UNIT – II **15 Periods**

Instruction set architecture of a CPU: Memory Locations and Addresses – Memory Operations - Instructions and Instruction Sequencing - Addressing Modes – Assembly Language - Stacks - Subroutines - Additional Instructions - CISC Instruction Sets.

Introduction to x86 architecture: The Intel IA-32 Architecture: Memory Organization - Register Structure - Addressing Modes – Instruction Set – Interrupts and Exceptions.

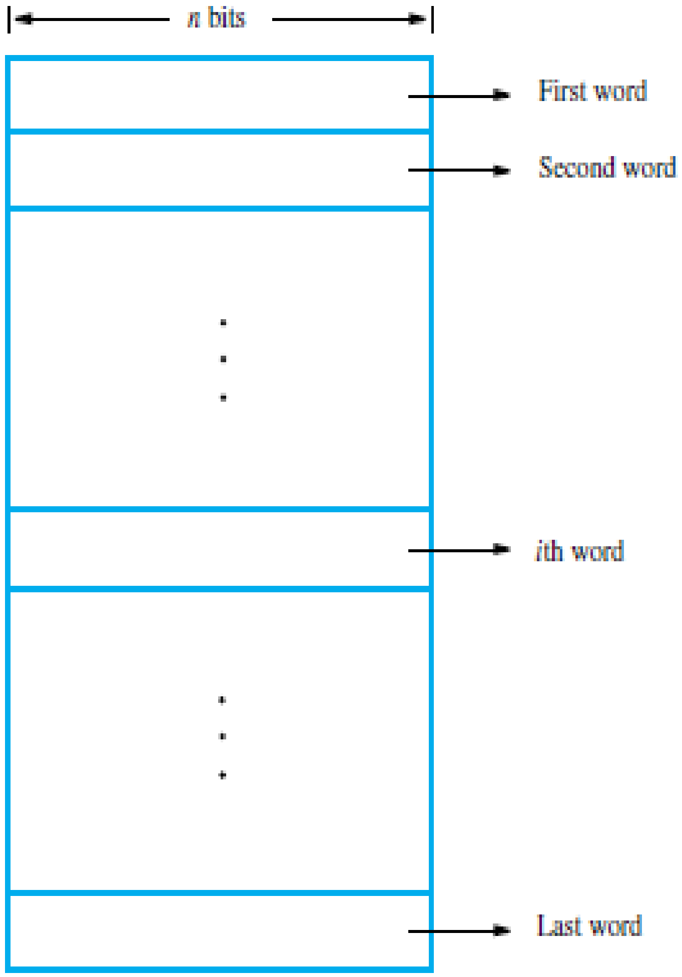
Memory Locations and Addresses

Memory Locations and Addresses

- The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1.

- The memory is organized so that a group of n bits can be stored or retrieved in a single, basic operation.

- Each group of n bits is referred to as a word of information, and n is called the word length.



Memory word

➢Modern computers have word lengths that typically range from 16 to 64 bits.

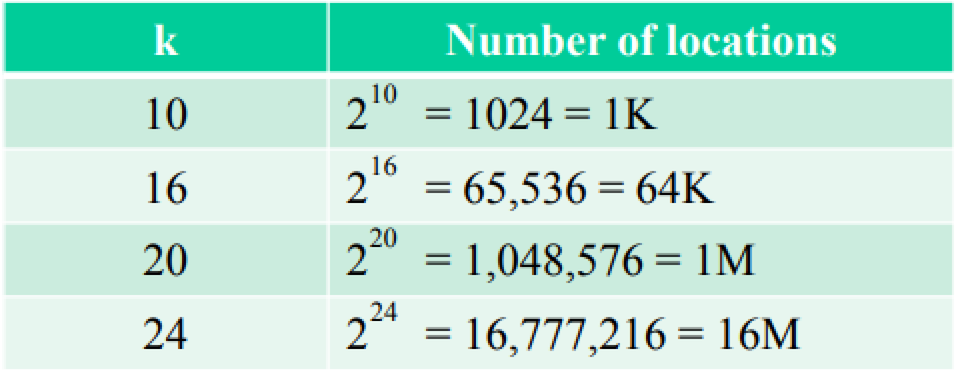
➢ If the word length of a computer is 32 bits, a single word can store a 32-bit signed number or four ASCII-encoded characters, each occupying 8 bits.

➢ A unit of 8 bits is called a byte.

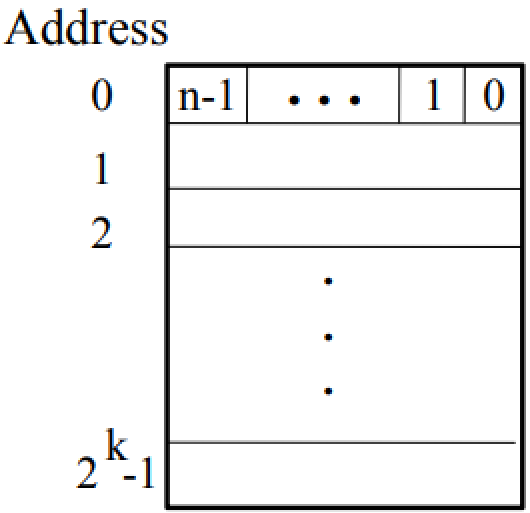
➢Machine instructions may require one or more words for their representation.

- ➢Memory holds both instructions and data.

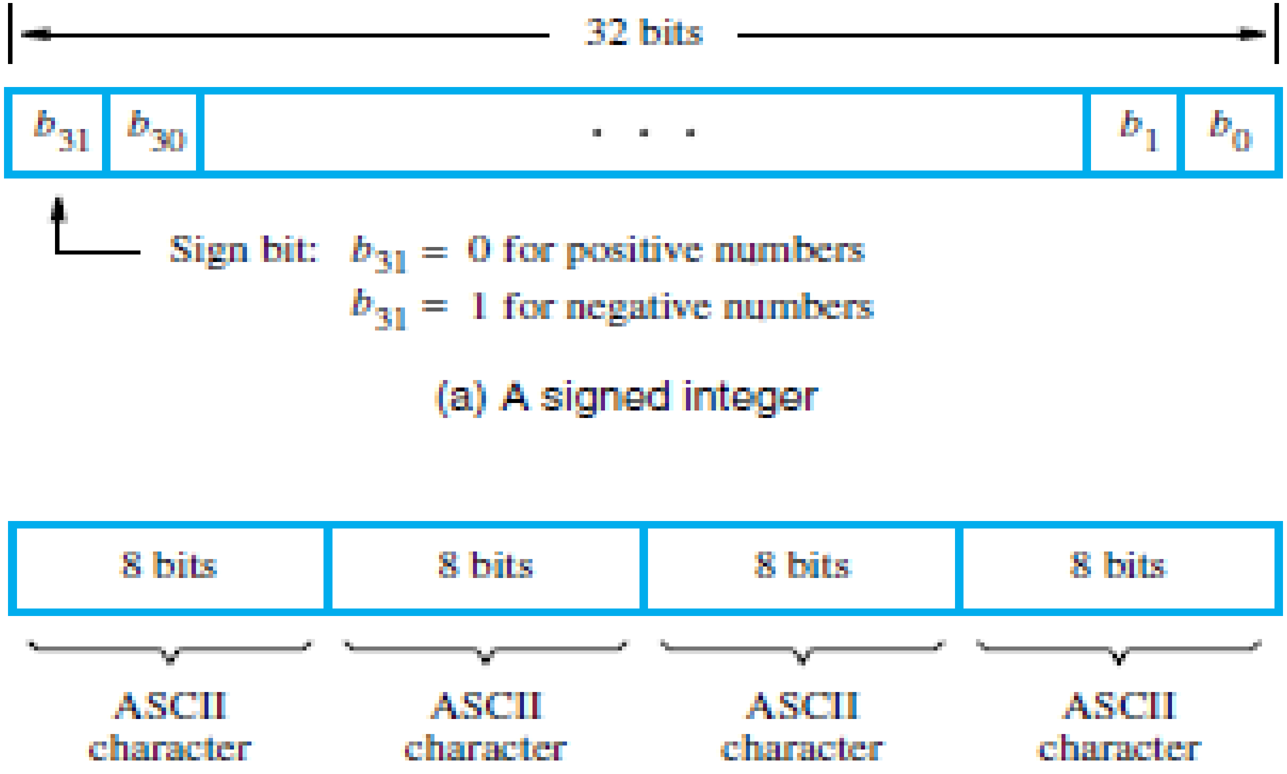
- ➢k address bits and n bits per location.



k	Number of locations
10	2^{10} = 1024 = 1K
16	2^{16} = 65,536 = 64K
20	2^{20} = 1,048,576 = 1M
24	2^{24} = 16,777,216 = 16M



- ➢n is typically 8 (byte), 16 (word), 32 (long word), ….



32 bits

b_{31} b_{30} . . . b_1 b_0

Sign bit: $b_{31} = 0$ for positive numbers

$b_{31} = 1$ for negative numbers

(a) A signed integer

8 bits | 8 bits | 8 bits | 8 bits

ASCII character | ASCII character | ASCII character | ASCII character

(b) Four characters

Examples of encoded information in a 32-bit word.

Byte Addressability

- ➢A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits.

- ➢It is impractical to assign distinct addresses to individual bit locations in the memory

- ➢The term byte-addressable memory is used for this assignment.

- ➢Byte locations have addresses 0, 1, 2, ...

- ➢Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0, 4, 8, . . . , with each word consisting of four bytes.

Big-Endian and Little-Endian Assignments

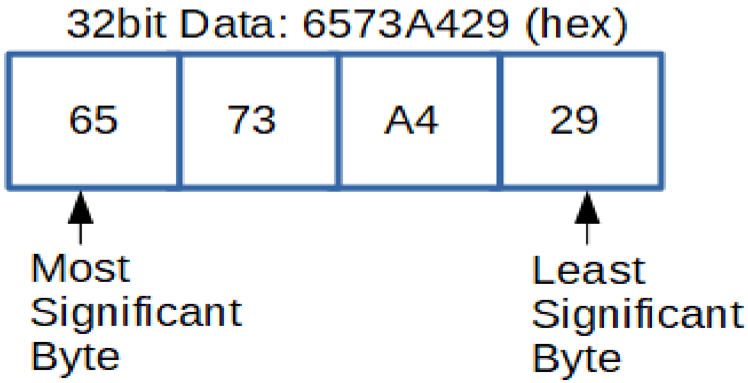
➢There are two ways that byte addresses can be assigned across words.

➢**Little-endian** – The bytes are ordered with the least significant byte placed at the lowest address.

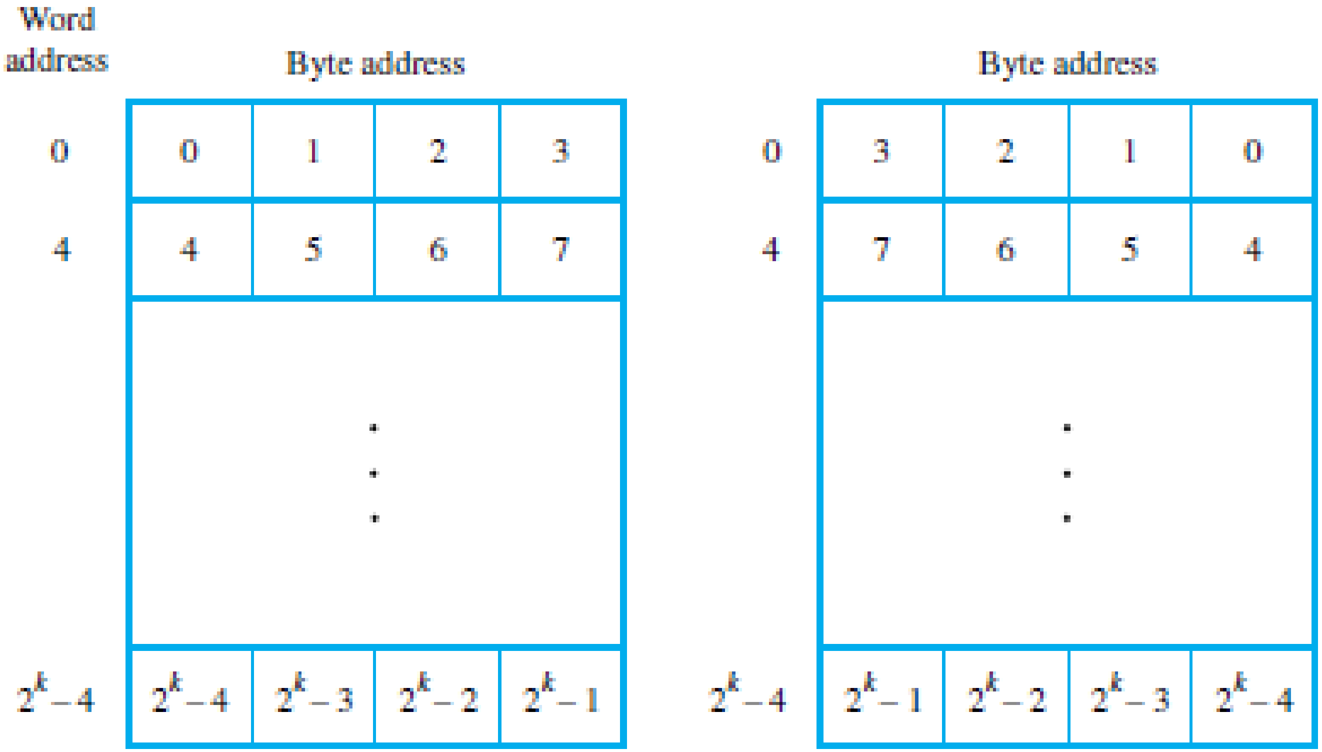
➢**Big-endian** – The bytes are ordered with the most significant byte placed at the lowest address.

➢The words "more significant" and "less significant" are used in relation

➢to the weights (powers of 2) assigned to bits when the word represents a number.



Visualization of a 32-bit data consisting of four bytes



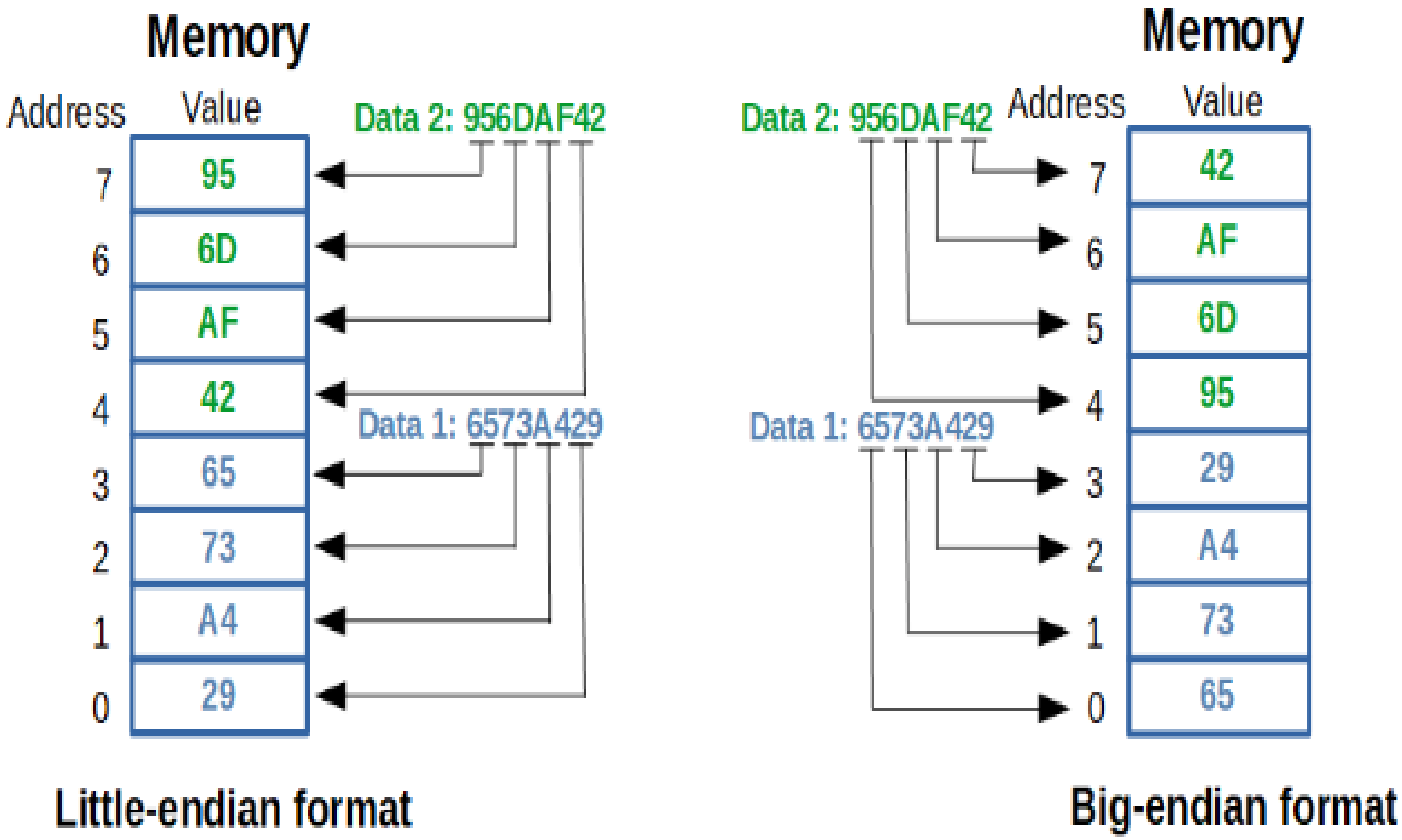
Word address	Byte address			
0	0	1	2	3
4	4	5	6	7
	• • •			
2^k-4	2^k-4	2^k-3	2^k-2	2^k-1

(a) Big-endian assignment

	Byte address			
0	3	2	1	0
4	7	6	5	4
	• • •			
2^k-4	2^k-1	2^k-2	2^k-3	2^k-4

(b) Little-endian assignment

Byte and word addressing.



Data 2: 956DAF42

Data 1: 6573A429

Memory

Address	Value
7	95
6	6D
5	AF
4	42
3	65
2	73
1	A4
0	29

Little-endian format

Memory

Address	Value
7	42
6	AF
5	6D
4	95
3	29
2	A4
1	73
0	65

Big-endian format

Little-endian vs big-endian format

Memory Operations

Memory operations:

- ➢Today, general-purpose computers use a set of instructions called a program to process data.

- ➢ A computer executes the program to create output data from input data

- ➢ Both program instructions and data operands are stored in memory

- ➢ Two basic operations requires in memory access
  - • Load operation (Read or Fetch)-Contents of specified memory location are read by processor
  - • Store operation (Write)- Data from the processor is stored in specified memory location

Instructions and Instruction Sequencing

INSTRUCTIONS AND INSTRUCTION SEQUENCING

A computer must have instructions capable of performing four types of operations:

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

Register Transfer Notation

To describe the transfer of information, the contents of any location are denoted by placing square brackets around its name.

$$\mathbf{R1 \leftarrow [LOC]}$$

Thus, this expression means that the contents of memory location LOC are transferred into processor register R1.

As another example, consider the operation that adds the contents of registers R1 and R2, and places their sum into register R3.

This action is indicated as

$$\mathbf{R3 \leftarrow [R1] + [R2]}$$

This type of notation is known as **Register Transfer Notation (RTN)**

Assembly-Language Notation

We need another type of notation to represent machine instructions and programs.

