



COMPUTER SYSTEM ARCHITECTURE

(Course code: AECD04)

B.Tech III Semester

Regulation: IARE BT-23

Prepared by:

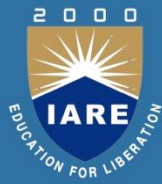
Mr. N Rajasekhar Reddy

Assistant Professor & Dean – Computer Center



Department of Computer Science and Engineering
INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
DUNDIGAL, HYDERABAD - 500 043

Course Outcomes



The course should enable the students to:

CO 1	Understand the organization and levels of design in computer architecture and To understand the concepts of programming methodologies.
CO 2	Describe Register transfer languages, arithmetic micro operations, logic micro operations, shift micro operations address sequencing, micro program example, and design of control unit.
CO 3	Understand the Instruction cycle, data representation, memory reference instructions, input-output, and interrupt, addressing modes, data transfer and manipulation, program control. Computer arithmetic: Addition and subtraction, floating point arithmetic operations, decimal arithmetic unit.
CO 4	Knowledge about Memory hierarchy, main memory, auxiliary memory, associative memory, cache memory, virtual memory Input or output Interface, asynchronous data transfer, modes of transfer, priority interrupt, direct memory access.
CO 5	Explore the Parallel processing, pipelining-arithmetic pipeline, instruction pipeline Characteristics of multiprocessors, inter connection structures, inter processor arbitration, inter processor Communication and synchronization

MODULE-I

Register Transfer And Micro operations

Course Outcomes

CO 1	Describe Register transfer languages, arithmetic micro operations, logic micro operations, shift micro operations address sequencing, micro program example, and design of control unit.
CO 2	Classify the functionalities of various micro operations such as arithmetic, logic and shift micro operations.
CO 3	Describe the Control unit and Control memory operations.
CO 4	Knowledge about address sequencing in Control memory
CO 5	Explore the micro program example and design of control unit

Contents

- ◉ Register transfer
- ◉ Bus and memory transfers
- ◉ Arithmetic microoperations
- ◉ Logic microoperations
- ◉ Shift microoperations
- ◉ Arithmetic logic shift unit

Computer arithmetic:

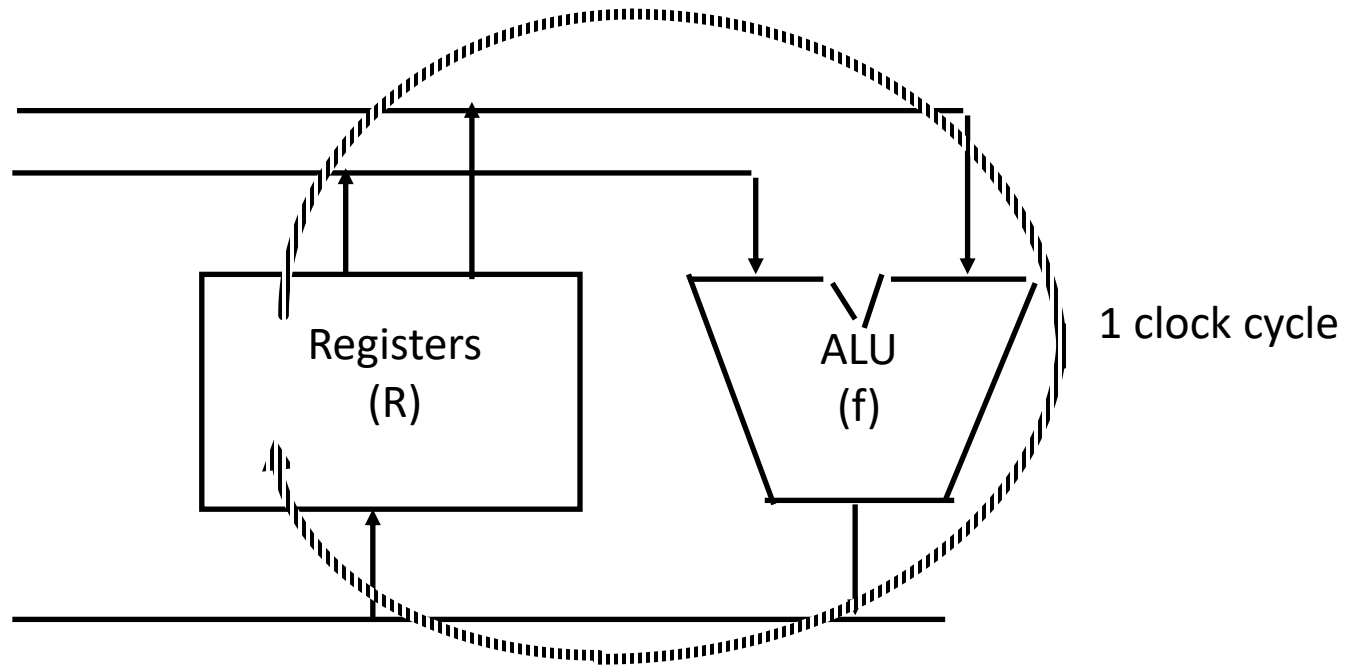
- ◉ Addition and Subtraction,
- ◉ Floating point arithmetic operations,
- ◉ Decimal arithmetic unit.

Register Transfer Language

- A digital system is a Interconnection of digital hardware modules that accomplish a specific information processing task.
- Digital system uses a modular approach.
- The modules are constructed from such as digital components as registers ,decoders , arithmetic elements and control logic.
- The operations are performed on the data in the registers.
- The operations executed on the data in registers are called micro operations.
- The functions built into registers are examples of micro operations
 - Shift
 - Load
 - Clear
 - Increment

Register Transfer Language

- The Micro operation is an elementary operation performed (during one clock pulse) on the information stored in one or more registers.



- $R \leftarrow f(R, R)$
- f: shift, load, clear, increment, add, subtract, complement, and, or, xor, ...

Register Transfer Language

- The internal organization of a computer is Defined as:
 - Set of registers and their functions
 - Micro operations set
 - Set of allowable microoperations provided by the organization of the computer
 - Control signals that initiate the sequence of micro operations (to perform the functions)
- The symbolic notation used to describe the micro operation transfers among registers is called a **register transfer language**.

Register Transfer

- 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
- Information Transfer from one register to another register is designated in symbolic form by using a replacement operator.

$R2 \leftarrow R1$

- Registers and their contents can be viewed and represented in *various ways*:
 - A register can be viewed as a single entity:



Register Transfer

- Registers may also be represented showing the bits of data they contain



Showing individual bits

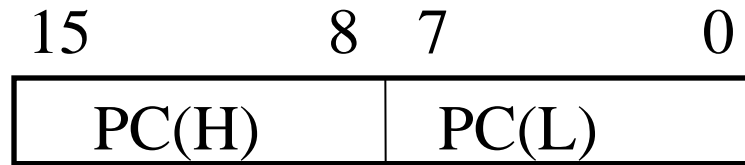
- Numbering of a registers



Numbering of bits

Register Transfer

- Portion of register



Subfields

Control Function

- The actions are performed only when certain conditions are true.
- This is similar to an “if” statement in a programming language.
- In digital systems, this is often done via a control signal, called a control function . control function is a Boolean variable.
 - If the signal is 1, the action takes place
- This is represented as:

P: $R2 \leftarrow R1$

Register Transfer

- Which means “if $P = 1$, then load the contents of register R1 into register R2”, i.e., if $(P = 1)$ then $(R2 \leftarrow R1)$.
- Implementation of controlled transfer

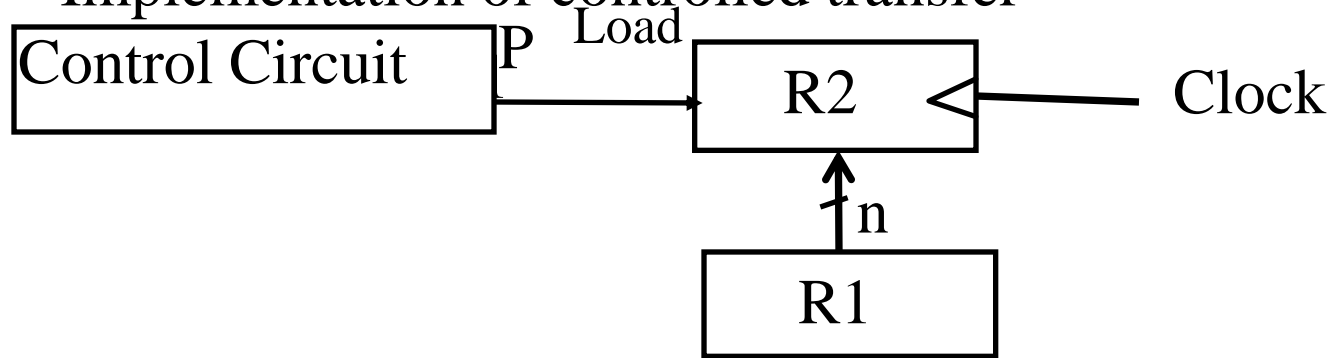


Fig: Block Diagram

Register Transfer

- The same clock controls the circuits that generate the control function and the destination register.

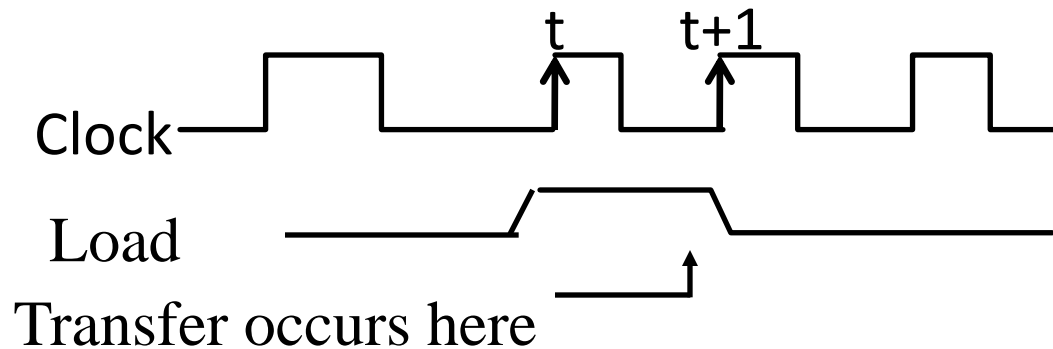


Fig :Timing Diagram

Register Transfer

- If two or more operations are to occur simultaneously, they are separated with commas

P: $R3 \leftarrow R5, MAR \leftarrow IR$

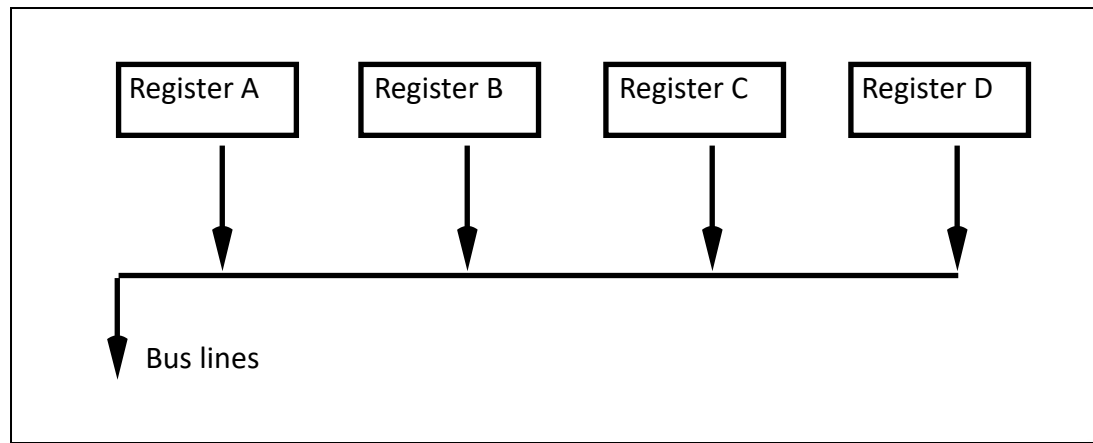
- Here, if the control function $P = 1$, load the contents of R5 into R3, and at the same time (clock), load the contents of register IR into register MAR
- Basic Symbols Used for Register Transfer is
 - Letters and Numerals to denote a registers. Ex: MAR, IR, R2 .
 - Parentheses $\overrightarrow{}$ () to denote a part of a register . Ex: $R2(0-7), R2(L)$.
 - Arrow \leftarrow to denote transfer of Information. Ex: $R2 \leftarrow R1$
 - Comma , Separates two micro operations . Ex: $R2 \leftarrow R1, R3 \leftarrow R4$

Bus Transfer

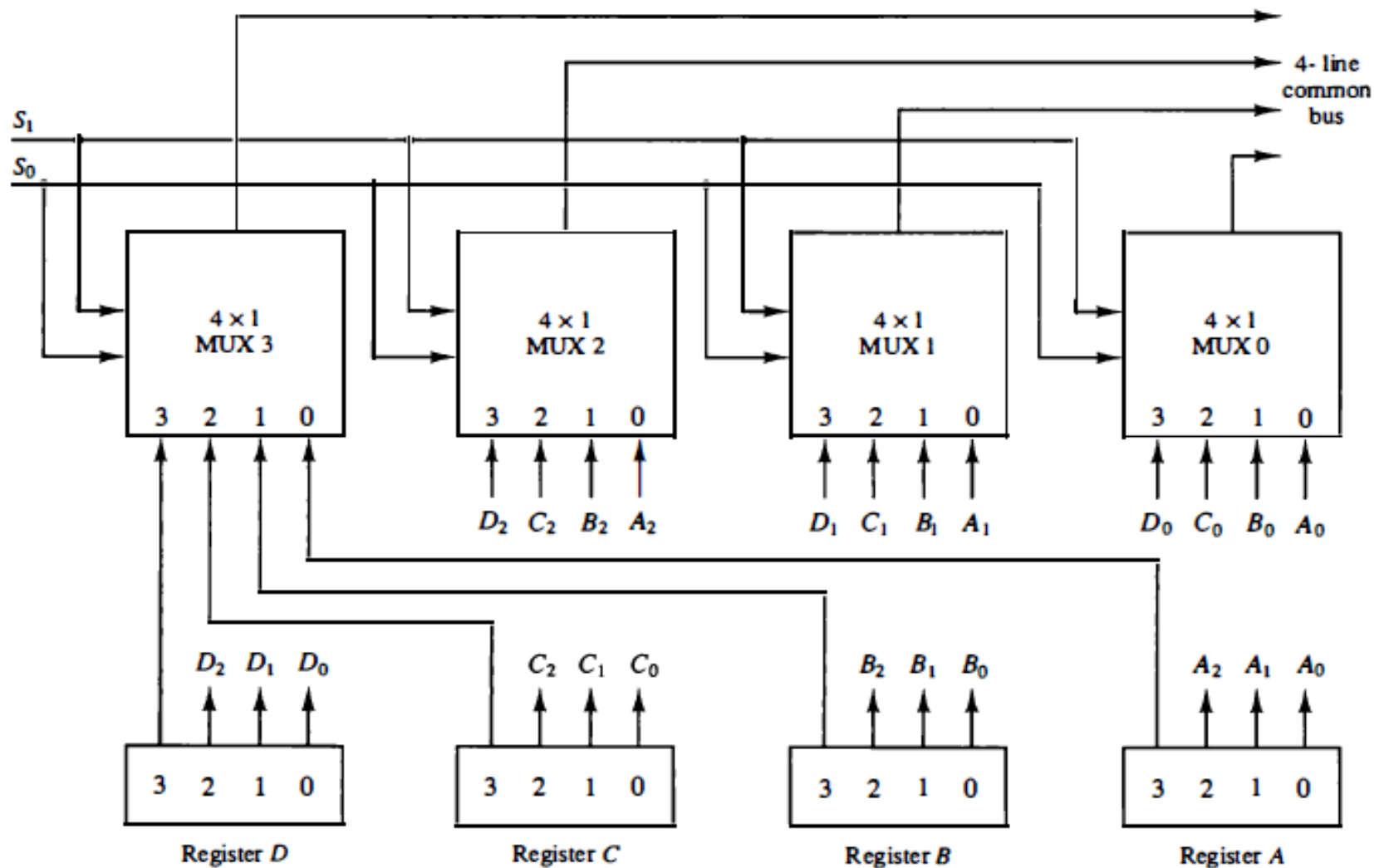
- In a digital system with many registers, it is impractical to have data and control lines to directly allow each register to be loaded with the contents of every possible other registers
- To completely connect n registers $\rightarrow n(n-1)$ lines
- $O(n^2)$ cost
 - This is not a realistic approach to use in a large digital system
- Instead, take a different approach
- Have one centralized set of circuits for data transfer – the bus
- Have control circuits to select which register is the source, and which is the destination

