SCSX1009 Computer architecture and Parallel Processing

UNIT II PROCESSOR DESIGN

Computer arithmetic – Addition – Subtraction – Multiplication and Division algorithms – Floating point arithmetic operations - Micro programmed Control – Control memory – Address sequencing – Micro program Example – Design of control unit

---

COMPUTER ARITHMETIC

Introduction:

Data is manipulated by using the arithmetic instructions in digital computers. Data is manipulated to produce results necessary to give solution for the computation problems. The Addition, subtraction, multiplication and division are the four basic arithmetic operations. If we want then we can derive other operations by using these four operations. To execute arithmetic operations there is a separate section called arithmetic processing unit in central processing unit. The arithmetic instructions are performed generally on binary or decimal data. Fixed-point numbers are used to represent integers or fractions. We can have signed or unsigned negative numbers. Fixed-point addition is the simplest arithmetic operation. If we want to solve a problem then we use a sequence of well-defined steps. These steps are collectively called algorithm.

Data types

- Fixed-point binary
  - ✓ Signed-magnitude representation ‹
  - ✓ Signed-2's complement representation „
- Floating-point binary „
- Binary-coded decimal (BCD)

Addition and Subtraction

There are three ways of representing negative fixed-point binary numbers: Signed-magnitude, signed-l's complement, or signed-2's complement. Most computers use the signed-2's complement representation when performing arithmetic operations with integers. For floating-point operations, most computers use the signed-magnitude representation for the mantissa.

Addition and Subtraction with Signed-Magnitude Data

Where the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table 2.1. The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to present a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0. The algorithms for addition and subtraction are derived from the table and can be stated as follows

Table 2.1: Addition and Subtraction of Signed-Magnitude Numbers

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)

When the signs of A and B are same, add the two magnitudes and attach the sign of result is that of A. When the signs of A and B are not same, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A, if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result will be positive.

HARDWARE IMPLEMENTATION:-

First, a parallel-adder is needed to perform the micro operation A + B . Second, a comparator circuit is needed to establish if A > B, A = B, or A < B. Third, two parallel-subtractor circuits are needed to perform the micro operations A - B and B - A. The sign relationship can be determined from an exclusive OR gate with A, and B, as inputs.

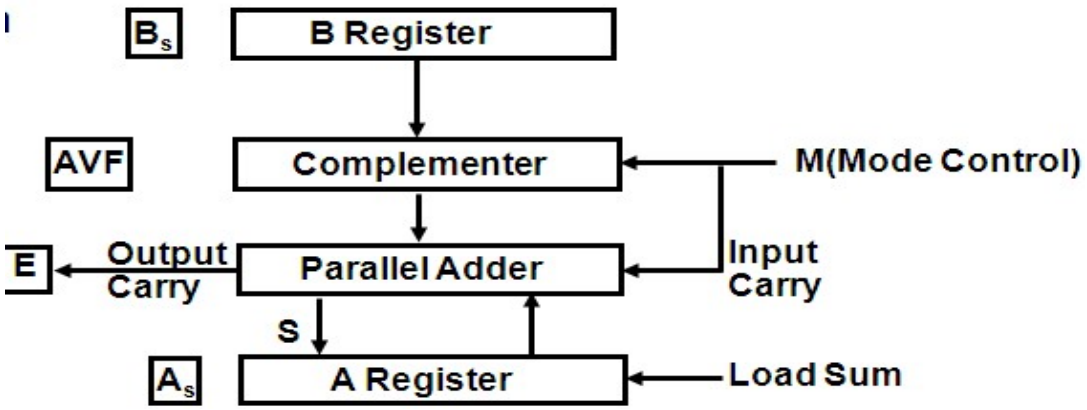


Fig. 2.1 Hardware for signed magnitude addition and subtraction

block diagram of the hardware for implementing the addition and subtraction operations. It consists of registers A and B and sign flip-flops A, and B, . Subtraction is done by adding A to the 2' s complement of B. The output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitudes of the two numbers. The add-overflow flip-flop AVF holds the overflow bit when A and B are added. The A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

Description

A_S Sign of A , B_S Sign of B ,A_S & A Accumulator , AVF Overflow bit for A + B

E Output carry for parallel adder

Data representation Signed magnitude – consists of the magnitude and negative sign (sign bit in binary, '0' for positive and '1' for negative)

– E.g. +14 = 0 0001110, -14= 1 0001110

Signed 1's complement – leaving out the sign bit, convert all 1's to 0's and 0's to 1's in the signed magnitude form of the data

– E.g. -14 = 1 1110001

Signed 2's complement – Add 1 to signed 1's complement representation of the data – E.g. -14 = 1 1110010

Hardware algorithm:

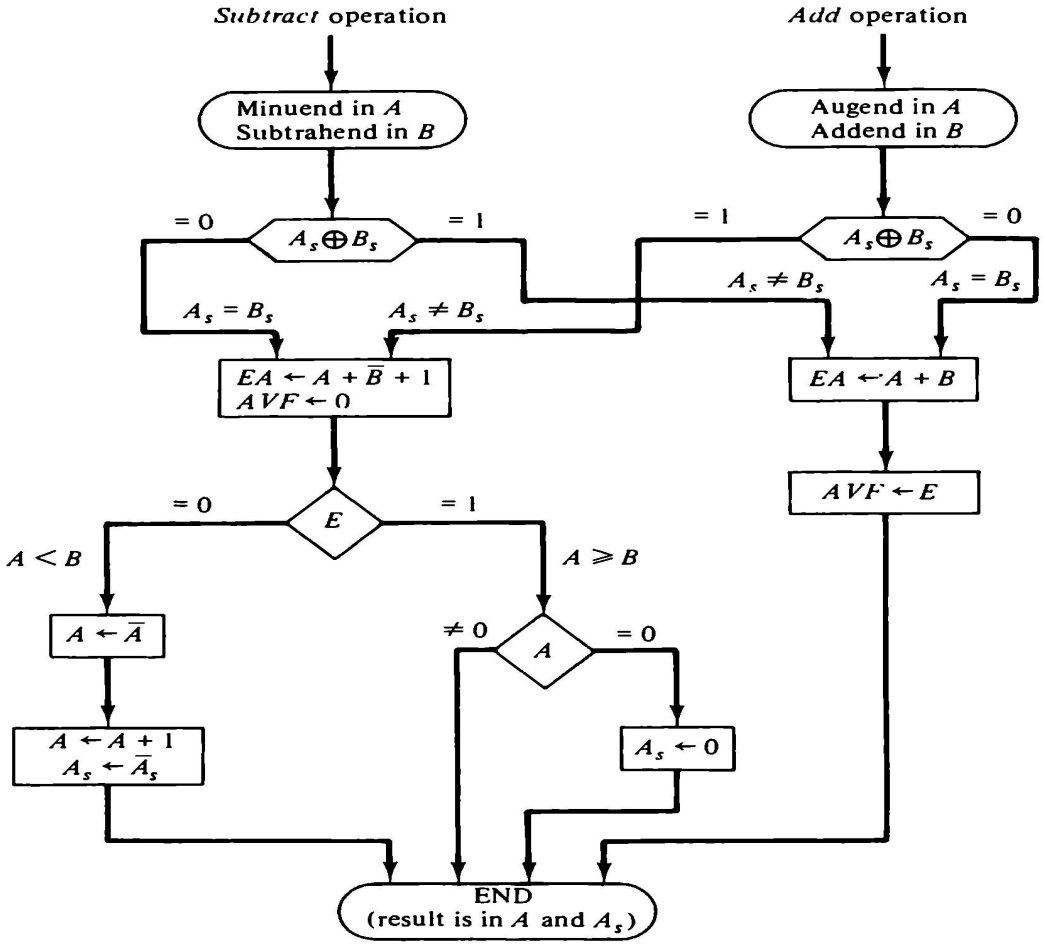


Fig.2.2 Flowchart for add and subtract operations

Signed magnitude addition and subtraction

- For an add operation, identical signs dictate that the magnitudes be added, different signs require that the magnitudes be subtracted
- For subtraction operation, different signs dictate that magnitudes be added, identical signs require that magnitudes be subtracted

AVF – Add-overflow flip-flop holds the overflow bit when A and B are added

Addition of A and B is done through parallel adder

2's complement addition and subtraction:

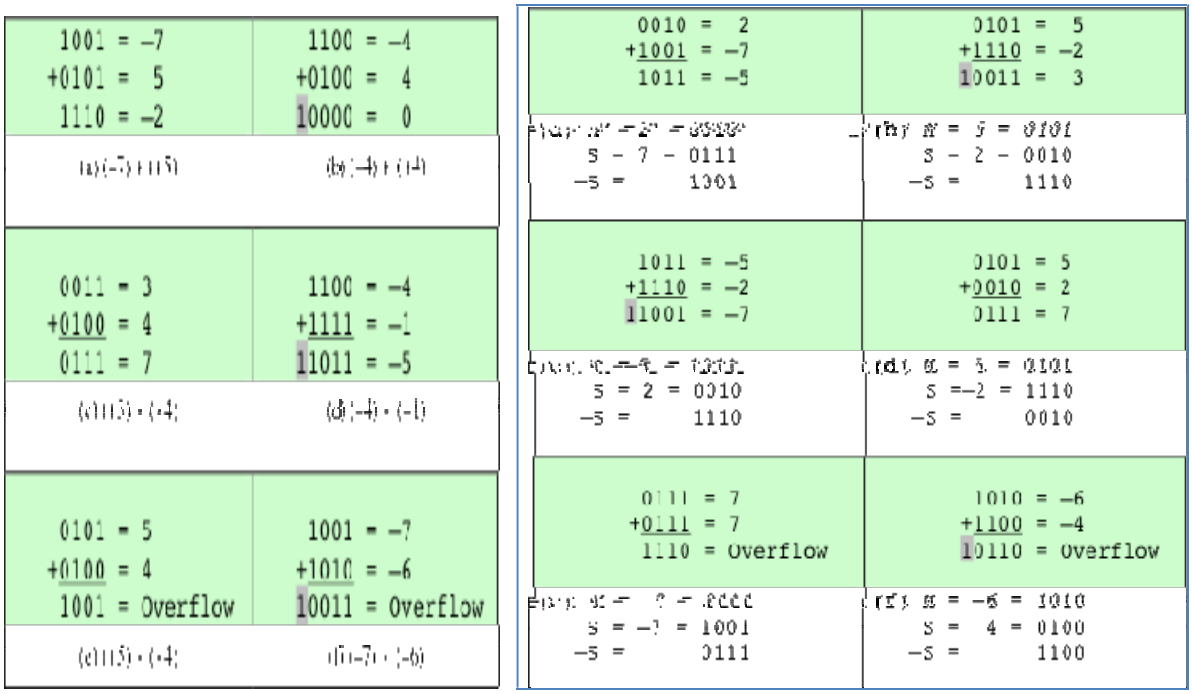


Fig. 2.3 Addition and subtraction of numbers in 2's complement representation

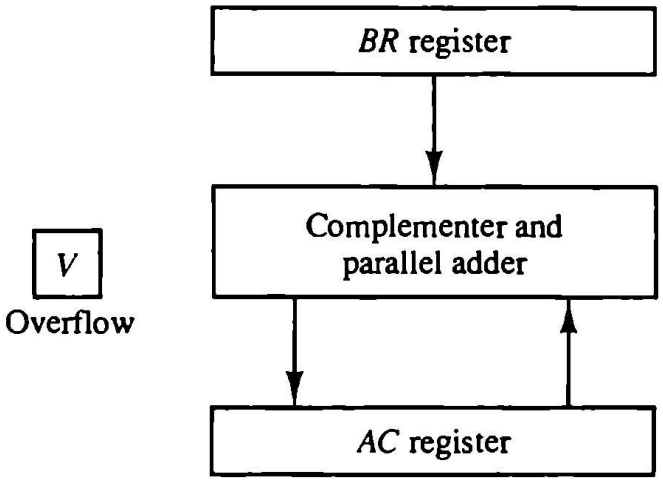


Fig.2.4 Block diagram of hardware for signed 2's complement addition and subtraction

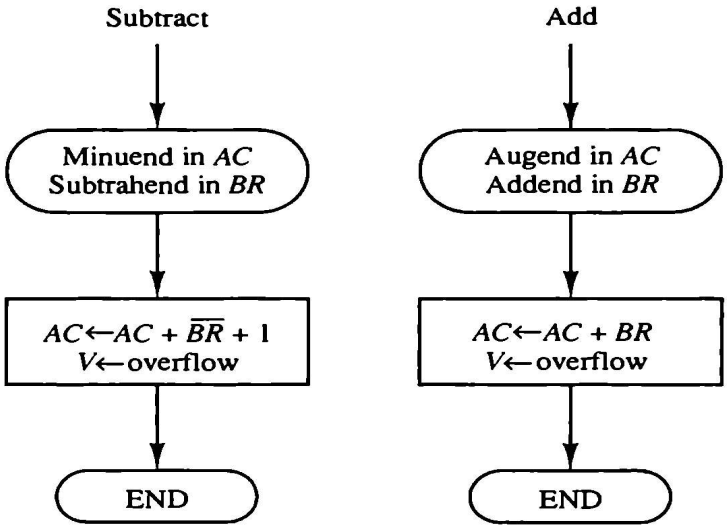
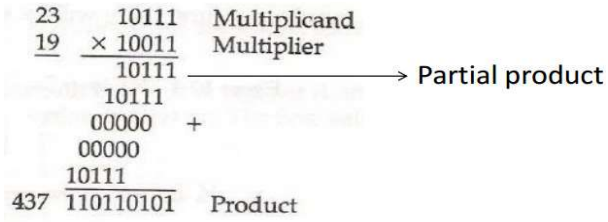


Fig.2.5 Algorithm for signed 2's complement addition and subtraction

Multiplication Algorithm:

A binary example:



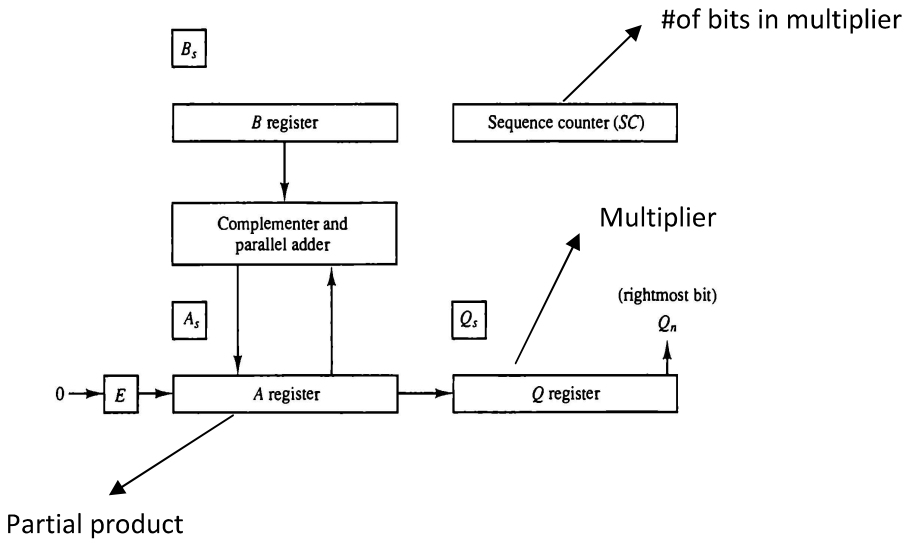
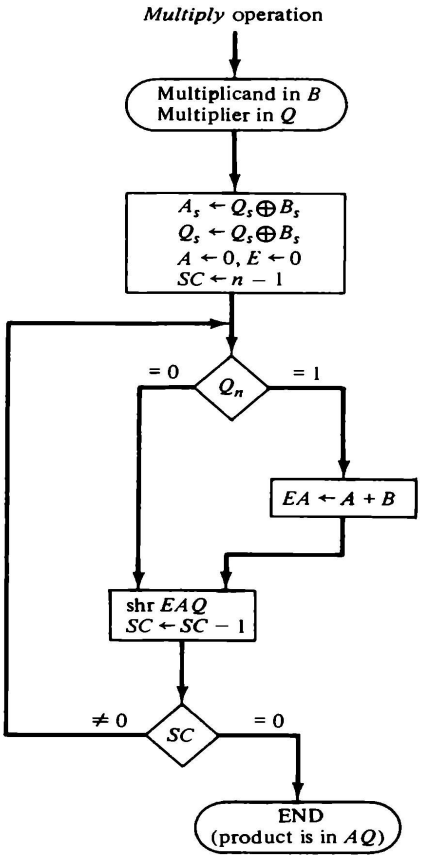


Fig.2.6 Block diagram of hardware for multiply operation



	B=11011 Q=00111
4	Q4=1,A=0,Qs=1 EA=A+B=1011 EAQ= 0 1011 0111 Shr EAQ= 0 0101 1011
3	Q3=1 EA = 1 0000 EAQ 1 0000 1011 Shr EAQ 0 1000 0101
2	Q2=1 EA= 1 0011 EAQ 1 0011 01 01 shr EAQ 0 1001 101 0
1	Q1=0 Shr EAQ 0 0100 1101=77
0	

Fig.2.7 Flow chart for multiply operation

Table 2.2 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 multiplication algorithm :

A= 00011 B= 00111 => A*B= A*(7) =A* (8-1) =A*8-A*1

Booth algorithm requires examination of the multiplier bits and shifting of the partial product

Q_n - LSB of multiplier Extra flip flop Q_{n+1} is appended to the multiplier bits to facilitate double bit inspection of the multiplier.

