System Design and Computer Architecture

Understanding Modern Computing Systems

Your Name

Department of Computer Science
Your Institution

September 8, 2025

Course Overview

1. Chapter 1: The Building Blocks of a Computer - Understanding CPU and Memory Architecture
2. Course Summary

Welcome to System Design & Computer Architecture

- **Course Objectives:**
  - Understand fundamental computer architecture concepts
  - Learn system design principles and patterns
  - Explore modern computing technologies (RISC, ARM, x86)
  - Apply knowledge to real-world system design problems
- **What You'll Learn:**
  - CPU and memory architecture
  - Instruction set architectures (RISC vs CISC)
  - System performance optimization
  - Scalable system design patterns
- **Prerequisites:** Basic programming knowledge, digital logic fundamentals

A Computer's Anatomy

- **Objective:** Understand the fundamental components of a computer system, focusing on CPU and memory architecture.
- **Topics Covered:**
  - The Von Neumann Architecture
  - CPU Components
  - Memory Hierarchy (Cache, RAM, Permanent Storage)

Why It Matters

- **Performance:** Knowing how CPU and memory interact helps in optimizing software performance.
- **System Design:** Essential for designing efficient systems and applications.
- **Troubleshooting:** Understanding hardware can aid in diagnosing performance bottlenecks.

Computer Fundamentals: Essential Terminology Part 1

- **CPU (Central Processing Unit):**
  - The "brain" of the computer that executes instructions
  - Contains multiple **cores** - independent processing units
  - Each core can handle one **thread** (sequence of instructions) at a time
- **Clock Speed (measured in GHz):**
  - Number of cycles per second the CPU can execute
  - 1 GHz = 1 billion cycles per second
  - Higher clock speed generally means faster processing
- **Cores vs Threads:**
  - **Core:** Physical processing unit within CPU
  - **Thread:** Virtual processing unit (software concept)
  - Modern CPUs can handle multiple threads per core (hyperthreading)

Computer Fundamentals: Essential Terminology Part 2

- **Memory Hierarchy (from fastest to slowest):**
  - **Registers:** Tiny, ultra-fast storage inside CPU cores
  - **Cache (L1/L2/L3):** Small, fast memory close to CPU cores
  - **RAM (Random Access Memory):** Main system memory, volatile
  - **Storage (SSD/HDD):** Permanent storage, non-volatile
- **GPU (Graphics Processing Unit):**
  - Specialized processor with hundreds/thousands of simple cores
  - Optimized for parallel processing (many tasks simultaneously)
  - Excellent for graphics, AI, and scientific computing
- **Architecture:**
  - The fundamental design and organization of a computer system
  - Defines how components communicate and process data

Computer Fundamentals: Essential Terminology Part 3

- **Instruction Set Architecture (ISA):**
  - The "language" that the CPU understands
  - Defines available operations (add, subtract, load, store, etc.)
  - Examples: x86, ARM, RISC-V
- **RISC vs CISC:**
  - **RISC:** Reduced Instruction Set Computer - simple, uniform instructions
  - **CISC:** Complex Instruction Set Computer - complex, variable instructions
  - Trade-off between instruction simplicity and capability
- **Pipeline:**
  - Technique to overlap instruction execution stages
  - Like an assembly line - multiple instructions in different stages
  - Improves overall throughput (instructions completed per second)

Performance Metrics: Understanding the Numbers

- **Frequency Measurements:**
  - **MHz:** Millions of cycles per second
  - **GHz:** Billions of cycles per second (1 GHz = 1000 MHz)
  - **Example:** 3.4 GHz = 3,400,000,000 cycles per second
- **Performance Measurements:**
  - **FLOPS:** Floating Point Operations Per Second (scientific computing)
  - **TOPS:** Tera Operations Per Second (AI/ML workloads)
  - **IPC:** Instructions Per Cycle (efficiency measure)
- **Memory Measurements:**
  - **Bandwidth:** Data transfer rate (GB/s - Gigabytes per second)
  - **Latency:** Time delay for memory access (nanoseconds)
  - **Capacity:** Amount of data storage (GB, TB)

