Unit 3

Programming the basic Computer

1. Program and Categories of programs:

Program
A program is a list of instructions or statements for directing the computer to perform a required data-processing task.

Categories of programs

- **Binary code:** This is a sequence of instructions and operands in binary that list the exact representation of instructions as they appear in computer memory.

- **Octal or hexadecimal code:** This is an equivalent translation of the binary code to octal or hexadecimal representation.

- **Symbolic code:** The user employs symbols (letters, numerals, or special characters) for the operation part, the address part, and other parts of the instruction code. Each symbolic instruction can be translated into one binary coded instruction. This translation is done by a special program called an *assembler*. Because an assembler translates the symbols, this type of symbolic program is referred to as an *assembly language program*.

- **High-level programming languages:** These are special languages developed to reflect the procedures used in the solution of a problem rather than be concerned with the computer hardware behavior. An example of a high-level programming language is Fortran. It employs problem-oriented symbols and formats. The program is written in a sequence of statements in a form that people prefer to think in when solving a problem. However, each statement must be translated into a sequence of binary instructions before the program can be executed in a computer. The program that translates a high level language program to binary is called a *compiler*.

2. Assembly language and also the rules of language:

- The symbolic program (contains letters, numerals, or special characters) is referred to as an *assembly language program*.
- The basic unit of an assembly language program is a line of code.
- The specific language is defined by a set of rules that specify the symbols that can be used and they may be combined to form a line of code.

Rules of the Language:

Each line of an assembly language program is arranged in three columns called fields.

The fields specify the following information.

- ***The label field may be empty or it may specify a symbolic address.***
  A symbolic address consists of one, two, or three, but not more than three alphanumeric characters. The first character must be a letter; the next two may be letters or numerals. The symbol can be chosen arbitrarily by the programmer. A symbolic address in the label field is terminated by a comma so that it will be recognized as a label by the assembler.
- The **instruction** field specifies a machine instruction or a pseudo instruction.
  The instruction field in an assembly language program may specify one of the following items:
  - A memory-reference instruction (MRI)
  - A register-reference or input-output instruction (non-MRI)
  - A pseudo instruction with or without an operand
- The **comment** field may be empty or it may include a comment.
  A line of code may or may not have a comment, but if it has, it must be preceded by a slash for the assembler to recognize the beginning of a comment field. Comments are useful for explaining the program and are helpful in understanding the step-by-step procedure taken by the program. Comments are inserted for explanation purpose only and are neglected during the binary translation process.

3. Pseudo instruction:

A pseudo instruction is not a machine instruction but rather an instruction to the assembler giving information about some phase of the translation. Four pseudo instructions that are recognized by the assembler are listed in Table 3.1.

Symbol	Information for the Assembler
ORG N	Hexadecimal number N is the memory location for the instruction or operand listed in the following line
END	Denotes the end of symbolic program
DEC N	Signed decimal number N to be converted to binary
HEX N	Hexadecimal number N to be converted to binary

Table3.1: Definition of Pseudo instructions

- The ORG (origin) pseudo instruction informs the assembler that the instruction or operand in the following line is to be placed in a memory location specified by the number next to ORG. It is possible to use ORG more than once in a program to specify more than one segment of memory.
- The END symbol is placed at the end of the program to inform the assembler that the program is terminated.
- The other two pseudo instructions (DEC and HEX) specify the radix of the operand and tell the assembler how to convert the listed number to a binary number.

4. Define Assembler and explain First Pass of an assembler with flow chart.

Assembler

- An assembler is a program that accepts a symbolic language program and produces its binary machine language equivalent.
- The input symbolic program is called the source program and the resulting binary program is called the object program.

- The assembler is a program that operates on character strings and produces an equivalent binary interpretation.

First Pass of an assembler

- During the first pass, it generates a table that correlates all user-defined address symbols with their binary equivalent value.
- The binary translation is done during the second pass.
- To keep track of the location of instructions, the assembler uses a memory word called a location counter (abbreviated LC).
- The content of LC stores the value of the memory location assigned to the instruction or operand presently being processed.
- The ORG pseudo instruction initializes the location counter to the value of the first location.
- Since instructions are stored in sequential locations, the content of LC is incremented by 1 after processing each line of code.
- To avoid ambiguity in case ORG is missing, the assembler sets the location counter to 0 initially.
- The tasks performed by the assembler during the first pass are described in the flowchart of figure 3.1.
- LC is initially set to 0.
- A line of symbolic code is analyzed to determine if it has a label (by the presence of a comma).
- If the line of code has no label, the assembler checks the symbol in the instruction field.
- If it contains an ORG pseudo instruction, the assembler sets LC to the number that follows ORG and goes back to process the next line.
- If the line has an END pseudo instruction, the assembler terminates the first pass and goes to the second pass.
- If the line of code contains a label, it is stored in the address symbol table together with its binary equivalent number specified by the content of LC Nothing is stored in the table if no label is encountered.
- LC is then incremented by 1 and a new line of code is processed.

