Control Unit Design

Unit-3

Contents:part-1

1. Control Memory
2. Address Sequencing
3. Microprogram Example
4. Design of Control Unit
5. Hardwired Control
6. Microprogrammed Control Parallel Processing

1. Control Memory

Microprogram

- A program stored in memory that generates all the control signals required to execute the instruction set correctly.

- Consists of microinstructions.

Microinstruction

- Contains a control word and a sequencing word

Control Word - All the control information required for one clock cycle.

Sequencing Word - Information needed to decide the next microinstruction address.

- Vocabulary to write a microprogram

Control Memory (Control Storage: CS)

- Storage in the microprogrammed control unit to store the microprogram.

Writeable Control Memory (Writeable Control Storage:WCS)

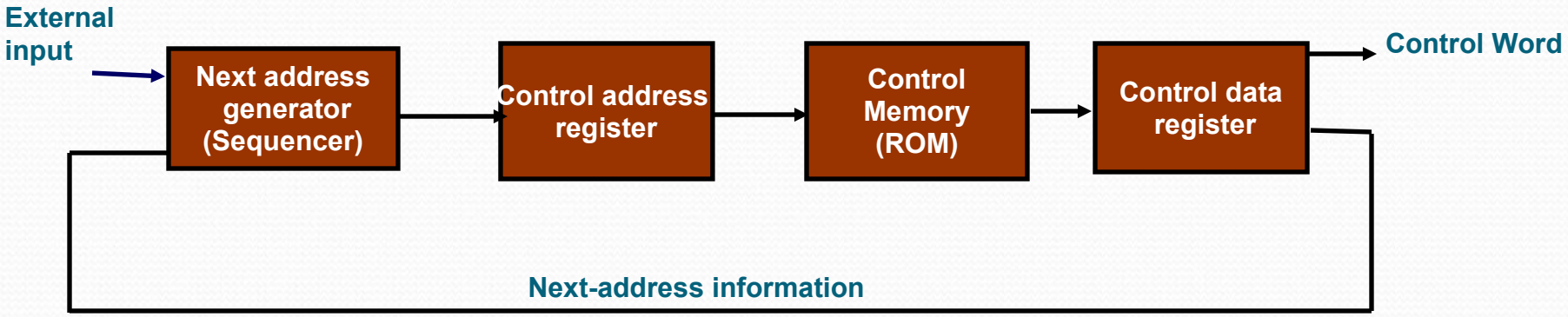
- CS whose contents can be modified

-> Allows the microprogram can be changed

-> Instruction set can be changed or modified

Microprogram

Microprogrammed Control Organization



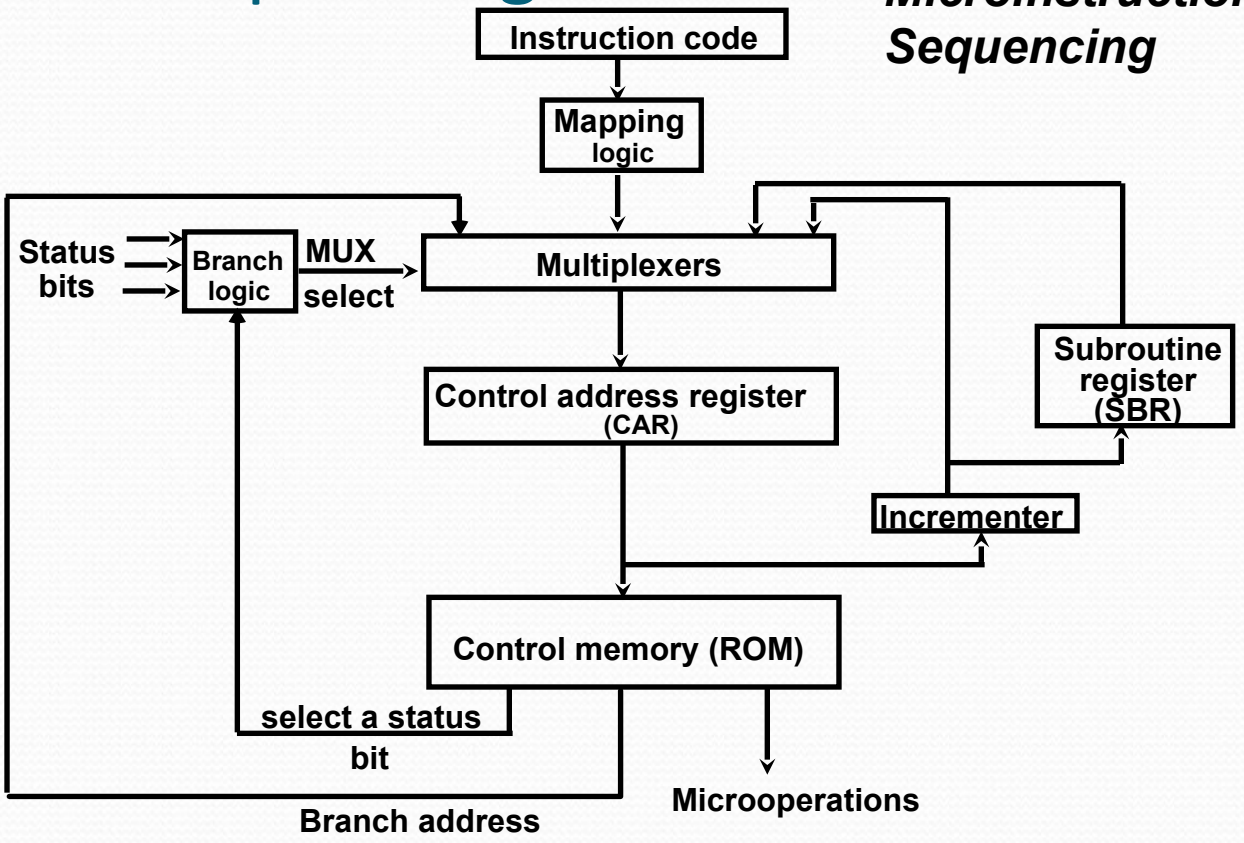
Dynamic Microprogramming

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

Sequencer (Microprogram Sequencer)

A Microprogram Control Unit that determines the Microinstruction Address to be executed in the next clock cycle.

2. Address Sequencing

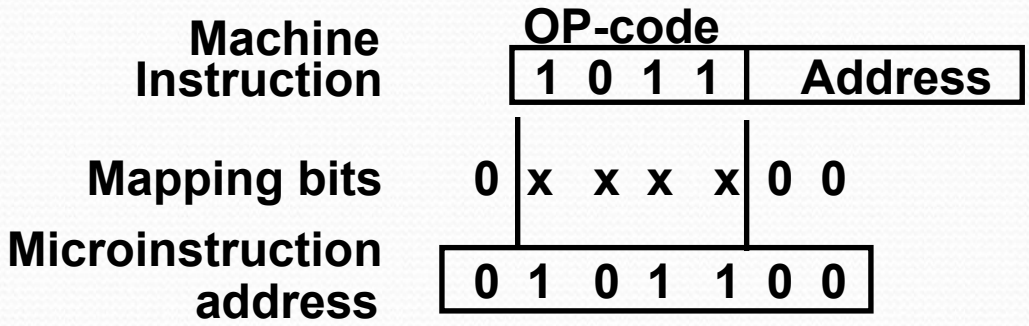


Sequencing Capabilities Required in a Control Storage

- Incrementing of the control address register
- Unconditional and conditional branches
- A mapping process from the bits of the machine instruction to an address for control memory
- A facility for subroutine call and return

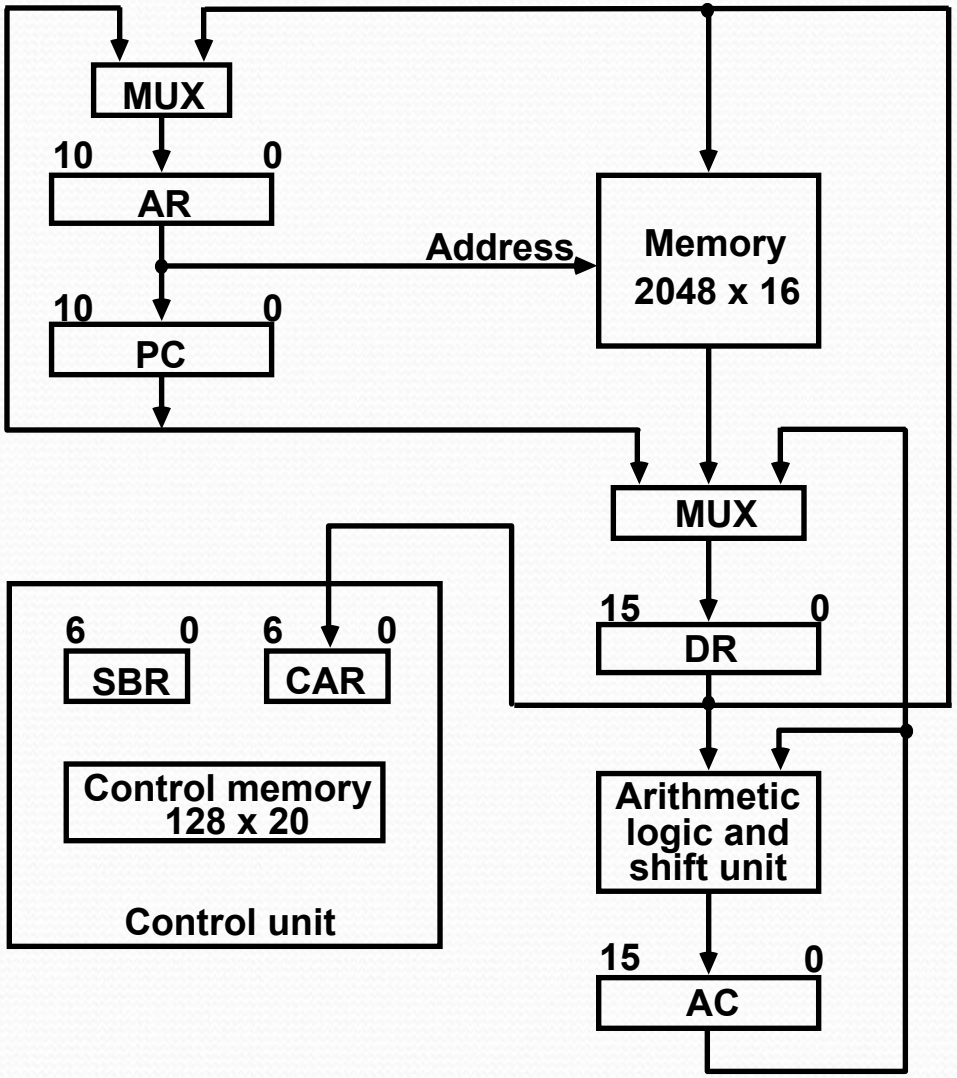
Mapping of instructions to microroutines

Mapping from the OP-code of an instruction to the address of the Microinstruction which is the starting microinstruction of its execution microprogram



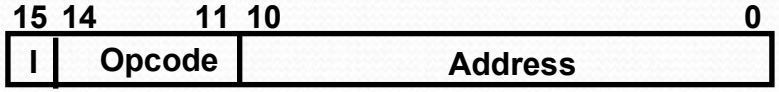
3. Microprogram Example

Computer Configuration



Machine Instruction Format

Machine instruction format

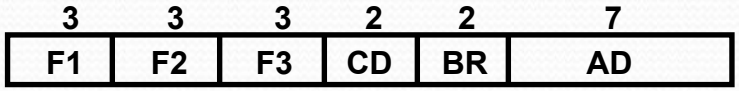


Sample machine instructions

Symbol	OP-code	Description
ADD	0000	AC ← AC + M[EA]
BRANCH	0001	if (AC < 0) then (PC ← EA)
STORE	0010	M[EA] ← AC
EXCHANGE AC	0011	AC ← M[EA], M[EA] ←

EA is the effective address

Microinstruction Format



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

Microinstruction Field Descriptions- F1,F2,F3

F1	Microoperation	Symbol
000	None	NOP
001	AC ← AC + DR	ADD
010	AC ← 0	CLRAC
011	AC ← AC + 1	INCAC
100	AC ← DR	DRTAC
101	AR ← DR(0-10)	DRTAR
110	AR ← PC	PCTAR
111	M[AR] ← DR	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	AC ← AC - DR	SUB
010	AC ← AC ∨ DR	OR
011	AC ← AC ∧ DR	AND
100	DR ← M[AR]	READ
101	DR ← AC	ACTDR
110	DR ← DR + 1	INCDR
111	DR(0-10) ← PC	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	AC ← AC ⊕ DR	XOR
010	AC ← AC'	COM
011	AC ← shl AC	SHL
100	AC ← shr AC	SHR
101	PC ← PC + 1	INCPC
110	PC ← AR	ARTPC
111	Reserved	

Microinstruction Field Descriptions - CD, BR

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

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

Symbolic Microinstructions

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

Sample Format

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

Label: may be empty or may specify a symbolic address terminated with a colon.

