# 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.

# COURSE 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

# COURSE OUTCOMES

| At the end of this course, the student should be able to: | | No. of Hours | Marks |
|---|---|---|---|
| CO1 | Describe the functionalities of the various units of computers and the instruction set architecture. | 10 | 20 |
| CO2 | Appreciate the hardware implementation of addition, subtraction, multiplication, and division and perform arithmetic operations. | 7 | 16 |
| CO3 | Design the control unit for simple algorithms. | 10 | 20 |
| CO4 | Explain basics of memory system such as cache memories, mapping functions, replacement algorithms and virtual memory concept and design simple memory systems. | 10 | 20 |
| CO5 | Outline the I/O handling techniques and realize the improvement in performance using the concepts of pipelining and parallel processing. | 11 | 24 |
| Total hours/ Marks | | 48 | 100 |

# 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

- 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.

# 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

MODULE 1

# 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 accessed via 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

<u>Instruction Set</u>

LOAD a number from RAM into the CPU

ADD two numbers together

STORE a number from the CPU back out to RAM

# 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

# 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
  i. Add R4, R2, R3
  ii. 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.

# TOPICS COVERED FROM

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

# 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

# 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 $B = b_{n-1} \ldots b_1 b_0$,
      - 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

| $b_3b_2b_1b_0$ | Value in decimal |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | 10 |
| 1011 | 11 |
| 1100 | 12 |
| 1101 | 13 |
| 1110 | 14 |
| 1111 | 15 |

