Flip-flop store 1 bit data

27/7/18

Date:
Page:

(ch-10)

COMPUTER ARITHMETIC

We consider addition, substraction, multiplication, division for following types of data:

→ Fixed point binary data in signed magnitude representation.
→ Fixed point binary data in signed 1's or 2's complement representation.
→ Floating point binary data. (mentiska stored in signed

◉ Addition & substraction with signed magnitude data. (sign of number is separately kept in flip-flop.)

operations :-

(+A) + (+B)	(+A) - (+B)	8 different operations for add. & sub.
(+A) + (-B)	(+A) - (-B)	
(-A) + (+B)	(-A) - (+B)	
(-A) + (-B)	(-A) - (-B)	

Operations	Add Mag.	Sub. mag. when A>B	A<B	A=B.
(+A)+(+B)	+(A+B)			Prevent -ve zero
(+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)

→ Operation is addition & sign are same then result is addition of two nos. and sign of A.

→ Operation is substraction, signs are different result is addition and sign of A.

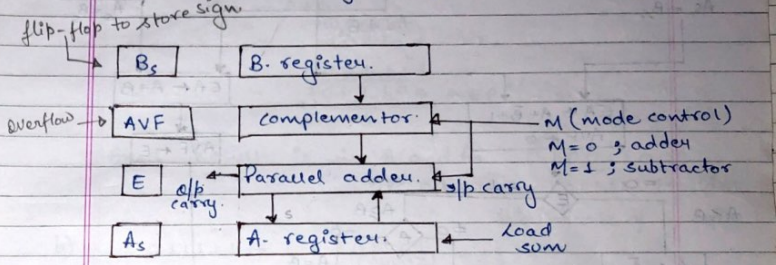
→ Operation is addition and signs are different then result is substraction and sign of result is that of greater number. (greater in magnitude)
⊙ if magnitude are equal then we write +(A-B) to prevent -ve zero.

→ when operation is substraction and i/p signs are same then result is substraction

→ Addition (substraction) algorithm –

⊙ when the signs of A & B are identical (different) add the two magnitudes and attach the sign of A to the result.
⊙ when the signs of A & B are different (same) compare the magnitudes and substract the smaller number from the larger, choose the sign of the result to be same as A if A>B or complement of A if A<B.
when A=B substract B from A and sign is +ve.

→ Hardware implementation :-
Basics hardware required are 2 register, 2 flip-flop, 1 adder, 2 subtractor (A-B, B-A), 1 comparator.
But we minimize these, because in substraction we add 2's complement to minuend so we can reduce the used of subtractor.
So, finally we'll need 2 register, 2 flip-flop, (sign) complementor, adder, 2 flip-flop (to store carry and overflow) [AVF → addition overflow flip-flop; if we get overflow then AVF = 1]



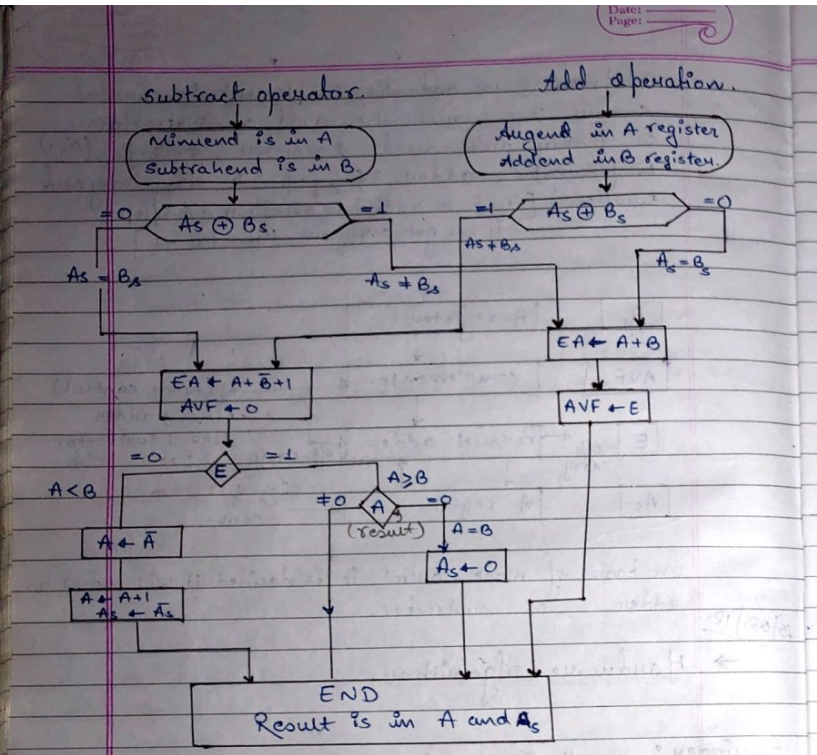
On basis of mode control, it is decided it will work as adder or subtractor.

3/08/18

→ Hardware Algorithm.

Reminder : Truth Table of Ex-OR

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0



Ques: a) perform the operation when the following signed magnitude numbers are computed in each case give the value of AVF. The left most bit in following numbers represent sign bit

a) 0 1011101 + 0 0111111
b) 1 0111111 + 1 1011101
c) 0 1011101 − 0 0111111
d) 0 1011101 − 0 1011101
e) 1 0111111 − 0 1011101

a) 0 1011101 + 0 0111111
+(45) + (+)(31) = +76

To verify ans calculate value of EA in decimal not of A

A = 1011101 As = 0
B = 0111111 Bs = 0
As ⊕ Bs = 0

EA ← A+B

```
  1 1 1 1 1
  1011101
+ 0111111
-------------
 1001100   ← EA = 64+8+4 = 76 Ans
 E  A
```

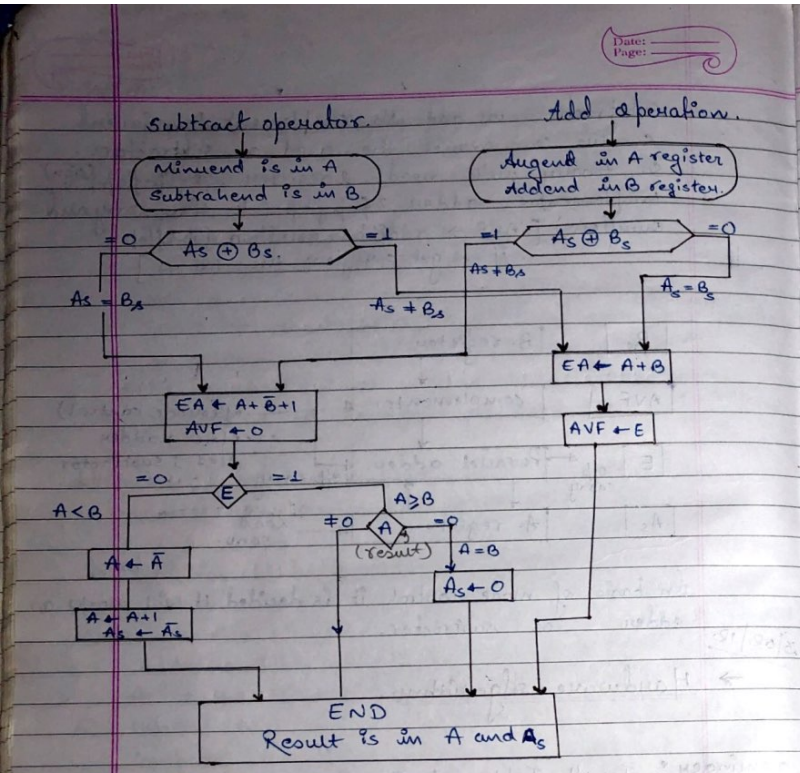
Ans: Result is in A and As. sign of result
0 001100
AVF = 1
= 12 should be 76
This is because of overflow

b) 1 0111111 + 1 1011101
−(31) + (−)(45)
As = 1 A = 0111111
Bs = 1 B = 1011101
As ⊕ Bs = 0
EA = A+B

```
  1 1 1 1 1
  0111111
+ 1011101
-------------
 10011100
 E  A
```

Ans: Result is in A and As 1 0011100
AVF = 1.

c) 0 1011101 − 0 0111111
+(45) − (+31) = +14
A = 1011101 As = 0
B = 0111111 Bs = 0
As ⊕ Bs = 0
EA ← A−B



Ques: a) perform the operation when the following signed magnitude numbers are computed in each case give the value of AVF. The left most bit in following numbers represent sign bit

a) 0 1011101 + 0 0111111
b) 1 0111111 + 1 1011101
c) 0 1011101 − 0 0111111
d) 0 1011101 − 0 1011101
e) 1 0111111 − 0 1011101

a) 0 1011101 + 0 0111111
+(45) + (+)(31) = +76

To verify ans. calculate value of EA in decimal not of A

A = 1011101 As = 0
B = 0111111 Bs = 0
As ⊕ Bs = 0
EA ← A+B

1 0 1 1 1 0 1
+ 0 1 1 1 1 1 1
1 0 0 1 1 0 0 ← EA = 64+8+4 = 76 Ans.
E A

Ans: Result is in A and As. 0 001100 (sign of result)
AVF = 1
= 12 should be 76
This is becoz of overflow

b) 1 0111111 + 1 1011101
−(31) + (−)(45)
As = 1 A = 0111111
Bs = 1 B = 1011101
As ⊕ Bs = 0
EA = A+B

1 1 1 1 1 1 1
0 1 1 1 1 1 1
+ 1 0 1 1 1 0 1
1 0 0 1 1 0 0
E A

Ans: Result is in A and As 1 0011100
AVF = 1.

c) 0 1011101 − 0 0111111
+(45) − (+31) = +14
A = 1011101 As = 0
B = 0111111 Bs = 0
As ⊕ Bs = 0
EA ← A+B

Date: Page:

1011101 ← Minuend
S = 0111111 ← subtrahend

$\bar{B}$ = 100000
$\bar{B}+1$ = 100001
$A+\bar{B}+1$ = 1011101
+ 100001
1001110
E A

Ans: Result is in A and As. 0 001110
AVF = 0 (on subs. no overflow) As.

d) 0101101 - 0101101
A = 1011101 As = 0
B = 1011101 Bs = 0
As ⊕ Bs = 0
EA ← A-B = A+$\bar{B}$+1
$\bar{B}+1$ = 0100010 + 1
$\bar{B}+1$ = 0100011
A=0 (in result)
⇒ A=B.

A+$\bar{B}$+1
1011101
+ 0100011
1000000
E A
=0

Ans: As ← 0 AVF = 0
A = 000000

e) -31 - (+45) = -76
1011111 - 0101101
A = 0111111 As = 1 As ⊕ Bs = 1.
B = 1011101 Bs = 0
EA ← A+B. EA ← 1001100
E = 1 ; AVF = 1
As = 1.

