





Computer Arithmetic



Recall

- ❑ In previous lectures, we have discussed the representation of signed binary numbers, and showed that 2's-complement is the best representation from the standpoint of performing addition and subtraction operations.
 - ❑ It is shown that two, n-bit, signed numbers can be added using n-bit binary addition, treating the sign bit the same as the other bits.
 - ❑ In other words, a logic circuit that is designed to add unsigned binary numbers can also be used to add signed numbers in 2's-complement.
 - ❑ If overflow does not occur, the sum is correct, and any output carry can be ignored.
- 
- 

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

$$s_i = \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

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

Example:

$$\begin{array}{r} X \\ + Y \\ \hline Z \end{array} = \begin{array}{r} 7 \\ + 6 \\ \hline 13 \end{array} = \begin{array}{r} 0111 \\ + 0110 \\ \hline 1101 \end{array}$$

Legend for stage i

At the i^{th} stage:

Input:

c_i is the carry-in

Output:

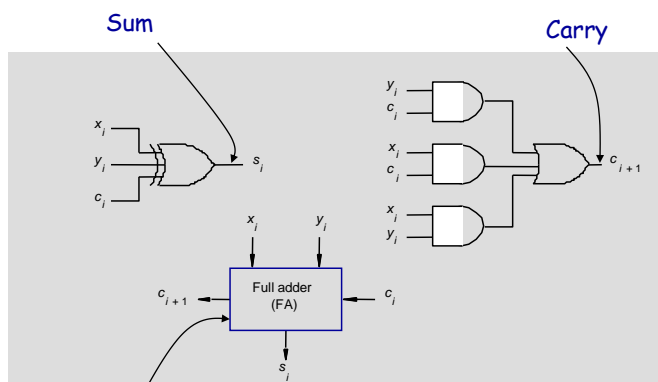
s_i is the sum

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

State

- Note that each stage of the addition process must accommodate a carry-in bit.
- We use c_i to represent the carry-in to the i^{th} stage, which is the same as the carry-out from the $(i-1)^{st}$ stage.

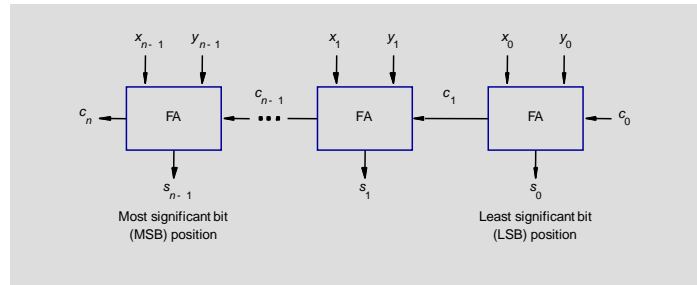
Addition logic for a single stage



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

n-bit adder

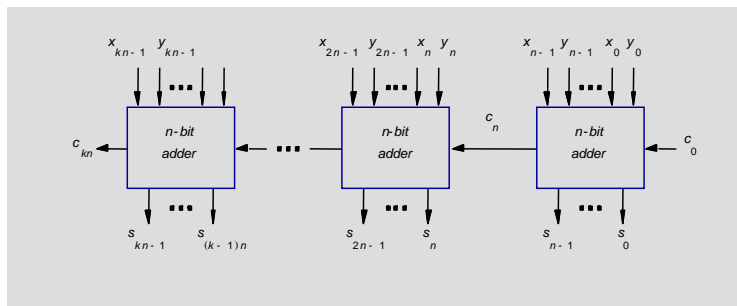
- ❑ Cascade n full adder (FA) blocks to form a n -bit adder.
- ❑ Carries propagate or ripple through this cascade, n -bit ripple carry adder.



- ❑ Carry-in c_0 into the LSB position provides a convenient way to perform subtraction.
- ❑ The carry signals are also useful for interconnecting k adders to form an adder capable of handling input numbers that are kn bits long

K n-bit adder

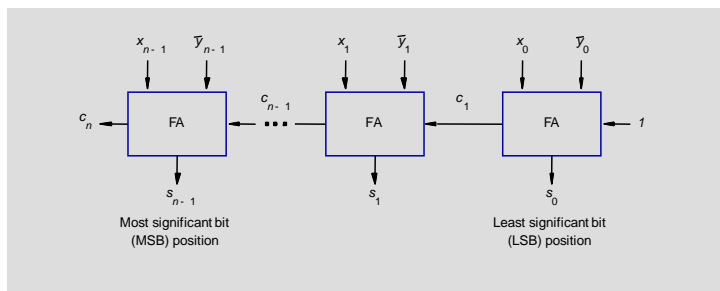
K n -bit numbers can be added by cascading k n -bit adders.



