ELEC 374

Digital Systems Engineering

Winter 2017

Iman Faraji

Course web site

https://onq.queensu.ca/d2l/le/calendar/95107

Course Administration

- Instructor: Iman Faraji (WLH-706)

  Email: <i.faraji@queensu.ca>
  Office hours: WLH-706 Thursdays 5:30 pm to 6:30 pm

- TAs: Chaoyu (Patrick) Wu < 14cw16@queensu.ca> Office Hours: WLH-706 Thursdays 3:00 pm to 4:00 pm

  Priya Balasubramaniam < 16rb25@queensu.ca> Office Hours: WLH-706 Wednesday 11:00 am to 12 pm

  Nicholas Petrelli <13np8@queensu.ca>

  Chris Cartwright <0cc22@queensu.ca>

  Ryan Dick <12rjd6@queensu.ca>

- Lectures – Section 001 (Miller 210): Mondays 2:30 - 3:30pm, Tuesday 4:30 - 5:30pm, and Thursdays 3:30 - 4:30pm

- Lectures – Section 004 (Jeffery 225): Mondays 1:30 - 2:30pm, Wednesdays 12:30 - 1:30pm, and Thursdays 10:30 - 11:30pm

- Tutorials (Walter Light 210): Wednesdays 3:30 - 4:30pm - there will be 6 tutorials – 4 quizzes will be held in the tutorials

- Labs for Section 001 (BMH-314): Monday 8:30am – 11:30 am, starting week 3

  Students that are enrolled in Section 001 can only participate this lab*

- Labs for Section 004 (BMH-214): Tuesday 6:30pm – 9:30 pm, starting week 3

  Students that are enrolled in Section 003 can only participate this lab*

*Conditions may apply

- Problem Sets: 6 sets; no marking
- Quizzes: 4 quizzes, tentatively set for week 4, 7/8, 10/11 and 12
- Prerequisites: ELEC-271, ELEC-274 and ELEC-252 or with instructor permission

Course Administration (Cont'd)

- Textbook: Course Reader (required) available at campus bookstore.
- Lecture slides: will be available on the course website as the term progresses. You should use them as an aid to note taking in class and for studying, but you should not see them as a substitute for attending the lectures. Additional course material and examples will be covered in the lectures, therefore attendance and active class participation is highly recommended.
- Grading:
  - Labs 25% plus up to 5% bonus
  - Machine Problems: 10%
  - Quizzes 20% (or 10%) - 4 quizzes, 5% (or 2.5%) each
  - Final Exam 45% (or 55%)

  Quizzes and final exam together is worth 65% of the course grade: quizzes 20% and final 45%, or quizzes 10% and final 55%, whichever yields the better result.

Course Outline

- High-performance logic design for arithmetic circuits
- Hardware description languages (VHDL, Verilog)
- GPU architectures and computing
- Fault testing, design for testability, built-in self-test, memory testing, and boundary-scan architecture
- Asynchronous digital systems design
- Static, dynamic, and read-mostly memory system design
- Computer bus protocols and standard I/O interfaces (PCI, PCIe, QPI, HyperTransport, InfiniBand, NVLink, USB, etc.)
- Mass storage technologies

- The course is supplemented by a term-length CPU design project that allows students to become proficient with Field Programmable Gate Array (FPGA) devices and associated CAD tools. Students will also work with a 4-node GPU cluster, consisting of NVIDIA Tesla C2075 cards, for their GPU computing.

ELEC 374

Digital Systems Engineering

Winter 2017

Computer Arithmetic

Outline

- **Addition/Subtraction** (HA, FA, RCA, serial, CLA, Carry-Select, memory-based)
- **Multiplication**
  - **of positive numbers** (array multipliers, sequential multipliers)
  - **of Signed-operand** (Booth algorithm)
  - **Fast multiplication**
    - Bit-pair recoding of multipliers, Carry Save addition of summands
    - Other methods: Table lookup, memory-based
- **Division**
  - **Integer Division** (Restoring and non-restoring algorithms, array division)
  - **Division by Convergence**
- **Floating-point numbers and operations**
  - Errors in floating-point representation
  - IEEE-754 floating-point standard
  - Implementing floating-point operations

References: Hamacher et al., Cavanagh (Part I); as well as Murdocca, and Heuring (ch. 2), and Parhami

Addition/Subtraction

- Half-adder (binary addition)

$$\begin{array}{r} x \\ +\,y \\ \hline c\ s \end{array} \qquad \begin{array}{r} 0 \\ +\,0 \\ \hline 0\ 0 \end{array} \qquad \begin{array}{r} 0 \\ +\,1 \\ \hline 0\ 1 \end{array} \qquad \begin{array}{r} 1 \\ +\,0 \\ \hline 0\ 1 \end{array} \qquad \begin{array}{r} 1 \\ +\,1 \\ \hline 1\ 0 \end{array}$$

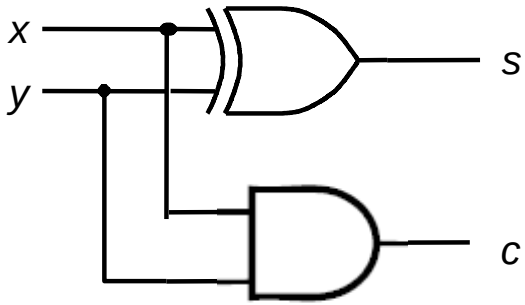
Carry Sum

Four possible cases

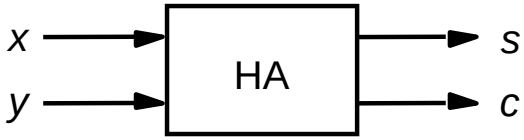
x	y	Carry c	Sum s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$c = xy$$

$$s = x \oplus y$$



Half-adder Circuit

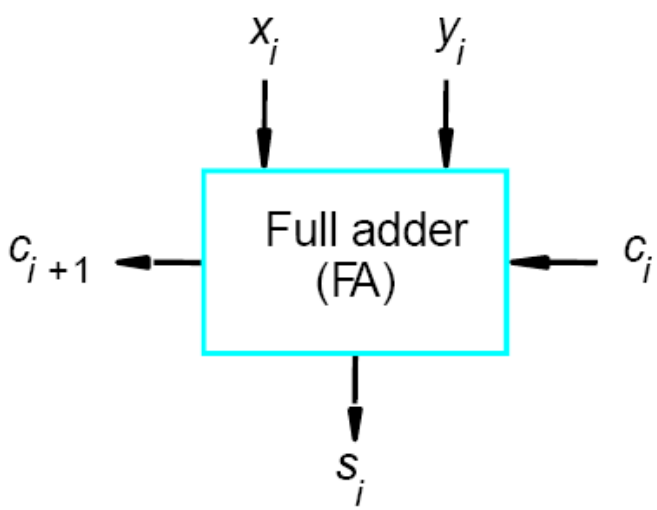


Graphical symbol

Addition/Subtraction (Cont'd)

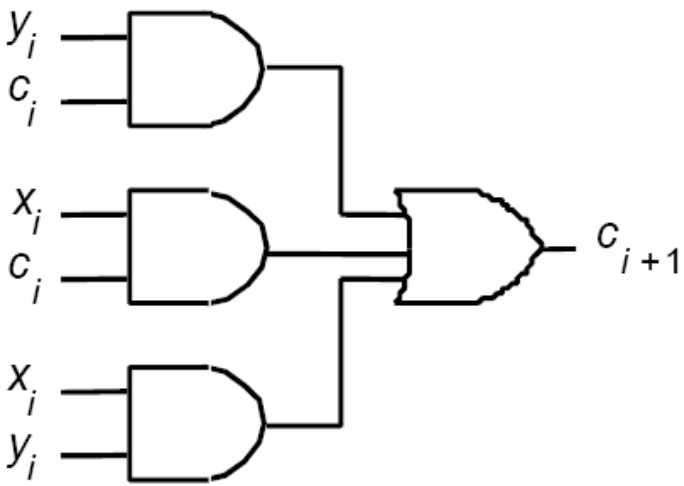
° Full-adder

c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



c_i \ x_iy_i	00	01	11	10
0			1	
1		1	1	1

$$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$$



Addition/Subtraction (Cont'd)

- Full-adder (cont'd)

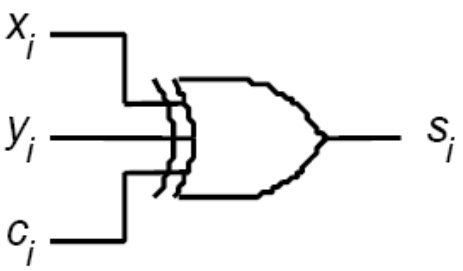
c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

c_i \ x_iy_i	00	01	11	10
0		1		1
1	1		1	

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

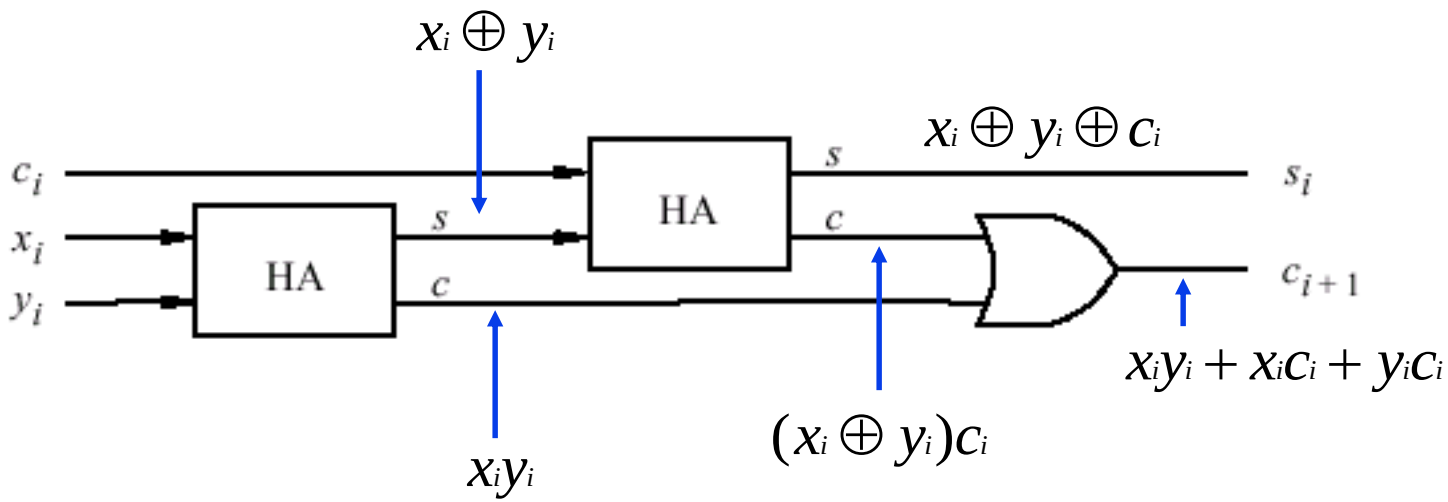
$$s_i = \overline{x}_i\overline{y}_i c_i + \overline{x}_i y_i \overline{c}_i + x_i y_i c_i + x_i \overline{y}_i \overline{c}_i$$

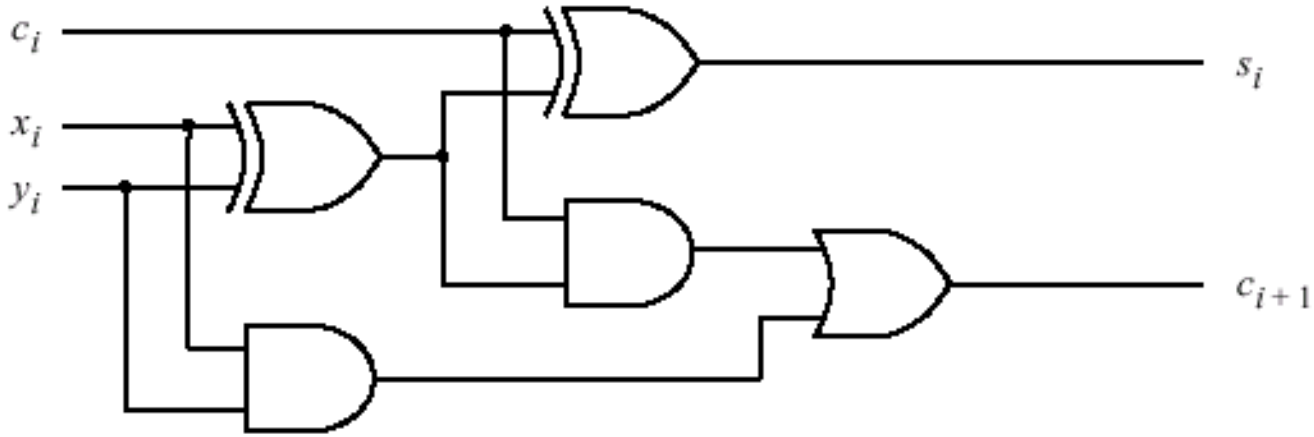
Proof:



Addition/Subtraction (Cont'd)

- Decomposed implementation of a full-adder circuit





Addition/Subtraction (Cont'd)

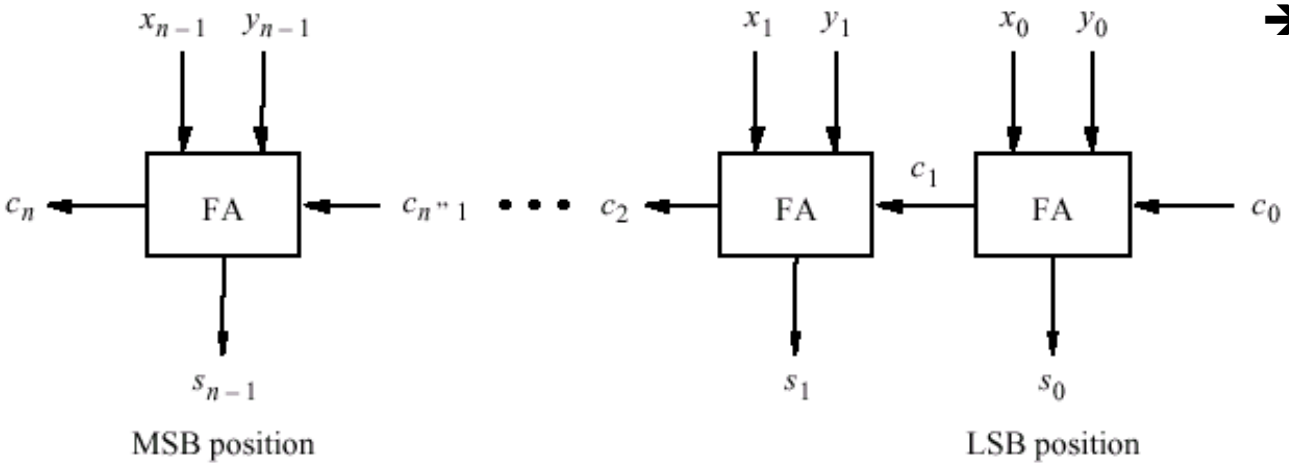
- Decomposed implementation of a full-adder circuit (cont'd)

