EC7551: COMPUTER ARCHITECTURE AND ORGANIZATION

UNIT II: DATAPATH DESIGN

Presented By,

Dr. V. SATHIESH KUMAR

Assistant Professor

Department of Electronics Engineering

MIT-Anna University

SYLLABUS:

UNIT II DATAPATH DESIGN

Fixed Point Arithmetic, Addition, Subtraction, Multiplication and Division, Combinational and Sequential ALUs, Carry look ahead adder, Robertson algorithm, Booth's Algorithm, Non restoring division algorithm, Floating point arithmetic, Coprocessor, Pipeline Processing, Pipeline Design, Modified Booth algorithm

TEXT BOOKS:

1. John P.Hayes, _Computer architecture and Organization', Tata McGraw-Hill, Third edition, 1998.
2. V.Carl Hamacher, Zvonko G. Varanesic and Safat G. Zaky, — Computer Organization—, V edition, McGraw-Hill Inc, 1996.

Presentation Slides:

www.sathieshkumar.com/tutorials

FIXED-POINT ARITHMETIC

- An instruction set processor consists of datapath (data processing) and control units.
- Four basic arithmetic instructions for fixed-point numbers are addition, subtraction, multiplication and division.

Addition and Subtraction :

- Add and subtract instructions for fixed–point binary numbers are found in the instruction set of every computer.
- The sum z_i, c_i of two 1-bit numbers x_i and y_i can be expressed by the half-adder logic equations,

$$z_i = x_i \oplus y_i$$

$$c_i = x_i y_i$$

Where z_i is the sum bit, c_i is the carry out bit.

FIXED-POINT ARITHMETIC

➢1-bit adder or full-adder logic equations,

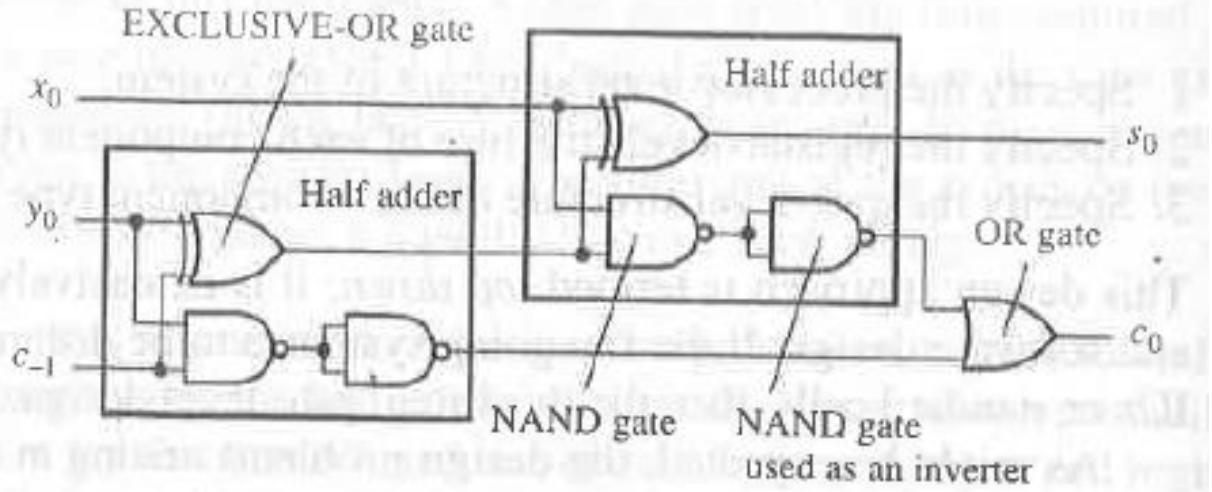
$$z_i = x_i \oplus y_i \oplus c_{i-1}$$

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

Where z_i is the sum bit, c_i is the carry out bit and c_{i-1} is the carry in signal.

Inputs			Outputs	
x_0	y_0	c_{-1}	c_0	s_0
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

(a)



(b)

FIXED-POINT ARITHMETIC

- Fast AND-OR realization of a 1-bit adder,

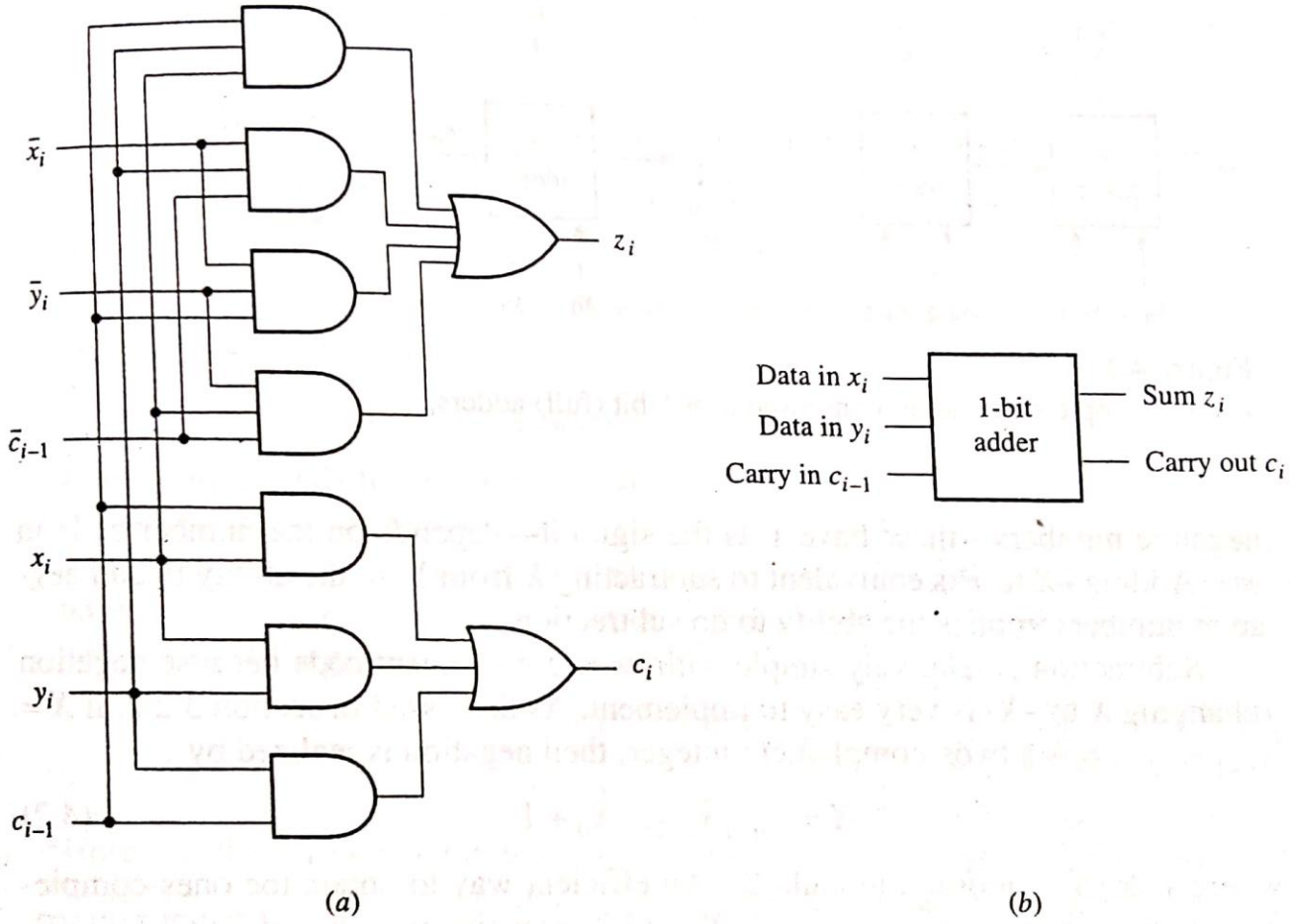


Figure 4.1
A 1-bit (full) adder: (*a*) two-level AND-OR logic circuit and (*b*) symbol.

FIXED-POINT ARITHMETIC

- The least expensive circuit in terms of hardware cost for adding two n-bit binary numbers is a **serial adder.**
- A serial adder adds the numbers bit by bit and so requires n clock cycles to compute the complete sum of two n-bit numbers.
- Although this adder is slow, its circuit size is very small and is independent of n.

FIXED-POINT ARITHMETIC

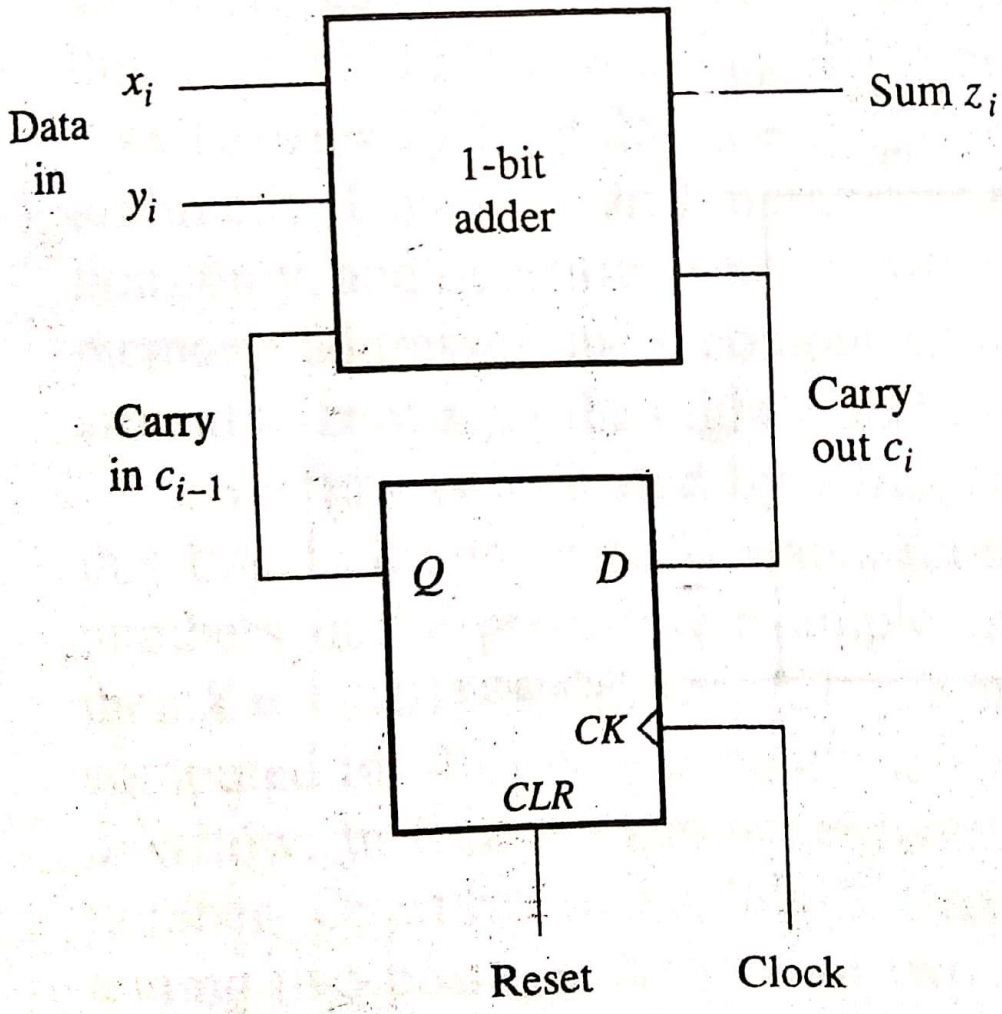


Figure 4.2
A serial binary adder.

FIXED-POINT ARITHMETIC

➢ Circuits that, in one clock cycle, add all bits of two n-bit numbers, as well as an external carry-in signal c_{in}, are called **n-bit parallel adders** or simply **n-bit adders**.

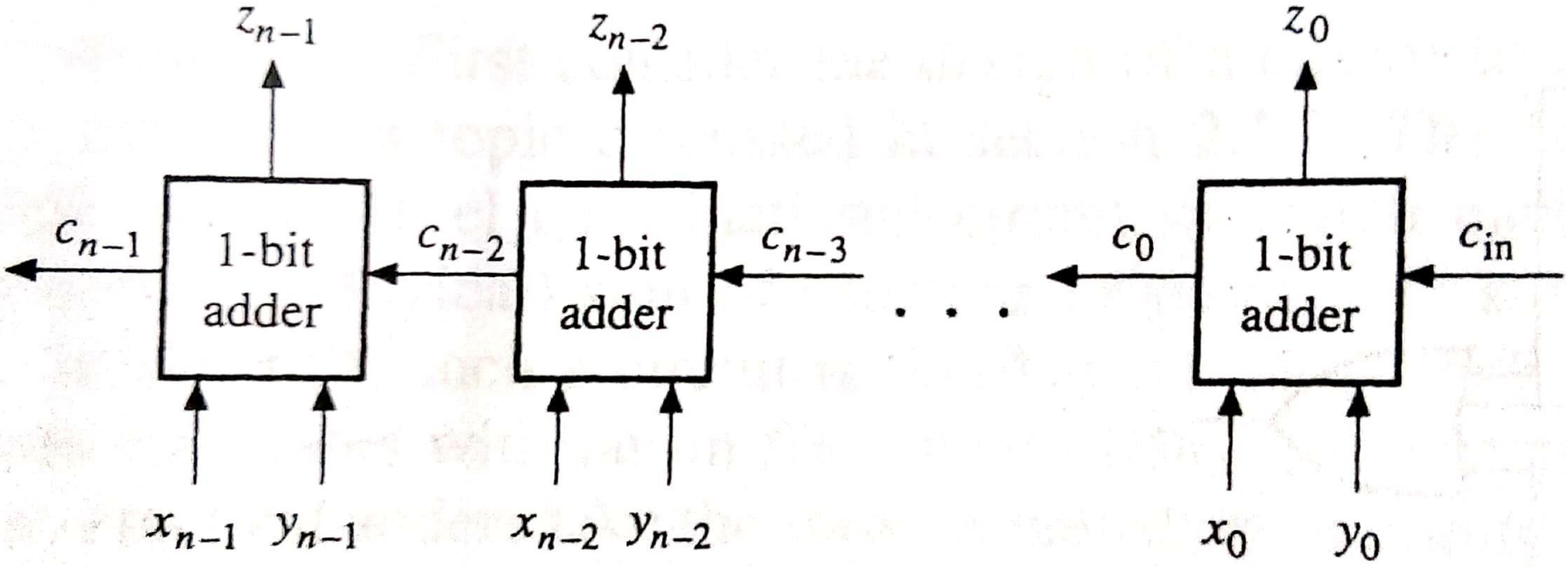
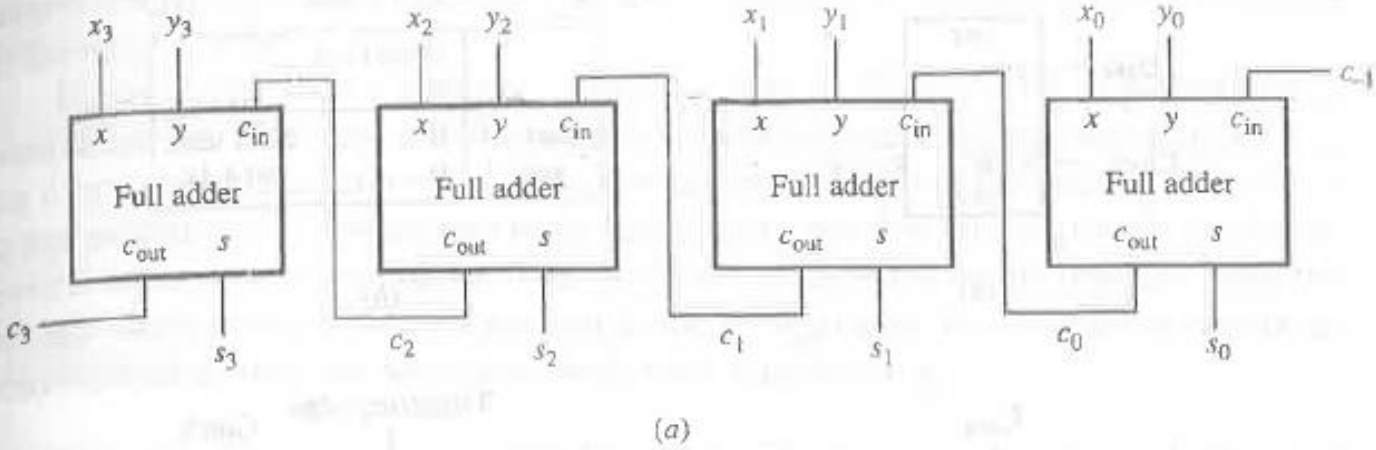


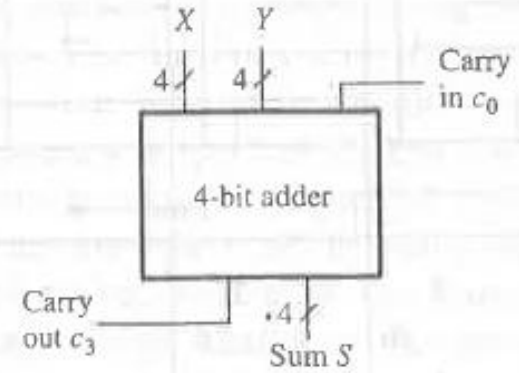
Figure 4.3
An *n*-bit ripple-carry adder composed of *n* 1-bit (full) adders.

FIXED-POINT ARITHMETIC

- 4-bit adder:



(a)



(b)

Figure 2.10
Four-bit ripple-carry: (*a*) logic structure; (*b*) high-level symbol.

FIXED-POINT ARITHMETIC

- Carry signals propagate through the adder from right to left, giving rise to the name **ripple-carry adder**.
- Worst case – a carry signal can ripple through all n stages of the adder.
- The input carry signal c_{in} is normally set to 0 for addition.
- The maximum signal propagation delay of an n-bit ripple-carry adder, which in synchronous circuit design determines the operating speed, is **nd**, where d is the delay of a full-adder stage.
- Unlike a serial adder, the amount of hardware in a ripple-carry adder increases linearly with n, the word size of the numbers being added.

FIXED-POINT ARITHMETIC

Subtracters:

- The best way to add negative numbers – these have 1 as the sign bit – depends on the number code in use.
- Adding –X to Y is equivalent to subtracting X from Y.
- Subtraction is relatively simple with twos-complement code because negation (changing X to -X) is very easy to implement.
- If $X=x_{n-1}\ x_{n-2}\ \ldots\ x_0$ is a twos complement integer, then negation is realized by,

$$-X = \overline{x_{n-1}}\,\overline{x_{n-2}}\ldots\overline{x_0} + 1$$

FIXED-POINT ARITHMETIC

Subtracters:

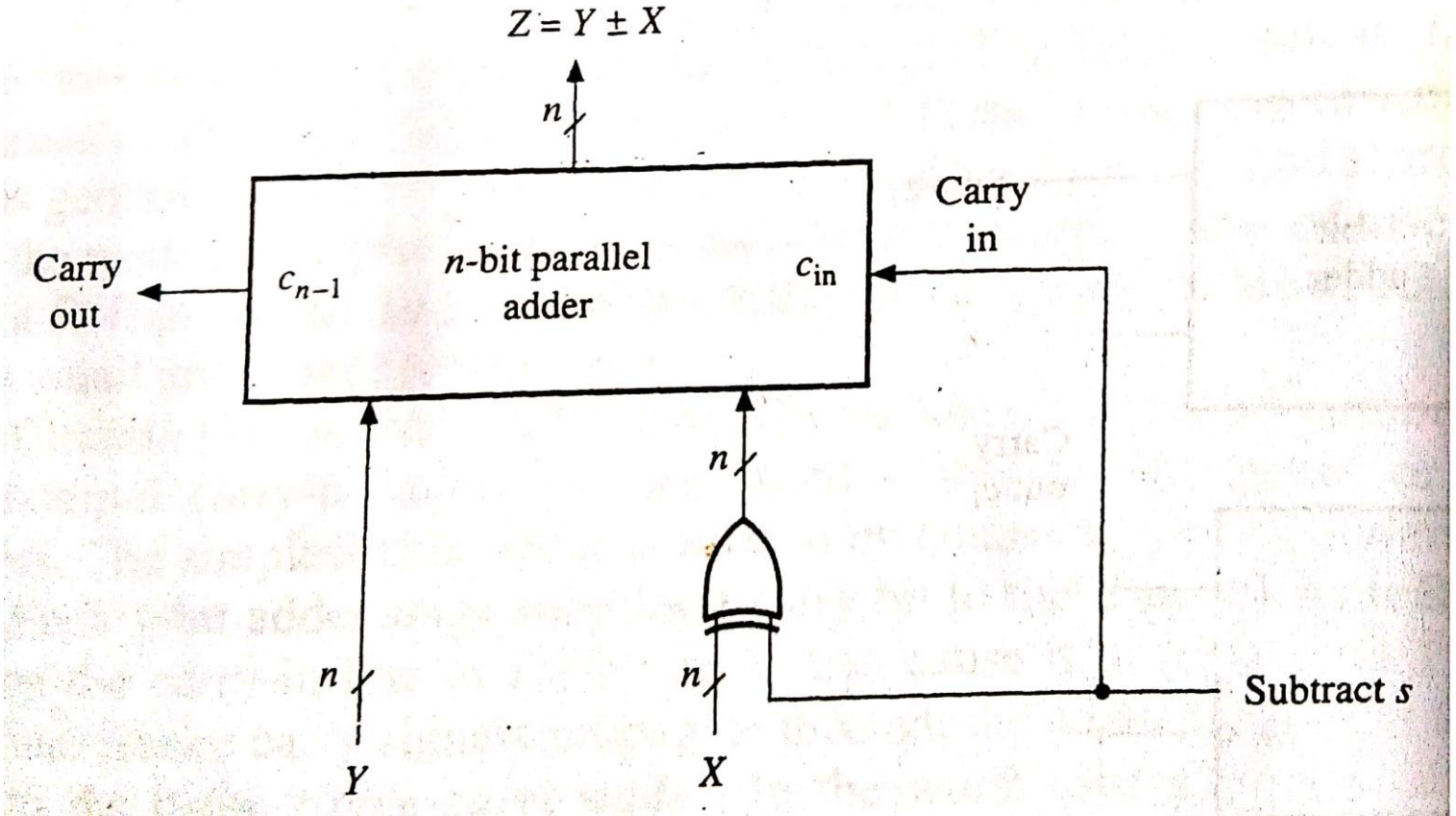


Figure 4.4
An *n*-bit twos-complement adder-subtracter.

FIXED-POINT ARITHMETIC

Subtracters:

- Example : let X=11101011 (denoting -21) and Y=00101000 (denoting 40) – in twos-complement code
- Bit by bit addition produces,

Z=X+Y=11101011 + 00101000 = 00010011

Which corresponds to $-21_{10} + 40_{10} = +19_{10}$

- 1-bit full subtractor function is given by $z_i = y_i - x_i - b_{i-1}$
- The operation is defined by the logic equations,

$$z_i = x_i \oplus y_i \oplus b_{i-1}$$

$$b_i = x_i \overline{y_i} + x_i b_{i-1} + \overline{y_i} b_{i-1}$$

Where z_i is the difference bit, b_{i-1} and b_i are the borrow-in and borrow-out bits.

- n-bit serial or parallel binary subtracters are constructed in essentially the same way as the corresponding adders with carry signals replaced by borrows.

FIXED-POINT ARITHMETIC

Subtracters:

- Subtracters are of minor interest compared with adders, because, an adder suffices for both addition and subtraction when twos-complement number code is used.

FIXED-POINT ARITHMETIC

Overflow:

- When the result of an arithmetic operation exceeds the standard word size n, **overflow** occurs.
- With n-bit unsigned numbers, overflow is indicated by an output carry bit $c_{n-1}=1$.
- Unsigned arithmetic operations are often viewed as modulo-2^n operations only, and overflow is not explicitly detected.
- This is the case when computing memory addresses in a computer, for instance, where addresses simply wrap around to zero after the highest address is reached.
- Overflow is indicated by a flag bit v in operations involving signed numbers; this flag is found in CPU status (condition code) registers.