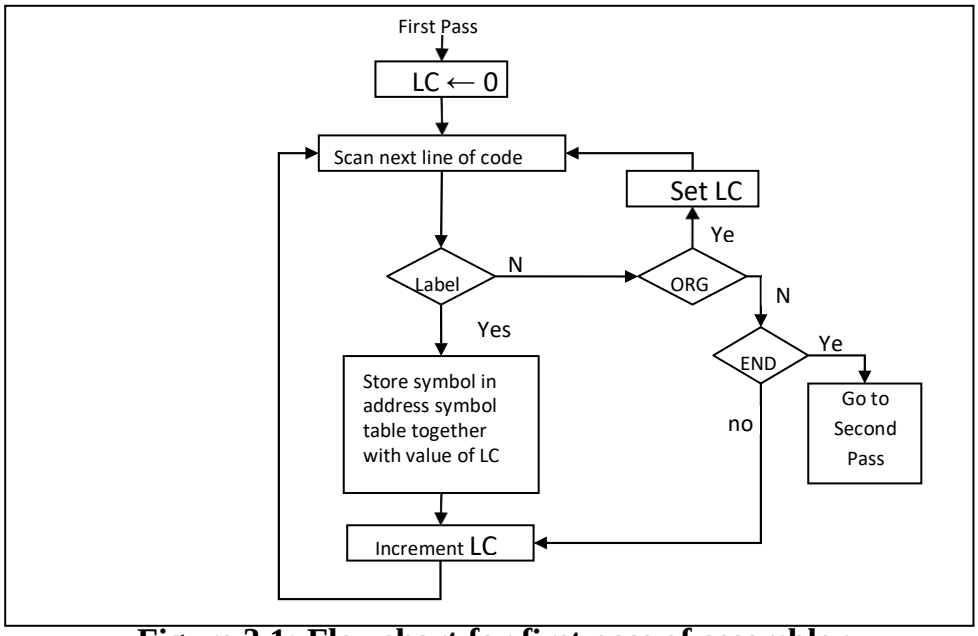


Figure 3.1: Flowchart for first pass of assembler

5. Explain the working of Second Pass Assembler with its flowchart.

- Machine instructions are translated during the second pass by means of table-lookup procedures.
- A table-lookup procedure is a search of table entries to determine whether a specific item matches one of the items stored in the table.
- The assembler uses four tables. Any symbol that is encountered in the program must be available as an entry in one of these tables; otherwise, the symbol cannot be interpreted.
    - MRI table
    - Non-MRI table
    - Address symbol table
    - Pseudo instruction table
- The entries of the pseudo instruction table are the four symbols ORG, END, DEC, and HEX.
- Each entry refers the assembler to a subroutine that processes the pseudo instruction when encountered in the program.
- The MRI table contains the seven symbols of the memory-reference instructions and their 3-bit operation code equivalent.
- The non-MRI table contains the symbols for the 18 register-reference and input-output instructions and their 16-bit binary code equivalent.
- The address symbol table is generated during the first pass of the assembly process.
- The assembler searches these tables to find the symbol that it is currently processing in order to determine its binary value.
- The tasks performed by the assembler during the second pass are described in the flowchart of Figure 3.2.
- LC is initially set to 0.
- Lines of code are then analyzed one at a time.
- Labels are neglected during the second pass, so the assembler goes immediately to the instruction field and proceeds to check the first symbol encountered.
- It first checks the pseudo instruction table.
- A match with ORG sends the assembler to a subroutine that sets LC to an initial value.

- A match with END terminates the translation process. An operand pseudo instruction causes a conversion of the operand into binary.
- This operand is placed in the memory location specified by the content of LC.
- The location counter is then incremented by 1 and the assembler continues to analyze the next line of code.
- If the symbol encountered is not a pseudo instruction, the assembler refers to the MRI table.
- If the symbol is not found in this table, the assembler refers to the non-MRI table.
- A symbol found in the non-MRI table corresponds to a register reference or input-output instruction.
- The assembler stores the 16-bit instruction code into the memory word specified by LC.
- The location counter is incremented and a new line analyzed.
- When a symbol is found in the MRI table, the assembler extracts its equivalent 3-bit code and inserts it m bits 2 through 4 of a word.
- A memory reference instruction is specified by two or three symbols.
- The second symbol is a symbolic address and the third, which may or may not be present, is the letter I.
- The symbolic address is converted to binary by searching the address symbol table.
- The first bit of the instruction is set to 0 or 1, depending on whether the letter I is absent or present.
- The three parts of the binary instruction code are assembled and then stored in the memory location specified by the content of LC.
- The location counter is incremented and the assembler continues to process the next line.
- One important task of an assembler is to check for possible errors in the symbolic program. This is called ***error diagnostics***.
- One such error may be an invalid machine code symbol which is detected by its being absent in the MRI and non-MRI tables.
- The assembler cannot translate such a symbol because it does not know its binary equivalent value.
- In such a case, the assembler prints an error message to inform the programmer that his symbolic program has an error at a specific line of code.

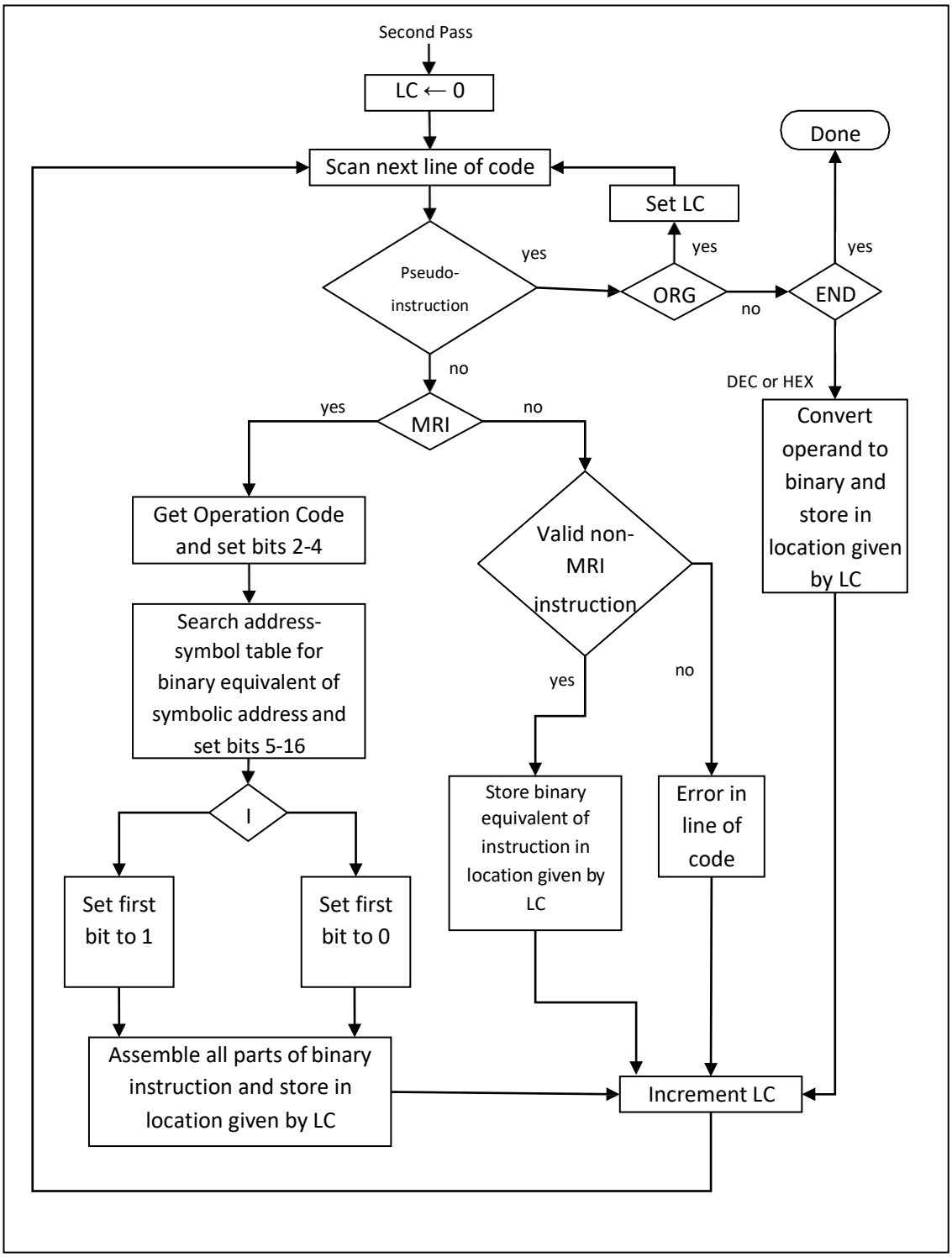


Figure 3.2: Flowchart for second pass of assembler

6. Write short note on subroutine.

- The same piece of code must be written over again in many different parts of a program.
- Instead of repeating the code every time it is needed, there is an advantage if the common instructions are written only once.
- A set of common instructions that can be used in a program many times is called a *subroutine*.

