CW25201: COMPUTER ORGANIZATION AND ARCHITECTURE

Unit I Introduction **9+3 hours**

Functional Units of a Digital Computer, Classes of Computer Systems, Hardware-Software Interface, Operation and Operands of Computer Hardware, Instruction Set Architecture, RISC and CISC Architectures, Addressing Modes, Assembly Language Programming, Translation from High-Level Language to Machine Language, Performance Metrics, Benchmarks, Transition from Uni-processors to Multiprocessors

Activities:

- C code to machine code mapping.
- Assembly of computer system components

1. Foundations of Computer Architecture and Organization

To fully comprehend the operational nature of digital computer systems, we must explore the distinct definitions and interactions that separate the physical hardware from the conceptual systems programmers see. Computer architecture acts as the fundamental abstract interface between the hardware and the lowest level of software. It defines what attributes are visible to the system programmer, including word sizes, instruction formats, addressing modes, and the various representations of data within memory.

1.1 Computer Architecture vs. Computer Organization

In general computer design terminology, we must make a sharp, academic distinction between 'Computer Architecture' and 'Computer Organization':

- **Computer Architecture:** Refers to those attributes of a system that are directly visible to a programmer, or to the compiler writer. These attributes have a direct, logical impact on the execution of programs. Examples include the instruction set, the data representation, memory addressing techniques, and I/O mechanisms. It dictates what the machine can do, serving as an abstract blueprint.
- **Computer Organization:** Refers to the operational units and their interconnections that realize the architectural specifications. It describes *how* those features are implemented within the system. Examples include control signals, interface mechanisms between the processor and its peripherals, memory technology, and physical bus structures. For instance, whether the CPU implements a hardware multiply instruction or handles it via repeated addition is a question of organization, whereas the existence of the multiply instruction itself is a question of architecture.
- **Computer Hardware:** Consists of the actual physical components, electronic circuits, logic gates, silicon-level transistors, printed circuit boards, displays, magnetic or optical storage media, and cabling that physically make up the system. It represents the concrete implementation of the organizational specifications.

1.2 The Von Neumann Architecture Model

The vast majority of modern general-purpose digital computers are based on the Stored Program Concept, a design paradigm commonly known as the Von Neumann Architecture, named after the mathematician John von Neumann. In this model, the computer is organized as a sequential processor featuring a single, unified memory structure that houses both program instructions and data elements.

Because instructions and data share the same memory structure and physical memory bus, the processor cannot fetch an instruction and read/write data simultaneously. This structural constraint is known as the Von Neumann Bottleneck, which limits overall system performance. The machine instructions are processed in a strictly sequential manner. A dedicated register, the Program Counter (PC), holds the memory address of the next instruction to be fetched and executed. This sequential execution model means that Neumann machines are characterized as control flow computers, since instruction sequencing is explicitly steered by the program counter, executing one instruction at a time.

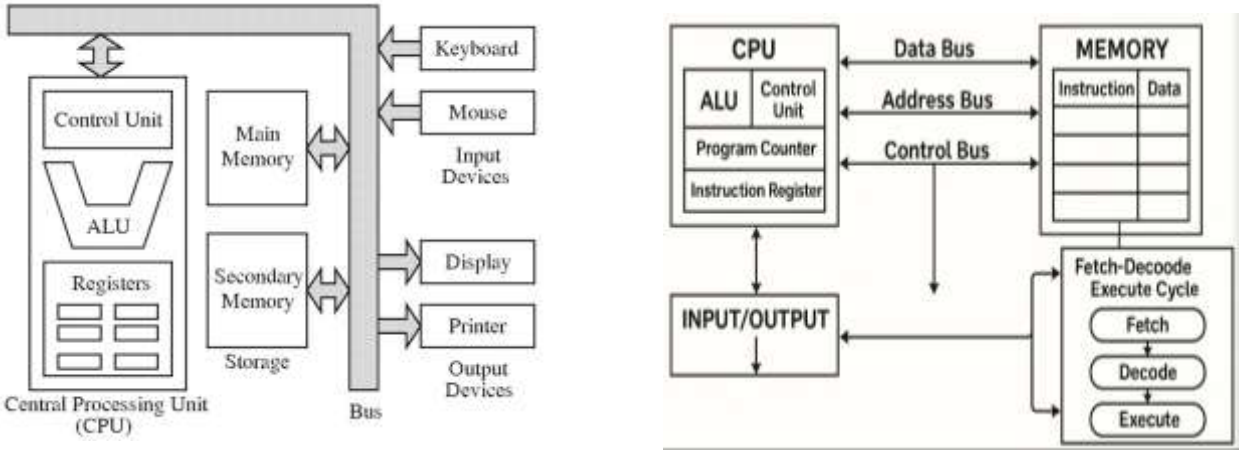


Fig 1.1 (a) Von Newmann Architecture (b) Program Execution

1.3 Detailed Examination of the Five Classic Functional Units

A standard digital computer system consists of five main, functionally independent units that communicate and coordinate to execute software programs:

1. **Input Unit** The gateway through which the computer accepts coded information from the outside world. Input devices, such as keyboards, mice, trackballs, joysticks, scanners, and microphones, read data from a source and translate it into electronic impulses (binary codes representing 0s and 1s) for transfer into the CPU or main memory. For example, when a user presses a key on a standard keyboard, the corresponding character or digit is translated into its equivalent binary ASCII or Unicode scan-code and transmitted over a cable to either the memory or processor registers.

2. **Output Unit** The counterpart to the input unit. Its primary function is to send processed results from the computer to the outside world, translating binary representations of data into human-understandable or physical forms. Output devices include monitors, displays, speakers, headphones, and printers. Some devices, such as touchscreens, hard disk drives, and flash memory drives, perform the dual role of both input and output, often grouped under the unified category of I/O units.

3. **Memory Unit** Responsible for storing active programs and the data currently needed by the processor. Memory is organized hierarchically into primary and storage frameworks:

- **Primary Memory (Main Memory):** Fast semiconductor storage that operates at electronic speeds. It is volatile, meaning its contents are lost when the system is powered down. It consists of millions of semiconductor storage cells, each capable of storing a single bit of information. These cells are processed in groups of fixed size called words. Each word is associated with a distinct, unique memory address. Successive addresses refer to successive word (or byte) locations in memory. Primary memory includes high-speed Static RAM (SRAM) for caches, and Dynamic RAM (DRAM) for the main memory subsystem.
- **Secondary Memory:** Non-volatile, high-capacity, and relatively slow magnetic or optical storage devices (such as magnetic hard disk drives, magnetic tapes, and optical disks) used for long-term file retention. Data must be transferred into primary memory before it can be accessed by the CPU.

4. Arithmetic and Logic Unit (ALU) The core digital circuit responsible for executing all mathematical computations and logical decisions. Arithmetic operations include basic binary addition, subtraction, multiplication, and division. Logical operations perform comparisons (e.g., testing whether one operand is equal to, less than, or greater than another) to redirect program execution flow based on results. The speed of the ALU is dramatically higher than that of memory; therefore, operands are loaded into a tiny set of extremely fast processor registers (with faster access times than even L1 caches) before the ALU operates on them.

5. Control Unit (CU) The nerve center of the entire computer system. It coordinates and controls all activities across the other four units. The control unit reads one instruction at a time from the memory unit, decodes it (interprets the binary pattern to determine what operations must be carried out), and generates a sequence of electronic control pulses and timing signals. These signals direct the flow of data traffic, instruct the ALU on what mathematical operations to perform, manage data transfers between memory and registers, and orchestrate input/output events. The control unit itself does not process data; rather, it acts as a director that maintains order and schedules execution sequences.

2. Hardware-Software Interface and Layers of Abstraction

To manage the immense complexity of computer systems, computer scientists rely on clear layers of abstraction. Software applications (such as word processors, spreadsheets, compilers, and browsers) are written in high-level programming languages that shield developers from the low-level physical realities of logic gates and silicon circuits. The Operating System (OS sit directly below these applications, acting as a resource allocator that manages hardware resources (CPU cycles, memory allocation, and peripheral access) fairly and efficiently, coordinating execution between concurrent tasks.

At the very bottom of the software layer sits firmware, typically stored in non-volatile Read-Only Memory (ROM) or Electrically Erasable Programmable ROM (EEPROM). When a computer system is powered up or rebooted, it executes a simple initial boot program (the bootstrap loader) located in firmware. This bootstrap program initializes all CPU registers, memory controllers, and peripheral devices, locates the operating system kernel on a secondary storage drive, loads the kernel into main

memory, and jumps CPU control to the first instruction of the operating system, allowing the software-to-hardware interface to seamlessly begin execution.

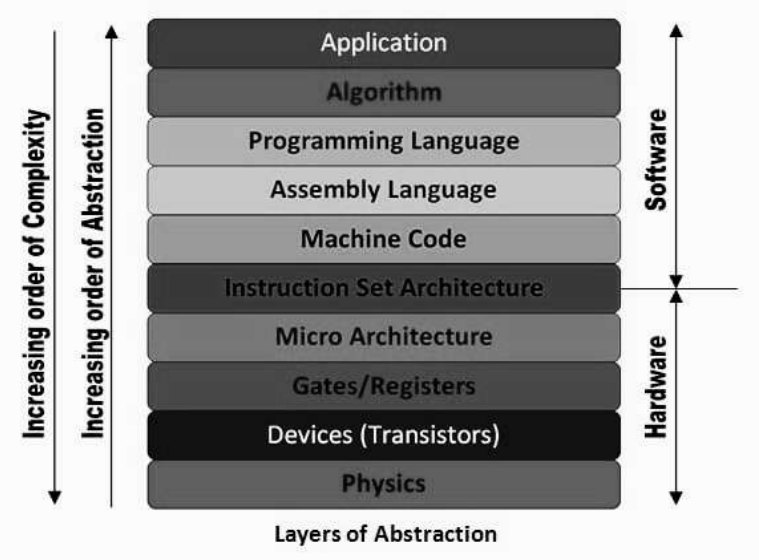


Figure 1.2 Layers of Abstraction

The **Layers of Abstraction** in a computer system illustrate how highly complex software applications are systematically translated, mapped, and executed on physical hardware and fundamental physics.

This hierarchical structure is divided into two distinct domains—**Software** and **Hardware**—bridged by a critical boundary layer:

2.1. The Software Domain (The Upper Layers)

This domain represents high-level logical abstractions that allow developers to write complex programs without worrying about physical transistor behavior.

- **Application**: At the highest level are the programs designed to solve user problems, such as word processors, web browsers, spreadsheets, and database systems.
- **Algorithm**: The structured logical procedures or recipes chosen to solve a computational problem, determining the algorithmic step complexity (such as linear, quadratic, or logarithmic growth).
- **Programming Language**: High-level, human-readable programming languages (e.g., C, C++, or Java) that express algorithms in a highly readable format independent of specific computer hardware.