Compare bits of Q_n and Q_{n+1}

Rules are:

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

   In 2's complement representation, we can use Booth algorithm without change

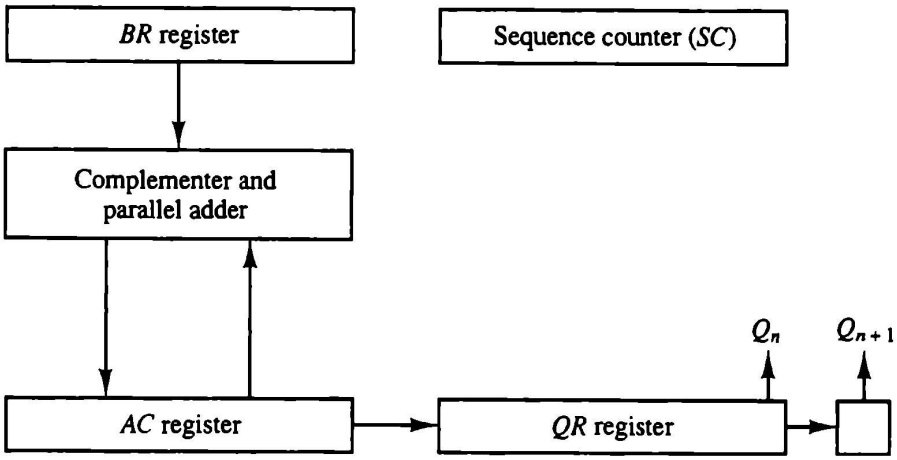


Fig.2.8 Block diagram of hardware for Booth algorithm

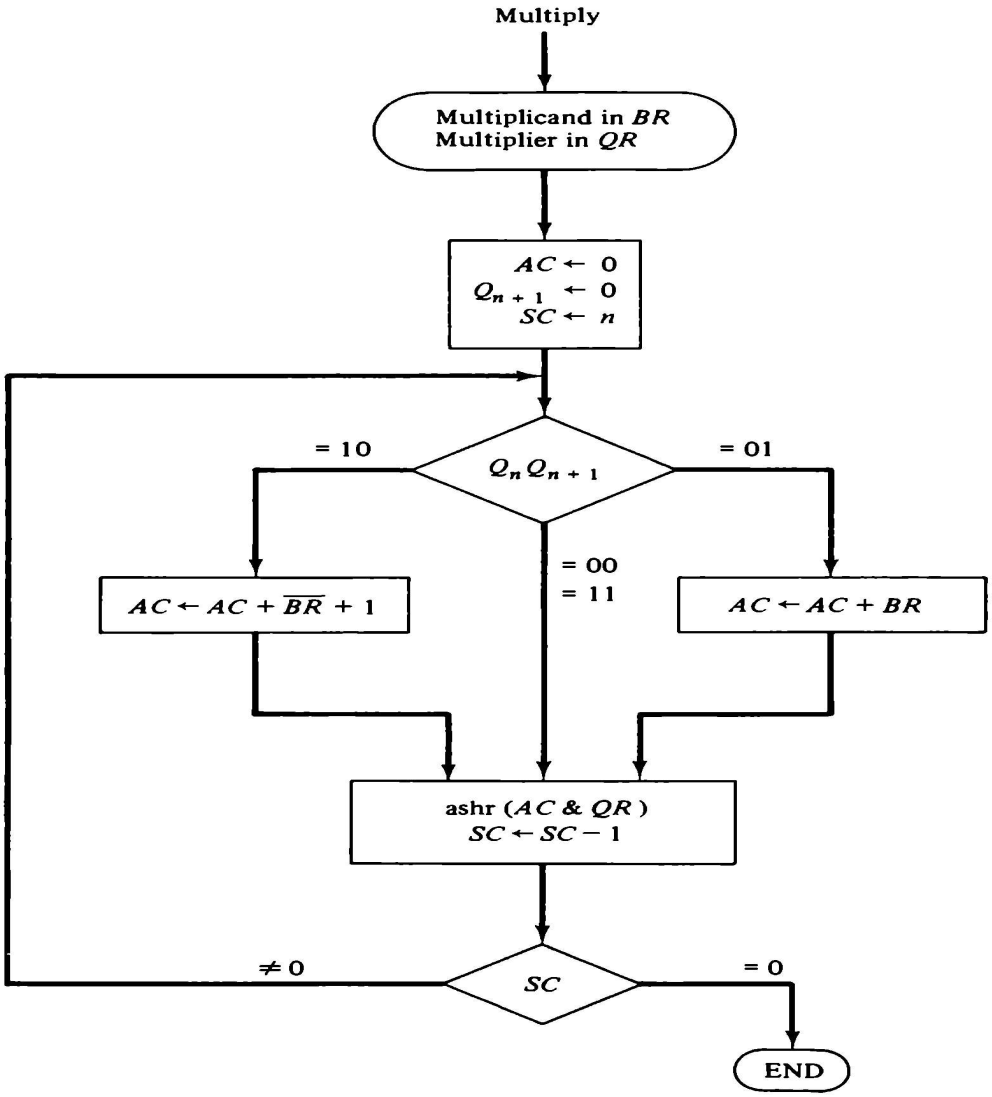


Fig. 2.9 Booth algorithm for multiplication of signed 2's complements numbers

Table 2.3 Example of multiplication with Booth Algorithm

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

Array multiplier: Fast approach

To check the bits of the multiplier one at a time and forming partial products is a sequential operation requiring a sequence of add and shift micro-operations. The multiplication of two binary numbers can be done with one micro-operation by using combinational circuit that forms the product bits all at once. This is a fast way since all it takes is the time for the signals to propagate through the gates that form the multiplication array. However, an array multiplier requires a large number of gates, and so it is not an economical unit for the development of ICs

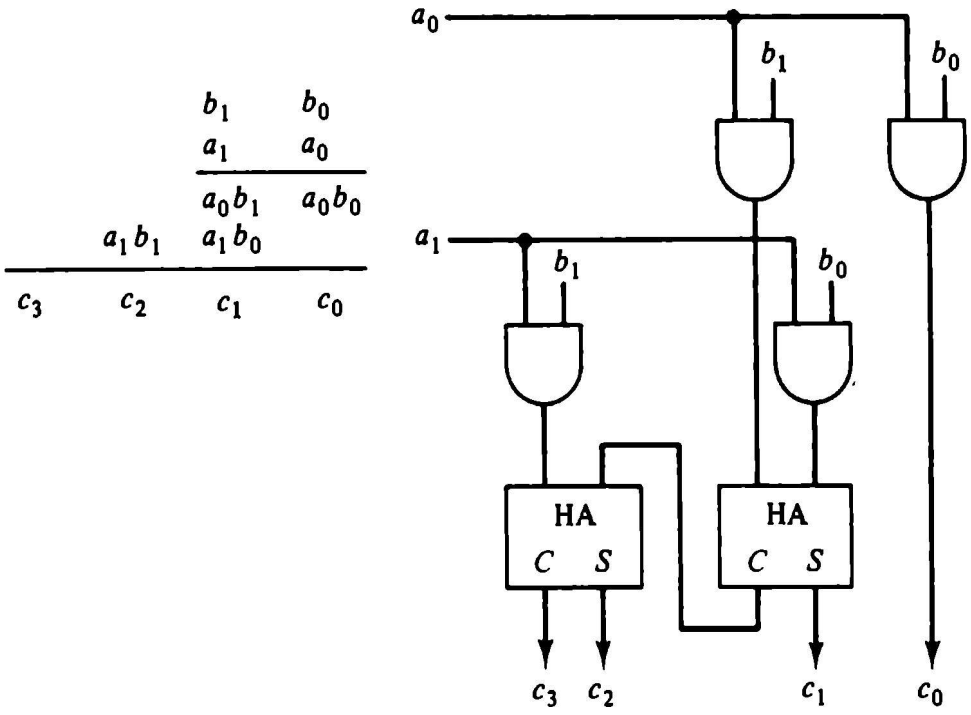


Fig. 2.10 : 2 bit by 2 bit array multiplier

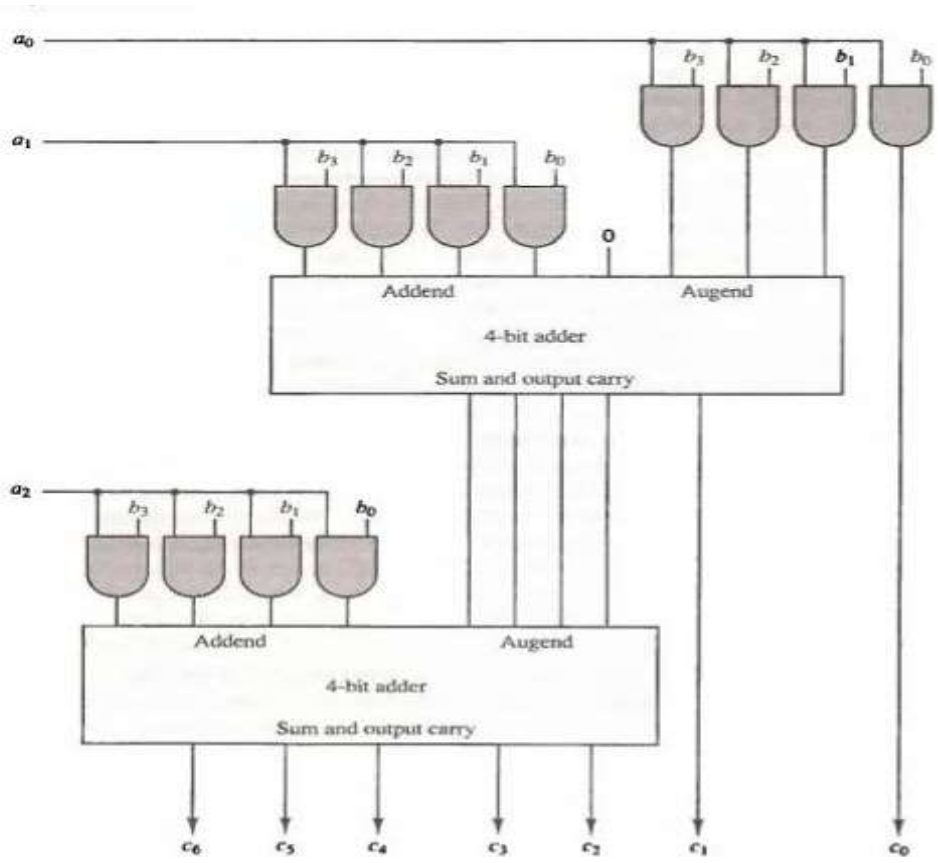


Fig. 2.11: 4 bit by 3bit array multiplier

Division algorithm:

Binary division – simpler because the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor

Division operation may result in a quotient with an overflow.

