ASSEMBLY LANGUAGE PROGRAMMING

USING

MC68000

Arulogun O. T Fakolujo O. A.
Omidiora E. O. Ajayi A. O.

All rights reserved

Published in May, 2005

ISBN: 978-066-755-5

Printed by:

DAPSON PRINTS

TEL: 01-582145, 08023708279.

CONTENTS

ACKNOWLEDGMENT

The authors would like to acknowledge the following people for helping in getting the work done, Williams Akinwumi for putting things in order, Olaniyi Mikhail for his invaluable encouragement, contribution and for his initial typing of the manuscript. To all our present and past students especially Kamal, Debo, Mayowa and Omorosi for their help in fixing occasional break downs of the computer system.

We also acknowledge our in-house pals, Biodun, Abbey and Babygirl_Bruce, for keeping the home front lively for writing; you are one in a million.

Lastly, precious thanks to our tutors and senior colleagues especially, Dr. Fakolujo, Dr. Olatunbosun, Mr. Okeyinka for giving us the chance and propelling us to achieve.

CHAPTER ONE

INTRODUCTION TO

MICROCOMPUTER PROGRAMING LANGUAGES

To communicate effectively between two people there must be some set of rules, conventions, symbols, and grammars agreed upon. This set is known as language. A language (system of rules, words) for communicating with computer is called computer language. It is designed to give precise information to a computer. Any task can be accomplished by taking a sequence of actions. The way to describe how to accomplish a task is to give the instructions for the sequence of actions needed to complete it. Computer identifies a task when it is given step-by-step instructions for the successive actions, which result in the completion of the task. Computer program is a sequence of instructions written in a defined computer language given to a computer to have a problem solved.

Microcomputer can be programmed using binary or hexadecimal number (Machine language), Semi-English language statements (Assembly language or a more understandable human-Oriented language called high-level language). Regardless of what type of language that is used to write the program, it has to be converted into appropriate binary form, because Microcomputer understands only binary numbers.

Microcomputer programming languages can be classified into three main groups, namely.

1. Machine language.
2. Assembly language.
3. High level language

A machine language program consists of either binary or hexadecimal OP (operation) codes that specify actions to be performed on data elements. Programming a microcomputer with either binary or hexadecimal op-codes is relatively difficult and prone to error, which are also difficult to debug, since one will deal only with numbers. The architecture of a microprocessor determines all its instructions. These instructions are known as the microprocessor's instruction set. Programs in assembly and high-level language are represented by instructions like natural- language-type statements. The programmer finds it relatively more convenient to write programs in assembly or high-level language than in machine language.

Since the microcomputer understands only binary numbers, a translator must be used to convert the assembly or high-level language programs into binary machine language so that the microcomputer can execute the programs. Figure 1.1 shows the translation the process of translation.

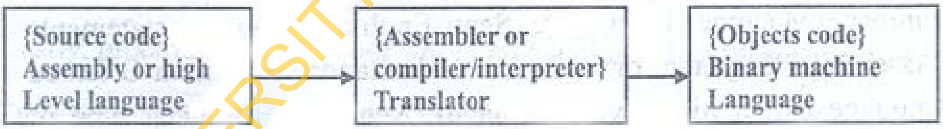


Figure 1.1: Translation of Assembly / high-level language to machine language

An assembler translates a program written in assembly language (source code) into a machine language program (object code). A compiler or interpreter converts a high-level language program written in PASCAL, FORTRAN, BASIC, C++ etc. into a machine language program.

3E
10
C6
20
21
00
02
76

It is easier to detect error in hexadecimal machine language program than in binary machine language program since each byte contains only two hexadecimal digits.

ASSEMBLY LANGUAGE

Assembly language is a mnemonic representation of a natural or native language of a computer called machine code. Its instructions are called macroinstruction.

Each macroinstruction is interpreted by means of primitive actions called microinstructions. A microinstruction is the smallest event that can take place within a computer. It may be moving a data bit from one register to another consisting of clocking a flip-flop. The process of carrying out a series of microinstruction is called macroinstructions. Process of executing a macroinstruction is called interpretation.

Each instruction in an assembly language program is composed of three or four field as follows.

1. Label field
2. Mnemonic field (Op-code)
3. Operand field

1. Comment field (optional).

They are organized or arrange as follows:

Label Field	Mnemonic Field	Operand field	Comment Field

Assembly language equivalent to the machine language program treated earlier that adds two 8-bit numbers is given below.

```
LABEL  MNEMONIC  OPERAND    COMMENT
START  MVI       A, 10H     ; move 10  into Accumulator
       ADI       20H        ;Add 20 to the contents of Accumulator
       LXI       H, 0200H   ; Load register Pair HL with 0200H
       MOV       M, A       ; move the contents of Accumulator into
                            locations 0200H
       HLT                  ; stop temporarily until Interrupt.
```

Obviously, programming in assembly language is more convenient than using machine language, since each mnemonic gives an idea of the type of operation it is supposed to perform. Therefore, with the assembly language, the programmer does not have to find the numerical Op-codes from the table of instruction set.

HIGH LEVEL LANGUAGE

Though, the programmer's efficiency increases while programming with assembly language programmer; the programmer however suffers the following anomalies.

i. Programmer must be well acquainted with the microprocessor's architecture and instructions set.

ii. He must provide an OP-code for each operation that the microprocessor has to carry out in order to execute a programming task.

iii. Complexity entails when writing many steps in large program.

In order to solve the above problems; a microcomputer can however be programmed by using **High-Level Language Programs**. This will rob the programmer of direct control of the hardware and uses more memory than other computer languages. High-level language program comprises of English type statements designed to rectify the above highlighted problems in assembly and machine language programming. It enjoys the following characteristics:

i. The programmer neither needs to be familiar with microprocessor internal architecture nor its instruction set. High-level language is problem oriented.

ii. Each statement is equivalent to a number of assembly or machine language instructions. Examples of High Level Languages are C, FORTRAN, C++, Delphi, Pascal, and COBOL.

Like assembly language, it requires a special program for converting the high-level language statements into machine object codes. The program can be either an **interpreter** or a **compiler.**

CHAPTER TWO
ASSEMBLY LANGUAGE PROGRAMMING CONCEPTS

An instruction manipulates stored data and a sequence of instruction makes up a program. In general, an assembly language instruction has two components namely:

1. ***Operation code (OP code) field***
2. ***Operand or Address of operand field***

The ***OP-code*** field specifies how data is to be manipulated. A data item may reside within a microprocessor register or in main memory. Thus, the purpose of the ***Address field*** is to indicate the location of a data item. For an example, consider the assembly language instruction below.

ADD	**RI, RO**
Op-code	Address field

Assume that the microcomputer that this instruction is meant for uses **RI** as the source register and **R0** as destination register. The Op-code **ADD** part, of the instruction means arithmetic addition operation. Therefore the instruction will add the contents of microprocessor register **RO** to **RI** and save the sum in register **RO**.

i.e. Destination register $\longleftarrow$ *Source register + Destination register*

The number and types of instructions supported by microprocessor may vary from one microprocessor to another and primarily depends on the architecture of a particular machine.

INSTRUCTION FORMATS

The following instruction formats are identifiable in assembly language based upon the number of addresses specified in the instruction.

1. *Zero-address instruction format*
2. *One-address instruction format*
3. *Two-address instruction format*
4. *Three-address instruction format*

1. **Zero-address instruction format:** An instruction that does not require any address is called a zero-address instruction format. Examples are ***STC (set carry flag), NOP (no operation), RAL (rotate accumulator left), RET (return from exception).***

2. **One-address instruction format:** An instruction with a single address is called a One-address instruction format. It takes the following format:

<Op-code> Address1.

e.g. ADD B ;Accumulator (ACC) $\longleftarrow$ ACC + register B
SUB C ;Accumulator (ACC) $\longleftarrow$ ACC - register C
CLR D1 ; D1 $\longleftarrow$ 0

3. **Two address instruction format:** An instruction containing two addresses is called two-address instruction. It takes the following format:

<Op-code>Addr1, Addr2

e.g. MOVE R2, R1 ; R1 $\longleftarrow$ R2
ADD R1, R2 ; R2 $\longleftarrow$ R2+R1
SUB R1, R2 ; R2 $\longleftarrow$ R2 - R1

4. **Three address instruction format:** An instruction with three addresses is called ~~one~~ Three-address instruction. It takes the following format:

<Op-code>Addr1,Addr2,Addr3

e.g. MULA,B,C ; C $\longleftarrow$ A*B
ADDA,B,C ; C $\longleftarrow$ A+B
SUB R1,R2,R3; R3 R1-R2

NOTE: ***The result of an operation is always saved in the destination address of the operand. All the above formats assume the last part of the address/operand to be the destination. It could be the reverse in some other microprocessors. Letters A, B, C, R0, R1, R2 and R3 are designations for microprocessor internal registers.***

ADDRESSING MODES

The sequence of operations that a microprocessor has to carry out while executing an instruction is called its *instruction cycle*. The most important activity in an instruction cycle is the determination of the operand and destination addresses. The manner in which a microprocessor accomplishes this task is known as the ***addressing mode.*** In other words, addressing mode is the manner in which the microprocessor calculates the addresses of source and destination registers.

There are many variations of addressing modes, the basic ones are stated below:

(i) **Inherent/Implied Addressing Modes:** The instructions using this mode have no operands. Examples: STC, CMA (Complement Accumulator), NOP (no operation), RET (return from exception).

(Ii) **Immediate Addressing Mode:** Whenever an instruction contains the operand value, it is called an immediate addressing mode. The actual data to be manipulated by the Op-code is specified as part of the instruction. The symbol # usually indicates that an instruction is in an immediate mode.

e.g. ADD #20, R0	;R0 $\longleftarrow$ R0+20	(MC68000)
MOV AX, 05	;AX $\longleftarrow$ 05	(Intel 8086)
ADI 05	;Acc $\longleftarrow$ Acc +05	(Intel 8085)

(iii) **Direct Addressing Mode**: Instructions using this mode specify the effective address of the operand as part of the instruction. Instructions using this mode contain three bytes, the first byte is the Op-code followed by two bytes that represent the address of data.

e.g. MOVE 2000, RO ; R0 $\longleftarrow$ M(2000): content of memory

location 2000 is transferred into R0

LDA 2035H ; A $\longleftarrow$ M (2035)

(iv) **Register Addressing Mode:** This mode specifies the register or register pair that contains data i.e. operand values are held in microprocessor registers.

e.g. ADD R1, R0 ; R0 $\longleftarrow$ R1+R0

MOVE R5, R6 ;R6 $\longleftarrow$ R5

(v) **Register Addressing Indirect Mode:** Whenever an instruction specifies a microprocessor register that holds the address of an operands, the resulting addressing mode is known as the register indirect mode. The address of the operand is specified indirectly in a register.

e.g. MOVE (R1),(R0) ;M((RO)) $\longleftarrow$ ((R1))M

In the above instruction, if the content of R1=3000, RO = 4000, M(3000) = 0554

M(4000) = 0548, then the instruction copies the content of the memory location whose address is specified in the microprocessor register R1 into the location specified into register R0. In other words, memory location 4000 will contain 0554 after the execution of above instruction,

$$LDAX\,B; A \longleftarrow M((BC))$$

This Intel 8085 instruction loads the accumulator A with the contents of a memory location addressed by the B, C register pair. BC register pair contains the address.

(vi) **Memory Indirect Addressing Mode:** Sometimes, it is possible that an instruction may contain the address of an address of an operand. In such a case, the addressing mode is referred to as a memory indirect addressing mode.

e.g. MOVE (2000), R0: R0 $\longleftarrow$ M(M(2000))

The content of memory location 2000 will be interpreted as the address of the operand to be copied into register R0.

(vii) Address Mode Based On Address Modification: In the context of address modification, effective address (EA) of an operand is expressed as a sum of two parameters known as modifier M or the offset or displacement and the reference address RA. i.e. EA = RA + M. There are four addressing modes based on this concept:

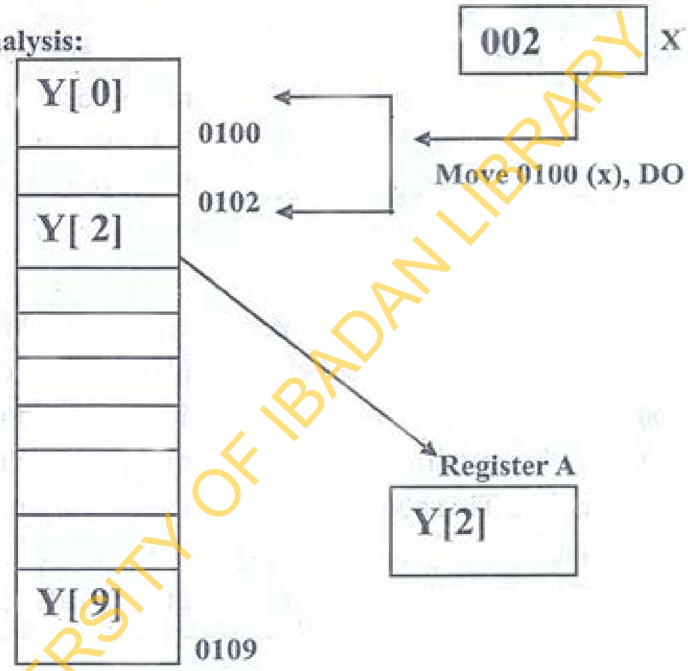
(i) **Indexed Register Mode:** This is used in accessing structured data type like an array. In this mode RA is included in the instruction and register X contains the value M i.e. modifier. The register X is called the index register. For example, consider the declaration in a typical Pascal program,

Var Y: ARRAY [0..9] Of integer;

and that element Y[2] is to be accessed. Let assume that variable Y is stored in memory location starting at $0100.

Load/move instruction such as ***MOVE 0100(X), D0*** could be used to accomplish this task

Analysis:



If index register X contain value 0002, then effective address EA, is given as EA=RA+(X) =0100+2

=0102

Therefore, the contents of memory location 0102 will be transferred to the A register, which contains array element Y[2]. Other elements of the array could be accessed in the same manner by changing the register X. For multidimensional array more than two parameters will be used to calculate the effective address of the operand. This mode allows a programmer to carry out array manipulations efficiently.

(ii) **Base Register Mode:**

In the base register mode, the parameter RA is held in a separate register called the Base register and the offset or the modifier is included in the instruction as offset. This mode is useful in microprocessor that utilizes segmented memory system.

EA = Base Register + offset

(iii) **BaseIndex Register Mode**

In the base-index register mode, the parameter RA is held in a separate register called the Base register which could be a data register and another parameter known as index of the address stored in an address register while the modifier or displacement is included in the instruction as offset. This could be used to access a multi-dimensional data structure such as array, records.

EA = Base Register + index register + offset

(iv) **Relative Addressing Mode**

In this mode, the program counter (PC) is configured as the base register. This mode is used in writing position independent code so that program could be located anywhere in memory. The address of the operand is relative to the content of the program counter by a specified 16 bit offset.

