COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

COURSE OBJECTIVES

This course will enable students to

- Summarize the fundamental concepts of the organization and architecture of a computer.
- Analyze taxonomy of Execution, Processor, Memory and I/O Units.
- Explain the pipelining principles, Data dependencies and hazards, SIMD and Multiprocessor concepts.

TEXTBOOKS AND REFERENCE BOOKS

Textbooks:

1. Carl Hamacher, ZvonkoVranesic and SafwatZaky, *Computer Organization and Embedded Systems,* (6e), McGraw Hill Publication, 2012.
2. William Stallings, *Computer Organization and Architecture – Designing for Performance,* (9e), PHI, 2015.
3. Mohammed Rafiquzzaman and Rajan Chandra, *Modern Computer Architecture,* Galgotia Publications Pvt. Ltd., 2010.

Reference Books:

1. D.A. Patterson and J. L. Hennessy, Computer Organization and Design-The Hardware/Software Interface, (5e), Morgan Kaufmann, 2014.
2. J. P. Hayes, Computer Architecture and Organization, McGraw Hill Publication, 1998.

INTRODUCTION

INTRODUCTION

- What is Computer Organization and Architecture?

INTRODUCTION

- Computer
- Addition of 2 bits

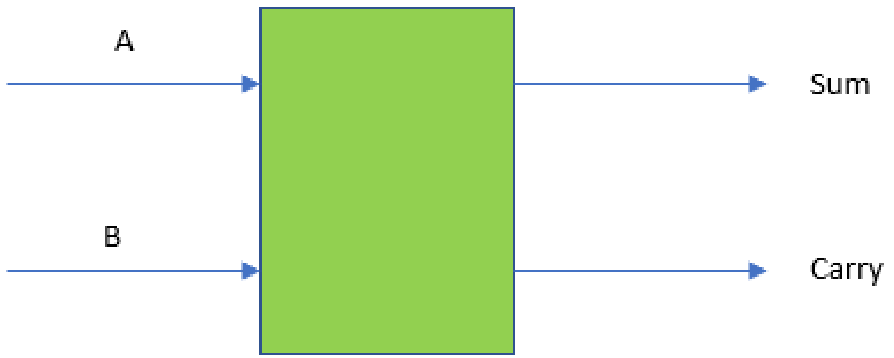


Figure 2: Structure to add two bits

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

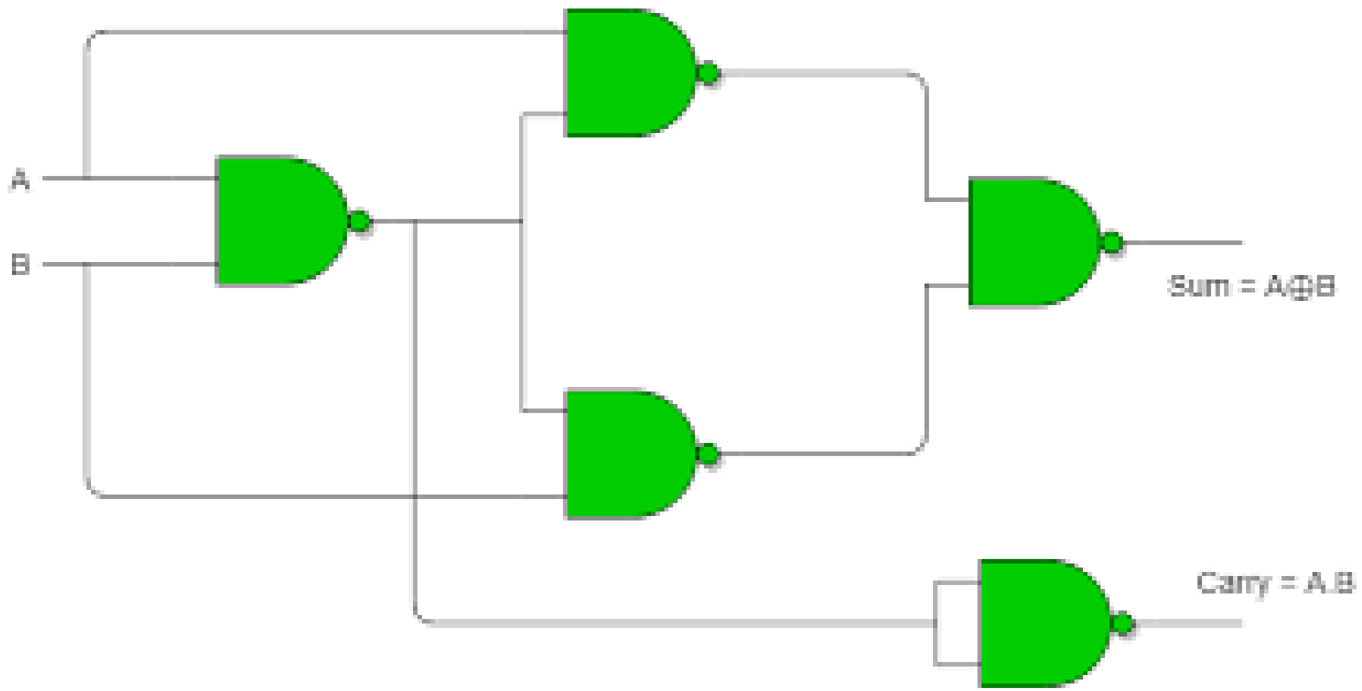


Figure 3: Half Adder using NAND gates.
Source: https://www.geeksforgeeks.org/

INTRODUCTION

- Computer
- Addition of 2 bits

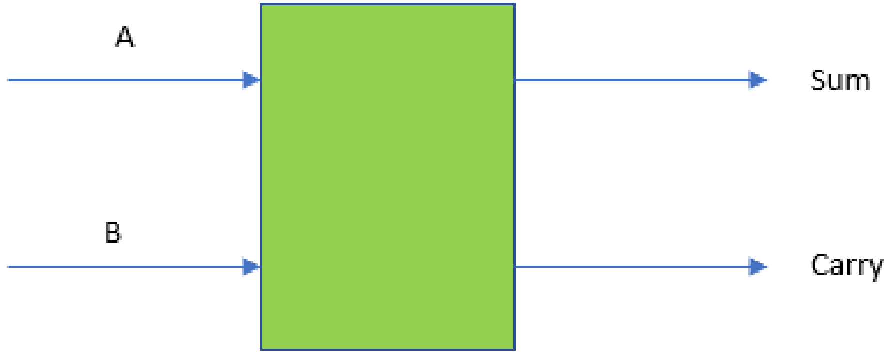


Figure 2: Structure to add two bits

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

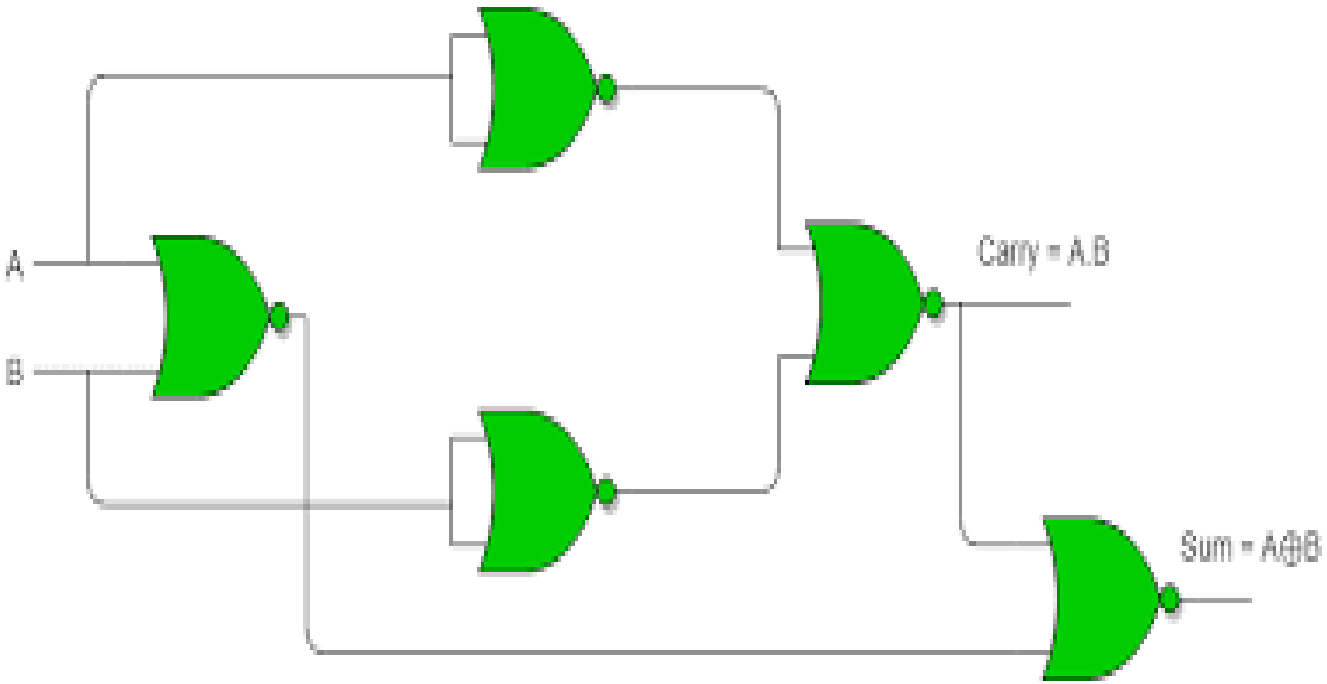


Figure 4: Half Adder using NOR gates.
Source: https://www.geeksforgeeks.org/

INTRODUCTION

- Computer Architecture: refers to those attributes of a system visible to a programmer or those attributes that have a direct impact on the logical execution of a program.
  - Example: instruction sets, the number of bits used to represent various datatypes, I/O mechanisms and techniques for addressing memory.

- Computer Organization: refers to the operational units and their interconnections that realize the architectural specifications.
  - Example: hardware details transparent to the programmer, such as control signals, interfaces between the computer and the peripherals; and the memory technology used.

INTRODUCTION

- Modules:

1. Basic structure of computers
2. Instruction set architecture
3. Arithmetic and logic unit
4. Control unit
5. Memory systems
6. Input/output organization
7. Introduction to parallel architecture

GUESS……..

- Internet is an example for input unit of a computer.
  A. True
  B. False

- The function of instruction "Load R1, LOC" is to read the data from R1 and store it in LOC.
  A. True
  B. False

BASIC STRUCTURE OF COMPUTERS

- The different types of computers
- The basic structure of a computer and its operation
- Machine instructions and their execution
- Number and character representations
- Addition and subtraction of binary numbers

TYPES OF COMPUTERS

1. Embedded computers
2. Personal computers
3. Servers and Enterprise systems
4. Supercomputers and Grid computers

TYPES OF COMPUTERS

- **Embedded computers**
  - Integrated into a larger device or system in order to automatically monitor and control a physical process or environment.
  - Used for a specific purpose.
  - Applications include industrial and home automation, appliances, telecommunication products, and vehicles

TYPES OF COMPUTERS

- **Personal computers**
  - Primarily for dedicated individual use.
  - Types of personal computers
    i. Desktop computers
    ii. Workstation computers
    iii. Portable and Notebook computers
  - Applications include general computation, document preparation, computer-aided design, audiovisual entertainment, interpersonal communication, and Internet browsing

TYPES OF COMPUTERS

- **Servers and Enterprise systems**
  - Large computers shared by many users through personal computer over a public or private network.
  - May host large databases and provide information processing for a government agency or a commercial organization,

TYPES OF COMPUTERS

- **Supercomputers and Grid computers**
  - Offer the highest performance
  - Most expensive and physically the largest category of computers.
  - **Supercomputer**
    - High-cost systems.
    - Performance of a supercomputer is measured in floating-point operations per second (FLOPS) instead of million instructions per second (MIPS)
    - Applications include high demanding computation systems such as weather forecasting, engineering design and simulation, and scientific work.

TYPES OF COMPUTERS

- **Supercomputers and Grid computers**
  - **Grid computers**
    - More cost-effective alternative
    - A grid of large number of personal computers and disk storage units are created using a physically distributed high-speed network, which is managed as a coordinated computing resource.
    - The computational workload is distributed across the grid to achieve high performance on large applications ranging from numerical computation to information searching.

TYPES OF COMPUTERS

- **Cloud Computing**
  - Distributed computing and storage server resources for individual, independent, computing need through personal computers.
  - Communication facility via internet.
  - Cloud hardware and software service providers operate as a utility and charge on a pay-as-you-use basis.

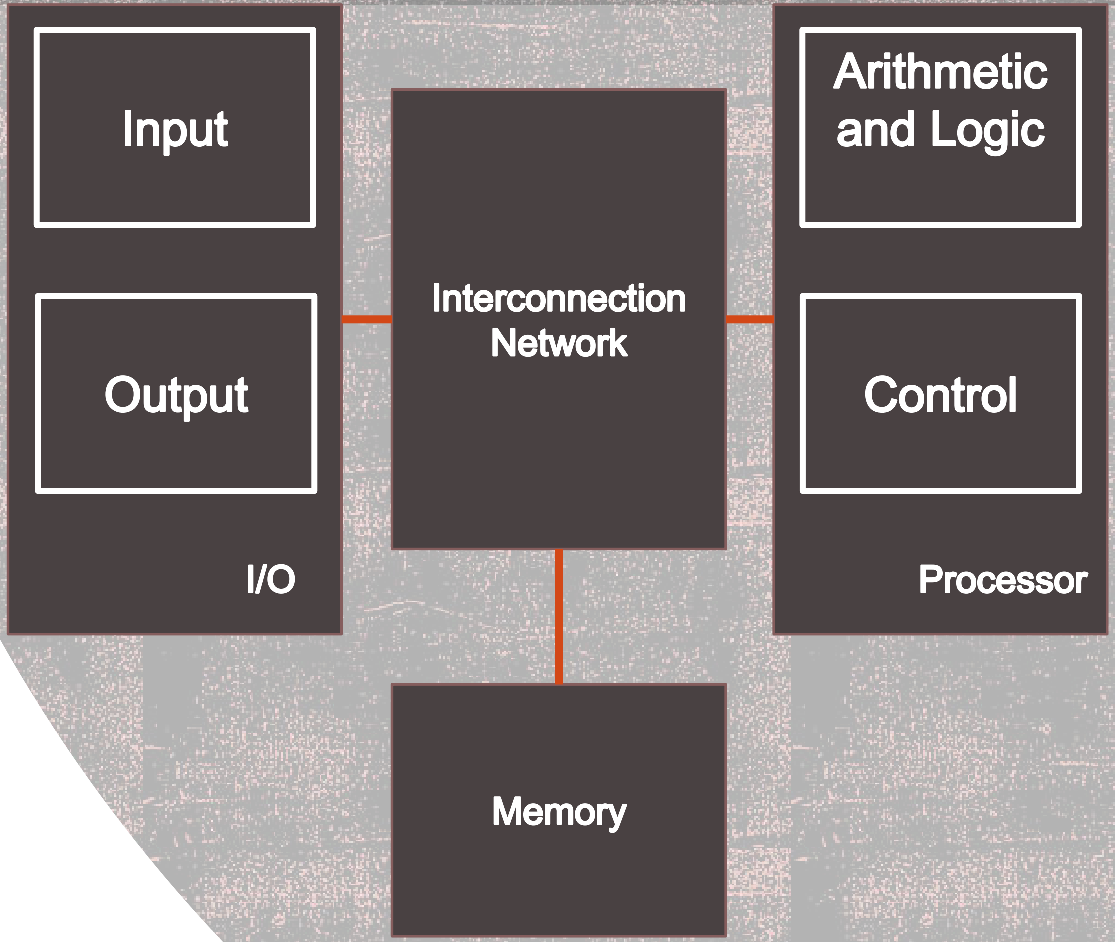
TYPES OF COMPUTERS

1. Embedded computers
2. Personal computers
3. Servers and Enterprise systems
4. Supercomputers and Grid computers

- Cloud Computing

FUNCTIONAL UNITS

- Input
- Output
- Memory
- Arithmetic and Logic Unit
- Control Unit

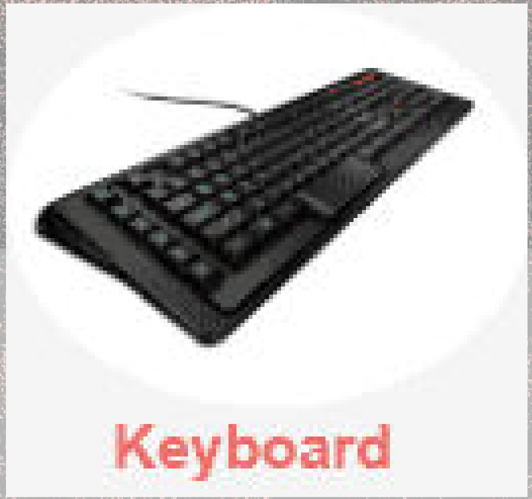


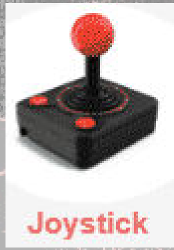
HOW DOES THE COMPUTER HANDLE INFORMATION?

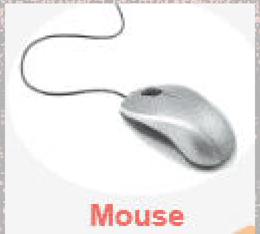
- Instructions are explicit commands that
  - govern the transfer of information within a computer as well as between the computer and its I/O devices
  - specify the arithmetic and logic operations to be performed
- A program is
  - a list of instructions which performs a task.
  - stored in the memory.
- Data are
  - numbers and characters that are used as operands by the instructions.
  - stored in the memory.
- Instructions and data are encoded in binary format (0 and 1)

INPUT UNIT

- **Keyboard**
  - Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor
- **Graphic input**
  - touchpad, mouse, joystick, and trackball.
- **Audio and Video input**
  - The audio input captured by microphones and video input captured by camera are sampled and converted into digital codes for storage and processing.
- **Digital communication facility: Internet**
  - provides input to a computer from other computers and database servers.

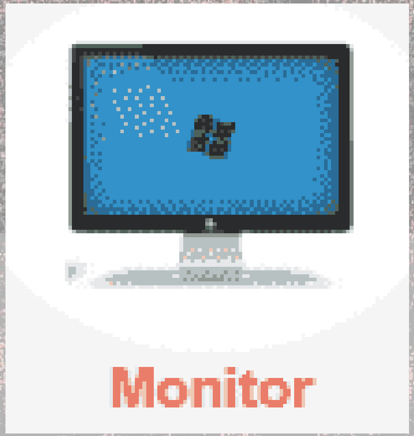


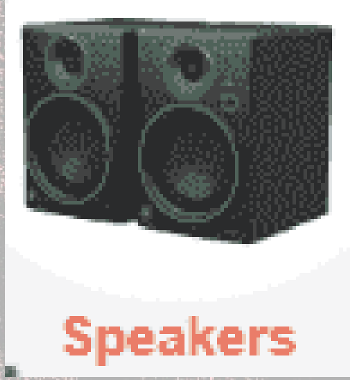




OUTPUT UNIT

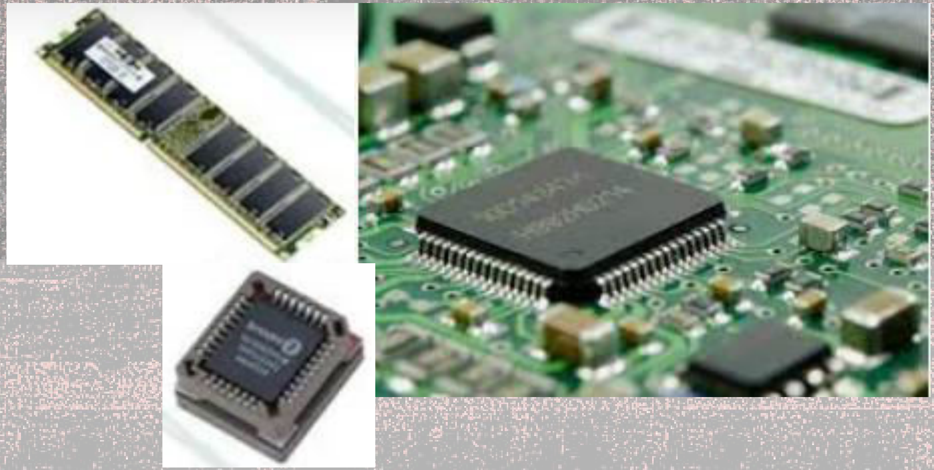
- send processed results to the outside world.
- Graphic display
- Printers
- Audio output: Speakers, headphones





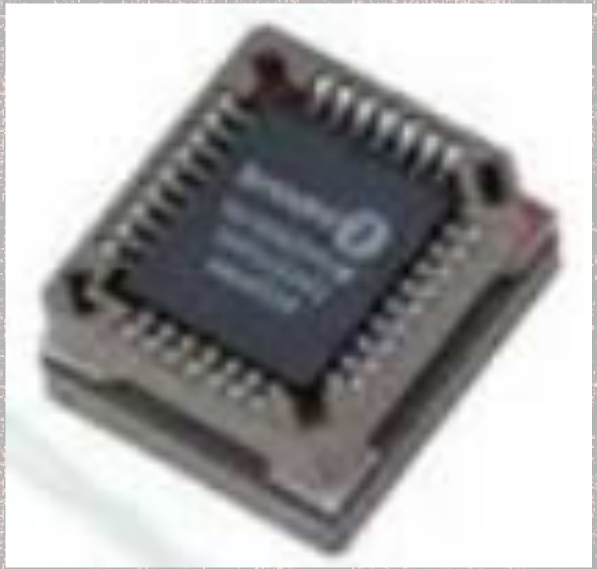
MEMORY UNIT

- **Store programs and data**
- **Two classes of storage**
  - Primary memory or main memory
  - Secondary Storage



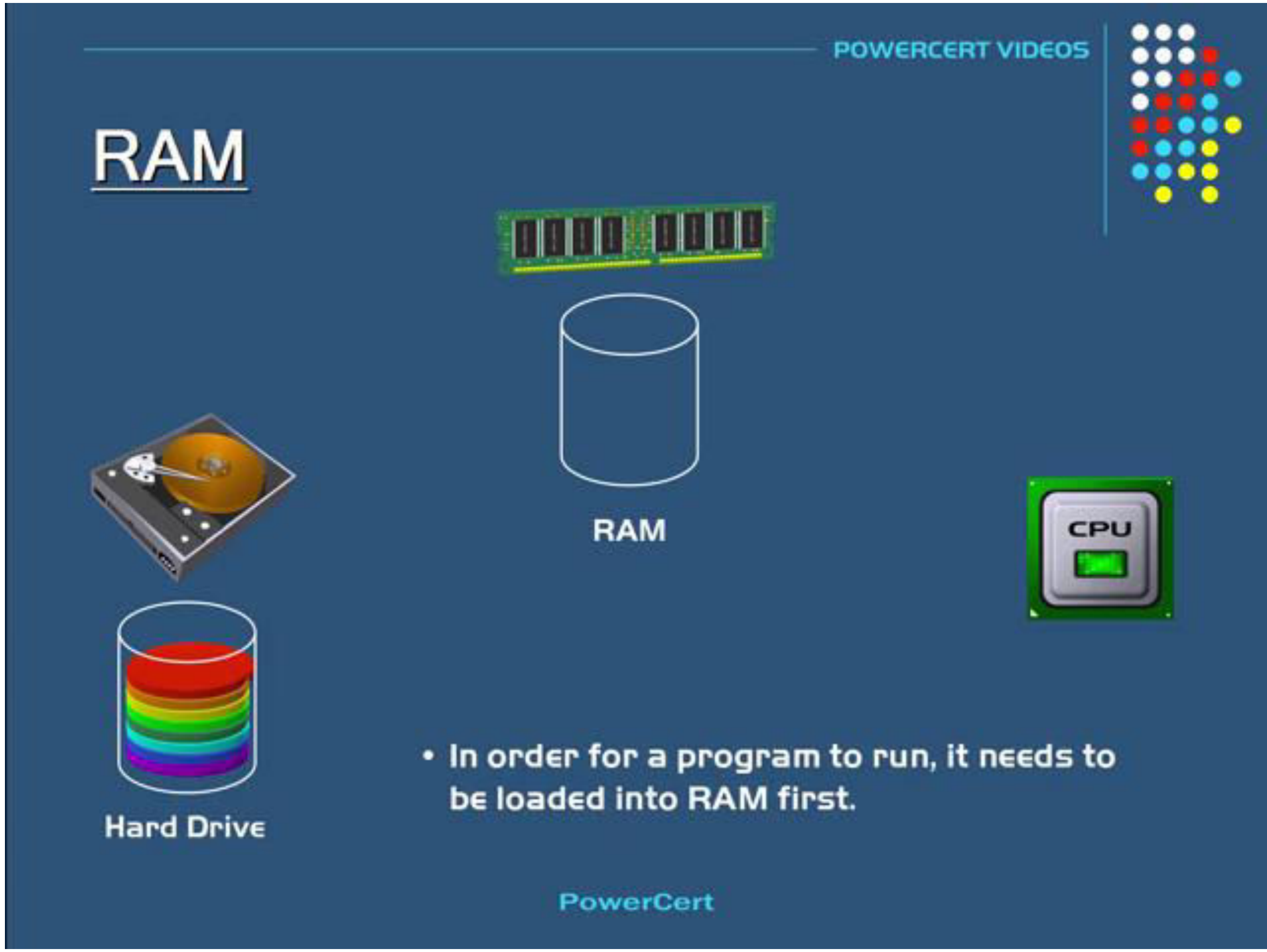
MEMORY UNIT

- **Primary Memory**
  - Fast memory
  - Programs are stored in this memory while being executed
  - Large number of semiconductor storage cells capable of storing one bit of information
  - Words: groups of cells, fixed size.
  - Word length of the computer: number of bits in each word -16, 32, or 64 bits.
  - Address: associated with each word location. Consecutive numbers, starting from 0, that identify successive locations.
  - Random-access memory (RAM): A memory in which any location can be accessed in a short and fixed amount of time after specifying its address
  - Memory access time: The time required to access one word. This time is independent of the location of the word being accessed. Typically ranges from a few nanoseconds (ns) to about 100 ns.



MEMORY UNIT

- **Primary Memory**
  - All programs stored in the secondary memory must be loaded from secondary memory to primary memory (RAM) before its execution.
  - The data is then transferred to the cache, a smaller, faster RAM unit, for fast retrieval of instructions to the CPU



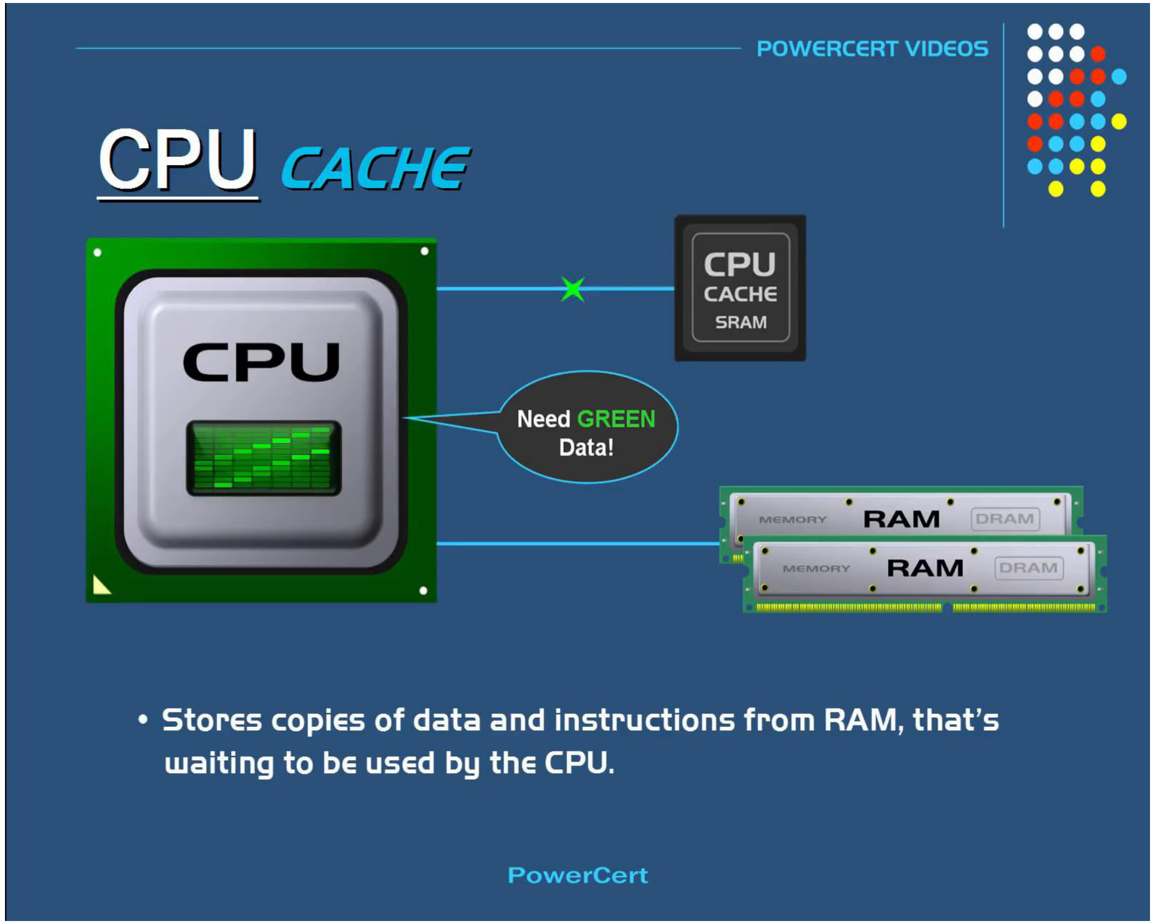
Source: https://gfycat.com/contentmeanacornbarnacle

MEMORY UNIT

- **Cache Memory**
  - Used to hold sections of a program that are currently being executed, along with any associated data
  - is tightly coupled with the processor and is usually contained on the same integrated-circuit chip
  - facilitates high instruction execution rates

MEMORY UNIT

- **Cache Memory**
  - At the start of program execution, the cache is empty.
  - As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache.
  - Suppose several instructions are executed repeatedly as happens in a program loop and these instructions are available in the cache, they can be fetched quickly during the period of repeated use.
  - Similarly, if the same data locations are accessed repeatedly while copies of their contents are available in the cache, they can be fetched quickly.



Source: https://gfycat.com/fluidcheapkitty

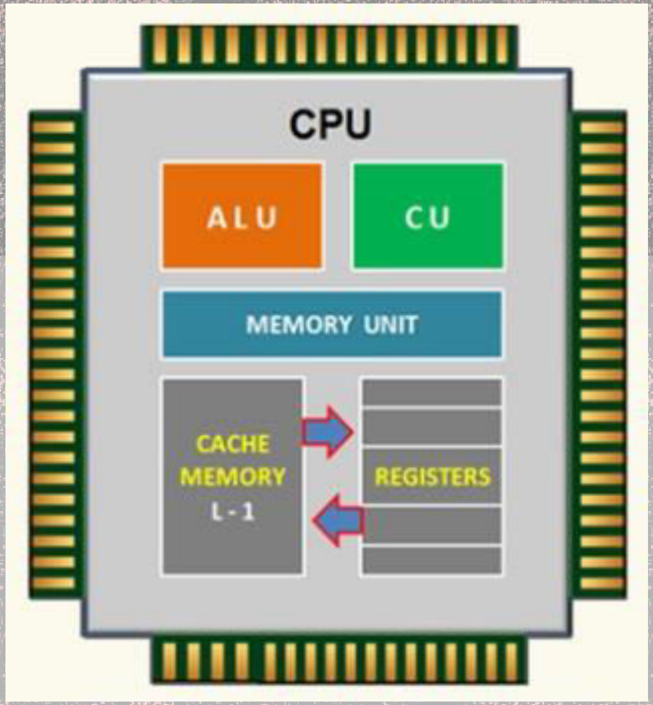
MEMORY UNIT

- **Secondary Memory**
  - used when large amounts of data and many programs must be stored, particularly for information that is accessed infrequently.
  - Access times for secondary storage are longer than for primary memory.
  - magnetic disks, optical disks (DVD and CD), and flash memory devices.

