CO224: Computer Architecture - Complete Lecture Notes

Dr. Isuru Nawinne

Sat 11/15/2025

Contents

Chapter 1

Lecture 1: Computer Abstractions and Technology

1.1 Introduction

This lecture introduces the fundamental concepts of computer system abstractions, exploring the relationship between hardware and software while providing an overview of the CO224 course structure and topics. We examine how computer systems are built as hierarchies of abstractions, each hiding complexity while providing services to the levels above.

1.2 The Big Picture of Computer Systems

1.2.1 Cross-Section of a Computer System (Top to Bottom)

The diagram above illustrates the complete hierarchy from problems and algorithms at the human level, through the compilation toolchain (Compiler/Assembler/Linker), down to the ISA, microarchitecture (RTL), functional units, logic gates, transistors, and finally the silicon substrate. Each colored layer represents a different abstraction level.

1.2.1.1 Human-Related Level (Gray)

- **Problems**: Real-world challenges to be solved
- **Algorithms**: Step-by-step solutions to problems
- **Programming Languages**: Tools to express algorithms

1.2.1.2 System Level (Blue)

- **Compilers**: Translate high-level code to assembly
- **Assemblers**: Convert assembly to machine code
- **Linkers**: Combine programs with libraries
- **Instruction Set Architecture (ISA)**: The hardware-software interface

1.2.1.3 RTL (Register Transfer Level) - Red/Orange

- **Microarchitecture**: The processor's internal organization
- **Functional Units**: Building blocks that perform operations

1.2.1.4 Logic Level (Green)

- **Gate-level circuits**: Digital logic implementations
- **Logic gates**: AND, OR, NAND, NOR, XOR, etc.

1.2.1.5 Circuit Level (Black)

- **Transistors**: BJT, CMOS devices
- **Voltage levels and currents**: Electrical signals

1.2.1.6 Substrate Level (Black)

- **Semiconductors**: Base materials
- **P-type and N-type semiconductors**: Doped materials
- **Electron currents**: Physical phenomena

1.2.2 Purpose of Computer Systems

- Built to solve problems (like any engineering system)
- Process: **Problems → Algorithms → Programs → Machine Code → Execution**
- Each level provides services to the level above
- Each level hides complexity from the level above

1.3 2. Instruction Set Architecture (ISA) - The Key Interface

1.3.1 What is an ISA?

Definition:

- A specification defining what the computer will understand
- Contains a list of basic instructions the processor can execute
- Examples: ARM version 8, MIPS, x86
- The critical interface between hardware and software

1.3.2 Example Instructions in an ISA

- Add two numbers together
- Subtract one number from another
- Multiply two numbers

- Load a number from memory into CPU
- Store a number from CPU into memory
- All basic operations are well-defined in the ISA

1.3.3 Importance of ISA

- Microarchitecture is built to support a specific ISA
- Programs must be written using instructions from the target ISA
- Compilers translate high-level code to ISA instructions
- ISA is the key point combining software with hardware

1.4 3. From Problem to Execution - The Translation Chain

1.4.1 High-Level Process

```
Problem → Algorithm → Programming Language (C, Python, etc.)
↓
Compiler (translates to assembly code)
↓
Assembler (translates to machine code)
↓
Linker (combines with libraries)
↓
Machine Code / Binary Image
↓
Runs on Microarchitecture (CPU)
```

1.4.2 Tool Chain Components

1.4.2.1 Compiler

- **Function**: Converts high-level language to assembly language
- **Complexity**: Complex task requiring optimization
- **Optimizations**: Performance and memory optimizations
- **Example**: ARM GCC compiler for ARM processors

1.4.2.2 Assembler

- **Function**: Converts assembly to machine code
- **Integration**: Built into the tool chain
- **Output**: Produces binary image (ones and zeros)

1.4.2.3 Linker

- **Function**: Combines program with libraries
- **Output**: Creates final executable
- **Process**: Resolves external references

1.4.3 Architecture-Specific Compilation

- If targeting ARM processor: Use ARM toolchain
- If targeting MIPS processor: Use MIPS toolchain
- Machine code is specific to the target ISA
- Cannot run ARM code on MIPS processor directly

1.5 4. Writing Programs at Different Levels

1.5.1 Machine Code (Binary)

Characteristics:

- Ones and zeros
- Directly executable by processor
- Very difficult for humans to write
- Error-prone and time-consuming

1.5.2 Assembly Language

Characteristics:

- Textual representation of machine instructions
- Example: “ADD R1, R2, R3” instead of binary
- One-to-one mapping with machine code
- Easier than machine code but still difficult for large programs
- Used in CO224 labs for ARM assembly programming

1.5.3 High-Level Languages (C, Python, etc.)

Characteristics:

- Easier to write and understand
- Good for large programs and general-purpose applications
- Requires compiler to translate to assembly/machine code
- Provides abstractions hiding hardware details

1.6 5. Microarchitecture Details

1.6.1 What is Microarchitecture?

Definition:

- A digital logic circuit built to support a given ISA
- Processes binary image (machine code)
- Understands meaning of ones and zeros
- Performs operations in actual hardware

1.6.2 Hierarchy of Microarchitecture Components

1.6.2.1 Microarchitecture Level

- Manipulates instructions
- Built using functional units and gate-level logic

1.6.2.2 Functional Units Level

- **Purpose**: Manipulates numbers
- **Examples**:
  - Adders (ripple carry, half adders, full adders)
  - Multiplexers
  - Encoders
  - Decoders
- Built using logic gates

1.6.2.3 Logic Gate Level

- **Purpose**: Manipulates logic levels (1s and 0s, HIGH and LOW)
- **Gates**: AND, OR, NAND, NOR, XOR, NOT
- Built using transistors

1.6.2.4 Transistor Level

- **Purpose**: Manipulates voltages and currents
- **Types**: BJT, CMOS
- Built using semiconductors
- Learned in electronics courses

1.6.2.5 Semiconductor Level

- Deals with electron currents
- P-type and N-type semiconductors
- Combined to create transistors

1.7 6. Abstraction Concept

1.7.1 What is an Abstraction?

Key Principles:

- A conceptual entity hiding internal details
- Provides interface to higher levels
- Hides complexity underneath
- Each level doesn’t worry about details above or below
- Encapsulates details and defines specific characteristics

1.7.2 Hardware Abstraction Hierarchy (Bottom to Top)

1.7.2.1 1. Substrate (Silicon, Germanium)

- Base semiconductor material

1.7.2.2 2. Transistors

- Built using semiconductor substrate
- Deal with voltage levels

1.7.2.3 3. Logic Gates

- Built using transistors
- Deal with logic levels (HIGH/LOW, 1/0)

1.7.2.4 4. Functional Units

- Built using logic gates
- Deal with numbers
- Examples: Adders, multiplexers

1.7.2.5 5. Microarchitecture

- Built using functional units and logic elements
- Deals with instructions
- Understands machine instructions

1.7.3 Software Abstraction Hierarchy (Bottom to Top)

1.7.3.1 1. Machine Instructions (Binary)

- Ones and zeros
- Collection of logic levels
- Executable by microarchitecture

1.7.3.2 2. Assembly Instructions

- Textual representation of machine code
- One-to-one mapping with machine instructions
- Easier for humans to read

1.7.3.3 3. Programs / Source Code

- Written in high-level languages
- Collections of instructions
- Represent algorithms

1.7.3.4 4. Algorithms and Data Structures

- Conceptual entities
- Represent solutions to problems
- Highest level abstraction

1.7.4 Relationships Between Hardware and Software Abstractions

1.7.4.1 Voltage Levels ↔ Logic Levels

- **Logic 1**: Higher voltage range (e.g., 4-5V)
- **Logic 0**: Lower voltage range (e.g., 0-1V)
- Ranges depend on transistor type (TTL vs CMOS)

1.7.4.2 Logic Levels ↔ Numbers

- Numbers represented as strings of binary digits
- Collections of logic levels form numbers

1.7.4.3 Numbers ↔ Instructions

- Instructions represented as binary numbers
- Microarchitecture interprets these numbers

1.7.4.4 Summary of Relationships

- **Transistors** ↔ Voltages (deal with)
- **Logic Gates** ↔ Logic Levels (deal with)
- **Functional Units** ↔ Numbers (deal with)
- **Microarchitecture** ↔ Instructions (understands)

1.7.5 Complete System

- All abstractions together create “the computer”
- Can deconstruct algorithm down to voltage levels
- Can deconstruct microarchitecture down to silicon

- Tight coupling between hardware and software abstractions
- Computer systems are everywhere due to these abstractions

1.8 7. Course Overview

1.8.1 Course Objectives

- Learn what happens to programs after writing them
- Understand how hardware runs/executes programs
- Learn about hardware-software interface (ISA)
- Study performance optimization (hardware and software)

1.8.2 Textbook

- **Title**: "Computer Organization and Design"
- Uploaded on Fields page
- Multiple chapters covered throughout course

1.8.3 Course Topics (Chapter-by-Chapter)

1.8.3.1 Chapter 1: Computer Abstractions and Technology

- Different abstractions (voltages, logic, numbers, instructions)
- Technology evolution and history
- Performance concepts
- Taken by current instructor

1.8.3.2 Chapter 2: Instructions - Language of the Computer

- Detailed study of Instruction Set Architecture
- ARM architecture used as example
- Taken by Professor Raghav

1.8.3.3 Chapter 3: Arithmetic for Computers

- Self-study assignment (not in classes)
- Integer representation
- Floating-point number representation
- Refresher from GP106 course

1.8.3.4 Chapter 4: The Processor (Microarchitecture)

- How processor is organized and built
- How it supports instruction set
- MIPS architecture used as example
- Building actual processor in labs

1.8.3.5 Chapter 5: Memory Hierarchy

- How memory is organized
- Memory's role in processor operation
- Cache memory concepts

1.8.3.6 Chapter 6: Storage and I/O

- Secondary storage (hard disks, etc.)
- Input/Output systems

1.8.3.7 Additional Topics

- Parallel processing
- Multi-core and many-core processors
- Multi-processors

1.8.4 Laboratory Work

1.8.4.1 First Half (~Weeks 1-7)

- Writing assembly programs
- ARM instruction set architecture
- Hands-on ARM assembly coding

1.8.4.2 Second Half (~Week 8 onwards)

- Building own processor/CPU
- Using Verilog HDL (Hardware Description Language)
- **Implementing**:
  - CPU core
  - Memory
  - Assembler
  - Cache memory
- Running assembly programs on self-built processor
- Complete end-to-end system

1.9 8. Performance Theme

1.9.1 Throughout the Course

Performance is a recurring theme that will be touched upon in every topic:

- How efficiently can CPU do things?
- How fast can operations be performed?
- How can performance be improved?
- Hardware-based improvements

- Software-based improvements

1.10 Key Takeaways

1. Computer systems are built as hierarchies of abstractions
2. Each abstraction level hides complexity and provides services to levels above
3. Instruction Set Architecture (ISA) is the critical interface between hardware and software
4. Hardware hierarchy: Substrate → Transistors → Gates → Functional Units → Microarchitecture
5. Software hierarchy: Machine Code → Assembly → Programs → Algorithms
6. Tight coupling exists between hardware and software abstractions
7. Voltages → Logic Levels → Numbers → Instructions (relationships between levels)
8. Covers ISA, microarchitecture, memory hierarchy, and system organization
9. Labs involve ARM assembly programming and building processor using Verilog
10. Understanding the complete system picture is essential for computer engineers
11. All computer systems, regardless of complexity, are built on these fundamental abstractions
12. Performance optimization is a central theme throughout the course

---

1.11 Summary

Computer systems represent one of the most sophisticated examples of hierarchical abstraction in engineering. From the physical movement of electrons in semiconductors to high-level programming languages, each layer builds upon and hides the complexity of the layers below. The Instruction Set Architecture serves as the critical bridge between hardware and software, enabling programmers to write code without worrying about transistor-level details while allowing hardware designers to optimize implementations without breaking software compatibility.

Throughout this course, we will explore these abstractions in depth, learning not just what they are, but why they exist and how they enable the remarkable computing capabilities we rely on every day. By understanding both hardware and software perspectives, computer engineers gain the ability to design, optimize, and innovate across the entire computing stack.

Chapter 2

Lecture 2: Technology Trends, Moore's Law, and Computer System Organization

2.1 Introduction

The evolution of computer technology over the past 50 years has been nothing short of revolutionary. From room-sized scientific calculators to powerful smartphones in our pockets, this transformation has been guided by a prediction made by Intel co-founder Gordon Moore. This lecture examines the technological trends that enabled this revolution, the physical limitations that eventually constrained traditional scaling approaches, and the architectural innovations that emerged in response.

We will trace the exponential growth in transistor density, explore how smaller feature sizes enabled both more complex circuits and faster operation, understand why clock frequencies stopped increasing around 2004, and see how the industry pivoted to multi-core architectures. Finally, we'll examine how computer systems are organized into three layers (hardware, system software, and application software) and follow the complete translation process from high-level code to binary execution.

2.2 1. Moore's Law - Foundation of Computer Technology Evolution

2.2.1 1.1 Who Was Gordon Moore?

Background and Influence:

- Co-founder of Intel Corporation
- Intel historically the biggest manufacturer of computer chips/processors
- Most personal computers and high-end servers use Intel processors
- Made a prediction that shaped the entire semiconductor industry

Intel's Dominance:

- Established industry standards for processor design
- Set pace for computational advancement
- Influenced competing manufacturers
- Created benchmark for technology expectations

2.2.2 1.2 Moore's Law Definition

The Prediction:

Moore's Law is NOT a physical law like the law of gravity. It is an observation and prediction about technology trends:

> **"The number of transistors that can be placed on a standard computer chip will double every two years."**

Practical Interpretation:

- Roughly translates to: Computational power doubles every two years
- Started in the 1950s and held true for many decades
- Based on continuous demand for increasing computational power
- Self-fulfilling prophecy driven by industry investment

Historical Context:

- Initial observation made in mid-1960s
- Revised and refined over subsequent decades
- Became guiding principle for semiconductor industry
- Influenced research priorities and manufacturing investments

2.2.3 1.3 Impact of Moore's Law

Computer Evolution Enabled:

Computers transformed from room-sized scientific calculators to:

- **Personal Computers:** Desktop and laptop systems in every home
- **Mobile Devices:** Smartphones with computational power exceeding 1990s supercomputers
- **Embedded Systems:** Computational intelligence in everyday objects
- **Wearables:** Smartwatches and fitness trackers

Revolutionary Applications:

Moore's Law made computationally intensive applications possible:

1. **Human Genome Decoding:**
    - Massive computational requirements
    - Processing billions of genetic sequences
    - Pattern recognition across enormous datasets
2. **World Wide Web and Internet Search:**

- Millisecond response times for complex queries
- Indexing billions of web pages
- Real-time information retrieval

3. **Artificial Intelligence and Machine Learning:**
    - Neural networks with billions of parameters
    - Real-time image and speech recognition
    - Autonomous systems and decision-making
4. **Complex Simulations and Scientific Computing:**
    - Weather prediction and climate modeling
    - Molecular dynamics simulations
    - Astrophysical calculations

Societal Impact:

- Computer software became ubiquitous and unavoidable
- Changed how we work, communicate, and learn
- Enabled digital transformation of industries
- Created new fields and destroyed old ones

2.3 2. Technology Scaling - Historical Data

2.3.1 2.1 Transistor Count Growth (1970-2010)

Chart Analysis:

The historical data shows remarkable consistency with Moore's prediction:

- **Vertical Axis:** Number of transistors (10^5 to 10^9 - millions to billions)
- **Horizontal Axis:** Time period (1970 to 2010)
- **Blue Dotted Line:** Doubling every 18 months (aggressive prediction)
- **Red-Brown Line:** Doubling every 24 months (Moore's actual prediction)

Real Intel Processor Models:

Tracking actual transistor counts across processor generations:

- **Early Processors:** 4004, 8008, 8080 (thousands of transistors)
- **1989 Milestone - 8086:** Crossed 1 million transistors
- **Middle Era:** Pentium and Itanium series
- **2008 Achievement:** Crossed 1 billion transistors (quad-core processors)
- **Trend Validation:** Actual counts closely followed the "doubling per 2 years" curve

Significance:

- Prediction held remarkably accurate for 40+ years
- Enabled long-term planning for semiconductor industry
- Guided investment in manufacturing technology
- Set performance expectations for consumers

2.3.2 2.2 The x86 Architecture

Origin and Naming:

- **8086 Processor:** First x86 architecture processor (notice "86" in the name)
- Established instruction set architecture (ISA) standard
- Created foundation for backward compatibility

x86 Architecture Family:

The architecture evolved through multiple generations while maintaining compatibility:

- **80286:** Enhanced memory management
- **80386:** True 32-bit processor (author's first computer in 1993, 16 MHz)
- **80486:** Integrated floating-point unit
- **Pentium Series:** Brand name change, performance leap
- **Modern Processors:** Core i3, i5, i7, i9 series

AMD's Adoption:

- AMD also uses x86 architecture
- Compatible instruction set
- Competitive alternative to Intel
- Drives innovation through competition

Evolution Strategy:

- Architecture evolved significantly over decades
- Maintained backward compatibility throughout
- Old programs run on new processors
- Balanced innovation with stability

2.3.3 2.3 Historical Context

Early Computing Era (1985-1990):

- 1985: 80386 computers first arrived on market
- No graphical user interfaces (GUIs) existed
- Black screen with text-only displays
- DOS operating system (text-based console)
- Command-line interaction only

Transformation Period (Mid-Late 1990s):

- GUIs emerged (Windows 95 and similar systems)
- Point-and-click interfaces replaced command lines
- Multimedia capabilities became standard
- Internet connectivity became widespread

User Experience Revolution:

- Significant transformation in how people interacted with computers
- Democratized computing beyond technical experts
- Enabled productivity for non-technical users
- Set expectations for modern computing

2.4 3. Feature Size Scaling - Lithography Improvements

2.4.1 3.1 What Made Transistor Count Increase Possible?

The Answer: Smaller Transistors

The exponential growth in transistor count was enabled primarily by reducing transistor size through improved manufacturing processes.

Lithography Process:

- Etching transistors onto silicon wafer using photolithographic techniques
- Patterns created using light masks and photosensitive materials
- Feature size: Measure of transistor dimensions in nanometers (nm)
- Smaller features = more transistors per unit area

Feature Size Timeline:

The relentless march toward smaller dimensions:

- **2004:** 90 nanometer manufacturing process
- **2006:** 65 nanometer
- **2008:** 45 nanometer (very famous generation, many developments)
- **Continuing:** 32 nm, 22 nm processes
- **Current (2021):** 7 nanometer and 5 nanometer in production
- **Future Roadmap:** 3 nanometer, 2 nanometer, 1 nanometer under discussion

2.4.2 3.2 What is “Feature Size”?

Original Definition:

- Originally represented physical measurement: minimum distance between source and drain of transistor
- Also called channel width, gate size, or half-pitch
- Directly related to transistor dimensions

Modern Reality:

- **NOT a precisely defined physical measurement anymore**
- More of a **marketing term** in current usage
- General measure of manufacturing process advancement
- Smaller number suggests more advanced technology

Alternative Names:

Different terms referring to approximately the same concept:

- Gate size
- Channel width
- Half-pitch

- Process node
- Technology node

Why Ambiguity Developed:

- Manufacturing processes became more complex
- Multiple dimensions define transistor performance
- 3D structures don't have simple linear measurements
- Marketing convenience over physical precision

2.4.3 3.3 How Tiny Are Transistors?

Mind-Boggling Scale:

Putting modern transistor sizes in perspective:

- **45 nanometer technology:** Can fit **30 million transistors** on the head of a pin
- **Across human hair:** Over **1,000 transistors** fit across the width of a single human hair
- **Comparison to past:** Incredibly small compared to transistors 40-50 years ago

Manufacturing Precision:

- Requires cleanroom environments cleaner than surgical suites
- Dust particle can destroy multiple chips
- Atomic-level precision required
- Remarkable engineering achievement

2.4.4 3.4 Transistor Structure

Basic Components:

- **Silicon Substrate:** Base semiconductor material
- **Source and Drain:** Two metal contacts on either side
- **Gate:** Control electrode positioned between source and drain
- **Insulator:** Separates gate from channel

Feature Size Definition:

- Distance between drain and source (channel width)
- Critical dimension for transistor operation
- Determines electrical characteristics

Electrical Properties:

- **Capacitance Load:** Inherent property based on semiconductor material and structure
- Affects switching speed and power consumption
- Function of transistor geometry and materials
- Critical parameter for circuit performance

2.5 4. Technology Roadmaps - ITRS Predictions

2.5.1 4.1 ITRS Organization

International Technology Roadmap for Semiconductors:

- **Established:** Around 2001
- **Purpose:** Predict feature size scaling for next 10 years
- **Membership:** Major semiconductor manufacturers and research institutions
- **Methodology:** Based on technology capabilities and market demand

Prediction Basis:

The roadmaps considered multiple factors:

- Demand for computational power
- Available manufacturing technology
- Potential technological improvements
- Economic feasibility
- Physical limitations

Regular Updates:

- Produced updated roadmaps regularly
- Adjusted predictions based on actual progress
- Guided industry research priorities
- Dissolved in 2015 due to paradigm shift

2.5.2 4.2 Original Roadmap (2001)

Optimistic Projections:

The initial roadmap predicted steady exponential decrease in feature size:

- **2001 Baseline:** 130 nm technology in production
- **2006 Target:** 65 nm
- **2008 Target:** 45 nm
- **2012 Projection:** Continuing decrease following Moore's Law

Assumptions:

- Linear continuation of historical trends
- Traditional planar transistor scaling
- Continued improvements in lithography
- Economic sustainability of smaller features

2.5.3 4.3 Revised Roadmap (2013)

Adjusted Expectations:

By 2013, reality required revised predictions:

- **2013 Actual:** 22 nm achieved
- **2015 Target:** 16 nm predicted

- **2019 Target:** 12 nm
- **2023 Target:** 7 nm
- **2028 Target:** 5 nm

Key Observation:

- **Rate of reduction slowed down** compared to original predictions
- Still following exponential trend but slower pace
- Physical and economic challenges becoming apparent
- Need for alternative approaches emerging

2.5.4 4.4 Final Roadmap (2015)

Dramatic Shift in Direction:

The 2015 roadmap marked a fundamental change:

- **2015 Status:** Still around 25-24 nm (behind 2013 predictions)
- **Near-term Projection:** Fast improvements predicted to reach 10 nm
- **2021 Target:** 10 nm technology
- **Long-term Direction:** Feature size would **NOT decrease further beyond 10 nm**
- **Plateau:** Would stick with 10 nm for foreseeable future

Significance:

- Sudden departure from decades of continuous scaling
- Recognition of fundamental physical limits
- Industry acknowledgment of new paradigm
- End of traditional Moore's Law scaling

2.5.5 4.5 Why the Change? - 3D Technology

Major Paradigm Shift (2013-2015):

The industry pivoted to a fundamentally different approach:

Traditional Approach (Before):

- Single layer of transistors on silicon surface
- Scaling by making transistors smaller
- Two-dimensional planar structures

New Approach (After):

- **3D Chips:** Multiple layers of transistors stacked vertically
- **3D FinFET Technology:** Transistor fins extending upward from surface
- **Vertical Integration:** Third dimension for density increase

Impact on Moore's Law:

- Transistor count **still increasing** (Moore's Law continues)
- But **NOT by making individual transistors smaller**
- Instead: **Stacking transistors on top of each other**

- Adds thickness dimension to chip design

Technical Innovations:

- Gate-all-around (GAA) transistors
- Through-silicon vias (TSVs) for vertical connections
- Advanced packaging techniques
- Thermal management solutions

2.5.6 4.6 Dissolution of ITRS (2015)

Reasons for Dissolution:

- **Technology Divergence:** Multiple paths to increase transistor density
- **End of Simple Scaling:** No longer just reducing feature size
- **3D Stacking:** Fundamentally different approach
- **Heterogeneous Integration:** Combining different technologies on same chip

Multiple Methods for Transistor Density:

Modern approaches include:

- 3D stacking of transistor layers
- FinFET and GAA transistor structures
- Chiplet architectures
- Advanced packaging technologies
- Heterogeneous integration

Moore's Law Status:

- Transistor count **still doubling every 2 years** (as of 2021)
- But through **different means** than traditional scaling
- More complex and diverse strategies
- Higher costs per transistor (economic Moore's Law ending)

2.6 5. Why Smaller Transistors Improve Performance

2.6.1 5.1 Reason 1: More Complex Circuits

Increased Transistor Budget:

More transistors available on chip enables more sophisticated functionality.

Comparison Example:

Limited Transistor Count (100 transistors):

- Can only build simple functional units
- Complex tasks must be broken down into simple operations
- Must use simple functional units repeatedly
- Sequential processing of sub-tasks
- **Result: SLOWER overall execution**

Abundant Transistors (1 billion):

- Can build extremely complex circuits
- Perform complex operations in single step
- Don't need to decompose into simple operations
- Dedicated hardware for sophisticated functions
- **Result: FASTER overall execution**

Architectural Implications:

- Larger caches for better hit rates
- More sophisticated branch predictors
- Wider execution units (SIMD)
- More parallel functional units
- Hardware accelerators for specific tasks

2.6.2 5.2 Reason 2: Faster Switching

Electrical Advantages of Smaller Size:

Smaller transistors possess superior electrical characteristics:

Lower Operating Voltage:

- Smaller channel width requires less voltage to switch transistor
- Voltage scaling: From ~5V (1980s) to ~1V (modern)
- Reduces power consumption
- Enables higher switching frequencies

Reduced Impedance:

- Lower resistance in transistor channel
- Faster current flow
- Quicker charging/discharging of capacitances

Faster State Changes:

- Can switch transistor on/off faster
- Less time needed for signal propagation
- Shorter gate delays

Overall Impact:

- Faster transistor switching → Higher possible clock rate
- Higher clock rate → More operations per second
- Faster overall computation

Physical Explanation:

The relationship between size and speed involves:

- Reduced gate capacitance (smaller area)
- Shorter carrier transit time (shorter channel)
- Lower RC time constants
- Improved frequency response

2.7 6. Clock Rate Trends - The Power Wall

2.7.1 6.1 Clock Rate Increases (1982-2004)

Exponential Growth Era:

Processor clock frequencies increased dramatically for over two decades:

Historical Progression:

- **286 (1982):** 12.5 MHz
- **386 (1985):** 16 MHz (author's first computer)
- **486 (Early 1990s):** 25-33 MHz
- **Pentium (Mid-1990s):** 60-200 MHz
- **Pentium 4 (2001):** 2 GHz (2000 MHz) - **First to break 2 GHz barrier**
- **Pentium 4 Prescott (2004):** 3.6 GHz (3600 MHz) - **Peak of single-core era**

Growth Rate:

- Nearly 300× increase in 20 years
- Roughly doubled every 18-24 months
- Parallel to Moore's Law for transistor count
- Consumer expectation of continuous frequency increases

2.7.2 6.2 The Turning Point (2004-2007)

Sudden Deceleration:

Around 2004, the decades-long trend dramatically changed:

- Clock rate increase **slowed dramatically**
- Reached peak around **3.6-4 GHz**
- Settled and plateaued at that level
- **Despite transistors continuing to get smaller**

The Paradox:

- Manufacturing processes still improving
- More transistors available
- Smaller, potentially faster transistors
- **But clock frequencies stopped increasing**

Industry Recognition:

- Fundamental limitation encountered
- Alternative approaches needed
- Architectural innovation required
- End of "free" performance scaling

2.7.3 6.3 The Power Wall Problem

Power Consumption Growth Crisis:

As clock rates increased, power consumption grew unsustainably:

Pentium 4 Prescott Example:

- Required more than **100 watts** of power
- Power supply could provide the necessary electrical power
- **But HEAT became the critical limiting issue**

The Thermal Crisis:

Physical reality of heat generation:

1. **Heat Generation Mechanism:**
    - Billions of transistors switching billions of times per second
    - Each switching event involves current flow
    - Current through resistance generates heat (I^2R losses)
    - Accumulated heat from all transistors
2. **Heat Dissipation Challenge:**
    - Heat generation outpaced heat removal capability
    - Chips would overheat and potentially burn
    - Thermal damage to silicon
    - Reliability concerns and failure modes

The 100-Watt Rule of Thumb:

Industry consensus emerged:

- **Maximum practical limit: ~100 watts per chip**
- Cooling solutions couldn't effectively handle more
- Would not cross that boundary for desktop processors
- Required alternative approaches to improve performance

Attempted Solutions (All Insufficient):

Various cooling methods were tried:

- **Improved Air Cooling:**
    - Larger heatsinks
    - More powerful fans
    - Better thermal interface materials
- **Liquid Cooling:**
    - Water cooling systems (like car radiators)
    - More efficient heat transfer
    - Complex and expensive
- **Exotic Solutions:**
    - Phase-change cooling

  - Thermoelectric coolers
  - Ultimately impractical for consumer systems

None Sufficient:

- Couldn't overcome fundamental heat generation problem
- Cost and complexity prohibitive
- Reliability concerns
- Not scalable to mass market

2.7.4 6.4 Dynamic Power Equation

The Physics of Power Consumption:

Dynamic power consumption follows this relationship:

```
Power = Capacitance Load × Voltage² × Frequency
```

Factor Analysis (1982-2004):

Capacitance Load:

- **Relatively Constant** per transistor
- Inherent to transistor structure and materials
- Determined by semiconductor physics
- Cannot be arbitrarily reduced

Voltage Reduction:

- Decreased from **~5V to ~1V**
- **5× voltage reduction**
- Squared effect: **25× power reduction** contribution
- Significant mitigation strategy

Frequency Increase:

- Increased **~300× (12 MHz to 3600 MHz)**
- Direct linear effect on power
- **300× power increase** contribution
- Overwhelmed voltage reduction benefits

Net Effect Calculation:

```
Power Scaling = (Capacitance) × (Voltage²) × (Frequency)
              = (1×) × (1/5)² × (300×)
              = (1×) × (1/25) × (300×)
              = 12× power increase
```

Key Insight:

- Despite aggressive voltage scaling (25× reduction in V^2 term)
- Frequency increase (300×) overwhelmed the benefit
- Net result: **Massive power increase**
- Power grew faster than could be managed thermally
- Fundamental limitation reached

Why Voltage Couldn't Scale Further:

- Transistor threshold voltages have physical limits
- Signal-to-noise ratio requirements
- Reliability constraints
- Leakage current increases at lower voltages

2.7.5 6.5 Overclocking Phenomenon

Marketing and User Community Response:

Emerged prominently around early 2000s during the MHz wars:

Manufacturer Approach:

- **"Official" Specifications:** Conservative clock speed (e.g., 3.6 GHz)
- **Actual Capability:** Could run at higher speeds without guarantees
- **Marketing Tactic:** Appeal to gamers and power users
- **Risk Disclaimer:** No warranty at higher speeds

User Overclocking:

Users could manually increase clock speed beyond rated specification:

Process:

- Change BIOS/UEFI settings
- Increase multiplier or bus speed
- Often increase voltage
- Improve cooling solutions

Risks:

- **Generate More Heat:** Exceed thermal design power (TDP)
- **Potential Damage:** Could permanently destroy processor
- **Instability:** System crashes and data corruption
- **Reduced Lifespan:** Accelerated aging of components
- **Voided Warranty:** No manufacturer support

Target Audience:

- **Gamers:** Seeking maximum frame rates
- **Enthusiasts:** Hobbyists and competitors
- **Overclockers:** Specialized community
- **Benchmarkers:** Competitive performance testing

Industry Impact:

- Created enthusiast market segment
- Influenced product differentiation (K-series Intel chips)
- Added revenue from premium products
- Many processors destroyed but market remained

2.8 7. Shift to Multi-Core Processors

2.8.1 7.1 The Challenge

The Industry Dilemma:

By mid-2000s, the semiconductor industry faced a paradox:

Available Resources:

- Moore's Law still valid: More transistors available every generation
- Manufacturing processes continuing to improve
- Silicon area increasing or transistor density growing

Constraints:

- **Cannot use all transistors simultaneously** (power wall/heat problem)
- **Cannot increase clock rate** (thermal limitations)
- Traditional performance scaling broken

Critical Questions:

- How to utilize available transistors?
- How to continue improving computational power?
- How to maintain Moore's Law performance benefits?

2.8.2 7.2 Solution: Multiple Processor Cores

Paradigm Shift (2004-2008):

Industry pivoted from single-core to multi-core architectures:

Core Concept:

Instead of one powerful processor, put **multiple complete processors on same chip**:

- Each core is a complete CPU
- Cores share cache and memory interface
- Can execute different programs simultaneously
- Parallel execution at thread/process level

Early Multi-Core Processors:

AMD Barcelona (2007):

- **4 cores** on single die
- Shared L3 cache
- Integrated memory controller

Intel Core Series:

- Multiple models with **4 cores**
- Hyperthreading technology (2 threads per core)
- Competitive performance

IBM Processors:

- Server-oriented multi-core designs
- High core counts for enterprise
- Power-efficient designs

Extreme Designs:

- Some manufacturers: **8 cores** per chip
- Specialized server processors with more
- Graphics processors (GPUs) with hundreds of simple cores

Power Management:

- **Dynamic Power Allocation:** Cores powered on/off as needed
- **Turbo Boost:** Temporarily increase frequency of active cores
- **Per-Core Voltage/Frequency Scaling:** Independent control
- **Power Gating:** Completely shut down unused cores
- **Thermal Management:** Distribute heat across die

2.8.3 7.3 The Plan

Initial Industry Vision:

Following Moore's Law principle for core counts:

Projected Growth:

- **Every 2 years:** Double the number of cores per chip
- Use increased transistor budget for more cores
- Each generation: 2× cores, same power envelope

Timeline Projection:

- **2006:** 4 cores
- **2008:** 8 cores
- **2010:** 16 cores
- **2012:** 32 cores
- **2014:** 64 cores
- **By 2021:** Should have **hundreds of cores** in consumer processors

Theoretical Benefits:

- Continuous performance improvement
- Utilizing Moore's Law transistor growth
- Working within power constraints
- Parallel computing becoming mainstream

Reality Check:

- **This did NOT happen**
- Current consumer chips: Typically **4-16 cores** (2021)
- Server processors: Up to 64-128 cores
- Not the hundreds predicted
- Growth much slower than initial projections

2.8.4 7.4 Why Multi-Core Growth Slowed

The Fundamental Problem: Parallel Programming Difficulty

Software Challenge:

Multi-core processors require fundamentally different programming approach:

Sequential Programming (Traditional):

- Single thread of execution
- One operation after another
- Natural mental model
- Straightforward debugging
- Predictable behavior

Parallel Programming (Required for Multi-Core):

- Multiple simultaneous threads of execution
- Programmer must **explicitly** write code for multiple processors
- Must think: "I'm writing for 4, 8, or 16 processors"
- Coordinate and synchronize multiple processes/threads
- Manage shared resources and data

Available Parallel Programming Techniques:

Multi-Threading:

- **POSIX Threads (pthreads)** in C/C++
- Java threading primitives
- Python threading/multiprocessing
- Operating system thread scheduling

Multiple Processes:

- Fork/join models
- Message passing (MPI for scientific computing)
- Process pools

Communication Mechanisms:

- Shared memory
- Message queues
- Pipes and sockets
- Synchronization primitives (mutexes, semaphores, barriers)

Language Support:

- Available in most major programming languages
- Library support varies in quality
- Language-level primitives vs library-based approaches

Inherent Difficulties:

1. Parallel Programming is HARD:

- Much more difficult than sequential programming

- Different mental model required
- Non-deterministic behavior
- Difficult to reproduce bugs
- Race conditions and deadlocks

2. Requires Deep Understanding:

- **Hardware Architecture:** How cores communicate, cache coherency
- **Processor Organization:** Memory hierarchy, interconnects
- **Communication Overhead:** Cost of data transfer between cores
- **Synchronization Overhead:** Cost of coordinating execution

Key Technical Challenges:

Load Balancing:

- **Problem:** Distribute work evenly across all cores
- **Bad Scenario:** One processor idle while another overloaded
- **Requirement:** Dynamic or static work distribution
- **Complexity:** Workload often unknown until runtime
- **Solution Difficulty:** NP-hard problem in general case

Communication Optimization:

- **Problem:** Minimize data transfer between cores
- **Reality:** Communication takes time (overhead)
- **Amdahl's Law:** Communication is sequential bottleneck
- **Cache Coherency:** Hardware protocol overhead
- **Solution:** Locality-aware algorithms, minimize sharing

Synchronization:

- **Problem:** Coordinate execution between cores
- **Bad Scenario:** One thread waiting indefinitely for another
- **Overhead:** Synchronization primitives have cost
- **Deadlock Risk:** Circular dependencies can halt system
- **Solution:** Careful design, lock-free algorithms

Performance Consequences:

If parallel programming not done well:

- **Wasting Available Hardware:** Cores sitting idle
- **No Performance Gain:** Sequential sections dominate
- **Worse Performance:** Overhead exceeds benefits
- **Unpredictable Results:** Race conditions cause incorrect output

2.8.5 7.5 Instruction-Level Parallelism vs Multi-Core Parallelism

Instruction-Level Parallelism (ILP):

Characteristics:

- **Hardware-Based Solution:** Processor automatically finds parallelism
- **Automatic Execution:** Fetches multiple instructions simultaneously

- **Out-of-Order Execution:** Reorders for efficiency
- **Compiler Support:** Helps but not required
- **Transparent to Programmer:** No special code needed
- **Automatic and Hidden:** Works without programmer awareness

Techniques:

- Superscalar execution
- Out-of-order execution
- Register renaming
- Speculative execution
- Branch prediction

Benefits:

- Free performance improvement
- Works on existing sequential code
- No programmer burden
- Automatic optimization

Multi-Core Parallelism:

Characteristics:

- **Explicit Programming Required:** Programmer must manually parallelize
- **Not Automatic:** No hardware magic
- **Much More Difficult:** Requires expertise
- **Programmer Responsibility:** Must handle all coordination

Programmer Must:

- Break program into parallel threads
- Distribute work across cores
- Handle inter-core communication
- Manage synchronization
- Deal with race conditions
- Avoid deadlocks
- Balance load
- Minimize communication overhead

Contrast:

Aspect	ILP	Multi-Core
Who does work	Hardware	Programmer
Transparency	Invisible	Explicit
Difficulty	Automatic	Hard
Applicability	All code	Limited patterns
Overhead	Hidden	Must manage

2.8.6 7.6 Impact on Software Development

For Regular Programmers:

- **Too Difficult:** Most cannot effectively parallelize
- **Not Worth Effort:** For many applications
- **Sequential Sufficient:** Many programs don’t need parallel performance
- **Training Gap:** Most programmers not trained in parallel programming

For Computer Engineers:

- **Essential Skill:** Must learn parallel programming
- **Career Requirement:** High-performance computing demands it
- **Necessary Understanding:** Must understand hardware deeply
- **Specialized Constructs:** Must master threading, synchronization
- **Architecture Knowledge:** Must understand cache coherency, memory models

Application Domains:

High-Performance Applications Requiring Parallelism:

- Scientific computing and simulations
- Video encoding/decoding
- Machine learning training
- Real-time graphics rendering
- Big data processing
- Financial modeling

Applications That Remain Sequential:

- Many business applications
- Simple utilities
- I/O-bound programs
- Interactive applications
- Legacy software

Education Impact:

- Computer science curricula adding parallel programming courses
- Need for hardware architecture understanding
- Gap between industry needs and graduate preparation
- Specialized training for HPC (high-performance computing)

2.9 8. Computer System Organization - Three Layers

2.9.1 8.1 Hardware Layer (Bottom)

Physical Components:

Processor (CPU):

- Central Processing Unit
- Executes machine instructions
- Contains control and datapath
- Includes registers and functional units

Microarchitecture:

- Internal organization of processor
- Pipeline structure
- Execution units
- Cache organization
- Bus interfaces

Memory Hierarchy:

- **Level 1 Cache (L1):**
  - Smallest, fastest
  - Separate instruction and data caches
  - On-core, immediate access
  - Typically 32-64 KB per core
- **Level 2 Cache (L2):**
  - Larger, slightly slower
  - May be shared or per-core
  - Typically 256 KB - 1 MB per core
- **Level 3 Cache (L3):**
  - Largest, slower than L2
  - Shared across all cores
  - Typically several MB
- **Main Memory (RAM):**
  - Dynamic RAM (DRAM)
  - Several GB capacity
  - Much slower than cache
  - Volatile storage

Input/Output Controllers:

- USB controllers
- Network interfaces
- Display adapters
- Storage controllers

Secondary Storage Interfaces:

- SATA for hard drives/SSDs
- NVMe for fast SSDs
- External storage connections

Purpose:

- Actual physical components that execute computation
- Store and retrieve data
- Interact with peripherals and external world
- Provide computational substrate

2.9.2 8.2 System Software Layer (Middle)

Tool Chain Components:

Compiler:

- **Function:** Translates high-level language to assembly
- **Input:** Source code (C, Java, Python, etc.)
- **Output:** Assembly language or intermediate representation
- **Optimization:** Improves performance, reduces size
- **Examples:** GCC, Clang, MSVC, Javac

Assembler:

- **Function:** Translates assembly to machine code
- **Input:** Assembly language (human-readable mnemonics)
- **Output:** Object files (binary machine code)
- **Tasks:** Symbol resolution, address assignment
- **Examples:** GNU Assembler (as), NASM

Linker:

- **Function:** Combines object files and libraries
- **Tasks:** Resolves external references, creates executable
- **Output:** Complete executable program
- **Link Types:** Static linking, dynamic linking
- **Examples:** GNU ld, MSVC linker

Purpose of Tool Chain:

- Support application development
- Bridge high-level abstractions to machine code
- Enable programmer productivity
- Provide optimization opportunities

Operating System:

Core Responsibilities:

Resource Management:

- CPU time allocation
- Memory space allocation
- I/O device arbitration
- Storage space management

Memory Management:

- Virtual memory implementation
- Page tables and address translation
- Memory protection between processes
- Swap space management

Storage Management:

- File system implementation

- Directory structures
- File permissions and security
- Disk block allocation

Input/Output Handling:

- Device drivers
- Interrupt handling
- Buffering and caching
- Asynchronous I/O

Task Scheduling:

- Process scheduling algorithms
- Thread scheduling
- Priority management
- Time-slicing and preemption

Resource Sharing:

- Prevents conflicts between programs
- Enforces isolation
- Provides controlled sharing mechanisms

Why Operating System Needed:

Trust and Security:

- **Cannot trust application software**
- Programs can be malicious or buggy
- Programs don't consider other programs
- Need supervision and enforcement

Coordination and Protection:

- Prevents programs from breaking hardware
- Enforces rules set by hardware (privileged instructions)
- Provides abstraction hiding hardware details
- Mediates access to shared resources

Programmer Benefits:

Abstractions Provided:

Programmers don't need to worry about:

- Where program code resides in physical memory
- Where variables are stored in RAM
- Hardware resource conflicts
- Direct hardware access
- Physical device characteristics

OS Guarantees:

- Safe hardware usage
- Process isolation
- Consistent interfaces

- Reliable file storage
- Network communication

Example Services:

- File I/O without knowing disk geometry
- Memory allocation without physical addresses
- Network communication without protocol details
- Device I/O without hardware specifics

2.9.3 8.3 Application Software Layer (Top)

User-Level Programs:

- Programs written by application programmers
- Solve specific problems or provide services
- Interact with users
- Implement business logic

High-Level Programming Languages:

Popular Languages:

- **C:** Systems programming, performance-critical
- **Java:** Enterprise applications, portability
- **Python:** Scripting, data science, machine learning
- **R:** Statistical analysis, data science
- **JavaScript:** Web development, client-side
- **C++:** Performance with abstraction
- **Go:** Concurrent systems, cloud services
- **Rust:** Systems programming, memory safety

Language Characteristics:

Hundreds/Thousands Available:

- Each optimized for specific application domains
- Different paradigms (imperative, functional, object-oriented)
- Trade-offs between performance and productivity
- Community and ecosystem considerations

Domain Optimization:

- **Machine Learning:** Python (NumPy, TensorFlow, PyTorch), R
- **Systems Programming:** C, C++, Rust
- **Enterprise Applications:** Java, C#
- **Web Development:** JavaScript, TypeScript, PHP, Ruby
- **Scientific Computing:** Python, Julia, MATLAB, Fortran
- **Mobile Development:** Swift, Kotlin, Java
- **Game Development:** C++, C#

Level of Abstraction:

- Represents algorithms and solutions to problems

- Closest to problem domain
- Furthest from hardware details
- Highest productivity for programmers
- Requires compilation/interpretation to execute

2.10 9. From High-Level Code to Machine Code - The Translation Process

2.10.1 9.1 Example: Swap Function in C

Source Code:

```
void swap(int v[], int k) {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Function Purpose:

- **Operation:** Swap two values in array
- **Parameters:**
  - `v[]`: Array pointer (base address)
  - `k`: Index of first element to swap
- **Elements Swapped:** Positions k and k+1
- **Method:** Uses temporary variable
- **Simplicity:** Basic operation used frequently in sorting algorithms

Algorithm:

1. Store v[k] in temporary variable
2. Copy v[k+1] to v[k]
3. Copy temporary to v[k+1]

2.10.2 9.2 After Compilation - MIPS Assembly Code

Assembly Translation:

The compiler generates 7 MIPS instructions to implement the swap function:

```
MUL  $2, $5, 4      # Multiply k by 4 (array index to byte offset)
ADD  $2, $4, $2     # Add base address to offset (address of v[k])
LW   $15, 0($2)     # Load v[k] into register $15 (temp = v[k])
LW   $16, 4($2)     # Load v[k+1] into register $16
SW   $16, 0($2)     # Store v[k+1] to v[k]
SW   $15, 4($2)     # Store temp to v[k+1]
```

Translation Analysis:

- **5 C statements → 7 assembly instructions**
- Expansion due to instruction granularity
- Each assembly instruction is simple operation

Key Operations Explained:

1. Address Calculation:

- **Multiply by 4:** Each integer occupies 4 bytes in memory
- **Index k** must be converted to **byte offset (k×4)**
- Calculate memory address of v[k]

2. Memory Addressing:

- Base address of array in register $4
- Offset calculated and added to base
- Results in absolute memory address

3. Register Usage:

- **$4:** Base address of array v (parameter)
- **$5:** Value of k (parameter)
- **$2:** Temporary register for address calculation
- **$15:** Temporary storage for v[k] value
- **$16:** Temporary storage for v[k+1] value

Instruction Set Details:

- **MIPS ISA** used in example (not ARM, but similar concepts)
- Load-Store architecture
- Register-to-register operations
- Explicit memory addressing

2.10.3 9.3 After Assembly - Machine Code

Binary Representation:

Each assembly instruction translates to 32-bit binary instruction:

```
00000000101000100001000000011000  # MUL $2, $5, 4
00000000100000100001000000100001  # ADD $2, $4, $2
10001100010011110000000000000000  # LW  $15, 0($2)
10001100010100000000000000000100  # LW  $16, 4($2)
10101100010100000000000000000000  # SW  $16, 0($2)
10101100010011110000000000000100  # SW  $15, 4($2)
```

One-to-One Mapping:

- Each assembly instruction → Exactly one 32-bit machine instruction
- No information lost or gained
- Deterministic translation
- Assembly is human-readable form of machine code

Instruction Format:

Different instruction types have different bit field layouts:

R-Type (Register) Format:

```
[Opcode 6 bits][Rs 5 bits][Rt 5 bits][Rd 5 bits][Shamt 5 bits][Funct 6 bits]
```

I-Type (Immediate) Format:

