# LECTURE-02
# MACHINE STRUCTURE, MACHINE LANGUAGE, AND ASSEMBLY LANGUAGE

GENERAL MACHINE STRUCTURE

All the conventional modern computers are based upon the concept of *stored program computer*, the model that was proposed by John von Neumann.

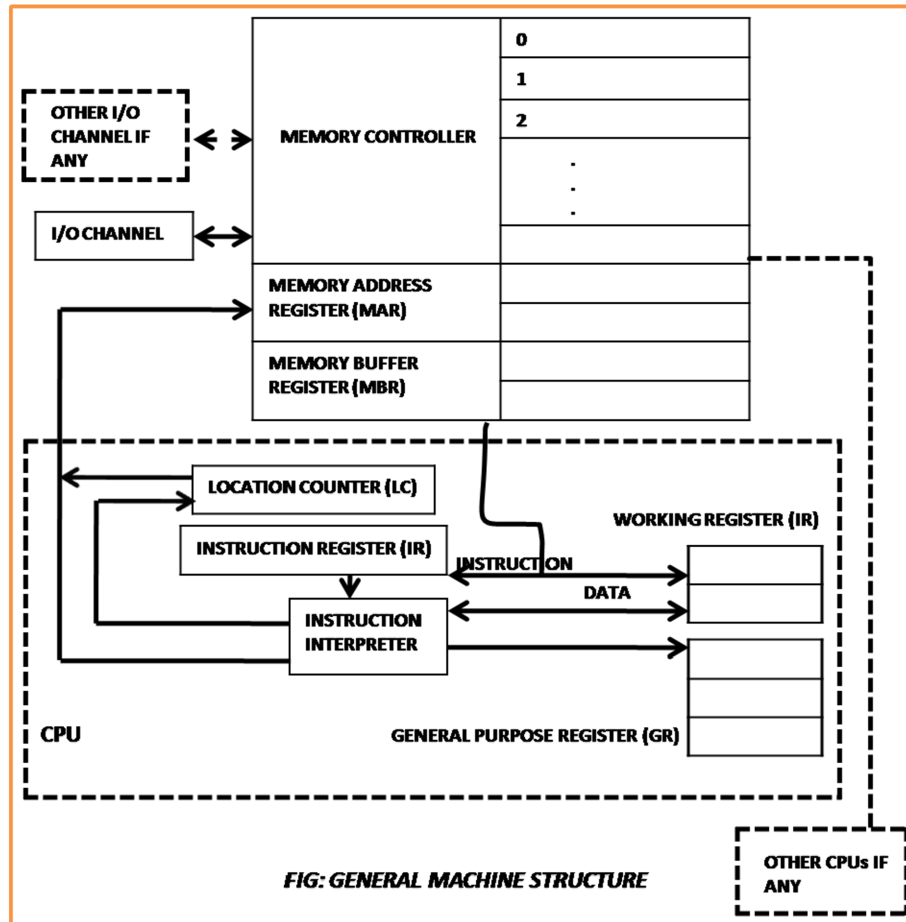


*FIG: GENERAL MACHINE STRUCTURE*

The components of a general machine are as follows:

***Instruction interpreter:*** A group of electronic circuits performs the intent of instruction of fetched from memory.

***Location counter:*** LC otherwise called as *program counter PC* or *instruction counter IC*, is a hardware memory device which denotes the location of the current instruction being executed.

***Instruction register:*** A copy of the content of the LC is stored in IR.

***Working register:*** are the memory devices that serve as "scratch pad" for the instruction interpreter.

***General register:*** are used by programmers as storage locations and for special functions.

***Memory address register (MAR):*** contains the address of the memory location that is to read from or stored into.

***Memory buffer register (MBR):*** contain a copy of the content of the memory location whose address is stored in MAR. The primary interface between the memory and the CPU is through *memory buffer register*.

***Memory controller:*** is a hardware device whose work is to transfer the content of the MBR to the core memory location whose address is stored in MAR.

***I/O channels:*** may be thought of as separate computers which interprets special instructions for inputting and outputting information from the memory.

To illustrate how these components of machine structure interact, let us take a command ADD 2,176.

This instruction has three parts first the opcode i.e. ADD, second is the number of the register that contain the first operator, third is the memory location address that contain the second operand.

- At first, the address from the IC is copied to the MAR.
- Then the instruction is fetched to the MBR.
- The instruction is then transferred to the IR.

