FOUNDATIONS OF COMPUTING: FROM HARDWARE ESSENTIALS TO WEB DESIGN

Course Code : GXEST203

MODULE 2

SYLLABUS

MODULE	SUB MODULE	TOPIC	No.of Hours	Level	CO
2	2.1	Number systems	1	L2	CO2
2	2.2	Binary representation of data and numbers,	1	L2	CO2
2	2.3	Integer representation	1	L2	CO2
2	2.4	Data Storage units - bits, bytes, kilobytes, etc. ASCII and unicodes	2	L2	CO2
2	2.5	Basic CPU architecture	1	L2	CO2
2	2.6	ALU, registers, control unit	1	L2	CO2
2	2.7	Instruction format and assembly language (basics only)	1	L2	CO2
2	2.8	Fetch - execute cycle and instruction execution	1	L2	CO2

How Computers Represent Data

From a very early age, we arc introduced to the concept of numbers and counting. Toddlers learn early that they can carry two cookies, one in each hand. Kinder partners start counting by twos and fives. Invariably, we use the decimal number system. Our number system is based on 10, most likely because we have 10 fingers. A number system is simply a manner of counting. Many different number systems exist. Consider a clock. Clocks have 24 hours, each composed of 60 minutes. Each minute is composed of 60 seconds. When we time a race, we count in seconds and minutes.

Computers, like clocks, have their own numbering system, the binary number system.

Number Systems

- To a computer, everything is a number. Numbers arc numbers; letters and punctuation marks are numbers; sounds and pictures are numbers. Even the computer's own instructions are numbers. When you see letters of the alphabet on a computer screen, you arc seeing just one of the computer's ways of representing numbers.
- For example, consider the following sentence:

Here are some words.

This sentence may look like a string of alphabetic characters to you, but to a computer it looks like the string of ones and zeros

Number Systems

If base or radix of a number system is ‘r’, then the numbers present in that number system are ranging from zero to r-1. The total numbers present in that number system is ‘r’.

The following number systems are most commonly used.

- Decimal Number system
- Binary Number system
- Octal Number system
- Hexadecimal Number system

Decimal Number System

- The base or radix of Decimal number system is 10.
- Numbers ranging from 0 to 9 are used.
- The part of the number that lies to the left of the decimal point is known as integer part.
- The part of the number that lies to the right of the decimal point is known as fractional part.

Binary Number System

- All digital circuits and systems use this binary number system.
- The base or radix of this number system is 2.
- The numbers 0 and 1 are used in this number system.
- The part of the number, which lies to the left of the binary point is known as integer part.
- The part of the number, which lies to the right of the binary point is known as fractional part.

Octal Number System

- The base or radix of octal number system is 8.
- Numbers ranging from 0 to 7 are used in this number system.
- The part of the number that lies to the left of the octal point is known as integer part.
- The part of the number that lies to the right of the octal point is known as fractional part.

Hexadecimal Number System

- The base or radix of Hexa-decimal number system is 16.
- Numbers ranging from 0 to 9 and the letters from A to F are used in this number system.
- The decimal equivalent of Hexa-decimal digits from A to F are 10 to 15.
- The part of the number, which lies to the left of the hexadecimal point is known as integer part.
- The part of the number, which lies to the right of the Hexa decimal point is known as fractional part.

Equivalent Decimal, Binary, Octal and Hexadecimal Numbers

DECIMAL (BASE 10)	BINARY (BASE 2)	OCTAL (BASE 8)	HEXADECIMAL (BASE 16)
0	00000	0	0
1	00001	1	1
2	00010	2	2
3	00011	3	3
4	00100	4	4
5	00101	5	5
6	00110	6	6
7	00111	7	7
8	01000	10	8
9	01001	11	9
10	01010	12	A

11	01011	13	B
12	01100	14	C
13	01101	15	D
14	01110	16	E
15	01111	17	F
16	10000	20	10
Examples			
255	11111111	377	FF
256	100000000	400	100

Binary Representations of Data and Numbers

The Binary Numbering System

- **Introduction**

Computers store and process information using the **binary numbering system**. Unlike humans, who use **decimal (base-10)**, computers use **binary (base-2)** for internal data storage.

Binary Representation of Numeric and Textual Information

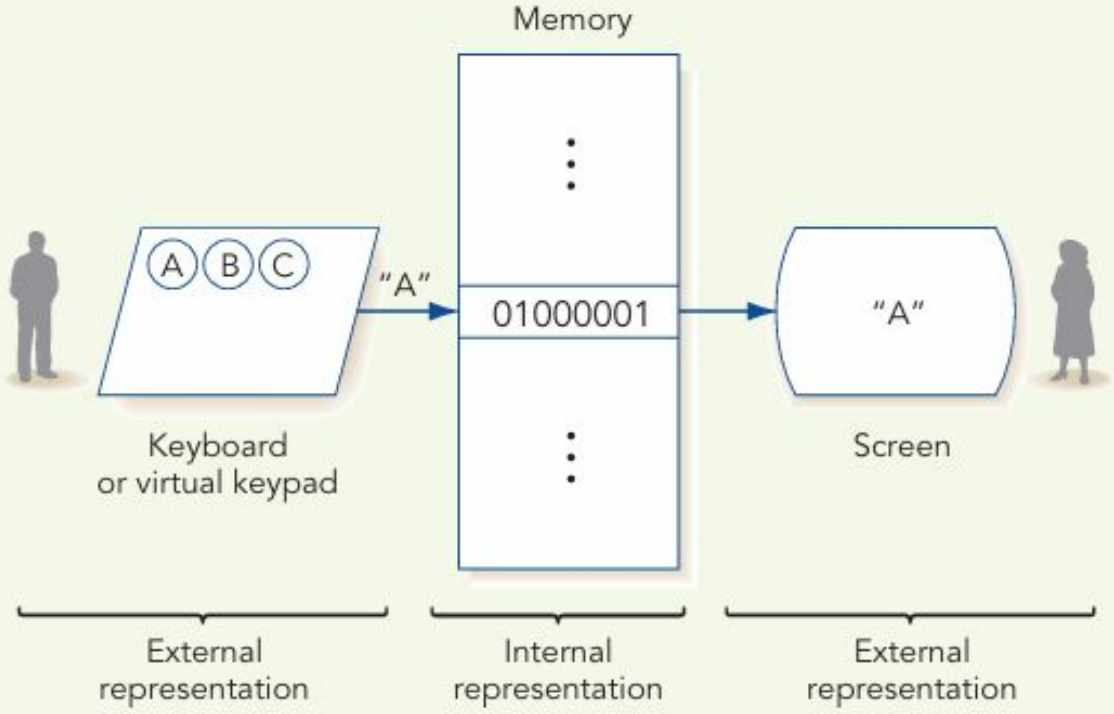
1. How Humans Represent Information:

1. **Decimal Digits (0-9)** – Used for numeric values (e.g., 459).
2. **Sign/Magnitude Notation** – A sign (+/-) placed before numbers (e.g., -131, +2789).
3. **Decimal Notation for Real Numbers** – A decimal point separates whole and fractional parts (e.g., 12.34).
4. **Alphabets and Symbols** – A-Z, a-z, and punctuation marks for text representation.

2. How Computers Represent Information:

- Externally, computers use **decimal digits, alphabets, and symbols**.
- Internally, all data—numbers, text, images, and sound—are stored using the **binary numbering system**.

FIGURE 4.1



Distinction between external and internal representation of information

Binary Numbering System

Binary is a base-2 numbering system (only two digits: **0 and 1**).

Just like decimal numbers use **powers of 10**, binary numbers use **powers of 2**.

Example: Decimal Number Representation

```
Decimal: 2,359
= (2 × 10^3) + (3 × 10^2) + (5 × 10^1) + (9 × 10^0)
= 2,000 + 300 + 50 + 9
= 2,359
```

Example: Binary Number Representation

```
Binary: 1101 (equivalent to 13 in decimal)
= (1 × 2^3) + (1 × 2^2) + (0 × 2^1) + (1 × 2^0)
= 8 + 4 + 0 + 1
= 13
```

Binary Conversion Methods

1. Decimal to Binary Conversion

Steps:

1. Divide the decimal number by 2.
2. Record the remainder (0 or 1).
3. Continue dividing until the quotient is 0.
4. Read the remainders from bottom to top.

Example: Convert 19 to Binary

```
19 ÷ 2 = 9  remainder = 1
 9 ÷ 2 = 4  remainder = 1
 4 ÷ 2 = 2  remainder = 0
 2 ÷ 2 = 1  remainder = 0
 1 ÷ 2 = 0  remainder = 1
```

2. Binary to Decimal Conversion

```
10011 = (1 × 2^4) + (0 × 2^3) + (0 × 2^2) + (1 × 2^1) + (1 × 2^(
      = 16 + 0 + 0 + 2 + 1
      = 19 (Decimal)
```

Binary Representation Limits

- Computers have a **fixed number of bits** (e.g., 16, 32, or 64 bits) to store integers.
- The maximum unsigned integer is determined by the number of bits.

Example:

- **5-bit system** → Maximum value = **11111 (31 in decimal)**.
- **16-bit system** → Maximum value = **1111 1111 1111 1111 (65,535 in decimal)**.