Micro-ops: consists of one, two, or three symbols separated by commas.

CD: one of {U, I, S, Z}, where

U: Unconditional Branch
I: Indirect address bit
S: Sign of AC
Z: Zero value in AC

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

AD: one of {Symbolic address, NEXT, empty}

Symbolic Microprogram – Fetch Routine

During FETCH, Read an instruction from memory and decode the instruction and update PC

Sequence of microoperations in the fetch cycle:

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

Symbolic microprogram for the fetch cycle:

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

Binary equivalents translated by an assembler

Binary address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

Symbolic Microprogram

- Control Storage: 128 20-bit words
- The first 64 words: Routines for the 16 machine instructions
- The last 64 words: Used for other purpose (e.g., fetch routine and other subroutines)
- Mapping: OP-code XXXX into 0XXXX00, the first address for the 16 routines are 0(0 0000 00), 4(0 0001 00), 8, 12, 16, 20, ..., 60

Partial Symbolic Microprogram

Label	Microops	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

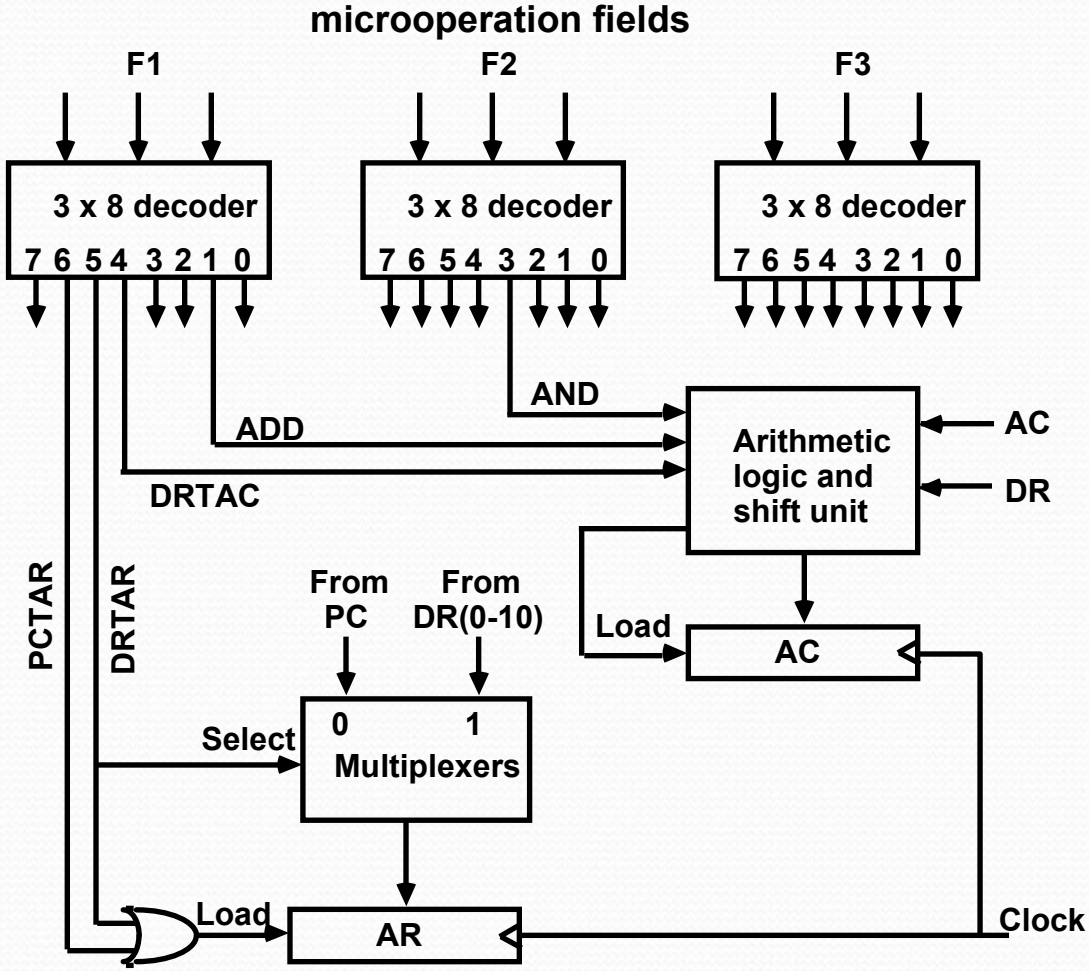
Binary Microprogram

Micro Routine	Address		Binary Microinstruction				
	Decimal	Binary	F1	F2	F3	CD	BR
AD							
ADD	0	0000000	000	000	000	01	01 1000011
	1	0000001	000	100	000	00	00 0000010
	2	0000010	001	000	000	00	00 1000000
	3	0000011	000	000	000	00	00 1000000
BRANCH	4	0000100	000	000	000	10	00 0000110
	5	0000101	000	000	000	00	00 1000000
	6	0000110	000	000	000	01	01 1000011
	7	0000111	000	000	110	00	00 1000000
STORE	8	0001000	000	000	000	01	01 1000011
	9	0001001	000	101	000	00	00 0001010
	10	0001010	111	000	000	00	00 1000000
	11	0001011	000	000	000	00	00 1000000
EXCHANGE	12	0001100	000	000	000	01	01 1000011
	13	0001101	001	000	000	00	00 0001110
	14	0001110	100	101	000	00	00 0001111
	15	0001111	111	000	000	00	00 1000000
FETCH	64	1000000	110	000	000	00	00 1000001
	65	1000001	000	100	101	00	00 1000010
	66	1000010	101	000	000	00	11 0000000

This microprogram can be implemented using ROM

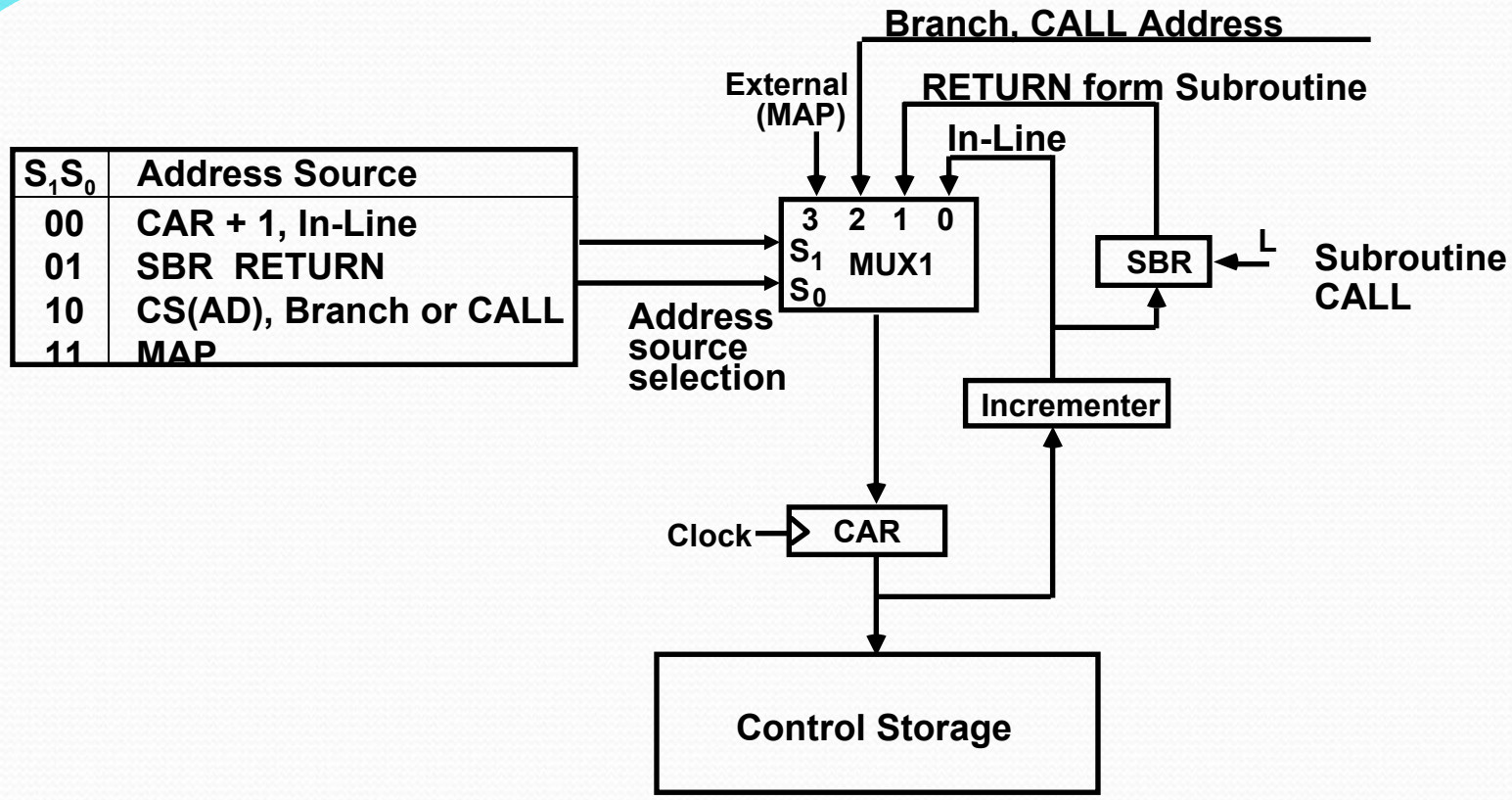
4. Design Of Control Unit

- DECODING ALU CONTROL INFORMATION -



Microprogram Sequencer

- NEXT MICROINSTRUCTION ADDRESS LOGIC -

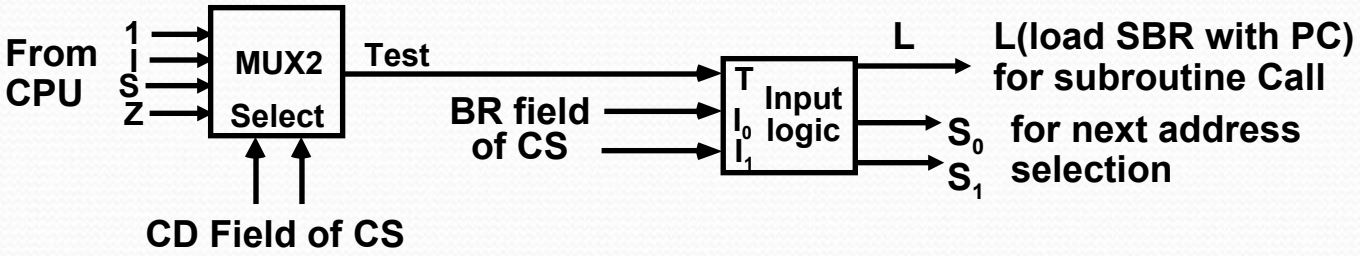


MUX-1 selects an address from one of four sources and routes it into a CAR

- In-Line Sequencing → CAR + 1
- Branch, Subroutine Call → CS(AD)
- Return from Subroutine → Output of SBR
- New Machine instruction → MAP