Bus Transfer

- Bus is a path(of a group of wires) over which information is transferred, from any of several sources to any of several destinations.
- One way to construct a common bus system is by using multiplexers.
- The multiplexer select the source register whose binary information is then placed into the bus.
- To transfer data from $R1 \leftarrow C$. i.e $BUS \leftarrow C$, $R1 \leftarrow BUS$



Bus Transfer

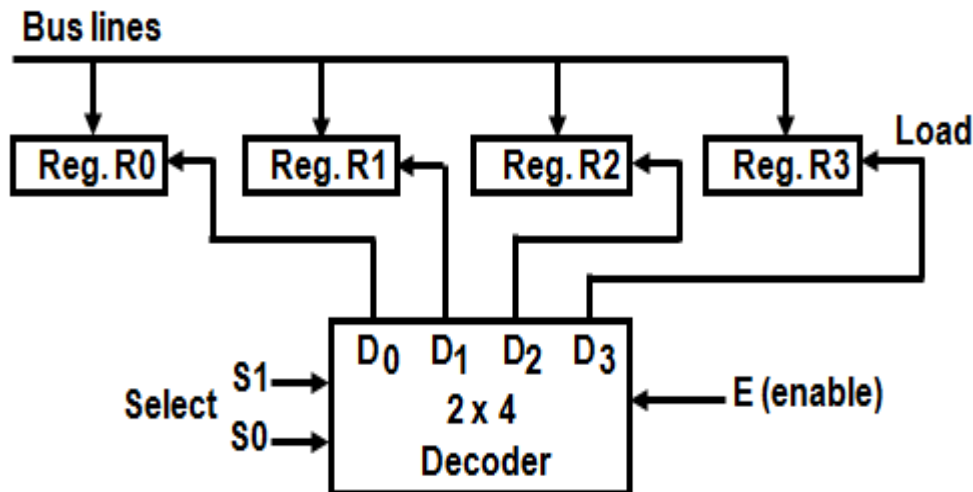


Bus Transfer

S1	S2	Register Selected
0	0	A
0	1	B
1	0	C
1	1	D

Function Table For Bus

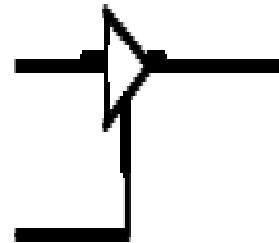
Fig : Transfer From Bus to Destination Register



Three State Bus Buffers

- A bus system can be implemented with three state gates instead of multiplexers.
- A three state gate is a digital circuit that exhibits three states.
- Two states are normal signal states 1 and 0. The third state is a high impedance state.
- High impedance output is open circuit

Normal input A
Control input C



Output $Y = A$ if $C = 1$
High-impedance if $C = 0$

ns the out

Fig: Graphic Symbol for Three State buffer

Three State Bus Buffers

Bus line with three-state buffers

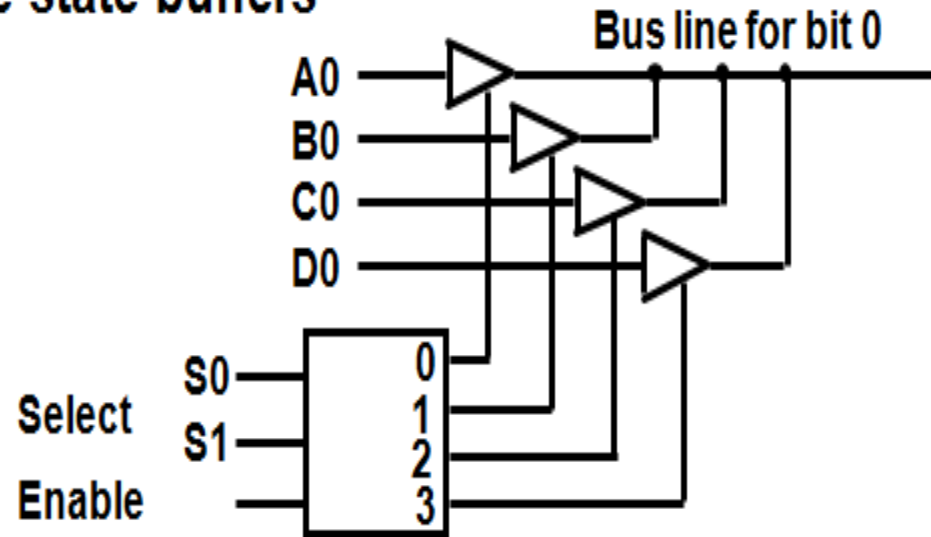


Fig : Bus Line with three state –Buffer

- To Construct a Common bus for four registers of n bits each using three state-bus buffer, we need n circuits with four buffers.

Memory Transfer

- The transfer of information from memory word to the outside environment is called Memory read operation.
- The transfer of new information to be stored into the memory is called memory write operation.
- Memory word is symbolically represented by using letter M.

Read : $DR \leftarrow M[AR]$

Write : $M[AR] \leftarrow R1$

SUMMARY OF R. TRANSFER MICROOPERATIONS



$A \leftarrow B$	Transfer content of reg. B into reg. A
$AR \leftarrow DR(AD)$	Transfer content of AD portion of reg. DR into reg. AR
$A \leftarrow \text{constant}$	Transfer a binary constant into reg. A
$ABUS \leftarrow R1,$ $R2 \leftarrow ABUS$	Transfer content of R1 into bus A and, at the same time, transfer content of bus A into R2
AR	Address register
DR	Data register
$M[R]$	Memory word specified by reg. R
M	Equivalent to $M[AR]$
$DR \leftarrow M$	Memory <i>read</i> operation: transfers content of memory word specified by AR into DR
$M \leftarrow DR$	Memory <i>write</i> operation: transfers content of DR into memory word specified by AR

Micro Operations

- Micro operation is an elementary operation performed with the data stored in registers.
- Computer system micro operations are of four types:
 1. Register transfer micro operations
 2. Arithmetic micro operations
 3. Logic micro operations
 4. Shift micro operations
- In register transfer micro operations the contents of the register can not be altered when transfer the data from the source to destination.
- The remaining three micro operations alter the data when transfer the data from one place to another.

Arithmetic Micro Operations

- The basic arithmetic micro operations are addition , subtraction , increment , decrement and shift operations.
- $R1 \leftarrow R2 + R3$ this micro operation specifies an add operation .
- It states that the contents of register R2 and R3 are added and result stored into register R1.
- $R1 \leftarrow R2 + R3 + 1(R2 - R3)$ this micro operation specifies a subtract operation .
- The subtract operation is implemented through complementation and addition.

Micro Operations

Arithmetic Micro Operations

$R3 \leftarrow R1 + R2$

Contents of R1 plus R2 transferred to R3

$R3 \leftarrow R1 - R2$

Contents of R1 minus R2 transferred to R3

$R2 \leftarrow \overline{R2}$

Complement the contents of R2

$R2 \leftarrow \overline{R2} + 1$

2's complement the contents of R2 (negate)

$R3 \leftarrow R1 + \overline{R2} + 1$

subtraction

$R1 \leftarrow R1 + 1$

Increment

$R1 \leftarrow R1 - 1$

Decrement

Binary Adder

- Basic hardware required to implement add operation is registers and digital component that perform add operation.
- The digital circuit that forms the arithmetic sum of two bits and previous carry is called a full adder.
- The digital circuit that generates the arithmetic sum of two binary numbers of any length is called a binary adder.
- The binary adder is constructed with full adder circuits connected in cascade , with the out put carry from one full adder connected to the input carry of the next full adder.
- An n bit binary adder requires n full adders.
- The output carry from each full adder is connected to the input carry of the next-higher –order full adder.

Micro Operations

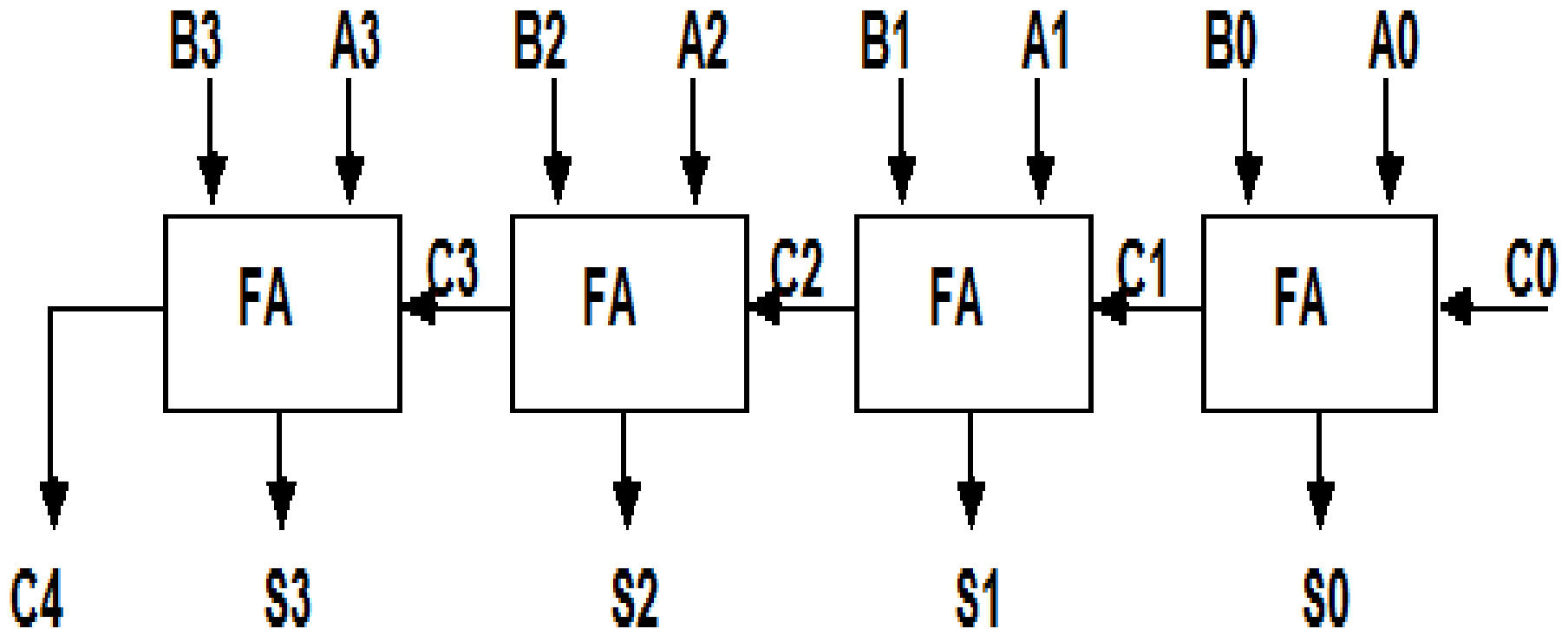


Fig: 4-bit binary adder

Binary Adder- Sub tractor

- The Subtraction of $A-B$ can be done by taking the 2's Complement of B and adding it to A .
- The addition and subtraction can be implemented by one common

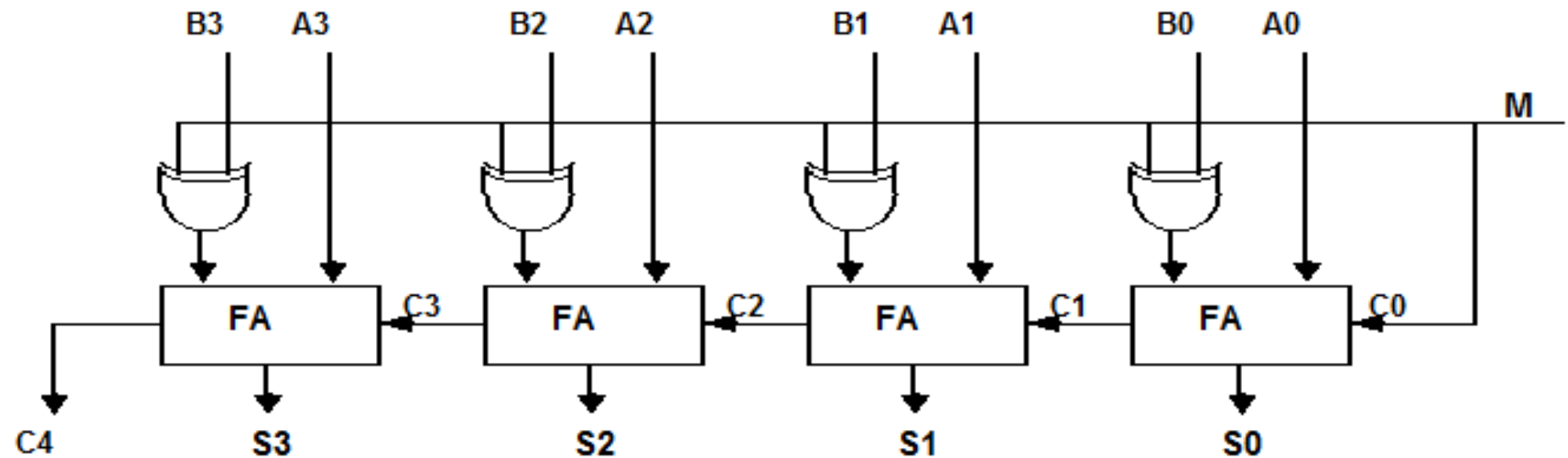


Fig: 4-bit Adder- subtractor

Binary Adder- Sub tractor

- Here the mode input M controls the operation.
- When $M=0$ the circuit works like an adder and $M=1$ the circuit works like a subtractor .
- Each exclusive-OR gate receives input M and one of the inputs of B.
- When $M=0$ the operation specifies $B \oplus 0 = B$.
- The full-adder receives the value of B ,the input carry is 0 ,and the circuit performs A plus B.
- When $M=1$ the operation specifies $B \oplus 1 = B'$ and $C_0=1$.
- The B inputs are all complemented and 1 is added through the input carry .
- Then the circuit performs operation A +2's Complement of B.
- Unsigned Numbers it gives A-B if $A \geq B$ or the 2's complement of (B-A)if $A < B$.
- Signed Numbers the result is A-B provided that there is no overflow.

