

Chapter 9

Arithmetic

Chapter Outline

- Adders, subtractors, multipliers, and dividers
- High-speed adders
 using carry-lookahead
- High-speed multipliers
 using carry-save addition trees
- IEEE standard floating-point arithmetic

Addition

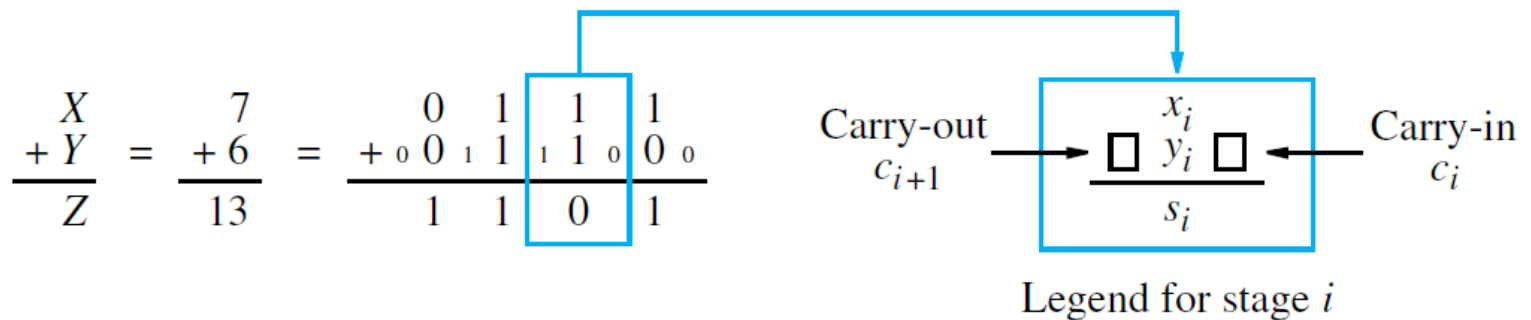
- Full adder (FA) logic circuit:
adds two bits of the same weight, along with a carry-in bit, and produces a sum bit and a carry-out bit
- Ripple-carry adder:
a chain of n FA stages, linked by carry bits, can add two n -bit numbers

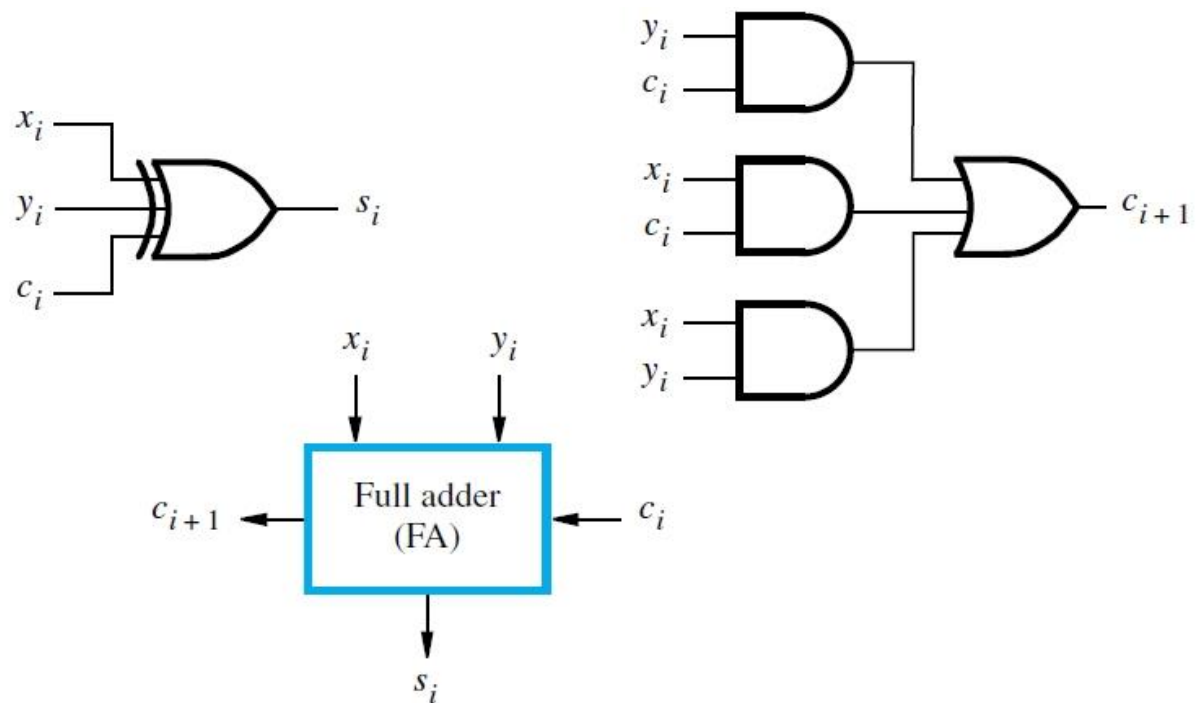
x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

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

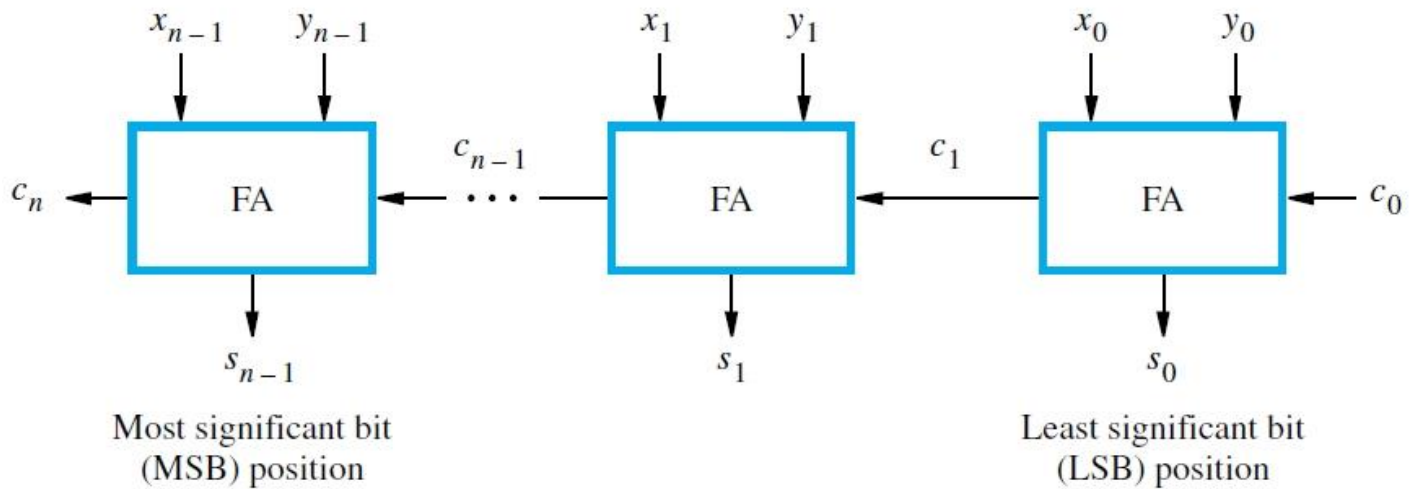
$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

Example:





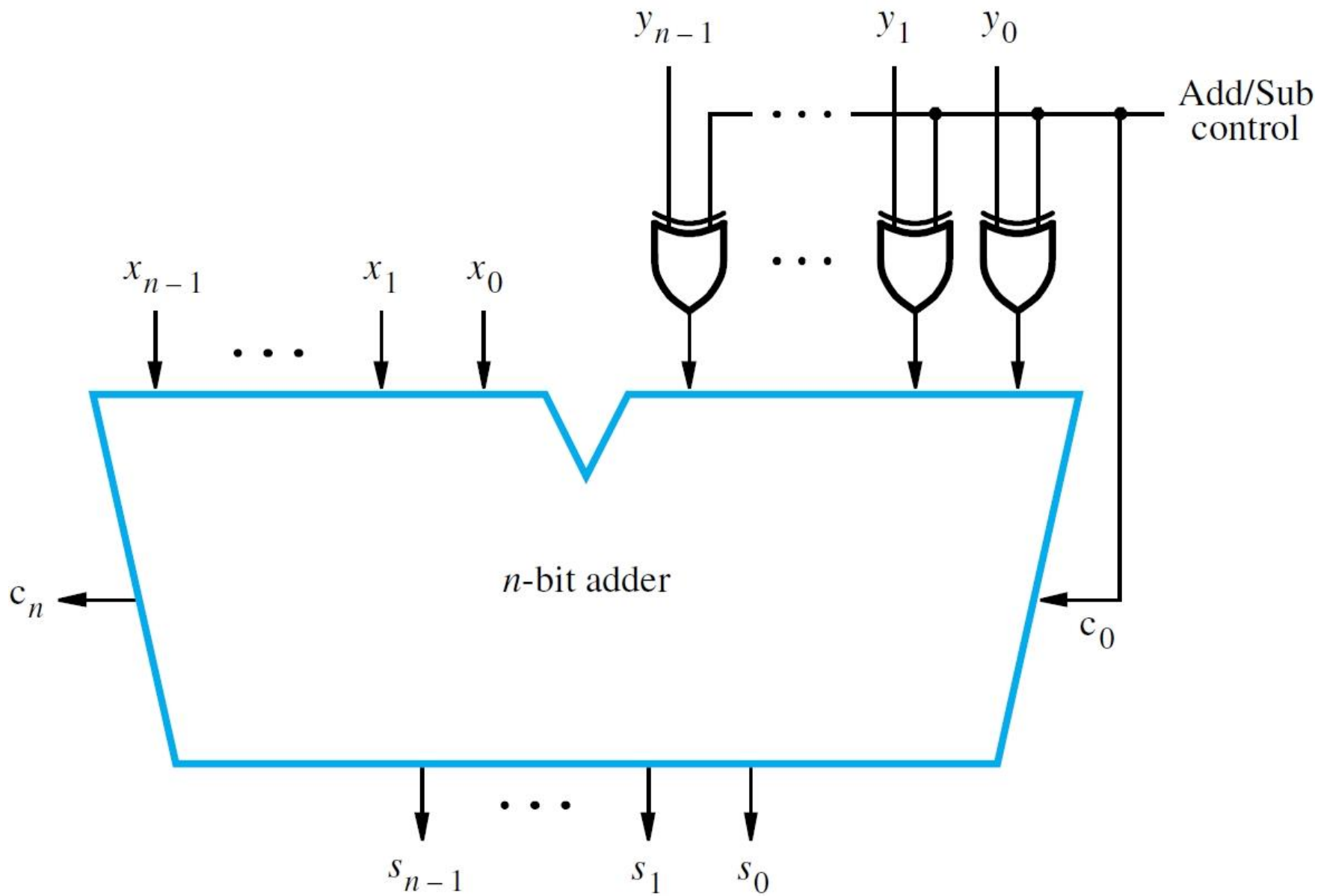
(a) Logic for a single stage



(b) An n -bit ripple-carry adder

Addition/subtraction circuit

- An n -bit adder with external XOR gates can add or subtract two operands
- An FA stage produces its outputs after 2 logic gate delays
- Longest **delay path** through the adder/subtractor circuit: $2n$ gate delays, assuming a ripple-carry design