- **Memory Hierarchy**
  - Secondary > Primary > Cache (storage)
  - Secondary < Primary < Cache (speed and cost)

ARITHMETIC & LOGIC UNIT

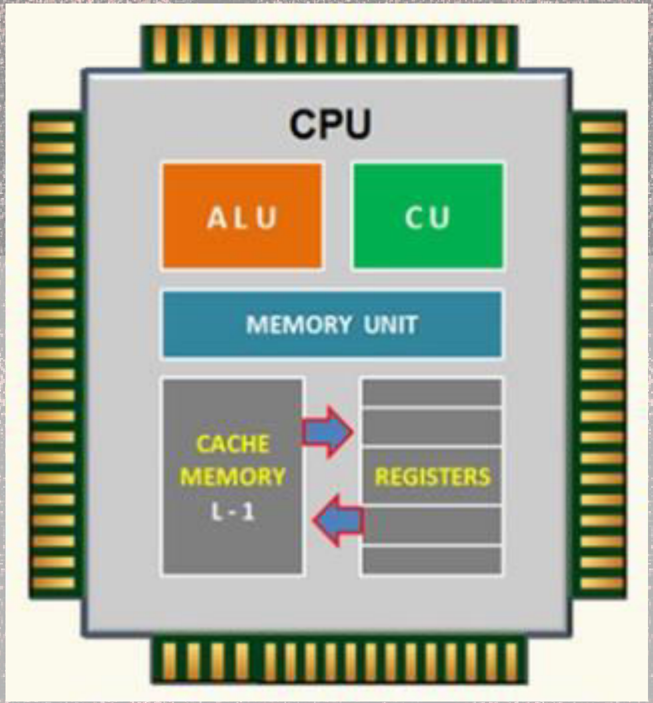
- Most computer operations are executed in ALU of the processor.
- Load the operands into memory – bring them to the processor – perform operation in ALU – store the result back to memory or retain in the processor.
- Registers-high-speed storage elements-store one word of data
- Fast control of ALU



CONTROL UNIT

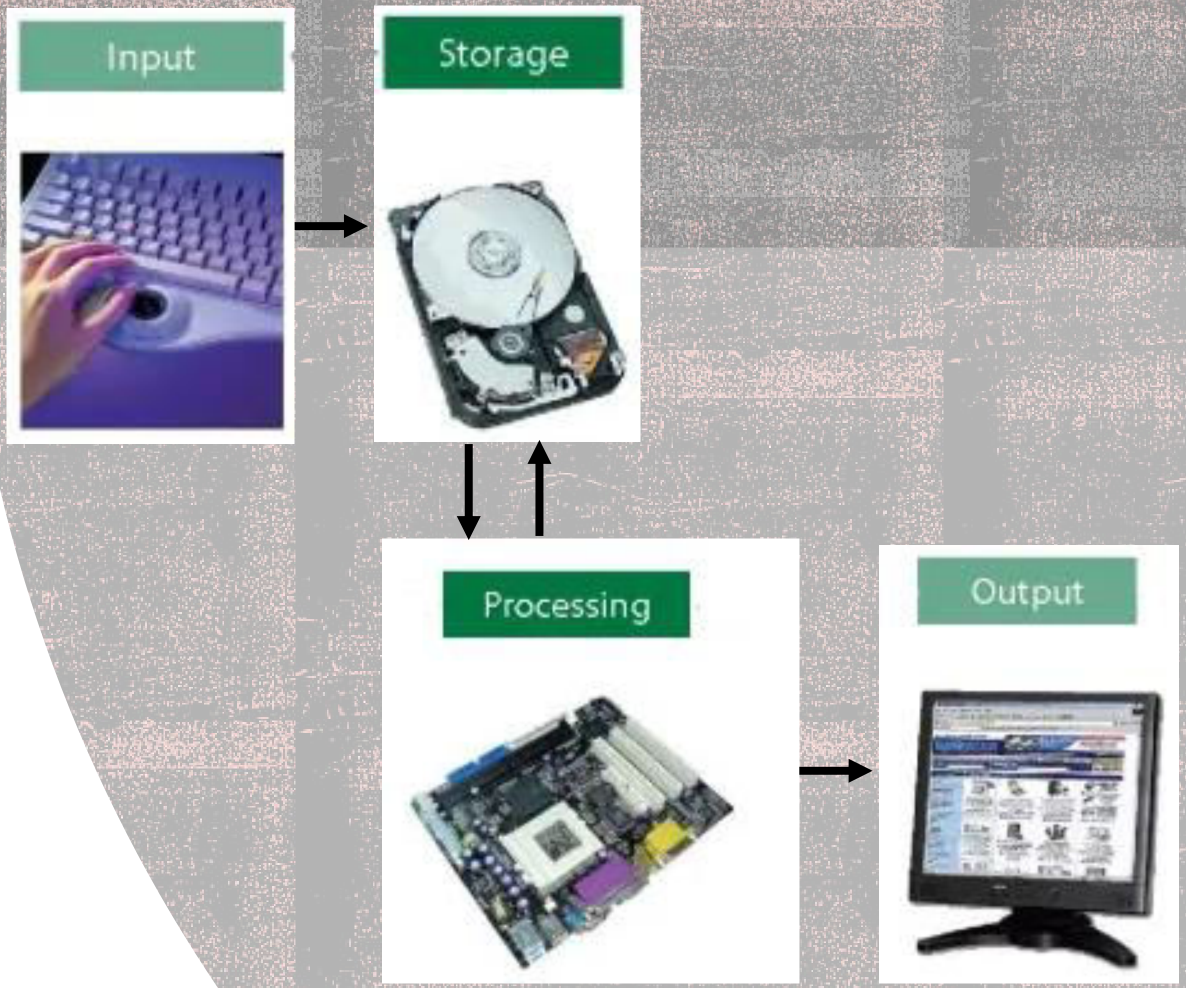
- All computer operations are controlled by the control unit.
- The timing signals that govern the I/O transfers are also generated by the control unit.
- Control unit is usually distributed throughout the machine instead of standing alone.

- What is inside CPU?
  - https://youtu.be/NKYgZH7SBjk



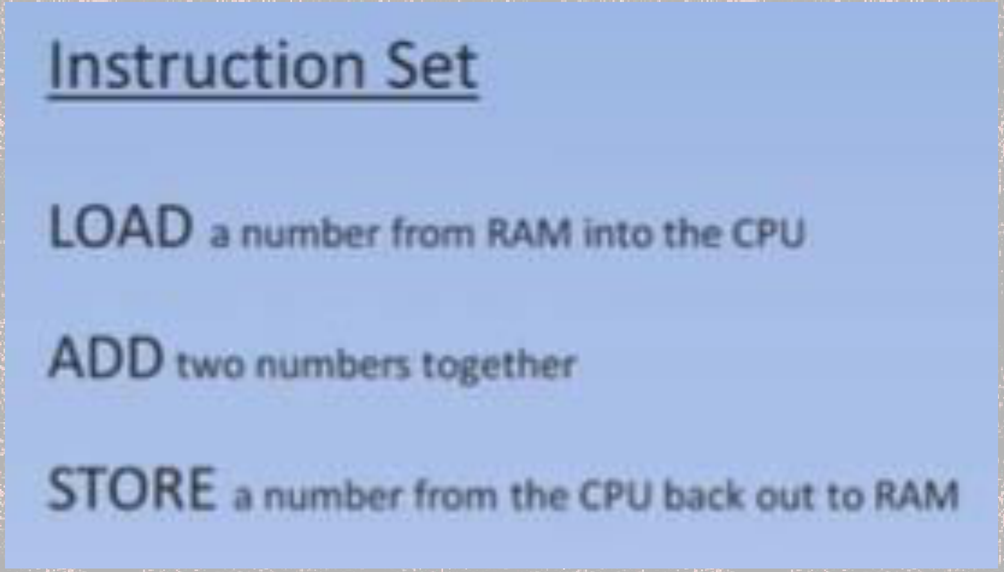
OPERATIONS OF A COMPUTER

- Accept information in the form of programs and data through an input unit and store it in the memory
- Fetch the information stored in the memory, under program control, into an ALU, where it is then processed
- Output the processed information through an output unit
- All activities inside the machine are controlled through a control unit

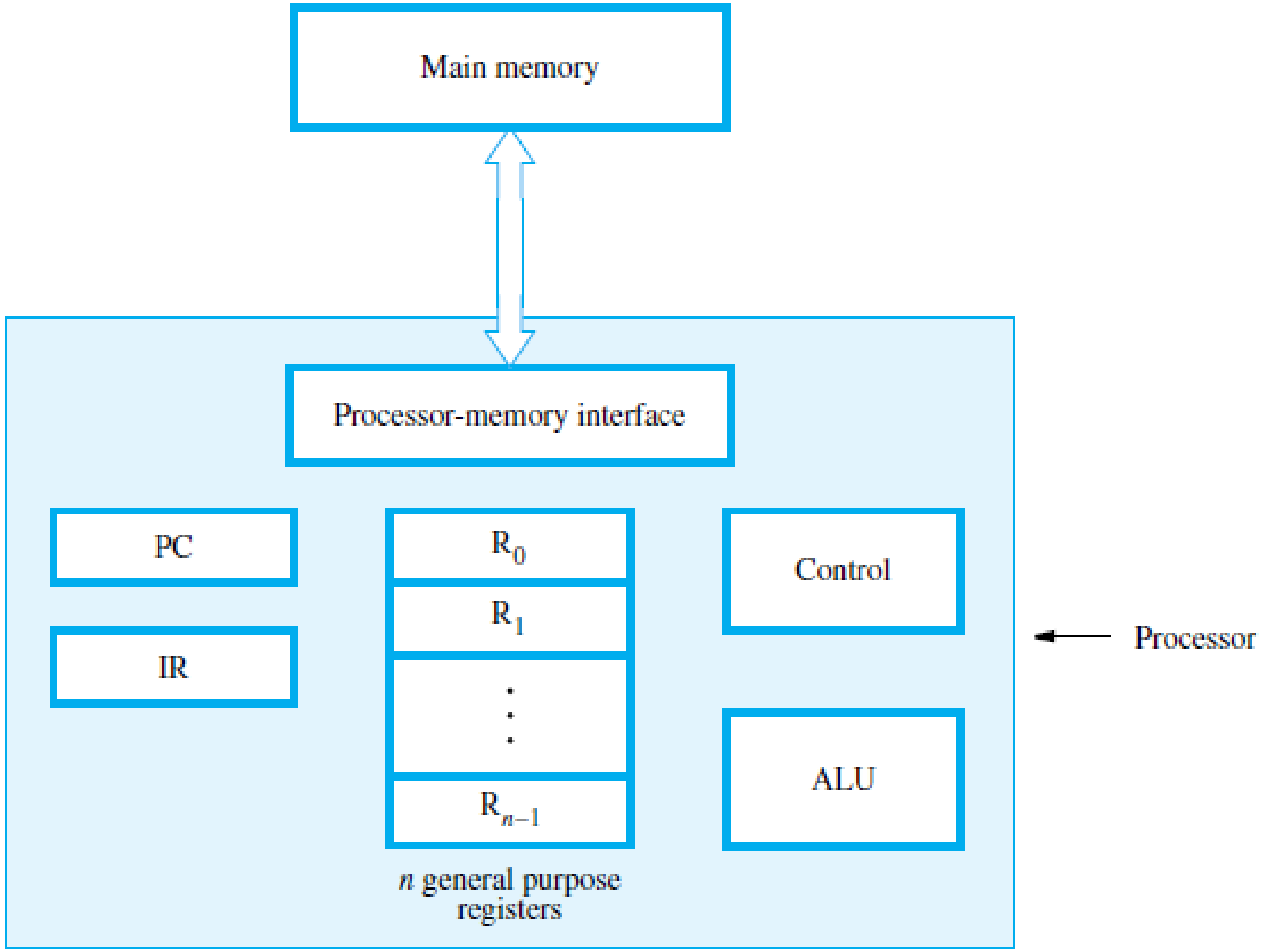


BASIC OPERATIONAL CONCEPTS

- A typical instruction set:
  - Load R2, LOC
  - Add R4, R2, R3
  - Store R4, LOC
- Load R2, LOC
  - reads the contents of a memory location with address represented as LOC and loads them into processor register R2
- Add R4, R2, R3
  - adds the contents of registers R2 and R3 and stores the result in R4
- Store R4, LOC
  - copies the operand in register R4 to memory location LOC

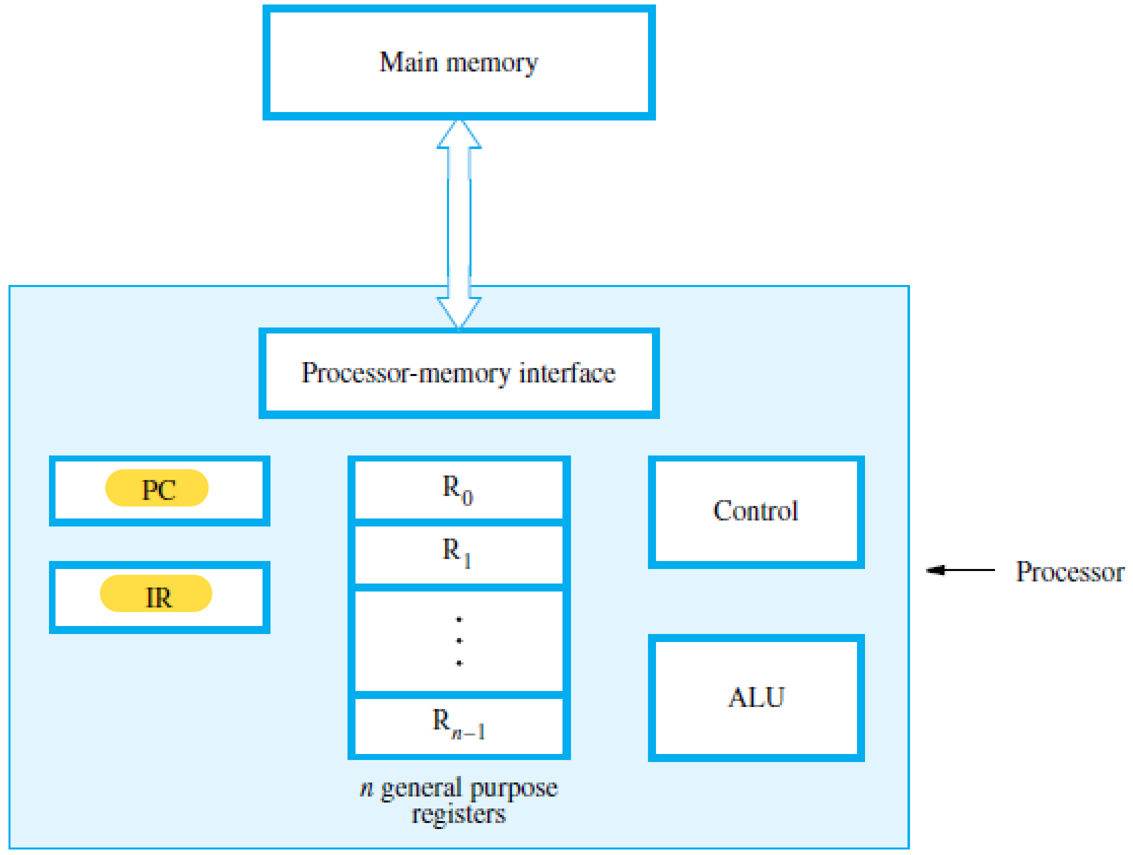


CONNECTION BETWEEN THE PROCESSOR AND THE MEMORY



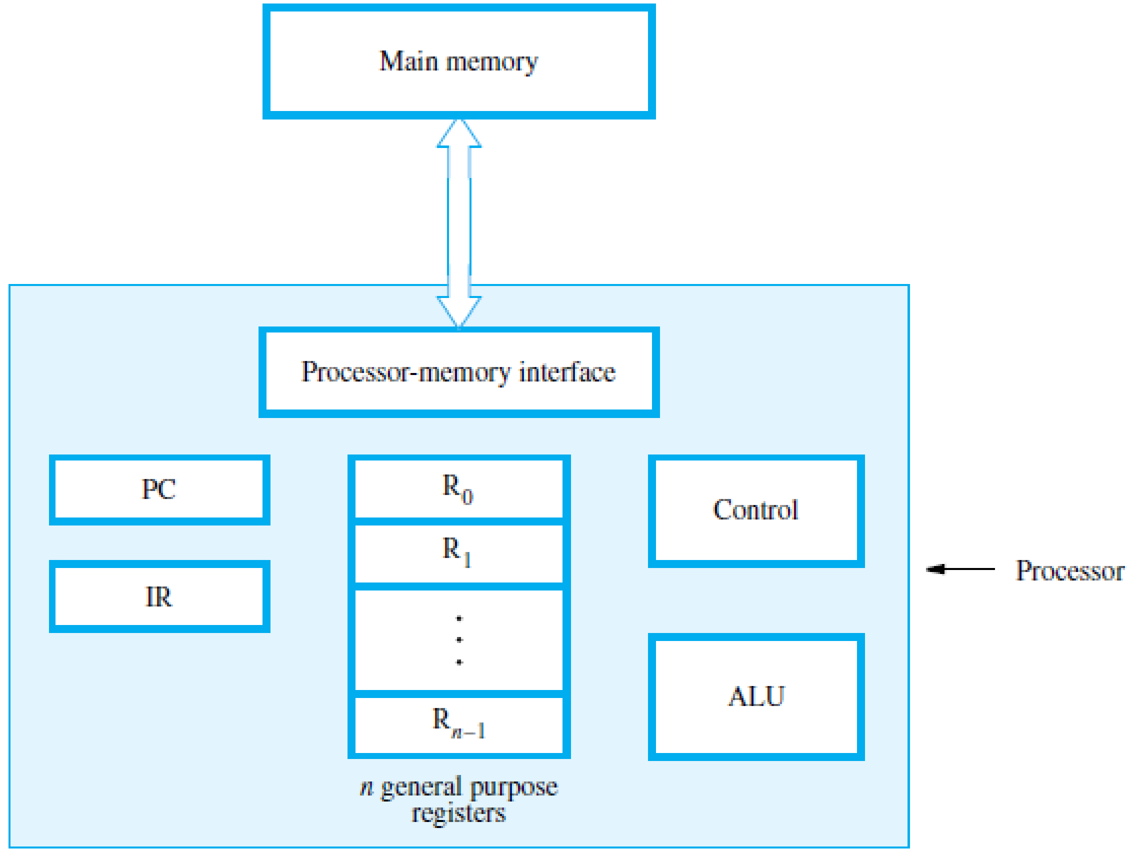
REGISTERS

- Instruction register (IR)
  - holds the instruction that is currently being executed. Its output is available to the control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction
- Program counter (PC)
  - points to the next instruction that is to be fetched from the memory
- General-purpose register
  - $R_0 – R_{n-1}$
  - Holds operands that have been loaded from the memory for processing



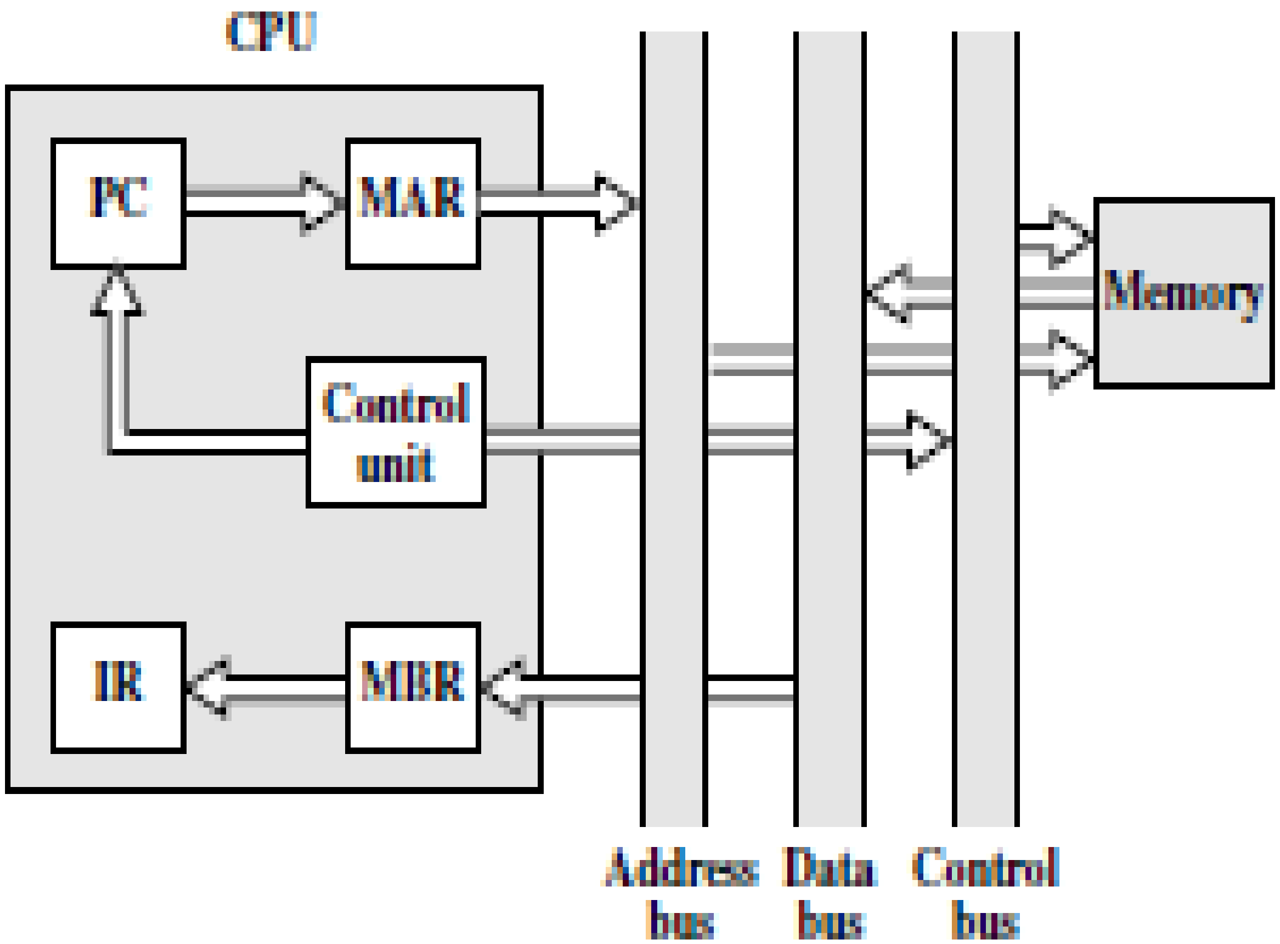
PROCESSOR-MEMORY INTERFACE

- It is a circuit which manages the transfer of data between the main memory and the processor.
- If a word is to be read from the memory, the interface sends the address of that word to the memory along with a Read control signal.
- The interface waits for the word to be retrieved, then transfers it to the appropriate processor register.
- If a word is to be written into memory, the interface transfers both the address and the word to the memory along with a Write control signal.
- How CPU works?
  - https://youtu.be/cNN_tTXABUA



TYPICAL OPERATING STEPS

- Programs to be executed must be in main memory
- PC is set to point to the first instruction
- The contents of PC are transferred to memory along with a Read control signal
- The first instruction read out is transferred to IR
- The instruction is ready to be decoded and executed
- Get operands for ALU
  - Memory
  - General-purpose register
- Perform operation in ALU
- Store the result back
  - To general-purpose register
  - To memory
- During the execution, PC is incremented to the next instruction



TYPICAL OPERATING STEPS (X=A+B)

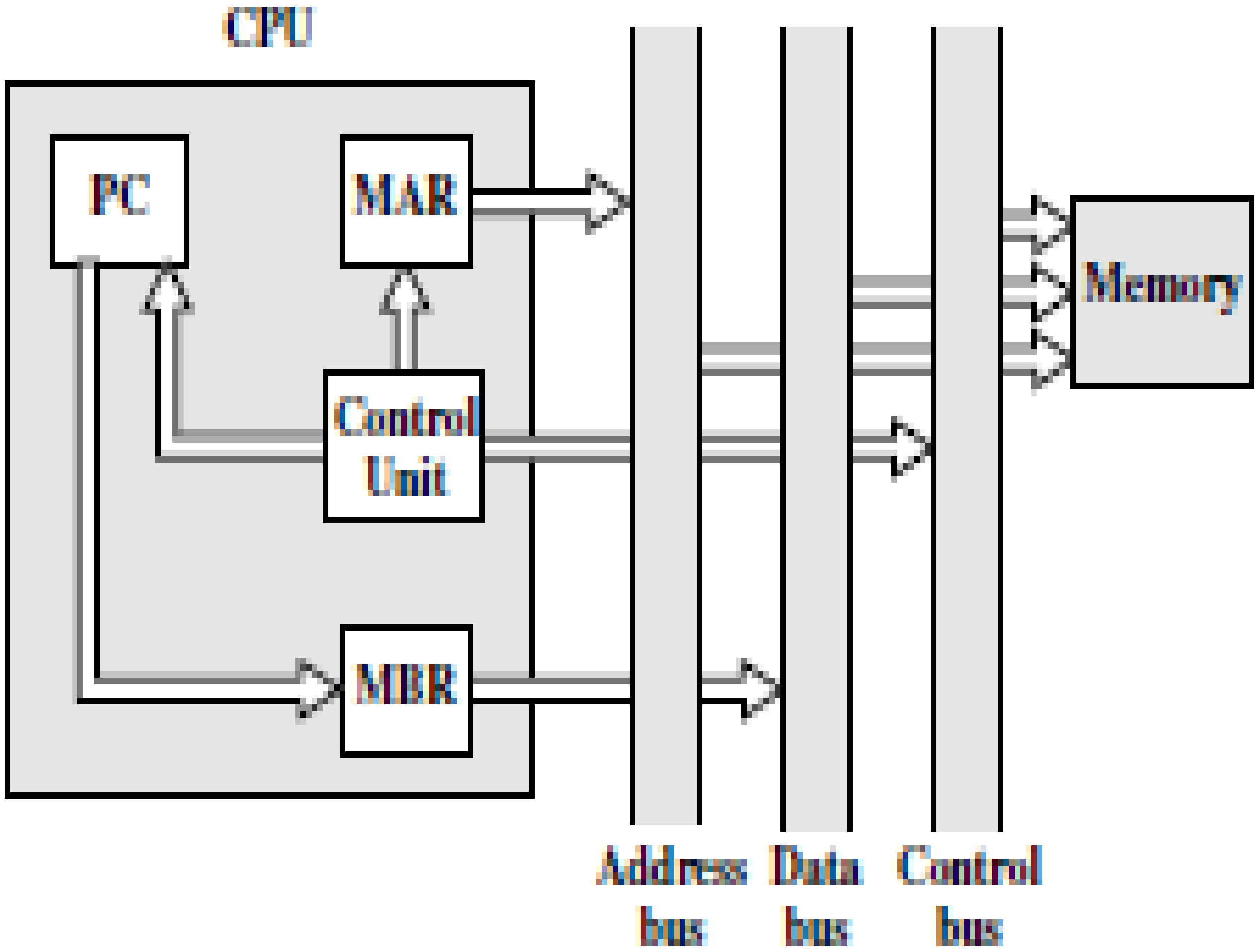
CPU Fetch/Decode/Execute cycle

Showing the calculation X= a + b

Representing a physical structure which is typical of a very simple processor

INTERRUPTS

- Normal execution of programs may be preempted if some device requires urgent servicing.
- The normal execution of the current program must be interrupted – the device raises an interrupt signal.
- Interrupt-service routine
- Current system information backup and restore (PC, general-purpose registers, control information, specific information)



EXERCISE

- List the steps needed to execute the machine instruction

  Load R2, LOC

- **Solution:**
  - Send the address of the instruction word from register PC to the memory and issue a Read control command.
  - Wait until the requested word has been retrieved from the memory
  - load it into register IR, where it is interpreted (decoded) by the control circuitry to determine the operation to be performed.
  - Increment the contents of register PC to point to the next instruction in memory.
  - Send the address value LOC from the instruction in register IR to the memory and issue a Read control command.
  - Wait until the requested word has been retrieved from the memory, then load it into register R2

EXERCISE

- Repeat for
  1. Add R4, R2, R3
  2. Store R4, LOC

EXERCISE

- 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
  - Following instructions are the only instructions available to transfer data between the memory and the general-purpose registers.
    - Load Ri, LOC
    - Store Ri, LOC
  - Do not change the contents of either location A or B.

~~GUESS~~....... ANSWER

- Internet is an example for input unit of a computer.
  A. **True**
  B. False

- The function of instruction “Load R1, LOC” is to read the data from R1 and store it in LOC. (This function is based on the notation used in this textbook.)
  A. True
  B. **False**

COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

EXERCISE

- 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
  - Following instructions are the only instructions available to transfer data between the memory and the general-purpose registers.
    - Load Ri, LOC
    - Store Ri, LOC
  - Do not change the contents of either location A or B.
- **Solution:**
  - Load R3, A
  - Load R4, B
  - Add R5, R3, R4
  - Store R5, C

GUESS……

1.
```
#include <iostream>
using namespace std;
int main() {
   unsigned short x=65535, y=65537;       //size of unsigned short is 2 bytes
   cout<<x<<" "<<y<<endl;
   return 0;
}
```

2. Find the difference of 10000 – 0101. (In 2's complement, this method is used to find the negative value of a given number by subtracting it from binary equivalent of 2^n).

NUMBER REPRESENTATION AND ARITHMETIC OPERATIONS

- Number representation
  - Decimal number system:
    - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    - Each digit has a position value in terms of powers of 10
    - $123 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0$
  - Binary number system
    - 0, 1
    - Each digit has a position value in terms of powers of 2
    - $101 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0$
    - n-bit vector $B = b_{n-1} \dots b_1 b_0$,
      - where $b_i = 0$ or 1 for $0 \leq i \leq n - 1$
    - unsigned integer value $V(B)$ in the range 0 to 2^{n-1}, where
      - $V(B) = b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$

INTEGER

- Unsigned integer
  - If the integers are represented using 4 bit
    - $0_{(10)}$ in binary?

      0000
    - $15_{(10)}$ in binary?

      1111 $= 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$
- Signed integer
  - Sign and magnitude
  - One's complement
  - Two's complement

  - In all three systems,
    - the **positive numbers** have the **same** bit representation
    - the **negative numbers** have **different** bit representation
    - positive numbers - the leftmost bit is 0
    - negative numbers- the leftmost bit is 1

SIGNED INTEGER: SIGN AND MAGNITUDE

- Negative values - most significant bit of the corresponding positive value changed from 0 to 1
- Sign- MSB
- Magnitude (or number)- remaining bits
- 0 represented as positive and negative

Positive Value	$b_3\ b_2b_1b_0$	$b_3\ b_2b_1b_0$	Negative Value
+7	0111	1111	-7
+6	0110	1110	-6
+5	0101	1101	-5
+4	0100	1100	-4
+3	0011	1011	-3
+2	0010	1010	-2
+1	0001	1001	-1
+0	0000	1000	-0

SIGNED INTEGER: ONE'S COMPLEMENT

- Negative values - complementing each bit in the corresponding positive value
- Negative to positive- complementing each bit in the corresponding negative value

  +6 0110

  -6 1001
- For n-bit numbers, this operation is equivalent to subtracting the number from $2^n - 1$.
- Example: +6 to -6 in a 4-bit representation
  - 2^n-1 =15 1111

  +6 0110

  -6 1001
- 0 represented as positive and negative

Positive Value	$b_3\ b_2 b_1 b_0$	$b_3\ b_2 b_1 b_0$	Negative Value
+7	0111	1000	-7
+6	0110	1001	-6
+5	0101	1010	-5
+4	0100	1011	-4
+3	0011	1100	-3
+2	0010	1101	-2
+1	0001	1110	-1
+0	0000	1111	-0

SIGNED INTEGER: TWO'S COMPLEMENT

- 2's-complement of an n-bit number is done by subtracting the number from 2^n.
- How to subtract 2 numbers?
- 0101 – 0100
  - 001
- 01100 – 01000
  - 0100
- 010000 – 0101
  - 01011
- 01000 – 0011
  - ?

First Value	Second Value	Difference
0	0	0
0	1	1 (by borrowing)
1	0	1
1	1	0

SIGNED INTEGER: TWO'S COMPLEMENT

- Negative values - adding 1 to the 1's-complement of corresponding positive value
- Example: +6 to -6 in a 4-bit representation
  - 2^n-1 =15 1111

    +6 0110 conversion using

    1001 1's complement

    +1 0001

    -6 **1010**
- For n-bit numbers, this operation is equivalent to subtracting the number from 2^n.
- Example: +6 to -6 in a 4-bit representation
  - 2^n 10000

    +6 00110

    -6 **01010**
- 0 represented as positive

Positive Value	$b_3\ b_2 b_1 b_0$	$b_3\ b_2 b_1 b_0$	Negative Value
+7	0111	1000	-8
+6	0110	1001	-7
+5	0101	1010	-6
+4	0100	1011	-5
+3	0011	1100	-4
+2	0010	1101	-3
+1	0001	1110	-2
+0	0000	1111	-1

SIGNED INTEGER: TWO'S COMPLEMENT

- 4 bits: -8 to +7

  -2^{4-1} to $+2^{4-1}-1$

- 5 bits: -16 to +15

  -2^{5-1} to $+2^{5-1}-1$

- 6 bits: -32 to +31

  -2^{6-1} to $+2^{6-1}-1$

- n bits: -2^{n-1} to $+2^{n-1}-1$

SIGNED INTEGERS

- For 4-bit numbers, the value −8 is representable in the 2's-complement system but not in the other systems.
- Sign-and-magnitude system seems the most natural
- 1's-complement system is easily related to this system
- 2's-complement appears unusual.
  - However, it leads to the most efficient way to carry out addition and subtraction operations.
  - It is the one most often used system in modern computers.