- Each time that a subroutine is used in the main part of the program, a branch is executed to the beginning of the subroutine.
- After the subroutine has been executed, a branch is returned to the main program.
- A subroutine consists of a self-contained sequence of instructions that carries out a given task.
- A branch can be made to the subroutine from any part of the main program.
- This poses the problem of how the subroutine knows which location to return to, since many different locations in the main program may make branches to the same subroutine.
- It is therefore necessary to store the return address somewhere in the computer for the subroutine to know where to return.
- Because branching to a subroutine and returning to the main program is such a common operation, all computers provide special instructions to facilitate subroutine entry and return.
- In the basic computer, the link between the main program and a subroutine is the BSA instruction (branch and save return address).

Example of Subroutine:

```
        ORG 100         /Main program
100     LDA X           /Load X
101     BSA SH4         /Branch to subroutine
102     STA X           /Store shifted number
103     LDA Y           /Load Y
104     BSA SH4         /Branch to subroutine again
105     STA Y           /Store shifted number
106     HLT
107 x,  HEX 1234
108 Y,  HEX 4321
                        /Subroutine to shift left 4 times
109 SH4, HEX 0          /Store return address here
10A     CIL             /Circulate left once
10B     CIL
10C     CIL
10D     CIL             /Circulate left fourth time
10E     AND  MSK        /Set AC(13-16) to zero
10F     BUN  SH4 I      /Return to main program
110 MSK, HEX FFF0       /Mask operand
        END
```

Unit 4

Micro programmed control

1. Important terms:

- **Hardwired Control Unit:**
  When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired.

- **Micro programmed control unit:**
  A control unit whose binary control variables are stored in memory is called a micro programmed control unit.

- **Dynamic microprogramming:**
  A more advanced development known as dynamic microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing.

- **Control Memory:**
  Control Memory is the storage in the microprogrammed control unit to store the microprogram.

- **Writeable Control Memory:**
  Control Storage whose contents can be modified, allow the change in microprogram and Instruction set can be changed or modified is referred as Writeable Control Memory.

- **Control Word:**
  The control variables at any given time can be represented by a control word string of 1 's and 0's called a control word.

- **Microoperations:**

  - In computer central processing units, micro-operations (also known as a micro- ops or μops) are detailed low-level instructions used in some designs to implement complex machine instructions (sometimes termed macro-instructions in this context**).**

- **Micro instruction:**
  - A symbolic microprogram can be translated into its binary equivalent by means of an assembler.
  - Each line of the assembly language microprogram defines a symbolic microinstruction.

- Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD.

- **Micro program:**
  - A sequence of microinstructions constitutes a microprogram.
  - Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read-only memory (ROM).
  - ROM words are made permanent during the hardware production of the unit.
  - The use of a micro program involves placing all control variables in words of ROM for use by the control unit through successive read operations.
  - The content of the word in ROM at a given address specifies a microinstruction.

- **Microcode:**
  - Microinstructions can be saved by employing subroutines that use common sections of microcode.
  - For example, the sequence of micro operations needed to generate the effective address of the operand for an instruction is common to all memory reference instructions.
  - This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

2. Draw and explain the organization of micro programmed control unit.

- The general configuration of a micro-programmed control unit is demonstrated in the block diagram of Figure 4.1.
- The control memory is assumed to be a ROM, within which all control information is permanently stored.

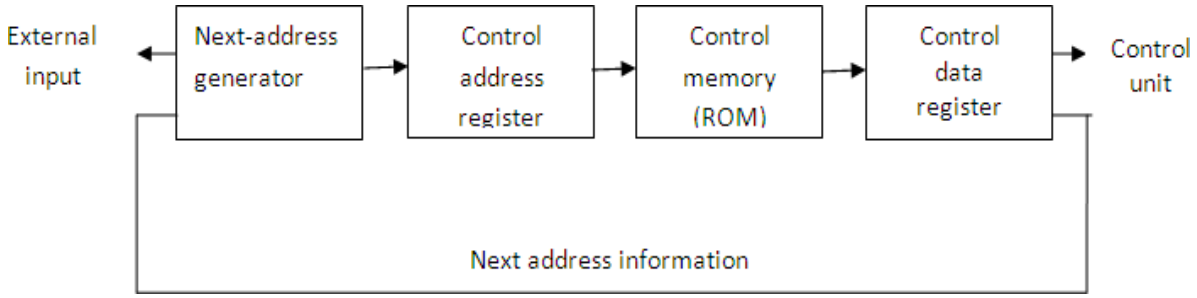


figure 4.1: Micro-programmed control organization

- The control memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory.
- The microinstruction contains a control word that specifies one or more micro-operations for the data processor. Once these operations are executed, the control must determine the next address.
- The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory.
- While the microoperations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next

microinstruction.

- Thus a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.
- The next address generator is sometimes called a ***micro-program sequencer,*** as it determines the address sequence that is read from control memory.
- Typical functions of a micro-program sequencer are incrementing the control address register by one, loading into the control address register an address from control memory, transferring an external address, or loading an initial address to start the control operations.
- The control data register holds the present microinstruction while the next address is computed and read from memory.
- The data register is sometimes called a ***pipeline register****.*
- It allows the execution of the microoperations specified by the control word simultaneously with the generation of the next microinstruction.
- This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register.
- The main advantage of the micro programmed control is the fact that once the hardware configuration is established; there should be no need for further hardware or wiring changes.
- If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstructions for control memory.

3. Explain the steps of Address Sequencing in detail.

- Microinstructions are stored in control memory in groups, with each group specifying a *routine.*
- To appreciate the address sequencing in a micro-program control unit, let us specify the steps that the control must undergo during the execution of a single computer instruction.

Step-1:

- An initial address is loaded into the control address register when power is turned on in the computer.
- This address is usually the address of the first microinstruction that activates the instruction fetch routine.
- The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions.
- At the end of the fetch routine, the instruction is in the instruction register of the computer.

