

# Arithmetic

Unit 4

# 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

At the  $i^{th}$  stage:

Input:

$c_i$  is the carry-in

Output:

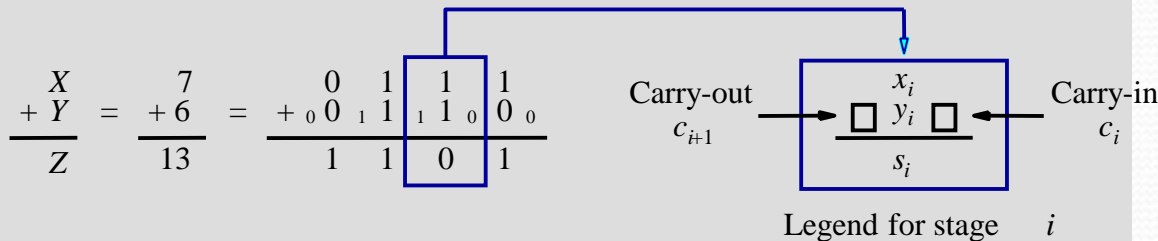
$s_i$  is the sum

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

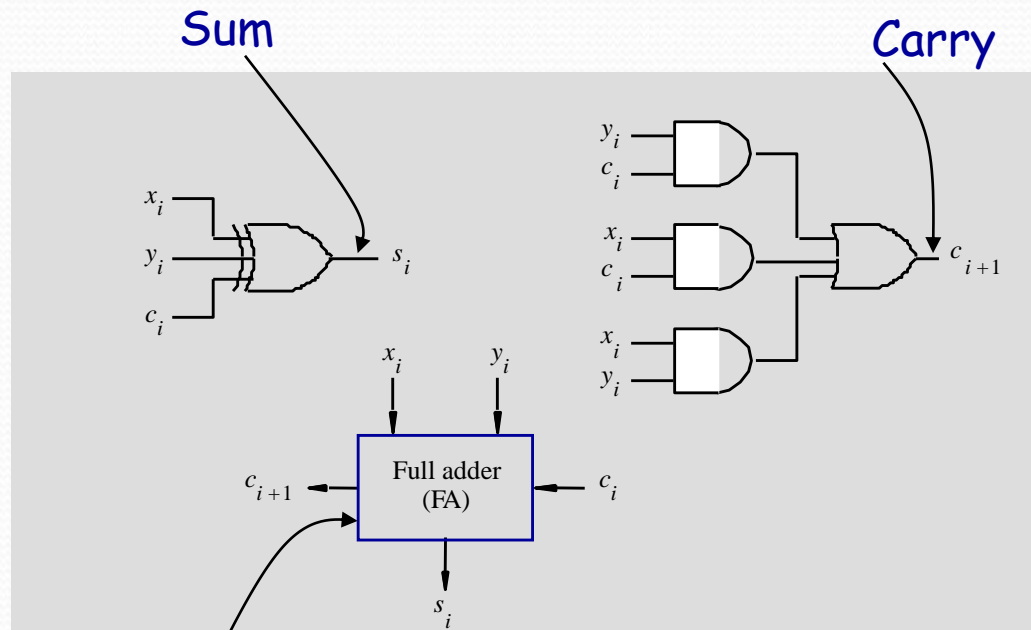
$$s_i = \overline{x_i} y_i c_i + x_i \overline{y_i} c_i + x_i y_i \overline{c_i} + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

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

Example:



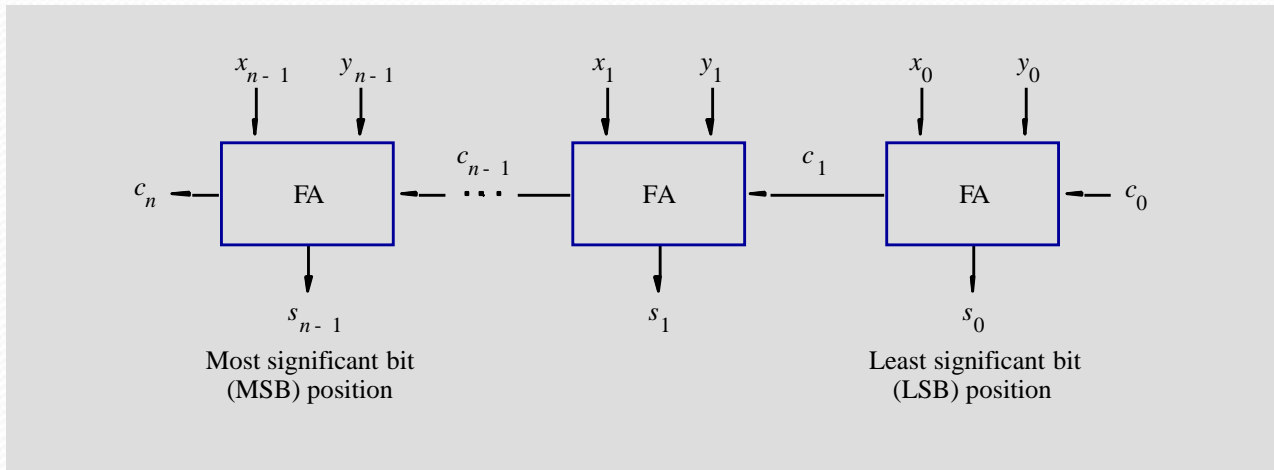
# Addition logic for a single stage



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](#).

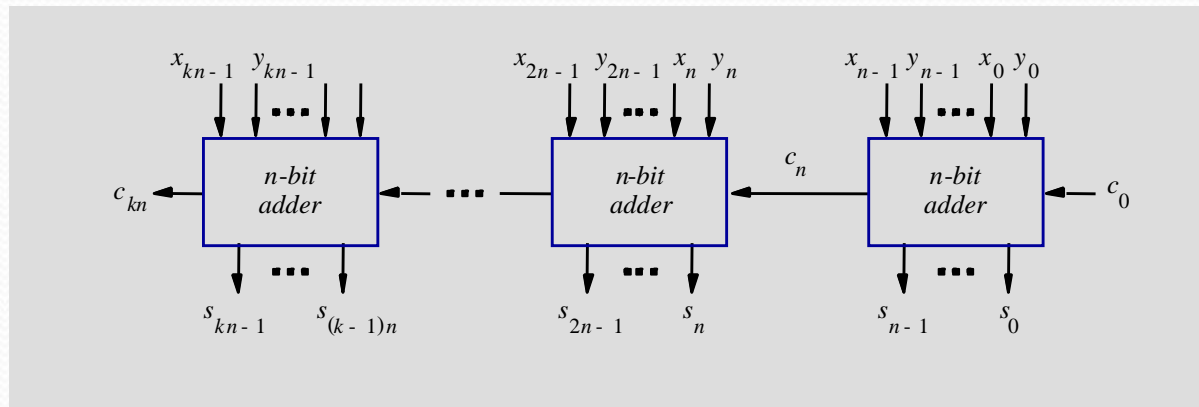


Carry-in  $c_0$  into the LSB position provides a convenient way to perform subtraction.