```
[Opcode 6 bits][Rs 5 bits][Rt 5 bits][Immediate 16 bits]
```

Instruction Components Specify:

- **Opcode:** Operation category
- **Destination Register:** Where result goes
- **Source Registers:** Where operands come from
- **Immediate Values:** Constant values (like 4 in multiply)
- **Function Code:** Specific operation for R-type

Example Analysis:

In the immediate value 4:

- Appears in specific bit positions
- Encoded in binary (0000000000000100)
- Part of instruction encoding

Binary Image:

- Complete program represented as sequence of 32-bit words
- Called **executable** or **binary image**
- Stored in secondary storage (hard disk, SSD)
- Loaded into memory when program executes
- CPU fetches and executes instructions sequentially

2.11 10. Program Execution - Inside the CPU

2.11.1 10.1 Block Diagram of Computer

System Components:

Compiler/Tool Chain:

- Translates human-written program to machine code
- Optimization and code generation
- Produces executable binary

Memory:

- Stores program instructions
- Stores program data
- Hierarchical (cache, RAM, disk)

CPU (Central Processing Unit):

- Executes machine instructions
- Performs arithmetic and logic
- Controls program flow

Input/Output:

- Peripherals (keyboard, display, network)
- Storage devices (disk, SSD)
- Communication interfaces

Program Execution Flow:

1. Compile Stage:

- Source code → Assembly → Machine code
- Performed once (or when code changes)
- Output: Executable binary file

2. Store Stage:

- Machine code saved to secondary storage
- Persistent storage (survives power off)
- Typically on hard disk or SSD

3. Load Stage:

- Machine code loaded into main memory (RAM) when program runs
- Operating system performs loading
- Program becomes "process"

4. Execute Stage:

- CPU fetches instructions from memory one by one
- Executes each instruction in sequence (or out-of-order)
- Updates registers and memory

5. Results Stage:

- Computed values stored back in memory
- Output sent to I/O devices
- Results displayed or saved

2.11.2 10.2 Inside the CPU - Two Main Components

Datapath:

Structure:

- Collection of logic circuits interconnected
- Forms a path through CPU
- Instruction and data travel through this path
- Sequential stages of processing

Components:

- Functional units (adders, multipliers, shifters, logic units)

- Registers for temporary storage
- Multiplexers for routing
- Buses for data transfer

Function:

- Instruction travels from one logic circuit to another
- Each circuit performs specific operation on data
- Transforms inputs to outputs
- Executes the computational work

Examples of Functional Units:

- Arithmetic Logic Unit (ALU)
- Floating-Point Unit (FPU)
- Load-Store Unit
- Branch Unit

Control:

Structure:

- Another logic circuit (or set of circuits)
- Generates control signals
- Coordinates datapath operation

Function:

- Governs instruction/data flow through datapath
- Ensures instructions execute correctly
- Selects appropriate functional units
- Controls multiplexers and enables

Responsibilities:

- Decode instructions
- Generate appropriate control signals
- Coordinate timing
- Handle exceptions and interrupts

Interaction:

- **Control** tells **Datapath** what to do
- **Datapath** performs the actual computation
- **Control** monitors **Datapath** status
- Together implement instruction execution

2.11.3 10.3 Execution Process (Conveyor Belt Analogy)

Instruction Execution Cycle:

1. Fetch:

- Instructions stored in memory
- Control fetches one instruction at a time

- Brings instruction into CPU
- Increments program counter

2. **Decode:**

- Instruction enters datapath
- Control decodes instruction
- Determines operation type
- Identifies operands

3. **Execute:**

- Instruction travels through logic circuits in datapath
- Operations performed on data
- Functional units activated
- Intermediate results produced

4. **Memory:**

- Memory accesses performed if needed (load/store)
- Data read from or written to memory
- Address calculation completed

5. **Writeback:**

- Results generated
- Written back to registers
- Results may be sent to memory or I/O

6. **Repeat:**

- Cycle repeats for next instruction
- Like conveyor belt: continuous flow
- One instruction after another (in simple model)

Conveyor Belt Metaphor:

- Instructions like items on conveyor belt
- Each station performs specific operation
- Continuous movement through system
- Pipelining overlaps multiple instructions (discussed in later lectures)

2.11.4 10.4 Cache Memory

Purpose and Motivation:

The Performance Gap:

- CPU can process data very fast
- Main memory access is relatively slow
- Speed mismatch creates bottleneck
- CPU would waste time waiting for memory

Cache Solution:

- Fast memory located on CPU chip

- Very close to processor core physically
- Stores copies of frequently used instructions and data
- Exploits locality of reference

Cache Hierarchy:

Level 1 Cache (L1):

- Smallest capacity (32-64 KB)
- Fastest access (1-2 cycles)
- Closest to core
- Often split: L1-I (instruction), L1-D (data)

Level 2 Cache (L2):

- Medium capacity (256 KB - 1 MB)
- Medium access time (4-10 cycles)
- May be per-core or shared
- Unified (instructions and data)

Level 3 Cache (L3):

- Largest capacity (several MB)
- Slower access (20-40 cycles)
- Shared across all cores
- Last level cache (LLC)

Performance Impact:

- Cache hit: Data found in cache (fast)
- Cache miss: Must access main memory (slow)
- Hit rate critical for performance
- Well-designed cache can achieve >95% hit rate

Will Learn in Course:

- Cache organization
- Mapping strategies (direct-mapped, set-associative)
- Replacement policies
- Write policies
- Cache coherency in multi-core

2.12 11. Real CPU Layout - AMD Barcelona Example

2.12.1 11.1 Overview

AMD Barcelona Processor:

- Released around 2007
- Quad-core processor (4 cores on single die)
- 65nm manufacturing process
- Actual chip much smaller than magnified images

- Can visually identify individual components

Die Photo Analysis:

- Optical or electron microscope image
- Shows physical layout of components
- Different functional units visible
- Reveals organizational decisions
- Educational value for understanding architecture

2.12.2 11.2 Four Processor Cores

Core Distribution:

Physical layout shows clear quadrant organization:

- **Core 1:** Upper left area of die
- **Core 2:** Upper right area of die
- **Core 3:** Lower left area of die
- **Core 4:** Lower right area of die

Layout Strategy:

- **Mirror Image Layouts:** Cores identical but mirrored
- **Symmetry:** Simplifies design and manufacturing
- **Thermal Distribution:** Spreads heat across die
- **Interconnect Balance:** Equal distances to shared resources

2.12.3 11.3 Inside Each Core

Floating-Point Unit (FPU):

Characteristics:

- **Large Component:** Significant silicon area in each core
- **Complex Circuitry:** Handles IEEE 754 floating-point arithmetic
- **High Transistor Count:** Precision requires many gates

Operations:

- Addition, subtraction of floating-point numbers
- Multiplication of floating-point numbers
- Division of floating-point numbers
- Square root and other mathematical functions

Why So Large:

- Floating-point math more complex than integer
- Requires normalization, rounding, exception handling
- Multiple pipeline stages
- High precision demands

Load-Store Unit:

Function:

- Handles all memory operations
- Loads data from memory to CPU registers
- Stores data from CPU registers to memory
- Critical for data transfer

Operations:

- Address calculation
- Cache access
- TLB (Translation Lookaside Buffer) lookup
- Memory ordering and consistency

Integer Execution Unit:

Characteristics:

- **Smaller than FPU:** Integer operations generally simpler
- **High Frequency:** Often faster than floating-point

Operations:

- Integer arithmetic (add, subtract, multiply, divide)
- Bitwise logical operations (AND, OR, XOR, NOT)
- Shifts and rotates
- Comparisons

Why Smaller:

- Simpler algorithms
- No normalization needed
- Exact arithmetic (no rounding)
- Fewer pipeline stages

Fetch and Decode Unit:

Responsibilities:

Instruction Fetch:

- Fetches instructions from memory (via I-cache)
- Predicts branch targets
- Manages instruction buffer

Instruction Decode:

- Makes sense of binary instruction encoding
- Determines instruction type
- Identifies operands
- Generates micro-ops (for CISC architectures)

Pipeline Frontend:

- Prepares instructions for execution
- Handles instruction-level parallelism
- Feeds execution units

Level 1 Data Cache (L1 D-Cache):

Characteristics:

- Stores frequently used **data** (not instructions)
- Very fast access (1-2 cycle latency)
- Close to execution units
- Separate from instruction cache (Harvard architecture)

Typical Specifications:

- 32-64 KB capacity
- 8-way set associative
- Write-through or write-back policy

Level 1 Instruction Cache (L1 I-Cache):

Characteristics:

- Stores frequently used **instructions** (program code only)
- Very fast access
- Feeds fetch unit
- Separate from data cache

Benefits of Separation:

- No structural hazards (simultaneous instruction fetch and data access)
- Optimized for different access patterns
- Simpler control logic

Level 2 Unified Cache (L2 Cache):

Characteristics:

- **Larger than L1:** Typically 512 KB per core in Barcelona
- Stores **both instructions and data** (unified)
- Further from execution units (higher latency)
- Victim cache for L1 misses

Architecture:

- Dedicated control logic for coherency
- Interface to L3 cache or memory
- May use different associativity than L1

2.12.4 11.4 Shared Components

North Bridge (Central Hub):

Location:

- Central/middle area of chip
- Strategic position for communication

Functions:

- **L2-to-Memory Connection:** Connects all L2 caches to main memory
- **Inter-Core Communication:** Coordinates between cores

- **Memory Controller:** May include integrated memory controller
- **Cache Coherency:** Maintains coherency protocol between cores

Critical Role:

- Central communication circuit
- Bandwidth bottleneck if not designed well
- Affects multi-core scaling

DDR PHY (Physical Controller):

DDR Memory:

- **DDR:** Dual Data Rate SDRAM
- Transfers data on both rising and falling clock edges
- Industry-standard memory interface

PHY (Physical Layer):

- **PHY:** Physical layer controller
- Interfaces CPU to DDR RAM modules
- Handles physical signaling

Responsibilities:

- Electrical interface to memory chips
- Signal timing and termination
- Training and calibration
- Error detection/correction

HyperTransport Controllers:

HyperTransport Technology:

- High-speed interconnect technology (AMD proprietary)
- Point-to-point serial communication
- Replaces legacy parallel buses
- High bandwidth, low latency

Connections:

- **External Devices:** Graphics cards, other processors
- **Chipset Communication:** Northbridge, southbridge links
- **I/O Device Connectivity:** Network, storage, peripherals

Benefits:

- Scalable bandwidth
- Lower pin count than parallel buses
- NUMA (Non-Uniform Memory Access) support for multi-socket systems

2.12.5 11.5 Additional Information - WikiChip.org

WikiChip Database:

Website: https://en.wikichip.org

Comprehensive Processor Information:

Major Manufacturers Covered:

- **Intel Processors:** x86 architecture, Core series, Xeon servers
- **AMD Processors:** x86 architecture, Ryzen, EPYC, Threadripper
- **ARM Processors:** Mobile devices, embedded systems, servers
- **Samsung Exynos:** Smartphones and tablets
- **Apple A-Series:** iPhone and iPad processors
- **Apple M-Series:** Mac computers (ARM-based)
- **Qualcomm:** Snapdragon mobile processors
- **NVIDIA, Broadcom, Texas Instruments, and many more**

Available Information:

Visual Content:

- Processor die photographs and diagrams
- Block diagrams showing architecture
- Cache hierarchy visualizations
- Microarchitecture pipeline diagrams

Technical Specifications:

- Manufacturing process (nm technology)
- Transistor counts and density
- Transistor types and structures
- Die size and area
- Power consumption (TDP)
- Clock speeds (base and turbo)
- Core counts and threading
- Cache sizes and organization

Advanced Topics:

- 3D stacking technology details
- FinFET and GAA transistor structures
- Packaging technologies
- Memory interface specifications
- I/O capabilities

Current Technology Landscape (2021):

Mainstream Manufacturing:

- **10 nm and 7 nm** processes in volume production
- Multiple manufacturers at this node

Future Direction:

- **Next Few Years:** Shift to 5 nm and 3 nm
- 2 nm and 1 nm in research

Important Clarification:

- **Numbers don't represent actual gate size anymore**

- **Marketing terms** more than physical measurements
- **Example:** 5 nm transistors may have wider channels than 10 nm
- **Density Increase Through:**
  - 3D stacking (vertical integration)
  - FinFET and GAA structures
  - Improved layouts and design rules
  - Multi-patterning lithography

2.13 Key Takeaways

1. **Moore's Law predicted transistor doubling every 2 years** - remarkably accurate for over 40 years, guiding semiconductor industry planning and investment

2. **Smaller transistors enabled by improved lithography** - progression from 90nm → 45nm → 22nm → 7nm → 5nm through advancing manufacturing processes

3. **Feature size now marketing term rather than physical measurement** - modern processes use 3D structures making simple linear dimensions misleading

4. **Smaller transistors provide dual benefits** - enable more complex circuits (more transistors available) and faster switching (lower voltage, reduced impedance)

5. **Clock rate increased exponentially until ~2004** - grew from 12.5 MHz (1982) to 3.6 GHz (2004), then hit fundamental thermal limitations

6. **Power wall halted frequency scaling** - heat generation ($P = CV^2f$) exceeded cooling capability, establishing ~100W practical limit for consumer processors

7. **Dynamic power equation explains the crisis** - despite 25× power reduction from voltage scaling, 300× frequency increase overwhelmed the benefit

8. **Overclocking emerged as risky performance technique** - users could exceed rated speeds at risk of destroying processors, popular among gaming enthusiasts

9. **Industry pivoted to multi-core processors** - solution to utilize Moore's Law transistors without exceeding power limits, starting ~2004-2008

10. **Multi-core growth slowed due to programming difficulty** - initial projection of hundreds of cores didn't materialize; parallel programming remains challenging

11. **Parallel programming requires explicit management** - unlike automatic instruction-level parallelism, multi-core requires programmers to handle threads, synchronization, communication

12. **Three major parallel programming challenges** - load balancing across cores, minimizing communication overhead, optimizing synchronization

13. **3D chip technology changed scaling paradigm (2013-2015)** - industry shifted from pure 2D shrinking to vertical stacking of transistor layers

14. **ITRS dissolved in 2015** - technology roadmap organization ended as multiple paths to density replaced simple feature size scaling

15. **Computer systems organized in three layers** - hardware (physical components), system software (OS, compilers, tools), application software (user programs)
16. **System software provides abstraction and protection** - OS prevents malicious programs from damaging hardware, hides complexity from application programmers
17. **Program translation is multi-stage process** - high-level language → assembly language → machine code through compiler, assembler, linker
18. **CPU contains datapath and control** - datapath performs computation by routing data through functional units; control coordinates execution and generates signals
19. **Cache memory critical for performance** - fast on-chip memory (L1, L2, L3) stores frequently accessed data/instructions, hiding main memory latency
20. **Real CPUs have complex layouts** - die photos reveal intricate organization with multiple cores, cache hierarchies, shared interconnects, memory controllers

---

2.14 Summary

This lecture provided a comprehensive examination of computer technology evolution from the 1970s to present day. Moore's Law, predicting transistor count doubling every two years, served as the guiding principle for the semiconductor industry and enabled the transformation of computers from room-sized machines to powerful pocket devices.

The progression of manufacturing technology steadily reduced feature sizes from 90 nanometers to current 7nm and 5nm processes. Smaller transistors provided two key advantages: more transistors per chip enabling complex functionality, and faster switching speeds enabling higher clock frequencies. Clock rates grew exponentially from 12.5 MHz in 1982 to 3.6 GHz in 2004.

However, around 2004, the industry encountered the power wall - a fundamental thermal limitation. The dynamic power equation ($P = CV^2f$) revealed that despite aggressive voltage scaling, the massive frequency increases caused power consumption and heat generation to exceed cooling capabilities. The ~100-watt limit for consumer processors could not be overcome by improved cooling solutions.

The solution was multi-core processors: placing multiple complete CPU cores on a single chip. This allowed continued performance improvement within power constraints by exploiting thread-level parallelism. However, the initial vision of exponentially growing core counts didn't materialize due to the difficulty of parallel programming. Unlike automatic instruction-level parallelism, multi-core requires programmers to explicitly manage threads, balance loads, minimize communication, and handle synchronization - a significantly more challenging paradigm.

Around 2013-2015, the industry made another major shift to 3D chip technology. Instead of only shrinking transistors in two dimensions, manufacturers began stacking transistor layers vertically using FinFET and similar technologies. This represented such a fundamental change that the International Technology Roadmap for Semiconductors (ITRS) dissolved in 2015, as simple feature-size predictions no longer captured the diverse approaches to increasing transistor density.

The lecture concluded by examining computer system organization across three layers: hardware (processor, memory, I/O), system software (compilers, assemblers, operating system), and application software (programs written in high-level languages). We traced the complete journey from high-level code through compilation and assembly to binary machine code, and explored how programs execute through the interaction of control and datapath components within the CPU. Cache memory's critical role in hiding main memory latency was emphasized, and real-world processor layouts illustrated the complex organization of modern multi-core chips.

Understanding these technology trends and architectural responses provides essential context for studying computer architecture and explains why processors are organized as they are today.

Chapter 3

Lecture 3: Understanding Performance

3.1 Introduction

Understanding computer performance is fundamental to computer architecture and system design. This lecture explores how performance is measured, the factors that influence it, and the principles that guide performance optimization. We examine the metrics used to evaluate systems, the mathematical relationships between performance factors, and Amdahl's Law—a critical principle for understanding the limits of performance improvements.

3.2 1. Defining and Measuring Performance

3.2.1 1.1 Response Time vs. Throughput

Response Time (Execution Time)

- Time to complete a single task
- Includes all overhead and waiting time
- User-perceived performance metric
- Example: Time for a program to run from start to finish

Throughput (Bandwidth)

- Number of tasks completed per unit time
- Measures system capacity
- Important for servers and data centers
- Example: Number of transactions processed per second

Relationship Between Metrics

- Improving response time often improves throughput
- Improving throughput doesn't always improve response time

- Different optimization strategies for each metric
- System design must balance both considerations

3.2.2 1.2 Performance Definition

Mathematical Definition

```
Performance = 1 / Execution Time
```

Performance Comparison

- If System A is faster than System B:
  - Execution Time_A < Execution Time_B
  - Performance_A > Performance_B

Relative Performance

```
Performance_A / Performance_B = Execution Time_B / Execution Time_A
```

Example: If System A is 2× faster than System B:

- Performance_A / Performance_B = 2
- Execution Time_B / Execution Time_A = 2
- System A takes half the time of System B

3.3 2. CPU Time and Performance Factors

3.3.1 2.1 Components of Execution Time

Total Execution Time

- CPU time: Time CPU spends computing the task
- I/O time: Time waiting for input/output operations
- Other system activities: OS overhead, other programs

CPU Time Focus

- Primary metric for processor performance
- Excludes I/O and system effects
- Directly reflects processor and memory system performance
- Most relevant for comparing processor architectures

3.3.2 2.2 The CPU Time Equation

Basic Formula

```
CPU Time = Clock Cycles × Clock Period
```

Or equivalently:

```
CPU Time = Clock Cycles / Clock Rate
```

Key Relationships

- Clock Period = 1 / Clock Rate
- Clock Rate measured in Hz (cycles/second)
- Clock Cycles = total cycles to execute program
- Higher clock rate → shorter clock period → faster execution

Example Calculation

```
Program requires 10 billion cycles
Processor runs at 4 GHz (4 × 10^9 Hz)

CPU Time = 10 × 10^9 cycles / (4 × 10^9 cycles/sec)
         = 2.5 seconds
```

3.3.3 2.3 Instruction Count and CPI

Cycles Per Instruction (CPI)

- Average number of clock cycles per instruction
- Varies by instruction type and implementation
- Key microarchitecture metric

Extended CPU Time Equation

```
CPU Time = Instruction Count × CPI × Clock Period
```

Or:

```
CPU Time = (Instruction Count × CPI) / Clock Rate
```

Three Performance Factors

1. **Instruction Count**: Number of instructions executed
2. **CPI**: Average cycles per instruction
3. **Clock Rate**: Speed of the processor clock

Factor Dependencies

- Instruction Count: Determined by algorithm, compiler, ISA
- CPI: Determined by processor implementation (microarchitecture)
- Clock Rate: Determined by hardware technology and organization

3.4 3. Understanding CPI in Detail

3.4.1 3.1 CPI Variability

Different Instructions, Different CPIs

- Simple operations: May complete in 1 cycle (ADD, AND)
- Memory operations: May take multiple cycles (LOAD, STORE)
- Branch instructions: Variable cycles (depends on prediction)
- Multiply/Divide: Often take many cycles

Calculating Average CPI

```
Average CPI = Σ (CPI_i × Instruction Count_i) / Total Instruction Count
```

Where:

- CPI_i = cycles per instruction for instruction type i
- Instruction Count_i = number of times instruction i executed

3.4.2 3.2 CPI Example Calculation

Given:

- Program executes 100,000 instructions
- 50,000 ALU operations (CPI = 1)
- 30,000 load instructions (CPI = 3)
- 20,000 branch instructions (CPI = 2)

Calculation:

```
Total Cycles = (50,000 × 1) + (30,000 × 3) + (20,000 × 2)
             = 50,000 + 90,000 + 40,000
             = 180,000 cycles

Average CPI = 180,000 / 100,000 = 1.8
```

3.4.3 3.3 Instruction Classes

Common Instruction Categories

1. **Integer arithmetic**: ADD, SUB, AND, OR
2. **Data transfer**: LOAD, STORE
3. **Control flow**: BRANCH, JUMP, CALL
4. **Floating-point**: FADD, FMUL, FDIV

CPI Characteristics by Class

- Integer arithmetic: Usually 1 cycle
- Data transfer: 1-3 cycles (cache hit) or more (cache miss)
- Control flow: 1-2 cycles (correct prediction) or more (misprediction)
- Floating-point: 2-20+ cycles depending on operation

3.5 4. Performance Optimization Principles

3.5.1 4.1 Make the Common Case Fast

Core Principle

- Optimize frequent operations rather than rare ones
- Greater impact on overall performance
- Focus resources where they matter most

Examples

- Optimize ALU operations (common) over division (rare)
- Fast cache for recent data (commonly accessed)
- Branch prediction for likely paths
- Simple instructions execute quickly

Application in Design

- Identify common operations through profiling
- Allocate hardware resources accordingly
- Accept slower performance for rare cases
- Trade-offs guided by usage patterns

3.5.2 4.2 Amdahl's Law

The Fundamental Principle The speedup that can be achieved by improving a particular part of a system is limited by the fraction of time that part is used.

Mathematical Formula

```
Speedup_overall = 1 / [(1 - P) + (P / S)]
```

Where:

- P = Proportion of execution time that can be improved
- S = Speedup of the improved portion
- (1 - P) = Proportion that cannot be improved

Alternative Formulation

```
Execution Time_new = Execution Time_old × [(1 - P) + (P / S)]
```

3.5.3 4.3 Amdahl's Law Examples

Example 1: Multiply Operation Speedup

Given:

- Multiply operations take 80% of execution time
- New hardware makes multiplies 10× faster

Calculation:

```
P = 0.80 (80% can be improved)
S = 10 (10× speedup)

Speedup_overall = 1 / [(1 - 0.80) + (0.80 / 10)]
                = 1 / [0.20 + 0.08]
                = 1 / 0.28
                = 3.57×
```

Key Insight: Despite 10× improvement in multiplies, overall speedup is only 3.57× because 20% of time is unaffected.

Example 2: Limited Improvement Fraction

Given:

- Only 30% of execution can be improved
- Improvement is 100× faster

Calculation:

```
P = 0.30
S = 100

Speedup_overall = 1 / [(1 - 0.30) + (0.30 / 100)]
                = 1 / [0.70 + 0.003]
                = 1 / 0.703
                = 1.42×
```

Key Insight: Even with 100× improvement, overall speedup is only 1.42× because only 30% of execution benefits.

3.5.4 4.4 Implications of Amdahl's Law

Limitations of Parallelization

- Serial portions limit parallel speedup
- As parallelism increases, serial portion dominates
- Cannot achieve infinite speedup regardless of cores

Optimization Strategy

- Focus on largest contributors to execution time
- Consider what fraction can realistically be improved
- Multiple small improvements may beat one large improvement
- Balance improvements across components

Example: Multicore Scaling

```
If 90% of program parallelizes perfectly:
2 cores:  Speedup = 1.82×
4 cores:  Speedup = 3.08×
8 cores:  Speedup = 4.71×
16 cores: Speedup = 6.40×
∞ cores:  Speedup = 10.00× (maximum possible)
```

The 10% serial portion ultimately limits speedup to 10×.

3.6 5. Complete Performance Analysis

3.6.1 5.1 The Complete Performance Equation

Bringing It All Together

```
CPU Time = (Instruction Count × CPI × Clock Period)
```

Expanded:

```
CPU Time = (Instructions) × (Cycles/Instruction) × (Seconds/Cycle)
```

What Affects Each Factor

Instruction Count:

- Algorithm: Efficient algorithms execute fewer instructions
- Programming language: High-level vs low-level
- Compiler: Optimization quality
- ISA: Instruction complexity and capabilities

CPI:

- ISA: Instruction complexity
- Microarchitecture: Pipeline depth, branch prediction
- Cache performance: Hit rates affect memory access CPI
- Instruction mix: Distribution of instruction types

Clock Period (or Clock Rate):

- Technology: Transistor speed (nm process)
- Organization: Pipeline depth, critical path length
- Power constraints: Higher frequency requires more power
- Cooling limitations: Heat dissipation capacity

3.6.2 5.2 Performance Comparison Example

Scenario: Compare two implementations of the same ISA

- System A: Clock Rate = 2 GHz, CPI = 2.0
- System B: Clock Rate = 3 GHz, CPI = 3.0
- Same program with 1 million instructions

System A:

```
CPU Time_A = (1 × 10^6 instructions) × (2.0 cycles/instruction) / (2 × 10^9 cycles/se
           = 2 × 10^6 cycles / (2 × 10^9 cycles/sec)
           = 0.001 seconds = 1 millisecond
```

System B:

```
CPU Time_B = (1 × 10^6 instructions) × (3.0 cycles/instruction) / (3 × 10^9 cycles/se
           = 3 × 10^6 cycles / (3 × 10^9 cycles/sec)
           = 0.001 seconds = 1 millisecond
```

Result: Both systems have identical performance despite different clock rates and CPIs.

3.6.3 5.3 Trade-offs in Design

Clock Rate vs. CPI Trade-off

- Higher clock rate may require deeper pipeline
- Deeper pipeline often increases CPI (more stalls)
- Must balance frequency gains against CPI losses

Instruction Count vs. CPI Trade-off

- Complex instructions reduce instruction count
- But complex instructions may increase CPI
- CISC vs RISC architecture debate

Power vs. Performance

- Higher clock rate increases power consumption
- Power = Capacitance × Voltage² × Frequency
- Mobile systems prioritize power over peak performance

3.7 6. Practical Performance Considerations

3.7.1 6.1 Benchmarking

Purpose of Benchmarks

- Measure real-world performance
- Compare different systems objectively
- Standard workloads for reproducibility

Types of Benchmarks

- Synthetic: Artificial programs (e.g., Dhrystone, Whetstone)
- Application: Real programs (e.g., SPEC CPU, databases)
- Workload: Representative task mixes

Benchmark Pitfalls

- May not represent your workload
- Can be optimized for unfairly
- Need multiple benchmarks for complete picture

3.7.2 6.2 Performance Metrics in Practice

MIPS (Million Instructions Per Second)

```
MIPS = Instruction Count / (Execution Time × 10^6)
     = Clock Rate / (CPI × 10^6)
```

Limitations of MIPS:

- Doesn’t account for instruction complexity
- Different ISAs have different instruction capabilities
- Higher MIPS doesn’t guarantee better performance
- “Meaningless Indication of Processor Speed”

Better Metrics:

- Execution time for specific workloads
- Throughput for server applications
- Energy efficiency (performance per watt)
- Performance per dollar

3.7.3 6.3 Power and Energy Considerations

Power Wall

- Cannot increase clock rate indefinitely
- Power consumption limits frequency scaling
- Led to multi-core era

Dynamic Power Equation

```
Power = Capacitance × Voltage² × Frequency
```

Energy Equation

```
Energy = Power × Time
```

Implications:

- Lowering voltage reduces power dramatically (squared effect)
- Higher frequency increases power linearly
- Faster execution may save energy overall (less time)
- Energy efficiency increasingly important metric

3.8 Key Takeaways

1. **Performance is the inverse of execution time** - faster systems have shorter execution times and higher performance values.

2. **Three key factors determine CPU performance:**

   - Instruction Count (algorithm, compiler, ISA)
   - CPI (microarchitecture, instruction mix)
   - Clock Rate (technology, organization)

3. **Amdahl's Law limits speedup** - the potential speedup from improving any part of a system is limited by how much time that part is used.

4. **"Make the common case fast"** - optimize frequently executed operations for maximum impact on overall performance.

5. **CPI varies by instruction type** - average CPI depends on the mix of instructions and their individual costs.

6. **Trade-offs are fundamental** - improvements in one area (e.g., clock rate) may harm another (e.g., CPI or power consumption).

7. **Benchmarking is essential** - real workloads provide the most meaningful performance measurements.

8. **Power is a critical constraint** - modern performance optimization must consider power and energy efficiency, not just speed.

9. **Multiple factors must be optimized together** - focusing on only one aspect (like clock rate) can be counterproductive.

10. **Understanding performance equations** enables rational design decisions and accurate performance predictions.

3.9 Summary

Performance analysis is central to computer architecture, providing the foundation for making informed design decisions. By understanding the relationship between instruction count, CPI, and clock rate, architects can identify optimization opportunities and predict the impact of changes. Amdahl's Law reminds us that the benefit of any improvement is constrained by what fraction of execution time it affects, emphasizing the importance of focusing on the common case. As we design systems, we must balance competing factors—clock rate, CPI, power consumption, and cost—to achieve the best overall performance for target applications. The principles covered in this lecture provide the analytical framework for evaluating processor designs and optimization strategies throughout the study of computer architecture.

Chapter 4

Lecture 4: Introduction to ARM Assembly

4.1 Introduction

This lecture introduces ARM assembly language programming, providing the foundation for understanding how high-level programs translate to machine code. We explore the ARM instruction set architecture (ISA), focusing on its RISC design philosophy, register organization, basic instruction formats, and the toolchain used for development. Understanding assembly language is essential for comprehending how processors execute programs and for optimizing performance-critical code.

4.2 1. ARM Architecture Overview

4.2.1 1.1 RISC Philosophy

Reduced Instruction Set Computer (RISC)

- Simple, uniform instruction format
- Fixed instruction length (32 bits in ARM)
- Load/store architecture (only LOAD/STORE access memory)
- Large number of general-purpose registers
- Few addressing modes
- Hardware simplicity for higher clock rates

Contrasted with CISC (Complex Instruction Set Computer)

- CISC: Variable-length instructions, complex operations
- CISC: Fewer registers, memory operands in arithmetic
- RISC: Simpler instructions, more instructions per program
- RISC: Regular structure enables pipelining

ARM Design Principles

- Simplicity enables high performance

- Regular instruction encoding aids decoding
- Load/store architecture simplifies memory access
- Large register file reduces memory traffic
- Consistent design across instruction types

4.2.2 1.2 ARM Registers

General-Purpose Registers

- **R0 to R15**: 16 registers total
- **32 bits wide**: Can hold integers, addresses, or data
- **R0-R12**: General computation and data storage
- **R13 (SP)**: Stack Pointer - points to top of stack
- **R14 (LR)**: Link Register - stores return address
- **R15 (PC)**: Program Counter - address of next instruction

Register Usage Conventions

```
R0-R3:   Argument/result registers
         - Pass parameters to functions
         - Return values from functions
         - Scratch registers (not preserved)

R4-R11:  Local variable registers
         - Must be preserved across function calls
         - Callee saves/restores if used

R12:     Intra-procedure-call scratch register
         - Can be corrupted by function calls
         - Not preserved

R13 (SP): Stack Pointer
         - Points to top of stack
         - Must always be valid

R14 (LR): Link Register
         - Stores return address on function call
         - Contains address to return to

R15 (PC): Program Counter
         - Always points to next instruction
         - Modifying PC changes execution flow
```

Why So Many Registers?

- Reduces memory accesses (faster than cache/RAM)
- Enables register allocation by compiler
- Supports efficient function calls
- Improves performance through locality

4.2.3 1.3 Memory Organization

Little-Endian Byte Ordering

- Least significant byte at lowest address
- Example: 0x12345678 stored as:

```
Address:  [base+0] [base+1] [base+2] [base+3]
Content:     78       56       34       12
```

Word Alignment

- Words are 32 bits (4 bytes)
- Word addresses should be multiples of 4
- Accessing unaligned words may cause errors or slowdown

Address Space

- 32-bit addresses can access 2^{32} bytes = 4 GB
- Byte-addressable memory
- Instructions and data in same address space (Von Neumann architecture)

4.3 2. ARM Instruction Format

4.3.1 2.1 Instruction Structure

Fixed 32-Bit Length

- Every instruction exactly 32 bits
- Simplifies instruction fetch and decode
- Enables predictable pipeline operation

Typical Instruction Fields

```
[Condition][Opcode][Operands]
  4 bits    varies   varies
```

Example: ADD Instruction

```
ADD R1, R2, R3    ; R1 = R2 + R3

Encoding includes:
- Condition code (usually "always")
- Opcode for ADD operation
- Destination register (R1)
- Source register 1 (R2)
- Source register 2 (R3)
```

4.3.2 2.2 Instruction Types

Data Processing Instructions

- Arithmetic: ADD, SUB, RSB (reverse subtract)

- Logical: AND, ORR, EOR (XOR), BIC (bit clear)
- Comparison: CMP, CMN, TST, TEQ
- Move: MOV, MVN (move negated)
- Shift/Rotate: LSL, LSR, ASR, ROR

Data Transfer Instructions

- Load: LDR (word), LDRB (byte), LDRH (halfword)
- Store: STR (word), STRB (byte), STRH (halfword)
- Multiple: LDM, STM (load/store multiple registers)

Control Flow Instructions

- Branch: B (unconditional), BEQ, BNE, BGE, BLT, etc.
- Function call: BL (branch and link)
- Return: MOV PC, LR

4.3.3 2.3 Operand Types

Register Operands

```
ADD R0, R1, R2    ; R0 = R1 + R2 (all registers)
```

Immediate Operands

```
ADD R0, R1, #5    ; R0 = R1 + 5 (# indicates immediate)
MOV R2, #100      ; R2 = 100
```

Immediate Value Constraints

- Limited to certain patterns due to 32-bit instruction encoding
- 8-bit immediate + 4-bit rotation
- Assembler warns if immediate cannot be encoded

Shifted Register Operands

```
ADD R0, R1, R2, LSL #2    ; R0 = R1 + (R2 << 2)
SUB R3, R4, R5, LSR #1    ; R3 = R4 - (R5 >> 1)
```

4.4 3. Basic ARM Instructions

4.4.1 3.1 Arithmetic Instructions

Addition

```
ADD Rd, Rn, Rm       ; Rd = Rn + Rm
ADD Rd, Rn, #imm     ; Rd = Rn + immediate

Examples:
ADD R0, R1, R2       ; R0 = R1 + R2
ADD R3, R3, #1       ; R3 = R3 + 1 (increment)
```

Subtraction

```
SUB Rd, Rn, Rm        ; Rd = Rn - Rm
SUB Rd, Rn, #imm      ; Rd = Rn - immediate
RSB Rd, Rn, #imm      ; Rd = immediate - Rn (reverse subtract)

Examples:
SUB R0, R1, R2        ; R0 = R1 - R2
SUB R4, R4, #10       ; R4 = R4 - 10 (decrement)
RSB R5, R6, #0        ; R5 = 0 - R6 (negate)
```

Multiplication (covered in later tutorials)

```
MUL Rd, Rn, Rm        ; Rd = Rn × Rm (lower 32 bits)
```

4.4.2 3.2 Logical Instructions

AND Operation

```
AND Rd, Rn, Rm        ; Rd = Rn AND Rm
AND Rd, Rn, #imm      ; Rd = Rn AND immediate

Usage: Bit masking, clearing specific bits
Example: AND R0, R0, #0xFF  ; Keep only lower 8 bits
```

OR Operation

```
ORR Rd, Rn, Rm        ; Rd = Rn OR Rm (ORR in ARM)
ORR Rd, Rn, #imm      ; Rd = Rn OR immediate

Usage: Setting specific bits
Example: ORR R1, R1, #0x80  ; Set bit 7
```

Exclusive OR

```
EOR Rd, Rn, Rm        ; Rd = Rn XOR Rm
EOR Rd, Rn, #imm      ; Rd = Rn XOR immediate

Usage: Toggling bits, fast comparison
Example: EOR R2, R2, R2     ; R2 = 0 (XOR with itself)
```

Move and Move Not

```
MOV Rd, Rm            ; Rd = Rm
MOV Rd, #imm          ; Rd = immediate
MVN Rd, Rm            ; Rd = NOT Rm (bitwise complement)

Examples:
MOV R0, R1            ; Copy R1 to R0
MOV R2, #0            ; Clear R2
MVN R3, R4            ; R3 = ~R4 (invert all bits)
```

4.4.3 3.3 Shift Operations

Logical Shift Left (LSL)

```
LSL Rd, Rn, #shift   ; Rd = Rn << shift
MOV Rd, Rn, LSL #shift

Effect: Multiplies by 2^shift
Example: LSL R0, R1, #2  ; R0 = R1 × 4
```

Logical Shift Right (LSR)

```
LSR Rd, Rn, #shift   ; Rd = Rn >> shift (unsigned)
MOV Rd, Rn, LSR #shift

Effect: Divides by 2^shift (unsigned)
Example: LSR R0, R1, #3  ; R0 = R1 / 8
```

Arithmetic Shift Right (ASR)

```
ASR Rd, Rn, #shift   ; Rd = Rn >> shift (signed)

Effect: Divides by 2^shift, preserves sign
Example: ASR R0, R1, #2  ; R0 = R1 / 4 (signed)
```

Rotate Right (ROR)

```
ROR Rd, Rn, #shift   ; Rotate Rn right by shift

Effect: Bits rotated off right end reappear at left
Example: ROR R0, R1, #8  ; Rotate R1 right by 8 bits
```

4.5 4. Memory Access Instructions

4.5.1 4.1 Load Instructions

Load Word (LDR)

```
LDR Rd, [Rn]         ; Rd = Memory[Rn]
LDR Rd, [Rn, #offset]; Rd = Memory[Rn + offset]

Examples:
LDR R0, [R1]         ; Load word from address in R1
LDR R2, [R3, #4]     ; Load from address R3+4
LDR R4, [R5, #-8]    ; Load from address R5-8
```

Load Byte (LDRB)

```
LDRB Rd, [Rn, #offset]; Load one byte, zero-extend to 32 bits

Example:
LDRB R0, [R1]        ; R0 = (byte at R1), upper 24 bits = 0
```

Load Halfword (LDRH)

```
LDRH Rd, [Rn, #offset]; Load 16 bits, zero-extend to 32 bits

Example:
LDRH R0, [R1, #2]    ; R0 = (halfword at R1+2), upper 16 bits = 0
```

Pseudo-Instruction for Loading Addresses

```
LDR Rd, =label       ; Load address of label into Rd
LDR Rd, =value       ; Load 32-bit constant into Rd

Examples:
LDR R0, =array       ; R0 = address of array
LDR R1, =0x12345678  ; R1 = 0x12345678 (large immediate)
```

4.5.2 4.2 Store Instructions

Store Word (STR)

```
STR Rd, [Rn]         ; Memory[Rn] = Rd
STR Rd, [Rn, #offset]; Memory[Rn + offset] = Rd

Examples:
STR R0, [R1]         ; Store R0 to address in R1
STR R2, [R3, #8]     ; Store R2 to address R3+8
```

Store Byte (STRB)

```
STRB Rd, [Rn, #offset]; Store lower 8 bits of Rd

Example:
STRB R0, [R1]        ; Store lower byte of R0 to address R1
```

Store Halfword (STRH)

```
STRH Rd, [Rn, #offset]; Store lower 16 bits of Rd

Example:
STRH R0, [R1, #4]    ; Store lower halfword of R0 to R1+4
```

4.5.3 4.3 Addressing Modes

Offset Addressing

```
LDR R0, [R1, #4]     ; R0 = Memory[R1 + 4], R1 unchanged
```

Pre-indexed Addressing

```
LDR R0, [R1, #4]!    ; R1 = R1 + 4, then R0 = Memory[R1]
                      ; ! indicates update base register
```

Post-indexed Addressing

```
LDR R0, [R1], #4     ; R0 = Memory[R1], then R1 = R1 + 4
```

Register Offset

```
LDR R0, [R1, R2]     ; R0 = Memory[R1 + R2]
LDR R0, [R1, R2, LSL #2] ; R0 = Memory[R1 + (R2 << 2)]
```

4.6 5. Assembly Program Structure

4.6.1 5.1 Directives

Section Directives

```
.text                ; Code section (instructions)
.data                ; Data section (initialized variables)
.bss                 ; Uninitialized data section
```

Global and External

```
.global main         ; Make symbol visible to linker
.extern printf       ; Declare external symbol
```

Data Definition

```
.word value          ; Define 32-bit word
.byte value          ; Define byte
.asciz "string"      ; Define null-terminated string
.space n             ; Reserve n bytes of space
```

4.6.2 5.2 Labels

Purpose

- Mark locations in code or data
- Provide symbolic names for addresses
- Enable jumps and references

Syntax

```
label:               ; Label for instruction
    MOV R0, #1
    ADD R1, R0, R2

array:               ; Label for data
    .word 1, 2, 3, 4
```

4.6.3 5.3 Simple Program Example

```
    .text
    .global main

main:
    MOV R0, #5        ; R0 = 5
    MOV R1, #10       ; R1 = 10
    ADD R2, R0, R1    ; R2 = R0 + R1 = 15
    MOV R0, R2        ; R0 = R2 (return value)
    MOV PC, LR        ; Return from main

    .data
message:
    .asciz "Hello, ARM!"
```

4.7 6. ARM Development Tools

4.7.1 6.1 Toolchain Components

Cross-Compiler

- `arm-linux-gnueabi-gcc`: Compiles C to ARM code
- Runs on x86 PC, produces ARM binaries
- Necessary because development machine ≠ target machine

Assembler

- `arm-linux-gnueabi-as`: Assembles ARM assembly to object code
- Part of binutils package

Linker

- `arm-linux-gnueabi-ld`: Links object files to executable
- Resolves symbols, combines code sections

Emulator

- `qemu-arm`: Emulates ARM processor on x86
- Allows running ARM binaries on PC
- Useful for testing without ARM hardware

4.7.2 6.2 Compilation Process

From C to Executable

```
C Source (.c)
    ↓ [gcc -S]
Assembly (.s)
    ↓ [as]
Object Code (.o)
    ↓ [ld]
Executable (a.out)
    ↓ [qemu-arm]
Execution
```

Command Examples

```
# Compile C to assembly
arm-linux-gnueabi-gcc -S program.c -o program.s

# Assemble to object code
arm-linux-gnueabi-as program.s -o program.o

# Link to executable
arm-linux-gnueabi-gcc program.o -o program

# Run with emulator
qemu-arm program
```

One-Step Compilation

```
# Compile, assemble, and link in one command
arm-linux-gnueabi-gcc program.c -o program
```

4.7.3 6.3 Debugging and Inspection

GDB (GNU Debugger)

```
# Debug with QEMU and GDB
qemu-arm -g 1234 program &      # Start QEMU, wait for debugger
arm-linux-gnueabi-gdb program   # Start GDB
(gdb) target remote :1234       # Connect to QEMU
(gdb) break main                # Set breakpoint
(gdb) continue                  # Run to breakpoint
(gdb) step                      # Execute one instruction
(gdb) info registers            # Show register values
```

Objdump

```
# Disassemble binary to assembly
arm-linux-gnueabi-objdump -d program
```

nm

```
# List symbols in object file
arm-linux-gnueabi-nm program.o
```

4.8 7. Programming in ARM Assembly

4.8.1 7.1 Translating C to ARM

C Code:

```
int a = 5;
int b = 10;
int c = a + b;
```

ARM Assembly:

```
    MOV R0, #5       ; a = 5
    MOV R1, #10      ; b = 10
    ADD R2, R0, R1   ; c = a + b
```

C Code with Array:

```
int arr[3] = {1, 2, 3};
int x = arr[1];
```

ARM Assembly:

```
    .data
arr:
    .word 1, 2, 3

    .text
    LDR R0, =arr     ; R0 = address of arr
    LDR R1, [R0, #4] ; R1 = arr[1] (offset 4 bytes)
```

4.8.2 7.2 Common Patterns

Clearing a Register

```
MOV R0, #0           ; Method 1
EOR R0, R0, R0       ; Method 2 (XOR with itself)
```

Negating a Value

```
RSB R0, R0, #0       ; R0 = 0 - R0
MVN R0, R0           ; R0 = ~R0 (bitwise, not arithmetic)
ADD R0, R0, #1       ; Then add 1 (two's complement)
```

Multiplying by Powers of 2

```
LSL R0, R1, #3       ; R0 = R1 × 8 (faster than MUL)
```

Dividing by Powers of 2

```
LSR R0, R1, #2       ; R0 = R1 / 4 (unsigned)
ASR R0, R1, #2       ; R0 = R1 / 4 (signed)
```

Swapping Two Registers

```
EOR R0, R0, R1       ; XOR-based swap (no temporary)
EOR R1, R0, R1
EOR R0, R0, R1
```

4.9 Key Takeaways

1. **ARM follows RISC principles** - simple instructions, load/store architecture, large register file, fixed instruction length.

2. **16 registers (R0-R15)** with special purposes: R13 (SP), R14 (LR), R15 (PC), and calling conventions for R0-R11.

3. **Three main instruction categories** - data processing (arithmetic/logic), data transfer (load/store), control flow (branches).
4. **Fixed 32-bit instruction format** simplifies hardware and enables efficient pipelining.
5. **Little-endian byte ordering** - least significant byte stored at lowest address.
6. **Immediate values** indicated by # symbol, with encoding constraints due to fixed instruction size.
7. **Memory access only through LOAD/STORE** - arithmetic operations work on registers only (load/store architecture).
8. **Rich addressing modes** - offset, pre-indexed, post-indexed, register offset with optional shifts.
9. **Cross-compilation toolchain** - arm-linux-gnueabi-gcc, as, ld, and qemu-arm for development on x86.
10. **Assembly programming requires understanding** of register allocation, instruction selection, and calling conventions.

4.10 Summary

ARM assembly language provides the low-level interface between software and hardware, revealing how high-level constructs translate to machine operations. The ARM architecture's RISC design emphasizes simplicity and regularity, with a uniform 32-bit instruction format, a generous 16-register set, and a clean separation between computation (using registers) and memory access (through explicit load/store instructions). Understanding ARM assembly is crucial for optimizing performance-critical code, implementing system-level software, and comprehending how processors execute programs. The development toolchain—including cross-compilers, assemblers, linkers, and emulators—enables efficient development and testing of ARM software. Mastering these fundamentals prepares us for more advanced topics including function calling conventions, stack management, and processor microarchitecture implementation.

Chapter 5

Lecture 5: Number Representation and Data Processing

5.1 Introduction

This lecture delves into how computers represent and manipulate data at the binary level. We explore number systems, two's complement representation for signed integers, instruction encoding formats in ARM assembly, and logical operations for bit manipulation. Understanding these fundamentals is essential for programming efficiently in assembly language and comprehending how processors execute arithmetic and logical operations.

5.2 1. Number Representation Systems

5.2.1 1.1 Unsigned Binary Integers

Binary System Basics

- Base-2 number system using digits 0 and 1
- Each bit position represents a power of 2
- Rightmost bit is least significant (LSB)
- Leftmost bit is most significant (MSB)

Place Value Calculation

```
Binary: 1011
Value = (1 × 2³) + (0 × 2²) + (1 × 2¹) + (1 × 2 )
      = 8 + 0 + 2 + 1
      = 11 (decimal)
```

N-Bit Unsigned Range

- N bits can represent 2^N different values
- Range: 0 to (2^N - 1)

- 8 bits: 0 to 255
- 32 bits: 0 to 4,294,967,295

Binary to Decimal Conversion

```
Example: 10110101
= 1×128 + 0×64 + 1×32 + 1×16 + 0×8 + 1×4 + 0×2 + 1×1
= 128 + 32 + 16 + 4 + 1
= 181
```

5.2.2 1.2 Two's Complement Representation

Purpose of Two's Complement

- Represents both positive and negative integers
- Simplifies hardware (same adder for signed/unsigned)
- Unique zero representation
- Natural overflow behavior

Sign Bit

- MSB indicates sign
- MSB = 0: Positive number
- MSB = 1: Negative number

Positive Numbers

- Same as unsigned binary
- MSB is always 0
- Example: +5 in 8 bits = 00000101

Negative Numbers

- Represented as 2^N - |value|
- Example: -5 in 8 bits:

  ```
  2^8 - 5 = 256 - 5 = 251 = 11111011
  ```

Two's Complement Conversion Method 1 (Invert and Add):

1. Write positive value in binary
2. Invert all bits (0→1, 1→0)
3. Add 1 to result

Example: -5 in 8 bits

```
+5:         00000101
Invert:     11111010
Add 1:      11111011  (this is -5)
```

Method 2 (Subtraction):

```
-5 = 2^8 - 5 = 256 - 5 = 251 = 11111011
```

N-Bit Signed Range

- Range: -(2^(N-1)) to +(2^(N-1) - 1)
- 8 bits: -128 to +127
- 32 bits: -2,147,483,648 to +2,147,483,647

Special Cases

- Zero: 00000000 (unique representation)
- Most negative: 10000000 (-128 in 8 bits)
  - Has no positive counterpart!
  - Negating gives overflow

5.2.3 1.3 Sign Extension

Purpose

- Extend smaller signed value to larger width
- Preserve numerical value
- Required when loading bytes/halfwords into 32-bit registers

Process

- Replicate the sign bit (MSB) to fill new bits
- Preserves positive/negative value

Examples

```
8-bit to 32-bit:
00000101 (+5) → 00000000 00000000 00000000 00000101 (+5)
11111011 (-5) → 11111111 11111111 11111111 11111011 (-5)
```

ARM Instructions for Sign Extension

- **LDRH**: Load halfword (16 bits), zero-extend to 32 bits
- **LDRSH**: Load signed halfword, sign-extend to 32 bits
- **LDRB**: Load byte (8 bits), zero-extend to 32 bits
- **LDRSB**: Load signed byte, sign-extend to 32 bits

Example Usage

```
LDRH R0, [R1]      ; R0 = 0x0000ABCD (zero-extended)
LDRSH R0, [R1]     ; R0 = 0xFFFFABCD (sign-extended if bit 15 = 1)

LDRB R0, [R1]      ; R0 = 0x000000AB (zero-extended)
LDRSB R0, [R1]     ; R0 = 0xFFFFFFAB (sign-extended if bit 7 = 1)
```

5.2.4 1.4 Hexadecimal Notation

Why Hexadecimal?

- Compact representation of binary
- One hex digit = 4 binary bits
- Easier to read than long binary strings
- Common in programming and debugging

Hex Digits

```
Binary   | Hex | Decimal
--------|-----|--------
0000    |  0  |   0
0001    |  1  |   1
0010    |  2  |   2
0011    |  3  |   3
0100    |  4  |   4
0101    |  5  |   5
0110    |  6  |   6
0111    |  7  |   7
1000    |  8  |   8
1001    |  9  |   9
1010    |  A  |  10
1011    |  B  |  11
1100    |  C  |  12
1101    |  D  |  13
1110    |  E  |  14
1111    |  F  |  15
```

Conversion Examples

```
Binary: 1011 0110 1101 0010
Hex:       B    6    D    2
Result: 0xB6D2

Hex: 0x3F
Binary: 0011 1111
Decimal: 63
```

ARM Hexadecimal Usage

```
MOV R0, #0xFF        ; R0 = 255
MOV R1, #0x100       ; R1 = 256
LDR R2, =0xDEADBEEF  ; R2 = 3735928559
```

5.3 2. ARM Instruction Encoding

5.3.1 2.1 Fixed-Length Instructions

32-Bit Instruction Format

- Every ARM instruction is exactly 32 bits
- Simplifies instruction fetch and decode
- Enables efficient pipelining

Advantages

- Predictable instruction boundaries
- Simple PC increment (always +4)

- Fast decode logic

Trade-offs

- Some instructions may “waste” bits
- Immediate values limited in size
- Code density lower than variable-length (e.g., x86)

5.3.2 2.2 Data Processing Instruction Format

Format Structure

```
[Cond][00][I][Opcode][S][Rn][Rd][Operand2]
 4-bit 2  1   4-bit   1  4   4   12-bit
```

Field Descriptions

Condition (4 bits, bits 28-31)

- Conditional execution feature
- 0000 = EQ (equal, Z=1)
- 0001 = NE (not equal, Z=0)
- 1010 = GE (greater or equal, signed)
- 1110 = AL (always execute, default)

I bit (bit 25)

- 0 = Operand2 is register
- 1 = Operand2 is immediate value

Opcode (4 bits, bits 21-24)

- Specifies operation (AND, EOR, SUB, ADD, etc.)
- 0100 = ADD
- 0010 = SUB
- 0000 = AND
- 1100 = ORR

S bit (bit 20)

- 0 = Don’t update condition flags
- 1 = Update flags (CPSR)

Rn (4 bits, bits 16-19)

- First operand register number
- 0000 = R0, 0001 = R1, etc.

Rd (4 bits, bits 12-15)

- Destination register number

Operand2 (12 bits, bits 0-11)

- If I=0: Shift amount and second register
- If I=1: 8-bit immediate + 4-bit rotation

Example: ADD R0, R1, R2

```
Encoding fields:
- Cond: 1110 (always)
- I: 0 (register operand)
- Opcode: 0100 (ADD)
- S: 0 (don't update flags)
- Rn: 0001 (R1)
- Rd: 0000 (R0)
- Operand2: 0002 (R2, no shift)

Result: 0xE0810002
```

5.3.3 2.3 Data Transfer Instruction Format

Format Structure

```
[Cond][01][I][P][U][B][W][L][Rn][Rd][Offset]
 4-bit 2   1  1  1  1  1  1  4    4   12-bit
```

Key Fields

L bit (bit 20)

- 0 = Store (STR)
- 1 = Load (LDR)

B bit (bit 22)

- 0 = Word transfer (32 bits)
- 1 = Byte transfer (8 bits)

P bit (bit 24)

- 0 = Post-indexed addressing
- 1 = Pre-indexed or offset addressing

U bit (bit 23)

- 0 = Subtract offset from base
- 1 = Add offset to base

W bit (bit 21)

- 0 = No write-back
- 1 = Write-back (update base register)

Rn (base register)

- Contains memory address or base address

Rd (data register)

- For Load: Destination register
- For Store: Source register

Offset (12 bits)

- Memory address offset
- Can be immediate or register

Example: LDR R0, [R1, #4]

```
Encoding fields:
- Cond: 1110 (always)
- L: 1 (load)
- B: 0 (word)
- P: 1 (offset addressing)
- U: 1 (add offset)
- Rn: 0001 (R1)
- Rd: 0000 (R0)
- Offset: 004 (immediate 4)

Result: 0xE5910004
```

5.3.4 2.4 Immediate Value Encoding

Challenge

- 32-bit instruction must fit: opcode, registers, immediate
- Cannot fit full 32-bit immediate

ARM Solution: 8-bit + 4-bit Rotation

- Immediate field: 12 bits total
- Lower 8 bits: Immediate value (0-255)
- Upper 4 bits: Rotation amount (0-15)
- Rotation: Right by (2 × rotation field) bits

Calculation

```
Actual Value = Immediate × ROR (2 × Rotation)
```

Examples

```
Immediate=0xFF, Rotation=0:
  Value = 0xFF ROR 0 = 0x000000FF

Immediate=0xFF, Rotation=8:
  Value = 0xFF ROR 16 = 0x00FF0000

Immediate=0xFF, Rotation=12:
  Value = 0xFF ROR 24 = 0xFF000000
```

Allowed Immediates

- Not all 32-bit values can be encoded
- Valid: 0xFF, 0xFF00, 0xFF0000, 0xFF000000
- Valid: 0xFF000000FF (rotation wraps around)
- Invalid: 0x123 (cannot be formed by rotation)

Assembler Handling

- Assembler checks if immediate is valid
- Gives error if immediate cannot be encoded
- Use LDR pseudo-instruction for arbitrary values:

  ```
  LDR R0, =0x12345678  ; Loads from literal pool
  ```

5.4 3. Logical Operations

5.4.1 3.1 Bitwise AND

Operation

- Performs logical AND on each bit pair
- Result bit = 1 only if both input bits are 1

Truth Table

```
A | B | A AND B
--|---|--------
0 | 0 |   0
0 | 1 |   0
1 | 0 |   0
1 | 1 |   1
```

ARM Instruction

```
AND Rd, Rn, Rm       ; Rd = Rn AND Rm
AND Rd, Rn, #imm     ; Rd = Rn AND immediate
```

Common Uses

Bit Masking (Extract Specific Bits)

```
; Extract lower 8 bits of R1
MOV R0, R1
AND R0, R0, #0xFF    ; R0 = R1 & 0xFF (keep bits 0-7)

; Extract bits 8-15
MOV R0, R1
AND R0, R0, #0xFF00  ; R0 = R1 & 0xFF00 (keep bits 8-15)
```

Clearing Specific Bits

```
; Clear bit 5 of R1
AND R1, R1, #0xFFFFFFDF  ; Bit 5 mask: ~(1 << 5)
```

Checking if Bit Set

```
AND R2, R1, #0x80    ; Check if bit 7 is set
CMP R2, #0           ; Compare with zero
BEQ bit_clear        ; Branch if bit was clear
```

5.4.2 3.2 Bitwise OR

Operation

- Performs logical OR on each bit pair
- Result bit = 1 if either input bit is 1

Truth Table

```
A | B | A OR B
--|---|-------
0 | 0 |   0
0 | 1 |   1
1 | 0 |   1
1 | 1 |   1
```

ARM Instruction

```
ORR Rd, Rn, Rm        ; Rd = Rn OR Rm (ORR in ARM)
ORR Rd, Rn, #imm      ; Rd = Rn OR immediate
```

Common Uses

Setting Specific Bits

```
; Set bit 3 of R1
ORR R1, R1, #0x08     ; Bit 3 mask: (1 << 3) = 0x08

; Set bits 4 and 5
ORR R1, R1, #0x30     ; Mask: 0x30 = 0b00110000
```

Combining Values

