Arithmetic

Unit II -Chapter 4

Representation of Signed integer

- There are three different schemes to represent negative number:
  - Signed-Magnitude form.
  - 1’s complement form.
  - 2’s complement form.

Signed magnitude form

- For an n-bit number, one bit (generally, the MSB) is used to indicate the sign of the number and remaining (n-1) bits are used to represent magnitude.

- Therefore, the range is from : $-2^{n-1} – 1$ to $+2^{n-1} – 1$

- 0 in sign bit indicates positive number and 1 in sign bit indicates negative number.

- For example, 01011001 represents + 169 and

  11011001 represents - 169

- What is 00000000 and 10000000 in signed magnitude form?
  - +0 and -0

Representation of Signed integer in 1's complement form

- Consider the eight bit number 01011100,
- 1's complements of this number is 10100011.
- If we perform the following addition:

0 1 0 1 1 1 0 0

1 0 1 0 0 0 1 1

----

1 1 1 1 1 1 1 1 ... this is -0 (negative zero) in 1’s complement.

Representation of Signed integer in 1's complement form

- Since the addition of two number is 0, so one can be treated as the negative of the other number.

- So, 1's complement can be used to represent negative number.

Representation of Signed integer in 2's complement form

- Consider the eight bit number 01011100,
- 2's complements of this number is 10100100.
- If we perform the following addition:

```
   0 1 0 1 1 1 0 0
   1 0 1 0 0 0 1 1
   ----------------------
 1 0 0 0 0 0 0 0 0
   ----------------------
```

- In 2’s complement addition the carry bit is discarded.

Representation of Signed integer in 2's complement form

- Therefore, the final result is 00000000.
- Since the addition of two number is 0, so one can be treated as the negative of the other number.
- So, 2's complement can be used to represent negative number.

4-bit numbers in 1's, 2's and signed magnitude form

Decimal	2's Complement	1's complement	Signed Magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	-----	1111	1000
-1	1111	1110	1001
-2	1110	1101	1010
-3	1101	1100	1011
-4	1100	1011	1100
-5	1011	1010	1101
-6	1010	1001	1110
-7	1001	1000	1111
-8	1000	------	-------

Overflow in Integer arithmetic

- Occurs when the result of an arithmetic operation is outside the representable range

- Range for n bit 2's complement number

$$-2^{n-1} \text{ to } +2^{n-1} - 1$$

if n=4 then

range of values = -8 to +7

Overflow - examples

a) (+7) + (+4)

= +11

2's complement add:

0 1 1 1

0 1 0 0

= 1 0 1 1 = -5

No carry out

b) (-4) +(-6)

= -10

2's complement add:

1 1 0 0

1 0 1 0

=1 $\leftarrow$ 0 1 1 0 = +6

Carry out

Overflow - examples

c) (+7) + (-3)

= +4

2's complement add:

0 1 1 1

1 1 0 1

= 1 ← 0 1 0 0 = +4

Carry out

Overflow - Contd

- Overflow occurs when adding two nos. that have **SAME SIGN**

- Carry out signal from Sign bit position is **not a Sufficient indicator** of overflow

To Detect Overflow

- Examine the sign bit of 2 nos, X and Y.
- Examine the sign bit of result, S.
- Overflow occurs if X and Y has same sign and S has different sign.

1’s complement vs 2s complement

- The primary advantage of two's complement over one's complement is that two's complement has only one value for zero.
- One's complement has a "positive" zero and a "negative" zero
- Next, to add numbers using one's complement you have to first do binary addition, then add in an end-around carry value (if any).
- In 2’s complement addition the end-around carry value (if any) will be discarded .
- This is the reason why 2' s complement is generally used.

2's complement arithmetic – Example 1

- Perform (-8) + (+5)

+8 = 01000

Note that a positive number in 2's complement system has 0 in the MSB and a –ve number has 1 in the MSB

Take 2's complement of +8 to get -8

1's Complement	10111
Add 1 to get 2s complement	1
	11000

So, -8 =11000

+5 = 0101

So, (-8) + (5)

 11000

 <u>00101</u>

<u> 11101 </u>…… Answer

Result is negative (i.e. MSB is 1)

How do you find out its value (magnitude)?

Ans: Take 2's complement of the –ve number

<u>11101 …. Answer</u>

00010 …. 1s complement

    1……add 1 to get 2's complement

<u>**00011**</u> ...... Which is +3.

Therefore the answer is -3

2's complement arithmetic – Example 2

- Perform (8) + (-5)

- **+8 = 01000**
- +5 =00101
- Note: 0101 is +5.
- 00101 is also +5
- Take 2's complement of +5 to get -5
- 11010 … 1's complement of +5
- 1 … add 1 to get 2's complement
- **11011 …. Which is -5**

- Perform (01000) + (11011)
- 01000
- <u>11011</u>
- <u>00011</u> … answer
- **Answer is +3**

2's complement arithmetic – Example 3

- (25) - (20)
- $25_{(10)} = 011001_{(2)}$
- $20_{(10)} = 010100_{(2)}$
- Subtract 20 means add 2's compl 20
- Take 2's complement of 20, which is -20
- 010100 …. +20
- 101011 …. 1s complement
-          1 .. Add 1 to get 2s comp
- 101100 …. -20
- (25) - (20)
- 011001
- 101100
- 000101 …. Which is +5

2 | 25
2 | 12 – 1
2 | 6 – 0
2 | 3 – 0
1 – 1

2 | 20
2 | 10 – 0
2 | 5 – 0
2 | 2 – 1
1 – 0

2's complement arithmetic – Example 4

- Perform (50) – (-20)
- Note: range of numbers that can be represented by n bit 2's complement number is -2^{n-1} to $+2^{n-1}-1$
- If n=8 the range is = -2^7 to $+2^7-1$ = -128 to +127
- If n=7 the range is -64 to +63
- If n=6 the range is -32 to +31.
- The answer of (50) – (-20) is 70, which requires 8 bits
- Hence represent the 50 and 20 also as 8 bit numbers
- 50= $00110010_{(2)}$
- – (-20) = 20 = $010100_{(2)}$ = $0010100_{(2)}$

- (50) – (-20) = 50+20
- 00110010
- 00010100
- 01000110 ... Answer is 70

2 | 50
2 | 25 – 0
2 | 12 – 1
2 | 6 – 0
2 | 3 – 0
1 – 1

Note: You can observe overflow when you consider 7 bit numbers:

0110010
0010100
1000110
Sign of the summands is same.
But sign of the sum is different.
Hence overflow.

2's complement arithmetic – Example 5

- Perform **$00110010_{(2)}$** – $1101100_{(2)}$
- Subtracting $1101100_{(2)}$ equivalent to adding 2's complement of $1101100_{(2)}$
- So, take 2's complement of 1101100
- 1101100 …given number

- 0010011 … 1's complement
- <u>   1</u> … add 1 to get 2s complement
- **<u>0010100</u>**

- Add the numbers
- 00110010
- <u>00010100</u>
- <u>01000110</u> …. Which is 70

2's complement arithmetic – Example 6

- Perform (30) + (-40)
- +30 =011110 or 0011110
- +40 = 0101000
- -40= 2's complement of +40
- 0101000 … +40
- 1010111 … 1s compl
- 1 .. .add 1 to 1s compl
- 1011000 … -40
- Add 30 to -40
- 0011110 … +30
- 1011000 … -40
- 1110110 …which is -10
- To know the magnitude of 1110110 take its 2's complement,
- which is 0001010, which is ten. So the answer is -10

2 | 30
2 | 15 - 0
2 | 7 - 1
2 | 3 - 1
1 - 1

2 | 40
2 | 20 - 0
2 | 10 - 0
2 | 5 - 0
2 | 2 - 1
1 - 0

Multiplication

Multiplication of unsigned numbers: Manual method

				1	1	0	1	(13) Multiplicand M
				1	0	1	1	(11) Multiplier Q
				1	1	0	1	
			1	1	0	1		
		0	0	0	0			
	1	1	0	1				
1	0	0	0	1	1	1	1	(143) Product P

Product of 2 *n*-bit numbers is at most a *2n*-bit number.

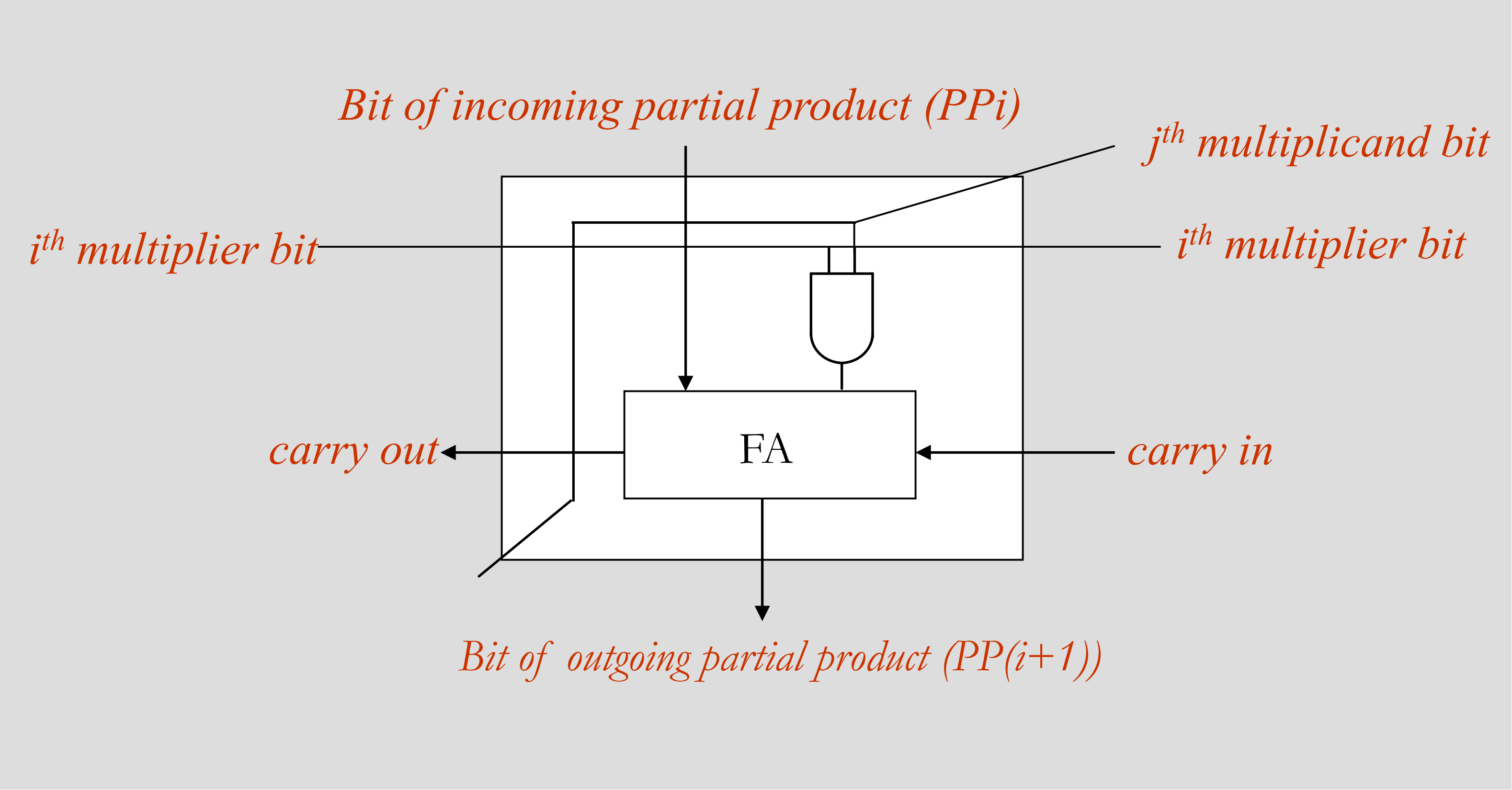
Unsigned multiplication can be viewed as addition of shifted versions of the multiplicand.

Multiplication of unsigned numbers (contd..)

