Chapter 6

Arithmetic Unit

Assistant Professor

Er. Shiva Ram Dam

Contents:

6.1 Arithmetic Unit

- The Arithmetic unit is a part of processor that executes arithmetic operations.
- A microprocessor is capable of processing numeric data in one or more formats.

Data Representation

- Data representation should require three provisions:
  - Provision for +ve number(or Unsigned number) representation
  - Provision for –ve number (or signed number) representation.
  - Provision for fractional number representation.

Representation of Binary number and Arithmetic in signed Notation

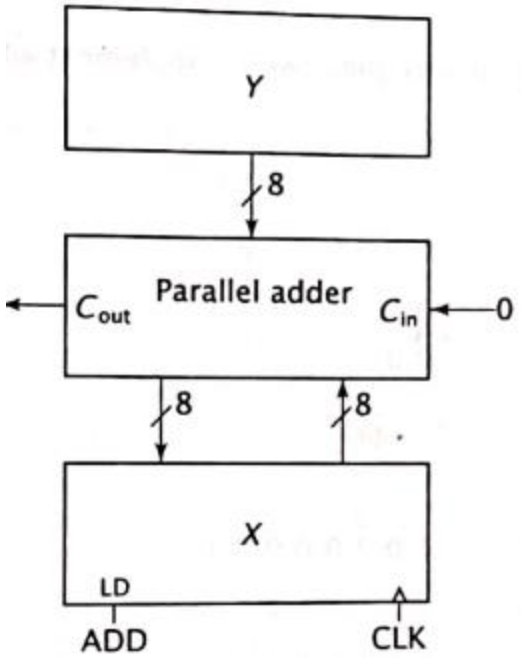
a) Unsigned Number Representation

- In unsigned notation, there is no any separate bit to represent the sign (Positive or Negative) of the number.
- Eg:
  - 1111
  - This is equivalent to 15 in decimal value.

Binary Representation	Unsigned Non-Negative	Unsigned Two's Complement
0000 0000	0	0
0000 0001	1	1
...	...	...
0111 1111	127	127
1000 0000	128	−128
1000 0001	129	−127
...	...	...
1111 1111	255	−1

i) Addition in Unsigned Notation

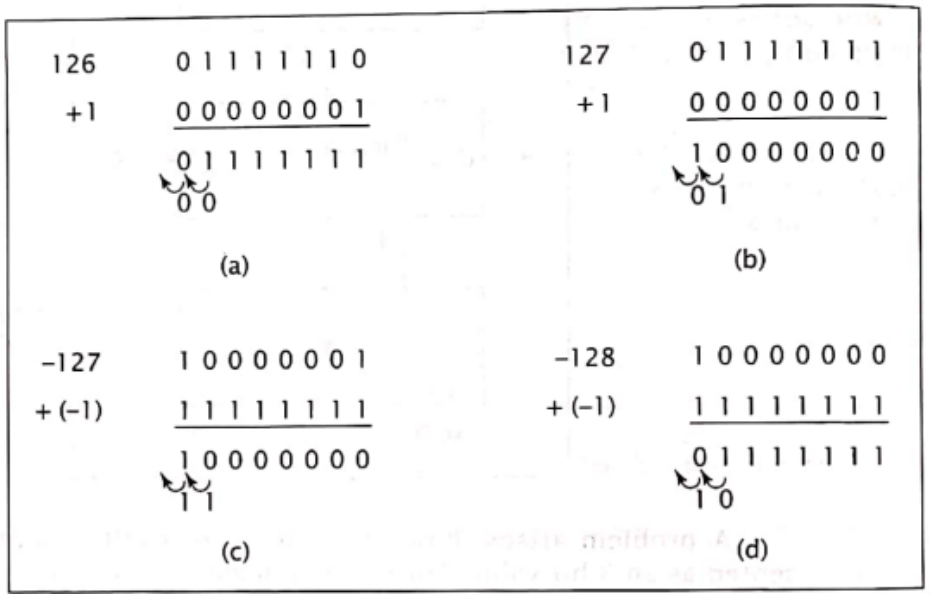
- For both the non-negative and two-s complement notations, addition and subtraction are fairly straightforward.
- Addition is implemented as a straight binary addition.
- It is realized in hardware by using a parallel adder as shown in the figure.
- Here, X and Y are 8-bit registers.
- The circuit performs the micro-operation
- ADD: X← X+Y.



Implementation of the micro-operation $X \leftarrow X + Y$

Overflow during Addition in Unsigned Notation

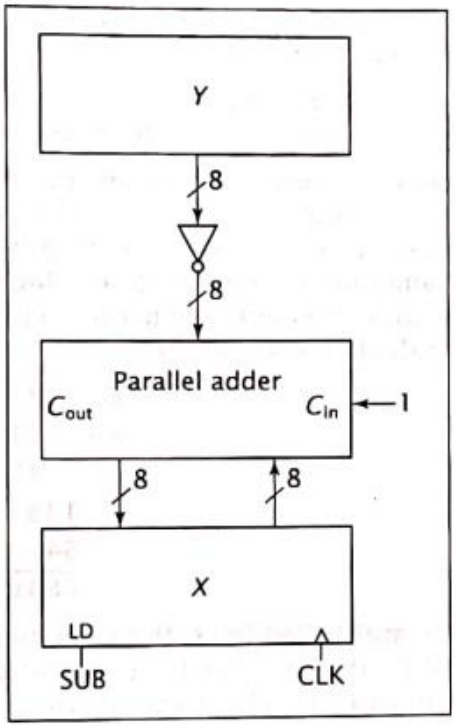
- Overflow only occurs when two numbers with the same sign are added.
- Adding two numbers with different signs always produce a valid result.
- Subtraction can be converted to addition and performed in a similar manner.



Implementation of the micro-operation $X \leftarrow X + Y$

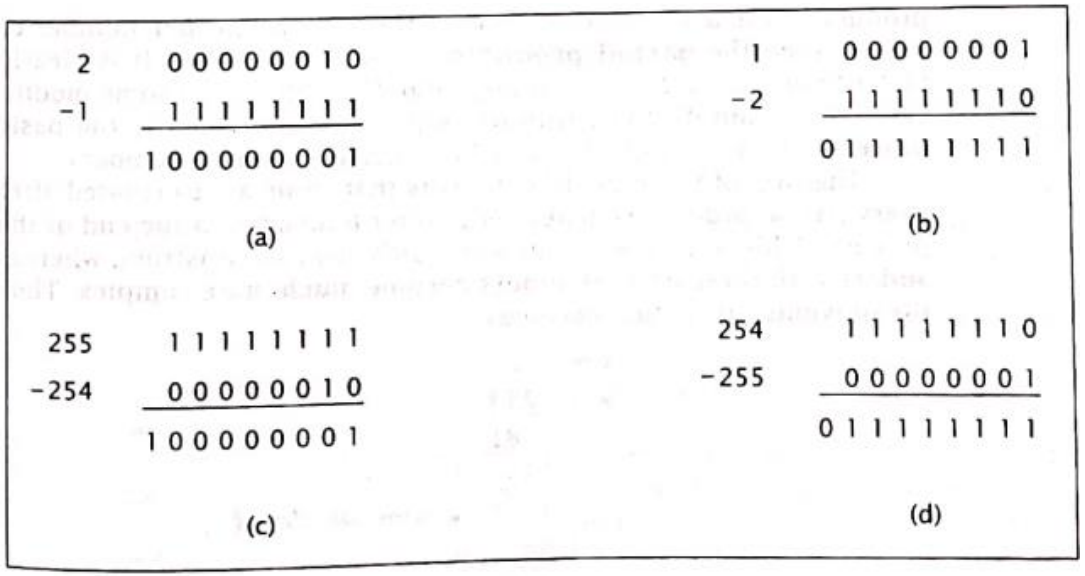
ii) Subtraction in Unsigned Notation

- Subtracting numbers in two's complement notation is also implemented as X-Y = X+(-Y).



Implementation of the micro-operation $X \leftarrow X - Y$

Overflow during Subtraction in Unsigned Notation



Overflow generation in unsigned two's complement subtraction

b) Signed Number Representation

- There are three ways to represent –ve numbers.
  - Signed Magnitude Representation
  - 1's Complement Representation
  - 2's Complement Representation

1. ***Signed Magnitude Representation***
   - Here, an extra bit is required as MSB to represent the sign of a number.
   - If sign bit is 0, the number is +ve.
   - If sign bit is 1, the number is –ve.
   - A n bit number requires n+1 bits.

Eg: +10 = 01010
-10 = 11010

Limitation:

- It represents +0 and -0 by different value.
  Eg: +0 = 00000
  - 0 = 10000
- Signed bit must be considered while performing addition or subtraction.

2) One's Complement Representation

- Here, negative numbers are represented by complementing each bit.
- Eg: +6 = 0110

  - 6 = 1001
- Limitation: It represents +0 and -0 by different value.

- Eg:

  +0 = 0000

  -0 = 1111

3) Two's Complement Representation

- Two's complement representation overcomes the fore-mentioned limitation.
- Two's complement = One's complement + 1
- Eg:
  - +6 = 0110
  - -6 = 2's complement of 0110 = 1001+1 = 1010

b) Fractional Number Representation

i) Fixed Point Representation

The decimal position is either at beginning or at end of the number.

Eg: $2.2 = 0.22 \times 10$

$= 22 \times 10^{-1}$

ii) Floating Point Representation

Here, a second register is separately used to determine the position of decimal.

Addition and Subtraction in signed Notation

6.3 Multiplication Algorithm

- Unsigned Multiplication
  - Shift-Add Algorithm
- Signed Multiplication
  - Booth's Algorithm

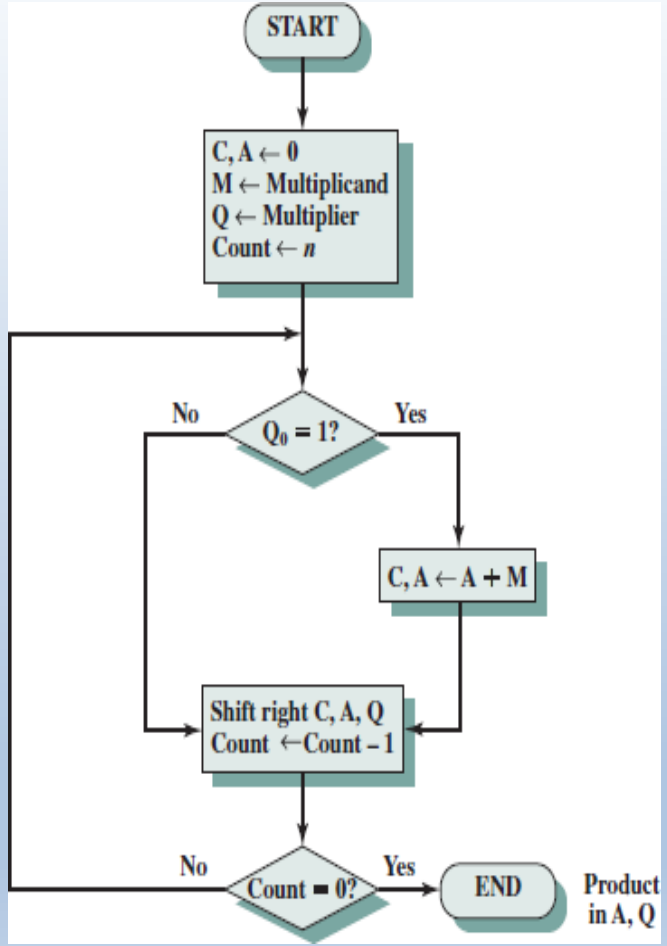
Unsigned Multiplication

Let us consider:

