Computer Arithmetic Addition and Subtraction

Presented by

A.Srinivasan, Associate Professor

Addition and Subtraction with Signed magnitude data

- There are 3 ways of representing negative fixed point binary numbers. They are
  1. Signed Magnitude representation.
  2. Signed one's Complement representation
  3. Signed two's Complement representation
- Most computers use the Signed two's Complement representation when performing operation on integers.
- Consider the magnitude of any two numbers A and B and the eight different operation are listed below depending on the sign of the number.

Eight Conditions for Signed-Magnitude Addition/Subtraction

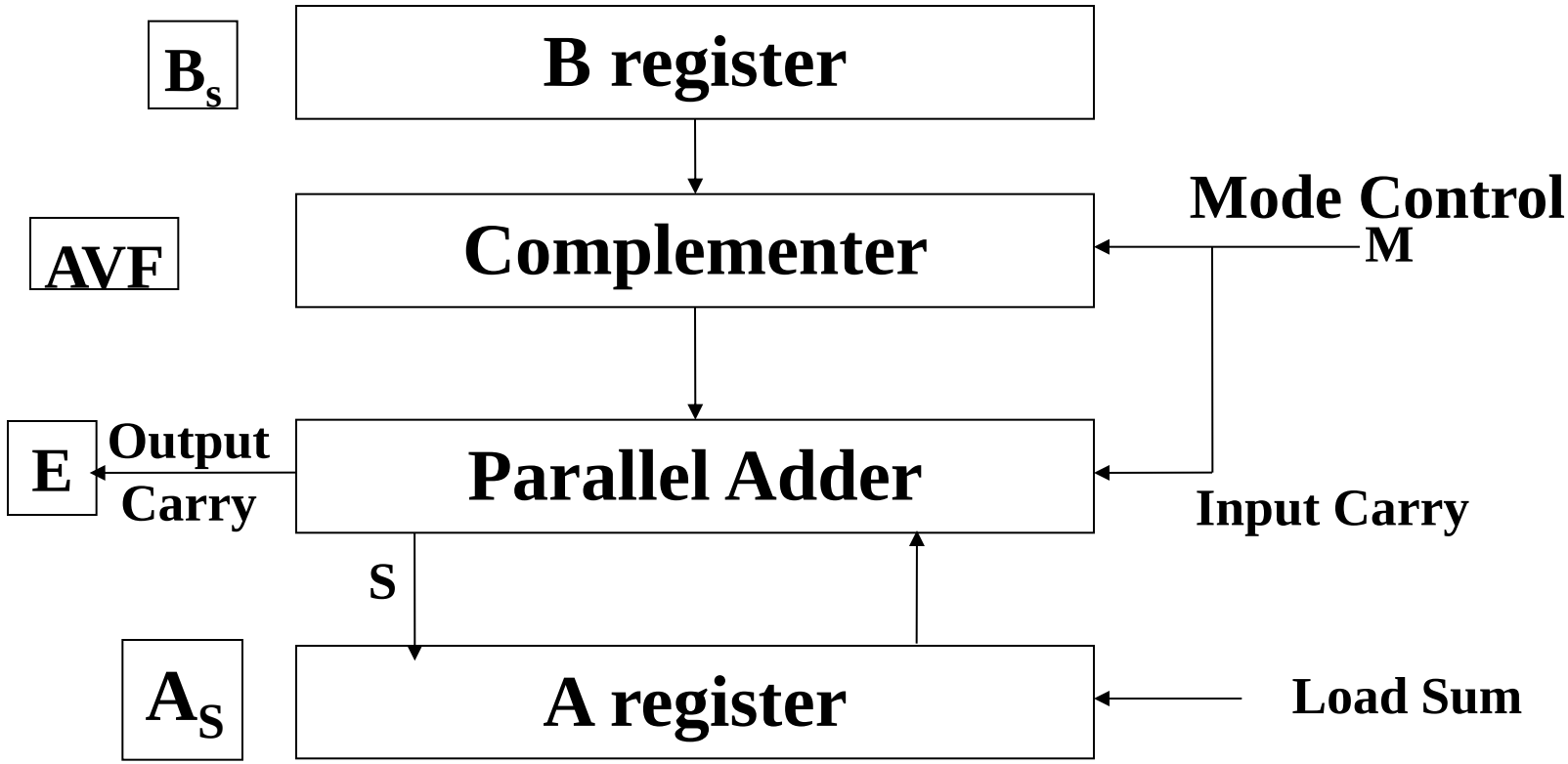
	Operation	ADD Magnitudes	SUBTRACT Magnitudes		
			A > B	A < B	A = B
1	(+A) + (+B)	+ (A + B)			
2	(+A) + (-B)		+ (A – B)	- (B – A)	+ (A – B)
3	(-A) + (+B)		- (A – B)	+ (B – A)	+ (A – B)
4	(-A) + (-B)	- (A + B)			
5	(+A) - (+B)		+ (A – B)	- (B – A)	+ (A – B)
6	(+A) - (-B)	+ (A + B)			
7	(-A) - (+B)	- (A + B)			
8	(-A) - (-B)		- (A – B)	+ (B – A)	+ (A – B)

Addition and Subtraction with Signed magnitude data

ALGORITHM:

- When the sign of A and B are identical, add the two magnitudes and attach the sign of A to the result.
- When the sign of A and B are different, compare the magnitudes, subtract smaller number from the larger.
- Choose the sign of the result to be same as A if A>B or complement the sign of A if A<B.
- If the two magnitudes are equal, subtract B from A and Make the sign of the result positive.

Hardware for signed-magnitude addition and subtraction



Hardware for signed-magnitude addition and subtraction

- Let A and B be the two registers that holds the magnitudes of the numbers and As and Bs be two flipflops that holds the corresponding sign
- The result of the operation may be transferred to the third register or the result is transferred to A and As.
- First parallel adder is needed to perform microoperation A+B.
- Second comparator circuit needed to establish if A<B, A>B or A==B.
- Third subtractor circuit is needed to perform the microoperation A-B and B-A.

Hardware for signed-magnitude addition and subtraction

- The block diagram consist of register A and B and the sign flipflops As and Bs. Subtraction is done by adding A to the 2's complement of B.
- The o/p carry is transferred to E. The add overflow flipflop(AVF) holds the overflow bit when A and B are added.
- The addition A+B is done through the parallel adder and the sum is transferred to A register.
- When the Mode bit M=0 the o/p of B is transferred to the adder, the i/p carry is 0 and the o/p of the adder is equal to sum A+B
- When M=1, the 1's complement of B is applied to adder, the i/p carry is 1 and the o/p is equal to A+ B'+1.

Hardware Algorithm

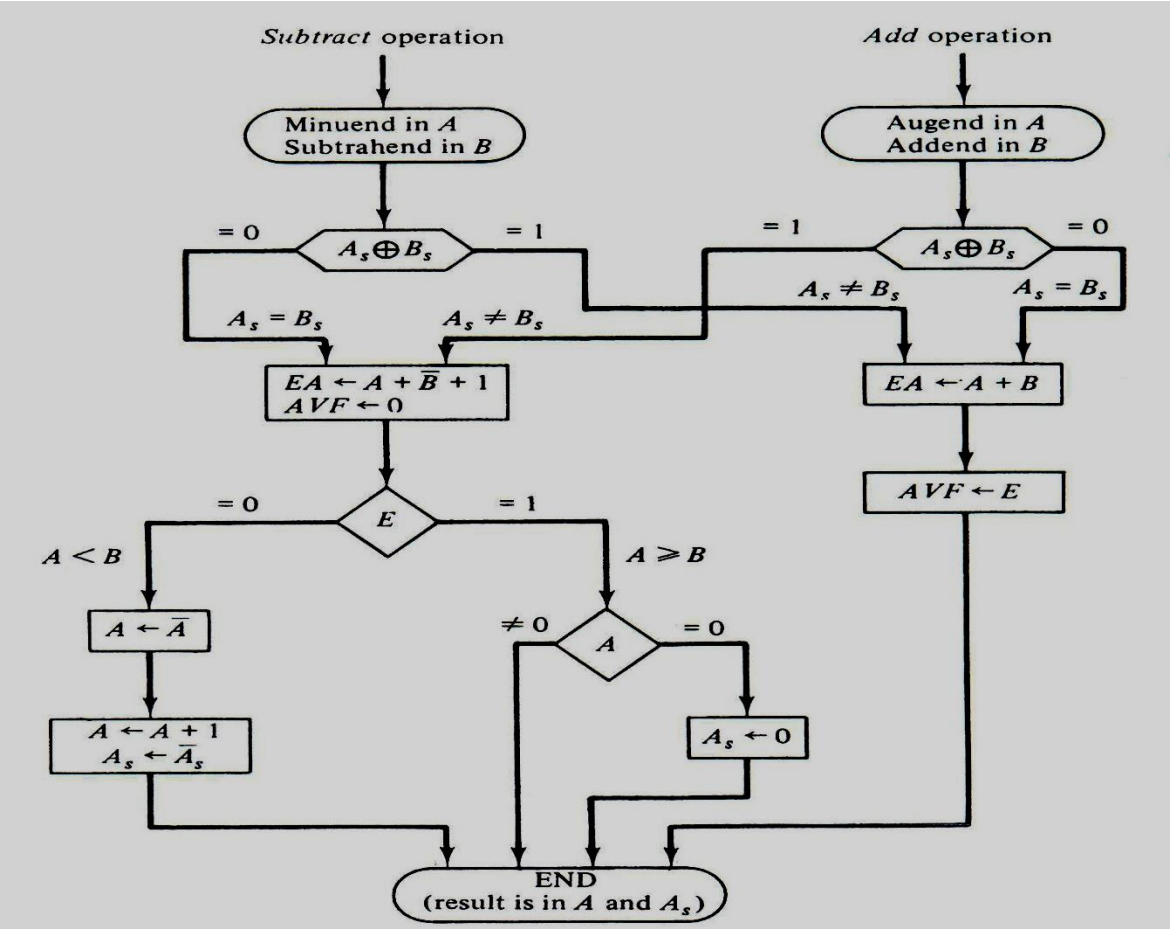


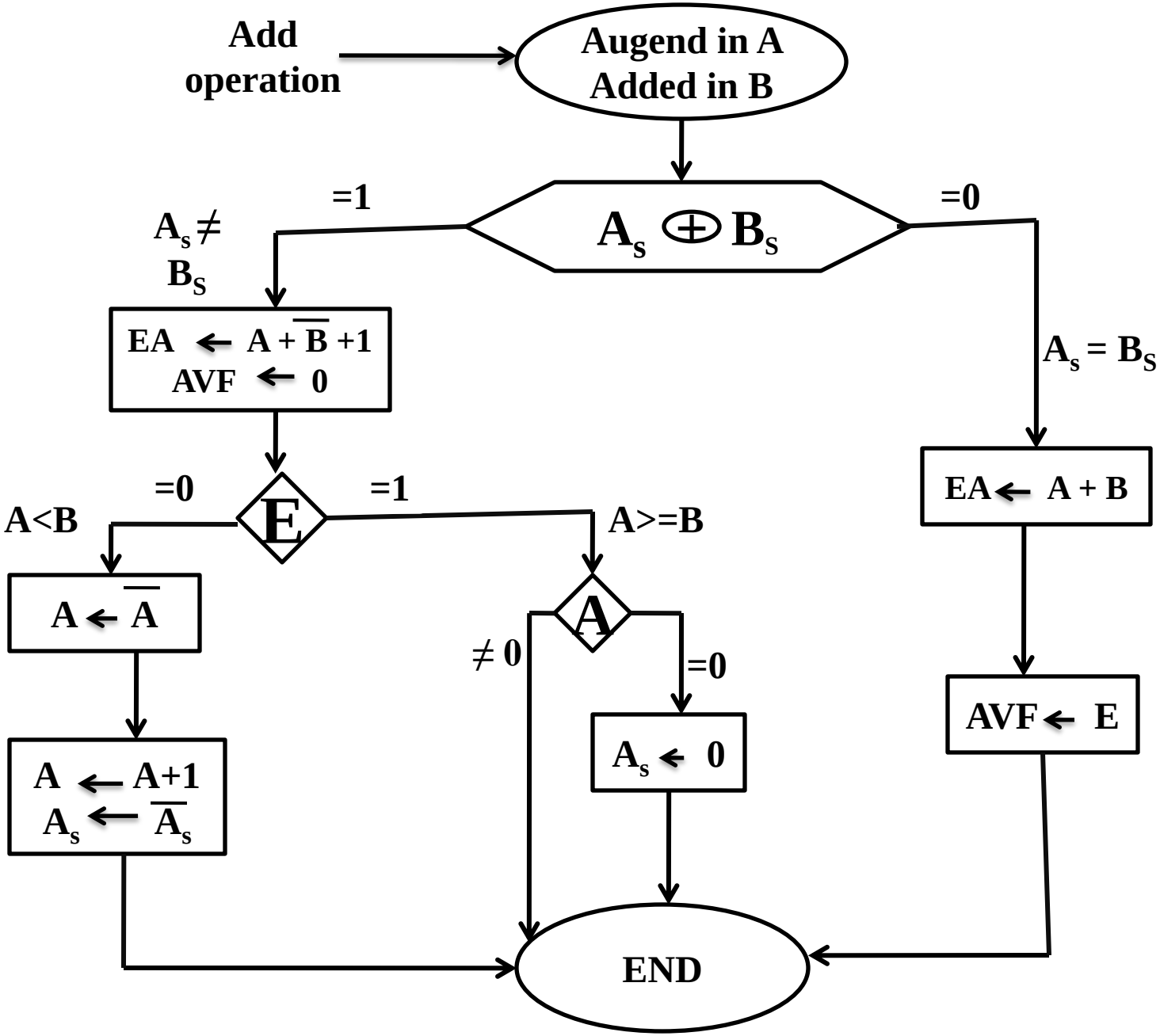
Figure 10-2 Flowchart for add and subtract operations.

Hardware Algorithm

- The two sign bits As and Bs are compared by XOR gate. If the o/p is 0, the sign are identical and if the o/p is 1, the sign are different.
- For an add operation the identical sign indicates that magnitudes are to be added.
- For the subtraction operation different sign indicate that magnitude are to added.
- The magnitudes are added with microoperation EA=A+B.

Hardware Algorithm

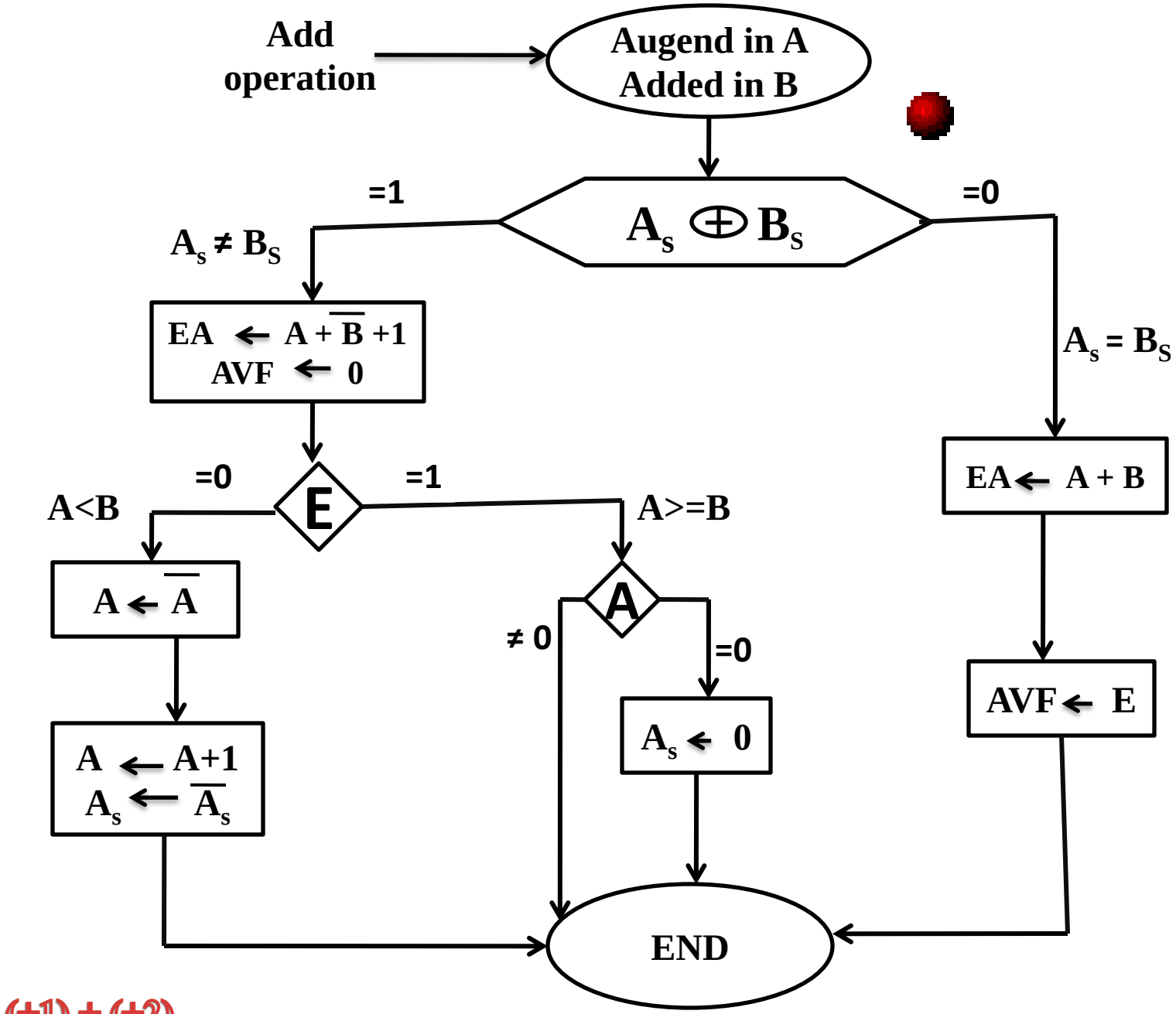
- The two magnitudes are subtracted if the sign are different for an add operation or identical for subtraction operation.
- If E=1, then the condition is A>=B and the number in A is the correct result.
- If E=0 then the condition is A<B and the number in A is taken 2's complement which is the correct result.
- If the sign of the result is same as the sign of A, So no change in As is required.
- When A<B the sign of the result is the complement of the original sign of A.
- The Final result is found in register A and its sign in As.