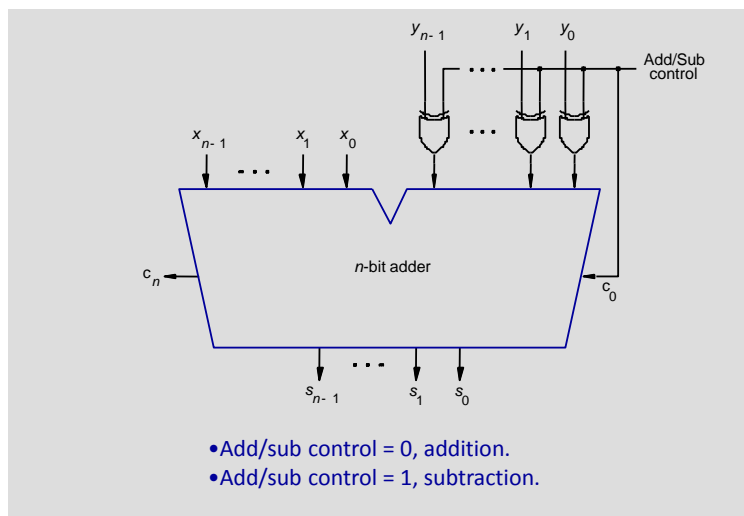
Each n -bit adder forms a block, so this is cascading of blocks.
Carries ripple or propagate through blocks, [Blocked Ripple Carry Adder](#)

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..)



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:

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

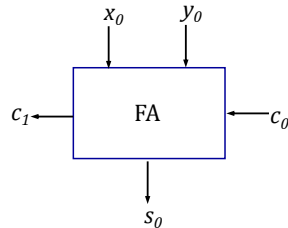
$$Overflow = c_n \oplus c_{n-1}$$

Design of Fast Adder

- ❑ If an n -bit ripple-carry adder is used in the addition/subtraction unit, it may have too much delay in developing its outputs, s_0 through s_{n-1} and C_n .
- ❑ The delay through a network of logic gates depends on the integrated circuit electronic technology, used in fabricating the network and on the number of gates in the paths from inputs to outputs.
- ❑ The delay through any combinational logic network constructed from gates in a particular technology is determined by adding up the number of logic-gate delays along the longest signal propagation path through the network.
- ❑ In the case of the n -bit ripple-carry adder, the longest path is from inputs x_0, y_0 , and c_0 at the LSB position to outputs C_n and s_{n-1} at the most significant-bit (MSB) position.

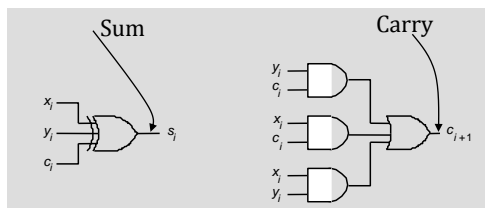


Computing the add time



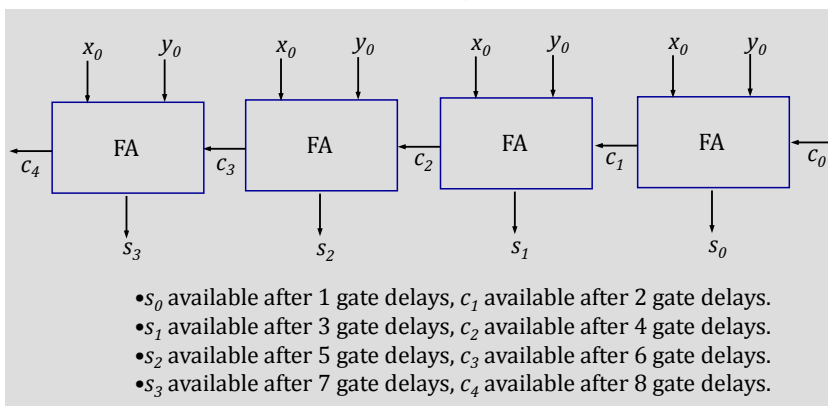
Consider 0^{th} stage:

- 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



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

Design of Fast Adder

- ❑ Two approaches can be taken to reduce delay in adders.
- ❑ The first approach is to use the fastest possible electronic technology in implementing the ripple-carry logic design or variations of it.
- ❑ The second approach is to use an augmented logic gate network structure that is larger than that used in n-bit ripple carry adder.
- ❑ In practice, a number of design techniques have been used to implement high-speed adders. They include electronic circuit designs for fast propagation of carry signals as well as variations on the basic network structure.

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:

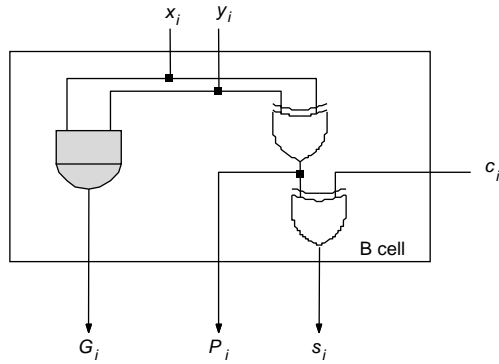
$$c_{i+1} = G_i + P_i c_i$$

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

- ❑ 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.