Microprogram Sequencer

- CONDITION AND BRANCH CONTROL -



Input Logic

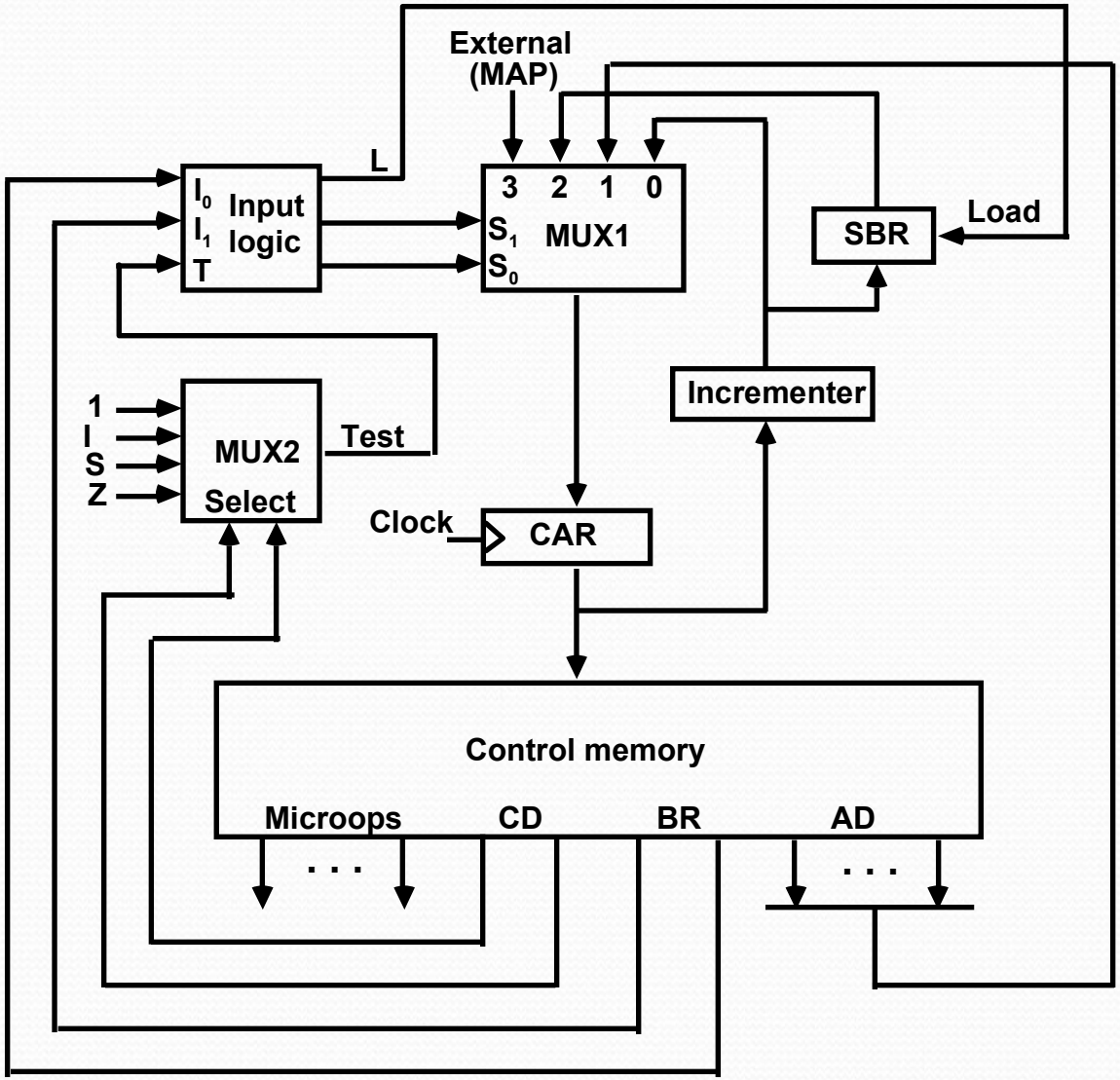
I_0I_1T	Meaning	Source of Address	S_1S_0	L
000	In-Line	CAR+1	00	0
001	JMP	CS(AD)	10	0
010	In-Line	CAR+1	00	0
011	CALL	CS(AD) and SBR <- CAR+1	10	1
10x	RET	SBR	01	0
11x	MAP	DR(11-14)	11	0

$S_0 = I_0$

$S_1 = I_0I_1 + I_0'T$

$L = I_0'I_1T$

Microprogram Sequencer



Microinstruction Format

Information in a Microinstruction

- **Control Information**
- **Sequencing Information**
- **Constant**

  Information which is useful when feeding into the system

These information needs to be organized in some way for

- **Efficient use of the microinstruction bits**
- **Fast decoding**

Field Encoding

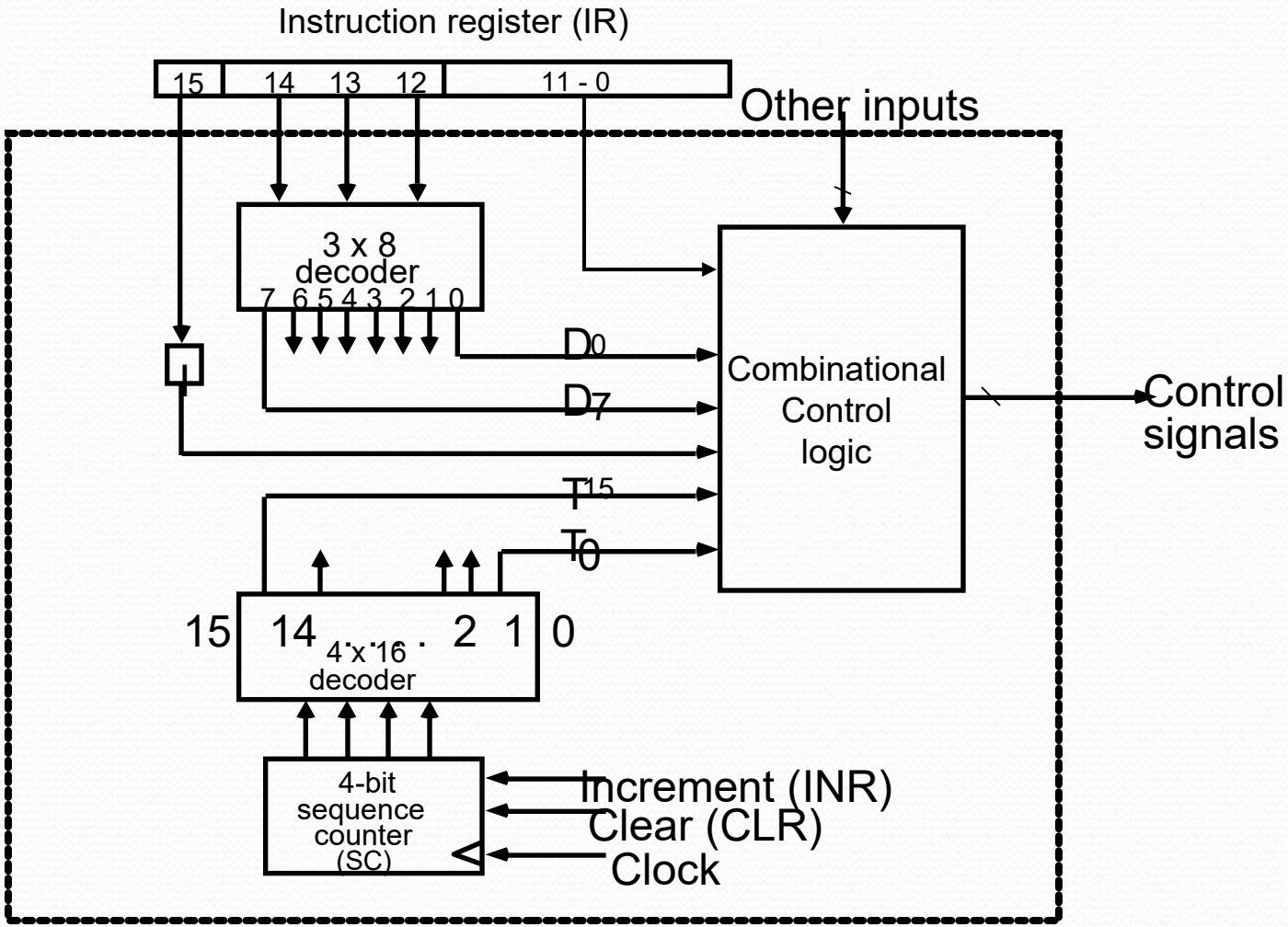
- **Encoding the microinstruction bits**
- **Encoding slows down the execution speed due to the decoding delay**
- **Encoding also reduces the flexibility due to the decoding hardware**

5. Hardwired control (CONTROL UNIT)

- Control unit (CU) of a processor translates from machine instructions to the control signals for the microoperations that implement them.
- Control units are implemented in one of two ways.
- *Hardwired* Control
  - CU is made up of sequential and combinational circuits to generate the control signals.
- *Microprogrammed* Control
  - A control memory on the processor contains microprograms that activate the necessary control signals.
- We will consider a hardwired implementation of the control unit for the Basic Computer.

Timing and Control

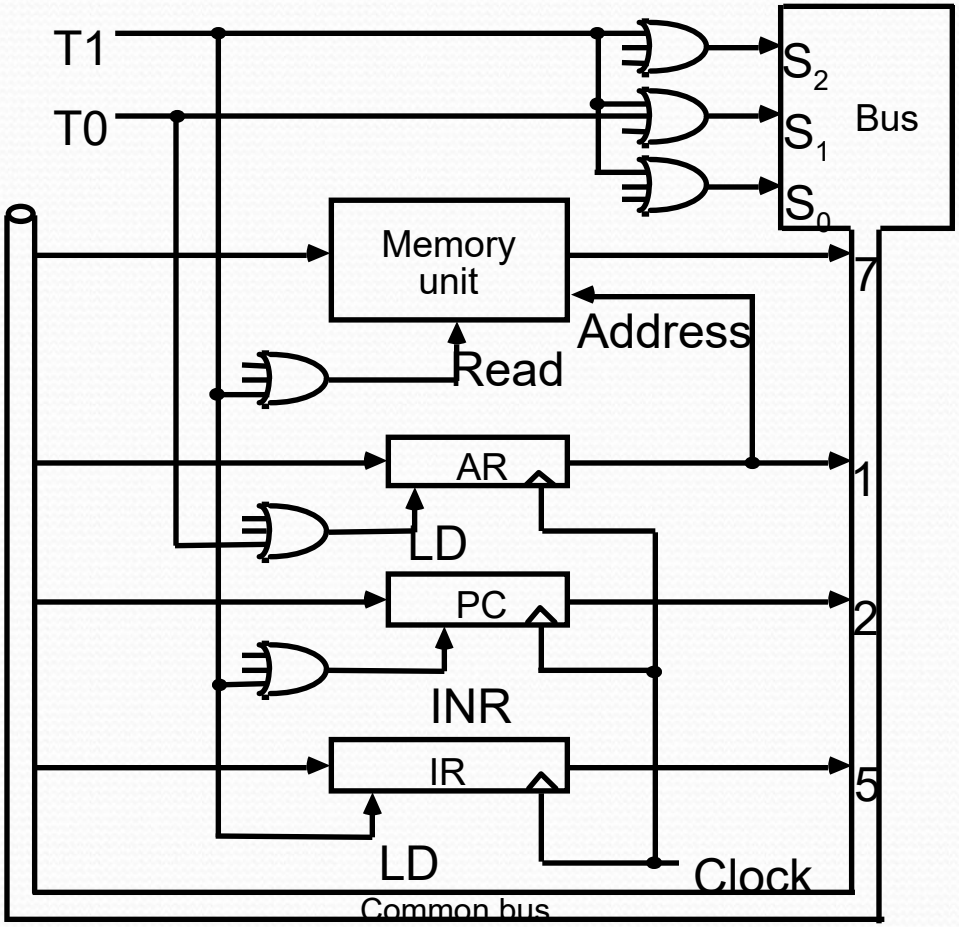
Control unit of Basic Computer



Fetch and Decode

- Fetch and Decode

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



Computer Arithmetic

Unit-III

Contents:part-2

1. Addition and subtraction
2. Multiplication algorithms
3. Division algorithms
4. Floating point arithmetic operations
5. Decimal arithmetic unit
6. Decimal arithmetic operations

1. Addition and subtraction with signed-magnitude data

- Addition algorithm:
- When the signs of A and B are **identical**, add the magnitudes and attach the sign of A to the result.
- When the signs of A and B are **different**, compare the magnitudes and subtract the smaller number from the larger.

  Case 1: Choose the sign of the result to be the same as A if A > B

  Case 2: Choose the sign of the result to be the complement of the sign of A if A < B.

  Case 3: If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Examples

- Ex: $(+2) + (+3) = +5$
- Ex : $(-2) + (-3) = (-5)$
- Ex: $(+3) + (-2) = +1$
- Ex: $(-2) + (+3) = +1$
- Ex : $(+3) + (-5) = (-2)$
- Ex : $(-5) + (+3) = (-2)$

Subtraction algorithm

- When the signs of A and B are **different**, add the magnitudes and attach the sign of A to the result.
- When the signs of A and B are **identical**, compare the magnitudes and subtract the smaller number from the larger.