Proof:

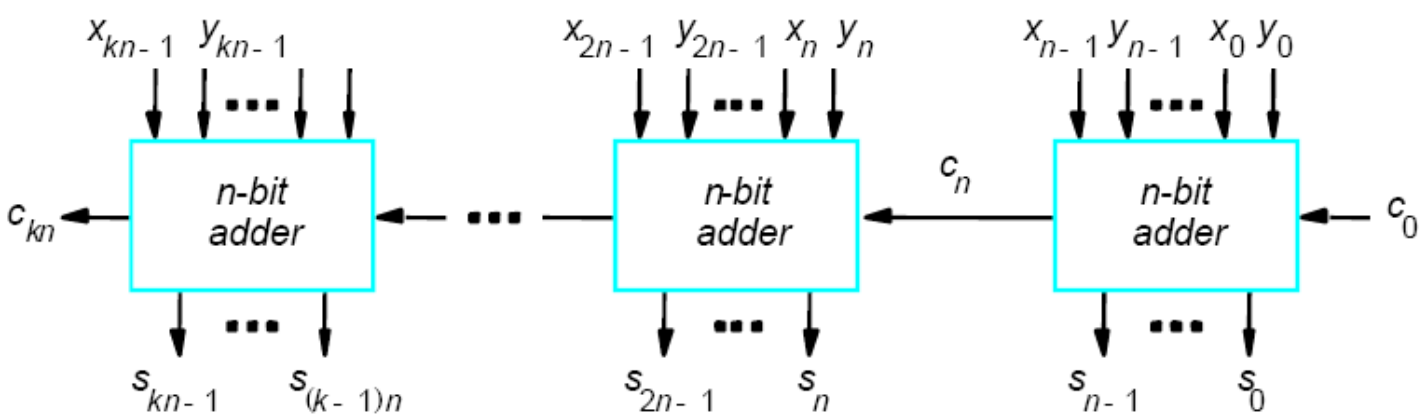
Addition/Subtraction (Cont'd)

- An *n*-bit Ripple-Carry Adder (RCA)

Problem: long delay
Let $delay_{FA}$ = 10ns
➔ a 32-bit addition takes 320ns



- Cascade of *k* *n*-bit adder



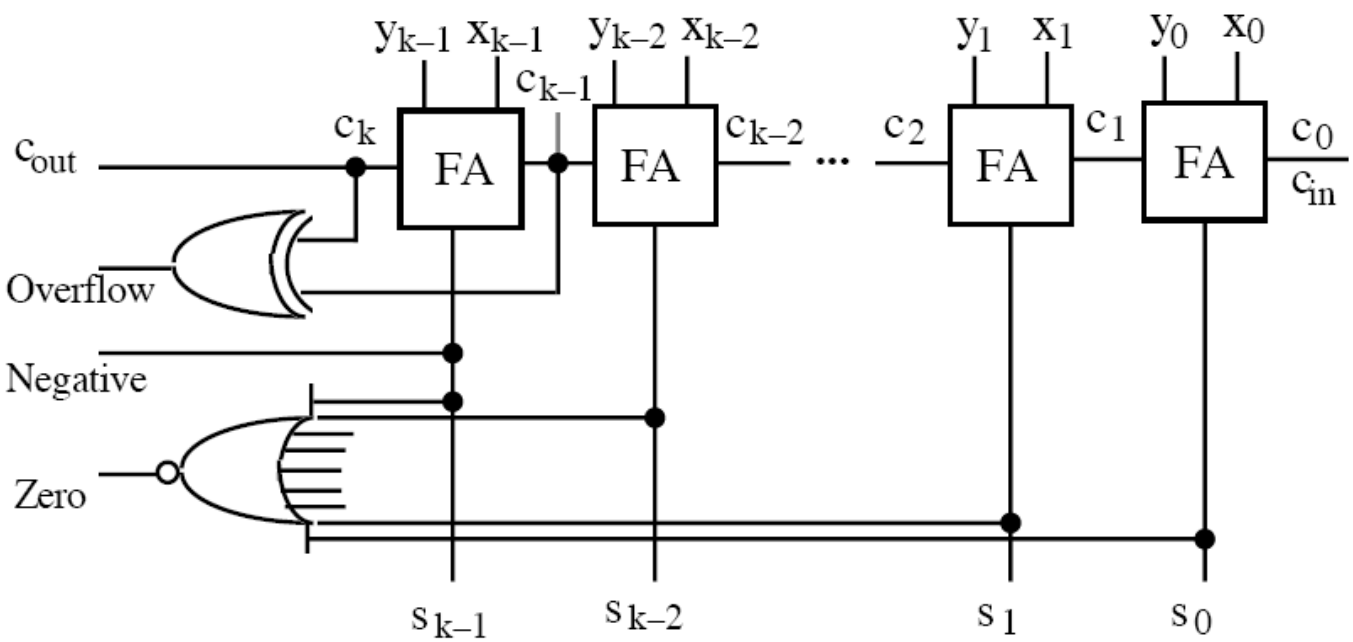
Addition/Subtraction (Cont'd)

- 2's complement addition with provision for detecting conditions and exceptions

$$\text{overflow}_{2\text{'s-compl}} = x_{k-1}\, y_{k-1}\, \bar{s}_{k-1} + \bar{x}_{k-1}\, \bar{y}_{k-1}\, s_{k-1}$$

$$\text{overflow}_{2\text{'s-compl}} = c_k \oplus c_{k-1} = c_k\, \bar{c}_{k-1} + \bar{c}_k\, c_{k-1}$$

Why?



Addition/Subtraction (Cont'd)

- 2's complement addition with provision for detecting conditions and exceptions (cont'd)

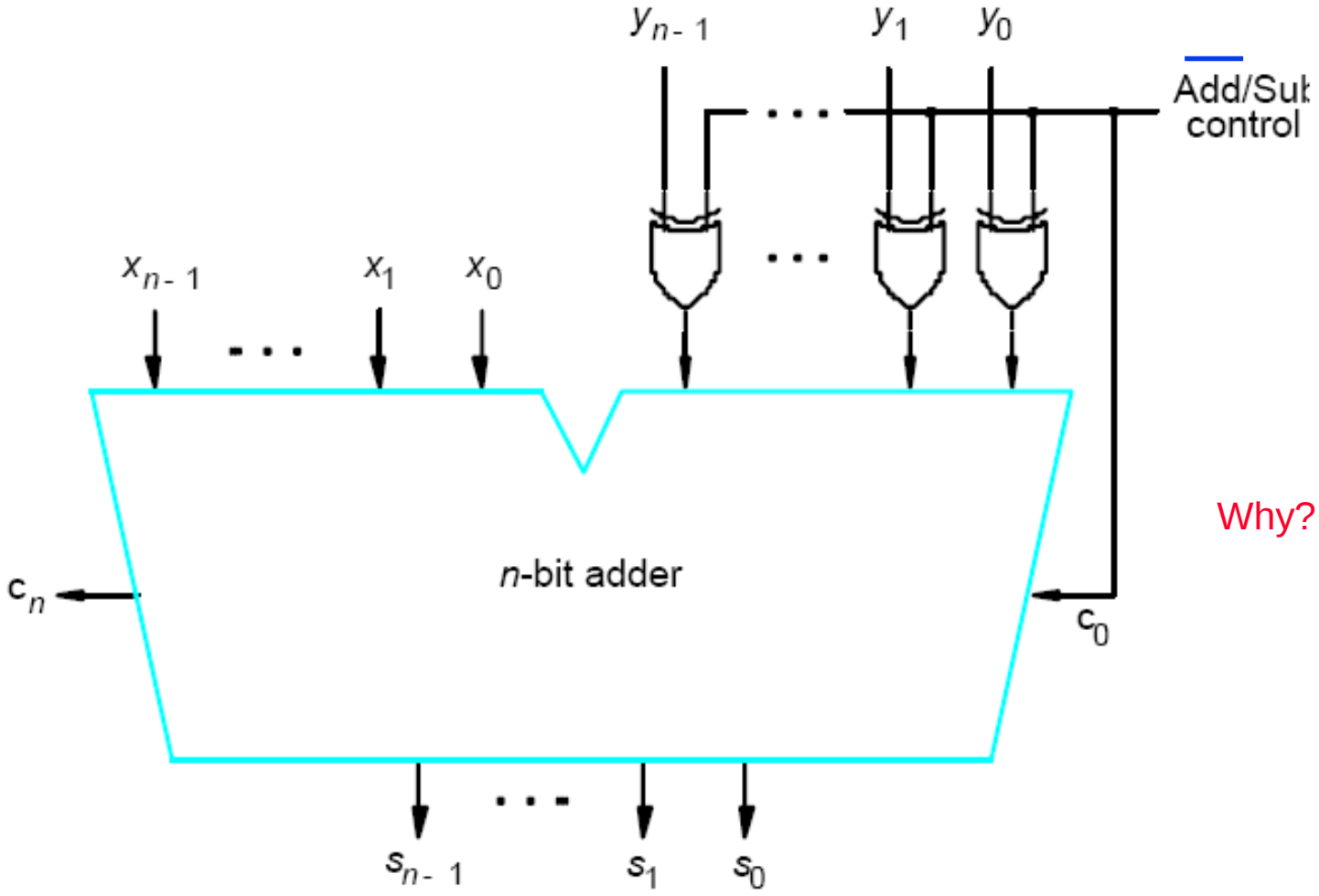
$$\text{overflow}_{2\text{'s-compl}} = x_{k-1}\, y_{k-1}\, \bar{s}_{k-1} + \bar{x}_{k-1}\, \bar{y}_{k-1}\, s_{k-1}$$

$$\text{overflow}_{2\text{'s-compl}} = c_k \oplus c_{k-1} = c_k\, \bar{c}_{k-1} + \bar{c}_k\, c_{k-1}$$

Proof:

Addition/Subtraction (Cont'd)

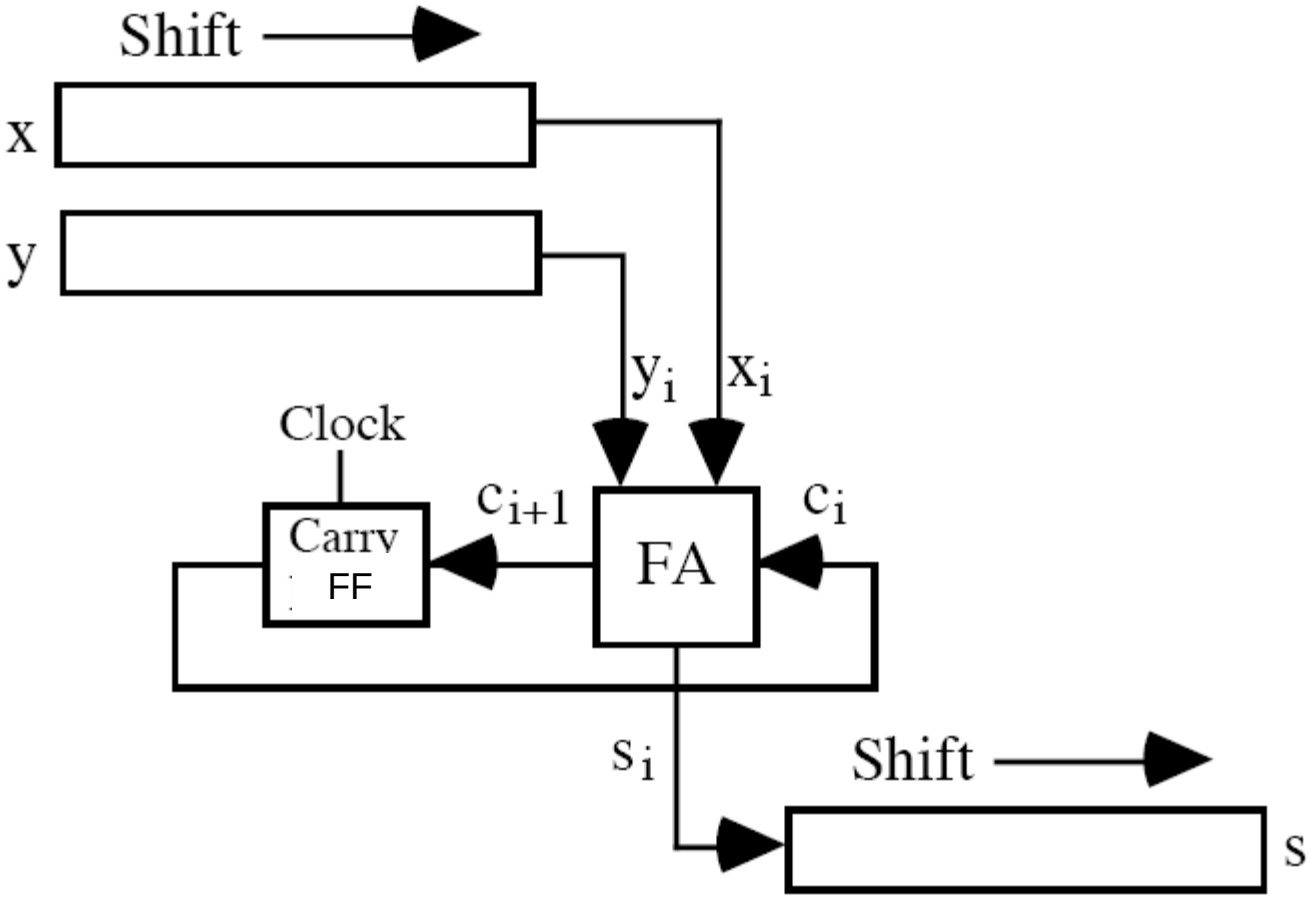
- Adder-subtractor unit



Why?

Addition/Subtraction (Cont'd)

- Bit-serial adder



Addition/Subtraction (Cont'd)

- Fast Adder: Carry-Lookahead Adders (CLA) - we want to evaluate quickly for each stage whether the carry-in from the previous stage in a RCA will have a value of 0 or 1.
- Recall for a full-adder:
  - $S_i = x_i \oplus y_i \oplus c_i$ and $c_{i+1} = x_i y_i + x_i c_i + y_i c_i$
  - We can re-write it as $c_{i+1} = G_i + P_i c_i$

    where $G_i = x_i y_i, P_i = x_i + y_i$
- The function G_i is equal to 1 when both x_i and y_i are equal to 1, regardless of the value of the incoming carry to this stage, c_i. Since in this case, stage i is guaranteed to generate a carry-out C_{i+1}, G_i is called the Generate function.
- The function P_i is equal to 1 when at least one of the inputs x_i and y_i is equal to 1. In this case, a carry-out is produced if $c_i = 1$. The effect is that the carry-in of 1 is propagated through stage i, hence P_i is called the Propagate function.