FIXED-POINT ARITHMETIC

High-speed adders:

- General strategy – reduce the time required to form carry signals.
- One approach is to compute the input carry needed by stage i directly from carrylike signals obtained from all the preceding stages i-1, i-2,…,0, rather than waiting for normal carries to ripple slowly from stage to stage – **carry lookahead adders**.
- An **n-bit carry lookahead adder** is formed from n stages, each of which is basically a full adder modified by replacing its carry output line c_i by two auxiliary signals called $\mathbf{g_i}$ and $\mathbf{p_i}$ or **generate** and **propagate**, respectively, which are defined by the following equations,

$$g_i = x_i\, y_i \qquad p_i = x_i + y_i$$

- The name **generate** comes from the fact that stage i generates a carry of 1 ($c_i = 1$) independent of the value of c_{i-1} if both x_i and y_i are 1; that is, if $x_i\, y_i = 1$.
- Stage i propagates c_{i-1}; that is, it makes c_i=1 in response to $c_{i-1} = 1$ if x_i or y_i is 1.

$$x_i + y_i = 1$$

FIXED-POINT ARITHMETIC

High-speed adders:

- Carry signal c_i is given by,

$$c_i = g_i + p_i c_{i-1}$$

- Sum signal z_i is given by,

$$z_i = x_i \oplus y_i \oplus c_{i-1}$$

or is equivalent to ,

$$z_i = p_i \oplus g_i \oplus c_{i-1}$$

FIXED-POINT ARITHMETIC

High-speed adders:

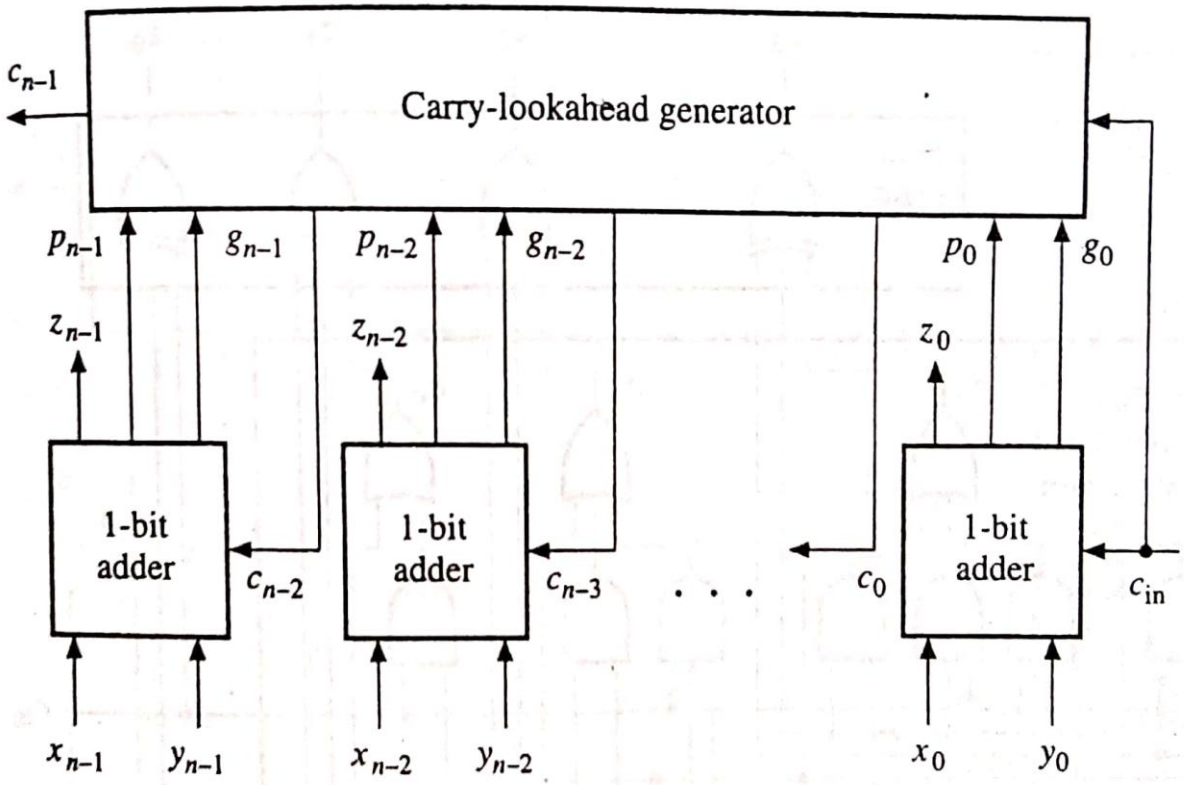


Figure 4.5
Overall structure of carry-lookahead adder.

FIXED-POINT ARITHMETIC

High-speed adders: 4-bit carry lookahead adder

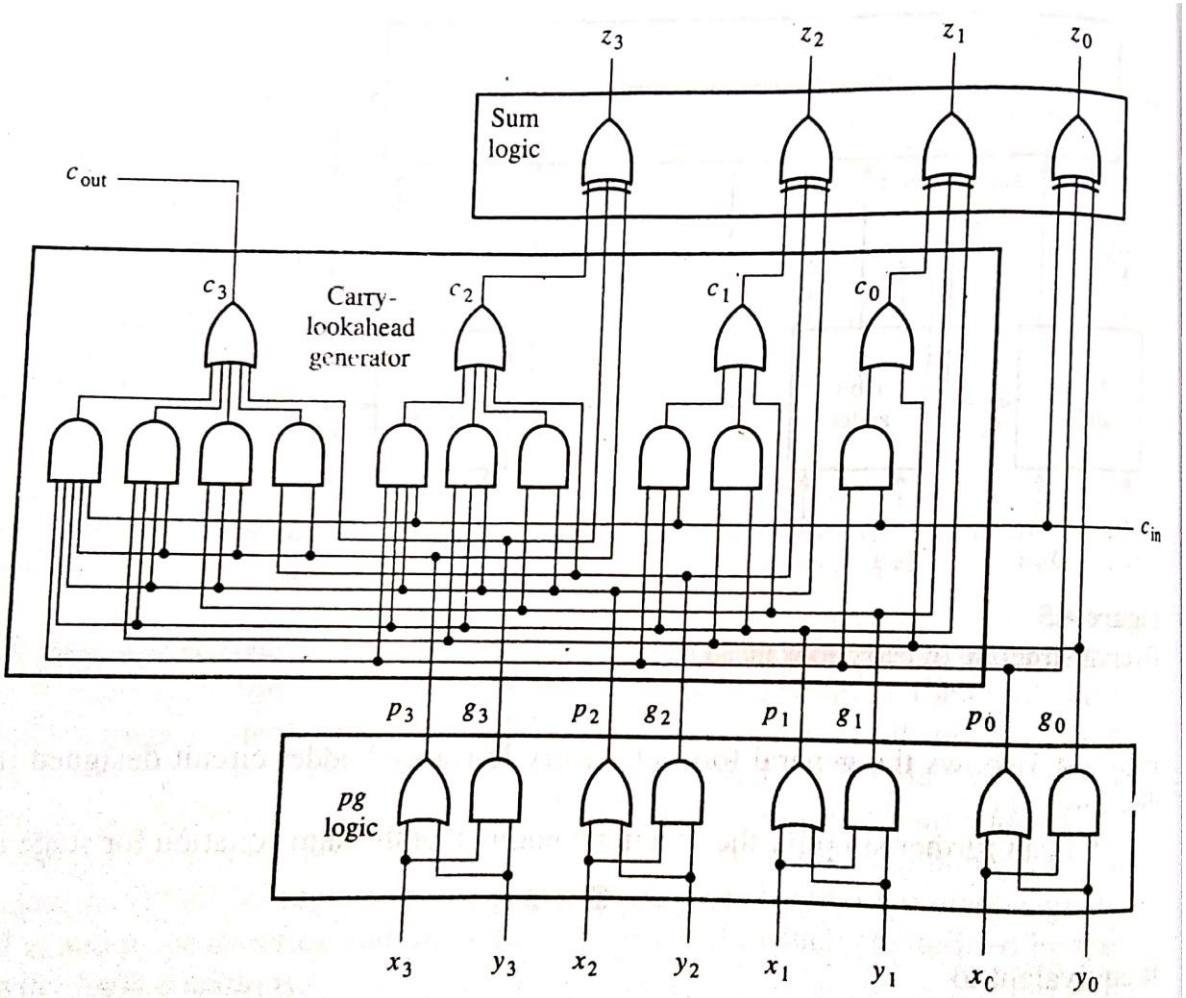


Figure 4.6
A 4-bit carry-lookahead adder.

FIXED-POINT ARITHMETIC

High-speed adders: 4-bit carry lookahead adder

- Practical adder found in IC 74283.
- It has four levels of logic gates, so the adder's maximum delay is 4d, where d is the (average) gate delay.
- This delay is independent of the number of inputs n as long as carry generation is defined by two level logic.
- However the number of gates grow in proportion to n^2 as n increases.

FIXED-POINT ARITHMETIC

Adder expansion: 16-bit adder using ripple carry propagation

- Cheap and slow

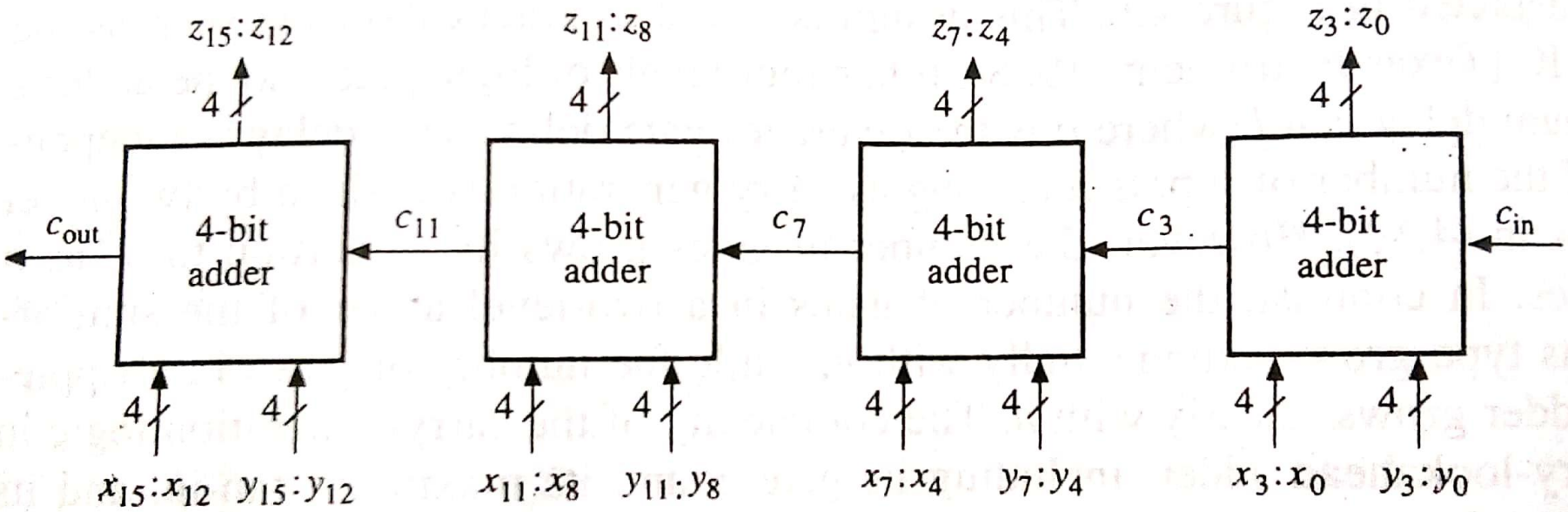


Figure 4.7
A 16-bit adder composed of 4-bit adders linked by ripple-carry propagation.

FIXED-POINT ARITHMETIC

Adder expansion: 16-bit adder using carry lookahead adder

- fast, expensive and impractical because of the complexity of its carry-generation logic.

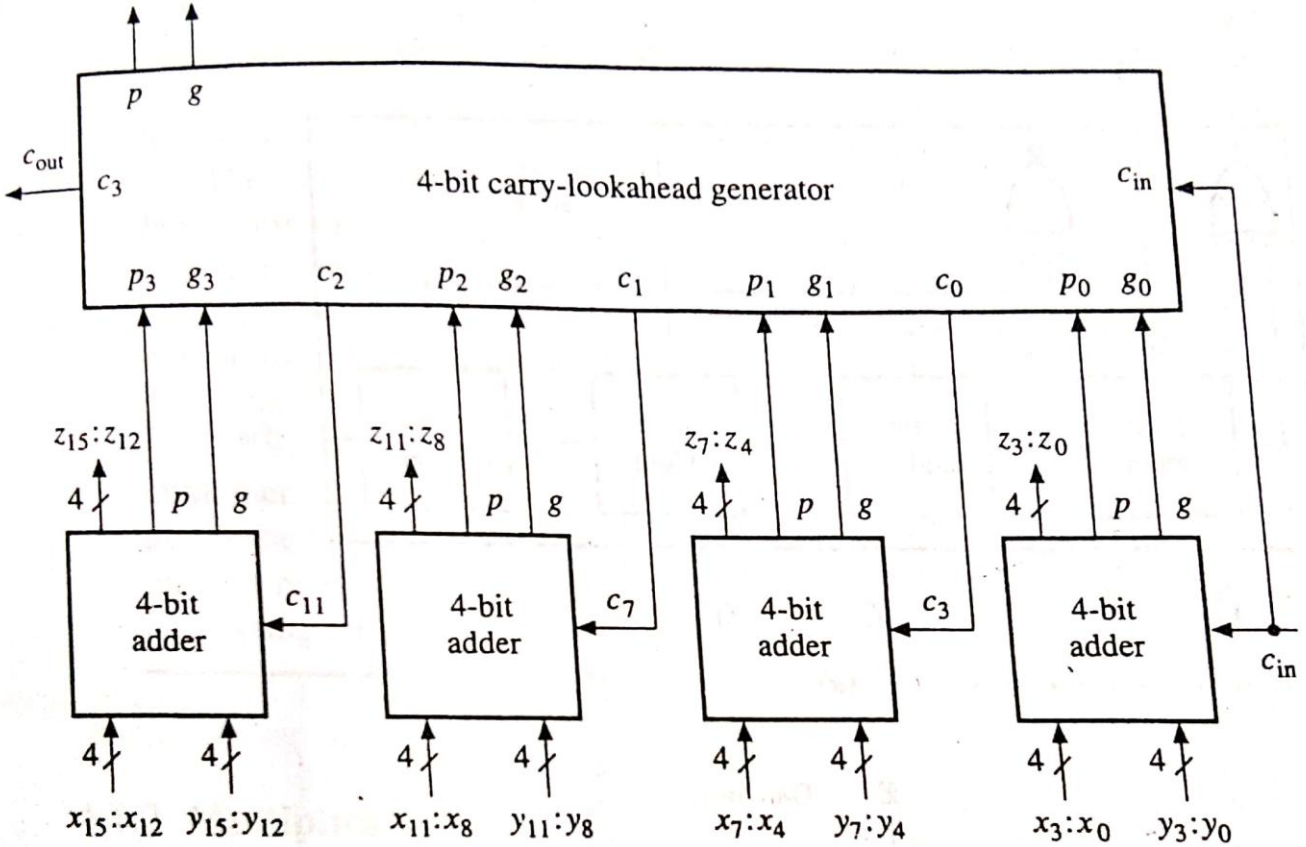
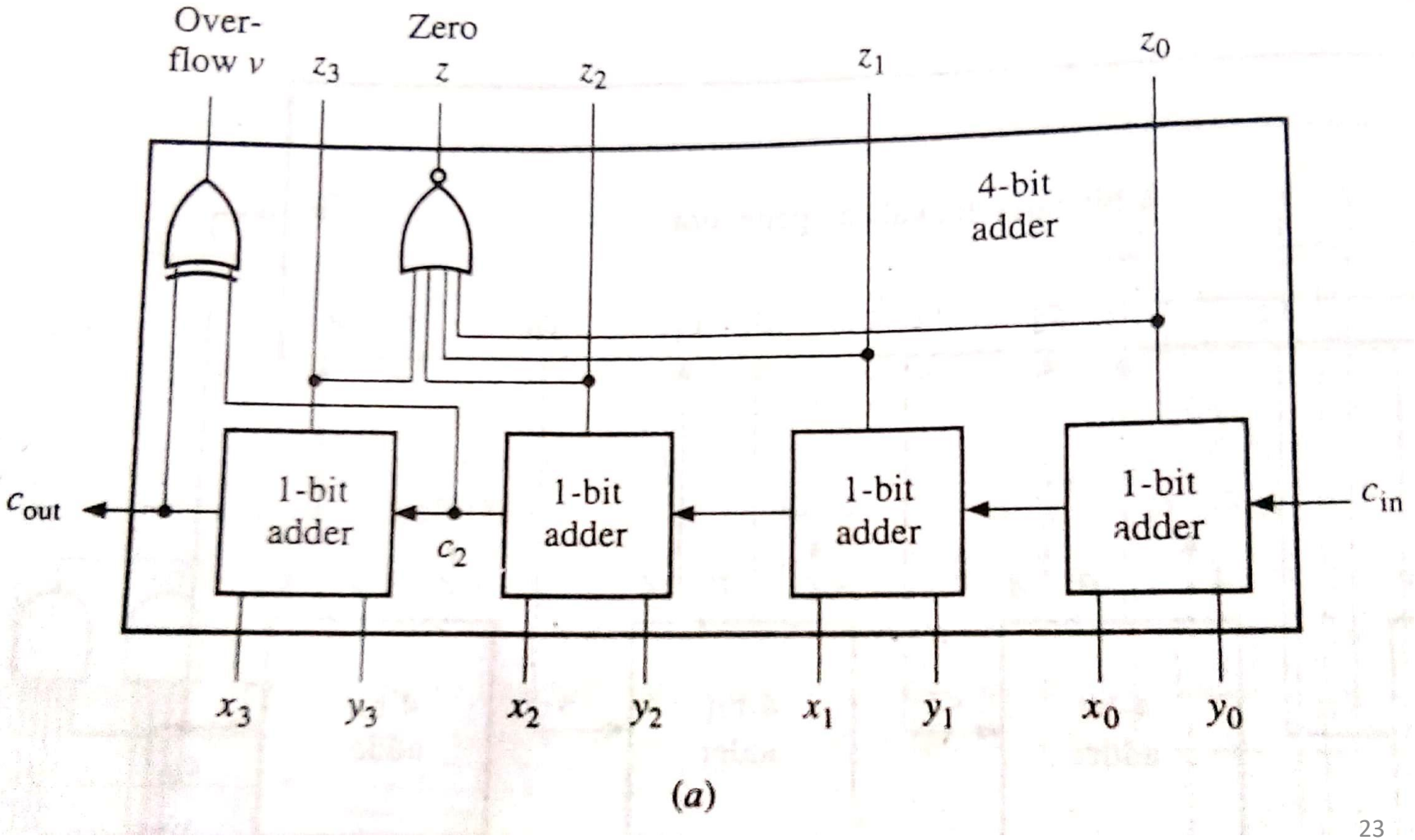


Figure 4.8
A 16-bit adder composed of 4-bit adders linked by carry lookahead.

FIXED-POINT ARITHMETIC

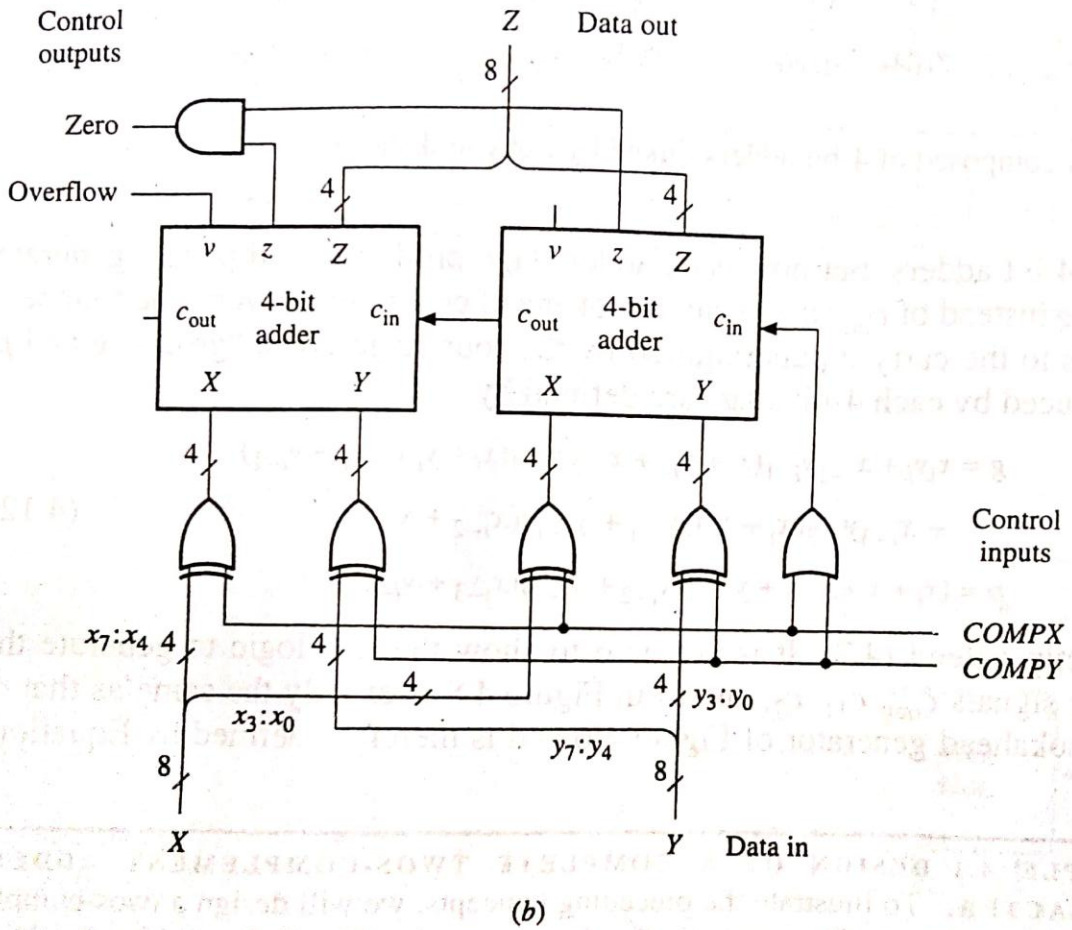
TWOS-COMPLEMENT ADDER-SUBTRACTOR

- Computes three quantities – X+Y, X-Y and Y-X as well as overflow and zero flags..



FIXED-POINT ARITHMETIC

TWOS-COMPLEMENT ADDER-SUBTRACTOR



(b)

Figure 4.9
Low-cost addition and subtraction of twos-complement numbers: (*a*) 4-bit adder module and (*b*) 8-bit adder-subtracter.

FIXED-POINT ARITHMETIC

MULTIPLICATION

- requires substantially more hardware than fixed-point addition.
- implemented by some form of repeated addition.
- Main operation involved are shifting and addition.
- The computation involved in processing one multiplier bit x_j can be described by the register-transfer statement of the form,

$$P_{i+1} := P_i + x_j 2^i Y$$

Where P_i and P_{i+1} are called partial products.

1010	Multiplicand Y
1101	Multiplier $X = x_3x_2x_1x_0$
1010	x_0Y
0000	$x_1 2Y$
1010	$x_2 2^2 Y$
1010	$x_3 2^3 Y$
10000010	Product $P = \sum_{j=0}^{3} x_j 2^j Y$

FIXED-POINT ARITHMETIC

MULTIPLICATION

- $2^i Y$ is equivalent to Y shifted i positions to the left.

1010	Multiplicand Y
1101	Multiplier $X = x_3x_2x_1x_0$
00000000	$P_0 = 0$
1010	x_0Y
00001010	$P_1 = P_0 + x_0Y$
0000	$x_1 2Y$
00001010	$P_2 = P_1 + x_1 2Y$
1010	$x_2 2^2 Y^j$
00110010	$P_3 = P_2 + x_2 2^2 Y$
1010	$x_3 2^3 Y$
10000010	$P_4 = P_3 + x_3 2^3 Y = P$

Figure 4.11
The multiplication of Figure 4.10 modified for machine implementation.