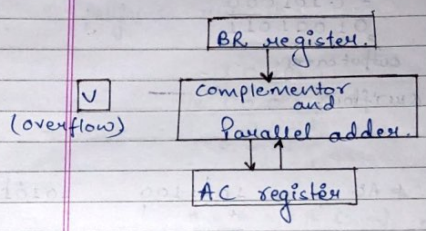
→ number stored with sign bit

Date: 04/08/18 Page:

⊙ Addition and substraction with signed 2's complement data.

→ Hardware Implementation.

no flip-flop required, number are stored with sign bit, V flipflop to store overflow.



[we check overflow by comparing i/p & o/p carry.]

→ Hardware Algorithm.

Subtract operation

Minuend is in AC
Subtrahend is in BR
AC ← AC + $\overline{BR}$ + 1
V ← overflow
End

Addition operation.

Augend is in AC
Addend is in BR.
AC ← AC + BR
V ← overflow
End.

Ques. Perform arithmetic operation of binary numbers and with -ve nos. in signed 2's complement. Use 7-bits to accomodate each number together with its sign. In each case, determine the overflow.

(a) $(+35)_A + (+40)$

(b) $(-35)_A + (-40)$

(c) $(-35)_A + (+40)$

2	35	2	40
2	17 – 1	2	20 – 0
2	8 – 1	2	10 – 0
2	4 – 0	2	5 – 0
2	2 – 0	2	2 – 1
	1 – 0		1 – 0

a) $+35 = 0\ 100011 \Leftarrow AC$

$+40 = 0101000 \Leftarrow BR$

$AC + BR =$ 0 ← ip carry.

$0\ 100011$

$+\ 0101000$

$0\ 1001011$

↗ output carry.

V = overflow.

b) $+35 = 0100011$

$-35 = 1011101 \Leftarrow AC$

$+40 = 0101000$

$-40 = 1011000 \Leftarrow BR.$

$AC + BR.$

1011101

1011000

$1\ 0\ 1\ 0\ 1\ 0\ 1$

$1011100 + 1 = 1011101$

$1010111 + 1 = 1011000$

V = overflow.

11/08/18

Booth → 2's comp.

Multiplication Algorithms.

→ we use partial product to reduce number of required registers.

→ we do right shift in system (but manually we shift left)

→ Multiplication with zero has no effect so we don't perform with it.

10111 ← multiplicant

$\times 10011$ ← multiplier.

10111

$10111\times$

$\times\times$

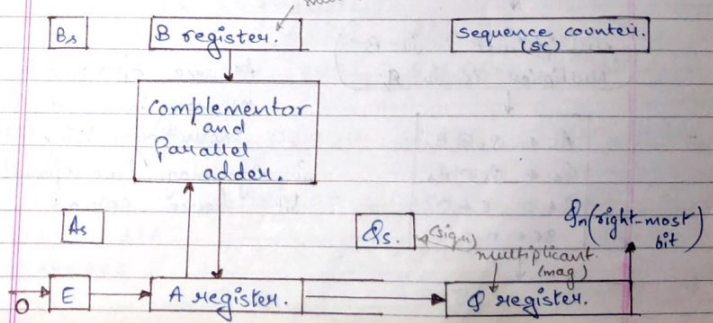
$\times\times\times$

$*******$ (som)

In signed magnitude, we perform partial product i.e. first 1 bit of multiplier gives 1 partial product

This method reduces required number of registers. and secondly multiplicant shifts to right. and we ignore zero.

Hardware implementation for signed magnitude data.



- B register stored multiplicand magnitude and B_s stores its sign.
- A register stores multiplier magnitude and Q_s stored its sign.
- Sequence counter stores the bit of multiplier.
- A register is initialised with zero. Sequence counter decrements its value and multiplication continues till its sequence counter becomes zero.
- Result is stored in AQ register.
- Q_n has right most bit of multiplier if its value is 1 then it adds multiplicand with value in A.

→ shr EAQ logical shift (right)

Ex. A: 10010

shr A : 01001

shl A : 00100

shl (logical left shift)

→ cir (circular shift)

Hardware Algorithm.

Multiply operation

multiplicand is in B

Multiplier is in Q.

$A_s \leftarrow Q_s \oplus B_s$

$Q_s \leftarrow Q_s \oplus B_s$

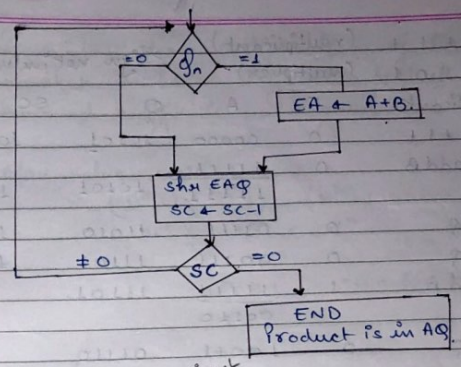
$A \leftarrow 0$, $E \leftarrow 0$

$SC \leftarrow n-1$

Assume operand in memory is of n bit including sign bit.

in sign-mag. we separate sign bit hence $SC = n-1$

Date: 11/08/18



Ex multiplicand

10111 = B (signs are not included)

× 10011 = Q

	E	A	Q	SC
multiplicand B = 10111				
(B = 10111)				
Multiplier in Q.	0	00000	10011	101 (5)
		10111		
Q_n = 1, add B.	0	10111	10011	
(first partial Product)				
shr EAQ	0	01011	11001	100 (4)
Q_n = 1, add B		10111		
	1	00010	11001	
shr EAQ	0	10001	01100	011 (3)
Q_n = 0, shr EAQ	0	01000	10110	010 (2)
Q_n = 0, shr EAQ	0	00100	01011	001 (1)
		10111		
Q_n = 1, add B.	0	11011		
shr EAQ	0	01101	10101	(000) (0)

Result

Date: 14/8/18
Page:

Ques. 11111 (multiplicand) Sign not included.
10101 (multiplier)

Ans.

Multiplicand	E	A	Q	SC.
B = 11111	0	00000	10101	101
Q_n=1, add B	0	11111		
		11111	10101	101
Shr EAQ	0	01111	11010	100
Shr EAQ	0	00111	11101	~~100~~ 011
Q_n=1 Add B	1	11111		
		00110	11101	011
Shr EAQ	0	10011	01110	010
Shr EAQ	0	01001	10111	001
Q_n=1 add B	1	11111		
		01000	10111	001
Shr EAQ	0	10100	01011	000

result (1010001011)

(b) Sign included.

B_s = 1 11111 = -15
A_s = 1 10101 = -5
$A_s \oplus B_s = 1 \oplus 1 = 0$

Multiplicand	E	A	Q	SC
B = 1111	0	0000	0101	100
Q_n=1, add		1111		
		1111	0101	100
Shr EAQ	0	0111	1010	011
Shr EAQ	0	0011	1101	010
Q_n=1, add B	1	1111		
		0010	1101	010
Shr EAQ	0	1001	0110	001
Shr EAQ	0	0100	1011	000

Result : 0 0100 1011
sign

Ques. Prove that the multiplication of two n-digit number in base r gives a product no more than 2 n-digit in length.

Max value $= r^n - 1$

$(r^n-1)(r^n-1) \leq (r^{2n}-1)$

$r^{2n} + 1 - 2r^n \leq r^{2n} - 1$

$2 \leq 2r^n$

$r^n \geq 1$ true for $r \geq 2, n \geq 1$.

$1 \leq r^n$

18/8/18

Booth Multiplication

→ Numbers (-ve) are in 2's complement form.
→ Sequence counter value will be n
→ (Sign bit is included with number)
→ We do 2-bit inspection Q_n and Q_{n+1} = 10 then we substract, 00 & 11 → no operation (only shift), 01 → add

* Booth algorithm gives procedure for multiplying binary integers in signed 2's complement representation.
It operates on the fact that, string of 0's in the multiplier require no addition but just shifting. but a strings of 1's in multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$

+14 = 01110 ⇒ 2^3 to 2^1
k=3, m=1

$2^{k+1} - 2^m = 2^4 - 2^1 = 14$

M x (+14)
⇒ M x 2^4 - M x 2^1

Q_n Q_{n+1} (initialised zero)

+14 = 0 1 1 1 0

$2^4 - 2^1$ at 1 position we're bringing -ve sign at 4 position " +ve sign.

-14 = 1 0 0 0 1 0

+ 1

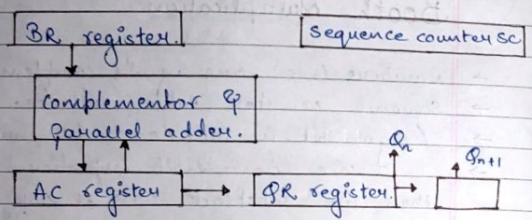
1 0 0 1 1 0

Q_n Q_{n+1} = 0

1 0 0 1 1 0 → no op^n, sub, add

$-2^1 + 2^2 - 2^4 = -14$

Hardware implementation

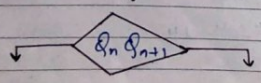


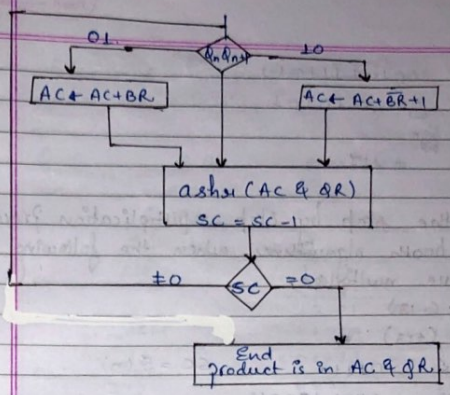
Hardware Algorithm

Multiply op^n

Multiplicant is in BR.
Multiplier is in QR.

AC ← 0
Q_{n+1} ← 0
SC ← n





Ques. (-9)×(-13)

BR = -9 QR = -13

+9 = 01001 +13 = 01101

-9 = 10111 (2's comp^n) -13 = 10011

BR ← 10111 QR ← 10011

8 4 2 1
1 0 0 1
1 1 0 1

1 0 0 1
+ 1
1 0 1 1 0
→ 1
1 0 1 1 1

$Q_n Q_{n+1}$		BR = 10111, $\overline{BR}$+1 = 01001	AC	QR	Q_{n+1}	SC
	Initial (AC↑$\overline{BR}$+1)		00000	10011	Q_n 0 Q_{n+1}	101 (5)
10	sub. BR		01001			
			01001	10011	0	
	ashr	Sign bit →	00100	11001	1	100 (4)
11	ashr		00010	01100	1	011 (3)
01	add BR		01011			
			01001	01100	1	
	ashr		00100	10110	0	010 (2)
00	ashr		00010	01011	0	001 (1)
10	sub. BR	2's comp^n	01001			
		①	01011	01011	0	
	ashr		00101	10101	1	000 (0)

Result

Result : 00101111101

00011101011

Sign bit.

+117

Ques. Show the step by step multiplication process using booth algorithm when the following binary nos. are multiplied.

(a) (+15) × (−13)

(b) (+15) × (+13)

BR = +15 01111

QR = −13 11001

$\bar{Q}_n$

SC = 5 (no.)

