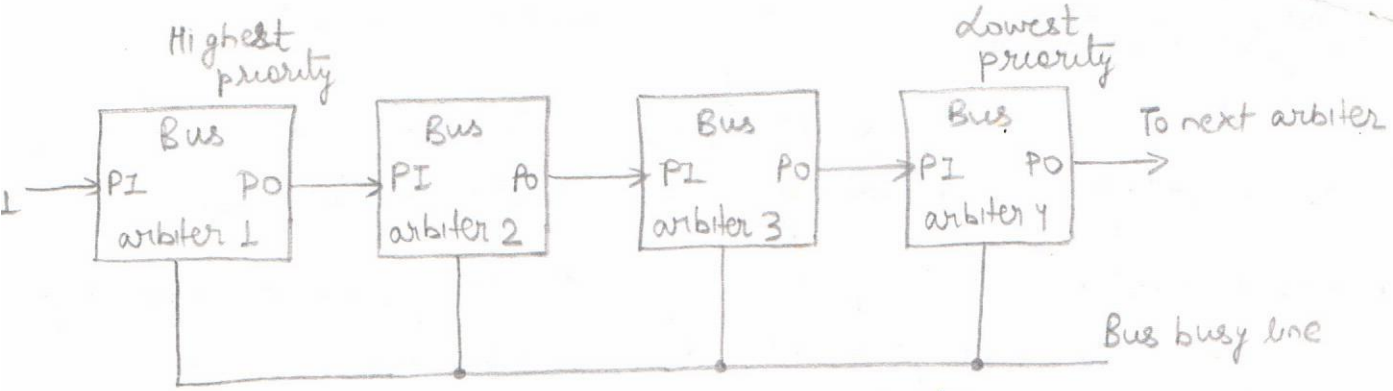
PRIYANKA GOEL
ASST PROF
DEPTT OF CSE

3. Bus Arbitration

* Bus Master is a device attached to the bus that is capable of initiating and controlling communication on the bus.
* Bus Arbitration is the process of determining which competing bus master will be permitted access to the bus.

3.1. Serial Arbitration

* The arbitration procedures service all processor requests on the basis of established priorities.
* A hardware bus priority resolving technique can be establishe by means of a serial or parallel connection of the units requesting control of the system bus.
* In serial arbitration, priority resolving technique is obtained from daisy chain connection of bus arbitration circuits.
* The processors connected to the system bus are assigned priority according to their position along the priority control line.
* The device closest to the priority line is assigned the highest priority.
* When multiple devices concurrently request the use of the bus, the device with the highest priority is granted access to it.



Serial (daisy-chain) arbitration

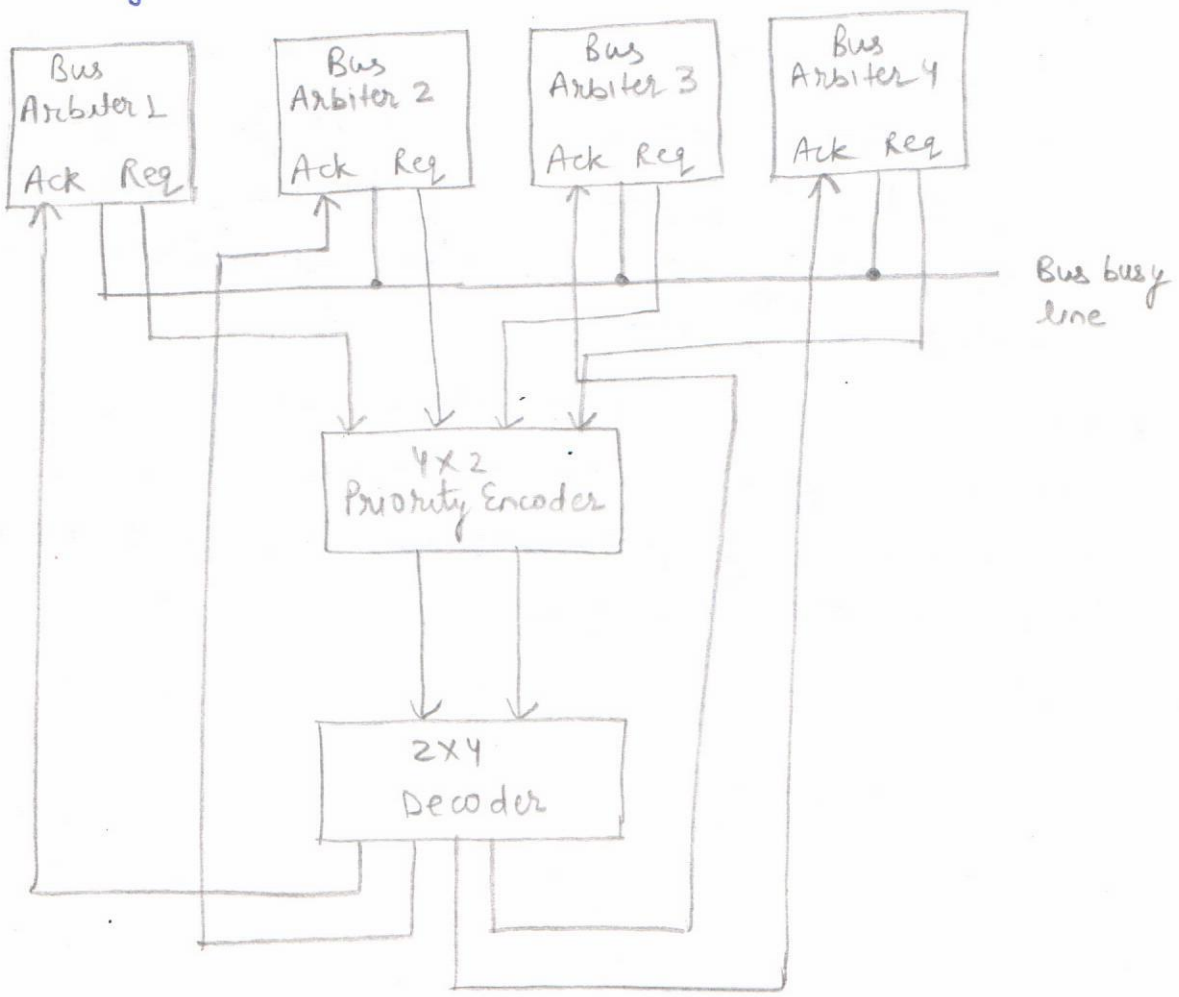
Above figure shows daisy chain connection of four arbiters

* It is assumed that each processor has its own bus arbiter logic with priority-in and priority-out lines.
* The priority out (PO) of each arbiter is connected to the priority in (PI) of the next-lower priority arbiter.
* The PI of the highest priority unit is maintained at a logic 1 value.
* The highest-priority unit in the system will always receive access to the system bus when it requests it.
* The PO output for a particular arbiter is equal to 1 if its PI input is equal to 1 and the processor associated with the arbiter logic is not requesting control of the bus.
* In this way, priority is passed to the next unit in the chain.
* If the processor requests control of the bus and the corresponding arbiter finds its PI input = 1, it sets its PO output = 0.
* Thus, lower priority arbiter receives a 0 in PI and generate a 0 in PO.
* That is, the processor whose arbiter has a PI = 1 and PO = 0 is the one that is given control of the system bus.

* It may be possible that a processor may be in the middle of a bus operation when a higher-priority processor requests the bus.
* In such case, the lower-priority processor must complete its bus operation before it relinquishes control of the bus.
* The bus-busy line (shown in the figure) provides a mechanism for an orderly transfer of control.
* When an arbiter receives control of the bus (as its PI=1 and PO=0), it examines busy line.
* If the line is inactive, it means that no other processor is using the bus. The arbiter activates the busy line & its processor takes control of the bus.
* But if the arbiter finds the busy line active, it means that another processor is currently using the bus. The arbiter keeps examining the busy line while the lower-priority processor that lost control of the bus completes its operation.
* When the bus busy line returns to its inactive state, the higher priority arbiter enables the busy line and its corresponding processor can then conduct the required bus transfers.

2. Parallel Arbitration

* The parallel bus arbitration technique uses an external priority encoder and a decoder.
* Each bus arbiter in this scheme has a bus request output line and a bus acknowledge input line.
* Each arbiter enables the request line when its processor is requesting access to the system bus.
* The processor takes control of the bus if its acknowledge input line is enabled.
* The bus busy line provides an orderly transfer of control (as in daisy-chaining)



* The figure shows the request lines from four arbiters going into a 4x2 priority encoder.
* The output of the encoder generates a 2-bit code which represents the highest priority unit among those requesting the bus.
* Truth table of priority encoder is—

Inputs				Outputs		
I_0	I_1	I_2	I_3	x	y	IST
1	x	x	x	0	0	1
0	1	x	x	0	1	1
0	0	1	x	1	0	1
0	0	0	1	1	1	1
0	0	0	0	x	x	0

$x = I_0' I_1'$

$y = I_0' I_1 + I_0' I_2'$

$(IST) = I_0 + I_1 + I_2 + I_3$

Working :-

The priority encoder implements the priority function. The logic is that if two or more inputs arrive at the same time, the input having the highest priority will take precedence.

The X's in truth table represents dont care conditions.

The interrupt IST is set only when one or more inputs are equal to 1.

If all inputs are 0, IST is cleared to 0 and the other outputs of the encoder are not used, so they are marked with dont care conditions.

The 2-bit code from the encoder output drives a 2×4 decoder which enables the proper acknowledge line to grant bus access to the highest-priority unit.

Dynamic Arbitration Algorithm

The serial and parallel arbitration techniques use a static priority algorithm since the priority of each device is fixed by the way it is connected to the bus.