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

UNSIGNED INTEGERS: OPERATIONS

```
#include <iostream>
using namespace std;
int main() {
   unsigned short x=65535, y=65537;
   cout<<x<<" "<<y<<endl;
   return 0;
}
```

- x=65535, y=1
- unsigned short: 16 bits
  - 0 to 65535 (0 to 2^n-1)
    - If the value is out of range, it is divided by *largest_number+1* of that datatype, and only the remainder kept.
    - Here, 65537 % 65536= 1
    - Any number bigger than the largest number, "wraps around" the largest number in that type.

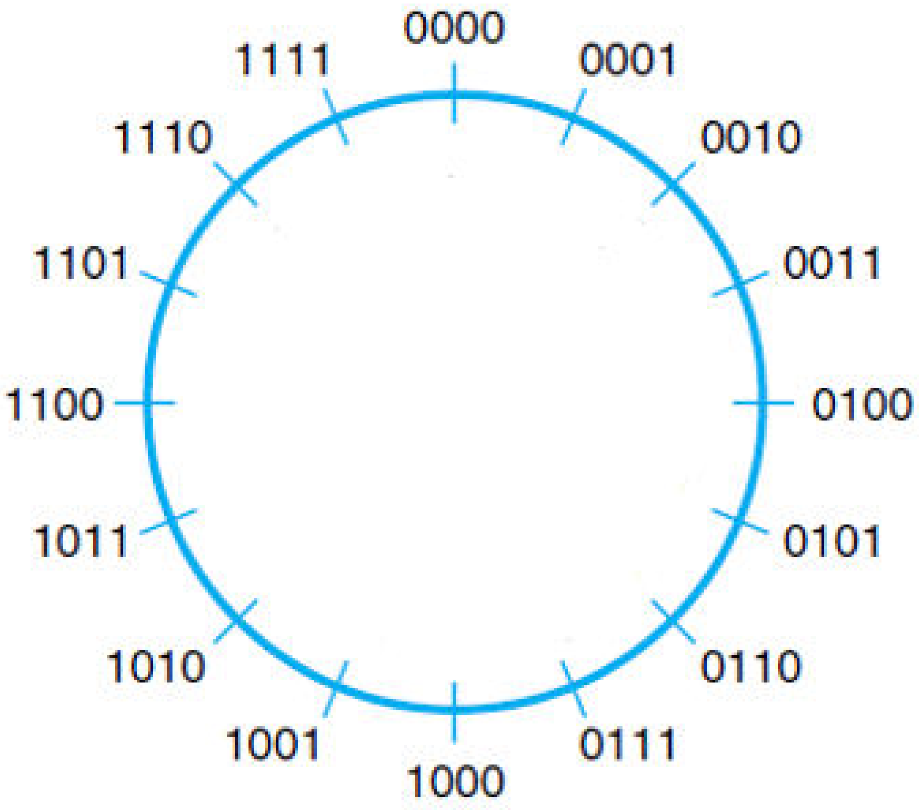
UNSIGNED INTEGERS: OPERATIONS

```
#include <iostream>
using namespace std;
int main() {
    unsigned short x=0, y=-1;
    cout<<x<<" "<<y<<endl;
    return 0;
}
```

- x=0, y= 65535
- wraps around to the top of the range.

UNSIGNED INTEGER: ADDITION

- **7** (0111) + **5** (0101) = ?
  - From 7 move 5 units in clockwise direction
  - 12
- 9 + 14=?
  - From 9 (1001) move 14 units in clockwise direction
  - 7



UNSIGNED INTEGERS: ADDITION

- 2-bit addition:

$$
\begin{array}{r} 0 \\ +\ 0 \\ \hline 0 \end{array}
\qquad
\begin{array}{r} 1 \\ +\ 0 \\ \hline 1 \end{array}
\qquad
\begin{array}{r} 0 \\ +\ 1 \\ \hline 1 \end{array}
\qquad
\begin{array}{r} 1 \\ +\ 1 \\ \hline 1\,0 \end{array}
$$

Carry-out

- Multiple bit addition
  - Similar to decimal addition method
  - add bit pairs starting from the low-order end of the bit vectors, transmitting the carries toward the high-order end
  - The carry-out from the previous bit pair becomes the carry-in to the current bit pair
  - The carry-out from the bit pair in the right must be added to the current bit pair to generate the sum and carry-out at that position
  - If both bits of a pair are 1 and the carry-in is 1, then the sum is 1 and the carry-out is 1

UNSIGNED INTEGERS: SUBTRACTION

- ```
  #include <iostream>
  using namespace std;
  int main(){
      unsigned int x=1, y=2;
      cout<<x-y;
      return 0;
  }
  ```
- 4294967295
  - This occurs due to -1 wrapping around to a number close to the top of the range of a 4-byte integer
  - $2^{32}-1 \rightarrow$ 4294967296-1
  - 32 bits = 4 bytes (size of int)
- Hence, subtraction is not recommended.
  ```

GUESS……… ANSWER

1.
```
#include <iostream>
using namespace std;
int main() {
   unsigned short x=65535, y=65537;        //size of unsigned short is 2 bytes
   cout<<x<<" "<<y<<endl;
   return 0;
}
```

Solution: x=65535, y=1

2. Find the difference of 10000 – 0101. (In 2's complement, this method is used to find the negative value of a given number by subtracting it from binary equivalent of 2^n).

Solution: 01011

TOPICS COVERED FROM

- Textbook 1:
  - Chapter 1: 1.4.1

TOPICS COVERED FROM

- Textbook 1:
  - Chapter 1: 1.1, 1.2, 1.3

COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

IN THE PREVIOUS CLASS……

- Binary number system
- Integers
  - Unsigned
    - Operations
    - Addition
    - Subtraction
  - Signed
    - Sign and magnitude
    - One’s complement
    - Two’s complement

EXERCISE

- 01000 – 0011
- **Solution:**
  - 00101

GUESS…….

1. Overflow occurs when carry-in to the high-order bit does not equal carry out
   A. True
   B. False

2. ____________ integer representation is used in most of the modern computers.
   A. Sign and magnitude
   B. 1's complement
   C. 2's complement

SIGN AND MAGNITUDE: ARITHMETIC OPERATIONS

- Add +5 and -2 = +3 (4-bit representation)

  0 101 +

  <u>1 010</u>

  111 (7) wrong answer!!

- Subtract -6 and 1 = -7 (4-bit representation)

  1 110 -

  <u>0 001</u>

  101 (5) wrong answer!!

- Hence, not suitable for arithmetic operations

1'S COMPLEMENT: ARITHMETIC OPERATION

- Add +1 and -1 = 0 (4-bit representation)

  0 001 +

  1 110

  111 (wrong answer!!)

  \+ 1

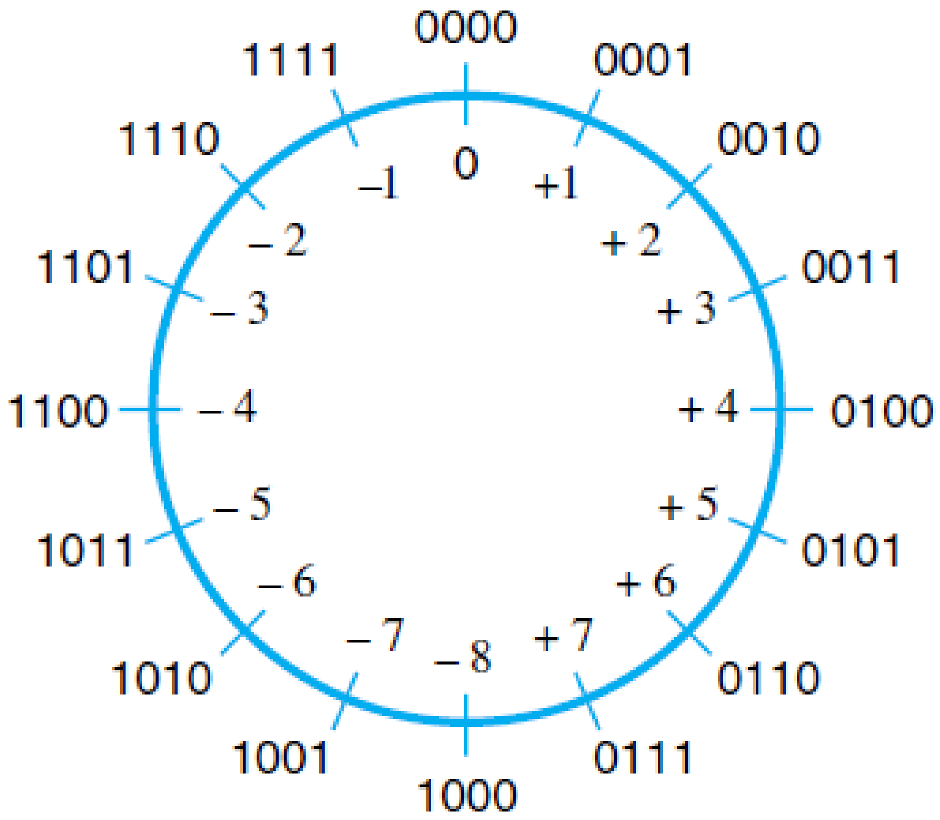
  000 (correct answer)

- Additional step (hardware) required to arrive at the correct answer.
- Hence, not suitable for arithmetic operations

2'S COMPLEMENT: ARITHMETIC OPERATION (ADDITION)

- Add +7 and -3
  - +7 is 0111 and -3 is 1101,
  - Locate 0111 in the diagram and move 1101 (13) steps in clockwise
  - Solution: 0100
  - 2's-complement representation of −3 is interpreted as an unsigned value for the number of steps to move.
- Adding bit pair wise:
  - ignore the carry-out

$$
\begin{array}{cccccc}
 & & 0 & 1 & 1 & 1 \\
+ & & 1 & 1 & 0 & 1 \\
\hline
 & 1 & 0 & 1 & 0 & 0 \\
 & \uparrow & & & & \\
 & \text{Carry-out} & & & &
\end{array}
$$



2'S COMPLEMENT: ADDITION

- +2 + (+3)

+2→	0010 +
+3→	0011
+5→	0101

- +4 + (-6)

+4→	0100 +
-6→	1010
-2→	1110

- -5 + (-2)

-5→	1011 +
-2→	1110
-7→	1001

Carry is ignored

- +7 + (-3)

+7→	0111 +
-3→	1101
+4→	0100

Carry is ignored

$b_3b_2b_1b_0$	Value in decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

2'S COMPLEMENT: ADDITION AND SUBTRACTION

- To add two numbers (X+Y),
  - Represent X and Y in binary format (2's complement representation)
  - add their n-bit representations,
  - ignore the carry-out bit from the most significant bit (MSB) position.
  - The sum will be the 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 (X − Y),
  - Represent X and Y in binary format (2's complement representation)
  - form the 2's-complement of Y,
  - add it to X using the add rule.
  - The result will be the algebraically correct value in 2's-complement representation if the actual result is in the range -2^{n-1} through $+2^{n-1} - 1$.

2'S COMPLEMENT: SUBTRACTION

- -7- (-5)

-7→	1001 -		-7→	1001 +
-5→	1011	Becomes:	5→	0101
			-2→	1110

- -7- (+1)

-7→	1001 -		-7→	1001 +
+1→	0001	Becomes:	-1→	1111
			-8→	1000

- +2- (-3)

+2→	0010 -		+2→	0010 +
-3→	~~1011~~ 1101	Becomes:	+3→	0011
			+5→	0101

$b_3b_2b_1b_0$	Value in decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

2'S COMPLEMENT: SUBTRACTION

- -3- (-7)

-3→	1101 -		-3→	1101 +
-7→	1001	Becomes:	7→	0111
			+4→	0100

- +2- (+4)

+2→	0010 -		+2→	0010 +
+4→	0100	Becomes:	-4→	1100
			-2→	1110

- +6- (+3)

+6→	0110 -		+6→	0110 +
+3→	0011	Becomes:	-3→	1101
			+3→	0011

$b_3b_2b_1b_0$	Value in decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

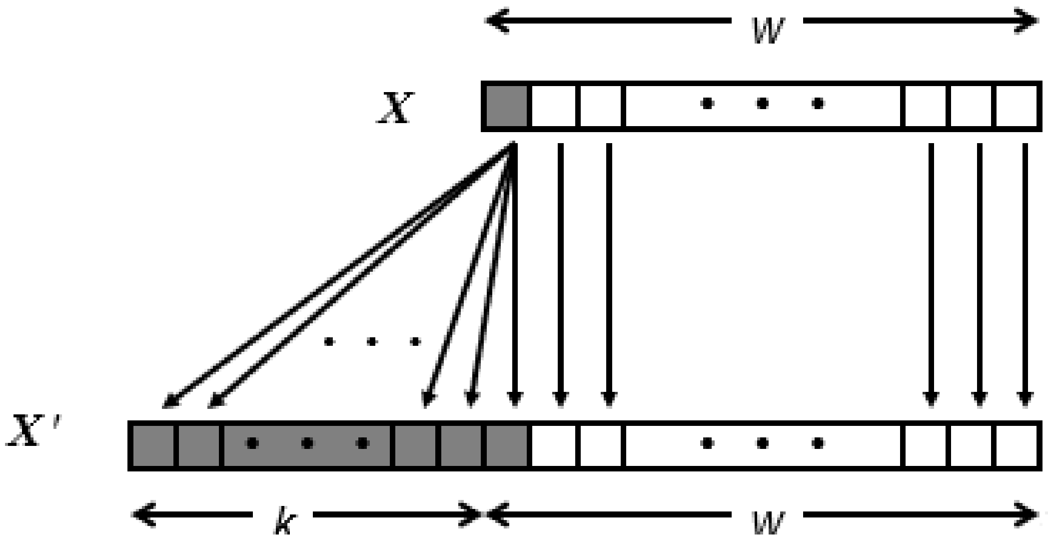
SIGNED INTEGERS: ADDITION AND SUBTRACTION

- Sign and Magnitude:
  - Undesired results
- 1's Complement:
  - The results are not always correct
- 2's Complement:
  - simplicity of adding and subtracting signed numbers
  - used in modern computers

SIGN EXTENSION

- Represent a value given in a certain number of bits by using a larger number of bits
- Positive numbers: zeroes are added to the left
- Negative numbers: ones are added to the left

- To convert a given (w)-bit, signed integer x to (w+k)-bit integer with same value
  - Make k copies of sign bit:
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$



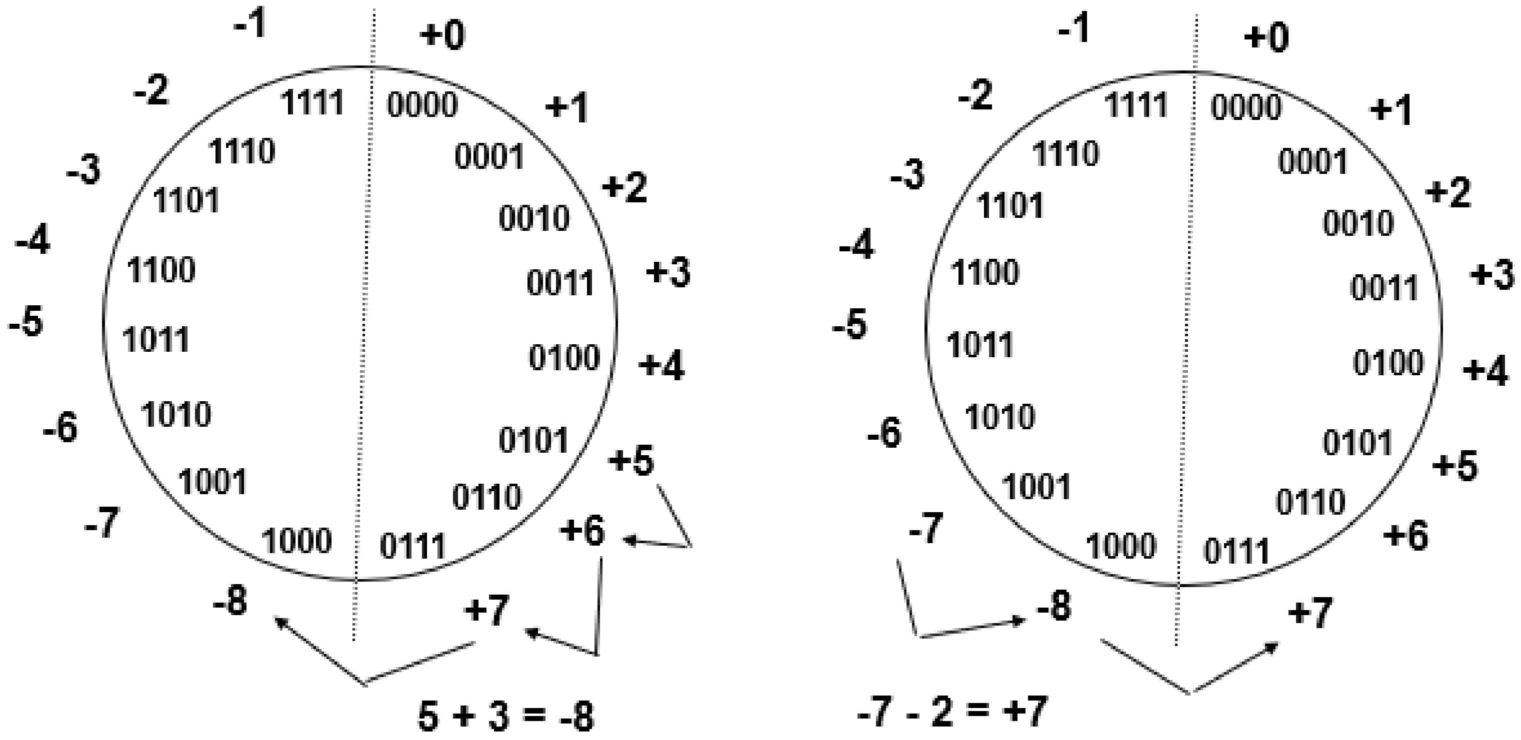
SIGN EXTENSION: EXAMPLE

```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Binary
x	15213	00111011 01101101
ix	15213	00000000 00000000 00111011 01101101
y	-15213	11000100 10010011
iy	-15213	11111111 11111111 11000100 10010011

OVERFLOW IN INTEGER ARITHMETIC

- Arithmetic overflow:
  - The actual result of an arithmetic operation is outside the representable range
- Overflow occurs when
  - two positive numbers are added which results in a negative number or
  - two negative numbers are added which results in a positive number



OVERFLOW IN INTEGER ARITHMETIC

- +7 + (+4)

```
carry:   0100
+7→       0111  +
+4→       0100
-5→       1011
```

- -4 + (-6)

```
carry:   1000
-4→       1100  +
-6→       1010
+6→       0110
```

- The value of the carry-out bit from the sign-bit position is not an indicator of overflow
- Overflow occurs when carry-in to the high-order bit does not equal carry out

$b_3b_2b_1b_0$	Value in decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

OVERFLOW IN INTEGER ARITHMETIC

- +5 + (+3)

```
carry:  0111
+5→     0101 +
+3→     0011
-8→     1000
```

Overflow

- -7 + (-2)

```
carry:  1000
-7→     1001 +
-2→     1110
-8→     0111
```

Overflow

- +5 + (+2)

```
carry:  0000
+5→     0101 +
+2→     0010
+7→     0111
```

No overflow

- -3 + (-5)

```
carry:  1111
-3→     1101 +
-5→     1011
-8→     1000
```

No overflow

$b_3b_2b_1b_0$	Value in decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

CHARACTER REPRESENTATION

- The most common encoding scheme for characters is ASCII
- Alphanumeric characters, operators, punctuation symbols, and control characters represented using 7-bit codes
- 8-bit byte is used to represent and store a character
- The code occupies the low-order seven bits
- The high-order bit is usually set to 0

FLOATING-POINT NUMBERS

- The basic IEEE format is a 32-bit representation that comprises of
  - a sign bit,
  - 23 significant bits, and
  - 8 bits for a signed exponent of the scale factor
- IEEE standard also defines a 64-bit representation to accommodate
  - more significant bits, and
  - more bits for the signed exponent, resulting in much higher precision and a much larger range of values
- In general, a binary floating-point number can be represented by (2008 version of IEEE Standard 754):
  - a sign for the number
  - some significant bits
  - a signed scale factor exponent for an implied base of 2

~~GUESS~~……. ANSWER

1. Overflow occurs when carry-in to the high-order bit does not equal carry out
   A. **True**
   B. False

2. ____________ integer representation is used in most of the modern computers.
   A. Sign and magnitude
   B. 1's complement
   C. **2's complement**

COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

IN THE PREVIOUS CLASS……

- Floating Point IEEE754 representation
  - 32-bit representation
    - Decimal to 32-bit
    - 32-bit to decimal
    - Special values and Exceptions
    - Addition
    - Subtraction
  - 64-bit representation

EXERCISE

1. Represent each of the given base ten numbers in IEEE754 format.
   A. 5.1

      Ans: 0 10000001 01000110011001100110011

   B. -14.25

      Ans: 1 10000010 11001000000000000000000

   C. 6.425

      Ans: 0 10000001 10011011001100110011001

GUESS…….

1. In the three additional bits in mantissa before truncation, the guard bit will contain the result obtained by OR operation on remaining lower order bits.
   A. True
   B. False

2. Big-Endian: lower byte addresses are used for the most significant bytes of the word.
   A. True
   B. False

ADDITION EXAMPLE- BINARY

- A=15.5 + B=15.5
- **Step i:** Convert A to binary representation
  - 1111.1 ➔After normalizing we get 1.1111 X 2^3
  - E'= 3+127=130 =10000010
  - In IEEE 32-bit format: A= 01000001011110000000000000000000
- **Step ii:** Convert B to binary representation
  - 1111.1 ➔After normalizing we get 1.1111 X 2^3
  - E'= 3+127=130 =10000010
  - In IEEE 32-bit format: B= 01000001011110000000000000000000
- **Step 1:** Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
  - Both are of equal exponent. Hence no shift
- **Step 2:**
  - Set the exponent of the result equal to the larger exponent
    - 10000010 (exponent of A or B)

ADDITION EXAMPLE- BINARY

- **Step 3:**
  - Perform addition on the mantissas and determine the sign of the result

```
   1.11110000000000000000000  +
   1.11110000000000000000000
  ---------------------------
  11.11100000000000000000000
```

- **Step 4:**
  - Normalize the resulting value, if necessary
    - $1.11110000000000000000000 \times 2^{1}$
  - Adjust resultant exponent E' by adding exponent from the normalized resultant mantissa

```
   10000010  +        (if the exponent is –ve, subtract it from resultant E')
   00000001
  ---------
   10000011
```

- In 32-bit format: 0 10000011 11110000000000000000000
- which is 31.0 in decimal

MULTIPLY AND DIVIDE RULE

- Multiply Rule
  A. Add the exponents and subtract 127 to maintain the excess-127 representation.
  B. Multiply the mantissas and determine the sign of the result.
  C. Normalize the resulting value, if necessary.

- Divide Rule
  A. Subtract the exponents and add 127 to maintain the excess-127 representation.
  B. Divide the mantissas and determine the sign of the result.
  C. Normalize the resulting value, if necessary.

MULTIPLICATION EXAMPLE- BINARY

- A= 0 10000100 0100 × B= 1 00111100 1100

- **Step 1:** add exponents and subtract 127
  - 132+60-127=65. and unsigned representation for 65 is 01000001.
- **Step 2:** Multiply the mantissa. Don't forget hidden bit

```
          1.0100 x 1.1100
           00000
          00000
         10100
        10100
       10100
    1000110000        becomes   10.00110000
```

normalize the result:

1.000110000 X 2^1

Step 3: Adjust the exponent : 65+1=66 = 01000010

1 01000010 000110000

MULTIPLICATION EXAMPLE- BINARY

- A=96.625 × B=12.125
- **Step i:** Convert A to binary representation
  - 1100000.101 ➔After normalizing 1.100000101 X 2^6
  - E'= 6+127=133 =10000101
  - In IEEE 32-bit format: A= 0 10000101 10000010100000…….
- **Step ii:** Convert B to binary representation
  - 1100.001 ➔After normalizing 1.100001 X 2^3
  - E'= 3+127=130 = 10000010
  - In IEEE 32-bit format: B= 0 10000010 10000100000……
- **Step 1:** add exponents and subtract 127
  - 133+130-127=136 ➔ 10001000 (unsigned representation)
- **Step 2:** Multiply the mantissa

1.100000101 x 1.100001= 10.010010011100101

After normalizing 1.0010010011100101 X 2^1

```
       1.100000101 x 1.100001
        1100000101
       0000000000
      0000000000
     0000000000
    0000000000
   1100000101
  1100000101
 10010010011100101
```

➔10.010010011100101

- **Step 3:** Adjust the exponent:
  - 136+1=137➔ 10001001
- 32-bit representation:
  - 0 10001001 0010010011100101
  - = 1.0010010011100101 X 2^{10}
  - =10010010011.100101
  - =1024+128+16+2+1+ 0.5+.0625+.015625
  - =1171.578125

DIVISION EXAMPLE- BINARY

- A= 127.03125 ÷ B= 16.9375
- **Step i:** Convert A to binary representation
  - 1111111.00001 ➔After normalizing 1.11111100001 X 2^6
  - E'= 6+127=133 =10000101
  - In IEEE 32-bit format: A= 0 10000101 11111100001……
- **Step ii:** Convert B to binary representation
  - 10000.1111 ➔After normalizing 1.00001111 X 2^4
  - E'= 4+127=131 = 10000011
  - In IEEE 32-bit format: B= 0 10000011 00001111000……
- **Step 1:** subtract exponents and add 127
  - 133-131+127=129 ➔ 10000001 (unsigned representation)
- **Step 2:** Divide the mantissa

1.11111100001 ÷ 1.00001111= 1.111

The result is already normalized.

- The result in IEEE 32-bit format:

  0 10000001 11100000000…….

```
              1.1 1 1
1.00001111 | 1.1 1 1 1 1 1 0 0 0 0 1
             1.0 0 0 0 1 1 1 1
             0 1 1 1 0 1 1 0 1 0
               1 0 0 0 0 1 1 1 1
               0 1 1 0 0 1 0 1 1 0
                 1 0 0 0 0 1 1 1 1
                 0 1 0 0 0 0 1 1 1 1
                   1 0 0 0 0 1 1 1 1
                   0 0 0 0 0 0 0 0 0
```

DIVISION EXAMPLE- BINARY

- A= 97.0 ÷ B= 12.125
- **Step i:** Convert A to binary representation
  - 1100001.0 ➔After normalizing 1.100001 X 2^6
  - E'= 6+127=133 =10000101
  - In IEEE 32-bit format: A= 0 10000101 10000100……
- **Step ii:** Convert B to binary representation
  - 1100.001 ➔After normalizing 1.100001 X 2^3
  - E'= 3+127=130 = 10000010
  - In IEEE 32-bit format: B= 0 10000010 10000100000……
- **Step 1:** subtract exponents and add 127
  - 133-130+127=130 ➔ 10000010 (unsigned representation)
- **Step 2:** Divide the mantissa: 1.100001 ÷ 1.100001= 1.0

The result is already normalized.

- The result in IEEE 32-bit format:

  0 10000010 00000000000…….

i.e.,1.0 X2^3 = 1000 =8 in decimal

```
          1.0
1.100001 | 1.100001
           1.100001
           --------
            0000000
```

GUARD BITS AND TRUNCATION

- Mantissas of initial operands and final results are limited to 24 bits, including the implicit leading 1
- To attain maximum accuracy in the final results it is important to retain extra bits, often called guard bits, during the intermediate steps.
- Removing guard bits in generating a final result requires that the extended mantissa be truncated to create a 24-bit number that approximates the longer version.

WAYS OF TRUNCATION: CHOPPING

- Remove the guard bits and make no changes in the retained bits
  - e.g. To truncate from 6 bits to 3 bits:
    - $0.b_{-1}b_{-2}b_{-3}000$ to $0.b_{-1}b_{-2}b_{-3}111$ are truncated to $0.b_{-1}b_{-2}b_{-3}$
- The error in the 3-bit result ranges from 0 to 0.000111
- The error in chopping, ranges from 0 to almost 1 in the least significant position of the retained bits.
- In the example above, it is b_{-3} position. The result of chopping is a *biased approximation* because the error range is not symmetrical about 0

WAYS OF TRUNCATION: VON NEUMANN ROUNDING

- If the bits to be removed are all 0s,
  - they are simply dropped, with no changes to the retained bits.
- If any of the bits to be removed are 1,
  - the least significant bit of the retained bits is set to 1
- All 6-bit fractions where $b_{-4}b_{-5}b_{-6} \neq 000$ are truncated to $0.b_{-1}b_{-2}1$
- The error in this truncation method ranges between -1 and $+1$ in the LSB position of the retained bits
- approximation is unbiased because the error range is symmetrical about 0.
- When three guard bits are used, the value 0.001100 is truncated to 0.001

WAYS OF TRUNCATION: VON NEUMANN ROUNDING

- 0.001 00000 → 0.001 (truncate. 0 error)
- 0.001 11111 → 0.001 (but the value is near 0.010, hence -1 error at LSB position of retaining bits)
- 0.010 11111 → 0.011 (almost nearest value. Almost 0 error)
- 0.010 00001 → 0.011 (+1 error)

WAYS OF TRUNCATION: ROUNDING

- Achieves the closest approximation to the number being truncated and is an unbiased technique.
- A 1 is added to the LSB position of the bits to be retained if there is a 1 in the MSB position of the bits being removed. Thus,
  - $0.b_{-1}b_{-2}b_{-3}1\ldots$ is rounded to $0.b_{-1}b_{-2}b_{-3} + 0.001$
  - $0.b_{-1}b_{-2}b_{-3}0\ldots$ is rounded to $0.b_{-1}b_{-2}b_{-3}$.
- Except for the case in which the bits to be removed are $10\ldots0$.
  - This is a tie situation; the longer value is halfway between the two closest truncated representations.
- To break the tie
  - choose the retained bits to be the nearest even number
  - the value $0.b_{-1}b_{-2}0100$ is truncated to $0.b_{-1}b_{-2}0$
  - the value $0.b_{-1}b_{-2}1100$ is truncated to $0.b_{-1}b_{-2}1 + 0.001$.
- Also termed as "round to the nearest number or nearest even number in case of a tie"
- The error range is approximately $-1/2$ to $+1/2$ in the LSB position of the retained bits.

WAYS OF TRUNCATION: ROUNDING

- Best method
- But most difficult to implement because it requires an addition operation and a possible renormalization.
- This rounding technique is the default mode for truncation specified in the IEEE floating-point standard
- When three guard bits are used, using Rounding procedure, the value 0.001100 is truncated to 0.010
- Similarly,
  i. 0.111 011=0.111
  ii. 0.110 011=0.110
  iii. 0.111 101=0.111+0.001=1.000
  iv. 0.111 100=0.111+0.001=1.000
  v. 0.110 100=0.110
  vi. 0.101 100=0.101+0.001=0.110

IMPLEMENTING ROUNDING

- Requires only three guard bits to be carried along during the intermediate steps in performing an operation.
- The first two of these bits are the two most significant bits of the section of the mantissa to be removed.
- The third bit is the logical OR of all bits beyond these first two bits in the full representation of the mantissa.
- It should be initialized to 0.
- If a 1 is shifted out through this position while aligning mantissas, the bit becomes 1 and retains that value; hence, it is usually called the sticky bit.

INSTRUCTION SET ARCHITECTURE

- Machine instructions and program execution
- Addressing methods for accessing register and memory operands

MODULE 2

MEMORY LOCATIONS, ADDRESSES, AND OPERATIONS

- Memory consists of many millions of storage cells, each of which can store 1 bit.
- Data is usually accessed in n-bit groups. n is called word length.