$Q_n Q_{n+1}$		BR	AC	QR	Q_{n+1}	SC
		= 01111				
		$\overline{BR}+1$ = 10001				
~~10~~	Initially,		00000	11001	0	101
10	sub BR		10001			
			10001	11001	0	
	ashr		11000	11100	1	100
11	ashr		11100	01100	1	011
01	~~add BR~~		01111			
			01011	01100	1	
	add BR		01011	10110	0	010
	ashr		00101	11011	0	
00	ashr		00010	11011	0	001
10	sub BR		10001			
			10011	11011	0	
			11001	11101	1	

00110000011

+195

2's

−195

−ve 512 32 16 8 4 1

2's comp form.

convert to get 195

(b)

BR = +15 = 01111

QR = +13 = 01101

$Q_n Q_{n+1}$		BR	AC	QR	Q_{n+1}	SC
		= 01111				
		$\overline{BR}+1$ = 10001				
	Initially.		00000	01101	0	101
10	sub BR		10001			
			10001	01101	0	
	ashr		11000	10110	1	100
01	add BR		01111			
			00111	10110	1	
	ashr		00011	11011	0	011
10	sub BR		10001			
			10100	11011	0	
	ashr		11010	01101	1	010
11	ashr		11101	00110	1	001
01	add BR		01111			
			01100	00110	1	
	ashr		00110	00011	0	

11001

01011

01000100

Ques. 100 x (-110)

BR = +100 0001100100
QR = +200 0011001000
2's 1100110111
QR = -110 = 11001010
110 = 1101110

$Q_n Q_{n+1}$		BR	AC	QR	Q_{n+1}	SC
		=01100100				
		BR+1 = 10011100				
	Initially.		AC 00000000	Q_n 10010010	0	1000
00	ashr		00000000	01001001	0	0111
10	sub BR	10011100				
		10011100	01001001	0		
	ashr	11001110	00100100	1		0110
01	add BR	01100100				
		00110010	00100100	1		
	ashr	00011001	00010010	0		0101
00	ashr	00001100	10001001	0		100
10	sub BR	10011100				
		10101000	10001001	0		
	ashr	11010100	01000100	1		011
01	add BR	01100100				
		00111000	01000100	1		
	ashr	00011100	00100010	0		010
00	ashr	00001110	00010001	0		001
10	sub BR	10011100				
		10101010	00010001	0		
	ashr	11010101	00001000	1		

11000

Look-at-carry

Date: 21/8/18

ARRAY MULTIPLIER — uses adders gates, encoders

It is an combinational circuit, produces all product bit at once. It produces multiplication fast.

→ checking the bits of the multiplier one at a time and forming partial products is a sequential operations that requires a sequence of add and shift micro-operations.

→ Multiplication of 2 binary numbers can be done with one microoperation by means of a combinational circuit that forms the product bits all at once.

→ This is a fast way of multiplying two numbers since, all it takes the time for the signals to propagate through the gates that form the multiplication array.

for Ex: 2bit x 2bit

b_1 b_0 (multiplicand) 2x2 = 4 AND gate required.
a_1 a_0 (multiplier)

a_0b_1 a_0b_0 — works as and gate

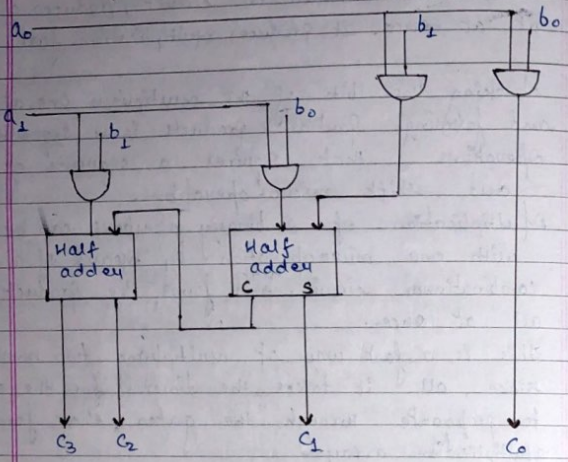
a_1b_1 a_1b_0 x

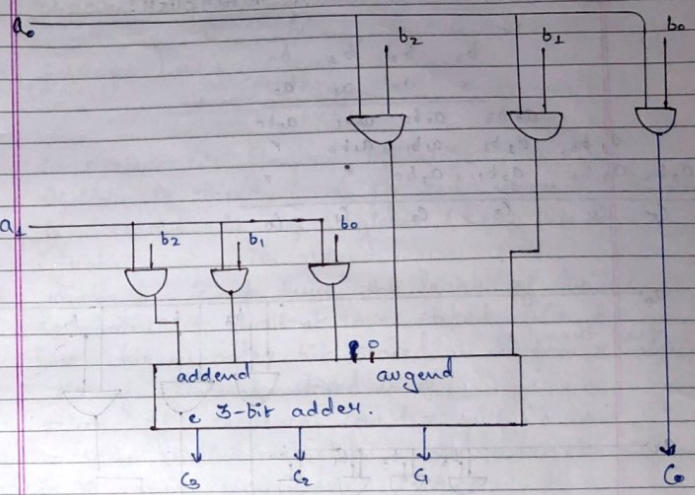
C_3 C_2 C_1 C_0

C_3 ↗ output carry.

a_0b_0, a_0b_1 are 1st partial product.
a_0b_0 give result as AND gate.
a_0b_1, a_1b_0 require adder b/w them to produce C_1 and produce o/p carry which will be input to another adder along with a_1b_1. So we'll need 2 half-adders because we need to give 2 inputs.

2 bit by 2-bit Array Multiplier.





NOTE

- j ≡ multiplier bits
- k ≡ multiplicant bits.
- $(j+k)$ ≡ bits in final product (result)
- $(j \times k)$ ≡ number of AND gates.
- $(j-1)$, k bits adders.
- 2^j ≡ no. of adders.

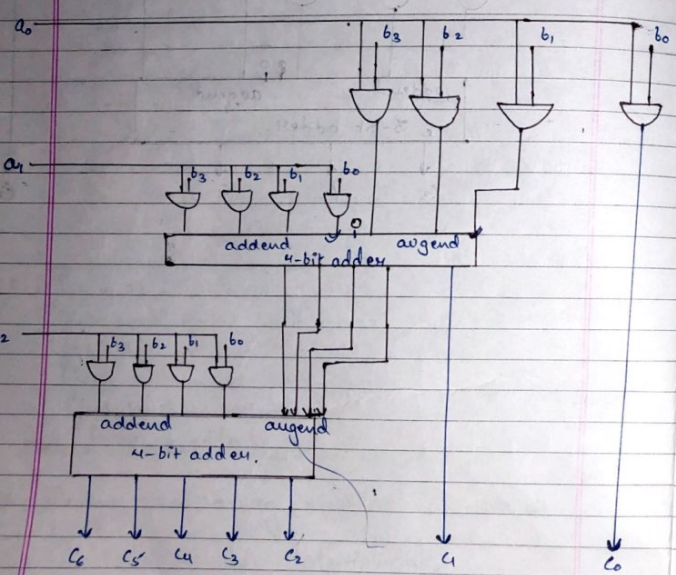
Ques. Design an array multiplier of 3x2 bit.

b_2 b_1 b_0 (multiplicant)
× a_1 a_0 (multiplier)

a_0b_2 a_0b_1 a_0b_0
a_1b_2 a_1b_1 a_1b_0
C_3 C_2 C_1 C_0

Ques. Design 4x3 array multiplier.

b_3 b_2 b_1 b_0
× a_2 a_1 a_0
a_0b_3 a_0b_2 a_0b_1 a_0b_0
a_1b_3 a_1b_2 a_1b_1 a_1b_0 ×
a_2b_3 a_2b_2 a_2b_1 a_2b_0 × ×
C_6 C_5 C_4 C_3 C_2 C_1 C_0



Division Algorithms

Ex: 10001 (divisor)) 01100000000 (1---- quotient.
(dividend)
-10001

→ In system, we perform left shift.
→ Divisor is stored in B register. whereas, dividend is stored in AQ register (here we don't initialise A with zero).
→ Dividend is of twice the length of divisor.
→ Remainder & quotient are stored in AQ.
→ ~~First~~ next half higher order bit stored in A ~~four~~ remaining bit of dividend stored in Q.
→ Value of sequence counter will be n-1, as SC = 0, division stops, and we get quotient in Q while remainder in A.
→ SC value is same as that number of bit in Q.
→ Hardware is that of signed magnitude of multiply.
→ As ⊕ Qs = sign of result.
→ In system, we get divide overflow condition.
[if we get quotient $\overset{(5+1)}{n+1}$ bit that is divide overflow] Q = 5 bit A = 5 bit according to above example.

when value A reg < B reg. we don't get divide overflow.
when value A reg ≥ B reg. we get divide overflow.

Ex 10001) 11100000000 (1-----
10001

0

Higher order half bit of dividend ≥ divisor ⇒ Overflow

→ For divide overflow, we use DVF overflow. if overflow exists DVF = 1 else DVF = 0

Hardware implementation for signed magnitude data.

Hardware is identical to required for multiplication & consists of components –

register EAQ is now shifted to the left with zero inserted into the Q_n & previous value of E is lost.

Divisor is stored in B register and double length dividend is stored in A and Q register.

Divide overflow.

A divide overflow condition occurs if the high order half bits of the dividend constitute a number is greater than equal to the divisor.

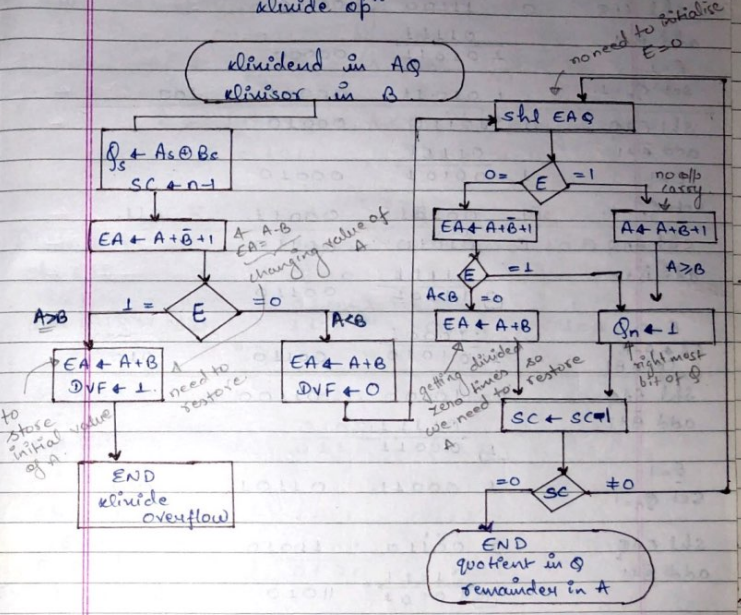
Another problem associated with division is that division by zero must be avoided.

