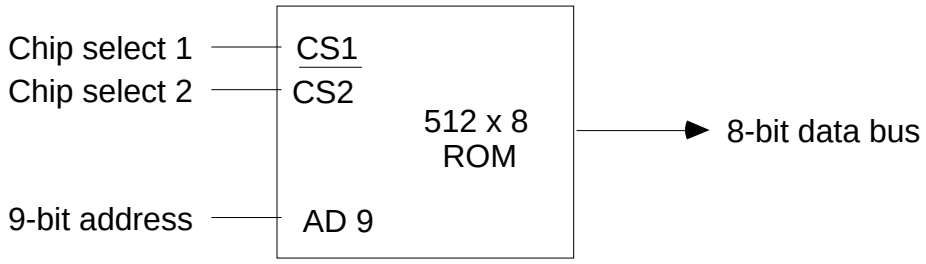
RAM chip



Two control signals chip select (CS) is used for enabling the RAM chip. Bar above CS2 indicated that chip is enabled only when CS1=1 and CS2=0. RD and WR are read and write control signals that are used to define the mode of transfer. Bidirectional data bus indicates that data can go in and out of the memory. Since the size of Ram of 128 word we need 7 bit address. Working of the chip is described by the function table given below:

CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	x	x	Inhibit	High-impedence
0	1	x	x	Inhibit	High-impedence
1	0	0	0	Inhibit	High-impedence
1	0	0	1	Write	Input data to RAM
1	0	1	x	Read	Output data from RAM
1	1	x	x	Inhibit	High-impedence

ROM Chip



Two control signals chip select (CS) is used for enabling the ROM chip. Bar above CS2 indicated that chip is enabled only when CS1=1 and CS2=0. RD and WR are not used here because ROM is read-only memory. Unidirectional data bus indicates that data can only out of the memory. Since the size of Ram of 512 words we need 9 bit address. Working of the chip can also be described by using function table similar to above.

Memory Address Map

Memory address map is a process of assigning address space to a memory system of a computer system. Suppose a memory system with 512 words of Ram and 512 words of ROM. If we use Ram chip with 128 words we need

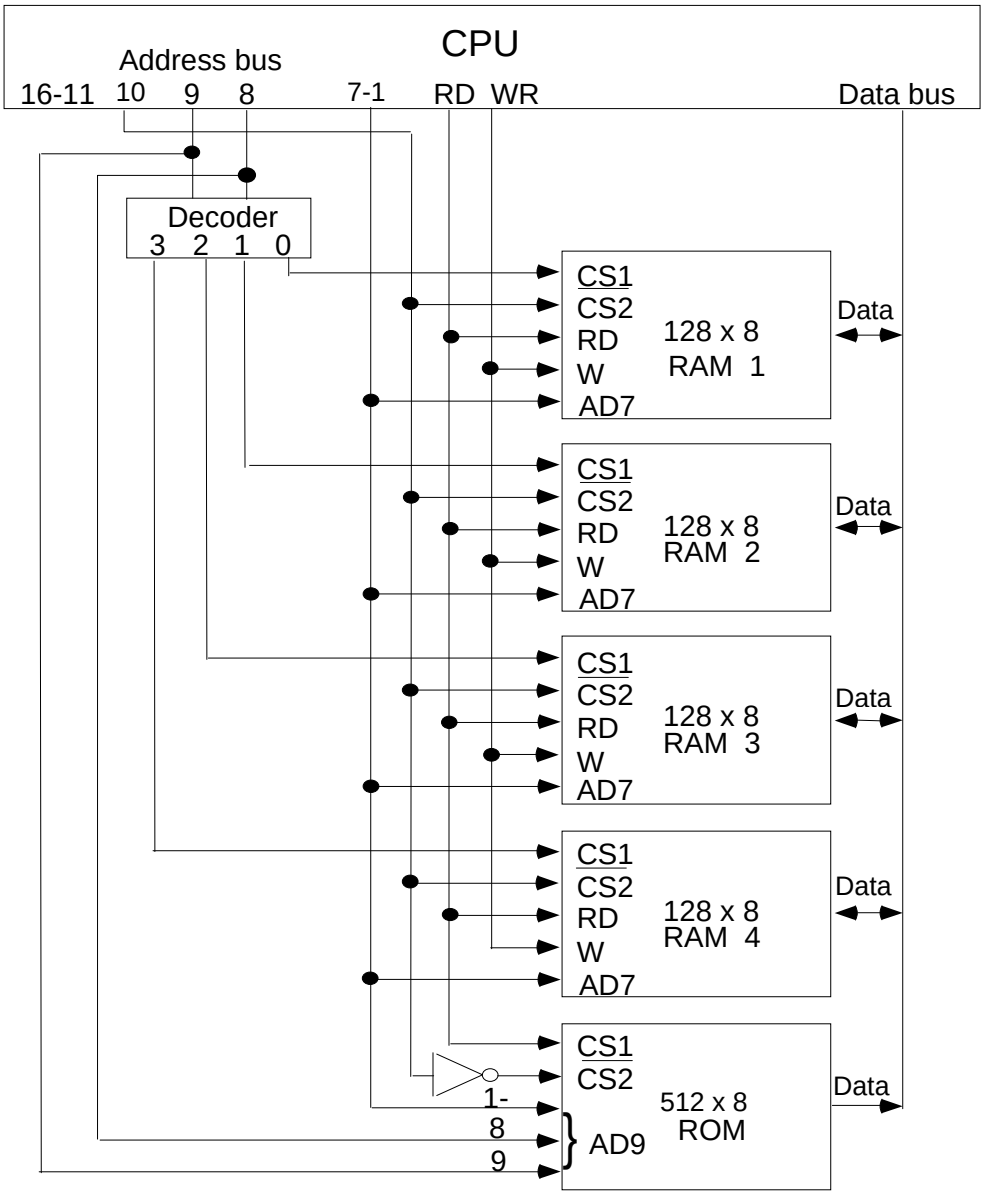
to use 4 Ram chips. For this situation memory address map can be done as given in the table below:

Component	Hexa address	Address bus
		10 9 8 7 6 5 4 3 2 1
RAM 1	0000 - 007F	0 0 0 x x x x x x x
RAM 2	0080 - 00FF	0 0 1 x x x x x x x
RAM 3	0100 - 017F	0 1 0 x x x x x x x
RAM 4	0180 - 01FF	0 1 1 x x x x x x x
ROM	0200 - 03FF	1 x x x x x x x x x

Bus line 8 and nine are used to make distinction between four different RAM's and bus line 10 is used to make distinction between RAM and ROM chips. Address lines 1-7 are used to represent address of RAM chips because their size is 128 words but address lines 1-9 are used to represent address of ROM chip because size of ROM is 512 words.

Memory CPU connection

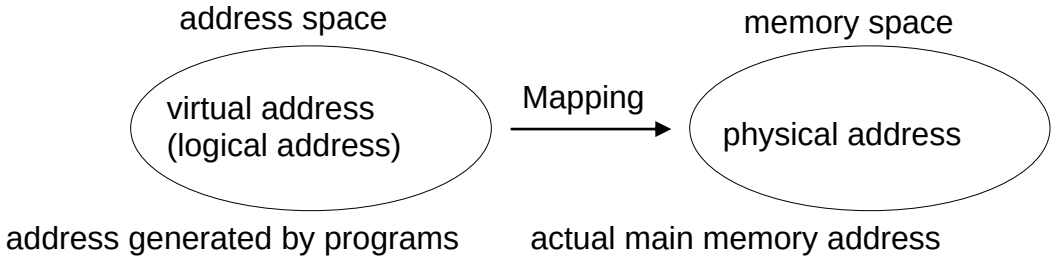
RAM and ROM chips are connected to a CPU through the data and address buses. The low-order lines in the address bus selects the byte within the chips and other lines in the address bus selects a particular chip through its chip select inputs.


CPU
Address bus
16-11
10
9
8
7-1
RD
WR
Data bus
Decoder
3 2 1 0
CS1
CS2
RD
W
AD7
128 x 8
RAM 1
Data
CS1
CS2
RD
W
AD7
128 x 8
RAM 2
Data
CS1
CS2
RD
W
AD7
128 x 8
RAM 3
Data
CS1
CS2
RD
W
AD7
128 x 8
RAM 4
Data
CS1
CS2
1-
8
9
AD9
512 x 8
ROM
Data

Virtual Memory

Virtual memory gives the programmer the illusion that the system has a very large memory, even though the computer actually has a relatively small main memory