* In contrast, dynamic priority algorithm gives the system the capability for changing the priority of the devices while the system is in operation.
* Some of them are described here :-

(a) Time-slice algorithm : It allocates a fixed length time slice of bus time that is offered sequentially to each processor, in round robin way. The service given to each system component with this scheme is independent of its location along the bus. No preference is given to any particular device.

(b) Polling : In this technique, the bus grant signal is replaced by a set of lines called poll lines which are connected to all units. These lines are used by the bus controller sequences to define an address for each device connected to the bus. The bus controller sequences through the addresses in a prescribed manner. When a processor that requires access recognizes its address, it activates the bus busy line and then accesses the bus.

* After a number of bus cycles, the polling process continues by choosing a different processor. This polling sequence is normally programmable & thus, selection priority can be altered under program control.

(c) Least Recently Used (LRU) : It gives the highest priority to the requesting device that has not used the bus for the longest interval. The priorities are adjusted after a number of bus cycles according to LRU algorithm. With this procedure, no processor is favoured over any other since the priorities are dynamically changed to give every device an opportunity to access the bus.

(d) First-come, first serve (FCFS) : This scheme serves the request in the order they are received. To implement this algorithm, the bus controller establishes a queue arranged according to the time that the bus requests arrive.

(e) Rotating daisy-chain : This technique is the dynamic extension of the daisy chain algorithm. In this scheme, there is no central bus controller and the priority line is connected from the priority-out of the last device back to the priority-in of the first device in a closed loop. Each arbiter priority for a given bus cycle is determined by its position along the bus priority line from the arbiter whose processor is controlling the bus currently. Once the arbiter releases the bus, it has the lowest priority.

3.4. Types of buses on the basis of transfer

* Data transfers over the system may be synchronous or asynchronous.

(a) **Synchronous bus** : In this bus, each data item is transferred during a time slice known in advance to both source and destination units. Synchronization is achieved by driving both units from a common clock source. Another alternative can be to have separate clocks of approximately same frequency in each unit. Synchronization signals are transmitted periodically in order to keep all clocks in the system in step with each other.

(b) **Asynchronous bus** : In this bus, each data item being transferred is accompanied by handshake control signals to indicate when the data are transferred from the source and received by the destination.

—— x ——

Bus Arbitration can also be performed as —

* Centralized arbitration : a single bus arbiter performs the required arbitration
* Distributed arbitration
  ↳ all devices participate in the selection of the next bus master

—— x ——

* The operations executed on data stored in registers are called micro-operations.
* A micro-operation is an elementary operation performed on the information stored in one or more registers.
* The result of the operation may replace the previous binary information of a register or may be transferred to another register.

e.g shift, count, clear and load

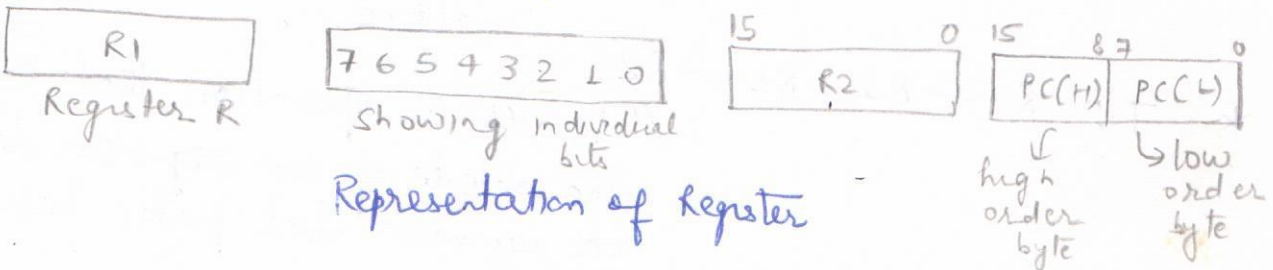
* The symbolic notation used to describe the micro-operation transfers among registers is called register transfer language.

4. Register transfer

4.1. Notation

* Computer registers are designated by capital letters (sometimes followed by numerals).

e.g MAR (Memory Address Register)
↳ that holds an address for the memory unit
PC (Program Counter)
IR(Instruction Register)
R1 (processor register)



Representation of Register

R2 ← R1 // denotes a transfer of the content of register R1 into register R2

It indicates replacement of the content of R2 by the content of R1. The content of the source register R1 doesnot change after the transfer.

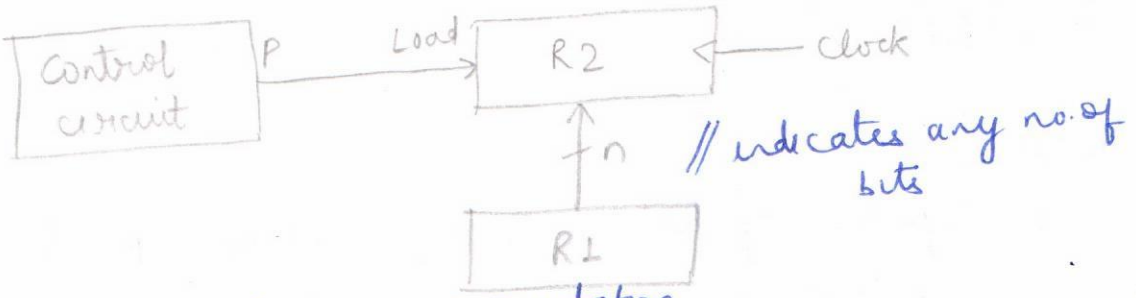
* If the transfer occurs under a predetermined control condition, it can be shown as —

(a) If (P=1) then (R2 ← R1)

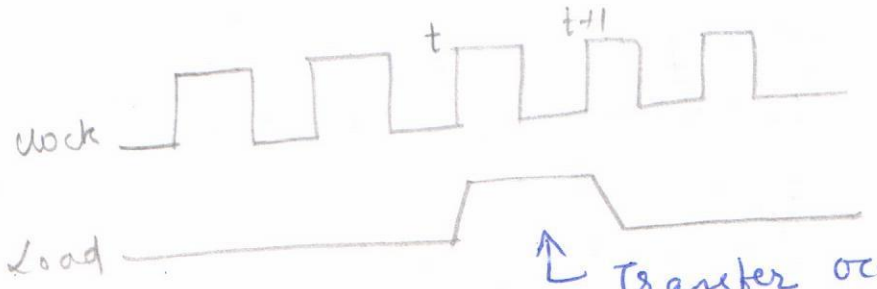
where P is a control signal generated in the control section.

(b) P: R2 ← R1 // transfer will take place only if P=1

where P is a control function as a Boolean variable whose value is equal to 1 or 0.



Hardware representation



Transfer occurs here when P is active, load signal is active

* T: R2 ← R1, R1 ← R2 // denotes an operation that exchanges the contents of two registers during one common clock pulse provided that T=1

Symbol	Description	Example
Letters (and numerals)	denotes a register	MAR, R2
Parentheses ()	denotes a part of register	R2(0-7), R2(L)
Arrows ←	denotes transfer of i/f	R2 ← R1
Comma ,	separates 2 micro-operations	R2 ← R1, R1 ← R2

5. Bus Transfer

* The transfer of information from a bus into one of many destination registers can be done by connecting the bus lines to the inputs of all destination registers and activating the load control of the particular destination register selected.

```
BUS ← C    // content of register C is placed on the bus
R1 ← BUS   // content of the bus is loaded into register
              R1 by activating its load control input.
```

6. Memory Transfer

* The transfer of information from memory word to the outside environment is called read operation.
* The transfer of new information to be stored into the memory is called write operation.
* A m/m word is designated by the letter M.
* Which word among the many available, is selected by the memory address during the transfer.

Thus, it is necessary to specify the address of M when writing memory transfer operations. This can be done by enclosing address in square brackets following the letter M.

Read : DR ← M[AR] // indicates a transfer of i/f into DR from the memory word M selected by the address in AR.

Here DR is Data Register
AR is Address Register

Write : M[AR] ← R1
// indicates transfer of i/f from R1 into memory word M selected by the address in AR.

7. Micro-operations

* A micro-operation is an elementary operation performed with the data stored in registers.
* Following are four types of micro-operations :
  (1) Register transfer micro-operations : transfer binary information from one register to another.
  (2) Arithmetic Microoperations : perform arithmetic operation on numeric data stored in registers.
  (3) Logic micro-operations : perform bit manipulation operations on non-numeric data stored in registers.
  (4) Shift micro-operations : perform shift operations on data stored in registers.

* The register transfer micro-operations donot change the information content when the binary information moves from the source register to destination register.
* The other three types of micro-operations change the information content during the transfer.