Divide overflow flip flop (DVF) is used to detect overflow

Divisor – B register, Dividend – A and Q register

If the signs of divisor and dividend are alike, the sign of the quotient is plus. Otherwise it is minus.

Best way to avoid divide overflow is to use floating point data.

Example for binary division :

Divisor:	11010	Quotient = Q
B = 10001	)0111000000	Dividend = A
	01110	5 bits of $A < B$, quotient has 5 bits
	011100	6 bits of $A \geq B$
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder $\geq B$
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder $< B$; enter 0 in Q; shift right B
	---010100	Remainder $\geq B$
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder $< B$; enter 0 in Q
	-----00110	Final remainder

Divisor B = 10001, $\overline{B} + 1$ = 01111

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
E = 1	1	01011		
Set Q_n = 1	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
E = 1	1	00101		
Set Q_n = 1	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
E = 0; leave Q_n = 0	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
E = 1	1	00011		
Set Q_n = 1	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
E = 0; leave Q_n = 0	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Fig.2.12 Example of binary division with digital hardware

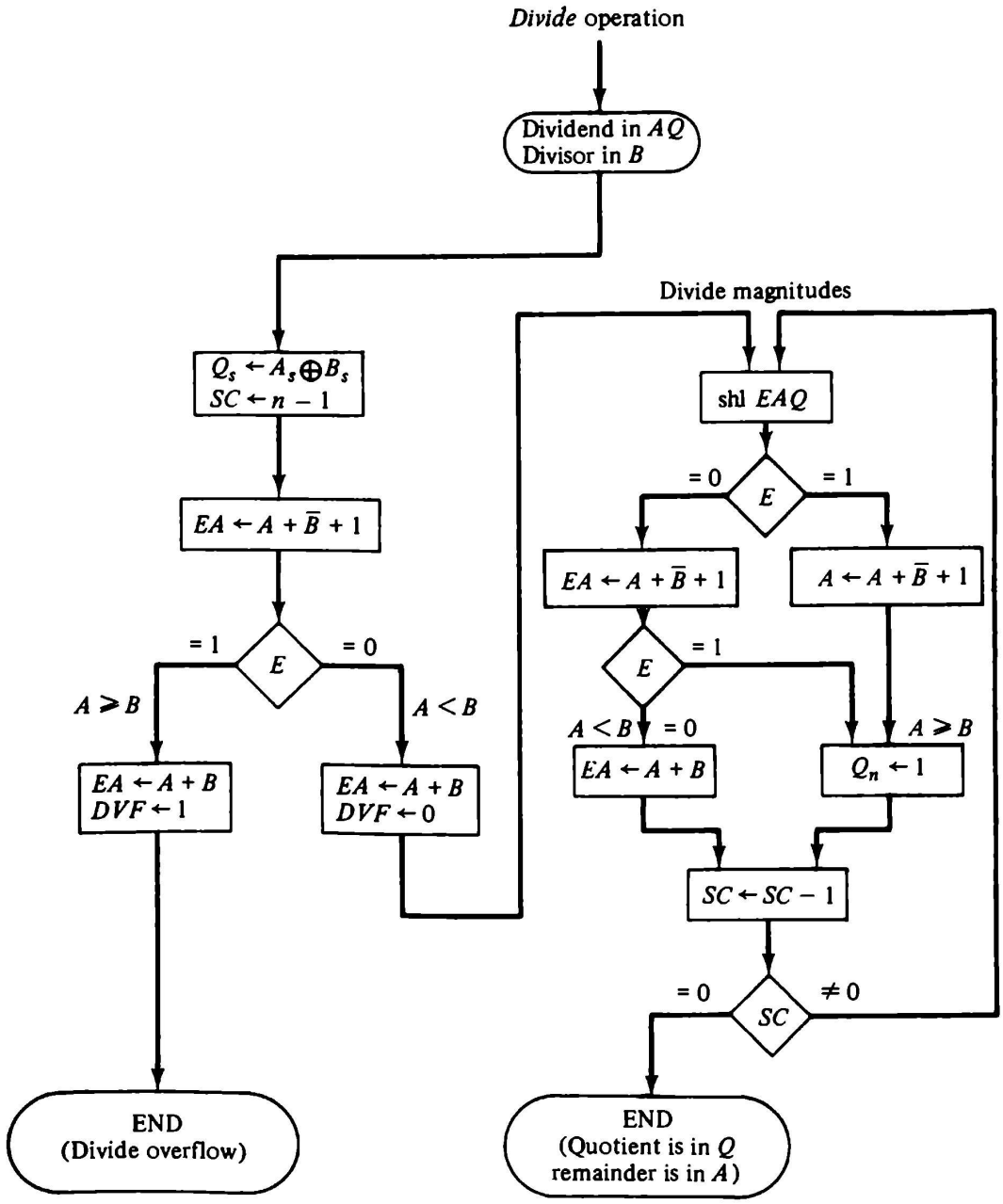


Fig. 2.13 Flow chart for divide operation

Floating Point Arithmetic Operations:

- Numbers too large for standard integer representations or that have fractional components are usually represented in scientific notation, a form used commonly by scientists and engineers.
- Examples: 4.25×10^1
- Addition and subtraction are more complex than multiplication and division

• Need to align mantissas

• Algorithm: — Check for zeros — Align significant (adjusting exponents) — Add or subtract significant — Normalize result

$$F = m \times r^e$$

where m: Mantissa, r: Radix, e: Exponent

It is necessary to make two exponents be equal before the mantissas can be added. We can either shift the first number three positions to the left, or shift the second number three positions to the right. When we store the mantissas in registers, shifting to the left causes a loss of most significant digits. Shifting to the right causes a loss of least significant digits. The second method is preferable because it only reduces the accuracy, while the first method may cause an error. The usual alignment procedure is to shift the mantissa that has the smaller exponent to the right by a number of places equal to the difference between the exponents. Now, the mantissas can be added.

$$\begin{array}{r} .5372400 \times 10^2 \\ +\ .0001580 \times 10^2 \\ \hline .5373980 \times 10^2 \end{array}$$

When two normalized mantissas are added, the sum may contain an overflow digit. An overflow can be corrected easily by shifting the sum once to the right and incrementing the exponent. When two numbers are subtracted, the result may contain most significant zeros as shown in the following example:

$$\begin{array}{r} .56780 \times 10^5 \\ -\ .56430 \times 10^5 \\ \hline .00350 \times 10^5 \end{array}$$

An underflow occurs if a floating-point number that has a 0 in the most significant position of the mantissa. To normalize a number that contains an underflow, we shift the mantissa to the left and decrement the exponent until a nonzero digit appears in the first position. Here, it is necessary to shift left twice to obtain $.35000 \times 10^3$. In most computers a normalization procedure is performed after each operation to ensure that all results are in a normalized form.

Floating-point multiplication and division need not do an alignment of the mantissas. Multiplying the two mantissas and adding the exponents can form the product. Dividing the mantissas and subtracting the exponents perform division.

Register Configuration:

Three registers are there, BR, AC, and QR. Each register is subdivided into two parts. The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part may use corresponding lower-case letter symbol.

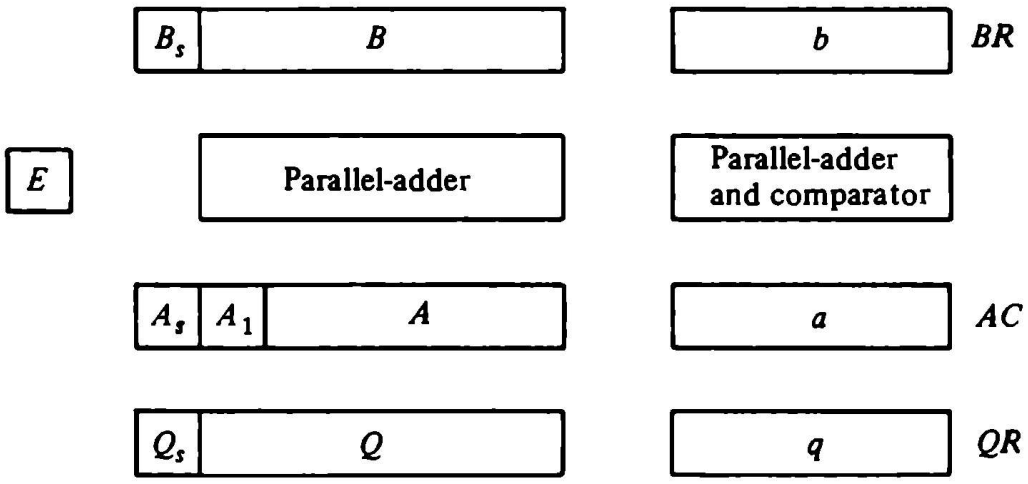


Fig. 2.14 Registers for floating point arithmetic operations

Assuming that each floating-point number has a mantissa in signed-magnitude representation and a biased exponent. Thus the AC has a mantissa whose sign is in A_s, and a magnitude that is in A. The diagram shows the most significant bit of A, labeled by A1. The bit in his position must be a 1 to normalize the number. Note that the symbol AC represents the entire register, that is, the concatenation of A_s, A and a. In the similar way, register BR is subdivided into B_s, B, and b and QR into Q_s, Q and q. A parallel-adder adds the two mantissas and loads the sum into A and the carry into E.

