

Part III

The Arithmetic/Logic Unit

Parts	Chapters
I. Background and Motivation	1. Combinational Digital Circuits 2. Digital Circuits with Memory 3. Computer System Technology 4. Computer Performance
II. Instruction-Set Architecture	5. Instructions and Addressing 6. Procedures and Data 7. Assembly Language Programs 8. Instruction-Set Variations
III. The Arithmetic/Logic Unit	9. Number Representation 10. Adders and Simple ALUs 11. Multipliers and Dividers 12. Floating-Point Arithmetic
IV. Data Path and Control	13. Instruction Execution Steps 14. Control Unit Synthesis 15. Pipelined Data Paths 16. Pipeline Performance Limits
V. Memory System Design	17. Main Memory Concepts 18. Cache Memory Organization 19. Mass Memory Concepts 20. Virtual Memory and Paging
VI. Input/Output and Interfacing	21. Input/Output Devices 22. Input/Output Programming 23. Buses, Links, and Interfacing 24. Context Switching and Interrupts
VII. Advanced Architectures	25. Road to Higher Performance 26. Vector and Array Processing 27. Shared-Memory Multiprocessing 28. Distributed Multicomputing

About This Presentation

This presentation is intended to support the use of the textbook *Computer Architecture: From Microprocessors to Supercomputers*, Oxford University Press, 2005, ISBN 0-19-515455-X. It is updated regularly by the author as part of his teaching of the upper-division course ECE 154, Introduction to Computer Architecture, at the University of California, Santa Barbara. Instructors can use these slides freely in classroom teaching and for other educational purposes. Any other use is strictly prohibited. © Behrooz Parhami

Edition	Released	Revised	Revised	Revised	Revised
First	July 2003	July 2004	July 2005	Mar. 2006	Jan. 2007
		Jan. 2008	Jan. 2009	Jan. 2011	Oct. 2014

III The Arithmetic/Logic Unit

Overview of computer arithmetic and ALU design:

- Review representation methods for signed integers
- Discuss algorithms & hardware for arithmetic ops
- Consider floating-point representation & arithmetic

Topics in This Part

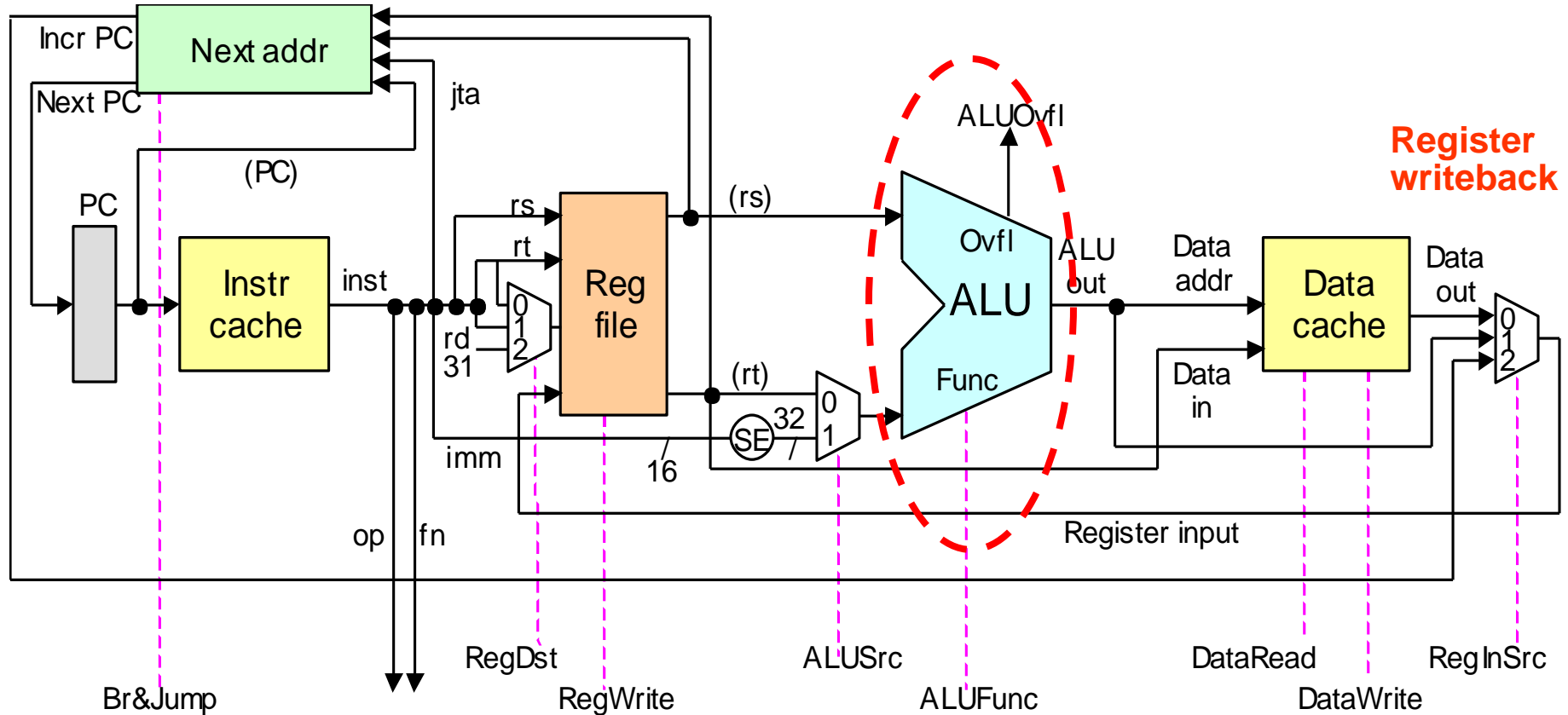
Chapter 9 Number Representation

Chapter 10 Adders and Simple ALUs

Chapter 11 Multipliers and Dividers

Chapter 12 Floating-Point Arithmetic

Preview of Arithmetic Unit in the Data Path



Instruction fetch

Reg access / decode

ALU operation

Data access

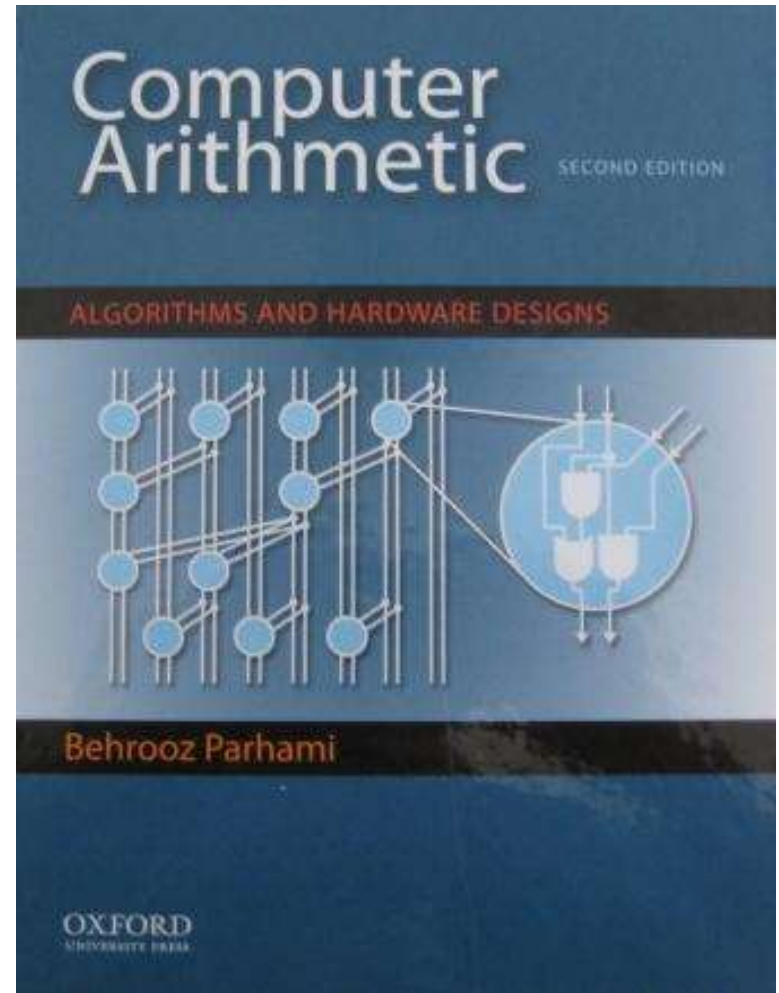
Fig. 13.3 Key elements of the single-cycle MicroMIPS data path.

Computer Arithmetic as a Topic of Study

Brief overview article –
Encyclopedia of Info Systems,
Academic Press, 2002,
Vol. 3, pp. 317-333

Our textbook's treatment
of the topic falls between
the extremes (4 chaps.)

Graduate course
ECE 252B – Text:
Computer Arithmetic,
Oxford U Press, 2000
(2nd ed., 2010)



9 Number Representation

Arguably the most important topic in computer arithmetic:

- Affects system compatibility and ease of arithmetic
- Two's complement, flip, and unconventional methods

Topics in This Chapter

9.1 Positional Number Systems

9.2 Digit Sets and Encodings

9.3 Number-Radix Conversion

9.4 Signed Integers

9.5 Fixed-Point Numbers

9.6 Floating-Point Numbers

9.1 Positional Number Systems

Representations of natural numbers {0, 1, 2, 3, ...}

|||| | |||| | |||| | |||| | ||

sticks or *unary* code

27

radix-10 or *decimal* code

11011

radix-2 or *binary* code

XXVII

Roman numerals

Fixed-radix positional representation with k digits

Value of a number: $x = (x_{k-1}x_{k-2} \dots x_1x_0)_r = \sum_{i=0}^{k-1} x_i r^i$

For example:

$$27 = (11011)_{\text{two}} = (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$$

Number of digits for $[0, P]$: $k = \lceil \log_r(P + 1) \rceil = \lfloor \log_r P \rfloor + 1$

Unsigned Binary Integers

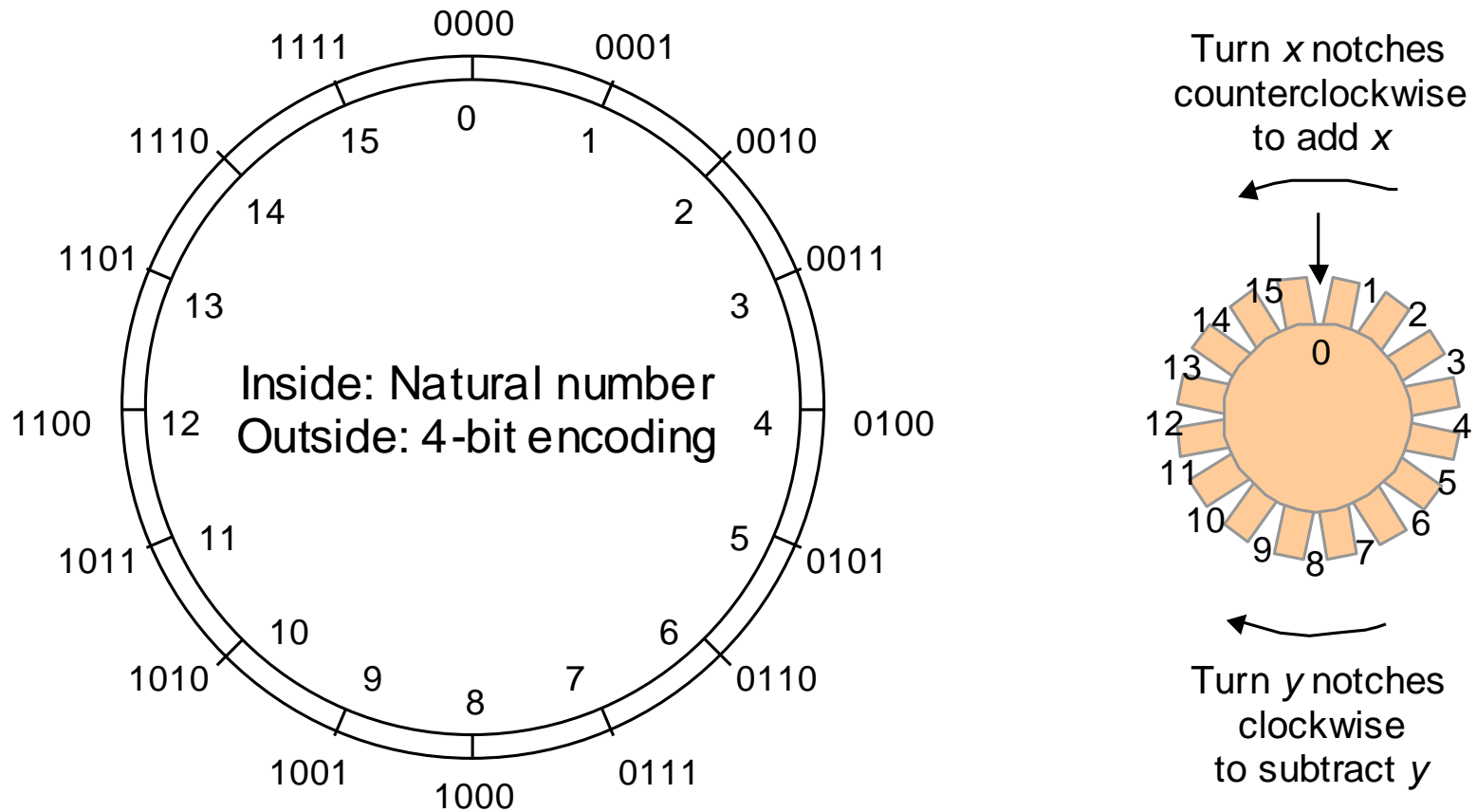


Figure 9.1 Schematic representation of 4-bit code for integers in [0, 15].

Representation Range and Overflow

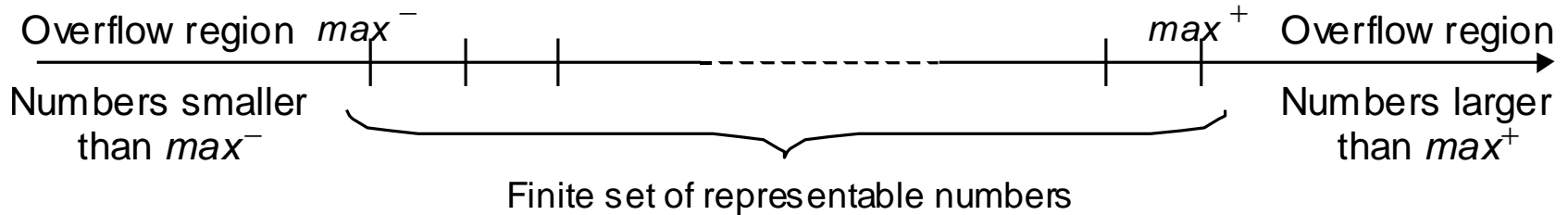


Figure 9.2 Overflow regions in finite number representation systems. For unsigned representations covered in this section, $max^- = 0$.

Example 9.2, Part d

Discuss if overflow will occur when computing $3^{17} - 3^{16}$ in a number system with $k = 8$ digits in radix $r = 10$.

Solution

The result 86 093 442 is representable in the number system which has a range $[0, 99\,999\,999]$; however, if 3^{17} is computed en route to the final result, overflow will occur.

9.2 Digit Sets and Encodings

Conventional and unconventional digit sets

- Decimal digits in $[0, 9]$; 4-bit BCD, 8-bit ASCII
- Hexadecimal, or hex for short: digits 0-9 & a-f
- Conventional ternary digit set in $[0, 2]$
Conventional digit set for radix r is $[0, r - 1]$
Symmetric ternary digit set in $[-1, 1]$
- Conventional binary digit set in $[0, 1]$
Redundant digit set $[0, 2]$, encoded in 2 bits
 $(0\ 2\ 1\ 1\ 0)_{\text{two}}$ and $(1\ 0\ 1\ 0\ 2)_{\text{two}}$ represent 22

Carry-Save Numbers

Radix-2 numbers using the digits 0, 1, and 2

Example: $(1\ 0\ 2\ 1)_{\text{two}} = (1 \times 2^3) + (0 \times 2^2) + (2 \times 2^1) + (1 \times 2^0) = 13$

Possible encodings

(a) Binary

0	00
1	01
2	10
	11 (Unused)

(b) Unary

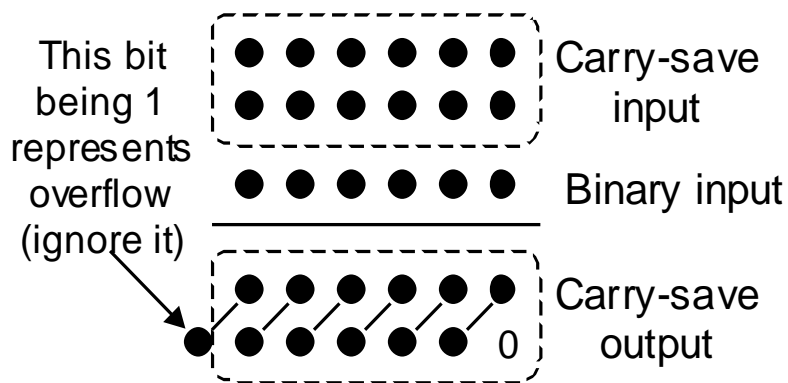
0	00
1	01 (First alternate)
1	10 (Second alternate)
2	11

	<u>1 0 2 1</u>
MSB	0 0 1 0 = 2
LSB	1 0 0 1 = 9

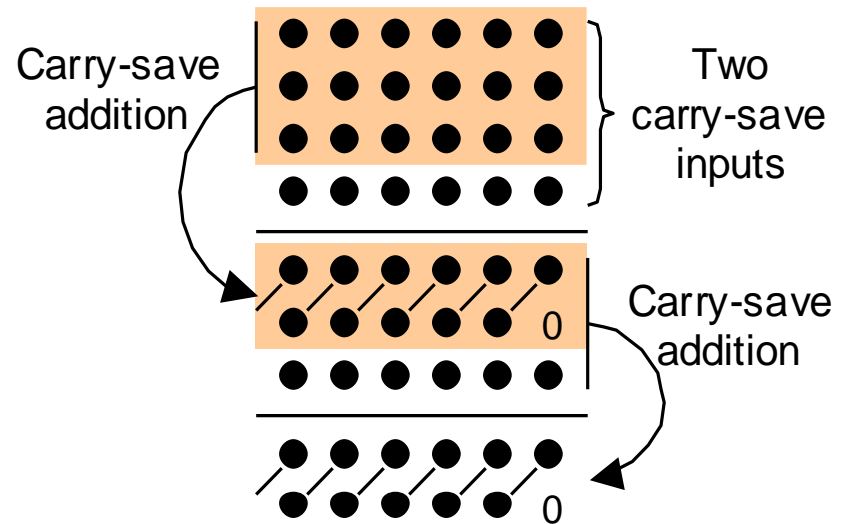
	<u>1 0 2 1</u>
First bit	0 0 1 1 = 3
Second bit	1 0 1 0 = 10

The Notion of Carry-Save Addition

Digit-set combination: $\{0, 1, 2\} + \{0, 1\} = \{0, 1, 2, 3\} = \{0, 2\} + \{0, 1\}$



a. Carry-save addition.



b. Adding two carry-save numbers.

Figure 9.3 Adding a binary number or another carry-save number to a carry-save number.

9.3 Number Radix Conversion

Two ways to convert numbers from an old radix r to a new radix R

- Perform arithmetic in the new radix R

Suitable for conversion from radix r to radix 10

Horner's rule:

$$(x_{k-1}x_{k-2} \dots x_1x_0)_r = (\dots((0 + x_{k-1})r + x_{k-2})r + \dots + x_1)r + x_0$$

$$(1\ 0\ 1\ 1\ 0\ 1\ 0\ 1)_{\text{two}} = 0 + 1 \rightarrow 1 \times 2 + 0 \rightarrow 2 \times 2 + 1 \rightarrow 5 \times 2 + 1 \rightarrow 11 \times 2 + 0 \rightarrow 22 \times 2 + 1 \rightarrow 45 \times 2 + 0 \rightarrow 90 \times 2 + 1 \rightarrow 181$$

- Perform arithmetic in the old radix r

Suitable for conversion from radix 10 to radix R

Divide the number by R , use the remainder as the LSD
and the quotient to repeat the process

$19 / 3 \rightarrow \text{rem } 1, \text{ quo } 6 / 3 \rightarrow \text{rem } 0, \text{ quo } 2 / 3 \rightarrow \text{rem } 2, \text{ quo } 0$

Thus, $19 = (2\ 0\ 1)_{\text{three}}$

Justifications for Radix Conversion Rules

$$\begin{aligned}(x_{k-1}x_{k-2}\cdots x_0)_r &= x_{k-1}r^{k-1} + x_{k-2}r^{k-2} + \cdots + x_1r + x_0 \\ &= x_0 + r(x_1 + r(x_2 + r(\cdots)))\end{aligned}$$

Justifying Horner's rule.



Figure 9.4 Justifying one step of the conversion of x to radix 2.

9.4 Signed Integers

- We dealt with representing the natural numbers
- Signed or directed whole numbers = integers
 $\{ \dots, -3, -2, -1, 0, 1, 2, 3, \dots \}$

- Signed-magnitude representation

+27 in 8-bit signed-magnitude binary code 0 0011011

–27 in 8-bit signed-magnitude binary code 1 0011011

–27 in 2-digit decimal code with BCD digits 1 0010 0111

- Biased representation

Represent the interval of numbers $[-N, P]$ by the unsigned interval $[0, P + N]$; i.e., by adding N to every number

Two's-Complement Representation

With k bits, numbers in the range $[-2^{k-1}, 2^{k-1} - 1]$ represented. Negation is performed by inverting all bits and adding 1.

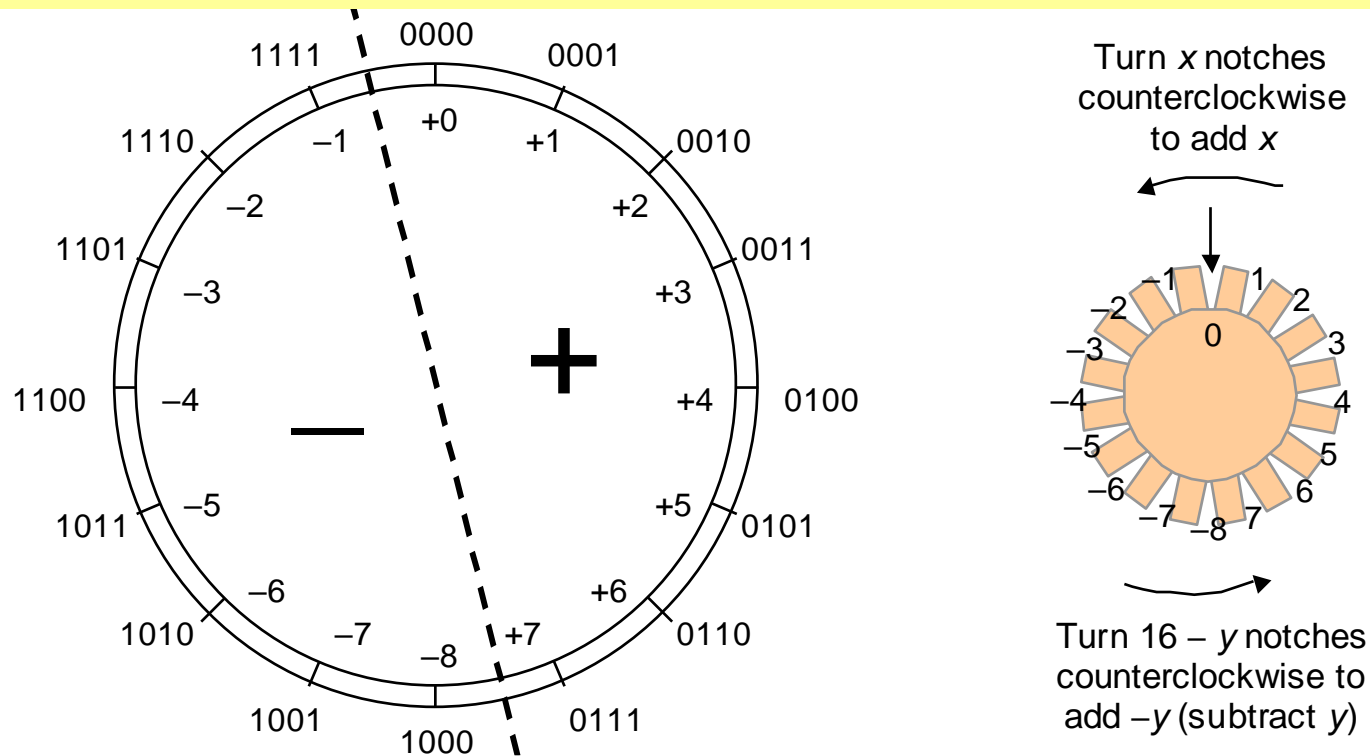


Figure 9.5 Schematic representation of 4-bit 2's-complement code for integers in $[-8, +7]$.

Conversion from 2's-Complement to Decimal

Example 9.7

Convert $x = (1\ 0\ 1\ 1\ 0\ 1\ 0\ 1)_{2\text{'s-compl}}$ to decimal.

Solution

Given that x is negative, one could change its sign and evaluate $-x$.

Shortcut: Use Horner's rule, but take the MSB as negative

$$\begin{aligned} -1 \times 2 + 0 &\rightarrow -2 \times 2 + 1 \rightarrow -3 \times 2 + 1 \rightarrow -5 \times 2 + 0 \rightarrow -10 \times 2 + 1 \\ &\rightarrow -19 \times 2 + 0 \rightarrow -38 \times 2 + 1 \rightarrow -75 \end{aligned}$$

Sign Change for a 2's-Complement Number

Example 9.8

Given $y = (1\ 0\ 1\ 1\ 0\ 1\ 0\ 1)_{2\text{'s-compl}}$, find the representation of $-y$.

Solution

$$-y = (0\ 1\ 0\ 0\ 1\ 0\ 1\ 0) + 1 = (0\ 1\ 0\ 0\ 1\ 0\ 1\ 1)_{2\text{'s-compl}} \quad (\text{i.e., } 75)$$

Two's-Complement Addition and Subtraction

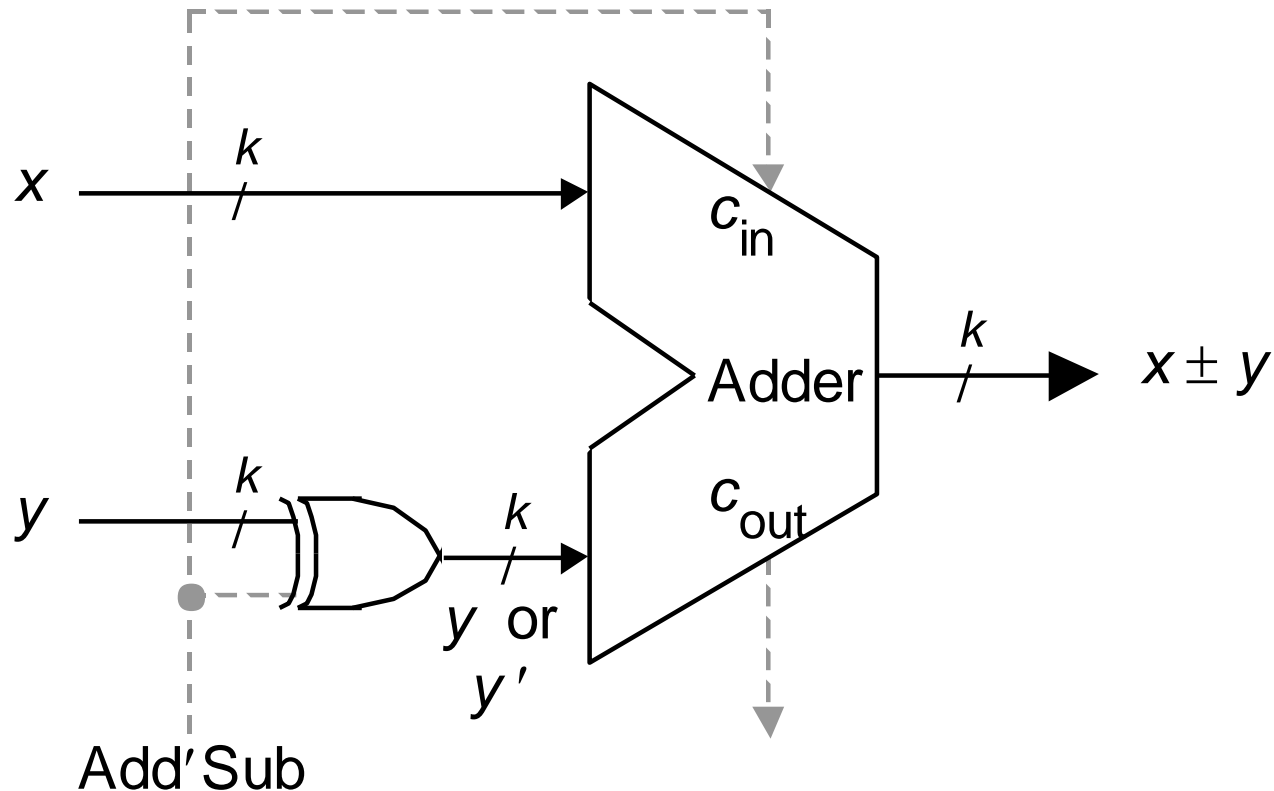


Figure 9.6 Binary adder used as 2's-complement adder/subtractor.