- **Assembly Language**: A symbolic, low-level representation of machine code specific to a processor family. It translates abstract programming constructs into readable mnemonics (like `add`, `sub`, `lw`, `sw`) and explicit register allocations.
- **Machine Code**: The raw binary representation (composed of base-2 numbers represented physically as high and low electronic signals, or `0`s and `1`s) that processor hardware can directly decode and execute.

2.2. The Core Boundary

- **Instruction Set Architecture (ISA)**: Acting as the crucial interface between the software stack and physical hardware, the ISA defines the abstract machine model visible to the assembly language programmer or compiler writer. It specifies the supported instruction formats, opcodes, register files, memory addressing modes, and data representations.

2.3. The Hardware Domain (The Lower Layers)

This domain represents the physical implementation of the ISA, descending recursively into microelectronic circuitry.

- **Micro Architecture (Computer Organization)**: The high-level design of the internal processor. This includes the physical pipeline structure, control unit design, ALU datapath layout, and memory/cache hierarchies that realize the architectural specifications defined by the ISA.
- **Gates/Registers**: The digital logic layer. It consists of combinational logic gates (AND, OR, XOR) and sequential storage cells (flip-flops, registers, SRAM cells) that process and temporarily hold individual bits of data.
- **Devices (Transistors)**: The microelectronic silicon components. It primarily consists of silicon field-effect transistors (FETs or CMOS technology) configured as physical switches to control, switch, and amplify electronic currents.
- **Physics**: The fundamental layer of solid-state semiconductor physics, governing the movement of charge carriers (negatively charged electrons and positively charged holes) through engineered silicon crystal lattices.

The Dynamic Vectors (Left Arrows)

The diagram highlights two opposite, logical movements across the hierarchy:

1. **Increasing Order of Abstraction (Upward Arrow)**: Moving upward hides lower-level microarchitectural, electrical, and physical complexities. This enables software developers to program productively using clean, high-level interfaces without needing to manage individual transistors or silicon gates.
2. **Increasing Order of Complexity (Downward Arrow)**: Descending recursively from abstract applications down to physics reveals a massive expansion in structural complexity, where a

simple high-level loop translates down to the coordinated operations of billions of physical transistors switching at picosecond speeds.

3. Classes of Computer Systems and Their Characteristics

The rapid evolution of integrated circuit technology has driven computer design into multiple diverse computing markets. Each market is characterized by completely different price points, applications, requirements, and hardware constraints. Computer architects must design processors that specifically target the goals of these distinct classes.

3.1 The Five Mainstream Computer Classes

Patterson and Hennessy classify modern computer systems into five diverse mainstream computing environments:

- **Personal Mobile Devices (PMDs):** Wireless, hand-held, battery-powered systems with rich multimedia user interfaces, such as smartphones and tablet computers. PMD designers must balance strict cost constraints (as consumer products sell for a few hundred dollars) against aggressive energy efficiency demands to extend battery life. Because PMDs lack cooling fans, heat dissipation must be minimized through power-efficient microprocessors. Storage is implemented exclusively via non-volatile Flash memory rather than magnetic disks. PMD applications are heavily web-centric and media-oriented, requiring highly predictable, soft real-time performance (e.g., processing video frames at constant frame rates without stuttering) and minimal memory footprints.
- **Desktop Computing:** The classic PC market, encompassing low-end netbooks under $300 to heavily configured, high-end workstations up to $2500. A significant portion of the desktop market is battery-operated laptop computers. Desktop systems are designed to offer a balanced trade-off between price and performance. Designers focus on single-thread performance, rich graphics, and responsiveness, with power consumption being a secondary concern compared to PMDs since they have access to continuous power grids or larger batteries and active fan cooling.
- **Servers:** Systems designed to provide large-scale, reliable computation, database hosting, and file services for enterprise applications, replacing traditional mainframes. For servers, performance is measured as throughput (the number of concurrent transactions or queries processed per second) rather than single-thread execution speed. Most importantly, availability and reliability are critical system design issues. Because server downtime results in catastrophic financial losses (such as bank transaction processing or airline reservation failures), server hardware includes extensive error-correcting code (ECC) memory protection, dual redundant power supplies, and hot-swappable drives.
- **Clusters and Warehouse-Scale Computers (WSCs):** Extreme-scale networks of tens of thousands of cheap, redundant commodity servers acting as a single giant machine. WSCs are built by internet giants like Google, Amazon, Microsoft, and Meta to support Software as a Service (SaaS) applications, including global search engines, social media networks, video streaming, and online shopping. In total, a WSC building and its hardware can cost up to $200 million. WSC designers

prioritize throughput, high internet bandwidth, power efficiency, and energy proportionality (consuming power proportionally to the computational load). Unlike traditional servers that rely on expensive high-dependability hardware, WSC architecture relies on a robust software layer to detect, isolate, and recover from the constant hardware failures that naturally occur when operating tens of thousands of commodity nodes.

- **Internet of Things (IoT) and Embedded Computers:** The most prevalent class of computers, consisting of low-cost microprocessors embedded inside everyday machines, appliances, and industrial controllers, such as car engines, washing machines, microwaves, networking switches, and medical equipment. Embedded microprocessors can cost less than a dollar. The primary design goals are minimum cost, ultra-low power consumption (often operating for years on small coin batteries), and highly specific application-oriented performance. They typically run proprietary software that cannot be modified by the user, frequently operating under strict hard real-time constraints where missing an execution window by even a fraction of a millisecond represents a catastrophic failure.

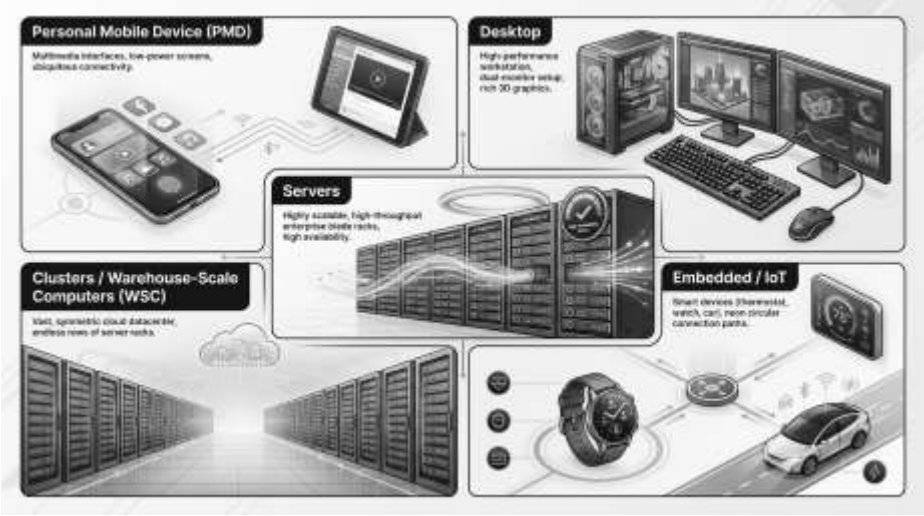


Figure 1.3 Classes of Computers

3.2 Summary of Computing Classes

To appreciate the massive differences in scale and cost, we must review the classic quantitative market characteristics of these five classes. WSCs represent the absolute pinnacle of scale, supporting massive request-level parallelism across thousands of servers, whereas embedded systems prioritize cost-efficiency in high-volume shipments.

Table 1.1 Quantitative Analysis

Feature	PMD	Desktop	Server	WSC
System Price	$100 - $1,000	$300 - $2,500	$5,000 - $10,000,000	$100,000,000 - $200,000,000
Microprocessor Price	$10 - $100	$50 - $500	$200 - $2,000	$50 - $250

Critical Design Issues	Cost, energy, media responsiveness, size	Price-performance, graphics, energy	Throughput, availability, scalability, energy	Price-performance, throughput, energy proportionality
Sales Volume (approx.)	1.6 Billion units per year	275 Million units per year	15 Million units per year	Varies (composed of commodity servers)
Typical Storage	Flash Memory	Solid-State Drive (SSD) / HDD	Redundant HDD / SSD arrays	Redundant, inexpensive commodity disk arrays

3.3 Infrastructure Scaling, SaaS, and Cloud TCO

The growth of Software as a Service (SaaS) has driven a massive shift toward commercial cloud computing. Cloud giants like Amazon Web Services (AWS) rent virtualized server nodes (known as instances) to third parties. AWS pricing models provide clear insights into the true Total Cost of Ownership (TCO) of hardware, taking into account hardware depreciation, power distribution, heat dissipation, networking bandwidth, and building maintenance over a standard 3-year amortization schedule.

For example, in standard cloud markets, a basic compute-oriented instance composed of two x86 chips with 20 physical cores rents on-demand for $1.75 per hour, amounting to approximately $17,962 over 3 years. In contrast, an accelerated instance that adds 16 NVIDIA K80 GPUs to a 36-core system rents on-demand for $15.84 per hour, or $184,780 over 3 years. These pricing differentials highlight the significant premium placed on specialized hardware acceleration (such as GPUs or TPUs) in modern data centers, where power budgets and physical space are highly constrained.

4. Operations and Operands of Computer Hardware

At the lowest layer of the software stack, programs are realized as a sequence of hardware instructions. Each machine instruction specifies a fundamental operation to be performed by the processor's electronic circuitry upon distinct data values known as operands. The hardware architecture represents these instructions as structured binary patterns (machine code) that align with the Stored Program Concept, where both instructions and data are housed in the same physical memory space.

4.1 The Opcode-Operand Relationship

A machine instruction is conceptually divided into two primary logical components:

- **Operation Code (Opcode):** Specifies the basic task or operational command to be executed by the Central Processing Unit (such as addition, subtraction, data load, or conditional branch).
- **Operands:** Specify the source values that act as inputs to the operation, and the destination location where the resulting output must be stored.

4.2 MIPS/RISC Hardware Design Philosophy

A standard uniprocessor design requires strict constraints on instruction formats to simplify the decoding hardware. For example, in the MIPS and RISC-V instruction set architectures (ISAs), every arithmetic-logical instruction is constrained to perform exactly one operation and must always specify exactly three operands (two sources and one destination).

This rigid structure is guided by a core computer architecture design principle:

> **Design Principle 1: Simplicity favors regularity.**
> By enforcing a fixed number of operands (exactly three) for all arithmetic operations, the hardware decoding circuitry is kept simple and consistent. Enforcing a regular layout avoids variable-complexity decoders, which would complicate control logic paths, increase transistor area, and severely slow down maximum clock frequencies.