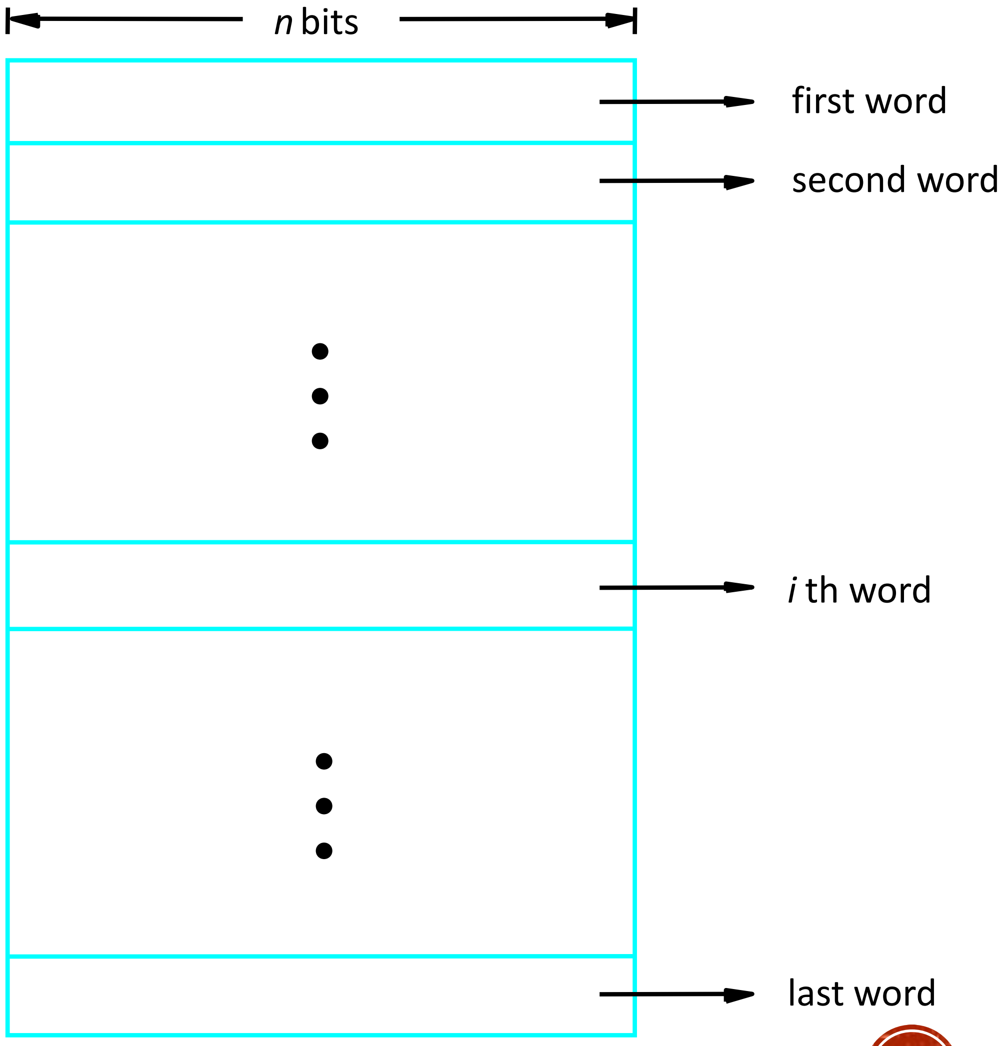
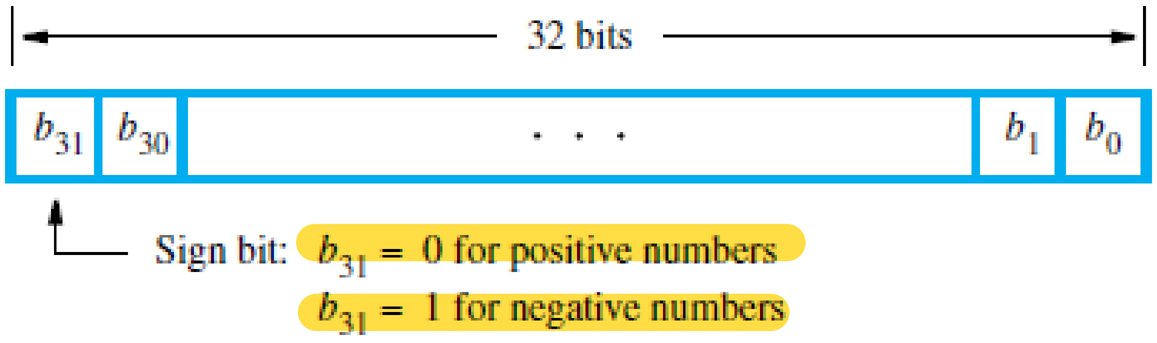


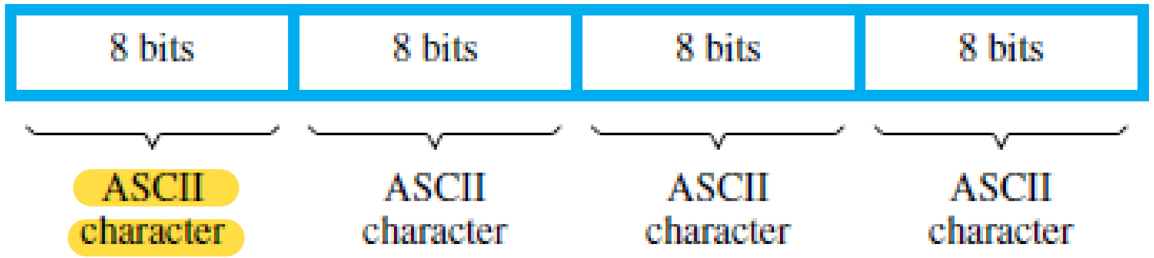
Figure 2.5. Memory words.

MEMORY LOCATIONS, ADDRESSES, AND OPERATIONS

- 32-bit word length example
- To retrieve information from memory, either for one word or one byte (8-bit), addresses for each location are needed.
- A k-bit address memory has 2^k memory locations, namely 0 - 2^k-1, called memory space.
- 24-bit memory: 2^{24} = 16,777,216 = 16M (1M=2^{20})
- 32-bit memory: 2^{32} = 4G (1G=2^{30})
- 1K(kilo)=2^{10}
- 1T(tera)=2^{40}



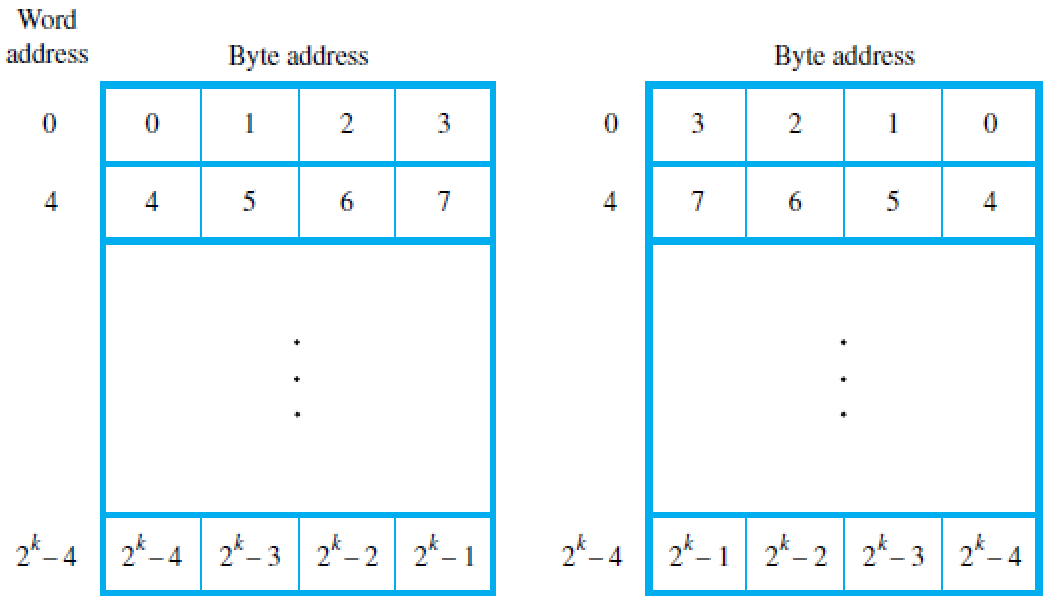
(a) A signed integer



(b) Four characters

BYTE ADDRESSABILITY

- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory – byte-addressable memory.
- Byte locations have addresses 0, 1, 2, … If word length is 32 bits, the successive words are located at addresses 0, 4, 8,…
- two ways that byte addresses can be assigned across words
  - 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



(a) Big-endian assignment

(b) Little-endian assignment

~~GUESS~~........ ANSWER

1. In the three additional bits in mantissa before truncation, the guard bit will contain the result obtained by OR operation on remaining lower order bits.
   A. True
   B. False

2. Big-Endian: lower byte addresses are used for the most significant bytes of the word.
   A. True
   B. False

EXERCISE

1. Take any two numbers and
   i. Convert it to floating point IEEE 754 representation
   ii. Perform addition, subtraction, multiplication and division operations
   iii. Obtain the results equivalent decimal value
   iv. Check whether obtained result is correct

TOPICS COVERED FROM

- Textbook 1:
  - Chapter 9: 9.7.1, 9.7.2
  - Chapter 2: 2.1

TOPICS COVERED FROM

- Textbook 1:
  - Chapter 1: 1.4.1, 1.4.2, 1.5

COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

IN THE PREVIOUS CLASS……

- Byte Addressability
- Word Alignment
- Accessing Numbers and Characters
- Memory Operations
- Instruction and Instruction Sequencing
- Register Transfer Notation
- Assembly Language Notation
- RISC and CISC Instruction Set: Characteristics RISC Instruction Set
- Instruction Execution and Straight-Line Sequencing

GUESS.......

1. In a conditional branch instruction, if the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.
   A. True
   B. False

2. # in front of the value indicates that this value is to be used as an immediate operand.
   A. True
   B. False

INSTRUCTION EXECUTION AND STRAIGHT-LINE SEQUENCING

- a possible program segment for the task, C=A+B, as it appears in the memory of a computer.
- Straight-line sequencing: using the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses.
- Executing a given instruction is a two-phase procedure.
  - Instruction fetch (First phase)
  - Instruction execute (Second phase)

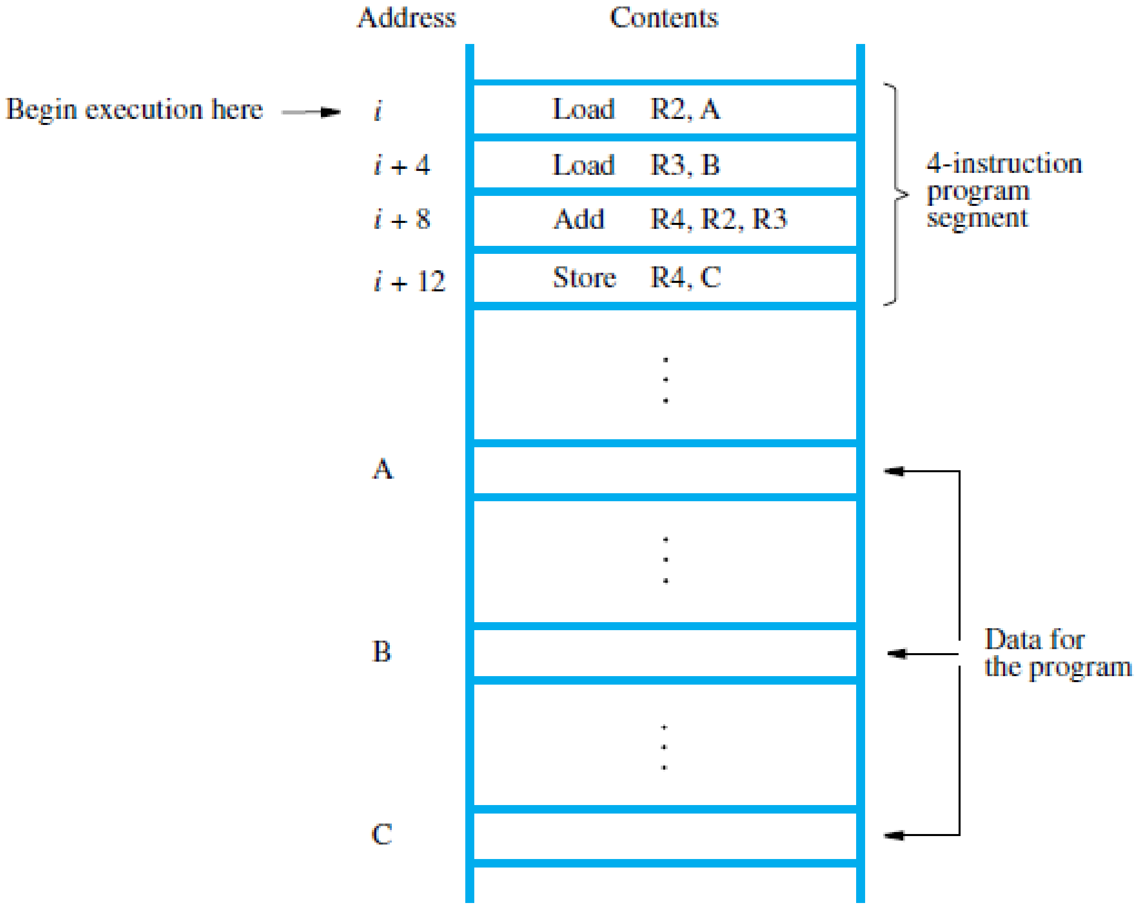


Figure 2.4 A program for C ← [A] + [B].

BRANCHING

- Consider the task of adding a list of n numbers.

i	Load	R2, NUM1
$i + 4$	Load	R3, NUM2
$i + 8$	Add	R2, R2, R3
$i + 12$	Load	R3, NUM3
$i + 16$	Add	R2, R2, R3
	⋮	
$i + 8n - 12$	Load	R3, NUMn
$i + 8n - 8$	Add	R2, R2, R3
$i + 8n - 4$	Store	R2, SUM
	⋮	
SUM		
NUM1		
NUM2		
	⋮	
NUMn		

Figure 2.5 A program for adding n numbers.

BRANCHING

- Consider the task of adding a list of n numbers in a loop.

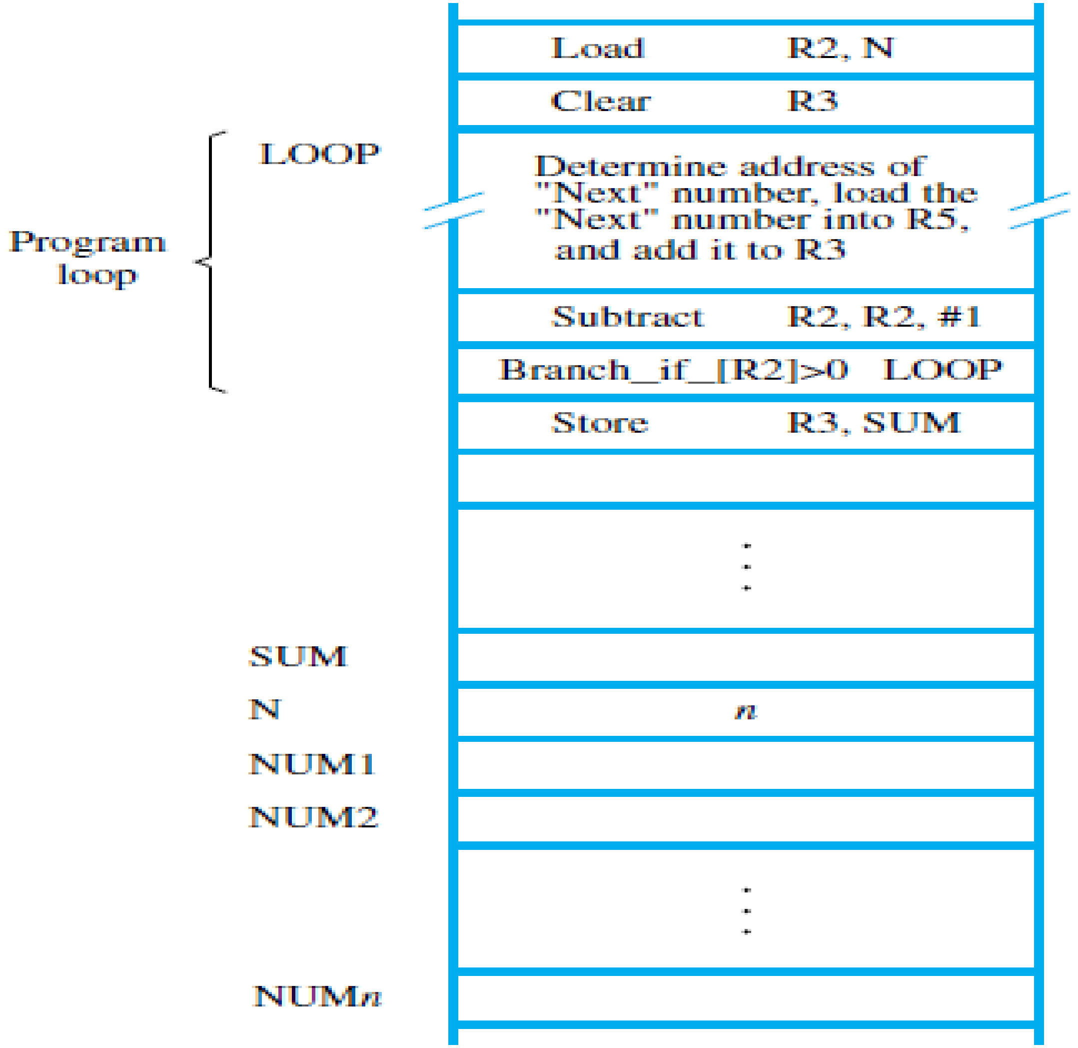


Figure 2.6 Using a loop to add n numbers.

BRANCHING

- Branch_if_[R4]>[R5] LOOP
  - In generic assembly language as: Branch_greater_than R4, R5, LOOP
  - Using an actual mnemonic as: BGT R4, R5, LOOP

GENERATING MEMORY ADDRESSES

- The purpose of the instruction block starting at LOOP is to add successive numbers from the list during each pass through the loop
- must refer to a different address during each pass
- memory operand address cannot be given directly in a single Load instruction in the loop.

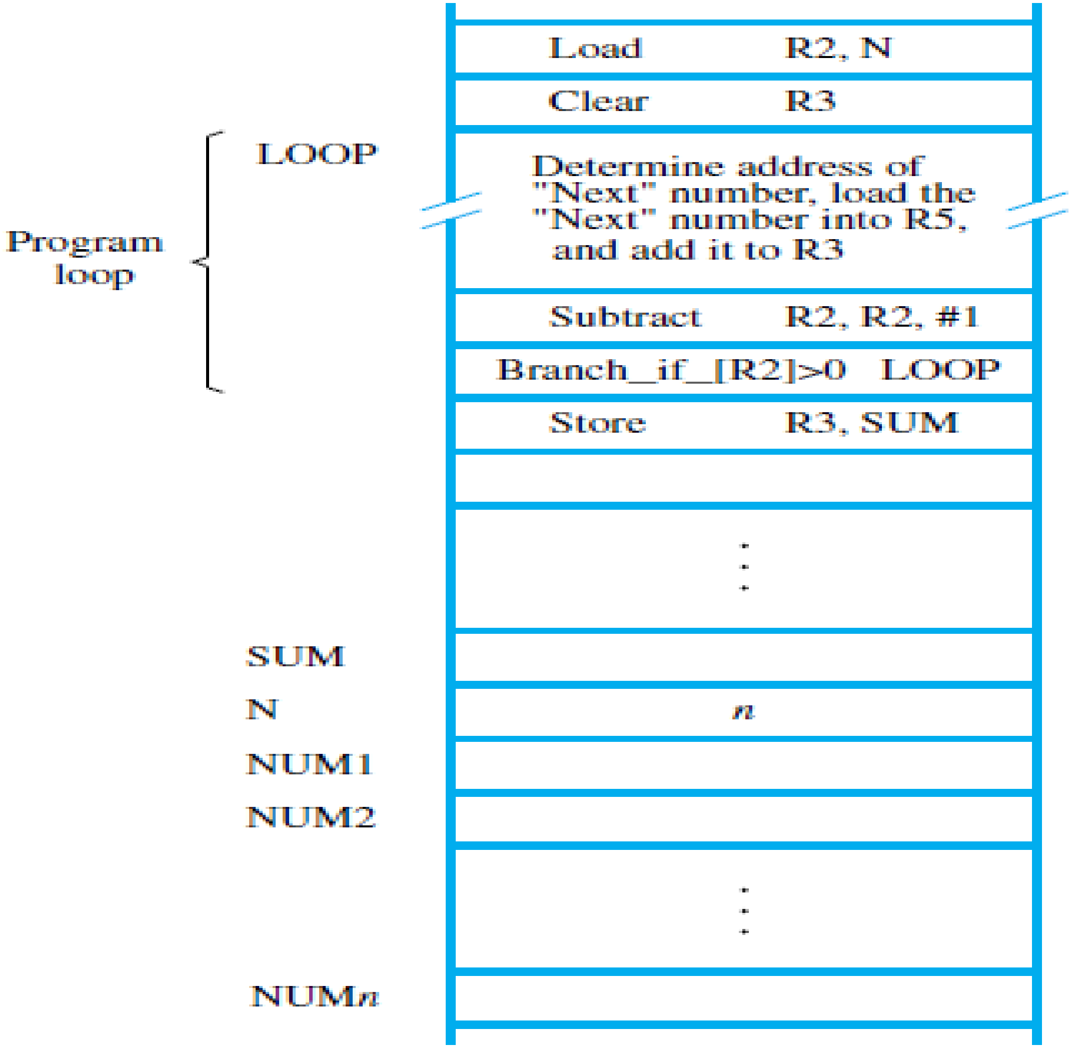


Figure 2.6 Using a loop to add n numbers.

ADDRESSING MODES

- Addressing modes:
  - Different ways for specifying the locations of instruction operands.
- RISC-style processors basic addressing modes Table 2.1
- The assembler syntax defines the way in which instructions and the addressing modes of their operands are specified

Table 2.1 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

IMPLEMENTATION OF VARIABLES AND CONSTANTS

- Register mode
  - The operand is the contents of a processor register; the name of the register is given in the instruction.
  - Example: Add R4, R2, R3
- Absolute mode
  - The operand is in a memory location; the address of this location is given explicitly in the instruction.
  - Load R2, NUM1
- Immediate mode
  - The operand is given explicitly in the instruction.
  - Add R4, R6, $200_{\text{immediate}}$
  - Add R4, R6, #200

INDIRECTION AND POINTERS

- Indirect mode:
  - The effective address of the operand is the contents of a register that is specified in the instruction
  - Load R2, (R5)

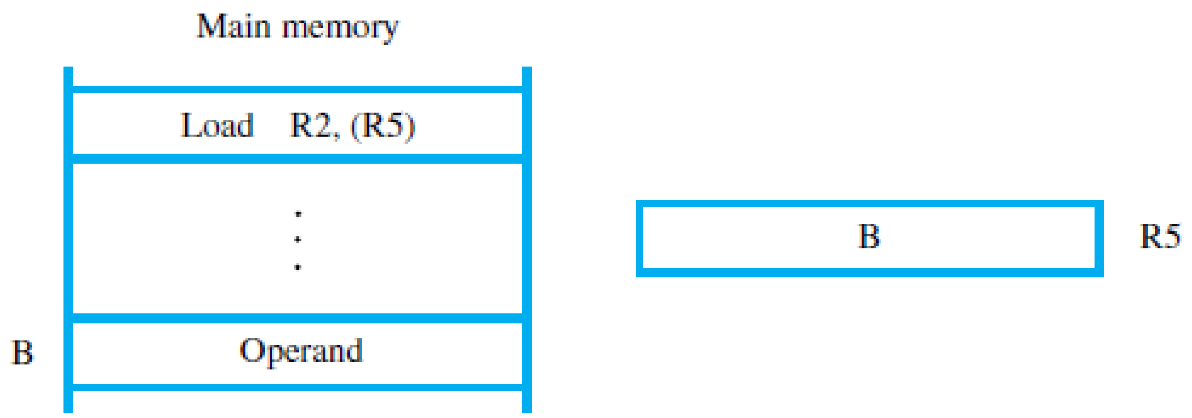


Figure 2.7 Register indirect addressing.

~~GUESS.......~~ ANSWER

1. In a conditional branch instruction, if the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.
   A. **True**
   B. False

2. # in front of the value indicates that this value is to be used as an immediate operand.
   A. **True**
   B. False

TOPICS COVERED FROM

- Textbook 1:
  - Chapter 2: 2.3, 2.4

COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

IN THE PREVIOUS CLASS……

- Branching
- Generating Memory Addresses
- Addressing Modes
  - Register
  - Absolute
  - Immediate
  - Indirect

GUESS…….

1. In CISC Instruction Set, most of the instructions use two address format.
   A. True
   B. False

2. The status register is set based on the output of most recent operation.
   A. True
   B. False

INDIRECTION AND POINTERS

- Indirect mode:
  - The effective address of the operand is the contents of a register that is specified in the instruction
  - Load R2, (R5)

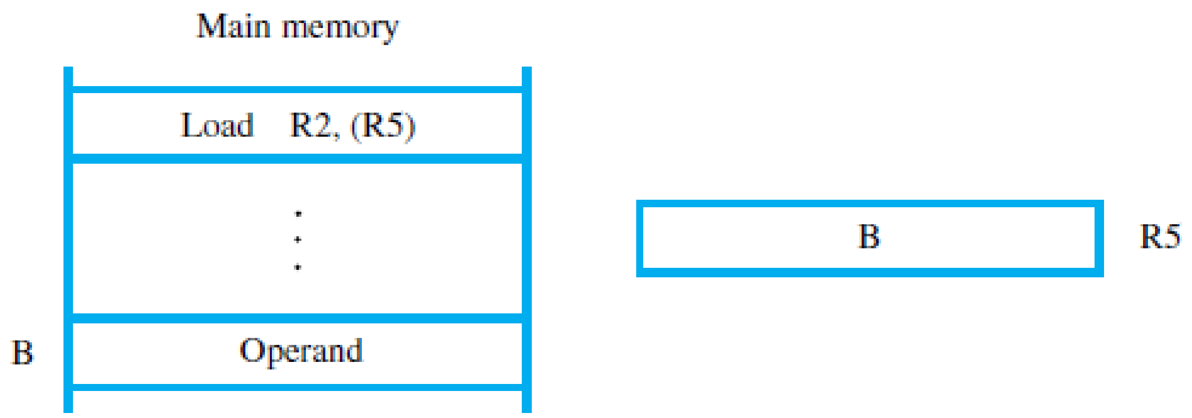


Figure 2.7 Register indirect addressing.

INDIRECTION AND POINTERS

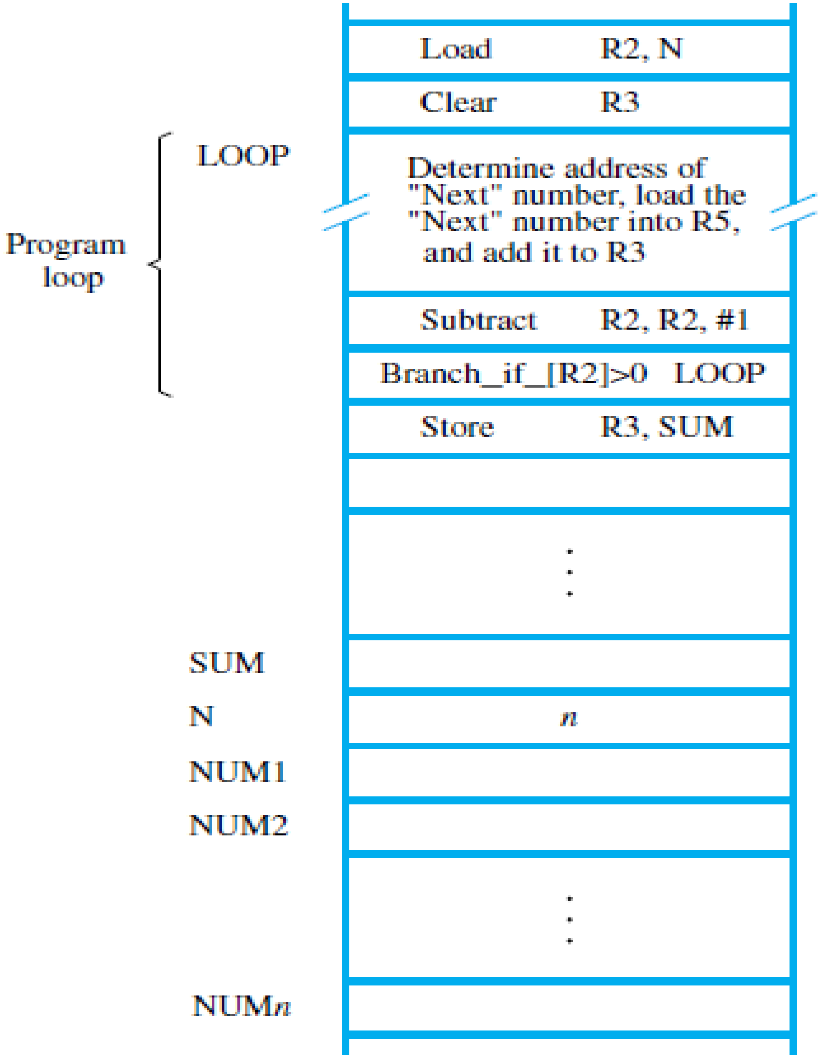


Figure 2.6 Using a loop to add *n* numbers.

	Load	R2, N	Load the size of the list.
	Clear	R3	Initialize sum to 0.
	Move	R4, #NUM1	Get address of the first number.
LOOP:	Load	R5, (R4)	Get the next number.
	Add	R3, R3, R5	Add this number to sum.
	Add	R4, R4, #4	Increment the pointer to the list.
	Subtract	R2, R2, #1	Decrement the counter.
	Branch_if_[R2]>0	LOOP	Branch back if not finished.
	Store	R3, SUM	Store the final sum.

Figure 2.8 Use of indirect addressing in the program of Figure 2.6.

INDIRECTION AND POINTERS

- C-language statement

  A = *B;

- Compiled to:

  Load R2, B

  Load R3, (R2)

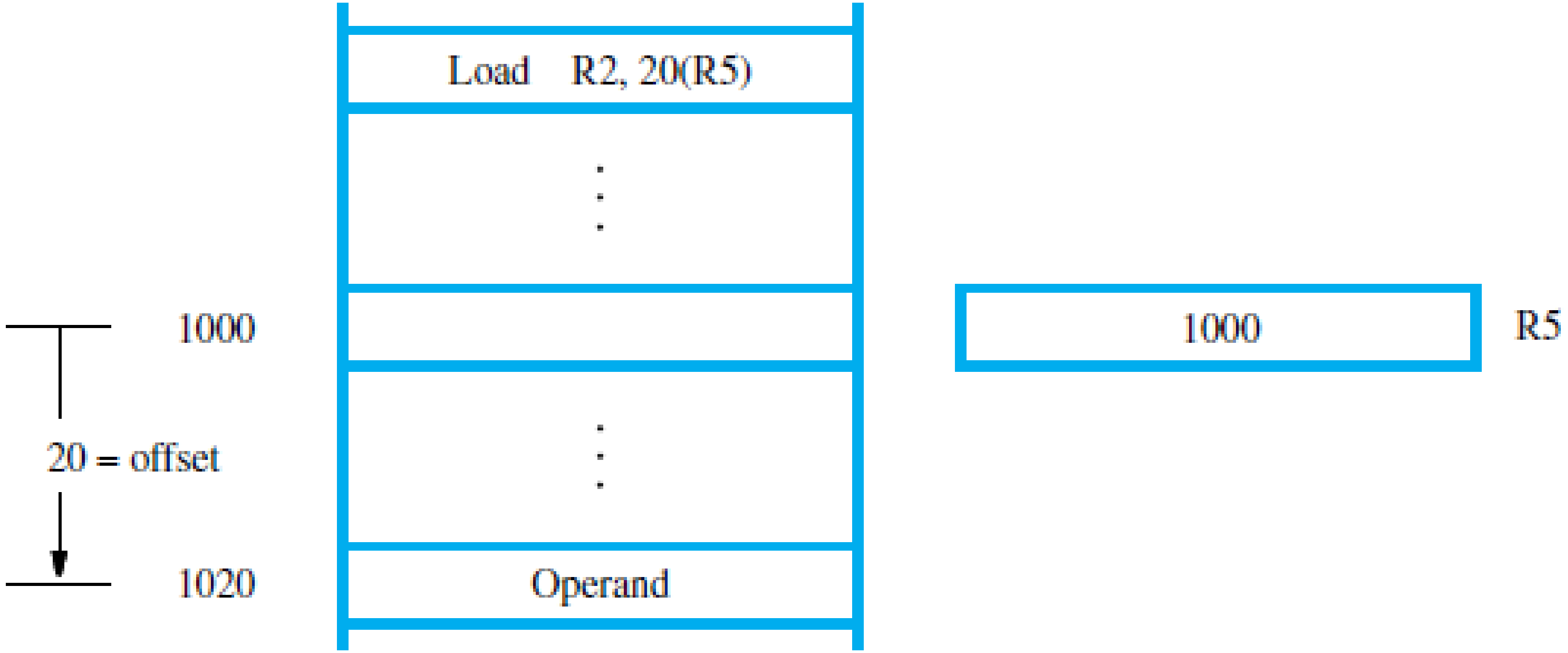
  Store R3, A

INDEXING AND ARRAYS

- Index mode:
  - the effective address of the operand is generated by adding a constant value to the contents of a register.
- Index register: The register used in this mode is known as the index 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.

