Course – Computer Organization and Architecture

Course Instructor

Dr. Umadevi V

Department of CSE, BMSCE

Unit-1

Basic Structure of Computers and Instruction Set

Architecture: Functional Units, Basic Operational Concepts, Number Representation and Arithmetic Operations , Memory Locations and Addresses, Memory Operations, Instructions and Instruction Sequencing, Addressing Modes, Assembly Language

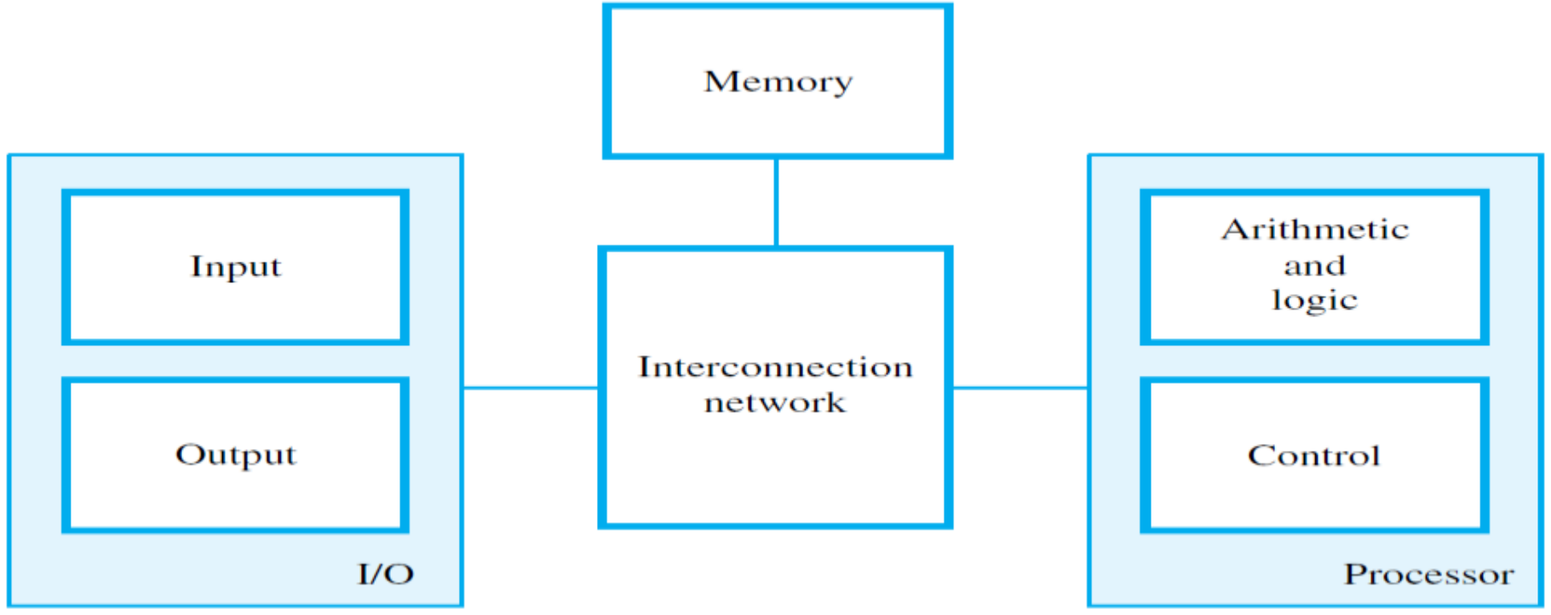
A Computer

Computer is a “machine”

- As long as power is supplied, processor keeps executing instructions
  - Stored program model
  - Sequential order of execution
- Memory: Program and data storage

Functional Units

A computer consists of 5 functionally independent main parts: **1)** Input **2)** Memory **3)** ALU (Arithmetic and Logic Unit) **4)** Output and **5)** Control unit



Basic functional units of a computer

Functional Units

1. Input Unit: Computer accepts encoded information through input unit. The standard input device is a keyboard. Whenever a key is pressed, keyboard controller sends the code to CPU/Memory.

- Examples include Keyborad, Mouse, Joystick, Tracker ball, Light pen, Digitizer, Scanner etc.

2. Output Unit: Computer after computation returns the computed results, error messages, etc. via output unit. The standard output device is a video monitor, LCD/TFT monitor. Other output devices are printers, plotters etc.

Functional Units

3. Memory Unit: Memory unit stores the program instructions (Code), data and results of computations etc. Memory unit is classified as:

- Primary /Main Memory
- Secondary /Auxiliary Memory

Primary memory is a semiconductor memory that provides access at high speed. Run time program instructions and operands are stored in the main memory. Main memory is classified again as ROM and RAM. ROM holds system programs and firmware routines such as BIOS, POST, I/O Drivers that are essential to manage the hardware of a computer. RAM is termed as Read/Write memory or user memory that holds run time program instruction and data. While primary storage is essential, it is volatile in nature and expensive. Additional requirement of memory could be supplied as auxiliary memory at cheaper cost. **Secondary memories** are non volatile in nature.

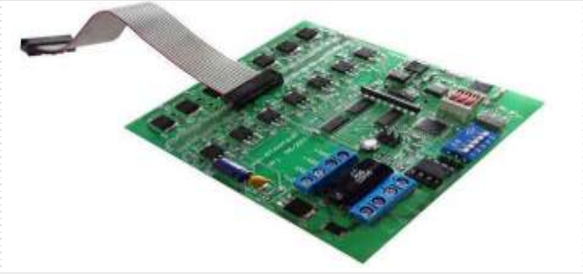
Functional Units

4. Arithmetic and logic unit: ALU consist of necessary logic circuits like adder, comparator etc., to perform operations of addition, multiplication, comparison of two numbers etc.

5. Control Unit: Control unit co-ordinates activities of all units by issuing control signals. Control signals issued by control unit govern the data transfers and then appropriate operations take place. Control unit interprets or decides the operation/action to be performed.

The control Unit and the Arithmetic and Logic unit of a computer system are jointly known as the Central Processing Unit (CPU).

Processor



Control Unit

Next we will learn

Unit 1: Basic Operational Concepts

Background information to understand “Basic Operational Concepts”

High Level Language (HLL)
Program **P.c**

```
#include <stdio.h>
main()
{
.
.
C=A+B;
.
.
}
```

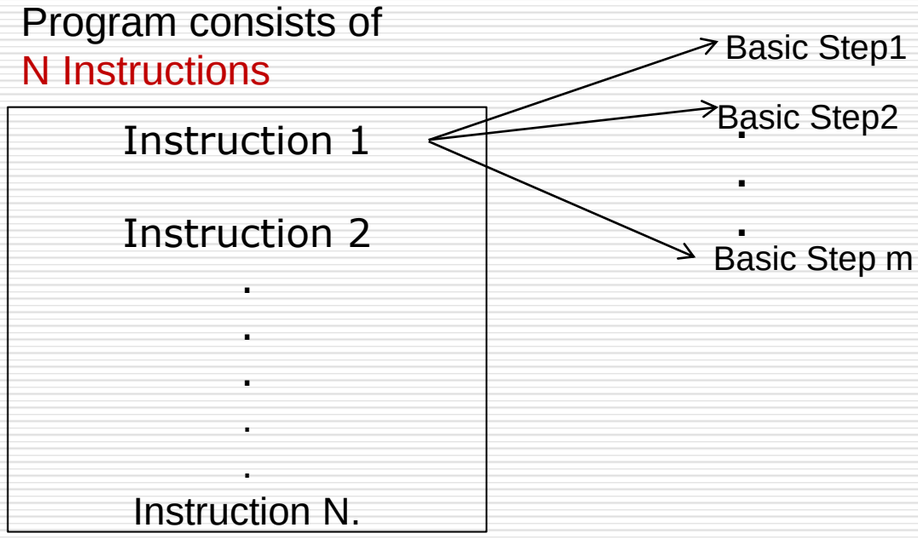
Assembly Level Language (ALL)
Program **T.asm**

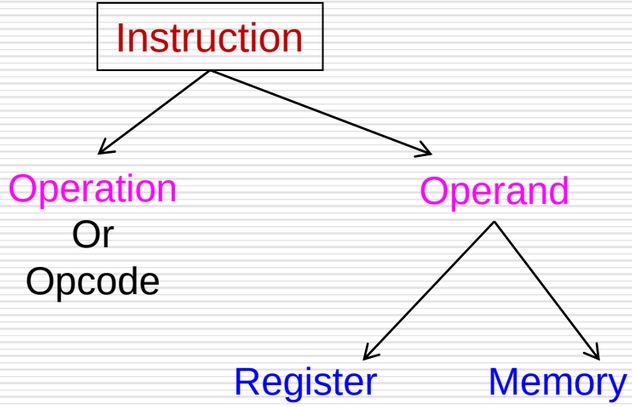
N Instructions

```
.
.
.
Load R2, A
Load R3, B
Add R4, R2, R3
Store R4, C
.
.
```

A, B, and C, are labels representing memory word addresses
Ri are processor registers

Background information to understand “Basic Operational Concepts”



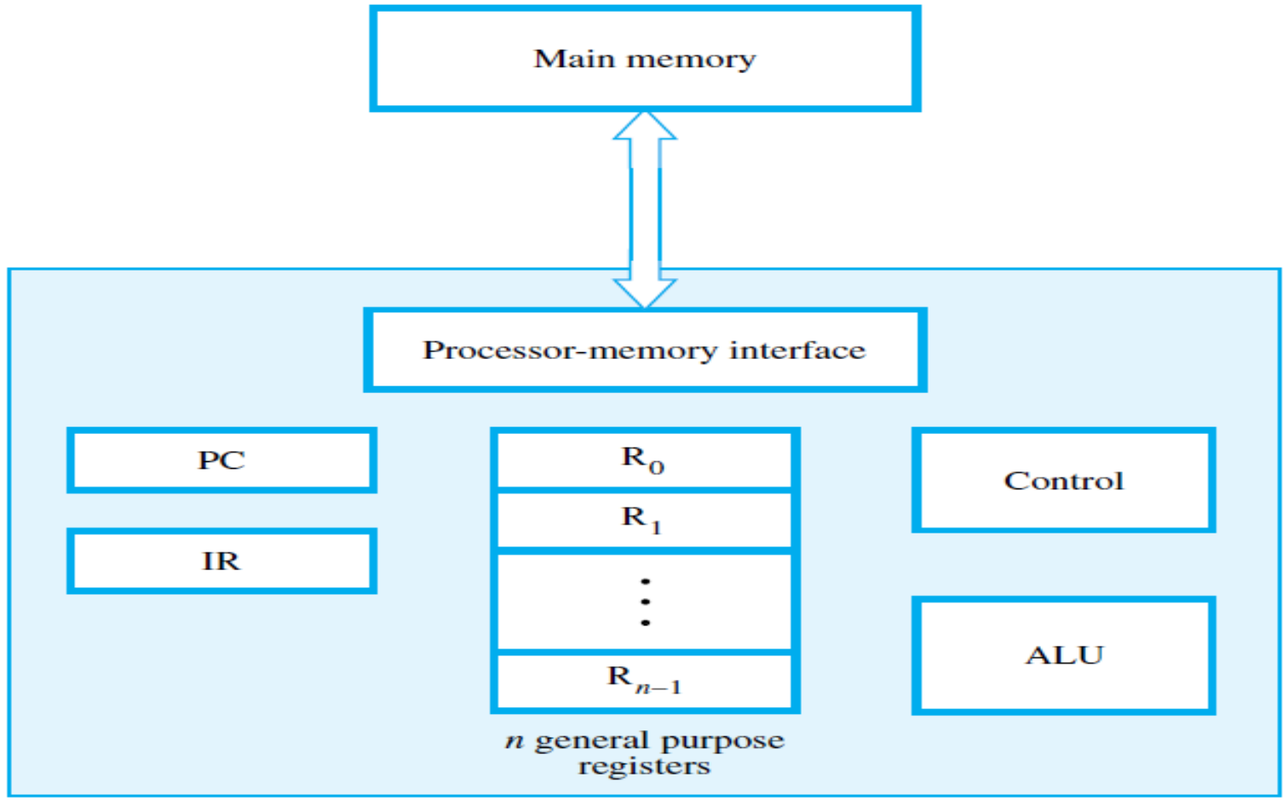


Background information to understand “Basic Operational Concepts”: Instruction types

Three basic instruction types:

- Load - Read a data operand from memory or an input device into the processor
- Store - Write a data operand from a processor register to memory or an output device
- Operate - Perform an arithmetic or logic operation on data operands in processor registers

BASIC OPERATIONAL CONCEPTS



- The Program Counter (PC) register holds the memory address of the current instruction
- The Instruction Register (IR) holds the current instruction
- General-purpose registers hold data and addresses (R0, R1, ….)
- Control circuits and the Arithmetic and Logic Unit (ALU) fetch and execute instructions

Connection Between the Processor and the Main Memory

Example: **Load R2, LOC**

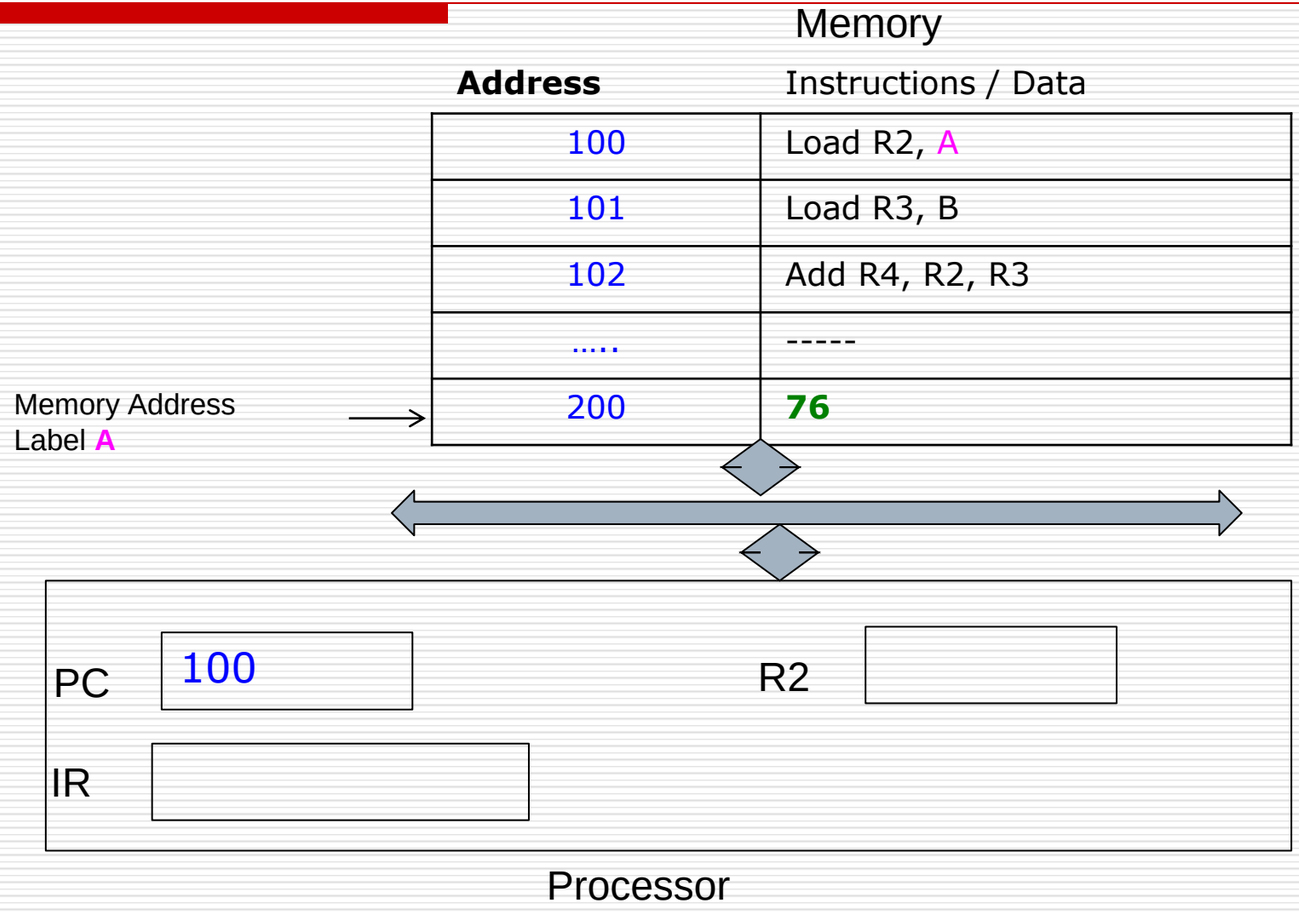
The processor control circuits do the following:

- Send address in **PC** to memory; issue Read
- Load instruction from memory into **IR**
- Increment **PC** to point to next instruction
- Send address **LOC** to memory; issue Read
- Load word from memory into register **R2**

BASIC OPERATIONAL CONCEPTS: Fetching and executing instructions : Illustration with example