7.1. Arithmetic Micro-operations

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

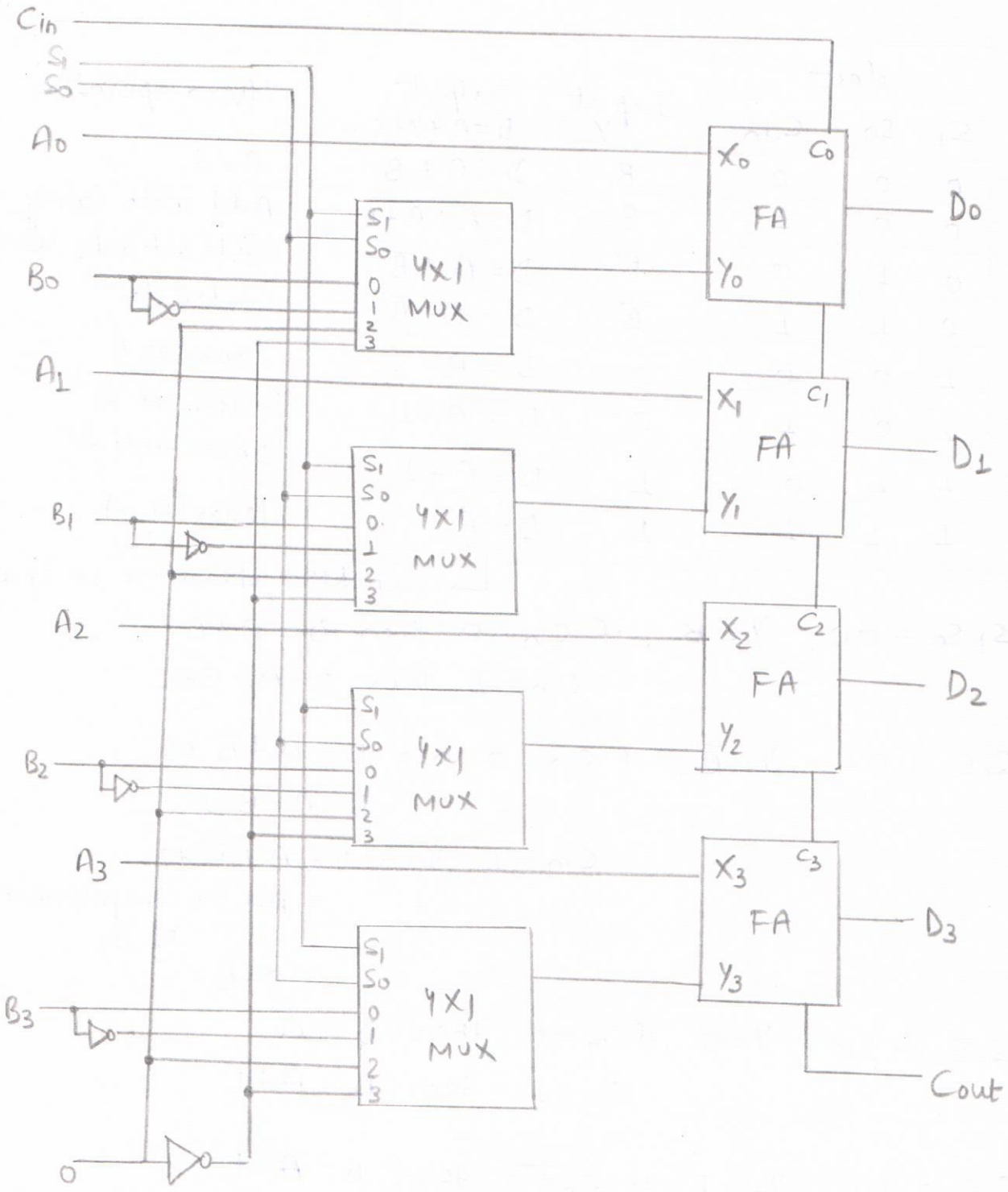
* The arithmetic operations of multiply and divide are valid arithmetic operations but are not included in the basic set of micro-operations.
* In most computers, the multiplication operation is implemented with a sequence of add and shift micro-operations.
* Division is implemented with a sequence of subtract and shift micro-operations.

7.1.1. Construction of Arithmetic Circuit for basic arithmetic micro-operations

* Basic component used is parallel adder
* By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.
* To implement seven arithmetic operations, four full adder circuits (consisting of 4-bit adders) and four multiplexers are used.
* The output of binary adder is calculated as—

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

- A: 4-bit input that goes directly to the X inputs of adder
- Y: 4-bit input that is controlled by selection inputs S_1 and S_0
- C_{in}: carry input which can be equal to 0 or 1


Cin
S1
S0
A0
B0
A1
B1
A2
B2
A3
B3
0
4X1 MUX
4X1 MUX
4X1 MUX
4X1 MUX
FA
FA
FA
FA
X0
C0
Y0
D0
X1
C1
Y1
D1
X2
C2
Y2
D2
X3
C3
Y3
D3
Cout

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

↳ repeated operation so ignored

$S_1 S_0 = 00$, $Y = B$, if $C_{in} = 0$ then $D = A + B$

$C_{in} = 1$ then $D = A + B + 1$

$S_1 S_0 = 01$, $Y = \overline{B}$, if $C_{in} = 0$ then $D = A + \overline{B}$ or

$D = A - B - 1$

$C_{in} = 1$ then $D = A + \overline{B} + 1$

$= A +$ 2's complement of B

$= A - B$

$S_1 S_0 = 10$, $Y = 0$, if $C_{in} = 0$ then $D = A$

$C_{in} = 1$ then $D = A + 1$

$S_1 S_0 = 11$, $Y = 1$, if $C_{in} = 0$ then $D = A - 1$

(a number with all 1's is 2's complement of 1)

2's complement of $0001 = 1111$

$C_{in} = 1$ then $D = A - Y + Y$

$= A$

7.2. Logic micro-operations

* Logic micro-operations specify binary operations for strings of bits stored in registers.
* These operations consider each bit of the register separately and treat them as binary variables.

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

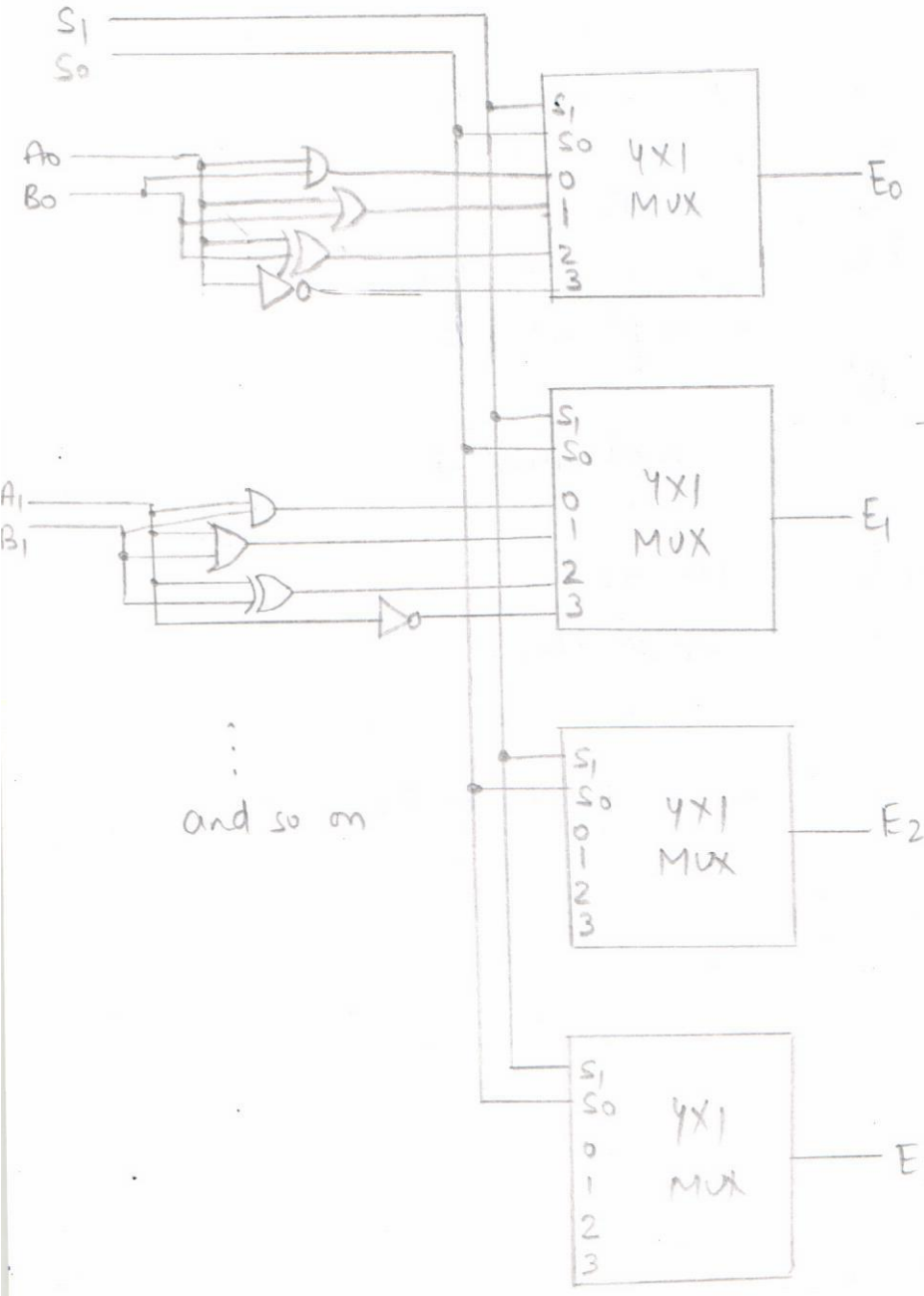
x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

P+Q : R1 ← R2 + R3 , R4 ← R5 ∪ R6

- P+Q : OR operation between two variables of control function
- R2 + R3 : arithmetic add operation
- R5 ∪ R6 : OR operation between registers R5 and R6

* The logic micro-operations are rarely used is scientific computations, but they are very useful for bit manipulation of binary data and for making logical decisions.