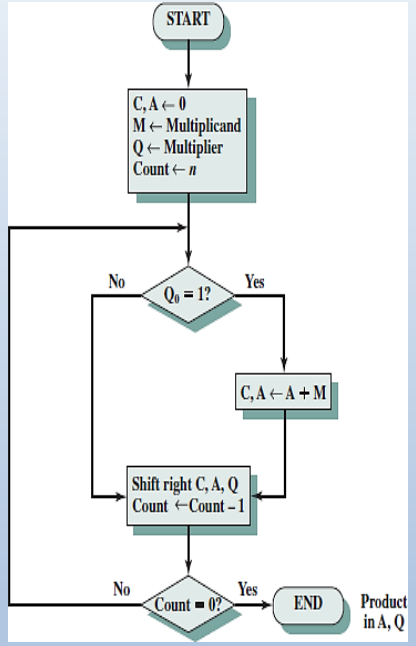
- M=Multiplicand
- Q= Multiplier
- A= Result stored register
- C= 1 bit register for carry

Algorithm:

1. Start
2. Represent operands by binary number.
3. Load M register by multiplicand, Q by multiplier, A and C by 0. And a variable count by N (i.e. no of bits in multiplier).
4. Check Q_0 bit =1.
   If Q_0 bit is 1, then perform a←A+M
5. Right shift C, A, Q by 1 bit.
6. Decrement count.
7. Check if count = 0?
   Yes: go to step 4
   No: go to step 8
8. Store result in A, Q register.
9. Stop.



This is just for reference. Do not follow this



Example: Perform (13_{10}) x (11_{10})

Here: 13= 1101 so, M←1101

11= 1011 Q ←1011

Now,

Iteration	C	A	Q	M	Count	operation
1	0	0000	101**1**	1101	4	Initialization
	0	1101	1011	1101	4	A←A+M
	0	0110	110**1**	1101	3	Right shift C, A, Q
2	1	0011	1101	1101	3	C, A ←A+M
	0	1001	111**0**	1101	2	Right shift C, A, Q
3	0	0100	111**1**	1101	1	Right shift C, A, Q
	1	0001	1111	1101	1	C, A ←A+M
4	0	**1000**	**1111**	1101	0	Right shift C, A, Q / stop

Result store in A, Q = 1000 1111

= 143_{10}

Hence, (13_{10}) x (11_{10}) = (143_{10})

This is just for reference.
Do not follow this

a) Shift Add Algorithm for Unsigned Multiplication

- **Algorithm**

```
U = 0;
FOR i = 1 TO n DO
        {IF Y0 = 1 THEN CU = U + X;
        linear shift right CUV;
        circular shift right Y}
```

- **RTL Code**

1: $U \leftarrow 0,\ i \leftarrow n$

Y_0**2**: $CU \leftarrow U + X$

2: $i \leftarrow i - 1$

3: shr(CUV), cir(Y)

Z'**3**: GOTO **2**

Z **3**: $FINISH \leftarrow 1$

Multiplication of 13 x11

- Initially X=1101 and Y=1011

```
U = 0;
FOR i = 1 TO n DO
        {IF Y0 = 1 THEN CU = U + X;
        linear shift right CUV;
        circular shift right Y}
```

Function	*i*	*C*	*U*	*V*	*Y*	Comments
$U = 0$			0000	0000	1011	
if $Y_0 = 1$	1	0	1101			$Y_0 = 1$, add
shr (*CUV*)		0	0110	1000		
cir (*Y*)					1101	
if $Y_0 = 1$	2	1	0011			$Y_0 = 1$, add
shr (*CUV*)		0	1001	1100		
cir (*Y*)					1110	
if $Y_0 = 1$	3					$Y_0 = 0$
shr (*CUV*)		0	0100	1110		
cir (*Y*)					0111	
if $Y_0 = 1$	4	1	0001			$Y_0 = 1$, add
shr (*CUV*)			1000	1111		
cir (*Y*)					1011	Original value
DONE			1000	1111		Result = 143

Trace of the shift-add algorithm

RTL code to realize Add-shift Algorithm

- The RTL code to realize this algorithm, with these changes, is as follows.
- Note that Z=1 when i=0 and 1,2 and are consecutive states, that is, the algorithm goes from 1 to 2 to 3.

1: $U \leftarrow 0,\ i \leftarrow n$

Y_0**2**: $CU \leftarrow U + X$

2: $i \leftarrow i - 1$

3: shr(CUV), cir(Y)

Z'**3**: GOTO **2**

Z **3**: $FINISH \leftarrow 1$

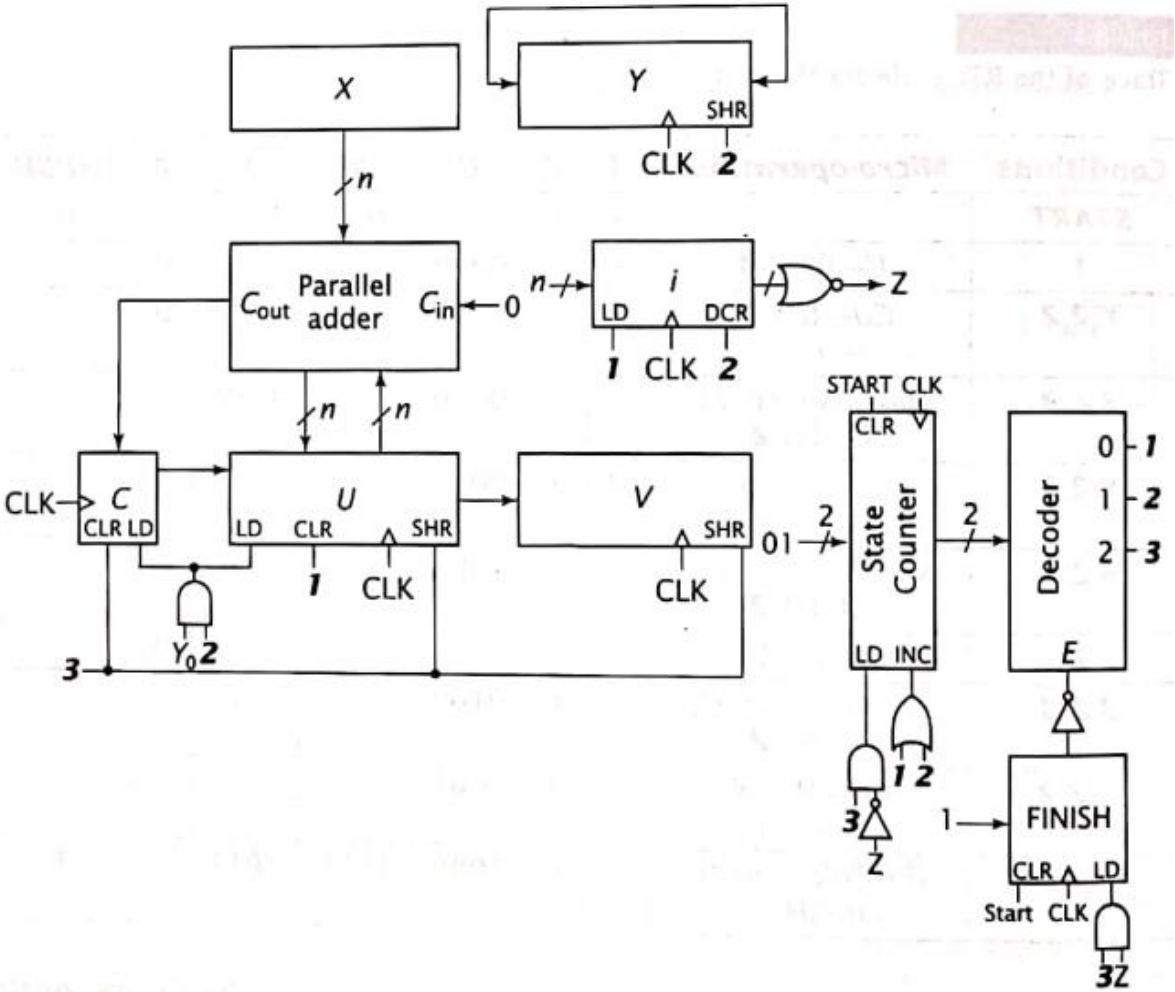
RTL code for Unsigned Multiplication for 13 x11

- Again X=1101 and Y=1011
- It is same as the trace of above table,
- Except it shows the condition met and micro-operations performed during every cycle.

Conditions	Micro-operations	i	C	U	V	Y	Z	FINISH
START		x	x	xxxx	xxxx	1011		0
1	$U \leftarrow 0$, $i \leftarrow 4$	4		0000			0	
$Y_0$2,2	$CU \leftarrow U + X$, $i \leftarrow i - 1$	3	0	1101			0	
3,Z'3	shr(CUV), cir(Y), GOTO 2		0	0110	1xxx	1101		
$Y_0$2,2	$CU \leftarrow U + X$, $i \leftarrow i - 1$	2	1	0011			0	
3,Z'3	shr(CUV), cir(Y), GOTO 2		0	1001	11xx	1110		
2	$i \leftarrow i - 1$	1					0	
3,Z'3	shr(CUV), cir(Y), GOTO 2		0	0100	111x	0111		
$Y_0$2,2	$CU \leftarrow U + X$, $i \leftarrow i - 1$	0	1	0001			1	
3,Z3	shr(CUV), cir(Y), FINISH←1		0	1000	1111	1011		1

Trace of the RTL code for the shift-add algorithm

Hardware Implementation of Shift-Add Multiplication Algorithm



Classwork:

1. Perform: $10_{10} \times 12_{10} = ?$

b) Signed Multiplication

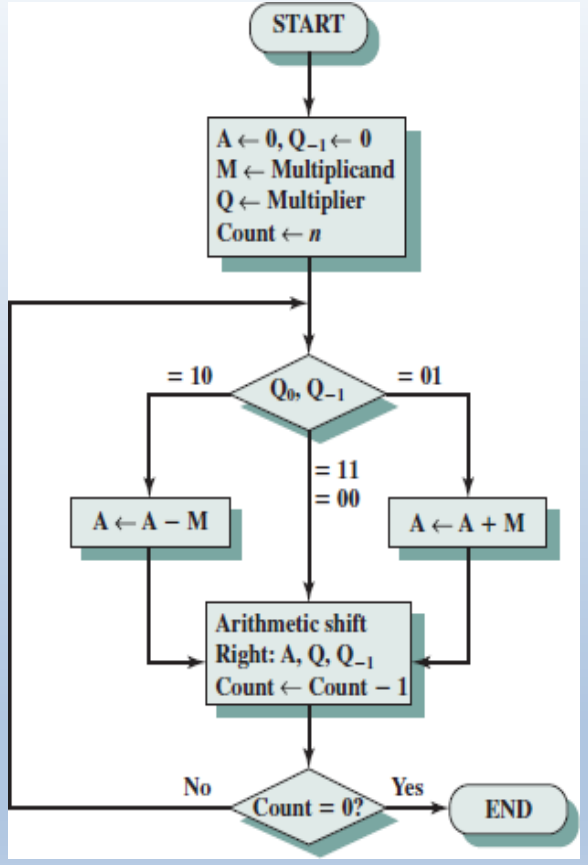
- It is also called Booth's algorithm or Two's complement multiplication.
- Here, negative numbers are represented by 2's complement representation.