9.5 Fixed-Point Numbers

Positional representation: k whole and l fractional digits

Value of a number: $x = (x_{k-1}x_{k-2} \dots x_1x_0 \cdot x_{-1}x_{-2} \dots x_{-l})_r = \sum x_i r^i$

For example:

$$2.375 = (10.011)_{\text{two}} = (1 \times 2^1) + (0 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3})$$

Numbers in the range $[0, r^k - ulp]$ representable, where $ulp = r^{-l}$

Fixed-point arithmetic same as integer arithmetic
(radix point implied, not explicit)

Two's complement properties (including sign change) hold here as well:

$$(01.011)_{2\text{'s-compl}} = (-0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) = +1.375$$

$$(11.011)_{2\text{'s-compl}} = (-1 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) = -0.625$$

Fixed-Point 2's-Complement Numbers

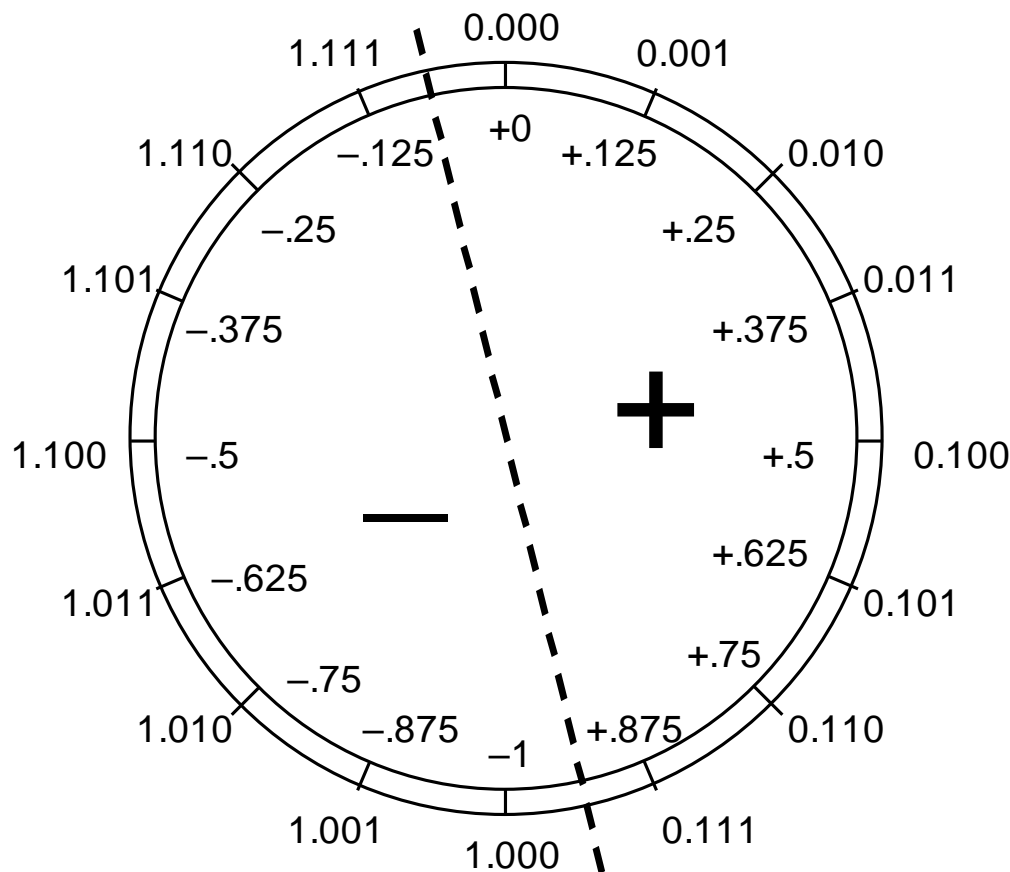


Figure 9.7 Schematic representation of 4-bit 2's-complement encoding for (1 + 3)-bit fixed-point numbers in the range $[-1, +7/8]$.

Radix Conversion for Fixed-Point Numbers

Convert the whole and fractional parts separately.

To convert the fractional part from an old radix r to a new radix R :

- **Perform arithmetic in the new radix R**

Evaluate a polynomial in r^{-1} : $(.011)_{\text{two}} = 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$

Simpler: View the fractional part as integer, convert, divide by r^l

$$(.011)_{\text{two}} = (?)_{\text{ten}}$$

Multiply by 8 to make the number an integer: $(011)_{\text{two}} = (3)_{\text{ten}}$

$$\text{Thus, } (.011)_{\text{two}} = (3 / 8)_{\text{ten}} = (.375)_{\text{ten}}$$

- **Perform arithmetic in the old radix r**

Multiply the given fraction by R , use the whole part as the MSD and the fractional part to repeat the process

$$(.72)_{\text{ten}} = (?)_{\text{two}}$$

$0.72 \times 2 = 1.44$, so the answer begins with 0.1

$0.44 \times 2 = 0.88$, so the answer begins with 0.10

9.6 Floating-Point Numbers

Useful for applications where very large and very small numbers are needed simultaneously

- Fixed-point representation must sacrifice precision for small values to represent large values

$$x = (0000\ 0000\ .\ 0000\ 1001)_{\text{two}} \quad \text{Small number}$$

$$y = (1001\ 0000\ .\ 0000\ 0000)_{\text{two}} \quad \text{Large number}$$

- Neither y^2 nor y/x is representable in the format above

- Floating-point representation is like scientific notation:

$$-20\,000\,000 = -2 \times 10^7 \quad +0.000\,000\,007 = +7 \times 10^{-9}$$

Sign

Significand

Exponent base

Exponent

Also, $7\text{E}-9$

ANSI/IEEE Standard Floating-Point Format (IEEE 754)

Revision (IEEE 754R) was completed in 2008: The revised version includes 16-bit and 128-bit binary formats, as well as 64- and 128-bit decimal formats

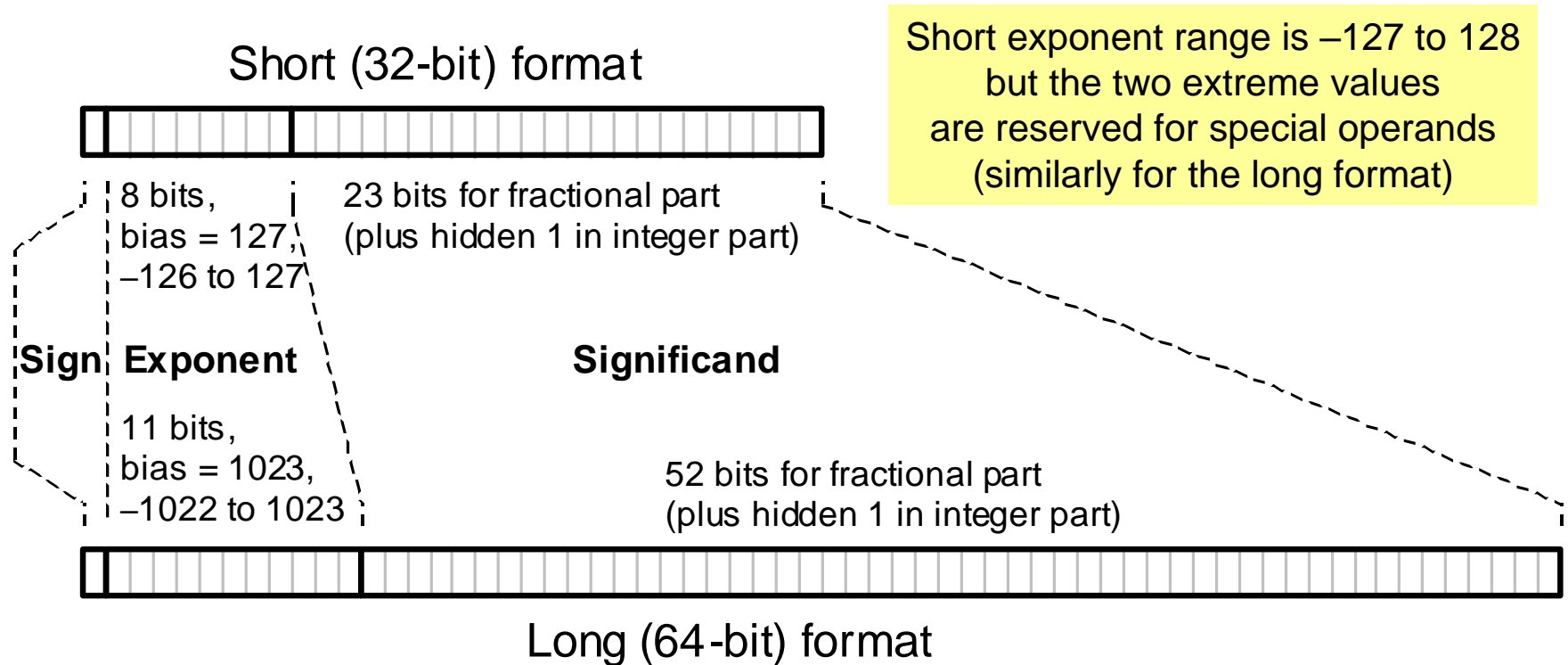


Figure 9.8 The two ANSI/IEEE standard floating-point formats.

Short and Long IEEE 754 Formats: Features

Table 9.1 Some features of ANSI/IEEE standard floating-point formats

Feature	Single/Short	Double/Long
Word width in bits	32	64
Significand in bits	23 + 1 hidden	52 + 1 hidden
Significand range	$[1, 2 - 2^{-23}]$	$[1, 2 - 2^{-52}]$
Exponent bits	8	11
Exponent bias	127	1023
Zero (± 0)	$e + \text{bias} = 0, f = 0$	$e + \text{bias} = 0, f = 0$
Denormal Subnormal	$e + \text{bias} = 0, f \neq 0$ represents $\pm 0.f \times 2^{-126}$	$e + \text{bias} = 0, f \neq 0$ represents $\pm 0.f \times 2^{-1022}$
Infinity ($\pm \infty$)	$e + \text{bias} = 255, f = 0$	$e + \text{bias} = 2047, f = 0$
Not-a-number (NaN)	$e + \text{bias} = 255, f \neq 0$	$e + \text{bias} = 2047, f \neq 0$
Ordinary number	$e + \text{bias} \in [1, 254]$ $e \in [-126, 127]$ represents $1.f \times 2^e$	$e + \text{bias} \in [1, 2046]$ $e \in [-1022, 1023]$ represents $1.f \times 2^e$
<i>min</i>	$2^{-126} \cong 1.2 \times 10^{-38}$	$2^{-1022} \cong 2.2 \times 10^{-308}$
<i>max</i>	$\cong 2^{128} \cong 3.4 \times 10^{38}$	$\cong 2^{1024} \cong 1.8 \times 10^{308}$

10 Adders and Simple ALUs

Addition is the most important arith operation in computers:

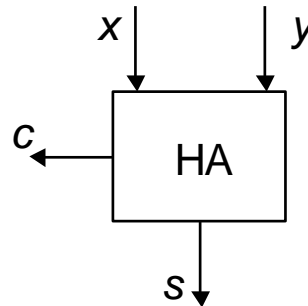
- Even the simplest computers must have an adder
- An adder, plus a little extra logic, forms a simple ALU

Topics in This Chapter

10.1	Simple Adders
10.2	Carry Propagation Networks
10.3	Counting and Incrementation
10.4	Design of Fast Adders
10.5	Logic and Shift Operations
10.6	Multifunction ALUs

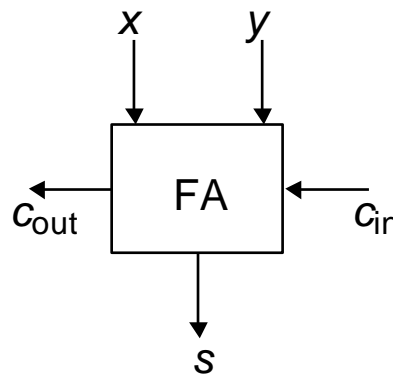
10.1 Simple Adders

Inputs		Outputs	
x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Digit-set interpretation:
 $\{0, 1\} + \{0, 1\}$
 $= \{0, 2\} + \{0, 1\}$

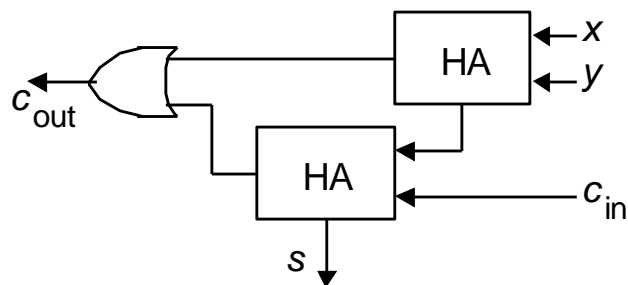
Inputs			Outputs	
x	y	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



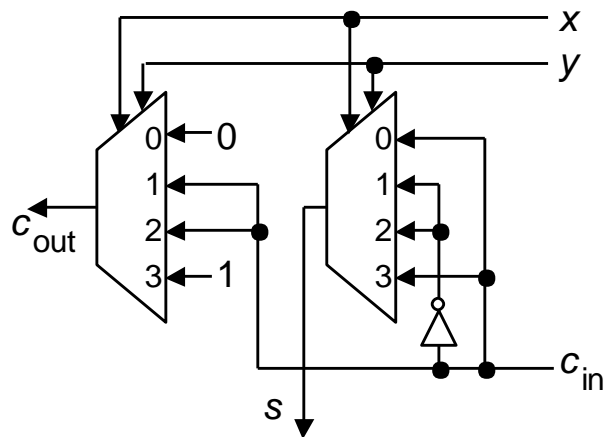
Digit-set interpretation:
 $\{0, 1\} + \{0, 1\} + \{0, 1\}$
 $= \{0, 2\} + \{0, 1\}$

Figures 10.1/10.2 Binary half-adder (HA) and full-adder (FA).

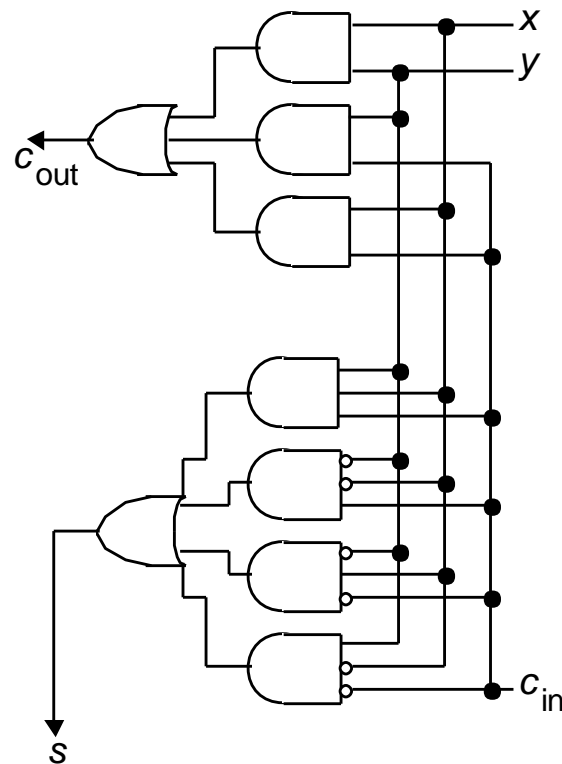
Full-Adder Implementations



(a) FA built of two HAs



(b) CMOS mux-based FA



(c) Two-level AND-OR FA

Figure 10.3 Full adder implemented with two half-adders, by means of two 4-input multiplexers, and as two-level gate network.

Ripple-Carry Adder: Slow But Simple

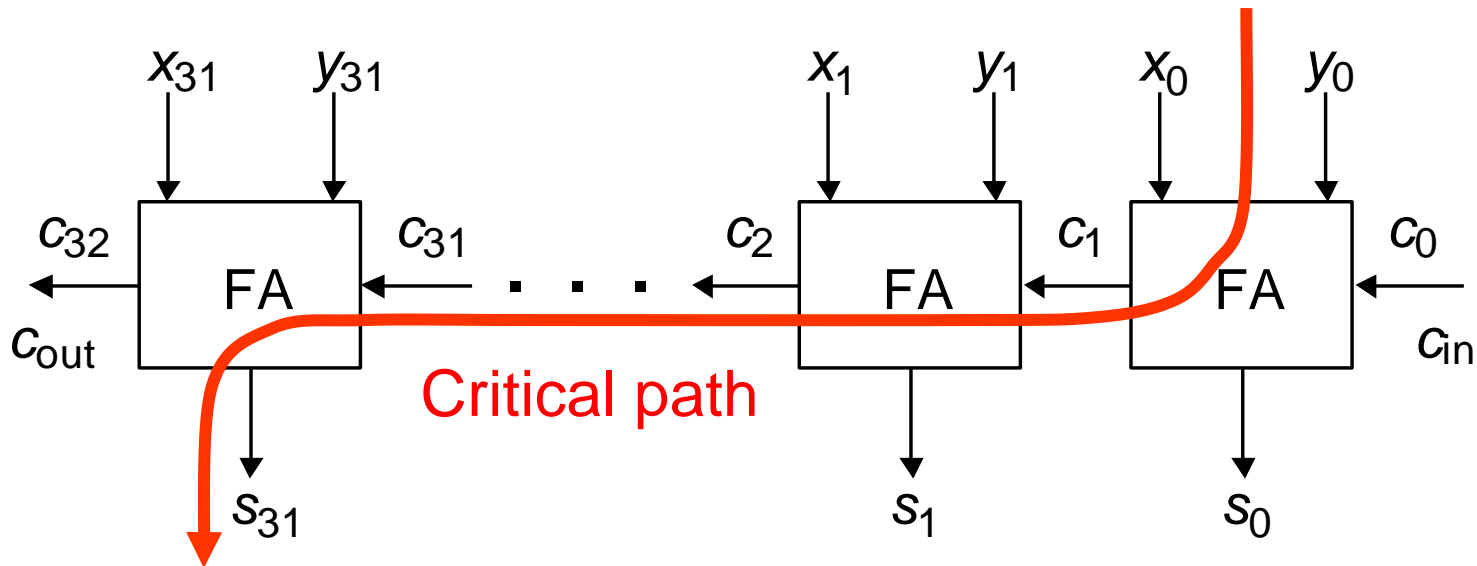
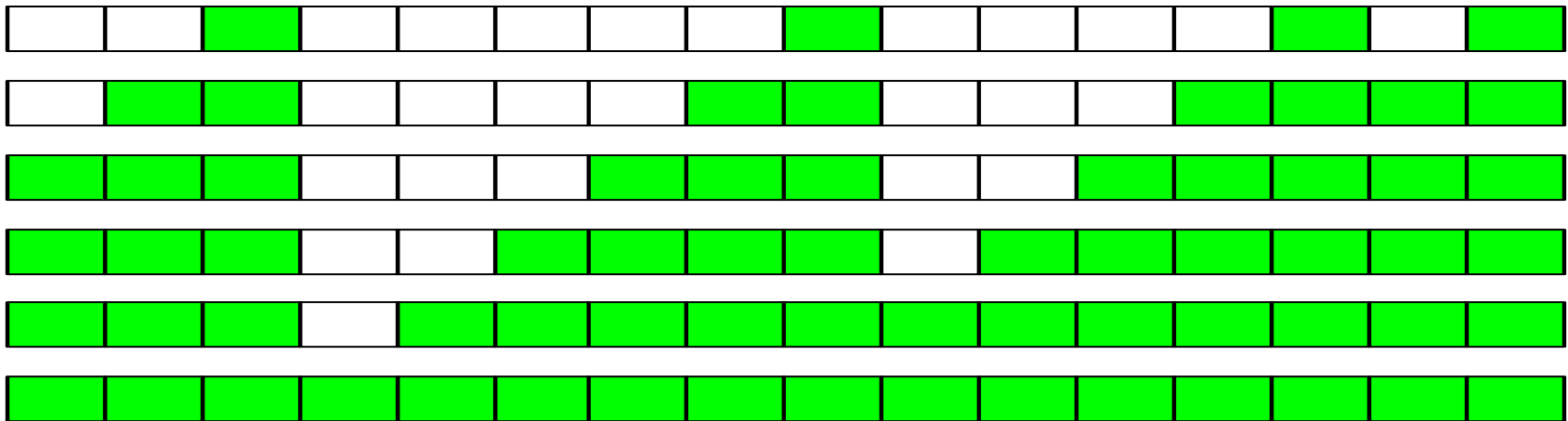
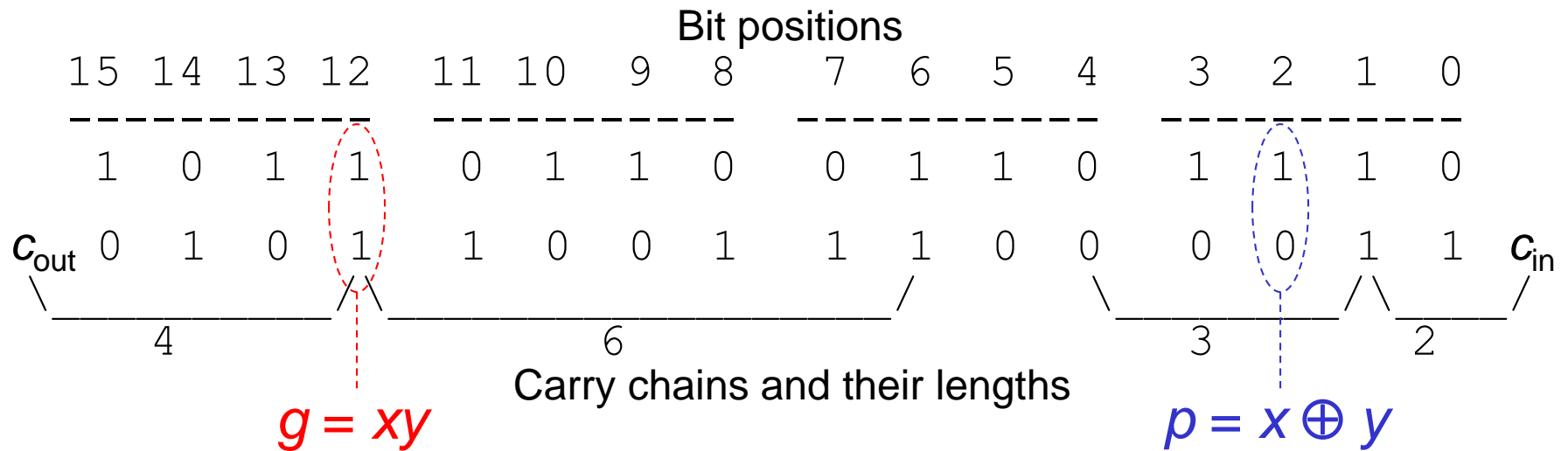
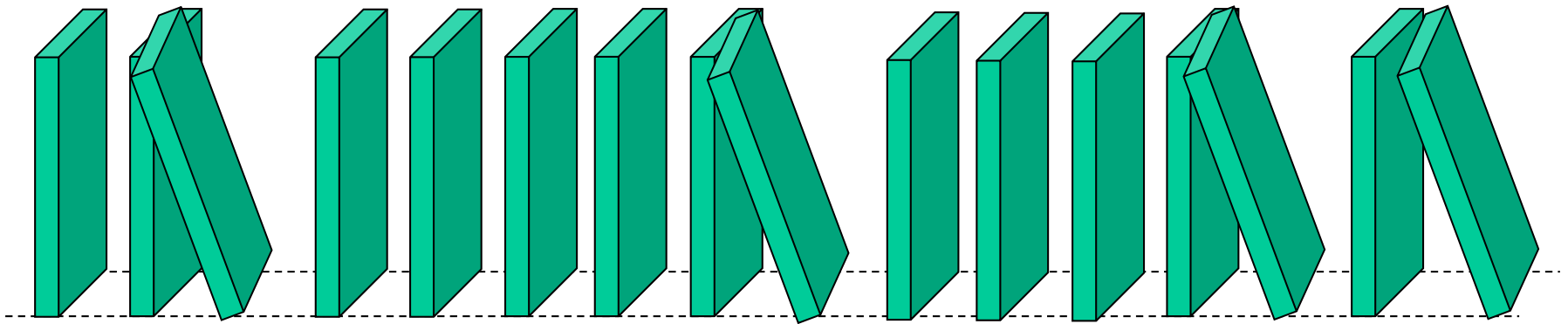
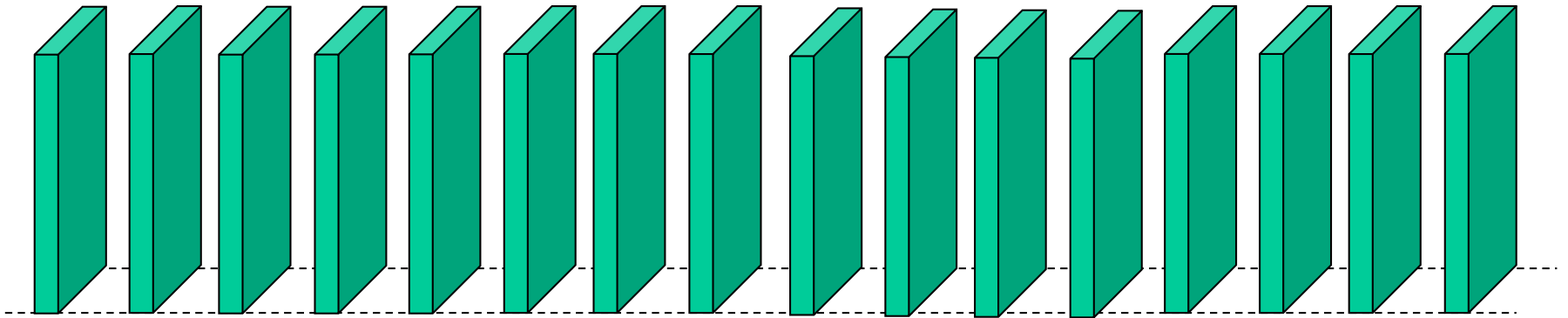
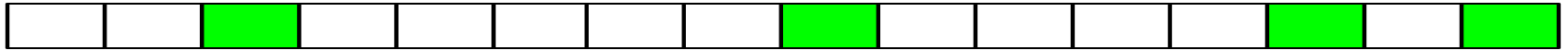


Figure 10.4 Ripple-carry binary adder with 32-bit inputs and output.