# 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

| *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 |

# 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 | $b_3\ b_2b_1b_0$ | $b_3\ b_2b_1b_0$ | 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<br>( 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 | $b_3\ b_2b_1b_0$ | $b_3\ b_2b_1b_0$ | 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 ➔ 4294967296-1
  - 32 bits = 4 bytes (size of int)
- Hence, subtraction is not recommended.
  ```

# TOPICS COVERED FROM

- Textbook 1:
  - Chapter 1: 1.4.1

# COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

# SIGN AND MAGNITUDE: ARITHMETIC OPERATIONS

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

  0 101 +

  1 010

  111 (7) wrong answer!!

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

  1 110 -

  0 001

  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→ | 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
+7→     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

# 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

# FLOATING-POINT NUMBERS

- A binary floating-point number is 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

# IEEE STANDARD FLOATING-POINT FORMATS (32 BIT)

- 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

$$\text{Value represented} = \pm 1.M \times 2^{E'-127}$$

# IEEE STANDARD FLOATING-POINT FORMATS (32 BIT)

- Example:
  - Sign bit: 0, hence +ve number
  - Mantissa, M: 1.00101000000000000000000
  - Exponential, E: E'-127, ➔(E'→$00101000_2$= $40_{10}$)

    E: 40-127= -87,

    E: $2^{-87}$

| | | |
|---|---|---|
| 0 | 0 0 1 0 1 0 0 0 • 0 0 1 0 1 0 . . . | 0 |

Value represented = $1.001010\ldots0 \times 2^{-87}$

- Actual binary number:
  - 0.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000100101

- Corresponding decimal value:
  - 0.0000000000000000000000000007472090494253423820859892970358551167464611353352665901118408203125

- In the value represented, what about the digit found to the left side of the decimal point?
  - always be equal to 1
  - can be left out in IEEE floating-point representation

# STEPS TO CONVERT 32-BIT REPRESENTATION TO DECIMAL.

1. Obtain the mantissa and rewrite the value as V=1.M
2. To obtain E
   i. Convert E' to its equivalent decimal value
   ii. E= E'-127 ➔$2^E$
3. Move point in V towards left or right based on E
   i. Move right if +ve
   ii. Move left if –ve
   iii. Digits before point is the integral part and after is the fractional part.
4. Convert the integral part of binary to decimal equivalent
   i. Multiply each digit separately from left side of point till the first digit by $2^0$, $2^1$, $2^2$, … respectively.
   ii. Find the sum of all the products obtained in step 1.i.
5. Convert the fractional part of binary to decimal equivalent
   i. Divide each digit separately from right side of point till the end by $2^1$, $2^2$, $2^3$, … respectively.
   ii. Find the sum of all the products obtained in step 2.i.
6. Add both integral and fractional part to obtain decimal number.

# 32-BIT REPRESENTATION TO DECIMAL: EXAMPLE

- IEEE754 32-bit format: 0 10000001 01000110011001100110011
  S E' M
- S=0, hence +ve number
- E'= $10000001_2 = 129_{10}$
  - E = E'-127 =129-127

    E = 2
- Value represented = 1.01000110011001100110011 $\times 2^2$

  = 101.00011001100110011

  = 5 . 1

# DECIMAL TO 32-BIT REPRESENTATION

- 40.15625
  - S=0, since it is a positive number
  - Mantissa, M,
    - 40➔ $101000_2$ 0.15625➔ $.00101_2$
    - $=101000.00101_2$ If point moved towards left, then +ve exponent else -ve exponent
    - $=1.\mathbf{0100000101} \times 2^5$ Ignore 1 before decimal point
    - Therefore, M=**0100000101**0000000000000 Fill the remaining positions in the right with zeroes
  - Exponent, E', in excess-127 representation:
    - E'=E+127
    - =5+127
    - =132
    - $E'=10000100_2$
  - 32-bit representation:
    - 0 10000100 01000001010000000000000

# STEPS TO CONVERT DECIMAL TO 32-BIT REPRESENTATION

A. Sign bit, S:
   i. If positive, the first bit will be a 0
   ii. If negative, the first bit will be a 1.

B. Mantissa, M:
   i. Divide the integral part by 2 to get its binary equivalent, I (remainder in bottom-up)
   ii. Multiply the fractional part by 2 to get its binary equivalent, F (value before point in top-down)
   iii. Represent it as I.F
   iv. Adjust the point to obtain 1.M
   v. M must be 23-bit long. Fill the remaining bits in vector with zeroes

# STEPS TO CONVERT DECIMAL TO 32-BIT REPRESENTATION

C. Exponent, E', in "Excess 127 form":

i. Count the number of places the binary point needs to be moved until a single digit of 1 sits by itself on the left side of the binary point.

   a. Point moved towards left count is positive else it is negative.

ii. Add 127 to your result above.

   a. 8-bit binary numbers can range from 0 to 255,