Power and Efficiency Fundamentals

- **Power Consumption:**
  - **Watts (W):** Rate of energy consumption
  - **TDP:** Thermal Design Power - maximum heat generated
  - **Idle vs Load:** Power varies based on computational demand
- **Efficiency Metrics:**
  - **Performance per Watt:** TOPS/W, FLOPS/W
  - Higher efficiency = more computation per unit of power
  - Critical for mobile devices and data centers
- **Thermal Management:**
  - **Heat Generation:** Power consumption creates heat
  - **Thermal Throttling:** CPU reduces speed when too hot
  - **Cooling Solutions:** Fans, heat sinks, liquid cooling

Parallel Processing Concepts

- **Sequential vs Parallel Processing:**
  - **Sequential:** One task at a time (traditional approach)
  - **Parallel:** Multiple tasks simultaneously
  - **Speedup:** Theoretical maximum = number of cores/processors
- **Types of Parallelism:**
  - **Instruction-Level:** Multiple instructions per cycle
  - **Thread-Level:** Multiple threads on different cores
  - **Data-Level:** Same operation on multiple data (SIMD)
- **Challenges:**
  - **Dependencies:** Some tasks must wait for others to complete
  - **Communication:** Cores must share data and coordinate
  - **Load Balancing:** Distribute work evenly across cores

What is an Instruction? Understanding Assembly Language

- **Instruction Definition:**
  - A single, basic operation that the CPU can perform
  - Think of it as one step in a recipe
  - Written in assembly language (human-readable CPU commands)
- **Common Instruction Types:**
  - **Arithmetic:** ADD, SUB, MUL, DIV
  - **Data Movement:** MOV, LOAD, STORE
  - **Control Flow:** JUMP, BRANCH, CALL, RETURN
  - **Logic:** AND, OR, XOR, NOT
- **Example Translation:**
  - **C Code:** `int c = a + b;`
  - **Assembly:** `ADD R3, R1, R2` (R3 = R1 + R2)

Assembly Instruction Examples

- **ARM Assembly Examples:**

```
// Arithmetic Instructions
ADD R1, R2, R3        // R1 = R2 + R3
SUB R1, R2, #5        // R1 = R2 - 5 (immediate value)
MUL R1, R2, R3        // R1 = R2 * R3

// Data Movement Instructions
MOV R1, #10           // Move value 10 into R1
LDR R1, [R2]          // Load from memory[R2] into R1
STR R1, [R2, #4]      // Store R1 to memory[R2 + 4]

// Control Flow Instructions
B label               // Branch (jump) to label
BL function           // Branch with link (function call)
CMP R1, R2            // Compare R1 and R2
BEQ equal_label       // Branch if equal
```

Instruction Execution: Cycles and Timing

- **Does ADD take one cycle or multiple cycles?**
  - **Answer:** It depends on the CPU architecture!
  - Different architectures handle instructions differently
- **Simple/Early CPUs:** One instruction = Multiple cycles
  - **Cycle 1:** Fetch instruction from memory
  - **Cycle 2:** Decode instruction (understand what it does)
  - **Cycle 3:** Execute the operation (perform addition)
  - **Cycle 4:** Write result back to register
  - **Total:** 4 cycles per ADD instruction
- **Modern CPUs:** Use pipelining for efficiency
  - Each instruction still takes multiple cycles to complete
  - But CPU can start a new instruction every cycle
  - Like an assembly line in a factory

CPU Pipelining: The Assembly Line Approach

- **Pipeline Stages (typical 5-stage pipeline):**
  - **IF:** Instruction Fetch
  - **ID:** Instruction Decode
  - **EX:** Execute
  - **MEM:** Memory Access (if needed)
  - **WB:** Write Back
- **Pipeline Timeline Example:**
  - **Clock 1:** Instruction 1 starts (IF stage)
  - **Clock 2:** Instruction 1 (ID), Instruction 2 starts (IF)
  - **Clock 3:** Instruction 1 (EX), Instruction 2 (ID), Instruction 3 (IF)
  - **Clock 4:** Instruction 1 (MEM), Instruction 2 (EX), Instruction 3 (ID), Instruction 4 (IF)
  - **Clock 5:** Instruction 1 (WB), Instruction 2 (MEM), Instruction 3 (EX), Instruction 4 (ID)
- **Result:** Each instruction takes 5 cycles, but CPU completes one instruction per cycle!