  Case 1: Choose the sign of the result to be the same as A if A > B

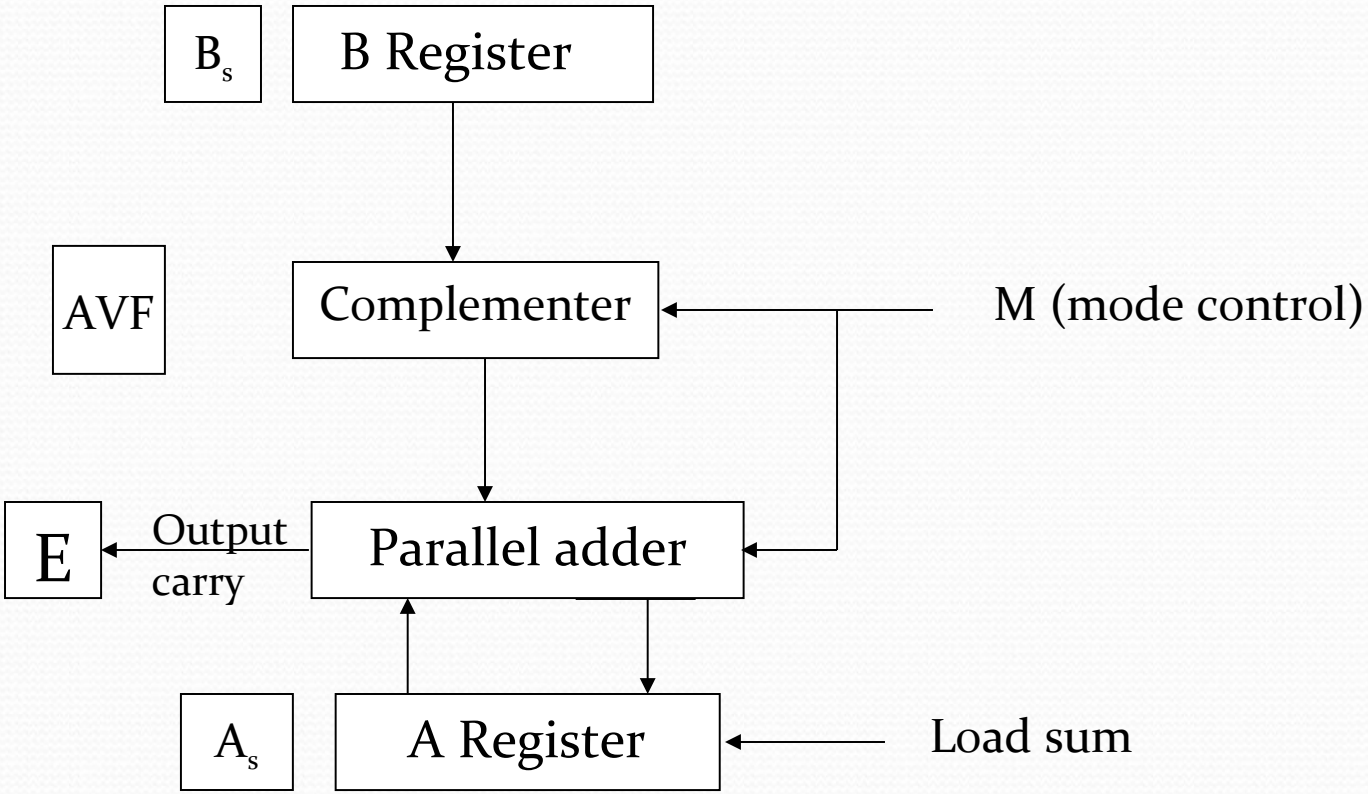
  Case 2: Choose the sign of the result to be the complement of the sign of A if A < B.

  Case 3: If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Examples

- Ex: (-2) - (3) = (+5)
- Ex : (2) - (-3) = (5)
- Ex: (3) - (2) = 1
- Ex: (2) - (3) = -1
- Ex : (-3) - (-5) = 2
- Ex : (-5) - (-3) = -2

Hardware implementation



Hardware implementation

- The addition of A and B is done through 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 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 + B +1 . This is equal to A plus the 2's complement of B, which is equivalent to the subtraction A – B .

TABLE 10-1 Addition and Subtraction of Signed-Magnitude Numbers

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

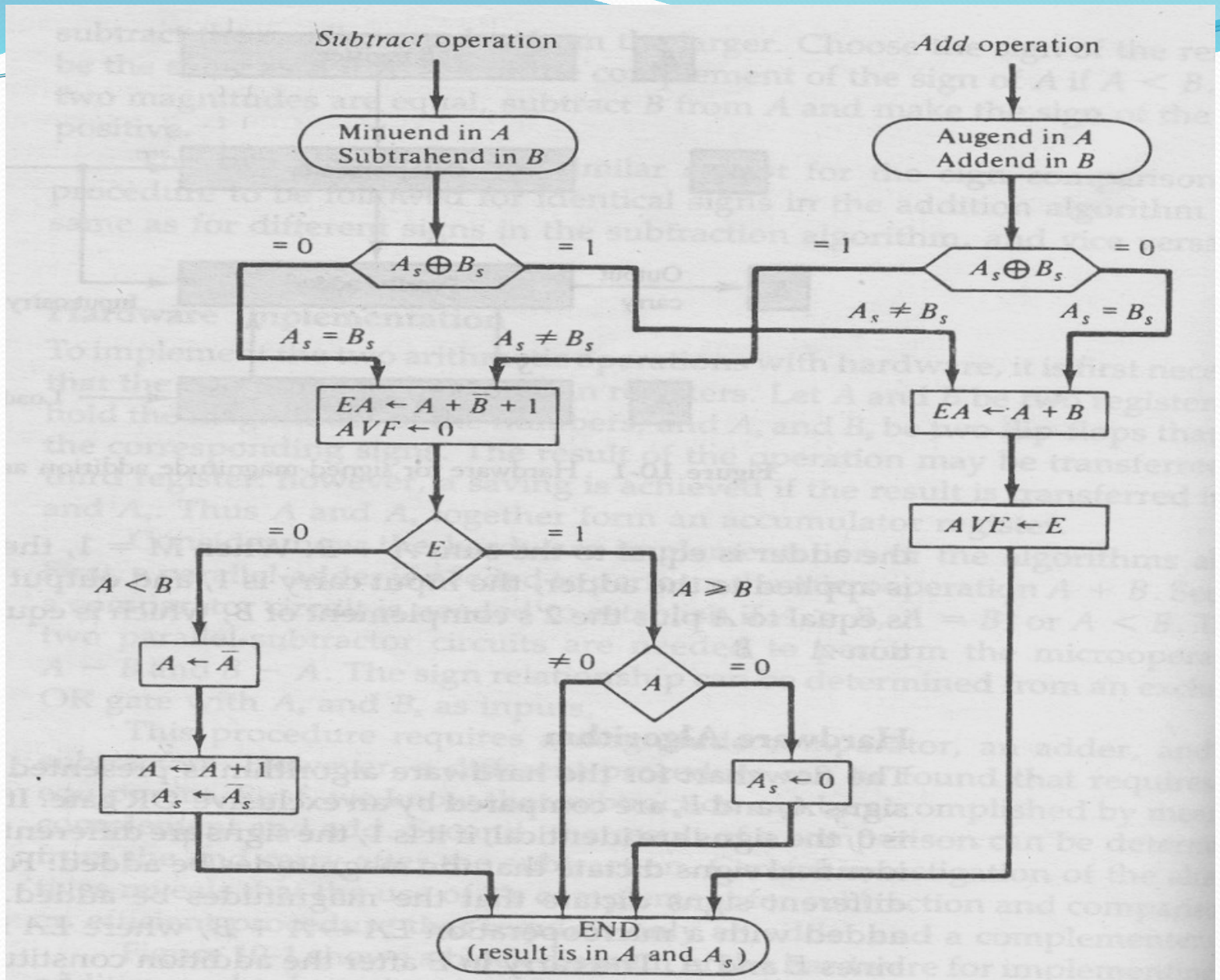


Figure 10-2 Flowchart for add and subtract operations.

Addition and subtraction with signed 2's complement data

- In signed 2's complement representation, the leftmost bit of a binary number represents the sign bit: 0 for positive and 1 for negative.
- If the sign bit is 1, the entire number is represented in 2's complement form.
- Ex. +33 is represented by 0 0 1 0 0 0 0 1

  and -33 is represented by 1 1 0 1 1 1 1 1
- **Note** : 1 1 0 1 1 1 1 1 is 2's complement of 0 0 1 0 0 0 0 1

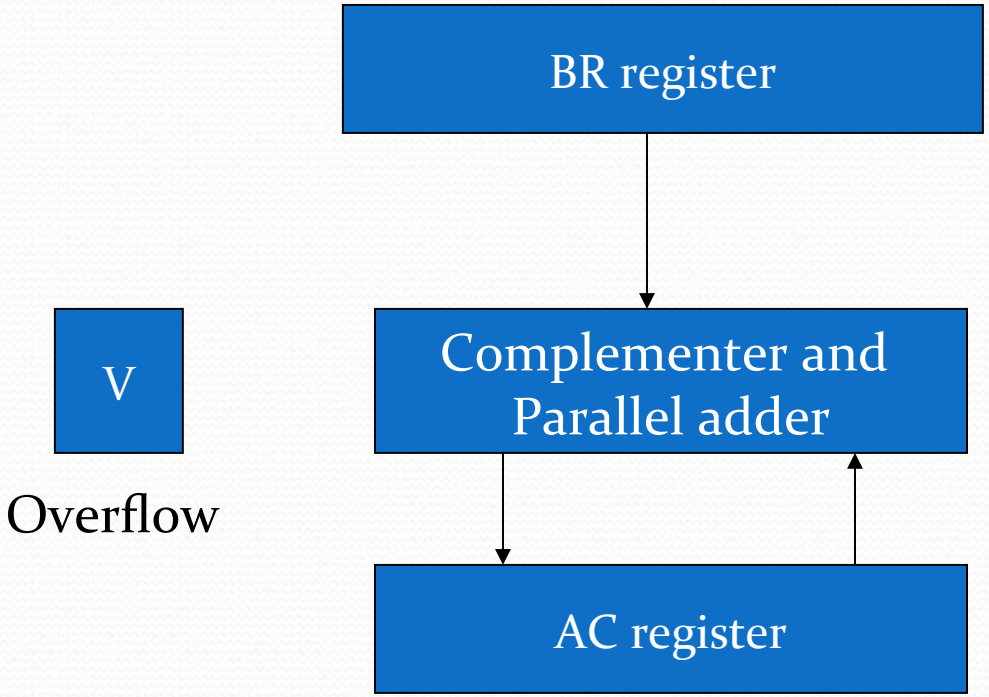
  and vice versa.

Contd.,

- The addition of two numbers in signed-2's complement form consists of adding the numbers with the sign bits treated the same as other bits of the number.
- A carry-out of the sign-bit position is discarded.
- The subtraction 'A – B' consists of first taking the 2's complement of the subtrahend B and then adding to the minuend A.
- **Overflow**

  when the last two carries are applied to an exclusive OR gate, the overflow is detected when the output of the gate is equal to 1.

Hardware for signed 2's complement addition and subtraction

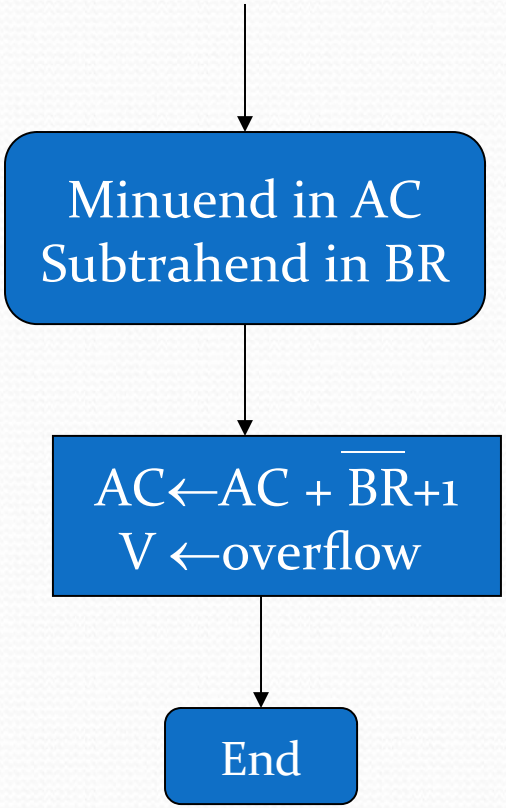


Contd.,

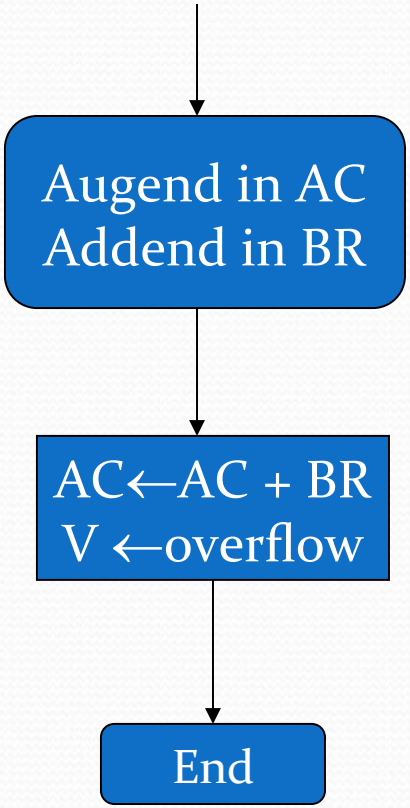
- The register configuration for the hardware implementation is shown.
- We name the A register AC (accumulator) and the B register BR.
- The leftmost bits in AC and BR represent the sign bits of the numbers.
- The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder .
- The overflow bit is set to 1 if there is an overflow.
- The output carry in this case is discarded.