Binary Arithmetic Basics

- Binary arithmetic is simpler than decimal arithmetic because it involves only two digits: **0** and **1**.

- Arithmetic operations such as addition and subtraction have fewer rules.

- Binary addition follows four fundamental rules:

Binary Digits	Sum
0 + 0	0
0 + 1	1
1 + 0	1
1 + 1	10 (0 with a carry of 1)

The last rule states: **1 + 1 = 10**, which is equivalent to **2 in decimal**.

Example: Adding 7 (00111) and 14 (01110)

Step 1: Add the rightmost column (**1 + 0 = 1**). No carry is generated.

Step 2: Move to the next column (**1 + 1 = 10**), sum is 0 with a carry of 1.

Step 3: Add the next column (**1 + 1 + 1 = 11**), sum is 1 with a carry of 1.

Step 4: Add the next column (**0 + 1 + 1 = 10**), sum is 0 with a carry of 1.

Step 5: Add the leftmost column (**0 + 0 + 1 = 1**), sum is 1.

Introduction to Fractional Numbers in Binary

Fractional Numbers in Binary Representation

Key Points:

- Fractional numbers can be positive or negative (e.g., 12.34, -0.001275).
- Binary representation uses signed-integer techniques.
- Convert the number to scientific notation:
  - $\pm M \times B \pm E$
  - M: Mantissa
  - B: Base (2)
  - E: Exponent

Normalizing Binary Numbers

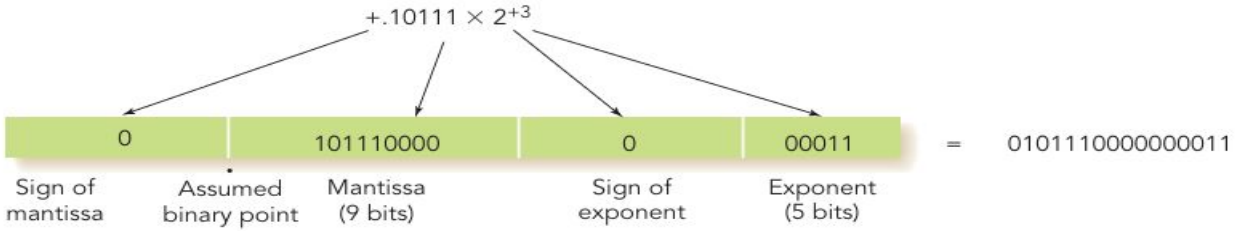
Title: Normalizing Binary Representation

Example:

- **$5.75 = 101.11 \times 2^0$**
- Normalize so the first significant digit is right of the binary point.

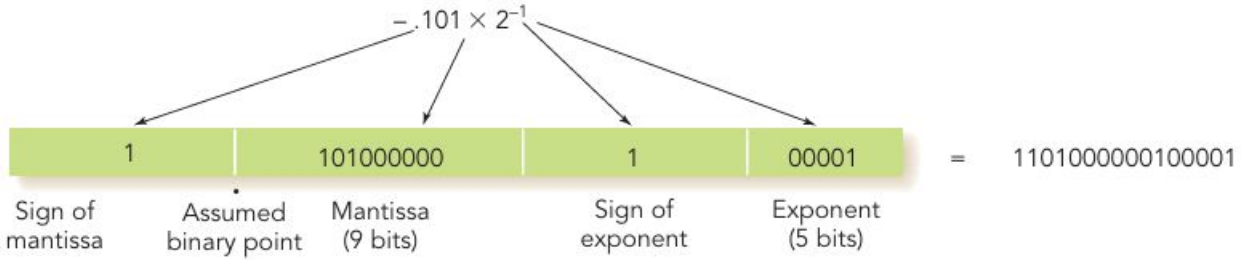
Steps:

1. **101.11×2^0**
2. **10.111×2^1**
3. **1.0111×2^2**



For another example, let's determine the internal representation of the fraction

−5/16. −5/16 = −(1/4 + 1/16) =−.0101 ×20 =−.101 ×2−1 (this is the value −5/16 in scientific notation) (after normalization)



Integer Representation

Representation of Un-Signed Binary Numbers

The bits present in the un-signed binary number holds the magnitude of a number. That means, if the un-signed binary number contains 'N' bits, then all N bits represent the magnitude of the number, since it doesn't have any sign bit.

Example Consider the decimal number 108.
The binary equivalent of this number is 1101100.

This is the representation of unsigned binary

number.

10810 = 11011002 It is having 7 bits. These 7 bits represent the magnitude of the number 108.

Representation of Signed Binary Numbers

The Most Significant Bit MSB of signed binary numbers is used to indicate the sign of the numbers. Hence, it is also called as sign bit. The positive sign is represented by placing '0' in the sign bit. Similarly, the negative sign is represented by placing '1' in the sign bit. If the signed binary number contains 'N' bits, then N−1 bits only represent the magnitude of the number since one-bit MSB is reserved for representing sign of the number.

There are three types of representations for signed binary numbers

- Sign-Magnitude form
- 1's complement form
- 2's complement form

Representation of a positive number in all these 3 forms is same. But, only the representation of negative number will differ in each form.

Example

Consider the positive decimal number +108. The binary equivalent of magnitude of this number is 1101100. These 7 bits represent the magnitude of the number 108. Since it is positive number, consider the sign bit as zero, which is placed on left most side of magnitude.

$+10810 = 011011002$

Therefore, the signed binary representation of positive decimal number +108 is 01101100. So, the same representation is valid in sign-magnitude form, 1's complement form and 2's complement form for positive decimal number +108.

Sign-Magnitude form

In sign-magnitude form, the MSB is used for representing sign of the number and the remaining bits represent the magnitude of the number. So, just include sign bit at the left most side of unsigned binary number. This representation is similar to the signed decimal numbers representation.

For an n-bit number representation:

– The most significant bit (MSB) indicates sign (0: positive, 1: negative).

– The remaining (n-1) bits represent the magnitude of the number.

Decimal	Signed magnitude representation in 4 bits	Decimal	Signed magnitude representation in 4 bits
+0	0000	-0	1000
+1	0001	-1	1001
+2	0010	-2	1010
+3	0011	-3	1011
+4	0100	-4	1100
+5	0101	-5	1101
+6	0110	-6	1110
+7	0111	-7	1111

Note: A problem- Two different representations for zero.

Example

Consider the negative decimal number -108. The magnitude of this number is 108. We know the unsigned binary representation of 108 is 1101100. It is having 7 bits. All these bits represent the magnitude. Since the given number is negative, consider the sign bit as one, which is placed on left most side of magnitude. $-108_{10} = 11101100_2$.

Therefore, the sign-magnitude representation of -108 is 11101100.

1’s complement form

The 1’s complement of a number is obtained by complementing all the bits of signed binary number. So, 1’s complement of positive number gives a negative number.

Similarly, 1’s complement of negative number gives a positive number. That means, if you perform two times 1’s complement of a binary number including sign bit, then you will get the original signed binary number.

Basic idea:

– Positive numbers are represented exactly as in sign-magnitude form.

– Negative numbers are represented in 1’s complement form.

- How to compute the 1’s complement of a number?

  – Complement every bit of the number (1 to 0, and 0 to 1).

  – Most Significant Bit (MSB) will indicate the sign of the number (0: positive, 1: negative).

Decimal	1's complement representation in 4 bits	Decimal	1's complement representation in 4 bits
+0	0000	-0	1111
+1	0001	-1	1110
+2	0010	-2	1101
+3	0011	-3	1100
+4	0100	-4	1011
+5	0101	-5	1010
+6	0110	-6	1001
+7	0111	-7	1000

Note: A problem- Two different representations for zero. Advantage of 1's complement representation: – Subtraction can be done using addition. – Leads to substantial saving in circuitry.

Example Consider the negative decimal number -108.

The magnitude of this number is 108.

We know the signed binary representation of 108 is 01101100.

It is having 8 bits. The MSB of this number is zero, which indicates positive number.

Complement of zero is one and vice-versa. So, replace zeros by ones and ones by zeros in order to get the negative number. $-10810 = 100100112$

Therefore, the 1's complement of -10810 is 10010011 2.

2’s complement form

The 2’s complement of a binary number is obtained by adding one to the 1’s complement of signed binary number. So, 2’s complement of positive number gives a negative number. Similarly, 2’s complement of negative number gives a positive number.

That means, if you perform two times 2’s complement of a binary number including sign bit, then you will get the original signed binary number.

Basic idea:

– Positive numbers are represented exactly as in sign-magnitude form.
– Negative numbers are represented in 2's complement form.

How to compute the 2's complement of a number?

– Complement every bit of the number (1 to 0 and 0 to 1), and then add one to the resulting number.

– MSB will indicate the sign of the number (0: positive, 1: negative).

Decimal	2’s complement representation in 4 bits	Decimal	2’s complement representation in 4 bits
+0	0000	-0	-
+1	0001	-1	1111
+2	0010	-2	1110
+3	0011	-3	1101
+4	0100	-4	1100
+5	0101	-5	1011
+6	0110	-6	1010
+7	0111	-7	1001

Note: Unique representations for zero.