To illustrate how a compiler bridges high-level programs to regular hardware structures, consider compiling two simple high-level assignment statements:

C Code Segment:

```
f = (g + h) - (i + j);
```

Because MIPS arithmetic hardware can execute only one instruction with exactly three operands at a time, the compiler must disintegrate this assignment statement into a serial pipeline of three primitive instructions, utilizing temporary registers (such as t0 and t1) to store intermediate results:

Compiled MIPS Assembly Language:

```
add t0, g, h      # Temporary register t0 = g + h
add t1, i, j      # Temporary register t1 = i + j
sub f, t0, t1     # Destination f = t0 - t1
```

4.3 Computer Hardware Operands: The Register File

To maximize execution speeds, arithmetic logic units (ALUs) cannot pull operands directly from main memory. Accessing main memory DRAM cells requires slow off-chip transfers that take up to 20,000 times more energy and latency than an arithmetic operation. Instead, operands must be loaded into high-speed, on-chip storage locations within the CPU core known as registers.

4.3.1 Properties of the Processor Register File

Registers are grouped and managed in a fast combinational hardware block called the Register File. The register file possesses multiple read and write ports, allowing the ALU to access its operands and write results back within a single, highly compressed clock cycle. In MIPS architecture:

- **Word Size:** Registers are accessed as 32-bit groups. A 32-bit unit of data is termed a Word.

- **Register Count:** The hardware supports exactly **32 independent registers**. Because **32 = 2 raised to the 5th power**, a register can be uniquely addressed using a **tiny 5-bit field in the instruction format**.
- **Register Mapping:** Software compilers map high-level variable names to physical register numbers (ranging from 0 to 31). To keep calling conventions consistent across libraries, standard application binary interfaces (ABIs) assign names to registers:
  - **s0 to s7 (Registers 16 - 23):** Saved temporary registers, used to store long-term program variables. These are preserved across function calls.
  - **t0 to t9 (Registers 8 - 15, 24 - 25):** Temporary registers, used to store scratchpad values. These are volatile and not preserved across function calls.
  - **zero (Register 0):** Hardwired to the constant value 0. Any write to register 0 is ignored by the hardware, and reading register 0 always yields 0, allowing for quick constant comparisons and register moves.

4.4 Memory Operands: Byte-Addressability and Alignment

The physical organization of the memory subsystem requires clear standards on how data is stored, addressed, and retrieved. When the processor communicates with the memory unit over the system bus, it must interpret binary byte sequences in a highly consistent manner.

While registers are extremely fast, their capacity is limited to 32 words. Large data structures, such as arrays, matrices, records, and objects, must reside in main memory. To operate on these memory-resident data structures, the processor must execute data transfer instructions that copy values back and forth between registers and memory locations.

4.4.1 Byte-Addressable Memory Organization

Virtually all modern digital computer systems utilize a byte-addressable memory assignment, meaning that a unique memory address is assigned to every individual 8-bit byte of storage. Main memory is structurally organized as a giant, single-dimensional array, with each cell storing 8 bits (1 byte) of data. This provides extremely fine-grained access to individual characters or small data fields.

In a byte-addressable system, every unique memory address refers to a single byte. However, the CPU typically processes data in wider groups of fixed size called words (ranging from 16 to 64 bits). For a 32-bit machine, a single word consists of four bytes, which means that successive words are located at aligned byte boundaries representing multiples of four (addresses 0, 4, 8, 12, etc.).

Word Address	Byte Offset +0	Byte Offset +1	Byte Offset +2	Byte Offset +3
Address 0	Byte 0	Byte 1	Byte 2	Byte 3

4.4.2 Big-Endian vs. Little-Endian Data Organization

When a multi-byte data object (such as a 32-bit integer or a 64-bit doubleword) is stored in a byte-addressable memory, the system must decide in what order to place the individual bytes across successive addresses:

- **Big-Endian Assignment:** The most significant (leftmost) byte of the word is stored at the lowest (first) memory address, while the least significant byte is stored at the highest address. This is the natural reading order for humans in western languages.
- **Little-Endian Assignment:** The least significant (rightmost) byte of the word is stored at the lowest (first) memory address, while the most significant byte is stored at the highest address. This is the standard assignment used by x86 and RISC-V systems.

Endianness Storage Comparison Example

Suppose the 32-bit hexadecimal value 0x12345678 must be stored at byte address 1000:
Here, the most significant byte is 0x12, and the least significant byte is 0x78.

Big-Endian Storage Mapping:
- Address 1000: 0x12 (Most Significant Byte)
- Address 1001: 0x34
- Address 1002: 0x56
- Address 1003: 0x78 (Least Significant Byte)

Little-Endian Storage Mapping:
- Address 1000: 0x78 (Least Significant Byte)
- Address 1001: 0x56
- Address 1002: 0x34
- Address 1003: 0x12 (Most Significant Byte)

4.4.3 Word Boundaries and Natural Memory Alignment

Natural word boundaries occur at addresses that are clean multiples of the size of the data object. For example, a 2-byte halfword is naturally aligned if its address is a multiple of 2; a 4-byte word is naturally aligned if its address is a multiple of 4; and an 8-byte doubleword is naturally aligned if its address is a multiple of 8.

While some processors permit misaligned accesses (allowing a 4-byte word to be fetched from address 1001), doing so incurs a severe performance penalty. If an object is misaligned and spans across a natural boundary (such as a 4-byte word spanning from address 1002 to 1005), the memory controller must perform two successive memory accesses to fetch the two parts of the word, recombine them using a physical alignment shifter network, and route them to CPU registers. Naturally aligned accesses, by contrast, are guaranteed to complete in a single memory cycle, significantly reducing pipeline lat

4.4.4 Base-Displacement Address Calculations

A load word (lw) or store word (sw) instruction specifies a memory location using base-displacement addressing. The effective memory address is calculated by adding a starting address stored in a base register to a fixed constant offset (the displacement) embedded in the instruction:

Effective Address = Contents of Base Register + Sign-Extended Offset

To see how this works, we analyze the compilation of a C statement accessing an array element. Suppose the variable 'h' is associated with register s2 (register 18), and the base starting address of array 'A' is stored in register s3 (register 19):

C Code Segment:

```
A[12] = h + A[8];
```

To compile this code, the compiler calculates the byte offsets for elements 8 and 12 of the 32-bit integer array. Because each integer occupies 4 bytes, the physical offsets are:

- **Offset for A[8] =** 8 × 4 = 32 bytes.
- **Offset for A[12] =** 12 × 4 = 48 bytes.

The resulting MIPS assembly sequence executes three sequential phases:

Compiled MIPS Assembly Code:

```
lw  t0, 32(s3)    # Step 1: Load word from address [s3 + 32] (A[8]) into temp
                            register t0
add t0, s2, t0    # Step 2: Add register s2 (h) to t0, placing the sum in t0
sw  t0, 48(s3)    # Step 3: Store word from t0 back into address [s3 + 48]
                            (A[12])
```

4.5 Constant and Immediate Operands

Computer programs make frequent use of constants within loops, comparisons, and counter increments (such as adding 1 to a index counter). In a baseline load-store model, utilizing a constant would require loading the constant value from memory into a register before running the addition. This overhead slows execution down and wastes power.

To solve this, instruction set architectures support immediate operands, where a 16-bit constant is embedded directly inside the instruction word. This is governed by a core design principle:

> **Design Principle 3: Make the common case fast.**
> Since constant-based arithmetic is highly common in programming, incorporating immediate fields directly inside instructions avoids the need to fetch constants from data memory. This makes constant execution drastically faster, reduces processor bus traffic, and minimizes chip dynamic power consumption.

To leverage immediate operands, MIPS provides the Add Immediate (addi) instruction, where the third operand is a constant embedded directly in the instruction word:

MIPS Assembly:

```
addi s3, s3, 10    # Register s3 = s3 + 10 (executed in a single step)
```

4.6 Summary Reference: Operations and Operands

The following structured reference table summarizes the primary instruction categories, their symbolic assembly syntax, physical hardware operation mappings, and operand behaviors in uniprocessor design:

Category	MIPS Instruction Syntax	Mathematical Operation	Operand Locations
Arithmetic	add s1, s2, s3 sub s1, s2, s3	s1 = s2 + s3 s1 = s2 - s3	Register file only (three-operand format)
Immediate	addi s1, s2, 50	s1 = s2 + 50	Two registers + 16-bit constant in instruction word
Load Word	lw s1, 50(s2)	s1 = Memory[s2 + 50]	Destination register s1, base s2, 16-bit offset constant

5. Instruction Set Architecture Principles and Classes

The Instruction Set Architecture (ISA) is the most critical conceptual boundary in computer systems design. It serves as the abstract interface between the physical processor hardware and the software that runs on it. The ISA defines the complete set of programmer-visible instructions, the registers available for temporary storage, memory addressing models, data types, and logical execution behaviors, ensuring that a software program compiled for a specific ISA can run on any physical hardware implementation of that processor family.

5.1 The Seven Dimensions of an ISA

To fully specify a new instruction set architecture, computer designers must define seven independent abstract dimensions:

1. **Class of ISA** Nearly all modern general-purpose processors are classified as General-Purpose Register (GPR) architectures, meaning that all operands are explicitly named in registers or memory. GPR designs are divided into Register-Memory architectures (such as the x86, which can reference memory operands directly as part of arithmetic instructions) and Register-Register or Load-Store

architectures (such as ARM and RISC-V, which restrict memory access to dedicated load and store instructions, requiring all arithmetic operations to occur strictly on register operands).

2. **Memory Addressing** Determines how memory addresses are interpreted and accessed. Virtually all modern computers are byte-addressable, providing access to 8-bit bytes, 16-bit halfwords, 32-bit words, and 64-bit doublewords. An important design choice is whether accesses must be naturally aligned in memory. An access to an object of size 's' bytes at byte address 'A' is aligned if 'A mod s = 0'. Many architectures require aligned accesses because they are simpler to route through physical data buses and execute significantly faster than misaligned accesses.

3. **Addressing Modes** Define the mathematical techniques used by instructions to specify the memory addresses of operands. Standard modes include immediate (specifying constants directly in the instruction), register, and displacement (where a constant offset is added to the contents of a register).

4. **Types and Sizes of Operands** Defines the hardware-supported data formats. Common types include 8-bit bytes (ASCII characters), 16-bit halfwords (Unicode), 32-bit words (standard integers), 64-bit doublewords (long integers), and IEEE 754 floating-point representations in single precision (32-bit) and double precision (64-bit).

5. **Operations** The fundamental classes of instructions supported by the CPU. These include data transfer (moving data between registers and memory), arithmetic/logical (addition, logical AND/OR, shifts), control flow (conditional branches and jumps), and system operations (operating system calls, virtual memory management, and hardware privilege control).

