

Arithmetic

UNIT 2

Addition/subtraction of signed numbers

x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \overline{x_i}y_i\overline{c_i} + x_i\overline{y_i}\overline{c_i} + x_iy_i\overline{c_i} + \overline{x_i}y_icy_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = y_icy_i + x_icy_i + x_iy_i$$

At the i^{th} stage:

Input:

c_i is the carry-in

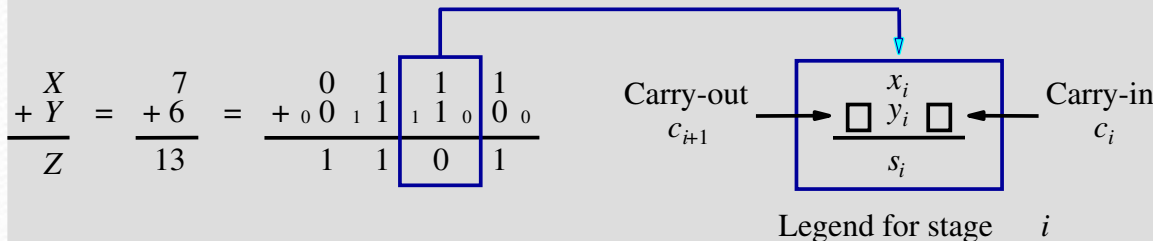
Output:

s_i is the sum

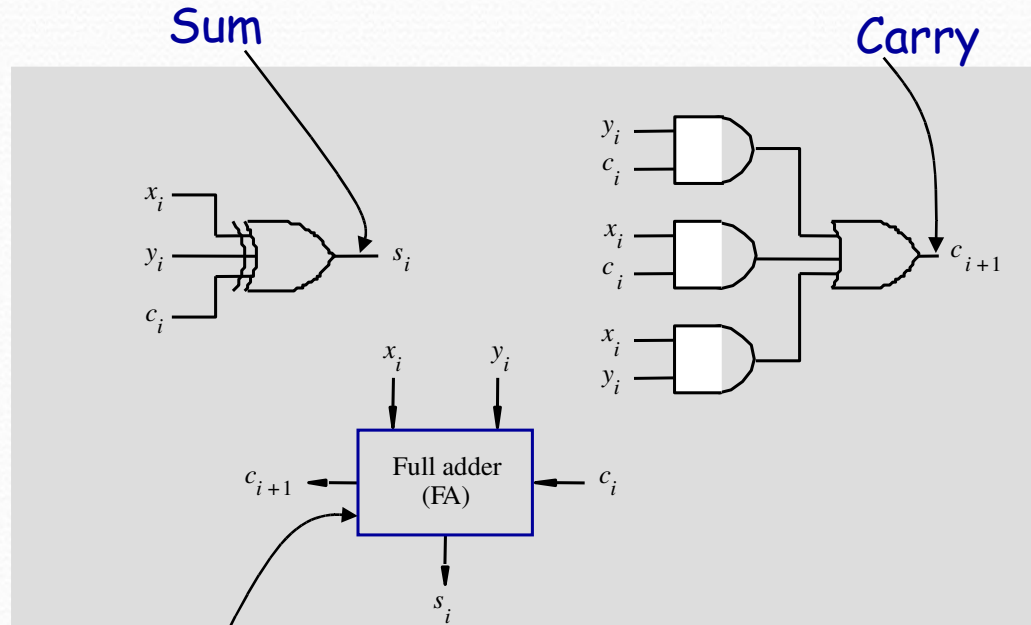
c_{i+1} carry-out to $(i+1)^{st}$

state

Example:



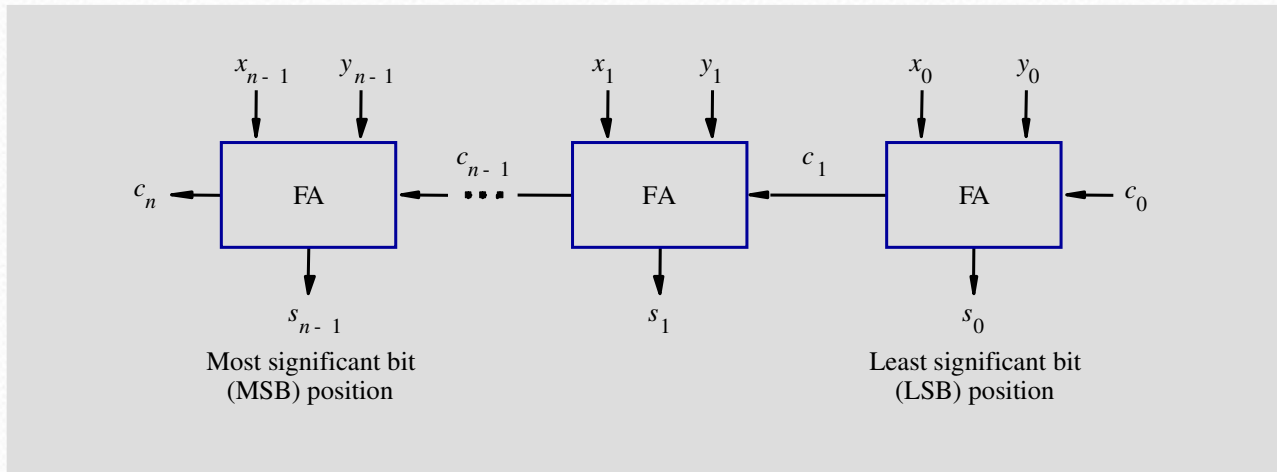
Addition logic for a single stage



Full Adder (FA): Symbol for the complete circuit for a single stage of addition.

n -bit adder

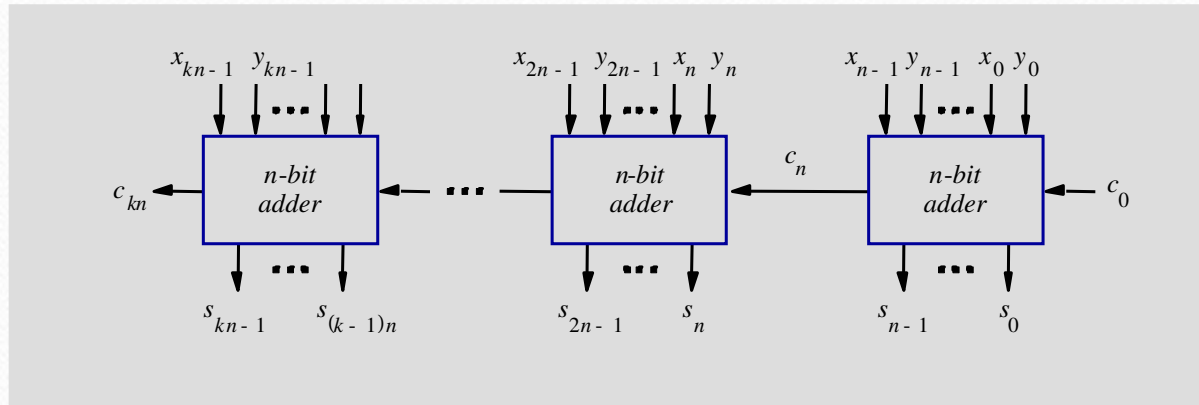
- Cascade n full adder (FA) blocks to form a n -bit adder.
- Carries propagate or ripple through this cascade, [\$n\$ -bit ripple carry adder.](#)



Carry-in c_0 into the LSB position provides a convenient way to perform subtraction.

K n -bit adder

K n -bit numbers can be added by cascading k n -bit adders.

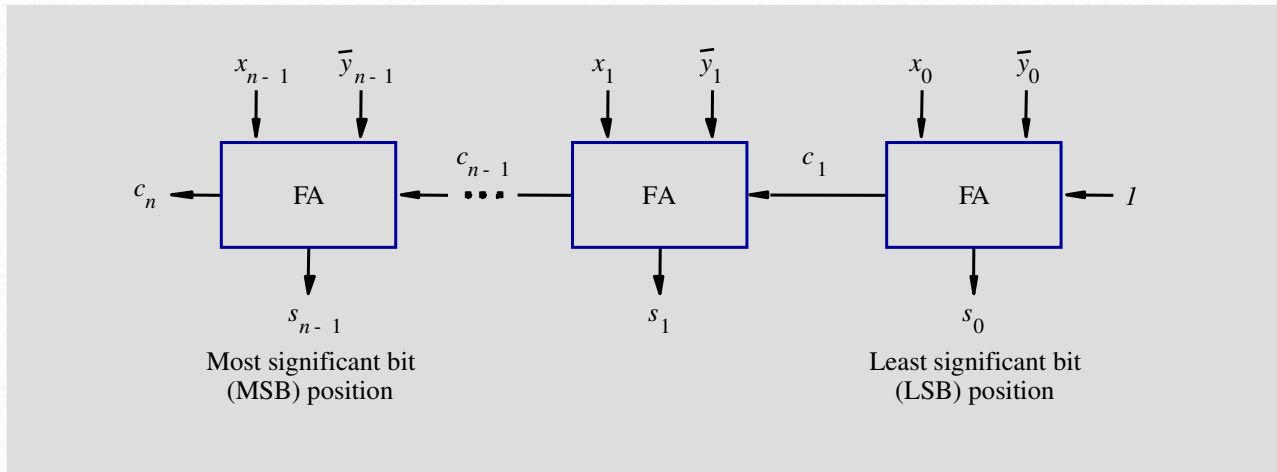


Each n -bit adder forms a block, so this is cascading of blocks.

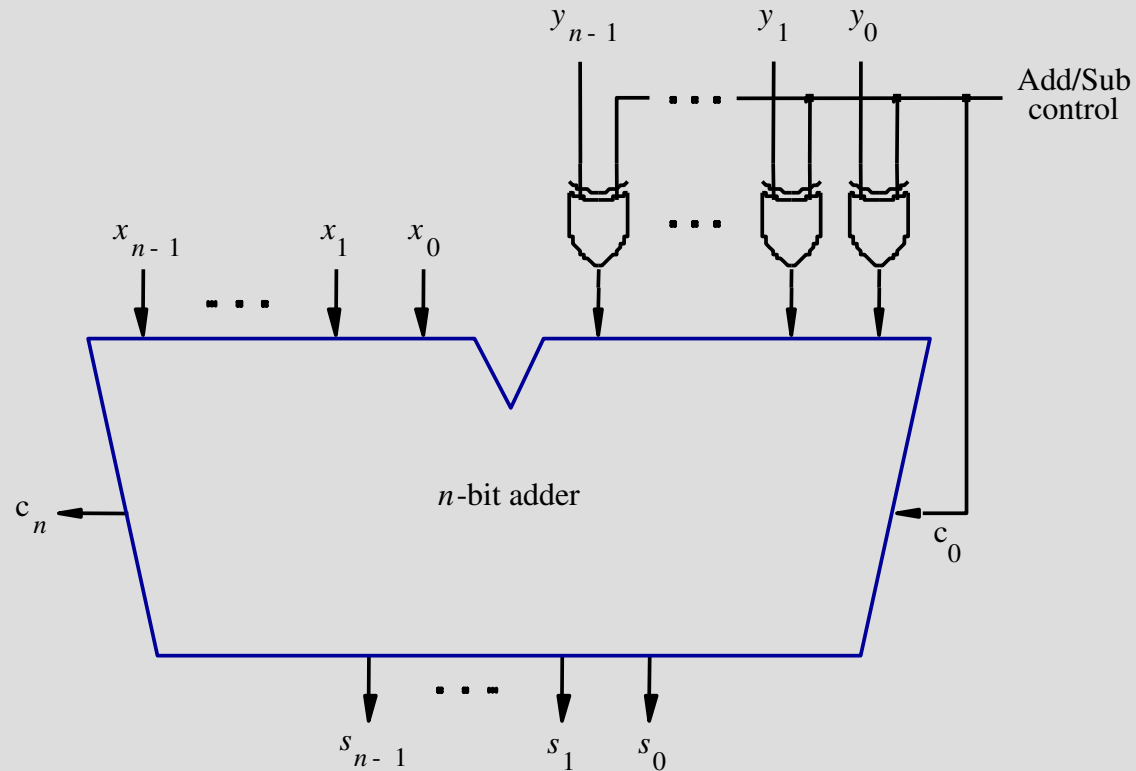
Carries ripple or propagate through blocks, [Blocked Ripple Carry Adder](#)

n -bit subtractor

- Recall $X - Y$ is equivalent to adding 2's complement of Y to X .
- 2's complement is equivalent to 1's complement + 1.
- $X - Y = X + \bar{Y} + 1$
- 2's complement of positive and negative numbers is computed similarly.



n -bit adder/subtractor (contd..)