Advantage of 2’ s complement representation:

– Unique representation of zero.

– Subtraction can be done using addition.

– Leads to substantial saving in circuitry.

Example Consider the negative decimal number -108.

We know the 1’s complement of (-108)10 is (10010011)2

2’s compliment of 10810 = 1’s compliment of 10810 + 1 = 10010011 + 1 = 10010100

Therefore, the 2’s complement of 10810 is 100101002.

Data storage units - bits, bytes, kilobytes, etc

Bit

- A **bit** is the smallest unit of data in computing.
- It can have **two states**:
  - **On (1)**
  - **Off (0)**
- Bits are the foundation of all digital information.
- Used to **encode**, **process**, and **transmit** data in computers.

Byte

- A **byte** is a group of **8 bits**.
- Widely used in computing to represent data.
- **Historically**: A byte was used to encode a single character (e.g., a letter or number).
- It is a fundamental building block for **digital data representation**.

Nibble

- A **nibble** is a group of **4 bits**.
- Represents **half a byte**.
- Plays an important role in defining data structures.
- Both bytes and nibbles are essential for storage and computing systems.

Kilobyte (KB)

- A **kilobyte** is often referred to as **1,000 bytes** in the **decimal system**, commonly used in marketing.
- In **computing**, a kilobyte is typically **1,024 bytes**, based on the **binary system** (2^10=1,024).
- This difference exists because computers operate on binary principles, while the decimal system is more familiar to consumers.

Unit	Shortened	Capacity
Bit	b	1 or 0 (on or off)
Byte	B	8 bits
Kilobyte	KB	1024 bytes
Megabyte	MB	1024 kilobytes
Gigabyte	GB	1024 megabytes

Unit	What can I store?
Byte	• 1 character of text, such as the letter “c”
Kilobyte	• 2 or 3 paragraphs of text (1,200 characters)
Megabyte	• 873 pages of plain text • 1 minute of an MP3 audio
Gigabyte	• 341 digital pictures (3MB average file size) • 256 MP3 audio files (4MB average file size) • 4,473 books (200 pages)

ASCII and Unicode

ASCII and Unicode

The two-character encoding schemes that are currently most widely utilized around the world are Unicode and ASCII. As opposed to ASCII, which is used to represent text in computers as symbols, characters, and numbers, Unicode is a character encoding that may be used to process, store, and exchange text data in any language.

ASCII

- ASCII (pronounced AS-key) stands for the American Standard Code for Information Interchange. Today, the ASCII character set is by far the most commonly used in computers of all types. Tabic 4A.3 shows the 128 ASCII codes. ASCII is an eight-bit code that specifics characters for values from 0 to 127.
- Extended ASCII- Extended ASCII is an eight-bit code that specifies the characters for values from 128 to 255. The first 40 symbols represent pronunciation and special punctuation. The remaining symbols are graphic symbols.

For Example:

Name	Symbol/char	ASCII Value	Hexadecimal Code
Dollar	$	36	24
The ambersand	&	38	26
Asterisk	*	42	2A
Minus sign/Hyphen	-	45	2D
Decimal Point	.	46	2E

Unicode

- The Unicode Worldwide Character Standard provides up to four bytes—32 bits—to represent each letter; number, or symbol. W ith four bytes, enough Unicode codes can be created to represent more than 4 billion different characters or symbols.
- This total is enough for every unique character and symbol in the world, induding the vast Chinese, Korean, and Japanese character sets and those found in known classical and historical texts.
- In addition to world letters, special mathematical and scientific symbols are represented in Unicode.
- One major advantage that Unicode has over other text code systems is its compatibility with ASCII codcs.
- The first 256 codes in Unicode are identical to the 256 codes used by the ASCII and Extended ASCII systems.

ASCII VS UNICODE

Let us now see some of the main differences between ASCII and UNICODE:

ASCII	UNICODE
Lowercase letters (a-z), uppercase letters (A-Z), numerals (0-9), and symbols like parenthesis, dollars, the ampersand, and others are represented by the ASCII code.	A wider range of characters than ASCII is represented by Unicode, including letters from languages like English, Arabic, Greek, and others, mathematical symbols, historical scripts, and emoji.
Only 128 different characters can be encoded using ASCII using a 7-bit range.	Unicode encrypts 154 written scripts.
ASCII is a proper subset of UNICODE	UNICODE is a superset of ASCII.

Basic CPU Architecture

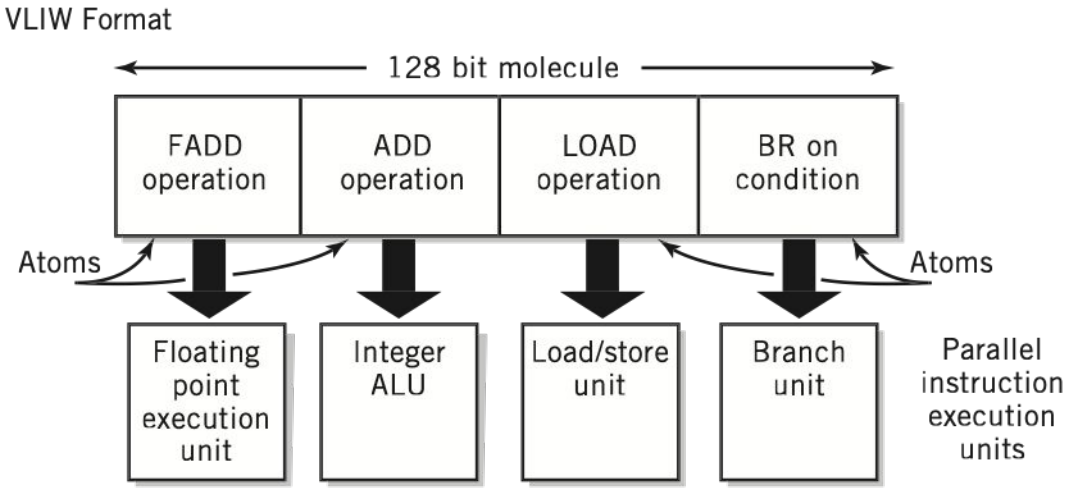
Experimental Architectures

- **VLIW (Very Long Instruction Word)**: Proposed by **Transmeta**, focuses on parallel execution.
- **EPIC (Explicitly Parallel Instruction Computing)**: Proposed by **Intel**, also focuses on parallelism.
- Still **new** and not fully proven for long-term viability.

VLIW Architecture

- **Example**: **Transmeta Crusoe** and **Efficeon** processors.
- **Instruction Word**: 128-bit **"molecule"**, divided into four 32-bit **atoms**.
- Each atom can execute a separate operation simultaneously.
- **Registers**: 64 general-purpose registers for fast register-to-register processing.
- **Key Feature**:
  - Instruction translation and reordering is handled by the **code-morphing layer**.
  - **Code-morphing layer** translates instructions from other CPUs and optimizes execution by resolving data dependencies.

Figure shows an example of a typical molecule.



EPIC Architecture

- **Example**: **Intel Itanium IA-64** series.
- **Instruction Set**: New, with **x86 compatibility** for backward support.
- **Registers**:
  - **128 general-purpose 64-bit registers**.
  - **128 82-bit floating-point registers**.
- **Instruction Word**: 41-bit wide instructions, grouped into **128-bit bundles** (3 instructions + 5 bits for type).

EPIC Execution and Programming

- Instructions are organized into **128-bit bundles** for parallel execution.
- **Assembly Language**:
  - Programmers follow guidelines to identify dependencies and allow parallel execution.
  - **Compiler**: High-level compilers ensure the code meets parallel execution requirements.
- **Responsibility**:
  - **VLIW**: Code-morphing software handles instruction sequencing.
  - **EPIC**: The **programmer or compiler** must optimize instruction sequencing and handle dependencies.

VLIW vs EPIC

Feature	VLIW (Transmeta Crusoe)	EPIC (Intel Itanium)
Instruction Word Size	128-bit (molecule)	41-bit instructions, 128-bit bundles
Parallel Execution	Automatically handled by code-morphing layer	Managed by programmer/ compiler
Register Count	64 general-purpose registers	128 64-bit general-purpose registers
Special Software	Code-morphing layer for instruction translation	Programmer/compiler optimizes execution
Compatibility	Translates from other CPU instruction sets	Includes x86 compatibility

Von Neumann Architecture

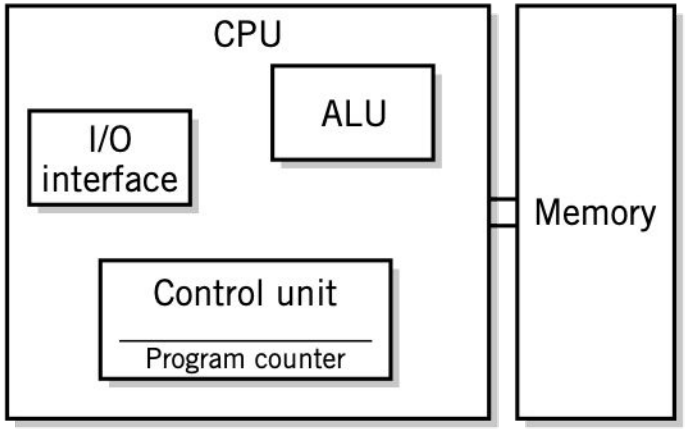
- Most modern CPUs are based on the **von Neumann architecture**.
- **Stored-program concept**: Program instructions and data are stored in the same memory.
- **Sequential execution**: Instructions are executed one after the other.
- All mentioned CPU architectures (including experimental ones) are **consistent** with this model.