RISC vs CISC: Instruction Complexity Impact

- **RISC Instructions (ARM example):**
  - **Simple, uniform:** All instructions same size (32-bit)
  - **Predictable timing:** ADD always takes same number of cycles
  - **Pipeline friendly:** Easy to overlap execution
  - **Example:** `ADD R1, R2, R3` $\rightarrow$ 1 cycle (after pipeline filled)
- **CISC Instructions (x86 example):**
  - **Variable complexity:** Instructions can be 1-15 bytes
  - **Variable timing:** Cycles depend on instruction complexity
  - **Pipeline challenges:** Complex decoding required
  - **Examples:**
    - `ADD EAX, EBX` $\rightarrow$ 1 cycle (simple)
    - `ADD EAX, [EBX + 8]` $\rightarrow$ 3-5 cycles (memory access)
    - `ADD EAX, [EBX + ECX*2 + 16]` $\rightarrow$ 5-8 cycles (complex addressing)

Understanding Threads: Sequential Instruction Streams

- **What is a Thread?**
  - A sequence of instructions to be executed
  - One thread = one sequential pipeline of instructions
  - Each thread has its own program counter and register state
- **Single Thread Execution:**
  - Instructions execute in order: Inst1 $\rightarrow$ Inst2 $\rightarrow$ Inst3 $\rightarrow$ Inst4
  - Each instruction flows through the CPU pipeline
  - One core can execute one thread at a time (traditionally)
- **Thread Example:**
  - **Thread A:** Calculate sum of array elements
  - **Instructions:** LOAD, ADD, LOAD, ADD, LOAD, ADD, STORE
  - **Execution:** Sequential through pipeline stages

Modern CPU Capabilities: Beyond Single Threading

- **Multiple Threads per Core (Hyperthreading/SMT):**
  - **Thread A:** Inst1A $\rightarrow$ Inst2A $\rightarrow$ Inst3A
  - **Thread B:** Inst1B $\rightarrow$ Inst2B $\rightarrow$ Inst3B
  - **Interleaved execution:** CPU switches between threads each cycle
  - Helps utilize pipeline when one thread stalls
- **Superscalar Execution (Multiple Pipelines):**
  - Single thread, but multiple execution units
  - `ADD R1, R2, R3` and `MUL R4, R5, R6` can execute simultaneously
  - No dependency = parallel execution within single thread
- **Multi-Core Systems:**
  - Each core can execute different threads independently
  - 12 cores = up to 12 threads executing truly in parallel
  - With hyperthreading: 12 cores = 24 threads

Real-World Example: Jetson Orin AGX Instruction Execution

- **ARM Cortex-A78 Specifications:**
  - **Clock Speed:** 3.4 GHz (3.4 billion cycles/second)
  - **Pipeline Depth:** 11 stages
  - **Execution Units:** 4 parallel units
  - **Threads per Core:** 1 (no hyperthreading)
- **Single ADD Instruction Performance:**
  - **Latency:** 11 cycles to complete (pipeline depth)
  - **Throughput:** 1 instruction started per cycle
  - **Parallel Execution:** Up to 4 instructions per cycle
  - **Real Performance:** 2-3 instructions per cycle (average)
- **System-Level Performance:**
  - **12 cores $\times$ 3.4 billion cycles $\times$ 2.5 avg IPC = 102 billion instructions/second**
  - Each core handles one instruction stream (thread)
  - Total system can handle 12 independent threads