Carry Chains and Auxiliary Signals



Carry Chains Illustrated with Dominoes



10.2 Carry Propagation Networks

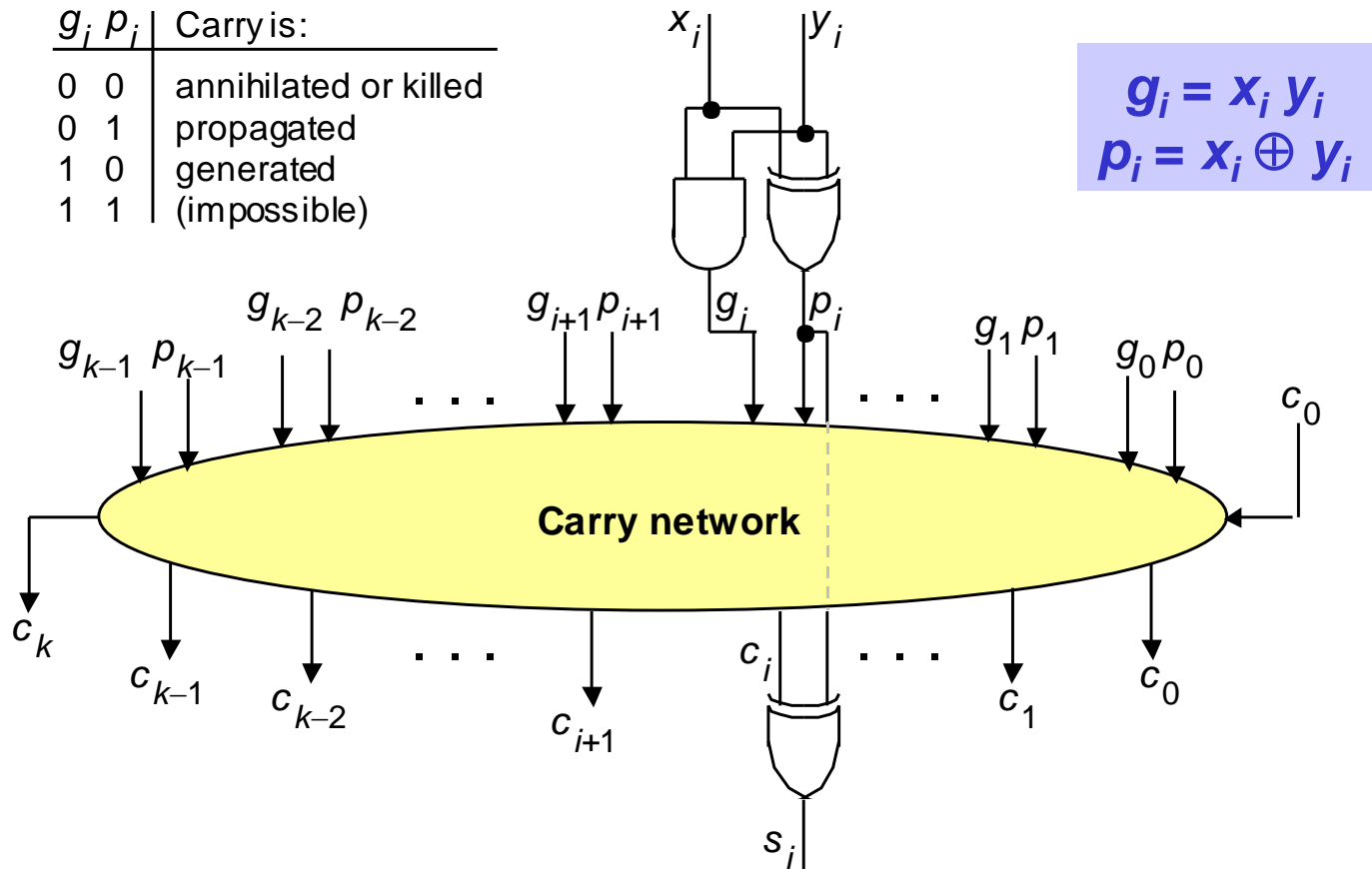


Figure 10.5 The main part of an adder is the carry network. The rest is just a set of gates to produce the g and p signals and the sum bits.

Ripple-Carry Adder Revisited

The carry recurrence: $c_{i+1} = g_i \vee p_i c_i$

Latency of k -bit adder is roughly $2k$ gate delays:

1 gate delay for production of p and g signals, plus
2($k - 1$) gate delays for carry propagation, plus
1 XOR gate delay for generation of the sum bits

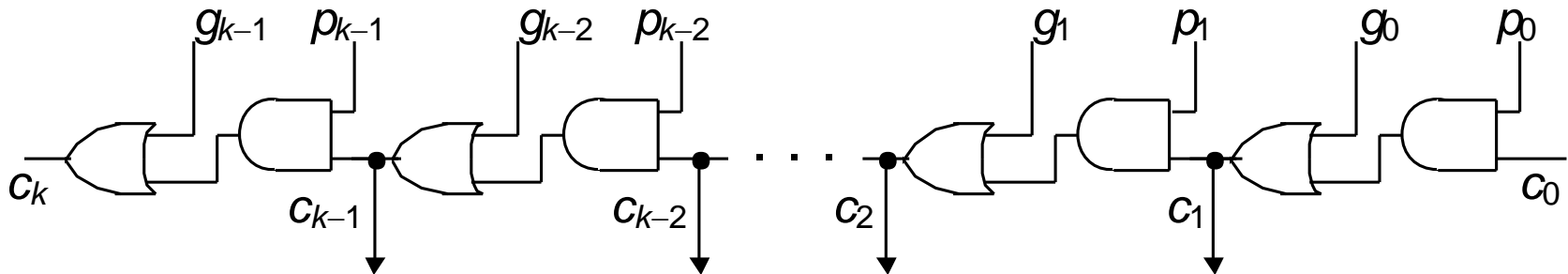


Figure 10.6 The carry propagation network of a ripple-carry adder.

The Complete Design of a Ripple-Carry Adder

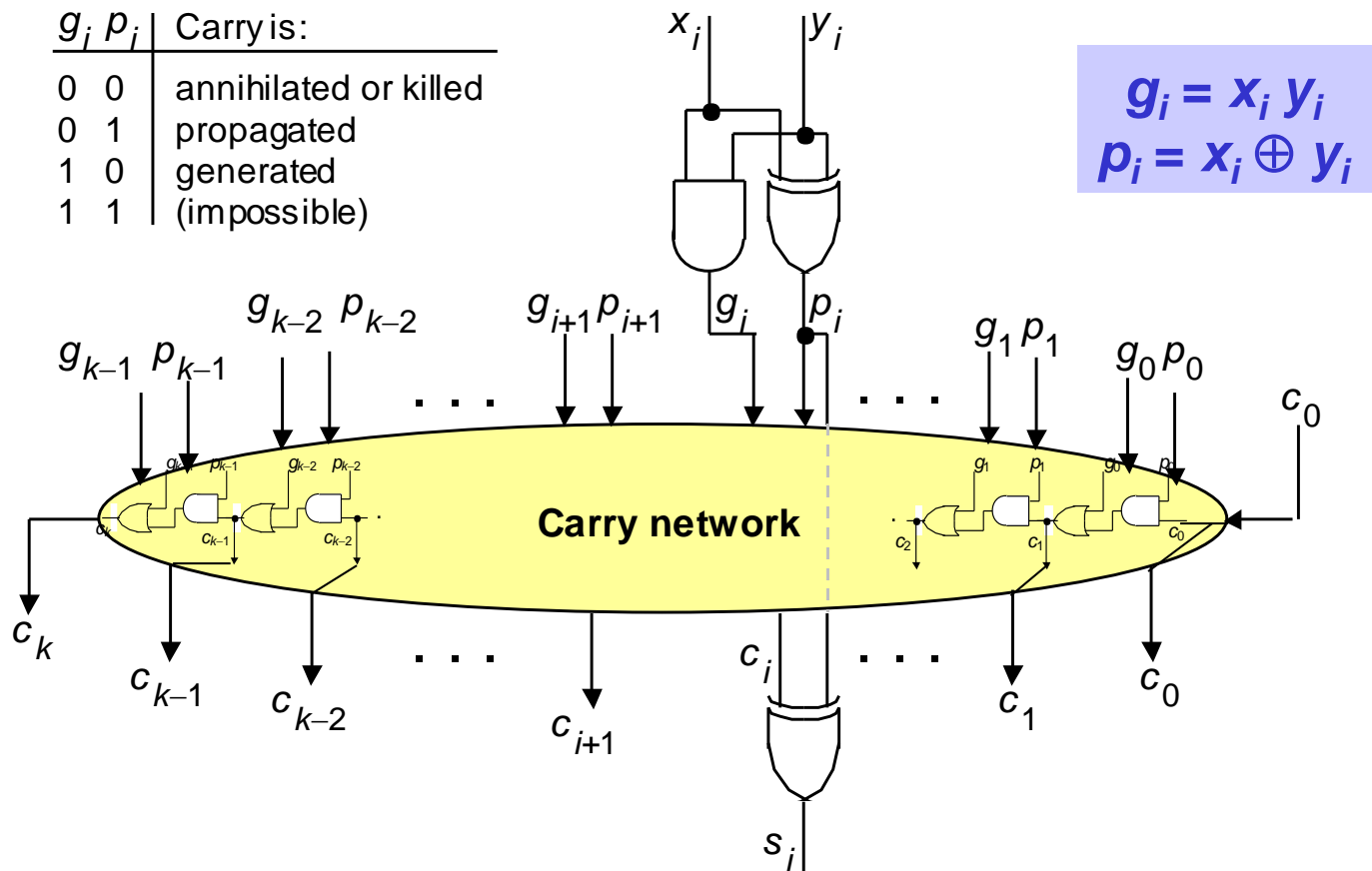
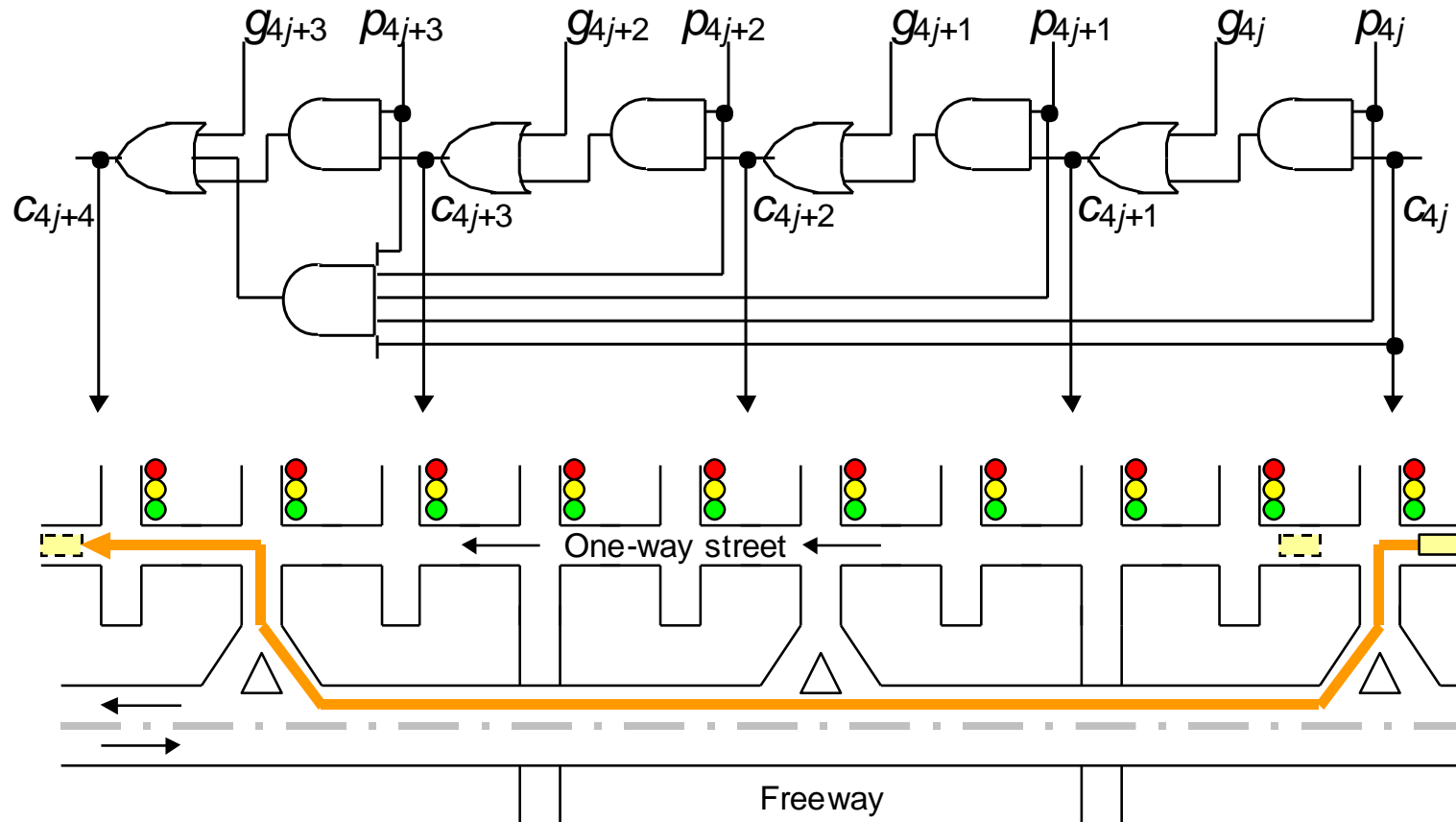


Figure 10.6 (ripple-carry network) superimposed on Figure 10.5 (general structure of an adder).

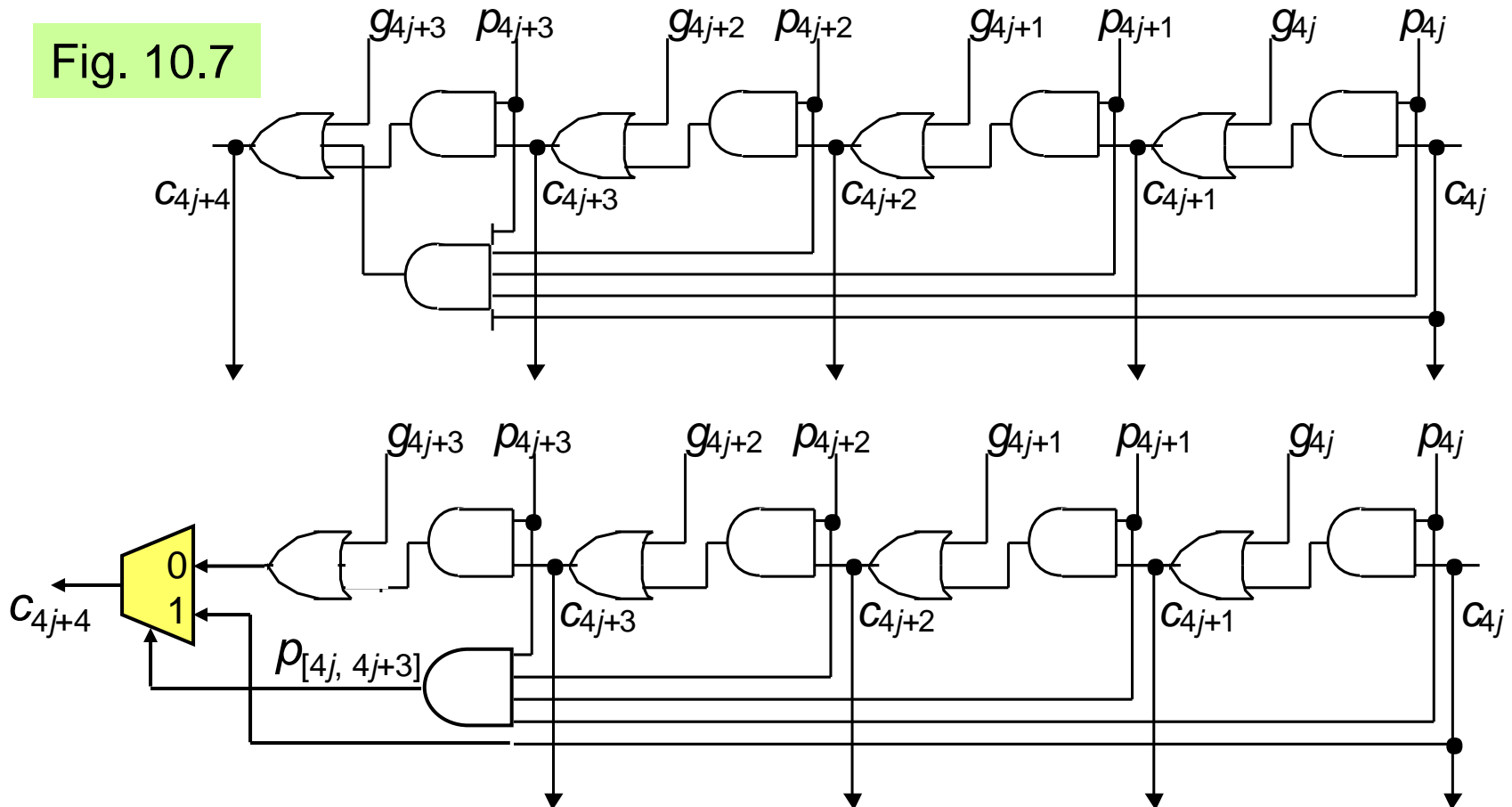
First Carry Speed-Up Method: Carry Skip



Figures 10.7/10.8 A 4-bit section of a ripple-carry network with skip paths and the driving analogy.

Mux-Based Skip Carry Logic

Fig. 10.7



The carry-skip adder of Fig. 10.7 works fine if we begin with a clean slate, where all signals are 0s; otherwise, it will run into problems, which do not exist in this mux-based implementation

10.3 Counting and Incrementation

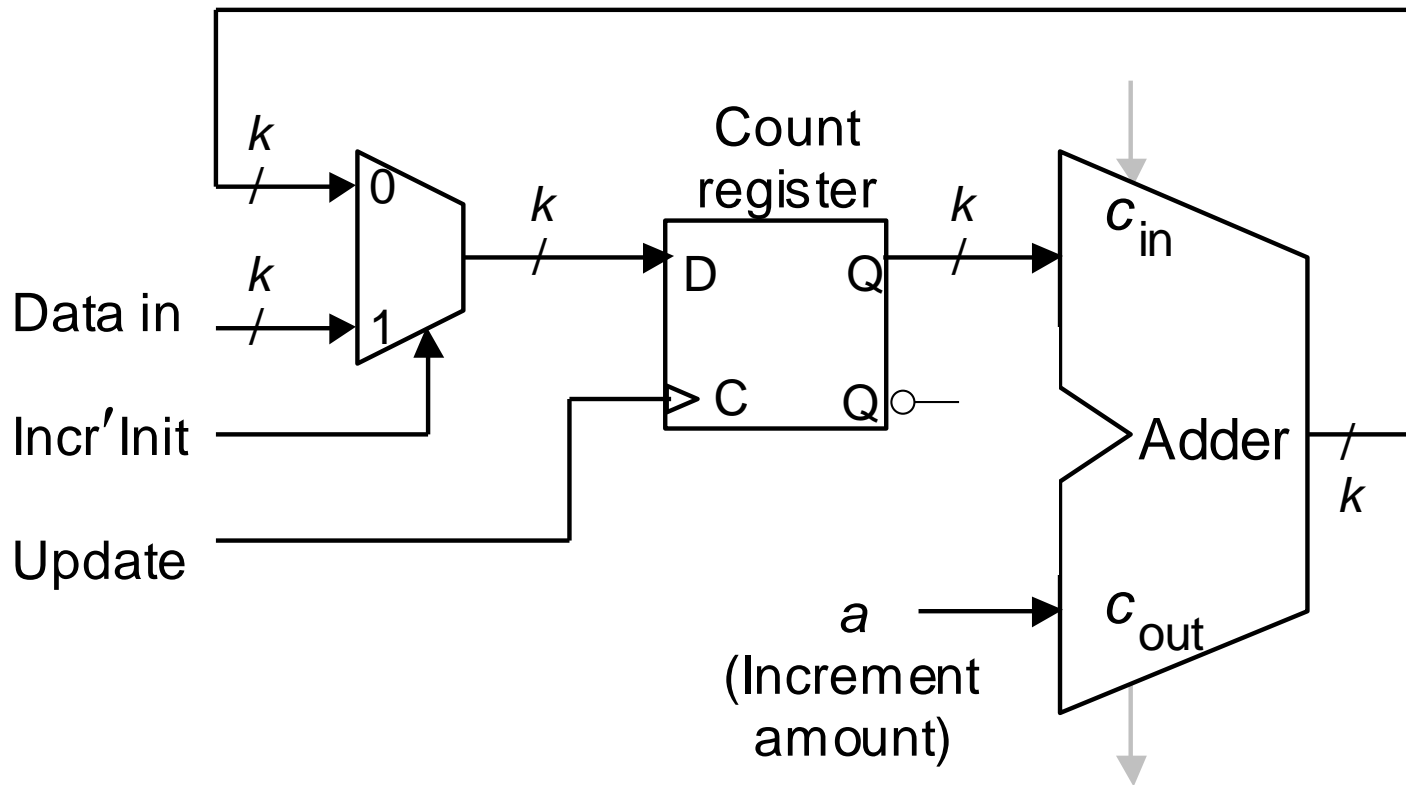


Figure 10.9 Schematic diagram of an initializable synchronous counter.

Circuit for Incrementation by 1

Figure 10.6

Substantially simpler than an adder

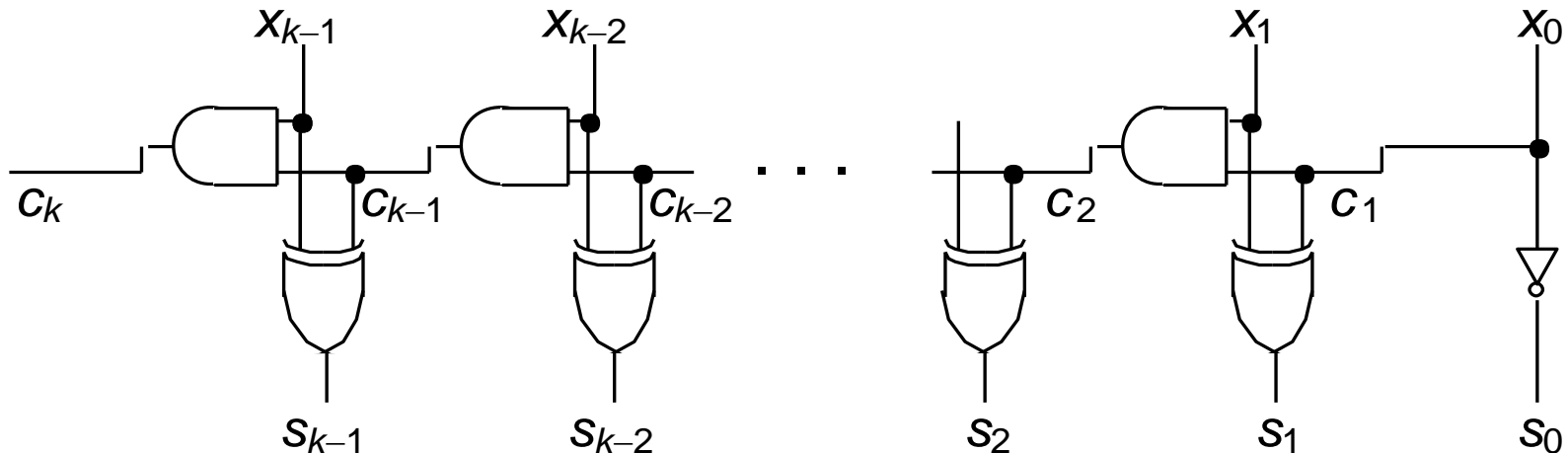
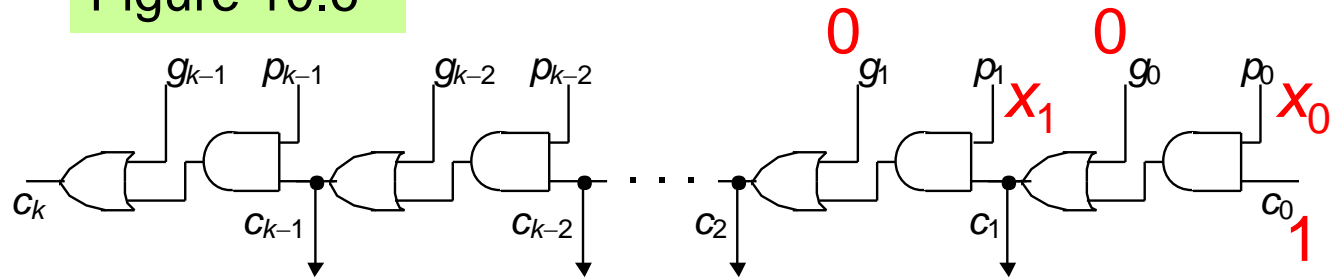


Figure 10.10 Carry propagation network and sum logic for an incrementer.

10.4 Design of Fast Adders

- Carries can be computed directly without propagation
- For example, by unrolling the equation for c_3 , we get:

$$c_3 = g_2 \vee p_2 c_2 = g_2 \vee p_2 g_1 \vee p_2 p_1 g_0 \vee p_2 p_1 p_0 c_0$$

- We define “generate” and “propagate” signals for a block extending from bit position a to bit position b as follows:

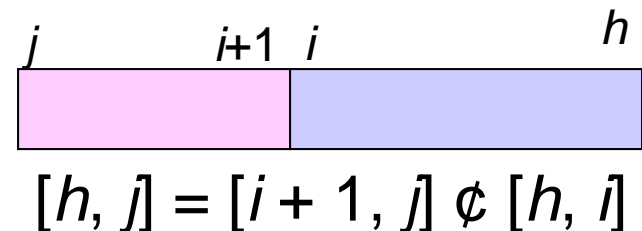
$$g_{[a,b]} = g_b \vee p_b g_{b-1} \vee p_b p_{b-1} g_{b-2} \vee \dots \vee p_b p_{b-1} \dots p_{a+1} g_a$$

$$p_{[a,b]} = p_b p_{b-1} \dots p_{a+1} p_a$$

- Combining g and p signals for adjacent blocks:

$$g_{[h,j]} = g_{[i+1,j]} \vee p_{[i+1,j]} g_{[h,i]}$$

$$p_{[h,j]} = p_{[i+1,j]} p_{[h,i]}$$



Carries as Generate Signals for Blocks $[0, i]$

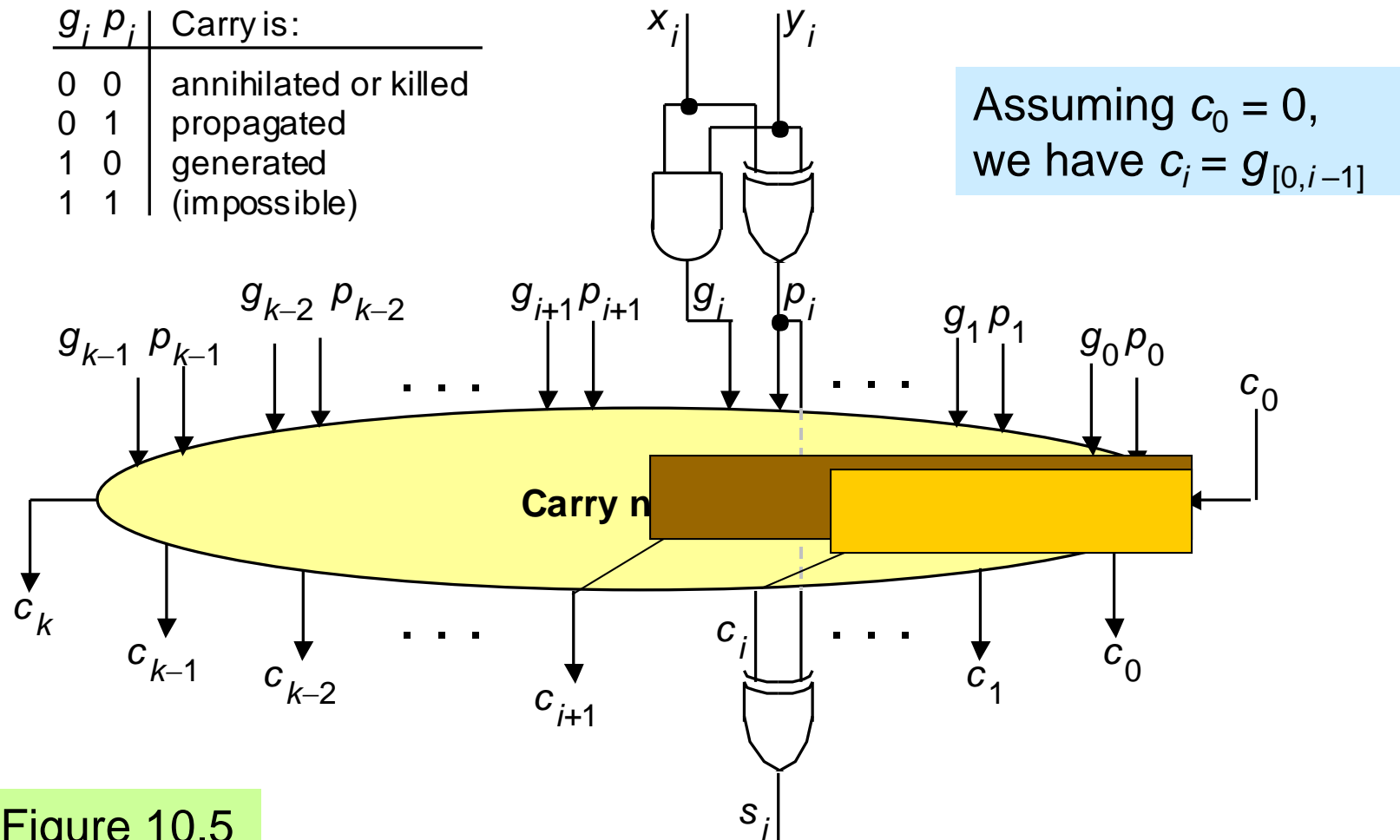


Figure 10.5

Second Carry Speed-Up Method: Carry Lookahead

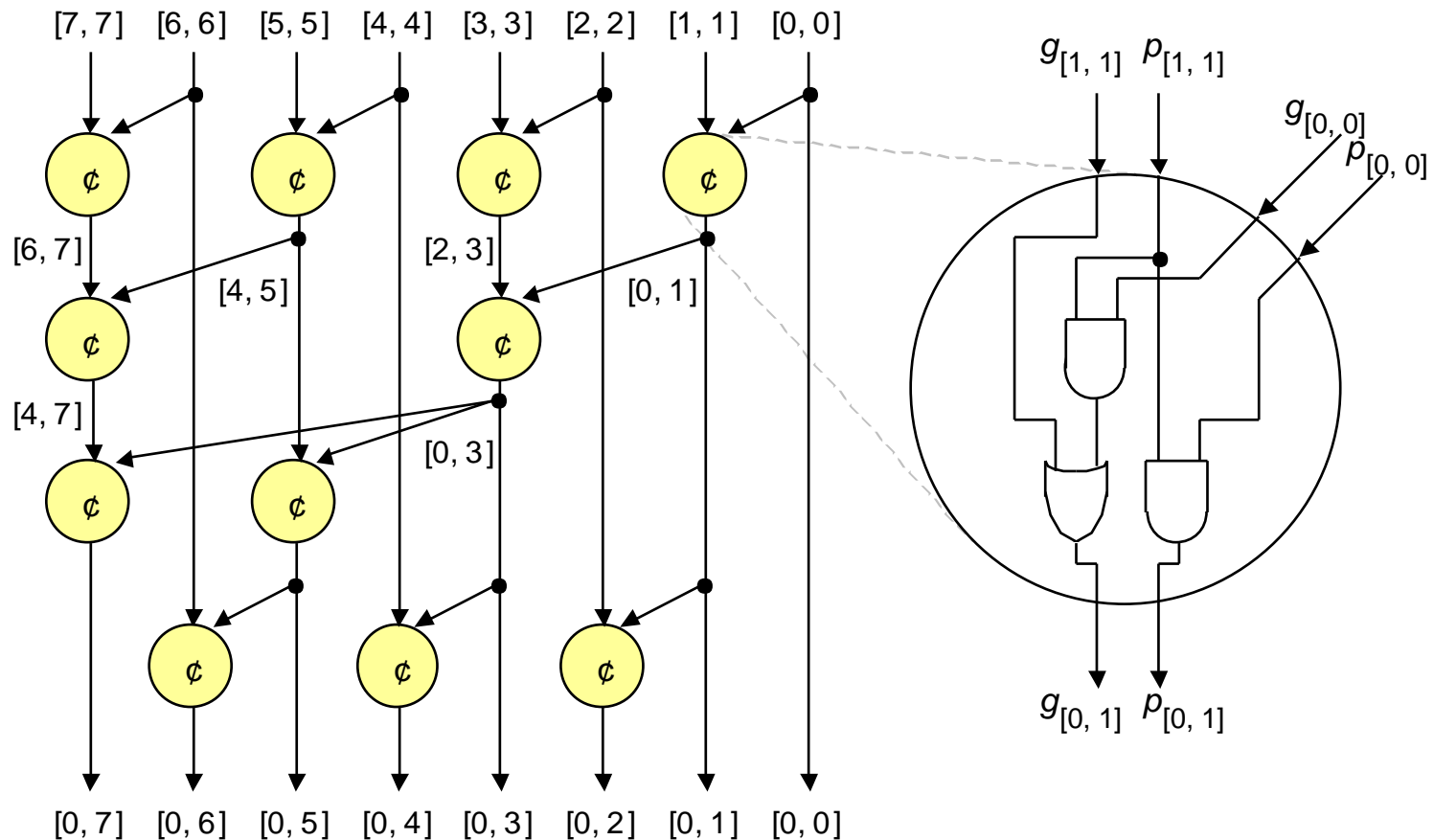


Figure 10.11 Brent-Kung lookahead carry network for an 8-digit adder, along with details of one of the carry operator blocks.