- Add/sub control = 0, addition.
- Add/sub control = 1, subtraction.

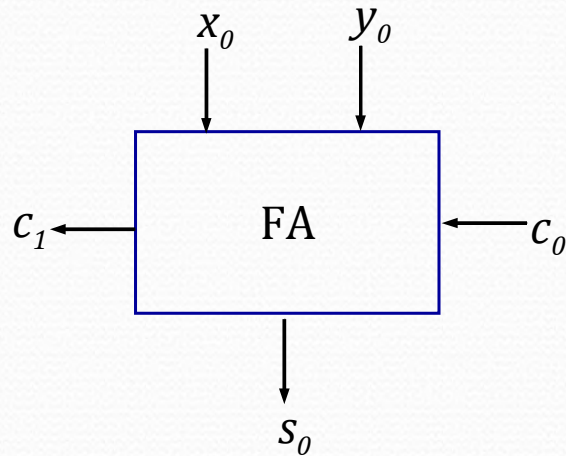
Detecting overflows

- Overflows can only occur when the sign of the two operands is the same.
- Overflow occurs if the sign of the result is different from the sign of the operands.
- Recall that the MSB represents the sign.
 - x_{n-1} , y_{n-1} , s_{n-1} represent the sign of operand x , operand y and result s respectively.
- Circuit to detect overflow can be implemented by the following logic expressions:

$$\text{Overflow} = x_{n-1}y_{n-1}\bar{s}_{n-1} + \bar{x}_{n-1}\bar{y}_{n-1}s_{n-1}$$

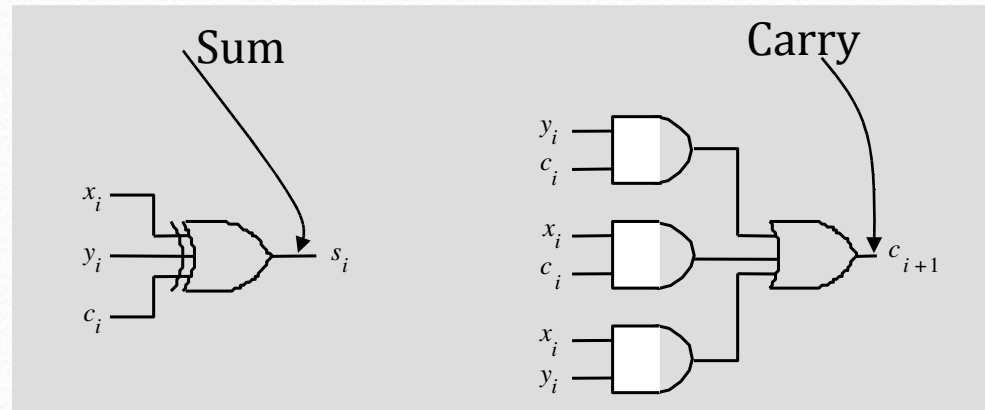
$$\text{Overflow} = c_n \oplus c_{n-1}$$

Computing the add time



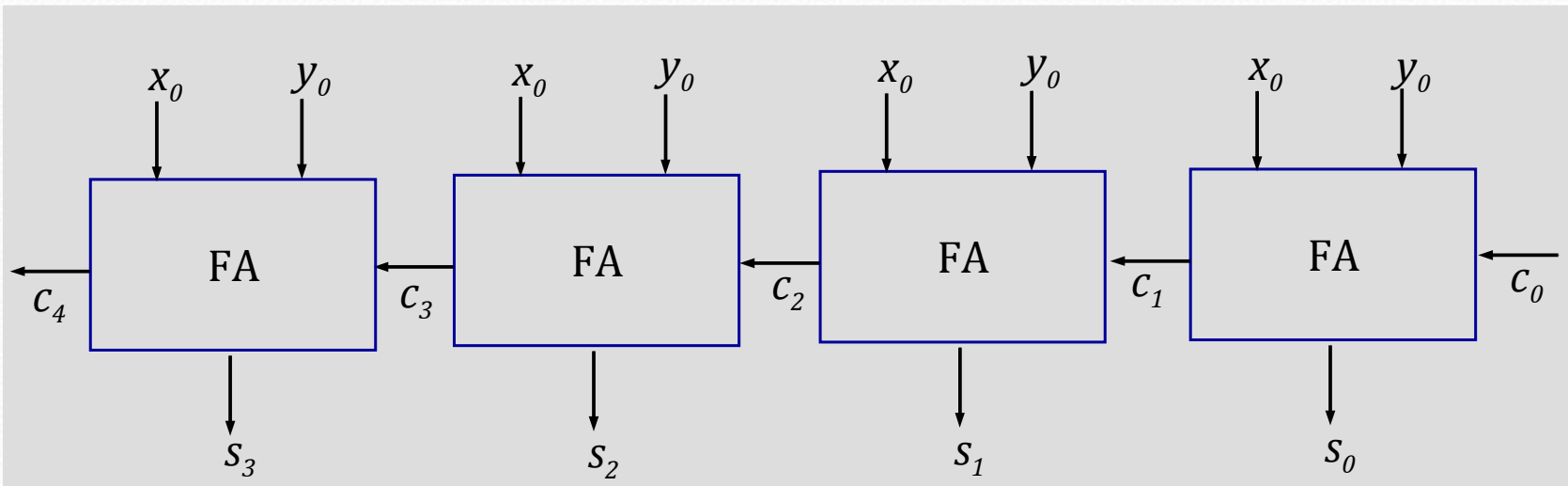
Consider 0^{th} stage:

- c_1 is available after 2 gate delays.
- s_1 is available after 1 gate delay.



Computing the add time (contd..)

Cascade of 4 Full Adders, or a 4-bit adder



- s_0 available after 1 gate delays, c_1 available after 2 gate delays.
- s_1 available after 3 gate delays, c_2 available after 4 gate delays.
- s_2 available after 5 gate delays, c_3 available after 6 gate delays.
- s_3 available after 7 gate delays, c_4 available after 8 gate delays.

For an n -bit adder, s_{n-1} is available after $2n-1$ gate delays

c_n is available after $2n$ gate delays.

Fast addition

Recall the equations:

$$s_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Second equation can be written as:

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

We can write:

$$c_{i+1} = G_i + P_i c_i$$

$$\text{where } G_i = x_i y_i \text{ and } P_i = x_i + y_i$$

- G_i is called generate function and P_i is called propagate function
- G_i and P_i are computed only from x_i and y_i and not c_i , thus they can be computed in one gate delay after X and Y are applied to the inputs of an n -bit adder.

Carry lookahead

$$c_{i+1} = G_i + P_i c_i$$

$$c_i = G_{i-1} + P_{i-1} c_{i-1}$$

$$\Rightarrow c_{i+1} = G_i + P_i (G_{i-1} + P_{i-1} c_{i-1})$$

continuing

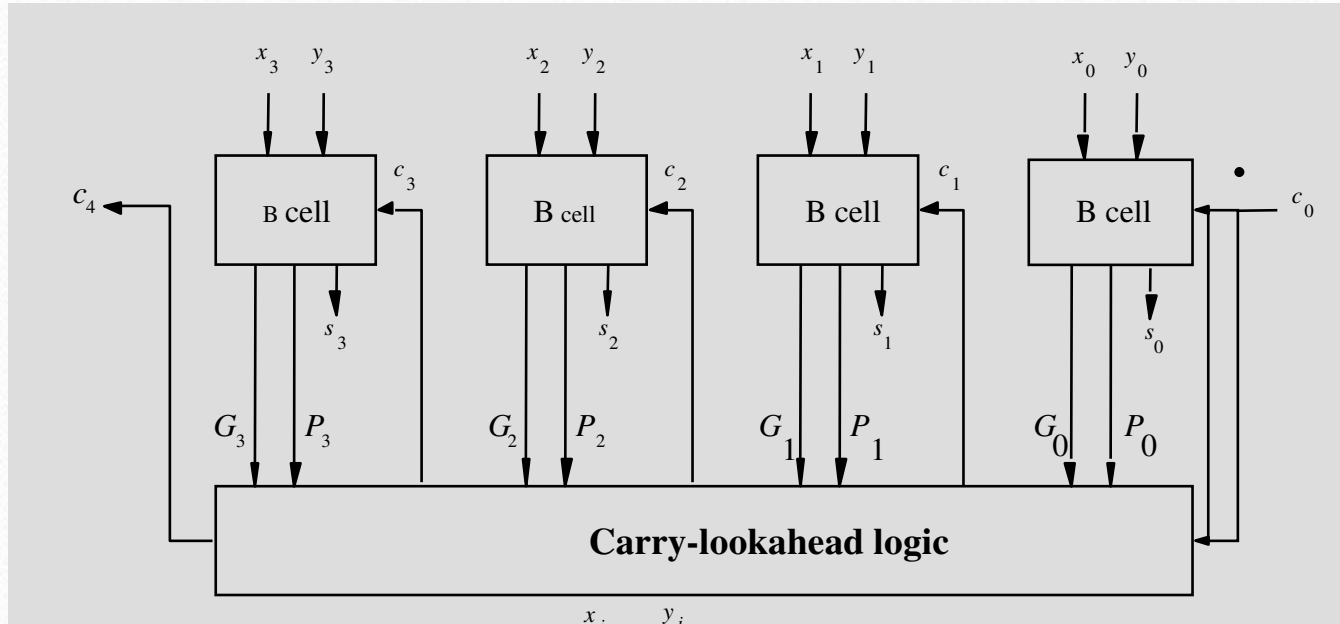
$$\Rightarrow c_{i+1} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} c_{i-2}))$$

until

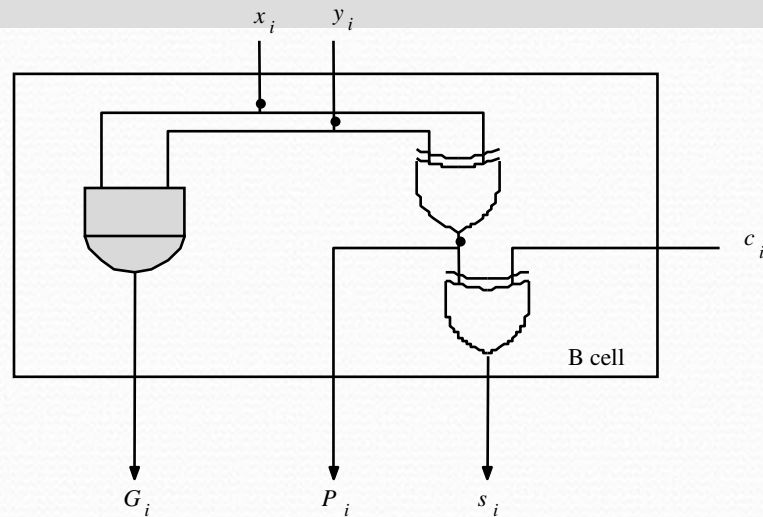
$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \dots + P_i P_{i-1} \dots P_1 G_0 + P_i P_{i-1} \dots P_0 c_0$$

- All carries can be obtained 3 gate delays after X , Y and c_0 are applied.
 - One gate delay for P_i and G_i
 - Two gate delays in the AND-OR circuit for c_{i+1}
- All sums can be obtained 1 gate delay after the carries are computed.
- Independent of n , n -bit addition requires only 4 gate delays.
- This is called Carry Lookahead adder.

