

I Introduction

1.1 CPU Performance and Clocking

Digital hardware operations are synchronized by a constant-rate clock, which provides the fundamental temporal reference for state transitions within a processor. The clock period defines the duration of a single clock cycle (measured in seconds), while its reciprocal, the clock frequency (or rate), defines the number of cycles per second (measured in Hertz). To accurately evaluate the performance of a computer system, we must isolate the components that contribute to the total execution time. The most direct measure of performance is the CPU Time, which is the actual time the processor spends executing a specific program. This can be expressed as:

$$\text{CPU Time} = \frac{\text{CPU Clock Cycles for a Program}}{\text{Clock Rate}} \quad (1)$$

However, this basic formula does not account for the efficiency of the Instruction Set Architecture (ISA) or the underlying microarchitecture. To provide deeper insight, we introduce the Instruction Count (IC) - the total number of instructions executed in a program - and the Cycles Per Instruction (CPI), which represents the average number of clock cycles required to execute a single instruction:

$$\text{CPI} = \frac{\text{CPU Clock Cycles for a Program}}{\text{Instruction Count}} \quad (2)$$

By reorganizing these relationships, we derive the CPU Performance Equation (often called the "Iron Law" of performance). This equation demonstrates that execution time is the product of three distinct factors:

$$\text{CPU Time} = \underbrace{\text{Instruction Count}}_{\text{ISA and Compiler}} \times \underbrace{\text{Cycles Per Instruction}}_{\text{Hardware and Optimization}} \times \underbrace{\text{Clock Cycle Time}}_{\text{Microarchitecture and ISA}} \quad (3)$$

Each term in the performance equation is influenced by different aspects of the system design:

- *Instruction Count* is determined by the complexity of the program, the efficiency of the compiler, and the Instruction Set Architecture itself.
- *Cycles Per Instruction (CPI)* depends on the processor's microarchitecture (e.g., pipeline depth, superscalar execution, and cache hits/misses).
- *Clock Cycle Time* is a function of the hardware technology, the circuit's critical path, and the physical constraints of the manufacturing process.

1.2 Instruction Set Architecture

An Instruction Set Architecture (ISA) serves as the abstract interface between the hardware and the lowest-level software. While different computing architectures utilize distinct instruction sets, they often share fundamental operational principles. Early computer systems utilized simplified

instruction sets to facilitate easier implementation, a design philosophy that persists in many modern architectures, including the MIPS (Microprocessor without Interlocked Pipelined Stages) instruction set. MIPS is particularly prominent in the embedded systems market, powering a vast array of consumer electronics, networking infrastructure, and peripheral devices like printers and cameras.

Design Principles and Register Organization. A core tenet of MIPS design is that simplicity favors regularity. By maintaining a consistent instruction format - such as ensuring all arithmetic operations utilize two source operands and one destination - the hardware implementation is simplified, which in turn enables higher performance at a lower silicon cost. In MIPS, arithmetic instructions operate exclusively on registers rather than directly on memory. The architecture features a 32×32 -bit register file, which stores the most frequently accessed data. Each 32-bit data unit is referred to as a word. Within this register file, specific naming conventions help the assembler and compiler manage data:

- `$t0 - $t9` : Registers used for temporary values that do not need to be preserved across procedure calls.
- `$s0 - $s7` : Registers used for "saved" variables that must be preserved.
- `$0` : A hardwired constant value of 0 that cannot be overwritten, facilitating common operations like data movement or comparisons.

While registers provide high-speed access, the main memory is required for large or complex data structures, such as arrays, dynamic data, and objects. To perform arithmetic on memory-resident data, the processor must execute "load" instructions to move values into registers and "store" instructions to write results back to memory. MIPS employs byte addressing, meaning every unique address identifies a single 8-bit byte. However, because the architecture operates on 32-bit words, word alignment is mandatory: the starting address of any word must be a multiple of four. Furthermore, MIPS traditionally follows a Big-Endian byte ordering, where the most significant byte (MSB) is stored at the lowest memory address of the word.

Because register access is significantly faster than memory access, a primary goal for compilers is to maximize register utilization. When the number of variables exceeds the available registers, the compiler must "spill" less frequently used data to memory, a process that increases instruction count and latency. To further enhance efficiency, MIPS adheres to the principle of making the common case fast. Since small constants are frequently used in code (e.g., incrementing a loop counter), MIPS supports immediate operands. These allow constant values to be embedded directly within the instruction itself, eliminating the need to execute a separate load instruction from memory.

Representing Instructions. In the MIPS architecture, instructions are encoded as 32-bit binary words, commonly referred to as machine code. To maintain hardware efficiency and regularity, MIPS utilizes a few fixed-length instruction formats. Each format partitions the 32-bit word into specific fields that define the operation (opcode), the registers involved, and any immediate values or offsets.

MIPS R-Format / The R-format is used primarily for computational instructions where all operands reside in registers. The 32 bits are divided into six distinct fields:

- **op** (6 bits): The operation code (opcode) which defines the general type of instruction. For R-format instructions, this value is typically zero.
- **rs** (5 bits): The first source register number.
- **rt** (5 bits): The second source register number.
- **shamt** (5 bits): The shift amount, used specifically for shift instructions (e.g., Logical Shift Left). For standard arithmetic, this field is set to zero.
- **funct** (6 bits): The function code. Since the opcode for R-format instructions is zero, this field is used to select the specific arithmetic or logical operation (e.g., addition vs. subtraction).

MIPS I-Format / The I-format is designed for instructions that require an immediate constant or a memory address offset. It consolidates the final three fields of the R-format into a single 16-bit field, organized as follows:

- **op** (6 bits): The opcode, which uniquely identifies the instruction type (e.g., `addi`, `lw`, `sw`).
- **rs** (6 bits): The source register (used as a base address for memory operations or a source operand for arithmetic).
- **rt** (6 bits): The destination register (for loads and immediate arithmetic) or the source register (for store operations).
- **Address/Immediate** (16 bits): A constant value ranging from -2^{15} to $2^{15} - 1$. In memory-access instructions, this acts as a signed offset added to the base address in **rs**.

Instruction	Type	Operation	Example
<i>Arithmetic & Logical</i>			
<code>add</code>	R	$\$rd = \$rs + \$rt$	<code>add \$t0, \$s1, \$s2</code>
<code>sub</code>	R	$\$rd = \$rs - \$rt$	<code>sub \$t0, \$s1, \$s2</code>
<code>addi</code>	I	$\$rt = \$rs + imm$	<code>addi \$t0, \$s1, 10</code>
<code>and</code>	R	$\$rd = \$rs \& \$rt$	<code>and \$t0, \$t1, \$t2</code>
<code>sll</code>	R	$\$rd = \$rt \ll shamt$	<code>sll \$t0, \$s1, 2</code>
<i>Data Transfer</i>			
<code>lw</code>	I	$\$rt = \text{Mem}[\$rs + off]$	<code>lw \$t0, 8(\$s1)</code>
<code>sw</code>	I	$\text{Mem}[\$rs + off] = \rt	<code>sw \$t0, 4(\$s1)</code>
<code>lui</code>	I	$\$rt = imm \ll 16$	<code>lui \$t0, 0x1234</code>
<i>Branch & Jump</i>			
<code>beq</code>	I	<code>if (\$rs == \$rt) branch</code>	<code>beq \$t0, \$t1, Label</code>
<code>slt</code>	R	<code>if (\$rs < \$rt) \$rd = 1</code>	<code>slt \$t0, \$s1, \$s2</code>
<code>j</code>	J	<code>jump to addr</code>	<code>j Label</code>
<code>jr</code>	J	<code>jump to addr in register</code>	<code>jr \$ra</code>

Table 4 / Common MIPS instructions categorized by function—arithmetic/logical, data transfer, and control flow—with operation semantics and illustrative examples.

The fundamental breakthrough of modern computing is the stored-program concept, which treats instructions and data as logically equivalent. Both are represented as binary sequences and reside in the same memory space. This uniformity allows programs to operate on other programs, as seen in compilers, linkers, and loaders, enabling a sophisticated software ecosystem. Furthermore, the standardization of these instruction sets ensures binary compatibility, allowing compiled software to execute reliably across different hardware implementations of the same ISA.

Management of memory during function execution. In MIPS assembly and similar RISC architectures, the management of memory during function execution relies on a set of specialized registers. These registers coordinate the lifecycle of data and control flow, ensuring that variables are accessible and that the processor can return to the correct execution point after a function completes.