Add operation
Augend in A
Added in B
$A_s \oplus B_S$
=1
=0
$A_s \neq B_S$
$A_s = B_S$
EA ← A + $\overline{B}$ +1
AVF ← 0
EA ← A + B
E
=0
=1
A<B
A>=B
A ← $\overline{A}$
A
≠ 0
=0
AVF ← E
A ← A+1
A_s ← $\overline{A}_s$
A_s ← 0
END

- For Example of **Addition**

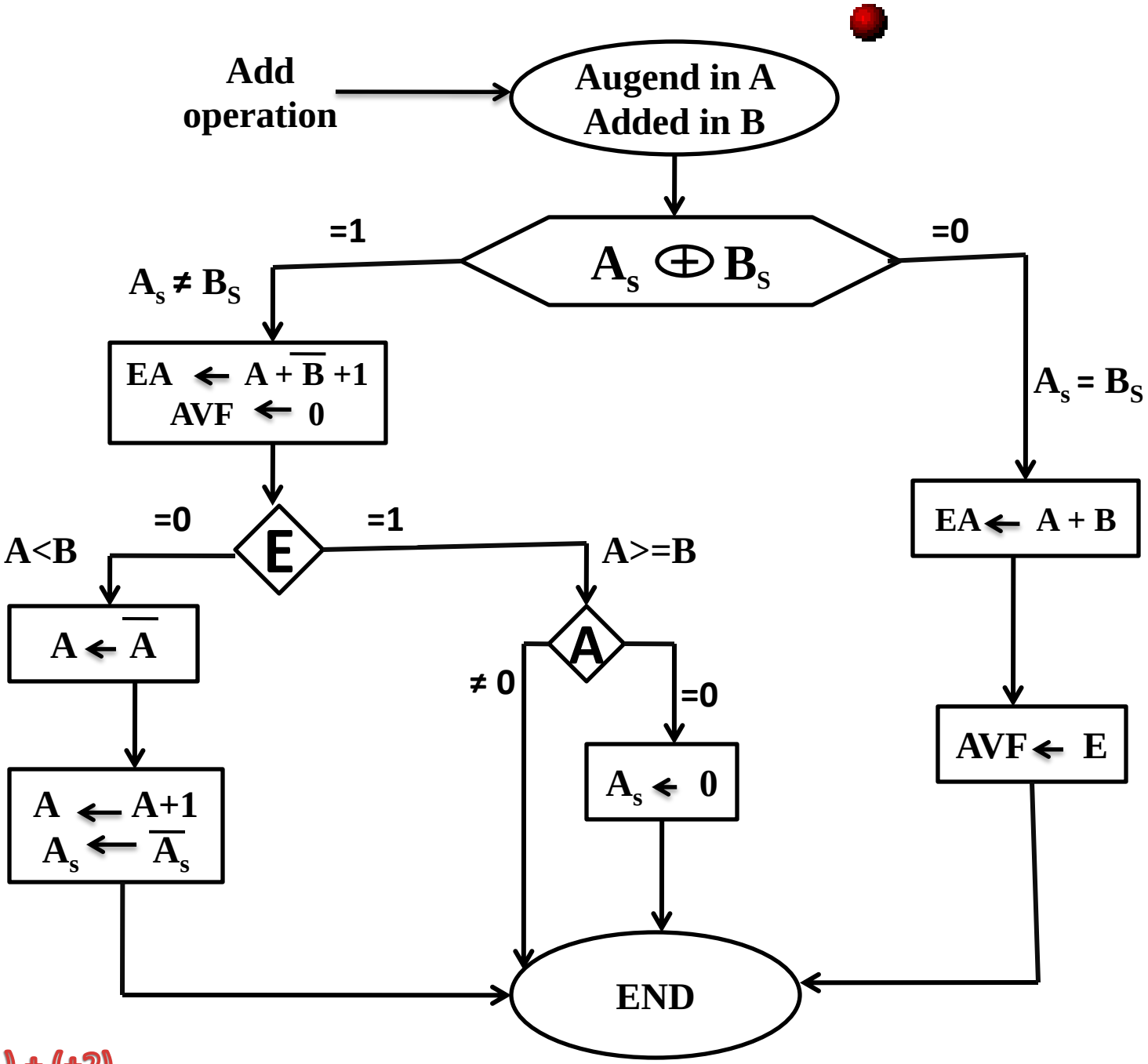
- (+1) + (+2)

  (+A) + (+B)


Add operation
Augend in A
Added in B
$A_s \oplus B_s$
=1
=0
$A_s \neq B_S$
$A_s = B_S$
EA ← A + $\overline{B}$ +1
AVF ← 0
EA ← A + B
=0
=1
E
A<B
A>=B
A ← $\overline{A}$
A
≠ 0
=0
AVF ← E
A ← A+1
A_s ← $\overline{A}_s$
A_s ← 0
END
(+1) + (+2)

- (-1) + (+2)

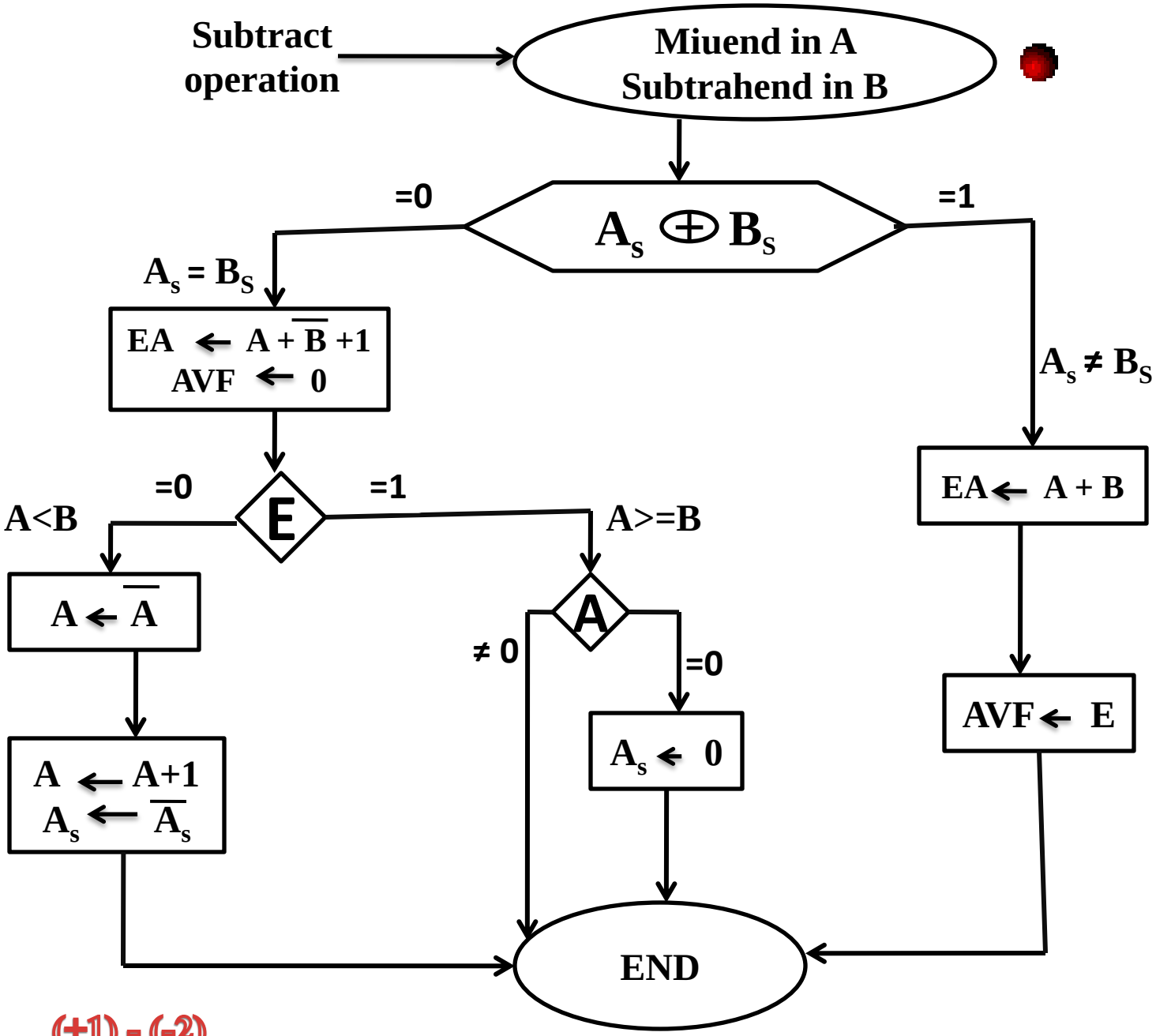
  (-A) + (+B)


Add operation
Augend in A
Added in B
$A_s \oplus B_S$
=1
=0
$A_s \neq B_S$
$A_s = B_S$
EA ← A + $\overline{B}$ +1
AVF ← 0
EA ← A + B
E
=0
=1
A<B
A>=B
A ← $\overline{A}$
A
≠ 0
=0
AVF ← E
A ← A+1
A_s ← $\overline{A}_s$
A_s ← 0
END
(-1) + (+2)

- For Example of **Subtraction**

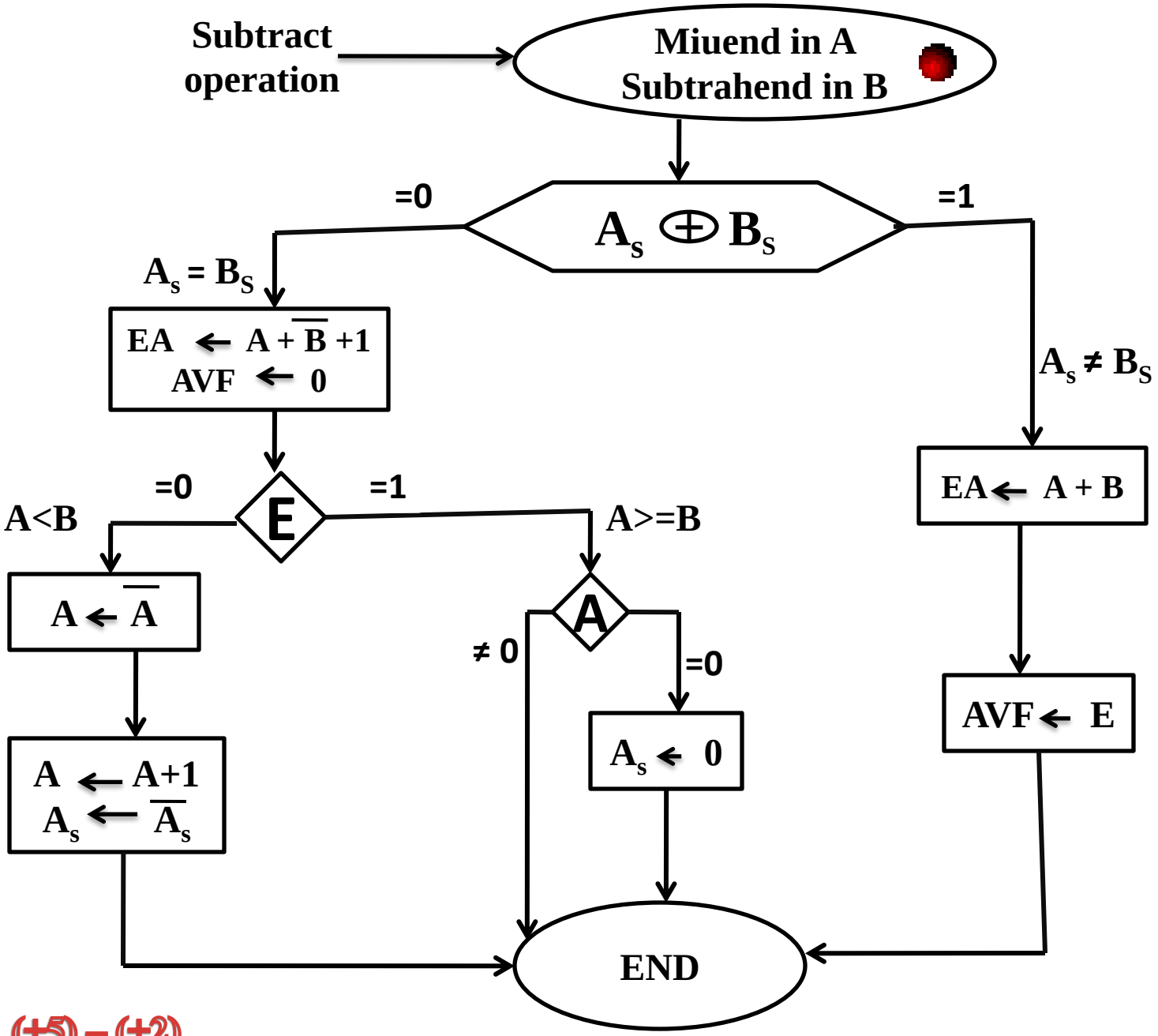
- (+1) - (-2)

  (+A) - (-B)


Subtract operation
Miuend in A
Subtrahend in B
$A_s \oplus B_S$
=0
=1
$A_s = B_S$
$A_s \neq B_S$
EA ← A + $\overline{B}$ +1
AVF ← 0
E
=0
=1
A<B
A>=B
EA ← A + B
A ← $\overline{A}$
A
≠ 0
=0
AVF ← E
A ← A+1
A_s ← $\overline{A_s}$
A_s ← 0
END
(+1) - (-2)

- $(+5) - (+2)$

  $(+A) - (+B)$


Subtract operation
Miuend in A
Subtrahend in B
$A_s \oplus B_S$
=0
=1
$A_s = B_S$
$A_s \neq B_S$
EA ← A + $\overline{B}$ +1
AVF ← 0
E
=0
=1
A<B
A>=B
EA ← A + B
A ← $\overline{A}$
A
≠ 0
=0
AVF ← E
A ← A+1
A_s ← $\overline{A_s}$
A_s ← 0
END

(+5) – (+2)

Addition and Subtraction with Signed 2's Complement Data

Presented by

A.Srinivasan, Associate Professor.

Addition and Subtraction with Signed 2's Complement Data

- When two numbers of n digits are added, the sum occupies n+1 digits, then the overflow is occurred.
- The overflow can be detected by inspecting the last two carries out of addition.
- When two carries are applied to an XOR gate the overflow is detected when the o/p of the gate is equal to 1.

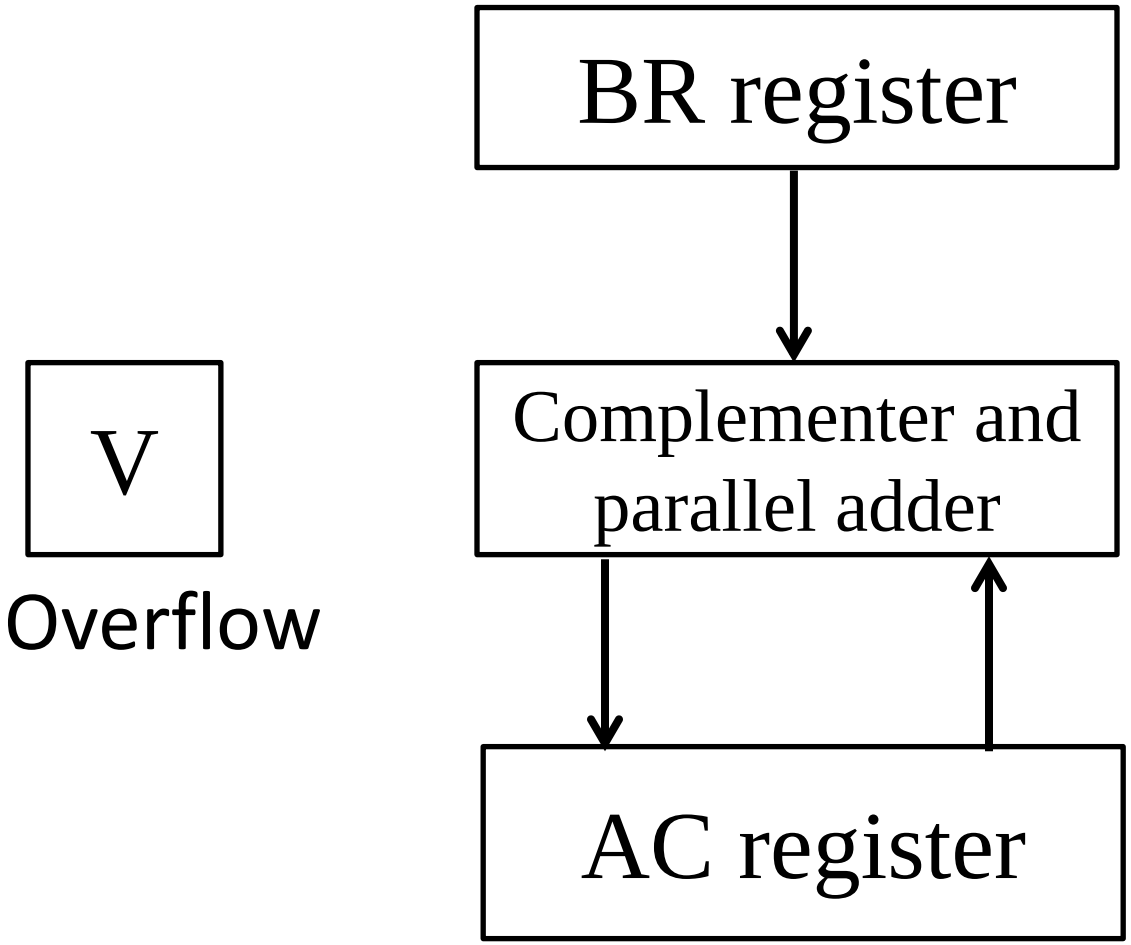


Figure: Hardware for signed-2's complement addition and subtraction.

Addition and Subtraction with Signed 2's Complement Data

- The leftmost bit in AC and BR represents the sign bits of the number.
- The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder.
- The overflow flipflop V is set to 1 if there is an overflow.
- The algorithm for adding and subtracting numbers using signed 2's complement representation is shown in the below flowchart.