Addition/Subtraction (Cont'd)

° Carry-lookahead adders (cont'd): $c_{i+1} = G_i + P_i c_i$

- Expanding the expression in terms of stage *i* - 1

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

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1}$$

Continuing so ➔

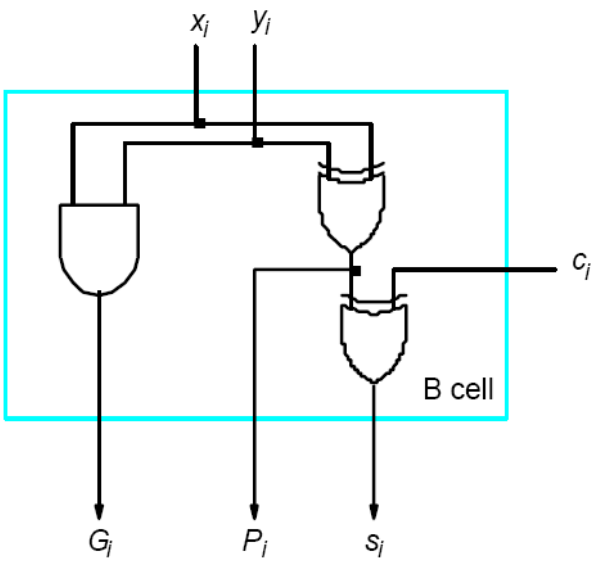
$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \ldots + P_i P_{i-1} \ldots P_1 G_0 + P_i P_{i-1} \ldots P_0 c_0$$

- Carry out delay: 3 gate delay
- Sum delay: 4 gate delay $s_i = x_i \oplus y_i \oplus c_i$
  - when expanded: 6 gate delay; three for C_i, one for inverting carry, and two for AND-OR $s_i = \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i y_i c_i + x_i \bar{y}_i \bar{c}_i$
- Problem: gate fan-in constraint

Addition/Subtraction (Cont'd)

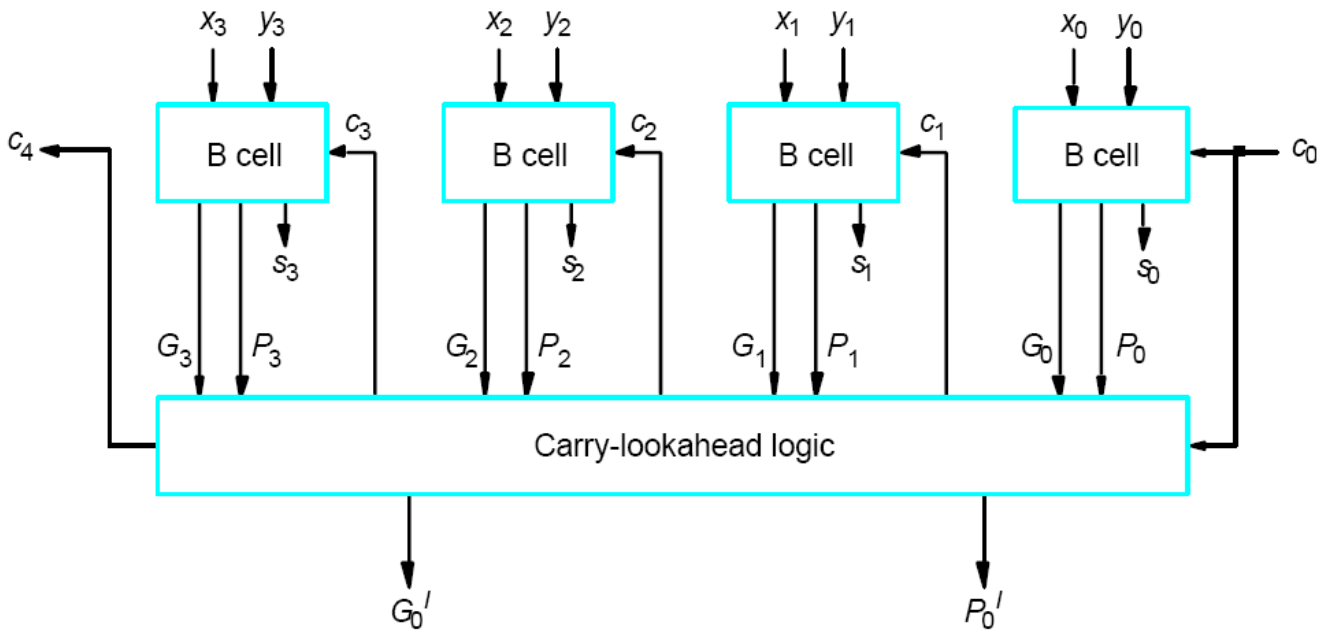
° Bit-stage cell

➔



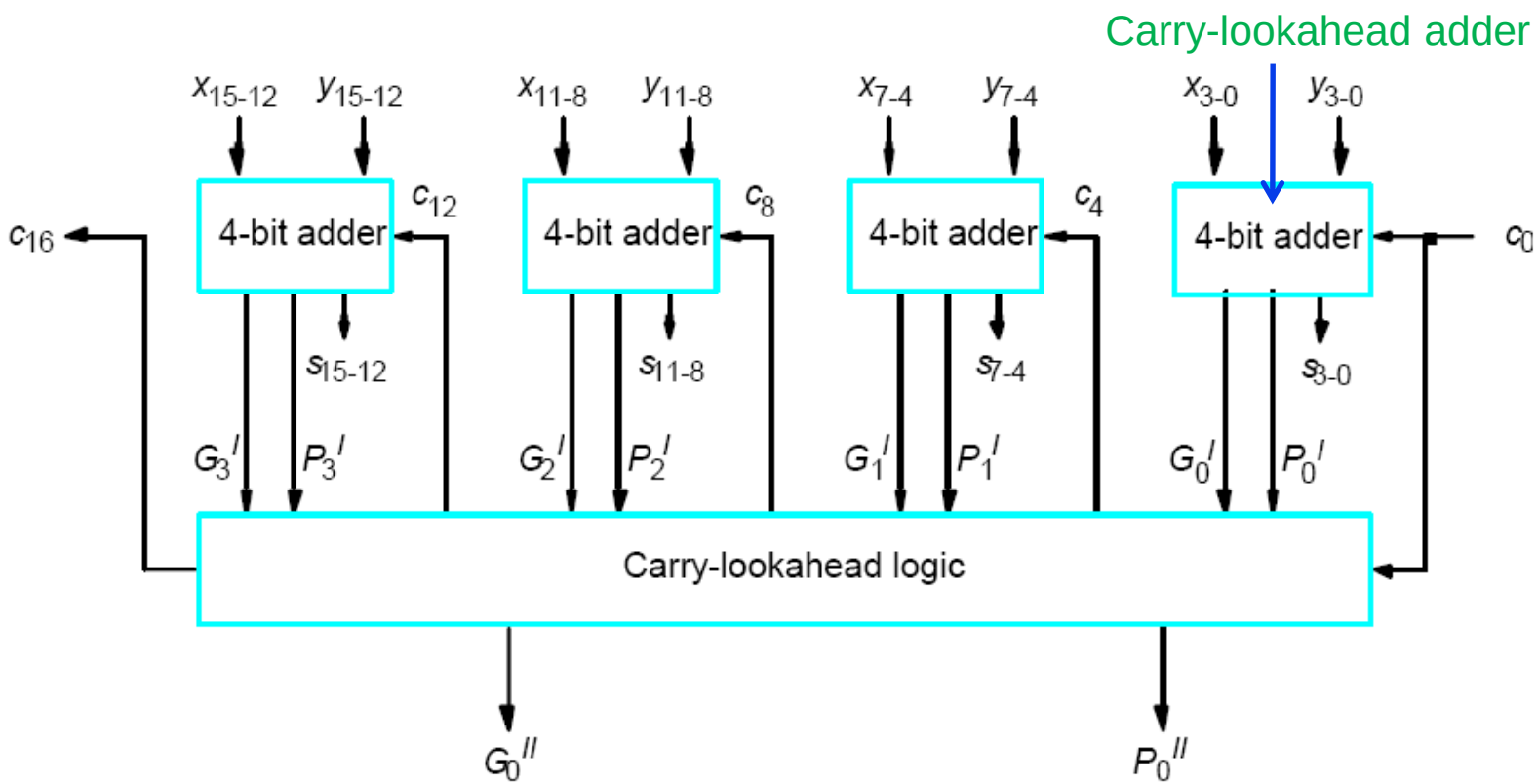
° 4-bit Carry-lookahead adder

➔ Compare this with a 4-bit ripple-carry adder



Addition/Subtraction (Cont'd)

- One can design a hierarchical carry-lookahead adder with ripple-carry between the blocks, to reduce the complexity of designing a large CLA.
- Improved version - 16-bit hierarchical carry-lookahead adder with a second-level carry-lookahead: to produce the carry signals between blocks in parallel.



➔ C_{16} =

Addition/Subtraction (Cont'd)

- Carry-Select Adders: is not as fast as the carry lookahead adder, however it has a unique organization that is interesting.
  - Each section of the adder is partitioned into, for example, two 4-bit groups consisting of identical 4-bit adders (e.g., CLAs) to which the same 4-bit operands are applied.

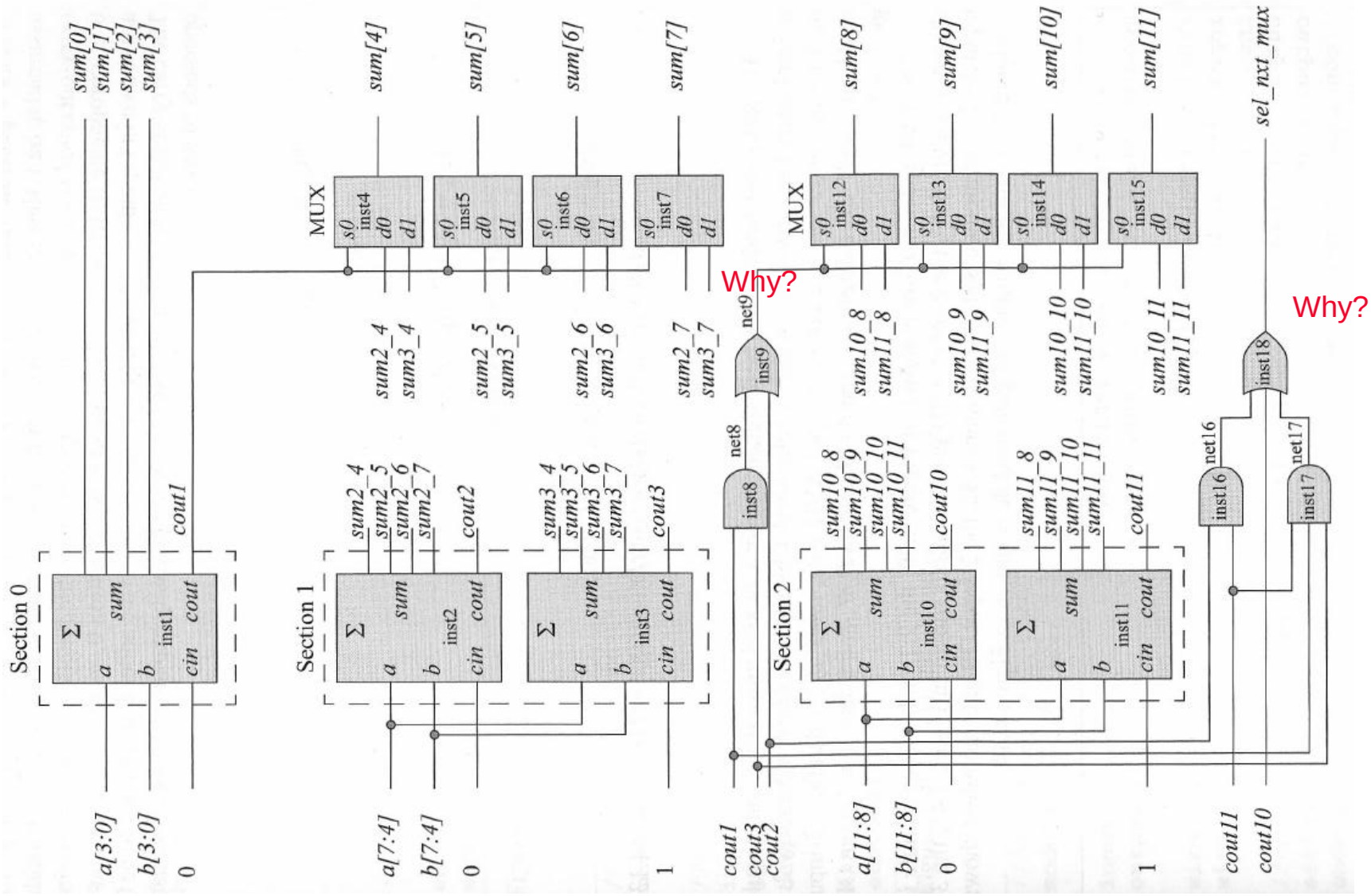
- Since the low-order stage of an adder does not usually have a carry-in, the low-order section consists of only one 4-bit adder with a carry-in of 0.

- The carry-select principle produces two sums that are generated simultaneously. One sum assumes that the carry-in to that group was a 0; the other sum assumes that the carry-in was a 1.

- The logic diagram for a carry-select adder to add two 12-bit operands is shown in the next page.

Addition/Subtraction (Cont'd)

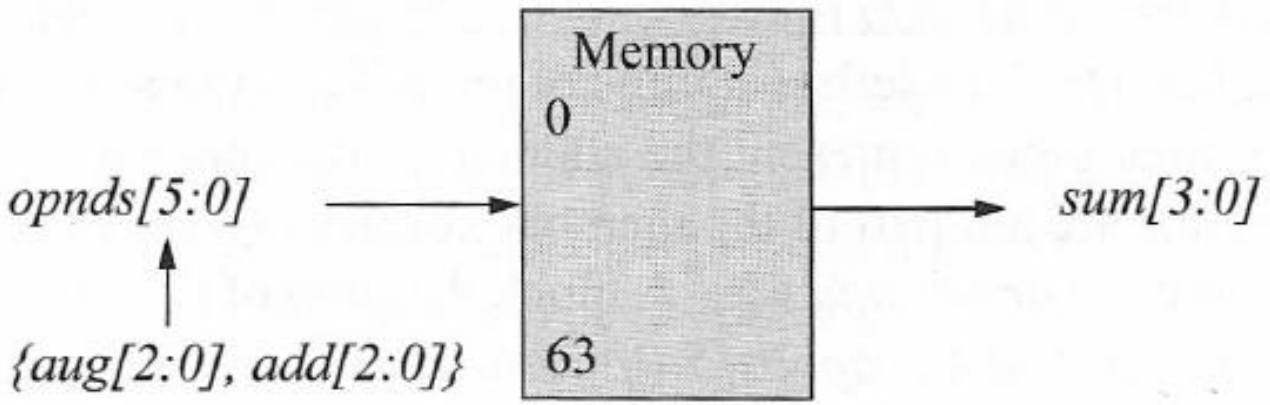
° Carry-Select Adders (cont`d):



Addition/Subtraction (Cont'd)

- Memory-based Adders: With the advent of high-density, high-speed memories, addition can be easily accomplished by applying the augend and addend as address inputs to the memory – the outputs are the sum.