Key Takeaways: Instructions, Cycles, and Threads

- **Instructions:**
  - Basic CPU operations written in assembly language
  - Translated from high-level programming languages
  - Different types: arithmetic, data movement, control flow
- **Cycles and Timing:**
  - Each instruction takes multiple cycles to complete
  - Pipelining allows high throughput despite multi-cycle latency
  - RISC: predictable timing, CISC: variable timing
- **Threads:**
  - One thread = one sequential stream of instructions
  - Modern CPUs can handle multiple threads per core
  - Multi-core systems enable true parallel thread execution
- **Performance Factors:**
  - Clock speed, pipeline efficiency, instruction-level parallelism
  - Thread management and core utilization
  - Architecture design choices (RISC vs CISC)

Computer Architecture Overview

- **Computer Architecture:** The conceptual design and fundamental operational structure of a computer system.
- **Key Components:**
  - **CPU (Central Processing Unit):** Executes instructions and processes data.
  - **Memory:** Stores data and instructions temporarily (RAM) or permanently (SSD/HDD).
  - **I/O Devices:** Facilitate interaction with the external environment (keyboard, mouse, display).
- **Data Flow:** The CPU fetches instructions from memory, processes them, and may read/write data to/from memory or I/O devices.

Types of architectures

- **Von Neumann Architecture:** Single memory space for instructions and data.
- **Harvard Architecture:** Separate memory spaces for instructions and data.
- **ARM Architecture:** A RISC-based design with a load/store model, unified memory, conditional execution, and deep pipelines; widely used in mobile and embedded systems for its power efficiency.
- **Comparison:** Von Neumann is simpler and more common, Harvard can be faster for certain applications, while ARM combines RISC efficiency with a flexible, unified memory system.

Instruction Set Architectures: RISC vs CISC

- **RISC (Reduced Instruction Set Computer):**
  - Simple, uniform instructions (typically 32-bit)
  - Load/Store architecture - only load/store access memory
  - One instruction per clock cycle (ideally)
  - More registers, simpler hardware
- **CISC (Complex Instruction Set Computer):**
  - Complex, variable-length instructions
  - Instructions can directly access memory
  - Multiple clock cycles per instruction
  - Fewer registers, more complex hardware
- **Philosophy:** RISC favors simple hardware + smart compilers, CISC favors complex hardware + simple compilers

RISC vs CISC: Design Trade-offs

RISC Advantages:

- Simpler processor design
- Lower power consumption
- Better pipelining performance
- Easier to optimize
- Higher clock speeds possible

RISC Disadvantages:

- More instructions needed
- Larger code size
- Complex compiler required

CISC Advantages:

- Fewer instructions needed
- Smaller code size
- Rich instruction set
- Backward compatibility

CISC Disadvantages:

- Complex processor design
- Higher power consumption
- Difficult to pipeline
- Slower clock speeds

ARM Architecture: The RISC Champion

- **ARM (Advanced RISC Machine):** Dominant RISC architecture
  - Founded by Acorn Computers (1985), now ARM Holdings
  - License-based business model - designs sold to manufacturers
  - Powers 95% of smartphones and tablets worldwide
- **Key ARM Characteristics:**
  - **Load/Store Architecture:** Only load/store instructions access memory
  - **Fixed 32-bit Instructions:** Uniform instruction length (ARM64: 64-bit)
  - **Conditional Execution:** Most instructions can be conditionally executed
  - **Low Power Design:** Optimized for battery-powered devices
- **ARM Processor Families:**
  - **Cortex-A:** Application processors (smartphones, tablets)
  - **Cortex-R:** Real-time processors (automotive, industrial)
  - **Cortex-M:** Microcontrollers (IoT, embedded systems)

ARM Instruction Set Example: Load/Store and Arithmetic

- **ARM Assembly Examples:**

```
// Load/Store Operations
LDR R1, [R2]         // Load word from memory[R2] to R1
STR R1, [R2, #4]     // Store R1 to memory[R2 + 4]

// Arithmetic Operations
ADD R1, R2, R3       // R1 = R2 + R3
SUB R1, R2, #5       // R1 = R2 - 5 (immediate value)
```

- **Note:** Load/Store and Arithmetic instructions form the core of data manipulation.