Carry-lookahead addition

- Delay reduction: produce carry signals in parallel using carry-lookahead circuits
- First, form **generate** and **propagate** functions in each stage i

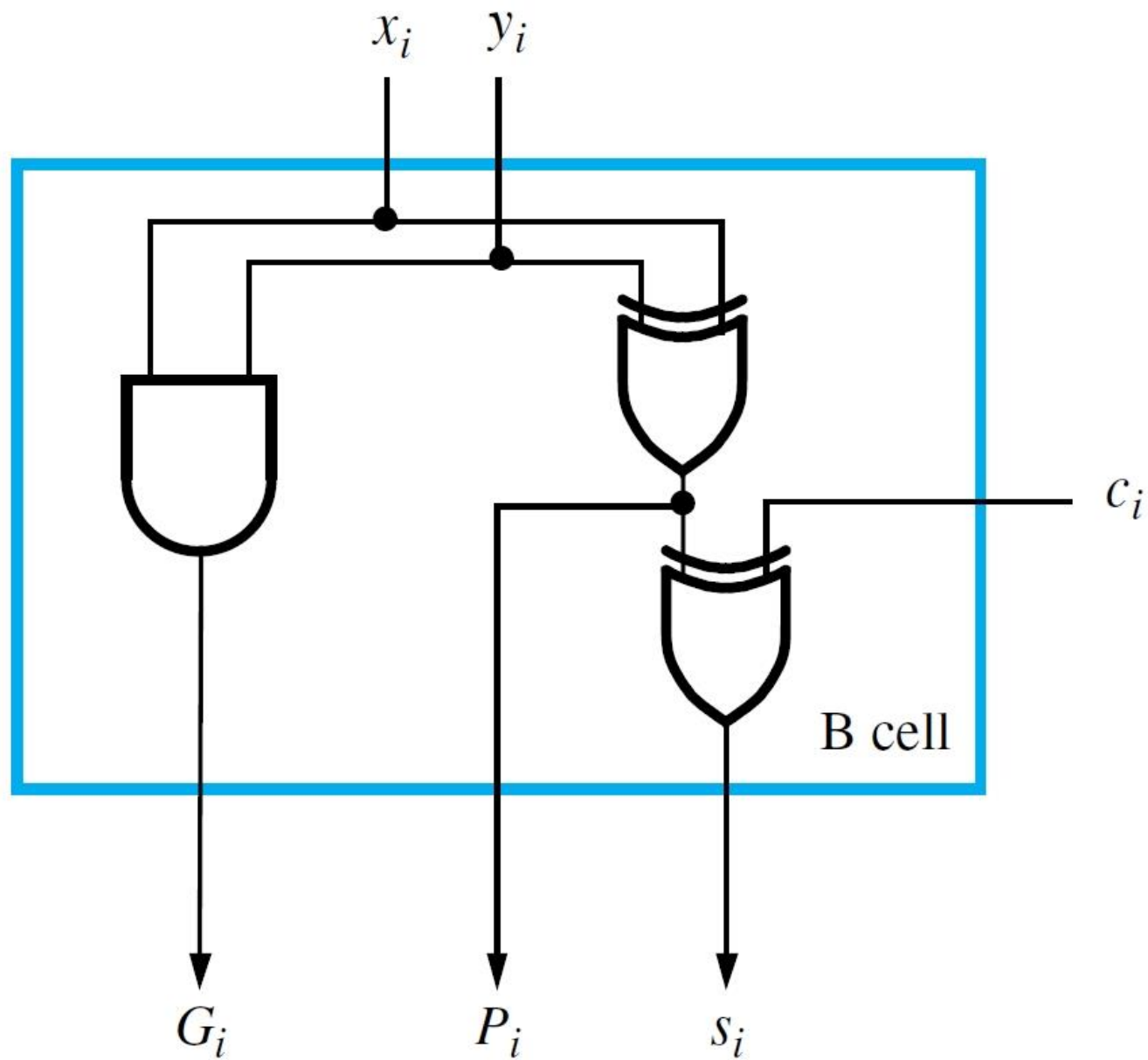
$$C_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

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

$$G_i = x_i y_i \qquad P_i = x_i + y_i$$

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

P_i can be treated as XOR of x_i and y_i (Why??)



Carry-lookahead circuits

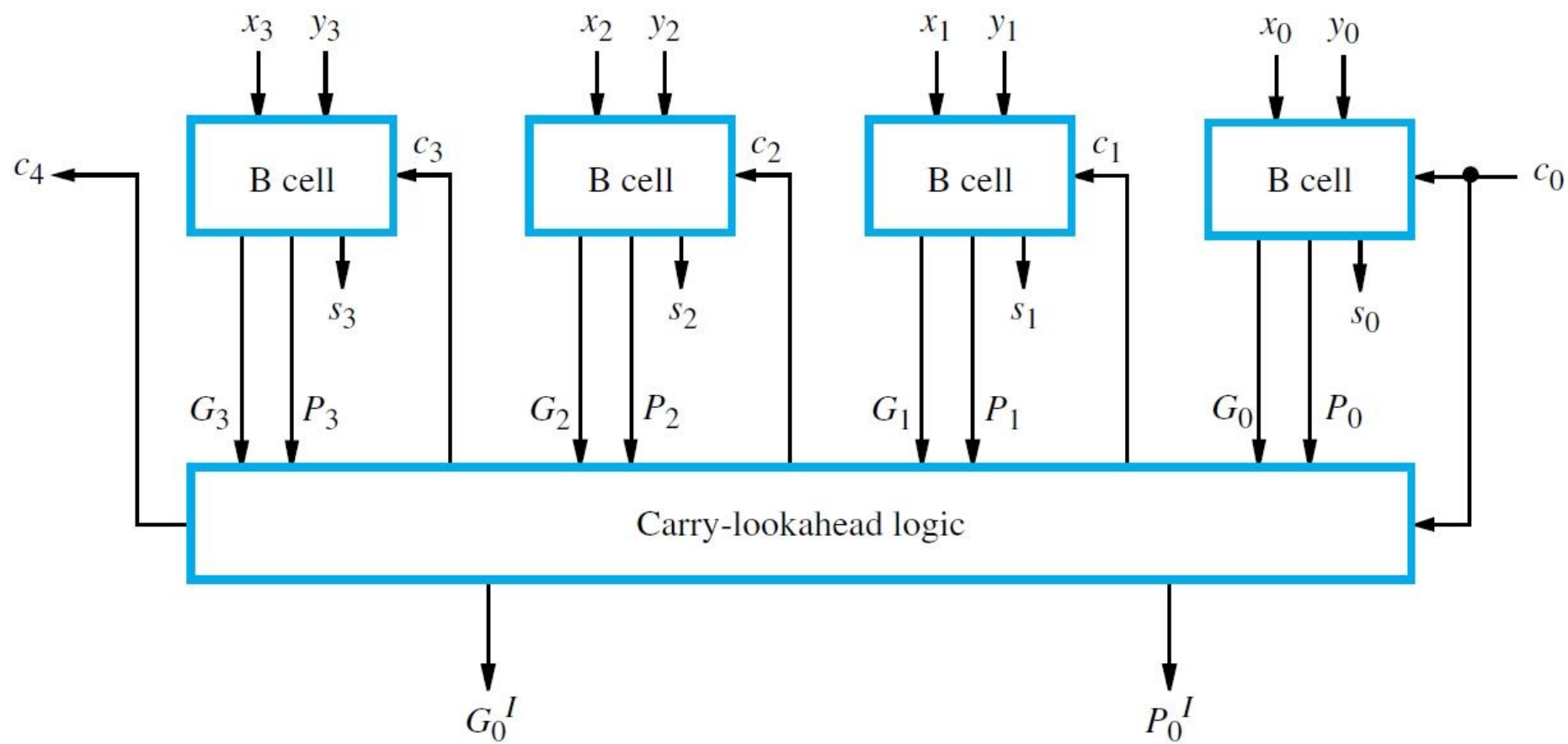
- A 4-bit adder has four carry-out signals:

$$c_1 = G_0 + P_0c_0$$

$$c_2 = G_1 + P_1G_0 + P_1P_0c_0$$

$$c_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0$$

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



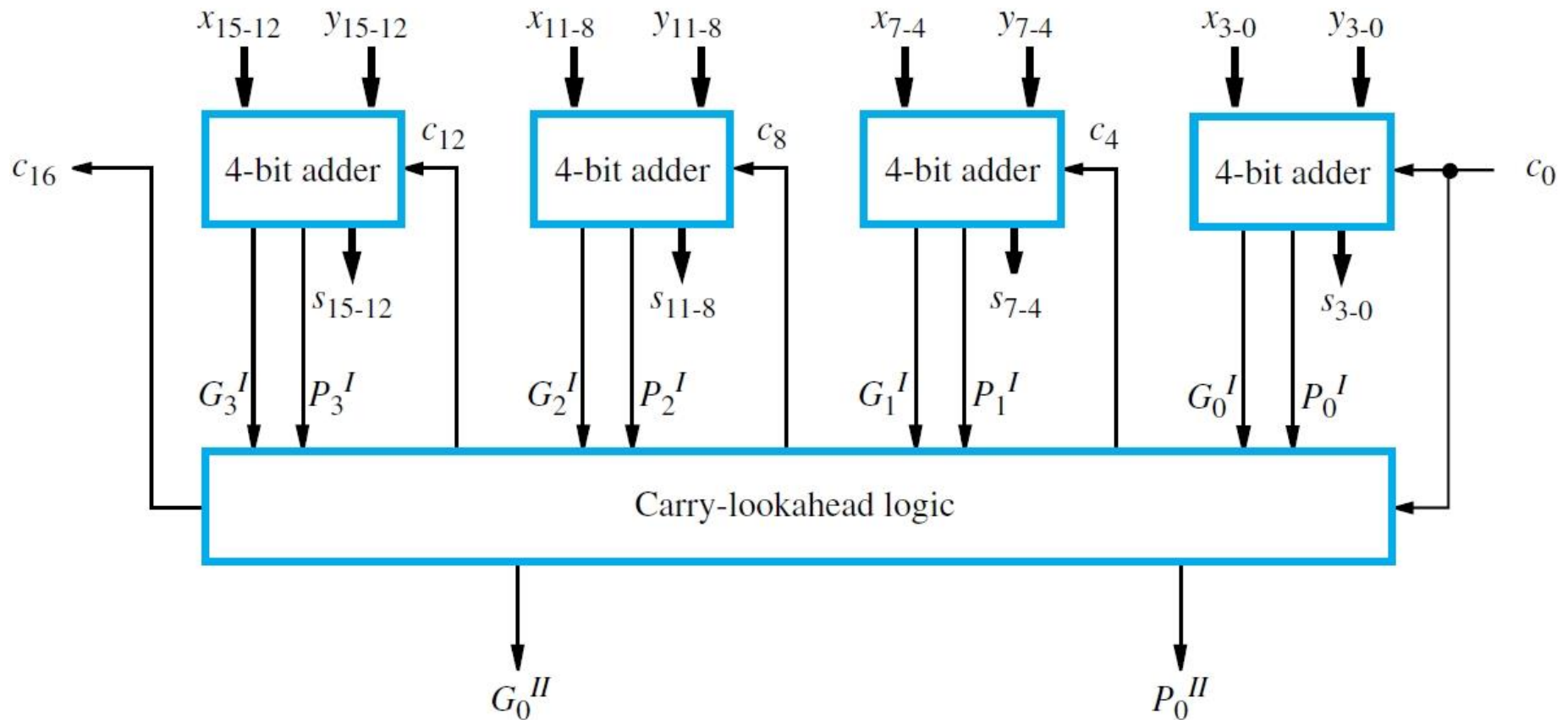
Delay in 4-bit adder

- Ripple-carry design:
8 gate delays: $(2 \text{ for each FA}) \times 4$
- Carry-lookahead design:
1 for all P_i and G_i
2 for all c_i
+ 1 for all s_i

4 gate delays

Carry-lookahead for larger n

- Ideally, 2 gate delays for all c_i regardless of n
- But max. number of inputs for AND/OR gates increases linearly with n
- **Fan-in constraints** for actual logic gates makes 2-level logic for c_i less practical for $n > 4$
- Therefore, higher-level gen./prop. functions are used to produce carry bits in parallel for 4-bit and 16-bit adder blocks



$$P_0^I = P_3 P_2 P_1 P_0 \quad G_0^I = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

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

Delays in larger adders using higher-level gen./prop.

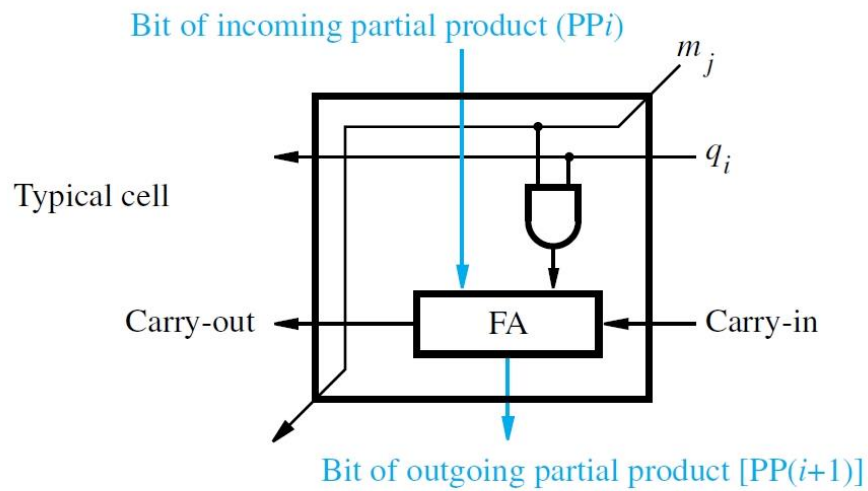
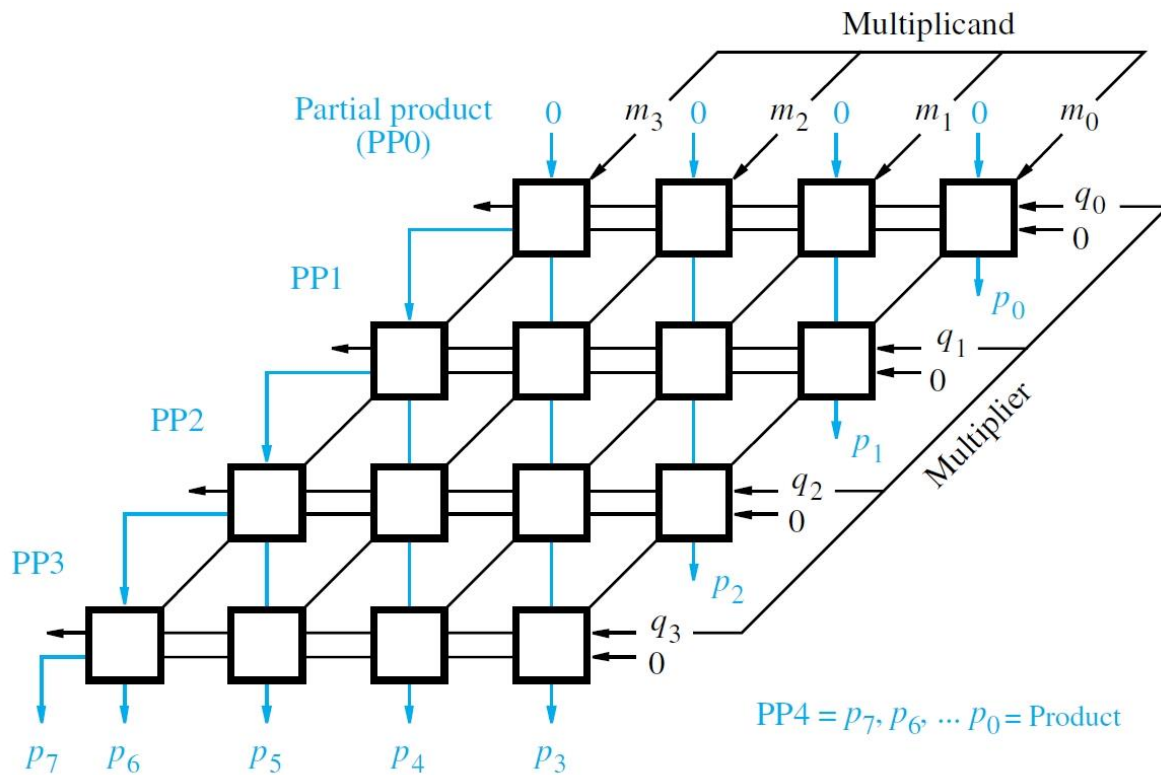
- 16-bit adder:
 - 8 gate delays for carry-lookahead
using 4-bit adder blocks
(pure ripple-carry requires 32 gate delays)
- 64-bit adder:
 - 12 gate delays for carry-lookahead
using four of the above 16-bit adders
(pure ripple-carry requires 128 gate delays)
- Higher-level gen./prop. adder still *much* faster

Multiplication

- Two, n -bit, unsigned numbers produce a $2n$ -bit product when they are multiplied
- Multiplication can be done in a 2-dimensional **combinational array** composed of n^2 basic cells, each containing an FA block, arranged in a trapezoidal shape
- Longest delay path is approx. $6n$ gate delays, along the right edge and across the bottom of the array

				1	1	0	1		(13) Multiplicand M
			×	1	0	1	1		(11) Multiplier Q
				<hr/>					
				1	1	0	1		
			1	1	0	1			
		0	0	0	0				
	1	1	0	1					
<hr/>									
1	0	0	0	1	1	1	1		(143) Product P