Carry-lookahead adder



**4-bit
carry-lookahead
adder**



B-cell for a single stage

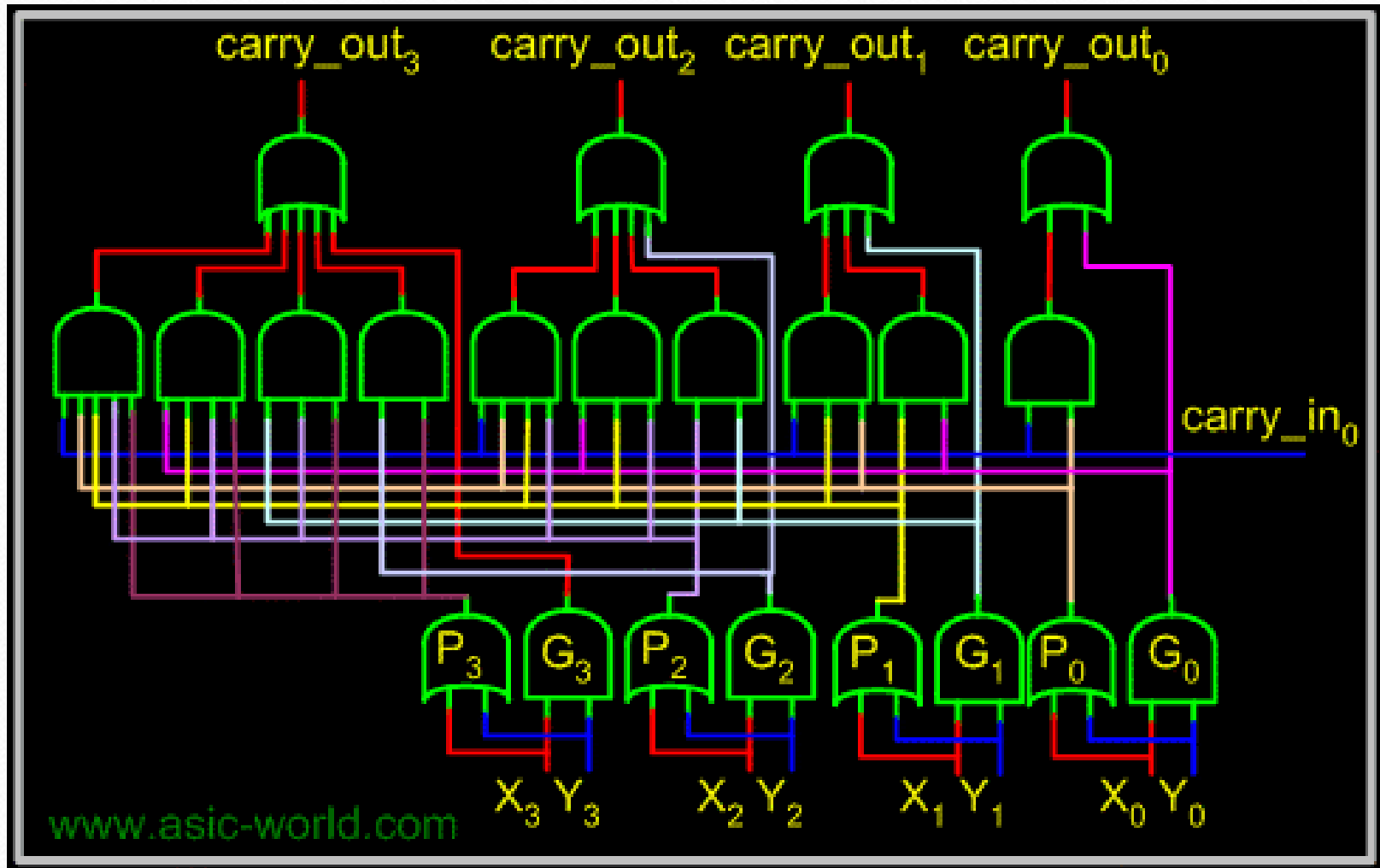
Carry lookahead adder (contd..)

- Performing n -bit addition in 4 gate delays independent of n is good only theoretically because of fan-in constraints.

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \dots + P_i P_{i-1} \dots P_1 G_0 + P_i P_{i-1} \dots P_0 c_0$$

- Last AND gate and OR gate require a fan-in of $(n+1)$ for a n -bit adder.
 - For a 4-bit adder ($n=4$) fan-in of 5 is required.
 - Practical limit for most gates.
- In order to add operands longer than 4 bits, we can cascade 4-bit Carry-Lookahead adders. Cascade of Carry-Lookahead adders is called Blocked Carry-Lookahead adder.

4-bit carry-lookahead Adder



Blocked Carry-Lookahead adder

Carry-out from a 4-bit block can be given as:

$$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$

Rewrite this as:

$$P_0^I = P_3P_2P_1P_0$$

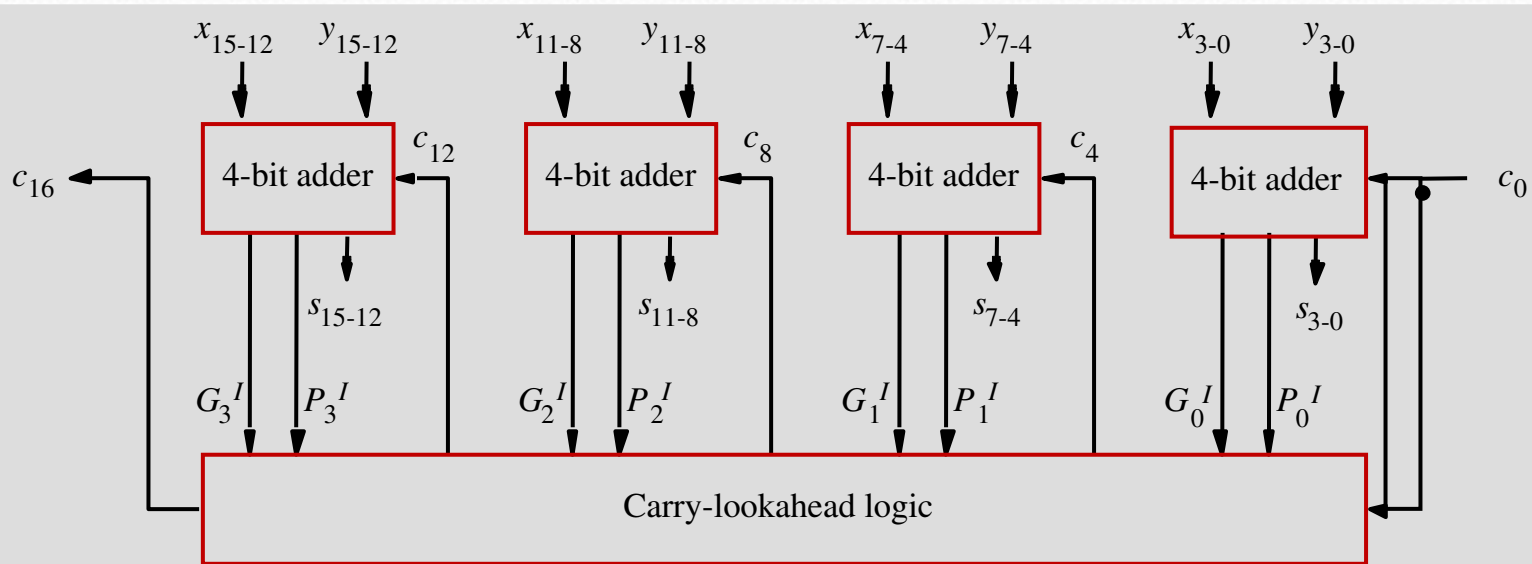
$$G_0^I = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$

Subscript I denotes the blocked carry lookahead and identifies the block.

Cascade 4 4-bit adders, c_{16} can be expressed as:

$$c_{16} = G_3^I + P_3^I G_2^I + P_3^I P_2^I G_1^I + P_3^I P_2^I P_1^0 G_0^I + P_3^I P_2^I P_1^0 P_0^0 c_0$$

Blocked Carry-Lookahead adder



After x_i, y_i and c_0 are applied as inputs:

- G_i and P_i for each stage are available after 1 gate delay.
- P^I is available after 2 and G^I after 3 gate delays.
- All carries are available after 5 gate delays.
- c_{16} is available after 5 gate delays.
- s_{15} which depends on c_{12} is available after 8 (5+3) gate delays
(Recall that for a 4-bit carry lookahead adder, the last sum bit is available 3 gate delays after all inputs are available)

Multiplication

Multiplication of unsigned numbers

$$\begin{array}{r} 1\ 1\ 0\ 1 \quad (13) \text{ Multiplicand M} \\ \cdot 1\ 0\ 1\ 1 \quad (11) \text{ Multiplier Q} \\ \hline 1\ 1\ 0\ 1 \\ 1\ 1\ 0\ 1 \\ 0\ 0\ 0\ 0 \\ 1\ 1\ 0\ 1 \\ \hline 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \quad (143) \text{ Product P} \end{array}$$

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

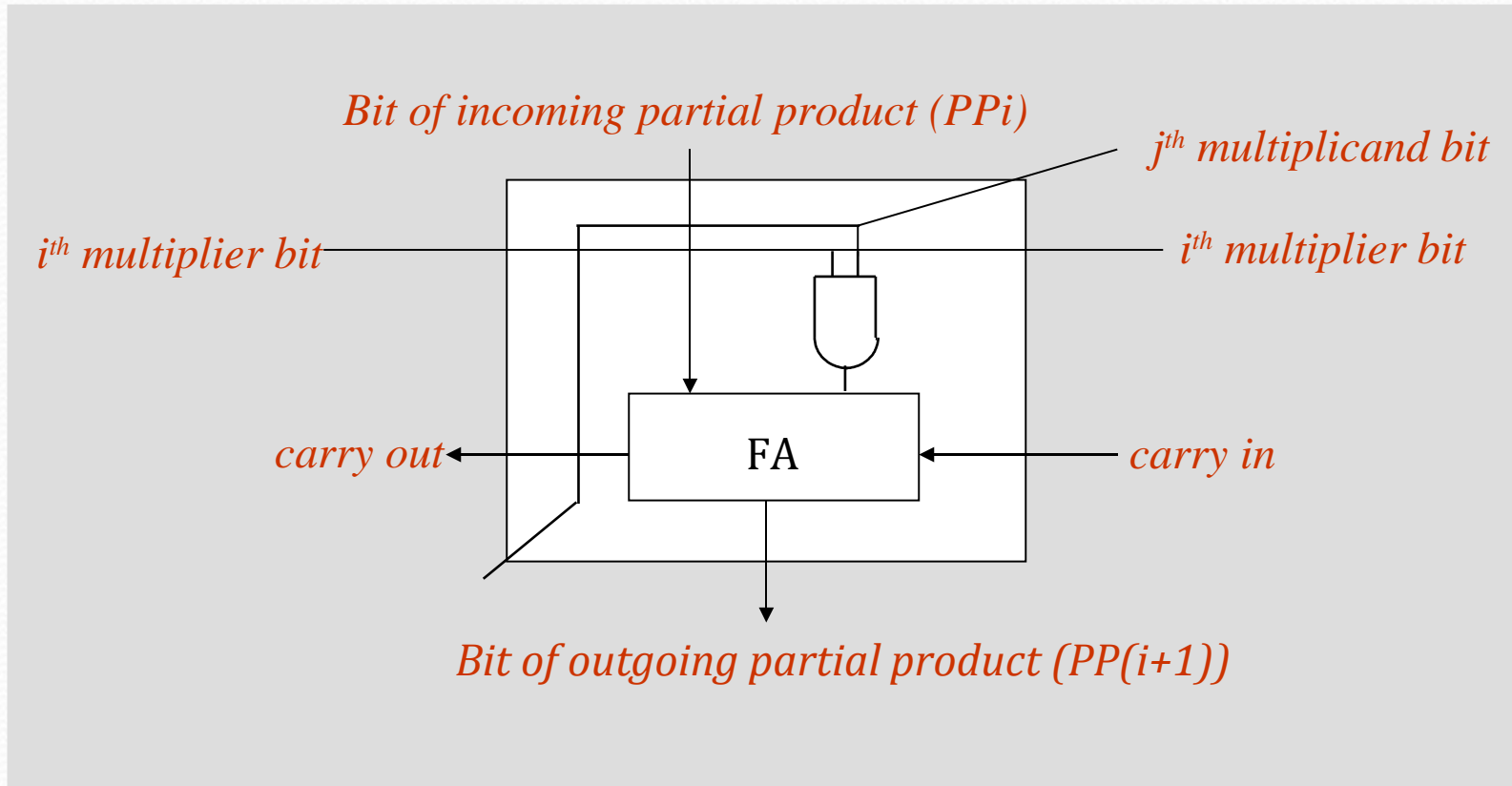
Unsigned multiplication can be viewed as addition of shifted versions of the multiplicand.