EA=PC + offset

PROGRAM CONTROL INSTRUCTION

In a conventional microcomputer, instructions are always executed in the same order (sequence) in which they are presented to the computer, irrespective of the programming language being employed. In many real-life programs, the flow of control depends on the result of computation. In this situation, a program can select a particular sequence of instructions to execute based on the results of computation. In assembly language programming the Instructions that could be used to realize this idea are called ***program control instructions***. This group of instructions alters normal sequential flow of program. These instructions can be classified into the following groups:

(a) **Unconditional Branch Instruction**
(b) **Conditional Branch Instruction**
(c) **Subroutine Call and Return Instruction**

(a) ***Unconditional Branch Instruction*****:** This transfer the control to the specified address regardless of the status of the preceding program computation. The program control is transferred automatically without any condition testing.

e.g. ***JMP address, BRA address***

(b) **Conditional Branch Instructions**: Before program control could be transferred to another segment of the main program, condition(s) must be tested. If condition is satisfied, the transfer of control is effected, otherwise the normal program sequence is continued. A conditional branch instruction works as follows:

```
If (condition) then
        branch to execute a new instruction
        else
        execute the next instruction.
```

The status of a microprocessor after executing an instruction is always reflected in the microprocessor's ***status register*** (SR). The results of condition testing can be obtained from the status register. The contents of the status register are interpreted as individual bit, with each bit (flag) representing a condition. The status register will be explained in detail under microprocessor's *programming model*. We assume that the status flags are already ***set or reset*** by an instruction that immediately precedes the conditional branch instruction except ***MOVE instructions***. Using the status flag value, we can realize traditional relational operators such as equal to, not equal to, greater than, greater than or equal to ,less than , less than or equal to and so on. These relational operators are used for testing conditions so as to know if transfer of program control will take place.

For example, consider the following MC68000 instruction sequence.

```
          MOVE#25, D0; D0 ←— 25
          MOVE#15, D1 ; D1←— 05
AGAIN; SUB    D1, D0   ; D0=D0-D1
       BEQ    END      ; Jump to address END, if zero Flag is set.
       SUB    #05, D1 ; otherwise, do the following sequence of
                         instruction
       JMP    AGAIN ; Repeat the subtraction ( unconditional
                         jump)
END:   HALT            ; Stop
```

In this example, let start from the instruction ***BEQ END,*** which means branch to label ***END,*** if the result of the previous subtraction instruction ***SUB D1, D0*** is zero (i.e. Z flag =1). If the result of subtraction is not equal to zero (i.e. Z flag =0), then execute the next instructions ***ADD #5, D1,*** followed by ***JMP AGAIN*** and ***END: HALT.*** The instruction JMP AGAIN is an unconditional jump to address AGAIN. Other instructions that could be used for conditional transfer of program control are: *BNE, BGE, BLE, BGT, BLT, BMI, BPL, BLS, BHI, BCS, and BCC.*

(C). **Subroutine Call and Return Instruction:** A subroutine is a name that is given to a group of program instructions that collectively perform a single function. A subroutine can be called to perform repeatedly needed tasks such as searching, sorting, binary to ASCII code conversion, square root, etc. Subroutine may be written, assembled and tested separately. It replaces several lines of code and optimizes the (Main) calling program.

Subroutine forms a key word in modern software approach called modular programming. In modular programming, large program can be thought of as collection of independent program modules called subroutine or set of subroutines. Modular programming encourages problem sharing; where a large program can be efficiently developed within a short period of time.

Assembly language programming provides means of transferring program control to and from subroutine within a program. The means of transferring program control to or from Subroutines are handled by two special instructions, namely ***CALL*** and ***RETURN*** respectively.

The CALL instruction is of the form *CALL Address,* where the parameter *address* refers to the address of the first instruction in the subroutine. When this instruction is to be executed, the current contents of program counter (PC) are saved onto the stack (memory) and PC is loaded with the *Address* of the subroutine. Program control is now in the subroutine program. After the execution of the subroutine, program control must be transferred back to the calling program. This is achieved by restoring the previous contents of the PC from the stack. The ***RETURN*** instruction will perform this operation of restoring the previous contents of PC back. Therefore the ***RETURN*** instruction should be the last instruction of the subroutine. It is of note that the previous contents of the PC pushed onto the stack provide the address of the instruction that immediately follows the *CALL* instruction. This address is also known as the return address because this is the point where after exiting the subroutine, the control is transferred back to the calling program by the ***RETURN*** instruction.

The *CALL* instruction is similar to Intel microprocessors'

```
        PUSH PC         ; save return address on stack
        JMP Address     ; Branch to subroutine Address
```

The ***RETURN*** instruction is equivalent to

```
        POP PC
```

The CALL and RETURN instructions have conditional and unconditional variants. These variants will be discussed later in the next chapter. Subroutine call and return instructions are similar to conditional and unconditional branch instructions except that program control must be transferred back to the calling program where the subroutine was initiated.

INPUT/OUTPUT INSTRUCTIONS

These instructions allow a microprocessor to perform input/output operations through a configured input/output port. An input instruction allows a peripheral device to transfer a word to either a register or the main memory and similarly, an output instruction enables a microprocessor to transfer a word unto the buffer register of a peripheral device. For example, we have *IN* and *OUT* instructions for 8085 for input and output respectively. Z80 provides an enhancement to the input/output instructions, of the 8885 by providing instructions such as ***INTR***, ***OTIR*** that can transfer a block of words from an input device to the main memory and from the main memory to an output device, respectively. MC68000 has no explicit input/output instruction but treats peripheral device, as if they are memory location that can be written into or read from using the *MOVE* instructions.

INPUT/ OUTPUT OPERATION

This can be defined as the transfer of data between the microcomputer system and the external devices: There are three types of input output operations common to microcomputer system. They are:

Programmed Input/0utput (Polling I/O)

With this technique, the microprocessor executes a program to perform all data transfer between the microcomputer system and the external devices via one or more register called input/output port. The main characteristic of this type of input/output technique is that the external devices carry out the functions as dictated by the program inside the microcomputer memory. In other words, the microprocessor completely controls all the data transfer between the peripheral and the microprocessor.

Interrupt Input/Output

In this technique an external device can force the microcomputer system to stop executing the current program temporarily so that it can execute another program known as the interrupt service routine. This routine satisfies the needs of the external device. After having completed this interrupt service routine, the microprocessor returns to the program that it was executing before the interrupt, the microprocessor completely controls all the data transfer. This is also known as hardware initiated subroutine.

Direct Memory Access (DMA)

In this type of input/output technique, data can be transferred between the microcomputer memory and external devices without any direct microprocessor's involvement. DMA is typically used to transfer blocks of data between microcomputer memory and external devices such as hard and floppy disk drives. An interface chip called the DMA controller chip is used by the microprocessor for transferring data via DMA.

CHAPTER THREE

INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING

The most primitive language in which programs are written in native or host language of a computer is called assembly language. It uses mnemonic to represent various operations performed by the computer. Mnemonics are self-evident symbolic name that refers to an operation.

e.g. ***ADD*** denotes addition operation
SUB denotes subtraction operation
BRA denotes branching operation
MOVE denotes copy operation

Consider a typical assembly language instruction given below

SUB.B Number, D3

The instruction above means subtract 8-bit number stored in memory location named ***Number*** from the contents of a data register ***D3***. The ***.B*** following mnemonic indicates size of source data that the instruction will work on i.e. 8-bits. The data register ***D3*** referred to in the instruction is a special purpose data storage element within the microprocessor and ***Number*** refers to memory location of the source operand.

Programs in assembly language are represented by instructions that use English languagetype commands. The programmer finds it relatively more convenient to write the programs in assembly language than in machine language. However, a translator called the ***Assembler*** must be used to convert the assembly language programs into binary machine language programs (objects codes) so that the microprocessor can execute them.

An assembler is a program that translates symbolically written programs in assembly language into machine language. There are various types of assemblers available today. Some of them are described below.

One-pass Assembler:

This assembler scans through the assembly language program once and translates the assembly language program. This assembler has the problem of defining forward references. This means that a ***JUMP*** instruction using an address that appears later in the program must be define the programmer after the program is assembled.

Two-pass Assembler:

This assembler scans the assembly language program twice. In the first pass, the assembler creates a symbol table. A symbol table consists of labels with addresses assigned to them . This way labels can be used for JUMP statements, and no address calculation has to be done by the user. ON the second pass, the assembler translates the assembly language program into the machine code. The two-pass assembler is more desirable and much easier to use because of ease of address computation.

Macro assembler:

This type of assembler translates a program written in macro language into the machine language. This assembler allows the programmer to define all instruction sequences using macros. By using macros, the programmer can assign a name to an instruction sequence that appears repeatedly in a program.

The programmer canthus avoid writing an instruction sequence that is required many times in a program by using macros. The macro-assembler replaces a macro name with the appropriate instruction sequence each time it encounter a macro name. The difference between a subroutine and a macro is that in the former, a specific subroutine occurs once in a program. A subroutine is executed by **CALL**ing it from a main program. The program execution jumps out of the main program and then executes the subroutine. At the end of the subroutine, a **RET**urn instruction is used to resume program execution following the CALL SUBROUTINE instruction in the main program. A MACRO, on the other hand, *does not cause the program execution to branch out of the main program*. Each time a macro occurs, it is replaced with the appropriate instruction sequence in the main program.

Resident Assembler:

This type of assembler assembles programs for a microprocessor in which it is resident. The resident assembler may slow down the operation of the processor on which it runs.

Cross Assembler:

This type of assembler is typically resident on the microprocessor and assembles programs for another microprocessor for which it is written. The cross assembler program is written in a high level language so that it can run on different types of processors that understand the same high-level language.

Meta-Assembler:

This type of assembler can assemble programs for many different types of microprocessor. The programmer usually defines the particular processor being used.

ASSEMBLER SYNTAX

As mentioned earlier in chapter one, each line of an assembly language program consists of four fields. These fields are:

1. **Label field.**
2. **Mnemonic / Op-code field.**
3. **Operand field.**
4. **Comment field.**

The assembler ignores the comment field but translates the other three fields. The label field must start with an upper case alpha character. The assembler must know where one fields start and another ends. Most assemblers allow the programmer to use a special symbols or delimiter to indicate the beginning or end of each field. Typical delimiters used are spaces, commas, and semi-colons: and colons.

Space	between each field
,	between operands
;	before a comments
:	or none after a label.

In order to handle numbers, most assemblers consider all numbers as decimal numbers unless specified. Most assemblers will also allow some ways. Using a letter following the number usually does this. Typical letter used are:

B	for binary
Q	for octal
H or $	for hexadecimal.

Assemblers generally require hexadecimal numbers to start with a digit. A **0** (zero) is typically used if the first digit is an alphabet to distinguish between hexadecimal numbers and label. For example, most assemblers will require the number A5H to be represented as 0A5H. Therefore, the 8085 assembler will not accept the 8085 instruction MVIA, FFH and will give error. The correct format for this instruction is MVIA, 0FFH.

ASSEMBLER DIRECTIVE

These pseudo-instructions are not directly translated into machine language instruction. Assembler uses pseudo-instructions or directives to make the formatting of the edited text easier. They equate labels to addresses, assign the program in certain areas of memory, or insert titles, page numbers etc.

In order to used the assembler directive or pseudo-instruction the programmer must put them under the OP-code field, and if the pseudo-instructions require an address or data, the data or address is put under the operand field. Typical pseudo-instructions are: ***ORIGIN (ORG), EQUATE (EQU), DEFINE BYTE (DB), DEFINE CONSTANT (DC) and TITTLE (TTL).***

ORIGIN (ORG): The pseudo-instruction ORG lets the programmer place the programs any where in the memory. *ORG address* tells the assembler to load the program into memory starting at the specified *address*. Internally, the assembler maintains program counter types register called the address counter. This counter maintains the address of the next instruction or data to be processed. Recall that the jump instruction causes the microprocessor to place a new address in the program counter. Similarly the *ORG* pseudo-instruction causes the assembler to place a new value in the address counter, typical *ORG* statements are:

```
    ORG      7000H
   MOVE D1, #$02
```

The M68000 assembler will generate the following code for these statements:

```
          7000        3E
          7001        02
```

The *ORG* will assign memory address 7000H to the first Op-code (***MOVE D1, data***), then the next address 7001 to the operand (02_{16}). Note that 3E is the hexadecimal code for the instruction **MOVE D1, *data***, for the M68000.

EQUATE (EQU): *EQU* assign a value in its operand field to an address in its label field. This allows the user to assign a numeric value to a symbolic name. The user can then use the symbolic name in the program instead of its numeric value. This reduces error in programming.

A typical example is

```
      START EQU $0200
```

This pseudo-code instruction assigns the value 0200 in hexadecimal to the label START.

Another example is:

```
        PORTA   EQU   40H
                      MOVE #$FF, D2
                      MOVE D2, PORTA
```

In this example, the EQU gives PORTA the value 40 hex, and FF hex is the data to be written into the D2 register by the instruction *MOVE D2, #$FF*. The *MOVE D2, PORTA* instruction will output this data FF hex in the D2 register to port with address 40H.
Note that , if a label in the operand field is equated to another label in the field, then the label in the operand field must be previously defined. For example, the *EQU* statement in the instruction statement below:

```
                BEGIN            EQU          START
```

will generate an error unless ***START*** is defined previously with a numeric value.

DEFINE BYTE (DB):
The pseudo-instruction DB is usually used to set a memory location to a certain byte value. For example,

```
                START            DB           45H
```

Will assign the data 45 hex to the memory location pointed to by the label *START*. With some assemblers, the DB pseudo-instruction can be used to generate a table of data as follows:

```
                       ORG 7000H
                TABLE   DB   20H,   30H,   40H,   50H
```

In this case, 20 hex is the first data of the memory location 7000; 30 hex, 40 hex, 50 hex occupy the next three memory locations. Therefore, the data in memory will look like this:

memory	content
7000	20
7001	30
7002	40
7003	50

DEFINE CONSTANT (DC): The Define constant (DC) is a directive to the assembler to set up one or more data values as constant in memory, for example:

```
        CONST1   DC.B        $12      ; Set up the 8-bit byte 00010010
        CONST2   DC.W        $1234    ; Set up a 16-bit word
        CONST3     DC.B           $13, $14, $BC; Set up a list of data
values.
```

The constants CONST1, CONST2, CONST3, CONST4, and CONST5 are assigned the values $12, $1234, $13, $14, and $BC respectively.

CHAPTER FOUR

MOTOROLA MC68000 MICROPROCESSOR

INTRODUCTION

A series of microprocessor widely used as a controller and in PCs is the MC68XX and MC68XXX microprocessor series developed by Motorola. The former series of chips include MC6800, MC6801, MC6805, MC6802, while the latter, which evolved from the former, includes MC68000 (the subject matter), MC68010, MC68012, MC68020, MC68030, MC68040. MC68000 microprocessor is the Motorola's first 16-32bit-microprocessor chip. That is, it has 16-bit data path and capable of 32-bit internal operations. Other members of the former series are improved versions of MC68000 microprocessor, with many features added along the way. Its address and data register are all 32-bit wide. It can be operated from a maximum internal clock frequency of 25 MHz (available in several frequencies including 6, 10, 12.5, 16.67 and 25MHz). It requires a single 5v supply. It does not have on chip clock circuitry, it therefore, require external clock generator\driver circuit of a crystal oscillator to generate the clock input.