Overflow condition can be detected when a special flip-flop is set (DVF)

Hardware

Hardware Algorithm. (Restoring Method)

divide op^n



Solve :

Ques. 10001) 011110000000 (

AQ ← 01111 0000000

A Q

B ← 10001

A<B , no divide overflow.

Divisor B = 10001. , B̄+1 = 01111

	E	A	Q	SC	
dividend		01110	00000	101 (5)	
shl EAQ	0	11100	00000		
add $\bar{B}+1$		00101			
E=1.	1	01001	00000		
set $Q_n=1$.			00001	100	
shl EAQ	0	10010	00010		
add $\bar{B}+1$		00101			
E=1	0	00111	00010		
set $Q_n=1$.			00011	011	
shl EAQ	0	01110	00110		
add $\bar{B}+1$		00101			
E=0	0	10011	00110		
EA ← A+B (add B)		01010	00110	010	divide zero times
shl EAQ	0	10100	01100		
add $\bar{B}+1$		00101			
E=1	1	01001	01100		
set $Q_n=1$			01101	001	
shl EAQ	0	10011	11010		
add $\bar{B}+1$		00101			
E=0, add B	0	11000	11010	000	
		00110	11010		
		=6	16+10=26		
		Remainder	quotient		

Ques Show the contents of register EAQ and SC during the process of division of –

a) 10100011 by 1011

b) 00001111 by 0011.

(a) AQ ← 10100011

A = 1010, Q = 0011

B ← 1011. A < B, no overflow divide

Divisor B = 1011, $\bar{B}+1$ = 0101

	E	A	Q	SC	
Dividend		1010	0011	100	
shl EAQ	1	0100	0110		
E=1		0101			
add $\bar{B}+1$ in A	1	1001	0111	011	0
$Q_n=1$					
shl EAQ	0	0010	1110		
E=1		0101			
add $\bar{B}+1$	1	0111	1111		010
$Q_n=1$					
shl EAQ	0	1111	1110		
E=0		0101			
EA ← A+$\bar{B}+1$	1	0100	1111		001
E=1, $Q_n=1$					
shl EAQ	0	1001	1110		
E=0		0101			
	0	1110	1110		
EA = A+B		1011			
	1	1001	1110	000	
		remainder	quotient		

(b) AQ ← 0000 1111 (8 4 2 1 / 5 3 15)

B ← 0011 A < B, no divide overflow.

divisor B = 0011 $\bar{B}+1$ = 1101

	E	A	Q	SC
dividend.		0000	1111	100
shl EAQ	0	0001	1110	
EA ← A+$\bar{B}$+1		1101		
E = 0	0	1110		
add B.		0011		
	1	0001	1110	011
shl EAQ	0	0011	1100	
EA ← A+$\bar{B}$+1		1101		
E = 1	1	0000		
Set Q_n = 1			1101	010
ashl EAQ	0	0001	1010	
E = 0		1101		
EA ← A+$\bar{B}$+1	0	1110	1010	
E = 0 EA ← A+B		0011		
	1	0001	1010	001.
shl EAQ	0	0011	0100	
E = 0, EA ← A+$\bar{B}$+1		1101		
E = 1, Q_n = 1	1	0000	0101	SC = 000

Quotient : 0101.

Remainder : 0000

Date: 27/8/18
Page:

⊙ Floating Point arithematic operations.

$\pm m \times r^{\pm e}$ → exponent
→ raddix
↓ mentissa (int or frac.)

Ex represent 543.276 as floating point.

$.543276 \times 10^{3}$ ← exponent
mentissa

Mentissa is stored in another register whereas exponent is stored in other register. in system.

→ For any operation to be performed, floating nos. should be normalised. initially (non-zero)

Normalise

If msb of mentissa is 1 then number is normalised.

Ex. $.\underset{\text{non-zero}}{1}0101 \times 2^{101}$ ← normalised
$.\underset{\text{zero}}{0}0101 \times 2^{101}$ ← not normalised.

we can normalise a number by shifting zero, and exponent will be decremented.
After operation also, numbers should be normalised.

→ when we add / sub. floating nos. we need to check exponent. If exponents are same then directly perform op^{n}. else make them (exp.) equal before addition / substraction.

Ex. $.53724800 \times 10^{2}$
$.12341200 \times 10^{4}$

left shift (exponent decreases)

537.24800×10^{4}
$.12341 2 \times 10^{4}$
$537.\text{----} \times 10^{4}$

now can add mentissa.

OR $\cdot 537 \times 24800 \times 10^{+2}$

$\cdot 0001 2341 \times 10^{2}$ (right shift)

$\cdots \times 10^{2}$ ↓ exponent increment.

Right shift is preferred because in left shift we lose MSB which produce more chance of error. LSB loss can be accepted because it produces least error.

Lower value exponent is increased by right shift. Above process is called. Align Mentissa.

→ $\cdot 1010101 \times 2^{101}$

$+ \cdot 100001 \times 2^{101}$

1.010110×2^{101}

Overflow.

overflow can be removed by shifting it to right once and exponent is incremented by 1.

→ Underflow is seen when we substract two nos. and we get 0 at MSB, it can be removed by left shift.

→ On multiplication & division we don't do mentissa align. directly exponent is added & substracted respectively.

* When 2 normalised mentissa's are added the sum may contain an overflow digit. An overflow can be corrected by shifting the sum once to the right and incrementing the exponent.

* when 2 nos. are substracted, the result may contain MSB 0

A floating point number that has 0 in the most significant position of mentissa is said to have an underflow.

To normalise a number that has underflow it is necessary to shift the number to the left and decrement the exponent.

* The op^n performed with mentissa's are same as in fixed point nos.

The op^n performed with exponents are compare & increment. (align the mentissa), add and substract (multiplication & division) and decrement (to normalise the result)

* The exponent may be represented in any of three representations.

→ signed magnitude

→ signed 1's complement

→ signed 2's complement.

The fourth representation is biased exponent

we assume [Mentissa — signed magnitude

Exponent — biased exponent.

bias value = $r^{k-1} - 1$

r = base

k = no. of bits of exp.

biased exponent ⇓

bias value + True exponent.

for ex: we have exp value -50 to 49

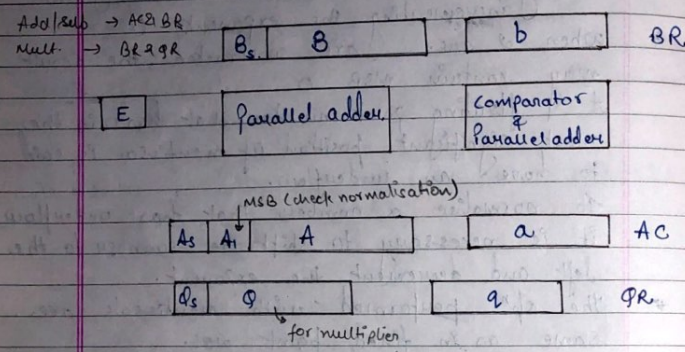
bias = 50] how many bit we reserve.

-50+50 to 49+50

0 to 99 ← +ve exponent

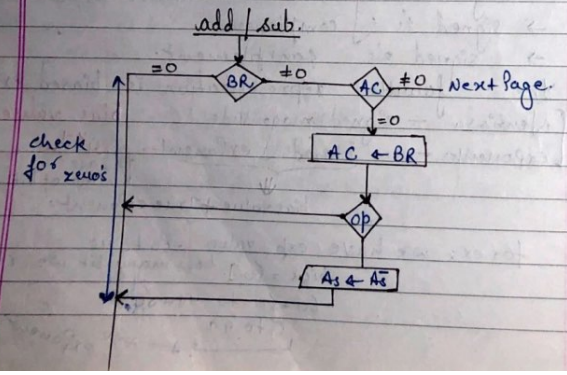
Register Configuration for floating point arithmetic operations.

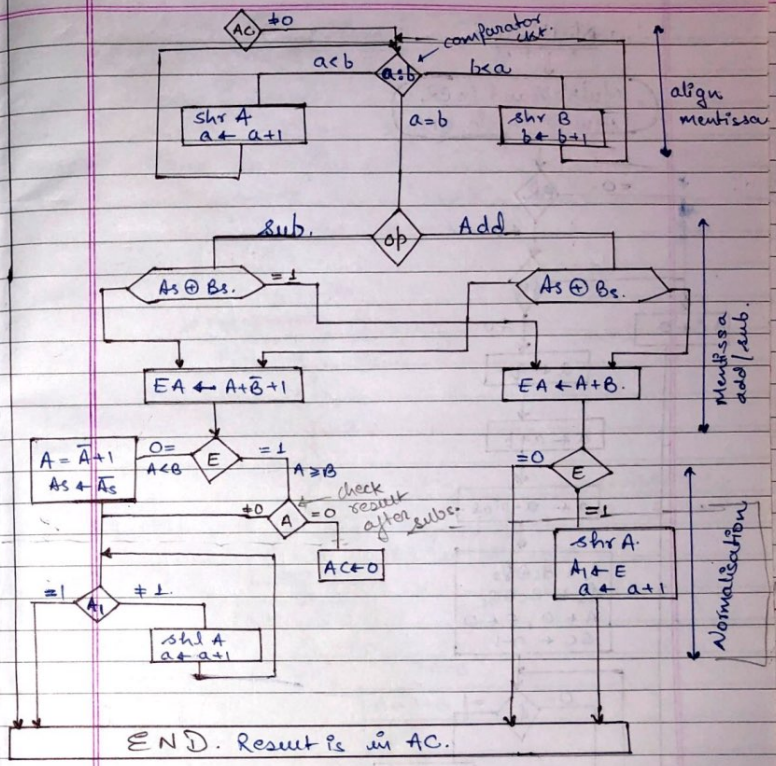
uppercase - mentissa ; lowercase - exponent.



Addition & substraction algorithm.