Suppose:

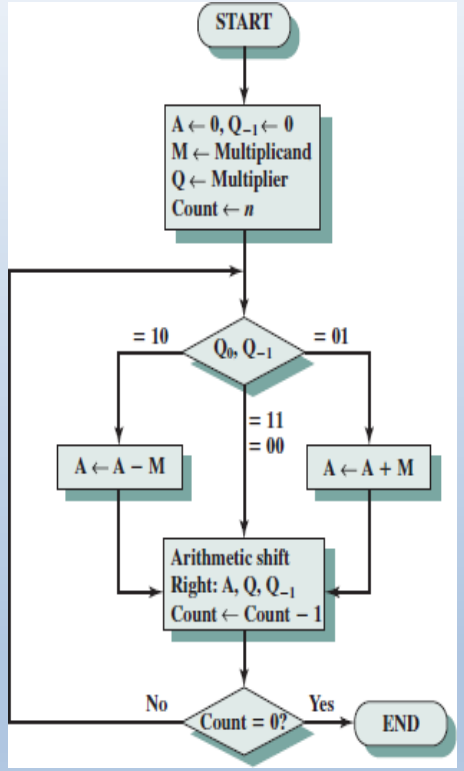
- M= multiplicand
- Q=multiplier
- A,Q= result stored register
- Q_{-1} = 1 bit register which is adjusted after Q_0.

Algorithm:

1. Start
2. Represent operands in 2's complement form.
3. Load M by multiplicand and Q by multiplier.
4. Initialize, A←0
   Q_{-1} ←1
   Count← no. of bits in multiplier.
5. Check, Is Q_0Q_{-1} = 01or 10?
   If Q_0Q_{-1} = 01 then perform A← A+M and go to step 5.
   Elseif Q_0Q_{-1} = 10 then perform A← A-M and go to step 5
   Else do nothing and go to step 5.
6. Perform Arithmetic Right Shift operation on A, Q, Q_{-1} by 1 bit.
7. Decrement count by 1 (i.e. count = count -1).
8. Check if count = 0?
   Yes: go to step 8.
   No: go to step 4.
9. Store result in A, Q.
10. Stop



This is just for reference. Do not follow this



Example: Perform $(-7_{10}) \times (-3_{10})$

Here, In 2's complement: $-7 = 1001$ so, M←1001

$-3 = 1101$ Q ←1101

Iteration	A	Q	Q_{-1}	M	Count	operation
1	0000	1101	0	1001	4	Initialization
	0111	1101	0	1001	4	A←A-M
	0011	1110	1	1001	3	Right shift, count--
2	1100	0111	1	1001	3	A←A-M
	1110	0111	0	1001	2	Right shift, count--
3	0101	0111	0	1001	2	A←A-M
4	0010	1011	1	1001	1	Right shift, count--
5	0001	0101	1	1001	0	Right shift, count-- / stop

The result is stored in A, Q = 0001 0101

Since the sign bit is 0, the result is +ve. So, do not need to re-complement.

$0001\ 0101_2 = 21_{10}$

Hence, $(-7_{10}) \times (-3_{10}) = 21_{10}$

This is just for reference.
Do not follow this

b) Booth's Algorithm for Signed Multiplication

- **Algorithm**

```
U = 0; Y_-1 = 0;
FOR i = 1 TO n DO
{IF start of a string of 1's in Y THEN U = U − X (= U + X' + 1);
 IF end of a string of 1's in Y THEN U = U + X;
 arithmetic shift right UV;
 circular shift right Y AND copy Y_0 to Y_-1}
```

- **RTL Code**

1: $U \leftarrow 0,\ Y_{-1} \leftarrow 0,\ i \leftarrow n$

$Y_0 Y_{-1}' 2$: $U \leftarrow U + X' + 1$

$Y_0' Y_{-1} 2$: $U \leftarrow U + X$

2: $i \leftarrow i - 1$

3: $\text{ashr}(UV),\ \text{cir}(Y),\ Y_{-1} \leftarrow Y_0$

$Z'3$: GOTO 2

$Z3$: $FINISH \leftarrow 1$

Multiplication of -3 x-5 using Booth's Algorithm

- Initially X=1101 and Y=1011
- U=0000
- V=0000
- Y_{-1}= 0

```
U = 0; Y_-1 = 0;
FOR i = 1 TO n DO
{IF start of a string of 1's in Y THEN U = U − X (= U + X' + 1);
IF end of a string of 1's in Y THEN U = U + X;
arithmetic shift right UV;
circular shift right Y AND copy Y_0 to Y_-1}
```

Function	*i*	*U*	*V*	*Y*	Y_{-1}	Comments
$U = 0, Y_{-1} = 0$		0000	0000	1011	0	
if start of string	1	0011				Start of string
if end of string						
ashr(*UV*)		0001	1000			
cir(*Y*), $Y_{-1} \leftarrow Y_0$				1101	1	
if start of string	2					Still within string
if end of string						
ashr(*UV*)		0000	1100			
cir(*Y*), $Y_{-1} \leftarrow Y_0$				1110	1	
if start of string	3					
if end of string		1101				End of string
ashr(*UV*)		1110	1110			
cir(*Y*), $Y_{-1} \leftarrow Y_0$				0111	0	
if start of string	4	0001				Start of string
if end of string						
ashr(*UV*)		0000	1111			
cir(*Y*), $Y_{-1} \leftarrow Y_0$				1011	1	Original value
DONE		0000	1111			Result = +15

Trace of Booth's algorithm

Assignment:

- Multiply 3 x -5 using Booth's Algorithm.

RTL code to realize Booth's Algorithm

- The RTL code to realize this algorithm, with these changes, is as follows.
- U,V, X and Y are n-bit values and Y-1 is a 1-bit value
- i counts down from n to 0.

$\mathbf{1}$: $U \leftarrow 0,\ Y_{-1} \leftarrow 0,\ i \leftarrow n$

$Y_0 Y_{-1}' \mathbf{2}$: $U \leftarrow U + X' + 1$

$Y_0' Y_{-1} \mathbf{2}$: $U \leftarrow U + X$

$\mathbf{2}$: $i \leftarrow i - 1$

$\mathbf{3}$: $\text{ashr}(UV),\ \text{cir}(Y),\ Y_{-1} \leftarrow Y_0$

$Z' \mathbf{3}$: GOTO $\mathbf{2}$

$Z \mathbf{3}$: $FINISH \leftarrow 1$

RTL code for Booth's Algorithm for -3 x -5

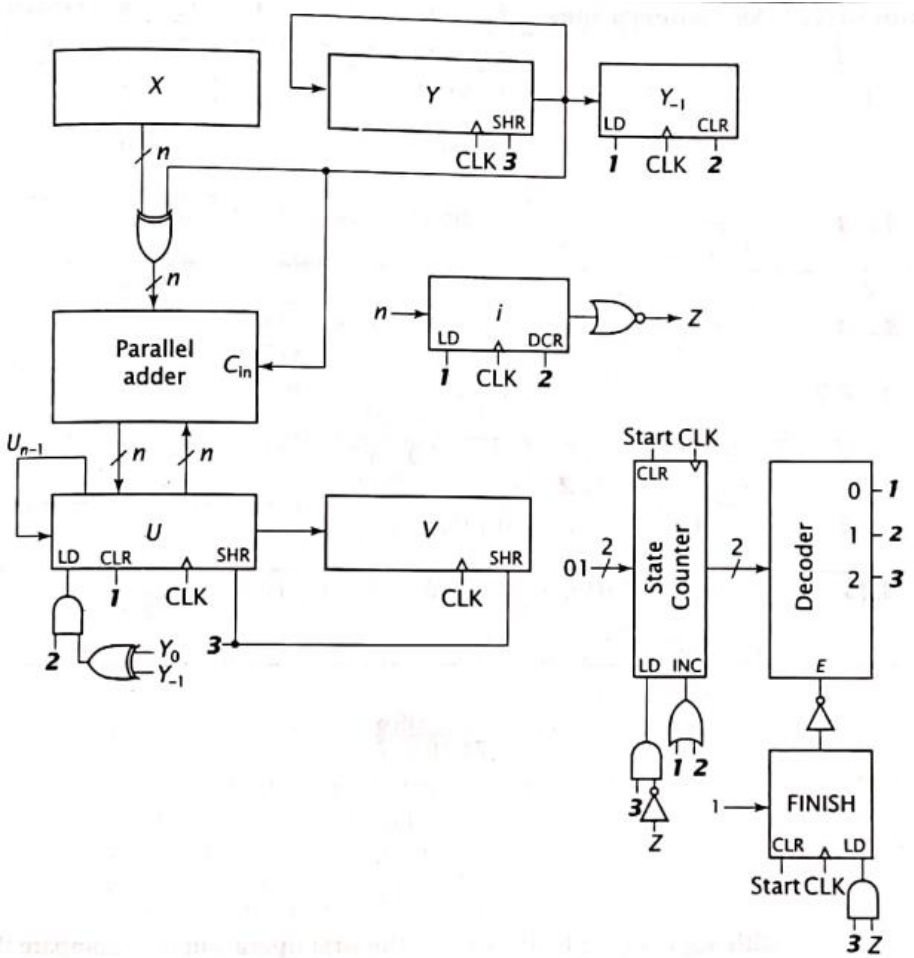
```
1:     U←0, Y_-1←0, i←n
Y0Y-1'2:  U←U + X' + 1
Y0'Y-1 2:  U←U + X
2:     i←i − 1
3:     ashr(UV), cir(Y), Y_-1←Y0
Z'3:   GOTO 2
Z3:    FINISH←1
```

Conditions	Micro-operations	i	U	V	Y	Y_{-1}	Z	FINISH
START		x	xxxx	xxxx	1011	x		0
1	$U\leftarrow 0$, $Y_{-1}\leftarrow 0$, $i\leftarrow 4$	4	0000			0	0	
Y_0Y_{-1}'2,2	$U\leftarrow U + X' + 1$, $i\leftarrow i - 1$	3	0011				0	
3,Z'3	ashr(UV), cir(Y), $Y_{-1}\leftarrow Y_0$, GOTO 2		0001	1xxx	1101	1		
2	$i\leftarrow i - 1$	2					0	
3,Z'3	ashr(UV), cir(Y), $Y_{-1}\leftarrow Y_0$, GOTO 2		0000	11xx	1110	1		
$Y_0'Y_{-1}$2,2	$U\leftarrow U + X$, $i\leftarrow i - 1$	1	1101				0	
3,Z'3	ashr(UV), cir(Y), $Y_{-1}\leftarrow Y_0$, GOTO 2		1110	111x	0111	0		
Y_0Y_{-1}'2,2	$U\leftarrow U + X' + 1$, $i\leftarrow i - 1$	0	0001				1	
3,Z3	ashr(UV), cir(Y), $Y_{-1}\leftarrow Y_0$, FINISH←1		0000	1111	1011			1

Trace of the RTL code for Booth's algorithm

Note:
If you get 1 at MSB of U, then your result is negative. So, you need to re-complement it to verify your answer.