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.

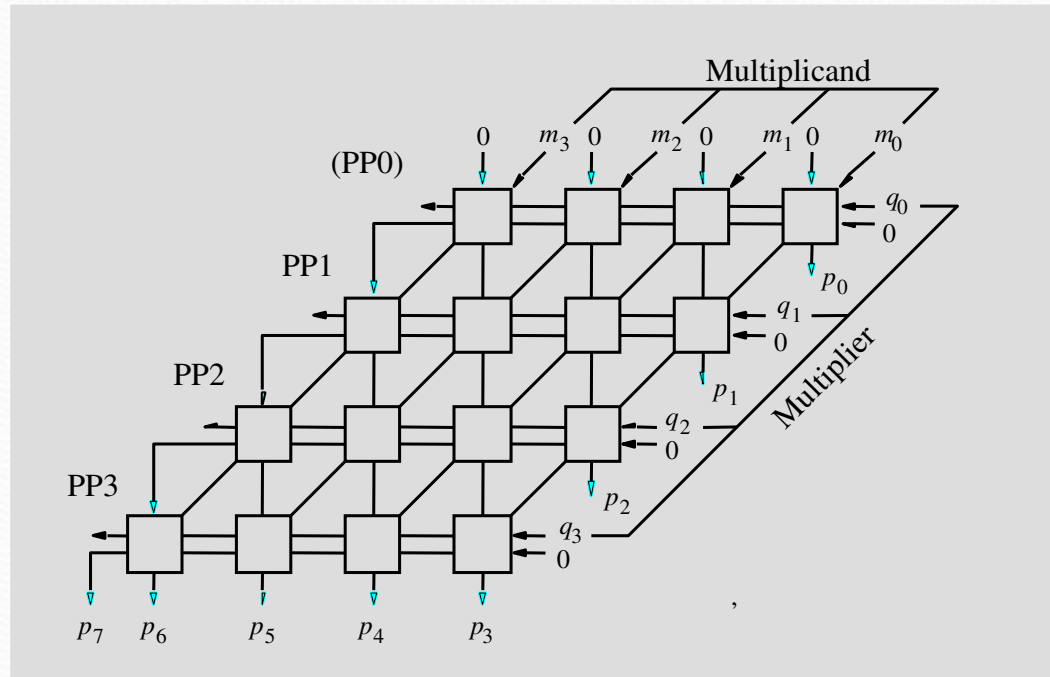
Multiplication of unsigned numbers

Typical multiplication cell



Combinatorial array multiplier

Combinatorial array multiplier



Product is: $p_7 p_6 \dots p_0$

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

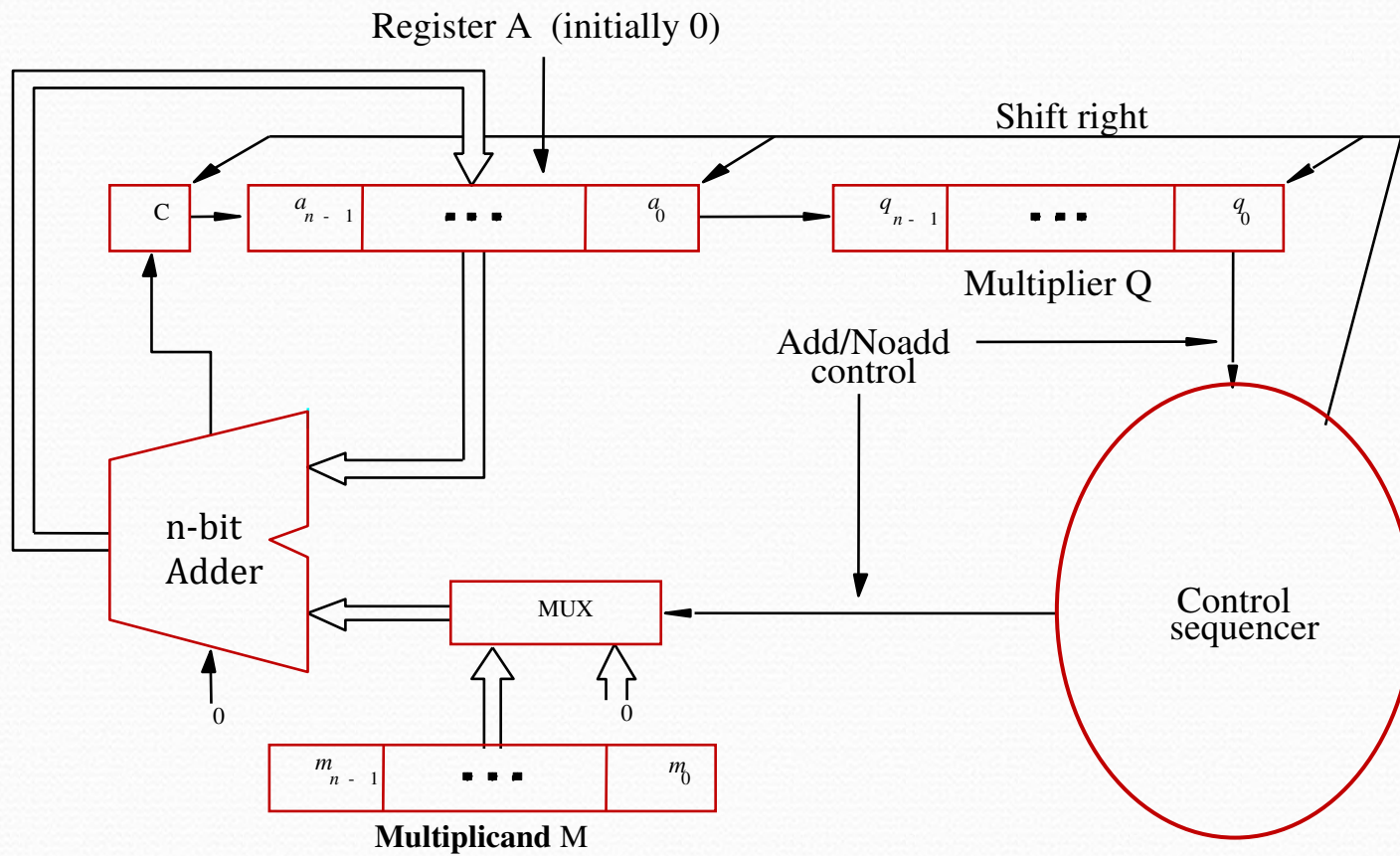
Combinatorial array multiplier (contd..)

- 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 combinatorial array techniques and sequential techniques requiring less combinational logic.

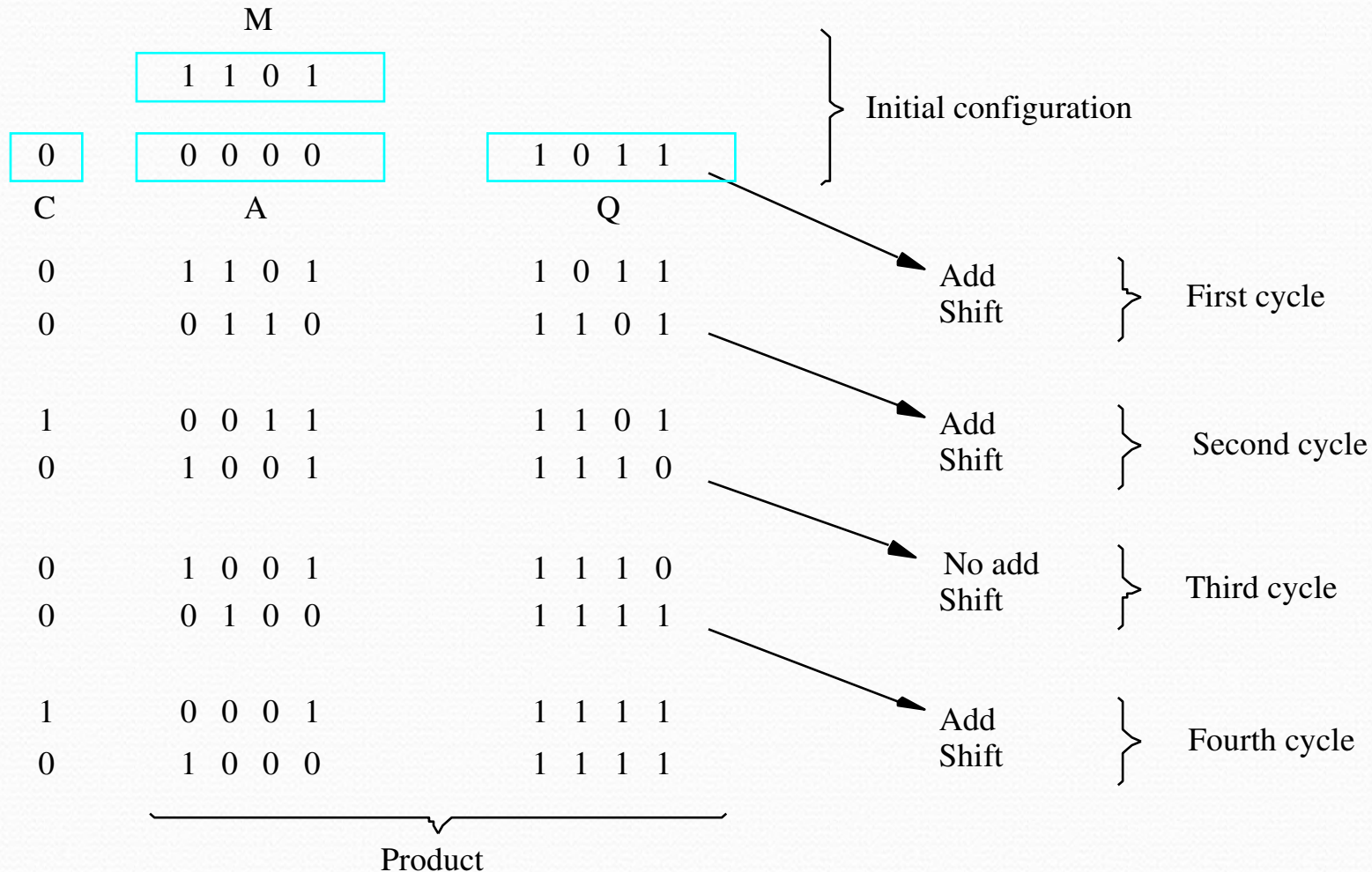
Sequential multiplication

- 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 left 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 (contd..)



