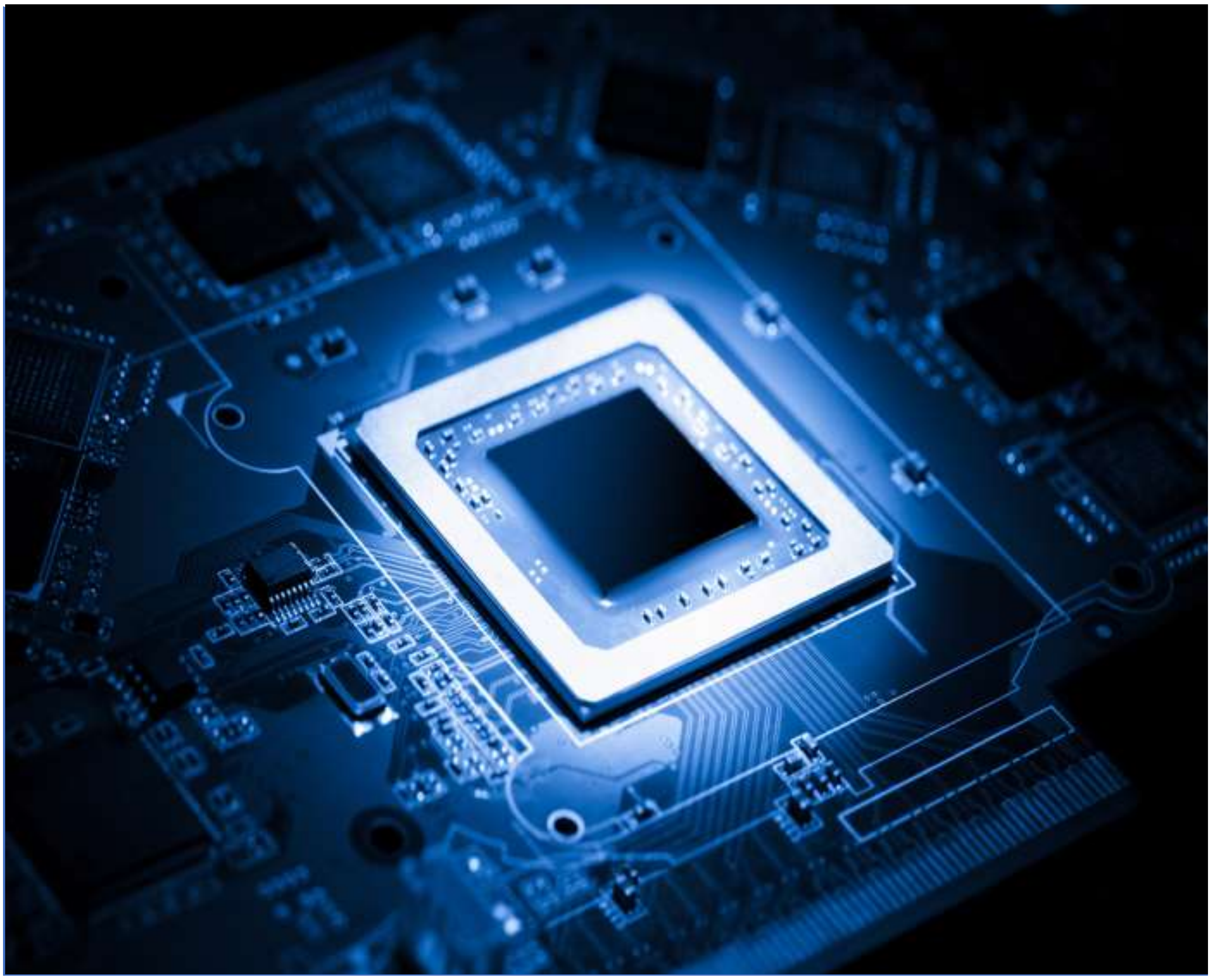
COMPUTER ORGANIZATION



Chapter Three

Instruction Set Architecture and Design

1. INSTRUCTION MNEMONICS AND SYNTAX

Assembly language is the symbolic form of machine language.

Assembly programs are written with short abbreviations called mnemonics.

A mnemonic is an abbreviation that represents the actual machine instruction.

Assembly language programming is the writing of machine instructions in mnemonic form, where each machine instruction (binary or hex value) is replaced by a mnemonic.

An assembly program consists of a sequence of assembly statements, where statements are written one per line. Each line of an assembly program is split into the following four fields: label, operation code (opcode), operand, and comments.

Label (Optional)	Operation Code (Required)	Operand (Required in some instructions)	Comment (Optional)

Labels are used to provide symbolic names for memory addresses. A label is an identifier that can be used on a program line in order to branch to the labeled line.

The operation code (opcode) field contains the symbolic abbreviation of a given operation.

The operand field consists of additional information or data that the opcode requires. The operand field may be used to specify constant, label, immediate data, register, or an address.

The comments field provides a space for documentation to explain what has been done for the purpose of debugging and maintenance.

```
START     LD X     \ copy the contents of location X into AC
```

The label of the instruction LD X is START, which means that it is the memory address of this instruction.

2. MEMORY LOCATIONS AND OPERATIONS

The (main) memory can be modeled as an array of millions of adjacent cells, each capable of storing a binary digit (bit), having value of 1 or 0.

These cells are organized in the form of groups of fixed number, say n, of cells that can be dealt with as an atomic entity.

An entity consisting of 8 bits is called a byte. In many systems, the entity consisting of n bits that can be stored and retrieved in and out of the memory using one basic memory operation is called a word (the smallest addressable entity).

Typical size of a word ranges from 16 to 64 bits. It is, however, customary to express the size of the memory in terms of bytes. For example, the size of a typical memory of a personal computer is 256 Mbytes, that is, $256 \times 2^{20} = 2^{28}$ bytes.

there are two basic memory operations. These are the **memory write** and **memory read** operations.

A memory write operation a word is stored into a memory location whose address is specified.

A memory read operation a word is read from a memory location whose address is specified.

Typically, memory read and memory write operations are performed by the central processing unit (CPU).

Three basic steps are needed in order for the CPU to perform a write operation into a specified memory location:

1. The word to be stored into the memory location is first loaded by the CPU into a specified register, called the memory data register (MDR).
2. The address of the location into which the word is to be stored is loaded by the CPU into a specified register, called the memory address register (MAR).
3. A signal, called write, is issued by the CPU indicating that the word stored in the MDR is to be stored in the memory location whose address in loaded in the MAR.

Similar to the write operation, three basic steps are needed in order to perform a memory read operation:

1. The address of the location from which the word is to be read is loaded into the MAR.
2. A signal, called read, is issued by the CPU indicating that the word whose address is in the MAR is to be read into the MDR.
3. After some time, corresponding to the memory delay in reading the specified word, the required word will be loaded by the memory into the MDR ready for use by the CPU.

It is worth mentioning that the MDR and the MAR are registers used exclusively by the CPU and are not accessible to the programmer.