OPERATING MODE:

Mc68000 can be operated in two modes, ***User mode*** and ***Supervisor mode***. The user mode is used to execute user programs while the supervisor mode is used to implement operating system and protection features by the microprocessor.

It operates in either of this modes based on the logic level of "S" bit i.e. the supervisor flag of the status register (SR). When the 'S' bit is set i.e. S=1; then it

operate in the supervisor mode; however, when the S bit is zero i.e. S = 0 then the user mode is assumed.
The table below shows major difference between the two modes:

	Supervisory Mode	Users mode
Logic level of function code FC2	1	0
Enter mode by	Trap, reset, interrupt ac Knowledgment, S=1	Clearing Sbitof SR i.e. Supervisory flag
System stack Pointer	Supervisor stack pointer (A7)	User stack pointer') (A7)
Available Instructions	All instructions including STOP, RESET, RTE, MOVE to/from SR, ANDI SR, ORI SR, ,MOVE to/from USP to (An).	All except those Listed in supervisory mode.

1. The logical level of the microprocessor function code pin (i.e.FC2) indicates whether it's operating in either of the modes.

2. Via an instruction, program execution in supervisor mode can enter the user mode by modifying the S bit of status register to zero, such instructions include;

MOVE to/ from SR
ORI to / from SR
EORI to/ from SR

However, switching from User to supervisor mode can be caused by error such as TRAP, system reset, software reset, and interrupts recognition.

An interrupt will automatically be generated, if privileged instructions solely designed for supervisor mode are executed in user mode. 68000 has three function code pins (i.e. FC2, FC1, FCO) which specify to the external device whether it is accessing "*SUPERVISOR PROGRAM /DATA*", *performing an "INTERRUPT ACKNOWLEDGE CYCLE" OR "USER PROGRAM/DATA"*.

DATA TYPES, INSTRUCTION SET AND ADDRESSING MODE

MC68000 supports five different data types .They are 1-bit, 4-bit BCD digits, 8-bits (byte), 16-bits (word), and 32-bits(long word). Its instruction set includes 56 basic instruction types.14 addressing modes, and over 1000 ***Op-codes***. It executes the fastest and slowest instructions at 500ns {i.e. the one that copies contents of one register into another register} & 21.25µs at 8MHZ respectively.

It has no *input and output instruction*, hence, all input and output are *memory mapped*. The MC68000 is a general-purpose register microprocessor with many data registers which can be configure either as an "*ACCUMULATOR*" or as "SCRATCHPAD *REGISTER*". It has seven data registers and eight address registers including the supervisor stack pointer. Any data or address register can be used as an index register for addressing purpose. Although, it has 32-bit internal operated microprocessor; only the low-order 24 bits are used. It's also a byte addressable processor and can address up to 16MB of memory locations.

PROGRAMMING MODEL

The programming model of a microprocessor describes in detail all registers that are available in the microprocessor to a programmer, which can be manipulated. There are other registers that are not visible to the programmer. MC68000 has adequate registers for various manipulations. The programming model of MC68000 microprocessor is shown in figure 4.1.

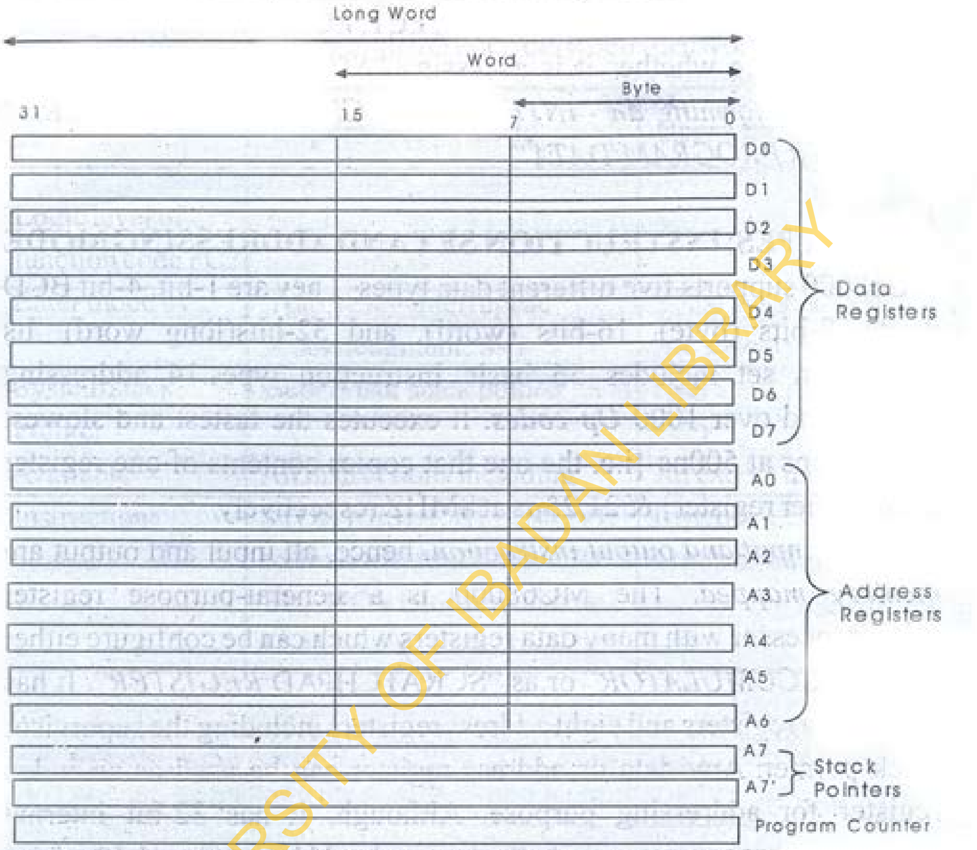


Figure 4.1: MC68000 Programming Model

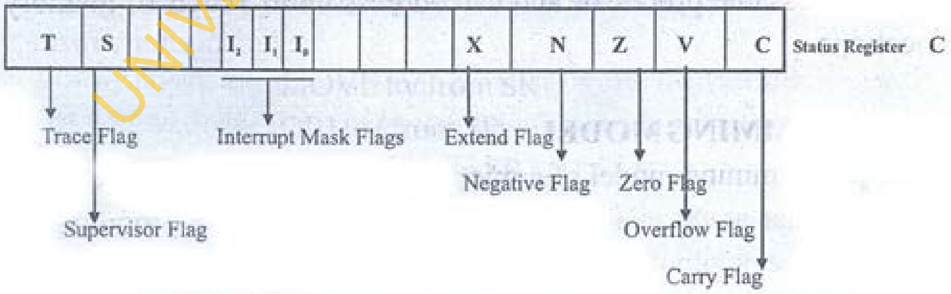


Figure 4.2: MC68000 Status Register with the Flags

Data Registers

The processor has eight; 32bit data registers DO through D7. Data registers hold arithmetic values and data item in form of 1-bit, 8-bit (byte), 16-bit word, 32-bit long word and 4-bit BCD numbers. Byte and word operations on data registers affect only the lower portion of the registers. For example a byte size movement into a data register affects only the 8 lowest significant bits of the register, the rest upper 24-bits are not affected.

Address Registers

There are nine, 32-bit address registers, A0 through A7 plus A7'. It uses A7 and A7' as the user and supervisor stack pointers (USP and SSP) as shown in figure 4.1. Address Registers are used to hold memory pointers (addresses) and index values of an operand. Seven of the address registers (A0-A7) are general purpose register. The eight, A7' is the supervisor stack pointer SSP. The stack is a memory area set aside to store data temporarily in memory in Last In First Out order (LIFO).

Program Counter

The program counter PC keeps track of the address of next instruction to be executed. It always contains the address of the next instruction in memory to be executed. PC is a 32-bit long register.

Status Register

The status register SR is composed of two bytes:

1. user byte
2. system byte.

The user byte includes the condition code flags such as *carry (C), overflow (V), negative (N), zero (Z) and extend (X)* flags as shown in figure 4.2. The functions of the flags are as explained below.

Carry (C): Carry flag can either be set (1) or cleared (0). It is Set, if arithmetic and logical operation carried out results in a carry or borrow bit from bit 7 to bit 8 in a destination data register. Otherwise it is cleared.

Overflow (V): Overflow flag is set to logical level of one, if the result of manipulation is larger than the destination register, that is the result is no longer reliable. Otherwise V = 0, signifying that the result is still reliable.

Zero (Z): Z flag is set (1), if result of arithmetic and logical operations is equal to numerical value zero. Otherwise, the Z flag is clear.

Extend (X): Extend flag is similar to carry flag because they are always affected the same way as carry flag, but extend flag is used in multi-precision instructions such as ADDX and SUBX. The reason for this is that MC68000 has no instruction for addition and subtraction with carry (ADC & SUBC).

Negative (N): Negative flag is set (1), if result of arithmetic and logical operations is equal to negative numerical value. For example N flag is set for instruction such as ***SUB D0, D1*** where D0 = 5, D1 = 4 is executed. Otherwise N is cleared to zero.

The system byte includes a 3-bit interrupt mask (I_2, I_1, I_0), a supervisor flag (S) and Trace flag (T).

Interrupt Mask: The interrupt mask bits form a binary number that specifies the currently acceptable interrupt level. Any interrupt request from peripheral devices whose level is less than or equal to the mask bits are not attended to. The programmer can set the interrupt mask bit by copying the desired level into the status register from a data register or by performing logical operations with the status register.

Trace Flag: When T flag is set (1), the processor generates a trap (internal interrupt) after executing each instruction. This will allow single stepping of instruction for debugging facility after execution of each instruction. when T = 0, such facility is disabled. Execution of ***ORI #$8000, SR*** set the trace flag to one in supervisory mode and ***ORI #$0000,SR*** clears the trace flag.

Supervisor Flag: When Supervisor flag S is set to one, the system operate in the supervisory mode, otherwise, the user mode operation is assumed. While the supervisor mode is intended for operating system usage, the user mode is for application program. Some instructions can only be run in the supervisor mode. They are called privileged instructions.

For Intel processors like Intel 80286/80386/80486, these modes are called PROTECTED and REAL MODE respectively.

ADDRESSING MODE

MC68000 microprocessor's addressing mode can be divided into six basic groups:

Register Direct
Address register indirect
Immediate Address
Absolute Address
Program Counter Relative
Implied Address

REGISTER DIRECT: Register Direct Addressing Mode specifies the register or register pairs that contain the data. In MC68000, the eight data registers (D0 D7) or seven address registers contain data operand(s). For example, consider the instruction ***ADD 004500, D0***. The destination of this instruction is in data register direct mode. If [004500] = 0004_{16} & [D0] = 0002_{16}: then after the execution of the above instructions the content of [D0] = $0004_{16} + 0002_{16} = 0006_{16}$. Another example of register direct addressing mode is ***MOVE D1, D4.***

IMMEDIATE MODE: Instruction in this mode specifies actual data value to be operated upon rather the address of the operand. There are two types immediate addressing mode available in MC68000. They are immediate mode and quick immediate modes.

In immediate mode, the source operand is a constant data and is part of instruction. The data could be as long as four byte in size.
For example, ***ADD* #$000FF005, D0;**
If [DO] = 0007H then after execution of the instruction,
[D0] = 0007H + 0005H = 000CH.
The symbol # is used by Motorola to indicate immediate mode.
In quick immediate mode, the immediate data could be a value from 0 to 7. For example, ***ADDQ #1, D0*** will add immediate data 1 to D0. The remaining upper 24 bits of the data register D0 is sign extended. This means the value of bit 7 of the D0 is copied into bits 8 -24.

IMPLIED MODE: In this mode, address of the operand is not specified in the instruction but it is assumed by the microprocessor. There are two types of implied mode in MC68000. They are implicit and explicit.
Implicit mode does not require any operand. Registers such as PC, SP, and SR are implicitly referenced. For example, ***RTE*** means returns from an exception routine to main program using SR and PC registers implicitly.
Explicit mode on the other hand, allows loading an operand or an address into program counter. For example, ***JMP address*** instruction will load the address supplied into PC .

ABSOLUTE MODE: The effective address is specified as part of the instruction in an absolute addressing mode. There are two types of absolute addressing mode in MC68000 .They are absolute short addressing and absolute long addressing. The former is used for 16-bit address while the latter is used for 24-bit address.

For example, Consider ***ADD $2000, D2;*** is an example of absolute short addressing while ***ADD.W $240200, D5;*** is an absolute long addressing.

ADDRESS REGISTER INDIRECT: In address register indirect, the instruction contains microprocessor's register(s) that contains the address(es) of instruction operand. The address of the operand is specified indirectly in an address register. There are five different types of Address Register Indirect in MC68000.

(i) ***Register Indirect***

(ii) ***Post Increment Register Indirect***

(iii) ***Pre-decrement Register indirect***

(iv) ***Register indirect with Offset***

(v) ***Index Register Indirect with offset***

REGISTER INDIRECT: in the register indirect mode, an address register (An) contains the effective address (EA). For example, consider ***CLR (A1).*** If [A1] = $003000, then after execution of ***CLR (A1),*** the content of memory location $003000 will be cleared to zero. The contents of address register A1 are used as address of the operand.

POST INCREMENT ADDRESS INDIRECT: this mode is similar to register indirect mode, except that the. concerned address register is incremented by '1' for byte(B), '2' for word(W) and '4' for long word (L), after execution of the instruction by the microprocessor.

Example: consider the instruction ***CLR.L (A0)+***

This instruction clears the content of the memory location pointed to by A0, and then adds '4' to A0 after execution because of the .L after the Op-code.

ANALYSIS: If the contents of memory locations 005000_{16} & 005002_{16} are 1234_{16} & 4567_{16} and [A0] = 005000_{16} then after execution, content of 005000_{16} & 005002_{16} are cleared to zero and [A0] = 005000 + 4, i.e. [A0] = 005004.

Post increment mode is typically used with "MEMORY ARRAYS" stored from "LOW TO HIGH" memory locations. For example, supposing, 1000_{16} words is to be set (write ones) starting at memory location 003000_{16}. The following instruction sequence can be used.

```
LABEL   OPCODE     OPERAND          COMMENTS
        MOVE. W    #1000, D0        ;load length of data into D0
        MOVEA. L   #$003000, A0     ; load starting address into
                                    A0
        REPEAT SET.W (A0)+          ;set a location pointed to by
                                    A0 and increment A0 by 2
        SUBQ#1, D0                  ;decrement D0 by 1
        BNE REPEAT                  ; branch to REPEAT if Z=0;
                                    otherwise go to next
                                    instruction.

        MOVE D4, D2
```

PREDECREMENT REGISTER INDIRECT: In pre-decrement register indirect, concerned address register is decremented by '1' for byte(**B**), '2' for word(**W**)and '4' for long word(**L**) before using a register. For example, Consider the instruction *CLR.W-(A0)*

This instruction first decrements the address register A0 by 'W' i.e. 2 and clears memory location addressed by A0

ANALYSIS: If [A0] =002004, then after execution of above instruction, the contents of A0 is first decremented by 2; since [A0] =002004 so therefore it will be [A0] = 002002_{16}. Then, the content of memory location 002002 is then cleared to zero.