Fast addition



B-cell for a single stage

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

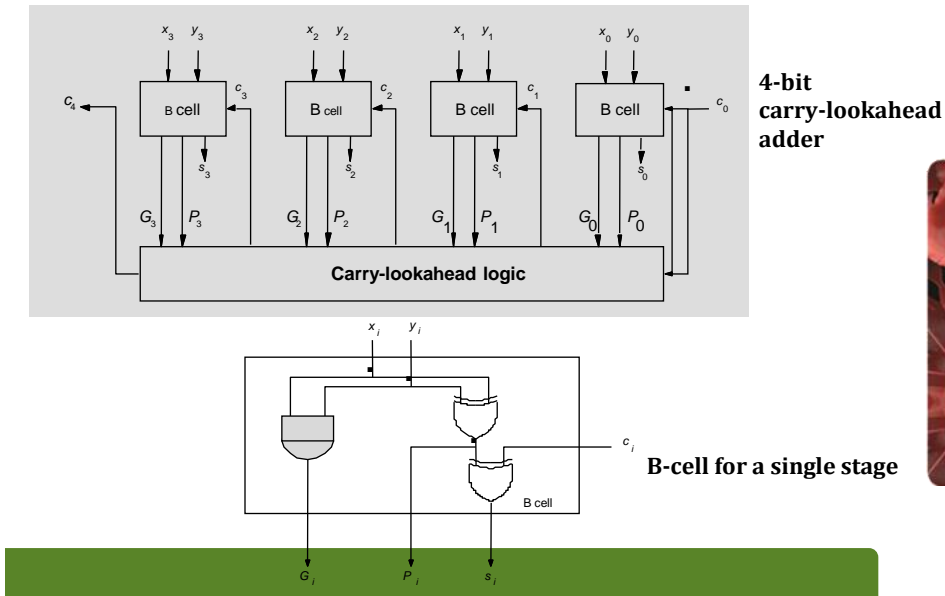
$$\Rightarrow c_{i+1} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} c_{i-2}))$$

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



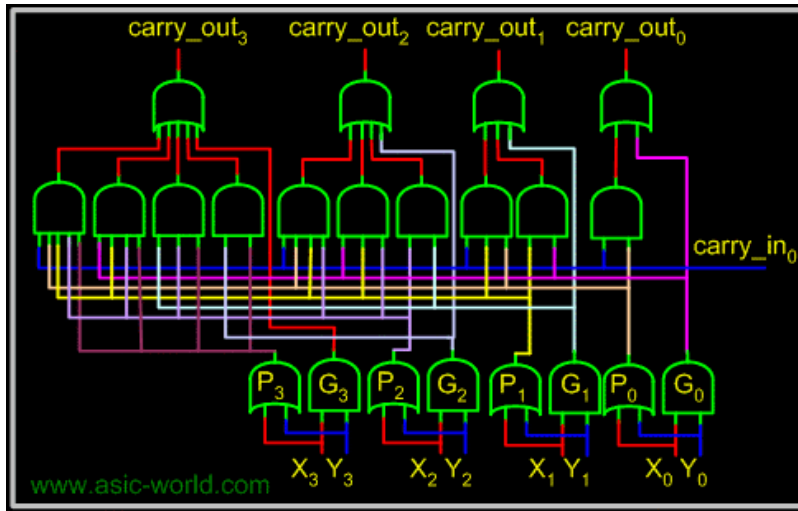
Carry lookahead adder (contd..)

- 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:

$$P_0^I = P_3P_2P_1P_0$$

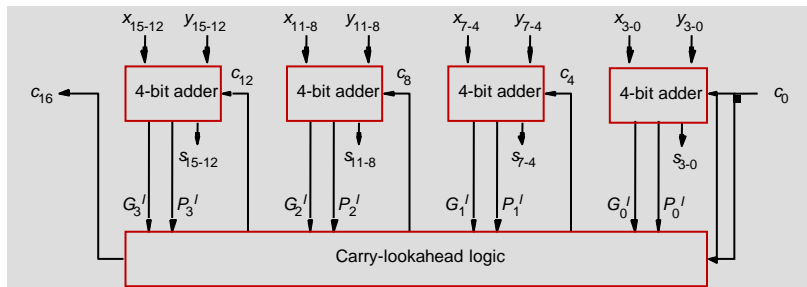
$$G_0^I = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$

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

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

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

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

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

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..)