Algorithm for signed 2's complement addition and subtraction – flowchart

- Subtract



Add



2. Multiplication algorithms

- **Hardware implementation for signed-magnitude data**
- 1) Instead of providing registers to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers and successively accumulate the partial products in the registers.
- 2) Instead of shifting the multiplicand to the left, the partial product is shifted to the right, which results in leaving the partial product and the multiplicand in the required relative positions.
- 3) When the corresponding bit of the multiplier is 0, there is no need to add all zeros to the partial product since it will not alter its value.

Contd.,

- 4) Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier.

The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

Multiplication of two fixed-point binary numbers in Signed-magnitude

1011 Multiplicand (11_{10})

x 1101 Multiplier (13_{10})

1011 Partial products

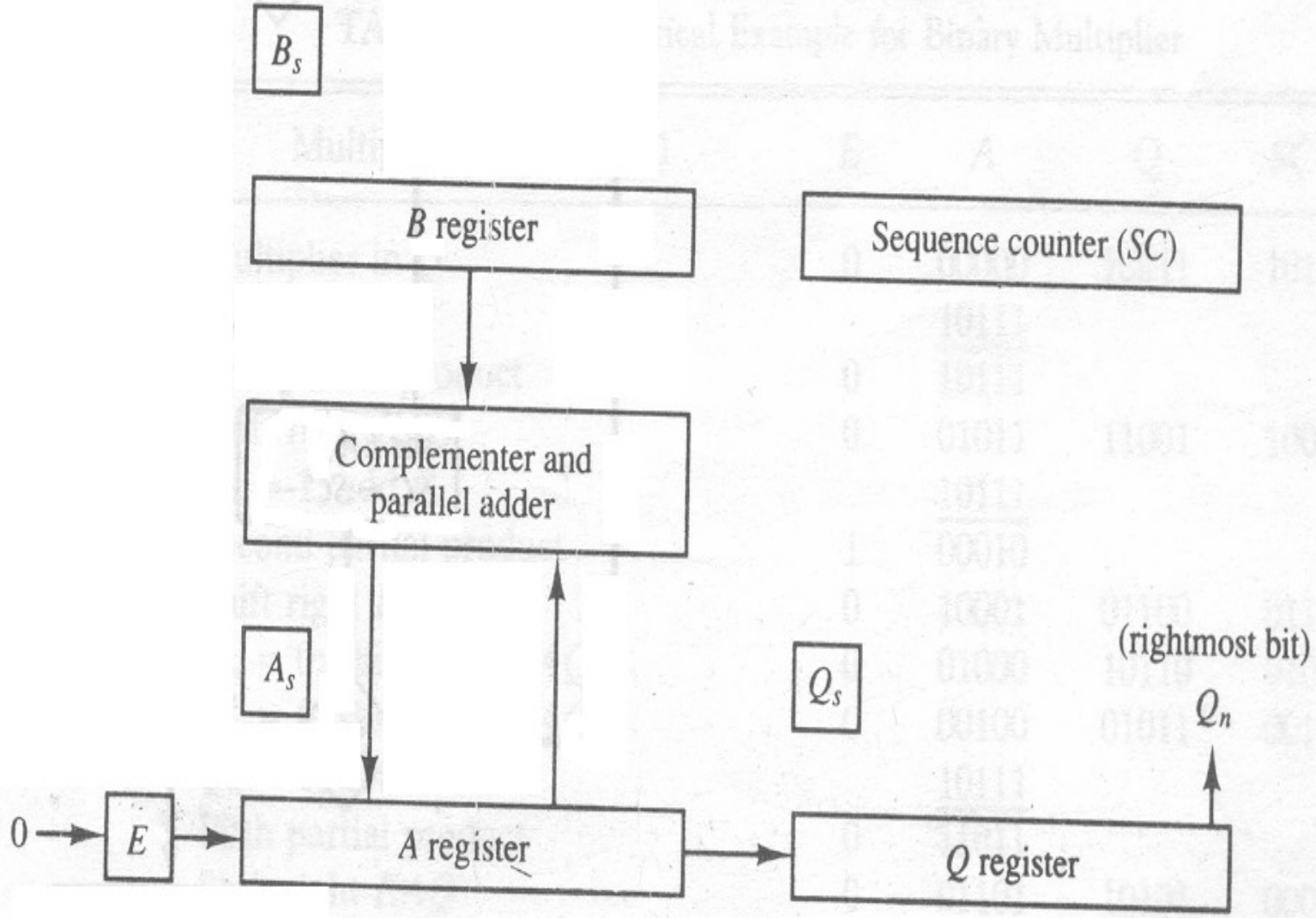
0000 Note: if multiplier bit is 1

1011 copy multiplicand (place value)

1011 otherwise zero

10001111 Product (143_{10})

Figure 10-5 Hardware for multiply operation.



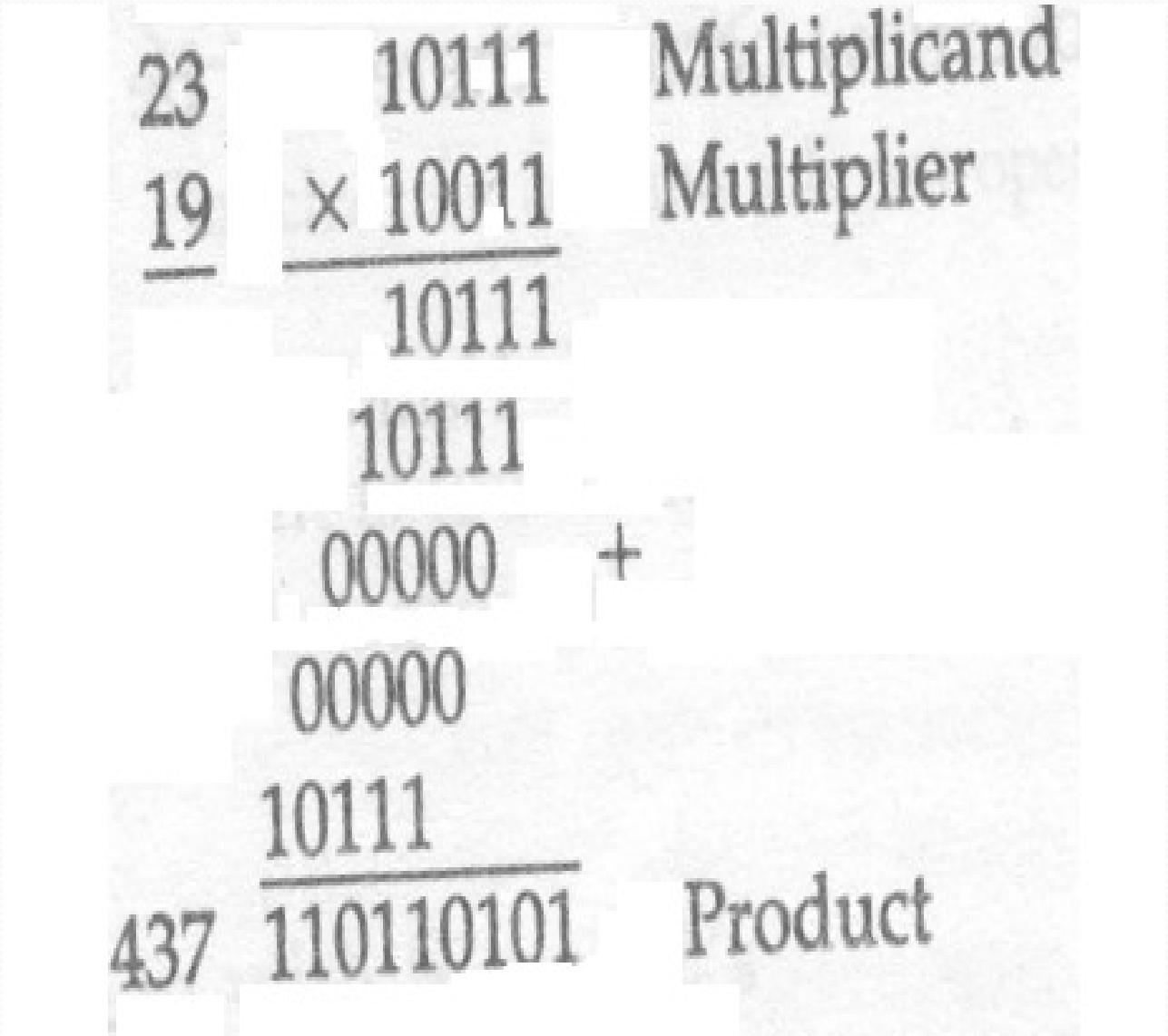

23 10111 Multiplicand
19 × 10011 Multiplier
10111
10111
00000 +
00000
10111
437 110110101 Product

Figure 10-6 Flowchart for multiply operation.

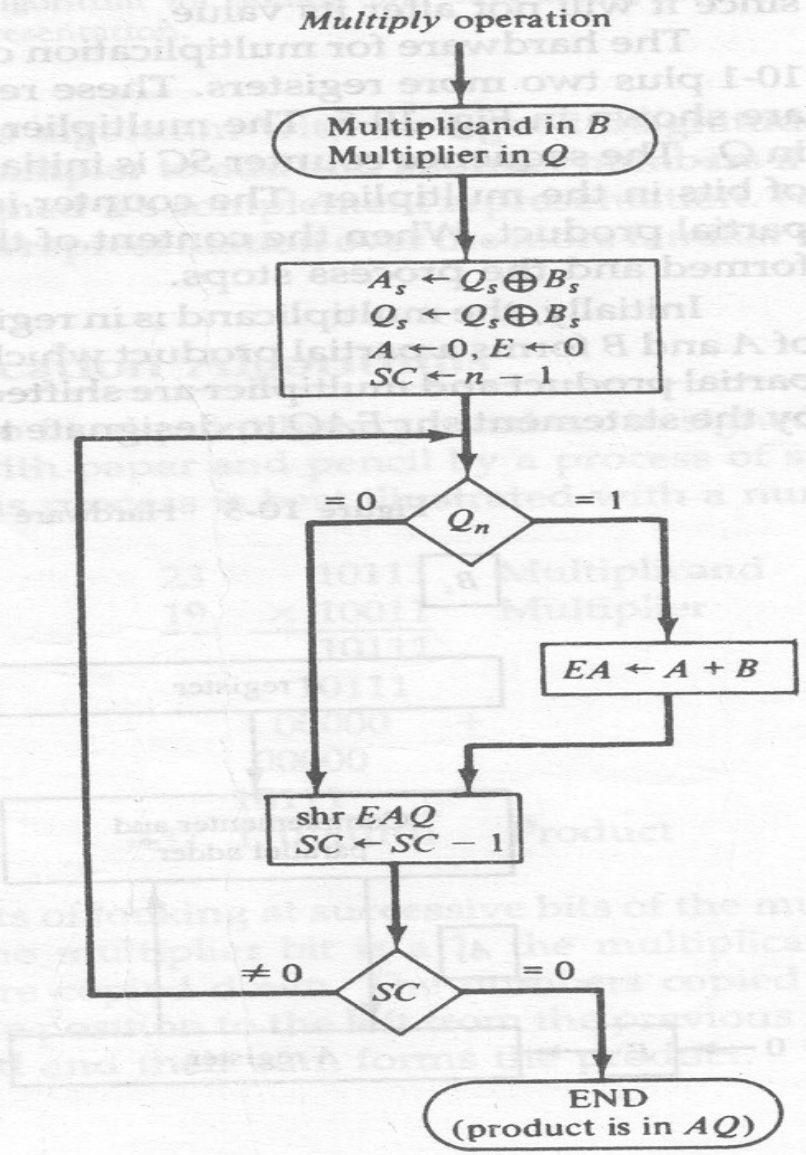


TABLE 10-2 Numerical Example for Binary Multiplier

Multiplicand $B = 10111$	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		10111		
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		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in $AQ = 0110110101$				

Booth multiplication algorithm

- **Booth algorithm** gives a procedure for multiplying binary integers in signed-2's complement representation.