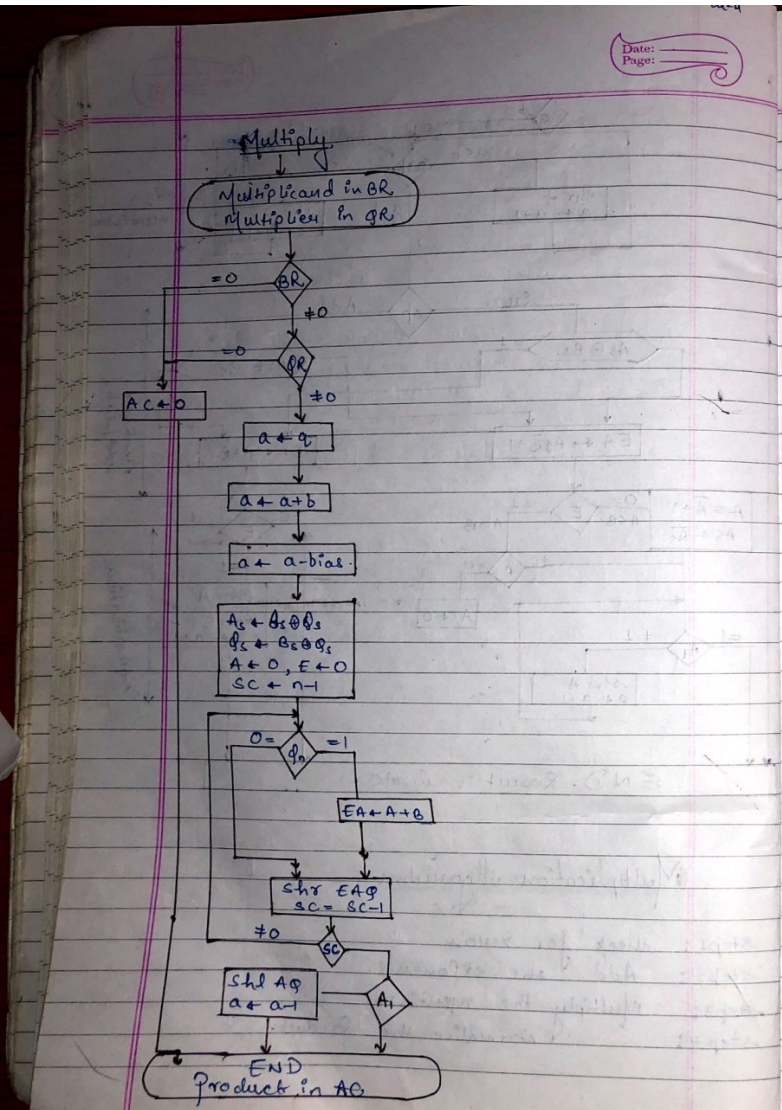
step 1: check for zero's
step 2: Align the mentissa.
step 3: Mentissa addition or substraction.
step 4: Normalise the result.





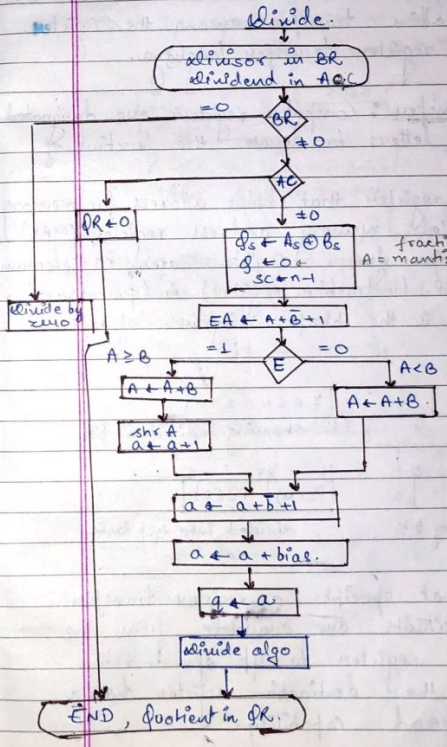
Multiplication Algorithm

step1: check for zero's
step2: Add the exponents.
step3: Multiply the mentissa
step4: Normalise the product.



Division Algorithm.

Steps 1: check for zeros.
step 2: Initialize registers & evaluate the sign.
Steps 3: Align the dividend.
step 4: subtract the exponent
steps 5: divide the mantissas



P: $R_2 \leftarrow R_1$] control function

control Func = 1 means
= 0 no transfer b/w R_1 & R_2

if (P=1) then $R_2 \leftarrow R_1$] symbolic

control signal value = 1

Date: 30/9/18
Page:

(ch-4)

Register Transfer & Micro-operation

RTL = Register Transfer Language ← Symbolic notation.
Micro-operation is the elementary op" performed on data stored in registers.

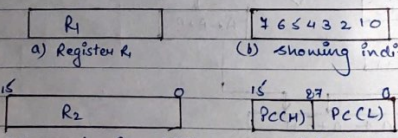
→ Register
→ Operation
→ control unit
→ initiate the op"
→ Descriptive
→ Symbolic (RTL)

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

Register Transfer : Computer registers are designated by capital letters to denote the function of register.

For ex: the register that holds address for memory unit is called memory address register. (MAR)

⊙ Other designation for registers are PC (program counter) IR (instruction register) R_1 (processor register). This is the block diagram of register.



⊙ A statement that specifies a register transfer implies that circuits are available from the o/p of source register to i/p of destination register. and the destination register has a parallel load capability.

MAR ← Memory add. reg.
PC ← Program counter
R1, R2, R3 ← Process reg
(called IR letters) Instruction reg.

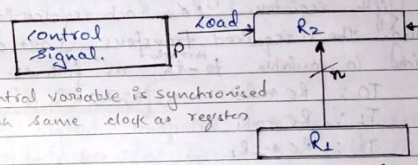
Register
↓
individual flip-flop
7 6 5 4 3 2 1 0
R2 R1 PC(H) PC(L)

Date:
Page:

This can be shown by if-then statement -
if (P=1) then ($R_2 \leftarrow R_1$)
where P is control signal generated in the control section.

⊙ It is sometimes convinient to separate control variables from the register transfer operation by specifying a control function.
A control function is a boolean variable. That is equal to 1 or 0.

P: $R_2 \leftarrow R_1$ ↴ Block diagram.

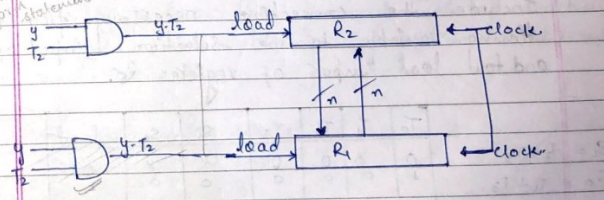


Ques. show the block diagram of hardware that implements the following register transfer statements.

yT_2 : $R_2 \leftarrow R_1$, $R_1 \leftarrow R_2$

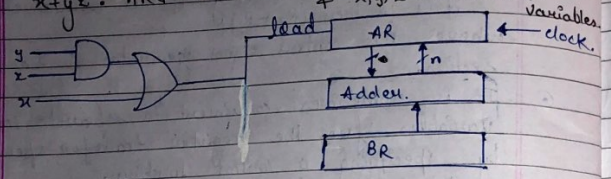
if P=1 then perform statement

y, T_2 variables



(b) $x+yz$: $AR \leftarrow AR + BR$.

AR & BR are 2 nbit registers. & x, y, z are control variables.



Ques. The output of four registers ~~are~~ R_0, R_1, R_2, R_3 are connected through $2^2 = 2$ Select Lines; 4x1 MUX to the inputs of a fifth register R_5. Each register is 8 bit long. The required transfers are dictated by 4 time in variable T_0 through T_3 as follows.

T_0 : $R_5 \leftarrow R_0$ $\quad T_3 : R_5 \leftarrow R_3$.

T_1 : $R_5 \leftarrow R_1$

T_2 : $R_5 \leftarrow R_2$

The timing variables are mutually exclusive which means that only one variable is equal to 1 while the others are equal to 0.

Draw a block diagram showing the hardware implementation of register transfers. Include the connections necessary from 4 timing variable to the selection input of MUX and to load input of register R_5.

4.9
4.10
4.11

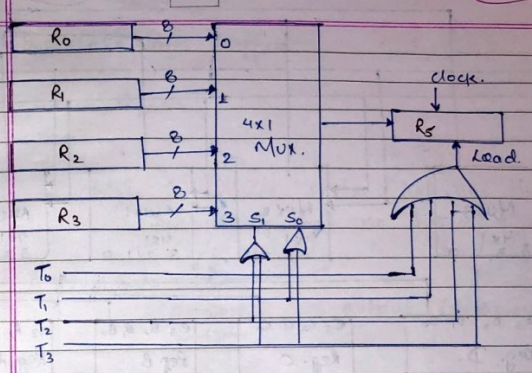
T_0	T_1	T_2	T_3	S_1	S_0	load
0	0	0	0	x	x	0
1	0	0	0	0	0	1
0	1	0	0	0	1	1
0	0	1	0	1	0	1
0	0	0	1	1	1	1

$S_1 = T_2 + T_3$

$S_0 = T_1 + T_3$.

$Load = T_0 + T_1 + T_2 + T_3$

4x1 MUX.



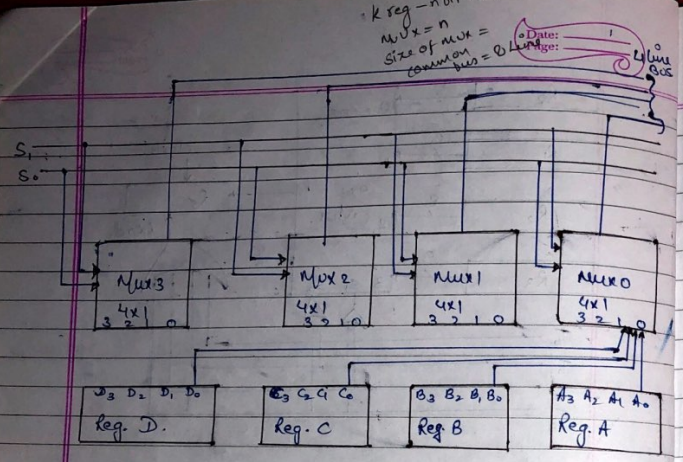
Bus Transfer

A digital computer has many registers & path must be provided to transfer information from one register to another.

→ A efficient scheme for transferring information between register in a multiple register configuration is a common bus systems.

→ A bus structure consists of a set of common lines, one for each bit of register through which binary information is transfered one at a time.

S_1	S_0	Reg. selected
0	0 = 0	A (0 at Mux)
0	1 = 1	B
1	0 = 2	C
1	1 = 3	D



K-registers. Each of n-bits.

→ number of MUXs = n

→ Size of MUX = K x 1

→ no. of Select lines in MUX = $\log_2 K$

n-line common bus.

Ques. A digital computer has a common bus system for 16 registers of 32 bits each. The bus is constructed with MUX.

a) How many selection inputs are there in each MUX.

b) what size of MUX are needed.

c) How many MUX are there in bus.

K = 16, n = 32. $16 = 2^{\text{no. of select lines}}$

b) Size of MUX = 16 x 1

c) no. of MUX = 32.

a) $\log_2 16 = \log_2 2^4 = 4 \log_2 2 = 4$

Tue M3, DLD
Wed DM, DS
Thu COA, EVS

Date: 3/9/20
Page:

Three 3-state Bus Buffers.

Three state

→ 0

→ 1

→ high impedence state (behaves like open circuit & has no logical significance)

output is same as input.

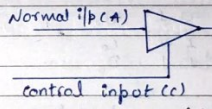


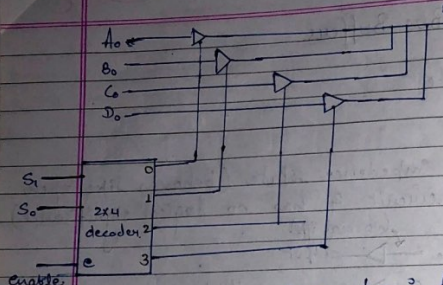
Fig: graphical symbol of 3-state buffer.

A bus system can be constructed with three state gates.