Addition and Subtraction of Floating Point Numbers

During addition or subtraction, the two floating-point operands are kept 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 mantissas.

3. Add or subtract the mantissas

4. Normalize the result

A floating-point number cannot be normalized, if it is 0. If this number is used for computation, the result may also be zero. Instead of checking for zeros during the normalization process we check for zeros at the beginning and terminate the process if necessary. The alignment of the mantissas must be carried out prior to their operation. After the mantissas are added or subtracted, the result may be un-normalized. The normalization procedure ensures that the result is normalized before it is transferred to memory.

For adding or subtracting two floating-point binary numbers, if BR is equal to zero, the operation is stopped, with the value in the AC being the result. If AC = 0, we transfer the content of BR into AC and also complement its sign we have to subtract the numbers. If neither number is equal it to zero, we proceed to align the mantissas. The magnitude comparator attached to exponents a and b gives three outputs, which show their relative magnitudes.

If the two exponents are equal, we go to perform the arithmetic operation. If the exponents are not equal, the mantissa having the smaller exponent is shifted to the right and its exponent incremented. This process is repeated until two exponents are equal.

The addition and subtraction of the two mantissas is similar to the fixed-point addition and subtraction algorithm presented in the following Fig

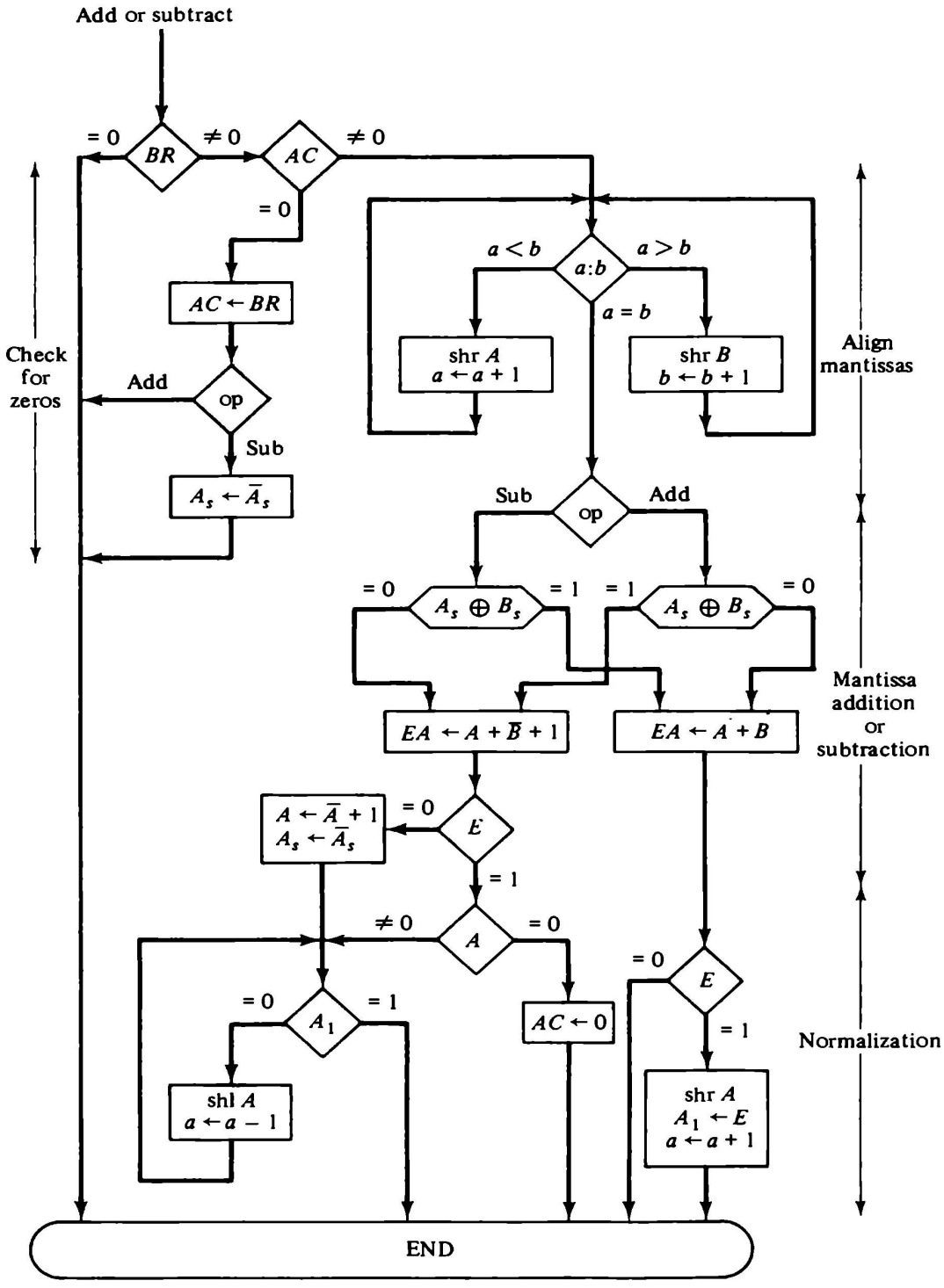


Fig. 2.15 Addition and subtraction of floating point numbers

Multiplication:

The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Add the exponents.
3. Multiply the mantissas
4. Normalize the result

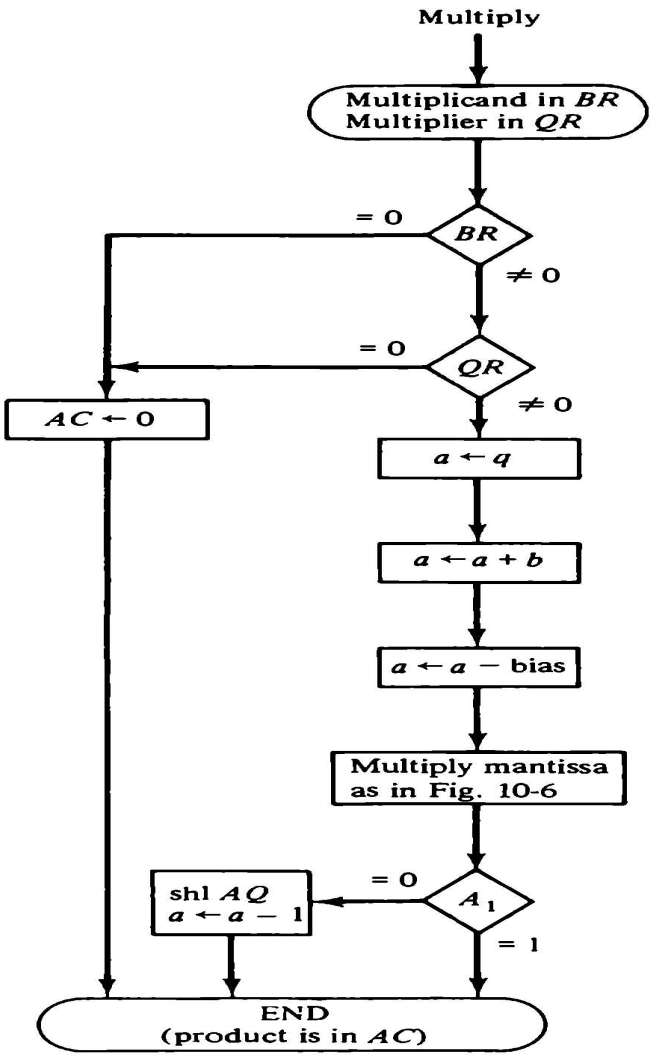


Fig. 2.16 Multiplication of floating point numbers

Division:

The algorithm can be subdivided into five consecutive parts:

1. Check for zeros.
2. Initialize the register and evaluate the sign
3. Align the dividend
4. Subtract the exponents.
5. Divide the mantissa