(a) Manual multiplication algorithm

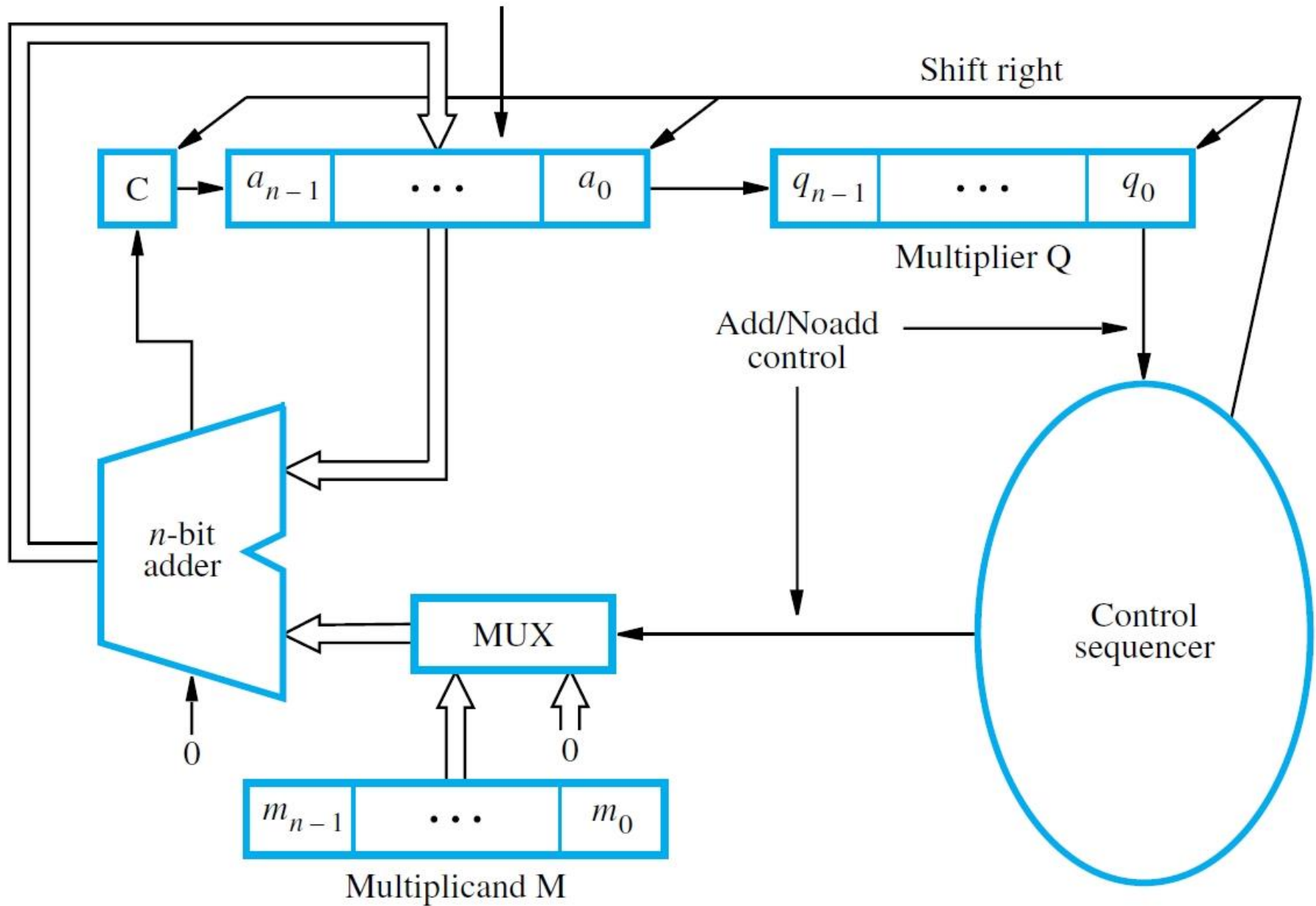


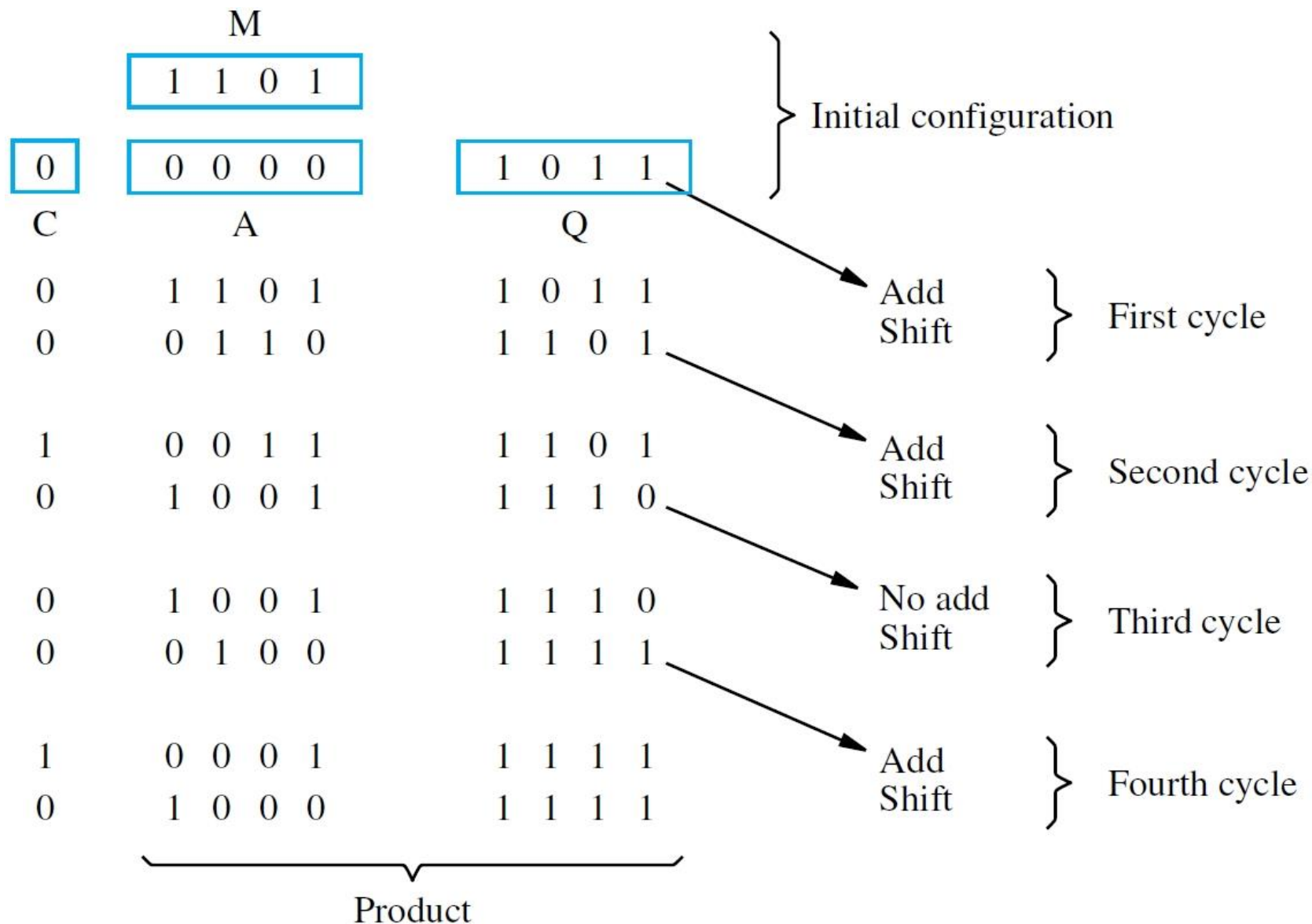
(b) Array implementation

Multiplication

- Sequential circuit multiplier
is composed of three n -bit registers, an n -bit adder, and a control sequencer
- A sequence of n addition cycles generates a $2n$ -bit product

Register A (initially 0)





(b) Multiplication example

Multiplying signed numbers

- The next six figures, Figures 6.8 through 6.13 from the textbook, show how to perform multiplication of signed numbers in 2's-complement representation
- Dealing with a negative multiplicand with basic sign extension is described first
- Then, the Booth algorithm is introduced as a way to deal with negative multipliers
- Benefit of Booth: potentially fewer additions

Sign extension is
shown in blue

$$\begin{array}{r}
 \begin{array}{ccccccccc}
 & & & & & 1 & 0 & 0 & 1 & 1 & (-13) \\
 & & & & & \times & 0 & 1 & 0 & 1 & 1 & (+11) \\
 \hline
 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & \\
 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & & \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & \\
 1 & 1 & 1 & 0 & 0 & 1 & 1 & & & & \\
 0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
 \hline
 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & (-143)
 \end{array}
 \end{array}$$

Sign extension of negative multiplicand.

Booth Multiplication

$$\begin{array}{r}
 0011110 \\
 0100000 \\
 - 0000010 \\
 \hline
 0011110
 \end{array}$$

(32)
(2)
(30)

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. For convenience, we can describe the

sequence of required operations by recoding the preceding multiplier as $0 +1 0 0 0 -1 0$.

								0	1	0	1	1	0	1
								0	0	+1	+1	+1	+1	0
								<hr/>						
								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						
	<hr/>													
0	0	0	1	0	1	0	1	0	0	0	1	1	0	

								0	1	0	1	1	0	1
								0	+1	0	0	0	-1	0
								<hr/>						
								0	0	0	0	0	0	0
								0	1	0	0	1	1	
								0	0	0	0	0		
								0	0	0	0	0		
								0	0	0	0	0		
								0	0	0	1	0	1	
								0	0	0	0	0		
								<hr/>						
0	0	0	1	0	1	0	1	0	0	0	1	1	0	

2's complement of
the multiplicand
(With appropriate
shifting)



Normal and Booth multiplication schemes.

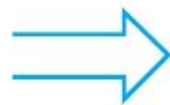
0 0 1 0 1 1 0 0 1 1 1 0 1 0 1 1 0 0



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

Booth recoding of a multiplier.

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




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

Booth multiplication with a negative multiplier.


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.


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 recoded multipliers.