```
; Combine lower byte of R1 with upper bytes of R2
AND R1, R1, #0xFF         ; Keep only lower byte
AND R2, R2, #0xFFFFFF00   ; Keep only upper bytes
ORR R0, R1, R2            ; Combine
```

5.4.3 3.3 Bitwise XOR (Exclusive OR)

Operation

- Performs logical XOR on each bit pair
- Result bit = 1 if input bits differ

Truth Table

```
A | B | A XOR B
--|---|--------
0 | 0 |   0
0 | 1 |   1
1 | 0 |   1
1 | 1 |   0
```

ARM Instruction

```
EOR Rd, Rn, Rm       ; Rd = Rn EOR Rm (EOR in ARM)
EOR Rd, Rn, #imm     ; Rd = Rn EOR immediate
```

Common Uses

Toggling Specific Bits

```
; Toggle bit 2 of R1
EOR R1, R1, #0x04    ; Bit 2 mask: (1 << 2)
; If bit was 0, becomes 1; if was 1, becomes 0
```

Fast Zero

```
EOR R0, R0, R0       ; R0 = 0 (XOR with itself)
```

Comparison

```
; Check if R1 and R2 are equal
EOR R3, R1, R2       ; R3 = R1 XOR R2
CMP R3, #0           ; If R3 = 0, R1 == R2
BEQ values_equal
```

Swapping Without Temporary

```
; Swap R0 and R1 without using another register
EOR R0, R0, R1
EOR R1, R0, R1
EOR R0, R0, R1
; Now R0 and R1 are swapped
```

5.4.4 3.4 Bitwise NOT

Operation

- Inverts all bits (0→1, 1→0)
- Also called complement

ARM Instruction

```
MVN Rd, Rm           ; Rd = NOT Rm (Move Not)
MVN Rd, #imm         ; Rd = NOT immediate
```

Common Uses

Creating Bit Masks

```
; Create mask with all bits set except bit 3
MOV R0, #0x08        ; 0x08 = 0b00001000
MVN R1, R0           ; R1 = 0xFFFFFFF7 (all except bit 3)
```

Negation (with ADD)

```
; Negate R1 (two's complement)
MVN R1, R1           ; Invert all bits
ADD R1, R1, #1       ; Add 1
; Now R1 = -R1 (original)
```

5.4.5 3.5 Shift Operations

Logical Shift Left (LSL)

```
LSL Rd, Rn, #shift   ; Rd = Rn << shift
MOV Rd, Rn, LSL #shift
```

- Shifts bits left, fills right with zeros
- Each shift left multiplies by 2
- Example: 0b00001010 LSL 2 = 0b00101000

Logical Shift Right (LSR)

```
LSR Rd, Rn, #shift   ; Rd = Rn >> shift (unsigned)
MOV Rd, Rn, LSR #shift
```

- Shifts bits right, fills left with zeros
- Each shift right divides by 2 (unsigned)
- Example: 0b10100000 LSR 2 = 0b00101000

Arithmetic Shift Right (ASR)

```
ASR Rd, Rn, #shift   ; Rd = Rn >> shift (signed)
```

- Shifts bits right, fills left with sign bit
- Preserves sign for signed division
- Example: 0b11110000 ASR 2 = 0b11111100 (sign preserved)

Rotate Right (ROR)

```
ROR Rd, Rn, #shift   ; Rotate Rn right by shift
```

- Bits shifted out right reappear at left
- No information lost
- Example: 0b10000001 ROR 1 = 0b11000000

Common Shift Applications

Fast Multiplication/Division by Powers of 2

```
LSL R0, R1, #3       ; R0 = R1 × 8 (2^3)
LSR R0, R1, #2       ; R0 = R1 / 4 (unsigned)
ASR R0, R1, #2       ; R0 = R1 / 4 (signed)
```

Bit Extraction

```
; Extract bits 8-11 from R1
LSR R0, R1, #8       ; Shift bits 8-11 to bits 0-3
AND R0, R0, #0xF     ; Mask to keep only 4 bits
```

Bit Positioning

```
; Move bit 0 to bit 7
LSL R0, R1, #7       ; Shift left 7 positions
AND R0, R0, #0x80    ; Keep only bit 7
```

5.5 4. Practical Bit Manipulation Examples

5.5.1 4.1 Extracting Bit Fields

Extract bits 16-23

```
LSR R0, R1, #16      ; Shift right to position
AND R0, R0, #0xFF    ; Mask to 8 bits
```

Extract bits 4-9 (6 bits)

```
LSR R0, R1, #4       ; Shift to position 0
AND R0, R0, #0x3F    ; Mask to 6 bits (0b111111)
```

5.5.2 4.2 Setting and Clearing Bits

Set bits 8-15

```
ORR R1, R1, #0xFF00  ; Set bits 8-15
```

Clear bits 16-23

```
LDR R0, =0xFF00FFFF  ; Mask with bits 16-23 clear
AND R1, R1, R0       ; Clear bits 16-23 of R1
```

Toggle bits 0-7

```
EOR R1, R1, #0xFF    ; Toggle lower byte
```

5.5.3 4.3 Checking Flags

Check if any of bits 4-7 are set

```
AND R2, R1, #0xF0    ; Mask bits 4-7
CMP R2, #0           ; Check if zero
BNE bits_set         ; Branch if any bit was set
```

Check if specific pattern matches

```
; Check if bits 8-11 are 0b1010
LSR R0, R1, #8       ; Position bits
AND R0, R0, #0xF     ; Mask 4 bits
CMP R0, #0xA         ; Compare with 0b1010
BEQ pattern_match
```

5.5.4 4.4 Color Packing/Unpacking

Pack RGB values (8 bits each)

```
; R0 = Red, R1 = Green, R2 = Blue
LSL R1, R1, #8       ; Green << 8
LSL R2, R2, #16      ; Blue << 16
ORR R3, R0, R1       ; Combine Red and Green
ORR R3, R3, R2       ; Combine with Blue
; R3 now contains 0x00BBGGRR
```

Unpack RGB values

```
; R0 contains 0x00BBGGRR
AND R1, R0, #0xFF      ; Extract Red
LSR R2, R0, #8
AND R2, R2, #0xFF      ; Extract Green
LSR R3, R0, #16
AND R3, R3, #0xFF      ; Extract Blue
```

5.6 Key Takeaways

1. **Unsigned binary integers** represent values from 0 to 2^N - 1 using N bits.
2. **Two's complement** represents signed integers, with MSB as sign bit and range -(2^(N-1)) to +(2^(N-1) - 1).
3. **Sign extension** preserves value when expanding narrower signed values to wider registers.
4. **Hexadecimal notation** provides compact representation with one hex digit per 4 binary bits.
5. **ARM instructions are fixed 32-bit length**, simplifying fetch/decode but limiting immediate values.
6. **Data processing format** includes condition, opcode, source/destination registers, and operand.
7. **Data transfer format** specifies load/store, byte/word, addressing mode, and offset.
8. **Immediate encoding** uses 8-bit value + 4-bit rotation, limiting which constants can be encoded directly.
9. **Bitwise AND** used for masking (extracting specific bits) and clearing bits.
10. **Bitwise OR** used for setting specific bits and combining values.
11. **Bitwise XOR** used for toggling bits, fast zero, and comparisons.
12. **Shift operations** enable fast multiplication/division by powers of 2 and bit positioning.
13. **Bit manipulation** is fundamental for low-level programming, hardware control, and optimization.
14. **Understanding encoding** helps write efficient assembly and debug machine code issues.

5.7 Summary

Number representation and instruction encoding form the foundation of low-level programming. Two’s complement enables efficient signed arithmetic with simple hardware, while sign extension preserves values across different data sizes. ARM’s fixed 32-bit instruction format provides regularity but imposes constraints on immediate values, solved through clever encoding schemes. Logical operations—AND, OR, XOR, and NOT—combined with shift operations, provide powerful tools for bit manipulation essential in systems programming, embedded development, and performance optimization. Mastering these concepts enables efficient assembly programming and deeper understanding of how high-level operations translate to machine instructions. These fundamentals prepare us for more complex topics including branching, function calls, and memory management.

Chapter 6

Lecture 6: Branching and Control Flow

6.1 Introduction

Control flow is what distinguishes computers from simple calculators—the ability to make decisions and alter execution based on conditions. This lecture explores conditional operations and branching in ARM assembly, covering comparison instructions, conditional branches, loop implementation, and PC-relative addressing. Understanding these mechanisms is essential for translating high-level control structures (if statements, loops) into assembly code and for comprehending how processors implement dynamic program behavior.

6.2 1. Fundamentals of Conditional Execution

6.2.1 1.1 Decision-Making in Computers

What Makes Computers Powerful

- Ability to make decisions based on data
- Execute different instructions depending on conditions
- Implement if statements, loops, and function calls
- Respond dynamically to input and computed values

Control Flow Concepts

- **Sequential execution**: Default behavior (PC += 4)
- **Conditional branching**: Jump if condition is true
- **Unconditional branching**: Always jump
- **Function calls**: Branch with return address saving

6.2.2 1.2 Program Status Register (PSR)

Status Flags

- **N (Negative)**: Set if result is negative (bit 31 = 1)
- **Z (Zero)**: Set if result is zero
- **C (Carry)**: Set if unsigned overflow occurred
- **V (oVerflow)**: Set if signed overflow occurred

How Flags Are Set

- Comparison instructions (CMP, CMN, TST, TEQ)
- Arithmetic/logic instructions with S suffix (ADDS, SUBS)
- Flags reflect the result of the operation
- Used by subsequent conditional branches

Example

```
CMP R1, R2          ; Compare R1 and R2 (computes R1 - R2)
                     ; Sets flags based on result
```

If R1 = 5, R2 = 3:

- Result of R1 - R2 = 2 (positive, non-zero)
- N = 0 (not negative)
- Z = 0 (not zero)
- C = 1 (no borrow needed)
- V = 0 (no overflow)

6.3 2. Comparison Instructions

6.3.1 2.1 Compare (CMP)

Syntax

```
CMP Rn, Rm          ; Compare Rn with Rm
CMP Rn, #imm        ; Compare Rn with immediate
```

Operation

- Performs Rn - Rm (subtraction)
- Updates PSR flags based on result
- Does NOT store the result
- Does NOT modify any register

Example Usage

```
MOV R1, #10
MOV R2, #5
CMP R1, R2          ; Compares 10 with 5
                     ; Result: 10 - 5 = 5 (positive, non-zero)
                     ; Z = 0, N = 0
```

6.3.2 2.2 Compare Negative (CMN)

Syntax

```
CMN Rn, Rm          ; Compare Negative
CMN Rn, #imm
```

Operation

- Performs Rn + Rm (addition)
- Updates PSR flags
- Equivalent to CMP Rn, -Rm
- Useful for checking if sum equals zero

6.3.3 2.3 Test (TST)

Syntax

```
TST Rn, Rm          ; Test bits
TST Rn, #imm
```

Operation

- Performs Rn AND Rm (bitwise AND)
- Updates PSR flags
- Result not stored
- Used to test if specific bits are set

Example: Check if bit 5 is set

```
TST R1, #0x20       ; Test bit 5
BEQ bit_clear       ; Branch if bit was clear (Z=1)
```

6.3.4 2.4 Test Equivalence (TEQ)

Syntax

```
TEQ Rn, Rm          ; Test Equivalence
TEQ Rn, #imm
```

Operation

- Performs Rn XOR Rm (exclusive OR)
- Updates PSR flags
- Z=1 if values are equal
- Used to compare values without affecting C or V flags

6.4 3. Conditional Branch Instructions

6.4.1 3.1 Branch if Equal (BEQ)

Syntax

```
BEQ label            ; Branch if equal (Z=1)
```

Condition

- Branches if Zero flag is set (Z = 1)
- Typically used after CMP to check equality

Example

```
CMP R1, R2           ; Compare R1 and R2
BEQ equal_label      ; Jump to equal_label if R1 == R2
; Code if not equal
equal_label:
; Code if equal
```

6.4.2 3.2 Branch if Not Equal (BNE)

Syntax

```
BNE label            ; Branch if not equal (Z=0)
```

Condition

- Branches if Zero flag is clear (Z = 0)
- Opposite of BEQ

Example

```
CMP R3, #0
BNE not_zero         ; Jump if R3 != 0
; Code if R3 is zero
not_zero:
; Code if R3 is non-zero
```

6.4.3 3.3 Signed Comparison Branches

Branch if Greater or Equal (BGE)

```
BGE label            ; Branch if Rn >= Rm (signed)
                      ; Condition: N == V
```

Branch if Less Than (BLT)

```
BLT label            ; Branch if Rn < Rm (signed)
                      ; Condition: N != V
```

Branch if Greater Than (BGT)

```
BGT label            ; Branch if Rn > Rm (signed)
                      ; Condition: Z==0 AND N==V
```

Branch if Less or Equal (BLE)

```
BLE label            ; Branch if Rn <= Rm (signed)
                      ; Condition: Z==1 OR N!=V
```

Example

```
CMP R1, R2
BGE greater_equal    ; Branch if R1 >= R2 (signed)
; Code if R1 < R2
greater_equal:
; Code if R1 >= R2
```

6.4.4 3.4 Unsigned Comparison Branches

Branch if Higher or Same (BHS) (also called BCS - Branch if Carry Set)

```
BHS label            ; Branch if Rn >= Rm (unsigned)
                      ; Condition: C == 1
```

Branch if Lower (BLO) (also called BCC - Branch if Carry Clear)

```
BLO label            ; Branch if Rn < Rm (unsigned)
                      ; Condition: C == 0
```

Branch if Higher (BHI)

```
BHI label            ; Branch if Rn > Rm (unsigned)
                      ; Condition: C==1 AND Z==0
```

Branch if Lower or Same (BLS)

```
BLS label            ; Branch if Rn <= Rm (unsigned)
                      ; Condition: C==0 OR Z==1
```

6.4.5 3.5 Signed vs. Unsigned Example

Key Difference

```
MOV R0, #0xFFFFFFFF  ; R0 = -1 (signed) or 4,294,967,295 (unsigned)
MOV R1, #1           ; R1 = 1
CMP R0, R1

BLO lower_unsigned   ; BRANCH NOT TAKEN
                      ; Unsigned: 4,294,967,295 > 1

BLT less_signed      ; BRANCH TAKEN
                      ; Signed: -1 < 1
```

When to Use Each

- **Signed**: Comparing integers that can be negative (temperatures, offsets, differences)
- **Unsigned**: Comparing addresses, array indices, sizes, counts

6.4.6 3.6 Unconditional Branch

Syntax

```
B label              ; Branch always
```

Purpose

- Jump without checking any condition
- Skip code sections
- Implement infinite loops
- Return to loop start

Example

```
B end                ; Skip this section
; Code to skip
end:
; Continue execution here
```

6.5 4. Labels in Assembly

6.5.1 4.1 Label Definition

Purpose

- Mark specific instruction locations
- Provide symbolic names for addresses
- Enable branches and data references

Syntax

```
label:               ; Label definition (note colon)
    MOV R0, #1      ; Instruction at this label
```

Naming Rules

- Can be almost any identifier
- Common conventions: loop, exit, done, L1, L2
- Cannot conflict with instruction mnemonics
- Case-sensitive

Example

```
start:
    MOV R0, #0
loop:
    ADD R0, R0, #1
    CMP R0, #10
    BLT loop         ; Branch to loop label
    B start          ; Branch to start label
```

6.5.2 4.2 Label Resolution

Assembly Process

1. First pass: Record label addresses

2. Second pass: Replace labels with addresses
3. Calculate offsets for PC-relative branches

Virtual Addresses

- Assembler assigns virtual addresses from 0
- First instruction: address 0
- Second instruction: address 4
- Third instruction: address 8
- Physical addresses determined at load time

6.6 5. Implementing Control Structures

6.6.1 5.1 If Statement

C Code

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

ARM Assembly (Method 1: Branch on False)

```
    CMP R3, R4        ; Compare i (R3) and j (R4)
    BNE else          ; Branch to else if not equal
    ADD R0, R1, R2    ; f = g + h (then clause)
    B exit            ; Skip else clause
else:
    SUB R0, R1, R2    ; f = g - h (else clause)
exit:
    ; Continue...
```

ARM Assembly (Method 2: Conditional Execution)

```
    CMP R3, R4        ; Compare i and j
    ADDEQ R0, R1, R2 ; f = g + h (executed only if equal)
    SUBNE R0, R1, R2 ; f = g - h (executed only if not equal)
```

6.6.2 5.2 If-Else Ladder

C Code

```
if (x < 0)
    result = -1;
else if (x == 0)
    result = 0;
else
    result = 1;
```

ARM Assembly

```
    CMP R1, #0       ; Compare x with 0
    BLT negative     ; Branch if x < 0
    BEQ zero         ; Branch if x == 0
    ; x > 0
    MOV R0, #1
    B done
negative:
    MOV R0, #-1
    B done
zero:
    MOV R0, #0
done:
    ; Continue...
```

6.6.3 5.3 While Loop

C Code

```
while (i < n) {
    sum += i;
    i++;
}
```

ARM Assembly

```
loop:
    CMP R1, R2       ; Compare i (R1) with n (R2)
    BGE end_loop     ; Exit if i >= n
    ADD R0, R0, R1   ; sum = sum + i
    ADD R1, R1, #1   ; i++
    B loop           ; Branch back to loop start
end_loop:
    ; Continue...
```

6.6.4 5.4 For Loop

C Code

```
for (i = 0; i < 10; i++) {
    sum += i;
}
```

ARM Assembly

```
    MOV R1, #0       ; i = 0 (initialization)
for_loop:
    CMP R1, #10      ; Compare i with 10
    BGE end_for      ; Exit if i >= 10
    ADD R0, R0, R1   ; sum = sum + i (loop body)
    ADD R1, R1, #1   ; i++ (increment)
    B for_loop       ; Branch back to loop start
```

```
end_for:
    ; Continue...
```

6.6.5 5.5 Do-While Loop

C Code

```
do {
    sum += i;
    i++;
} while (i < n);
```

ARM Assembly

```
do_loop:
    ADD R0, R0, R1    ; sum = sum + i (loop body first)
    ADD R1, R1, #1    ; i++
    CMP R1, R2        ; Compare i with n
    BLT do_loop       ; Branch back if i < n
    ; Continue...
```

Key Difference from While

- Body executes at least once
- Condition checked at end, not beginning

6.7 6. Array Access in Loops

6.7.1 6.1 Static Array Indexing

C Code

```
while (save[i] == k)
    i++;
```

ARM Assembly

```
    ; R6 = base address of save array
    ; R3 = i (index)
    ; R5 = k (comparison value)
loop:
    ADD R12, R6, R3, LSL #2  ; address = base + (i * 4)
    LDR R0, [R12, #0]        ; R0 = save[i]
    CMP R0, R5               ; Compare save[i] with k
    BNE exit                 ; Exit if not equal
    ADD R3, R3, #1           ; i++
    B loop                   ; Continue loop
exit:
    ; Continue...
```

Dynamic Offset Calculation

- `R3, LSL #2` means R3 × 4 (shift left 2 = multiply by 4)
- Words are 4 bytes, so array element i is at base + (i × 4)
- Efficient: shift is faster than multiplication

6.7.2 6.2 Array Traversal

C Code

```
int sum = 0;
for (int i = 0; i < 10; i++) {
    sum += arr[i];
}
```

ARM Assembly

```
    LDR R6, =arr      ; R6 = base address of array
    MOV R0, #0        ; sum = 0
    MOV R1, #0        ; i = 0
loop:
    CMP R1, #10
    BGE done
    ADD R12, R6, R1, LSL #2  ; address = base + i*4
    LDR R2, [R12]            ; R2 = arr[i]
    ADD R0, R0, R2           ; sum += arr[i]
    ADD R1, R1, #1           ; i++
    B loop
done:
    ; R0 contains sum
```

6.8 7. PC-Relative Addressing

6.8.1 7.1 Branch Instruction Encoding

32-Bit Format

```
[Cond][1010][Offset]
 4-bit 4-bit 24-bit
```

Fields

- **Cond**: Condition code (EQ, NE, LT, etc.)
- **1010**: Fixed format field for branch
- **Offset**: 24-bit signed offset

6.8.2 7.2 Address Calculation

Problem with Absolute Addressing

- 24 bits can address 2^{24} = 16 MB
- Limits program size to 16 MB

- Fixed addresses complicate relocation

PC-Relative Solution

- Store offset from current PC, not absolute address
- Target = PC + offset
- Can branch ±16 MB from current instruction
- Total program can exceed 16 MB

Offset Calculation

```
Offset = (Target Address - PC) / 4
```

Why Divide by 4?

- All instructions are 4-byte aligned
- Least significant 2 bits always 00
- Omit these bits in encoding
- Effective range: ±64 MB (24-bit offset × 4)

Example

```
Current PC: 0x1000
Target: 0x1020
Offset = (0x1020 - 0x1000) / 4 = 0x20 / 4 = 8 instructions

Encoded offset in branch instruction: 8
At execution: PC = 0x1000 + (8 × 4) = 0x1020
```

6.8.3 7.3 Advantages of PC-Relative

Position-Independent Code

- Code can load at any address
- Branches remain correct regardless of location
- Essential for libraries and shared code

Simplified Linking

- Linker doesn't need to patch all branches
- Only external function calls need adjustment

Branch Locality

- Most branches are to nearby instructions
- PC-relative naturally handles this case
- Absolute addressing wastes bits for nearby targets

6.9 8. Conditional Execution (Alternative to Branching)

6.9.1 8.1 Conditional Instruction Suffixes

Concept

- Add condition code to instruction mnemonic
- Instruction executes only if condition is true
- Otherwise, instruction is skipped (NOP)

Available Suffixes

- EQ (equal), NE (not equal)
- GT, LT, GE, LE (signed comparisons)
- HI, LO, HS, LS (unsigned comparisons)
- Many others (see ARM documentation)

Examples

```
CMP R1, R2
ADDEQ R0, R3, R4      ; Execute ADD only if R1 == R2
SUBNE R0, R3, R4      ; Execute SUB only if R1 != R2
MOVGT R5, #10         ; Execute MOV only if R1 > R2
```

6.9.2 8.2 Conditional Execution Example

C Code

```
if (a == b)
    max = a;
else
    max = b;
```

Method 1: Branching

```
    CMP R1, R2        ; Compare a and b
    BNE else
    MOV R0, R1        ; max = a
    B done
else:
    MOV R0, R2        ; max = b
done:
```

Method 2: Conditional Execution

```
    CMP R1, R2        ; Compare a and b
    MOVEQ R0, R1      ; max = a (if equal)
    MOVNE R0, R2      ; max = b (if not equal)
```

6.9.3 8.3 Advantages and Limitations

Advantages

- More compact code (fewer instructions)
- No branch misprediction penalty
- Faster for simple conditions
- Clearer intent in some cases

Limitations

- Only works for simple, short sequences
- Cannot conditionally execute blocks of code
- All conditional instructions must fit in pipeline
- May execute both paths (but discard one result)

When to Use

- Simple assignments
- Min/max operations
- Short computations with single result
- Performance-critical paths where branches hurt

6.10 9. Basic Blocks

6.10.1 9.1 Definition

Basic Block Characteristics

- Sequence of instructions with:
  - No embedded branches (except possibly at end)
  - No branch targets (except possibly at beginning)
- Executed atomically: all or nothing
- Single entry point, single exit point

Example

```
; Basic Block 1 (entry point)
    MOV R0, #0
    MOV R1, #10
    CMP R1, #10
    BNE block2       ; Exit point of block 1

; Basic Block 2 (entry and exit point)
block2:
    ADD R0, R0, #1
    CMP R0, R1
    BLT block2       ; Exit point of block 2
```

6.10.2 9.2 Importance in Compilation

Compiler Optimizations

- Identify basic blocks for analysis
- Optimize within blocks (register allocation, scheduling)
- Build control flow graph from blocks
- Apply inter-block optimizations

Processor Optimizations

- Predict block execution
- Prefetch instructions in block

- Schedule instructions more aggressively
- Reduce branch overhead

6.11 Key Takeaways

1. **Conditional execution** distinguishes computers from calculators, enabling decision-making and dynamic behavior.
2. **CMP instruction** sets PSR flags by performing subtraction without storing the result.
3. **Conditional branches** (BEQ, BNE, BGE, BLT, etc.) check PSR flags to decide whether to jump.
4. **Signed vs. unsigned branches** interpret the same bit patterns differently based on context.
5. **Labels** provide symbolic names for addresses, enabling readable branch targets.
6. **If statements** translate to compare + conditional branch + unconditional branch to skip alternate path.
7. **Loops** use compare + conditional branch (to exit) + unconditional branch (to continue).
8. **Array access** in loops uses dynamic offset calculation with shifts (LSL #2 for word arrays).
9. **PC-relative addressing** stores branch offset from current PC, enabling position-independent code and large programs.
10. **Word-based offsets** effectively quadruple branch range by encoding instruction count instead of byte offset.
11. **Conditional execution** provides alternative to branching for simple cases, improving performance and code density.
12. **Basic blocks** are atomic instruction sequences used by compilers and processors for optimization.
13. **Branch locality** means most branches target nearby instructions, making PC-relative addressing natural and efficient.

6.12 Summary

Branching and conditional execution form the foundation of program control flow, translating high-level constructs like if statements and loops into machine instructions. The ARM architecture provides a rich set of conditional branches for both signed and unsigned comparisons, enabling efficient implementation of diverse control structures. Understanding the distinction between comparison (which sets flags) and branching (which checks flags) is essential for correct assembly programming. PC-relative addressing solves program size limitations while enabling position-independent code, and conditional execution offers a performant alternative to branching for simple cases. Mastering these concepts is crucial for translating algorithms into assembly code, optimizing performance-critical sections, and understanding how processors implement dynamic program behavior. These fundamentals prepare us for more advanced topics including function calls, stack management, and processor pipelining.

Chapter 7

Lecture 7: Function Call and Return

7.1 Introduction

Function calling is a fundamental mechanism that enables modular programming and code reuse. This lecture explores how ARM assembly implements function calls, covering parameter passing, return value handling, the call stack, register preservation conventions, and recursion. Understanding these mechanisms is essential for translating high-level function-based programs into assembly and for comprehending how processors manage execution context across function boundaries.

7.2 1. Function Calling Fundamentals

7.2.1 1.1 Function Calling Steps

Complete Call Sequence

1. **Place parameters** in argument registers (R0-R3)
2. **Transfer control** to callee function using BL
3. **Acquire stack storage** for temporary values
4. **Back up registers** that need preservation (R4-R11)
5. **Perform function operations** (the actual work)
6. **Place result** in return register (R0)
7. **Restore backed-up registers** from stack
8. **Return to caller** using MOV PC, LR

Why This Complexity?

- Enables nested and recursive function calls
- Protects caller's data in registers
- Provides local storage for function variables
- Supports arbitrary call depth

7.2.2 1.2 Why Use Functions?

Benefits

- **Code reuse**: Write once, call many times
- **Modularity**: Break complex problems into manageable pieces
- **Abstraction**: Hide implementation details
- **Maintainability**: Easier to debug and modify

Example

```
int add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(5, 3);  // Function call
}
```

7.3 2. ARM Register Conventions

7.3.1 2.1 Register Usage Rules

Register Classification

```
R0-R1:    Arguments and return results
          - Caller does NOT expect these preserved
          - Scratch registers

R2-R3:    Additional arguments
          - Also scratch registers
          - Caller does NOT expect preservation

R4-R11:   Local variables
          - MUST be preserved across function calls
          - Callee saves if it uses these registers

R12:      Intra-procedure-call scratch register
          - Can be corrupted by function calls
          - Not preserved

R13 (SP): Stack Pointer
          - Points to top of stack
          - MUST always be valid

R14 (LR): Link Register
          - Stores return address
          - Set by BL instruction

R15 (PC): Program Counter
```

```
         - Next instruction address
         - Modified to return from function
```

7.3.2 2.2 Shared Register File

Key Concept

- ALL functions share the SAME 16 registers
- No separate register sets per function
- Registers are a shared resource requiring careful management

Implications

- Functions must coordinate register usage
- Conventions prevent conflicts
- Callee must preserve certain registers (R4-R11)
- Caller can assume R4-R11 unchanged after call

Example Scenario

```
main:
    MOV R4, #10      ; main uses R4
    MOV R0, #5       ; Pass argument
    BL function      ; Call function
    ; R4 still contains 10 (guaranteed)
    ADD R5, R4, R0   ; Use preserved R4 and return value

function:
    ; Must preserve R4 if we use it
    ; Can freely modify R0-R3, R12
    MOV R0, #20      ; Return value
    MOV PC, LR       ; Return
```

7.4 3. Function Call Instructions

7.4.1 3.1 Branch and Link (BL)

Syntax

```
BL function_label    ; Branch and Link
```

Operation

1. **Save return address**: LR = address of next instruction
2. **Jump to function**: PC = function_label address

Example

```
    MOV R0, #10      ; Address: 0x1000
    BL fun           ; Address: 0x1004
    ADD R1, R0, #5   ; Address: 0x1008 (return point)
```

```
fun:
    ; LR contains 0x1008 (address after BL)
    ; Function code here
    MOV PC, LR          ; Return to 0x1008
```

Why "Link"?

- Creates a "link" back to caller
- LR provides the connection
- Enables function to return

7.4.2 3.2 Return from Function

Basic Return

```
MOV PC, LR              ; Copy LR to PC
```

Operation

- PC = LR (jump to return address)
- Execution continues at instruction after BL
- Simple and fast

Alternative (older ARM)

```
BX LR                   ; Branch and Exchange
```

7.5 4. Parameter Passing

7.5.1 4.1 Using R0-R3

Convention

- First 4 arguments in R0-R3
- Arguments loaded before BL instruction
- Callee reads R0-R3 to get parameters

Example: Two Parameters

```
int multiply(int a, int b) {
    return a * b;
}

int result = multiply(6, 7);
```

ARM Assembly

```
    MOV R0, #6          ; First argument (a)
    MOV R1, #7          ; Second argument (b)
    BL multiply         ; Call function
    ; R0 now contains result (42)

multiply:
    MUL R0, R0, R1    ; R0 = R0 × R1
    MOV PC, LR        ; Return
```

7.5.2 4.2 More Than 4 Arguments

Solution: Use Stack

- Arguments 1-4 in R0-R3
- Additional arguments pushed to stack
- Callee reads from stack

Example: 6 Arguments

```
int sum6(int a, int b, int c, int d, int e, int f) {
    return a + b + c + d + e + f;
}
```

ARM Assembly

```
    MOV R0, #1        ; arg1
    MOV R1, #2        ; arg2
    MOV R2, #3        ; arg3
    MOV R3, #4        ; arg4
    MOV R4, #5
    MOV R5, #6
    SUB SP, SP, #8    ; Space for 2 more args
    STR R4, [SP, #0]  ; arg5 on stack
    STR R5, [SP, #4]  ; arg6 on stack
    BL sum6
    ADD SP, SP, #8    ; Clean up stack

sum6:
    ; R0-R3 have first 4 args
    ; Load arg5 and arg6 from stack
    LDR R4, [SP, #0]  ; arg5
    LDR R5, [SP, #4]  ; arg6
    ADD R0, R0, R1
    ADD R0, R0, R2
    ADD R0, R0, R3
    ADD R0, R0, R4
    ADD R0, R0, R5
    MOV PC, LR
```

7.6 5. Return Values

7.6.1 5.1 Primary Return Register (R0)

Convention

- Result placed in R0

- Caller reads R0 after function returns
- Works for 32-bit values

Example

```
add:
    ADD R0, R0, R1    ; R0 = R0 + R1
    MOV PC, LR        ; Return with result in R0

main:
    MOV R0, #10
    MOV R1, #20
    BL add            ; Call function
    ; R0 now contains 30
```

7.6.2 5.2 64-Bit Return Values

Convention

- Lower 32 bits in R0
- Upper 32 bits in R1
- Example: 64-bit integer or two 32-bit values

Example

```
long long multiply64(int a, int b) {
    return (long long)a * b;
}
```

ARM Assembly

```
multiply64:
    SMULL R0, R1, R0, R1   ; Signed multiply long
    ; R0 = lower 32 bits
    ; R1 = upper 32 bits
    MOV PC, LR
```

7.7 6. The Stack

7.7.1 6.1 Stack Structure

Definition

- Last In, First Out (LIFO) data structure
- Part of main memory
- Used for temporary storage

Characteristics

- **Starts at high address**: Top of memory
- **Grows downward**: Toward lower addresses
- **Stack Pointer (SP/R13)**: Points to top of stack

- **Dynamic size**: Grows and shrinks as needed

Memory Layout

```
High Address

    Stack    ← SP points here
     ↓         (grows downward)


    Heap
     ↑         (grows upward)

    Data       (static variables)

    Text       (instructions)

Low Address
```

7.7.2 6.2 Stack Uses

Primary Purposes

1. **Saving register values** (preserve R4-R11)
2. **Storing local variables** (arrays, structures)
3. **Preserving return addresses** (nested calls)
4. **Extra function arguments** (beyond R0-R3)
5. **Storing local arrays** that don't fit in registers

7.8 7. Stack Operations

7.8.1 7.1 Allocating Stack Space (Pushing)

Decrement Stack Pointer

```
SUB SP, SP, #4       ; Allocate 4 bytes (1 register)
SUB SP, SP, #12      ; Allocate 12 bytes (3 registers)
```

Why Subtract?

- Stack grows toward lower addresses
- Allocating space moves SP downward
- Each 32-bit register needs 4 bytes

7.8.2 7.2 Storing Values to Stack

Single Register

```
SUB SP, SP, #4       ; Allocate space
STR R4, [SP, #0]     ; Store R4 at top of stack
```

Multiple Registers

```
SUB SP, SP, #12      ; Space for 3 registers
STR R4, [SP, #0]     ; Store R4
STR R5, [SP, #4]     ; Store R5
STR R6, [SP, #8]     ; Store R6
```

Push Multiple (Convenient)

```
PUSH {R4-R6}         ; Allocate and store in one instruction
```

7.8.3 7.3 Loading Values from Stack

Single Register

```
LDR R4, [SP, #0]     ; Load R4 from stack
ADD SP, SP, #4       ; Release space
```

Multiple Registers

```
LDR R4, [SP, #0]     ; Restore R4
LDR R5, [SP, #4]     ; Restore R5
LDR R6, [SP, #8]     ; Restore R6
ADD SP, SP, #12      ; Release space
```

Pop Multiple

```
POP {R4-R6}          ; Restore and release in one instruction
```

7.8.4 7.4 Stack Space Lifecycle

Pattern

1. **Allocate**: SUB SP, SP, #n
2. **Use**: STR/LDR with [SP, offset]
3. **Release**: ADD SP, SP, #n

Important: Balance

- Every SUB must have corresponding ADD
- Unbalanced stack causes bugs and crashes
- SP must be restored before return

7.9 8. Register Preservation

7.9.1 8.1 Why Preserve R4-R11?

Problem

- All functions share same registers
- Main function may be using R4-R11
- Called function needs registers for its work
- Must not corrupt caller's data

Solution

- Callee saves R4-R11 to stack at function start
- Uses registers freely during execution
- Restores R4-R11 from stack before return
- Caller expects R4-R11 unchanged

7.9.2 8.2 Preservation Pattern

Function Template

```
function:
    ; Prologue: Save registers
    SUB SP, SP, #12      ; Allocate space
    STR R4, [SP, #0]     ; Save R4
    STR R5, [SP, #4]     ; Save R5
    STR R6, [SP, #8]     ; Save R6

    ; Function body: Use R4-R6 freely
    ; ...

    ; Epilogue: Restore registers
    LDR R4, [SP, #0]     ; Restore R4
    LDR R5, [SP, #4]     ; Restore R5
    LDR R6, [SP, #8]     ; Restore R6
    ADD SP, SP, #12      ; Release space
    MOV PC, LR           ; Return
```

Optimization

- Only preserve registers actually used
- If function doesn't use R5, don't save/restore it
- Saves stack space and execution time

7.10 9. Nested Function Calls (Non-Leaf Functions)

7.10.1 9.1 The Problem

Leaf Function

- Doesn't call other functions
- LR preserved automatically (not overwritten)
- Simple return: MOV PC, LR

Non-Leaf Function

- Calls other functions
- BL overwrites LR with new return address
- Original LR lost!
- Cannot return to original caller

Example Problem

```
main:
    BL funcA         ; LR = address after this BL

funcA:
    ; LR contains return address to main
    BL funcB         ; LR OVERWRITTEN with return to funcA!
    MOV PC, LR       ; Returns to funcA, not main (WRONG!)

funcB:
    MOV PC, LR       ; Correctly returns to funcA
```

7.10.2 9.2 Solution: Save LR to Stack

Pattern

```
function:
    ; Save LR first!
    SUB SP, SP, #4
    STR LR, [SP, #0]

    ; Now safe to call other functions
    BL other_function

    ; Restore LR before return
    LDR LR, [SP, #0]
    ADD SP, SP, #4
    MOV PC, LR
```

Complete Example

```
main:
    MOV R0, #5
    BL outer         ; LR = return_to_main
    ; Execution returns here

outer:
    SUB SP, SP, #4
    STR LR, [SP, #0] ; Save LR (return_to_main)

    MOV R1, R0
    ADD R0, R0, #10
    BL inner         ; LR = return_to_outer (overwrites!)

    ADD R0, R0, R1
    LDR LR, [SP, #0] ; Restore LR (return_to_main)
    ADD SP, SP, #4
    MOV PC, LR       ; Returns to main
```

```
inner:
    MUL R0, R0, R0
    MOV PC, LR        ; Returns to outer
```

7.11 10. Recursion Example: Factorial

7.11.1 10.1 Factorial Function

C Code

```
int fact(int n) {
    if (n <= 1)
        return 1;
    else
        return n * fact(n-1);
}
```

Key Points

- Base case: n ≤ 1, return 1
- Recursive case: return n × fact(n-1)
- Each call creates new stack frame
- Stack unwinds as recursion returns

7.11.2 10.2 ARM Assembly Implementation

```
fact:
    ; Save LR and n
    SUB SP, SP, #8
    STR LR, [SP, #4]      ; Save return address
    STR R0, [SP, #0]      ; Save n

    ; Base case: if (n <= 1) return 1
    CMP R0, #1
    BGT recursive
    MOV R0, #1            ; Return 1
    B fact_end

recursive:
    ; Recursive case: n * fact(n-1)
    SUB R0, R0, #1        ; n-1
    BL fact               ; fact(n-1)
    LDR R1, [SP, #0]      ; Restore original n
    MUL R0, R0, R1        ; n * fact(n-1)

fact_end:
    ; Restore and return
    LDR LR, [SP, #4]
    ADD SP, SP, #8
    MOV PC, LR
```

7.11.3 10.3 Stack Growth During Recursion

Call: fact(3)

```
Initial: SP = 0x1000

fact(3) call:
  SP = 0x0FF8: [LR_main, 3]

fact(2) call:
  SP = 0x0FF0: [LR_fact3, 2]

fact(1) call:
  SP = 0x0FE8: [LR_fact2, 1]

Base case returns 1
Unwinds to fact(2): returns 1*2 = 2
Unwinds to fact(3): returns 2*3 = 6
Returns to main with result 6

Final: SP = 0x1000 (restored)
```

Stack Space Per Call

- 8 bytes (LR + n)
- fact(5) needs 5 × 8 = 40 bytes
- fact(10) needs 80 bytes
- Deep recursion can overflow stack!

7.12 11. Memory Layout and Stack vs. Heap

7.12.1 11.1 Complete Memory Layout

```
High Address (0xFFFFFFFF)

     Reserved     OS and system

      Stack       ← SP (grows down)
        ↓           Automatic storage
                    Function call data
                    Local variables


     (unused)


        ↑
      Heap          Dynamic allocation
```

```
                    malloc/free, new/delete

    Static Data     Global variables
                    String constants

       Text         Program instructions
    (Code)          Read-only

Low Address (0x00000000)
```

7.12.2 11.2 Stack Characteristics

Automatic Storage

- Allocated when function called
- Released when function returns
- Managed automatically by compiler/runtime

Fast Access

- Fixed addressing pattern
- SP always points to top
- Simple offset calculations

Limited Size

- Typically 1-8 MB
- Stack overflow if exceeded
- Recursion depth limited

Scope

- Local to function
- Not accessible after return
- Perfect for temporary data

7.12.3 11.3 Heap Characteristics

Dynamic Allocation

- malloc/free in C
- new/delete in C++
- Programmer controls lifetime

Flexible Size

- Can grow large (limited by available memory)
- Variable-sized allocations

Manual Management

- Must explicitly free memory
- Memory leaks if not freed
- Fragmentation possible

Global Scope

- Persists until explicitly freed
- Can pass pointers across functions
- Suitable for data structures

7.13 Key Takeaways

1. **Function calling requires** parameter passing, return value handling, and register preservation.
2. **R0-R3 for arguments and returns** - caller doesn't expect preservation.
3. **R4-R11 must be preserved** by callee if used, protecting caller's data.
4. **BL instruction** saves return address in LR and jumps to function.
5. **Return via MOV PC, LR** copies link register to program counter.
6. **Stack is LIFO structure** growing downward from high addresses, pointed to by SP.
7. **Stack usage** includes saving registers, local variables, return addresses, and extra arguments.
8. **Allocate with SUB SP, release with ADD SP** - must balance allocations and releases.
9. **Non-leaf functions** must save LR to stack before making nested calls.
10. **Recursion** creates multiple stack frames, one per call, unwinding as calls return.
11. **Stack vs. Heap** - stack is automatic/local/fast/limited, heap is manual/global/flexible/larger.
12. **Register conventions** enable modularity and prevent conflicts in shared register file.

7.14 Summary

Function calling mechanisms enable modular programming by providing structured ways to pass control, data, and return values between code sections. ARM's register conventions balance efficiency (passing arguments in registers) with safety (preserving callee-saved registers). The stack provides essential temporary storage for register preservation, local variables, and handling nested calls including recursion. Understanding these mechanisms is crucial for translating high-level function-based code to assembly, optimizing performance, and debugging stack-related issues. The interplay between registers, stack, and calling conventions forms the foundation for understanding how real programs execute, preparing us for more advanced topics like exception handling, operating systems, and compiler optimization.

Chapter 8

Lecture 8: Memory Access and String Operations

8.1 Introduction

This lecture explores character data handling, string operations, and the compilation/linking/loading process. We examine byte and half-word memory operations, implement string manipulation functions, use library functions like scanf and printf, and understand how programs transform from source code to executable binaries. These topics bridge high-level programming concepts and low-level assembly implementation, essential for systems programming and understanding program execution.

8.2 1. Character Data and Encoding

8.2.1 1.1 ASCII Encoding

Basic 7-Bit Standard

- Represents 128 characters using 7 bits ($2^7 = 128$)
- 95 graphic symbols (printable): A-Z, a-z, 0-9, punctuation
- 33 control symbols: newline ('\n'), tab ('\t'), null ('\0')
- Most basic and widely used encoding

ASCII Examples

```
'A' = 65 (0x41)
'a' = 97 (0x61)
'0' = 48 (0x30)
'\n' = 10 (0x0A)
'\0' = 0 (0x00) - null terminator
```

8.2.2 1.2 Latin-1 Encoding

Extended 8-Bit Standard

- Supports 256 characters using 8 bits ($2^8 = 256$)
- Includes all ASCII characters (first 128)
- Adds 96 additional graphic characters
- European language support (accented characters)

8.2.3 1.3 Unicode Encoding

Modern Universal Standard

- Uses 32-bit character set (2^{32} possible characters)
- Can represent most world alphabets and symbols
- Used in modern languages (Java, C++, Python 3)
- Variable-length encodings: UTF-8, UTF-16
- UTF-8: 1-4 bytes per character (backward compatible with ASCII)

Why Unicode?

- Global language support
- Emoji and special symbols
- Mathematical and technical symbols
- Historical scripts and languages

8.3 2. Byte Load/Store Operations

8.3.1 2.1 Load Register Byte (LDRB)

Syntax

```
LDRB Rd, [Rn, #offset]   ; Load byte from memory
```

Operation

- Reads 8 bits (1 byte) from memory
- Fills upper 24 bits of register with zeros (zero-extension)
- Lower 8 bits contain the loaded byte

Example

```
; Memory[0x1000] = 0x42 ('B')
LDR R1, =0x1000
LDRB R0, [R1]
; R0 = 0x00000042
```

Use Cases

- Loading single characters
- Reading byte arrays
- Accessing packed data structures
- I/O port access

8.3.2 2.2 Store Register Byte (STRB)

Syntax

```
STRB Rd, [Rn, #offset]   ; Store byte to memory
```

Operation

- Writes lower 8 bits of register to memory
- Upper 24 bits of register ignored
- Only affects 1 byte in memory

Example

```
MOV R0, #0x41        ; 'A'
LDR R1, =0x2000
STRB R0, [R1]        ; Memory[0x2000] = 0x41
```

8.3.3 2.3 Load Register Signed Byte (LDRSB)

Syntax

```
LDRSB Rd, [Rn, #offset]  ; Load signed byte
```

Operation

- Loads 8 bits from memory
- Replicates sign bit (bit 7) to fill upper 24 bits
- Sign-extension preserves signed value

Example

```
; Memory[0x1000] = 0xFE (-2 in signed byte)
LDR R1, =0x1000
LDRSB R0, [R1]
; R0 = 0xFFFFFFFE (-2 in 32-bit signed)

; Memory[0x1001] = 0x7F (+127)
LDRSB R0, [R1, #1]
; R0 = 0x0000007F (+127)
```

When to Use

- Loading signed characters (int8_t)
- Temperature values
- Signed offsets or deltas

8.3.4 2.4 Memory Alignment

LDRB Advantages

- Can access ANY byte address
- No alignment requirement
- Example: addresses 0, 1, 2, 3, 4, 5…

LDR Requirement

- Must use word-aligned addresses (multiples of 4)
- Valid addresses: 0, 4, 8, 12, 16…
- Invalid: 1, 2, 3, 5, 6, 7, 9…
- Unaligned access causes errors or performance penalties

8.4 3. Half-Word Load/Store Operations

8.4.1 3.1 Load Register Half-word (LDRH)

Syntax

```
LDRH Rd, [Rn, #offset]   ; Load 16 bits
```

Operation

- Loads 16 bits (2 bytes) from memory
- Fills upper 16 bits with zeros (zero-extension)

Example

```
; Memory[0x1000-0x1001] = 0xABCD
LDR R1, =0x1000
LDRH R0, [R1]
; R0 = 0x0000ABCD
```

Use Cases

- Loading 16-bit integers (short)
- Unicode characters (UTF-16)
- 16-bit data types

8.4.2 3.2 Store Register Half-word (STRH)

Syntax

```
STRH Rd, [Rn, #offset]   ; Store 16 bits
```

Operation

- Writes lower 16 bits of register to memory
- Upper 16 bits ignored

Example

```
MOV R0, #0x1234
LDR R1, =0x2000
STRH R0, [R1]
; Memory[0x2000-0x2001] = 0x1234
```

8.4.3 3.3 Load Register Signed Half-word (LDRSH)

Syntax

```
LDRSH Rd, [Rn, #offset]  ; Load signed 16-bit
```

Operation

- Loads 16 bits from memory
- Replicates sign bit (bit 15) to upper 16 bits
- Sign-extension

Example

```
; Memory = 0x8000 (-32768 as signed 16-bit)
LDRSH R0, [R1]
; R0 = 0xFFFF8000 (-32768 as signed 32-bit)
```

8.5 4. String Copy Example (strcpy)

8.5.1 4.1 C Implementation

Code

```
void strcpy(char x[], char y[]) {
    int i = 0;
    while ((x[i] = y[i]) != '\\0') {
        i++;
    }
}
```

Algorithm

1. Copy characters from y to x one at a time
2. Stop when null terminator ('\0') encountered
3. Null terminator also copied

8.5.2 4.2 ARM Assembly Implementation

Register Allocation

```
R0: Base address of x (destination)
R1: Base address of y (source)
R4: Loop counter i
R2: Address of y[i]
R3: Value of y[i]
R12: Address of x[i]
```

Complete Assembly

```
strcpy:
    ; Prologue: Save R4 (must preserve)
    SUB SP, SP, #4
    STR R4, [SP, #0]

    ; Initialize counter
    MOV R4, #0          ; i = 0

loop:
    ; Calculate address of y[i]
    ADD R2, R4, R1      ; R2 = y + i

    ; Load y[i]
    LDRB R3, [R2, #0]   ; R3 = y[i]

    ; Calculate address of x[i]
    ADD R12, R4, R0     ; R12 = x + i

    ; Store to x[i]
    STRB R3, [R12, #0]  ; x[i] = y[i]

    ; Check for null terminator
    CMP R3, #0          ; Is y[i] == '\\0'?
    BEQ done            ; If yes, exit loop

    ; Increment counter
    ADD R4, R4, #1      ; i++
    B loop              ; Continue loop

done:
    ; Epilogue: Restore R4
    LDR R4, [SP, #0]
    ADD SP, SP, #4
    MOV PC, LR          ; Return
```

8.5.3 4.3 Key Points

Why LDRB/STRB?

- Strings are char arrays (8-bit elements)
- Must use byte operations

Register Preservation

- R4 must be saved/restored (callee-saved)
- R12 doesn't need preservation (scratch register)

Offsets Are Immediate

- `[R2, #0]` uses immediate offset (hash symbol)
- Cannot use `[R2, R3]` directly without proper syntax

8.6 5. Library Functions: scanf and printf

8.6.1 5.1 scanf Function

Purpose

- Read input from standard input (keyboard)
- Parse formatted input

C Signature

```
int scanf(const char *format, ...);
```

Arguments

- R0: Address of format string (“%d”, “%c”, “%s”, etc.)
- R1: Address where to store input (NOT the value!)
- R2, R3: Additional addresses for more inputs

Example: Read Integer

C Code

```
int x;
scanf("%d", &x);  // Note: &x (address of x)
```

ARM Assembly

```
.data
formatS: .asciz "%d"

.text
    ; Allocate space for variable
    SUB SP, SP, #4         ; Space for x

    ; Load format string address
    LDR R0, =formatS       ; R0 = address of "%d"

    ; Load stack address
    MOV R1, SP             ; R1 = address where to store

    ; Call scanf
    BL scanf

    ; Value now stored at [SP]
    LDR R2, [SP, #0]       ; R2 = x
```

8.6.2 5.2 printf Function

Purpose

- Print output to standard output (screen)
- Format and display data

C Signature

```
int printf(const char *format, ...);
```

Arguments

- R0: Address of format string
- R1, R2, R3: VALUES to print (not addresses!)

Example: Print Integer

C Code

```
printf("Result: %d\\n", result);
```

ARM Assembly

```
.data
formatP: .asciz "Result: %d\\n"

.text
    ; Load value to print
    LDR R1, [SP, #0]      ; R1 = result (value, not address)

    ; Release stack space (before printf)
    ADD SP, SP, #4

    ; Load format string
    LDR R0, =formatP

    ; Call printf
    BL printf
```

8.6.3 5.3 Data Section and Format Strings

Data Section

```
.data
formatS: .asciz "%d"      ; Input format
formatP: .asciz "Result: %d\\n"  ; Output format
array: .word 1, 2, 3, 4   ; Array
message: .asciz "Hello"   ; String
```

.asciz Directive

- Defines null-terminated string
- Automatically adds '\0' at end
- Stored in data section (separate from code)

Pseudo-Operation: LDR Rd, =label

```
LDR R0, =formatS     ; Loads ADDRESS of formatS into R0
```

- Not actual LDR instruction
- Assembler converts to appropriate instruction(s)

- Loads memory address (pointer), not content

8.6.4 5.4 scanf vs printf Argument Differences

scanf: Needs Addresses

```
SUB SP, SP, #4
MOV R1, SP          ; R1 = address (where to store)
BL scanf
```

printf: Needs Values

```
LDR R1, [SP]        ; R1 = value (what to print)
BL printf
```

Why This Difference?

- scanf modifies variables (needs addresses to write to)
- printf only reads values (copies values)

8.6.5 5.5 Calling Convention Rules

Follow Exact Order

- R0 first, R1 second, R2 third, R3 fourth
- Library functions expect specific argument positions
- Assembly won't check violations
- Mistakes cause wrong behavior or crashes

Know Function Signatures

- Read documentation
- Understand parameter types and order
- Match assembly to C function prototype

8.7 6. Compilation, Linking, and Loading

8.7.1 6.1 Translation Overview

Complete Process

```
C Program (.c)
    ↓ [Compiler]
Assembly (.s)
    ↓ [Assembler]
Object Module (.o)
    ↓ [Linker]
Executable (a.out)
    ↓ [Loader]
Memory (running program)
```

8.7.2 6.2 Compiler

Function

- Converts high-level C code to assembly language
- Complex task requiring sophisticated algorithms
- Performs optimizations

Optimizations

- Register allocation
- Instruction selection
- Loop unrolling
- Dead code elimination
- Function inlining

Example