Basic Steps to execute the instruction **Load R2, A**

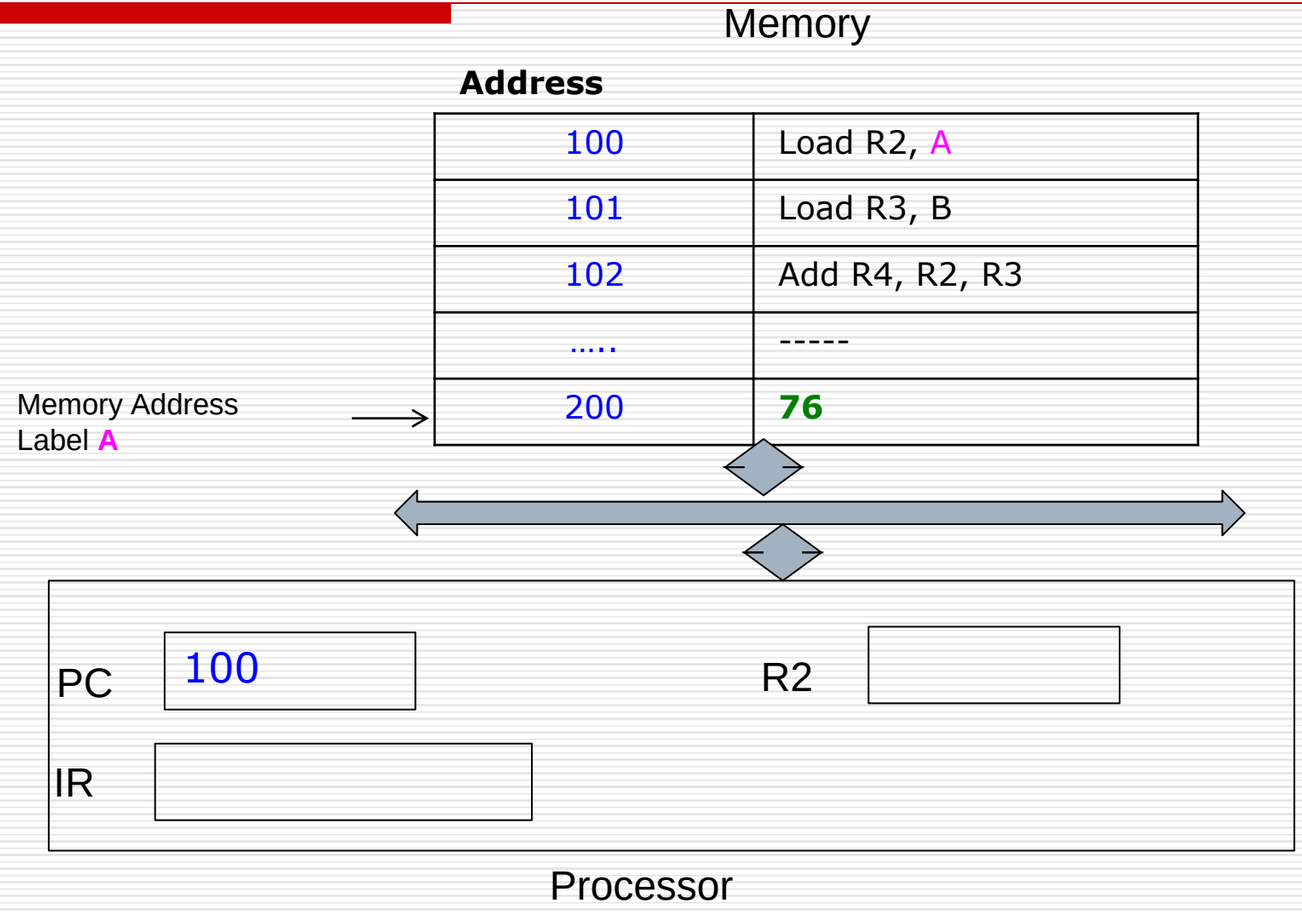
1. Send address in PC to memory; issue Read
2. Load instruction from memory into IR
3. Increment PC to point to next instruction
4. Send address A to memory; issue Read
5. Load word from memory into register R2



BASIC OPERATIONAL CONCEPTS: Fetching and executing instructions : Illustration with example

Basic Steps to execute the instruction **Load R2, A:**

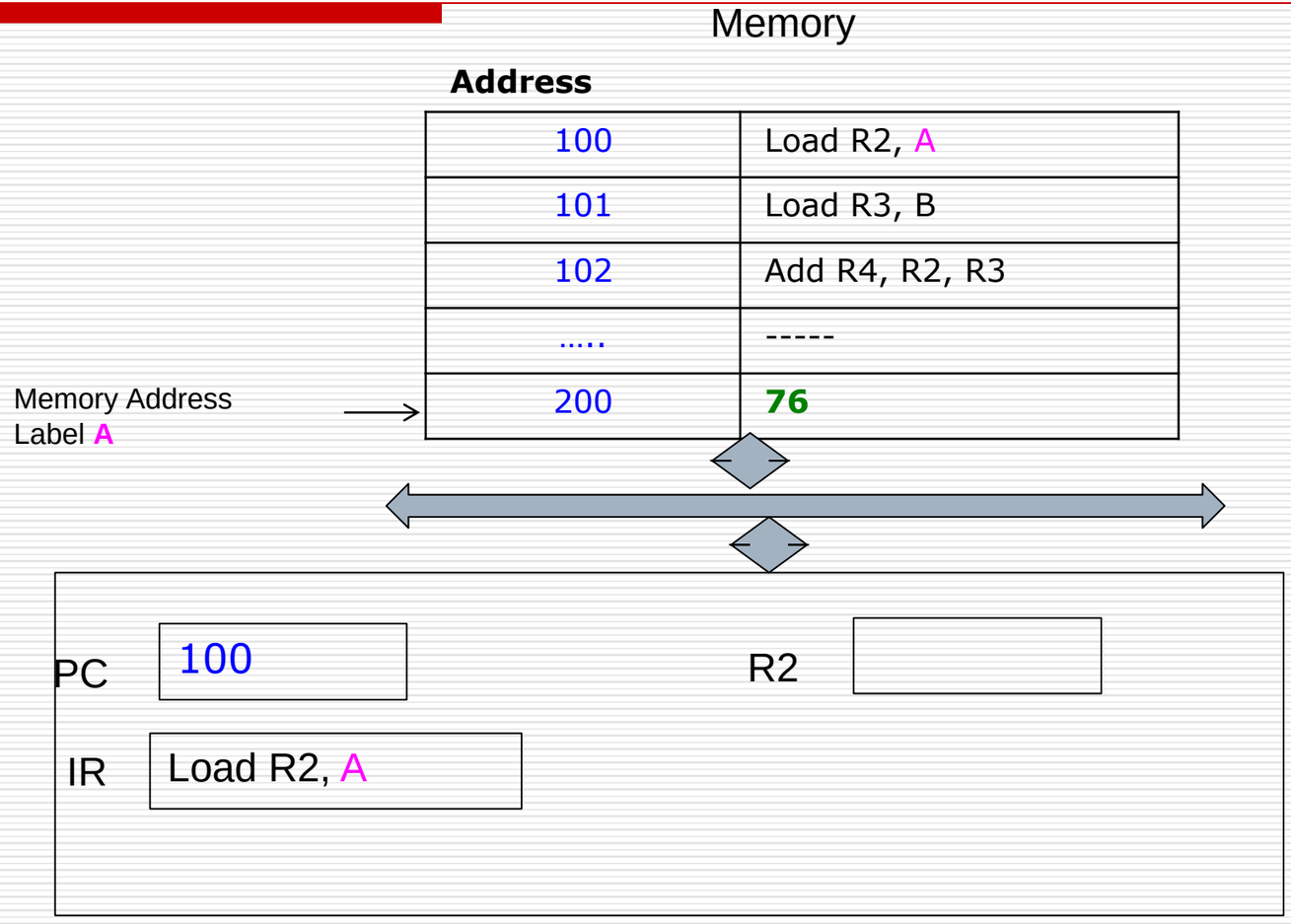
1. Send address in PC to memory; issue Read
2. Load instruction from memory into IR
3. Increment PC to point to next instruction
4. Send address A to memory; issue Read
5. Load word from memory into register R2



BASIC OPERATIONAL CONCEPTS: Fetching and executing instructions : Illustration with example

Basic Steps to execute the instruction **Load R2, A:**

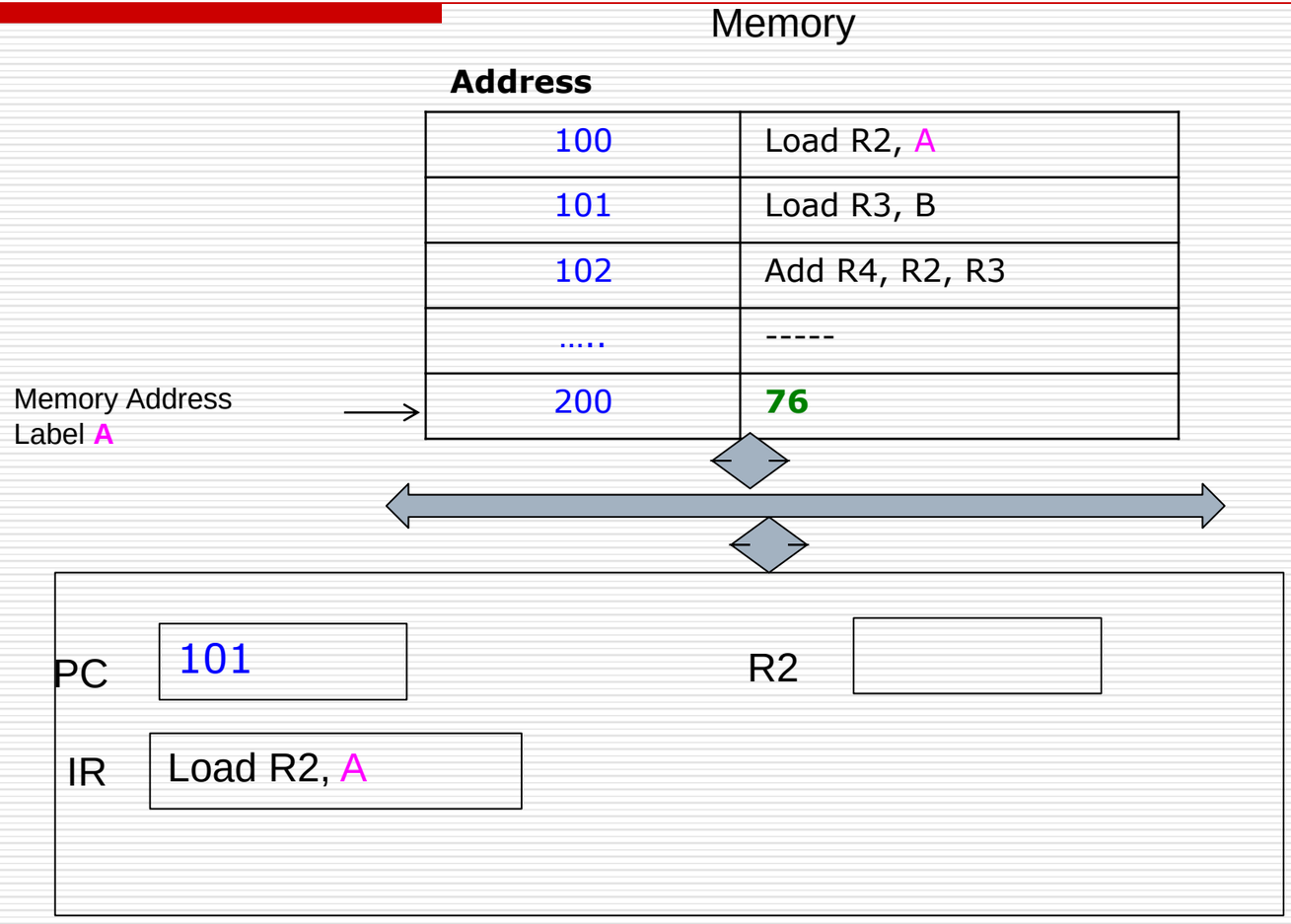
1. Send address in PC to memory; issue Read
2. Load instruction from memory into IR
3. Increment PC to point to next instruction
4. Send address A to memory; issue Read
5. Load word from memory into register R2



BASIC OPERATIONAL CONCEPTS: Fetching and executing instructions : Illustration with example

Basic Steps to execute the instruction **Load R2, A:**

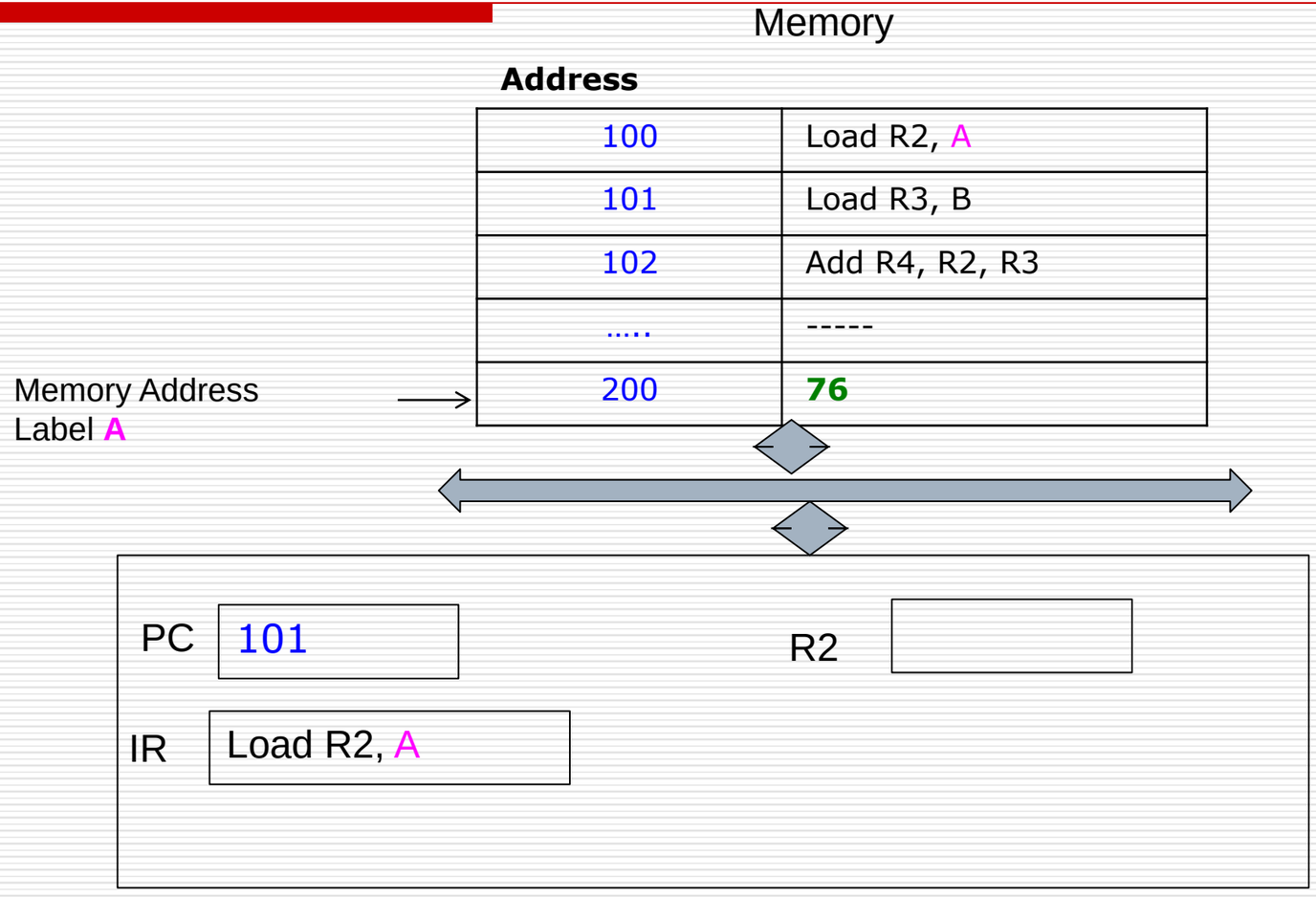
1. Send address in PC to memory; issue Read
2. Load instruction from memory into IR
3. Increment PC to point to next instruction
4. Send address A to memory; issue Read
5. Load word from memory into register R2



BASIC OPERATIONAL CONCEPTS: Fetching and executing instructions : Illustration with example

Basic Steps to execute the instruction **Load R2, A:**

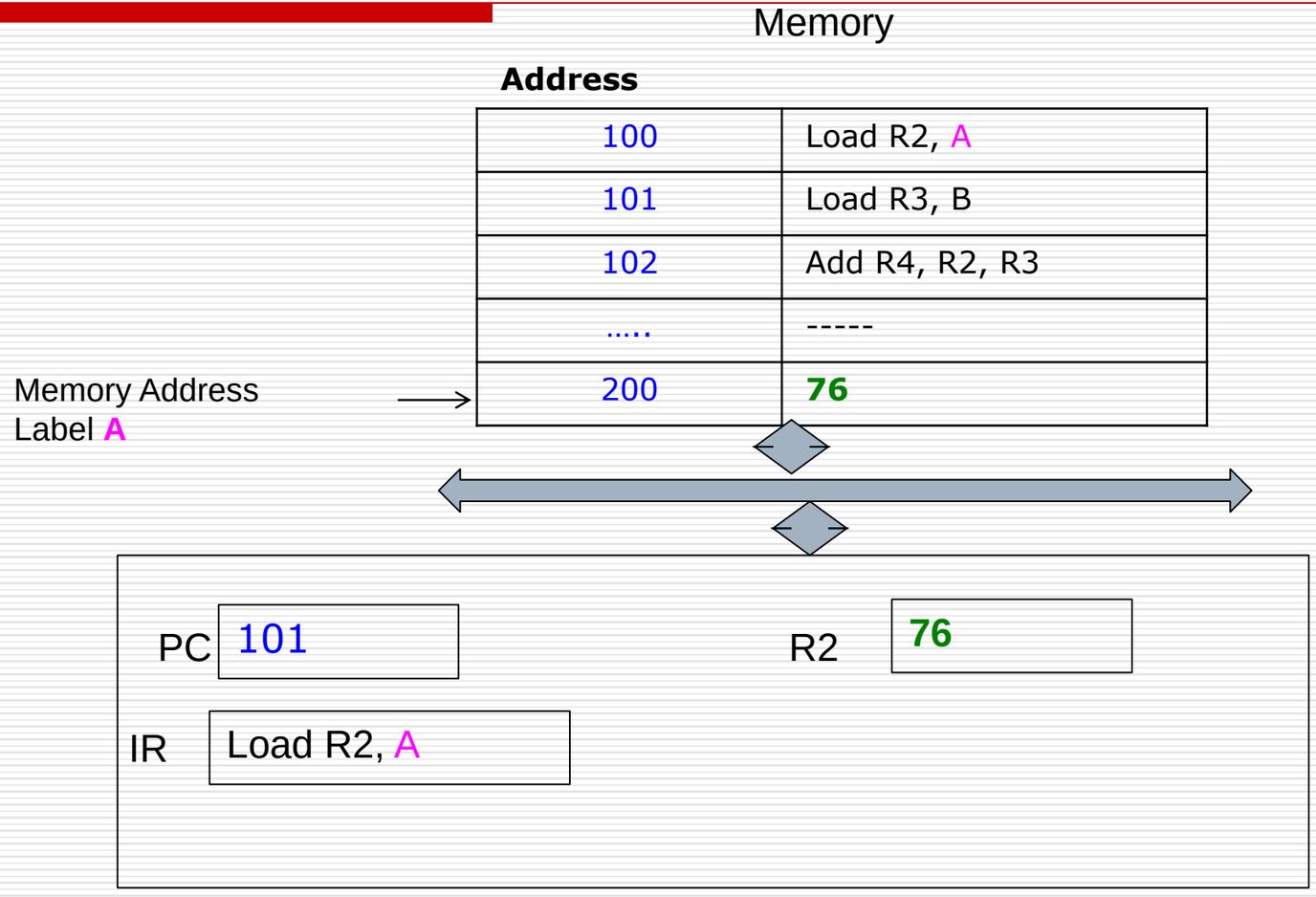
1. Send address in PC to memory; issue Read
2. Load instruction from memory into IR
3. Increment PC to point to next instruction
4. Send address A to memory; issue Read
5. Load word from memory into register R2