Recursive Structure of Brent-Kung Carry Network

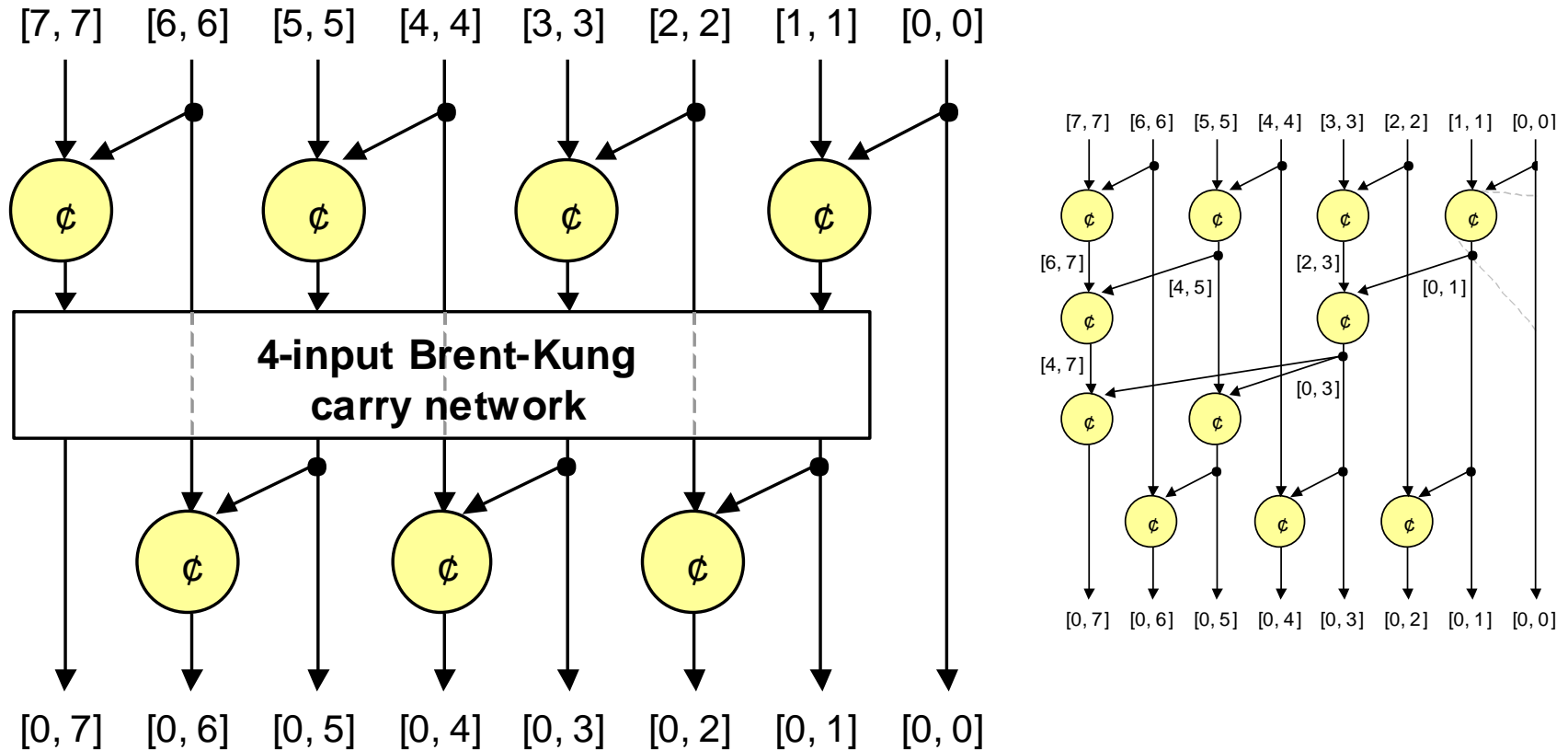
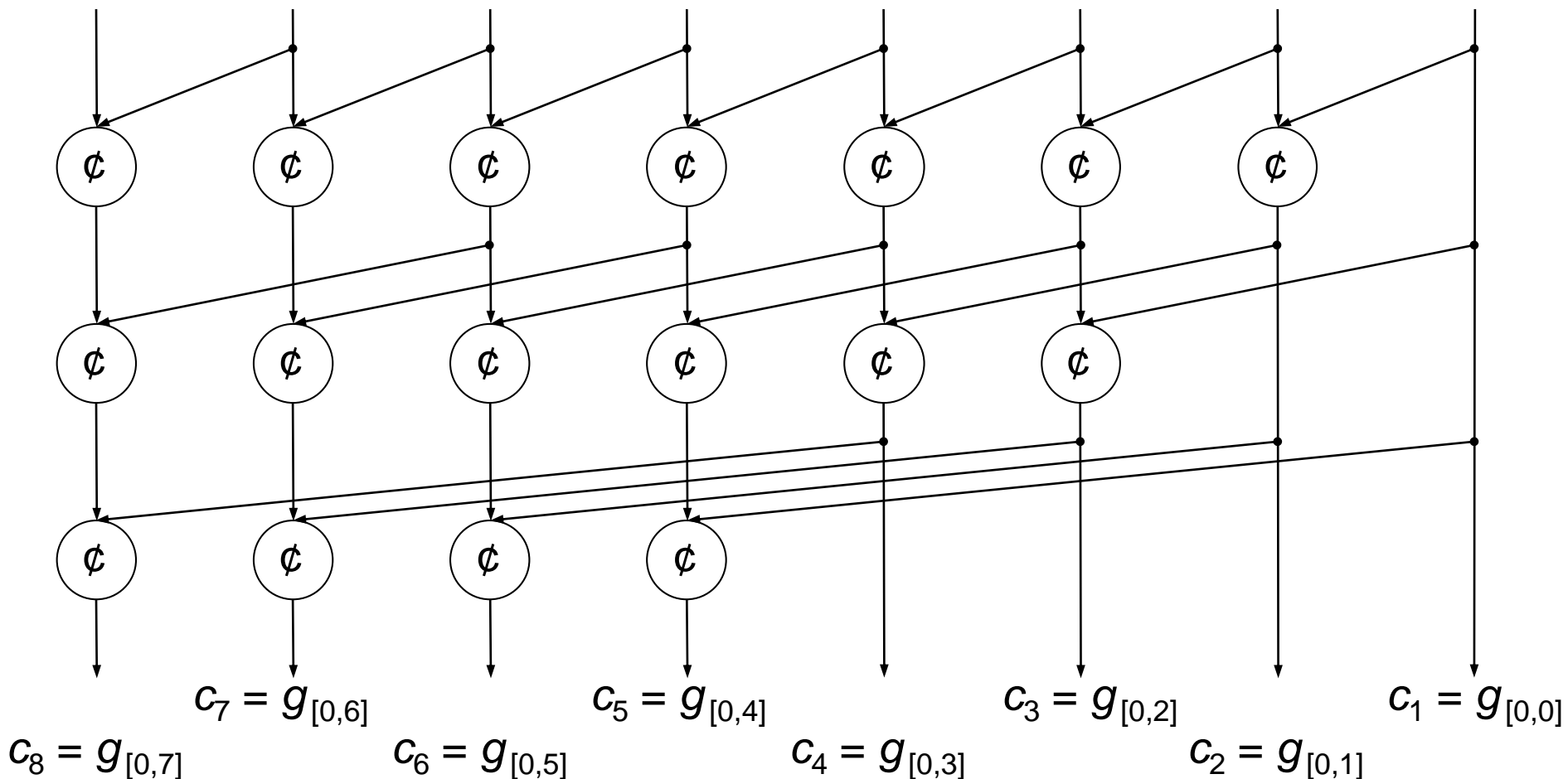


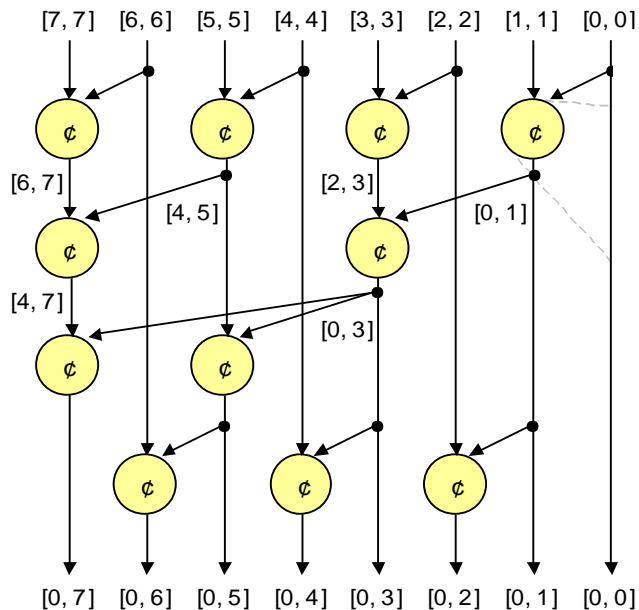
Figure 10.12 Brent-Kung lookahead carry network for an 8-digit adder, with only its top and bottom rows of carry-operators shown.

An Alternate Design: Kogge-Stone Network

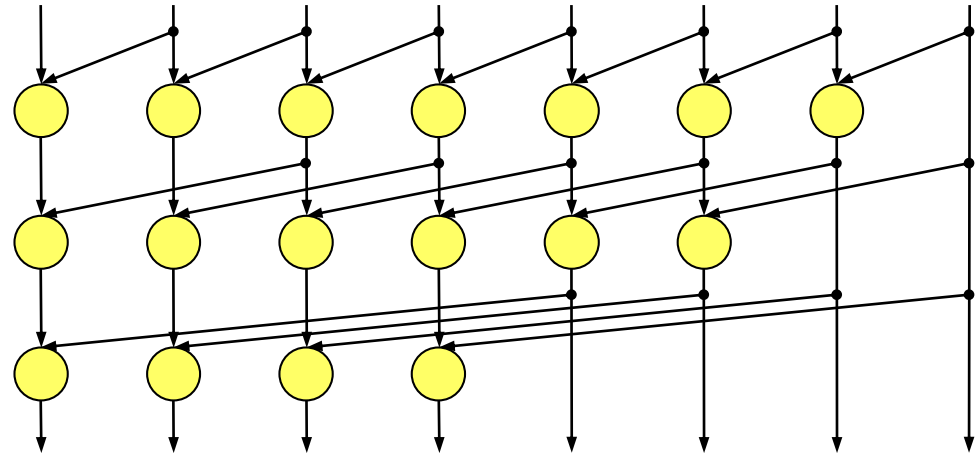


Kogge-Stone lookahead carry network for an 8-digit adder.

Brent-Kung vs. Kogge-Stone Carry Network



11 carry operators
4 levels



17 carry operators
3 levels

Carry-Lookahead Logic with 4-Bit Block

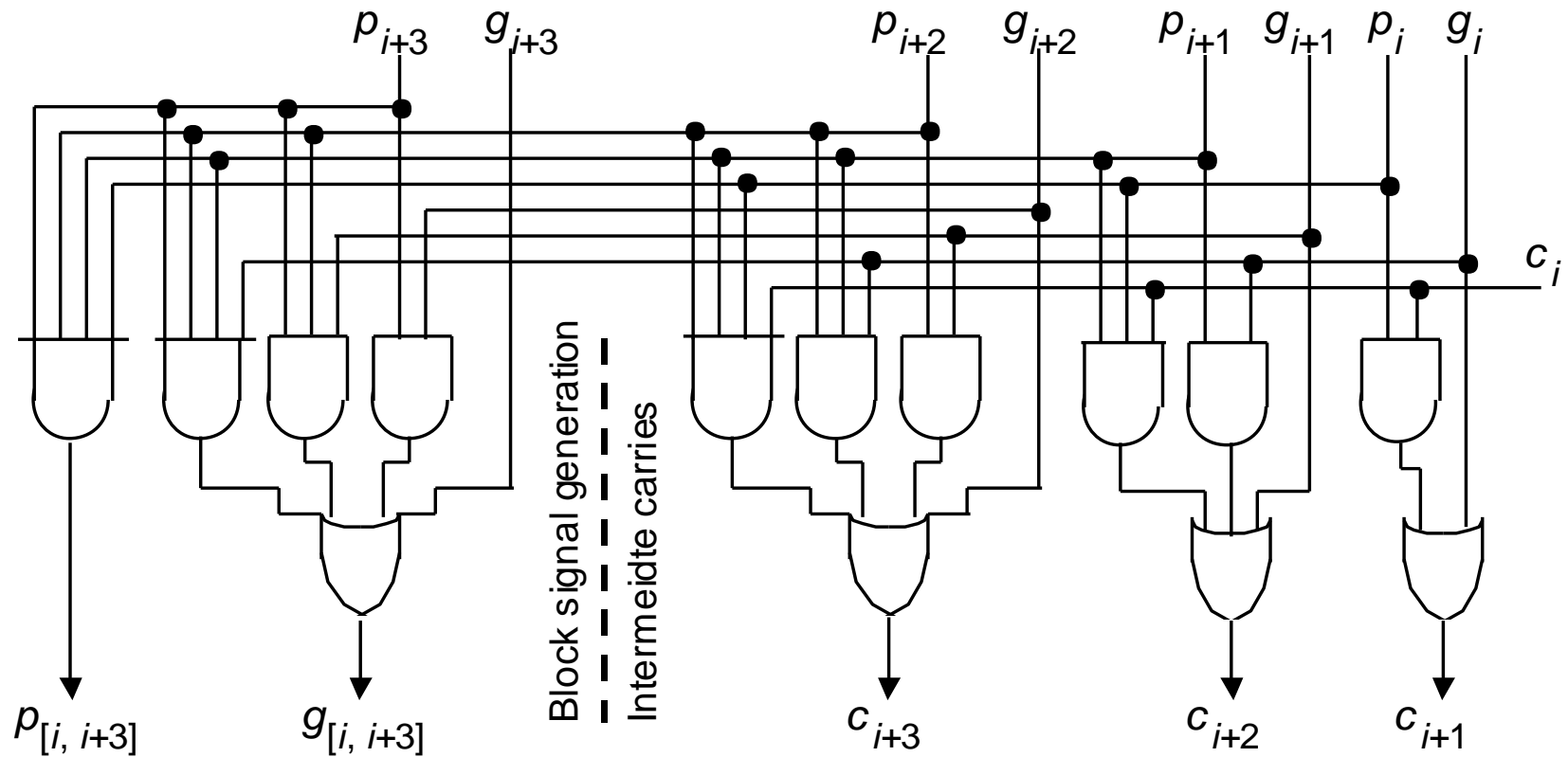
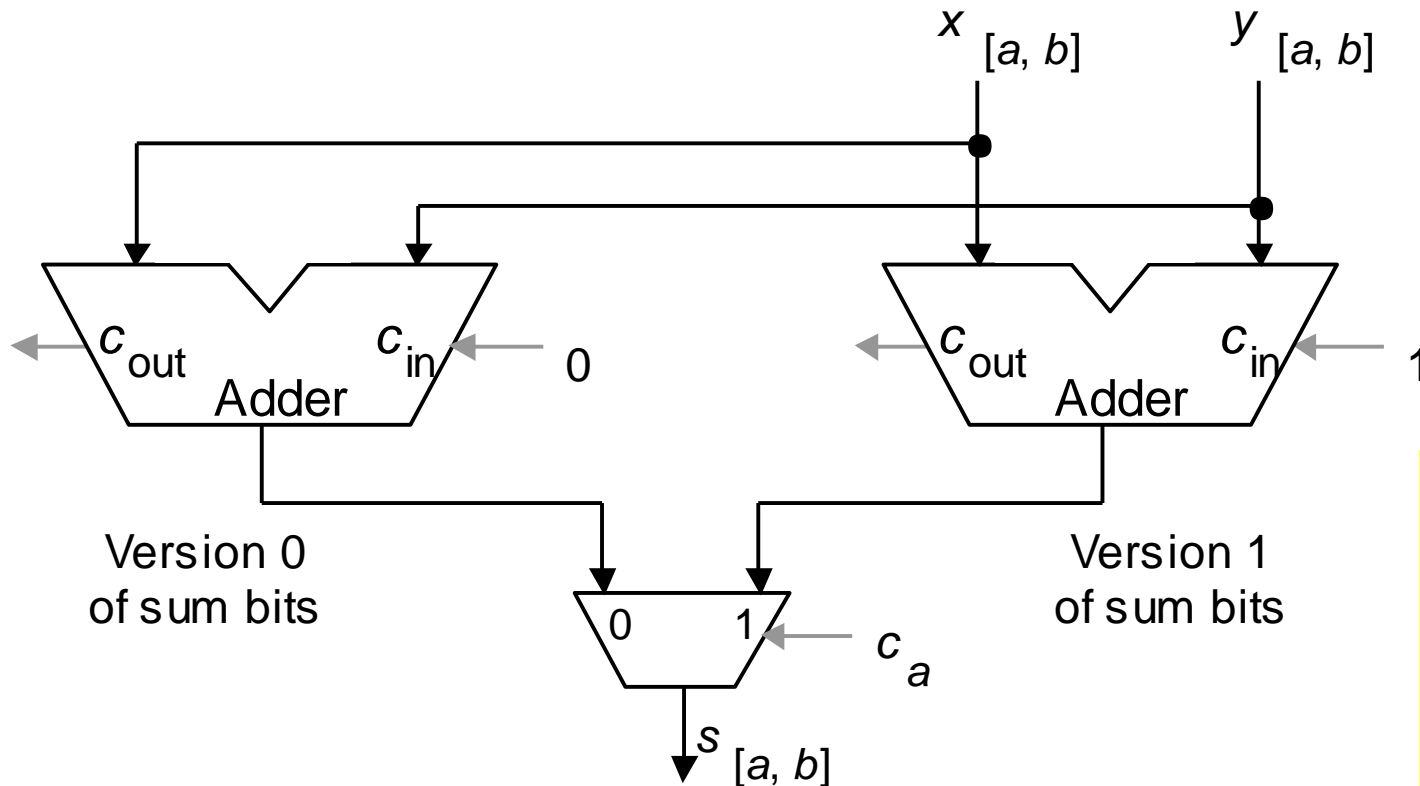


Figure 10.13 Blocks needed in the design of carry-lookahead adders with four-way grouping of bits.

Third Carry Speed-Up Method: Carry Select

Allows doubling of adder width with a single-mux additional delay



The lower a positions, (0 to $a - 1$) are added as usual

Figure 10.14 Carry-select addition principle.

10.5 Logic and Shift Operations

Conceptually, shifts can be implemented by multiplexing

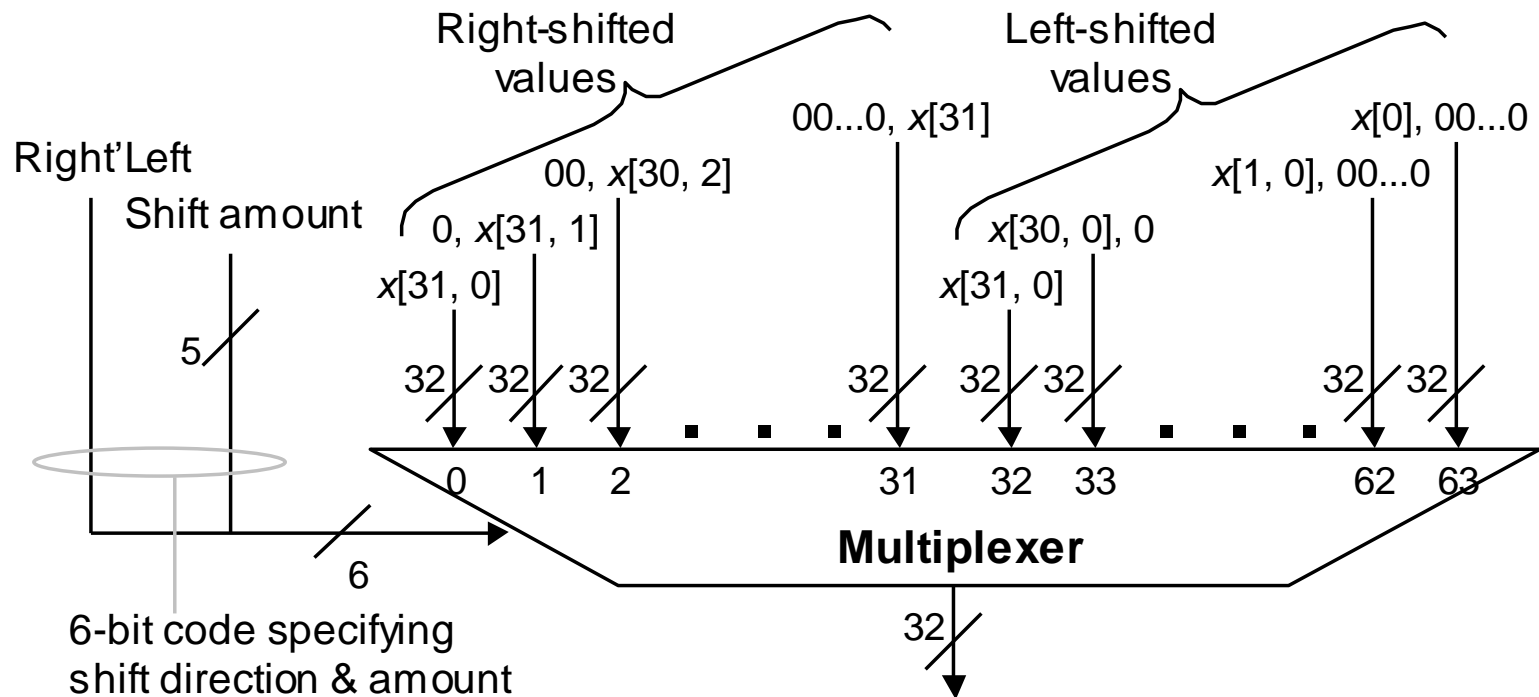


Figure 10.15 Multiplexer-based logical shifting unit.

Arithmetic Shifts

Purpose: Multiplication and division by powers of 2

```
sra  $t0,$s1,2      # $t0 ← ($s1) right-shifted by 2
sra  $t0,$s1,$s0     # $t0 ← ($s1) right-shifted by ($s0)
```

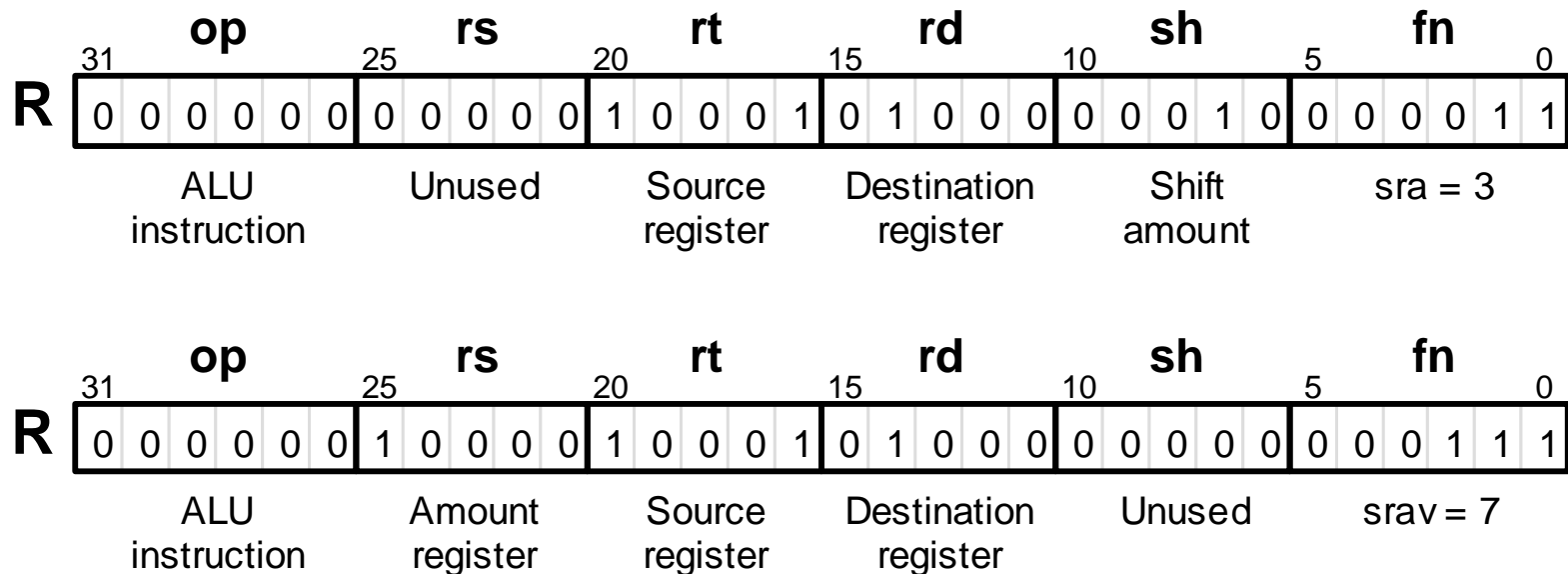
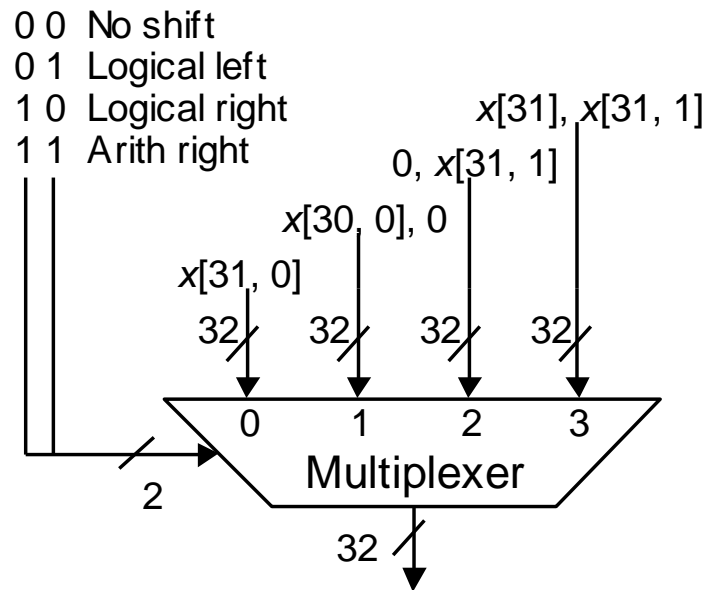
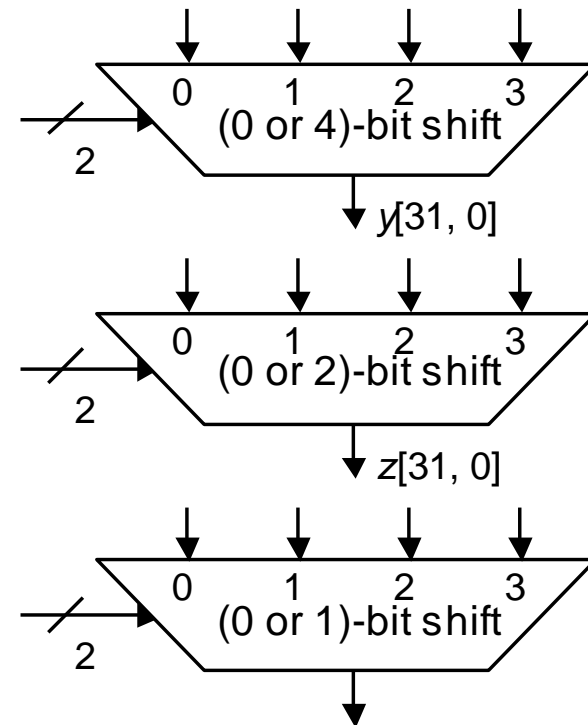


Figure 10.16 The two arithmetic shift instructions of MiniMIPS.