   b. exponents in single precision format can range from -126 to +127, that is from $2^{-126}$ to $2^{127}$ or,

   c. approximately, $10^{-38}$ to $10^{38}$ in size.

   d. In "excess 127 form" negative exponents range from 0 to 126, and positive exponents range from 128 to 255.

   e. 127 represents a power of zero.

iii. Translate the sum (which will always be a positive value after adding 127) into binary form.

iv. This should be represented using 8 bits, so zeros may be added to the left side to ensure a string length of 8 bits.

v. This 8-bit string represents the exponent.

# DECIMAL TO 32-BIT REPRESENTATION

- Represent -0.09375 in IEEE754 format
  - **S=1**, since it is a negative number
  - **Mantissa, M:** 0.09375 to binary
    - 0.09375 x 2 = 0.1875 0 (remember, read downwards)
    - 0.1875 x 2 = 0.375 0
    - 0.375 x 2 = 0.75 0
    - 0.75 x 2 = 1.50 1
    - 0.50 x 2 = 1.00 1
  - $= 0.00011_2$
  - $=1.\mathbf{1} \times 2^{-4}$ If point moved towards left, then +ve exponent else -ve exponent. Ignore 1 before point
  - Therefore, M=10000000000000000000000 Fill the remaining positions in the right with zeroes
  - **Exponent, E',** in excess-127 representation:
    - E'=E+127
    - =-4+127
    - =123
    - $E'=1111011_2$= 01111011
  - -0.09375 will be represented in IEEE754 format as
    - 1 01111011 10000000000000000000000

# IEEE STANDARD FLOATING-POINT FORMATS (64 BIT)

- 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
- The 11-bit excess-1023 exponent E' has the
  - range $1 \leq E' \leq 2046$ for normal values, with 0 and 2047 used to indicate special values,
  - The actual exponent E is in the range $-1022 \leq E \leq 1023$
  - providing scale factors of $2^{-1022}$ to $2^{1023}$ (approximately $10^{\pm 308}$)

$$\text{Value represented} = \pm 1.M \times 2^{E'-1023}$$

# IEEE754 FORMAT: FLOATING-POINT NUMBER

- The full 24-bit string, B, of significant bits, called the mantissa, always has a leading 1, with the binary point immediately to its right. Therefore, the mantissa
  - B = 1.M

    $= 1.b_{-1}b_{-2}\ldots b_{-23}$ has the value

  $V(B) = 1 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdot \cdot \cdot + b_{-23} \times 2^{-23}$

- By convention, when the binary point is placed to the right of the first significant bit, the number is said to be normalized.
- Note that the base 2, of the scale factor and the leading 1 of the mantissa are both fixed. They do not need to appear explicitly in the representation.

# IEEE754 FORMAT: FLOATING-POINT NUMBER

excess-127 exponent

| | | |
|---|---|---|
| 0 | 1 0 0 0 1 0 0 0 | • 0 0 1 0 1 1 0 . . . |

(There is no implicit 1 to the left of the binary point.)

$$\text{Value represented} = +0.0010110\ldots \times 2^{9}$$

(a) Unnormalized value

| | | |
|---|---|---|
| 0 | 1 0 0 0 0 1 0 1 | • 0 1 1 0 . . . |

$$\text{Value represented} = +1.0110\ldots \times 2^{6}$$

(b) Normalized version

# SPECIAL VALUES: 32-BIT REPRESENTATION

| Sl.No. | E' | M | Meaning |
|---|---|---|---|
| 1. | =0 | = 0 | value 0 is represented |
| 2. | =255 | = 0 | value ∞ is represented |
| 3. | =0 | ≠ 0 | denormal numbers are represented. Their value is $\pm 0.M \times 2^{-126}$. There is no implied one to the left of the binary point, and M is any nonzero 23-bit fraction |
| 4. | =255 | ≠ 0 | the value represented is called Not a Number (NaN). A NaN represents the result of performing an invalid operation such as 0/0 or $\sqrt{-1}$ |

# EXCEPTIONS

- In conforming to the IEEE Standard, a processor must set exception flags if any of the following conditions arise when performing operations
  - underflow,
  - overflow,
  - divide by zero,
  - Inexact: name for a result that requires rounding in order to be represented in one of the normal formats
  - Invalid: exception occurs if operations such as 0/0 or $\sqrt{-1}$ are attempted
- When an exception occurs, the result is set to one of the special values.

# ADDITION EXAMPLE- BINARY

- A=96.625 + B=12.125
- **Step i:** Convert A to binary representation
  - 1100000.101 ➔After normalizing we get 1.100000101 X $2^6$
  - E'= 6+127=133 =10000101
  - In IEEE 32-bit format: A= 01000010110000010100000…….
- **Step ii:** Convert B to binary representation
  - 1100.001 ➔After normalizing we get 1.100001 X $2^3$
  - E'= 3+127=130 = 10000010
  - In IEEE 32-bit format: B= 01000001010000100000……
- **Step 1:** Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents (Shift point to left).
  - Shift the mantissa of smaller number, B, to the right by 3 bits we get
    - 1.10000100000 ---Original mantissa (with hidden bit considered)
    - 0.11000010000 ---shifting by 1 bit
    - 0.011000010000 ---shifting by 2 bits
    - 0.001100001000 ---shifting by 3 bits

# ADDITION EXAMPLE- BINARY

- **Step 2:**
  - Set the exponent of the result equal to the larger exponent
    - 10000101 (exponent of A)
- **Step 3:**
  - Perform addition on the mantissas and determine the sign of the result