INDEX MODE: TYPES

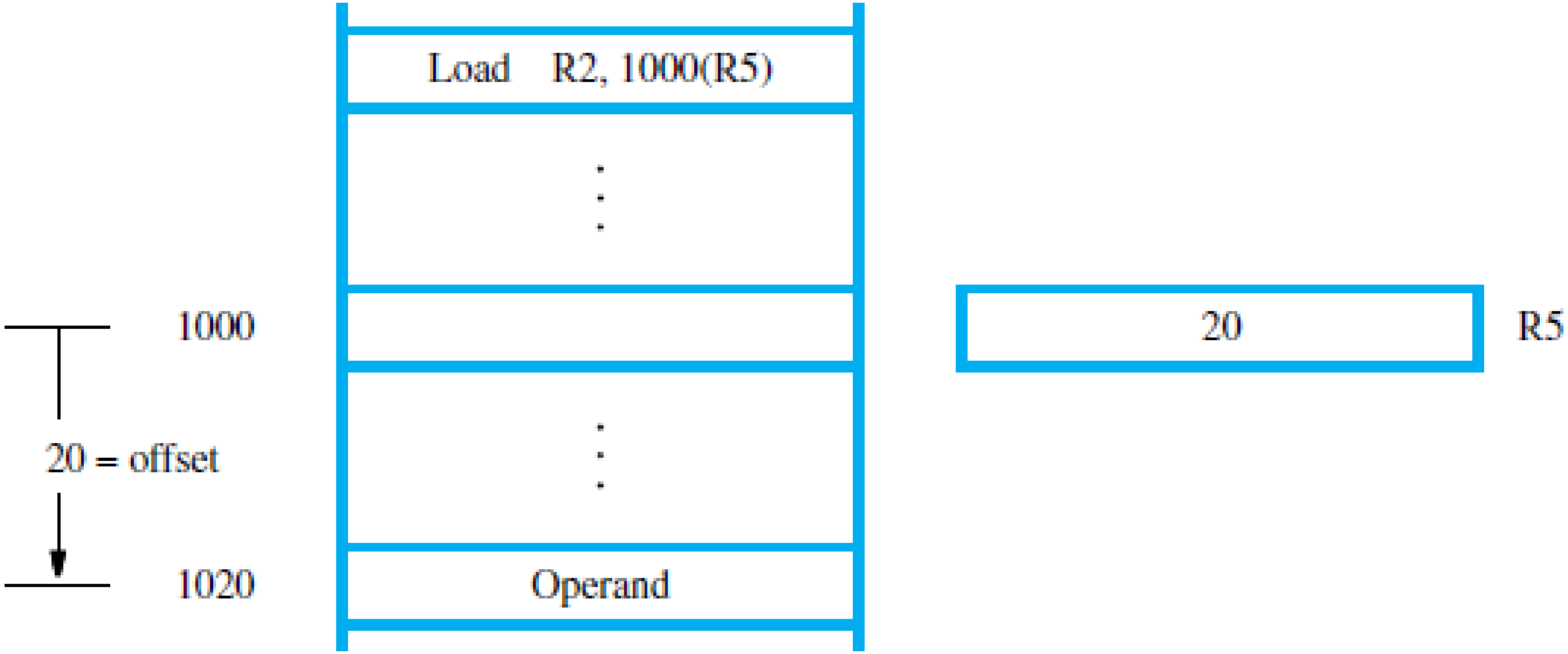
- Offset is given as a constant



(a) Offset is given as a constant

INDEX MODE: TYPES

- Offset is in the index register



(b) Offset is in the index register

INDEXED ADDRESSING: EXAMPLE

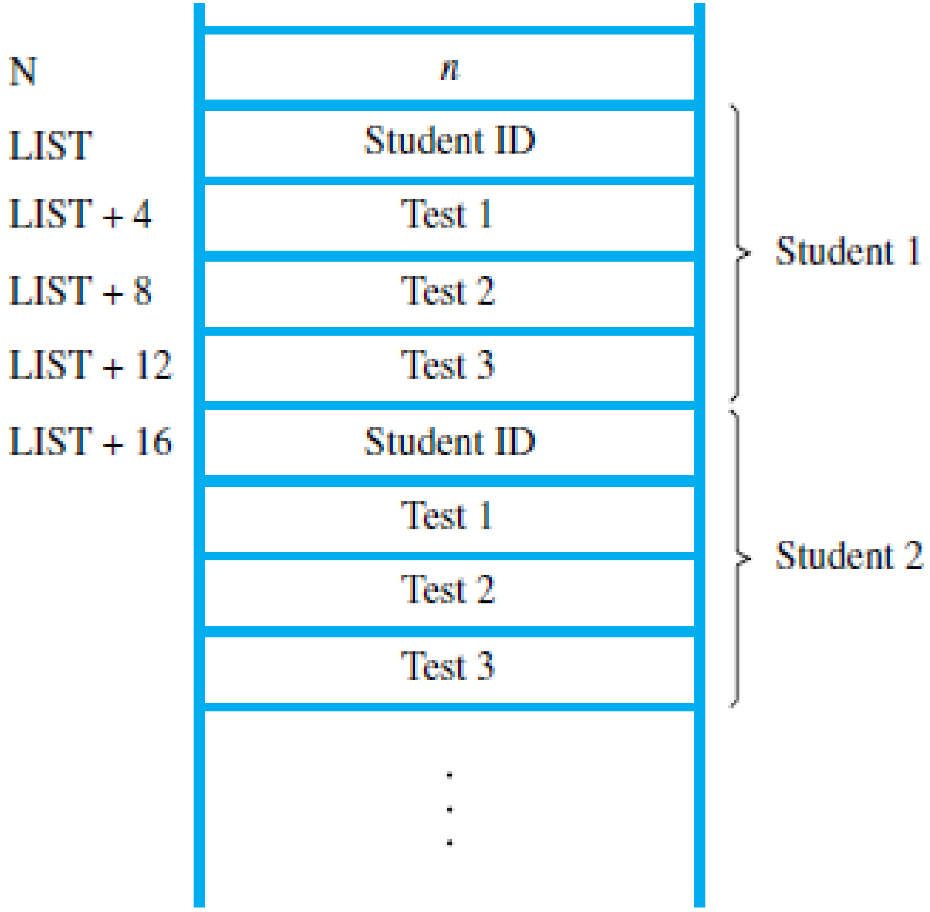


Figure 2.10 A list of students' marks.

INDEXED ADDRESSING: EXAMPLE

```
        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.

INDEXED ADDRESSING: VARIATIONS

- Base with Index: (R*i*,R*j*)
  - EA = [Ri] + [Rj]

- $X(R_i, R_j)$:
  - $EA = X + [R_i] + [R_j]$

ADDRESSING MODES

Table 2.1 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

RISC STYLE

- RISC style is characterized by:
  - Simple addressing modes
  - All instructions fitting in a single word
  - Fewer instructions in the instruction set, because of simple addressing modes
  - Arithmetic and logic operations that can be performed only on operands in processor registers
  - Load/store architecture that does not allow direct transfers from one memory location to another; such transfers must take place via a processor register
  - Simple instructions that are conducive to fast execution by the processing unit using techniques such as pipelining
  - Programs that tend to be larger in size, because more, but simpler instructions are needed to perform complex tasks

CISC STYLE

- CISC style is characterized by:
  - More complex addressing modes
  - More complex instructions, where an instruction may span multiple words
  - Many instructions that implement complex tasks
  - Arithmetic and logic operations that can be performed on memory operands as well as operands in processor registers
  - Transfers from one memory location to another by using a single Move instruction
  - Programs that tend to be smaller in size, because fewer, but more complex instructions are needed to perform complex tasks

CISC INSTRUCTION SET

- CISC instruction sets are not constrained to the load/store architecture, in which arithmetic and logic operations can be performed only on operands that are in processor registers.
- Instructions do not necessarily have to fit into a single word.
- Some instructions may occupy a single word, but others may span multiple words
- Most arithmetic and logic instructions use the two-address format

    Operation destination, source

- An Add instruction of type

    Add B, A

- is written as

$$B \leftarrow [A] + [B]$$

CISC INSTRUCTION SET: EXAMPLE

- Consider the task of adding two numbers where all three operands may be in memory locations

$$C = A + B$$

- This cannot be done with a single two-address instruction.
- Another two-address instruction is required that copies the contents of one memory location into another.

Move C, B

- which performs the operation C←[B] (contents of location B is unchanged).
- The operation C←[A] + [B] can now be performed by the two-instruction sequence

Move C, B

Add C, A

CISC INSTRUCTION SET

- 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
- The general form of the Move instruction is

  Move destination, source
- where both the source and destination may be either a memory location or a processor register

ADDITIONAL ADDRESSING MODES

- Autoincrement mode: (Ri)+
  - 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

- Computers that have the Autoincrement mode automatically increment the contents of the register by a value that corresponds to the size of the accessed operand.
- Increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands.

- Autodecrement mode: −(Ri)
  - The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand

ADDITIONAL ADDRESSING MODES

- To push a new item on the stack,

    Subtract SP, #4

    Move (SP), NEWITEM

- just one instruction can be used

    Move −(SP), NEWITEM

- Similarly, to pop an item from the stack,

    Move ITEM, (SP)

    Add SP, #4

- We can use just

    Move ITEM, (SP)+

ADDITIONAL ADDRESSING MODES

- 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
- Example:

  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 signed number

~~GUESS……~~ ANSWER

1. In CISC Instruction Set, most of the instructions use two address format.
   A. **True**
   B. False

2. The status register is set based on the output of most recent operation.
   A. **True**
   B. False

COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

IN THE PREVIOUS CLASS……

- Addressing Modes
  - Indirect
  - Index
- CISC Instruction Set
- Additional Addressing Modes
  - Autoincrement
  - Autodecrement
  - Relative

CONDITION CODES

- Operations performed by the processor typically generate results such as numbers that are positive, negative, or zero
- Maintain the information about these results for use by subsequent conditional branch instructions
- 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

CONDITION CODES

- Commonly used flags:

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

- e.g. Branch>0 LOOP
- 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

CONDITION CODES

- CISC style programming to add a list of numbers

```
         Move        R2, N        Load the size of the list.
         Clear       R3           Initialize sum to 0.
         Move        R4, #NUM1    Load address of the first number.
LOOP:    Add         R3, (R4)+    Add the next number to sum.
         Subtract    R2, #1       Decrement the counter.
         Branch>0    LOOP         Loop back if not finished.
         Move        SUM, R3      Store the final sum.
```

Figure 2.26 A CISC version of the program of Figure 2.8.

EXAMPLE: VECTOR DOT PRODUCT PROGRAM (RISC STYLE)

- Dot Product $= \sum_{i=0}^{n-1} A(i) \times B(i)$

	Move	R2, #AVEC	R2 points to vector A.
	Move	R3, #BVEC	R3 points to vector B.
	Load	R4, N	R4 serves as a counter.
	Clear	R5	R5 accumulates the dot product.
LOOP:	Load	R6, (R2)	Get next element of vector A.
	Load	R7, (R3)	Get next element of vector B.
	Multiply	R8, R6, R7	Compute the product of next pair.
	Add	R5, R5, R8	Add to previous sum.
	Add	R2, R2, #4	Increment pointer to vector A.
	Add	R3, R3, #4	Increment pointer to vector B.
	Subtract	R4, R4, #1	Decrement the counter.
	Branch_if_[R4]>0	LOOP	Loop again if not done.
	Store	R5, DOTPROD	Store dot product in memory.

Figure 2.27 A RISC-style program for computing the dot product of two vectors.

EXAMPLE: VECTOR DOT PRODUCT PROGRAM (CISC STYLE)

- Dot Product $=\sum_{i=0}^{n-1} A(i) \times B(i)$

```
        Move         R2, #AVEC      R2 points to vector A.
        Move         R3, #BVEC      R3 points to vector B.
        Move         R4, N          R4 serves as a counter.
        Clear        R5             R5 accumulates the dot product.
LOOP:   Move         R6, (R2)+      Compute the product of
        Multiply     R6, (R3)+        next components.
        Add          R5, R6         Add to previous sum.
        Subtract     R4, #1         Decrement the counter.
        Branch>0     LOOP           Loop again if not done.
        Move         DOTPROD, R5    Store dot product in memory.
```

Figure 2.28 A CISC-style program for computing the dot product of two vectors.

ARITHMETIC AND LOGIC UNIT

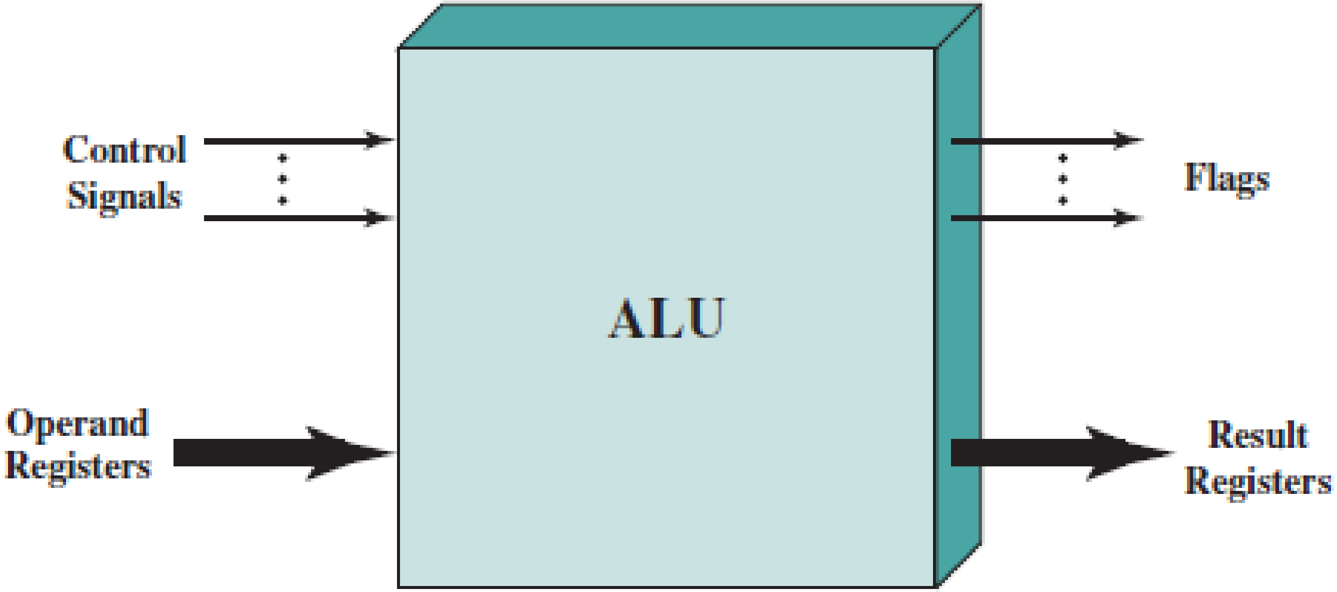


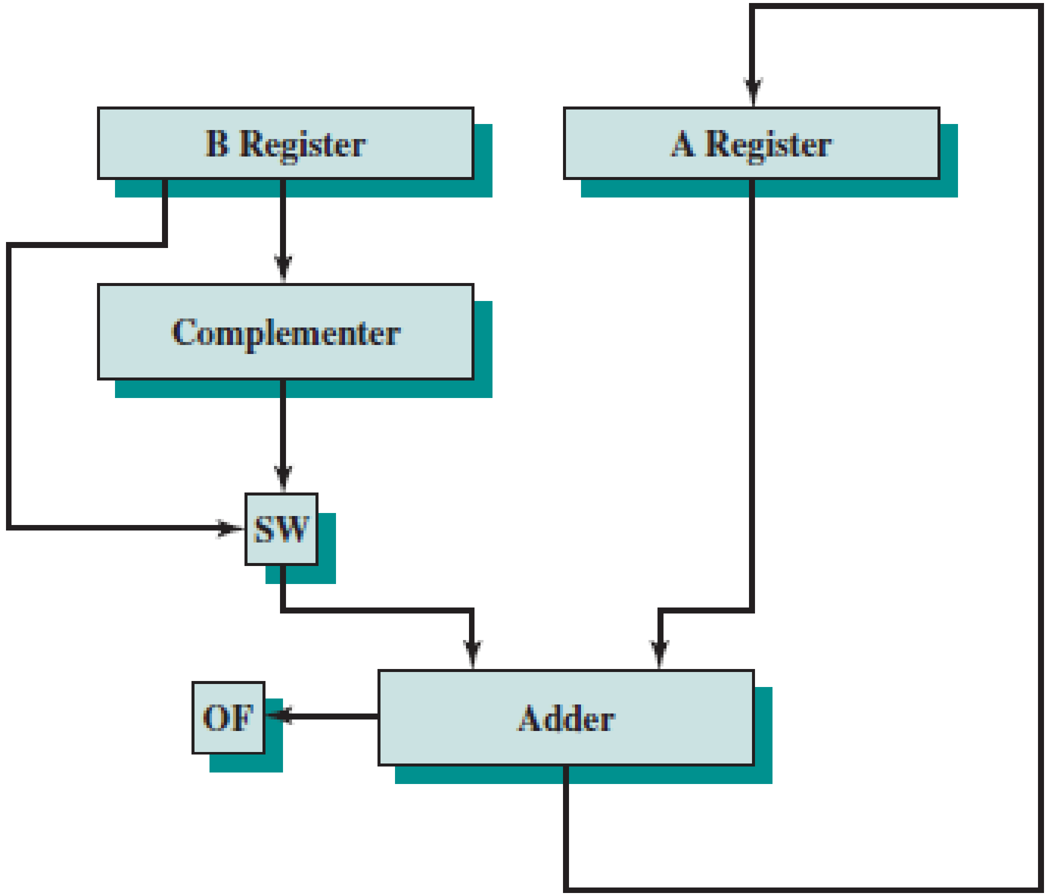
Figure 10.1 ALU Inputs and Outputs

ADDITION AND SUBTRACTION

- OVERFLOW RULE:
  - If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

- SUBTRACTION RULE:
  - To subtract one number (subtrahend) from another (minuend), take the twos complement (negation) of the subtrahend and add it to the minuend.

ADDITION AND SUBTRACTION: HARDWARE



OF = Overflow bit
SW = Switch (select addition or subtraction)

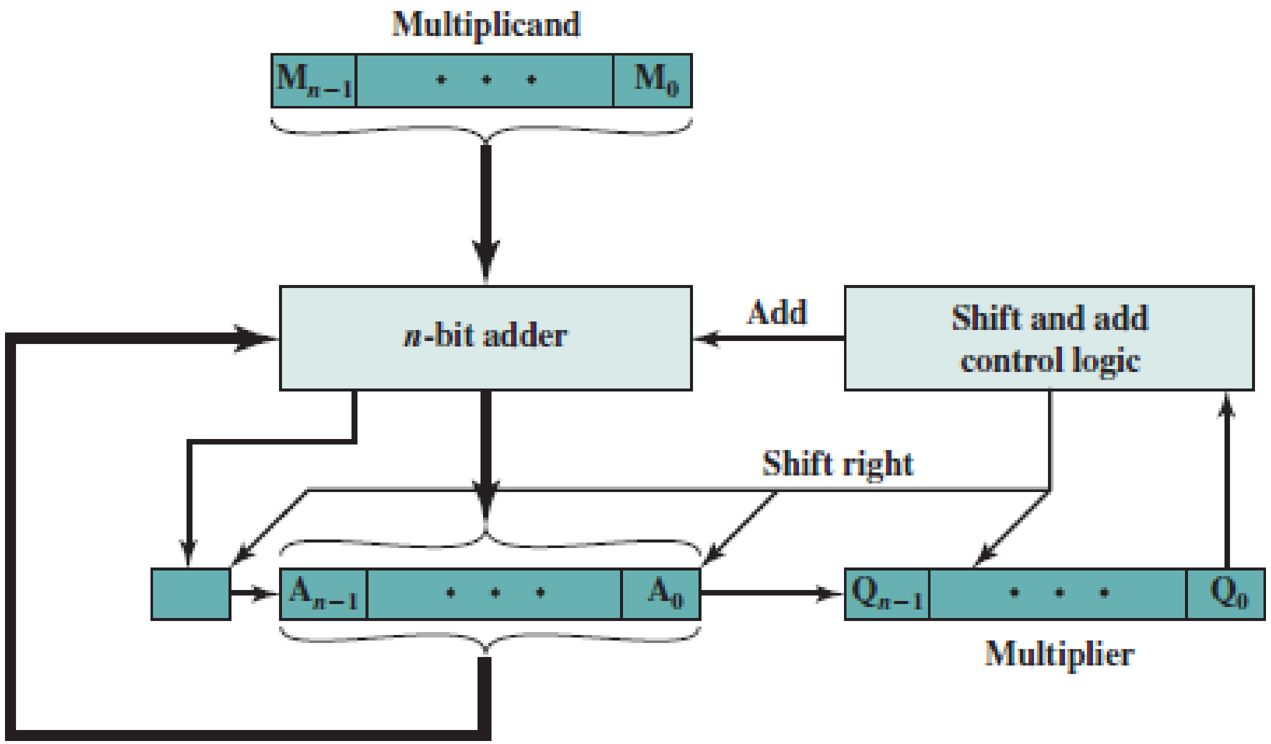
Figure 10.6 Block Diagram of Hardware for Addition and Subtraction

MULTIPLICATION: UNSIGNED INTEGERS

```
    1011      Multiplicand (11)
   ×1101      Multiplier (13)
   -----
    1011
   0000       Partial products
  1011
 1011
--------
10001111      Product (143)
```

Figure 10.7 Multiplication of Unsigned Binary Integers

MULTIPLICATION: UNSIGNED INTEGERS



(a) Block diagram

C	A	Q	M		
0	0000	1101	1011	Initial values	
0	1011	1101	1011	Add	First cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second cycle
0	1101	1111	1011	Add	Third cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth cycle
0	1000	1111	1011	Shift	

(b) Example from Figure 10.7 (product in A, Q)

MULTIPLICATION: UNSIGNED INTEGERS

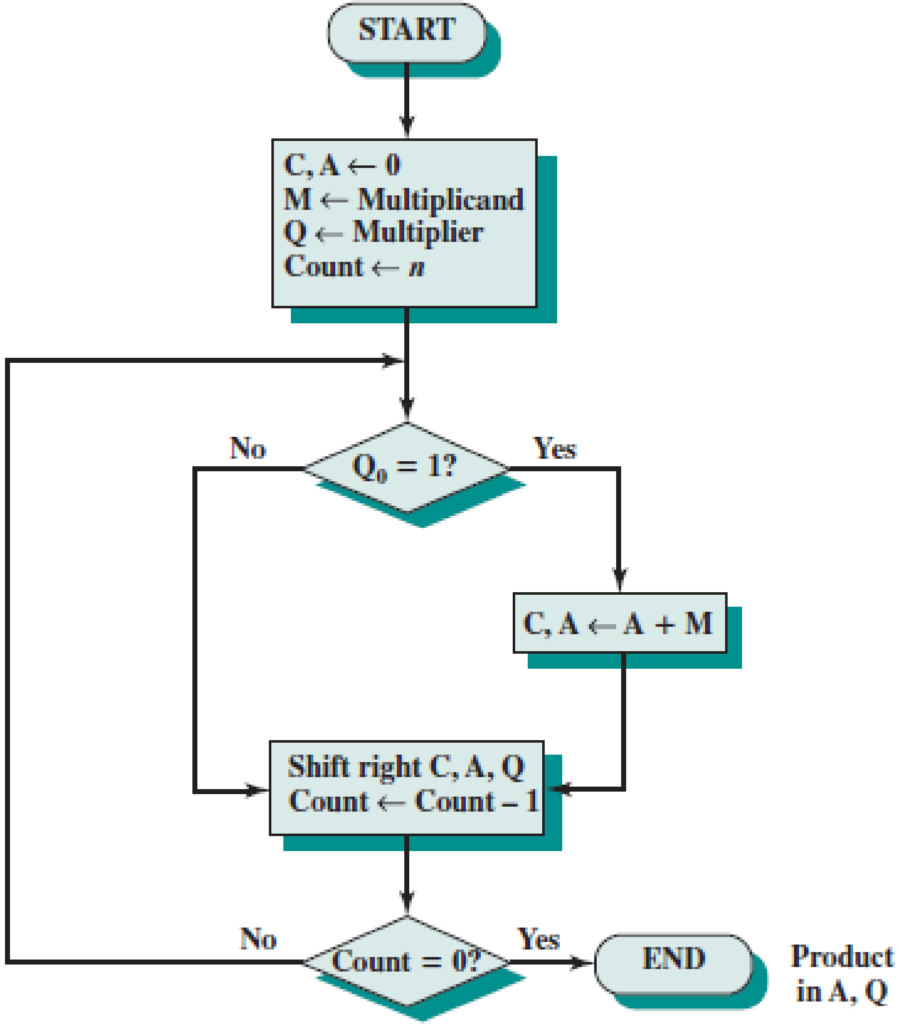


Figure 10.9 Flowchart for Unsigned Binary Multiplication

MULTIPLICATION: UNSIGNED INTEGERS

- 45 (101101) X 33 (100001)=1485

C	A	Q	M	
0 0 0	000000 101101 101101 010110	100001 100001 110000	101101 101101	 Add First Cycle Shift
0	001011	011000	101101	Shift 2^{nd} Cycle
0	000101	101100	101101	Shift 3^{rd} cycle
0	000010	110110	101101	Shift 4^{th} cycle
0	000001	011011	101101	Shift 5^{th} cycle
 0 0	101101 101110 **010111**	 011011 **001101**	 101101 101101	 Add Shift 6^{th} cycle
	Product			

MULTIPLICATION: SIGNED INTEGERS- 2'S COMPLEMENT

- -5 (1011) X -3 (1101) = -113 (10001111)

$$
\begin{array}{rl}
1011 & \\
\times\, 1101 & \\
\hline
00001011 & 1011 \times 1 \times 2^0 \\
00000000 & 1011 \times 0 \times 2^1 \\
00101100 & 1011 \times 1 \times 2^2 \\
01011000 & 1011 \times 1 \times 2^3 \\
\hline
10001111 &
\end{array}
$$

Figure 10.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

MULTIPLICATION: NEGATIVE MULTIPLICAND

- -13 (10011) X +11 (01011) = -143 (1101110001)

						1	0	0	1	1	(−13)
					×	0	1	0	1	1	(+11)
	1	1	1	1	1	1	0	0	1	1	
	1	1	1	1	1	0	0	1	1		
Sign extension is shown in blue	0	0	0	0	0	0	0	0			
	1	1	1	0	0	1	1				
	0	0	0	0	0	0					
	1	1	0	1	1	1	0	0	0	1	(−143)

Figure 9.8 Sign extension of negative multiplicand.

MULTIPLICATION: UNSIGNED V/S SIGNED

$\begin{array}{rl} 1001 & (9) \\ \times\,0011 & (3) \\ \hline 00001001 & 1001 \times 2^0 \\ 00010010 & 1001 \times 2^1 \\ \hline 00011011 & (27) \end{array}$	$\begin{array}{rl} 1001 & (-7) \\ \times\,0011 & (3) \\ \hline 11111001 & (-7) \times 2^0 = (-7) \\ 11110010 & (-7) \times 2^1 = (-14) \\ \hline 11101011 & (-21) \end{array}$
(a) Unsigned integers	(b) Twos complement integers

Figure 10.11 Comparison of Multiplication of Unsigned and Twos Complement Integers

READ FROM….

- Go through examples given in Section 2.12 and 2.15 of the Reference Book (textbook 1)

TOPICS COVERED FROM

- Textbook 1:
  - Chapter 2: 2.11, 2.12
- Textbook 2:
  - Chapter 10: 10.3

COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

IN THE PREVIOUS CLASS……

- Condition codes
- Example program (RISC and CISC style programming)
- Arithmetic and Logic Unit
  - Addition and Subtraction: Hardware
  - Multiplication: Unsigned Integers
  - Multiplication: Signed Integers

MULTIPLICATION: UNSIGNED V/S SIGNED

(a) Unsigned integers	(b) Twos complement integers
1001 (9)	1001 (−7)
× 0011 (3)	× 0011 (3)
00001001 1001 × 2^0	11111001 (−7) × 2^0 = (−7)
00010010 1001 × 2^1	11110010 (−7) × 2^1 = (−14)
00011011 (27)	11101011 (−21)

Figure 10.11 Comparison of Multiplication of Unsigned and Twos Complement Integers

MULTIPLICATION: BOOTH'S ALGORITHM

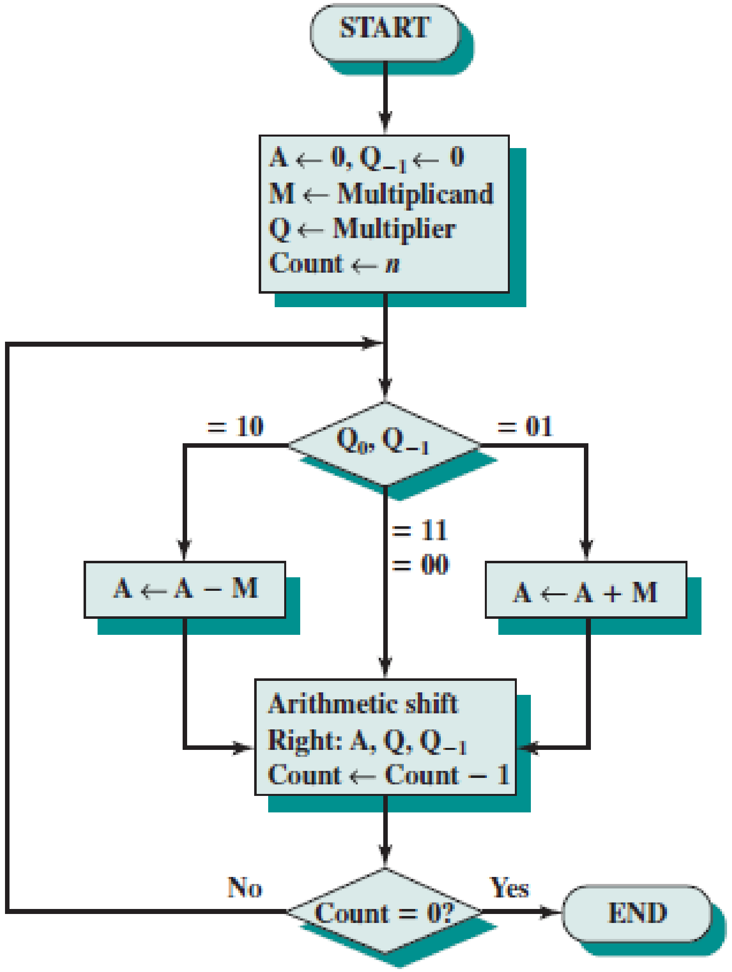


Figure 10.12 Booth's Algorithm for Twos Complement Multiplication

MULTIPLICATION: BOOTH'S ALGORITHM

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	A ← A − M	First cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	A ← A + M	Third cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth cycle

Figure 10.13 Example of Booth's Algorithm (7 × 3)

MULTIPLICATION: BOOTH'S ALGORITHM

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i-1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Figure 9.12 Booth multiplier recoding table.

MULTIPLICATION: BOOTH'S ALGORITHM

```
     0111
   × 0011      (0)
 --------
 11111001      1–0
 0000000       1–1
 000111        0–1
 --------
 00010101      (21)
```

(a) (7) × (3) = (21)

```
     0111
   × 1101      (0)
 --------
 11111001      1–0
 0000111       0–1
 111001        1–0
 --------
 11101011      (–21)
```

(b) (7) × (–3) = (–21)

```
     1001
   × 0011      (0)
 --------
 00000111      1–0
 0000000       1–1
 111001        0–1
 --------
 11101011      (–21)
```

(c) (–7) × (3) = (–21)