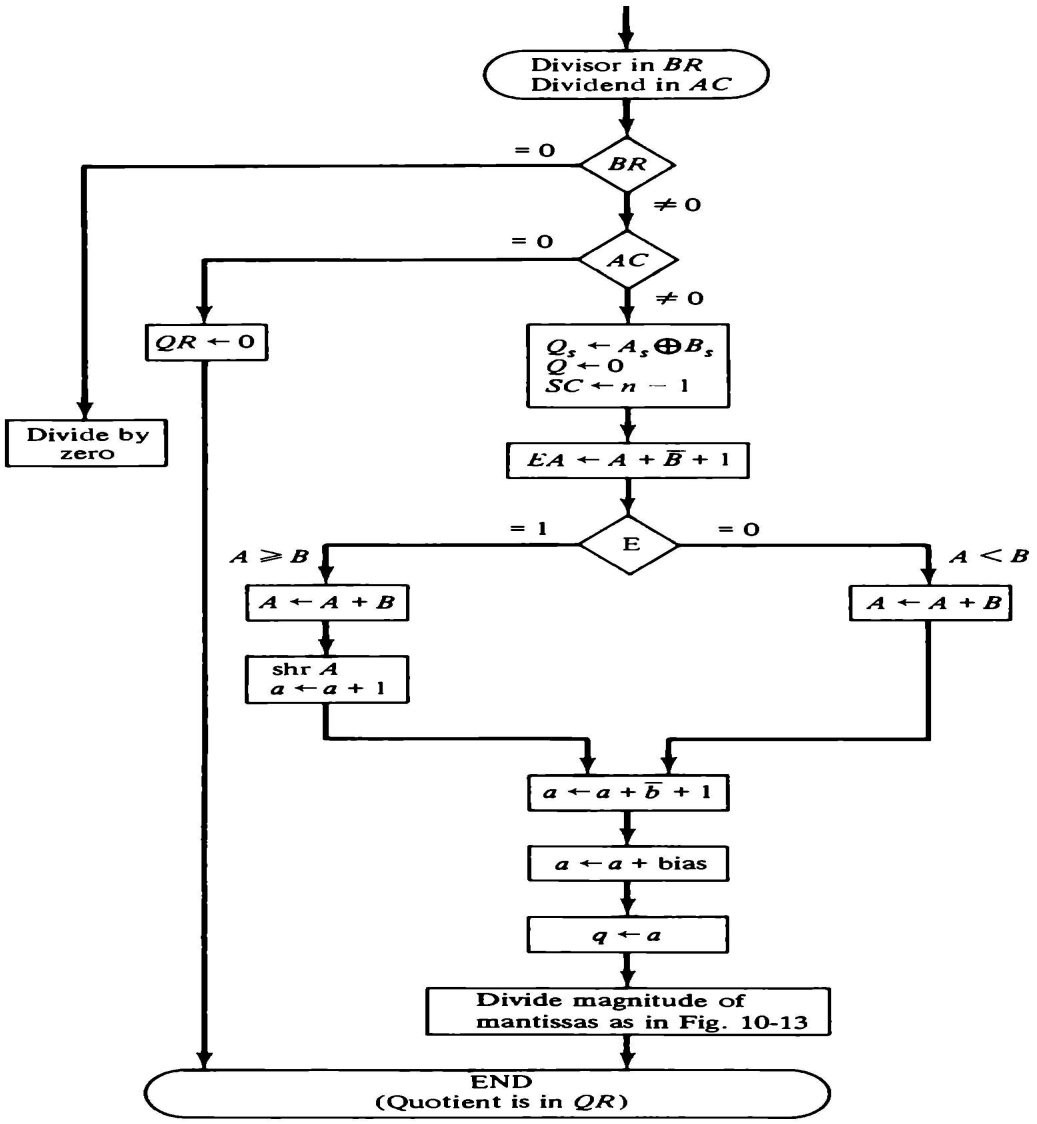


Fig. 2.17 Division of floating point numbers

Microprogrammed Control

Introduction:

Microprogram

- Program stored in memory that generates all the control signals required to execute the instruction set correctly
- Consists of microinstructions

Microinstruction

- Contains a control word and a sequencing word

Control Word - All the control information required for one clock cycle

Sequencing Word - Information needed to decide the next microinstruction adress

Control Memory(Control Storage: CS)

- Storage in the microprogrammed control unit to store the microprogram

Writeable Control Memory(Writeable Control Storage:WCS)

- CS whose contents can be modified

-> Allows the microprogram can be changed

-> Instruction set can be changed or modified

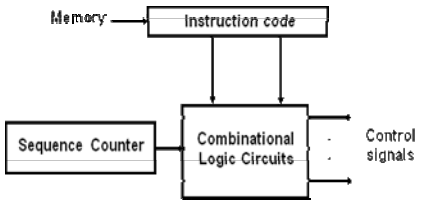
Dynamic Microprogramming

- Computer system whose control unit is implemented with a microprogram in WCS
- Microprogram can be changed by a systems programmer or a user

a. Control Memory

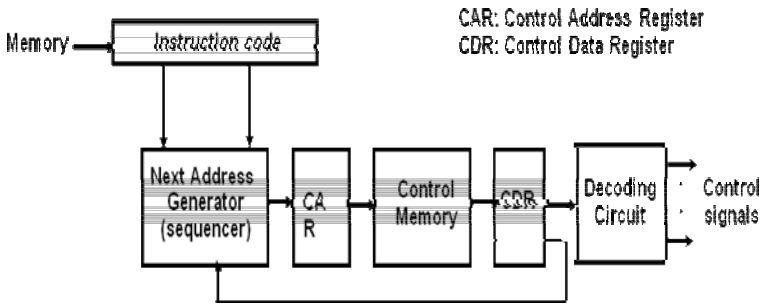
◆ Control Unit

- Initiate sequences of microoperations
- » Control signal *(that specify microoperations)* in a bus-organized system
  - ○ groups 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 : *Control Word in Control Memory*

- 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 :

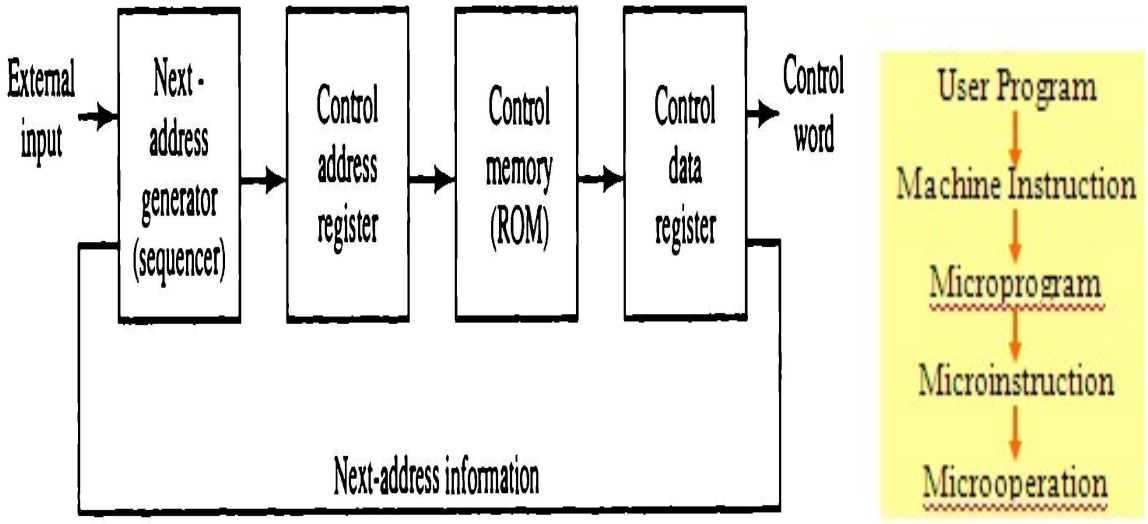


Fig. 2.18 Microprogrammed control operation

1) Control Memory
   - A memory is part of a control unit : *Microprogram 0/*
   - Computer Memory (*employs a microprogrammed control unit*)
   - Main Memory : for storing user program (*Machine instruction/data*)
   - Control Memory : for storing microprogram (*Microinstruction*)

2) Control Address Register
   - Specify the address of the microinstruction

3) Sequencer (= *Next Address Generator*)
   - 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

4) Control Data Register (= *Pipeline 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

◆ RISC Architecture Concept

RISC(Reduced Instruction Set Computer) system use hardwired control rather than microprogrammed control :

b. Address Sequencing

◆ Address Sequencing = Sequencer : Next Address Generator

- Selection of address for control memory

◆ Routine → *Subroutine : program used by other ROUTINES*

- Microinstruction are stored in control memory in groups

◆ Mapping

- Instruction Code - Address in control memory(*where routine is located*)

◆ Address Sequencing Capabilities : ***control memory address***

1) Incrementing of the control address register

2) Unconditional branch or conditional branch, depending on status bit conditions

3) Mapping process (bits *of the instruction address for control memory*)

4) A facility for subroutine return

◆ Selection of address for control memory :

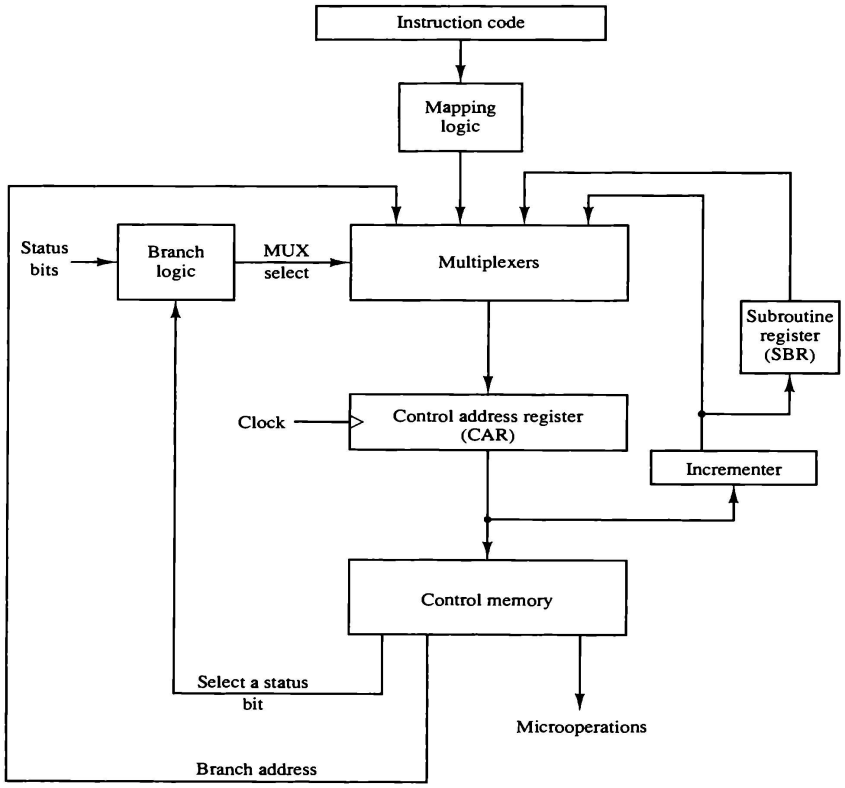


Fig. 2.19 Selection of address for control memory

- Multiplexer

  ① CAR Increment

  ② JMP/CALL

  ③ Mapping

  ④ Subroutine Return

- CAR : Control Address Register
  - » CAR receive the address from

    4 different paths

    1) Incrementer
    2) Branch address from

       control memory
    3) Mapping Logic
    4) SBR : Subroutine Register

- SBR : Subroutine Register
  - » Return Address can not be stored in ROM

» Return Address for a subroutine is stored in SBR

◆ Conditional Branching

- Status Bits
  - » Control the conditional branch decisions generated in the ***Branch Logic***
- Branch Logic
  - » Test the specified condition and Branch to the indicated address if the condition is met ; otherwise, the control address register is just incremented.
- Status Bit Test - Branch Logic
  - » 4 X 1 Mux - Input Logic