- *Global Pointer:* The Global Pointer (`$gp`) is designed to streamline access to static data, which consists of variables whose size and location are known at compiler time (e.g., global variables or constants). In memory, these variables are stored in the "Static" or "Data" segment. Without a global pointer, accessing these variables would require a two-step process: first loading the upper 16 bits of the address and then the lower 16 bits. By convention, `$gp` is initialized to point to the middle of a 64 KB block within this static segment. This allows the processor to access any variable in that range using a single instruction with a 16-bit signed offset relative to `$gp`. This optimizes performance by reducing the instruction count for global data manipulation.
- *Stack Pointer:* The Stack Pointer (`$sp`) manages the stack segment, a region of memory that grows and shrinks dynamically as functions are called and return. In MIPS, the stack grows "downwards" from higher memory addresses to lower ones. The primary role of `$sp` is to track the current top of the stack. When a function requires space for local variables or needs to preserve register values, it "allocates" space by incrementing `$sp` back to its original position. This area of memory is strictly LIFO, making it ideal for managing the nested nature of function calls.
- *Frame Pointer:* While `$sp` moves during the execution of a function (for instance, when pushing temporary arguments onto the stack), the Frame Pointer (`$fp`) remains stationary, pointing to the base of the current activation record (or stack frame). The `$fp` provides a stable reference point for accessing local variables and parameters. Because the distance between `$fp` and a specific local variable remains constant throughout the function's execution, the compiler can generate fixed offsets to access that data. While some compilers can optimize code to use only `$sp`, the use of `$fp` is critical in functions where the stack size may change dynamically, such as when using `malloc()` in C.
- *Return Address:* The Return Address register (`$ra`) is fundamental to the "Link" part of the `jal` instruction. When a function is called via `jal`, the processor automatically stores the address of the next instruction into `$ra`. The effect on memory depends on the type of function:
 - *Leaf Procedures:* If a function does not call any other functions, it is a "leaf". It can simply perform its task and return using `jr $ra` without ever touching the memory stack for control purposes.

- *Non-Leaf Procedures:* If a function calls another function, the new `jal` would overwrite the current value in `$ra`. Therefore, the calling function must first save the current `$ra` onto the stack (using `$sp`) during its prologue and restore it during its epilogue.

1.3 The Processor

The fundamental operation of a processor involves a continuous cycle of fetching, decoding, and executing instructions. This process begins with the Program Counter (PC), which holds the address of the next instruction to be retrieved from instruction memory. Once the instruction is fetched, its bit fields specify register numbers within the register file, enabling the processor to read the required operands. Depending on the instruction class, the Arithmetic Logic Unit (ALU) is then employed to calculate a numerical result, compute a memory address for load/store operations, or determine a branch target address. In standard sequential execution, the PC is updated to $PC + 4$ to point to the next word-aligned instruction.

At the hardware level, all information is encoded in binary, where low voltage levels represent a logical 0 and high voltage levels represent a logical 1. These signals are transmitted via single wires for individual bits or through multi-wire buses for multi-bit data. The processor's internal circuitry is composed of two functional building blocks:

- *Combinational Elements:* These elements (such as the ALU, multiplexers, and adders) transform data such that the output is a pure function of the current inputs. They do not retain memory of previous states.
- *State Elements:* These components (such as registers and memory) are capable of storing information. They define the "state" of the computer and require a synchronization signal to manage data updates.

State elements rely on a clock signal to determine when to update their stored values. Most modern processors utilize edge-triggered clocking, meaning the state is updated only when the clock signal transitions (typically from a low to a high voltage, or the "rising edge"). For more precise control, certain registers include a write control input; the stored value is only updated on the clock edge if this signal is asserted. This is critical for maintaining data stability when a value needs to be preserved over multiple cycles. The interaction between these elements defines the processor's timing requirements. Combinational logic performs its transformations during the interval between clock edges. Signals flow from one state element, through the combinational logic, and arrive at the input of the next state element. Consequently, the clock period, and by extension the processor's frequency, is dictated by the longest propagation delay through the combinational logic. The clock cycle must be sufficiently long to allow all signals to stabilize before the next active edge triggers the next state update.

Functional Datapath and Instruction Execution. The datapath is the hardware representation of the steps required to execute an instruction. In a basic MIPS implementation, the flow of data is determined by the instruction type, which dictates how the Arithmetic Logic Unit (ALU), the register file, and memory interact.

- *R-Format Instructions:* R-format instructions follow a strictly register-based flow: they read two operands from the register file, perform a specified arithmetic or logical operation within the ALU, and write the resulting value back into a destination register.

- *I-Format Instructions:* Load and store instructions, however, must interface with data memory. These instructions use the I-format, which includes a 16-bit immediate field representing a displacement. To compute the effective memory address, the processor adds this displacement to a base address stored in a register. Because the internal datapath is 32 bits wide, the 16-bit offset must first undergo sign-extension. This process replicates the most significant bit of the 16-bit constant across the upper 16 bits of the 32-bit bus, ensuring the numerical value remains the same, whether positive or negative, when converted to a larger format. Once the address is calculated, a load operation reads data from memory into a register, while a store operation writes a register's value into memory.
- *Branching Instructions:* Branch instructions, such as `beq`, determine the next instruction's address based on a comparison. The processor reads two register operands and utilizes the ALU to subtract one from the other. If the "Zero" output of the ALU is asserted, the condition is met. Calculating the branch target address requires a specific multistep transformation of the 16-bit displacement field:
 - (1) The 16-bit displacement is sign-extended to 32 bits to allow for both forward and backward branches.
 - (2) Since all MIPS instructions are 32 bits long and word-aligned, the immediate field stores the jump distance in words rather than bytes to maximize the branching range. Shifting left by two bits effectively multiplies the value by four, converting the word offset into a byte offset.
 - (3) This byte offset is then added to $PC + 4$. The use of $PC + 4$ is a result of the hardware naturally incrementing the PC during the initial fetch stage before the branch offset is calculated.

In a "first-cut" or single-cycle datapath, the hardware is designed to complete one full instruction within a single clock cycle. This architectural choice introduces specific hardware requirements. Because a single functional element cannot perform two different tasks simultaneously within one cycle, the system requires separate instruction and data memories. This prevents a "structural hazard" where the processor would otherwise need to fetch a new instruction and access data memory at the same time. Furthermore, because different instructions require data from different sources, the datapath utilizes multiplexers at critical junctions. These Muxes act as data selectors, directed by the Control Unit to choose the appropriate path. For example, a Mux is used to decide whether the ALU's second operand comes from the register file (for R-format) or from the sign-extended immediate field (for load/store). Another Mux determines whether the value written to the register file originates from the ALU or from data memory.

Memory Hierarchy

In an ideal world, a computer's memory would be instantaneous, infinitely large, and incredibly cheap. In reality, we face a trade-off: fast memory, like Static RAM (SRAM), is extremely expensive and small, while large-capacity storage, like Magnetic Disks, is cheap but orders of magnitude slower. This gap is the central challenge of memory system design. The solution is not a single perfect memory but a Memory Hierarchy.

Principle of Locality. Programs do not access their entire address space randomly. Instead, they tend to focus on a small portion of it at any given time. This behavior is broken into two types:

- *Temporal Locality*: If an item is accessed, it is likely to be accessed again soon (e.g., instructions inside a loop or a variable used repeatedly).
- *Spatial Locality*: If an item is accessed, items with nearby addresses are likely to be accessed soon (e.g., in sequential instruction fetching or when iterating through data in an array).

Hierarchical Structure. The memory hierarchy takes advantage of locality by creating a "pyramid" of memory levels. The levels closer to the processor are smaller, faster, and more expensive, while the levels further away are larger, slower, and cheaper:

- (1) *L1 Cache (SRAM)*: The level closest to the CPU. Holds the most recently used data and instructions.
- (2) *L2 Cache (DRAM)*: A larger, slightly slower cache that services misses from the L1 cache.
- (3) *Main Memory (DRAM)*: The main working memory of the system.
- (4) *Storage (Disk/Flash)*: The largest, slowest level that holds everything permanently.

When the CPU needs data, it first checks the L1 cache. If the data is there, it's a Hit. If not, it's a Miss, and the CPU must request the data from the next level down. If the L2 misses, it requests from Main Memory, and so on. The time it takes to fetch data from a lower level and bring it to the upper level is called the Miss Penalty. The goal is to maximize the Hit Ratio (hits/accesses) and minimize the Miss Ratio (misses/accesses).

2.1 Memory Technologies

DRAM. DRAM is the technology used for main memory. Its key characteristics include:

- *Technology*: Stored data as a charge in a tiny capacitor. Because this charge leaks, DRAM must be periodically refreshed (read and written back) to prevent data loss.
- *Organization*: DRAM is organized as a rectangular array of bits. Access involves activating an entire row.
- *Performance*: To improve bandwidth, modern DRAM uses several techniques:

- Synchronous DRAM (SDRAM): Uses a clock to allow for "burst mode", which supplied consecutive words from a row without needing a new address for each one.
- Double Data Rate (DDR): Transfers data on both the rising and falling edges of the clock, effectively doubling the transfer rate.
- DRAM Banking: Allows multiple DRAM chips to be accessed simultaneously to improve bandwidth.

Storage: Flash and Disk. These technologies form the nonvolatile base of the hierarchy.

- *Flash Storage:* A nonvolatile semiconductor memory. It is significantly faster than disk (100x - 1000x) and more robust, but more expensive. NAND flash is the denser, cheaper type used for USB keys and media storage. Its main drawback is that flash bits wear out after thousands of accesses, requiring "wear leveling" techniques.
- *Magnetic Disk:* A nonvolatile, rotating magnetic storage. Accessing a sector of data involves several time-consuming steps: queuing delay, seek time (moving the heads), rotational latency (waiting for the data to rotate under the head), and finally the data transfer.

2.2 Cache Memory

The cache is the part of the memory hierarchy closest to the CPU. When the CPU requests an address, the cache must quickly answer two questions: Is the data already present in the cache? If so, where is it? To manage this, the cache is divided into blocks (also called lines), which are the unit of data transfer. A memory address is subdivided to determine where it fits in the cache:

- *Block Offset:* The low-order bits that identify a specific word or byte within a block.
- *Index:* The middle bits that determine which cache set or line the block maps to.
- *Tag:* The high-order bits that are stored in the cache to identify which specific memory block is currently occupying that line.