Binary Incrementer

- The increment micro operation adds one to a number in a register.
- This is implemented by using a binary counter.
- The binary counter is implemented by using half adders connected in cascade.

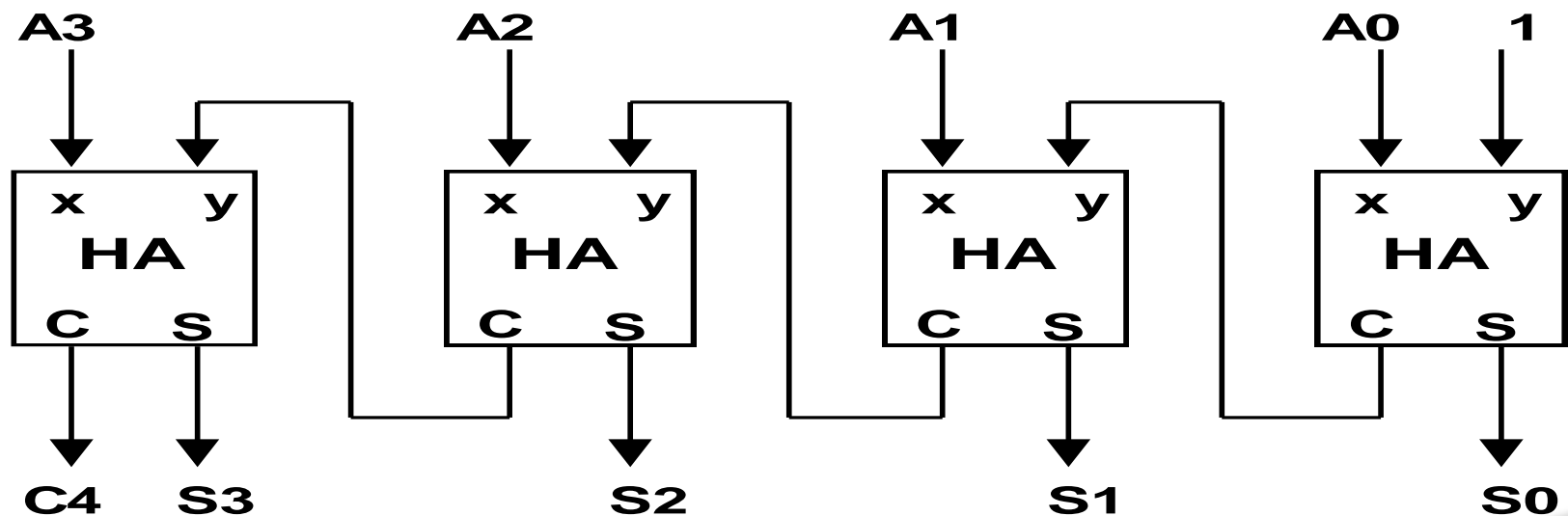


Fig : 4- bit binary Incrementer

Binary Incrementer

- One of the inputs to the least significant half adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented.
- The output carry from one half adder is connected to one of the inputs of the next-higher-order half adder.
- The generated out is displayed in s0 through s3.
- The above circuit can be implemented to an n-bit binary incrementer by including n-half adders.

Arithmetic Circuit

- The arithmetic operation can be implemented by using single composite arithmetic circuit.
- The basic component of an arithmetic circuit is parallel adder. By controlling the input of the parallel adder we obtain the different types of arithmetic operations.
- In a 4-bit Arithmetic circuit it contains 4 full adders and four multiplexers to choose different operations.
- There are two 4-bit inputs A and B and a 4-bit output D.
- The four Inputs From A directly connected to Adder and the B input is given to input of Multiplexers.
- The multiplexers data also receives the complement of B .
- The remaining two data inputs are connected to the Logic-0 and 1.
- The four multiplexers are controlled by two selection inputs , S1 and S0.

Arithmetic Circuit

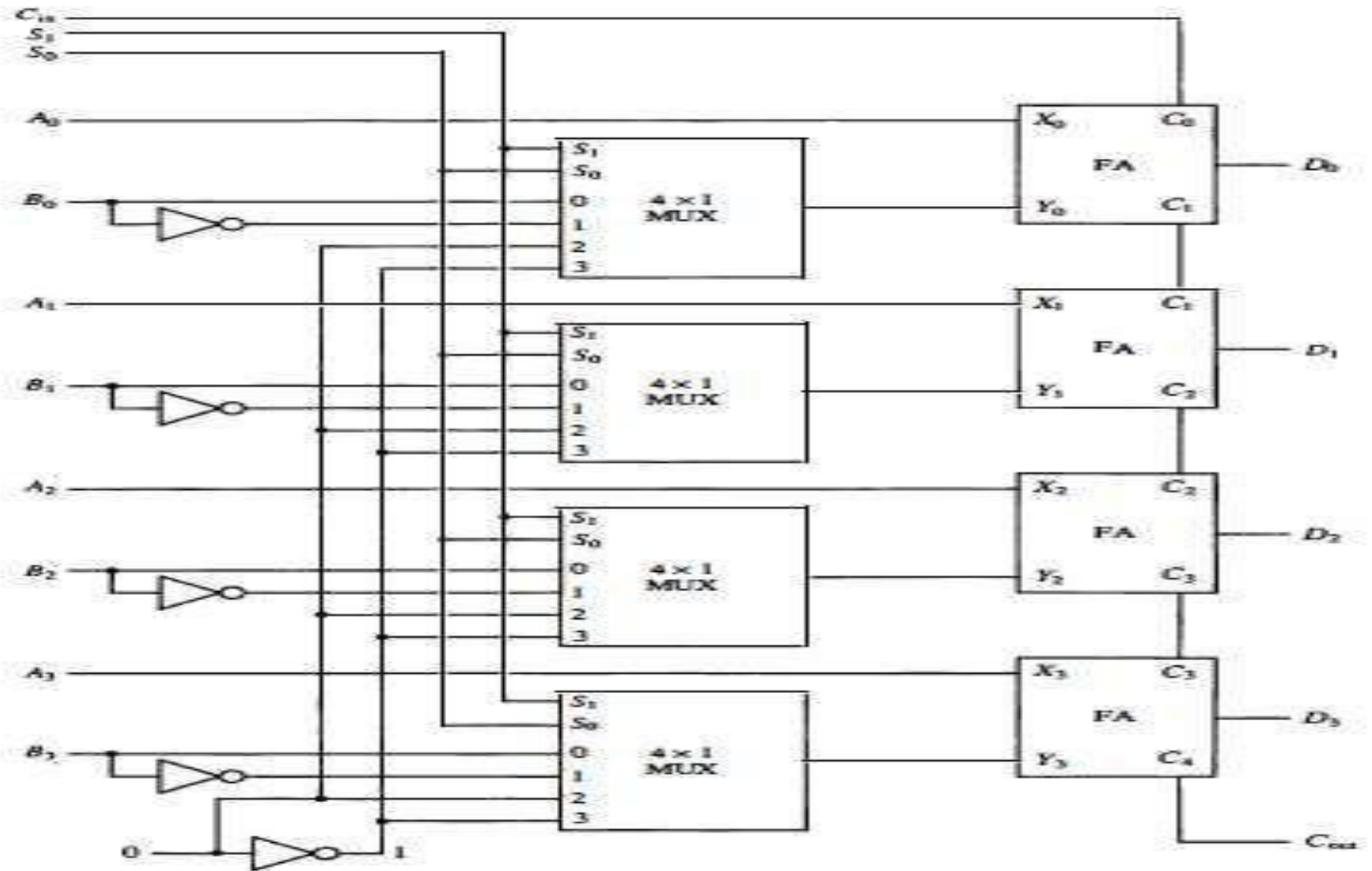


Fig: 4-bit Arithmetic circuit

Arithmetic Circuit

- The input carry C_{in} goes to the carry input of the FA in the least significant position .
- The other carries are connected from one stage to the next.
- The output of the Binary Adder is calculated by using Arithmetic sum
$$D = A + Y + C_{in}$$
here A is 4-bit Binary number at X inputs.
 Y is 4-bit binary number at Y inputs of the binary adder.
 C_{in} is the input carry.
- In the above equation by controlling the value of Y with two control inputs S_1 and S_0 and making the C_{in} 1 or 0 the above circuit performs all the arithmetic operations.

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	\overline{B}	$D = A + \overline{B}$	Subtract with borrow
0	1	1	\overline{B}	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

Logic Micro operations

- Logic micro operations specify the operations for strings of bits stored in registers.
- Logic microoperations are bit-wise operations, i.e., they work on the individual bits of data.
- The logic microoperation exclusive-OR with the contents of two registers R1 and R2 is symbolized by the statement:

$$P : R1 \leftarrow R1 \oplus R2$$

Ex: R1=1010 R2=1100



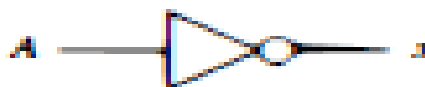

if P=1 then

1010 =R1

1100 =R2

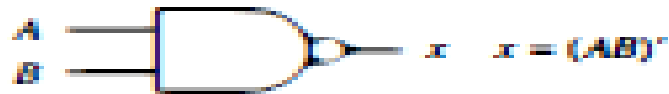
0110 =R1 after P=1

Logic Micro operations

Name	Graphic symbol	Algebraic function	Truth table															
AND		$x = A \cdot B$ or $x = AB$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	x	0	0	0	0	1	0	1	0	0	1	1	1
A	B	x																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$x = A + B$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	x	0	0	0	0	1	1	1	0	1	1	1	1
A	B	x																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
Inverter		$x = A'$	<table><tr><th>A</th><th>x</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	x	0	1	1	0									
A	x																	
0	1																	
1	0																	
Buffer		$x = A$	<table><tr><th>A</th><th>x</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	A	x	0	0	1	1									
A	x																	
0	0																	
1	1																	

Logic Micro operations

NAND



A	B	x
0	0	1
0	1	1
1	0	1
1	1	0

NOR



A	B	x
0	0	1
0	1	0
1	0	0
1	1	0

Exclusive-OR
(XOR)



A	B	x
0	0	0
0	1	1
1	0	1
1	1	0

Exclusive-NOR
or equivalence



A	B	x
0	0	1
0	1	0
1	0	0
1	1	1

Logic Micro operations

- Special Symbols used for logic microoperations OR , AND and Complement.

OR = \vee , AND = \wedge and Complement = \neg

- The main aim of adopting two sets of symbols is to differentiate between logic microoperations and control (Boolean) functions.

- $P+Q : R1 \leftarrow R2+R3 , R4 \leftarrow R5 \vee R6$

In the above statement + symbol performs OR operation between P and Q .It performs arithmetic addition between R2 and R3 .

List Of Logic Microoperations

- There are 16 different microoperations that can be performed with two binary operations.
- Most of the systems implement four of these \wedge , \vee , $_$ and \oplus .

X	Y	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
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

Table :Truth Table for 16 functions of Two Variables

Logic Micro operations

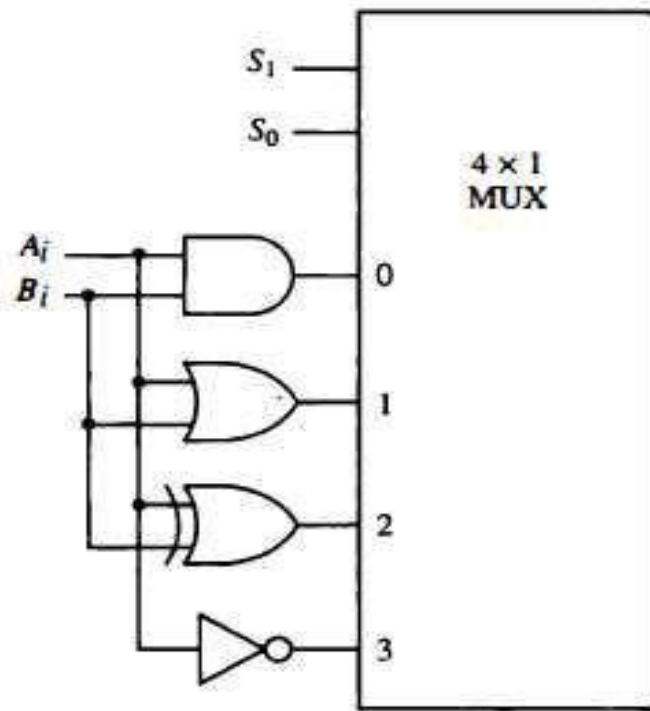
<i>Boolean Function</i>	<i>Micro-Operations</i>	<i>Name</i>
F0 = 0	$F \leftarrow 0$	Clear
F1 = xy	$F \leftarrow A \wedge B$	AND
F2 = xy'	$F \leftarrow A \wedge B'$	
F3 = x	$F \leftarrow A$	Transfer A
F4 = $x'y$	$F \leftarrow A' \wedge B$	
F5 = y	$F \leftarrow B$	Transfer B
F6 = $x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
F7 = $x + y$	$F \leftarrow A \vee B$	OR
F8 = $(x + y)'$	$F \leftarrow (A \vee B)'$	NOR
F9 = $(x \oplus y)'$	$F \leftarrow (A \oplus B)'$	Exclusive-NOR
F10 = y'	$F \leftarrow B'$	Complement B
F11 = $x + y'$	$F \leftarrow A \vee B'$	
F12 = x'	$F \leftarrow A'$	Complement A
F13 = $x' + y$	$F \leftarrow A' \vee B$	
F14 = $(xy)'$	$F \leftarrow (A \wedge B)'$	NAND
F15 = 1	$F \leftarrow \text{all 1's}$	Set to all 1's

Table : 16 Microoperations

Hardware Implementation

- To implement these microoperations it uses Logic gates.
- Most of the computers implement only four functions like AND , OR, NOT and XOR .