A three-state is a digital circuit that exhibits 3 states. The two states are signals equivalent to logic 1 and 0. And the third state is a high-impedence state.

The high-impedence state behaves like an open circuit which means that output is dis connected and doesn't have logical significance. Most commonly used in the design of Bus system is the buffer-gate.

8 reg - 16 bit each
no. of circuit = 16 = n
no. of buffer gate = 8x8, 8x16 = 128



e = 0, bus in high-impedance state

e = 1,

select	
S_1	S_0
0	0
0	1
1	0
1	1

Note: To construct a common bus for 4-registers of n-bits each. using 3 state buffers we need n-circuits with 4 buffer in each. Each group of 4 buffers receive 1 significant bit from 4 registers, each common o/p produces one of the lines for the common bus for total of n-lines. Only 1 decoder is necessary to select between the 4 registers.

Memory Transfer.

The transfer of information from memory world to outside environment is called a read operation.

read : DR ← M[AR]

DR: Data reg.; M[AR]: pointing data at AR register; AR: address reg.

AR ← takes only address
M[AR] ← take data from add. in AR

The transfer of new information to be stored in memory is called write operation.

write : M[AR] ← R_1

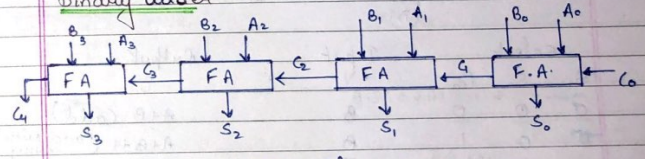
R_1: Processor reg.; write at add. pointed by AR.

4/9/18
(i) Register transfer micro-operations
(ii) Arithmetic micro-operation → add → ++, → sub → --
(iii) shift micro-operation.
(iv) logic micro-operation. (AND, OR, NOT, Ex-OR --)

ARITHEMATIC micro-operation

Arithematic micro-op^n perform arithematic op^n on numeric data stored in registers.

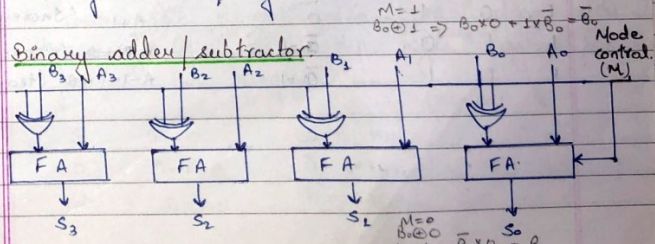
Binary adder

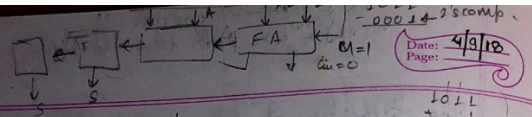


4-bit adder.

no. of full adders required depends on number of bit of addend / augend.

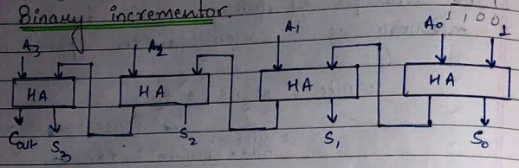
Binary adder / subtractor



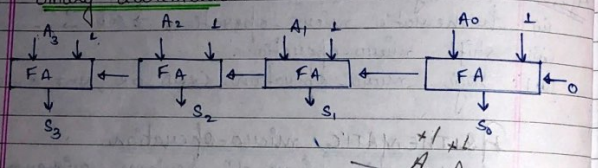


Date: 4/9/18
Page:

Binary incrementor.



Binary decrementor.



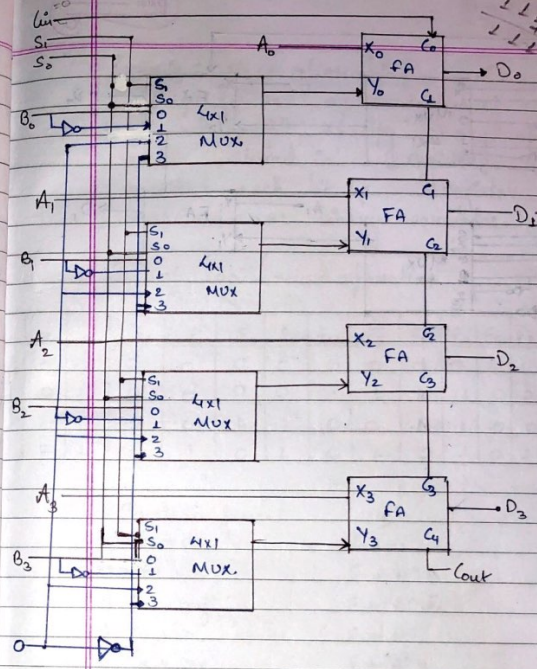
*1 *1
A1 A0

Arithematic circuit.

Output = A + Y + Cin
(D)

Select		Input	Output D
S1	S0	Cin (select Mux 0 i.e. B)	
0	0	0	B — A+B (add)
0	0	1	B — A+B+1 (add with carry)
0	1	0	$\bar{B}$ — A+$\bar{B}$ (sub with borrow)
0	1	1	$\bar{B}$ — A+$\bar{B}$+1 (subtract)
1	0	0	0 — A (transfer)
1	0	1	0 — A+1 (Incrementor)
1	1	0	$\bar{0}$ = 1 — A−1 (decrementor)
1	1	1	$\bar{0}$ = 1 — A−1+1 = A (transfer)

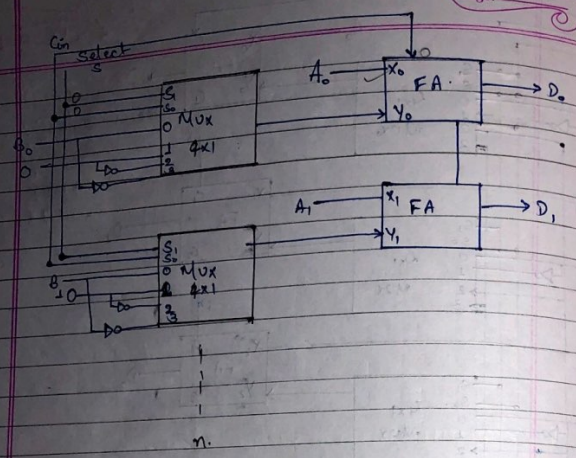
Mux 1 → $\bar{B}$
Mux 2 → 0
Mux 3 → $\bar{0}$ = 1
like decrementor
Hence



Ques: Design an arithematic circuit with one selection variable S and two n-bit data inputs (A & B). The circuit generates the following 4 arithematic operation in conjunction with i/p carry Cin. Draw the logic diagram for first two stages.

MUX = 2x1

S	Cin = 0	Cin = 1
0	D = A+B	D = A+1
1	D = A−1	D = A+$\bar{B}$+1



S	C_{in}	Input.	o/p.
0	0		A+B
0	1		A+1
1	0		A−1
1	1		A+$\bar{B}$+1

S_1 S_0

$A+C_{in}$

V → OR
∧ → AND
⊕ → EXOR,
− → NOT

Logic Micro-operation

performed on contents of registers

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

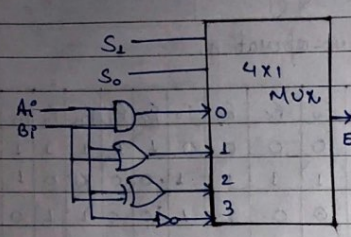
List of logic micro-operation.

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

	Boolean func.	op^n	Name
1)	$F_0 = 0$	$F \leftarrow 0$	clear
2)	$F_1 = xy$	$F \leftarrow A \wedge B$	AND
3)	$F_2 = xy'$	$F \leftarrow A \wedge \bar{B}$	
4)	$F_3 = x$ ($xy' + xy$, $x(y'+y)$)	$F \leftarrow A$	Transfer A.
5)	$F_4 = \bar{x}y$	$F \leftarrow \bar{A} \wedge B$	
6)	$F_5 = y$	$F \leftarrow B$	Transfer B
7)	$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Ex-OR
8)	$F_7 = x + y$	$F \leftarrow A \vee B$	OR
9)	$F_8 = (x+y)'$	$F \leftarrow \overline{A \vee B}$	NOR
10)	$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Ex-NOR
11)	$F_{10} = y'$	$F \leftarrow \bar{B}$	Complement B
12)	$F_{11} = x + y'$	$F \leftarrow A \vee \bar{B}$	
13)	$F_{12} = x'$	$F \leftarrow \bar{A}$	Comp. A.
14)	$F_{13} = x' + y$	$F \leftarrow \bar{A} \vee B$	
15)	$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
16)	$F_{15} = 1$	$F \leftarrow$ all 1's	set to all 1's

Hardware Implementation.

Logic gates is inserted for each bit or pair of bits in registers to perform the required logic functions.



S_1	S_0	O/P
0	0	AND
0	1	OR
1	0	XOR
1	1	NOT ($\bar{A}$)

∴ Functional Table

Applications of logic micro-operation.

Logic micro-op^n are used for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values., delete a group of bits or insert new bit values into a register.

→ Register A is a processor register and the bits of register B constitute logic operand extracted from memory & placed in register B.

(i) Selective set. used OR micro-op^n.

```
1010    A before
1100    B (logic operand)
1110    A after.
```

Sets to 1, the bits in register A where there are corresponding 1s in register B.

(ii) Selective compliment.

```
1010    A before.
1100    B (logic operand)
0110    A after.
```

Selective compliment uses Ex-OR micro-op^n.

(iii) Selective clear.

```
1010
1100
0010    A after.
```

Selective clear uses $A\bar{B}$ micro-op^n

(iv) Mask. It is similar to selective clear except that bits of A are cleared only where there are corresponding 0's in B.

```
1010
1100        A∧B = AND op^n.
1000
```

(v) Insert. inserts a new value in group of bits.

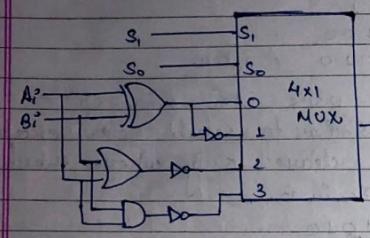
```
1001  insert  0110   1010    A before
AND   [ 0000   1111    B  (for masking)
      [ 0000   1010
OR    [ 1001   0000  } insert op^n
        1001   1010
```

(vi) Clear. (Ex-OR)

```
1010    A
1010    B (logic operand)
0000
```

when we have two words same i.e. A = B then only we perform clear.

Ques: Design a digital ckt that performs the 4 logic operations of XOR, XNOR, NOR and NAND. Use 2 selection variables. Show logic diag. of 1 typical stage.



S_1	S_0	o/p
0	0	XOR
0	1	XNOR
1	0	NOR
1	1	NAND

7/9/18

Shift Micro-operation.

are used for Serial Transfer of data.