Hardware Implementation of Booth's Algorithm



Classwork:

1. Perform: $7_{10} \times -3_{10} = ?$
2. Perform: $9_{10} \times -3_{10} = ?$

[Hint: Take 5 bits for both X and Y]

6.4 Division Algorithm

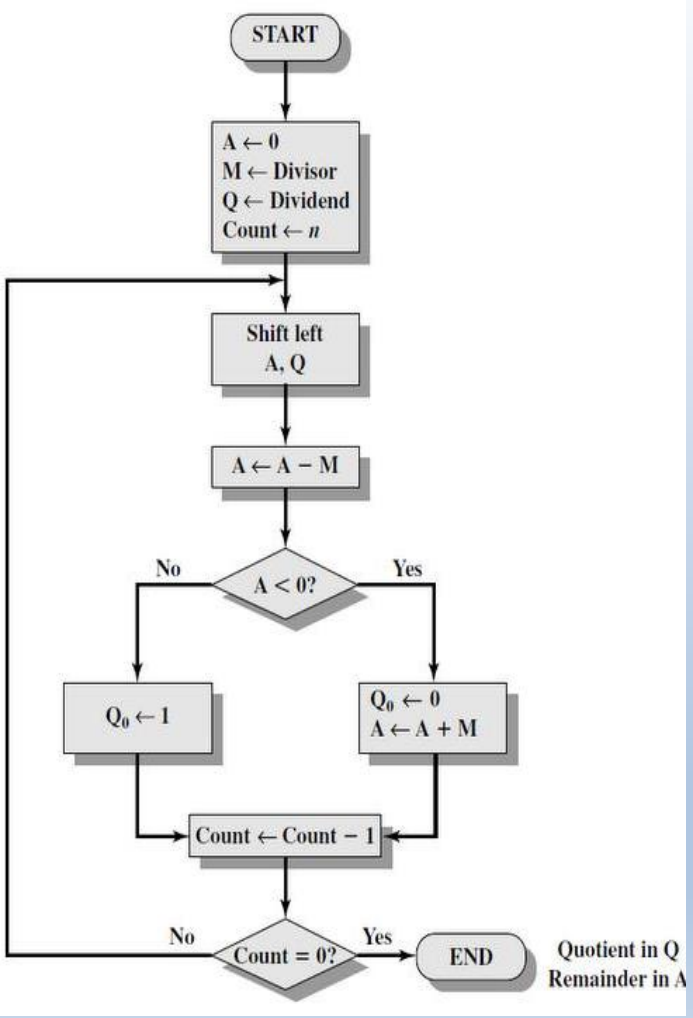
- Division in Unsigned notation
  - Shift-Subtract Division Algorithm

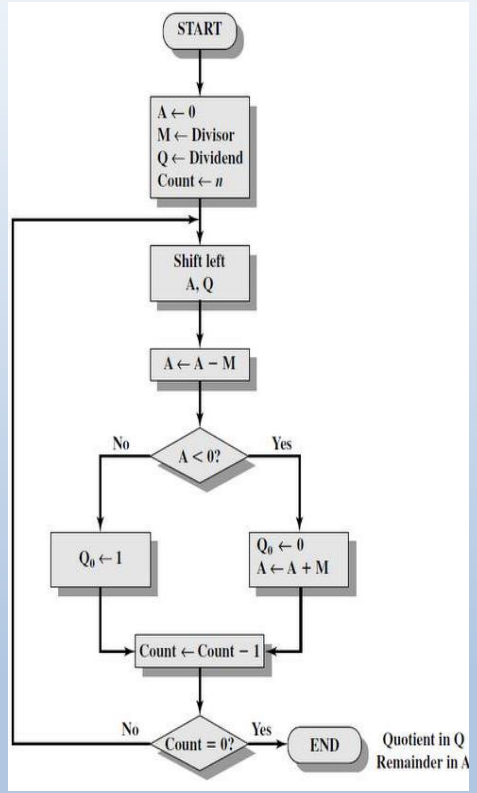
a) Unsigned Division

Algorithm:

1. Start
2. Store Q←Dividend
   M←Divisor
   Register← 0
   Count= no. of bits in divisor
3. Left shift the bits of A, Q by 1 bit position.
4. Perform A← A + M
5. Check the sign bit of magnitude in A. (i.e. if A is positive or negative). Is A<0?
   Yes: Perform Q_0 ←0. Restore the value of A (A←A+M) and go to step 6.
   No: Perform Q_0 ←1
6. Decrement counter (i.e. count = count-1)
7. Is count=0?
   Yes: go to step 8
   No: go to step 3
8. Read quotient in register Q and remainder in register A
9. Stop.

This is just for reference.
Do not follow this





Problem: Perform $7_{10} \div 3_{10}$

Here: 7 = 0111 Q← Dividend

3 = 0011 M← Divisor

Now,

Iteration	A	Q	M	Count	Operation
1	0000	0111	0011	4	Initialization
	0000	1110	0011	4	Left shift A, Q
	1101	1110	0011	4	A←A-M
	0000	1110	0011	3	Q_0 ←0, Restore A, Decrease count
2	0001	1100	0011	3	Left shift A, Q
	1110	1100	0011	3	A←A-M
	0001	1100	0011	2	Q_0 ←0, Restore A, Decrease count
3	0011	1000	0011	2	Left shift A, Q
	0000	1000	0011	2	A←A-M
	0000	1001	0011	1	Q_0 ←1, count--
4	0001	0010	0011	1	Left shift A, Q
	1110	0010	0011	1	A←A-M
	0001	0010	0011	0	Restore A, count-- , stop

Here, Result is stored in Q and A.

Quotient → stored in Q = 0010 = 2_{10}

Remainder→ stored in A= 0001 = 1_{10}

Hence, $7_{10} \div 3_{10}$ gives quotient 2 and remainder 1.

This is just for reference.
Do not follow this

Shift-Subtract Division

- **Algorithm**

```
IF U ≥ X THEN exit with overflow;
Y = 0; C = 0;
FOR i = 1 TO n DO
        {linear shift left CUV;
        linear shift left Y;
        IF CU ≥ X THEN {Y0 = 1, U = CU - X}}
```

- **RTL code**

```
G1:      FINISH←1, OVERFLOW←1
2:       Y←0, C←0, OVERFLOW←0, i←n
3:       shl(CUV), shl(Y), i←i − 1
(C + G)4: Y0←1, U←U + X′ + 1
Z′4:     GOTO 3
Z 4:     FINISH←1
```

Division of 147/13 using Shift-Subtract Division Algorithm

- Initially U=1001, V=0011
- X=1101 and
- N=4

```
IF U ≥ X THEN exit with overflow;
Y = 0; C = 0;
FOR i = 1 TO n DO
        {linear shift left CUV;
        linear shift left Y;
        IF CU ≥ X THEN {Y0 = 1, U = CU - X}}
```

Function	*i*	*C*	*U*	*V*	*Y*	Comments
initial		0	1001	0011	0000	
if $U \geq X$						$U < X$, no exit
shl(*CUV*)	1	1	0010	0110		
shl(*Y*)					0000	
if $CU \geq X$			0101		0001	1 0010 ≥ 1101
shl(*CUV*)	2	0	1010	1100		
shl(*Y*)					0010	
if $CU \geq X$						$CU < X$
shl(*CUV*)	3	1	0101	1000		
shl(*Y*)					0100	
if $CU \geq X$			1000		0101	1 0101 ≥ 1101
shl(*CUV*)	4	1	0001	0000		
shl(*Y*)					1010	
if $CU \geq X$			0100		1011	1 0001 ≥ 1101

RTL code to realize Shift-subtract Division Algorithm

- In the following RTL code for this algorithm, X, U, V and Y are n-bit values and C and OVERLOW are 1-bit values.
- As before Z=1 when i= 0.
- For this algorithm, G=1 if U>=X.
- FINISH is the same as on shift- add multiplication algorithm

$G\,\mathbf{1}$: $FINISH \leftarrow 1,\ OVERFLOW \leftarrow 1$

$\mathbf{2}$: $Y \leftarrow 0,\ C \leftarrow 0,\ OVERFLOW \leftarrow 0,\ i \leftarrow n$

$\mathbf{3}$: $\text{shl}(CUV),\ \text{shl}(Y),\ i \leftarrow i - 1$

$(C + G)\,\mathbf{4}$: $Y_0 \leftarrow 1,\ U \leftarrow U + X' + 1$

$Z'\,\mathbf{4}$: GOTO $\mathbf{3}$

$Z\,\mathbf{4}$: $FINISH \leftarrow 1$

$CU \geq X$

RTL code to realize Shift-subtract Division Algorithm

G1: $FINISH \leftarrow 1$, $OVERFLOW \leftarrow 1$
2: $Y \leftarrow 0$, $C \leftarrow 0$, $OVERFLOW \leftarrow 0$, $i \leftarrow n$
3: $shl(CUV)$, $shl(Y)$, $i \leftarrow i - 1$
(C + G)4: $Y_0 \leftarrow 1$, $U \leftarrow U + X' + 1$
Z'4: GOTO **3**
Z 4: $FINISH \leftarrow 1$