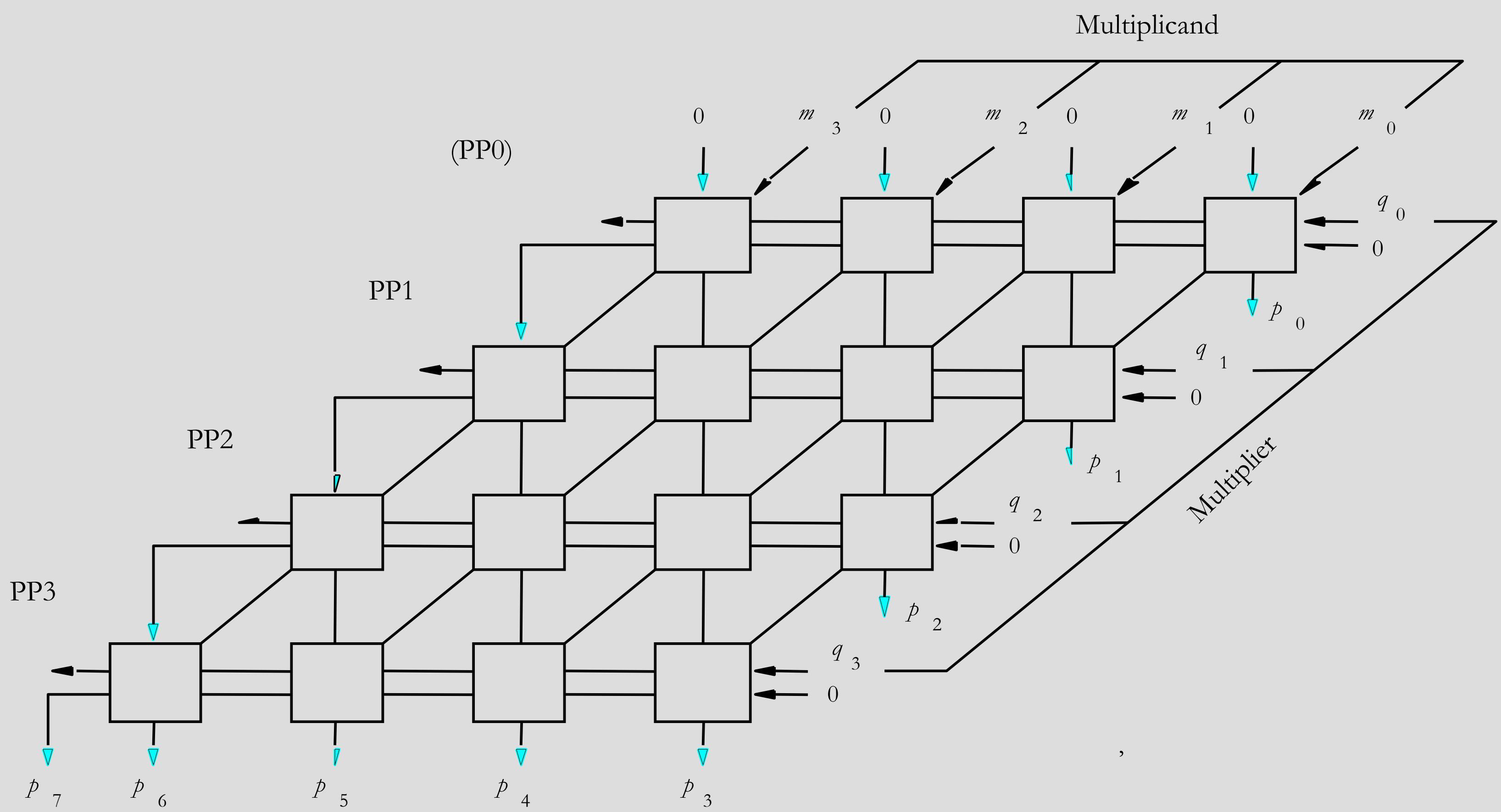
- We added the partial products at end.
  - Alternative would be to add the partial products at each stage.
- Rules to implement multiplication are:
  - If the i^{th} bit of the multiplier is 1, shift the multiplicand and add the shifted multiplicand to the current value of the partial product.
  - Hand over the partial product to the next stage
  - Value of the partial product at the start stage is 0.

Multiplication of unsigned numbers

Typical multiplication cell



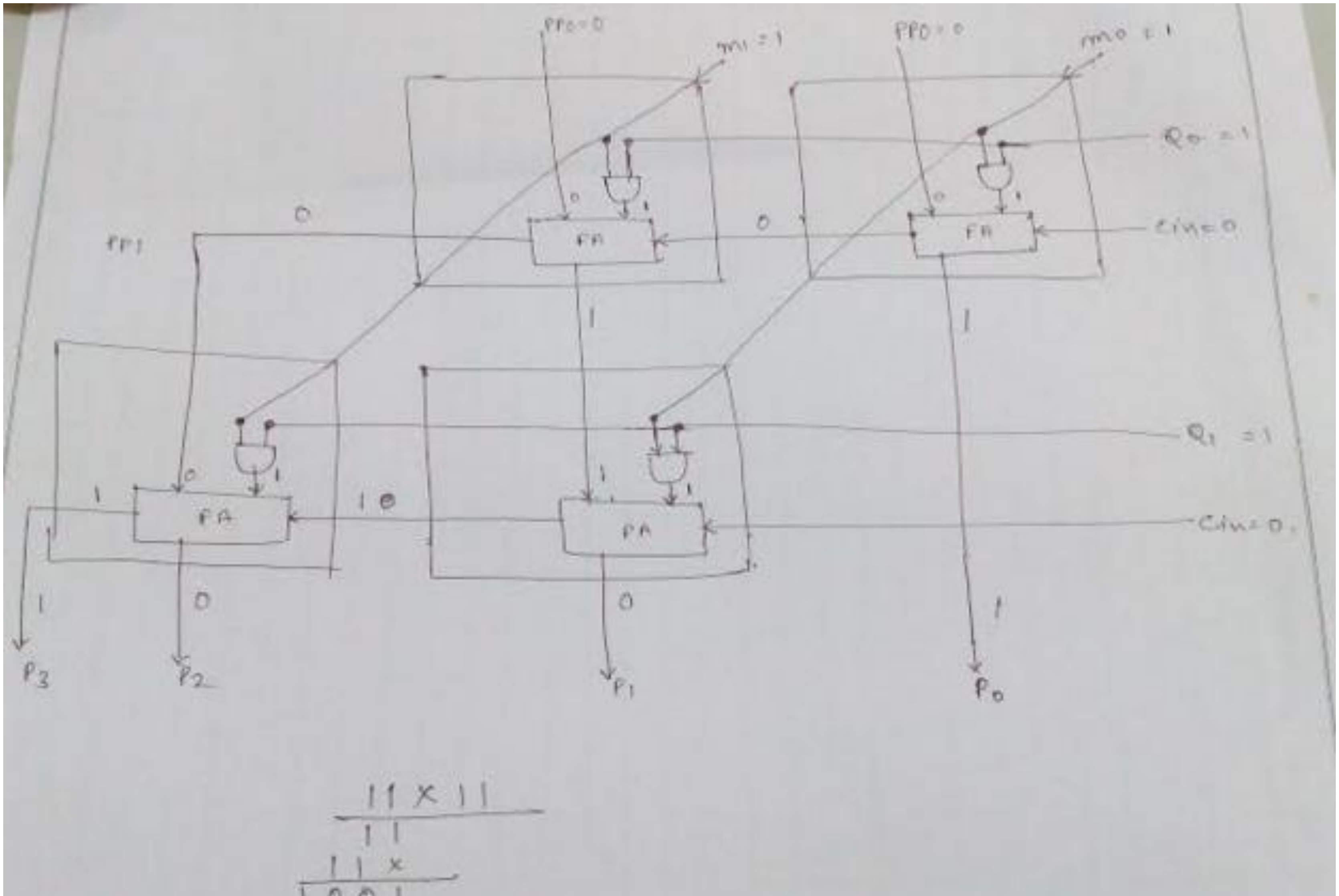
Combinational array multiplier



Product is: $p_7, p_6, ..p_0$

Multiplicand is shifted by displacing it through an array of adders.

Illustration of 11x11 on combinational array multiplier



Combinatorial array multiplier (contd..)

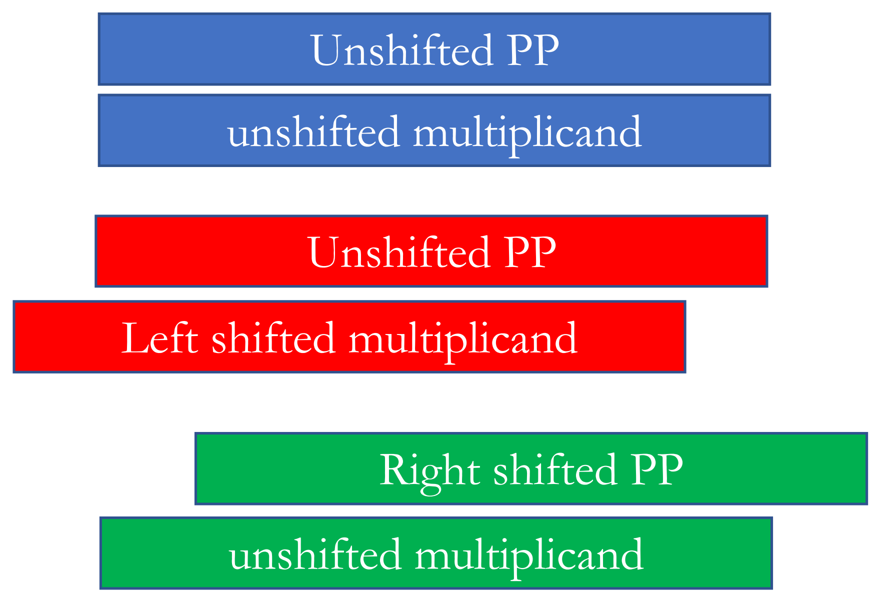
- Combinatorial array multipliers are:
  - Extremely inefficient.
  - Have a high gate count for multiplying numbers of practical size such as 32-bit or 64-bit numbers.
  - Perform only one function, namely, unsigned integer product.

- Improve gate efficiency by using a mixture of combinational array techniques and sequential techniques requiring less combinational logic.

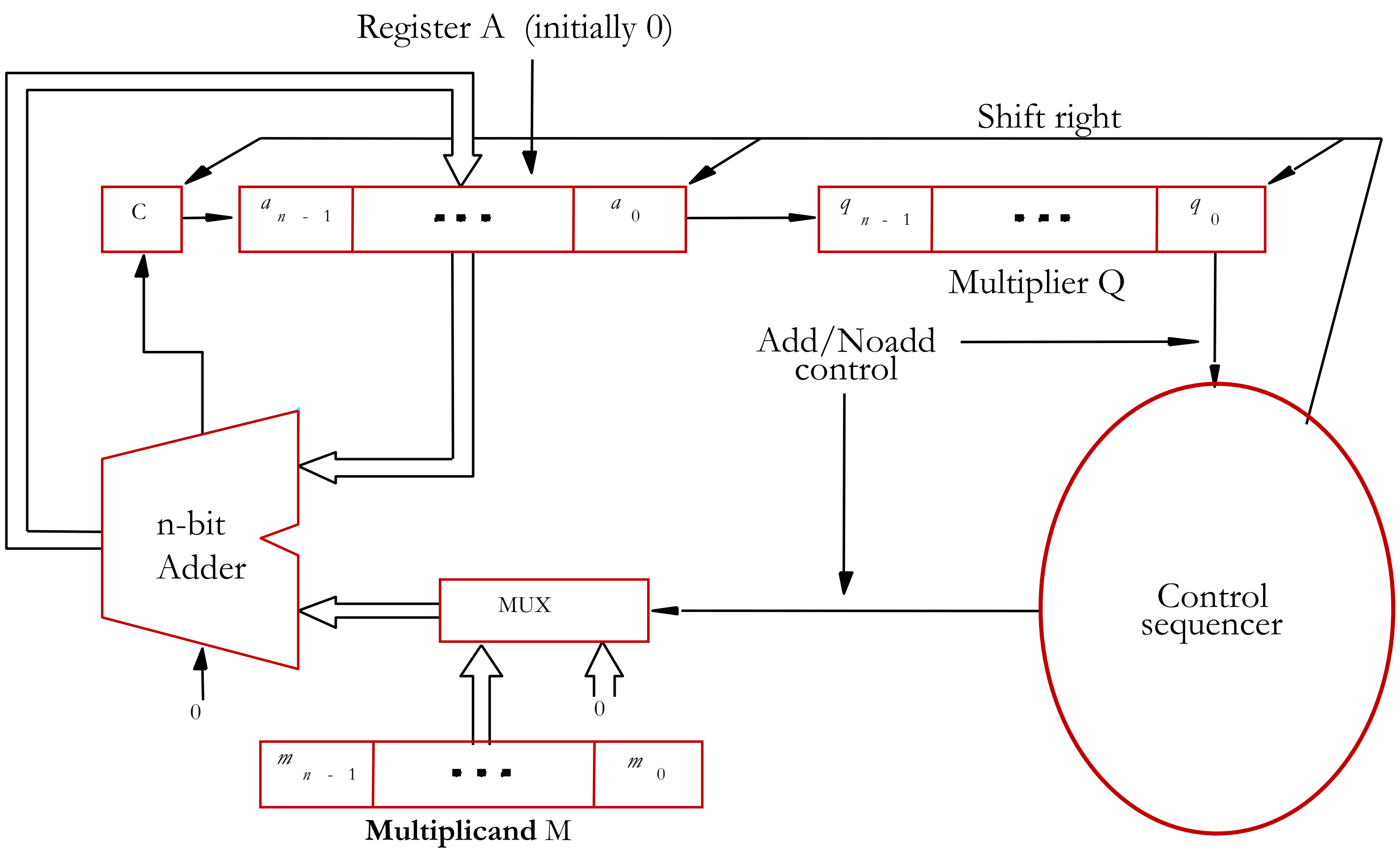
Sequential multiplication

- Recall the rule for generating partial products:
  - If the i^{th} bit of the multiplier is 1, add the appropriately shifted multiplicand to the current partial product.
  - Multiplicand has been shifted <u>left</u> when added to the partial product.

- However, adding a left-shifted multiplicand to an unshifted partial product is equivalent to adding an unshifted multiplicand to a right-shifted partial product.