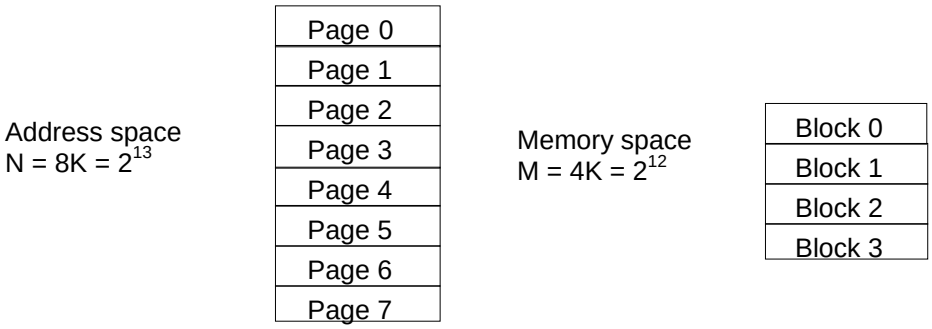
Address Space (Logical) and Memory Space(Physical)



Address Mapping

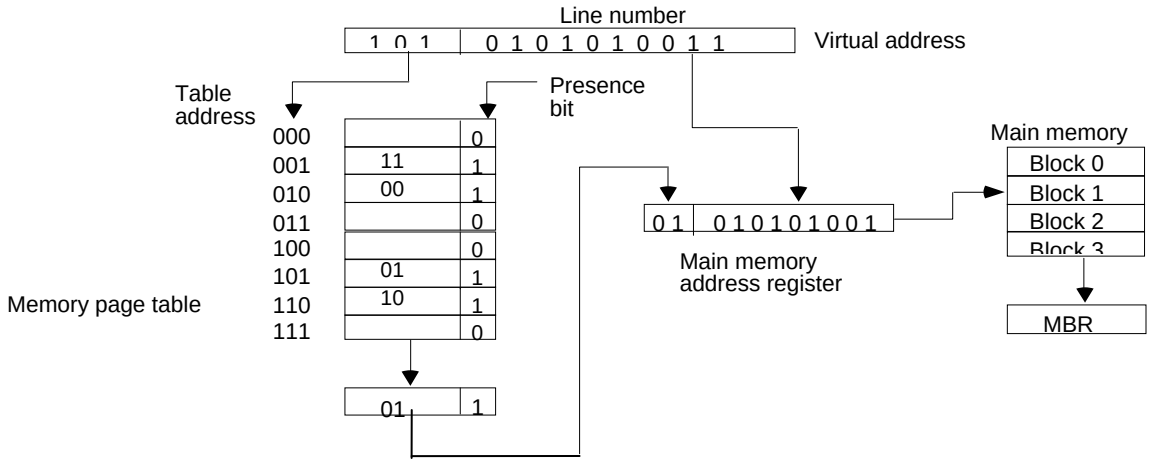
Memory *Mapping Table* for *Virtual Address -> Physical Address*

Address Space and Memory Space are each divided into fixed size group of words called *blocks* or *pages*
Consider 1K words group



Virtual address is divided into two parts page no. and line no. (Offset). Since there are 8 pages, page no. requires three bits and since there are 1k = 1024words in block 10 bits are required in line no.

Page no.