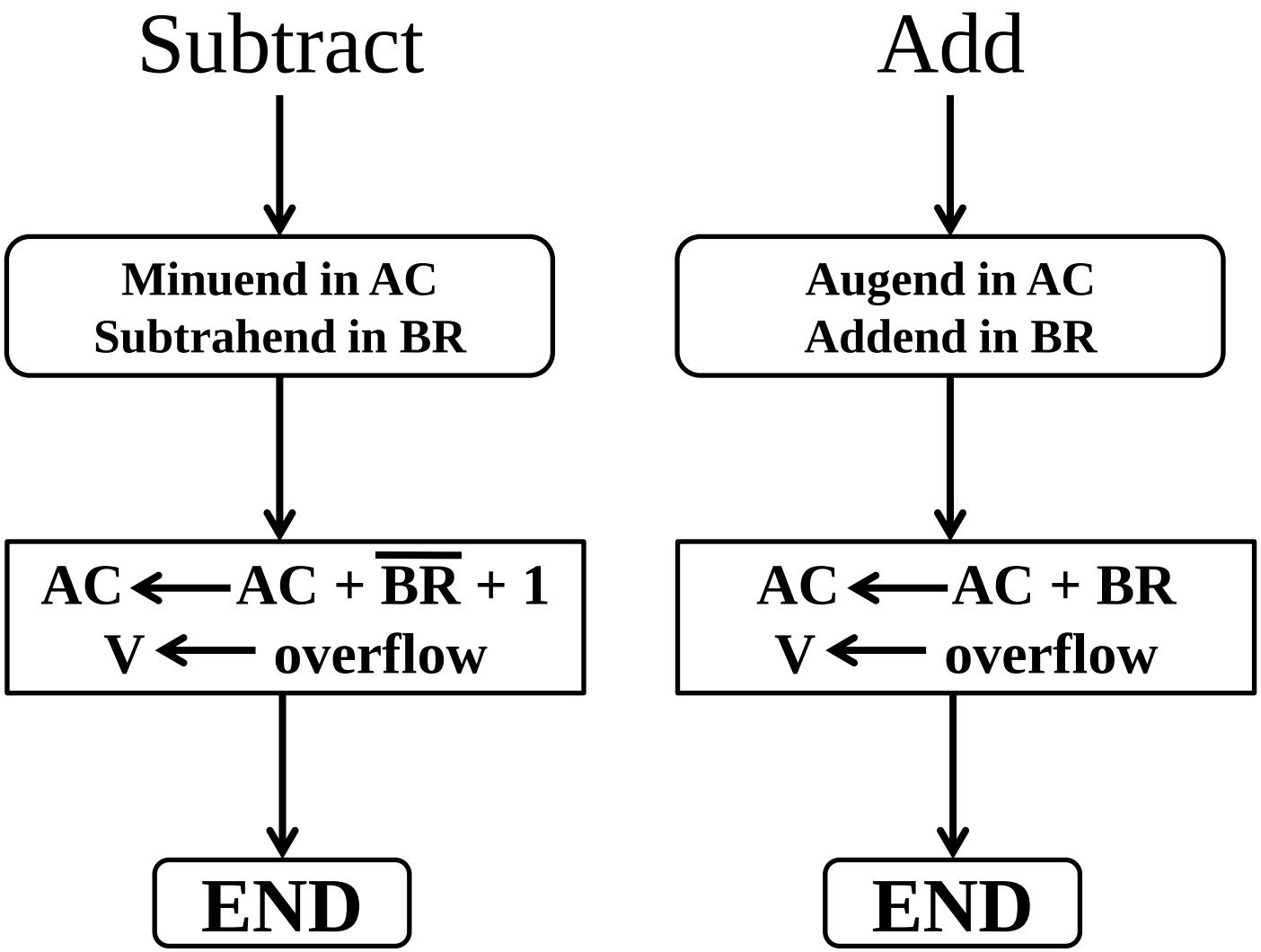


Figure: Algorithm for adding and subtracting numbers in signed-2's complement representation.

Addition and Subtraction with Signed 2's Complement Data

- The sum is obtained by adding the content of AC and BR.
- The overflow bit is set to 1 if the XOR of last two carries is set to 1 otherwise to 0.
- The subtraction operation is obtained by adding the content of AC to the 2's complement of BR.
- Comparing this algorithm with signed magnitude, it is simpler to add and subtract numbers if negative numbers are in signed 2's complement form.
- So the most of the computers adopt this representation over signed magnitude form.

Multiplication algorithms

Presented by
A.Srinivasan, Associate Professor

Multiplication algorithms

- Multiplication of two fixed point binary numbers in signed magnitude representation is done with paper and pencil of successive shift and add operation.

```
23        10111     Multiplicand
19      × 10011     Multiplier
        ------
          10111
         10111
        00000    +
       00000
      10111
     --------
437  110110101     Product
```

Multiplication algorithms

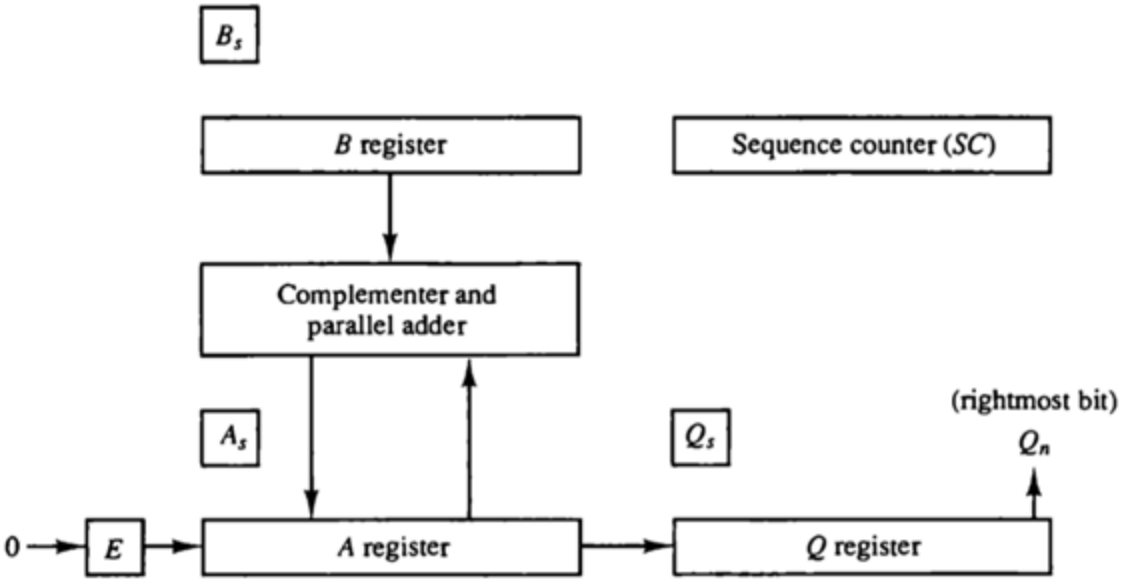
- If the multiplier bit is a 1,the multiplicand is copied down; otherwise zero are copied down.
- Finally all the partial products are added to get the desired product.
- The sign of the product is determined from the sign of the multiplicand and multiplier.
- If they are same sign then product is positive and if they are of different sign, the sign of the product is negative.

Hardware Implementation for Signed-Magnitude data

- First instead of providing register to store and add simultaneously as many binary numbers as there are bits in the multiplier , it is convenient to provide an adder for the summation of only two binary numbers and successively accumulate the partial products in a register.
- Second instead of shifting the multiplicand to the left , the partial product is shifted to the right .
- Third when the corresponding bits of the multiplier is 0 there is no need al add all zero’s to the partial product.

Hardware Implementation for Signed-Magnitude data

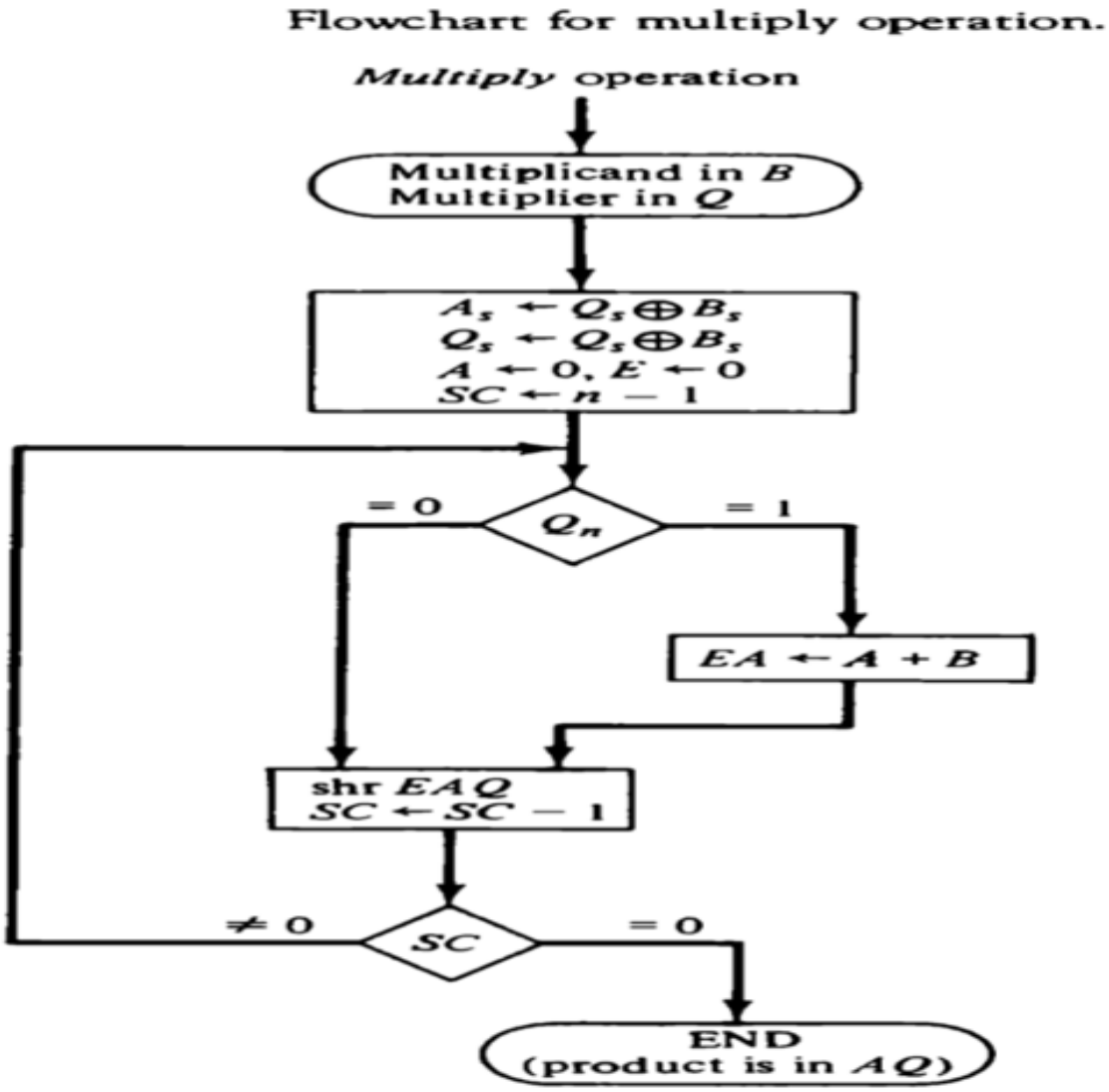
- The hardware for multiplication consists of the equipment shown in following fig.



Hardware Implementation for Signed-Magnitude data

- The multiplier stored in the Q register and its sign in Q_s The sequence counter SC is initially set to a number equal to the number of bits in the multiplier.
- The counter is decremented by 1 after forming each partial product. When the counter reaches to zero, the product is formed and process stops.
- Initially the multiplicand is in B register and multiplier in Q register.
- The sum of A and B forms a partial product which is transferred to the EA register .
- The shift will be denoted by the statement shr EAQ to designate the right shift.
- The least significant bit of A is shifted into the most significant position of Q.

Hardware Algorithm



Hardware Algorithm

- Initially the multiplicand is in B and the multiplier in Q and there corresponding signs are in B_s and Q_s respectively.
- Register A and E are cleared and the sequence counter SC is set to a number equal to the number of bits of the multiplier.
- After the initialization , the low order bit of the multiplier is in Qn is tested. If it is 1,the multiplicand in B is added to the present partial product in A . If it is 0, nothing is done.
- Register EAQ shifted once to the right to form the new partial product.
- The SC is decremented by one and its new value is checked. If it is not zero the process is repeated and it it is zero the process stops.
- The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

Example
Multiply 23 X 19 = 437

TABLE 10-2 Numerical Example for Binary Multiplier

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

Booth’s Multiplication Algorithm

Presented by
A.Srinivasan, Associate Professor

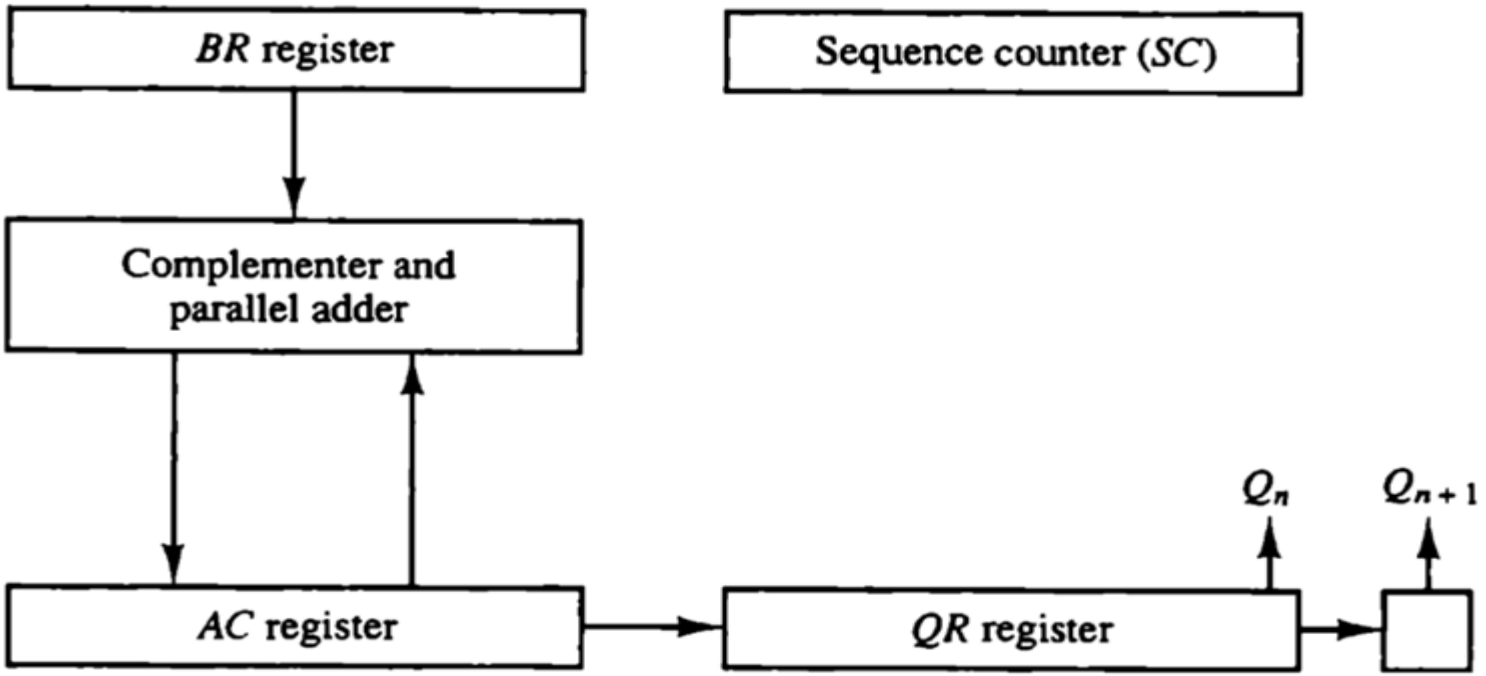
Booth's Multiplication Algorithm

- Booth's Algorithm gives a procedure for multiplying binary integers in signed-2's complement representation.
- It operates on the fact that string 0's in the multiplier requires no addition or subtraction but just shifting and string of 1's in the multiplier require addition or subtraction followed by shifting.

Hardware for booth algorithm

- The algorithm requires the register configuration as shown in fig.

Hardware for Booth algorithm.



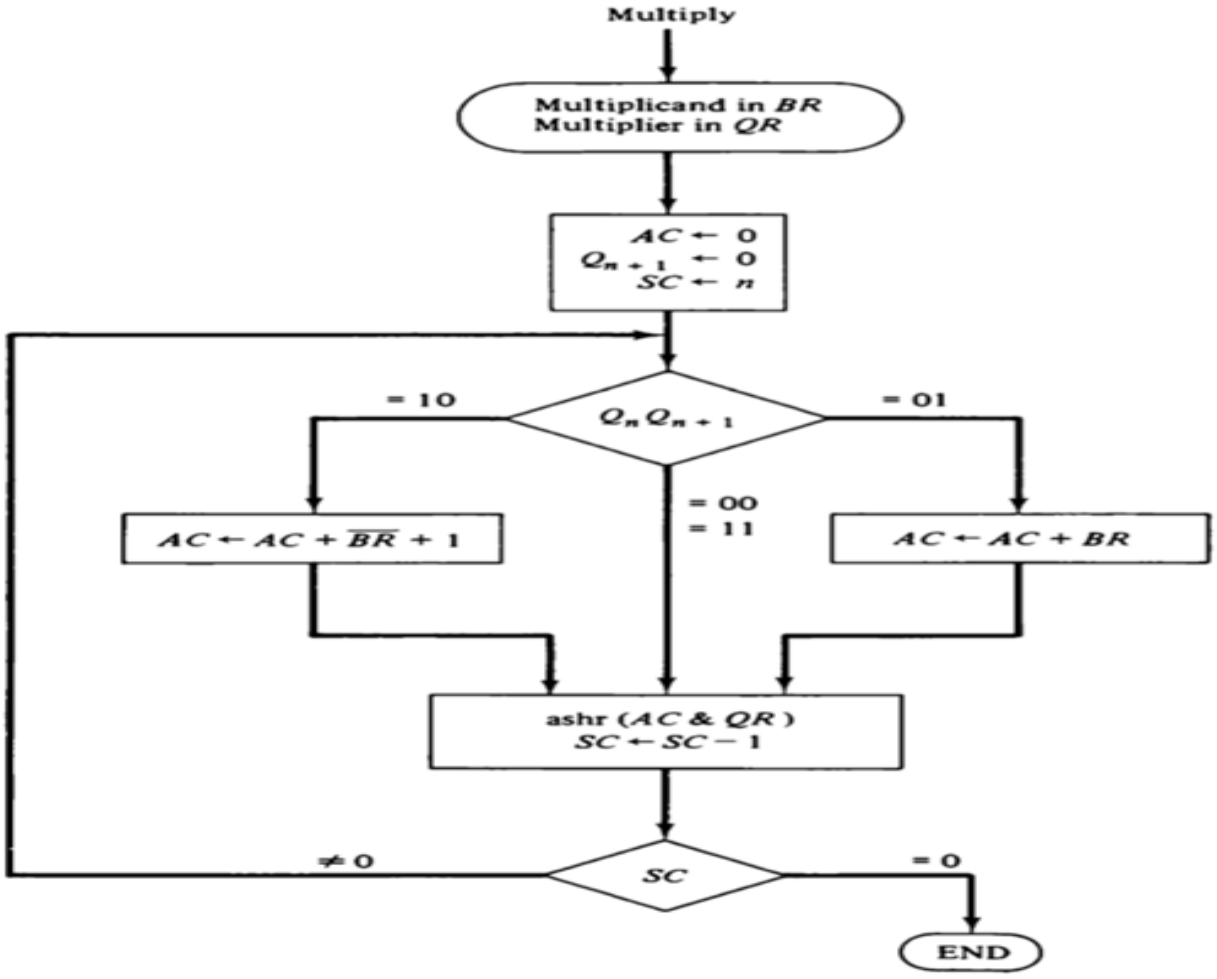
Hardware for booth algorithm

The following rules are required for Booths Algorithm

1. The multiplicand is subtracted from partial product upon encountering first LSB 1 in a multiplier.
2. The multiplicand is added to the partial product upon encountering the first 0(provided that there was a previous 1) in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

Here an extra flip-flop Q_{n+1} is appended to QR to facilitate a double bit inspection of the multiplier.