The pre-decrement mode is used with MEMORY ARRAYS stored from High to low memory locations. For example, supposing, 1000_{16} words is to be cleared starting at memory location 4000_{16}. The following instruction sequence can be used.

```
LABEL  OP-CODE    OPERANDS    COMMENTS
       MOVE.W # $1000, DO        ; load length of data into DO
       MOVEA.W # $ 0004002,A0   ; load starting address plus 2
                                   Into A0
REPEAT CLR. W -(A0)             ;Decrement A0 by 2 and clear
                                 memory location addressed by
                                 A0.
       SUBQ#1,DO                ; Decrement D0 by 1
       BNE REPEAT               ; If Z = 0 branch to REPEAT;
                                  Otherwise go to the next
                                  instruction.
```

This instruction decrements A0 by 2 and then clears the memory location. The instruction sequence is repeated until $1000 memory locations have been cleared. Register A0 must initially be initialized to 004002_{16}, since the starting address is 004000_{16}.

REGISTER INDIRECT WITH OFFSET: In register indirect with offset; the EA is determined by adding a 16 bit signed integer (offset) to the content of an address register. For example, Consider the instruction ***MOVE.W $10(A3), D3***

***ANALYSIS*:** The source operand is an address register indirect with offset mode. Supposing [A3] $=00002000_{16}$ and $[002010]_{16}=0014_{16}$

i.e. EA=002000+10 (offset) =002010

Therefore after execution, content of memory location $[002010]_{16}$ $=0014_{16}$ will be moved to register D3, i.e. D3= 0014_{16}

INDEXED REGISTER INDIRECT WITH OFFSET: In the index register indirect with offset, the EA is determined by adding an 8-bit (offset) signed integer, the contents of an address register (base register) and a data register (index register). The index register indirect with offset is usually used when the offset from the base address register needs to be varied during program execution. This mode is usually used for accessing multidimensional data types. Size of the index register can be 16 bit integer or 32-bit integer value.

Indexed register indirect with offset addressing mode Instruction arrangement is given below:

Op-code offset (Base register, Index register.), Destination register.

MOVE.W $10(A4,D3.W), D4

Where 10_{16} is the 8bit offset that will be signed extended to 32-bit

A4 = Base register

D3. W =16 bit index register (sign extended to 32 bits)

D4 = Destination data register

Analysis:-

If $[A4] = 00003000_{16}$

$[D3] = 0200_{16}$

Therefore, EA = [A4] + [D3] +offset =003000+0200+10

EA = 003210

Suppose that, $[003210]_{16} = 0024_{16}$

Therefore, the low 16 bits of D4 register will be loaded with 0024_{16}.

INSTRUCTION SET

MC68000 contains 56 basic instructions. The instruction set repertoire is very versatile and allows an efficient means to handle high-level language (HLL) structures like linked lists and array. As shown below, notation 'B','W','L' is placed after each MC68000 mnemonic to depict the operand size whether it is byte, word or long word. All MC68000 instructions may be classified into eight groups as follows:

DATA MOVEMENT INSTRUCTIONS

ARITHMETIC INSTRUCTIONS

LOGICAL INSTRUCTIONS

SHIFT AND ROTATE INSTRUCTIONS

BIT MANIPULATION INSTRUCTIONS

BINARY CODED DECIMAL INSTRUCTIONS

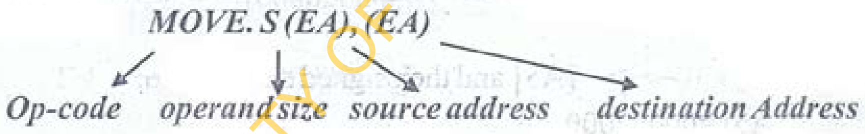
DATA MOVEMENT INSTRUCTIONS (DMI):

DMI instructions allow data transfers from register to register, register to memory, and memory to register, memory to memory. Consequently, special data movement instruction such as MOVEM (Move multiple registers) are possible as well as, Byte, word, long word data transfer are also permissible. $\overset{\text{There}}{\text{These}}$ are eleven data movement instructions in MC68000. They are: MOVE, MOVEM, MOVEP, MOVEQ, MOVEA, EXG, SWAP, LEA, PEA, LINK and UNLINK.

MOVE INSTRUCTIONS

1. **MOVE:** Move or copy data from source to destination.

The format for basic move instructions is:



EA can either be a register or memory depending on the addressing mode used. For example, ***MOVE B. D3, D1***

The above instruction uses source & destination data register & the operand size is 8-bit or a byte. If $[D3] = 05_{16}$ & $[D1] = 01_{16}$; Then, after the executions of the above information:

$[D1] = 05_{16}$ & $[D3] = 05_{16}$.

i.e. $[D1] \longleftarrow [D3]$

Another example, consider the instruction ***MOVE $02000, D0***

The instruction copies the content of memory location $02000 to data register D0. i.e. $[02000] \longrightarrow D0$

2. **MOVEA**: The instruction Moves data from the effective address of source operand to an address register. The assembler syntax is:

$$\begin{tikzcd} & \textbf{\textit{MOVEA.S (EA), An}} \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\ \textbf{\textit{Op-code operand}} & \textbf{\textit{size}} & \textbf{\textit{source address}} & \textbf{\textit{address register}} \end{tikzcd}$$

MOVEA instruction can be used to load an address into an address register from memory or for loading an immediate data into address register. . The
EA is calculated using the addressing mode specified in the instruction. For example, consider the instruction ***MOVEA.W #$2000,A5***
This instruction moves the immediate 16bits word 2000_{16} into the low 16-bit of A5 Address register. It signs extend 2000_{16} to 32-bit i.e. 00002000_{16}

i.e. $2000_{16} \longrightarrow [A5]$ and then signed extended to:

$[A5]=00002000_{16.}$

Consider the instruction ***MOVEA.L $04(A5,D2.W),A2***

Assume: $[A5] = 00024580_{16}$, $[D2] = 0045_{16}$ and $[0245C9_{16}] = 89764512_{16}$

Then, $EA = 04+00024580+0045 = 000245C9_{16}$

After the execution of the above instruction, 89764512_{16} is copied into data register A2.

$[A2] = 89765412_{16}$

MOVEA can use all available (14) addressing modes.

3. **MOVEM**: MOVEM instruction can be used to move multiple registers to an effective address. It can also be used to push or pop multiple registers to or from the stack.

Case 1: Multiple registers data transfer; for example, consider

D0..D7/A0..A7

MOVEM.W D0D7/A0A7,(A0) assume [A0] =00002000H

'W' indicates that the operand size is 16 bit long. Therefore, after the execution of the above instruction, D0 D7 and A0 A7 contents subject to the above assumption are:

D7	$\longrightarrow$	$[00002000_H]$	A7	$\longrightarrow$	$[00002010_H]$
D6	$\longrightarrow$	$[00002002_H]$	A6	$\longrightarrow$	$[00002012_H]$
D5	$\longrightarrow$	$[00002004_H]$	A5	$\longrightarrow$	$[00002014_H]$
D4	$\longrightarrow$	$[00002006_H]$	A4	$\longrightarrow$	$[00002016_H]$
D3	$\longrightarrow$	$[00002008_H]$	A3	$\longrightarrow$	$[00002018_H]$
D2	$\longrightarrow$	$[0000200A_H]$	A2	$\longrightarrow$	$[0000201A_H$
D1	$\longrightarrow$	$[0000200C_H]$	A0	$\longrightarrow$	$[0000201C_H]$
D0	$\longrightarrow$	$[0000200E_H]$	A0	$\longrightarrow$	$[0000201E_H]$

The instruction moves the lower word of data registers starting with D7-D0 to memory locations starting at 02000_H, and then moves the contents of address registers starting with A7-A0, while also considering the size of the operand by incrementing memory address by 2 for word sized operand.

Case 2: ***Push**ing and **Pop**ping multiple registers to and from the stack.*

Stack is a special memory buffer used as a temporary holding area for addresses and data. Each location on the stack is pointed to by address register (A7 or A7') called the ***'stack pointer'*** (SP). Stack pointer holds the address of the last data element added to, or pushed onto the stack. The last value added to a stack is also the first one to be removed or 'popped' from the stack. Most stacks are implemented using Last In first Out (LIFO) structure. For MC68000 address register (A7) is the user SP and it is either decremented or incremented depending on whether data or addresses are popped or pushed onto the stack. When data are pushed the stack pointer is decremented by the data size and incremented by the data size when data are $\overset{\text{Poped}}{\underline{\text{pushed}}}$ onto the stack.

Example: Consider the instruction below;

MOVEM.L D0D7/A1 A6,-(SP)

It saves (Transfer) the contents of all 8 data registers, 'Do D7' and seven address registers 'A0 A7' onto the user stack. However, the order 'A6 A0' is first stored into the stack, followed by the data registers in the order D7-D0 , regardless of the order in the register list. Also, the addressing mode of this instruction as provided is in a pre-decrement register indirect mode. Meaning that, address register will be decremented by the operand size before using the register. That is, if SP is pointing to the location 00002007 before execution, numeric value 4 is subtracted from SP to give new top of the stack (NTOS).

NTOS is given as 000020007– 4 = 00002003. Then, the registers are pushed to memory locations starting from NTOS. The pushed data can be popped back by using the instruction

MOVEM.L(SP)+, D0-D7/A0-A6

The above instruction will restore the registers' contents starting from D0-D7 and then A0-A6 because the stack is implemented using LIFO order.

USES OF THE STACK

1. Stack is an excellent temporary store for content of registers, if values in registers are to be preserved.
2. During programming, when a subroutine is called the program saves a return address on the stack i.e. the location in the program to which the subroutine is to return to.
3. High-level languages create an area on the memory for subroutine. It is called the stack frames where local variables are created while the subroutine is active.
4. **MOVEQ**: MOVEQ instruction moves the immediate 8-bit data in the instruction into the specified data register. It copies a small literal to a destination. The 8 bit data is then sign extended to 32 bits.

   Assembler Syntax: ***MOVEQ #data, Dn***

   For example: (i) ***MOVEQ.L #d8, Dn*** moves the immediate 8 bit data into the low byte of data register Dn.

   (ii) ***MOVEQ.L #$8F, D5***

The instruction copies $FFFFFF8F into D5 i.e. D5 $\longleftarrow$ FFFFFF8F

5. **MOVEP**: MOVEP instruction transfers data between data register (Dn) and alternate byte of memory locations, starting at the location specified by "d (Ay)" and incremented by the operand size. Data transferred can either be two (w) or Four (L) bytes. High order byte is transferred first, and low-order byte is transferred last.

Syntax: ***MOVEP Dn, d (Ay)*** or ***MOVE P d (Ay), Dn.***

Register indirect with offset (displacement) is the only addressing mode used with this instruction. If the address is even, all the transfers are made on the high order half of the data-bus; else if address is odd all the transfers are made on the low order half of the data bus.

For example, ***MOVEP.L $0020(A2), D1***

$$\begin{tikzcd}[column sep=small] \$0020 \arrow[d] & (A2) \arrow[d] & D1 \arrow[d] \\ \textit{Offset} & \textit{Address} & \textit{Destination} \end{tikzcd}$$

Let [A2] = 00002000

EA=[A2]+offset => 0020+00002000

.:. Computed EA= 00002020

.:. If [002020] = 02

[002022] = 04

[002024] = 06

[002026] = 08

Therefore, after execution of the above instruction; 02040608H will be contained in D1 i.e. [D1] = 02040608H.

6. **EXG and SWAP INSTRUCTIONS**:

EXG: EXG instruction exchanges the 32-bit contents of two registers.

SYNTAX: ***EXG Rx, Ry*** i.e. Rx $\longrightarrow$ Ry i.e. Rx= Ry

Rx or Ry can be any data or address register. Flags are not affected. It exchanges only 32-bit long words. The data size does not have to be specified after the instruction.

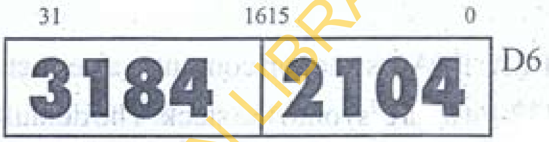
SWAP: SWAP instruction exchanges 16-bit halves of a data register.
SYNTAX: ***SWAP Dn***

For example, ***SWAP D6***

If [D6] =21043184

After execution;

[D6] =31842104



7. LEA AND PEA INSTRUCTIONS:

LEA: Load Effective Address (LEA) instruction moves an effective address (EA) from source operand into a specified address register (An). The effective Address is calculated using the addressing mode employed in the instruction.

SYNTAX: ***LEA (EA), An***

Size is long byte (32 bit), where EA specifies the actual data to be loaded into the register An.

For example, (i) LEA $00256011, A4

∴ [A4] $00256011

(ii) LEA $ 04 (A5, D2. W), A3.

If [A5] =0000200016

[D2]=002816

∴ EA=offset +[A5] + [D2] =04 +00002000 +0028

.:. EA=0000202616

The calculated EA is copied into address register A3.

NOTE: The instruction above is equivalent to ***MOVEA.L #$00002026, A3***.

The remarkable difference is that while the 'EA' in ***LEA EA, An*** specifies
the actual data to be loaded into address register (An). EA in ***MOVEA EA, An*** specifies the address of data to be loaded into An.

PEA: PEA instruction computes an effective address and then pushes it (32-bit address) onto the stack. The default EA size is long word.

SYNTAX: ***PEA (EA)***

For example: ***PEA $ 7504 (A5)***

If [A5] =00400100

Then, EA=00400100+7504

EA=00467604.

.:. The computed EA is pushed onto the stack.

9. LINK AND UNLNK INSTRUCTIONS:

As stated above, stack is used by High-level languages to create an area called stack frame to create local variables when subroutine is called. These variables can then be used by the subroutine for computations. The MC68000 ***LINK*** and ***UNLNK*** instructions are used for the purpose of reserving memory locations and releasing memory location when using stack for subroutines respectively.

LINK: LINK instruction is usually used at the beginning of a subroutine to allocate stack space for storing local variables and parameters for nested subroutine calls.

SYNTAX: ***LINK An, #-displacement.***

i Contents of specified address register An are pushed onto the stack.

ii The address register An is loaded from the updated stack pointer after the push.
i.e. An = SP = SP - N where 'N' is the size of the operand.

iii The 16 bit sign extended #-displacement is added to the SP computed in (ii)
i.e. SP = SP + (- displacement).

For example, consider:

LINK A0, #-40

i.e. 40 bytes of memory spaces are to be allocated to the subroutine as provided by the displacement in the instruction.