ARM Instruction Set Example: Conditional Execution and Branching

- **ARM Assembly Examples:**

```
// Conditional Execution
ADDEQ R1, R2, R3    // Add only if equal flag set
MOVNE R1, #0        // Move 0 to R1 if not equal

// Branch Instructions
B label             // Unconditional branch
BEQ label           // Branch if equal
BL function         // Branch with link (function call)
```

- **Note:** Use conditional flags and branching for control flow.

ARM's Modern Success: Apple Silicon

- **Apple's ARM Transition:**
  - **M1 Chip (2020):** First ARM-based Mac processor
  - **M1 Pro/Max (2021):** High-performance variants
  - **M2 Series (2022+):** Next generation ARM processors
- **ARM Advantages in Apple Silicon:**
  - **Power Efficiency:** Exceptional battery life in MacBooks
  - **Unified Memory:** CPU and GPU share same memory pool
  - **Custom Silicon:** Apple designs custom ARM cores
  - **Performance:** Competitive with Intel/AMD x86 processors
- **Market Impact:**
  - Proved ARM can compete in laptop/desktop market
  - Microsoft developing ARM-based Windows
  - Amazon's Graviton ARM servers gaining adoption

x86 Architecture: The CISC Powerhouse

- **x86 History:**
  - **Intel 8086 (1978):** Original 16-bit processor
  - **80386 (1985):** First 32-bit x86 processor
  - **x86-64/AMD64 (2003):** 64-bit extension by AMD
  - Dominates desktop, laptop, and server markets
- **x86 CISC Characteristics:**
  - **Variable Instruction Length:** 1 to 15 bytes per instruction
  - **Complex Instructions:** Single instruction can do multiple operations
  - **Memory-to-Memory Operations:** Direct memory manipulation
  - **Rich Addressing Modes:** Multiple ways to specify operands
- **Modern x86 Complexity:**
  - Hundreds of instructions in instruction set
  - Backward compatibility maintained since 8086
  - Internal RISC-like execution (micro-ops)

x86 Instruction Set Example: Memory & Variable-Length Instructions

- **x86 Assembly Examples:**

```
// Complex Memory Operations
ADD [EBX], EAX      // Add EAX to memory[EBX], store in memory
MOV EAX, [EBX+4]    // Load from memory[EBX+4] to EAX

// Variable Length Instructions
MOV AL, 5           // 2 bytes: Move immediate to 8-bit register
MOV EAX, 0x12345678 // 5 bytes: Move 32-bit immediate to register
```

- **Note:** Variable-length encoding ranges from 1 to 15 bytes

x86 Instruction Set Example: Addressing, Strings & Stack

- **x86 Assembly Examples:**

```
// Complex Addressing Modes
MOV EAX, [EBX + ECX*2 + 8]  // EAX = memory[EBX + ECX*2 + 8]

// String Operations
REP MOVSB           // Repeat move string bytes (hardware loop)

// Stack Operations
PUSH EAX            // Push EAX onto stack
CALL function       // Call function (push return address + jump)
```

- **Note:** Complex decoding required for advanced addressing and control instructions

Modern x86: CISC Outside, RISC Inside

- **Micro-Operation Translation:**
  - Complex x86 instructions decoded into simple micro-ops
  - Internal execution core is RISC-like
  - Best of both worlds: CISC compatibility + RISC performance
- **Example Translation:**
  - `ADD [EBX], EAX` becomes:
  - `LOAD temp, [EBX]` (micro-op 1)
  - `ADD temp, EAX` (micro-op 2)
  - `STORE temp, [EBX]` (micro-op 3)
- **Performance Techniques:**
  - **Out-of-Order Execution:** Execute micro-ops as dependencies allow
  - **Superscalar:** Multiple execution units run in parallel
  - **Branch Prediction:** Predict which way branches will go
  - **Speculative Execution:** Execute ahead speculatively

ARM vs x86: Performance and Power Comparison

ARM Strengths:

- **Power Efficiency:** 3-5x better performance per watt
- **Heat Generation:** Runs cooler, enables fanless designs
- **Battery Life:** Exceptional in mobile devices
- **Custom Silicon:** Licensees can customize designs
- **Cost:** Lower licensing and manufacturing costs

x86 Strengths:

- **Raw Performance:** Higher peak performance in many workloads
- **Software Ecosystem:** Decades of optimized software
- **Enterprise Features:** Advanced virtualization, security
- **Backward Compatibility:** Runs legacy software unchanged
- **Manufacturing:** Advanced process nodes (Intel, TSMC)

- **Current Trend:** ARM gaining ground in servers and laptops, x86 still dominant in desktop/enterprise
- **Future:** Likely convergence with both architectures borrowing from each other

Real-World Applications: Choosing the Right Architecture

- **ARM Dominates:**
  - **Mobile Devices:** Smartphones, tablets (95% market share)
  - **IoT/Embedded:** Sensors, smart devices, automotive
  - **Apple Ecosystem:** M1/M2 MacBooks, iPhones, iPads
  - **Cloud Computing:** Amazon Graviton, custom server chips
- **x86 Dominates:**
  - **Desktop/Laptop PCs:** Gaming, productivity, development
  - **Enterprise Servers:** Data centers, high-performance computing
  - **Legacy Systems:** Existing infrastructure and software
  - **High-End Gaming:** Maximum performance requirements
- **Decision Factors:**
  - Power efficiency vs raw performance
  - Software compatibility requirements
  - Cost constraints and development timeline
  - Target market and use case

The Von Neumann Architecture

- **The Von Neumann Architecture:** The core model of a modern computer.
  - Central Processing Unit (CPU): The "brain."
  - Main Memory (RAM): The workspace.
  - Input/Output (I/O) Systems.
- **A Deeper Look at the CPU:**
  - Control Unit (CU), Arithmetic Logic Unit (ALU), Registers.
- **The Memory Hierarchy:** A pyramid of speed, cost, and size.
  - **L1/L2/L3 Cache:** Ultra-fast memory on the CPU.
  - **RAM (Random Access Memory):** Volatile, fast memory for active programs.
  - **Permanent Storage:** Non-volatile, slower storage (SSDs, HDDs).

Real-World Example: NVIDIA Jetson Orin AGX

- **Why Jetson Orin AGX?**
  - High-performance AI computer for edge computing
  - Combines ARM CPU architecture with powerful GPU
  - Perfect example of modern heterogeneous computing
  - Real specifications we can analyze and understand
- **What We'll Learn:**
  - What does "3.4 GHz" actually mean?
  - How memory hierarchy works in practice
  - Understanding core counts and parallel processing
  - Power consumption and thermal design
  - GPU vs CPU architecture differences
- **Context:** Used in autonomous vehicles, robotics, industrial AI, and edge computing applications

Jetson Orin AGX: CPU Specifications Deep Dive

- **CPU Architecture:**
  - **12-core ARM Cortex-A78AE** (ARM v8.2 64-bit architecture)
  - **Base Clock: 2.2 GHz, Boost Clock: 3.4 GHz**
  - **Process Node:** Samsung 8nm (8LPP) technology
- **What does 3.4 GHz mean?**
  - **Clock Speed:** 3.4 billion cycles per second
  - Each cycle can potentially execute one instruction
  - Higher frequency = more operations per second (generally)
  - **Reality:** Modern CPUs execute multiple instructions per cycle
- **12 Cores Explained:**
  - Each core can run independent threads simultaneously
  - Total theoretical capacity: 12×3.4 billion $= 40.8$ billion operations/second
  - Perfect for parallel workloads (AI inference, multimedia processing)