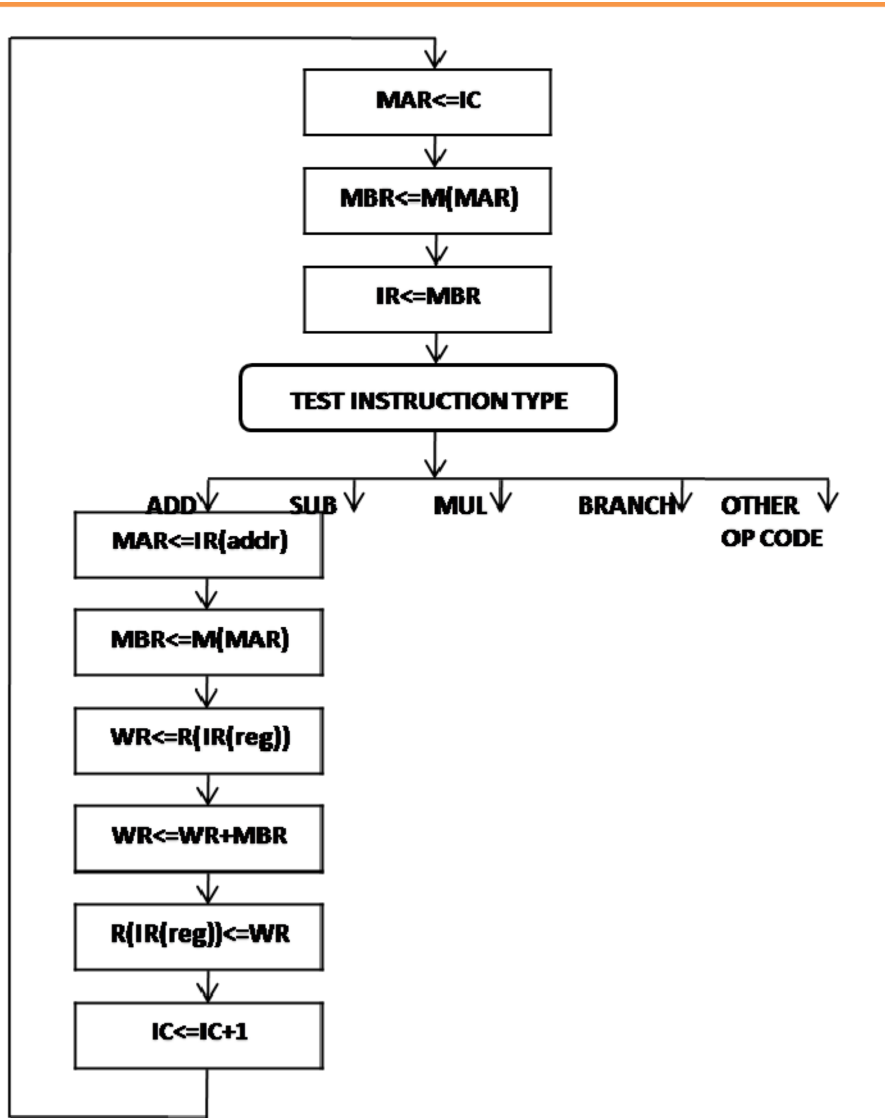



***FIG: Example of micro flow chart for ADD instruction.***

- Then the operand of the instruction is checked and the corresponding branch is taken, here ADD branch is chosen.
- Then the memory location of the second operand is placed in the MAR.
- Then the content is placed in MBR.
- Now the first operand is placed in the WR.
- Finally the sum of WR and MBR is calculated and the stored in WR.
- The content of WR is stored to the register that contained first operand.
- And then IC is incremented to point to the next instruction.

GENERAL APPRACHES TO A NEW MACHINES

In order to know a new machine we have a number of questions in mind. These questions can be categorized as follows.

***Memory:*** Basic unit, size and addressing scheme.

***Registers*** Number of registers, and size, functions, interrelation of each register.

***Data:*** Types of data and their storing scheme.

***Instruction:*** Classes of instructions, allowable operations and their storing scheme.

***Special Features:*** Additional features like interrupt and protections.

## ***Machine structure -360 and 370***

All the parameter defined above will be discussed for IBM 360 and 370 machines.

***Memory***

The basic unit of memory in 360 and 370 is a *byte* (eight bits of information). It means, each addressable position in memory can contain a byte of information. There are facilities to operate on contiguous bytes in basic units. The basic units are as follows:

| Unit of memory | Bytes | Length in bits |
|---|---|---|
| Byte | 1 | 8 |
| Half word | 2 | 16 |
| Word | 4 | 32 |
| Double word | 8 | 64 |

A smaller unit of memory of size 4 bits is called as *nibble*. The size of a 360 memory is upto $2^{24}$ bytes. The addressing of a 360 memory consists of three parts. Specifically the value of an address equals to value of the *offset,* plus the content of the *base register*, plus the content of the *index register.*

***Registers***

There are a total of **16 *general purpose registers*** of 32 bits each. In addition there are ***4 floating point register*** of 64 bits each. It also has a 64 **bits *program status word (PSW)*** that contains the value of the location counter, protection information and interrupt status.

The general purpose registers are basically used in arithmetic and logical operations as *base registers* and helps in address formation. The general purpose registers also used as scratch pads for the programmers. Let us take an instruction A 1,901(2,15).

A(opcode)1(operand in register 1),901(offset) (2(index register),15(base register))

This is how the memory locations are addressed in case of 360 and 370. The use of base registers in addressing is twofold.

First it helps the loader in the process of relocation (changing the content of base register causes the code to be relocated to the specified location).

Secondly it decreases the size of instruction as follows: since the memory of 360 is of $2^{24}$ hence a total of 24 bits are required to specify a particular location of memory. This increases the size of instruction as opcode takes 8 bits and the registers require 4 bits each and the address requires a 24 bits hence the size of the instruction is 8+4+4+24=40 bits (without base register)

Fig. Addressing without the use of base register.

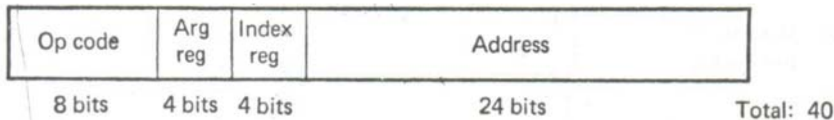
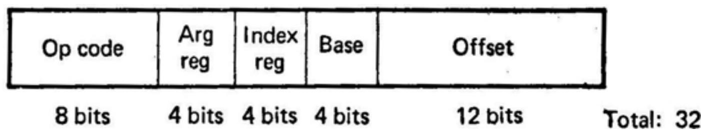



Fig. Addressing using base register.




In the second scheme when we use the base register then a total of 8+4+4+4+12(offset)=32 bits are used, saving a total of 40-32=8 bits.

The only disadvantage with this method is that the offset can take value from 0 to 4,095 locations away from the core location to which the base register is pointing.

***Data***

The 360 may store several different types of data as is depicted in the figure.