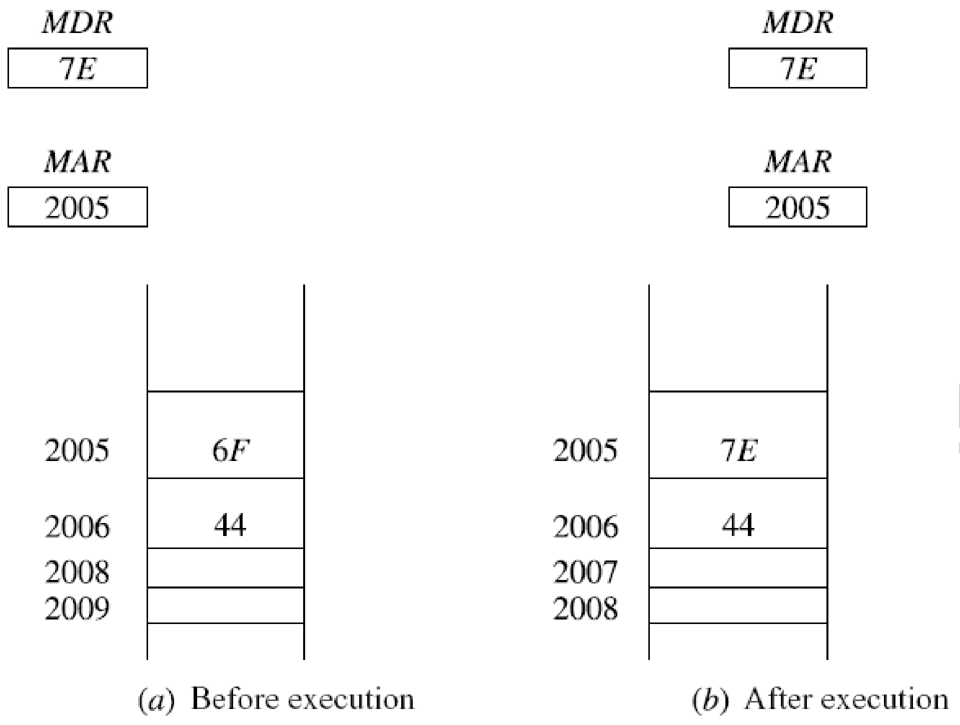


Illustration of the memory write operation

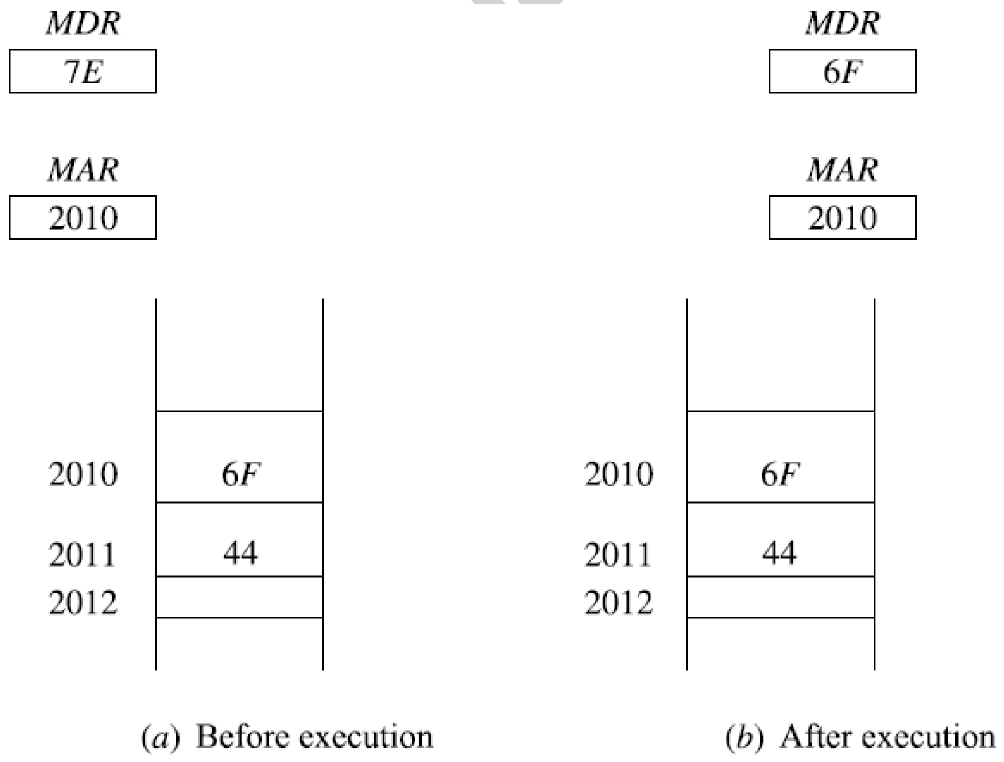


Illustration of the memory read operation

❖ The Fetch-Decode-Execute Cycle

The *fetch-decode-execute cycle* represents the steps that a computer follows to run a program.

Fetches: transfers instruction from main memory to the instruction register.

Decodes: determines the opcode and fetches any data necessary to carry out the instruction.

Executes: performs the operation(s) indicated by the instruction.

Notice that a large part of this cycle is spent copying data from one location to another. When a program is initially loaded, the address of the first instruction must be placed in the PC. The steps in this cycle are listed below. Note that Steps 1 and 2 make up the fetch phase, Step 3 makes up the decode phase, and Step 4 is the execute phase.

1- Copy the contents of the PC to the MAR: MAR 🡐 PC.

2- Go to main memory and fetch the instruction found at the address in the MAR, placing this instruction in the IR; increment PC by 1 (PC now points to the next instruction in the program):

IR 🡐 M[MAR]

PC 🡐 PC+1.

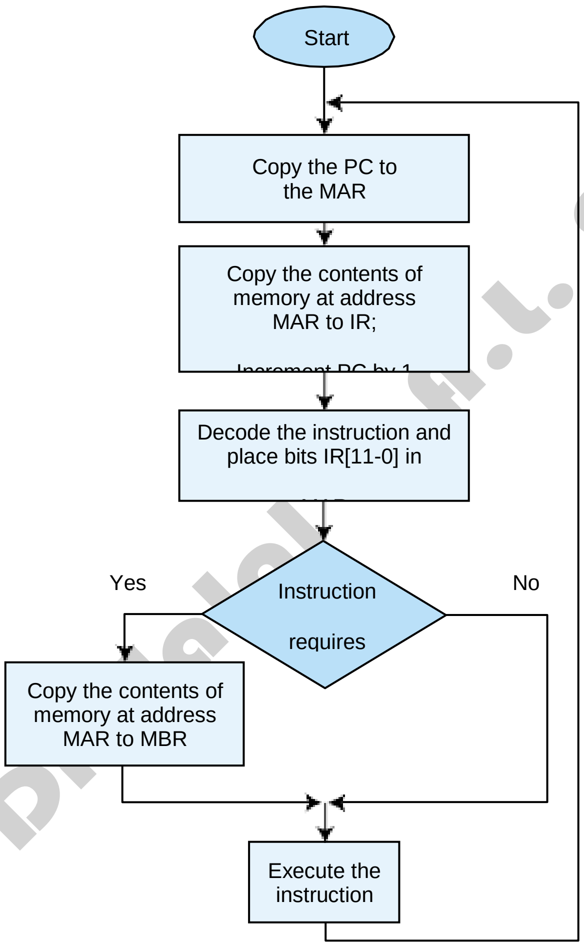
Note: Because MARIE is word-addressable, the PC is incremented by one, which results in the next word's address occupying the PC. If MARIE were byte- addressable, the PC would need to be incremented by 2 to point to the address of the next instruction, because each instruction would require two bytes. On a byte-addressable machine with 32-bit words, the PC would need to be incremented by 4.

3- Copy the rightmost 12 bits of the IR into the MAR; decode the leftmost four bits to determine the opcode, MAR 🡐 IR[11–0], and decode IR[15–12].