Sequential Circuit Multiplier



Sequential multiplication- example 1

	C	A	Q		
M		1 1 0 1			Initial configuration
	0	0 0 0 0	1 0 1 1		
	0	1 1 0 1	1 0 1 1	Add	First cycle
	0	0 1 1 0	1 1 0 1	Shift	
	1	0 0 1 1	1 1 0 1	Add	Second cycle
	0	1 0 0 1	1 1 1 0	Shift	
	0	1 0 0 1	1 1 1 0	No add	Third cycle
	0	0 1 0 0	1 1 1 1	Shift	
	1	0 0 0 1	1 1 1 1	Add	Fourth cycle
	0	1 0 0 0	1 1 1 1	Shift	

Product

Sequential multiplication-
example 5

Addition In step-1

0000

<u>1101</u>

1101

in step-2

0110

<u>1101</u>

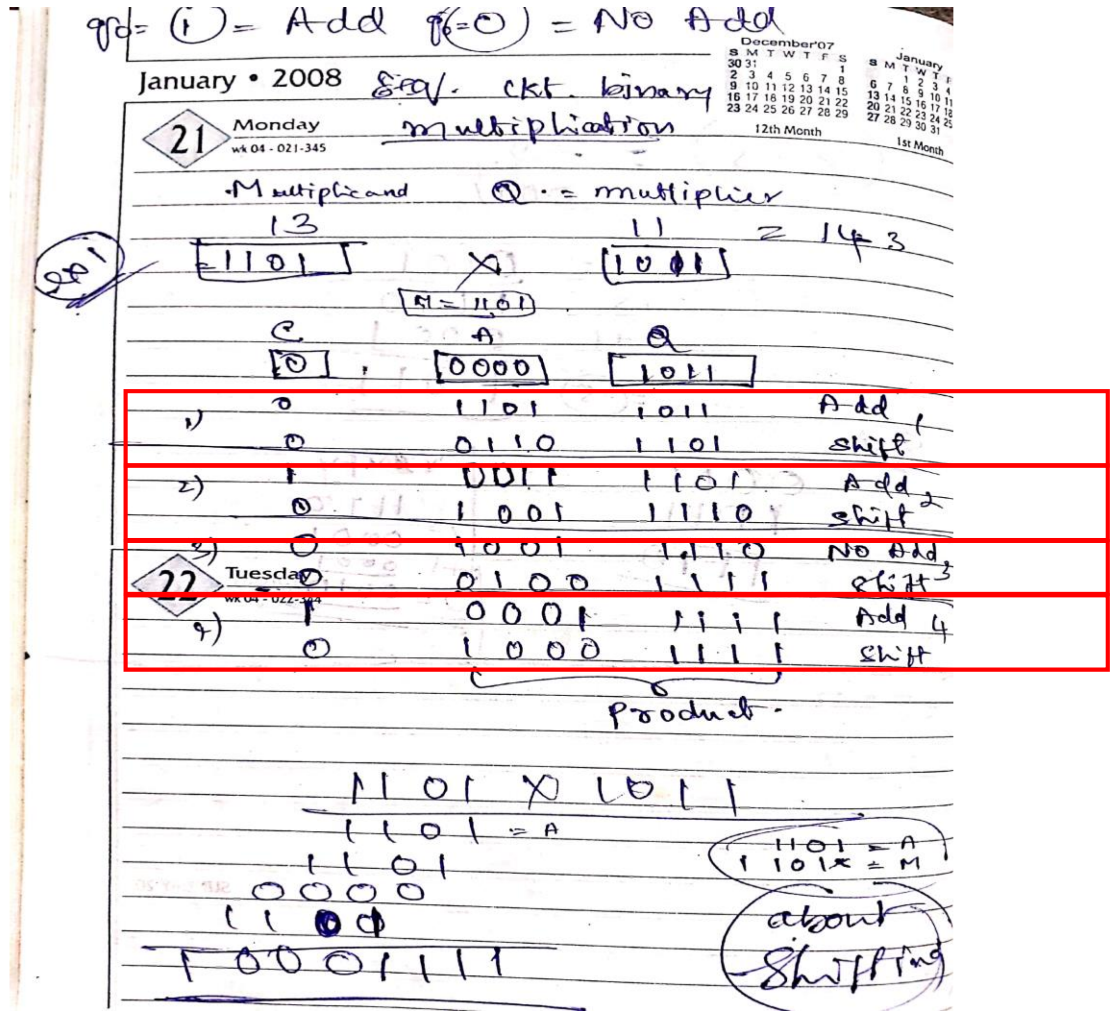
10011

in step-3

0100

<u>1101</u>

10001



Sequential multiplication-example 2

$15 \times 10 = 150$

$1111 \times 1010 = \underline{10010110}$

$\boxed{M = 1111}$

	C	A	Q	
	0	0000	1010	
1	0	0000	0101	No add / Shift
2	0	1111	0101	Add
	0	0111	1010	Shift
3	0	0011	1101	No Add / Shift
4	1	0010	1101	Add
	0	1001	0110	Shift

Sequential multiplication-
example 3

18 × 5 = 90

10010 × 101 = 1011010

$\boxed{M = 10010}$

	C	A	Q	
init	0	00000	00101	
1	0	10010	00101	Add
	0	01001	00010	Shift
2	0	00100	10001	No add / Shift
		10010		Add
	0	10110	10001	
3	0	01011	01000	Shift
4	0	00101	10100	No Add / Shift
5	0	00010	11010	No Add / Shift

Sequential multiplication-
example 4

$20 \times 23 = 460$

$10100 \times 10111 = 111001100$

$M = \boxed{10100}$

	C	A	Q	
init	0	00000	10111	
1	0	10100	10111	Add
	0	01010	01011	shift
2		10100		Add
	0	11110	01011	
	0	01111	00101	shift
3		10100		Add
	1	00011	00101	
	0	10001	10010	shift
4	0	01000	11001	No Add/shift
5		10100		Add
	0	11100	11001	
	0	01110	01100	shift

Sequential multiplication-
example 6

M Q

11110 = 30 × 00100 = 4 = 120

120 = 1111000

C	A	Q		
0	00000	00100	No add	1
0	00000	00010	Shift	
0	00000	00010	NA	2
0	00000	00001	Shift	
0	11110	00001	Add	3
0	01111	00000	Sh	
0	01111	00000	NA	4
0	00111	10000	Sh	
0	00111	10000	NA	5
0	00011	11000	Sh	

0001111000

M Q

001110 × 00100

00000 A
00000
11110

1111000

Signed Multiplication

Signed Multiplication

- Considering 2's-complement signed operands, what will happen to (-13)×(+11) if following the same method of unsigned multiplication?

					1	0	0	1	1	(- 13)
					0	1	0	1	1	(+11)
1	1	1	1	1	1	0	0	1	1	
1	1	1	1	1	0	0	1	1		
0	0	0	0	0	0	0	0			
1	1	1	0	0	1	1				
0	0	0	0	0	0					
1	1	0	1	1	1	0	0	0	1	(- 143)

Sign extension is shown in blue

1101110001
0010001110
1
0010001111

Sign extension of negative multiplicand.

Signed Multiplication

- For a negative multiplier, a straightforward solution is to form the 2's-complement of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier.
- This is possible because complementation of both operands does not change the value or the sign of the product.
- A technique that works equally well for both negative and positive multipliers – Booth algorithm.

Booth Algorithm

- Consider in a multiplication, the multiplier is positive 0011110, how many appropriately shifted versions of the multiplicand are added in a standard procedure?

$$
\begin{array}{cccccccccccccc}
 & & & & & & & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
 & & & & & & & 0 & 0 & +1 & +1 & +1 & +1 & 0 \\
\hline
 & & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & 0 & 1 & 0 & 1 & 1 & 0 & 1 & \\
 & & & & & 0 & 1 & 0 & 1 & 1 & 0 & 1 & & \\
 & & & & 0 & 1 & 0 & 1 & 1 & 0 & 1 & & & \\
 & & & 0 & 1 & 0 & 1 & 1 & 0 & 1 & & & & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & \\
\hline
0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0
\end{array}
$$

Booth Algorithm

- Since 0011110 = 0100000 – 0000010, if we use the expression to the right, what will happen?

							0	1	0	1	1	0	1	
							0	+1	0	0	0	-1	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	1	1	1	1	1	1	0	1	0	0	1	1		← 2's complement of the multiplicand
0	0	0	0	0	0	0	0	0	0	0	0			
0	0	0	0	0	0	0	0	0	0	0				
0	0	0	0	0	0	0	0	0	0					
0	0	0	1	0	1	1	0	1						
0	0	0	0	0	0	0	0							
0	0	0	1	0	1	0	1	0	0	0	1	1	0	

Booth Algorithm

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit i-1	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Booth multiplier recoding table.

Booth Algorithm

- In general, in the Booth scheme, -1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left.

0 0 1 0 1 1 0 0 1 1 1 0 1 0 1 1 0 0

$\Downarrow$

0 +1 -1 +1 0 -1 0 +1 0 0 -1 +1 -1 +1 0 -1 0 0

Booth recoding of a multiplier.

Booth Algorithm – Example 1

	0	1	1	0	1	(+13)
X	1	1	0	1	0	(- 6)

$\Longrightarrow$

					0	1	1	0	1	
					0	-1	+1	-1	0	
0	0	0	0	0	0	0	0	0	0	
1	1	1	1	1	0	0	1	1		
0	0	0	0	1	1	0	1			
1	1	1	0	0	1	1				
0	0	0	0	0	0					
1	1	1	0	1	1	0	0	1	0	(- 78)

Booth multiplication with a negative multiplier.

Booth Algorithm – Example 2: -5 X -6

+5 =0101
1s compl =1010
2s compl =1011

So, -5=1011

+6 =0110
1s compl =1001
2s compl =1010

So, -6=1010

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i-1$	
0	0	0 X M
0	1	+ 1 X M
1	0	− 1 X M
1	1	0 X M

Multiplier				1	0	1	0	0	
Code				-1	1	-1	0		
				1	0	1	1	Multiplicand	
				-1	1	-1	0	Multiplier code	
0	0	0	0	0	0	0	0		
0	0	0	0	1	0	1	X		
1	1	1	0	1	1	X	X		
0	0	1	0	1	X	X	X		
0	0	0	1	1	1	1	0	=30	

Implicit 0 to the right of LSB

which is $2^4+2^3+2^2+2^1=30$

Booth Algorithm – Example 3: -7 X 5

+7	=0111
1s compl	=1000
2s compl	=1001
So, -7=1001	

+5	=0101

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i-1$	
0	0	0 X M
0	1	+ 1 X M
1	0	− 1 X M
1	1	0 X M

	Multiplier			0	1	0	1	0		
	Code			1	-1	1	-1			
				1	0	0	1	Multiplicand		
				1	-1	1	-1	Multiplier code		
0	0	0	0	0	1	1	1			
1	1	1	1	0	0	1	X			
0	0	0	1	1	1	X	X			
1	1	0	0	1	X	X	X			
1	1	0	1	1	1	0	1	=-35		

Magnitude of –ve product

1111011101

0000100010 1's complement

0000100011 2's complement

which is $2^5+2^1+2^0=35$

Booth Algorithm – Example 4: -15 X 7

+15 =01111
1s compl =10000
2s compl =10001

So, -15=10001

+7 =00111

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i-1$	
0	0	0 X M
0	1	+ 1 X M
1	0	− 1 X M
1	1	0 X M

	Multiplier				0	0	1	1	1	0	
	Code				0	1	0	0	-1		
					1	0	0	0	1	Multiplicand	
					0	1	0	0	-1	Multiplier code	
0	0	0	0	0	0	1	1	1	1		
0	0	0	0	0	0	0	0	0	X		
0	0	0	0	0	0	0	0	X	X		
1	1	1	0	0	0	1	X	X	X		
0	0	0	0	0	0	X	X	X	X		
1	1	1	0	0	1	0	1	1	1	=-105	

Magnitude of –ve product
1110010111
0001101000 1’s complement
0001101001 2’s complement

which is $2^6+2^5+2^3+2^0=105$

Booth Algorithm – Example 5: -45 X -15

+45 =0101101

1s compl =1010010
1

2s compl =1010011

So, -45= 1010011+

+15 =01111

1s compl=10000
1

So, -15 =10001

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i-1$	
0	0	0 X M
0	1	+ 1 X M
1	0	− 1 X M
1	1	0 X M

		Multiplier				1	0	0	0	1	0		
		Code				-1	0	0	1	-1			
				1	0	1	0	0	1	1	Multiplicand		
						-1	0	0	1	-1	Multiplier code		
0	0	0	0	0	1	0	1	1	0	1			
1	1	1	1	0	1	0	0	1	1	x			
0	0	0	0	0	0	0	0	0	x	x			
0	0	0	0	0	0	0	0	x	x	x			
0	1	0	1	1	0	1	x	x	x	x			
0	1	0	1	0	1	0	0	0	1	1	=675		
	which is												

$2^9+2^7+2^5+2^1+2^0=675$

Booth Algorithm

- Best case – a long string of 1's (skipping over 1s)
- Worst case – 0's and 1's are alternating

Worst-case multiplier	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1
Ordinary multiplier	1	1	0	0	0	1	0	1	1	0	1	1	1	1	0	0
	0	-1	0	0	+1	-1	+1	0	-1	+1	0	0	0	-1	0	0
Good multiplier	0	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1
	0	0	0	+1	0	0	0	0	-1	0	0	0	+1	0	0	-1

Fast Multiplication

Bit-Pair Recoding of Multipliers

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position i
$i+1$	i	$i-1$	
0	0	0	0 X M
0	0	1	+ 1 X M
0	1	0	+ 1 X M
0	1	1	+ 2 X M
1	0	0	− 2 X M
1	0	1	− 1 X M
1	1	0	− 1 X M
1	1	1	0 X M

(b) Table of multiplicand selection decisions

Bit-Pair Recoding of Multipliers

	0	1	1	0	1	(+13)
$\times$	1	1	0	1	0	(- 6)

					0	1	1	0	1	
					0	-1	+1	-1	0	
0	0	0	0	0	0	0	0	0	0	
1	1	1	1	1	0	0	1	1		
0	0	0	0	1	1	0	1			
1	1	1	0	0	1	1				
0	0	0	0	0	0					
1	1	1	0	1	1	0	0	1	0	(- 78)

					0	1	1	0	1
					0		-1		-2
1	1	1	1	1	0	0	1	1	0
1	1	1	1	0	0	1	1		
0	0	0	0	0	0				
1	1	1	0	1	1	0	0	1	0

Figure 6.15. Multiplication requiring only $n/2$ summands.