Key Differences Between Traditional and Modern Architectures

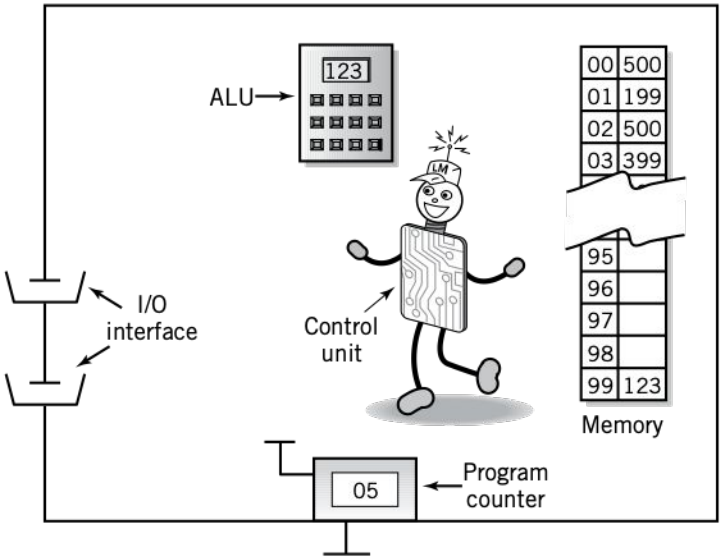
Feature	Traditional (e.g., Intel x86)	Modern (e.g., Sun SPARC)
General-Purpose Registers	Few	Many
Addressing Methods	Numerous	Single, register-based LOAD/STORE
Instruction Word Size	Variable (1 to 15 bytes)	Fixed (32 bits)
Specialized Instructions	Many	Few or none
Instruction Decoding	Slower (variable-length)	Faster (fixed-length, parallel)

ALU, REGISTER, CONTROL UNIT

System Block Diagram



The Little Man Computer



- The computer unit is made up conceptually of three major components, the arithmetic/logic unit (ALU), the control unit (CU), and memory.
- The ALU and CU together are known as the central processing unit (CPU).
- An input/output (I/O) interface is also included in the diagram.
- The I/O interface corresponds in function roughly to the input and output baskets, although its implementation and operation differ from that of the Little Man Computer in many respects.
- The arithmetic/logic unit is the component of the CPU where data is held temporarily and where calculations take place.
- It corresponds directly to the calculator in the Little Man Computer.
- The control unit controls and interprets the execution of instructions.
- It does so by following a sequence of actions that correspond to the fetch–execute instruction cycle.

- Most of these actions are retrievals of instructions from memory followed by movements of data or addresses from one part of the CPU to another.
- The control unit determines the particular instruction to be executed by reading the contents of a program counter (PC), sometimes called an instruction pointer (IP), which is a part of the control unit.
- Like the Little Man's location counter, the program counter contains the address of the current instruction or the next instruction to be executed.
- Normally, instructions are executed sequentially.
- The sequence of instructions is modified by executing instructions that change the contents of the program counter.
- The Little Man branch instructions are examples of such instructions.

- A memory management unit within the control unit supervises the fetching of instructions and data from memory.
- The I/O interface is also part of the control unit.
- In some CPUs, these two functions are combined into a single bus interface unit.
- The program counter in the CPU obviously corresponds to the location counter in the Little Man Computer, and the control unit itself corresponds to the Little Man.
- Memory, of course, corresponds directly to the mailboxes in the LMC.

The Concept of Registers

- A register is a single, permanent storage location within the CPU used for a particular, defined purpose.
- A register is used to hold a binary value temporarily for storage, for manipulation, and/or for simple calculations.
- Each register is wired within the CPU to perform its specific role.
- That is, unlike memory, where every address is just like every other address, each register serves a particular purpose.
- The register's size, the way it is wired, and even the operations that take place in the register reflect the specific function that the register performs in the computer.
- Registers also differ from memory in that they are not addressed as a memory location would be, but instead are manipulated directly by the control unit during the execution of instructions.

- Registers may be as small as a single bit or as wide as several bytes, ranging usually from 1 to 128 bits.
- Registers are used in many different ways in a computer.
- Depending on the particular use of a register, a register may hold data being processed, an instruction being executed, a memory or I/O address to be accessed, or even special binary codes used for some other purpose, such as codes that keep track of the status of the computer or the conditions of calculations that may be used for conditional branch instructions.
- Some registers serve many different purposes, while others are designed to perform a single, specialised task.
- There are even registers specifically designed to hold a number in floating point format, or a set of related values representing a list or vector, such as multiple pixels in an image.

- Registers are basic working components of the CPU.
- In the CPU, the equivalent to the calculator is known as an accumulator.
- Modern CPUs provide several accumulators; these are often known as general-purpose registers.
- Some vendors also refer to general-purpose registers as user-visible or program-visible registers to indicate that they may be accessed by the instructions in user programs.
- Groups of similar registers are also sometimes referred to collectively as a register file.
- General-purpose registers or accumulators are usually considered to be a part of the arithmetic/logic unit, although some computer manufacturers prefer to consider them as a separate register unit.
- As in the Little Man Computer, accumulator or general-purpose registers hold the data that are used for arithmetic operations as well as the results.
- In most computers, these registers are also used to transfer data between different memory locations, and between I/O and memory, again similar to the LMC.

- **The control unit contains several important registers.**
  - The **program counter register** (PC or IP) holds the address of the current instruction being executed.
  - The **instruction register (IR)** holds the actual instruction being executed currently by the computer. In the Little Man Computer, this register was not used; the Little Man himself remembered the instruction he was executing. In a sense, his brain served the function of the instruction register.
  - The **memory address register (MAR)** holds the address of a memory location.
  - The **memory data register (MDR)**, sometimes known as the *memory buffer register*, will hold a data value that is being stored to or retrieved from the memory location currently addressed by the memory address register.
- Although the memory address register and memory data register are part of the CPU, operationally these two registers are more closely associated with memory itself.

- The control unit will also contain several 1-bit registers, sometimes known as flags, that are used to allow the computer to keep track of special conditions such as arithmetic carry and overflow, power failure, and internal computer error.
- Usually, several flags are grouped into one or more status registers.
- In addition, our typical CPU will contain an I/O interface that will handle input and output data as it passes between the CPU and various input and output devices, much like the LMC *in* and *out* baskets.
- For simplification, we will view the I/O interface as a pair of I/O registers, one to hold an I/O address that addresses a particular I/O device, the other to hold the I/O data.
- These registers operate similarly to the memory address and data registers.
- Most instructions are executed by the sequenced movement of data between the different registers in the ALU and the control unit. each instruction has its own sequence.

Most registers support four primary types of operations:

1. Registers can be loaded with values from other locations, in particular from other registers or from memory locations. This operation destroys the previous value stored in the destination register, but the source register or memory location remains unchanged.

2. data from another location can be added to or subtracted from the value previously stored in a register, leaving the sum or difference in the register.

3. data in a register can be shifted or rotated right or left by one or more bits. This operation is important in the implementation of multiplication and division.

4. The value of data in a register can be tested for certain conditions, such as zero, positive, negative, or too large to fit in the register.

Instruction Format and Assembly Language

Instruction Format

Classification of Instructions

A) Data Movement Instructions: LOAD, STORE

B) Arithmetic Instructions: Add, Sub

C) Boolean Instructions: AND, OR,

D) Single Operand Instructions: Increment, Decrement

E) Shift and Rotate Instructions

Data Movement Instructions (load, store, and Other Moves)

- The move category commonly includes instructions to move data:
  - **From memory to registers**
  - **From registers to memory**
  - **Between registers**
  - **Between memory locations directly** (in some systems)

- There may be many different addressing modes available within a single computer.
- Additionally, variations on these instructions are frequently used to handle different data sizes.

  Data Size Variations:
  - Load Byte (1 byte)
  - Load Half-Word (2 bytes)
  - Load Word (4 bytes)
  - Load Double Word (8 bytes)

- Incidentally, the concept of a “word” is not consistent between manufacturers.

Word Size Variations

- **16 bits**, **32 bits**, or **64 bits** (depending on manufacturer)

- The Little Man load and store instructions are simple, though adequate, examples of move instructions.

- The major limitation of the Little Man load and store instructions is the fact that they are designed to operate with a single accumulator.

When More Registers are Added:

- Instructions need to specify which register to use
- **4 bits** required for each register (e.g., for 16 registers)
- **Instruction Size Decreases** if registers store memory addresses

- Additionally, it is desirable to have the capability to move data directly between registers, since such moves do not require memory access and are therefore faster to execute.
- In fact, some modern CPUs, including the ARM and Oracle SPARC architectures, provide only a minimal set of load/store or move instructions for moving data between the CPU and memory.
- All other instructions in these CPUs move and manipulate data only between registers.
- This allows the instruction set to be executed much more rapidly.