```
  1.100000101000000  +
  0.001100001000000
  1.101100110000000
```

- **Step 4:**
  - Normalize the resulting value, if necessary
    - Result is already normalized
- In 32-bit format: 0 10000101 101100110000
- which is 108.75 in decimal

# ADD/SUBTRACT RULE

1. Choose the number with the smaller exponent
2. Shift its mantissa right to the number of steps equal to the difference in exponents. (Shift point to left)
3. Set the exponent of the result equal to the larger exponent.
4. Perform addition/subtraction on the mantissas and determine the sign of the result.
5. Normalize the resulting value, if necessary.

# ADDITION EXAMPLE- BINARY

- 0.25 = 0 01111101 00000000000000000000000 + 100 = 0 10000101 10010000000000000000000
- **Step 1:** align radix points
  - shifting the mantissa LEFT by 1 bit DECREASES THE EXPONENT by 1 and RIGHT by 1 INCREASES THE EXPONENT by 1
  - we want to shift the mantissa right, because the bits that fall off the end should come from the least significant end of the mantissa
    - choose to shift the .25, since we want to increase it's exponent.
    - shift by

```
 10000101
-01111101
 00001000   (8) places.
```

00000000000000000000000 (original value)
10000000000000000000000 (shifted 1 place) (note that hidden bit is shifted into msb of mantissa)
01000000000000000000000 (shifted 2 places)
00100000000000000000000 (shifted 3 places)
00100000000000000000000 (shifted 4 places)
00001000000000000000000 (shifted 5 places)
00000100000000000000000 (shifted 6 places)
00000010000000000000000 (shifted 7 places)
00000001000000000000000 (shifted 8 places)

# ADDITION EXAMPLE- BINARY

- Step 2: add (don't forget the hidden bit for the 100)

  1.10010000000000000000000 (100) +

  0.00000001000000000000000 (.25)

  1.10010001000000000000000

- Step 3: normalize the result (get the "hidden bit" to be a 1)
  - Result is already normalized

- In 32-bit format: 0 10000101 10010001000000000000000

# SUBTRACTION EXAMPLE- BINARY

- A=96.625 - B=12.125
- Convert A to binary representation
  - 1100000.101 ➔After normalizing we get 1.100000101 X $2^6$
  - E'= 6+127=133 =10000101
  - In IEEE 32-bit format: 01000010110000010100000…….
- Convert B to binary representation
  - 1100.001 ➔After normalizing we get 1.100001 X $2^3$
  - E'= 3+127=130 = 10000010
  - In IEEE 32-bit format: 01000001010000100000……
- **Step 1:**
- Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
  - A=0 10000101 10000010100000……. B=0 10000010 10000100000……
- Shift the mantissa of smaller number, B, to the right by 3 bits we get
  - 1.10000100000 ---Original mantissa (with hidden bit considered)
  - 0.11000010000 ---shifting by 1 bit
  - 0.011000010000 ---shifting by 2 bits
  - 0.001100001000 ---shifting by 3 bits

# SUBTRACTION EXAMPLE- BINARY

- **Step 2:**
  - Set the exponent of the result equal to the larger exponent
    - 10000101 (exponent of A)
- **Step 3:**
  - Perform subtraction on the mantissas and determine the sign of the result

```
 1.10000010100000  -
 0.00110000100000
 1.01010010000000
```

- **Step 4:**
  - Normalize the resulting value, if necessary
    - Result is already normalized
- In 32-bit format: 0 10000101 01010010000000
- which is 84.5 in decimal

# SUBTRACTION EXAMPLE- BINARY

- 0.25 = 0 01111101 00000000000000000000000 + 100 = 0 10000101 10010000000000000000000
- **Step 1:** align radix points
  - shifting the mantissa LEFT by 1 bit DECREASES THE EXPONENT by 1 and RIGHT by 1 INCREASES THE EXPONENT by 1
  - we want to shift the mantissa right, because the bits that fall off the end should come from the least significant end of the mantissa
    - choose to shift the .25, since we want to increase it's exponent.
    - shift by