Bit pair recoding – Example 2: -5 X -6

+5 =0101
1s compl =1010
2s compl =1011

So, -5=1011

+6 =0110
1s compl =1001
2s compl =1010

So, -6=1010

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position i
$i+1$	i	$i-1$	
0	0	0	0 X M
0	0	1	+1 X M
0	1	0	+1 X M
0	1	1	+2 X M
1	0	0	−2 X M
1	0	1	−1 X M
1	1	0	−1 X M
1	1	1	0 X M

Multiplier: 1 0 1 0 0

Code: -1 -2

				1	0	1	1	Multiplicand
					-1		-2	Multiplier code
0	0	0	0	1	0	1	0	
0	0	0	1	0	1	X	X	
0	0	0	1	1	1	1	0	=30

which is $2^4+2^3+2^2+2^1=30$

Bit pair recoding – Example 3: -7 X 5

+7 =0111
1s compl =1000
2s compl =1001

So, -7=1001

+5 =0101

Multiplier bit-pair $i+1$	i	Multiplier bit on the right $i-1$	Multiplicand selected at position i
0	0	0	0 X M
0	0	1	+1 X M
0	1	0	+1 X M
0	1	1	+2 X M
1	0	0	−2 X M
1	0	1	−1 X M
1	1	0	−1 X M
1	1	1	0 X M

	Multiplier:			0	1	0	1	0		
	Code:			+1		+1				
				1	0	0	1	Multiplicand		
					+1		+1	Multiplier code		
1	1	1	1	1	0	0	1			
1	1	1	0	0	1	X	X			
1	1	0	1	1	1	0	1	=-35		

Magnitude of –ve product
1111011101
0000100010 1's complement
0000100011 2's complement

which is $2^5+2^1+2^0=35$

Bit pair recoding– Example 4: -15 X 7

+15 =01111
1s compl =10000
2s compl =10001

So, -15=10001

+7 =00111

Multiplier bit-pair $i+1$	Multiplier bit-pair i	Multiplier bit on the right $i-1$	Multiplicand selected at position i
0	0	0	0 X M
0	0	1	+1 X M
0	1	0	+1 X M
0	1	1	+2 X M
1	0	0	−2 X M
1	0	1	−1 X M
1	1	0	−1 X M
1	1	1	0 X M

multiplier:			0	0	0	1	1	1	0	
code:			0		+2		-1			

					1	0	0	0	1	Multiplicand
				0		+2		-1		Multiplier code
0	0	0	0	0	0	1	1	1	1	
1	1	1	0	0	0	1	0	X	X	
0	0	0	0	0	0	X	X	X	X	
1	1	1	0	0	1	0	1	1	1	=-105

Magnitude of –ve product
1110010111
0001101000 1's complement
0001101001 2's complement

which is $2^6+2^5+2^3+2^0=105$

Bit pair recoding – Example 5: +5 X -2

+5 =0101	+2 =010 1s compl =101 2s compl =110 So, -2=110

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position i
$i+1$	i	$i-1$	
0	0	0	0 X M
0	0	1	+1 X M
0	1	0	+1 X M
0	1	1	+2 X M
1	0	0	−2 X M
1	0	1	−1 X M
1	1	0	−1 X M
1	1	1	0 X M

multiplier:	1	1	1	0	0			
code:	0		-2					

				0	1	0	1	Multiplicand
				0		-2		Multiplier code
1	1	1	1	0	1	1	0	
0	0	0	0	0	0	x	x	
1	1	1	1	0	1	1	0	=-10

Magnitude of the –ve product

11110110

00001001 1's complement

00001010 2's complement

which is ten.

Bit pair recoding – Example 5: -45 X -15

+45 =0101101

1s compl =1010010
1

2s compl =1010011

So, -45= 1010011

+15 =01111

1s compl=10000
1

So, -15 =10001

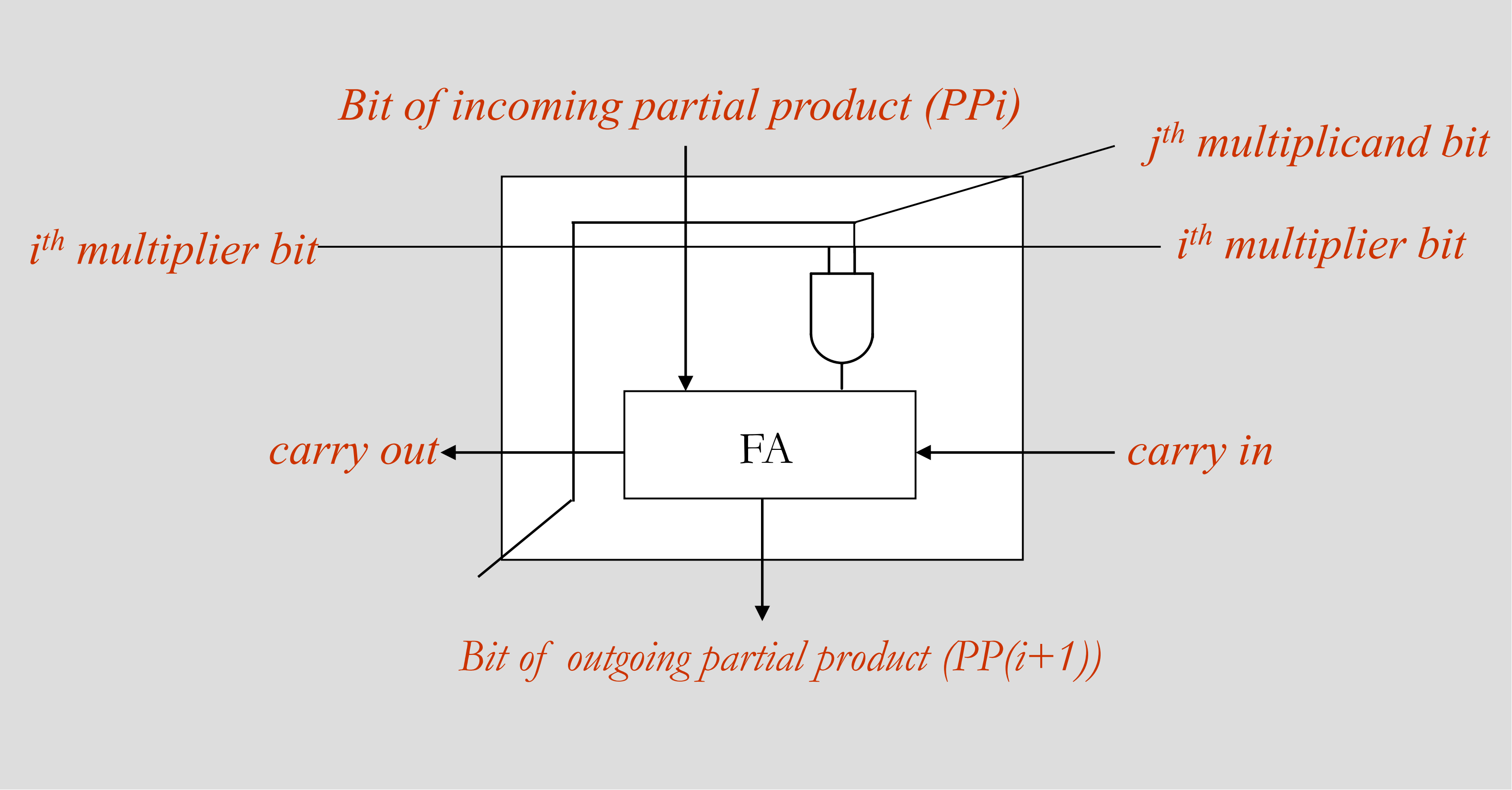
Multiplier bit-pair $i+1$	Multiplier bit-pair i	Multiplier bit on the right $i-1$	Multiplicand selected at position i
0	0	0	0 X M
0	0	1	+1 X M
0	1	0	+1 X M
0	1	1	+2 X M
1	0	0	−2 X M
1	0	1	−1 X M
1	1	0	−1 X M
1	1	1	0 X M

		Multiplier				1	0	0	0	1	0
		Code				-1	0	0	1	-1	
				1	0	1	0	0	1	1	Multiplicand
								-1	0	1	Multiplier code
1	1	1	1	1	0	1	0	0	1	1	
0	0	0	0	0	0	0	0	0	x	x	
0	1	0	1	1	0	1	x	x	x	x	
0	1	0	1	0	1	0	0	0	1	1	=675