For this, we use assembly language. For example, a generic instruction that causes the transfer described above, from memory location LOC to processor register R1, is specified by the statement

Move LOC, R1

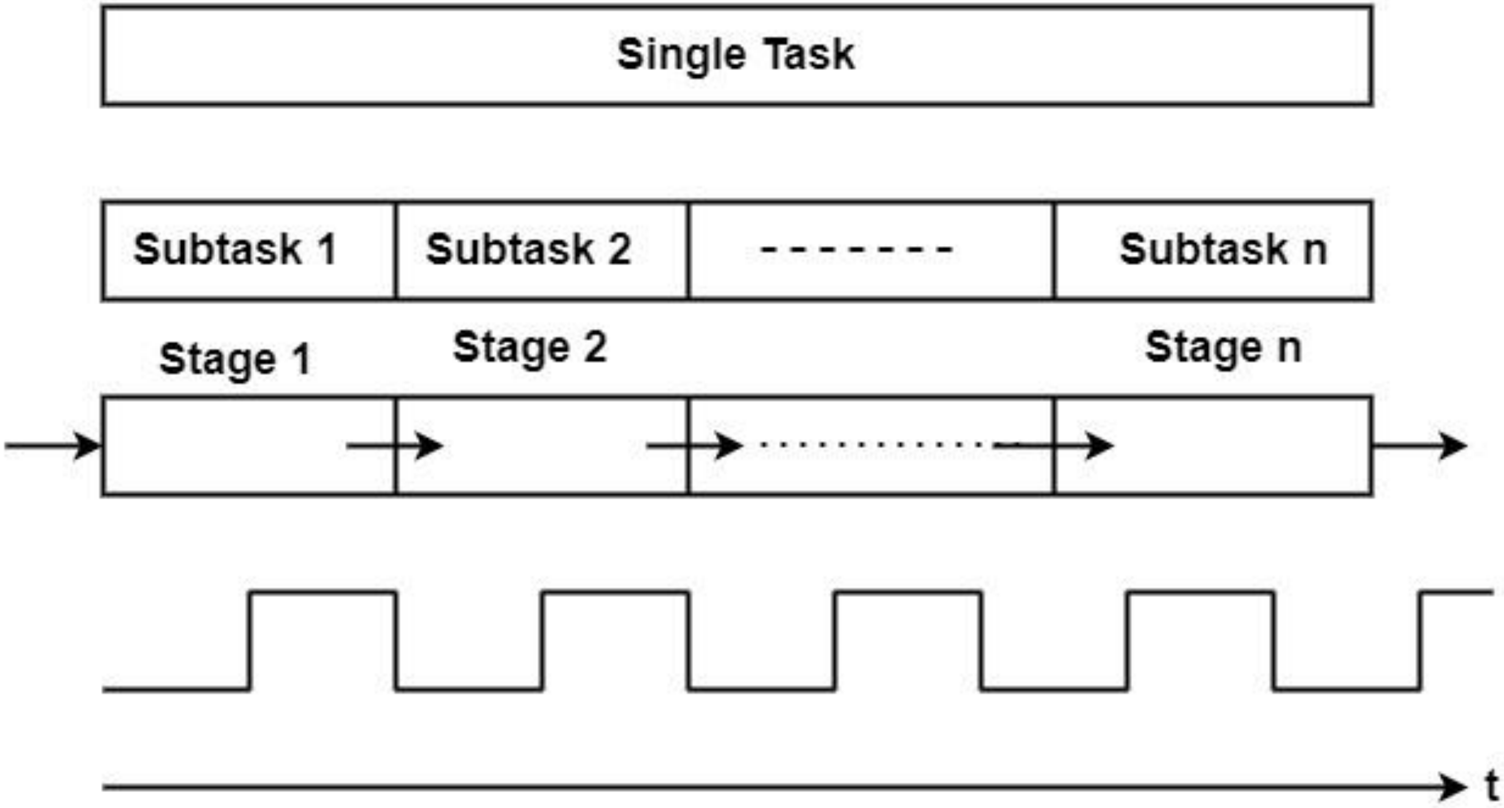
The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.

The second example of adding two numbers contained in processor registers R1 and R2 and placing their sum in R3 can be specified by the assembly-language statement

Add R1, R2, R3

In this case, registers R1 and R2 hold the source operands, while R3 is the destination.

Basic Principle of Pipelining



Single Task

Subtask 1

Subtask 2

- - - - - - - -

Subtask n

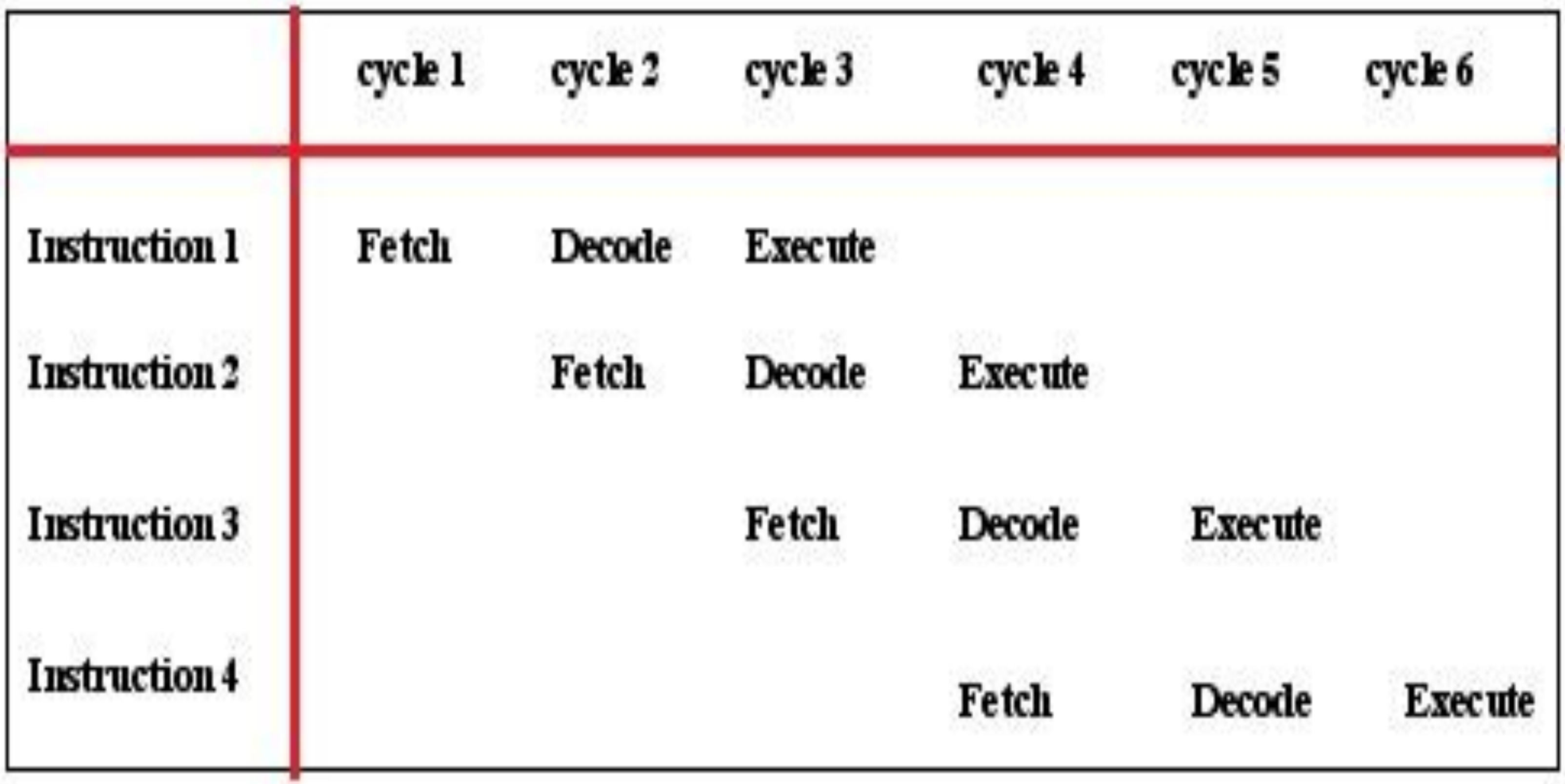
Stage 1

Stage 2

Stage n

t

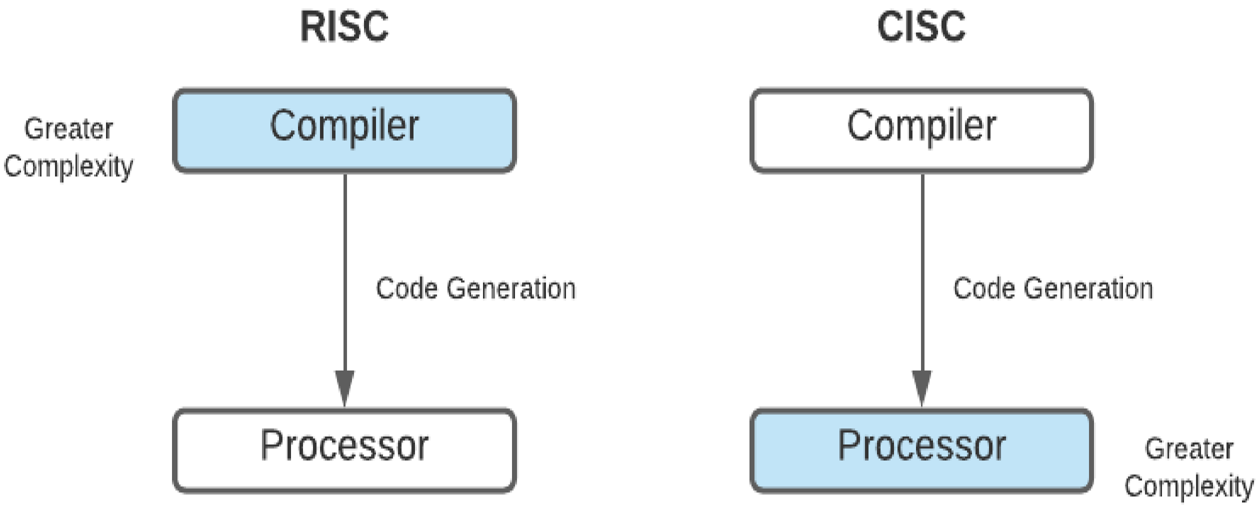
Pipelining:

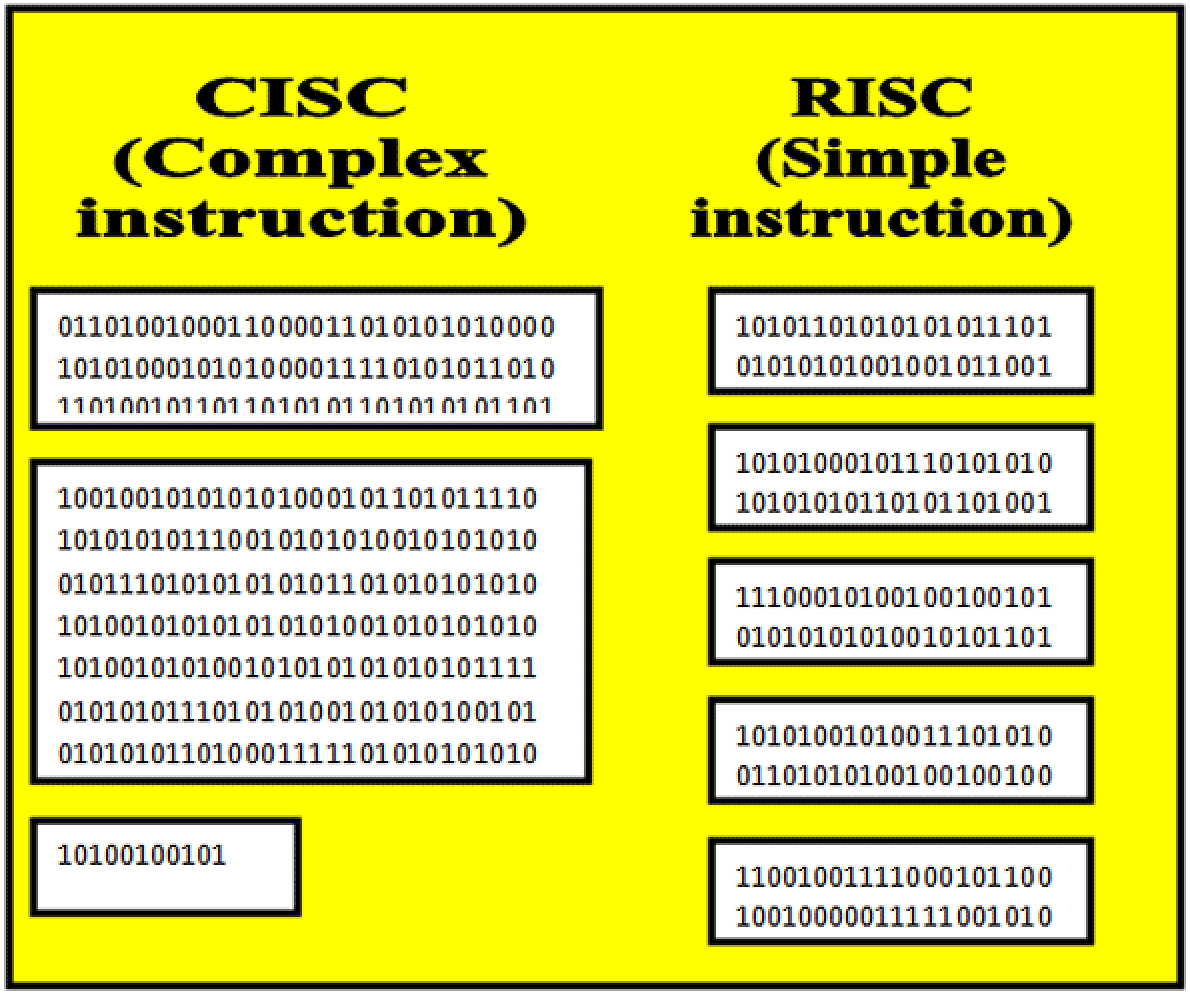


	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6
Instruction 1	Fetch	Decode	Execute			
Instruction 2		Fetch	Decode	Execute		
Instruction 3			Fetch	Decode	Execute	
Instruction 4				Fetch	Decode	Execute

RISC	CISC
1. RISC stands for Reduced Instruction Set Computer.	1. CISC stands for Complex Instruction Set Computer.
2. RISC processors have simple instructions taking about one clock cycle. The average clock cycle per instruction (CPI) is 1.5	2. CSIC processor has complex instructions that take up multiple clocks for execution. The average clock cycle per instruction (CPI) is in the range of 2 and 15.
3. Performance is optimized with more focus on software	3. Performance is optimized with more focus on hardware.
4. It has no memory unit and uses separate hardware to implement instructions..	4. It has a memory unit to implement complex instructions.
5. It has a hard-wired unit of programming.	5. It has a microprogramming unit.
6. The instruction set is reduced i.e. it has only a few instructions in the instruction set.	**6.** The instruction set has a variety of different instructions that can be used for complex operations.
7. Multiple register sets are present	**7.** Only has a single register set
8. RISC processors are highly pipelined	8. They are normally not pipelined or less pipelined

RISC	CISC
11. The complexity of RISC lies with the compiler that executes the program - Compiler	11. The complexity lies in the microprogram - processor
12. Execution time is very less	12. Execution time is very high
13. Code expansion can be a problem	13. Code expansion is not a problem
14. The decoding of instructions is simple.	14. Decoding of instructions is complex
15. It does not require external memory for calculations	15. It requires external memory for calculations
16. The most common RISC microprocessors are Alpha, ARC, ARM, AVR, MIPS, PA-RISC, PIC, Power Architecture, and SPARC.	16. Examples of CISC processors are the System/360, VAX, PDP-11, Motorola 68000 family, AMD, and Intel x86 CPUs.
17. RISC architecture is used in high-end applications such as video processing, telecommunications, and image processing.	17. CISC architecture is used in low-end applications such as security systems, home automation, etc.


RISC
CISC
Greater Complexity
Compiler
Code Generation
Processor
Compiler
Code Generation
Processor
Greater Complexity



CISC
(Complex
instruction)

0110100100011000011010101010000
1010100010101000011110101011010
1101001011011010101101010101101

10010010101010100010110101 11110
10101010111001010101001010101 0
01011101010101010110101010101 0
10100101010101010100101010101 0
10100101010010101010101010111 1
01010101110101010010101010010 1
01010101101000111110101010101 0

10100100101

RISC
(Simple
instruction)

10101101010101011101
01010101001001011001

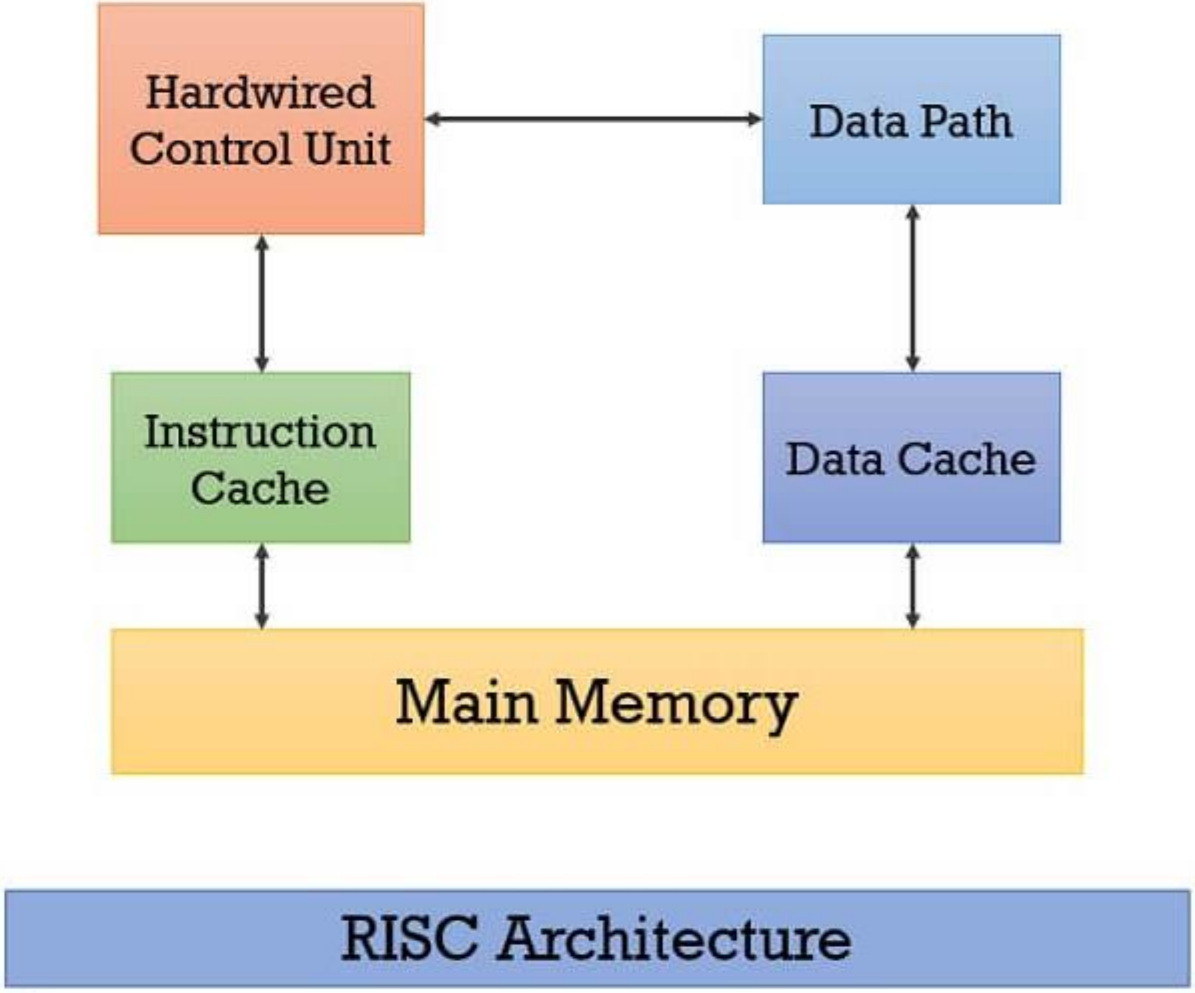
10101000101110101010
10101010110101101001

11100010100100100101
01010101010010101101

10101001010011101010
01101010100100100100

11001001111000101100
10010000011111001010

Introduction to RISC Instruction Sets



Two key characteristics of RISC instruction sets are:

- Each instruction fits in a single word.
- A load/store architecture is used, in which

– Memory operands are accessed only using Load and Store instructions.