BASIC OPERATIONAL CONCEPTS: Fetching and executing instructions : Illustration with example

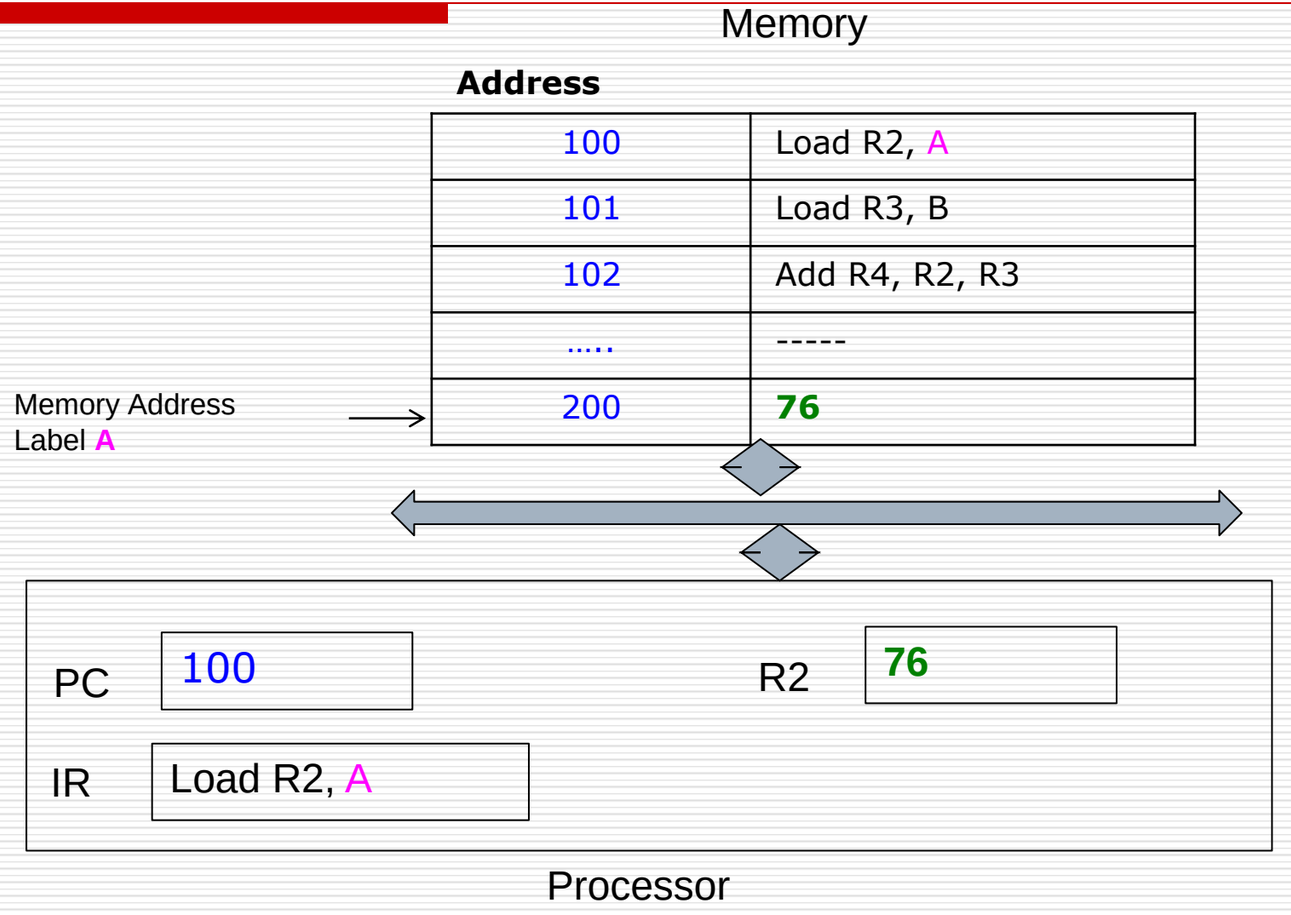
Basic Steps to execute the instruction **Load R2, A**:

1. Send address in PC to memory; issue Read
2. Load instruction from memory into IR
3. Increment PC to point to next instruction
4. Send address A to memory; issue Read
5. Load word from memory into register R2



BASIC OPERATIONAL CONCEPTS:
Fetching and executing instructions

Basic Steps to execute the instruction **Load R2, A**:
1. Send address in PC to memory; issue Read
2. Load instruction from memory into IR
3. Increment PC to point to next instruction
4. Send address A to memory; issue Read
5. Load word from memory into register R2



Test Your Knowledge

W.r.t Computer Processor,

What is the role of PC ?

What is the role of IR ?

Test Your Knowledge

What is the role of PC ?

- The Central Processing Unit (CPU) contains a register called the Program Counter (PC), which holds the address of instruction to be executed next.. to begin the execution of the program the address of its First instruction must be placed into the PC.

What is the role of IR ?

- The instruction register (IR) is used to hold the instruction that is currently being executed. The contents of IR are available to the control unit, which generate the timing signals that control the various processing elements involved in executing the instruction.

Example: **Store LOC, R4**

The processor control circuits do the following:

- Send address in **PC** to memory; issue Read
- Load instruction from memory into **IR**
- Increment **PC** to point to next instruction
- Send address **LOC** to memory; issue Write
- Store the word from register **R4** into **LOC**

Example: **Add R4, R2, R3**

The processor control circuits do the following:

- Send address in **PC** to memory; issue Read
- Load instruction from memory into **IR**
- Increment **PC** to point to next instruction
- **Add** the content of Register **R2** and the contents of register **R3**.
- Store the result (sum) in **R4**.

Question

(a) Give a short sequence of machine instructions for the task "Add the contents of memory-location A to those of location B, and place the answer in location C". Instructions:

Load Ri , LOC

and

Store LOC , Ri

are the only instructions available to transfer data between memory and the general purpose registers. Add instructions of type ADD LOCA, R0 and Add R1, R0 are available. Do not change contents of either location A or B.

(b) Suppose that Move and Add instructions are available with the formats:

Move Location1, Location2

and

Add Location1, Location2

These instructions move or add a copy of the operand at the second location to the first location, overwriting the original operand at the first location. Either or both of the operands can be in the memory or the general-purpose registers.

Is it possible to use fewer instructions of these types to accomplish the task in part (a)? If yes, give the sequence.

Answer

(a) Give a short sequence of machine instructions for the task "Add the contents of memory-location A to those of location B, and place the answer in location C". Instructions:

Load Ri , LOC

and

Store LOC , Ri

are the only instructions available to transfer data between memory and the general purpose registers. Add instructions of type ADD LOCA, R0 and Add R1, R0 are available. Do not change contents of either location A or B.

Answer

(a) Give a short sequence of machine instructions for the task "Add the contents of memory-location A to those of location B, and place the answer in location C". Instructions:

Load Ri , LOC

and

Store LOC , Ri

are the only instructions available to transfer data between memory and the general purpose registers. Add instructions of type ADD LOCA, R0 and Add R1, R0 are available. Do not change contents of either location A or B.

Solution:

(a)

Load R0 , A

Load R1 , B

Add R1, R0

Store C , R1

Answer

(b) Suppose that Move and Add instructions are available with the formats:

Move Location1, Location2

and

Add Location1, Location2

These instructions move or add a copy of the operand at the second location to the first location, overwriting the original operand at the first location. Either or both of the operands can be in the memory or the general-purpose registers.

Is it possible to use fewer instructions of these types to accomplish the task in part (a)? If yes, give the sequence.

Answer

(b) Suppose that Move and Add instructions are available with the formats:

Move Location1, Location2

and

Add Location1, Location2

These instructions move or add a copy of the operand at the second location to the first location, overwriting the original operand at the first location. Either or both of the operands can be in the memory or the general-purpose registers.

Is it possible to use fewer instructions of these types to accomplish the task in part (a)? If yes, give the sequence.

Solution:

(b) Yes;

Move C, B

Add C, A

Test Your Knowledge

- _____ contains the memory address of the next instruction to be fetched and executed.

a. Memory Address Register

b. Memory Data Register

c. Instruction Register

d. Program Counter

Test Your Knowledge

- _____ contains the memory address of the next instruction to be fetched and executed.

a. Memory Address Register

b. Memory Data Register

c. Instruction Register

d. Program Counter

Test Your Knowledge

- _____ holds the instruction that is currently being executed.

a. Memory Address Register
b. Memory Data Register
c. Instruction Register
d. Program Counter

Test Your Knowledge

- _____ holds the instruction that is currently being executed.

a. Memory Address Register
b. Memory Data Register
c. Instruction Register
d. Program Counter

Next we will learn

Unit 1: Number Representation and Arithmetic Operations

Three major representations of Signed Integer

1. Sign and Magnitude
2. One's complement
3. Two's complement

B	Values represented		
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	- 0	- 7	- 8
1 0 0 1	- 1	- 6	- 7
1 0 1 0	- 2	- 5	- 6
1 0 1 1	- 3	- 4	- 5
1 1 0 0	- 4	- 3	- 4
1 1 0 1	- 5	- 2	- 3
1 1 1 0	- 6	- 1	- 2
1 1 1 1	- 7	- 0	- 1

Decimal Number Representation or Base 10

In Decimal or Base 10 System, digits used are:
0 1 2 3 4 5 6 7 8 9

Representing 537 (Five hindered and thirty Seven)

10000	1000	100	10	1
0	0	5	3	7

(5 * 100) + (3 * 10) + (7 * 1)

Binary Number Representation or Base 2

- Binary Digit or **Bit** is the smallest unit of computation on most digital computers
- Bit has two states
  - 0 represents zero voltage (0v) or ground
  - 1 represents positive voltage (+5v)

Binary Number Representation or Base 2

Power of 2	Calculation	Value
2^0	1	1
2^1	2	2

Binary Number Representation or Base 2

Power of 2	Calculation	Value
2^0	1	1
2^1	2	2

Decimal	Binary	
0	00	
1	01	
2	10	
3	11	

Binary Number Representation or Base 2

Power of 2	Calculation	Value
2^0	1	1
2^1	2	2

Decimal	Binary	
0	00	$2^1 * 0 + 2^0 * 0 = 2*0 + 1*0 = 0+0=0$
1	01	
2	10	
3	11	

Binary Number Representation or Base 2

Power of 2	Calculation	Value
2^0	1	1
2^1	2	2

Decimal	Binary	
0	00	$2^1 * 0 + 2^0 * 0 = 2*0 +1*0 = 0+0=0$
1	01	$2^1 * 0 + 2^0 * 1 = 2*0 +1*1 =0+1= 1$
2	10	$2^1 * 1 + 2^0 * 0 = 2*1 +1*0 = 2+0=2$
3	11	$2^1 * 1 + 2^0 * 1 = 2*1 +1*1 = 2+1=3$

Example: 3-bit binary numbers

Decimal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Power of 2	Calculation	Value
2^0		1
2^1	2	2
2^2	2 * 2	4

Question: List out all 4-bit binary numbers

Decimal	Binary
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

Power of 2	Calculation	Value
2^0		1
2^1	2	2
2^2	2 * 2	4
2^3	2 * 2 * 2	8

Example: 4-bit binary numbers

Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Power of 2	Calculation	Value
2^0		1
2^1	2	2
2^2	2 * 2	4
2^3	2 * 2 * 2	8

Representation of a Binary Number

- Converting from decimal to binary (base 10 to base 2) will also produce a weighted binary number with the right-hand most bit being the **Least Significant Bit** or **LSB**, and the left-hand most bit being the **Most Significant Bit**or **MSB**, and we can represent this as:

MSB	Binary Digit							LSB
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1

- Convert binary to decimal by finding the decimal equivalent of the binary array of digits 101100101_2 and expanding the binary digits into a series with a base of 2giving an equivalent of 357_{10} in decimal or denary.

Decimal Digit Value	256	128	64	32	16	8	4	2	1
Binary Digit Value	1	0	1	1	0	0	1	0	1

$(256) + (64) + (32) + (4) + (1) = 357_{10}$

Question: Convert the following Binary number to Decimal

- Binary number: 100011

MSB	Binary Digit							LSB
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1

Answer

- Binary number: 100011
- Equivalent Decimal number is: 35

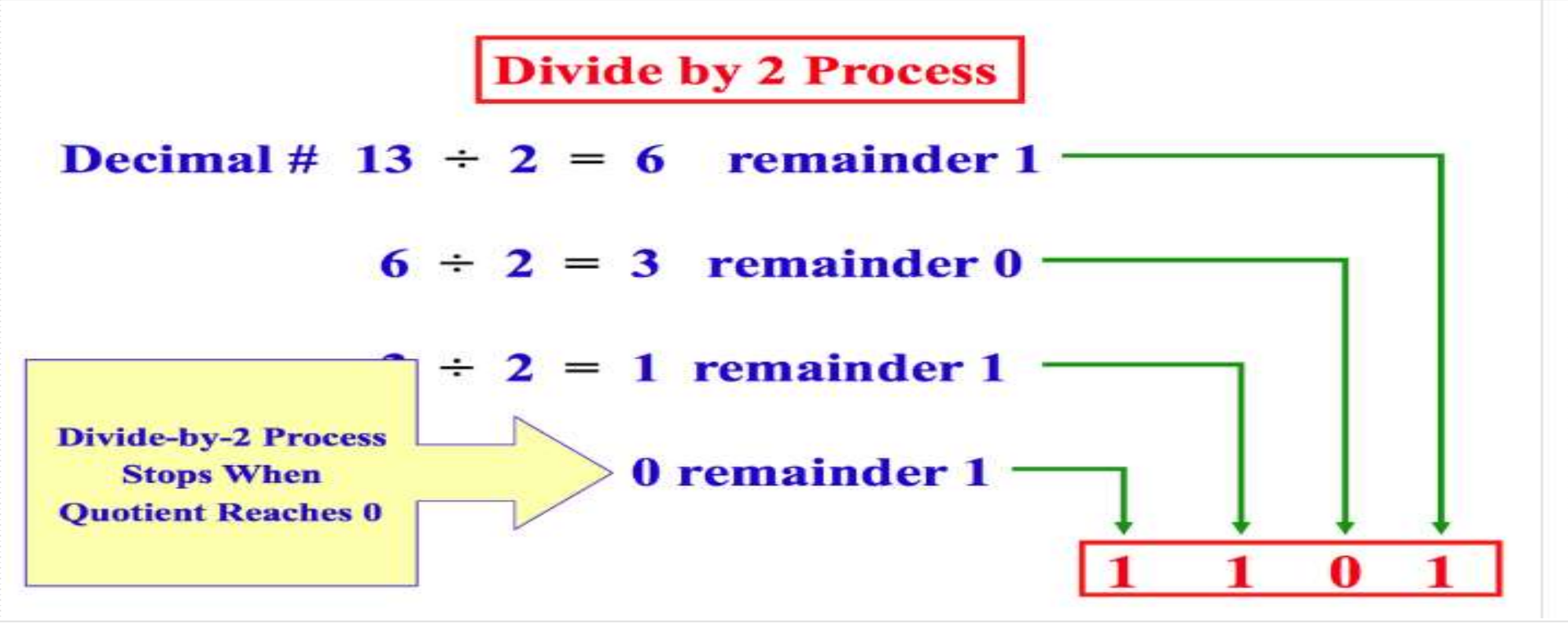
Base^{Exponent}	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Place Value	128	64	32	16	8	4	2	1
Example: Convert decimal 35 to binary	0	0	1	0	0	0	1	1

$35 = 2^5 + 2^1 + 2^0$

$35 = (32 * 1) + (2 * 1) + (1 * 1)$

MSB	Binary Digit							LSB
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1

Decimal to Binary Conversion



Question: Represent the following Decimal number in 7-bit binary numbers

Decimal	7-bit Binary Number
5	
14	
26	
53	

MSB	Binary Digit							LSB
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1

Answer

Decimal	7-bit Binary Number
5	0000101
14	0001110
26	0011010
53	0110101

MSB	Binary Digit							LSB
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1

Question

- What is the biggest decimal number we can represent in binary using 8-bit ?

- How many different decimal numbers that we can represent in binary using 8-bit ?

Answer

What is the biggest decimal number that you can represent in binary using 8-bit ?

- 255 (i.e., 2^8 -1=256-1=255)

How many different decimal numbers that you can represent in binary using 8-bit ?