1. LOAD Instruction:

The `LOAD` instruction is used to transfer data from memory into a register. In other words, it loads the contents of a memory location into a processor register for further operations.

- **Syntax Example:**

  LOAD R1, [Address]

- This instruction means: load the data from the memory location specified by `Address` into register `R1`.
- **Function:** The `LOAD` instruction typically involves fetching data from a specific address in memory (RAM) and placing it into a register so that the processor can perform computations on it.
- **Use Case:** Whenever the processor needs data that is stored in memory, it issues a `LOAD` instruction to bring that data into a register. This is a crucial step before performing any computation or modification on that data.

2. STORE Instruction:

The `STORE` instruction is used to transfer data from a register to a memory location. It stores the contents of a register into a specified memory address.

- **Syntax Example:**

  ```
  STORE [Address], R1
  ```

- This instruction means: store the data from register `R1` into the memory location specified by `Address`.
- **Function:** The `STORE` instruction takes the data in a register and writes it into a memory location. This is typically used to save the results of computations or to store data temporarily for future use.
- **Use Case:** After performing operations on data, the processor might use the `STORE` instruction to save the results back to memory. This is essential for persisting results, especially when working with large datasets or needing to share data between different parts of a program.

- **LOAD:** Data is moved from memory to a register.
- **STORE:** Data is moved from a register to memory.

Arithmetic Instructions : Add, Sub

- Every CPU includes **integer addition** and **subtraction**
- Most CPUs also support **multiplication** and **division**
- Many instruction sets provide integer arithmetic for several different word sizes.
- Most current CPUs also provide floating point arithmetic capabilities.
- Some CPUs offer floating point arithmetic as an extension to the basic architecture.
- Floating point instructions usually operate on a separate set of floating point data registers with 32-, 64-, or 128-bit word sizes and formats conforming to Ieee Standard 754.
- The instruction set generally provides standard arithmetic operations and instructions that convert data between various integer and floating point formats.

- Some architectures also offer instructions for other, more specialized operations, such as square root, log, and trigonometry functions.
- extensive floating point calculations are required for many graphics applications, such as CAd/CAM programs, animation, and computer games.
- The same is true for solving sophisticated mathematical problems in science, economics, and business analytics.
- The presence of floating point instructions reduces the processing time for such calculations significantly.
- most modern CPUs also provide at least a minimal set of arith- metic instructions for BCd or packed decimal format, which simplifies the programming of business data processing applications.
- **Multiplication** and **division** could be done using **repeated addition** and **subtraction**
- Internal techniques use **shifting** for binary multiplication/division (a more efficient method than "long" multiplication)

- Modern hardware implements multiplication and division efficiently
- **Shift and add** operations are optimized with parallelization
- The subtract instruction is theoretically not necessary,
- Integer subtraction is performed internally by the process of complementing and adding.
- The same is true of BCd and floating point instructions.
- On the now rare computers that do not provide floating point instructions, there is usually a library of software procedures that are used to simulate floating point instructions.

Addition Instruction (ADD)

- **Purpose**:
  - Adds two values (usually registers or a register and a memory value).

- **Types of Operands**:
  - **Register + Register**
  - **Register + Memory**
  - **Memory + Register**

- **Implementation**:
  - Most CPUs use **binary addition** (1 + 1 = 0, carry 1).
  - Can handle **signed** or **unsigned integers**.

- **Common Use**:
  - **Arithmetic calculations**
  - **Loop counters**
  - **Array indexing**
  - **Pointer arithmetic**

Subtraction Instruction (SUB)

- **Purpose**:
  - Subtracts one value from another.

- **Types of Operands**:
  - **Register - Register**
  - **Register - Memory**
  - **Memory - Register**

- **Implementation**:
  - Uses **binary subtraction** with borrowing (just like in elementary subtraction).
  - For signed integers, the **two's complement** method is typically used to handle negative results.

- **Common Use**:
  - **Finding differences**
  - **Adjusting values**
  - **Decrementing loop counters**

Boolean Logic Instructions: AND, OR,

- Most modern instruction sets provide instructions for performing Boolean algebra.
- **AND** and **OR** are fundamental instructions for **bitwise logic operations** in digital circuits and CPU computations.

AND Instruction (Bitwise AND)

- **Purpose**:
  - Performs a **bitwise AND** between two operands.
  - Each corresponding bit in the operands is compared, and the result is 1 if both bits are 1, otherwise 0.

Operation:

- **1 AND 1 = 1**
- **1 AND 0 = 0**
- **0 AND 1 = 0**
- **0 AND 0 = 0**

Eg: AND R0, R1, R2
R0 = R1 AND R2

Example:

- `A = 1101 (binary)`
- `B = 1011 (binary)`

OR Instruction (Bitwise OR)

- **Purpose**:
  - Performs a **bitwise OR** between two operands.
  - Each corresponding bit in the operands is compared, and the result is 1 if at least one of the bits is 1.

Operation:

- **1 OR 1 = 1**
- **1 OR 0 = 1**
- **0 OR 1 = 1**
- **0 OR 0 = 0**

Example : OR R0, R1, R2

R0 = R1 OR R2

Example:

- `A = 1101 (binary)`
- `B = 1011 (binary)`
- `A OR B = 1111 (binary)`

Single Operand Instructions: Increment, Decrement

- Most commonly, the instruction set will contain instructions for negating a value, for incrementing a value, for decrementing a value, and for setting a register to zero.
- These instructions operate on only **one operand**.
- On some computers, the increment or decrement instruction causes a branch to occur automatically when zero is reached; this simplifies the design of loops by allowing the programmer to combine the test and branch into a single instruction.
- **Purpose**:
  - Modify the value of a single operand by either **adding 1** (increment) or **subtracting 1** (decrement).

Increment Instruction (INCR)

- **Increments** (adds 1) to the value of the operand.

Operation:

- **Operand = Operand + 1**

Example:

- If `A = 5`, after **incrementing**:
  - `A = A + 1` → `A = 6`
- INCR R0 ; R0 = R0 + 1

Decrement Instruction (DECR)

- **Decrements** (subtracts 1) from the value of the operand.

Operation:

- **Operand = Operand - 1**

Example:

Shift and Rotate Instructions

- Shift instructions move the data bits left or right one or more bits.
- Rotate instructions also shift the data bits left or right, but the bit that is shifted out of the end is placed into the vacated space at the other end.
- depending on the design of the particular instruction set, bits shifted out the end of the word may be shifted into a different register or into the carry or overflow flag bit, or they may simply “fall off the end” and be lost.
- Rotate instructions can be used to process data one bit at a time and are also used in many encryption algorithms.
- There are two types of shifts: **Logical Shifts** and **Arithmetic Shifts**.
- Both types shift data left or right but differ in how they handle the shifted-out bits, particularly when dealing with **signed numbers**.

Rotate Operations:

- Rotate instructions involve shifting data and wrapping the bits around to the other end.

Logical Shifts

- **Shifts bits left or right**.
- Vacated bit positions are filled with **zeros**.

Left Logical Shift **(Logical SHL)**:

- Bits are shifted to the left.
- **Zeroes** are inserted into the vacated rightmost positions.

Right Logical Shift **(Logical SHR)**:

- Bits are shifted to the right.
- **Zeroes** are inserted into the leftmost positions.

Example:

- `1010 (binary)` → Logical Left Shift → `0100 (binary)`

Arithmetic Shifts

- Designed to maintain the **sign** of a **signed number** when shifting.

Left Arithmetic Shift **(Arithmetic SHL)**:

- **Leftmost bit (sign bit)** is **not shifted**.
- **Zeroes** replace vacated positions on the right.
- **Effect**: **Doubles the value for each bit shifted left.**

Right Arithmetic Shift **(Arithmetic SHR)**:

- **Sign bit** (leftmost bit) is **shifted into** the vacated space.
- **Maintains the sign** of the number.
- **Effect**: **Halves the value for each bit shifted right.**

Example:

- **Left Shift**:
  - `1100` (binary) → Left Arithmetic Shift → `1000` (binary) (doubles the value)

Rotate Instructions

- Rotate operations shift bits left or right, and the bits that are shifted out are inserted back into the other end of the word.

Left Rotate (**ROL**):

- Shifts bits left.
- The leftmost bit that is shifted out is placed into the rightmost position.

Right Rotate (**ROR**):

- Shifts bits right.
- The rightmost bit that is shifted out is placed into the leftmost position.

Rotate with Carry:

- Some instruction sets use the **carry or overflow bit** as part of the rotation.
- The **carry bit** may be shifted in or out during the rotation.

Rotate Between Registers:

- Some CPUs allow rotating between **two registers**, which is useful for **data exchange**.

