UNIT - III

Outline

General registers Organization – Control Word, Examples of micro operation,
Stack Organization – Register stack, Memory stack, Reverse Polish Notation,
Evaluation of Arithmetic Expressions

Instruction Formats – Three Address Instructions, Two Address Instructions
One Address Instructions, Zero Address Instructions,

Addressing Modes-Types
Data Transfer and Manipulation –
Data Transfer Instructions,
Data Manipulation Instructions – Arithmetic instructions,
Logical and bit manipulation instructions, shift instructions,
Program Control – Conditional Branch instructions, Subroutine Call and Return.

Introduction

Major Components of CPU

- Storage Components: To store the data (Processors)
  - Registers
  - Flip-flops
- Execution (Processing) Components: Which performs calculations
  - Arithmetic Logic Unit (ALU):
  - Arithmetic calculations, Logical computations, Shifts/Rotates
- Transfer Components: to transfer data from ALU and Registers
  - Bus
- Control Components:
  - Control Unit

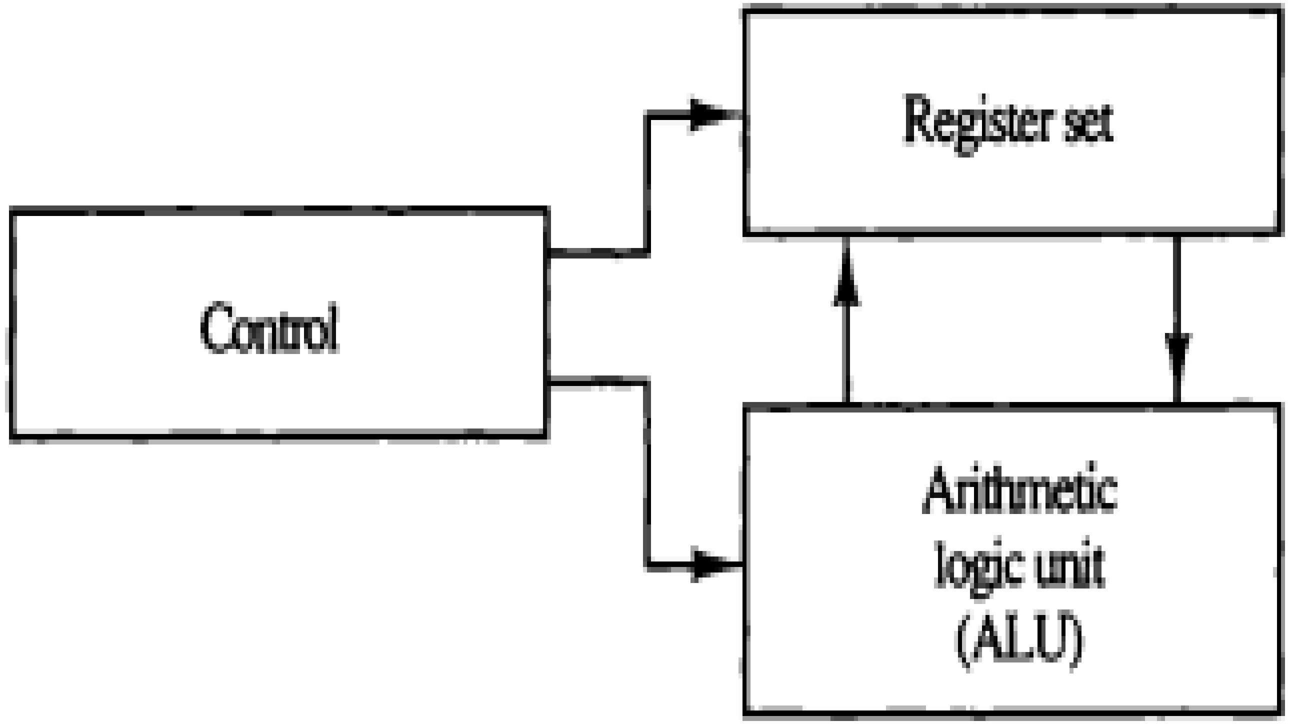
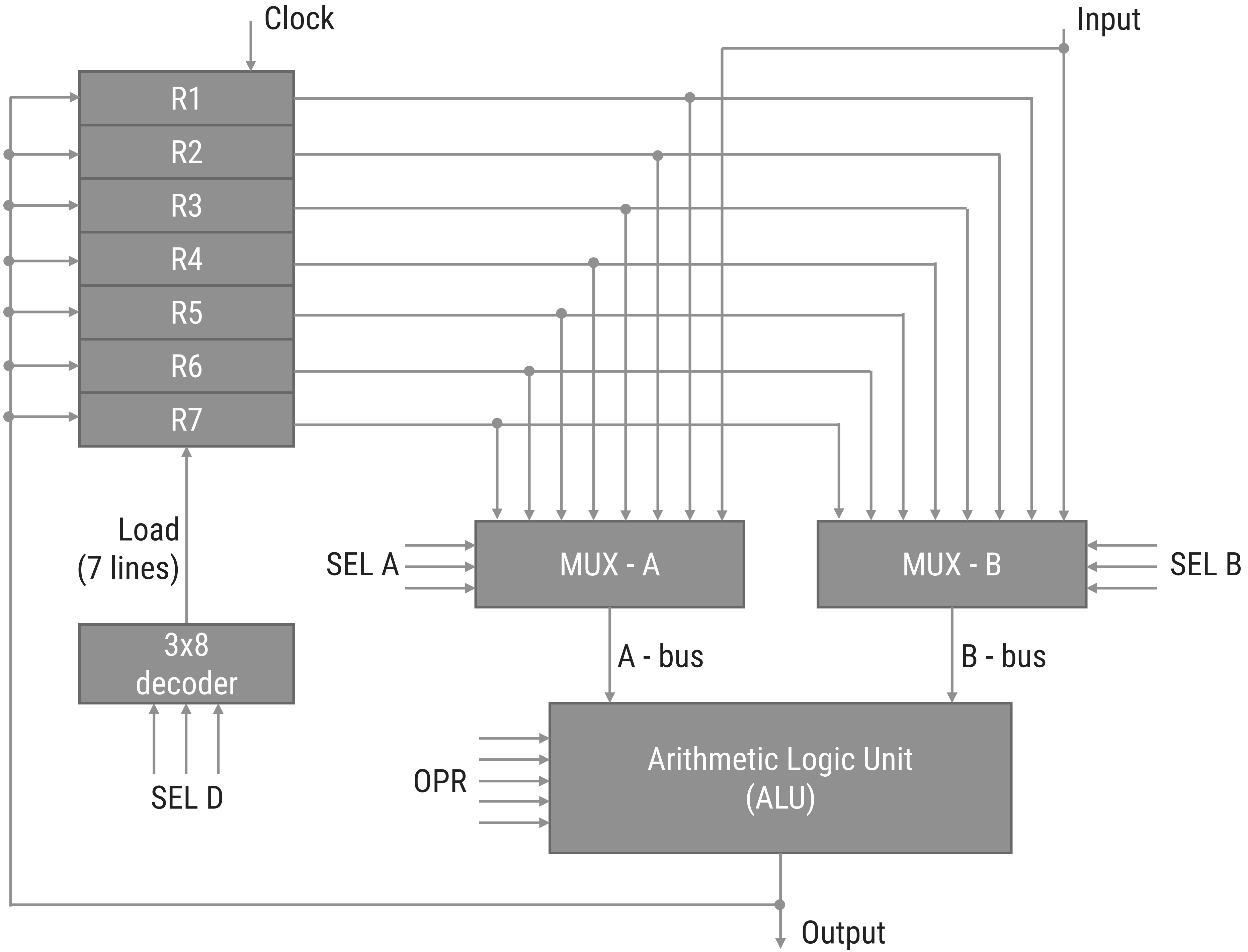


Figure 8-1 Major components of CPU.

General Register Organization

Section - 1


Clock
Input
R1
R2
R3
R4
R5
R6
R7
Load
(7 lines)
3x8
decoder
SEL D
SEL A
MUX - A
MUX - B
SEL B
A - bus
B - bus
OPR
Arithmetic Logic Unit
(ALU)
Output

General Register Organization

- Example: $R1 \longleftarrow R2 + R3$
- To perform the above operation, the control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of R2 into bus A.
2. MUX B selector (SELB): to place the content of R3 into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition A + B.
4. Decoder destination selector (SELD): to transfer the content of the output bus into R1.

- *Control Word:*

3	3	3	5
SELA	**SELB**	**SELD**	**OPR**

Binary Code	**SELA**	**SELB**	**SELD**
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

General Register Organization

OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	A + B	ADD
00101	A − B	SUB
00110	Decrement A	DECA
01000	A and B	AND
01010	A or B	OR
01100	A xor B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA
11000	Shift left A	SHLA

Encoding of ALU Operations

Examples of Microoperations

A control word of 14 bits is needed to specify a microoperation in the CPU. The control word for a given microoperation can be derived from the selection variables. For example, the subtract microoperation given by the statement

$$R1 \leftarrow R2 - R3$$

specifies $R2$ for the A input of the ALU, $R3$ for the B input of the ALU, $R1$ for the destination register, and an ALU operation to subtract $A - B$. Thus the control word is specified by the four fields and the corresponding binary value for each field is obtained from the encoding listed in Tables 8-1 and 8-2. The binary control word for the subtract microoperation is 010 011 001 00101 and is obtained as follows:

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

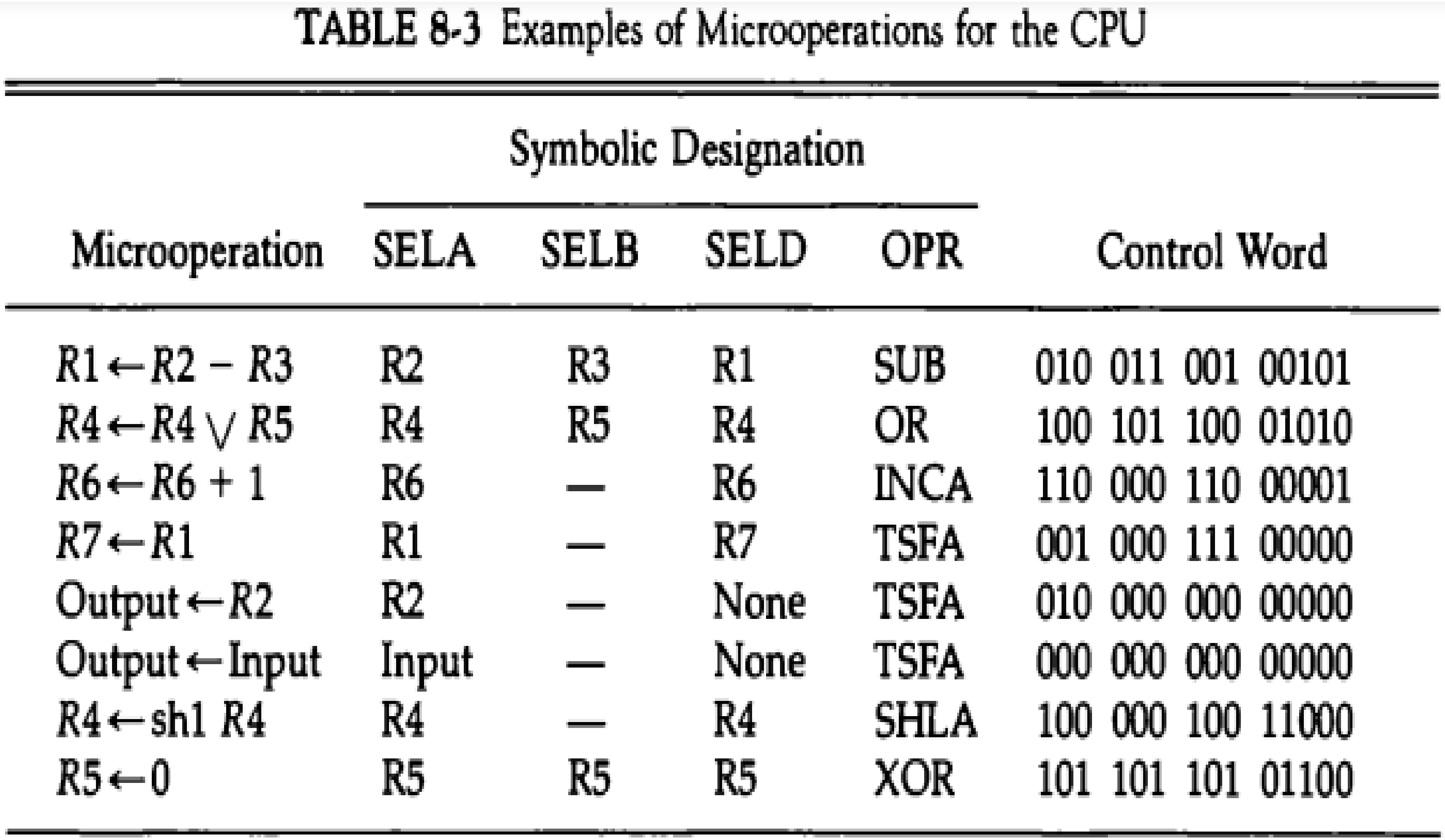


TABLE 8-3 Examples of Microoperations for the CPU

Microoperation	Symbolic Designation				Control Word
	SELA	SELB	SELD	OPR	
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	—	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	—	R7	TSFA	001 000 111 00000
Output ← $R2$	R2	—	None	TSFA	010 000 000 00000
Output ← Input	Input	—	None	TSFA	000 000 000 00000
$R4 \leftarrow$ sh1 $R4$	R4	—	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