4- If necessary, use the address in the MAR to go to memory to get data, placing the data in the MBR, and then execute the instruction

MBR ← M[MAR]

and execute the actual instruction


Start
Copy the PC to the MAR
Copy the contents of memory at address MAR to IR;
Increment PC by 1
Decode the instruction and place bits IR[11-0] in
MAR
Instruction
requires
Yes
No
Copy the contents of memory at address MAR to MBR
Execute the instruction

The Fetch-Decode-Execute Cycle

3. ADDRESSING MODES

Information involved in any operation performed by the CPU needs to be addressed. In computer terminology, such information is called the **operand**.

Any instruction issued by the processor must carry at least two types of information. These are the operation to be performed, encoded in what is called the **op-code field**, and the address information of the operand on which the operation is to be performed, encoded in what is called the **address field**.

Instructions can be classified based on the number of operands as: *three-address*, *two-address*, *one-and-half-address*, *one-address*, and *zero-address*.

❖ Three-Address Machines

In three-address machines, instructions carry all three addresses explicitly. Most current processors use three addresses.

A = B + C * D - E + F + A

is converted to the following code:

```
MULT  T,C,D    ; T = C*D
ADD  T,T,B     ; T = B + C*D
SUB   T,T,E    ; T = B + C*D - E
ADD  T,T,F     ; T = B + C*D - E + F
```

❖ Two-Address Machines

In two-address machines, one address doubles as a source and destination. Usually, we use *dest* to indicate that the address is used for destination. But you should note that this address also supplies one of the source operands.

A = B + C * D - E + F + A

is converted to the following code:

```
LOAD  T,C      ; T = C
MULT  T,D      ; T = C*D
ADD  T,B       ; T = B + C*D
SUB  T,E       ; T = B + C*D - E
ADD  T,F       ; T = B + C*D - E + F
ADD  A,T       ; A = B + C*D - E + F + A
```

❖ **One-Address Machines**

In the early machines, when memory was expensive and slow, a special set of registers was used to provide one of the input operands as well as to receive the result of the operation.

Because of this, these registers are called the **accumulators**. In most machines, there is just a single accumulator register. This kind of design, called the **accumulator machines**.

A = B + C * D - E + F + A

is converted to the following code:

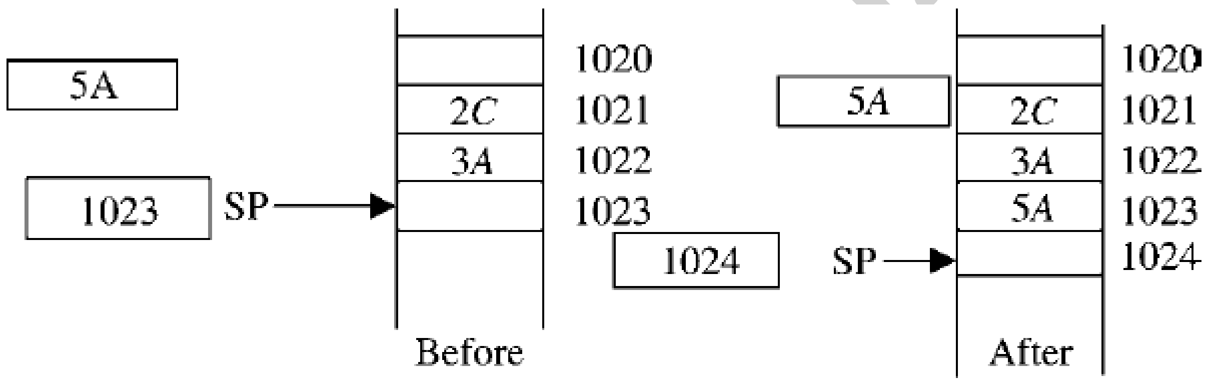
```
LOAD  C        ;Acc←C
MULT  D        ;Acc= Acc * D
ADD   B        ;Acc=Acc + B
SUB   E        ;Acc=Acc – E
ADD   F        ;Acc=Acc + F
ADD   A        ;Acc=Acc + A
```

❖ Zero-Address Machines

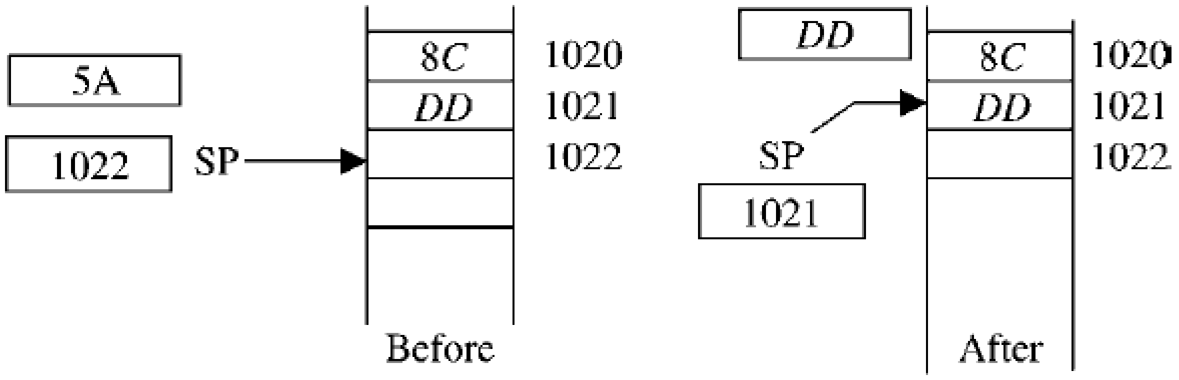
In zero-address machines, the locations of both operands are assumed to be at a default location. These machines use the stack as the source of the input operands and the result goes back into the stack.

Stack is a LIFO (*last-in–first-out*) data structure that all processors support.

A stack is a data organization mechanism in which the last data item stored is the first data item retrieved. Two specific operations can be performed on a stack. These are the **push** and the **pop** operations.



Stack *push* operation

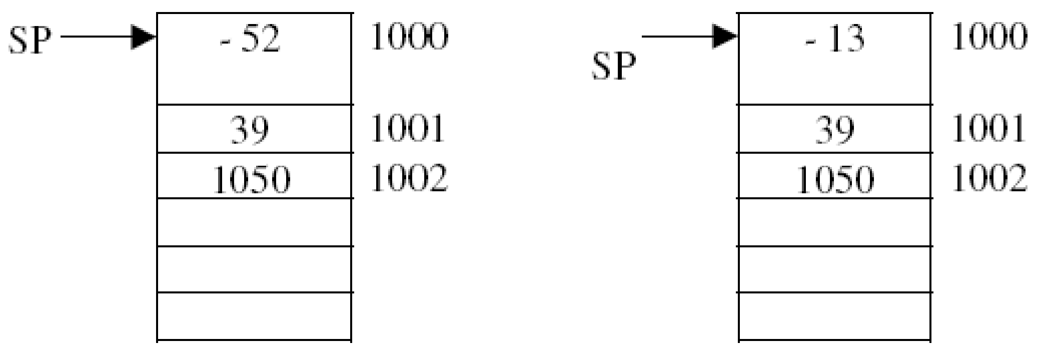


Stack *pop* operation

As can be seen, a specific register, called the **stack pointer** (SP), is used to indicate the stack location that can be addressed. In the stack **push** operation, the SP value is used to indicate the location (called the top of the stack) in which the value (5A) is to be stored (in this case it is location 1023). After storing (pushing) this value the SP is incremented to indicate to location 1024.

In the stack **pop** operation, the SP is first decremented to become 1021. The value stored at this location (DD in this case) is retrieved (popped out) and stored in the shown register.

Different operations can be performed using the stack structure. Consider, for example, an instruction such as *ADD* *(SP)+,(SP)*. The instruction adds the contents of the stack location pointed to by the SP to those pointed to by the *SP+1* and stores the result on the stack in the location pointed to by the current value of the SP.