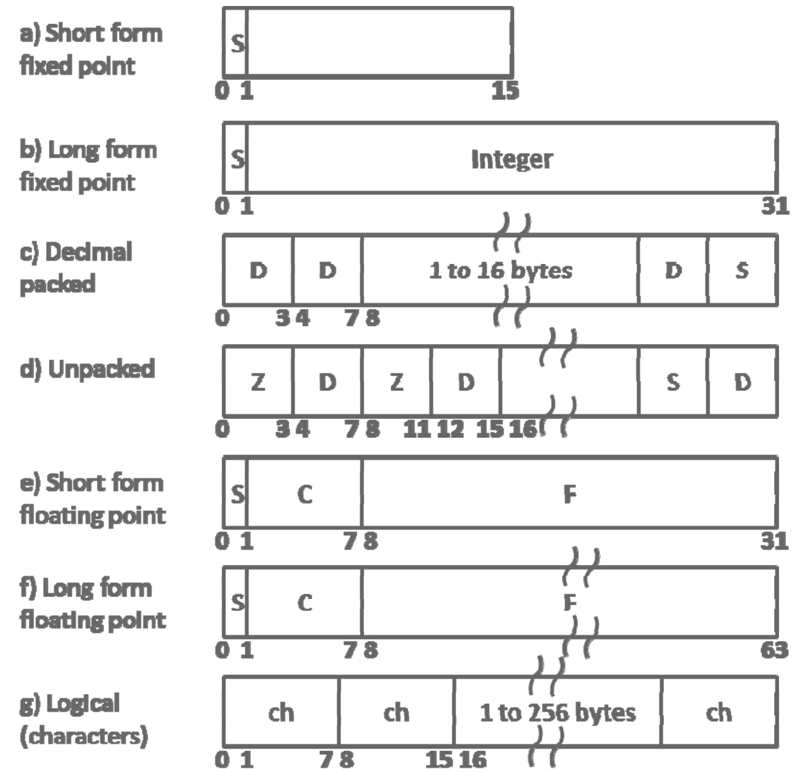



**Fig. Data formats for system 360/370**

The groups of bits stored in memory are interpreted by 360 processor in several ways. The list of different interpretation are shown in the figure are as follow.

Short form fixed integer
Long form fixed integer
Decimal packed
Unpacked
Short form floating point
Long form floating point
Logical

E.g. +541 is represented as 0000 0010 0001 1101, the first bit represents the sign i.e. + and the rest 15 bits represents the integer 15. Where are -021 can be represented as 0000 0010 0001 1101 i.e. 0 2 1 and -, in decimal packed form representation. Where are + 300 is represented as 0000 0001 0010 1100.

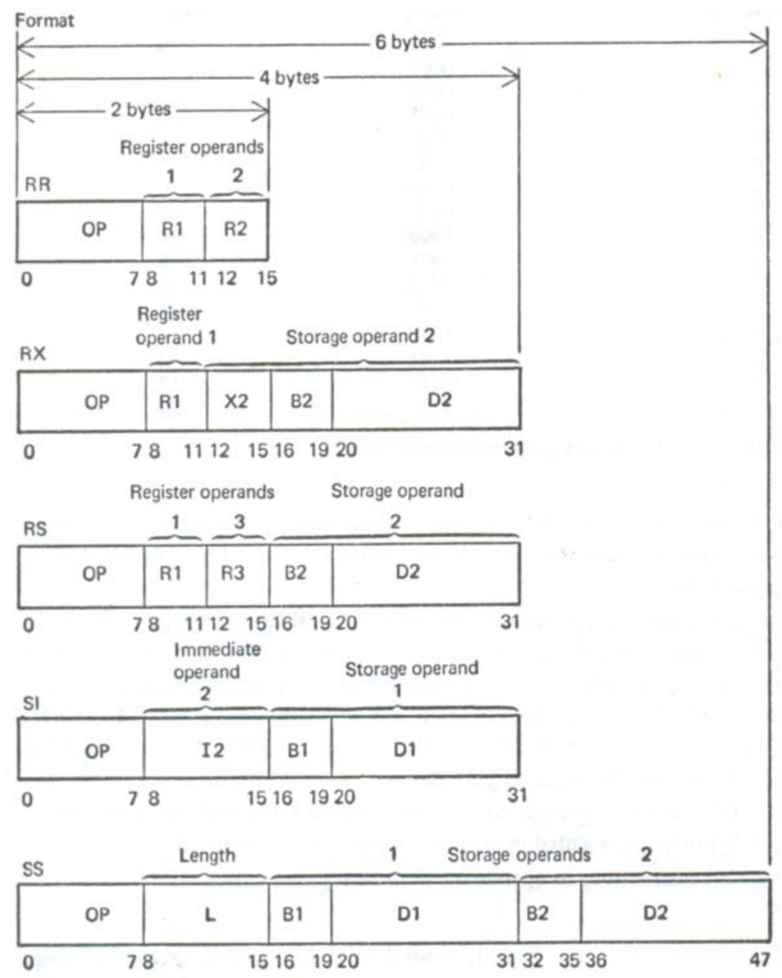
It is more convenient to represent the numbers in hexadecimal as every hexadecimal digit represents four binary digits and vice versa. Thus +300 which is B'0000 0001 0010 1100' in binary can be written as X'012C' in hexadecimal. The prefix B and X represents binary and hexadecimal respectively.

| *Hexadecimal* | *Binary* | *Decimal* |
|---|---|---|
| 0 | 0000 | 0 |
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 3 |
| 4 | 0100 | 4 |
| 5 | 0101 | 5 |
| 6 | 0110 | 6 |
| 7 | 0111 | 7 |
| 8 | 1000 | 8 |
| 9 | 1001 | 9 |
| A | 1010 | 10 |
| B | 1011 | 11 |
| C | 1100 | 12 |
| D | 1101 | 13 |
| E | 1110 | 14 |
| F | 1111 | 15 |

**FIGURE** Hexadecimal-binary-decimal conversion

Fixed point numbers can be represented by a halfword or fullword as in figure a, b. Decimal number can be represented in a form similar to that of binary as 12 can be represented as 0001 0010 as shown in figure c, d. Floating point numbers and logical data can be stored as in figure e, f, g.