◆ Mapping of Instruction :

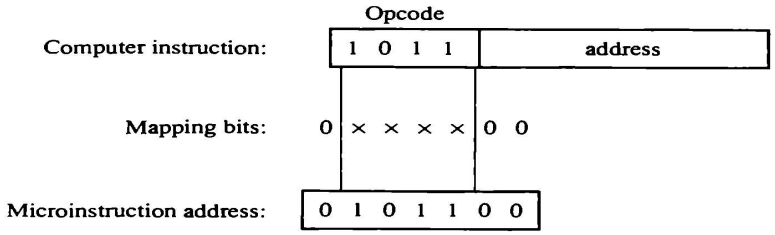


Fig. 2.20 Mapping from instruction code to microinstruction address

- 4 bit Opcode = specify up to 16 distinct instruction
- Mapping Process : Converts *the 4-bit Opcode* to *a 7-bit control memory address*
  - » 1) Place a “0” in the most significant bit of the address
  - » 2) Transfer 4-bit Operation code bits
  - » 3) Clear the two least significant bits of the CAR
- Mapping Function : Implemented by *Mapping ROM* or *PLD*

Control Memory Size : 128 words ($= 2^7$)

◆ Subroutine

- Subroutines are programs that are used by other routines

» Subroutine can be called from any point within the main body of the microprogram

- Microinstructions can be saved by subroutines that use common section of microcode
- Subroutine must have a provision for
  - » storing the return address during a subroutine call
  - » restoring the address during a subroutine return
    - Last-In First Out(LIFO) Register Stack

c. Microprogram Example

◆ Computer Configuration :

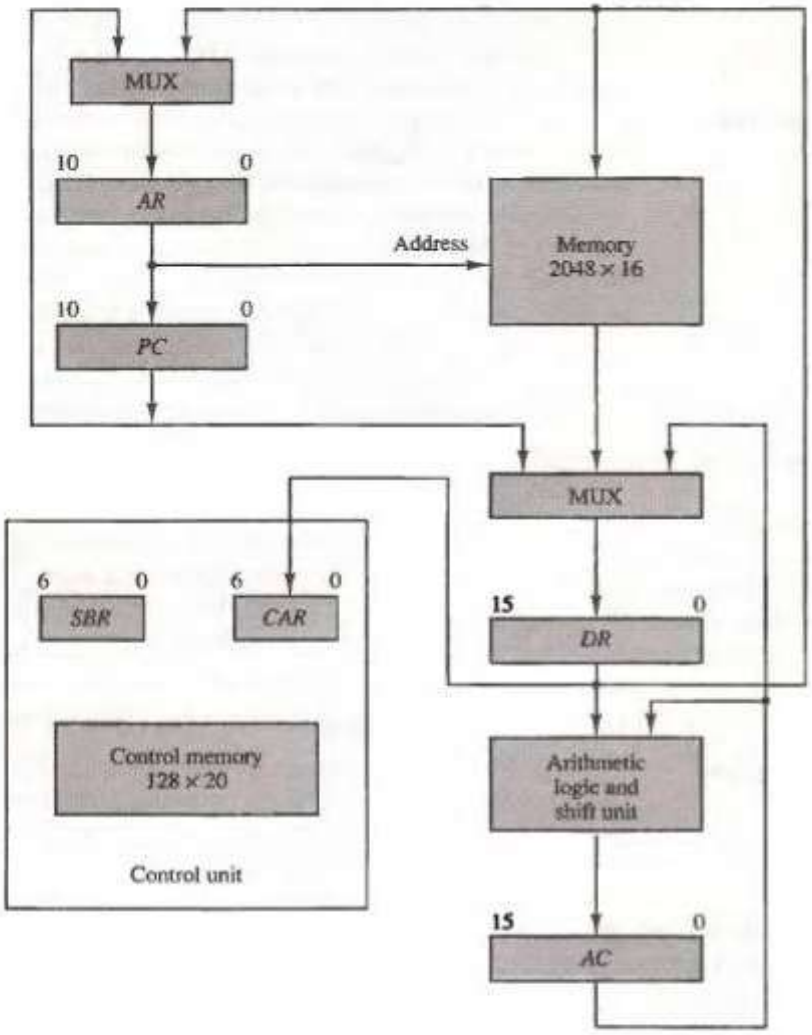
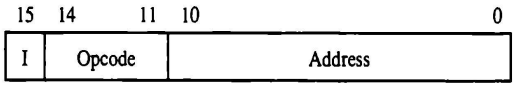


Fig. 2.21 Computer hardware configuration

- 2 Memory : Main memory(*instruction/data*), Control memory(*microprogram*)
  - » Data written to memory come from DR, and Data read from memory can go only to DR
- 4 CPU Register and ALU : DR, AR, PC, AC, ALU
  - » DR can receive information from AC, PC, or Memory (*selected by MUX*)
  - » AR can receive information from PC or DR (*selected by MUX*)
  - » PC can receive information only from AR
  - » ALU performs microoperation with data from AC and DR
- 2 Control Unit Register : SBR, CAR

◆ Instruction Format

- Instruction Format : ***Fig. 2.22***



(a) Instruction format

Fig. 2.22 Computer instruction format

- » I : 1 bit for indirect addressing
- » Opcode : 4 bit operation code
- » Address : 11 bit address for system memory

- Computer Instruction : Fig 2.23

Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M\ [EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M\ [EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

Fig. 2.23 Computer instruction- four computer instruction

◆ Microinstruction Format : Fig. 2.24

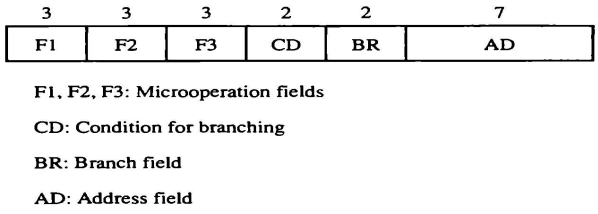


Fig. 2.24 Microinstruction code format (20 bits)

- 3 bit Micro operation Fields : F1, F2, F3
  - » 21Microoperation : ***Tab. 2.4***
  - » two or more conflicting microoperations can not be specified simultaneously
    - 010 001 000
  - » Clear AC to 0 and subtract DR from AC at the same time
  - » Symbol **DRTAC**(F1 = 100)
    - stand for a transfer from DR to AC (*T = to*)
- » 2 bit Condition Fields : CD
  - » 00 : Unconditional branch, **U** = 1
  - » 01 : Indirect address bit, **I** = DR(15)
  - » 10 : Sign bit of AC, **S** = AC(15)
  - » 11 : Zero value in AC, **Z** = AC = 0
- » 2 bit Branch Fields : BR
  - » 00 : **JMP**
    - Condition = 0 :
    - Condition = 1 :
  - » 01 : **CALL**
    - Condition = 0 :
    - Condition = 1 :
  - » 10 : **RET**
  - » 11 : **MAP**
- » 7 bit Address Fields : AD128 word : 128 X 20 bit

Table 2.4 Symbols and binary code for microinstruction fields

F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0\text{–}10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0\text{-}10) \leftarrow PC$	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow$ shl AC	SHL
100	$AC \leftarrow$ shr AC	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

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	$CAR \leftarrow AD$ if condition = 1 $CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1 $CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2\text{–}5) \leftarrow DR(11\text{–}14)$, $CAR(0,1,6) \leftarrow 0$

◆ Symbolic Microinstruction

① Label Field : Terminated with a colon (**:**)

② Microoperation Field : one, two, or three symbols, separated by commas

③ CD Field : U, I, S, or Z

④ BR Field : JMP, CALL, RET, or MAP

⑤ AD Field

a. Symbolic Address : Label (= Address)

b. Symbol "NEXT" : next address

c. Symbol "RET" or "MAP" : AD field = 0000000

- ORG : Pseudoinstruction(*define the origin, or first address of routine)*

◆ Fetch (**Sub**)Routine

- Memory Map(*128 words*) : ***Tab. 2.5, Tab. 2.6***

Table 2.5 Symbolic micro program

TABLE 7-2 Symbolic Microprogram (Partial)

Label	Microoperations	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

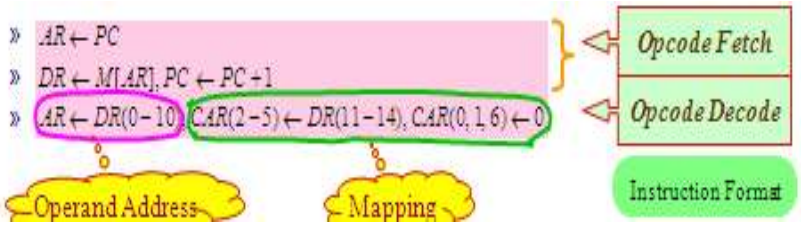
Table 2.6 Binary micro program for control memory (Partial)