Signed Multiplication

Signed Multiplication

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

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

Sign extension is shown in blue

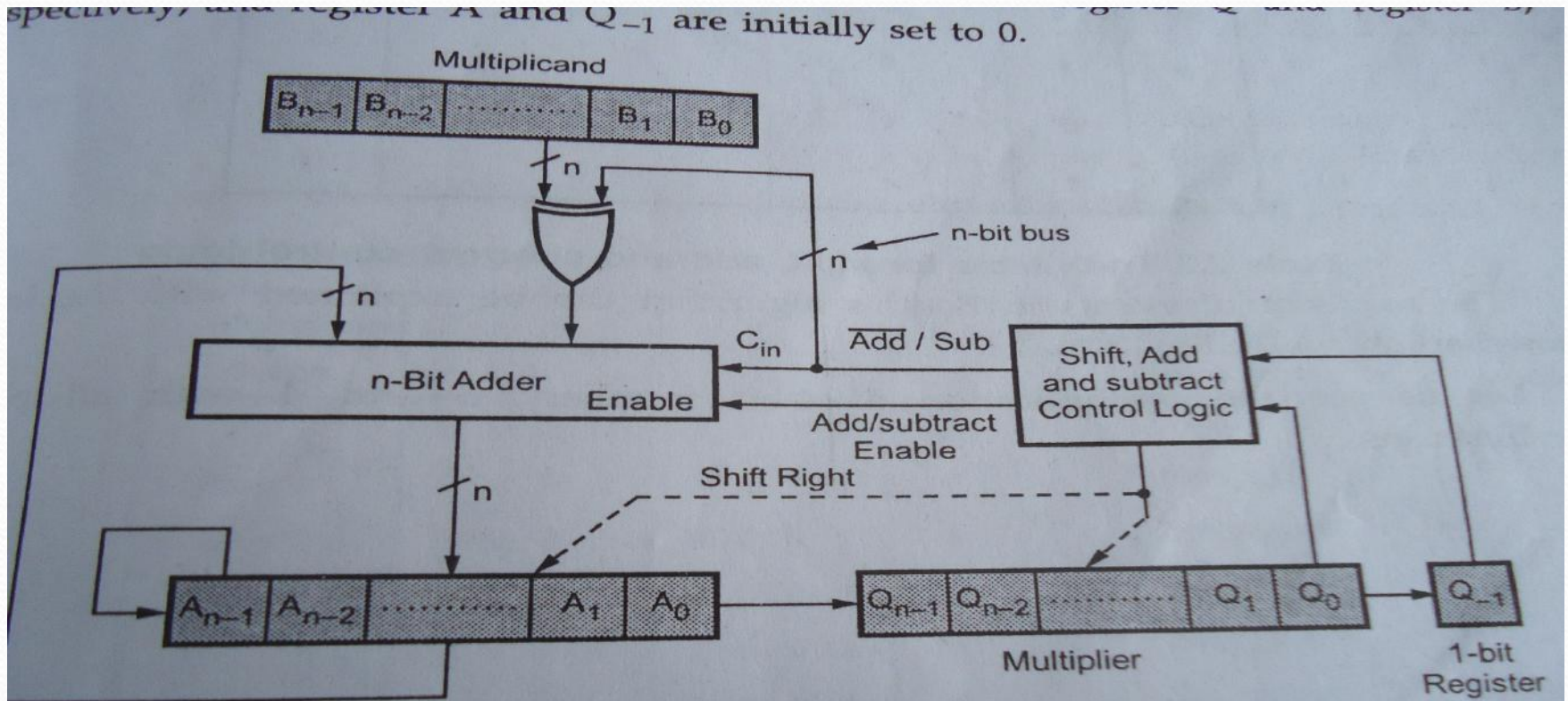
Sign extension of negative multiplicand.

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.

Signed Multiplication

Hardware Implementation



- Initial settings : A \leftarrow 0 and $Q_{-1} = 0$

Fig. 2.23 Hardware implementation of signed binary multiplication

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?

$$\begin{array}{r}
 0011110 \\
 00+1+1+1+1+10 \\
 \hline
 000000000 \\
 0101101 \\
 0101101 \\
 0101101 \\
 00000000 \\
 00000000 \\
 \hline
 000101010000110
 \end{array}$$

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	0
1	1	1	1	1	1	1	1	0	1	0	0	1	1	←
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	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	0	1	0	1	0
								0	1	0	0	0	1	1
								0	0	0	1	1	0	

2's complement of the multiplicand

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
									↓								
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

$$\begin{array}{r}
 0\ 1\ 1\ 0\ 1\ \text{(+13)} \\
 \times 1\ 1\ 0\ 1\ 0\ \text{(-6)} \\
 \hline
 \end{array}
 \quad \Rightarrow \quad
 \begin{array}{r}
 0\ 1\ 1\ 0\ 1 \\
 0\ -1\ +1\ -1\ 0 \\
 \hline
 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
 1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1 \\
 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1 \\
 1\ 1\ 1\ 0\ 0\ 1\ 1 \\
 0\ 0\ 0\ 0\ 0\ 0 \\
 \hline
 1\ 1\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 0\ \text{(-78)}
 \end{array}$$

Booth multiplication with a negative multiplier.

Booth Algorithm

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


Booth multiplier recoding table.

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



Booth Algorithm

Flow Chart

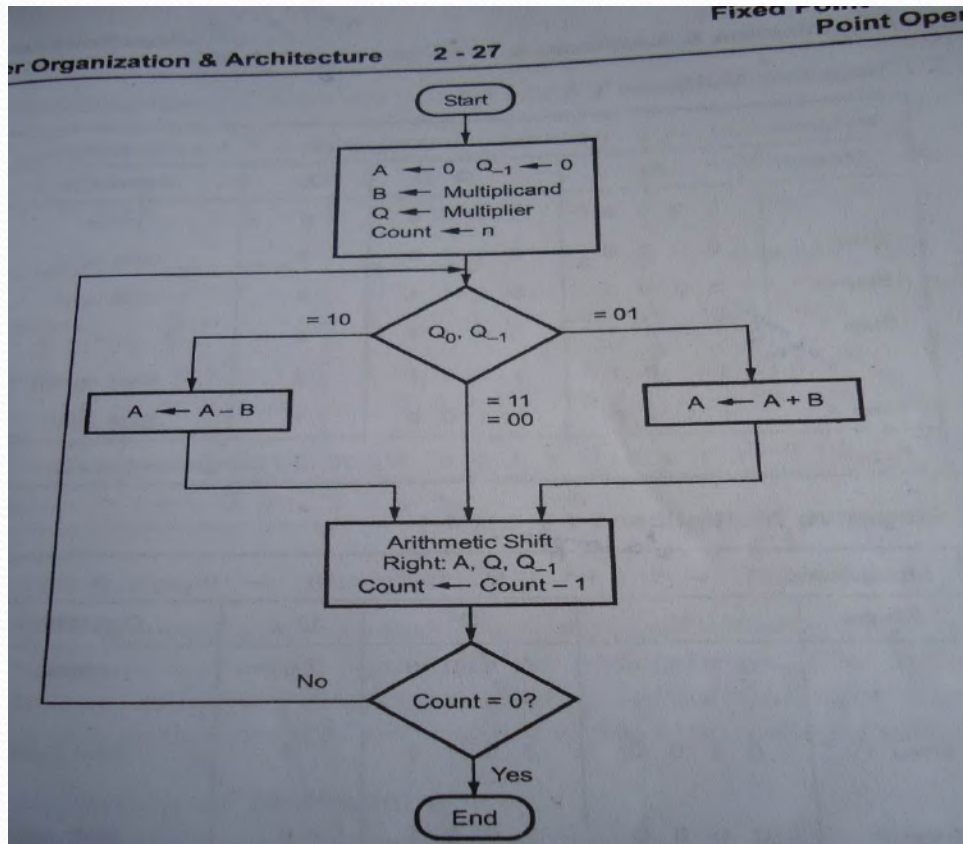


Fig. 2.24 Booth's algorithm for signed multiplication

Table

subtraction right shift occurs such that the leftmost bit of the multiplier is shifted into A_{n-2} , but also remains in A_{n-1} . This is required to preserve the sign bit in A and Q. It is known as an *arithmetic shift*, since it pre-

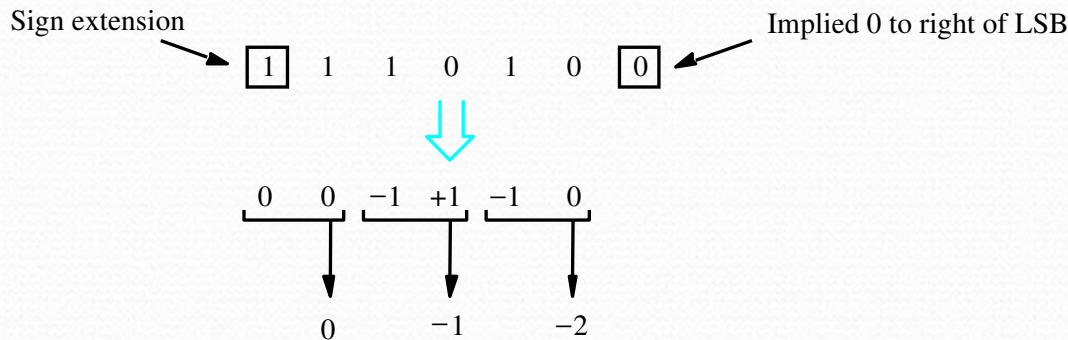
Q_0	Q_{-1}	Add/sub	Add/Subtract Enable	Shift
0	0	x	0	1
0	1	0	1	1
1	0	1	1	1
1	1	x	0	1

Fig. 2.3 Truth table for shift, add and subtract control logic. The sequence of events in Booth's algorithm can be explained as follows:

Fast Multiplication

Bit-Pair Recoding of Multipliers

- Bit-pair recoding halves the maximum number of summands (versions of the multiplicand).



(a) Example of bit-pair recoding derived from Booth recoding

Bit-Pair Recoding of Multipliers

Multiplier bit-pair		Multiplier bit on the right $i - 1$	Multiplicand selected at position i
$i + 1$	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

(b) Table of multiplicand selection decisions

Bit-Pair Recoding of Multipliers

$$\begin{array}{r}
 0\ 1\ 1\ 0\ 1\ (+13) \\
 \times 1\ 1\ 0\ 1\ 0\ (-6) \\
 \hline
 \end{array}$$



$$\begin{array}{r}
 0\ 1\ 1\ 0\ 1 \\
 0\ -1\ +1\ -1\ 0 \\
 \hline
 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
 1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1 \\
 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1 \\
 1\ 1\ 1\ 0\ 0\ 1\ 1 \\
 0\ 0\ 0\ 0\ 0\ 0 \\
 \hline
 1\ 1\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 0\ (-78)
 \end{array}$$

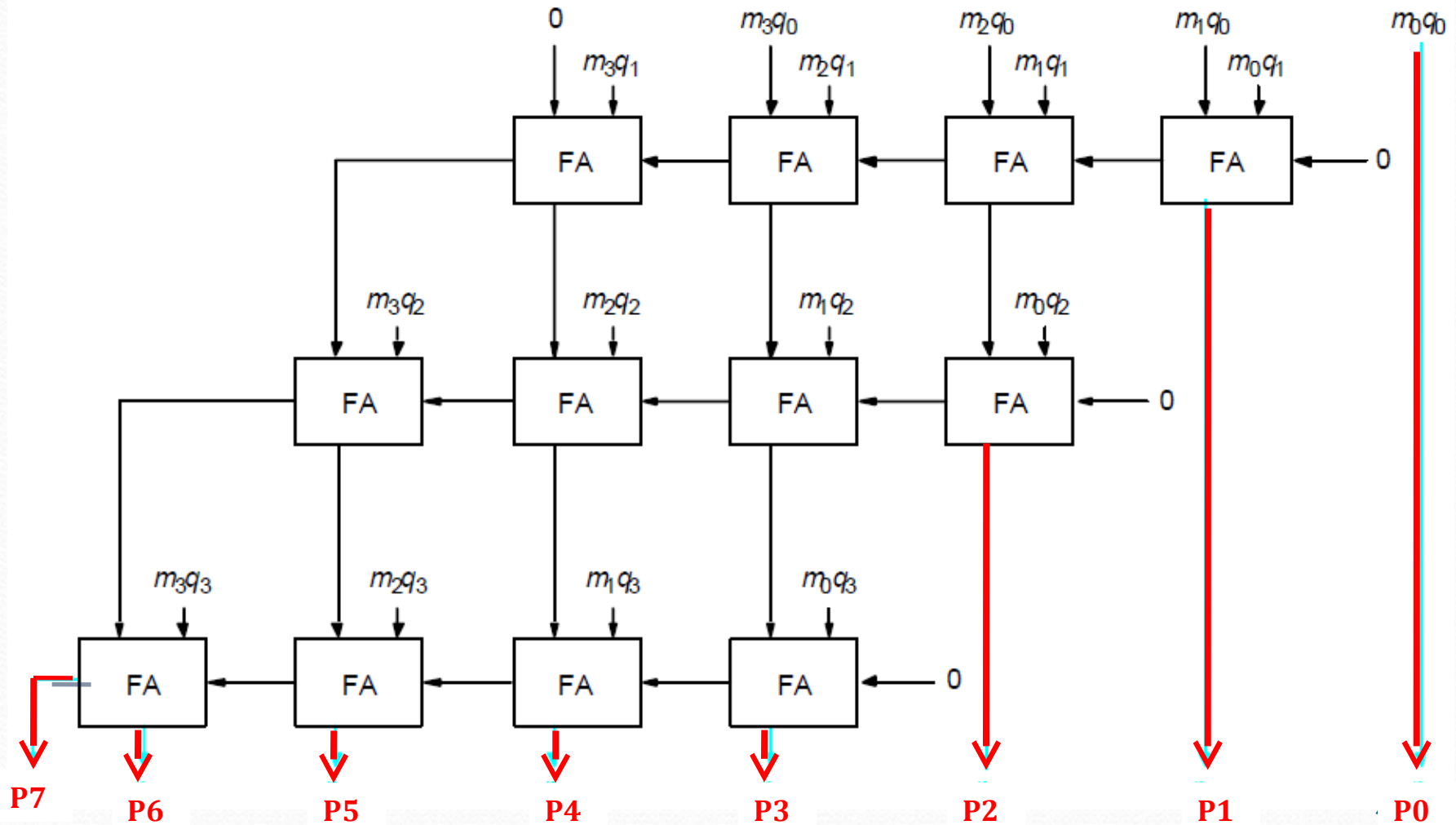


$$\begin{array}{r}
 0\ 1\ 1\ 0\ 1 \\
 0\ -1\ -2 \\
 \hline
 1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 0 \\
 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1 \\
 0\ 0\ 0\ 0\ 0\ 0 \\
 \hline
 1\ 1\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 0
 \end{array}$$