***Instruction***



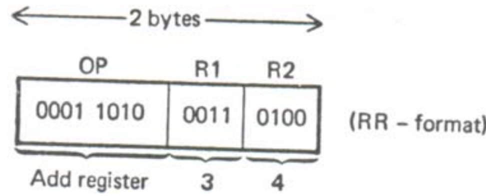

Mnemonics used:
OP ~ operation code
Ri ~ contents of general register used as operand
Xi ~ contents of general register used as index
Bi ~ contents of general register used as base
Di ~ displacement
Ii ~ immediate data
L ~ operand length

**FIGURE 2.5** Basic 360 instruction formats

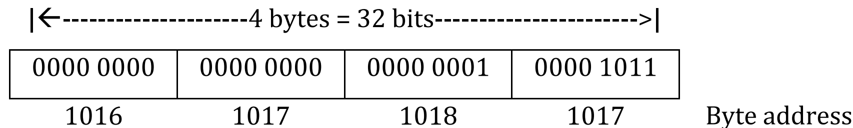
The instructions in 360 can be arithmetic, logical, control or transfer and special interrupt instructions. The format of 360 instructions is as in figure above.

There are five types of instructions that differ in the type of operands they use.

1. *Register operand* refers to the data stored in the 16 general purpose registers (32 bits each). Registers being high-speed circuits provide faster access to data than the data in the core. E.g. *Add register 3, 4* causes the contents of the contents of the register 4 to be added to that of register 3 and stored back in the register 3. The instruction is represented as given in the diagram. The is called as *RR format*. A total of two bytes are required to represent the RR instruction 8 bits for opcode and 4 bits each of register(8+4+4=16 bits =2 bytes).




2. *Stored Operand* refers to the data stored in the core memory. The length of the operand depends on the specific data types. For operand of length more that one byte the address is specified by the lowest address byte (leftmost). E.g. 32 bits binary fixed point fullword with value +237(X'00 00 01 0B' in hexadecimal), stored in location 1016, 1017, 1018 and 1019 as depicted below is said to be located at 1016.



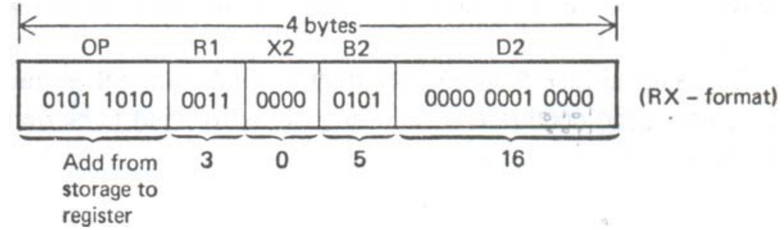
|←--------------------4 bytes = 32 bits----------------------->|

| 0000 0000 | 0000 0000 | 0000 0001 | 0000 1011 | |
|---|---|---|---|---|
| 1016 | 1017 | 1018 | 1017 | Byte address |

The address of ith storage operand is computed from the instruction in the following manner:

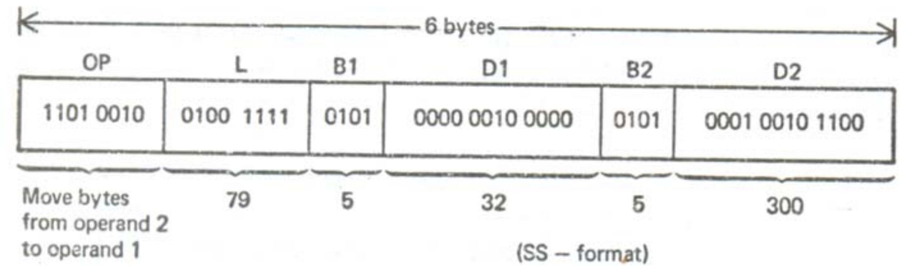
Address = c(Bi)+c(Xi)+Di (RX format)
or = c(Bi) +Di (RS, SI, SS format)

Where c(Bi) and c(Xi) represents the content of base and index register respectively. If Xi=0 the c(Xi)=0 likewise for Bi. E.g. here in the figure Bi=5 and it contain i.e. c(Bi)=1000 and Xi=0 so c(Xi)=0. Then the address of the second operand is calculated as follows for the RX instruction in the figure.




Address = c(B2) + c(X2) + D2
= c(5) + c(0) + 16
= 1000 + 0 +16
= 1016

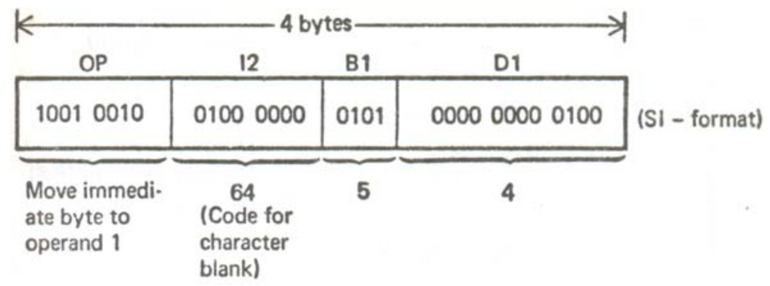
Hence the instruction cause the content of the word (32 bits) located at address 1016 to be added to the contents of general purpose register 3 (32 bits), with the resulting sum left in general register 3.




The above figure represents an example of SS instruction format that copies the contents from location one to location 2. The amount of content to be coped is given by L. Here storage operand 1 addr = c(B1) + D1
= c(5) + 32
= 1000 + 32 = 1032
Storage operand 2 addr = c(B2) + D2
=c(5) + 300
=1000 + 300 = 1300

The instruction copies (moves) the 80 bytes from location 1032 -1111 to locations 1300-1379. Since a character is stored as a byte, this instruction could be viewed as copying an 80-character from one area to another.