If [A0] = 00002100 &
[SP] = 00004104

Using above syntax for LINK instruction;

PUSHING: pushing [A0] onto stack

[A0] =00002100

00004104

(ii) UPDATING: SP = SP - N

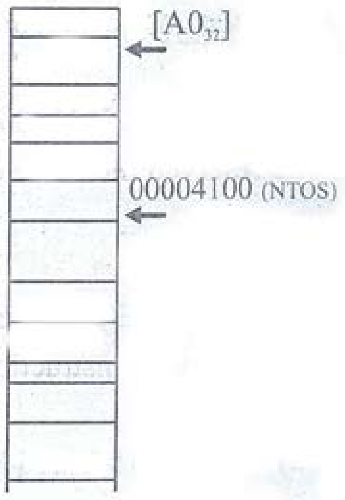
=00004104-4

SP = 00004100

(iii) ADDING DISPLACMENT TO SP

SP = SP + (-displacement)

=00004100 + (-40)



$=000040C0$

Therefore, locations 00004100 to location 000040C0 are available for the subroutine to use. A0 can be used as the base register for accessing the 40 bytes allocated spaces for the subroutine.

UNLNK: UNLNK is used at the end of a subroutine before the RETURN instruction to release the reserved local area and restore the stack pointer contents.

SYNTAX: ***UNLK An***

$$An \longrightarrow SP;$$
$$(SP)+ \longrightarrow An;$$

For example, ***UNLK A0***, the instruction will restore all the 40 byte locations reserved with ***LINK A0, #-displacement***, instruction above.

Since $A0 \longrightarrow SP$.

.:. SP=00004100

and $(SP)+ \longrightarrow A0$

SP=SP+4 {popping}

=00004100+4

Updated SP=00004104

:. Content of the memory location pointed to by SP is transferred into A0.

[A0]=00002100

The reserved memory locations have been de-allocated with **UNLK** instruction.

2. ARITHMETIC INSTRUCTIONS

Arithmetic instructions are used to manipulate data. It allows the following:

(a.)To perform 8-bit,16-bit,and 32-bit additions and subtractions.

(b.) To perform 16-bit by 16-bit multiplication (Both signed and unsigned) and 32-bit by 16-bit division (Both signed and unsigned).

(c.) Compare, clear and negate operations

(d.) Extended arithmetic instructions for performing multi-precision arithmetic.

(e.) Test instruction for comparing operand with zero.

(f). Test and set instruction which can be used for synchronization in a Multi-Processor system.

ADDITION AND SUBTRACTION INSTRUCTION

ADDITION:

There are various types of addition instructions depending on the state of source operand. The source operand can be an immediate data, contents of a data register or an address. The destination register is always a data register. Basic addition instructions are as discussed below:

ADD:

Add the content of the source register to the destination register. The assembler syntax is as follows:

ADD (EA), (EA)

Operation: EA $\longleftarrow$ (EA) + (EA)

For example, consider i. **ADD.W D1, D2**

If [D1]=0050 and [D2]=0008 then after execution of the above instruction [D20]=0050+0008=0058H

ii. **ADD.W $12200,D0**

If [12200]=0050H and [D2]=0008 then after execution of above instruction [D2]=[12200]+[D2]

[D2]=0050+0008=0058H

ADDI:

ADDI instruction adds immediate data to a register or memory location. The immediate data follows the word instruction. The assembler syntax is as follows:

ADDI #data, (EA)

Operation:

$(EA) \longleftarrow (EA)+data$

i. **ADDI.W #0012, $100200**

If [10020]=0008 then after execution of above instruction [10020]=0012+0002=0014H

ADDQ:

ADDQ instruction adds immediate integer data between 0 and 7 to the register or memory location in the destination operand.

The assembler syntax is of the form:

ADDQ #dx ,(EA)

Operation:

$(EA) \longleftarrow (EA)+dx$

where, dx is an integer between 0 and 7.

i. **ADDQ.W #02H,D1**

If [D1]=0020H then after execution of above instruction

[D1]=[D1]+#02

[D1]=0020H+02H

[D1]=22H

SUBTRACTION:

All subtraction instructions subtract the source from the destination operand. The source operand can be an immediate data or an address. There are also three types of subtraction instruction. They are as discussed below.

SUB:

Subtracts the content of source register to the destination register. The assembler syntax is as follows:

SUB (EA),(EA)

Operation: EA $\longleftarrow$ (EA) - (EA)

E.g. i **SUB.W D1, D2**

If [D1]=0005 and [D2]= 0008 then after execution of above instruction [D2] = 0008+0005 = 00003H

ii. **SUB.W D2, $1222H**

If [1222]=0070H and [D2]=0005H then after execution of above Instruction

[1222] =[12200] -[D2]

[D2]=0070-0038=0048H

SUBI:

The instruction subtracts an immediate data from a register or memory location. That is, from the effective address of a destination operand. The immediate data follows the word instruction. The assembler syntax is as follows:

SUBI #data, (EA)

Operation:

$$(EA) \longrightarrow (EA)\ \text{data}$$

for example, consider i. **SUBI.W #0012,$100200**

If [10020]=0008 then after execution of the above instruction,

[10020]=0012 - 0002 = 0010H

SUBQ:

The instruction subtracts an immediate data (0 to 7) from the contents of register or memory location in the destination operand The assembler syntax is of the form:

SUBQ #dx ,(EA)

Operation:

$$(EA) \longrightarrow (EA)\text{-}dx$$

where, dx is an integer between 0 and 7.

For example, consider i. **SUBQ.W #02H, D1**

If [D1]=0022H then after execution of above instruction [D1]=[D1]-#02

[D1]=0022H-02H

[D1]=0020H

MULTIPLICATION AND DIVISION INSTRUCTIONS

MULTIPLICATION:

Multiplication instruction set includes both signed and unsigned multiplication of integers. Basic multiplication instructions are as discussed below.

MULS:

Multiplies two 16-bit signed numbers and provides 32-bit result. The assembler syntax is as follows:

MULS (EA), Dn

Operation:

$(Dn)_{32} \longleftarrow (EA)_{16} * (Dn)_{16}$

for example: MULS #-02H, D5

If [D5] = 0003H therefore [D5] = 0003 * -02

$= -6_{10} =$ FAH

the result is however sign extended to 32-bit

i.e. $[D5]_{32}$ =FFFFFFFAH.

MULU:

Multiplies two 16-bit unsigned numbers and provides 32-bit result. The assembler syntax is as follows:

MULU (EA), Dn

Operation:

$(Dn)_{32} \longleftarrow (EA)_{16} * (Dn)_{16}$

for example MULS (A0), D5

If (A0)=0010200, [10200]=0300H and [D5]=0200H

Then [D5]=06000H

the result is however sign extended to 32-bit

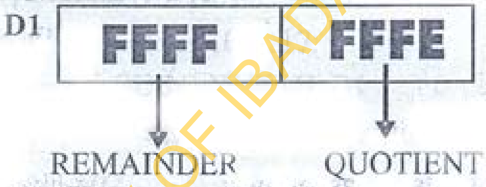
$[D5]_{32}$ = 00006000H.

DIVISION
Division instruction set includes both signed and unsigned division of integers.

DIVS:
DIVS instruction divides two signed numbers with each other. The high
Word of data register Dn contains the remainder and low word contains the quotient. The assembler syntax is as follows:

DIVS (EA),Dn

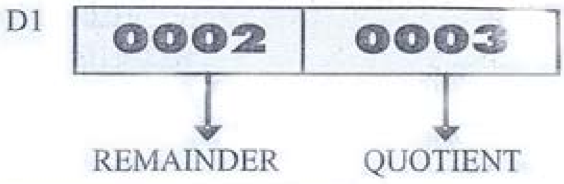
Example, Consider DIVS #2,D1
If [D1]=-5_{10}=FFFFFFFBH then after execution,
[D1]=-5/2= FFFFFEH
The value is sign extended to 32-bit.



After execution of DIVS, the remainder is always the same sign as the dividend unless the remainder is equal to zero. Internal interrupt is automatically generated when division by zero occurs.

DIVU:
DIVU is similar to DIVS except that the division is Unsigned. The assembler syntax is as follows:

DIVU (EA),Dn
Example Consider **DIVS #4, D1**
If [D1]= 14_{10}=000000EH then after execution [D1]=14/4 =3 Remainder 2
The value is sign extended to 32-bit.



Division by zero using DIVU instruction causes internal interrupt and V flag is set

i.e V=1

COMPARE, CLEAR AND NEGATE INSTRUCTION:

CLEAR: CLR

CLR instruction clears the content pointed to by the effective address. The assembler syntax is of the form:

CLR (EA):

Operation:

$$(EA) \longleftarrow 0;$$

Example. (i) **CLR.L D5.** Clears the content of 32-bit data register D5.

If [D5]=2222444 then after execution [D5]=0000000

(ii.) **CLR.L (A0)+.** Clears the content of where address register A0 points to. If (A0) = 00002420H then

$$(00002420) \longleftarrow 0$$

$$(00002421) \longleftarrow 0$$

$$(00002242) \longleftarrow 0$$

$$(00002423) \longleftarrow 0$$

A0 is incremented to = 00002224.

NEGATE:

NEG instruction negates the content pointed to by the effective address. The assembler syntax is given as:

NEG (EA);

Operation:

$$0 - (EA) \longrightarrow (EA)$$

Example: Consider **NEG.W (A0)**

If [A0]=00200000H and [200000]=5_{10} then, after negating [200000]=-5_{10} =FFFFBH (Sign extended to word length)

COMPARE:

CMP instruction compares the source operand with the destination operand. In such comparison, the source operand is subtracted from the destination operand without affecting the destination operand. The condition code flags are set or cleared to show the result of comparison. MC68000 Compare instructions and their corresponding assembler syntax are as follows:

CMP : CMP (EA) , Dn ;full range of operands

CMPA : CMPA (EA) , An ;Valid address operands only

CMPI : CMPI #d , Dn ;d may be immediate byte, word or long word

CMPM : CMPM (Ay)+ ,(Ay)+ ; Compares contents of memory with memory.

Example: **CMP.W (AO), D6**

If [A0] =00100020H and [100020]=0007H and [D6] =0009H then, after execution of the above instruction; N=C=X=V=Z=0, [A0] =01000020H, [100020] = 0007H and [D6] =0009H.

TEST INSTRUCTIONS:

TST instructions subtract zero from the content of an effective address. The assembler syntax is given as:

TST (EA);

Operation: (EA)-0 $\longrightarrow$ Flags affected.

Example: Consider **TST.W (A0)**

If [A0] = 00300000H and [300000] =FFFFH then after execution of the above instruction. operation FFFFH-0000H is internally performed and Z=V=C=0 and

N=1.

TEST AND SET INSTRUCTIONS: (TAS)

This instruction is similar to TST instruction. It allows only a byte operand. The value of the byte operand is tested and the N and Z flags are affected accordingly. If the result of test is equal to Zero then Z flag is set, else Z=0, N=1 and bit 7 of EA is set to 1. The assembler syntax is given as;

TAS (EA);

Operation:

If (EA)=0 then Z=1,N=0 else Z=0 and N=1 then, bit 7 of EA is set to 1.

Example: **TAS (0200000$_{16}$);** If [00200000] = 02H then comparing with Zero i.e.

02-00=02H.Therefore [EA] ? O. Hence Z=0, N=1and bit 7 of

3. **LOGICAL INSTRUCTION**

These are instructions such as AND, OR, EXOR (Exclusive-Or), and Logical NOT used for all sizes of integer data operands. The contents of address register are not use as an operand.

The assembler syntax are as follows:

AND: AND (EA),Dn
AND Dn,(EA)

ANDI: ANDI #d,(EA)
ANDI #d, SR; A word operation that affects all bits of SR
ANDI #d, CCR; A byte operation that affects CCR only

EOR: EOR Dn,(ea)

EORI: EORI #d,(ea)
EORI #d. SR; A word operation that affects all bits of SR
EORI #d, CCR; A byte operation that affects CCR.

OR: OR (EA), Dn;

ORI: ORI # d,(EA);
ORI # d,(EA);
ORI # d,(EA);

NOT: NOT (EA);

Examples:

(i.) **AND D1, D5;**

If [D5] =FFFFH and [D1] =0001H the logical AND of D1 and D5 i.e D1 ^ D5 gives 1111(F) 1111(F) 1111(F) 1111(F) i.e D5

<u>0000(0) 0000(0) 0000(0) 0001(1)</u> i.e D1

[D5] = 0000 0000 0000 0001_2 =0001H

(ii) **ORI.B #05, D2;**

If [D2] =04H then logical OR immediate D2 and 05H i.e D2 V 05H

0000000100 (04H)
0000000101 (05H)
0000000101 (05H)

(iii) **EXOR . W D1,D2 ;** i.e [D2]? [D2] EX0R [D1]

If [D1] =FFFFH and[D2] =AAAAH the logical EXOR of D1 and D2 s 1111(F) 1111(F) 1111(F) 1111(F) i.e. D2

1010(A) 1010(A) 1010(A) 1010(A) i.e. D1

0101 0101 0101 0101_2 =5555H

(iv) **NOT.B D5.**

If [D5]=02H i.e. $0000\,0010_2$ then, by inverting yields

$1111\ 1101_2$ = FDH

1. SHIFT AND ROTATE INSTRUCTIONS

These instructions perform shift operations to left and right and rotation operation with or without X flag. All shift and rotate operations can be performed on either data registers or on memory contents. Register shifts and rotates support all operand sizes (Byte, word and long word)and allow a shift count, which is specified as part of the instruction. Memory shift and rotates are for word operands only, and only single bit shifts and rotates are allowed.

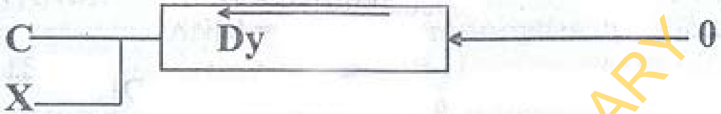
The assembler syntax of arithmetic and logical shift to the left and right instruction are as follows:

Arithmetic Shift to the Left:

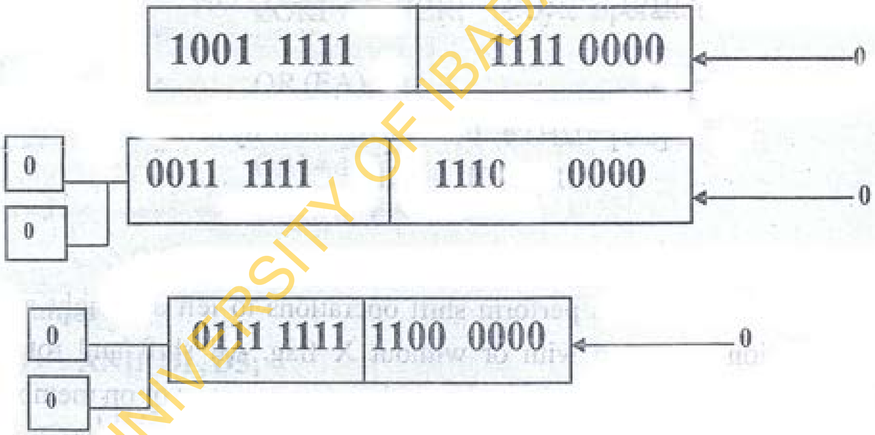
ASL : ASL Dx, Dy ; Shift Dy a number of times specified by Dx **ASL #d, Dn** ; =d= 8

ASL (EA) ; Shift memory contents pointed to by EA once.