- Block diagram of a memory adding two 3-bit operands:



- The contents of the memory is shown in the next page.

Addition/Subtraction (Cont'd)

- Memory-based Adders (cont`d):

address aug add	sum 3210	address aug add	sum 3210
000 000	**0000**	**100 000**	**0100**
000 001	0001	100 001	0101
000 010	0010	100 010	0110
000 011	0011	100 011	0111
000 100	0100	100 100	1000
000 101	0101	100 101	1001
000 110	0110	100 110	1010
000 111	0111	100 111	1011
001 000	**0001**	**101 000**	**0101**
001 001	0010	101 001	0110
001 010	0011	101 010	0111
001 011	0100	101 011	1000
001 100	0101	101 100	1001
001 101	0110	101 101	1010
001 110	0111	101 110	1011
001 111	1000	101 111	1100
010 000	**0010**	**110 000**	**0110**
010 001	0011	110 001	0111
010 010	0100	110 010	1000
010 011	0101	110 011	1001
010 100	0110	110 100	1010
010 101	0111	110 101	1011
010 110	1000	110 110	1100
010 111	1001	110 111	1101
011 000	**0011**	**111 000**	**0111**
011 001	0100	111 001	1000
011 010	0101	111 010	1001
011 011	0110	111 011	1010
011 100	0111	111 100	1011
011 101	1000	111 101	1100
011 110	1001	111 110	1101
011 111	1010	111 111	1110

Multiplication of Positive Numbers

```
Multiplicand M    (14)          1 1 1 0
Multiplier Q      (11)        × 1 0 1 1
                              ---------
                                1 1 1 0
                              1 1 1 0
                            0 0 0 0
                          1 1 1 0
                          -----------
Product P         (154)   1 0 0 1 1 0 1 0
```

Manual multiplication algorithm

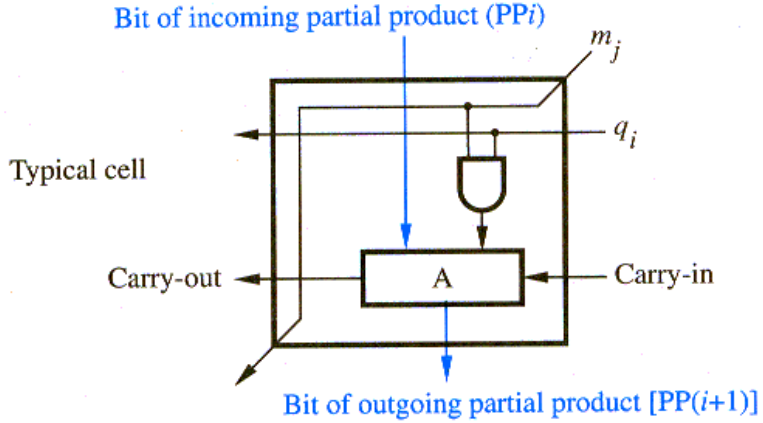
If $q_i = 1$, add the multiplicand (appropriately shifted) to the incoming partial product, PPi, to generate the outgoing partial product, PP(i+1). If $q_i = 0$, PPi is passed vertically downward unchanged.

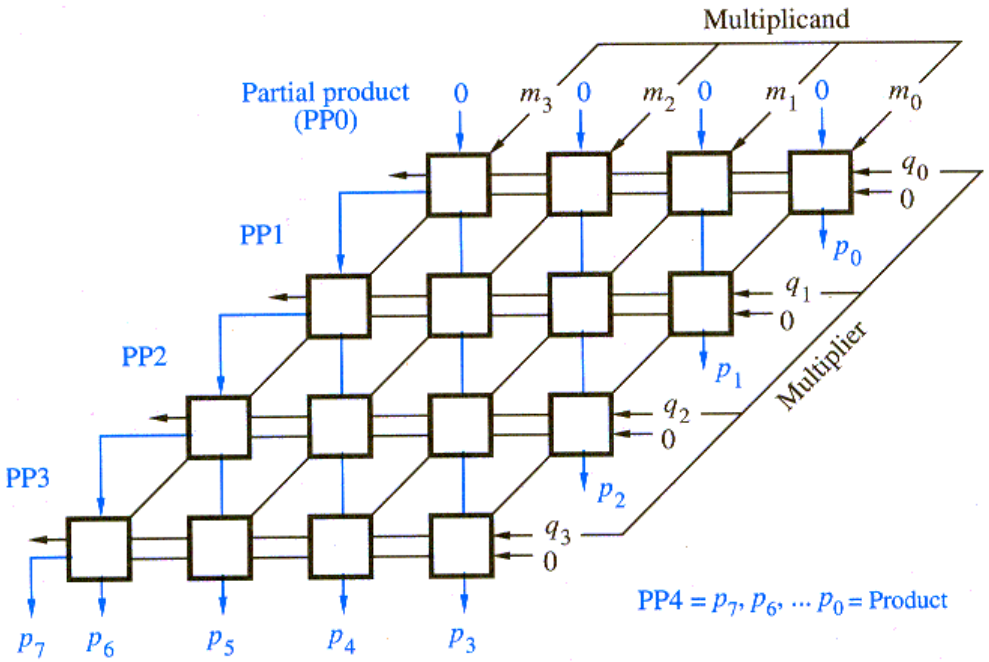
Array multipliers

```
Partial product 0                 0 0 0 0
Multiplicand M    (14)            1 1 1 0
Multiplier Q      (11)          × 1 0 1 1
                                ---------
Partial product 1               1 1 1 0
                              + 1 1 1 0
                              ---------
Partial product 2             1 0 1 0 1
                            + 0 0 0 0
                            ---------
Partial product 3           0 1 0 1 0
                          + 1 1 1 0
                          ---------
Product P         (154)   1 0 0 1 1 0 1 0
```

Combinational array multiplication

Multiplication of Positive Numbers (Cont'd)





Using ripple-carry adder

Array multipliers

Partial product 0		0 0 0 0
Multiplicand M	(14)	1 1 1 0
Multiplier Q	(11)	× 1 0 1 1
Partial product 1		1 1 1 0
		+ 1 1 1 0
Partial product 2		1 0 1 0 1
		+ 0 0 0 0
Partial product 3		0 1 0 1 0
		+ 1 1 1 0
Product P	(154)	1 0 0 1 1 0 1 0

Combinational array multiplication

Multiplication of Positive Numbers (Cont'd)

- Sequential circuit multiplier:

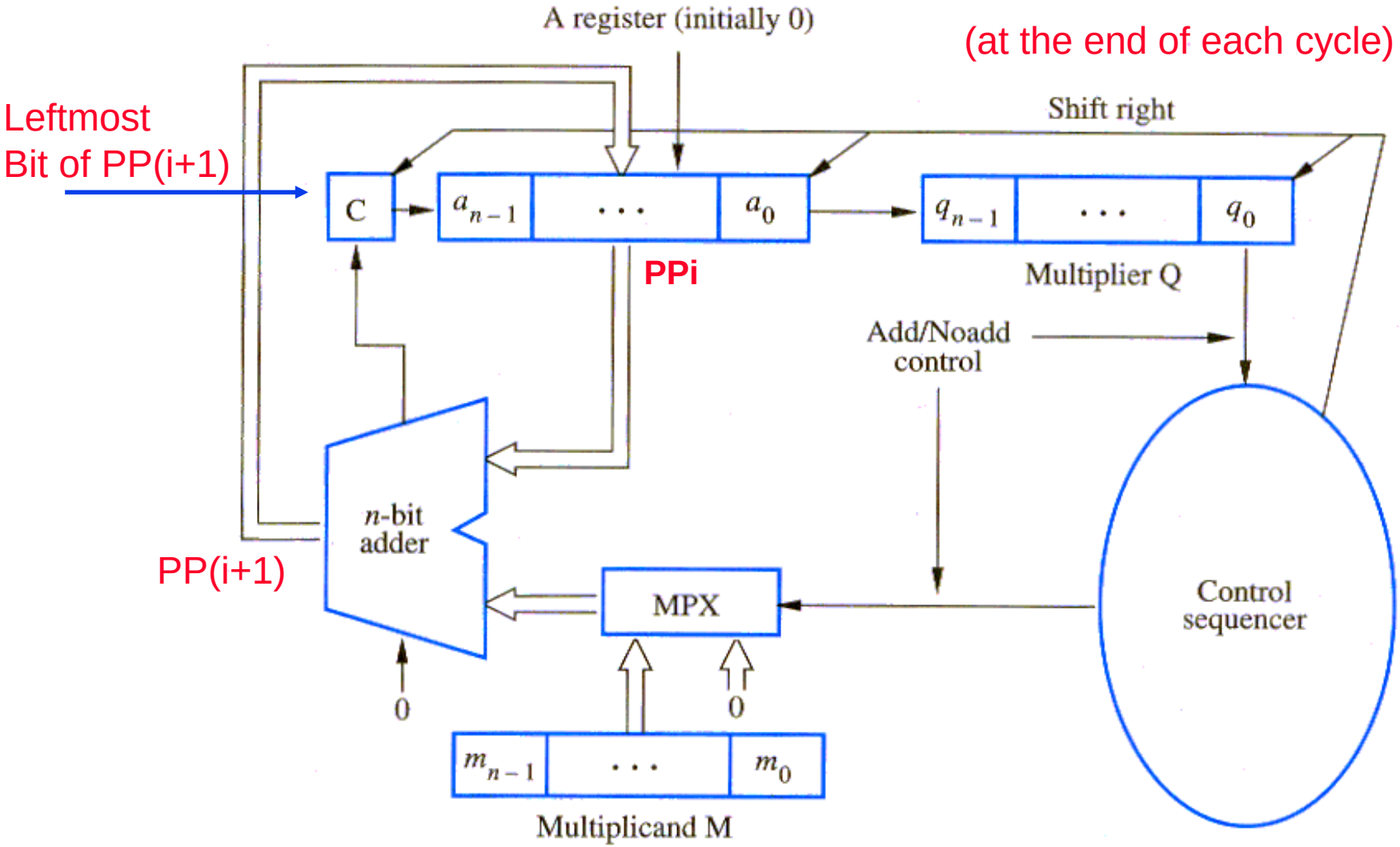
Multiplicand M = 1101 = 13
Multiplier Q = 1011 = 11
Product P = 10001111 = 143

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

Product **P**

Multiplication of Positive Numbers (Cont'd)

- Sequential circuit multiplier implementation: takes a long time (uses the adder in CPU).



Multiplication of Positive Numbers (Cont'd)

- Example

Signed-Operand Multiplication

- Sign extension of negative multiplicand: accumulate partial products by adding versions of the multiplicand as selected by the multiplier bits.
  - Positive multiplier, negative multiplicand
    - Rule: when adding a negative multiplicand to a partial product, extend the sign-bit value to the left as far as the product will extend. The same hardware discussed can be used if sign extension is supported.

						1	0	0	1	1	(−13)	Multiplicand
					×	0	1	0	1	1	(+11)	Multiplier
	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				
	1	1	1	0	0	1	1					
	0	0	0	0	0	0						
	1	1	0	1	1	1	0	0	0	1	(−143)	

  - Negative multiplier, negative multiplicand ➔ 2's complement and do the same algorithm

Signed-Operand Multiplication (Cont'd)

- Example

- What about a negative multiplier and a positive multiplicand?

Signed-Operand Multiplication (Cont'd)

- Booth algorithm: works well for both positive and negative multipliers.
- Consider a positive multiplier with a block of 1s such as 0011110.
  - Instead of adding 4 shifted versions of multiplicand, as in the standard procedure, multiplier can be regarded as: 0 +1 0 0 0 –1 0

```
  0100000   (32)
- 0000010   (2)
---------------
  0011110   (30)
```

➔ less number of operations

Skipping over 1s: one can describe the sequence of required operations as:

0 +1 0 0 0 –1 0

Signed-Operand Multiplication (Cont'd)

- Booth algorithm: with a one-to-one comparison, the Booth multiplier recoding table is

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	$0 \times M$

- Example: Booth recoding of a multiplier

$$0\ \ 0\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 0\ \ 1\ \ 1\ \ 1\ \ 0\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 0$$

$$\Downarrow$$

$$0\ \ +1\ \ -1\ \ +1\ \ 0\ \ -1\ \ 0\ \ +1\ \ 0\ \ 0\ \ -1\ \ +1\ \ -1\ \ +1\ \ 0\ \ -1\ \ 0\ \ 0$$

Signed-Operand Multiplication (Cont'd)

- Booth algorithm: example

(45) Multiplicand
(30) Multiplier

							0	1	0	1	1	0	1
							0	0	+1	+1	+1	+1	0
							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	0	0	1	0	1	0	1	0	0	0	1	1	0

Normal multiplication algorithm

- Note that only a few version of multiplicand (summands) are added in reality. This makes the Booth algorithm fast (in best case).

(45) Multiplicand
(30) Multiplier

							0	1	0	1	1	0	1
							0	+1	0	0	0	−1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	0	1	0	0	1	1	
0	0	0	0	0	0	0	0	0	0	0	0		
0	0	0	0	0	0	0	0	0	0	0			
0	0	0	0	0	0	0	0	0	0				
0	0	0	1	0	1	1	0	1					
0	0	0	0	0	0	0	0						
0	0	0	1	0	1	0	1	0	0	0	1	1	0

2's complement of the multiplicand

Booth multiplication algorithm

Signed-Operand Multiplication (Cont'd)

° Booth algorithm (with a negative multiplier)

$$
\begin{array}{rrrrrrl}
 & 0 & 1 & 1 & 0 & 1 & (+13) \\
\times & 1 & 1 & 0 & 1 & 0 & (-6) \\
\hline
\end{array}
\quad \Longrightarrow \quad
\begin{array}{rrrrrrrrrrl}
 & & & & & 0 & 1 & 1 & 0 & 1 & \\
 & & & & & 0 & -1 & +1 & -1 & 0 & \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & & \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & & & \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
\hline
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & (-78)
\end{array}
$$

Can you revise the circuit on slide 28 for the Booth algorithm?