1010

left 010 ← This value is transferred by serial input. (It is a flip-flop)

Types of shift :

(i) Logical → left (shl), right (shr)

(ii) Circular → left (cil), right (cir)

(iii) Arithmetic → left (ashl), right (ashr)] used for signed nos. (-ve nos. are in 2's comp. for) (we copy sign bit as it is.)

7/9/18

R = 0010

shl 0100]
shr 0001] when we apply logical shift then, Serial transfer 0 to vacant bit.
cil 0100
cir 0001

when we apply ashr once then that number is divided by 2., twice divided by 2^2 so on.
when we apply ashl, once number is multiplied by 2. twice, number is ~~divided~~ multiplied by 2^2.

R = 01100 = +4 (signed no.)

ashr 00010 (preserve sign bit)
↳ = +2

R = 00100 = +4

ashl 01000 = +8
↳ always insert zero

Overflow can come in ashl.
If sign bit and next bit are different, then there is overflow.

0100 = +4
ashl 1000 can't be -ve.
↳ sign bit = 1 (-ve no.)

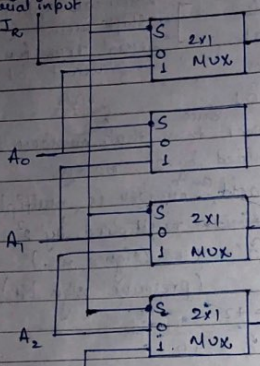
(when we multiply 2 in number we can't get -ve number.)
when we see, that overflow can come then after performing ashl we can perform sign reversal.

Hardware Implementation.

Data inputs. A_0 A_1 A_2 A_3.
Output. H_0 H_1 H_2 H_3.

Select
S = 0 for right shift
1 for left "

Serial input I_R



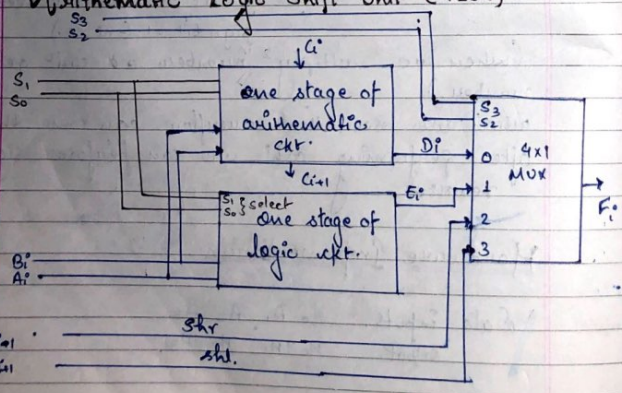
(4-bit shifter)

A_0 A_1 A_2 A_3
I_R A_0 A_1 A_2

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

10/09/18

Arithematic Logic Shift Unit (ALSU)



Date: 10/9/18
Page:

OPR select. S_3	S_2	S_1	S_0	C_i	operation.
0	0	0	0	0	F = A (transfer)
0	0	0	0	1	F = A+1 (increment)
0	0	0	1	0	F = A+B (add)
0	0	0	1	1	F = A+B+1 (add with carry)
0	0	1	0	0	F = A+$\bar{B}$ (sub. with borrow)
0	0	1	0	1	F = A+$\bar{B}$+1 (sub).
0	0	1	1	0	F = A-1 (decrement)
0	0	1	1	1	F = A (transfer.)
0	1	0	0	x	F = A^B (AND)
0	1	0	1	x	F = A$^{\vee}$B (OR)
0	1	1	0	x	F = A⊕B (XOR)
0	1	1	1	x	F = $\bar{A}$ (NOT)
1	0	x	x	x	F = shr (shift right)
1	1	x	x	x	F = shl (shift left)

Ques: Register A holds the 8-bit binary number. 11011001 determine the B operand & logic micro-op^n to be performed in order to change the value in A to
a) 01101101
b) 11111101

11011001
011

Ques: The 8-bit registers AR, BR, CR, DR initially have the following values -
AR = 11110010
BR = 11111111
CR = 10111001
DR = 11010110
Determine 8 bit values in each register after execution of following sequence of micro-op^n.

AR ← AR + BR
CR ← CR ^ DR , BR ← BR + 1
AR ← AR - CR.

Ques: An 8-bit register contains the binary value 10011100 what is register value after. ashr. and ashl. state if overflow or not.

Ques: Starting from an initial value of R = 11011101 Determine the sequence of binary value in R after a logical shift left, followed cir. (circular right), followed by shr. and cil.

R = 1 1 0 1 1 1 0 1.
shl. 1 0 1 1 1 0 1 0
cir. 0 1 0 1 1 1 0 1
shr. 0 0 1 0 1 1 1 0
cil. 0 1 0 1 1 1 0 0 Ans.
R = 0 1 0 1 1 1 0 0

Ans 2:
R = 1 0 0 1 1 1 0 0
ashr. 1 1 0 0 1 1 1 0
ashl. 0 0 1 1 1 0 0 0 (overflow)

Ans 1 (i) A = 1 1 0 1 1 0 0 1
B = ? 1 0 1 1 0 1 0 0 (selective compliment)
A = 0 1 1 0 1 1 0 1

(ii) A = 1 1 0 1 1 0 0 1
B = ? 1 1 1 1 1 1 0 1 (selective set)
A = 1 1 1 1 1 1 0 1

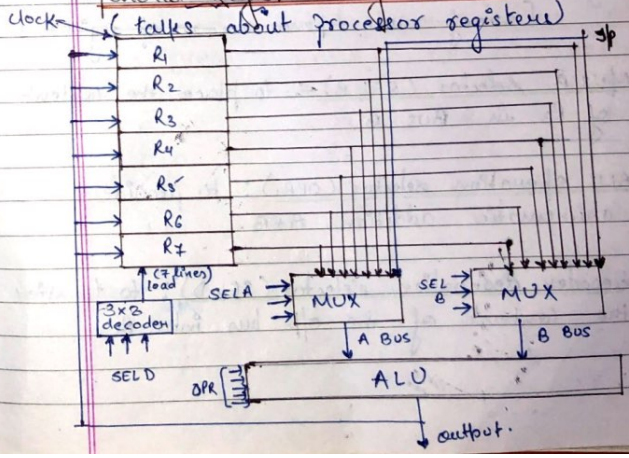
Date: 11/9/10
(ch-8)

Central Processing Unit (CPU)

ALU → perform operation on registers.
It has select opR. which selects the op" to be performed.
CU → supervise the instruction.
(control unit)
Registers set → store data.

→ The CPU is made up of 3 major parts:
(i) register set: stores intermediate data during execution of instruction.
(ii) ALU: performs the required micro-op" for executing the instructions.
(iii) control unit: supervise the transfer of information among the registers & instruct the ALU to which operation to be performed.

General Register Organisations.
(talks about processor registers)



Block diagram of CPU registers

SEL A	SEL B	SEL D	OPR
3bit	3	3	6

control word

↓

Specifies the micro-opn to be performed.

SelA = 5
SelB = 5
decoder = 5
selD
OPR = 7

If,
← 32 processor register.
ALU performs 65 opn

For ex. to perform the operation i.e. $R_1 \leftarrow R_2 + R_3$. the control must provide binary selection variables to the following selector inputs

1) MUX A selector (SEL A) : to place the content of R_2 in BUS A (no. lines depends on no. of bits in each registers.

~~[no. of mux depend on no. of register.]~~

2) MUX B selector (SEL B) : to place the content of R_3 in BUS B.

3) ALU operation selector (OPR) : to provide arithematic addition A+B.

4) Decoder destination selector (SEL D) : to transfer the content of the o/p bus into R_1

control word.

There are 14 binary selection i/p in the unit & their combined value specifies the control word.

Ques. A Bus organized CPU has 16 registers with 32 bits in each, and ALU and a destination decoder.

a) How many MUX are there in A bus. 16 32 and what is size of each mux. 16×1

b) How many selection i/p are needed for MUX A and MUX B. 4

c) How many i/p and o/p are there in decoder. 4×16

d) How many i/p and o/p are there in ALU for data including i/p & o/p carry's. $32+32 = 64+1 = 65$

o/p lines = 33

e) Formulate a control word for the system assuming that ALU has 35 opn.

4 + 4 + 4 + 6 = 18
A B D OPR

12/9/18

Encoding of register selection fields.

Binary code	SEL A	SEL B	SEL D
	Input	Input	None
000			
001	R_1	R_1	R_1
010	R_2	R_2	R_2
⋮			
111	R_7	R_7	R_7

Date: 12/9/10
Page:

Encoding of ALU operations.

OPR select	operation.	symbol.
00000	Transfer A	TSF A
00001	increment A	INC A
00010	add A+B	ADD
00011	sub A-B	SUB
00110	decrement A	DEC A
01000	AND A∧B	AND
01010	OR	OR
01100	XOR	XOR
01110	complement A	COM A
10000	shift right A	SHRA
11000	shift left A	SHLA

Ex. $R_1 \leftarrow R_2 - R_3$ (micro-op^n to control word)

fields:	SEL A	SEL B	SEL D	OPR
symbol:	R_2	R_3	R_1	SUB
control word:	010	011	001	00101

Given cw: 010 011 001 00101.
(SEL A: R_2, B: R_3, D: R_1, OPR: SUB.)
} control word to micro-op^n
$R_1 \leftarrow R_2 - R_3$

Ques.

1) $R_6 \leftarrow R_6 + 1$ — ~~SELA 011 011 00001~~
2) $R_7 \leftarrow R_1$ — 111 001.
3) output $\leftarrow R_2$
4) output $\leftarrow$ input.

1)

SEL A	SEL B	SEL D	OPR
~~110~~ R_6	—	~~110~~ R_6	INC A
110	—	110	00001

control word: 110 — 110 00001

2)

SEL A	SEL B	SEL D	OPR
R_1	—	R_7	TSF A
001	—	111	00000

3)

SEL A	SEL B	SEL D	OPR
R_2	—	000	TSF A

control word: 010 _ 00000000

4)

SEL A	SEL B	SEL D	OPR
~~R7~~ 000	—	000	TSF A

control word: 000 _ 00000000

Stack Organisation

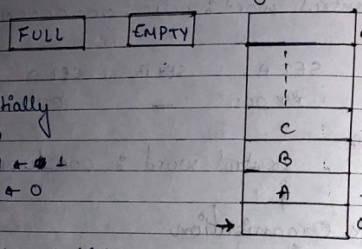
Stack pointer is register that points top of stack.
insert ← DR to stack
pop ← stack to DR. (data register)
flip-flop [full = 1 (stack is full)
empty = 1 (stack is empty)

→ The stack in a digital computer is a memory unit with an address register that can count only.
→ The register that holds the address for stack is called stack pointer ~~(SP)~~ (SP) because its value always point at the top item in stack.