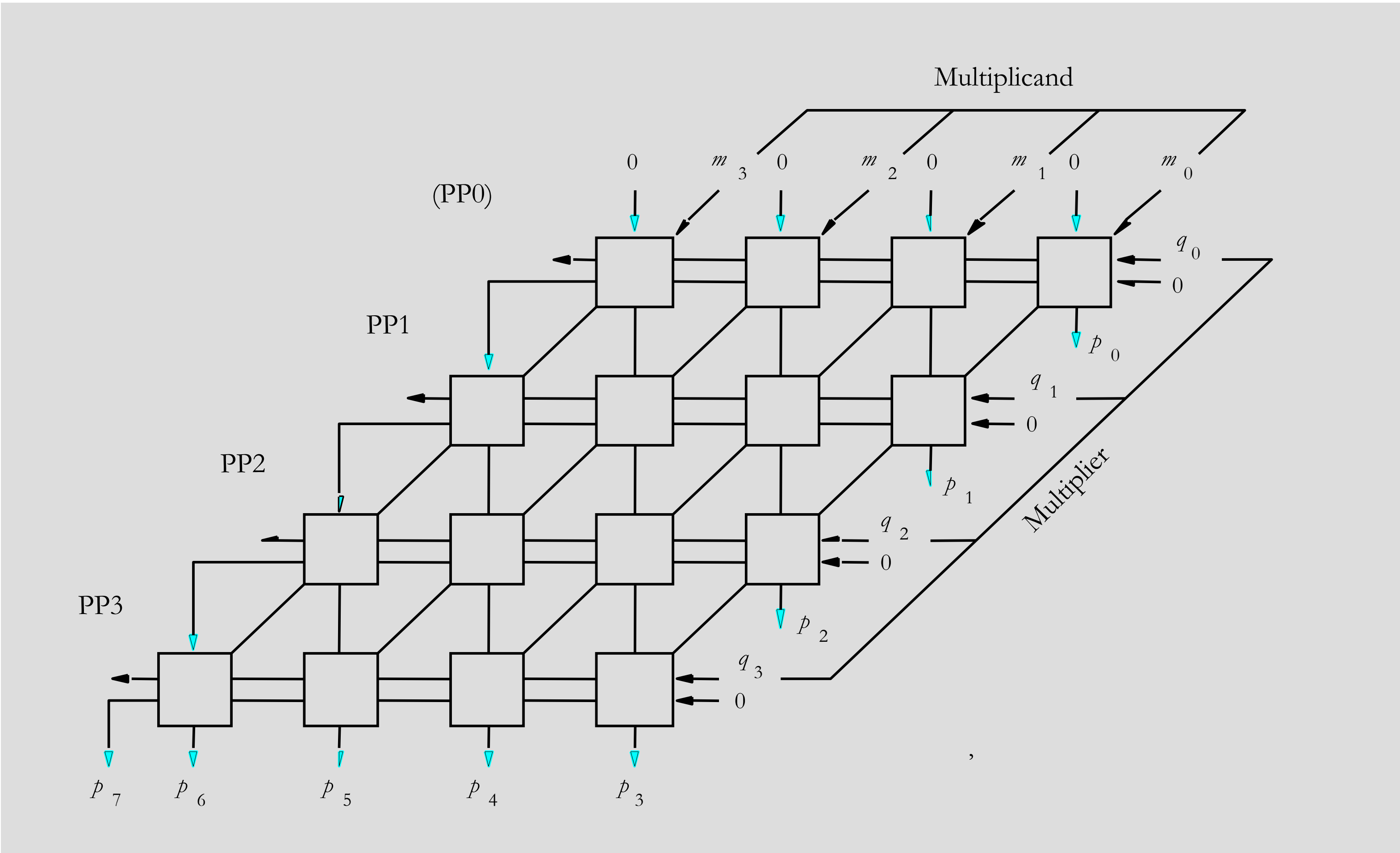
which is
$2^9+2^7+2^5+2^1+2^0=675$

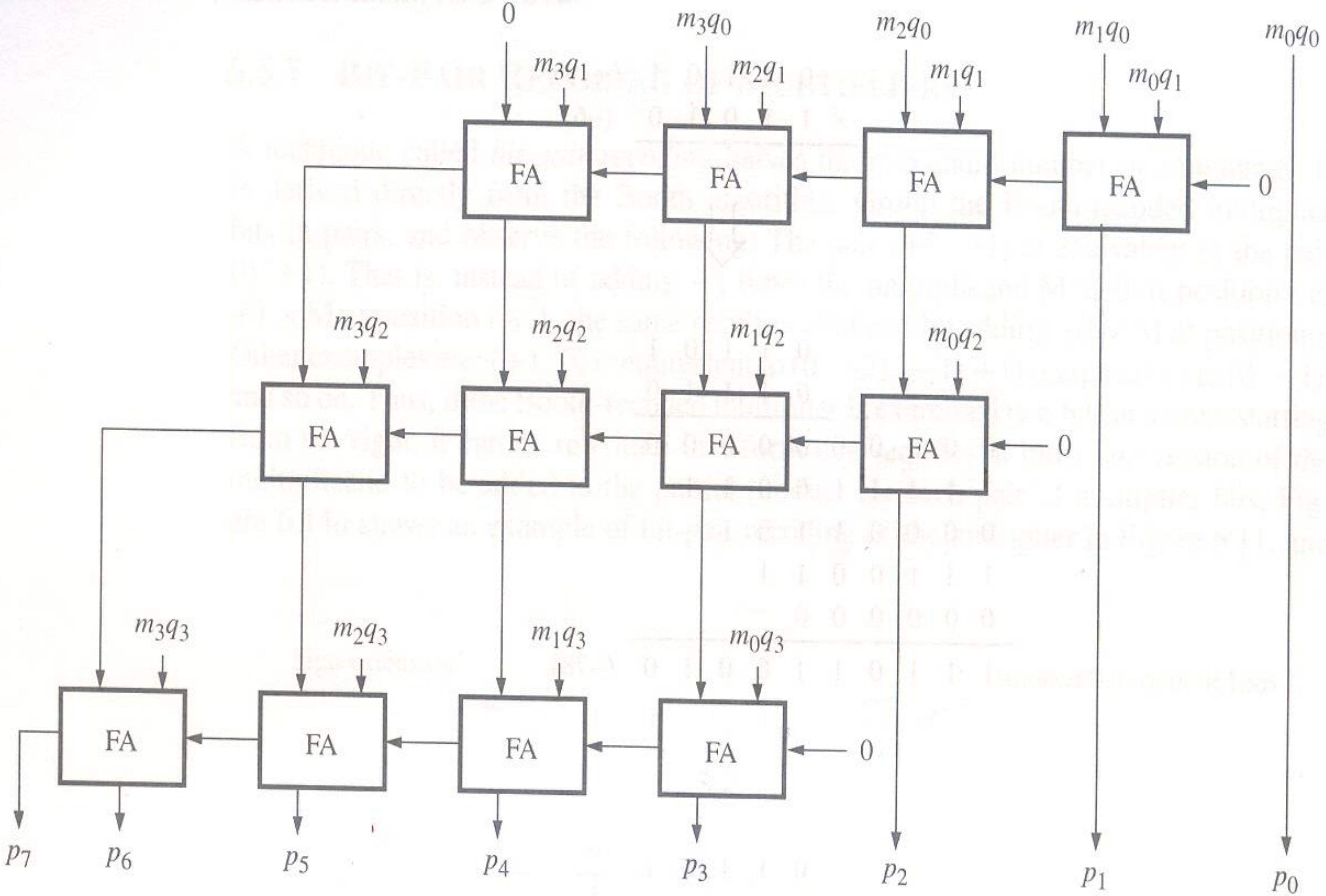
Multiplication of unsigned numbers

Typical multiplication cell



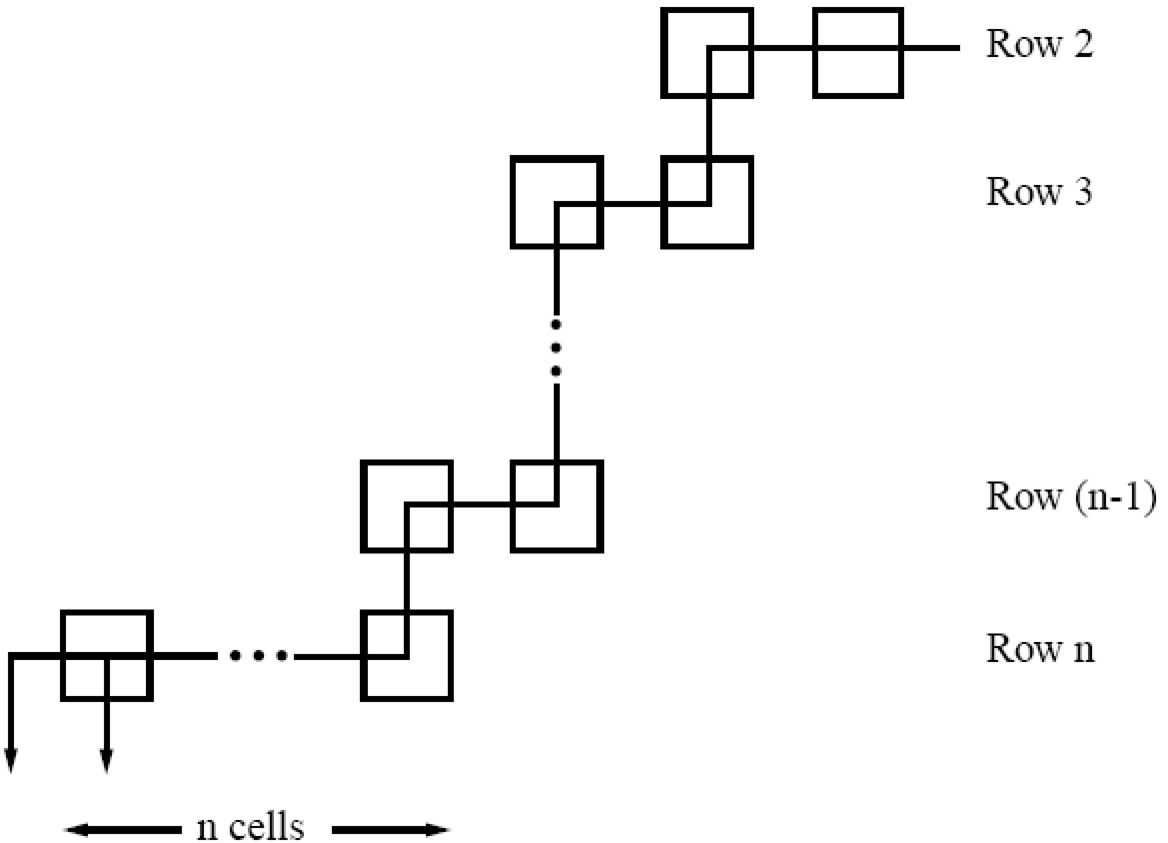
Combinatorial array multiplier





(a) Ripple-carry array (Figure 6.6 structure)

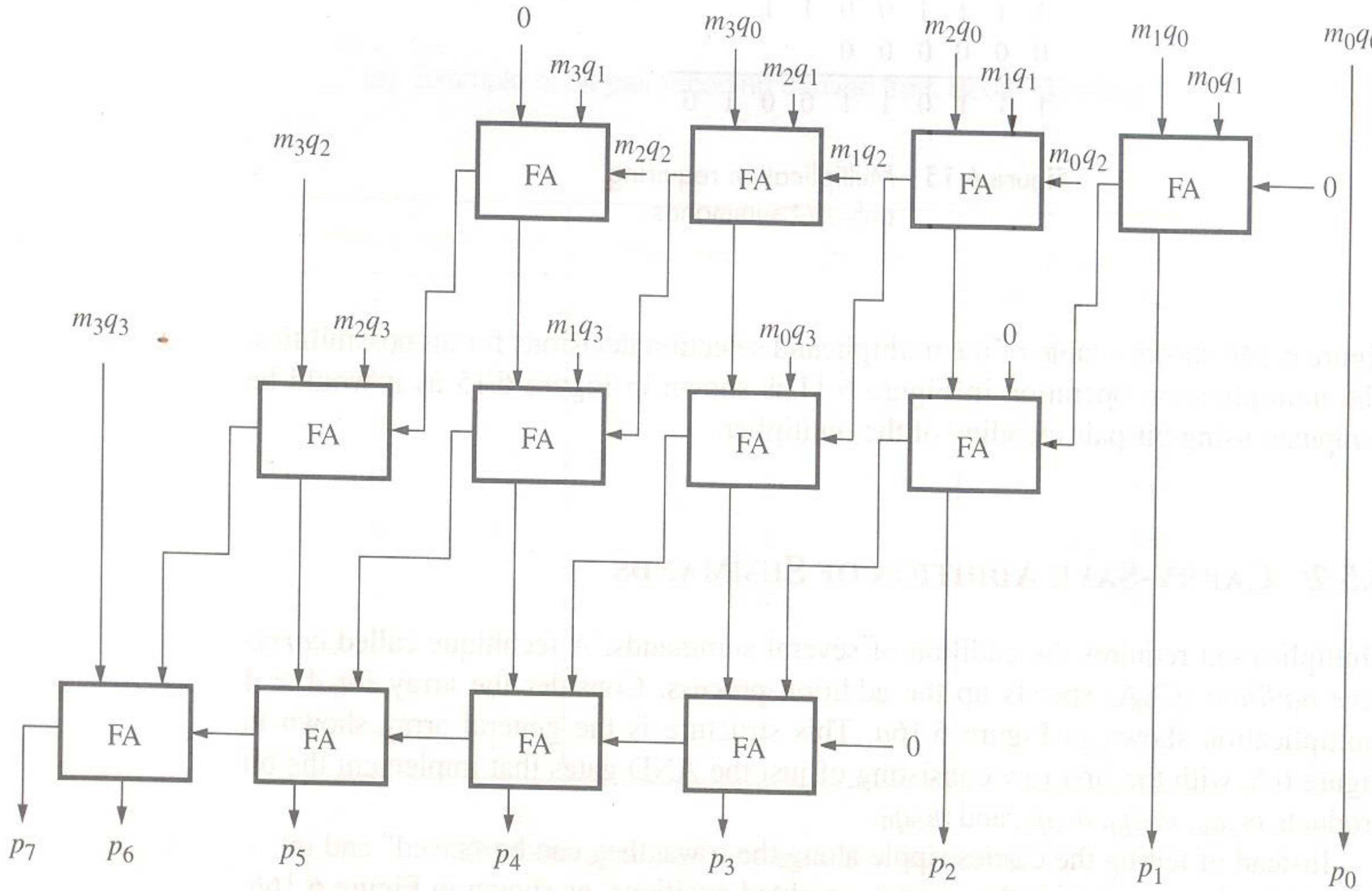
Worst case delay through nxn ripple-carry array



- Full adder delay - 2 , AND gate delay – 1
- $4(n-2)+ 2n +1 = 6n – 8 + 1 =$ **$6(n-1) - 1$**

Worst case delay through 4x4 ripple-carry array

- The worst case delay path is =6(n-1)-1 = 6 x 3 – 1 = **17**
  - the initial AND gate delay to develop all bit products =1
  - vertically through the first two rows
    - (a total of two FA block delays), =4x2=4
  - followed by the four FA blocks in the third row
    =4x2
  - Total delay
    - i.e 1+4x2+4x2 =**17 gate delays**



(b) Carry-save array

Worst case delay through nxn carry-save array

- The delay through (n -2) carry-save rows of FA blocks = 2(n - 2) gate delays,
- followed by 2n gate delays along the n FA blocks of the last row = 2n
- the initial AND gate delay to develop all bit products = 1
- for a total of
- =2(n -2) + 2n + 1 = **4(n -1) + 1** gate delays

Worst case delay through 4x4 carry-save array

- the worst case delay path is
  - the initial AND gate delay to develop all bit products =1
  - vertically through the first two rows
    - (a total of two FA block delays), =2x2=4
  - followed by the four FA blocks in the third row
    =4x2
  - Total delay
    - i.e 1+2x2+4x2 =**13 gate delays**

Carry-Save Addition of Summands(Cont.,)

- Consider the addition of many summands, we can:
  - Group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay
  - Group all of the S and C vectors into threes, and perform carry-save addition on them, generating a further set of S and C vectors in one more full-adder delay
  - Continue with this process until there are only two vectors remaining
  - They can be added in a Ripple Carry Adder or Carry Lookahead Adder to produce the desired product

Carry-Save Addition of Summands

						1	0	1	1	0	1	(45)	M
					x	1	1	1	1	1	1	(63)	Q
						1	0	1	1	0	1		A
					1	0	1	1	0	1			B
				1	0	1	1	0	1				C
			1	0	1	1	0	1					D
		1	0	1	1	0	1						E
	1	0	1	1	0	1							F
1	0	1	1	0	0	0	1	0	0	1	1	(2,835)	Product

Figure 6.17. A multiplication example used to illustrate carry-save addition as shown in Figure 6.18.

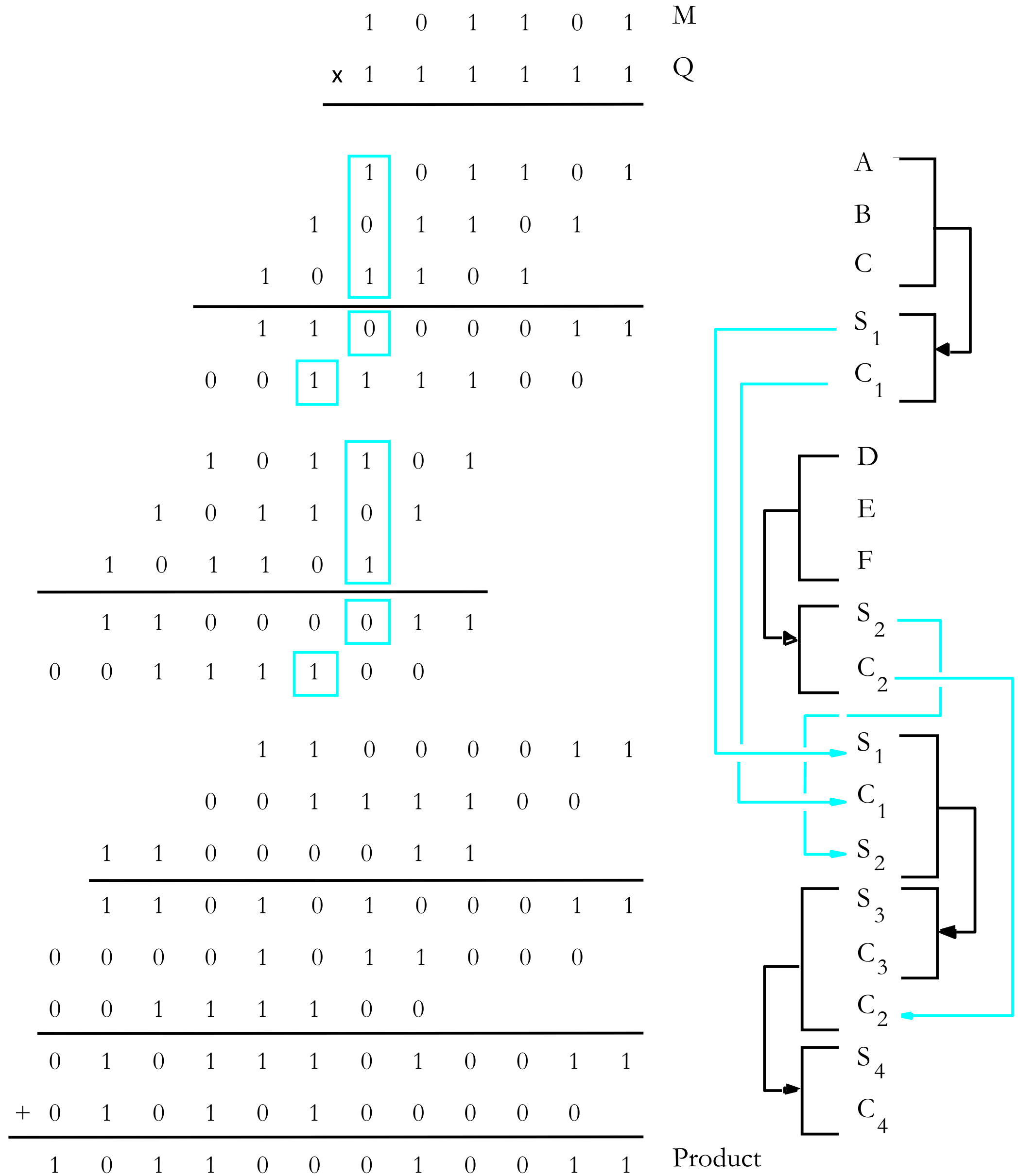
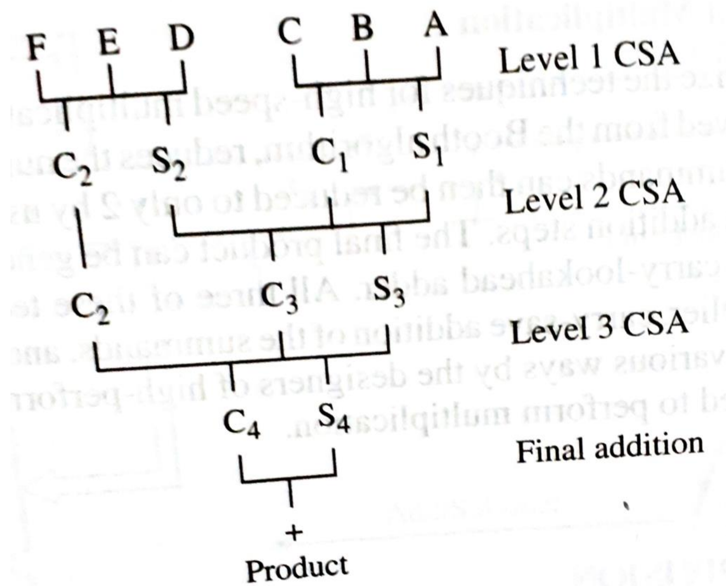


Figure 6.18. The multiplication example from Figure 6.17 performed using carry-save addition.