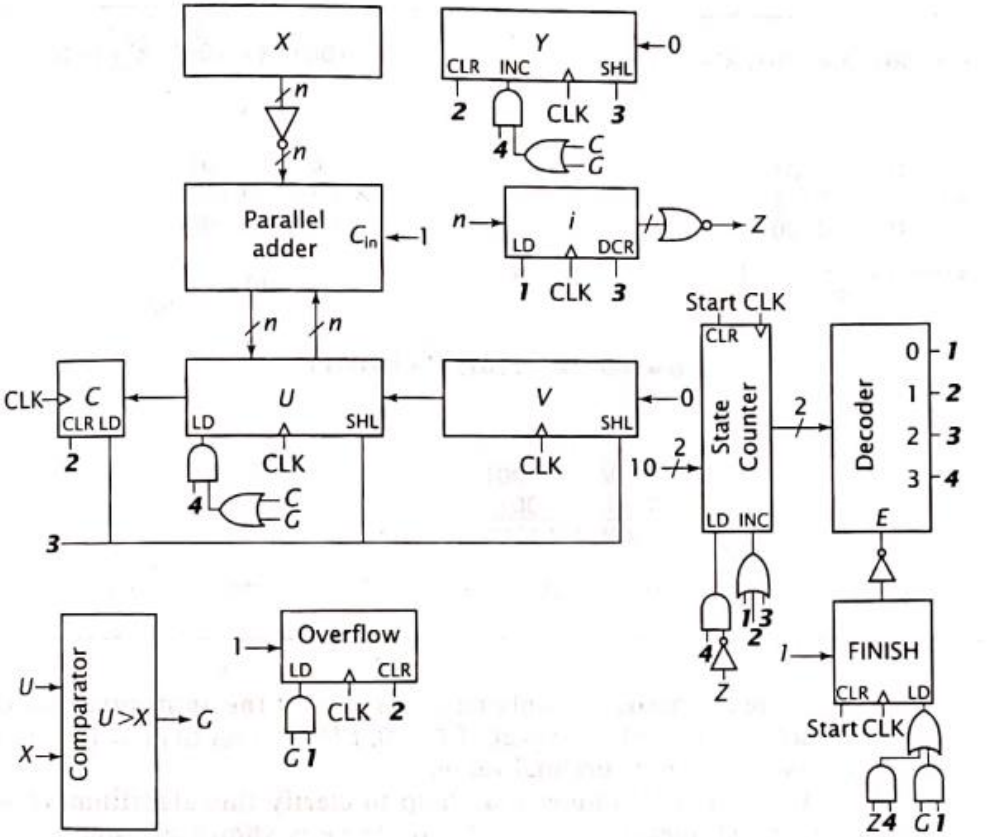
Conditions	Micro-operations	i	C	U	V	Y	Z	FINISH
START		x	x	1001	0011	xxxx		0
1	NONE							
2	$Y \leftarrow 0$, $C \leftarrow 0$, $OVERFLOW \leftarrow 0$, $i \leftarrow 4$	4	0			0000	0	
3	$shl(CUV)$, $shl(Y)$, $i \leftarrow i - 1$	3	1	0010	0110	0000	0	
(C + G)4,Z'4	$Y_0 \leftarrow 1$, $U \leftarrow U + X' + 1$, GOTO **3**			0101		0001		
3	$shl(CUV)$, $shl(Y)$, $i \leftarrow i - 1$	2	0	1010	1100	0010	0	
Z'4	GOTO **3**							
3	$shl(CUV)$, $shl(Y)$, $i \leftarrow i - 1$	1	1	0101	1000	0100	0	
(C + G)4,Z'4	$Y_0 \leftarrow 1$, $U \leftarrow U + X' + 1$, GOTO **3**			1000		0101		
3	$shl(CUV)$, $shl(Y)$, $i \leftarrow i - 1$	0	1	0001	0000	1010	1	
(C + G)4,Z4	$Y_0 \leftarrow 1$, $U \leftarrow U + X' + 1$, $FINISH \leftarrow 1$		0100		1011			1

Remainder

Quotient

$CU \geq X$

Hardware Implementation of Shift-Subtract Division Algorithm



Classwork:

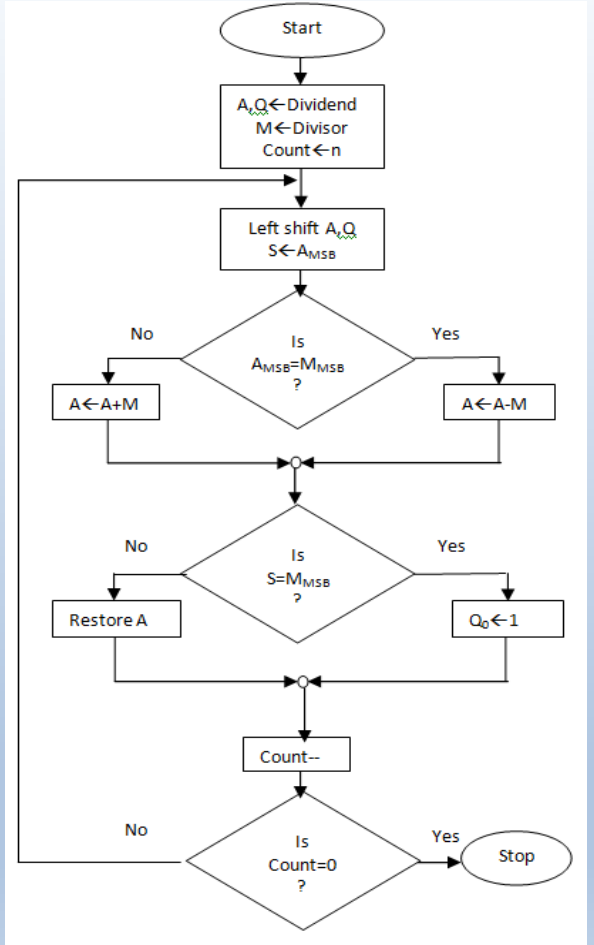
a) Perform 14 ÷5

b) Signed Division

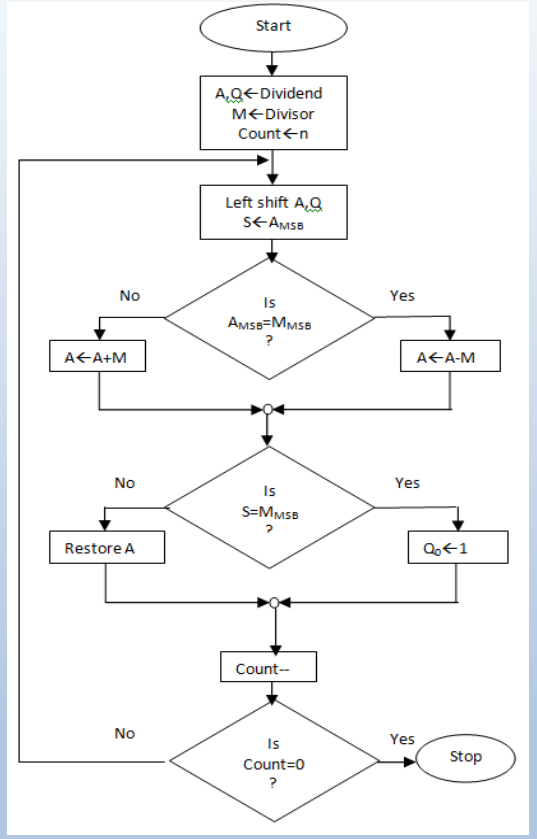
Rules:

1. Remainder and dividend must have the same sign.
2. Negate the obtained quotient if the signs of divisor and dividend disagree (i.e. positive an negative)
3. Examples:

case	Q ÷ M	Quotient	Remainder	Negate quotient?
1	7 ÷ 3	2	1	No
	0111 ÷ 0011	0010	0001	Q=0010 , R=0001
2	7 ÷ -3	-2	1	Yes
	0111 ÷ 1101	0010	0001	Q= 1110 , R=0001
3	-7 ÷ 3	-2	-1	Yes,
	1001 ÷ 0011	0010	1111	Q=1110 , R=1111
4	-7 ÷ -3	2	-1	No
	1001 ÷ 1101	0010	1111	Q=0010 , R=1111



This is just for reference.
Do not follow this



Perform: 7 ÷ -3

Here, 7= 0000 0111 →A, Q

-3 = 1101 →M

Iteration	S	A	Q	M	Count	Operation
1		0000	0111	1101	4	Initialization
	0	0000	1110	1101	4	Left shift, S← A_{MSB}
		1101	1110	1101	4	A←A+M
		0000	1110	1101	3	Restore A, S≠A_{MSB}, count--
2	0	0001	1100	1101	3	Left shift
		1110	1100	1101	3	A←A+M
		0001	1100	1101	2	Restore A, S≠A_{MSB}, count--
3	0	0011	1000	1101	2	Left shift
		0000	1000	1101	2	A←A-M
		0000	1001	1101	1	Q0←1, count--
4	0	0001	0010	1101	1	Left shift
		1110	0010	1101	1	A←A+M
		0001	**0010**	1101	0	Restore A, count--, stop

Here, dividend and divisor disagreed the sign. So, we need to complement the quotient.

Quotient (Q) = $\overline{0010}$ = 1110 = -2_{10}

Remainder (A) = 0001 = 1_{10}

This is just for reference.
Do not follow this

6.7 Binary Coded Decimal

- It is the most popular format to represent decimal data
- In BCD, each 4 bits represent one decimal digit.

Decimal	Binary	BCD
0	0000 0000	0000 0000
1	0000 0001	0000 0001
2	0000 0010	0000 0010
3	0000 0011	0000 0011
4	0000 0100	0000 0100
5	0000 0101	0000 0101
6	0000 0110	0000 0110
7	0000 0111	0000 0111
8	0000 1000	0000 1000
9	0000 1001	0000 1001
10	0000 1010	0001 0000
11	0000 1011	0001 0001
12	0000 1100	0001 0010
13	0000 1101	0001 0011
14	0000 1110	0001 0100
15	0000 1111	0001 0101
16	0001 0000	0001 0110
17	0001 0001	0001 0111
18	0001 0010	0001 1000
19	0001 0011	0001 1001
20	0001 0100	0010 0000

BCD Addition

- Like other number system in BCD arithmetical operation may be required.
- BCD is a numerical code which has several rules for addition.
- The rules are given below in three steps with an example to make the idea of **BCD Addition** clear.
  1. At first the given number are to be added using the rule of binary. For example,

Case 1:

$$\begin{array}{r} 1010 \\ +\ 0101 \\ \hline 1111 \end{array}$$

Case 2:

$$\begin{array}{r} 0001 \\ +\ 0101 \\ \hline 0110 \end{array}$$

BCD Addition

- In second step we have to judge the result of addition. Here two cases are shown to describe the rules of **BCD Addition**. In case 1 the result of addition of two binary number is greater than 9, which is not valid for BCD number. But the result of addition in case 2 is less than 9, which is valid for BCD numbers.

- If the four bit result of addition is greater than 9 and if a carry bit is present in the result then it is invalid and we have to add 6 whose binary equivalent is $(0110)_2$ to the result of addition. Then the resultant that we would get will be a valid binary coded number. In case 1 the result was $(1111)_2$, which is greater than 9 so we have to add 6 or $(0110)_2$ to it.

$$(1111)_2 + (0110)_2 = 0001\ 0101 = 15$$

Case 1:

$$\begin{array}{r} 1010 \\ +\ 0101 \\ \hline 1111 \end{array}$$

Case 2:

$$\begin{array}{r} 0001 \\ +\ 0101 \\ \hline 0110 \end{array}$$

BCD Addition

- As you can see the result is valid in BCD.
  But in case 2 the result was already valid BCD, so there is no need to add 6. This is how BCD Addition could be.
- Now a question may arrive that why 6 is being added to the addition result in case BCD Addition instead of any other numbers. It is done to skip the six invalid states of binary coded decimal i.e from 10 to 15 and again return to the BCD codes.