MULTIPLICATION IN MACHINE

MULTIPLICATION OF UNSIGNED NUMBERS – MACHINE IMPLEMENTATION

+6 → 0 1 1 0 [MULTIPLICAND]
+3 → 0 0 1 1 [MULTIPLIER]
6 × 3 = 18 [0 0 0 1 0 0 1 0]

ACTION	ACCUMULATOR [4-BIT]	Q[4-BIT] MULTIPLIER	M[4-BIT] MULTIPLICAND
INITIALIZE CHECK Q[0]	0 0 0 0	0 0 1 [1]	0 1 1 0
IF Q[0] == 1, ADD M RSHIFT → A.Q	0 1 1 0 0 1 1 0 0 0 1 1	0 0 1 1 0 0 0 [1]	
IF Q[0] == 1, ADD M RSHIFT → A.Q	0 1 1 0 1 0 0 1 0 1 0 0	0 0 0 1 1 0 0 [0]	
IF Q[0] == 0, ADD 0 RSHIFT → A.Q	0 0 0 0 0 1 0 0 0 0 1 0	0 1 0 [0]	
IF Q[0] == 0, ADD 0 RSHIFT → A.Q	0 0 0 0 0 0 1 0 0 0 0 1 PRODUCT OUTPUT	0 1 0 0 0 0 1 0 = +18	

ROBERTSON ALGORITHM – 2'S COMPLEMENT MULTIPLIER

TWOS COMPLEMENT MULTIPLICATION :

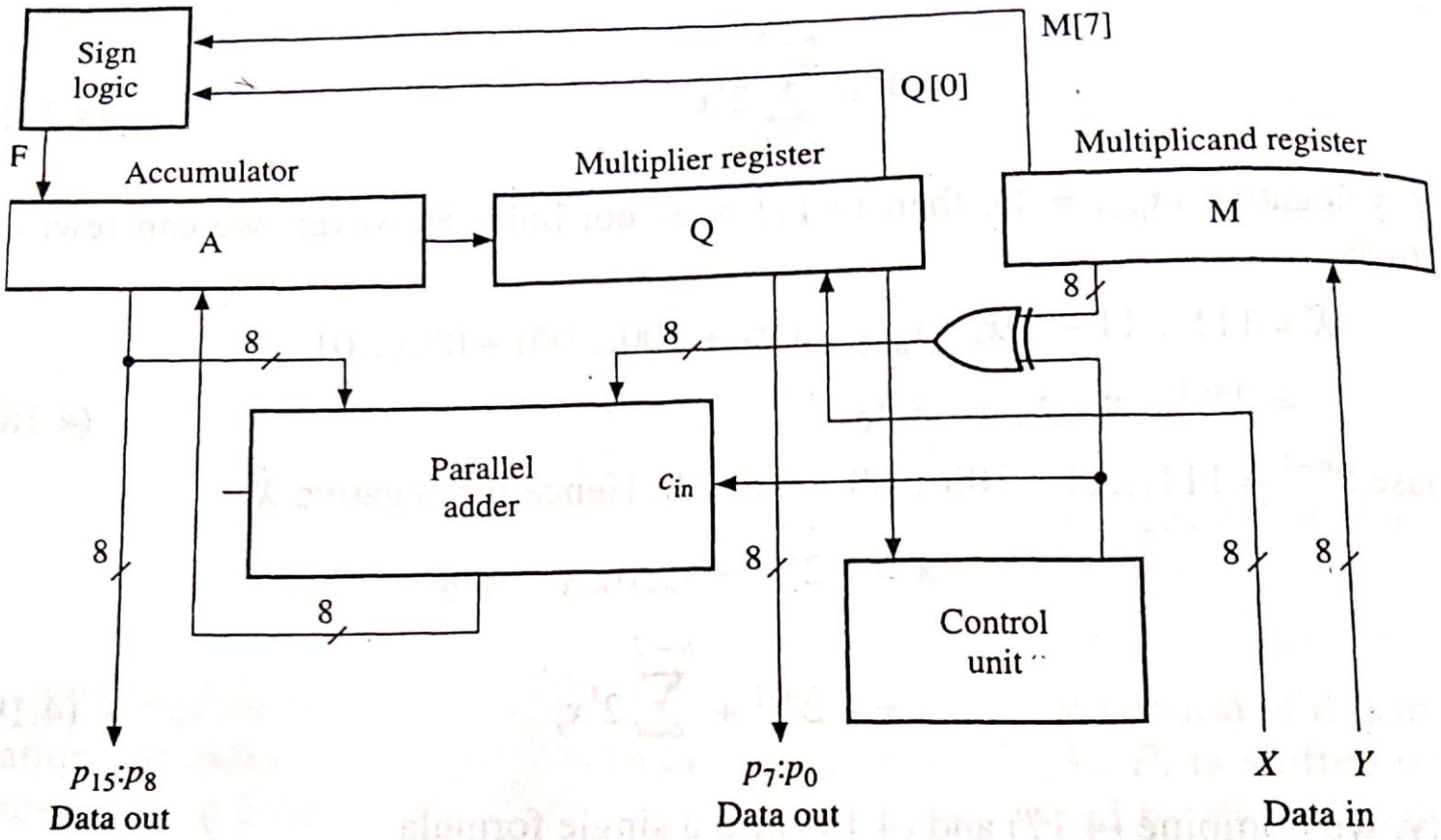


Figure 4.12
The datapath of the twos-complement multiplier.

ROBERTSON ALGORITHM – 2'S COMPLEMENT MULTIPLIER

ROBERTSON ALGORITHM FOR 2'S COMPLIMENT MULTIPLICATION

[MULTIPLICATION FOR SIGNED NUMBERS]

BEGIN : A := 0 , COUNT := 0 , f := 0

INPUT : M := MULTIPLICAND , Q := MULTIPLIER

ADD : $A[3:0] := A[3:0] + (M[3:0] \times Q[0])$;

CALCULATE f_{NEW} : $F_{NEW} := (M[3]$ and $Q[0])$ or f_{OLD} ;

RSHIFT (F.A.Q) : $A[3] := f_{NEW}$, $A[2:0].Q := A.Q[3:1]$, COUNT = COUNT +1

TEST : IF COUNT ≠ 3 THEN GO TO ADD ;

SUBTRACT : $A[3:0] := A[3:0] - (M[3:0] \times Q[0])$;

CALCULATE F_{FINAL} : $F_{FINAL} := M[3]$ xor $Q[0]$;

RSHIFT (F.A.Q) : $A[3] := f_{FINAL}$, $A[2:0].Q := A.Q[3:1]$;

OUTPUT : PRODUCT IN A.Q

ROBERTSON ALGORITHM – 2'S COMPLEMENT MULTIPLIER

ROBERTSON ALGORITHM – EXAMPLE

2's Comp No

$-6 =$ 1 0 1 0

$-3 =$ 1 1 0 1

$-6 \times -3 = +18$ [0 0 0 1 0 0 1 0]

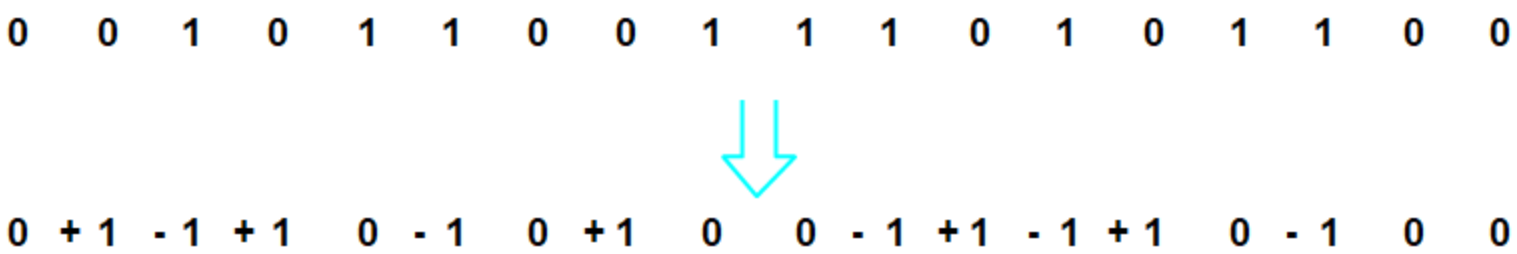
F	ACCUMULATOR A[3:0]	Q[3:0] MULTIPLIER	COUNT	M[3:0] MULTIPLICAND	ACTION
0	0 0 0 0	1 1 0 [1]	0	1 0 1 0	INITIALIZATION
1	1 0 1 0 1 0 1 0 1 1 0 1	0 1 1 [0]	1		ADD A+M IF Q[0] = 1 CALCULATE F_{NEW} RSHIFT (F.A.Q)
1	0 0 0 0 1 1 0 1 1 1 1 0	1 0 1 [1]	2		CHECK COUNT ≠ 3 [T] ADD A+0 IF Q[0] = 0 CALCULATE F_{NEW} RSHIFT (F.A.Q)
1	c 1 0 1 0 [1] 1 0 0 0 1 1 0 0	0 1 0 [1]	3		CHECK COUNT ≠ 3 [T] ADD A+M IF Q[0] = 1 DISCARD CARRY IN 2's COMPLEMENT ADDITION CALCULATE F_{NEW} RSHIFT (F.A.Q)
0	c 0 1 1 0 [1] 0 0 1 0 0 0 0 1	0 0 1 0			CHECK COUNT ≠ 3 [F] A-M IF Q[0] = 1 -M = obtained by 2's Comp of M -M = 0110 A+C(-M) DISCARD CARRY WHILE 2's Comp ADDITION CALCULATE F_{FINAL} RSHIFT (F.A.Q)

PRODUCT OUTPUT = +18

BOOTH ALGORITHM :

TWOS COMPLEMENT MULTIPLICATION :

- employs both addition and subtraction, but it treats positive and negative operands uniformly – no special actions are required for negative numbers.
- In Booth's approach two adjacent bits $x_i x_{i-1}$ are examined in each step.
- If $x_i x_{i-1}$=01, then Y is added to the current partial product P_i, while if $x_i x_{i-1}$=10, Y is subtracted from P_i.
- If $x_i x_{i-1}$=00 or 11, then neither addition or subtraction is performed; only the subsequent arithmetic right shift of P_i takes place.



Booth recoding of a multiplier.

BOOTH ALGORITHM

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

Booth multiplier recoding table.

BOOTH ALGORITHM

$A = 010111 \ (+23)$

$B = 110110 \ (-10 \rightarrow \text{2's compliment of } 110110)$

010111 × 110110

0 1 0 1 1 1 × 0 −1 +1 0 −1 0

23 × −10 (2's compliment of 110110)

0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 0 1 0 0 1 -2's compliment

0 0 0 0 0 0 0 0 0 0

0 0 0 0 1 0 1 1 1

1 1 1 0 1 0 0 1

0 0 0 0 0 0 0

1 1 1 1 0 0 0 1 1 0 1 0 ⇒ (−230)

BOOTH ALGORITHM :

TWOS COMPLEMENT MULTIPLICATION :

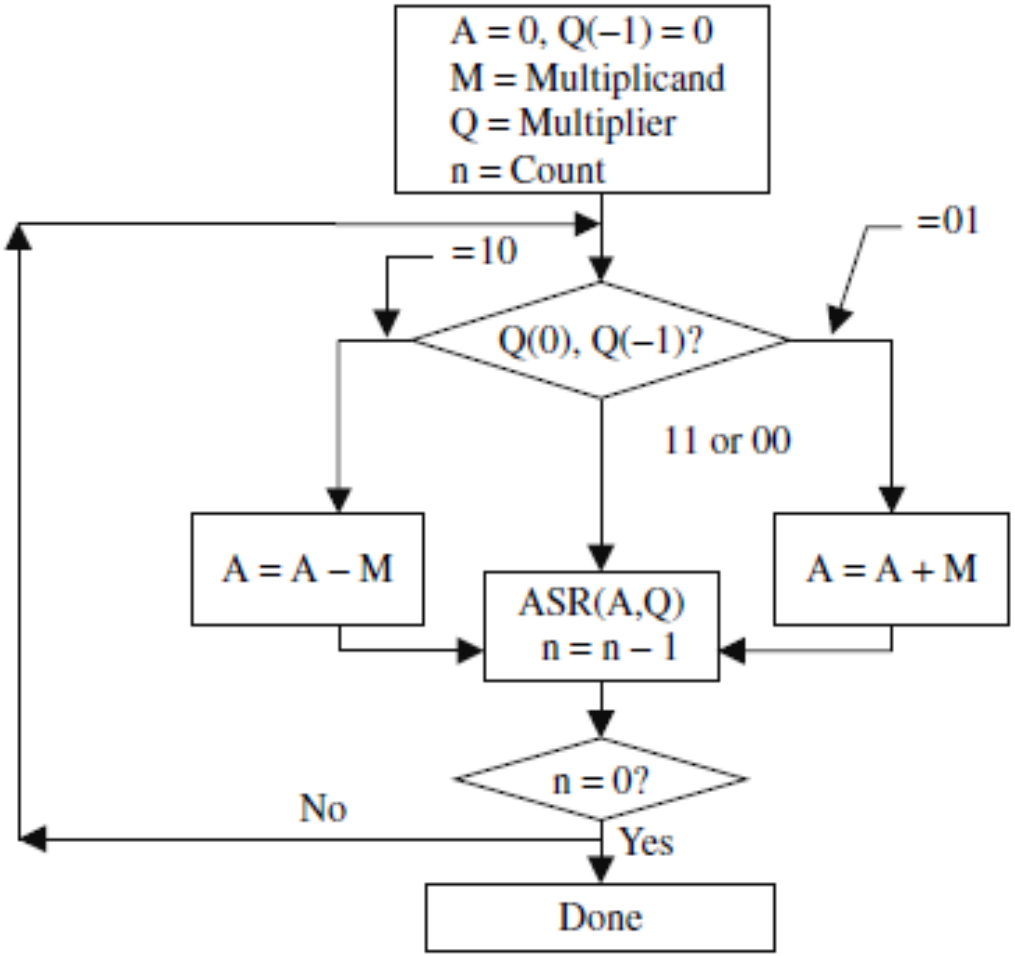


Figure 4.8 Booth's algorithm

BOOTH ALGORITHM :

TWOS COMPLEMENT MULTIPLICATION : BOOTH ALGORITHM

Example Consider the multiplication of the two numbers $M = 0111$ (7) and $Q = 1101$ (-3) and assuming that $n = 4$. The steps needed are tabulated below.

M	A	Q	$Q(-1)$		
0111	0000	1101	0	Initial value	
0111	1001	1101	0	$A = A - M$	
0111	1100	1110	1	ASR	End cycle #1
0111	0011	1110	1	$A = A + M$	
0111	0001	1111	0	ASR	End cycle #2
0111	1010	1111	0	$A = A - M$	
0111	1101	0111	1	ASR	End cycle #3
0111	1110	1011	1	ASR	End cycle #4

-21 (correct result)

BOOTH ALGORITHM :

TWOS COMPLEMENT MULTIPLICATION :

Example Consider the multiplication of the two positive numbers $M = 0111$ (7) and $Q = 0011$ (3) and assuming that $n = 4$. The steps needed are tabulated below.

M	A	Q	$Q(-1)$		
0111	0000	0011	0	Initial value	
0111	1001	0011	0	$A = A - M$	
0111	1100	1001	1	ASR	End cycle #1
0111	1110	0100	1	ASR	End cycle #2
0111	0101	0100	1	$A = A + M$	
0111	0010	1010	0	ASR	End cycle #3
0111	0001	0101	1	ASR	End cycle #4

+21 (correct result)

BOOTH ALGORITHM : HARDWARE IMPLEMENTATION

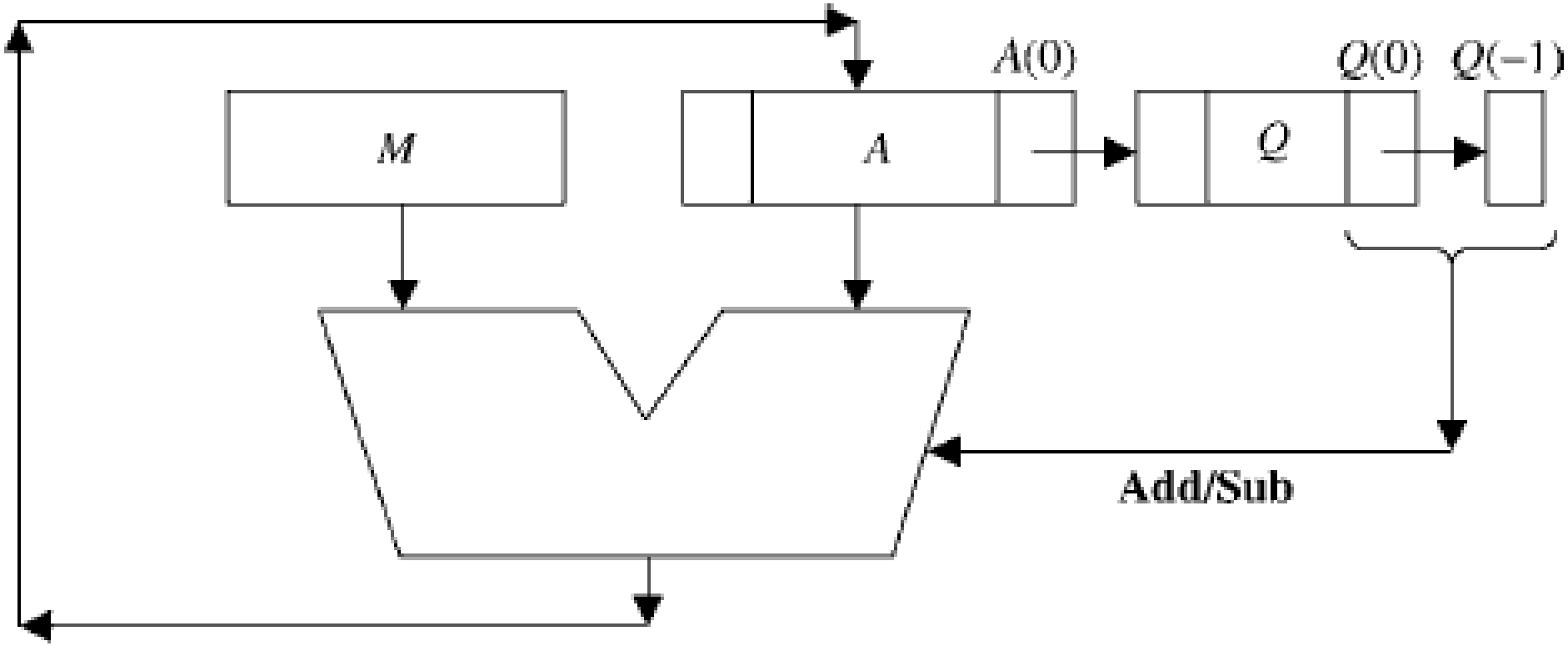
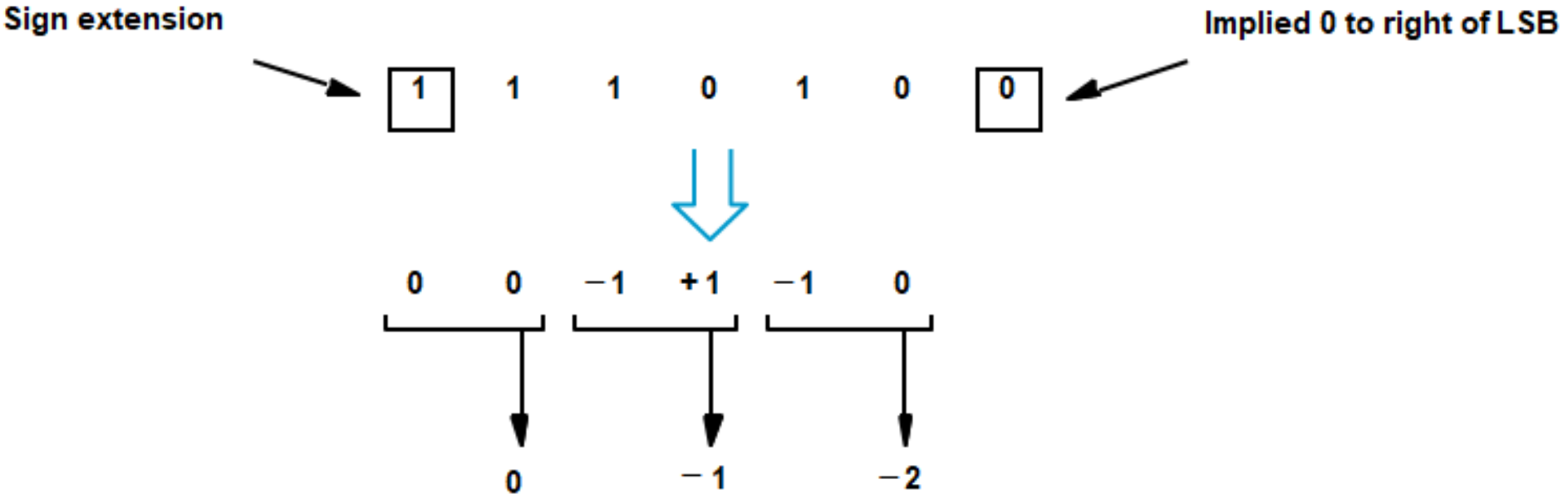


Figure 4.9 Hardware structure implementing Booth's algorithm

MODIFIED BOOTH ALGORITHM



Example of bit-pair recoding derived from Booth recoding

MODIFIED BOOTH ALGORITHM

Booth recoding table for radix-4

Multiplier Bits Block			Recoded 1-bit pair		2 bit booth	
i+1	i	i-1	i+1	i	Multiplier Value	Partial Product
0	0	0	0	0	0	Mx0
0	0	1	0	1	1	Mx1
0	1	0	1	-1	1	Mx1
0	1	1	1	0	2	Mx2
1	0	0	-1	0	-2	Mx-2
1	0	1	-1	1	-1	Mx-1
1	1	0	0	-1	-1	Mx-1
1	1	1	0	0	0	Mx0

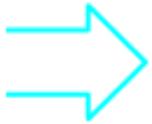
MODIFIED BOOTH ALGORITHM

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

Table of multiplicand selection decisions

MODIFIED BOOTH ALGORITHM

```
    0  1  1  0  1   (+13)
    1  1  0  1  0   (- 6)
```



```
                          0  1  1  0  1
                          0 -1 +1 -1  0
                    ------------------
    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
    ------------------------------------
    1  1  1  0  1   1  0  0  1  0   (- 78)
```

```
                          0  1  1  0  1
                          0    -1    -2
                    ------------------
    1  1  1  1  1   0  0  1  1  0
    1  1  1  1  0   0  1  1
    0  0  0  0  0   0
    ------------------------------------
    1  1  1  0  1   1  0  0  1  0
```

FIXED-POINT ARITHMETIC

COMBINATIONAL ARRAY MULTIPLIERS : TWO DIMENSIONAL RIPPLE-CARRY ADDER

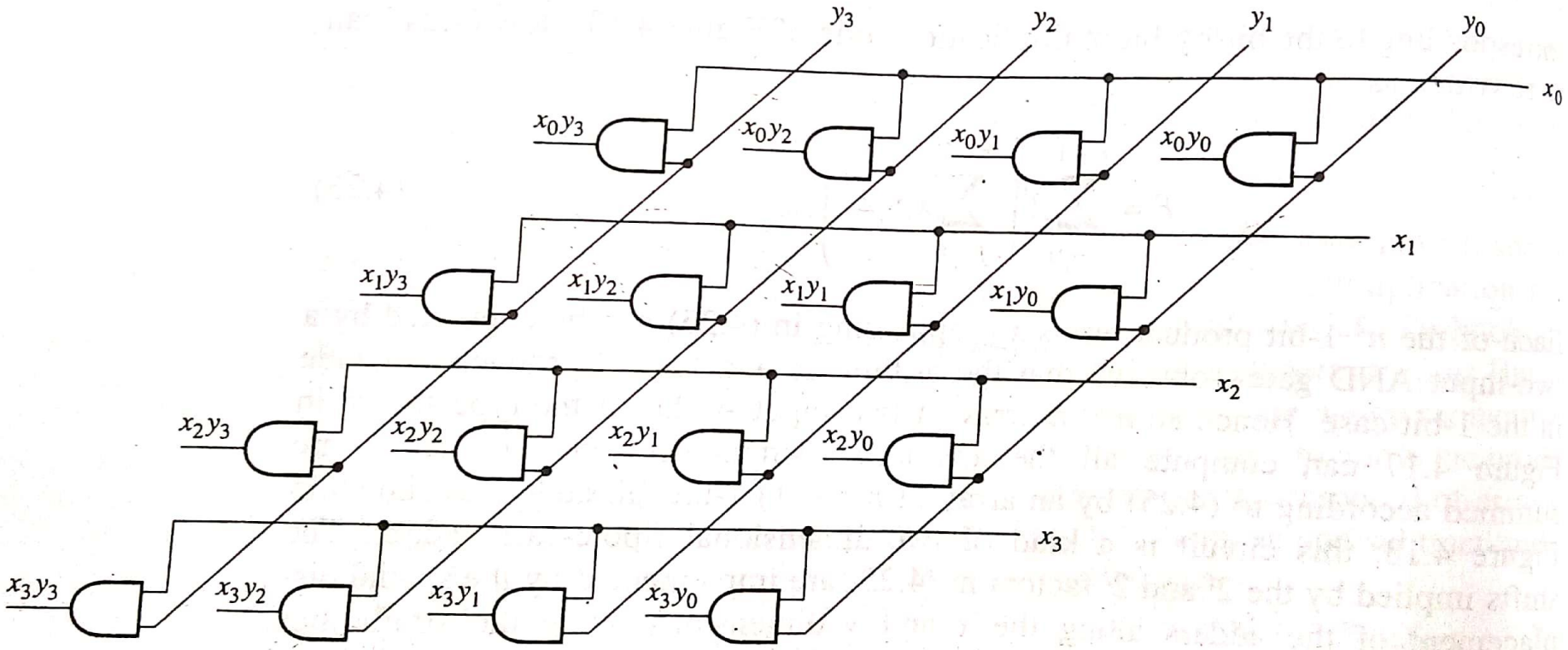


Figure 4.17
AND array for 4 × 4-bit unsigned multiplication.