# $K$ $n$ -bit adder

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

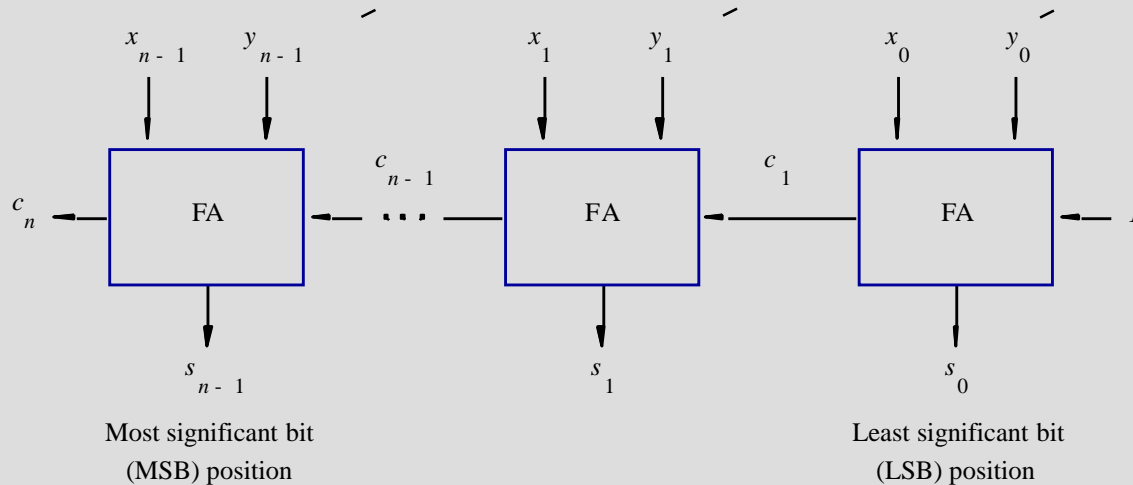


Each  $n$ -bit adder forms a block, so this is cascading of blocks.

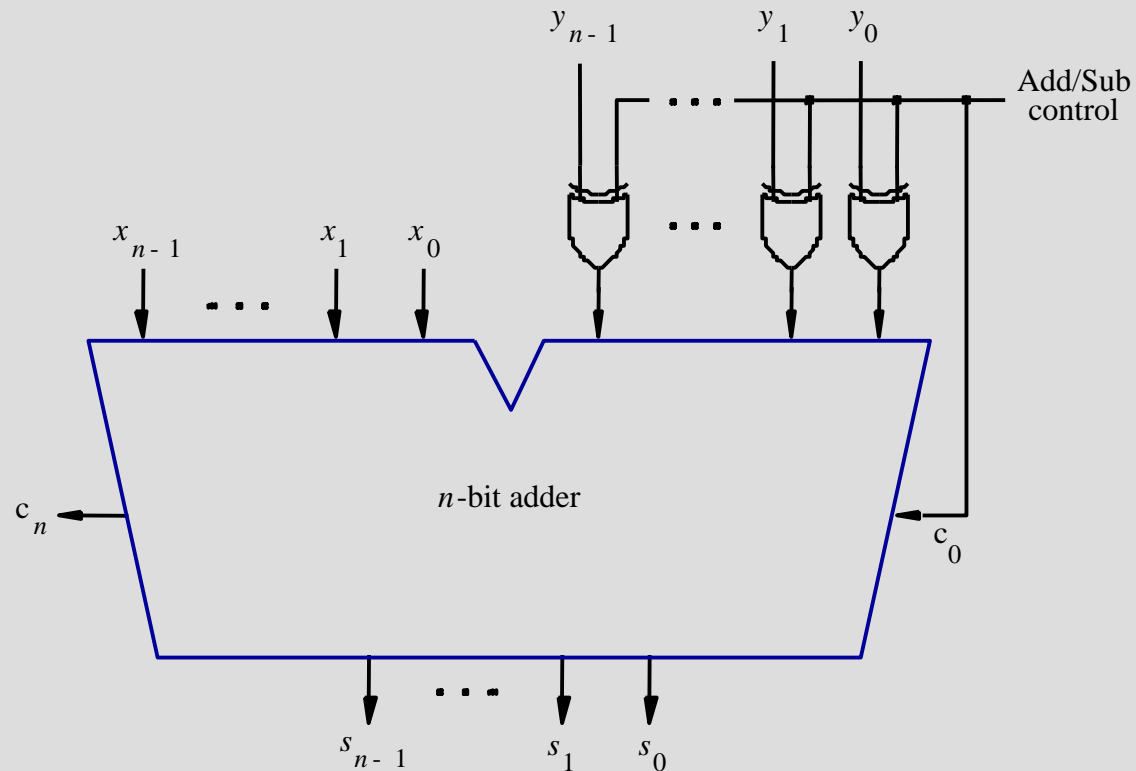
Carries ripple or propagate through blocks, [Blocked Ripple Carry Adder](#)

# $n$ -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.
- $X - Y = X + \overline{Y} + 1$
- 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

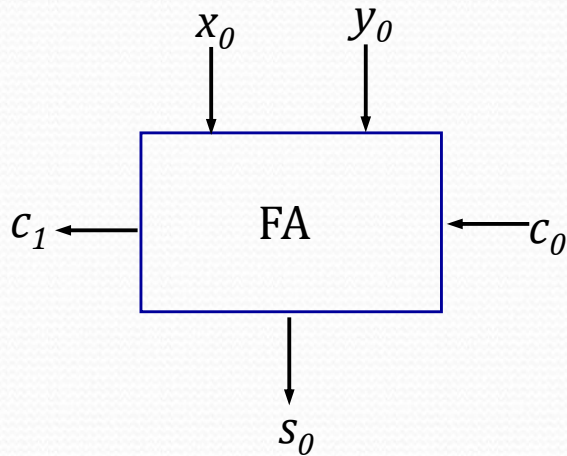
- Overflows can only occur when the sign of the two operands is the same.
- Overflow occurs if the sign of the result is different from the sign of the operands.
- Recall that the MSB represents the sign.
  - $x_{n-1}$ ,  $y_{n-1}$ ,  $s_{n-1}$  represent the sign of operand  $x$ , operand  $y$  and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