Booth algorithm for multiplication of signed-2's complement



Booth algorithm for multiplication of signed-2's complement

- Initially AC and Qn+1 bit is cleared to zero and SC is set to number n equal to number of bits in multiplier.
- The two bits of the multiplier in Qn and Qn+1 are inspected.
- If two bits are equal to 01, the multiplicand is added to partial product in AC.
- If two bits are equal to 10, the multiplicand is subtracted from the partial product in AC.

Booth algorithm for multiplication of signed-2's complement

- When two bits are equal, partial product does not change.
- The next step is to do arithmetic shift right the partial product in AC and Multiplier in QR which leaves the sign bit in AC unchanged.
- The SC is decremented by one and the loop computation is repeated for n times.
- Finally the result is available in AC and QR with 2's complement representation for negative numbers.

Example

-9 X -13 = + 117

$Q_n Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	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

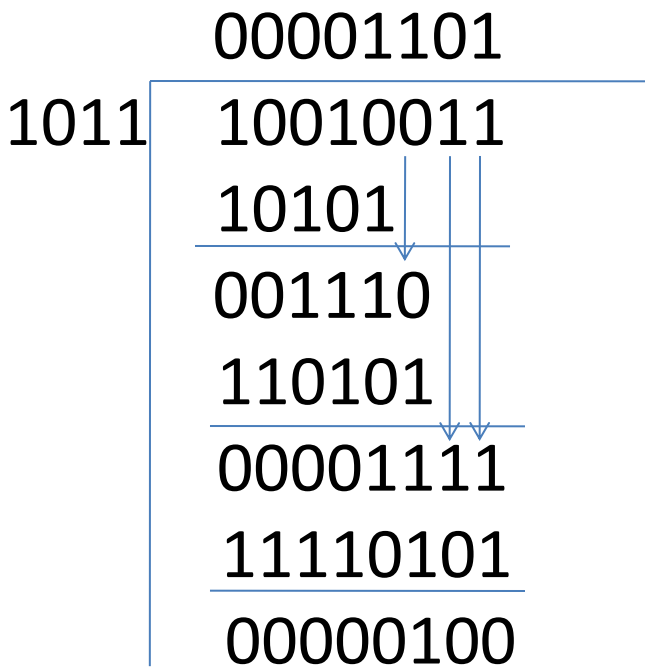
Division Algorithm

Presented by
A.Srinivasan, Associate Professor

Division Algorithm

- Division is somewhat more complex than multiplication. The basis for the algorithm involves repetitive shifting and addition or subtraction operation.
- The long division of unsigned binary integer is shown below.

Division algorithm



Divisor : 01011 = 11, Dividend: 10010011 = 147

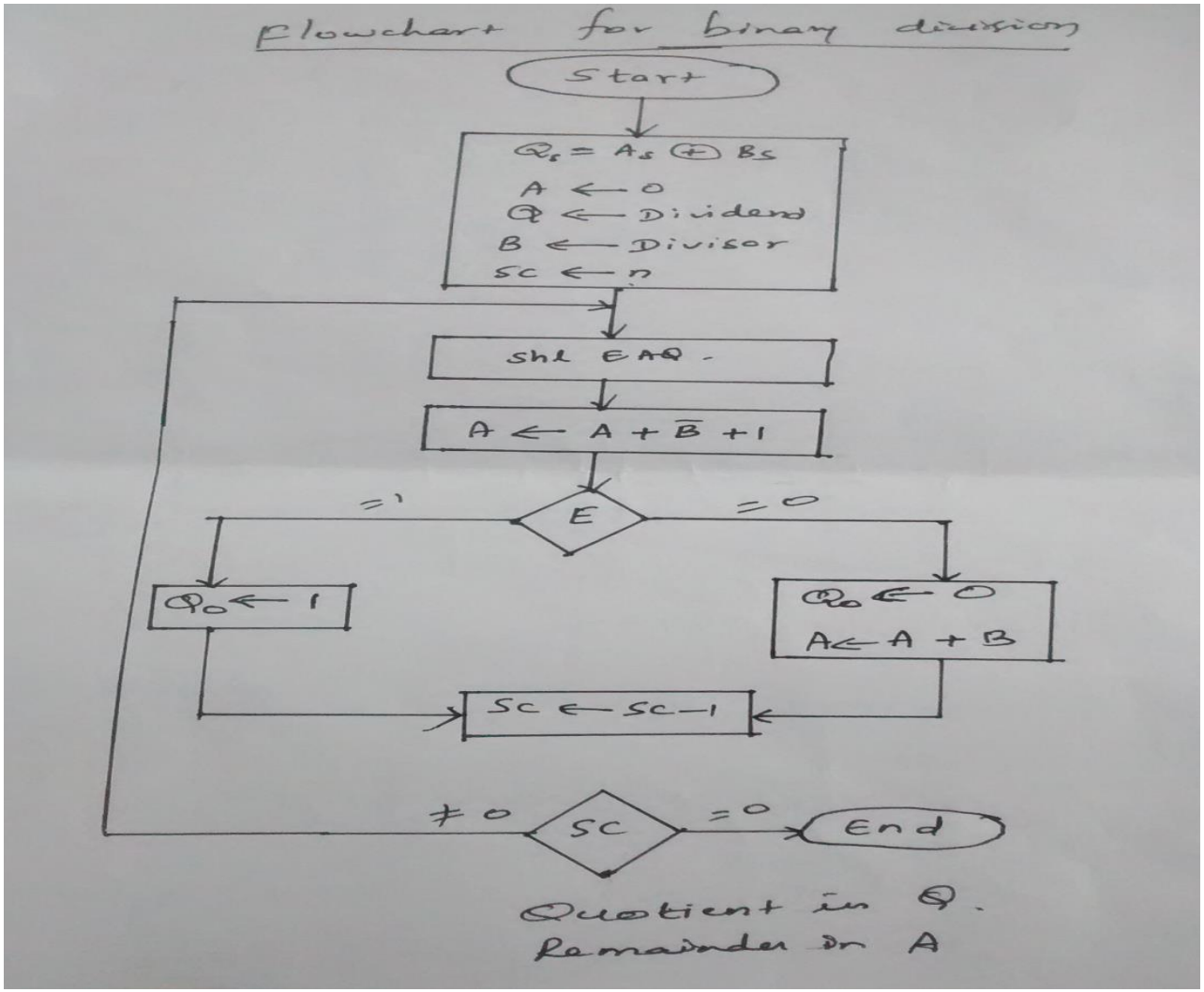
Two's Complement of Divisor: 10101

Quotient: 1101 = 13 Remainder: 100 = 4

Division Algorithm

- First the bits of dividend are examined from left to right, until the set of bits examined represent the number greater than or equal to divisor.
- Until this event occurs 0's are placed in quotient from left to right.
- When the event occurs 1 is placed in the quotient and divisor is subtracted from the partial dividend. The result is referred as partial remainder.
- At each cycle of this process, an additional bits from the dividend are appended to the partial remainder, until the result is greater than or equal to the divisor.
- This process continues until all the bits of the dividend are exhausted.

Division Algorithm



Division Algorithm

- The algorithm can be summarized as follows:

  1. Load the divisor into B register, dividend into Q register, SC with n number of bits in divisor and A and E register with zero's.

  2. Shift EAQ left 1 bit position.

  3. Perform the subtract operation

  $$EA \leftarrow A + B' + 1$$

  4. E value is checked whether it is equal to 0, if it does Q0 bit is set to 0 and B is added back to A.

  $$EA \leftarrow A + B$$

  5. Otherwise Q0 is set to 1

  6. Sequence counter SC is then decremented by 1.

  7. The process continues for n times and finally quotient will be in Q and remainder in A .

Floating Point Arithmetic

Presented by
A.Srinivasan, Associate Professor

Floating Point Arithmetic

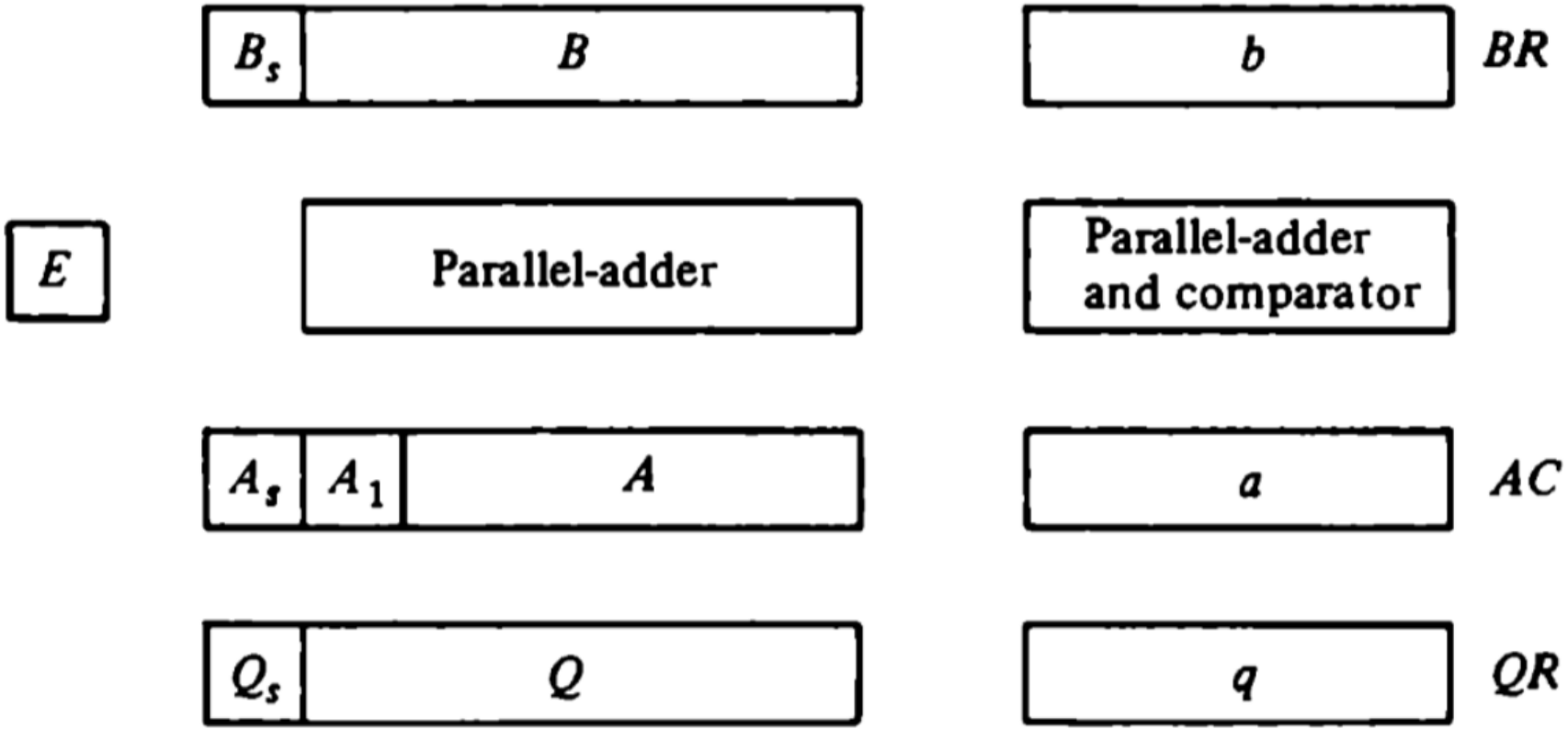
- Floating point number in computer register consists of two parts : a mantissa m and exponent e which will be represented as $m \times r^e$
- A floating point number that has a 0 in the most significant position of the mantissa is said to have an UNDERFLOW.
- To normalize a number that contains an underflow, it is necessary to shift the mantissa to the left and decrement the exponent until a nonzero digit appears in the first position.

Register configuration

- The register configuration for floating point operation is quite similar to the layout for fixed point operation.
- As a general rule, the same register and adder used for fixed point arithmetic are used for processing the mantissas.
- The difference lies in the way the exponents are handled.

Register configuration

Figure 10-14 Registers for floating-point arithmetic operations.



Register configuration

- There are three registers BR, AC and QR.
- Each register is subdivided into two parts.
- The mantissa part has same upper case letters, the exponent part uses the corresponding lower case letters.
- A parallel adder adds the two mantissas and transfers the sum into A and the carry into E.
- A separate parallel adder is used for the exponents. Since the exponents are biased.

Addition and subtraction

- During addition and subtraction , the two floating point operands are in AC and BR. The sum or difference is formed in the AC .
- The algorithm can be divided into four consecutive parts :
  1. Check for zeros.
  2. Align the mantissa.
  3. Add or subtract the mantissa.
  4. Normalize the result.

Addition and subtraction

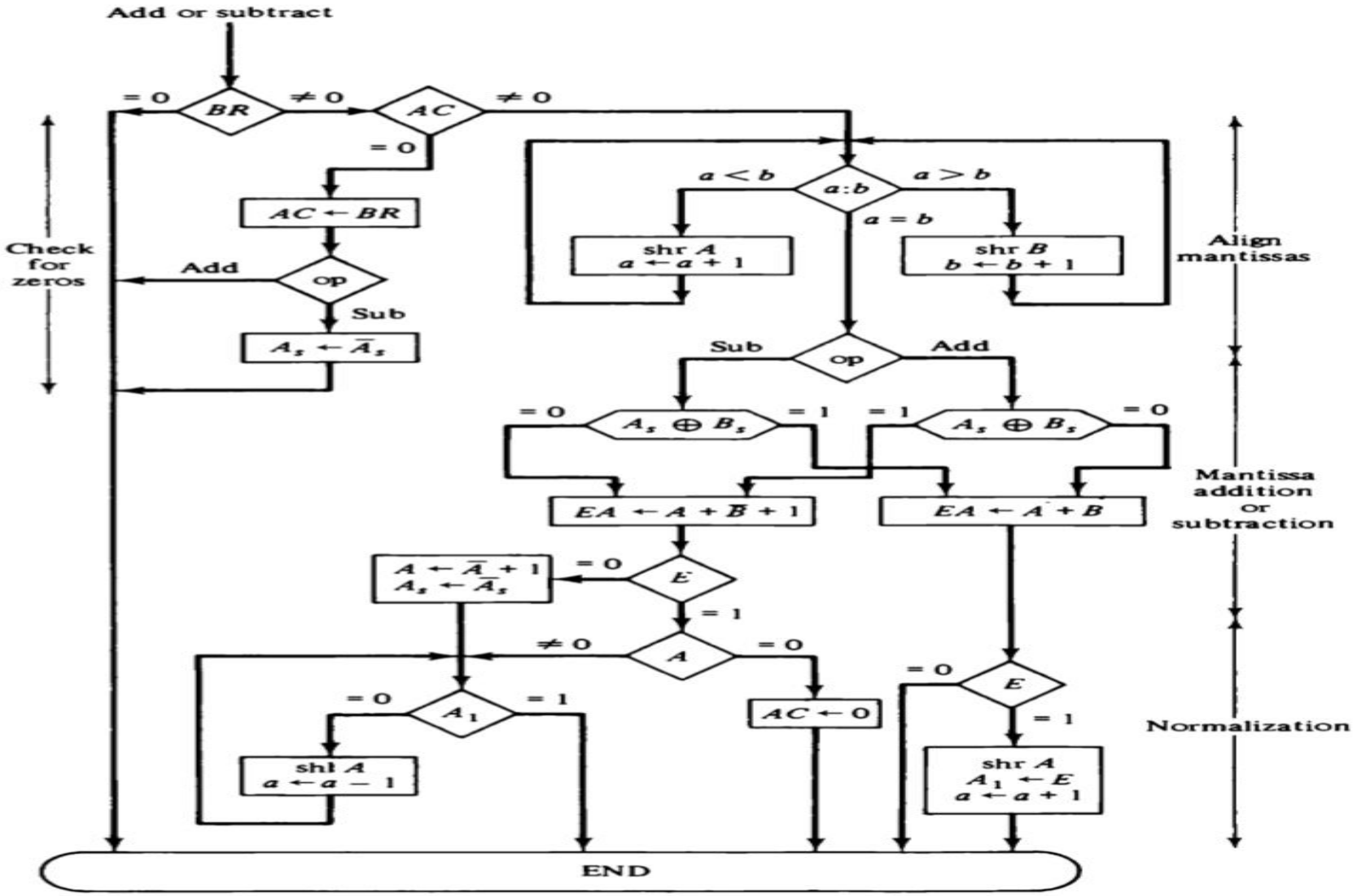
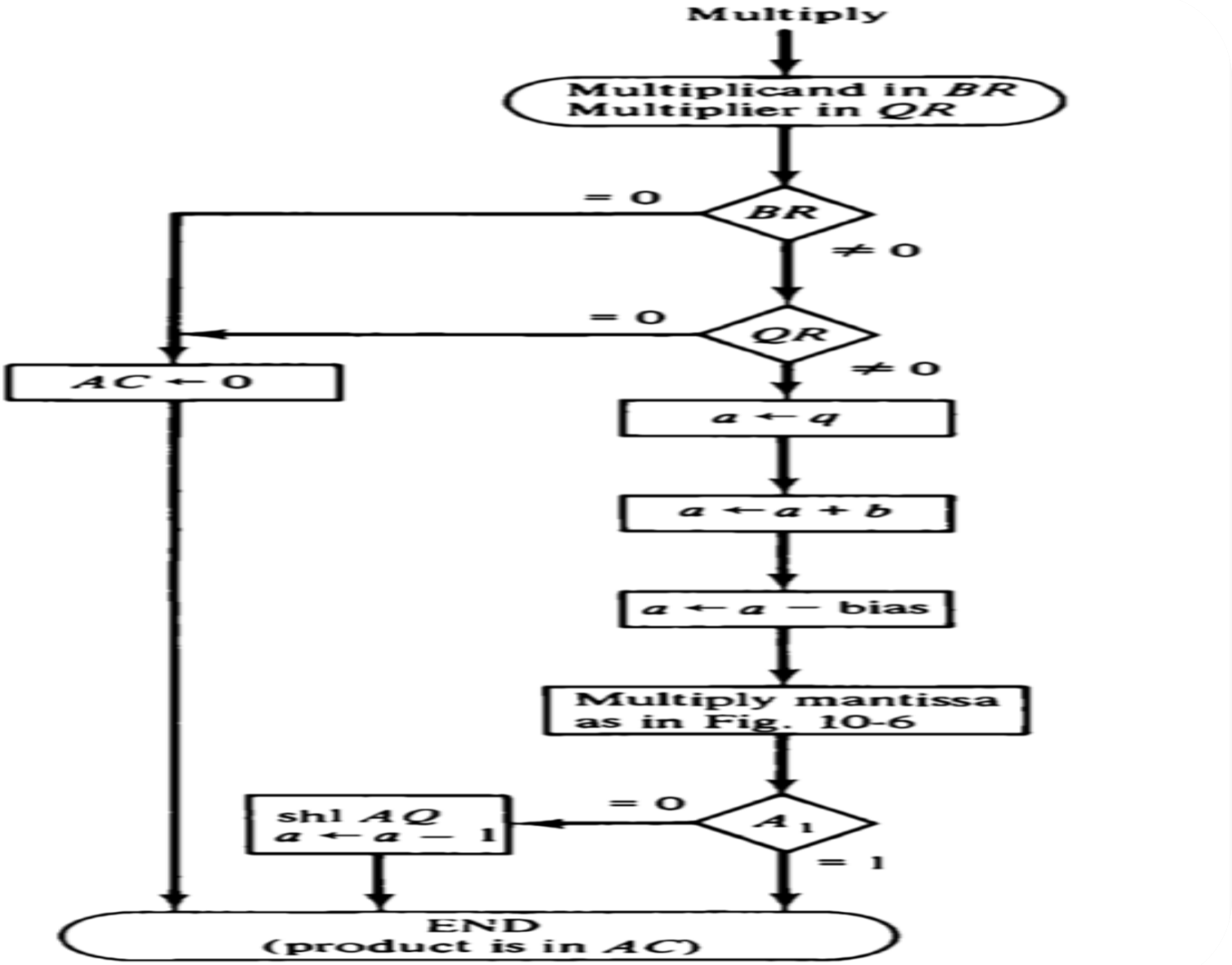


Figure 10-15 Addition and subtraction of floating-point numbers.