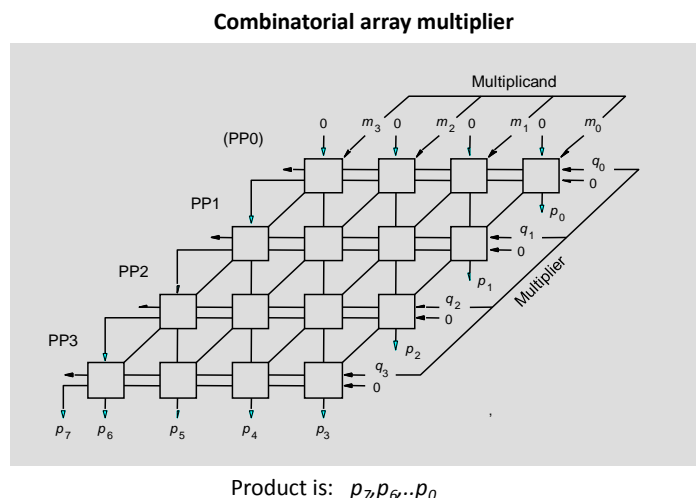
- ☐ We added the partial products at end.
 - ☐ Alternative would be to add the partial products at each stage.
- ☐ Rules to implement multiplication are:
 - ☐ If the i^{th} bit of the multiplier is 1, shift the multiplicand and add the shifted multiplicand to the current value of the partial product.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



Combinatorial array multiplier

- Binary multiplication of positive operands can be implemented in a combinational, two-dimensional logic array, as shown in the next slide.

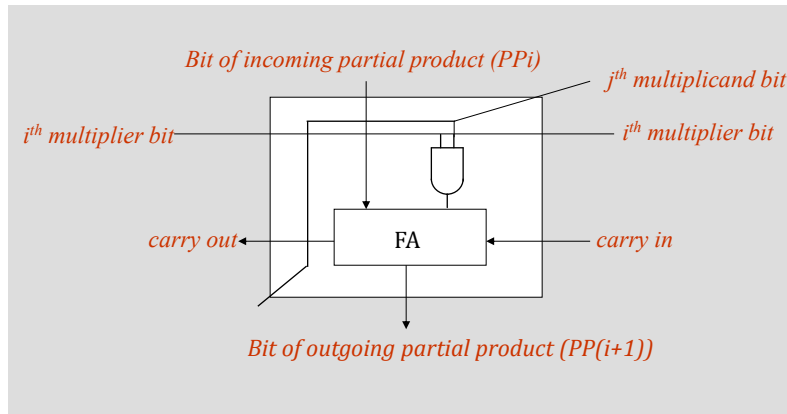
Combinatorial array multiplier



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

A cell of combinatorial array multiplier



Typical multiplication cell





Combinatorial array multiplier (contd..)

- ❑ The main component in each cell is a full adder FA. The AND gate in each cell determines whether a multiplicand bit, m_j , is added to the incoming partial-product bit, based on the value of the multiplier bit, q_i .
- ❑ Each row i , where $0 \leq i \leq 3$, adds the multiplicand (appropriately shifted) to the incoming partial product, PP_i , to generate the outgoing partial product, $PP_{(i+1)}$, if $q_i = 1$.
- ❑ If $q_i = 0$, PP_i is passed vertically downward unchanged.
- ❑ PP_0 is all 0s, and PP_4 is the desired product.
- ❑ The multiplicand is shifted left one position per row by the diagonal signal path.

Propagation path delay analysis

- ❑ The worst case signal propagation delay path is from the upper right corner of the array to the high-order product bit output at the bottom left corner of the array.
 - ❑ The path consists of the staircase pattern that includes the two cells at the right end of each row, followed by all the cells in the bottom row.
 - ❑ Assuming that there are two gate delays from the inputs to the outputs of a full adder block, the path has a total of $6(n-1) - 1$ gate delays, including the initial AND gate delay in all cells, for the $n \times n$ array.
 - ❑ Only the AND gates are actually needed in the first row of the array because the incoming partial product PP_0 is zero.
- 
- 

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.

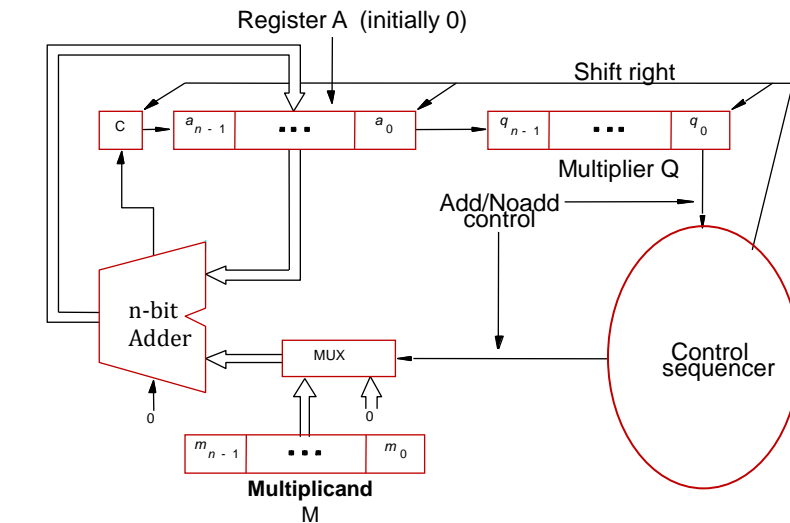


Sequential multiplication

- ❑ The simplest way to perform multiplication is to use the adder circuitry in the ALU for a number of sequential steps.
- ❑ The block diagram on next slide shows the hardware arrangement for sequential multiplication.
- ❑ This circuit performs multiplication by using a single n -bit adder n times to implement the spatial addition performed by the n rows of combinatorial multiplier.