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

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

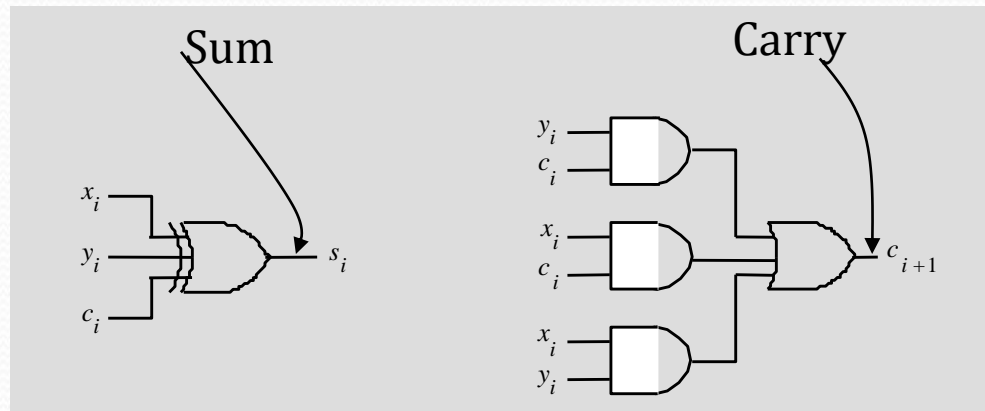


# Computing the add time



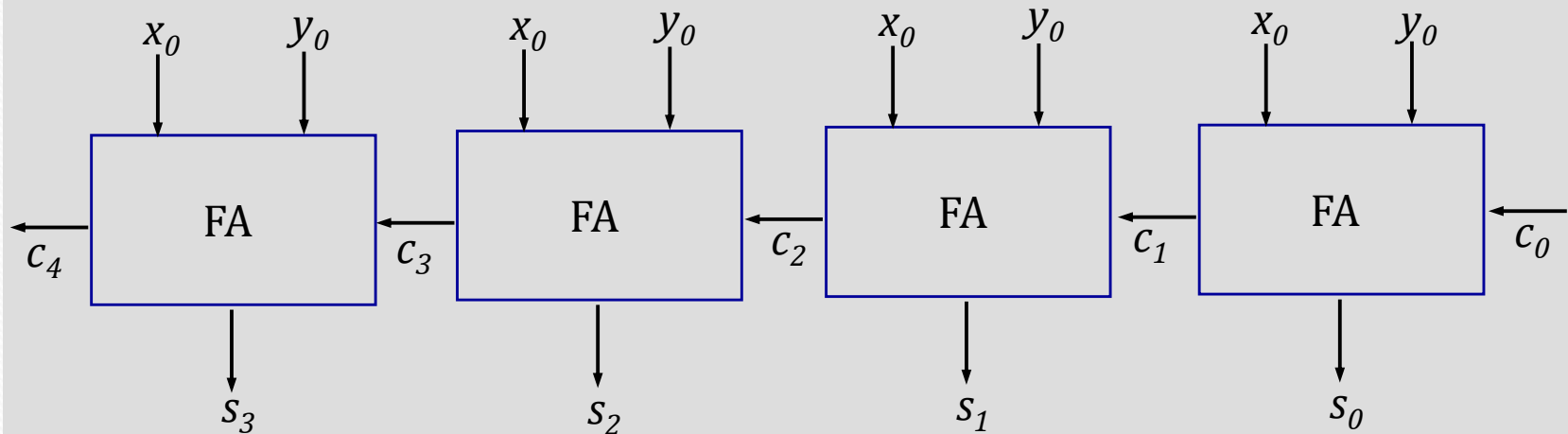
Consider 0<sup>th</sup> stage:

- $c_1$  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*

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

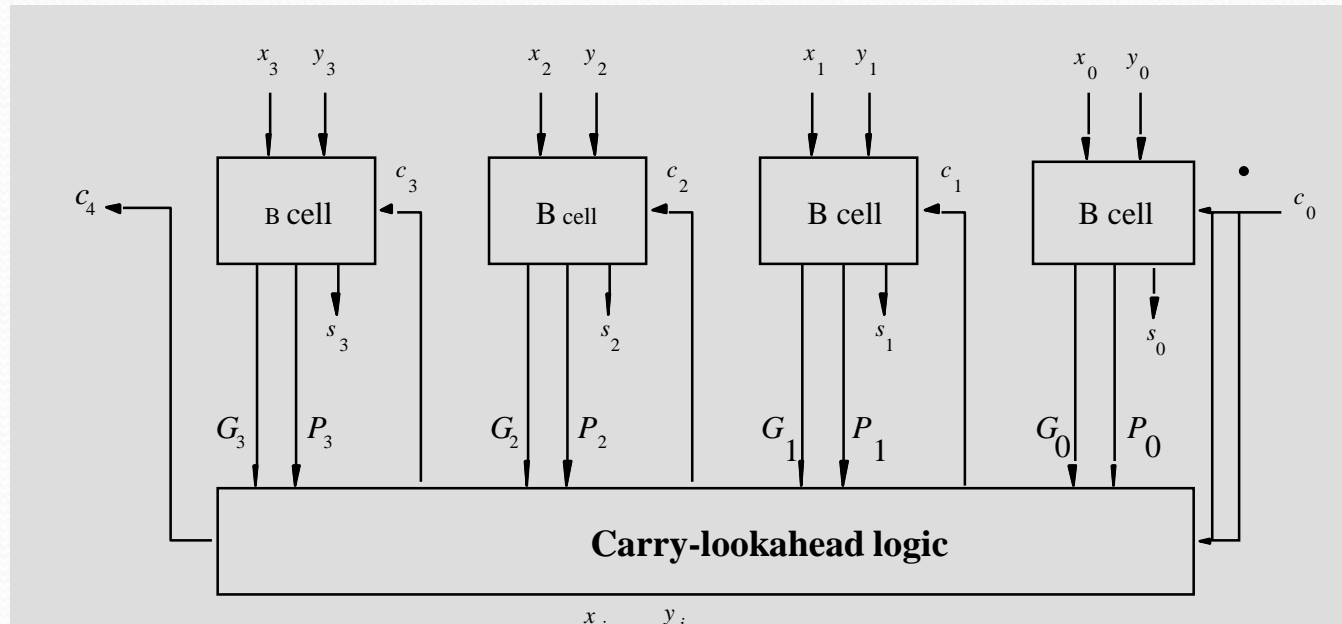
*until*

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

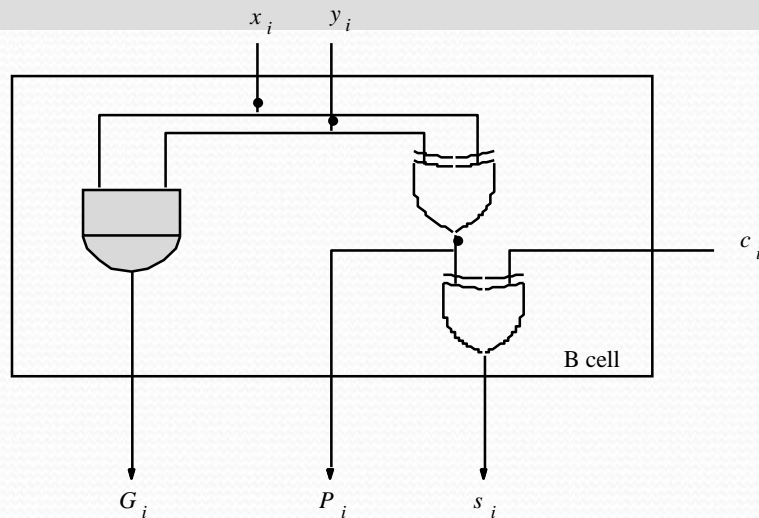
- All carries can be obtained 3 gate delays after  $X$ ,  $Y$  and  $c_0$  are applied.
  - One gate delay for  $P_i$  and  $G_i$
  - Two gate delays in the AND-OR circuit for  $c_{i+1}$
- All sums can be obtained 1 gate delay after the carries are computed.
- Independent of  $n$ ,  $n$ -bit addition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