Multiplication

- The multiplication of two floating point numbers requires that we multiply the mantissas and add the exponents. No comparison of exponents or alignment of mantissa is necessary.
- The multiplication of the mantissa is performed same as fixed point to provide a double precision product.
- The multiplication algorithm can be subdivided into four parts :-
  1. Check for zeros.
  2. Add the exponents.
  3. Multiply the mantissa.
  4. Normalize the product.

Multiplication



Division

- Floating point division requires that the exponents be subtracted and the mantissa divided.
- The mantissa division is done as in fixed point division.
- The division algorithm can be divided into five parts..
  1. Check for zeros.
  2. Initialize registers and evaluate the sign.
  3. Align the dividend
  4. Subtract the exponents.
  5. Divide the mantissa.

Division

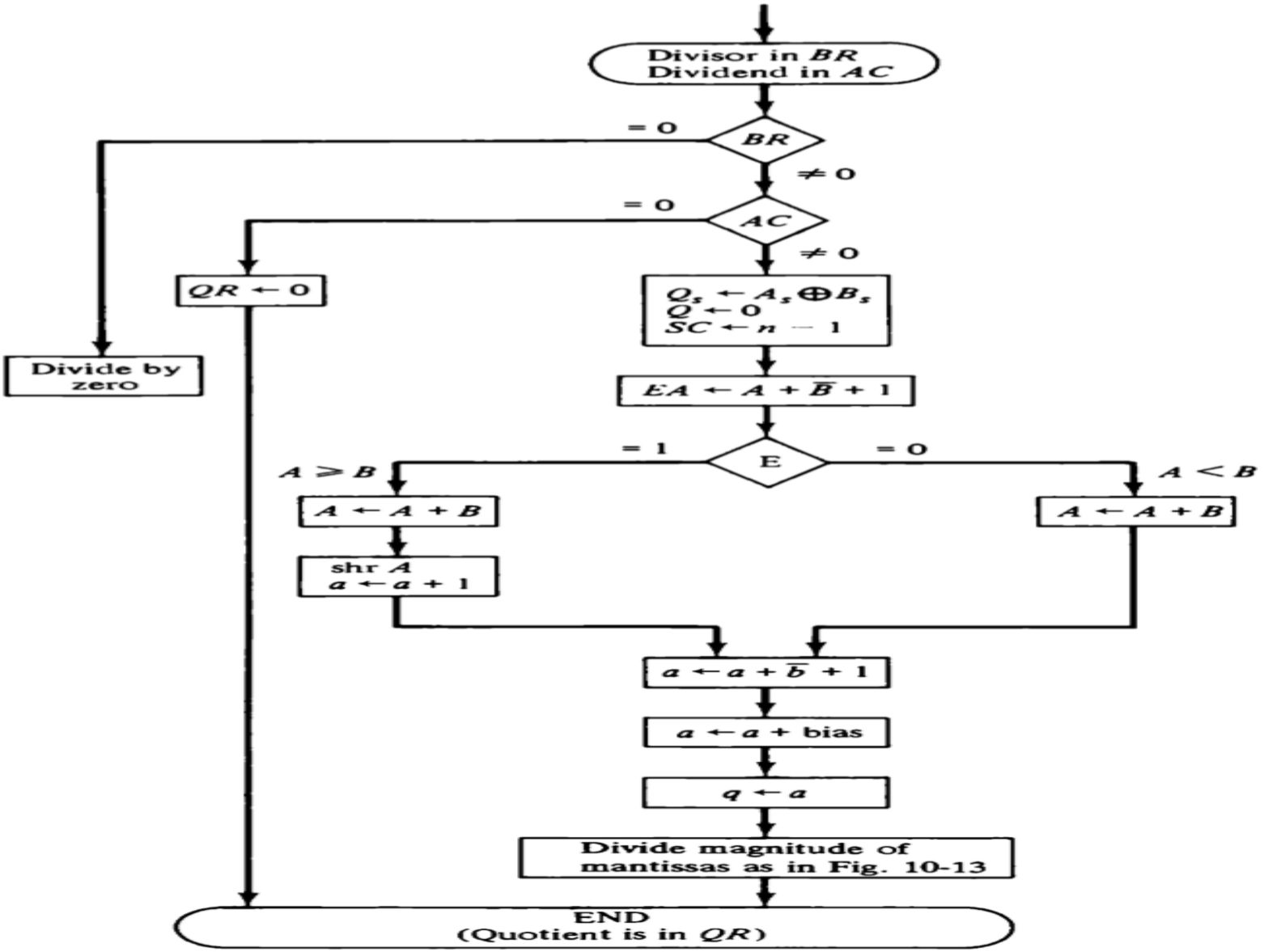


Figure 10-17 Division of floating-point numbers.

INSTRUCTION CODES

Presented by
A.Srinivasan, Assoc Professor

INSTRUCTION CODES

- Every different processor type has its own design (different registers, buses, microoperations, machine instructions, etc)
- It contains
  - Many registers
  - Multiple arithmetic units, for both integer and floating point calculations
  - The ability to pipeline several consecutive instructions to speed execution

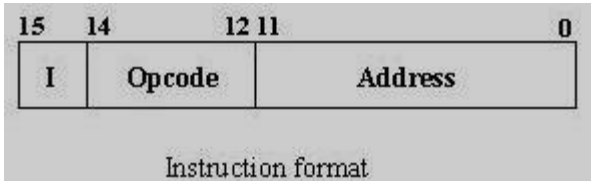
INSTRUCTION CODES

- Program
  - A sequence of (machine) instructions
- (Machine) Instruction
  - A group of bits that tell the computer to *perform a specific operation* (a sequence of micro-operation)
- The instructions of a program, along with any needed data are stored in memory
- The CPU reads the next instruction from memory
- It is placed in an *Instruction Register* (IR)

INSTRUCTION CODES

- Control circuitry in control unit then translates the instruction into the sequence of microoperations necessary to implement it
- A computer instruction is often divided into two parts
  - An *opcode* (Operation Code) that specifies the operation for that instruction
  - An *address* that specifies the registers and/or locations in memory to use for that operation

Stored Program Organization



15 0

operand

Memory 4096 x 16

Instructions (Program)

Operands (Data)

Processor Register (Accumulator or AC)

Stored Program Organization

- The Basic Computer has two components, a processor and memory
- The memory has 4096 words in it
  - 4096 = 2^{12}, so it takes 12 bits to select a word in memory
- Each word is 16 bits long
- In the Basic Computer, since the memory contains 4096 (= 2^{12}) words, we needs 12 bit to specify which memory address.

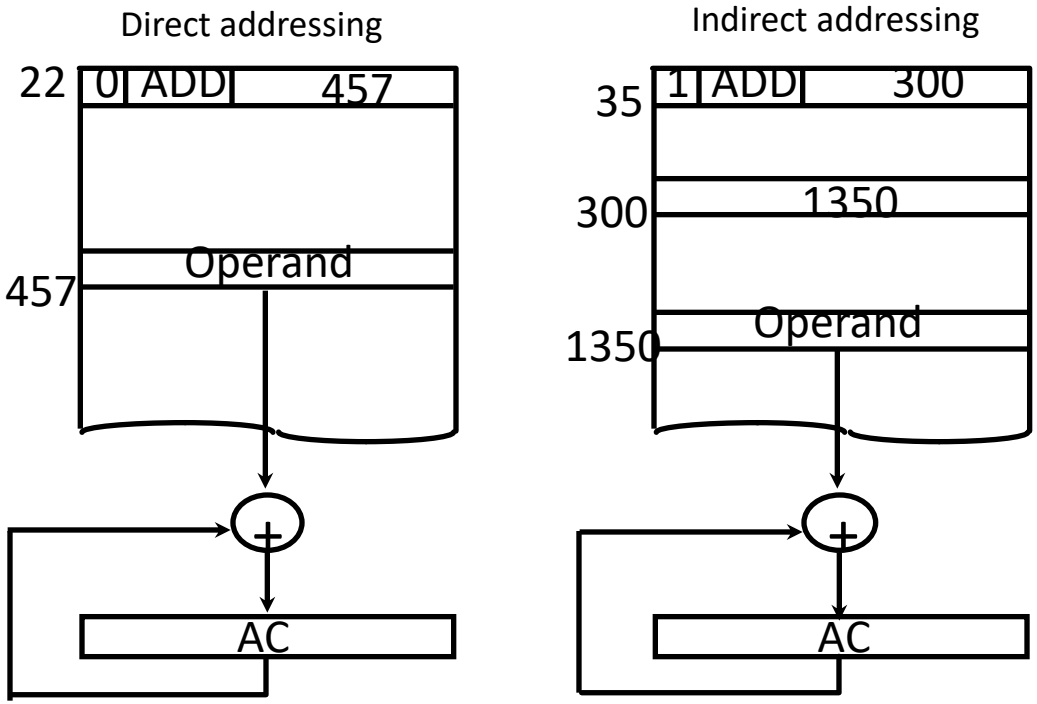
Stored Program Organization

- In the Basic Computer, bit 15 of the instruction specifies the *addressing mode*

  (0: direct addressing, 1: indirect addressing)
- If the operation in an instruction does not need an operand from memory, the rest of the bits in the instruction can be used for other purposes
- Ex: Clear AC, Complement AC, Increment AC.

Addressing Modes

- The address field of an instruction can represent either
  - Direct address
  - Indirect address



Direct Addressing

- The instruction is placed at the location 22 in memory.
- The opcode specifies ADD instruction
- Address part specifies the address of the operand that is 457.
- The operand at the location 457 is added with AC content and the result is stored in AC by overwriting its previous content.

Indirect Addressing

- The instruction is placed at the location 35 in memory.
- The opcode specifies ADD instruction.
- Address part specifies the address 300 and mode bit 1 specifies indirect address.
- The control goes to the address 300 to find the address of the operand.
- In this case the address of the operand is 1350
- The operand at the location 1350 is added with AC content and the result is stored in AC by overwriting its previous content.

Indirect Addressing

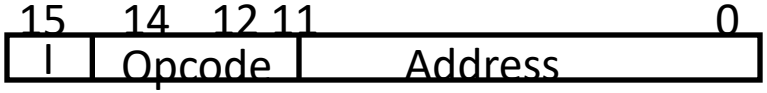
- The indirect address instruction needs two references to memory to fetch an operand.
- First reference to fetch address of operand
- Second reference to fetch the operand itself.

Computer Instructions

BASIC COMPUTER INSTRUCTIONS

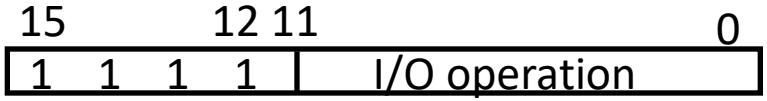
- Basic Computer Instruction Format

Memory-Reference Instructions (OP-code = 000 ~ 110)



Register-Reference Instructions (OP-code = 111, I = 0)

Input-Output Instructions (OP-code =111, I = 1)



BASIC COMPUTER INSTRUCTIONS

Symbol	Hex Code I = 0	Hex Code I = 1	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instr. if AC is positive
SNA	7008		Skip next instr. if AC is negative
SZA	7004		Skip next instr. if AC is zero
SZE	7002		Skip next instr. if E is zero
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

MEMORY-REFERENCE INSTRUCTIONS

MEMORY-REFERENCE INSTRUCTIONS

- The decoded output Di for i = 0,1, 2, 3, 4, 5, and 6 from the operation decoder that belongs to each instruction.
- The effective address of the instruction is in the address register AR and was placed there during timing signal T2 when I = 0, or during timing signal T3 when I = 1.
- The execution of the memory-reference instructions starts with timing signal T4.

MEMORY-REFERENCE INSTRUCTIONS

- The symbolic description of each instruction is specified in the following table in terms of register transfer notation.

Symbol	Operation	Decoder Symbolic description
AND	D0	AC ← AC Λ M[AR]
ADD	D1	AC ← AC + M[AR], E ←Cout
LDA	D2	AC ← M[AR]
STA	D3	M[AR] ← AC
BUN	D4	PC ← AR
BSA	D5	M[AR ← PC, PC ← AR + 1
ISZ	D6	M[AR] ← M[AR] + 1, If M[AR] + 1 = 0 then PC ←PC + 1

AND to AC

- This is an instruction that performs the AND logic operation on pairs of bits in AC and the memory word specified by the effective address.
- The result of the operation is transferred to AC.
- The microoperations that execute this instruction are :

  D0 T4 : DR ← M[AR]

  D0 T5: AC ← AC Λ DR, SC ← 0

ADD to AC

- The instruction adds the content of the memory word specified by the effective address to the value of AC.
- The sum is transferred into AC and the output carry Cout is transferred to the E (extended accumulator) flip-flop.
- The microoperations needed to execute this instruction are

  D1 T4: DR ← M[AR]

  D1T5: AC ← AC + DR, E ← Cout, SC ← 0

LDA : Load to AC

- This instruction transfers the memory word specified by the effective address to AC.
- The microoperations needed to execute this instruction are

  D2 T4: DR ← M[AR]

  D2 T5: AC ← DR, SC ← 0

STA : Store AC

- This instruction stores the content of AC into the memory word specified by the effective address.
- Since the output of AC is applied to the bus and the data input of memory is connected to the bus, we can execute this instruction with one microoperation:

D3 T4: M[AR] ← AC, SC ← 0

BUN : Branch Unconditionally

- This instruction transfers the program to the instruction specified by the effective address.
- This instruction is executed with one microoperation:

  D4 T4: PC ← AR, SC ← 0

BSA : Branch and Save Return Address

- This instruction is useful for branching to a portion of the program called a subroutine.
- When executed, the BSA instruction stores the address of the next instruction in sequence (which is available in PC) into a memory location specified by the effective address.
- The effective address plus one is then transferred to PC to serve as the address of the first instruction in the subordinate.
- Micro operation required for this instruction are:

$$M[AR] \leftarrow PC, PC \leftarrow AR + 1$$

BSA : Branch and Save Return Address

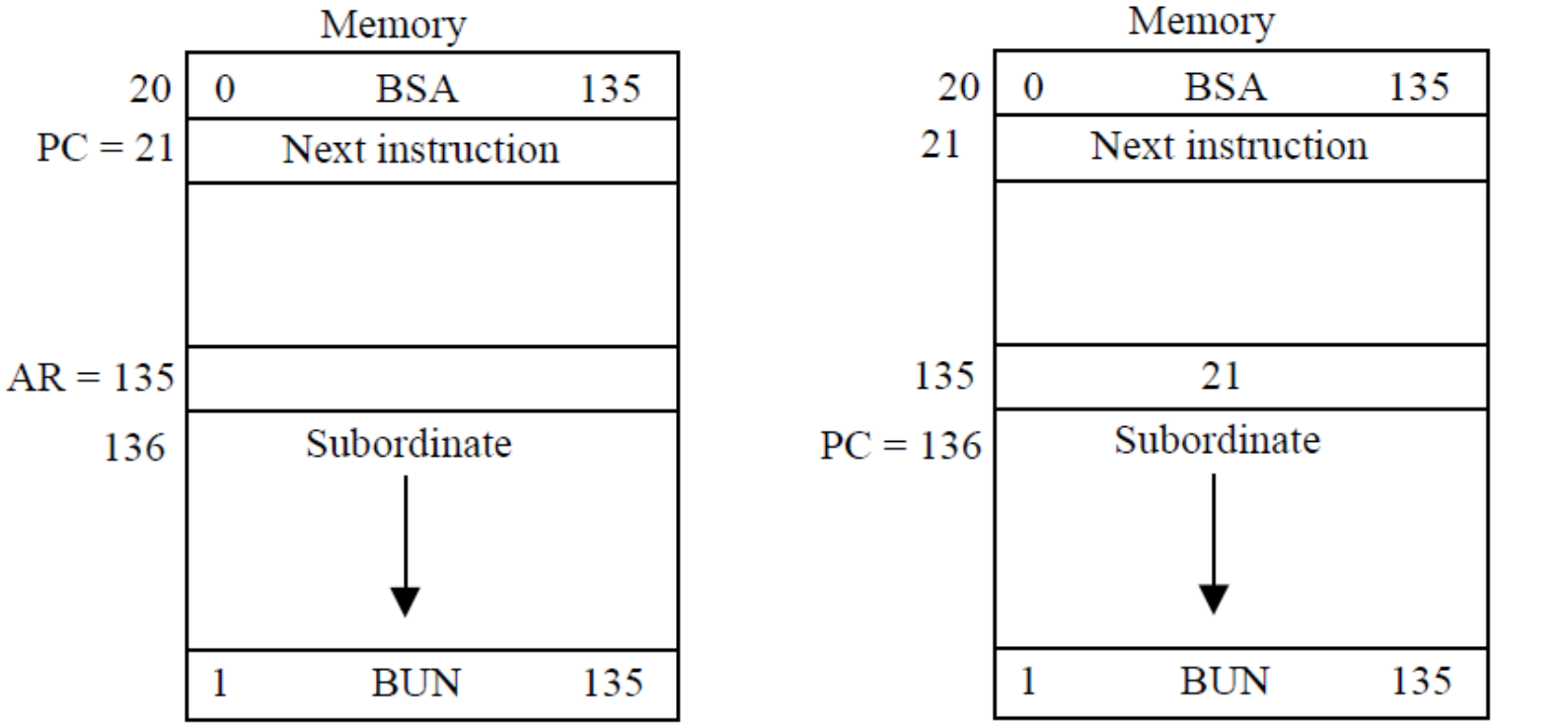


Figure 3.6 Example of BSA instruction execution.