Example

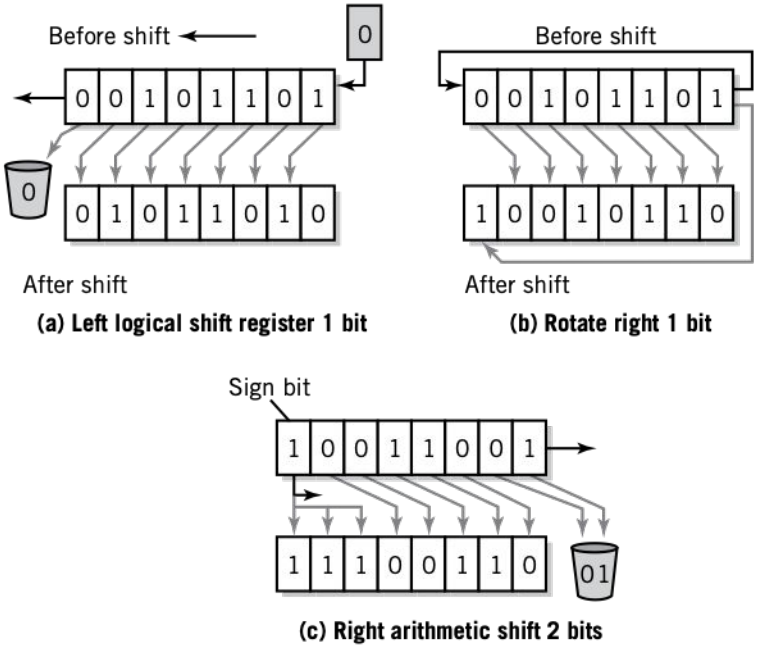


FIGURE : Typical Register Shifts and Rotates

Example:

Logical Left Shift:

```
SHL R0, #1   ; Shift R0 left by 1 bit (logical)
```

Logical Right Shift:

```
SHR R1, #1   ; Shift R1 right by 1 bit (logical)
```

Arithmetic Left Shift:

```
SAL R0, #1   ; Shift R0 left by 1 bit (arithmetic)
```

Arithmetic Right Shift:

```
SAR R1, #1   ; Shift R1 right by 1 bit (arithmetic)
```

Rotate Left:

Instruction Word Format - Unary, Binary, and Ternary

- Instructions in the Little Man Computer were made up entirely of three-digit decimal numbers, with a single-digit opcode, and a two-digit address field.
- The address field was used in various ways: for most instructions, the address field contained the two-digit address where data for the instruction could be found (e.g., load) or was to be placed (store).
  - **Memory Address**: Specifies where data can be found or stored.
  - **Branching Instructions**: The address field holds the **address of the next instruction**.
  - **I/O Instructions**: The address field defines the **I/O device address** (e.g., 01 for input, 02 for output).

Example : `LDA 05`: Load data from memory location 05 into the accumulator.

- The instruction set in a typical real CPU is similar.
- Instruction word can be divided into an opcode and zero or more address fields.
- A simple 32-bit instruction format with one address field are divided into :
  - **8-bit Opcode**: Specifies the operation.

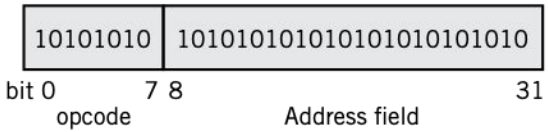


FIGURE : A Simple 32-bit Instruction Format

Address Field:

- **Address** can refer to either a **memory address** or a **general-purpose register**.
- We'll use "address" to refer to both types of data locations, and specify **memory address** when referring specifically to memory.

Example:

- A typical instruction might look like:
  - `Opcode (8 bits)|Address (24 bits)`

Explicit Address:

- The **address field** directly specifies the **location of data**.
- In the **Little Man Computer**, an instruction like **Load** uses an explicit memory address (e.g., `LDA 05`).

Implicit Address:

- The **destination** may be **implicit** in certain instructions.
- For example, in the **Little Man Computer**, the **accumulator** is always the **destination** in load and arithmetic operations.

Example:

- `LDA 05` – Data is loaded from memory location 05 into the **accumulator** (implicit destination).

Add/Subtract Instructions:

- Require two source addresses:
  - **Source 1**: Explicitly defined by the instruction.
  - **Source 2**: Implicitly the **accumulator**.
- The **result** is placed in the **accumulator** (implicit destination).

Move Instructions:

- For moving data between registers or memory, **two explicit addresses** are required.
  - **Source Address**: The data's current location.
  - **Destination Address**: The location where data will be moved.

- Example:
  - A move instruction from **register 5** to **register 10**:
  - `MOV R5, R10`: Move data from register 5 to register 10.

Register-to-Register Moves

The instruction consists of:

- **Opcode**: The operation to perform (e.g., move).
- **Register Address 1**: The source register.
- **Register Address 2**: The destination register.

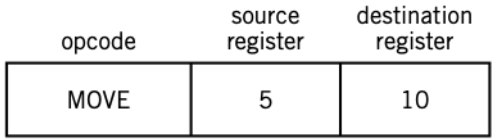


FIGURE : Typical Two Operation Register Move Format

Example:

- `MOV R5, R10`: Moves data from **register 5** to **register 10**.

Address Field in Modern Computers

- **Multiple Registers**:
  - Modern CPUs often have many general-purpose registers.
  - **Explicit Register Addresses**: For operations involving multiple registers, **two explicit address fields** are required to specify source and destination registers.

Instruction Example:

- **Memory to Register**:
  - `LDA R5, 02`: Load data from memory location 02 into register 5.

- **Register to Register**:
  - `MOV R1, R2`: Move data from register 1 to register 2.

Operands:

- The **source** and **destination** of data in an instruction.
- These can be **explicit** (clearly stated in the instruction) or **implicit** (assumed based on the instruction's definition).

Types of Instructions:

- **Move Instructions**: Require **two operands** (source and destination).
- **Arithmetic Operations (Add/Subtract)**: Require **three operands** (two sources and one destination).

Operand Fields:

- These are the **address fields** in the instruction that specify where the data is located or where it will go.

Operand Types:

- **Source Operand**: The location where data is being taken from.
- **Destination Operand**: The location where the result of the operation will be placed.

Address Fields:

- Instructions commonly have **explicit** address fields to specify **source** and **destination** locations.

Address Field Types:

- **In-place operations** (e.g., increment, complement): Typically use **one address field** (explicit or implicit).
- **Move operations** (e.g., load, store): Typically use **two address fields**.
- **Arithmetic operations** (e.g., add, subtract): Typically use **three address fields** (two sources and one destination).

Implicit Addressing:

- The address is **not explicitly stated** in the instruction. For example, the **accumulator** is often implicitly used in **load, add**, or **subtract** operations.

Explicit Addressing:

- Unary Instruction:
  - **One operand field**.
  - Example: **Increment**, **Complement** (operates on a single operand).

    INC R1
    - **Operand**: Register 1 (incremented in place).

- Binary Instruction:
  - **Two operand fields**.
  - Example: **Move**, **Add**, **Subtract** (requires two data locations).

    MOV R1, R2
    - **Source Operand**: Register 2
    - **Destination Operand**: Register 1

- Ternary Instruction:
  - **Three operand fields**.
  - Example: **Multiply**, **Divide** (operates on three operands, two sources and one destination).

    ADD R1, R2, R3
    - **Source Operand 1**: Register 1
    - **Source Operand 2**: Register 2

Assembly Language

The Role of Assembly Language

- **Second-Generation Language:** Marked a significant step in programming evolution.
- **Closer to Machine Language:** Each assembly instruction directly maps to a machine language instruction.
- **Still Low-Level:** It's a low-level language, meaning it's closer to hardware but easier to work with than raw machine code.

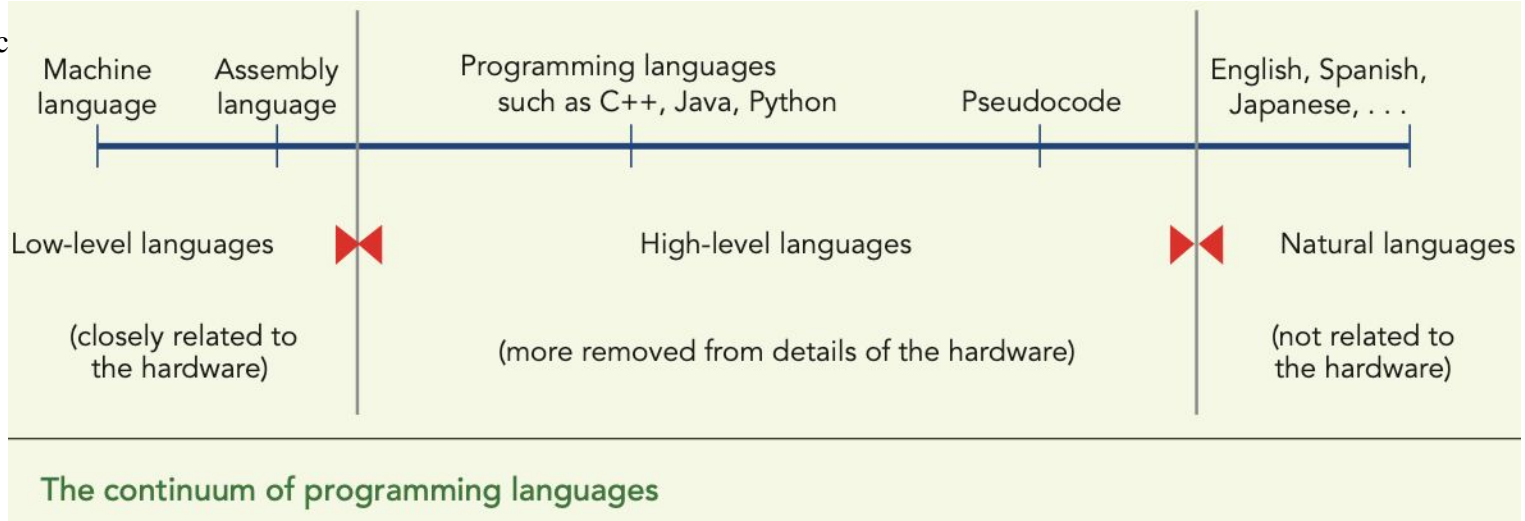
Compa

Feature	Low-Level (Assembly)	High-Level (C++, Java, Python)
Proximity to Hardware	Very close	Abstracted from hardware
Ease of Use	Difficult to read, write, modify	Easier to understand and use
Machine Instructions	1 instruction = 1 machine command	1 instruction = many machine commands
Portability	Machine-specific	Portable across different systems
Syntax	Symbols and mnemonics	Closer to natural language