Stack Organization

Section - 2

Stack Organization

- A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved (LIFO).
- The register that holds the address for the stack is called a *stack pointer (SP)* because its value always points at the top item in the stack.
- The physical registers of a stack are always available for reading or writing. It is the content of the word that is inserted or deleted.
- There are two types of stack organization
  1. Register stack – built using registers
  2. Memory stack – logical part of memory allocated as stack

Register Stack

PUSH Operation

SP ← SP + 1

M[SP] ← DR

IF (SP= 0) then (FULL ← 1)

EMTY ← 0

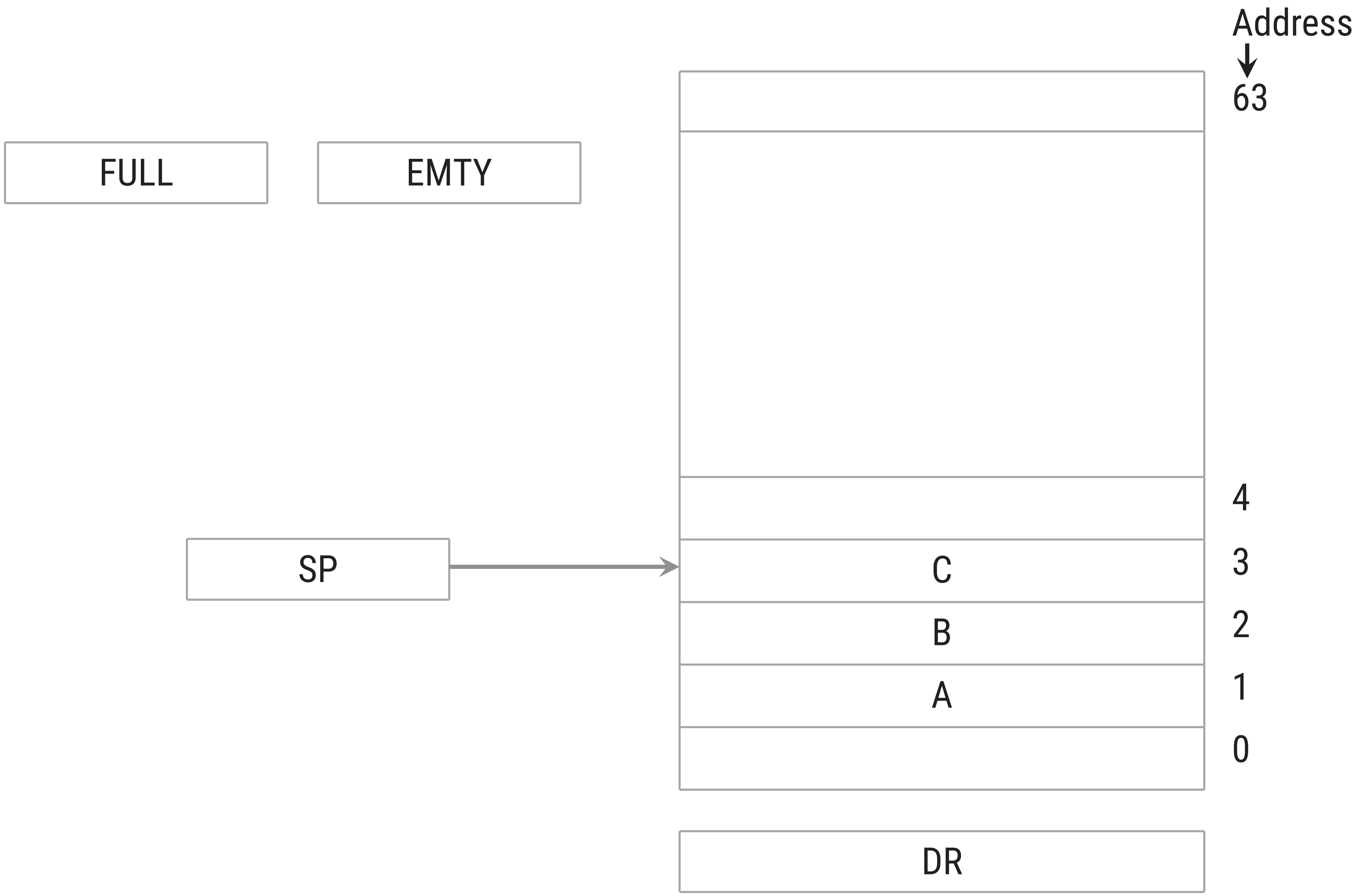
POP Operation

DR ← M[SP]

SP ← SP - 1

IF (SP= 0) then (EMTY ← 1)

FULL ← 0



Register Stack

- A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure shows the organization of a 64-word register stack.
- The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack.
- In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$.
- Since SP has only six bits, it cannot exceed a number greater than 63 (111111 in binary).
- The one-bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items.
- DR is the data register that holds the binary data to be written into or read out of the stack.

Memory Stack

- **PUSH Operation**

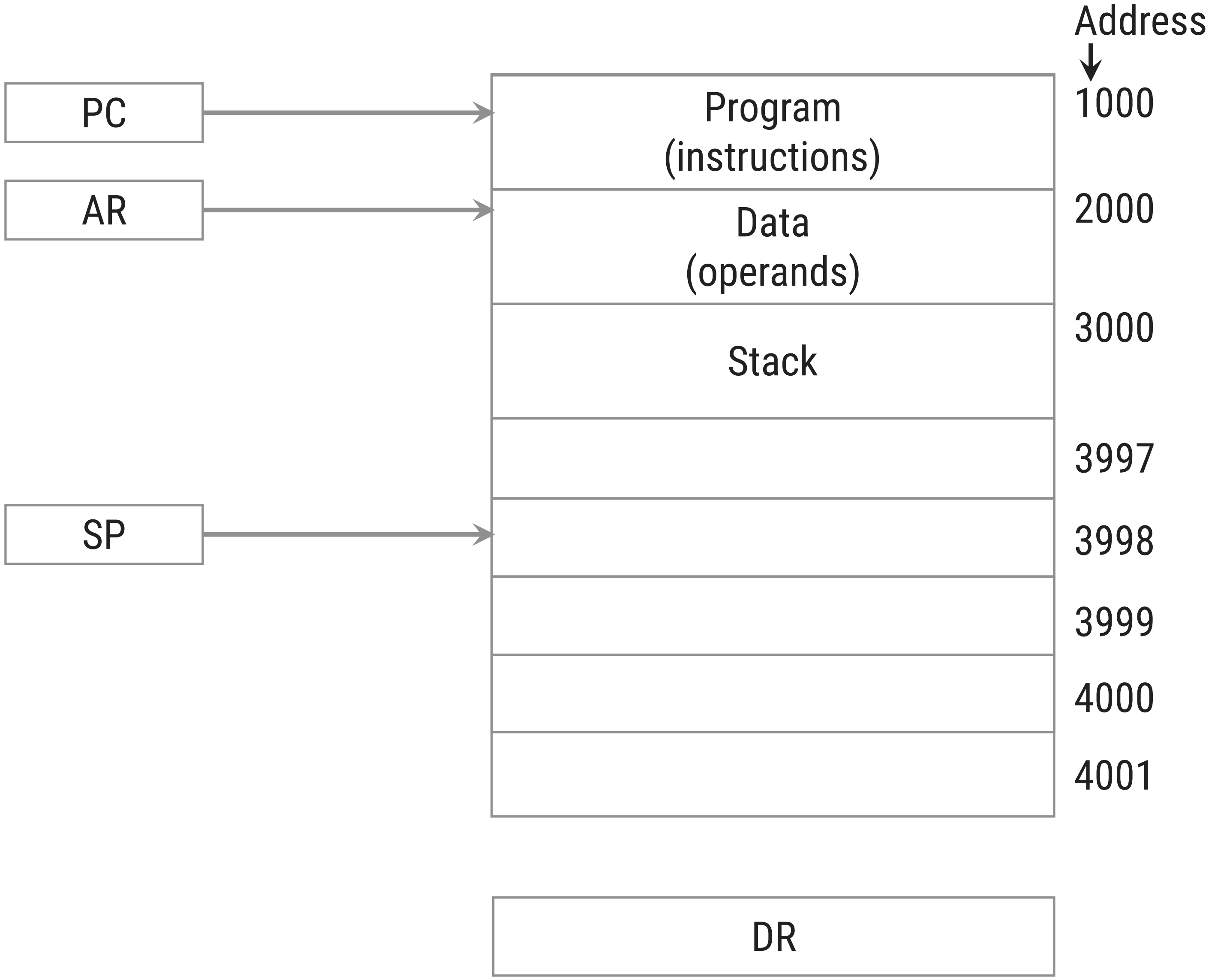
  SP ← SP - 1

  M[SP] ← DR

- **POP Operation**

  DR ← M[SP]

  SP ← SP + 1



Memory Stack

- The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer.
- Figure shows a portion of computer memory partitioned into three segments: program, data, and stack.
- The program counter PC points at the address of the next instruction in the program which is used during the fetch phase to read an instruction.
- The address registers AR points at an array of data which is used during the execute phase to read an operand.
- The stack pointer SP points at the top of the stack which is used to push or pop items into or from the stack.
- We assume that the items in the stack communicate with a data register DR.

Reverse Polish Notation

- The common mathematical method of writing arithmetic expressions imposes difficulties when evaluated by a computer.
- The Polish mathematician Lukasiewicz showed that arithmetic expressions can be represented in prefix notation as well as postfix notation.

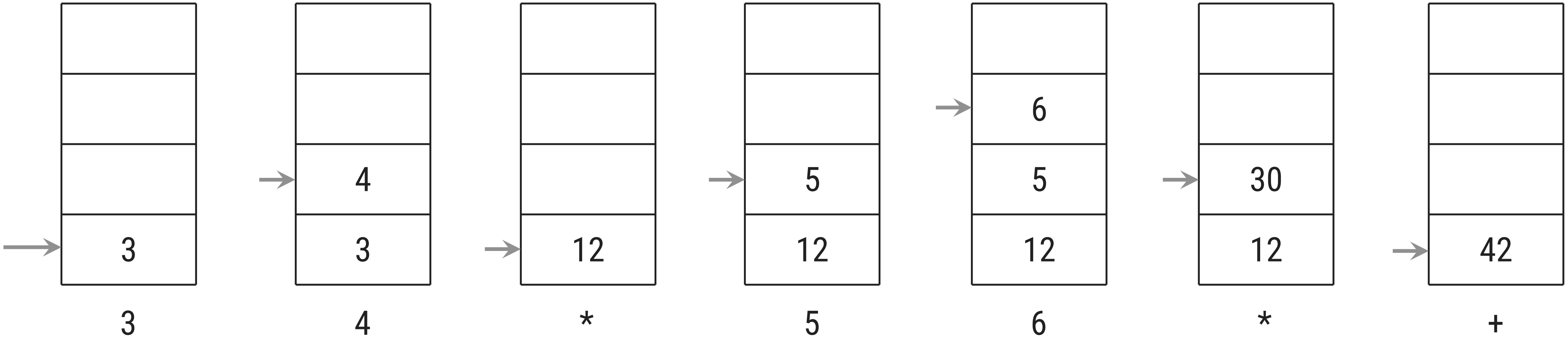
Infix	Prefix or Polish	Postfix or Reverse Polish
$A + B$	$+ AB$	$AB +$

$$A * B + C * D \longrightarrow AB * CD * +$$

Reverse Polish

Evaluation of Arithmetic Expression

(3 * 4) + (5 * 6) ⟶ 3 4 * 5 6 * + ⟶ 42



Instruction format

Section - 3

INSTRUCTION FORMAT

- Instruction Fields

OP-code field - specifies the operation to be performed

Address field - designates memory address(s) or a processor register(s)-Address of the memory and Address of the registers

Mode field - specifies the way the operand or the effective address is determined

- **The number of address fields in the instruction format depends on the internal organization of CPU**

- - The three most common CPU organizations

- **Single accumulator organization:**

```
ADD     X             /* AC ▪ AC + M[X]  */
```

General register organization:

```
ADD     R1, R2, R3      /* R1 ▪ R2 + R3  */
ADD     R1, R2        /* R1 ▪ R1 + R2  */
MOV     R1, R2        /* R1 ▪ R2  */
ADD     R1, X         /* R1 ▪ R1 + M[X]  */
```

Stack organization:

```
PUSH    X             /* TOS ▪ M[X]  */
ADD
```

Instruction Formats

- Instructions are categorized into different formats with respect to the operand fields in the instructions.