High-speed multipliers

- Neither the combinational array nor the sequential circuit multiplier are fast enough for high performance processors
- Two approaches are used for higher speed:
 1. Reduce the number of summands
 2. Use more parallelism in adding them

Reducing summands

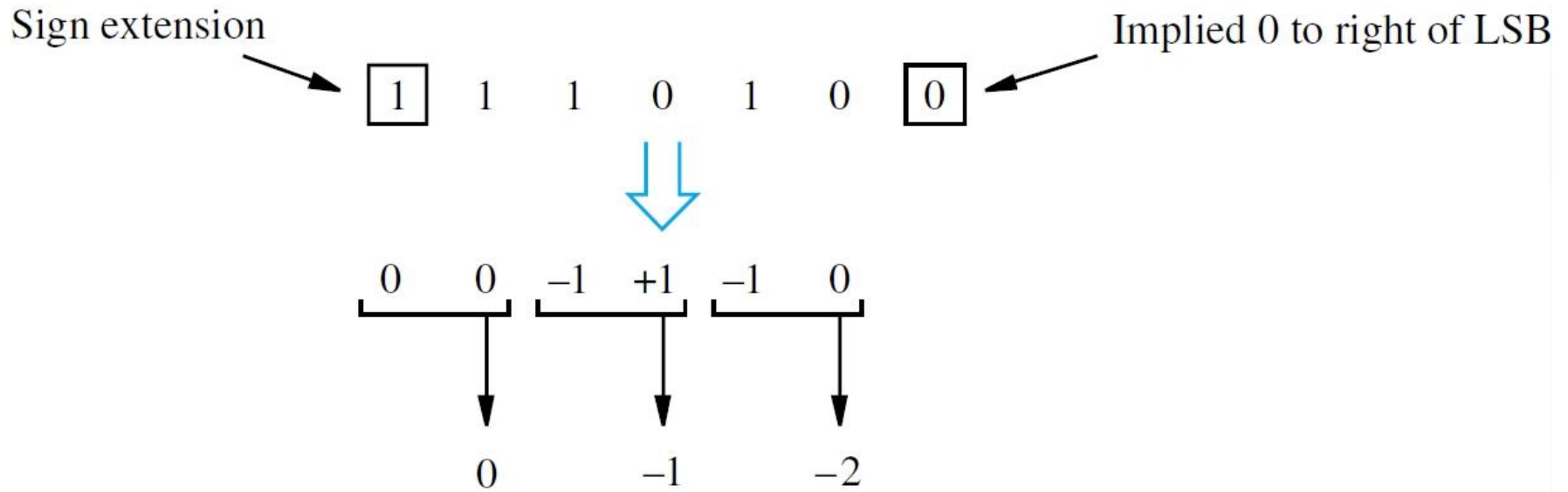
- Normally, to multiply a number M by 15_{10} ($=1111_2$), four shifted versions of M are added
- Alternatively, the same result is obtained by computing $16M - M$, where $16M$ is formed by shifting M to the left 4 times
- This basic idea, derived from the Booth algorithm, can be applied to reduce the number of summands

Reducing summands

- Each pair of multiplier bits selects one summand from 5 possible versions of the multiplicand M : $0, M, -M, 2M, -2M$
- Example: 6-bit, 2's-complement operands

Multiplier $Q =$ 1 1 1 0 1 0

M version selected = 0 -1 -2



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

Multiplier bit-pair recoding

- The full table of multiplicand selection decisions based on bit-pairing of the multiplier is shown in the next figure
- Since only one version of the multiplicand is added into the partial product for each pair of multiplier bits, only $n/2$ summands are added to do an $n \times n$ multiplication

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



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

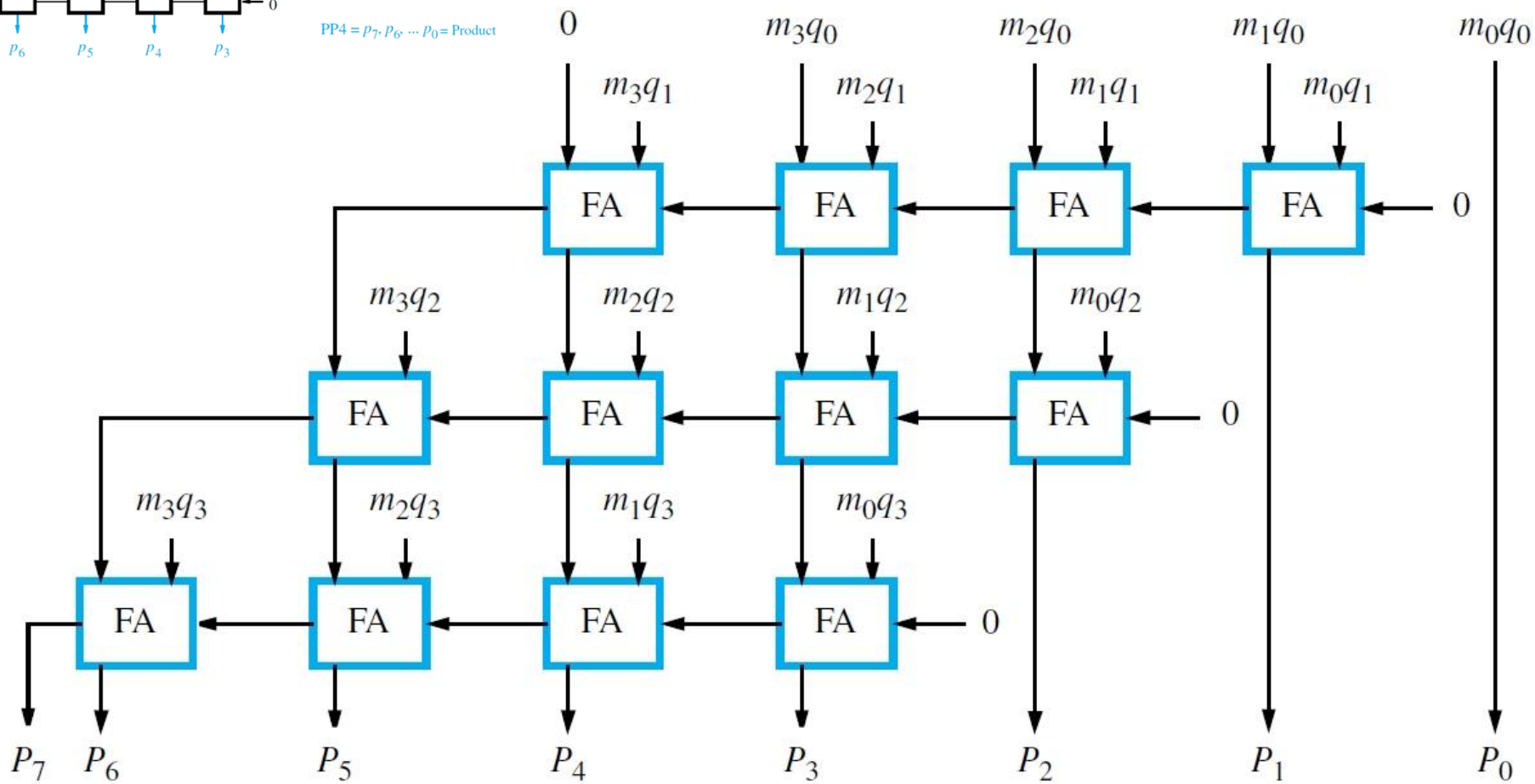
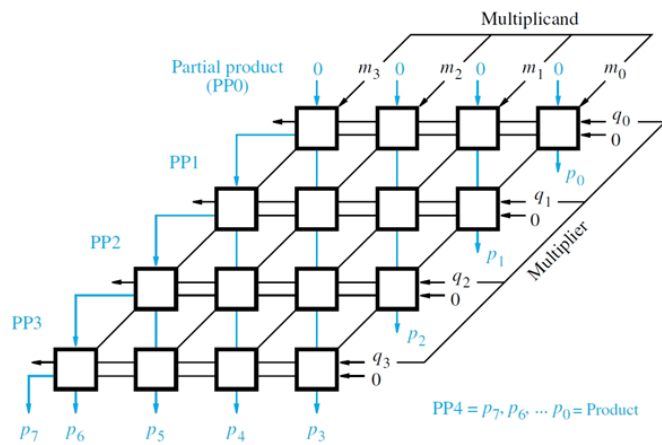


$$\begin{array}{r}
 0 \ 1 \ 1 \ 0 \ 1 \\
 0 \ \ \ -1 \ \ \ -2 \\
 \hline
 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \\
 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \\
 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
 \hline
 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0
 \end{array}$$

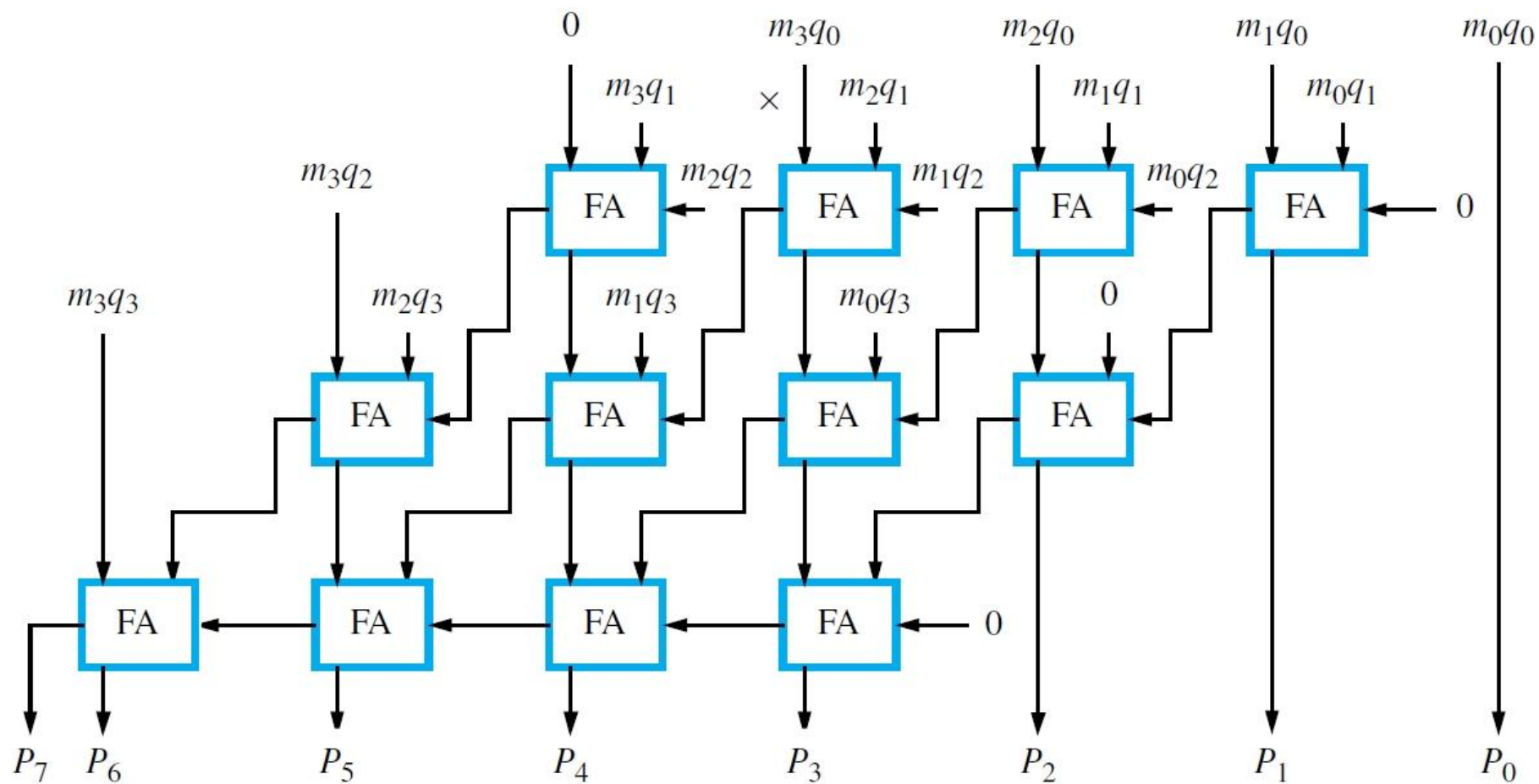
Multiplication requiring only $n/2$ summands.