- 0 to 255 i.e., 256 decimal numbers

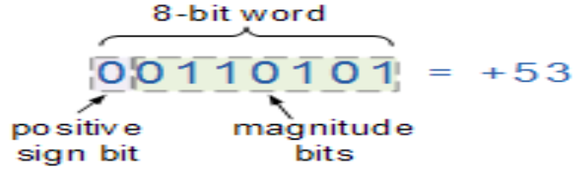
Signed Binary Number Representation

- We can use a single bit to identify the sign of a *signed binary number* as being positive or negative in value. So to represent a positive binary number (+n) and a negative (-n) binary number, we can use them with the addition of a sign.
- For signed binary numbers the most significant bit (MSB) is used as the sign bit. If the sign bit is "0", this means the number is positive in value. If the sign bit is "1", then the number is negative in value. The remaining bits in the number are used to represent the magnitude of the binary number in the usual unsigned binary number format way.
- Then we can see that the Sign-and-Magnitude (SM) notation stores positive and negative values by dividing the "n" total bits into two parts: 1 bit for the sign and n–1 bits for the value which is a pure binary number. For example, the decimal number 53 can be expressed as an 8-bit signed binary number as follows:

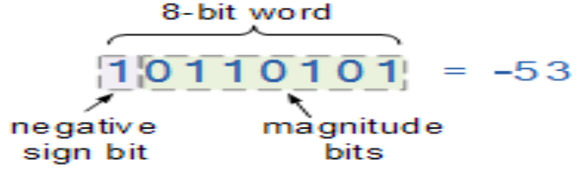
Signed Binary Number Representation

- We can use a single bit to identify the sign of a *signed binary number* as being positive or negative in value. So to represent a positive binary number (+n) and a negative (-n) binary number, we can use them with the addition of a sign.
- For signed binary numbers the most significant bit (MSB) is used as the sign bit. If the sign bit is "0", this means the number is positive in value. If the sign bit is "1", then the number is negative in value. The remaining bits in the number are used to represent the magnitude of the binary number in the usual unsigned binary number format way.
- Then we can see that the Sign-and-Magnitude (SM) notation stores positive and negative values by dividing the "n" total bits into two parts: 1 bit for the sign and n–1 bits for the value which is a pure binary number. For example, the decimal number 53 can be expressed as an 8-bit signed binary number as follows:

Positive Signed Binary Numbers



Negative Signed Binary Numbers

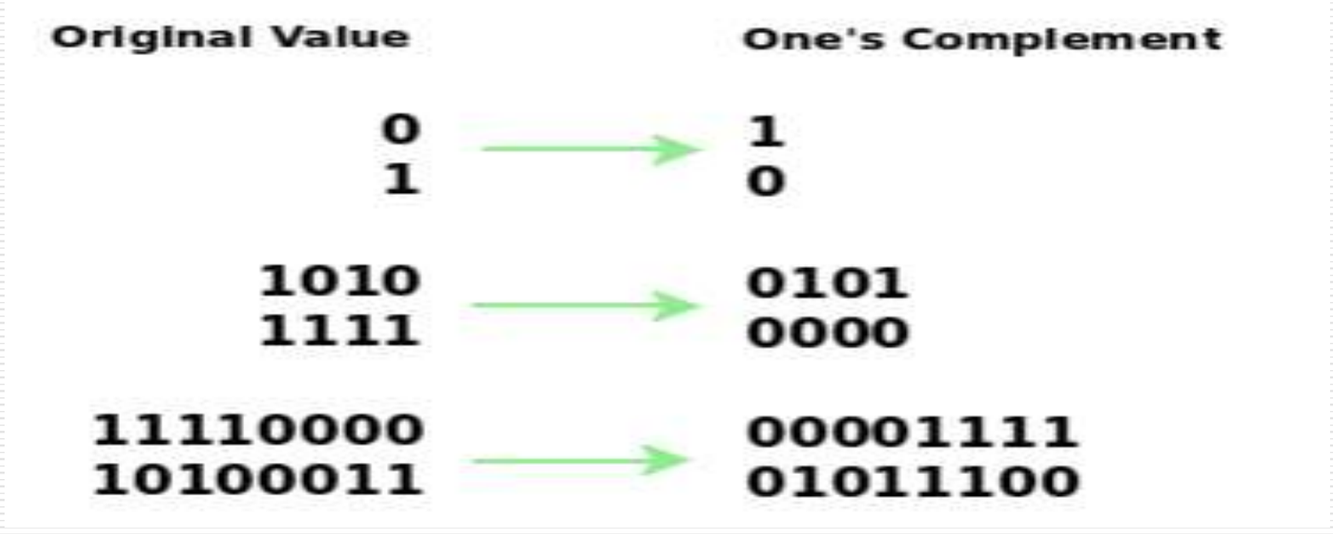


Example: Signed numbers

$b_3\,b_2\,b_1\,b_0$	Sign and magnitude
0 1 1 1	+ 7
0 1 1 0	+ 6
0 1 0 1	+ 5
0 1 0 0	+ 4
0 0 1 1	+ 3
0 0 1 0	+ 2
0 0 0 1	+ 1
0 0 0 0	+ 0
1 0 0 0	- 0
1 0 0 1	- 1
1 0 1 0	- 2
1 0 1 1	- 3
1 1 0 0	- 4
1 1 0 1	- 5
1 1 1 0	- 6
1 1 1 1	- 7

1’s (One’s) complement number representation

- If all bits in a byte are inverted by changing each 1 to 0 and each 0 to 1, we have formed the one’s complement of the number.



Binary Sign-Magnitude and One's Complement representation

B	Values represented	
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement
0 1 1 1	+ 7	+ 7
0 1 1 0	+ 6	+ 6
0 1 0 1	+ 5	+ 5
0 1 0 0	+ 4	+ 4
0 0 1 1	+ 3	+ 3
0 0 1 0	+ 2	+ 2
0 0 0 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0
1 0 0 0	- 0	- 7
1 0 0 1	- 1	- 6
1 0 1 0	- 2	- 5
1 0 1 1	- 3	- 4
1 1 0 0	- 4	- 3
1 1 0 1	- 5	- 2
1 1 1 0	- 6	- 1
1 1 1 1	- 7	- 0

In One's Complement representation, negative values are obtained by complementing each bit of the corresponding positive number.

Two’s Complement of Number representation

The two’s complement is a method for representing positive and negative integer values in binary. The useful part of two’s complement is that it automatically includes the sign bit.

Rule: To form the two’s complement, add 1 to the one’s complement.

Step 1: Begin with the original binary value

10011001 Original binary number

Step 2: Find the one's complement

01100110 One's complement

Step 3: Add 1 to the one's complement

01100110 One's complement

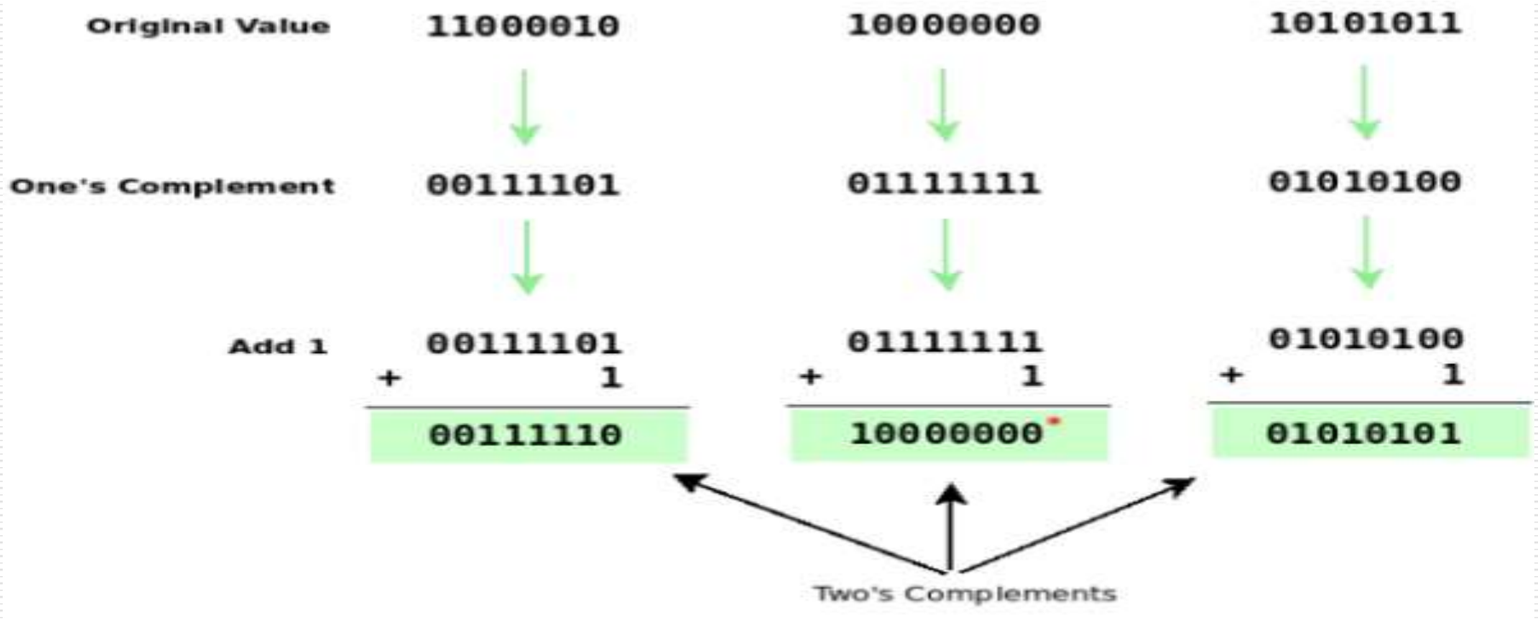
+   1 Add 1

01100111 <--- Two's complement

Two’s Complement

The two’s complement is a method for representing positive and negative integer values in binary. The useful part of two’s complement is that it automatically includes the sign bit.

Rule: To form the two’s complement, add 1 to the one’s complement.



Binary Sign-Magnitude, One's Complement representation and Two's Complement

B	Values represented		
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	- 0	- 7	- 8
1 0 0 1	- 1	- 6	- 7
1 0 1 0	- 2	- 5	- 6
1 0 1 1	- 3	- 4	- 5
1 1 0 0	- 4	- 3	- 4
1 1 0 1	- 5	- 2	- 3
1 1 1 0	- 6	- 1	- 2
1 1 1 1	- 7	- 0	- 1

Conversion of Negative Numbers to Two's Complement

- These examples show conversion of a decimal number to **4-bit** twos complement.
- The **bit size** is always important with twos complement, since you must be able to tell where the sign bit is.
- The steps are simple.
  - First, you convert the magnitude of the number to binary, and pad to the word size (4 bits).
  - If the original number was positive, you are done.
  - Otherwise, you must negate the binary number by inverting the bits and adding 1.

Conversion of Negative Numbers to Two's Complement

- These examples show conversion of a decimal number to 4-bit twos complement.
- The bit size is always important with twos complement, since you must be able to tell where the sign bit is.
- The steps are simple.
  - First, you convert the magnitude of the number to binary, and pad to the word size (4 bits).
  - If the original number was positive, you are done.
  - Otherwise, you must negate the binary number by inverting the bits and adding 1.
- Convert **-6** to an **4-bit**, *twos complement binary number*.
  - Convert the magnitude, 6 to binary. So $6_{10} = 110_2$.
  - Pad to 4 bits: 0110
  - Negate the number by inverting the bits and adding 1.

```
          0110
Negate    1001
Add 1        1
          ----
          1010   Two's complement of -6
```

Conversion of Negative Numbers to Two's Complement

- These examples show conversion of a decimal number to **8-bit twos complement.**
- The bit size is always important with twos complement, since you must be able to tell where the sign bit is.
- The steps are simple.
  - First, you convert the magnitude of the number to binary, and pad to the word size (8 bits).
  - If the original number was positive, you are done.
  - Otherwise, you must negate the binary number by inverting the bits and adding 1.
- Convert **-72** to an **8-bit, twos complement** binary number.
  - Convert the magnitude, 72 to binary. So $72_{10} = 1001000_2$.
  - Pad to 8 bits: 01001000
  - Negate the number by inverting the bits and adding 1.

	0	1	0	0	1	0	0	0
−	1	0	1	1	0	1	1	1
+								1
	1	0	1	1	1	0	0	0

Question

- Using 7 bits to represent each number, write the representations of 23 and -23 in signed-magnitude , 1’s complement and 2's complement system

Answer

- Using 7 bits to represent each number, write the representations of 23 and -23 in signed magnitude and 2's complement integers

	Signed Magnitude	1's Complement	2's Complement
23	0010111	0010111	0010111
-23	1010111	1101000	1101001

Question

Represent the decimal values 5, -2, 14, -10, 26, -19, 51 and -43 as signed 7-bit numbers in the following formats

a. Sign-Magnitude
b. 1’s complement
c. 2’s complement

Answer

Represent the decimal values 5, -2, 14, -10, 26, -19, 51 and -43 as signed 7-bit numbers in the following formats

a. Sign-Magnitude
b. 1's complement
c. 2's complement

Decimal values	Sign-and-magnitude representation	1's-complement representation	2's-complement representation
5	0000101	0000101	0000101
−2	1000010	1111101	1111110
14	0001110	0001110	0001110
−10	1001010	1110101	1110110
26	0011010	0011010	0011010
−19	1010011	1101100	1101101
51	0110011	0110011	0110011
−43	1101011	1010100	1010101

Conversion of Negative Numbers to Two's Complement

- Convert 47 to an 8-bit, twos complement binary number. This is positive, so all that is needed is to convert to binary and pad to eight bits. So $47_{10} = 101111_2$. So 47 as an 8-bit two's complement number is just 00101111.
- Convert -109 to an 8-bit, twos complement number. So $109_{10} = 1101101_2$.

	0	1	1	0	1	1	0	1
¬	1	0	0	1	0	0	1	0
+								1
	1	0	0	1	0	0	1	1

- Convert -67 to an 8-bit, twos complement number. So $67_{10} = 1000011_2$.

	0	1	0	0	0	0	1	1
¬	1	0	1	1	1	1	0	0
+								1
	1	0	1	1	1	1	0	1

- Convert 81 to an 8-bit, twos complement number. Since this is positive, it's just a matter of converting to binary and padding to 8 bits. So $81_{10} = 1010001_2$, giving 01010001

Under Signed Number Representation

- Number of bits used for representation is important because

Under 8-bit representation	
00000101	+5
10000101	-5
Under 4-bit representation	
0101	+5
1101	-5

4-bit Signed Binary Number Representation

Decimal	Signed Magnitude	Signed One's Complement	Signed Two's Complement
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	1000	1111	–
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001

Problem with arithmetic

- Under Sign-magnitude representation
- Under One's Complement representation

Sign-Magnitude (4-bit Representation)			
Adding +5-5	0101	+5	
	1101	-5	
Total	**10010** (**1** Carryout)	INCORRECT, because the result should be zero i.e., 00000	

One's Complement (4-bit Representation)			
Adding +5-5	0101	+5	
	1010	-5 One's Complement of -5	
Total	1111	INCORRECT, because the result should be zero i.e., 00000 But it is Minus Zero	

Arithmetic under Two's Complement

Decimal	Signed Two's Complement
+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
+0	0000
-0	–
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001

Two's Complement (4-bit Representation)		
Adding +5-5	0101	+5
	1011	-5
Total	**10000** (1 Carryout)	CORRECT(Zero) Ignore the Carryout

Two's Complement (4-bit Representation)		
Adding +6-2	0110	+6
	1110	-2
Total	**10100** (1 Carryout)	CORRECT(+4) Ignore the Carryout

Addition and Subtraction of Signed Numbers using two's Complement

The 2's complement is the most efficient method for performing addition and subtraction operations

The rules governing addition and subtraction of n-bit signed numbers using 2's complement representation system may be stated as follows:

- To **add two numbers**, add their n-bit representation , ignoring the carry-out bit from the Most Significant Bit (MSB) position. The sum will be algebraically correct value in 2's complement representation if the actual result is in the range -2^{n-1} through $+2^{n-1} -1$
- To **subtract two numbers** X and Y, that is, to perform X-Y, form the 2's complement of Y, then add it to X using the add rule. Again, the result will be algebraically correct value in 2's complement representation if the actual result is in the range -2^{n-1} through $+2^{n-1} -1$

Addition and Subtraction of Signed Numbers

- Using 2's complement system and considering 5-bit representation
- Example

5+4	+5	00101	
	+4	00100	
		01001	+9

Subtraction by 2’s Complement:

The operation is carried out by means of the following steps:

(i) At first, 2’s complement of the subtrahend is found.

(ii) Then it is added to the minuend.

(iii) Throw away extra carries

(iv) If leading number (MSB) is 0, answer is positive. If leading number is 1, answer is negative.

Note:
Subtrahend: what is being subtracted
Minuhend: what it is being subtracted from

Addition and Subtraction of Signed Numbers

- Using 2's complement system and considering 5-bit representation
- Example

-5+4	-5	11011	
	+4	00100	
		11111	-1

2's complement of -5

00101	+5
11010	1's complement
1	Add 1
11011	-5

Addition and Subtraction of Signed Numbers

- Using 2's complement system and considering 5-bit representation
- Example

+5-4

5	00101	
-4	11100	
	100001	+1

Carry Out
Ignore

2's complement of -4

00100	+4
11011	1's complement
1	Add 1
11100	-4

Addition and Subtraction of Signed Numbers

- Using 2's complement system and considering 5-bit representation
- Example

-5-4

-5	11011	
-4	11100	
	110111	-9

Carry Out
Ignore

2's complement of -5

00101	+5
11010	1's complement
1	Add 1
11011	-5

2's complement of -4

00100	+4
11011	1's complement
1	Add 1
11100	-4

Test Your knowledge

Convert the following pairs of decimal numbers to 5-bit 2’s-complement numbers, then add them.

a. -5 and 7

b. -3 and -8

Answer

Convert the following pairs of decimal numbers to 5-bit 2's-complement numbers, then add them.

a. -5 and 7

```
  11011
+ 00111
-------
  00010
```

b. -3 and -8

```
  11101
+ 11000
-------
  10101
```

Example of two's Complement Arithmetic

P 1) **Perform arithmetic on the following numbers using their binary equivalents.**

i) **5 + 4**

ii) **–5 + 4**

iii) **5 – 4**

iv) **–5 – 4**

Sol. : Consider the following arithmetic using 2's complement procedure. The rules to perform arithmetic is as follows.

Step 1 : For any positive number, let it's representations be in a normal signed magnitude representation.

Step 2 : In case of negative numbers, initially consider their 2's complement representation.

Step 3 : Perform only addition (eventhough subtraction is prescribed to be performed).