```
     1001
   × 1101      (0)
 --------
 00000111      1–0
 1111001       0–1
 000111        1–0
 --------
 00010101      (21)
```

(d) (–7) × (–3) = (21)

Figure 10.14 Examples Using Booth's Algorithm

HOW BOOTH'S ALGORITHM WORKS: +VE MULTIPLIER

- Consider the case of a positive multiplier consisting of one block of 1s surrounded by 0s

$$M * (00011110) = M * (2^4 + 2^3 + 2^2 + 2^1) = M * (16 + 8 + 4 + 2)$$

$$= M * 30$$

$$M * (00011110) = M * (2^5 - 2^1) = M * (32 - 2)$$

$$= M * 30$$

- In general, $$2^n + 2^{n-1} + \cdots + 2^{n-K} = 2^{n+1} - 2^{n-K} \qquad (10.3)$$

- the product can be generated by one addition(Adding the content of 2^5 place value) and one subtraction (Subtracting the content of 2^1 place value) of the multiplicand.
- Booth's algorithm conforms to this scheme by performing a subtraction when the first 1 of the block is encountered (1–0) and an addition when the end of the block is encountered (0–1).

$$M * (01111010) = M * (2^6 + 2^5 + 2^4 + 2^3 + 2^1)$$

$$= M * (2^7 - 2^3 + 2^2 - 2^1)$$

HOW BOOTH'S ALGORITHM WORKS: -VE MULTIPLIER

- Let X be a negative number in twos complement notation: $X = \{1x_{n-2}x_{n-3} \ldots\ldots x_1x_0\}$
- Then the value of X can be expressed as follows:

$$X = -2^{n-1} + (x_{n-2} \times 2^{n-2}) + (x_{n-3} \times 2^{n-3}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0) \quad \textbf{(10.4)}$$

- The leftmost bit of X is 1, because X is negative. Assume that the leftmost 0 is in the k^{th} position. Then,

$$X = \{111 \ldots 10x_{k-1}x_{k-2} \ldots x_1x_0\} \qquad \textbf{(10.5)}$$

- And the value of X is: $X = -2^{n-1} + 2^{n-2} + \cdots + 2^{k+1} + (x_{k-1} \times 2^{k-1}) + \cdots + (x_0 \times 2^0)$ **(10.6)**

- From Equation $2^n + 2^{n-1} + \cdots + 2^{n-K} = 2^{n+1} - 2^{n-K}$ **(10.3)**
- we can say that: $2^{n-2} + 2^{n-3} + \ldots + 2^{k+1} = 2^{n-1} - 2^{k+1}$
- Rearranging: $-2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2^{k+1} = -2^{k+1}$ (10.7)
- Substituting Equation (10.7) into Equation (10.6), we have

$$X = -2^{k+1} + (x_{k-1} * 2^{k-1}) + \ldots + (x_0 * 2^0) \qquad (10.8)$$

HOW BOOTH'S ALGORITHM WORKS: -VE MULTIPLIER

- Consider the multiplication of some multiplicand by (-6). In twos complement representation, using an 8-bit word, (-6) is represented as 11111010.
- $-6 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1$
- $M * (11111010) = M * (-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1)$
- $M * (11111010) = M * (-2^3 + 2^1)$
- $M * (11111010) = M * (-2^3 + 2^2 - 2^1)$

BOOTH'S ALGORITHM: 13X-6

- M=13=01101 Q=-6=11010 -M=10011

A	**Q**	**Q** $_{-1}$	**M**	
00000	11010	0	01101	Initial values
00000	01101	0	01101	Shift Right-1st cycle
<u>10011</u> 10011 *11001*	 01101 *10110*	 0 *1*	 01101 01101	A=A-M [Subtract M from A(Adding 2's complement of M)] *Shift Right- 2nd cycle*
<u>01101</u> 00110 *00011*	 10110 *01011*	 1 *0*	 01101	A=A+M Shift Right- 3rd cycle
<u>10011</u> 10110 *11011*	 01011 *00101*	 0 *1*	 01101 01101	A=A-M [Subtract M from A(Adding 2's complement of M)] Shift Right- 4th cycle
11101	**10010**	1	01101	Shift Right 5th cycle
Taking 2's complement	**0001001110→78** **Product=-78**			

BOOTH'S ALGORITHM: 23X29

- M=23=010111 Q=29=011101 -M=101001

A	**Q**	**Q_{-1}**	**M**	
000000	011101	0	010111	Initial values
<u>101001</u> 101001 110100	 011101 101110	 0 1	 010111	 A=A-M Shift Right-1st cycle
<u>010111</u> 001011 000101	 101110 110111	 1 0	 010111	 A=A+M *Shift Right- 2nd cycle*
<u>101001</u> 101110 110111	 110111 011011	 0 1	 010111	 A=A-M Shift Right- 3rd cycle
111011	101101	1	010111	Shift Right- 4th cycle
111101	110110	1	010111	Shift Right 5th cycle
<u>010111</u> 010100 001010	 110110 011011	 1 0	 010111	 A=A+M *Shift Right- 6th cycle*
	001010011011→667 **Product=667**			

DIVISION

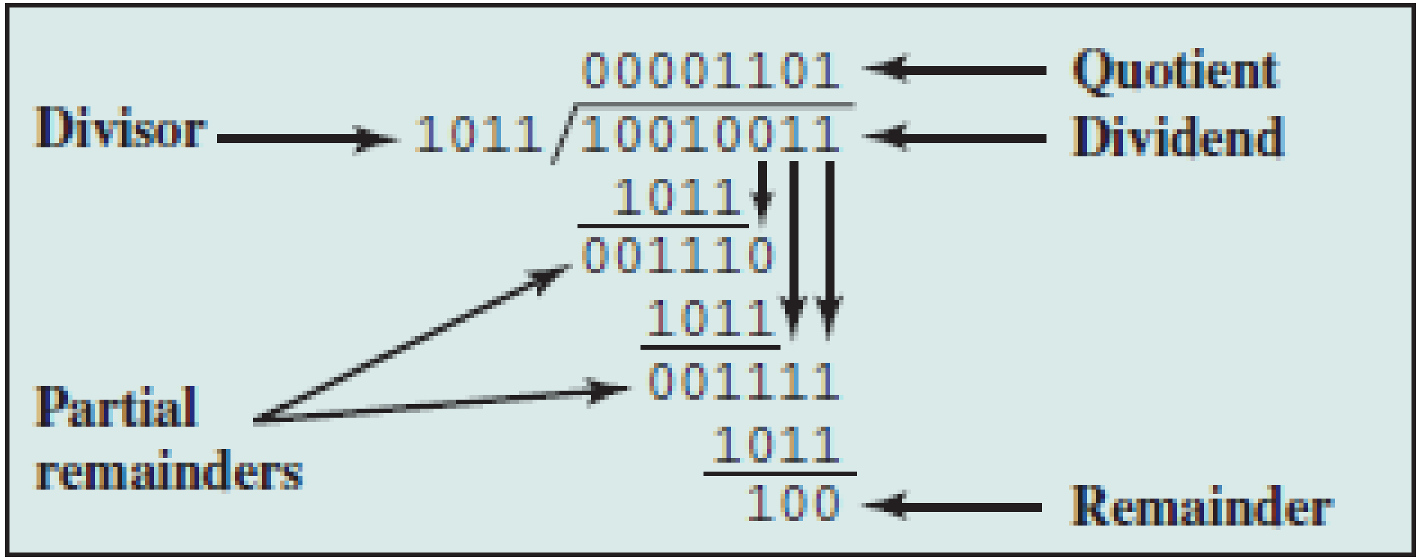


Figure 10.15 Example of Division of Unsigned Binary Integers

DIVISION

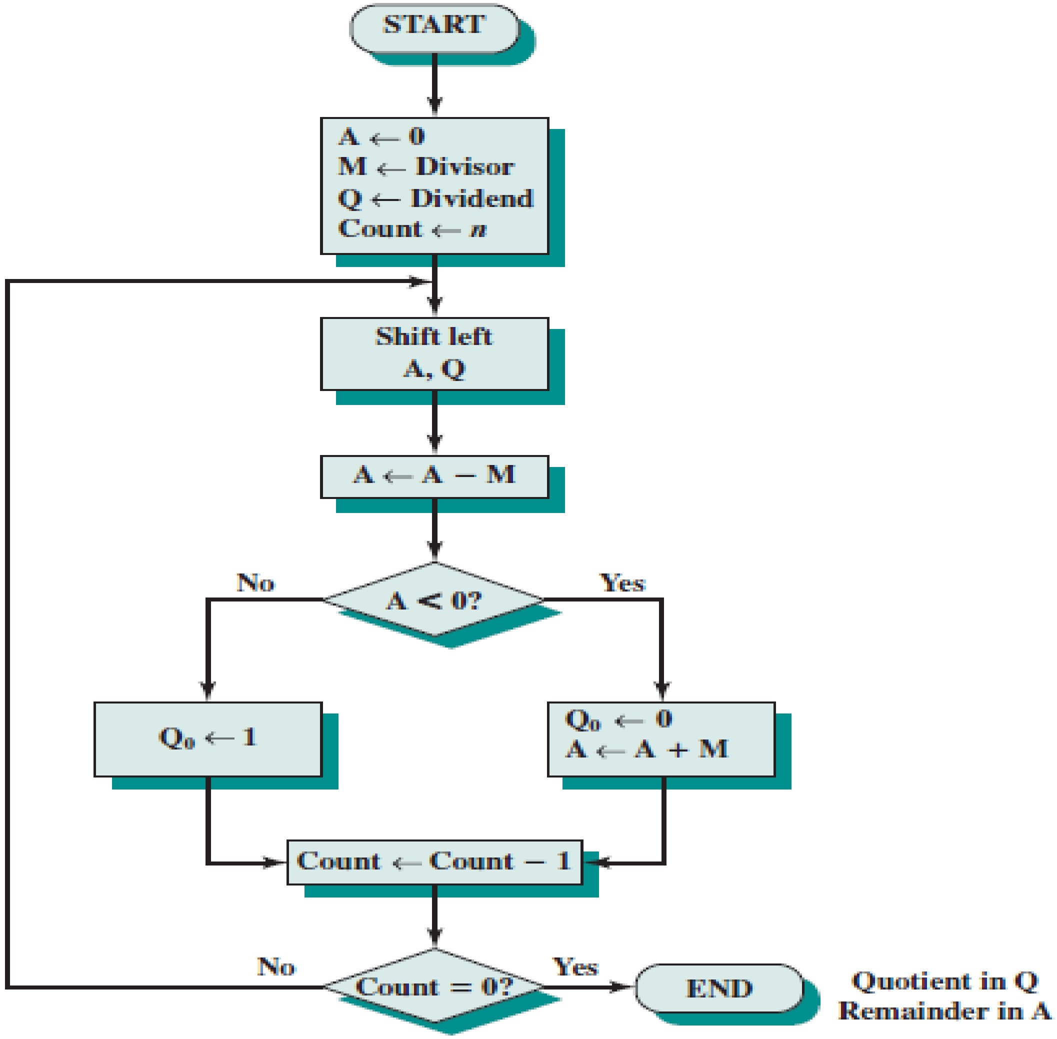


Figure 10.16 Flowchart for Unsigned Binary Division

DIVISION

A	Q	
0000	0111	Initial value
0000	1110	Shift
1101		Use twos complement of 0011 for subtraction
1101		Subtract
0000	1110	Restore, set $Q_0 = 0$
0001	1100	Shift
1101		
1110		Subtract
0001	1100	Restore, set $Q_0 = 0$
0011	1000	Shift
1101		
0000	1001	Subtract, set $Q_0 = 1$
0001	0010	Shift
1101		
1110		Subtract
0001	0010	Restore, set $Q_0 = 0$

Figure 10.17 Example of Restoring Twos Complement Division (7/3)

DIVISION: ALGORITHM

- Assumption: divisor V and the dividend D are positive and that $|V| < |D|$.
- If $|V| = |D|$, then the quotient =1 and the remainder=0.
- If $|V| > |D|$, then Q=0 and R=D. The algorithm can be summarized as follows:
  1. Load the twos complement of the divisor into the M register; that is, the M register contains the negative of the divisor. Load the dividend into the A, Q registers. The dividend must be expressed as a 2n-bit positive number. Thus, for example, the 4-bit 0111 becomes 00000111.
  2. Shift A, Q left 1 bit position.
  3. Perform A=A-M. This operation subtracts the divisor from the contents of A.
  4.
     a. If the result is nonnegative (most significant bit of A=0), then set Q0=1
     b. If the result is negative (most significant bit of A=1), then set Q0=0, and restore the previous value of A.
  5. Repeat steps 2 through 4 as many times as there are bit positions in Q.
  6. The remainder is in A and the quotient is in Q.

DIVISION: EXAMPLE

- Divide 8 by 3; M=-3=11101; Q=01000

A	Q	
00000	01000	Initial values
00000 11101 11101 + 00011 00000	10000 10000 10000	Shift Left Subtract Divisor(Add 2's complement of divisor) Set Q0=0 Restore 1st Cycle
00001 11101 11110 + 00011 00001	00000 00000 00000	Shift Left Subtract Divisor(Add 2's complement of divisor) Set Q0=0 Restore 2nd Cycle
00010 11101 11111+ 00011 00010	00000 00000 00000	Shift Left Subtract Divisor(Add 2's complement of divisor) Set Q0=0 Restore 3rd Cycle
00100 11101 00001	00000 00001	Shift Left Subtract Set Q0=1 4th cycle
00010 11101 11111 + 00011 00010	00010 00010 00010	Shift left Subtract Divisor(Add 2's complement of divisor) Set Q0=0 Restore 5th Cycle

DIVISION

- Consider the following examples of integer division with all possible combinations of signs of D and V:

D = 7	V = 3	➔	Q = 2	R = 1
D = 7	V = -3	➔	Q = -2	R = 1
D = -7	V = 3	➔	Q = -2	R = -1
D = -7	V = -3	➔	Q = 2	R = -1

- (-7)/(3) and (7)/(-3) produce different remainders.
- The magnitudes of Q and R are unaffected by the input signs
- The signs of Q and R are easily derivable from the signs of D and V.
  - sign(R) = sign(D)
  - sign(Q) = sign(D) * sign(V).
- One way to do twos complement division is to convert the operands into unsigned values and, at the end, to account for the signs by complementation where needed.
- This is the method of choice for the restoring division algorithm [PARH10].

EXERCISE

1. Given x and y in twos complement notation i.e., x=0101 and y=1010,compute the product p=x*y with Booth's algorithm
2. Use the Booth algorithm to multiply 23 (multiplicand) by 29 (multiplier), where each number is represented using 6 bits
3. Divide 145 by 13 in binary twos complement notation, using 12-bit words. Use the restoring division algorithm

TOPICS COVERED FROM

- Textbook 2:
  - Chapter 10: 10.3

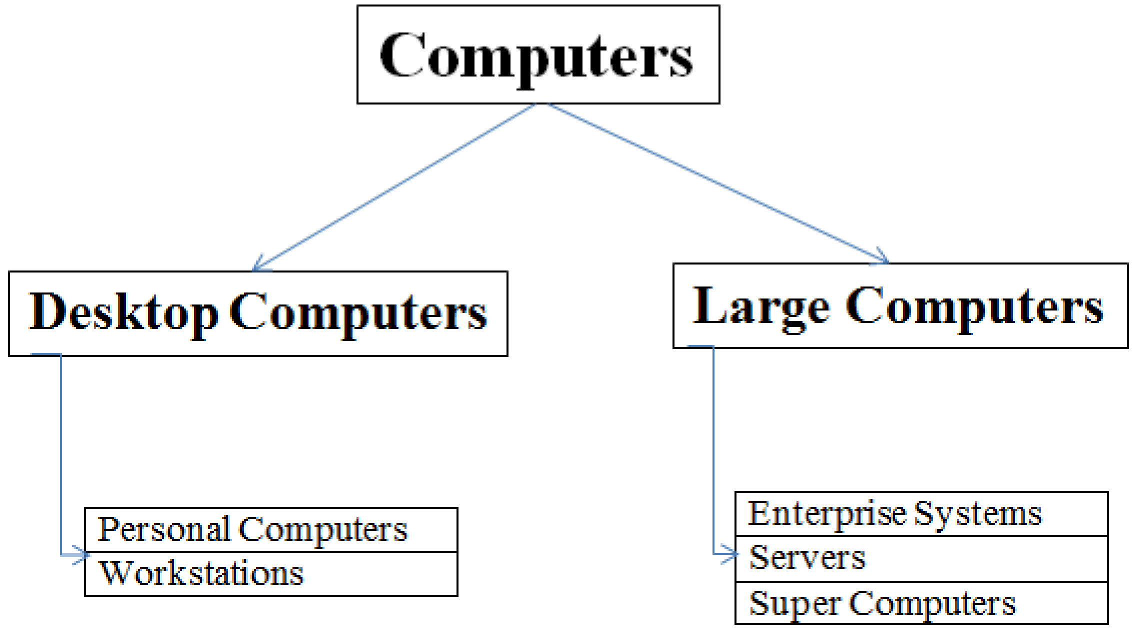
TOPICS COVERED FROM

- Textbook 1:
  - Chapter 2: 2.4, 2.10, 2.11

- ***Computer Types***:
  - **Definition of Computer:**
  - Fast Electronic Calculating machine
  - Input: Digitized info
  - Process: Digitized info as per the stored instructions
  - Output: Resulting information
  - **Program:** The list of instructions that process Digitized info to produce the output
  - **Memory**: Internal storage

Continued…

- **Types:**



Continued…

- Computers are classified into Desktop Computers and Large systems based on Size, Cost, Computational Power
- **Desktop Computers:**
- General Properties:
- They have Processing units, Storage units, Visual display, Audio O/P units, Keyboard
- Point(s) specific to Personal Computers:
- Most Common form of desktop computers
- Uses: Homes, Schools, Business offices
- Portable notebook computers are compact flavor of PCs i.e., all the components are packaged into an unit
- Point(s) specific to Work Stations:
- This has high resolution graphics I/O capabilities
- Computational Power(Work Stations)>Computational Power(PCs)
- Uses: Engineering applications

Continued…

- ✓ **Large Systems:**
- ✓ General Properties:
- ✓ Large and very powerful than Desktop Computers
- ✓ Multiple Processors, Technology like Parallel programming
- ✓ Enterprise Systems, Servers at the lower end of the range, Super Computers at the higher end
- ✓ Point(s) specific to Enterprise Systems
- ✓ Enterprise Systems/Mainframes
- ✓ Used in business data processing in medium to large corporations
- ✓ Computing power and storage capacity is much more than workstations

Continued…

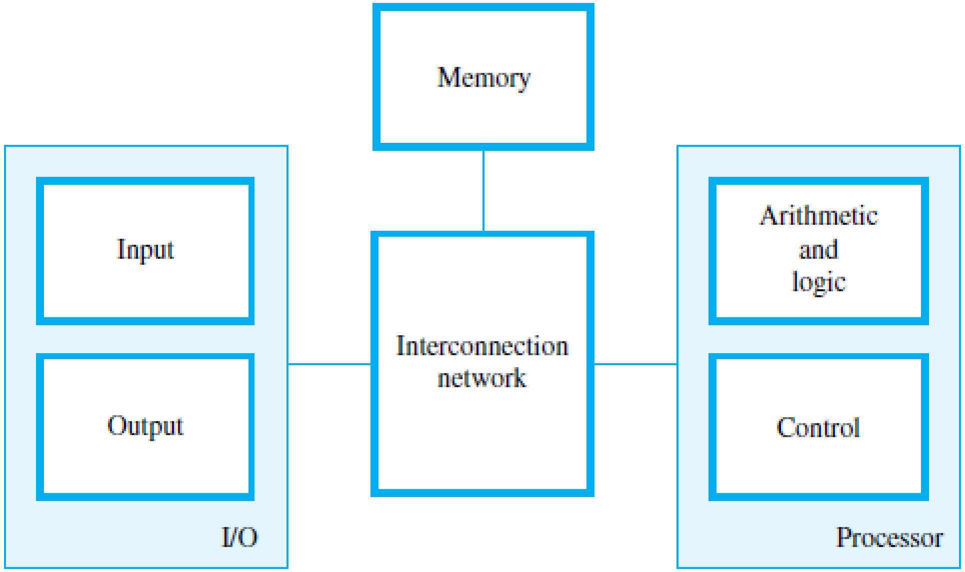
- Point(s) specific to Servers: (IRCTC, GOOGLE….)
- They contain database storage
- They are capable of handling large volumes of requests
- Uses: Education, Business, Personal user communities
- Requests and responses are transported over internet
- Internet communication happens via high speed fiber-optic links
- Point(s) specific to Super Computers:
- Used for Large scale numerical calculations
- Weather Forecasting applications, Satellite image processing applications, aircraft design and simulation

Continued…

- ***Functional Units:***
  - **Introduction:**
  - A computer has 5 ***functionally*** independent parts
  - They are input, output, memory, control, arithmetic and logic units
  - Input and Output units are collectively called I/O
  - ALU and Control units are collectively called Processor
  - Next Slide

Continued…

Functional Units Diagram:



Continued…

- Input unit accepts coded info from electromechanical devices like keyboard , from other computers over digital communication links
- Memory stores the inputted data for later reference
- Inputted data may be given directly given to ALU and performs arithmetic/logic operations (Program stored in memory)
- Output unit sends the result to the outside world
- Control unit coordinates these actions
- There are several ways in which these units can be connected

Continued…

- Info handled by a computer:
- Instructions/Values/Addresses-----Data
- About instructions:
- Specify arithmetic, logical operations to be performed
- Govern transfer of info within computer, between computer and I/O devices
- Program is a list of instructions
- When a program is to be executed it must be in memory
- Processor fetches the instructions of the program to perform operations

Continued…

- Difference between values and addresses
- Values can be numbers/characters
- But addresses are numbers(integers in binary)
- Example wherein a program can be called data
- **<u>Compiler</u>**
- Information handled by a computer must be encoded in suitable form?
- Alphanumeric characters: **ASCII** (to denote a character 8 bits)

Continued…

- Input Unit:
- Computers accept coded info through input units
- Keyboard
- What happens during a Key press?
- **<u>When a key is pressed the ascii value for the key gets stored in a memory location</u>**
- Others include joysticks, trackballs, mouse
- Graphic input devices

Continued…

- Memory Unit:
- To store programs and data
- RAM: (Primary memory/Main memory)
- Memory has a large no of **semiconductor storage cells** each of which will store one bit
- Individually they are rarely read/written
- Instead they are read/written in groups
- ?
- Word (16 to 64 bits and depends on ALU)
- Memory is organized in a way that a word can be read/written in an operation

Continued…

- ✓ Addresses are associated with words for easy retrieval
- ✓ Address and control command
- ✓ Memory capacity determines size of computer
- ✓ Smaller machines---tens of millions of words
- ✓ Larger--- hundreds of millions of words
- ✓ For memory word is the basic building block
- ✓ RAM---Random Access Memory
- ✓ Memory where any word can be accessed in the same amount of time
- ✓ Memory Access Time?
- ✓ Time required to access a word
- ✓ In modern RAMs it is few ns to 100ns
- ✓ Cache---Small, fast RAM unit(s)

Continued…

- ALU:
- Most of the operations are executed in ALU
- Addition of 2 numbers. How?
- Any other operation like mul, comparison can be done by bringing operands into processor and operation being performed by ALU
- Operands are stored in the registers of the processor
- CU and ALU are many times faster than other computer devices.
- That is how processor controls a number of devices

Continued…

- Output Unit:
- Counterpart
- Send processed results outside
- Printer
- Ink Jet printers, Laser printers (photocopying technique—Xerox machines) for printing
- We can have a printer that prints 10000 lines/min
- This speed may be great for mechanical device but nothing in front of processor speed

Continued…

- Control Unit:
- The operations of input, output, memory, ALU must be coordinated
- CU
- Its job is to send control signals to other units to get to know the states
- I/O transfers, input and output operations are controlled by instructions
- But then actual timing signals for transfers are generated by CU
- Timing signals determine when a given action has to take place
- Data transfers b/w processor and memory are also controlled by CU
- CU theoretically can be thought of as a unit that interacts with other parts
- In practice the circuitry is spread throughout the machine

- ***Basic Operational Concepts***:
  - Program in memory
  - Individual instructions are brought from memory into processor (also data) to perform the operations
  - Ex1: Load R2, LOC
    - This instruction reads the contents of a memory location whose address is represented symbolically by the label LOC and loads them into processor register R2.
    - The original contents of location LOC are preserved, whereas those of register R2 are overwritten.
  - Ex2: Add R4, R2, R3
    - adds the contents of registers R2 and R3, then places their sum into register R4. The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum.

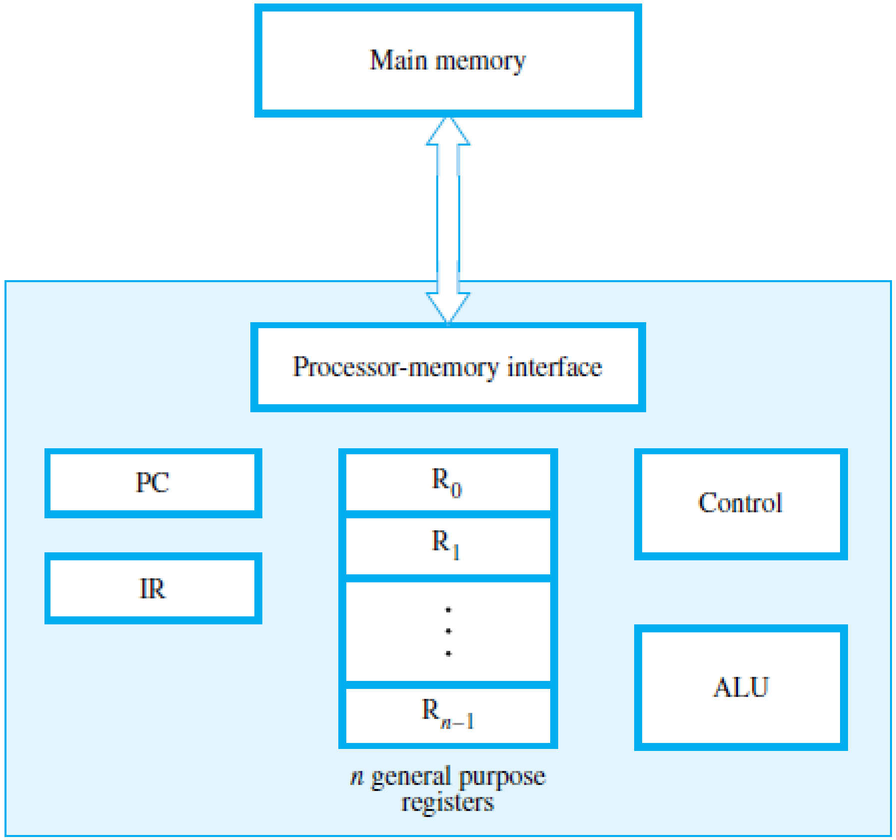
Continued…

- Ex3: Store R4, LOC
- This instruction copies the operand in register R4 to memory location LOC.
- The original contents of location LOC are overwritten, but those of R4 are preserved.

- PTO

Continued…

- Connections between memory and processor (one of the ways)



Continued…

- ✓ IR—Instruction Register
- ✓ It holds the instruction currently being executed
- ✓ PC—Program Counter
- ✓ It contains the address of the next instruction to be fetched and executed
- ✓ During the execution of an instruction PC is updated
- ✓ General purpose registers hold addresses/values (R0-Rn-1)

Continued…

- Only 2 registers communicate with memory
- MAR—Memory Address Register
- MDR—Memory Data Register
- MAR holds the address of location to be accessed
- MDR holds the data read from/written into the addressed location
- Scenario: **<u>MAR, MDR, Control Signal</u>**
- 1. Instruction has to be fetched whose address is in PC
- 2. Data has to be fetched from memory
- 3. Result has to be stored in memory

Continued…

- ✓ Interrupts?
- ✓ A request from I/O devices
- ✓ Scenario: I/O device wants immediate attention of Processor
- ✓ Sends interrupt signals
- ✓ Processor before servicing the interrupt saves the PC, GPR of the current program
- ✓ Processor executes ISR for that interrupt
- ✓ Can be returned to the previously executing program

- ***Numbers:***
  - Computers operate on info represented by 2 electric signals 0 and 1
  - 01000111001010—Bit/Binary digit
  - What is the way to represent numbers and characters?
  - String of bits
  - Number representation:
  - Unsigned integers
  - A n-bit vector can represent unsigned integers in the range?
  - 0 to (2^n) -1

Continued…

- It is equally impt to represent signed numbers
- How?
- 3 Systems
- 1. Sign and Magnitude
- 2. 1’s Complement
- 3. 2’s Complement
- Thing that is common to all 3 is for +ve numbers MSB is 0, for –ve numbers MSB is 1
- Positive values have identical representation in all 3 systems but negatives have different representations

Continued…

- Sign and Magnitude:
- In this system 1 bit (MSB) is reserved for sign and the rest of the bits to represent magnitude
- Ex: +5 (4-bit binary format)
- 0101
- Ex: -5 (4-bit binary format)
- 1101
- Just change the MSB from 0 to 1 if the negative value of a positive value needs to be represented
- Next Slide

Continued…

- 1’s Complement:
- For +ve numbers:
- Normal Binary form (Unsigned number)
- For –ve numbers:
- The binary rep of the corresponding +ve number is taken and all bits are complemented
- Ex: +7 (4-bit format)
- 0111
- Ex: -7 (4-bit format)
- Take binary rep of +7 and complement all bits
- 1000

Continued…

- 2’s Complement:
- Ex: +7 (4-bit binary form)
- 0111
- You subtract this from 10000
- What you get is 1001 which is the 2’s complement for -7
- Table (PTO)

Continued…

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

Continued…

- ✓ Sign and magnitude, 1's complement have diff representations for +0 and -0
- ✓ But 2's complement has only one representation
- ✓ -8 has representation only in 2's complement form
- ✓ 2's complement is the most efficient way to carry out arithmetic operations

- ***Arithmetic Operations:***
- Addition of Positive/ Unsigned numbers (4-bit binary format)
- Adding 15 with 15 (30)
- 1111 with 1111 (11110) Now the carry generated as a result of addition is considered overflow since overflow has occurred

Continued…

- Addition and Subtraction of Signed numbers
- When the 2 nos that are to be added are in the range of system and if after addition the result is in the range of the system then no overflow has occurred. **Carry generated if any in this case** will not be considered overflow (+7+(-3))
- When the 2 nos that are to be added are in the range of system and if after addition the result is not in the range of the system then overflow has occurred. **Carry generated in this case** will be considered overflow -7+(-2)= -9
- **When 2 operands have same sign and the MSB of the result is of opposite sign then overflow is said to have occurred** -7+(-2)= -9

Continued…

- Addition rules:
- Add
- If the result is in the range, any carry generated is ignored
- If the result is out of the range, carry generated in that case is considered overflow
- Subtraction rules:
- X-Y
- Find 2’s complement for Y and add it to X
- Addition rules apply here

Continued…

(a)
$$\begin{array}{r} 0010 \\ +\ 0011 \\ \hline 0101 \end{array} \quad \begin{array}{r} (+2) \\ (+3) \\ \hline (+5) \end{array}$$

(b)
$$\begin{array}{r} 0100 \\ +\ 1010 \\ \hline 1110 \end{array} \quad \begin{array}{r} (+4) \\ (-6) \\ \hline (-2) \end{array}$$

(c)
$$\begin{array}{r} 1011 \\ +\ 1110 \\ \hline 1001 \end{array} \quad \begin{array}{r} (-5) \\ (-2) \\ \hline (-7) \end{array}$$

(d)
$$\begin{array}{r} 0111 \\ +\ 1101 \\ \hline 0100 \end{array} \quad \begin{array}{r} (+7) \\ (-3) \\ \hline (+4) \end{array}$$

(e)
$$\begin{array}{r} 1101 \\ -\ 1001 \\ \hline \end{array} \quad \begin{array}{r} (-3) \\ (-7) \\ \hline \end{array} \quad \Longrightarrow \quad \begin{array}{r} 1101 \\ +\ 0111 \\ \hline 0100 \end{array} \quad \begin{array}{r} \\ \\ \hline (+4) \end{array}$$

(f)
$$\begin{array}{r} 0010 \\ -\ 0100 \\ \hline \end{array} \quad \begin{array}{r} (+2) \\ (+4) \\ \hline \end{array} \quad \Longrightarrow \quad \begin{array}{r} 0010 \\ +\ 1100 \\ \hline 1110 \end{array} \quad \begin{array}{r} \\ \\ \hline (-2) \end{array}$$

(g)
$$\begin{array}{r} 0110 \\ -\ 0011 \\ \hline \end{array} \quad \begin{array}{r} (+6) \\ (+3) \\ \hline \end{array} \quad \Longrightarrow \quad \begin{array}{r} 0110 \\ +\ 1101 \\ \hline 0011 \end{array} \quad \begin{array}{r} \\ \\ \hline (+3) \end{array}$$

(h)
$$\begin{array}{r} 1001 \\ -\ 1011 \\ \hline \end{array} \quad \begin{array}{r} (-7) \\ (-5) \\ \hline \end{array} \quad \Longrightarrow \quad \begin{array}{r} 1001 \\ +\ 0101 \\ \hline 1110 \end{array} \quad \begin{array}{r} \\ \\ \hline (-2) \end{array}$$

(i)
$$\begin{array}{r} 1001 \\ -\ 0001 \\ \hline \end{array} \quad \begin{array}{r} (-7) \\ (+1) \\ \hline \end{array} \quad \Longrightarrow \quad \begin{array}{r} 1001 \\ +\ 1111 \\ \hline 1000 \end{array} \quad \begin{array}{r} \\ \\ \hline (-8) \end{array}$$

(j)
$$\begin{array}{r} 0010 \\ -\ 1101 \\ \hline \end{array} \quad \begin{array}{r} (+2) \\ (-3) \\ \hline \end{array} \quad \Longrightarrow \quad \begin{array}{r} 0010 \\ +\ 0011 \\ \hline 0101 \end{array} \quad \begin{array}{r} \\ \\ \hline (+5) \end{array}$$

Continued…

- Sign Extension?
- Suppose if I want to store -8 in 16-bit register
- The sign bit is extended to the rest of the bit positions for the value to be retained
- 2's complement is used for signed numbers rep in modern computers

Continued…

- ***Characters:***
  - Computers should handle text also
  - Representing characters, digits as characters, punctuations is impt
  - ASCII character set

Performance

The most imp measure of the performance of a comp is how quickly it can execute prgms.

The speed with which a comp executes pgms is affected by the design of its instr set, its H/W and S/W including OS, and the technology in which the H/W is implemented.

i) **Technology**

ii) **Parallelism**

a) Instruction-level Parallelism: *pipelining*

overlap the exec of the steps of successive instrs.

Total exec time will be reduced.

b) Multicore Processors: *dual-core*, *quad-core*, and *octo-core* processors

c) Multiprocessors: *shared-memory multiprocessor*

message passing multicomputers

- We need a system that can accommodate very large integers and very small fractions and that system should take care of point because in FP nos point is not fixed its variable
- Floating point representations
- What constitute the representation?
- Sign, String of digits (Mantissa) and exponent constitute the representation
- Next Slide

Continued…

- IEEE standard for floating point nos:
- 3 standards/formats
- Single precision---32 bit format
- Double precision--- 64 bit format
- Single Precision:
- In single precision format, 1 bit for sign, 8 bits for exponent, 23 bits for mantissa

Continued…

✓

(a) Single format

Continued…

- Biased Exponent/ Exponent in Excess 127 representation
- ?
- To store the signed exponent as an unsigned integer
- If the original exponent is -100
- Then it will be biased to 27
- Which value is represented by the single precision format 0|00101000|001010 (Normalized version)
- ?
- Represent -23.75 in single precision format

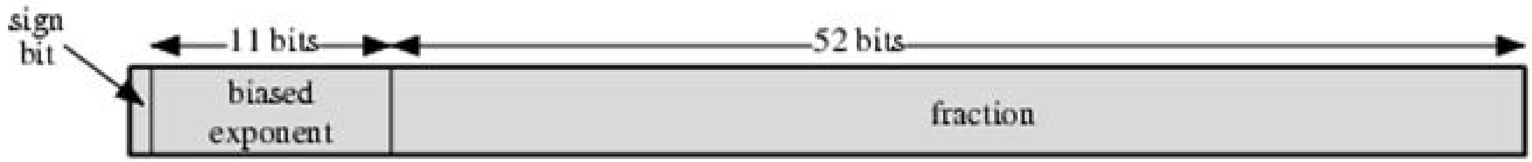
Continued…

- Special Cases:
- Using 8 bits we can represent signed exponent in the range -128 to +127
- Exponents are biased
- The range of values of biased exponent if the exponent field has 8 bits is 0 to 255
- But 0 and 255 are special values
- The range is 1 to 254
- Hence the range of signed unbiased exponents that can be represented is -126 to +127
- 23 bits represent fractional part of mantissa
- It is called single precision since it occupies a single 32 bit word
- Scale factor of 10^-38 to +10^+38 is possible here

Continued…

- Double precision:
- It occupies 2 32 bit words
- 1 bit for sign
- 11 bits for exponent
- 52 bits for mantissa
- Here also the exponent is biased and the biasing value is 1023 (2^(n-1)-1)
- Range of biased exponents: 1 to 2046 (0, 2047 special values)
- Range of signed unbiased exponents: -1022 to +1023
- Scale factor: 10^+308 , 10^-308 is possible here

Continued…



(b) Double format

Continued…

- Special Values:
- E’= 0 && M=0 (exact zero) or E’=255 && M=0 (Infinity)
- The corresponding Es are -127, +128
- Now let’s say I am adding 2 binary fractions that are in single precision whose exponents are -126, -126 and let’s say the exponent of the result is -127 and M=0
- ?
- Underflow has occurred and the result is 0 (+ or -)
- If exponent is -127 and M!=0 then the result is a denormal number (smaller than normal)
- Similarly, let’s say I am adding 2 binary fractions that are in single precision whose exponents are 127 and 127 and let’s say the exponent of the result is +128 and M=0
- ?
- Overflow has occurred and the result is infinity (+ or -)

- Arithmetic operations:
- Addition
- Subtraction
- Multiplication
- Division
- Rules are given with respect to single precision format and these rules can be used for format that is of less or equal length of single precision
- Next Slide

Continued…

- Addition/Subtraction rules:
- 1. Convert the given decimal fractions to binary fractions
- 2. Normalize the converted binary fractions
- 3. Sign of the result: Sign of the number with the largest exponent is the sign of result
- 4. Find the biased exponents and represent the numbers
- 5. Take the number with smaller exponent and logical right shift its mantissa by difference in the exponents (Don't forget the 1 that is not represented).Now the exponent of this number would have become equal to the exponent of no with larger exponent
- 6. Add/Subtract the shifted mantissa of the no in step5 with the unshifted mantissa of the other number
- 7. Normalize the final answer, Represent it in single precision and check the answer for correctness

Continued…

- The numbers have to be represented in single precision and same is the result
- 1. 1.25+0.25
- 2. 1.25-0.25
- Multiplication Rules:
- 1. Convert the given decimal fractions to binary fractions
- 2. Normalized the converted binary fractions
- 3. Sign of the result is the XOR of sign of 2 nos
- 4. Find the biased exponents and represent the numbers
- 5. Add the biased exponents and subtract 127
- 6. Multiply mantissas: (Don't forget the hidden 1) and keep the result obtained
- 7. Normalize the mantissa if required and add the generated exponent to the biased exponent obtained in step 5
- 8. Represent the result in single precision
- 9. Check the correctness of the result

Continued…

- ✓ Multiply : -5.75 *1.5=-8.625
- ✓ Division Rules:
- ✓ 1. Convert the given decimal fractions to binary fractions
- ✓ 2. Normalized the converted binary fractions
- ✓ 3. Sign of the result is XOR of sign of 2 nos
- ✓ 4. Find the biased exponents and represent them
- ✓ 5. Subtract the biased exponents and add 127
- ✓ 6. Divide mantissas: (Don't forget the hidden 1) and keep the result obtained
- ✓ 7. Normalize the mantissa if required and add the generated exponent to the biased exponent obtained in step 5
- ✓ 8. Represent the result in single precision
- ✓ 9. Check the correctness of the result

Continued…

- Division: -5.75/1.5=-3.833333…………
- Guard Bits and Truncation:
- We have guard bits (3)
- First bit is guard bit, second bit is round bit, third bit is sticky bit
- When bit value 1 comes into sticky bit ,1 is retained
- This yields more accurate final result

Continued…

- Suppose if the final result is to be rounded off
- ?
- Rounding techniques are there
- 1. Chopping
- 2. Von Neumann Rounding
- 3. Rounding
- 1. Chopping: Let's say I want to truncate from n bits to m bits
- Simply the n-m bits from right to left are chopped

Continued…

- ✓ 2. Van Neumann rounding:
- ✓ If the bits to be removed are all 0s they are chopped
- ✓ If any one is 1, then the LSB of retained bits set to 1
- ✓ 3. Rounding:
- ✓ If the MSB of the bits to be removed is 1, then 1 is added to the LSB of retained bits
- ✓ Best method

Continued…

- **Round the final result 1.110110 from 6 bits to 3 bits in the fraction part using**
- **1. Chopping**
- **2. Van Neumann**
- **3. Rounding**
- Guard bit, Round bit, Sticky bit + Rounding:
- G=1, R=1 add 1 to LSB (Round UP)
- G=0, R=0||1 No change (Round down)
- G=1, R=0 Look at S
- If S=1 Round Up
- If S=0 Round to nearest even (If LSB is 0 leave it or else add 1 to LSB)

Continued…

- A=0 10001 011011
- B=0 01111 101010
- Do Subtract, Truncate the result using rounding

Floating point Arithmetic

- Addition
- Subtraction
- Multiplication
- Division
- IEEE 32-bit single precision
- IEEE 64-bit single precision

- *FOR FLOATING POINT PORTION*

As per the syllabus floating point portion is from William stallings Text 2).

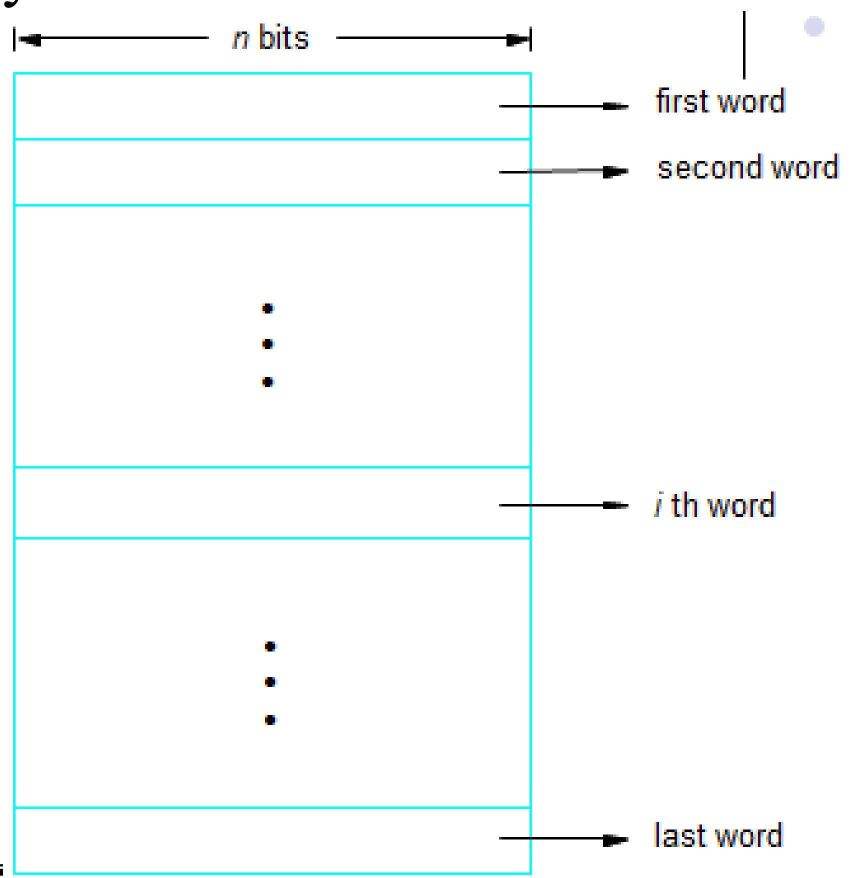
But it is covered from Hamacher Text 1.

9.7
9.7.1
9.7.2

- ***Memory Locations and Addresses***:
  - How memory is organized?
  - Memory has millions of storage cells
  - Each cell can store only a bit (Either 0 or 1)
  - Bits are rarely handled individually
  - Instead handled in groups of fixed size
  - Words
  - This can be stored or retrieved in a single operation
  - Memory---Collection of words

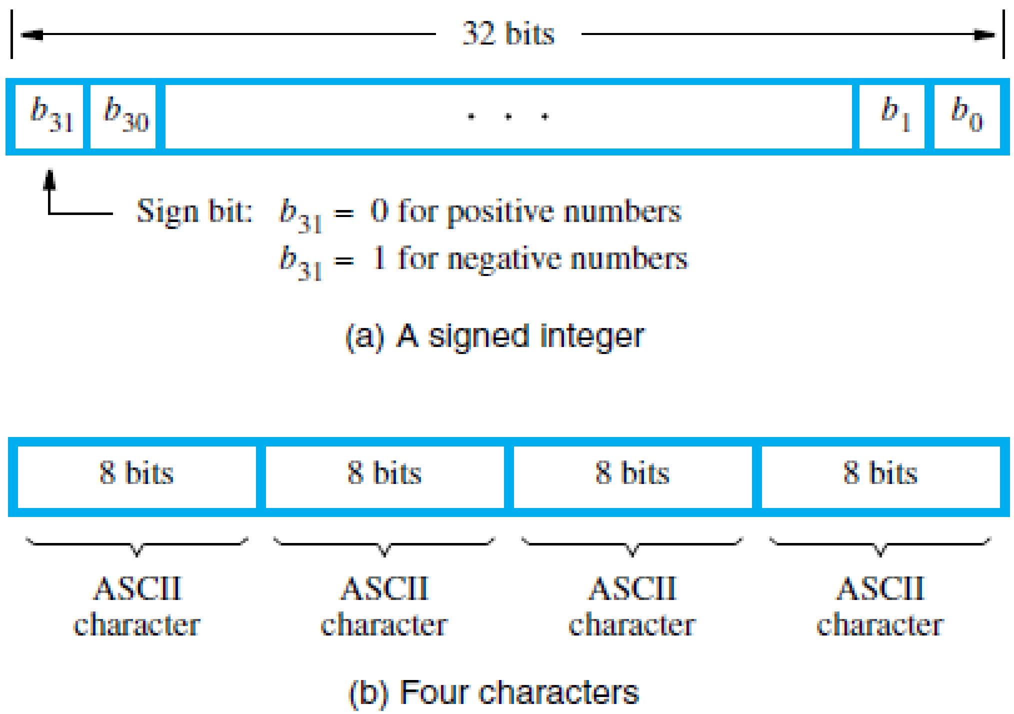
Continued…

- ✓ Memory Words



Continued…

- Word lengths vary from 8—64 bits
- ASCII--- 8 bits are used to represent a character
- A single word (length=32 bits) can store a 32-bit 2's complement number
- A single word (length=32 bits) can store 4 characters



Continued…

- Why do we need addresses?
- To access a word/or a byte we need distinct names or addresses
- _____ bit address is required to address 2^24 locations
- Range of addresses? Address space?
- Suppose if the memory of a machine has 2147483648 words (1 word=2 bytes) then _________ bit address is required to address all the locations?
- 1 KiloBytes=1K=1024 bytes=2^10
- 1 MegaBytes=1M=1024K=2^10*2^10=2^20
- 1 GigaBytes=1G=1024M=2^10*2^20=2^30
- 1 TeraBytes=1T=1024G=2^10*2^30=2^40

Continued…

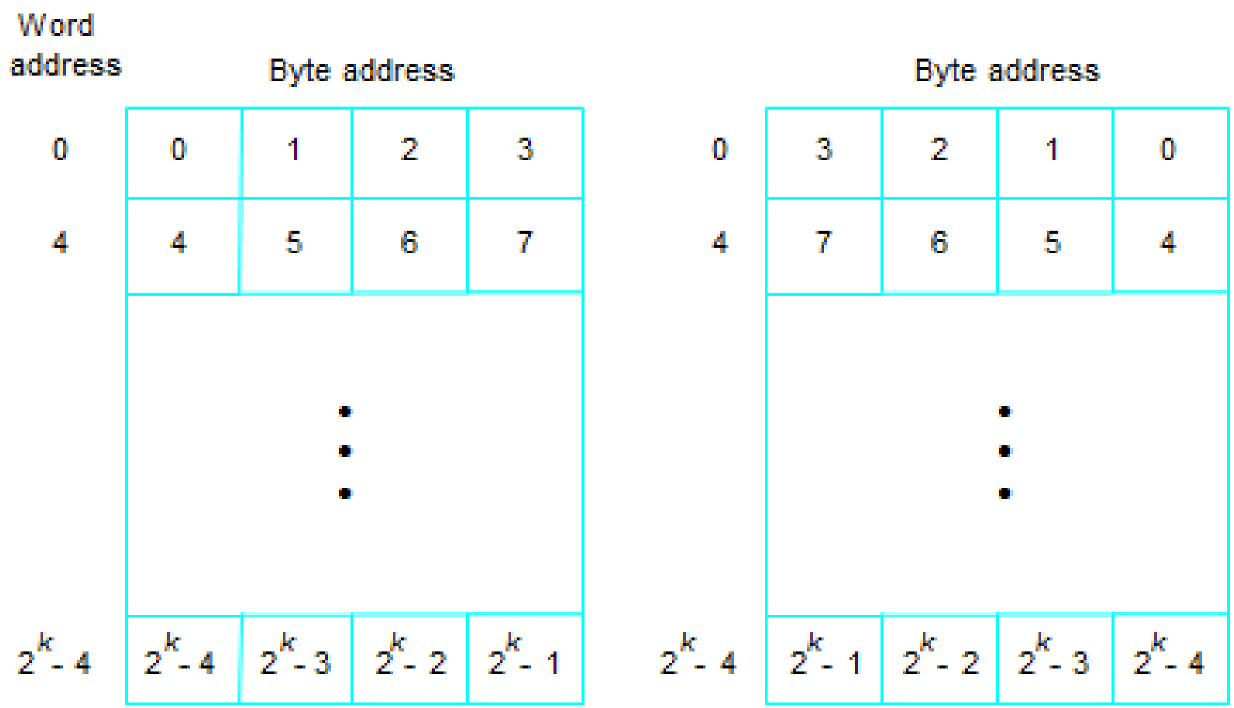
- Byte Addressability:
- 1 byte = 8 bits always
- But not that 1 word=2 bytes always
- It depends on word length
- Word length—16
- 1 word=2 bytes
- Word length---32
- 1 word=4 bytes
- Word length—64
- 1 word= 8 bytes
- Rather addresses are assigned for every byte. It is called Byte addressability

Continued…

- ✓ Big Endian and Little Endian Assignment:
- ✓ Let's say 1 word=2 bytes i.e, 1 word=16 bits
- ✓ ALU is 16-bit and Registers—16 bits
- ✓ EX: 1
- ✓ How this ABCD**H** will be stored in memory in Big Endian?
- ✓ CD which is the lower order byte value is moved to the higher order byte address of that word and AB is moved to lower order byte address of that word
- ✓ CD is stored in address 0001 and AB in address 0000
- ✓ How this ABCD**H** will be stored in memory in Little Endian?
- ✓ AB in Higher order byte address and CD in lower order byte address
- ✓ AB in 0001 and CD in 0000
- ✓ In both addresses for words are not going to change

Continued…

- ✓ Big Endian and Little Endian Diagrams:



(a) Big-endian assignment

(b) Little-endian assignment

Continued…

- Word Alignment:
- When a word is said to have aligned in memory?
- When they begin at addresses that are multiple of the no of bytes that make up a word
- If not they are unaligned
- Ex 1: 1 word=4 bytes
- Then words begin at addresses that are multiples of 4 i.e., 0,4,8,12…
- Word boundaries
- Ex 2: 1 word=8 bytes How about word boundaries
- 0,8,16…. (Multiple of 8)

Continued…

- ✓ Accessing numbers, characters, character strings:
- ✓ A number usually occupies a word
- ✓ By giving word address a number can be accessed
- ✓ A character usually occupies a byte
- ✓ By giving byte address a character can be accessed
- ✓ About character string?
- ✓ Beginning---- we can give the byte address of the first character
- ✓ Ending----- 1. It is better to have a control character after the actual string that says “End of String” ($)
- ✓ 2. A separate loc that contains the length of string

Continued…

- ***Memory Operations:***
  - ✓ 2 basic operations: Load (Read/Fetch), Store(Write)
  - ✓ Load operation: transfers copy of contents of ML to processor
  - ✓ The Memory contents are not changed
  - ✓ For load operation, processor sends address and read control signal
  - ✓ Store operation: transfers an item from processor to ML
  - ✓ The memory contents are changed
  - ✓ For store operation, processor sends address ,data and write control signal

Continued…

❖ ***Instructions and Instructions Sequencing:***

- ✓ A computer must have ins for performing 4 types of operations
- ✓ Data transfer between memory and processor registers
- ✓ A/L operations on data
- ✓ Program sequencing and control
- ✓ I/O transfers
- ✓ Some notations are required

Continued…

- Register Transfer notation:
- Possible locations in transfers may be memory loc, registers, registers in I/O
- How to refer to the content of a loc?
- By enclosing the name by square brackets
- Ex 1: R1<-[LOC]
- Meaning?
- Ex 2: R3<-[R1]+[R2]
- Meaning?
- This is Register Transfer notation

Continued…

- ✓ Assembly Language notation:
- ✓ We need another notation to represent machine ins
- ✓ Assembly Language notation
- ✓ Ex 1: Load R1, LOC
- ✓ Ex 2: Add R3, R1, R2

Difference between RISC and CISC Instruction sets

- RISC:
- Each instruction occupies exactly one word in memory
- All operands involved in an arithmetic or logic operation must either be in processor registers
- Memory operands are accessed only using Load and Store instructions
- CISC:
- Instructions may span more than one word of memory
- Not all operands involved in an arithmetic or logic operation must either be in processor registers
- Move in place of Load and Store instructions

Consider the statement

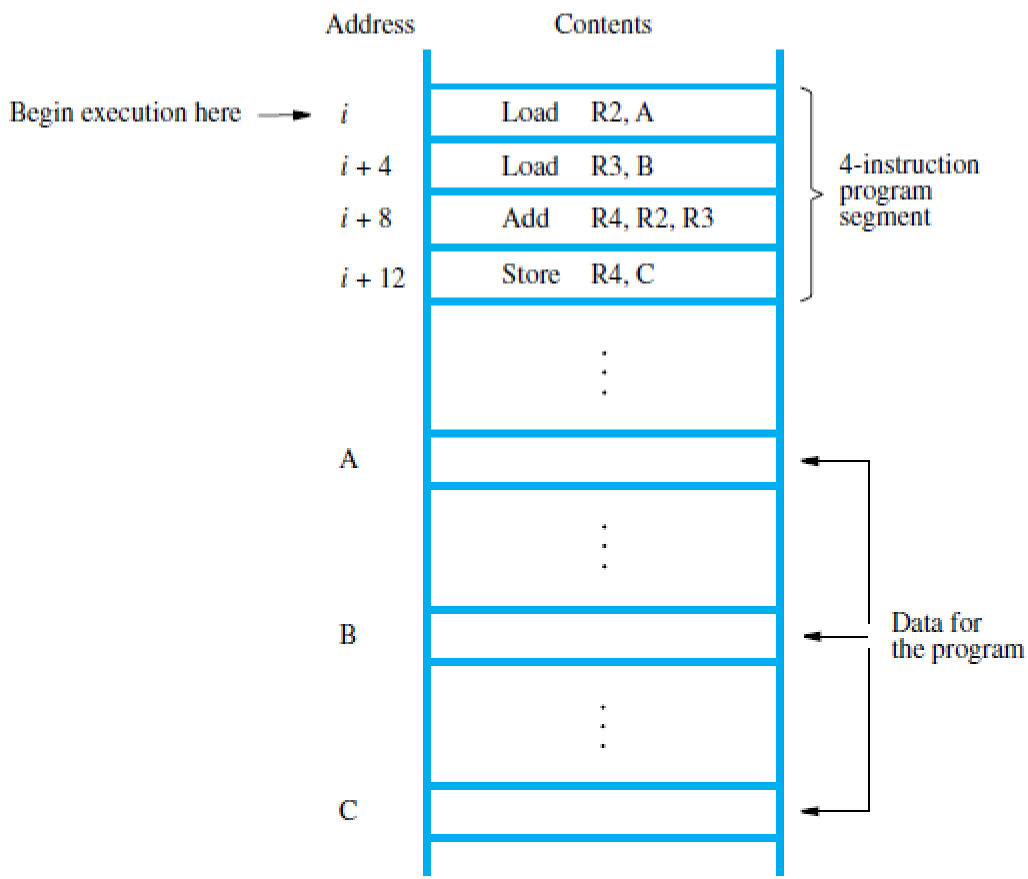
C = A + B

- ✓ RTN is $C \leftarrow [A] + [B]$
- ✓ RISC program:

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

Instruction Execution and Straight Line Sequencing

✓ Program segment for C←[A] + [B],



Continued…

- Addresses of 4 instructions are i, i+4, i+8, i+12
- Memory is byte addressable and 1 word=4 bytes.
- How the program is executed?
- To begin execution, the address of first instruction must be placed in PC
- With this instructions are fetched and executed one after the other
- Straight Line Sequencing
- During execution of every instruction PC is incremented by 4 to point to next instruction to be fetched and executed
- How an instruction is executed?

Continued…

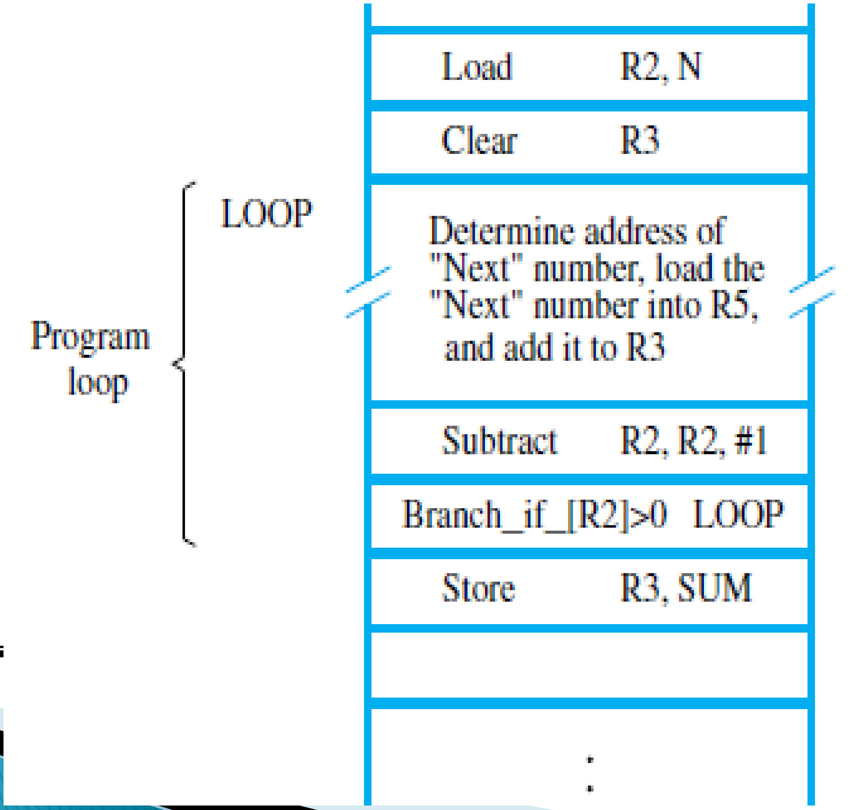
- 2 phases
- Instruction fetch: Instruction is fetched from memory location and placed in IR of processor
- Instruction execution: Instruction is examined for operation and it is performed
- PTO

Continued…

- Branching:
- Problem: Task of adding a list of n numbers
- 1: Straight Line program

i	Load	R2, NUM1
$i + 4$	Load	R3, NUM2
$i + 8$	Add	R2, R2, R3
$i + 12$	Load	R3, NUM3
$i + 16$	Add	R2, R2, R3
	⋮	
$i + 8n - 12$	Load	R3, NUMn
$i + 8n - 8$	Add	R2, R2, R3
$i + 8n - 4$	Store	R2, SUM
	⋮	

Continued…

- Is this a good approach?
- 2: Instead of this long list, Let's put add instruction in a program loop
- 

Continued…

- Comment---addresses of numbers, N, Sum
- Loop:
- Starts at LOOP
- Ends at Branch>0
- Each time thru loop address of the next number is determined , entry fetched and added to R3
- How ?
- Addressing modes
- N contains number of numbers to be added
- Moved to R2, R2 is used as counter that determines no of times loop is executed
- R2 is decremented by 1 each time in the loop
- Loop is repeated till R2 becomes 0

Continued…

- Branch>0 LOOP
- Branch instruction
- Makes the control go to branch target
- How?
- Up on branching to branch target, PC is loaded with the address of the first instruction of loop
- Branch>0 LOOP is a conditional branch
- If satisfied, PC is loaded with address of first instruction of loop
- If not, PC is incremented in normal way
- How long it repeats? (Branch>0 LOOP)

Continued…

- RISC Addressing modes:
- Immediate
- Register
- Absolute
- Register indirect
- Index: Index
- Base with Index
- Base with Index and Displacement

Continued…

- ✓ CISC Addressing modes:
- ✓ Immediate
- ✓ Register
- ✓ Absolute
- ✓ Register indirect
- ✓ Index: Index
- ✓ Base with Index
- ✓ Base with Index and Displacement
- ✓ Program Counter Relative
- ✓ Additional modes: Autoincrement and Autodecrement

Continued…

- Program operates on data in memory
- Data is organized into several structures---Data Structures
- Addressing modes: Diff ways in which location of an operand is specified
- PTO

Continued…

- Immediate mode:
- Operand is given explicitly
- When the value is not going to change
- Application: Constants are represented using this mode
- Syntax: #Value
- Ex:
- Add R4, R6, #200
- Register mode:
- Name of the register that has the operand is given
- Application: Variables are represented using this mode
- Syntax: Ri
- Effective address: Address from where operand can be determined
- EA: Ri
- Ex:
- Add R4, R2, R3

Continued…

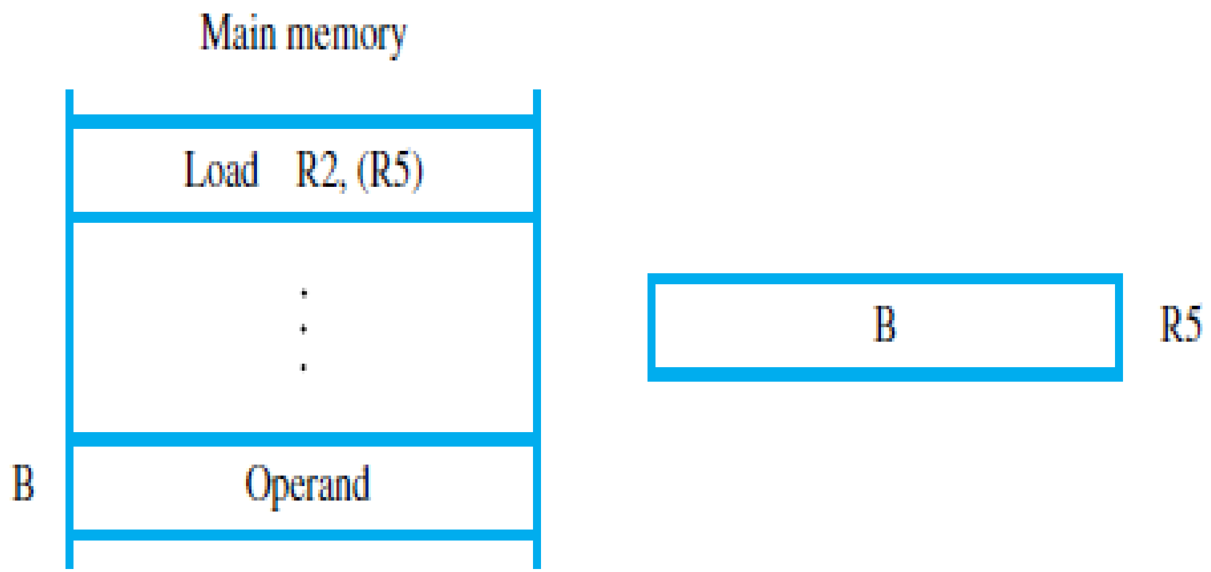
- Absolute mode:
- Name of the memory location that has the operand is given
- Application: Variables are represented
- Syntax: LOC
- EA: LOC
- Ex:
- Load R2, NUM1

Continued…

- *Indirection and Pointers:* (Register indirect)
- Operand or its address is not given directly
- The address of operand is in a register
- Syntax: Register indirect: (Ri)
- EA: Register indirect: [Ri]
- Register or that has address is called pointer
- Ex: Next slide

Continued…

✓



Continued…

- Register indirect:
- First processor reads the contents of R1 and then uses B in R1 as address to obtain the operand
- APP: To add successive numbers in the list
- Next Slide

Continued…

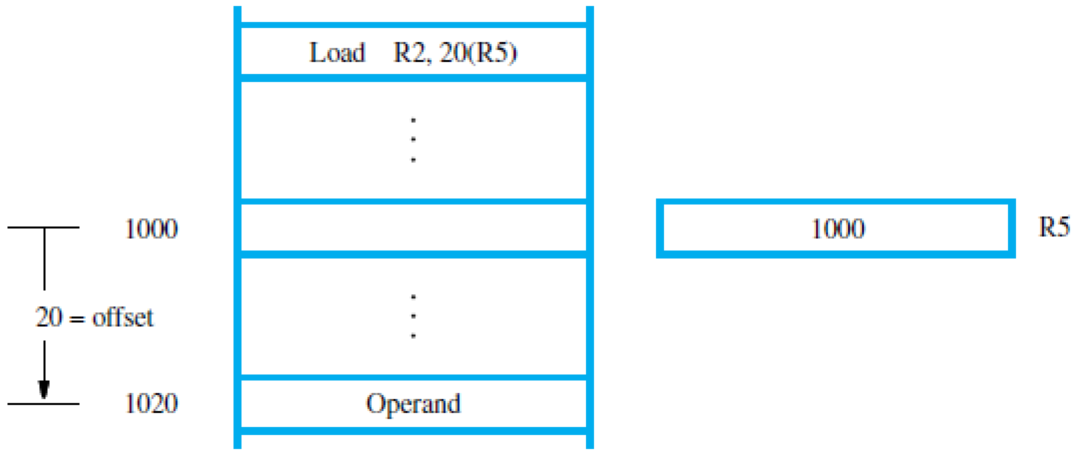
✓