Addition using the stack

All operations on this type of machine assume that the required input operands are the top two values on the stack. The result of the operation is placed on top of the stack.

A = B + C * D - E + F + A

is converted to the following code:

```
PUSH   A        ;Stack ←A
PUSH   F        ;Stack ←F
PUSH   E        ;Stack ←E
PUSH   B        ;Stack ←B
PUSH   D        ;Stack ←D
PUSH   C        ;Stack ←C
MULT
ADD
```

```
SUB
ADD
ADD
POP  A          ;A← TopS
```

The different ways in which operands can be addressed are called the **addressing modes**. Addressing modes differ in the way the address information of operands is specified.

a. Immediate Mode

According to this addressing mode, the value of the operand is (immediately) available in the instruction itself.

For example, the case of loading the decimal value 1000 into a register *Ri*. This operation can be performed using an instruction such as the following: *LOAD #1000, Ri*. In this instruction, the operation to be performed is to load a value into a register.

It is customary to prefix the operand by the special character (#)

b. Direct (Absolute) Mode

According to this addressing mode, the address of the memory location that holds the operand is included in the instruction.

For example, the case of loading the value of the operand stored in memory location 1000 into register *Ri*. This operation can be performed using an instruction such as *LOAD 1000, Ri*. In this instruction, the source operand is the value stored in the memory location whose address is 1000, and the destination is the register Ri.

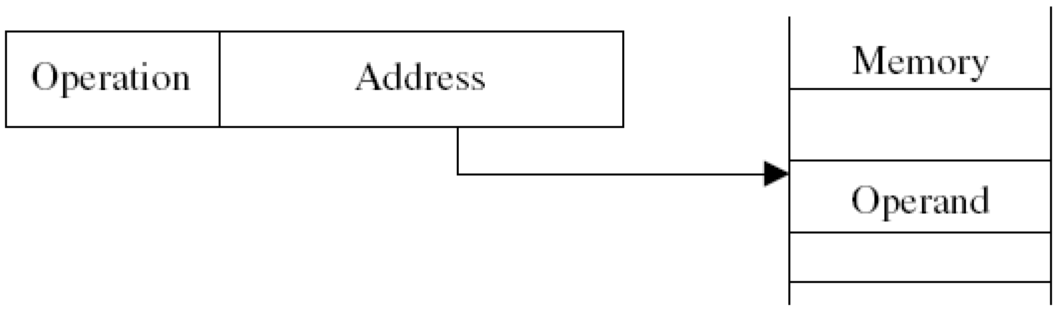


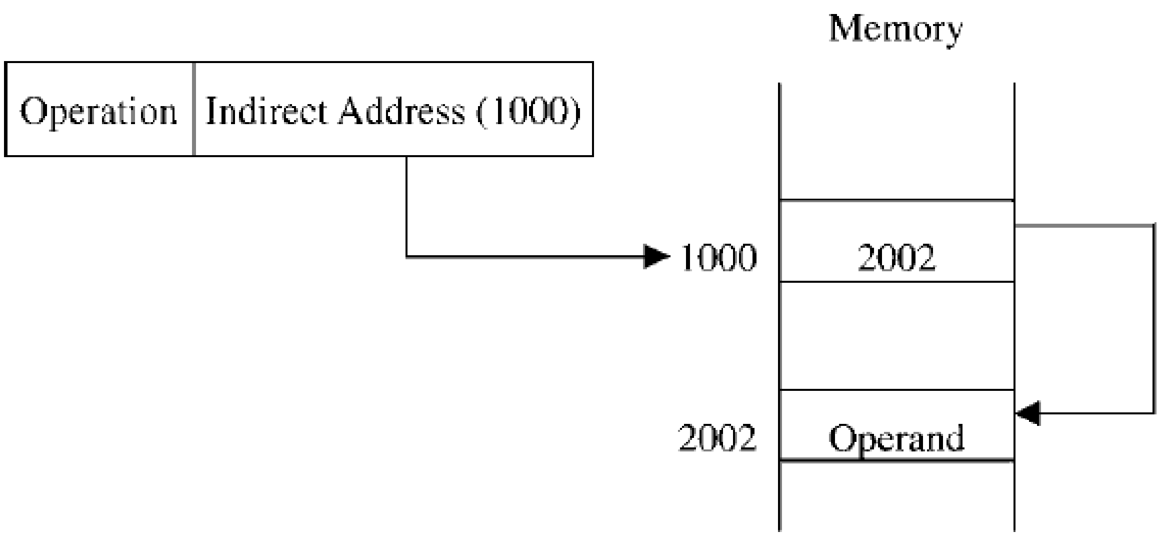
Illustration of the direct addressing mode

c. Indirect Mode

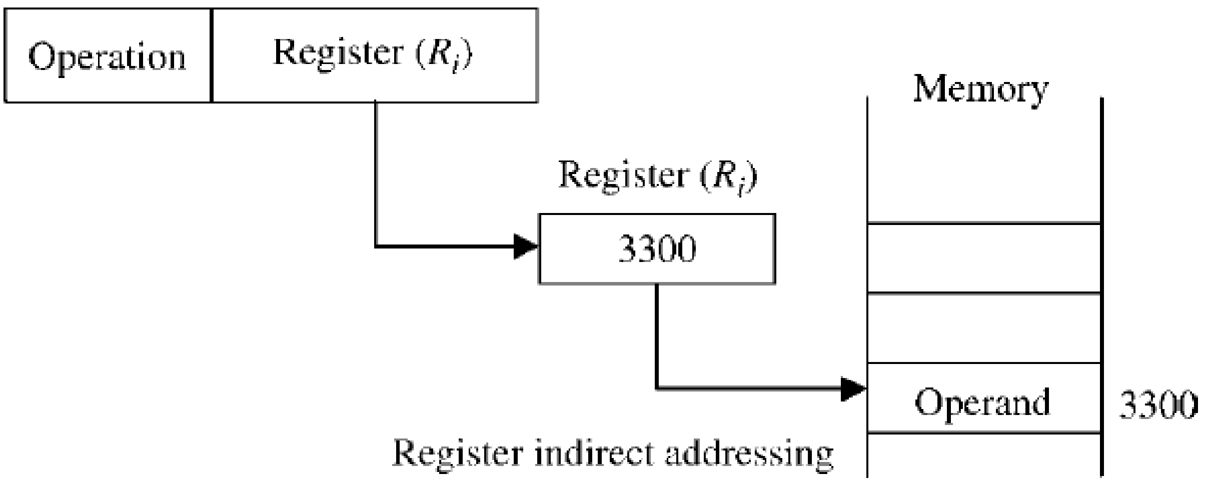
In the indirect mode, what is included in the instruction is not the address of the operand, but rather a name of a register or a memory location that holds the (effective) address of the operand. In order to indicate the use of indirection in the instruction, it is customary to include the name of the register or the memory location in parentheses.

For example, the instruction *LOAD (1000), Ri*. This instruction has the memory location 1000 enclosed in parentheses, thus indicating indirection. The meaning of this instruction is to load register *Ri* with the contents of the memory location whose address is stored at memory address 1000.

Because indirection can be made through either a register or a memory location, therefore, we can identify two types of indirect addressing. These are **register indirect addressing**, if a register is used to hold the address of the operand, and **memory indirect addressing**, if a memory location is used to hold the address of the operand.



Memory indirect addressing



Register indirect addressing

Illustration of the indirect addressing mode

d. Indexed Mode

In this addressing mode, the address of the operand is obtained by adding a constant to the content of a register, called the *index register*.

For example, the instruction *LOAD X(R_{ind}), Ri*. This instruction loads register *Ri* with the contents of the memory location whose address is the sum of the contents of register R_{ind} and the value *X*.