- Delay through nxn ripple-carry array = 6(n-1) – 1
  - **6x5-1= 29**
- Gate delays required to perform 6x6 multiplication using carry-save addition
  - After 1 AND gate delay all six summands (A-F) are available as i/p to CSA
  - S4 and C4 are available after 6 gate delays
  - 8 gate delays to add S4 and C4 using CLA
  - **Total=15** ; 50% reduction

Integer Division

Manual Division

$$\begin{array}{r} 21 \\ 13 \overline{)\,274} \\ \underline{26} \\ 14 \\ \underline{13} \\ 1 \end{array} \qquad\qquad \begin{array}{r} 10101 \\ 1101 \overline{)\,100010010} \\ \underline{1101} \\ 10000 \\ \underline{1101} \\ 1110 \\ \underline{1101} \\ 1 \end{array}$$

Longhand division examples.

Longhand Division Steps

- Position the divisor appropriately with respect to the dividend and perform a subtraction.
- If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed.
- If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction.

Circuit Arrangement

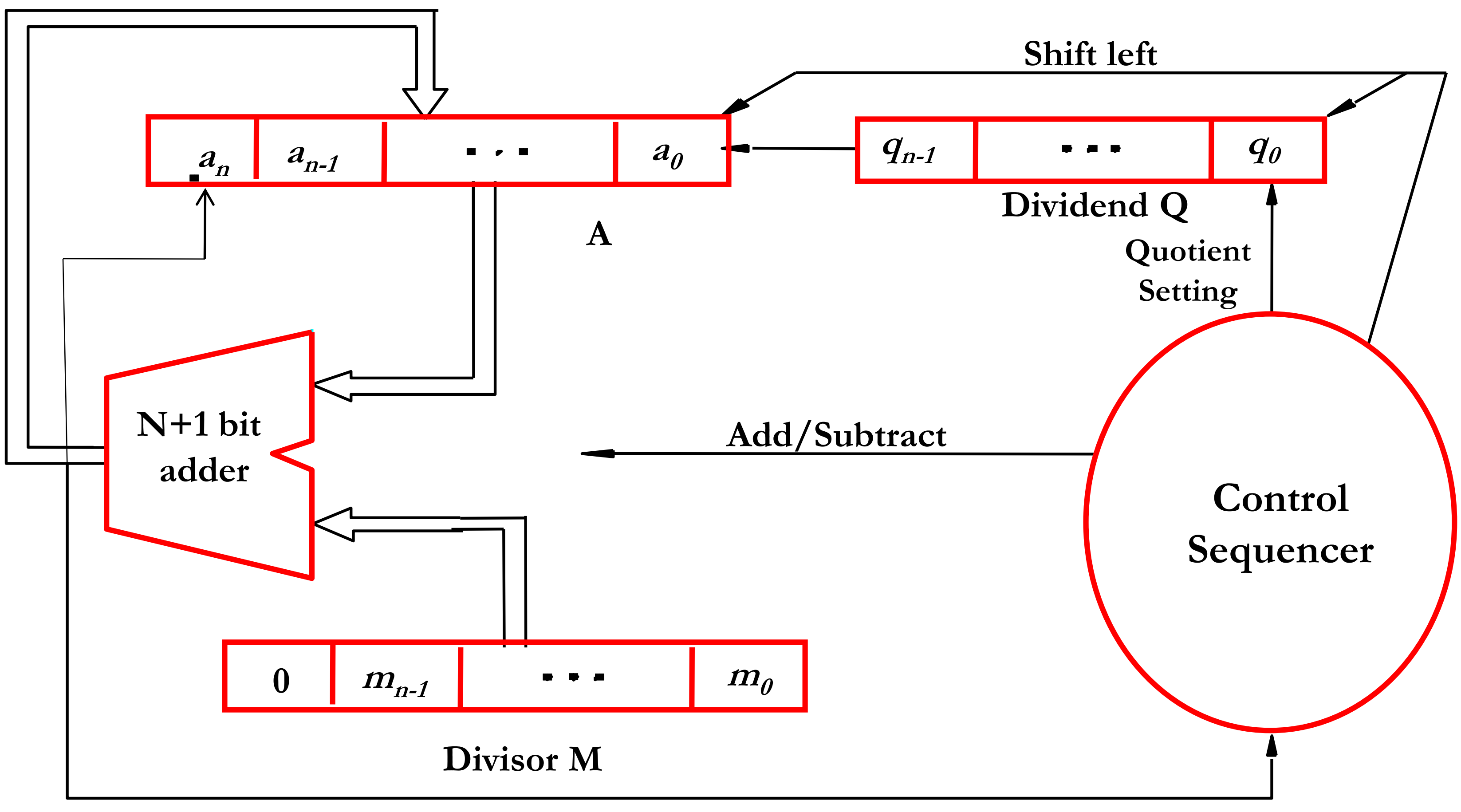


Figure 6.21. Circuit arrangement for binary division.

Restoring Division

- Shift A and Q left one binary position
- Subtract M from A, and place the answer back in A
- If the sign of A is 1, set q_0 to 0 and add M back to A (restore A); otherwise, set q_0 to 1
- Repeat these steps *n* times

Examples

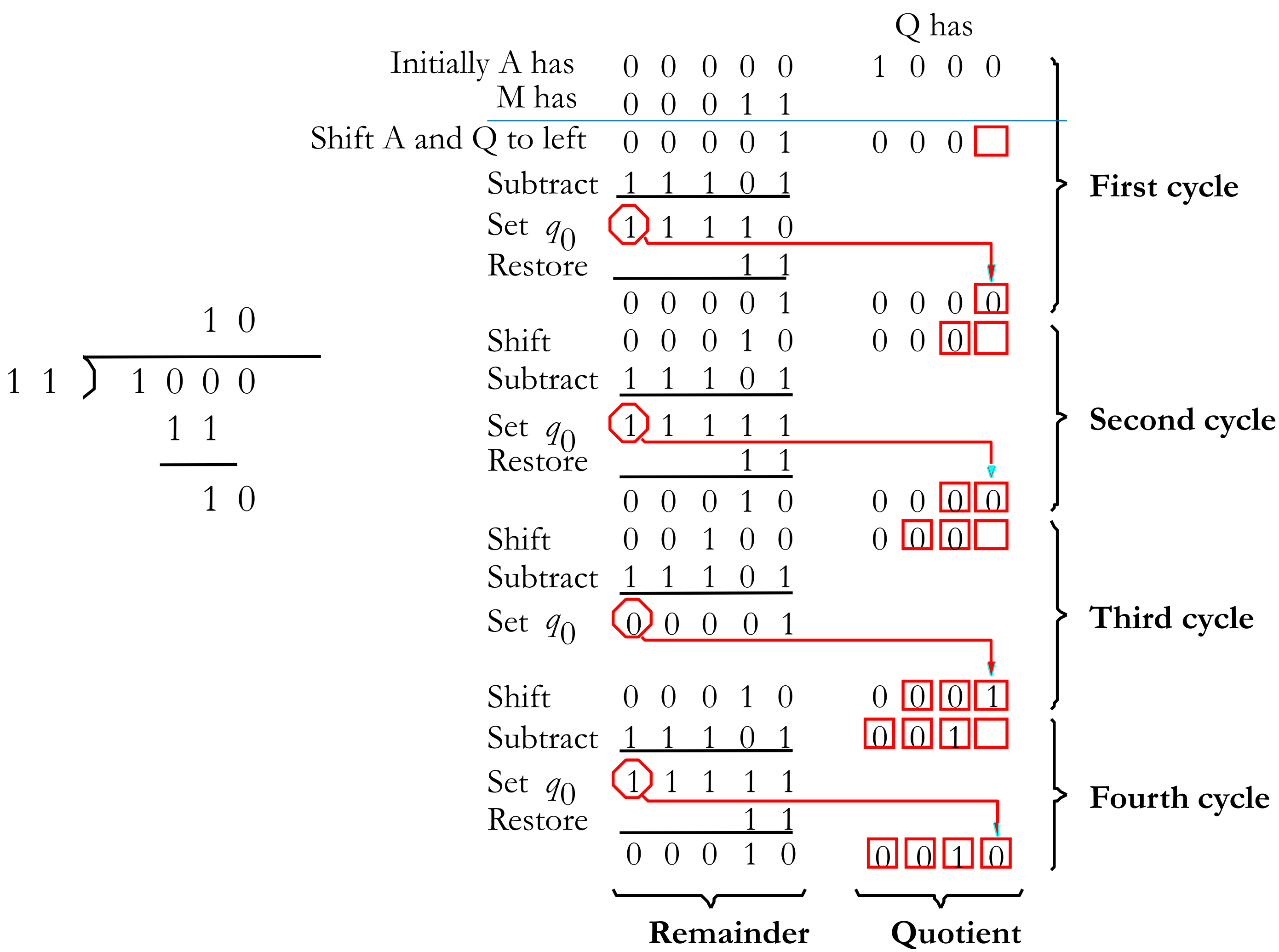
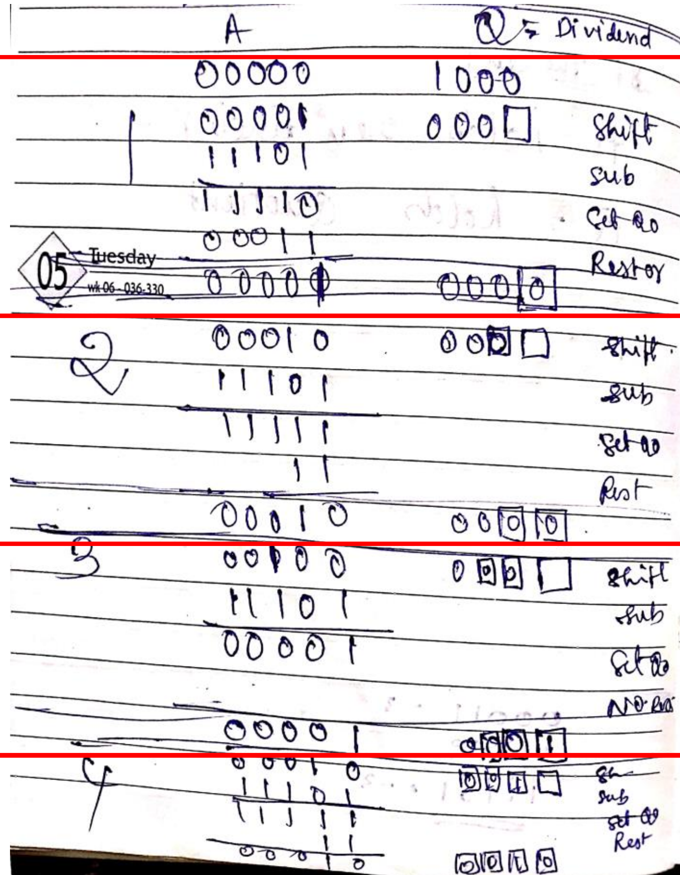


Figure 6.22. A restoring-division example.

Restoring division-example2 : 8÷3



3=00011
1s compl
=11100
2s compl
=11101

So, -3=11101

Restoring division-example3

$21 \div 5$

$10101 \div 101$

Restoring

2's = 101011

A	Q	
000000	10101	
000001	0101	Shift
111011		sub
111100	01010	set Q0.
000001	01010	Restore.
000010	1010	Shift
111011		sub.
111101	10100	set Q0.
000010	10100	Rest
000101	0100	Shift
111011		sub.
000000	01001	set Q0. no rest
000000	1001	Shift
111011		sub
111011	10010	rest
000000	10010	
000001	0010	Shift
111011		sub.
111100	00100	set Q0
000001	00100.	rest

21 =000101

1s compl = 111010

2s compl = 111011

So, -5= 111011

Nonrestoring Division

- Avoid the need for restoring A after an unsuccessful subtraction.
- Any idea?
- Step 1: (Repeat *n* times)
  - If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
  - Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.
- Step2: If the sign of A is 1, add M to A

Examples

Initially	0	0	0	0	0	1	0	0	0	First cycle
	0	0	0	1	1					
Shift	0	0	0	0	1	0	0	0	$\square$	
Subtract	1	1	1	0	1					
Set q_0	1	1	1	1	0	0	0	0	0	
Shift	1	1	1	0	0	0	0	0	$\square$	Second cycle
Add	0	0	0	1	1					
Set q_0	1	1	1	1	1	0	0	0	0	
Shift	1	1	1	1	0	0	0	0	$\square$	Third cycle
Add	0	0	0	1	1					
Set q_0	0	0	0	0	1	0	0	0	1	
Shift	0	0	0	1	0	0	0	1	$\square$	Fourth cycle
Subtract	1	1	1	0	1					
Set q_0	1	1	1	1	1	0	0	1	0	
						Quotient				

	1	1	1	1	1	Restore remainder
	0	0	0	1	1	
Add	0	0	0	1	0	
	Remainder					

A nonrestoring-division example.