Figure 10 One stage of logic circuit.



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \bar{A}$	Complement

(b) Function table

Applications

- 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 | $A \leftarrow A \oplus B$ |

Logic Micro operations

Selective-Set

- The Selective-Set Operation sets to 1 the bits in register A where there are corresponding 1's in Register B.
- It does Not affect bit positions that have 0 in B.

$$\begin{array}{rcl}
 1\ 1\ 0\ 0 & A(\text{ before}) & \\
 \hline
 1\ 0\ 1\ 0 & B(\text{logic operand}) & \\
 1\ 1\ 1\ 0 & A(\text{after}) & (A \leftarrow A + B)
 \end{array}$$

- The OR microoperation can be used to selective set bits of a Register.

Selective-Complement

- The selective-complement operation complements bits in register A where there are corresponding 1's in B.
- It does not affect the bit positions that have 0's in B.

$$\begin{array}{r} 1\ 0\ 1\ 0\ A\ (A\ \text{before}) \\ \underline{1\ 1\ 0\ 0\ B\ (\text{Logic Operand})} \\ 0\ 1\ 1\ 0\ A\ (A\ \text{after}) \end{array} \quad (A \leftarrow A \oplus B)$$

- The exclusive microoperation can be used to selectively complement bits of register.

Selective-Clear

- The Selective-clear operation clears to 0 the bits in A only where there are corresponding 1's in B .

$$\begin{array}{rcl} 1\ 1\ 0\ 0 & A & (\text{A before}) \\ 1\ 0\ 1\ 0 & B & (\text{Logical Operand}) \\ \hline 0\ 1\ 0\ 0 & A & (A \leftarrow A \wedge B') \end{array}$$

- The Boolean operation performed on the individual bits in $A \wedge B'$.

Logic Micro operations

Mask

- The Mask operation is similar to the selective-clear operation except that the bits of A are cleared only where there are corresponding 0's in B.
- The mask operation is similar to an AND microoperation.

$$\begin{array}{rcl}
 1\ 1\ 0\ 0 & A & \text{(Before)} \\
 \hline
 1\ 0\ 1\ 0 & B & \text{(Logical Operand)} \\
 1\ 0\ 0\ 0 & A & (A \leftarrow A \cdot B)
 \end{array}$$

Logic Microoperations

Insert

- 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)

Clear

- The Clear operation compares the words in A and B and produces an all 0's result if the two numbers are equal.
- This operation is achieved by exclusive-OR operation.

$$\begin{array}{rcl} 1\ 1\ 0\ 0\ A & \text{(Before)} & \\ 1\ 1\ 0\ 0\ B & \text{(Logic Operand)} & \\ \hline 0\ 0\ 0\ 0\ A & (A \leftarrow A \oplus B) & \end{array}$$

Shift Microoperations

- Shift micro operations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations.
- The information transferred through the serial input determines the type of shift. There are three types of shifts: logical, circular, and arithmetic.
- A logical shift is one that transfers 0 through the serial input. We will adopt the symbols SHL and SHR for logical shift-left and shift-right micro operations. For example:
 -
 - $R1 \leftarrow \text{SHL } R1$
 - $R2 \leftarrow \text{SHR } R2$

Shift Microoperations

TABLE 7 Shift Microoperations

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

Shift Microoperations

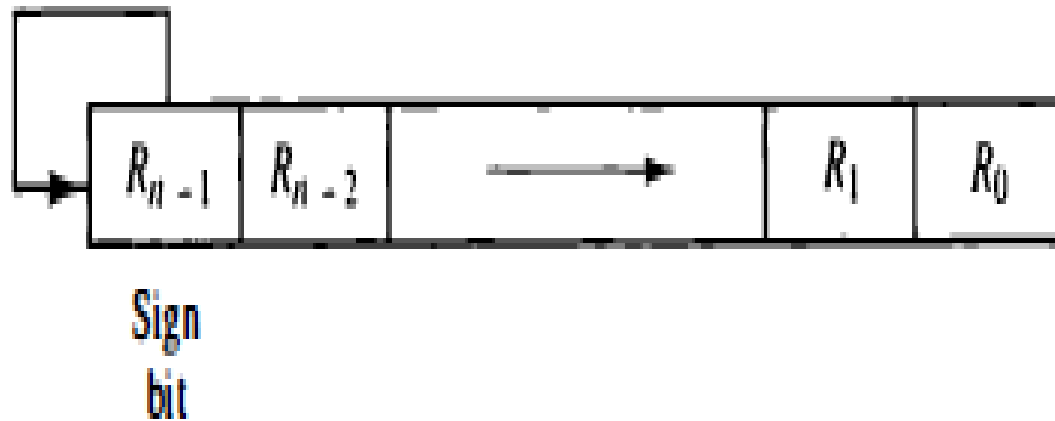
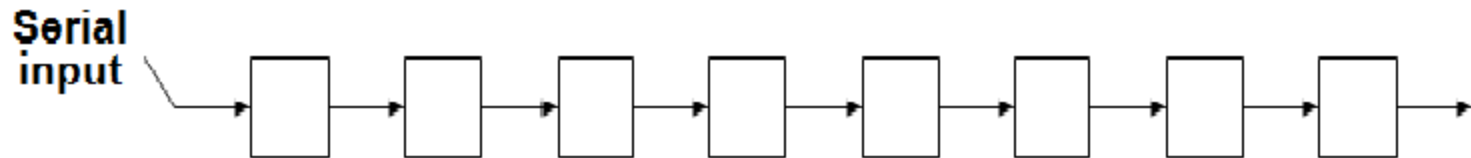


Figure 11 Arithmetic shift right.

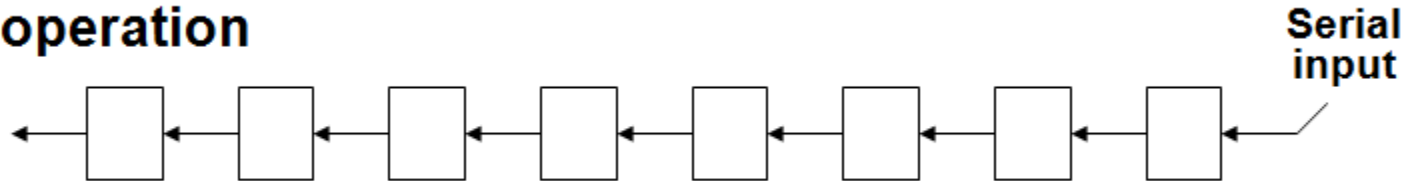
Shift Microoperations

- The contents of a register can be shifted to right or left.

- A right shift operation**



- A left shift operation**



- The information transferred through the serial input determines the type of shift.
- There are three types of shifts
 - Logical shift
 - Circular shift
 - Arithmetic shift

Logical Shift Microoperations

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

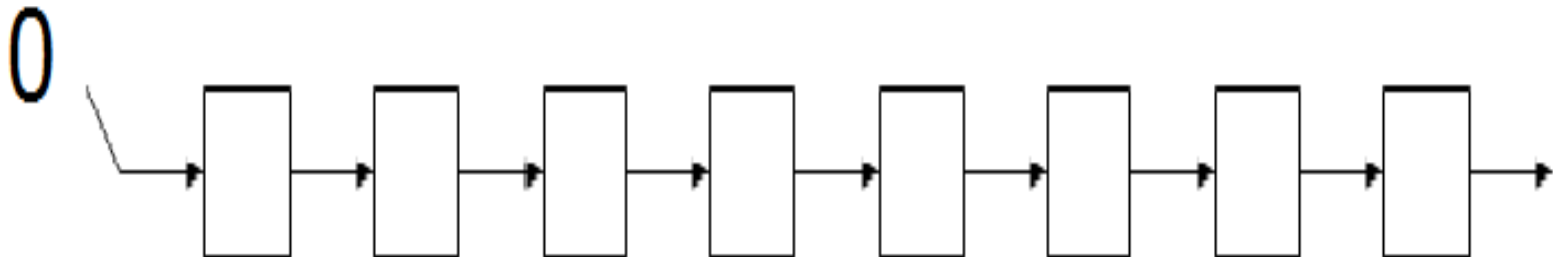


Fig : Right Logical Shift Operation

Shift Microoperations

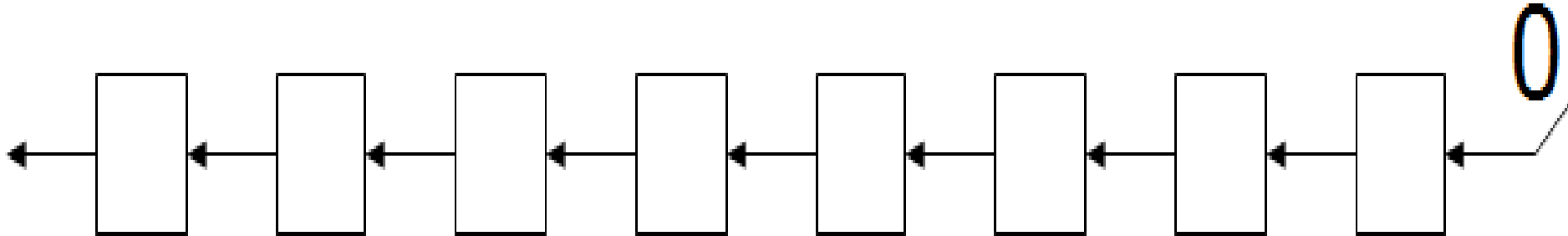


Fig: Left logical shift Microoperation

- Notations used to denote logical shift microoperations are:

shl ----> for logical shift left

shr ----> For logical shift right

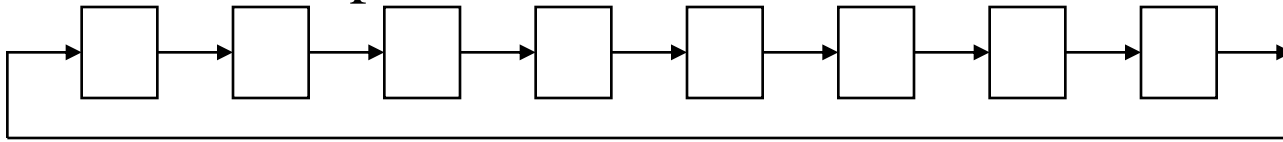
Examples :

$R2 \leftarrow \text{shl } R2$

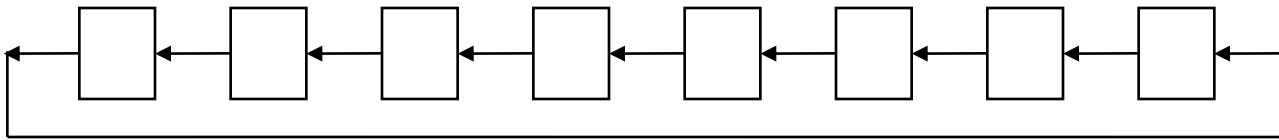
$R3 \leftarrow \text{shr } R3$

Shift Microoperations

- In a circular shift the serial input is the bit that is shifted out of the other end of the register.
- A right circular shift operation:



- A left circular shift operation:

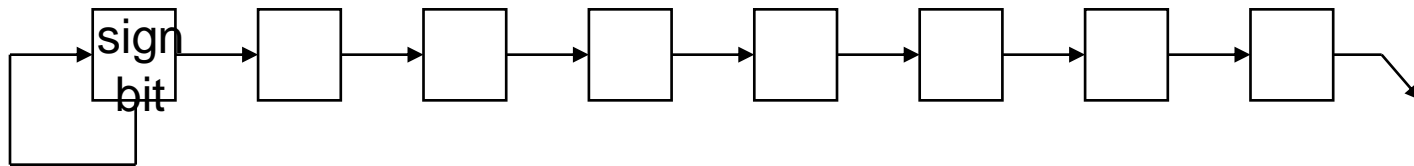


- In a RTL, the following notation is used
 - *cil* for a circular shift left
 - *cir* for a circular shift right
 - Examples:
 - $R2 \leftarrow cir R2$
 - $R3 \leftarrow cil R3$

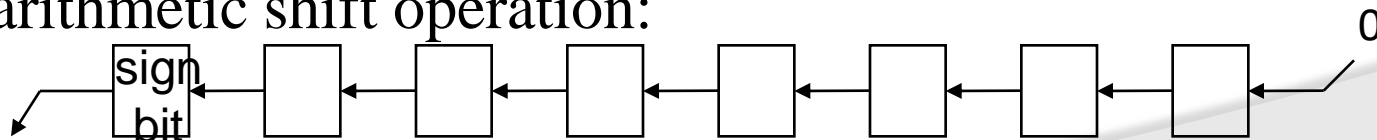
Shift Microoperations

Arithmetic Shift

- An arithmetic shift is meant for signed binary numbers (integer)
- An arithmetic left shift multiplies a signed number by two
- 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
- A right arithmetic shift operation:

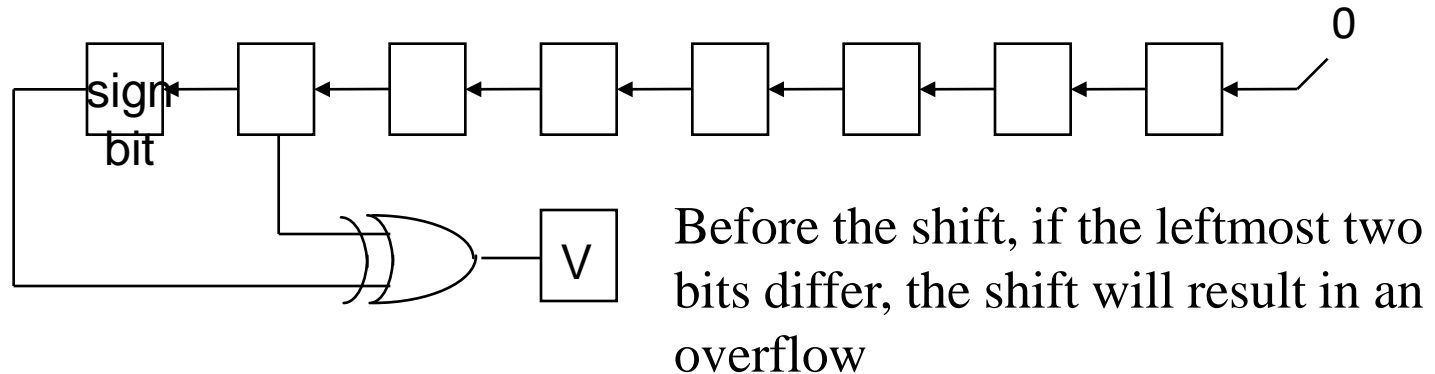


- A left arithmetic shift operation:



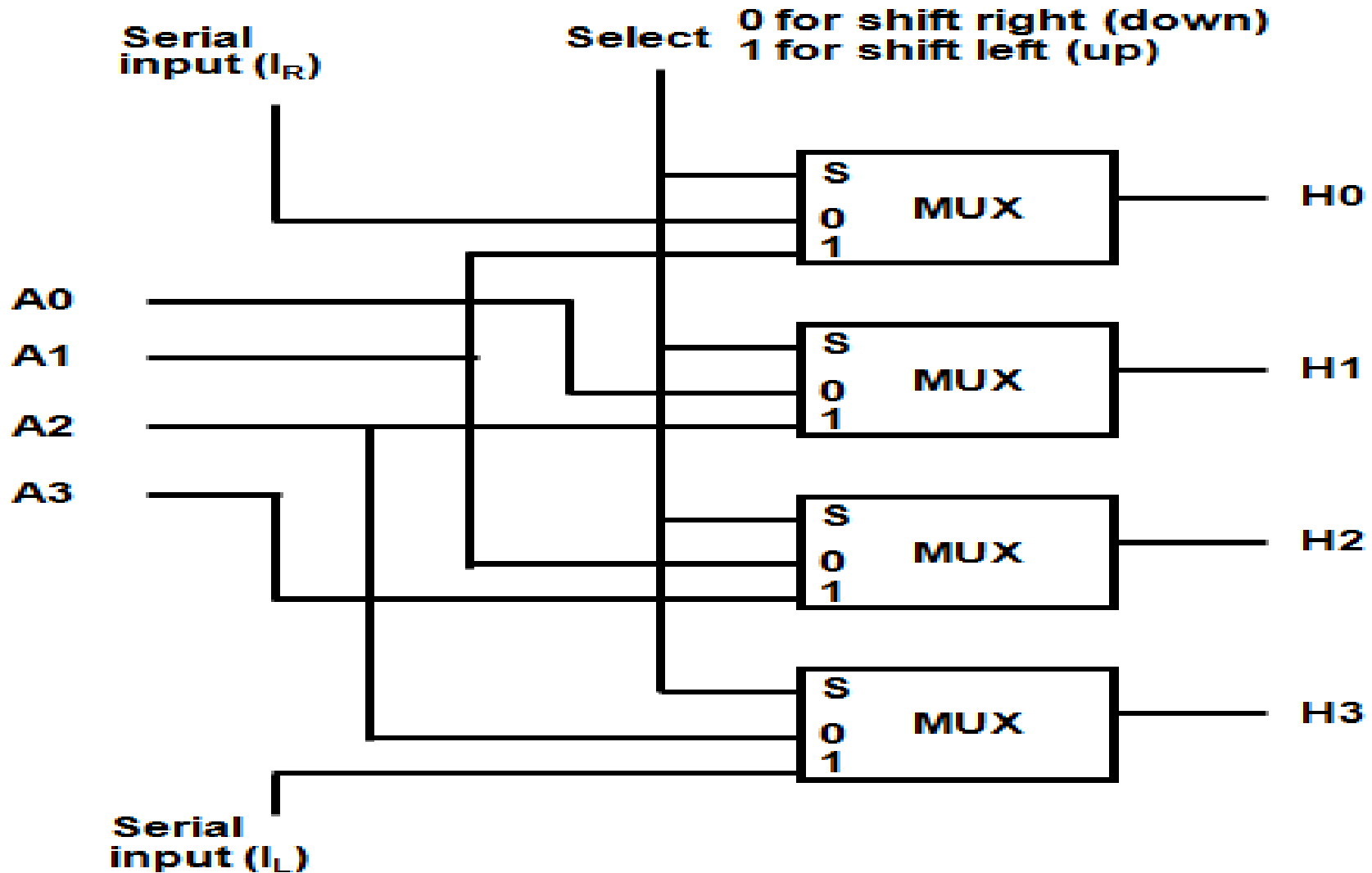
Shift Microoperations

- 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:
 - » $R2 \leftarrow ashr R2$
 - » $R3 \leftarrow ashl R3$

Hardware Implementation



4-bit Combinational Shifter

Hardware Implementation

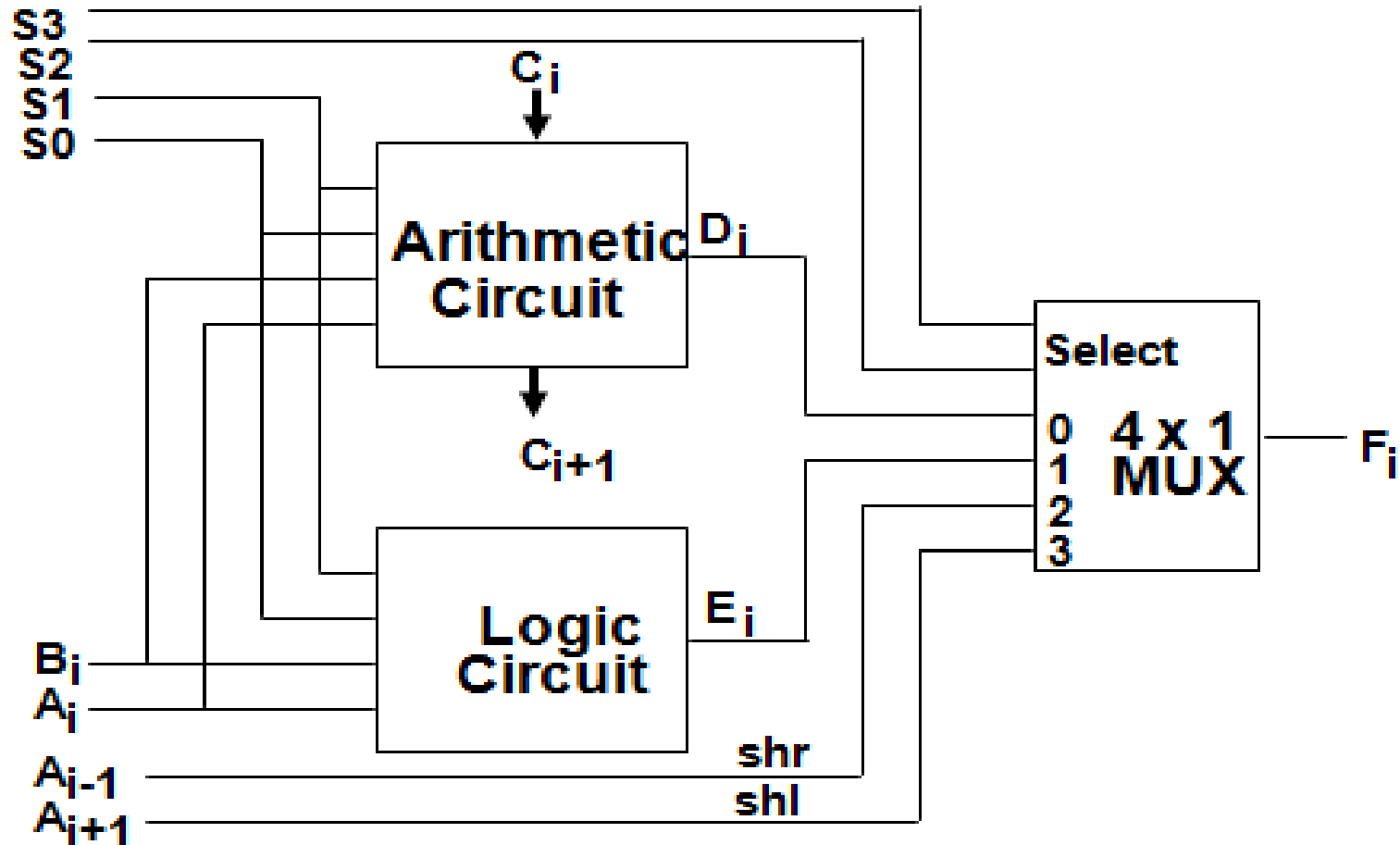
Select bit S	output			
	H0	H1	H2	H3
0	IR	A0	A1	A2
1	A1	A2	A3	IL

Table : Function Table

- When $S=1$ the input data is shifted to left
- When $s=0$ the input data is shifted to right.

Arithmetic Logic Shift Unit

- All the three operations are implemented with a single circuit.



Arithmetic Logic Shift Unit

S3	S2	S1	S0	Cin	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + B'$	Subtract with borrow
0	0	1	0	1	$F = A + B' + 1$	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	$F = A'$	Complement A
1	0	X	X	X	$F = \text{shr } A$	Shift right A into F
1	1	X	X	X	$F = \text{shl } A$	Shift left A into F

Computer arithmetic:

- Addition and subtraction
- Floating point arithmetic operations
- Decimal arithmetic unit.

Addition and subtraction

Addition (subtraction) algorithm:

- When the signs of A and B are identical (**different**), add the two magnitudes and attach the sign of A to the result.
- When the signs of A and B are different (**identical**), compare the magnitudes and subtract the smaller number from the larger.
- Choose the sign of the result to be the same as A if $A > B$ or the complement of the sign of A if $A < B$.
- If the **two magnitudes are equal**, subtract B from A and make the sign of the result positive.

Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

TABLE: Addition and Subtraction of Signed Magnitude Numbers

Addition and subtraction

Hardware Implementation:

- Let A and B be two registers that hold the magnitudes of the numbers, and As and Bs be two flip-flops that hold the corresponding signs. The result of the operation may be transferred to a third register: the result is transferred into A and As . Thus A and As together form an accumulator register.
- Consider now the hardware implementation of the algorithms below.
 1. **First, a parallel-adder is needed to perform the micro operation $A + B$.**
 2. **Second, a comparator circuit is needed to establish if $A > B$, $A = B$, or $A < B$.**
 3. **Third, two parallel-subtractor circuits are needed to perform the micro operations $A - B$ and $B - A$.**
 4. **The sign relationship can be determined from an exclusive OR gate with A, and B, as inputs.**

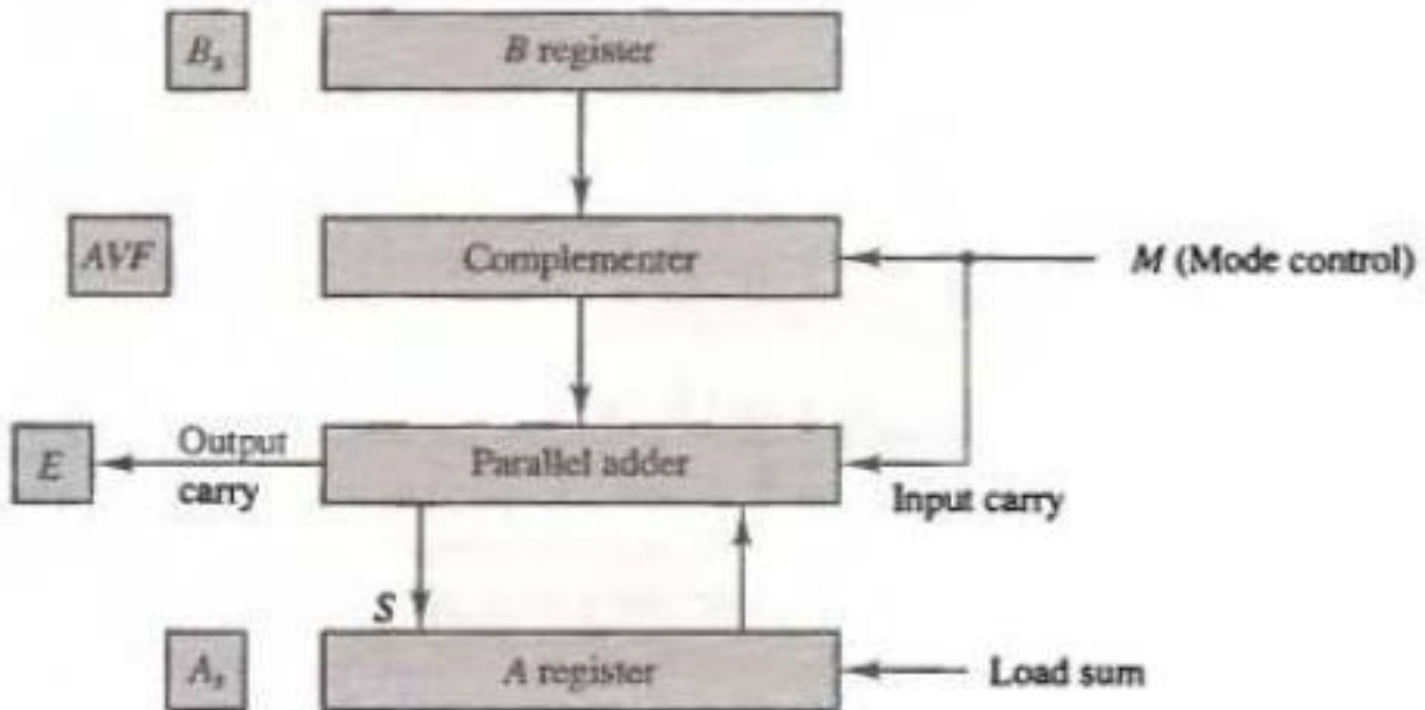


Figure: Hardware for signed magnitude addition and subtraction.

Addition and subtraction

This procedure requires a magnitude comparator, an adder, and two subtractors.

1. First, we know that subtraction can be accomplished by means of complement and add.
2. Second, the result of a comparison can be determined from the end carry after the subtraction.

Figure shows a block diagram of the hardware for implementing the addition and subtraction operations.