Micro Routine	Address		Binary Microinstruction					
	Decimal	Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
FETCH	64	1000000	110	000	000	00	00	1000001
	65	1000001	000	100	101	00	00	1000010
	66	1000010	101	000	000	00	11	0000000
INDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	101	000	000	00	10	0000000

» Address 0 to 63 : Routines for the 16 instruction

» Address 64 to 127 : Any other purpose (*Subroutines : FETCH, INDRCT*)

Microinstruction for FETCH Subroutine



- Fetch Subroutine : address 64

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

- ◆ Symbolic Microprogram : ***Tab. 7-2***
  - The execution of MAP microinstruction in FETCH subroutine
    - » Branch to address 0**xxxx**00 (*xxxx = 4 bit Opcode*)
      - ADD : 0 **0000** 00 = 0
      - BRANCH : 0 **0001** 00 = 4
      - STORE : 0 **0010** 00 = 8
      - EXCHANGE : 0 **0011** 00 = 12, (*16, 20, … , 60*)
  - Indirect Address : **I** = 1
  - Indirect Addressing : $AR \leftarrow M[AR]$
  - INDRCT subroutine

Label	Microoperation	CD	BR	AD
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	0

$$DR \leftarrow M[AR]$$
$$AR \leftarrow DR$$

- Execution of Instruction
  - ADD instruction
  - BRANCH instruction
  - STORE instruction
  - EXCHANGE instruction

d. Design of Control Unit

- ◆ Decoding of Microinstruction Fields : ***Fig. 2.25***

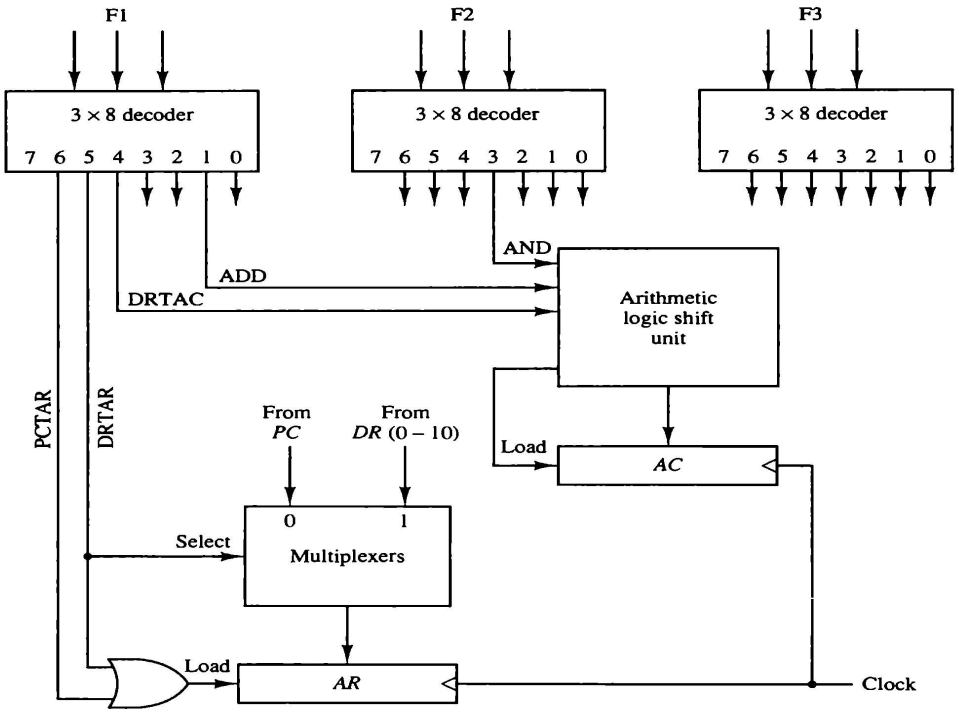


Fig. 2.25 Decoding of micro operation fields

- F1, F2, and F3 of Microinstruction are decoded with a 3 x 8 decoder
- Output of decoder must be connected to the proper circuit to initiate the corresponding microoperation
- F1 = 101 (5) : **DRTAR**
- F1 = 110 (6) : **PCTAR**
- Output 5 and 6 of decoder F1 are connected to the load input of AR (*two input of OR gate*)
- Multiplexer select the data from DR when output 5 is active
- Multiplexer select the data from AC when output 5 is inactive
- Arithmetic Logic Shift Unit
- Control signal of ALU in *hardwired control*
- Control signal will be now come from the *output of the decoders* associated with the AND, ADD, and DRTAC.

◆ Microprogram Sequencer : ***Fig. 2.26***

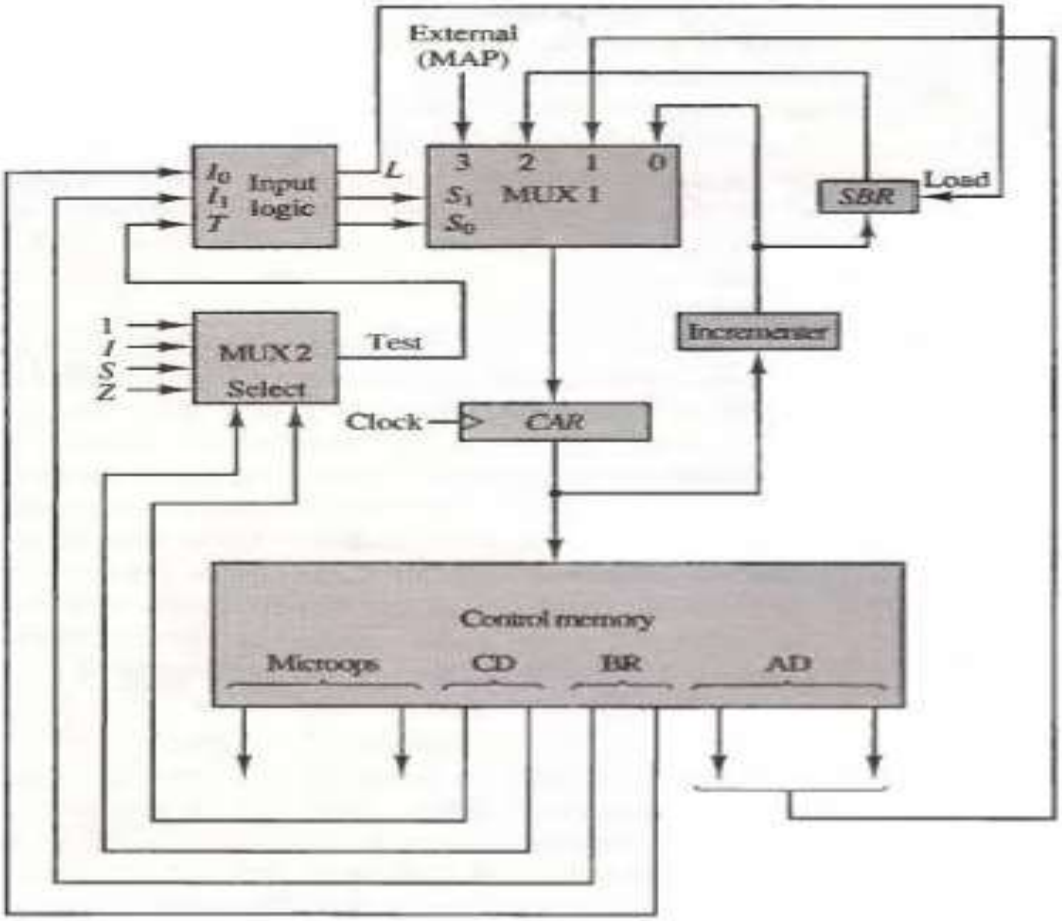


Fig. 2.26 Microprogram sequencer for a control memory

- Microprogram Sequencer select the next address for control memory
- MUX 1
  - Select an address source and route to CAR

  ① CAR + 1

  ② JMP/CALL

  ③ Mapping

  ④ Subroutine Return
- MUX 2
  - Test a status bit and the result of the test is applied to an input logic circuit
  - One of 4 Status bit is selected by Condition bit (**CD**)
- Design of Input Logic Circuit

- Select one of the source address($\mathbf{S_0}$, $\mathbf{S_1}$) for CAR
- Enable the load input(**L**) in SBR
- Input Logic Truth Table : ***Tab. 2.7***
    - » Input :
        - I_0, I_1 from Branch bit (**BR**)
        - T from MUX 2 (**T**)
    - » Output :
        - MUX 1 Select signal ($\mathbf{S_0}$, $\mathbf{S_1}$)

$S1 = I_1I_0' + I_1I_0 = I_1(I_0' + I_0) = I_1$

$S0 = I_1'I_0'T + I_1'I_0T + I_1I_0$

$= I_1'T(I_0' + I_0) + I_1I_0$

$= I_1'T + I_1I_0$

- SBR Load signal (**L**)

$L = I_1'I_0T$

$S_1 = I_1$

$S_0 = I_1I_0 + I_1'T$

$L = I_1'I_0T$

Table 2.7: Input logic truth table for micro program sequencer

BR Field		Input			MUX 1		Load *SBR*
		I_1	I_0	T	S_1	S_0	L
0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	1	0	1	0	0	0	0
0	1	0	1	1	0	1	1
1	0	1	0	×	1	0	0
1	1	1	1	×	1	1	0