BCD Addition

Example:1

Let, 0101 is added with 0110.

$$\begin{array}{rl} & 0101 \\ + & 0110 \\ \hline & 1011 \rightarrow \text{Invalid BCD number} \\ + & 0110 \rightarrow \text{Add 6} \\ \hline 0001 & 0001 \rightarrow \text{Valid BCD number} \end{array}$$

Check your self.

$$(0101)_2 \rightarrow (5)_{10} \; and \; (0110)_2 \rightarrow (6)_{10}(5)_{10} + (6)_{10} = (11)_{10}$$

BCD Addition

Example:2

Now let 0001 0011 is added to 0010 0110.

$$\begin{array}{r} 0001\ 0001 \\ +\ 0010\ 0110 \\ \hline 0011\ 0111 \end{array} \longrightarrow \text{Valid BCD number}$$

$$(0001\ 0001)_{BCD} \rightarrow (11)_{10},\ (0010\ 0110)_{BCD} \rightarrow (26)_{10}\ and\ (0011\ 0111)_{BCD}$$
$$\rightarrow (37)_{10}(11)_{10} + (26)_{10} = (37)_{10}$$

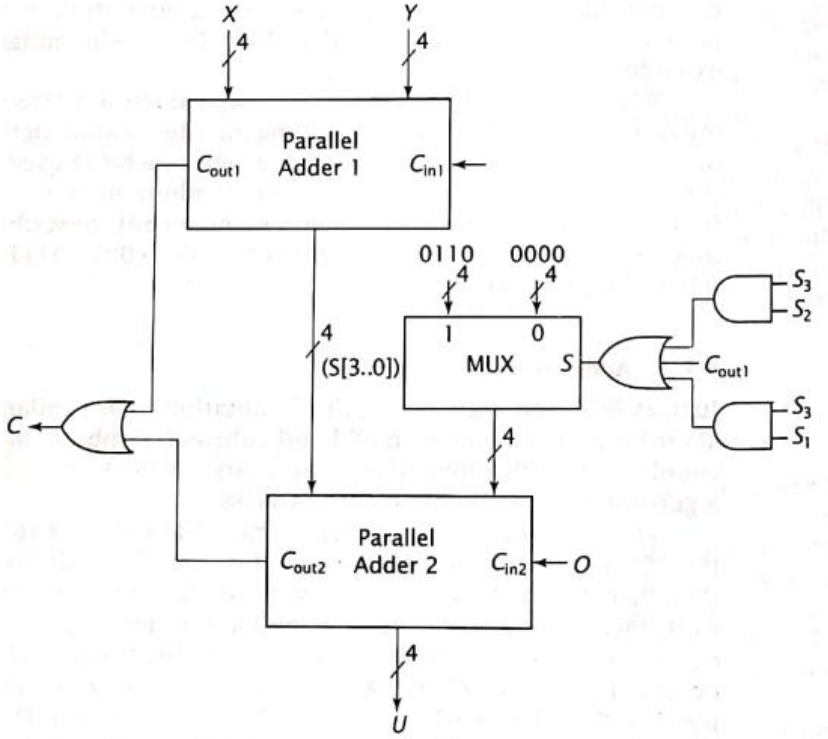
So no need to add 6 as because both

$$(0011)_2 = (3)_{10}\ and\ (0111)_2 = (7)_{10}$$

are less than $(9)_{10}$. This is the process of BCD Addition.

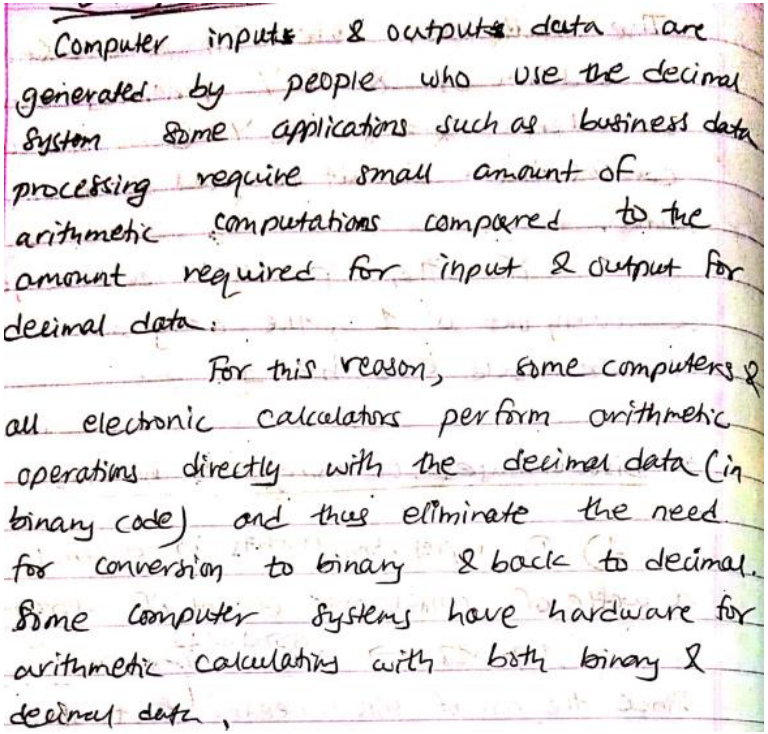
BCD Adder Hardware Implementation

... In this figure, notice that the two digits, X and Y, are first added together. Then this result is added to either 0 (0000) or 6 (0110) to produce the correct BCD sum. If the result, $S_3S_2S_1S_0$, is not a valid BCD digit, then either $S_3 \wedge S_2 = 1$ or $S_3 \wedge S_1 = 1$. (We can easily verify that the six invalid bit patterns 1010 through 1111 meet one or both of these conditions, and that the 10 valid BCD values 0000 through 1001 do not.) In this case, the multiplexer control bit is set to 1, which causes 6 to be added to the result, correcting its decimal representation. The multiplexer select bit is also set to 1 if the addition of X and Y generates a carry out, as when $X = 8$ and $Y = 9$. This also causes the circuit to add 6 to the original sum, again producing the correct value. If either parallel adder generates a carry out of 1, the carry out of the BCD adder should also be 1. Just as with binary adders, multiple copies of this hardware can be cascaded by connecting their carry out/carry in signals to form a multidigit BCD adder.



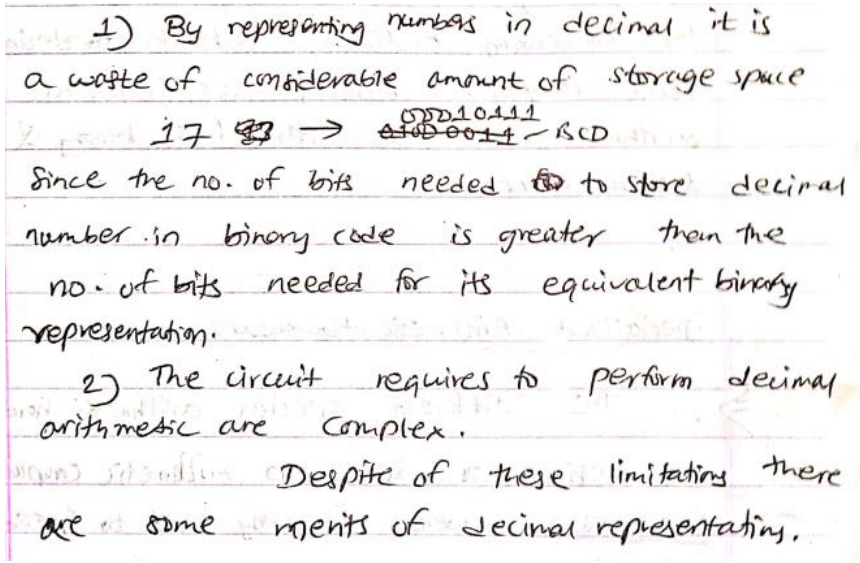
Advantages and Disadvantages of using BCD notations

- Advantages:


Computer inputs & outputs data are generated by people who use the decimal system. Some applications such as business data processing require small amount of arithmetic computations compared to the amount required for input & output for decimal data.

For this reason, some computers & all electronic calculators perform arithmetic operations directly with the decimal data (in binary code) and thus eliminate the need for conversion to binary & back to decimal. Some computer systems have hardware for arithmetic calculations with both binary & decimal data.

- Disadvantages:


1) By representing numbers in decimal it is a waste of considerable amount of storage space

17 → 0001 0111 – BCD

Since the no. of bits needed to store decimal number in binary code is greater than the no. of bits needed for its equivalent binary representation.

2) The circuit requires to perform decimal arithmetic are complex.

Despite of these limitations there are some merits of decimal representation.

6.8 Specialized Arithmetic Hardware

- The different special arithmetic hardware are designed to speed up arithmetic computation.
- Coprocessors were formerly used to speed up computations.
- The coprocessor chip had specialized hardware that performed arithmetic calculations much more quickly than microprocessor.
- Th coprocessor monitored instructions on system databus.
- When the microprocessor fetched an instruction that the coprocessor could execute, the coprocessor sent a signal to microprocessor indicating that it would perform computation.
- The coprocessor then calculates the result and send it to microprocessor.
- If the coprocessor was not present, the microprocessor executes the instruction in slow way.

Specialized Arithmetic Hardware

Three techniques to improve speed of arithmetic calculation:

1. Pipelining
2. Look up Tables
3. Wallace Tree multipliers

1. Pipelining

- An arithmetic pipeline is similar to an assembly line in a factory.
- Data enters a stage of the pipeline, which performs some arithmetic operation on the data.
- The results are then passes to the next stage, which performs its operations, and so on until the final computation has been performed.
- Each stage performs only its specific function.
- It improves performance by overlapping computations; each stage can operate on different data simultaneously.

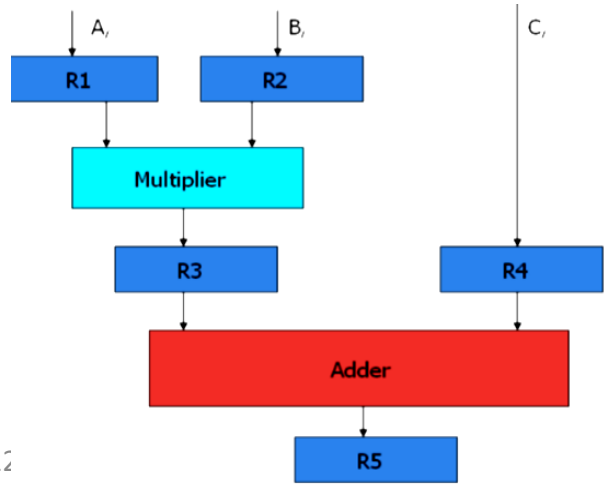
Pipelining example

- Pipelining is a technique of decomposing a sequential process into sub-operations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments.
- The overlapping of computation is made possible by associating a register with each segment in the pipeline.
- The registers provide isolation between each segment so that each can operate on distinct data simultaneously.
- A clock is applied to all registers after enough time has elapsed to perform all segment activity.

Pipelining example

- Consider the below example: To perform the combined multiply and add operations with a stream of numbers $A_i * B_i + C_i$ for i = 1, 2, 3, …, 7
- Each sub-operation is to be implemented in a segment within a pipeline.