Practical Shifting in Multiple Stages



(a) Single-bit shifter



(b) Shifting by up to 7 bits

Figure 10.17 Multistage shifting in a barrel shifter.

Bit Manipulation via Shifts and Logical Operations

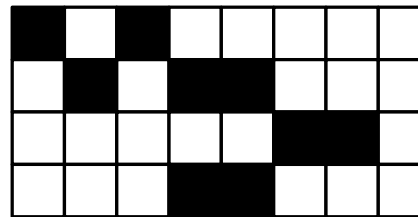
Bits 10-15

AND with mask to isolate a field: 0000 0000 0000 0000 1111 1100 0000 0000

Right-shift by 10 positions to move field to the right end of word

The result word ranges from 0 to 63, depending on the field pattern

32-pixel (4×8) block of
black-and-white image:



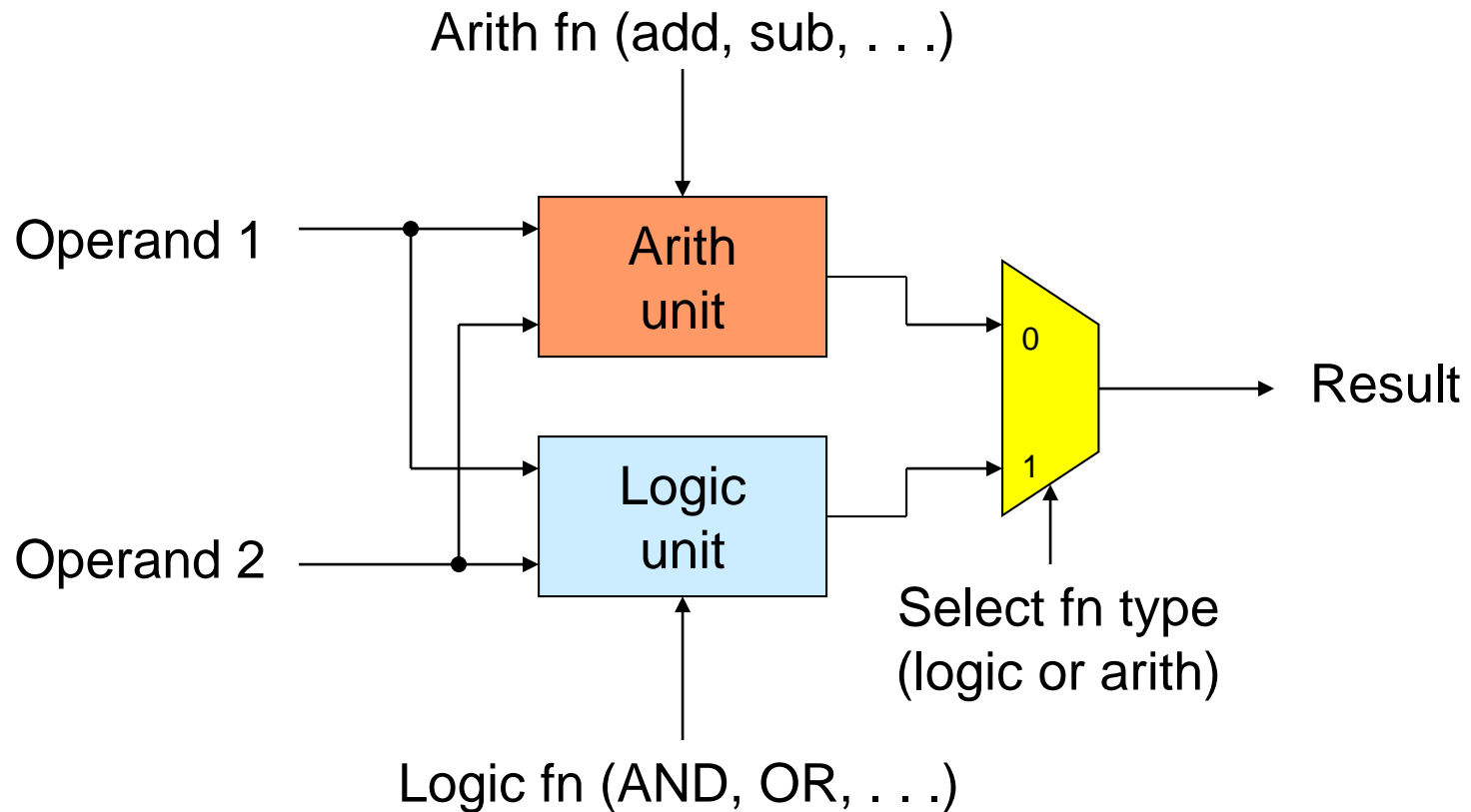
Representation as 32-bit word:

Row 0		Row 1		Row 2		Row 3	
1010	0000	0101	1000	0000	0110	0001	0111

Hex equivalent: 0xa0a80617

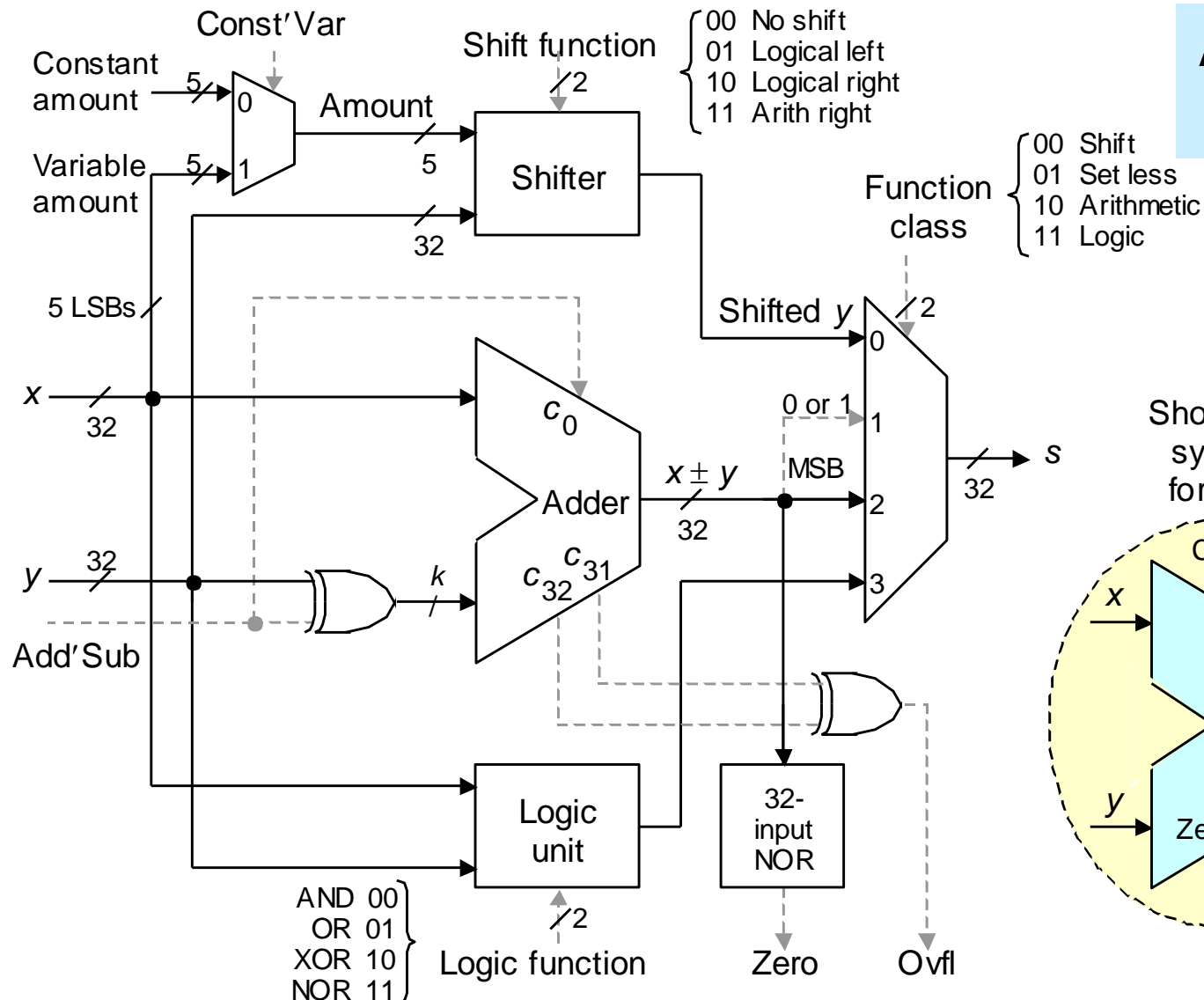
Figure 10.18 A 4×8 block of a black-and-white image represented as a 32-bit word.

10.6 Multifunction ALUs



General structure of a simple arithmetic/logic unit.

An ALU for MiniMIPS



Shorthand
symbol
for ALU

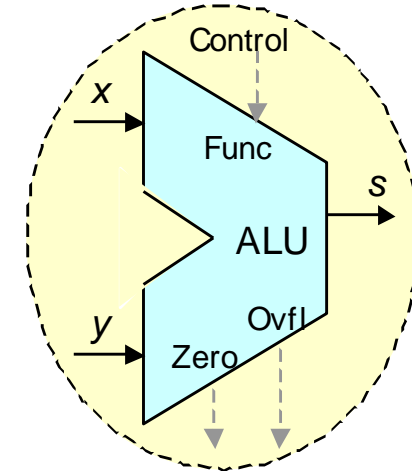


Figure 10.19 A multifunction ALU with 8 control signals (2 for function class, 1 arithmetic, 3 shift, 2 logic) specifying the operation.

11 Multipliers and Dividers

Modern processors perform many multiplications & divisions:

- Encryption, image compression, graphic rendering
- Hardware vs programmed shift-add/sub algorithms

Topics in This Chapter

11.1 Shift-Add Multiplication

11.2 Hardware Multipliers

11.3 Programmed Multiplication

11.4 Shift-Subtract Division

11.5 Hardware Dividers

11.6 Programmed Division

11.1 Shift-Add Multiplication

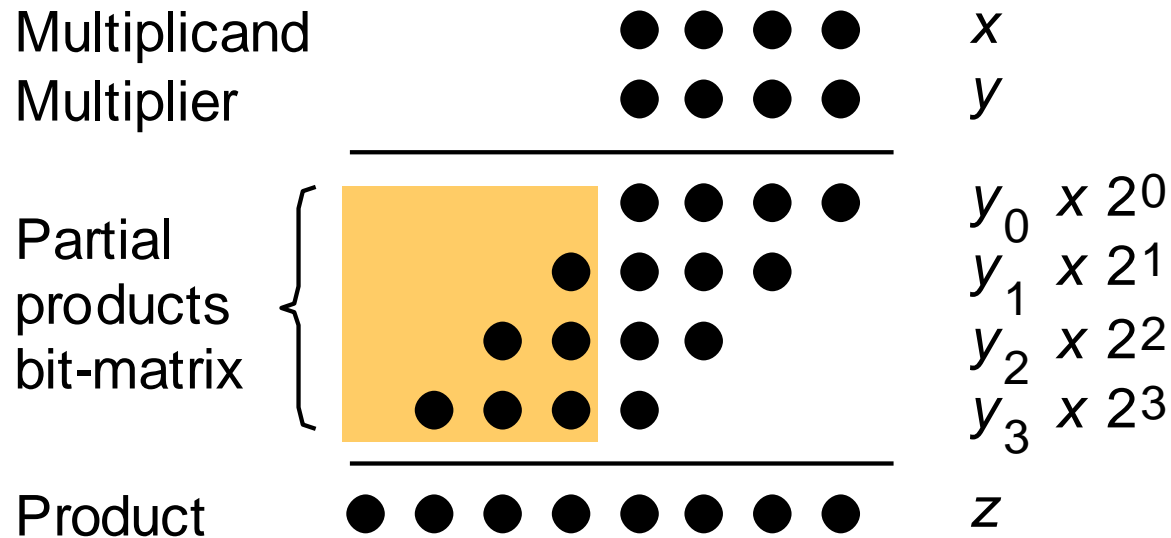


Figure 11.1 Multiplication of 4-bit numbers in dot notation.

$$z^{(j+1)} = \underbrace{(z^{(j)} + y_j \times 2^k)}_{\text{add}} \underbrace{2^{-1}}_{\text{shift right}} \quad \text{with } z^{(0)} = 0 \text{ and } z^{(k)} = z$$

Example 11.1

Position	7	6	5	4	3	2	1	0
$x10^4$	3	5	2	8				
y					4	0	6	7
$z^{(0)}$	0	0	0	0				
$+y_0x10^4$	2	4	6	9	6			
$10z^{(1)}$	2	4	6	9	6			
$z^{(1)}$	0	2	4	6	9	6		
$+y_1x10^4$	2	1	1	6	8			
$10z^{(2)}$	2	3	6	3	7	6		
$z^{(2)}$	2	3	6	3	7	6		
$+y_2x10^4$	0	0	0	0	0			
$10z^{(3)}$	0	2	3	6	3	7	6	
$z^{(3)}$	0	2	3	6	3	7	6	
$+y_3x10^4$	1	4	1	1	2			
$10z^{(4)}$	1	4	3	4	8	3	7	6
$z^{(4)}$	1	4	3	4	8	3	7	6

Figure 11.2 Step-by-step multiplication examples for 4-digit unsigned numbers.

Example 11.2

Position	7	6	5	4	3	2	1	0
$x2^4$	1	0	1	0				
y					1	0	1	1
$z^{(0)}$	0	0	0	0	0			
$+y_0x2^4$	1	1	0	1	0			
$2z^{(1)}$	1	1	0	1	0			
$z^{(1)}$	1	1	1	0	1	0		
$+y_1x2^4$	1	1	0	1	0			
$2z^{(2)}$	1	0	1	1	1	0		
$z^{(2)}$	1	1	0	1	1	1	0	
$+y_2x2^4$	0	0	0	0	0			
$2z^{(3)}$	1	1	0	1	1	1	0	
$z^{(3)}$	1	1	1	0	1	1	1	0
$+(-y_3x2^4)$	0	0	1	1	0			
$2z^{(4)}$	0	0	0	1	1	1	1	0
$z^{(4)}$	0	0	0	1	1	1	1	0

Figure 11.3 Step-by-step multiplication examples for 2's-complement numbers.

11.2 Hardware Multipliers

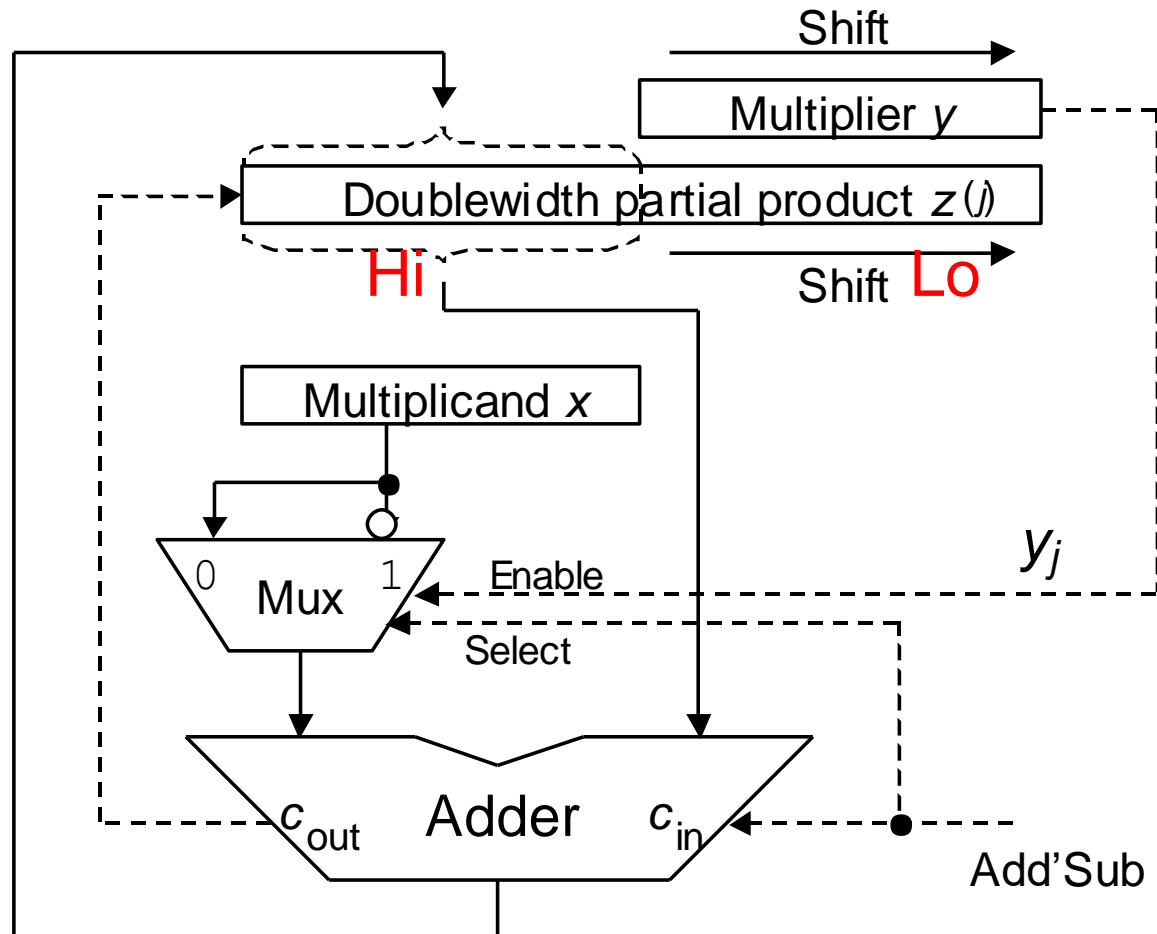


Figure 11.4 Hardware multiplier based on the shift-add algorithm.

The Shift Part of Shift-Add

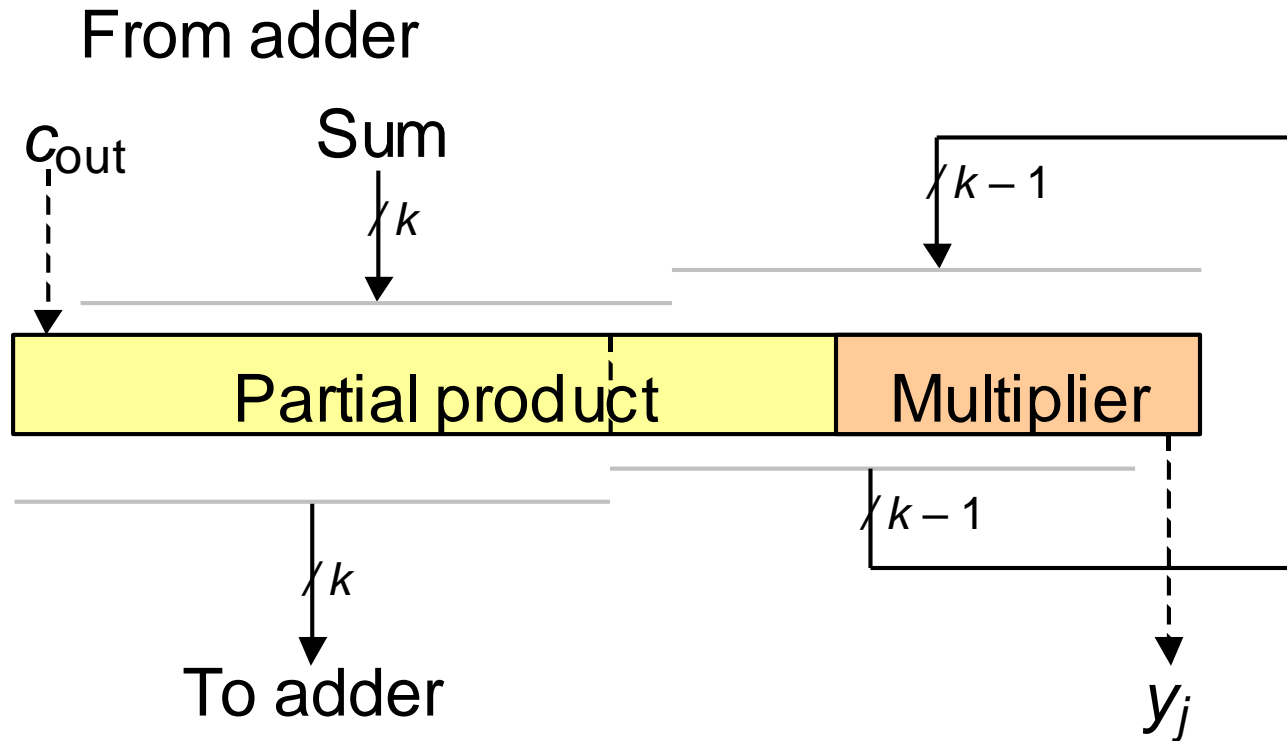
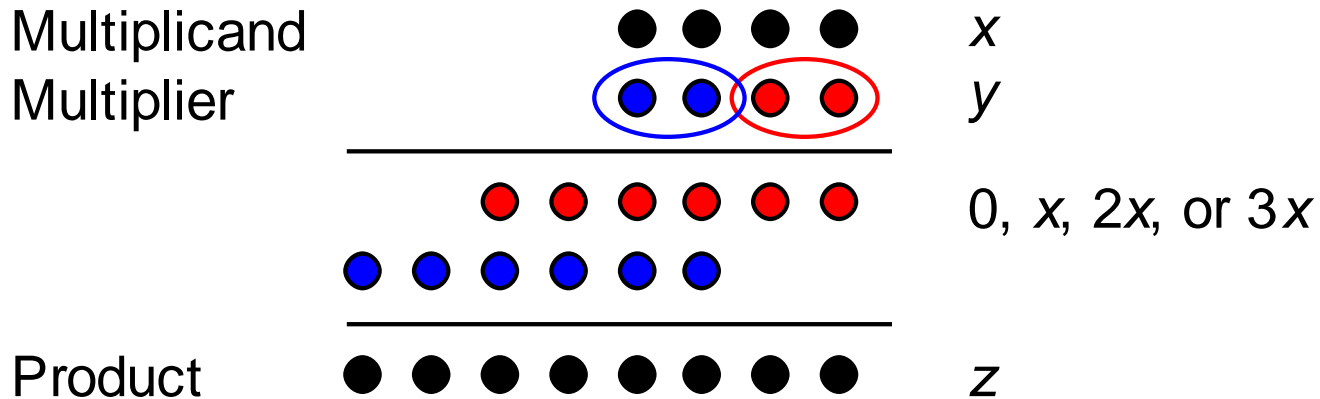


Figure 11.5 Shifting incorporated in the connections to the partial product register rather than as a separate phase.

High-Radix Multipliers



Radix-4 multiplication in dot notation.

$$z^{(j+1)} = \underbrace{(z^{(j)} + y_j x 2^k)}_{\text{add}} 4^{-1} \quad \text{with } z^{(0)} = 0 \text{ and } z^{(k/2)} = z$$

|— shift right —|

Assume k even

Tree Multipliers

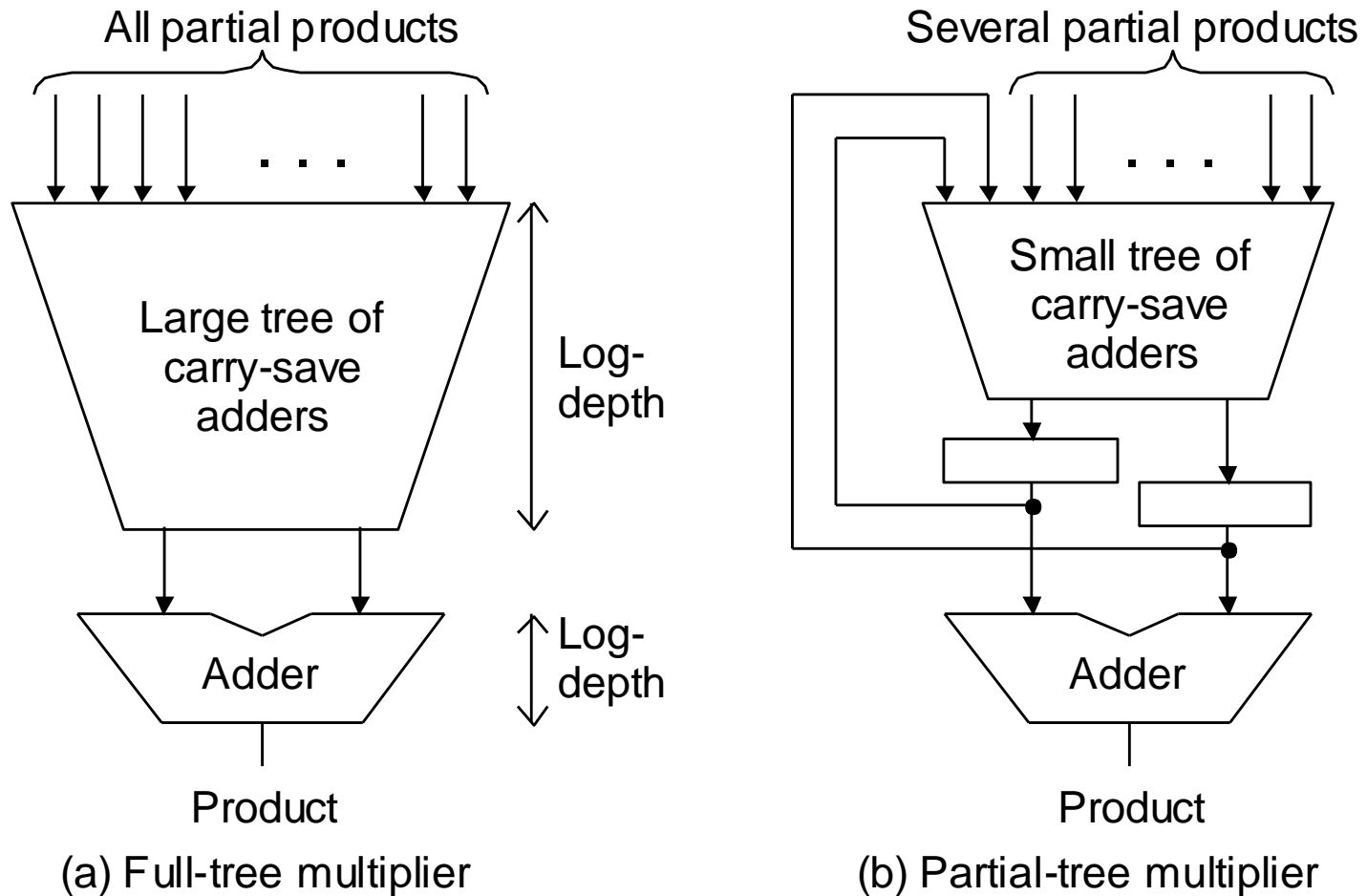


Figure 11.6 Schematic diagram for full/partial-tree multipliers.