FIXED-POINT ARITHMETIC

COMBINATIONAL ARRAY MULTIPLIERS : TWO DIMENSIONAL RIPPLE-CARRY ADDER

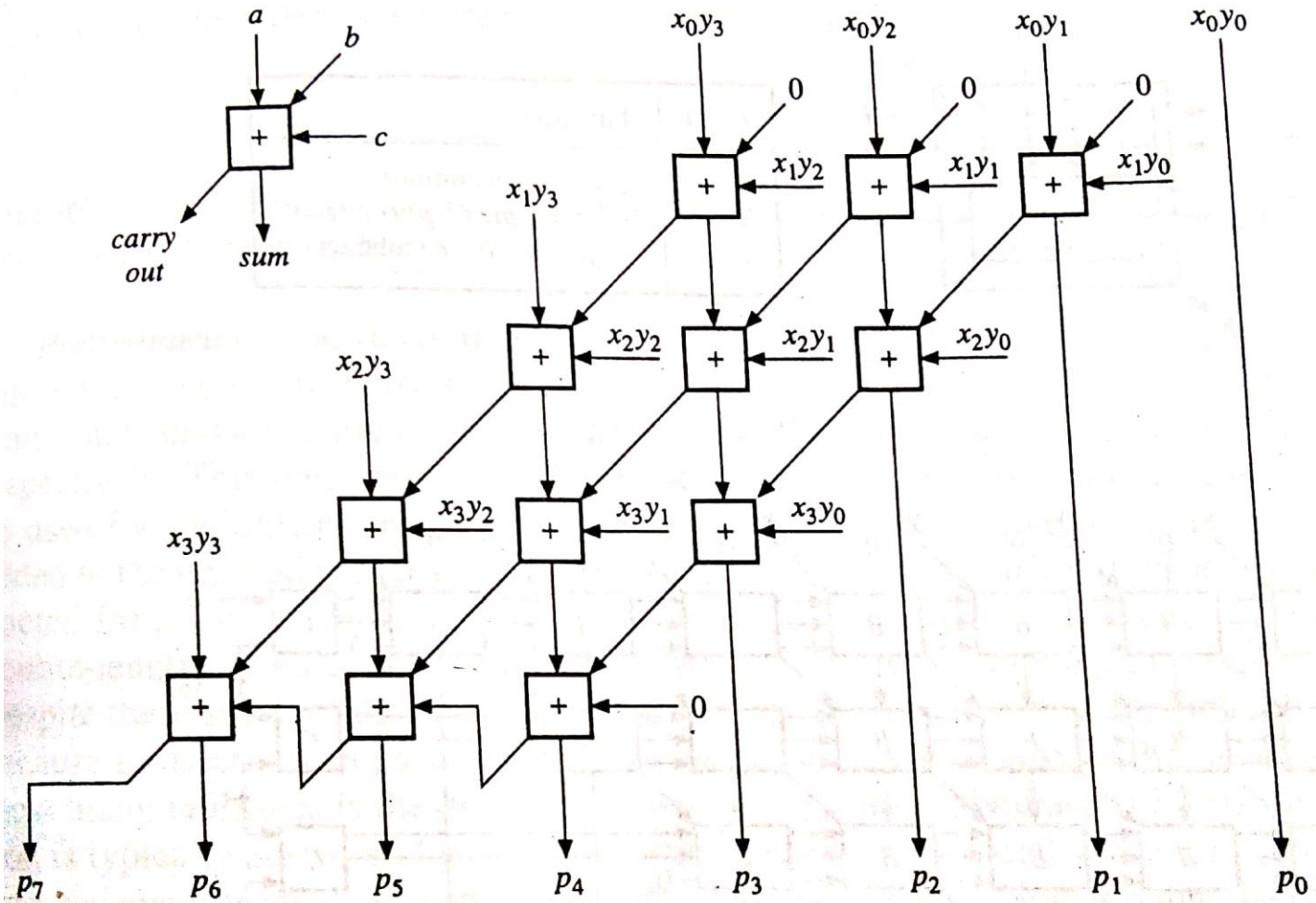


Figure 4.18
Full-adder array for 4 × 4-bit unsigned multiplication.

FIXED-POINT ARITHMETIC

DIVISION

- Divisor V
- Dividend D
- Quotient Q
- Remainder R (required o be less than V, $0 \leq R < V$)
- $D = Q \times V + R$
- $D/V = Q + R/V$, here R/V is a small quantity representing the error in using Q alone to represent D/V; this error is zero if R=0.

FIXED-POINT ARITHMETIC

DIVISION

	0111	Quotient $Q = q_3q_2q_1q_0$
Divisor $V = 101$	100110	Dividend $D = R_0$
	000	q_3V
	100110	R_1
	101	$q_2 2^{-1} V$
	10010	R_2
	101	$q_1 2^{-2} V$
	1000	R_3
	101	$q_0 2^{-3} V$
	011	R_4 = remainder R

Figure 4.21
Typical pencil-and-paper method for division of unsigned numbers.

FIXED-POINT ARITHMETIC

DIVISION

Divisor V			Quotient Q
101	100110	Dividend $D = 2R_0$	
	000	q_3V	0
	100110	R_1	
	1001100	$2R_1$	
	101	q_2V	01
	100100	R_2	
	1001000	$2R_2$	
	101	q_1V	011
	100000	R_3	
	1000000	$2R_3$	
	101	q_0V	0111
	011000	$R_4 = 2^3R$	

Figure 4.22
The division of Figure 4.21 modified for machine implementation.

FIXED-POINT ARITHMETIC

DIVISION

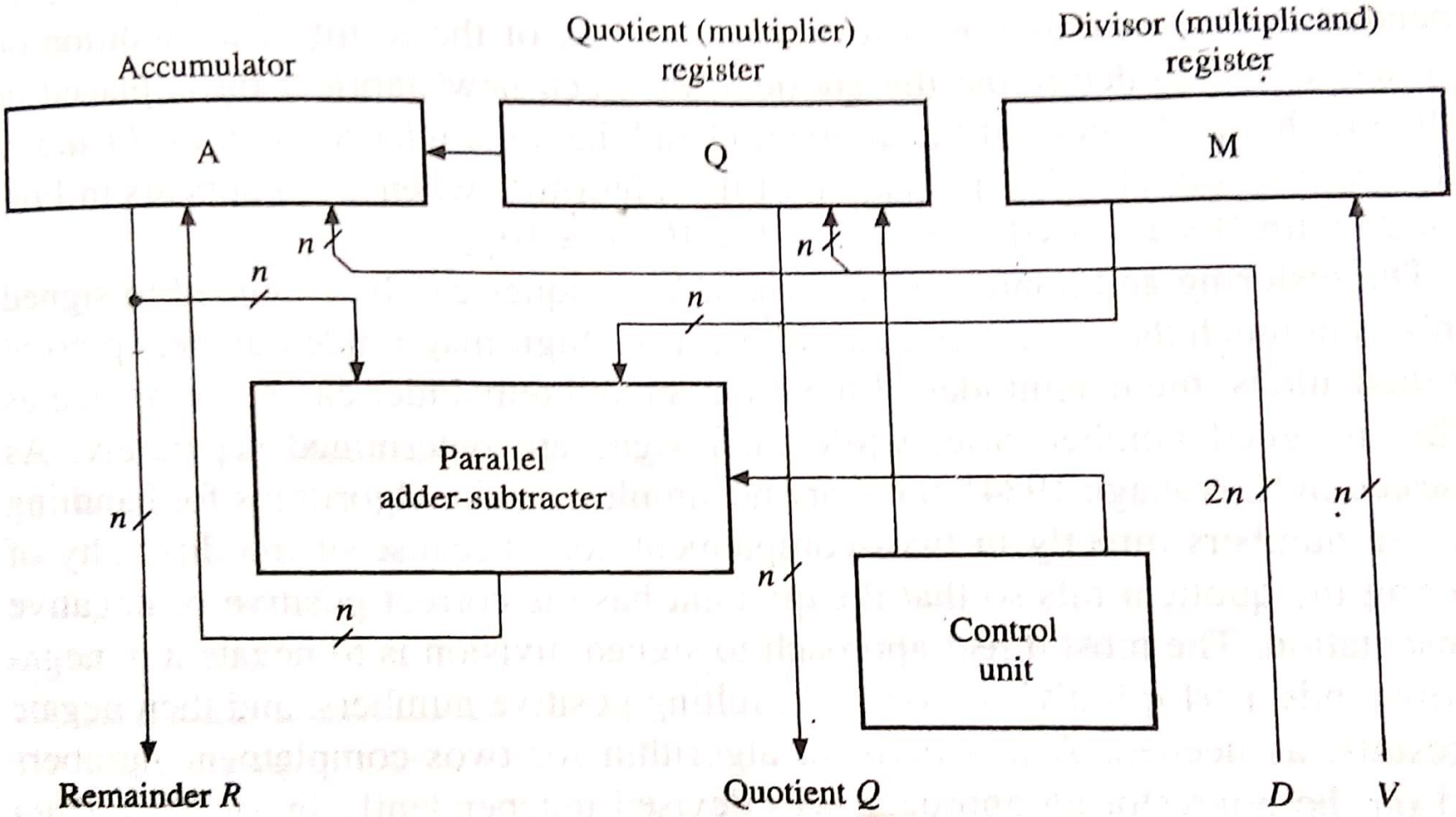
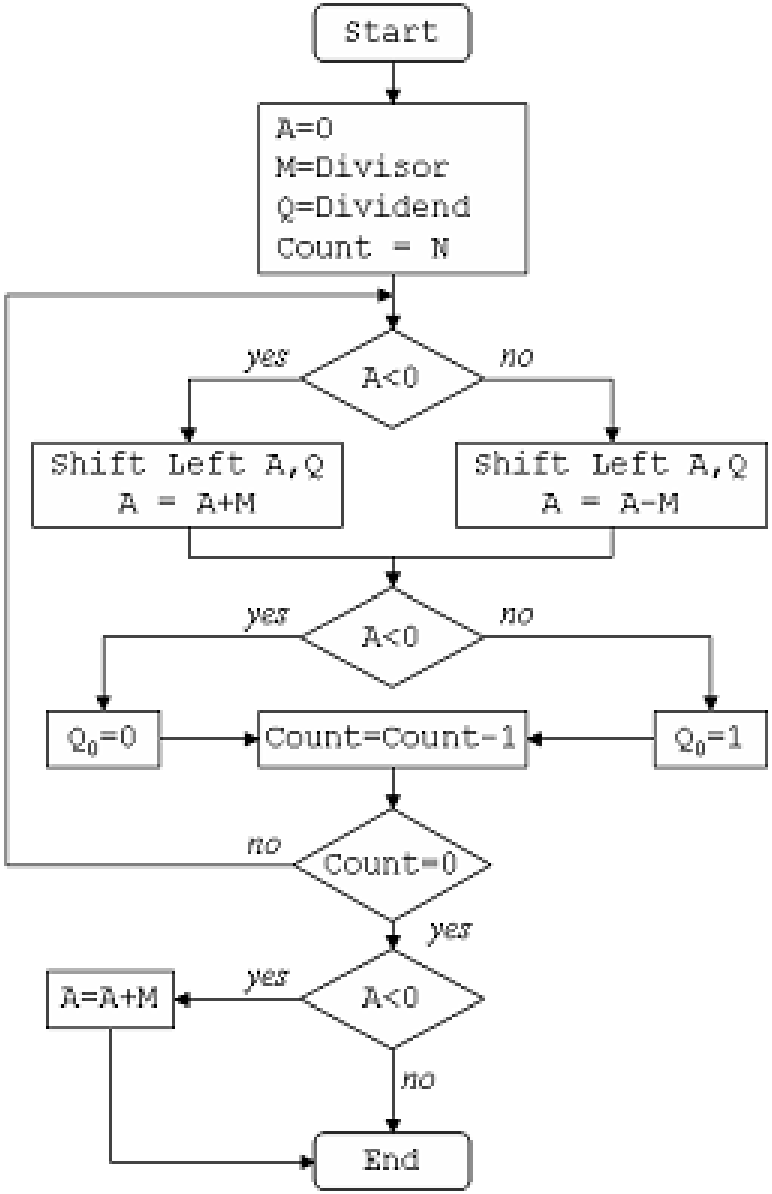


Figure 4.23
The datapath of a sequential *n*-bit binary divider.

NON-RESTORING DIVISION ALGORITHM



NON-RESTORING DIVISION ALGORITHM : EXAMPLE (11/3)

Dividend =11

Divisor =3

-M =11101

N	M	A	Q	ACTION
4	00011	00000	1011	Start
		00001	011_	Left shift AQ
		11110	011_	A=A-M
3		11110	0110	Q[0]=0
		11100	110_	Left shift AQ
		11111	110_	A=A+M
2		11111	1100	Q[0]=0
		11111	100_	Left Shift AQ

NON-RESTORING DIVISION ALGORITHM : EXAMPLE (11/3)

Dividend =11

Divisor =3

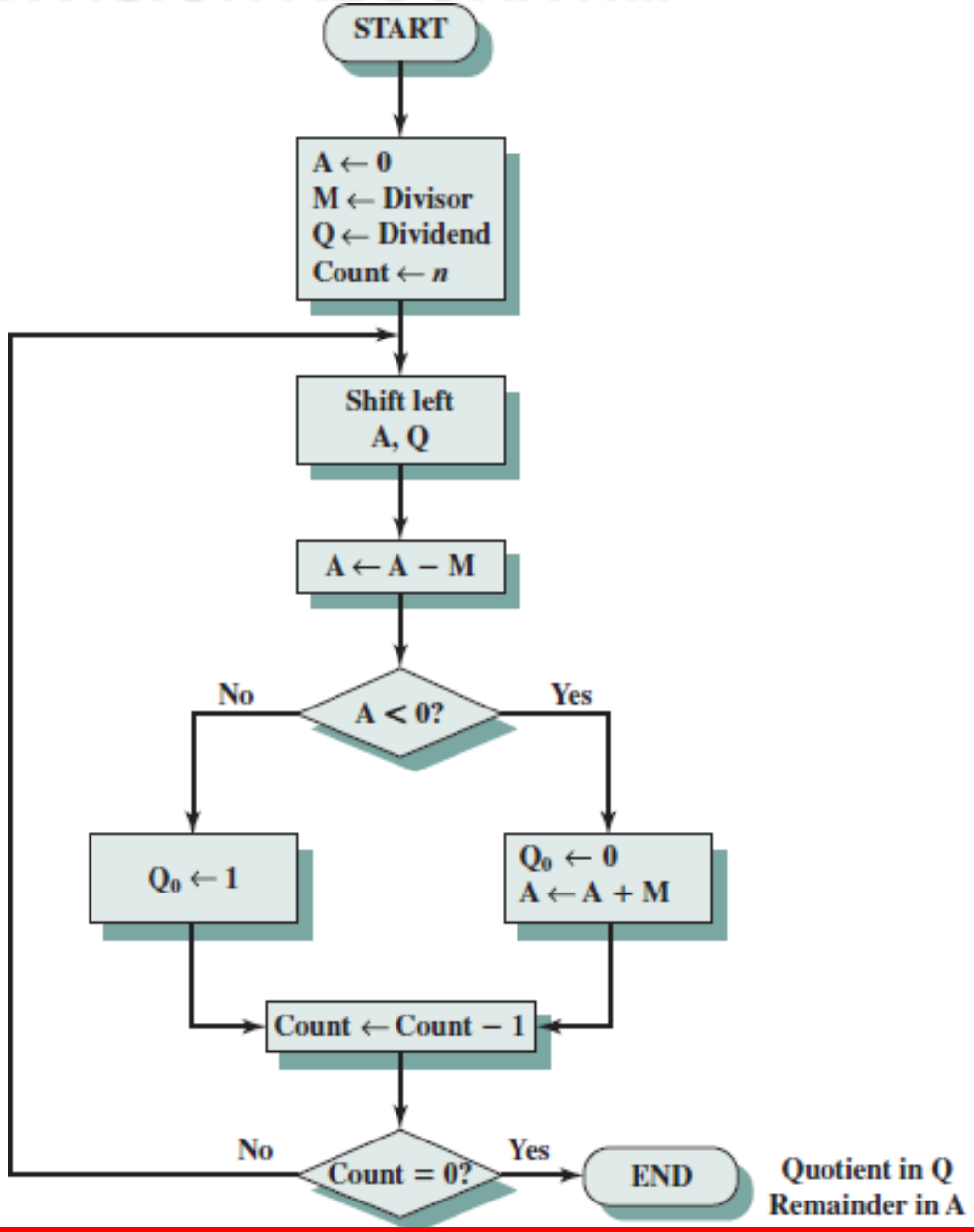
-M =11101

N	M	A	Q	ACTION
2		11111	1100	Q[0]=0
		11111	100_	Left Shift AQ
		00010	100_	A=A+M
1		00010	1001	Q[0]=1
		00101	001_	Left Shift AQ
		00010	001_	A=A-M
0		00010	0011	Q[0]=1

Quotient = 3 (Q)

Remainder = 2 (A)

RESTORING DIVISION ALGORITHM

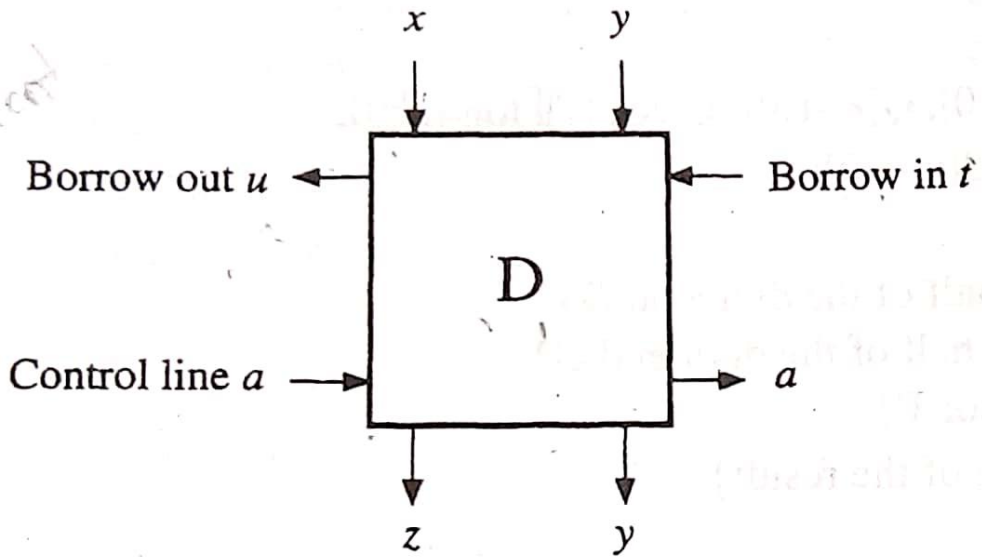


RESTORING DIVISION ALGORITHM : EXAMPLE (7/3)

A	Q	
0000	0111	Initial value
0000	1110	Shift
1101		Use twos complement of 0011 for subtraction
1101		Subtract
0000	1110	Restore, set $Q_0 = 0$
0001	1100	Shift
1101		
1110		Subtract
0001	1100	Restore, set $Q_0 = 0$
0011	1000	Shift
1101		
0000	1001	Subtract, set $Q_0 = 1$
0001	0010	Shift
1101		
1110		Subtract
0001	0010	Restore, set $Q_0 = 0$

FIXED-POINT ARITHMETIC

COMBINATIONAL ARRAY DIVIDERS:



a	Function
0	$uz = x$ *minus* y *minus* t
1	$z = x$

Figure 4.26
A cell D for array implementation of restoring division.

FIXED-POINT ARITHMETIC

COMBINATIONAL ARRAY DIVIDERS:

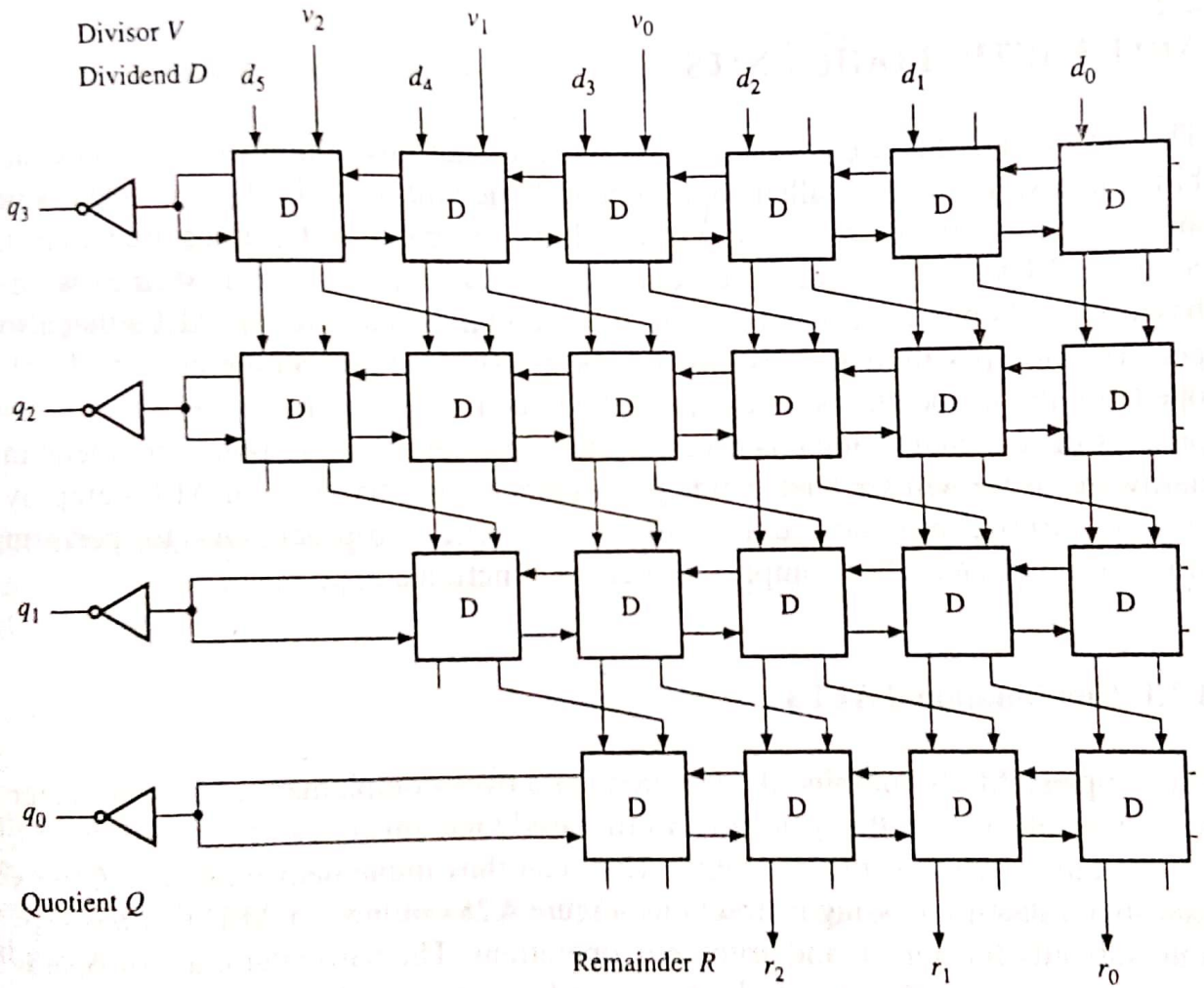


Figure 4.27
A divider array for 3-bit unsigned numbers using the cell D of Figure 4.26.

ARITHMETIC LOGIC UNITS

- The various circuits used to execute data processing instructions are usually combined in a single circuit called an **arithmetic-logic unit** or **ALU.**
- Simple ALUs that perform fixed-point addition and subtraction, as well as word based logical operations, can be realized by combinational circuits.
- Some processors having fixed-point ALUs employ special-purpose auxiliary units called **arithmetic (co)processors** to perform floating-point and other complex numerical functions.