```
int add(int a, int b) {
    return a + b;
}
```

↓ Compiler

```
add:
    ADD R0, R0, R1
    MOV PC, LR
```

8.7.3 6.3 Assembler

Function

- Converts assembly language to machine code (binary)
- Simpler than compilation (mostly 1-to-1 mapping)
- Produces object modules

Tasks

1. Translate instructions to binary opcodes
2. Resolve local labels to addresses
3. Generate symbol table
4. Create relocation information

Object Module Structure

Header

- Describes contents and sizes

Text Segment

- Machine instructions (binary code)

Static Data Segment

- Initialized global variables
- String constants (format strings)

Relocation Info

- Instructions/data depending on absolute addresses
- Needed when program loaded at different address

Symbol Table

- Global definitions: functions, variables defined here
- External references: functions/variables from other modules
- Enables linking

Debug Info

- Maps machine code to source code lines
- Used by debuggers (gdb)

8.7.4 6.4 Linker

Function

- Combines multiple object modules into executable
- Links program code with library code

Tasks

1. Merge Segments

```
program.o:      lib.o:          Result:
[Text1]         [Text2]     →   [Text1+Text2]
[Data1]         [Data2]     →   [Data1+Data2]
```

2. Resolve Labels

- Convert symbolic names to actual addresses
- Example: “printf” → 0x80481234
- Processor only understands addresses

3. Patch References

- Update function calls to correct addresses
- Fix relocatable addresses
- May leave some for loader

8.7.5 6.5 Static vs Dynamic Linking

Static Linking

- Library code copied into executable at compile time
- Larger executable files
- Self-contained (no external dependencies)
- All code in one file

Advantages

- No runtime dependencies
- Faster load time

- Predictable behavior

Disadvantages

- Large file sizes
- No benefit from library updates
- Memory duplication across programs

Dynamic Linking

- Library code loaded at runtime when called
- Smaller executables
- Shared libraries on system

Advantages

- Smaller executables
- Shared libraries (less memory usage)
- Automatic library updates
- Less disk space

Disadvantages

- Requires libraries installed on system
- “DLL not found” errors
- Slightly slower initial load

DLL (Dynamic Link Library) - Windows

- File extension: .dll
- Shared by multiple programs
- Must be present on system
- Example: msvcrt.dll (C runtime library)

8.7.6 6.6 Loader

Function

- Loads executable from disk into memory
- Prepares program for execution
- Initializes execution environment

Loading Steps

1. Read Header

- Determine segment sizes
- Text segment size
- Data segment size
- Other metadata

2. Create Virtual Address Space

- Allocate memory for program
- Set up page tables (virtual memory)
- Map segments to physical memory

3. Copy Segments to Memory

- Text segment (instructions)
- Initialized data
- Set up page table entries
- Mark text as read-only, data as read-write

4. Set Up Arguments on Stack

- Command-line arguments: argc, argv
- Environment variables
- Initial stack frame

Example

```
./program arg1 arg2
```

- argc = 3
- argv[0] = "./program"
- argv[1] = "arg1"
- argv[2] = "arg2"

5. Initialize Registers

- Set up register file
- PC points to entry point (_start)
- SP points to top of stack
- Other registers to initial values

6. Jump to Startup Routine

- Calls C runtime initialization
- Sets up standard library
- Calls main() function
- When main returns, calls exit()

8.8 7. Lab Exercise Topics

8.8.1 7.1 Common String Operations

String Length

```
int strlen(char *s) {
    int len = 0;
    while (s[len] != '\\0')
        len++;
    return len;
}
```

String Reverse

```
void strrev(char *s) {
    int len = strlen(s);
    for (int i = 0; i < len/2; i++) {
        char temp = s[i];
        s[i] = s[len-1-i];
        s[len-1-i] = temp;
    }
}
```

8.8.2 7.2 Integer I/O

Read Two Integers, Print Sum

```
; Read x and y
; Print x + y
```

Read n, Print 1 to n

```
; Read n
; Loop from 1 to n, print each
```

8.8.3 7.3 Skills Required

- Character data handling (LDRB/STRB)
- String manipulation
- scanf for input
- printf for output
- Stack management
- Function calling conventions
- Loop implementation
- Array indexing

8.9 Key Takeaways

1. **ASCII (7-bit), Latin-1 (8-bit), Unicode (32-bit)** represent character data with increasing capacity.

2. **LDRB/STRB for byte operations**, LDRH/STRH for half-words - smaller than word operations.

3. **Byte operations don't require alignment** unlike word operations (LDR/STR).

4. **Sign extension (LDRSB/LDRSH)** replicates sign bit to preserve signed values.

5. **Strings in C are char arrays** terminated with null character ('\0' = 0).

6. **scanf and printf are library functions** called via BL instruction.

7. **scanf needs addresses (where to store)**, printf needs values (what to print).

8. **Format strings stored in .data section** using .asciz directive.

9. **Arguments passed in R0-R3** following ARM calling convention.

10. **Compilation chain: Compile → Assemble → Link → Load → Execute**.

11. **Static linking includes libraries in executable**, dynamic linking loads at runtime.
12. **Loader sets up virtual memory, copies segments, initializes stack** with arguments.

8.10 Summary

Character data handling and library function usage bridge high-level programming concepts and assembly implementation. Understanding byte/half-word operations enables efficient string manipulation and compact data storage. The scanf/printf functions demonstrate how assembly code interfaces with system libraries, requiring careful attention to calling conventions and argument types. The compilation, linking, and loading process reveals how source code transforms into running programs, involving multiple stages with distinct responsibilities. Static and dynamic linking represent different trade-offs between self-containment and flexibility. These concepts are essential for systems programming, understanding program structure, and debugging low-level issues. This knowledge prepares us for advanced topics including operating systems, compilers, and system-level optimization.

Chapter 9

Lecture 9: Microarchitecture and Datapath

9.1 Introduction

This lecture transitions from instruction set architecture (ISA) to microarchitecture—the hardware implementation of the ISA. We explore how to build a processor that executes MIPS instructions, covering instruction formats, digital logic fundamentals, datapath construction, and single-cycle processor design. Understanding microarchitecture reveals how software instructions translate to hardware operations and provides the foundation for studying advanced processor designs including pipelining and superscalar execution.

9.2 1. Course Context and MIPS ISA

9.2.1 1.1 Transition to Hardware Implementation

Previous Focus: ARM ISA

- Instruction set
- Assembly programming
- Software perspective

Current Focus: MIPS Microarchitecture

- Hardware implementation
- Processor design
- Hardware perspective

Why MIPS for Hardware Study?

- Simpler than ARM (educational clarity)
- Clean RISC design
- Well-documented architecture

- Concepts apply to all processors

9.2.2 1.2 MIPS Instruction Categories

Three Instruction Types (based on encoding)

I-Type (Immediate)

- Contains one immediate operand
- Covers data processing, data transfer, control flow
- Examples: ADDI, LW, SW, BEQ
- Most common type

R-Type (Register)

- All operands are registers
- Primarily arithmetic and logic
- Examples: ADD, SUB, AND, OR
- Opcode always 0, funct field specifies operation

J-Type (Jump)

- Jump instructions
- Examples: J, JAL
- 26-bit address field

Contrast with ARM

- ARM: Data processing, data transfer, flow control
- MIPS: I-type, R-type, J-type
- Different classification philosophy

9.2.3 1.3 MIPS Instruction Encoding

Fixed 32-Bit Length

- Every instruction exactly 32 bits
- Simplifies fetch and decode
- Enables efficient pipelining

R-Type Format

```
[Opcode][RS][RT][RD][SHAMT][Funct]
 6 bits  5   5   5    5      6 bits
```

Fields:

- **Opcode**: Always 0 for R-type
- **RS**: Source register 1 (5 bits for 32 registers)
- **RT**: Source register 2
- **RD**: Destination register
- **SHAMT**: Shift amount (for shift instructions)
- **Funct**: Function code (actual operation)

I-Type Format

```
[Opcode][RS][RT][Immediate]
 6 bits  5   5   16 bits
```

Fields:

- **Opcode**: Varies by instruction
- **RS**: Source/base register
- **RT**: Source/destination register
- **Immediate**: 16-bit immediate value or offset

J-Type Format

```
[Opcode][Address]
 6 bits  26 bits
```

Fields:

- **Opcode**: 2 for J, 3 for JAL
- **Address**: 26-bit jump target (word address)

9.3 2. Digital Logic Review

9.3.1 2.1 Information Encoding

Binary Representation

- Low voltage = Logic 0
- High voltage = Logic 1
- Digital signals immune to analog noise

Multi-Bit Signals

- One wire per bit
- 32-bit instruction needs 32 wires
- Parallel transmission within CPU

9.3.2 2.2 Combinational Elements

Definition

- Output is function of inputs ONLY
- No internal state or memory
- Purely functional relationship

Examples

- AND, OR, NOT gates
- Multiplexers: `Y = (S == 0) ? I0 : I1`
- Adders: `Y = A + B`
- ALU: `Y = function(A, B, operation)`

Characteristics

- Output changes immediately with input (plus propagation delay)

- Can draw complete truth table
- Asynchronous operation (no clock needed)

9.3.3 2.3 Sequential Elements (State Elements)

Definition

- Output is function of inputs AND internal state
- Has memory—stores information over time
- State persists between clock cycles

Examples

- Registers
- Flip-flops
- Register files
- Memory units

Characteristics

- Store information
- Synchronized to clock signal
- Output depends on history

9.3.4 2.4 Clocking and Timing

Clock Signal

- Periodic alternating signal: Low → High → Low → High…
- Synchronizes all sequential operations

Edge-Triggered

- Rising edge: Transition 0 → 1
- Falling edge: Transition 1 → 0
- Most processors use rising edge

Clock Period and Frequency

```
Clock Period (T): Duration of one cycle
Clock Rate (f): Cycles per second

Relationship: f = 1/T

Example:
T = 250 ps = 0.25 ns
f = 1/(250 × 10^-12) = 4 GHz
```

9.3.5 2.5 Register Operations

Basic Register

- Stores multi-bit value (e.g., 32 bits)

- Updates on clock edge: D (input) → Q (output state)

Register with Write Control

- Additional Write Enable signal
- Updates ONLY when clock edge AND Write Enable = 1
- Otherwise holds previous value

Timing Example

```
Clock: __| |__| |__| |__
Write:    |___|
Data:   [A][B][C][D][E]
State:  [A][A][A][D][D]
```

9.3.6 2.6 Critical Path and Clock Period

Combinational Logic Delay

- All combinational elements have propagation delay
- Different elements, different delays

Clock Period Constraint

```
Clock Period  Longest Path Delay

Path: Register → Combinational Logic → Register

Must allow time for:
1. Register output stabilization
2. Combinational logic computation
3. Result reaching next register input
4. Setup time before next clock edge
```

Critical Path

- Longest delay path from register to register
- Determines minimum clock period
- Limits maximum clock frequency

Single-Cycle Constraint

- Complete one instruction per clock cycle
- Clock period must accommodate slowest instruction
- All instructions take same time (inefficient!)

9.4 3. CPU Execution Stages

9.4.1 3.1 Instruction Fetch (IF)

Purpose: Retrieve next instruction from memory

Steps:

1. Use Program Counter (PC) for instruction address
2. Access Instruction Memory with PC
3. Retrieve 32-bit instruction word
4. Instruction now in CPU for processing

Hardware:

- Program Counter (32-bit register)
- Instruction Memory (read-only during execution)
- Address bus from PC to memory
- Data bus from memory to CPU

9.4.2 3.2 Instruction Decode (ID)

Purpose: Interpret instruction and extract fields

Decode Operations:

1. **Examine Opcode** (bits 26-31):
    - If opcode = 0: R-type
    - If opcode = 2 or 3: J-type
    - Otherwise: I-type
2. **Extract Register Numbers**:
    - R-type: RS, RT, RD (three 5-bit fields)
    - I-type: RS, RT (two 5-bit fields)
    - J-type: No registers
3. **Extract Immediate/Address**:
    - I-type: 16-bit immediate
    - J-type: 26-bit address
4. **Extract Function/Shift** (R-type only):
    - Funct: bits 0-5 (ALU operation)
    - SHAMT: bits 6-10 (shift amount)

Control Unit Role:

- Decodes opcode
- Generates control signals
- Determines datapath activation

9.4.3 3.3 Execute (EX)

Purpose: Perform operation or calculate address

Operations by Type:

Arithmetic/Logic (R-type, I-type arithmetic):

- Send operands to ALU
- ALU performs operation

- Operation from funct field (R-type) or opcode (I-type)

Memory Access (Load/Store):

- ALU calculates address: Base + Offset
- Always performs addition
- Result is memory address

Branch:

- ALU compares registers: RS - RT
- Zero flag indicates equality
- Result determines branch decision

9.4.4 3.4 Memory Access (MEM)

Purpose: Read or write data memory

Applies To:

- Load instructions: Read from memory
- Store instructions: Write to memory
- NOT arithmetic/logic (skip this stage)

Load Operation:

1. Use address from ALU
2. Read data from memory
3. Data will be written to register

Store Operation:

1. Use address from ALU
2. Get data from RT register
3. Write data to memory

9.4.5 3.5 Register Write-Back (WB)

Purpose: Write result to destination register

Applies To:

- Arithmetic/Logic: Write ALU result
- Load: Write memory data
- NOT store or branch

Source Selection:

- Arithmetic/Logic: Data from ALU
- Load: Data from memory
- Multiplexer selects appropriate source

9.4.6 3.6 PC Update

Purpose: Determine next instruction address

Default: PC = PC + 4 (sequential)

Branch/Jump: PC = calculated target address

Control Flow:

- Multiplexer selects next PC value
- Sequential or branch/jump target
- Update happens at clock edge

9.5 4. R-Type Instruction Datapath

9.5.1 4.1 Register File

Structure:

- 32 registers (R0-R31), 32 bits each
- Three ports: 2 read, 1 write

Read Ports:

- Read Address 1: RS (5 bits)
- Read Address 2: RT (5 bits)
- Read Data 1: 32-bit output
- Read Data 2: 32-bit output
- Combinational (no clock)

Write Port:

- Write Address: RD (5 bits)
- Write Data: 32-bit input
- Write Enable: Control signal
- Synchronized (clock edge)

9.5.2 4.2 R-Type Execution Flow

Instruction: `ADD $t0, $t1, $t2` (R0 = R1 + R2)

Step 1: Register Read

- Extract RS (R1) and RT (R2) fields
- Register file outputs two 32-bit values

Step 2: ALU Operation

- Inputs: Two register values
- Funct field (6 bits) → ALU control (4 bits)
- ALU performs specified operation
- Examples: ADD, SUB, AND, OR, SLT

Step 3: Write-Back

- ALU result → Register file write data
- RD field specifies destination
- Write Enable = 1
- At clock edge: Result written

9.5.3 4.3 ALU Control

Function Field Encoding:

Funct	Operation	ALU Control
0x20	ADD	0010
0x22	SUB	0110
0x24	AND	0000
0x25	OR	0001
0x2A	SLT	0111

ALU Control Logic:

- Input: 6-bit funct field
- Output: 4-bit ALU operation
- Combinational logic (lookup table)

9.6 5. I-Type Instruction Datapath

9.6.1 5.1 Differences from R-Type

Operand Sources:

- R-type: Both from registers
- I-type: One register, one immediate

Register Usage:

- RS: Source register
- RT: Destination register (NOT source!)
- Immediate: 16-bit operand

9.6.2 5.2 Sign Extension

Problem: 16-bit immediate, 32-bit ALU

Process:

1. Take 16-bit immediate
2. Examine bit 15 (sign bit)
3. Replicate sign bit to bits 16-31
4. Result: 32-bit signed value

Examples:

```
16-bit: 0x0005 → 32-bit: 0x00000005 (+5)
16-bit: 0xFFFB → 32-bit: 0xFFFFFFFB (-5)
```

Hardware: Simple wire replication (fast)

9.6.3 5.3 Multiplexer for ALU Input

ALU Input B Selection:

- Input 0: Register data (RT) for R-type
- Input 1: Sign-extended immediate for I-type
- Select: ALUSrc control signal

ALUSrc Signal:

```
ALUSrc = 0: Use register (R-type, branch)
ALUSrc = 1: Use immediate (I-type)
```

9.7 6. Load/Store Instruction Datapath

9.7.1 6.1 Address Calculation

Formula: Address = Base + Offset

Components:

- Base: RS register (32-bit pointer)
- Offset: 16-bit signed immediate (sign-extended)
- ALU: Always performs addition

Examples:

```
LW $t1, 8($t0)    # Load from $t0 + 8
SW $t2, -4($sp)   # Store to $sp - 4
```

9.7.2 6.2 Load Word (LW)

Instruction Format:

- RS: Base register
- RT: Destination register
- Immediate: Offset

Execution:

1. Read RS (base address)
2. Sign-extend immediate (offset)
3. ALU adds: Address = RS + offset
4. Read data from memory at address
5. Write data to RT register

Critical Path: Longest in single-cycle design

- Fetch → Register Read → ALU → Memory → Register Write

9.7.3 6.3 Store Word (SW)

Instruction Format:

- RS: Base register
- RT: Source register (data to store)
- Immediate: Offset

Execution:

1. Read RS (base) and RT (data)
2. ALU calculates address
3. Write RT data to memory at address
4. NO register write-back

Key Difference:

- Reads TWO registers (RS and RT)
- Memory write instead of read
- No register write stage

9.7.4 6.4 Data Memory

Interface:

- Address: From ALU (32 bits)
- Write Data: From RT register
- Read Data: To register file (for loads)

Control Signals:

- MemRead: Enable read (LW)
- MemWrite: Enable write (SW)

Multiplexer for Write-Back:

- Input 0: ALU result (arithmetic/logic)
- Input 1: Memory data (load)
- Select: MemtoReg signal

9.8 7. Branch Instruction Datapath

9.8.1 7.1 Branch Types

BEQ (Branch if Equal):

- Compare RS and RT
- Branch if RS == RT

BNE (Branch if Not Equal):

- Compare RS and RT
- Branch if RS != RT

9.8.2 7.2 Branch Target Calculation

Components:

1. PC + 4 (next sequential instruction)
2. Offset from immediate (in instructions)
3. Target = (PC + 4) + (Offset × 4)

Why PC + 4?

- Offset relative to NEXT instruction
- PC already incremented

Word to Byte Conversion:

- Immediate: Number of instructions
- Multiply by 4: Byte offset
- Shift left 2 (wire routing, no hardware!)

9.8.3 7.3 Branch Execution

Step 1: Register Comparison

- Read RS and RT
- ALU subtracts: RS - RT
- Generate Zero flag

Step 2: Zero Flag Evaluation

- Zero = 1: Values equal
- Zero = 0: Values different

Step 3: Target Calculation (parallel)

- Sign-extend immediate
- Shift left 2
- Add to PC + 4

Step 4: PC Update Decision

```
BEQ: PCSrc = Branch AND Zero
BNE: PCSrc = Branch AND NOT(Zero)
```

Multiplexer:

- Input 0: PC + 4 (sequential)
- Input 1: Branch target
- Select: PCSrc

9.8.4 7.4 Sign Extension and Shifting

Sign Extension: Preserves signed offset

- Forward branch: Positive offset
- Backward branch: Negative offset

Shift Left 2: Wire routing trick

- Take bits 0-29 of sign-extended value
- Connect to bits 2-31 of result
- Append two zero wires at bits 0-1
- NO actual shifter hardware!

9.9 8. Complete Single-Cycle Datapath

9.9.1 8.1 Integrated Components

Instruction Fetch:

- PC register
- Instruction memory
- PC + 4 adder

Register File:

- 32 registers with 3 ports
- Two read, one write

ALU:

- Two 32-bit inputs
- Operation control
- Result output
- Zero flag

Data Memory:

- Address from ALU
- Write data from register
- Read data to register

Sign Extender:

- 16-bit input
- 32-bit output

Branch Logic:

- Target adder
- PC multiplexer

Multiplexers:

- ALU input B (register vs immediate)
- Register write data (ALU vs memory)

- Next PC (PC+4 vs branch target)

9.9.2 8.2 Control Signals

Generated by Control Unit:

1. RegDst: Register destination select
2. Branch: Branch instruction indicator
3. MemRead: Memory read enable
4. MemtoReg: Memory to register select
5. MemWrite: Memory write enable
6. ALUSrc: ALU source select
7. RegWrite: Register write enable
8. ALUOp: ALU operation type

9.9.3 8.3 Parallel Operations

Key Insight: Hardware operates in PARALLEL

- All datapath elements active simultaneously
- Some produce meaningless results
- Control signals select valid paths

Example: R-type instruction

- Sign extender operates on bits 0-15
- Produces meaningless output (no immediate in R-type)
- Multiplexer doesn't select it (ALUSrc = 0)

9.9.4 8.4 Critical Path Analysis

Path for Load Word (longest):

```
1. Instruction fetch:     200 ps
2. Register read:         150 ps
3. Sign extend:           50 ps
4. Multiplexer:           25 ps
5. ALU address calc:      200 ps
6. Data memory access:    200 ps
7. Multiplexer:           25 ps
8. Register write setup:  100 ps
Total:                    950 ps
```

Clock Period: Must be ≥ 950 ps **Max Frequency**: 1/950 ps ≈ 1.05 GHz

Inefficiency:

- ALL instructions take 950 ps
- Fast R-type (650 ps) waits
- Wasted time per fast instruction

9.9.5 8.5 Single-Cycle Disadvantages

Inefficiency:

- Fast instructions wait for slow ones
- Clock period by worst case
- Cannot optimize common case

Hardware Duplication:

- Separate instruction/data memories
- Multiple adders
- Cannot reuse hardware in same cycle

No Parallelism:

- One instruction at a time
- Hardware mostly idle
- Poor resource utilization

Advantages:

- Simple design
- Simple control
- One instruction per cycle (conceptually)
- Good for learning

9.10 Key Takeaways

1. **Microarchitecture is hardware implementation of ISA** - translating instruction semantics to hardware operations.

2. **MIPS uses three instruction types**: R-type (registers), I-type (immediate), J-type (jump).

3. **Fixed 32-bit instructions** simplify fetch/decode and enable efficient pipelining.

4. **Combinational elements** have output as function of inputs only; sequential elements have state.

5. **Clock period must exceed longest combinational path** between sequential elements.

6. **Six execution stages**: Fetch, Decode, Execute, Memory, Write-back, PC Update.

7. **Register file has three ports**: two read (combinational), one write (clocked).

8. **Sign extension** converts 16-bit immediate to 32-bit preserving signed value.

9. **Multiplexers select between data sources** based on control signals.

10. **ALU operations vary by instruction**: addition (load/store), subtraction (branch), varies (R-type).

11. **Critical path determines clock period** - load word is longest in single-cycle design.
12. **Single-cycle processor completes one instruction per cycle** but inefficiently (all take same time).
13. **Separate instruction and data memories** required for single-cycle (both accessed same cycle).
14. **Control signals orchestrate datapath** - generated by control unit from opcode.
15. **All hardware operates in parallel** - control signals select valid results, ignore others.

9.11 Summary

Microarchitecture bridges the gap between software instructions and hardware implementation, revealing how processors execute programs. Building a single-cycle MIPS processor requires understanding digital logic fundamentals, datapath component design, and control signal generation. While conceptually simple (one instruction per cycle), the single-cycle design is inefficient because all instructions must complete within the time required by the slowest instruction. The critical path—typically the load word instruction—determines the maximum clock frequency. Understanding this foundation prepares us for more sophisticated designs including multi-cycle processors (which break execution into multiple stages) and pipelined processors (which overlap instruction execution for higher throughput). These microarchitecture concepts apply broadly across processor design, from embedded systems to high-performance superscalar processors.

Chapter 10

Lecture 10: Processor Control

10.1 Introduction

This lecture completes the single-cycle MIPS processor design by exploring the control unit—the component that generates control signals based on instruction opcodes. We examine ALU control generation using a two-stage approach, design the main control unit, analyze control signal purposes, and create truth tables mapping instructions to control patterns. Understanding control unit design reveals how hardware interprets instructions and orchestrates datapath operations, completing our understanding of processor implementation.

10.2 1. Control Unit Overview

10.2.1 1.1 Recap of Datapath Components

Previously Covered:

- Register File (32 registers, 3 ports)
- ALU (arithmetic/logic operations)
- Instruction Memory (stores program)
- Data Memory (stores data)
- Adders (PC+4, branch target)
- Multiplexers (data source selection)
- Sign Extender (16-bit to 32-bit)
- Shifter (branch offset left 2)

10.2.2 1.2 Control Unit Purpose

Function: Generate control signals based on instruction

Inputs:

- Opcode (bits 26-31, 6 bits)

- Funct field (bits 0-5, 6 bits) for R-type

Outputs: Control signals for datapath

- Multiplexer selections
- Register write enable
- Memory read/write
- ALU operation
- Branch decision

10.2.3 1.3 Instruction Subset for Study

Selected Instructions:

- **Load Word (LW)**: Memory read
- **Store Word (SW)**: Memory write
- **Branch if Equal (BEQ)**: Conditional branch
- **R-type**: Arithmetic, logic, shift

Coverage:

- Uses almost all datapath hardware
- Representative of most control signals
- Excludes: Jump instructions, I-type arithmetic

10.3 2. ALU Operations for Different Instructions

10.3.1 2.1 Load/Store Instructions

Address Calculation:

```
Address = Base Register + Immediate Offset
        = RS + Sign_Extend(Immediate)
```

ALU Function: ADDITION (always)

- Input A: RS register value
- Input B: Sign-extended immediate
- Operation: ADD
- ALU Control: 0010 (binary)
- Result: Memory address

Example:

```
LW $t1, 8($t0)    # Address = $t0 + 8
SW $t2, -4($sp)   # Address = $sp + (-4)
```

10.3.2 2.2 Branch Instructions

Comparison Operation:

```
Compare RS and RT for equality
Method: Subtract RT from RS
```

ALU Function: SUBTRACTION

- Input A: RS register value
- Input B: RT register value
- Operation: SUB
- ALU Control: 0110 (binary)
- Result: RS - RT
- Zero Flag: Indicates if result is zero (equal)

Branch Decision:

```
Zero = 1: RS == RT, take branch
Zero = 0: RS != RT, don't take branch
```

10.3.3 2.3 R-Type Instructions

Variable Operations: Determined by funct field

ALU Function: DEPENDS ON FUNCT

- Input A: RS register value
- Input B: RT register value
- Operation: From funct field
- ALU Control: Varies
- Result: Written to RD register

Funct Field Mapping:

```
Funct    | Operation | ALU Control
---------|-----------|-------------
0x20     | ADD       | 0010
0x22     | SUB       | 0110
0x24     | AND       | 0000
0x25     | OR        | 0001
0x2A     | SLT       | 0111
```

10.4 3. ALU Control Signal

10.4.1 3.1 Signal Format

4-Bit Signal: Specifies ALU operation

Possible Operations ($2^4 = 16$):

```
0000: AND
0001: OR
0010: ADD
0110: SUBTRACT
0111: Set on Less Than (SLT)
```

`1100: NOR`

Usage:

- Not all 16 combinations used
- Could use 3 bits for 8 operations
- 4-bit standard allows expansion

10.4.2 3.2 Control Signal Usage by Instruction

Load/Store:

- ALU Control = 0010 (ADD)
- Fixed operation
- Independent of instruction specifics

Branch:

- ALU Control = 0110 (SUBTRACT)
- Fixed operation
- Zero flag is critical output

R-Type:

- ALU Control = Varies
- Must decode funct field
- Different operations need different controls

10.5 4. Two-Stage ALU Control Generation

10.5.1 4.1 Design Rationale

Why Two Stages?

Efficiency:

- Some instructions don't need funct field
- Separates opcode-level from operation-level
- Faster for non-R-type instructions

Timing Optimization:

- Other control signals needed faster
- Examples: Register addressing, immediate routing
- ALU control can afford slight delay

Modularity:

- Stage 1: Main control (opcode-based)
- Stage 2: ALU control (operation-specific)
- Cleaner design separation

10.5.2 4.2 Stage 1: Generate ALUOp

Input: Opcode (6 bits)

Output: ALUOp (2 bits)

Encoding:

Instruction	Opcode	ALUOp
Load Word	100011	00
Store Word	101011	00
Branch Equal	000100	01
R-type	000000	10

ALUOp Meaning:

- **00**: Perform ADD (address calculation)
- **01**: Perform SUBTRACT (comparison)
- **10**: Operation from funct field

Logic: Purely combinational based on opcode

10.5.3 4.3 Stage 2: Generate ALU Control

Inputs:

- ALUOp (2 bits from Stage 1)
- Funct field (6 bits from instruction)
- Total: 8 input bits

Output: ALU Control (4 bits)

Truth Table:

ALUOp	Funct	ALU Control	Operation
00	XXXXXX	0010	ADD (LW/SW)
01	XXXXXX	0110	SUB (BEQ)
10	100000	0010	ADD (R-type)
10	100010	0110	SUB (R-type)
10	100100	0000	AND
10	100101	0001	OR
10	101010	0111	SLT

"X" Notation: Don't Care

- For ALUOp = 00 or 01, funct irrelevant
- Simplifies logic design
- Reduces gate count

10.5.4 4.4 Complete ALU Control Path

Flow Diagram:

```
Instruction Opcode (6 bits)
        ↓
   [Main Control Unit]
        ↓
    ALUOp (2 bits)  +  Funct Field (6 bits)
        ↓                    ↓
             [ALU Control Unit]
                    ↓
           ALU Control (4 bits)
                    ↓
                  [ALU]
```

Advantages:

- Modular design
- Simplified main control
- Localized R-type complexity
- Easier to verify

10.6 5. Main Control Signals

10.6.1 5.1 Complete Signal List

Signals Generated:

1. **RegDst** (1 bit): Register destination select
2. **Branch** (1 bit): Branch instruction indicator
3. **MemRead** (1 bit): Memory read enable
4. **MemtoReg** (1 bit): Memory to register select
5. **MemWrite** (1 bit): Memory write enable
6. **ALUSrc** (1 bit): ALU source select
7. **RegWrite** (1 bit): Register write enable
8. **ALUOp** (2 bits): To ALU control unit

Total: 9 control bits from main control

10.6.2 5.2 RegDst (Register Destination)

Purpose: Select which field specifies write destination

Multiplexer Control:

- Input 0: RT field (bits 16-20)
- Input 1: RD field (bits 11-15)
- Output: Register write address (5 bits)

Settings:

```
RegDst = 0: Write to RT (Load Word)
RegDst = 1: Write to RD (R-type)
```

Rationale:

- Load Word: RT is destination (I-type format)
- R-type: RD is destination (R-type format)
- Store/Branch: Don't care (no write)

Examples:

```
LW $t1, 8($t0)     # Write to $t1 (RT) → RegDst = 0
ADD $t2, $t3, $t4  # Write to $t2 (RD) → RegDst = 1
```

10.6.3 5.3 Branch

Purpose: Indicate if instruction is branch

Usage: Combined with Zero flag for PC selection

Settings:

```
Branch = 0: Not a branch (LW, SW, R-type)
Branch = 1: Branch instruction (BEQ, BNE)
```

PC Selection Logic:

```
For BEQ:
  PCSrc = Branch AND Zero
  (Take branch if instruction is branch AND comparison equal)

For BNE:
  PCSrc = Branch AND NOT(Zero)
  (Take branch if instruction is branch AND comparison not equal)
```

10.6.4 5.4 MemRead

Purpose: Enable reading from data memory

Settings:

```
MemRead = 0: No memory read (R-type, SW, BEQ)
MemRead = 1: Read from memory (LW)
```

Function:

- Controls data memory read enable
- When high: Memory outputs data
- When low: Memory read inactive

10.6.5 5.5 MemtoReg (Memory to Register)

Purpose: Select source of register write data

Multiplexer Control:

- Input 0: ALU result
- Input 1: Data memory read data
- Output: Register write data (32 bits)

Settings:

```
MemtoReg = 0: Write ALU result (R-type)
MemtoReg = 1: Write memory data (LW)
```

Examples:

```
ADD $t1, $t2, $t3  # $t1 = ALU result → MemtoReg = 0
LW $t1, 8($t0)     # $t1 = memory data → MemtoReg = 1
```

10.6.6 5.6 MemWrite

Purpose: Enable writing to data memory

Settings:

```
MemWrite = 0: No memory write (R-type, LW, BEQ)
MemWrite = 1: Write to memory (SW)
```

Function:

- Controls data memory write enable
- When high: Data written (on clock edge)
- When low: Memory write disabled

10.6.7 5.7 ALUSrc (ALU Source)

Purpose: Select second ALU operand source

Multiplexer Control:

- Input 0: Register file Read Data 2 (RT value)
- Input 1: Sign-extended immediate
- Output: ALU Input B (32 bits)

Settings:

```
ALUSrc = 0: Use register (R-type, BEQ)
ALUSrc = 1: Use immediate (LW, SW)
```

Examples:

```
ADD $t1, $t2, $t3  # Use $t3 → ALUSrc = 0
LW $t1, 8($t0)     # Use imm 8 → ALUSrc = 1
```

10.6.8 5.8 RegWrite

Purpose: Enable writing to register file

Settings:

```
RegWrite = 0: No register write (SW, BEQ)
RegWrite = 1: Write to register (R-type, LW)
```

Usage by Instruction:

```
R-type:    RegWrite = 1 (write ALU result)
Load Word: RegWrite = 1 (write memory data)
Store Word: RegWrite = 0 (no write)
Branch:    RegWrite = 0 (no write)
```

10.7 6. Control Signal Truth Table

10.7.1 6.1 Complete Table

```
Instruction | RegDst | ALUSrc | MemtoReg | RegWrite | MemRead | MemWrite | Branch | A
------------|--------|--------|----------|----------|---------|----------|--------|--
R-type      |   1    |   0    |    0     |    1     |    0    |    0     |   0    |
Load Word   |   0    |   1    |    1     |    1     |    1    |    0     |   0    |
Store Word  |   X    |   1    |    X     |    0     |    0    |    1     |   0    |
Branch Eq   |   X    |   0    |    X     |    0     |    0    |    0     |   1    |
```

Legend:

- **0**: Signal low/false/select input 0
- **1**: Signal high/true/select input 1
- **X**: Don't Care (not used, can be anything)

10.7.2 6.2 R-Type Control

Settings:

```
RegDst = 1:     Write to RD field
ALUSrc = 0:     Second operand from register (RT)
MemtoReg = 0:   Write ALU result
RegWrite = 1:   Enable register write
MemRead = 0:    No memory read
MemWrite = 0:   No memory write
Branch = 0:     Not a branch
ALUOp = 10:     Consult funct field
```

Active Elements:

- Instruction fetch
- Register file (read RS, RT; write RD)
- ALU (operation from funct)
- Register write from ALU
- PC updated to PC + 4

Inactive Elements:

- Data memory (not accessed)
- Branch target (computed but not used)
- Sign extender (operates but ignored)

10.7.3 6.3 Load Word Control

Settings:

```
RegDst = 0:     Write to RT field
ALUSrc = 1:     Second operand from immediate
MemtoReg = 1:   Write memory data
RegWrite = 1:   Enable register write
MemRead = 1:    Enable memory read
MemWrite = 0:   No memory write
Branch = 0:     Not a branch
ALUOp = 00:     ALU performs ADD
```

Active Elements:

- Instruction fetch
- Register file (read RS; write RT)
- Sign extender
- ALU (ADD for address)
- Data memory (read)
- Register write from memory
- PC updated to PC + 4

Critical Path: Longest delay

- Fetch → Reg Read → Sign Extend → ALU → Memory → Reg Write

10.7.4 6.4 Store Word Control

Settings:

```
RegDst = X:     Don't care (no register write)
ALUSrc = 1:     Second operand from immediate
MemtoReg = X:   Don't care (no register write)
RegWrite = 0:   No register write
MemRead = 0:    No memory read
MemWrite = 1:   Enable memory write
Branch = 0:     Not a branch
ALUOp = 00:     ALU performs ADD
```

Key Difference from Load:

- Read TWO registers (RS for base, RT for data)
- Memory write instead of read
- No register write stage

10.7.5 6.5 Branch if Equal Control

Settings:

```
RegDst = X:     Don't care (no register write)
ALUSrc = 0:     Second operand from register (RT)
MemtoReg = X:   Don't care (no register write)
```

```
RegWrite = 0:   No register write
MemRead = 0:    No memory read
MemWrite = 0:   No memory write
Branch = 1:     This is a branch
ALUOp = 01:     ALU performs SUBTRACT
```

Active Elements:

- Instruction fetch
- Register file (read RS, RT)
- ALU (SUBTRACT for comparison, Zero flag)
- Sign extender + shift (branch target)
- Branch target adder (PC + 4 + offset)
- PC multiplexer (select based on Branch AND Zero)

Branch Decision Logic:

```
Zero = (RS - RT == 0)
PCSrc = Branch AND Zero
If PCSrc:
  Next PC = PC + 4 + (SignExtend(Imm) << 2)
Else:
  Next PC = PC + 4
```

10.8 7. Control Unit Implementation

10.8.1 7.1 Input to Control Unit

Primary Input: Opcode (bits 26-31, 6 bits)

- Identifies instruction type
- Determines all control signal values

Secondary Input: Funct field (bits 0-5, 6 bits)

- Only for R-type (opcode = 000000)
- Specifies ALU operation

10.8.2 7.2 Combinational Logic Design

Method: Standard digital logic techniques

Steps:

1. Create truth table (opcode → control signals)
2. List all control signals as outputs
3. Fill in values for each instruction
4. Use Karnaugh maps or Boolean algebra to minimize
5. Implement with logic gates

Example for RegWrite:

```
RegWrite = (R-type) OR (Load Word)
RegWrite = (opcode == 000000) OR (opcode == 100011)
```

10.8.3 7.3 Control Unit Structure

ROM-Based Implementation:

- Opcode as ROM address
- ROM location stores control pattern
- Simple but inflexible

PLA (Programmable Logic Array):

- Implements minimized logic equations
- More efficient than ROM
- Standard for simple processors

Hardwired Logic:

- Custom logic gates
- Fastest implementation
- Most common for high-performance

Microcode (not typical for RISC):

- Control signals stored in memory
- More flexible but slower
- Used in CISC (e.g., x86)

10.8.4 7.4 Timing Considerations

Signal Generation Time:

- Must complete early in clock cycle
- Before datapath elements need signals
- Critical for clock frequency

Signal Stability:

- Must remain stable throughout cycle
- Changes only between instructions
- Combinational logic ensures this

Clock Period Impact:

- Control logic adds delay
- Typically small vs. ALU/memory
- Well-designed control has minimal impact

10.9 8. Why Separate MemRead and MemWrite?

10.9.1 8.1 Initial Observation

Question: Seem mutually exclusive—why not one signal?

- Could use: 0 = Read, 1 = Write
- Appears redundant

10.9.2 8.2 Answer: Yes, Separate Signals Needed

Timing Control:

- Write Enable: Specifies WHEN to write
- Read Enable: Specifies WHEN valid data available
- Different timing requirements

No Operation State:

- Both = 0: No memory access
- Common for R-type and branch
- Single signal couldn't represent this

Three States Required:

```
MemRead=1, MemWrite=0: Read
MemRead=0, MemWrite=1: Write
MemRead=0, MemWrite=0: No access
(MemRead=1, MemWrite=1: Invalid)
```

10.9.3 8.3 Future: Pipelined Processors

Concurrent Access:

- Different pipeline stages access memory
- One stage reading, another writing
- Separate signals essential

Memory Banking:

- Separate read/write ports
- Enables simultaneous access
- Separate signals control independent ports

10.9.4 8.4 Design Philosophy

Orthogonality:

- Each signal controls independent function
- Easier to understand and verify
- Reduces design errors

Flexibility:

- Supports future enhancements
- Allows memory optimization
- Standard practice

10.10 9. Complete Datapath with Control

10.10.1 9.1 Integrated System

Components Connected:

- Control Unit (generates signals)
- Datapath (executes operations)
- Blue lines: Control signals
- Black lines: Data paths

Control Unit Connections:

- Input: Instruction opcode
- Outputs: All control signals
- Fan out to datapath elements

ALU Control Unit:

- Separate box near ALU
- Inputs: ALUOp, Funct
- Output: ALU Control (4 bits)

10.10.2 9.2 Example: Load Word Execution

Instruction: `LW $t1, 8($t0)`

Step 1: Fetch

```
PC → Instruction Memory
Opcode = 100011 (LW)
```

Step 2: Control Signals

```
RegDst=0, ALUSrc=1, MemtoReg=1, RegWrite=1,
MemRead=1, MemWrite=0, Branch=0, ALUOp=00
```

Step 3: Register Read

```
RS field ($t0) → Register file
Read Data 1 = $t0 value
```

Step 4: ALU

```
Immediate = 8
Sign-extended to 32 bits
ALUSrc=1: Selects immediate
ALU performs ADD: $t0 + 8 = address
```

Step 5: Memory

```
MemRead=1: Memory reads at address
Data output from memory
```

Step 6: Write-Back

```
MemtoReg=1: Selects memory data
RegDst=0: Selects RT ($t1)
RegWrite=1: Enables write
At clock edge: Memory data → $t1
```

Step 7: PC Update

```
Branch=0: PCSrc=0
PC updated to PC + 4
```

10.11 Key Takeaways

1. **Control unit generates signals based on instruction opcode**, orchestrating datapath operations.
2. **ALU control uses two-stage generation**: Opcode → ALUOp (2 bits) → ALU Control (4 bits).
3. **Stage 1 (Main Control)**: Opcode to ALUOp - identifies operation category.
4. **Stage 2 (ALU Control)**: ALUOp + Funct to ALU Control - specifies exact operation.
5. **Two-stage design optimizes timing and modularity**, separating concerns.
6. **Main control signals**: RegDst, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, ALUOp.
7. **Load/Store always use ADD** for address calculation, regardless of other details.
8. **Branch uses SUBTRACT** for comparison, with Zero flag indicating equality.
9. **R-type ALU operation from funct field**, providing operation flexibility.
10. **Instruction format regularity simplifies control**, with consistent field positions.
11. **Register roles vary by instruction type**, especially RT (destination vs. source).
12. **Control signals mutually exclusive** for proper operation - only valid combinations used.
13. **Separate MemRead/MemWrite needed** for no-op state and future pipelining.
14. **Control logic is combinational** (no state), generating signals each cycle.
15. **Truth tables map opcode to control patterns**, enabling systematic design.
16. **“Don’t care” values simplify logic minimization**, reducing gate count.
17. **Control unit design uses standard digital logic techniques**, including K-maps and Boolean algebra.

18. **Datapath elements may operate but outputs ignored** if not selected by control signals.

19. **Complete processor integrates datapath and control**, with control signals orchestrating all operations.

20. **Single-cycle design simple but inefficient** - foundation for advanced multi-cycle and pipelined designs.

10.12 Summary

The control unit completes the single-cycle MIPS processor, generating control signals that orchestrate datapath operations based on instruction opcodes. The two-stage ALU control generation (opcode → ALUOp → ALU Control) elegantly separates concerns, with the main control handling instruction-level decisions and the ALU control handling operation-specific details. Each control signal serves a specific purpose, from selecting multiplexer inputs (RegDst, ALUSrc, MemtoReg) to enabling register and memory operations (RegWrite, MemRead, MemWrite) to handling branches (Branch). Truth tables systematically map instructions to control patterns, with "don't care" values simplifying logic design. While the single-cycle processor provides conceptual clarity and simplicity, its inefficiency (all instructions taking the same time as the slowest) motivates more sophisticated designs. Understanding this foundation prepares us for multi-cycle processors (which break execution into variable-length stages) and pipelined processors (which overlap instruction execution for higher throughput), both building on the control principles established here.

Chapter 11

Lecture 11: Complete Single-Cycle MIPS Processor and Performance Analysis

11.1 Introduction

This lecture completes the single-cycle MIPS processor design by providing comprehensive analysis of control signals for all instruction types (R-type, Branch, Load, Store, Jump), introducing detailed timing analysis with concrete delay values, and demonstrating the fundamental performance limitations that motivate the evolution toward multi-cycle and pipelined implementations. We build upon previous datapath and control unit knowledge to create a functioning processor while understanding why single-cycle design, though conceptually simple, proves inefficient in practice.

11.2 1. Lecture Overview and Context

11.2.1 1.1 Recap from Previous Lectures

The foundational work completed in previous lectures includes:

Completed Topics:

- Datapath components: Register file, ALU, memories, adders, multiplexers
- Sign extension and shifting for immediate operands
- Control unit concept and ALU control generation
- Control signal purposes and functions

Current Focus:

- Complete control signal analysis for all instructions
- Detailed walkthrough of instruction execution
- Jump instruction integration
- Timing analysis with concrete delay values

- Performance limitations of single-cycle design

11.2.2 1.2 Instruction Subset Review

Selected Instructions for Study:

- **R-type**: ADD, SUB, AND, OR (arithmetic/logic operations)
- **Load Word (LW)**: Memory read
- **Store Word (SW)**: Memory write
- **Branch if Equal (BEQ)**: Conditional branch
- **Jump (J)**: Unconditional jump

Coverage:

- Represents 95% of MIPS microarchitecture hardware
- Comprehensive enough for understanding design principles
- Omits some I-type arithmetic (covered conceptually)
- Foundation for complete processor understanding

11.3 2. Control Unit Inputs and Outputs

11.3.1 2.1 Control Unit Inputs

Total Input Bits: 12 bits

11.3.1.1 Primary Input - Opcode (6 bits):

- Bits 26-31 of instruction
- Identifies instruction type
- Used for almost all control signal generation
- Most significant determinant of control behavior

11.3.1.2 Secondary Input - Funct Field (6 bits):

- Bits 0-5 of instruction
- Only relevant for R-type instructions (opcode = 000000)
- Specifies ALU operation for R-type
- Ignored for I-type and J-type instructions

Usage Pattern:

- Opcode always examined
- Funct field examined only when opcode = 0 (R-type)
- Combined with ALUOp for final ALU control signal

11.3.2 2.2 Control Unit Outputs

Total Output Bits: 9 bits (8 signals, one is 2-bit)

Control Signals Generated:

1. **RegDst** (1 bit): Select register write address
2. **Branch** (1 bit): Instruction is branch type
3. **MemRead** (1 bit): Enable memory read
4. **MemtoReg** (1 bit): Select register write data source
5. **MemWrite** (1 bit): Enable memory write
6. **ALUSrc** (1 bit): Select ALU second operand source
7. **RegWrite** (1 bit): Enable register file write
8. **ALUOp** (2 bits): ALU operation category

Additional Signal for Jump:

9. **Jump** (1 bit): Select jump target for PC

Implementation:

- Combinational logic circuit
- Inputs: Opcode and Funct field
- Outputs: Control signals
- Design method: Truth tables, Karnaugh maps, Boolean minimization
- To be implemented in Lab 5

11.4 3. R-Type Instruction Detailed Analysis

11.4.1 3.1 Instruction Format

Encoding Structure (32 bits):

- **Bits 26-31**: Opcode = 000000 (0) - ALL R-type instructions
- **Bits 21-25**: RS (5 bits) - First source register
- **Bits 16-20**: RT (5 bits) - Second source register
- **Bits 11-15**: RD (5 bits) - Destination register
- **Bits 6-10**: SHAMT (5 bits) - Shift amount
- **Bits 0-5**: Funct (6 bits) - Function code (specifies operation)

Example: ADD $1, $2, $3

```
Encoding: 000000 00010 00011 00001 00000 100000
          |Opcode| RS   | RT   | RD   |SHAMT| Funct |
          |  0   |  2   |  3   |  1   |  0  |  32   |
```

Operation: `$1 = $2 + $3`

11.4.2 3.2 Datapath Elements Used

Active Elements (shown in black):

- Instruction Memory: Fetch instruction
- Program Counter: Current instruction address
- PC + 4 Adder: Calculate next sequential address
- Register File: Read RS, RT; Write RD
- Multiplexer (RegDst): Select RD as write address
- Multiplexer (ALUSrc): Select RT value (not immediate)
- ALU: Perform operation specified by funct
- Multiplexer (MemtoReg): Select ALU result (not memory)
- Multiplexer (PC source): Select PC+4 (not branch)

Inactive Elements (grayed out):

- Data Memory: Not accessed
- Sign Extender: Not used (no immediate value)
- Branch Target Adder: Calculated but not used
- Shift Left 2: Not used

11.4.3 3.3 Control Signal Values for R-Type

Exercise Example: ADD $1, $2, $3

Signal	Value	Reason
RegDst	1	Write to RD (bits 11-15), not RT
Branch	0	Not a branch instruction
MemRead	0	Not reading from memory
MemtoReg	0	Write ALU result (not memory data)
ALUOp	10	R-type: Consult funct field
MemWrite	0	Not writing to memory
ALUSrc	0	Second operand from register RT (not immediate)
RegWrite	1	Write result to destination register

Detailed Explanations:

RegDst = 1:

- Multiplexer selects input 1
- Input 1: Bits 11-15 (RD field)
- Input 0: Bits 16-20 (RT field)
- R-type destination always in RD

Branch = 0:

- Not a branch instruction
- Branch control AND Zero → 0 AND X = 0
- PC source multiplexer selects PC+4

MemRead = 0, MemWrite = 0:

- R-type doesn’t access data memory
- Memory control signals disabled
- Data memory outputs ignored (don’t care)

MemtoReg = 0:

- Multiplexer selects ALU result
- Not memory data (memory not accessed)
- ALU result goes to register write data

ALUOp = 10 (binary):

- Indicates R-type instruction
- ALU Control Unit examines funct field
- For ADD: funct = 100000 → ALU Control = 0010 (ADD)

ALUSrc = 0:

- Multiplexer selects register value
- Register file Read Data 2 (RT value)
- Not sign-extended immediate

RegWrite = 1:

- Enable register file write
- Result written to RD at clock edge
- Essential for saving computation result

11.4.4 3.4 Execution Steps for R-Type

Step 1: Instruction Fetch

- PC value → Instruction Memory address
- Instruction word retrieved
- Opcode (000000) sent to Control Unit

Step 2: Control Signal Generation

- Control Unit decodes opcode = 0
- Identifies R-type instruction
- Generates all control signals
- Sends funct field to ALU Control

Step 3: Register Read

- RS field (00010 = 2) → Read Address 1
- RT field (00011 = 3) → Read Address 2
- Read Data 1 = $2 value
- Read Data 2 = $3 value

Step 4: ALU Operation

- ALUSrc = 0: Select RT value for Input B
- Input A = $2 value, Input B = $3 value
- ALU Control = 0010 (ADD operation)
- ALU Result = $2 + $3

Step 5: Register Write Preparation

- MemtoReg = 0: Select ALU result

- RegDst = 1: Select RD (00001 = 1)
- Write Data = ALU result
- Write Address = $1

Step 6: Clock Edge Actions

- RegWrite = 1 enabled
- ALU result written to $1
- PC updated to PC + 4
- Next instruction fetch begins

11.5 4. Branch If Equal Instruction Detailed Analysis

11.5.1 4.1 Instruction Format

Encoding Structure (32 bits):

- **Bits 26-31**: Opcode = 000100 (4) - BEQ
- **Bits 21-25**: RS (5 bits) - First comparison register
- **Bits 16-20**: RT (5 bits) - Second comparison register
- **Bits 0-15**: Immediate (16 bits) - Branch offset (in instructions)

Example: BEQ $1, $2, 100

```
Encoding: 000100 00001 00010 0000000001100100
         |Opcode|  RS |  RT |    Immediate      |
         |  4   |  1  |  2  |       100         |
```

Operation: `If ($1 == $2) then PC = PC + 4 + (100 × 4)`

11.5.2 4.2 Datapath Elements Used

Active Elements:

- Instruction Memory: Fetch instruction
- Program Counter & PC+4 Adder
- Register File: Read RS, RT (no write)
- ALU: Subtract RT from RS
- Zero Flag: Compare result to zero
- Sign Extender: Extend 16-bit offset to 32-bit
- Shift Left 2: Convert word offset to byte offset
- Branch Target Adder: Calculate PC + 4 + (offset × 4)
- AND Gate: Combine Branch signal and Zero flag
- PC Source Multiplexer: Select next PC value

Inactive Elements:

- Data Memory: Not accessed
- Register Write: Not writing to registers
- ALU Result (except Zero flag): Not used

11.5.3 4.3 Control Signal Values for BEQ

Exercise Example: BEQ $1, $2, 100

Signal	Value	Reason
RegDst	X	Don't care (not writing to register)
Branch	1	This IS a branch instruction
MemRead	0	Not reading from memory
MemtoReg	X	Don't care (not writing to register)
ALUOp	01	Perform SUBTRACT for comparison
MemWrite	0	Not writing to memory
ALUSrc	0	Compare two register values (not immediate)
RegWrite	0	Not writing to register file

Detailed Explanations:

RegDst = X (Don't Care):

- RegWrite = 0, so write address irrelevant
- No register write operation
- Multiplexer output ignored
- Using X simplifies Boolean logic

Branch = 1:

- Identifies instruction as branch type
- Feeds into AND gate with Zero flag
- PCSrc = Branch AND Zero
- If Zero = 1 (values equal): Take branch
- If Zero = 0 (values differ): Don't take branch

MemRead = 0, MemWrite = 0:

- Branch doesn't access memory
- Memory control signals disabled

MemtoReg = X (Don't Care):

- RegWrite = 0, so write data source irrelevant
- Multiplexer output ignored

ALUOp = 01:

- Specifies SUBTRACT operation
- ALU Control receives 01
- Generates ALU Control = 0110 (SUB)
- Independent of funct field

ALUSrc = 0:

- Need RT value from register (not immediate)
- Immediate used for branch target (not ALU input)
- ALU compares RS and RT register values

RegWrite = 0:

- Branch doesn't modify registers
- Essential to prevent accidental writes
- If =1, would corrupt register file

11.5.4 4.4 Branch Target Calculation

Word Offset to Byte Offset:

- Immediate field: 100 (in instructions/words)
- Sign extend to 32 bits: 0x00000064
- Shift left by 2: 0x00000190 (multiply by 4)
- Result: 400 bytes (100 instructions × 4 bytes/instruction)

Branch Target Address:

- Current PC + 4: Address of next sequential instruction
- Offset: 400 bytes
- Branch Target = (PC + 4) + 400

Example:

- Current instruction at address 1000
- PC + 4 = 1004
- Branch Target = 1004 + 400 = 1404
- If branch taken: Next instruction at 1404
- If branch not taken: Next instruction at 1004

PCSrc Selection:

```
PCSrc = Branch AND Zero
      = 1 AND (RS == RT ? 1 : 0)

If PCSrc = 1: PC ← Branch Target (1404)
If PCSrc = 0: PC ← PC + 4 (1004)
```

11.6 5. Load Word Instruction Detailed Analysis

11.6.1 5.1 Instruction Format

Encoding Structure (32 bits):

- **Bits 26-31**: Opcode = 100011 (35) - LW
- **Bits 21-25**: RS (5 bits) - Base address register
- **Bits 16-20**: RT (5 bits) - Destination register
- **Bits 0-15**: Immediate (16 bits) - Address offset

Example: LW $8, 32($9)

```
Encoding: 100011 01001 01000 0000000000100000
         |Opcode|  RS |  RT |    Immediate     |
         | 35   |  9  |  8  |       32         |
```

Operation: `$8 = Memory[$9 + 32]`

11.6.2 5.2 Datapath Elements Used

Active Elements:

- Instruction Memory: Fetch instruction
- Program Counter & PC+4 Adder
- Register File: Read RS (base); Write RT (destination)
- Sign Extender: Extend offset to 32 bits
- Multiplexer (ALUSrc): Select immediate
- ALU: Add base + offset
- Data Memory: Read at calculated address
- Multiplexer (MemtoReg): Select memory data
- Multiplexer (RegDst): Select RT for write

Inactive Elements:

- Second register read (RT as source): Not used
- Branch circuitry: Not used

11.6.3 5.3 Control Signal Values for LW

Exercise Example: LW $8, 32($9)

Signal	Value	Reason
RegDst	0	Write to RT (bits 16-20), not RD
Branch	0	Not a branch instruction
MemRead	1	Reading from data memory
MemtoReg	1	Write memory data (not ALU result)
ALUOp	00	Perform ADD for address calculation
MemWrite	0	Not writing to memory (reading only)
ALUSrc	1	Add immediate offset (not register)
RegWrite	1	Write loaded data to destination register

Detailed Explanations:

RegDst = 0:

- I-type format: Destination in RT field
- Multiplexer selects bits 16-20
- RT = 01000 (register 8)
- Different from R-type (RD field)

Branch = 0:

- Sequential execution
- PC updated to PC + 4

MemRead = 1:

- Enable data memory read
- Essential for memory timing
- Memory outputs data at calculated address
- If 0: Memory output undefined (ignored anyway)

MemtoReg = 1:

- Multiplexer selects memory data
- Input 1: Data memory read output
- Input 0: ALU result (address, not data!)
- Must select memory data for load

ALUOp = 00:

- Address calculation requires ADD
- Base address + offset
- ALU Control = 0010 (ADD)

MemWrite = 0:

- Reading, not writing
- Critical: Prevents memory corruption
- If 1: Would write garbage to memory

ALUSrc = 1:

- Need immediate offset for address calculation
- Multiplexer selects sign-extended immediate
- Input 1: Sign-extended offset
- Input 0: RT value (not used for address calc)

RegWrite = 1:

- Must write loaded data to RT
- Data from memory → Register $8
- If 0: Data lost, load ineffective

11.6.4 5.4 Critical Path for Load Word

Longest Delay in Single-Cycle:

1. Instruction Memory read
2. Register File read (base address)
3. Sign Extension
4. ALU address calculation
5. Data Memory read
6. Register write setup

Load Word is the slowest instruction!

- Determines minimum clock period
- All other instructions must wait for this worst case
- Major performance bottleneck

11.7 6. Store Word Instruction Detailed Analysis

11.7.1 6.1 Instruction Format

Encoding Structure (32 bits):

- **Bits 26-31**: Opcode = 101011 (43) - SW
- **Bits 21-25**: RS (5 bits) - Base address register
- **Bits 16-20**: RT (5 bits) - Source data register
- **Bits 0-15**: Immediate (16 bits) - Address offset

Example: SW $8, 32($9)

```
Encoding: 101011 01001 01000 0000000000100000
         |Opcode|  RS |  RT |    Immediate     |
         | 43   |  9  |  8  |        32        |
```

Operation: `Memory[$9 + 32] = $8`

Note: Fixed error in lecture (was “$32”, should be “32”)

11.7.2 6.2 Datapath Elements Used

Active Elements:

- Instruction Memory
- Program Counter & PC+4 Adder
- Register File: Read RS (base) AND RT (data source)
- Sign Extender
- Multiplexer (ALUSrc): Select immediate
- ALU: Add base + offset
- Data Memory: Write RT data at calculated address

Inactive Elements:

- Register Write: No register write
- Memory Read: Writing, not reading
- MemtoReg multiplexer: Output not used

Key Difference from Load:

- TWO register reads: RS for base, RT for data
- Memory write instead of read
- NO register write operation

11.7.3 6.3 Control Signal Values for SW

Exercise Example: SW $8, 32($9)

Signal	Value	Reason
RegDst	X	Don't care (not writing to register)
Branch	0	Not a branch instruction
MemRead	0	Not reading from memory (writing)
MemtoReg	X	Don't care (not writing to register)
ALUOp	00	Perform ADD for address calculation
MemWrite	1	Writing to data memory
ALUSrc	1	Add immediate offset
RegWrite	0	Not writing to register file

Detailed Explanations:

RegDst = X (Don't Care):

- RegWrite = 0: No register write
- Write address irrelevant
- Could be 0 or 1, doesn't matter
- Using X simplifies logic design

CRITICAL: RegWrite = 0:

- Must prevent register file write
- **If RegWrite = 1:** Disaster!
  - Some register address fed to write port
  - Either ALU result (address) or memory data (garbage, MemRead=0)
  - Would corrupt random register
  - Data integrity violated

Why It Matters:

- Hardware operates in parallel
- Multiplexers produce outputs even if not used
- Without RegWrite = 0:
  - RegDst mux outputs some address
  - MemtoReg mux outputs some data
  - If RegWrite = 1: This garbage written to register!
- Control signal correctness essential

MemRead = 0, MemWrite = 1:

- Writing to memory, not reading
- MemRead = 0: Memory read output undefined
- MemWrite = 1: Memory accepts write data
- Opposite of Load Word

MemtoReg = X (Don't Care):

- RegWrite = 0: Write data source irrelevant

- Output not used
- Even if wrong data selected, RegWrite prevents write

ALUOp = 00:

- Same as Load Word
- Address calculation: ADD operation

ALUSrc = 1:

- Need immediate offset
- Same as Load Word

11.7.4 6.4 Important Lesson: Don't Care vs Zero

Student Confusion: *"RegDst = 0 is not wrong, but best answer is X"*

Clarification:

- **Functionally:** 0 works (doesn't cause error)
- **Logically:** X is correct (truly doesn't matter)
- **Design perspective:** X simplifies Boolean expressions
- **Karnaugh map minimization:** X allows more groupings

However:

- RegWrite MUST be 0 (not X!)
- MemWrite MUST be correct (not X!)
- Read/Write enables are critical for data integrity

11.8 7. Jump Instruction Integration

11.8.1 7.1 Instruction Format

Encoding Structure (32 bits):

- **Bits 26-31**: Opcode = 000010 (2) - J
- **Bits 0-25**: Address (26 bits) - Jump target (word address)

Alternative: JAL (Jump and Link)

- Opcode = 000011 (3)
- Used for function calls
- Saves return address in register $31

Example: J 100

```
Encoding: 000010 00000000000000000001100100
         |Opcode|        Target Address         |
         |  2   |            100                |
```

Operation: `PC = {PC+4[31:28], Address, 2'b00}`

11.8.2 7.2 Jump Target Address Calculation

Word Address to Byte Address:

- Target field: 26 bits (word address)
- Shift left by 2: Append 2 zero bits
- Result: 28-bit byte address

Upper 4 Bits:

- Take from PC+4 current value
- Bits 31:28 of next sequential instruction
- Preserves region (256 MB regions)
- Jump within same region as current PC

Concatenation:

```
PC+4:         [31:28] [27:2] [1:0]
              ↓       (ignored)
Jump Target:  [31:28] [Target×4] [00]
              ↑       ↑          ↑
              From    From       Append
              PC+4    instruction zeros
```

Example:

- PC = 0x10000000
- PC+4 = 0x10000004
- Target = 100 = 0x000064
- Shift left 2: 0x000190
- Upper 4 bits: 0x1
- Jump Address: 0x10000190

Limitation:

- Can only jump within 28-bit range (256 MB)
- Upper 4 bits fixed by current PC region
- For larger jumps: Use jump register (JR) instruction

11.8.3 7.3 Additional Datapath Hardware

New Components:

Shift Left 2 (for jump):

- Input: 26-bit target field
- Output: 28-bit byte offset
- Implementation: Wire routing (no actual shifter!)

Concatenation Logic:

- Input 1: PC+4 bits [31:28] (4 bits)
- Input 2: Shifted target (28 bits)
- Output: 32-bit jump address
- Implementation: Wire concatenation

New Multiplexer:

- **Input 0**: Output from branch/sequential mux
  - Could be PC+4 or branch target
- **Input 1**: Jump target address (32 bits)
- **Select**: Jump control signal
- **Output**: Next PC value

Original PC Source Mux:

- Input 0: PC + 4
- Input 1: Branch target
- Select: PCSrc (Branch AND Zero)

New Jump Mux (outer):

- Input 0: Original mux output (PC+4 or branch target)
- Input 1: Jump target
- Select: Jump signal
- Output: Final next PC value

11.8.4 7.4 Jump Control Signal

Jump Signal:

- 10th control output bit
- Generated by Control Unit
- Based on opcode = 2 (J) or 3 (JAL)

Values:

- Jump = 1: Select jump target
- Jump = 0: Select sequential/branch

Other Control Signals for Jump:

Signal	Value	Reason
RegDst	X	Don't care
Branch	0	Not a branch (different mechanism)
MemRead	0	Not accessing memory
MemtoReg	X	Don't care
ALUOp	XX	Don't care (ALU not used)
MemWrite	0	Not writing memory
ALUSrc	X	Don't care
RegWrite	0	Not writing register (J instruction)
Jump	1	This IS a jump instruction

Note: JAL (Jump and Link) different:

- RegWrite = 1 (saves return address)
- RegDst = ? (special: write to $31)
- Additional logic needed for return address

11.8.5 7.5 Complete Datapath with Jump

All Instruction Types Supported:

- **R-type**: Arithmetic, logic, shift
- **I-type**: Load, Store, Branch, Immediate arithmetic
- **J-type**: Jump, Jump and Link

Coverage:

- 95%+ of MIPS ISA hardware
- Complete single-cycle implementation
- Additional variants (BNE, shifts, etc.) need minor additions

Datapath Completeness:

- Two memories: Instruction and Data
- One ALU for computation
- Multiple adders: PC+4, Branch target
- Many multiplexers for data routing
- Sign extender
- Shift left 2 circuits (wire routing)
- Control unit with 10 control signal bits

11.9 8. Timing Analysis with Concrete Delays

11.9.1 8.1 Assumed Component Delays

Delay Values (in nanoseconds):

Component	Delay	Notes
Instruction Memory	2 ns	Read instruction at PC address
Register File (Read)	1 ns	Output data after address change
Register File (Write)	1 ns	At clock edge (next cycle)
Sign Extender	~0 ns	Negligible (wire replication)
Multiplexers	~0 ns	Negligible compared to other delays
ALU Operation	2 ns	Arithmetic/logic/comparison
Data Memory (Read)	2 ns	Output data after address provided
Data Memory (Write)	2 ns	At clock edge (next cycle)
PC+4 Adder	2 ns	Simple addition
Branch Target Adder	2 ns	Addition with offset

Assumptions:

- Simplified for analysis
- Real delays depend on technology, circuit design
- Memory accesses typically slowest
- Combinational logic relatively fast

11.9.2 8.2 Critical Path Analysis

Definition:

- Longest delay path from clock edge to clock edge
- Determines minimum clock period
- All combinational logic between sequential elements

Single-Cycle Constraint:

- Entire instruction must complete in one clock cycle
- Clock period ≥ Critical path delay
- All instructions take same time (worst case)

11.9.3 8.3 Load Word Instruction Timing

Step-by-Step Delay Calculation:

Step 1: Instruction Fetch (2 ns)

- Clock edge: PC updated
- PC → Instruction Memory
- Instruction Memory reads and outputs instruction
- Delay: 2 ns
- Running total: 2 ns

Step 2: Register Read (1 ns)

- Instruction decoded
- RS field extracted
- RS → Register File Read Address 1
- Register File outputs base address
- Delay: 1 ns
- Running total: 2 + 1 = 3 ns

Step 3: Sign Extension (~0 ns)

- Immediate field extracted
- Sign extended to 32 bits
- Delay: Negligible
- Running total: ~3 ns

Step 4: ALU Address Calculation (2 ns)

- Base address + offset
- ALU performs addition
- Output: Memory address
- Delay: 2 ns
- Running total: 3 + 2 = 5 ns

Step 5: Memory Read (2 ns)

- ALU result → Data Memory address
- MemRead = 1 asserted
- Data Memory reads and outputs data

- Delay: 2 ns
- Running total: 5 + 2 = 7 ns

Step 6: Register Write Setup (~0 ns)

- Memory data → Register Write Data input
- RT → Register Write Address
- Ready for clock edge
- Delay: Setup time negligible
- Running total: ~7 ns

Clock Edge: Register Write (next cycle)

- At next positive clock edge
- Data written to register
- Takes 1 ns but in next cycle

Minimum Clock Period: 7 nanoseconds
Maximum Clock Frequency: 1/7 ns ≈ 143 MHz

Load Word is Critical Path!

- Longest instruction in single-cycle design
- Determines clock period for ALL instructions

11.9.4 8.4 Store Word Instruction Timing

Step-by-Step Delay:

1. **Instruction Fetch:** 2 ns (total: 2 ns)
2. **Register Read:** 1 ns (total: 3 ns)
    - Read RS (base) AND RT (data)
3. **Sign Extension:** ~0 ns (total: 3 ns)
4. **ALU Address Calculation:** 2 ns (total: 5 ns)
5. **Memory Write Setup:** ~0 ns (total: 5 ns)
    - Address and data ready at memory inputs

Clock Edge: Memory Write (end of cycle)

- Data written to memory at clock edge
- Takes 2 ns but next instruction fetch also 2 ns
- Next instruction register read starts after 3 ns total
- Memory write completes before register read needs data
- No conflict

Minimum Time Required: 5 nanoseconds

Note:

- Faster than Load Word (no memory read delay)
- But must use 7 ns clock period anyway (single-cycle)
- Wastes 2 ns per Store instruction

11.9.5 8.5 Arithmetic Instruction Timing (ADD, SUB, AND, OR)

Step-by-Step Delay:

1. **Instruction Fetch:** 2 ns (total: 2 ns)
2. **Register Read:** 1 ns (total: 3 ns)
    - Read RS and RT
3. **ALU Operation:** 2 ns (total: 5 ns)
    - Perform arithmetic/logic operation
4. **Register Write Setup:** ~0 ns (total: 5 ns)
    - ALU result ready at register write data input

Clock Edge: Register Write

- Result written to RD

Minimum Time Required: 5 nanoseconds

Efficiency Loss:

- Could run at 5 ns clock period
- Forced to wait 7 ns (Load Word limitation)
- Wastes 2 ns = 28.6% time wasted per R-type instruction

11.9.6 8.6 Branch Instruction Timing

Step-by-Step Delay:

1. **Instruction Fetch:** 2 ns (total: 2 ns)
2. **Register Read:** 1 ns (total: 3 ns)
    - Read RS and RT for comparison
3. **ALU Comparison:** 2 ns (total: 5 ns)
    - Subtract RS - RT
    - Generate Zero flag
4. **Branch Target Calculation:** 2 ns (parallel with ALU)
    - Sign extend offset: ~0 ns
    - Shift left 2: ~0 ns (wire routing)
    - Add to PC+4: 2 ns
    - Can happen in parallel with ALU operation!
5. **PC Update Setup:** ~0 ns (total: 5 ns)
    - Zero flag + Branch → PCSrc
    - Multiplexer selects next PC
    - Ready for clock edge

Minimum Time Required: 5 nanoseconds

Key Insight:

- Branch target calculation parallel to ALU
- PC+4 already available from fetch stage
- No memory access needed
- Fast like R-type

11.9.7 8.7 Jump Instruction Timing

Step-by-Step Delay:

1. **Instruction Fetch:** 2 ns (total: 2 ns)
    - Also calculates PC+4 in parallel
2. **Jump Target Calculation:** ~0 ns
    - Extract 26-bit target
    - Shift left 2: Wire routing, ~0 ns
    - Concatenate with PC+4[31:28]: Wire connection, ~0 ns
    - No ALU, no memory, no registers!
3. **PC Update Setup:** ~0 ns (total: 2 ns)

Minimum Time Required: 2 nanoseconds

Fastest Instruction:

- Only instruction fetch needed
- Jump target calculation: Wire operations only
- No sequential dependencies
- Wastes 5 ns waiting for clock period!

11.9.8 8.8 Timing Summary Table

Instruction Type	Time Required	Wasted Time	Efficiency
Load Word (LW)	7 ns	0 ns	100%
Store Word (SW)	5 ns	2 ns	71.4%
R-type (ADD, etc.)	5 ns	2 ns	71.4%
Branch (BEQ)	5 ns	2 ns	71.4%
Jump (J)	2 ns	5 ns	28.6%

Clock Period (Single-Cycle): 7 ns (determined by LW)
Clock Frequency: ~143 MHz

Performance Impact:

- Most instructions waste time
- Only Load Word fully utilizes clock cycle
- Tremendous inefficiency

11.10 9. Performance Analysis

11.10.1 9.1 Program Composition Example

Typical MIPS Program Profile:

Instruction Type	Percentage	Time if Variable	Time (Fixed 7ns)
Arithmetic	48%	5 ns	7 ns
Load Word	22%	7 ns	7 ns
Store Word	11%	5 ns	7 ns
Branch	19%	5 ns	7 ns

11.10.2 9.2 Average Time Calculation

Variable Time (Ideal):

```
Average = (0.48 × 5) + (0.22 × 7) + (0.11 × 5) + (0.19 × 5)
        = 2.40 + 1.54 + 0.55 + 0.95
        = 5.44 ns per instruction
```

Single-Cycle (Actual):

```
Average = 7 ns per instruction (all instructions)
```

Performance Loss:

```
Overhead = 7 - 5.44 = 1.56 ns per instruction
Efficiency = 5.44 / 7 = 77.7%
Waste = 22.3% of time
```

11.10.3 9.3 Critical Path Problem

Critical Path Determination:

- Load Word uses most datapath elements
- Sequential dependencies:
    1. Instruction Memory
    2. Register File
    3. ALU
    4. Data Memory
    5. (Register Write in next cycle)

Design Principle Violation:

- **"Make the common case fast"**
- Common case: Arithmetic instructions (48%)
- Slow case (Load Word) determines speed
- Common case forced to slow down
- Design is inefficient

11.10.4 9.4 Clock Period Inflexibility

Single-Cycle Constraint:

- Clock period MUST be constant
- Cannot vary by instruction
- Must accommodate worst case (slowest instruction)

- All faster instructions penalized

Implications:

- Arithmetic: Could run at 143 MHz, forced to 143 MHz □
- Load: Needs 143 MHz, gets 143 MHz □
- Jump: Could run at 500 MHz, forced to 143 MHz □

Efficiency by Instruction:

Instruction	Efficiency	Waste
Jump	28.6%	71.4%
Arithmetic	71.4%	28.6%
Store	71.4%	28.6%
Branch	71.4%	28.6%
Load	100.0%	0%

11.11 10. Path to Better Performance: Multi-Cycle Design

11.11.1 10.1 Multi-Cycle Concept

Basic Idea:

- Break instruction execution into multiple stages
- Each stage completes in one (shorter) clock cycle
- Different instructions use different number of cycles
- Only use stages actually needed

Advantages:

- Shorter clock period (faster clock)
- Instructions take only time they need
- Better average performance
- More efficient resource utilization

11.11.2 10.2 Stage Division

Typical Stages:

Stage 1: Instruction Fetch (IF)

- Read from instruction memory
- Update PC to PC+4
- Store instruction in register

Stage 2: Instruction Decode (ID)

- Decode opcode

- Read registers
- Generate control signals
- Sign extend immediate

Stage 3: Execute (EX)

- ALU operation
- Or address calculation
- Or branch comparison

Stage 4: Memory Access (MEM)

- Read from data memory (if load)
- Write to data memory (if store)
- Or skip this stage

Stage 5: Write-Back (WB)

- Write result to register file
- Or skip if no write needed

Not All Instructions Use All Stages:

- **R-type**: IF, ID, EX, WB (skip MEM) = 4 cycles
- **Load**: IF, ID, EX, MEM, WB (all stages) = 5 cycles
- **Store**: IF, ID, EX, MEM (skip WB) = 4 cycles
- **Branch**: IF, ID, EX (skip MEM, WB) = 3 cycles
- **Jump**: IF, ID (skip EX, MEM, WB) = 2 cycles

11.11.3 10.3 Clock Period in Multi-Cycle

Determining Clock Period:

- Clock period = Longest stage delay
- NOT longest instruction delay
- Much shorter than single-cycle

Example Stage Delays:

Stage	Delay
IF (Instr Memory)	2 ns
ID (Register Read)	1 ns
EX (ALU)	2 ns
MEM (Data Memory)	2 ns
WB (Register Write)	1 ns

Longest Stage: 2 ns
Clock Period: 2 ns (vs 7 ns single-cycle)
Clock Frequency: 500 MHz (vs 143 MHz single-cycle)

11.11.4 10.4 Performance Comparison

Single-Cycle:

```
All instructions: 1 cycle × 7 ns = 7 ns
```

Multi-Cycle (with 2 ns clock):

Instruction	Cycles	Time
Arithmetic	4	8 ns
Load	5	10 ns
Store	4	8 ns
Branch	3	6 ns
Jump	2	4 ns

Weighted Average (same program profile):

```
Average = (0.48 × 8) + (0.22 × 10) + (0.11 × 8) + (0.19 × 6)
        = 3.84 + 2.20 + 0.88 + 1.14
        = 8.06 ns per instruction
```

Wait, That's Worse!

- Multi-cycle: 8.06 ns average
- Single-cycle: 7 ns always
- Multi-cycle slower?!

Resolution:

- Example delays assumed equal stage times
- In reality, stages have different delays
- Need to balance stage delays
- Goal: Make all stages approximately equal
- Then multi-cycle becomes efficient

Ideal Multi-Cycle (balanced 1.4 ns stages):

Instruction	Cycles	Time
Arithmetic	4	5.6 ns
Load	5	7.0 ns
Store	4	5.6 ns
Branch	3	4.2 ns

```
Average = (0.48 × 5.6) + (0.22 × 7.0) + (0.11 × 5.6) + (0.19 × 4.2)
        = 2.69 + 1.54 + 0.62 + 0.80
        = 5.65 ns per instruction

Speedup = 7 / 5.65 = 1.24× faster
```

11.11.5 10.5 Design Challenge

Stage Balancing:

- Goal: Roughly equal delay per stage
- Challenge: Memory slower than ALU
- Memory stage limits clock period
- Need techniques:
  - Faster memory
  - Cache memory (next topic)
  - Pipeline (next lecture)

Resource Reuse:

- Single ALU used across multiple cycles
- Single memory port can be reused
- Fewer hardware resources needed
- More control complexity (FSM needed)

11.12 11. Preview: Pipelining

11.12.1 11.1 Next Step Beyond Multi-Cycle

Pipelining Concept:

- Multiple instructions in flight simultaneously
- Each instruction at different stage
- Like assembly line
- **Stage 1:** Fetch instruction A
- **Stage 2:** Decode A, Fetch B
- **Stage 3:** Execute A, Decode B, Fetch C
- **Stage 4:** Memory A, Execute B, Decode C, Fetch D
- **Stage 5:** Write A, Memory B, Execute C, Decode D, Fetch E

Benefits:

- One instruction completes per cycle (like single-cycle)
- But clock period short (like multi-cycle)
- Best of both worlds
- Dramatic performance improvement

Challenges (Covered Next Lecture):

- Hazards: Data dependencies between instructions
- Control hazards: Branches affect pipeline
- Structural hazards: Resource conflicts
- Need forwarding and stall logic
- More complex control

11.12.2 11.2 Coming Next

Topics:

- Pipelined datapath design
- Hazard detection and resolution
- Forwarding (bypassing)
- Branch prediction
- Performance analysis
- MIPS pipeline implementation

11.13 Key Takeaways

1. **Single-cycle design executes each instruction in one clock cycle**, with clock period determined by the slowest instruction (Load Word at 7 ns).

2. **Control unit generates signals based on opcode**, orchestrating datapath operations for R-type, Load, Store, Branch, and Jump instructions.

3. **Load Word is the critical path** (Instruction Fetch → Register Read → ALU → Memory Read → Register Write), determining minimum clock period.

4. **Jump instruction uses PC[31:28]** concatenated with shifted immediate to form 32-bit target address, enabling 256 MB jump range.

5. **Control signals must prevent data corruption**, with RegWrite=0 for Store and Branch to avoid unintended register modifications.

6. **"Don't care" values (X) simplify control logic**, allowing optimization when signals don't affect instruction outcome.

7. **Hardware operates concurrently**, not sequentially—multiple operations happen simultaneously within each clock cycle.

8. **Performance inefficiency drives design evolution**, as most instructions finish early but must wait for full clock period.

9. **Resource utilization varies dramatically**, with arithmetic instructions using ~43% of clock period while Load uses 100%.

10. **Timing analysis reveals optimization opportunities**, showing that memory access dominates critical path (4 ns of 7 ns total).

11. **Write operations occur at clock edge**, ensuring data stability and preventing race conditions in sequential logic.

12. **Branch target calculation happens in parallel** with ALU comparison, optimizing branch instruction timing.

13. **Sign extension is effectively instantaneous** (combinational logic), adding negligible delay to critical path.

14. **Clock period sets maximum frequency** (~143 MHz for 7 ns period), directly impacting overall processor performance.
15. **Common case (arithmetic) runs slowly**, violating fundamental design principle of making common case fast.
16. **Stage division concept emerges from timing analysis**, suggesting multi-cycle implementation could improve efficiency.
17. **Control signal truth tables systematically define behavior**, mapping each instruction to specific control patterns.
18. **PC update mechanisms vary by instruction type**, using PC+4, branch target, or jump target based on control signals.
19. **Data memory access only for Load/Store**, with MemRead and MemWrite controlling when memory participates in execution.
20. **Performance analysis quantifies inefficiency**, providing concrete motivation for pipelined processor designs in subsequent lectures.

11.14 Summary

The single-cycle MIPS processor represents a complete, functioning implementation where each instruction executes in exactly one clock cycle. While conceptually straightforward and easy to understand, the design reveals fundamental performance limitations that drive modern processor architecture evolution. The critical path analysis shows Load Word requiring 7 nanoseconds while simpler instructions like arithmetic operations complete in just 3 nanoseconds, forcing all instructions to wait for the slowest operation. This inefficiency—with most instructions utilizing less than half the available clock period—violates the crucial design principle of “making the common case fast.” The systematic control signal analysis demonstrates how the control unit orchestrates datapath operations for different instruction types (R-type, Load, Store, Branch, Jump), with careful attention to preventing data corruption through proper RegWrite and MemWrite signals. The jump instruction introduces pseudo-direct addressing, concatenating PC upper bits with shifted immediate for 256 MB addressability. While the single-cycle design provides essential conceptual foundation for understanding processor operation, the detailed timing analysis and resource utilization metrics clearly motivate the need for more sophisticated approaches—multi-cycle processors that divide execution into variable-length stages, and pipelined processors that overlap instruction execution for dramatically improved throughput. These performance limitations aren’t flaws but rather inevitable consequences of the single-cycle constraint, establishing why modern processors universally adopt pipelining despite the additional complexity it introduces.

Chapter 12

Lecture 12: Pipelining and Hazards in MIPS Processors

12.1 Introduction

This lecture introduces pipelining as the primary performance enhancement technique in modern processor design, transforming the inefficient single-cycle architecture into a high-throughput execution engine. We explore how pipelining applies assembly-line principles to instruction execution, dramatically improving processor throughput while maintaining individual instruction latency. The lecture examines the three fundamental types of hazards—structural, data, and control—that threaten pipeline efficiency, and discusses practical solutions including forwarding, stalling, and branch prediction that enable real-world pipelined processors to achieve near-ideal performance.

12.2 1. Recap: Single-Cycle Performance Limitations

12.2.1 1.1 Critical Path Problem

Load Word as Bottleneck:

- Uses most resources: Instruction Memory → Register File → ALU → Data Memory → Register File
- Determines clock period for entire CPU
- Forces all other instructions to wait

Performance Issue:

- Most instructions (arithmetic, branch) take less time than load
- Jump instruction takes even less time
- Clock period set by slowest instruction (load word)

Design Principle Violated:

- “Make the common case fast”

- Common case (arithmetic) forced to run slowly
- Majority of instructions underutilize available time

12.2.2 1.2 Multi-Cycle as First Improvement

Basic Concept:

- Divide datapath into stages
- Each stage completes in one clock cycle
- Shorter clock cycles than single-cycle

Five Stages Identified:

1. Instruction Fetch (IF)
2. Register Reading
3. ALU Operations
4. Memory Access
5. Register Writing

Variable Stage Usage:

- Load: Uses all 5 stages
- Most instructions: Skip memory access (4 stages)
- Jump: Only 2 stages (manipulating PC)

Clock Period Determination:

- Decided by slowest stage (not slowest instruction)
- Adjust work in each stage for balance
- Maximize utilization of each clock cycle

Limitation:

- Instruction must finish before next instruction starts
- Hardware still idle during many cycles
- Room for further improvement

12.3 2. Pipelining Concept: The Laundry Shop Analogy

12.3.1 2.1 Non-Pipelined Laundry Shop

Setup:

- One employee
- Four customers: A, B, C, D
- First-come, first-serve basis
- Four stages of work per customer:
  1. Washing: 30 minutes
  2. Drying: 30 minutes

3. Folding/Ironing: 30 minutes
4. Packaging: 30 minutes

- Total per customer: 2 hours

Sequential Processing:

```
Time:  6pm    7pm    8pm    9pm    10pm   11pm   12am   1am    2am
A:     [Wash] [Dry]  [Fold] [Pack]
B:                            [Wash] [Dry]  [Fold] [Pack]
C:                                                   [Wash] [Dry]  [Fold] [Pack]
D:                                                                          [Wash] [D
```

Metric	Value
Total Time	8 hours (6pm to 2am)
Time per Customer	2 hours
Shop Closes	2am

Problems:

- Machines idle while employee works on other stages
- Washer idle during drying, folding, packaging
- Dryer idle except during drying stage
- Tremendous resource underutilization

12.3.2 2.2 Pipelined Laundry Shop

Key Idea:

- Use idle machines for next customers
- Overlap execution of different loads
- Parallel processing maximizes hardware utilization

Pipelined Schedule:

```
Time:  6:00   6:30   7:00   7:30   8:00   8:30   9:00   9:30
A:     [Wash] [Dry]  [Fold] [Pack]
B:            [Wash] [Dry]  [Fold] [Pack]
C:                   [Wash] [Dry]  [Fold] [Pack]
D:                          [Wash] [Dry]  [Fold] [Pack]
E:                                 [Wash] [Dry]  [Fold] [Pack]
```

Timeline Analysis:

- 6:00-6:30: A washing (1 station busy)
- 6:30-7:00: A drying, B washing (2 stations busy)
- 7:00-7:30: A folding, B drying, C washing (3 stations busy)
- 7:30-8:00: A packing, B folding, C drying, D washing (4 stations - **ALL BUSY!**)
- 8:00-8:30: A done, B packing, C folding, D drying, E washing

Steady State:

- Reached at 7:30-8:00 when all 4 stations occupied

- Pipeline full
- Maximum hardware utilization
- One customer finishes every 30 minutes

12.3.3 2.3 Performance Analysis

Time Comparison:

- Non-pipelined: 8 hours for 4 customers
- Pipelined: 3.5 hours for 4 customers

Speedup Calculation:

```
Speedup = Non-pipelined Time / Pipelined Time
        = 8 hours / 3.5 hours
        = 2.3×
```

Includes Pipeline Fill Time:

- First 1.5 hours: Filling pipeline (not all stations busy)
- After 1.5 hours: Steady state (all stations busy)

Steady State Analysis (ignoring fill time):

```
Non-pipelined: 2n hours for n loads (2 hours per load)
Pipelined: 0.5n hours for n loads (0.5 hours per load)

Steady State Speedup = 2n / 0.5n = 4×
```

Theoretical Maximum Speedup:

- Equals number of stages
- 4 stages → 4× speedup maximum
- 8 stages → 8× speedup maximum (if achievable)

12.3.4 2.4 Key Performance Terms

Latency:

- Time to complete one individual job
- Customer A: Still 2 hours in both cases
- Per-instruction time unchanged

Throughput:

- How often one job completes
- Non-pipelined: 1 job every 2 hours
- Pipelined: 1 job every 30 minutes (steady state)
- Throughput is the relevant metric for pipelines

Observation:

- Pipelining doesn’t reduce individual job latency
- Pipelining dramatically improves throughput
- Overall system performance greatly enhanced

Analogy Summary:

- Customers = Instructions
- Stages = Pipeline stages
- Time saved = Performance improvement
- Overlapping execution = Instruction-level parallelism

12.4 3. MIPS Five-Stage Pipeline

12.4.1 3.1 Pipeline Stage Definitions

12.4.1.1 Stage 1: Instruction Fetch (IF)

- Use current Program Counter (PC)
- PC points to next instruction to execute
- Access instruction memory
- Fetch instruction word
- Duration: One clock cycle

12.4.1.2 Stage 2: Instruction Decode / Register Read (ID)

- Decode opcode field
- Determine instruction category
- Identify remaining bit organization
- Extract register addresses
- Read register file
- Both operations in one stage (workload balancing)

12.4.1.3 Stage 3: Execution (EX)

- Arithmetic/Logic instructions: ALU computes result
- Memory instructions: ALU computes address (base + offset)
- Branch instructions: ALU performs comparison
- One clock cycle

12.4.1.4 Stage 4: Memory Access (MEM)

- Load instructions: Read from data memory
- Store instructions: Write to data memory
- Other instructions: Skip this stage
- One clock cycle

12.4.1.5 Stage 5: Write Back (WB)

- Write result to register file
- Source: ALU result (arithmetic) OR memory data (load)
- Multiplexer selects appropriate source

- One clock cycle

Workload Distribution Goal:

- Evenly distribute work across stages
- Minimize clock cycle time
- Maximize hardware utilization

12.4.2 3.2 Stage Timing Example

Assumed Component Delays:

Component	Delay (picoseconds)
Instruction Fetch	200 ps
Register Read/Write	100 ps
ALU Operation	200 ps
Data Memory Access	200 ps
Sign Extension	negligible
Multiplexers	negligible

Single-Cycle Instruction Times:

Instruction Type	Stages Used	Total Time
Load Word (LW)	IF+ID+EX+MEM+WB	800 ps
Store Word (SW)	IF+ID+EX+MEM	700 ps
R-type (ADD, etc.)	IF+ID+EX+WB	600 ps
Branch (BEQ)	IF+ID+EX	500 ps

Load Word Critical Path: 800 ps determines clock period

12.4.3 3.3 Pipeline Implementation Details

Clock Cycle Determination:

- Must accommodate longest stage
- Longest stage: 200 ps (IF, ALU, MEM)
- Clock cycle: 200 ps
- Some stages underutilize cycle (register read/write: 100 ps)

Register Read/Write Timing:

- **CRITICAL:** Register write first half, read second half of same clock cycle
- Enables same-cycle read-after-write
- Prevents data hazards in some cases

Stage Alignment to Clock Cycles:

Stage	Work	Time	Cycle Time
IF	Instruction Memory read	200 ps	200 ps □
ID	Decode + Register Read	100 ps	200 ps (space left)
EX	ALU operation	200 ps	200 ps □
MEM	Data Memory access	200 ps	200 ps □
WB	Register write	100 ps	200 ps (first half only)

Space in ID Stage:

- Register read: 100 ps
- Decoding: Fits in remaining 100 ps
- Combinational logic for opcode decode
- Total: ~200 ps utilized

Space in WB Stage:

- Register write: 100 ps (first half)
- Second half: Available for next instruction's register read

12.4.4 3.4 Load Word Pipeline Example

Instruction Stream: All Load Word instructions

```
LW $1, 0($10)
LW $2, 4($10)
LW $3, 8($10)
LW $4, 12($10)
...
```

Pipeline Timing Diagram:

```
Time (ps):  0-200  200-400  400-600  600-800  800-1000  1000-1200
LW $1:      IF     ID       EX       MEM      WB
LW $2:             IF       ID       EX       MEM       WB
LW $3:                      IF       ID       EX        MEM
LW $4:                               IF       ID        EX
```

Single-Cycle Comparison:

- Non-pipelined: 800 ps per instruction
- Pipelined: 200 ps per instruction (after pipeline fills)

Throughput Improvement:

```
Non-pipelined: 1 instruction every 800 ps
Pipelined: 1 instruction every 200 ps

Speedup = 800 / 200 = 4×
```

Absolute Time per Instruction:

- Still ~800 ps (slightly more with alignment overhead)

- Latency unchanged or slightly worse
- Throughput dramatically improved

12.4.5 3.5 Ideal vs Actual Speedup

Ideal Case (balanced stages):

```
Time between instructions (pipelined) = Time per instruction (non-pipelined) / Number
```

```
Maximum Speedup = Number of Stages
```

Actual Implementation:

- Stages not perfectly balanced
- Register operations faster than memory/ALU
- Speedup < Number of stages
- Example: 5 stages → 4× speedup (not 5×)

Reasons for Less Than Ideal:

1. Unbalanced stage delays
2. Pipeline fill time overhead
3. Hazards (discussed later)
4. Added synchronization logic

12.5 4. MIPS ISA Design for Pipelining

12.5.1 4.1 Fixed Instruction Length

MIPS Characteristic:

- All instructions exactly 32 bits
- Same as ARM (also designed for pipelining)

Benefits for Pipelining:

- Simple instruction fetch (always 32 bits)
- Simple decode (fixed format)
- Bus width fully utilized every time
- No variable-width handling logic

Alternative (Variable-Length):

- Complicates fetch stage
- Requires width detection logic
- May need multiple fetch cycles
- Added combinational logic delays

12.5.2 4.2 Fewer Regular Instruction Formats

MIPS Formats:

- Only 3-4 instruction formats (R, I, J types)
- Small opcode field (6 bits)
- Regular register field positions

Benefits:

- Fast decoding (small opcode → simple logic)
- Fits decode + register read in one stage
- Minimal combinational delay

Register Field Consistency:

- RS (bits 21-25): First source register
- RT (bits 16-20): Second source / destination
- RD (bits 11-15): Destination (R-type)
- Same positions across formats

Decoding Simplification:

- Small opcode → simple decode logic
- Regular formats → minimal mux complexity
- Fast enough for single clock cycle

12.5.3 4.3 Separate ALU Operation Field

Function Field (funct):

- Bits 0-5: Specifies ALU operation for R-type
- Separate from opcode
- Only examined for R-type (opcode = 0)

Design Rationale:

- ALU operation determined in EX stage
- Opcode used in ID stage
- Temporal separation matches pipeline stages

Benefit:

- funct field processed later (EX stage)
- Opcode processed early (ID stage)
- Separating them simplifies each stage
- Avoids large opcode (keeps decode simple)

Alternative Design:

- Include funct in opcode
- Larger opcode field needed
- More complex decode logic
- Slower ID stage
- Worse pipeline balance

12.5.4 4.4 Load/Store Addressing Mode

MIPS Addressing:

- Base register + offset
- Address = $rs + immediate
- Calculation: Simple addition

Pipeline Fit:

- Address calculation: EX stage (ALU)
- Memory access: MEM stage (next cycle)
- Clean separation into two stages

Design Philosophy:

- ISA designed with pipeline in mind
- Not optimized for single-cycle
- Performance through pipelining

MIPS vs Other ISAs:

- MIPS: Designed for pipelining from start
- x86: Complex instructions, harder to pipeline
- ARM: Similar philosophy to MIPS
- RISC principles support pipelining

12.6 5. Instruction-Level Parallelism (ILP)

12.6.1 5.1 Parallel Execution Concept

Definition:

- Multiple instructions executing simultaneously
- Each at different pipeline stage
- Overlapping execution

Example at Steady State:

```
Time Window: 800-1000 ps

Instruction A: WB stage (writing result)
Instruction B: MEM stage (memory access)
Instruction C: EX stage (ALU operation)
Instruction D: ID stage (decode, register read)
Instruction E: IF stage (fetch)

Five instructions active simultaneously!
```

Instruction-Level Parallelism (ILP):

- Lowest granularity of parallelism

- Inside CPU microarchitecture
- Transparent to software
- Hardware manages parallelism

12.6.2 5.2 Levels of Parallelism

Instruction-Level Parallelism:

- Multiple instructions in pipeline
- Same program/thread
- Within CPU core
- Microsecond/nanosecond scale

Thread-Level Parallelism:

- Multiple threads on same core
- Context switching
- OS-managed
- Millisecond scale

Program-Level Parallelism:

- Multiple programs/processes
- Multi-core execution
- OS-scheduled
- Varied time scales

Application-Level Parallelism:

- Distributed computing
- Multiple machines
- Network communication
- Seconds to minutes scale

ILP Focus:

- Fine-grained parallelism
- Hardware implementation
- Transparent to programmer (mostly)
- Foundation for all higher levels

12.7 6. Pipeline Hazards: Structural Hazards

12.7.1 6.1 Hazard Definition

General Concept:

- Situations preventing next instruction from starting
- Violates basic pipelining goal
- Reduces throughput
- Requires pipeline stalls (bubbles)

Three Categories:

1. **Structural Hazards:** Hardware resource busy
2. **Data Hazards:** Need data from previous instruction
3. **Control Hazards:** Decision depends on previous result

12.7.2 6.2 Structural Hazard: Single Memory

Scenario:

- Single memory device for both instructions and data
- No separate instruction/data memory
- Same device holds program and data

Conflict Example:

```
Time:    0-200   200-400  400-600  600-800
LW $1:   IF      ID       EX       MEM
LW $2:           IF       ID       EX
LW $3:                    IF       ID
LW $4:                             IF  ← CONFLICT!

At 600-800 ps:
- LW $1 needs data memory (MEM stage)
- LW $4 needs instruction memory (IF stage)
- Same physical memory device!
- Cannot access simultaneously
```

Problem:

- Memory can only service one request per cycle
- Instruction fetch AND data access conflict
- Hardware resource (memory) busy

12.7.3 6.3 Pipeline Stall (Bubble)

Solution: Insert Bubble

```
Time:    0-200   200-400  400-600  600-800  800-1000  1000-1200
LW $1:   IF      ID       EX       MEM      WB
LW $2:           IF       ID       EX       [BUBBLE]  MEM
LW $3:                    IF       ID       EX        [BUBBLE]
LW $4:                             IF       [BUBBLE]  ID
```

Bubble Characteristics:

- No instruction in that pipeline stage
- Like air bubble in water pipeline
- Hardware idle for that stage
- Wastes one clock cycle
- Propagates through pipeline stages

Impact:

- One instruction delayed
- Subsequent instructions delayed
- Throughput reduced
- Performance loss

Bubble Analogy:

- Water pipeline: Continuous flow
- Air bubble: Break in flow
- Takes time to propagate through
- Reduces effective flow rate

12.7.4 6.4 Solutions to Structural Hazards

Solution 1: Separate Memories

- Instruction memory separate from data memory
- Harvard architecture
- Simultaneous access possible
- No structural hazard

Solution 2: Separate Caches

- Single main memory
- Separate instruction cache (I-cache)
- Separate data cache (D-cache)
- Cache: Fast buffer between CPU and memory
- Caches can be accessed simultaneously
- Details in future lectures (memory hierarchy)

Design Recommendation:

- Modern processors use separate caches
- Necessary for high-performance pipelining
- Small area overhead for large performance gain

12.8 7. Data Hazards

12.8.1 7.1 Data Hazard Definition

Concept:

- Subsequent instruction needs data from previous instruction
- Data not yet available (still being computed/written)
- Reading too early → wrong value
- Writing too early → data corruption

Example:

```
ADD $s0, $t0, $t1       # $s0 = $t0 + $t1
SUB $t2, $s0, $t3       # $t2 = $s0 - $t3 (uses $s0 from ADD)
```

Problem:

- ADD computes $s0 value in EX stage
- SUB needs $s0 value in ID stage (register read)
- Timing mismatch

12.8.2 7.2 Data Hazard Example Analysis

Instruction Sequence:

```
ADD $s0, $t0, $t1
SUB $t2, $s0, $t3
```

Pipeline Without Stalls:

```
Time:    0-200   200-400  400-600  600-800  800-1000
ADD:     IF      ID       EX       MEM      WB
SUB:             IF       ID       EX       MEM
                          ↑
                    Reads $s0 here (old value!)