```
        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
```

Continued…

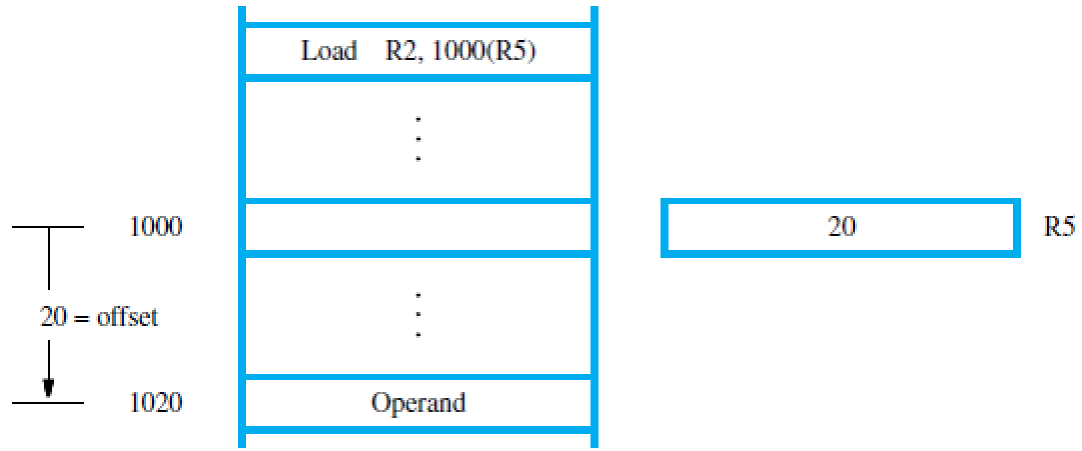
- *Indexing and Arrays :*
- Operand or its address is not given directly
- The address of the operand is the content of register(s) together with a displacement value
- APP: To deal with arrays and lists
- Syntax: Simple Index: X(Ri)
- EA: X+[Ri]

Continued…



(a) Offset is given as a constant

Continued…



Continued…

- ✓ PE: Calculating sum of Test Score 1, Test Score 2, Test Score 3 of n students and place results in Sum1,Sum2,Sum3

N	n
LIST	Student ID
LIST+4	Test1
LIST+8	Test2
LIST+12	Test3
…….	Student ID
……	Test1
…..	Test2
……	Test3

Continued…

- This should be thought of as 2D array
- Each row contains entry for a student
- Columns give IDs and Test scores
- Program in next slide

Continued…