6. **Control Flow Instructions** Manage the execution path of programs. This dimension specifies how target addresses are calculated for conditional branches (using a PC-relative offset added to the program counter) and unconditional jumps (using absolute addresses or register indirect jumps for function pointers and procedure returns).

7. **Encoding an ISA** The bit-level representation of instructions. The architect must choose between fixed-length encoding (where every instruction is exactly 32 bits long, simplifying hardware decoding at the cost of program size, as in classic RISC-V or ARM) and variable-length encoding (where instructions vary from 1 to 18 bytes, maximizing code density at the cost of highly complex decoding hardware, as in the x86). Hybrid approaches use a mix of 16-bit and 32-bit instructions (such as ARM Thumb-2 or RISC-V Compressed RV64IC) to achieve a code size reduction of up to 40%.

5.2 Historical and Structural Classes of ISAs

The history of computer design features four primary structural classes of instruction set architectures, distinguished by where the ALU fetches its operands and where it stores the result:

- **Stack Architecture:** All operands are implicitly located on the top of a Last-In, First-Out (LIFO) stack. The ALU pops the top two operands, performs the operation, and pushes the result back onto the stack. No explicit operand names are used in arithmetic instructions. While stack architectures yield extremely compact code sizes, they are highly sequential and extremely difficult to pipeline in hardware.

- **Accumulator Architecture:** Features a single, dedicated register called the Accumulator (AC). One operand of any arithmetic instruction is implicitly the accumulator, which also serves as the destination. The second operand is explicitly fetched from a memory address. This model simplifies CPU circuitry but requires constant memory accesses, creating a severe memory bottleneck.
- **Register-Memory Architecture:** Permits one operand of an arithmetic instruction to be located in a general-purpose register, while the other is fetched directly from memory. This style reduces the total instruction count in a program but results in variable-latency instruction execution, complicating pipeline hazard resolution.
- **Register-Register (Load-Store) Architecture:** The standard model for all modern processor designs. Arithmetic and logical operations can only operate on data stored in general-purpose registers. Separate, dedicated load (LD) and store (SD) instructions are the only operations permitted to access memory. It offers highly predictable, single-cycle instruction execution times, simple addressing modes, and is extremely easy to optimize for deep hardware pipelining.

5.3 Case Study: C = A + B Across ISA Classes

To clearly contrast these four historical classes, we review the required assembly code sequence to compute C = A + B (assuming A, B, and C are memory variables that must not be altered):

Code Sequence Comparison for C = A + B

Stack Architecture:

```
PUSH A      # Push memory operand A onto top of stack
PUSH B      # Push memory operand B onto top of stack
ADD         # Pop A and B, add them, push sum back to stack
POP C       # Pop top of stack (sum) and store into memory C
```

Accumulator Architecture:

```
LOAD A      # Copy memory operand A into the Accumulator
ADD B       # Add memory operand B to Accumulator; Accumulator = A + B
STORE C     # Copy Accumulator contents into memory location C
```

Register-Memory Architecture:

```
LOAD R1, A  # Load memory operand A into Register R1
ADD R1, B   # Add memory operand B to R1; R1 = R1 + B
STORE R1, C # Store register R1 (sum) into memory location C
```

Register-Register (Load-Store) Architecture:

```
LOAD R1, A  # Load memory operand A into Register R1
LOAD R2, B  # Load memory operand B into Register R2
ADD R3, R1, R2 # Add registers R1 and R2, store sum in R3
STORE R3, C # Store register R3 (sum) into memory location C
```

This comparison demonstrates a clear trade-off: the Register-Register (Load-Store) architecture requires four instructions compared to only three in the Register-Memory and Accumulator architectures, resulting in a larger program code size. However, because the Register-Register instructions are uniform and simple, they execute significantly faster in pipelined hardware, yielding a dramatically lower clock cycle time and lower overall execution time than their complex counterparts.

6. RISC vs. CISC Architectures

The evolution of modern microprocessors has been shaped by the intense debate and competing philosophies of two primary design methodologies: Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC).

6.1 Philosophical and Design Trade-offs

RISC and CISC design philosophies address completely different historical constraints in computer technology:

Historically, when programs were written directly in assembly language and memory was extremely expensive and limited, hardware designers prioritized minimizing program code size. This led to the CISC philosophy, which aims to provide complex, highly powerful instructions that perform multiple tasks (such as loading from memory, performing an addition, and storing the result back to memory in a single instruction). To implement these complex instructions, CISC processors rely on extensive microcode ROM inside the control unit, which decodes the instructions and coordinates multi-cycle execution.

With the rise of high-level programming languages and optimizing compilers, the necessity of assembly-level programming declined. Furthermore, compiler writers found it extremely difficult to generate code that effectively used complex, highly specific CISC instructions. Studies revealed that compilers primarily generated simple instructions (such as loads, stores, additions, and branches), leaving a vast majority of the complex CISC instructions unused. This paved the way for the RISC philosophy in the early 1980s, pioneered by David Patterson at Berkeley and John Hennessy at Stanford. RISC prioritizes simplicity, uniform fixed-length instructions, single-cycle execution, and an extensive general-purpose register file, transferring the complexity of instruction scheduling and optimization from physical hardware to the compiler.

6.2 CPU Performance Perspective

To analyze the performance trade-offs, we must evaluate both architectures against the fundamental CPU Performance Equation:

The CPU Performance Equation

CPU Time = (Instructions / Program) x (Cycles / Instruction) x (Seconds / Cycle)

CISC architectures attempt to minimize the first variable, the number of instructions per program, by compiling code into dense, complex instructions. However, because these instructions vary in execution path, they result in a highly complex physical implementation, which significantly increases the second and third variables: Cycles Per Instruction (CPI) and Clock Cycle Time (Seconds per Cycle).

Conversely, RISC architectures prioritize a very low CPI (targeting exactly 1.0 cycle per instruction) and a highly optimized clock cycle time through a simple, easily pipelined hardware path. Although this simplicity results in a larger compiled code size (more instructions per program), the reduction in CPI and Clock Cycle Time heavily dominates, enabling RISC processors to achieve significantly higher performance than pure CISC designs.

6.3 Comprehensive RISC vs. CISC Comparison Matrix

The physical and structural differences between modern implementations of RISC and CISC designs are summarized in the comparative matrix below:

Design Parameter	RISC Architecture	CISC Architecture
Instruction Set Size	Small, uniform set of simple instructions	Large, highly rich set of complex instructions
Instruction Length	Fixed length (typically exactly 32 bits)	Variable length (from 1 to 18 bytes)
Execution Time	Single-cycle execution (typical CPI of 1.0)	Multiple cycles per instruction
Memory Reference	Strictly load/store; only LD/SD access memory	Many instructions can reference memory directly
Addressing Modes	Simple, highly limited modes (immediate, displacement)	Complex and many addressing modes
Registers Available	Large register file (typically 32 general-purpose)	Fewer registers; rely on memory for storage
Control Unit Design	Hardwired control (highly optimized for speed)	Microprogrammed control (microcode ROM)
Pipelining	Highly optimized, extremely easy to implement	Highly complex due to instruction size variations
Primary Examples	ARM, RISC-V, MIPS	x86, Intel 8086, VAX, IBM 360/370

It is important to note that modern processor designs have heavily merged these two philosophies. For example, Intel x86 processors, while maintaining a CISC instruction set externally to ensure backward compatibility with legacy software binaries, utilize hardware inside the chip to dynamically translate complex x86 instructions into simple, RISC-like operations (called micro-operations) which are then scheduled and pipelined at extremely high speeds. Conversely, modern RISC processors have

incorporated specialized complex operations (such as vector arithmetic and media extensions) to boost application-specific performance.

ency.

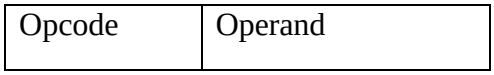
7. Analysis of Memory Addressing Modes

Addressing modes are the various mathematical techniques defined in the Instruction Set Architecture to specify where an operand is located. The operand can reside within the instruction itself, inside a processor register, or at a specific address in main memory. The address in main memory where the operand is physically located is called the Effective Address (EA). Different modes represent trade-offs between instruction length, programming flexibility (for complex data structures), and memory latency.

7.1 Detailed Breakdown of the Nine Major Addressing Modes

1. **Immediate Addressing** The operand is explicitly specified as a constant value directly within the instruction itself. No memory reference other than the instruction fetch is required, making this the fastest possible mode. It is represented symbolically with a hash sign (#) in some assembly languages (e.g., ADD #3 or MOVE #200, R0) and used to initialize variables or define constants. Its disadvantage is that the size of the operand is strictly limited by the instruction's immediate field size.

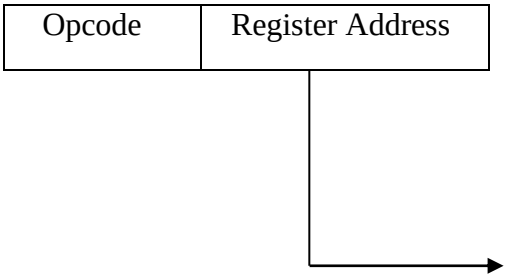
Instruction

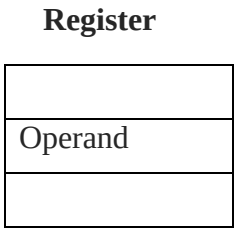


- When any instruction with two operands uses immediate addressing, the first operand may be a register or memory location, and the second operand is an immediate value.
- Since the instruction does not require an extra memory access to fetch the operand, it executes faster.
- Example: ADD 5 // ACC ← ACC + 5

2. **Register Addressing** The operand is stored directly within a general-purpose processor register, and the instruction specifies the register number.

Instruction

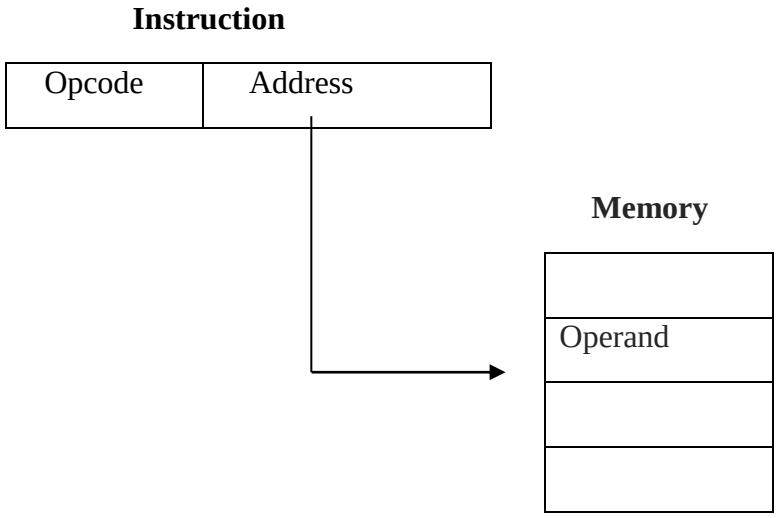




- Example: ADD R1, R2, R3

Because the CPU can access its registers at electronic speeds within a fraction of a clock cycle, register-to-register operations are extremely fast. Since registers are limited in number (typically 8 to 32), only a very small address field (3 to 5 bits) is needed in the instruction format, reducing overall instruction length (e.g., MOV R1, R2).