ISZ : Increment and Skip if Zero

- This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1.
- As this negative number is repeatedly incremented by one, it eventually reaches the value of zero. At that time PC is incremented by one in order to skip the next instruction in the program.
- This is done with the following sequence of microoperations:

  D6 T4: DR ← M[AR]

  D6 T5: DR ← DR + 1

  D6 T6 : M[AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ← 0

REGISTER REFERENCE INSTRUCTIONS

REGISTER REFERENCE INSTRUCTIONS

Register Reference Instructions are identified when

- **$D_7 = 1$, $I = 0$**
- **Register Ref. Instr. is specified in B_0 ~ B_{11} of IR**
- **Execution starts with timing signal T_3**

$r = D_7\, I'\, T_3$ => Register Reference Instruction
$B_i = IR(i)$, i=0,1,2,...,11, the ith bit of IR.

	r:	SC ← 0
CLA	rB_{11}:	AC ← 0
CLE	rB_{10}:	E ← 0
CMA	rB_9:	AC ← AC'
CME	rB_8:	E ← E'
CIR	rB_7:	AC ← shr AC, AC(15) ← E, E ← AC(0)
CIL	rB_6:	AC ← shl AC, AC(0) ← E, E ← AC(15)
INC	rB_5:	AC ← AC + 1
SPA	rB_4:	if (AC(15) = 0) then (PC ← PC+1)
SNA	rB_3:	if (AC(15) = 1) then (PC ← PC+1)
SZA	rB_2:	if (AC = 0) then (PC ← PC+1)
SZE	rB_1:	if (E = 0) then (PC ← PC+1)
HLT	rB_0:	S ← 0 (S is a start-stop flip-flop)

BASIC COMPUTER INSTRUCTIONS

Symbol	Hex Code I = 0	Hex Code I = 1	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instr. if AC is positive
SNA	7008		Skip next instr. if AC is negative
SZA	7004		Skip next instr. if AC is zero
SZE	7002		Skip next instr. if E is zero
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

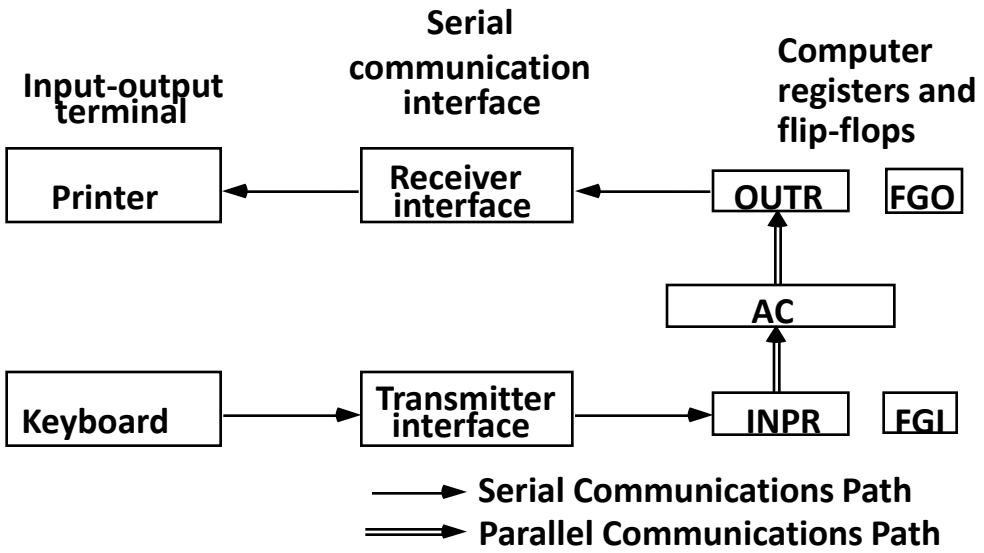
Input-Output and Interrupt

Presented by
A.Srinivasan, Assoc Professor

Input-Output Configuration

- **A Terminal with a keyboard and a Printer**

INPR	Input register - 8 bits
OUTR	Output register - 8 bits
FGI	Input flag - 1 bit
FGO	Output flag - 1 bit
IEN	Interrupt enable - 1 bit



Input-Output Configuration

- Input and output terminal sends and receives serial information's
- Each quantity of information has 8 bits of alphanumeric code.
- The serial information from the keyboard is shifted into I/P register INPR.
- The serial information for the printer is stored in the O/P register OUTR.
- These two registers communicate with a communication interface serially and with AC in parallel
- Transmitter interface receives serial information from keyboard and transmit to INPR.
- Receiver interface receives information from OUTR and sends to printer serially.

Input Register

- Input register INPR consist of 8 bits and holds alphanumeric input information.
- The 1 bit input flag FGI is a control flip-flop which is set to 1 when new information is available in INPR and it is cleared to 0 when the information is accepted by the computer.
- The process of information transfer is as follows:

  1. Initially the input flag FGI is cleared to 0.
  2. When a key is pressed in the keyboard, an 8 bit alphanumeric code is shifted to INPR and the FGI is set to 1.
  3. The computer checks the flag bit, if it is 1 the information from INPR is transferred in parallel into AC and FGI is cleared to 0.
  4. Once the FGI is cleared, new information can be shifted into INPR by striking another key.

Output Register

- The output register OUTR works similarly but direction of information flow is reversed.
- The process of information transfer is as follows:
  1. Initially the output flag FGO is set to 1.
  2. The computer checks the flag bit, if it is 1 the information from AC is transferred in parallel into OUTR and FGO is cleared to 0.
  3. The output device accepts the coded information , prints the corresponding character and when the operation is completed , it sets FGO to 1
  4. The computer does not load a new character into OUTR when FGO is 0 because this indicates that output device is in the process of printing the character.

INPUT-OUTPUT INSTRUCTIONS

- I/p and O/p instruction have the opcode 1111 and recognized by the control when D7=1 and I=1.
- The remaining bits of instruction specify the particular operation.
- The control function and micro operation for input output instruction are listed below

INPUT-OUTPUT INSTRUCTIONS

$D_7IT_3 = p$
$IR(i) = B_i$, i = 6, …, 11

p:		SC ← 0	Clear SC
INP	pB_{11}:	AC(0-7) ← INPR, FGI ← 0	Input char. to AC
OUT	pB_{10}:	OUTR ← AC(0-7), FGO ← 0	Output char from AC
SKI	pB_9:	if(FGI = 1) then (PC ← PC + 1)	Skip on input flag
SKO	pB_8:	if(FGO = 1) then (PC ← PC + 1)	Skip on output flag
ION	pB_7:	IEN ← 1	Interrupt enable on
IOF	pB_6:	IEN ← 0	Interrupt enable off

BASIC COMPUTER INSTRUCTIONS

Symbol	Hex Code I = 0	Hex Code I = 1	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instr. if AC is positive
SNA	7008		Skip next instr. if AC is negative
SZA	7004		Skip next instr. if AC is zero
SZE	7002		Skip next instr. if E is zero
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

INSTRUCTION CYCLE

PRESENTED BY
A.SRINIVASAN, ASSOC PROFESSOR

Instruction Cycle

- Computer Program consist of sequence of instruction residing in memory
- The program is executed by going through a cycle for each instruction.
- Each instruction cycle is subdivided into sequence of sub cycles as follows:
    1. Fetch an instruction from memory
    2. Decode the instruction
    3. Read the effective address from memory if the instruction has an indirect address
    4. Execute the instruction
- The above 4 step process continues until a HALT instruction is encountered

Fetch and Decode Phase

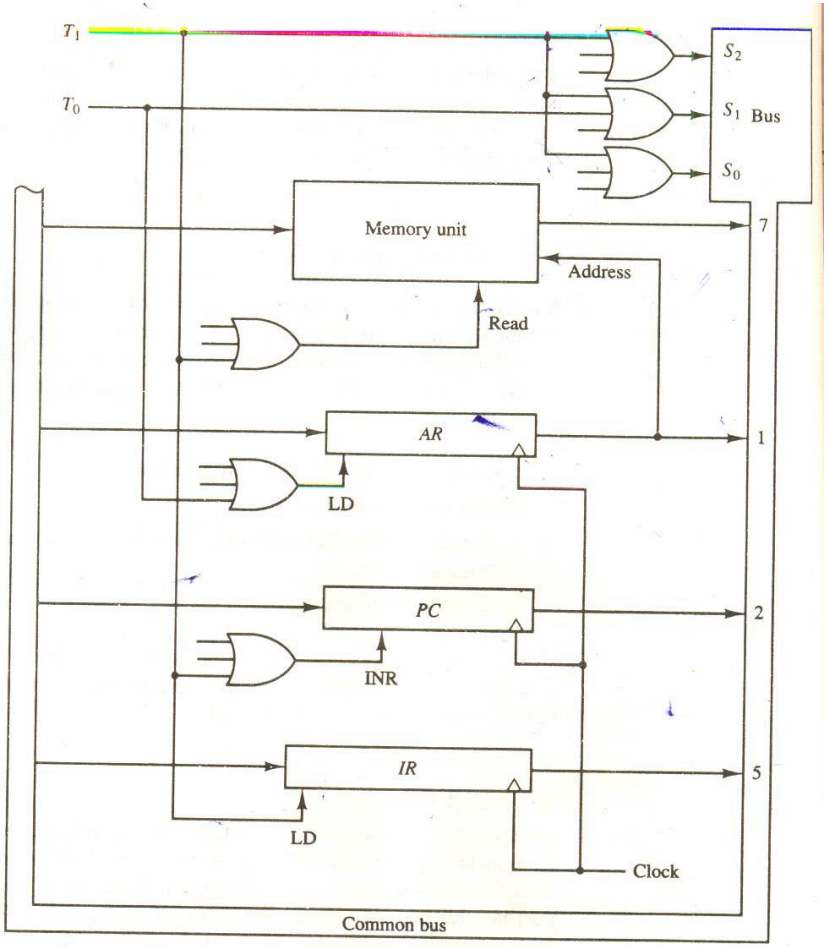
- Program counter(PC) is loaded with the address of first instruction initially.
- The sequence counter(SC) is cleared to zero, providing the timing signal T0.
- After each clock pulse, SC is incremented by 1, so timing goes through T1,T2,…..
- The Microoperation for fetch and decode phase are given below:

T0: AR←PC

T1: IR ←M[AR], PC ←PC + 1

T2: D0, ..., D7 ←Decode IR(12-14), AR ←IR(0-11),
I ←IR(15)

Register transfers for the fetch phase



Fetch and Decode Phase

T0: AR←PC

- Since only AR is connected to the address inputs of memory, the address of instruction is transferred from PC to AR.
- Place the content of PC onto the bus by making the bus selection inputs S2S1S0 = 010.
- Transfer the content of the bus to AR by enabling the LD input

Fetch and Decode Phase

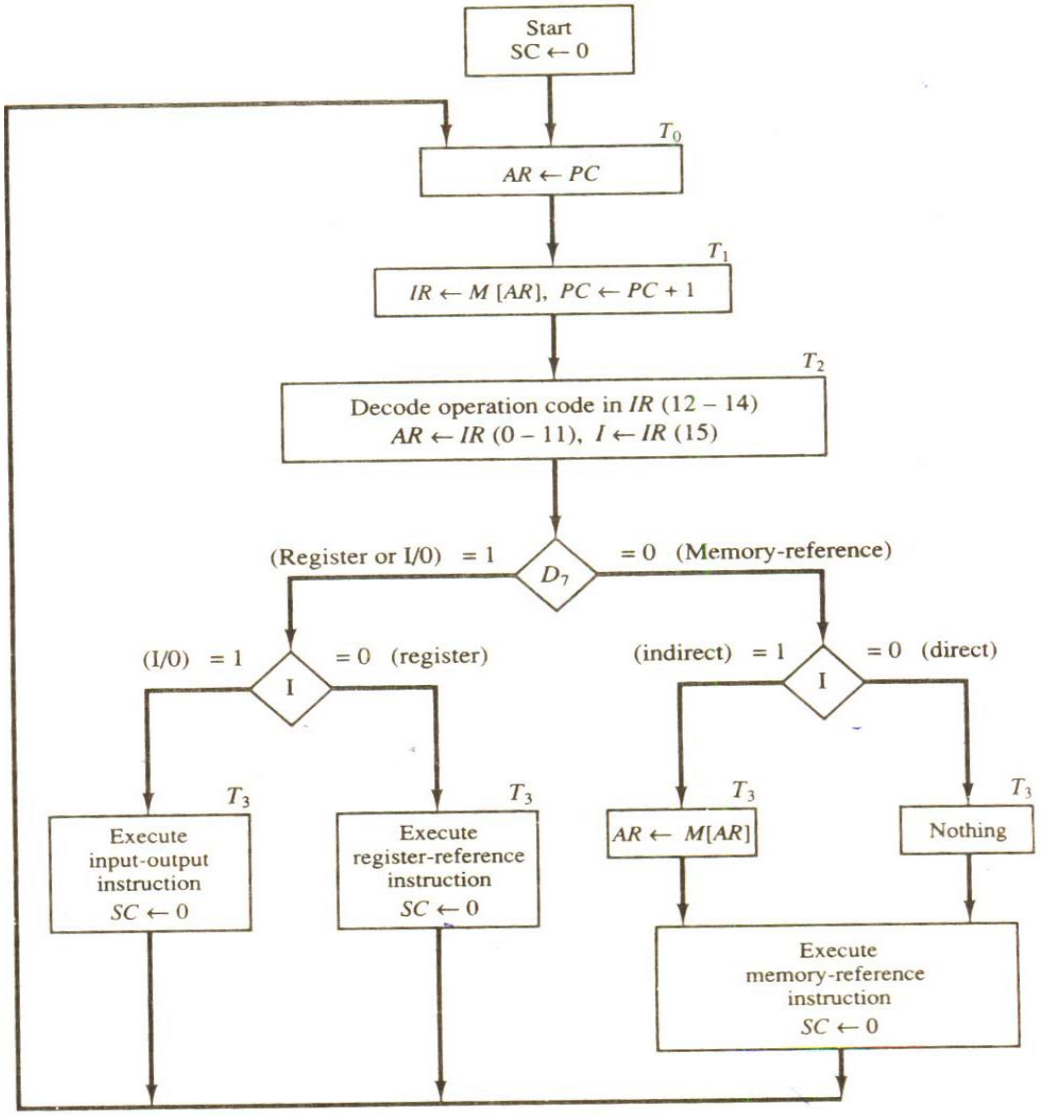
T1: IR ←M[AR], PC ←PC + 1

1. Enable the read input of memory.
2. Place the content of memory onto the bus by making S2S1S0 = 111.
3. Transfer the content of the bus to IR by enabling the LD input of IR.
4. Increment PC by enabling the INR input of PC.

DETERMINE THE TYPE OF INSTRUCTION

- The timing signal that is active after the decoding is T3.
- During time T3, the control unit determines the type of instruction that was just read from memory.
- The following flowchart presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding.

DETERMINE THE TYPE OF INSTRUCTION



DETERMINE THE TYPE OF INSTRUCTION

- Decoder output D7 is equal to 1 if the operation code is equal to binary 111.
- We determine that if D7 = 1, the instruction must be a register-reference or input-output type.
- If D7 = 0, the operation code must be one of the other seven values 000 through 110, specifying memory reference instruction.
- Control then inspects the value of the first bit of the instruction, which is now available in flip-flop I.
- If D7 = 0 and I = 1, we have a memory-reference instruction with an indirect address.
- It is then necessary to read the effective address from memory. The micro operation for the indirect address condition can be symbolized by the register transfer statement

$$AR \leftarrow M[AR]$$

DETERMINE THE TYPE OF INSTRUCTION

- Initially, AR holds the address part of the instruction. This address is used during the memory read operation.
- The word at the address given by AR is read from memory and placed on the common bus.
- The LD input of AR is then enabled to receive the effective address of the operand

DETERMINE THE TYPE OF INSTRUCTION

- The three instruction types are subdivided into four separate paths.
- The selected operation is activated with the clock transition associated with timing signal T3.
- This can be symbolized as follows:

  D7 = 0, I = 1 --- D7’ IT3: AR← M [AR]

  D7 = 0, I = 0 --- D7’ I’T3: Nothing

  D7 = 1, I = 0 --- D7 I’ T3: Execute a register-reference instruction

  D7 = 1, I = 1 --- D7 IT3: Execute an input-output instruction

REGISTER TRANSFER LANGUAGE AND DESIGN OF CONTROL UNIT

Prepared by

A.Srinivasan,

Assoc Professor

UNIT-III
REGISTER TRANSFER LANGUAGE AND DESIGN OF CONTROL UNIT

- **Register Transfer:** Register Transfer Language - Register Transfer - Bus and Memory Transfers - Arithmetic Micro operations - Logic Micro operations - Shift Micro Operations.
- **Control Unit:** Control Memory - Address Sequencing – Micro program Example - Design of Control Unit.

Register Transfer Language

- Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic.
- The modules are interconnected with common data and control paths to form a digital computer system.
- The operations executed on data stored in registers are called *microoperations*
- A microoperation is an elementary operation performed on the information stored in one or more registers.
- Examples are shift, clear and load.

Register Transfer Language

- The internal hardware organization of a digital computer is best defined by specifying
  - The **set of registers** it contains and their functions.
  - The **sequence of microoperations** performed on the binary information stored.
  - The **control that initiates** the sequence of microoperations.

Register Transfer Language

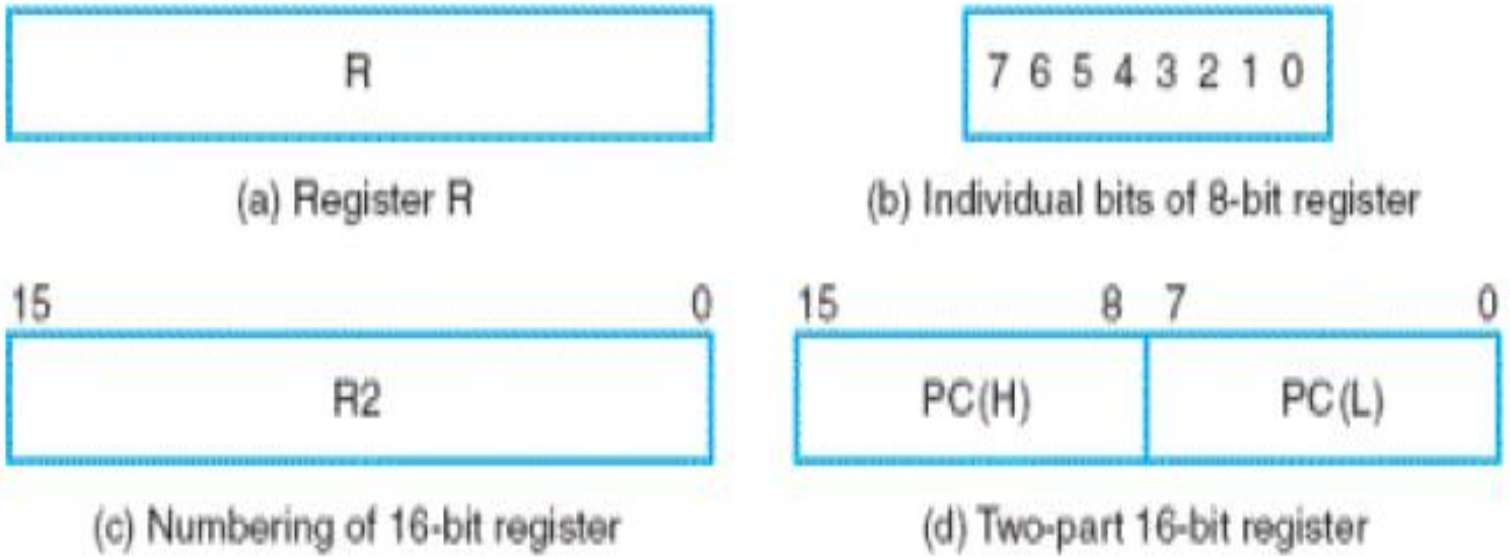
- The symbolic notation used to describe the micro operation transfers among register is called a register transfer language.
- A programming language is a procedure for writing symbols to specify a given computational process.
- A register transfer language is a system for expressing in symbolic form the micro operation sequences among the register of a digital module

Register Transfer

- Computer Registers are designated by capital letters to denote its function.
- The register that holds an address for the memory unit is called as MAR.
- The program counter register that holds the address of next instruction is called as PC .
- IR is the instruction register and R1 is a processor register.