Sequential Circuit Multiplier





Sequential multiplication

- ❑ Registers A and Q combined hold PP_i while multiplier bit q_i generates the signal Add/Noadd.
- ❑ This signal controls the addition of the multiplicand, M , to PP_i to generate $PP_{(i+1)}$.
- ❑ The product is computed in n cycles.
- ❑ The partial product grows in length by one bit per cycle from the initial vector, PP_0 , of n 0_s in register A.
- ❑ The carry-out from the adder is stored in flip-flop C , shown at the left end of register A.

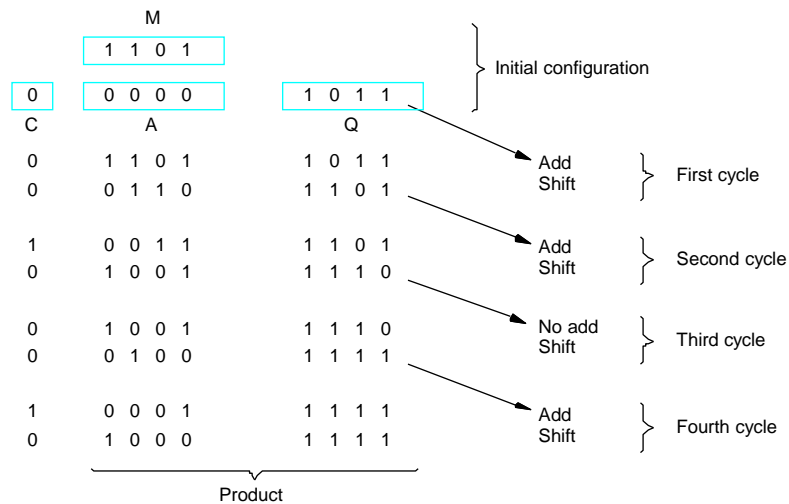
Sequential multiplication

- ❑ At the start, the multiplier is loaded into register Q, the multiplicand into register M, and C and A are cleared to 0.
 - ❑ At the end of each cycle, C, A, and Q are shifted right one bit position to allow for growth of the partial product as the multiplier is shifted out of register Q.
 - ❑ Because of this shifting, multiplier bit q_i appears at the LSB position of Q to generate the Add/Noadd signal at the correct time, starting with q_0 during the first cycle; q_1 during the second cycle, and so on.
- 
- 

Sequential multiplication

- ❑ After they are used, the multiplier bits are discarded by the right-shift operation.
 - ❑ Note that the carry-out from the adder is the leftmost bit of $PP_{(i+1)}$, and it must be held in the C flip-flop to be shifted right with the contents of A and Q.
 - ❑ After n cycles, the high-order half of the product is held in register A and the low-order half is in register Q.
- 
- 

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)
						<hr/>					
	1	1	1	1	1	1	0	0	1	1	
	1	1	1	1	1	0	0	1	1		
Sign extension is shown in blue	0	0	0	0	0	0	0	0	0		
	1	1	1	0	0	1	1				
	0	0	0	0	0	0					
	<hr/>										
	1	1	0	1	1	1	0	0	0	1	(-143)

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?

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

Booth Algorithm

- To derive the product, we could add four appropriately shifted versions of the multiplicand, as in the standard procedure. However, we can reduce the number of required operations by regarding this multiplier as the difference between two numbers:
- Since $0011110 = 0100000 - 0000010$,
- This suggests that the product can be generated by adding 2^5 times the multiplicand to the 2's-complement of 2^1 times the multiplicand.

Booth Algorithm

- For convenience, we can describe the sequence of required operations by recoding the preceding multiplier as $0 + 1000 - 10$.

								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	← 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	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	1	0	0	0	1	1	0	

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

Booth recoding of a multiplier.

Booth Algorithm

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

Booth multiplication with a negative multiplier.

Booth Algorithm

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

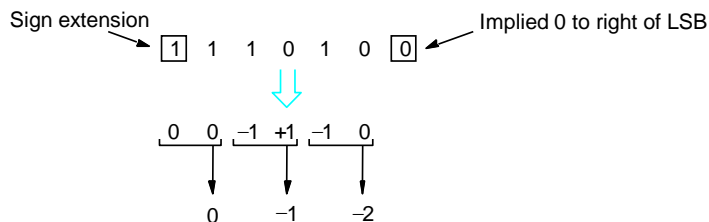
Fast Multiplication

Fast Multiplication

- ❑ There are two techniques for speeding up the multiplication operation.
 - ❑ Bit-Pair Recording
 - ❑ Carry-save Addition
- ❑ The first technique guarantees that the maximum number of summands (versions of the multiplicand) that must be added is $n/2$ for n -bit operands; The second technique reduces the time needed to add the summands.