Index addressing is indicated in the instruction by including the name of the index register in parentheses and using the symbol X to indicate the constant to be added.

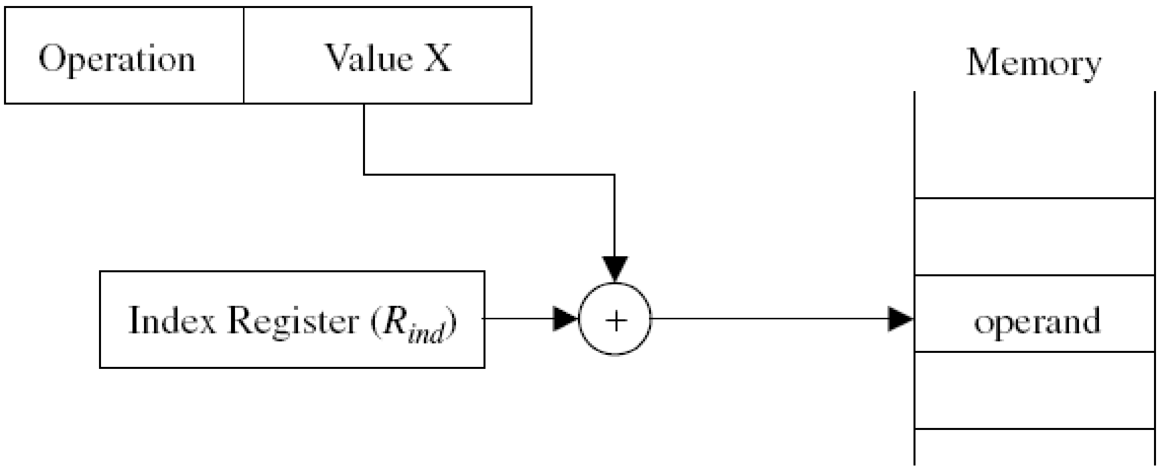


Illustration of the indexed addressing mode

e. Relative Mode

Recall that in indexed addressing, an index register, R_{ind}, is used. Relative addressing is the same as indexed addressing except that the *program counter* (PC) replaces the index register.

For example, the instruction *LOAD X(PC), Ri* loads register *Ri* with the contents of the memory location whose address is the sum of the contents of the program counter (PC) and the value X.

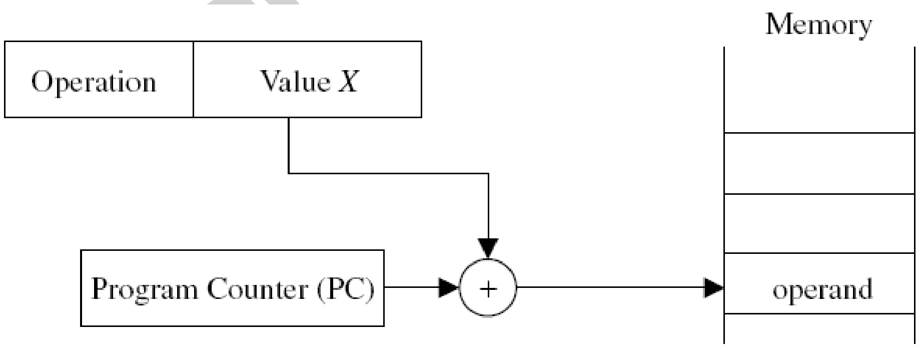


Illustration of relative addressing mode

f. Autoincrement Mode

This addressing mode is similar to the register indirect addressing mode in the sense that the effective address of the operand is the content of a register, call it the *autoincrement register*, that is included in the instruction.

However, with autoincrement, the content of the autoincrement register is automatically incremented after accessing the operand.

for example, the instruction *LOAD* (*R*$_{auto}$)+, *Ri*. This instruction loads register *Ri* with the operand whose address is the content of register *R*$_{auto}$. After loading the operand into register *Ri*, the content of register *R*$_{auto}$ is incremented, pointing for example to the next item in a list of items.

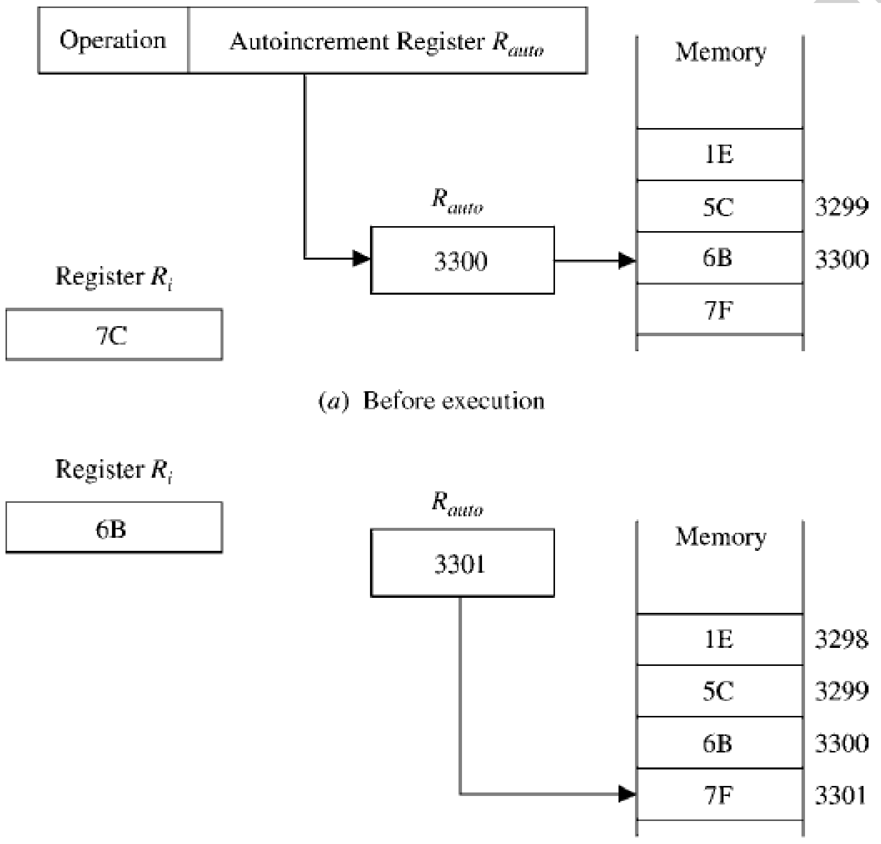


Illustration of the autoincrement addressing mode

g. Autodecrement Mode

Similar to the autoincrement, the autodecrement mode uses a register to hold the address of the operand.

However, in this case the content of the autodecrement register is first decremented and the new content is used as the effective address of the operand.

For example, the instruction *LOAD* *-(R$_{auto}$)*, *Ri*. This instruction decrements the content of the register *R$_{auto}$* and then uses the new content as the effective address of the operand that is to be loaded into register *Ri*.