Register Transfer

REPRESENTATION OF REGISTER



(a) Register R

(b) Individual bits of 8-bit register

(c) Numbering of 16-bit register

(d) Two-part 16-bit register

Register Transfer

- The individual flip-flops in an *n*-bit register are numbered from 0 to *n*-1 (from the right position toward the left position)

- The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig(a).

- The individual bits can be distinguished as in Fig(b).

- The numbering of bits in a 16-bit register can be marked on top of the box as shown in Fig(c).

Register Transfer

- A 16-bit register is partitioned into two parts in Fig (d).

- Bits 0 through 7 are assigned the symbol L (for low order byte) and bits 8 through 15 are assigned the symbol H (for high order byte).

- The name of the 16-bit register is PC. The symbol PC(O—7) or PC(L) refers to the low-order byte and PC(8—15) or PC(H) to the high-order byte.

Register Transfer

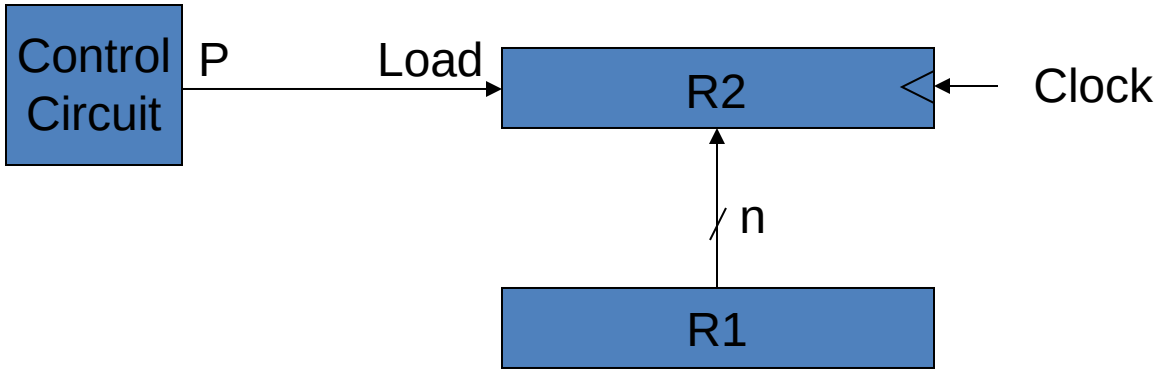
- Information transfer from one register to another is described by a *replacement operator:* **R2 ← R1**
- This statement denotes a transfer of the content of register R1 into register R2
- The content of the R1 (source) does not change
- The content of the R2 (destination) will be lost and replaced by the new data transferred from R1
- If the transfer is to occur only under a predetermined control condition, designate it by

  If (P = 1) *then* (R2 ← R1) or P: R2 ← R1,

  where P is a control function that can be either 0 or 1

Register Transfer

- Hardware implementation of a controlled transfer: P: R2 ← R1



Register Transfer

Basic Symbols for Register Transfers		
Symbol	Description	Examples
Letters & numerals	Denotes a register	MAR, R2
Parenthesis ( )	Denotes a part of a register	R2(0-7), R2(L)
Arrow ←	Denotes transfer of information	R2 ← R1
Comma ,	Separates two microoperations	R2 ← R1, R1 ← R2

Bus and Memory Transfers

- Paths must be provided to transfer information from one register to another.

- A *Common Bus System* is a scheme for transferring information between registers in a multiple-register configuration.

- A bus is a set of common lines, one for each bit of a register, through which binary information is transferred one at a time.

- Control signals determines which register is selected by the bus during each particular register transfer

COMMON BUS SYSTEM CONFIGURATION

- Constructing a common bus system is done by

  a. Using multiplexers

  b. Using three state buffers.

Using Multiplexers:

- The multiplexers select the source register whose binary information is then placed on the bus.
- Each register has four bits, numbered 0 through 3.

Using Multiplexers

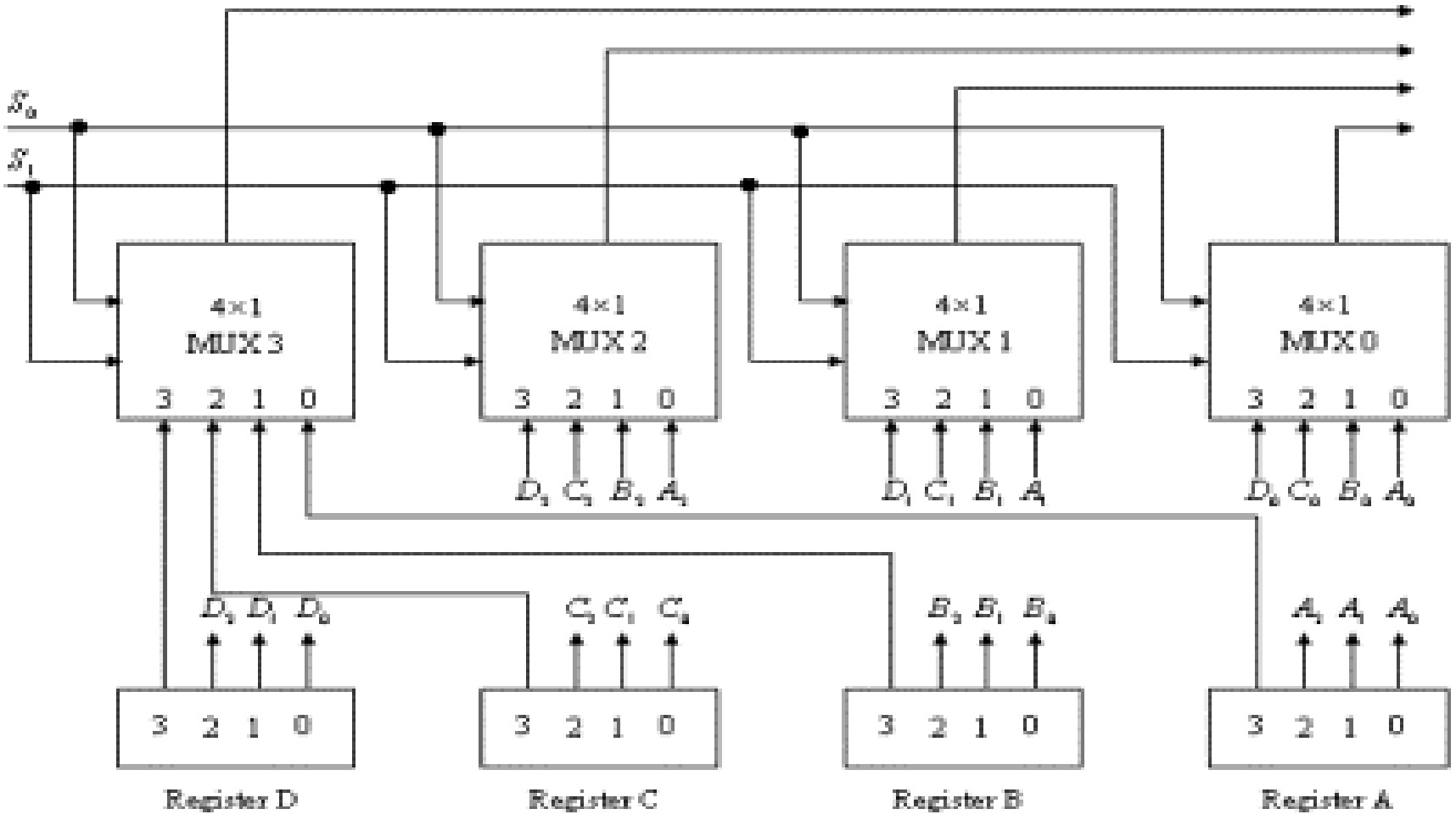
- The bus consists of four 4 x 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, S1 and S0.
- we use labels to show the connections from the outputs of the registers to the inputs of the multiplexers.
- For example, output 1 of register A is connected to input 0 of MUX 1 because this input is labeled A1.

Using Multiplexers

- The two selection lines S0 and S1 are connected to the selection inputs of all four multiplexers.
- The following table shows the register that is selected by the bus for each of the four possible binary values of the selection lines.

S1	S0	Register Selected
0	0	A
0	1	B
1	0	C
1	1	D

Using Multiplexers



Using Multiplexers

- The number of multiplexers needed to construct the bus is equal to n.
- The size of each multiplexer must be k x 1 since it multiplexes k data lines.

For example:

- A common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus.
- So Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.

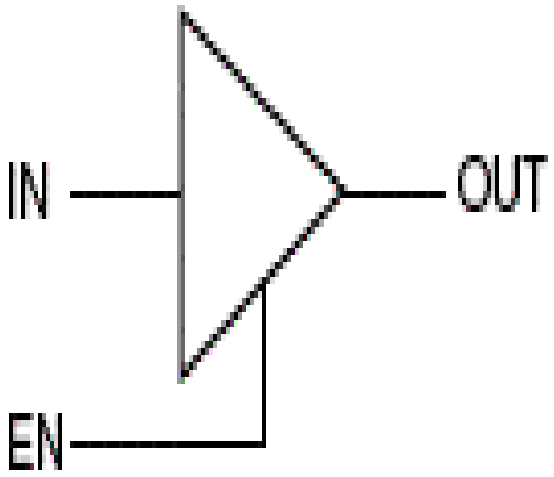
Three-state gate

- Three state gate is a digital circuit that exhibits three states.
- Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate and the third state is a high impedance state.
- The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have logic significance.

Three-state gate

- Three-state gates may perform any conventional logic, such as AND or NAND.
- However, the one most commonly used in the design of a bus system is the buffer gate.
- It is distinguished from a normal buffer by having both a normal input and a control input.

Three-state gate



(a) Logic symbol

EN	IN	OUT
0	X	Hi-Z
1	0	0
1	1	1

(b) Truth table

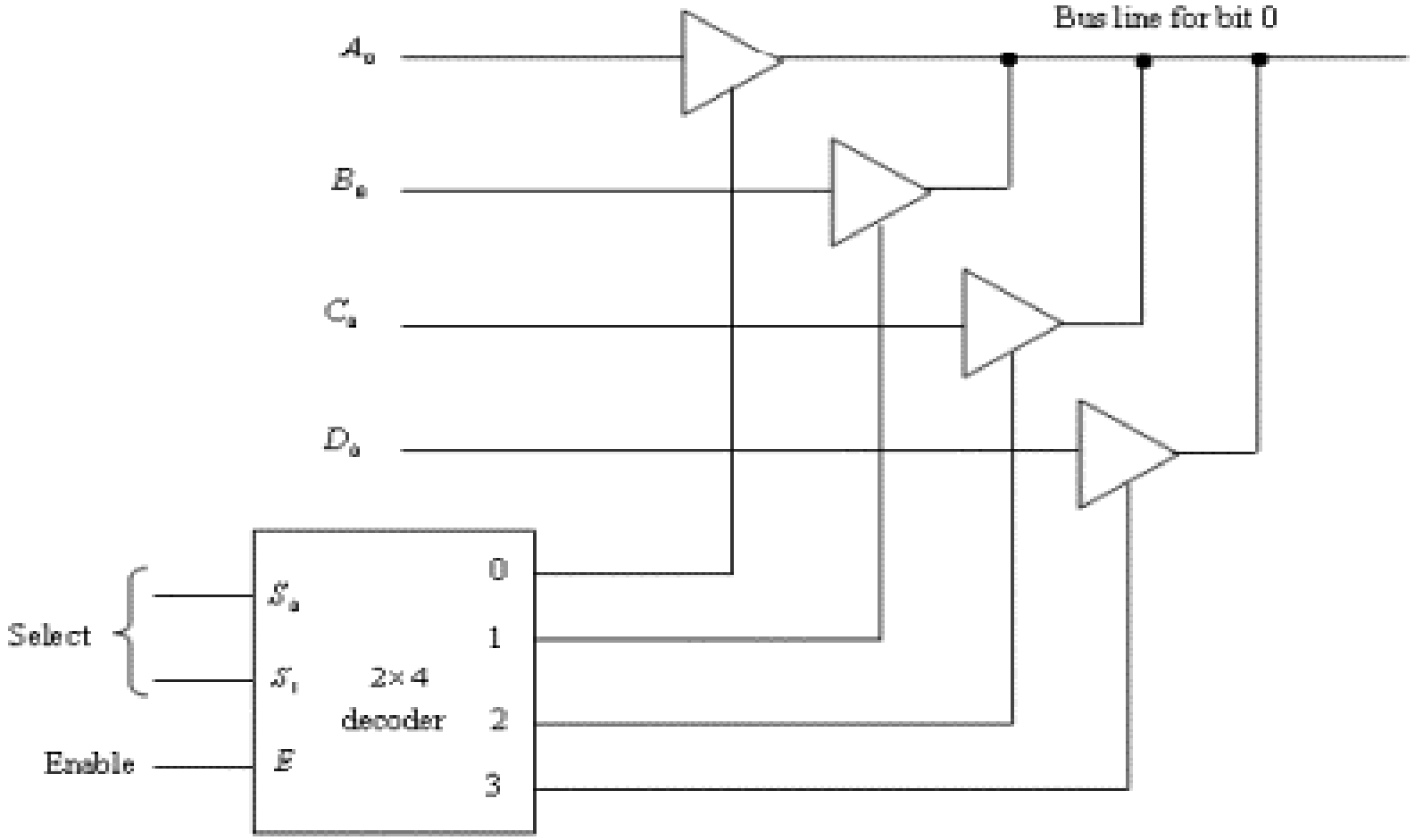
Three-state gate

- When the control input is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output equal to the normal input.

- When the control input is 0, the output is disabled and the gate goes to a high-impedance state**,** regardless of the value in the normal input.

- The high-impedance state of a three-state gate provides a special feature not available in other gates.

Three-state gate



Three-state gate

- The outputs of four buffers are connected together to form a single bus line.
- The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line.
- No more than one buffer may be in the active state at any given time.
- The connected buffers must be controlled so that only one three-state buffer has access to the bus line while all other buffers are maintained in a high-impedance state.

Three-state gate

- One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram.
- To construct a common bus for four registers of n bits each using three state buffer, we need n circuit with four buffer receives one significant bit from the four registers.
- Each common output produces one of the lines for the common bus for a total of n lines .
- Only one decoder is necessary to select between the four registers.

MEMORY TRANSFER

- A memory word will be symbolized by the letter **M.**
- The particular memory word among the many available word is selected by the memory address during the transfer.
- This will be done by enclosing the address in square brackets following the letter M.
- Consider a memory unit that receives the address from a register, called the address register, symbolized by **AR.**
- **The data are transferred to** another register, called the data register, symbolized by **DR**

MEMORY TRANSFER

READ OPERATION:

- The transfer of information from a memory word to the outside environment.

Read: DR ← M[AR]

- This causes a transfer of information into DR from the memory word M selected by the address in AR.

MEMORY TRANSFER

WRITE OPERATION:

- The write operation transfers the content of a data register to a memory word M selected by the address.

Write: M[AR] ← DR

- This causes a transfer of information from DR into the memory word M selected by the address in AR.

Control Memory

- **The function of Control unit is to Initiate sequences of microoperations**

•In a bus-organized system, Control signals *specifies microoperations to be executed in* group of bits that select the paths in multiplexers, decoders, and arithmetic logic units

- **Two major types of Control Unit**

»Hardwired Control :

- ❖ The control logic is implemented with gates, F/Fs, decoders, and other digital circuits
- ❖ Fast operation.
- ❖ Wiring change(if the design has to be modified)

»Microprogrammed Control:

- ➢ The control information is stored in a control memory, and the control memory is programmed to initiate the required sequence of microoperations
- ➢ Any required change can be done by updating the microprogram in control memory.
- ➢ Slow operation

Control Word

The control variables at any given time can be represented by a string of 1's and 0's.

Microprogrammed Control Unit

A control unit whose binary control variables are stored in memory (*control memory*).

Microinstruction

The microinstruction specifies one or more microoperations

Microprogram

A sequence of microinstruction

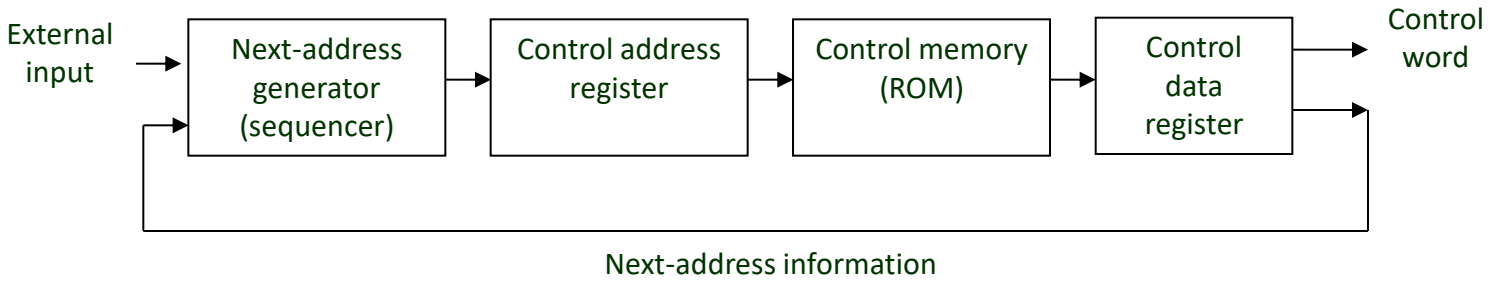
»Dynamic microprogramming :*Control Memory* =RAM

- RAM can be used for writing (*to change a writable control memory*)
- Microprogram is loaded initially from an auxiliary memory such as a magnetic disk

»Static microprogramming : *Control Memory* =ROM

- Control words in ROM are made permanent during the hardware production.