Step-2:

- The control memory next must go through the routine that determines the effective address of the operand.
- A machine instruction may have bits that specify various addressing modes, such as indirect address and index registers.
- The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction.
- When the effective address computation routine is completed, the address of the operand

is available in the memory address register.

Step-3:

- The next step is to generate the microoperations that execute the instruction fetched from memory.
- The microoperation steps to be generated in processor registers depend on the operation code part of the instruction.
- Each instruction has its own micro-program routine stored in a given location of control memory.
- The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a ***mapping*** process.
- A mapping procedure is a rule that transforms the instruction code into a control memory address.

Step-4:

- Once the required routine is reached, the microinstructions that execute the instruction may be sequenced by incrementing the control address register.
- Micro-programs that employ subroutines will require an external register for storing the return address.
- Return addresses cannot be stored in ROM because the unit has no writing capability.
- When the execution of the instruction is completed, control must return to the fetch routine.
- This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine.

In summary, the address sequencing capabilities required in a control memory are:

1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on status bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return.

4. Draw and explain selection of address for control memory.

- Figure 4.2 shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address.
- The microinstruction in control memory contains a set of bits to initiate microoperations in computer registers and other bits to specify the method by which the next address is obtained.
- The diagram shows four different paths from which the control address register (CAR) receives the address.
- The incrementer increments the content of the control address register by one, to select the next microinstruction in sequence.
- Branching is achieved by specifying the branch address in one of the fields of the microinstruction.
- Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition.

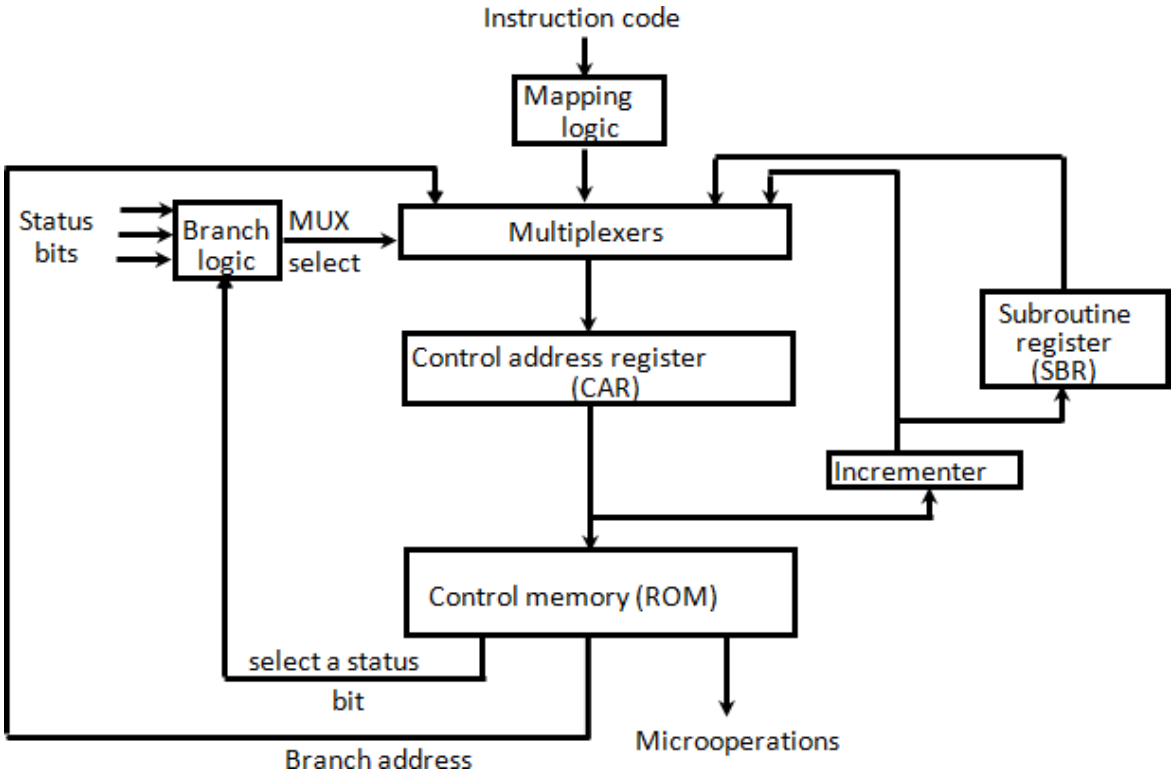


Figure 4.2: Selection of address for control memory

- An external address is transferred into control memory via a mapping logic circuit.
- The return address for a subroutine is stored in a special register whose value is then used when the micro-program wishes to return from the subroutine.
- The branch logic of figure 4.2 provides decision-making capabilities in the control unit.
- The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions.
- The status bits, together with the field in the microinstruction that specifies a branch address, control the conditional branch decisions generated in the branch logic.
- A 1 output in the multiplexer generates a control signal to transfer the branch address from the microinstruction into the control address register.
- A 0 output in the multiplexer causes the address register to be incremented.

5. Explain Mapping of an Instruction

- A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruction is located.
- The status bits for this type of branch are the bits in the operation code part of the instruction.For example, a computer with a simple instruction format as shown in figure 4.3 has an operation code of four bits which can specify up to 16 distinct instructions.
- Assume further that the control memory has 128 words, requiring an address of seven bits.
- One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in figure 4.3.
- This mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register.

- This provides for each computer instruction a microprogram routine with a capacity of four microinstructions.
- If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111. If it uses fewer than four microinstructions, the unused memory locations would be available for other routines.

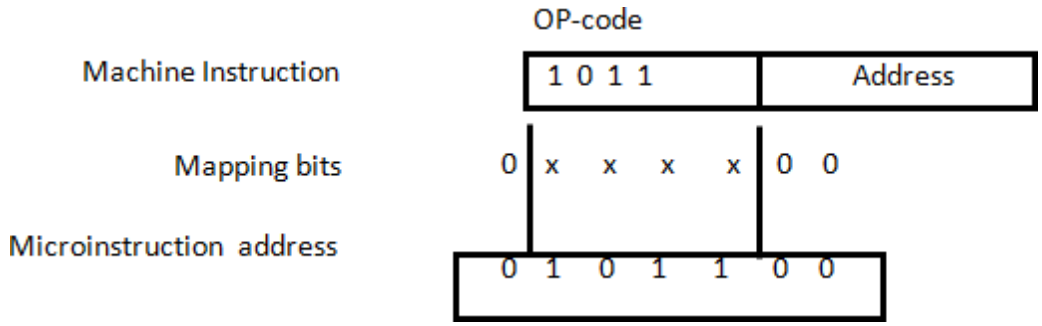


Figure 4.3: Mapping from instruction code to microinstruction address

- One can extend this concept to a more general mapping rule by using a ROM to specify the mapping function.
- The contents of the mapping ROM give the bits for the control address register.
- In this way the microprogram routine that executes the instruction can be placed in any desired location in control memory.
- The mapping concept provides flexibility for adding instructions for control memory as the need arises.