Continuum of Programming Languages

- **Machine Language** (binary)
- **Assembly Language** (low-level)
- **High-Level Languages (e.g., C++, Python)**
- **Natural Languages** (e.g., English)

• Mac



The continuum of programming languages

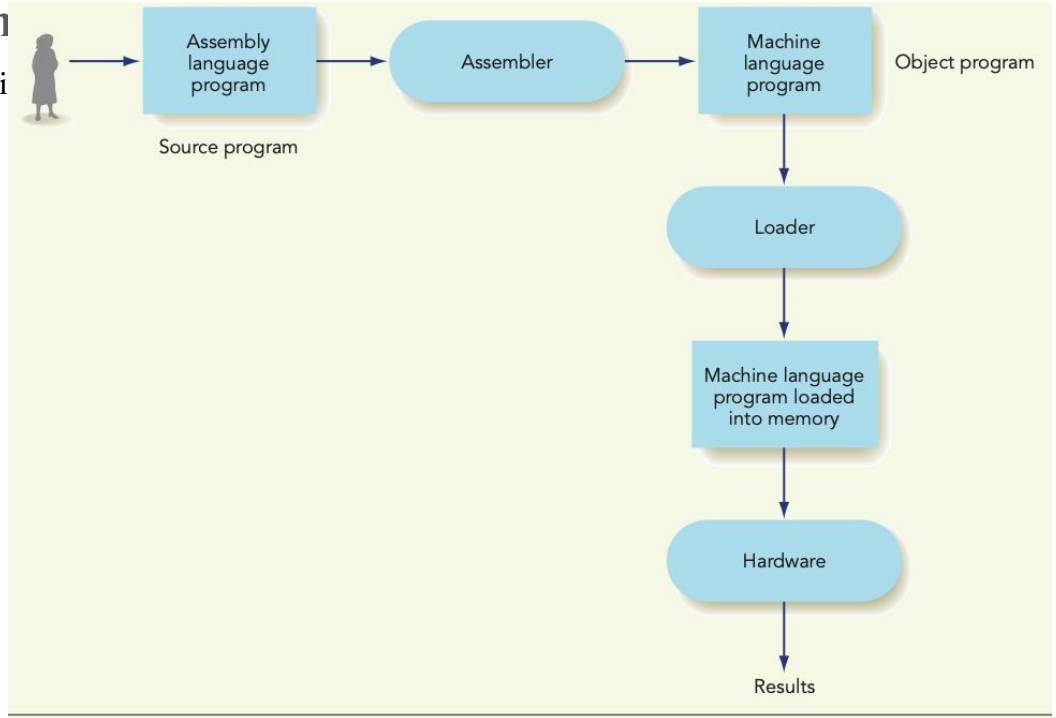
Why Assembly Language Matters Today

- **First Time Questioning Usability:** The first programming language to consider the user's experience.
- **Legacy:** Set the foundation for creating more accessible programming languages.
- **Modern Relevance:** The question of making machines easier to use is still at the heart of modern programming language development.

Understanding Assembly Language and Its Role

- Programs written in assembly language are called **source programs**.
- The processor cannot directly execute assembly instructions.
- The source program needs to be translated into machine language (object program).
- **Assembler**: A system software tool that performs this translation.

- Translation Process diagram:
  1. **Source Program** (Assembly Language)
  2. **Assembler** (Translation)
  3. **Object Program**
  4. **Processor** (Executi



The translation/loading/execution process

Advantages of Assembly Language

- **Symbolic Operation Codes**: Use mnemonics instead of numeric (binary) opcodes.
- **Symbolic Memory Addresses**: Use human-readable names instead of numeric (binary) addresses.
- **Pseudo-operations**: Provide helpful user-oriented services like data generation and easier debugging.

Structure of an Assembly Language Instruction

- **Format:**
    - `label: op code mnemonic address field -- comment`
- **Label**: Optional; identifies a memory location or function.
- **Op Code**: Operation to be performed (e.g., ADD, MOV).
- **Mnemonic**: Human-readable code for the operation.

Sample Assembly Language Instruction

LOAD: MOV A, 1000 -- Load the value at memory address 1000 into register A

- `LOAD`: Label (could indicate a memory location or block of code).
- `MOV`: Operation code (Mnemonic for moving data).
- `A`: Operand (the register to load data into).
- `1000`: The memory address containing the data.
- — `Comment`: Describes what the instruction does for clarity.

How the Assembler Works

- **Assembler** converts each instruction in the source program into a corresponding machine language instruction.
- Example:
  - Assembly Instruction: `MOV A, 1000`

Key Benefits of Assembly Language

- **Symbolic Operation Codes**: Easier to remember and understand than binary opcodes.
- **Symbolic Memory Addresses**: Can refer to memory locations by meaningful names instead of numbers.
- **Pseudo-operations**: Enhance the programmer's productivity with data generation and other tools.

• Assembly language uses symbolic mnemonics (e.g., `LOAD`, `ADD`, `STORE`) instead of binary opcodes.

• Provides human-readable instructions instead of obscure binary codes like `0000`, `0011`, etc.

• Helps bridge the gap between high-level programming languages and machine code.

Binary Op Code	Operation	Meaning
0000	LOAD X	CON(X) → R
0001	STORE X	R → CON(X)
0010	CLEAR X	0 → CON(X)
0011	ADD X	R + CON(X) → R
0100	INCREMENT X	CON(X) + 1 → CON(X)
0101	SUBTRACT X	R – CON(X) → R
0110	DECREMENT X	CON(X) – 1 → CON(X)
0111	COMPARE X	if CON(X) > R then GT = 1 else 0 if CON(X) = R then EQ = 1 else 0 if CON(X) < R then LT = 1 else 0
1000	JUMP X	Get the next instruction from memory location X.
1001	JUMPGT X	Get the next instruction from memory location X if GT = 1.
1010	JUMPEQ X	Get the next instruction from memory location X if EQ = 1.
1011	JUMPLT X	Get the next instruction from memory location X if LT = 1.
1100	JUMPNEQ X	Get the next instruction from memory location X if EQ = 0.
1101	IN X	Input an integer value from the standard input device and store into memory cell X.
1110	OUT X	Output, in decimal notation, the value stored in memory cell X.
1111	HALT	Stop program execution.

Typical assembly language instruction set

Symbolic Labels in Assembly

- Labels make programs more readable and maintainable.
- Labeling instructions like `LOOPSTART: LOAD X` provides a permanent reference to an instruction.
- Benefits:
  - **Clarity:** Names like `LOOPSTART` carry meaning, unlike numeric addresses.
  - **Maintainability:** Addresses don't need to be manually updated when instructions are added or removed.

Example : LOOPSTART: LOAD X
JUMP LOOPSTART

Addressing with Symbolic Labels

- Labels provide a permanent identifier for instructions or data.
- When the program is modified, labels ensure no manual address updates are needed.
- The assembler updates addresses automatically.

Example : JUMP LOOP
.
.
.
LOOP: LOAD X

Pseudo-ops and Data Generation

- Pseudo-ops, like `.DATA`, allow data generation and conversions.
- `.DATA` converts signed integers into the proper binary representation.

Example: FIVE: .DATA +5
NEGSEVEN: .DATA -7

- The assembler translates the decimal values into binary.

Using Labels with Data

- After using `.DATA`, the program can reference data by label.
- Example:
  - `LOAD FIVE` loads the value 5 into register `R`.
  - `ADD NEGSEVEN` adds -7 to the contents of register `R`.

Data vs. Instructions

- Data values and instructions use the same binary format but are distinguished by their context.
- If data is accidentally treated as an instruction, it can cause unexpected results.

Example : LOAD X
.DATA +1

The `.DATA +1` could be misinterpreted as an instruction by the CPU.

Avoiding Data/Instruction Confusion

- Ensure data is placed in memory areas that cannot be interpreted as instructions (e.g., after a `HALT` instruction).
- This prevents data from accidentally being executed.

Pseudo-ops for Program Construction

- `.BEGIN` and `.END` are pseudo-ops used for program construction.
- These do not generate machine instructions or data but mark the beginning and end of the assembly process.