1. It consists of registers A and B and sign flip-flops A, and B, . Subtraction is done by adding A to the 2' s complement of B.
2. The output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitudes of the two numbers.
3. The add-overflow flip-flop AVF holds the overflow bit when A and B are added

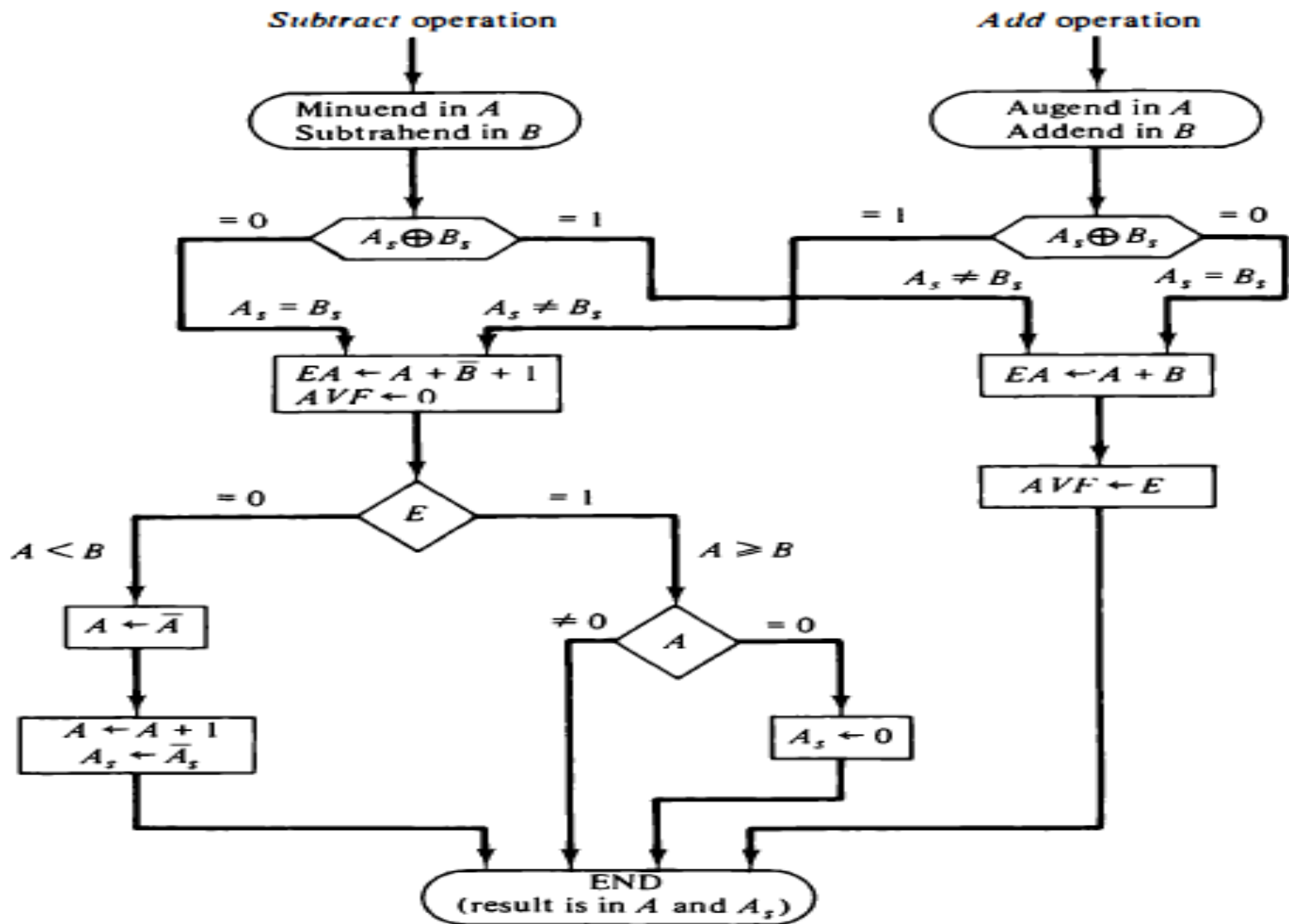
Addition and subtraction

- 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 complementor consists of exclusive-OR gates and the parallel adder consists of full-adder circuits as shown in Fig. 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 1's complement of B is applied to the adder, the input carry is 1, and output $S = A + \bar{B} + 1$. This is equal to A plus the 2's complement of B, which is equivalent to the subtraction $A - B$.

Addition and subtraction

- 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 micro operation $E \leftarrow A + B$. where EA is a register that combines 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 $A \geq B$ 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 $A \leftarrow -A + 1$.

Addition and subtraction



Addition and subtraction

However, we assume that the A register has circuits for micro operations complement and increment, so the 2' s complement is obtained from these two micro operations.

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 complement of the original sign of As.

Addition and subtraction

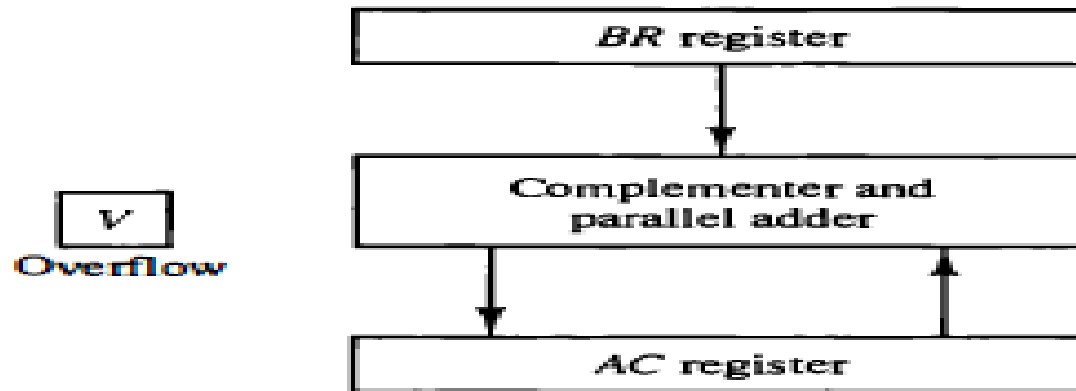


Figure: Hardware for signed 2's complement addition and subtraction.

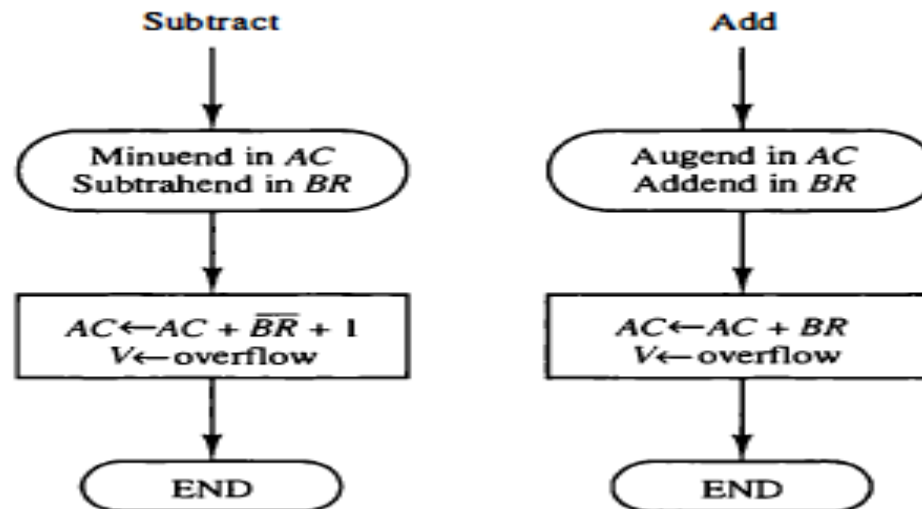


Figure :Algorithm for adding and subtracting numbers in signed 2's complement representation.

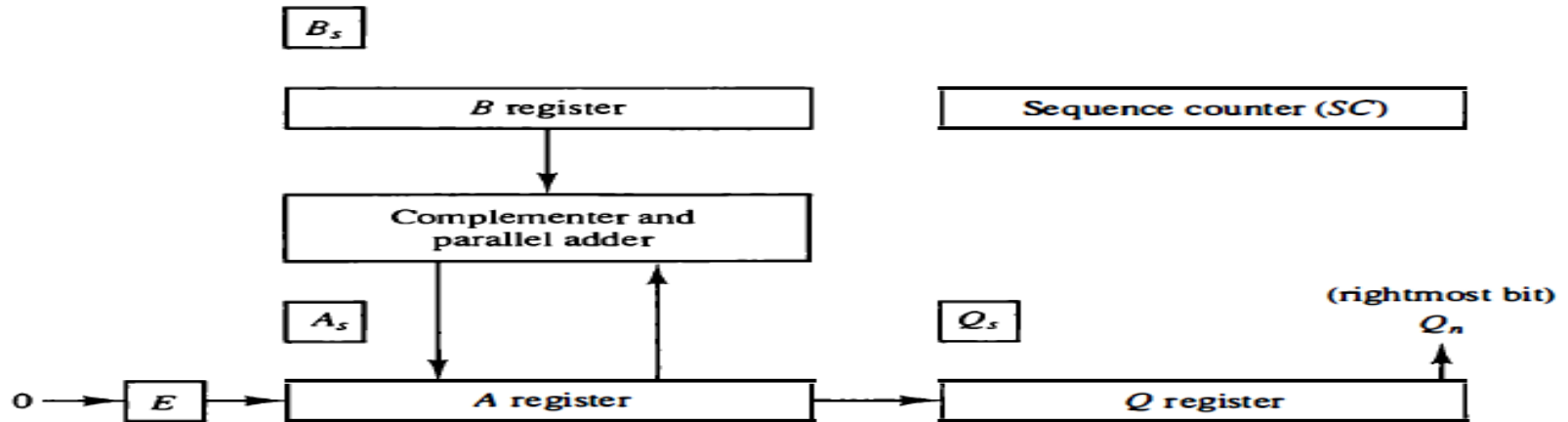
Multiplication Algorithms

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.

23	10111	Multiplicand
<u>19</u>	<u>× 10011</u>	Multiplier
	10111	
	10111	
	00000	+
	00000	
	<u>10111</u>	
437	<u>110110101</u>	Product

Multiplication Algorithms

Hardware Implementation for Signed-Magnitude Data:



- The hardware for multiplication consists of the equipment shown in Figure plus two more registers. These registers together with registers A and B are shown in Figure
- The multiplier is stored in the Q register and its sign in Q_s . 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

Multiplication Algorithms

- 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 Figure.
- 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.

Multiplication Algorithms

Hardware Algorithm:

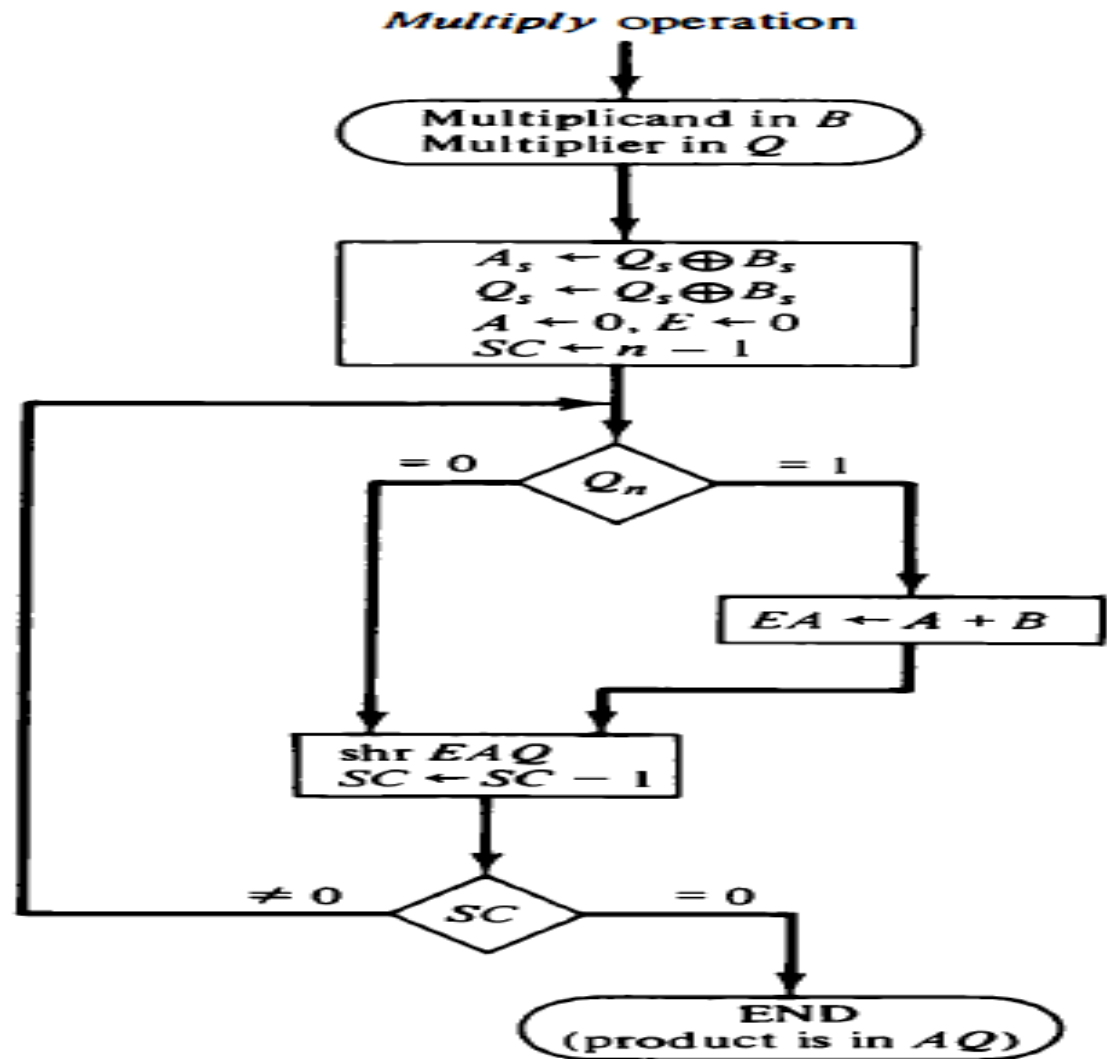


Figure : Flowchart for multiply operation

Multiplication Algorithms

- Flowchart of the hardware multiply algorithm. Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs, and Qs, 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.**
- After the initialization, the low-order bit of the multiplier in Q, 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 $SC = 0$. The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

Multiplication Algorithms

Multiplicand $B = 10111$	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		<u>10111</u>		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		<u>10111</u>		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
$Q_n = 0$; shift right EAQ	0	01000	10110	010
$Q_n = 0$; shift right EAQ	0	00100	01011	001
$Q_n = 1$; add B		<u>10111</u>		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in $AQ = 0110110101$				

TABLE : Numerical Example for Binary Multiplier

Multiplication Algorithms

Booth Multiplication Algorithm:

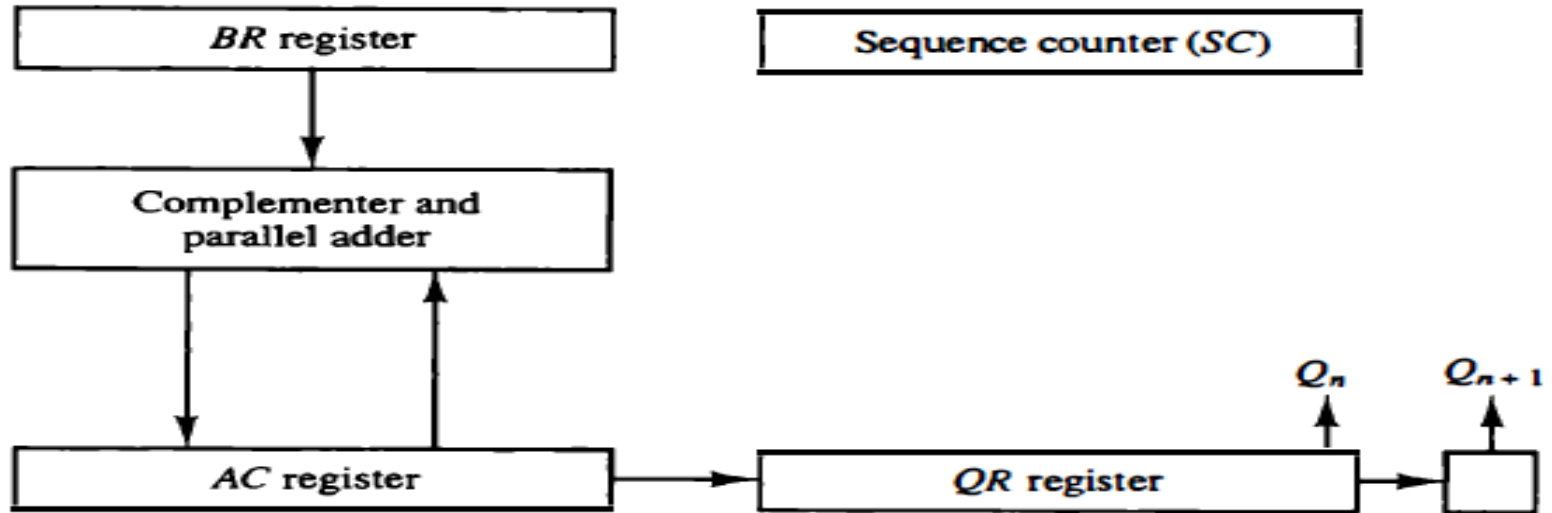


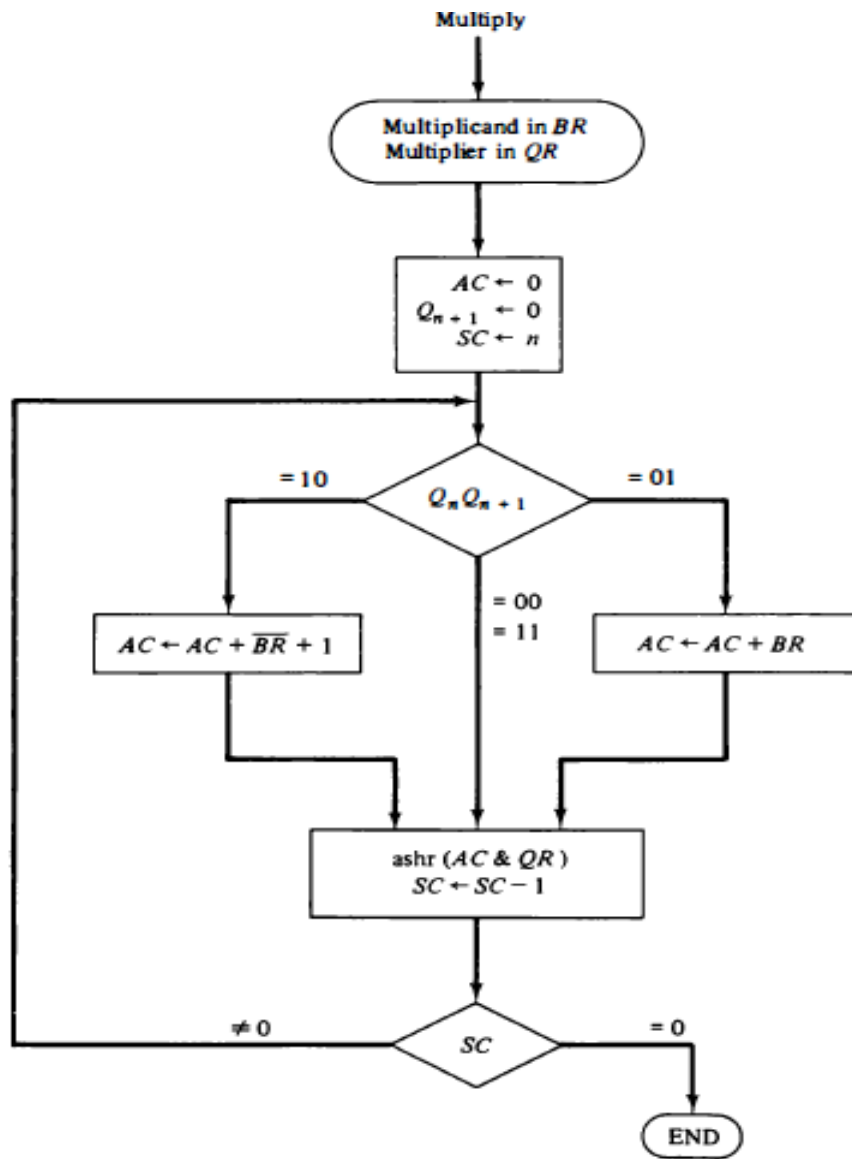
Figure: Hardware for Booth algorithm

- The hardware implementation of Booth algorithm requires the register configuration shown in Figure. This is similar to Figure except that the sign bits are not separated from the rest of the registers.
- To show this difference, we rename registers A, B, and Q, as AC, BR, and QR, respectively. Q, designates the least significant bit of the multiplier in register QR .

Multiplication Algorithms

- An extra flip-flop Q_{n+1} is appended to QR to facilitate a double bit inspection of the multiplier.
- AC and the appended bit Q_{n+1} are initially cleared to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier. The two bits of the multiplier in Q_n and Q_{n+1} are inspected.
- The flowchart for **Booth algorithm** is shown in below figure.

Multiplication Algorithms



A	Q	Q-1	Operation
0000	0011	0	0000
			1001
<hr/>			
			1001

1001	0011	0
1100	1001	1
1110	0100	1
0101	0100	1
0010	1010	0
0001	0101	0

Q3 Q2 Q1 Q0 Q-1

7(M) X 3(multiplier) = 21 (Q)

7 ---- 0111 21 ----- 10101
3 ----- 0011

Multiplication Algorithms

- If the two bits are equal to 10, it means that the first 1 in a string of 1' s has been encountered. This requires a subtraction of the multiplicand from the partial product in AC.
- **If the two bits are equal to 01, it means that the first 0 in a string of 0' s has been encountered. This requires the addition of the multiplicand to the partial product in AC .**
- When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other.
- As a consequence, the two numbers that are added always have opposite signs, a condition that excludes an overflow.

Multiplication Algorithms

- The next step is to shift right the partial product and the multiplier (including bit Q_{n+1}). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC unchanged.
- The sequence counter is decremented and the computational loop is repeated n times.
- **A numerical example of Booth algorithm is shown in Table 10-3 for $n = 5$. It shows the step-by-step multiplication of $(-9) \times (-13) = +117$.**
- Note that the multiplier in QR is negative and that the multiplicand in BR is also negative. The 10-bit product appears in AC and QR and is positive.
- The final value of Q_{n+1} is the original sign bit of the multiplier and should not be taken as part of the product.

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

TABLE: Example of Multiplication with Booth Algorithm

Division Algorithms

- **The hardware divide algorithm is shown in the flowchart of below Figure.**
- The dividend is in A and Q and the divisor in B . The sign of the result is transferred into Q, 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 .

Division Algorithms

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

Figure : Example of binary division.

Division Algorithms

Divisor $B = 10001$,

$\bar{B} + 1 = 01111$

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

Figure: Example of binary division with digital hardware.

Division Algorithms

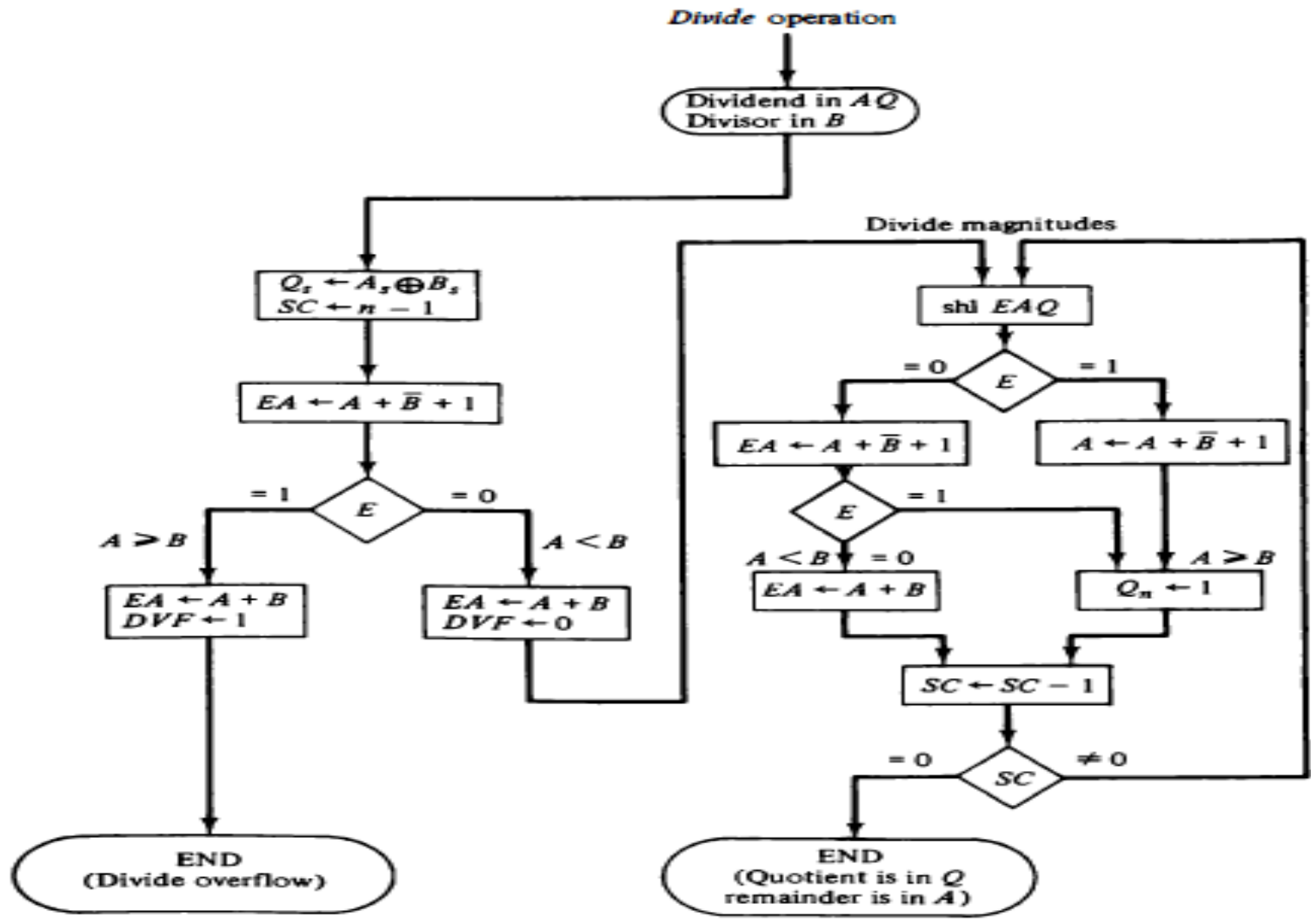


Figure : Flowchart for divide operation.

Floating-Point Arithmetic

- The register organization for floating-point operations is shown in Figure. There are three registers, BR, AC, and QR.
- Each register is subdivided into two parts. The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part uses the corresponding lowercase letter symbol.

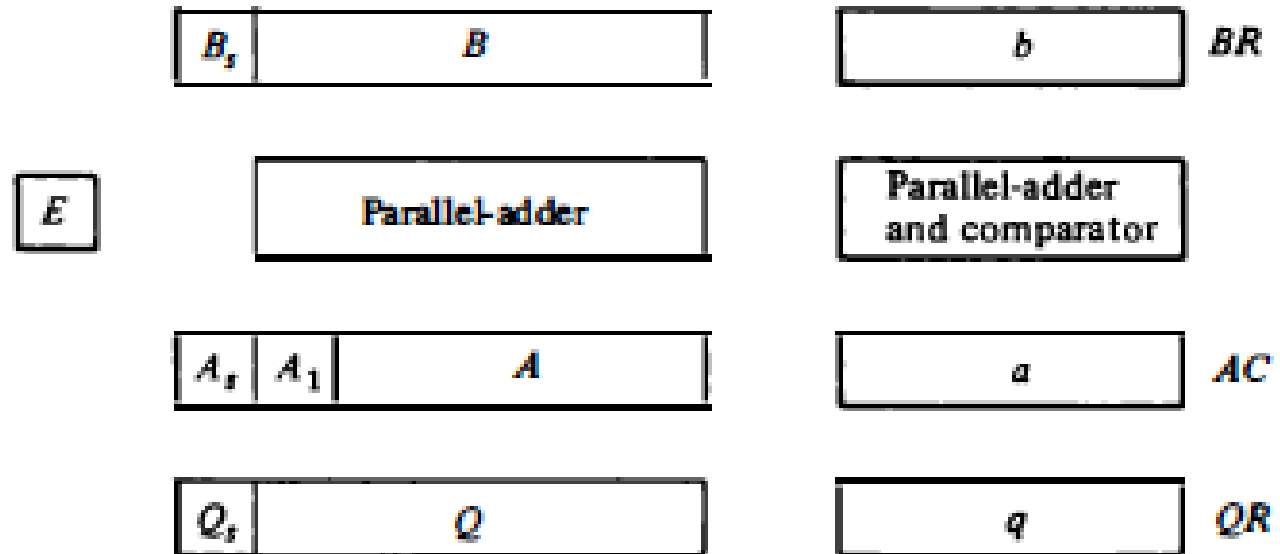


Figure : Registers for floating-point arithmetic operations.

Floating-Point Arithmetic

- It is assumed that each floating-point number has a mantissa in signed magnitude representation and a biased exponent. Thus the AC has a mantissa.
- whose sign is in A, and a magnitude that is in A. The exponent is in the part of the register denoted by the lowercase letter symbol a.
- The Figure shows explicitly the most significant bit of A, labeled by A1. The bit in this position must be a 1 for the number to be normalized.
- Note that the symbol AC represents the entire register, that is, the concatenation of As, A, and a. Similarly, register BR is subdivided into Bs, B, and b, and Q R into Qs, Q, and q.

A parallel-adder adds the two mantissas and transfers the sum into A and the carry into E . A separate parallel-adder is used for the exponents. Since the exponents are biased, they do not have a distinct sign bit but are represented as a biased positive quantity.

Different Floating Point operations are,

- **Addition and Subtraction**
- **Multiplication**
- **Division**
- **BCD Adder**

Decimal Arithmetic Unit

Decimal Arithmetic Unit and Operations:

- A decimal arithmetic unit is a digital function that performs decimal micro-operations.
- It mainly performs two operations: Addition and Subtraction.
- The addition operation is performed with BCD Adder whereas Subtraction is performed with BCD Subtractor.
- Here, BCD stands for binary-coded decimal.
- This decimal arithmetic unit first accepts coded decimal numbers and then generates output in the binary form.