- It operates on the fact that strings of zeros in the multiplier require no addition but just shifting, and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$.

- For example, the number 001110 (+14) has a string of 1's from 2^3 to 2^1 (k=3, m=1) .

  This number can be represented as $2^{k+1} - 2^m = 2^4 - 2^1 = 16 - 2 = 14$.

Contd.,

- Therefore, the multiplication $M \times 14$, where M is the multiplicand and 14 is the multiplier, can be done as $M \times 2^4 - M \times 2^1$.

- Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

- **Booth algorithm** requires examination of the multiplier bits and shifting of the partial product.

  Prior to the shifting, the multiplicand may be added to the partial product, or subtracted from the partial product, or left unchanged according to the following rules.

Contd.,

- 1) The multiplicand is **subtracted** from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.

- 2) The multiplicand is **added** to the partial product upon encountering the first 0 (provided that there was a previous 1)in a string of 0's in the multiplier.

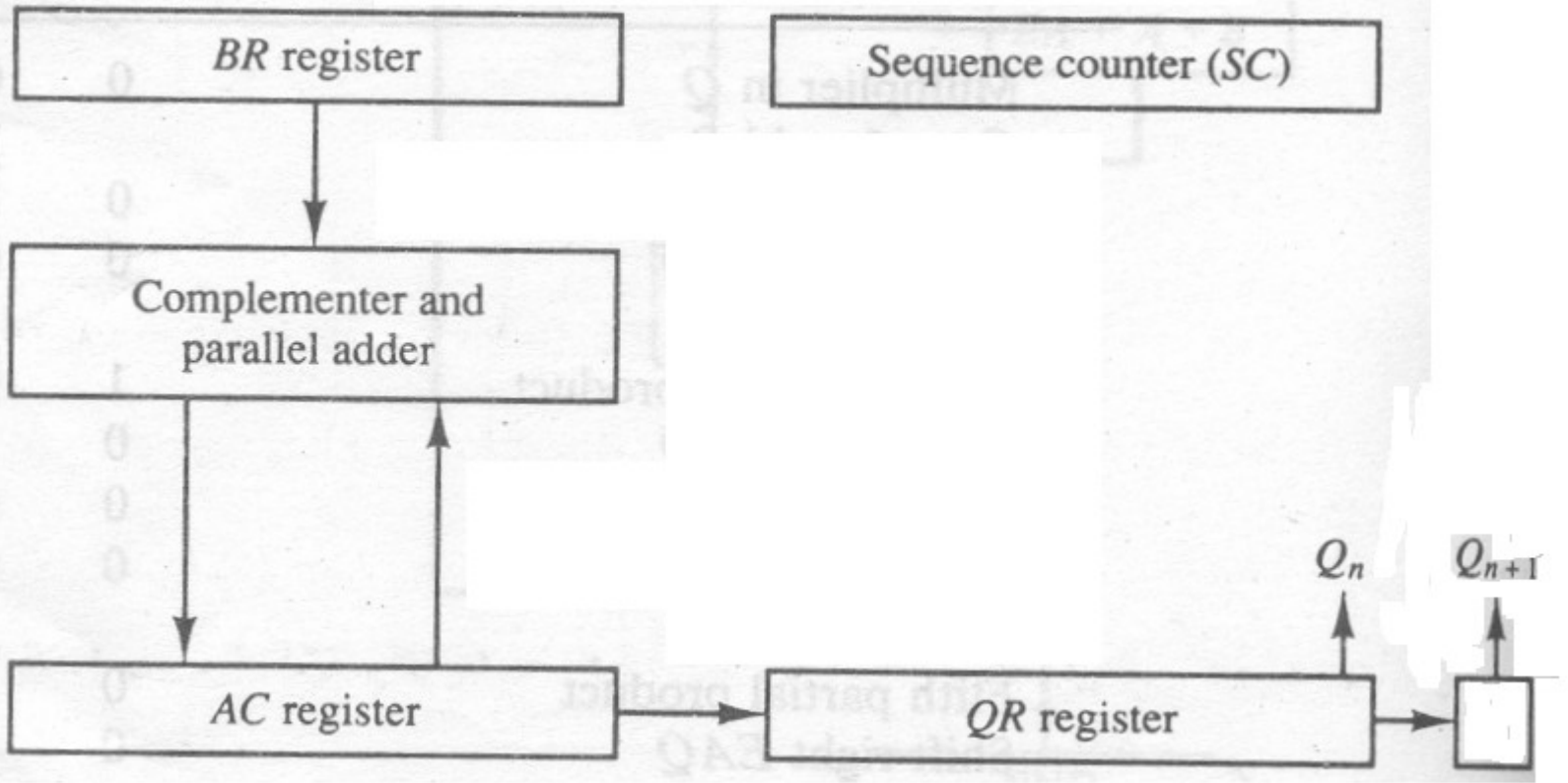
- 3) The partial product **does not change** when the multiplier bit is identical to the previous multiplier bit.

- Note : The algorithm works for positive or negative multipliers in 2's complement representation.

Hardware implementation

- The hardware implementation of Booth algorithm requires the register configuration shown here.

Figure 10-7 Hardware for Booth algorithm.



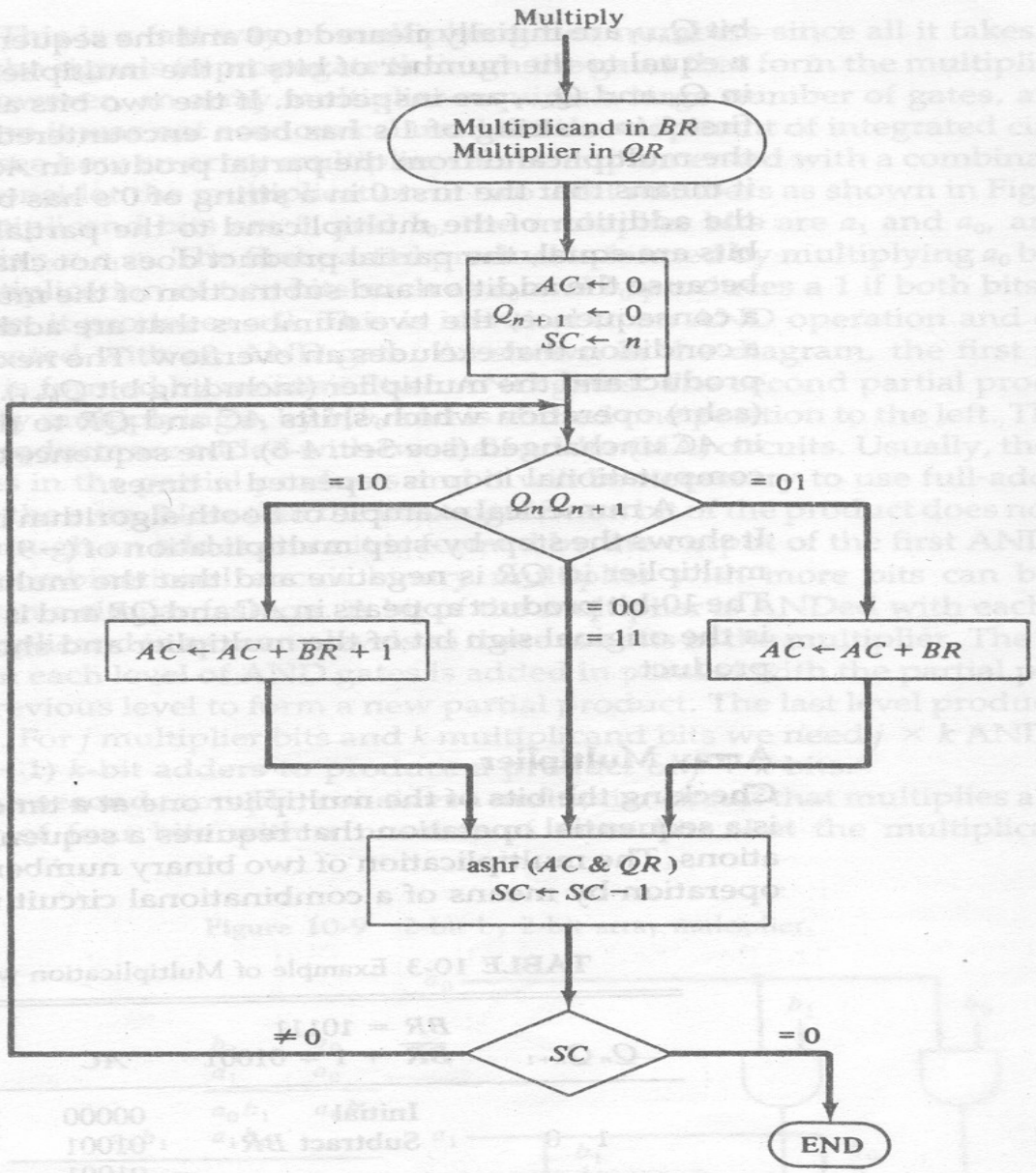


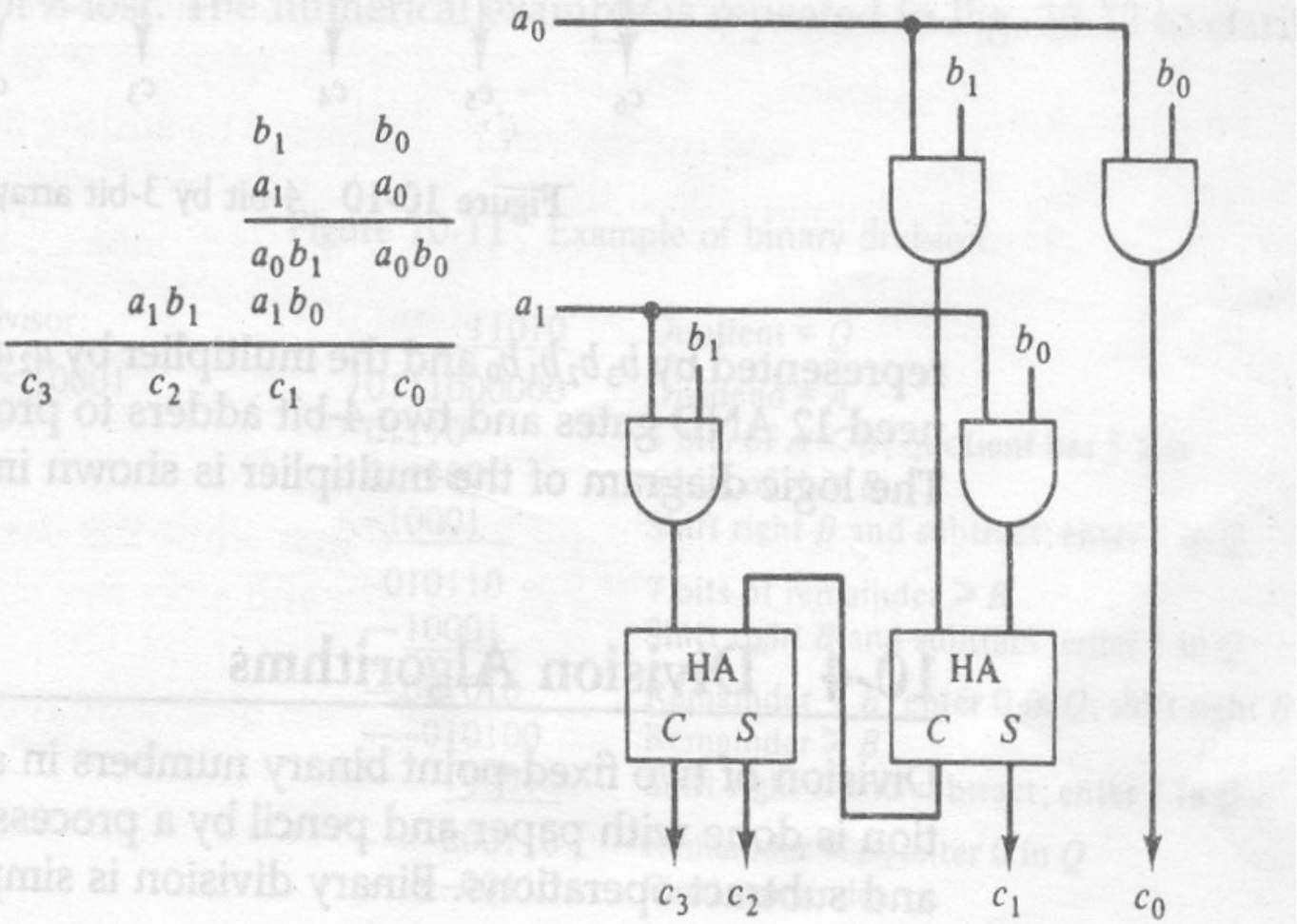
Figure 10-8 Booth algorithm for multiplication of signed-2's complement numbers.