6. Draw and explain Computer Hardware Configuration in detail.

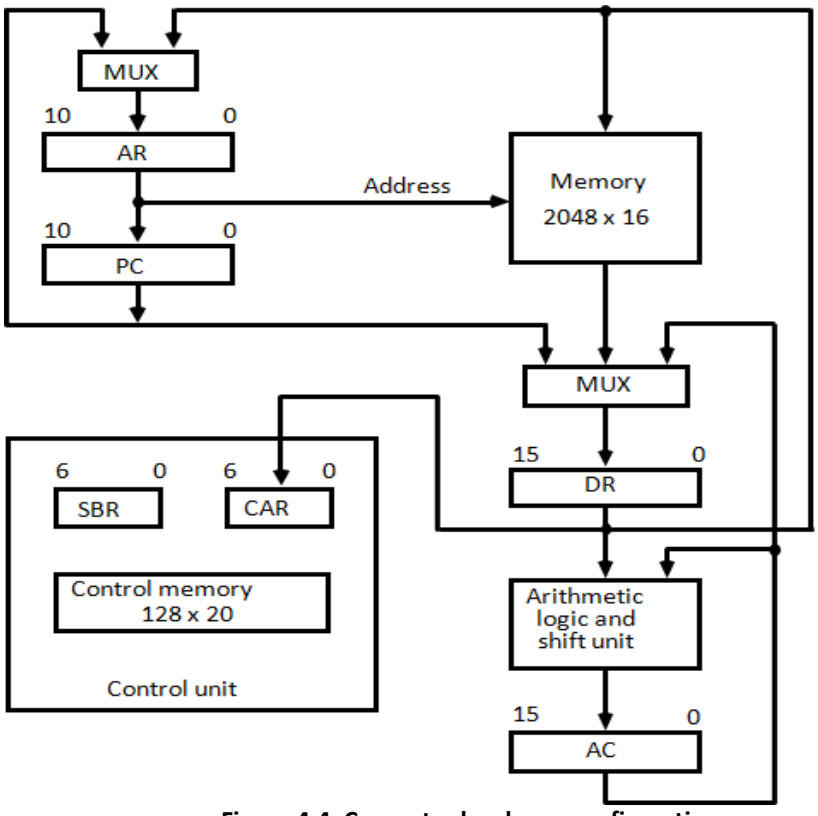


Figure 4.4: Computer hardware configuration

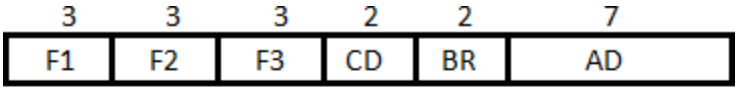
The block diagram of the computer is shown in Figure 4.4. It consists of

1. **Two memory units:**
   Main memory -> for storing instructions and data, and
   Control memory -> for storing the microprogram.
2. **Six Registers:**
   Processor unit register: AC(accumulator),PC(Program Counter), AR(Address Register), DR(Data Register)
   Control unit register: CAR (Control Address Register), SBR(Subroutine Register)
3. **Multiplexers:**
   The transfer of information among the registers in the processor is done through multiplexers rather than a common bus.
4. **ALU:** The arithmetic, logic, and shift unit performs microoperations with data from AC and DR and places the result in AC.
   - DR can receive information from AC, PC, or memory.
   - AR can receive information from PC or DR.
   - PC can receive information only from AR.
   - Input data written to memory come from DR, and data read from memory can go only to DR.

7. Explain Microinstruction Format in detail.

The microinstruction format for the control memory is shown in figure 4.5. The 20 bits of the microinstruction are divided into four functional parts as follows:

- The three fields F1, F2, and F3 specify microoperations for the computer.
- The microoperations are subdivided into three fields of three bits each. The three bits in each field are encoded to specify seven distinct microoperations. This gives a total of 21 microoperations.
- The CD field selects status bit conditions.
- The BR field specifies the type of branch to be used.
- The AD field contains a branch address. The address field is seven bits wide, since the control memory has $128 = 2^7$ words.



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

Figure 4.5: Microinstruction Format

- As an example, a microinstruction can specify two simultaneous microoperations from F2 and F3 and none from F1.

  DR ←M[AR] with F2 = 100
  PC ← PC + 1 with F3 = 101
- The nine bits of the microoperation fields will then be 000 100 101.

- The CD (condition) field consists of two bits which are encoded to specify four status bit conditions as listed in Table 4.1.

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

Table 4.1: Condition Field

- The BR (branch) field consists of two bits. It is used, in conjunction with the address field AD, to choose the address of the next microinstruction shown in Table 4.2.

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

Table 4.2: Branch Field

8. Explain Symbolic Microinstruction.

- Each line of the assembly language microprogram defines a symbolic microinstruction.
- Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD. The fields specify the following Table 4.3.

1.	Label	The label field may be empty or it may specify a symbolic address. A label is terminated with a colon (:).
2.	Microoperations	It consists of one, two, or three symbols, separated by commas, from those defined in Table 5.3. There may be no more than one symbol from each F field. The NOP symbol is used when the microinstruction has no microoperations. This will be translated by the assembler to nine zeros.
3.	CD	The CD field has one of the letters U, I, S, or Z.
4.	BR	The BR field contains one of the four symbols defined in Table 5.2.

5.	AD	The AD field specifies a value for the address field of the microinstruction in one of three possible ways: i. With a symbolic address, this must also appear as a label. ii. With the symbol NEXT to designate the next address in sequence. iii. When the BR field contains a RET or MAP symbol, the AD field is left empty and is converted to seven zeros by the assembler.