BCD Code:

- In this code, each decimal digit **(0-9)** is represented by a **4-bit** binary number.
- Example, **374** is a decimal number whereas, 3, 7, and 4 are the decimal digits.
- Henceforth, by using BCD we can represent decimal digits by a **4-bit** binary number.
- Positional weights are **8-4-2-1**. And sometimes BCD is known as **8-4-2-1** code.
- For example, to represent 10 in BCD, the BCD code for 1 and 0 will be concatenated.
- Thus, **(10)10 = 00010000**
- Let's convert a decimal number **15** in binary as well as in BCD.
(15)10 = (1111)2 in binary **(15)10 = (00010101)** in BCD

BCD Addition

- BCD addition is a replica of the binary addition.
- In BCD addition, we've to deal with three cases which are:
- Sum ≤ 9 , final carry = 0, the obtained result is correct.
- Sum ≤ 9 , final carry = 1, the obtained result is incorrect.
- To correct the answer, we just need to add 6(or 0110) to the obtained result. Sum > 9 , final carry = 0, again the result obtained is incorrect.
- To correct the answer the same will be followed, i.e., the addition of 6(or 0110) to the result

(2)2 + (6)2 #BCD Addition

$$\begin{array}{r}
 11 \\
 0010 \\
 + 0110 \\
 \hline
 1000 \quad (8 < 9)
 \end{array}$$

Carry - 0
 Ans < 9
 The result is correct

$(3)_{10} + (7)_{10}$ #BCD Addition

$$\begin{array}{r}
 111 \\
 0011 \\
 + 0111 \\
 \hline
 1010 \text{ (10 > 9)}
 \end{array}
 \quad \leftarrow \quad
 \begin{array}{l}
 \text{Carry - 0} \\
 \text{Ans > 9} \\
 \text{The result is correct as} \\
 \text{the BCD code for 10 is} \\
 \text{not 1010}
 \end{array}$$

- By adding 6 we can obtain the correct result. Now the question may arise why 6 only?
- We know that we use the 4-bit binary number to represent the decimal digits which are from (0-9).
- Also, we know that with 4-bit there can be 16 (0-15) possibilities.
- But the valid cases are only from (0-9), by this, we can calculate the invalid cases, i.e., $15 - 9 = 6$
- To correct the above answer, let's add 6 to the sum.

$(10)_{10} = 0001\ 0000$

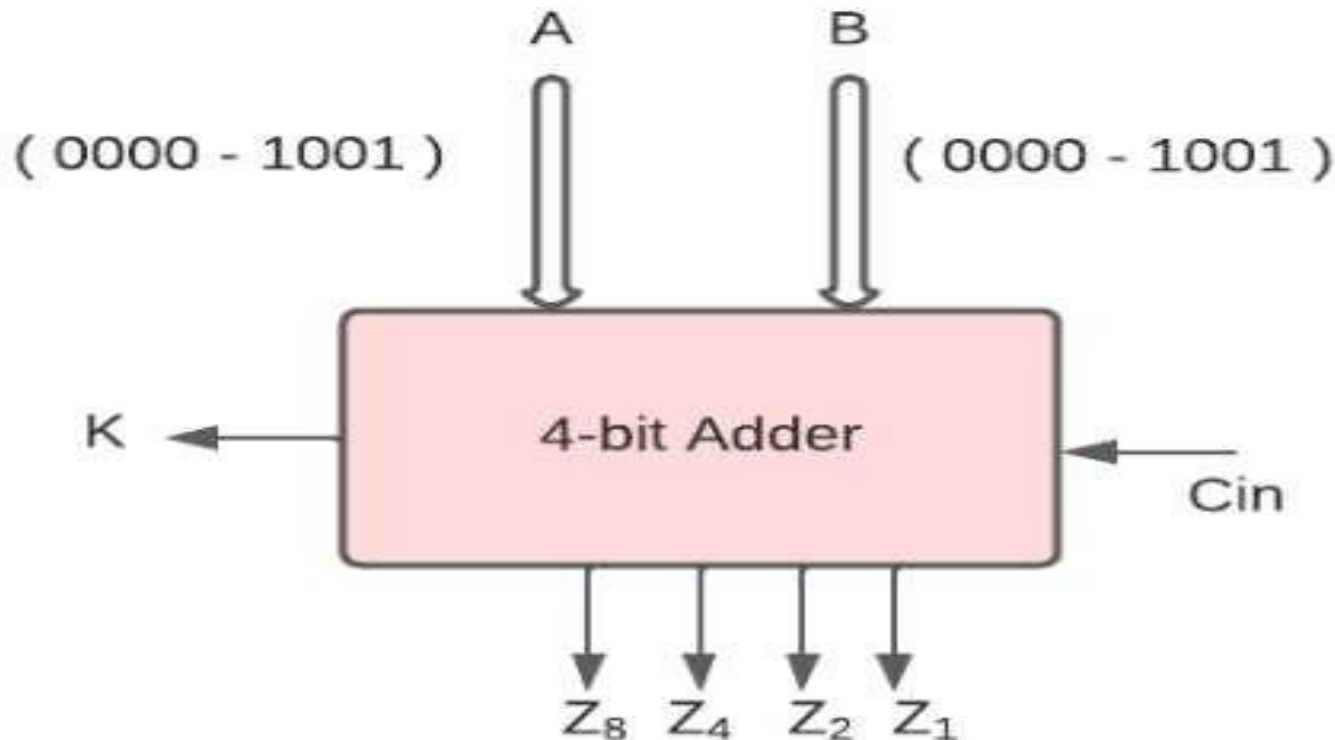
$$\begin{array}{r}
 11 \\
 11010 \\
 + 0110 \\
 \hline
 10000
 \end{array}$$

BCD Adder:

The digital system handles the decimal number in the form of binary-coded decimal numbers (BCD).

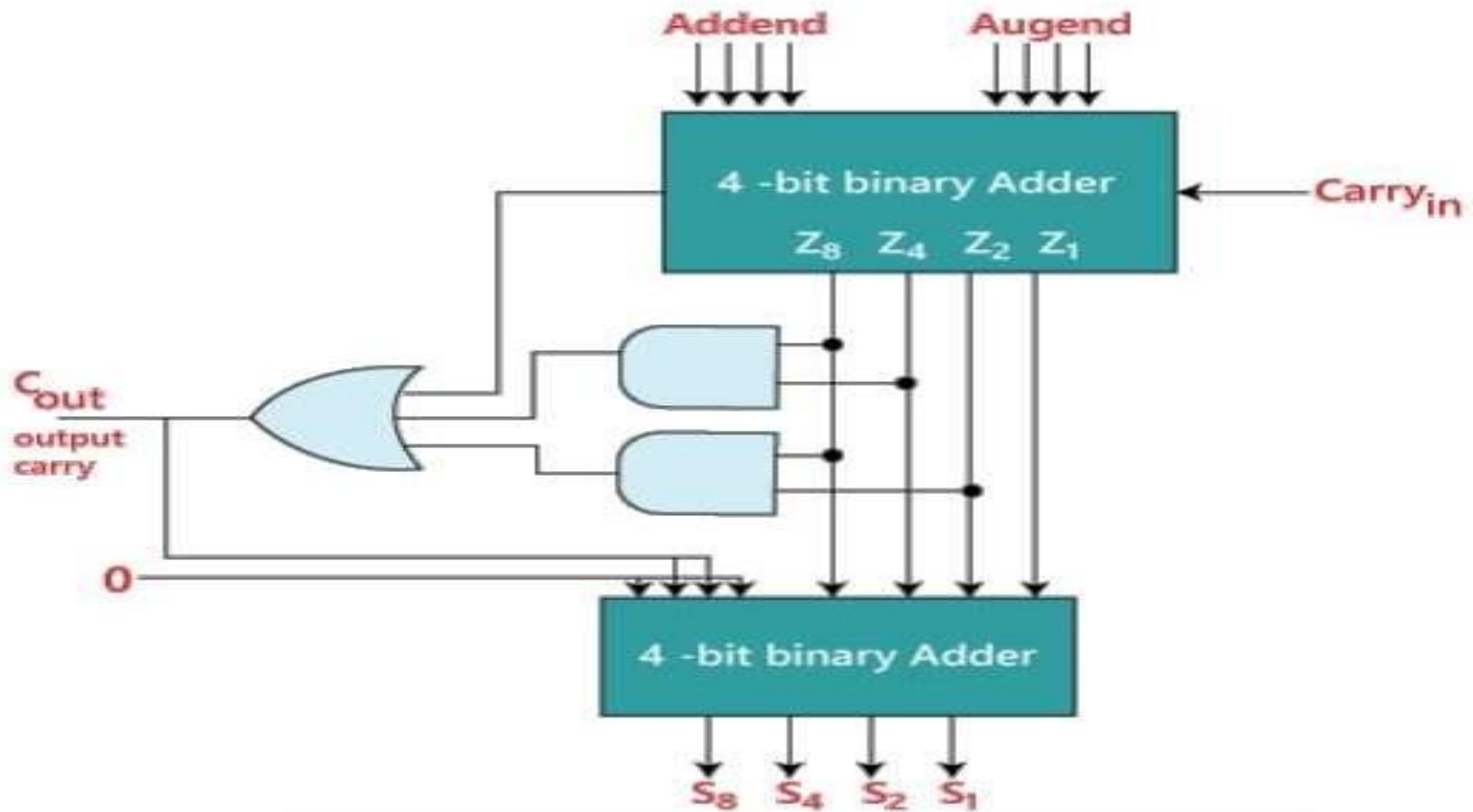
A BCD Adder Circuit adds two BCD digits and returns a BCD sum digit.

BCD numbers use digits, ranging from 0 to 9, which are represented in binary as 0000 to 1001, i.e. each BCD digit is a 4-bit binary number.



- The maximum value of output will be 19 (i.e. $9(1001)+9(1001)+1(\text{carry}=1) = 19$).
- Here, we will only be obtaining the binary addition of the two numbers. To convert them into the BCD form by using the BCD adder.
- As a result, in order to build a BCD Adder Circuit, we'll need 4-bit binary adder for initial addition
- Logic circuit to detect sum greater than 9 and one more 4-bit adder to add 0110(6) in the sum if the sum is greater than 9 or carry is 1.

Sum of Binary Digits					Sum of BCD Digits					Decimal
K	Z ₈	Z ₄	Z ₂	Z ₁	C	S ₈	S ₄	S ₂	S ₁	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	1	0	10
0	1	0	1	1	1	0	0	1	1	11
0	1	1	0	0	1	0	0	0	0	12
0	1	1	0	1	1	0	0	0	1	13
0	1	1	1	0	1	0	1	1	0	14
0	1	1	1	1	1	0	1	1	1	15
1	0	0	0	0	1	0	1	0	0	16
1	0	0	0	1	1	0	1	0	1	17
1	0	0	1	0	1	1	0	1	0	18
1	0	0	1	1	1	1	0	1	1	19

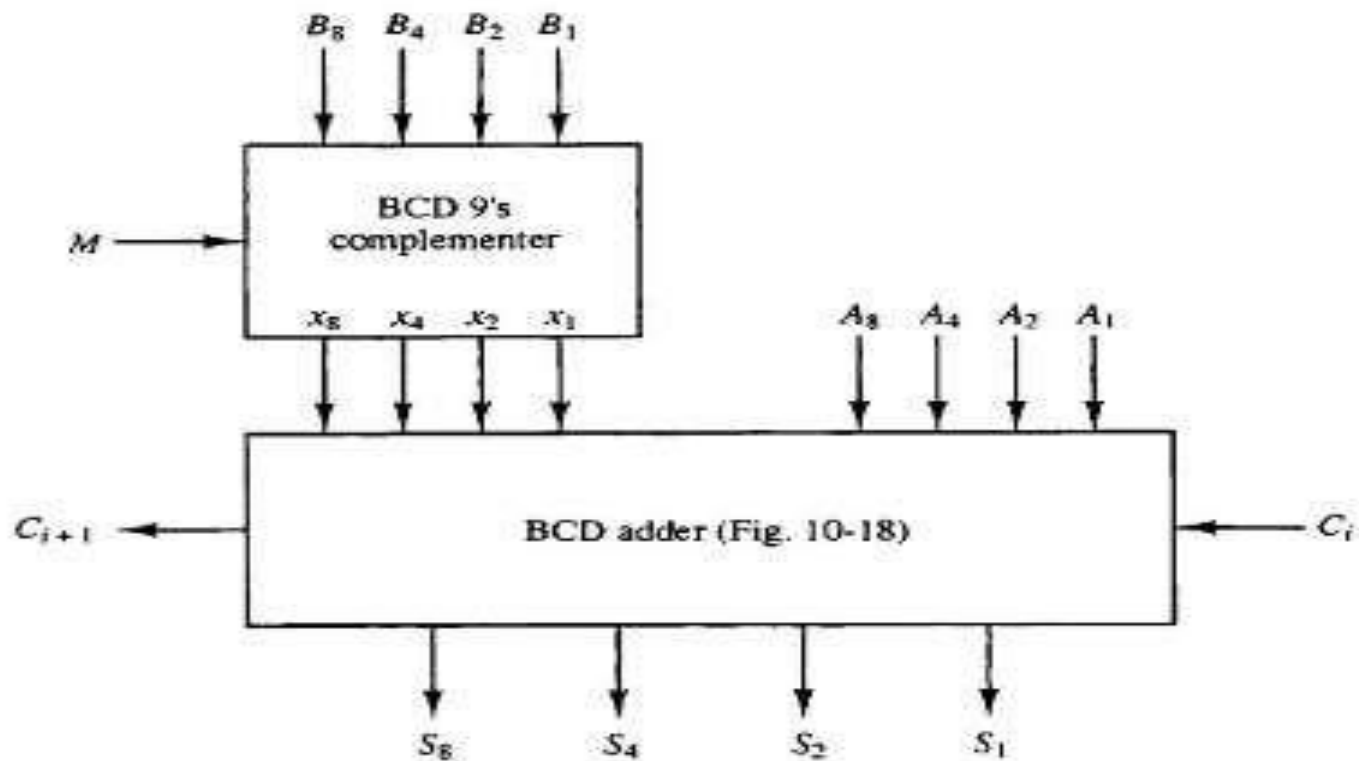


Block diagram of a BCD adder

In the above diagram,

- We'll use a **4-bit Binary-Adder** as an example, which accepts addend and augend bits as input and has a 'Carry in' input carry.
- The Binary-Adder generates **five outputs**: Z8, Z4, Z2, Z1, and a carry K output.
- The logical circuit is designed to identify the **C out** using the **output carry K and Z8, Z4, Z2, Z1 outputs**.
- The binary adder's **Z8, Z4, Z2, and Z1 outputs are fed into the 2nd 4-bit binary adder as an Augend**.
- The addend bit of the **2nd 4-bit** binary adder is designed in such a way that the **1st and the 4th bit** of the addend number are **0 and the 2nd and the 3rd bit** are the same as **C out**.
- When the value of **C out is 0**, the addend number will be **0000**, which generates the same result as the **1st 4-bit** binary number.
- But when the value of the **C out is 1**, the addend bit will be **0110(6)** which adds with the **augend** to get the **valid BCD number**.

Decimal Addition & Subtraction



One stage of Decimal Arithmetic Unit