```
 10000101
-01111101
 00001000   (8) places.
```

00000000000000000000000 (original value)
10000000000000000000000 (shifted 1 place) (note that hidden bit is shifted into msb of mantissa)
01000000000000000000000 (shifted 2 places)
00100000000000000000000 (shifted 3 places)
00100000000000000000000 (shifted 4 places)
00001000000000000000000 (shifted 5 places)
00000100000000000000000 (shifted 6 places)
00000010000000000000000 (shifted 7 places)
00000001000000000000000 (shifted 8 places)

# SUBTRACTION EXAMPLE- BINARY

- Step 2: add (don't forget the hidden bit for the 100)

  1.10010000000000000000000 (100) -

  0.00000001000000000000000 (.25)

  1.10001111000000000000000

- Step 3: normalize the result (get the "hidden bit" to be a 1)
  - Result is already normalized

- In 32-bit format: 0 10000101 10001111000000000000000

  133 1.10001111000000000000000

  133-127=6

  $=1.10001111000000000000000 \times 2^6$

  =1100011.11000000000000000

  = 99.75

# 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.

# TOPICS COVERED FROM

- Textbook 1:
  - Chapter 1: 1.4.2,
  - Chapter 9: 9.7, 9.7.1

# COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

# 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  +
  00000001
  10000011
```

(if the exponent is –ve, subtract it from resultant E')

- 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:**

$1.000110000 \times 2^1$

**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= 10.01001001100101

After normalizing 1.001001001100101 X $2^1$

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

➔10.01001001100101

- **Step 3:** Adjust the exponent:
  - 136+1=137➔ 10001001
- 32-bit representation:
  - 0 10001001 001001001100101
  - = 1.001001001100101 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

```
                    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 0
```

The result is already normalized.

- The result in IEEE 32-bit format:

  0 10000001 11100000000…….

# 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

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

The result is already normalized.

- The result in IEEE 32-bit format:
  0 10000010 00000000000…….

i.e.,1.0 X$2^3$ = 1000 =8 in decimal

# 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)=$2^{40}$

# 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

# TOPICS COVERED FROM

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

# COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

# BYTE ADDRESSABILITY

- 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
    - Example: 15326→ 1, 53 and 26
    - Big-Endian: Little-Endian:
    - Address: Value Address: Value
    - 1000 : 01 1000 : 26
    - 1001 : 53 1001 : 53
    - 1002 : 26 1002 : 01
- The words "more significant" and "less significant" are used in relation to the powers of 2 assigned to bits when the word represents a number.

# WORD ALIGNMENT

- Word locations have aligned addresses if they begin at a byte address i.e., a multiple of the number of bytes in a word.
  - 16-bit word: word addresses: 0, 2, 4,….
  - 32-bit word: word addresses: 0, 4, 8,….
  - 64-bit word: word addresses: 0, 8,16,….
- If words begin at arbitrary byte address, words are said to have unaligned addresses.

# ACCESSING NUMBERS AND CHARACTERS

- A number usually occupies one word and can be accessed in the memory by specifying its word address.
- Individual characters can be accessed by their byte address.

# MEMORY OPERATIONS

- Load (or Read or Fetch)
  - Copy the content. The memory content doesn't change.
  - Address – Load
  - Registers can be used
- Store (or Write)
  - Overwrite the content in memory
  - Address and Data – Store
  - Registers can be used

# INSTRUCTION AND INSTRUCTION SEQUENCING

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

# REGISTER TRANSFER NOTATION

- Identify a location by a symbolic name standing for its hardware binary address (LOC, R0,…)
- Contents of a location are denoted by placing square brackets around the name of the location
- e.g. R1←[LOC]

  R3 ←[R1]+[R2]
- Register Transfer Notation (RTN)