1. Three Address Instructions
2. Two Address Instruction
3. One Address Instruction
4. Zero Address Instruction

Three Address Instruction

- Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand.
- The program in assembly language that evaluates X = (A + B) * (C + D) is shown below.

```
ADD   R1, A, B       R1←M[A]+M[B]
ADD   R2, C, D       R2←M[C]+M[D]
MUL   X, R1, R2      M[X]←R1 * R2
```

- The advantage of three-address format is that it results in short programs when evaluating arithmetic expressions.
- The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

Two Address Instruction

- Two address instructions are the most common in commercial computers. Here again each address field can specify either a processor register or a memory word.
- The program to evaluate X = (A + B) * (C + D) is as follows:

```
MOV   R1, A           R1←M[A]
ADD   R1, B           R1←R1+M[B]
MOV   R2, C           R2←M[C]
ADD   R2, D           R2←R2+M[D]
MUL   R1, R2          R1←R1 * R2
MOV   X, R1           M[X]←R1
```

One Address Instruction

- One address instructions use an implied accumulator (AC) register for all data manipulation.
- For multiplication and division these is a need for a second register.
- However, here we will neglect the second register and assume that the AC contains the result of all operations.
- The program to evaluate X = (A + B) * (C + D) is

```
LOAD   A          AC← M[A]
ADD    B          AC← AC+M[B]
STORE           T M[T]←AC
LOAD   C          AC← M[C]
ADD    D          AC← AC+M[D]
MUL    T          AC← AC*M[T]
STORE           X M[X]←AC
```

Zero Address Instruction

- A stack-organized computer does not use an address field for the instructions ADD and MUL.
- The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack.
- The program to evaluate X = (A + B) * (C + D) will be written for a stack-organized computer.
- To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse polish notation.

```
PUSH A        TOS← M[A]
PUSH B        TOS← M[B]
ADD           TOS←(A+B)
PUSH C        TOS← M[C]
PUSH D        TOS← M[D]
ADD           TOS←(C+D)
MUL           TOS←(C+D)∗(A+B)
POP   X       M[X] ←TOS
```

- $(3 + 4)\ [10\ (2 + 6) + 8]$

Addressing Modes

Section - 4

Addressing Modes

- The way of choosing operands during program execution is dependent on addressing modes of the instruction.
- Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:
  1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
  2. To reduce the number of bits in the addressing field of the instruction.
- There are basic 10 addressing modes supported by the computer.

1. Implied Mode
2. Immediate Mode
3. Register Mode
4. Register Indirect Mode
5. Autoincrement or Autodecrement Mode
6. Direct Address Mode
7. Indirect Address Mode
8. Relative Address Mode
9. Indexed Addressing Mode
10. Base Register Addressing Mode

1. Implied Mode & 2. Immediate Mode

1. Implied Mode

- Operands are specified *implicitly* in the definition of the instruction.
- For example, the instruction "complement accumulator (CMA)" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction.
- In fact, all register reference instructions that use an accumulator and zero address instructions are implied mode instructions.

2. Immediate Mode

- Operand is specified in the instruction itself.
- In other words, an immediate-mode instruction has an operand field rather than an address field.
- The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction.
- Immediate mode of instructions is useful for initializing register to constant value.
- E.g. MOV R1, 05H

  instruction copies immediate number 05H to R1 register.

3. Register Mode & 4. Register Indirect Mode

3. Register Mode

- Operands are in registers that reside within the CPU.
- The particular register is selected from a register field in the instruction.
- E.g. MOV AX, BX

  move value from BX to AX register

4. Register Indirect Mode

- In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory.
- Before using a register indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction.
- The advantage of this mode is that address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.
- E.g. MOV [R1], R2

  value of R2 is moved to the memory location specified in R1.

5. Autoincrement or Autodecrement Mode & 6. Direct Address Mode

5. Autoincrement or Autodecrement Mode

- This is similar to the register indirect mode expect that the register is incremented or decremented after (or before) its value is used to access memory.
- When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table. This can be achieved by using the increment or decrement instruction.

6. Direct Address Mode

- In this mode the effective address is equal to the address part of the instruction.
- The operand resides in memory and its address is given directly by the address field of the instruction.
- E.g. ADD 457

7. Indirect Address Mode & 8. Relative Address Mode

7. Indirect Address Mode

- In this mode the address field of the instruction gives the address where the effective address is stored in memory.
- Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.
- The effective address in this mode is obtained from the following computational:

8. Relative Address Mode

- In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.
- The address part of the instruction is usually a signed number which can be either positive or negative.

Effective address = address part of instruction + content of PC

9. Indexed Addressing Mode & 10. Base Register Addressing Mode

9. Indexed Addressing Mode

- In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.
- The indexed register is a special CPU register that contain an index value.
- The address field of the instruction defines the beginning address of a data array in memory.
- Each operand in the array is stored in memory relative to the beginning address.

Effective address = address part of instruction + content of index register

10. Base Register Addressing Mode

- In this mode the content of a base register is added to the address part of the instruction to obtain the effective address.
- A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address.
- The base register addressing mode is used in computers to facilitate the relocation of programs in memory.

Effective address = address part of instruction + content of base register

Addressing Modes (Example)

PC = 200

R1 = 400

XR = 100

AC

Address	Memory	
200	Load to AC	Mode
201	Address = 500	
202	Next instruction	
399	450	
400	700	
500	800	
600	900	
702	325	
800	300	

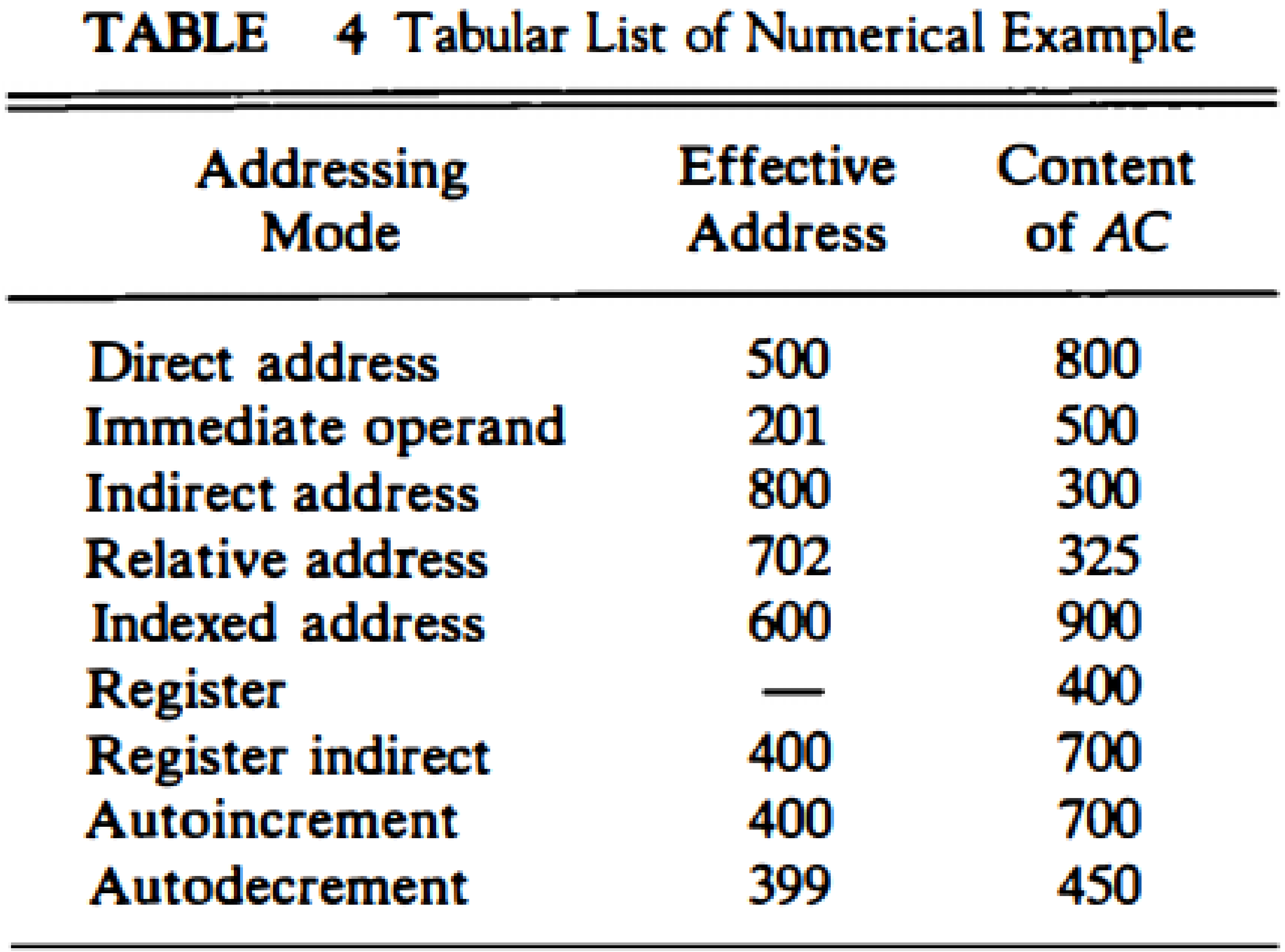


TABLE 4 Tabular List of Numerical Example

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

UNIT-III
Computer Arithmetic

✓ Outline

- **Addition and Subtraction**

  Addition & Subtraction with signed-magnitude data.
  Addition & Subtraction with signed 2’s complement data.

- **Multiplication Algorithms** (Booth Multiplication Algorithm)