– All operands involved in an arithmetic or logic operation must either be in processor registers, or one of the operands may be given explicitly within the instruction word.

RISC Instruction Sets

The instructions that have arithmetic and logic operation should have their operand either in the processor register or should be given directly in the instruction.

Like in both the instructions below we have the operands in registers

Add R2, R3
Add R2, R3, R4

The operand can be mentioned directly in the instruction as below:

Add R2, 100

At the start of execution of the program, all the operands are in memory. So, to access the memory operands, the RISC instruction set has load and store instructions.

The Load instruction loads the operand present in memory to the processor register. The load instruction is of the form:

Load destination, Source

Example Load R2, A // memory to register

The load instruction above will load the operand present at memory location A to the processor register R2.

The Store instruction stores the operand back to the memory.

Generally, the Store instruction is used to store the intermediate result or the final result in the memory. It is of the form:

Store source, destination

Example Store R2, A // register to memory

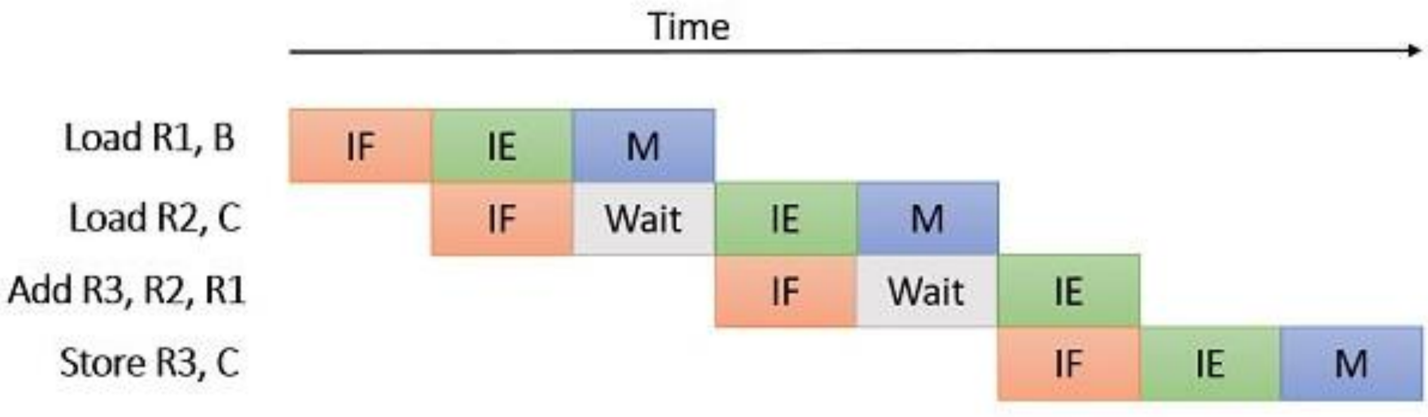
The Store instruction above will store the content in register **R2** into the **A** a memory location.

Consider the following instruction:

A = B + C

Creating a RISC instruction set for the above instruction will be.

Load R1, B
Load R2, C
Add R3, R2, R1
Store R3, C

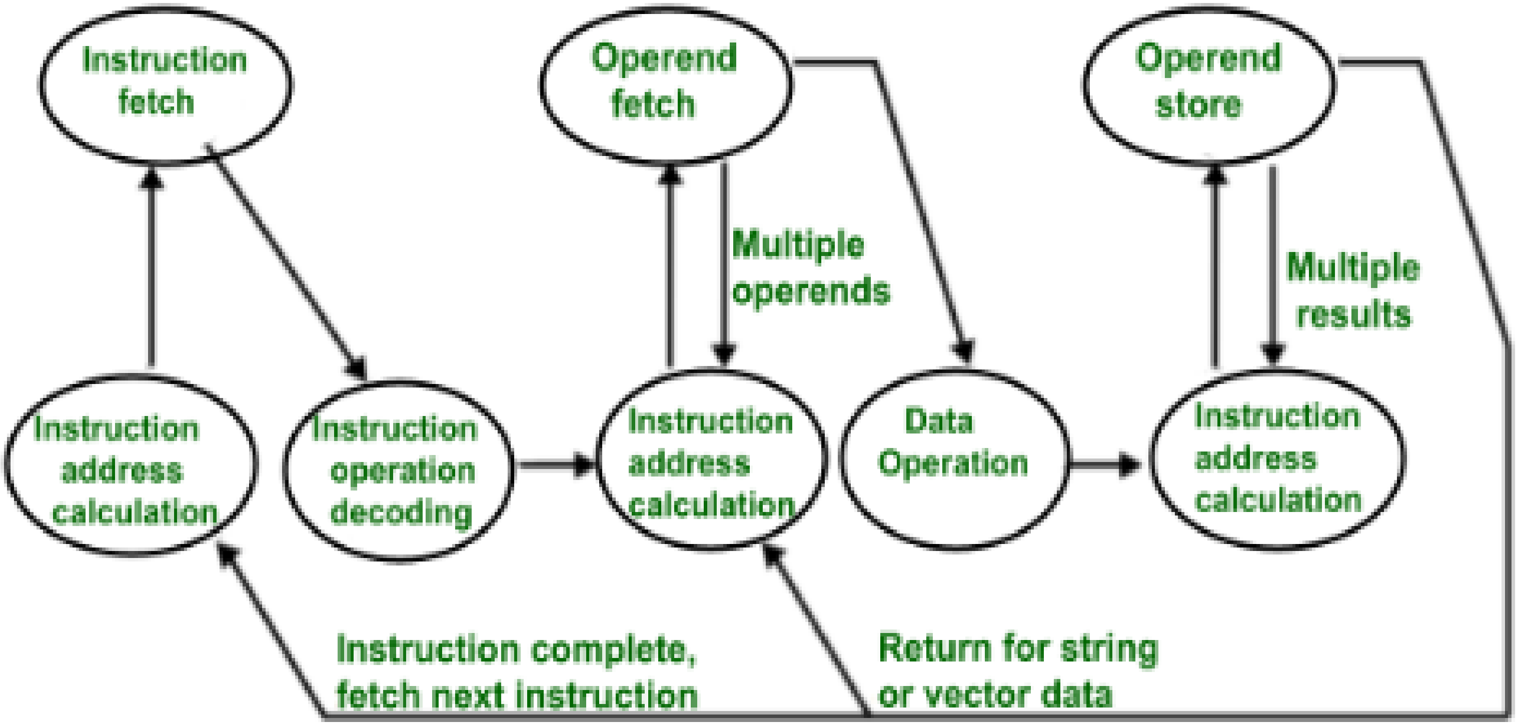


Time

Load R1, B	IF	IE	M					
Load R2, C		IF	Wait	IE	M			
Add R3, R2, R1				IF	Wait	IE		
Store R3, C						IF	IE	M

Instruction Fetch (IF): Fetching the instruction
Instruction Execute (IE): Calculate memory address
Memory Store (M): Register to register operation or memory to memory operation

Instruction Execution and Straight-Line Sequencing


Instruction fetch
Operend fetch
Operend store
Multiple operends
Multiple results
Instruction address calculation
Instruction operation decoding
Instruction address calculation
Data Operation
Instruction address calculation
Instruction complete, fetch next instruction
Return for string or vector data

Instruction cycle state transition diagram

Instruction execution :

Instruction execution needs the following steps, which are

➢PC (program counter) register of the processor gives the address of the instruction which needs to be fetched from the memory.

➢If the instruction is fetched then, the instruction opcode is decoded. On decoding, the processor identifies the number of operands.

➢ If there is any operand to be fetched from the memory, then that operand address is calculated.

➢Operands are fetched from the memory.

➢ If there is more than one operand, then the operand fetching process may be repeated (i.e. address calculation and fetching operands).

➢After this, the data operation is performed on the operands, and a result is generated.

➢If the result has to be stored in a register, the instructions end here.

➢If the destination is memory, then first the destination address has to be calculated. Then the result is then stored in the memory.

➢ If there are multiple results which need to be stored inside the memory, then this process may repeat (i.e. destination address calculation and store result).

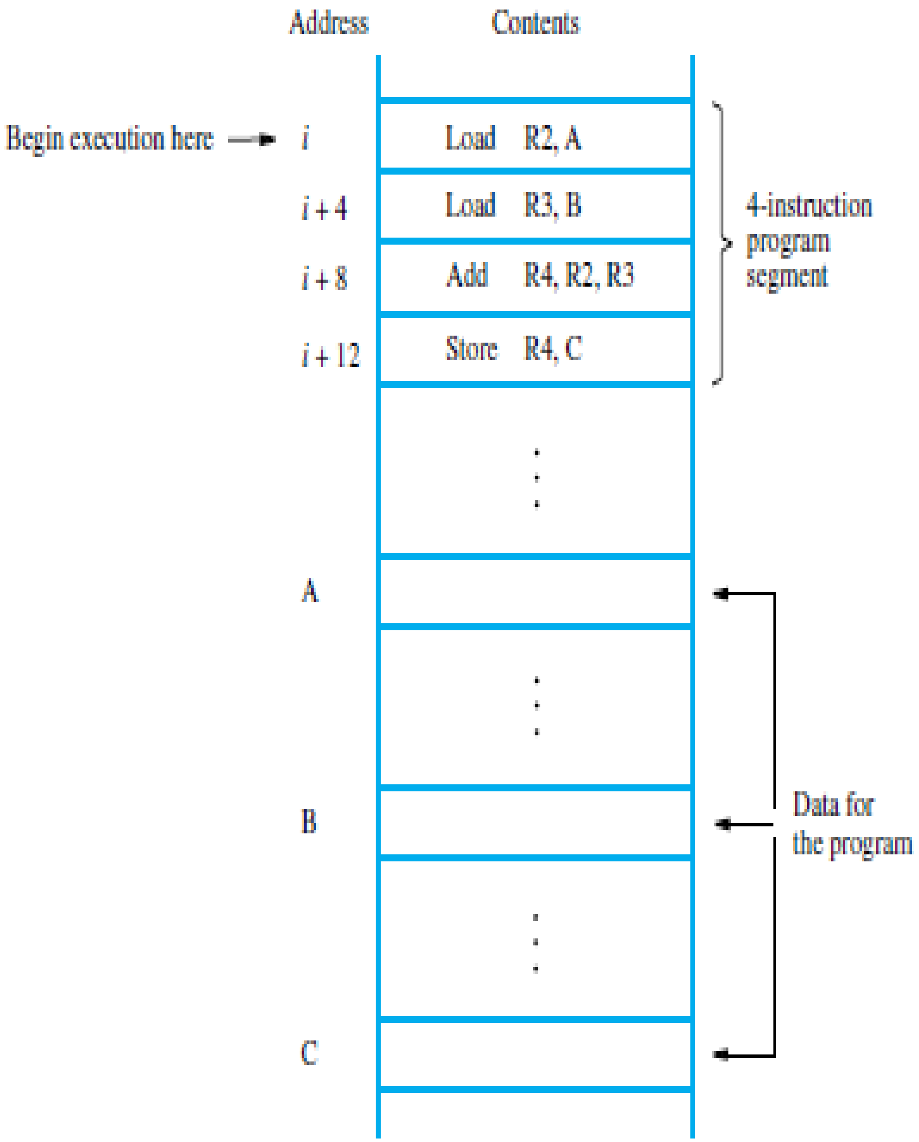
➢Now the current instructions have been executed.

➢ Side by side, the PC is incremented to calculate the address of the next instruction.

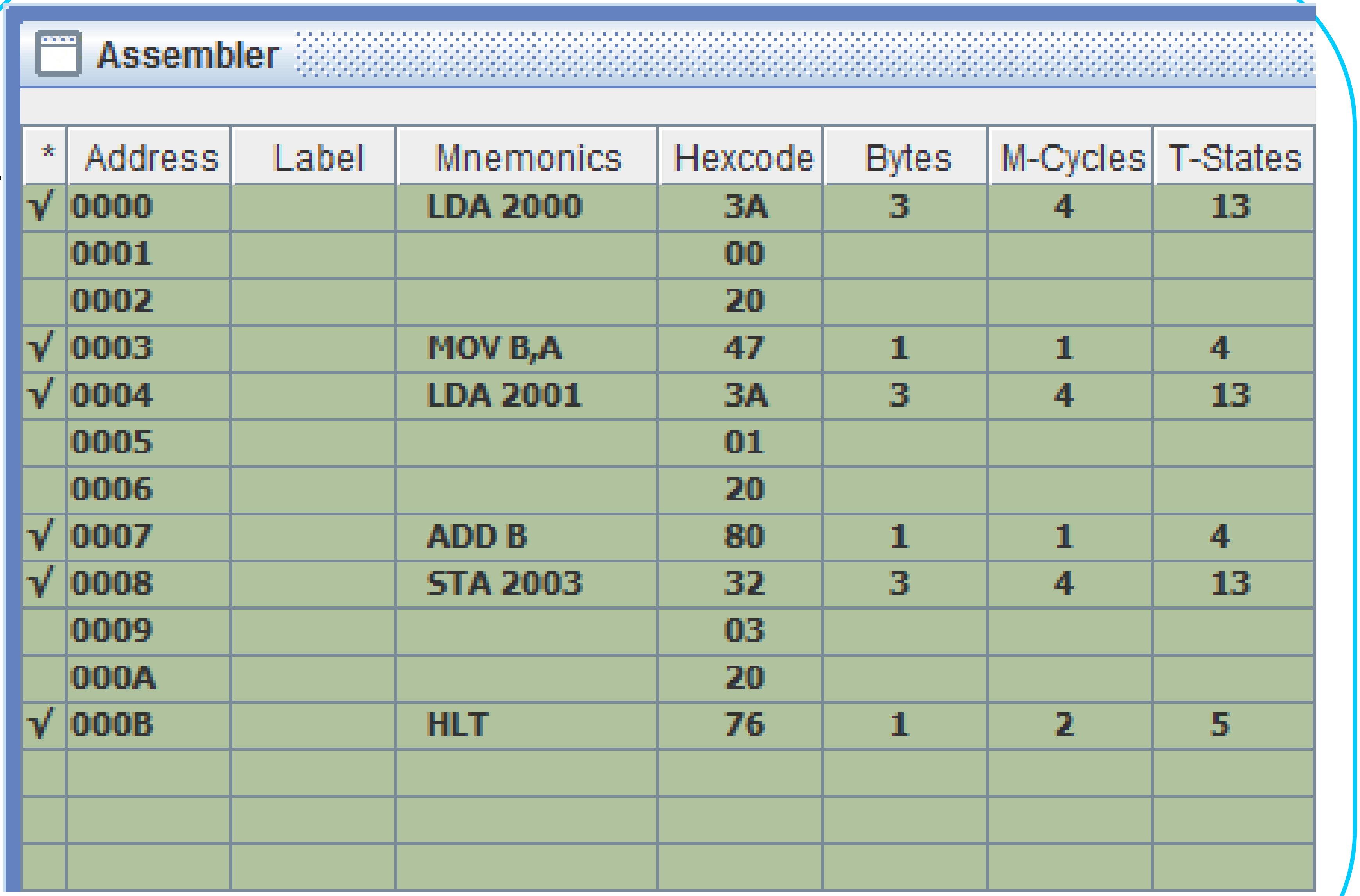
Straight line sequencing:

➢Straight line sequencing means the instruction of a program is executed in a sequential manner(i.e. every time PC is incremented by a fixed offset).

➢And no branch address is loaded on the PC.



A program for C ← [A] + [B].



Assembler

*	Address	Label	Mnemonics	Hexcode	Bytes	M-Cycles	T-States
√	0000		LDA 2000	3A	3	4	13
	0001			00			
	0002			20			
√	0003		MOV B,A	47	1	1	4
√	0004		LDA 2001	3A	3	4	13
	0005			01			
	0006			20			
√	0007		ADD B	80	1	1	4
√	0008		STA 2003	32	3	4	13
	0009			03			
	000A			20			
√	000B		HLT	76	1	2	5

Addressing Modes

Addressing modes:

➢Programs are normally written in a high-level language, which enables the programmer to conveniently describe the operations to be performed on various data structures.

➢When translating a high-level language program into assembly language, the compiler generates appropriate sequences of low-level instructions that implement the desired operations.

➢The different ways for specifying the locations of instruction operands are known as *addressing modes.*

Effective Address (EA)

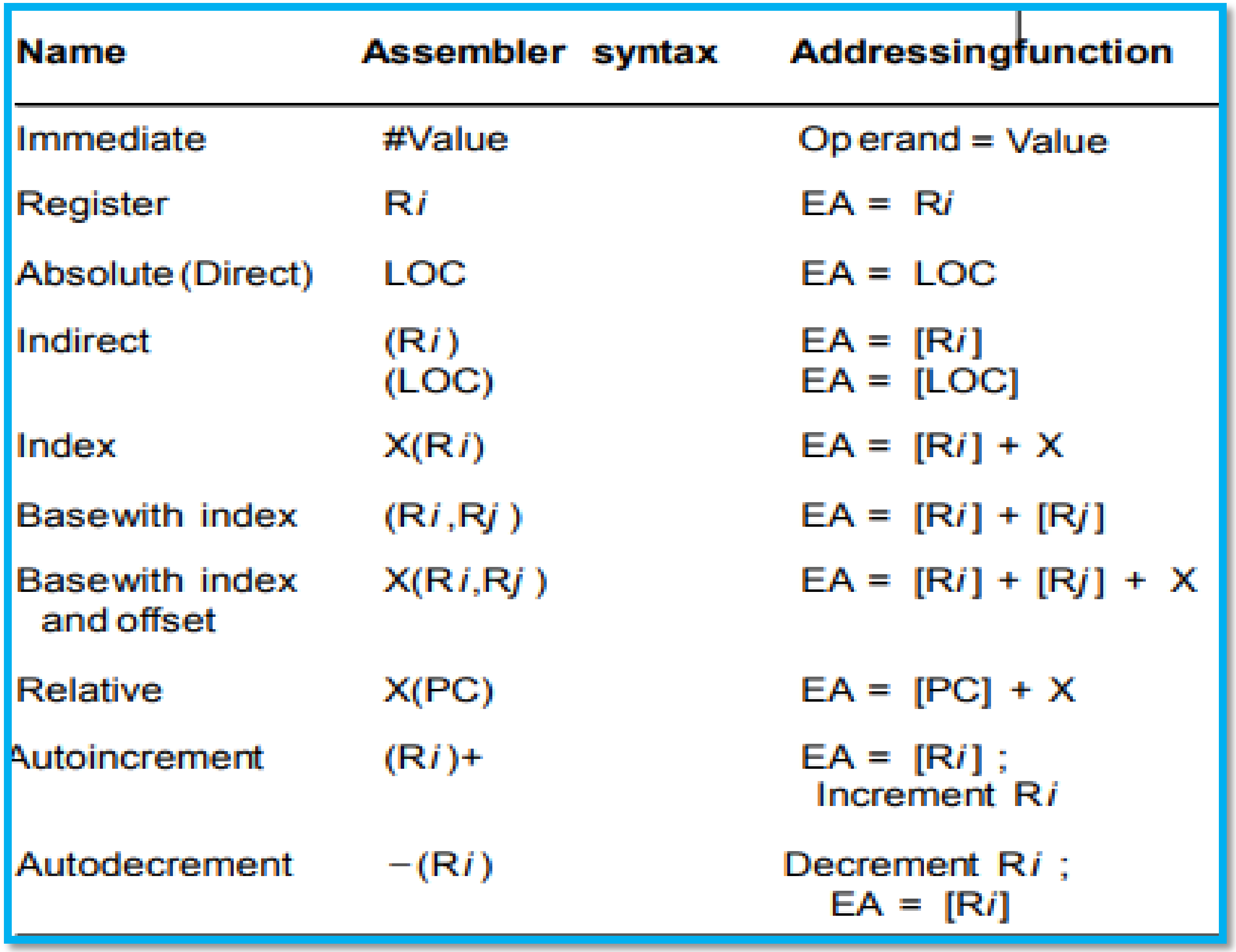
- Effective address is the address of the exact memory location where the value of the operand is present

- In the addressing modes that follow, the instruction does not give the operand or its address explicitly.