Non restoring division – Example-1

P	A	
00000	1110	Divide 14 = 1110_2 by 3 = 11_2. B always contains 0011_2.
00001	110	step 1(i-b): shift.
+11101		step 1(ii-b): subtract b (add two's complement).
11110	110**0**	step 1(iii): P is negative, so set quotient bit to 0.
11101	10**0**	step 2(i-a): shift.
+00011		step 2(ii-a): add b.
00000	10**01**	step 2(iii): P is nonnegative, so set quotient bit to 1.
00001	0**01**	step 3(i-b): shift.
+11101		step 3(ii-b): subtract b.
11110	0**010**	step 3(iii): P is negative, so set quotient bit to 0.
11100	**010**	step 4(i-a): shift.
+00011		step 4(ii-a): add b.
11111	**0100**	step 4(iii): P is negative, so set quotient bit to 0.
+00011		Remainder is negative, so do final restore step.
00010		The quotient is 0100_2 and the remainder is 00010_2.

B

0001

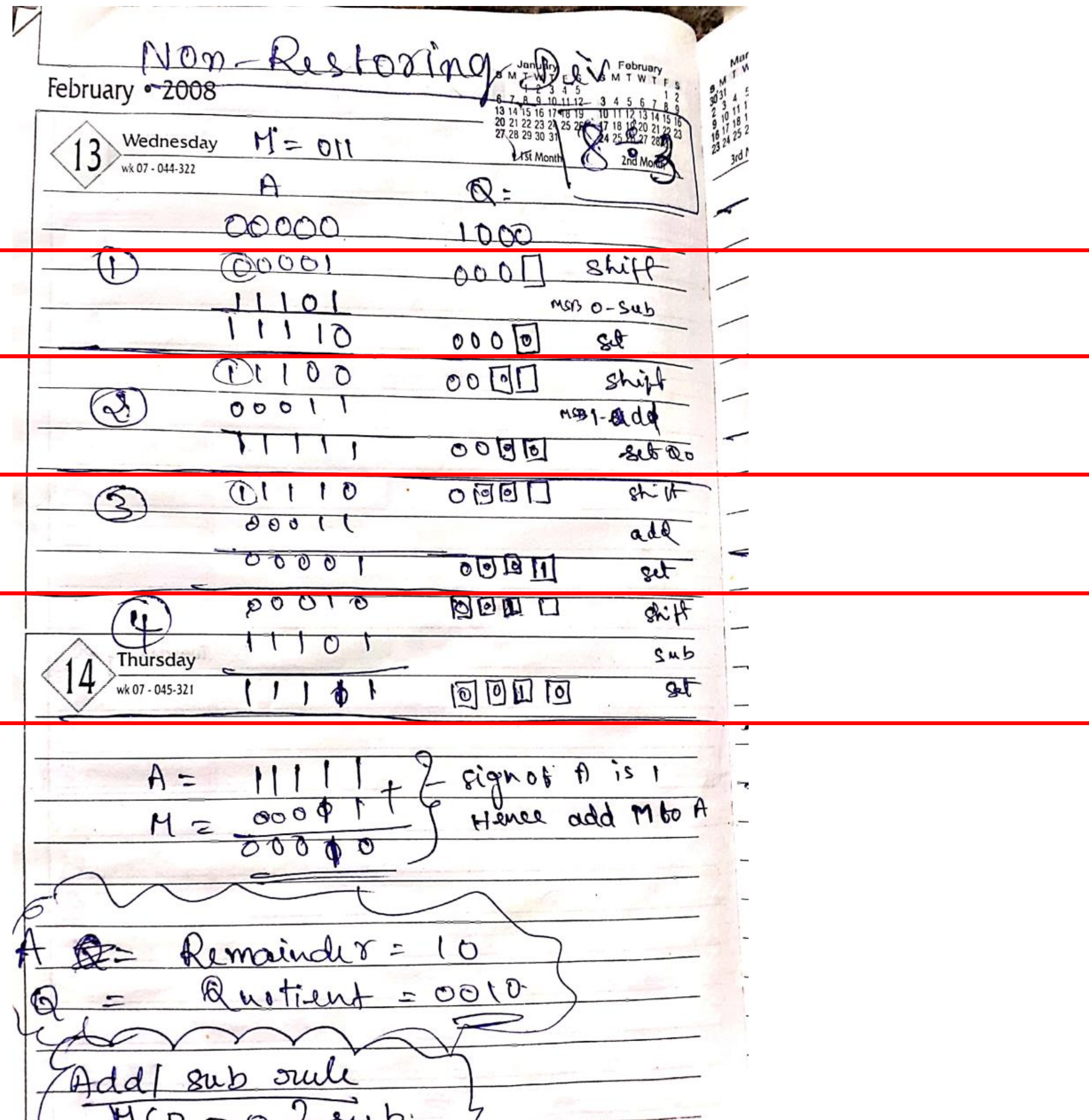
1

Non restoring division – Example-2: **8÷3**

3=00011
1s compl
=11100
2s compl
=11101

So, -3=11101

Rule:
If MSB of A is 1; then add the divisor
If MSB of A is 0; then subtract the divisor



After 4th step if the value of A is positive it is the remainder.

If A is negative it is to be restored.

Non restoring division– Example-2

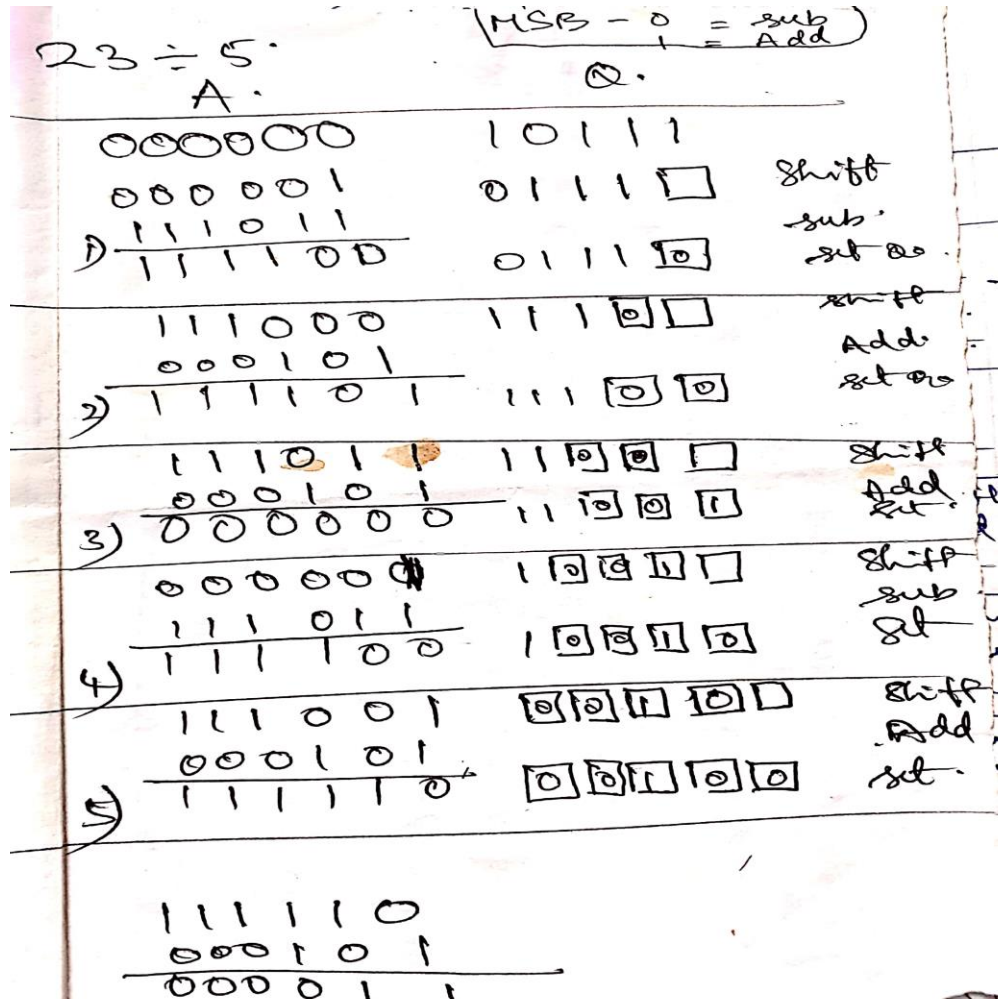
+5
=0101
1s compl
=1010
2s compl
=1011

So, -5=1011

Rule:
If MSB of A is 1; then add the divisor
If MSB of A is 0; then subtract the divisor

After 5th step if the value of A is positive it is the remainder.

If A is negative it is to be restored.



Non restoring division – Example-3

$14 = 1110 \quad 3 = 0011 \quad \therefore -3 = 11101$

	A	Q	
	00000	1110	Init
I	00001	110 $\boxed{\ }$	Shift
	11101	-do-	MSB = 0 ; sub.
	11110	110 $\boxed{0}$	Set Q0.
II	11101	10 $\boxed{0}$ $\boxed{\ }$	Shift
	00011	-do-	MSB = 1 ; Add
	00000	10 $\boxed{0}$ $\boxed{1}$	Set Q0.
III	00001	0 $\boxed{0}$ $\boxed{1}$ $\boxed{\ }$	Shift
	11101		MSB = 0 ; sub.
	11110	0 $\boxed{0}$ $\boxed{1}$ $\boxed{0}$	Set Q0.
IV	11100	$\boxed{0}$ $\boxed{1}$ $\boxed{0}$ $\boxed{\ }$	Shift.
	00011		MSB = 1 ; Add.
	11111	$\boxed{0}$ $\boxed{1}$ $\boxed{0}$ $\boxed{0}$	Set Q0.

A = 11111 → Sign of A = 1
Add M to A.

M = 00011

00010

Quotient = 100
Rem = 10

$$3 \overline{)\,14\,}\ 4$$
12
02

Floating-Point Numbers and Operations

Procedure for conversion of decimal fraction to binary

- Let X be a decimal fraction: $\mathbf{0.d_1d_2..d_n}$
  i = 1
- Repeat until X = 0 or i = required no. of binary fractional digits
  {
  Y = X * 2
  X = fractional part of Y
  b_i = integer part of Y
  i = i + 1
  }

- Binary equivalent fraction =b1b2b3…

EXAMPLE 1

- Convert X=0.75 to binary

- X = 0.75 (initial value)
- X* 2 = 1.50. Set b1 = 1, X = 0.5
- X* 2 = 1.0. Set b2 = 1, X = 0.0

- The binary representation for 0.75 is thus 0.b1b2 = 0.11

EXAMPLE 2

Convert the decimal value 4.9 into binary

- Part 1: convert the integer part into binary: 4 = 100
- Part 2: Convert the fractional part (i.e x=0.9) into binary using multiplication by 2:

1. X*2 = 0.9*2 = 1.8 Set b_1 = 1, X = 0.8
2. X*2 = 0.8*2 = 1.6 Set b_2 = 1, X = 0.6
3. X*2 = 0.6*2 = 1.2 Set b_3 = 1, X = 0.2
4. X*2 = 0.2*2 = 0.4 Set b_4 = 0, X = 0.4
5. X*2 = 0.4*2 = 0.8 Set b_5 = 0, X = 0.8,

which repeats after the 4^{th} step above

- Since X is now repeating the value 0.8, we know the representation will repeat.
- The binary representation of 4.9 is thus:

100.1110011001100...

Convert 12.67 to binary

12 =1100

1) 0.67 x 2 =1.34 ; b1=1 X=0.34

2) 0.34 x 2 = 0.68; b2=0 X =0.68

3) 0.68x2 =1.36 ; b3=1 x=0.36

4) 0.36x2=0.72 ; b4=0 x=0.72

5) 0.72x2=1.44 ; b5=1 x=0.44

6) 0.44x2 =0.88; b6=0 x=0.88

7) 0.88x2=1.76; b7=1 x=0.76

8) 0.76x2=1.52; b8=1 x=0.52

12.67 in binary 1100.10101011…

0.5
0.125
0.03125

0.65625

$1100.10101011 = 1\times2^3 + 1\times2^2 + 0\times2^1 + 0\times2^0 + 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3} + 0\times2^{-4} + 1\times2^{-5}$

$=12.65625$

Procedure for conversion of binary fraction to decimal

- In the binary representation $0.b_1b_2...b_m$
- b_1 represents 2^{-1} (i.e., 1/2)
- b_2 represents 2^{-2} (i.e., 1/4)

  ...

- b_m represents 2^{-m} (i.e, $1/(2^m)$)

- So, 0.11 binary represents

  $2^{-1} + 2^{-2} = 1/2 + 1/4 = 3/4 = 0.75$

Fractions

If b is a binary vector, then we have seen that it can be interpreted as an unsigned integer by:

$$V(b) = b_{31}.2^{31} + b_{30}.2^{30} + b_{n-3}.2^{29} + ... + b_1.2^1 + b_0.2^0$$

This vector has an implicit binary point to its immediate right:

$b_{31}b_{30}b_{29}\ldots\ldots\ldots\ldots\ldots\ldots.b_1b_0.$ implicit binary point

Suppose if the binary vector is interpreted with the implicit binary point is just left of the sign bit:

implicit binary point $.b_{31}b_{30}b_{29}\ldots\ldots\ldots\ldots\ldots\ldots.b_1b_0$

The value of b is then given by:

$$V(b) = b_{31}.2^{-1} + b_{30}.2^{-2} + b_{29}.2^{-3} + ... + b_1.2^{-31} + b_0.2^{-32}$$

Range of fractions

The value of the unsigned binary fraction is:

$$V(b) = b_{31}.2^{-1} + b_{30}.2^{-2} + b_{29}.2^{-3} + ... + b_{1}.2^{-31} + b_{0}.2^{-32}$$

The range of the numbers represented in this format is:

$$0 \leq V(b) \leq 1 - 2^{-32} \approx 0.9999999998$$

In general for a *n*-bit binary fraction (a number with an assumed binary point at the immediate left of the vector), then the range of values is:

$$0 \leq V(b) \leq 1 - 2^{-n}$$

Scientific notation

- Previous representations have a fixed point. Either the point is to the immediate right or it is to the immediate left. This is called Fixed point representation.
- The drawback of fixed point representation is that it can only represent a finite range (and quite small) range of numbers.
- To enable representing very large and very small numbers, the position of the binary point is varied (floated) as the computation proceeds.