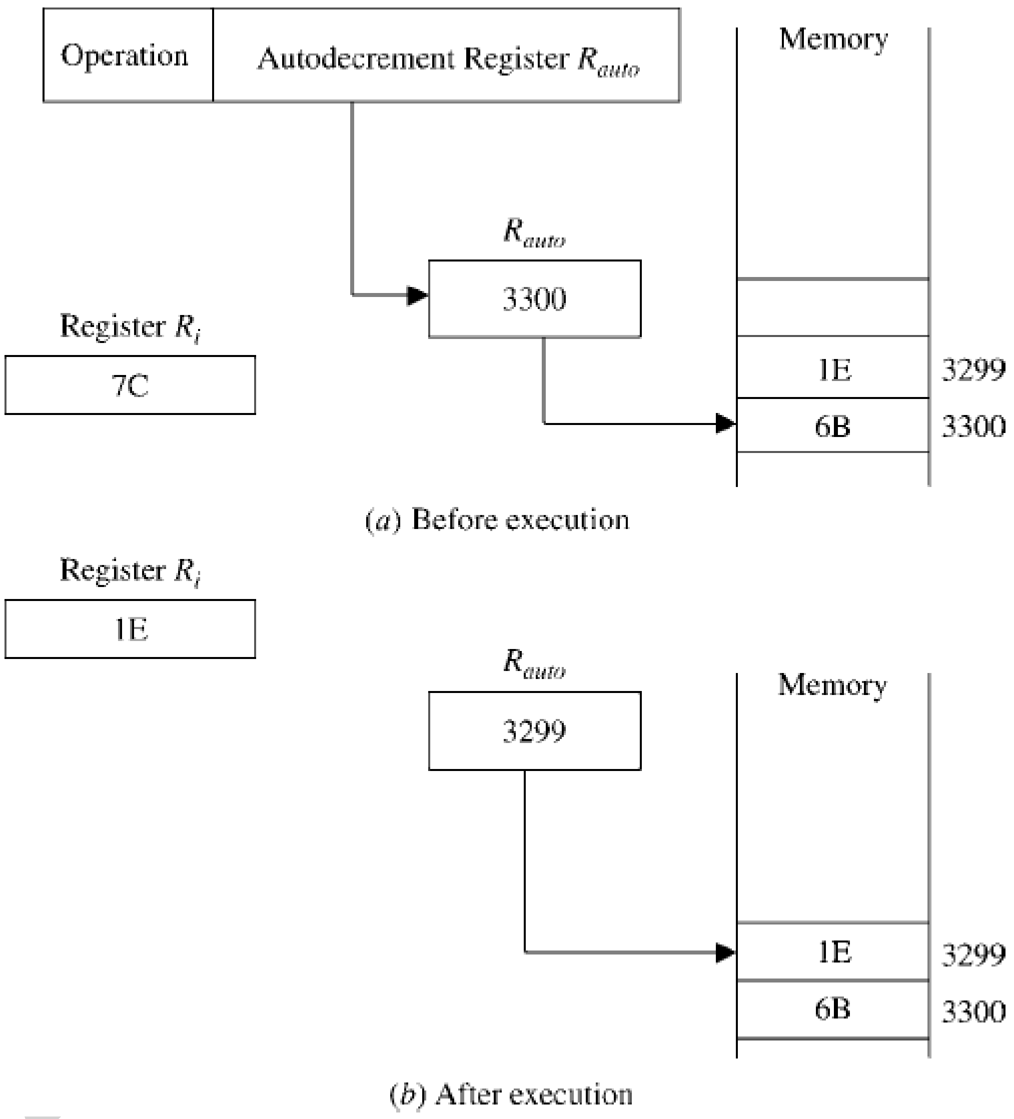


Illustration of the autodecrement addressing mode

Addressing mode	Definition	Example	Operation
Immediate	Value of operand is included in the instruction	*load* #1000, R_i	$R_i \leftarrow 1000$
Direct (Absolute)	Address of operand is included in the instruction	*load* 1000, R_i	$R_i \leftarrow M[1000]$
Register indirect	Operand is in a memory location whose address is in the register specified in the instruction	*load* (R_j), R_i	$R_i \leftarrow M[R_j]$
Memory indirect	Operand is in a memory location whose address is in the memory location specified in the instruction	*load* (1000), R_i	$R_i \leftarrow M[1000]$
Indexed	Address of operand is the sum of an index value and the contents of an index register	*load* $X(R_{ind})$, R_i	$R_i \leftarrow M[R_{ind} + X]$
Relative	Address of operand is the sum of an index value and the contents of the program counter	*load* $X(PC)$, R_i	$R_i \leftarrow M[PC + X]$
Autoincrement	Address of operand is in a register whose value is incremented after fetching the operand	*load* $(R_{auto})+$, R_i	$R_i \leftarrow M[R_{auto}]$ $R_{auto} \leftarrow R_{auto} + 1$
Autodecrement	Address of operand is in a register whose value is decremented before fetching the operand	*load* $-(R_{auto})$, R_i	$R_{auto} \leftarrow R_{auto} - 1$ $R_i \leftarrow M[R_{auto}]$

4. INSTRUCTION TYPES

Instructions can in general be classified as in the following Subsections:

❖ Data Movement Instructions

Data movement instructions are used to move data among the different units of the machine.

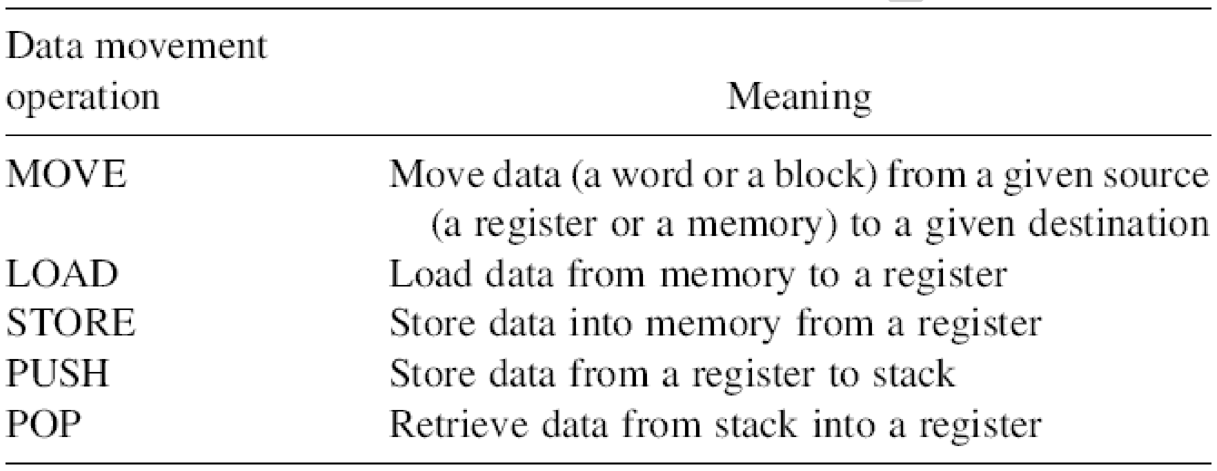
A simple register to register movement of data can be made through the instruction

MOVE *Ri, Rj*

This instruction moves the content of register *Ri* to register *Rj*. The effect of the instruction is to override the contents of the (destination) register *Rj* without changing the contents of the (source) register *Ri*.

Data movement instructions include those used to move data to (from) registers from (to) memory. These instructions are usually referred to as the *load* and *store* instructions, respectively. Examples of the two instructions are

LOAD 25838, *Rj*
STORE *Ri*, 1024



Data movement operation	Meaning
MOVE	Move data (a word or a block) from a given source (a register or a memory) to a given destination
LOAD	Load data from memory to a register
STORE	Store data into memory from a register
PUSH	Store data from a register to stack
POP	Retrieve data from stack into a register

❖ **Arithmetic and Logical Instructions**

Arithmetic and logical instructions are those used to perform arithmetic and logical manipulation of registers and memory contents.