# ASSEMBLY-LANGUAGE NOTATION

- Represent machine instructions and programs.
- Move R1,LOC → R1←[LOC]
- Add R3, R1, R2 → R3 ←[R1]+[R2]

# RISC AND CISC INSTRUCTION SET

- Most important characteristics that distinguish different computers is the nature of their instructions
- Reduced Instruction Set Computers (RISC)
  - Higher performance can be achieved if each instruction occupies exactly one word in memory, and all operands needed to execute a given arithmetic or logic operation specified by an instruction are already in processor registers
  - Various operations needed to process a sequence of instructions are performed in "pipelined" fashion
  - Number of different types of instructions may be included in the instruction set of a computer.
- Complex Instruction Set Computers (CISC)
  - An alternative to RISC
  - makes use of more complex instructions which may span more than one word of memory, and
  - may specify more complicated operations.

# RISC INSTRUCTION SETS

- Two key characteristics of RISC instruction sets are:
  1. Each instruction fits in a single word.
  2. A load/store architecture is used, in which
     - Memory operands are accessed only using Load and Store instructions.
     - All operands involved in an arithmetic or logic operation must either be in processor registers, or one of the operands may be given explicitly within the instruction word.
- At the start of execution of a program, all instructions and data used in the program are stored in the memory of a computer.
- Processor registers do not contain valid operands at that time.
- Load instructions: Operands expected to be in processor registers before they can be used by an instruction, are transferred from memory location into the registers.
- Load instructions are of the form
  - Load destination, source
- In general,
  - Load processor_register, memory_location

# RISC INSTRUCTION SETS

- C = A + B
- C ← [A] + [B]
- Assembly:
  - Load R2, A
  - Load R3, B
  - Add R4, R2, R3
  - Store R4, C
- Three-Address Instructions
  - Add destination, source1, source2
- The Store instruction is of the form
  - Store source, destination

# 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].

# TOPICS COVERED FROM

- Textbook 1:
  - Chapter 2: 2.1, 2.2, 2.3

# COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

# 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, NUM$n$ |
| $i + 8n - 8$ | Add | R2, R2, R3 |
| $i + 8n - 4$ | Store | R2, SUM |
| | ⋮ | |
| SUM | | |
| NUM1 | | |
| NUM2 | | |
| | ⋮ | |
| NUM$n$ | | |

**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, $200_{immediate}$
  - 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.

# TOPICS COVERED FROM

- Textbook 1:
  - Chapter 2: 2.3, 2.4

# COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

# 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        // loads the content in B i.e., the address of location where B is pointing at (ex: AAA0)
Load R3, (R2)     //go to location AAA0 and copy the data to R3
Store R3, A       // store the content of R3 in 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

- 2D array representation

**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.

**Figure 2.10** A list of students' marks.

# INDEXED ADDRESSING: VARIATIONS

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

- Base with Index plus constant: $X(R_i, R_j)$
  - 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

# 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 both N and Z are 0, that is, if the result produced by previous arithmetic 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.