Signed-Operand Multiplication (Cont'd)

- Booth algorithm

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

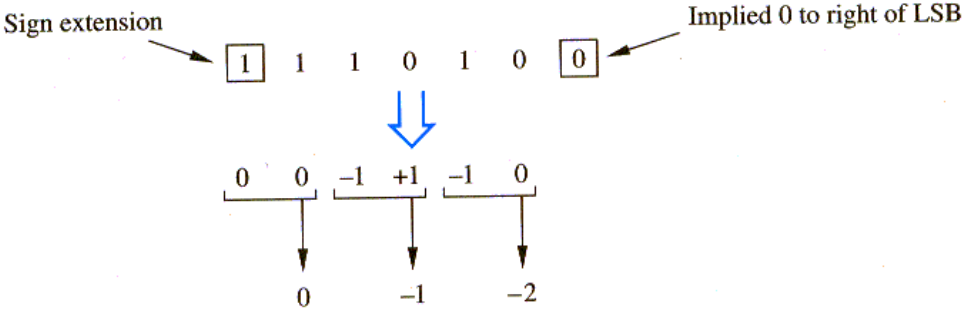
Booth recorded multipliers

Signed-Operand Multiplication (Cont'd)

- Example

Fast Multiplications

1. Bit-Pair Recoding of Multipliers: reduces the number of summands (versions of summands) by half by examining Booth-recoded multiplier two bits at a time.



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

Fast Multiplications (Cont'd)

- Bit-Pair Recoding of Multipliers: example

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

$\Downarrow$

Booth algorithm $\longrightarrow$

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

Multiplication requires only half the number of summands as a normal algorithm.

$\Downarrow$

Bit-pair recoding algorithm $\longrightarrow$

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

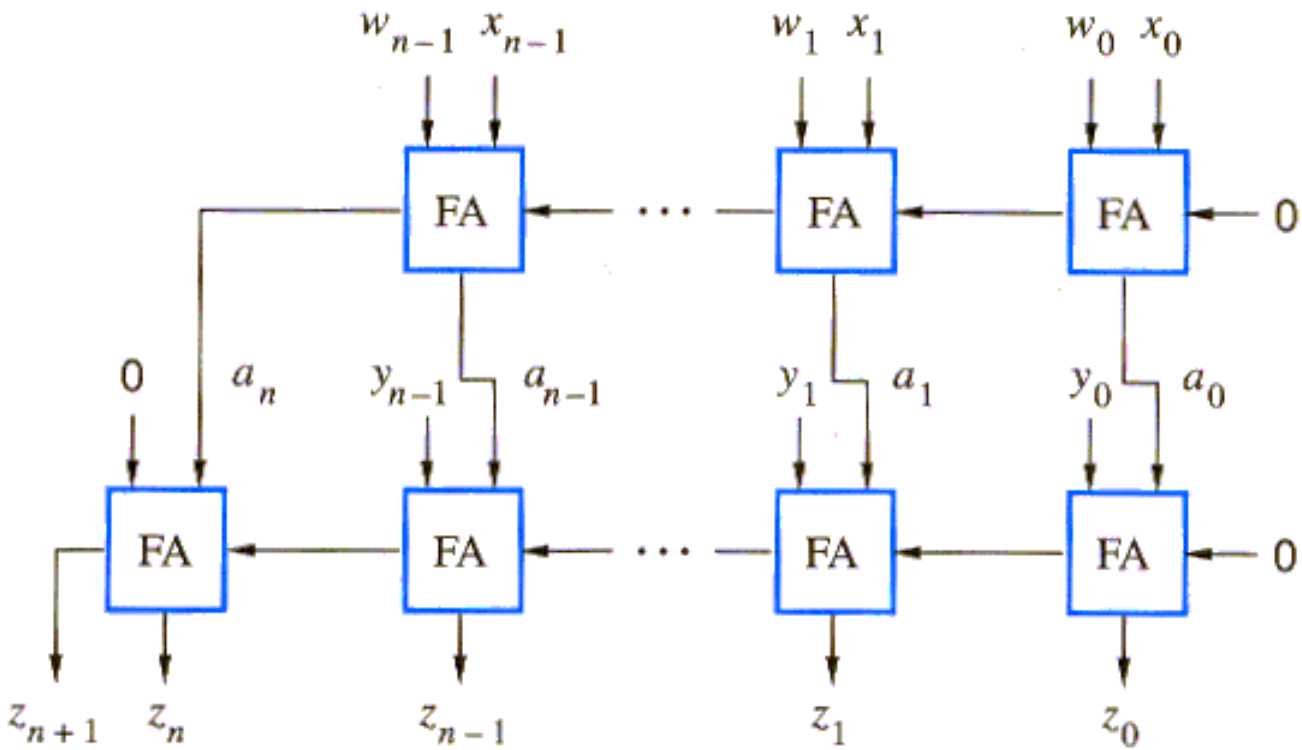
Fast Multiplications (Cont'd)

- Example

Fast Multiplications (Cont'd)

2. Carry-Save addition of summands: multiplication requires the addition of several summands. Carry-save addition speeds up the addition process.

° Example: Z = W + X + Y = (W + X) + Y = A + Y using normal addition



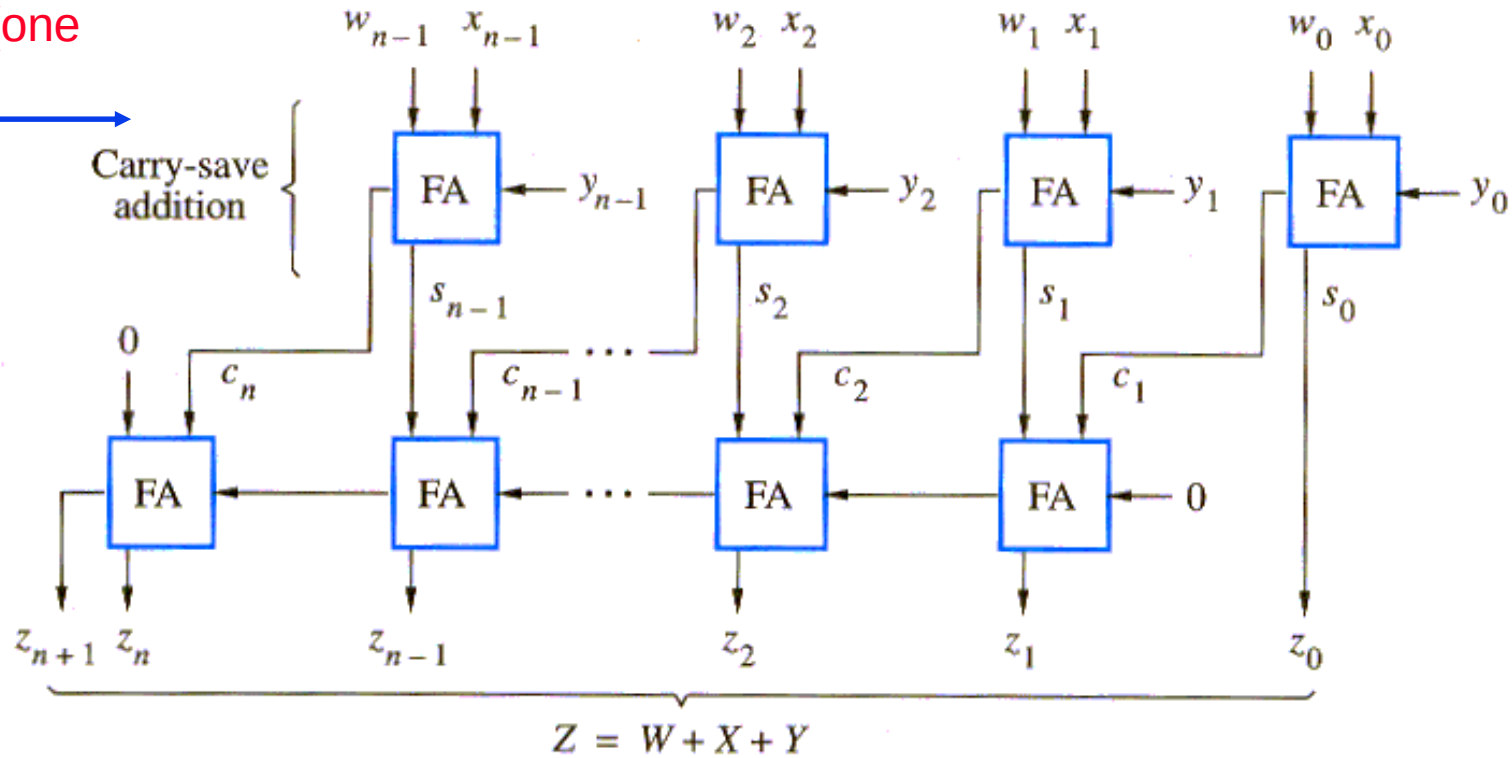
Part (a)

Using two ripple-carry adders

Fast Multiplications (Cont'd)

- Carry-Save addition of Summands: instead of adding W and X to produce A, introduce the bits of Y into the carry inputs. This generates the vector S, and the *saved* (stored) carries, C. Then add S and C using a ripple-carry adder.

Done in a short fixed time (one FA Delay)



Carry-lookahead can be used instead

Part (b)

Fast Multiplications (Cont'd)

- Carry-Save addition of Summands: example

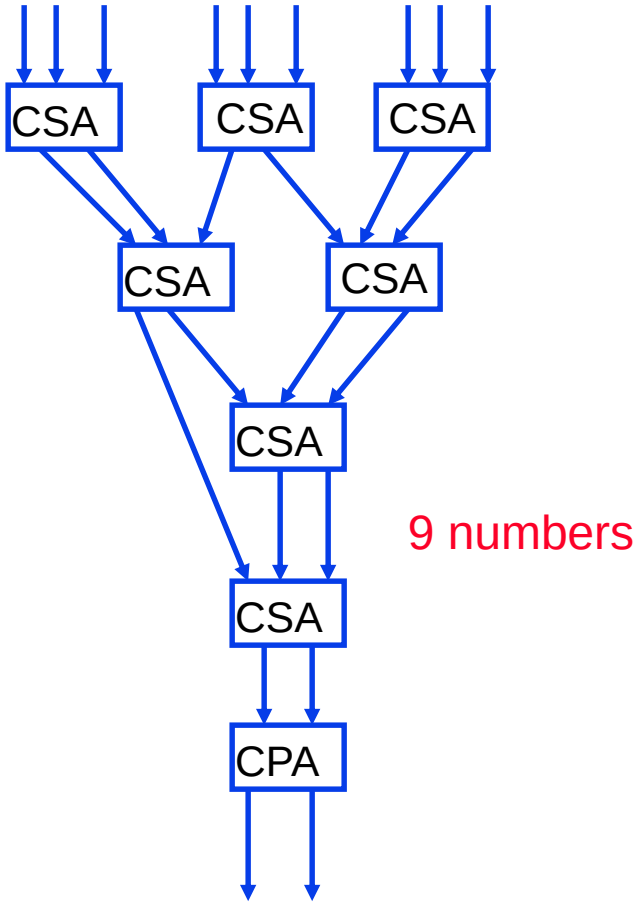
$$
\begin{array}{r}
W \\
+\ X \\
\hline
A \\
+\ Y \\
\hline
Z
\end{array}
\Rightarrow
\begin{array}{r}
10101 \\
+\ 11011 \\
\hline
110000 \\
+\ 010100 \\
\hline
1000100
\end{array}
\qquad
\text{Carry-save addition}
\left\{
\begin{array}{r}
W \\
X \\
+\ Y \\
\hline
S \\
+\ C \\
\hline
Z
\end{array}
\right.
\Rightarrow
\begin{array}{r}
10101 \\
11011 \\
+\ 10100 \\
\hline
11010 \\
+\ 10101 \\
\hline
1000100
\end{array}
$$

Using the network in part (a) Using the network in part (b)

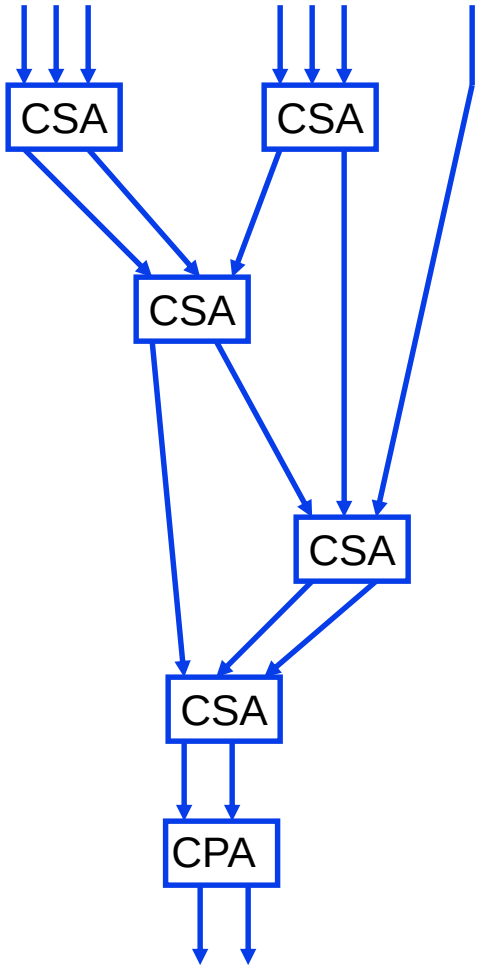
$$Z = W + X + Y$$

Fast Multiplications (Cont'd)

- Carry-Save addition of Summands: Wallace tree



9 numbers



7 numbers

Tree of carry-save adders

Fast Multiplications (Cont'd)

- Example

Fast Multiplications (Cont'd)

- Table Lookup Multiplication: multiplication can be accomplished using a memory that contains different versions of the multiplicand to be added to the partial products. This method is faster than the sequential add-shift algorithm, because it shifts the partial products three bit positions after the add operation.
- Sequential Add-Three-Bit-Shift multiplier: for 8-bit multiplicands and multipliers