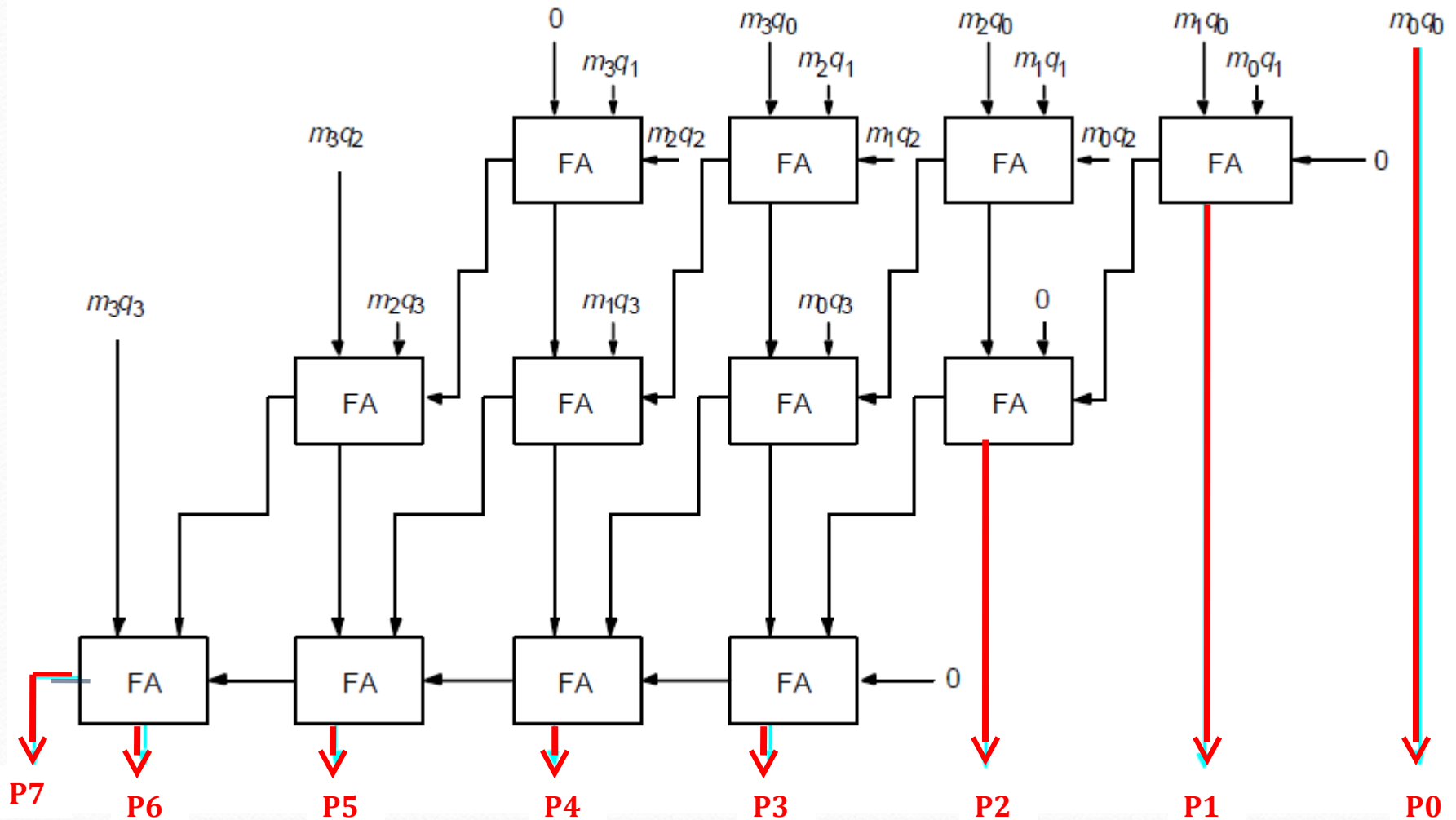
Figure 6.15. Multiplication requiring only $n/2$ summands.⁴¹

Carry-Save Addition of Summands

- CSA speeds up the addition process.



Carry-Save Addition of Summands(Cont.,)



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 RCA or CLA to produce the desired product

Integer Division

Manual Division

$$\begin{array}{r} 21 \\ 13 \overline{) 274} \\ \underline{26} \\ 14 \\ \underline{13} \\ 1 \end{array}$$

$$\begin{array}{r} 10101 \\ 1101 \overline{) 100010010} \\ \underline{1101} \\ 10000 \\ \underline{1101} \\ 1110 \\ \underline{1101} \\ 1 \end{array}$$

Longhand division examples.

Longhand Division Steps

- Position the divisor appropriately with respect to the dividend and performs 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

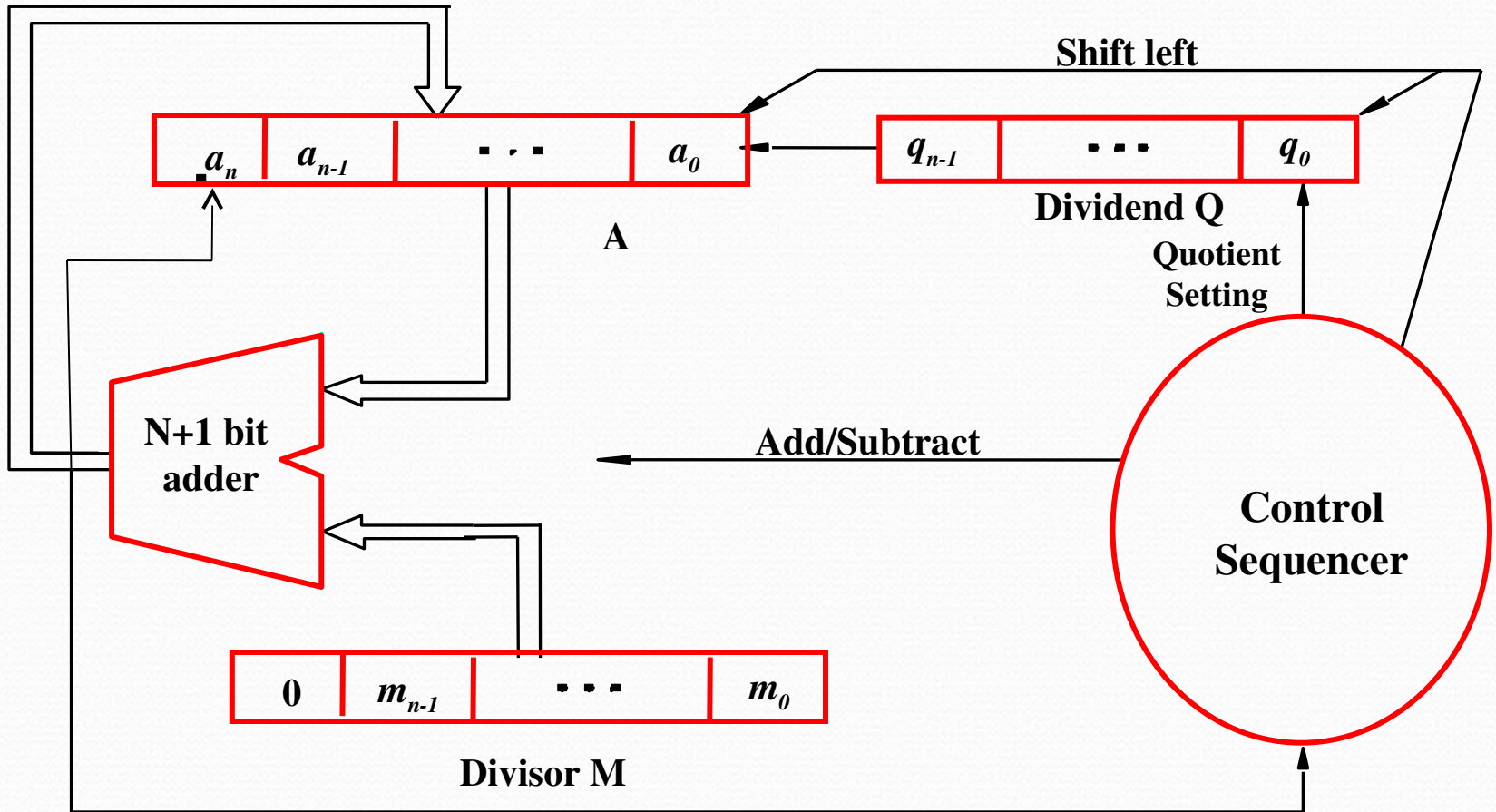


Figure 6.21. Circuit arrangement for binary division.

Binary Division

Hardware Implementation

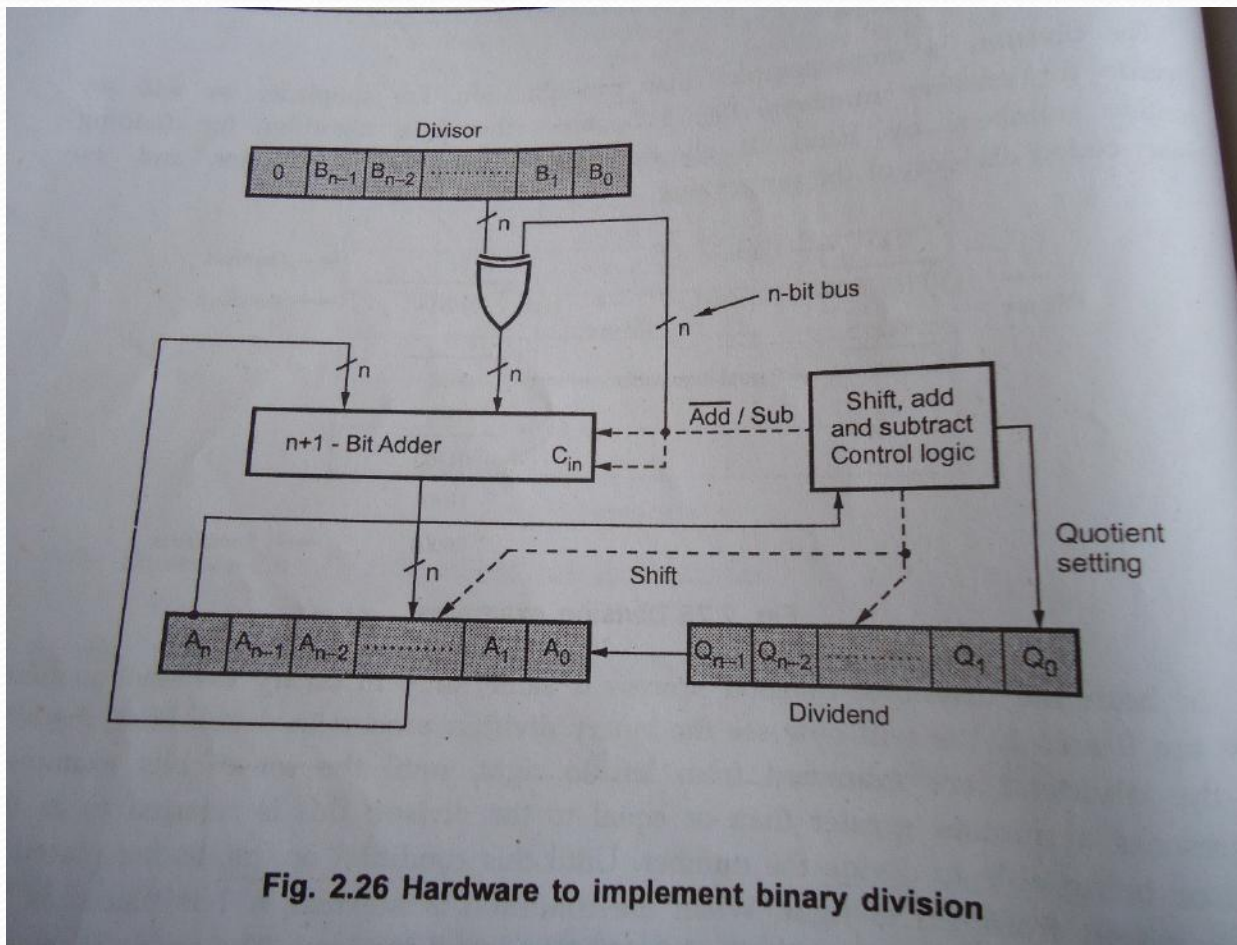


Fig. 2.26 Hardware to implement binary division

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

$$\begin{array}{r}
 10 \\
 11 \overline{) 1000} \\
 \underline{11} \\
 10
 \end{array}$$