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

- Initially, a "0" is placed to the right most bit of the multiplier.
- Then 3 bits of the multiplicand is recoded according to table on previous slide or according to the following equation:

$$Z_i = -2x_{i+1} + x_i + x_{i-1}$$

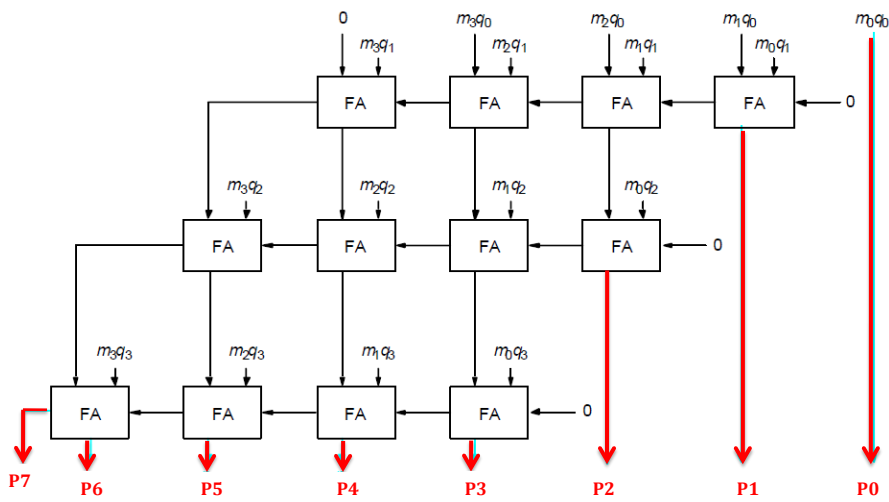
- The 3 digits are selected at a time with overlapping left most bit.
- Here -2 multiplicand is actually the 2s complement of the multiplicand with an equivalent left shift of one bit position.
- Also, $+2$ multiplicand is the multiplicand shifted left one bit position which is equivalent to multiplying by 2.

Carry-Save Addition of Summands

- ❑ Multiplication requires the addition of several summands. A technique called *carry save addition* (CSA) speeds up the addition process.
- ❑ Consider the array for 4 x 4 multiplication shown in next slide. This structure is the same as general array multiplier, with the first row consisting of just the AND gates that implement the bit products m_3q_0 , m_2q_0 , m_1q_0 , and m_0q_0 .

Carry-Save Addition of Summands

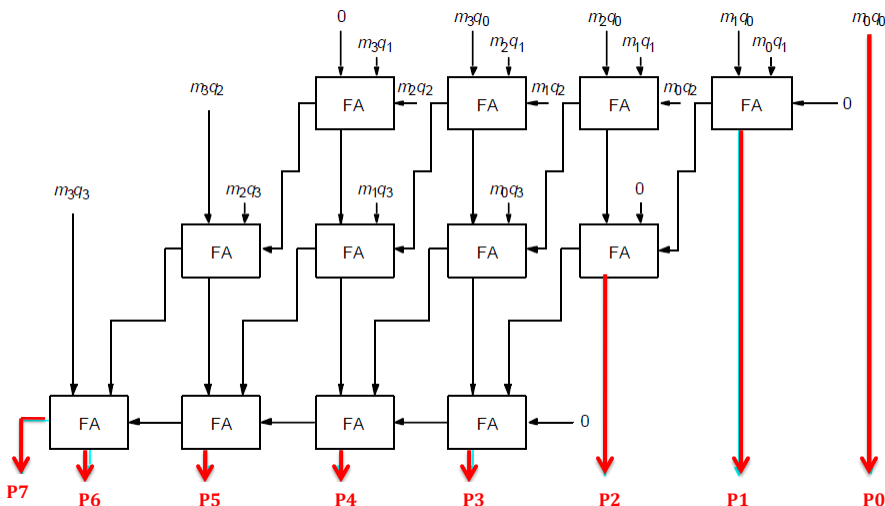
- ❑ CSA speeds up the addition process.



Carry-Save Addition of Summands

- ❑ Instead of letting the carries ripple along the rows, they can be "saved" and introduced into the next row, at the correct weighted positions, as shown in Figure of next slide.
- ❑ This frees up an input to three full adders in the first row. These inputs are used to introduce the third summand bit products m_2q_2 , m_1q_2 , and m_0q_2 .
- ❑ Now, two inputs of each full adder in the second row are fed by sum and carry outputs from the first row.
- ❑ The third input is used to introduce the bit products m_2q_3 , m_1q_3 , and m_0q_3 of the fourth summand.
- ❑ The high-order bit products m_3q_2 and m_3q_3 of the third and fourth summands are introduced into the remaining free inputs at the left end in the second and third rows.
- ❑ The saved carry bits and the sum bits from the second row are now added in the third row to produce the final product bits.