Examples of arithmetic instructions include the *ADD* and *SUBTRACT* instructions. These are

ADD *R1, R2, R0*

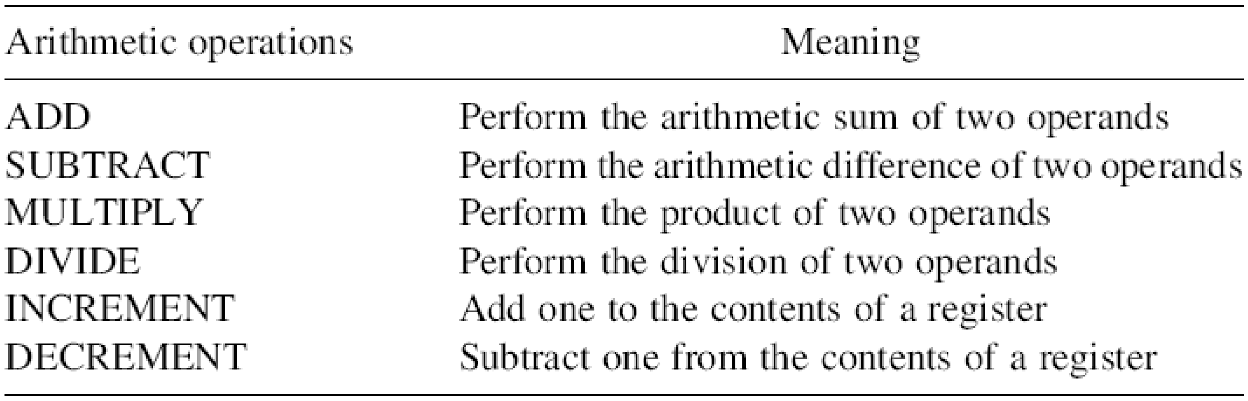
SUBTRACT *R1, R2, R0*

The first instruction adds the contents of source registers R1 and R2 and stores the result in destination register R0.

The second instruction subtracts the contents of the source registers R1 and R2 and stores the result in the destination register R0.

The contents of the source registers are unchanged by the ADD and the SUBTRACT instructions.

Some machines have *MULTIPLY* and *DIVIDE* instructions. These two instructions are expensive to implement and could be substituted by the use of repeated addition or repeated subtraction. Therefore, most modern architectures do not have *MULTIPLY* or *DIVIDE* instructions on their instruction set.



Arithmetic operations	Meaning
ADD	Perform the arithmetic sum of two operands
SUBTRACT	Perform the arithmetic difference of two operands
MULTIPLY	Perform the product of two operands
DIVIDE	Perform the division of two operands
INCREMENT	Add one to the contents of a register
DECREMENT	Subtract one from the contents of a register

Logical instructions are used to perform logical operations such as *AND, OR, SHIFT, COMPARE,* and *ROTATE*. As the names indicate, these instructions perform, respectively, and, or, shift, compare, and rotate operations on register or memory contents.

Logical operation	Meaning
AND	Perform the logical ANDing of two operands
OR	Perform the logical ORing of two operands
EXOR	Perform the XORing of two operands
NOT	Perform the complement of an operand
COMPARE	Perform logical comparison of two operands and set flag accordingly
SHIFT	Perform logical shift (right or left) of the content of a register
ROTATE	Perform logical shift (right or left) with wraparound of the content of a register

❖ Sequencing Instructions

Control (sequencing) instructions are used to change the sequence in which instructions are executed.

They take the form of *CONDITIONAL BRANCHING (CONDITIONAL JUMP), UNCONDITIONAL BRANCHING (JUMP)*, or *CALL* instructions.

A common characteristic among these instructions is that their execution changes the program counter (*PC*) value. The change made in the *PC* value can be unconditional.

On the other hand, the change made in the *PC* by the branching instruction can be conditional based on the value of a specific *flag*.

Examples of these flags include the *Negative (N), Zero (Z), Overflow (V), and Carry (C)*. These flags represent the individual bits of a specific register, called the *CONDITION CODE (CC) REGISTER*. The values of flags are set based on the results of executing different instructions.

Flag name	Meaning
Negative (N)	Set to 1 if the result of the most recent operation is negative, it is 0 otherwise
Zero (Z)	Set to 1 if the result of the most recent operation is 0, it is 0 otherwise
Overflow (V)	Set to 1 if the result of the most recent operation causes an overflow, it is 0 otherwise
Carry (C)	Set to 1 if the most recent operation results in a carry, it is 0 otherwise

For example, the following group of instructions.

```
        LOAD        #100, R1
Loop:   ADD          (R2) , R0
        DECREMENT       R1
        BRANCH-IF-GREATER-THAN Loop
```

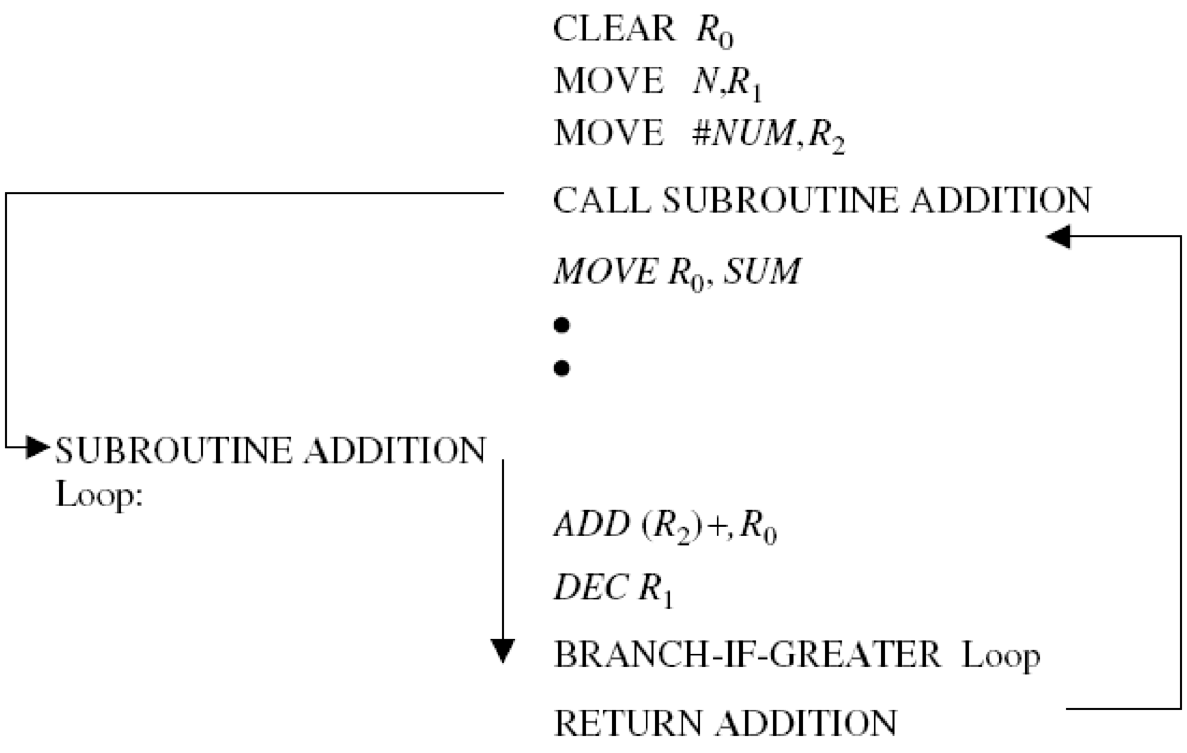
The fourth instruction is a conditional branch instruction, which indicates that if the result of decrementing the contents of register *R1* is greater than zero, that is, if the *Z* flag is not set, then the next instruction to be executed is that labeled by Loop. It should be noted that conditional branch instructions could be used to execute program loops.

The *CALL* instructions are used to cause execution of the program to transfer to a subroutine.