ARITHMETIC LOGIC UNITS: COMBINATIONAL ALUs

- The simplest ALUs combine the functions of a twos-complement adder-subtractor with those of a circuit that generates word-based logic functions of the form f(X,Y) – AND, XOR and NOT.
- implement most of a CPU's fixed-point data processing instructions.
- M – mode control (logical or arithmetic)
- S – select line (16 operations – logical or arithmetic)
- The arithmetic operations can be X+Y, X-Y, Y-X, X+1 (increment), X-1(decrement) and so on.
- The logical operations can be obtained by generating all four minterms of $f(x_i,y_i)$, namely,

$m_3=x_i\ y_i$ $m_2=x_i\ y_i'$ $m_1=x_i'y_i$ $m_0=x_i'y_i'$

$f(x_i,y_i) = m_3S_3 + m_2S_2 + m_1S_1 + m_0S_0$

$= x_iy_iS_3 + x_iy_i'S_2 + x_i'y_iS_1 + x_i'y_i'S_0$

ARITHMETIC LOGIC UNITS: COMBINATIONAL ALUs

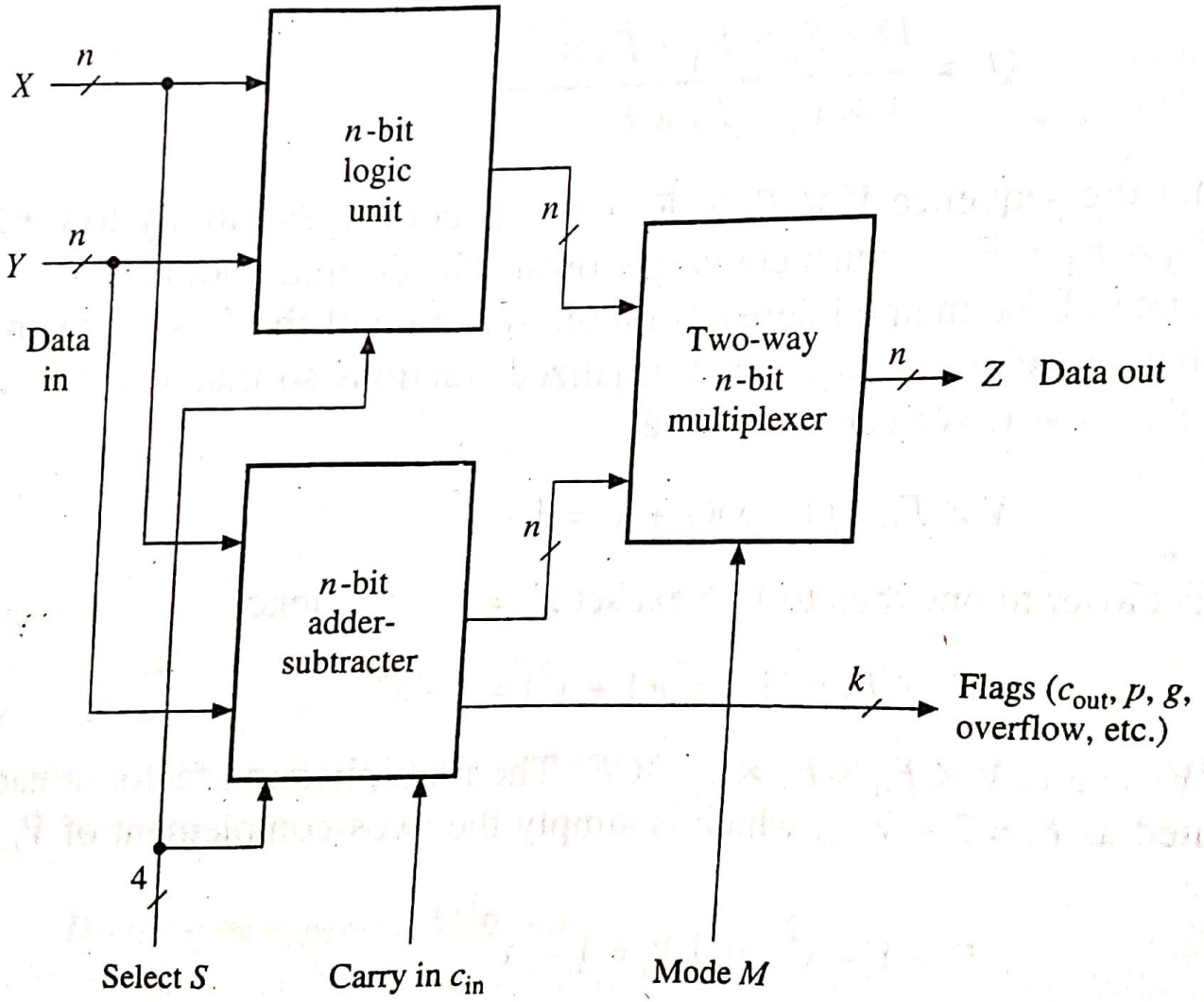


Figure 4.28
A basic n-bit arithmetic-logic unit (ALU).

ARITHMETIC LOGIC UNITS: COMBINATIONAL ALUs

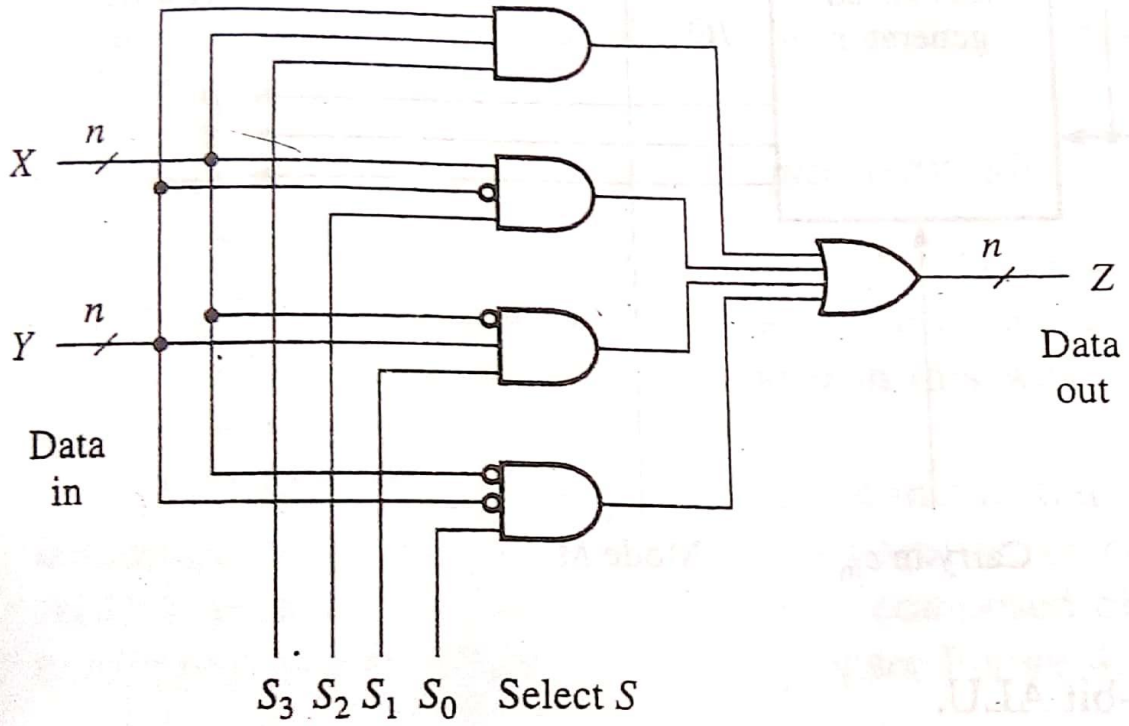


Figure 4.29
An n-bit logic unit that realizes all 16 two-variable functions.

ARITHMETIC LOGIC UNITS: COMBINATIONAL ALUs

- Despite its conceptual simplicity, the ALU is more expensive and slower.
- For n=4, the logic subunit employs about 25 gates and invertors.
- arithmetic subunit with carry lookahead requires around 60 gates.
- The multiplexer also requires additional gates.
- The complete 4-bit ALU can therefore be expected to contain more than 100 gates of various kinds and have a depth 9 or so.

ARITHMETIC LOGIC UNITS: SEQUENTIAL ALUs

- Even though multiplication and division can be implemented by combinational logic, it is generally impractical to merge these operations with addition and subtraction into a single, combinational ALU.
- Combinational multipliers and dividers are costly in terms of hardware.
- They are much slower than addition and subtraction circuits, a consequence of their many logic levels.
- A n-bit combinational multiplier or divider is typically composed of n or more levels of add-subtract logic, making multiplication and division at least n times slower than addition or subtraction.
- The number of gates in the multiply-divide logic is also greater by a factor of about n.
- When n is very small, complete ALUs are usually constructed from low-cost sequential circuits where add and subtract each take one clock cycle, while multiplication and division are multicycle operations.

ARITHMETIC LOGIC UNITS: SEQUENTIAL ALUs

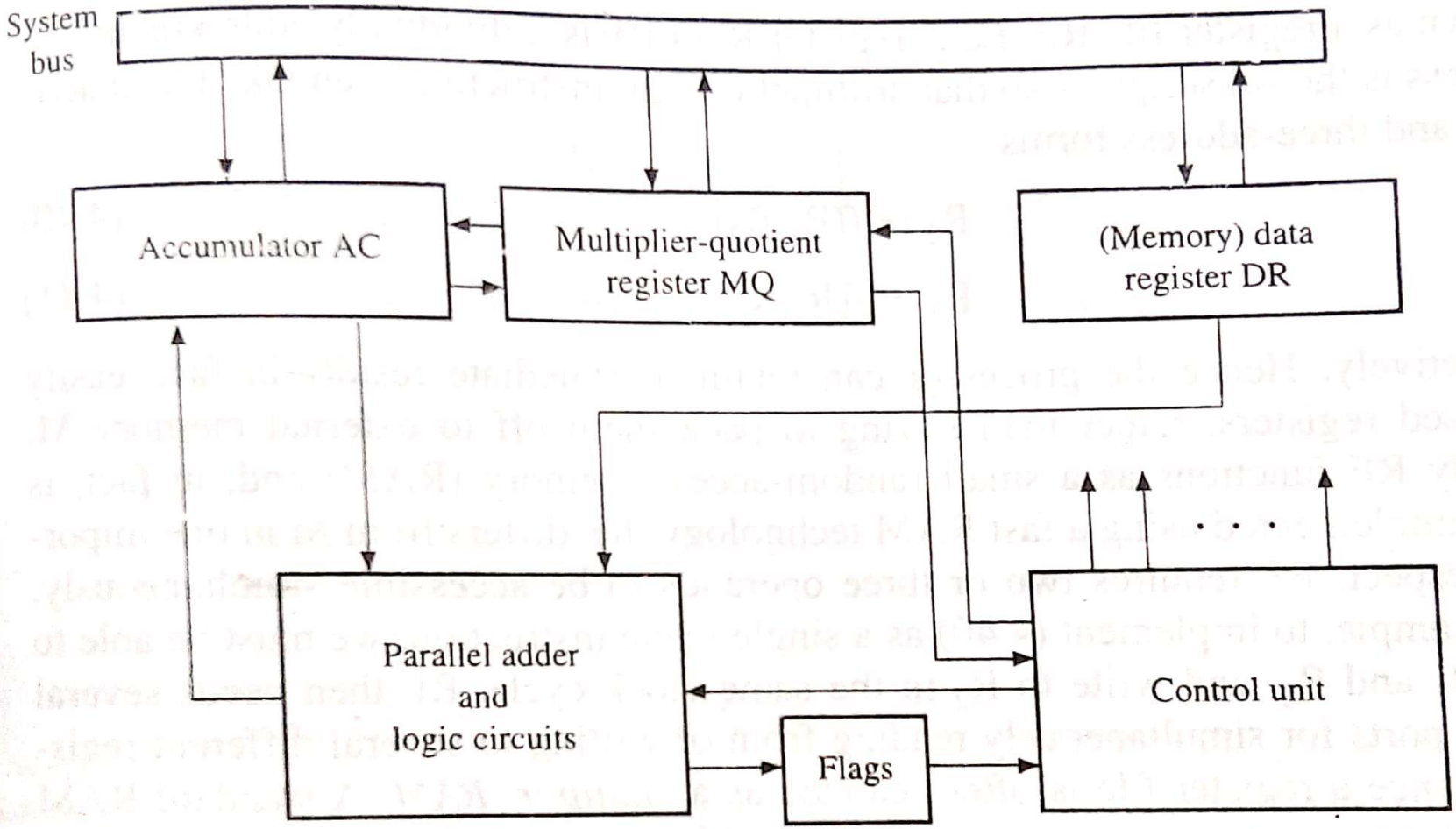


Figure 4.32
Structure of a basic sequential ALU.

ARITHMETIC LOGIC UNITS: SEQUENTIAL ALUs

- Minimizes hardware costs
- found in IAS computer and in many computers built after IAS.
- It is intended to implement multiplication and division using one of the sequential digit-by-digit shift-and add/subtract algorithms.
- Three one-word registers are used for operand storage : the accumulator (AC), the multiplier-quotient register MQ, and the data register DR.
- AC and MQ are organized as a single register AC.MQ capable of left and right shifting.
- MQ- stores multiplier during multiplication and the quotient during division.
- DR – stores the multiplicand or divisor.
- AC.MQ – result (product or quotient and reminder)

ARITHMETIC LOGIC UNITS: SEQUENTIAL ALUs

- The role of registers is defined concisely as folllows,

Addition	AC := AC + DR
Subtraction	AC := AC – DR
Multiplication	AC.MQ := DR × MQ
Division	AC.MQ := MQ/DR
AND	AC := AC ***and*** DR
OR	AC := AC ***or*** DR
EXCLUSIVE-OR	AC := AC ***xor*** DR
NOT	AC := ***not***(AC)

ARITHMETIC LOGIC UNITS: SEQUENTIAL ALUs

Register Files:

- Modern CPUs retain special registers like the multiplier-quotient register MQ for multiplication and division, but the accumulator AC and data register DR are usually replaced by a set of general purpose registers R_0:R_{m-1} known as a **register file RF.**
- Each register R_i in RF is individually addressable.
- arithmetic-logic instructions can take the generic two and three address forms,

$$R_2 := f(R_1, R_2)$$

$$R_3 := f(R_1, R_2)$$

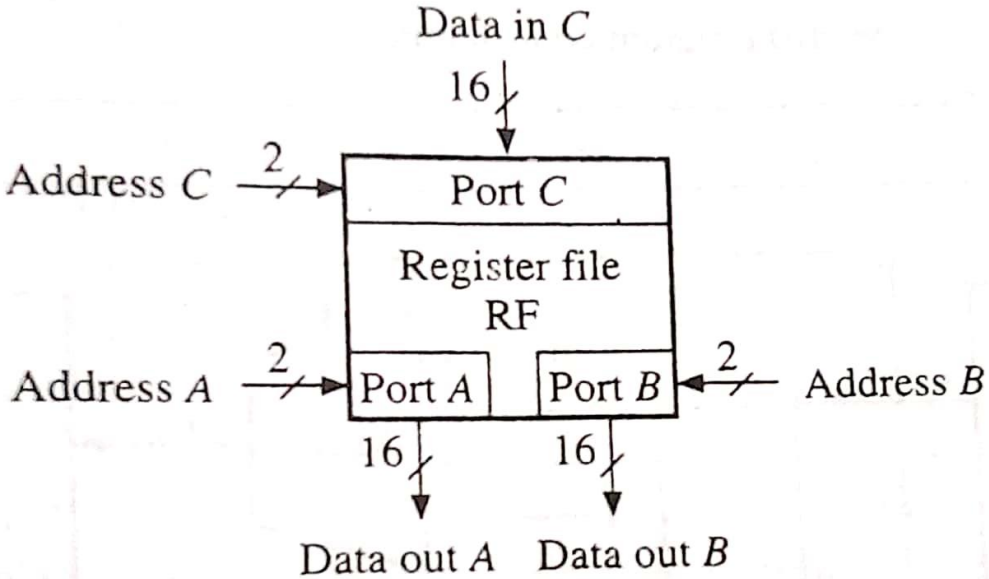
- Register file is implemented using a fast RAM technology.
- RF differs from M in one important aspect : RF requires two or three operands to be accessible simultaneously.
- For example, to implement $R_2 := f(R_1, R_2)$ as a single cycle instruction, we must be able to read R_1 and R_2, and write to R_2 in the same clock cycle.

ARITHMETIC LOGIC UNITS: SEQUENTIAL ALUs

Register Files:

➢RF then needs several access ports for simultaneously reading from or writing to several different registers. Hence a register file is often realized as a **multiport RAM**.

➢ Three port register file that supports simultaneous reads from two ports A and B, while writing can take place via a third port C.



(a)

ARITHMETIC LOGIC UNITS: SEQUENTIAL ALUs

Register Files:

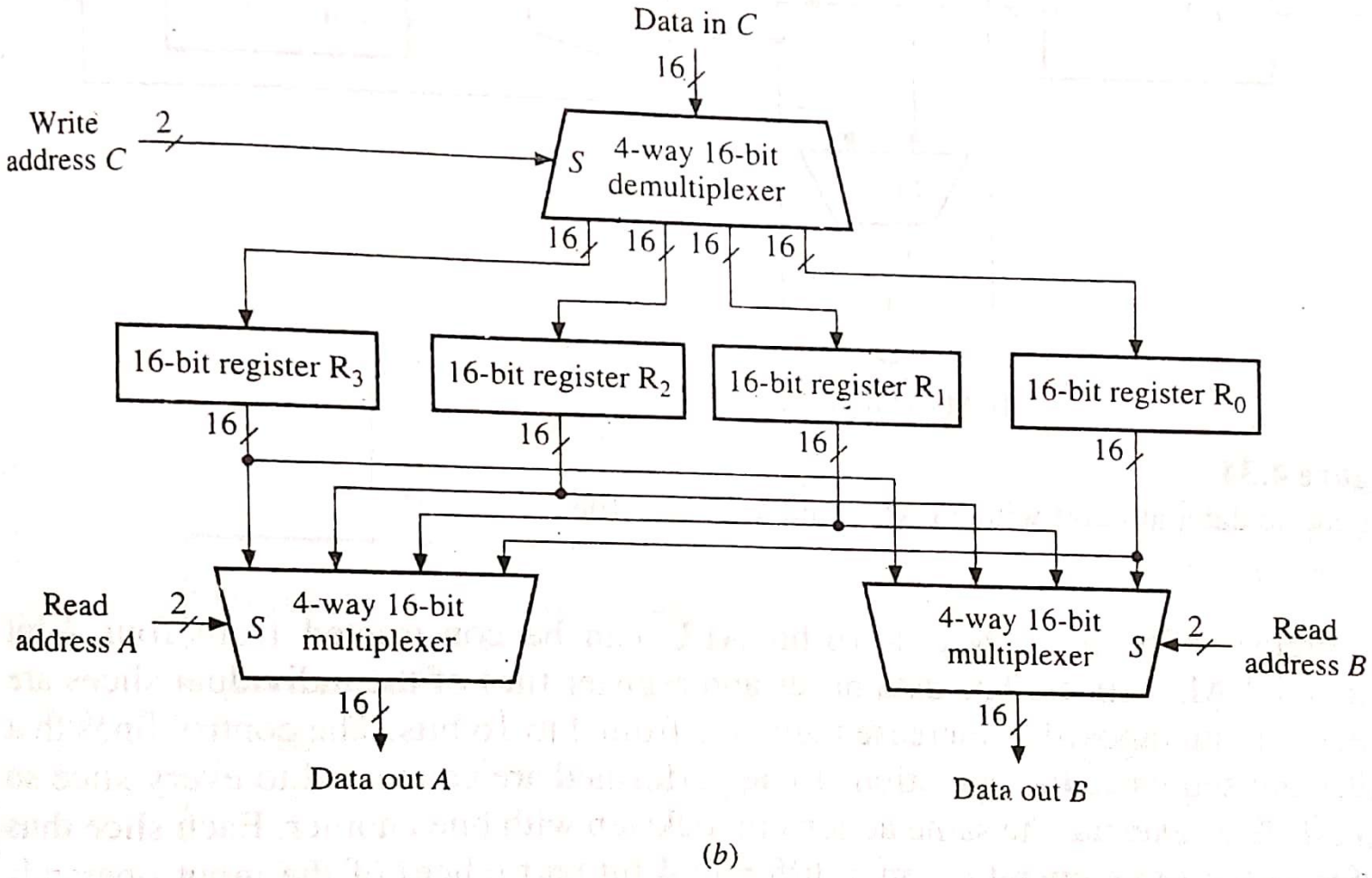


Figure 4.33
A register file with three access ports: (*a*) symbol and (*b*) logic diagram.

ARITHMETIC LOGIC UNITS: SEQUENTIAL ALUs

Register Files:

➢ Representative datapath unit for implementing logical and fixed-point operations; it is often referred to as an **integer** or **fixed-point unit**.

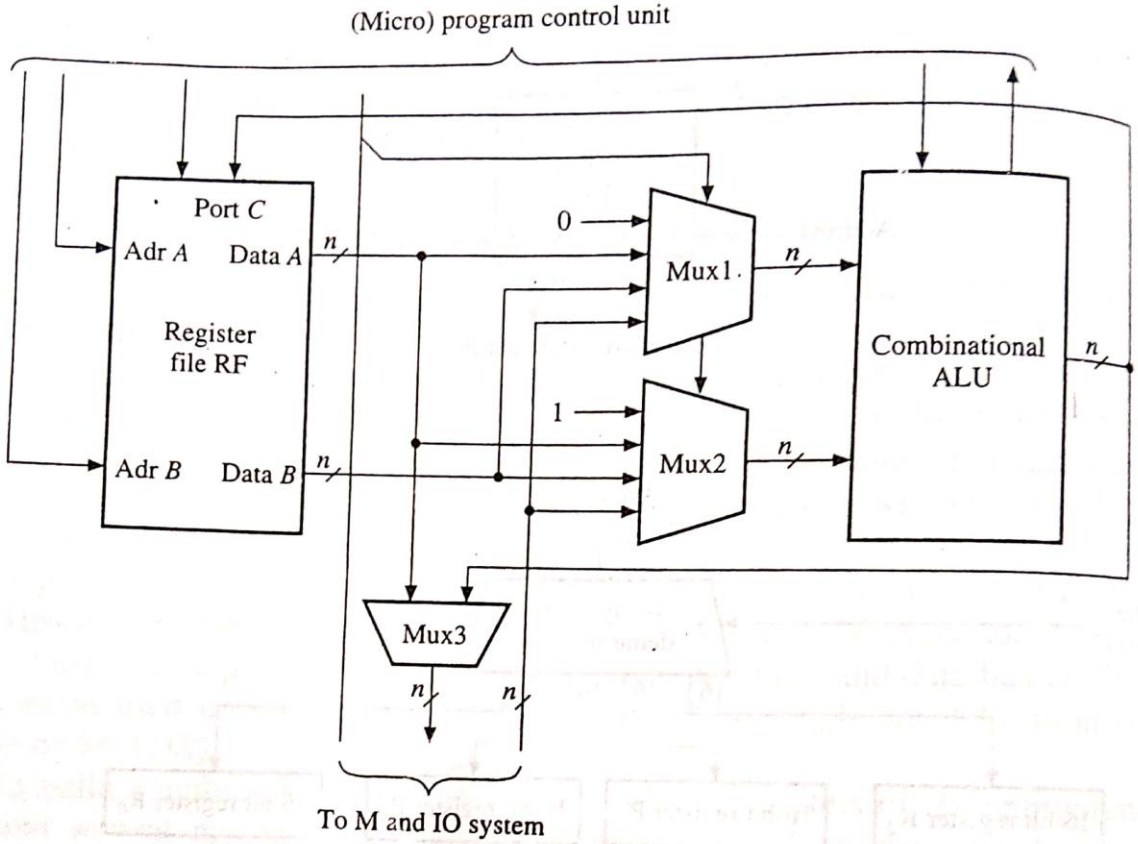


Figure 4.34
A generic datapath unit with an ALU and a register file.

ARITHMETIC LOGIC UNITS: SEQUENTIAL ALUs

ALU expansion:

- feasible to manufacture an entire sequential ALU for fixed-point m-bit numbers on a single IC chip.
- ALU can be easily expanded to handle operands of size n=km or indeed any word size n>m, in two ways:

1. **Spatial expansion** : Connect k copies of the m-bit ALU in the manner of a ripple carry adder form a single ALU capable of processing km-bit words directly. The resulting array-like circuit is said to be **bit sliced** because each component ALU concurrently processes a separate "slice" of m bits from each km-bit operand.
2. **Temporal expansion** : Use one copy of the m-bit ALU chip in the manner of a serial adder to perform an operation on km-bit words in k consecutive steps (clock cycles). In each step the ALU processes a separate m-bit slice of each operand. This processing is called **multicycle** or **multiple-precision processing**.