Parallelism in adding summands

- Three n -bit summands can be reduced to two by using n FA blocks, operating independently and in parallel
- This technique can be applied in the array multiplier, as shown in the next two figures
- The technique is called **carry-save addition**



(a) Ripple-carry array



(b) Carry-save array

Carry-save addition

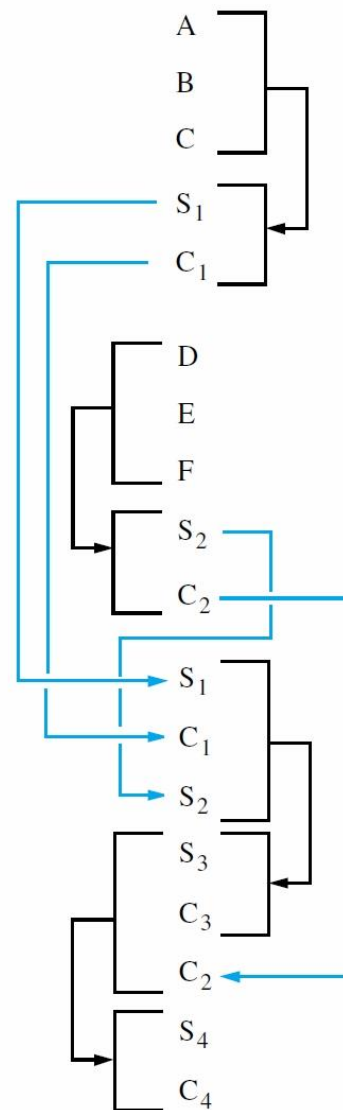
- More parallelism than exploited in the previous figure can be achieved
- Group summands in threes and reduce each group to two in parallel
- Repeat until only two summands remain
- Add them in a conventional adder to generate the final sum

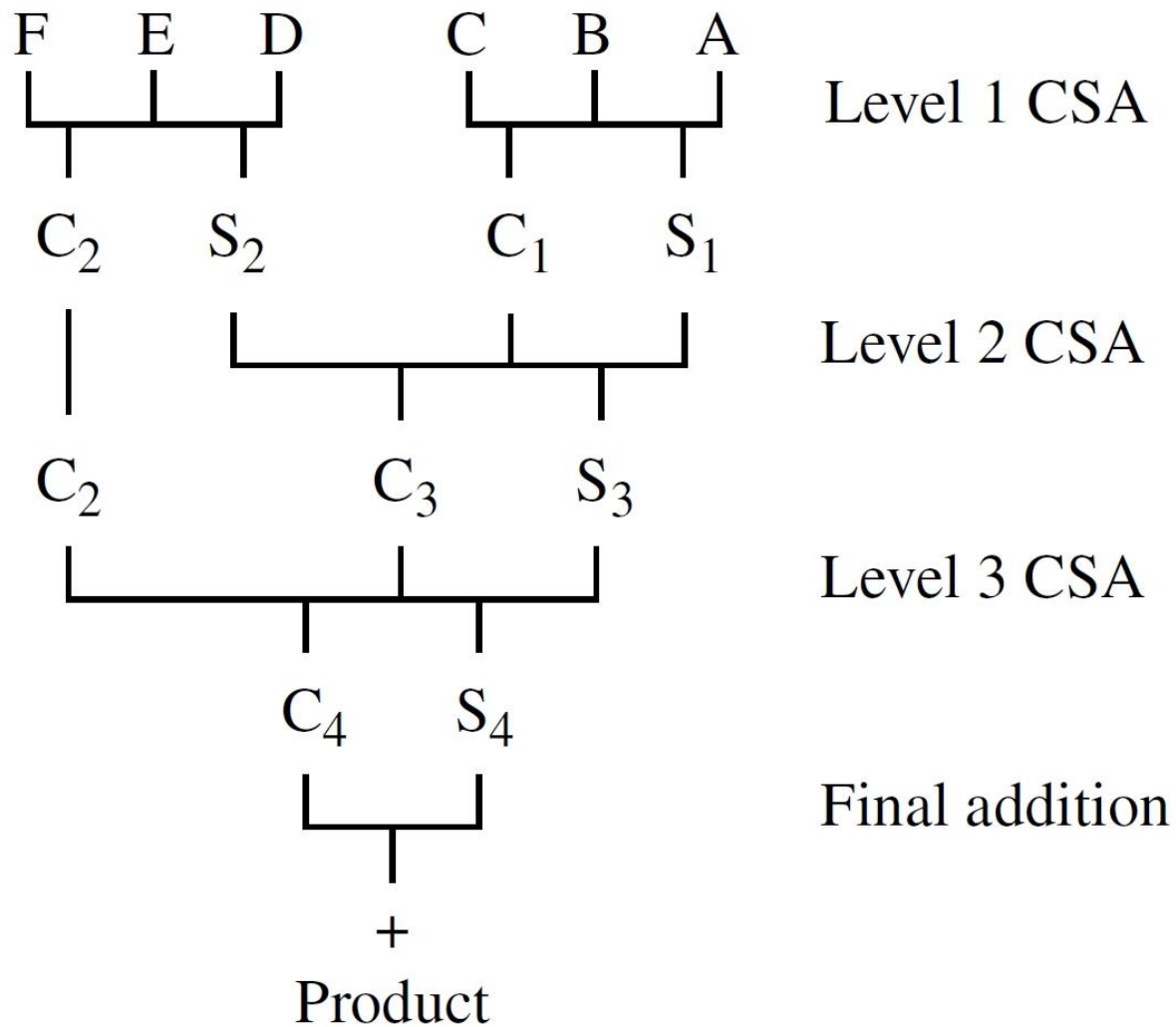
							1	0	1	1	0	1	(45)	M
						×	1	1	1	1	1	1	(63)	Q
<hr/>														
							1	0	1	1	0	1	A	
					1		0	1	1	0	1		B	
			1	0			1	1	0	1			C	
				1	0	1	1	0	1				D	
		1	0	1	1		0	1					E	
1	0	1	1	0			1						F	
<hr/>														
1	0	1	1	0	0	0	0	1	0	0	1	1	(2,835)	Product

A multiplication example used to illustrate carry-save addition.

$$\begin{array}{r}
 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ M \\
 \times 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ Q \\
 \hline
 \end{array}$$

$$\begin{array}{r}
 1 \ 0 \ 1 \ 1 \ 0 \ 1 \\
 1 \ 0 \ 1 \ 1 \ 0 \ 1 \\
 1 \ 0 \ 1 \ 1 \ 0 \ 1 \\
 \hline
 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \\
 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \\
 \hline
 1 \ 0 \ 1 \ 1 \ 0 \ 1 \\
 1 \ 0 \ 1 \ 1 \ 0 \ 1 \\
 1 \ 0 \ 1 \ 1 \ 0 \ 1 \\
 \hline
 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \\
 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \\
 \hline
 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \\
 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \\
 \hline
 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \\
 + 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
 \hline
 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ \text{Product}
 \end{array}$$