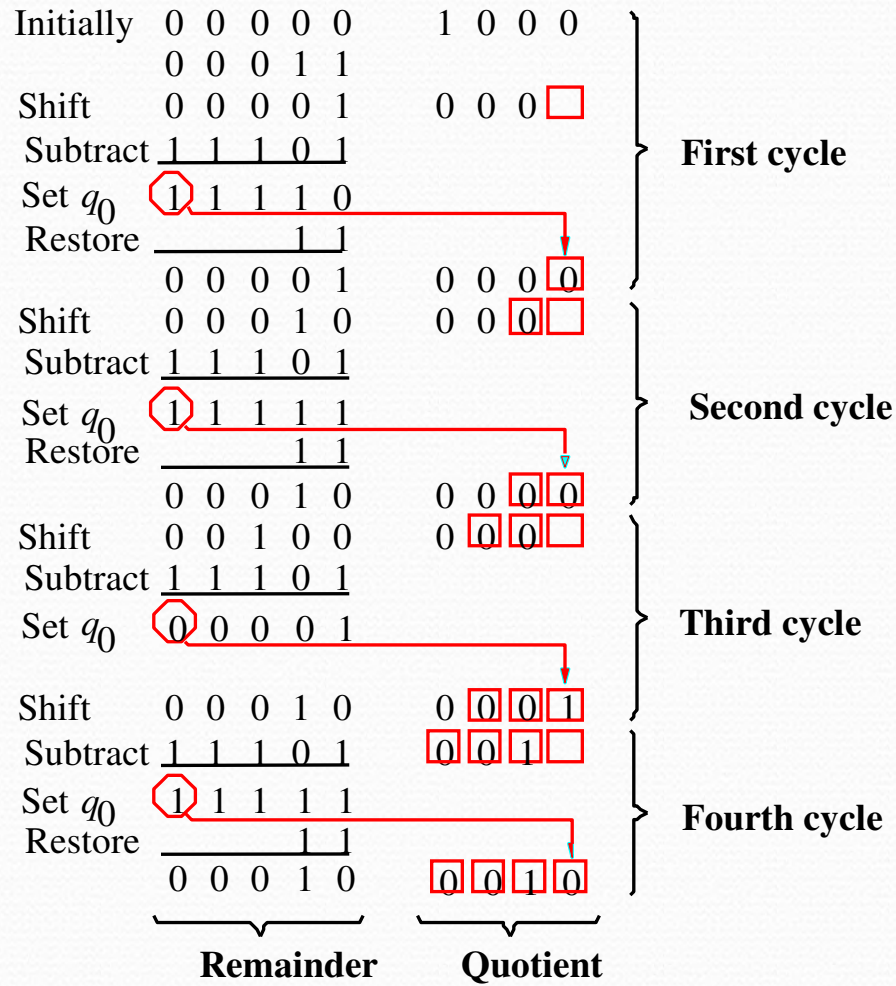


Figure 6.22. A restoring-division example.

Restoring Division

Flow Chart

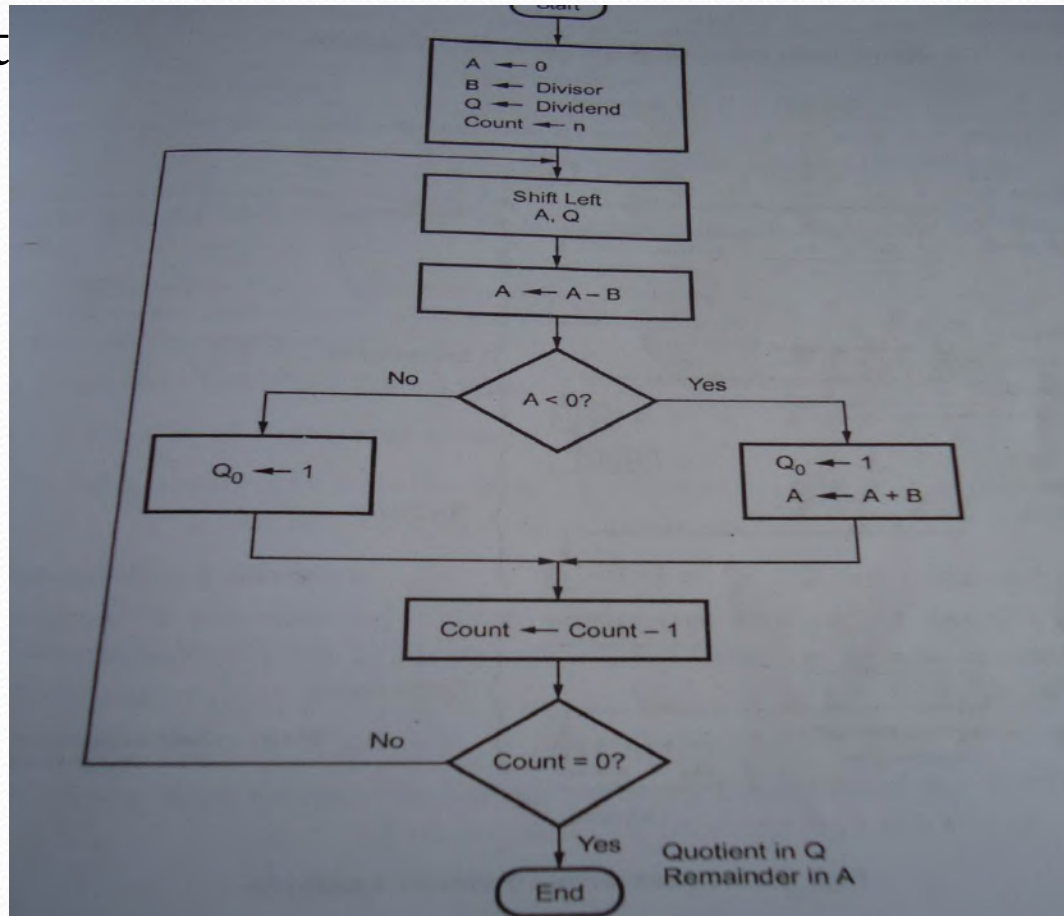
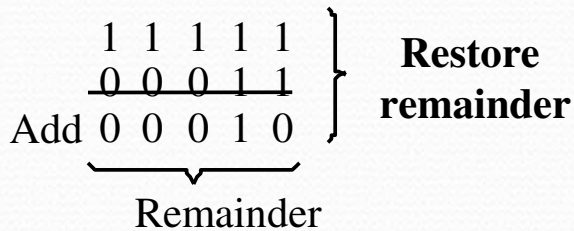
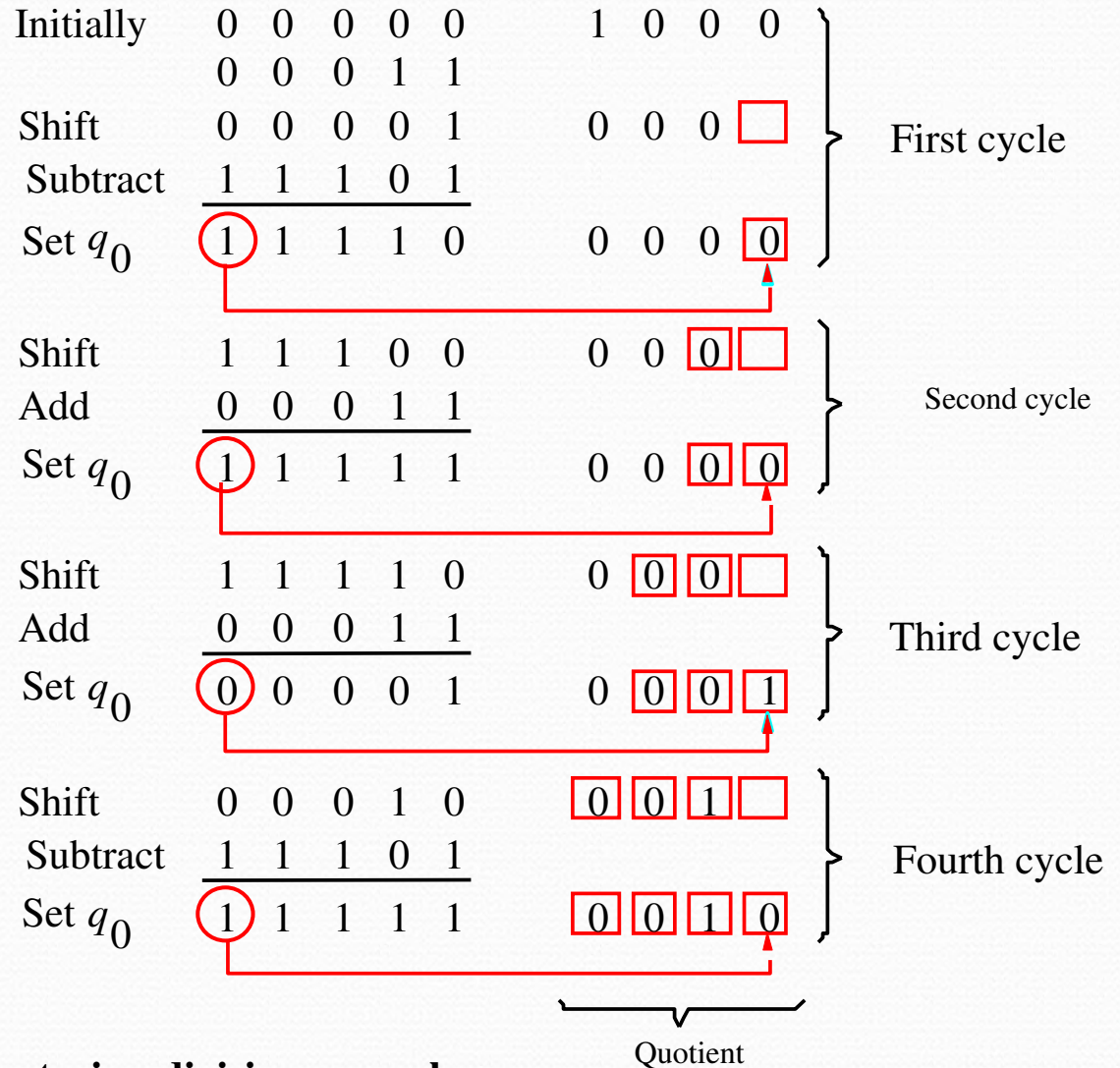


Fig. 2.27 Flowchart for restoring division operation

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



A nonrestoring-division example.