$R1 \leftarrow A_i$, $R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2$, $R4 \leftarrow C_i$	Multiply and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product



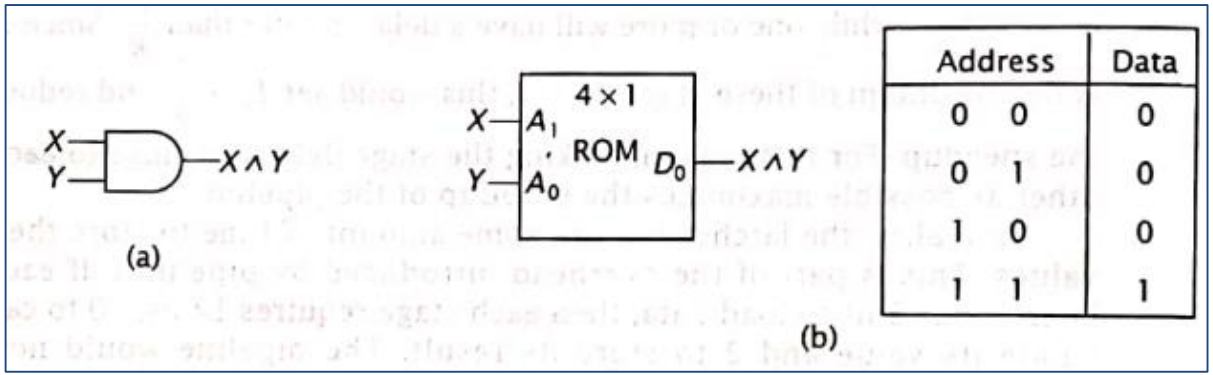
Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	--	--	--
2	A_2	B_2	A_1*B_1	C_1	--
3	A_3	B_3	A_2*B_2	C_2	$A_1*B_1+C_1$
4	A_4	B_4	A_3*B_3	C_3	$A_2*B_2+C_2$
5	A_5	B_5	A_4*B_4	C_4	$A_3*B_3+C_3$
6	A_6	B_6	A_5*B_5	C_5	$A_4*B_4+C_4$
7	A_7	B_7	A_6*B_6	C_6	$A_5*B_5+C_5$
8	--	--	A_7*B_7	C_7	$A_6*B_6+C_6$
9	--	--	--	--	$A_7*B_7+C_7$

2. Lookup Table

- A combinational circuit can be implemented by a ROM configured as LookUp table.
- The inputs to the combinational circuit serve as address inputs of the ROM.
- The data outputs of ROM corresponds to the outputs of combinational circuit.
- The ROM is programmed with data such that the correct values are output for any possible input values.

2. Lookup Table

- Example: Let us consider 4 x 1 ROM programmed to mimic a two input AND gate.
- The inputs to the AND gate serve as the address inputs to the ROM, and the output of the ROM corresponds to the AND gate output.
- By programming the ROM with the data shown, it outputs the same values as the AND gate for all possible of X and Y.

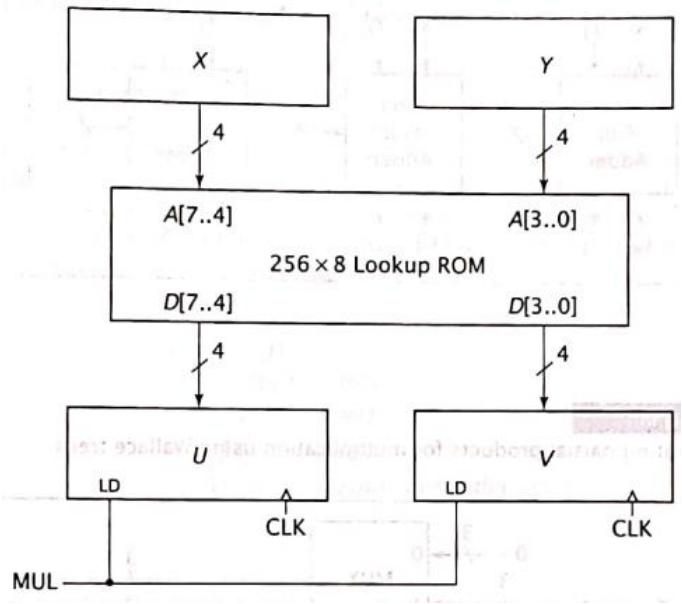


Address		Data
0	0	0
0	1	0
1	0	0
1	1	1

(a) An AND gate and (b) its lookup ROM equivalent

A Multiplier implemented using a Lookup ROM

- Recalling the shift-add multiplication algorithm for unsigned numbers, we can construct a hardware implementation for a multiplier using Lookup table.
- UV←X.Y
- Where U, V, X and Y are 4-bit registers.
- Registers X and Y supply the address inputs to the lookup ROM. Its inputs are the product of X and Y, and are routed to registers U and V.
- Each of the 256 locations must contain the 8-bit product of X and Y.
- For example, location 1011 1101 contains the data 1000 1111 (i.e 143) , the product of 11 and 13.



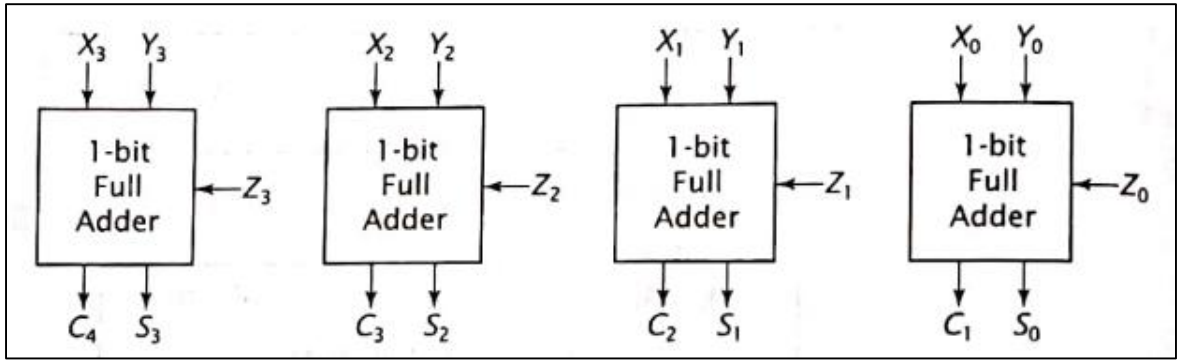
Learn more from: https://hardwarebee.com/overview-of-lookup-tables-in-fpga-design/

Assignment:

- What is Lookup ROM? Describe in brief demonstrating a Lookup ROM working as XOR gate. [PU 2018]
- Show the memory content of a Lookup ROM equivalent to input OR gate. [PU 2017 Spring]
- Design a Lookup ROM to implement the function $f=xy + y'z$. [PU 2019 Spring]

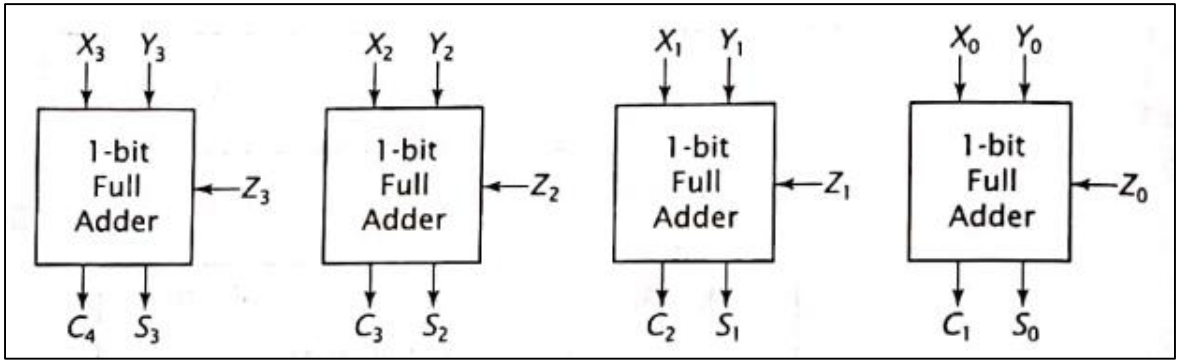
3. Wallace Tree

- A Wallace tree is a combinational circuit used to multiply two numbers. It requires more hardware but can produce a product in less time.
- It requires carry save adders and only one parallel adder.
- A carry-save adder can add 3 values simultaneously instead of just two. It outputs both sum and a set of carry bits.
- This is illustrated below:



A carry-save adder

Wallace Tree



A carry-save adder

- Each bit S_i is the binary sum of bits X_i, Y_i and Z_i
- Carry bit C_{i+1} is the carry generated by this sum.
- To form a final sum, C and S must be added together.
- Because carry bits do not propagate through the adder, it is faster than a parallel adder.

An example of addition for three numbers : 7+11+2

C	0 0 1 1 0
X	0 1 1 1
Y	1 0 1 1
Z	0 0 1 0
S	1 1 1 0

Adding S and C using a parallel adder, it produces the result 10100 (20) which is by 00110+1110 =10100.

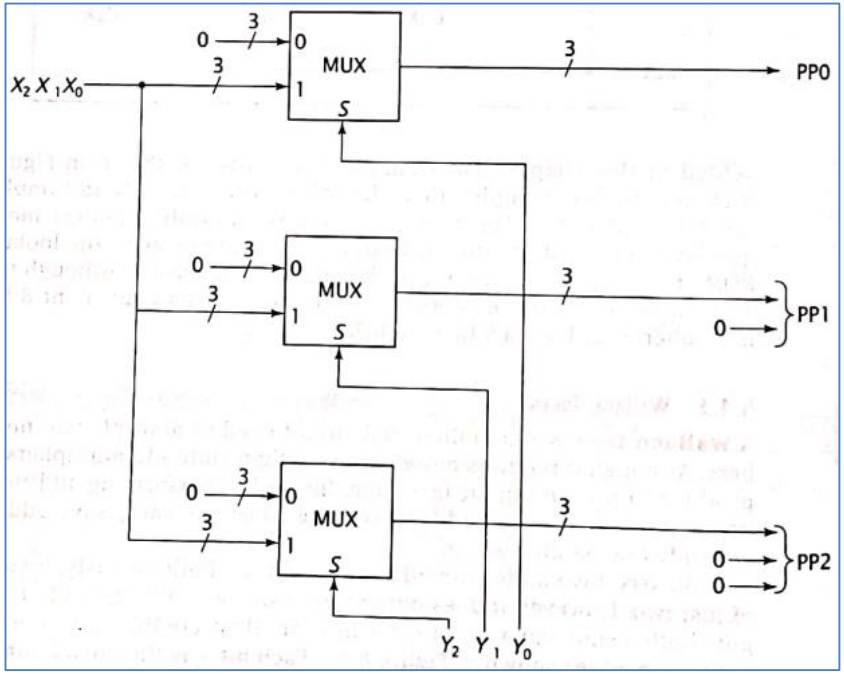
Wallace Tree

- To use a carry-save adder to perform multiplication we first calculate the partial products (PP) of the multiplication, then input them to the carry-save adder.
- Consider a numeric example multiplication of 7x6.