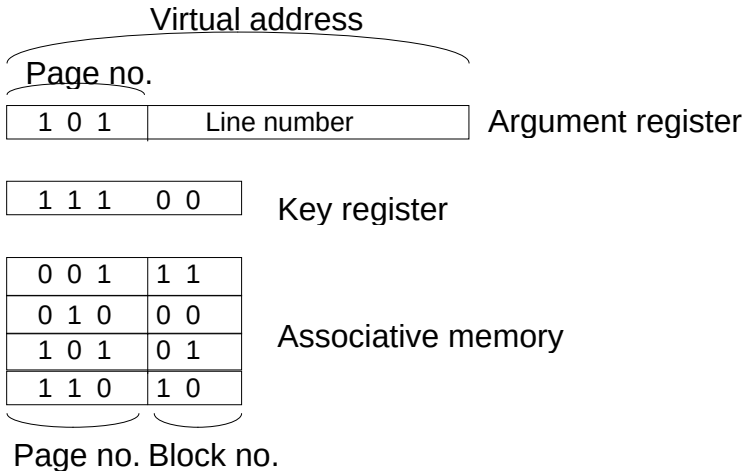


Page Fault

If a page number cannot be found in the Page Table it is called page fault

Associative Memory Page Table

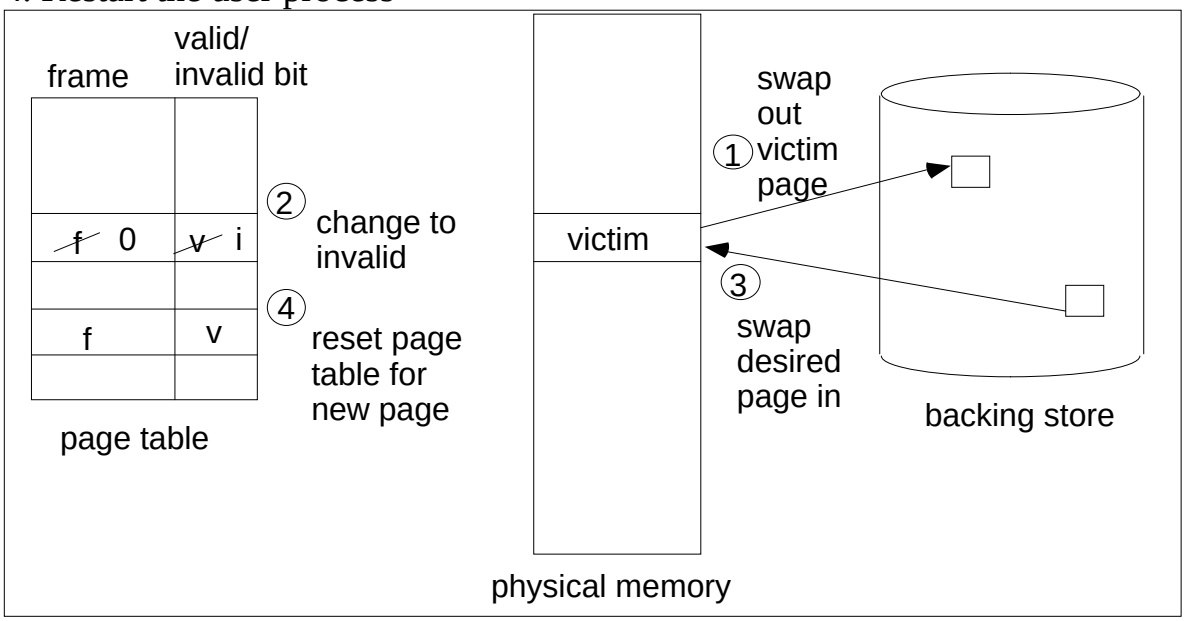
Suppose there are m blocks in memory n number of pages in virtual address space. In this organization page Table requires n entry table in memory. Out of which n-m entries of the table are empty. This is inefficient storage space utilization. More efficient method is use a m-entry page table where page table is made up of an Associative Memory. Associative memory is a memory which stores both data and its actual address. Therefore if page table is implemented by using associative memory it stores both page no. and its associated block no. associative memory is fast because here normally parallel search is dine on the basis of content.



Here key register is used for masking. That is we need to match only first three bits.

Page Replacement
Page Replacement determines which page to displace from memory to make room for an incoming page when no free frame is available

1. Find the location of the desired page on the backing store (secondary storage)
2. Find a free frame
    - If there is a free frame, use it
    - Otherwise, use a page-replacement algorithm to select a *victim* frame
    - Write the victim page to the backing store
3. Read the desired page into the (newly) free frame
4. Restart the user process

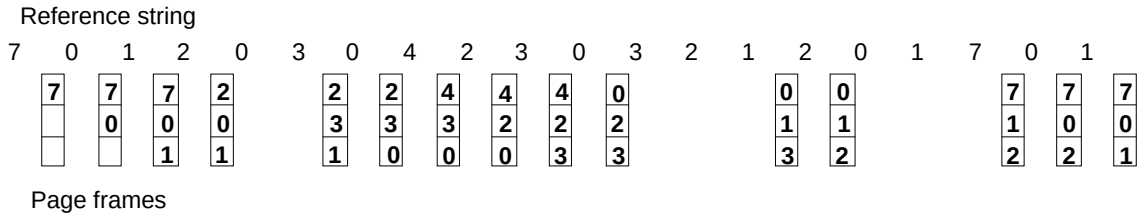


Page Replacement algorithms

First In First Out (FIFO)