Step 4 : If the answer is negative, then the answer is in 2's complement form.

Step 5 : Discard the digit, if there is a carry beyond the MSB (Most Significant Bit)

i) **5 + 4 :** As both the numbers are positive, hence represent them in normal sign magnitude representation and perform addition.

+5 ⟶ 0 0 0 0 0 1 0 1

+4 ⟶ 0 0 0 0 0 1 0 0

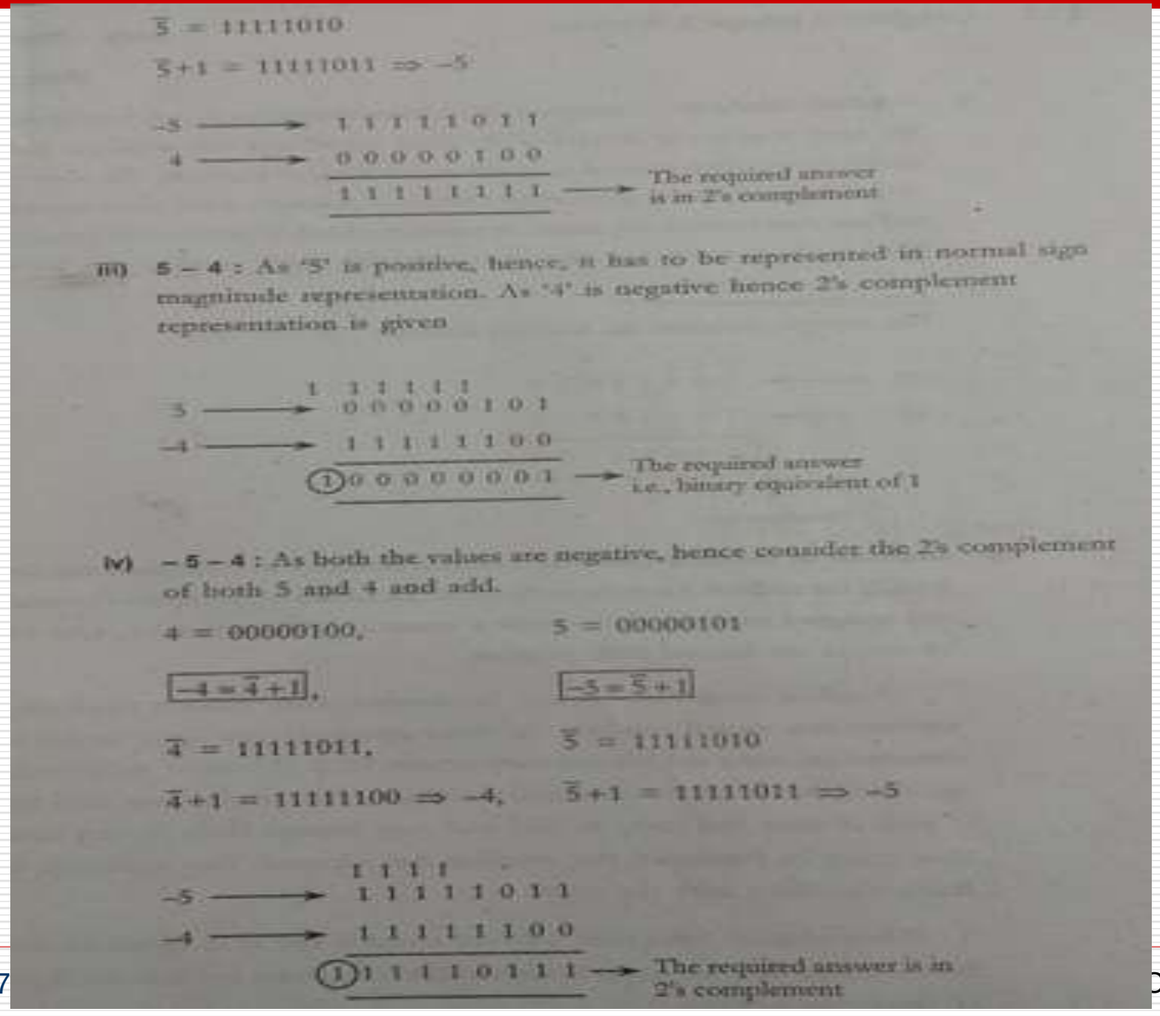
+9 ⟶ 0 0 0 0 1 0 0 1

ii) **– 5 + 4 :** As '5' is negative, hence represent –5 in 2's complement representation and 4 in normal sign magnitude representation and perform addition.

5 = 00000101

$\boxed{-5 = \overline{5} + 1}$

Example of two's Complement Arithmetic



$\overline{5}$ = 11111010

$\overline{5}+1$ = 11111011 ⇒ –5

–5 ⟶ 1 1 1 1 1 0 1 1

4 ⟶ 0 0 0 0 0 1 0 0

1 1 1 1 1 1 1 1 ⟶ The required answer is in 2's complement

iii) **5 – 4 :** As '5' is positive, hence, it has to be represented in normal sign magnitude representation. As '4' is negative hence 2's complement representation is given

1 1 1 1 1 1

5 ⟶ 0 0 0 0 0 1 0 1

–4 ⟶ 1 1 1 1 1 1 0 0

(1) 0 0 0 0 0 0 0 1 ⟶ The required answer i.e., binary equivalent of 1

iv) **– 5 – 4 :** As both the values are negative, hence consider the 2's complement of both 5 and 4 and add.

4 = 00000100, 5 = 00000101

$\boxed{-4=\overline{4}+1}$, $\boxed{-5=\overline{5}+1}$

$\overline{4}$ = 11111011, $\overline{5}$ = 11111010

$\overline{4}+1$ = 11111100 ⇒ –4, $\overline{5}+1$ = 11111011 ⇒ –5

1 1 1 1

–5 ⟶ 1 1 1 1 1 0 1 1

–4 ⟶ 1 1 1 1 1 1 0 0

(1) 1 1 1 1 0 1 1 1 ⟶ The required answer is in 2's complement

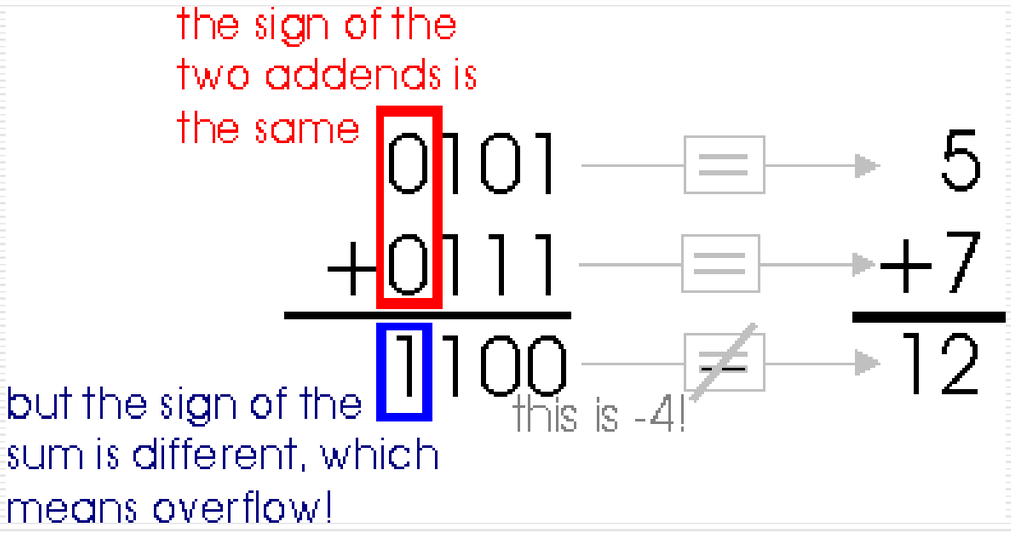
Arithmetic Overflow

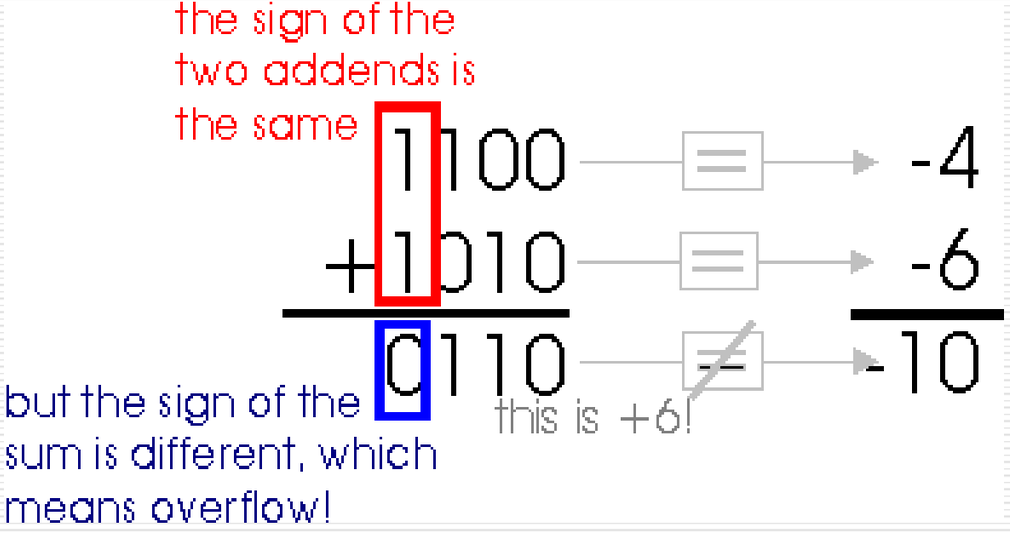
- In 2's complement number representation system, n-bits can represent values in the range (-2^{n-1}) to $(+2^{n-1} - 1)$.
- When the result of an arithmetic operation is outside the representable range, an arithmetic overflow has occurred.

Arithmetic Overflow

Care must be taken when adding numbers of like sign since *overflow* can occur.

- If you add two numbers of like sign and the result is of the opposite sign, then the result cannot be used. This "overflow" condition occurs because, in order to represent the result, we would need more bits than are available in the bit field. (Remember, we can't just "enlarge" the size of the result- it must remain the same size as the operands.) Here are examples for adding two negative numbers, and adding two positive numbers, each of which results in overflow.





Detecting overflow when adding two 2’s complement numbers

If 2 Two's Complement numbers are added, and they both have the same sign (both positive or both negative), then overflow occurs if and only if the result has the opposite sign. Overflow never occurs when adding operands with different signs. i.e. Adding two positive numbers must give a positive result Adding two negative numbers must give a negative result

Overflow occurs if

(+A) + (+B) = −C

(−A) + (−B) = +C

Example: Using 4-bit Two's Complement numbers (−8 ≤ x ≤ +7)

  (−7) 1001

+(−6) 1010

(−13) 1 0011 = 3 : Overflow (largest −ve number is −8)

Test Your Knowledge

Convert the following pairs of decimal numbers to 5-bit 2’s-complement numbers, then add them. State whether or not overflow occurs in each case

a. 7 and 13

b. -10 and -13

Answer

Convert the following pairs of decimal numbers to 5-bit 2's-complement numbers, then add them. State whether or not overflow occurs in each case

a. 7 and 13

```
  00111
+ 01101
-------
  10100
overflow
```

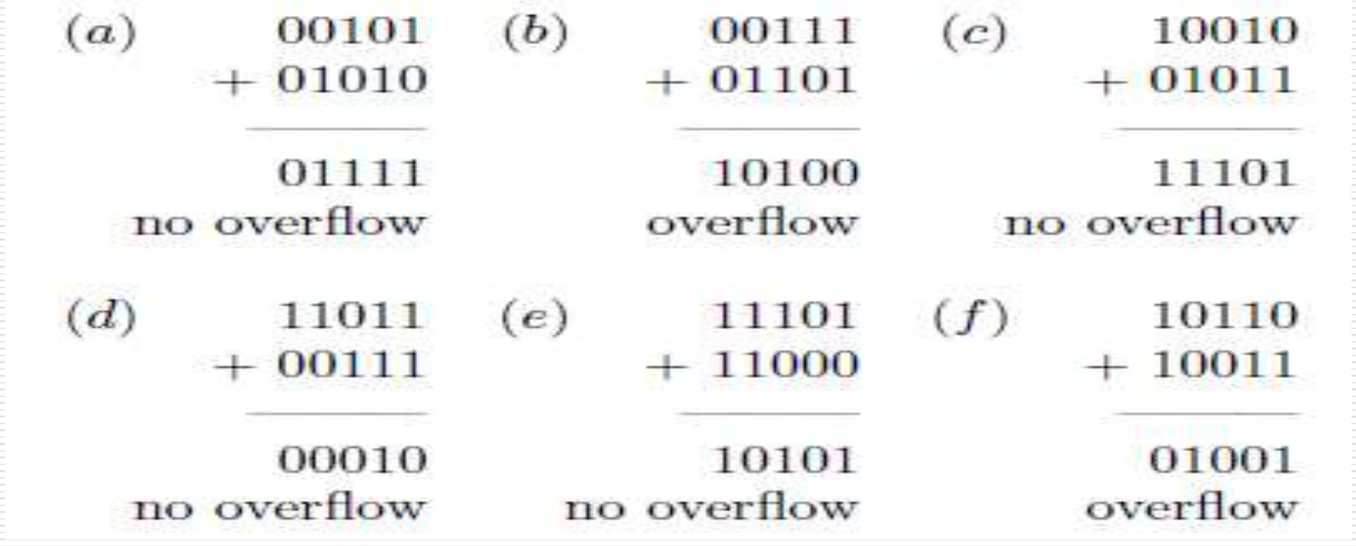
b. -10 and -13

```
  10110
+ 10011
-------
  01001
overflow
```

Question

Convert the following pairs of decimal numbers to 5-bit 2's-complement numbers, then add them. State whether or not overflow occurs in each case

a. 5 and 10
b. 7 and 13
c. -14 and 11
d. -5 and 7
e. -3 and -8
f. -10 and -13



Detecting overflow when subtracting two 2’s complement numbers

If two 2's Complement numbers are subtracted, and their signs are different, then overflow occurs if and only if the result has the same sign as the subtrahend.

Overflow occurs if

(+A) − (−B) = −C

(−A) − (+B) = +C

Example: Using 4-bit Two's Complement numbers (−8 ≤ x ≤ +7)

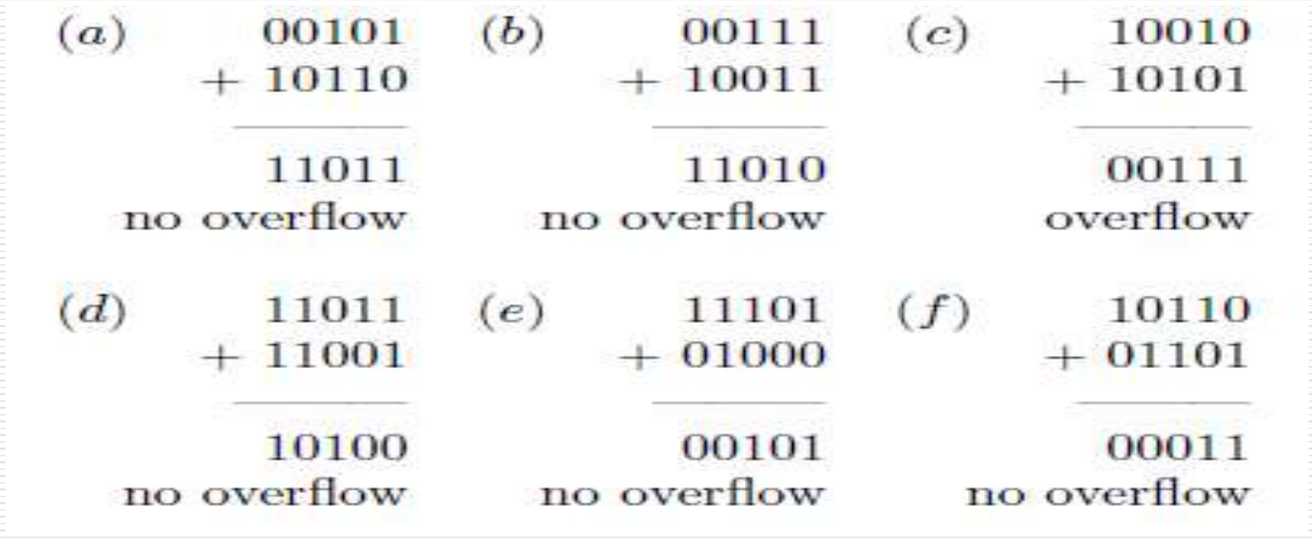
Subtract −6 from +7

```
  (+7)       0111                      0111
−(−6)        1010   -> Negate ->      +0110
           ----------                  -----
   13                                 1101  (−3) : Overflow
```

Question

Repeat for subtraction operation, where the second number of each pair to be subtracted from first number. State whether or not overflow occurs in each case

a. 5 and 10
b. 7 and 13
c. -14 and 11
d. -5 and 7
e. -3 and -8
f. -10 and -13



How to avoid Overflow

- **You can enlarge the size of the bit field, but only before you perform any operations, and it must be done a certain way.** If you find that the size of the bit field is too small and overflow is occurring, you can *promote* the values to larger bit fields. This is done by a technique called *sign extension*. To enlarge the bit field, add bits on the left, duplicating the most significant bit. This preserves the sign of the number and does not alter its value. Remember, you must promote all values to the same size. The table below illustrates sign-extension of a 4-bit number to 5, 6, and 8-bit fields. In each case, the most significant bit of the original 4-bit field (in blue) is simply repeated as many times as necessary on the left(in red).