3. *Immediate operand* are a single byte of data and are stored as part of the instruction. With the same assumption that the register 5 contain 1000. The SS instruction given in the figure below cause the bytes 0100 0000 (bits 8 through 15 of instruction) to be stored at location 1004.




Hence 0100 0000 will be stored at location 1004.

***Representative 360/370 instructions***

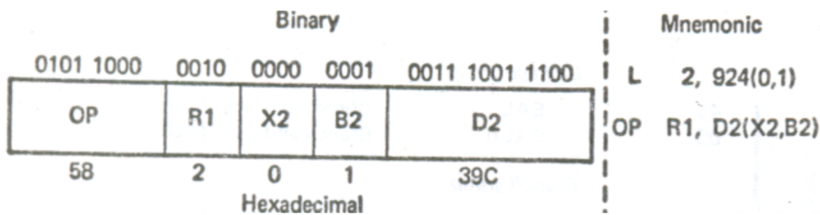
A list of 360 op code with the instruction format and their hexadecimal code are listed in the table below:

| | Hexadecimal op code | Mnemonic | Meaning (format) |
|---|---|---|---|
| Load-store-register | | *Load group* | |
| | 58 | L | Load (RX) |
| | 48 | LH | Load halfword (RX) |
| | 98 | LM | Load multiple (RS) |
| | 18 | LR | Load (RR) |
| | 12 | LTR | Load and test (RR) |
| | | *Store group* | |
| | 50 | ST | Store (RX) |
| | 40 | STH | Store halfword (RX) |
| | 90 | STM | Store multiple (RS) |
| Fixed-point arithmetic | | *Add-group* | |
| | 5A | A | Add (RX) |
| | 4A | AH | Add halfword (RX) |
| | 1A | AR | Add (RR) |
| | | *Compare-group* | |
| | 59 | C | Compare (RX) |
| | 49 | CH | Compare halfword (RX) |
| | 19 | CR | Compare (RR) |
| | | *Divide-group* | |
| | 5D | D | Divide (RX) |
| | 1D | DR | Divide (RR) |
| Fixed-point arithmetic | | *Multiply-group* | |
| | 5C | M | Multiply (RX) |
| | 4C | MH | Multiply halfword (RX) |
| | 1C | MR | Multiply (RR) |
| | | *Subtract-group* | |
| | 5B | S | Subtract (RX) |
| | 4B | SH | Subtract halfword (RX) |
| | 1B | SR | Subtract (RR) |
| Logical | | *Compare-group* | |
| | 55 | CL | Compare logical (RX) |
| | D5 | CLC | Compare logical (SS) |
| | 95 | CLI | Compare logical (SI) |
| | 15 | CLR | Compare logical (RR) |
| | | *Move-group* | |
| | D2 | MVC | Move (SS) |
| | 92 | MVI | Move (SI) |
| | | *And-group* | |
| | 54 | N | Boolean AND (RX) |
| | D4 | NC | Boolean AND (SS) |
| | 94 | NI | Boolean AND (SI) |
| | 14 | NR | Boolean AND (RR) |
| | | *Or-group* | |
| | 56 | O | Boolean OR (RX) |
| | D6 | OC | Boolean OR (SS) |
| | 96 | OI | Boolean OR (SI) |
| | 16 | OR | Boolean OR (RR) |
| | | *Exclusive-or group* | |
| | 57 | X | Exclusive-or (RX) |
| | D7 | XC | Exclusive-or (SS) |
| | 97 | XI | Exclusive-or (SI) |
| | 17 | XR | Exclusive-or (RR) |
| | | *Shift* | |
| | 8D | SLDL | Shift left (double logical) (RS) |
| | 89 | SLL | Shift left (single logical) (RS) |
| | 8C | SRDL | Shift right (double logical) (RS |
| | 88 | SRL | Shift right (single logical) (RS) |

| | Hexadecimal op code | Mnemonic | Meaning (format) |
|---|---|---|---|
| | | *Linkage group* | |
| Transfer | 45 | BAL | Branch and link (RX) |
| | 05 | BALR | Branch and link (RR) |
| | | *Branch group* | |
| | 47 | BC | Branch on condition (RX) |
| | 07 | BCR | Branch on condition (RR) |
| | 46 | BCT | Branch on count (RX) |
| | 06 | BCTR | Branch on count (RR) |
| | | *Miscellaneous* | |
| Miscellaneous | 9E | HIO | Halt I/O (RX) |
| | 41 | LA | Load address (RX) |
| | 9C | SIO | Start I/O (RX) |
| | 0A | SVC | Supervisor call (SI) |
| | 9D | TIO | Test I/O (RX) |
| | 43 | IC | Insert character (RX) |
| | 91 | TM | Test under mask (SI) |
| | 42 | STC | Store character (RX) |

**MACHINE LANGUAGE**

In this section we will dealing with the machine language of 360 machine. We will not be using 0's and 1's or hexadecimal, rather we will be using mnemonics for writing machine level programs.