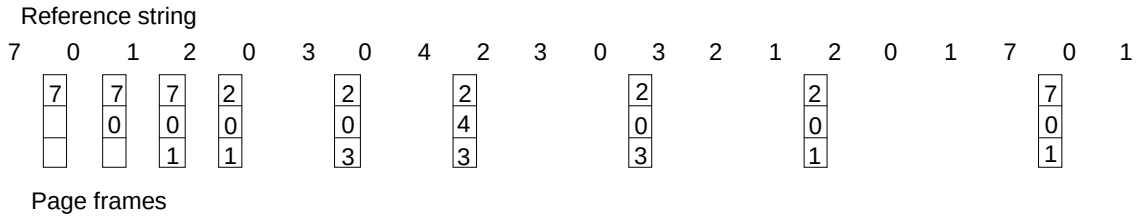
- FIFO algorithm selects the page that has been in memory the longest time using a queue
- Every time a page is loaded, its identification is inserted in the queue

- Easy to implement:: may result in a frequent page fault



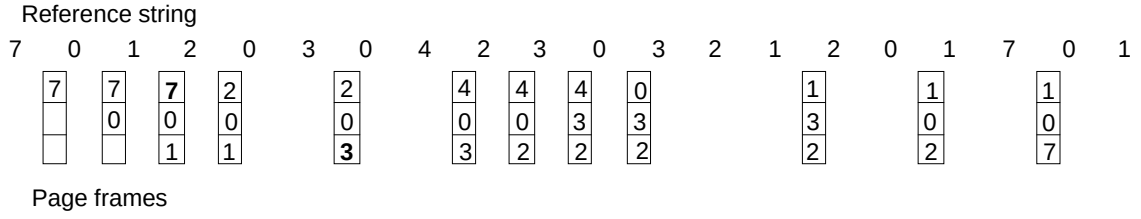
Optimal Replacement (OPT) –

- Lowest page fault rate of all algorithms
- Replace that page which will not be used for the longest period of time



Least Recently Used (LRU)

- OPT is difficult to implement since it requires future knowledge
- LRU uses the recent past as an approximation of near future.
- Replace that page which has not been used for the longest period of time



Memory Management Hardware

Basic Function

- Dynamic Storage Relocation - mapping logical memory references to physical memory references
- Provision for sharing common information stored in memory by different users
- Protection of information against unauthorized access
-

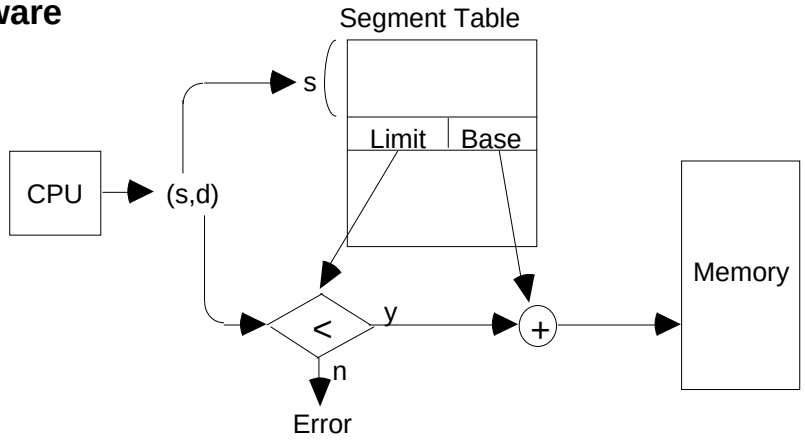
Segmentation

- A memory management scheme where logical address space is a collection of segments
- Each segment has a name and a length
- Address specifies both the segment name and the offset within the segment.
- For simplicity of implementations, segments are numbered.