# 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:

$$P_0^I = P_3P_2P_1P_0$$

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

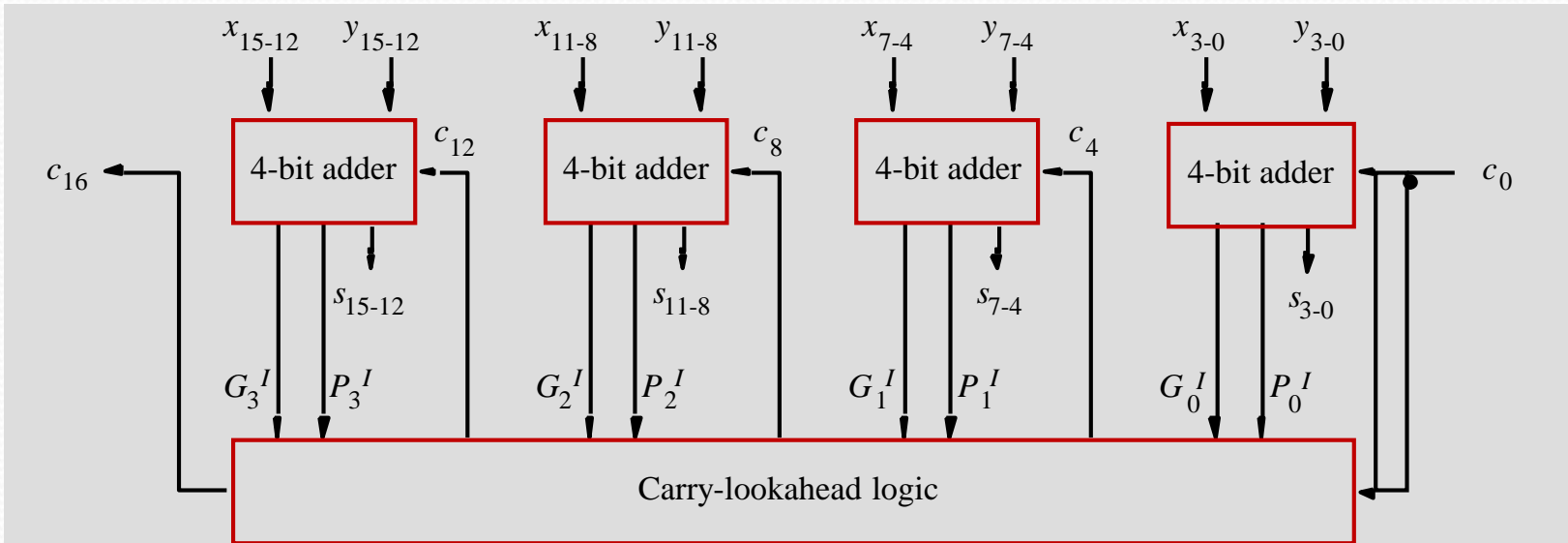
*Subscript I denotes the blocked carry lookahead and identifies the block.*