ARITHMETIC LOGIC UNITS: SEQUENTIAL ALUs

ALU expansion:

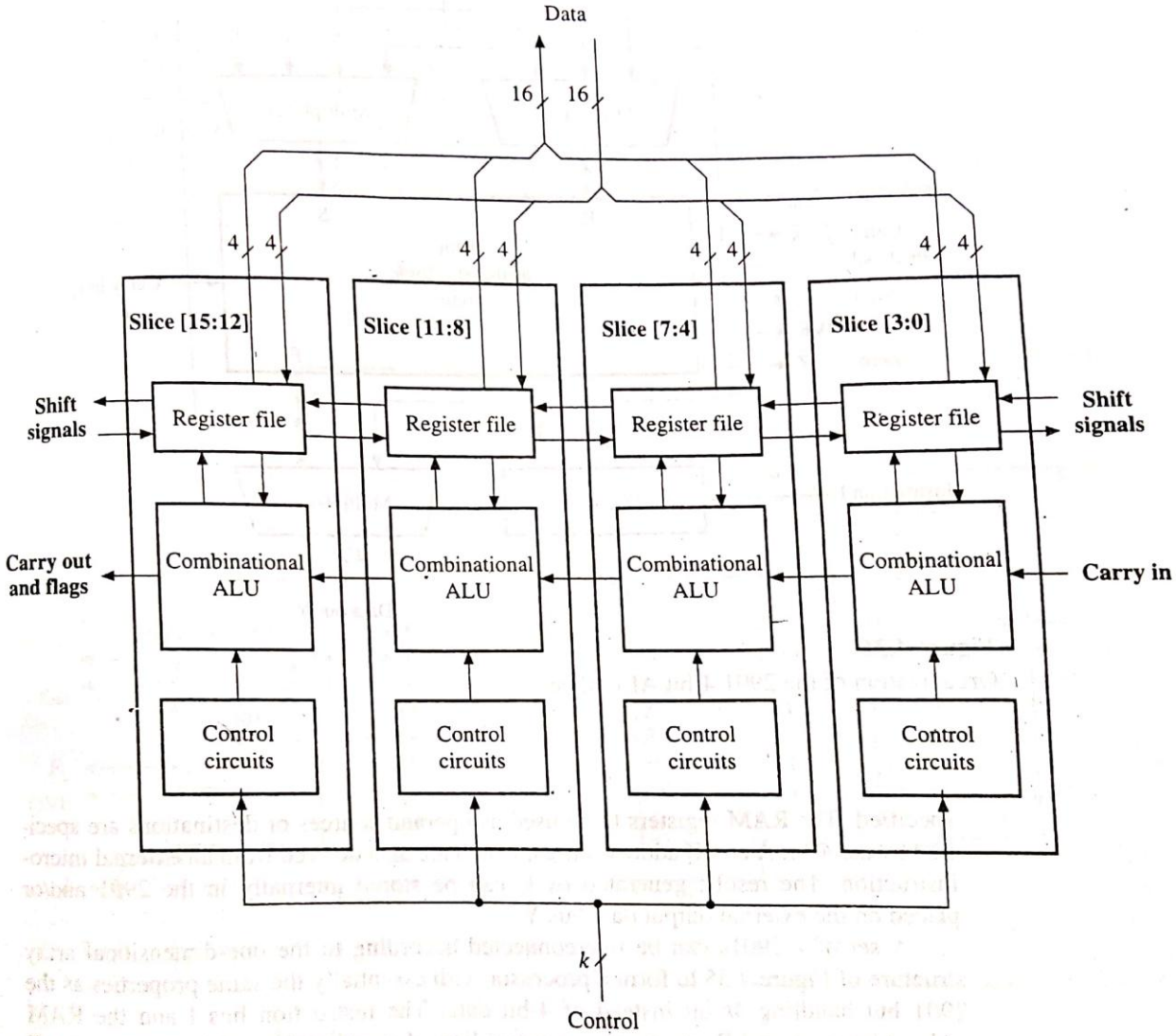


Figure 4.35
Sixteen-bit ALU composed of four 4-bit slices.

FLOATING POINT ARITHMETIC

➢Let (X_M, X_E) be the floating-point representation of a number X, which therefore has the numerical value $X_M \times B^{X_E}$.

➢ Mantissa X_M and the exponent X_E are fixed point numbers and that the base B is the same as the base (radix) of X_M.

➢ Following assumptions are made:

1. X_M is an n_M-bit binary (twos-complement or sign-magnitude) fraction.
2. X_E is an n_E-bit integer in excess-2^{n_E-1} code, implying an exponent bias of 2^{n_E-1}.
3. B=2
4. Final result of each floating-point arithmetic operation should be normalized.

:

FLOATING POINT ARITHMETIC : BASIC OPERATION

➢General formulas for floating-point addition, subtraction, multiplication and division are given below,

Addition : $X + Y = (X_M 2^{X_E - Y_E} + Y_M) \times 2^{Y_E}$ where $X_E \leq Y_E$

Subtraction : $X - Y = (X_M 2^{X_E - Y_E} - Y_M) \times 2^{Y_E}$ where $X_E \leq Y_E$

Multiplication : $X \times Y = (X_M \times Y_M) \times 2^{X_E + Y_E}$

Division : $X / Y = (X_M / Y_M) \times 2^{X_E - Y_E}$

➢ Multiplication and division are simple because the mantissas and exponents can be processed independently.

➢Floating-point multiplication requires a fixed-point multiplication of the mantissas and the fixed-point addition of the exponents.

➢ Eg: X = 1.32400111 x 10^{17} and Y = 1.04799245 x 10^{21}, the product X x Y is given by, (1.32400111 x 1.04799245) x $10^{(17+21)}$ = 1.38758607 x 10^{38}.

FLOATING POINT ARITHMETIC : BASIC OPERATION

- Floating-point division requires a fixed-point division involving the mantissas and a fixed-point subtraction involving the exponents.
- Floating-point addition and subtraction are complicated by the fact that the exponent of the two input operands must be made equal before the corresponding mantissas can be added or subtracted.
- The exponent equalization can be done by right shifting the mantissas X_M associated with the smaller exponent X_E a total of Y_E-X_E digit positions to form a new mantissa $X_M 2^{X_E - Y_E}$, which can be combined directly with Y_M.
- Thus floating point addition and subtraction have three main steps:

1. Compute Y_E-X_E, a fixed-point subtraction.
2. Shift X_M by Y_E-X_E places to the right to form $X_M 2^{X_E - Y_E}$.
3. Compute $X_M 2^{X_E - Y_E} \pm Y_M$, a fixed-point addition or subtraction.

FLOATING POINT ARITHMETIC : BASIC OPERATION

- Eg : To add decimal floating-point numbers X = 1.32400111 x 10^{17} and Y = 1.04799245 x 10^{21}, we first compute Y_E-X_E = 21-17 = 4, identifying X_E as the smaller exponent.
- We then right-shift X_M by four places to obtain $X_M 2^{-4}$ = 0.00013240.
- Finally, we perform the mantissa addition $X_M 2^{-4}+Y_M$ = 0.00013240 + 1.04799245 = 1.04812485.
- So the final result has mantissa 1.04812485 and exponent 21.

Difficulties :

1. Exponent Biasing – If the biased exponents are added or subtracted using fixed-point arithmetic in the course of a floating-point calculation, the resulting exponent is doubly biased and must be corrected by subtracting the bias.
2. all-0 representation – If X x Y is computed as $(X_M \times Y_M) \times 2^{X_E+Y_E}$ and either X_M or Y_M is zero, the resulting product has an all-0 mantissa but may not have an all-0 exponent. A special step is then needed to make the exponent bits 0.

FLOATING POINT ARITHMETIC : BASIC OPERATION

Difficulties :

3. Overflow or underflow – If the result is too large or too small to be represented. Overflow or underflow resulting from mantissa operations can usually be corrected by shifting the mantissa of the result and modifying its exponent; this is done automatically during floating-point processing. However, the exponent overflows or underflows, an error signal indicating floating-point overflow or underflow is generated.

FLOATING POINT ARITHMETIC : FLOATING POINT UNITS

- Floating point arithmetic can be implemented by two loosely connected fixed-point datapath circuits, an exponent unit and a mantissa unit.
- The **mantissa unit** performs all four basic operations on the mantissas; hence a generic fixed-point arithmetic circuit can be used.
- A simpler circuit capable of only adding, subtracting and comparing exponents suffices for the **exponent unit**.
- Exponent comparison can be done by a comparator or by subtracting the exponents.
- The exponents of the input operands are put in register E1 and E2, which are connected to an adder that computes E1+E2.
- The exponent comparison required for addition and subtraction is made by computing E1-E2 and placing it in a counter register E.
- The larger exponent is then determined from the sign of E.
- The shifting of one mantissa required before the mantissa addition or subtraction can occur is controlled by E.

FLOATING POINT ARITHMETIC : FLOATING POINT UNITS

- The magnitude of E is sequentially decremented to zero.
- After each decrement , the appropriate mantissa is shifted one digit position.
- Once the mantissas have been aligned, they are processed in the usual manner.
- The exponent of the result is also computed and placed in E.

FLOATING POINT ARITHMETIC : FLOATING POINT UNITS

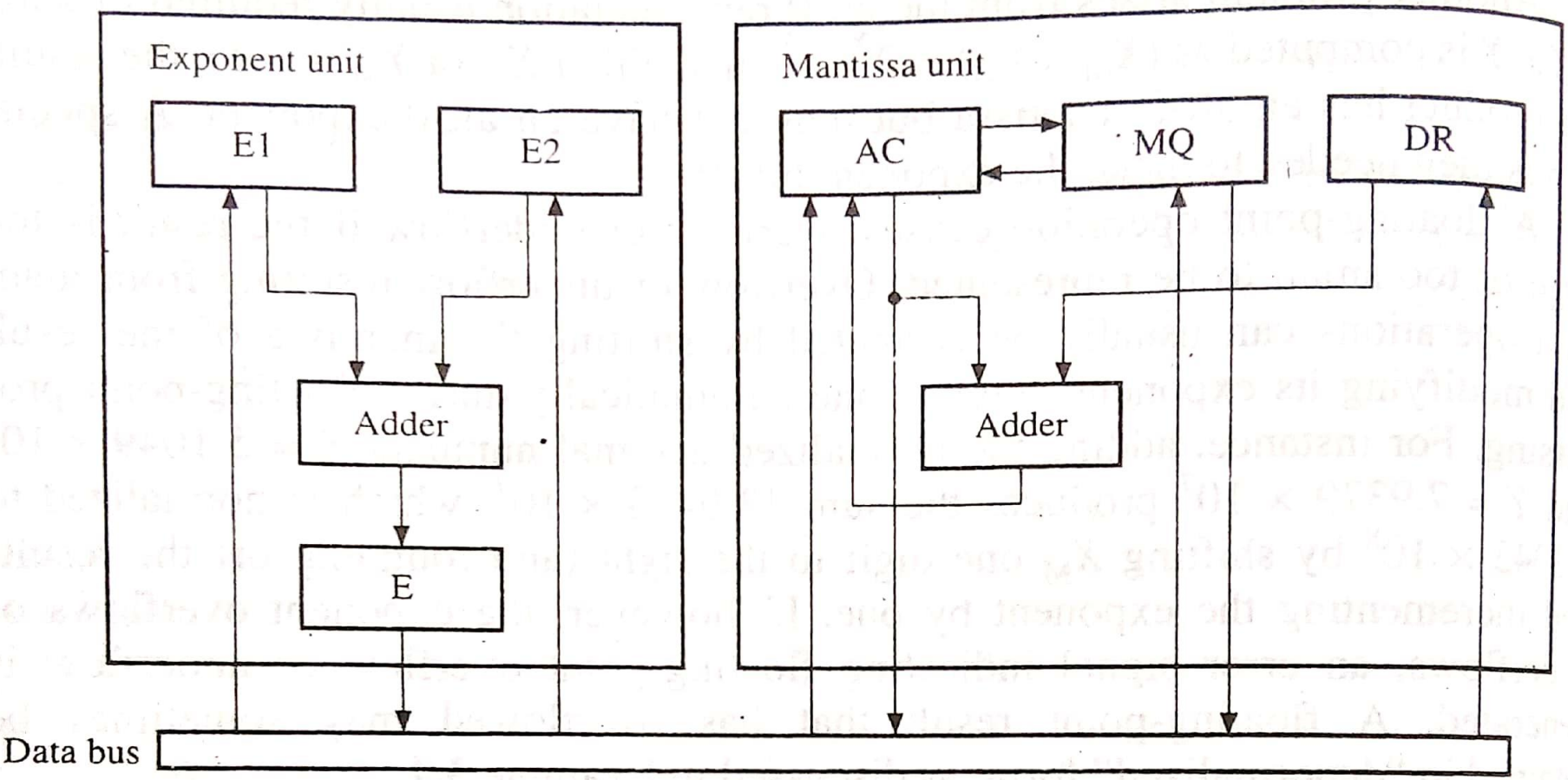


Figure 4.41
Datapath of a floating-point arithmetic unit.

FLOATING POINT ARITHMETIC : FLOATING POINT UNITS

Implementation of floating point addition in detail :

- Addition algorithm for two 32-bit floating point numbers.
- IEEE Standard 754
- In this format each number N has a 23-bit fractional mantissa M with a hidden bit, an 8-bit exponent E in excess-127 code, and a base B=2.
- The value of N is therefore given by the formula,

$$N=(-1)^S 2^{E-127}(1.M)$$

- The numbers to be added in this instances are:

$$X = 0\ 01111111\ 10000000000000000000000\ (+1.5_{10})$$

$$Y = 0\ 10000111\ 00101011010000000000000\ (+299.25_{10}=1.0010101101 \times 2^8)$$

- The exponent subtraction X_E-Y_E in the compare step is done using excess-127 code and produces $01110111 = -8_{10}$.
- Note that a 0 in the left-most bit position of E always indicates a negative number in this code.

FLOATING POINT ARITHMETIC : FLOATING POINT UNITS

Implementation of floating point addition in detail :

- Now the EQUALIZE step is executed, causing E to be incremented and AC, which contains the mantissa of X (including its hidden bit), to be right-shifted.
- After eight shifts, E reaches zero, indicated by its left-most bit changing from 0 to 1.
- Then the mantissa addition takes place, and the larger exponent is transferred from E1 to E.
- The sum appearing in AC is normalized, so the final result X+Y = 300.75_{10} has its exponent in E and its mantissa in AC.

FLOATING POINT ARITHMETIC : FLOATING POINT UNITS

```
                    register AC [n_M–1:0], DR[n_M–1:0], E[n_E–1:0], E1[n_E–1:0], E2[n_E–1:0],
                    AC_OVERFLOW, ERROR;
BEGIN:              AC_OVERFLOW := 0, ERROR := 0,
LOAD:               E1 := X_E, AC := X_M;
                    E2 := Y_E, DR := Y_M;
{Compare and equalize exponents}
COMPARE:            E := E1 – E2;
EQUALIZE:           if E < 0 then AC := right-shift(AC), E := E + 1,
                    go to EQUALIZE; else
                    if E > 0 then DR := right-shift(DR), E := E – 1,
                      go to EQUALIZE;
{Add mantissas}
ADD:                AC := AC + DR, E := max(E1,E2);
{Adjust for mantissa overflow and check for exponent overflow}
OVERFLOW:           if AC_OVERFLOW = 1 then begin
                      if E = E_MAX then go to ERROR;
                      AC := right-shift(AC), E := E + 1, go to END; end
{Adjust for zero result}
ZERO:               if AC = 0 then E := 0, go to END;
{Normalize result}
NORMALIZE:          if AC is normalized then go to END;
UNDERFLOW:          if E > EMIN then
                      AC := left-shift(AC), E := E – 1, go to NORMALIZE;
{Set error flag indicating overflow or underflow}
ERROR:              ERROR := 1;
END:                . . .
```

Figure 4.42
Algorithm for floating-point addition.

FLOATING POINT ARITHMETIC : FLOATING POINT UNITS

	Exponent registers			Mantissa registers	
Step	E1	E2	E	AC	DR
LOAD	01111111 $= X_E$	10000111 $= Y_E$	00000000	11000000000000 . . . 00 $= 1.X_M$	10010101101000 . . . 00 $= 1. Y_M$
COMPARE			01110111 $= X_E - Y_E$		
EQUALIZE			01111000 01111001 01111010 01111011 01111100 01111101 01111110 01111111 10000000	01100000000000 . . . 00 00110000000000 . . . 00 00011000000000 : . . 00 00001100000000 . . . 00 00000110000000 . . . 00 00000011000000 . . . 00 00000001100000 . . . 00 00000000110000 . . . 00	
ADD			10000111 $= Y_E$	10010110011000 . . . 00 $= AC + DR$	
Result			10000111 $= (X + Y)_E$	10010110011000 . . . 00 $= 1. (X + Y)_M$	

Figure 4.43
Illustration of the floating-point addition algorithm of Figure 4.42.

FLOATING POINT ARITHMETIC : FLOATING POINT UNITS

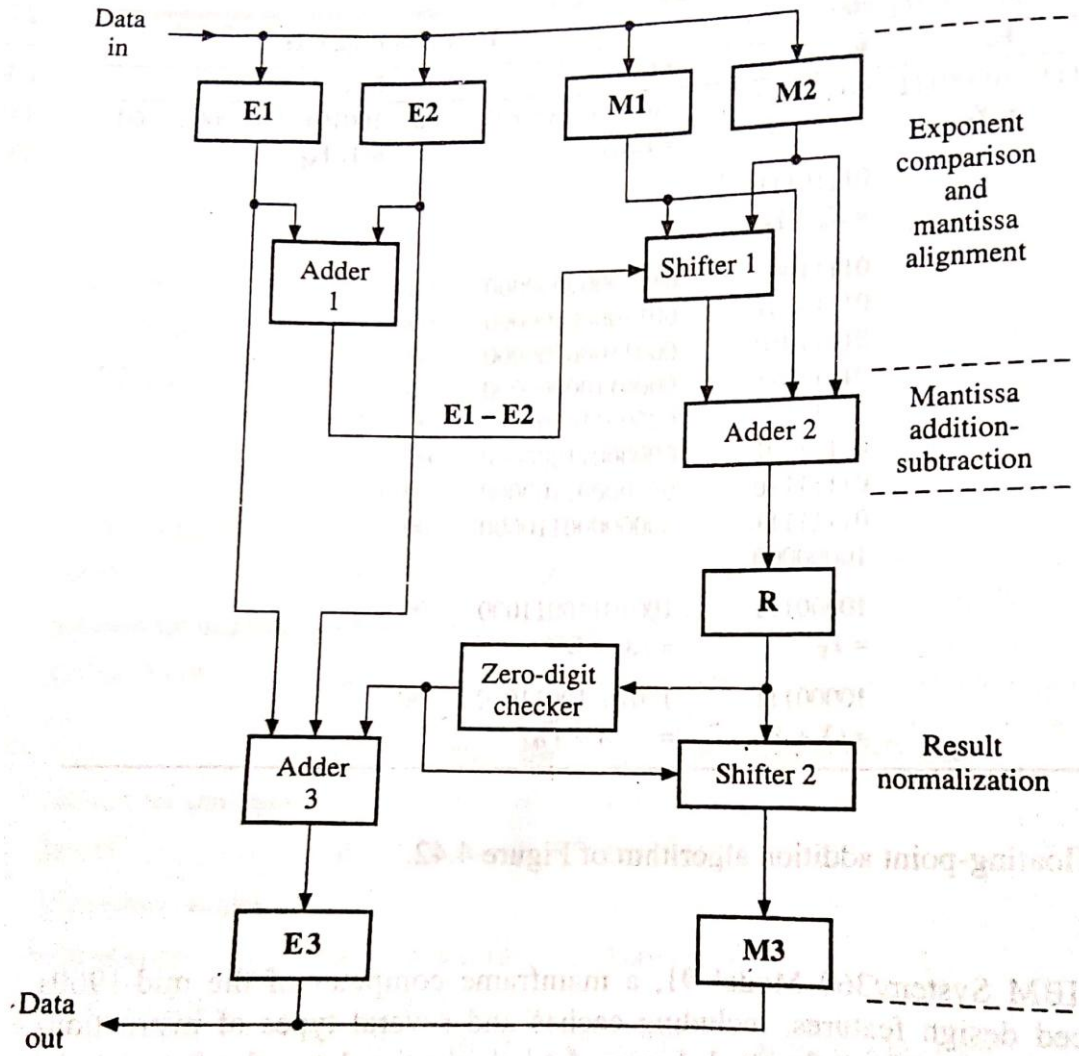


Figure 4.44
Floating-point add unit of the IBM System/360 Model 91.

COPROCESSOR

- Complicated arithmetic operations like exponentiation and trigonometric functions are costly to implement in CPU hardware, while software implementations of these operations are slow.
- Auxiliary processors called as arithmetic coprocessors to provide fast, low-cost hardware implementations of these special functions.
- Coprocessor is a separate instruction set processor that is closely coupled to the CPU and whose instructions and registers are direct extensions of the CPU's.
- Instructions intended for the coprocessor are fetched by the CPU, jointly decoded by the CPU and the coprocessor, and executed by the coprocessor in a manner that is transparent to the programmer.
- The coprocessor is attached to the CPU by several control lines that allows the activities of the two processors to be coordinated.
- To the CPU, the coprocessor is a passive or slave device whose registers can be read and written into in much the same manner as external memory.

COPROCESSOR

➢Even if no coprocessor is actually present, coprocessor instructions can be included in CPU programs, because if the CPU knows that no coprocessor is present, it can transfer program control to a predetermined memory location where a software routine implementing the desired coprocessor instruction is stored. This type of CPU-generated interruption of normal program flow is termed as a **coprocessor trap**.

➢A coprocessor instruction typically contains the following three fields:

1. An opcode F_0 that distinguishes coprocessor instructions from other CPU instructions
2. The address F_1 of the particular coprocessor to be used if several coprocessors are allowed.
3. Finally the type F_2 of the particular operation to be executed by the coprocessor. The F_2 field can include operand addressing information.

COPROCESSOR

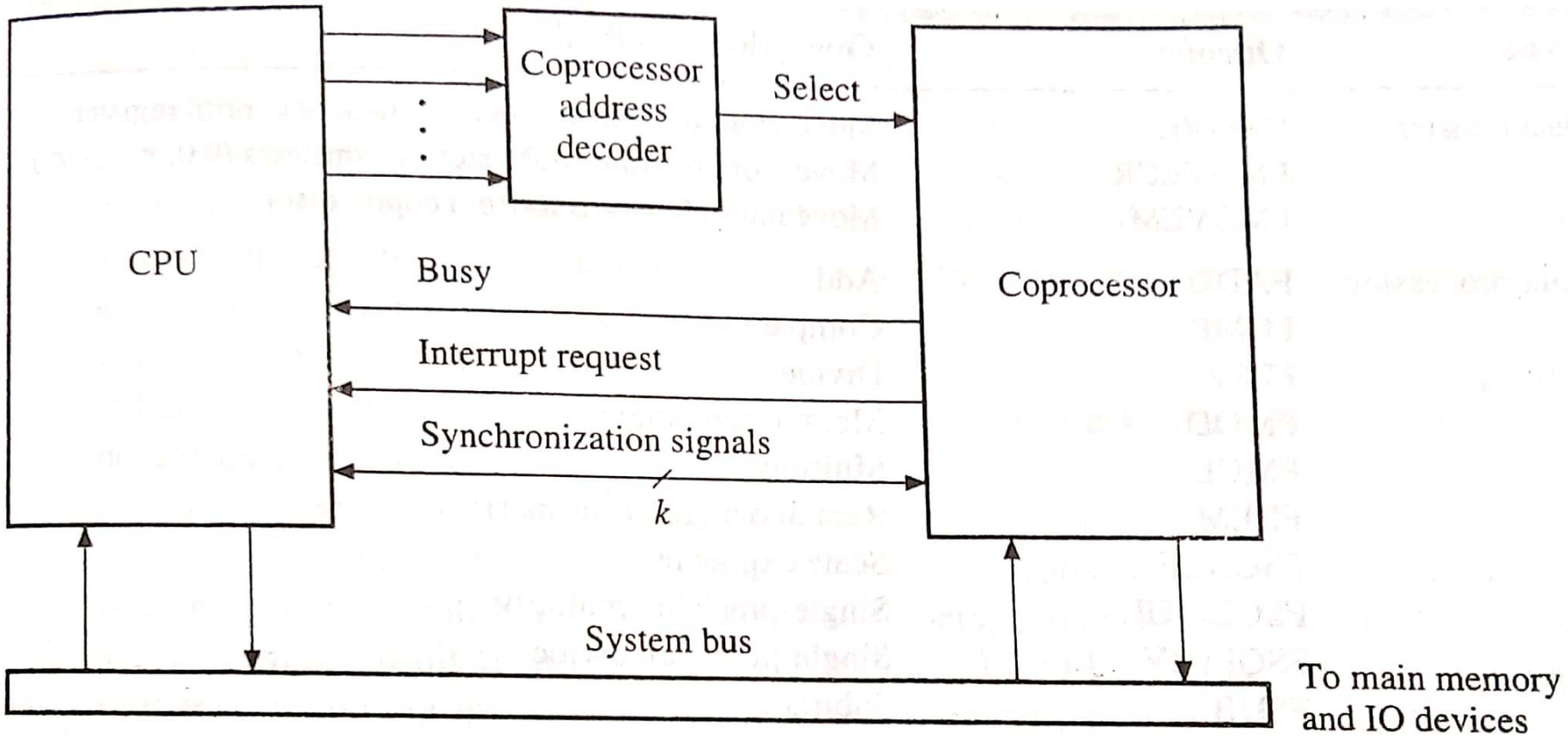


Figure 4.45
Connections between a CPU and a coprocessor.