(ii) **Register Stack.**

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

(Block diag. of 64 word stack)



Initially
SP = 0
EMTY ← 1
FULL ← 0

SP → 111111 (63)
Trying to insert
then
SP = 111111
+ 1
1 000000
discard

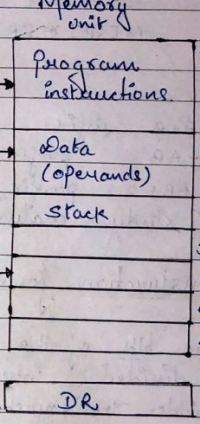
means after 63 it will come to 0
then
EMTY ← 0 (when we come to 0, before that there should be 63 address)
FULL ← 1.

Push:
SP ← SP+1 (increment stack pointer)
M[SP] ← DR (write item on top of stack)
if (SP==0) then (FULL ← 1)
EMTY ← 0

pop:
DR ← M[SP]
SP ← SP-1
if (SP==0) then (EMTY ← 1)
FULL ← 0
(when we come to 0 address, before that there should be 1 address)

19/9/18

(iii) **MEMORY STACK**



→ A stack can exist as a standalone unit or can be implemented in random access memory (RAM) attached to CPU.

→ The portion of computer memory is partitioned into 3 segments.
1st partition – program
2nd " – data
3rd " – stacks.

→ The program counter PC points at the address of next instruction in program.

→ The address register AR points at an array of data.

→ The stack pointer SP points at the 'top' of stack.

→ PC is used during the fetch phase to read the instruction. AR is used during execution phase

Date: 13/9/18
Page:

to read an operand. SP is used to push or pop items into or from the stack.

pop: (we check upper and lower limit by hardwares)
DR ← M[SP]
SP ← SP+1

PUSH:
SP ← SP-1
M[SP] ← DR

A+B infix
+AB prefix or polish
AB+ postfix or reverse polish.
(study evaluation of postfix, infix → postfix)

Instruction format.

4096 x 16 (16 bit)
0-4095 ←

→ The bits of the instructions are divided into groups called fields. The most common fields.
2^12 address = 12bit (memory add / reg add)
found in instruction format are -

1) An operation code field (op code) : that specifies the op^n to be performed.
2) An address field : that designates a memory address or a processor register.
3) A mode field : that specifies the way the operand or the effective address is determined.

Operations specified by computer instructions are executed on some data stored in

Date:
Page:

memory or processor registers. operands residing in memory are specified by the memory address & operands residing in processor register are specified by register address.

14/9/18

Most of the computers are classified into one of 3 types of CPU organisations -

1) Single accumulator type organisations :-
All the operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field.
Ex: ADD X := AC ← AC + M[X]

content of AC & content at address X will be added.

2) General register organisation
consider it destination register.
Three-address instruction: ADD (R1), R2, R3 — R1 ← R2+R3

Two address instruction:
ADD R1, R2 — R1 ← R1+R2
MOV R1, R2 — R1 ← R2
ADD R1, X — R1 ← R1 + M[X]

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

3) Stack organisation.
Stack organisation uses (zero) no address field.

Eg:
PUSH A	TOS ← A
PUSH B	TOS ← B
ADD	TOS ← (A+B)

(0 address.)

Ex X = (A+B) * (C+D)
We assume that operands are in memory addresses A, B & C and the result is stored in memory at the address X.

4) Following expression in 3-address instructions.

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

ADD A, A, B	M[A] ← M[A] + M[B]

(So, this is not wrong, but by doing this we have to access memory many time & (memory transfer will be(slow, so we use a temporary storage for fast transfer)

2-address instructions.

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

1-address instructions.

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

Zero-address instructions.
Here, expression is first converted to postfix form.

X = (A+B) * (C+D)
AB+ CD+ *

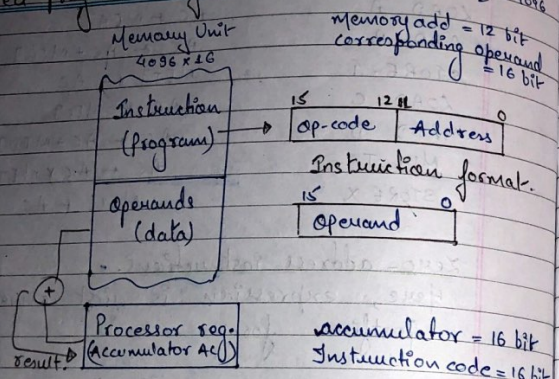
PUSH A	TOS ← A
PUSH B	TOS ← B
ADD	TOS ← (A+B)
PUSH C	TOS ← C
PUSH D	TOS ← D
ADD	TOS ← (C+D)
MUL	TOS ← (C+D) * (A+B)
POP	M[X] ← TOS

Ques $X = \frac{(A-B) + C * (D * E - F)}{G + H * K}$

Write for 3, 2, 1, 0.

Date: 20/09/18
Page:

Stored Program Organisation



$2^{12} = 4096$

memory add = 12 bit
corresponding operand = 16 bit

accumulator = 16 bit
Instruction code = 16 bit

To execute any instruction in memory,
instruction code = 16 bit

memory address (op^n on operand) — op-code (type of op^n)

performs operation with accumulator & stores result in accumulator.

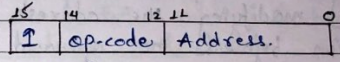
If we don't want operand from memory address then we don't specify memory address part in instruction format.

→ If an operation in an instruction code doesn't need an operand from memory rest of the bits in instruction can be used for the other purposes.

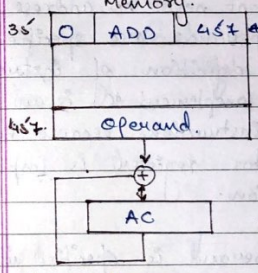
Date:
Page:

For example, operation such as clear AC, complement AC & increment AC operate on data stored in AC.

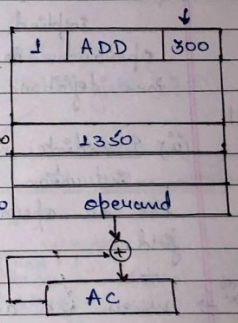
Direct and Indirect



I = 0 ⇒ direct
I = 1 ⇒ indirect.



In direct, memory is referred only once while, in indirect it is referred twice.



EA = 1350

when 2nd part specifies the operand then it is called immediate.

ADDRESSING MODES

The way the operands are chosen during program execution dependent on the addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

→ There are 9 modes that need no address field

(i) Implied mode: operands are specified implicitly in the definition of instruction. For ex. instruction complement AC is an implied mode instruction because the operand in accumulator register is implied in the definition of instruction.

(ii) Immediate mode: operand is specified in the instruction itself. An immediate mode instruction has an operand field rather than an address field.

22/9/18

→ when the address field specifies a processor register, the instruction is said to be in Register mode. (operand in processor register)

* value in address field is operand itself

→ Register indirect mode.: Instruction specifies a register in the CPU whose content give the address of the operand in memory.

add: 400
auto increment (400) = x+1
auto decre. (399) = y.

address decrement

→ Auto increment or auto decrement: It is similar to register indirect mode except that the register is incremented or decremented after or before its value is used to access memory.

increment / decrement

respect to memory

→ Direct address mode.

EA = address part of instruction.

Indirect address mode.

address field of instruction gives the address where effective address is stored in memory.

→ A few addressing mode require that the address part of instruction is added to the content of a specific register in the CPU.

EA = address part of instruction + content of CPU register. } ③

Base register (Base register addressing mode)

Index register. (index addressing mode)

Program counter (PC) relative addressing mode.

Ques.

PC = 200

R1 = 400

XR = 100 (index register)

AC

address	memory
	load to AC / Mode
200	Address = 500
201	Next instruct.
202	
399	400
400	700
500	800
600	900
702	325
800	300

2 word address

The two word instruction at address 200 and 201 is 'load to AC' instruction with an address field = 500. The first word of instruction specifies the op-code and mode. and 2nd word specifies the address part. AC receives the operand after instruction is executed.

i) immediate -
operand = 500
EA = 201

ii) Register mode - → auto decrement
operand = 400 EA = 399
EA = x op. = 400

iii) Register indirect. → Auto increment.
operand = 700 operand = 700
EA = 400 EA = 400

iv) direct address mode.
EA = address part of instruction = 500
operand = 800

v) Indirect address mode :
EA = 800
operand = 300

vi) Relative address mode.
EA = 500 + 202 = 702
↑ next instruction
operand = 325

vii) Index addressing mode
EA = 500 + 400 = 600 (content of XR)
operand = 900

Qus. An instruction is stored at location 300 with its address field at loc. 301. The address field has value 400. A processor register R_1 contains the number 200. Evaluate the EA. if the addressing mode of instruction is -
a) Direct
b) Relative
c) Immediate
d) Register indirect
e) index with R1 as index register.

$R_1 = 200$

300	
301	add. = 400
302	next instruct.

a) direct.
EA = 400

b) Relative.
EA = 400 + 302 = 702

c) Immediate
EA = 301.

d) Register indirect -
EA = 200

e) index mode
EA = 400 + 200 = 600

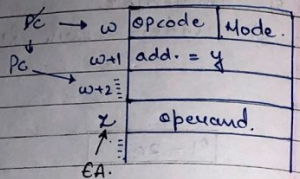
25/9/18

Qus. A two-word instruction is stored in memory at an address designated by the symbol w. The address field of instruction stored at w+1 is designated by symbol y. The operand used during the execution of instruction is stored at an address symbolized by x. An index register contains the value x. state how x is calculated from other addresses if the addressing mode of instruction is.

a) Direct (c) Relative
b) Indirect (d) Indexed

XR = x

x = Eff. address.



a) Direct -
EA = y
x = y

b) Indirect.
x = M[y]

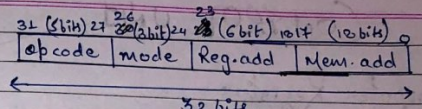
c) Relative.
x = y + ω + 2

d) Indexed
x = ~~400~~ + ~~xx~~ y + x

mem-add = 18 bit → 1024

Ques: The memory unit of a computer has 256K words of 32 bits each. The computer has an instruction format with 4 fields - 1) an operation code field, 2) a mode field, (to specify one of 7 addressing modes.), 3) a register address field to specify one of 60 processor registers and a memory address. specify the instruction format and the number of bits in each field. if the instruction is in one memory word.

memory add = 18 bits.
mode = ~~7bit~~ 3 bits. = $2^3 = 8 \geq 7$
Register add = 6 = $2^6 = 64 \geq 60$
op-code = 5 bits (remaining bits.)



Ques: A computer has 32 bit instructions and 12 bit addresses. If there are 250 two-address instructions, how many one-address instructions can be formulated = 6×2^{12}

2-address
- op-code = 8 bit (32 - 24 = 8)
- address 1 = 12 bit
- address 2 = 12 bit.

max- op^n = 2^8 = 256

250 - 2 address instruction

(256 - 250) = 6 op^n - 1 address instruction.
(op code, address 1. (12 bit))

1 op-code = 2^{12}
2 op-code = 2^{12}
⋮
6 op-code = 6×2^{12}
Total = 6×2^{12}

Two-address instruction

8 bit	12 bit	12 bit
Op-code	Add. 1	Addr. 2

← 12 →

op-code	address