- Instead, it provides information from which an effective address (EA) can be derived by the processor when the instruction is executed.

- The effective address is then used to access the operand.

RISC-type addressing modes.



Name	Assembler syntax	Addressingfunction
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC

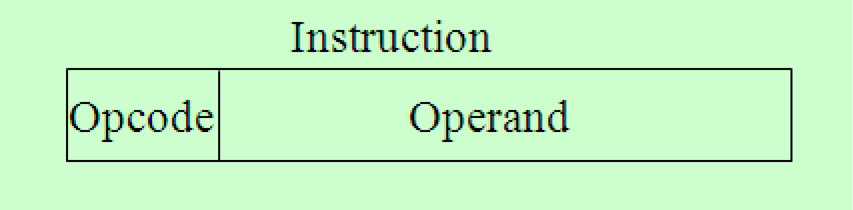
Indirect	(R*i*)
	(LOC)
		EA = [R*i*]
		EA = [LOC]

Index	X(R*i*)	EA = [R*i*] + X
Basewith index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Basewith index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*] ; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i* ; EA = [R*i*]

Addressing Modes

Immediate

- ➢Operand is part of instruction
- ➢Operand = address field



Instruction

Opcode | Operand

e.g. ADD 5

Add 5 to contents of accumulator

5 is operand

- ➢No memory reference to fetch data
- ➢The use of a constant in “MOV 5, R1” or

“MOV #5, R1” i.e. R1 ← 5

MOV #NUM1, R2 ; to copy the variable memory address

Direct Addressing

- Address field contains address of operand
- Effective address (EA) = address field (A)

e.g. ADD A

Add contents of cell A to accumulator

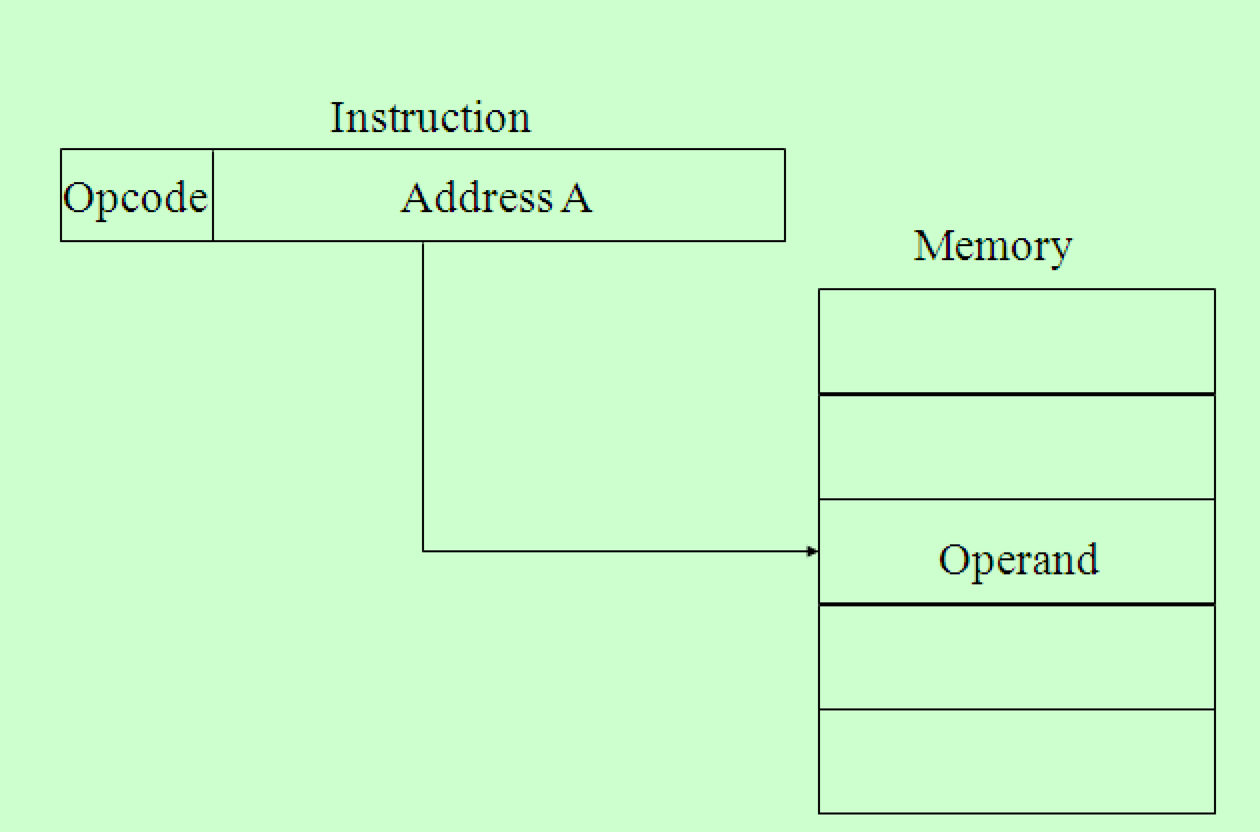
Look in memory at address A for operand

- Single memory reference to access data
- No additional calculations to work out effective address
- Limited address space
- Use the given address to access a memory location

E.g. Move NUM1, R1

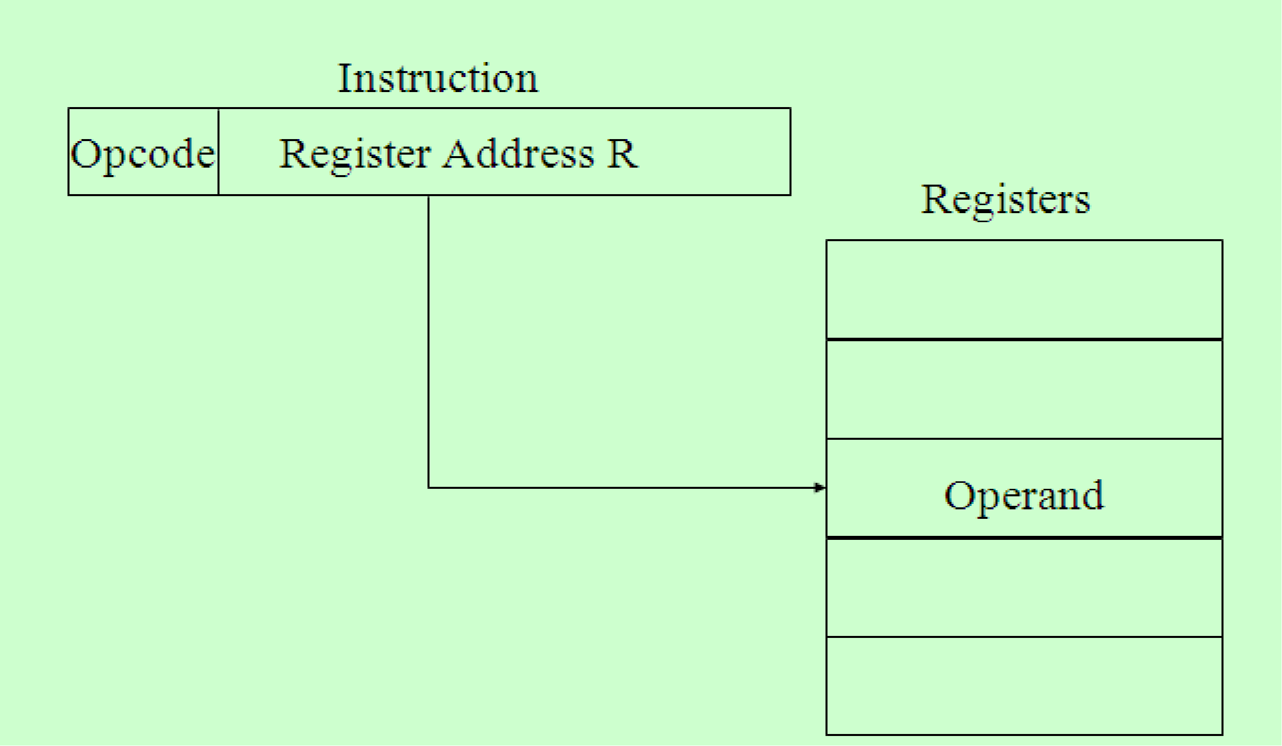
Move R0, SUM

Direct Addressing

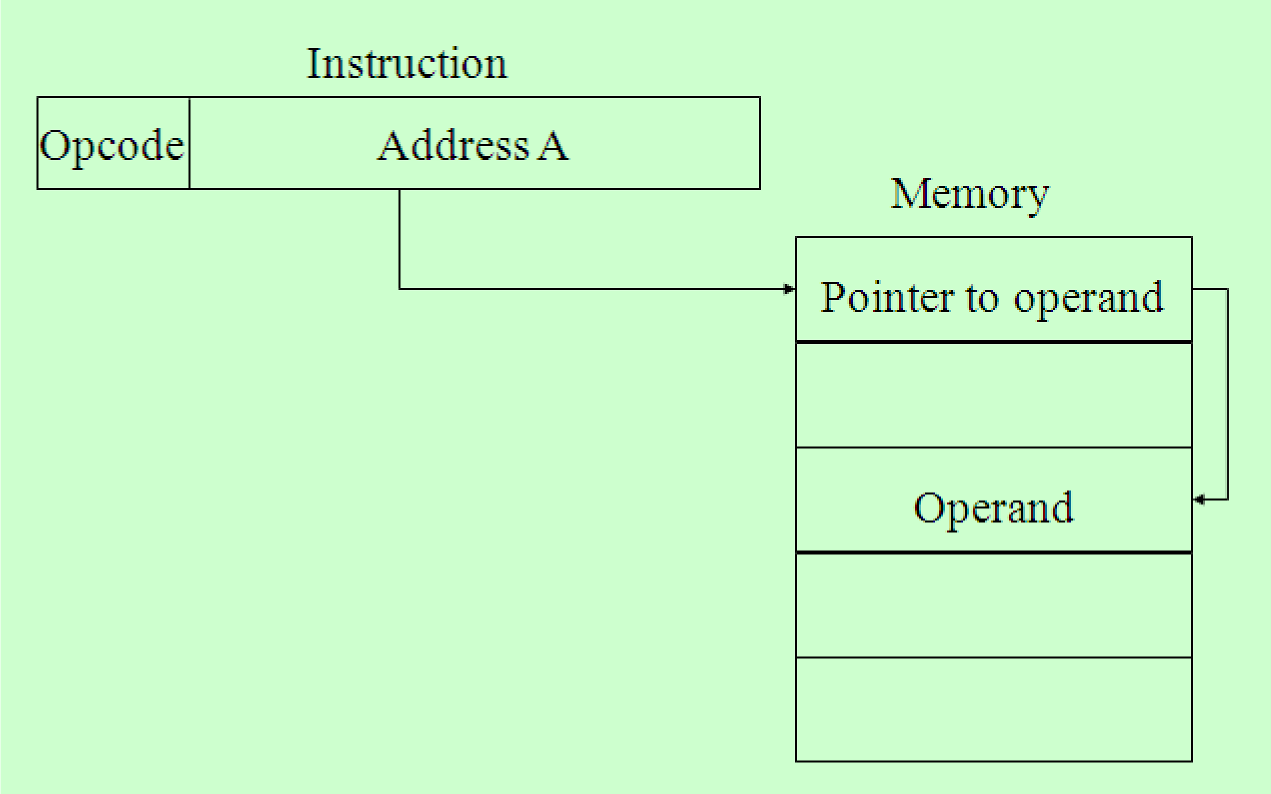


Register addressing

- Indicate which register holds the operand.
- EA = R
- Limited number of registers
- Very small address field needed



Indirect Addressing

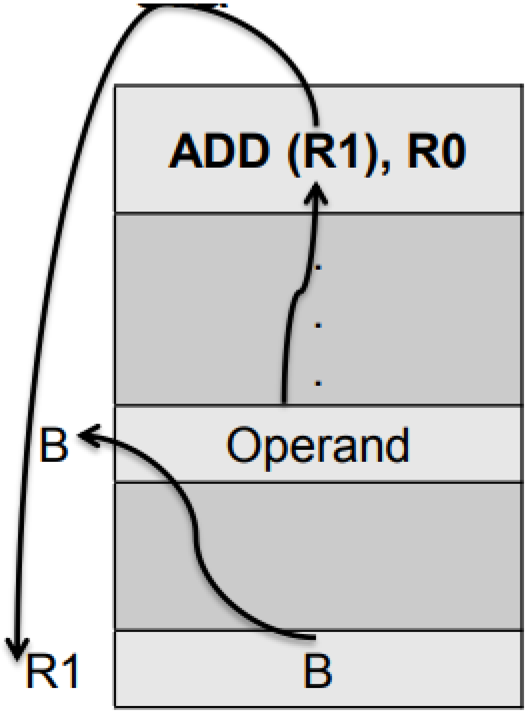


Indirect Addressing

- Indirect addressing through a general purpose register.

- Indicate the register (e.g. R1) that holds the address of the variable (e.g. B) that holds the operand.

ADD (R1), R0



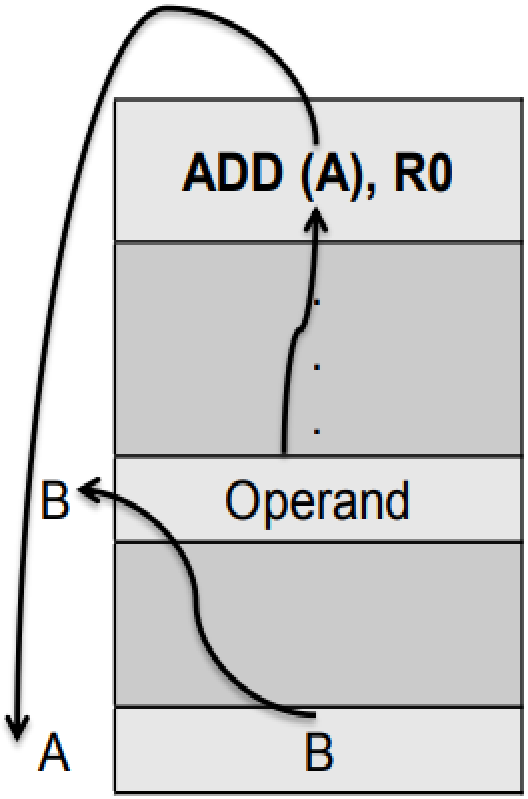
- The register or memory location that contain the address of an operand is called a pointer.

Indirect Addressing

- Indirect addressing through a memory addressing.

- Indicate the memory variable (e.g. A)that holds the address of the variable (e.g. B) that holds the operand

ADD (A), R0



Indirect Addressing

Example

Addition of N numbers

```
1. Move N,R1 ;                  N = Numbers to add
2. Move #NUM1,R2 ;              R2= Address of 1st no.
3. Clear R0 ;                   R0 = 00
4. Loop : Add (R2), R0 ;        R0 = [NUM1] + [R0]
5. Add #4, R2 ;                 R2= To point to the next ; number
6. Decrement R1 ;               R1 = [R1] -1
7. Branch>0 Loop ;             Check if R1>0 or not if ; yes go to Loop
8. Move R0, SUM ;               SUM= Sum of all no.
```

Indexing and Arrays

- The EA of the operand is generated by adding a constant value to the contents of a register.

X(Ri) ; EA= X + (Ri)   X= Signed number

- X defined as offset or displacement

- Index mode – the effective address of the operand is generated by adding a constant value to the contents of a register.

X(Ri): EA = X + [Ri]

- The constant X may be given either as an explicit number or as a symbolic name representing a numerical value.

- If X is shorter than a word, sign-extension is needed

Indexing and Arrays

- In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears.