Quotient

Nonrestoring Division

Flow Chart

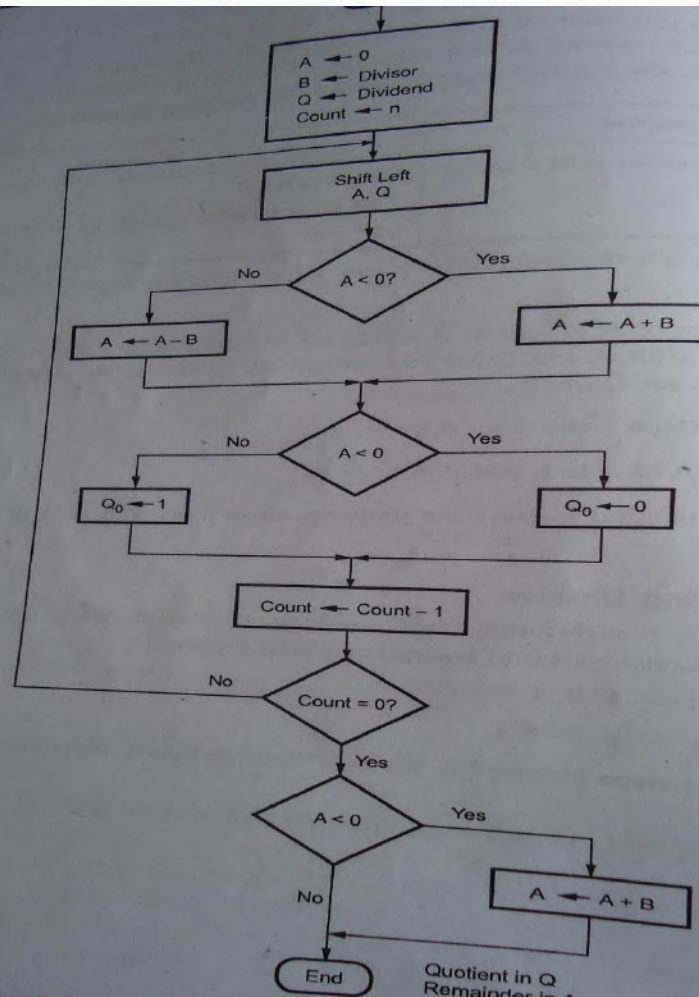


Fig. 2.29 Flowchart for non-restoring division operation

Floating-Point Numbers and Operations

Fractions

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

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

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

$$b_{31}b_{30}b_{29}\dots\dots\dots b_1b_0 \quad \text{implicit binary point}$$

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

$$\text{implicit binary point} \quad .b_{31}b_{30}b_{29}\dots\dots\dots b_1b_0$$

The value of b is then given by:

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

Range of fractions

The value of the unsigned binary fraction is:

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

The range of the numbers represented in this format is:

$$0 \leq V(b) \leq 1 - 2^{-32} \approx 0.99999999998$$

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.
- Fixed point representation suffers from a drawback that the representation can only represent a finite range (and quite small) range of numbers.

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

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

Components of these numbers are:

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

Significant digits

A number such as the following is said to have 7 significant digits

$$x = \pm 0.m_1m_2m_3m_4m_5m_6m_7 \times b^{\pm e}$$

Fractions in the range 0.0 to 0.9999999 need about 24 bits of precision (in binary). For example the binary fraction with 24 1's:

$$111111111111111111111111 = 0.9999999404$$

Not every real number between 0 and 0.9999999404 can be represented by a 24-bit fractional number.

The smallest non-zero number that can be represented is:

$$000000000000000000000001 = 5.96046 \times 10^{-8}$$

Every other non-zero number is constructed in increments of this value.

Sign and exponent digits

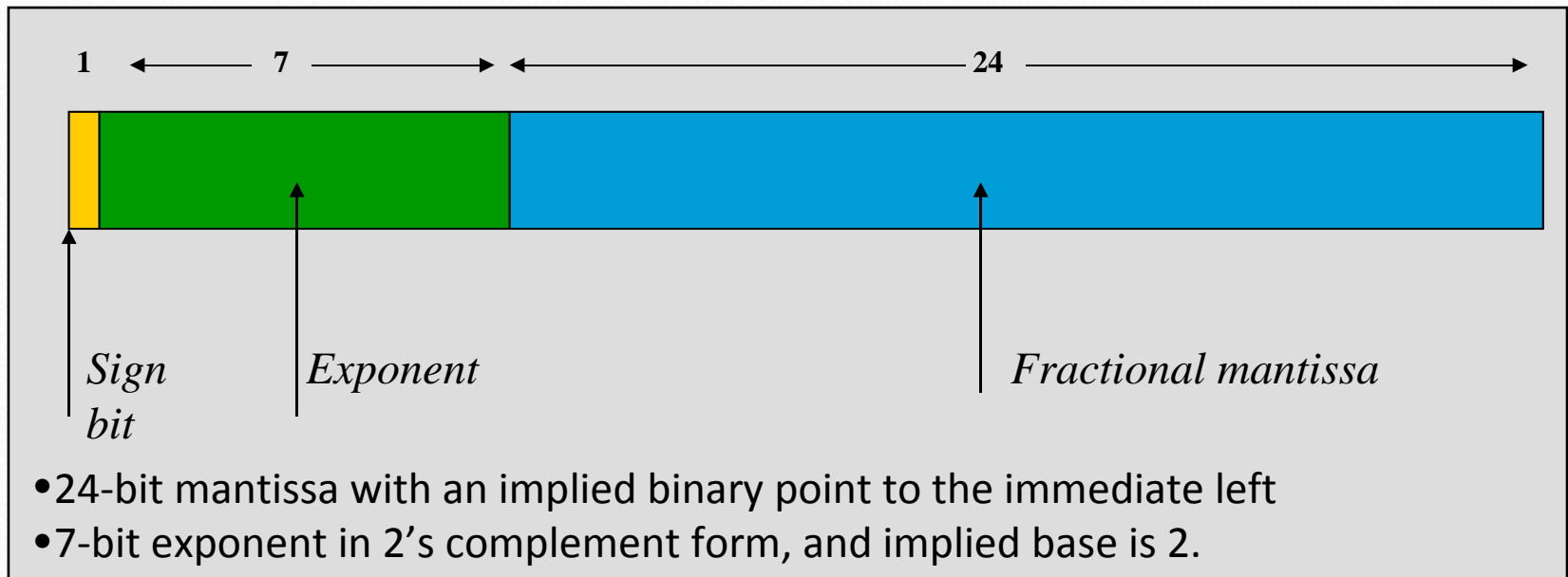
- In a 32-bit number, suppose we allocate 24 bits to represent a fractional mantissa.
- Assume that the mantissa is represented in sign and magnitude format, and we have allocated one bit to represent the sign.
- We allocate 7 bits to represent the exponent, and assume that the exponent is represented as a 2's complement integer.
- There are no bits allocated to represent the base, we assume that the base is implied for now, that is the base is 2.
- Since a 7-bit 2's complement number can represent values in the range -64 to 63, the range of numbers that can be represented is:

$$0.0000001 \times 2^{-64} \leq |x| \leq 0.9999999 \times 2^{63}$$

- In decimal representation this range is:

$$0.5421 \times 10^{-20} \leq |x| \leq 9.2237 \times 10^{18}$$

A sample representation



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 .

A number which is brought into a form so that all of the available mantissa digits are optimally used (this is different from all occupied which may not hold), is called a normalized number.

Same methodology holds in the case of binary mantissas

$$0001101000(10110) \times 2^8 = 1101000101(10) \times 2^5$$

Normalization (contd..)

- A floating point number is in normalized form if the most significant 1 in the mantissa is in the most significant bit of the mantissa.
- All normalized floating point numbers in this system will be of the form:

$$0.1xxxxx.....xx$$

Range of numbers representable in this system, if every number must be normalized is:

$$0.5 \times 2^{-64} \leq |x| < 1 \times 2^{63}$$

Normalization, overflow and underflow

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

Applying the normalization procedure to:

$.000111001110\dots0010 \times 2^{-62}$

gives:

$.111001110\dots \times 2^{-65}$

But we cannot represent an exponent of -65 , in trying to normalize the number we have underflowed our representation.

Applying the normalization procedure to:

$1.00111000\dots \times 2^{63}$

gives:

$0.100111\dots \times 2^{64}$

This overflows the representation.

Changing the implied base

So far we have assumed an implied base of 2, that is our floating point numbers are of the form:

$$x = m 2^e$$

If we choose an implied base of 16, then:

$$x = m 16^e$$

Then:

$$y = (m \cdot 16) \cdot 16^{e-1} = (m \cdot 2^4) \cdot 16^{e-1} = m \cdot 16^e = x$$

- Thus, every four left shifts of a binary mantissa results in a decrease of 1 in a base 16 exponent.
- Normalization in this case means shifting the mantissa until there is a 1 in the first four bits of the mantissa.

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.
- If 7 bits are allocated to the exponent, exponents can be represented in the range of -64 to +63, that is:

$$-64 \leq e \leq 63$$

Exponent can also be represented using the following coding called as excess-64:

$$E' = E_{true} + 64$$

In general, excess-p coding is represented as:

$$E' = E_{true} + p$$

True exponent of -64 is represented as 0

0 is represented as 64

63 is represented as 127

This enables efficient comparison of the relative sizes of two floating point numbers.

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.

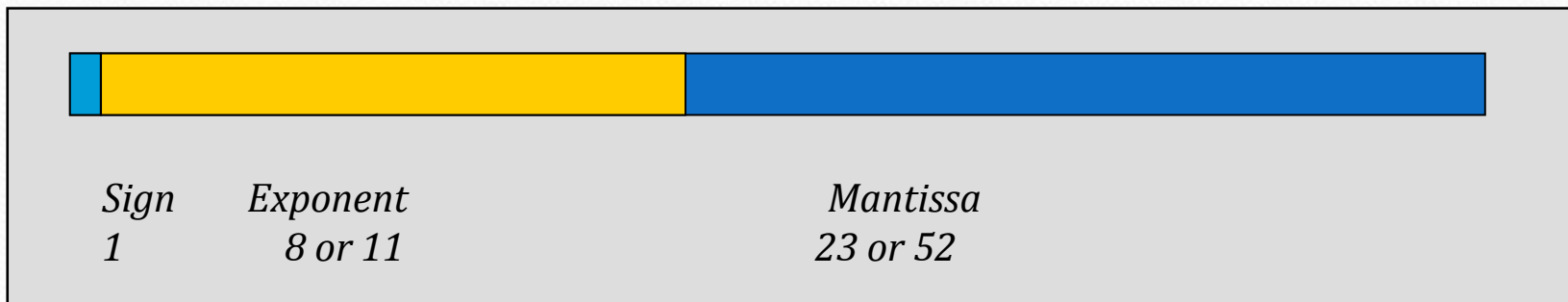
Single precision:

- 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.



Peculiarities of 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.
- So the real MSB of a number in the IEEE notation is either a 0 or a 1.
- The values of the numbers represented in the IEEE single precision notation are of the form:

$$(+,-) 1.M \times 2^{(E - 127)}$$

- 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:

$$\begin{aligned} & -126 \leq E \leq 127 \quad \text{and} \quad -1022 \leq E \leq 1023 \\ & \text{not} \\ & -127 \leq E \leq 128 \quad \text{and} \quad -1023 \leq E \leq 1024 \end{aligned}$$

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

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 $x+y+2p$.

Biases should be handled in floating point arithmetic.

Floating point arithmetic: ADD/SUB rule

- Choose the number with the smaller exponent.
- Shift its mantissa right until the exponents of both the numbers are equal.
- Add or subtract the mantissas.
- Determine the sign of the result.
- 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

- Add the exponents.
- Subtract the bias.
- Multiply the mantissas and determine the sign of the result.
- Normalize the result (if necessary).
- Truncate/round the mantissa of the result.

Floating point arithmetic: DIV rule

- Subtract the exponents
- Add the bias.
- Divide the mantissas and determine the sign of the result.
- Normalize the result if necessary.
- Truncate/round the mantissa of the result.

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

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 denormalizes the truncated number.

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

- IEEE uses the rounding method.