7.2.1 Hardware implementation for 4 logic micro-operations



S_1	S_0	Output	Oper^n
0	0	$E = A \wedge B$	ADD
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Comple-ment

7.3. Shift micro-operations

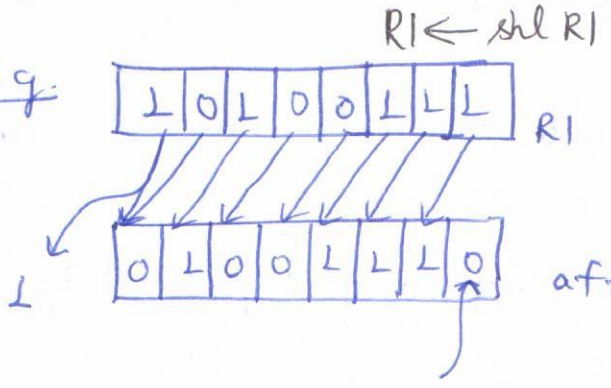
* These are used for serial transfer of data.
* They are also used in conjunction with arithmetic, logic and other data-processing operations.
* The contents of a register can be shifted to the left or the right.
* There are three types of shifts —
  a) Logical
  b) Circular
  c) Arithmetic

7.3.1 Logical shift

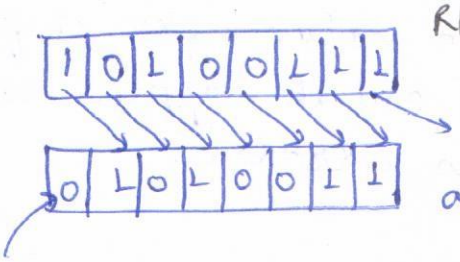
```
R1 ← shl R1     // specify 1-bit shift to the left of content of register R1
R2 ← shr R2     // specify 1-bit shift to the right of content of register R2
```

* The bit transferred to the end position through the serial input is assumed to be 0 during logical shift.

e.g.



after one bit shifting to left



after one bit shifting to right

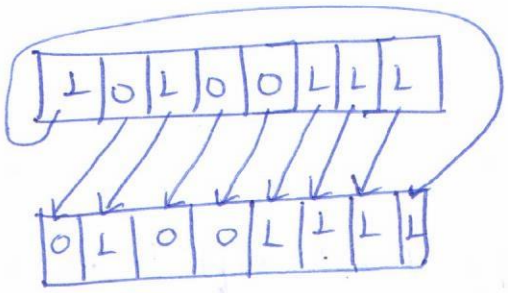
7.3.2 Circular shift

* also known as rotate operation
* It circulates the bits of the register around the two ends without loss of information.
* This can be done by connecting the serial output of the shift register to its serial input.

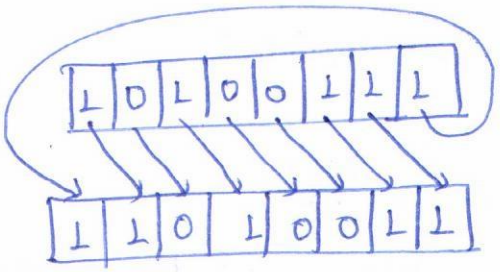
R ← cil R // circular shift left register R

R ← cir R // circular shift right register R

e.g.



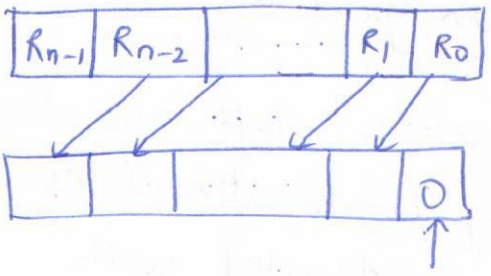
R ← cil R



R ← cir R

7.3.3 Arithmetic shift

* This micro-operation shifts a signed binary number to the left or right.
* An arithmetic shift-left multiplies a signed binary number by 2.
* Arithmetic shift-left inserts a 0 into R_0 and shifts all other bits to the left.
* The initial bit of R_{n-1} is lost and replaced by the bit from R_{n-2}.
* If the bit in R_{n-1} changes in value after shift, then sign reversal has occured and it happens when multiplication by 2 causes an overflow.



$R \leftarrow ashl\ R$

i.e. overflow occurs if, before the shift, $R_{n-1} \neq R_{n-2}$

* An overflow flip-flop V_s can be used to detect an arithmetic shift-left overflow:

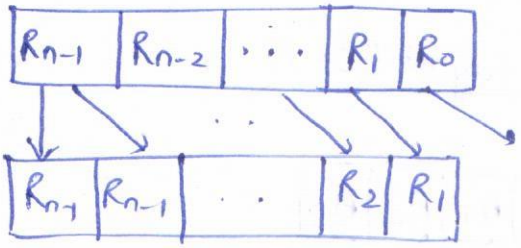
$$V_s = R_{n-1} \oplus R_{n-2}$$

If $V_s = 0$ then there is no overflow

$V_s = 1$ then there is an overflow and sign reversal has occured after the shift.

* Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2.
(The leftmost bit in a register holds the sign bit and the remaining bits hold the number)

* The arithmetic shift right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right.

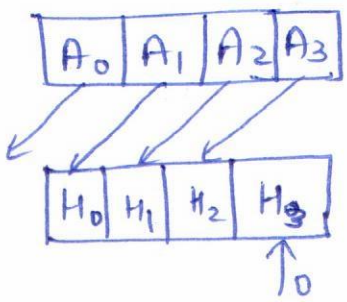


$R \leftarrow ashr\ R$

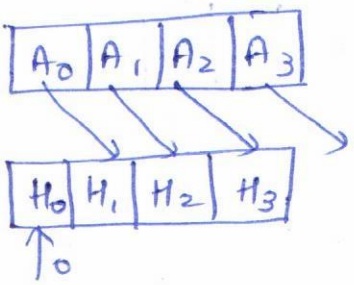
7.3.4
Hardware Implementation of shift micro-operations

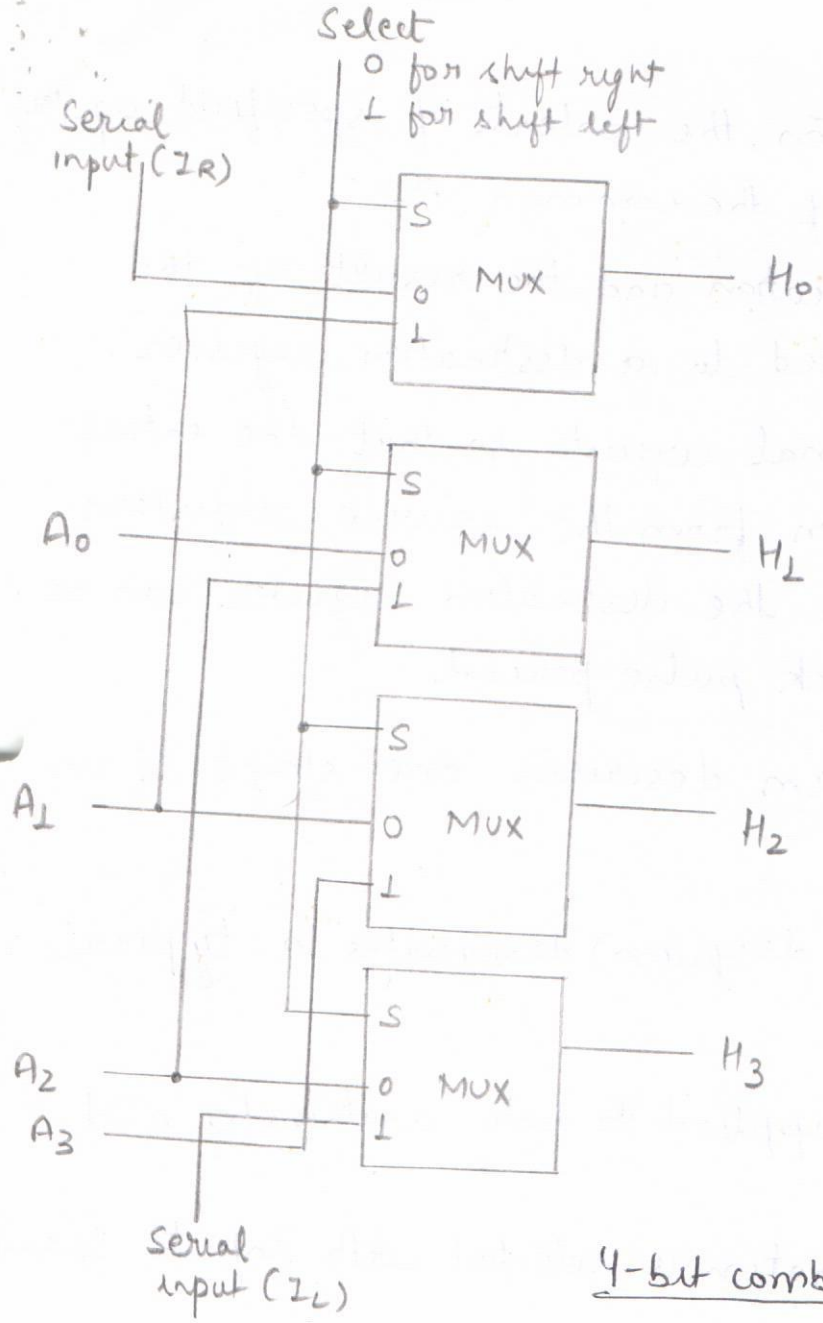
- In a processor unit with many registers, shift operations can be implemented efficiently using combinational circuit.
- Such a combinational circuit can be constructed with multiplexers.
- The data inputs ~~through~~ are A_0 through A_3 (four bits) and the four data outputs are H_0 through H_3.
- There are two serial inputs — one for shift left (I_L) and other for shift right (I_R)
- When the selection input $S=0$, the input data are shifted right.
- When the selection input $S=1$, the input data are shifted left.
- A data shifter with n data inputs and outputs requires n multiplexers
- The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.

Shift left ($S=1$)



Shift right ($S=0$)





Functional Table

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

4-bit combinational shifter

7.4. Arithmetic Logic Shift Unit

* Instead of having individual registers performing the micro-operations directly, computer systems employ a number of storage dev registers connected to a common operational unit called an arithmetic logic unit (ALU).