Our original
dot-notation
representing
multiplication

Straightened
dots to depict
array multiplier
to the left



11.3 Programmed Multiplication

MiniMIPS instructions related to multiplication

```
mult    $s0,$s1    # set Hi,Lo to ($s0)×($s1); signed
multu   $s2,$s3    # set Hi,Lo to ($s2)×($s3); unsigned
mfhi    $t0         # set $t0 to (Hi)
mflo    $t1         # set $t1 to (Lo)
```

Example 11.3

Finding the 32-bit product of 32-bit integers in MiniMIPS

Multiply; result will be obtained in Hi, Lo

For unsigned multiplication:

Hi should be all-0s and Lo holds the 32-bit result

For signed multiplication:

Hi should be all-0s or all-1s, depending on the sign bit of Lo

Emulating a Hardware Multiplier in Software

Example 11.4 (MiniMIPS shift-add program for multiplication)

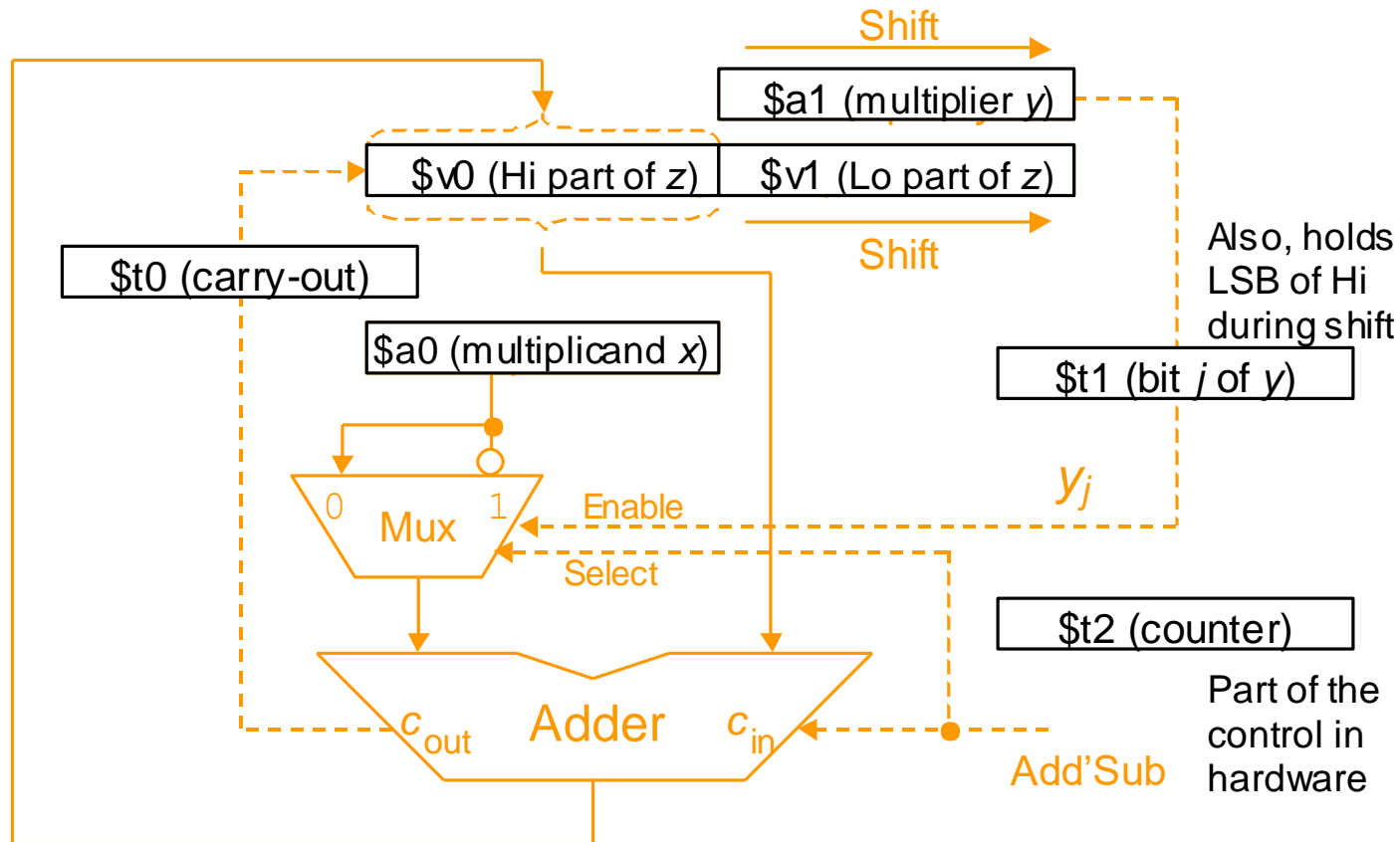



Figure 11.8 Register usage for programmed multiplication superimposed on the block diagram for a hardware multiplier.

Multiplication When There Is No Multiply Instruction

Example 11.4 (MiniMIPS shift-add program for multiplication)

```
shamu:  move  $v0,$zero      # initialize Hi to 0
        move  $v1,$zero      # initialize Lo to 0
        addi  $t2,$zero,32    # init repetition counter to 32
mloop:  move  $t0,$zero      # set c-out to 0 in case of no add
        move  $t1,$a1        # copy ($a1) into $t1
        srl   $a1,1          # halve the unsigned value in $a1
        subu  $t1,$t1,$a1     # subtract ($a1) from ($t1) twice to
        subu  $t1,$t1,$a1     # obtain LSB of ($a1), or y[j], in $t1
        beqz  $t1,noadd      # no addition needed if y[j] = 0
        addu  $v0,$v0,$a0     # add x to upper part of z
        sltu  $t0,$v0,$a0     # form carry-out of addition in $t0
noadd:  move  $t1,$v0        # copy ($v0) into $t1
        srl   $v0,1          # halve the unsigned value in $v0
        subu  $t1,$t1,$v0     # subtract ($v0) from ($t1) twice to
        subu  $t1,$t1,$v0     # obtain LSB of Hi in $t1
        sll   $t0,$t0,31      # carry-out converted to 1 in MSB of $t0
        addu  $v0,$v0,$t0     # right-shifted $v0 corrected
        srl   $v1,1          # halve the unsigned value in $v1
        sll   $t1,$t1,31      # LSB of Hi converted to 1 in MSB of $t1
        addu  $v1,$v1,$t1     # right-shifted $v1 corrected
        addi  $t2,$t2,-1      # decrement repetition counter by 1
        bne   $t2,$zero,mloop # if counter > 0, repeat multiply loop
        jr    $ra            # return to the calling program
```



A control flow diagram is overlaid on the code. A red arrow starts at the 'mloop:' label, goes down the left margin, and then curves back up to point at the 'mloop:' label. A green arrow starts at the 'beqz \$t1,noadd' instruction and points to the 'noadd:' label. Another green arrow starts at the 'jr \$ra' instruction and points back to the 'mloop:' label, completing the loop.

11.4 Shift-Subtract Division

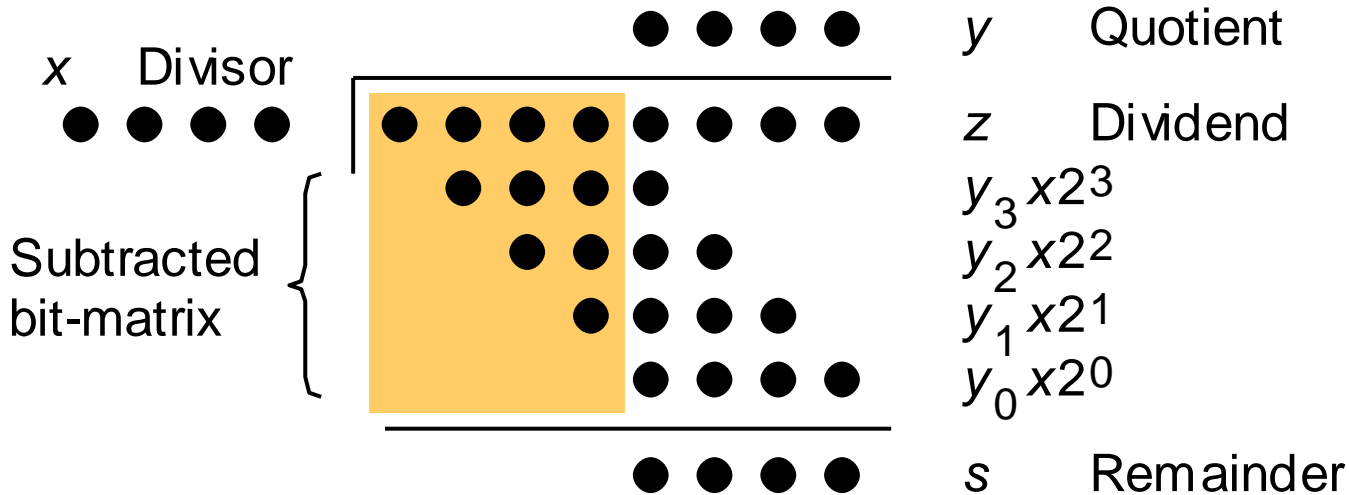


Figure 11.9 Division of an 8-bit number by a 4-bit number in dot notation.

$$z^{(j)} = 2z^{(j-1)} - y_{k-j} x 2^k \quad \text{with } z^{(0)} = z \text{ and } z^{(k)} = 2^k s$$

| shift |

| — subtract — |

Integer and Fractional Unsigned Division

Example 11.5

Position	7	6	5	4	3	2	1	0
=====								
z	0	1	1	1	0	1	0	1
$x2^4$	1	0	1	0				
=====								
$z^{(0)}$	0	1	1	1	0	1	0	1
$2z^{(0)}$	0	1	1	1	0	1	0	1
$-y_3x2^4$	1	0	1	0				
								$y_3=1$
=====								
$z^{(1)}$	0	1	0	0	1	0	1	
$2z^{(1)}$	0	1	0	0	1	0	1	
$-y_2x2^4$	0	0	0	0				
								$y_2=0$
=====								
$z^{(2)}$	1	0	0	1	0	1		
$2z^{(2)}$	1	0	0	1	0	1		
$-y_1x2^4$	1	0	1	0				
								$y_1=1$
=====								
$z^{(3)}$	1	0	0	0	1			
$2z^{(3)}$	1	0	0	0	1			
$-y_0x2^4$	1	0	1	0				
								$y_0=1$
=====								
$z^{(4)}$	0	1	1	1				
s					0	1	1	1
y					1	0	1	1
=====								

Position	-1	-2	-3	-4	-5	-6	-7	-8
=====								
z	.1	4	3	5	1	5	0	2
x	.4	0	6	7				
=====								
$z^{(0)}$.1	4	3	5	1	5	0	2
$10z^{(0)}$	1.4	3	5	1	5	0	2	
$-y_{-1}x$	1.2	2	0	1				
								$y_{-1}=3$
=====								
$z^{(1)}$.2	1	5	0	5	0	2	
$10z^{(1)}$	2.1	5	0	5	0	2		
$-y_{-2}x$	2.0	3	3	5				
								$y_{-2}=5$
=====								
$z^{(2)}$.1	1	7	0	0	2		
$10z^{(2)}$	1.1	7	0	0	2			
$-y_{-3}x$	0.8	1	3	4				
								$y_{-3}=2$
=====								
$z^{(3)}$.3	5	6	6	2			
$10z^{(3)}$	3.5	6	6	2				
$-y_{-4}x$	3.2	5	3	6				
								$y_{-4}=8$
=====								
$z^{(4)}$.3	1	2	6				
s	.0	0	0	0	3	1	2	6
y	.3	5	2	8				
=====								

Figure 11.10 Division examples for binary integers and decimal fractions.

Example 11.6

Position	-1	-2	-3	-4	-5	-6	-7	-8
=====								
z	.0	1	0	1				
x	.1	1	0	1				
=====								
z ⁽⁰⁾	.0	1	0	1				
2z ⁽⁰⁾	0.	1	0	1	0			
-y ₋₁ x	0.	0	0	0	0			y ₋₁ =0
=====								
z ⁽¹⁾	.1	0	1	0				
2z ⁽¹⁾	1.	0	1	0	0			
-y ₋₂ x	0.	1	1	0	1			y ₋₂ =1
=====								
z ⁽²⁾	.0	1	1	1				
2z ⁽²⁾	0.	1	1	1	0			
-y ₋₃ x	0.	1	1	0	1			y ₋₃ =1
=====								
z ⁽³⁾	.0	0	0	0	1			
2z ⁽³⁾	0.	0	0	1	0			
-y ₋₄ x	0.	0	0	0	0			y ₋₄ =0
=====								
z ⁽⁴⁾	.0	0	1	0				
s	.0	0	0	0	0	0	0	1
y	.0	1	1	0				
=====								

Figure 11.11 Division examples for 4/4-digit binary integers and fractions.

Signed Division

Method 1 (indirect): strip operand signs, divide, set result signs

<u>Dividend</u>	<u>Divisor</u>		<u>Quotient</u>	<u>Remainder</u>
$z = 5$	$x = 3$	\Rightarrow	$y = 1$	$s = 2$
$z = 5$	$x = -3$	\Rightarrow	$y = -1$	$s = 2$
$z = -5$	$x = 3$	\Rightarrow	$y = -1$	$s = -2$
$z = -5$	$x = -3$	\Rightarrow	$y = 1$	$s = -2$

Method 2 (direct 2's complement): develop quotient with digits -1 and 1 , chosen based on signs, convert to digits 0 and 1

Restoring division: perform trial subtraction, choose 0 for q digit if partial remainder negative

Nonrestoring division: if sign of partial remainder is correct, then subtract (choose 1 for q digit) else add (choose -1)

11.5 Hardware Dividers

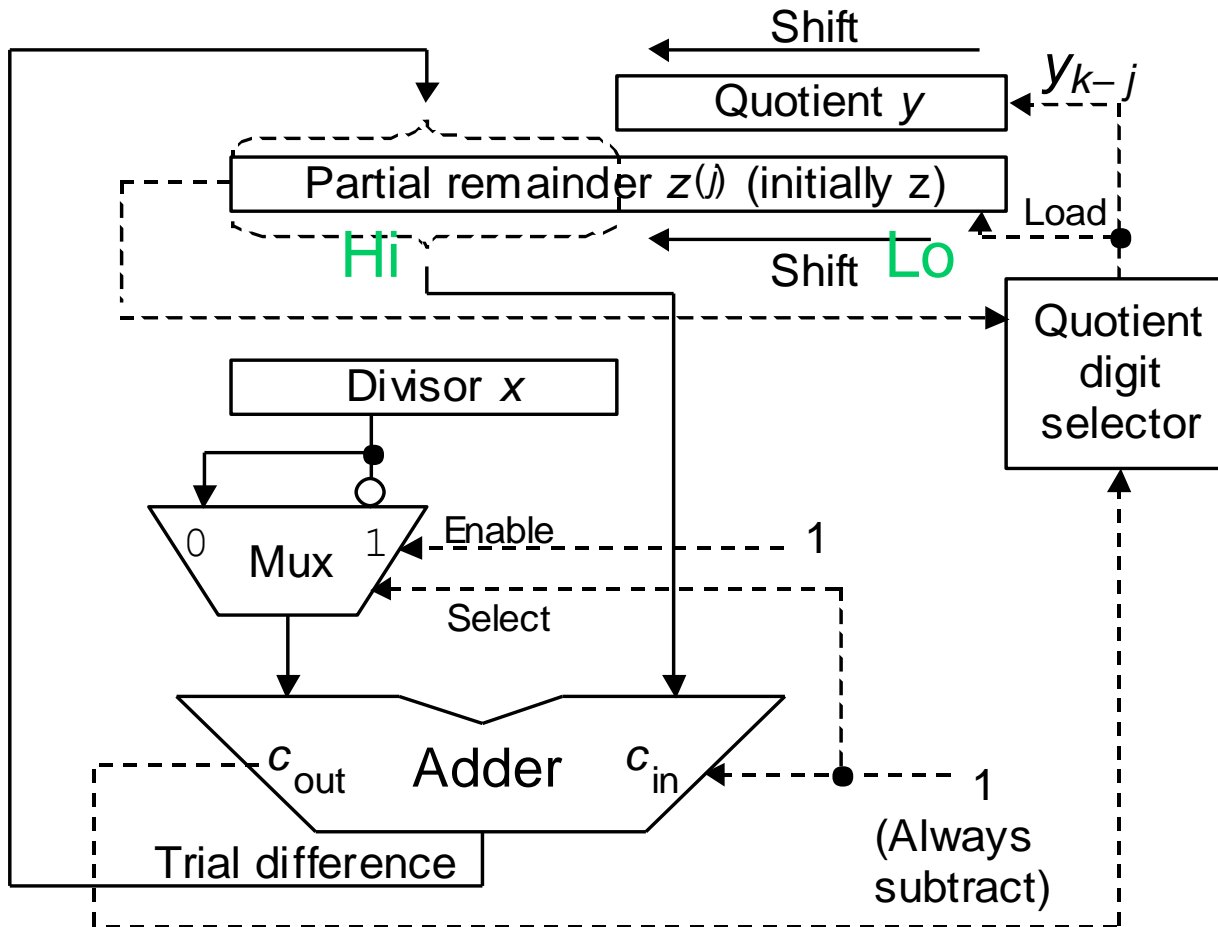


Figure 11.12 Hardware divider based on the shift-subtract algorithm.

The Shift Part of Shift-Subtract

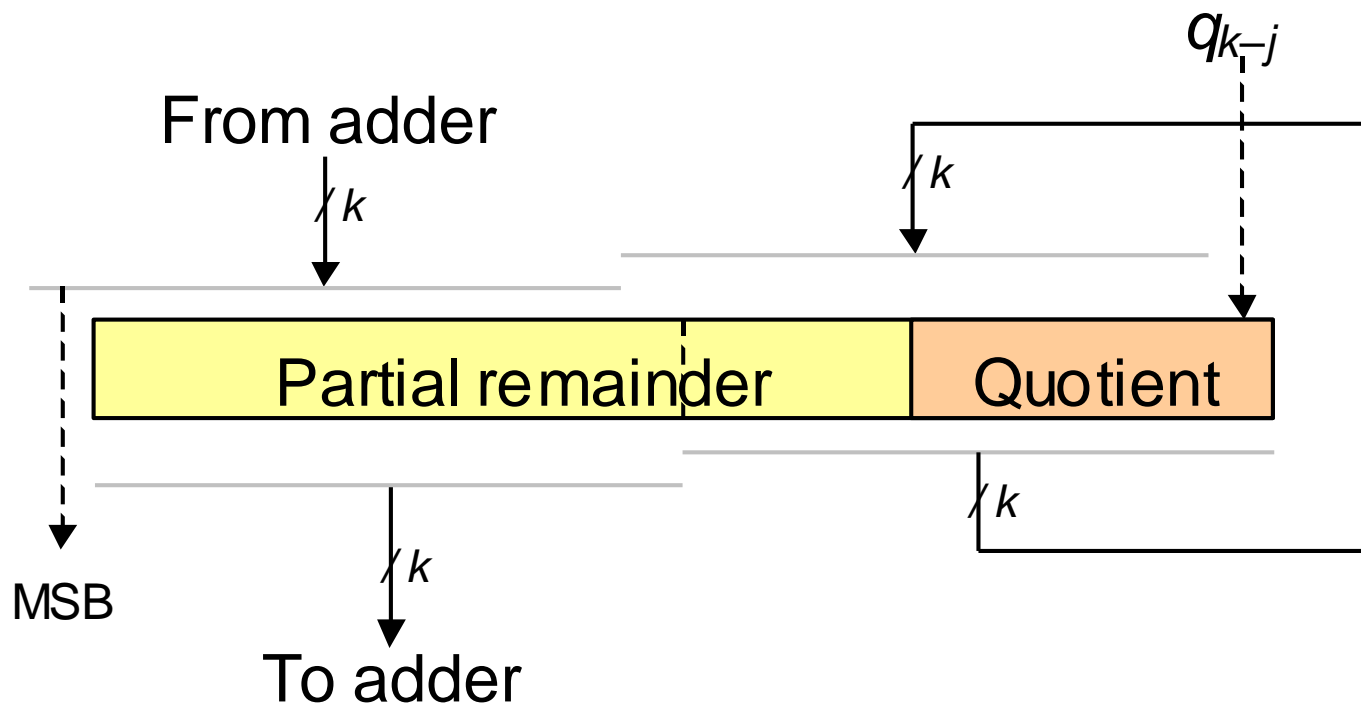
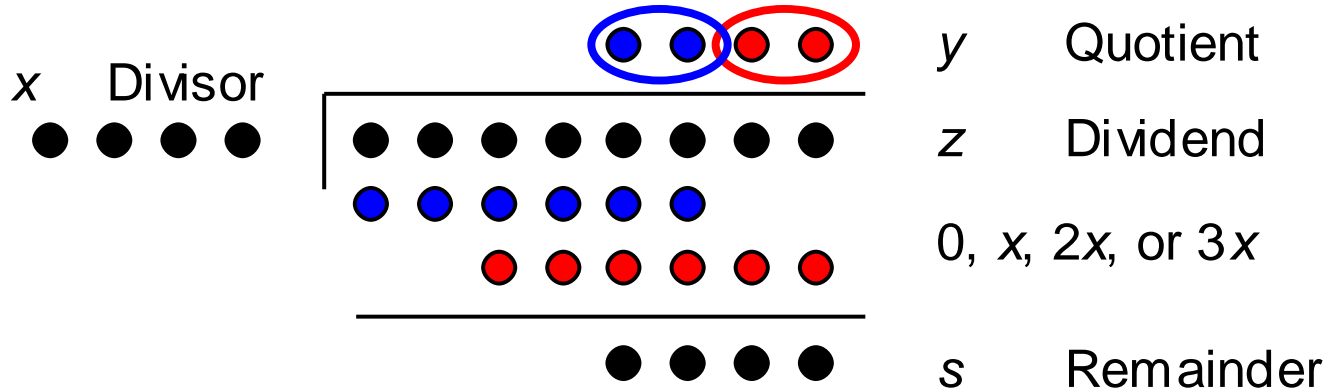


Figure 11.13 Shifting incorporated in the connections to the partial remainder register rather than as a separate phase.

High-Radix Dividers



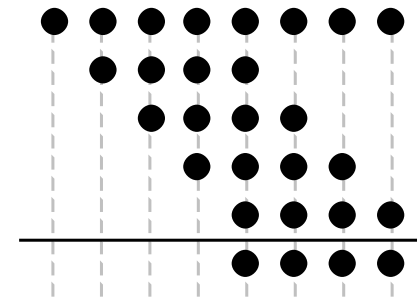
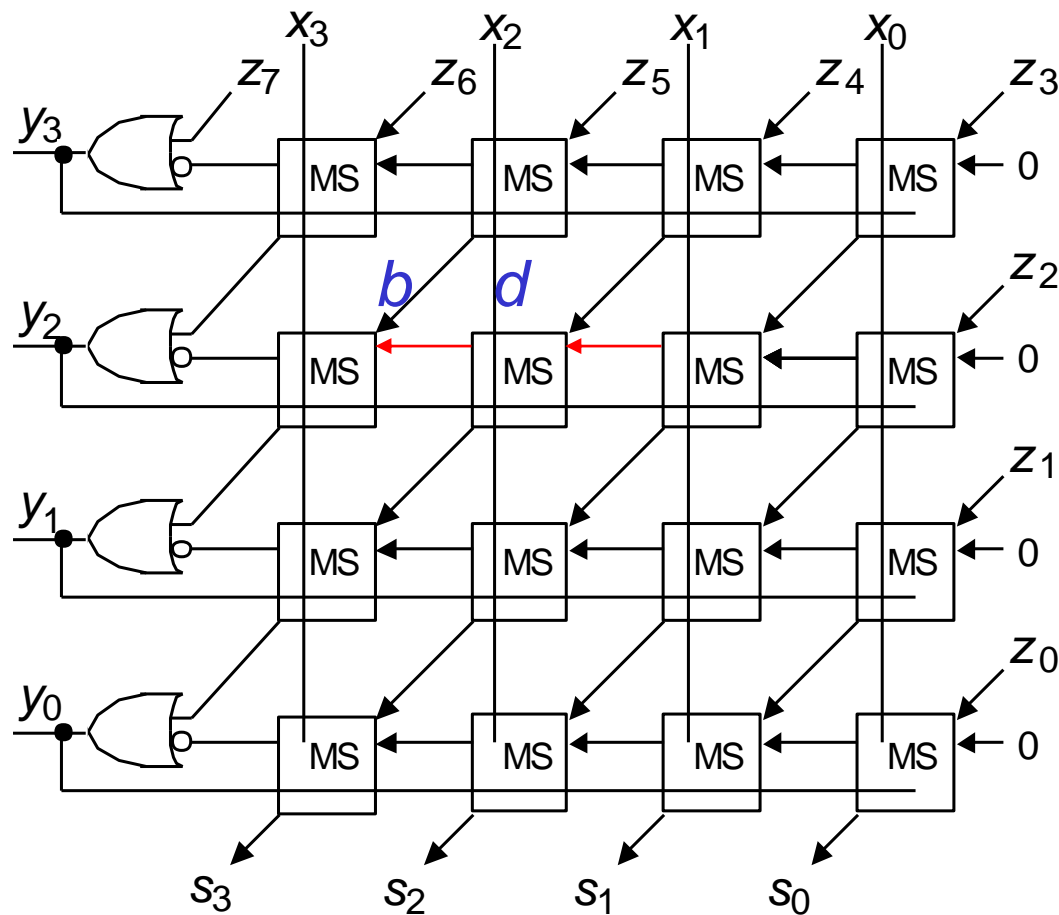
Radix-4 division in dot notation.

$$z^{(j)} = 4z^{(j-1)} - (y_{k-2j+1} \ y_{k-2j})_{\text{two}} x \ 2^k \quad \text{with } z^{(0)} = z \text{ and } z^{(k/2)} = 2^k s$$

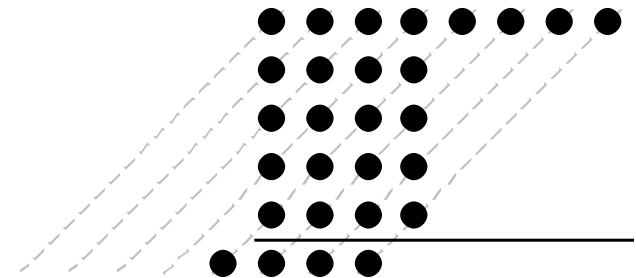
|shift|
 |————— subtract —————|

Assume k even

Array Dividers



Our original
dot-notation
for division



Straightened
dots to depict
an array divider

Figure 11.14 Array divider for 8/4-bit unsigned integers.