- 2D Array

  (Ri, Rj) so EA = [Ri] + [Rj]

  Rj is called the base register

- 3D Array

  X(Ri, Rj) so EA = X + [Ri] + [Rj]

Indexing and Arrays

Address	Memory
	Add 20(R1), R2
	. . . .
10000H	
Offset=20	. . . .
10020H	Operand

R1	10000H

Offset is given as a Constant

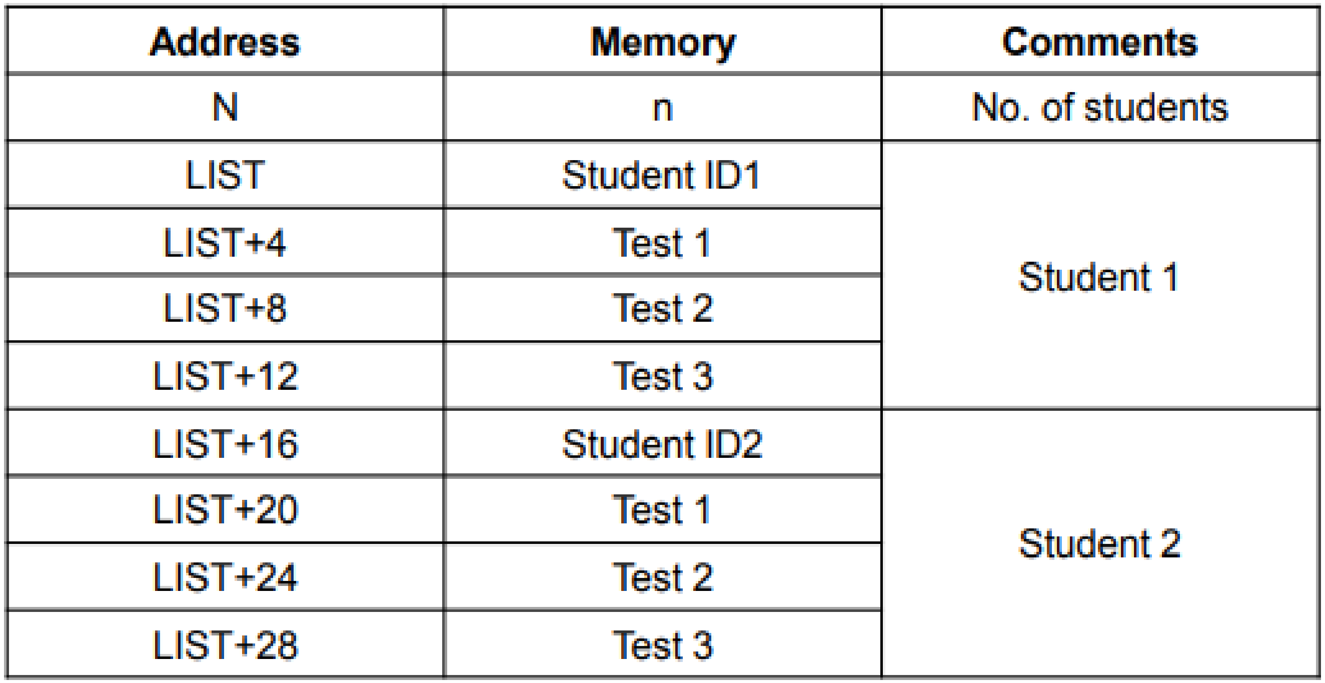
Address	Memory
	Add 10000H(R1), R2
	. . . .
10000H	
Offset=20	. . . .
10020H	Operand

R1	20H

Offset is in the index register

Indexing and Arrays

- Array
- E.g. List of students marks

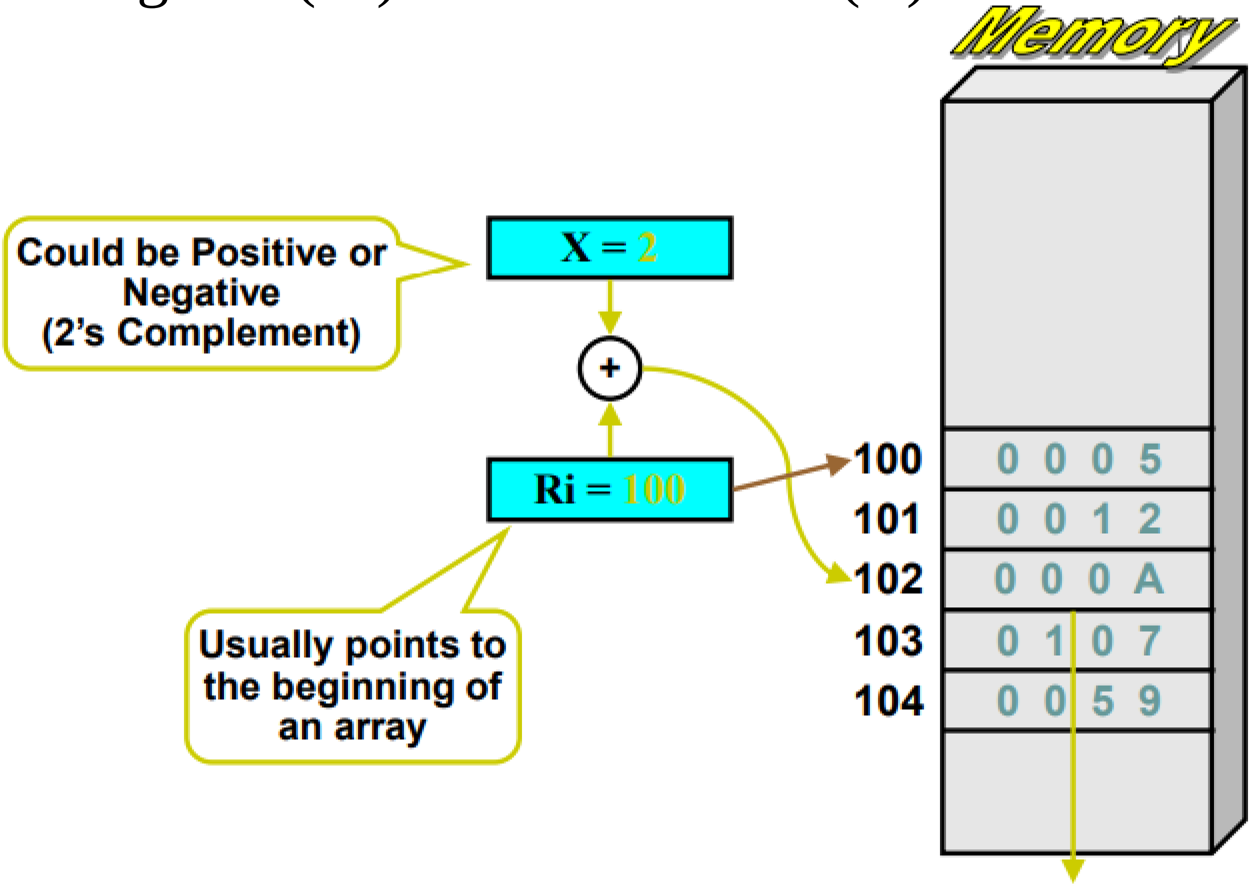


Address	Memory	Comments
N	n	No. of students
LIST	Student ID1	Student 1
LIST+4	Test 1	
LIST+8	Test 2	
LIST+12	Test 3	
LIST+16	Student ID2	Student 2
LIST+20	Test 1	
LIST+24	Test 2	
LIST+28	Test 3	

- Indexed addressing used in accessing test marks from the list

Base Register

EA = Base Register (Ri) + Relative Addr (X)



Could be Positive or Negative (2’s Complement)

Usually points to the beginning of an array

100	0 0 0 5
101	0 0 1 2
102	0 0 0 A
103	0 1 0 7
104	0 0 5 9

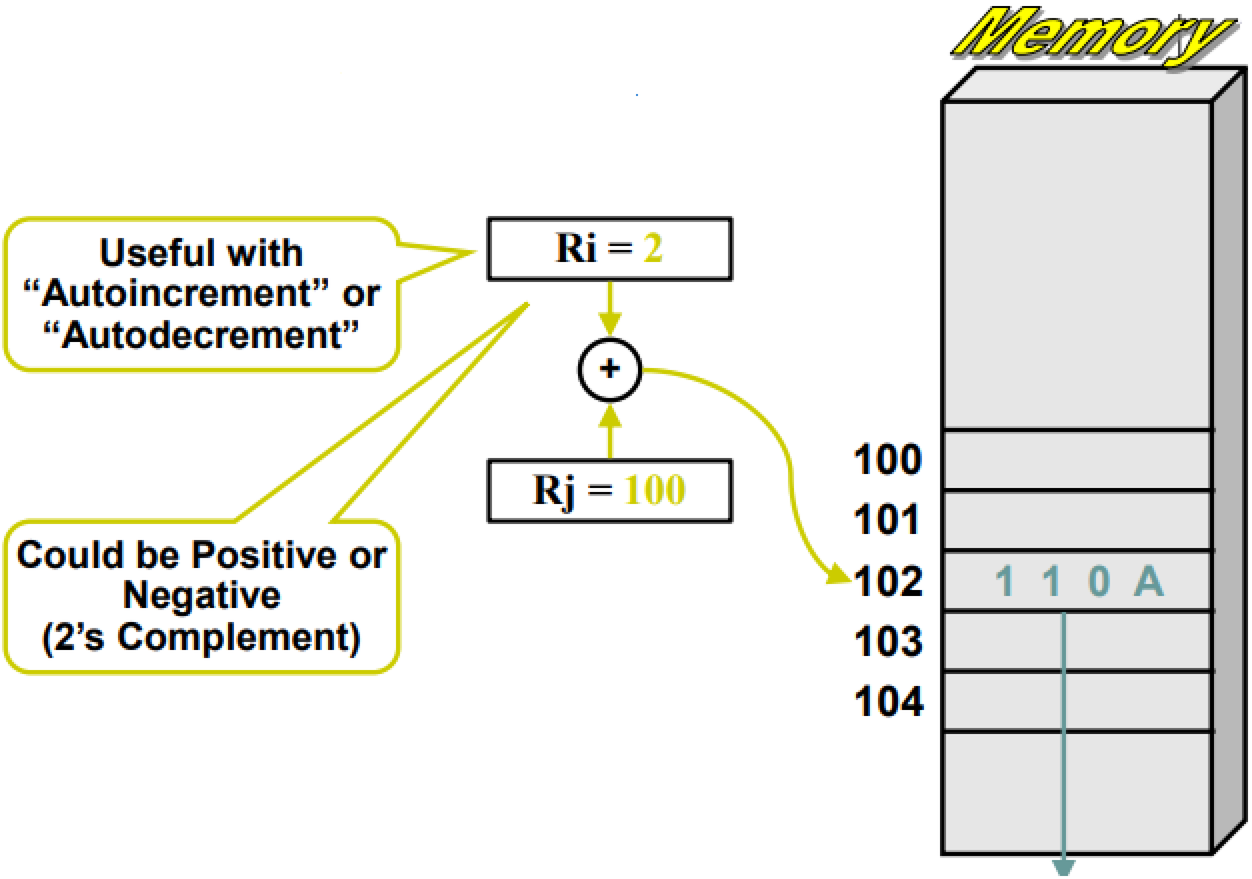
Indexing and Arrays

Program to find the sum of marks of all subjects of reach students and store it in memory.

```
Move #LIST, R0
Clear R1
Clear R2
Move #SUM, R2
Move N, R4
Loop : Add 4(R0), R1
Add 8(R0), R1
Add 12(R0),R1
Move R1, (R2)
Clear R1
Add #16, R0
Add #4, R2
Decrement R4
Branch>0 Loop
```

Indexed

EA = Index Register (Ri) + Relative Addr (Rj)


Memory
Useful with "Autoincrement" or "Autodecrement"
Ri = 2
+
Rj = 100
Could be Positive or Negative (2's Complement)
100
101
102
1 1 0 A
103
104

Relative Addressing

- Relative mode – the effective address is determined by the Index mode using the program counter in place of the general-purpose register.

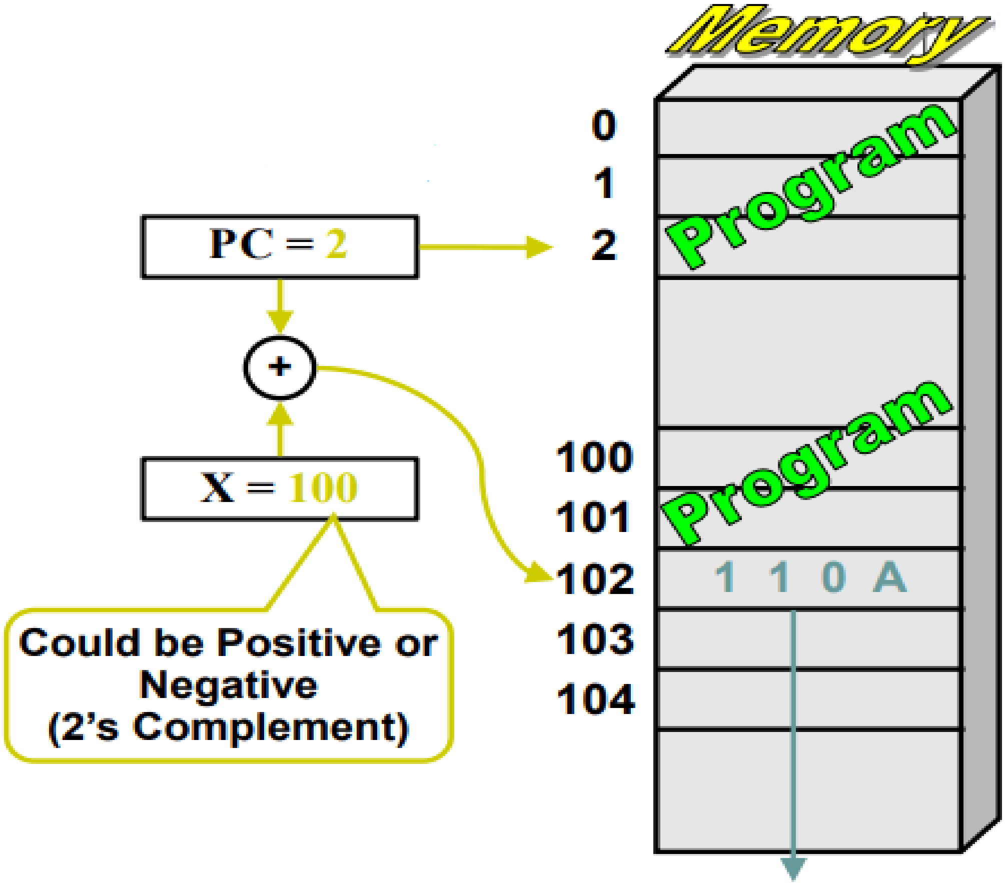
- X(PC) – note that X is a signed number
- Branch>0 LOOP

- This location is computed by specifying it as an offset from the current value of PC.

- Branch target may be either before or after the branch instruction, the offset is given as a singed num.

Relative Addressing

Relative Address EA = PC + Relative Addr (X)



Auto Increment

This addressing mode is a special case of Register Indirect Addressing Mode where-

Effective Address of the Operand = Content of Register

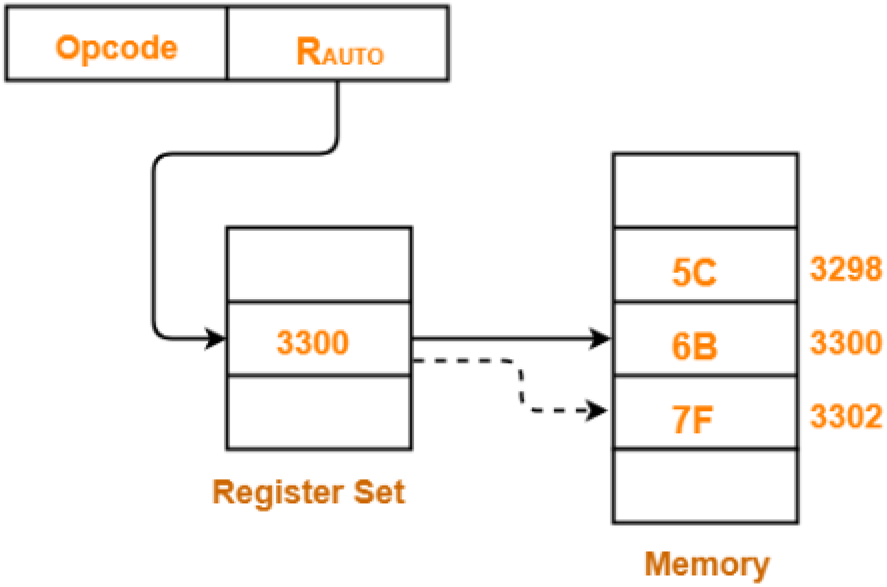
In this addressing mode,

- After accessing the operand, the content of the register is automatically incremented by step size ‘d’.
- Step size ‘d’ depends on the size of operand accessed.
- Only one reference to memory is required to fetch the operand.

In auto-increment addressing mode,

- First, the operand value is fetched.
- Then, the instruction register R_{AUTO} value is incremented by step size ‘d’.

Auto Increment



Assume operand size = 2 bytes.

Here,

- After fetching the operand 6B, the instruction register R_{AUTO} will be automatically incremented by 2.
- Then, updated value of R_{AUTO} will be 3300 + 2 = 3302.
- At memory address 3302, the next operand will be found.

Auto Decrement

Effective Address of the Operand = Content of Register – Step Size

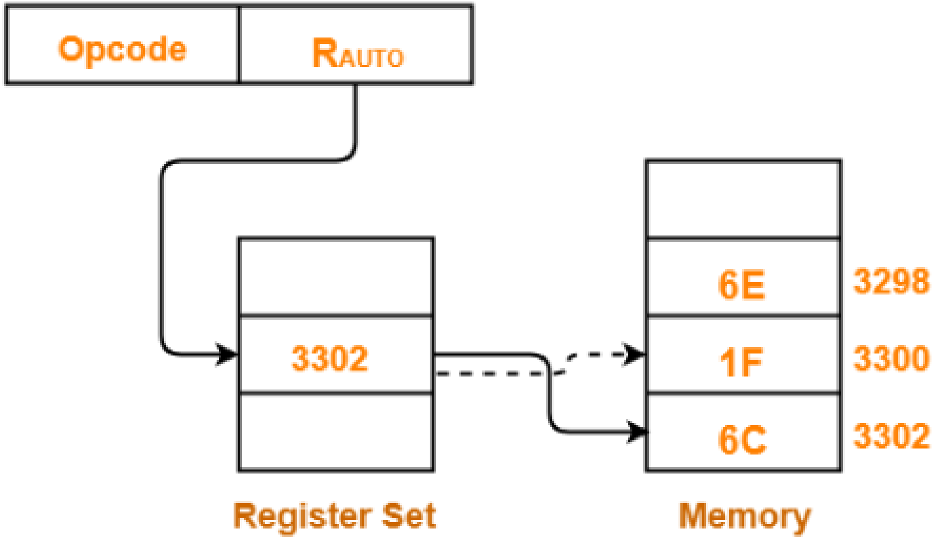
In this addressing mode,

- First, the content of the register is decremented by step size ‘d’.
- Step size ‘d’ depends on the size of operand accessed.
- After decrementing, the operand is read.
- Only one reference to memory is required to fetch the operand.

In auto-decrement addressing mode,

- First, the instruction register R_{AUTO} value is decremented by step size ‘d’.
- Then, the operand value is fetched.

Auto Decrement



Assume operand size = 2 bytes.

Here,

- First, the instruction register R_{AUTO} will be decremented by 2.
- Then, updated value of R_{AUTO} will be 3302 – 2 = 3300.
- At memory address 3300, the operand will be found.

ASSEMBLY LANGUAGE

Assembly language:

- Machine instructions are represented by patterns of 0s and 1s. So these patterns represented by symbolic names called “mnemonics”

E.g. Load, Store, Add, Move, BR, BGTZ

- A complete set of such symbolic names and rules for their use constitutes a programming language, referred to as an assembly language.

- The set of rules for using the mnemonics and for specification of complete instructions and programs is called the syntax of the language.

- Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an assembler.

- The assembler program is one of a collection of utility programs that are a part of the system software of a computer.

- The user program in its original alphanumeric text format is called a source program, and the assembled machine-language program is called an object program.

- The assembly language for a given computer is not case sensitive.

E.g. MOVE R1, SUM

Assembler Directives

➢In addition to providing a mechanism for representing instructions in a program, assembly language allows the programmer to specify other information needed to translate the source program into the object program.

➢Assign numerical values to any names used in a program.