Table 4.3: Symbolic Microinstruction

Unit 6

Computer Arithmetic Algorithm

1. Explain the procedure for Addition and Subtraction with signed magnitude data with the help of flowchart.

- The flowchart is shown in Figure 7.1. The two signs A, and B, are compared by an exclusive-OR gate. If the output of the gate is 0 the signs are identical; If it is 1, the signs are different.
- For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added.
- The magnitudes are added with a microoperation EA ←A + B, where EA is a register that combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip-flop AVF.
- The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complemented B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0.
- 1 in E indicates that A >= B and the number in A is the correct result. If this numbs is zero, the sign A must be made positive to avoid a negative zero.
- 0 in E indicates that A < B. For this case it is necessary to take the 2's complement of the value in A. The operation can be done with one microoperation A← A' +1.
- However, we assume that the A register has circuits for microoperations complement and increment, so the 2's complement is obtained from these two microoperations.
- In other paths of the flowchart, the sign of the result is the same as the sign of A. so no change in A is required. However, when A < B, the sign of the result is the complement of the original sign of A. It is then necessary to complement A, to obtain the correct sign.
- The final result is found in register A and its sign in A_S. The value in AVF provides an overflow indication. The final value of E is immaterial.
- Figure 7.2 shows a block diagram of the hardware for implementing the addition and subtraction operations.
- It consists of registers A and B and sign flip-flops A_S and B_S.
- 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 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.

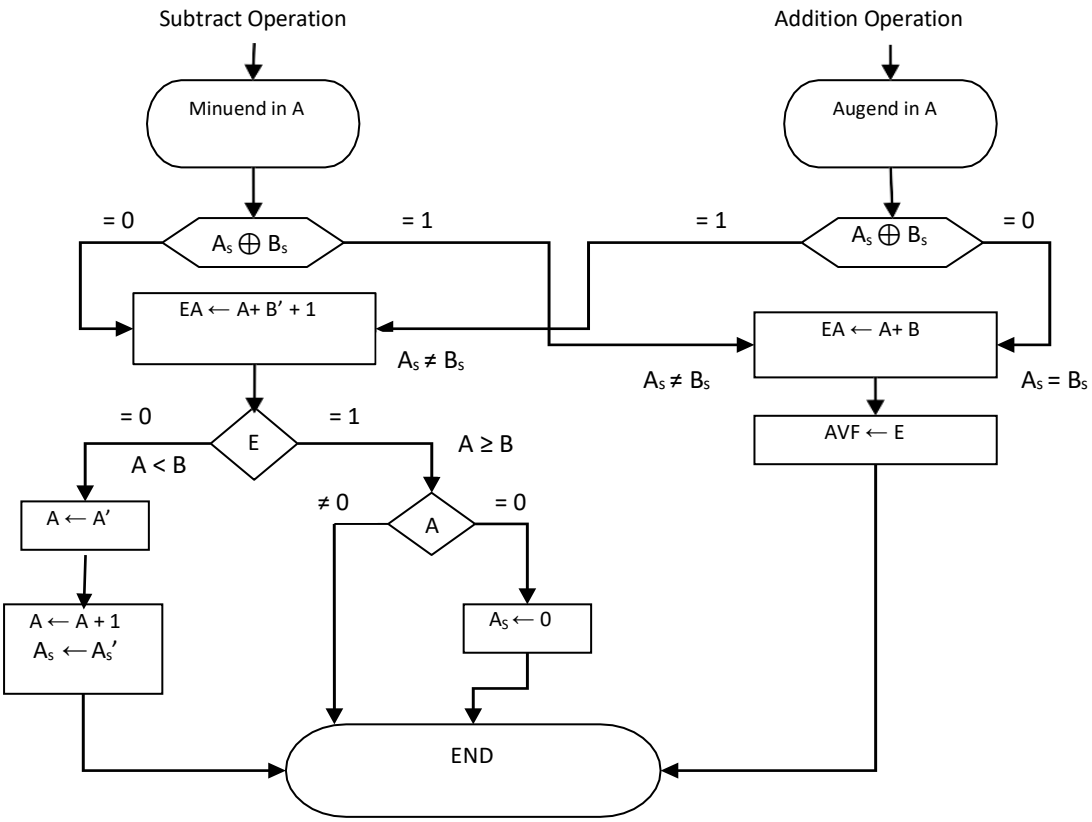


Figure 7.1: Flowchart for add and subtract operations.

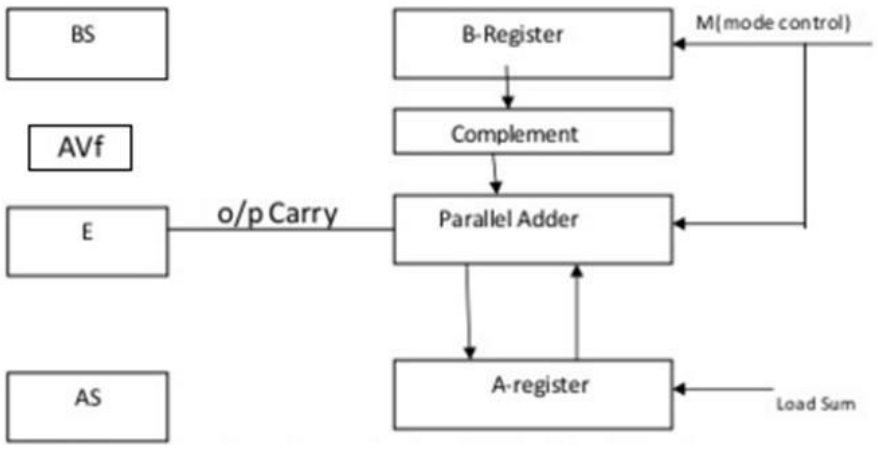


Figure 7.2: Hardware for signed-magnitude addition and subtraction

2. Explain the Booth's algorithm with the help of flowchart.

- Booth algorithm gives a procedure for multiplying binary integers in signed- 2's complement representation.
- It operates on the fact that strings of 0's in the multiplier require no addition but just shifting, and a

string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$.

- For example, the binary number 001110 (+14) has a string 1's from 2^3 to 2^1 (k=3, m=1). The number can be represented as $2^{k+1} - 2^m$. = $2^4 - 2^1$ = 16 – 2 = 14. Therefore, the multiplication M X 14, where M is the multiplicand and 14 the multiplier, can be done as M X 2^4 – M X 2^1.
- Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