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.
- $P_i^I$  is available after 2 and  $G_i^I$  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

```

      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.**

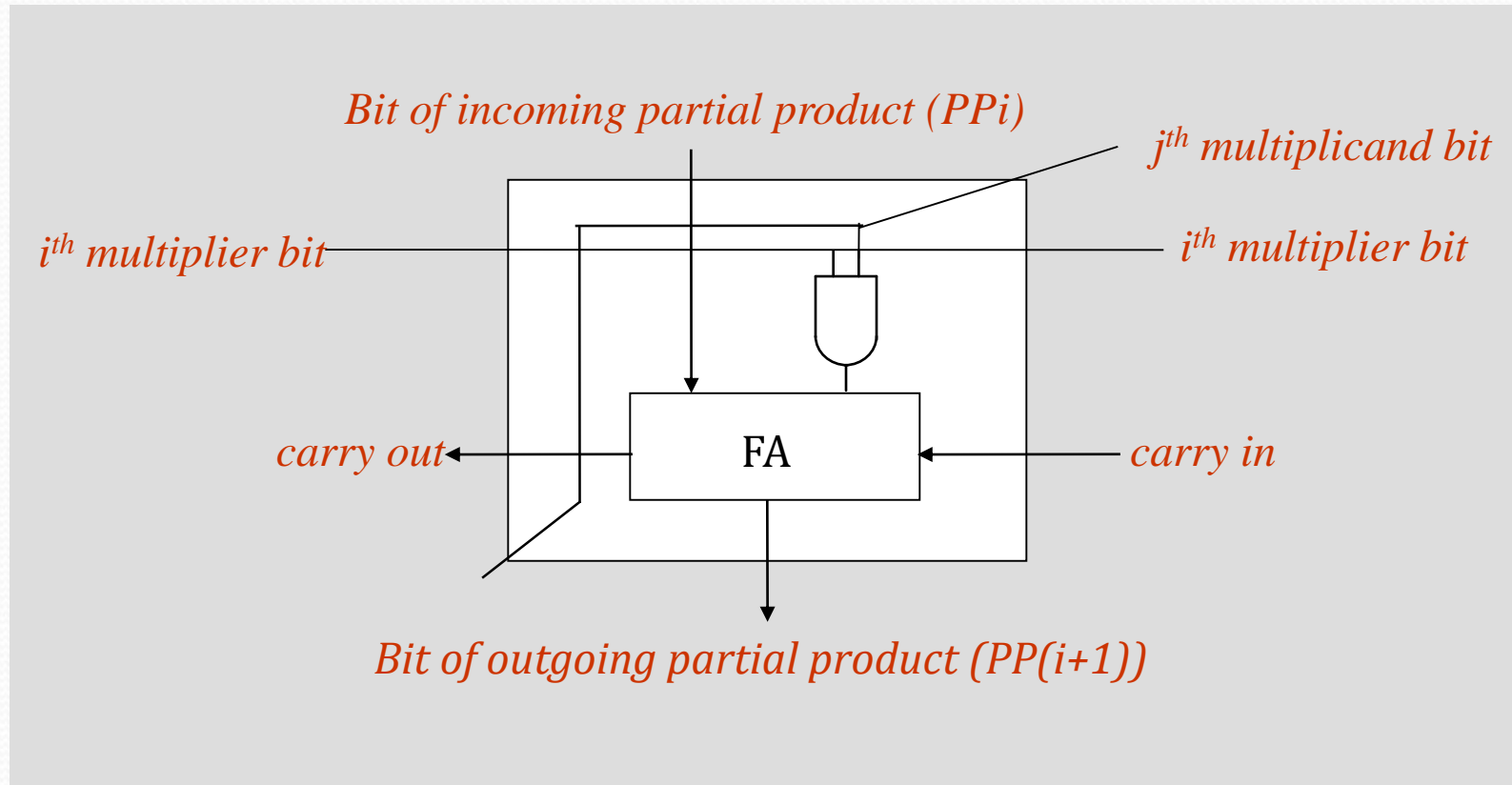
**Unsigned multiplication can be viewed as addition of shifted versions of the multiplicand.**

# 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  $j^{th}$  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

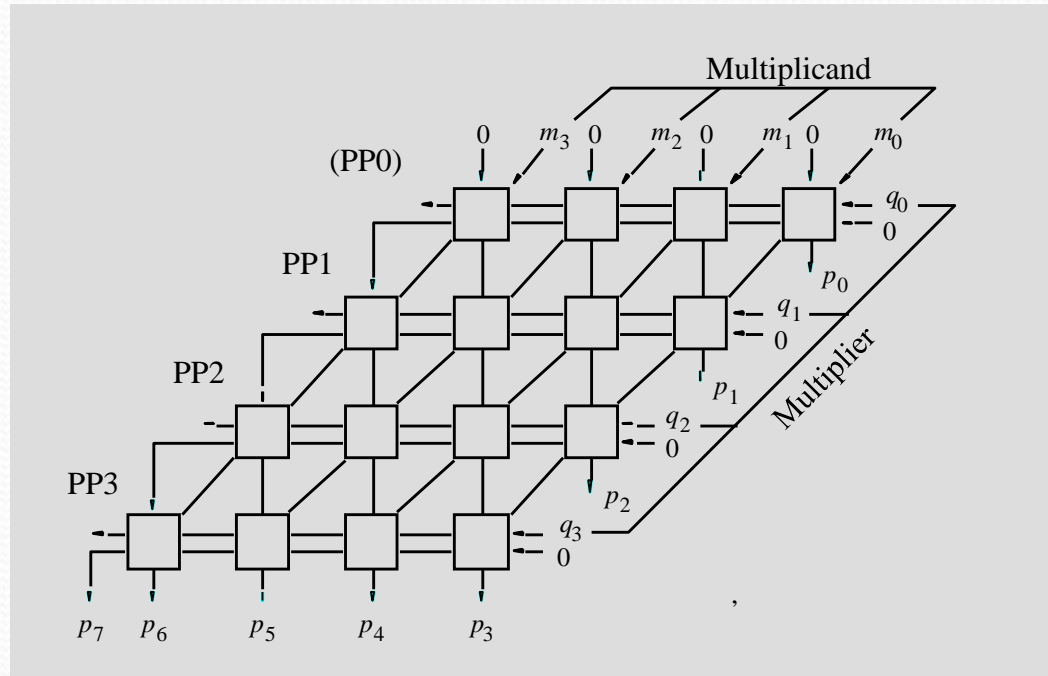
Typical multiplication cell





# 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..)

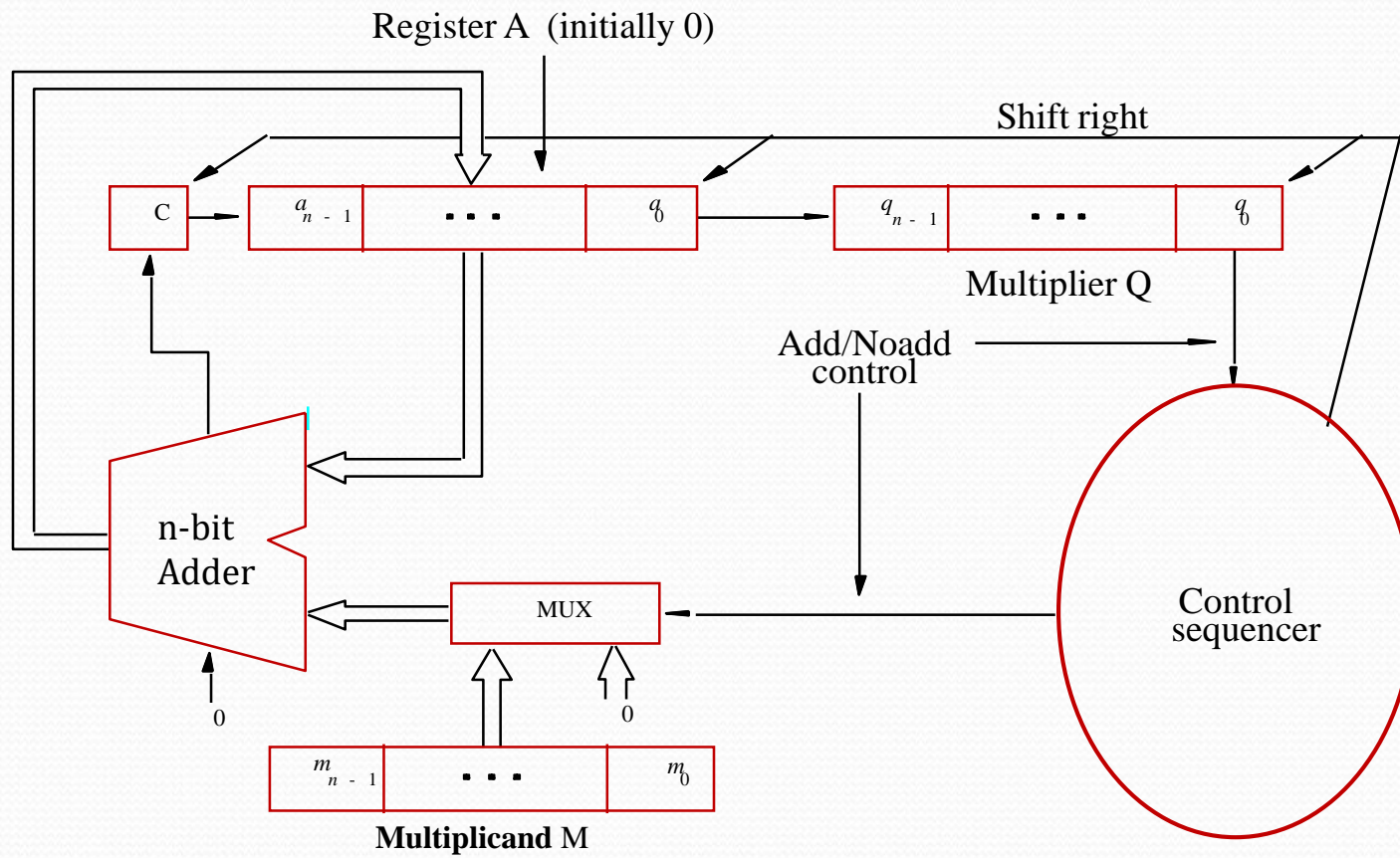
- Combinatorial array multipliers are:
  - Extremely inefficient.
  - Have a high gate count for multiplying numbers of practical size such as 32-bit or 64-bit numbers.
  - Perform only one function, namely, unsigned integer product.
- Improve gate efficiency by using a mixture of combinatorial array techniques and sequential techniques requiring less combinational logic.



# Sequential multiplication

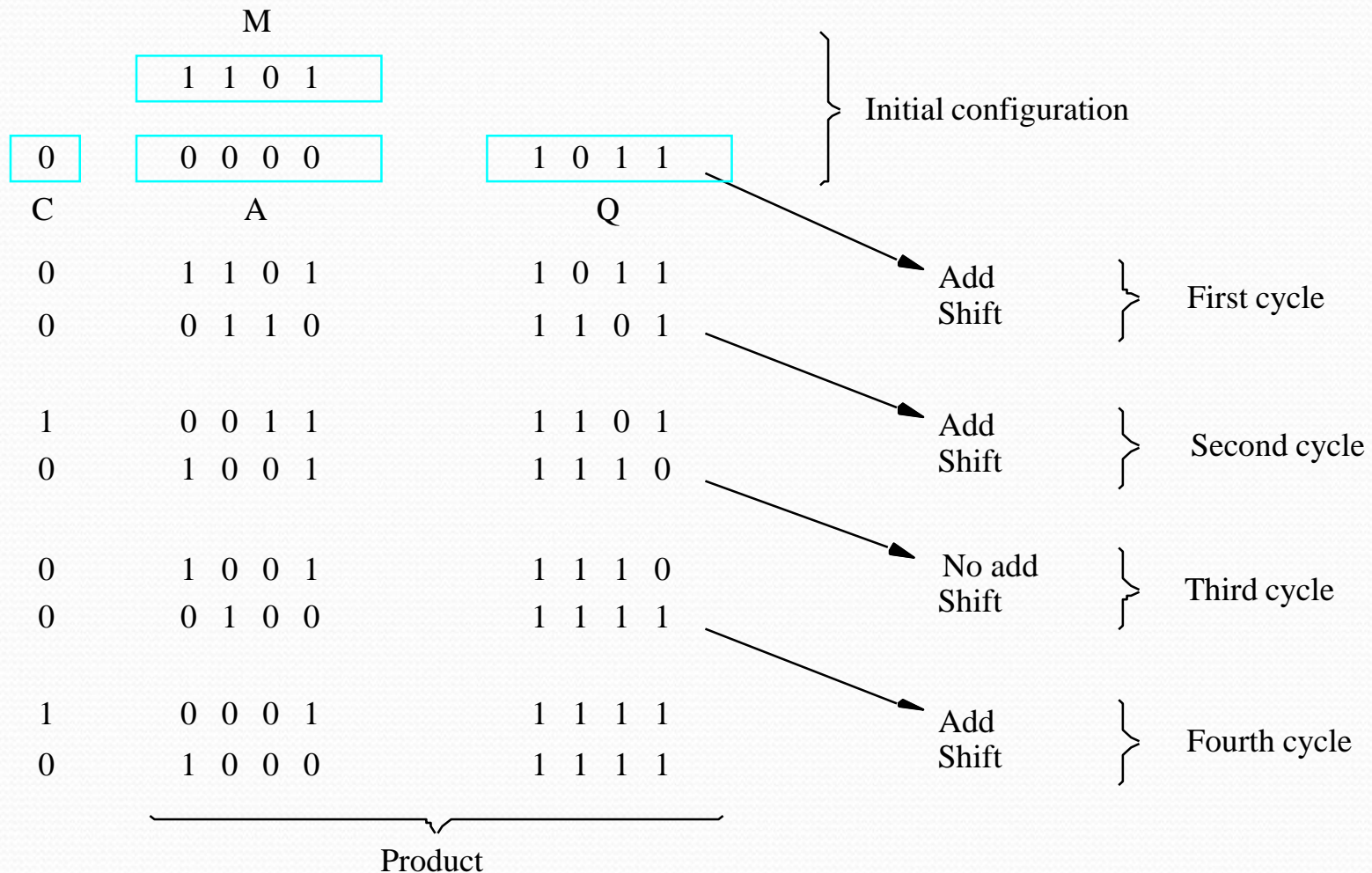