Diagrammatic representation:

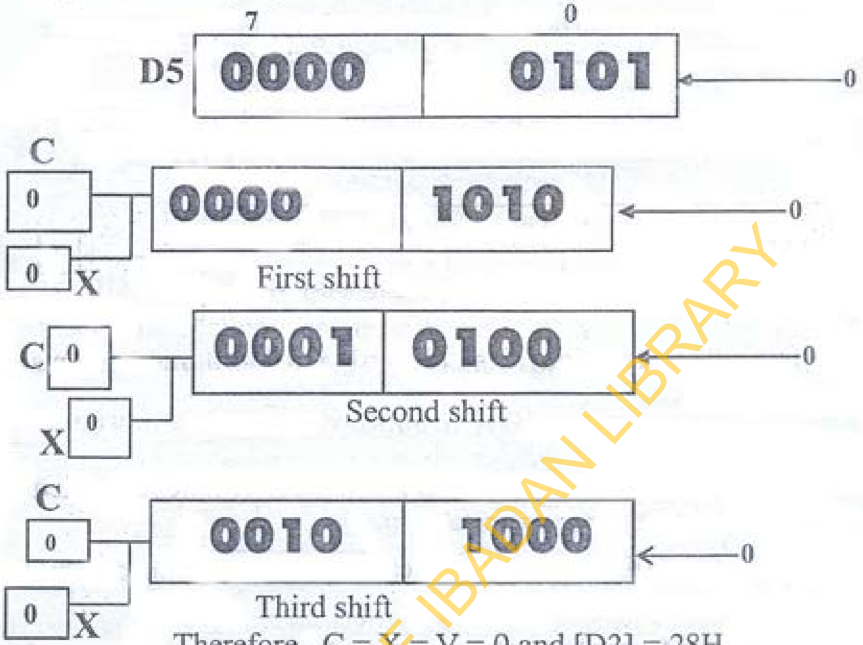


Example: Consider (i) **ASL. W D1, D5.** If [D1] =0002H and [D5] = 9FF0H, then Shifting D5 twice can be diagrammatically represented as:



After execution of the above instruction, the content of Flag C and X as specified after second shift is 0 i.e. C = X =0 and [D5] = 7FCOH.

ii) **ASL.B #03,D2;** Shifts D3 three times



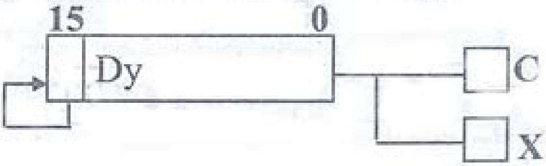
Therefore, C = X = V = 0 and [D2] = 28H

Arithmetic Shift to the Right: These instructions arithmetically shift right destination operand and still retains the sign bit.

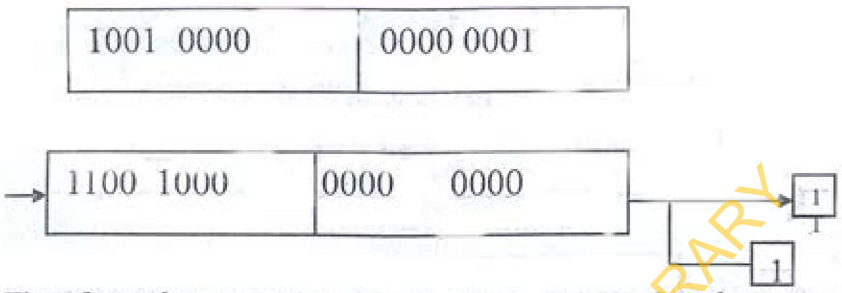
ASR:	**ASR Dx, Dy**	;Shift Dy a number of times specified
by Dx	**ASR #d, Dn**	;1=d= 8
	ASR (EA)	;Shift memory contents pointed to by

EA once.

Diagrammatic representation using 16 bits register content:



(i) **ASR.W D1, D5**. If [D1] = 0001H and [D5] = 9001H then
Shifting D5 once can be diagrammatically represented as:

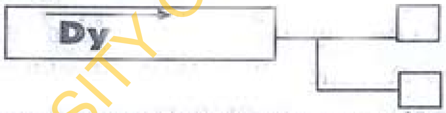


Therefore after execution, C = 1, X =1 and V =0 and contents of D5=C800H

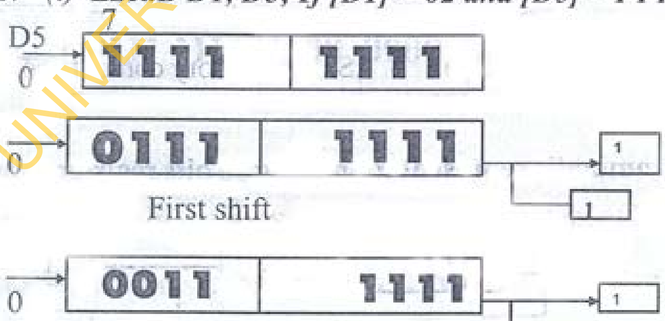
Logical Shift to Right:

LSR: **LSR Dx, Dy** ;Shift Dy a number of times specified by Dx **LSR #d, Dy** ;1=d= 8
LSR (EA) ;Shift memory contents pointed to by EA once.

Diagrammatic Representation:



Example: (i) LSR.B D1, D5; If [D1] = 02 and [D5] = FFH



First shift

Second shift

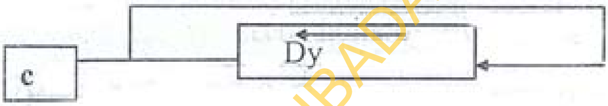
Therefore after execution of the above instruction C = X = 1 and V = 0 and D5 = 3FH

Logical shift to left: follows the same pattern with Logical shift to right except the direction of the shift is to the left of the reader.

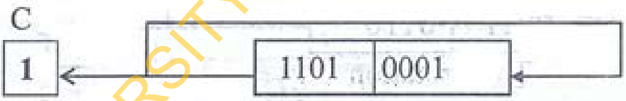
LSL: **LSL Dx, Dy** ; Shift Dy in number of times specified by DX **LSL #d, Dy** ; 1=d= 8
LSL (EA) ; Shift memory contents pointed to by EA once

Rotate to the Left and Right (ROL and ROR) Instructions: These instructions rotate contents of a register or memory depending on numbers of rotation specified.

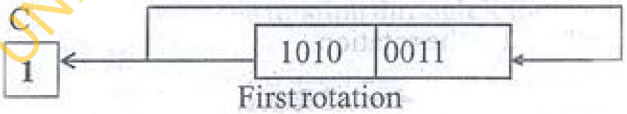
Diagrammatic representation of ROL:



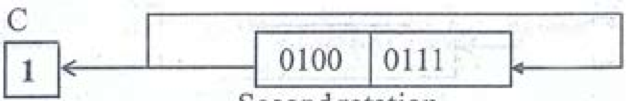
Example: Consider **ROL.B #02, D2;** IF [D2] = D1H AND C = 1.



Contents of D2 register and carry flag before execution



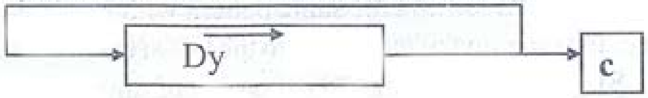
First rotation



Second rotation

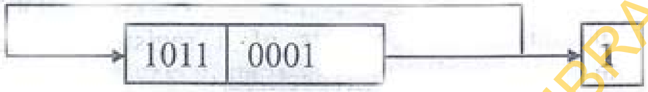
Therefore, after execution of the above instruction C=1 and D2=47H

Diagrammatic representation of ROR:

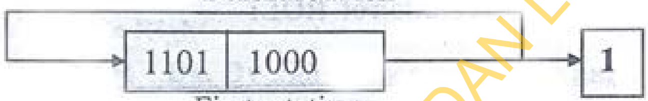


For example: **ROR #03, D2**

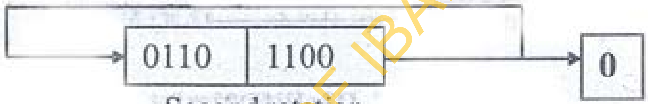
If [D2] = B1 and C=1. Then rotating D2 thrice gives:



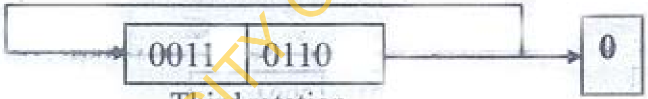
Before rotation



First rotation



Second rotation

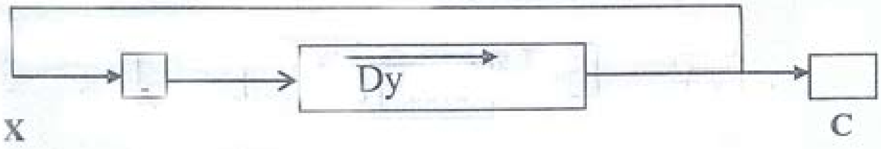


Third rotation

C=0 and D2=36H after execution of the above instruction.

ROXR is another rotational instruction that is similar to **ROR** but X-flag is included in the rotation

Diagrammatic representation of ROXR is given as:



For example: **ROXR.W D2, D1**; Supposing [D1] =F201H, [D2] =0003H C=0 and X=1 then rotation through X flag looks like:

X	D1		C
1	1111 0010	0000 0001	0

Before executing ROXR

1	1111 1001	0000 0000	1

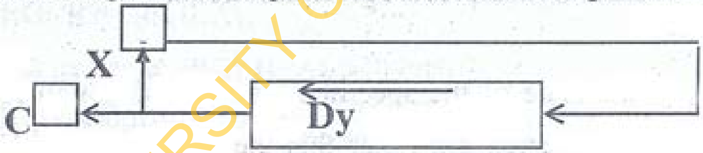
First rotation with X flag

0	1111 1100	1000 0000	0

Second rotation with X flag

So that [D1]=FC80H, C=X=0 after execution.

ROXL instruction is similar to **ROL** but X-Flag is included in the rotation. Diagrammatic representation of ROXL is given below:



For example: **ROXL.W D2, D1**; Supposing [D1] =F201H, [D2] =0003H,C=0 and x=1, then rotation through X flag looks like:

C	X	D1	
0	1	1110 0010	0000 0001

Before execution of ROXL

1	1	1100 0100	0000 0011

First rotation

$\boxed{1}$ $\boxed{1}$ $\boxed{1000\ 1000\ \ 0000\ 0111}$

Second rotation

$\boxed{1}$ $\boxed{1}$ $\boxed{0001\ 0000\ \ 0000\ 1111}$

Third rotation

After the third rotation, [D1]=100FH and C=X=1.

6. BIT MANIPULATION INSTRUCTIONS:

These instructions are used to test and modify the state of any specified bit (0-31) in a data register or any specified bit (0-7) of a byte in memory. The state of the specified bit affects the Z-flag. if the bit is zero then Z=1 else Z=0. These instruction and their corresponding assembler syntax, and examples are given as follows:

BCHG: Tests the state of the specified bit and Z flag will reflect complement of the tested bit. After this the bit is inverted in the destination operand. The assembler syntax is of the form:

BCHG Dn, (EA); Dn holds the bit position

BCHG #data, (EA); data specifies the bit position

For examples: BCHG D1, A0;

IF [A0] =0040220, [040220]= 06H and [D1]=2 i.e.

```
b7 b6 b5 b4   b3 b2 b1 b0
  0000          0110        Before
```

The instruction changes bit 2 to 0, we have:

```
b7 b6 b5 b4   b3 b2 b1 b0
  0000          0010
```

After, [040220] = 02H and Z-flag =0. (reflects the complement of b2)

BCLR: the instruction causes Z flag to contain complement of the specified bit and then clears the specified bit in the destination operand to zero. The assembler syntax is of the form:

BCLR Dn, (EA); Dn holds the bit position

BCLR #data, (EA);

For example: **BCLR #3, $004000;**

IF [004000]=08. That is,

```
b7 b6 b5 b4   b3 b2 b1 b0
  0000          1000
```

Before clearing bit 3 to 0

```
b7 b6 b5 b4   b3 b2 b1 b0
  0000          0000
```

After, [003000] = 00H and Z-flag =0.

BSET: The instruction causes Z flag to contain complement of the specified bit and then sets the specified source bit in the destination operand to one. The assembler syntax is of the form:

BSET Dn, (EA); *Dn* holds the bit position

BSET #data, (EA); immediate *data* holds the bit position

For example: **BSET #6, $004020;**

IF [004020] = ABH. That is,

b7 b6 b5 b4	b3 b2 b1 b0
1010	**1011**

Before setting bit 5 to 1

b7 b6 b5 b4	b3 b2 b1 b0
1110	**1011**

After, [004020] = EBH and Z-flag = 1.

BTST: Tests specified source bit in the destination operand and the result of testing reflect in the Z- flag(complement of the tested bit). The assembler syntax is of the form :

BTST Dn, (EA); Dn holds the bit position

BTST #data, (EA); immediate *data* holds the bit position l.

3. BINARY CODED DECIMAL (BCD) INSTRUCTION:

Four bits in binary coded decimal represent each of the decimal digits of a number. For example, the number 4530_{10} is represented as 0100 0101 0011 0000

They in BCD. MC68000 binary coded decimal instructions include ***ABCD, SBCD AND NBCD***. They are used for carrying out manipulations involving BCD arithmetic.

ABCD: ABCD instruction adds the source operand to the destination operand along with the extend bit of X-flag and stores the result in the destination operand. Addition is performed on BCD data. The assembler syntax is given as.

ABCD Dy, Dx; Dx = DX_{10} + Dy_{10}+X-flag

Example:

Consider ABCD D1,D2

If [d2] =15_{10} and [d1]=25_{10} and x=0, then

D2=15_{10}+25_{10}+0

[D2]=40_{10} and x=0 and z=0.

SBCD: This instruction is similar to **ABCD** except that it subtracts two BCD operands taking into consideration the X flag, which serves as borrow from bit 4 to bit3. The syntax is given below:

SBCD Dy, Dx; Dx = DX_{10}-Dy_{10}-X-FLAG

NBCD: This instruction negates BCD operand. The assembler syntax is given below.

NBCD Dx; Dx = ~(Dx): 1's complement of Dx

8. **PROGRAM CONTROL INSTRUCTIONS:** Program control instructions provide means of controlling the sequence of execution of the instructions in a

program. Transfer of program flow using program control instructions can be divided into four categories:

i) Unconditional transfer of control

ii) Conditional transfer of control

iii) Transfer of control to subroutine

iv) Return of control from subroutine.

i) **Unconditional Transfer of control:**

Under this category; program control instructions transfer control to the computed effective address in the instruction. These instructions are **JMP** and **BRA.**

JMP: Transfers control to the effective address unconditionally by coping the EA into PC. The assembler syntax is given as:

JMP (EA);

For example: JMP LOOP

Causes the address label **LOOP** to be set up in the PC, so that instruction in location **LOOP** is the next to be executed.