PIPELINE PROCESSING

- It is a general technique for increasing processor throughput without requiring large amounts of extra hardware.
- It is applied to the design of the complex datapath units such as multipliers and floating point adders.
- A pipeline processor consists of a sequence of m data-processing circuits, called **stages** or **segments** which collectively perform a single operation on a stream of data operands passing through them.
- Some processing takes place in each stage, but a final result is obtained only after an operand set has passed through the entire pipeline.

PIPELINE PROCESSING

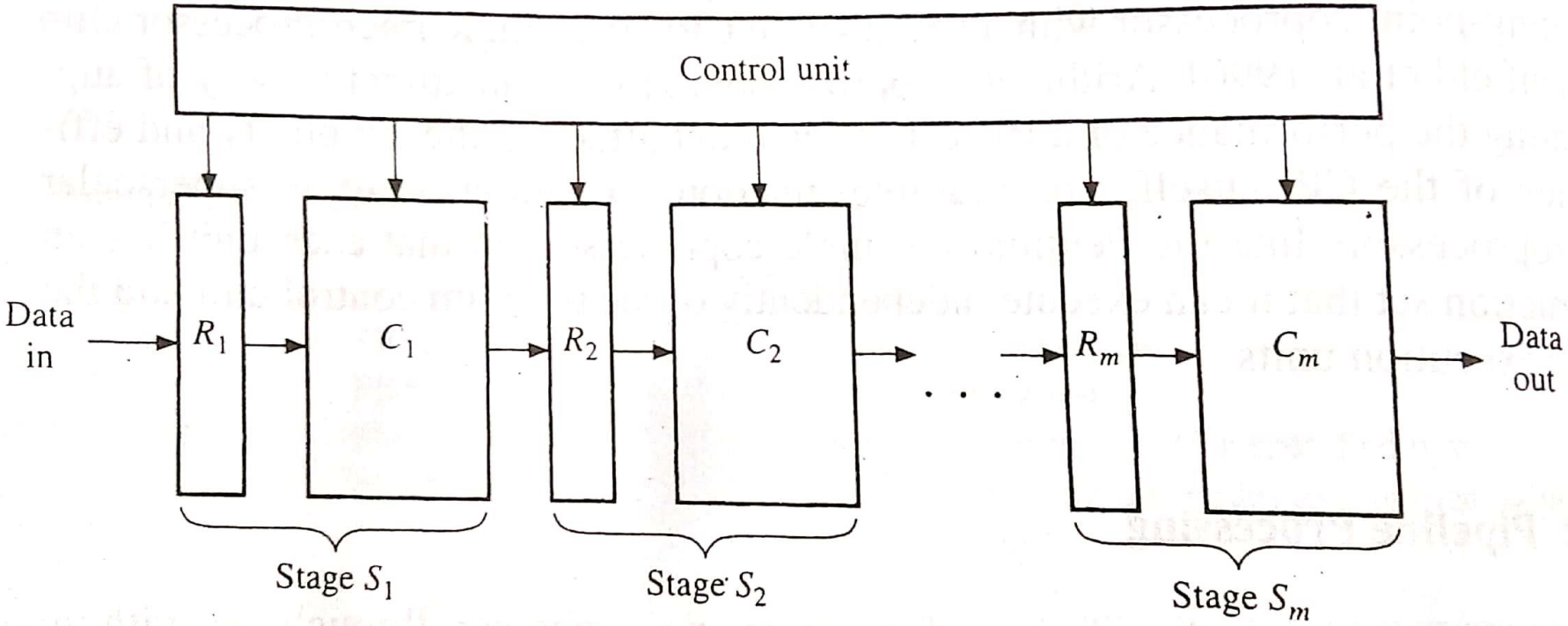


Figure 4.47
Structure of a pipeline processor.

PIPELINE PROCESSING

- A stage S_i contains a multiword input register or latch R_i, and a datapath circuit C_i that is usually combinational.
- The R_i's hold partially processed results as they move through the pipeline.
- A common clock signal causes the R_i's to change state synchronously.
- Each R_i receives a new set of input data D_{i-1} from the preceding stage S_{i-1} except for R_1 whose data is supplied from an external source.
- Pipeline's advantage is that an m-stage pipeline can simultaneously process up to m independent sets of data operands.
- These data sets move through the pipeline stage by stage so that when the pipeline is full, separate operations are being executed concurrently, each in a different stage.
- Furthermore, a new, final result emerges from the pipeline every clock cycle.

PIPELINE PROCESSING

- Suppose that each stage of the m-stage pipeline takes T seconds to perform its local suboperation and store its results. The **T is the pipeline clock period**.
- The **delay** or **latency** of the pipeline is the time to complete a single operation, is therefore **mT**.
- However the **throughput** of the pipeline is the maximum number of operations completed per second is **1/T**.
- When performing a long sequence of operations in the pipeline, its performance is determined by the delay (latency) T of a single stage, rather than by the delay mT of the entire pipeline.
- Hence an m-stage pipeline provides a speedup factor of m compared to a non-pipelined implementation of the same target operation.

PIPELINE PROCESSING : EXAMPLE

- The addition of two normalized floating-point numbers x and y.
- Four step sequence :

1. Compare the exponents
2. Align the mantissas (equalize the exponents)
3. Add the mantissas
4. Normalize the result.

- These operations require the four stage pipeline processor.
- **x** is represented by ($\mathbf{x_M, x_E}$) where x_M is the mantissa and x_E is the exponent
- **y** is represented by ($\mathbf{y_M, y_E}$) where y_M is the mantissa and y_E is the exponent.

PIPELINE PROCESSING : EXAMPLE

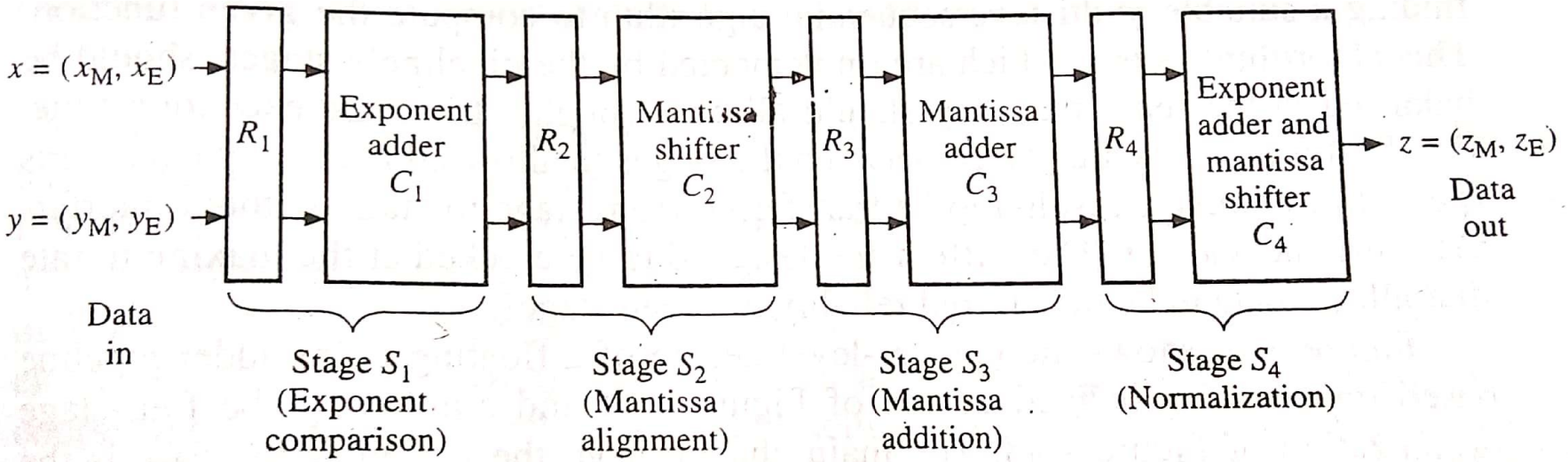


Figure 4.48
Four-stage floating-point adder pipeline.

PIPELINE PROCESSING : EXAMPLE

➢Performing a sequence of N floating point additions of the form $x_i + y_i$ for the case N=6.

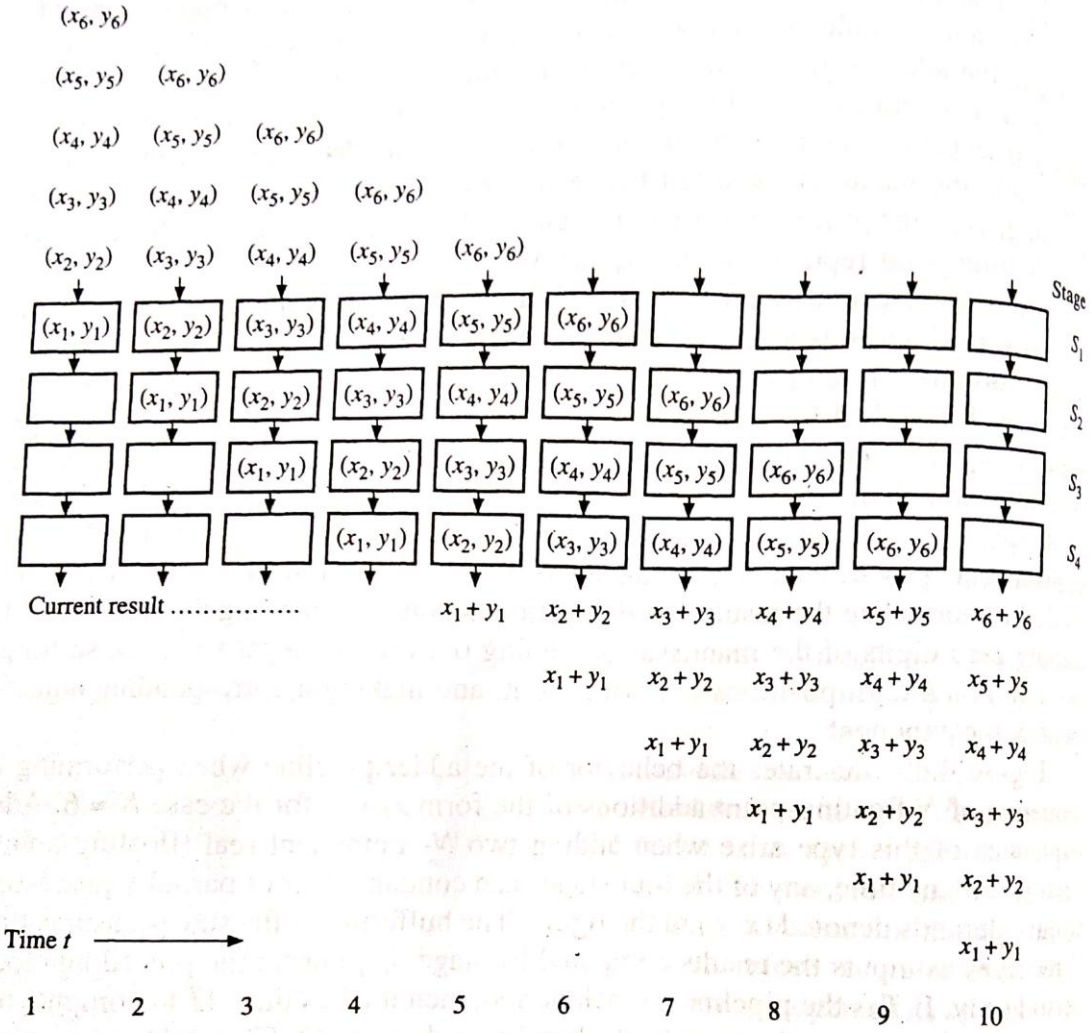


Figure 4.49
Operation of the four-stage floating-point adder pipeline.

PIPELINE PROCESSING : EXAMPLE

- Once all four stages of the pipeline have been filled with data, a new sum emerges from the last stage S_4 every T seconds.
- Consequently, N consecutive additions can be done in time **(N+3)T**.

PIPELINE PROCESSING : PIPELINE DESIGN

- First find a suitable multistage sequential algorithm to compute the given function.
- The algorithm steps should be balanced – stages should all have roughly the same execution time.
- Fast buffer registers are placed between the stages to allow all necessary data items (partial or complete) result to be transferred from stage to stage without interfering with one another.
- The buffers are designed to be clocked at the maximum rate that allows data to be transferred reliably between stages.
- The main change from the non-pipelined case is the inclusion of buffer registers to define and isolate the four stages.
- A further modification has been made to implement fixed-point as well as floating-point addition.

PIPELINE PROCESSING : PIPELINE DESIGN

- Without pipelining

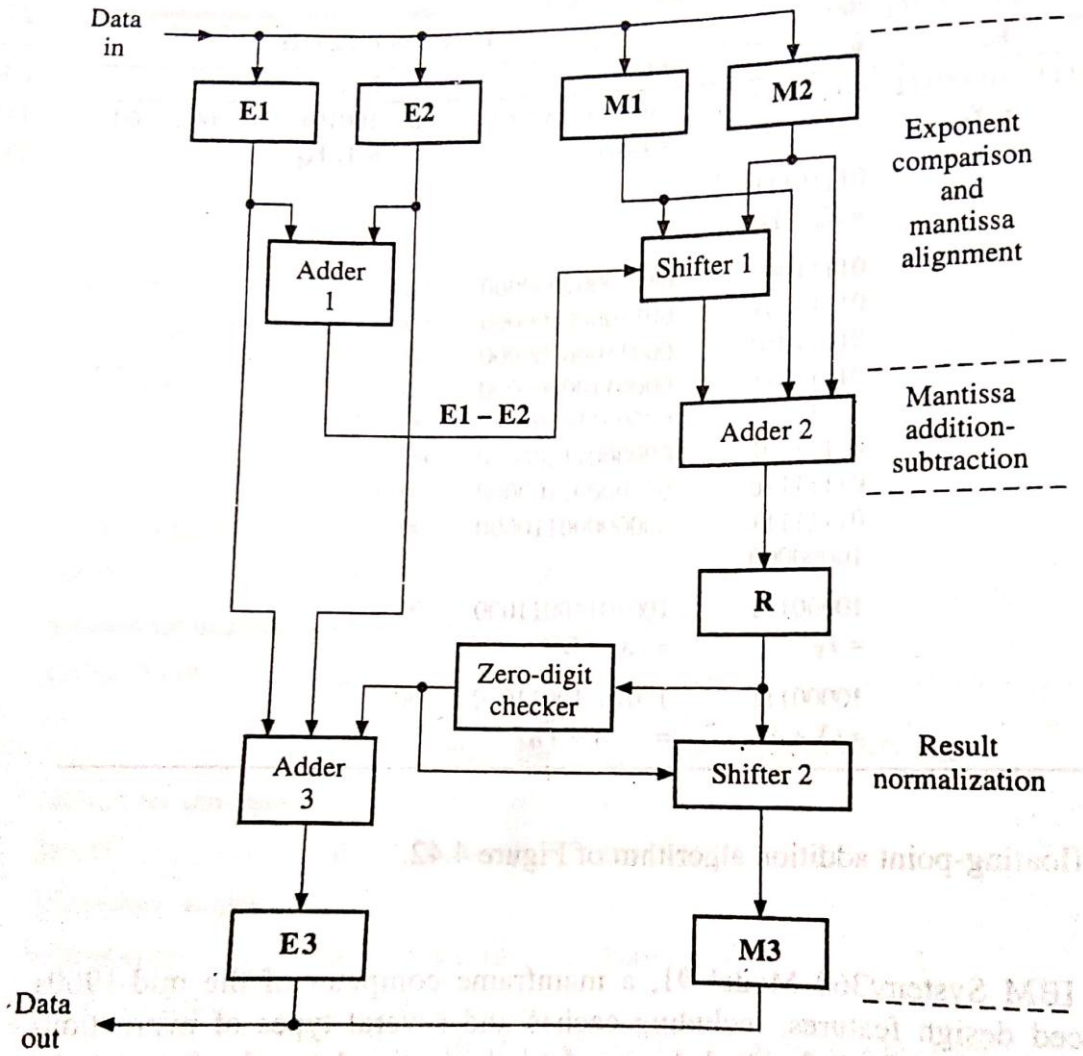


Figure 4.44
Floating-point add unit of the IBM System/360 Model 91.

PIPELINE PROCESSING : PIPELINE DESIGN

- With pipelining

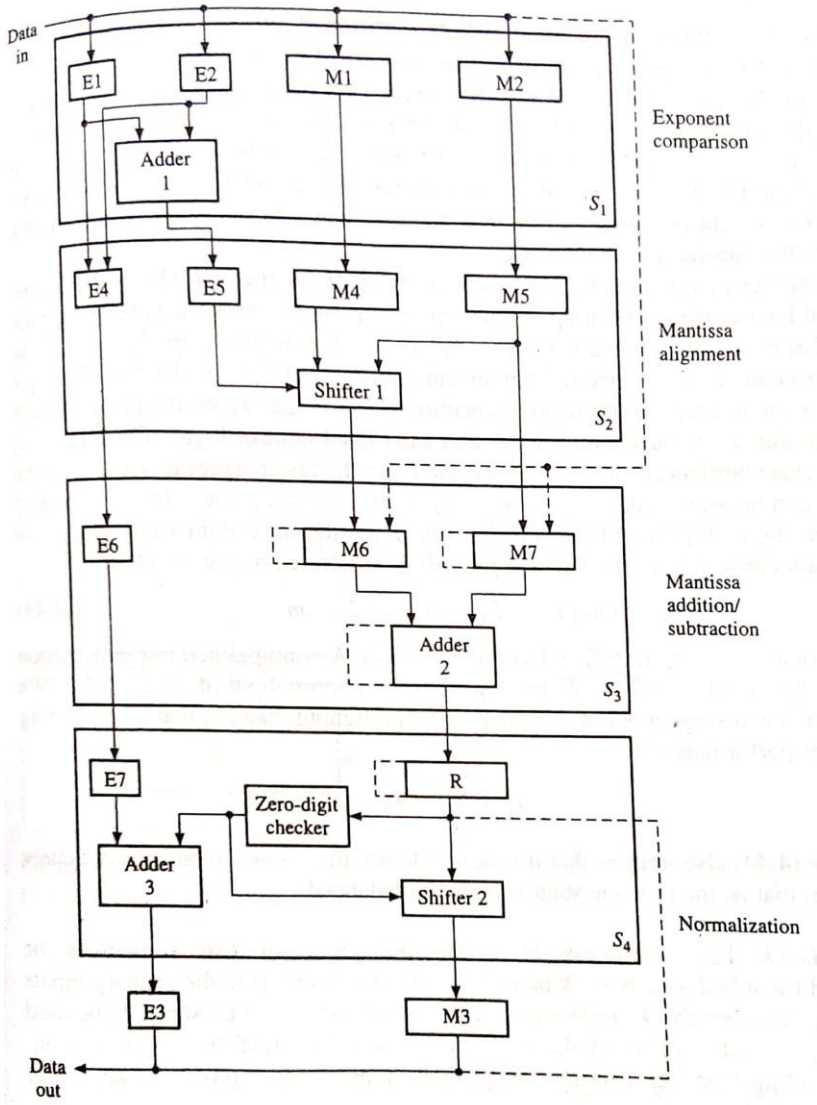


Figure 4.50
Pipelined version of the floating-point adder of Figure 4.44.

PIPELINE PROCESSING : PIPELINE DESIGN

➢The circuits that perform the mantissa addition in stage S_3 and the corresponding buffers are enlarged, as shown by broken lines in Fig. 4.50, to accommodate full-size fixed-point operands.

➢To perform a fixed-point addition, the input operands are routed through S_3 only, bypassing the other three stages. Thus the circuit is called as **multifunction pipeline** that can be configured either as a four stage floating point adder or as a one-stage fixed-point adder.

Feedback: The usefulness of the pipeline processor can sometimes be enhanced by including feedback paths from the stage outputs to the primary inputs of the pipeline.

➢Feedback enables the results computed by certain stages to be used in subsequent calculations by the pipeline.

PIPELINE PROCESSING : PIPELINE DESIGN

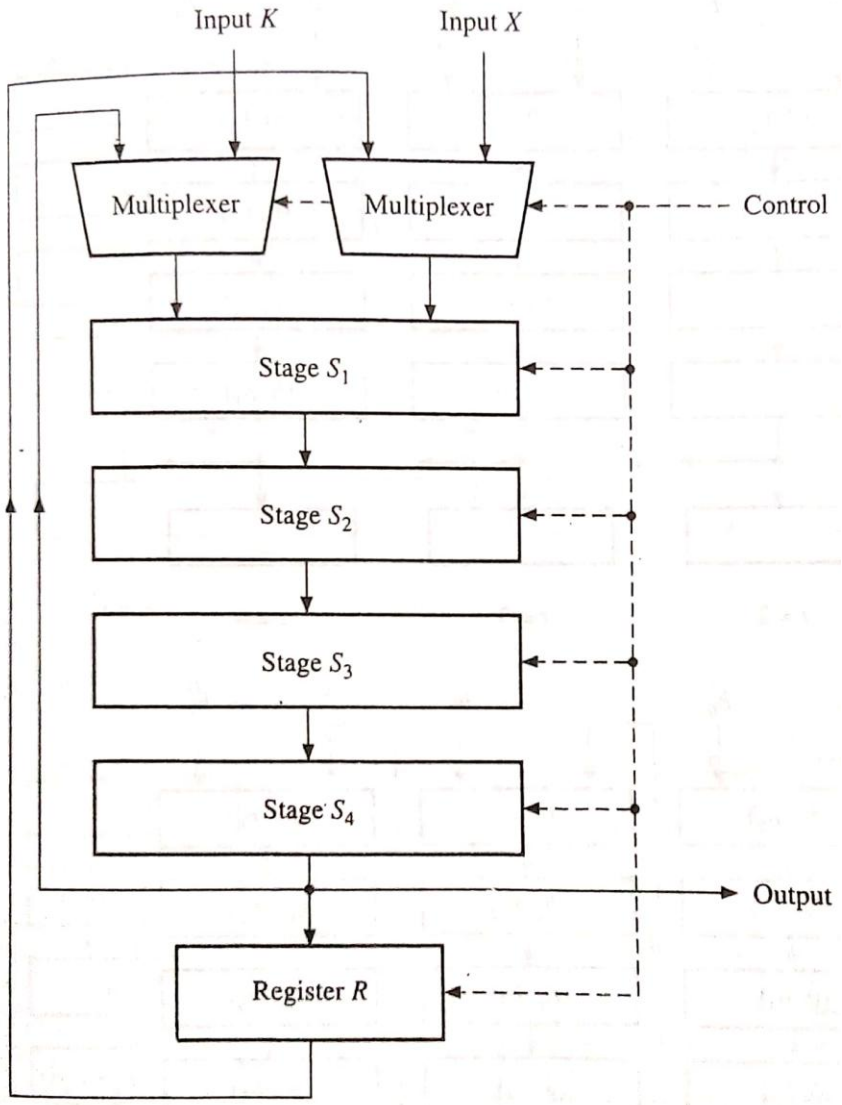


Figure 4.51
Pipelined adder with feedback paths.