- Recall the rule for generating partial products:
  - If the  $i$ th bit of the multiplier is 1, add the appropriately shifted multiplicand to the current partial product.
  - Multiplicand has been shifted left when added to the partial product.
- However, adding a left-shifted multiplicand to an unshifted 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?

Sign extension is shown in blue

					1	0	0	1	1	(- 13)
					0	1	0	1	1	(+ 11)
					<hr/>					
	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				
	<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?

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



# 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
								<hr/>								
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	1	0	1	1	0	1								
0	0	0	0	0	0	0	0	0								
<hr/>								0	0	0	1	0	1	0	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	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

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

- 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

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

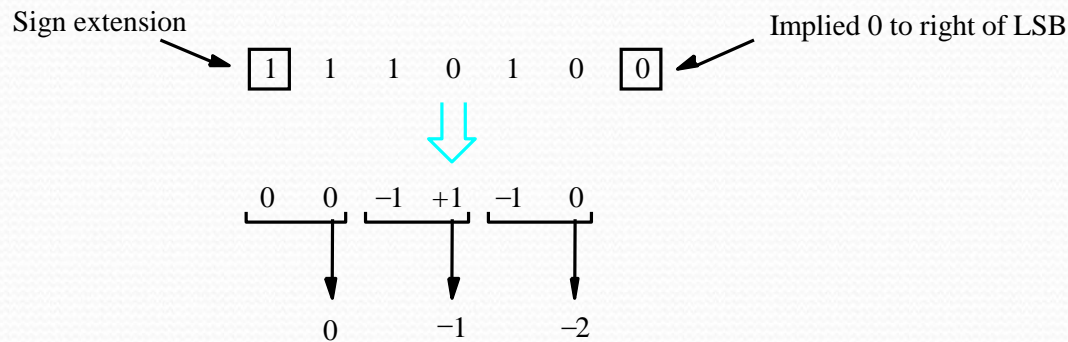
$$\begin{array}{r}
 \begin{array}{cccccc}
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & \\
 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & & \\
 1 & 1 & 1 & 0 & 0 & 1 & 1 & & & \\
 0 & 0 & 0 & 0 & 0 & 0 & & & & \\
 \hline
 1 & & & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & (-78)
 \end{array}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

Booth multiplication with a negative multiplier.

# 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

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



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

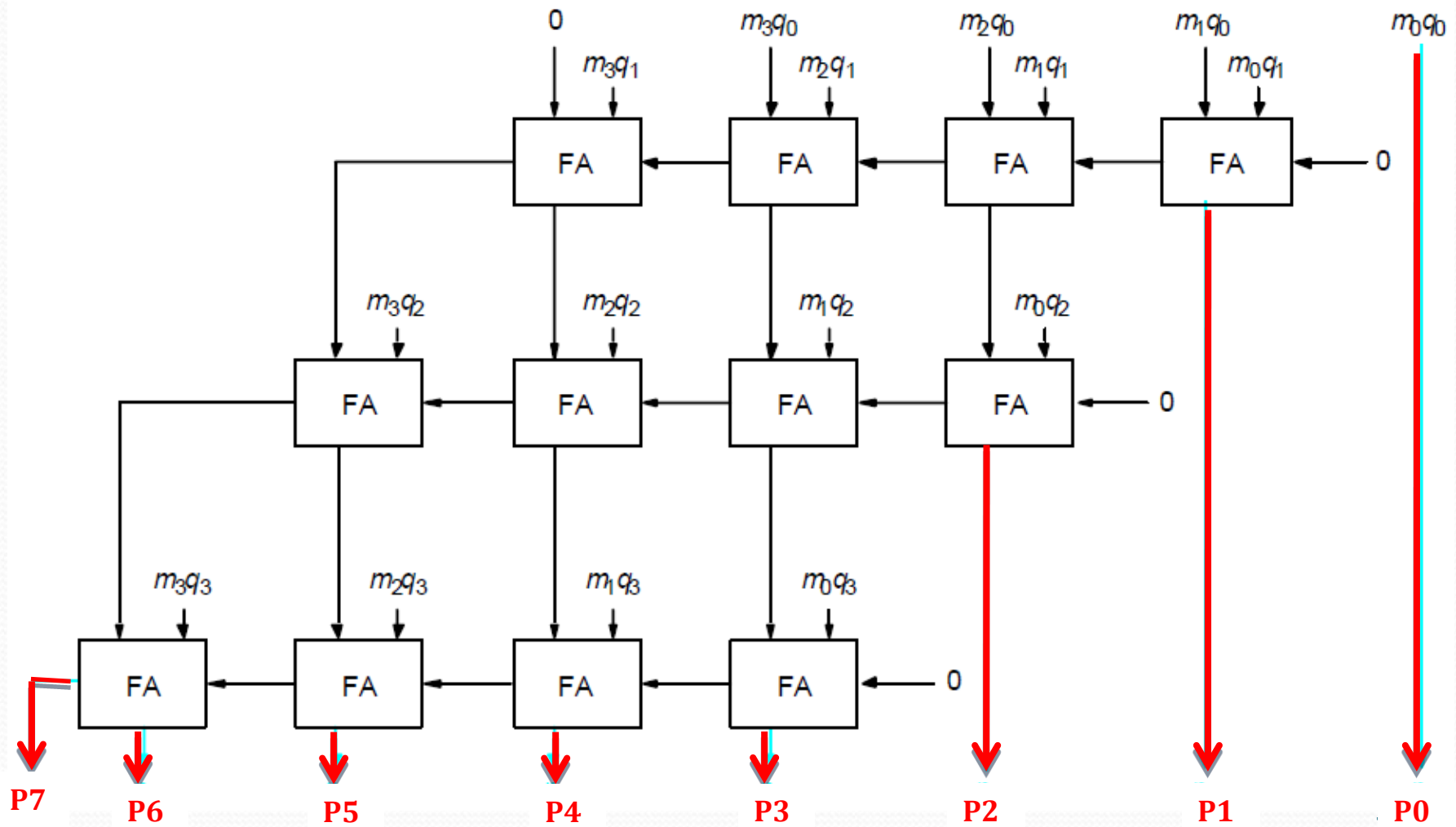


$$\begin{array}{r}
 \begin{array}{ccccc} 0 & 1 & 1 & 0 & 1 \\ 0 & -1 & -2 & & \end{array} \\
 \hline
 \begin{array}{cccccc} 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array} \\
 \hline
 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0
 \end{array}$$

Figure 6.15. Multiplication requiring only  $n/2$  summands.<sup>38</sup>