A *CALL* instruction has the same effect as that of the *JUMP* in terms of loading the PC with a new value from which the next instruction is to be executed.

However, with the *CALL* instruction the incremented value of the *PC* (*to point to the next instruction in sequence*) is pushed onto the stack.

Execution of a *RETURN* instruction in the subroutine will load the *PC* with the popped value from the stack. This has the effect of resuming program execution from the point where branching to the subroutine has occurred.



```
                                   CLEAR  R0
                                   MOVE   N,R1
                                   MOVE   #NUM,R2
                                   CALL SUBROUTINE ADDITION
                                   MOVE R0, SUM
                                   •
                                   •
SUBROUTINE ADDITION
Loop:
                                   ADD (R2)+,R0
                                   DEC R1
                                   BRANCH-IF-GREATER  Loop
                                   RETURN ADDITION
```

Transfer of control operation	Meaning
BRANCH-IF-CONDITION	Transfer of control to a new address if condition is true
JUMP	Unconditional transfer of control
CALL	Transfer of control to a subroutine
RETURN	Transfer of control to the caller routine

❖ Input/output Instructions

Input and output instructions (*I/O instructions*) are used to transfer data between the computer and peripheral devices.

The two basic I/O instructions used are the *INPUT* and *OUTPUT* instructions. The INPUT instruction is used to transfer data from an input device to the processor.

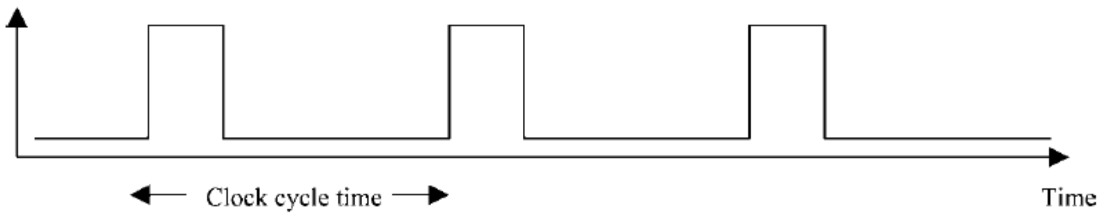
]

H.W. Consider a computer that has a number of registers such that the three registers R0 = 1500, R1 = 4500, and R2 1000. Show the effective address of memory and the registers' contents in each of the following instructions (assume that all numbers are decimal).

(a) ADD	*(R0), R2*
(b) SUBTRACT	*2 (R1), R2*
(c) MOVE	*500(R0), R2*
(d) LOAD	*#5000, R2*
(e) STORE	*R0, 100(R2)*

5. PERFORMANCE MEASURES

Performance analysis should help answering questions such as *how fast can a program be executed using a given computer?* In order to answer such a question, we need to determine the time taken by a computer to execute a given job. We define the clock cycle time as the time between two consecutive rising (trailing) edges of a periodic clock signal.



Clock signal

Clock cycles allow counting unit computations, because the storage of computation results is synchronized with rising (trailing) clock edges. The time required to execute a job by a computer is often expressed in terms of clock cycles.

T*t*: - the clock time

$$Tt = 1/f.$$

Where *f* is CPU clock frequency

Each instruction has different number of clock cycle. The CPU can do one of the following:

- Fetch instruction (3 T)

  T1: PC$\rightarrow$MAR

  T2: WMFC (Wait Memory Function Complete)

  T3: ID$\leftarrow$MDR

- Read data from memory (3 T)

  T1: address$\rightarrow$MAR

  T2: WMFC

  T3: Register$\leftarrow$MDR

- Write data to memory (3 T)

  T1: address$\rightarrow$MAR

  T2: Register$\rightarrow$MDR

  T3: WMFC

Examples:

MOV AL, BL, this instruction is 2 bytes, and need 2 clock cycle, 6 T:

1- Fetch opcode
2- Read operand

MOV AX, [2000], this instruction is 4 bytes, and need 4 clock cycle, 12 T:

1- Fetch opcode
2- Read operand
3- Read 1st data
4- Read 2nd data

These examples assume that the memory time is equal to CPU time. If the memory time is slower than CPU time, that mean we need to wait memory in every access to it (TW).

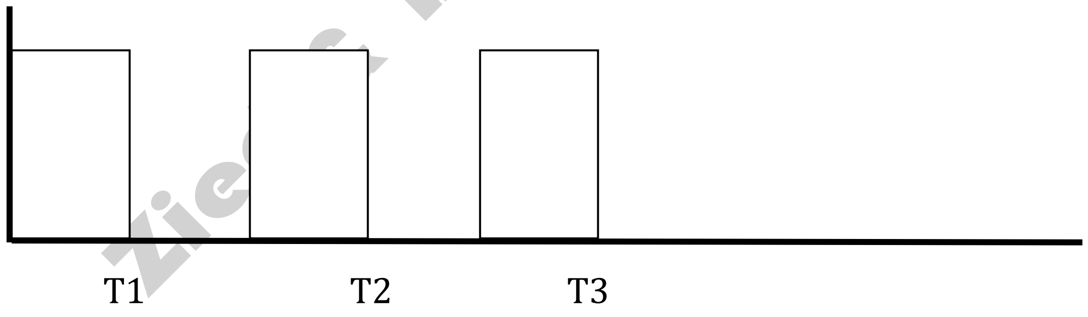
Suppose that the memory time is double time for CPU, that mean any access to memory need one (TW), for the same examples:

MOV AL, BL, this instruction is 2 bytes, and need 4 clock cycle, 8 T:

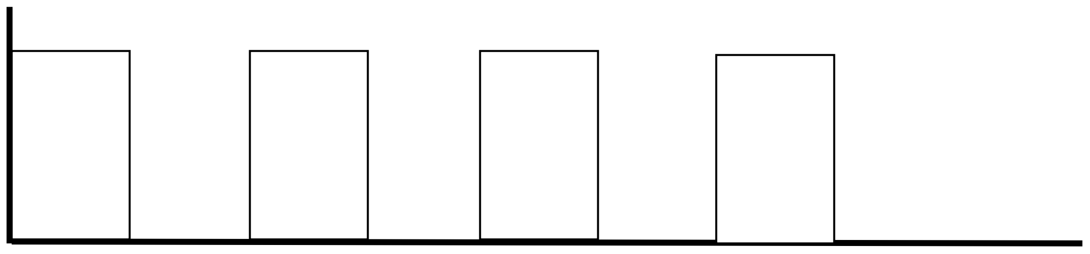
1- Fetch opcode
2- Read operand

MOV AX, [2000], this instruction is 4 bytes, and need 4 clock cycle, 16 T:

1- Fetch opcode
2- Read operand
3- Read 1st data
4- Read 2nd data



Fetch clock cycle without wait



Fetch clock cycle with wait

Example:

Compute time required to execute the instruction (MOV AX, BX) in CPU with f= 1MHz.

Sol.

Tt= 1/f = 1/ 10^6 = 10^{-6} = 1ɱs

MOV AX, BX is 2 bytes, 2 clock cycle: Fetch (3 T) and Read (3 T)

Execution time = No. of T × Tt= 6 × 1= 6 ɱs

Example:

Compute time required to execute the instruction (MOV AX, BX) in CPU with f= 1MHz and memory time is 1.5ɱs.

Sol.

Tt= 1/f = 1/ 10^6 = 10^{-6} = 1ɱs

Because memory is slower than CPU then each access to memory need 1 TW

MOV AX, BX is 2 bytes, 2 clock cycle: Fetch (4 T) and Read (4 T)

Execution time = No. of T × Tt= 8× 1= 8 ɱs