11.6 Programmed Division

MiniMIPS instructions related to division

```
div    $s0,$s1    # Lo = quotient, Hi = remainder
divu   $s2,$s3    # unsigned version of division
mfhi   $t0        # set $t0 to (Hi)
mflo   $t1        # set $t1 to (Lo)
```

Example 11.7

Compute $z \bmod x$, where z (signed) and $x > 0$ are integers

Divide; remainder will be obtained in H_i

```

if remainder is negative,
then add |x| to (Hi) to obtain z mod x
else Hi holds z mod x

```


Emulating a Hardware Divider in Software

Example 11.8 (MiniMIPS shift-add program for division)

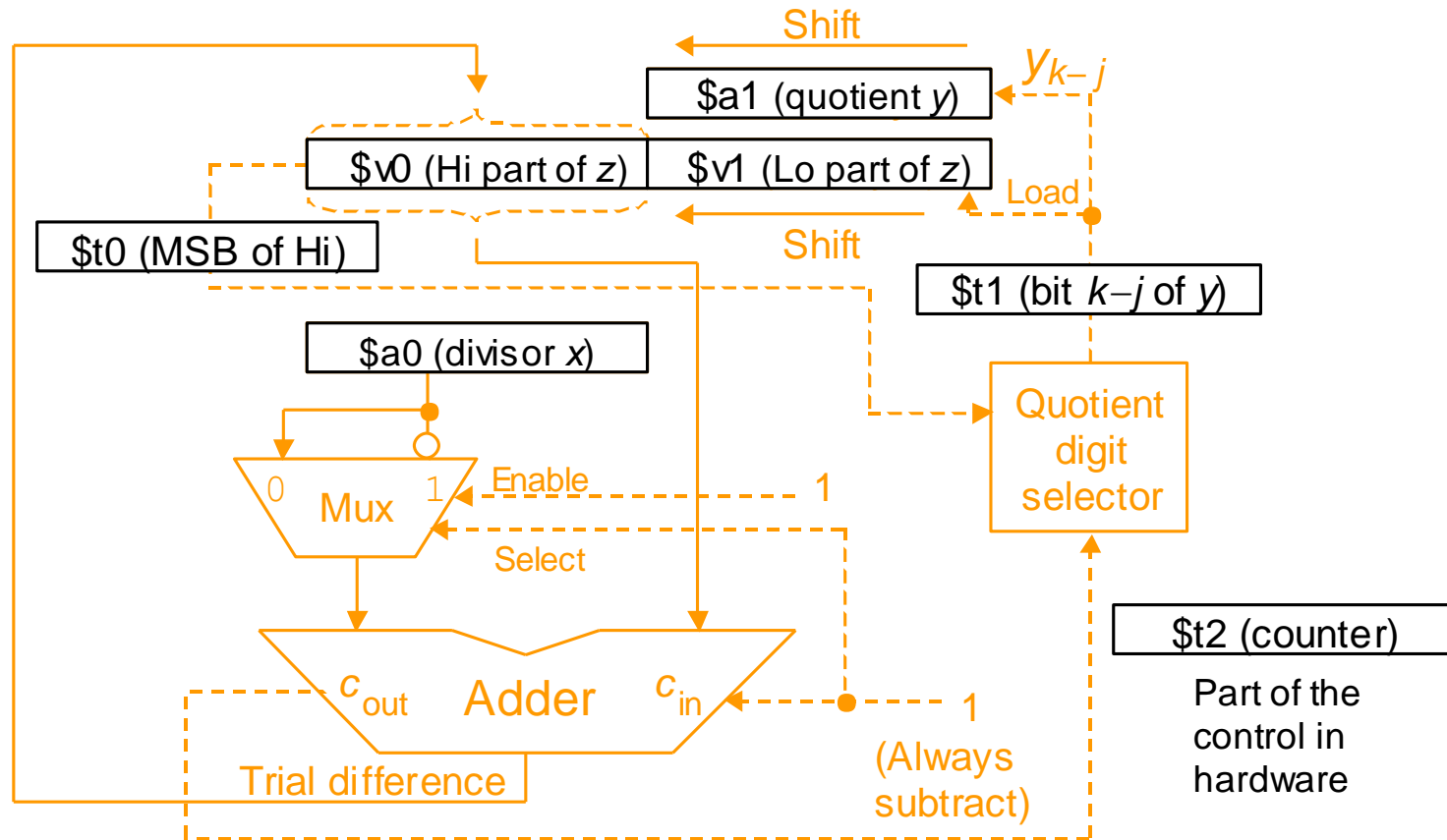



Figure 11.15 Register usage for programmed division superimposed on the block diagram for a hardware divider.

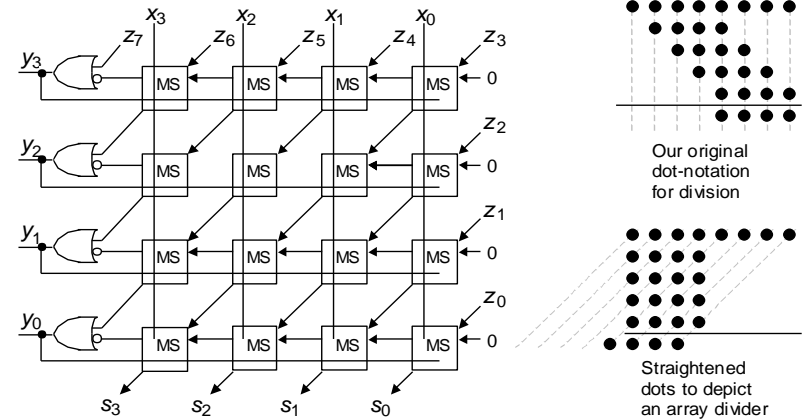
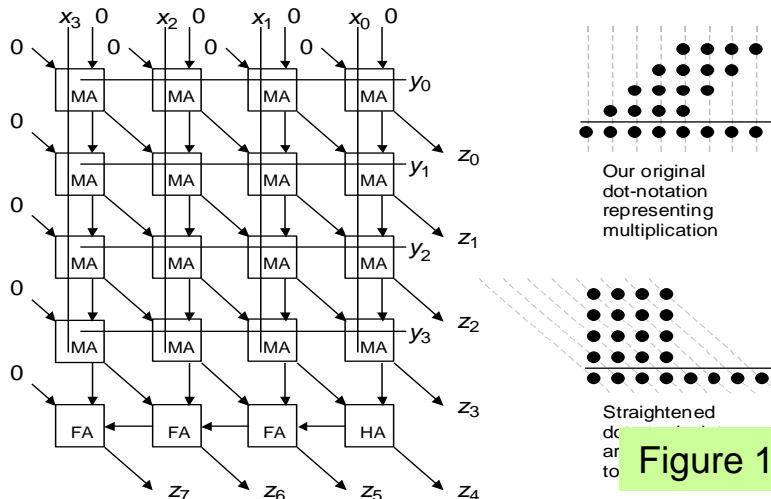
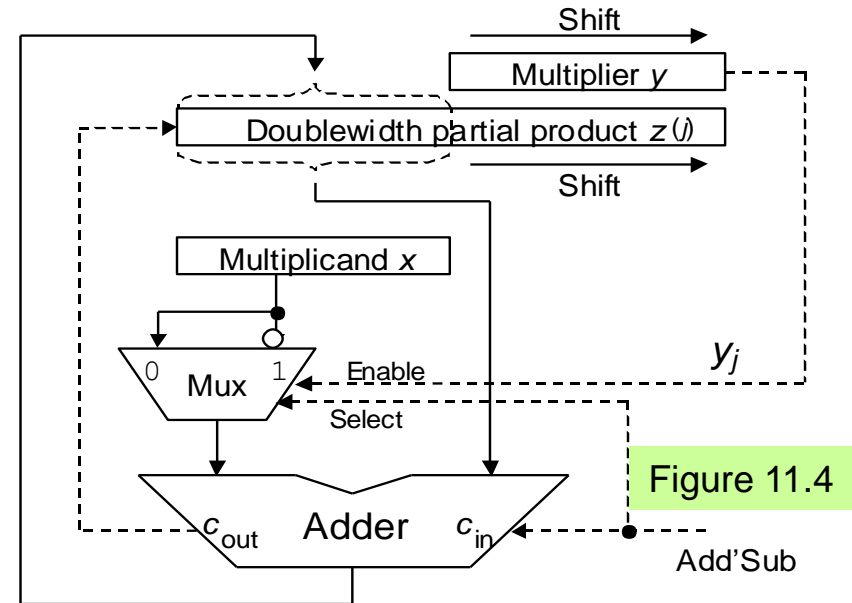
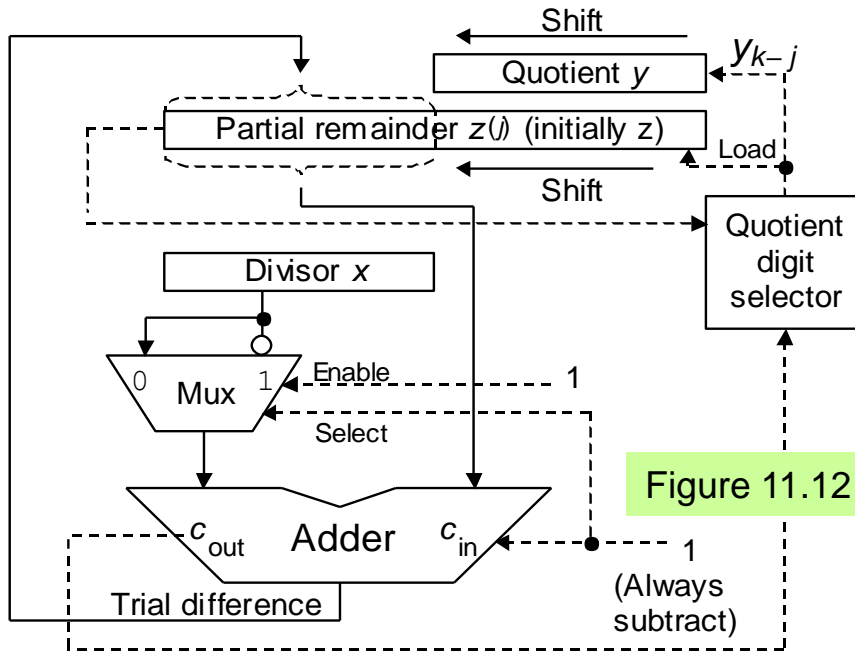
Division When There Is No Divide Instruction

Example 11.7 (MiniMIPS shift-subtract program for division)

```
shsdi:  move $v0,$a2          # initialize Hi to ($a2)
        move $v1,$a3          # initialize Lo to ($a3)
        addi $t2,$zero,32     # initialize repetition counter to 32
dloop:  slt  $t0,$v0,$zero     # copy MSB of Hi into $t0
        sll  $v0,$v0,1        # left-shift the Hi part of z
        slt  $t1,$v1,$zero     # copy MSB of Lo into $t1
        or   $v0,$v0,$t1      # move MSB of Lo into LSB of Hi
        sll  $v1,$v1,1        # left-shift the Lo part of z
        sge  $t1,$v0,$a0      # quotient digit is 1 if (Hi) ≥ x,
        or   $t1,$t1,$t0      # or if MSB of Hi was 1 before shifting
        sll  $a1,$a1,1        # shift y to make room for new digit
        or   $a1,$a1,$t1      # copy y[k-j] into LSB of $a1
        beq  $t1,$zero,nosub   # if y[k-j] = 0, do not subtract
        subu $v0,$v0,$a0      # subtract divisor x from Hi part of z
nosub:  addi $t2,$t2,-1        # decrement repetition counter by 1
        bne  $t2,$zero,dloop   # if counter > 0, repeat divide loop
        move $v1,$a1          # copy the quotient y into $v1
        jr   $ra              # return to the calling program
```



Divider vs Multiplier: Hardware Similarities



Turn upside-down

12 Floating-Point Arithmetic

Floating-point is no longer reserved for high-end machines

- Multimedia and signal processing require flp arithmetic
- Details of standard flp format and arithmetic operations

Topics in This Chapter

12.1 Rounding Modes

12.2 Special Values and Exceptions

12.3 Floating-Point Addition

12.4 Other Floating-Point Operations

12.5 Floating-Point Instructions

12.6 Result Precision and Errors

12.1 Rounding Modes

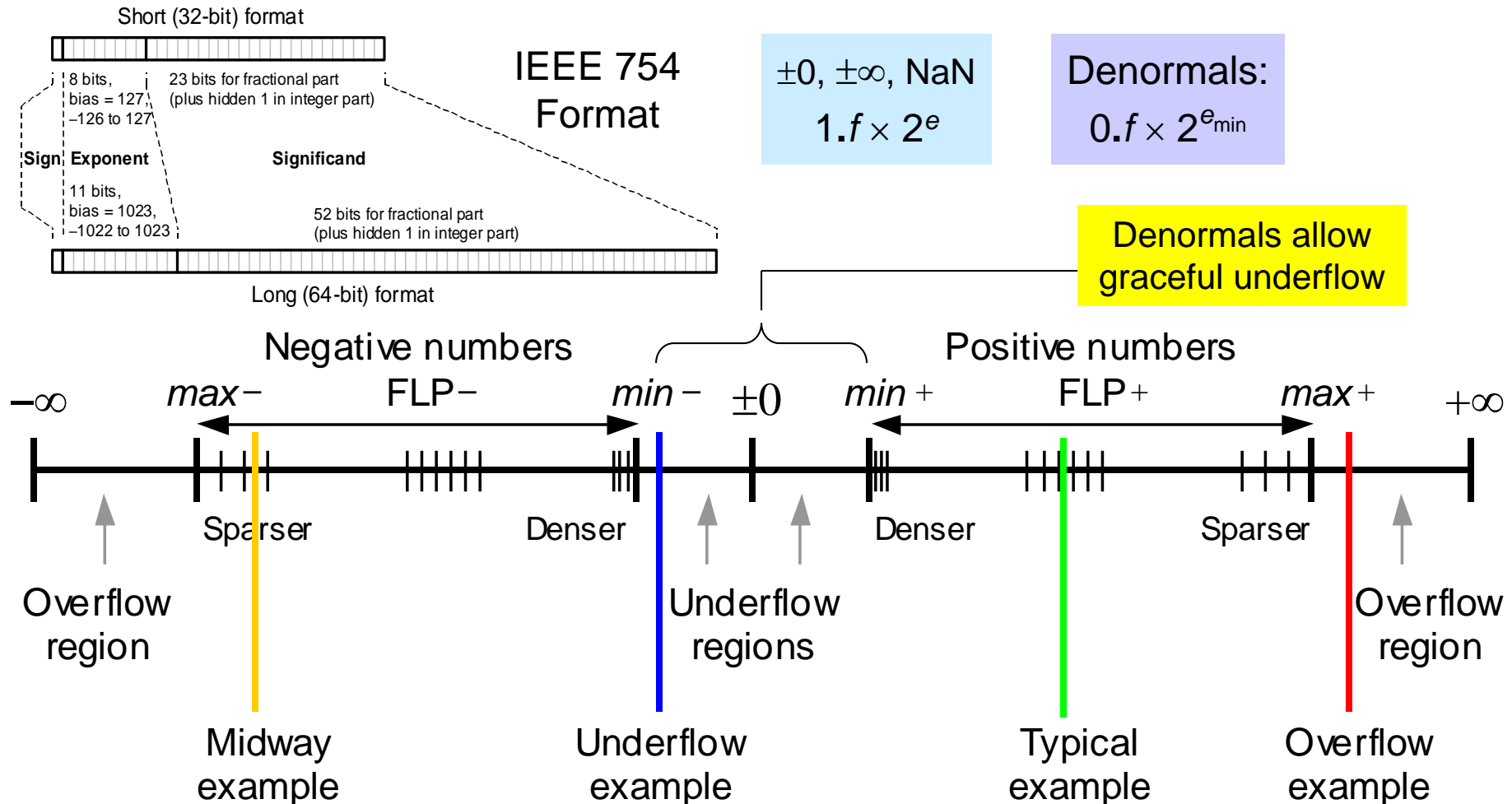
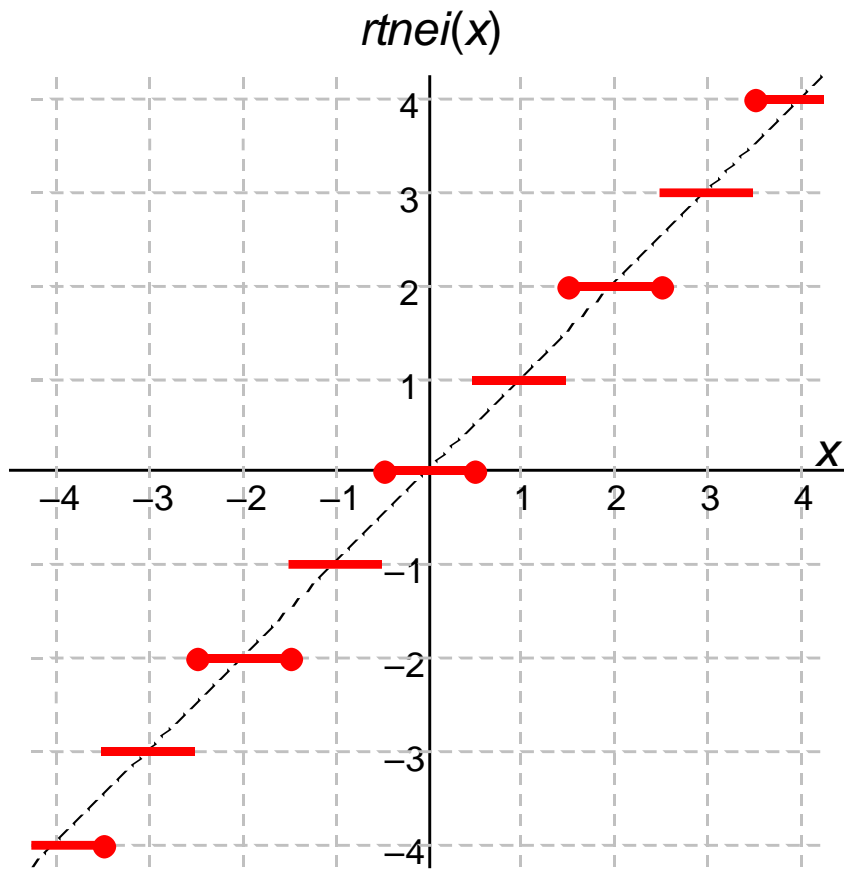
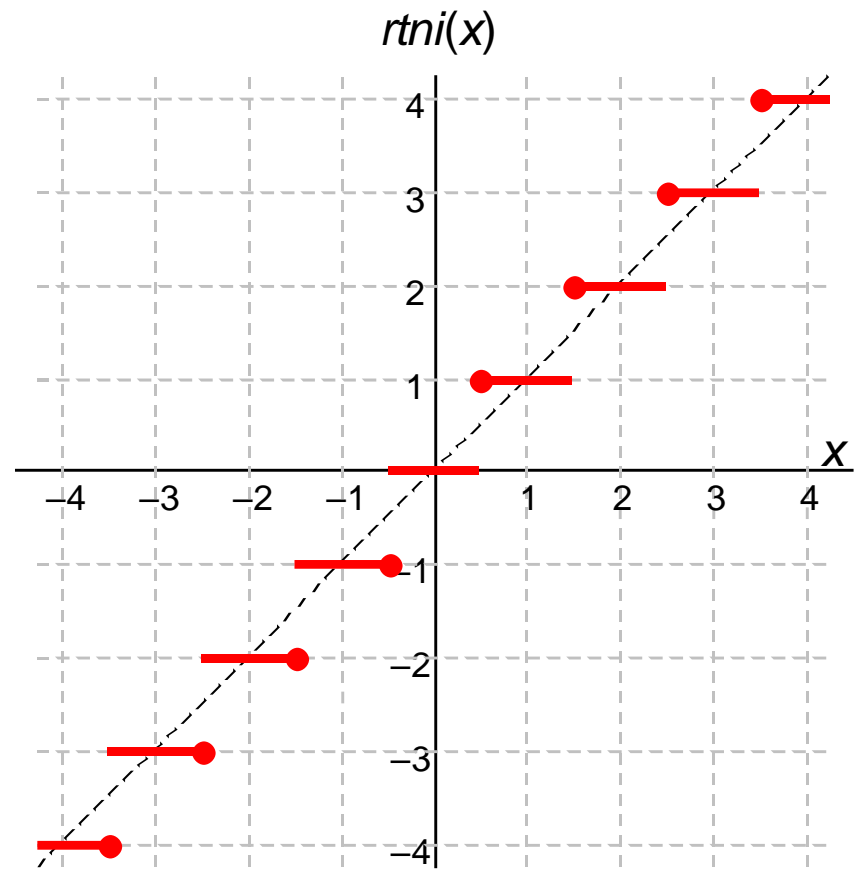


Figure 12.1 Distribution of floating-point numbers on the real line.

Round-to-Nearest (Even)



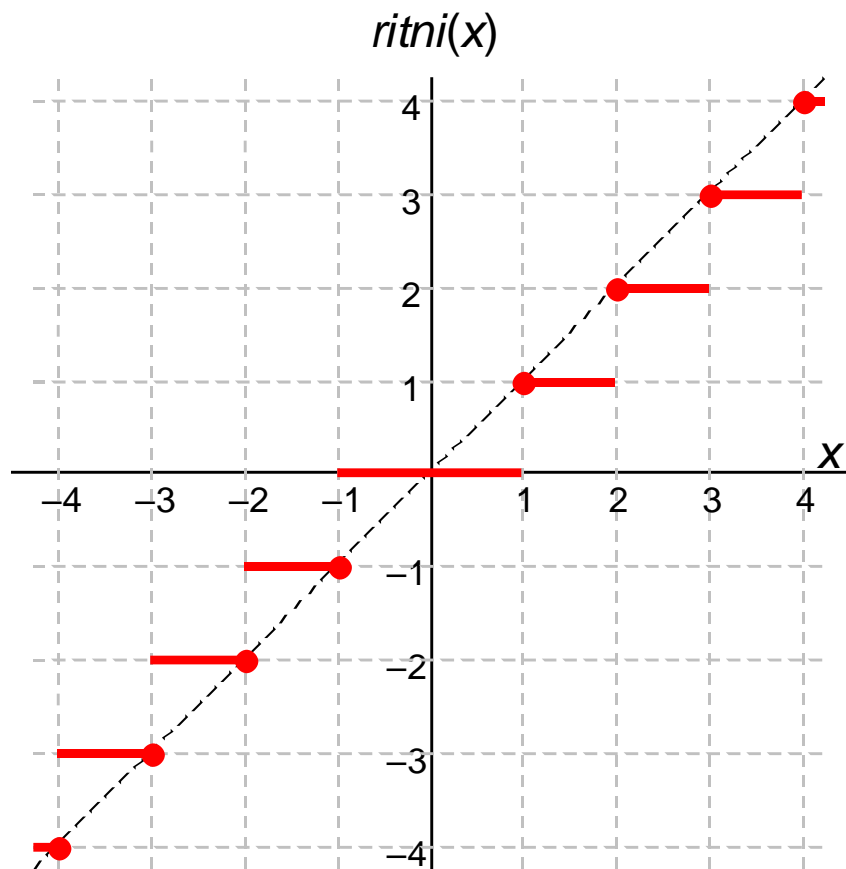
(a) Round to nearest even integer



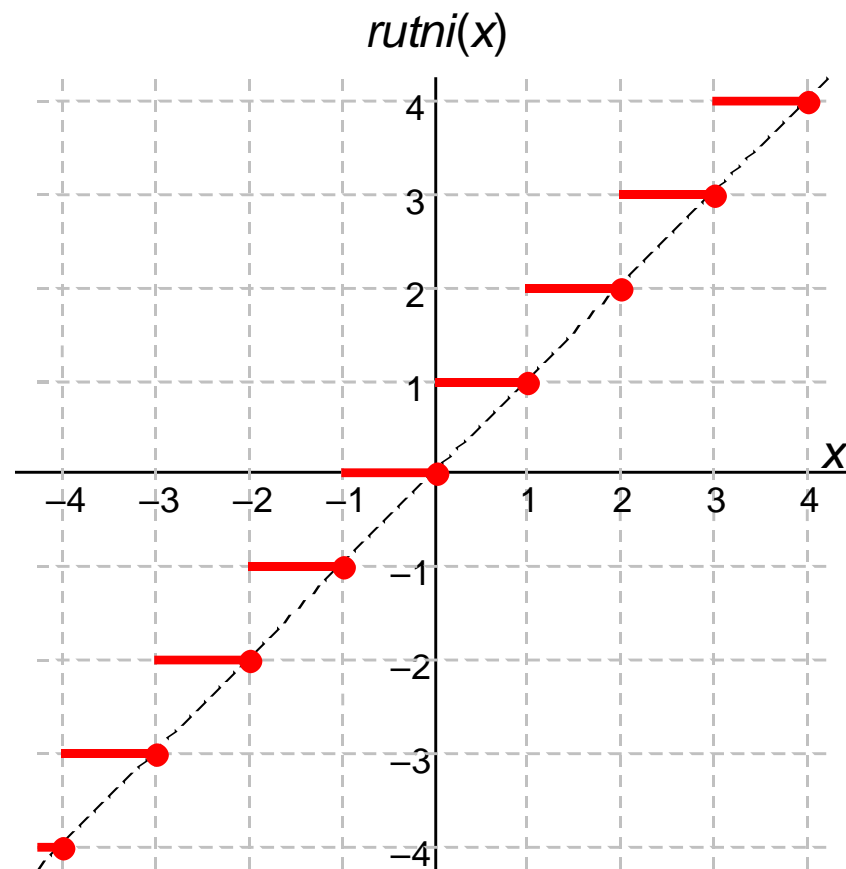
(b) Round to nearest integer

Figure 12.2 Two round-to-nearest-integer functions for x in $[-4, 4]$.

Directed Rounding



(a) Round inward to nearest integer



(b) Round upward to nearest integer

Figure 12.3 Two directed round-to-nearest-integer functions for x in $[-4, 4]$.

12.2 Special Values and Exceptions

Zeros, infinities, and NaNs (not a number)

± 0 Biased exponent = 0, significand = 0 (no hidden 1)

$\pm \infty$ Biased exponent = 255 (short) or 2047 (long), significand = 0

NaN Biased exponent = 255 (short) or 2047 (long), significand $\neq 0$

Arithmetic operations with special operands

$$(+0) + (+0) = (+0) - (-0) = +0$$

$$(+0) \times (+5) = +0$$

$$(+0) / (-5) = -0$$

$$(+\infty) + (+\infty) = +\infty$$

$$x - (+\infty) = -\infty$$

$$(+\infty) \times x = \pm\infty, \text{ depending on the sign of } x$$

$$x / (+\infty) = \pm 0, \text{ depending on the sign of } x$$

$$\sqrt{(+\infty)} = +\infty$$

Exceptions

Undefined results lead to NaN (not a number)

$$(\pm 0) / (\pm 0) = \text{NaN}$$

$$(+\infty) + (-\infty) = \text{NaN}$$

$$(\pm 0) \times (\pm \infty) = \text{NaN}$$

$$(\pm\infty) / (\pm\infty) = \text{NaN}$$

Arithmetic operations and comparisons with NaNs

NaN + x = NaN

NaN + NaN = NaN

NaN \times 0 = NaN

NaN \times NaN = NaN

NaN < 2 → false

NaN = Nan \rightarrow false

NaN $\neq (+\infty) \rightarrow \text{true}$

NaN \neq NaN \rightarrow true

Examples of invalid-operation exceptions

Addition: $(+\infty) + (-\infty)$

Multiplication: $0 \times \infty$

Division: $0 / 0$ or ∞ / ∞

Square-root: Operand < 0

12.3 Floating-Point Addition

$$(\pm 2^{e_1} s_1) + (\pm 2^{e_1} (s_2 / 2^{e_1 - e_2})) = \pm 2^{e_1} (s_1 \pm s_2 / 2^{e_1 - e_2})$$

$(\pm 2^{e_2} s_2)$ ————— ↑

Numbers to be added:

$$x = 2^5 \times 1.00101101$$

$$y = 2^1 \times 1.11101101$$

Operand with
smaller exponent
to be preshifted

Operands after alignment shift:

$$x = 2^5 \times 1.00101101$$

$$y = 2^5 \times 0.000111101101$$

Result of addition:

$$s = 2^5 \times 1.010010111101$$

$$s = 2^5 \times 1.01001100$$

Extra bits to be
rounded off

Rounded sum

Figure 12.4 Alignment shift and rounding in floating-point addition.