TABLE 10-3 Example of Multiplication with Booth Algorithm

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

Array Multiplier

Figure 10-9 2-bit by 2-bit array multiplier.



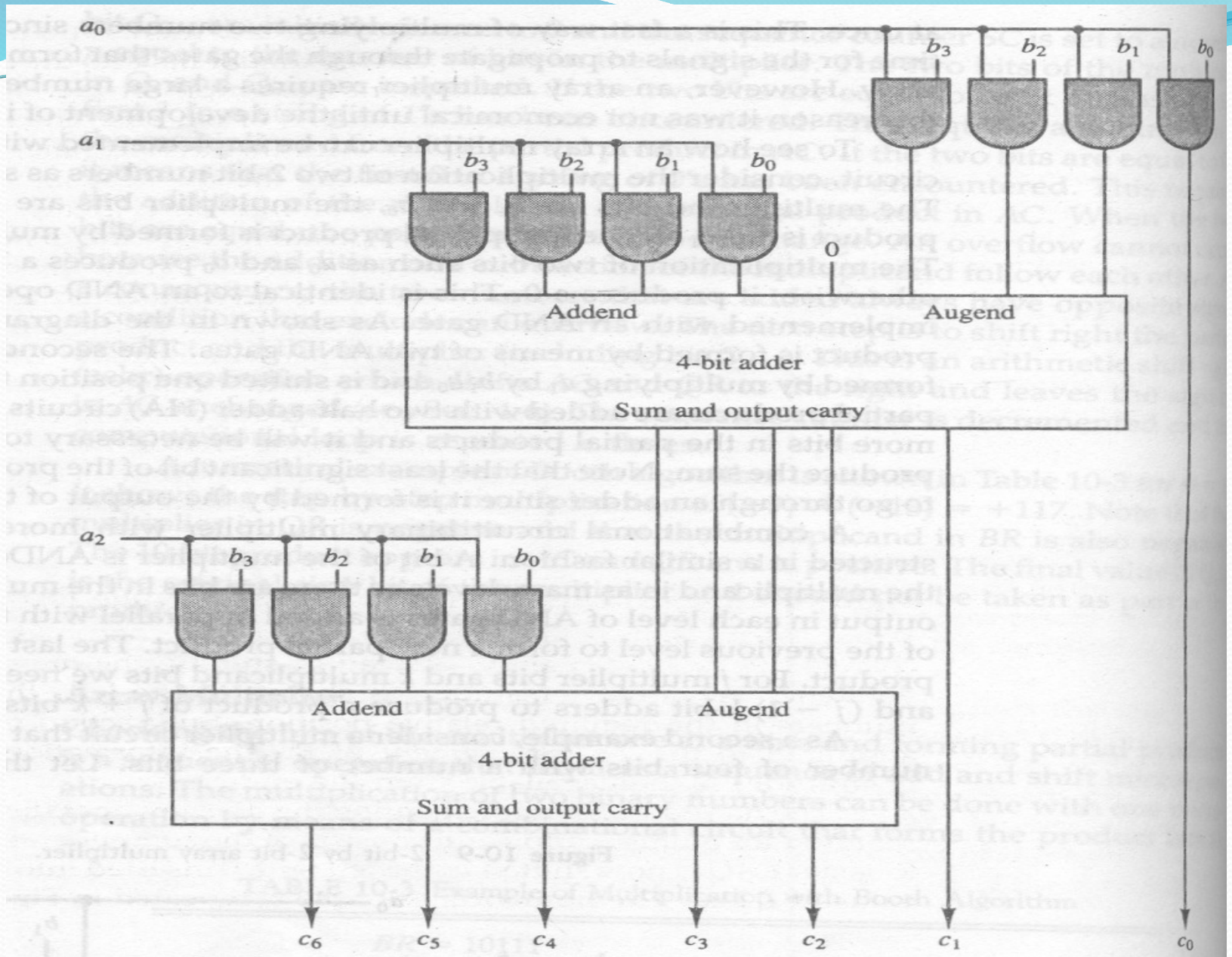


Figure 10-10 4-bit by 3-bit array multiplier.

3. Division Algorithms

Division of Unsigned Binary Integers

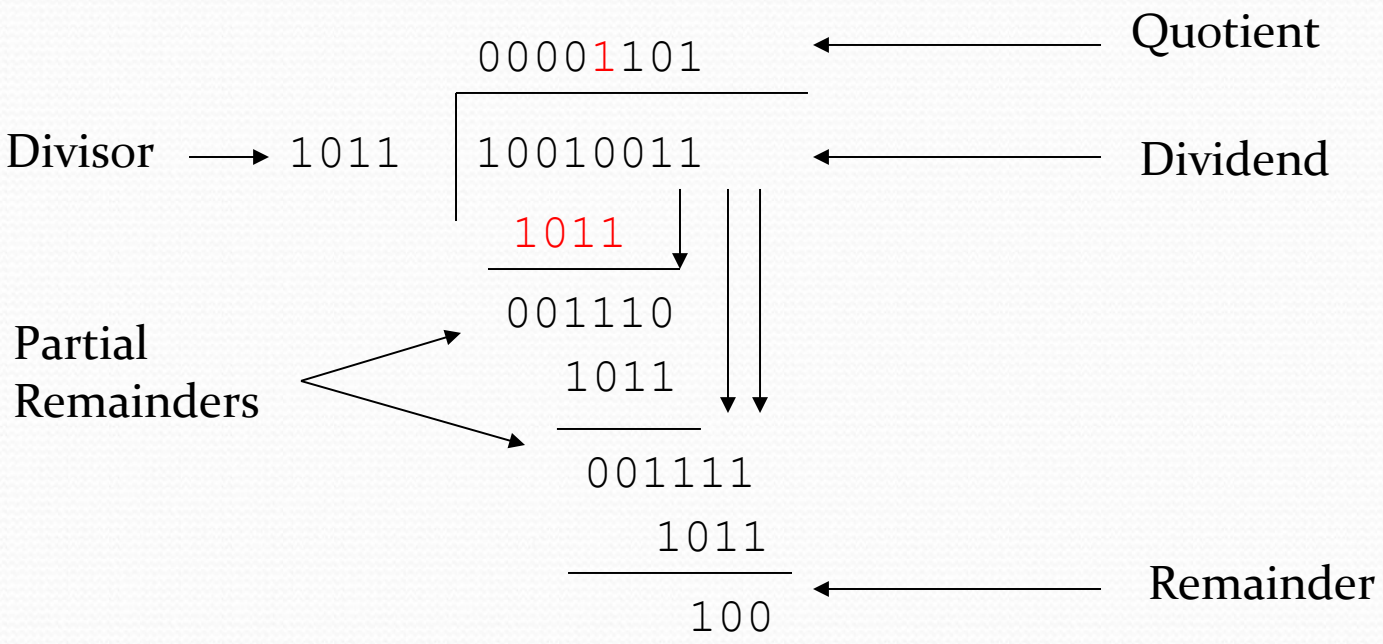
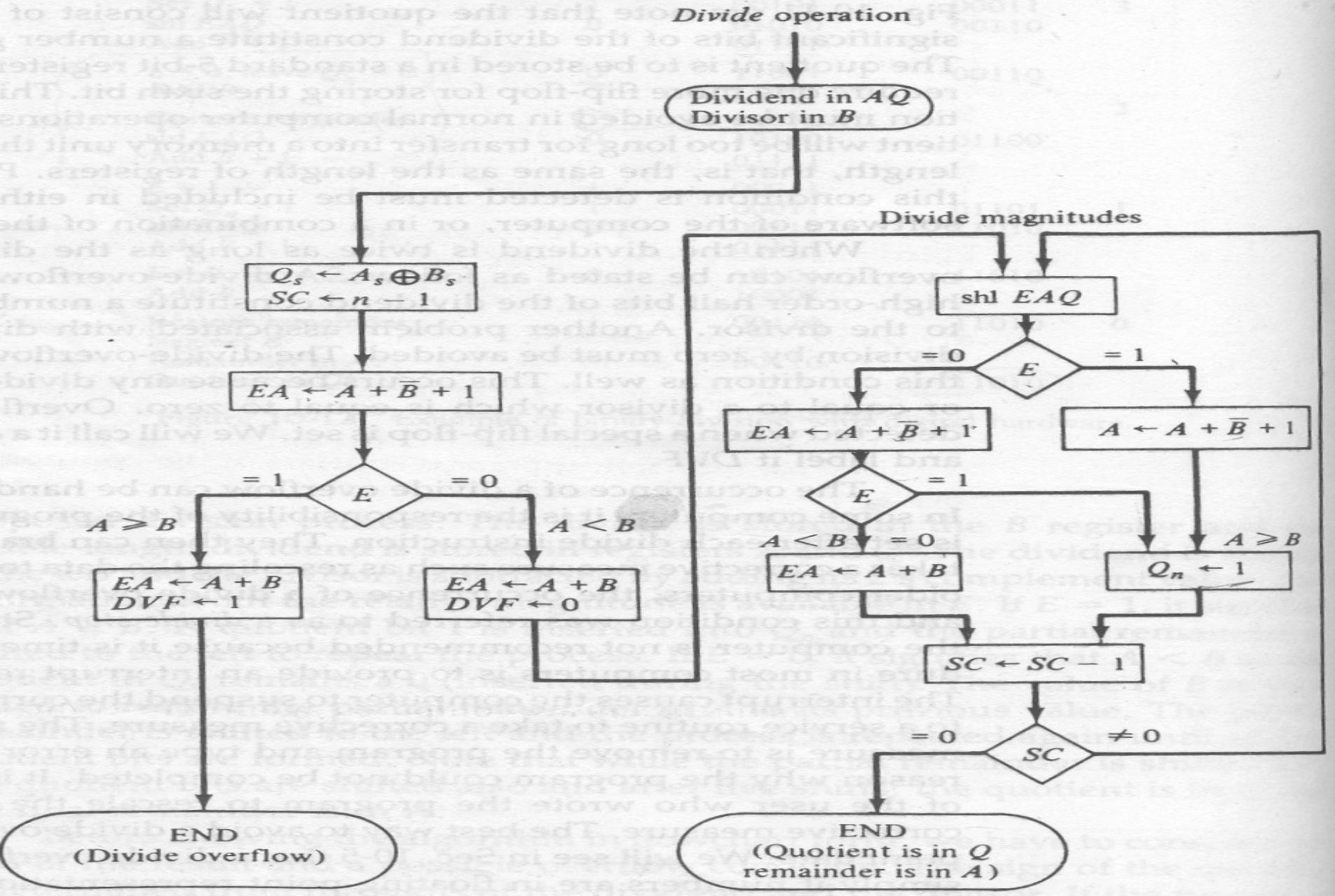


Figure 10-11 Example of binary division.

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

Figure 10-13 Flowchart for divide operation.



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

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

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

4. Floating point Arithmetic operations.

- A **floating point number** in computer registers consists of two parts:

  a mantissa **m** and an exponent **e**.

- The two parts represent a number obtained from multiplying m times a radix r raised to the value of e; thus

  $$m \times r^e$$

- Ex: The decimal number 537.25 is represented in a register as

  $$.53725 \times 10^3$$

- A floating point number is **normalized** if the most significant digit of the mantissa is nonzero.
- The number **zero** cannot be normalized.
- Floating point representation **increases the range of numbers** that can be accommodated in a register.

Alignment

- Adding or subtracting two numbers requires first an **alignment** of the radix point since the exponent parts must be made equal before adding or subtracting the mantissas.
- The alignment is done by shifting one mantissa while its exponent is adjusted until it is equal to the other exponent.
- Ex. $.5372400 \times 10^{2}$

  $+ \ .1580000 \times 10^{-1}$

$$.5372400 \times 10^{2}$$

$$+ \ .0001580 \times 10^{2}$$

$$.5373980 \times 10^{2}$$

Overflow and underflow

- When two normalized mantissas are added, the sum may contain an **overflow** digit.

- An **overflow** can be corrected easily by shifting the sum once to the right and incrementing the exponent.