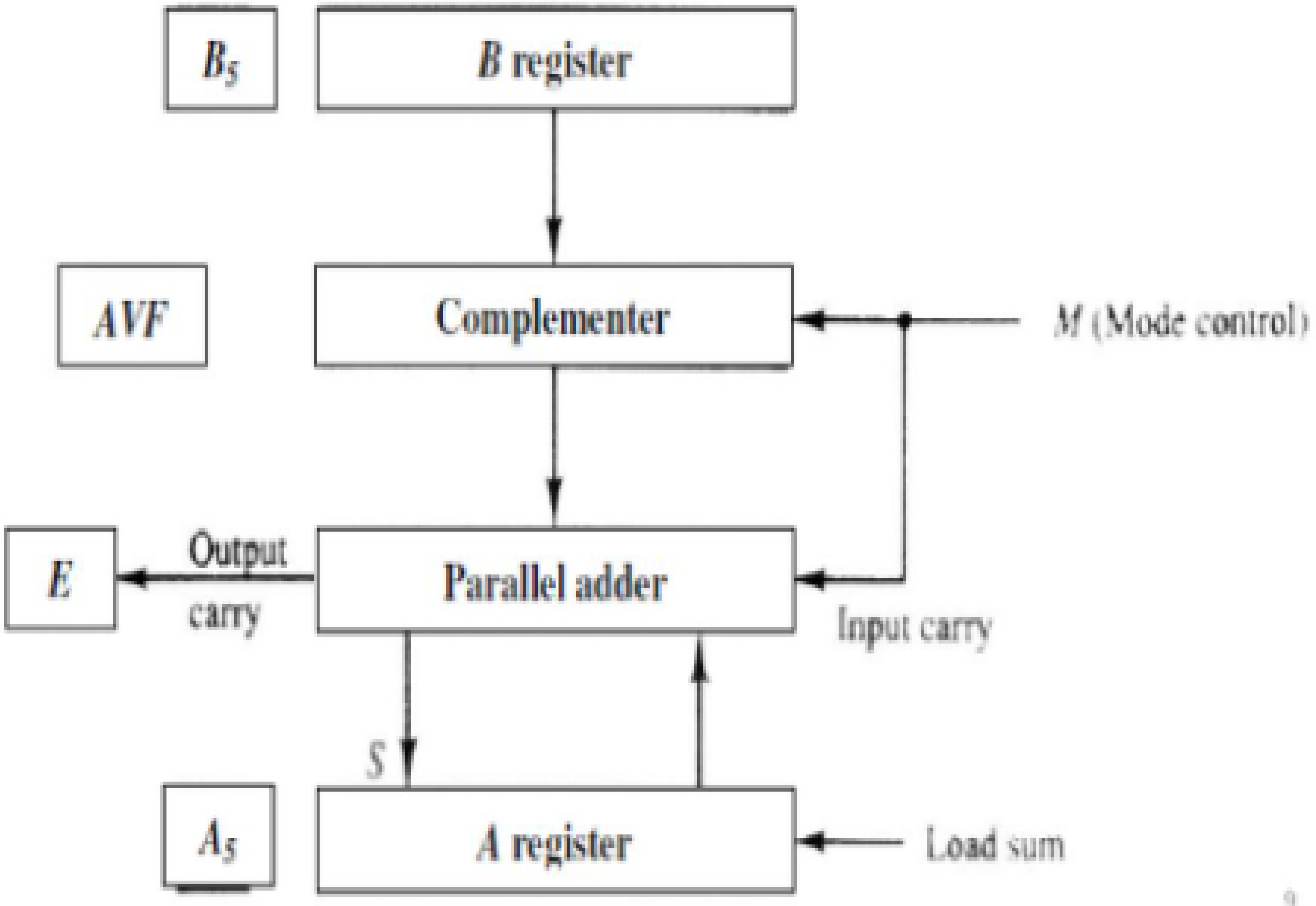
Addition and Subtraction

Section - 1

Addition and Subtraction

Operation	Add Magnitudes	Subtract Magnitudes		
		When A B	When A B	When A = B
$(+A) + (+B)$	$+ (A + B)$			
$(+A) + (-B)$		$+ (A - B)$	$- (B - A)$	$+ (A - B)$
$(-A) + (+B)$		$- (A - B)$	$+ (B - A)$	$+ (A - B)$
$(-A) + (-B)$	$- (A + B)$			
$(+A) - (+B)$		$+ (A - B)$	$- (B - A)$	$+ (A - B)$
$(+A) - (-B)$	$+ (A + B)$			
$(-A) - (+B)$	$- (A + B)$			
$(-A) - (-B)$		$- (A - B)$	$+ (B - A)$	$+ (A - B)$

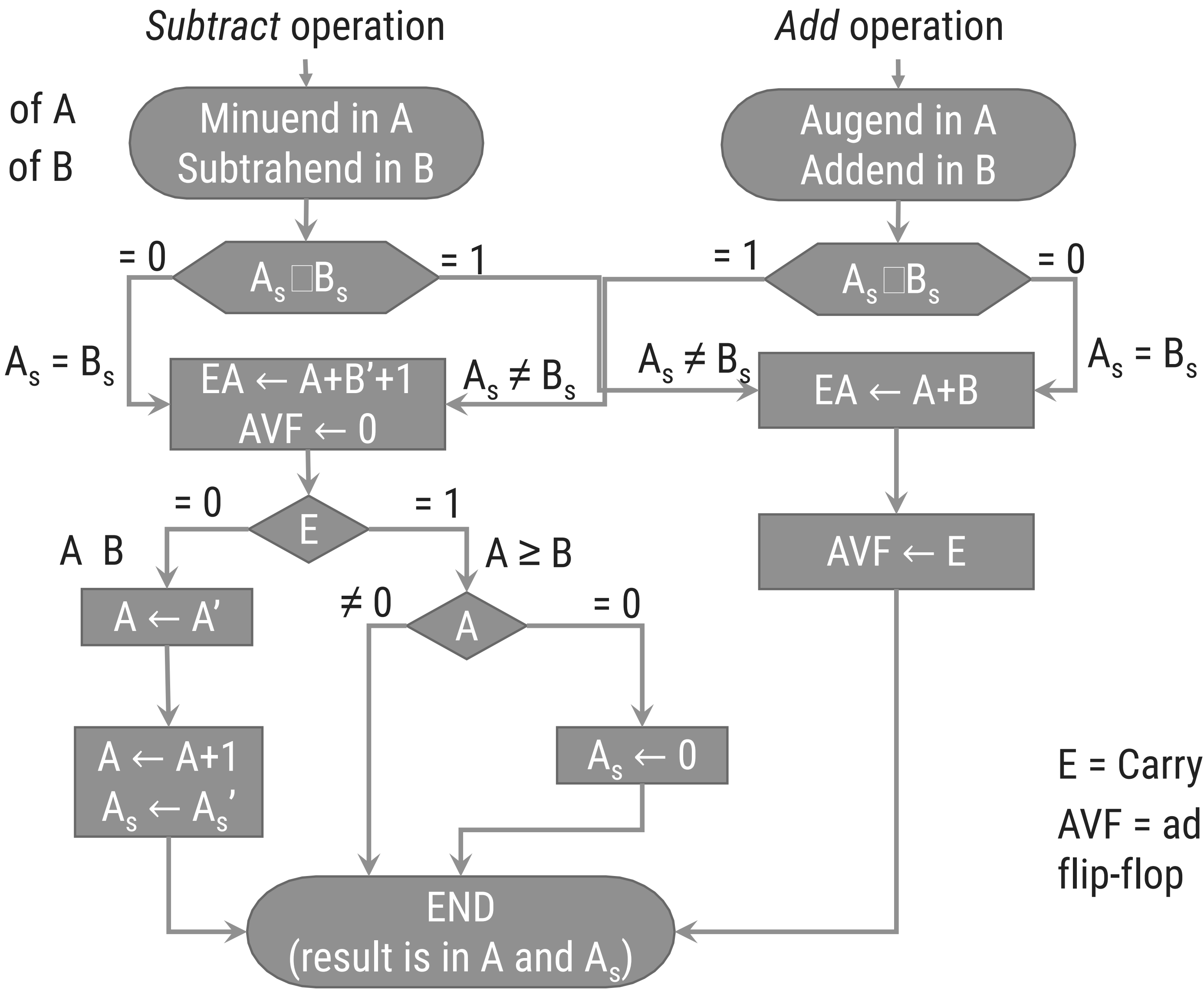
Hardware Implementation



Flowchart for Addition & Subtraction

A_s = Sign bit of A

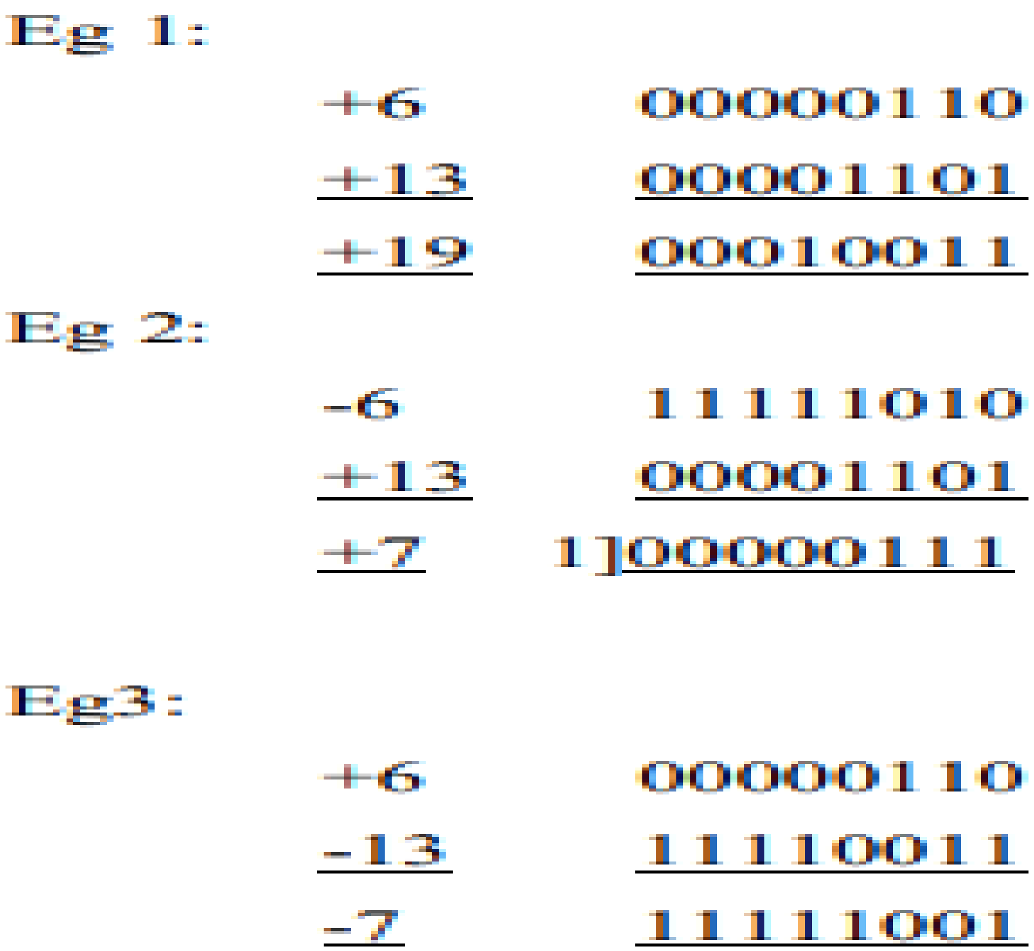
B_s = Sign bit of B



E = Carry

AVF = add overflow flip-flop

2'S COMPLEMENT ADDITION



Eg 1:

+6 00000110

+13 00001101

+19 00010011

Eg 2:

-6 11111010

+13 00001101

+7 1]00000111

Eg3:

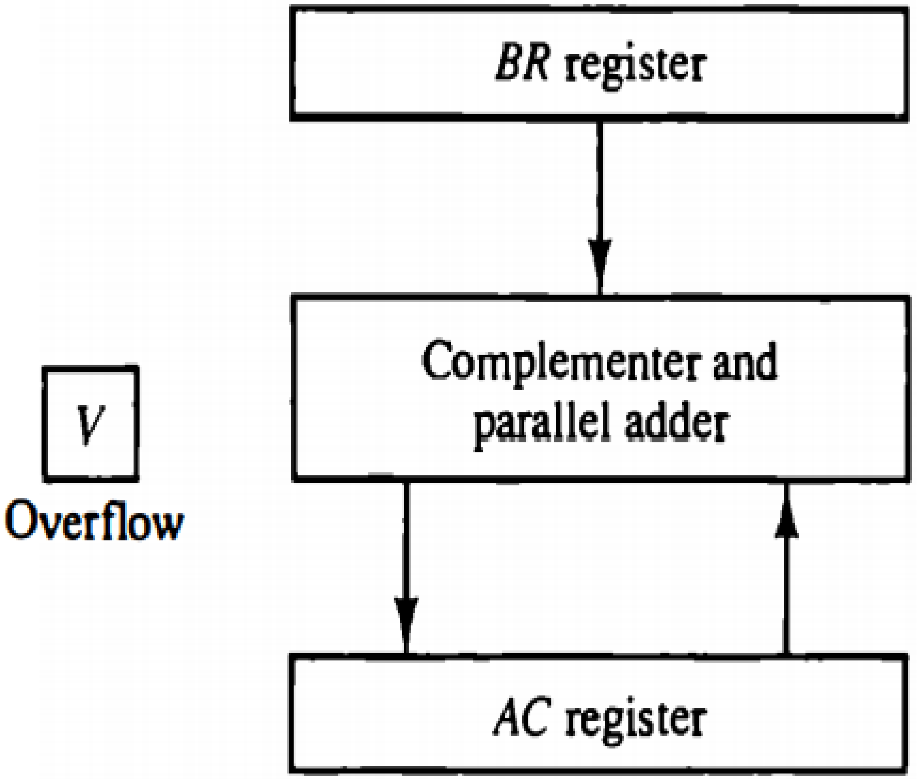
+6 00000110

-13 11110011

-7 11111001

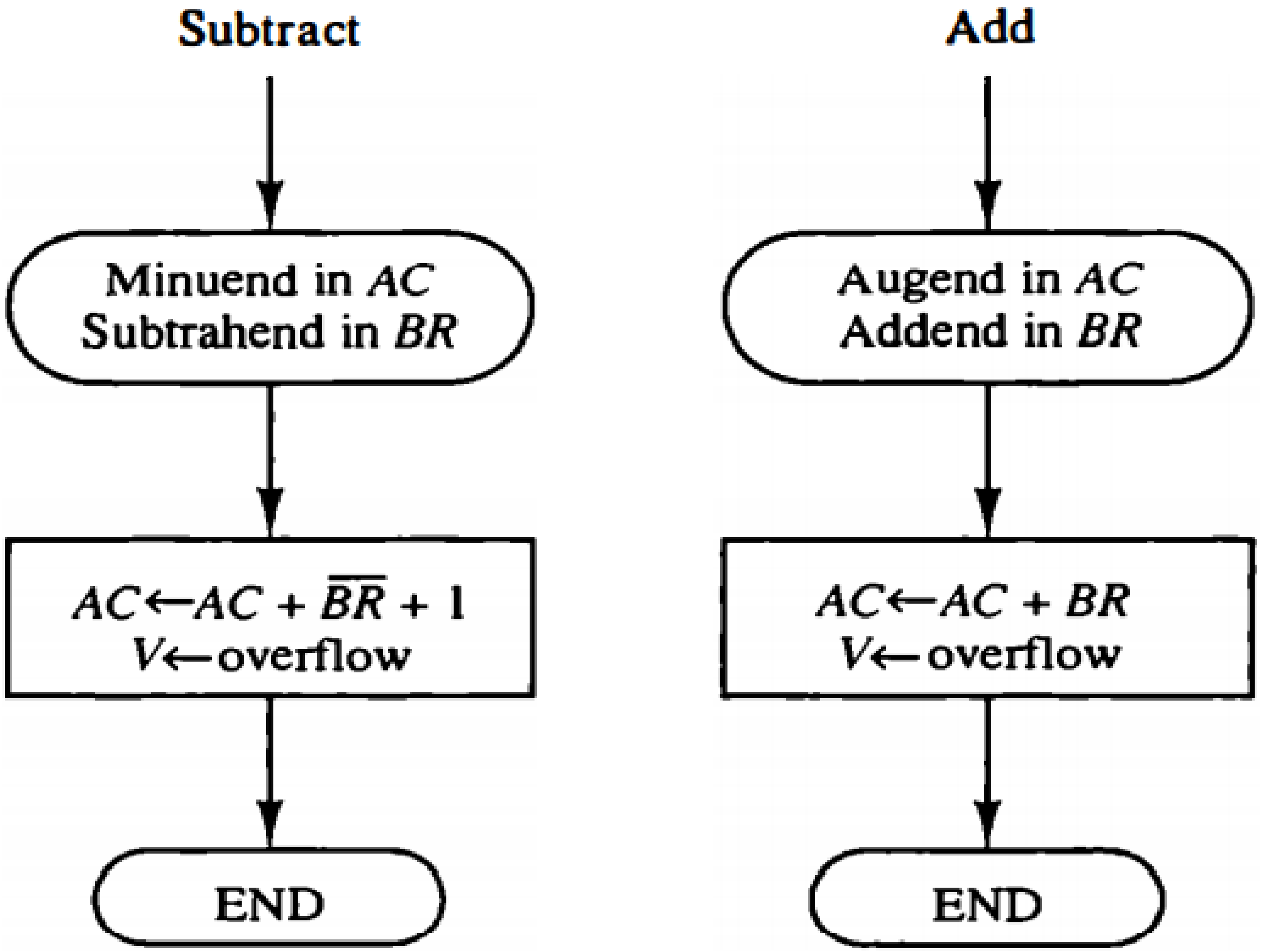
Addition and subtraction using signed 2’s complement data

Hardware for signed-2's complement addition and subtraction.



+33=00100001

Algorithm for adding and subtracting numbers in signed 2s complement representation.



Subtract

Minuend in *AC*
Subtrahend in *BR*

$AC \leftarrow AC + \overline{BR} + 1$
$V \leftarrow$ overflow

END

Add

Augend in *AC*
Addend in *BR*

$AC \leftarrow AC + BR$
$V \leftarrow$ overflow

END

- Perform the arithmetic operations below with binary numbers and with negative numbers In slgned-2s complement representation. Use seven bits to accommodate each number together with its sign. In each case, determine if there is an overflow by checking the carries Into and out of the sign bit position .
- (+35) + (+40)
- b. (-35) + (-40)
- c. (-35) - (+40)

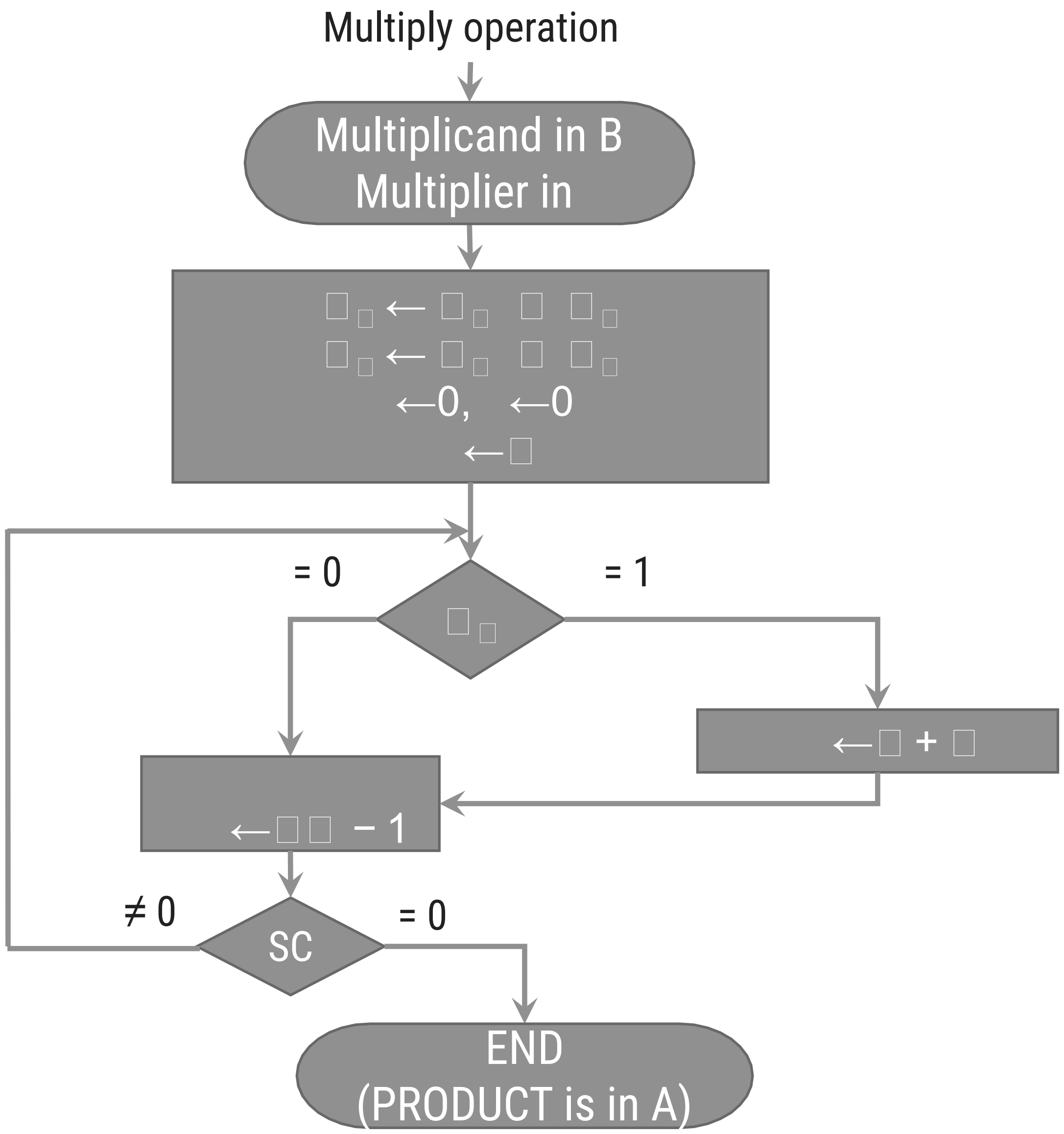
Multiplication Algorithms (Booth Multiplication Algorithm)

Section - 2

Multiplication

Traditional way of binary multiplication

```
   23       10111
  x19     x 10011
  ---     -------
            10111
           10111
          00000
         00000
        10111
      ---------
  437   110110101
```



Multiply operation

Multiplicand in B
Multiplier in

□□ ← □□ □ □□
□□ ← □□ □ □□
←0, ←0
←□

□□

= 0

= 1

←□ + □

←□□ − 1

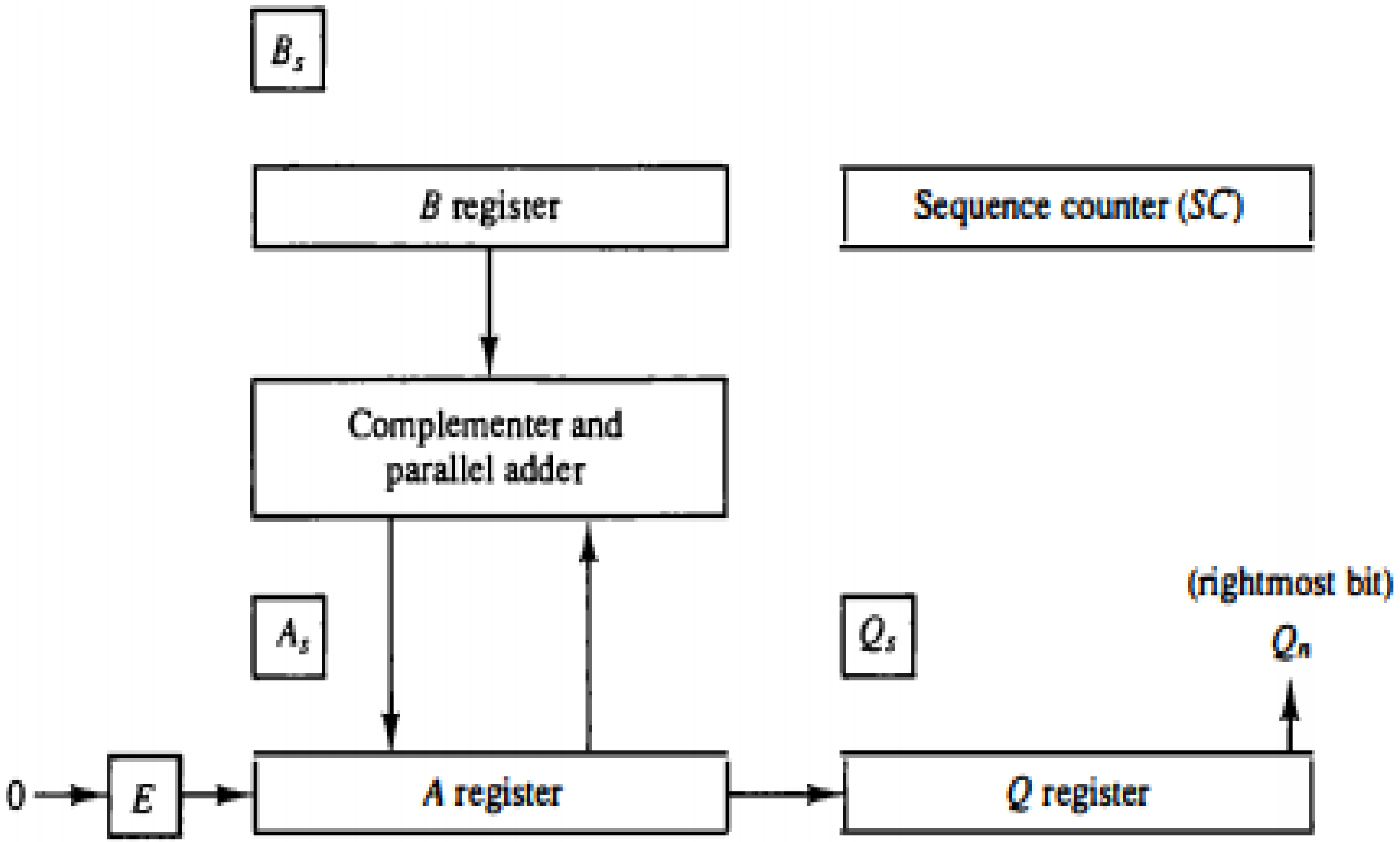
SC

≠ 0

= 0

END
(PRODUCT is in A)