To perform a micro-operation, the contents of specified registers are placed in the inputs of the common ALU.

* The ALU performs an operation and the result of the operation is then transferred to a destination register.

* The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period.

* Following block diagram describes one stage of an arithmetic logic unit.

* The subscript i (in the diagram) designates a typical stage.

* Inputs A_i and B_i are applied to both arithmetic and logic units.

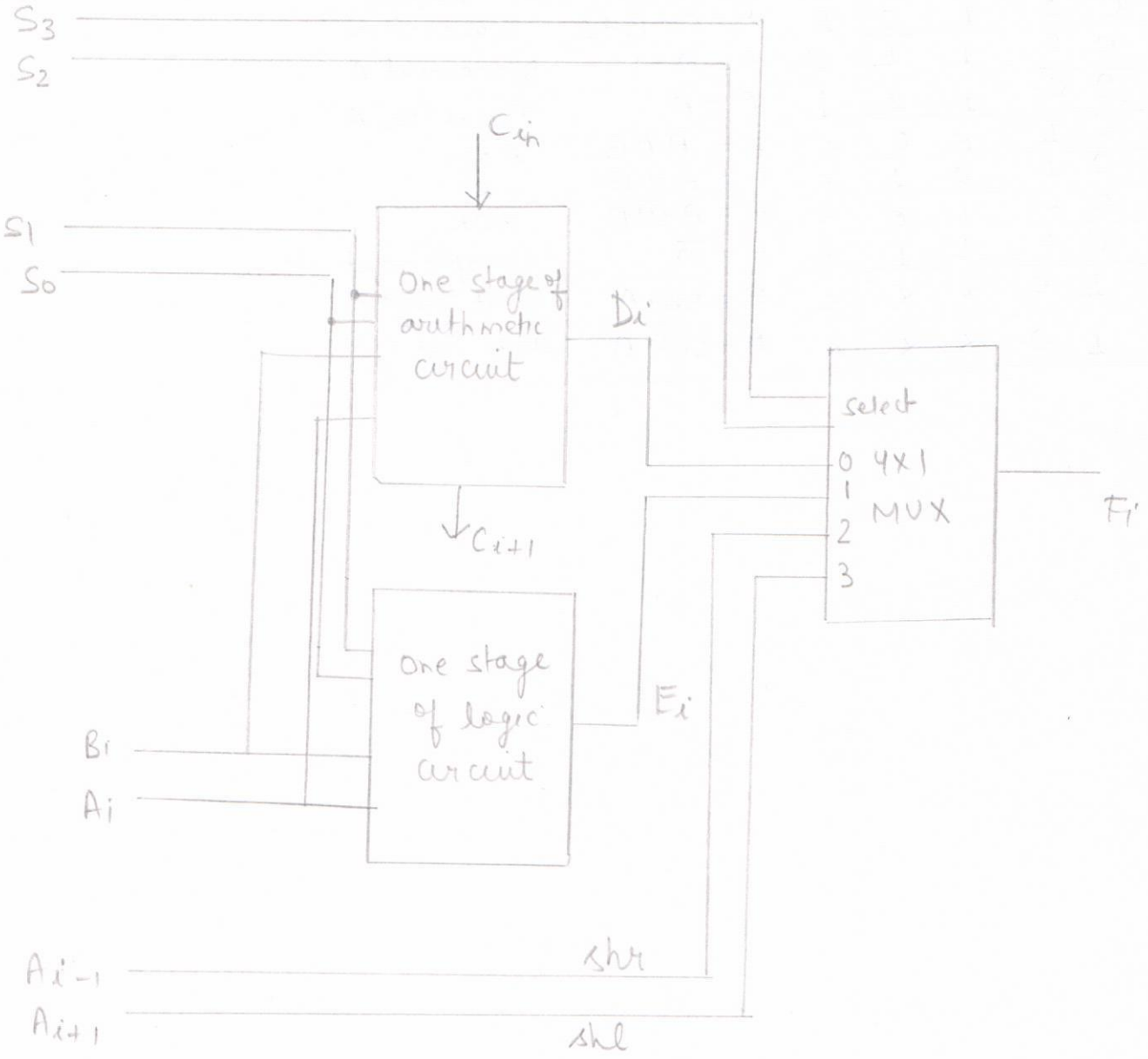
* A particular micro-operation is selected with inputs S_1 and S_0.

* A 4X1 multiplexer at the output chooses between an arithmetic output in E_i and logic output in H_i.

* The type of micro-operation ~~is~~ is selected with inputs S_3 and S_2.

* The other two data inputs to the multiplexer receive inputs A_{i-1} for shift right operation and A_{i+1} for shift left operation.

* This circuit must be repeated n times for an n-bit ALU.
* The output carray C_{i+1} for given arithmetic stage must be connected to the input carry C_i of the next stage in sequence.
* The input carray C_{in} to the first stage, provides a selection variable for the arithmetic operations.

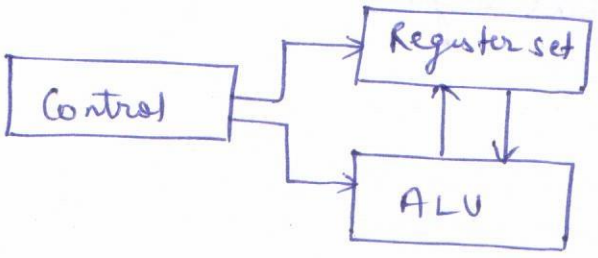


One stage of ALU

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

* CPU is that part of computer system which performs the bulk of data processing operations.

It is made up of three parts —



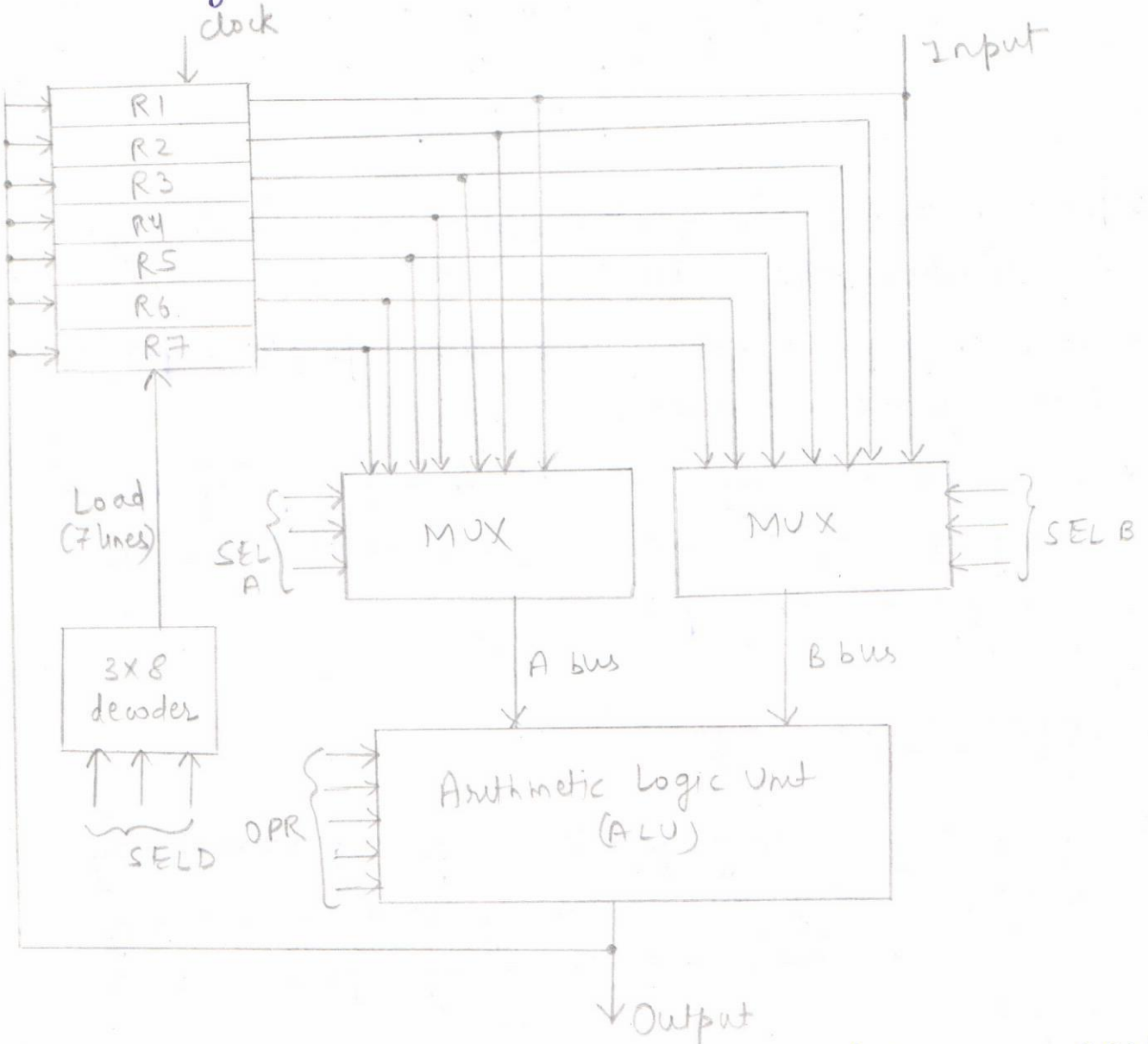
* The register set stores intermediate data used during the execution of the instructions.
* The ALU performs the required micro-operations for executing the instructions.
* The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

12. General Register Organization

* During any ~~memory~~ operation, referring to memory locations is very time consuming because memory access is the most time-consuming operation in a computer.
* Thus, it is convenient and more efficient to store these intermediate values in processor registers.
* When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system.