3. **Direct or Absolute Addressing** The effective address of the memory operand is explicitly given in the address field of the instruction (e.g., MOVE LOC, R2). The CPU fetches the instruction, extracts the address field, and performs a single memory reference to retrieve the operand. This mode is simple but provides only a highly limited address space because the address field size in a single-word instruction is usually much smaller than the full word length of the machine.



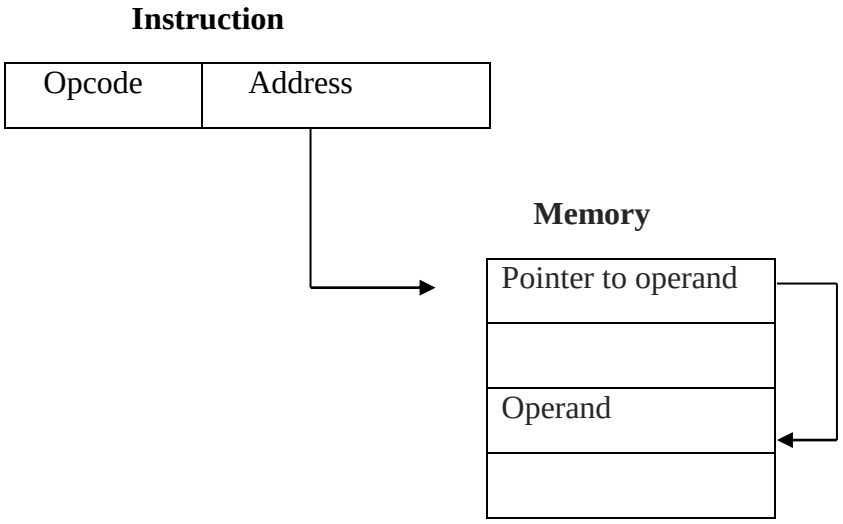
- One of the operands in the instruction refers to a memory location and the other operand references a register.
- Example:

  ADD A

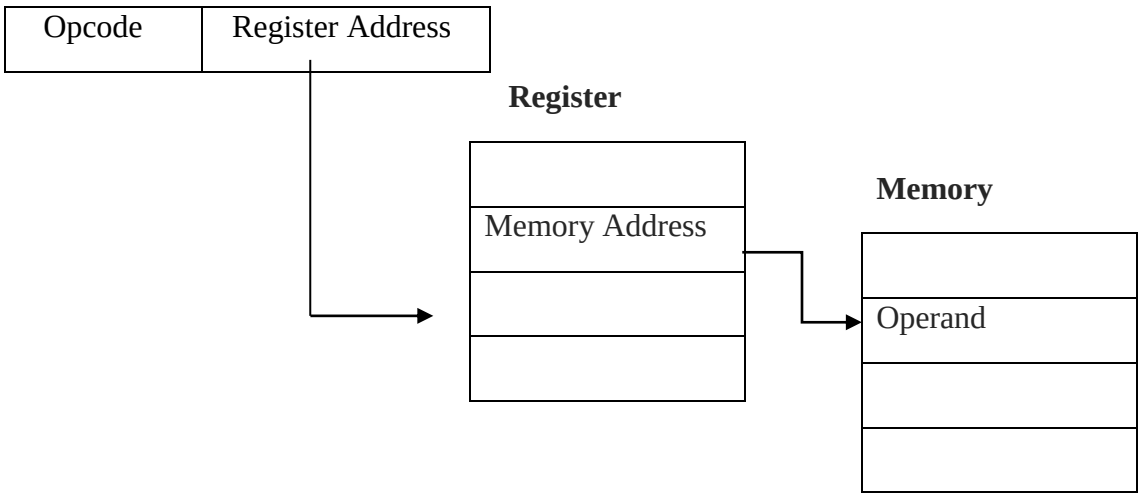
4. **Indirect Addressing** The address field of the instruction refers to a memory location which in turn contains the full-length effective address of the operand (represented symbolically as EA = (A)). The control unit must first fetch the instruction, access memory at address 'A' to retrieve the effective address, and then access memory a second time using that effective address to fetch the operand. While this mode overcomes address space limitations, it is extremely slow, requiring two memory accesses. It can also be cascaded as multilevel indirect addressing.

- In indirect addressing mode, the offset value is specified by the pointer available in the instruction.

- The address field contains the effective address of the operand.
  Effective Address, EA = (A)

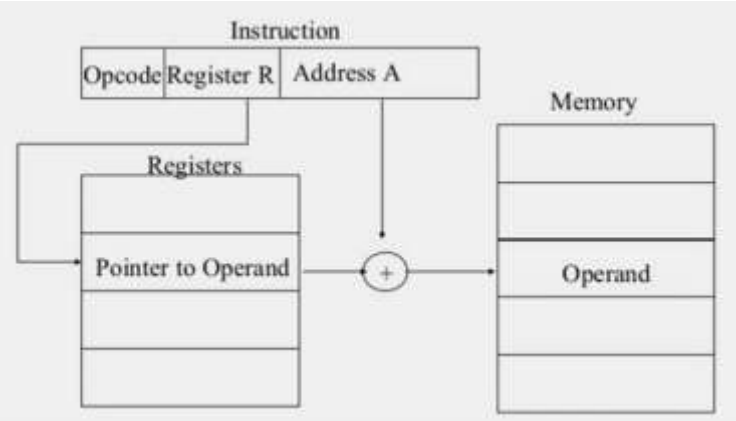


5. **Register Indirect Addressing** A highly common variation of indirect addressing. The instruction specifies a general-purpose register that contains the effective address of the operand in memory (represented as EA = [Ri], e.g., MOV AL, [BX]). The control unit reads the address from the register and performs a single memory access to fetch the data. This requires one fewer memory access than pure indirect addressing and is heavily used for accessing data pointed to by pointers in high-level languages.



6. Displacement Addressing Combines the capabilities of direct addressing and register indirect addressing. The instruction contains two address fields: an explicit base offset 'A' and a register reference 'R'. The effective address is computed by adding the constant offset to the contents of the register: EA = A + [R]. It is represented symbolically as X(Ri). Displacement addressing is divided into three primary types:

- **Relative Addressing:** The implicitly referenced register is the Program Counter (PC). The effective address is the sum of the constant displacement 'X' and the current address in the PC: EA = X + [PC]. This is heavily used for conditional branch instructions to specify a branch target location, facilitating run-time program relocation.
- **Base-Register Addressing:** The specified register contains a main memory base address, and the displacement field contains an offset. This is convenient for segmenting memory or implementing local variable storage blocks.
- **Indexed Addressing:** The constant field contains the base address of a data array in memory, and the register (the index register) contains a positive displacement representing the index of the element being accessed: EA = A + [Index]. The index register can be easily incremented in loops to access consecutive array elements.



7.Stack Addressing The operand is implicitly located on top of the system stack. The stack pointer (SP) is a dedicated CPU register that holds the memory address of the top of the stack. Stack operations (PUSH and POP) implicitly increment or decrement the SP. This mode is also known as implicit addressing and is standard in stack-organized machines.

8. Auto-Increment Addressing Similar to register indirect addressing, but the register is automatically incremented after its value is used to access memory (symbolically written as (Ri)+). This is extremely useful for stepping through tables or arrays of data in sequential loops. After accessing the operand at address [Ri], the register is automatically incremented by the size of the accessed data element 'd'.

9. Auto-Decrement Addressing The reverse of auto-increment. The contents of the specified register are first automatically decremented by the size of the data element 'd', and then used as the effective address to access memory (symbolically written as -(Ri)). Together, auto-increment and auto-decrement are used to implement software stacks in general-purpose architectures.

7.2 Real-World Applications of Addressing Modes

Choosing the correct addressing mode is critical for **implementing high-level programming language data structures efficiently**.

- **Pointers and reference variables** rely heavily on **Indirect and Register Indirect addressing** modes.
- **Array data structures** are best supported using **Indexed addressing**, where the base address points to the start of the array and the index register represents the loop variable multiplied by the element size.
- Finally, **loop iterations and stack frames** are naturally managed using **Auto-increment, Auto-decrement, and Stack addressing modes.**

7.3 Step-by-Step Relative Addressing Effective Address Calculation

To ensure a clear, mathematical understanding of displacement calculations, we review the classic PC-relative branch target example from the curriculum:

PC-Relative Addressing Math Example

Assume the following initial conditions:

- The Program Counter (PC) contains the address 825.
- The address field of the branch instruction contains the displacement value 24.

Step-by-Step Effective Address (EA) Derivation:

1. During the Fetch Phase, the control unit reads the instruction from memory address 825.
2. Immediately after fetching the instruction, the hardware automatically increments the PC by one word length (to point to the next sequential instruction), so the PC becomes 826.
3. During the Execution Phase, the effective address is computed by adding the PC and displacement:

EA = A + PC
EA = 24 + 826
EA = 850

Result:

The CPU successfully branches by copying the calculated effective address 850 into the PC, causing the next instruction to be fetched from memory location 850.

7.4 Summary Comparison Matrix of Addressing Modes

The operational characteristics and memory latency costs of the standard addressing modes are summarized below:

Mode Name	Syntax	Effective Address (EA)	Memory Accesses	Primary Use Case
Immediate	#Value	None (Operand in instruction)	0	Constant initializations
Register	Ri	None (Operand in Register)	0	Active variable operations
Direct (Absolute)	LOC	EA = LOC	1	Static global variable access
Indirect	(LOC)	EA = [LOC]	2	Pointers, reference variables
Register Indirect	(Ri) or [Ri]	EA = [Ri]	1	Pointer dereferencing
Displacement	X(Ri)	EA = X + [Ri]	1	Local variables on stack
Indexed	A(Ri)	EA = A + [Ri]	1	Array element access
Auto-increment	(Ri)+	EA = [Ri]; Ri = Ri + d	1	Loop-based array sweeps
Auto-decrement	-(Ri)	Ri = Ri - d; EA = [Ri]	1	Software stack operations

8. Assembly Language Programming

8.1 Assembly Language Syntax and Structure

Unlike high-level languages where statements are nested and structurally complex, assembly language consists of a linear sequence of instructions. Each line represents a single hardware-level statement and is structured around four potential fields:

```
[Label:]  Mnemonic  [Operand_1, Operand_2, Operand_3]  [# Comments]
```

- **Label:** An optional field used to mark a specific instruction or data address in memory. Labels always begin in the first column, are followed by a colon (e.g., `Loop:`), and serve as human-readable symbols representing jump or branch targets. The assembler translates these into actual address offsets.
- **Mnemonic:** The symbolic name for the hardware operation to be performed. Examples include arithmetic operations (`add, sub`), data transfers (`lw, sw`), logical operations (`and, or`), and control flow branches (`beq, bne`).
- **Operands:** The source and destination targets for the instruction. Operands can be general-purpose registers (such as `$s0, $t1`), memory addresses, or constant values (immediate operands, e.g., `100`).
- **Comments:** Prefaced by the # symbol. Comments are completely ignored by the assembler but are essential for documenting low-level hardware routines.