ADD writes $s0 here ↓
```

Problem Timeline:

- 200-400: ADD reads $t0, $t1; SUB fetched
- 400-600: ADD computes in ALU; SUB reads registers (gets OLD $s0!)
- 600-800: ADD result available but not in register yet
- 800-1000: ADD writes $s0 to register (first half of cycle)

SUB reads $s0 at 400-600, but correct value not available until 800-1000!

12.8.3 7.3 Solution 1: Pipeline Stalls

Insert Two Bubbles:

```
Time:    0-200   200-400  400-600  600-800  800-1000  1000-1200  1200-1400
ADD:     IF      ID       EX       MEM      WB
[BUBBLE]                  IF       [BUBBLE] [BUBBLE]
[BUBBLE]                           IF       [BUBBLE]
SUB:                                         IF        ID
```

Result:

- SUB fetched at 1000-1200
- SUB reads registers at 1200-1400 (second half at 1200)
- ADD writes $s0 at 800-1000 (first half at 800)
- Sufficient time gap: Correct value available

Cost:

- Two clock cycles wasted
- Throughput reduced
- Performance penalty

Critical Timing:

- Register write: First half of WB cycle
- Register read: Second half of ID cycle
- Enables back-to-back reading of just-written value

12.8.4 7.4 Solution 2: Forwarding (Bypassing)

Key Observation:

- ADD result available after EX stage (400-600)
- Result at ALU output
- Not yet written to register file
- But SUB's ALU operation at 600-800
- Can forward ALU output directly to ALU input!

Forwarding Logic:

```
Time:    0-200   200-400  400-600  600-800  800-1000
ADD:     IF      ID       EX       MEM      WB
SUB:             IF       ID       EX       MEM
                          ↑        ↑
                    Read regs   Use forwarded value!
```

Implementation:

- Multiplexer at ALU input
- Selects between:
  - Register file output (normal path)
  - Forwarded value from previous ALU output
- Control logic detects dependency
- Routes correct value

Benefit:

- Eliminates two stalls
- No performance penalty
- Requires additional hardware:
  - Forwarding multiplexers
  - Forwarding detection logic
  - Forwarding paths (wires)
  - Pipeline registers to hold values

Complexity:

- Careful synchronization required
- Detect true dependencies
- Avoid false positives
- Additional control signals

Result:

- SUB can execute immediately after ADD
- No stalls needed

- Correct value forwarded

12.8.5 7.5 Load-Use Data Hazard

Special Case:

```
LW  $s0, 0($t0)        # Load from memory into $s0
SUB $t2, $s0, $t3      # Use $s0 immediately
```

Problem:

- Load result available after MEM stage (data from memory)
- SUB needs value in EX stage
- Even forwarding can't help!

Timeline:

```
Time:    0-200   200-400  400-600  600-800  800-1000
LW:      IF      ID       EX       MEM      WB
SUB:             IF       ID       EX       MEM
                          ↑        ↑
                    Need value   Value first available here!
```

LW result available at 600-800, but SUB's EX at 600-800 (simultaneous!)

Unavoidable Stall:

```
Time:    0-200   200-400  400-600  600-800  800-1000  1000-1200
LW:      IF      ID       EX       MEM      WB
[BUBBLE]                  IF       [BUBBLE] ID
SUB:                                         IF        ID
```

One stall bubble required:

- Cannot be eliminated by forwarding
- Can forward from MEM to EX (saves one stall vs two)
- But at least one stall unavoidable

12.8.6 7.6 Compiler Solution: Code Reordering

C Code Example:

```
a = b + e;
c = b + f;
```

Naive Assembly (Load-Use Hazards):

```
LW   $t1, 0($t0)    # Load b into $t1
LW   $t2, 4($t0)    # Load e into $t2
ADD  $t3, $t1, $t2  # a = b + e ← HAZARD: uses $t2 immediately after LW
SW   $t3, 8($t0)    # Store a

LW   $t4, 12($t0)   # Load f into $t4
```

```
ADD  $t5, $t1, $t4  # c = b + f ← HAZARD: uses $t4 immediately after LW
SW   $t5, 16($t0)   # Store c
```

Total: 7 instructions + 2 stalls = 9 clock cycles

Optimized Assembly (Reordered):

```
LW   $t1, 0($t0)    # Load b into $t1
LW   $t2, 4($t0)    # Load e into $t2
LW   $t4, 12($t0)   # Load f into $t4 ← Moved here!
ADD  $t3, $t1, $t2  # a = b + e ← No hazard! $t2 available
SW   $t3, 8($t0)    # Store a ← Moved here!
ADD  $t5, $t1, $t4  # c = b + f ← No hazard! $t4 available
SW   $t5, 16($t0)   # Store c
```

Total: 7 instructions + 0 stalls = 7 clock cycles

Technique:

- Load f earlier (between loading b and e)
- Fills stall slot with useful work
- Store a before second ADD (fills another gap)
- No bubbles needed

Savings: 2 clock cycles (22% improvement)

Compiler Responsibility:

- Analyze dependencies
- Reorder instructions safely
- Fill stall slots with independent instructions
- Maintain program semantics

Programmer Awareness:

- Understand pipeline behavior
- Write code amenable to reordering
- Separate dependent instructions when possible
- Help compiler optimize

12.9 8. Control Hazards

12.9.1 8.1 Control Hazard Definition

Concept:

- Branch/Jump outcome determines next instruction
- Decision depends on previous computation
- Can't fetch next instruction until decision made
- Pipeline must wait

Example:

```
BEQ $1, $2, target     # Branch if $1 == $2
ADD $3, $4, $5         # Next sequential instruction
...
target: SUB $6, $7, $8 # Branch target
```

Which instruction to fetch after BEQ?

- ADD if branch NOT taken
- SUB if branch IS taken
- Decision requires comparison: $1 vs $2

12.9.2 8.2 Branch Execution in Pipeline

Branch Instruction:

```
BEQ $1, $2, 40         # Branch 40 instructions ahead if equal
```

Pipeline Stages:

1. IF: Fetch BEQ instruction
2. ID: Read $1, $2 from register file
3. EX: ALU compares (subtract $2 from $1, check zero flag)
4. Result available after EX stage

Problem:

- Next instruction fetch at cycle 2 (IF for next instruction)
- Branch outcome known at cycle 3 (after EX)
- Must guess which instruction to fetch!

Without Optimization:

```
Time:    0-200   200-400  400-600  600-800
BEQ:     IF      ID       EX       MEM
???:             IF       ???
```

Two bubbles required if wait for outcome

12.9.3 8.3 Solution 1: Early Branch Resolution

Add Hardware in ID Stage:

- Small adder for comparison
- Compute branch condition early (ID instead of EX)
- Subtract $1 - $2 in ID stage
- Parallel to register read

Modified Pipeline:

```
Time:    0-200   200-400  400-600
BEQ:     IF      ID       EX
                 ↑
          Decision here!
Next:            IF
```

Benefit:

- Decision after ID (one cycle earlier)
- Only one bubble needed (vs two)
- Better performance

Cost:

- Additional adder hardware
- Extra combinational logic in ID stage
- More complex ID stage

Limitation:

- Still one unavoidable stall
- Can't know outcome in same cycle as fetch

12.9.4 8.4 Solution 2: Branch Prediction

Static Branch Prediction:

- Guess branch outcome
- Fetch based on guess
- If correct: No penalty
- If wrong: Discard fetched instruction, fetch correct one

Strategy: Predict Not Taken

- Assume branch will NOT be taken
- Always fetch PC + 4 (sequential instruction)
- Proceed normally if correct
- Stall and correct if wrong

Example (Prediction Correct):

```
ADD  $3, $4, $5
BEQ  $1, $2, 14        # Actually NOT taken
LW   $8, 0($9)         # Fetch this (prediction: not taken)
```

Timeline:

```
Time:    0-200   200-400  400-600  600-800
ADD:     IF      ID        EX       MEM
BEQ:             IF      ID       EX
LW:                      IF       ID
                         ↑ Fetched based on prediction
```

At 400-600 (after BEQ's ID):

- Determine branch NOT taken
- Prediction correct!
- LW continues normally
- No stall!

Example (Prediction Incorrect):

```
ADD  $3, $4, $5
BEQ  $1, $2, 14        # Actually IS taken
LW   $8, 0($9)         # Fetched (but shouldn't execute)
...
target: SUB $6, $7, $8 # Should execute this instead
```

Timeline:

```
Time:    0-200   200-400  400-600  600-800
ADD:     IF      ID        EX       MEM
BEQ:             IF       ID       EX
LW:                       IF       [DISCARD]
SUB:                               IF
```

At 400-600 (after BEQ's ID):

- Determine branch IS taken
- Prediction wrong!
- Discard LW (clear pipeline stage)
- Fetch SUB from branch target
- One bubble inserted

Result Analysis:

- Correct prediction: Save one cycle
- Incorrect prediction: Same as no prediction (one stall)
- Net benefit if prediction often correct
- No additional penalty for wrong guess

12.9.5 8.5 Static Branch Prediction Strategies

Simple Static: Always Predict Not Taken

- Fixed prediction
- Ignore branch type
- Ignore branch history
- Simple hardware

Program Behavior-Based Static:

- Analyze typical branch patterns
- Make predictions based on code structure

Backward Branches:

- Usually taken
- Example: Loops

```
loop:
    ...
    BEQ $t0, $zero, loop   # Backward branch
```

- Loop iterations: Branch taken many times
- Loop exit: Branch not taken once

- Prediction: Taken → Correct most of time

Forward Branches:

- Usually not taken
- Example: If statements

```
    BEQ $t0, $zero, skip
    ...                      # True case
skip:
    ...                      # After if
```

- True case: Branch not taken
- False case: Branch taken
- Prediction depends on code style

Strategy: Backward Taken, Forward Not Taken

- 90%+ accuracy possible
- Based on empirical program analysis
- Requires code analysis

12.9.6 8.6 Dynamic Branch Prediction

Concept:

- Hardware learns branch behavior
- Predicts based on history
- Adapts to current code execution
- Not fixed prediction

Branch History Table:

- Hardware table storing recent branch outcomes
- Indexed by branch instruction address
- Each entry: Branch taken or not taken recently
- Predicts based on recent behavior

Simple 1-Bit Predictor:

- One bit per branch: Last outcome
- Predict same as last time
- Updates after each execution

Example:

```
Loop iteration 1: Taken → Predict taken next
Loop iteration 2: Taken → Predict taken next
...
Loop iteration 100: Taken → Predict taken next
Loop exit: Not taken → Predict not taken next (wrong for next loop!)
```

Problem: Wrong twice per loop (entry and exit)

2-Bit Saturating Counter:

- Two bits per branch: State machine
- Four states:
  - 00: Strongly not taken
  - 01: Weakly not taken
  - 10: Weakly taken
  - 11: Strongly taken
- Change prediction after two consecutive wrong predictions
- More stable

Advanced Predictors:

- Correlating predictors (look at multiple branches)
- Two-level adaptive predictors
- Tournament predictors (combine multiple algorithms)
- Very high accuracy (>95%)

Hardware Cost:

- Branch history table (memory)
- Prediction logic (comparators, counters)
- Update logic
- Worthwhile for performance gain

12.10 9. Summary and Key Concepts

12.10.1 9.1 Pipelining Benefits

Performance Improvement:

- Throughput increased by number of stages
- 5-stage pipeline → 4-5× speedup
- Latency unchanged or slightly worse
- Overlapping execution key

Hardware Utilization:

- All stages active in steady state
- Parallel processing
- Maximum efficiency

12.10.2 9.2 Pipeline Challenges

Hazards:

1. **Structural:** Hardware resource conflicts
2. **Data:** Instruction dependencies
3. **Control:** Branch/jump decisions

Solutions:

- Structural: Separate memories/caches

- Data: Forwarding, stalls, code reordering
- Control: Early resolution, branch prediction

12.10.3 9.3 MIPS Design Philosophy

ISA Designed for Pipelining:

- Fixed 32-bit instruction length
- Regular instruction formats
- Separate funct field
- Simple addressing modes
- Balanced pipeline stages

Performance Through Hardware:

- Pipelining fundamental to MIPS
- Not optimized for single-cycle
- Hardware complexity for software simplicity

12.10.4 9.4 Key Takeaways

1. Pipelining improves throughput, not latency
2. Steady state determines peak performance
3. Pipeline fill time overhead for small programs
4. Hazards reduce pipelining efficiency
5. Forwarding eliminates many data hazards
6. Load-use hazard always requires one stall
7. Branch prediction crucial for control flow
8. Compiler optimization reduces stalls
9. ISA design significantly impacts pipeline efficiency
10. ILP fundamental to modern processor performance

12.11 10. Important Formulas and Metrics

12.11.1 Speedup Calculation

```
Speedup = Non-pipelined Time / Pipelined Time

Ideal Speedup = Number of Pipeline Stages

Actual Speedup = Number of Stages / (1 + Hazard Impact)
```

12.11.2 Throughput

```
Throughput = 1 instruction / Clock Period

Throughput Improvement = Clock Period (non-pipelined) / Clock Period (pipelined)
```

12.11.3 Pipeline Performance

```
Time = (Number of Instructions + Stages - 1) × Clock Period

CPI (Cycles Per Instruction) = 1 + Stall Cycles per Instruction

Effective CPI = 1 + (Structural Stalls + Data Stalls + Control Stalls)
```

12.11.4 Branch Prediction Accuracy

```
Accuracy = Correct Predictions / Total Branches

Stall Reduction = Accuracy × Cycles Saved per Correct Prediction
```

12.12 Key Takeaways

1. **Pipelining improves throughput, not latency**—individual instructions take same or longer time, but more instructions complete per unit time.
2. **Five-stage MIPS pipeline**: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), Write-Back (WB).
3. **Ideal speedup equals number of stages**—five-stage pipeline theoretically achieves 5× speedup over single-cycle design.
4. **Assembly line analogy clarifies concept**—like manufacturing, each stage works on different item simultaneously for maximum efficiency.
5. **Pipeline registers store intermediate results** between stages, enabling independent operation and preventing data corruption.
6. **Three hazard types threaten pipeline efficiency**: Structural (resource conflicts), Data (register dependencies), Control (branch/jump delays).
7. **Structural hazards resolved by hardware duplication**—separate instruction and data caches eliminate memory access conflicts.
8. **Data hazards occur when instructions depend on previous results**—forwarding (bypassing) allows ALU results to skip write-back stage.
9. **Forwarding paths connect pipeline stages directly**, enabling result use before register file write completes.
10. **Load-use hazard requires one-cycle stall**—memory data unavailable in time for immediate ALU use even with forwarding.
11. **Compiler code reordering can eliminate some stalls**—moving independent instructions into load delay slots maintains pipeline flow.
12. **Control hazards arise from branch/jump instructions**—don't know next PC until branch resolves in third cycle.

13. **Branch delay of 3 cycles** in basic pipeline—fetch/decode/execute complete before decision known, wasting 3 instruction slots.
14. **Early branch resolution reduces penalty**—dedicated comparison hardware in ID stage cuts delay to 1 cycle.
15. **Static branch prediction** assumes direction (e.g., always not-taken)—simple but limited effectiveness.
16. **Dynamic branch prediction** learns patterns from history—branch target buffer with 2-bit saturating counters achieves >90% accuracy.
17. **Two-bit counters prevent single misprediction disruption**—requires two wrong predictions to change direction, handling loop patterns well.
18. **Pipeline performance** = 1 CPI + Structural Stalls + Data Stalls + Control Stalls—minimizing hazards approaches ideal throughput.
19. **Modern processors use sophisticated prediction**—multi-level predictors, pattern history tables, and return address stacks minimize control hazards.
20. **Pipeline complexity trades off with performance**—deeper pipelines increase throughput but amplify hazard penalties and design difficulty.

12.13 Summary

Pipelining revolutionizes processor performance by applying manufacturing assembly-line principles to instruction execution, allowing multiple instructions to occupy different pipeline stages simultaneously. The five-stage MIPS pipeline (IF, ID, EX, MEM, WB) theoretically achieves 5× speedup by keeping all hardware components busy every cycle, transforming the inefficient single-cycle design where most hardware sat idle most of the time. However, three hazard types threaten this ideal performance: structural hazards from resource conflicts (solved by hardware duplication like separate instruction and data caches), data hazards from register dependencies (addressed by forwarding paths that bypass results directly between stages, though load-use cases still require one-cycle stalls), and control hazards from branches that don't resolve until the third cycle (mitigated by early branch resolution hardware, static prediction strategies, and sophisticated dynamic branch predictors using two-bit saturating counters that achieve over 90% accuracy). The effectiveness of forwarding demonstrates how careful hardware design can eliminate most data hazard stalls, while compiler optimizations like instruction reordering can fill remaining delay slots with useful work. Branch prediction evolution from simple static schemes to complex dynamic predictors with branch target buffers reflects the critical importance of minimizing control hazards in modern high-performance processors. Pipeline registers between stages serve as the crucial mechanism enabling independent stage operation, storing intermediate results and control signals while preventing data corruption across instruction overlaps. While pipelining introduces significant design complexity compared to single-cycle implementations, the dramatic performance improvements—approaching 5× speedup in practice—justify this added sophistication, making pipelining universal in modern processor architectures from embedded systems to supercomputers. Understanding these hazards and their solutions provides essential foundation for comprehending real-world processor implementations and the tradeoffs between pipeline depth, clock frequency, and hazard penalties that define contemporary computer architecture.

Chapter 13

Lecture 13: Detailed MIPS Pipeline Operation and Pipeline Registers

13.1 Introduction

This lecture provides comprehensive, cycle-by-cycle analysis of MIPS five-stage pipeline operation, examining how instructions flow through pipeline stages with detailed attention to the pipeline registers that store intermediate results between stages. We explore the critical role of these registers in enabling independent stage operation, trace complete execution sequences for load and store instructions, analyze timing constraints and delay contributions, and work through practical exercises calculating clock frequencies and optimizing pipeline performance. This detailed examination reveals the hardware mechanisms that transform the conceptual pipeline model into functioning silicon.

13.2 1. Lecture Introduction and Recap

13.2.1 1.1 Previous Topics Review

Pipelining Concept:

- Instruction-level parallelism exploitation
- Five-stage MIPS pipeline: IF, ID, EX, MEM, WB
- Staggered instruction execution
- All hardware utilized simultaneously

Performance Metric:

- Throughput improved (not latency)
- Instructions/unit time increased
- Individual instruction latency same or worse
- Overall system performance dramatically better

Hazards Covered:

1. **Structural:** Hardware resource conflicts
2. **Data:** Register/memory dependencies
3. **Control:** Branch/jump decision delays

Solutions Discussed:

- Structural: Separate I-cache and D-cache
- Data: Forwarding, code reordering
- Control: Early branch resolution, prediction

13.2.2 1.2 Today's Focus

Detailed Pipeline Analysis:

- Cycle-by-cycle operation walkthrough
- Pipeline register requirements
- Timing and delay analysis
- Load/Store instruction examples
- Common implementation errors
- Practical exercises

13.3 2. Five-Stage MIPS Pipeline Review

13.3.1 2.1 Stage 1: Instruction Fetch (IF)

Operations:

- PC value determines instruction address
- Access instruction memory
- Fetch 32-bit instruction word
- Calculate PC + 4 for next sequential instruction

Hardware Elements:

- Program Counter register
- Instruction Memory
- PC + 4 Adder

Key Point:

- Both operations (memory read, PC+4 calculation) occur in parallel

13.3.2 2.2 Stage 2: Instruction Decode / Register Read (ID)

Operations:

- Decode opcode (6 bits)
- Determine instruction type
- Identify register fields
- Read register file (RS, RT)

- Sign-extend immediate value (16→32 bits)
- Generate control signals

Hardware Elements:

- Instruction decoder (combinational logic)
- Register file (read ports)
- Sign extension unit
- Control unit

Workload Balancing:

- Decode + register read fit in one cycle
- Even distribution of work
- Register read dominates timing

Control Signal Generation:

- 9-10 control signal bits generated
- Based on opcode
- Used by subsequent stages
- Must be preserved through pipeline

13.3.3 2.3 Stage 3: Execution (EX)

Operations:

- ALU performs computation OR address calculation
- Multiplexer selects second operand (register vs immediate)
- Branch: Compare registers, compute target address

Hardware Elements:

- ALU (Arithmetic Logic Unit)
- Input multiplexer (register/immediate selection)
- Branch target adder (parallel to ALU)
- Shift left 2 unit (for branch offset)

Key Characteristics:

- ALU operation dominates timing
- Branch hardware operates in parallel
- Multiple functions depending on instruction type

13.3.4 2.4 Stage 4: Memory Access (MEM)

Operations:

- Load: Read from data memory
- Store: Write to data memory
- Other instructions: Skip (no memory access)
- Branch: PC update decision

Hardware Elements:

- Data Memory
- PC source multiplexer (for branches)

Timing Consideration:

- Memory access slowest operation
- Dominates stage timing
- Critical path component

13.3.5 2.5 Stage 5: Write Back (WB)

Operations:

- Select data source (ALU result OR memory data)
- Write to destination register
- Load: Memory data → register
- Arithmetic: ALU result → register

Hardware Elements:

- MemtoReg multiplexer
- Register file (write port)

Minimal Hardware:

- Mostly multiplexer visible
- Register file shared with ID stage
- Shortest stage conceptually

13.4 3. Pipeline Registers: Necessity and Function

13.4.1 3.1 Problem Without Pipeline Registers

Scenario:

- Multiple instructions in different stages
- All sharing same hardware components
- Data from different instructions □□

Example Issues:

1. Register file: ID stage reads while WB stage writes
2. Control signals: Generated in ID, needed in later stages
3. Data values: Computed in EX, needed in MEM
4. Overwriting: New instruction data overwrites previous instruction data

Result Without Pipeline Registers:

- Data hazards everywhere
- Control hazards from signal conflicts
- Structural hazards from resource contention
- Pipeline cannot function correctly

13.4.2 3.2 Pipeline Register Purpose

Key Function:

- Store information from previous stage
- Make data available to next stage
- Synchronize operations across clock cycles
- Prevent interference between instructions

Placement:

- One between each pair of consecutive stages
- **IF/ID:** Between instruction fetch and decode
- **ID/EX:** Between decode and execution
- **EX/MEM:** Between execution and memory
- **MEM/WB:** Between memory and write back

Exception:

- No register between WB and IF
- PC register serves this purpose
- Register file contains storage
- No additional register needed

13.4.3 3.3 Pipeline Register Contents

IF/ID Pipeline Register:

- 32-bit instruction word
- 32-bit PC+4 value
- **Total:** 64 bits

ID/EX Pipeline Register:

- Two 32-bit register values (from register file)
- 32-bit sign-extended immediate
- 32-bit PC+4 value (for branches)
- 5-bit write register address
- 9-10 control signal bits
- **Total:** ~140+ bits (largest pipeline register)

EX/MEM Pipeline Register:

- 32-bit ALU result
- 32-bit register value (for stores)
- 32-bit branch target address
- 1-bit ALU zero flag
- 5-bit write register address
- Control signals for MEM/WB stages
- **Total:** ~105+ bits

MEM/WB Pipeline Register:

- 32-bit memory read data

- 32-bit ALU result
- 5-bit write register address
- Control signals for WB stage
- **Total:** ~75+ bits

13.4.4 3.4 Timing: Writing and Reading Pipeline Registers

At Rising Clock Edge:

1. Pipeline register write begins
2. Small hold time delay (~10-30 ps)
3. Data captured and stored
4. Writing delay consumed

After Writing: 5. Reading delay begins 6. Data propagates to output (~10-30 ps) 7. Outputs stabilize at new values 8. Next stage begins operations

Combined Overhead:

- Write delay + read delay = ~20-60 ps
- Occurs at start of every stage
- Reduces time available for actual computation
- Pipelining overhead cost

Critical Observation:

- These delays don't exist in single-cycle
- Pipelining adds latency overhead
- But throughput gain outweighs latency cost

13.5 4. Load Word Instruction: Detailed Cycle-by-Cycle Analysis

13.5.1 4.1 Load Word Instruction Format

Encoding:

```
LW $rt, offset($rs)

Opcode: 100011 (bits 26-31)
RS:     Base register (bits 21-25)
RT:     Destination register (bits 16-20)
Offset: 16-bit immediate (bits 0-15)
```

Operation: $rt = Memory$[rs + offset]

Example: LW $8, 32($9)

- Base address in $9
- Add offset 32

- Load from memory into $8

13.5.2 4.2 Clock Cycle 1: Instruction Fetch (IF)

Start of Cycle:

- New PC value available (from previous cycle)
- PC write delay: ~10-30 ps
- PC read delay: ~10-30 ps

Operations:

1. Update PC register (rising edge)
2. Read PC value (small delay)
3. Access instruction memory with PC address
4. Instruction memory read delay: ~200 ps (dominant)
5. Compute PC + 4 in parallel: ~70 ps

End of Cycle:

- 32-bit LW instruction available
- PC + 4 value available
- Both ready to write to IF/ID register

Hardware Shading Convention:

- Right side shaded: Device READ
- Left side shaded: Device WRITTEN
- Example: Instruction Memory right-side shaded (read)
- IF/ID register left-side shaded (written to)

Total Stage Time: ~200+ ps (instruction memory dominant)

13.5.3 4.3 Clock Cycle 2: Instruction Decode / Register Read (ID)

Start of Cycle (Rising Edge):

1. IF/ID register write: ~30 ps
2. IF/ID register read: ~30 ps
3. Combined delay: ~60 ps

After Pipeline Register: 4. Instruction word available 5. Extract fields:

- Opcode: bits 26-31 → Control Unit
- RS (bits 21-25) → Register file address 1
- RT (bits 16-20) → Register file address 2 AND write address
- Offset (bits 0-15) → Sign extender

Parallel Operations:

- Control Unit: Decode opcode → Generate control signals (~50 ps)
- Register File: Read RS ($9) and RT ($8 address, value not needed)
  - Read delay: ~90 ps (dominant)
- Sign Extender: Extend 32 to 32 bits (~10 ps, negligible)

End of Cycle:

- Base address value (from $9) available
- RT address read (discarded for LW)
- Sign-extended offset (32) available
- PC + 4 value forwarded
- Control signals generated
- All ready for ID/EX register

Why Read Both Registers:

- Hardware simplicity: Always read both
- Multiplexer decides usage later
- Store would need RT value
- Simpler than conditional reading

Total Stage Time: ~60 + 90 = ~150 ps (register read dominant)

13.5.4 4.4 Clock Cycle 3: Execution (EX)

Start of Cycle:

1. ID/EX register write: ~30 ps
2. ID/EX register read: ~30 ps

ALU Input Preparation: 3. Input A: Base address (from $9) directly from pipeline register 4. Input B: Multiplexer selects immediate OR register

- Control signal ALUSrc = 1 (select immediate)
- Multiplexer delay: ~20 ps
- Sign-extended offset (32) selected

ALU Operation: 5. Add base address + offset 6. ALU delay: ~90 ps (dominant) 7. Result: Memory address = $9 + 32

Parallel Operations (for branches, not used here):

- Shift left 2: Offset × 4 (~10 ps)
- Branch target adder: PC+4 + (offset×4) (~70 ps)
- Zero flag generation

End of Cycle:

- Memory address available at ALU output
- Branch target available (unused)
- Zero flag available (unused)
- RT value forwarded (unused for LW)
- Control signals forwarded
- Write register address (RT) forwarded
- All ready for EX/MEM register

Total Stage Time: ~30 + 30 + 20 + 90 = ~170 ps (ALU dominant)

13.5.5 4.5 Clock Cycle 4: Memory Access (MEM)

Start of Cycle:

1. EX/MEM register write: ~30 ps
2. EX/MEM register read: ~30 ps

Memory Access: 3. ALU result (address) → Data memory address input 4. MemRead control signal = 1 (enable read) 5. MemWrite control signal = 0 (disable write) 6. Data memory read delay: ~250 ps (**DOMINANT** - slowest operation!)

Parallel Operations (unused for LW):

- Zero flag + Branch → PCSrc decision
- Branch target → PC multiplexer

End of Cycle:

- Loaded data available from memory
- ALU result (address) forwarded for R-type instructions
- Write register address (RT) forwarded
- Control signals forwarded
- All ready for MEM/WB register

Critical Path:

- Load Word determines minimum clock period
- Memory access slowest component
- All other instructions wait for this

Total Stage Time: ~30 + 30 + 250 = ~310 ps (memory READ dominant!)

13.5.6 4.6 Clock Cycle 5: Write Back (WB)

Start of Cycle:

1. MEM/WB register write: ~30 ps
2. MEM/WB register read: ~30 ps

Data Selection: 3. MemtoReg multiplexer:

- Control signal MemtoReg = 1 (select memory data)
- Input 0: ALU result (not used for LW)
- Input 1: Memory read data (**SELECTED**)
- Multiplexer delay: ~20 ps

Register Write Preparation: 4. Write data: Memory data from multiplexer 5. Write address: RT ($8) from pipeline register 6. RegWrite control signal = 1 (enable write)

CRITICAL ERROR IN TEXTBOOK DIAGRAM:

- Many diagrams show write address from IF/ID register
- **WRONG!** IF/ID has current instruction (4 cycles later!)
- **Correct:** Write address propagated through ALL pipeline registers
- Must use write address from MEM/WB register

At Rising Edge (End of Cycle / Start of Next): 7. Register $8 written with loaded data 8. Write occurs in first half of cycle 9. Subsequent ID stage can read in second half (same cycle!)

Register File Timing Trick:

- Write: First half of clock cycle
- Read: Second half of clock cycle
- Enables read-after-write in adjacent cycles
- Critical for data forwarding

Total Stage Time: ~30 + 30 + 20 = ~80 ps (shortest stage!)

13.5.7 4.7 Load Word Complete Pipeline Summary

Cycle	Stage	Operations	Dominant Delay	Time
1	IF	Fetch instruction, PC+4	Inst Memory	200ps
2	ID	Decode, read regs, control	Reg Read	150ps
3	EX	ALU: base + offset	ALU	170ps
4	MEM	Read data memory	Memory Read	**310ps** ← **CRITICAL!**
5	WB	Select memory, write register	Multiplexer	80ps

Minimum Clock Period: 310 ps (limited by MEM stage) **Maximum Clock Frequency:** 1 / 310ps ≈ 3.2 GHz

Pipeline Overhead:

- Pipeline register delays: ~(30+30) × 5 stages = 300ps
- Actual useful work: ~(200+90+90+250) = 630ps
- Total latency: ~930ps
- Overhead: ~32% of execution time

Comparison to Single-Cycle:

- Single-cycle latency: ~800 ps (no pipeline register overhead)
- Pipelined latency: ~930 ps (with overhead)
- But pipelined throughput: 5× better (ideally)

13.6 5. Store Word Instruction: Key Differences

13.6.1 5.1 Store Word Instruction Format

Encoding:

```
SW $rt, offset($rs)
```

```
Opcode: 101011 (bits 26-31)
```

```
RS:     Base register (bits 21-25)
RT:     Source data register (bits 16-20)
Offset: 16-bit immediate (bits 0-15)
```

Operation: Memory[$rs + offset] = $rt

Example: SW $8, 32($9)

- Base address in $9
- Add offset 32
- Store $8 value to memory

Key Difference from Load:

- RT is SOURCE (not destination)
- RT value needed for memory write

13.6.2 5.2 Stages IF, ID, EX: Same as Load Word

Instruction Fetch: Identical to LW

Instruction Decode: Identical to LW

- Read both RS and RT
- RT value NOW IMPORTANT (not discarded)
- Sign-extend offset

Execution: Identical to LW

- Compute memory address: base + offset
- ALU operation same

13.6.3 5.3 Memory Access Stage: KEY DIFFERENCE

Start of Cycle:

- EX/MEM register contains:
  - Memory address (from ALU)
  - RT data value (from register file, preserved through pipeline)

Memory Access:

- Address → Data memory address input
- RT value → Data memory write data input
- MemWrite = 1 (**ENABLE** write)
- MemRead = 0 (**DISABLE** read)

Operation:

- Write RT value to computed address
- Memory write delay: ~250 ps

End of Cycle:

- Data written to memory
- Memory read output INVALID (MemRead=0)

- Not used by subsequent stage

Control Signal Critical:

Control Signal	Load	Store
MemRead	1	0
MemWrite	0	1
RegWrite (WB stage)	1	**0** ← **CRITICAL!**

13.6.4 5.4 Write Back Stage: NO OPERATION

Store Word WB Stage:

- NO register write needed
- Store wrote to MEMORY (not register)
- RegWrite = 0 (**DISABLE**)

Why RegWrite MUST Be 0:

- Pipeline registers still contain data
- MemtoReg multiplexer produces output
- If RegWrite = 1: **DISASTER!**
  - Random data written to random register
  - Data corruption
  - Program failure

Hardware Still Operates:

- Multiplexer produces output (garbage)
- Write address present (RT from pipeline)
- Write data present (memory output = invalid, or ALU result)
- But RegWrite = 0 prevents write

Lesson: Control Signals Essential

- Must prevent unwanted operations
- Hardware runs in parallel
- Only control signals prevent corruption

Store Word Pipeline Summary:

Cycle	Stage	Operations	Notes
1	IF	Fetch SW instruction	Same as LW
2	ID	Decode, read RS, RT	RT value USED (not discarded)
3	EX	Compute address	Same as LW
4	MEM	Write RT to memory	**WRITE** instead of read
5	WB	Nothing (bubble)	RegWrite=0, stage idle

13.7 6. Common Pipeline Diagram Errors

13.7.1 6.1 Error 1: Write Register Address Source

Incorrect Diagram Shows:

- Write register address from IF/ID pipeline register
- Connected directly to register file write port

Why This Is Wrong:

- IF/ID contains CURRENT instruction (just fetched)
- Write back for instruction 4 cycles ago
- Wrong register would be written!

Example:

```
Cycle 1: LW $8, 0($10) fetched  (IF)
Cycle 2: LW $9, 4($10) fetched  (IF), LW $8 in ID
Cycle 3: LW $10, 8($10) fetched (IF), LW $8 in EX
Cycle 4: ADD $11, $12, $13 fetched (IF), LW $8 in MEM
Cycle 5: SUB $14, $15, $16 fetched (IF), LW $8 in WB

At Cycle 5:
- IF/ID contains SUB (writes $14)
- WB should write $8 (from LW)
- If using IF/ID: Would write to $14 instead of $8!
- WRONG REGISTER!
```

Correct Implementation:

- Propagate write address through ALL pipeline registers
- ID/EX stores it
- EX/MEM stores it
- MEM/WB stores it
- WB uses address from MEM/WB register

Additional Lines Required:

- 5-bit write address bus through each pipeline register
- Increases pipeline register size
- Essential for correctness

13.7.2 6.2 Error 2: Incorrect Memory Access Indication

Diagram Error from Textbook:

- ADD instruction shown accessing data memory (wrong!)
- LW instruction shown NOT accessing data memory (wrong!)

Correct Resource Usage:

Instruction	IF	ID	EX	MEM	WB
LW	□	□	□	□ Read	□ Write Reg
SW	□	□	□	□ Write	No action
ADD	□	□	□	□ No access	□ Write Reg
BEQ	□	□	□	□ PC update	□ No write

Shading Convention:

- Shaded box: Resource USED
- Unshaded box: Resource NOT USED (idle)

ADD Instruction Correct:

- MEM stage: No memory access, stage mostly idle
- Just forwards ALU result

LW Instruction Correct:

- MEM stage: Read from data memory
- Memory data forwarded to WB

13.7.3 6.3 Error 3: Store Word Memory Read

Another Common Error:

- Store instruction shown with MemRead = 1
- Memory output shown as valid

Why Wrong:

- Store WRITES to memory (MemWrite = 1)
- Should NOT read (MemRead = 0)
- Memory read output undefined/invalid

Correct:

- MemWrite = 1, MemRead = 0
- Memory input: Address and write data
- Memory output: Ignored (invalid)

13.8 7. Multi-Clock-Cycle Pipeline Diagrams

13.8.1 7.1 Single-Clock vs Multi-Clock Diagrams

Single-Clock-Cycle Diagram:

- Shows ONE stage at ONE clock cycle
- Detailed resource usage
- Specific delays visible
- Good for understanding individual stage

Multi-Clock-Cycle Diagram:

- Shows MULTIPLE instructions at MULTIPLE cycles
- Cross-sectional view of pipeline
- Parallel execution visible
- Good for understanding overall flow

13.8.2 7.2 Traditional Multi-Cycle Diagram

Format:

```
          Cycle: 1    2    3    4    5    6    7    8    9
Instr 1:         IF   ID   EX   MEM  WB
Instr 2:              IF   ID   EX   MEM  WB
Instr 3:                   IF   ID   EX   MEM  WB
Instr 4:                        IF   ID   EX   MEM  WB
Instr 5:                             IF   ID   EX   MEM  WB
```

Shows:

- Staggered execution
- Steady state (cycle 5: all stages busy)
- Pipeline fill time (cycles 1-4)
- Pipeline drain time (cycles 7-9)

Does NOT Show:

- Resource usage details
- Hardware components used
- Delays and timing

13.8.3 7.3 Enhanced Multi-Cycle Diagram with Resources

Format:

```
Cycle 1:  Instr 1: [IM][RF][  ][  ][  ]
Cycle 2:  Instr 1: [  ][IM][RF][  ][  ]   Instr 2: [IM][RF][  ][  ][  ]
Cycle 3:  Instr 1: [  ][  ][IM][RF][  ]   Instr 2: [  ][IM][RF][  ][  ]   Instr 3: [I
...

Legend:
IM: Instruction Memory
RF: Register File
ALU: ALU operation
DM: Data Memory
WB: Write Back
```

Shows:

- Which resources used when
- Parallel resource usage
- Resource conflicts (if any)

- Detailed pipeline state

Benefits:

- Visualize structural hazards
- Understand resource contention
- See idle hardware
- Verify correctness

Textbook Error Example:

- ADD instruction marked with DM (wrong!)
- LW instruction NOT marked with DM (wrong!)
- Always verify diagrams carefully

13.9 8. Timing and Clock Frequency Analysis

13.9.1 8.1 Component Delays (Typical Values)

Component	Delay (picoseconds)
Instruction Memory	200
Register File Read	90
Register File Write	90
ALU Operation	90
Data Memory Read	250
Data Memory Write	250
Sign Extension	10 (negligible)
Multiplexer	20
Adder (PC+4, branch)	70
Shift Left 2	10 (wire routing)
Pipeline Register Write	30
Pipeline Register Read	30

Key Observations:

- Memory operations slowest (200-250 ps)
- ALU and register file moderate (90 ps)
- Small combinational logic fast (10-20 ps)
- Pipeline register overhead (60 ps per stage)

13.9.2 8.2 Stage Timing Calculation

Stage 1: Instruction Fetch (IF)

```
Pipeline Register Write:   N/A (PC register)
Pipeline Register Read:    N/A
Instruction Memory:        200 ps
```

```
PC + 4 Adder:              70 ps (parallel)

Total: 200 ps (memory dominant)
```

Stage 2: Instruction Decode (ID)

```
IF/ID Write + Read:        60 ps
Register File Read:        90 ps (dominant)
Control Unit Decode:       50 ps (parallel)
Sign Extension:            10 ps (parallel)

Total: 60 + 90 = 150 ps
```

Stage 3: Execution (EX)

```
ID/EX Write + Read:        60 ps
Multiplexer:               20 ps
ALU Operation:             90 ps
Branch Adder:              70 ps (parallel)
Shift Left 2:              10 ps (parallel)

Total: 60 + 20 + 90 = 170 ps
```

Stage 4: Memory Access (MEM)

```
EX/MEM Write + Read:       60 ps
Data Memory Access:        250 ps (DOMINANT)

Total: 60 + 250 = 310 ps ← CRITICAL PATH!
```

Stage 5: Write Back (WB)

```
MEM/WB Write + Read:       60 ps
MemtoReg Multiplexer:      20 ps
Register File Write:       30 ps (first half of cycle)

Total: 60 + 20 + 30 = 110 ps
```

13.9.3 8.3 Clock Frequency Determination

Minimum Clock Period:

- Determined by SLOWEST stage
- MEM stage: 310 ps
- All stages must use this period

Maximum Clock Frequency:

```
f_max = 1 / T_min
      = 1 / 310 ps
      = 1 / (310 × 10^-12 s)
      = 3.226 GHz
        3.2 GHz
```

Efficiency Analysis:

Stage	Time	Utilization	Wasted Time
IF	200	65%	110 ps
ID	150	48%	160 ps
EX	170	55%	140 ps
MEM	310	100%	0 ps
WB	110	35%	200 ps

Average utilization: ~60% **Wasted time:** ~40% average

13.9.4 8.4 Performance Improvement Strategies

Strategy 1: Pipeline Balancing

- Reduce MEM stage delay (dominant)
- Options:
  - Faster memory technology
  - Separate instruction/data caches
  - Smaller, faster cache
  - Multi-ported memory

Strategy 2: Increase ALU Time

- Question: If ALU shortened by 25%, does it help?
- Answer: Depends on critical path
- If MEM is critical (usual case): NO improvement
- If EX is critical (rare): YES, improves throughput

Strategy 3: Additional Pipeline Stages

- Subdivide long stages (especially MEM)
- Memory access in 2-3 sub-stages
- Shorter clock period possible
- More stages = more overhead
- Diminishing returns beyond certain point

Strategy 4: Cache Memory

- Fast cache between CPU and main memory
- Cache hit: Fast access (~10-20 ps)
- Cache miss: Slow access (~250 ps)
- High hit rate → effective fast memory
- (Covered in next lectures)

Real-World Example:

- Intel Atom processors: ~30 pipeline stages
- Achieved by extreme subdivision
- Very short clock period
- High frequency possible

- But diminishing returns and hazard complexity

13.10 9. Practical Exercises and Solutions

13.10.1 9.1 Exercise: Maximum Clock Frequency Calculation

Given Component Delays:

```
Instruction Memory:      200 ps
Register File (read):    90 ps
Register File (write):   90 ps
ALU:                     90 ps
Data Memory (read):      250 ps
Data Memory (write):     250 ps
Sign Extend:             ~0 ps
Multiplexer:             20 ps
Adder:                   70 ps
Shift Left 2:            10 ps
Pipeline Register:       30 ps (write), 30 ps (read)
```

Step 1: Calculate each stage timing

- IF: 200 + 60 (pipeline reg) = 260 ps
- ID: 60 + 90 = 150 ps
- EX: 60 + 20 + 90 = 170 ps
- MEM: 60 + 250 = 310 ps ← CRITICAL
- WB: 60 + 30 = 90 ps

Step 2: Identify critical path

- Longest stage: MEM at 310 ps

Step 3: Calculate maximum frequency

```
f_max = 1 / 310 ps
      = 3.226 GHz
```

13.10.2 9.2 Exercise: Improving Clock Frequency

Question: Suggest mechanisms to increase clock frequency. Discuss negative impacts.

Suggestion 1: Faster Memory Technology

- Use SRAM instead of DRAM
- Reduce memory access time to ~100 ps
- **Pros:**
  - Significantly reduces critical path
  - New critical path: IF at 260 ps
  - Frequency increase: 310→260 (1.2× improvement)

- **Cons:**
  - SRAM very expensive
  - Much larger area
  - Higher power consumption
  - Limited capacity

Suggestion 2: Cache Memory (BEST)

- Add small, fast cache
- Cache access: ~50-100 ps
- Most accesses hit cache
- **Pros:**
  - Cost-effective
  - Good performance
  - Scalable
  - Industry standard
- **Cons:**
  - Cache misses still slow
  - Complex cache management
  - Additional hardware

Suggestion 3: Split Memory Stage

- Divide MEM into MEM1 and MEM2
- Each sub-stage: 185 ps
- Total stages: 6
- **Pros:**
  - More balanced pipeline
  - Higher frequency possible
- **Cons:**
  - More pipeline registers (overhead)
  - Increased latency
  - More complex control

Suggestion 4: Eliminate Pipeline Register Overhead

- Use transparent latches
- Reduce write+read delay
- **Pros:**
  - Removes 60 ps overhead per stage
  - Significant improvement
- **Cons:**
  - Timing more complex
  - Clock skew issues
  - Less reliable

13.10.3 9.3 Exercise: ALU Optimization Impact

Question: ALU time shortened by 25%. Does it affect speedup?

Analysis:

- Current ALU delay: 90 ps
- Reduced ALU delay: 90 × 0.75 = 67.5 ps
- Savings: 22.5 ps

Scenario 1: MEM is Critical Path (Typical)

- Current EX stage: 60 + 20 + 90 = 170 ps
- Optimized EX stage: 60 + 20 + 67.5 = 147.5 ps
- Current critical path: MEM at 310 ps
- New critical path: Still MEM at 310 ps
- Clock period: Still 310 ps
- Speedup: **NONE**

Conclusion: No improvement when not on critical path

Scenario 2: EX is Critical Path (Hypothetical)

- Assume faster memory: MEM = 200 ps
- Current EX stage: 170 ps (critical)
- Optimized EX stage: 147.5 ps
- New critical path: EX at 147.5 ps
- Clock period: 170 → 147.5 ps
- Improvement: 1.15× faster

Conclusion: Significant improvement when on critical path

General Principle:

- Only optimizing critical path improves throughput
- Non-critical optimizations: No throughput benefit
- May reduce latency slightly (instruction-by-instruction)

13.10.4 9.4 Exercise: Pipeline Speedup Calculation

Given:

- 10^7 instructions (10 million)
- Non-pipelined: 100 ps per instruction
- Perfect 20-stage pipeline

Part A: Non-pipelined execution time

```
Time = Instructions × Time per instruction
     = 10^7 × 100 ps
     = 10^9 ps
     = 1 ms (0.001 seconds)
```

Part B: Speedup from 20-stage perfect pipeline

```
Ideal Speedup = Number of stages = 20×
```

Part C: Time with perfect pipeline

```
Time = (10^7 × 100 ps) / 20
     = 10^9 / 20 ps
     = 5 × 10^7 ps
     = 0.05 ms
```

Part D: Real pipeline overhead impact

- Overheads affect both latency AND throughput
- Pipeline register delays: Add to latency
- Unbalanced stages: Reduce throughput
- Hazards and stalls: Reduce throughput further

Answer: BOTH latency and throughput affected

Latency Impact:

- Pipeline register overhead adds to per-instruction time
- 100 ps → ~130 ps per instruction (with overhead)

Throughput Impact:

- Unbalanced stages reduce effective speedup
- Perfect 20× becomes ~15-17× in reality
- Critical path limits clock speed

13.11 10. Summary and Key Takeaways

13.11.1 10.1 Pipeline Operation Fundamentals

Pipeline Registers Essential:

- Synchronize operations across stages
- Store intermediate values
- Prevent data interference
- Enable parallel execution

Timing Critical:

- Write delay + read delay at every stage
- Pipeline register overhead significant
- Critical path determines clock period
- Throughput limited by slowest stage

13.11.2 10.2 Design Principles

Make Common Case Fast:

- Memory accesses most critical
- Optimize memory access time first
- Cache memory industry solution

Balance Pipeline Stages:

- Even workload distribution

- Minimize wasted time
- Maximize efficiency

Control Signals Matter:

- Prevent unwanted operations
- Propagate through pipeline
- Essential for correctness

13.11.3 10.3 Common Mistakes to Avoid

Write Register Address:

- Must propagate through ALL pipeline registers
- Cannot use current instruction's address
- 4-cycle delay between fetch and write back

Control Signal Errors:

- RegWrite must be 0 for store/branch
- MemRead/MemWrite must be mutually exclusive
- Incorrect signals cause data corruption

Diagram Interpretation:

- Verify resource usage carefully
- Textbooks contain errors
- Understand shading conventions

13.11.4 10.4 Performance Considerations

Critical Path Analysis:

- Identify slowest stage
- Optimize critical path components
- Non-critical optimizations don't help throughput

Speedup Limitations:

- Ideal speedup = number of stages
- Actual speedup < ideal
- Reasons:
  - Pipeline register overhead
  - Unbalanced stages
  - Hazards and stalls
  - Pipeline fill/drain time

13.11.5 10.5 Looking Ahead

Memory Hierarchy (Next Topics):

- Cache memory introduction
- Memory performance optimization

- Cache design and organization
- Virtual memory
- Performance bottleneck solutions

Real-World Pipelines:

- 10-30 stages common
- Superscalar (multiple issue)
- Out-of-order execution
- Speculative execution
- Branch prediction sophistication

13.12 11. Important Formulas

13.12.1 Clock Period

```
T_clock = max(T_IF, T_ID, T_EX, T_MEM, T_WB)

Where each T_stage includes:
- Pipeline register write delay
- Pipeline register read delay
- Dominant component delay
```

13.12.2 Maximum Frequency

```
f_max = 1 / T_clock
```

13.12.3 Pipeline Speedup

```
Speedup = T_non-pipelined / T_pipelined_steady_state
          Number of stages (ideal)
        < Number of stages (actual)
```

13.12.4 Stage Timing General Formula

```
T_stage = T_pipe_write + T_pipe_read + T_dominant_component + T_other_parallel

Where parallel components don't add (take maximum)
```

13.12.5 Throughput

```
Throughput = 1 instruction / T_clock (steady state)
```

13.12.6 Latency

```
Latency = (Number of stages) × T_clock + Pipeline overhead
```

13.13 Key Takeaways

1. **Four pipeline registers separate five stages**: IF/ID, ID/EX, EX/MEM, MEM/WB store all information needed by subsequent stages.
2. **Pipeline registers capture data and control signals**—instruction fields, register values, ALU results, memory data, and control bits all propagate through pipeline.
3. **Each register updates on clock edge**—enabling clean separation between pipeline stages and preventing data corruption from simultaneous operations.
4. **Load instruction takes 5 cycles to complete**—IF (fetch), ID (decode/read), EX (address calc), MEM (read memory), WB (write register).
5. **Store instruction uses 4 active stages**—skips WB stage since no register write occurs, but occupies pipeline for 5 cycles.
6. **Instruction and data must travel together**—control signals propagate alongside data through pipeline to ensure correct operations at later stages.
7. **Register file has two write ports and three read ports** in practice—enabling simultaneous read in ID and write in WB stages.
8. **Forwarding paths bypass pipeline registers**—directly connecting EX/MEM and MEM/WB outputs to ALU inputs for data hazard resolution.
9. **Load-use hazard requires pipeline stall**—memory data not available until MEM/WB register, too late for immediate ALU use even with forwarding.
10. **Clock frequency** = 1 / (Register Delay + Maximum Stage Delay)—pipeline register overhead reduces frequency below ideal calculation.
11. **Pipeline registers introduce 20-50 ps overhead** per stage—must account for setup/hold times and propagation delays in timing analysis.
12. **Stage delays must balance for optimal performance**—uneven stages waste time as clock period determined by slowest stage.
13. **Separate instruction and data caches essential**—prevent structural hazards from simultaneous IF and MEM stage memory access.
14. **Pipeline depth tradeoff**: Deeper pipelines increase clock frequency but amplify hazard penalties and register overhead.
15. **Write-back stage coincides with fetch of fifth instruction**—demonstrating true parallelism with five instructions in pipeline simultaneously.
16. **Control signals generated in ID stage** propagate through pipeline with instruction—EX/MEM/WB stages use stored control bits.

17. **ALU result available in EX stage** can forward to dependent instruction in EX stage—eliminating most RAW hazard stalls.

18. **Memory data available in MEM stage** can forward to dependent instruction in EX stage—but not soon enough for load-use case.

19. **Throughput approaches 1 instruction per cycle** in steady state—achieving near 5× speedup over single-cycle design.

20. **Pipeline timing analysis critical for clock frequency determination**—must consider all delay components including registers, logic, and wire delays.

13.14 Summary

The detailed examination of MIPS pipeline operation reveals the sophisticated hardware mechanisms that enable efficient instruction-level parallelism through careful staging and register design. Four pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB) serve as the critical infrastructure separating five pipeline stages, capturing and propagating not only instruction data but also all control signals needed by downstream stages. The cycle-by-cycle analysis of load and store instructions demonstrates how each pipeline stage performs its designated function while simultaneously handling different instructions—instruction fetch occurring for instruction N while instruction N-1 decodes, N-2 executes, N-3 accesses memory, and N-4 writes back results. This true parallelism, with five instructions simultaneously occupying different pipeline stages, achieves the dramatic throughput improvement that justifies pipeline complexity. The timing analysis introduces crucial practical considerations: pipeline registers add 20-50 picoseconds overhead per stage, stage delays must balance to avoid wasting clock cycles, and clock frequency equals the reciprocal of register delay plus maximum stage delay. Forwarding paths that bypass pipeline registers—connecting EX/MEM and MEM/WB outputs directly to ALU inputs—eliminate most data hazard stalls by making results available before register write-back completes, though load-use hazards still require one-cycle stalls since memory data arrives too late even with forwarding. The register file's dual-port design enables simultaneous reading in ID stage and writing in WB stage, essential for maintaining pipeline flow. Practical exercises in clock frequency calculation reinforce understanding of how component delays, register overhead, and stage balancing determine ultimate processor performance. The separation of instruction and data caches emerges as non-negotiable requirement, preventing structural hazards from simultaneous memory access in IF and MEM stages. This comprehensive pipeline view—from register-level mechanisms through timing analysis to performance optimization—provides essential foundation for understanding real processor implementations and the engineering tradeoffs between pipeline depth, clock frequency, hazard penalties, and design complexity that characterize modern computer architecture.

Chapter 14

Lecture 14: Introduction to Memory Systems and Cache Memory

14.1 Introduction

This lecture marks a crucial transition from CPU-centric topics to memory systems, introducing cache memory as the elegant solution to the fundamental processor-memory speed gap. We begin with historical context, tracing how stored-program concept revolutionized computing, then explore the memory hierarchy that creates the illusion of large, fast memory through careful exploitation of temporal and spatial locality. The direct-mapped cache organization receives detailed treatment, establishing foundational concepts of blocks, tags, indices, and valid bits that underpin all cache designs. Understanding cache memory proves as essential as understanding processor architecture, as memory system performance often determines overall computer system speed in practice.

14.2 1. Lecture Introduction and Historical Context

14.2.1 1.1 Course Transition

Previous Topics:

- CPU datapath and control (ARM, MIPS, pipelining)

New Focus:

- Memory systems (equally important as CPU)

Motivation:

- Memory plays as significant a role as CPU in modern computer architecture

14.2.2 1.2 Historical Background

14.2.2.1 Early Computing Machines (1940s)

Examples:

- ENIAC (University of Pennsylvania)
- Harvard Mark I (ASTC)

Characteristics:

- Filled entire rooms
- Built using vacuum tubes and electrical circuitry
- Developed for war efforts (World War II)
- Used for artillery planning, nuclear weapon calculations
- No concept of software or memory as we know today

Programming Method:

- Rewiring the entire machine for each algorithm
- Engineers spent days/weeks reconfiguring machines
- No stored program concept

14.2.3 1.3 Key Historical Figures

14.2.3.1 Alan Turing (1936)

- British mathematician, brilliant mind
- First conceived the stored program computer concept
- Designed the Universal Turing Machine (hypothetical machine)
- First notion of memory, programs stored in computers, data read/write operations
- Later involved in World War II cryptography (Enigma Machine, “The Imitation Game”)

14.2.3.2 John von Neumann (1940s)

- Hungarian mathematician, regarded as “last of the brilliant mathematicians”
- Prodigy: Solving calculus problems by age 8
- Contributed across many fields
- Got involved with EDVAC computer project
- Implemented stored program concept based on Turing’s ideas

14.2.4 1.4 First Stored Program Computers

14.2.4.1 EDVAC (1948)

- Commissioned by U.S. Army
- John von Neumann involved as consultant
- Memory: Initially 1044 words, upgraded to 1024 words (power of 2)
- First machine with stored program concept
- Memory stored program electrically (not in wiring)
- Engineers created the first “memory” device

- First test programs: Nuclear weapon detonation calculations, hydrogen bomb calculations

14.2.4.2 Von Neumann Architecture

Key Concept:

- Data AND instructions both in SAME memory
- Access data and programs through SAME connection pathways
- Unified memory for instructions and data
- This concept became foundation of modern computers

14.2.4.3 EDSAC (Cambridge University)

- Built about a year after EDVAC
- First machine fully implementing Von Neumann architecture
- Memory: 512 words of 18 bits each
- Also built for war effort

14.2.4.4 Harvard Architecture (Contrasted)

- Separate storages for instructions and data
- Separate connections to instruction memory and data memory
- Used in MIPS datapath design (separate instruction memory and data memory)

Modern Computers:

- Use a MIX of both Von Neumann and Harvard architectures
- Features from both types incorporated

14.3 2. Memory Technologies: Types and Characteristics

14.3.1 2.1 Commonly Used Memory Technologies Today

- SRAM (Static RAM)
- DRAM (Dynamic RAM)
- Flash Memory
- Magnetic Disk
- Magnetic Tape

14.3.2 2.2 SRAM (Static RAM)

Property	Value/Description
Technology	Built using flip-flops

Property	Value/Description
Volatility	Volatile (loses content when power lost)
Access Time	Less than 1 nanosecond (< 1 ns)
Clock Frequency	More than 1 GHz
Cycle Time	Less than 1 nanosecond (< 1 ns)
Capacity	Kilobytes to Megabytes range
Cost	~$2000 per gigabyte (VERY EXPENSIVE)
Speed	Extremely fast
Usage	Cache memories (small amounts due to cost)

Note on Cycle Time:

- Cycle time = minimum time between two consecutive memory accesses
- Access time ≈ Cycle time for SRAM

14.3.3 2.3 DRAM (Dynamic RAM)

Property	Value/Description
Technology	Transistors + Capacitors
Volatility	Volatile (requires power AND periodic refresh)
Access Time	~25 nanoseconds (50 ns in some contexts)
Cycle Time	~50 nanoseconds (double the access time)
Capacity	Gigabytes (8 GB, 16 GB, or more)
Cost	~$10 per gigabyte
Usage	Main memory in computers

Key Characteristics:

- Capacitor charge must be maintained
- “Destructive read”: Reading loses the charge, requires rewrite/refresh
- Longer cycle time due to refresh requirement
- After reading, must rewrite data to same cell
- Significantly slower than SRAM (25-50 ns vs < 1 ns)

14.3.4 2.4 Flash Memory

Property	Value/Description
Technology	NAND MOSFET (NAND gate with two gates)
Volatility	Non-volatile (retains data without power)
Access Time	~70 nanoseconds
Cycle Time	~70 nanoseconds
Capacity	Gigabytes range
Cost	Less than $1 per gigabyte
Usage	Secondary storage (SSDs - Solid State Devices/Drives)

Limitation:

- Limited read/write cycles
- After several thousand cycles, memory cells may degrade
- Integrity decreases, capacity effectively decreases
- Slightly slower than DRAM, but non-volatile

14.3.5 2.5 Magnetic Disk

Property	Value/Description
Technology	Magnetic (mechanical device)
Access Time	5 to 10 milliseconds (MUCH slower than electronic memory!)
Cycle Time	Similar to access time (~5-10 ms)
Capacity	Several terabytes
Cost	Fraction of a dollar per gigabyte (very cheap)

Usage:

- Previously: Main secondary storage
- Currently: Being replaced by flash/SSDs for secondary storage
- Now used primarily for tertiary storage, backups
- Good for long-term data retention, low cost
- Slowness acceptable for infrequent backup operations

Note: Average numbers; varies by data location on disk. Mechanical: spinning platters, moving read/write heads.

14.4 3. The Memory Performance Problem

14.4.1 3.1 The CPU-Memory Speed Gap

CPU Clock Cycle:

- Modern CPUs: > 1 GHz clock frequency
- Clock cycle: < 1 nanosecond (1 ns corresponds to 1 GHz)

Main Memory (DRAM):

- Cycle time: ~50 nanoseconds
- Time between starts of two consecutive memory accesses: 50 ns

14.4.2 3.2 The Problem

Speed Discrepancy:

- CPU cycle: < 1 ns
- Memory cycle: 50 ns

- **Memory is 50× SLOWER than CPU!**

14.4.3 3.3 Impact on Pipelining

The Challenge:

- In MIPS pipeline, MEM stage must finish in ONE clock cycle
- Every pipeline stage must take same time
- How can MEM stage complete in 1 ns when memory takes 50 ns?
- Pipeline performance would be severely degraded

The Contradiction:

- CPU expects 1 ns memory access
- Actual DRAM takes 50 ns
- "Something is not right" - how can this work?

14.5 4. Memory Hierarchy Concept

14.5.1 4.1 The Solution: Memory Hierarchy

Core Idea:

- Trick the CPU into thinking memory is BOTH fast AND large
- Desired characteristics:
  - Fast access times (like SRAM: < 1 ns)
  - Large capacity (like Disk: terabytes)
- These characteristics don't exist in single technology
- Solution: Implement memory as a HIERARCHY

14.5.2 4.2 Memory Hierarchy Structure