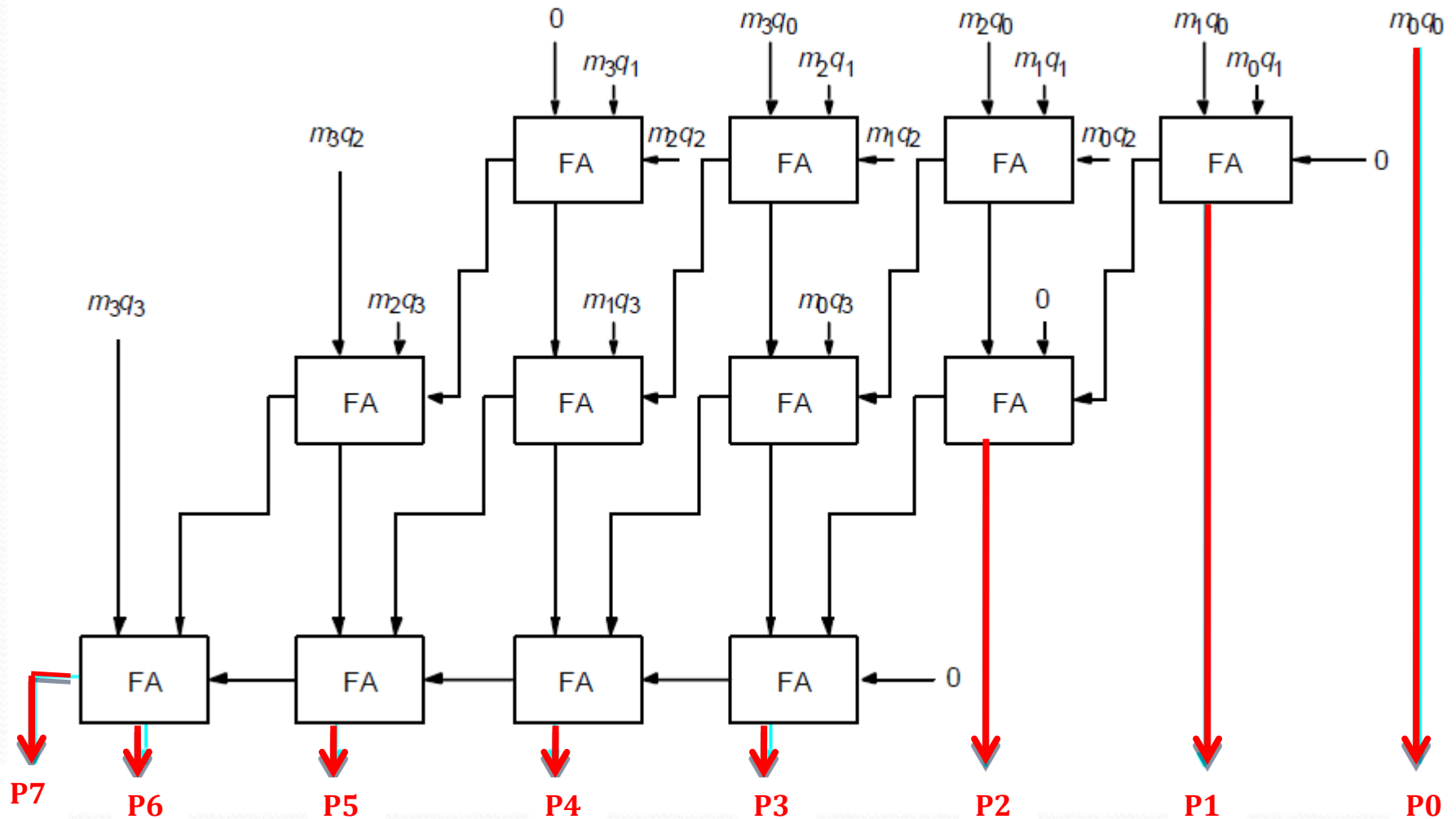
# Carry-Save Addition of Summands

- 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

# 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.





# Integer Division

# Manual Division

$$\begin{array}{r} 21 \\ 13 \overline{) 274} \\ \underline{26} \phantom{0} \\ 14 \\ \underline{13} \\ 1 \end{array}$$

$$\begin{array}{r} 10101 \\ 1101 \overline{) 100010010} \\ \underline{1101} \phantom{0000} \\ 10000 \\ \underline{1101} \phantom{000} \\ 1110 \\ \underline{1101} \\ 1 \end{array}$$

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, 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

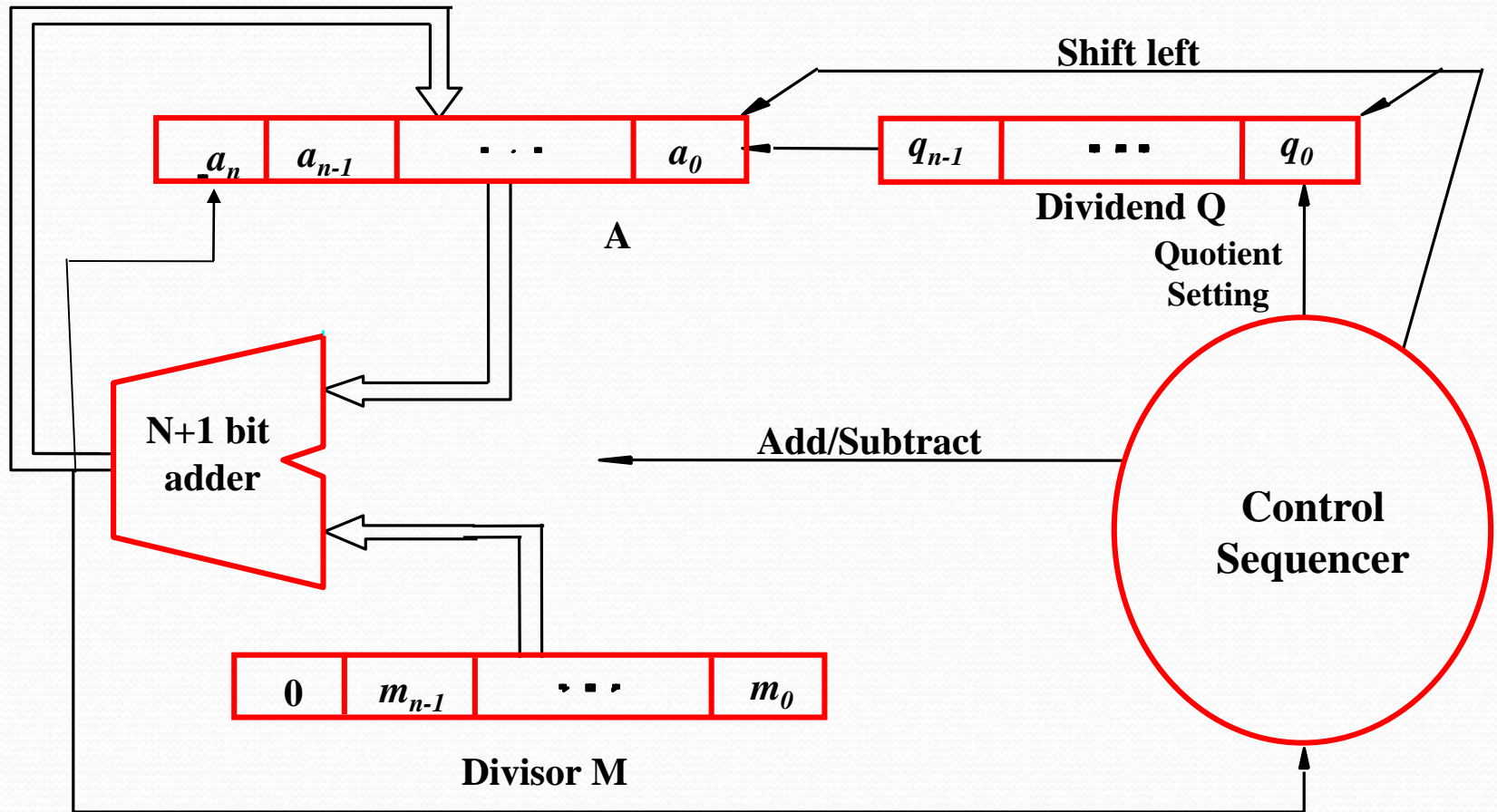


Figure 6.21. Circuit arrangement for binary division.



# 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

$$\begin{array}{r}
 10 \\
 11 \overline{) 1000} \\
 \underline{11} \phantom{0} \\
 10
 \end{array}$$

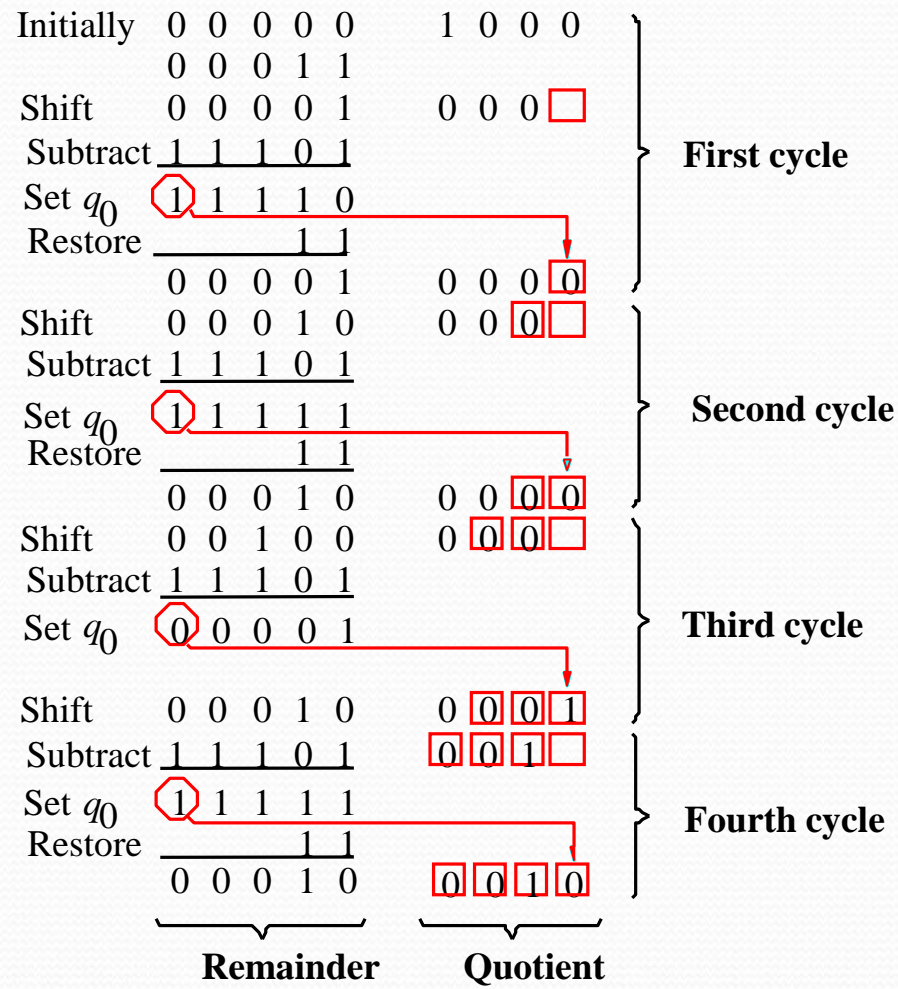


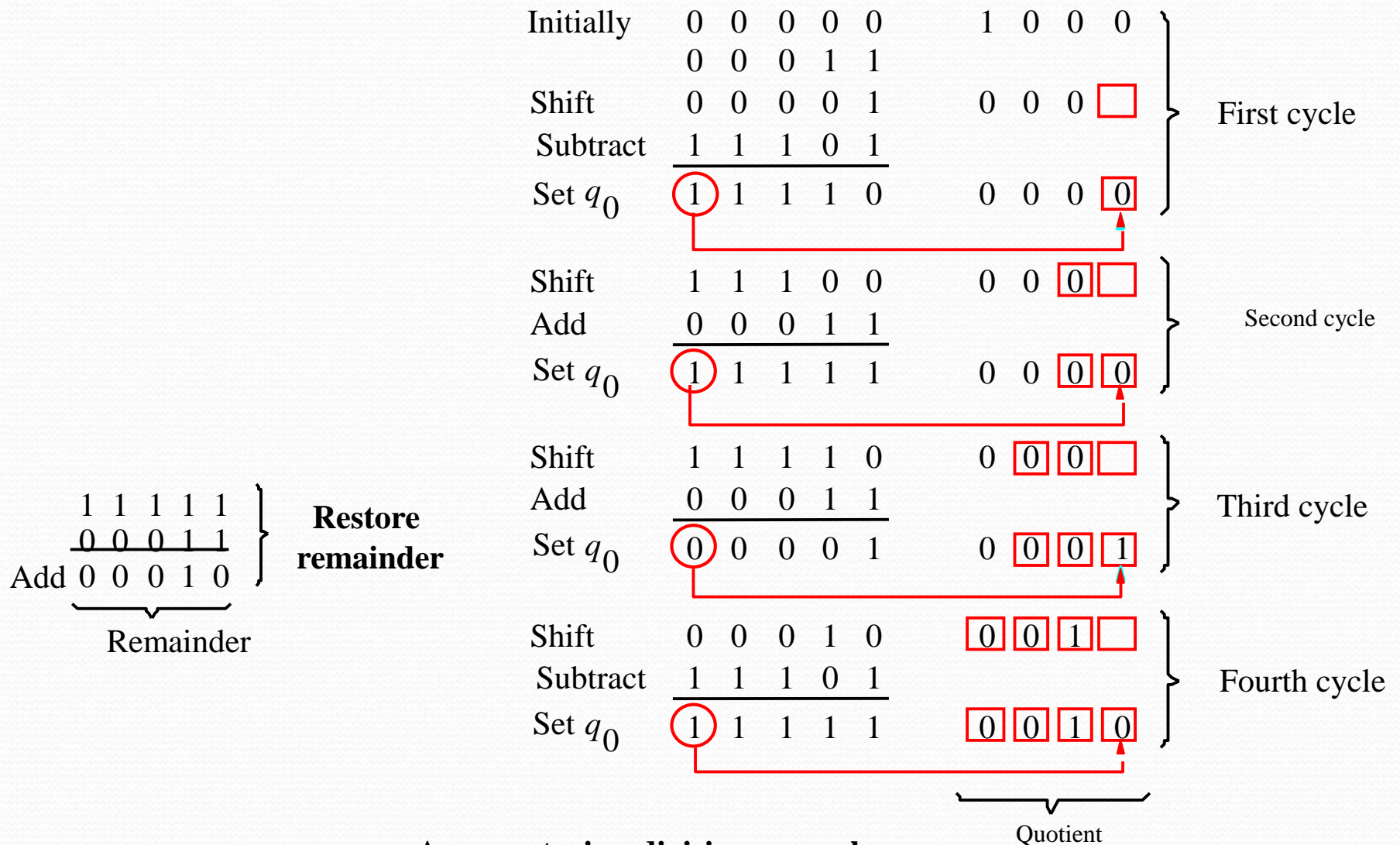
Figure 6.22. A restoring-division example.

# 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.**