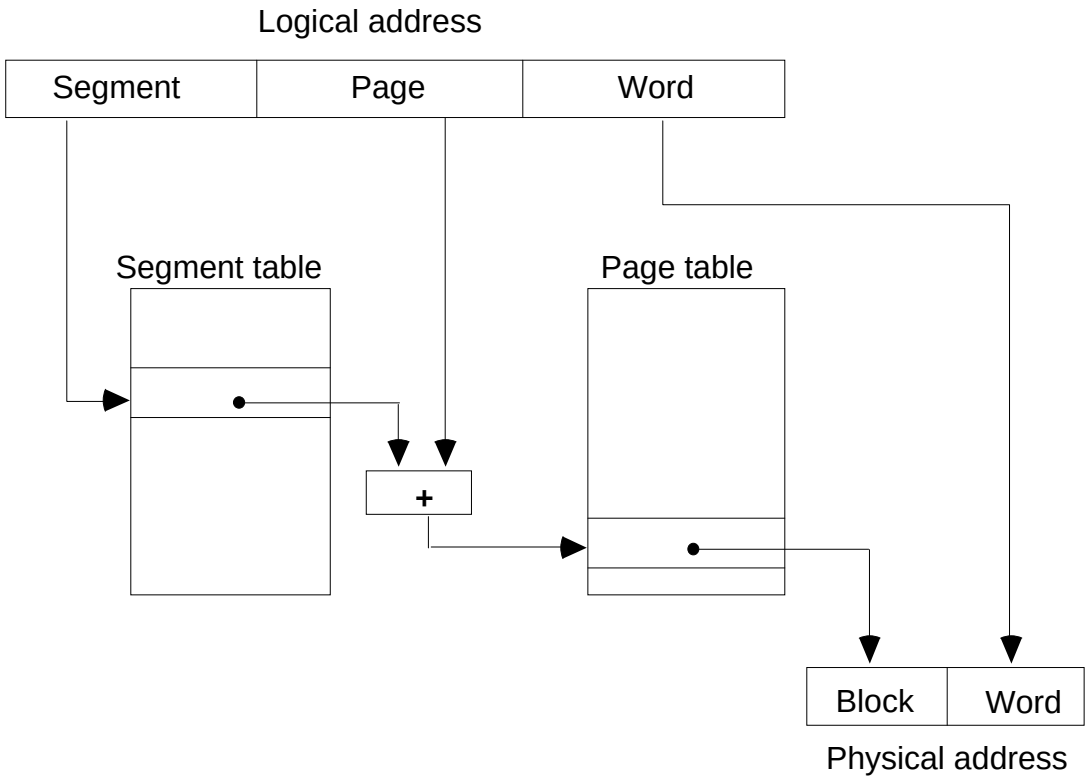
Segment Table

	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

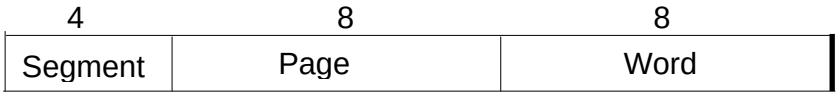
Segmentation Hardware



Segmented Page Mapping

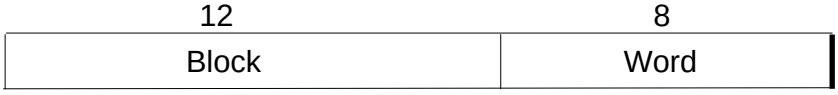


Logical address format: 16 segments of 256 pages each, each page has 256 words

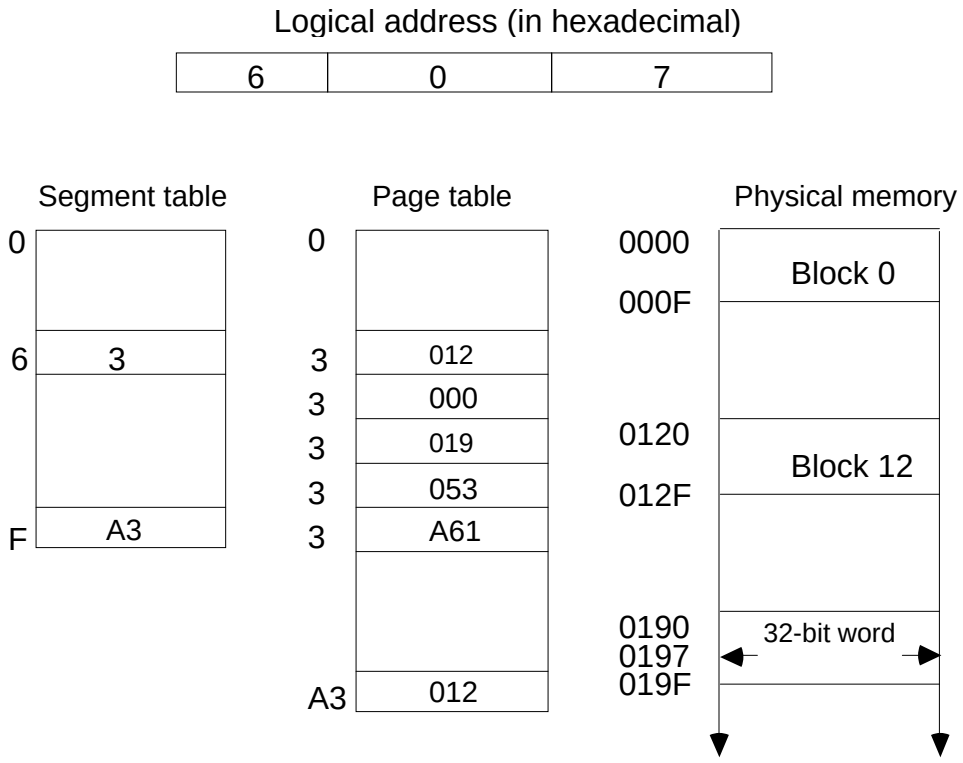


2^2 x 32 Physical memory

Physical address format: 4096 blocks of 256 words each, each word has 32bits



Example:



Physical address = Page table [page no. + segment table [segment no]] + line no (word)

[x] → Content at location x

Memory Protection

Memory protection can be assigned to the physical address or the logical address. The protection of memory through the physical address can be done by assigning to each block in memory a number of protection bits that indicate the type of access allowed to its corresponding block. Every time a page is moved from one block to another it would be necessary to update the block protection bits. A much better place to apply protection is in the logical address space rather than the physical address space. This can be done by including protection information within the segment table or segment register of the memory management hardware.
The content of each entry in the segment table or a segment register is called a descriptor. A typical descriptor would contain, in addition to a base address field, one or two additional fields for protection purposes. A typical format for a segment descriptor is shown in Fig. 12-25. The base address

field gives the base of the page table address in a segmented-page organization or the block base address in a segment register organization. This is the address used in mapping from a logical to the physical address. The length field gives the segment size by specifying the maximum number of pages assigned to the segment. The length field is compared against the page number is the logical address. A size violation occurs if the page number falls outside the segment length boundary. Thus a given program and its data cannot access memory not assigned to it by the operating system.