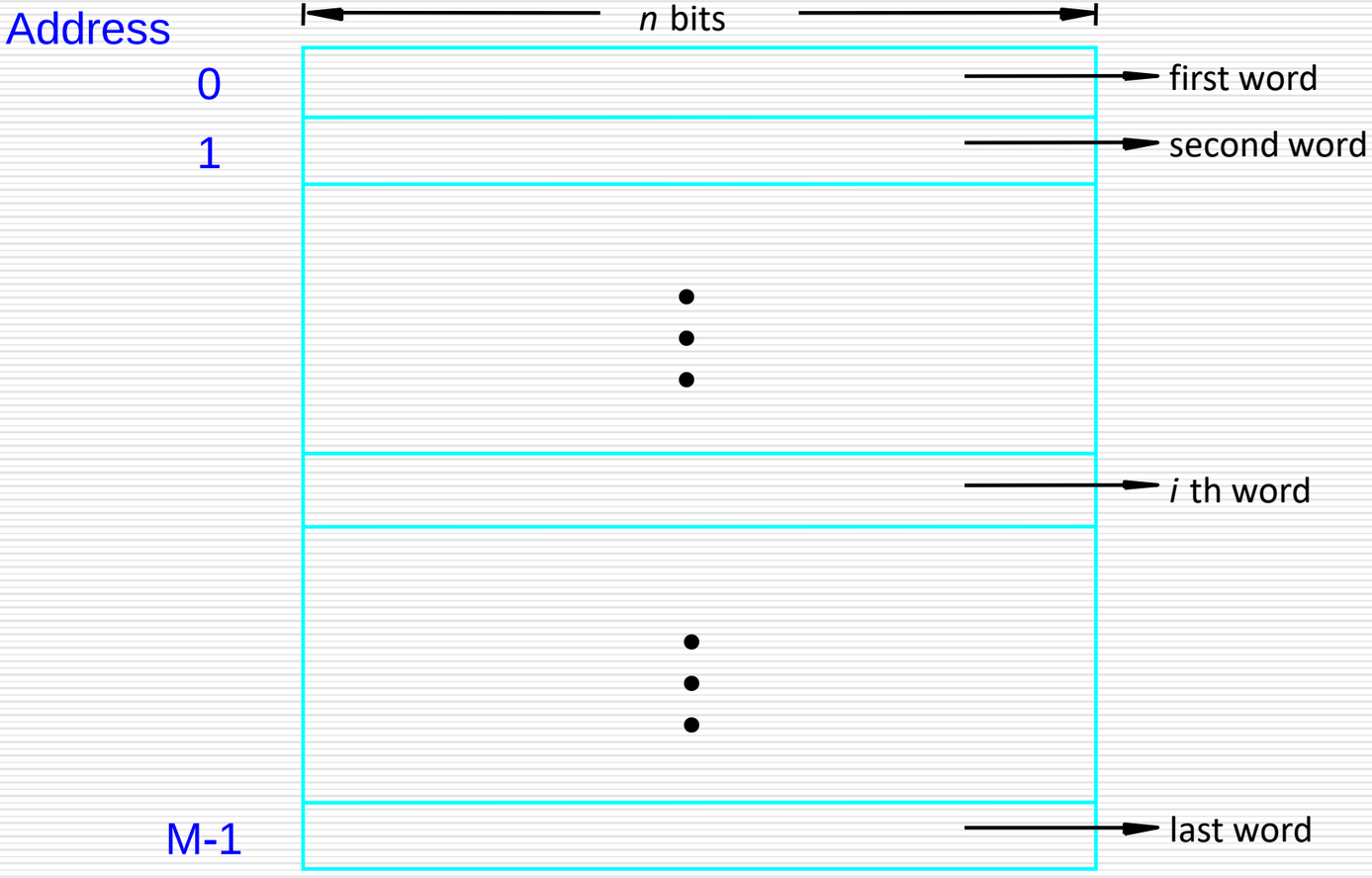
Original 4-bit Value	Extended to 5-bits	Extended to 6-bits	Extended to 8-bits
0010	00010	000010	00000010
1000	11000	111000	11111000

Unit1:Memory Locations and Addresses

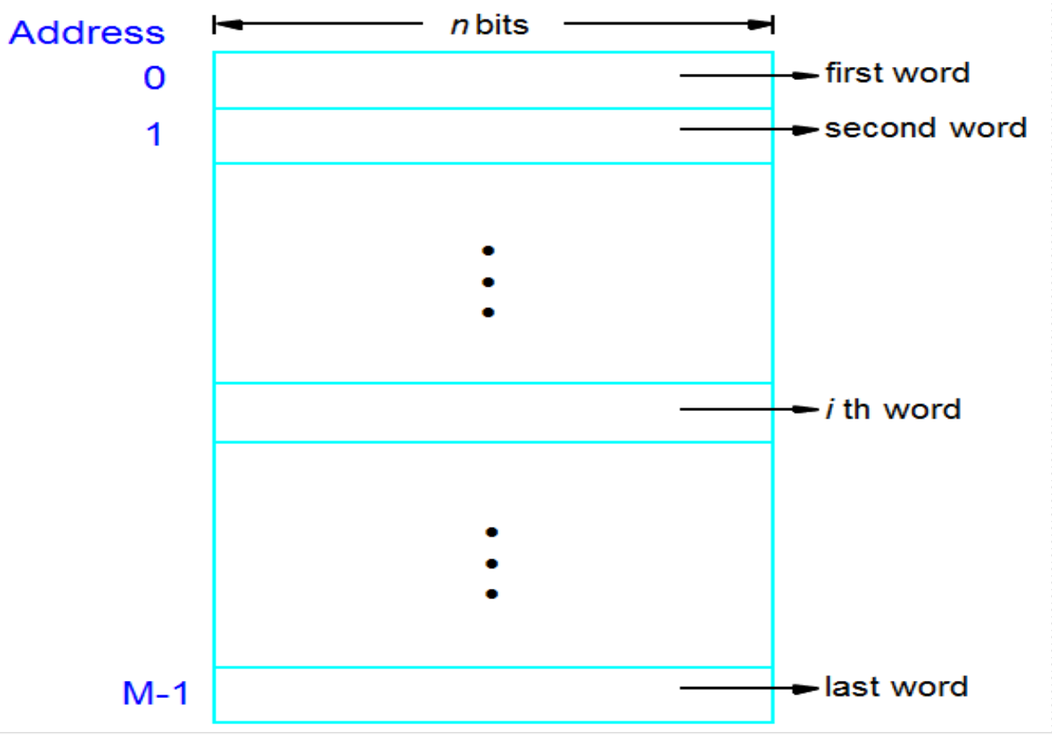
Memory Locations and Addresses

- **Memory** consists of many millions of storage cells (flip-flops).
- Each cell can store a bit of information i.e. 0 or 1
- Each group of n bits is referred to as a **word** of information, and n is called the **word length**.
- The word length can vary from 8 to 64 bits.
- A unit of 8 bits is called a **byte**.
- Accessing the memory to store or retrieve a single item of information (word/byte) requires distinct addresses for each item location. (It is customary to use numbers from 0 through 2^k-1 as the addresses of successive-locations in the memory).
- If 2^k = no. of addressable locations;
  then 2^k addresses constitute the address-space of the computer.
  For example, a 24-bit address generates an address-space of 2^{24} locations (16 MB).

Memory Locations and Addresses



Memory Locations and Addresses



Address Length **K-bits**	Addressable Locations **2^k**
2	2^2=4 Locations
3	2^3=8 Locations
4	2^4=16 Locations

Memory

Address K=2bits		Word Length n =8bits = 1 Byte	
0	00	0000 0110	1st Byte or word
1	01	0000 0111	2nd Byte or word
2	10	0000 1000	3rd Byte or word
3	11	0000 1010	4th Byte or word

Memory Locations and Addresses

Address Length **K-bits**	Addressable Locations **2^k**
2	2^2=4 Locations
3	2^3=8 Locations
4	2^4=16 Locations

Address K=3bits		Memory: Word Length n =8bits = 1 Byte	
0	000	0000 0110	1st Byte or word
1	001	0000 0111	2nd Byte or word
2	010	0000 1000	3rd Byte or word
3	011	0000 1010	4th Byte or word
4	100	0000 1011	5th Byte or word
5	101	0000 1100	6th Byte or word
6	110	0000 1101	7th Byte or word
7	111	0000 1110	8th Byte or word

Examples: $k = 20 \rightarrow 2^{20}$ or 1M locations,
$k = 32 \rightarrow 2^{32}$ or 4G locations

Question

Address Length **K-bits**	Addressable Locations **2^k**
2	2^2=4 Locations
3	2^3=8 Locations
4	2^4=16 Locations

Question:

Consider in one memory location,
One byte of information can be stored
i.e., n=8bits. To store 1024 bytes of information,
How many address bits should be used
i.e., what should be the k value ?

Memory

Address K=3bits		Word Length n =8bits = 1 Byte	
0	000	0000 0110	1st Byte or word
1	001	0000 0111	2nd Byte or word
2	010	0000 1000	3rd Byte or word
3	011	0000 1010	4th Byte or word
4	100	0000 1011	5th Byte or word
5	101	0000 1100	6th Byte or word
6	110	0000 1101	7th Byte or word
7	111	0000 1110	8th Byte or word

Question

Address Length **K-bits**	Addressable Locations **2^k**
2	2^2=4 Locations
3	2^3=8 Locations
4	2^4=16 Locations

Address K=3bits		Word Length n =8bits = 1 Byte	
0	000	0000 0110	1st Byte or word
1	001	0000 0111	2nd Byte or word
2	010	0000 1000	3rd Byte or word
3	011	0000 1010	4th Byte or word
4	100	0000 1011	5th Byte or word
5	101	0000 1100	6th Byte or word
6	110	0000 1101	7th Byte or word
7	111	0000 1110	8th Byte or word

Question:
Consider in one memory location,
One byte of information can be stored
i.e., n=8bits. To store 1024 bytes of information
How many address bits should be used
i.e., what should be the k value ?

Answer

Kilo Bytes (10^3) or 1024 Bytes	2^{10} =1024 Therefore number of address bits should be 10 bits

Byte Addressability

- Byte size is always 8 bits
- But word length may range from 16 to 64 bits
- Impractical to assign an address to each bit
- Instead, provide a byte-addressable memory that assigns an address to each byte
- Byte locations have addresses 0, 1, 2, …
- Assuming that the word length is 32 bits, word locations have addresses 0, 4, 8, …

Two ways of Byte address assignment across words

- Big-endian and little-endian are terms that describe the order in which a sequence of bytes are stored in computer memory.

1. Big-endian addressing assigns Lower byte addresses to Most Significant (leftmost) bytes of word

2. Little-endian addressing assigns Lower byte addresses to Least Significant (rightmost) bytes of word

Two ways of Byte address assignment across words

- Example: Consider storing the number 2064 i.e., Two thousand Sixty four. We will assume one digit occupies 4bits.

2064	2 0 6 4	
	0010 0000	0110 0100
	MSB Most Significant Byte	**LSB** Least Significant Byte

Two ways of Byte address assignment across words

- Example: Consider storing the number 2064 i.e., Two thousand Sixty four. We will assume one digit occupies 4bits.

2064	2 0 6 4	
	0010 0000	0110 0100
	MSB Most Significant Byte	**LSB** Least Significant Byte

Big-Endian Approach

Address	Word Length n=8bits=1 Byte		
00	0010 0000	20	MSB
01	0110 0100	64	LSB

Big-endian addressing assigns Lower byte addresses to Most Significant (leftmost) bytes of word

Little-Endian Approach

Address	Word Length n=8bits=1 Byte		
00	0110 0100	64	LSB
01	0010 0000	20	MSB

Little-endian addressing assigns Lower byte addresses to Least Significant (rightmost) bytes of word

Big-Endian and Little-Endian Assignments

- Big-Endian: lower byte addresses are used for the most significant bytes of the word
- Little-Endian: opposite ordering. lower byte addresses are used for the less significant bytes of the word

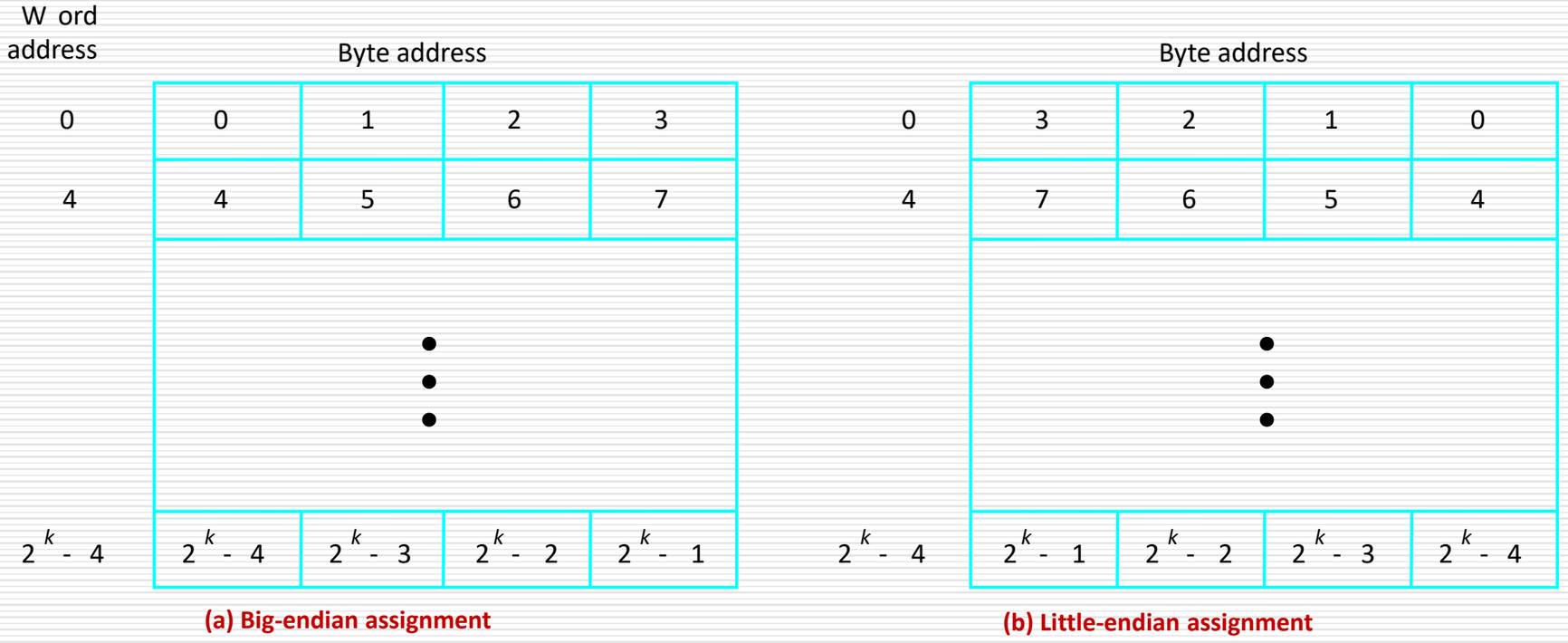


Figure :. Byte and word addressing.

Hexadecimal numbers

- A group of 4 bits can take any value between 0 (0000 binary) and 15 (1111 binary).
- In hexadecimal, we replace each group of 4 bits with a single digit to represent the value 0 to 15. Since we only have digits 0 to 9, we use letters A to E to represent values 10 to 15. Here is a table of binary, denary and hex values:

Denary	Binary	Hex
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
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Note: To specify Hexadecimal numbers Prefix 0x will be used
i.e.,
0x123
or
123**h**

Question

- Consider a computer has a byte-addressable memory organized in 32-bit words according to the big-endian scheme. A program reads ASCII characters entered at a keyboard and stores them in successive byte locations, starting at location 1000. Show the contents of the two memory words at locations 1000 and 1004 after the word “computer” has been entered. Values corresponding to the characters are as shown below:

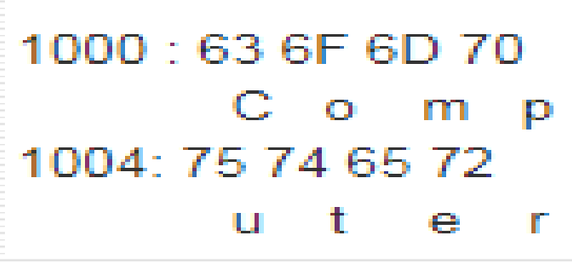
	Hex
c	0x63
o	0x6F
m	0x6D
p	0x70
u	0x75
t	0x74
e	0x65
r	0x72

Answer

- Consider a computer has a byte-addressable memory organized in 32-bit words according to the big-endian scheme. A program reads ASCII characters entered at a keyboard and stores them in successive byte locations, starting at location 1000. Show the contents of the two memory words at locations 1000 and 1004 after the word “computer” has been entered. Values corresponding to the characters are as shown below:

	Hex
c	63
o	6F
m	6D
p	70
u	75
t	74
e	65
r	72

Address	Word Length n=8bits=1 Byte		
1000	0110 0011	c 63	MSB
1001	0110 1111	o 6F	
1002	0110 1101	m 6D	
1003	0111 0000	p 70	
1004	0111 0101	u 75	
1005	0111 0100	t 74	
1006	0110 0101	e 65	
1007	0111 0010	r 72	LSB



Big-Endian Approach

Question

- Consider a computer has a byte-addressable memory organized in 32-bit words according to the **Little-endian** scheme. A program reads ASCII characters entered at a keyboard and stores them in successive byte locations, starting at location 1000. Show the contents of the two memory words at locations 1000 and 1004 after the word “computer” has been entered. Values corresponding to the characters are as shown below:

	Hex
c	63
o	6F
m	6D
p	70
u	75
t	74
e	65
r	72

Answer

- Consider a computer has a byte-addressable memory organized in 32-bit words according to the **Little-endian** scheme. A program reads ASCII characters entered at a keyboard and stores them in successive byte locations, starting at location 1000. Show the contents of the two memory words at locations 1000 and 1004 after the word "computer" has been entered. Values corresponding to the characters are as shown below:

	Hex
c	63
o	6F
m	6D
p	70
u	75
t	74
e	65
r	72

Address	Word Length n=8bits=1 Byte		
1000	0110 0011	r 72	LSB
1001	0110 1111	e 65	
1002	0110 1101	t 74	
1003	0111 0000	u 75	
1004	0111 0101	p 70	
1005	0111 0100	m 6D	
1006	0110 0101	o 6F	
1007	0111 0010	c 63	MSB

1000: 72 65 74 75
r e t u
1004: 70 6D 6F 63
p m o c

Little-Endian Approach

Memory Word Alignment

- Words are said to be Aligned in memory if they begin at a byte-address that is a multiple of number of bytes in a word.
- For example,
  - If the word length is 16 (2 Bytes), aligned words begin at byte addresses 0, 2, 4,....
  - If the word length is 32 (4 Bytes), aligned words begin at byte addresses 0, 4, 8,....
- Words are said to have Unaligned Addresses, if they begin at an arbitrary byte-address

Memory Operations

- Two memory operations are:
  1) Load (Read/Fetch) &
  2) Store (Write).
- The **Load** operation transfers a copy of the contents of a specific memory-location to the processor. The memory contents remain unchanged.
- Steps for Load operation:
  1) Processor sends the address of the desired location to the memory.
  2) Processor issues 'read' signal to memory to fetch the data.
  3) Memory reads the data stored at that address.
  4) Memory sends the read data to the processor.
- The **Store** operation transfers the information from the register to the specified memory-location. This will destroy the original contents of that memory-location.
- Steps for Store operation are:
  1) Processor sends the address of the memory-location where it wants to store data.
  2) Processor issues 'write' signal to memory to store the data.
  3) Content of register(MDR) is written into the specified memory-location.