A more convenient representation is the scientific representation, where the numbers are represented in the form:

$$\boxed{x = m_1.m_2m_3m_4 \times b^{\pm e}}$$

Components of these numbers are:

Mantissa (m), implied base (b), and exponent (e)

Example for decimal scientific notation: -6.0257×10^{23}
1.0341×10^{-2}

These numbers have five significand digits.

The scale factors 23, -2 explicitly indicate the position of the decimal point

Significant digits

- A number such as the following is said to have 7 significant digits
  $\pm X_{1.}X_2X_3X_4X_5X_6X_7 \times 10^{\pm Y1Y2}$
  Where Xi and Yi are the decimal digits

- 7 bit mantissa and exponent range (±99) are sufficient for a wide range of scientific calculations.
- We can approximate 7 digit mantissa and exponent range (±99) in a 32 bit binary representation
  - 24 bits mantissa can approximately represent a 7 digit decimal number. Since the mantissa is normalized, leading nonzero bit need not be included. So only 23 bits are sufficient.
  - 8 bit exponent provides a scale factor of reasonable range.
  - One bit is needed for the sign

Normalization

Consider the number: $x = 0.0004056781 \times 10^{12}$

If the number is to be represented using only 7 significant mantissa digits, the representation ignoring rounding is: $x = 0.0004056 \times 10^{12}$

If the number is shifted so that as many significant digits are brought into 7 available slots: $x = 0.4056781 \times 10^{9} = 0.0004056 \times 10^{12}$

Exponent of x was decreased by 1 for every left shift of x.

Same methodology holds in the case of binary mantissas

$$0.0001101000(10110) \times 2^{8} = 0.1101000101(10) \times 2^{5}$$

Normalization

- When the floating-point decimal point is placed to the right of the first nonzero digit, the number is said to be Normalized.

- A floating-point binary number is in normalized form if the binary point is placed to the right of the first nonzero bit
- All normalized floating point numbers in this system will be of the form:

1.xxxxx.......xx

- The procedure for normalizing a floating point number is:

```
Do (until MSB of mantissa = = 1)
        Shift the mantissa left (or right)
        Decrement (increment) the exponent by 1
end do
```

- Unnormalized number =+0.0010110… × 2^9
- **The above number in the** normalized form = +1.0110… × 2^6

Excess notation

- Rather than representing an exponent in 2's complement form, it turns out to be more beneficial to represent the exponent in excess notation.

- For an n bit exponent, the bias value is $=2^{n-1} - 1$

- For 8 bit exponent, the bias value is =127

- Exponent in the excess-127 notation is calculated as: $E' = E_{true} + 127$

- For 8 bit exponent, range of exponent values is -126 to +127.

- In general, excess-p coding is represented as: $E' = E_{true} + p$
- True exponent of -126 is represented as 1
  0 is represented as 127
  127 is represented as 254
- This enables efficient comparison of the relative sizes of two floating point numbers.
    - If two normalized floating point numbers have different exponents, the one with the bigger exponent is the bigger of the two

IEEE notation

IEEE Floating Point notation is the standard representation in use. There are two representations:

- Single precision.
- Double precision.

Both have an implied base of 2.

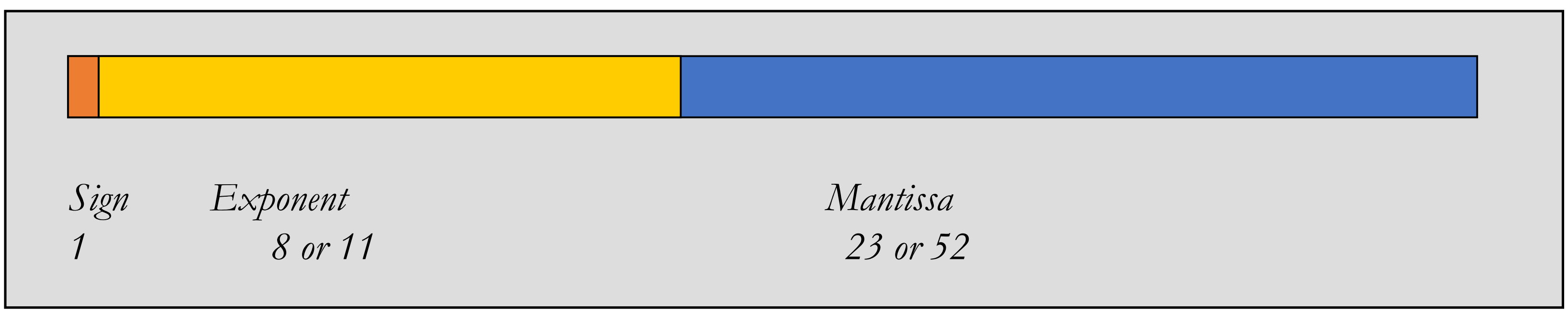
Single precision:

- 32 bits (23-bit mantissa, 8-bit exponent in excess-127 representation)

Double precision:

- 64 bits (52-bit mantissa, 11-bit exponent in excess-1023 representation)

Fractional mantissa, with an implied binary point at immediate left.



IEEE notation

- Floating point numbers have to be represented in a normalized form to maximize the use of available mantissa digits.
- In a base-2 representation, this implies that the MSB of the mantissa is always equal to 1.
- If every number is normalized, then the MSB of the mantissa is always 1. We can do away without storing the MSB.
- IEEE notation assumes that all numbers are normalized so that the MSB of the mantissa is a 1 and does not store this bit.
- The values of the numbers represented in the IEEE single precision notation are of the form:

$$\boxed{\pm 1.M \times 2^{(E - 127)}}$$

- The hidden 1 forms the integer part of the mantissa.
- Note that excess-127 and excess-1023 (not excess-128 or excess-1024) are used to represent the exponent.

Exponent field

In the IEEE representation, the exponent is in excess-127 (excess-1023) notation.
The actual exponents represented are:

$-126 <= E <= 127$ *and* $-1022 <= E <= 1023$

not

$-127 <= E <= 128$ *and* $-1023 <= E <= 1024$

This is because the IEEE uses the exponents -127 and 128 (and -1023 and 1024), that is the actual values 0 and 255 to represent special conditions:

- Exact zero
- Infinity

Special Values

- E'=0 and E'=255 are used to represent special values
- Exact Zero is represented by E'=0, and M=0
- $\pm\infty$ represented by S=0 or 1, E=255 and M=0
- NaN means Not a Number, produced by invalid operations
- Example for NaN: 0/0 or $\sqrt{-1}$
- NaN is represented by S=1, all bits in E'=255 and $M \neq 0$

Normalization, overflow and underflow

Applying the normalization procedure to: $\boxed{0.000111001110....0010 \;\; x\; 2^{-123}}$

gives: $\boxed{1.11001110........ \qquad x\; 2^{-127}}$

But we cannot represent an exponent of –127, in trying to normalize the number we have underflowed our representation.

Applying the normalization procedure to: $\boxed{1000.111000............x\; 2^{125}}$

gives: $\boxed{1.000111000...........x\; 2^{128}}$

This overflows the representation.

Floating point arithmetic

Addition:

$$3.1415 \times 10^8 + 1.19 \times 10^6 = 3.1415 \times 10^8 + 0.0119 \times 10^8 = 3.1534 \times 10^8$$

Multiplication:

$$3.1415 \times 10^8 \times 1.19 \times 10^6 = (3.1415 \times 1.19) \times 10^{(8+6)}$$

Division:

$$3.1415 \times 10^8 \;/\; 1.19 \times 10^6 = (3.1415 \;/\; 1.19) \times 10^{(8-6)}$$

Biased exponent problem:
If a true exponent e is represented in excess-p notation, that is as e+p.
Then consider what happens under multiplication:

$$a.\ 10^{(x+p)} * b.\ 10^{(y+p)} = (a.b).\ 10^{(x+p+y+p)} = (a.b).\ 10^{(x+y+2p)}$$

Representing the result in excess-p notation implies that the exponent should be $x+y+p$. Instead it is $x+y+2p$.
So, subtract p after adding the biased exponents.

Floating point arithmetic: ADD/SUB rule

1. Choose the number with the smaller exponent.
2. Shift its mantissa right until the exponents of both the numbers are equal.
3. Add or subtract the mantissas.
4. Determine the sign of the result.
5. Normalize the result if necessary and truncate/round to the number of mantissa bits.

- Note: This does not consider the possibility of overflow/underflow

Floating point arithmetic: MUL rule

1. Add the exponents.
2. Subtract the bias.
3. Multiply the mantissas and determine the sign of the result.
4. Normalize the result (if necessary).
5. Truncate/round the mantissa of the result.

Floating point arithmetic: DIV rule

1. Subtract the exponents
2. Add the bias.
3. Divide the mantissas and determine the sign of the result.
4. Normalize the result if necessary.
5. Truncate/round the mantissa of the result.

- Note: Multiplication and division does not require alignment of the mantissas the way addition and subtraction does.

Why add the bias in step 2 ?:

$$\boxed{a.\ 2^{(x+p)} * b.\ 2^{(y+p)} = (a/b).\ 2^{(x+p-y-p)} = (a.b).\ 2^{(x+y)}}$$

Representing the result in excess-p notation implies that the exponent should be $x+y+p$. Instead it is $x+y$.
So, add p after subtracting the biased exponents.

Decoding a floating point number

- Sign indicated by first bit
- Subtract 127 from biased exponent to obtain power of two:

  <be> – 127
- Use coefficient to construct a normalized binary value with a binary point:

  1.<coefficient>
- Number being represented is

  $\mathbf{1.<\text{coefficient}> \times 2^{<\text{be}> - 127}}$

First example

0	01111111	00000000000000000000000

- Sign bit is zero:

  Number is positive
- Biased exponent is 127. Therefore actual exponent = (127-bias value)
- Bias value is 127; so actual exponent is 0

  $\mathbf{2^0}$
- Normalized mantissa is

  1.0000000….
- Number is $\mathbf{+1\times 2^0 = 1}$

Second example

0	10000000	10000000000000000000000

- Sign bit is zero:

  Number is positive
- Biased exponent is 128
- Therefore actual exponent = (128 - bias value) = 128-127 = 1

  $\mathbf{2^1}$
- Normalized binary value is

  $\mathbf{1.1000000\ldots = 1*2^0 + 1*2^{-1} = 1.5_{10}}$
- Number in decimalis $\mathbf{1.5 \times 2^1 = +3_{10}}$

Third example

1	01111110	11000000000000000000000

1011=B 1111=F 0110=6 0000 =0 0000=0 0000=0 0000=0 0000=0

=BF600000 $_{(16)}$ …….Representation of above IEEE number in Hexadecimal

- Sign bit is 1:

  Number is negative

- Biased exponent is 126
- Therefore actual exponent = (126 - bias value) = 126-127 = -1

  $\mathbf{2^{-1}}$

- Normalized binary value is

  $\mathbf{1.1100000\ldots \quad 1*2^{0} + 1*2^{-1} + 1*2^{-2} = 1 + 0.5 + 0.25 = 1.75_{10}}$

- Number is $\mathbf{-1.11 \times 2^{-1} = -(1.75 * 2^{-1}) = -(1.75/2) = -0.875_{10}}$

Given IEEE No =

~~417F~~ $417C0000_{(16)}$

Decimal eq = ________

1	8	23
0	100 0001 0	111 1100 0000 0000 0000 0000
S	E'	M.

(128) (2)

$E' = 130$

$E = 130 - 127 = \underline{\underline{3}}$

$M = 1.11111$

$\therefore M = 1.96875$

0.5
0.25
0.125
0.0625
0.03125
$\overline{0.96875}$

$\therefore$ given No $= +1.96875 \times 2^3$

~~≈ 2×8 ≈ 16~~

$= \underline{\underline{15.75}}$

$\underline{15.75000}$

What is decimal value of following IEEE single precision number.

Given no =

1	8	23
0	01111011	10011001100110011001100

$E' = 123$

$E = 123 - 127 = -4$

$M = 1.10011001100\cdots_{(2)}$

$M = 1.59961_{(10)}$ $\therefore$ Given no $= +1.59961 \times 2^{-4}$

$\simeq \underline{\underline{0.1}}$

Thank you

Additional Slides

Guard bits

While adding two floating point numbers with 24-bit mantissas, we shift the mantissa of the number with the smaller exponent to the right until the two exponents are equalized.

This implies that mantissa bits may be lost during the right shift (that is, bits of precision may be shifted out of the mantissa being shifted).

To prevent this, floating point operations are implemented by keeping guard bits, that is, extra bits of precision at the least significant end of the mantissa.

The arithmetic on the mantissas is performed with these extra bits of precision.

After an arithmetic operation, the guarded mantissas are:

- Normalized (if necessary)
- Converted back by a process called truncation/rounding to a 24-bit mantissa.

Truncation/rounding

- Straight chopping:
  - The guard bits (excess bits of precision) are dropped.
- Von Neumann rounding:
  - If the guard bits are all 0, they are dropped.
  - However, if any bit of the guard bit is a 1, then the LSB of the retained bit is set to 1.
- Rounding:
  - If there is a 1 in the MSB of the guard bit then a 1 is added to the LSB of the retained bits.

Rounding

- Rounding is evidently the most accurate truncation method.
- However,
  - Rounding requires an addition operation.
  - Rounding may require a renormalization, if the addition operation de-normalizes the truncated number.

- IEEE uses the rounding method.

0.111111100000 rounds to 0.111111 + 0.000001
=1.000000 which must be renormalized to 0.100000