* These registers communicate with each other not only for direct data transfers but also while performing various micro-operations.

Following is the bus organization for seven CPU registers:



* The output of each register is connected to two multiplexers to form the two buses A and B.
* The selection lines in each multiplexer (i.e. SEL A and SEL B) select one register or the input data for the particular bus.

* The A and B buses form the inputs to a common ALU.
* The operation is selected using OPR control lines which determines the arithmetic or logic micro-operation to be performed.
* The result of the micro-operation is available for output data and also goes into the inputs of all the registers.
* The register that receives the information from the output bus is selected by the decoder.
* The decoder activates one of the register load inputs thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.

e.g. to perform operation: $R1 \leftarrow R2 + R3$

Following selector inputs must be provided with binary information by the control unit :-

(a) MUX A selector (SEL A) :- to place contents of R2 into bus A

(b) MUX B selector (SEL B) :- to place contents of R3 into bus B

(c) ALU operation selector (OPR): to provide arithmetic addition A + B

(d) Decoder destination register (SEL D): to transfer the content of the output bus into R1.

6

2.1.

Control Word

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

3	3	3	5
SELA	SELB	SELD	OPR

Encoding of Register Selection Fields

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

* When SELA or SELB is 000, the corresponding multiplexer selects the external input data.
* When SELD = 000, no destination register is selected but the contents of the output bus are available in the external output.

Encoding of ALU operations

The OPR field has five bits and each operation is designated with a symbolic name.
The codes of operation have been taken from ALU Design

OPR select	operation	Symbol
00000	Transfer A	TSFA
0000L	Increment A	INCA
000L0	Add A+B	ADD
00L0L	Subtract A-B	SUB
00LL0	Decrement A	DECA
0L000	AND A and B	AND
0L0L0	OR A and B	OR
0110 0	XOR A and B	XOR
0LLL0	Complement A	COMA
L0000	Shift right A	SHRA
LL000	Shift left A	SHL A

Few examples :

Micro-operation	Symbolic Designation				Control Word
	SELA	SELB	SEL D	OPR	
R1 ← R2-R3	R2	R3	R1	SUB	010 011 001 00101
R4 ← R4 ∨ R5	R4	R5	R4	OR	L00 L0L L00 0L0L0
R6 ← R6+1	R6	-	R6	INCA	1L0 000 LL0 0000L
R7 ← R1	R1	-	R7	TSFA	00L 000 LLL 00000
Output ← R2	R2	-	None	TSFA	010 000 000 00000
Output ← Input	Input	-	None	TSFA	000 000 000 00000
R4 ← shl R4	R4	-	R4	SHL A	L00 000 L00 LL000
R5 ← 0	R5	R5	R5	XOR	101 101 101 01100

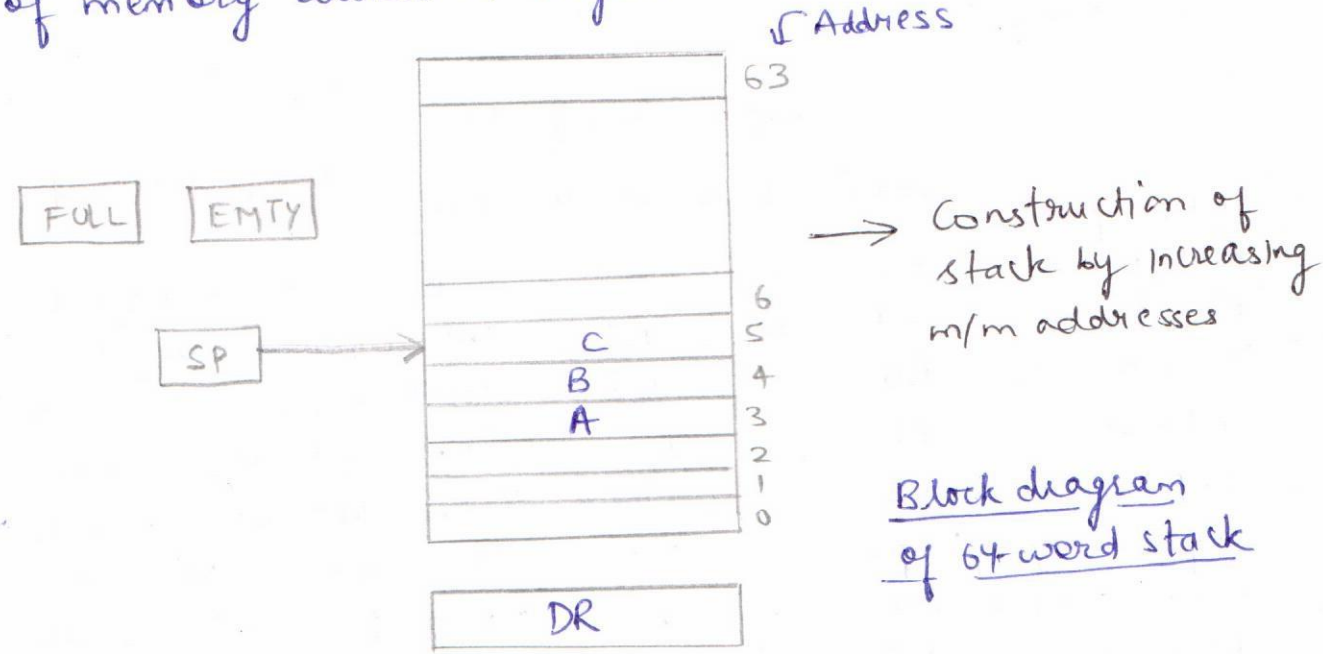
↳ Operation of clearing register value to 0

Stack Organization

* The stack in digital computers is essentially a memory unit with an address register that can count only (after an initial value is loaded into it).

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

* A stack can be placed in a portion of large memory or it can be organized as a collection of finite number of memory words or registers.



Block diagram of 64 word stack

* The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack.

* In 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$.

$SP \leftarrow SP + 1$

$M[SP] \leftarrow DR$

If (SP=0) then (FULL ← 1) Check if stack is full
EMTY ← 0 Mark the stack not empty

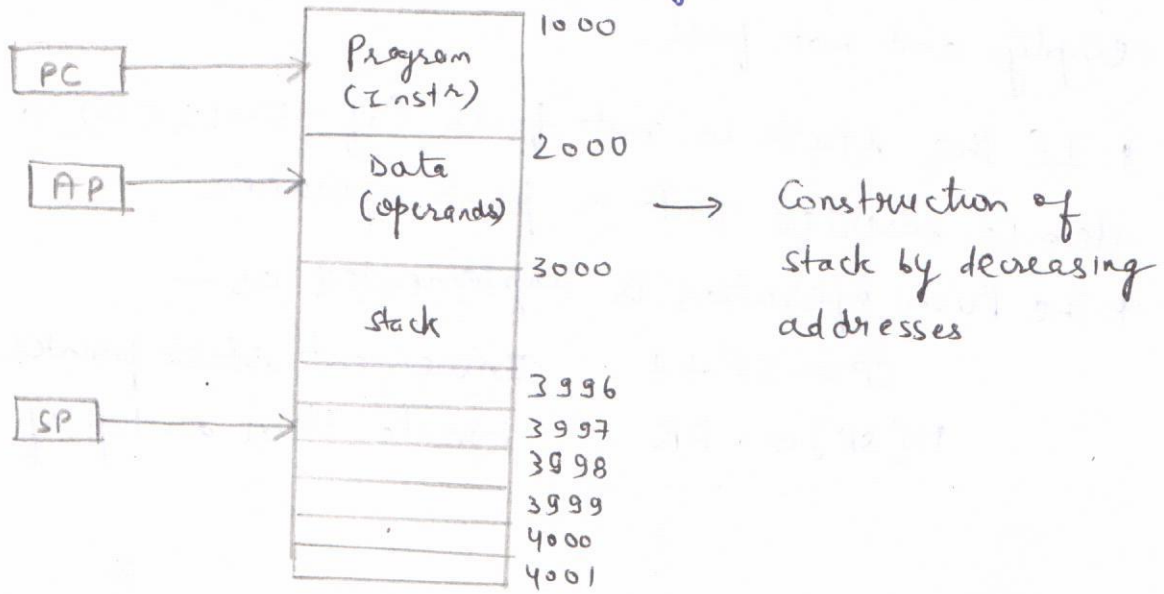
Note :- The first item is stored at the address 1. The last item is stored at address 0. If SP reaches 0, the stack is full of items so FULL is set to 1.

* A new item is deleted from the stack if the stack is not empty (if EMTY=0).
* The Pop operation is implemented as –

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

Note :- An erroneous operation will result if the stack is pushed when FULL=1 or popped when EMTY=1.

Now let us look at other way:



* So, value of SP cannot exceed a number greater than 63 (111111 in binary)
* When 63 is incremented by 1, the result is 0 since 111111 + 1 = 1000000 in binary but SP can accommodate only six least significant bits.
* Similarly when 000000 is decremented by 1, the result is 111111.
* To insert a new item, the stack is pushed by incrementing SP and writing a word in the next higher location in the stack.
* The one-bit register FULL is set to 1 when the stack is full and the one-bit register EMTY is set to 1 when the stack is empty of items.
* DR is the data register that holds the binary data to be written into or read out of the stack.
* Initially, SP is cleared to 0, EMTY is set to 1 and FULL is cleared to 0 so that SP points to the word at address 0 and the stack is marked empty and not full.
* If the stack is not full (if FULL=0), a new item is inserted with a push operation.
* The Push operation is implemented as —

SP ← SP + 1	Increment stack pointer
M[SP] ← DR	Write item on top of stack

51

In this case, PUSH operation is implemented as :-

$SP \leftarrow SP - 1$

$M[SP] \leftarrow DR$

and POP operation is implemented as -

$DR \leftarrow M[SP]$

$SP \leftarrow SP + 1$

8.1. Reverse Polish Notation

* A stack organization is very effective for evaluating arithmetic expressions.
* There are three ways of writing arithmetic expressions :-
  a) Infix notation : A + B
  b) Prefix / Polish notation : + A B
  c) Postfix / Reverse Polish notation : A B +
* The common infix notation poses certain difficulties while evaluating an arithmetic expression on a digital computer.
* So reverse Polish notation is used for stack manipulation.