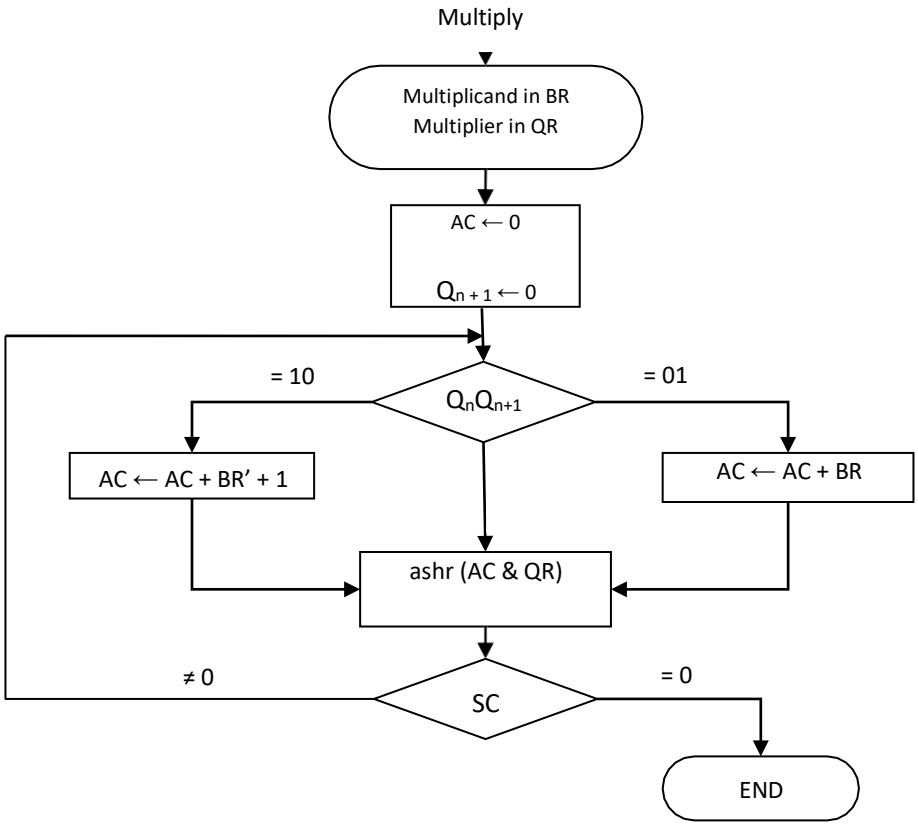


Figure 7.3: Booth algorithm for multiplication of signed-2's complement numbers

- As in all multiplication schemes, booth algorithm requires examination of the multiplier bits and shifting of partial product.
- Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial, or left unchanged according to the following rules:
  - The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.
  - The multiplicand is added to the partial product upon encountering the first 0 in a string of 0's in the multiplier.
  - The partial product does not change when multiplier bit is identical to the previous multiplier bit.
- The algorithm works for positive or negative multipliers in 2's complement representation.
- This is because a negative multiplier ends with a string of 1's and the last operation will be a subtraction of the appropriate weight.
- The two bits of the multiplier in Q_n and Q_{n+1} are inspected.
- If the two bits are equal to 10, it means that the first 1 in a string of 1 's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC.
- If the two bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC.

- When the two bits are equal, the partial product does not change.

3. Explain with proper block diagram the Multiplication Operation on two floating point numbers.

- The multiplication of two floating-point numbers requires that we multiply the mantissas and add the exponents.
- No comparison of exponents or alignment of mantissas is necessary.
- The multiplication of the mantissas is performed in the same way as in fixed-point to provide a double-precision product.
- The double-precision answer is used in fixed-point numbers to increase the accuracy of the product.
- In floating-point, the range of a single-precision mantissa combined with the exponent is usually accurate enough so that only single-precision numbers are maintained.
- Thus the half most significant bits of the mantissa product and the exponent will be taken together to form a single-precision floating-point product.
- The multiplication algorithm can be subdivided into four parts:
    1. Check for zeros.
    2. Add the exponents.
    3. Multiply the mantissas.
    4. Normalize the product.
- The flowchart for floating-point multiplication is shown in Figure 7.4. The two operands are checked to determine if they contain a zero.
- If either operand is equal to zero, the product in the AC is set to zero and the operation is terminated.
- If neither of the operands is equal to zero, the process continues with the exponent addition.
- The exponent of the multiplier is in q and the adder is between exponents a and b.
- It is necessary to transfer the exponents from q to a, add the two exponents, and transfer the sum into a.
- Since both exponents are biased by the addition of a constant, the exponent sum will have double this bias.
- The correct biased exponent for the product is obtained by subtracting the bias number from the sum.
- The multiplication of the mantissas is done as in the fixed-point case with the product residing in A and Q.
- Overflow cannot occur during multiplication, so there is no need to check for it.
- The product may have an underflow, so the most significant bit in A is checked. If it is a 1, the product is already normalized.
- If it is a 0, the mantissa in AQ is shifted left and the exponent decremented.
- Note that only one normalization shift is necessary. The multiplier and multiplicand were originally normalized and contained fractions. The smallest normalized operand is 0.1, so the smallest possible product is 0.01.
- Therefore, only one leading zero may occur.
- Although the low-order half of the mantissa is in Q, we do not use it for the floating- point product. Only the value in the AC is taken as the product.

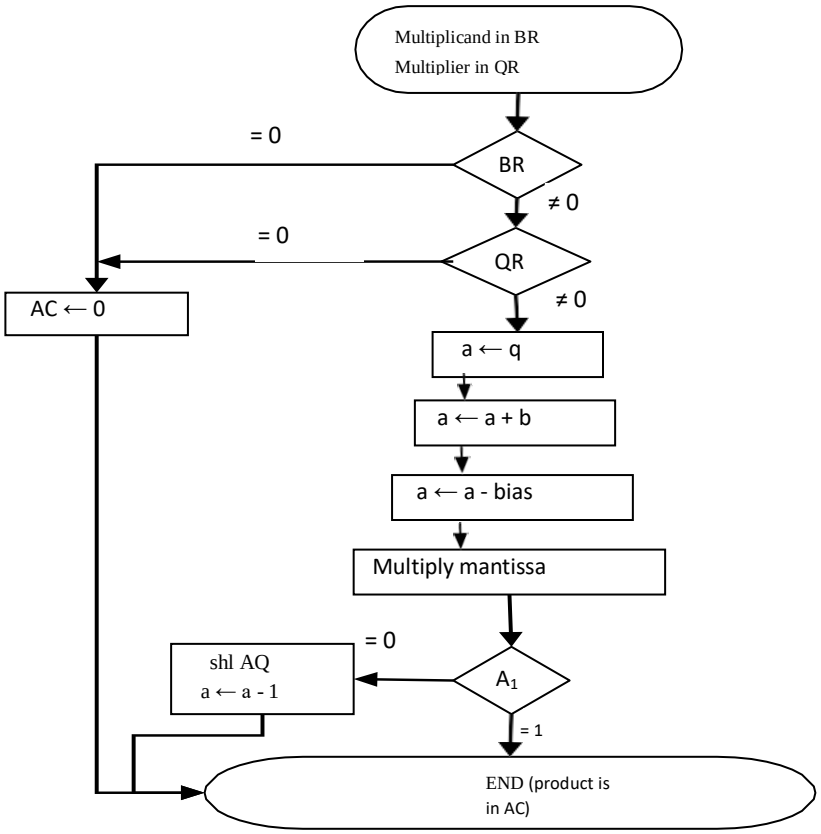


Figure 7.4: Multiplication of floating-point numbers

Example: Multiply the (-9) with (-13) using Booth's algorithm. Give each step.