Carry-Save Addition of Summands(Cont.,)



- _____

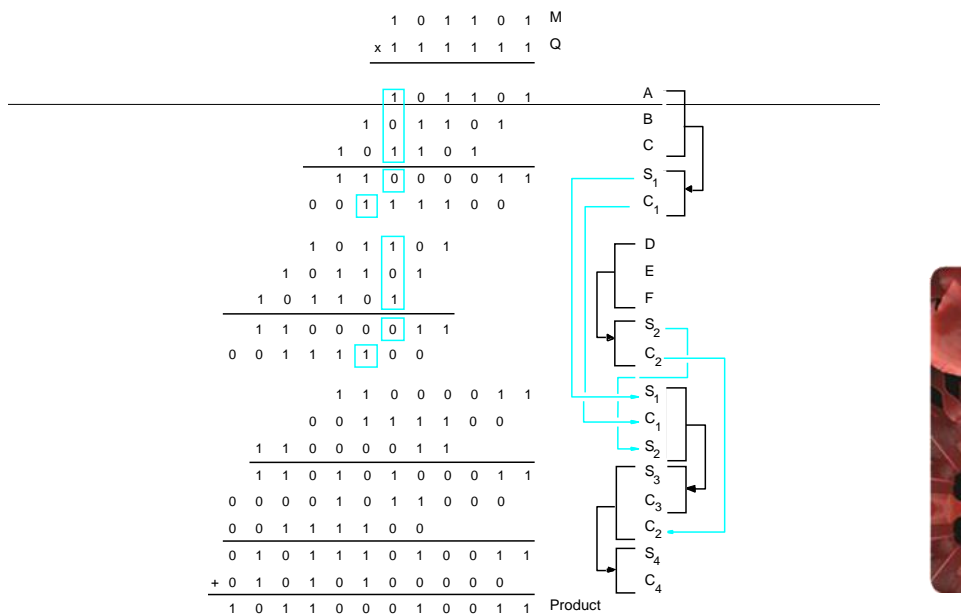


Figure. The multiplication example from previous slide performed using carry-save addition.

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

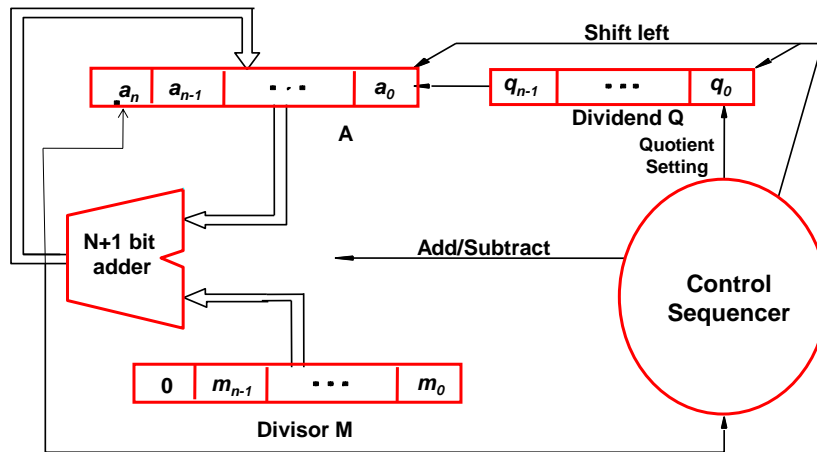


Figure. Circuit arrangement for binary division.

Restoring Division

- ❑ An n -bit positive divisor is loaded into register M and an n -bit positive dividend is loaded into register Q at the start of the operation.
- ❑ Register A is set to 0. After the division is complete, the n -bit quotient is in register Q and the remainder is in register A.
- ❑ The required subtractions are facilitated by using 2's-complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions.

Restoring Division

The following algorithm performs 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