Hardware for Floating-Point Addition

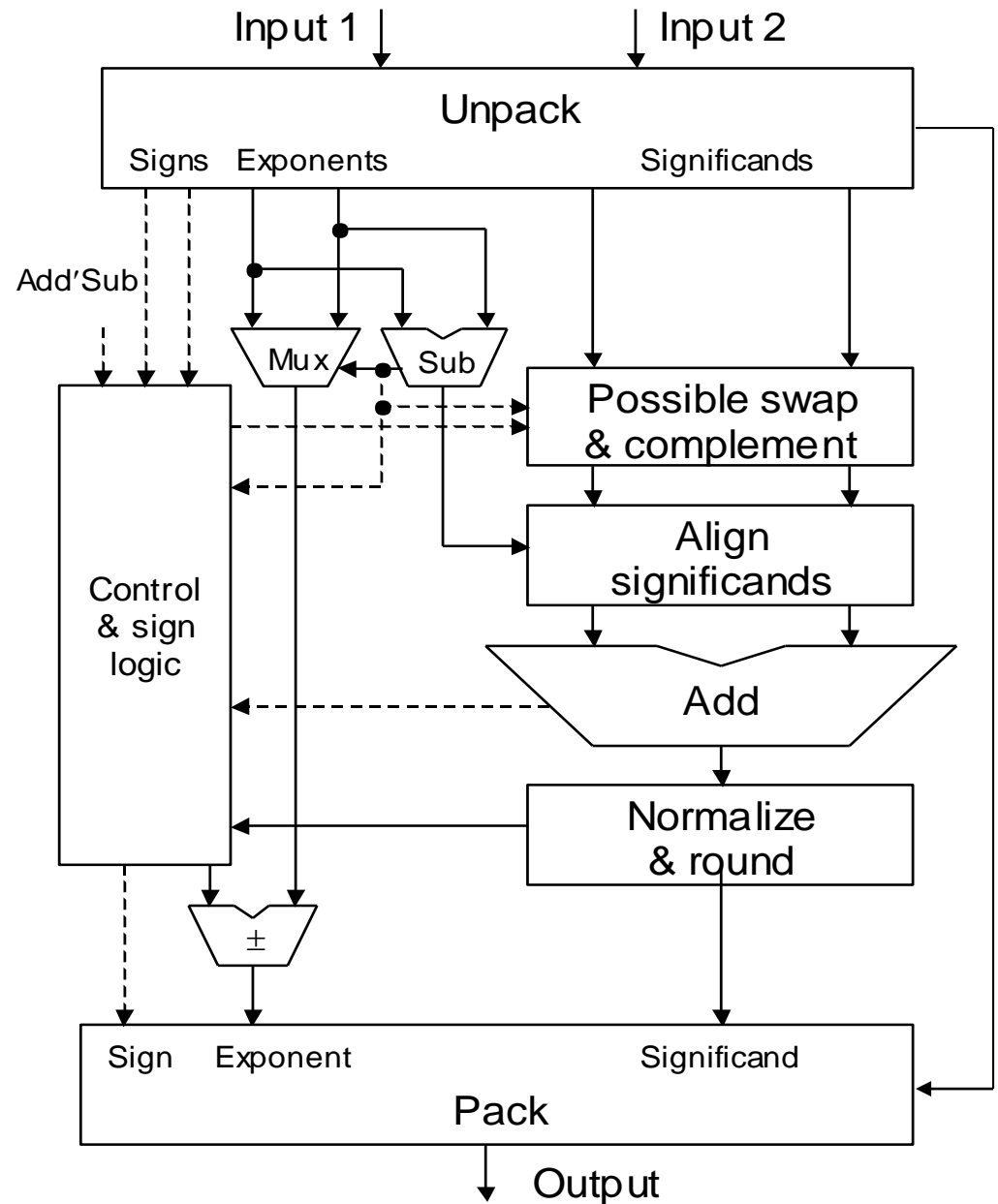


Figure 12.5
Simplified schematic of
a floating-point adder.

12.4 Other Floating-Point Operations

Floating-point multiplication

$$(\pm 2^{e_1} s_1) \times (\pm 2^{e_2} s_2) = \pm 2^{e_1 + e_2} (s_1 \times s_2)$$

Product of significands in $[1, 4)$

If product is in $[2, 4)$, halve to normalize (increment exponent)

Overflow
(underflow)
possible

Floating-point division

$$(\pm 2^{e_1} s_1) / (\pm 2^{e_2} s_2) = \pm 2^{e_1 - e_2} (s_1 / s_2)$$

Ratio of significands in $(1/2, 2)$

If ratio is in $(1/2, 1)$, double to normalize (decrement exponent)

Overflow
(underflow)
possible

Floating-point square-rooting

$$(2^e s)^{1/2} = 2^{e/2} (s)^{1/2}$$

when e is even

$$= 2^{(e-1)/2} (2s)^{1/2}$$

when e is odd

Normalization not needed

Hardware for Floating-Point Multiplication and Division

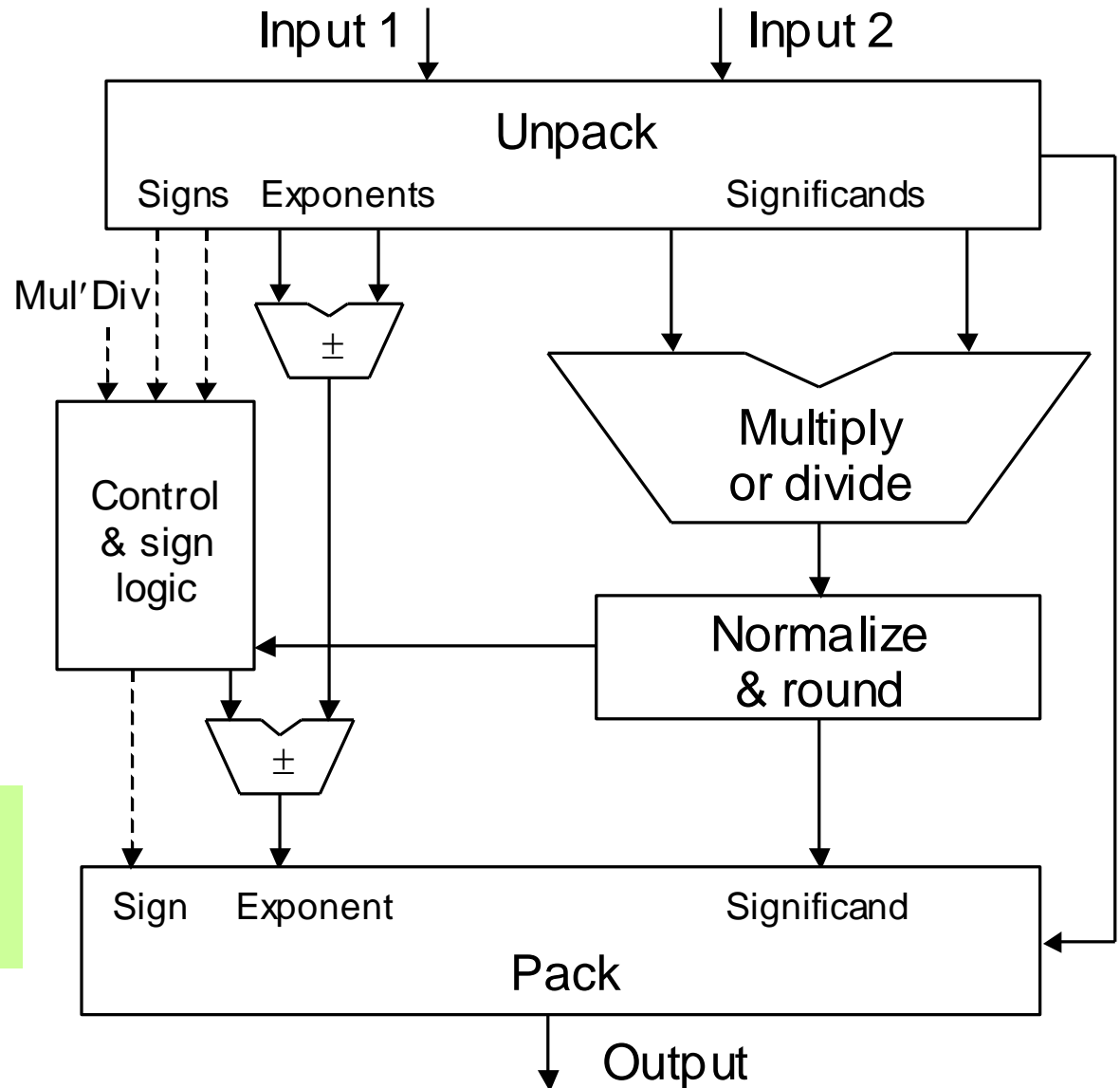


Figure 12.6 Simplified schematic of a floating-point multiply/divide unit.

12.5 Floating-Point Instructions

Floating-point arithmetic instructions for MiniMIPS:

```

add.s    $f0,$f8,$f10    # set $f0 to ($f8) +fp ($f10)
sub.d    $f0,$f8,$f10    # set $f0 to ($f8) -fp ($f10)
mul.d    $f0,$f8,$f10    # set $f0 to ($f8) ×fp ($f10)
div.s    $f0,$f8,$f10    # set $f0 to ($f8) /fp ($f10)
neg.s    $f0,$f8          # set $f0 to -($f8)
  
```

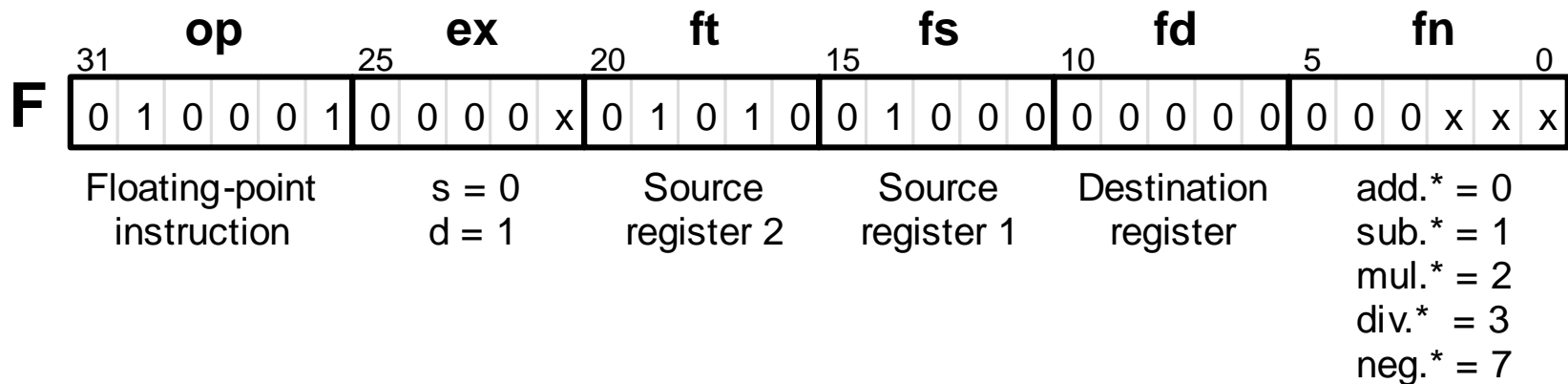


Figure 12.7 The common floating-point instruction format for MiniMIPS and components for arithmetic instructions. The extension (ex) field distinguishes single (* = s) from double (* = d) operands.

The Floating-Point Unit in MiniMIPS

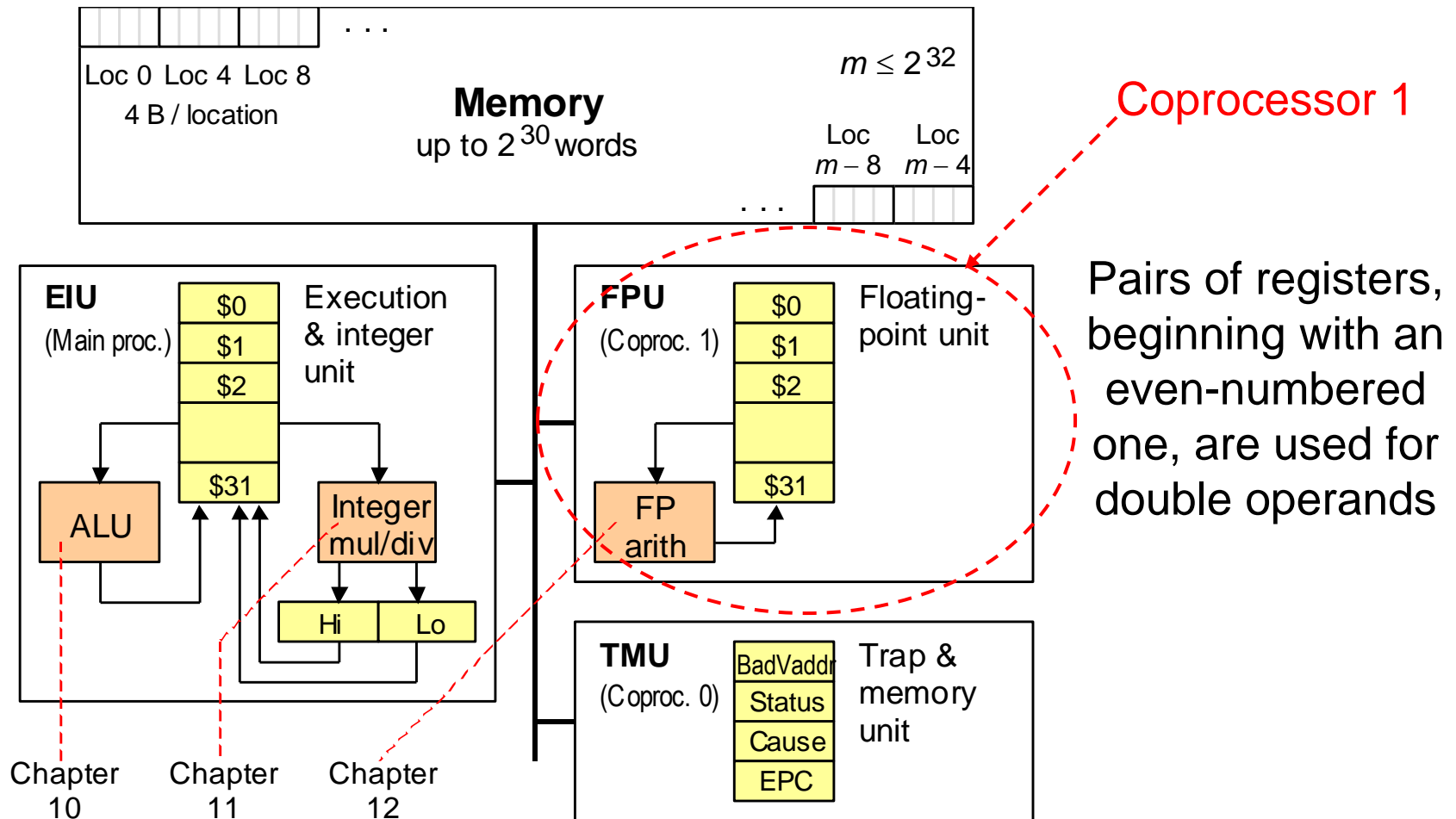


Figure 5.1 Memory and processing subsystems for MiniMIPS.

Floating-Point Data Transfers

MiniMIPS instructions for floating-point load, store, and move:

```
lwc1    $f8, 40($s3)    # load mem[40+($s3)] into $f8
swc1    $f8, A($s3)     # store ($f8) into mem[A+($s3)]
mov.s   $f0, $f8         # load $f0 with ($f8)
mov.d   $f0, $f8         # load $f0, $f1 with ($f8, $f9)
mfc1    $t0, $f12        # load $t0 with ($f12)
mtc1    $f8, $t4         # load $f8 with ($t4)
```

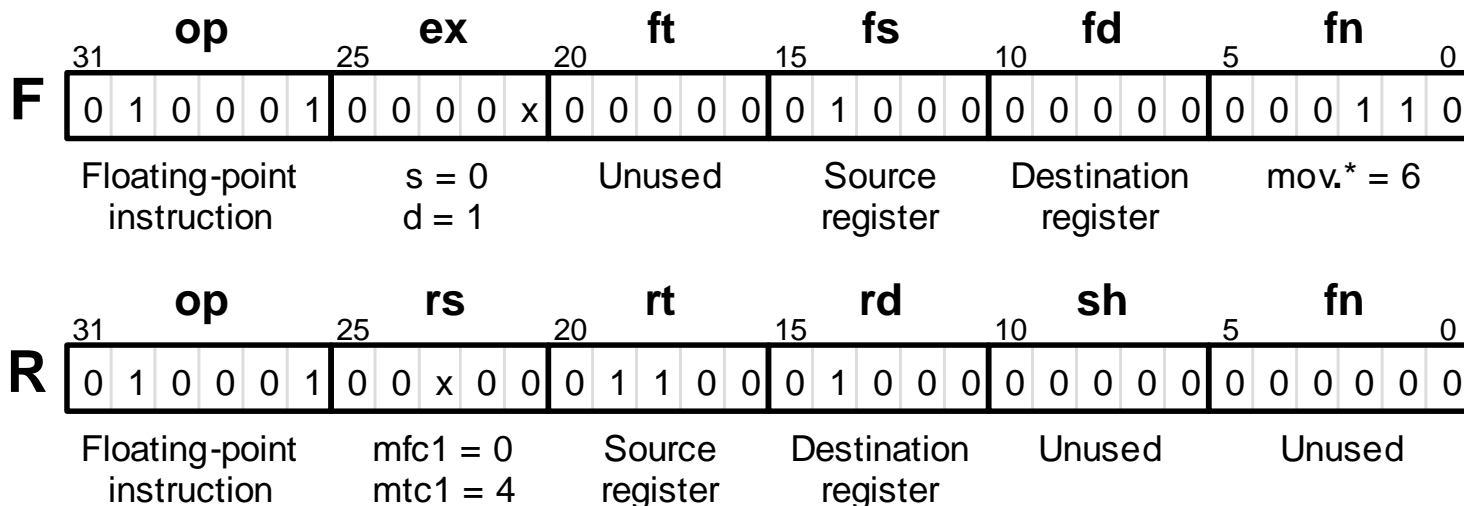


Figure 12.9 Instructions for floating-point data movement in MiniMIPS.

Floating-Point Branches and Comparisons

MiniMIPS instructions for floating-point load, store, and move:

```

bclt    L          # branch on fp flag true
bclf    L          # branch on fp flag false
c.eq.*  $f0,$f8     # if ($f0) = ($f8), set flag to "true"
c.lt.*  $f0,$f8     # if ($f0) < ($f8), set flag to "true"
c.le.*  $f0,$f8     # if ($f0) ≤ ($f8), set flag to "true"
    
```

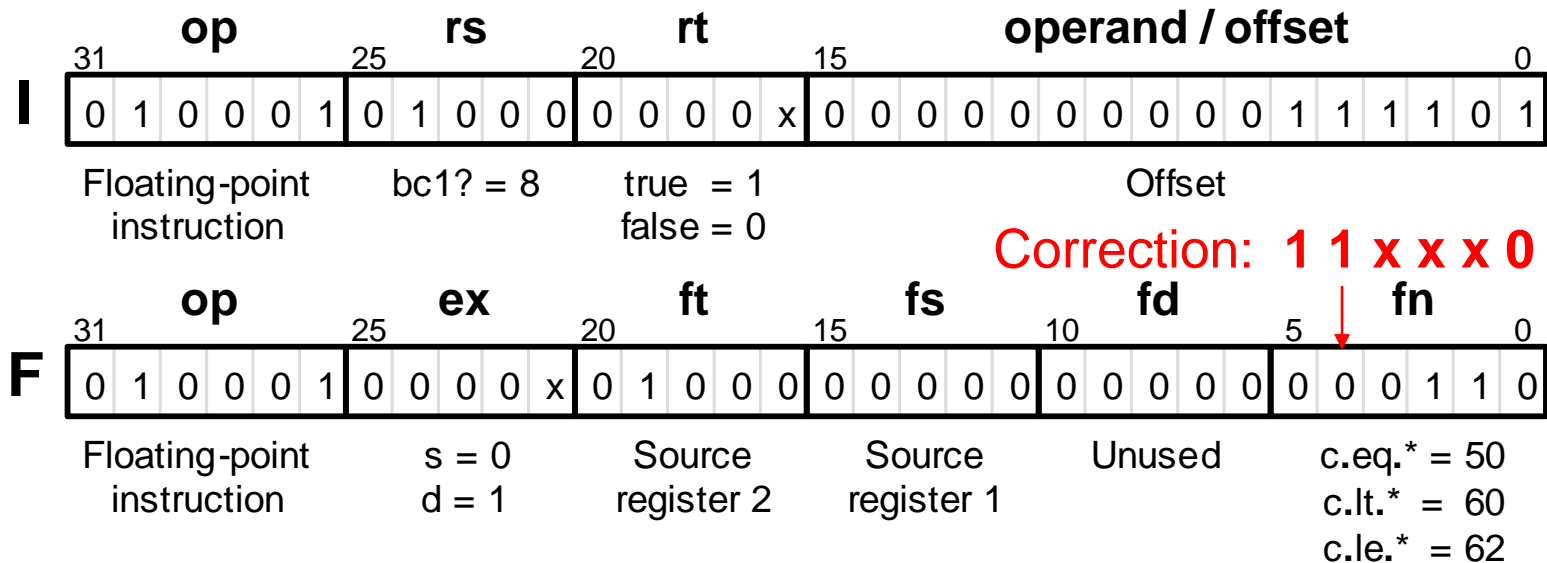


Figure 12.10 Floating-point branch and comparison instructions in MiniMIPS.

Floating-Point Instructions of MiniMIPS

Table 12.1

Copy

Arithmetic

Conversions

Memory access

Control transfer

* s/d for single/double
0/1 for single/double

Instruction	Usage
Move s/d registers	mov.* fd, fs
Move fm coprocessor 1	mfc1 rt, rd
Move to coprocessor 1	mtc1 rd, rt
Add single/double	add.* fd, fs, ft
Subtract single/double	sub.* fd, fs, ft
Multiply single/double	mul.* fd, fs, ft
Divide single/double	div.* fd, fs, ft
Negate single/double	neg.* fd, fs
Compare equal s/d	c.eq.* fs, ft
Compare less s/d	c.lt.* fs, ft
Compare less or eq s/d	c.le.* fs, ft
Convert integer to single	cvt.s.w fd, fs
Convert integer to double	cvt.d.w fd, fs
Convert single to double	cvt.d.s fd, fs
Convert double to single	cvt.s.d fd, fs
Convert single to integer	cvt.w.s fd, fs
Convert double to integer	cvt.w.d fd, fs
Load word coprocessor 1	lwcl ft, imm(rs)
Store word coprocessor 1	swcl ft, imm(rs)
Branch coproc 1 true	bclt L
Branch coproc 1 false	bclf L

ex	fn
#	6
0	
4	
#	0
#	1
#	2
#	3
#	7
#	50
#	60
#	62
0	32
0	33
1	33
1	32
0	36
1	36
rs	
rs	
8	
8	

12.6 Result Precision and Errors

Example 12.4

Laws of algebra may not hold in floating-point arithmetic. For example, the following computations show that the associative law of addition, $(a + b) + c = a + (b + c)$, is violated for the three numbers shown.

Numbers to be added first

$$a = -2^5 \times 1.10101011$$

$$b = 2^5 \times 1.10101110$$

Numbers to be added first

$$b = 2^5 \times 1.10101110$$

$$c = -2^{-2} \times 1.01100101$$

Compute $a + b$

$$2^5 \times 0.00000011$$

$$a+b = 2^{-2} \times 1.10000000$$

$$c = -2^{-2} \times 1.01100101$$

Compute $b + c$ (after preshifting c)

$$2^5 \times 1.101010110011011$$

$$b+c = 2^5 \times 1.10101011 \quad (\text{Round})$$

$$a = -2^5 \times 1.10101011$$

Compute $(a + b) + c$

$$2^{-2} \times 0.00011011$$

$$\text{Sum} = 2^{-6} \times 1.10110000$$

Compute $a + (b + c)$

$$2^5 \times 0.00000000$$

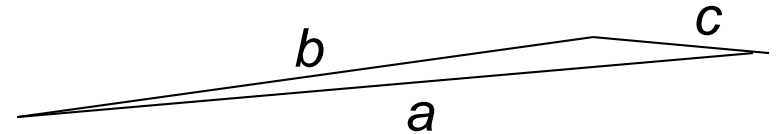
$$\text{Sum} = 0 \quad (\text{Normalize to special code for 0})$$

Error Control and Certifiable Arithmetic

Catastrophic cancellation in subtracting almost equal numbers:

Area of a needlelike triangle

$$A = [s(s - a)(s - b)(s - c)]^{1/2}$$



Possible remedies

Carry extra precision in intermediate results (guard digits):
commonly used in calculators

Use alternate formula that does not produce cancellation errors

Certifiable arithmetic with intervals

A number is represented by its lower and upper bounds $[x_l, x_u]$

Example of arithmetic: $[x_l, x_u] +_{\text{interval}} [y_l, y_u] = [x_l +_{\text{fp}\nabla} y_l, x_u +_{\text{fp}\Delta} y_u]$

Evaluation of Elementary Functions

Approximating polynomials

$$\ln x = 2(z + z^3/3 + z^5/5 + z^7/7 + \dots) \text{ where } z = (x - 1)/(x + 1)$$

$$e^x = 1 + x/1! + x^2/2! + x^3/3! + x^4/4! + \dots$$

$$\cos x = 1 - x^2/2! + x^4/4! - x^6/6! + x^8/8! - \dots$$

$$\tan^{-1} x = x - x^3/3 + x^5/5 - x^7/7 + x^9/9 - \dots$$

Iterative (convergence) schemes

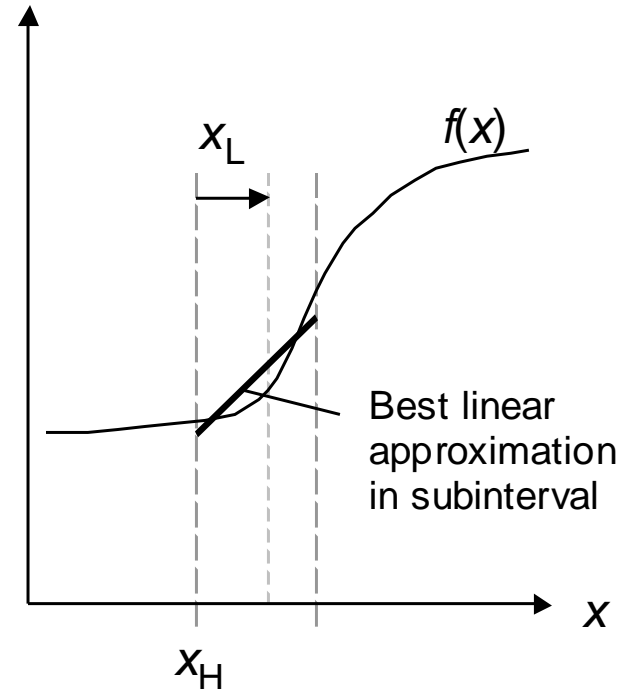
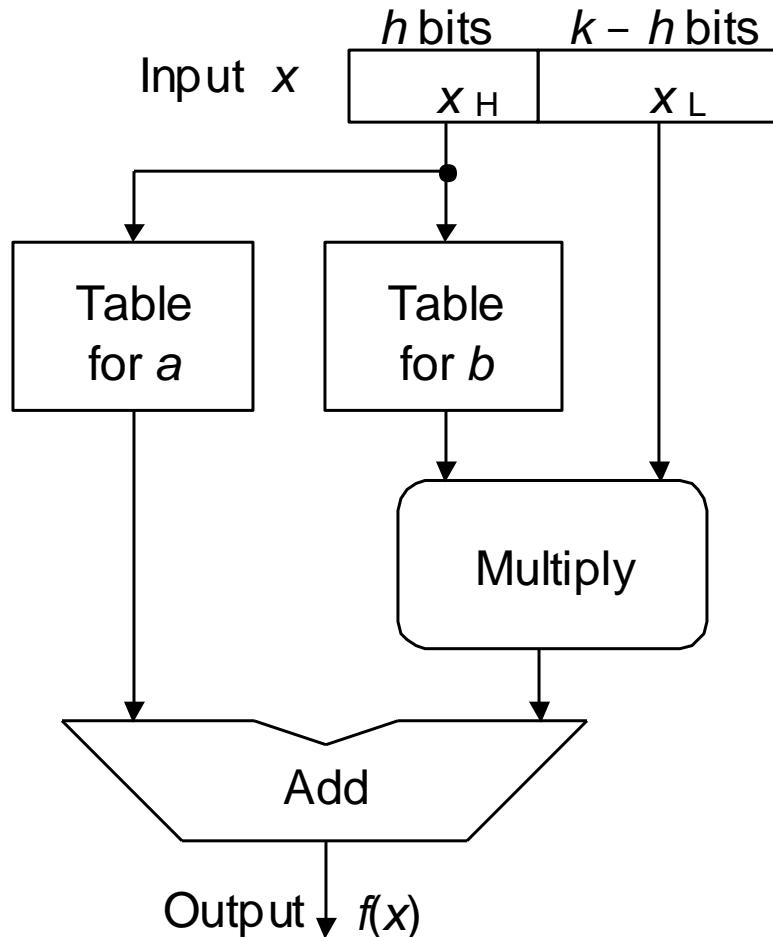
For example, beginning with an estimate for $x^{1/2}$, the following iterative formula provides a more accurate estimate in each step

$$q^{(i+1)} = 0.5(q^{(i)} + x/q^{(i)})$$

Table lookup (with interpolation)

A pure table lookup scheme results in huge tables (impractical); hence, often a hybrid approach, involving interpolation, is used.

Function Evaluation by Table Lookup



The linear approximation above is characterized by the line equation $a + b x_L$, where a and b are read out from tables based on x_H

Figure 12.12 Function evaluation by table lookup and linear interpolation.