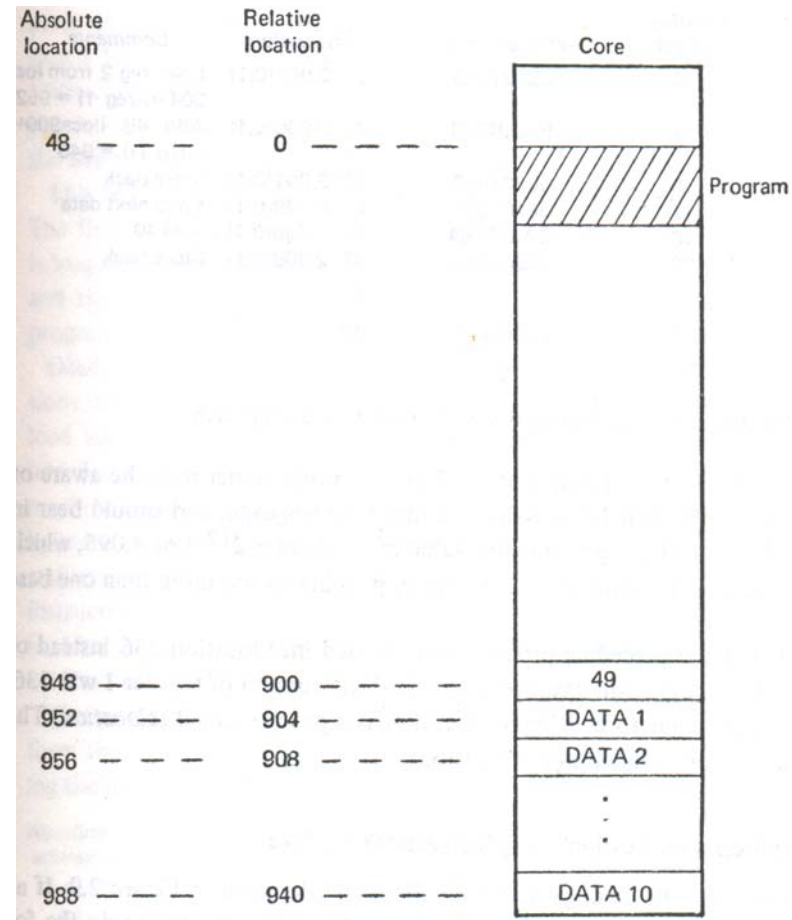



The figure shows a load instruction as a series of 0's and 1's that can be easily represented in the mnemonic code as L 2, 924(0,1). The program that would be used over here adds the number 49 to the contents of 10 adjacent fullwords in memory, under the following assumptions

1. The 10 numbers that are to be added to are in contiguous fullwords beginning at absolute core location 952.
2. The program is in core starting at absolute location 48.
3. The number 49 is a fullword at absolute location 948.
4. Register 1 contains a 48.

***Long way, no looping***

The above assumptions are illustrated in the figure.



| *Absolute address* | *Relative address* | *Hexadecimal* | *Instructions* | *Comments* |
|---|---|---|---|---|
| 48 | 0 | 58201388 | L 2,904(0,1) | Load reg 2 from loc 904+C(reg 1) = 952 |
| 52 | 4 | 5A201384 | A 2,900(0,1) | Add 49 (loc=900+ c(reg 1)) = 948 |
| 56 | 8 | 50201388 | ST 2,904(0,1) | Store back |
| 60 | 12 | 5820138C | L 2,908(0,1) | Load next data |
| 64 | 16 | 5A201384 | A 2,900(0,1) | Add 49 |
| 68 | 20 | 5020138C | ST 2,908(0,1) | Store back |
| ⋮ | ⋮ | ⋮ | ⋮ | |
| 948 | 900 | 00000031 | 49 | |
| 952 | 904 | ⋮ | ⋮ | |

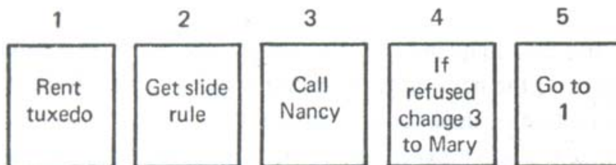
Each instruction over here is a fullword (4 bytes) stating from relative address 0 and absolute address 48. The program does the following.

1. Load R2 with the content of memory location 952=904+c(R1)=904+48.
2. Adds the contents of R2 to the memory location 948=900+c(R1)=900+48.
3. Stores back the content of R2 in the memory location 952.
4. Again loads R2 with the content of memory location 1 byte next to the previous memory location and does step 2 and 3. This process is repeated 10 times.

This process works fine with small amount of data, but poses problem while process a huge amount of data. Let us suppose we need to work on 300 data so a total of 300*3=900 instructions are required. As each instruction takes 4 bytes so a total of 900*4=3600 bytes. Hence the instruction may overlay the data in the core. And further more the data take 300*4= 1200 byte in core, so a total of 4800=3600+1200 bytes of core is used. But using R1 we would be able to access locations $4095=2^{12}-1$ only. So many of the codes are unreached.

***Address modification using instruction as data***

Over here we use the concept of students of M.I.T. Whenever they went for a date, they did the following as shown in the figure below.

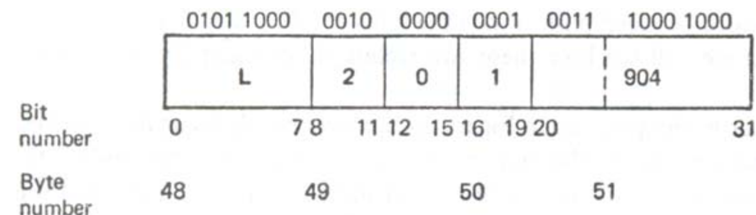



So whenever Nancy refused they replace Nancy with May in step 4 and repeated the process from the start. This process has two programming techniques. First was the instruction step 3 can be treated as data and can be changed. Secondly the process of looping. We will be using these two techniques in our programming. The modified program looks as below.