INSTRUCTIONS and INSTRUCTION SEQUENCING

Instructions

A computer must have instruction capable of performing the following operations. They are:

- Data transfer between memory and processor register (Ex.: MOVE, LOAD, STORE)
- Arithmetic and logical operations on data (Ex.: ADD, SUB)
- Program sequencing and control (Ex.: BRANCH, CALL, RET)
- I/O transfer (Ex. IN, OUT).

Register Transfer Notation

The possible locations that may be involved during data transfer are

1. Memory Location
2. Processor register
3. Registers in I/O sub-system.

- Use [...] to denote contents of a location
- Use $\leftarrow$ to denote transfer to a destination

Example: R2 $\leftarrow$ [LOC]
(transfer from LOC in memory to register R2)

Example: R4 $\leftarrow$ [R2] + [R3]
(add the contents of registers R2 and R3, place the sum in register R4)

Assembly Language Notation

- To represent machine instructions and programs, assembly language format is used

Example

```
Load    R2, LOC       ; R2 ← [LOC]

Add     R4, R2, R3    ;  R4 ← [R2] + [R3]
```

RISC and CISC Instruction Sets

- One of the most important characteristics that distinguish different computers is the nature of their instructions.

Two fundamental approaches in design of instruction sets for modern computers	
Reduced Instruction Set Computers (RISC)	Complex Instruction Set Computers (CISC)
Have one-word instructions and require arithmetic operands to be in registers.	Have multi-word instructions and allow operands directly from memory.

RISC Instruction Sets

- Focus on RISC first because it is simpler
- RISC instructions each occupy a single word
- A load/store architecture is used, meaning:
  - only Load and Store instructions are used to access memory operands
  - operands for arithmetic/logic instructions must be in registers, or one of them may be given explicitly in instruction word

RISC Instruction Sets

1. Load instruction format

Load Destination , Source **or**

Load *Processor_register*, M*emory_location*

2. Store instruction format

Store Source , Destination **or**

Store *Processor_register*, M*emory_location*

RISC Instruction Sets

Example, sequence of instructions to perform the task

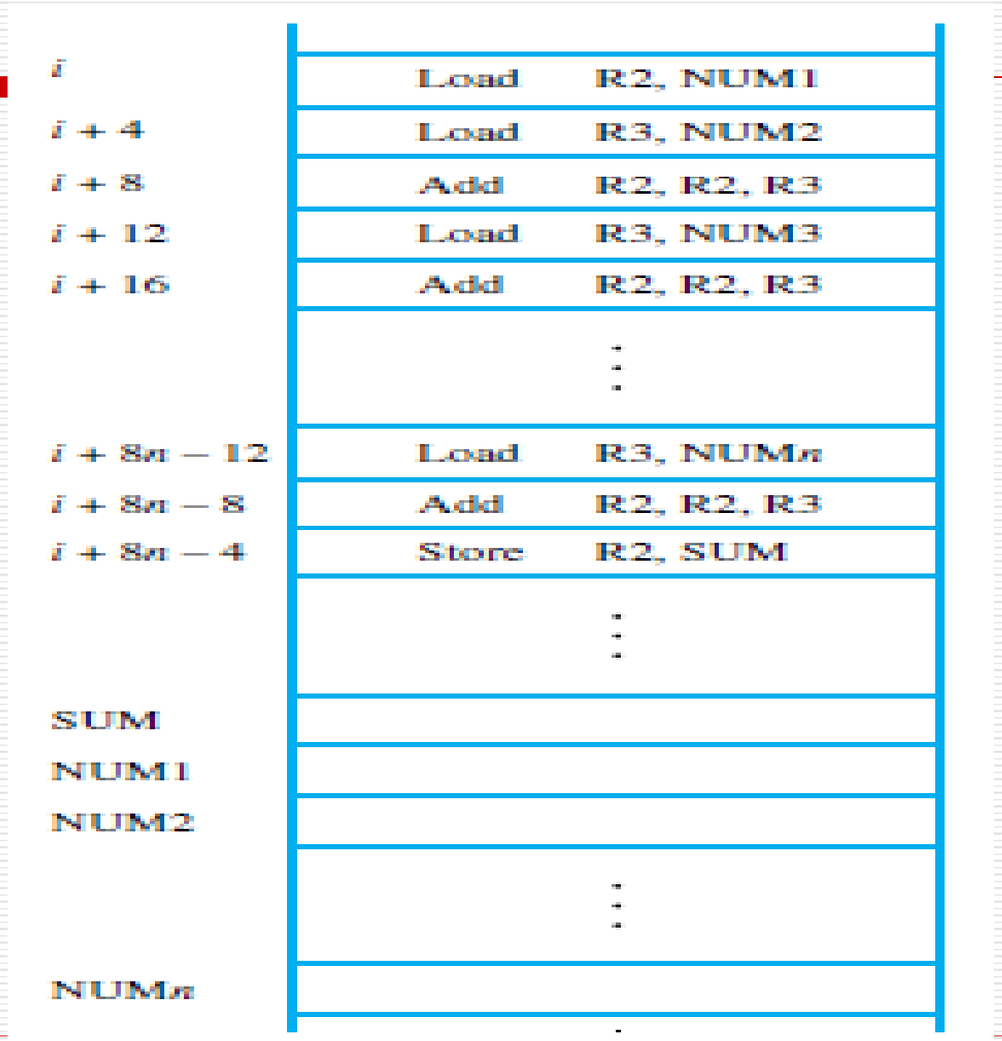
C = A + B ; C ← [A] + [B]

Sequence of simple RISC instructions for the task C = A + B :

```
Load   R2, A
Load   R3, B
Add    R4, R2, R3
Store  R4, C
```

Task of adding a list of n numbers Without Looping

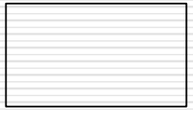
The program outlined in Figure is adding n numbers. The addresses of the memory locations containing the **n** numbers are symbolically given as **NUM1, NUM2, . . . , NUMn**, and separate *Load* and *Add* instructions are used to add each number to the contents of register **R2**. After all the numbers have been added, the result is placed in memory location **SUM**.



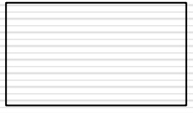
Rough Slide : To explain “Program to add n numbers” without Looping

	Memory 4 Bytes	Address
N	4	1000
Sum		1004
Num1	2	1008
Num2	1	1012
Num3	3	1016
Num4	10	1020

R2



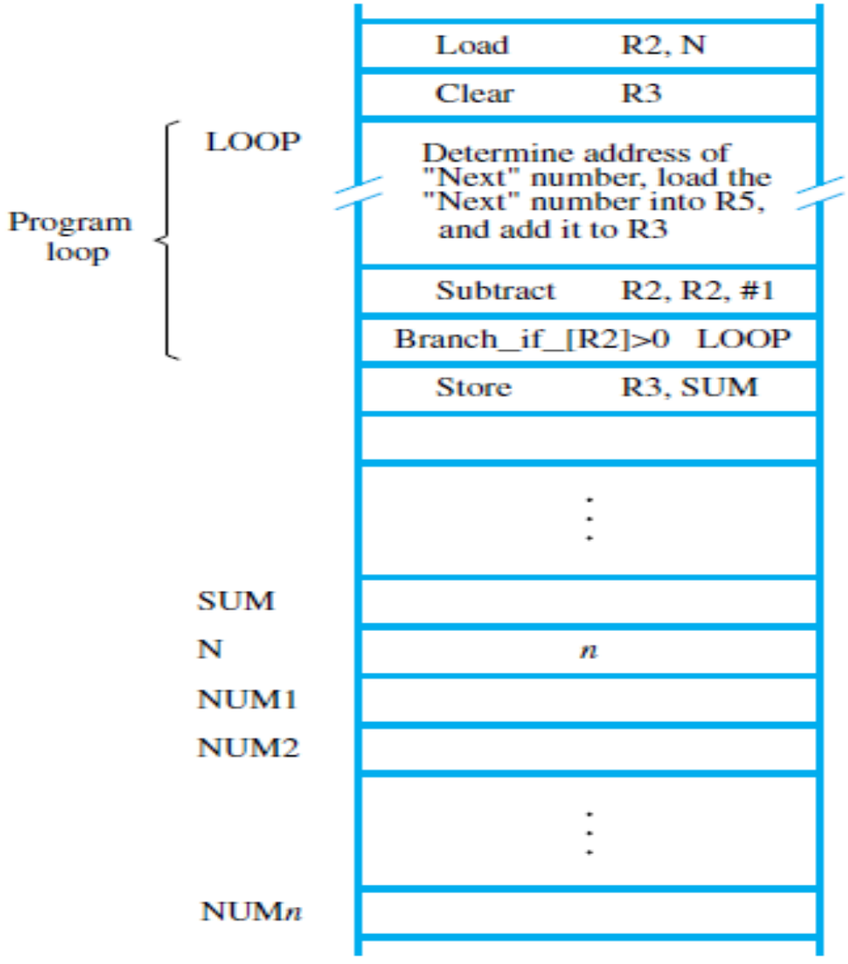
R3



	Address
Load R2, NUM1	3000
Load R3, Num2	3004
Add R2, R2, R3	3008
Load R3, Num3	3012
Add R2, R2, R3	3016
Load R3, Num4	3020
Add R2, R2, R3	3024
Store R2, SUM	3028

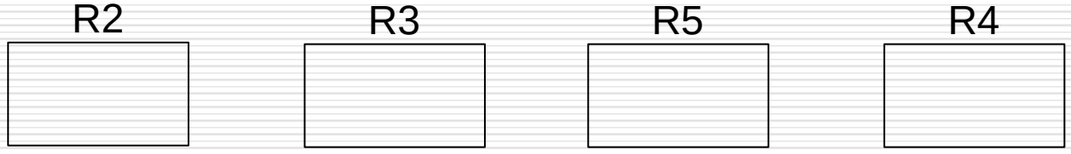
Task of adding a list of n numbers With Looping/Branching

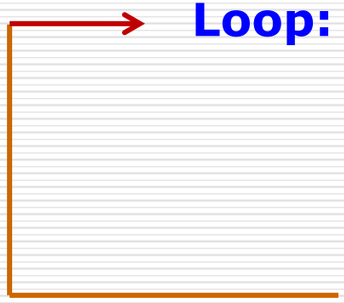
Instead of using a long list of Load and Add instructions,
it is possible to implement a program loop in which the instructions read the next number in the list and add it to the current sum. To add all numbers, the loop has to be executed as many times as there are numbers in the list. Figure shows the structure of the desired program. The body of the loop is a straight-line sequence of instructions executed repeatedly. It starts at location LOOP and ends at the instruction Branch_if_[R2]>0. During each pass through this loop, the address of the next list entry is determined, and that entry is loaded into R5 and added to R3.



Rough Slides : To explain “Program to add n numbers” with Branching

	Memory 4 Bytes	Address
N	4	1000
Sum		1004
Num1	2	1008
Num2	1	1012
Num3	3	1016
Num4	10	1020





	Address
Loop: Load R2, N	3000
Clear R3	3004
Move R4, #Num1	3008
Load R5, (R4)	3012
Add R3, R3, R5	3016
Add R4, R4, #4	3020
Subtract R2, R2, #1	3024
Branch_if_[R2] > 0 LOOP	3028
Store R3, SUM	3032

Addressing Modes

Addressing Modes

The term addressing modes refers to the way in which the operand of an instruction is specified. Information contained in the instruction code is the value of the operand or the address of the operand. Following are the main addressing modes that are used on various platforms and architectures.

1. Register Addressing Mode
2. Immediate Addressing Mode
3. Absolute (or Direct) Addressing Mode
4. Register Indirect Addressing Mode
5. Index Addressing Mode
6. Base with Index Addressing Mode

Register Addressing Mode

- The operand is the content of a processor register. Register name is specified in the instruction.
- Effective Address of the Operand: Register name specified in the instruction

ADD R1, R0, R1 ; R1 ← [R0] + [R1]

Before Executing the Instruction

Registers	Contents
R0	8
R1	**2**

After Executing the Instruction

Registers	Contents
R0	8
R1	**10**

Register Addressing Mode

- The operand is the content of a processor register. Register name is specified in the instruction.
- Effective Address of the Operand: Register name specified in the instruction

Move R1, R0 ; R1 ← [R0]

Before Executing the Instruction

Registers	Contents
R0	8
R1	**2**

After Executing the Instruction

Registers	Contents
R0	8
R1	**8**

Immediate Addressing Mode

- The operand is given explicitly in the instruction
- Effective Address of the Operand: Operand value given in the instruction

ADD R1, R1, #10 ; R1 <- R1 +10

Before Executing the Instruction

Registers	Contents
R1	2

After Executing the Instruction

Registers	Contents
R1	12

Immediate Addressing Mode

- The operand is given explicitly in the instruction
- Effective Address of the Operand: Operand value given in the instruction

Move R1 , #10 ; R1 <- 10

Before Executing the Instruction

Registers	Contents
R1	2

After Executing the Instruction

Registers	Contents
R1	10

Absolute (or Direct) Addressing Mode

- The operand is a Memory location. The address of the memory location is given in the instruction explicitly.
- Effective Address of the Operand: Address of the memory location given directly in the instruction

ADD R1 , R2, LOCA ; R1 ← [R2] + [LOCA]

Before Executing the Instruction

	Addr	Memory Contents
LOCA	0x1000	8

R1	2
R2	10

After Executing the Instruction

	Addr	Memory Contents
LOCA	0x1000	8

R1	**18**
R2	10

Register Indirect Addressing Mode

- Here neither the operands nor the their addresses are given explicitly. The instruction provides the information from which the address of the operand is determined i.e., the instruction provides effective address of the operand using register. The indirection is denoted by () sign around register.
- Effective Address of the Operand: Contents of a register that is specified in the instruction

ADD R2, R2, (R1) ; R2 ← R2 + [[R1]]

Before Executing the Instruction

	Addr	Memory Contents
	0x2000	**2**
R1	0x2000	
R2	**8**	

After Executing the Instruction

	Addr	Memory Contents
	0x2000	**2**
R1	0x2000	
R2	**10**	

Question

What will be the contents of the

- Register R1
- And Contents of the Memory location with address 0x1000 and 0x200

After executing the instruction **ADD (R1), (R1), R2**

ADD (R1) , (R1), R2 ; [[R1]] ← [[R1]] + [R]

Before Executing the Instruction

	Addr	Memory Contents
LOCA	0x1000	8
	0x2000	2
R1	0x2000	
R2	10	

After Executing the Instruction

	Addr	Memory Contents
LOCA	0x1000	??
	0x2000	??
R1	??	
R2	??	

Answer

What will be the contents of the

- Register R1
- And Contents of the Memory location with address 0x1000 and 0x200

After executing the instruction **ADD (R1), (R1), R2**

ADD (R1) , (R1), R2 ; [[R1]] ← [[R1]] + [R2]

Before Executing the Instruction

	Addr	Memory Contents
LOCA	0x1000	8
	0x2000	2
R1	0x2000	
R2	10	

After Executing the Instruction

	Addr	Memory Contents
LOCA	0x1000	8
	0x2000	**12**
R1	0x2000	
R2	10	

Question

What does the symbol '#' represent in the instruction MOVE R0 , #55H ?

a. Direct datatype

b. Indirect datatype

c. Immediate datatype

d. Indexed datatype

Immediate

What does the symbol '#' represent in the instruction MOVE R0 , #55H?

a. Direct datatype

b. Indirect datatype

c. Immediate datatype

d. Indexed datatype

Question

In which addressing mode, the operand is fetched from memory

a. Immediate addressing

b. Direct addressing

c. Register addressing

d. None of these

Answer

In which addressing mode, the operand is fetched from memory

a. Immediate addressing

b. Direct addressing

c. Register addressing

d. None of these

Index Addressing Mode

- The effective address of the operand is generated by adding a constant value to the contents of a register specified in the instruction. The register in this case is called as Index register.
- The operation is indicated as X(Ri).
- Effective Address of the Operand: X+Ri where X is a constant value (signed integer) and Ri is the index register.

LOAD R2, 5(R1) ; R2 ← [5+[R1]]

Before Executing the Instruction

	Addr	Memory Contents
	0x2005	**2**

R1	0x2000
R2	**8**

After Executing the Instruction

	Addr	Memory Contents
	0x2005	**2**

R1	0x2000
R2	**2**

Base with Index Addressing Mode

- The effective address of the operand is generated by adding two registers specified in the instruction.
- The operation is indicated as (Ri , Rj).
- Effective Address of the Operand: Ri + Rj where Ri is used to contain offset and Rj is the base register

LOAD R2, (R1, R3) ; R2 ← [[R1]+[R3]]

Before Executing the Instruction

	Addr	Memory Contents
	0x2005	**6**

R1	0x2000
R2	**8**
R3	5

After Executing the Instruction

	Addr	Memory Contents
	0x2005	**6**

R1	0x2000
R2	**6**
R3	5

Question

The addressing mode used in the instruction

Move R1, 8(R2) is

a. Register and Index
b. Register and Direct
c. Register and Immediate

Answer

The addressing mode used in the instruction

Move R1, 8(R2) is

a. Register and Index
b. Register and Direct
c. Register and Immediate

Question

Register R1 of the computer contain decimal value 1200. What is the effective address of the source operand for the instruction Load 20(R1),R5 . Assume instruction Format “Load SourceOperand, DestinationOperand”.