BRA: Is an unconditional branch always instruction. It transfers control to address label specified in the instruction field. The assembler syntax is given as:

BRA <displacement>

The effective address in **JMP** instruction is an absolute address, while that of BRA is an address calculated as a displacement from the value of the PC.

ii). Conditional Transfer of control:

This category of instruction causes the program to continue execution at a point other than the next instruction in sequence following the current instruction depending on the result of a condition tested in the preceding instruction.

facilitate the assembly language implementation of high level language control structures like IF-THEN, WHILE-DO, REPEAT-UNTIL etc. They are divided into three classes.

(a). **Bcc** instructions. (b). **Dbcc** Instructions and (c). **Scc** instructions

a) **Bcc instructions**: These instructions are conditional branch instructions. The suffix 'cc' is one of fourteen possible conditional branch instruction in the MC68000 instruction set corresponding to the flag bits of Condition Code Register (CCR) (lower byte of status register). The result of testing for a condition affects the CCR. Table 4.2 shows all such possible conditional branch instructions, the condition at which the transfer takes place and the affected flags.

INSTRUCTION	CONDITION TESTED	AFFECTED FLAG
BEQ	Equal to zero	$Z=1$
BNE	Not equal to zero	$Z=0$
BMI	Minus	$N=1$
BPL	positive	$N=0$
BGT	Greater than	$Z \wedge (N \vee V)=0$
BLT	less than	$N \vee V=1$
BLE	Less than or equal to	$Z \wedge (N \vee V)=1$
BHI	Higher than	$C \wedge Z=0$
BLS	Lower than or same	$C \vee Z=1$
BCC	Carry clear	$C=0$
BCS	Carry set	$C=1$

BVS	Overflow set	V=1
BVC	Overflow clear	V=0

Table 4.2: Conditional Branch Instructions

In this type of transfer of control, the suffix 'cc' is the condition being tested. For example, **BCS** tests if carry flag is set due to preceding instruction. If the condition is true then the branch takes place and the PC is loaded with the address to branch to else PC is unaffected and the next instruction in sequence is fetched and executed. Some possible illustrations are shown below:

a). **ADD.B DI, D3**; [D3] $\longleftarrow$ [D3] + [D0]
BLT *NEG*; Branch to address label **NEG**, if result stored in D3 is less than 0

b). **SUB.B DI, D3**; [D3] $\longleftarrow$ [D3] - [D0]
BEQ *SAME*; Branch to address **SAME** if the result stored In D3 is equal to zero; else continue the next sequence of instructions.

c). **CMP D1, D0;** Compare D1 with D0;
BEQ *SAME*; Branch to SAME if D1=D0

d). **CMPI.B #$41, D0; Compare** an immediate data byte with D0;

BLT ***BELOWD0***; Branch to BELOWD0 if D0<41

BEQ ***EQUALD0***; Branch to EQUAL D0 if D0 = 41

BGT ***ABOVED0***; Branch to ABOVED0 if D0>41

E). **MOVEQ #5, D7;** [D7] ⟵ 5 (counter)

LOOP: MOVE (A1)+, (A2)+; copy (A1) to (A2) and increment registers

SUBQ # 1, D7; Decrement counter

BNE LOOP; repeat copying until counter = 0.

b) **DBcc** Instructions:

These instructions test condition, decrement register contents and branch to the specified address if condition is true. They are used to provide a more compact and comprehensive loop instruction. The suffix 'cc' is the same as in the previous examples (Bcc instructions). General assembler syntax for this class of conditional instructions is given as:

Dbcc Dn , <Label>;

That is, tests condition, decrement and transfer control to address label provided in the instruction if condition is fulfilled. The instruction is effected as follows:

(i) If the condition specified in 'cc' is true then exit from the loop and jump to the next instruction, .i.e. PC=PC+2.

(ii) If the condition specified in 'cc' is false then the low word of D0 is decremented and the loop is re-entered.

Consider a loop below:

```
          MOVEQ #5, D2; [D5] <------ 5
LOOP:     ADD #2, D0;  [D0]<------ [DO]+2
          SUBQ #1, D2; [D2] <------ [D1] -1
          BGT LOOP;
          MOVE D0, D6
```

Can be re-written using **Dbcc** instruction as follows:

MOVEQ #5, D2; [D5] $\longleftarrow$ 5

LOOP: ADD#2, D0; [D0] $\longleftarrow$ [DO] +2

DBNE D2, LABEL; Test if D2$\overset{\neq}{?}$0, if true the loop is re-entered, else D0 is copied into D6,

DBEQ

MOVE D0, D6

c) Scc instructions:

These instructions set the byte contained in the EA according to the condition of the status register **cc** contained in the instruction. If the condition is true then all bits of the byte are set to 1, otherwise, all bits of the byte are reset to 0.

Syntax: **Scc (EA)**

iii) **Transfer of control to Subroutine:**

Program control instructions under this category transfer control from main program to subroutine either through branching or jumping. These instructions assem~~bler~~ syntax are given below:

BSR : **BSR <LABEL>;** Branch to subroutine

(iv) **Return of control from subroutine:**

These transfer control from the subroutine to main program after execution of the of last instruction in the subroutine. These instructions are:

RTR ; Return from subroutine and restore condition codes.

RTS ; Return from subroutine.

9. **SYSTEM CONTROL INSTRUCTIONS:** System control instructions are privileged instructions, trap generating instructions and instructions that use or modify status registers. These instructions are summarized as follows:

PRIVILEGED INSTRUCTION

STOP: stops program execution.

RTE; Return from exception

RESET; software reset for external devices.

Move An, USP; Move An to the User stack pointer (A7).

TRAP GENERATING INSTRUCTIONS

Chk (EA), Dn: if Dn <0 or Dn > (EA), then Trap, else do nothing.

TRAP#n; go to Trap vector number n

TRAPV; generate Trap on overflow;

STATUS REGISTER OPERATION

ANDI #D, CCR

EOR #D, CCR

MOVE SR, (ea)

ORI #D, CCR

CHAPTER FIVE

MC68000 ASSEMBLY LANGUAGE PROGRAMMING

INTRODUCTION

Following the introduction of Motorola 16-32bit microprocessor, MC68000 in previous chapter it is essential to understand in detail how assembly language programs are written using its instruction repertoire.

MC68000 has been selected as a good example for introducing assembly language due to the following reasons:

(i) Its state-of the-art architecture and assembly language are easy to use, and understand.

(ii) It supports up to five data types. They are 1-bit, 8-bit or byte, word (16- bit), long-word (32-bit) data types.

(iii) It has moderately sophisticated architecture incorporating many facilities found only on more powerful minicomputer and mainframe computers.

(iv) Assembly language written for it is upwardly compatible with other latter version of Motorola microprocessor.i.e.68010, 68008, 68020 and 68030.

In designing assembly language programs, unlike high level language where compiler performs data allocation to registers automatically, programmer must decide what goes in to any of the data registers and memory; address to a distinct address register; the type of data and address acquisition by the microprocessor for each of the program microinstructions. Programming in assembly language requires in-depth understanding of a particular microprocessor instruction set and its architecture. Assembly language programs are used to code device drivers to link peripherals with operating system.

PROGRAMMING TOOLS

Programming tools are used to transform precise requirement specification of a problem into a program. Among these tools are algorithm, pseudo code, and flowchart etc.

Algorithm is a set of instructions (unambiguous) that describes the step to be followed in order to carry out a task or an activity. When these finite instructions are represented using different symbols for various classes of operations performed in solving a task, a flowchart results. However, a convenient way of transforming problems into assembly language program is to use pseudo code or fake code as an intermediate step.

Pseudo code is a way of expressing algorithms in top-down and structured version. With this tool, the programmer can concentrate on what to do and leave the details of how to actually carry out individual actions until later stage. It is normally written in levels. The first level shows the major actions involved in problem solving while the latter levels shows the details of how to actually carry out actions specified in the previous level.

Consider a sequence of actions involved in writing a program to acts as a simple calculator. The first level and second level pseudo code are as follows.

FIRST LEVEL PSEUDO CODE

```
        Simple-Calculator
            Get variable
            Get operator
            Calculate result
            Print result
        End_Simple_calculator.
```

Second level elaborates on each of individual actions as follows.

SECOND LEVEL PSEUDO CODE

```
Simple-Calculator
            Get_variable
                -Get x
                -Get y
            Get operator op[]
            Calculate_result [z]
                -Determine operator and calculate
                    -If op [] = '+' then [z] = x+y else
                    -If op [] = '-' then [z] = x-y else
                    -If op [] = '*' then [z] = x*y else
                    -If op [] = '/' then [z] = x/y else
                Generate Error
            Print result
    End_Simple_calculator.
```

Process of elaboration continues until the point at which each action at the pseudo code level can be replaced by a relatively small number of instructions in a computer language.

SEQUENTIAL ASSEMBLY LANGUAGE PROGRAMS

These are programs without program control instructions. They are extensively used to program simple arithmetic operation that does not require iteration or branching.

Example: Given the following expressions write an appropriate assembly language program for the expressions.

(a) Y: = A+B

(b) Y: = 5A 3C

(c) Y: = 5A/3C

Solution:

If variable A and B are represented in memory as follows:

$\boxed{Y}$	=	$\boxed{A}$	+	$\boxed{B}$	**Content of add.**
$2222		$2000		$2001	

First level pseudo code	Second level pseudo code

Addition
    Get_variable
    Process variable
    Store_result
End_addition
	Addition
	  Get_variable
	    -Let A1 point to var A
	    -Let A2 point to var B
	    -Copy content of A1 to D0
	    -Copy content of A2 to D1
	  Process variable
	    -Manipulate D1 and DO with ADD
	  Store_result
	    Copy result to var Y
	End_addition

The program goes thus:

```
    LEA    A, A1; A1 points to A
    LEA    B, A2; A2  points to var B
   MOVE (A1), DO; DO←——(M (A1))
   MOVE (A1), DO; DO←——(M (A1))
   ADD    DO, D1; D1 ←———— [D0] + [D1]
  MOVE D1, $2222; var Y←———D1
  HALT; Stop
```

(b) Supposing variables A and B are located at address location $3000 and $4000 respectively. First level and second level pseudo code are as follows.

First level pseudo code	Second level pseudo code

Subtraction
    Get_variable
    process
    Store_result
End_Subtraction
	Subtraction
	  Get_variable
	    -Let A1 point to $3000
	    -Let A2 point to $4000
	    -Copy content of A1 to D0
	    -Copy content of A2 to D1
	  Process
	    -Multiply D0 with 5
	    -multiply D1 with 3
	    -Manipulate D1and DO with SUB
	  Store_result
	End_Subtraction

The program goes thus:

```
MOVEA#$3000,A1; A1 <———— $3000
MOVEA#$4000,A2; A2<———— $4000
MOVE(A1),D0; D0 <———— [M(A1)]
MOVE(A2),D1; D1<———— [M(A2)]
MULU#5,D1   ;D1<————  D1*5
MULU#3,DO   ;DO<————DO*3
SUB D1,D0    ;D0<———— DO - D1
HALT; Stop
```

(C) Modifying the pseudo code solution for example (b) above by changing *SUB to DIV*, we have the following lines of assembly language instruction for example C.

```
MOVEA#$3000,A1; A1 ←———— $3000
MOVEA#$4000,A2; A2 ←———— $4000
MOVE(A1),D0; D0 ←———— [MS(A1)]
MOVE(A2),D1; D1 ←———— [MS(A2)]
MULU#5,D1; D1 ←———— D1*5
MULU#3,DO; DO ←———— DO*3
DIV D1,D0   ; D0 ←———— DO/D1
HALT;  Stop
```

NON-SEQUENTIALASSEMBLY LANGUAGE PROGRAMS

The preceding section shows the sequential execution of instruction by conventional computer. There are occasions whereby the microprocessor is force to execute an instruction or set of instructions out of the normal sequence. These instructions that facilitates non-sequential execution of programs are called branching instruction.

Branch instruction can either be conditional or unconditional. Unconditional branch instructions transfer control to the address specified in the instruction. Examples are **BRA** and **JMP** instructions. Conditional branch instruction on the other hand, forces the microprocessor to take one or more courses of action depending on the result of prior action or instruction. There are fourteen conditional branch instructions of form Bcc in MC68000 instruction repertoire. Table 4.2, in chapter three shows MC68000 conditional branch instructions.

Illustrations:

A conditional branch instruction takes an argument or an operand, which is the address of the statement to be taken if the condition/ test is true. For example, consider:

```
          ADD D2, D3
          BMI ERROR
          MOVE D3, D4              ; beginning of the ELSE part
              -
              -
       ERROR:  MOVE D3, D5   ; beginning of the THEN part
```

In the above sequence of instructions, if N=1 i.e. negative flag in CCR, then a branch to the line addressed by label ERROR is made, and moves D3 into D5 else it moves D3 into D4.

However, if N$\neq$1 in the above illustration, ***ELSE*** part is executed, the execution falls through to the ***THEN*** part which is not the intended action to be performed. Therefore to cater for both conditions independently, an unconditional branch instruction can be used as follows.

```
          ADD D2, D3
          BMI ERROR
          MOVE D3,D4


          BRA EXIT
ERROR:    MOVE D3, D5
 EXIT:    HALT
```

The flow chart for the above code fragment is shown in figure

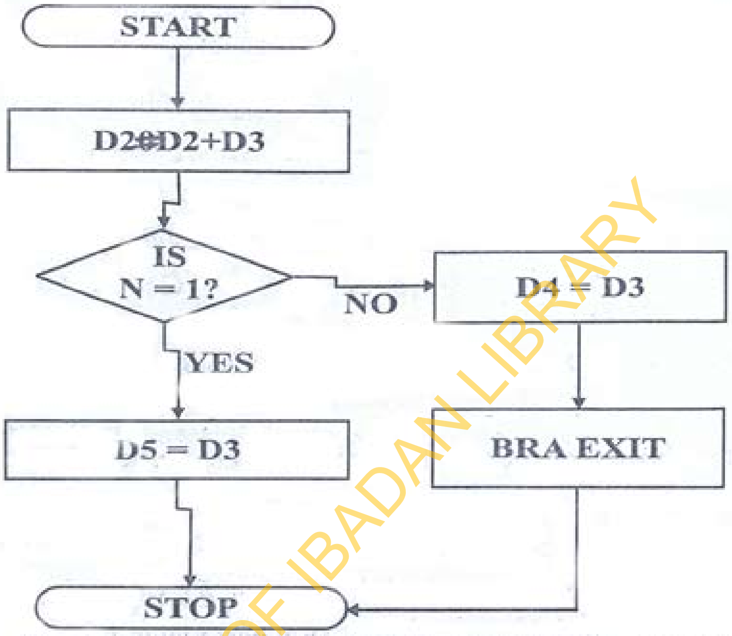


Figure 5.1: Flow Chart for Conditional Branch Instruction

TEMPLATES FOR CONTROL STRUCTURES

Familiar control structures in high-level language can be readily represented in templates in assembly language. This pattern of assembly language code can then be modified to suit similar conditional and iterative programming circumstances. Table 5.1 below shows the basic high-level language control structures and equivalent templates in assembly language.