Example showing `.B`

```
.BEGIN      --This must be the first line of the program
    ⋮       --Assembly language instructions like those in Figure 6.5
 HALT       --This instruction terminates execution of the program
    ⋮       --Data generation pseudo-ops such as
            --.DATA are placed here, after the HALT
.END        --This must be the last line of the program
```

Structure of a typical assembly language program

Instruction Format in Assembly

- **Instruction Format** refers to the way instructions are structured in
- memory. Each instruction typically consists of **3 parts**:
  1. **Opcode**: Specifies the operation to be performed (e.g., ADD, MOV).
  2. **Operand(s)**: Specifies the data or registers the operation acts upon.
  3. **Addressing Mode**: Specifies how the operand is accessed.

Basic Instruction Format

1. **Opcode (Operation Code)**: The operation to be performed (e.g., MOV, ADD, SUB).

3. **Operand(s)**: The data to be used or manipulated by the instruction (e.g., register, memory address).

5. **Addressing Mode**: The method by which the operand is located (e.g., direct, indirect, immediate).

Example:

- **MOV AX, 5**

- **Opcode**: MOV (Move data)
- **Operand**: AX (destination register), 5 (source value)

Components of an Assembly Instruction

1. **Opcode**: The command that tells the CPU what to do (e.g., ADD, SUB).
2. **Source Operand**: The data or register used in the operation.
3. **Destination Operand**: Where the result of the operation is stored.
4. **Addressing Mode**: Specifies how operands are accessed (e.g., register, immediate, direct).

- **ADD AX, BX**

•**Opcode**: ADD (Add values)
•**Source Operand**: BX (Second operand)
•**Destination Operand**: AX (First operand)

Example of an Assembly Instruction

- MOV AX, [2000h] ; Load value from memory address 2000h into AX
- ADD AX, BX ; Add value in BX to AX
- MOV [3000h], AX ; Store value in AX to memory address 3000h

Explanation:

- **MOV AX, [2000h]**: The value from memory at address 2000h is moved into the register AX.
- **ADD AX, BX**: The value in register BX is added to the value in register AX.
- **MOV [3000h], AX**: The value in AX is stored in memory at address 3000h.

Common Assembly Instructions

- **MOV**: Moves data between registers or from memory.
  - Example: `MOV AX, 5` (Moves the value 5 into register AX).
- **ADD**: Adds two operands.
  - Example: `ADD AX, BX` (Adds value in BX to AX).
- **SUB**: Subtracts one operand from another.
  - Example: `SUB AX, BX` (Subtracts BX from AX).
- **JMP**: Jumps to a specified instruction location.
  - Example: `JMP label` (Jumps to the specified label in the code).

The Fetch Execute Cycle and Instruction Execution

Introduction to Fetch-Execute Cycle

- The **fetch-execute instruction cycle** is the core process for **executing every instruction** in a CPU.
- All computer instructions, regardless of complexity, ultimately follow this cycle.
- The cycle involves moving data between **registers** (e.g., **A**, **GR**, **PC**, **IR**, **MAR**, **MDR**) and performing basic operations like **moving**, **adding**, **shifting**, or **testing** data.
- The CPU executes instructions based on these primary operations.

Key Registers in the Instruction Cycle

- **Program Counter (PC)**: Holds the address of the current instruction.
- **Instruction Register (IR)**: Holds the instruction while it's being executed.
- **Memory Address Register (MAR)**: Holds the address in memory to access.
- **Memory Data Register (MDR)**: Holds the data fetched from memory.
- **Accumulator (A)**: Holds data being processed by the CPU.

The Fetch-Execute Instruction Cycle

• The core operation of a computer is the fetch-execute instruction cycle.
• It drives every task a computer performs.
• Understanding the cycle provides insight into the internal workings of the computer.

Core Operations Performed in a CPU

- Data movement between registers
- Arithmetic (addition, subtraction)
- Data shifting within registers
- Testing register values (negative, positive, zero)
- These operations underpin how every instruction is executed.

Registers in the Instruction Cycle

- **General-purpose registers (A, GR):** Store data values
- **Program Counter (PC):** Holds the address of the current instruction
- **Instruction Register (IR):** Holds the current instruction during execution

Steps in the Fetch-Execute Cycle

Phase 1: Fetch

- Transfer value from PC to MAR
- Fetch instruction from memory to MDR
- Transfer instruction from MDR to IR

Phase 2: Execute

- The CPU decodes and performs the instruction based on what's in the IR.

Fetch Phase Steps

- **Step 1:** Transfer the current instruction address (PC) to MAR
- **Step 2:** Fetch the instruction from memory and load it into MDR
- **Step 3:** Move the instruction from MDR to IR (to begin execution)

Execute Phase - Instruction Specific

- **Instruction dependent:** The remaining steps vary depending on the instruction being executed.

- Example: Load instruction, store instruction, add instruction – all follow a similar structure but vary in specific data movement.

Fetch-Execute Cycle for a LOAD Instruction

- **Step 1**: **PC → MAR**
  - The value in **PC** (the current instruction address) is transferred to **MAR**.
- **Step 2**: **MDR → IR**
  - The instruction at the address specified in **MAR** is fetched from memory and loaded into **IR**.

Step 3: **IR[address] → MAR**

- The address part of the **LOAD** instruction in **IR** is transferred to **MAR** to access the data.

- **Step 4**: **MDR → A**
  - The data from **MDR** is loaded into the **Accumulator (A)**.
- **Step 5**: **PC + 1 → PC**
  - Increment **PC** to point to the next instruction.

Fetch-Execute Cycle for a STORE Instruction

- **Step 1**: **PC → MAR**
  - The value in **PC** is transferred to **MAR**.
- **Step 2**: **MDR → IR**
  - The instruction is fetched from memory and loaded into **IR**.
- **Step 3**: **IR[address] → MAR**
  - The address part of the **STORE** instruction in **IR** is transferred to **MAR**.
- **Step 4**: **A → MDR**
  - The value in **Accumulator (A)** is stored into **MDR** to be written back to memory.
- **Step 5**: **PC + 1 → PC**
  - Increment **PC** to the next instruction.

Fetch-Execute Cycle for an ADD Instruction

Step 1: **PC → MAR**

- The value in **PC** is transferred to **MAR**.

Step 2: **MDR → IR**

- The instruction is fetched from memory and loaded into **IR**.

Step 3: **IR[address] → MAR**

- The address part of the **ADD** instruction in **IR** is transferred to **MAR**.

Step 4: **A + MDR → A**

- The value in **A** is added to the value in **MDR**, and the result is stored back in **A**.

Step 5: **PC + 1 → PC**

- Increment **PC** to the next instruction.

Elegance of the Cycle

- Only **5 steps** for the load instruction
- 4 steps involve data movement between registers
- The 5th step is an addition (increment PC by 1)
- Simplicity and efficiency of the instruction cycle

Comparison with Other Instructions

- **Store Instruction:** Similar steps to load, but it writes to memory
- **Add Instruction:** Similar, but involves an addition operation to the accumulator
- All instructions follow a similar flow of data movement and simple operations

EXAMPLE

1st instruction LOAD 90:	PC → MAR	MAR now has 65
	MDR → IR	IR contains the instruction: 590
	- - - - - - - - - - - - ← end of fetch	
	IR [address] → MAR	MAR now has 90, the location of the data
	MDR → A	Move 111 from MDR to A
	PC + 1 → PC	PC now points to 66.
- -	end of execution, end of first instruction	
2nd instruction ADD 92:	PC → MAR	MAR now contains 66
	MDR → IR	IR contains the instructions: 192
	- - - - - - - - - - - - - ←end of fetch	
	IR [address] → MAR	MAR now has 92
	A + MDR → A	111+222=333 in A
	PC + 1 → PC	PC now points to 67
- -	end of execution, end of second instruction	
3rd instruction STORE 90:	PC → MAR	MAR now contains 67
	MDR → IR	IR contains 390
	- - - - - - - - - - - - - ← end of fetch	
	IR [address] → MAR	MAR now holds 90
	A → MDR	The value in A, 333, moves to mem location 90
	PC + 1 → PC	PC now points to 68
- -	end of execution, end of third instruction	
← ready for next instruction		

Example Program Execution

- **Program Segment**:
  1. **LOAD 90**
  2. **ADD 92**
  3. **STORE 90**

- **Initial Values**:
  1. **PC = 65**
  2. Memory:
     - 65: **590 (LOAD 90)**
     - 66: **192 (ADD 92)**
     - 67: **390 (STORE 90)**
     - 90: **111**
     - 92: **222**

•Each instruction is processed step-by-step through its **fetch-execute cycle**.

Program Execution Step-by-Step

1. **LOAD 90**:
   - **PC** → **MAR**, fetch instruction, **IR** → **MAR**, load **MDR** → **A**.
2. **ADD 92**:
   - **PC** → **MAR**, fetch instruction, **IR** → **MAR**, **A** + **MDR** → **A**.
3. **STORE 90**:
   - **PC** → **MAR**, fetch instruction, **IR** → **MAR**, **A** → **MDR**, store back to memory.