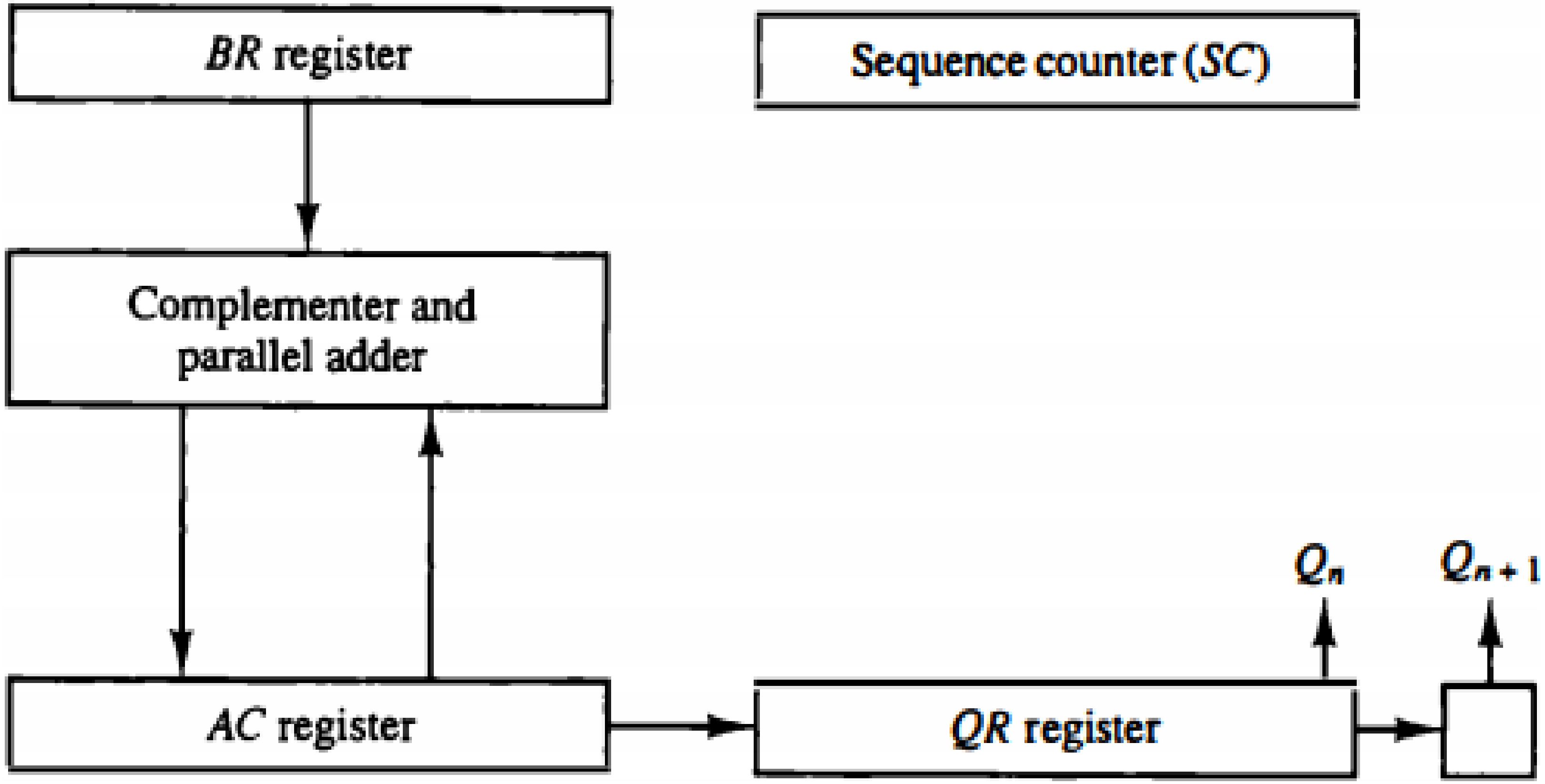
Hardware Implementation for signed magnitude data.



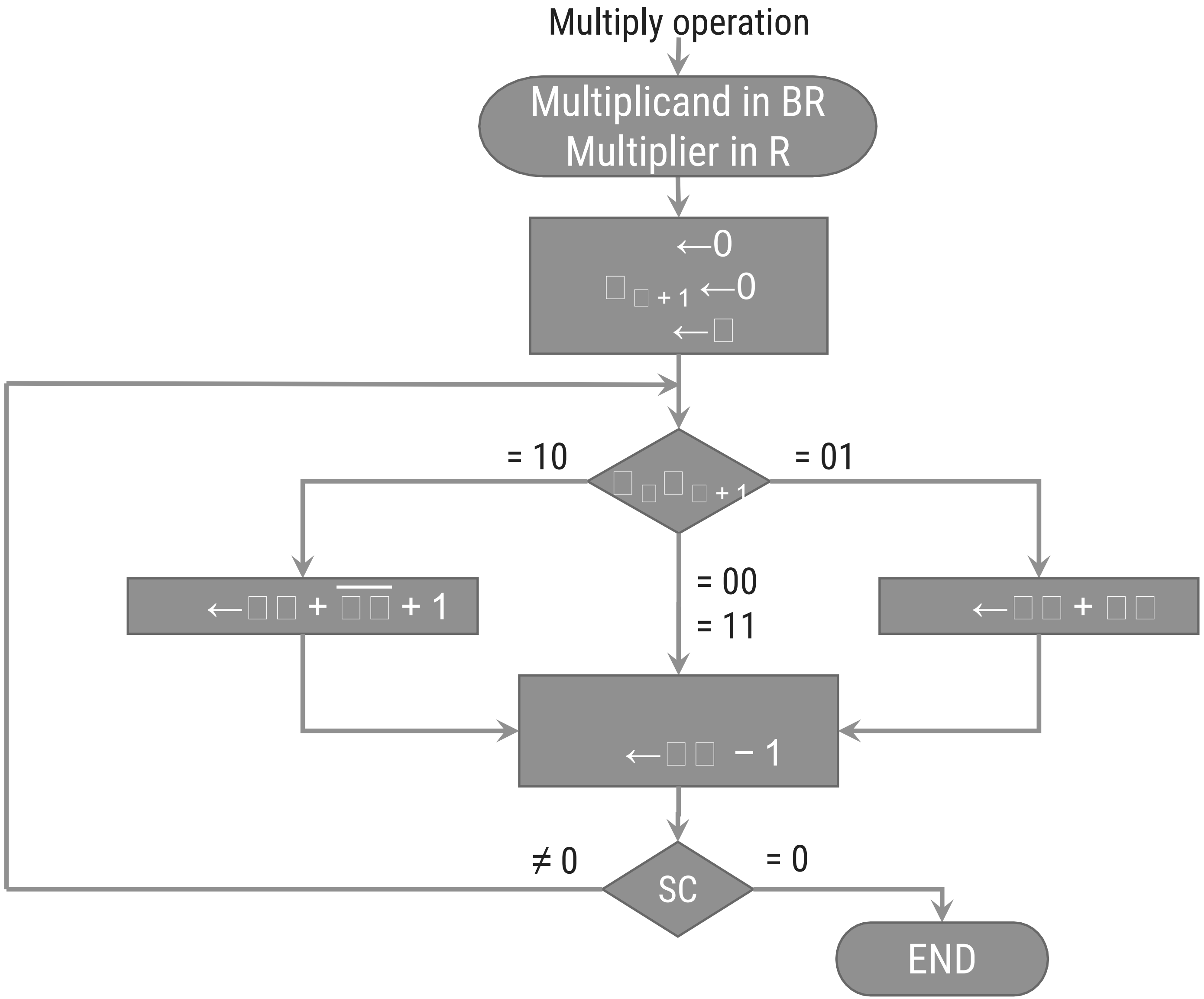
Perform 23 x 19

Multiplicand B = 10111	**E**	**A**		**SC**
Multiplier in	0	00000	10011	101
${}_n$ = 1 add B		10111		
First partial product	0	10111		
Shift right EA	0	01011	11001	100
${}_n$ = 1 add B		10111		
Second partial product	1	00010		
Shift right EA	0	10001	01100	011
${}_n$ = 0 shift right EA	0	01000	10110	010
${}_n$ = 0 shift right EA	0	00100	01011	001
${}_n$ = 1 add B		10111		
Fifth partial product	0	11011		
Shift right EA	0	01101	10101	000
Final product in A = 0110110101				

Booth Multiplication Hardware



Booth Multiplication Algorithm



Multiply operation

Multiplicand in BR
Multiplier in R

←0
□□ + 1 ←0
←□

= 10

□□□□ + 1

= 01

←□□ + □□ + 1

= 00
= 11

←□□ + □□

←□□ − 1

≠ 0

SC

= 0

END

Multiply (-9) x (-13) using Booth Algorithm

□□	□□ + □	□□ = 10111 (-9) □□ + □ = 01001	□□	(-13) □□	□□ + □	□□
		Initial	00000	10011	0	101
1	0	Subtract BR	01001			
			01001			
		ashr	00100	11001	1	100
1	1	ashr	00010	01100	1	011
0	1	Add BR	10111			
			11001			
		ashr	11100	10110	0	010
0	0	ashr	11110	01011	0	001
1	0	Subtract BR	01001			
			00111			
		ashr	00011	10101	1	000

Qn Qn+1

00
11

AC & QR

01

AC+BR

ashr

10

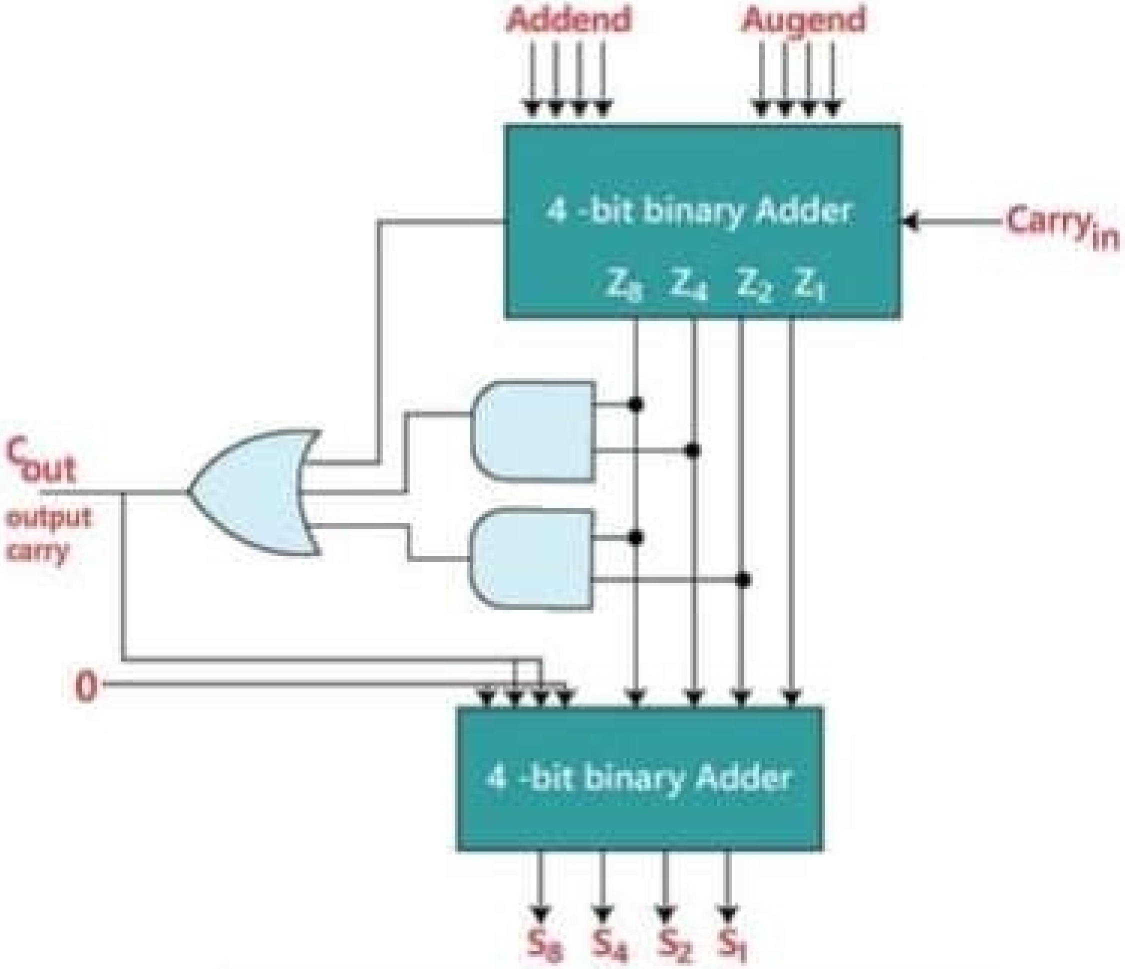
AC+B̄R+1

ashr

- Show the step-by-step multiplication process using Booth algorithm when the following binary numbers are multiplied. Assume 5-bit registers that hold signed numbers. The multiplicand in both cases is + 15.
- a. (+15) x (+13) b. (+15) X (-13)

Decimal Arithmetic unit

- A CPU with an arithmetic logic unit can perform arithmetic microoperations with binary data.
- A decimal arithmetic unit is a digital function that performs decimal microoperations.
- It can add or subtract decimal numbers, usually by forming the 9’s or 10’s complements of the subtrahend.
- The unit accepts coded decimal numbers and generates results in the same adopted binary code.