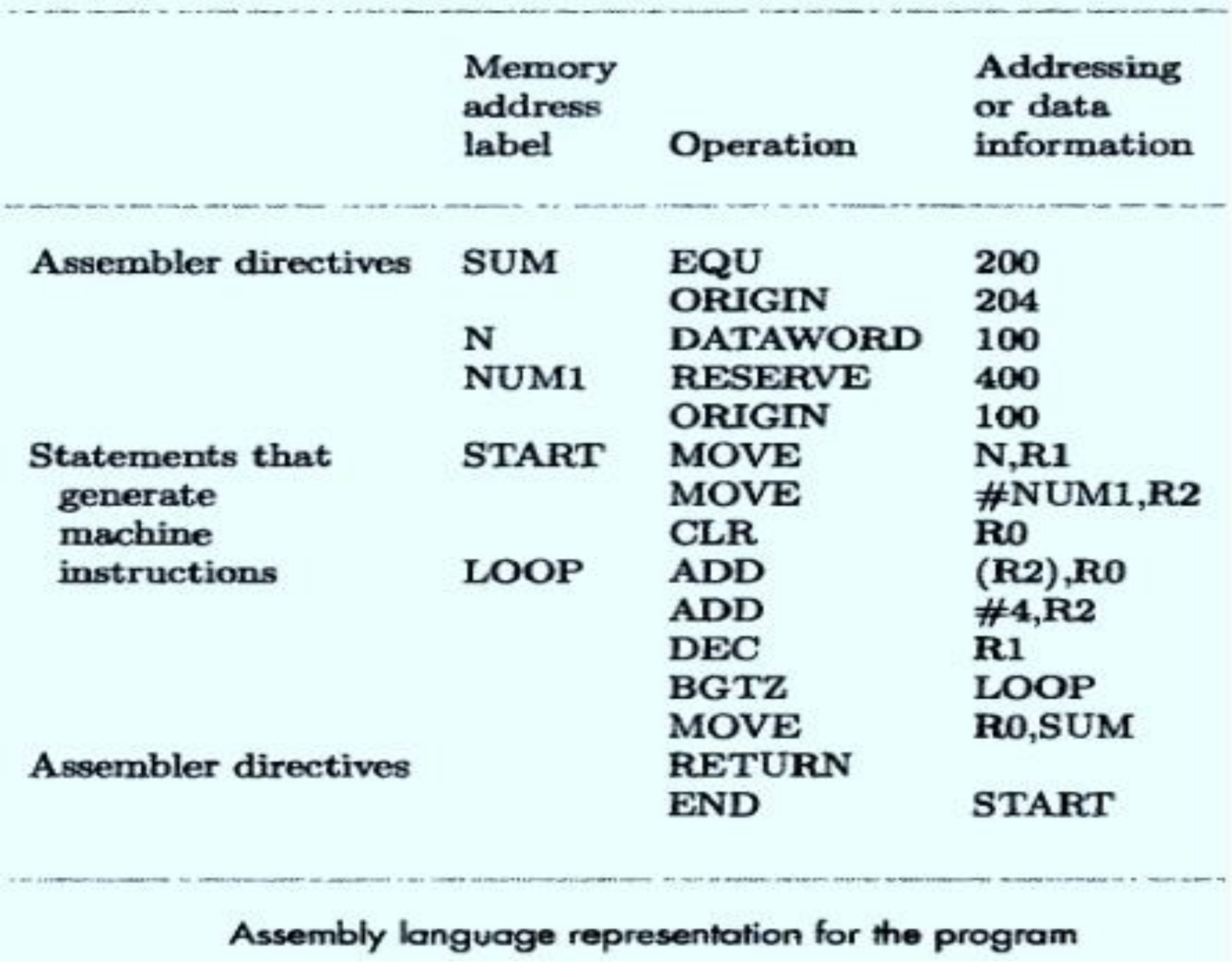
- For e,g, name TWENTY is used to represent the value 20. This fact may be conveyed to the assembler program through an equate statement such as TWENTY EQU 20

➢ If the assembler is to produce an object program according to this arrangement, it has to know

- How to interpret the names
- Where to place the instructions in the memory
- Where to place the data operands in the memory

Assembly language representation for the program

Label: Operation Operand(s) Comment



	Memory address label	Operation	Addressing or data information
Assembler directives	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that generate machine instructions	START	MOVE	N,R1
		MOVE	#NUM1,R2
		CLR	R0
	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
Assembler directives		RETURN	
		END	START

Assembly language representation for the program

Assembly and Execution of Programs

- A source program written in an assembly language must be assembled into a machine language object program before it can be executed.

- This is done by the assembler program, which replaces all symbols denoting operations and addressing modes with the binary codes used in machine instructions, and replaces all names and labels with their actual values.

- A key part of the assembly process is determining the values that replace the names. Assembler keep track of Symbolic name and Label name, create table called symbol table.

- The symbol table created by scan the source program twice.

- A branch instruction is usually implemented in machine code by specifying the branch target as the distance (in bytes) from the present address in the Program Counter to the target instruction.

- The assembler computes this branch offset, which can be positive or negative, and puts it into the machine instruction.

- The assembler stores the object program on the secondary storage device available in the computer, usually a magnetic disk.

- The object program must be loaded into the main memory before it is executed. For this to happen, another utility program called a loader must already be in the memory.

- Executing the loader performs a sequence of input operations needed to transfer the machine-language program from the disk into a specified place in the memory.

- The loader must know the length of the program and the address in the memory where it will be stored.

- The assembler usually places this information in a header preceding the object code (Like start/end offset address).

- When the object program begins executing, it proceeds to completion unless there are logical errors in the program.

- The user must be able to find errors easily. The assembler can only detect and report syntax errors.

- To help the user find other programming errors, the system software usually includes a debugger program.

- This program enables the user to stop execution of the object program at some points of interest and to examine the contents of various processor registers and memory locations.

Number Notation

Decimal Number

```
ADD #93,R1
```

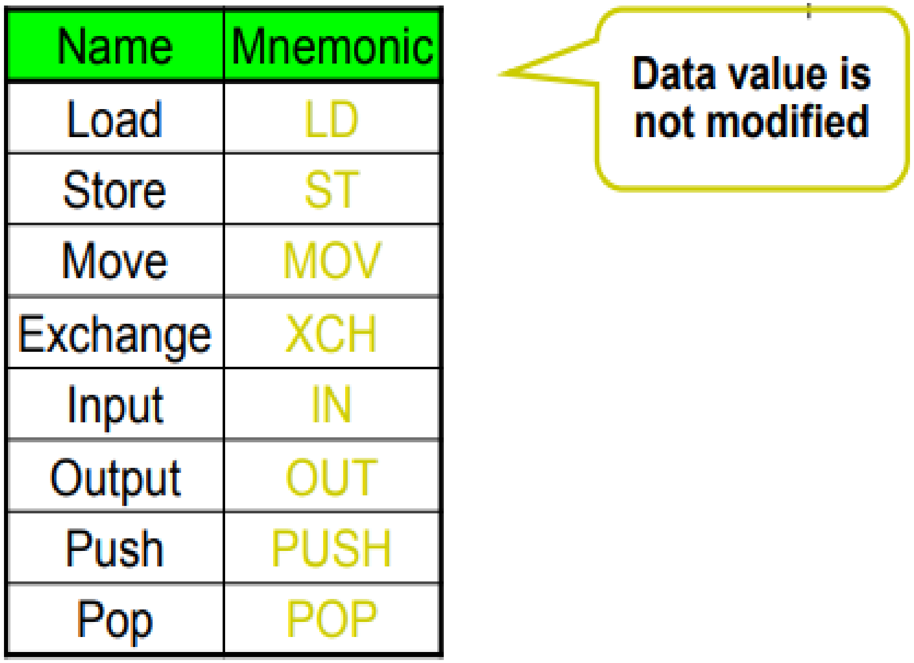
Binary Number

```
ADD #%0101110,R1
```

Hexadecimal Number

```
ADD #$5D,R1
```

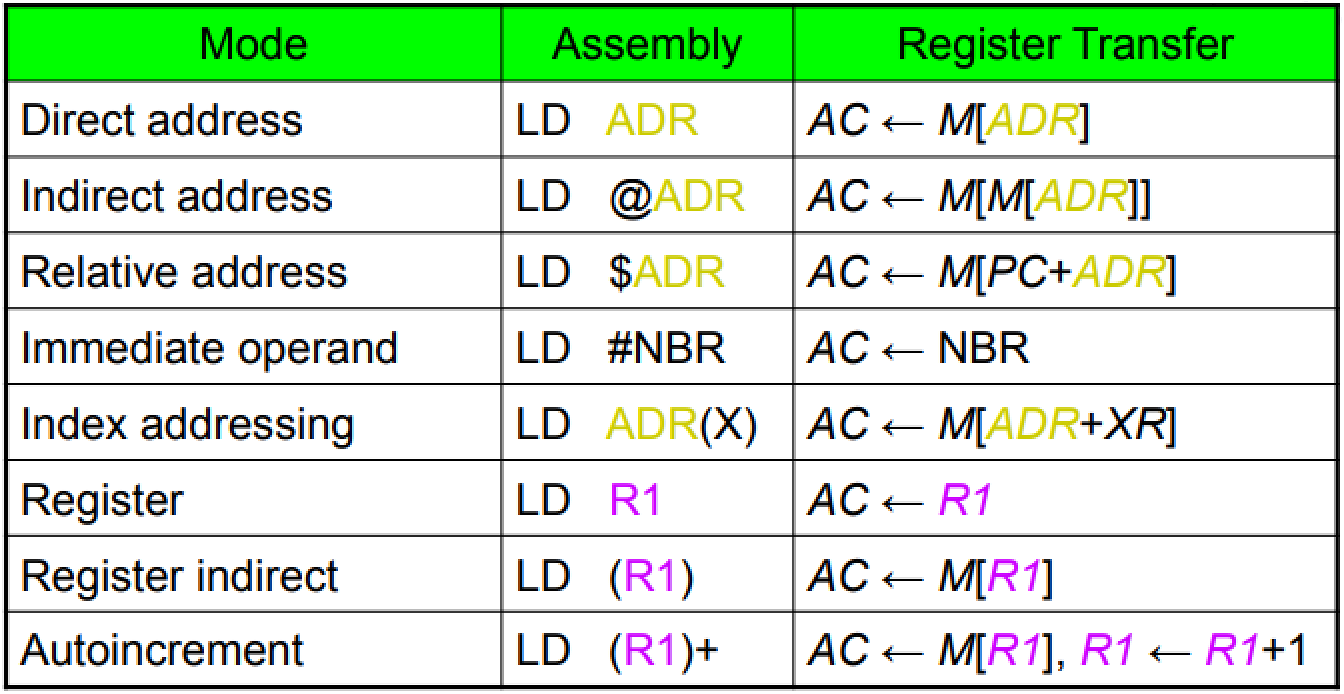
Types of Instructions
Data Transfer Instructions



Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Data value is not modified

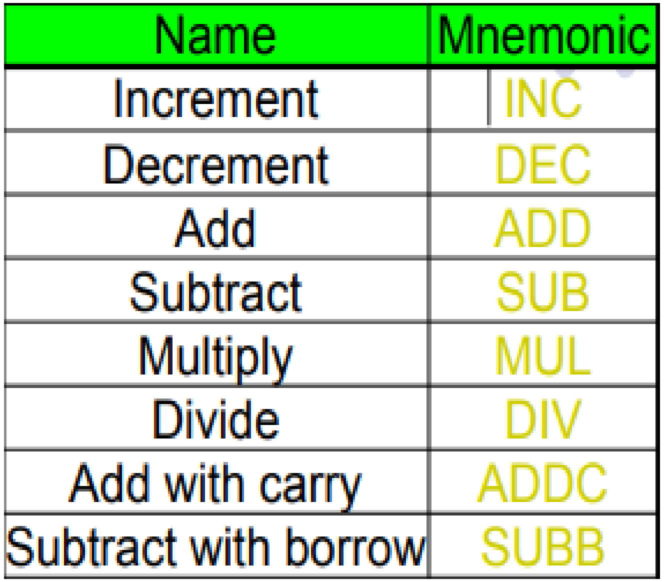
Data Transfer Instructions



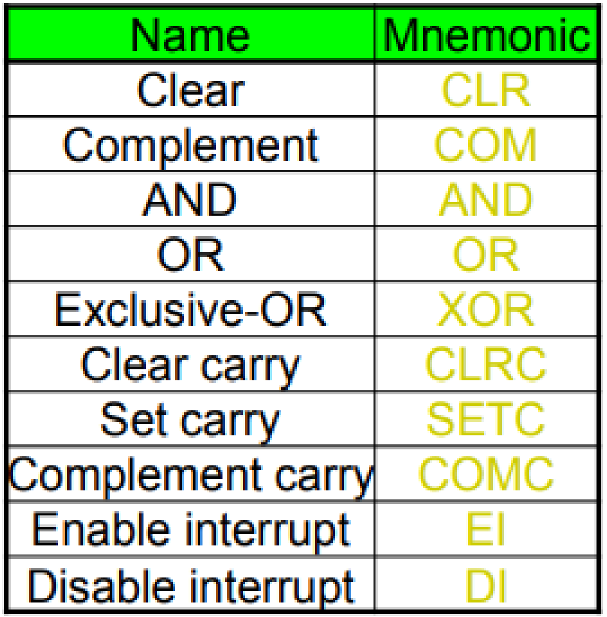
Mode	Assembly	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC+ADR]$
Immediate operand	LD #NBR	$AC \leftarrow \mathrm{NBR}$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR+XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1],\ R1 \leftarrow R1+1$

Data Manipulation Instructions

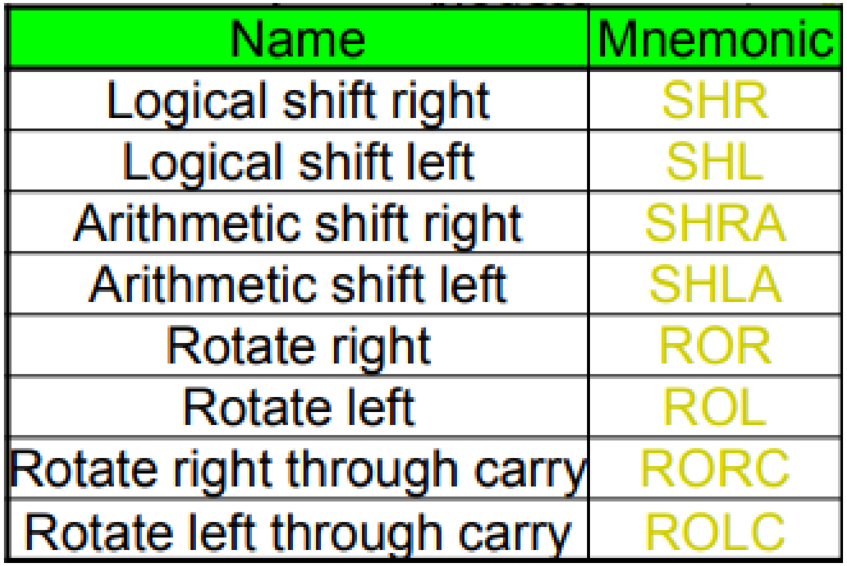
- Arithmetic
- Logical & Bit Manipulation
- Shift



Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB



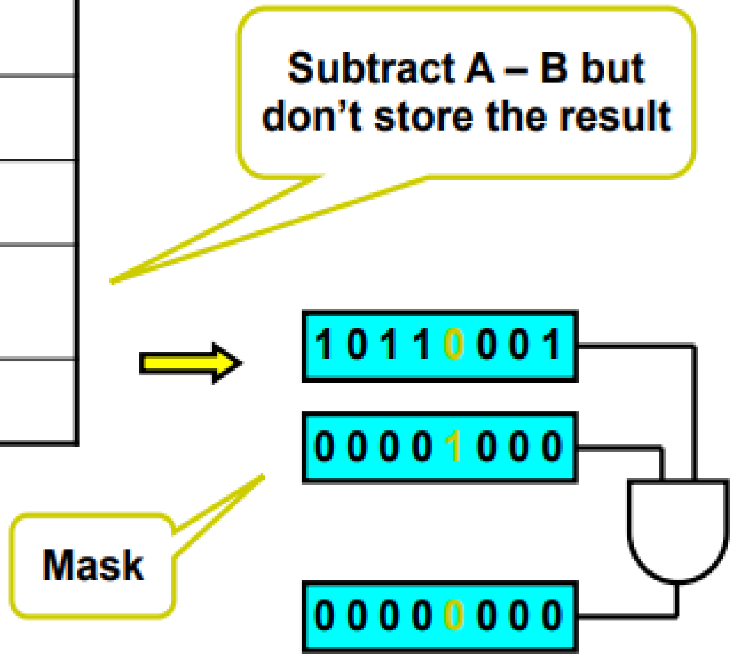
Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI



Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

Program Control Instructions

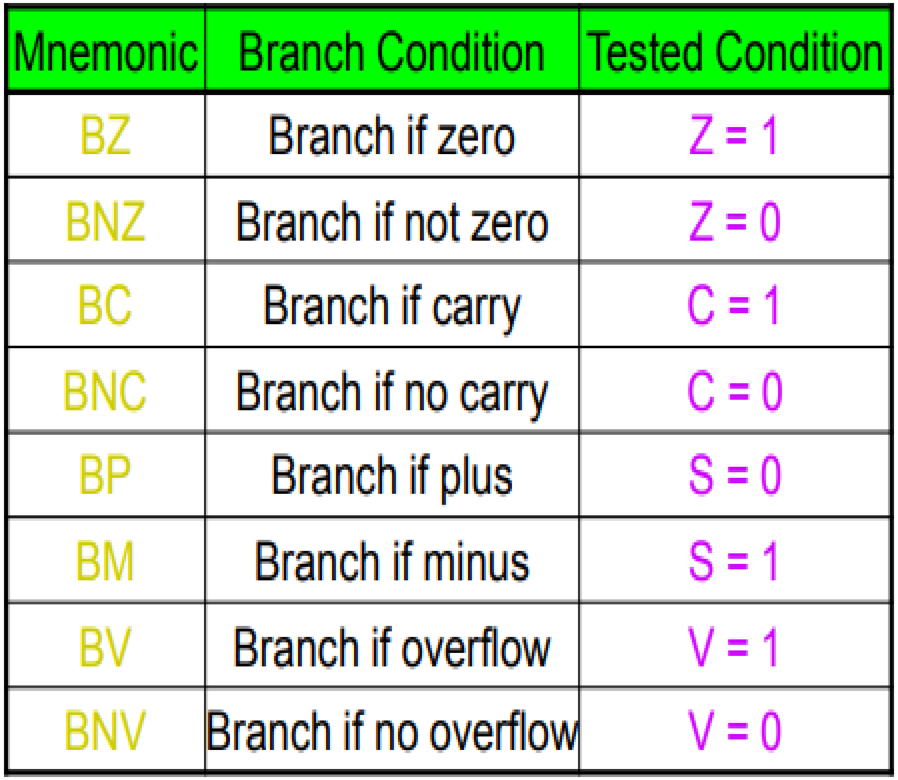
Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (Subtract)	CMP
Test (AND)	TST



Subtract A – B but don't store the result

Mask

Conditional Branch Instructions



Mnemonic	Branch Condition	Tested Condition
BZ	Branch if zero	Z = 1
BNZ	Branch if not zero	Z = 0
BC	Branch if carry	C = 1
BNC	Branch if no carry	C = 0
BP	Branch if plus	S = 0
BM	Branch if minus	S = 1
BV	Branch if overflow	V = 1
BNV	Branch if no overflow	V = 0

STACK

Stacks

- A stack is a list of data elements, usually words, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom. The structure is sometimes referred to as a pushdown stack.