A Valid Bit is also stored with each cache line. If this bit is 0, the data in that line is considered not present or invalid.

Cache Mapping. The "where to look" question is answered by the cache's associativity, which defines how flexibly memory blocks can be placed in the cache.

- (1) *Direct Mapped:* The simplest and cheapest policy. A memory block can go in only one specific location in the cache, determined by its index. This is very fast to check (only one location), but can cause conflict misses. If the program frequently alternates between two blocks that map to the same index, they will constantly evict each other, leading to misses even if the cache is mostly empty.
- (2) *N-Way Set Associative:* A compromise. The cache is divided into sets, each containing n blocks (or "ways"). A memory block maps to a single set (based on its index), but it can be placed in any of the n locations within that set. This dramatically reduces conflict misses compared to a direct-mapped cache. A 2-way or 4-way associative cache offers most of the benefits. However, it is more complex, as it must search all n entries in the set at once.

- (3) *Fully Associative*: The most flexible policy. A memory block can be placed in any available location in the entire cache. This eliminates conflict misses entirely, but it is extremely expensive. It requires a comparator for every single entry in the cache, making it practical only for very small caches.

For associative caches, a Replacement Policy is also needed when a set is full. Least-Recently Used (LRU) is a common strategy: the cache evicts the block that has been unused for the longest time.

An Example / Given 64 blocks of cache size and a block size of 16 bytes/block, to what block number does memory address 1200 map?

- (1) *Find the Block Address*: First, determine which block the address is in.

$$\text{Block Address} = \left\lfloor \frac{\text{Address}}{\text{Block Size}} \right\rfloor = \left\lfloor \frac{1200}{16} \right\rfloor = 75 \quad (5)$$

- (2) *Find the Cache Index (Block Number)*: Next, find where that block maps in the cache using the modulo operator.

$$\text{Block Number} = (\text{Block Address}) \bmod (\# \text{ Blocks in Cache}) = 75 \bmod 64 = 11 \quad (6)$$

Therefore, address 1200 maps to block 11 of the cache.

Cache Writes. Writing to memory introduces complexity because the cache and main memory can become inconsistent.

On a Write Hit

- *Write-Through*: The data is written to both the cache and main memory at the same time. This is simple but slow, as it makes every write as slow as a memory access. To mitigate this, a write buffer is often used to hold the outgoing data, allowing the CPU to continue immediately without waiting.
- *Write-Back*: The data is written only to the cache block. A dirty bit is set for that block to mark it as modified. The block is only written back to main memory later when it is replaced (and only if it is "dirty"). This is much faster for frequent writes but is more complex to implement.

On a Write Miss

- *Write Allocate*: The block is first fetched from main memory into the cache, and then the write is performed. This is the standard policy for write-back caches.
- *Write Around*: The write bypasses the cache and goes directly to main memory. This is often used with write-through policies, especially for initialization code that writes an entire block without reading it first.

Measuring Cache Performance. The ultimate goal of the memory hierarchy is to improve CPU performance. The time spent stalling for memory is a major component of this.

$$\begin{aligned} \text{CPU Time} &= \text{Program Execution Cycles} + \text{Memory Stall Cycles} \\ \text{Memory Stall Cycles} &= \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss Penalty} \end{aligned} \tag{7}$$

A key metric for evaluating a memory system is the Average Memory Access Time (AMAT):

$$\text{AMAT} = \text{Hit Time} + (\text{Miss Rate} \times \text{Miss Penalty}) \tag{8}$$

As CPUs become faster, the **Miss Penalty** (measured in lost CPU cycles) gets larger. This means that cache performance and a low miss rate become more critical to overall system performance, not less. To bridge the growing gap between fast CPUs and slow DRAM, nearly all modern systems use Multilevel Caches. By splitting the cache, designers can optimize the L1 for speed and the L2 for capacity, achieving better overall performance than a single, large cache.

Peripherals

3.1 Embedded Systems Communication

Embedded systems rely on hierarchical communication architectures that scale from tightly integrated on-die circuitry to wide-area networked interactions. These communications are conventionally categorized into four interdependent domains: On-Chip, On-board, In-System, and External, each defined by its physical scope, latency requirements, and protocol characteristics. Together, they enable seamless coordination between computation, hardware, subsystem, and end users.

- *On-Chip*: This innermost layer operates entirely within a single integrated circuit, where speed is paramount and physical area, while constrained, remains secondary to latency and bandwidth. Communication occurs between the CPU, memory, and integrated peripherals (e.g., timers, ADCs, DMA controllers) via high-speed parallel buses, commonly transferring data in 8-, 16-, or 32-bit chunks per clock cycle. Parallelism maximizes throughput with minimal protocol overhead, enabling cycle-accurate access to critical resources. Because signals remain confined within the silicon die, noise and skew are tightly controlled—allowing clock frequencies to reach hundreds of MHz or even GHz. This layer forms the real-time foundation: every instruction fetch, interrupt response, or peripheral register access depends on its efficiency.
- *On-board*: When communication extends beyond the chip-across discrete components on a single PCB-design priorities shift. Each additional signal requires a dedicated pin on the IC package and a trace on the board, increasing cost, size, and weight due to larger packages and more complex routing. As a result, serial buses dominate: they transmit data one bit at a time, minimizing pin count and simplifying PCB layout. Common protocols include:
 - SPI for high throughput, full-duplex, point-to-point links (e.g., to an SD card or display driver),
 - I²C for multi-master, moderate speed, lower-bandwidth buses (e.g., environmental sensors),
 - UART for asynchronous device-to-device links (e.g., GPS modules or debug consoles),
 - I²S for digital audio streaming.

If bandwidth becomes limiting, designers may widen the bus (e.g., dual- or quad-SPI), increase clock rates, or adopt embedded SerDes (serializer/deserializer) links—though the latter introduces greater complexity. Signal integrity (e.g., impedance matching, crosstalk mitigation) becomes critical, as trace lengths and board stackup now influence timing margins.

- *In-System*: This tier coordinates multiple embedded nodes—such as Electronic Control Units (ECUs) in an automobile, subsystems in industrial machinery, or distributed sensors in a smart building. Here, determinism, fault tolerance, and electromagnetic compatibility (EMC) outweigh raw speed. Protocols like CAN, LIN, and FlexRay are purpose-built for harsh, electrically noisy environments. CAN, for instance, uses differential signaling

and built-in error detection with acknowledgment and retransmission to ensure message integrity — critical for safety-critical functions like braking or steering. Bus arbitration, time-triggered scheduling (e.g., FlexRay), and redundancy further enhance robustness. Cabling and connectors add cost, weight, and failure points, so minimizing conductor count remains important — even as systems grow more interconnected. Modern platforms increasingly supplement classical buses with Ethernet (e.g., 100BASE-T1, TSN) to support high-bandwidth domains (e.g., cameras, LiDAR), while preserving legacy interfaces for low-complexity nodes.

- *External:* The outermost domain connects the embedded system to external entities—users, cloud services, or other physical systems (e.g., phone — thermostat, vehicle — traffic infrastructure). Communication now traverses uncontrolled environments: longer cables, connectors, or wireless channels introduce significant noise vulnerability, latency, and security risks. As a result, robust error control is mandatory: end-to-end checksums, acknowledgments (ACK/NACK), retransmission (e.g., TCP), and forward error correction (FEC) are common. Solutions diverge based on application needs:
 - Wired, high-performance: USB 2.0/3.0, Ethernet (10/100/1000BASE-T), or CAN FD—where bandwidth justifies cables and shielding. Wider buses or higher clock speeds (e.g., SuperSpeed USB at 5 Gbps) compensate for serial bit-by-bit transmission.
 - Wireless, portable: Wi-Fi (802.11 a/b/g/n/ac/ax), LTE/5G, or IEEE 802.15.4 (Zigbee/Thread)—enabling mobility and flexible deployment, albeit with trade-offs in power, range, and interference resilience.

While this layer often operates at human timescales (hundreds of milliseconds to seconds), it depends entirely on the underlying tiers: commands traverse the stack downward, while status and telemetry flow upward, making end-to-end latency and data consistency system-wide concerns.

These four domains form a cohesive, nested hierarchy: each builds on the services and abstractions of the one below it. Communication is inherently bidirectional: control and configuration descend, while sensing and status ascend, and system performance hinges on optimizing transitions between layers. Modern embedded design thus requires co-engineering of hardware, firmware, and protocols across all tiers to achieve responsiveness, reliability, and scalability in real-world applications.

Bus. In embedded systems, a bus is a shared communication infrastructure that enables data, address, and control signals to be exchanged among multiple system components (e.g., processor, memory, peripherals). Unlike point-to-point links, buses allow multiple devices to connect to a common pathway, reducing wiring complexity and cost, but introducing the need for arbitration and protocol coordination. A bus implementation involves two interdependent layers:

- *Hardware Infrastructure* defines the physical medium: conductive traces on a PCB, cables (e.g., ribbon or shielded twisted pair), connectors, and transceivers. For instance, while desktop PCI uses edge connectors and backplanes, embedded systems favor on-board traces (e.g., SPI lines routed between MCU and sensors) or compact connectors (e.g., CAN on a DB9 or OBD-II port).
- *Software Infrastructure* governs how communication occurs: addressing, data framing, error handling, clocking, and arbitration. The PCI bus protocol, for example, specifies command

phases, burst transfers, and retry mechanisms — similar concepts appear in embedded protocols like CAN (with identifier-based arbitration) or I²C (with START/STOP conditions and ACK/NACK).