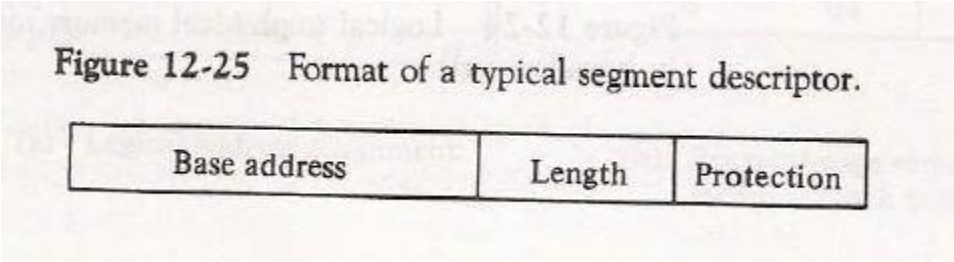


Figure 12-25 Format of a typical segment descriptor.

The protection field in a segment descriptor specifies the access rights available to the particular segment. In a segmented-page organization, each entry in the page table may have its own protection field to describe the access rights of each page. The protection information is set into the descriptor by the master control program of the operating system. Some of the access rights of interest that are used for protecting the programs residing in memory are:

1. Full read and write privileges

2. Read only (write protection)

3. Execute only (program protection)

4. System only (operating system protection)

Full read and write privileges are given to a program when it is executing its own instructions. Write protection is useful for sharing system programs such as utility programs and other library routines. These system programs are stored in an area of memory where they can be shared by many users. They can be read by all programs, but no writing is allowed. This protects them from being changed by other programs.

The execute-only condition protects programs from being copied. It restricts the segment to be referenced only during the instruction fetch phase but not during the execute phase. Thus it allows the users to execute the segment program instructions but prevents them from reading the instructions as data for the purpose of copying their content.

Portions of the operating system will reside in memory at any given time. These system programs must be protected by making them inaccessible to unauthorized users. The operating system protection condition is placed in the descriptors of all operating system programs to prevent the occasional user from accessing operating system segments.