- Ex . $.5372400 \times 10^2$

$$\underline{+ \quad .5700000 \times 10^2}$$

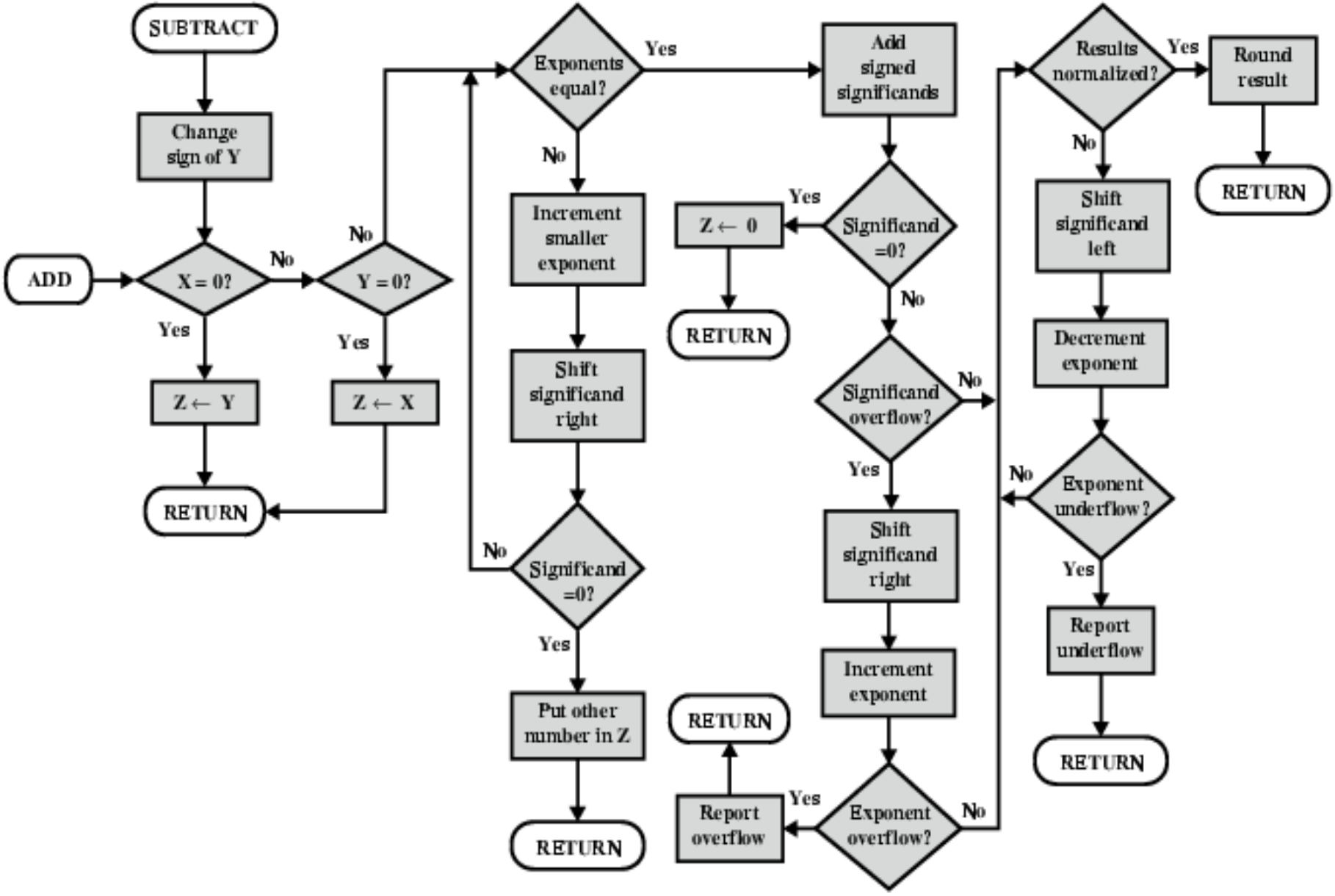
$$1.1072400 \times 10^2$$

$$= \quad .11072400 \times 10^3$$

Contd.,

- A floating point number that has a zero in the most significant position of the mantissa is said to have an **underflow**.
- To normalize a number that contains an **underflow**, it is necessary to **shift the mantissa to the left** and **decrement the exponent until a non zero digit appears in the first position.**
- Ex . $.00350 \times 10^6$

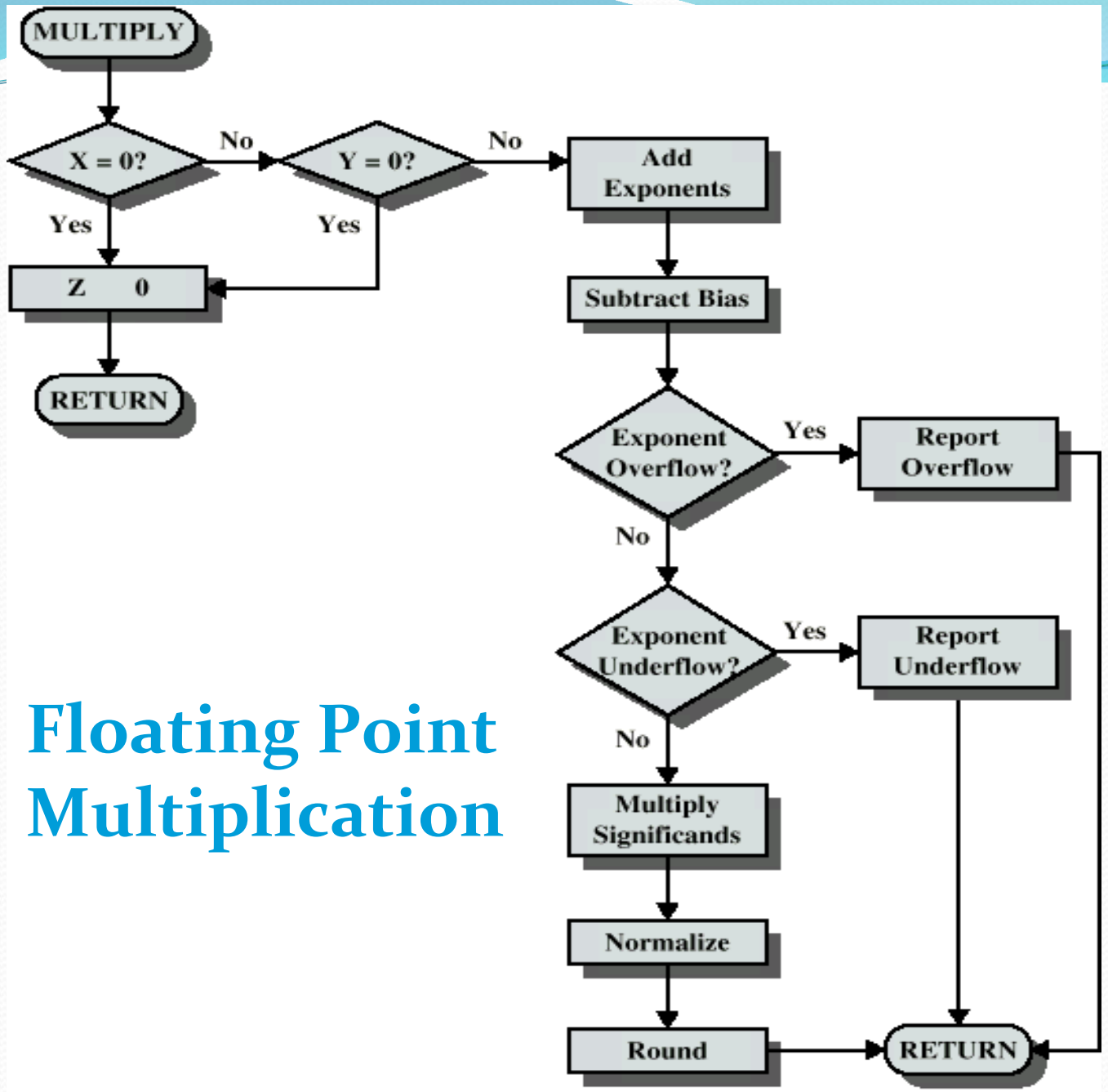
  $= .35000 \times 10^4$


SUBTRACT
Change sign of Y
ADD
X = 0?
No
Yes
Z ← Y
RETURN
Y = 0?
No
Yes
Z ← X
Exponents equal?
Yes
No
Increment smaller exponent
Shift significand right
Significand =0?
No
Yes
Put other number in Z
RETURN
Add signed significands
Significand =0?
Yes
Z ← 0
RETURN
No
Significand overflow?
No
Yes
Shift significand right
Increment exponent
Exponent overflow?
Yes
Report overflow
RETURN
No
Results normalized?
Yes
Round result
RETURN
No
Shift significand left
Decrement exponent
Exponent underflow?
No
Yes
Report underflow
RETURN

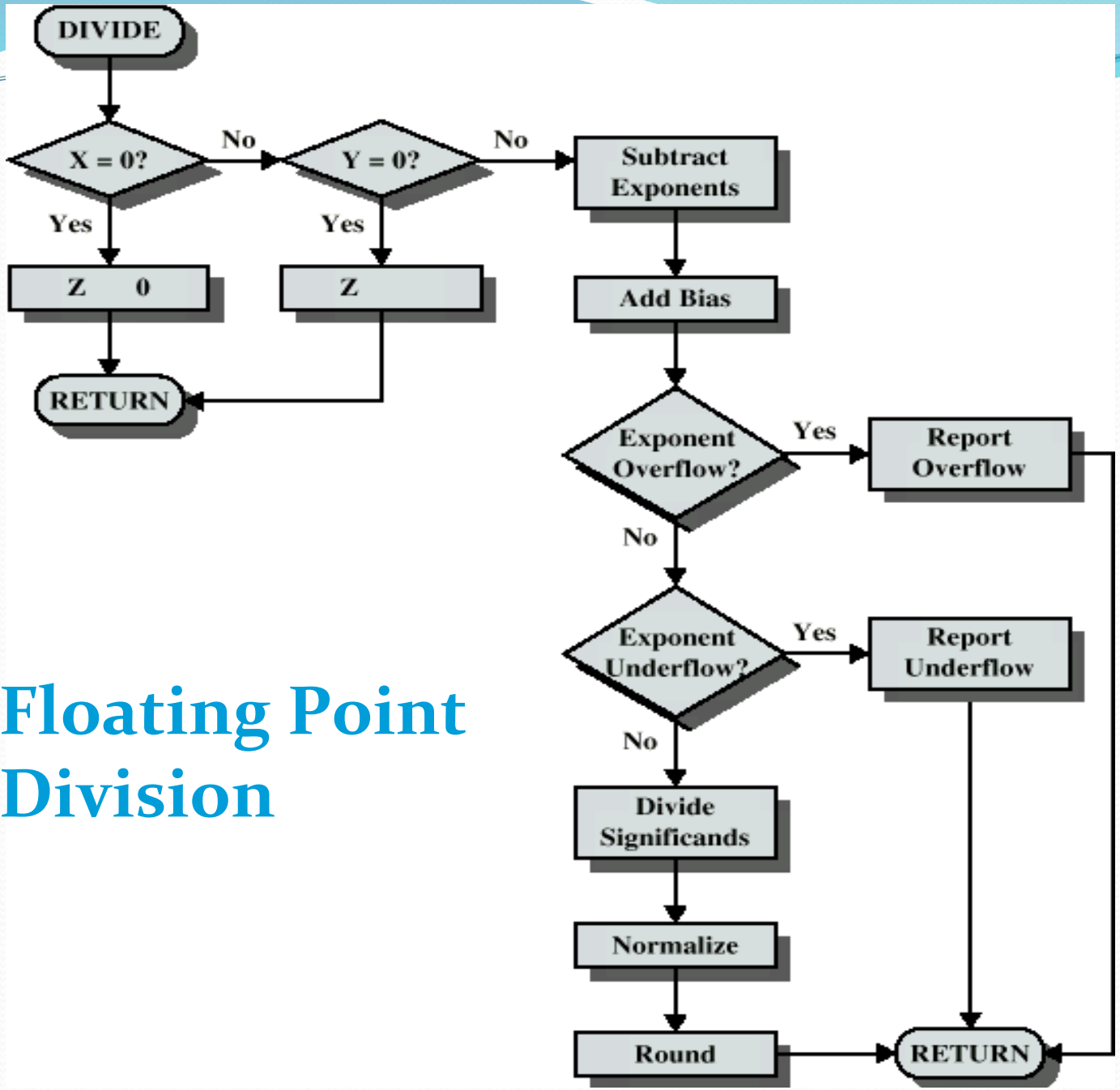
FP Arithmetic x / ÷

- Check for zero
- Add/subtract exponents
- Multiply/divide significands (watch sign)
- Normalize
- Round
- All intermediate results should be in double length storage

Floating Point Multiplication



Floating Point Division



The End