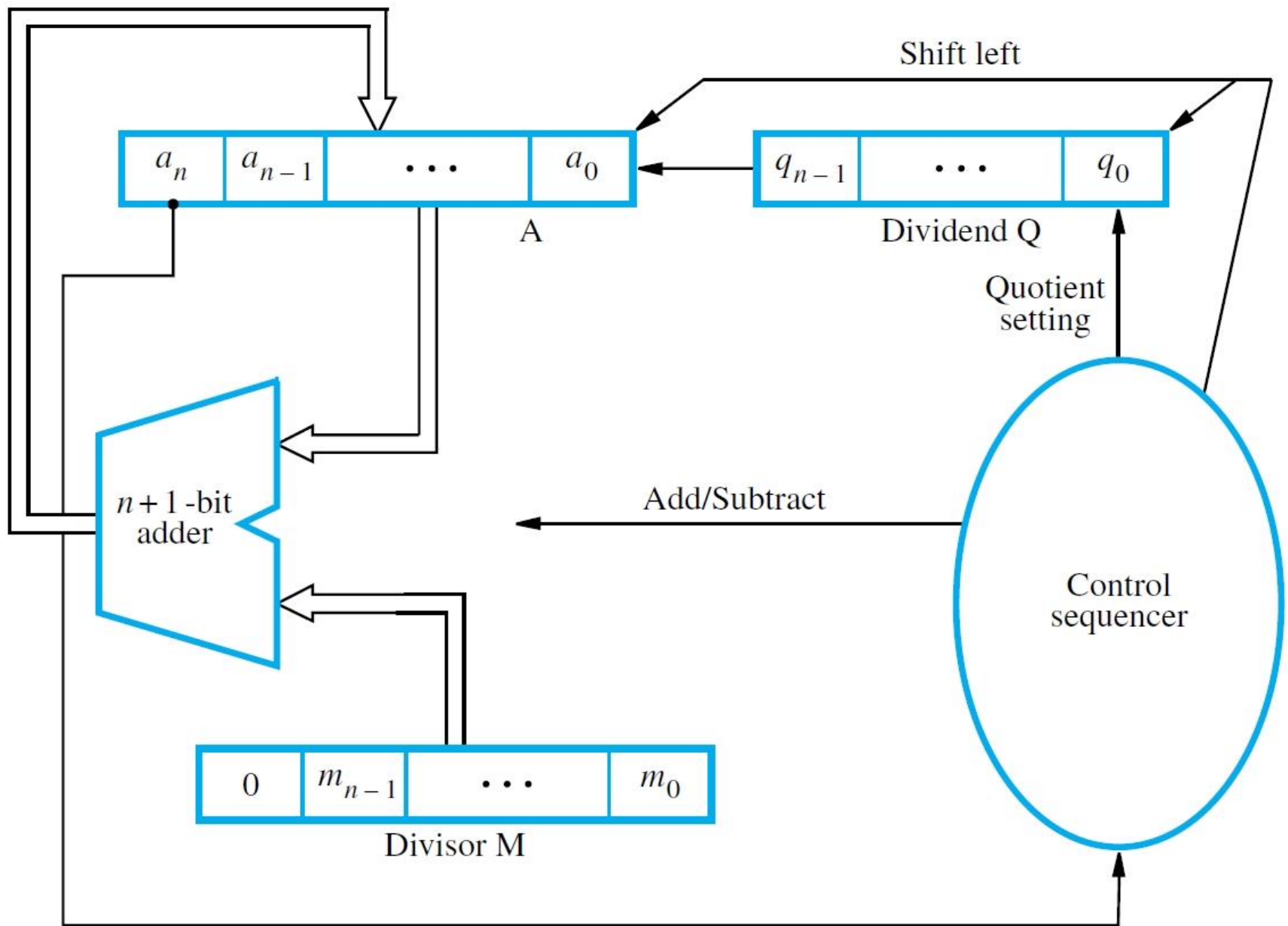
Schematic representation of
carry-save addition requirements.

Division

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

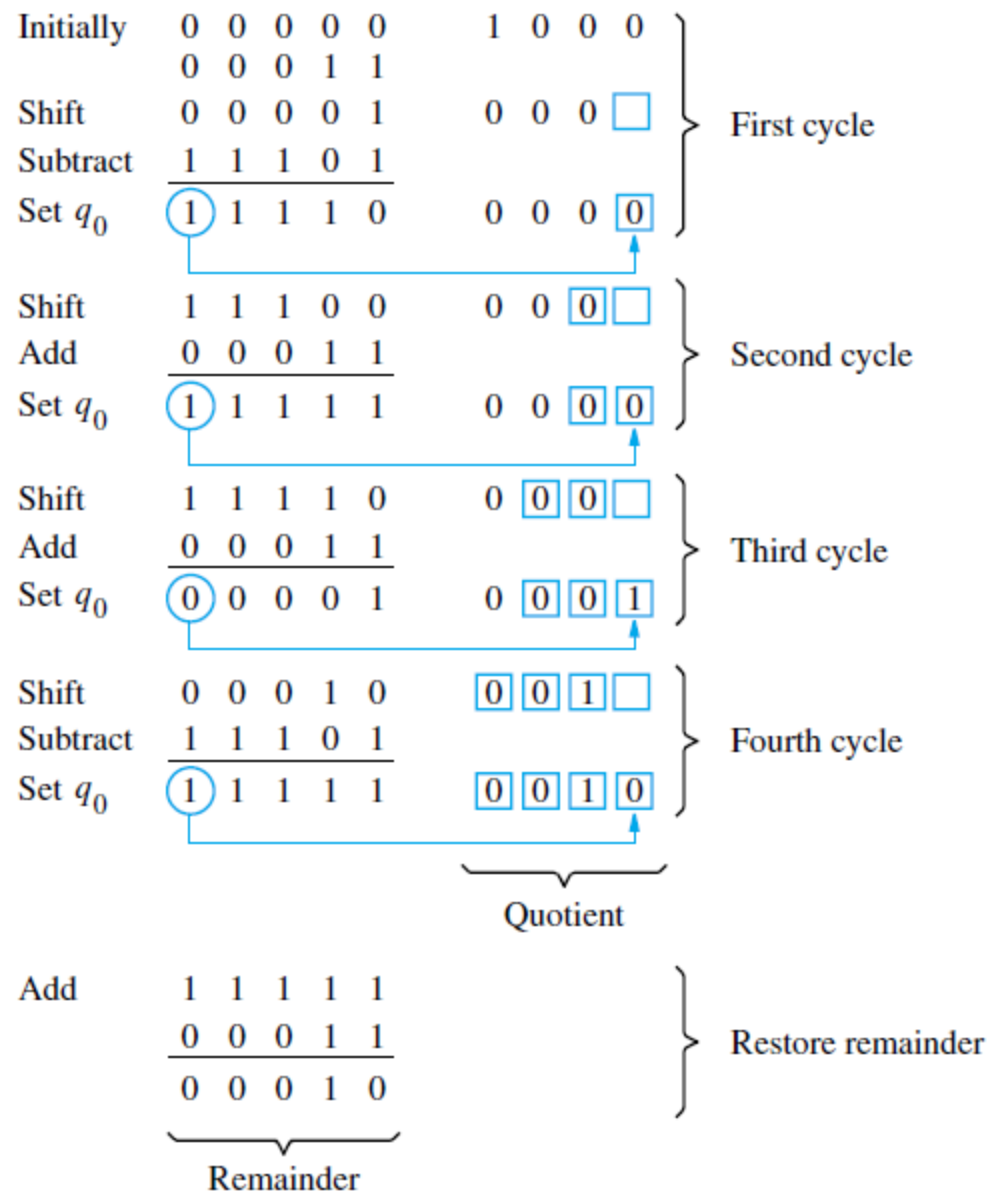
$$\begin{array}{r} 10101 \\ 1101 \overline{) 100010010} \\ \underline{1101} \\ 10000 \\ \underline{1101} \\ 1110 \\ \underline{1101} \\ 1 \end{array}$$

Longhand division examples.



Initially	0	0	0	0	0	1	0	0	0	}	First cycle
	0	0	0	1	1						
Shift	0	0	0	0	1	0	0	0	<input type="checkbox"/>		
Subtract	1	1	1	0	1						
Set q_0	1	1	1	1	0	↓				}	Second cycle
Restore				1	1						
	0	0	0	0	1	0	0	0	0		
Shift	0	0	0	1	0	0	0	0	<input type="checkbox"/>		
Subtract	1	1	1	0	1					}	Third cycle
Set q_0	1	1	1	1	1	↓					
Restore				1	1						
	0	0	0	1	0	0	0	0	0		
Shift	0	0	1	0	0	0	0	0	<input type="checkbox"/>	}	Fourth cycle
Subtract	1	1	1	0	1						
Set q_0	0	0	0	0	1	↓					
Shift	0	0	0	1	0	0	0	0	1		
Subtract	1	1	1	0	1	0	0	1	<input type="checkbox"/>		
Set q_0	1	1	1	1	1	↓					
Restore				1	1						
	0	0	0	1	0	0	0	1	0		
Remainder					Quotient						

A restoring-division example.



A nonrestoring-division example.

Floating-point (FP) numbers

- IEEE standard 754-2008 defines representation and operations for floating-point numbers

- The 32-bit single-precision format is:

A **sign** bit: S (0 for +, 1 for -)

An 8-bit signed **exponent**: E (base = 2)

A 23-bit **mantissa** fraction magnitude: M

FP numbers

- The value represented is

$$\pm 1.M \times 2^E$$

- E is actually encoded as $E' = E + 127$

which is called an **excess-127** representation

FP numbers

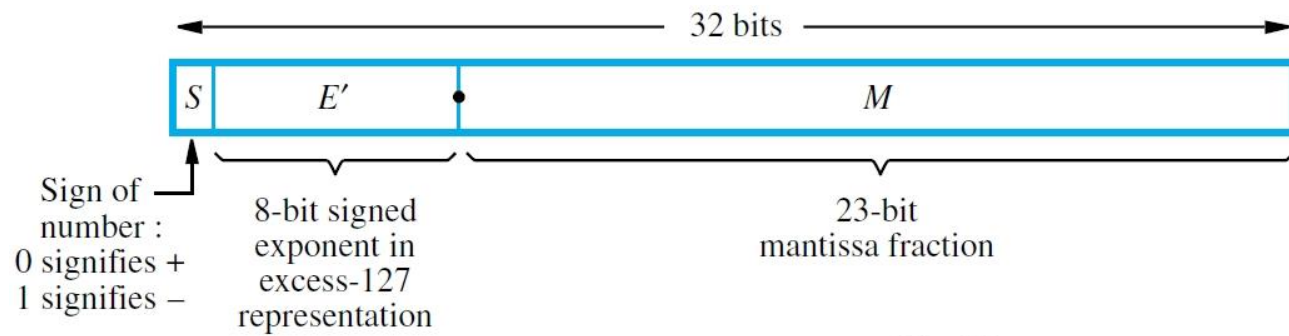
- Example of 32-bit number representation:

0	10000101	0110 ...
S	E'	M

- Value represented (with $E = E' - 127 = 133 - 127 = 6$):

$$+ 1.0110 \times 2^6$$

- This is called a **normalized representation**,
with binary point to the right of first significant bit
- 64-bit double-precision is similar with more bits for E' & M



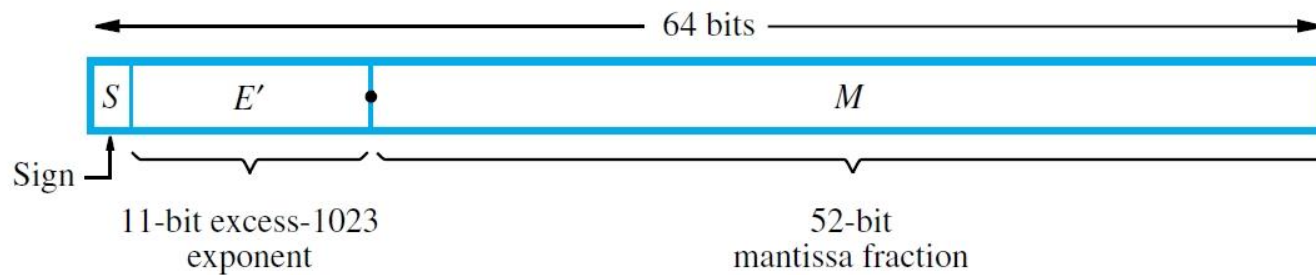
$$\text{Value represented} = \pm 1.M \times 2^{E' - 127}$$

(a) Single precision



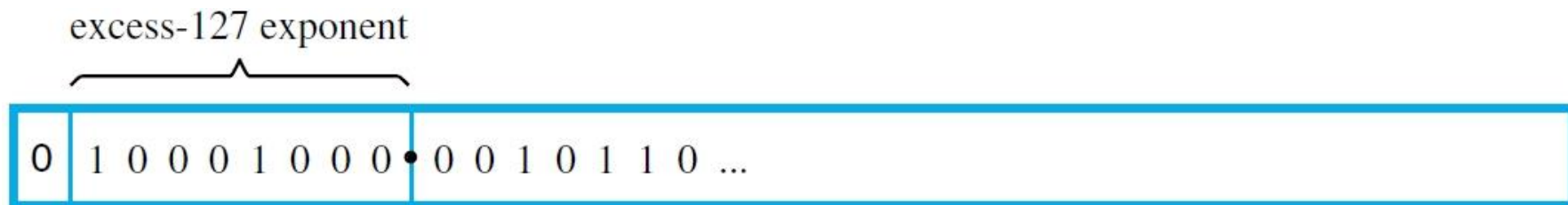
$$\text{Value represented} = 1.001010 \dots 0 \times 2^{-87}$$

(b) Example of a single-precision number



$$\text{Value represented} = \pm 1.M \times 2^{E' - 1023}$$

(c) Double precision



(There is no implicit 1 to the left of the binary point.)

$$\text{Value represented} = + 0.0010110... \times 2^9$$

(a) Unnormalized value



$$\text{Value represented} = + 1.0110... \times 2^6$$

(b) Normalized version

Try examples at: <http://evanw.github.io/float-toy/>

Overflow/ Underflow

- During computation, a number outside the representable range might be generated
- In single precision, this means normalized representation requires an exponent less than -126 or greater than $+127$
- In the first case, we say that *underflow* has occurred
- In the second case, we say that *overflow* has occurred

Special Values

- End values 0 and 255 of the excess-127 exponent E' are used to represent special values.
 - When $E' = 0$ and the mantissa fraction M is zero, the value 0 is represented.
 - When $E' = 255$ and $M = 0$, the value ∞ is represented, where ∞ is the result of dividing a normal number by zero. The sign bit is still used in these representations, so there are representations for ± 0 and $\pm \infty$.
 - When $E' = 255$ and $M \neq 0$, the value represented is called *Not a Number* (NaN). A NaN represents the result of performing an invalid operation such as $0/0$ or $\text{sqrt}(-1)$.

FP Addition/Subtraction

- Add/Subtract procedure:
 1. **Shift mantissa** of number with smaller exponent to the right
 2. **Set exponent** of result to larger exponent
 3. Perform **addition/subtraction** of mantissas and **set sign** of result
 4. **Normalize** the result, if necessary

FP Addition example

- Perform $C = A + B$ for
 $A = 0 \ 10000101 \ 0110\dots$
 $B = 0 \ 10000011 \ 1010\dots$
 1. Shift mantissa of B two places to right
 2. Set exponent of C to 10000101
 3. Add mantissas
$$\begin{array}{r} 1.011000\dots \\ + 0.011010\dots \\ \hline 1.110010\dots \end{array}$$
 4. $C = 0 \ 10000101 \ 110010\dots$

FP Multiplication

- Multiply procedure:
 1. Add exponents and subtract 127
(to maintain excess-127 representation)
 2. Multiply mantissas, determine sign of result
 3. Normalize result, if necessary

FP Division

- Divide procedure:
 1. Subtract exponents and add 127
(to maintain excess-127 representation)
 2. Divide mantissas, determine sign of result
 3. Normalize result, if necessary

Truncation of FP mantissas

- The mantissa resulting from an arithmetic operation on two floating-point numbers may be longer than 24 bits
- It must be **truncated** to 24 bits (for 32-bit FP)
- The IEEE standard requires that **rounding** to the nearest 24-bit value is the truncation method to be used

Rounding an FP mantissa

- Consider examples of rounding an 8-bit mantissa to a 5-bit length to illustrate the rounding operation:

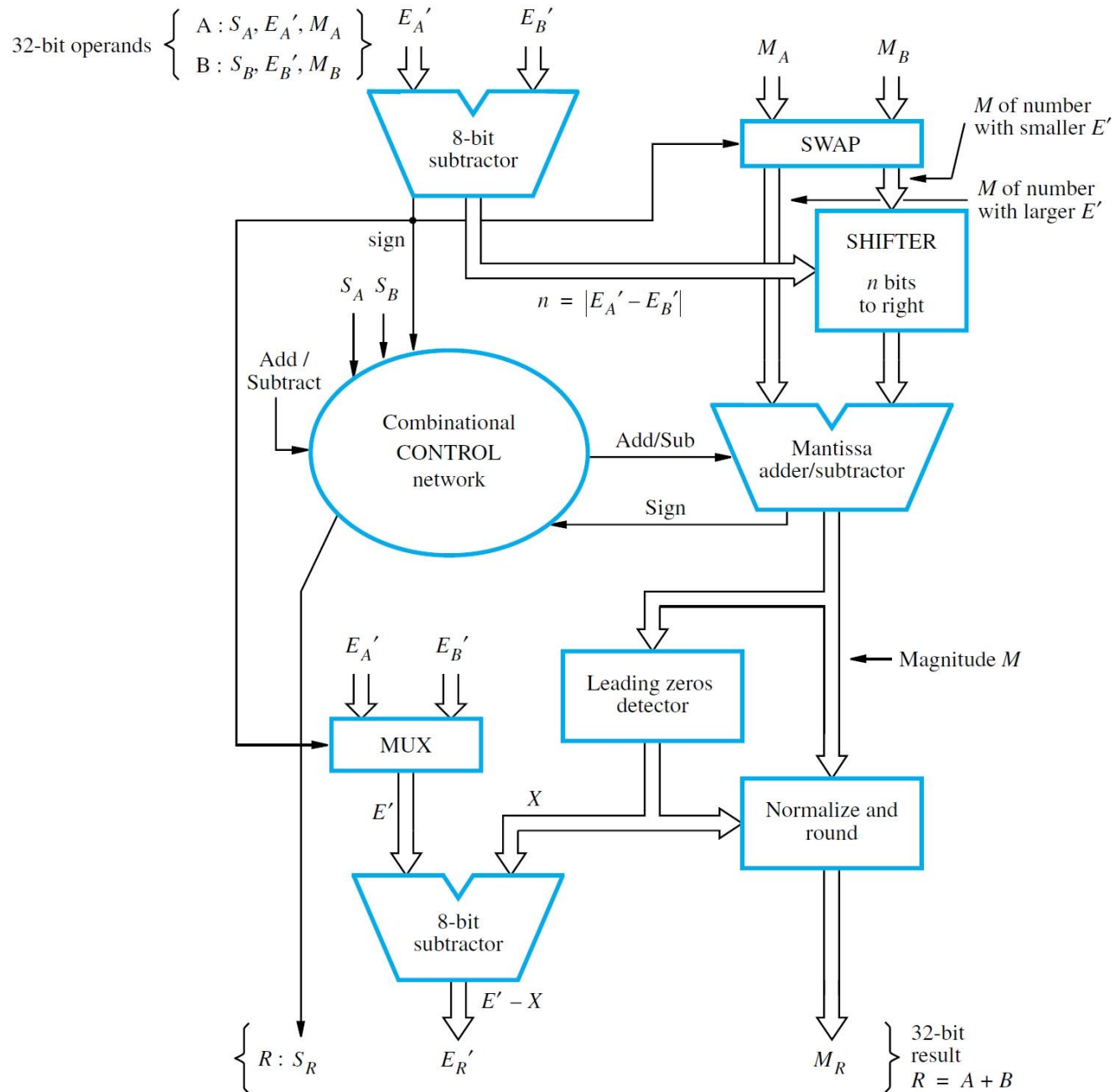
Ex. 1: Round 1.1011011
 Result = 1.1011

Ex. 2: Round 1.1011110
 1.1011
 + 0.0001

 Result = 1.1100

Implementation of FP operations

- A considerable amount of logic circuitry is needed to implement floating-point operations in hardware, especially if high performance is needed
- It is also possible to implement floating-point operations in software
- A hardware addition/subtraction unit is shown in the next figure



Floating-point addition-subtraction unit.

Sections to Read (From Hamacher's Book)

- Chapter on Arithmetic
 - All sections and sub-sections EXCEPT
 - Summand Addition Tree using 4-2 Reducers
 - Guard Bits and Truncation
 - Decimal to Binary Conversion