```
         Move              R2, #LIST
         Clear             R3
         Clear             R4
         Clear             R5
         Load              R6, N
LOOP:    Load              R7, 4(R2)
         Add               R3, R3, R7
         Load              R7, 8(R2)
         Add               R4, R4, R7
         Load              R7, 12(R2)
         Add               R5, R5, R7
         Add               R2, R2, #16
         Subtract          R6, R6, #1
         Branch_if_[R6]>0  LOOP
         Store             R3, SUM1
         Store             R4, SUM2
         Store             R5, SUM3
```

Continued…

- Content of R4 is changing between records not within records
- In the example student ids are the reference points
- Base with Index:
- Address of operand is the content of Register Ri together with content of register Rj
- Syntax: (Ri,Rj)
- EA: [Ri]+[Rj]
- Actually, One will be an index and other will be a base register

Continued…

- Base index with displacement
- Address of operand is sum of contents of 2 registers together with a displacement
- Syntax: X(Ri,Rj)
- EA: X+[Ri]+[Rj]
- PTO

Continued…

- CISC Instruction Sets:
- Most arithmetic and logic instructions use the two-address format
- Operation destination, source
- Ex1:
- Add B, A ; B $\leftarrow$ [A] + [B]
- Ex2:
- Move C, B
- Add C, A ; C$\leftarrow$[A] + [B]
- The general form of the Move instruction is
- Move destination, source

Continued…

- *Relative:*
- EA of operand is relative to the content of index register
- Here the difference is EA of operand is relative to content of PC
- Syntax: X(PC)
- EA: X+[PC]
- X is a signed number because the branch target may be above or below branch instruction

Continued…

- *Additional Modes:*
- *Auto increment, Auto decrement modes*
- Autoincrement: EA of operand is the contents of a register
- After the operand is accessed, the content of register is automatically incremented by no of bytes that make up a word if I want successive words to be accessed
- If I want successive bytes to be accessed then the content of the register will be incremented by 1
- Syntax: (Ri)+
- EA: [Ri] and Ri is incremented

Continued…

- Autodecrement mode:
- EA: It is obtained by decrementing the content of register first and then taking the decremented value as address of the operand
- Words? Bytes?
- Syntax: -(Ri)
- EA: Decrement Ri and then EA=[Ri]
- These modes can be used to implement stack

Continued…

- Condition Codes:
- Info about results of various A/L operations must be kept track off
- Whether result has generated carry, overflow, result is zero/negative
- They are recorded in individual bits called condition code flags
- These flags are grouped in a special register called condition code register/flag register/status register
- These flags take values 0 or 1

Continued…

- Flags:
- N-----set to 1 if result is negative
- Z-----Set to 1 if result is 0
- V----- set to 1 if result has overflown
- C----- Set to 1 if a carry results out of A/L operation
- These flags are affected as a result of A/L operations
- Branch>0 tests N and Z flags to cause a branch

Continued…

```
        Move       R2, N
        Clear      R3
        Move       R4, #NUM1
LOOP:   Add        R3, (R4)+
        Subtract   R2, #1
        Branch>0   LOOP
        Move       SUM, R3
```

Continued…

	Move	R2, #AVEC	R2 points to vector A.
	Move	R3, #BVEC	R3 points to vector B.
	Load	R4, N	R4 serves as a counter.
	Clear	R5	R5 accumulates the dot product.
LOOP:	Load	R6, (R2)	Get next element of vector A.
	Load	R7, (R3)	Get next element of vector B.
	Multiply	R8, R6, R7	Compute the product of next pair.
	Add	R5, R5, R8	Add to previous sum.
	Add	R2, R2, #4	Increment pointer to vector A.
	Add	R3, R3, #4	Increment pointer to vector B.
	Subtract	R4, R4, #1	Decrement the counter.
	Branch_if_[R4]>0	LOOP	Loop again if not done.
	Store	R5, DOTPROD	Store dot product in memory.

Figure 2.27 A RISC-style program for computing the dot product of two vectors.

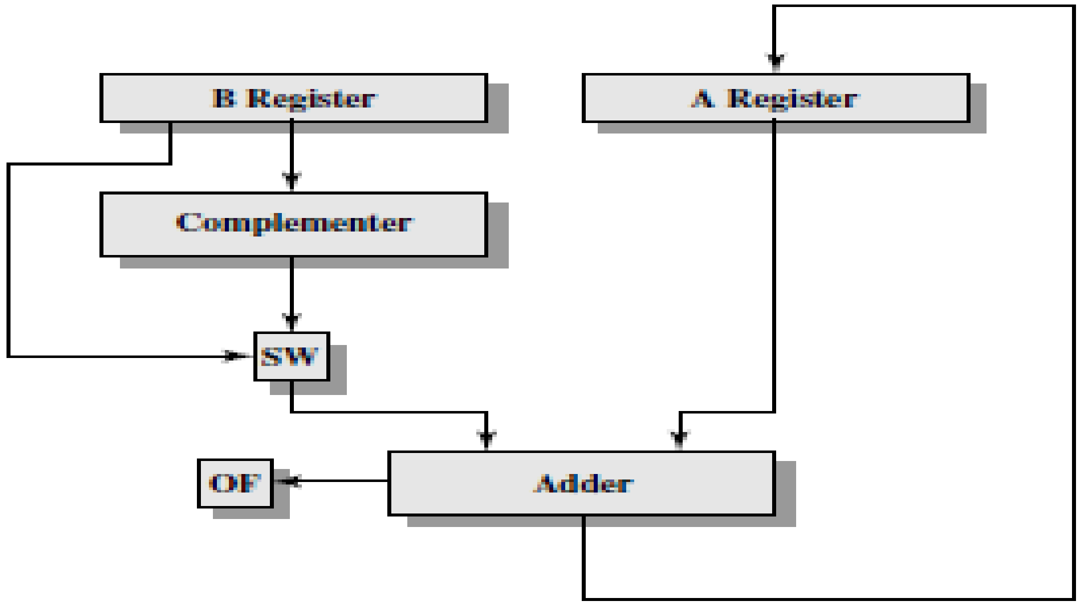
Continued…

	Move	R2, #AVEC	R2 points to vector A.
	Move	R3, #BVEC	R3 points to vector B.
	Move	R4, N	R4 serves as a counter.
	Clear	R5	R5 accumulates the dot product.
LOOP:	Move	R6, (R2)+	Compute the product of
	Multiply	R6, (R3)+	next components.
	Add	R5, R6	Add to previous sum.
	Subtract	R4, #1	Decrement the counter.
	Branch>0	LOOP	Loop again if not done.
	Move	DOTPROD, R5	Store dot product in memory.

Figure 2.28 A CISC-style program for computing the dot product of two vectors.

ARITHMETIC AND LOGIC UNIT

Hardware implementation for Addition and Subtraction:



OF = Overflow bit
SW = Switch (select addition or subtraction)

Figure 9.6 Block Diagram of Hardware for Addition and Subtraction

Multiplication

UNSIGNED INTEGERS

Multiplicand (11) **Multiplier (13)** **Product (143)**

```
      1011 X 1101
      ----------------
      1011
     0000
    1011
   1011
---------------------------
  10001111
```

C	A	Q	M		
0	0000	1101	1011	Initial values	
0	1011	1101	1011	Add	First cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second cycle
0	1101	1111	1011	Add	Third cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth cycle
0	1000	1111	1011	Shift	

(b) Example from Figure 9.7 (product in A, Q)

Figure 9.8 Hardware Implementation of Unsigned Binary Multiplication

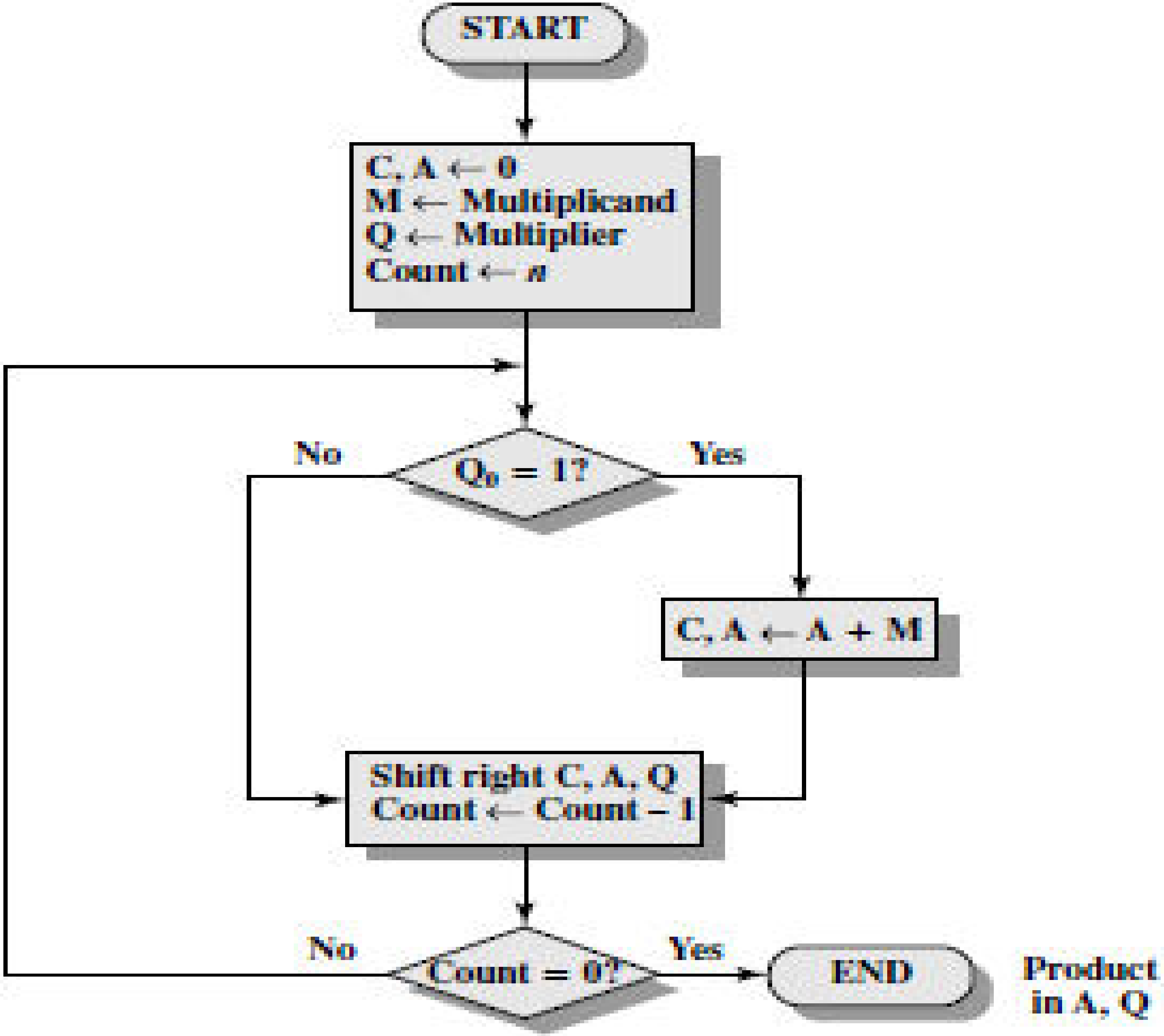
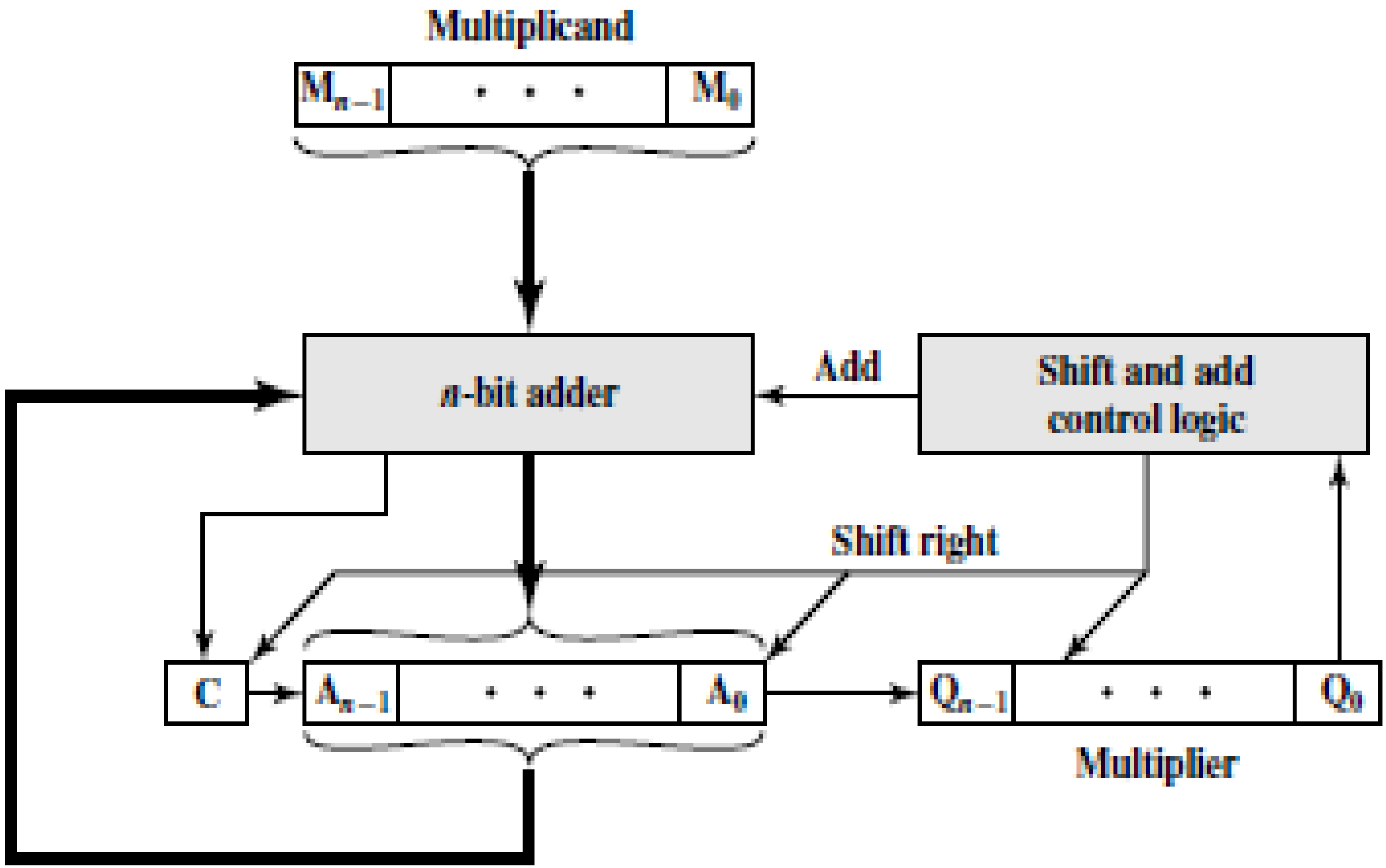


Figure 9.9 Flowchart for Unsigned Binary Multiplication


Multiplicand
M_{n-1}
M_0
n-bit adder
Add
Shift and add
control logic
Shift right
C
A_{n-1}
A_0
Q_{n-1}
Q_0
Multiplier

TWOS COMPLEMENT MULTIPLICATION

```
      1011 X 1101
     ---------------
       1011
      0000
     1011
    1011
 ---------------------------
 10001111
```

```
    1011
  × 1101
--------
00001011   1011 × 1 × 2^0
00000000   1011 × 0 × 2^1
00101100   1011 × 1 × 2^2
01011000   1011 × 1 × 2^3
--------
10001111
```

Figure 9.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Straightforward multiplication will not work if both the multiplicand and multiplier are negative.
In fact, it will not work if either the multiplicand or the multiplier is negative.

```
    1001   (9)
  × 0011   (3)
--------
00001001   1001 × 2^0
00010010   1001 × 2^1
--------
00011011   (27)
```

(a) Unsigned integers

```
    1001   (-7)
  × 0011   (3)
--------
11111001   (-7) × 2^0 = (-7)
11110010   (-7) × 2^1 = (-14)
--------
11101011   (-21)
```

(b) Twos complement integers

Figure 9.11 Comparison of Multiplication of Unsigned and 2's Complement Integers

Booth's algorithm:

Adv: Benefit of speeding up the multiplication process, relative to a more straightforward approach.

The multiplier in Reg Q and multiplicand in M are placed.

There is a 1-bit register Q_{-1} placed logically to the right of the least significant bit of the Q register.

The results of the mul will appear in A & Q regs.

A and Q_{-1} are initialized to 0.

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	A ← A − M	First cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	A ← A + M	Third cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth cycle

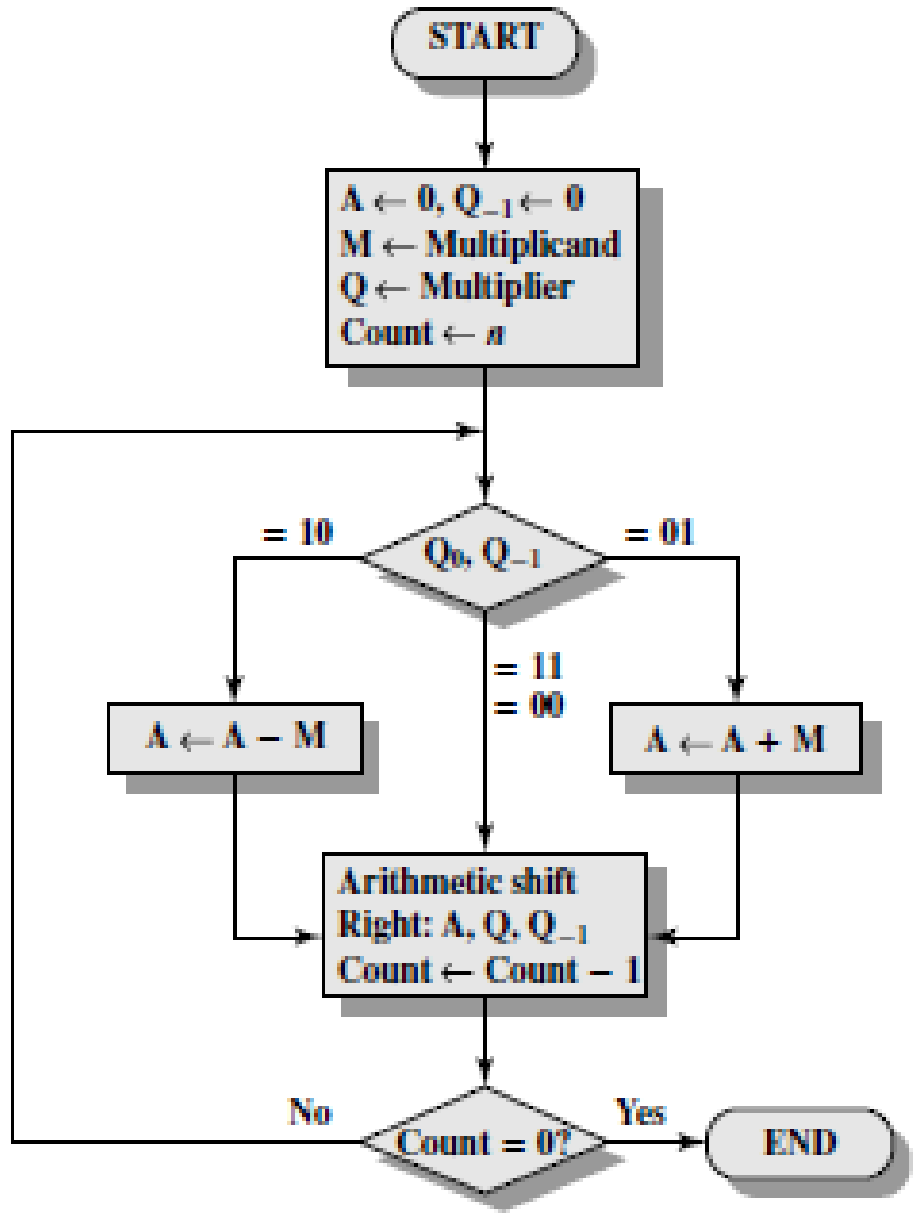


Figure 9.12 Booth's Algorithm for 2's Complement Multiplication

Consider a +ve multiplier 00011110 = $2^4 + 2^3 + 2^2 + 2^1 = 30$

Is also equal to $2^5 – 2^1$

The number of such operations can be reduced to two if we observe that

$2^n + 2^{n-1} + ... + 2^{n-K} = 2^{n+1} - 2^{n-K}$ ----Eq 1

So the product can be generated by one addition and one subtraction of the multiplicand.

This scheme extends to any number of blocks of 1s in a multiplier.

$$\begin{aligned} M \times (01111010) &= M \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= M \times (2^7 - 2^3 + 2^2 - 2^1) \end{aligned}$$

The same scheme works for a negative multiplier.
Let X be a negative number in twos complement notation. The leftmost bit of X is 1, because X is negative:

$$\text{Representation of } X = \{1x_{n-2}x_{n-3}\ldots x_1x_0\}$$

Then the value of X can be expressed as

$$X = -2^{n-1} + (x_{n-2} \times 2^{n-2}) + (x_{n-3} \times 2^{n-3}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Assume that the leftmost 0 is in the kth position. Thus, X is of the form

$$\text{Representation of } X = \{111\ldots10x_{k-1}x_{k-2}\ldots x_1x_0\}$$

Then the value of X is

$$X = -2^{n-1} + 2^{n-2} + \cdots + 2^{k+1} + (x_{k-1} \times 2^{k-1}) + \cdots + (x_0 \times 2^0) \qquad \text{----2}$$

From Equation (1), we can say that

$$2^{n-2} + 2^{n-3} + \cdots + 2^{k+1} = 2^{n-1} - 2^{k+1}$$

Rearranging,

$$-2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2^{k+1} = -2^{k+1} \qquad \text{----3}$$

Substituting Eqn 3 in eqn 2 we have

$$X = -2^{k+1} + (x_{k-1} \times 2^{k-1}) + \cdots + (x_0 \times 2^0)$$

Look at page 324, how -6 is handled with above principle.

DIVISION

Dividend = 147 Divisor = 11

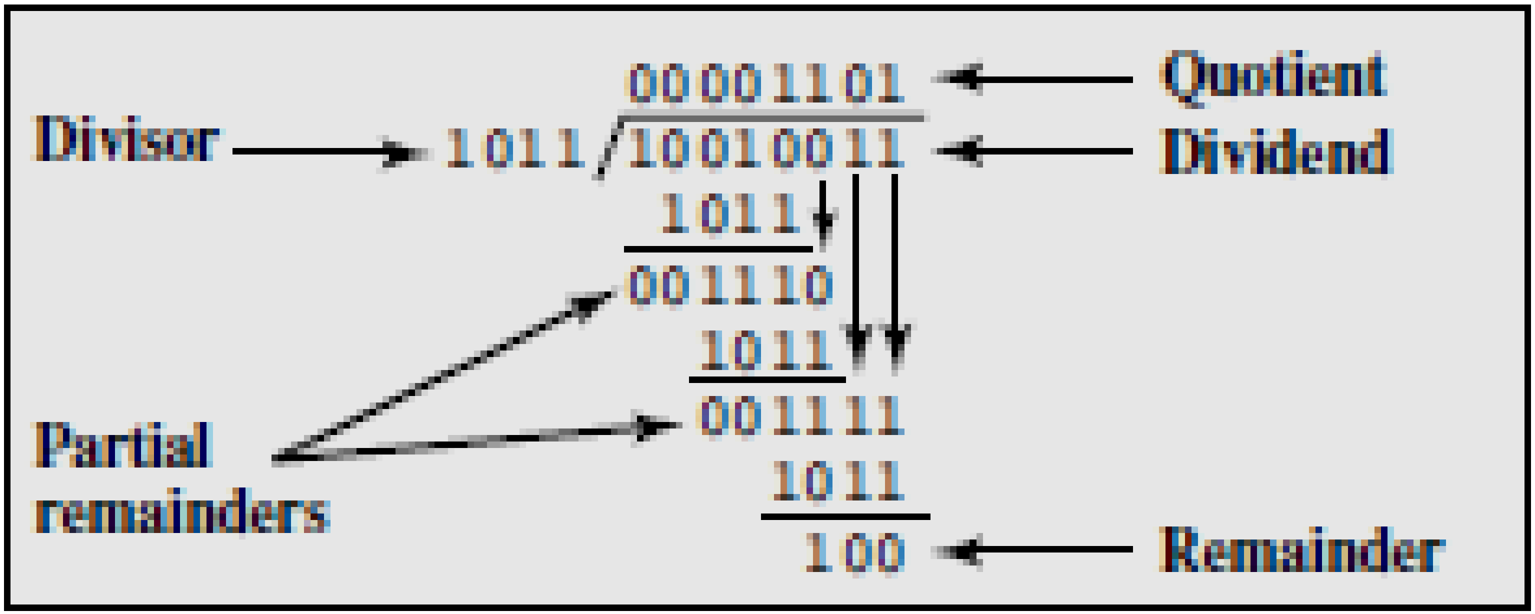


Figure 9.15 Example of Division of Unsigned Binary Integers

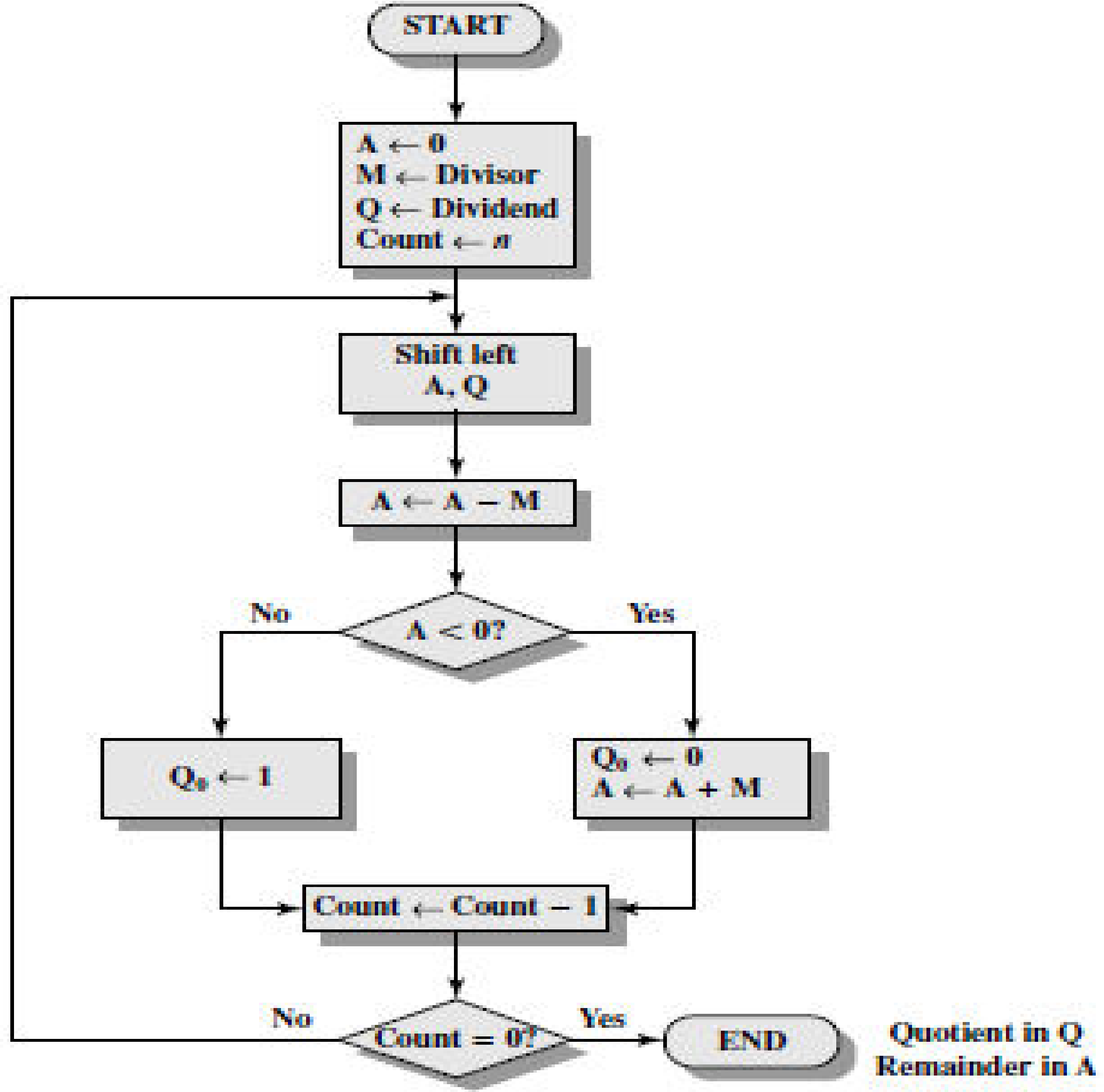


Figure 9.16 Flowchart for Unsigned Binary Division

A	Q	
0000	0111	Initial value
0000	1110	Shift
1101		Use twos complement of 0011 for subtraction
1101		Subtract
0000	1110	Restore, set $Q_0 = 0$
0001	1100	Shift
1101		
1110		Subtract
0001	1100	Restore, set $Q_0 = 0$
0011	1000	Shift
1101		
0000	1001	Subtract, set $Q_0 = 1$
0001	0010	Shift
1101		
1110		Subtract
0001	0010	Restore, set $Q_0 = 0$

Figure 9.17 Example of Restoring Twos Complement Division (7/3)

To deal with -ve numbers, we recognize that

$$D = Q * V + R$$

Here D – dividend V – Divisor Q – Quotient R – Remainder

Consider eg. of int div with all possible combinations of signs of D and V:

D = 7	V = 3	Q = 2	R = 1
D = 7	V = -3	Q = -2	R = 1
D = -7	V = 3	Q = -2	R = -1
D = -7	V = -3	Q = 2	R = -1

Note that the signs of Q and R are easily derivable from the signs of D and V. Specifically,

sign(R) = sign(D)

sign(Q) = sign(D) * sign(V)

Hence, 2's complement division is

-to convert the operands into unsigned values and,

-at the end, to account for the signs by complementation where needed.