| *Absolute address* | *Relative address* | *Instructions* | *Comments* |
|---|---|---|---|
| 48 | 0 | L 2,904(0,1) | Add 49 to a number |
| 52 | 4 | A 2,900(0,1) | |
| 56 | 8 | ST 2,904(0,1) | |
| 60 | 12 | L 2,0(0,1) | Increase displacement of load instruction by 4 |
| 64 | 16 | A 2,896(0,1) | |
| 68 | 20 | ST 2,0(0,1) | |
| 72 | 24 | L 2,8(0,1) | Increase displacement of store instruction by 4 |
| 76 | 28 | A 2,896(0,1) | |
| 80 | 32 | ST 2,8(0,1) | |
| | | Branch to relative location 0 nine times | |
| ⋮ | ⋮ | | |
| 944 | 896 | 4 | |
| 948 | 900 | 49 | |
| 952 | 904 | Numbers | |
| ⋮ | ⋮ | | |

The instruction can be explained as below:

1. Load the data at 952 in R2.
2. Add the content of 948 to R2 and store it in R2.
3. Store back the content of R2 in the memory location 952.
4. Load R2 with the instruction at relative address 0.
5. Add content of 896+48=944 i.e. 4 to the content of R2.
6. Store content of R2 back to the relative address 0.
7. Load R2 with the instruction at relative address 8.
8. Add content of 896+48=944 i.e. 4 to the content of R2.
9. Store content of R2 back to the relative address 8.
10. Branch to relative address 0 nine times.




Address modification of the instruction occurs as shown in the figure above.

This process is very simple as rightmost bits of the instructions contains of offsets, hence adding a value to the instruction changes the offset of the instruction there by changing the instruction.

One thing is paramount importance is that, treating instructions as data is a bad programming technique. If this process is employed, it becomes difficult for the programmer to know what were the codes initially. In multiprocessing systems, it violates the concept of pure procedures, which are procedure that do not modify themselves.

***Address modification using index registers***

The previous two processes do not make use of index registers. So we can make use of index register in addressing and the increment factor can be stored in the index register and can be incremented regularly with each iteration. The codes for the process is discussed as follows:

| Absolute address | Relative address | Instructions | | Comments |
|---|---|---|---|---|
| 48 | 0 | SR | 4,4 | Clear register 4 |
| 50 | 2 | L | 2,904(4,1) | Load data element of array |
| 54 | 6 | A | 2,900(0,1) | Add 49 |
| 58 | 10 | ST | 2,904(4,1) | Replace data element |
| 62 | 14 | A | 4,896(0,1) | Add 4 to index register |
| | | Branch back to relative location 2, nine times | | |

The instruction can be explained as below:

1. Clear the content of R4 by subtracting R4 by itself i.e. R4.
2. Load the content of 904+c(4)+c(1)=904+0+48=952 to R2.
3. Add contents of 948 to R2 and store it in R2.
4. Store the content of R2 back to 952.
5. Add content of 896+48=944 i.e. 4 to the content of R4.
6. Branch back to relative location 2, nine times.

In this process R4 is cleared first and used in the address calculation process. Subsequently R4 is incremented each time by 4 for each iteration.

***Looping***

With the use of index register, we have understood the concept of looping. We will be applying looping in two ways, but for this to happen we need to assumptions to be made, that are as follows:

1. Relative location 892 contains a 10.
2. Relative location 888 contains a 1 ( for first method only).

| Absolute address | Relative address | Instructions | | |
|---|---|---|---|---|
| 48 | 0 | SR | 4,4 | |
| 50 | 2 | L | 2,904(4,1) | Add 49 to a number |
| 54 | 6 | A | 2,900(0,1) | |
| 58 | 10 | ST | 2,904(4,1) | |
| 62 | 14 | A | 4,896(0,1) | Add 4 to index register |
| 66 | 18 | L | 3,892(0,1) | Load temp into register 3 |
| 70 | 22 | S | 3,888(0,1) | Subtract 1 |
| 74 | 26 | ST | 3,892(0,1) | Store temp |
| 78 | 30 | BC | 2,2(0,1) | Branch if result is positive (2 denotes a condition code) |
| – | – | – | | |
| – | – | – | | |
| – | – | – | | |
| 936 | 888 | 1 | | |
| 940 | 892 | (Initially 10 - decremented by 1 after each loop) | | |
| 944 | 896 | 4 | | |
| 948 | 900 | 49 | | |
| 952 | 904 | Numbers | | |

First process is similar to that of the process discussed in last technique. Rest of the steps in this process are as follows:

7. Load the content of 892+48=940 i.e. 10(initially) to R3.
8. Subtract the content of 888+48=936 i.e. 1 from the content of R3 and store the result in R3.
9. Store the content of R3 back to 940.
10. Check if the content of 940 is positive.
    a. If positive then send the control back to instruction stored are relative address 2.
    b. Else terminate.

Here BC is a conditional opcode is used, that check for a condition (here the condition is that location 940 is positive) and act accordingly depending on the result of the condition.

The second process used in looping makes use of a conditional opcode BCT, that works in a manner similar to that of BC opcode and in addition to that is decrements the looping factor by one each time it gets executed. Hence the use of BCT opcode in the programming cut short our program by three bytes. Since the process is iterative one so the 3*10 bytes of executable code are reduce by the use of BCT opcode. The code of this process is as in the figure.

| Absolute address | Relative address | Instructions | | |
|---|---|---|---|---|
| 48 | 0 | L | 3,892(0,1) | Load register 3 with 10 |
| 52 | 4 | SR | 4,4 | Clear register 4 |
| 54 | 6 | L | 2,904(4,1) | |
| 58 | 10 | A | 2,900(0,1) | Add 49 to a number |
| 62 | 14 | ST | 2,904(4,1) | |
| 66 | 18 | A | 4,896(0,1) | Add 4 to index register |
| 70 | 22 | BCT | 3,6(0,1) | Subtract one from register 3 and branch to relative location 6 when positive |
| ⋮ | ⋮ | ⋮ | | |
| 940 | 892 | 10 | | |
| 944 | 896 | 4 | | |
| 948 | 900 | 49 | | |
| 952 | 904 | Numbers | | |
| ⋮ | ⋮ | | | |
| 992 | unused | | | |
| ⋮ | | | | |

The code over here are much similar to last to codes except the program starts with a instruction that loads the content of 940 to R3. The process is short is discussed as follows:

1. Load the content of 940 to R3.
2. Clear the content of R4 by subtracting R4 by itself i.e. R4.
3. Load the content of 904+c(4)+c(1)=904+0+48=952 to R2.
4. Add contents of 948 to R2 and store it in R2.
5. Store the content of R2 back to 952.
6. Add content of 896+48=944 i.e. 4 to the content of R4.
7. Decrement the content of 940 and check if it is positive.
    a. If positive then send the control back to instruction stored are relative address 6.
    b. Else terminate.