Understanding Clock Speed: The 3.4 GHz Deep Dive

- **Clock Signal Fundamentals:**
  - **Clock Generator:** Creates regular electrical pulses
  - **3.4 GHz = 3,400,000,000 cycles per second**
  - **Clock Period:** $1/3.4$ GHz $= 0.29$ nanoseconds per cycle
- **What Happens in One Clock Cycle?**
  - **Fetch:** Read instruction from memory/cache
  - **Decode:** Understand what the instruction does
  - **Execute:** Perform the operation (add, multiply, etc.)
  - **Writeback:** Store the result
- **Pipeline Efficiency:**
  - Modern CPUs use **pipelining**: overlap instruction stages
  - Multiple instructions in different stages simultaneously
  - **Cortex-A78:** 13-stage pipeline for efficiency
  - Can complete **multiple instructions per cycle** when optimized

Jetson Orin AGX: Memory Hierarchy in Action

- **L1 Cache (Per Core):**
  - **64KB Instruction + 64KB Data cache**
  - **Access Time:** 1-2 clock cycles (0.6-1.2 nanoseconds)
  - **Purpose:** Store most frequently used instructions and data
- **L2 Cache (Per Core):**
  - **1MB per core** (12MB total L2 cache)
  - **Access Time:** 8-12 clock cycles (2.4-3.6 nanoseconds)
  - **Purpose:** Secondary cache for recent data
- **L3 Cache (Shared):**
  - **6MB shared across all cores**
  - **Access Time:** 20-30 clock cycles (6-9 nanoseconds)
  - **Purpose:** Reduce main memory access, improve inter-core communication

Jetson Orin AGX: System Memory and Storage

- **Main Memory (LPDDR5):**
  - **32GB LPDDR5 RAM** (Low Power DDR5)
  - **Memory Bandwidth:** 204.8 GB/s
  - **Access Time:** 60-100 nanoseconds (200-350 clock cycles)
  - **Unified Memory:** CPU and GPU share the same memory pool
- **Storage:**
  - **64GB eUFS 3.1** (embedded Universal Flash Storage)
  - **NVMe SSD support** via M.2 Key M slot
  - **Access Time:** Microseconds (millions of clock cycles)
- **Memory Hierarchy Performance Gap:**
  - **L1 Cache:** 1× (baseline speed)
  - **L2 Cache:** 6× slower than L1
  - **Main Memory:** 100× slower than L1
  - **Storage:** 10,000× slower than L1

GPU Architecture: Parallel Computing Powerhouse

- **Jetson Orin AGX GPU:**
  - **2048 CUDA Cores** (NVIDIA Ampere architecture)
  - **64 Tensor Cores** (4th generation, AI-optimized)
  - **GPU Clock:** Up to 1.3 GHz
  - **AI Performance:** 275 TOPS (Tera Operations Per Second)
- **CPU vs GPU Philosophy:**
  - **CPU:** 12 powerful cores, optimized for sequential tasks
  - **GPU:** 2048 simpler cores, optimized for parallel tasks
  - **CPU Core:** Complex, large, smart (out-of-order execution)
  - **GPU Core:** Simple, small, numerous (in-order execution)
- **Parallel Processing Example:**
  - **Image Processing:** Each pixel processed by different GPU core
  - **AI Inference:** Matrix operations split across hundreds of cores
  - **Speedup:** 10-100$\times$ faster than CPU for parallel workloads

Power and Thermal Management

- **Power Consumption:**
  - **Total System Power:** 15W to 60W (configurable)
  - **Idle Power:** 8W (power-saving features active)
  - **Peak Performance:** 60W (maximum computational load)
  - **Power Efficiency:** 4.6 TOPS/W (AI workloads)
- **Why Power Matters:**
  - **Heat Generation:** Power = Heat (thermodynamics)
  - **Battery Life:** Lower power = longer operation time
  - **Cooling Requirements:** Affects system design and cost
  - **Performance Scaling:** Higher power = higher performance (generally)
- **Thermal Design:**
  - **Operating Temperature:** -25°C to +80°C
  - **Thermal Throttling:** Reduces frequency when too hot
  - **Heat Sink Required:** Passive or active cooling needed