Answer

Register R1 of computer contain decimal value 1200. What is the effective address of the source operand for the instruction Load 20(R1),R5 . Assume instruction Format "Load SourceOperand, DestinationOperand".

Effective Address: 1220

Question

Register R1 and R2 of computer contains the decimal value 1200 and 4600. What is the effective address of the destination operand for the instruction

Store R5,30(R1,R2)

Assume instruction Format

“Store SourceOperand, DestinationOperand”

Answer

Register R1 and R2 of computer contains the decimal value 1200 and 4600. What is the effective address of the destination operand for the instruction

Store R5,30(R1,R2)

Assume instruction Format

“Store SourceOperand, DestinationOperand”

Effective Address: **5830**=30+1200+4600

Summarizing Addressing Modes

RISC-type addressing modes.

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute	LOC	EA = LOC
Register indirect	(R*i*)	EA = [R*i*]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]

EA = effective address
Value = a signed number
X = index value

Question

Registers R4 and R5 contain the decimal numbers 2000 and 3000 before each of the following addressing modes is used to access a memory operand. What is the effective address (EA) in each case?

i. 12(R4)

ii. (R4,R5)

iii. 28(R4,R5)

Question

Registers R4 and R5 contain the decimal numbers 2000 and 3000 before each of the following addressing modes is used to access a memory operand. What is the effective address (EA) in each case?

i. 12(R4)

EA=[R4]+12

2012=2000+12

ii. (R4,R5)

EA=[R4]+[R5]

5000=2000+3000

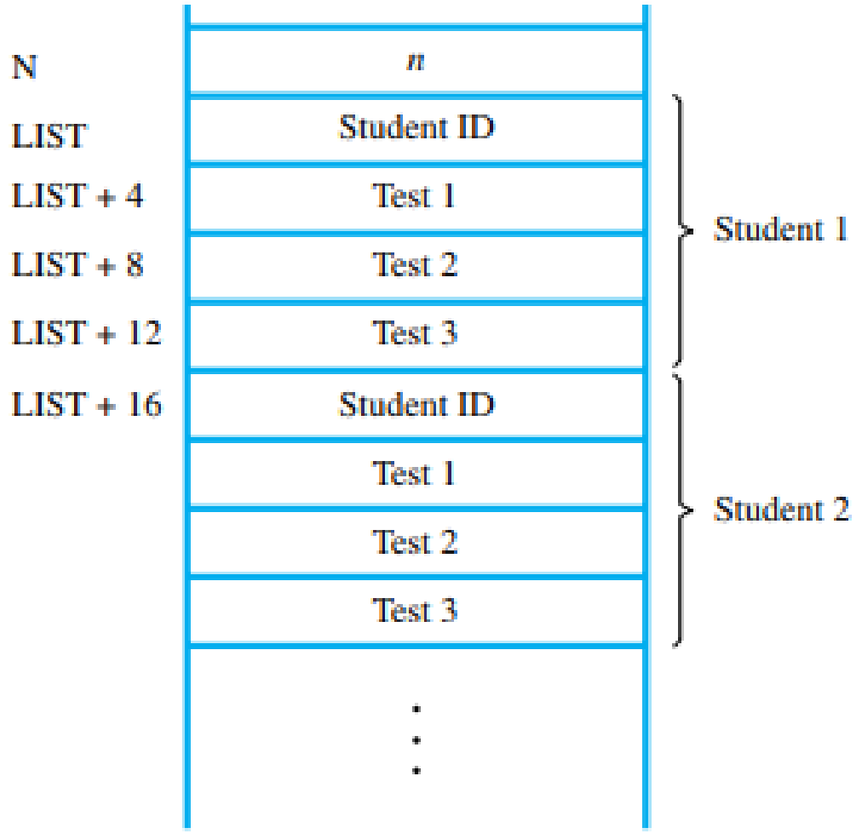
iii. 28(R4,R5)

EA= [28]+[R4]+[R5]

5028= 28+2000+3000

Index Addressing Mode used in accessing Test Scores

```
Move R2, #LIST          ; Get the address LIST.
Clear R3
Clear R4
Clear R5
Load R6, N              ; Load the value n.
LOOP: Load R7, 4(R2)    ; Add the mark for next student's
Add R3, R3, R7          ; Test 1 to the partial sum.
Load R7, 8(R2)          ; Add the mark for that student's
Add R4, R4, R7          ; Test 2 to the partial sum.
Load R7, 12(R2)         ; Add the mark for that student's
Add R5, R5, R7          ; Test 3 to the partial sum.
Add R2, R2, #16         ; Increment the pointer.
Subtract R6, R6, #1       ; Decrement the counter.
Branch_if_[R6]>0 LOOP   ; Branch back if not finished.
Store R3, SUM1          ; Store the total for Test 1.
Store R4, SUM2          ; Store the total for Test 2.
Store R5, SUM3          ; Store the total for Test 3.
```



Program to find sum of Test1, Test2 and Test3 marks of all students

	Memory 4 Bytes	Address	
N	2	1000	
List	BM01	1004	Student ID
	1	1008	Test1 Marks
	2	1012	Test2 Marks
	3	1016	Test3 Marks
	BM02	1020	Student ID
	10	1024	Test1 Marks
	20	1028	Test2 Marks
	30	1032	Test3 Marks
Sum1		1036	
Sum2		1040	
Sum3		1044	

Program to find sum of Test1, Test2 and Test3 marks of all students

	Memory 4 Bytes	Address	
N	2	1000	
List	BM01	1004	Student ID
	1	1008	Test1 Marks
	2	1012	Test2 Marks
	3	1016	Test3 Marks
	BM02	1020	Student ID
	10	1024	Test1 Marks
	20	1028	Test2 Marks
	30	1032	Test3 Marks
Sum1		1036	
Sum2		1040	
Sum3		1044	

	Move R2, #List	3000
	Clear R3	3004
	Clear R4	3008
	Clear R5	3012
	Load R6, N	3016
Loop:	Load R7, 4(R2)	3020
	Add R3, R3, R7	3024
	Load R7, 8(R2)	3028
	Add R4, R4, R7	3032
	Load R7, 12(R2)	3036
	Add R5, R5, R7	3040
	Add R2, R2, #16	3044
	Subtract R6, R6, #1	3048
	Branch_if_[R6] >0 **Loop**	3052
	Store R3, Sum1	3056
	Store R4, Sum2	3060
	Store R5, Sum3	3064

After executing the program

	Memory 4 Bytes	Address	
N	2	1000	
List	BM01	1004	Student ID
	1	1008	Test1 Marks
	2	1012	Test2 Marks
	3	1016	Test3 Marks
	BM02	1020	Student ID
	10	1024	Test1 Marks
	20	1028	Test2 Marks
	30	1032	Test3 Marks
Sum1	**11**	1036	
Sum2	**22**	1040	
Sum3	**33**	1044	

	Move R2, #List	3000
	Clear R3	3004
	Clear R4	3008
	Clear R5	3012
	Load R6, N	3016
Loop:	Load R7, 4(R2)	3020
	Add R3, R3, R7	3024
	Load R7, 8(R2)	3028
	Add R4, R4, R7	3032
	Load R7, 12(R2)	3036
	Add R5, R5, R7	3040
	Add R2, R2, #16	3044
	Subtract R6, R6, #1	3048
	Branch_if_[R6] >0 **Loop**	3052
	Store R3, Sum1	3056
	Store R4, Sum2	3060
	Store R5, Sum3	3064

Question

The list of student marks shown in Figure 2.10 is changed to contain j test scores for each student. Assume that there are n students. Write a RISC-style program for computing the sums of the scores on each test and store these sums in the memory word locations at addresses SUM, SUM + 4, SUM + 8,.... The type of program shown in Figure 2.11 for the 3-test case cannot be used. Use two nested loops. The inner loop should accumulate the sum for a particular test, and the outer loop should run over the number of tests, j. Assume that the memory area used to store the sums has been cleared to zero initially.

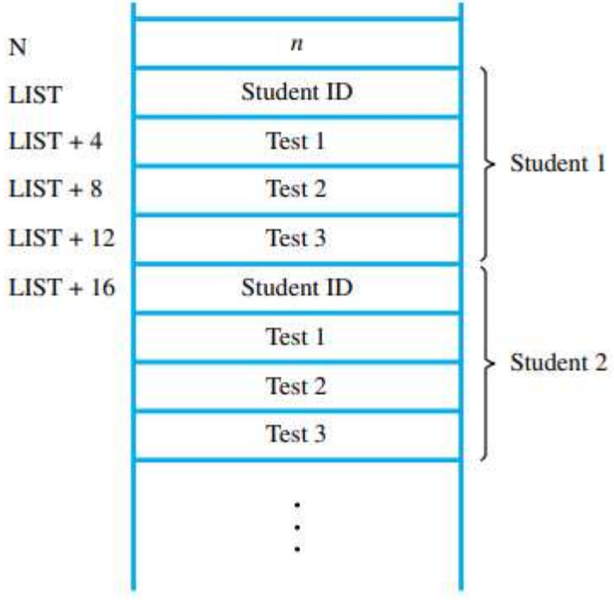


Figure 2.10 A list of students' marks.

	Move	R2, #LIST	Get the address LIST.
	Clear	R3	
	Clear	R4	
	Clear	R5	
	Load	R6, N	Load the value *n*.
LOOP:	Load	R7, 4(R2)	Add the mark for next student's
	Add	R3, R3, R7	Test 1 to the partial sum.
	Load	R7, 8(R2)	Add the mark for that student's
	Add	R4, R4, R7	Test 2 to the partial sum.
	Load	R7, 12(R2)	Add the mark for that student's
	Add	R5, R5, R7	Test 3 to the partial sum.
	Add	R2, R2, #16	Increment the pointer.
	Subtract	R6, R6, #1	Decrement the counter.
	Branch_if_[R6]>0	LOOP	Branch back if not finished.
	Store	R3, SUM1	Store the total for Test 1.
	Store	R4, SUM2	Store the total for Test 2.
	Store	R5, SUM3	Store the total for Test 3.

Figure 2.11 Indexed addressing used in accessing test scores in the list in Figure 2.10.

Solution Approach

Memory word location J contains the number of tests, j, and memory word location N contains the number of students, n. The list of student marks begins at memory word location List in the format as shown in figure. The parameter Stride = 4(j+1) is the distance in bytes between scores on a particular test for adjacent students in the list. The Based with index addressing mode (R1, R2) is used to access the scores on a particular test. Register R1 points to the test score for student-1 , and R2 is incremented by Stride in the inner loop to access scores on the sum test by successive students in the list.

	Memory 4 Bytes	Address	
J	3	1000	
N	2	1004	
List	BM01	1008	Student ID
	1	1012	Test1 Marks
	2	1016	Test2 Marks
	3	1020	Test3 Marks
	BM02	1024	Student ID
	10	1028	Test1 Marks
	20	1032	Test2 Marks
	30	1036	Test3 Marks
Sum	**11**	1040	
Sum+4	**22**	1044	
Sum+8	**33**	1046	

Solution Approach

Memory word location J contains the number of tests, j, and memory word location N contains the number of students, n. The list of student marks begins at memory word location List in the format as shown in figure. The parameter Stride = 4(j+1) is the distance in bytes between scores on a particular test for adjacent students in the list. The Based with index addressing mode (R1, R2) is used to access the scores on a particular test. Register R1 points to the test score for student-1 , and R2 is incremented by Stride in the inner loop to access scores on the sum test by successive students in the list.

	Memory 4 Bytes	Address	
J	3	1000	
N	2	1004	
List	BM01	1008	Student ID
	1	1012	Test1 Marks
	2	1016	Test2 Marks
	3	1020	Test3 Marks
	BM02	1024	Student ID
	10	1028	Test1 Marks
	20	1032	Test2 Marks
	30	1036	Test3 Marks
Sum	**11**	1040	
Sum+4	**22**	1044	
Sum+8	**33**	1046	

Program

	Memory 4 Bytes	Address	
J	3	1000	
N	2	1004	
List	BM01	1008	Student ID
	1	1012	Test1 Marks
	2	1016	Test2 Marks
	3	1020	Test3 Marks
	BM02	1024	Student ID
	10	1028	Test1 Marks
	20	1032	Test2 Marks
	30	1036	Test3 Marks
Sum		1040	
Sum+4		1044	
Sum+8		1046	

```
        Move R4, J
        Add R4, R4, #1
        Multply R4, R4, #4
        Move R1, #List
        Add R1, R1, #4
        Move R3, #Sum
        Move R10, J
OUTER:  Move R11, N
        Clear R2
        Clear R0
INNER:  Add R0, R0, (R1, R2)
        Add  R2, R2, R4
        Subtract R11, R11, #1
        Branch_if_[R11] > 0 INNER
        Load (R3), R0
        Add R3, R3, #4
        Add R1, R1, #4
        Subtract R10, R10, #1
        Branch_if_[R10] > 0 OUTER
```

Question

Can the program given will work to find the sum of test scores of all students if we increase the number of tests to four

	Memory 4 Bytes	Address	
J	4	1000	
N	2	1004	
List	BM01	1008	Student ID
	1	1012	Test1 Marks
	2	1016	Test2 Marks
	3	1020	Test3 Marks
	4	1024	Test4 Marks
	BM02	1028	Student ID
	10	1032	Test1 Marks
	20	1036	Test2 Marks
	30	1040	Test3 Marks
	40	1044	Test4 Marks
Sum	??	1046	
Sum+4	??	1050	
Sum+8	??	1054	
Sum+12	??		

	Move R4, J
	Add R4, R4, #1
	Multply R4, R4, #4
	Move R1, #List
	Add R1, R1, #4
	Move R3, #Sum
	Move R10, J
OUTER:	Move R11, N
	Clear R2
	Clear R0
INNER:	Add R0, R0, (R1, R2)
	Add R2, R2, R4
	Subtract R11, R11, #1
	Branch_if_[R11] > 0 **INNER**
	Load (R3), R0
	Add R3, R3, #4
	Add R1, R1, #4
	Subtract R10, R10, #1
	Branch_if_[R10] > 0 **OUTER**

Answer

Can the program given will work find the sum of test scores of all students if we increase the number of tests to four: **Yes**

	Memory 4 Bytes	Address	
J	4	1000	
N	2	1004	
List	BM01	1008	Student ID
	1	1012	Test1 Marks
	2	1016	Test2 Marks
	3	1020	Test3 Marks
	4	1024	Test4 Marks
	BM02	1028	Student ID
	10	1032	Test1 Marks
	20	1036	Test2 Marks
	30	1040	Test3 Marks
	40	1044	Test4 Marks
Sum	**11**	1046	
Sum+4	**22**	1050	
Sum+8	**33**	1054	
Sum+12	**44**		

```
        Move R4, J
        Add R4, R4, #1
        Multply R4, R4, #4
        Move R1, #List
        Add R1, R1, #4
        Move R3, #Sum
        Move R10, J
OUTER:  Move R11, N
        Clear R2
        Clear R0
INNER:  Add R0, R0, (R1, R2)
        Add  R2, R2, R4
        Subtract R11, R11, #1
        Branch_if_[R11] > 0 INNER
        Load (R3), R0
        Add R3, R3, #4
        Add R1, R1, #4
        Subtract R10, R10, #1
        Branch_if_[R10] > 0 OUTER
```

Assembler Directives

- Assembler Directive: a statement to give direction to the assembler to perform task of the assembly process.
- Instructions: translated to the machine code by the assembler
- Assembler Directives: are not translated to the machine codes

List of Assembler Directives

1. **ORIGIN** directive tells the assembler where to load instructions and data into memory. It changes the program counter to the value specified by the expression in the operand field.
2. **RESERVE** directives are used for reserving space for uninitialized data. The reserve directives take a single operand that specifies the number of units of space to be reserved
3. **DATAWORD** directives are used to initialize the memory location with specified value
4. **EQU**: The equate directive is used to substitute values for symbols or labels.

Assembly Language with Assembler directives: To add n numbers

```
        ORIGIN   100
        Load    R2, N
        Clear R3
        Move R4, #Num1
LOOP:   Load R5, (R4)
        ADD R3, R3, R5
        ADD R4, R4, #4
        Subtract  R2, R2, #1
        Branch_if_[R2] > 0 LOOP
        Store R3, SUM
        ORIGIN 200
SUM:    RESERVE 4
N     : DATAWORD 3
NUM:    DATAWORD 10, 20, 30
        END
```

Memory
4 Bytes

		Address
	Load R2, N	100
	Clear R3	104
	Move R4, #Num1	108
Loop:	Load R5, (R4)	112
	Add R3, R3, R5	116
	Add R4, R4, #4	120
	Subtract R2, R2, #1	124
	Branch_if_[R2] > 0 LOOP	128
	Store R3, SUM	132
	.....	...
SUM		200
N	3	204
NUM	10	208
	20	212
	30	216

Question

Write a RISC-style program that finds the number of negative integers in a list of n 32-bit integers and stores the count in location NEGNUM. The value n is stored in memory location N, and the first integer in the list is stored in location NUMBERS. Include the necessary assembler directives and a sample list that contains six numbers, some of which are negative.

```
                ORIGIN    0X100
                LOAD      R2, N
                CLEAR     R3
                MOVE      R4, #NUMBERS
LOOP1:          LOAD      R5, (R4)
BRANCH_IF[R5] > 0   LOOP2
                ADD       R3, R3, #1
LOOP2:          ADD       R4, R4, #4
                SUB       R2, R2, #1
BRANCH_IF[R2] > 0   LOOP1
                STORE     R3, NEGNUM

                ORIGIN    0X500
N:              DATAWORD  6
NEGNUM:         RESERVE   4
NUMBERS:        DATAWORD  10, -20, 5
                DATAWORD  - 10, 20, -15
```

Thanks for Listening

END of Unit-1