# TOPICS COVERED FROM

- Textbook 1:
  - Chapter 2: 2.4, 2.10, 2.11, 2.12

# COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

# ARITHMETIC AND LOGIC UNIT

MODULE 3

# 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   │
  1011    │  Partial products
 1011     ┘
--------
10001111     Product (143)
```

**Figure 10.7** Multiplication of Unsigned Binary Integers

- Perform immediate addition to eliminate the need for additional registers to store the partial products
- Save time:
  - 1 multiplier: add and shift operation
  - 0 multiplier: shift operation

# MULTIPLICATION: UNSIGNED INTEGERS

Figure 10.9 Flowchart for Unsigned Binary Multiplication

# 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

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

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

# MULTIPLICATION: SIGNED INTEGERS- 2'S COMPLEMENT

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

| | |
|---|---|
| 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) |
| (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 2:
  - Chapter 10: 10.3

# COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

# MULTIPLICATION: UNSIGNED V/S SIGNED

| | |
|---|---|
| 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) |
| (a) Unsigned integers | (b) Twos complement integers |

**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 | $A \leftarrow A - M$ | First cycle |
| 1100 | 1001 | 1 | 0111 | Shift | |
| 1110 | 0100 | 1 | 0111 | Shift | Second cycle |
| 0101 | 0100 | 1 | 0111 | $A \leftarrow A + M$ | Third cycle |
| 0010 | 1010 | 0 | 0111 | Shift | |
| 0001 | 0101 | 0 | 0111 | Shift | Fourth cycle |

**Figure 10.13** Example of Booth's Algorithm ($7 \times 3$)

# 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><br>10011<br>*11001* | <br>01101<br>*10110* | <br>0<br>*1* | <br>01101<br>01101 | A=A-M [Subtract M from A(Adding 2's complement of M)]<br>*Shift Right- 2nd cycle* |
| <u>01101</u><br>00110<br>*00011* | <br>10110<br>*01011* | <br>1<br>*0* | <br>01101 | A=A+M<br><br>Shift Right- 3rd cycle |
| <u>10011</u><br>10110<br>*11011* | <br>01011<br>*00101* | <br>0<br>*1* | <br>01101<br>01101 | A=A-M [Subtract M from A(Adding 2's complement of M)]<br>Shift Right- 4th cycle |
| **11101** | **10010** | 1 | 01101 | Shift Right 5th cycle |
| Taking 2's complement | **0001001110→78**<br>**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 |
| 101001<br>101001<br>110100 | 011101<br>101110 | 0<br>1 | 010111 | A=A-M<br>Shift Right-1st cycle |
| 010111<br>001011<br>000101 | 101110<br>110111 | 1<br>0 | 010111 | A=A+M<br>*Shift Right- 2nd cycle* |
| 101001<br>101110<br>110111 | 110111<br>011011 | 0<br>1 | 010111 | A=A-M<br>Shift Right- 3rd cycle |
| 111011 | 101101 | 1 | 010111 | Shift Right- 4th cycle |
| 111101 | 110110 | 1 | 010111 | Shift Right 5th cycle |
| 010111<br>010100<br>001010 | 110110<br>011011 | 1<br>0 | 010111 | A=A+M<br>*Shift Right- 6th cycle* |
| | **001010011011→667**<br>**Product=667** | | | |

# 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: $X = \{1x_{n-2}x_{n-3} \ldots x_1x_0\}$
- 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,

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

- 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

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

# 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.
- $-6 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1$
- $M * (11111010) = M * (-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1)$
- $M * (11111010) = M * (-2^3 + 2^1)$ which is equivalent to
  - $M * (11111010) = M * (-2^3 + 2^2 - 2^1)$ [right to left:0-1→ $-2^1$, 1-0→ $+2^2$, 0-1→$-2^3$]

# 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<br>11101<br>11101 + 00011<br>00000 | 10000<br><br>10000<br>10000 | Shift Left<br>Subtract Divisor(Add 2's complement of divisor)<br>Set Q0=0<br>Restore 1st Cycle |
| 00001<br>11101<br>11110 + 00011<br>00001 | 00000<br><br>00000<br>00000 | Shift Left<br>Subtract Divisor(Add 2's complement of divisor)<br>Set Q0=0<br>Restore 2nd Cycle |
| 00010<br>11101<br>11111+ 00011<br>00010 | 00000<br><br>00000<br>00000 | Shift Left<br>Subtract Divisor(Add 2's complement of divisor)<br>Set Q0=0<br>Restore 3rd Cycle |
| 00100<br>11101<br>00001 | 00000<br><br>00001 | Shift Left<br>Subtract<br>Set Q0=1 4th cycle |
| 00010<br>11101<br>11111 + 00011<br>00010 | 00010<br><br>00010<br>00010 | Shift left<br>Subtract Divisor(Add 2's complement of divisor)<br>Set Q0=0<br>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.

# 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