13.2. Converting an Infix expression into Reverse Polish notation

POLISH (Q, P)
where Q is an arithmetic expression written in infix notation and P is its equivalent postfix expression.

1 Push "(" onto STACK and add ")" to the end of Q.
2 Scan Q from left to right and repeat steps 3 to 6 for each element of Q until the STACK is empty:
3 If an operand is encountered, add it to P
4 If a left parenthesis is encountered, push it onto STACK
5 If an operator ⊗ is encountered then:
(a) Repeatedly pop from STACK and add to P each operator (on TOP of STACK) which has same precedence as or higher precedence than ⊗.
(b) Add ⊗ to stack.
[End of if]
6 If a right parenthesis is encountered then:
(a) Repeatedly Pop from ~~sta~~ STACK and add to P each operator (on the TOP of STACK) until a left parenthesis is encountered.
(b) Remove the left parenthesis [Do not add left parenthesis to P]
[End of If structure]
[End of step 2 loop]
7 Exit

e.g. 1 Q: A+(B*C-(D/E↑F)*G)*H

Symbol scanned	Stack	Expression P
A	(	A
+	(+	A
B	(+(	AB
*	(+(	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
↑	(+(-(/↑	ABC*DE
F	(+(-(/↑	ABC*DEF
)	(+(-	ABC*DEF↑/
*	(+(-*	ABC*DEF↑/
G	(+(-*	ABC*DEF↑/G
)	(+	ABC*DEF↑/G*-
*	(+*	ABC*DEF↑/G*-
H	(+*	ABC*DEF↑/G*-H
)		ABC*DEF↑/G*-H*+

Evaluation of a Postfix Expression

This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

1 Add a right parenthesis ")" at the end of P.

2 Scan P from left to right and repeat steps 3 and 4 for each element of P until the sentinel ")" is encountered.

3 If an operand is encountered, put it on STACK

4 If an operator is encountered, then :

(a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.

(b) Evaluate $B \otimes A$

(c) Place the result of (b) back on STACK.

[End of if structure]

[End of step 2 loop]

5 Set the VALUE equal to top element on STACK.

6 Exit

eg. 1 P: 5, 6, 2, +, *, 12, 4, /, -

Symbol scanned	STACK
5	5
6	5, 6,
2	5, 6, 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
-	37

14. Instruction formats

* A computer usually have a variety of instruction code formats.
* It is the function of control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction.
* The format of an instruction is depicted is a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register.
* The bits of the instruction are divided into groups called fields.
* The most common fields found in instruction formats are :-
  (a) An operational code field that specifies the operation to be performed.
  (b) An address field that designates a memory address or a processor register.
  (c) A mode field that specifies the way the operand or the effective address is determined. This field specifies a variety of alternatives for choosing the operands from the given address.
* Computers may have instructions of several different lengths containing varying no. of addresses.
* The number of address fields in the instruction format of a computer depends on the internal organization of its registers.

* Most computers fall into one of three types of CPU organizations :-
  a) Single accumulator organization
  b) General register organization
  c) Stack organization

14.1. Single Accumulator Organization

* All operations are performed with an implied accumulator register.
* The instruction format in this type of computer uses one address field.

e.g the instruction that specifies an arithmetic addition is defined as :

```
ADD X
```

where X is the address of the operand.
The ADD instruction in this case results in the operation $AC \leftarrow AC + M[X]$.

- AC → accumulator register
- M[X] → memory word located at address X

14.2. Stack Organization

* Computers with stack organization would have PUSH and POP instructions which require an address field.

* Thus the instruction PUSH X will push the word at address X to the top of stack.
* The stack pointer is updated automatically.
* Operations-type instructions do not need an address field in stack-organized computers, because the operation is performed on the two items that are on the top of stack.

e.g ADD

This operation will pop two top numbers from the stack, add the numbers and push the sum into the stack.

* There is no need to specify operands with an address field since all operands are implied to be in the stack.

14.3. General Register Organization

* These computers employ two or three address fields in their instruction format.
* Each address field may specify a processor register or a memory word.

e.g ADD R1, X

would specify the operation $R1 \leftarrow R1 + M[X]$.

* It has two address fields — one for register R1 and the other for the memory address X.

To illustrate the influence of the number of addresses on computer programs, let us take an example:

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

We will evaluate this arithmetic expression using zero, one, two or three address instructions.

14.3.1

Three-address instructions

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

It is assumed that the computer has two processor registers R1 and R2. The symbol M[A] denotes the operand at memory address symbolized by A.

Advantage :- It results in short programs when evaluating arithmetic expressions.

Disadvantage :- The binary-coded instructions require too many bits to specify three addresses.

14.3.2

Two address instructions

MOV R1, A	R1 ← M[A]
ADD R1, B	R1 ← ~~M[A]~~ R1 + M[B]
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

* Here each address field can specify either a processor register or a memory word.
* The MOV instruction moves or transfers the operands to and from memory and processor registers.
* The first symbol listed in instruction is assumed to be both a source and destination where the result of the operation is transferred.

14.3.3
One-address instructions

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

* These instructions use an implied accumulator (AC) register for all data manipulation.
* All operations are done between the AC register and a memory operand.
* T is the address of a temporary memory location required for storing the intermediate result.

14.3.4
Zero-address instructions

* A stack organized computer doesnot use an address field for the instructions ADD and MUL.

~~8760~~

* The PUSH and POP instructions need an address field to specify the operand that communicates with the stack.

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

* Here TOS represents Top of Stack

15. Addressing Modes

* Programs are normally written in a high-level language, which enables the programmer to use constants, local and global variables, pointers and arrays.
* When translating a high level language program into assembly language, the compiler must be able to implement these constructs using the facilities provided in the instruction set of the computer in which the program will be run.
* The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes.
* Following are addressing modes that will be discussed:-

(1) Implied mode
(2) Immediate mode
(3) Register mode
(4) Direct mode
(5) Indirect mode —— (a) Through Register
(b) Through memory location
(6) Relative address mode
(7) Indexed mode
(8) Base register mode
(9) Auto increment / Auto decrement mode

15.1. Implied Mode

* In this mode, the operands are specified implicitly in the definition of the instruction.

  e.g. "complement accumulator" is an implied mode instruction because the operand in the accumulator register is ~~as~~ implied in the definition of instruction.

* All register reference instructions that use an accumulator are implied mode instructions.

* Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

15.2. Immediate mode

* In this mode, the operand is given explicitly in the instruction.

* This mode is used to represent address and data constants in assembly language.

  e.g. MOV $200_{immediate}$, R0

  places the value 200 in register R0.

* Generally, the value to be used is represented as (in assembly language):

  MOV #200, R0

  ↳ used to indicate that this value is to be used as an immediate operand

15.3. Register mode

In this mode, the operands are in registers that reside within the CPU. A k-bit field can specify any one of 2^k registers.

15.4. Register Indirect Mode

* In this mode, the instruction specifies a register in the CPU whose contents give the address of the operand in memory.
* Before using a register indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction.

Advantage :- The address field of the instruction uses a fewer bits to select a register than would have been required to specify a memory address directly.

* The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory.
* Sometimes the value given in the address field is the address of the operand but sometimes it is just an address from which the address of the operand is calculated.
* The effective address is defined to be the memory address obtained from the computation given by the given addressing mode.

15.5. Direct Address Mode

In this mode, the address of operand is given explicitly in the instruction and the operand resides in a memory location. It is also called absolute mode.

In a branch type instruction, the address field specifies the actual branch address.

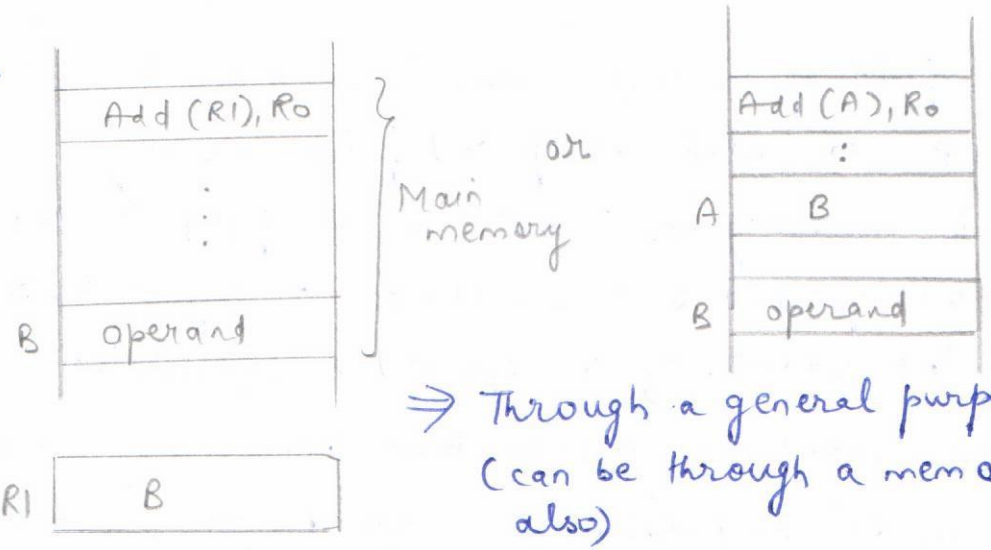
e.g. MOV M[A], R2

uses register mode and direct mode.