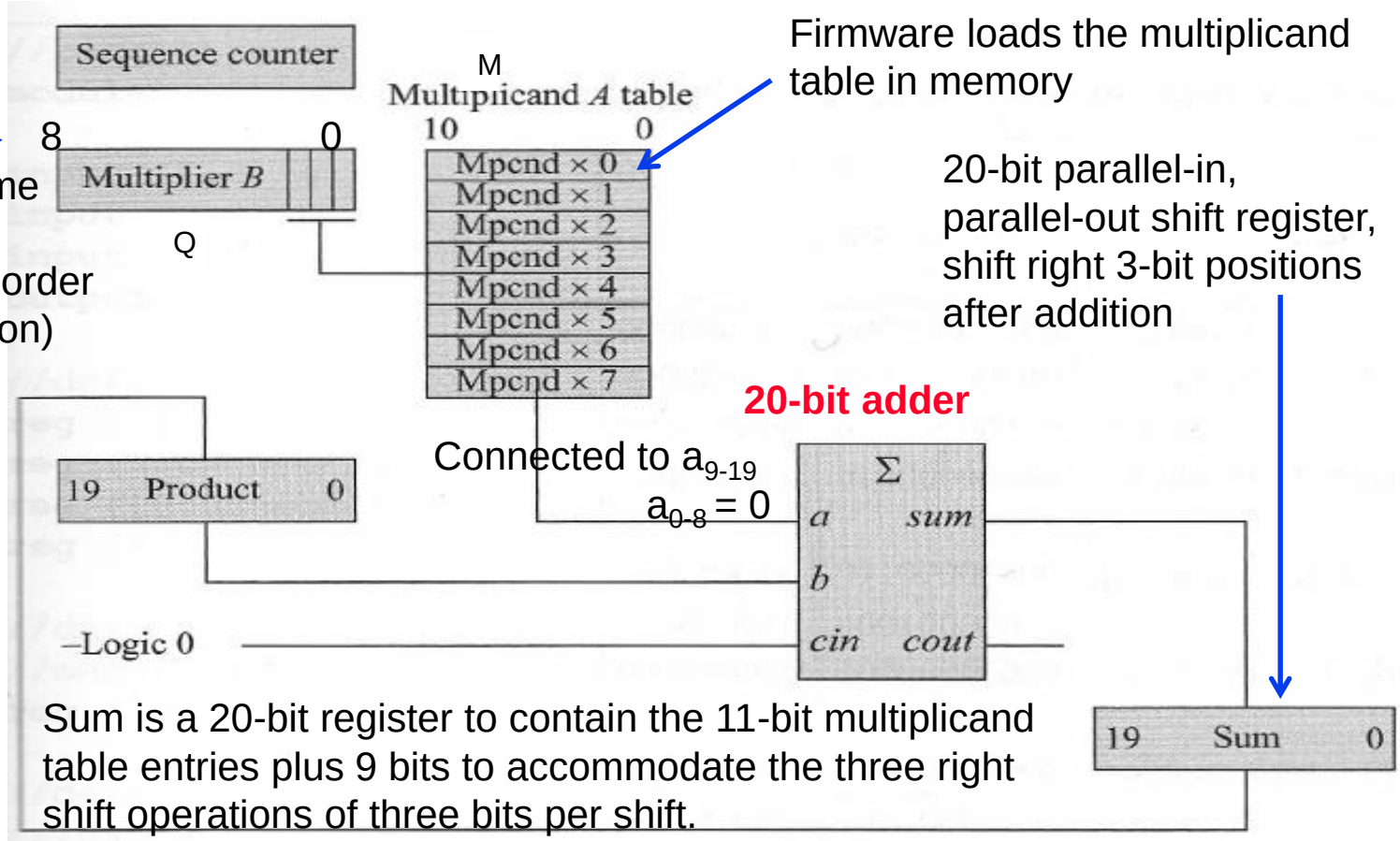
9-bit parallel-in, parallel-out shift register, 3 bits at a time

(divisible by 3 – high-order bit is the sign extension)

Initially,
Sum = Product = 0

Firmware loads the multiplicand table in memory

20-bit parallel-in, parallel-out shift register, shift right 3-bit positions after addition



Sum is a 20-bit register to contain the 11-bit multiplicand table entries plus 9 bits to accommodate the three right shift operations of three bits per shift.

Fast Multiplications (Cont'd)

- Example:

Multiplicand M = 0000 0110 = 6
Multiplier Q = 0 0000 1011 = 11
Product P = 00000000000 001000010 = 66

	19 9 Multiplicand Table	8 0	8 0 Multiplier	
Product =	00000000000	000000000		
+) Multiplicand × 3	00000010010	000000000	000001**011**	
Sum =	00000010010	000000000		
Shift right 3. Product =	**000**00000010	010000000		
+) Multiplicand × 1	00000000110	000000000	000000**001**	**Multiplier shifted right 3 times**
Sum =	00000001000	010000000		
Shift right 3. Product =	**000**00000001	000010000		
+) Multiplicand × 0	00000000000	000000000	000000**000**	**Multiplier shifted right 3 times**
Sum =	00000000001	000010000		
Shift right 3. Product =	**000**00000000	0**01000010**	(+66)	

Product

Fast Multiplications (Cont'd)

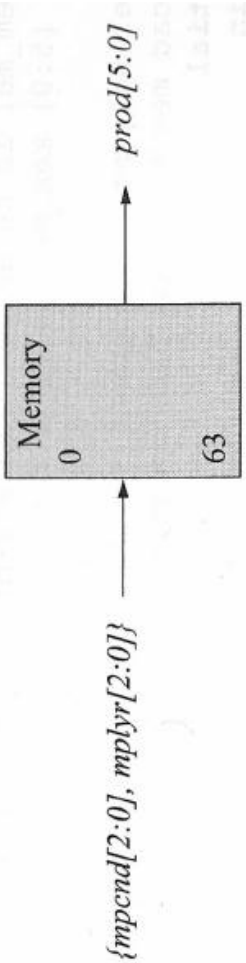
° Example (cont'd): Multiplicand table for a Multiplicand of +6

Multiplicand Versions											
Multiplicand × 0 =	0	0	0	0	0	0	0	0	0	0	0
Multiplicand × 1 =	0	0	0	0	0	0	0	0	1	1	0
Multiplicand × 2 =	0	0	0	0	0	0	0	1	1	0	0
Multiplicand × 3 =	0	0	0	0	0	0	1	0	0	1	0
Multiplicand × 4 =	0	0	0	0	0	0	1	1	0	0	0
Multiplicand × 5 =	0	0	0	0	0	0	1	1	1	1	0
Multiplicand × 6 =	0	0	0	0	0	1	0	0	1	0	0
Multiplicand × 7 =	0	0	0	0	0	1	0	1	0	1	0

Fast Multiplications (Cont'd)

- Memory-based Multiplication: With the advent of high-density, high-speed memories, multiplication using a RAM may be a viable option. The multiplicand and multiplier are used as address inputs to the memory - the outputs are the product.

A	B	prod						
000	**000**	**000000**	**011**	**000**	**000000**	**110**	**000**	**000000**
000	001	000000	011	001	000011	110	001	111110
000	010	000000	011	010	001100	110	010	111100
000	011	000000	011	011	001001	110	011	111010
000	100	000000	011	100	110100	110	100	001000
000	101	000000	011	101	110111	110	101	000110
000	110	000000	011	110	111010	110	110	000100
000	111	000000	011	111	111101	110	111	000010
001	**000**	**000000**	**100**	**000**	**000000**	**111**	**000**	**000000**
001	001	000001	100	001	111100	111	001	111111
001	010	000010	100	010	111000	111	010	111110
001	011	000011	100	011	110100	111	011	111101
001	100	111100	100	100	010000	111	100	000100
001	101	111101	100	101	001100	111	101	000011
001	110	111110	100	110	001000	111	110	000010
001	111	111111	100	111	000100	111	111	000001
010	**000**	**000000**	**101**	**000**	**000000**			
010	001	000010	101	001	111101			
010	010	000100	101	010	111010			
010	011	001100	101	011	110111			
010	100	111000	101	100	001100			
010	101	111010	101	101	001001			
010	110	111100	101	110	001100			
010	111	111110	101	111	000011			



Integer Division

- Positive number division

```
                Quotient
Divisor M        016
                 ____
   8 |  128      Dividend Q
     - 0
     ----
       12
     -  8
     ----
        48
     - 48
     ----
        0
     Remainder
```

Binary Division

```
        q5 to q0
        000111
        ______
110 |   101010
    - 0
    ------
      10
    -  0
    ------
      101
    -   0
    ------
      1010
    -  110
    ------
       1001
     -  110
     -------
       00110
     -   110
     -------
          0
```

$110 > 1 \rightarrow q_5 = 0$

$110 > 10 \rightarrow q_4 = 0$

$110 > 101 \rightarrow q_3 = 0$

$110 < 1010 \rightarrow q_2 = 1$

$110 < 1000 \rightarrow q_1 = 1$

$110 = 110 \rightarrow q_0 = 1$

Example of Division Algorithms:

Restoring Division

Non-Restoring Division

Integer Division (Cont'd)

- Restoring division algorithm:

```
        10
11 ) 1000
     11
      10
```

8/3 = 2 R 2

- Do the following n times:
  - Shift A and Q left one binary position
  - Subtract M from A, and place the result in A
  - If A is negative, set q_0 to 0 and restore A (add M back to A); otherwise, set q_0 to 1.

	A		Q	
Initially	0 0 0 0 0		1 0 0 0	First cycle
	0 0 0 1 1	M		
Shift	0 0 0 0 1		0 0 0 □	
Subtract	1 1 1 0 1	- M		
Set q_0	1 1 1 1 0			
Restore	1 1			
	0 0 0 0 1		0 0 0 0	
Shift	0 0 0 1 0		0 0 0 □	Second cycle
Subtract	1 1 1 0 1	- M		
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 □	Third cycle
Subtract	1 1 1 0 1	- M		
Set q_0	0 0 0 0 1			
Shift	0 0 0 1 0		0 0 0 1	Fourth cycle
Subtract	1 1 1 0 1	- M	0 0 1 □	
Set q_0	1 1 1 1 1			
Restore	1 1			
	0 0 0 1 0		0 0 1 0	
	Remainder		Quotient	

Integer Division (Cont'd)

- Example

Integer Division (Cont'd)

- Non-restoring division algorithm:
  - **Do the following n times (calculate Quotient):**
    - Shift A and Q left one binary position; If A ≥ 0, subtract M from A; otherwise, add M to A.
    - Now, If A ≥ 0, set q_0 to 1; otherwise, set q_0 to 0.
  - **Final Step (Calculate Remainder)**: If A < 0, add M to A (this is to make sure that the positive remainder is in A at the end of n cycles).

Integer Division (Cont'd)

° Non-restoring division algorithm: example

8/3 = 2 R 2

	A		Q	
Initially	0 0 0 0 0	M	1 0 0 0	First cycle
	0 0 0 1 1			
Shift	0 0 0 0 1		0 0 0 □	
Subtract	1 1 1 0 1	- M		
Set q_0	1 1 1 1 0		0 0 0 0	
Shift	1 1 1 0 0		0 0 0 □	Second cycle
Add	0 0 0 1 1	M		
Set q_0	1 1 1 1 1		0 0 0 0	
Shift	1 1 1 1 0		0 0 0 □	Third cycle
Add	0 0 0 1 1	M		
Set q_0	0 0 0 0 1		0 0 0 1	
Shift	0 0 0 1 0		0 0 1 □	Fourth cycle
Subtract	1 1 1 0 1	- M		
Set q_0	1 1 1 1 1		0 0 1 0	
			Quotient	
Add	1 1 1 1 1	M		Restore remainder
	0 0 0 1 1			
	0 0 0 1 0			
	Remainder			

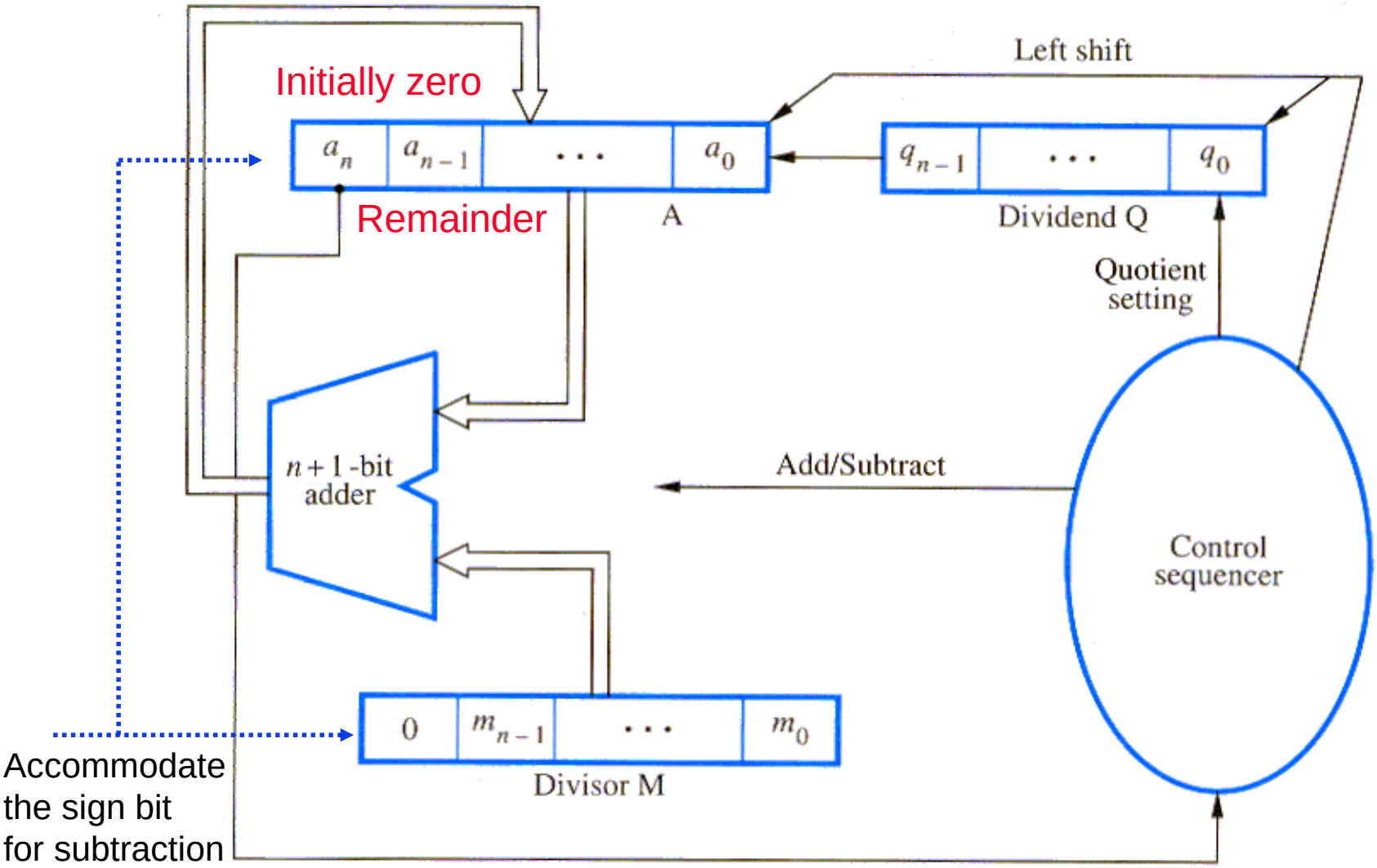
For signed division, transform dividend/divisor into positive numbers, use one of the algorithms above and then change the sign of the result.