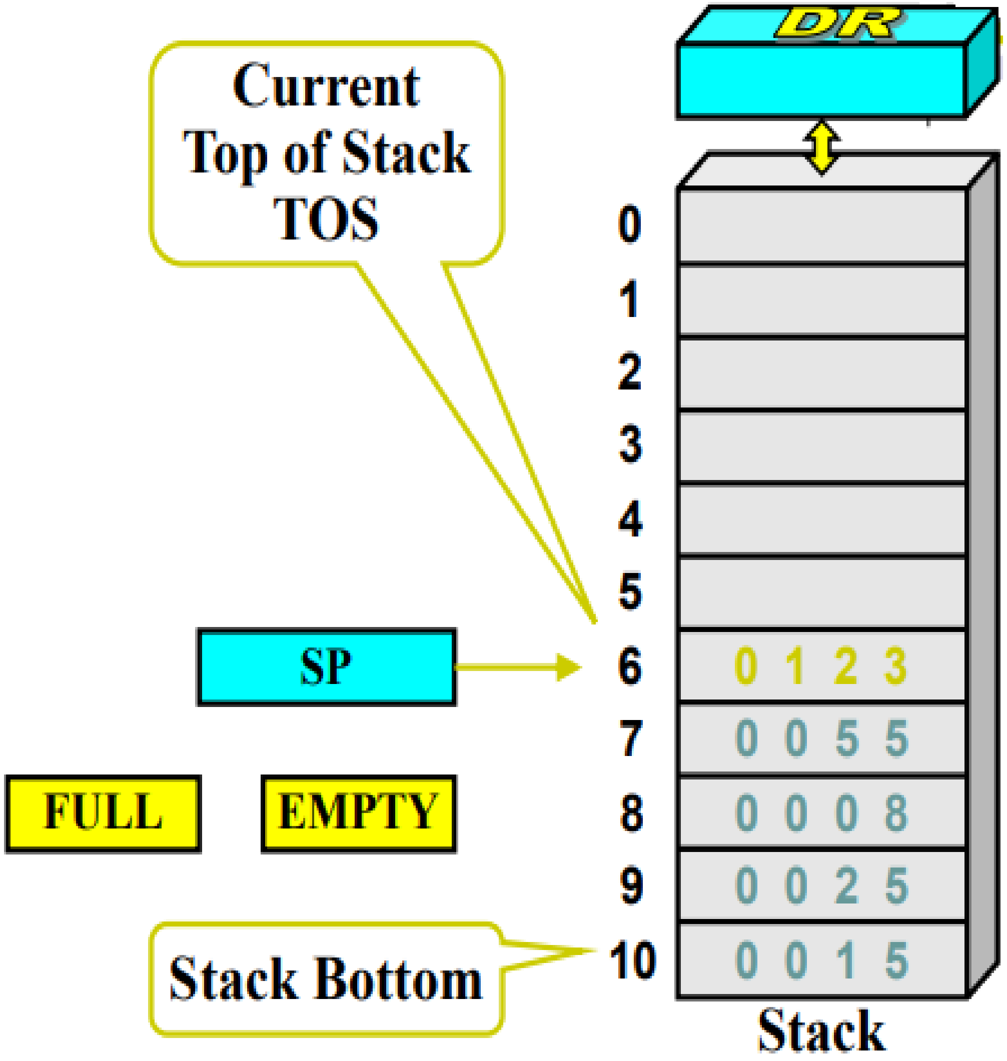
- Last-in–first-out (LIFO) stack working.

- The terms push and pop are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

- The stack pointer, SP, is used to keep track of the address of the element of the stack that is at the top at any given time.

Stack Organization

LIFO
Last In First Out



DR

Current
Top of Stack
TOS

SP

FULL

EMPTY

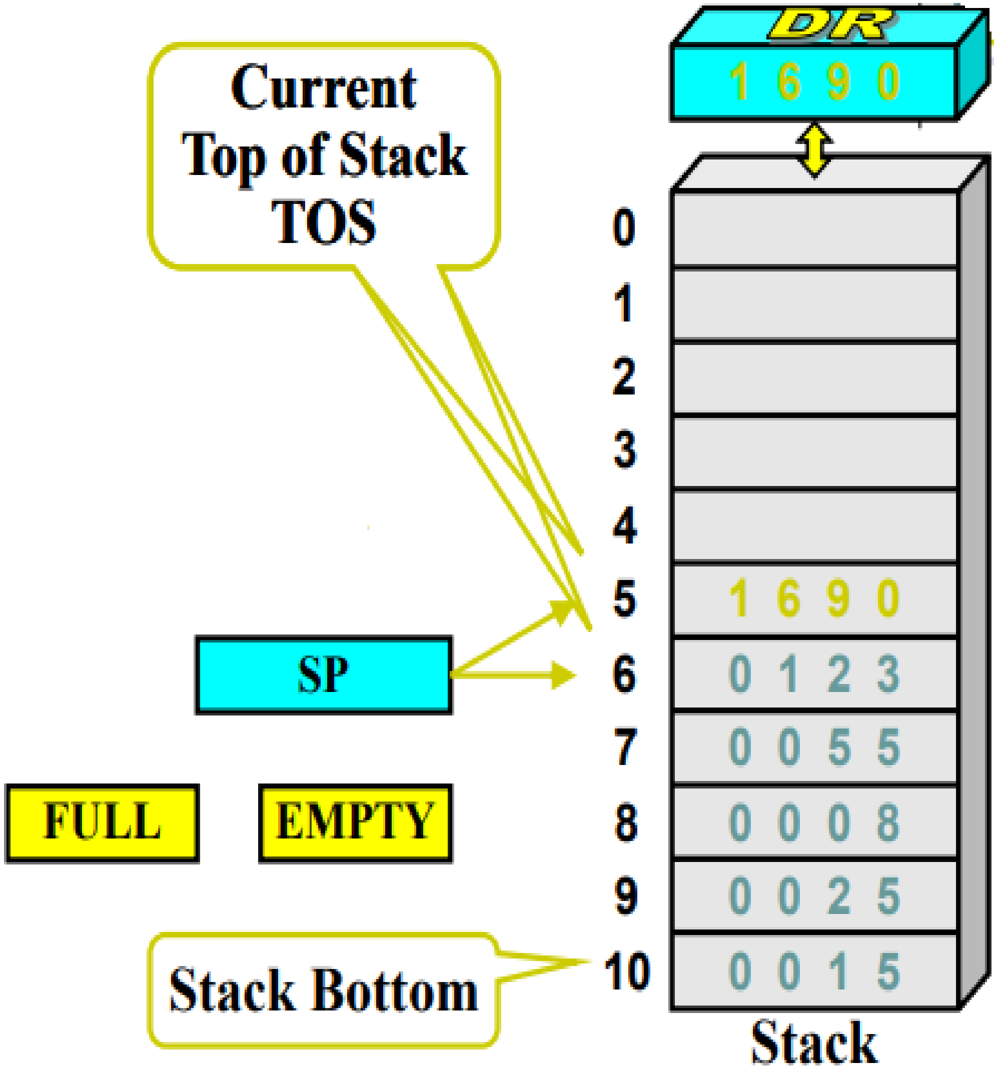
Stack Bottom

0	
1	
2	
3	
4	
5	
6	0 1 2 3
7	0 0 5 5
8	0 0 0 8
9	0 0 2 5
10	0 0 1 5

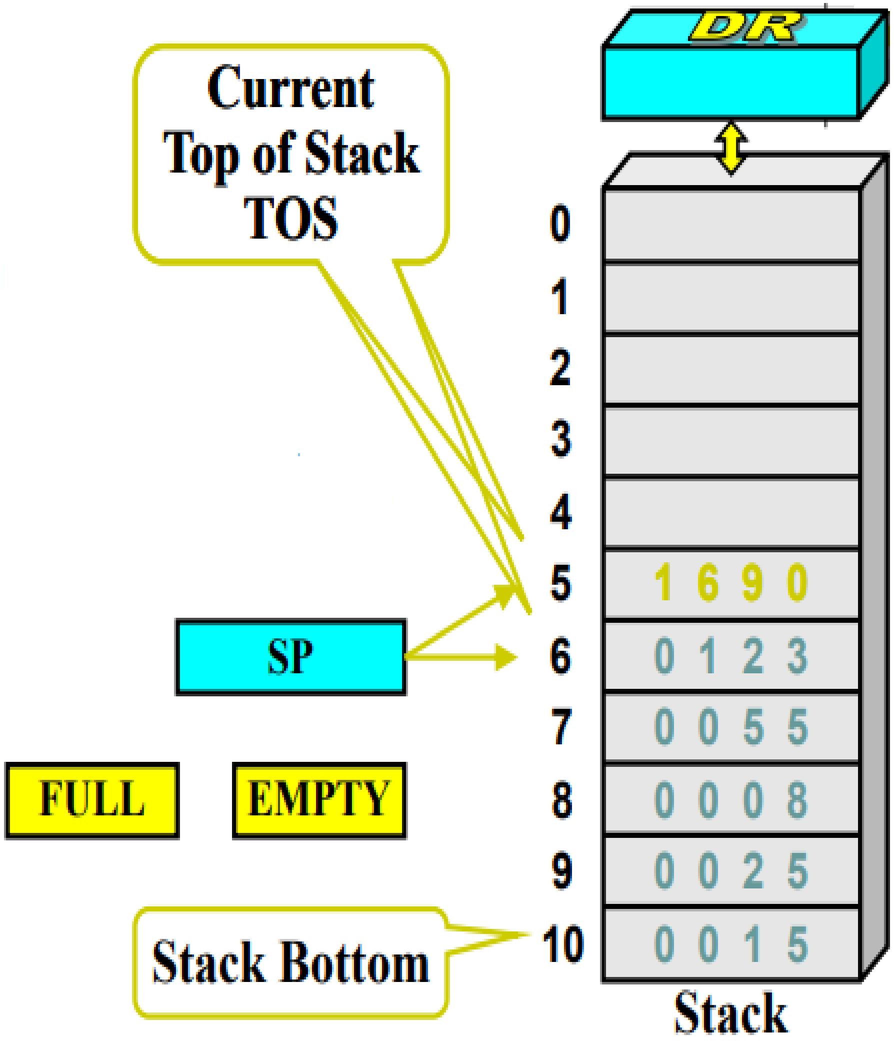
Stack

Stack Organization

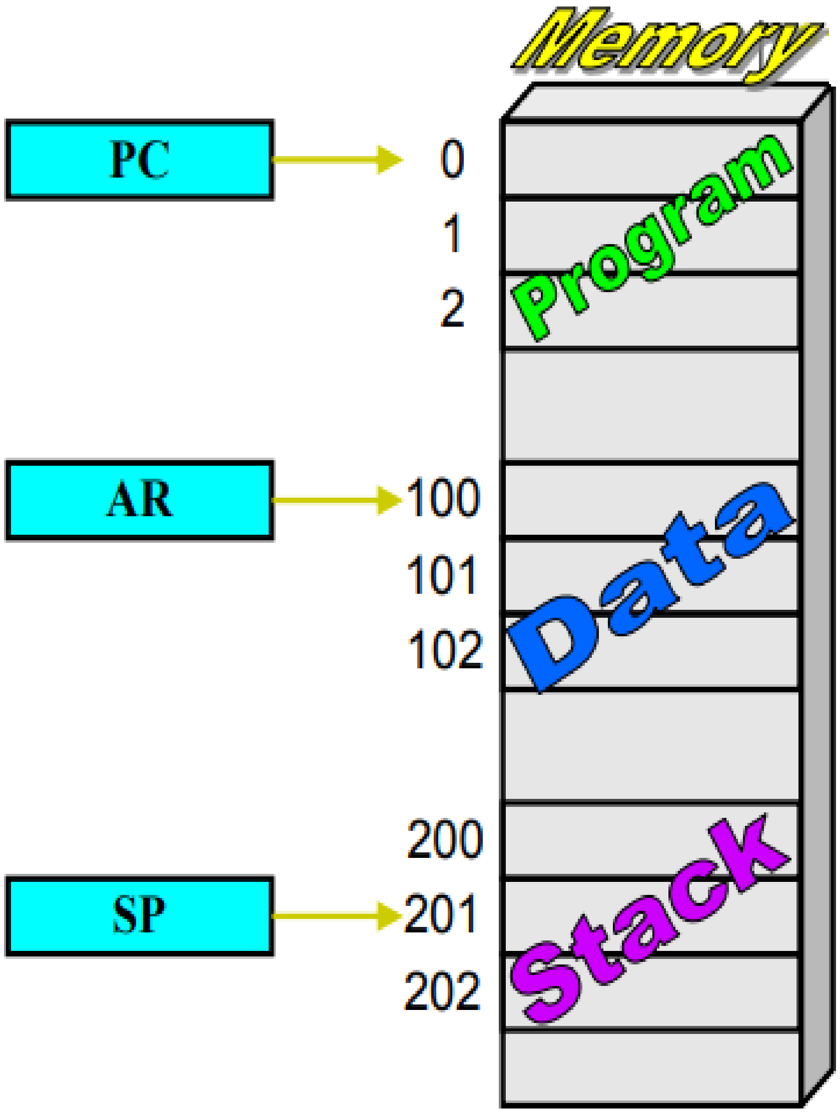
```
PUSH
 SP ← SP – 1
M[SP] ← DR
 If (SP = 0) then (FULI
 EMPTY ← 0
```



```
POP
DR ← M[SP]
 SP ← SP + 1
 If (SP = 11) then
(EMPTY ← 1)
 FULL ← 0
```



Memory Stack
PUSH SP ← SP – 1
M[SP] ← DR
POP DR ← M[SP]
SP ← SP + 1



SUBROUTINES

Subroutines

- In a given program, it is often necessary to perform a particular task many times on different data values.

- It is prudent to implement this task as a block of instructions that is executed each time the task has to be performed. Such a block of instructions is usually called a subroutine.

- However, to save space, only one copy of this block is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location.

- When a program branches to a subroutine we say that it is calling the subroutine.

- The instruction that performs this branch operation is named a Call instruction.

- After a subroutine has been executed, the calling program must resume execution, continuing immediately after the instruction that called the subroutine.

- The subroutine is said to return to the program that called it, and it does so by executing a Return instruction.

- Since the subroutine may be called from different places in a calling program, provision must be made for returning to the appropriate location.

- The location where the calling program resumes execution is the location pointed to by the updated program counter (PC) while the Call instruction is being executed.

- Hence, the contents of the PC must be saved by the Call instruction to enable correct return to the calling program.

- The way in which a computer makes it possible to call and return from subroutines is referred to as its subroutine linkage method.

- The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function.

- Such a register is called the link register. When the subroutine completes its task, the Return instruction returns to the calling program by branching indirectly through the link register.

Subroutines

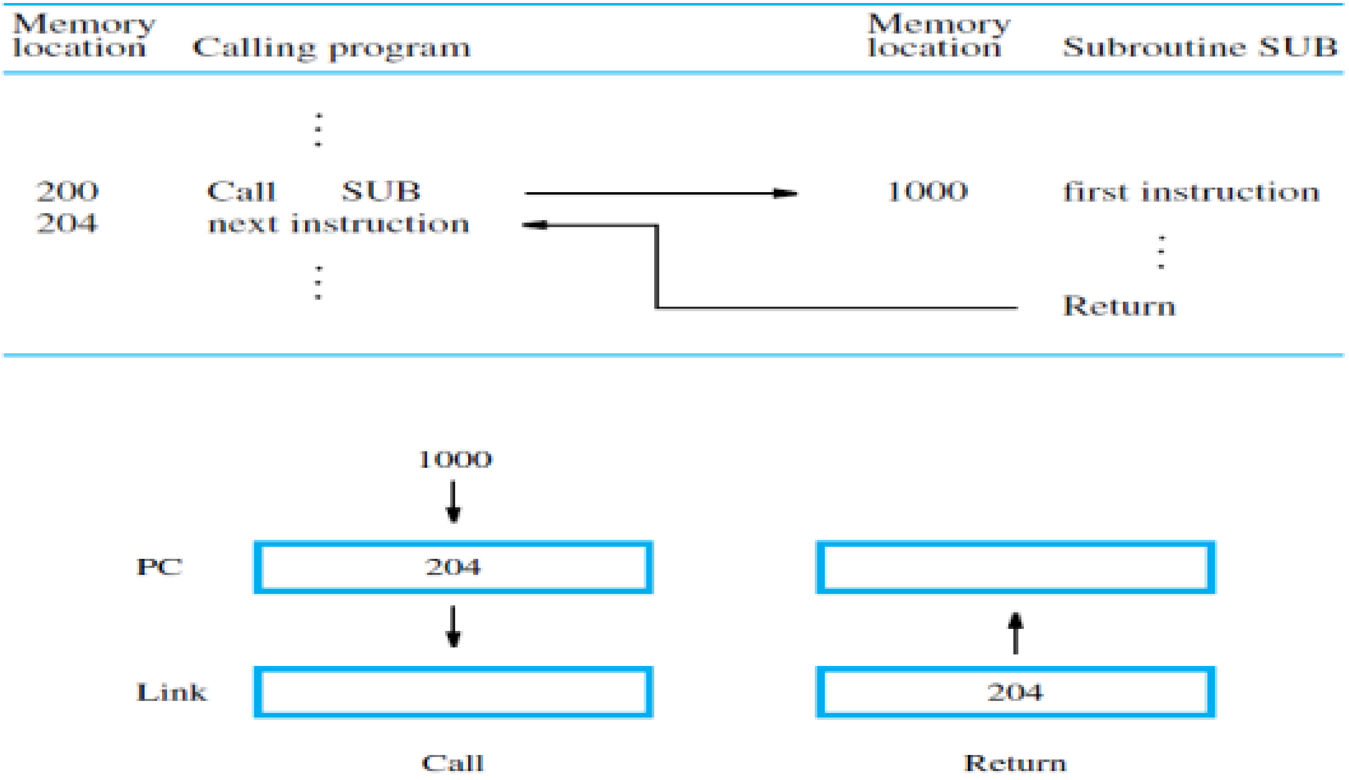
➢The Call instruction is just a special branch instruction that performs the following operations:

- Store the contents of the PC in the link register
- Branch to the target address specified by the Call instruction

➢ The Return instruction is a special branch instruction that performs the operation

- Branch to the address contained in the link register

Subroutines



Subroutine linkage using a link register.

Subroutine Nesting and the Processor Stack

- A common programming practice, called subroutine nesting, is to have one subroutine call another.

- In this case, the return address of the second call is also stored in the link register, overwriting its previous contents.

- Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. Otherwise, the return address of the first subroutine will be lost.

- That is, return addresses are generated and used in a last-in–first-out order. This suggests that the return addresses associated with subroutine calls should be pushed onto the processor stack.

Parameter Passing

- When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation.

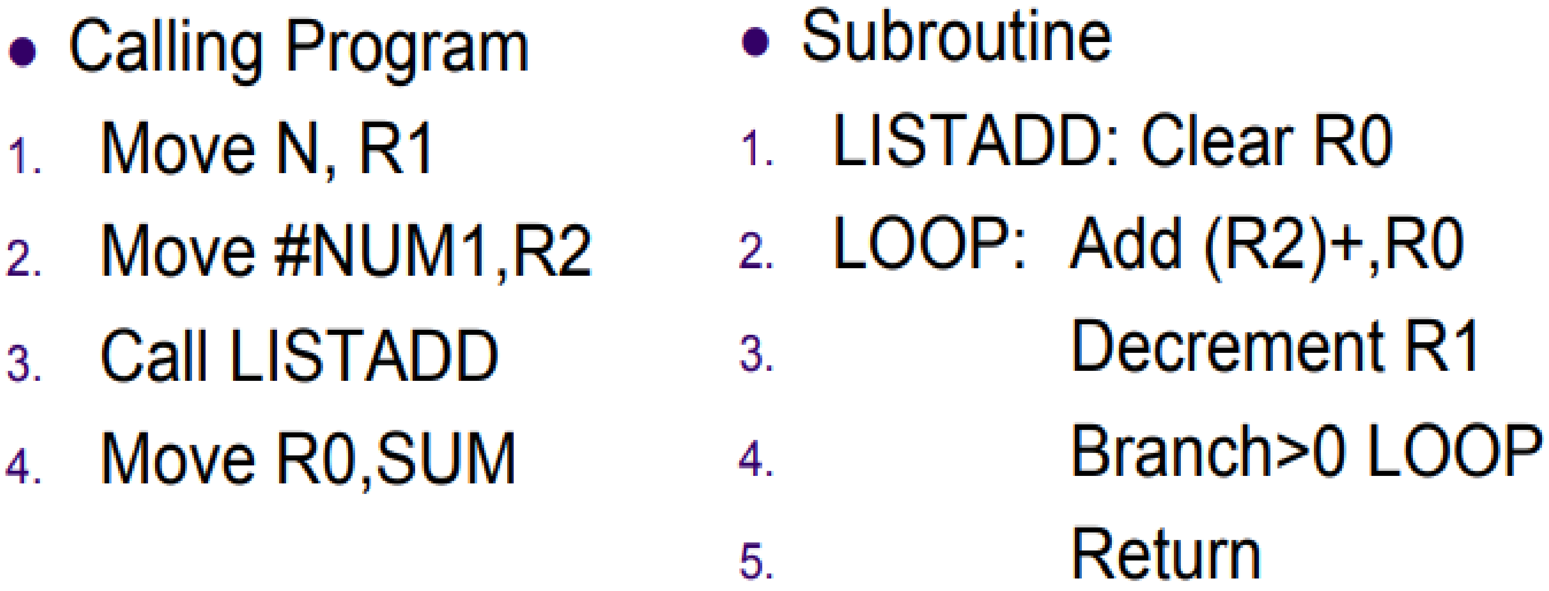
- Later, the subroutine returns other parameters, which are the results of the computation.