15.6. Indirect Address Mode

* In this mode, the address field of the instruction gives the address where the effective address is stored in memory.
* Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

e.g.



⇒ Through a general purpose register (can be through a memory location also)

In above example, to execute the Add instruction, the processor uses the value B, which is in register R1, as the effective address of the operand

1.5.7. Relative Address Mode

* In this mode, the content of the program counter is added to the address part of the instruction in order to obtain the effective address.
* The address part of the instruction is usually a signed number (in 2's complement) which can be either positive or negative.
* When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction.
* It can be represented as X(PC) where X is the offset value to be added to the address given by PC. (unlike indexed mode, which is represented by X(Ri)).

e.g.

Address	Contents	
	Move N, R1	} Initialization
	Move #NUM1, R2	
	Clear R0	
LOOP 1000	Add (R2), R0	
1004	Add #4, R2	
1008	Decrement R1	
1012	Branch>0 LOOP	
	Move R0, SUM	

Suppose in above example, the Relative Mode is used to generate the branch target address LOOP in the Branch instruction of the program.

Assume that the four instructions of the loop body, starting at LOOP, are located at memory locations 1000, 1004, 1008 and 1012.

It is also assumed that the computer used here, uses the updated value of PC (which points to next instruction) in computing the effective address in the Relative mode.

So, the updated contents of the PC at the time, the branch target address is generated will be 1016.

To branch to location LOOP(1000), the offset value needed is X = -16.

* Relative addressing is often used in branch type instructions.
* It results in shorter address field in the instruction format since the relative address can be specified with a smaller number of bits compared to the no. of bits required to designate the entire memory address.

Indexed Addressing Mode

* In this mode, the contents of an index register are added to the address part of the instruction to obtain the effective address.
* The index register is a CPU register that contains an index value.
* This mode is useful in dealing with lists and arrays.
* The address field of this instruction defines the beginning address of a data array in memory.
* Each operand in the array is stored in memory relative to the beginning address.
* The distance between the beginning address and the address of the operand is the index value stored in index register.
* The index register can be incremented to facilitate access to consecutive operands.
* Either one CPU register can be dedicated to function as an index register or any one of the CPU registers can contain the index number.
* In the latter case, the register must be specified explicitly in a register field within the instruction format.

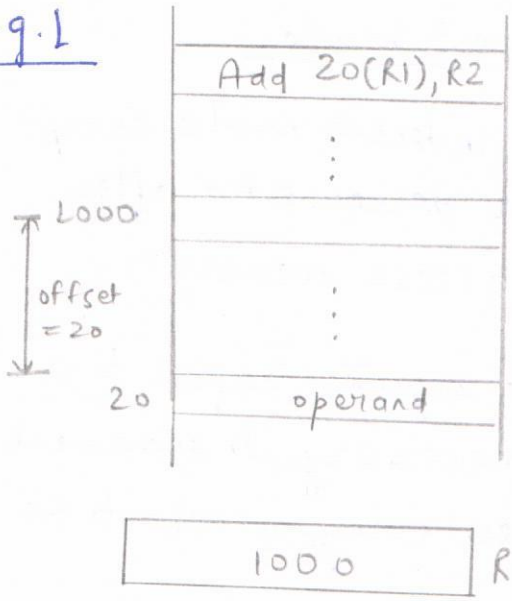
Note :- If an index-type instruction does not include an address field in its format, the instruction converts to the register indirect mode of operation.

* We can represent Index mode as X(Ri) where X denotes the constant value contained in the instruction and Ri is the name of the register involved.
* Thus, the effective address of the operand is –

$$EA = X + [R_i]$$

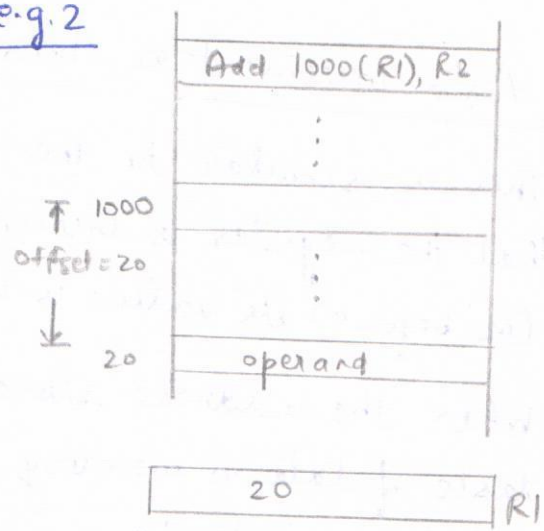
Here the contents of index register are not changed in the process of generating the effective address.

e.g.1



Offset is given as constant

e.g.2



Offset is in index register

15.9.

Base Register Addressing mode

* In this mode, the contents of base register is added to the address part of the instruction to obtain the effective address.
* It is similar to indexed addressing mode except that the register is now called base register instead of an index register.

* The difference is that index register is assumed to hold an index number that is relative to the address part of the instruction.

* But a base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address.

* The base register addressing mode is used in computers to facilitate the relocation of programs in memory.

15.10. Auto-increment or Auto-decrement mode

* This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.

* When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table.

* This can be achieved by using the increment or decrement instruction.

e.g. ADD R1, (R2)+

ADD R1, -(R2)

* These modes are useful for implementing 'LIFO' data structures

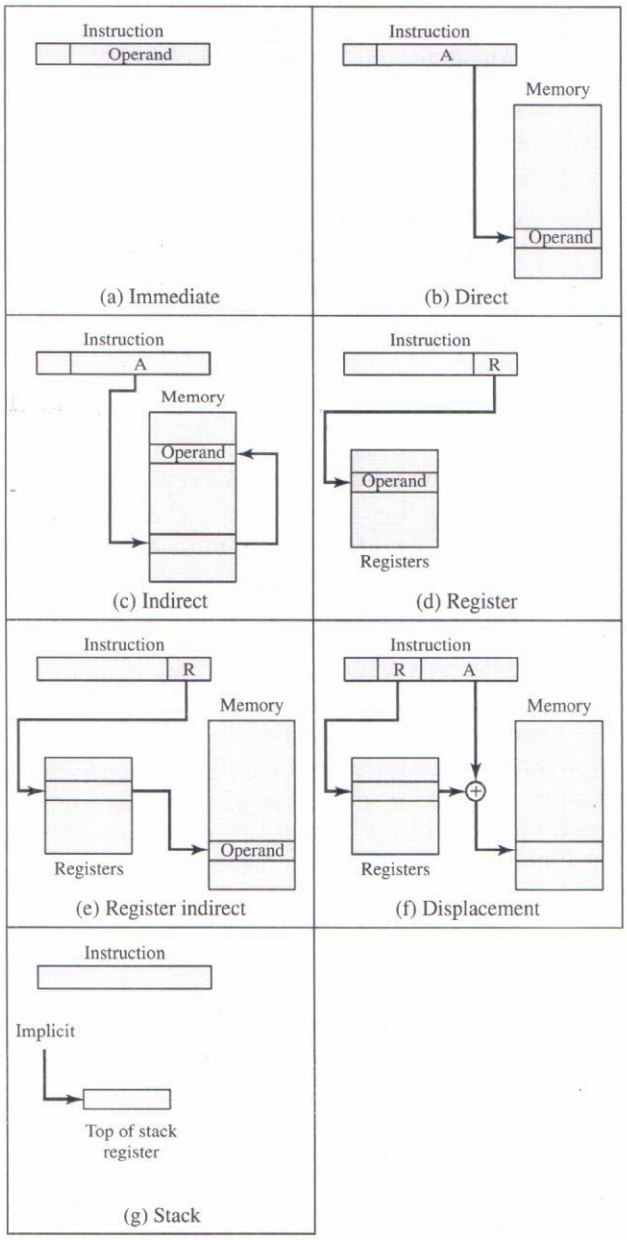


Figure 11.1 Addressing Modes

These modes are illustrated in Figure 11.1. In this section, we use the following notation:

A = contents of an address field in the instruction
R = contents of an address field in the instruction that refers to a register
EA = actual (effective) address of the location containing the referenced operand
(X) = contents of memory location X or register X

Table 11.1 Basic Addressing Modes

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

Table 11.1 indicates the address calculation performed for each addressing mode.

Before beginning this discussion, two comments need to be made. First, virtually all computer architectures provide more than one of these addressing modes. The question arises as to how the processor can determine which address mode is being used in a particular instruction. Several approaches are taken. Often, different opcodes will use different addressing modes. Also, one or more bits in the instruction format can be used as a *mode field.* The value of the mode field determines which addressing mode is to be used.

The second comment concerns the interpretation of the effective address (EA). In a system without virtual memory, the *effective address* will be either a main memory address or a register. In a virtual memory system, the effective address is a virtual address or a register. The actual mapping to a physical address is a function of the memory management unit (MMU) and is invisible to the programmer.

Immediate Addressing

The simplest form of addressing is immediate addressing, in which the operand value is present in the instruction

$$\text{Operand} = \text{A}$$

This mode can be used to define and use constants or set initial values of variables. Typically, the number will be stored in twos complement form; the leftmost bit of the operand field is used as a sign bit. When the operand is loaded into a data register, the sign bit is extended to the left to the full data word size. In some cases, the immediate binary value is interpreted as an unsigned nonnegative integer.

The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand, thus saving one memory or cache cycle in the instruction cycle. The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the word length.