Microprogrammed Control Organization

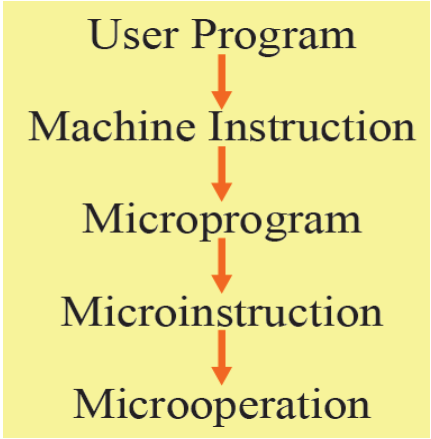


Control Memory

»A memory is part of a control unit :

»Computer Memory (*employs a microprogrammed control unit*)

- Main Memory : for storing user program (*Machine instruction/data*)
- Control Memory : for storing microprogram (*Microinstruction*)



Control Address Register

»Specify the address of the microinstruction

Sequencer

»Determine the address sequence that is read from control memory

»Next address of the next microinstruction can be specified several way depending on the sequencer input.

Sequencing Capabilities Required in a Control Storage

1. Incrementing of the control address register
2. Unconditional and conditional branches
3. A mapping process from the bits of the machine instruction to an address for control memory
4. A facility for subroutine call and return

Control data register

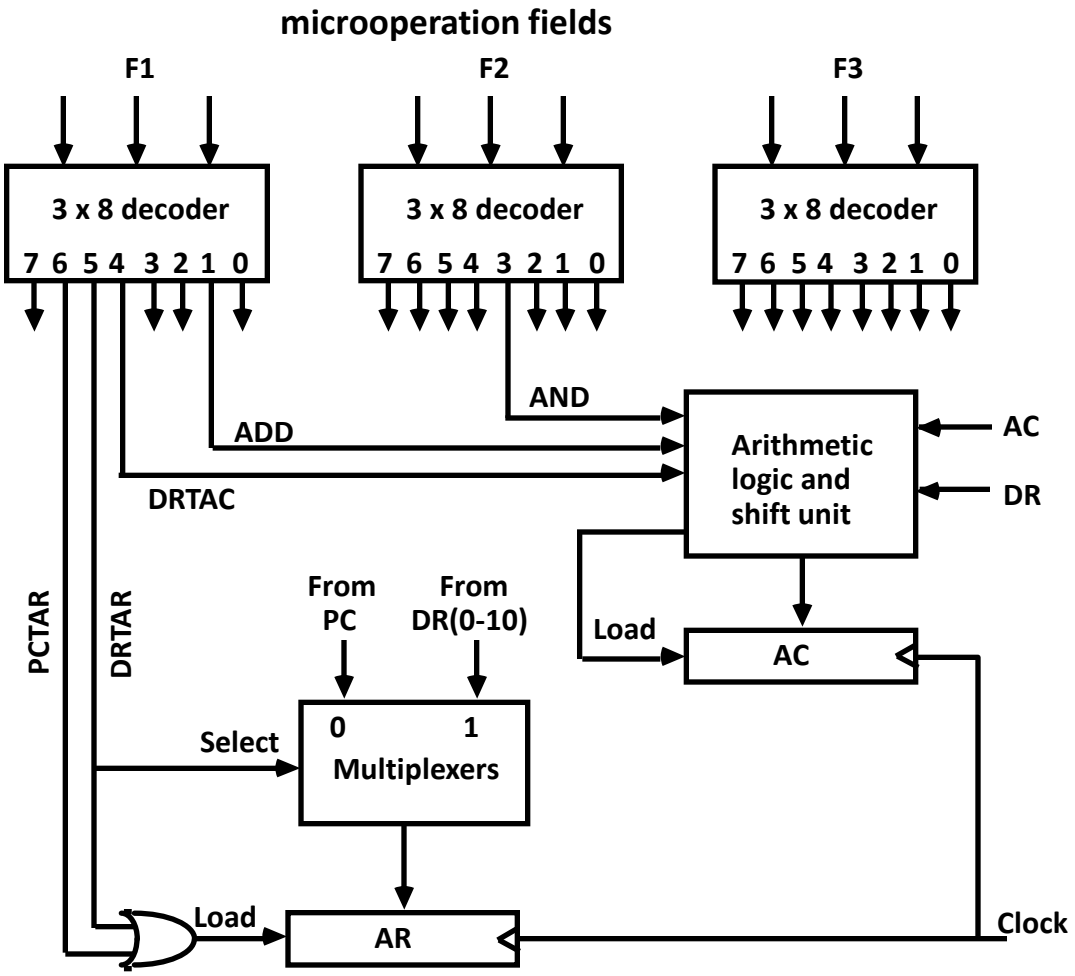
»Hold the microinstruction read from control memory

»Allows the execution of the microoperations specified by the control word ***simultaneously*** with the generation of the next microinstruction

DESIGN OF CONTROL UNIT

Presented by
A.Srinivasan, Associate Professor.

Design of Control Unit



Design of Control Unit

- The bits of microinstruction are usually divided into fields, with each field defining a distinct separate function.
- The various fields in instruction format provide control bits to initiate microoperation in the system.
- Each field requires a decoder to produce the corresponding control signals.
- The nine bits of microoperation field are divided into 3 subfields of 3 bits each.
- The control memory output of each subfield must be decoded to provide distinct microoperation.

Design of Control Unit

- The output of the decoder are connected to the appropriate input in the processor.
- In the above diagram, each of the output of 3 fields are decoded with 3x8 decoder to provide 8 outputs.
- Each of these outputs must be connected to proper circuit to initiate corresponding microoperation as specified.
- Ex: When F1=101, it transfers the content of DR(0-10) to AR. Similarly when F1=110, it transfers the content from PC to AR.
- As shown in above diagram, the o/p's 5 and 6 of decoder F1 are connected to the load i/p of AR. So that either one of these o/p is transferred to AR.

Design of Control Unit

- The multiplexer selects the information from DR when o/p 5 is active and from PC when o/p 6 is active.
- The other o/p's of decoder that initiate the transfer between register must be connected in a similar fashion.
- The ALU can be designed as shown in above diagram. In the above diagram the i/p's of ALU will come from the o/p of the decoder associated with the symbols AND, ADD and DRTAC respectively.
- The other o/p of the decoder that are associated with an AC operation must also be connected to ALU unit in a similar fashion.

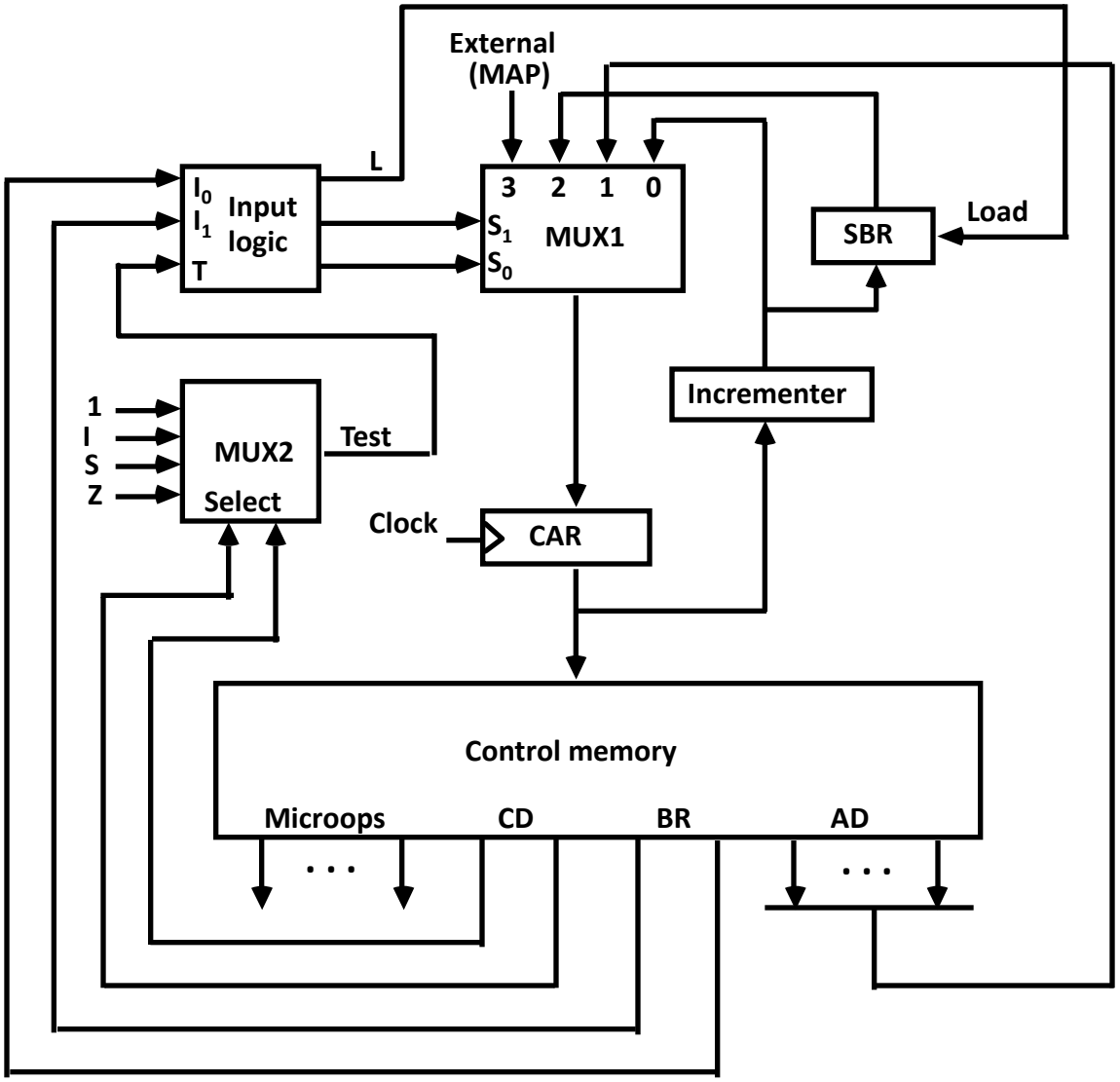
MICROPROGRAM SEQUENCER

Presented By
A.Srinivasan, Associate Professor

MICROPROGRAM SEQUENCER

- The basic component of microprogrammed control unit are the control memory and circuit that selects the next address.
- The address selection part is called as microprogram sequencer.
- The purpose of microprogram sequencer is to give address to control memory, so that microinstruction may be read and executed.
- The following diagram shows the interaction between sequencer and the memory attached to it.

Microprogram Sequencer



MICROPROGRAM SEQUENCER

- There are two multiplexer in the circuit. The first multiplexer selects an address from one of four sources and routes it into CAR.
- The second multiplexer test the value of a selected status bit and the result of the test is applied to an i/p logic circuit.
- The o/p from CAR provides the address to the control memory.
- The content of CAR is incremented and applied to the multiplexer's first i/p and to the subroutine register SBR.

MICROPROGRAM SEQUENCER

- The other 3 i/p's to the multiplexer comes from the address field of present microinstruction, o/p of SBR and from the external source that maps the instruction.
- The CD field of microinstruction selects one of the status bit in the second multiplexer.
- If the bit selected is equal to 1, the T(test) variable is equal to 1, otherwise it is 0.
- The T value together with the two bits from BR field go to i/p logic circuit.
- The i/p logic circuit will determine the type of operation that is available in the unit.

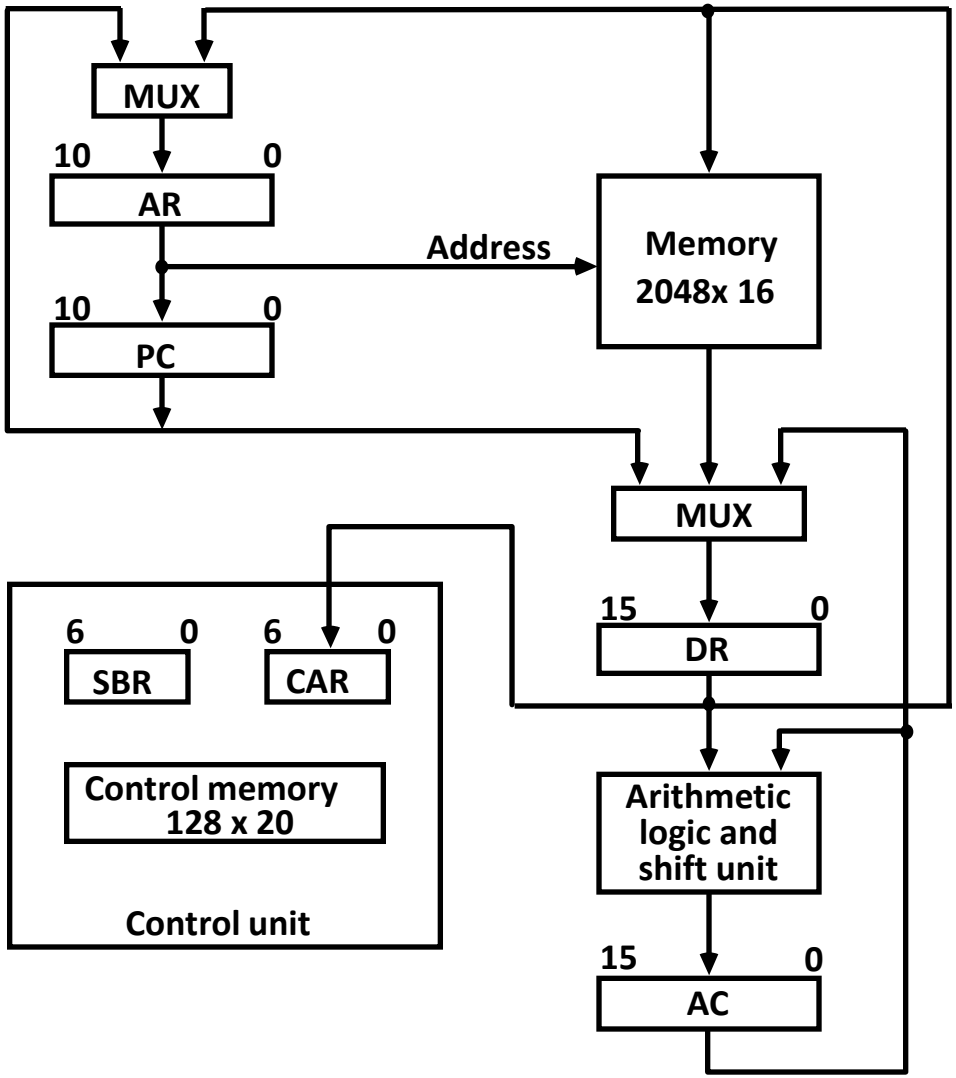
MICROPROGRAM SEQUENCER

- The i/p logic circuit has 3 i/p's I0, I1 and T and 3 o/p's S0,S1 and L (Load)
- Variables S0 and S1 selects one of the source address for CAR. Variable L enable the load i/p of SBR.
- The binary value of 2 selection variable determine the path of multiplexer.
- Ex: S1S0=10 Multiplexer i/p 2 is selected and establish a transfer path from SBR to CAR.

MICROPROAGRAM EXAMPLE

Microprogram Example

Computer Configuration



Microprogram Example

- It consist of two memory units such as

  -Main memory for storing instructions and data

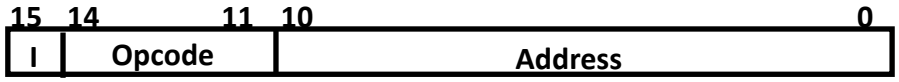
  -Control memory for storing micro programs
- Four registers are associated with the processor unit and two with control unit.
- The processor registers are PC, AR, DR and AC.
- The control unit has a CAR and Subroutine register SBR.

Microprogram Example

- The transfer of information among the registers in the processor is done through multiplexer.
- DR can receive information from AC, PC and Memory.
- AR can receive information from PC or DR.PC can receive information from AR.
- The ALU performs microoperation with data from AC and DR and places the result in AC.
- Input data written to memory come from DR and data read from memory can go only to DR.

Microprogram Example

Computer instruction format

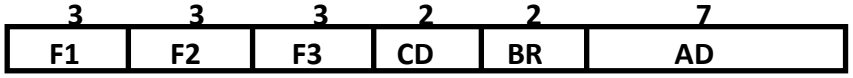


Four computer instructions

Symbol	OP-code	Description
ADD	0000	AC ← AC + M[EA]
BRANCH	0001	if (AC < 0) then (PC ← EA)
STORE	0010	M[EA] ← AC
EXCHANGE	0011	AC ← M[EA], M[EA] ← AC

EA is the effective address

Microinstruction Format



F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

Microprogram Example

- The 20 bits of the microinstruction are divided into 4 functional part.
- The 3 fields F1,F2 and F3 specify microoperation for the computer.
- The CD is the condition for branching which is used to select the status bit condition.
- The BR is branch field specify the type of branch to be used.
- The AD field contains a branch address.

Microinstruction Fields

F1	Microoperation	Symbol
000	None	NOP
001	AC ← AC + DR	ADD
010	AC ← 0	CLRAC
011	AC ← AC + 1	INCAC
100	AC ← DR	DRTAC
101	AR ← DR(0-10)	DRTAR
110	AR ← PC	PCTAR
111	M[AR] ← DR	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	AC ← AC - DR	SUB
010	AC ← AC ∨ DR	OR
011	AC ← AC ∧ DR	AND
100	DR ← M[AR]	READ
101	DR ← AC	ACTDR
110	DR ← DR + 1	INCDR
111	DR(0-10) ← PC	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	AC ← AC ⊕ DR	XOR
010	AC ← AC’	COM
011	AC ← shl AC	SHL
100	AC ← shr AC	SHR
101	PC ← PC + 1	INCPC
110	PC ← AR	ARTPC
111	Reserved	

Microinstruction Fields

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

BR	Symbol	Function
00	JMP	CAR ← AD if condition = 1
		CAR ← CAR + 1 if condition = 0
01	CALL	CAR ← AD, SBR ← CAR + 1 if condition = 1
		CAR ← CAR + 1 if condition = 0
10	RET	CAR ← SBR (Return from subroutine)
11	MAP	CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

Microinstruction Fields

- All transfer type microoperation symbols use five letters.
- The first 2 letters represents source register, the third letter is T and last 2 letters represents the destination register.

  Ex: DRTAC means DR to AC
- CD field consist of two bits needed to specify 4 status bit condition that uses the symbols U,I,S and Z.
- BR Field consist of 2 bits that works in conjunction address field(AD) to choose the address of the next microinstruction.

Symbolic Microinstruction

- **Sample Format**

Label:	**Micro-ops**	**CD**	**BR**	**AD**

- Label — may be empty or may specify symbolic address terminated with colon
- Micro-ops — consists of 1, 2, or 3 symbols separated by commas
- CD — one of {U, I, S, Z}
  U: Unconditional Branch
  I: Indirect address bit
  S: Sign of AC
  Z: Zero value in AC
- BR — one of {JMP, CALL, RET, MAP}
- AD — one of {Symbolic address, NEXT, empty}

Fetch Routine

- Fetch routine
  - Read instruction from memory
  - Decode instruction and update PC

Microinstructions for fetch routine:

```
AR ← PC
DR ← M[AR], PC ← PC + 1
AR ← DR(0-10), CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0
```

Symbolic microprogram for fetch routine:

```
        ORG 64
FETCH:  PCTAR          U  JMP  NEXT
        READ, INCPC    U  JMP  NEXT
        DRTAR          U  MAP
```

Binary microporgram for fetch routine:

Binary address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000