8.2 Assembler Directives

Assembler directives are special instructions embedded in the source code that are prefixed with a period (.). They are not executed by the physical processor CPU at run-time but are used to guide the assembler's actions during code translation:

- **.data:** Marks the beginning of the data segment where static and global program variables are declared and allocated in binary representation.
- **.text:** Marks the beginning of the text segment containing the raw symbolic assembly code instructions to be translated into machine instructions.
- **.word:** Allocates space in memory and initializes successive 32-bit (4-byte) integer words.
- **.align:** Specifies the byte alignment boundary for data items (e.g., aligning variables to multiples of 4 bytes) to satisfy hardware alignment restrictions.
- **.asciiz:** Stores a null-terminated (zero-terminated) character string in successive bytes of the data segment, standard for printing text.

8.3 CPU Registers and Register Files

In general-purpose register (GPR) architectures, arithmetic and logical operations occur strictly on values stored in processor registers. The MIPS architecture supports **32 registers**, each 32 bits (1 word) wide, grouped in a state element called the **register file**. Registers are extremely fast compared to primary memory (SRAM/DRAM) because they reside on-chip with the ALU. Registers are referred to symbolically in assembly to make programming easier. The table below represents the standard symbolic registers, physical numbers, and their compiler calling conventions:

Register Name	Physical Number	Primary Usage / Role	Calling Convention
`$zero`	`0`	Constant value 0 (Hardwired)	N.A. (Read-only)
`$at`	`1`	Reserved for the assembler (macro expansions)	N.A.
`$v0 - $v1`	`2 - 3`	Expression evaluation and function return values	Caller-saved
`$a0 - $a3`	`4 - 7`	Function argument registers	Caller-saved
`$t0 - $t7`	`8 - 15`	Temporary registers (unpreserved across calls)	Caller-saved
`$s0 - $s7`	`16 - 23`	Saved temporary registers (must be preserved)	Callee-saved
`$t8 - $t9`	`24 - 25`	Additional temporary registers	Caller-saved
`$k0 - $k1`	`26 - 27`	Reserved for the Operating System kernel	N.A.
`$gp`	`28`	Global Pointer (points to static global area)	N.A.
`$sp`	`29`	Stack Pointer (points to active top-of-stack)	Callee-saved

$ra	31	Return Address (stores procedure return target)	Caller-saved

8.3.1 Caller-Saved vs. Callee-Saved Conventions

- **The Caller**: The active function that pauses execution to invoke another function.
- **The Callee**: The target function that runs, processes the arguments, and returns control.

To ensure that independent compiled procedures do not corrupt each other's active variables during nested function calls, computer ABIs(Application Binary Interfaces; an API (source code level), an ABI works at the compiled binary level) define strict rules regarding register preservation:

- **Caller-Saved Registers ($t0-$t9, $v0-$v1, $a0-$a3):** These registers are considered volatile. If a calling function relies on their values after making a subroutine call, it must save them onto the system stack before the call and pop them back off after the procedure returns.
- **Callee-Saved Registers ($s0-$s7, $sp, $fp):** These registers are preserved. If a called subroutine needs to use any of these registers, it must first push their current values onto the stack to save them, and restore them before returning, leaving the caller's active variables uncorrupted.

8.4 Instruction Set Encodings and Binary Formats

A core hallmark of Reduced Instruction Set Computers (RISC) is that all instructions are of a single, rigid length (exactly **32 bits**) to simplify pipelined instruction decoding. To support different sets of operands, MIPS partitions this 32-bit container into three distinct structural instruction formats:

8.4.1 R-Format (Register-to-Register Operations)

R-type instructions are used for arithmetic and logical operations where all operands are located in CPU registers. The 32 bits are divided as follows:

op	rs	Rt	rd	shamt	funct
6 bits	*5 bits*	*5 bits*	*5 bits*	*5 bits*	*6 bits*

Field Explanations:
- op (6 bits): Opcode specifying the instruction class (always 0 for R-type instructions).
- rs (5 bits): First source register operand.
- rt (5 bits): Second source register operand.
- rd (5 bits): Destination register operand (gets the arithmetic result).
- shamt (5 bits): Shift amount (used for logical shift instructions, otherwise 0).
- funct (6 bits): Function code selecting the specific ALU operation (e.g., 32 for add, 34 for subtract).

8.4.2 I-Format (Immediate, Load, Store, and Branch Operations)

I-type instructions are used for operations involving constant values or offset-based memory addressing. The 32 bits are divided as follows:

op	rs	rt	immediate / address field
6 bits	*5 bits*	*5 bits*	*16 bits*

Field Explanations:

- op (6 bits): Opcode specifying the exact command (e.g., 8 for addi, 35 for lw, 43 for sw, 4 for beq).
- rs (5 bits): Source register containing the base address (for loads/stores) or the first operand (for arithmetic).
- rt (5 bits): Target register. For loads, it acts as the destination; for stores, it acts as the source register containing the value to be written to memory.
- immediate (16 bits): A 16-bit constant field (treated as signed displacement offset, allowing values from -32768 to 32767).

8.4.3 J-Format (Unconditional Jumps)

J-type instructions are used for direct unconditional jumps. The 32 bits are divided as follows:

op	target address field
6 bits	*26 bits*

8.5 Core Assembly Instruction Categories

In this section, we examine the symbolic syntax, binary mapping mechanics, and physical memory operations of the three core MIPS instruction categories:

8.5.1 Data Transfer Instructions

*In load-store architectures, arithmetic can only occur on registers. Therefore, dedicated load and store instructions are required to transfer variables between registers and memory. Words must start at byte addresses that are multiples of 4 (***alignment restriction***).*

```
lw  $t0, 32($s3)   # Load Word: $t0 = Memory[$s3 + 32]
sw  $t0, 48($s3)   # Store Word: Memory[$s3 + 48] = $t0
```

The memory address is dynamically computed at execution by adding the 16-bit sign-extended immediate offset value (e.g., 32 or 48) to the contents of the base register ($s3).

8.5.2 Arithmetic and Logical Instructions

Arithmetic operations must always have exactly three operands, conforming to the philosophy of keeping physical hardware simple:

```
add   $t0, $s1, $s2  # Register-Register: $t0 = $s1 + $s2
addi  $s3, $s3, 10   # Immediate Addition: $s3 = $s3 + 10
sll   $t1, $s3, 2    # Shift Left Logical: $t1 = $s3 << 2 (computes $s3 * 4)
```

8.5.3 Control Flow, Branching, and Procedure Calls

Control flow instructions redirect program execution by modifying the Program Counter (PC):

```
beq  $t0, $s5, Exit  # Branch if Equal: if ($t0 == $s5) jump to Exit label
bne  $t0, $s5, Loop  # Branch if Not Equal: if ($t0 != $s5) jump to Loop label
j    Loop            # Unconditional Jump: always jump to Loop label
jal  PrintString     # Jump and Link: save (PC + 4) in $ra, jump to PrintString
jr   $ra             # Jump Register: jump unconditionally to address in $ra
```

8.6 Case Study: Mapping High-Level Constructs to Assembly

To understand compiler mapping behavior, we trace how standard C programming constructs (conditional If-Else boundaries and While loop bounds) are translated into symbolic assembly instructions.

8.6.1 C Conditional If-Else Statement

Let variables f, g, h, i, and j be assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively:

```
// High-Level C Code:
if (i == j)
    f = g + h;
else
    f = g - h;
```

```
# Compiled MIPS Assembly:
        bne  $s3, $s4, Else   # Branch if i != j (opposite test is more efficient)
        add  $s0, $s1, $s2   # If-part: f = g + h
        j    Exit            # Jump over the else-part
Else:   sub  $s0, $s1, $s2   # Else-part: f = g - h
Exit:
```

8.6.2 C While Loop Statement

Let loop variables i and k correspond to registers $s3 and $s5, and let the base starting address of the integer array save be stored in register $s6:

```
// High-Level C Loop Code:
while (save[i] == k)
    i += 1;
```

```
# Compiled MIPS Assembly:
Loop:   sll  $t1, $s3, 2    # Temp reg $t1 = 4 * i (converts index to byte offset)
        add  $t1, $t1, $s6    # $t1 = Address of save[i] (base + offset)
```

```
        lw   $t0, 0($t1)     # Load $t0 = save[i] from memory
        bne  $t0, $s5, Exit   # Exit loop if save[i] != k
        addi $s3, $s3, 1     # Increment i = i + 1
        j    Loop            # Jump back to top of Loop
Exit:
```

8.7 Pseudo-instructions vs. Machine Instructions

To assist the programmer and simplify code writing, assemblers support **pseudo-instructions**. These are symbolic instructions that are not physically implemented in CPU hardware. During the first assembly pass, the assembler automatically expands or disintegrates each pseudo-instruction into one or more primitive physical machine instructions:

- **Load Immediate (li $t0, 100):** MIPS does not support loading constants directly from memory. The assembler translates this into a single register-immediate addition using the hardwired zero register: `addi $t0, $zero, 100`
- **Register Move (mv $s0, $t1):** Translates into an OR or ADD instruction against the zero register: `or $s0, $t1, $zero`
- **Branch on Less Than (blt $s1, $s2, Target):** MIPS hardware only supports equal or not-equal hardware branch decoders. The assembler expands this into a two-instruction physical sequence using the reserved register `$at` (Register 1):
  - `slt  $at, $s1, $s2    # Set on Less Than: $at = 1 if $s1 < $s2, else 0`
  - `bne  $at, $zero, Target # Branch: if $at != 0, jump to Target`

9. The Program Translation Hierarchy

A fundamental challenge in computer systems design is that while human developers write complex programs in abstract, high-level languages like C or C++, the physical CPU hardware can only understand and execute binary machine language (strings of 0s and 1s). To bridge this semantic gap, computer systems rely on a chain of system software programs known as the Program Translation Hierarchy. This pipeline systematically translates, links, and loads high-level source files into running machine memory.