- This exchange of information between a calling program and a subroutine is referred to as parameter passing.

- Parameter passing may be accomplished in several ways. The parameters may be placed in registers, in memory locations, or on the processor stack where they can be accessed by the subroutine.

Program of subroutine

Parameters passed through registers.



- Calling Program

```
1.  Move N, R1
2.  Move #NUM1,R2
3.  Call LISTADD
4.  Move R0,SUM
```

- Subroutine

```
1.  LISTADD: Clear R0
2.  LOOP:   Add (R2)+,R0
3.           Decrement R1
4.           Branch>0 LOOP
5.           Return
```

Parameter Passing by Value and by Reference

- Instead of passing the actual Value(s), the calling program passes the address of the Value(s). This technique is called passing by reference.

- The second parameter is passed by value, that is, the actual number of entries, is passed to the subroutine.

The Stack Frame

- If the subroutine requires more space for local memory variables, the space for these variables can also be allocated on the stack this area of stack is called Stack Frame.

- For e.g. during execution of the subroutine, six locations at the top of the stack contain entries that are needed by the subroutine.

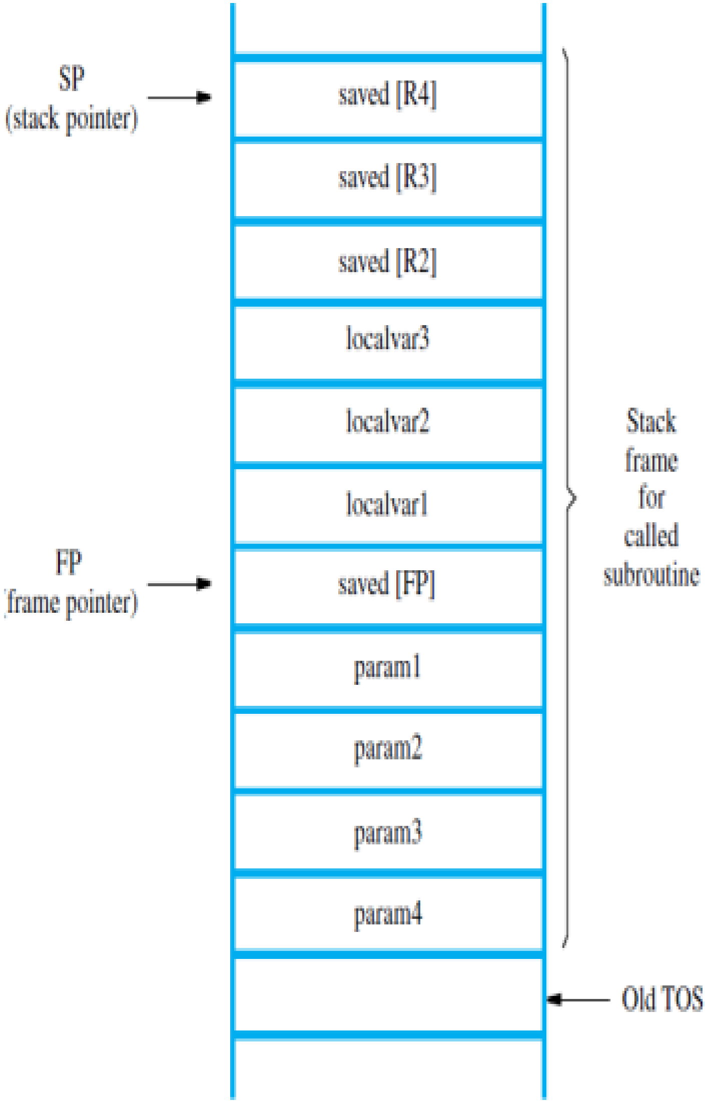
- These locations constitute a private work space for the subroutine, allocated at the time the subroutine is entered and deallocated when the subroutine returns control to the calling program.

The Stack Frame

➢ Frame pointer (FP), for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine .

➢ In the figure, we assume that four parameters are passed to the subroutine, three local variables are used within the subroutine, and registers R 2 , R 3 , and R 4 need to be saved because they will also be used within the subroutine .

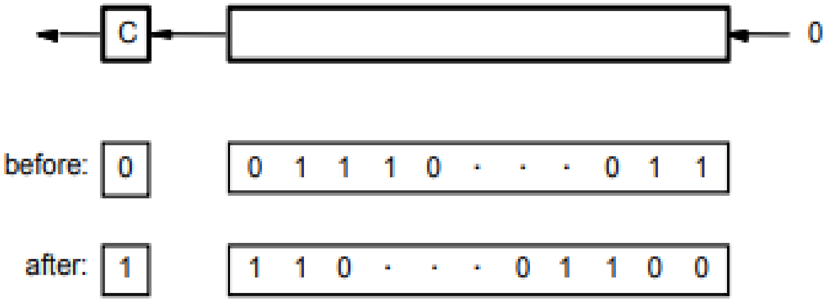
➢When nested subroutines are used, the stack frame of the calling subroutine would also include the return address.



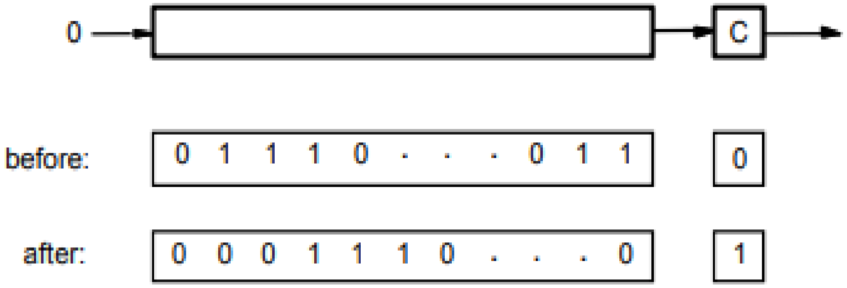
ADDITIONAL INSTRUCTIONS

Logical Shifts

Logical shift – shifting left (LShiftL) and shifting right (LShiftR)

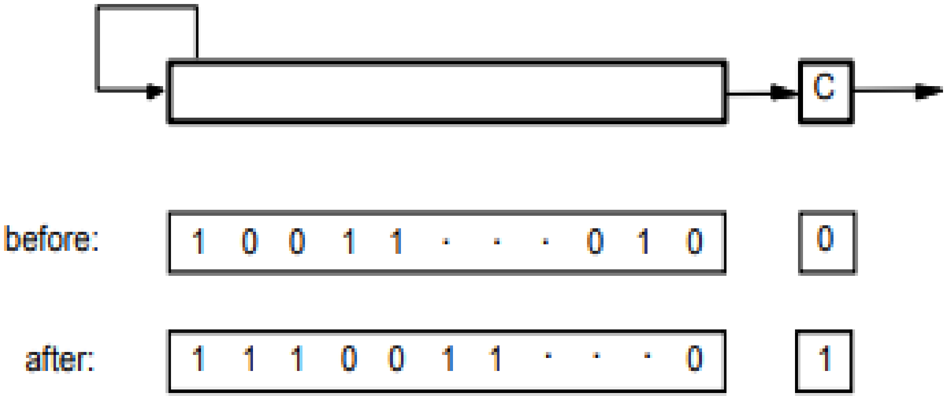


(a) Logical shift left LShiftL #2,R0



(b) Logical shift r ight LShiftR #2,R0

Arithmetic Shifts

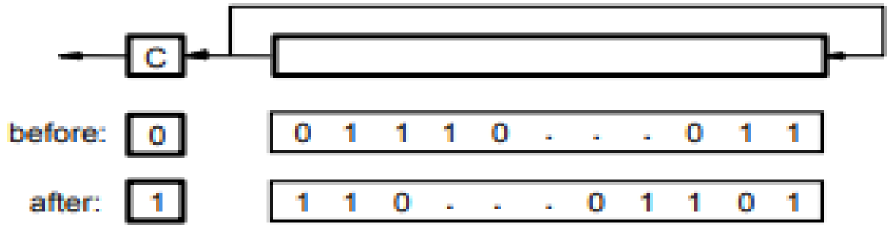


before: 1 0 0 1 1 · · · 0 1 0 0

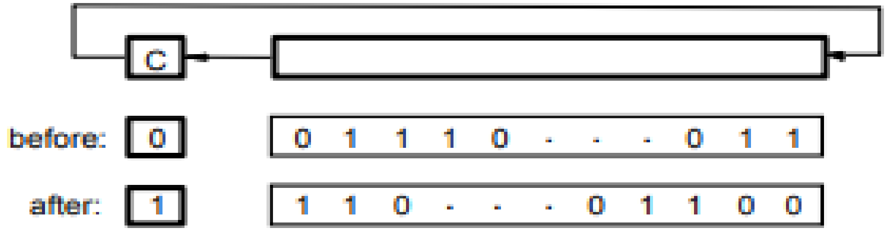
after: 1 1 1 0 0 1 1 · · · 0 1

(c) Arithmetic shift right AShiftR #2,R0

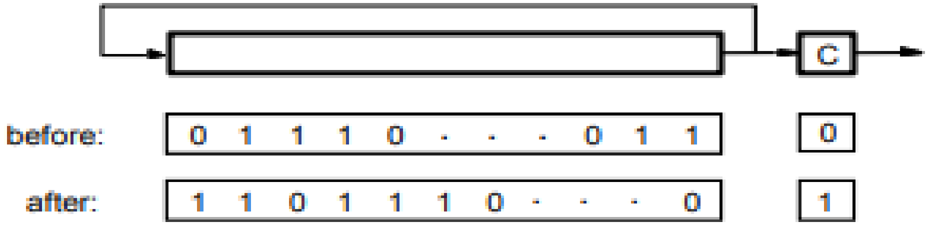
Rotate



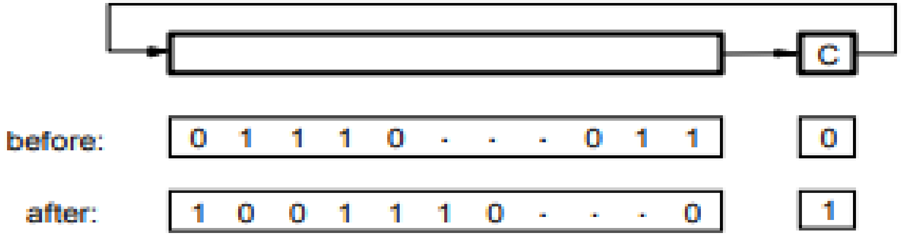
(a) Rotate left without carry RotateL #2,R0



(b) Rotate left with carry RotateLC #2,R0

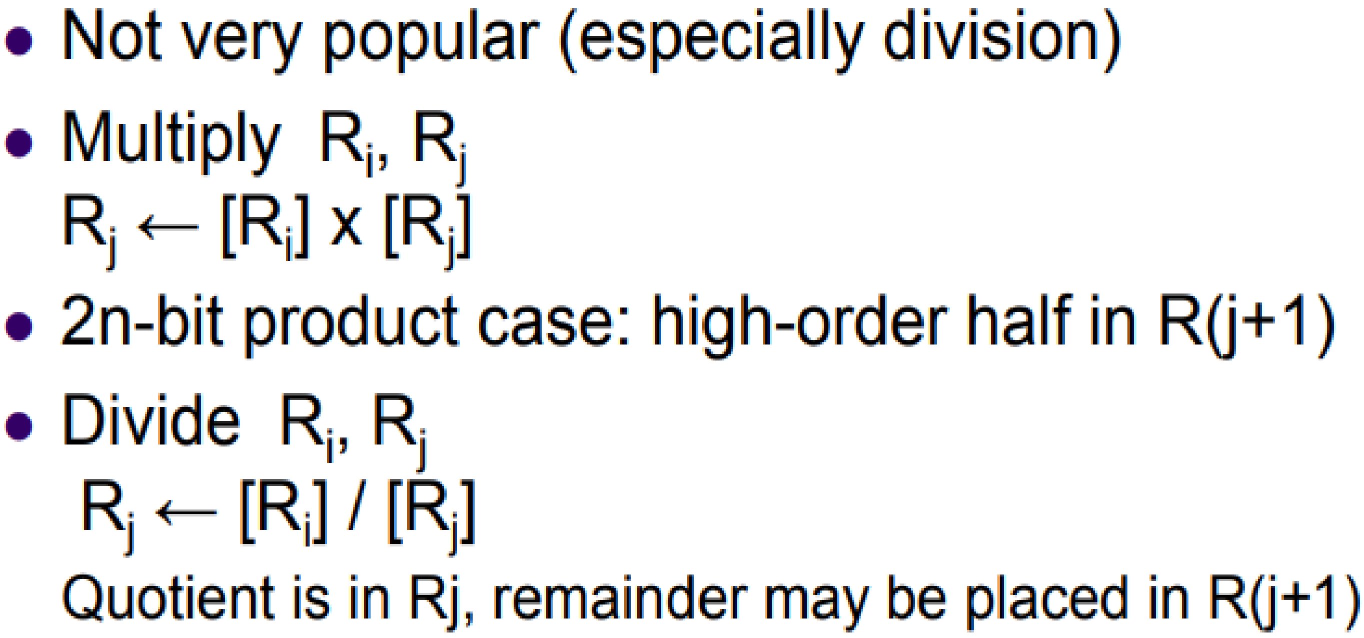


(c) Rotate right without carry RotateR #2,R0



(d) Rotate right with carry RotateRC #2,R0

Multiplication and Division



- Not very popular (especially division)
- Multiply R_i, R_j

  $R_j \leftarrow [R_i] \times [R_j]$
- 2n-bit product case: high-order half in R(j+1)
- Divide R_i, R_j

  $R_j \leftarrow [R_i] / [R_j]$

  Quotient is in Rj, remainder may be placed in R(j+1)

Logic Instructions

- And R2, R3, R4
- And #Value, R4, R2
- And #$0FF, R2, R2,

ENCODING OF MACHINE INSTRUCTIONS

Encoding of Machine Instructions

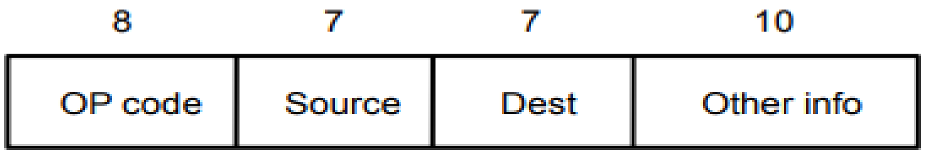
- Assembly language program needs to be converted into machine instructions. (ADD = 0100 in ARM instruction set)

- In the previous section, an assumption was made that all instructions are one word in length.

- OP code: the type of operation to be performed and the type of operands used may be specified using an encoded binary pattern

- Suppose 32-bit word length, 8-bit OP code (how many instructions can we have?), 16 registers in total (how many bits?), 3-bit addressing mode indicator.

Add R1, R2
Move 24(R0), R5
LshiftR #2, R0
Move #$3A, R1

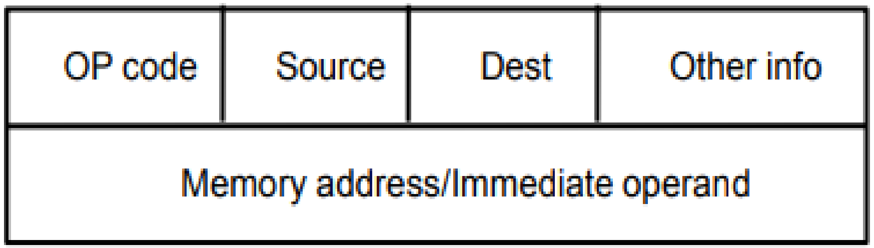


8	7	7	10
OP code	Source	Dest	Other info

One-word instruction

Encoding of Machine Instructions

- What happens if we want to specify a memory operand using the Absolute addressing mode?
- Move R2, LOC
- 14-bit for LOC – insufficient
- Solution – use two words



OP code	Source	Dest	Other info
Memory address/Immediate operand			

Two-word instruction

Encoding of Machine Instructions

- ➢Then what if an instruction in which two operands can be specified using the Absolute addressing mode?

- ➢ Move LOC1, LOC2

- ➢ Solution – use two additional words

- ➢This approach results in instructions of variable length.

- ➢Complex instructions can be implemented, closely resembling operations in high-level programming languages – Complex Instruction Set Computer (CISC)

Encoding of Machine Instructions

- If we insist that all instructions must fit into a single 32-bit word, it is not possible to provide a 32-bit address or a 32-bit immediate operand within the instruction.

- It is still possible to define a highly functional instruction set, which makes extensive use of the processor registers.

Add R1, R2 ----- yes
Add LOC, R2 ----- no
Add (R3), R2 ----- yes

CISC Instruction Sets

- Instructions in modern CISC processors typically do not use a three-address format.

- Most arithmetic and logic instructions use the two-address format

  Operation destination, source

- An Add instruction of this type is

  Add B, A

which performs the operation B ← [A] + [B] on memory operands

- In some CISC processors one operand may be in the memory but the other must be in a register. In this case, the instruction sequence for the required task would be

  Move Ri, A
  Add Ri, B
  Move C, Ri

Additional Addressing Modes
Autoincrement and Autodecrement Modes

Autoincrement mode—The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next operand in memory.

$$(Ri)+$$

Autodecrement mode—The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.

$$-(Ri)$$

Relative mode—The effective address is determined by the Index mode using the program counter in place of the general-purpose register Ri.

Condition Codes

➢Operations performed by the processor typically generate results such as numbers that are positive, negative, or zero.

➢ The processor can maintain the information about these results for use by subsequent conditional branch instructions.

➢ This is accomplished by recording the required information in individual bits, often called condition code flags.

➢ These flags are usually grouped together in a special processor register called the condition code register or status register.

➢ Individual condition code flags are set to 1 or cleared to 0, depending on the outcome of the operation performed.

Four commonly used flags are

- N (negative) Set to 1 if the result is negative; otherwise, cleared to 0
- Z (zero) Set to 1 if the result is 0; otherwise, cleared to 0
- V (overflow) Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
- C (carry) Set to 1 if a carry-out results from the operation; otherwise, cleared to 0

If condition codes are used, then the Subtract instruction would cause both N and Z flags to be cleared to 0 if the contents of register R2 are still greater than 0. The desired branching could be specified simply as

Branch>*0 LOOP*

without indicating the register involved in the test. This instruction causes a branch if neither N nor Z is 1, that is, if the result produced by the Subtract instruction is neither negative nor equal to zero.

Many conditional branch instructions are provided in the instruction set of a computer to enable a variety of conditions to be tested. The conditions are defined as logic expressions involving the condition code flags.

Text and Reference Books

Text Books:

- Carl Hamacher, Zvonko Vranesic, Safwat Zaky: Computer Organization, 5th Edition, Tata McGraw Hill, 2002.

- Carl Hamacher, Zvonko Vranesic, Safwat Zaky, Naraig Manjikian : Computer Organization and Embedded Systems, 6 th Edition, Tata McGraw Hill, 2012.

Reference Books:

- William Stallings: Computer Organization & Architecture, 9th Edition, Pearson, 2015.