Real-World Performance: What These Specs Mean

- **CPU Performance Examples:**
  - **Web Browsing:** 1-2 cores at 10-30% utilization
  - **Video Encoding:** 8-12 cores at 70-90% utilization
  - **Compilation:** All 12 cores at high utilization
  - **AI Inference:** CPU + GPU collaboration
- **Memory Access Patterns:**
  - **Cache Hit Rate:** 95-99% for optimized applications
  - **Cache Miss Penalty:** 100× performance impact
  - **Memory Bandwidth:** Critical for GPU-accelerated workloads
- **Practical Implications:**
  - **Algorithm Design:** Consider cache-friendly data access
  - **Parallel Programming:** Utilize multiple cores effectively
  - **Memory Management:** Minimize cache misses and memory allocations

Benchmark Comparison: Understanding Relative Performance

- **Jetson Orin AGX vs Other Systems:**
  - **Raspberry Pi 4:** 15× faster (CPU), 100× faster (AI)
  - **Intel i7-12700H Laptop:** 0.8× slower (CPU), competitive (AI)
  - **Desktop RTX 4080:** 0.3× performance, but 4× more power efficient
- **Performance Metrics Explained:**
  - **TOPS (Tera Ops/Sec):** AI/ML operations per second
  - **FLOPS (Floating Point Ops/Sec):** Scientific computing performance
  - **Instructions Per Second:** General computing capability
  - **Memory Bandwidth:** Data transfer capacity
- **Why Comparisons Are Complex:**
  - Different architectures excel at different tasks
  - Software optimization matters significantly
  - Power consumption vs performance trade-offs
  - Use case determines which metrics matter most

Key Takeaways: Understanding Computer Specifications

- **Clock Speed (3.4 GHz):**
  - 3.4 billion cycles per second
  - Not the only performance factor
  - Modern CPUs execute multiple instructions per cycle
- **Core Count (12 cores):**
  - Enables parallel processing
  - More cores = better multitasking and parallel workloads
  - Single-threaded performance still depends on individual core speed
- **Memory Hierarchy:**
  - Cache hits are crucial for performance
  - Memory bandwidth limits parallel processing
  - Unified memory simplifies programming but requires careful management
- **Specialized Processing:**
  - GPUs excel at parallel tasks (AI, graphics, scientific computing)
  - CPUs excel at sequential tasks (general computing, control logic)
  - Modern systems combine both for optimal performance

Key Takeaways

- **Architecture Matters:**
  - RISC vs CISC trade-offs shape modern computing
  - ARM's power efficiency revolutionizing mobile and laptop markets
  - x86's complexity enables high performance in servers and desktops
- **System Design Principles:**
  - Understand your hardware constraints and capabilities
  - Choose the right architecture for your use case
  - Balance performance, power, and cost requirements
- **Future Trends:**
  - ARM expanding into server and desktop markets
  - Heterogeneous computing (CPU + GPU + specialized processors)
  - Quantum and neuromorphic computing on the horizon

Next Steps

- **Hands-On Practice:**
  - Experiment with ARM and x86 assembly language
  - Profile applications to understand performance bottlenecks
  - Design systems with different architectural constraints
- **Further Learning:**
  - Advanced computer architecture courses
  - System design interview preparation
  - Open-source hardware projects (RISC-V)
  - High-performance computing and parallel programming
- **Career Applications:**
  - System architecture roles
  - Performance engineering
  - Embedded systems development
  - Cloud infrastructure design

References and Resources

- **Textbooks:**
  - Computer Organization and Design - Patterson & Hennessy
  - Computer Architecture: A Quantitative Approach - Hennessy & Patterson
  - ARM System Developer's Guide - Sloss, Symes & Wright
- **Online Resources:**
  - ARM Developer Documentation: `https://developer.arm.com`
  - Intel x86 Architecture Manuals: `https://intel.com/sdm`
  - RISC-V Foundation: `https://riscv.org`
- **Tools and Simulators:**
  - QEMU for architecture emulation
  - ARM Development Studio
  - Intel VTune Profiler
  - Online assembly simulators and debuggers

Questions?

Contact Information:
Email: your.email@institution.edu
Office Hours: By appointment

"The best way to learn computer architecture is to build one."
– Anonymous Computer Architect