At last by the use of looping we have reduce the program to 26 bytes of instruction and 52 bytes of data, in contrast to the 120 bytes of instruction and 44 bytes of data in our first attempt. Moreover all the other programs could be placed elsewhere in core, at location 400 rather than 48, for example, and only R1 need to be changed.

**ASSEMBLY LANGUAGE**

The era of programming languages starts with machine language and end in English language, thereby making a move from the language that are best for machine to that are best for programmers.

So far we have machine language and mnemonic machine language. Now we will be using assembly language for the follows reasons (advantages).

1. It is mnemonics, e.g. we use ST instead of the bit stream 0101000 for the store instruction.
2. Addresses are symbolic, not absolute.
3. Reading is easier.
4. Introduction of data to program is easier.

The main drawback of assembly language is that it makes use of an assembler to translate a source program to object code. The assembly language for 360 is much similar to assembly languages that are meant for other machines.

***An assembly language program***

We would write the same program in assembly language, that will be free from the assumptions used as earlier. One of the assumptions states that the program is loaded at absolute address 48. The program in assembly language is as below.

| | Program | | Comments |
|---|---|---|---|
| TEST | START | | Identifies name of program |
| BEGIN | BALR | 15,0 | Set register 15 to the address of the next instruction |
| | USING | BEGIN+2,15 | Pseudo-op indicating to assembler register 15 is base register and its content is address of next instruction |
| | SR | 4,4 | Clear register 4 (set index=0) |
| | L | 3,TEN | Load the number 10 into register 3 |
| LOOP | L | 2,DATA(4) | Load data (index) into register 2 |
| | A | 2,FORTY9 | Add 49 |
| | ST | 2,DATA(4) | Store updated value of data (index) |
| | A | 4,FOUR | Add 4 to register 4 (set index = index+4) |
| | BCT | 3,LOOP | Decrement register 3 by 1, if result non-zero, branch back to loop |
| | BR | 14 | Branch back to caller |
| TEN | DC | F'10' | Constant 10 |
| FOUR | DC | F'4' | Constant 4 |
| FORTY9 | DC | F'49' | Constant 49 |
| DATA | DC | F'1,3,3,3,3,4,5,8,9,0' | Words to be processed |
| | END | | |

It is very difficult to know for any programmer or machine to know the address in the core where the loader will load the program. It is only at the time of execution the stating address of the program is known. Assembly language has a machine code named BALR that sets the address of base register to the next instruction to be executed.

Assembly program has two type of opcode, they are machine and pseudo opcode. The machine opcodes are translated to machine code, while the pseudo opcodes are not. The pseudo instruction only give information to the assembler. The list of opcodes used in the program are as follows:

1. USING is a pseudo code the indicates to the assembler which general purpose register to use as a base and what its contents will be. As we do not have any specific general register acting as the base register, so it becomes necessary to inform indicate a base register for the program. Because the address are relative so by the knowledge of base and offset the program can be easily be located and executed.

2. BALR is machine opcode that load a register with the next address and branch to the address in the second field. Since second operand is 0 so the control will go to the next instruction.

3. START is a pseudo opcode that tell the assembler where the beginning of the program is and allows the user to give a name to the program. In this case the name is TEST.

4. END is a pseudo code that tells the assembler that the last card of the program has been reached.

5. Symbolic names like TEN, DATA(4), FORTY9 etc, are use in assembly language to reduce the burden of calculating the address of these data fields. DC are pseudo opcode used to specify the symbolic names to the constants.

6. BR 14, the last machine opcode is used to branch to the location whose address is in general purpose register 14. By convention, calling programs leave their return address in register 14.

***Literals***
The same program is repeated by using literals, that are mechanisms where by the assembler creates data areas for the programmer, containing constants he requests.

=F'10', =F'49', =F'4' are the literal which would be result in the creation of a data area containing 10, 49 and 4 and replacement of the literal operand with the address of the data it describes.

L 3, =F'10' is translated by the assembler to point to a full word that contains a 10. Generally the assembler keep track of the literal with the help of a literal table. This table will contain all the constants that have been requested through the use of literal. A pseudo opcode LTORG place the literal at an earlier location. This is required because, the program may be using 10000 data and it become difficult for the offset of the load instruction to reach the literal at the end of the program.

The program by the use of literal looks as follows:

```
TEST      START   0
BEGIN     BALR    BASE,0
          USING   BEGIN+2,BASE
          SR      4,4
          L       3, = F'10'
LOOP      L       2,DATA (4)
          A       2, = F'49'
          ST      2,DATA (4)
          A       4, = F'4'
          BCT     3, *-16
          BR      14
          LTORG
DATA      DC      F'1,3,3,3,3,4,5,8,9,0'
BASE      EQU     15
          END
```

In the program we make use of '*-16' with the BCT opcode. Star in a mnemonic means here. The expression *-16 refers to the address of the present instruction minus 16 locations, which is a simple loop. This type of programming is not a good practice.

The EQU pseudo code is used to assign values to the symbols. Here BASE is assigned 15. EQU can also take arithmetic expression as its operand.

The program on being translated to machine code do not generate any code for the pseudo code START and USING.