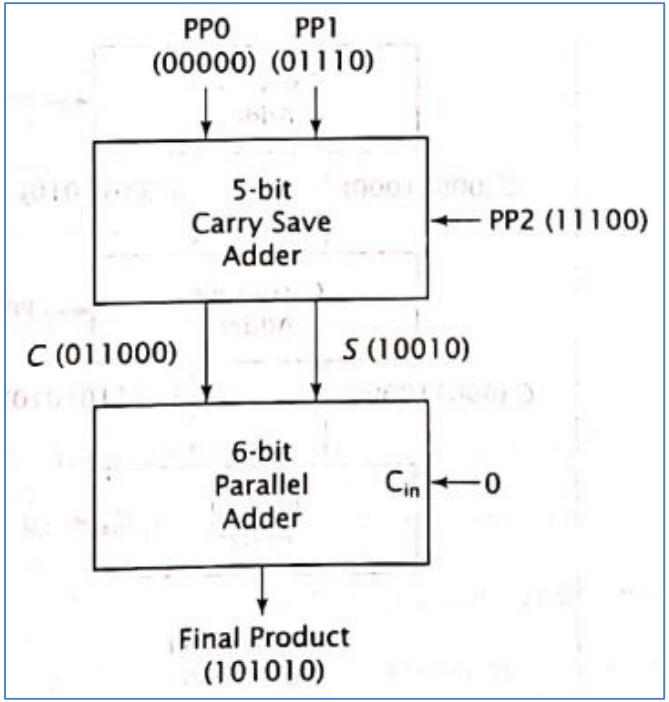
```
x =    111
y =    110
      000   ←PP0
     111    ←PP1
    111     ←PP2
   101010   ←Final sum calculated
```

A 3x3 Wallace Tree Multiplier

$$
\begin{array}{rl}
x = & 111 \\
y = & 110 \\
\hline
 & 000 \quad \leftarrow PP0 \\
 & 111 \quad \leftarrow PP1 \\
 & 111 \quad \leftarrow PP2 \\
\hline
 & 101010 \quad \leftarrow \text{Final sum calculated}
\end{array}
$$



Generating partial products for multiplication using Wallace trees



A 3 × 3 multiplier constructed using a carry-save adder

6.9 Floating Point Numbers

- The floating point representation of a number has two parts.
- The first part represent a signed fixed point number called **mantissa**.
- The second part designates the position of decimal (or binary) point and is called **exponent**.
- The fixed point mantissa may be fraction or an integer.
- For example:
- The decimal number **6132.789** is represented in floating point as 0.6132789×10^4
- This is in the form: $m \times r^e$
  - where m is mantissa and e is exponent.

Fraction	Exponent
0.6132789	04

- The value of exponent indicates that actual position of decimal point is four position to right of indicate decimal point in fraction

6.9 Floating Point Numbers

- For example a decimal number $3.25_{10} = 11.01_2$

$= 1.101_2 \times 10_2$

$= 1.101 \times 2^{1}$

Fraction	Exponent
1.101	01

6.10 IEEE 754 floating Point Standard, Numeric Format

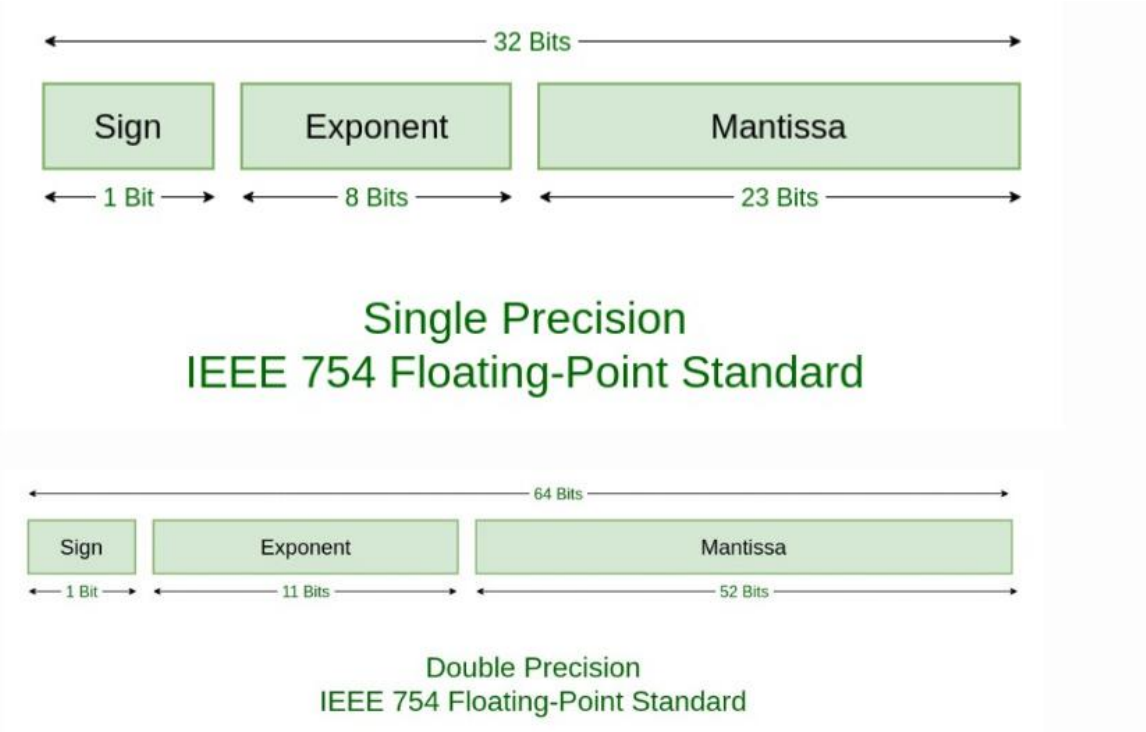
- The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation which was established in 1985 by the **Institute of Electrical and Electronics Engineers (IEEE)**.
- The standard addressed many problems found in the diverse floating point implementations that made them difficult to use reliably and reduced their portability.
- IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macs, and most Unix platforms.
- There are several ways to represent floating point number but IEEE 754 is the most efficient in most cases.

6.10 IEEE 754 floating Point Standard, Numeric Format

- There are several ways to represent floating point number but IEEE 754 is the most efficient in most cases. IEEE 754 has 3 basic components:
  - **The Sign of Mantissa –**
    This is as simple as the name. 0 represents a positive number while 1 represents a negative number.
  - **The Biased exponent –**
    The exponent field needs to represent both positive and negative exponents. A bias is added to the actual exponent in order to get the stored exponent.
  - **The Normalised Mantisa –**
    The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits. Here we have only 2 digits, i.e. O and 1. So a normalised mantissa is one with only one 1 to the left of the decimal.

6.10 IEEE 754 floating Point Standard, Numeric Format

- IEEE 754 numbers are divided into two based on the above three components: single precision and double precision.



Denormalization

- Denormalize: 1.101×2^1
- Here: exponent of base 2 =1

  Bias=127

  E=127+1 =128

Hence the denormalized encoding of 1.101×2^1 is:

Sign	Exponent	Significant no (fractional no)
0	1000 0000	101 0000 0000 0000 0000 0000
1 bit	*8 bits*	*23 bits*

Normalization

Sign	Exponent	Significant no (fractional no)
1	1000 0000	101 0000 0000 0000 0000 0000
1 bit	*8 bits*	*23 bits*

Here, sign bit =1 Hence, the no. Is negative
Exponent =1000 0000
So, E=128-127 =1

Also, S=101 0000 0000 0000 0000 0000
=1.101
Therefore, binary no is: $= -1.101 \times 2^1$
$=-1.101 \times 10$
$=-11.01_2$

Floating point Addition/Subtraction

- Let

  $X = \pm X_S + B^{\pm XE}$

  $Y = \pm Y_S + B^{\pm YE}$

- For $X_E = Y_E$,

Addition	$X+Y = (X_S + Y_S) \times B^{X_E}$
Subtraction	$X-Y = (X_S - Y_S) \times B^{X_E}$
Multiplication	$X*Y = (X_S * Y_S) \times B^{X_E + Y_E}$
Division	$X \div Y = (X_S \div Y_S) \times B^{X_E - Y_E}$

Given two fractional no: $X = 0.3 \times 10^2$ $Y = 3.0 \times 10^2$ Perform addition.

Solution:

Here, $X_S = 0.3$ $B^{XE} = 10^2$

$Y_S = 3.0$ $B^{YE} = 10^2$

Now,

$X+Y = (X_S + Y_S) \times B^{XE}$

$= (0.3 + 3.0) \times 10^2$

$= 3.3 \times 10^2$

Perform addition for the given binary floating numbers. And also normalize the result.

$X=-1.01 \times 2^2$ $Y=1.1 \times 2^4$

Solution:

$X = -1.01 \times 2^2$
$= -0.0101 \times 2^4$

$Y = 1.1 \times 2^4$
$= 1.1 \times 2^4$

Here,

$X+Y = (X_S+Y_S) \times B^{XE}$
$=(-0.0101 +1.1) \times 2^4$
$=1.0011X\ 2^4$

Now to normalize for 1.0011×2^4

Exponent of base 2 =4
Bias = 127 for 8 bit
E =127 + 4 = 131

Hence, normalized encoding of 1.0011×2^4 is:

0	**1000 1000**	**0011 0000 0000 0000 0000 0000**

Exam Questions:

1. Trace the multiplication of (4) and (3) using RTL code of shift Add Multiplication Algorithm. [PU 2017 Fall]
2. Write the RTL code for Booth's Algorithm. Show the hardware implementation for the RTL code. [PU 2017 Fall]
3. What is Wallace Tree? Describe in brief demonstrating a 3x3 multiplier being constructed using Wallace tree and carry save adder. [PU 2017 Fall]
4. What is Lookup ROM? Show the memory content of a Lookup ROM equivalent to input OR gate. [PU 2017 Spring]
5. Write the RTL code for Shift Add Algorithm. Show the hardware implementation for the RTL code. [PU 2017 Spring]
6. What is Arithmetic Pipelining? Describe in brief with an example. [PU 2017 Spring]
7. List out the RTL code for arithmetic and logical operations. Also show the implantation of addition and subtraction using parallel adder. [PU 2018 Fall]
8. Write down the RTL code for Booth's Algorithm. [PU 2018 Fall]
9. What is Lookup ROM? Describe in brief demonstrating a Lookup ROM working as XOR gate. [PU 2018 Fall]
10. Trace RTL code of shift add multiplication algorithm for multiplication of 3 and 4. [PU 2018 Spring]

Exam Questions:

11. What is Lookup ROM? Illustrate with an example how it works? [PU 2018 Spring]
12. Trace RTL code for Booth's Algorithm for multiplication of 9 and -3. [PU 2019 Spring]
13. What is Lookup ROM? Design a Lookup ROM to implement the function f=xy + y'z. [PU 2019 Spring]
14. Perform -7 x -2 using booth's algorithm. Can booths algorithm be used if both numbers are positive, if yes how? [PU 2020 Spring]
15. Write short notes on:
    a) Signed and Unsigned Number representation. [PU 2017 Spring]
    b) BCD addition [PU 2018 Fall]
    c) BCD adder [PU 2018 Spring]

End of Chapter