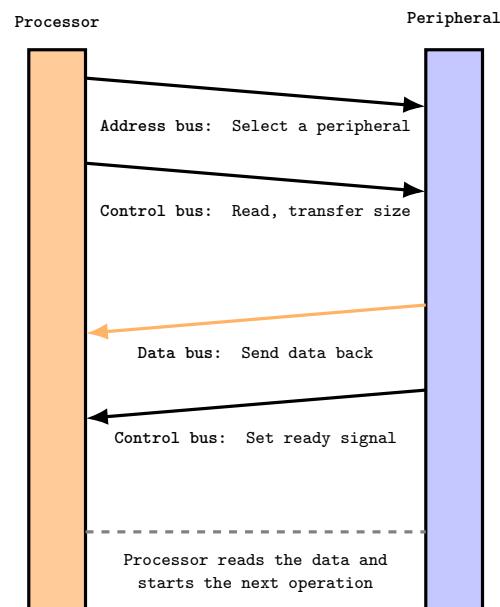
Embedded bus architecture commonly involve:

- *Master*: The device (typically the processor or DMA engine) that initiates and controls transactions by issuing addresses and control signals.
- *Slave*: A target device (e.g., memory chip, sensor, or peripheral register) that responds to master requests. Multiple slaves coexist on the same bus.
- *Address Decoder*: Logic (often built into the slave or a dedicated chip) that monitors the address bus and activates the correct slave when its address range is selected.
- *Multiplexer*: Used in some designs to route data or address lines between multiple potential sources or destinations — e.g., to share a data bus among several peripherals under master control.

A typical synchronous parallel bus — though increasingly rare in modern low-pin-count embedded designs — comprises three fundamental signal groups:

- (1) *Data Bus*: A bidirectional set of lines (e.g., 8, 16, or 32-bit wide) that carries payload information between master and slave.
- (2) *Address Bus*: A unidirectional set of lines specifying which slave (or which register within a slave) is being accessed. The width determines the addressable space (e.g., a 16-bit address bus can select up to 64 KB of memory-mapped I/O).
- (3) *Control Bus*: Timing and command signals that orchestrate the transaction, such as Read/Write to indicate data direction, Chip Select or Slave Select, Clock, and Ready or Wait for flow control.

An example. The communication cycle begins when the processor places the address of a target peripheral — or a specific register within it — onto the address bus. At the same time, it drives the control bus with signals indicating the operation type (e.g., read), data width, and timing parameters. The master then enters a wait state, allowing the peripheral time to prepare the requested data. Once ready, the slave places the data onto the data bus and asserts a ready (or acknowledge) signal on the control bus to confirm completion. Upon detecting this handshake, the master latches the data and concludes the transaction, immediately advancing to the next stage — whether initiating another transfer, processing the received value, or continuing program execution.



ARM AMBA System Bus. The Advanced Microcontroller Bus Architecture (AMBA) is an open-standard on-chip interconnect specification developed by Arm to define communication protocols between functional blocks (IP cores) in system-on-chip (SoC) designs. By providing a consistent, modular interface framework, AMBA significantly enhances IP reuse, accelerates integration, and supports right-first-time development — especially critical for complex multiprocessor systems incorporating numerous processors, memory controllers, and peripherals. As a cornerstone of modern SoC design, AMBA is extensively deployed in power and performance-sensitive applications, including smartphones, tablets, and embedded devices.

Among the AMBA protocol family, the Advanced High-performance Bus (AHB) targets high-speed, synthesizable system designs. AHB supports multiple bus masters, pipelined transfers, and high-bandwidth operation, making it suitable for connecting high-throughput components such as CPUs, DMA controllers, and high-speed memories. A streamlined variant, AHB-Lite, simplifies the full AHB specification — most notably by assuming a single bus master — thereby reducing logic complexity and verification effort. In an AHB-Lite transaction, the master initiates read or write operations by driving address, control, and data (for writes) signals onto the bus. Slaves respond with data (for reads), along with handshake signals: `HREADY` (indicating transfer completion) and `HRESP` (signaling success or error conditions). Slave selection is managed by the address decoder, which asserts the corresponding `HSELx` signal for the targeted slave during each access, ensuring only one slave responds per cycle.

3.2 Addressing

Memory Mapping. The ARMv6-M architecture (used in Cortex-M0 and M0+ processors) defines a flat, linear 32-bit address space, yielding a total of 4 GiB of addressable locations. This space is logically partitioned into predefined regions — not all of which correspond to physical memory — and the layout is standardized across all ARMv6-M implementations to ensure software portability. The canonical ARMv6-M memory map is divided as follows:

Address Range	Region Name	Description
<code>0x0000_0000</code> - <code>0x1FFF_FFFF</code>	Code (Alias)	Typically maps to Flash (via aliasing mechanisms); supports execution-in-place (XIP).
<code>0x2000_0000</code> - <code>0x3FFF_FFFF</code>	SRAM	On-chip RAM (often tightly coupled memory, TCP-like behavior).
<code>0x4000_0000</code> - <code>0x5FFF_FFFF</code>	Peripheral	Memory-mapped I/O space for system and external peripherals (e.g., UART, GPIO, timers).
<code>0x6000_0000</code> - <code>0x9FFF_FFFF</code>	External RAM	Optional external memory (e.g., SDRAM, SRAM via FSMC/EBI).
<code>0xA000_0000</code> - <code>0xDFFF_FFFF</code>	External Devices	Memory-mapped external peripherals (e.g., LCD controllers, Ethernet MACs).
<code>0xE000_0000</code> - <code>0xFFFF_FFFF</code>	Private Peripheral Bus (PPB)	System control space: NVIC, SysTick, MPU, debug components (e.g., DWT, ITM, ROM table).

Table 9 / Memory map of a typical ARM Cortex-M-based microcontroller.

Crucially, all I/O in ARMv6-M is memory-mapped — there is no dedicated I/O address space or I/O instructions. Peripherals — both general-purpose (e.g., GPIO at `0x4002_0000`) and system-critical (e.g., NVIC at `0xE000_E100`) — occupy fixed or configurable regions within the unified 32-bit address map.

Though all peripherals use the same address space, they are not treated like normal memory. The architecture assigns memory type attributes per region, enforced by the optional Memory Protection Unit (MPU) or hardwired in the bus matrix. These attributes ensure that accesses to peripheral registers have deterministic timing and side effects, preventing compiler or hardware optimizations (e.g., reordering, caching, write combining) that would break device driver logic. Additionally, while ARMv6-M is byte-addressable, not all peripheral support arbitrary byte or half-word accesses — even if the address is aligned. Access width must match the register definition; mismatched accesses may cause bus faults or undefined behavior.

Isolated (Port-Mapped) I/O. In contrast, isolated I/O, also known as port-mapped I/O (PMIO), uses a separate address space dedicated exclusively to I/O devices — distinct from the main memory address space. Key features:

- A dedicated I/O address space, typically much smaller (e.g., 64 KiB).
- Specialized instructions for I/O access, e.g., `IN` and `OUT` on x86, or `IN` / `OUT` mnemonics on 8051 or Z80.
- The CPU's memory-access instructions (e.g., `LDR`, `STR`) cannot access I/O ports; only I/O instructions can.
- Physical separation: often implemented via a distinct set of control signals (e.g., `/IOR`, `/IOW` on ISA bus vs. `/MEMR`, `/MEMW`).

Advantages for isolated I/O include a larger effective memory space (since I/O doesn't consume main memory addresses, the full physical memory address space remains available for RAM/ROM), clear separation of concerns (I/O operations are syntactically and semantically distinct, reducing accidental accesses), and simpler hardware decoding (I/O port decode logic can be simpler and faster than full 32-bit memory decoding). However, it also brings drawbacks, since it requires additional instructions and CPU microarchitecture support, cannot use general-purpose addressing modes for I/O without extra overhead, inline assembly is often needed, and is harder to virtualize or remap.

3.3 Peripherals.

In embedded systems, peripheral controllers like GPIO (General Purpose Input/Output), UART (Universal Asynchronous Receiver-Transmitter), and SPI (Serial Peripheral Interface) are typically accessed via memory-mapped interfaces. To interact with these peripherals, software reads from and writes to specific registers located at fixed address offsets. For example, the Xilinx UARTLite has a Receive FIFO at offset `0h`, a Transmit FIFO at `04h`, a Status Register at `08h`, and a Control Register at `0Ch`.

C-Level Memory Access. We can interact with these hardware registers using C pointers. We can define `peek` (read) and `poke` (write) functions to demonstrate this mechanism. The pointer is dereferenced to access the specific memory location assigned to the device.

```
C C

// Reading from a register
int peek(char *location) {
    return *location; // dereference location pointer to read
}

// Writing to a register
void poke(char *location, char newval) {
    (*location) = newval; // Write newval to address 'location'
}

// Example usage
#define DEV1 0x1000
dev_status = peek(DEV1); // Read status
poke(DEV1, 8); // Write 8 to the device
```

When writing this type of low-level code, specific keywords ensure the compiler handles variables correctly:

- `const` : Makes a variable or pointer parameter unmodifiable.
- `static` : Preserves a variable's value after its scope ends and prevents it from being placed on the stack.
- `volatile` : This is critical for embedded I/O. It tells the compiler that the variable (or register) can change in the background (e.g., by hardware) without the software's explicit action. It prevents the compiler from optimizing away reads/writes to that address.