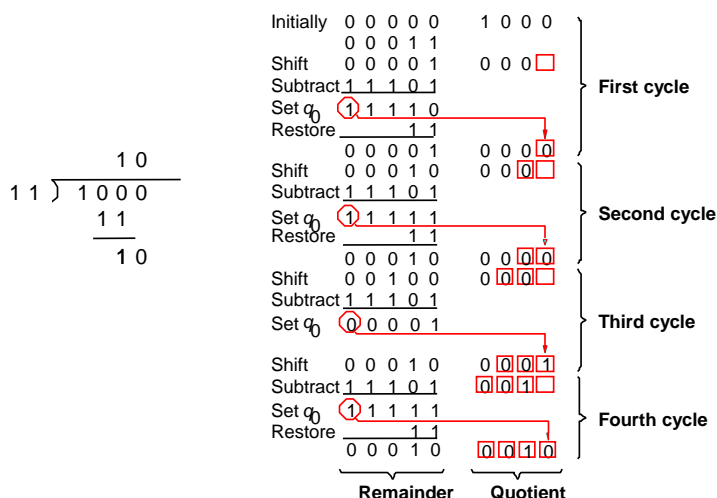
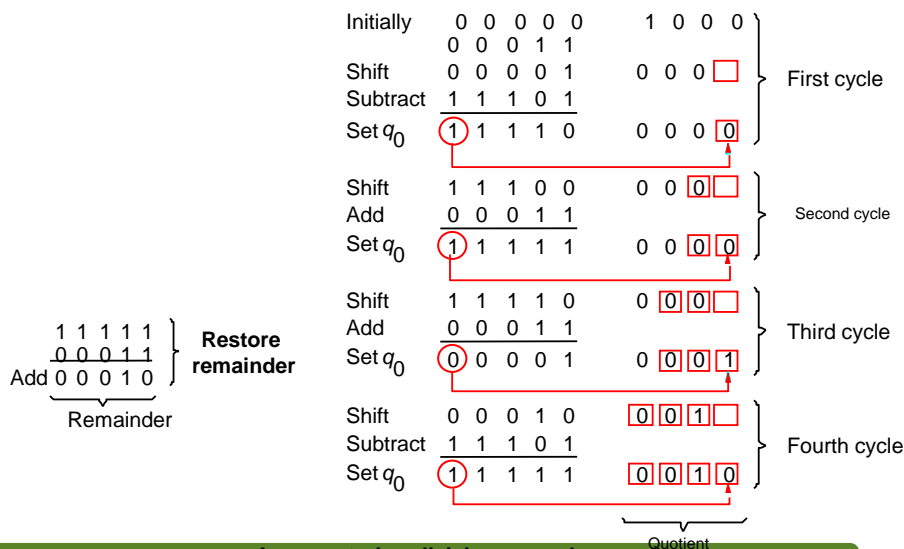


Figure. A restoring-division example.

Nonrestoring Division

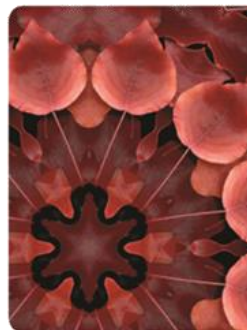
- ❑ Avoid the need for restoring A after an unsuccessful subtraction.
- ❑ Any idea?
- ❑ Step 1: (Repeat n times)
 - If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
 - Now, if the sign of A is 0, set q_0 to 1; otherwise, set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

Examples



A nonrestoring-division example.

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:

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

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:

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

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.
- ❑ 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:

$$\text{Mantissa } (m), \text{ implied base } (b), \text{ and exponent } (e)$$

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

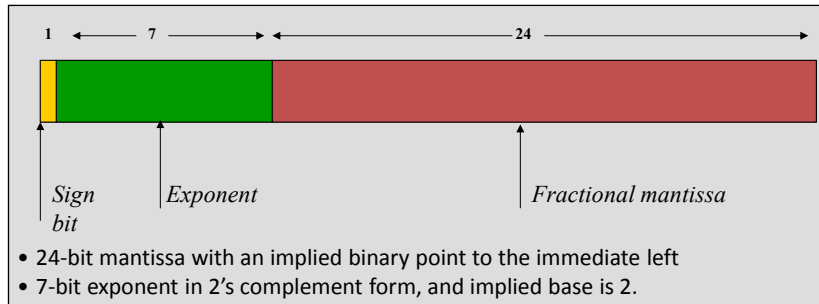
- ☐ In a 32-bit number, suppose we allocate 24 bits to represent a fractional mantissa.
- ☐ Assume that the mantissa is represented in sign and magnitude format, and we have allocated one bit to represent the sign.
- ☐ We allocate 7 bits to represent the exponent, and assume that the exponent is represented 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:

$.000111001110....0010 \times 2^{-62}$

gives:

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

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

Applying the normalization procedure to:

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

gives:

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

This overflows the representation.

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.

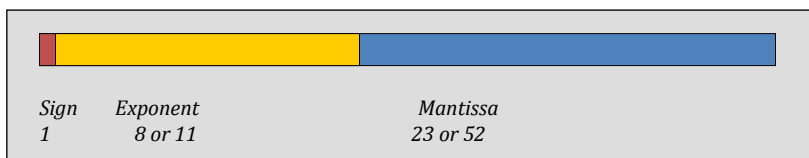
Single precision:

- 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:

$$\begin{array}{l} -126 \leq E \leq 127 \text{ and } -1022 \leq E \leq 1023 \\ \text{not} \\ -127 \leq E \leq 128 \text{ and } -1023 \leq E \leq 1024 \end{array}$$

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.