Block diagram of a BCD adder

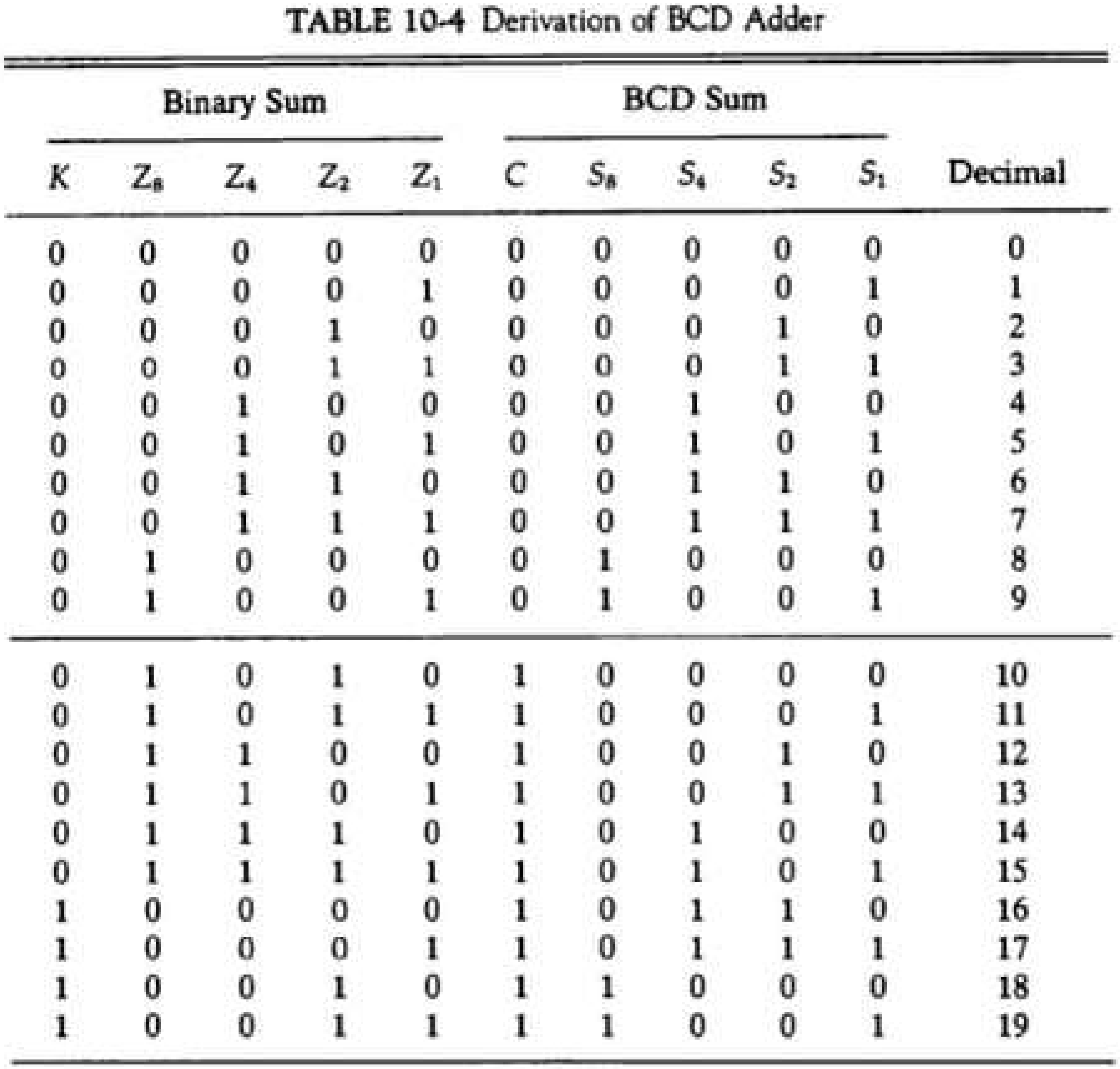


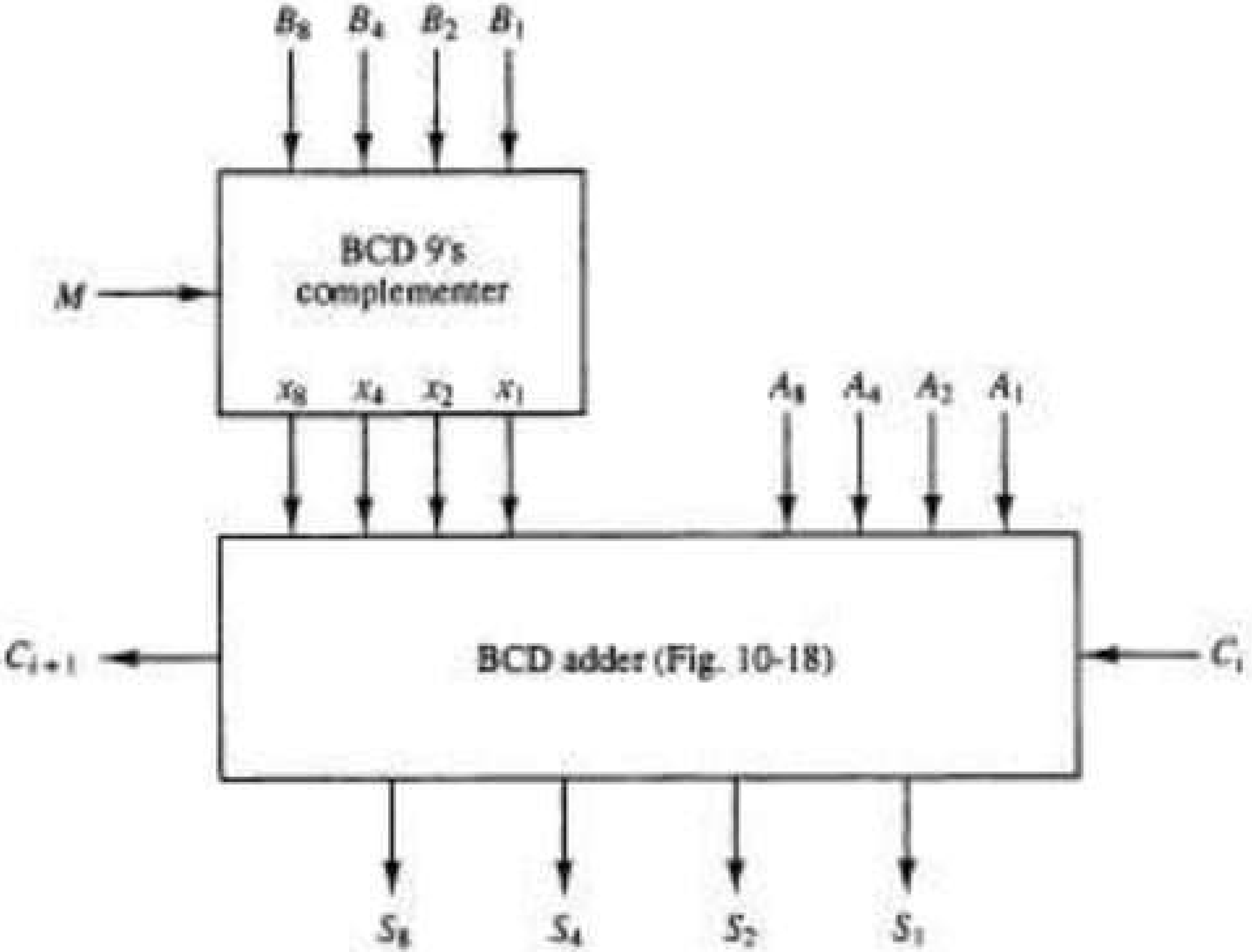
TABLE 10-4 Derivation of BCD Adder

Binary Sum					BCD Sum					
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	Decimal
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

CD Sutraction

- ⮐⮑⇆ ⌅▶⇦◀△⇨⇨◀↓⇕↔ ↓▽ ▲⇨△←⇕△↘⇨⇦ ⇦◢ ⇨⇦⇦↓↔→ ⇹ʹ▽ ⇨⇕↘▲↙⇨↘⇨↔◀ ⇕← ◀↑⇨ ▽▶⇦◀△⇨↑⇨↔⇦⇲
- ⌃↑⇨ ↳▽ ⮑⇕↘▲↙⇨↘⇨↔◀ ⇕← ⇨ ⇦⇨⇨↓↘⇨↙ ⇦↓→↓◀ ↓▽ △⇨▲△⇨▽⇨↔◀⇨⇦ ↓↔ ⮐⮑⇆ ↘⇨◢ ⇦⇨ ⇕⇦◀⇨↓↔⇨⇦ ▶▽↓↔→ ↕ ↘⇨◀↑⇕⇦▽⇲
- ⮍⇨◀↑⇕⇦↤↰ ⮐↓↔⇨△◢ ⇹⇹↘⇦⇨⇨↓↘⇨↙ ⇹⇤ ↓▽

Figure 10-19 One stage of a decimal arithmetic unit.



Decimal Arithmetic operations

- Decimal Arithmetic Unit is a digital function that performs decimal microoperations.
- can add or subtract decimal numbers, usually by forming the 9's or 10's complement of the subtrahend.
- The unit accepts coded decimal numbers and generate
- A single-stage decimal arithmetic unit consists of:
  - nine binary input variables
  - five binary output variables (since a minimum of four bits is required to represent each coded decimal digit)
- Each stage must have four inputs for the augends digit, four inputs for the addend digit, and an input-carry. s results in the same adopted binary code.
- The outputs include four terminals for the sum digit and one for the output-carry.