Data Transmission Protocols. Data transmission between devices relies on knowing when data is valid. This is handled in two main ways:

- (1) *Synchronous Communication*: Uses a separate clock signal. The receiver samples data on a specific clock edge (e.g., rising edge). This is common in parallel communication or protocols like SPI.
- (2) *Asynchronous Communication*: Does not use a clock signal. Instead, the receiver infers bit times based on a fixed baud rate and specific framing delays relative to a reference event (like a start bit).

UART (Universal Asynchronous Receiver-Transmitter) is the standard implementation for asynchronous serial communication. Since there is no clock, it relies on strict framing to signal the start and end of data. The line is held high during the Idle state. The transmission frame consists of:

- *Start Bit*: The transmitter pulls the line low to signal the beginning of a message.
- *Data Bits*: Usually 8 bits are sent, starting with the LSB.
- *Parity Bits*: Used for error detection. In Even Parity, the bit is set to make the total number of 1s even. In Odd Parity, it makes the total count odd. This can detect an odd number of bit errors but fails if an even number of bits are flipped.
- *Stop Bit*: The line is pulled high to end the frame and return to the Idle state.

There are two primary models for managing peripheral data flow:

- (1) *Polling (Busy Wait)*: In polling, the processor continually interrogates the peripheral's status register to check if it is ready to send or receive data. While simple to implement, this wastes CPU cycles because the processor is fully occupied "waiting" for the hardware. The following code demonstrates a blocking send routine for the UARTLite. It sits in a `while` loop checking the "Transmit FIFO Full" bit in the status register before writing a byte.

```
C

void XUartLite\_SendByte(int *BaseAddress, char Data) {
    // Busy wait: keep looping while the Transmit FIFO is full
    while (XUartLite\_IsTransmitFull(BaseAddress));

    // Write data to the TX FIFO register
    XUartLite\_WriteReg(BaseAddress, XUL\_TX\_FIFO\_OFFSET, Data);
}
```

- (2) *Interrupts*: Interrupts are asynchronous events that trigger the execution of a specific procedure called an interrupt handler. These allow the CPU to only be active regarding the peripheral when a status change actually occurs, allowing it to do other work otherwise, but they introduce complexity in debugging and predictability. The flow is as follows:
- The peripheral asserts an interrupt signal.
 - The processor saves its current context (PC, registers) and jumps to the interrupt handler.
 - The processor acknowledges the interrupt (often by toggling an acknowledgment bit in the interrupt controller or the peripheral).
 - After handling the event, the processor restores the context and resumes the main program.

In Xilinx designs, interrupts are managed by an AXI Interrupt Controller. To enable interrupts on a peripheral like the AXI GPIO, you must set the Global Interrupt Enable (GIE) bit and the Interrupt Enable Register (IER) for the specific channel.

Memory Hierarchy

The memory hierarchy is a fundamental design concept in embedded systems used to manage the trade-off between access speed, capacity, and cost. The goal is to provide the illusion of a large, fast, and inexpensive memory to the Central Processing Unit (CPU). The hierarchy is effective because it exploits the Principle of Locality, which states that program access tends to be localized in time and space:

- *Temporal Locality*: If an item is referenced, it is likely to be referenced again soon.
- *Spatial Locality*: If an item is referenced, items whose addresses are close to it are likely to be referenced soon.

Data Access Performance. When the CPU attempts to access data, one of two outcomes occurs:

- (1) *Hit*: The accessed data is present in the upper level. The access is satisfied quickly by this upper, faster level. The Hit Ratio is calculated as the ratio of hits to total accesses.

$$\text{Hit Ratio} = \frac{\text{Hits}}{\text{Accesses}} \quad (10)$$

- (2) *Miss*: The accessed data is absent from the upper level. A penalty is incurred as the block containing the required data must be copied from the lower, slower level to the upper level before the access can be satisfied.

4.1 Levels of the Hierarchy

The hierarchy is structured based on speed and proximity to the CPU:

- (1) *SRAM (Static RAM)*: This is the Cache Memory level, located closest to the CPU. It is small, fast, and expensive. Recently accessed data and spatially nearby items are copied here from DRAM.
- (2) *DRAM (Dynamic RAM)*: This is the Main Memory level. It is a larger, slower, and less expensive than SRAM. Recently accessed data and spatially nearby items are copied here from magnetic disk.
- (3) *Magnetic Disk/Solid State Drive (SSD)*: This constitutes the slowest and largest level, often referred to as Secondary Storage. All system data and programs are originally stored here.

Memory Type	Access Time	Cost per GB	Role
SRAM	0.5 ns - 2.5 ns	\$2,000 - \$5,000	Cache (Level 1, 2, 3)
DRAM	50 ns - 70 ns	\$20 - \$75	Main Memory
Magnetic Disk	5 ms - 20 ms	\$0.20 - \$2	Secondary Storage

Table 11 / Comparison of memory technologies by access time, cost, and hierarchical role.

DRAM Technology. DRAM stores each data bit as an electrical charge in a small capacitor, with a single transistor acting as a switch to control access to that charge. The primary challenge of DRAM is its volatility leakage: the charge stored in the capacitor dissipates over time. To prevent data loss, DRAM must be periodically refreshed. This process involves reading the contents of a capacitor and immediately writing the charge back, restoring it to its original level. Refreshing is performed on an entire DRAM row at once, rather than on individual cells. This row-based operation minimized the overhead of refreshing, but means that during the refresh cycle, the entire row is unavailable for CPU access.

Bits within a DRAM chip are organized into a two-dimensional rectangular array of row and columns.

- *Row Buffer:* When a row is addressed, the entire row is accessed and copied into a temporary high-speed register called the row buffer (or cache). This allows multiple consecutive words from the same row to be supplied with very low latency, known as burst mode access. This reuse is critical for exploiting spatial locality.
- *Synchronous DRAM (SDRAM):* SDRAM synchronizes its operation with the CPU clock, enabling control signals to be sent in anticipation of data needs. This allows for consecutive accesses to occur in bursts without having to send a full address for every word, significantly improving bandwidth.

Performance in advanced embedded systems is heavily reliant on memory bandwidth. Several techniques are used to increase the rate at which data can be transferred:

- (1) *Data Transfer Rates:* Double Data Rate (DDR) DRAM is a technology that increases the transfer rate by performing data transfers on both the rising and falling edge of the clock signal, effectively doubling the memory clock speed. Quad Data Rate (QDR) DRAM is an enhancement, often used in high-speed applications like network switches, where the DDR principles are applied to separate input and output ports, further maximizing concurrent data movement.
- (2) *Architectural Parallelism:* DRAM performance can be scaled up through physical organization. The memory bus width determines how many words (or bytes) can be transferred in a single access cycle. A 4-word wide bus, for example, transfers 4 words simultaneously, which dramatically reduces the effective latency per byte for contiguous data transfers and improves peak bandwidth. A DRAM chip is divided into independent sections called banks. DRAM banking allows for simultaneous access to multiple independent banks. Interleaving is the process of mapping sequential memory addresses across different banks. While accessing a new bank may introduce a delay (e.g., a 20-cycle bank switching penalty), interleaving allows the system to overlap the row access delays, dramatically improving sustained throughput under random or scattered access patterns.

Flash Storage. Flash storage is a type of non-volatile semiconductor memory. It retains data without power and provides significant performance advantages over traditional disk storage, being typically 100x to 1,000x faster. It also offers advantages in physical footprint, power consumption, and robustness (due to no moving parts), though it remains more expensive per gigabyte than magnetic disk. Flash is broadly divided into two architectures, defined by the arrangement of the memory cells:

- (1) *NOR Flash*: The memory cells are connected in parallel, similar to a NOR logic gate. It permits random read and write access at the byte level. It is primarily used for instruction memory in embedded systems (e.g., boot code) due to its rapid access and execution-in-place (XIP) capability.
- (2) *NAND Flash*: The memory cells are connected in series, resembling a NAND logic gate. It is denser and cheaper per GB, but requires access in large blocks at a time (block-at-a-time access). It is used for high-capacity storage devices like USB keys, Solid State Drives (SSDs), and media storage.

A critical limitation of flash storage is cell wear-out: individual flash memory blocks can only sustain a finite number of Program/Erase cycles before they become unreliable. This makes flash unsuitable as a direct, unlimited replacement for DRAM or traditional magnetic disk. To mitigate this, wear leveling algorithms are implemented by the flash controller. These algorithms dynamically remap logical data addresses to different physical blocks, ensuring that write operations are spread evenly across all available blocks in the device, thus maximizing the overall lifespan.

Magnetic Disk Storage. Magnetic Disk Storage is a traditional form of non-volatile rotating magnetic storage. Although slower than flash, it offers the lowest cost per gigabyte for mass storage. A disk is composed of:

- *Platters*: Circular magnetic surfaces where data is stored.
- *Tracks*: Concentric rings on the platter surfaces.
- *Cylinders*: The set of tracks at the same radial distance on all platters.
- *Sectors*: The smallest unit of physical data storage, typically holding 512 bytes (or 4096 bytes in newer formats).