HIGH LEVEL LANGUAGE CONTROL STRUCTURE	EQUIVALENT TEMPLATES IN ASSEMBLY LANGUAGE
IF (D0=D1) THEN ACTION1	**CMP D0,D1**;Perform test **BNE EXIT** ; If D0≠,/D1 then exit Else execute Action1 **ACTION1:..........** **EXIT: HALT;** Stop
IF (D0=D1) THEN ACTION1 ELSE ACTION2	**CMP D0,D1**;Perform test **BNE ACTION2** : If D0□,D1 then Action2 Else execute Action1 **ACTION1...** **BRA EXIT;** **ACTION2:.......** **EXIT: HALT;** Stop

FOR Z=1 TO P
BEGIN
ACTION1:
END.
	MOVE #1,D2 ; Load loop counter D2, With 1
	ACTION1...
	**ADD #1,D2;**increment loop counter
	CMP #P+1,D2; Test for end of loop
	BNE ACTION1; If not end of the loop then go round again
	Else exit
	EXIT: HALT; Exit from loop

WHILE **(D0=D1)** Do **ACTION1** **REPEAT**	**TEST: CMP DO,D1;** Perform test **BNE EXIT** :If D0≠D1 then exit Else carry out action 1 **ACTION1…**
ACTION1 **UNTIL DO=D1**	**CMP DO,D1;** Carry out test **BNE ACTION1** :Repeat as long as DO≠D1 **EXIT: HALT;** Exit from loop

Table 5.1: Control Template for High Level Language and Assembly Language

PROGRAMMING ILLUSTRATIONS:

1. Write an MC68000 assembly language program equivalent of the following segment of Pascal program code:

```
a.    Sum: =1;
      For i: = 1 To 14 DO
      Sum: =sum+1;

b.    Sum:=0; i=1:
    Repeat
          Sum: =sum + A[i];
          i: =I +1;
    Until (i=15);
```

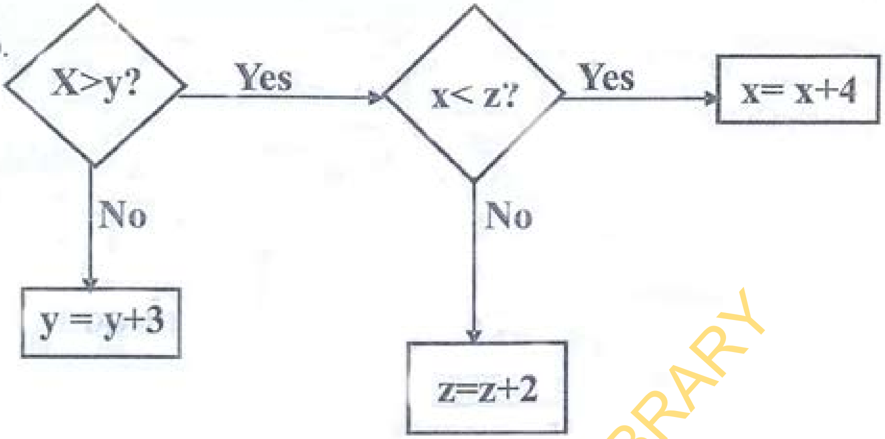
```
C.      If (x>y ) then
           If (x<z) then
             x:=x+1
           Else
             z:=z+2
        Else
             y:=y+2;
```

Solution:

```
(a)              MOVE #1,DO  ; Load loop counter, D0,With 1
               MOVE #1, D1; Initialize sum with 1
     NEXT   :ADD DO, D1; [D1] <---- [DO] + [D1]
               ADD #1, DO; [DO]  <------ [DO] + 1
               CMP #15, DO;   Perform test
               BNE EXIT
    EXIT:      JMP EXIT

b).      LEA  SUM, AO; AO <------  SUM
          LEA A, A1; A1 <------  A
         CLR DO; [DO] <------  0
          MOVE #14, D1;  D1 <-------D1 +14
NEXT:   ADD (A1)+, D0; D0 <----  D0 + ((M (A1)); A1 <-----A1+2
         DBF D1, NEXT;  Decrement D1 and branch until D1 is -1
         MOVE   DO, (AO);
END     JMP   END;
```

C).



```
        LEA X,A1;  A1 ←——X
        LEA  Y,A2;  A2 ←——Y
        LEA Z ,A3;A3 ←——Z
        MOVE (A1),DO;
        CMP (A2),DO; Compare x with y
         BGT  NEXT;
         ADD #3, (A2); y=y+3
         BRA FINISH;
    NEXT: MOVE (A3), D1;
         CMP D1,D0;  compare x with z
         BLT NEXTADD;
         ADDI #2,D1; z=z+2
         BRA FINISH;
NEXTADD: ADDI #4,DO; x=x+4
FINISH: JMP FINISH;
```

2. Write a 68000 program to compute $\sum_{i=1}^{N} Xi*Yi$. Where N=100

Solution:

```
          ORG $004000;
          LEA Y,AO; let A0 point to var Y
          LEA X,A1; let A1 point to var X
          MOVE #99,DO; let D0 be counter
         CLR D1; D1 serves as sum of X_i +Y_i
LOOP: MOVE (A0)+,D2
         MULS (A1)+,D2;
         ADD D2,D1;
         DBF DO,NEXT;
NEXT : JMP NEXT; stop
```

3. Write a 68000 program to compute $\sum_{1}^{N} X_i^2/N$. Where i =1 and N =500.

Solution:

```
         ORG $001000;
         LEA X,A0; A0 point to var X (X_i)
         CLR.B DI;
         MOVE #500, D0; load loop counter D0 with 500
NEXT: MOVE (A0)+,D2; copy X_i into register D2
          MUL D2, D2; square D2
          ADD D2, D1; D1 = ΣX_i^2
         SUBI #01, DO; decrement loop counter
```

BNE NEXT;

DIVS #500, D2; $D2\sum_{1}^{N} X_i^2/N$

BRA FINISH;

FINISH: JMP FINISH; Stop

4. Write a 68000 program to set 100_{10} consecutive bytes starting from location 003000H.

Solution:

```
              MOVEA.L #$3000, AO; load A0 with starting memory
 location
        CLR DO;
        ADD  #99,D0
 LOOP: SCC. B (A0)+; set contents of the memory location pointed to by
         A0
         DBF D0,LOOP; decrement D0 and  repeat the LOOP until
         D0=-1
         HALT; Stop
```

5. Write a 68000 program to clear 100_{10} consecutive bytes starting from location 3000H.

Solution:

```
              MOVEA.L #$3000,AO; load effective address into A0

              CLR DO;

              ADD  #25,D0; loop counter

       LOOP: CLR .L(A0)+; clears four consecutive bytes at once

              DBF D0, LOOP; repeat loop until D0 is minus.

              HALT;
```

6. Write a 68000-program segment to transfer 500 bytes of data from memory location named TABLE 1 to another memory location named TABLE 2..

Solution:

```
                LEA   TABLE1, A0; let A0 point to the beginning of the
first table
                LEA  TABLE2, A1; let A1 point to the second table
                MOVE #500, D0;
NEXTBYTE: MOVE.B (AO)+,(A1)+
                DBNE D0,NEXTBYTE;
END             JMP      END;
```

7. Write a program to determine whether the content of memory location 3000H is odd or even. If the content is even, add 00H to location 4000H else add FFH to location 4000H.

Solution:

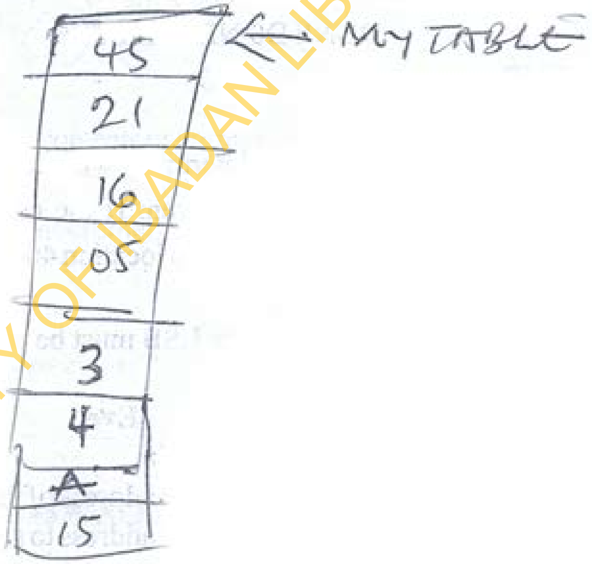
For a number to be even, its **LSB** must be equal to zero otherwise, the number is an odd number. $\boxed{00000010}$ **Even** $\boxed{00000101}$ **Odd**

```
  MOVEA #$3000, A0; load the address of the data to be tested into A0
  MOVEA #$4000, A2; load the address to store result into in A2
  MOVE (A0), DO; load the data into register D0
  LSR #1, DO; test the LSB of the data
  BCC EVEN; if the LSB is zero then go to the label EVE
  MOVE #$FF, (A2); else store FF in result location
  BRA END;

EVEN: MOVE #$00, (A2); store 00 in the result location
```

Assignment -

To generate the algorithm



Terminate the program

MC68000 Program equivalent to the above algorithm:

```
; MC68000 program to find the highest data in a table of data stating at
location
; MYTABLE and the set of data are terminated by ASCII character "A".
; The highest data is to be stored in location MAXDATA and its location
stored in
; memory location MAXLOC
;
MAXDATA      DC.B                    ; variables declaration in
MAXLOC       DC.L                    ; memory
MYTABLE      DC.B
              LEA MYTABLE, A0    ; A0 ←—— 4000H
             MOVE.B (AO), DO    ; D0 ←—— largest ←—— (AO)
TEST:        CMPI 'A'   , (AO)      ; Test if data = 'A'
             BEQ   END              ; if equal terminate
             CMP (AO), DO           ; Compare data
             BLE SRC-HIGHER         ; Branch if largest< (AO)
NEXTLOC   : MOVE.LAO, DI           : Increment address
              ADDI #I, DI            ; register AO to next
              MOVEA DI, AO           ; location
              BRA   TEST             ; next comparison
SRC-HIGHER: MOVE (AO), D0           ; copy data into largest.
ADR_OF_DATA: MOVE.L A0, D7          ; address of the largest element.
              BRA NEXTLOC
```

```
FINISH   :MOVEA D7, AO            ; transfer address of largest data
                                  ; into AO
         MOVE.L AO, MAXLOC        ; copy address of the highest data to memory
         MOVE.B (AO), MAXDATA     ; copy the highest data into memory
         END
```

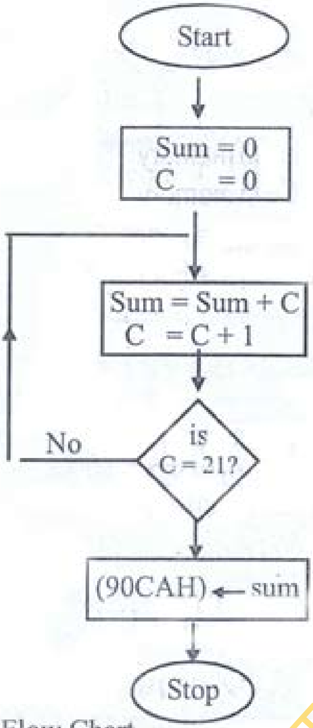
9. Write a 68000 assembly language program to load 100H consecutive memory locations, starting at address 2000H, with the sequence of data 0 ,1,2,3,4...

Solution:

```
; 68000 program to load 100H consecutive memory locations starting at
; addresses 2000H with the sequence of data 0, 1, 2, 3, 4,
;
              MOVEA #$2000, A0
              CLR. B DO
              MOVE #$ 100, D1
NEXT:         MOVE DO, (A0)+
              ADDI #1, DO
              DBNE D1, next
FINISH:       JMP FINISH
```

10. Draw flow chart and write a program to add together the numbers 1 to 20 and store the final, result in memory location 90CAH

Solution



```
; program to add together the number
; 1 to 20 and stores the final, result in
;location 90CAH
LEA SUM, AO
CLR DO
MOVE #21, DO
EXT: ADD (A0), DO
ADD #1, DO
 CMP DO ,D1
 BNE NEXT
MOVE (AO), & 90CA
END   : JMP END
```

Flow Chart   Program

11. Write a 68000 assembly language program for the simple calculator algorithm treated earlier in chapter five.

Solution

; Assembly language to implement simple calculator operation
; Z=X [+,-,*, /] Y using the pseudo codes given in chapter five. X, Y, Z and the ;operator are variables assigned to memory locations **VAR1, VAR2 RESULT** ;and **OPERATOR** respectively.

```
        ORG   4000
```

```
VAR1   :      DC.W                    ; memory location for X
VAR2   :      DC.W                    ; memory location for Y
OPERATOR: DC.B                        ; memory location for operator [+,-,*, /]
RESULT:       DS.L                    ; storage location for Z
              ORG 4100
              LEA VAR1, A0            ; point to variable in memory
              LEA VAR2, A1            ; point to variable in memory
              LEA OPERATOR, A2        ; point to variable in memory
              LEA RESULT, A3          ; point to variable in memory
              MOVE (A0), D0           ; get  var X
              MOVE (A1), D1           ; get  var Y
              MOVE (A2), D2           ; get  operator
              CMPI '+' ,D2            ;determine operator type
              BNE  SUBTR              ; if operator ? '+' go to SUBTR
              ADD D1,D0               ; X = X+Y
              BRA STORE               ;store result
SUBTR:        CMPI '-' ,D2            ;determine operator type
              BNE MULT                ; if operator ? '-' go to MULT
              SUB D1,D0
              BRA STORE
MULT:  CMP '*', D2                    ; determine operator type
              BNE DIVDE               ; if operator ? '*' go to DIVDE
              MUL D1,D0
              BRA STORE
DIVDE:        CMP '/', D2             ; determine operator type
              BNE  ERROR              ; if operator ? '/' go to ERROR
              DIV D1,D0
              BRA STORE
ERROR:        TRAP #0
STORE:        MOVE D0, (A3)           ; Z = X [OPERATOR] Y
              END
```

BIBLIOGRAPH

1. Bacon, J. (1986): The Motorola MC68000: An Introduction to Processor, Memory and Interfacing, Prentice-Hall, New York.

2. Rafiquzzaman, M. (1998): Microprocessor: Theory and Applications, Mac-Graw Hill, Singapore.

3. Hall, D.V. (1983): Microprocessor and Digital Systems, M a c - Graw Hill, Singapore.

4. Clements, A. (1999): Principles of Computer Hardware, Oxford University Press, New York.

INDEX