- A numerical example of booth algorithm is shown for n=5. It shows the step-by-step multiplication of (-9) X (-13) = +117.

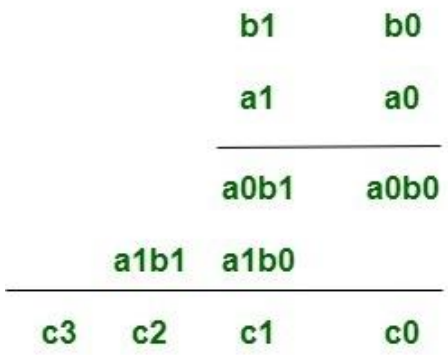
9:	01001	13:	01101
1's complement of 9:	10110	1's complement of 13:	10010
	+ 1		+ 1
2's complement of 9:	10111 (-9)	2's complement of 13:	10011 (-13)

AC	QR(-13)	Q_{n+1}	M(BR)(-9)	SC	Comments
00000	1001<u>1</u>	<u>0</u>	10111	5	Initial value
01001	10011	0	10111	4	Subtraction: AC=AC+BR'+1
00100	1100<u>1</u>	<u>1</u>	10111		Arithmetic Shift Right
00010	0110<u>0</u>	<u>1</u>	10111	3	Arithmetic Shift Right
11001	01100	1	10111	2	Subtraction: AC=AC+BR'+1
11100	1011<u>0</u>	<u>0</u>	10111		Arithmetic Shift Right
11110	0101<u>1</u>	<u>0</u>	10111	1	Arithmetic Shift Right
00111	01011	0	10111	0	Subtraction: AC=AC+BR'+1
00011	**10101**	1	10111		Arithmetic Shift Right

Answer: -9 X -13 =117 => 001110101

4. Array Multiplier

- An **array multiplier** is a digital combinational circuit used for multiplying two binary numbers by employing an array of full adders and half adders.
- This array is used for the nearly simultaneous addition of the various product terms involved.
- To form the various product terms, an array of AND gates is used before the Adder array.
- Checking the bits of the multiplier one at a time and forming partial products is a sequential operation that requires a sequence of add and shift micro-operations.
- The multiplication of two binary numbers can be done with one micro-operation by means of a combinational circuit that forms the product bits all at once.
- This is a fast way of multiplying two numbers 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 for this reason it was not economical until the development of integrated circuits.
- For implementation of array multiplier with a combinational circuit, consider the multiplication of two 2-bit numbers as shown in figure. The multiplicand bits are b1 and b0, the multiplier bits are a1 and a0, and the product is



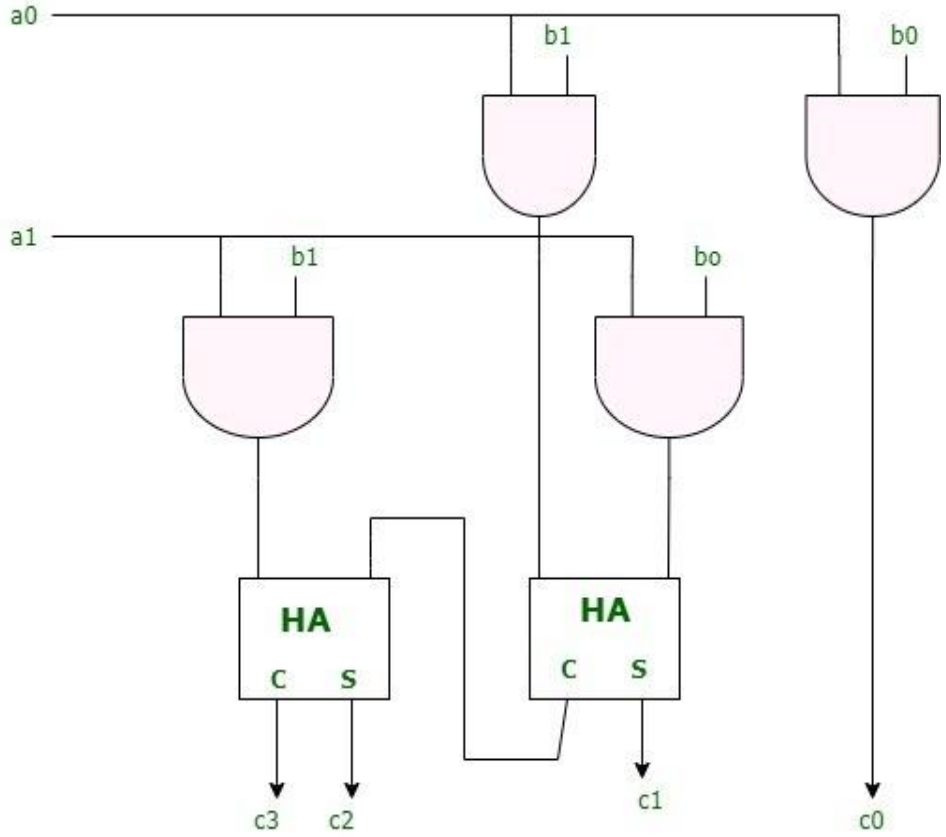
- Assuming A = a1a0 and B= b1b0, the various bits of the final product term P can be written as:-
  1. P(0)= a0b0
  2. P(1)=a1b0 + b1a0
  3. P(2) = a1b1 + c1 where c1 is the carry generated during the addition for the P(1) term.
  4. P(3) = c2 where c2 is the carry generated during the addition for the P(2) term.

- For the above multiplication, an array of four AND gates is required to form the various product terms like a0b0 etc. and then an adder array is required to calculate the sums involving the various product terms and carry combinations mentioned in the above equations in order to get the final Product bits.

- The first partial product is formed by multiplying a0 by b1, b0. The multiplication of two bits such as a0 and b0 produces a 1 if both bits are 1; otherwise, it produces 0. This is identical to an AND operation and can be implemented with an AND gate.

  1. The first partial product is formed by means of two AND gates.
  2. The second partial product is formed by multiplying a1 by b1b0 and is shifted one position to the left.

3. The above two partial products are added with two half-adder(HA) circuits. Usually there are more bits in the partial products and it will be necessary to use full-adders to produce the sum.
4. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.



A combinational circuit binary multiplier with more bits can be constructed in similar fashion.

Excersise:

1. Multiply the (7) with (3) using Booth's algorithm
2. Multiply the (+15) with (-13) using Booth's algorithm
3. Draw the block diagram for 4-bit by 3-bit array multiplier.