Each sector includes additional overhead fields, such as the sector ID, the actual payload, Error-Correcting Code (ECC) and synchronization fields and gaps.

Accessing a sector on a magnetic disk is a mechanical process involving multiple latency components:

$$T_{\text{access}} = T_{\text{queue}} + T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}} + T_{\text{controller}} \quad (12)$$

- (1) *Queuing Delay*: The time spent waiting if other I/O requests are pending.
- (2) *Seek Time*: The time required to move the read/write head assembly to the correct cylinder. Manufacturers quote an average seek time, but actual seek times are often smaller due to locality and operating system scheduling algorithms (like SCAN or C-SCAN) that order requests to minimize head movement.
- (3) *Rotational Latency*: The time required for the desired sector to rotate under the head.
- (4) *Data Transfer Time*: The time needed to physically read the data bits once the head is in position.
- (5) *Controller Overhead*: Time for the disk controller to manage the request.

Modern disk drives include a smart controller that manages the internal complexities of the physical drive, presenting a simple logical sector interface to the host system (via standards like SCSI, ATA or SATA). These controllers often include a disk cache to prefetch sectors in anticipation of future accesses, which helps mask rotational latency and seek delay for sequential access patterns.

4.2 Cache Memory

Cache memory is the level of the memory hierarchy closest to the CPU, typically implemented using fast Static RAM (SRAM). Its primary role is to act as a temporary staging area for data that the CPU is likely to access soon, thereby reducing memory access latency. In a Direct-Mapped Cache, each memory block has only one possible location where it can be stored in the cache. This location is determined by a simple, deterministic mapping function derived from the memory address.

Address Mapping. The memory address is divided into three fields:

$$\text{Address} = [\text{Tag}] [\text{Index}] [\text{Offset}] \quad (13)$$

- (1) *Block Offset:* These are the low-order bits of the address, which specify the byte offset within the block. The number of offset bits is $\log_2(\text{Block Size})$.
- (2) *Cache Index:* These bits determine the specific line in the cache where the block will be stored. Since the number of blocks in the cache (N) is typically a power of 2, the index is found by taking the block's address modulo N . These are the bits immediately to the left of the Block Offset.
- (3) *Tag:* These are the high-order bits of the address. Since many main memory blocks can map to the same cache index, the Tag is necessary to uniquely identify which particular memory block is currently stored in that cache location.

When the CPU issues an address, the cache controller extracts the Index to find the corresponding cache line. To ensure correctness, two checks are performed:

- *Tag Match:* The stored tag in the cache line must match the Tag field of the requested address.
- *Valid Bit:* A Valid Bit is stored with each cache line. It is set to '1' if the line currently holds valid data from main memory and '0' otherwise. All valid bits are initialized to '0'. A hit occurs only if both the Tag matches and the Valid Bit is '1'.

A crucial design parameter is the Block Size. Increasing the block size generally reduces the miss rate because it takes greater advantage of spatial locality: when a miss occurs, more neighboring data is brought into the cache. However, for a cache of fixed total capacity, increasing the block size means the cache holds fewer blocks overall. This leads to greater competition for the available cache lines. If the larger block brings in data that is not used, it occupies space needed by other, more frequently accessed data. This unused data is referred to as cache pollution and can ultimately lead to an increased miss rate, overriding the initial benefit. To mitigate the performance impact of a larger miss penalty, two techniques can be employed:

- *Early Restart*: The CPU is allowed to continue execution as soon as the specific word that caused the miss is retrieved from the next memory level, even if the rest of the block has not yet arrived.
- *Critical-Word-First*: A variation where the requested word is transferred first, and the rest of the block follows in a non-sequential order.

Cache Write Policies. In the case of a Cache Miss, the CPU pipeline stalls. The entire block containing the requested data is fetched from the next lower level of the memory hierarchy (e.g., main memory/DRAM). In the case of Instruction Cache Miss, the instruction fetch step of the pipeline must be restarted once the instruction block arrives. On the other hand, for a Data Cache Miss, the data access step of the pipeline must wait to complete the data access once the data block arrives.

Cache write policies dictate how and when a write operation from the CPU updates the data in the cache and the corresponding block in the next level of the memory hierarchy. The primary goal is to maintain data coherence.

- (1) *Write-Through*: In the Write-Through policy, on a data write hit, the block in the cache is updated, and the write is immediately forwarded to the next level of hierarchy. Cache and memory are consistently synchronized on every write. However, writes become significantly slower because the CPU must wait for the main memory access to complete, which is much slower than the cache. If the baseline Cycles Per Instruction (CPI) is 1, and 10% of instructions are write operations, and the write latency to memory is 100 cycles, the effective CPI is dramatically increased:

Write Policy	Miss Alternatives	Description
Write-Through	Write Allocate (Fetch on Miss)	The block is fetched from memory, the data is written on the cache block, and the write is passed through to memory.
	No-Write Allocate (Write Around)	The block is not fetched. The write operation bypasses the cache and is performed only on the next level of the hierarchy. This is often used because many programs write a full block of data sequentially before reading it, making the intermediate fetch unnecessary ("write around").
Write-Back	Write Allocate (Fetch on Miss)	The missing block is usually fetched from memory first. The data is then written into the cache block, and the dirty bit is set. This ensures that the entire block is coherent before modification.

Table 14 / How cache write policies (write-through/write-back) handle write misses.

$$\text{Effective CPI} = 1 + (0.10 \times 100) = 11 \quad (15)$$

- (2) *Write-Back*: In the Write-Back policy, on a data write hit, only the block in the cache is updated. The corresponding block in main memory is not immediately updated. A dirty bit (or modified bit) is associated with each cache block. It is set to '1' when the block is modified by a write. The block is written back to the next level of the hierarchy only when that dirty block is selected for replacement by a new block. This policy leads to much lower write traffic to memory and much faster write operations for the CPU, but it introduces complexity in maintaining coherence. A write buffer is often used here too, but to store the evicted dirty block temporarily while the replacement block is being read into the cache.

Performance. Cache performance is measured by decomposing CPU execution time into two key components: program execution cycles, which include the time spent on cache hits, and memory stall cycles, primarily incurred due to cache misses. Under simplifying assumptions, memory stall cycles can be estimated as the product of memory accesses per program, the miss rate, and the miss penalty. Since memory accesses are often proportional to instructions executed, this can also be expressed as:

$$\text{Instruction per Program} \times \text{Misses per Instruction} \times \text{Miss Penalty} \quad (16)$$

offering a practical way to quantify the performance cost of cache misses.

Average Memory Access Time (AMAT) is a critical metric for evaluating cache performance, as it combines both hit time and the cost of misses. While miss and penalty are often emphasized, hit time itself significantly impacts overall speed, especially in high-frequency CPUs where every cycle counts. The formula is straightforward:

$$\text{AMAT} = \text{Hit Time} + (\text{Miss Rate} \times \text{Miss Penalty}) \quad (17)$$

Cache Organization and Design

To overcome the high conflict misses inherent in a direct-mapped cache, where a memory block has only one possible location, Associative Caches are used. Associativity allows a given memory block to be placed in multiple possible cache entries:

- *Fully Associative Cache*: In a Fully Associative Cache, a block can be placed in any available cache entry. There is no index; the cache is searched using only the Tag. To check for a hit, all cache entries must be searched simultaneously. This requires a comparator circuit for every single entry, making it complex and expensive to implement.
- *N-Way Set Associative Cache*: The N-Way Set Associative Cache offers a compromise between the speed of direct mapping and the flexibility of fully associative mapping. The cache is divided into a number of sets, and each set contains N entries. A block is mapped to a specific set determined by the calculation: Block Number mod # Sets in cache. Once the set is determined, only the N entries within that specific set are searched simultaneously. This requires only N comparators, making it much less expensive than a fully associative cache.

When a set is full and a new block needs to be brought in, a replacement policy is needed to decide which existing block to evict. This choice is only necessary for set-associative and fully-associative caches, as a direct-mapped cache has no choice.

- (1) *Prefer Non-Valid Entry*: If an entry within the set has an invalid bit (meaning it contains no useful data), that entry is always chosen for replacement.
- (2) *Least-Recently Used (LRU)*: If all entries are valid, the policy chooses the entry that has been unused for the longest period of time. LRU is highly effective but becomes computationally complex for associativity beyond 4-way.
- (3) *Random*: This policy randomly selects an entry to replace. It provides approximately the same performance as LRU for caches with high associativity.

Multilevel Caches. Modern systems use multiple levels of caches to reduce the average memory access time.

- *Primary Cache (L1)*: Attached directly to the CPU. It is small and very fast (designed for minimal hit time). It services the vast majority of CPU requests.
- *Level-2 Cache (L2)*: Services the misses that occur in the primary cache. It is larger and slower than L1, but still significantly faster than main memory (designed for a low miss rate).
- *Main Memory (DRAM)*: Services the misses that occur in the L2 cache. Some high-end systems may also include a Level-3 (L3) cache.