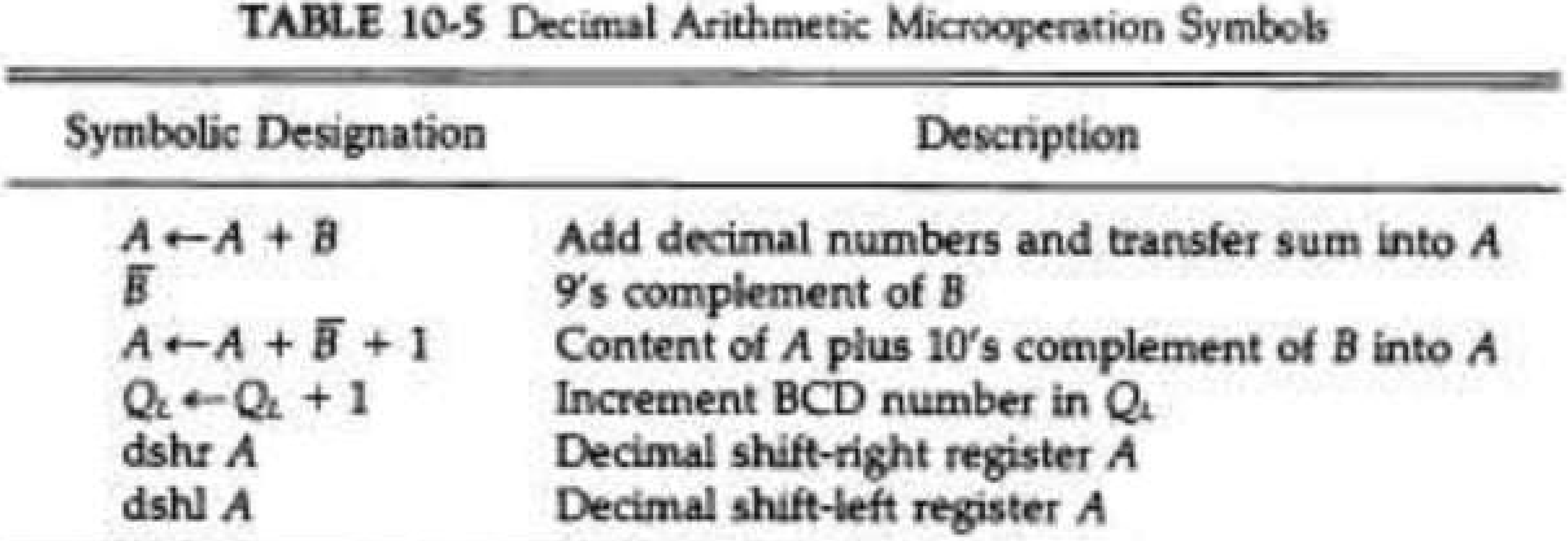
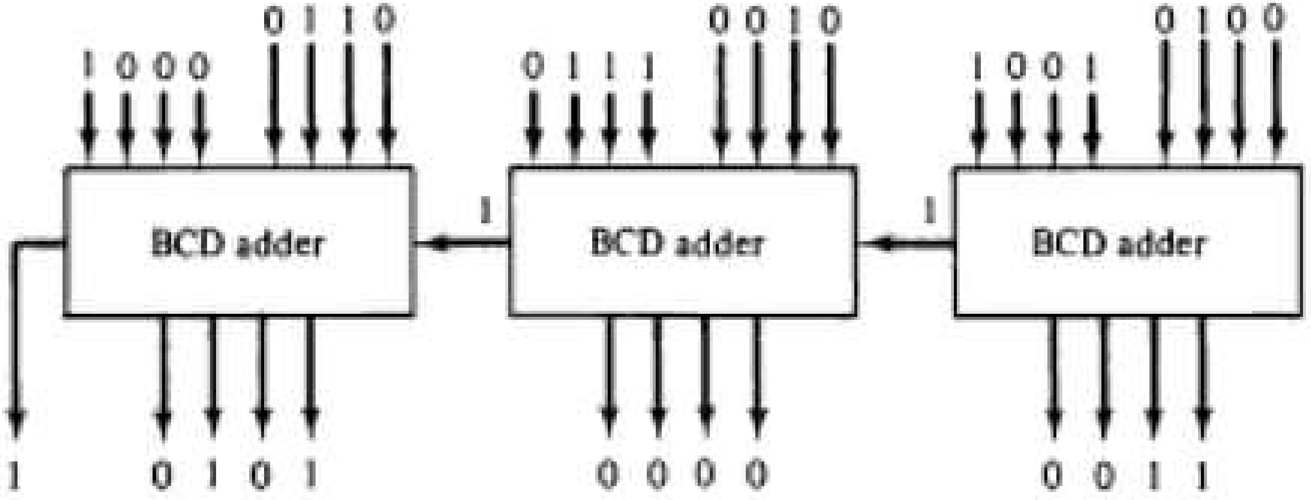


TABLE 10-5 Decimal Arithmetic Microoperation Symbols

Symbolic Designation	Description
$A \leftarrow A + B$	Add decimal numbers and transfer sum into A
$\overline{B}$	9's complement of B
$A \leftarrow A + \overline{B} + 1$	Content of A plus 10's complement of B into A
$Q_L \leftarrow Q_L + 1$	Increment BCD number in Q_L
dshr A	Decimal shift-right register A
dshl A	Decimal shift-left register A

Addition and Subtraction

- **Addition**
  - Parallel decimal adder
  - Digit-serial. Bit-parallel decimal addition
  - All serial decimal addition



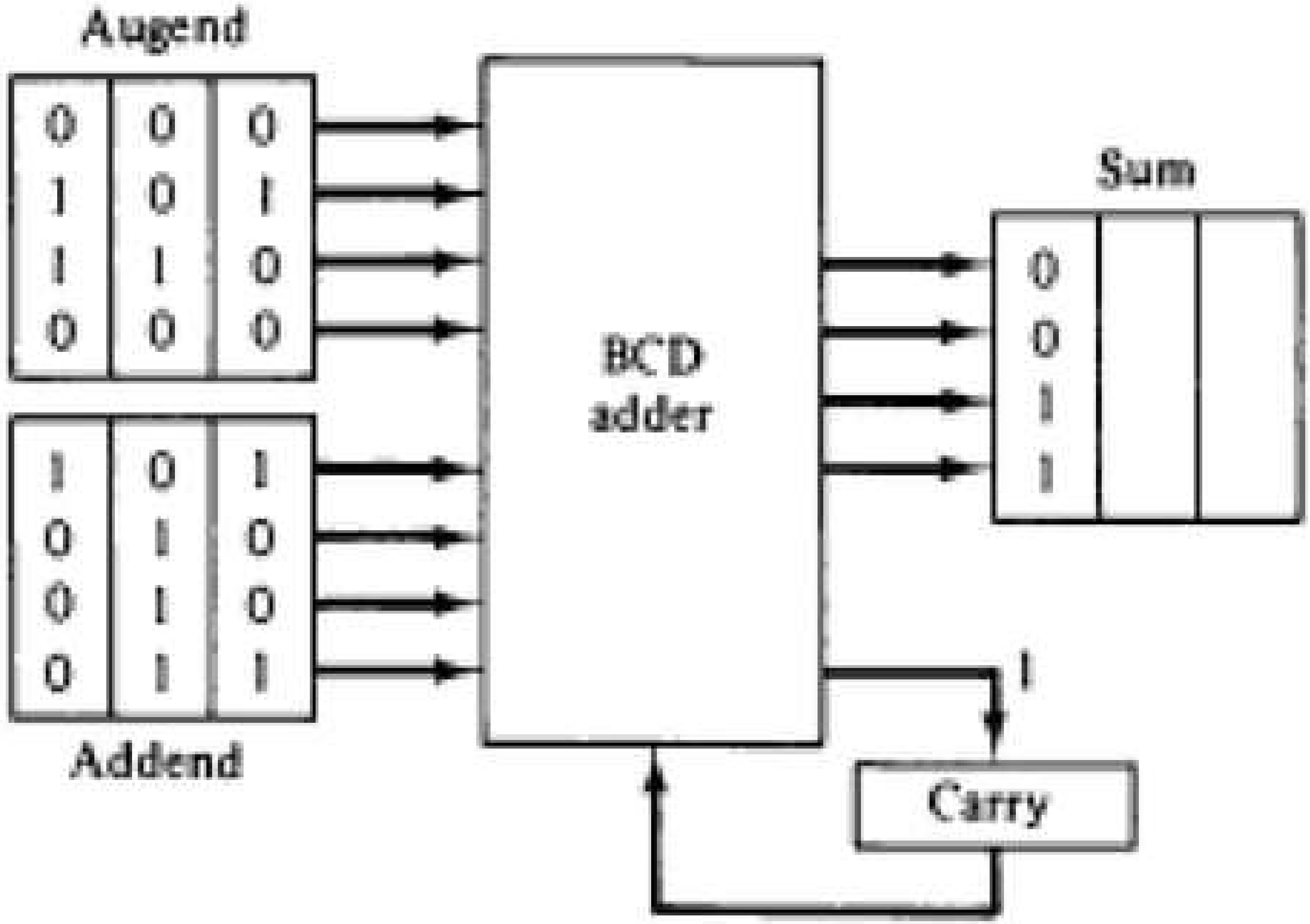
(a) Parallel decimal addition: 624 + 879 = 1503

↔⇲↻⇨△⇨↙↙⇨↙
⇆⇨⇨↓↘⇨↙ ↳⇦⇦⇨△

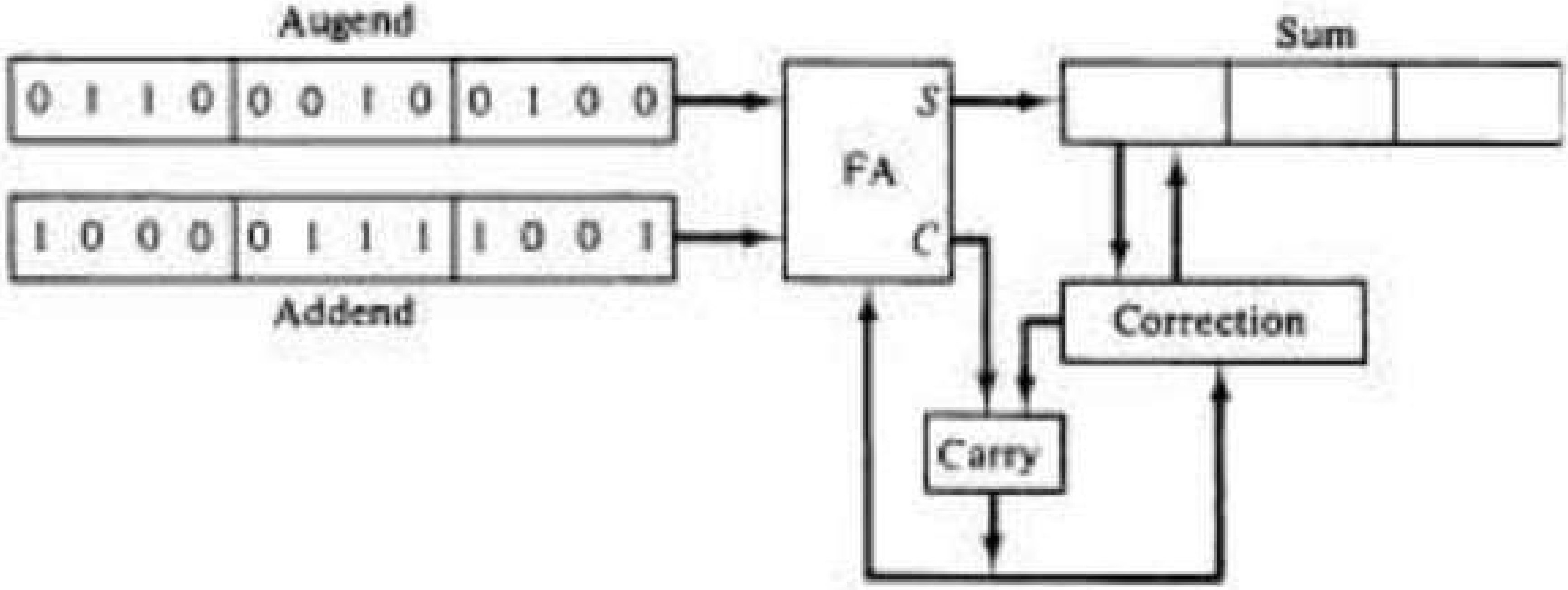
- ^↑⇨ ▲⇨△⇨↙↙⇨↙ ↘⇨◀↑↕⇦ ▶▽⇨▽ ⇨ ⇦⇨⇨↓↘⇨↙ ⇨◀↑⇨↘⇨◀↓⇨ ▶⇔↓◀ ⇨↕↘▲↕▽⇨⇦ ↕← ⇨▽ ↘⇨⇔◢ ⮐⮑⇆ ↳⇦⇦⇨△▽ ⇨▽ ◀↑⇨△⇨ ⇨△⇨ ⇦↓→↓◀▽ ↓⇔ ◀↑⇨ ⇔▶↘⇦⇨△⇲ ^↑⇨ ▽▶↘ ↓▽ ←↕△↘⇨⇦ ↓⇔ ▲⇨△⇨↙↙⇨↙ △⇨▼▶↓△⇨▽ ↕⇔↙◢ ↕⇔⇨ ↘↓⇨△↑⇲↑▲⇨△⇨◀↓↑⇔↘

↕↘ ⇆↓→↓◀ –▽⇨△↓⇨↙ ⇦↓◀⇱▲⇨△⇨↙↙⇨↙ ↳⇦⇦⇨△

- ^↑⇨ ⇦↓→↓◀▽ ⇨△⇨ ⇨▲▲↙↓⇨⇦ ◀↕ ⇨ ▽↓↔→↙⇨ ⇥⇤⇆ ↳⇦⇦⇨△ ▽⇨△↓⇨↙↙◢⤓ ▷↑↓↙⇨ ◀↑⇨ ⇦↓◀▽ ↕← ⇨⇨⇨↑ ⇨↕⇦⇨⇦ ⇦↓→↓◀ ⇨△⇨ ◀△⇨↔▽←⇨△△⇨⇦ ↓↔ ▲⇨△⇨↙↙⇨↙↘ ^↑⇨ ^▶↘ ↓▽ ←↕△↘⇨⇦ ⇦◢ ▽↑↓←◀↓↔→ ◀↑⇨ ⇦⇨⇨↓↘⇨↙ ↔▶↘⇦⇨△▽ ◀↑△↑▶→↑ ◀↑⇨ ⇥⇤← ↳⇦⇦⇨△



(b) Digit-serial, bit-parallel decimal addition



(c) All serial decimal addition

Figure 10-20 Three ways of adding decimal numbers.