PIPELINE PROCESSING : PIPELINED MULTIPLIERS

- Multiplying two n-bit fixed point numbers $X = x_{n-1}\, x_{n-2} \ldots x_0$ and $Y = y_{n-1}\, y_{n-2} \ldots y_0$.
- It consists of 1-bit multiply and add cell M
- n=3
- Each cell **M** computes a 1-bit product $x_i y_i$ and adds it to both a product bit from the preceding stage and a carry bit generated by the cell on its right.
- Thus the n cells in each stage S_i, $0 \le i \le n-1$, compute a partial product of the form,

$$P_i = P_{i-1} + x_i 2^i Y$$

- In addition to storing the partial products in the buffer registers denoted R_i, the multiplicand Y and all hitherto unused multiplier bits must also be stored in R_i.

Disadvantage : relatively slow speed of the carry propagation logic in each stage.

- The number of M cells needed is n^2 and the capacity of all the buffer registers is approximately $3n^2$.
- Hence this type of multiplier is fairly costly in hardware.

PIPELINE PROCESSING : PIPELINED MULTIPLIERS

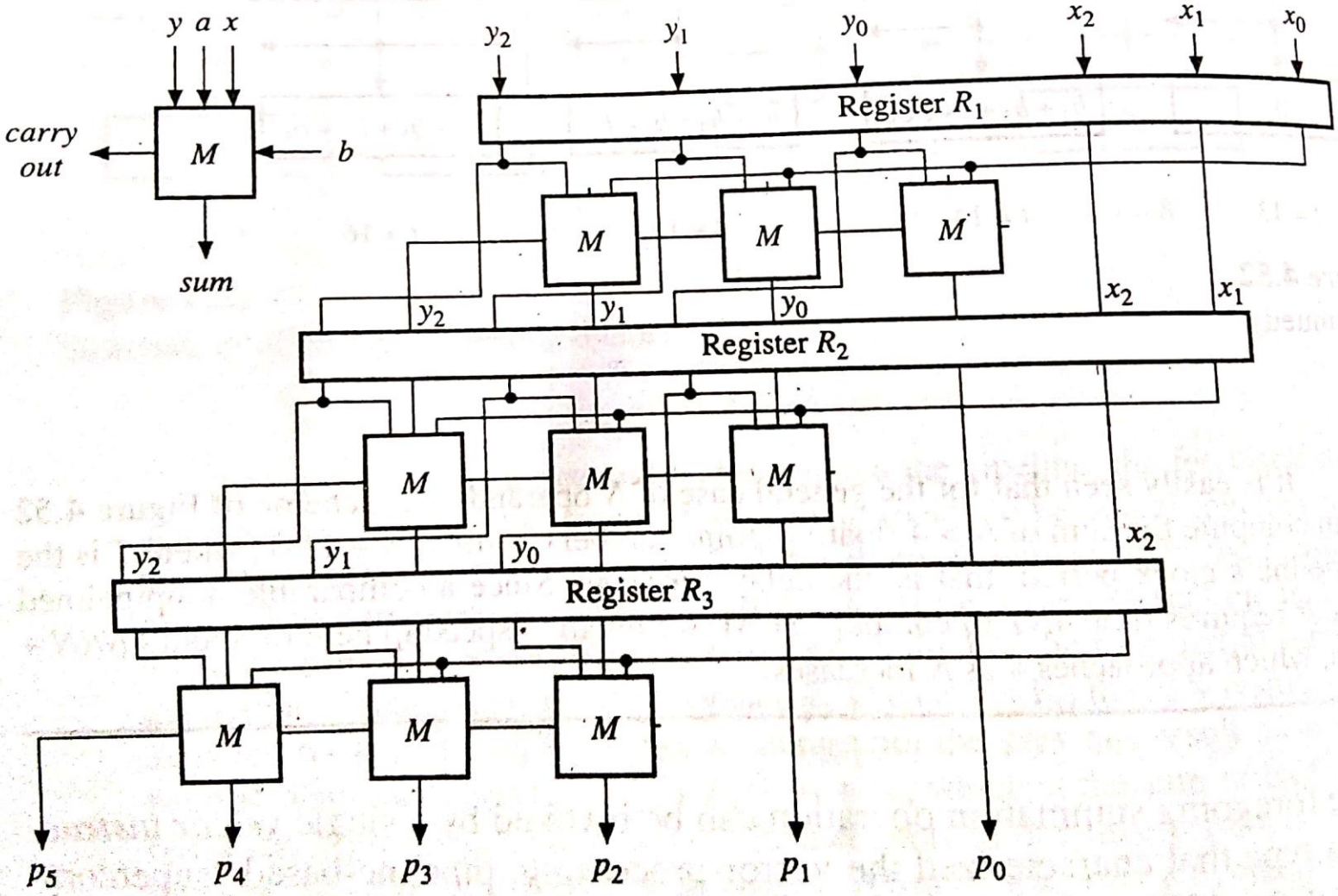
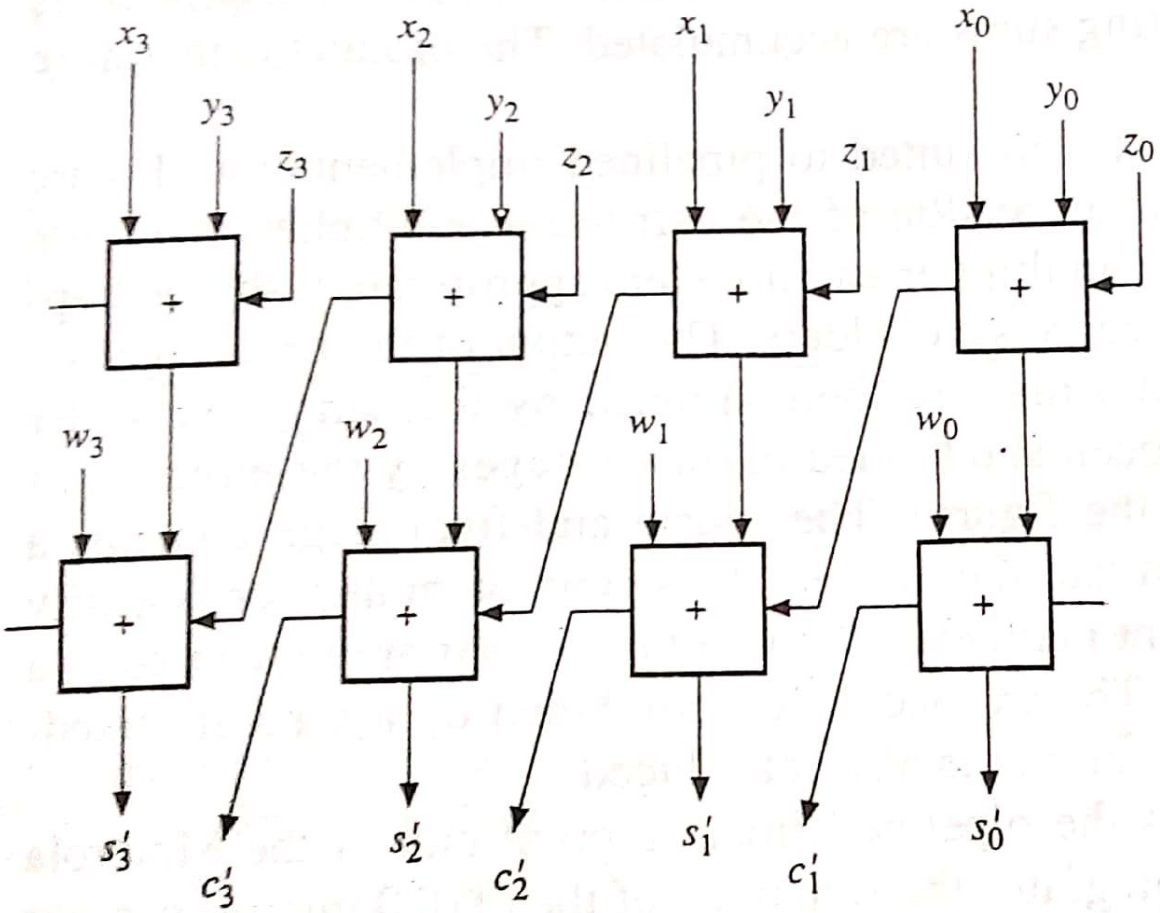


Figure 4.53
Multiplier pipeline using ripple-carry propagation.

PIPELINE PROCESSING : PIPELINED MULTIPLIERS

- Multipliers often employ a technique called carry-save addition, which is particularly well suited to pipelining.
- An n-bit carry-save adder consists of n disjoint full adders.
- Its input is three n-bit numbers to be added, while the output consists of the n sum bits forming a word S and n carry bits forming a word C.
- There is no carry propagation within the individual adders.
- The outputs S and C can be fed into another n-bit carry save adder where they can be added to a third n-bit number W.
- Observe that the carry connections are shifted to the left to correspond to normal carry propagation.
- In general, m numbers can be added by a treelike network of carry-save adders to produce result in the form (S,C).
- To obtain the final sum, S and C must be added by a conventional adder with carry propagation.

PIPELINE PROCESSING : PIPELINED MULTIPLIERS



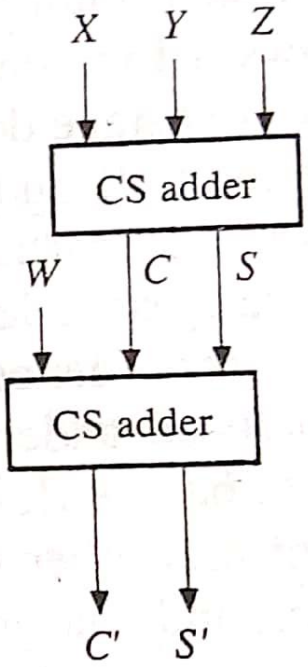


Figure 4.54
A two-stage carry-save adder.

PIPELINE PROCESSING : PIPELINED MULTIPLIERS

- Multiplication can be performed using a multistage carry-save adder circuit – **Wallace tree**.
- The inputs to the adder are n terms of the form $\mathbf{M_i = x_i Y 2^k}$.
- M_i represents the multiplicand Y multiplied by the ith multiplier bit weighted by the appropriate power of 2.
- Suppose that M_i is 2n bits long and that a full double-length product is required.
- The desired product P is $\sum_{i=0}^{n-1} M_i$
- This sum is computed by the carry-save adder tree, which produces a 2n-bit sum and a 2n-bit carry word.
- The final carry assimilation is performed by a fast adder – a carry lookahead adder, for instance – with normal internal carry propagation.

PIPELINE PROCESSING : PIPELINED MULTIPLIERS

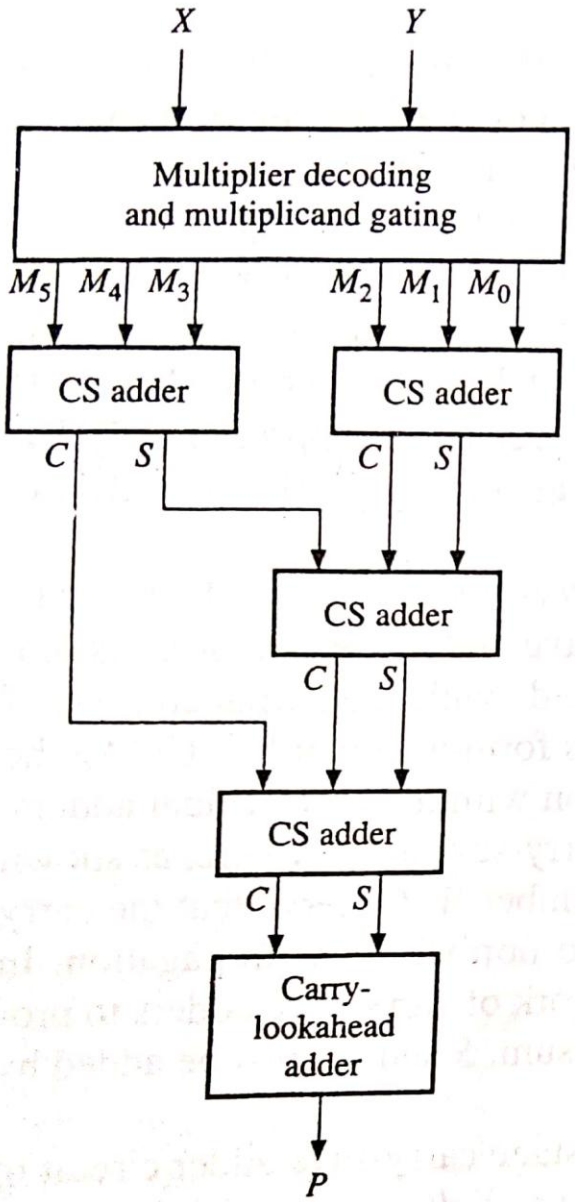


Figure 4.55
A carry-save (Wallace tree) multiplier.

PIPELINE PROCESSING : PIPELINED MULTIPLIERS

- Four stage pipelined version of the carry-save multiplier.
- First stage – decodes the multiplier and transfers appropriately shifted copies of the mutiplicand into the carry-save adders. Output – set of numbers (partial products) that are then summed by the carry-save adder tree.
- Second and Third stage – carry-save logic has been subdivided into two stages by the insertion of buffer registers (R).
- Fourth stage – contains a carry-lookahead adder to assimilates the carries.

PIPELINE PROCESSING : PIPELINED MULTIPLIERS

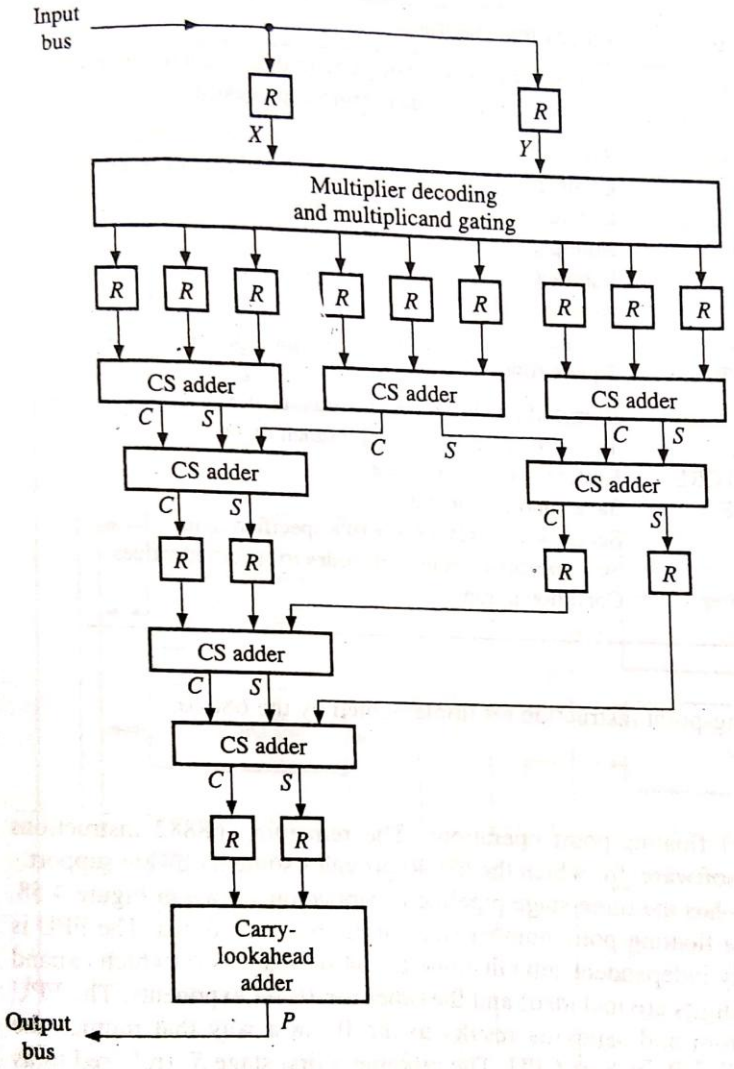


Figure 4.56
A pipelined carry-save multiplier.

PIPELINE PROCESSING : SYSTOLIC ARRAYS

- Systolic arrays – data processing circuits.
- Formed by interconnecting a set of identical data-processing cells in a uniform manner.
- Data words flow synchronously from cell to cell, with each cell performing a small step in the overall operation of the array.
- The data's are not fully processed until the end results emerge from the array's boundary cells.
- Applications : Convolution, matrix multiplication and solution techniques for linear equations.

PIPELINE PROCESSING : SYSTOLIC ARRAYS

- Let X be an nxn matrix of fixed-point or floating-point numbers defined by,

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,n} \\ x_{2,1} & x_{2,2} & x_{2,n} \\ x_{n,1} & x_{n,2} & x_{n,n} \end{bmatrix}$$

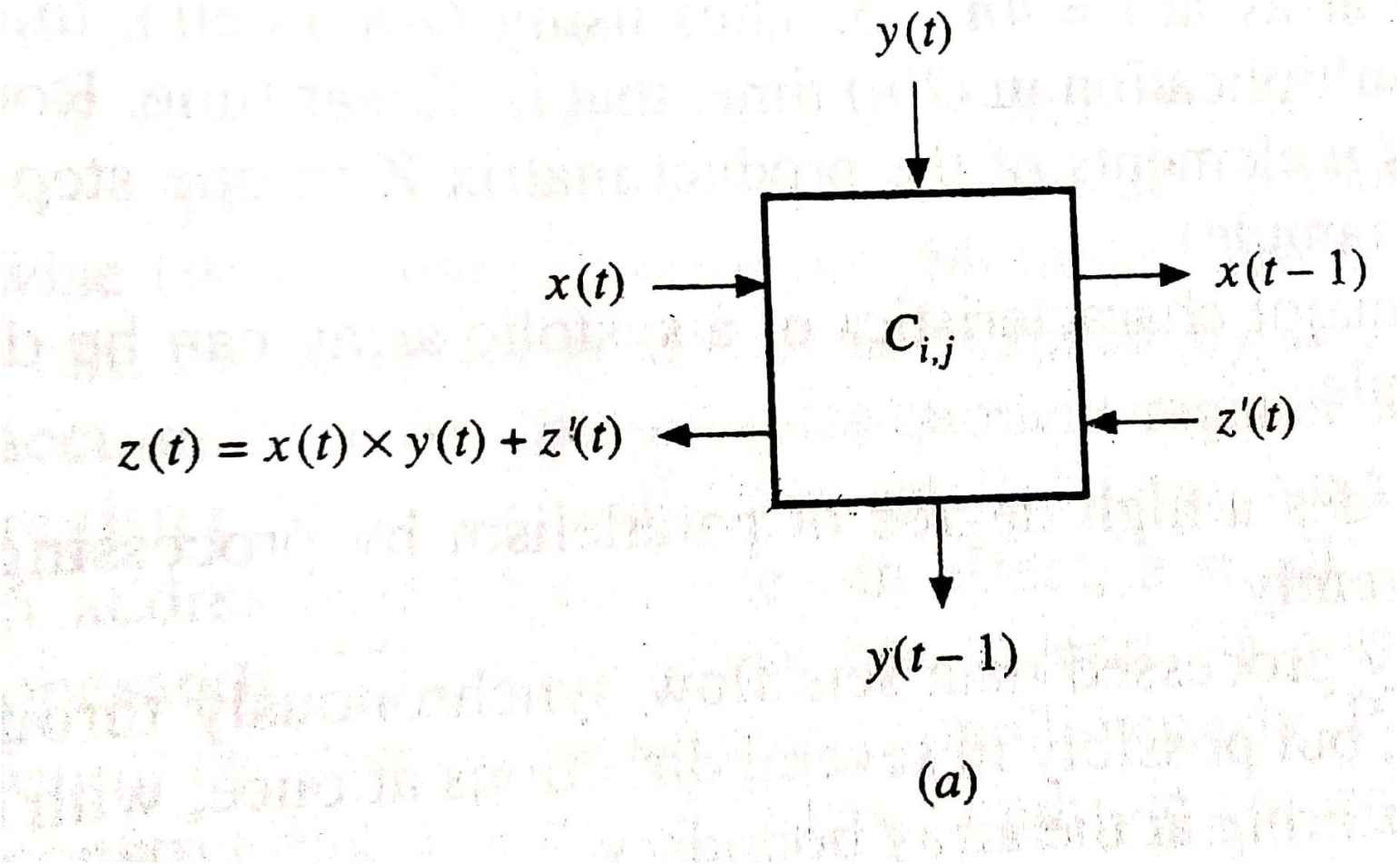
- $X = [x_{i,j}]$
- The product of X and another nxn matrix Y = $[y_{i,j}]$ is the nxn matrix Z = $[z_{i,j}]$ given by,

$$z_{i,j} = \sum_{k=1}^{n} x_{i,k} \times y_{k,j}$$

- A systolic array for matrix multiplication may be constructed from a cell that executes the following multiply-and-add operation on individual numbers (scalars),

z := z' +x x y

PIPELINE PROCESSING : SYSTOLIC ARRAYS



PIPELINE PROCESSING : SYSTOLIC ARRAYS

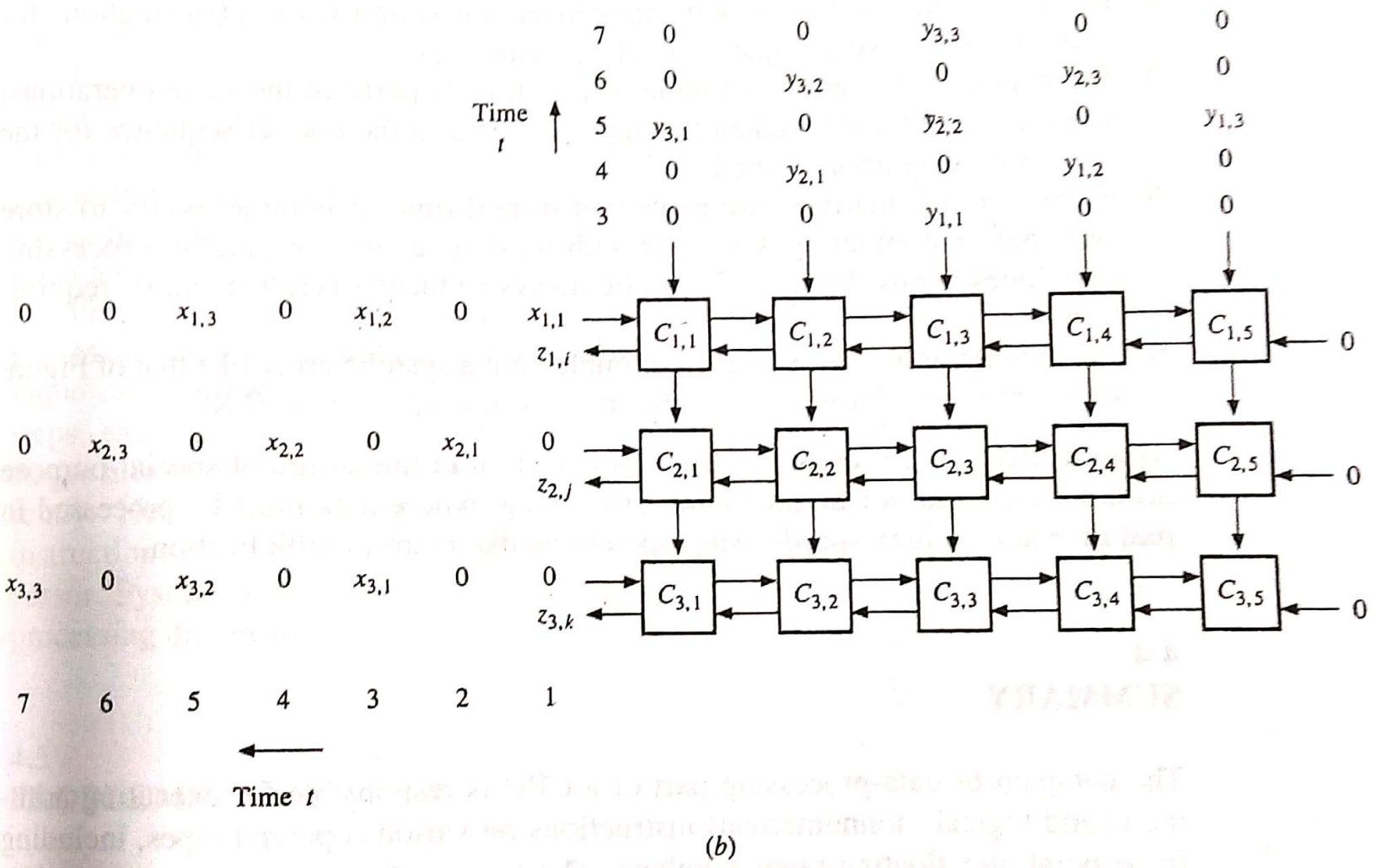


Figure 4.59
Systolic array for matrix multiplication: (*a*) basic cell and (*b*) 3 × 5 array.

SUMMARY

- Fixed point and floating point arithmetic operations
  - Addition
  - Subtraction
  - Multiplication
  - Division
- Combinational and sequential ALU
- Coprocessor
- Pipeline processing