Parallelism

5.1 Instruction-Level Parallelism (ILP)

Pipelining serves as the fundamental technique for ILP by overlapping the execution of multiple instructions. To further maximize ILP, systems can utilize deeper pipelines, which decompose instructions into more numerous, smaller stages. This reduction in work per stage allows for a higher clock frequency (shorter clock cycles). However, as pipeline depth increases, the overhead of hazards and the complexity of hardware logic also scale.

Multiple Issue. To push performance beyond the theoretical limit of a single-issue pipeline, where the Cycles Per Instruction (CPI) is at best 1, architectures employ Multiple Issue techniques. In these systems, the Instructions Per Cycle (IPC) becomes the primary metric of throughput.

- *Static Multiple Issue:* In a static framework, the responsibility of managing parallelism shifts to the compiler. The compiler groups instructions into issue packets, which are perceived by the hardware as a single "very long instruction" (VLIW).
 - *Packet Construction:* The compiler packages instructions based on available pipeline resources (issue slots).
 - *Hazard Management:* The compiler is responsible for detecting and resolving hazards. It must ensure that no dependencies exist within a single packet and manage dependencies between successive packets according to the specific ISA rules.
 - *Resource Alignment:* If the compiler cannot find enough independent instructions to fill a packet, it must pad the remaining slots with `nop` (no-operation) instructions.

A common implementation is the MIPS Static Dual-Issue processor, which fetches and executes a "packet" of two instructions every clock cycle. To simplify the hardware decoding logic, these issue packets are strictly formatted. They must be 64-bit aligned, containing two 32-bit instructions in a specific order:

- (1) *Slot 1:* An ALU or Branch instruction.
- (2) *Slot 2:* A Load or Store instruction.

If the compiler cannot find a pair of instructions that meet these criteria and lack dependencies, it must pad the unused slot with a `nop`. This ensures the hardware always receives a predictable instruction pair.

While dual-issue increases potential throughput, it introduces more restrictive timing constraints for data hazards, requiring more aggressive compiler scheduling:

- *EX Data Hazards:* In a single-issue pipeline, forwarding allows an ALU result to be used immediately by next instruction. However, in a dual-issue packet, an ALU result produced in Slot 1 cannot be used by a Load/Store in Slot 2 of the same packet. The instructions are executing in the same stage simultaneously, so the result has not yet been generated when the second instruction requires its operands.

- *Load-Use Hazards:* The "use latency" for a load instruction remains one cycle, but the impact is doubled. Because the processor issues two instructions per cycle, a one-cycle delay now stalls two potential instruction slots instead of one.

To maintain efficiency, the compiler must look further ahead in the code to find independent instructions to fill these gaps, a process known as Static Instruction Scheduling.

- *Dynamic Multiple Issue:* Superscalar processors represent a shift in responsibility from the compiler to the hardware. In this architecture, the CPU dynamically decides whether to issue zero, one, or multiple instructions during each clock cycle. The primary objective is to maximize throughput by avoiding structural and data hazards in real-time. While a compiler can still assist by reordering instructions to be "hardware-friendly", it is no longer strictly required for functional correctness. The CPU hardware ensures that the final code semantics remain identical to a sequential execution, regardless of how the instructions were shuffled internally.

To mitigate stalls, superscalar units often employ Dynamic Pipeline Scheduling. This allows the CPU to execute instructions out-of-order as soon as their operands are available. However, to maintain architectural integrity, the system must commit results to registers in-order. This ensures that if an interrupt or exception occurs, the machine state is inconsistent and predictable.

Loop unrolling is a code transformation technique that reduces loop overhead by replicating the loop body multiple times and decreasing the total number of iterations. It is a key method for exposing Instruction-Level Parallelism in both static and dynamic multiple-issue systems. By executing multiple iterations of the original loop within a single unrolled iteration, the processor significantly reduces the frequency of administrative instructions, which means fewer increments of the loop counter and fewer conditional branches are executed. In deep pipelines, every branch is a potential stall or misprediction risk; unrolling reduces the total number of branches, keeping the pipeline streaming longer.

Speculation. Speculation is a sophisticated technique used in both static and dynamic systems to preemptively execute instructions before the processor is certain they are required. This is particularly effective for overcoming the bottlenecks of branch delays and long-latency load operations. The process involves a "guess-check-recover" cycle: the system guesses the outcome of a control or data flow decision, executes the operation based on that guess, and then verifies the result. If the guess was correct, the operation is committed; if incorrect, the system performs a roll-back to restore the previous architectural state.

- *Compiler Speculation:* The compiler may reorder instructions, such as moving a `load` operation before a preceding `branch`. To handle potential errors (like a memory protection fault on a speculative load), the compiler integrates "fix-up" code to recover gracefully.
- *Hardware Speculation:* The processor uses look-ahead buffers to execute instructions out-of-order. The results are stored in temporary buffers (such as reorder buffers) and are only written to the permanent architectural state once the speculation is confirmed as correct. If the speculation fails, these buffers are simply flushed.

Conclusions. While compilers are powerful, they are limited by the information available at "compile-time". Hardware scheduling is superior for several reasons:

- *Unpredictable Stalls:* Many stalls, such as cache misses, are non-deterministic. A compiler cannot know exactly when a data request will miss the cache, but the CPU can detect the miss and immediately switch to other independent instructions.
- *Dynamic Branch Outcomes:* The specific path a program takes often depends on runtime data. Hardware can use dynamic branch predictors to speculatively execute paths that a static compiler cannot definitively identify.
- *Microarchitectural Portability:* Different implementations of the same Instruction Set Architecture (ISA) may have varying pipeline depths, latencies, and functional units. Hardware scheduling allows the same binary code to run efficiently across different CPU generations without needing to be recompiled for every specific hardware hazard.

Despite the sophistication of multiple-issue and speculation, several "walls" prevent us from achieving infinite parallelism. Programs possess inherent dependencies that create a ceiling for Instruction-Level Parallelism.

- *Data and Resource Dependencies:* Real dependencies, such as a calculation requiring the result of the immediately preceding one, create chains that cannot be parallelized.
- *Pointer Aliasing:* This is a significant hurdle in languages like C. If the compiler or hardware cannot be certain whether two different pointers refer to the same memory address, it must assume a dependency exists, which prevents reordering load/store operations.
- *Limited Windows Size:* A CPU can only "look ahead" at a certain number of instructions to find parallelism. Increasing this window requires massive amounts of power and silicon area, leading to diminishing returns.
- *Memory Bottlenecks:* Modern CPUs are significantly faster than the memory systems that feed them. Limited memory bandwidth and high latency make it difficult to keep deep, wide pipelines consistently full of useful instructions.

Speculation remains the most effective tool for breaking through these limits, but it must be executed with high accuracy. If speculation fails too often, the energy and cycles spent on "wrong-path" execution actually decrease performance below that of a simpler, non-speculative processor.

5.2 Single Instruction, Multiple Data (SIMD)

SIMD (Single Instruction, Multiple Data) is an architecture category designed to exploit Data-Level Parallelism (DLP). By applying a single operation to an entire vector of data simultaneously, SIMD significantly accelerates computationally intensive tasks like digital signal processing (DSP), audio/video encoding, and deep learning inference.

To understand SIMD, it is helpful to contrast it with the traditional processing model:

- *SISD:* This is the conventional scalar processing model (e.g., standard MIPS or ARM code). The CPU executes one instruction to process exactly one data item at a time. To add two arrays of 8 elements, a SISD processor must execute 8 separate addition instructions in a loop.
- *SIMD:* Also known as Vector Parallelism, this model allows one instruction to operate on a "pack" of data elements (a vector). In the same example of adding arrays, a SIMD processor

with an 8-lane vector width can perform all 8 additions in a single clock cycle using one instruction.

ARM NEON: Advanced SIMD. NEON is the ARM implementation of advanced SIMD technology. It functions as a dedicated accelerator integrated into the processor, specifically designed to handle the high-throughput requirements of modern multimedia and AI workloads. There are three primary ways to utilize NEON in an embedded system:

- (1) *Auto-vectorization*: The compiler automatically analyzes standard C/C++ loops and converts them into NEON instructions, which is an operation that requires specific compiler flags (e.g., `-O3 -mfpu=neon`).
- (2) *NEON Intrinsics*: These are special C/C++ function calls that map directly to NEON instructions. They provide low-level control over the hardware while allowing the compiler to handle register allocation and instruction scheduling.
- (3) *Hand-coded Assembly*: For maximum performance, developers can write raw NEON assembly. This is typically reserved for critical "hot spots" in a program where the compiler's output is not sufficiently optimized.

NEON utilizes a dedicated register file that is distinct from the general-purpose ARM registers (`r0 - r15`). This file is 256 bytes large and offers a dual-view architecture, allowing the hardware to treat the same physical storage in two different ways:

- *D Registers (Double-word)*: 32-bit registers, each 64 bits wide (`D0 - D31`).
- *Q Registers (Quad-word)*: 16-bit registers, each 128 bits wide (`Q0 - Q15`).

The registers are aliased, meaning `Q0` is physically composed of `D0` and `D1`, `Q1` is composed of `D2` and `D3`, and so on. This flexibility allows the programmer to choose the vector length that best fits the data type and algorithm. Each NEON register is viewed as a vector of elements of the same type, known as lanes. The number of lanes depends on the data size:

- A 128-bit Q register can hold:
 - 16 elements of 8-bit data (e.g., pixels).
 - 8 elements of 16-bit data (e.g., audio samples).
 - 4 elements of 32-bit data (e.g., floating-point values).
 - 2 elements of 64-bit data.