Integer Division (Cont'd)

- Example

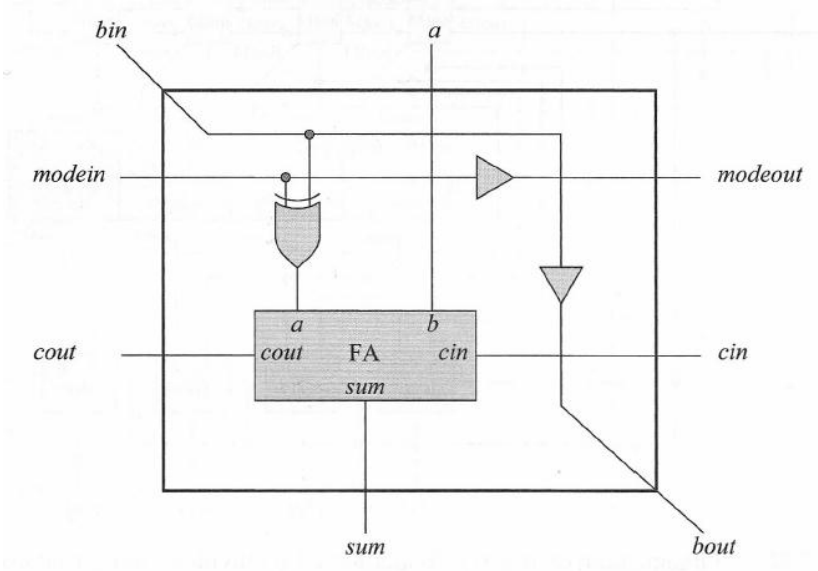
Integer Division (Cont'd)

- Restoring division circuit



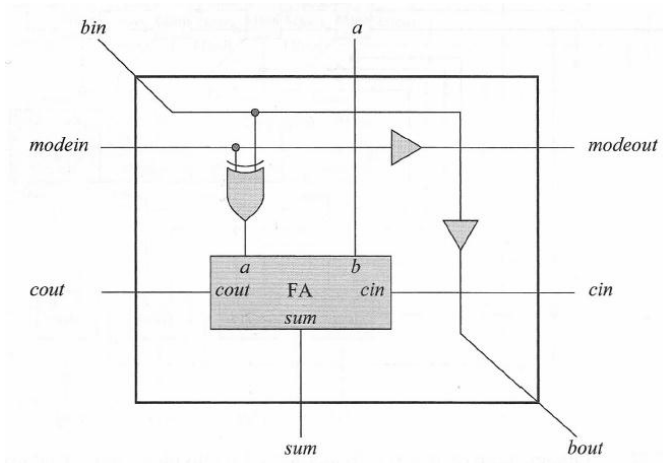
Integer Division (Cont'd)

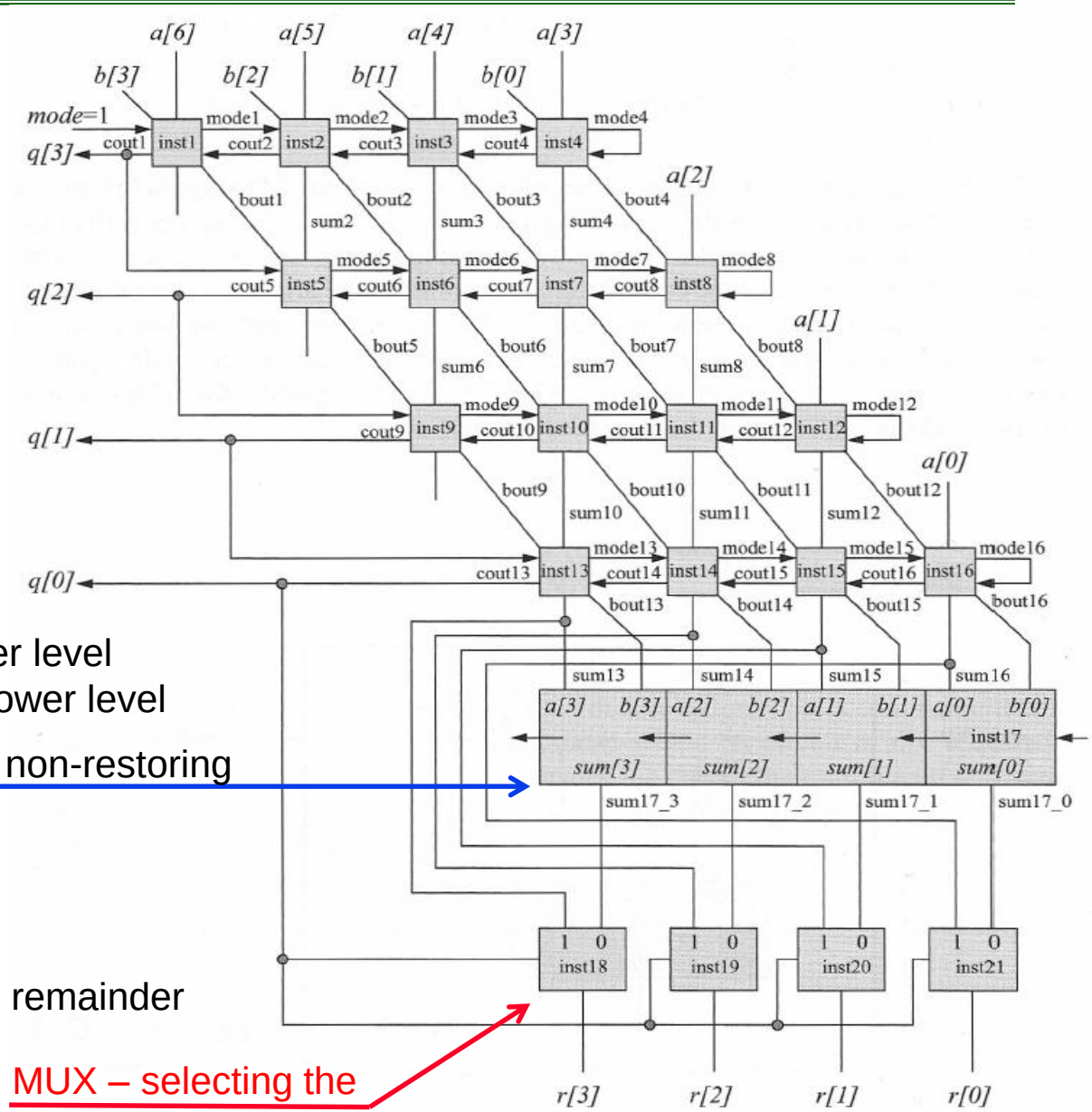
- Array Division (using non-restoring algorithm): extremely fast, and in principle, it is similar to array multiplication. Recall that only two operations are required in the shift-subtract/add non-restoring method: 2A – M and 2A + M.
  - The array consists of rows of identical cells incorporating a full adder in each cell with the ability to add or subtract. Subtraction or addition is determined by the state of the Mode input; if it is a logic 1, then the operation is subtraction; otherwise, it is addition.
  - Shifting is accomplished by the placement of cells along the diagonal of the array. This is done by moving the divisor to the right, thus providing the requisite shift of the divisor relative to the previous partial product.



Integer Division (Cont'd)

- Array Division





C_{out} = 0 ➔ addition at the next lower level
C_{out} = 1 ➔ subtraction at the next lower level

Taking care of the Final step of the non-restoring algorithm, in case it is needed.

Note: in this figure,
Q: dividend/quotient, B: divisor, R: remainder

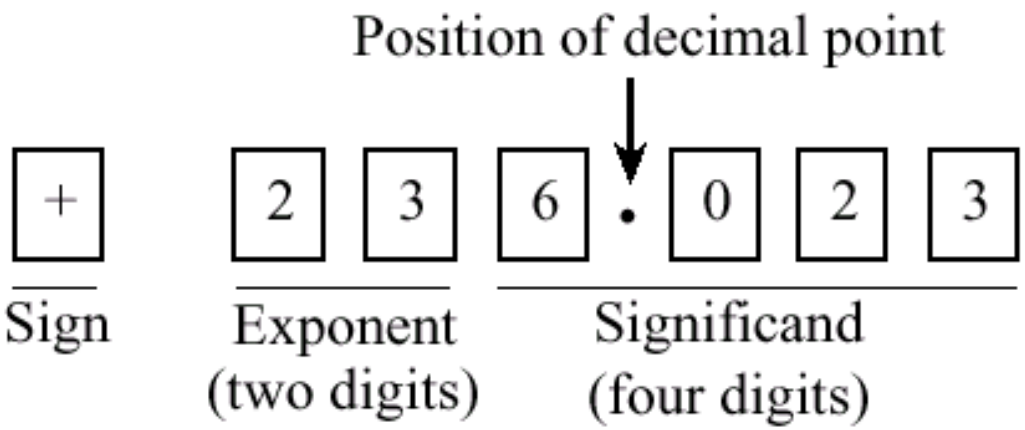
MUX – selecting the right remainder

Floating-Point Numbers

- Review: A fixed-point representation may need a great number of digits to represent a practical range of numbers. Also, a great deal of hardware is needed for manipulations.

- Using only a few digits, floating-point representation can represent very large and very small numbers at the expense of precision.

- Example: $+ 6.023 \times 10^{23}$



Floating-Point Numbers (Cont'd)

- Review: Many different ways to represent a number such as 345.1:
  - 345.1×10^0,
  - 34.51×10^1,
  - 3.451×10^2,
  - 0.3451×10^3,
  - 0.03451×10^4

  This creates a problem when making comparisons.

  ➔ Normalize the number such that the radix point is located in only one possible position. Normally, the radix point is placed immediately to the right of the leftmost, nonzero digit in the fraction, as in 3.451×10^2.

- Hidden bit: in base 2, the leading "1" in the normalized mantissa (or significand) is not stored in memory. This adds an additional bit of precision to the right of the number due to removing the bit on the left.
  - More on that later

Floating-Point Numbers (Cont'd)

- Review: Excess representation
- The idea is to assign the smallest numerical bit pattern, all zeros, to the negative of the bias, and assigning the remaining numbers in sequence as the bit patterns increase in magnitude. Positive and negative representations of a number are obtained by adding a bias (excess value) to the two's complement representation, ignoring any carry out from the most significant digit.

  ➔Numerically smaller numbers have smaller bit patterns, simplifying comparisons for floating-point exponents.

- <u>Example:</u> Excess-127
  - +25 = 10011000_2
  - -25 = 01100110_2
  - One representation for zero: +0 = 01111111_2; –0 = 01111111_2
  - Largest positive number is +128 = 11111111_2
  - Largest negative number is –127 = 00000000_2

Floating-Point Numbers (Cont'd)

° Review: 3-bit Integer Representation

Decimal	Unsigned	Sign-Mag.	1's Comp.	2's Comp.	Excess-4	Excess-3
7	111	-	-	-	-	-
6	110	-	-	-	-	-
5	101	-	-	-	-	-
4	100	-	-	-	-	111
3	011	011	011	011	111	110
2	010	010	010	010	110	101
1	001	001	001	001	101	100
+0	000	000	000	000	100	011
-0	-	100	111	000	100	011
-1	-	101	110	111	011	010
-2	-	110	101	110	010	001
-3	-	111	100	101	001	000
-4	-	-	-	100	000	-

Floating-Point Numbers (Cont'd)

- Review: floating-point example - Represent 254_{10} in a normalized base 8 floating-point format with a sign bit, followed by a 3-bit excess-3 exponent, followed by four base 8 digits.
  - Step 1: Convert to the target base using the remainder method.

    254/8 = 31 R 6; 31/8 = 3 R 7; 3/8 = 0 R 3, thus

    $254_{10} = 376_8 = 376 \times 8^0$

  - Step 2: Normalize the number in the target base.
    - $376 \times 8^0 = 3.76 \times 8^2$
  - Step 3: Fill in the bit fields, with a sign bit = 0 (positive), an exponent of 5 (2 + 3) in excess-3, and 4-digit fraction = 3.760

    0 101 011 . 111 110 000

(spaces are shown for clarity, and base point is not stored in the computer)

Floating-Point Numbers (Cont'd)

- Review: conversion example - Convert $(9.375 \times 10^{-2})_{10}$ to base 2 scientific notation.
- Step 1: Convert from base 10 floating-point to base 10 fixed-point.

  ➔ $(9.375 \times 10^{-2})_{10} = .09375_{10}$
- Step 2: Convert from base 10 fixed-point to base 2 fixed-point using multiplication method.

.09375	×	2	=	0.1875
.1875	×	2	=	0.375
.375	×	2	=	0.75
.75	×	2	=	1.5
.5	×	2	=	1.0

  ➔ Thus, $.09375_{10} = .00011_2$
- Step 3: Convert to normalized base 2 floating-point.

  ➔ $.00011_2 = .00011_2 \times 2^0 = 1.1 \times 2^{-4}$

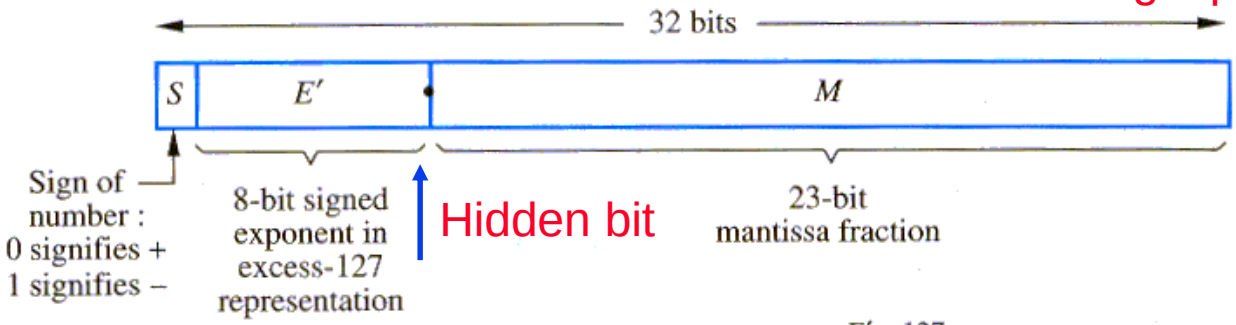
IEEE-754 Floating-Point Standard

- Review - Developed in 1985. It can be supported in hardware, or a mixture of hardware and software.