9.1 The Program Translation Pipeline

The conversion of source code into an active memory process is performed in four sequential stages, managed by specific system utilities:

1. **Compiler** Takes high-level source files (.c) as input and translates them into an optimized symbolic assembly language program (.s). The compiler is responsible for mapping complex abstract programming structures (such as while loops, conditional if-else statements, and object scopes) into linear machine instructions, and optimizing register allocations to minimize memory stalls.

2. **Assembler** Converts the symbolic assembly language program into a binary object module (.o), which contains machine-readable binary instructions, data segments, and structural metadata. The

assembler resolves symbolic mnemonics (like ADD or LW) into their raw hardware binary opcodes. To resolve symbolic labels representing branch targets, standard assemblers perform a two-pass read of the code:
- First Pass: The assembler sweeps the entire assembly file, counting instruction sizes, to construct a 'Symbol Table' that maps every symbolic label to its calculated binary offset address.
- Second Pass: The assembler uses the Symbol Table to replace all label occurrences with their binary representation, generating the final object code. Unresolved external references (functions defined in other source files) are cataloged in a relocation table for the linker.

3. **Linker (Link Editor)** Enables separate compilation, allowing large programming projects to be split across independent modules. The linker 'stitches' independently assembled object modules and pre-compiled system libraries into a single, cohesive binary executable file (.out or .exe). It allocates unique memory space for each module, resolves all external symbolic references across files, patches absolute memory addresses, and merges standard library routines.

4. **Loader** A critical component of the operating system that prepares the executable program for active run-time execution. When a program is launched, the loader reads the executable's headers to determine its text and data segment sizes, allocates a continuous block of virtual memory in RAM, copies the binary instructions and static variables into their allocated segments, initializes the stack pointer (SP) register, passes command-line arguments onto the stack, and jumps CPU control to the program's initial execution entry point.

9.1.1. Compiler

A **compiler** converts a high-level programming language (such as C) into an **assembly language** program.

- Takes a source file (.c) as input.
- Produces an assembly file (.s) as output.
- Converts programming constructs like **loops (while, for)**, **if-else statements**, and **functions** into machine-level instructions.
- Optimizes the program to improve execution speed and reduce memory usage.

Example:
`program.c` → Compiler → `program.s`

9.1.2. Assembler

An **assembler** converts an assembly language program into **machine code (binary)**.

- Takes an assembly file (.s) as input.
- Produces an object file (.o) as output.
- Converts assembly instructions (such as `ADD`, `SUB`, `LW`) into binary instructions.
- Uses a **two-pass process** to resolve labels.

Two-Pass Assembly

First Pass

- Reads the entire assembly program.
- Finds all labels.
- Creates a **Symbol Table** that stores the address of each label.

Second Pass

- Replaces labels with their actual addresses using the Symbol Table.
- Generates the object code.
- Stores unresolved external references in a **Relocation Table** for the linker.

Example:
`program.s` → Assembler → `program.o`

9.1.3. Linker (Link Editor)

A **linker** combines multiple object files and library files into a **single executable program**.

- Takes object files (.o) as input.
- Combines separately compiled modules.
- Resolves external function and variable references.
- Assigns memory addresses to different modules.
- Produces the final executable file (.exe or .out).

Example:
`file1.o + file2.o + library.o` → Linker → `program.exe`

9.1.4 Loader

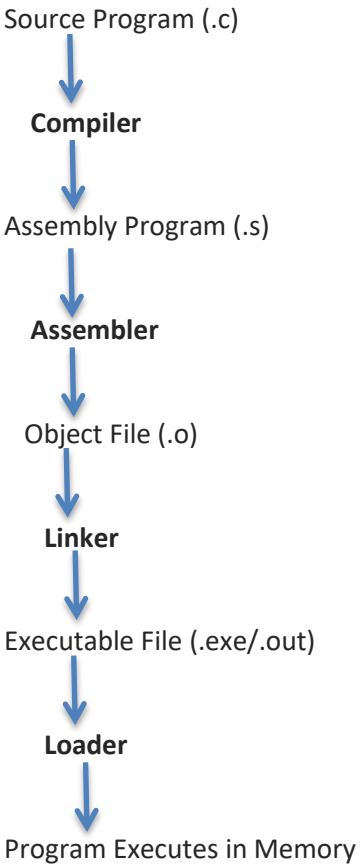
A **loader** is part of the operating system that loads the executable program into memory for execution.

- Reads the executable file.
- Allocates memory for the program.
- Loads the program's instructions and data into RAM.
- Initializes the stack and required registers.

- Passes command-line arguments (if any).
- Transfers control to the program's starting point so execution begins.

Example:
`program.exe` → Loader → Program runs in memory

Component	Input	Output	Main Function
Compiler	Source code (.c)	Assembly code (.s)	Converts high-level language into assembly language and optimizes the code.
Assembler	Assembly code (.s)	Object file (.o)	Converts assembly instructions into machine code and creates object files.
Linker	Object files (.o)	Executable file (.exe/.out)	Combines object files, resolves external references, and creates the executable.
Loader	Executable file	Program in memory	Loads the executable into memory and starts program execution.



9.2 The UNIX Object File Format(Example)

The output of the assembler is structured in a highly standardized binary format. In standard UNIX-like environments, an object file consists of six distinct sections:

- **Object File Header:** Contains metadata describing the sizes and starting locations of the other sections of the file, allowing the linker and loader to parse it correctly.
- **Text Segment:** Contains the raw compiled binary machine instructions of the program.
- **Data Segment:** Contains static and global variables that are initialized during program compilation.
- **Relocation Information:** Lists all instructions and data words whose absolute memory addresses must be patched by the linker because they depend on external references or memory layout.
- **Symbol Table:** A critical table that maps all internal symbolic labels to their binary offset addresses, and catalogs external symbols that the program references but does not define.
- **Debugging Information:** Contains symbolic mapping data (such as variable names and source code line numbers) to enable software debuggers to map binary instructions back to the original source code during testing.

9.3 Static vs. Dynamic Linking

When combining multiple object files, the linker can follow one of two models:

- **Static Linking:** The linker copies all required library routines directly into the final executable file. While this makes the executable standalone and fast to load, it results in extremely large file sizes and wastes memory when multiple running programs use the same library.
- **Dynamic Linking (Dynamically Linked Libraries - DLLs):**

  Dynamic linking is the process of linking library functions at run time using a **tiny stub** and the operating system's **dynamic loader**, instead of embedding the library code into the executable.

  In **dynamic linking**, the library functions are **not copied** into the executable file. Instead,the executable contains a **tiny stub**, which is a small piece of code that acts as a placeholder for the library function.

  When the program is executed:

  1. The operating system's **dynamic loader** checks the required shared libraries (DLLs in Windows or `.so` files in Linux).
  2. It loads the required library into **shared memory** if it is not already loaded.
  3. The **tiny stub** obtains the actual memory address of the required library function.
  4. Control is transferred to the library function, and the program continues execution.

  In many systems, **lazy linking (lazy loading)** is used:

  - The library function is **not linked when the program starts**.
  - Instead, the function is linked **only when it is called for the first time**.

- The resolved address is then saved, so future calls go directly to the function without repeating the lookup.

What is a Tiny Stub?

A **tiny stub** is a **small placeholder program** inside the executable.

Its job is to:

- Identify the required library function.
- Request the operating system's dynamic loader to find and load the function if necessary.
- Obtain the function's memory address.
- Transfer control to the actual library function.

The stub itself **does not contain the library code**; it simply helps locate and call it.

Example

Suppose a program uses the `printf()` function.

With **static linking**:

```
Program.exe
├──── Main Program
└──── printf() code copied into executable
```

With **dynamic linking**:

```
Program.exe
├──── Main Program
└──── Tiny Stub for printf()
```

At runtime:

```
Program Starts
```

```
Tiny Stub Executes
```

```
Dynamic Loader Searches for C Library
```

```
Loads Library (if needed)
```

```
Finds printf() Address
```

```
Calls printf()
```

Advantages of Dynamic Linking

- **Smaller executable size** because library code is not copied.
- **Lower memory usage** since multiple programs can share the same library in memory.
- **Easy library updates** without recompiling the application.
- **Faster software maintenance**, as only the shared library needs to be updated.

Key Points for Examination

- **Dynamic linking** links library functions **at run time**.
- The executable contains a **tiny stub** instead of the complete library code.
- The **dynamic loader** loads the required library and resolves function addresses.
- **Lazy linking** resolves a function only when it is called for the first time.
- Dynamic linking saves **disk space**, **memory**, and simplifies **library updates**.

Advantages of Dynamic Linking

- **Smaller executable size** because library code is not copied.
- **Lower memory usage** since multiple programs can share the same library in memory.
- **Easy library updates** without recompiling the application.
- **Faster software maintenance**, as only the shared library needs to be updated.

10. CPU Performance Metrics and Evaluation

Evaluating the performance of computer systems requires precise mathematical metrics. The ultimate measure of performance is execution time — the actual time required to complete a specific task. To analyze the performance of a processor and identify the impact of various architectural choices, computer architects expand execution time into three fundamental variables.

10.1 The CPU Performance Equation

Execution time is represented as a product of three distinct, hardware and software-dependent variables:

The CPU Performance Equation

CPU Time = Instruction Count x Cycles Per Instruction (CPI) x Clock Cycle Time

Where:

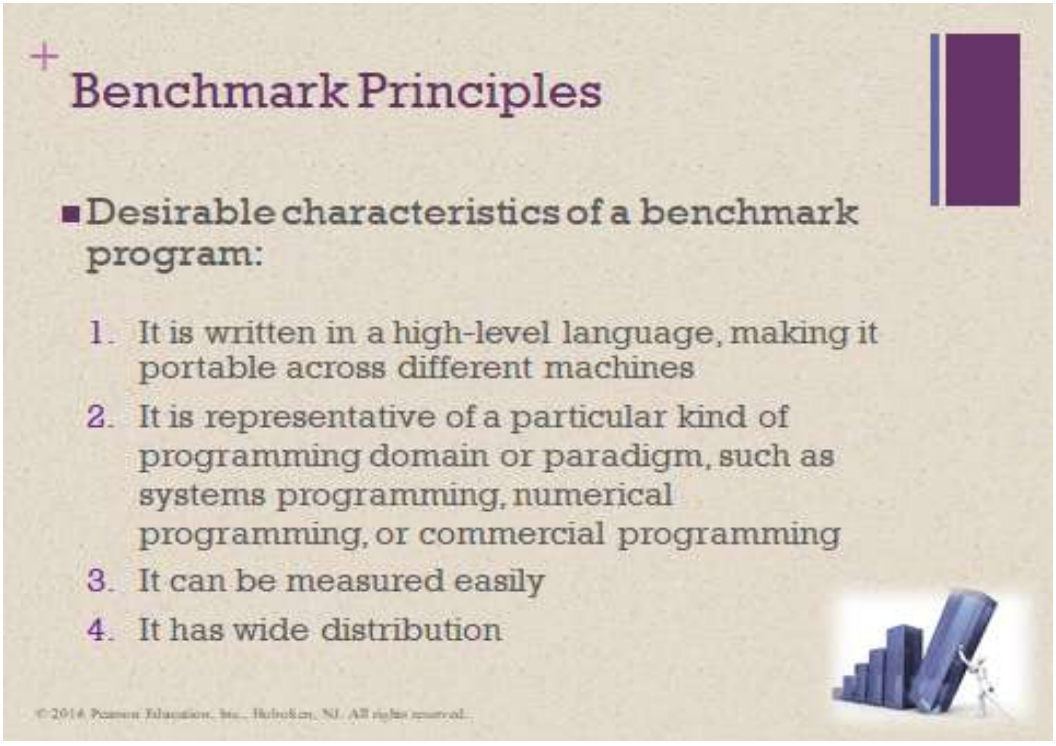
- **Instruction Count (IC)**: The total number of machine instructions executed to complete a program. (Determined by the compiler technology and the Instruction Set Architecture).
- **Cycles Per Instruction (CPI)**: The average number of clock cycles required to execute a single instruction. (Determined by the processor implementation and pipeline organization).