On many ARM architectures, the NEON register file is shared with the VFP (Vector Floating Point) unit. While they share registers, they serve different purposes: the VFP is a fully IEEE-754 compliant floating-point coprocessor for high-precision scalar math, whereas NEON is a high-speed, parallel engine for throughput-oriented vector math.

5.3 Parallel Processors

The fundamental objective of parallel processing is the interconnection of multiple processing elements to achieve enhanced computational performance. This architecture is generally categorized into three operational scales: task-level parallelism, which prioritizes high throughput for

independent workloads; parallel processing programs, where a single application is distributed across multiple processors; and multicore microprocessors, which integrate several processing units, or cores, onto a single physical substrate.

Power Dynamics and Efficiency. Power efficiency remains a primary constraint in the design of advanced embedded systems. The dynamic power consumption of a circuit is governed by the switching power equation:

$$P_{\text{switch}} = \alpha \cdot C \cdot V_{\text{dd}}^2 \cdot f \quad (18)$$

In this context, α represents the activity factor, C the capacitance, V_{dd} the supply voltage, and f the operating frequency. To optimize for power, designers often reduce the number of pipeline stages, which subsequently lowers the total flip-flop count. Furthermore, the synthesis process itself impacts the energy profile; enforcing higher frequency constraints during logic synthesis typically results in more power-hungry netlists. Multicore architectures mitigate some of these issues by allowing for core specialization, where specific cores are optimized for distinct tasks rather than relying on a single, high-frequency general-purpose unit.

The primary bottleneck in parallel computing is the complexity of the software rather than the hardware. To justify the transition from a uniprocessor to a multicore system, the software must demonstrate significant performance improvements; otherwise, the simplicity of a faster uniprocessor remains preferable. These challenges are primarily rooted in three areas:

- (1) *Partitioning*: Dividing the problem into discrete, parallelizable tasks.
- (2) *Coordination*: Managing the synchronization between concurrent threads.
- (3) *Communication Overhead*: The latency and bandwidth costs associated with data exchange between cores.

Amdahl's Law. Amdahl's Law provides a theoretical framework for predicting the maximum speedup of a system when only a portion of it is improved or parallelized. It establishes that the overall performance gain is strictly limited by the sequential fraction of the program, which cannot benefit from additional processors. The fundamental relationship for speedup is defined by the fraction of the code that is parallelizable (P). In an ideal scenario with infinite processors, the maximum speedup is constrained by the remaining serial fraction ($1 - P$):

$$\text{Speedup}_{\max} = \frac{1}{1 - P} \quad (19)$$

When considering a finite number of processors (N), the execution time is the sum of the time spent on the serial portion (S) and the time spent on the parallel portion (P) distributed across those processors. This is modeled by the equation:

$$\text{Speedup} = \frac{1}{S + \frac{P}{N}} \quad (20)$$

In this mode, S represents the serial fraction, P is the parallel fraction (where $S + P = 1$), and N is the number of processing elements. As N increases, the term $\frac{P}{N}$ diminishes, leaving the serial fraction S as the ultimate bottleneck.

The effectiveness of parallelization is often evaluated through two distinct lenses:

- *Strong Scaling*: This approach keeps the total problem size fixed while increasing the number of processors. The primary goal is to reduce the "time to solution". Strong scaling is heavily governed by Amdahl's Law, as the serial portion becomes increasingly dominant as the parallel work per processor shrinks.
- *Weak Scaling*: In this perspective, the problem size grows proportionally with the number of processors, maintaining a constant workload per processor. The objective is to solve larger, more complex problems in the same amount of time rather than solving a small problem faster. Weak scaling is often modeled by Gustafson's Law, which suggests that parallel performance can scale more linearly if the workload is allowed to expand.

Multithreading and Shared Memory Architectures

Multithreading is a technique that enables the parallel execution of multiple threads by replicating hardware resources, such as registers and Program Counters (PCs), to allow for rapid context switching. Unlike traditional process switching, which involves heavy OS overhead, hardware multithreading minimizes latency by maintaining thread states in the processor pipeline. The efficiency of multithreading depends on the frequency and conditions under which the processor switches execution contexts:

- *Fine-grained Multithreading*: The processor switches between threads after every clock cycle, interleaving instructions in a round-robin fashion. If one thread stalls (e.g., due to a dependency), the pipeline remains utilized by executing instructions from other threads. While this maximizes throughput, it can slow down the execution of an individual prime thread.
- *Coarse-grained Multithreading*: Context switching only occurs during significant stalls, such as an L2 cache miss. This approach simplifies hardware design as it does not require cycle-by-cycle switching; however, it is less effective at masking short-term stalls like data hazards or functional unit contention.
- *Simultaneous Multithreading*: Employed in multiple-issue, dynamically scheduled processors, SMT allows instructions from different threads to execute concurrently within the same clock cycle. By leveraging the superscalar architecture, SMT utilizes functional units that would otherwise remain idle. Within individual threads, dependencies are managed through dynamic scheduling and register renaming.

In an SMP system, the hardware provides a single, unified physical address space accessible by all processors. Communication between threads is managed by synchronizing shared variables, typically through the use of software locks. The performance of these systems is often categorized by the memory latency:

- *Uniform Memory Access (UMA)*: All processors share the same memory latency when accessing any part of the physical memory.

- *Non-Uniform Memory Access (NUMA)*: Memory is physically distributed; a processor can access its local memory faster than remote memory blocks assigned to other processors.

When two or more processors share a memory region, hardware-level support for atomic operations is required to prevent race conditions. An atomic read-modify-write operation ensures that no other processor can access or modify a specific memory location between the initial read and the final write. This atomicity is typically implemented via:

- A single atomic instruction (e.g., Test-and-Set).
- An atomic pair of instructions (e.g., Load-Linked / Store-Conditional).
- Hardware-level mutexes or semaphores managed by the memory controller.

Accelerator Control and Execution Models. Effective accelerator integration requires managing the flow of control between the CPU and specialized hardware. This is primarily handled through two execution models: single-threaded (blocking) and multithreaded (non-blocking) control. In a single-threaded environment, the CPU initiates an accelerator task and enters a stall or wait state until the accelerator completes its operation. This sequential execution creates significant idle time for the CPU. Conversely, a multithreaded model employs a split-join architecture. After the CPU triggers the accelerator (the "split"), it continues executing independent tasks while the accelerator runs in parallel. The execution flows then "join" once both the CPU and accelerator have reached their respective synchronization points.

Partitioning and Data Distribution. Partitioning is the process of decomposing a problem into smaller, parallel tasks. This decomposition generally follows two methodologies:

- *Domain Decomposition*: The problem data set is divided into discrete chunks, and each task performs similar operations on different pieces of data.
- *Functional Decomposition*: The problem instruction set is divided. In this model, different tasks perform different operations, often arranged in a pipeline where data flows from one task to the next.

For effective hardware mapping, data is distributed using 1D or 2D patterns. Common distribution strategies include Block, where contiguous data chunks are assigned to a processor, and Cyclic, where data elements are interleaved across processors to ensure a better load balance.

Scheduling, Mapping and Benchmarking. Scheduling involves the assignment of tasks to specific processors while satisfying several constraints. To optimize execution time, designers must account for the task graph, which defines the order of execution, and inter-task dependencies. Furthermore, communication-related overhead, such as the time data spends on the bus or network, must be factored into the total execution cost. For example, a task transfer might consume specific time units on the system bus, delaying the start of dependent tasks on remote processors.

To evaluate these systems, several standardized benchmark suites are utilized:

- *Linpack*: Measures matrix linear algebra performance.
- *SPECrate*: Measures the throughput of parallel runs of SPEC CPU programs.
- *SPLASH/PARSEC*: Suites focused on shared-memory multithreaded applications.

- *NAS*: Kernels focused on computational fluid dynamics.

The Roofline Model

The Roofline model is a visual framework used to determine the attainable performance of a kernel on a specific architecture. It relates floating-point performance to Arithmetic Intensity, which is defined as the ratio of floating-point operations (FLOPs) to the total bytes of memory accessed:

$$\text{Arithmetic Intensity} = \frac{\text{FLOPs}}{\text{Bytes}} \quad (21)$$

The attainable performance is constrained by two hardware limits: the Peak Floating-Point Performance and the Peak Memory Bandwidth. The formula for attainable GFLOPs/sec is:

$$\text{Att. GFLOPs/sec} = \min(\text{Peak Memory BW} \times \text{Arithmetic Intensity}, \text{Peak FP Performance}) \quad (22)$$

Optimization Regions. A kernel's position on the Roofline diagram dictates the necessary optimization strategy:

- (1) *Memory-Bandwidth Limited*: When a kernel has low arithmetic intensity, performance is limited by how fast data can be moved from memory. Improvements here require optimizing memory usage, such as using software prefetching or ensuring memory affinity to avoid non-local accesses.
- (2) *Computation Limited*: When arithmetic intensity is high, the kernel is limited by the processor's raw speed. Performance can be improved by balancing adds and multiplies, increasing instruction-level parallelism, or utilizing SIMD instructions.

Arithmetic intensity is not always a fixed value; it can scale with the problem size or be effectively increased through caching, which reduces the number of required memory access.