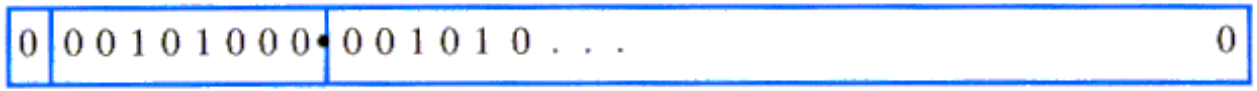
Single precision

0<E'<255

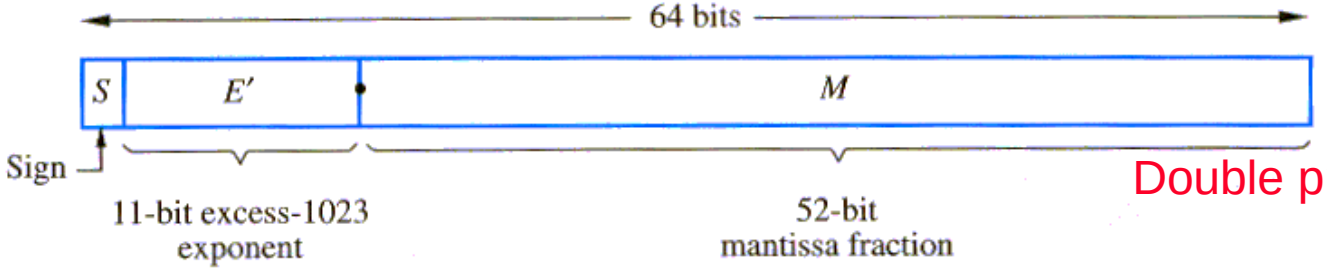
2^{-126} to 2^{+127}



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



Value represented = $1.001010\ldots0 \times 2^{-87}$



Double precision

0<E'<2047

2^{-1022} to 2^{1023}

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

IEEE-754 Floating-Point Standard (Cont'd)

° Review - Examples in the IEEE-754 Format

	Value	Bit Pattern		
		Sign	Exponent	Fraction
(a)	$+1.101 \times 2^5$	0	1000 0100	101 0000 0000 0000 0000 0000
(b)	-1.01011×2^{-126}	1	0000 0001	010 1100 0000 0000 0000 0000
(c)	$+1.0 \times 2^{127}$	0	1111 1110	000 0000 0000 0000 0000 0000
(d)	$+0$	0	0000 0000	000 0000 0000 0000 0000 0000
(e)	-0	1	0000 0000	000 0000 0000 0000 0000 0000
(f)	$+\infty$	0	1111 1111	000 0000 0000 0000 0000 0000
(g)	denormalized $+2^{-128}$	0	0000 0000	010 0000 0000 0000 0000 0000
(h)	+NaN	0	1111 1111	011 0111 0000 0000 0000 0000
(i)	$+2^{-128}$	0	011 0111 1111	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Double-precision

$\frac{0}{0}, \frac{\infty}{\infty}, \sqrt{-1}$

Clean zero

IEEE-754 Floating-Point Standard (Cont'd)

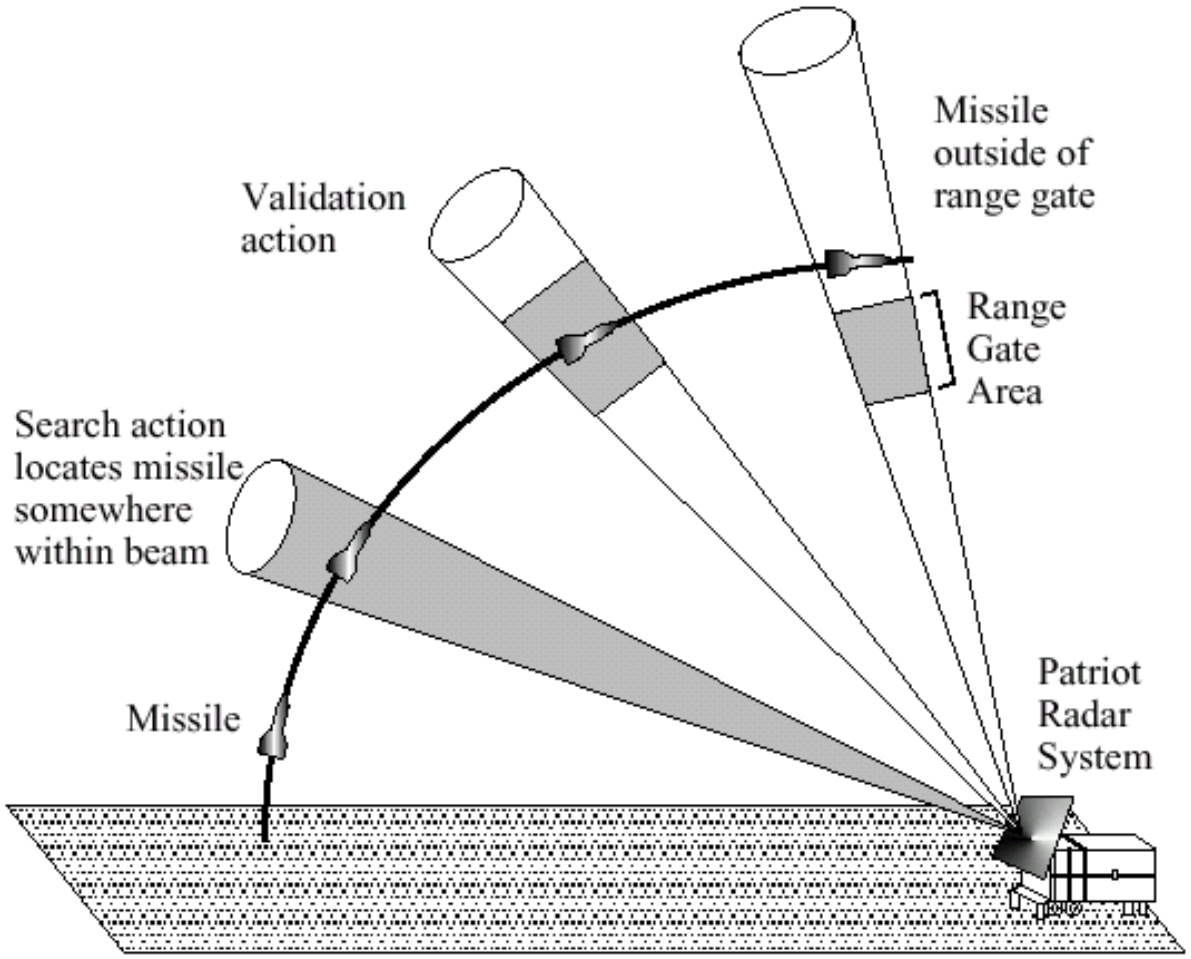
- Review - IEEE-754 conversion example: Represent -12.625_{10} in single precision IEEE-754 format.
- Step 1: Convert to the target base 2.
  - $-12.625_{10} = -1100.101_2$
- Step 2: Normalize the number in base 2.
  - $-1100.101_2 = -1.100101 \times 2^3$
- Step 3: Fill in bits.
  - Sign bit = 1 (negative)
  - Exponent is 3 + excess-127 ➔ $3 + 127 = 130 = 10000010_2$
  - Leading 1 of fraction is hidden, thus the final bit pattern is:

1 1000 0010 1001 0100 0000 0000 0000 000

IEEE-754 Floating-Point Standard (Cont'd)

° Effect of Loss of Precision

- **According to the General Accounting Office of the U.S. Government, a loss of precision in converting 24-bit integers into 24-bit floating point numbers was responsible for the failure of a Patriot anti-missile battery.**

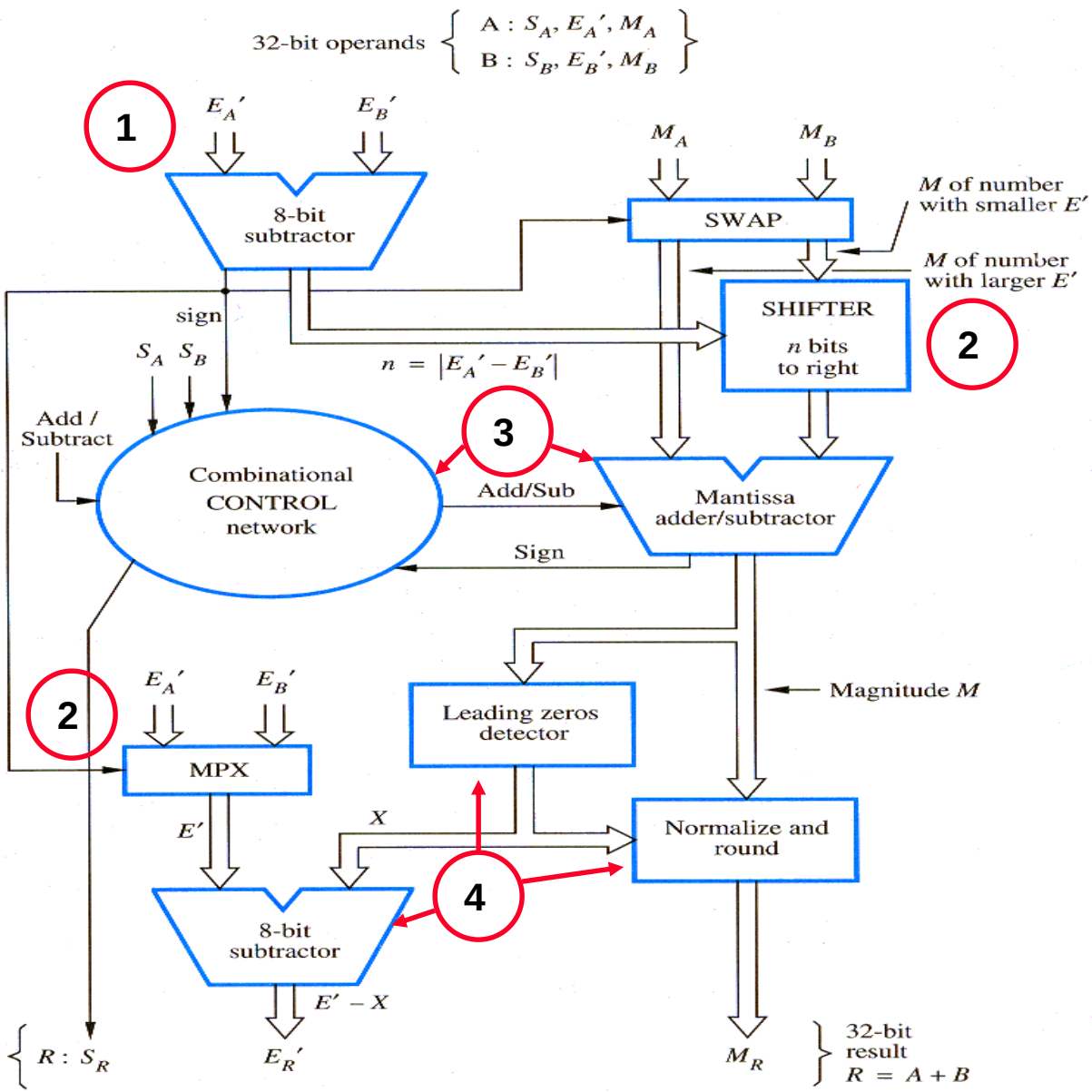


Floating-Point Arithmetic Operations

- IEEE-754 single-precision rules:
- Add/Subtract Rule:
  - Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
  - Set the exponent of the result equal to the larger exponent.
  - Perform addition/subtraction on the mantissas and determine the sign of the result.
  - Normalize the resulting value, if necessary.
- Multiply Rule:
  - Add the exponents and subtract 127.
  - Multiply the mantissas and determine the sign of result.
  - Normalize the resulting value, if necessary.
- Divide Rule:
  - Subtract the exponents and add 127.
  - Divide the mantissas and determine the sign of the result.
  - Normalize the resulting value, if necessary.

Floating-Point Arithmetic Operations (Cont'd)

- Floating-point addition/subtrac unit



Floating-Point Arithmetic Operations (Cont'd)

- Truncation methods: in the preceding algorithms, the mantissas of initial operands and final results are limited to 24 bits, including the implicit leading 1. However, it is important to retain extra bits, often called guard bits, during the intermediate steps. This yields maximum accuracy in the final results.
- In generating a final result, the extended mantissa bits (guard bits) need to be *truncated* such that the new 24-bit mantissa approximates the longer version. There are three different truncation methods.
- Suppose we want to truncate a fraction from six to three bits:
  - Chopping: where all fractions in the range $0.b_{-1}b_{-2}b_{-3}000$ to $0.\ b_{-1}b_{-2}b_{-3}111$ are truncated to $0.\ b_{-1}b_{-2}b_{-3}$.
  - Von Neumann rounding: if the three bits to be removed are all 0s, they are simply dropped. However, if any of the bits to be removed is 1, the least significant bit of the retained bits is set to 1. Thus, in our example, all 6-bit fractions with $b_{-4}b_{-5}b_{-6}$ not equal to 000 are truncated to $0.\ b_{-1}b_{-2}1$.

Floating-Point Arithmetic Operations (Cont'd)

- Rounding: a 1 is added to the LSB position of the bits to be retained if there is a 1 in the MSB position of the bits being removed.
  - Thus, $0.\ b_{-1}b_{-2}b_{-3}1\ .\ .\ .$ Is rounded to $0.\ b_{-1}b_{-2}b_{-3} + 0.001$, and $0.\ b_{-1}b_{-2}b_{-3}0\ .\ .\ .$ is rounded to $0.\ b_{-1}b_{-2}b_{-3}$.
  - This provides the closest approximation to the number being truncated, except for the case in which the bits to be removed are 10…0. This is a **tie** situation; the longer value is halfway between the two closest truncated representations. To break the tie in an unbiased way, one possibility is to choose the retained bits to be the nearest even number (similar to the default mode of IEEE-754). In this regard, the value $0.b_{-1}b_{-2}0100$ is truncated to the value $0.b_{-1}b_{-2}0$, and $0.b_{-1}b_{-2}1100$ is truncated to $0.b_{-1}b_{-2}1 + 0.001$. This technique is referred to as "round to the nearest number or nearest even number in case of a tie".
  - This method is the most difficult to implement because it requires an addition operation and a possible renormalization.

Floating-Point Arithmetic Operations (Cont'd)

- IEEE-754 Guard bits: rounding method (the default mode: rounding to the nearest representable number) is used using three guard bits to be carried along during the intermediate steps in performing the operations. The first 2 guard bits are the two most significant bits of the section of the mantissa to be removed. The third bit (sticky bit) is the logical OR of all bits beyond these first two bits in the full representation of mantissa.

- Floating-point arithmetic example: Consider a shortened version of the IEEE floating-point format that fits into 12 bits, including the sign bit. The scale factor has an implied base of 2 and a 5-bit excess-15 exponent, with the two end values of 0 and 31 used to signify the special values exact zero and infinity, respectively. The 6-bit mantissa is normalized as in the IEEE format, with an implied 1 to the left of the binary point.

Acknowledgment

- Some of the slides have been provided and adapted from previous course instructor, Dr. Ahmad Afsahi.