- **Clock Cycle Time**: The duration of a single clock cycle (inverse of Clock Frequency). (Determined by semiconductor fabrication technology and pipeline depth).

A change in one of these variables often affects the others. For example, a CISC processor reduces Instruction Count but increases CPI, whereas a RISC processor minimizes CPI at the cost of a higher Instruction Count.

10.2 Benchmarks and SPEC CPU Suites

Because raw hardware metrics like clock speed (GHz) or MIPS (Millions of Instructions Per Second) do not reliably predict real-world performance, the industry relies on standardized benchmarks. The Standard Performance Evaluation Corporation (SPEC) provides the gold standard for processor evaluation via suites of real-world programs modified to minimize I/O bias and facilitate portability. For example, SPEC CPU2017 consists of 10 integer benchmarks (CINT2017, e.g., compilers, compression algorithms) and 17 floating-point benchmarks (CFP2017, e.g., ray tracing, weather forecasting). Performance is reported as a ratio relative to a baseline reference computer.

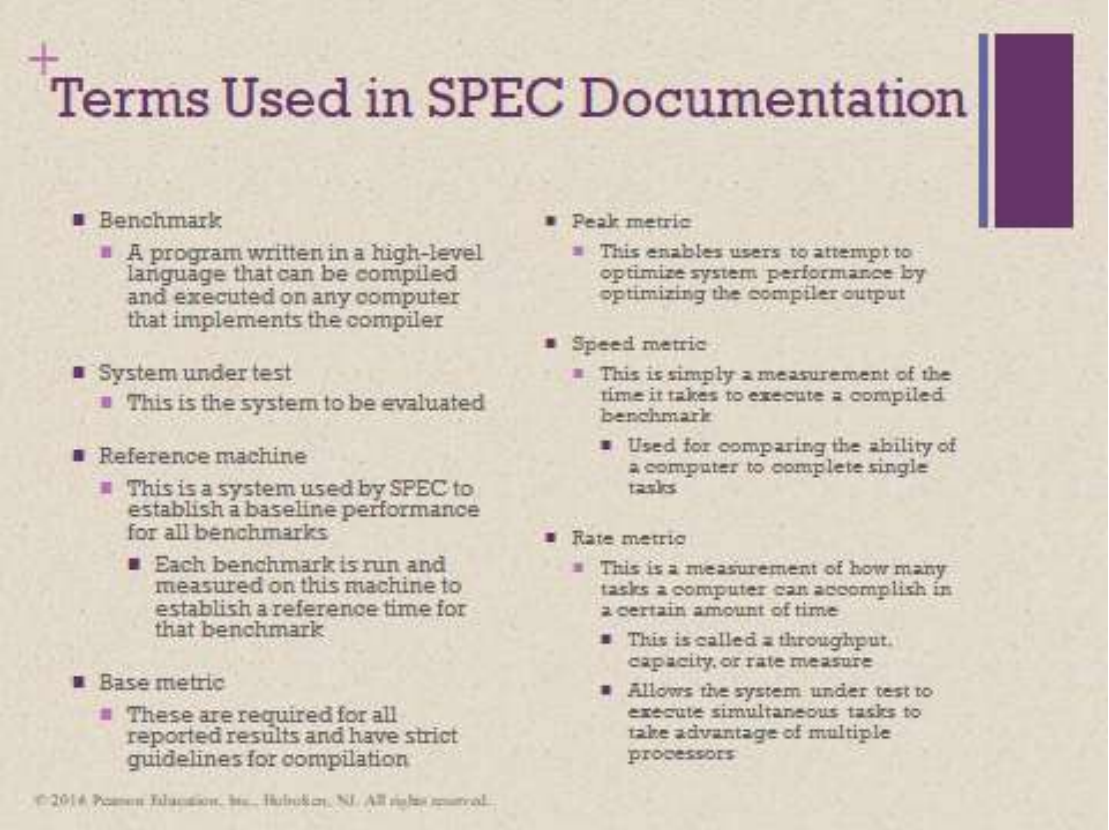


System Performance Evaluation Corporation (SPEC)

- Benchmark suite
  - A collection of programs, defined in a high-level language
  - Together attempt to provide a representative test of a computer in a particular application or system programming area
- SPEC
  - An industry consortium
  - Defines and maintains the best known collection of benchmark suites aimed at evaluating computer systems
  - Performance measurements are widely used for comparison and research purposes

SPEC CPU2006

- Best known SPEC benchmark suite
- Industry standard suite for processor intensive applications
- Appropriate for measuring performance for applications that spend most of their time doing computation rather than I/O
- Consists of 17 floating point programs written in C, C++, and Fortran and 12 integer programs written in C and C++
- Suite contains over 3 million lines of code
- Fifth generation of processor intensive suites from SPEC



10.3 Amdahl's Law

Amdahl's Law is a mathematical model that defines the law of diminishing returns when accelerating only a fraction of a computing task:

Amdahl's Law

$$\textbf{Speedup}_{\textbf{overall}} = \textbf{1} \,/\, [\ (1 - \text{Fraction}_{\text{enhanced}}) + (\text{Fraction}_{\text{enhanced}} \,/\, \text{Speedup}_{\text{enhanced}})\ ]$$

An important corollary of Amdahl's Law is that if an enhancement is only usable for a fraction of a task, the maximum overall speedup is strictly limited by the reciprocal of 1 minus that fraction (e.g., if you speed up an operation by 100x, but it is only used 40% of the time, the maximum overall speedup is less than 1.56).

11. Transition from Uniprocessors to Parallel Architectures

11.1 The Power Wall and Physical Limits of Uniprocessor Scaling

From 1978 to 2003, microprocessor performance increased at a dramatic rate of 52% per year. However, around 2003, this growth rate hit a physical barrier known as the Power Wall. For decades, chip designers relied on Dennard scaling: as transistors shrunk, they became faster and consumed less

power, keeping the power density (watts per square millimeter of silicon) constant. This allowed designers to continuously scale up clock frequencies.

Around 2003, Dennard scaling ended. Below 90 nm, leakage currents and static power dissipation skyrocketed, leading to thermal budgets that could no longer be cooled cheaply. This power dissipation limit stopped frequency scaling at around 3 to 4 GHz. This thermal limit also created the phenomenon of Dark Silicon: parts of the chip must be left unpowered ('dark') at any given moment because turning on all transistors simultaneously would melt the silicon. Faced with these physical limits, the entire computer architecture industry pivoted in 2004 from building faster, single-core uniprocessors to placing multiple processors (cores) on a single chip, giving birth to the multicore/multiprocessor era.

11.2 Classes of Parallelism and Parallel Architectures

To make programs run faster in the multicore era, software must exploit parallelism. There are two primary types of parallelism in applications:

1. **Data-Level Parallelism (DLP)** Arises because there are many data items that can be operated on at the same time.
2. **Task-Level Parallelism (TLP)** Arises because independent tasks of work are created that can execute in parallel.

Computer hardware, in turn, exploits these application properties in four major ways:

- **Instruction-Level Parallelism (ILP):** Exploits data-level parallelism at modest levels using hardware techniques like pipelining and speculative execution.
- **Vector Architectures / GPUs:** Exploit data-level parallelism by applying a single instruction to a collection of data in parallel (SIMD).
- **Thread-Level Parallelism:** Exploits either DLP or TLP in a tightly coupled hardware model allowing interactions between parallel threads.
- **Request-Level Parallelism:** Exploits parallelism among completely decoupled tasks specified by the operating system.

11.3 Flynn's Taxonomy of Computer Systems

Introduced by Michael Flynn in 1966, this taxonomy classifies all computer systems into four major categories based on the number of concurrent instruction and data streams:

1. **Single Instruction Stream, Single Data Stream (SISD)** The traditional sequential uniprocessor. It executes a single instruction stream on a single data stream, although it can exploit ILP internally (e.g., pipelining, speculative execution).
2. **Single Instruction Stream, Multiple Data Streams (SIMD)** A single instruction is fetched and dispatched to multiple processing lanes, each operating on different data. Highly efficient for regular data-parallel workloads (e.g., vector processors, GPUs, multimedia extensions like MMX/SSE).
3. **Multiple Instruction Streams, Single Data Stream (MISD)** Multiple instructions operate on the same data. No commercial systems have been built using this taxonomy, although it serves as a theoretical category (e.g., fault-tolerant systems).

4. **Multiple Instruction Streams, Multiple Data Streams (MIMD)** Each processor fetches its own instructions and operates on its own data stream. This is the model of all modern multicore systems and parallel supercomputers.

- Tightly Coupled MIMD: Processors share a unified address space (shared-memory multiprocessors or SMPs) communicating via memory.

- Loosely Coupled MIMD: Processors have private address spaces (clusters, warehouse-scale computers) and communicate via message-passing protocols over networks.

12. References and Recommended Readings

The material in these study notes is fully grounded in the course curriculum and aligns directly with the standard reference textbooks:

1. **Stallings, W. (2016).** Computer Organization and Architecture: Designing for Performance (10th Edition). Pearson Education.

2. **Patterson, D. A., & Hennessy, J. L. (2020).** Computer Organization and Design: The Hardware/Software Interface (RISC-V Edition). Morgan Kaufmann / Elsevier.

3. **Hennessy, J. L., & Patterson, D. A. (2019).** Computer Architecture: A Quantitative Approach (6th Edition). Morgan Kaufmann / Elsevier Publishers.

4. **Hayes, J. P. (2017).** Computer Organization and Architecture (3rd Edition). Tata McGraw Hill.