```
Level 1 (Top): SRAM (Cache)
- Smallest capacity
- Fastest speed
- Closest to CPU physically

Level 2: DRAM (Main Memory)
- Medium capacity
- Medium speed

Level 3 (Bottom): Disk
- Largest capacity
- Slowest speed
```

14.5.3 4.3 Key Principles

14.5.3.1 1. CPU Access Restriction

- CPU can ONLY access top level (SRAM cache)
- CPU thinks cache is the actual memory
- CPU cannot directly access DRAM or Disk

14.5.3.2 2. CPU's Perception

- Experiences the SPEED of SRAM
- Feels the CAPACITY of DRAM and Disk combined
- Illusion: Memory is as fast as SRAM AND as big as lowest level

14.5.3.3 3. Data Organization

- Upper levels contain SUBSET of data from lower levels
- SRAM (few MB) contains subset of DRAM (several GB)
- DRAM contains subset of Disk (several TB)
- At any given time, each level holds only a fraction of lower level's data

14.5.3.4 4. Hierarchy Characteristics

- Devices up the hierarchy: Smaller and faster
- Devices down the hierarchy: Larger but slower

14.5.4 4.4 The Challenge

What if CPU asks for data NOT in the cache (top level)?

- Need mechanism to copy data from lower levels
- This leads to the concepts of hits, misses, and cache management

---

14.6 5. Analogy: Music Library

14.6.1 5.1 Understanding Memory Hierarchy Through Music

14.6.1.1 Three-Level Music System

1. Mobile Phone (analogous to SRAM/Cache):

- Carries a subset of your favorite songs
- Always with you
- Listen to music directly from phone
- Limited storage (like cache has limited capacity)

2. Computer Hard Disk (analogous to DRAM/Main Memory):

- Main music collection stored here

- Larger collection than phone
- Not always accessible (not in pocket)
- Copy songs from here to phone when needed

3. Internet (analogous to Disk/Mass Storage):

- All songs available (massive storage)
- Download/buy songs from here
- Copy to computer, then to phone

14.6.2 5.2 Usage Scenarios

14.6.2.1 Scenario 1 (Hit)

- Want to listen to a song
- Song is already on phone
- Just play it directly
- Similar to cache hit: Data already in cache

14.6.2.2 Scenario 2 (Miss to Level 2)

- Want to listen to a song
- Song NOT on phone
- Must go to computer and copy to phone
- Then listen on phone
- Similar to cache miss: Must fetch from main memory

14.6.2.3 Scenario 3 (Miss to Level 3)

- Want to listen to a song
- Song NOT on phone AND NOT on computer
- Download from internet to computer
- Copy to phone
- Then listen
- Similar to cache miss to disk: Must fetch from lowest level

14.6.3 5.3 Key Parallels

- Always listen from phone (CPU always accesses cache)
- Main collection in computer (main memory holds primary data)
- All data available on internet (disk holds everything)
- Copy operations when data not available at higher levels

14.7 6. Memory Hierarchy Terminology

14.7.1 6.1 Essential Terms for Memory Access

14.7.1.1 HIT

Definition: Requested data IS available at the accessed level

- CPU requests data → Data found in cache
- Like wanting to listen to song already on your phone
- Can be served immediately from that level

14.7.1.2 MISS

Definition: Requested data is NOT available at the accessed level

- CPU requests data → Data NOT found in cache
- Like wanting to listen to song not on your phone
- Must fetch from lower level in hierarchy

14.7.1.3 HIT RATE

Definition: Ratio/percentage of accesses that result in hits

Formula:

```
Hit Rate = (Number of Hits) / (Total Accesses)
```

Example: 100 accesses, 90 hits → Hit Rate = 90% or 0.9

Indicates how often data is found at the accessed level. Higher hit rate = better performance.

14.7.1.4 MISS RATE

Definition: Ratio/percentage of accesses that result in misses

Formula:

```
Miss Rate = (Number of Misses) / (Total Accesses)
Miss Rate = 1 - Hit Rate
```

Example: 100 accesses, 10 misses → Miss Rate = 10% or 0.1

Lower miss rate = better performance.

14.7.1.5 HIT LATENCY

Definition: Time taken to determine if access is a hit AND serve the data

- Time to check if data is in cache and deliver it to CPU
- For SRAM cache: < 1 nanosecond

Components:

- Time to search cache
- Time to verify data presence
- Time to extract and send data to CPU

14.7.1.6 MISS PENALTY

Definition: EXTRA time required when access is a miss

Process:

1. Determine it's a miss (hit latency spent)
2. Go to next level (DRAM)
3. Find the data
4. Copy to cache
5. Put in appropriate place
6. Deliver to CPU

Key Points:

- Total time on miss = Hit Latency + Miss Penalty
- Miss penalty for DRAM access can be 100× hit latency
- Very expensive in terms of time!

14.8 7. Performance Impact and Requirements

14.8.1 7.1 Average Memory Access Time

Formula:

```
Average Access Time = Hit Latency + (Miss Rate × Miss Penalty)
```

Explanation:

- ALL accesses consume hit latency (must check cache)
- Only misses consume additional miss penalty
- Miss Rate determines portion of accesses incurring penalty

14.8.2 7.2 Example Analysis

Given:

- Hit Latency (SRAM): < 1 nanosecond
- Miss Penalty (DRAM access): ~100 nanoseconds (100× slower)
- CPU clock cycle: < 1 nanosecond

14.8.2.1 For Pipeline to Work

- MEM stage must complete in 1 clock cycle
- Memory access must complete in < 1 ns most of the time

14.8.2.2 Required Hit Rate Calculation

If Hit Rate = 99.9% (Miss Rate = 0.1%):

```
Average Time = 1 ns + (0.001 × 100 ns)
             = 1 ns + 0.1 ns
             = 1.1 ns
```

Still close to 1 clock cycle!

If Hit Rate = 90% (Miss Rate = 10%):

```
Average Time = 1 ns + (0.10 × 100 ns)
             = 1 ns + 10 ns
             = 11 ns
```

Unacceptable! 11× slower than CPU clock!

14.8.3 7.3 Critical Requirement

- Need VERY HIGH hit rate at cache level
- Not just high, but VERY, VERY high
- Target: **99.9% or better**
- Only 0.1% of accesses should go to memory

14.8.4 7.4 Performance Implications

14.8.4.1 With 99.9% Hit Rate

- 99.9% of time: CPU works fine, memory appears fast
- 0.1% of time: CPU must STALL, wait for data from DRAM
- Stall is unavoidable for misses
- Overall: CPU maintains illusion of fast, large memory

14.8.4.2 With Lower Hit Rate

- More frequent stalls
- Pipeline performance degrades significantly
- Average memory access time increases
- CPU slows down dramatically

Conclusion:

- Must ensure VERY high hit rate at SRAM level
- Memory hierarchy only works if locality principles hold
- Like having most songs you want to listen to already on phone
- Don’t want to copy from computer frequently (time-consuming)

14.9 8. Principles of Locality

14.9.1 8.1 Foundation for Memory Hierarchy Success

Nature of Computer Programs:

- Programs access only SMALL portion of entire address space at any given time
- Address space: Entire memory range (address 0 to maximum address)
- At any time window, program uses only small fraction of total data
- True by nature of how programs are written, compiled, and executed
- True for instruction sets like ARM, MIPS

14.9.2 8.2 Temporal Locality (Locality in Time)

14.9.2.1 Definition

"Recently accessed data are likely to be accessed again soon"

Explanation:

- If you access memory address A at time T
- High probability of accessing address A again at time T+ΔT (soon after)
- Same data accessed multiple times in short time window
- "Locality in time" - data clustered temporally

14.9.2.2 Common Examples in Programs

a) Loop Index Variables:

```
for (int i = 0; i < 100; i++) {
    // i is accessed every iteration
    // Same memory location for 'i' accessed repeatedly
}
```

b) Loop-Invariant Data:

```
for (int i = 0; i < n; i++) {
    result = result + array[i] * constant;
    // 'result' and 'constant' accessed every iteration
}
```

c) Function/Procedure Calls:

- Local variables accessed multiple times during function execution
- Same stack frame locations accessed repeatedly

d) Instructions:

- Loop body instructions executed many times
- Same instruction addresses accessed repeatedly

14.9.2.3 Music Analogy

- If you listen to a song, you’re likely to listen to it again soon
- Sometimes listen to same song 10 times in a row
- Want to replay favorite songs

14.9.2.4 Degree of Temporal Locality

- Varies from program to program
- But present in nearly ALL programs
- Stronger in some (tight loops) than others

14.9.3 8.3 Spatial Locality (Locality in Space)

14.9.3.1 Definition

“Data located close to recently accessed data are likely to be accessed soon”

Explanation:

- If you access memory address A at time T
- High probability of accessing addresses A+1, A+2, A+3, … soon after
- Sequential or nearby addresses accessed together
- “Locality in space” - data clustered spatially in memory

14.9.3.2 Common Examples in Programs

a) Array Traversal:

```
for (int i = 0; i < 100; i++) {
    sum += array[i];
    // Access array[0], then array[1], then array[2], ...
    // Sequential memory addresses
}
```

b) Sequential Instruction Execution:

- Instructions stored sequentially in memory
- PC increments: fetch instruction at PC, then PC+4, then PC+8, …
- Except for branches, mostly sequential

c) Data Structures:

```
struct Student {
    int id;
    char name[50];
    float gpa;
};
Student s;
// Accessing s.id, then s.name, then s.gpa
// Nearby memory locations
```

d) String Processing:

```
char str[] = "Hello";
for (int i = 0; str[i] != '\0'; i++) {
    // Access str[0], str[1], str[2], ...
    // Consecutive bytes in memory
}
```

14.9.3.3 Music Analogy

- If you listen to song by artist X, likely to listen to another song by artist X
- If you listen to song from album Y, likely to listen to next song in album Y
- Related/nearby songs accessed together

14.9.3.4 Degree of Spatial Locality

- Varies by data access patterns
- Strong in array-based algorithms
- Present in most structured programs

14.9.4 8.4 Universal Applicability

- Both principles hold true for NEARLY ALL programs
- Degree varies, but principles universally applicable
- Foundation assumptions for cache design

14.10 9. Cache Memory Concept and Block-Based Operation

14.10.1 9.1 Cache Memory Overview

Purpose:

- Memory device at top level of hierarchy
- Based on two principles of locality
- Decides what data to keep based on locality principles

14.10.2 9.2 Data Organization: BLOCKS

Key Concepts:

- CPU requests individual WORDS from memory
- Between cache and memory: Handle BLOCKS of data
- **Block** = multiple consecutive words
- Block size example: 8 bytes = 2 words (with 4-byte words)
- Hidden from CPU (CPU still thinks in words)

14.10.3 9.3 Why Blocks? (Spatial Locality)

14.10.3.1 Instead of Words

- Fetch single word CPU requested
- Next access likely nearby address
- Would require another fetch

14.10.3.2 Using Blocks

- Fetch requested word AND nearby words together
- Bring entire block (e.g., 8 consecutive bytes)
- Subsequent accesses likely in same block (spatial locality)
- Reduces future misses

14.10.3.3 Music Library Analogy

- Want to listen to one song → Copy entire album to phone
- Not just the single song you want right now
- Because you'll likely want other songs from same album soon
- Saves future copy operations

14.10.3.4 Block Benefits

- Exploits spatial locality
- Reduces miss rate
- Amortizes fetch cost over multiple words
- More efficient use of memory bandwidth

14.10.4 9.4 Cache Management Decisions

14.10.4.1 1. What to Keep in Cache

- Based on BOTH locality principles
- Recently accessed data (temporal locality)
- Blocks containing nearby data (spatial locality)

14.10.4.2 2. What to Evict from Cache

- Based on TEMPORAL locality
- When cache full and need space for new block
- Must throw out existing data

14.10.5 9.5 Eviction Strategy (Ideal)

Least Recently Used (LRU):

- Throw out LEAST RECENTLY USED (LRU) data
- If cache has 10 blocks, need to evict 1

- Choose the block that was used longest time ago
- Keep more recently used blocks
- Temporal locality suggests LRU block least likely to be accessed soon

Example:

- Cache has blocks A, B, C, D, E
- Last access times: A(10 cycles ago), B(2 cycles ago), C(50 cycles ago), D(5 cycles ago), E(1 cycle ago)
- Need to evict one block
- Evict C (least recently used, 50 cycles ago)
- Keep E, B, D, A (more recently used)

14.11 10. Memory Addressing: Bytes, Words, and Blocks

14.11.1 10.1 Byte Address

Definition: Address referring to individual byte in memory

Characteristics:

- Each byte-sized location has unique address
- Standard memory addressing
- Address Space: With 32-bit address, can access 2^{32} individual bytes

Example Address:

```
Address: 00000000000000000000000000001010 (binary)
       = 10 (decimal)
Points to: Byte at memory location 10
```

Memory Structure:

```
Address 0:  [byte 0]
Address 1:  [byte 1]
Address 2:  [byte 2]
...
Address 10: [byte 10]  ← This byte addressed by example
...
```

14.11.2 10.2 Word Address

Definition: Address referring to a word (multiple bytes) in memory

Typical Word Size: 4 bytes (32 bits)

14.11.2.1 Word Alignment

- Words start at addresses that are multiples of 4

- Word 0: Addresses 0, 1, 2, 3
- Word 1: Addresses 4, 5, 6, 7
- Word 2: Addresses 8, 9, 10, 11
- Word 3: Addresses 12, 13, 14, 15

14.11.2.2 Word Address Format (32-bit)

```
[30-bit word identifier][2-bit byte offset]
                          Always "00" for word-aligned addresses
```

Example:

```
Address: ...00001000 (binary)
         - Last 2 bits: 00 → Word-aligned
         - Remaining bits: Identify which word
         - This is address 8, start of word 2
```

14.11.2.3 Byte Within Word

Last 2 bits select byte within word:

- `00` → First byte (address 8)
- `01` → Second byte (address 9)
- `10` → Third byte (address 10)
- `11` → Fourth byte (address 11)

Key Points:

- Word addresses are multiples of 4
- Can divide by 4 without remainder
- Last 2 bits = 00 for word addresses
- NOT all addresses ending in 00 are word addresses, but word addresses end in 00
- Only portion of address except last 2 bits identifies the word

14.11.3 10.3 Block Address

Definition: Address referring to a block (multiple words) in memory

Example Block Size: 8 bytes = 2 words

14.11.3.1 Block Alignment

- Blocks start at addresses that are multiples of 8
- Block 0: Addresses 0-7
- Block 1: Addresses 8-15
- Block 2: Addresses 16-23
- Block 3: Addresses 24-31

14.11.3.2 Block Address Format (32-bit)

```
[Block Identifier][3-bit offset]
                     Last 3 bits for 8-byte blocks
```

Example:

```
Address: 00000000000000000000000000101101 (binary)
         = 45 (decimal)

Block Address Portion:
- Ignore last 3 bits: 00101 (offset part)
- Block address: 00000000000000000000000000101 (identifies block)
- This identifies the block containing address 45
```

14.11.3.3 Offset Within Block (3 bits for 8-byte blocks)

BYTE OFFSET (all 3 bits):

- Used to identify individual BYTE within block
- `000` → Byte 0
- `001` → Byte 1
- …
- `111` → Byte 7

WORD OFFSET (most significant bit of offset):

- Used to identify WORD within block (when block has 2 words)
- `0XX` → First word (bytes 0-3)
- `1XX` → Second word (bytes 4-7)
- Only need 1 bit to select between 2 words

14.11.4 10.4 Address Components Summary

For address with 8-byte blocks, 4-byte words:

```
[Block Address][Word Offset][Byte in Word]
     ^              ^              ^
     |              |                 2 bits: Select byte within word
     |                 1 bit: Select word within block
       Remaining bits: Identify which block
```

Example Breakdown:

```
Address: ...00101101
- Last 2 bits (01): Byte offset within word → Byte 1 of word
- 3rd bit from right (1): Word offset → Second word of block
- Remaining bits (...00101): Block address → Block 5
```

All Bytes in Same Block:

- Share same block address
- Differ only in offset bits

Important Distinctions:

- **Byte address:** Full 32 bits
- **Word address:** Term refers to full address of word-aligned location
- **Block address:** Term refers to portion of address identifying block (excluding offset)

14.12 11. The Cache Addressing Problem

14.12.1 11.1 Problem Statement

14.12.1.1 In Main Memory

- Direct addressing: Address 10 → Direct access to location 10
- Like array indexing: array[10] directly accesses index 10
- Straightforward: Address uniquely identifies memory location
- No search required: Hardware directly decodes address

14.12.1.2 In Cache

- Cache is MUCH smaller than memory
- Memory: Gigabytes (millions/billions of addresses)
- Cache: Kilobytes or Megabytes (thousands/few million bytes)
- Example: Memory has 1 million addresses, cache has only 8 slots

14.12.2 11.2 The Challenge

- CPU generates address from full address space (e.g., address 10)
- Cache has only 8 slots (indices 0-7)
- Cannot directly use memory address as cache index
- Address 10 doesn't directly map to cache location
- **How to find data in cache with memory address?**

14.12.3 11.3 Initial Solution Idea: Store Addresses with Data

Approach:

- Store memory address alongside data in cache
- Each cache entry: [Address | Data]
- When CPU requests address, search cache for matching address

Problems with This Approach:

1. Space Overhead:

- Must store full address (e.g., 32 bits) with each data block
- Significant storage overhead
- Example: 32-bit address + 256-bit data block = ~13% overhead

2. Search Time:

- Must search through ALL cache entries
- Sequential or parallel search required
- Example: 8 cache slots → Check all 8 tags
- Time-consuming, degrades hit latency
- Cannot directly access cache entry

14.12.4 11.4 Need for Better Solution

Requirements:

- Require MAPPING between memory addresses and cache locations
- Want DIRECT access (no search) if possible
- Must be efficient in both space and time

Requirements for Practical Cache:

1. Fast access (< 1 ns hit latency)
2. Minimal storage overhead
3. Direct or near-direct cache indexing
4. Efficient tag comparison (if needed)

Solution Preview: Address Mapping Functions

- Need function: Memory Address → Cache Location
- Different mapping strategies possible
- Simplest: Direct Mapping (discussed next)

14.13 12. Direct-Mapped Cache

14.13.1 12.1 Direct Mapping Concept

Definition:

- Each memory address maps to EXACTLY ONE cache location
- One-to-one deterministic mapping
- No choice in cache placement

Mapping Rule:

```
Cache Index = Block Address MOD (Number of Blocks in Cache)
```

Formula:

```
Cache Index = (Block Address) mod (Cache Size in Blocks)
```

Example:

- Cache has 8 blocks → Indices 0-7
- Block address = 13
- Cache index = 13 mod 8 = 5

- Block 13 maps ONLY to cache index 5

14.13.2 12.2 Mathematical Properties

14.13.2.1 Mod Operation with Powers of 2

- Cache sizes typically powers of 2 (1, 2, 4, 8, 16, 32, …)
- Mod by power of 2 = take least significant bits
- Example: N mod 8 = N mod 2^3 = last 3 bits of N

14.13.2.2 Hardware Implementation

- No division circuit needed!
- Simply extract least significant bits
- Very fast, pure combinational logic

14.13.3 12.3 Direct Mapping Example

Given:

- Block size: 8 bytes
- Cache size: 8 blocks
- Cache indices: 0, 1, 2, 3, 4, 5, 6, 7

Cache Structure (Initial View):

Index	Data Block
0	[64 bits]
1	[64 bits]
2	[64 bits]
3	[64 bits]
4	[64 bits]
5	[64 bits]
6	[64 bits]
7	[64 bits]

Example Addresses:

Address 1:

```
Binary: ...00000001[011]
           Block address = 0
           Offset = 3 bytes
Cache index = 0 mod 8 = 0
Maps to cache index 0
```

Address 2 (block address in focus):

```
Binary: ...00000101[000]
           Block address = 5
           Offset = 0
Cache index = 5 mod 8 = 5
Maps to cache index 5
```

14.13.4 12.4 Address Structure for Direct-Mapped Cache

```
[Tag][Index][Offset]
  ^     ^       ^
  |     |          Identifies byte/word within block
  |        Identifies cache location (index)
     Remaining bits to differentiate blocks mapping to same index
```

14.13.4.1 Bit Allocation (for 8-block cache, 8-byte blocks, 32-bit address)

- **Offset:** 3 bits (for 8-byte blocks: $2^3 = 8$)
- **Index:** 3 bits (for 8 cache blocks: $2^3 = 8$)
- **Tag:** 26 bits (remaining: 32 - 3 - 3 = 26)

14.13.4.2 Index Bits

- Least significant bits of block address
- Directly select cache location
- Number of bits = log□(cache blocks)
- 8 blocks → 3 index bits
- 16 blocks → 4 index bits
- 32 blocks → 5 index bits

14.14 13. The Tag Problem in Direct-Mapped Cache

14.14.1 13.1 Conflict Issue

Multiple Blocks → Same Index:

- Many memory blocks map to same cache index
- Example: Blocks 5, 13, 21, 29, … all map to index 5 (mod 8)
- Only ONE can occupy cache index 5 at a time

Example Addresses Mapping to Index 5:

Address A:

```
Block address: ...00000101
Index bits (last 3): 101 → Index 5
```

Address B:

```
Block address: ...00001101
Index bits (last 3): 101 → Index 5
```

Both map to index 5, but different blocks!

14.14.2 13.2 The Problem

- When CPU requests address with index 5
- Is data at index 5 for Address A or Address B?
- Need way to differentiate between conflicting blocks

14.14.3 13.3 Solution: TAG FIELD

Tag Definition:

- Remaining bits of block address (excluding index and offset)
- Stored WITH data in cache
- Used to verify correct block is present

```
Tag = Block Address (excluding index bits)
```

14.14.3.1 Example Address Breakdown

Full Address:

```
[26-bit Tag][3-bit Index][3-bit Offset]
```

Address A:

```
[00000000000000000000000000][101][000]
   Tag = 0                   Index=5   Offset
```

Address B:

```
[00000000000000000000000001][101][000]
   Tag = 1                   Index=5   Offset
```

Both have index 5, but DIFFERENT tags!

14.14.4 13.4 Cache Structure with Tags

Index	Valid	Tag	Data Block
0	V	Tag0	[64 bits]
1	V	Tag1	[64 bits]
2	V	Tag2	[64 bits]
3	V	Tag3	[64 bits]
4	V	Tag4	[64 bits]
5	V	Tag5	[64 bits]
6	V	Tag6	[64 bits]
7	V	Tag7	[64 bits]

Storage Requirements Per Cache Entry:

- Tag: 26 bits (in this example)
- Valid bit: 1 bit
- Data: 64 bits (8 bytes)
- Total: 91 bits per entry

Storage Overhead:

```
Overhead = (Tag + Valid) / Total
         = (26 + 1) / (26 + 1 + 64)
         = 27 / 91
           30% overhead in this small example
```

14.14.4.1 Note on Overhead

- Example uses VERY small cache (8 blocks)
- Real caches are much larger (thousands of blocks)
- Larger caches → More index bits
- More index bits → Fewer tag bits
- Overhead percentage decreases with larger caches

Example with Larger Cache:

- 1024 blocks (2^{10})
- Index: 10 bits
- Tag: 32 - 10 - 3 = 19 bits
- Overhead: (19+1)/84 ≈ 24% (better)

14.14.5 13.5 Valid Bit

Purpose:

- Indicates whether cache entry contains valid data
- Prevents using uninitialized/stale data

Initial State:

- At program start, cache is empty
- All entries contain garbage/random values
- All valid bits set to 0 (invalid)

After Data Loaded:

- When block loaded into cache, valid bit set to 1
- Indicates data is reliable

Uses Beyond Initialization:

- Cache coherence (multi-processor systems)
- Invalidating stale data
- Handling context switches
- Will see more uses later in course

14.15 14. Cache Read Access Operation

14.15.1 14.1 Read Access Process

CPU Provides:

1. Address (word or byte address)
2. Control Signal: Read/Write indicator (from control unit)

14.15.2 14.2 For Read Access

14.15.2.1 Step 1: ADDRESS BREAKDOWN

- Receive address from CPU
- Parse into three fields:
  - Tag bits
  - Index bits
  - Offset bits

Example Address (32-bit):

```
[26-bit Tag][3-bit Index][3-bit Offset]
```

14.15.2.2 Step 2: INDEXING THE CACHE

- Extract index bits from address
- Use index to directly access cache entry
- Combinational logic routes to correct entry
- Like array indexing: index 5 → entry 5
- No search needed!
- Fast: Pure combinational delay

Hardware:

- Decoder circuit takes index bits
- Selects one of N cache entries
- Activates corresponding row

14.15.2.3 Step 3: TAG COMPARISON

- Extract stored tag from selected cache entry
- Extract tag bits from incoming address
- Compare the two tags
- Use comparator circuit

Comparator Circuit:

- For each bit position: XNOR gate
- XNOR outputs 1 if bits match, 0 if different
- AND all XNOR outputs together
- Final output: 1 if all bits match (tags equal), 0 otherwise

Example (4-bit tags):

```
Stored tag:   1 0 1 1
Address tag:  1 0 1 1
XNOR:         1 1 1 1  → AND = 1 (MATCH!)

Stored tag:   1 0 1 1
Address tag:  1 0 0 1
XNOR:         1 1 0 1  → AND = 0 (NO MATCH)
```

For N-bit tag:

- N XNOR gates (parallel)
- 1 N-input AND gate
- Very fast combinational circuit

14.15.2.4 Step 4: VALID BIT CHECK

- Extract valid bit from selected cache entry
- Check if entry is valid
- Valid bit = 1 → Entry contains valid data
- Valid bit = 0 → Entry is invalid (ignore)

14.15.2.5 Step 5: HIT/MISS DETERMINATION

- Combine tag comparison and valid bit
- Hit = (Tag Match) AND (Valid Bit = 1)
- Miss = (Tag Mismatch) OR (Valid Bit = 0)

Logic Circuit:

```
Tag Match Output
                  AND  → Hit/Miss Signal
Valid Bit
```

Output:

- 1 → HIT (data present and valid)
- 0 → MISS (data not present or invalid)

Hit Latency:

- Time for steps 2-5
- Dominated by:
  - Indexing combinational delay
  - Tag comparator delay
  - Valid bit access
- Typically < 1 nanosecond for SRAM

14.15.2.6 Step 6: DATA EXTRACTION (Parallel with Tag Check)

- Can happen in PARALLEL with tag comparison
- Extract entire data block from selected cache entry

- Put data block on internal wires

Data Block:

- Contains multiple words
- Example: 8 bytes = 2 words (4 bytes each)

14.15.2.7 Step 7: WORD SELECTION (Using Offset)

- CPU wants a single WORD, not entire block
- Use offset bits to select correct word from block
- Offset bits → Multiplexer select signal

Multiplexer (MUX):

- Inputs: All words in the data block
- Select: Word offset bits from address
- Output: Selected word

Example (2 words per block):

- Block contains: Word0 (bytes 0-3), Word1 (bytes 4-7)
- Word offset = 0 → Select Word0
- Word offset = 1 → Select Word1
- Need 1-bit select for 2:1 MUX

Example (4 words per block):

- Block contains: Word0, Word1, Word2, Word3
- Word offset = 2 bits → Select among 4 words
- Need 4:1 MUX

Timing:

- Data extraction and word selection happen in parallel with tag check
- Both combinational circuits
- Similar delays
- Can overlap operations

14.15.2.8 Step 8: DECISION BASED ON HIT/MISS

If HIT (signal = 1):

- Selected word is correct data
- Send word to CPU immediately
- Access complete
- Total time: Hit latency (< 1 ns)

If MISS (signal = 0):

- Selected word is WRONG data (different block or invalid)
- CANNOT send to CPU
- Must fetch correct block from main memory (DRAM)
- CPU must STALL (wait)
- Cache controller takes over

- Total time: Hit latency + Miss penalty

Miss Handling:

- Will discuss in next lecture
- Involves accessing main memory
- Bringing block into cache
- Potentially evicting old block
- Then serving CPU request

14.16 15. Cache Circuit Components Summary

14.16.1 15.1 Key Circuit Elements

14.16.1.1 1. INDEXING CIRCUITRY

- **Input:** Index bits from address
- **Function:** Decoder to select cache entry
- **Output:** Activates one cache row
- **Type:** Combinational logic
- **Delay:** Part of hit latency

14.16.1.2 2. TAG COMPARATOR

- **Input:** Stored tag, Address tag
- **Function:** Multi-bit equality check
- **Components:**
  - N XNOR gates (N = tag bit width)
  - 1 N-input AND gate
- **Output:** 1 if equal, 0 if not equal
- **Type:** Combinational logic
- **Delay:** Part of hit latency

14.16.1.3 3. VALID BIT CHECK

- **Input:** Valid bit from cache entry
- **Function:** Read and check validity
- **Output:** 1 if valid, 0 if invalid
- **Type:** Simple wire/buffer
- **Delay:** Minimal

14.16.1.4 4. HIT/MISS LOGIC

- **Input:** Tag match signal, Valid bit
- **Function:** AND gate
- **Output:** Hit/Miss signal
- **Type:** Combinational logic

- **Delay:** Single gate delay

14.16.1.5 5. DATA ARRAY ACCESS

- **Input:** Index bits
- **Function:** Read data block from cache
- **Output:** Multi-word data block
- **Type:** SRAM memory read
- **Delay:** SRAM access time (parallel with tag check)

14.16.1.6 6. WORD SELECTOR (Multiplexer)

- **Input:** Data block, Word offset bits
- **Function:** Select one word from block
- **Output:** Single word
- **Type:** MUX (combinational)
- **Delay:** MUX delay (parallel with tag check)

14.16.1.7 7. CONTROL LOGIC (Cache Controller)

- **Input:** Hit/Miss signal, Read/Write control
- **Function:** Decide next actions
- **Output:** Control signals for CPU, memory
- **On Hit:** Enable data to CPU
- **On Miss:** Initiate memory fetch, stall CPU
- **Type:** Sequential logic (state machine)

14.16.2 15.2 Hit Latency Components

Contributing Factors:

- Indexing delay
- Tag comparison delay
- Valid bit check delay
- Hit/Miss determination delay
- Word selection delay (parallel)
- Wire delays

Dominant Delays:

- Indexing (decoder)
- Tag comparator (XNOR + AND)
- These determine critical path

Parallelism:

- Tag check and data extraction happen simultaneously
- Reduces total hit latency
- Only one path delay counts (whichever is longer)

14.17 16. Next Lecture Preview

14.17.1 16.1 Topics to Cover

14.17.1.1 1. Cache Miss Handling

- What happens after miss is determined?
- How to fetch block from main memory?
- Where to place new block in cache?
- What to do if cache location occupied?

14.17.1.2 2. Cache Controller State Machine

- Not just combinational logic
- Sequential control needed for misses
- Multiple clock cycles to handle miss
- States: Idle, Compare Tags, Allocate, Write Back, etc.

14.17.1.3 3. Write Operations

- Read operation covered this lecture
- Write more complex: Must update cache AND memory
- Write policies: Write-through, Write-back
- Dirty bits for modified blocks

14.17.1.4 4. Replacement Policies

- When cache full, which block to evict?
- Least Recently Used (LRU)
- Other policies: FIFO, Random, LFU

14.17.1.5 5. Performance Analysis

- Calculate average access time
- Impact of hit rate, miss penalty
- Cache size vs. performance tradeoffs

14.17.1.6 6. Advanced Cache Concepts

- Set-associative caches (beyond direct-mapped)
- Multi-level caches (L1, L2, L3)
- Fully associative caches

Confirmed Future Class:

- Additional class scheduled for Friday at 9:00 AM
- Notice to be posted on course platform
- Continue cache memory topics

14.18 17. Key Takeaways and Summary

14.18.1 17.1 Historical Foundations

- Early computers had no memory/software concept
- Alan Turing conceived stored program computer (1936)
- John von Neumann implemented it in EDVAC (1948)
- Von Neumann architecture: Unified memory for instructions and data
- Harvard architecture: Separate instruction and data memories

14.18.2 17.2 Memory Technologies Hierarchy

Technology	Speed	Size	Cost
SRAM	Fastest (< 1 ns)	Smallest (KB-MB)	Most expensive ($2000/GB)
DRAM	Medium (~50 ns)	Medium (GB)	Moderate ($10/GB)
Flash	Similar to DRAM	Gigabytes	Cheap (< $1/GB)
Disk	Slowest (5-10 ms)	Largest (TB)	Cheapest (cents/GB)

14.18.3 17.3 The Performance Problem

- CPU cycle time: < 1 nanosecond
- Main memory cycle time: ~50 nanoseconds
- **Memory 50× slower than CPU!**
- Pipeline requires memory access in 1 cycle
- Cannot directly use DRAM for CPU memory accesses

14.18.4 17.4 Memory Hierarchy Solution

- Multiple levels: SRAM (cache) → DRAM → Disk
- CPU accesses only top level (cache)
- Upper levels hold subsets of lower levels
- Trick CPU: Fast as SRAM, large as Disk
- Requires very high hit rate (> 99.9%) at cache level

14.18.5 17.5 Principles of Locality

1. **Temporal Locality:** Recently accessed data likely accessed again soon
    - Example: Loop variables, instructions in loops
2. **Spatial Locality:** Data near recently accessed data likely accessed soon
    - Example: Array elements, sequential instructions
    - Both principles present in virtually all programs
    - Foundation for cache effectiveness

14.18.6 17.6 Memory Addressing

- **Byte Address:** Individual byte reference (full address)
- **Word Address:** 4-byte word reference (last 2 bits = 00 for alignment)
- **Block Address:** Multiple-word block reference (excludes offset bits)
- Address structure: [Block Address][Offset]
- Offset subdivides: [Word Offset][Byte in Word]

14.18.7 17.7 Cache Terminology

- **Hit:** Data found in cache → Fast access (< 1 ns)
- **Miss:** Data not in cache → Slow access (+ ~100 ns penalty)
- **Hit Rate:** Fraction of accesses that hit (want > 99.9%)
- **Miss Rate:** Fraction of accesses that miss (1 - Hit Rate)
- **Hit Latency:** Time to determine hit and access data
- **Miss Penalty:** EXTRA time to fetch from memory on miss

14.18.8 17.8 Cache Organization (Direct-Mapped)

- Each memory block maps to exactly ONE cache location
- Mapping: Cache Index = Block Address mod (Cache Size)
- Address fields: [Tag][Index][Offset]
- **Index:** Selects cache entry directly (no search!)
- **Tag:** Differentiates blocks mapping to same index
- **Offset:** Selects word/byte within block
- **Valid bit:** Indicates if entry contains valid data

14.18.9 17.9 Direct-Mapped Cache Structure

- **Tag array:** Stores tags for verification
- **Valid bit array:** Validity indicators
- **Data array:** Stores actual data blocks
- Index not stored (implicit in position)

14.18.10 17.10 Cache Read Access Process

1. Extract index from address → Access cache entry
2. Extract tag from cache entry → Compare with address tag
3. Check valid bit from entry
4. Determine hit/miss: (Tag Match) AND (Valid)
5. In parallel: Extract data block, select word using offset
6. If HIT: Send word to CPU (done in < 1 ns)
7. If MISS: Must fetch from memory (will cover next lecture)

14.18.11 17.11 Critical Requirements

- Hit latency must be < 1 CPU clock cycle

- Hit rate must be very high (> 99.9%)
- Only way to achieve: Exploit locality principles
- Direct mapping enables fast indexing (no search)
- Parallel tag check and data extraction minimize latency

14.18.12 17.12 Average Access Time Formula

```
Average Access Time = Hit Latency + (Miss Rate × Miss Penalty)
```

- Must keep Miss Rate very low for performance
- Even 1% miss rate catastrophic if penalty is 100×
- Example: 1% miss rate → 1 + (0.01 × 100) = 2 ns average
- Example: 0.1% miss rate → 1 + (0.001 × 100) = 1.1 ns average
- Target: 99.9% or better hit rate

14.18.13 17.13 Pending Topics (Next Lectures)

- Cache miss handling and memory fetch
- Cache controller state machine
- Write operations and write policies
- Block replacement strategies (LRU, etc.)
- Set-associative and fully associative caches
- Multi-level cache hierarchies
- Performance analysis and optimization

14.18.14 17.14 Music Library Analogy Summary

- **Phone (cache):** Small, fast, always accessible
- **Computer (main memory):** Larger, slower, main collection
- **Internet (disk):** Huge, slowest, everything available
- Listen from phone (CPU accesses cache)
- Copy from computer when song not on phone (fetch on miss)
- Download from internet when not on computer (fetch from disk)
- Keep favorite songs on phone (exploit temporal locality)
- Copy whole album at once (exploit spatial locality)

14.19 Key Takeaways

1. **Stored-program concept** revolutionized computing—programs stored in memory like data, eliminating manual reconfiguration for each algorithm.

2. **Von Neumann architecture** established fundamental computer organization—CPU, memory, and I/O with instructions and data sharing same memory.

3. **Processor-memory speed gap** creates performance bottleneck—CPU operates at nanosecond scale while main memory requires tens of nanoseconds.

4. **Memory hierarchy** provides illusion of large, fast memory—small fast cache near CPU, larger slower DRAM main memory, massive slow disk storage.
5. **Temporal locality**: Recently accessed data likely accessed again soon—programs exhibit loops, function calls, and repeated variable access patterns.
6. **Spatial locality**: Nearby data likely accessed soon—programs access arrays sequentially and instructions execute in order.
7. **Cache exploits locality** to achieve high hit rates—keeping frequently accessed data in fast storage dramatically improves average access time.
8. **Cache organized in blocks**, not individual words—exploiting spatial locality by fetching multiple words together.
9. **Direct-mapped cache**: Each memory block maps to exactly one cache location—simplest cache organization using modulo arithmetic for mapping.
10. **Address breakdown**: Tag + Index + Offset—index selects cache entry, tag identifies specific block, offset selects word within block.
11. **Valid bit** indicates cache entry contains meaningful data—essential for distinguishing real data from uninitialized entries at startup.
12. **Cache hit** occurs when requested data found in cache—CPU receives data in ~1 nanosecond, avoiding slow main memory access.
13. **Cache miss** requires main memory fetch—takes ~100 nanoseconds, replacing cache entry with new block from memory.
14. **Hit rate determines cache effectiveness**—even 1% miss rate significantly impacts average memory access time with 100× penalty.
15. **Block size affects performance**—larger blocks exploit spatial locality better but reduce total number of blocks, potentially increasing conflicts.
16. **Cache size** represents total data storage capacity—typical L1 caches 32-64 KB, L2 caches 256 KB-1 MB.
17. **Tag comparison** happens in parallel with data access—enabling fast hit detection and maintaining single-cycle cache access.
18. **Music library analogy** clarifies cache concept—phone (cache) holds favorites, computer (DRAM) has main collection, internet (disk) contains everything.
19. **Cache transparent to programmer**—software sees uniform memory, hardware manages cache automatically for best performance.
20. **Memory hierarchy only works because programs exhibit locality**—without temporal and spatial locality, caching would fail catastrophically.

14.20 Summary

The introduction to memory systems and cache memory reveals how the fundamental processor-memory speed gap—with CPUs operating 100× faster than main memory—

drives sophisticated cache hierarchy designs that create the illusion of large, fast memory. Historical context from Alan Turing's theoretical foundations through Von Neumann's stored-program architecture establishes how modern computers execute instructions fetched from memory rather than requiring manual reconfiguration. The memory hierarchy concept, with small fast SRAM caches near the CPU, larger slower DRAM main memory, and massive disk storage, exploits two fundamental program properties: temporal locality (recently accessed data likely accessed again soon) and spatial locality (nearby data likely accessed soon). Cache memory, organized in blocks rather than individual words, dramatically improves average access time by maintaining frequently accessed data in fast storage, achieving hit rates often exceeding 95% in practice. Direct-mapped cache organization, the simplest mapping scheme, uses modulo arithmetic to assign each memory block to exactly one cache location, with address bits divided into tag (identifying specific block), index (selecting cache entry), and offset (choosing word within block). The valid bit distinguishes real cached data from uninitialized entries, essential at system startup when cache contains random values. Cache hits deliver data in approximately 1 nanosecond while misses require ~100 nanosecond main memory access, making even small miss rates significant—a 1% miss rate doubles average access time from 1 ns to 2 ns. The music library analogy effectively clarifies concepts: phone storage represents cache (small, fast, always accessible), computer storage represents main memory (larger, slower, main collection), and internet streaming represents disk (unlimited, very slow, backup). This cache transparency—programmer sees uniform memory while hardware automatically manages caching—enables software compatibility across different cache configurations. The critical insight remains that memory hierarchy effectiveness depends entirely on programs exhibiting locality; without these natural access patterns inherent to how we write code, caching would provide no benefit. Understanding cache fundamentals proves essential for both hardware designers optimizing cache architectures and software developers writing cache-friendly code that maximizes hit rates.

Chapter 15

Lecture 15: Cache Memory Operations – Read/Write Access and Write Policies

15.1 Introduction

This lecture provides a comprehensive, step□by□step examination of how a direct□mapped cache services read and write requests, differentiates hits from misses, and preserves data correctness. We finish the full read path (including stall + block fetch sequence), analyze write hits and misses, and introduce the write□through policy as the simplest consistency mechanism between cache and main memory. Performance consequences of constant memory writes, the need for high hit rates, and the motivation for more advanced write□back policies (next lecture) are emphasized. By the end you will understand exactly what the cache controller must do (state transitions, signals, data/tag/valid updates) for every access type and why write policies are a central architectural tradeoff.

15.2 1. Lecture Introduction and Recap

15.2.1 1.1 Previous Lecture Review

15.2.1.1 Memory Systems Foundation

- Memory hierarchy concept (SRAM → DRAM → Disk)
- Illusion of large and fast memory simultaneously
- CPU accesses only cache (top level)

15.2.1.2 Locality Principles

- **Temporal locality:** Recently accessed data likely accessed again soon
- **Spatial locality:** Nearby data likely accessed soon

- Foundation for cache effectiveness

15.2.1.3 Direct-Mapped Cache Introduction

- Each memory block maps to exactly ONE cache location
- Mapping function: Cache Index = Block Address MOD Cache Size
- Read access process partially covered

15.2.1.4 Cache Structure (Recap)

- **Data array:** Stores data blocks (not individual words)
- **Tag array:** Stores tags for block identification
- **Valid bit array:** Indicates valid/invalid entries
- **Index:** Not stored, implicit in position (for convenience in diagrams)

15.2.1.5 Address Breakdown (Recap)

```
[Tag][Index][Offset]
  ^      ^       ^
  |      |          Identifies word/byte within block
  |         Identifies cache entry (direct mapping)
    Remaining bits for block identification
```

15.2.2 1.2 Today's Focus

- Complete discussion of read miss handling
- Write access operations (hit and miss)
- Write policies and their implications
- Data consistency issues
- Performance considerations

15.3 2. Cache Read Access - Complete Process

15.3.1 2.1 Read Access Input Signals

From CPU to Cache Controller:

1. **Address** (word or byte address)
2. **Read Control Signal** (from CPU control unit)
    - Indicates this is a read operation (not write)
    - Part of memory control signals

15.3.2 2.2 Cache Read Steps (Detailed)

15.3.2.1 Step 1: Address Decomposition

- Parse incoming address into three fields:
  - **Tag:** For verification
  - **Index:** For cache entry selection
  - **Offset:** For word/byte selection within block

15.3.2.2 Step 2: Cache Entry Selection (Indexing)

- Extract index bits from address
- Cache controller knows which bits are index (by design)
- Use demultiplexer circuitry to access correct cache entry
- Example: Index = 101 (binary) → Access cache entry 5
- Direct access, no search needed
- Combinational logic (fast)

15.3.2.3 Step 3: Tag Comparison

- Extract stored tag from selected cache entry
- Extract tag from incoming address
- Use comparator circuit (XNOR gates + AND gate)
- Output: 1 if tags match, 0 if tags differ

15.3.2.4 Step 4: Valid Bit Check

- Extract valid bit from selected cache entry
- Check if entry contains valid data
- Output: 1 if valid, 0 if invalid

15.3.2.5 Step 5: Hit/Miss Determination

- Logic: **Hit = (Tag Match) AND (Valid Bit)**
- If both conditions true → HIT
- If either condition false → MISS
- Single AND gate combines both signals

15.3.2.6 Step 6: Data Extraction (Parallel Operation)

- Happens simultaneously with tag comparison
- Extract entire data block from cache entry
- Place data block on internal wires
- Example: 8-byte block = 2 words

15.3.2.7 Step 7: Word Selection (Using Offset)

- CPU requests a single WORD
- Use word offset bits as MUX select signal

- Example: 2 words in block
  - Offset MSB = 0 → Select first word
  - Offset MSB = 1 → Select second word
- Multiplexer extracts correct word from block

15.3.3 2.3 Timing Optimization

Parallel Operations:

- Tag comparison (Steps 3-5) and data extraction (Steps 6-7) happen in PARALLEL
- Both are combinational circuits
- Total delay = max(tag comparison delay, data extraction delay)
- Reduces overall hit latency

15.3.4 2.4 Read Hit Outcome

- Selected word is correct data
- Send word to CPU immediately
- No stall required
- Total time: Hit latency (< 1 nanosecond for SRAM)
- Completes within one CPU clock cycle
- Pipeline continues uninterrupted

15.3.5 2.5 Pipeline Integration

- In MIPS pipeline, MEM stage accesses memory
- With cache hit: Memory access completes in 1 cycle
- Pipeline maintains smooth operation
- No bubbles inserted

15.4 3. Cache Read Miss Handling

15.4.1 3.1 Read Miss Scenario

15.4.1.1 Miss Conditions

1. **Tag mismatch** (most common)
   - Requested block not in cache
   - Different block occupies that cache location
2. **Invalid entry**
   - Valid bit = 0
   - Entry contains no valid data (e.g., after initialization)
3. **Both conditions**
   - Tag mismatch AND invalid entry

15.4.2 3.2 Read Miss Response Required Actions

15.4.2.1 Action 1: STALL THE CPU

Process:

- CPU cannot proceed without requested data
- Data hazard would occur if CPU continues
- Cache controller sends STALL signal to CPU
- CPU must monitor stall signal continuously
- When stall signal high → Freeze CPU operation
  - Stop fetching new instructions
  - Freeze all pipeline stages
  - Hold current state

CPU's Perspective:

- CPU doesn't know cache and memory are separate
- CPU sees memory hierarchy as single "memory"
- Must respond to stall signal from memory subsystem
- In MEM stage: Check and respond to stall signal

15.4.2.2 Action 2: MAKE READ REQUEST TO MAIN MEMORY

Request Details:

- Request the missing DATA BLOCK (not just word!)
- Cache and memory trade in BLOCKS
- CPU trades in words/bytes, but cache-memory interface uses blocks
- Send block address to main memory
- Memory fetches entire block

Reason for Block Transfer:

- Exploits spatial locality
- Fetches requested word AND nearby words
- Reduces future misses for nearby addresses
- More efficient than fetching single words

Memory Access Time:

- DRAM access: Several CPU clock cycles
- Range: 10 to 100+ CPU clock cycles
- Much slower than cache (< 1 cycle)
- This is the **MISS PENALTY**

15.4.2.3 Action 3: WAIT FOR MEMORY RESPONSE

- Memory performs read operation
- Data travels from memory to cache
- Controller waits (CPU still stalled)
- Multiple clock cycles elapse

15.4.2.4 Action 4: UPDATE CACHE ENTRY

Three components to update:

a) Update Data Block:

- Write fetched block into cache entry
- Replace old data at that index

b) Update Tag:

- Extract tag from block address
- Write tag into tag array at that index
- Ensures future tag comparisons work correctly

c) Set Valid Bit:

- Set valid bit to 1
- Denotes entry now contains valid data

15.4.2.5 Action 5: SEND DATA TO CPU

- Extract requested word from newly loaded block
- Use offset to select correct word
- Put data on bus to CPU
- CPU receives requested data

15.4.2.6 Action 6: CLEAR STALL SIGNAL

- Cache controller clears (lowers) stall signal
- CPU detects stall signal going low
- CPU resumes operation
- Pipeline unfreezes and continues

15.4.3 3.3 Total Read Miss Time

Formula:

```
Read Miss Time = Hit Latency + Miss Penalty
```

Where:

- **Hit Latency:** Time to determine it's a miss (< 1 ns)
- **Miss Penalty:** Time to fetch from memory (10-100+ CPU cycles)

Example Calculation:

- Hit latency: 1 ns (1 cycle at 1 GHz)
- Miss penalty: 50 ns (50 cycles at 1 GHz)
- Total: 1 + 50 = 51 cycles

15.4.4 3.4 Performance Impact

- Single miss causes 50+ cycle stall
- Catastrophic for pipeline performance
- Emphasizes need for high hit rate (> 99.9%)

15.4.5 3.5 Question: What About the Old Block?

The Deferred Question:

- When fetching new block on miss
- Old block occupies that cache entry
- What happens to old block?
- Is it okay to discard it?

Initial Answer: "We'll discuss after introducing write policies"

- Answer depends on write policy
- Need to understand writes first
- Question will be revisited

15.5 4. Cache Write Access - Introduction

15.5.1 4.1 Write Access Input Signals

From CPU to Cache Controller:

1. **Address** (where to write)
2. **Data Word** (what to write)
3. **Write Control Signal** (indicates write operation)

Three inputs vs. two for read (no data input needed for read).

15.5.2 4.2 Write Access Process

15.5.2.1 Step 1: Address Decomposition

- Same as read: [Tag][Index][Offset]

15.5.2.2 Step 2: Cache Entry Selection

- Same as read: Use index bits
- Demultiplexer accesses correct entry
- Direct access based on index
- Example: Index 101 → Entry 5

15.5.2.3 Step 3: Tag Comparison

- Extract tag from cache entry
- Compare with incoming address tag
- Comparator circuit (same as read)
- Output: Match or no match

15.5.2.4 Step 4: Valid Bit Check

- Extract and check valid bit
- Same as read operation
- Ensures entry is valid

15.5.2.5 Step 5: Hit/Miss Determination

- Hit = (Tag Match) AND (Valid Bit)
- Same logic as read
- Determines write hit or write miss

15.5.2.6 Step 6: Data Writing (The Difference)

This is where write differs from read:

- Must write data word to correct location in block
- Use offset to determine which word in block

15.5.3 4.3 Writing Mechanism

Input:

- Incoming data word (from CPU)
- Offset bits from address

Demultiplexer Selection:

- Use word offset as demultiplexer select signal
- Example with 2 words per block:
  - Word offset = 0 → Write to first word
  - Word offset = 1 → Write to second word
- Demultiplexer directs data to correct word position

Example:

- Block has 2 words: Word0 (bytes 0-3), Word1 (bytes 4-7)
- Incoming data word: 0x12345678
- Offset MSB = 1 → Select Word1
- Demux directs data to Word1 position in block

Write Operation Control:

- Writing controlled by Write control signal from CPU
- Only write if signal indicates write operation

- Demultiplexer enabled by write signal

15.5.4 4.4 Critical Question: Can Write and Tag Compare Happen in Parallel?

15.5.4.1 For Read (Previous Discussion)

- **YES, both can happen in parallel**
- If miss, discard extracted data (no harm done)
- Reading doesn't change cache state

15.5.4.2 For Write (Current Question)

More problematic!

- What if we write and then discover tag mismatch?

Scenario:

- Write to cache entry simultaneously with tag comparison
- Tag comparison returns MISMATCH
- We've now CORRUPTED data in cache!
- Written to wrong block (different tag)
- Data integrity violated

Problem:

- If invalid entry: Not too serious (data was garbage anyway)
- If tag mismatch: **SERIOUS problem!**
  - Overwrote valid data for different block
  - That block's data now corrupted
  - Future accesses to that block get wrong data

Initial Conclusion:

- Cannot safely write and tag compare in parallel
- Need mechanism to prevent corruption
- Solution depends on write policy (discussed next)

15.6 5. Write Policies - Introduction

15.6.1 5.1 The Data Consistency Problem

Scenario:

- CPU writes to address A
- Address A hits in cache
- Cache controller writes new value to cache entry
- Cache now has updated value
- Main memory still has OLD value

- Two versions exist: Cache version ≠ Memory version

The Inconsistency:

- Cache entry now INCONSISTENT with main memory
- Same address has different values in different levels
- Data coherence problem

15.6.2 5.2 Why This Matters

- Future access to same address: Which value is correct?
- If cache entry replaced: New value lost
- I/O devices may access memory directly (bypass cache)
- Multi-processor systems: Other CPUs access memory
- Must maintain data consistency across hierarchy

15.6.3 5.3 Two Fundamental Write Policies

1. **Write-Through** (discussed this lecture)
2. **Write-Back** (mentioned, detailed in next lecture)

15.7 6. Write-Through Policy

15.7.1 6.1 Write-Through Definition

Policy Statement:

> "Always write to BOTH cache AND memory"

Mechanism:

- On every write operation:
    1. Write to cache (if hit)
    2. Simultaneously write to main memory
- Both levels updated together
- Ensures cache and memory always consistent

15.7.2 6.2 Write-Through Process

15.7.2.1 Write Hit with Write-Through

1. Determine it's a write hit (tag match + valid)
2. Write data word to cache block (using offset)
3. Also send write request to main memory
4. Update same address in memory
5. Wait for memory write to complete
6. Both cache and memory now have same value

15.7.2.2 Write Miss with Write-Through

1. Determine it’s a write miss
2. Stall CPU
3. Fetch missing block from memory (read operation)
4. Update cache entry with fetched block
5. Write the word to correct position in block
6. Also write to memory
7. Clear stall signal
8. Both levels updated

15.7.3 6.3 Advantages of Write-Through

15.7.3.1 Advantage 1: SIMPLICITY

- Straightforward to implement
- No complex consistency protocols
- Cache controller logic simpler
- Design principle: Keep cache simple

15.7.3.2 Advantage 2: CONSISTENCY GUARANTEED

- Cache and memory ALWAYS have same values
- No special handling for discarded blocks
- Can replace any cache entry anytime
- Memory always has correct, up-to-date data

15.7.3.3 Advantage 3: ANSWERS THE OLD BLOCK QUESTION

With write-through policy:

- Old block can be safely discarded
- All updates were written to memory
- Memory has latest version
- Future accesses can fetch from memory
- No data loss

Comparison:

- Read miss: Old block discarded, data available in memory
- Write with write-through: Always updated memory, safe to discard

15.7.3.4 Advantage 4: PARALLEL WRITE AND TAG COMPARE NOW POSSIBLE!

Critical Insight: Can now overlap write and tag comparison. Why? Two scenarios:

Scenario A: Write Hit

- Written to cache, will also write to memory
- Tag matches, write is correct
- Both cache and memory updated

- No problem

Scenario B: Write Miss

- Written to cache entry (possibly wrong block)
- Tag mismatch detected
- Will fetch correct block from memory anyway
- Will overwrite cache entry with correct block
- Corrupted data gets replaced immediately
- Memory has correct version (wasn't corrupted)
- No lasting damage

Result:

- Safe to write and tag compare in parallel
- Saves time (hit latency reduced)
- Both operations in same clock cycle
- If hit: Saved time
- If miss: No harm (will fix cache anyway)

Timing Optimization:

- Tag comparison time: T_comp
- Write time: T_write
- Without overlap: Total = T_comp + T_write
- With overlap: Total = max(T_comp, T_write)
- Typically similar delays → Nearly 2× speedup

15.7.4 6.4 Disadvantages of Write-Through

15.7.4.1 Disadvantage 1: EXCESSIVE WRITE TRAFFIC

- EVERY write goes to memory
- Memory writes are slow (10-100+ cycles)
- Generates continuous memory traffic
- Memory bus congestion

15.7.4.2 Disadvantage 2: CPU STALLS ON EVERY WRITE

Critical Problem:

- Every write requires memory access
- Memory much slower than cache
- CPU must stall for EVERY write
- Wait for memory write to complete

Stall Duration:

- Memory write: 10-100 CPU clock cycles
- Every store instruction causes stall
- Even on write HIT!

Example:

- Store instruction hits in cache
- Still must wait for memory write
- 50 cycle stall for every store
- Pipeline essentially stops

Impact on Programs with Many Writes:

- Programs with frequent store instructions
- Array updates, structure modifications
- Loop counters being updated
- String manipulation
- All suffer severe performance degradation

Performance Comparison:

- Read hit: < 1 cycle (fast!)
- Write hit with write-through: 50+ cycles (slow!)
- Asymmetry: Reads fast, writes catastrophically slow

Pipeline Impact:

- Recall pipelining lectures: Minimized stalls
- Worked hard to avoid 1-2 cycle stalls
- Write-through introduces 50+ cycle stalls regularly
- Contradicts pipeline optimization goals
- “Doesn’t add up” - unacceptable performance loss

Real-World Issue:

- Write-through used in some systems
- But with additional optimizations (write buffers, discussed later)
- Pure write-through too slow for modern systems

15.7.4.3 Disadvantage 3: POWER CONSUMPTION

- Memory accesses consume power
- Every write → Memory access → Power consumption
- Unnecessary power usage
- Critical for mobile/embedded systems

15.7.4.4 Disadvantage 4: MEMORY WEAR

- Flash memory: Limited write cycles
- SSDs wear out with writes
- Write-through accelerates wear
- Reduces memory lifespan

15.8 7. Resolving the Old Block Question

15.8.1 7.1 The Question Revisited

Original Question:

> "What happens to the old block when we fetch a new block from memory on a miss?"

Context:

- Read or write miss occurs
- Need to fetch missing block from memory
- Old block occupies target cache entry
- Must replace old block with new block
- Is it safe to discard old block?

15.8.2 7.2 Answer with Write-Through Policy

YES, Safe to Discard

15.8.2.1 Reason 1: Memory Has Updated Version

- Write-through ensures every write goes to memory
- All modifications reflected in memory
- Memory always has latest version of all blocks
- Old block's latest state is in memory

15.8.2.2 Reason 2: Can Re-fetch If Needed

- Future access to old block's address
- Will miss in cache (block was replaced)
- Can fetch from memory again
- Memory has correct, up-to-date data
- No data loss

15.8.3 7.3 Example Scenario

1. Block A in cache at index 3
2. Block A modified several times
3. Each modification written to cache AND memory
4. Block B (also maps to index 3) is requested
5. Miss occurs for Block B
6. Fetch Block B from memory
7. Replace Block A with Block B at index 3
8. Block A discarded from cache
9. Block A's data safe in memory
10. Later access to Block A: Miss, fetch from memory again

15.8.4 7.4 Comparison with Invalid Entry

- If miss due to invalid bit: Obviously safe to replace
- If miss due to tag mismatch: Safe because of write-through

15.8.5 7.5 Contrast with Future Policy (Teaser)

- With other write policies (write-back), answer may differ
- May NOT be safe to discard old block
- Will discuss in next lecture

Conclusion:

- Write-through simplifies replacement
- No special checks needed before replacing block
- Always safe to overwrite cache entry
- Memory serves as reliable backup

15.9 8. Parallelism in Write Access with Write-Through

15.9.1 8.1 The Parallel Write Problem Solved

Original Concern:

- Want to overlap write operation and tag comparison
- Reduce hit latency
- But risk corrupting data if tag mismatch

15.9.2 8.2 With Write-Through Policy

15.9.2.1 Case 1: Write Hit

- Write to cache and tag compare happen in parallel
- Tag matches → It was a hit
- Cache entry correctly updated
- Also write to memory (per write-through policy)
- Both cache and memory consistent
- Time saved: One cycle
- No problem!

15.9.2.2 Case 2: Write Miss

- Write to cache and tag compare happen in parallel
- Tag doesn't match → It was a miss
- Cache entry might be corrupted (wrote to wrong block)
- **BUT:** About to fetch correct block from memory
- Will OVERWRITE this cache entry with new block

- Corrupted data disappears immediately
- Also, write goes to memory (correct address in memory)
- End result: Cache fixed, memory correct

15.9.3 8.3 Key Insight

- Write-through to memory preserves correctness
- Memory write goes to CORRECT address (from address bus)
- Even if cache entry temporarily corrupted
- Cache entry will be fixed when correct block loaded
- Memory never corrupted

15.9.4 8.4 Timeline for Write Miss

```
Cycle 1: Write to cache (possibly wrong block) + Tag compare
Cycle 1: Also initiate memory write (correct address)
Cycle 2-50: Fetch correct block from memory
Cycle 51: Overwrite cache entry with correct block
Result: Cache correct, memory correct
```

15.9.5 8.5 Safety Guarantee

- **Memory write:** Targets address from address bus (always correct)
- **Cache write:** Targets index (might be for different block)
- **If miss:** Cache mistake corrected by fetch
- **If hit:** No mistake, everything correct
- **In both cases:** End state correct

15.9.6 8.6 Performance Benefit

- Saved cycles on write hit path
- Write and tag compare: Parallel instead of sequential
- Approximately 2× faster hit determination
- Critical for frequent write hits

15.9.7 8.7 Enabled by Write-Through

- Only possible because memory updated on every write
- Other policies may not allow this optimization
- Write-through sacrifices write performance for simplicity
- But enables some optimizations

15.10 9. Summary of Cache Operations

15.10.1 9.1 Complete Cache Operation Overview

15.10.1.1 READ HIT

- Index → Tag compare + Valid check → Match
- Extract data block → Select word → Send to CPU
- Time: < 1 cycle (hit latency only)
- No stall
- Pipeline continues

15.10.1.2 READ MISS

- Index → Tag compare + Valid check → No match
- Stall CPU
- Fetch block from memory (10-100+ cycles)
- Update cache: Data + Tag + Valid bit
- Extract word → Send to CPU
- Clear stall
- Time: Hit latency + Miss penalty
- Major pipeline disruption

15.10.1.3 WRITE HIT (with Write-Through)

- Index → Tag compare + Valid check (parallel with write)
- Write word to cache block
- Also write to memory (10-100+ cycles)
- Stall CPU until memory write completes
- Time: Hit latency + Memory write time
- Slower than read hit!

15.10.1.4 WRITE MISS (with Write-Through)

- Index → Tag compare + Valid check → No match
- Stall CPU
- Fetch block from memory
- Update cache: Data + Tag + Valid bit
- Write word to cache block
- Also write to memory
- Clear stall
- Time: Hit latency + Miss penalty + Memory write time
- Even slower than read miss!

15.10.2 9.2 Performance Characteristics

Case	Time	Comment
Best Case (Read Hit)	< 1 cycle	Optimal performance. Want this to be most common case
Moderate Case (Read Miss)	50+ cycles	Acceptable if infrequent. Reason for high hit rate requirement
Poor Case (Write Hit with Write-Through)	50+ cycles	Every write hits this case. Unacceptable for write-heavy programs
Worst Case (Write Miss with Write-Through)	100+ cycles	Rare but extremely slow. Catastrophic when occurs

Performance Goal:

- Maximize read hits
- Minimize write impact (better policy needed)
- Overall hit rate > 99.9%

15.11 10. Write-Through Policy Evaluation

15.11.1 10.1 Summary of Write-Through

Mechanism:

- Write to cache (if hit) AND memory
- Always keep both consistent
- Memory is authoritative backup

Implementation Complexity:

- Simple cache controller logic
- No complex state tracking
- Straightforward consistency maintenance

15.11.2 10.2 Advantages

Advantage	Description
1. Simplicity	Easy to understand, simple to implement, minimal controller complexity, aligns with design principle (simple cache)
2. Consistency	Cache and memory always consistent, no special synchronization needed, can discard blocks anytime, memory always reliable
3. Data Safety	No data loss on block replacement, memory has all updates, crash recovery simpler, I/O devices see correct data
4. Enables Optimizations	Can overlap write and tag compare, reduces hit latency, safe due to memory backup

15.11.3 10.3 Disadvantages

Disadvantage	Description
1. Performance Penalty	Every write stalls CPU, 10-100+ cycle stalls per write, unacceptable for write-intensive programs, contradicts pipeline optimization goals
2. Memory Traffic	Excessive write traffic to memory, memory bus congestion, reduces available bandwidth for read misses, slows down entire system
3. Power Consumption	Every write powers up memory, unnecessary power usage, battery drain in mobile devices, heat generation
4. Memory Wear	Flash/SSD: Limited write cycles, accelerated wear-out, reduced memory lifespan, particularly bad for SSDs

15.11.4 10.4 When Write-Through Used

15.11.4.1 Suitable Applications

- Read-heavy workloads
- Simple embedded systems
- Systems requiring guaranteed consistency
- Safety-critical applications

15.11.4.2 Real-World Usage

- Often combined with write buffers
- Write buffer: Small queue for pending writes
- CPU continues after writing to buffer
- Buffer drains to memory in background
- Reduces stall impact (will discuss if time permits)

15.11.4.3 Modern Systems

- Pure write-through rarely used alone
- Too slow for general-purpose computing
- Alternative: Write-back policy (next lecture)
- Trade complexity for performance

15.12 11. The Need for Alternative Write Policies

15.12.1 11.1 The Performance Problem

15.12.1.1 Write-Heavy Programs

Many programming patterns involve frequent writes:

- Array updates in loops
- Data structure modifications
- Counter increments
- Accumulator updates
- String/buffer operations

Example Code:

```
for (int i = 0; i < 1000; i++) {
    array[i] = compute(i);  // Store in every iteration
    sum += array[i];         // Read, accumulate, store
}
```

15.12.1.2 With Write-Through

- Loop iterations: 1000
- Stores per iteration: 2 (array[i], sum)
- Total stores: 2000
- Cycles per store: 50 (memory write)
- **Total stall cycles: 100,000!**
- Versus computation cycles: Maybe 10,000
- **Performance: 10× slower than necessary!**

15.12.2 11.2 Pipeline Impact

- Pipelining designed to execute 1 instruction/cycle (ideal)
- Write-through: 50 cycles per store instruction
- Pipeline utilization: ~2% (1/50)
- Completely defeats pipelining benefits

15.12.3 11.3 Comparison with Read Operations

Operation	Time	Frequency	Acceptability
Read hit	< 1 cycle	Common	Fast
Read miss	50 cycles	Rare	Acceptable
Write hit	50 cycles	Frequent	**Unacceptable**
Write miss	100+ cycles	Rare	Terrible

15.12.4 11.4 The Contradiction

- Spent lectures optimizing pipeline
- Minimized hazards, used forwarding, prediction
- Eliminated 1-2 cycle stalls
- Now introducing 50+ cycle stalls on every write!
- “Doesn’t add up” - need better solution

15.12.5 11.5 Question Raised

“What can we do to avoid this situation?”

Student Insight:

> “We can write to memory only when we want to replace that cache block with different data”

Instructor Response:

> “Exactly! That becomes a different write policy.”

15.12.6 11.6 Teaser for Next Lecture

- Alternative policy: **Write-Back**
- Write to cache only, not memory immediately
- Write to memory only when necessary
- Much better performance
- Added complexity in return
- Will discuss in detail next class

15.13 12. Lecture Conclusion

15.13.1 12.1 Topics Covered

15.13.1.1 1. Complete Read Access Process

- Index → Tag compare → Valid check → Hit/Miss
- Parallel data extraction and word selection
- Hit: Send data immediately
- Miss: Fetch from memory, stall CPU

15.13.1.2 2. Read Miss Handling

Six-step process:

1. Stall CPU
2. Request block from memory
3. Wait for response
4. Update cache entry (data, tag, valid)
5. Send data to CPU
6. Clear stall

- Miss penalty: 10-100+ cycles

15.13.1.3 3. Write Access Process

- Similar to read: Index → Tag compare → Valid check

- Difference: Must write data to cache
- Use demultiplexer to direct data to correct word

15.13.1.4 4. Data Consistency Problem

- Writing to cache creates inconsistency
- Cache has new value, memory has old value
- Need policy to maintain consistency

15.13.1.5 5. Write-Through Policy

- Write to both cache and memory on every write
- Advantages: Simple, consistent, safe
- Disadvantages: Slow, excessive traffic, poor performance

15.13.1.6 6. Old Block Question Resolved

- With write-through: Safe to discard
- Memory has updated version
- Can re-fetch if needed later

15.13.1.7 7. Parallel Write Optimization

- Can overlap write and tag compare
- Write-through makes this safe
- Reduces hit latency

15.13.1.8 8. Performance Issues

- Write-through too slow for write-intensive programs
- Every write causes long stall
- Need better policy

15.13.2 12.2 Next Lecture Preview

Topics to Cover:

- Write-Back policy (delayed writes)
- Dirty bit concept
- When to write back to memory
- Performance improvements
- Complexity tradeoffs
- Block replacement with write-back
- Comparison: Write-through vs. Write-back
- Real-world cache designs

Implementation Details:

- Write buffer optimization for write-through

- Handling dirty blocks on replacement
- Write-back state machine
- Performance analysis

Advanced Topics (if time):

- Write-allocate vs. no-write-allocate
- Write-combining
- Victim caches
- Multi-level caches with different policies

The Goal:

- Understand tradeoffs between simplicity and performance
- Choose appropriate policy for application
- Design efficient cache systems

Key Insight: Write-through sacrifices performance for simplicity. In modern systems, performance is critical, so more complex policies are necessary despite added complexity.

15.14 Key Takeaways

1. **Cache read hit** completes in single cycle—tag match and valid bit set indicate data available immediately from cache.

2. **Cache read miss** requires multiple cycles—must fetch entire block from main memory, update cache entry, set valid bit, then retry access.

3. **Cache controller** implements state machine—managing transitions between idle, compare tags, fetch block, and write cache states.

4. **Tag comparison** determines hit/miss—stored tag must match address tag AND valid bit must be set for successful hit.

5. **Block fetch** retrieves entire block from memory—exploiting spatial locality by bringing multiple words that will likely be accessed soon.

6. **Valid bit initialization** crucial at startup—all valid bits cleared to zero, preventing false hits on random cache data.

7. **Write operations** complicate cache design—must maintain consistency between cache and main memory through careful policy choices.

8. **Write-through policy** updates both cache and memory on every write—simple consistency but severe performance penalty.

9. **Write-through advantages**: Simple implementation, main memory always current, no dirty bit needed, straightforward crash recovery.

10. **Write-through disadvantages**: Every write causes slow memory access (~100 ns), dramatically reduces performance, wastes memory bandwidth.

11. **Write buffers** partially mitigate write-through penalty—CPU writes to buffer and continues, buffer writes to memory asynchronously.
12. **Write buffer depth** typically 4-8 entries—balances performance improvement against hardware cost and complexity.
13. **Write buffer full** forces CPU stall—occurs during write-intensive code sections, limiting write-through effectiveness.
14. **Write miss policies** determine cache behavior—write-allocate (fetch block first) versus no-write-allocate (write directly to memory).
15. **Write-allocate** exploits temporal locality—if just written location likely accessed again soon, fetching to cache improves future performance.
16. **No-write-allocate** avoids fetch overhead—appropriate when written locations unlikely to be accessed soon.
17. **Policy combinations** affect overall performance—write-through typically paired with no-write-allocate for consistency.
18. **Cache consistency** means cache and memory agree on data values—critical correctness requirement across all cache operations.
19. **Performance impact** of write policies substantial—write-through can increase memory traffic by 15-20% in typical programs.
20. **Write-back policy** introduced as superior alternative—defers memory writes until block eviction, dramatically reducing memory traffic.

15.15 Summary

Detailed examination of cache memory operations reveals the sophisticated control logic required to manage read and write accesses while maintaining data consistency between cache and main memory. Read operations follow straightforward paths: hits deliver data in single cycle via tag comparison confirming both tag match and valid bit set, while misses trigger multi-cycle sequences fetching entire blocks from main memory, updating cache entries, setting valid bits, and retrying accesses. The cache controller implements these sequences through state machine logic managing transitions between idle, tag comparison, block fetching, and cache writing states. Write operations introduce significant complexity and performance implications through policy choices determining how cache and memory stay synchronized. Write-through policy, updating both cache and memory on every write, offers simplicity and guaranteed consistency—main memory always reflects current data state, enabling straightforward crash recovery and multi-processor coherence. However, write-through's performance penalty proves severe: every write operation incurs ~100 nanosecond memory access delay, effectively eliminating cache benefit for write-heavy code sections and wasting substantial memory bandwidth on updates. Write buffers provide partial mitigation by decoupling CPU from memory write delays, allowing processors to write to small hardware queues and continue execution while buffer contents asynchronously propagate to main memory. Typical write buffers holding 4-8 entries balance performance improvement against hardware cost, though write-intensive code can still fill buffers and force CPU stalls. Write miss policies—write-allocate (fetch block before writing) versus no-write-allocate (write directly to memory)—represent additional design choices affecting performance based on program access patterns. Write-allocate exploits temporal locality, benefiting code that writes then soon reads same locations, while no-write-allocate avoids fetch overhead for write-once scenarios. Write-through typically pairs with no-write-allocate for policy consistency. The fundamental limitation—that write-through forces memory access on every write regardless of whether data will be accessed again—motivates write-back policies introduced in subsequent lectures, which defer memory writes until block eviction and thereby dramatically reduce memory traffic. Understanding these operational details and policy tradeoffs proves essential for appreciating how real cache implementations balance performance, complexity, consistency, and correctness requirements in practical computer systems.

Chapter 16

Lecture 16: Cache Hierarchies and Real Implementations

16.1 Introduction

This lecture explores cache hierarchies in modern computer systems, examining how multiple levels of cache work together to optimize memory access performance through careful balance of hit latency versus hit rate. We analyze real-world implementations including Intel's Skylake architecture, understanding the design decisions behind multi-level cache organizations where L1 caches prioritize speed, L2 caches balance capacity and latency, and L3 caches provide large shared storage across processor cores. The examination of associativity tradeoffs—from direct-mapped through set-associative to fully associative designs—reveals how hardware complexity, power consumption, and performance interact in practical cache systems.

16.2 1. Recap: Associativity Comparison Results

From the previous lecture's example using a 4-block cache with three different organizations:

16.2.1 Direct Mapped Cache

- **Result**: 5 misses, 0 hits
- **Cold misses**: 3 (compulsory, unavoidable)
- **Conflict misses**: 2 (data evicted then accessed again)
- **Utilization**: Poor - only 2 of 4 slots used
- **Hit rate**: 0% in this example

16.2.2 2-Way Set Associative Cache

- **Result**: 4 misses, 1 hit

- **Cold misses**: 3
- **Conflict misses**: 1
- **Utilization**: 2 of 4 slots used
- **Hit rate**: 20% - better than direct mapped

16.2.3 Fully Associative Cache (4-way)

- **Result**: 3 misses, 2 hits
- **Cold misses**: 3 (only unavoidable misses)
- **Conflict misses**: 0
- **Utilization**: Best - 3 of 4 slots used
- **Hit rate**: 40% - best performance

16.2.4 Key Observations

- Higher associativity → better hit rate
- Higher associativity → reduced conflict misses
- Cold misses occur at program start and when new addresses are accessed
- System reaches “steady state” with mostly conflict misses after initial cold misses
- Performance improvement comes at cost of complexity and power

---

16.3 2. Cache Configuration Parameters

16.3.1 Primary Parameters

16.3.1.1 1. Block Size

- Size of a single block in bytes
- Cache deals with memory in blocks
- CPU deals with cache in words/bytes

16.3.1.2 2. Set Size

- Number of sets in the cache
- Direct mapped: number of sets = number of entries
- Fully associative: only 1 set
- Can be confusing - refers to number of sets, not size of each set

16.3.1.3 3. Associativity

- Number of ways in a set
- Number of blocks that can be stored in one set
- 1-way = direct mapped
- 2-way = two-way set associative
- N-way = N blocks per set

16.3.2 Cache Size Calculation

```
Total Cache Size = Block Size × Set Size × Associativity
```

16.3.3 Secondary Parameters

16.3.3.1 4. Replacement Policy

- LRU (Least Recently Used)
- Pseudo-LRU (PLRU)
- FIFO (First In First Out)
- Others

16.3.3.2 5. Write Policy

- Write-through
- Write-back

16.3.3.3 6. Other Optimization Techniques

- Prefetching mechanisms
- Write buffer size
- Communication protocols

16.3.4 Configuration Definition

- Fixing values for all these parameters defines a specific cache configuration
- Performance and power consumption are determined by configuration
- External factors: memory access patterns from CPU/program

16.4 3. Improving Cache Performance - Comprehensive Review

16.4.1 Average Access Time Equation

```
T_avg = Hit Latency + Miss Rate × Miss Penalty
```

Three main factors can be optimized:

16.5 4. Hit Rate Improvement

16.5.1 Method 1: Increase Cache Size

Approach:

- Most obvious and intuitive method
- More slots → can hold more data → more likely to get hits

Limitations:

- Very expensive (SRAM costs ~$2000/GB)
- SRAM uses cutting-edge technology, same as CPU
- Must be fast enough to work at CPU speed
- Usually located inside CPU core
- Practical limit on how much cache can be added

16.5.2 Method 2: Increase Associativity

Benefits:

- Higher associativity → better hit rate
- Reduces conflict misses
- Most popular technique for given cache size

Trade-offs:

- Increases hit latency
- Increases power consumption
- Increases hardware cost

16.5.3 Method 3: Cache Prefetching

Concept:

- Fetch data before it's needed
- Similar to branch prediction in CPU
- Reduces cold misses (compulsory misses)
- Can also reduce conflict misses

Types of Prefetching:

- Software prefetching (compiler-based)
- Hardware prefetching
- Hybrid software-hardware approaches

Benefits:

- Can predict and fetch data before CPU requests it
- Reduces effective miss rate
- Can significantly improve performance for predictable access patterns

Limitations:

- Not 100% accurate
- Wrong predictions waste power and bandwidth
- Requires additional hardware
- Increases complexity

16.6 5. Hit Latency Optimization

16.6.1 Relationship with Hit Rate

Fundamental Trade-off:

- Hit rate and hit latency are tied together
- Improving hit rate often increases hit latency
- Improving hit latency often reduces hit rate
- Need to find optimal balance

Examples:

- Higher associativity → better hit rate BUT higher hit latency
- Smaller, simpler cache → lower hit latency BUT worse hit rate

Design Challenge:

- Must balance these competing factors
- Depends on application requirements
- Different trade-offs for different use cases

16.7 6. Miss Penalty Improvement

16.7.1 Miss Penalty Definition

- Time spent servicing a cache miss
- Time to fetch missing block from memory

16.7.2 Method 1: Optimize Communication

- Improve bus technology between cache and memory
- Increase bus width
- Increase bus speed
- Optimize bus arbitration
- Better communication protocols
- This assumes best possible communication is already in place

16.7.3 Method 2: Cache Hierarchy (Main Focus)

- Use multiple levels of cache
- Each level optimized differently
- Most effective technique for reducing miss penalty

16.8 7. Cache Hierarchy (Multi-Level Caches)

16.8.1 Concept

Instead of a single cache between CPU and memory, use multiple cache levels: L1, L2, L3, etc., with each level serving as backup for the level above.

16.8.2 Terminology

- **L1 (Level 1)**: Top-level cache, closest to CPU
- **L2 (Level 2)**: Second-level cache
- **L3 (Level 3)**: Third-level cache (in some systems)
- **Top-level cache**: Fastest, smallest
- **Last-level cache**: Slowest (but still fast), largest

16.8.3 Operation

1. CPU requests data from L1
2. L1 miss → request goes to L2 (not directly to memory)
3. L2 miss → request goes to L3 (if exists)
4. Last-level miss → request goes to main memory

16.8.4 Benefits

- Reduced effective miss penalty for L1
- Most L1 misses served by L2 in few cycles (2-4 cycles)
- Only L2 misses incur full memory penalty (100+ cycles)
- Overall average miss penalty greatly reduced

16.8.5 Effective Miss Penalty

For L1 cache:

```
Effective Miss Penalty = L2 Hit Latency + L2 Miss Rate × L2 Miss Penalty
```

If L2 has good hit rate:

- L2 miss rate is low
- Most L1 misses served quickly by L2
- Effective penalty much less than going to memory

16.8.6 Example Calculation

Given:

- L1 miss rate: 5%
- L2 hit rate: 99.9%
- L2 hit latency: 3 cycles
- Memory penalty: 100 cycles

```
L1 effective penalty = 3 + 0.001 × 100 = 3.1 cycles
```

vs. 100 cycles if going directly to memory

16.9 8. Optimization Strategies for Multi-Level Caches

16.9.1 Why Not One Big Cache?

- Different levels can be optimized for different goals
- Splitting allows specialized optimization
- Better overall performance than single large cache

16.10 9. L1 Cache Optimization - Optimize for Hit Latency

16.10.1 Goal

Minimize hit latency

16.10.2 Rationale

- Critical for CPU clock cycle time
- Memory access is slowest pipeline stage
- Determines overall CPU clock period
- Lower L1 hit latency → shorter clock cycle → higher CPU frequency

16.10.3 Characteristics

- Small size
- Lower associativity (2-way, 4-way, sometimes 8-way)
- Fast response time
- Accept moderate hit rate (e.g., 95%)

16.10.4 Trade-off

- Sacrifice some hit rate for speed
- Slightly higher miss rate acceptable
- Misses handled by L2

16.11 10. L2 Cache Optimization - Optimize for Hit Rate

16.11.1 Goal

Maximize hit rate

16.11.2 Rationale

- Serve most L1 misses
- Minimize accesses to main memory
- Reduce effective L1 miss penalty

16.11.3 Characteristics

- Larger size
- Higher associativity (8-way, 16-way, or even fully associative)
- Very high hit rate (99.9% or better)
- Can tolerate higher hit latency

16.11.4 Trade-off

- Higher latency acceptable
- Not on critical path for most accesses
- Priority is catching L1 misses

16.12 11. Associativity Comparison

Question: Which level has higher associativity?

Answer: L2 (and L3 if present) have higher associativity

16.12.1 Reasoning

- L2 optimized for hit rate
- Higher associativity → better hit rate
- L1 optimized for latency
- Lower associativity → faster access

16.12.2 Combined Effect

- **L1**: Fast but moderate hit rate (e.g., 95-98%)
- **L2**: Slower but excellent hit rate (e.g., 99-99.9%)
- **Most accesses**: L1 hit (fast path)
- **Most L1 misses**: L2 hit (medium path, few cycles)
- **Very few accesses**: Main memory (slow path, 100+ cycles)

Overall result: Much better average performance

16.13 12. Physical Implementation of Cache Hierarchy

16.13.1 L1 Cache

- Almost always on-chip (inside CPU die)
- Integrated within CPU core
- Smallest but fastest
- Typically split into:
  - L1 instruction cache (L1-I)
  - L1 data cache (L1-D)

16.13.2 L2 Cache

- Usually on-chip (same die as CPU)
- Can be off-chip in some designs
- Larger than L1
- May be unified (instruction + data) or split
- If multi-core: may be per-core or shared

16.13.3 L3 Cache

- Common in multi-processor/multi-core systems
- Usually on-chip in modern designs
- Can be off-chip in some architectures
- Typically unified and shared among all cores
- Largest cache level

16.13.4 Design Variations

Different implementations based on:

- Performance requirements
- Power budget
- Cost constraints
- Target application
- Number of cores

16.14 13. Real World Example: Intel Skylake Architecture

Source: wikichip.org (recommended resource)

16.14.1 Architecture Overview

- Mainstream Intel architecture from ~2015
- Used in Core i3, i5, i7 processors
- Standard desktop/PC processors

16.14.2 Dual-Core Layout Analysis

16.14.2.1 Execution Units

- Two separate processor cores visible
- Integer ALUs (arithmetic logic units)
- Floating-point units
- Multipliers, dividers
- Other arithmetic hardware

16.14.2.2 Pipeline Support Hardware

- Takes up as much space as execution units
- Out-of-order scheduling logic
- Branch prediction units
- Multiple issue hardware
- Decoding logic
- Control logic

16.14.3 Cache Implementation

16.14.3.1 L1 Data Cache

- Separate for each core
- Located close to execution units and memory management
- **8-way set associative**
- Smaller size (32KB typical)
- Close to where addresses are generated

16.14.3.2 L1 Instruction Cache

- Separate for each core
- Located close to instruction fetch and decode units
- Near out-of-order scheduling hardware
- **8-way set associative**
- Smaller size (32KB typical)

16.14.3.3 L2 Cache

- Shared between instruction and data
- Larger than L1 (256KB in this example)
- **4-way set associative** (in this design)
- Located between L1 and memory

- Serves both L1-I and L1-D misses

16.14.4 Memory Hierarchy

- Separate buffers for load and store instructions
- Buffers before and after cache
- Memory management unit
- Connection to L3 cache (if present) via bus

16.14.5 Design Observations

- Physical placement matches logical function
- Data cache near execution units
- Instruction cache near fetch/decode
- Shared L2 in middle position
- Significant die area for cache
- Even more area for pipeline optimization

16.14.6 Why Higher L1 Associativity Here?

- 8-way seems high for L1
- But size is small (32KB)
- Other pipeline stages may be bottleneck
- Clock period limited by other factors
- Can afford higher associativity without hurting cycle time
- Depends on overall CPU design

16.14.7 Multi-Core Configuration

- Each core has own L1-I and L1-D
- Each core has own L2
- All cores share L3
- L3 connects via bus system

16.14.8 Additional Features

- Physical register files (integer and vector)
- Store/load buffers
- Pre-decoding hardware
- Complex x86 instruction handling
- Many optimizations for real-world performance

16.15 14. Recommendations for Further Study

16.15.1 Resource: wikichip.org

Content Available:

- Detailed CPU architecture information
- Real implementation details
- Various processor families:
  - Intel x86 architectures
  - ARM implementations
  - AMD processors
  - Other architectures

Benefits:

- See concepts in real hardware
- Understand practical trade-offs
- Compare different design approaches
- Learn industry practices

16.16 15. Next Topics

16.16.1 Upcoming in Course

16.16.1.1 1. Virtual Memory

- Requires hardware support
- Memory management concepts
- Next week's lectures

16.16.1.2 2. Lab 6 (Next Week)

- Implement cache memory
- Add memory modules to CPU
- Build on Lab 5 CPU implementation
- Apply concepts learned

16.17 Key Takeaways

1. Cache hierarchies reduce effective miss penalty
2. Different levels optimized for different goals:
   - L1: Hit latency (speed)
   - L2/L3: Hit rate (coverage)
3. Multi-level caches balance competing requirements

4. Real implementations show concepts in practice
5. Design decisions depend on:
    - Performance targets
    - Power budget
    - Cost constraints
    - Application requirements
6. Modern CPUs use sophisticated cache hierarchies
7. Cache takes significant portion of CPU die area
8. Pipeline optimizations also require substantial hardware

16.18 Summary

Cache hierarchies represent one of the most effective techniques for improving memory system performance. By using multiple levels of cache, each optimized for different objectives, modern processors achieve both low latency and high hit rates. The L1 cache prioritizes speed to minimize clock cycle time, while L2 and L3 caches prioritize capacity and hit rate to reduce memory access frequency. Real-world implementations, such as Intel's Skylake architecture, demonstrate these principles in practice, showing how careful cache design enables high-performance computing while managing the constraints of power, cost, and chip area.

Chapter 17

Lecture 17: Virtual Memory

17.1 Introduction

Virtual memory represents one of the most elegant abstractions in computer architecture, creating a layer between physical memory hardware and the memory view presented to programs. This lecture explores how virtual memory enables programs to use more memory than physically available by treating main memory as a cache for disk storage, supports safe execution of multiple concurrent programs through address space isolation, and provides memory protection mechanisms preventing programs from corrupting each other's data. We examine page tables, translation lookaside buffers (TLBs), page faults, and the critical design decisions that make virtual memory both practical and performant despite the enormous speed gap between RAM and disk storage.

17.2 1. Introduction to Virtual Memory

Virtual memory allows programs to use more memory than physically available by using main memory as a cache for secondary storage.

17.2.1 Key Purposes of Virtual Memory

1. **Allow programs to use more memory than actually available**
2. **Support multiple programs running simultaneously on a CPU**
3. **Enable safe and efficient memory sharing between programs**
4. **Ensure programs only access their allocated memory**

17.3 2. CPU Word Size and Address Space

The relationship between CPU word size and addressable memory determines the maximum amount of memory that can be addressed.

17.3.1 Address Space by CPU Word Size

17.3.1.1 8-bit CPU

- **Maximum addressable memory**: 256 bytes (2^8)

17.3.1.2 16-bit CPU

- **Maximum addressable memory**: 64 kilobytes (2^16)

17.3.1.3 32-bit CPU

- **Maximum addressable memory**: 4 gigabytes (2^32)
- Became mainstream in early 1980s
- Was replaced when systems started reaching 4 GB memory limit

17.3.1.4 64-bit CPU

- **Maximum addressable memory**: 16 exabytes (2^64)
- About 16 million gigabytes
- Current mainstream word size
- Became mainstream around 2002-2003

17.3.2 Historical Pattern

- Maximum address space sizes were always much larger than commonly used RAM sizes
- Architectures were replaced when high-end systems started reaching the address space limits
- Personal computers typically had much less memory than the theoretical maximum

---

17.4 3. Virtual vs Physical Addresses

17.4.1 Virtual Address

- **Address generated by CPU**
- Refers to entire theoretical address space
- CPU thinks it has access to full address space
- In 64-bit CPU: can address up to 16 exabytes

17.4.2 Physical Address

- **Actual address in real memory (RAM)**
- Much smaller range than virtual addresses
- Typical modern RAM: 8-16 GB (much less than 16 exabytes)

17.4.3 Address Translation

- Virtual addresses must be translated to physical addresses
- Translation required every time memory is accessed
- Main mechanism for making virtual memory work

17.5 4. Memory Hierarchy with Virtual Memory

Complete hierarchy from top to bottom:

1. **CPU** (generates virtual addresses, thinks memory is large and fast)
2. **Cache** (virtually or physically addressed)
3. **Main Memory** (acts as cache for secondary storage)
4. **Secondary Storage/Disk** (contains all pages)

CPU accesses cache directly. Main memory acts as cache for disk, not just a second level cache - requires additional mechanisms.

17.6 5. Terminology

17.6.1 CPU Level

- **Accesses**: Words (1, 4, or 8 bytes)
- Hit/Miss terminology used

17.6.2 Cache Level

- **Transfers**: Blocks (16-256 bytes typically)
- Hit/Miss terminology used

17.6.3 Memory Level

- **Transfers**: Pages (1 KB to 64 KB typically)
- **Page Hit**: Page is present in memory
- **Page Fault**: Page is not present in memory (not “miss”)

17.7 6. Access Latencies

Understanding the latency differences is crucial for virtual memory design:

- **Cache Hit**: Under 1 cycle
- **Cache Miss** (accessing main memory): 10-100 cycles
- **Page Fault** (accessing disk): ~1 million cycles
  - Extremely large penalty
  - Influences design decisions significantly
  - Page faults handled in software by OS due to large penalty

---

17.8 7. Virtual and Physical Address Structure

17.8.1 Example with 32-bit Addresses

17.8.1.1 Virtual Address (32 bits)

- **Virtual Page Number**: 22 bits (most significant)
- **Page Offset**: 10 bits (least significant)
- **Virtual address space**: 4 GB
- **Number of virtual pages**: 2^22 pages
- **Page size**: 2^10 = 1 KB

17.8.1.2 Physical Address (28 bits)

- **Physical Page Number (Frame Number)**: 18 bits (most significant)
- **Page Offset**: 10 bits (least significant)
- **Physical address space**: 256 MB
- **Number of frames**: 2^18 frames
- **Page size**: 1 KB (same as virtual)

17.8.2 Key Points

- Page offset has same number of bits in virtual and physical addresses
- Physical address space is smaller than virtual address space
- Memory contains “frames” where pages can be placed
- Frame = slot in memory that can hold a page

17.9 8. Supporting Multiple Programs

Multiple programs can run simultaneously by sharing physical memory:

17.9.1 Each Program

- Has its own virtual address space
- Thinks it has entire memory to itself
- CPU switches between programs quickly
- Creates impression of simultaneous execution

17.9.2 Memory Sharing

- Physical memory contains active pages from all running programs
- Each program's virtual pages map to different physical frames
- Operating system ensures programs only access their own memory

17.9.3 Example

- Program 1 virtual address space: 8 virtual pages
- Program 2 virtual address space: 8 virtual pages
- Physical memory: Only 4 frames available
- Active pages from both programs share the 4 frames
- Same virtual page number from different programs can map to different physical frames

---

17.10 9. Page Table

The page table is a data structure stored in memory that contains address translations.

17.10.1 Purpose

- Stores virtual-to-physical address translations
- One page table per program
- Contains entries for ALL virtual pages (not just active ones)

17.10.2 Page Table Entry Contents

1. **Physical Page Number** (main component)
2. **Valid Bit**: Is the page currently in memory?
   - 1 = Page is in memory (translation valid)
   - 0 = Page not in memory (page fault)
3. **Dirty Bit**: Has page been modified?
   - 1 = Page modified, inconsistent with disk
   - 0 = Page not modified, consistent with disk
4. **Additional bits**: Access permissions, memory protection status

17.10.3 Finding Page Table

- Page tables stored at fixed locations in memory
- **Page Table Base Register (PTBR)**: Special CPU register storing starting address of active page table
- When CPU switches programs, OS updates PTBR to point to correct page table

---

17.11 10. Address Translation Process

Steps to access memory:

1. **CPU generates virtual address** (virtual page number + page offset)
2. **Access page table** using PTBR + virtual page number as index
3. **Read page table entry**:
    - If valid bit = 0: Page fault (handled by OS)
    - If valid bit = 1: Read physical page number
4. **Construct physical address**: Physical page number + page offset
5. **Access physical memory** with physical address
6. **Return data to CPU**

17.11.1 Memory Accesses Required

- One access for page table
- One access for actual data
- **Total**: Two memory accesses per data access

17.12 11. Page Table Size Calculation

17.12.1 Example: 4 GB Virtual, 1 GB Physical, 1 KB Pages

17.12.1.1 Number of Entries

- Virtual address: 32 bits
- Page offset: 10 bits (for 1 KB pages)
- Virtual page number: 22 bits
- **Number of entries**: 2^22 = ~4 million entries

17.12.1.2 Entry Size

- Physical address: 30 bits (for 1 GB)
- Page offset: 10 bits
- Physical page number: 20 bits
- Valid bit: 1 bit
- Dirty bit: 1 bit

- Total needed: 22 bits
- Actual storage: 32 bits (word-aligned)
- **Size per entry**: 4 bytes

17.12.1.3 Total Page Table Size

- 4 bytes × 2^22 entries = **16 MB**
- Significant memory overhead for each program

17.13 12. Write Policy for Virtual Memory

17.13.1 Write-Through: NOT USED

- Would require writing to disk on every write
- 1 million cycle penalty unacceptable
- Not a good design decision

17.13.2 Write-Back: USED (Standard Policy)

- Writes only update memory
- Dirty bit tracks modified pages
- Only write to disk when:
  - Page is evicted from memory
  - Page's dirty bit is 1
- Minimizes disk accesses

17.14 13. Placement Policy

17.14.1 Fully Associative Placement

- Any page from disk can go to any frame in memory
- Memory treated as one large set containing all frames
- No direct mapping or set restrictions
- Maximizes flexibility in page placement
- Reduces page faults

17.14.2 Why Fully Associative?

- Minimizes page faults (primary goal)
- Large page fault penalty (1 million cycles) justifies complexity
- Different from cache (doesn't use tag comparators in memory)
- Address translation through page table provides necessary mechanism

---

17.15 14. Page Fault Handling

17.15.1 What Operating System Must Do

17.15.1.1 1. Fetch Missing Page

- Access disk to retrieve page
- OS must know disk location of page
- OS maintains data structures tracking page locations

17.15.1.2 2. Find Unused Frame

- OS tracks which frames are currently used
- Can determine this through page tables
- If unused frame exists: Place page in unused frame

17.15.1.3 3. If Memory Full (No Unused Frames)

- Select active page to replace using replacement policy
- Common replacement policies:
  - Least Recently Used (LRU)
  - Pseudo-LRU (PLRU)
  - First-In-First-Out (FIFO)
  - Least Frequently Used (LFU)
- Goal: Exploit temporal locality (keep recently/frequently used pages)

17.15.1.4 4. Check Dirty Bit of Page to be Replaced

- If dirty bit = 1: Write page back to disk before replacement
- If dirty bit = 0: Can directly overwrite (data consistent with disk)
- Prevents data loss

17.15.1.5 5. Update Data Structures

- Update page table entry for new page
- Update page table entry for replaced page (set valid = 0)
- Place fetched page in frame

17.15.2 Optimization

- Many operations can occur in parallel during disk fetch
- While fetching data, OS can determine placement and handle replacement
- Use buffers for write-back operations

17.15.3 Why Software Handling?

- 1 million cycle penalty is so large that software overhead is negligible
- Complex replacement policies better suited to software

- Hardware optimization doesn't provide significant benefit

17.16 15. Translation Lookaside Buffer (TLB)

17.16.1 Purpose

- Avoid accessing memory twice for every data access
- Act as cache for page table entries
- Reduce address translation overhead

17.16.2 What is TLB?

- Hardware cache specifically for page table entries
- Stores recently used address translations
- Based on locality of page table entry accesses
- Exploits temporal and spatial locality of page accesses

17.16.3 TLB Entry Structure

- **Tag**: Virtual address tag (or physical address tag)
- **Physical Page Number**: The address translation
- **Valid Bit**: Is this TLB entry valid?
  - Different from page table valid bit
  - Indicates if TLB entry contains valid translation
- **Dirty Bit**: Same meaning as in page table

17.16.4 TLB Parameters

17.16.4.1 Size

- **16-512 page table entries** (typical range)

17.16.4.2 Block Size

- **1-2 address translations**
- Small blocks because spatial locality between pages is larger
- Adjacent pages not as closely related as adjacent cache blocks

17.16.4.3 Placement Policy

- Fully associative or set associative
- Fully associative for smaller TLBs (~16 entries)
- Set associative for larger TLBs
- Goal: Keep miss rate below 1%

17.16.4.4 Hit Latency

- **Much less than 1 cycle**

17.16.4.5 Miss Penalty

- **10-100 cycles** (memory access required)

17.16.5 TLB Operation

17.16.5.1 Hit

- Address translation available in TLB
- Use translation directly without accessing memory
- Only one memory access needed (for data)

17.16.5.2 Miss

- Translation not in TLB
- Must access page table in memory
- Total: Two memory accesses (page table + data)

17.16.6 Why Low Miss Rate Essential?

- TLB misses double memory access time
- Must access page table (10-100 cycles) then data
- Miss rate typically kept below 1%
- 99% of translations served by TLB

17.17 16. Complete Memory Access with TLB

Two different approaches for handling memory access with TLB:

17.18 17. Approach 1: Virtually Addressed Cache

17.18.1 Process

1. **CPU generates virtual address**
2. **Access cache with virtual address** (parallel with TLB)
3. **Cache Hit**: Return data to CPU immediately

4. **Cache Miss**:

    a. **Check TLB for address translation**

    b. **TLB Hit**:

    - Get physical address
    - Access memory with physical address
    - Fetch missing block
    - Update cache
    - Send word to CPU

    c. **TLB Miss**:

    - Access page table in memory
    - **Page Hit**:
        - Get translation
        - Access memory for data
        - Update TLB
        - Update cache
        - Send word to CPU
    - **Page Fault**:
        - OS accesses disk
        - Fetch missing page
        - Find unused frame or replace page
        - If replaced page dirty: write back
        - Update page table
        - Update TLB
        - Update cache
        - Send word to CPU

17.18.2 Advantage

- TLB access overlapped with cache access
- Both happen in parallel
- No additional latency for TLB access on cache hit

17.19 18. Approach 2: Physically Addressed Cache

17.19.1 Process

1. **CPU generates virtual address**

2. **Access TLB for translation first**

3. **TLB Hit**:

    a. Get physical address

    b. **Access cache with physical address**

c. **Cache Hit**: Return data to CPU

d. **Cache Miss**:

- Access memory with physical address
- Fetch missing block
- Update cache
- Send word to CPU

4. **TLB Miss**:

a. Access page table in memory

b. **Page Hit**:

- Get translation
- Update TLB
- Access cache with physical address
- If cache hit: return data
- If cache miss: fetch from memory, update cache, return data

c. **Page Fault**:

- OS handles as described above
- Update page table, TLB, cache
- Return data to CPU

17.19.2 Advantage

- Cache physically indexed and tagged
- Simpler cache design
- No aliasing issues

17.19.3 Key Difference

- **Approach 1**: Cache uses virtual addresses, TLB access parallel
- **Approach 2**: Cache uses physical addresses, TLB access sequential

Both approaches are valid, and the choice depends on cache indexing method (virtual vs physical).

17.20 Key Takeaways

1. Virtual memory provides memory abstraction and protection
2. Address translation is fundamental to virtual memory operation
3. Page tables map virtual addresses to physical addresses
4. TLB caches translations to avoid double memory access
5. Page faults are extremely expensive (~1 million cycles)
6. Write-back policy is essential for virtual memory
7. Fully associative placement minimizes page faults

8. Multiple programs can safely share physical memory
9. OS handles page faults in software
10. Virtual memory enables modern multitasking operating systems

---

17.21 Summary

Virtual memory represents a crucial abstraction in modern computing, enabling efficient and safe memory management across multiple concurrent programs while providing each program with the illusion of abundant, dedicated memory resources.

Chapter 18

Lecture 18: Virtual Memory

18.1 Introduction

Virtual memory represents one of the most elegant abstractions in computer architecture, creating a layer between physical memory hardware and the memory view presented to programs. This lecture explores how virtual memory enables programs to use more memory than physically available by treating main memory as a cache for disk storage, supports safe execution of multiple concurrent programs through address space isolation, and provides memory protection mechanisms preventing programs from corrupting each other's data. We examine page tables, translation lookaside buffers (TLBs), page faults, and the critical design decisions that make virtual memory both practical and performant despite the enormous speed gap between RAM and disk storage.

18.2 1. Introduction to Virtual Memory

Virtual memory allows programs to use more memory than physically available by using main memory as a cache for secondary storage.

18.2.1 Key Purposes of Virtual Memory

1. **Allow programs to use more memory than actually available**
2. **Support multiple programs running simultaneously on a CPU**
3. **Enable safe and efficient memory sharing between programs**
4. **Ensure programs only access their allocated memory**

18.3 2. CPU Word Size and Address Space

The relationship between CPU word size and addressable memory determines the maximum amount of memory that can be addressed.

18.3.1 Address Space by CPU Word Size

18.3.1.1 8-bit CPU

- **Maximum addressable memory**: 256 bytes (2^8)

18.3.1.2 16-bit CPU

- **Maximum addressable memory**: 64 kilobytes (2^16)

18.3.1.3 32-bit CPU

- **Maximum addressable memory**: 4 gigabytes (2^32)
- Became mainstream in early 1980s
- Was replaced when systems started reaching 4 GB memory limit

18.3.1.4 64-bit CPU

- **Maximum addressable memory**: 16 exabytes (2^64)
- About 16 million gigabytes
- Current mainstream word size
- Became mainstream around 2002-2003

18.3.2 Historical Pattern

- Maximum address space sizes were always much larger than commonly used RAM sizes
- Architectures were replaced when high-end systems started reaching the address space limits
- Personal computers typically had much less memory than the theoretical maximum

---

18.4 3. Virtual vs Physical Addresses

18.4.1 Virtual Address

- **Address generated by CPU**
- Refers to entire theoretical address space
- CPU thinks it has access to full address space
- In 64-bit CPU: can address up to 16 exabytes

18.4.2 Physical Address

- **Actual address in real memory (RAM)**
- Much smaller range than virtual addresses
- Typical modern RAM: 8-16 GB (much less than 16 exabytes)

18.4.3 Address Translation

- Virtual addresses must be translated to physical addresses
- Translation required every time memory is accessed
- Main mechanism for making virtual memory work

18.5 4. Memory Hierarchy with Virtual Memory

Complete hierarchy from top to bottom:

1. **CPU** (generates virtual addresses, thinks memory is large and fast)
2. **Cache** (virtually or physically addressed)
3. **Main Memory** (acts as cache for secondary storage)
4. **Secondary Storage/Disk** (contains all pages)

CPU accesses cache directly. Main memory acts as cache for disk, not just a second level cache - requires additional mechanisms.

18.6 5. Terminology

18.6.1 CPU Level

- **Accesses**: Words (1, 4, or 8 bytes)
- Hit/Miss terminology used

18.6.2 Cache Level

- **Transfers**: Blocks (16-256 bytes typically)
- Hit/Miss terminology used

18.6.3 Memory Level

- **Transfers**: Pages (1 KB to 64 KB typically)
- **Page Hit**: Page is present in memory
- **Page Fault**: Page is not present in memory (not “miss”)

18.7 6. Access Latencies

Understanding the latency differences is crucial for virtual memory design:

- **Cache Hit**: Under 1 cycle
- **Cache Miss** (accessing main memory): 10-100 cycles
- **Page Fault** (accessing disk): ~1 million cycles
  - Extremely large penalty
  - Influences design decisions significantly
  - Page faults handled in software by OS due to large penalty

---

18.8 7. Virtual and Physical Address Structure

18.8.1 Example with 32-bit Addresses

18.8.1.1 Virtual Address (32 bits)

- **Virtual Page Number**: 22 bits (most significant)
- **Page Offset**: 10 bits (least significant)
- **Virtual address space**: 4 GB
- **Number of virtual pages**: 2^22 pages
- **Page size**: 2^10 = 1 KB

18.8.1.2 Physical Address (28 bits)

- **Physical Page Number (Frame Number)**: 18 bits (most significant)
- **Page Offset**: 10 bits (least significant)
- **Physical address space**: 256 MB
- **Number of frames**: 2^18 frames
- **Page size**: 1 KB (same as virtual)

18.8.2 Key Points

- Page offset has same number of bits in virtual and physical addresses
- Physical address space is smaller than virtual address space
- Memory contains “frames” where pages can be placed
- Frame = slot in memory that can hold a page

18.9 8. Supporting Multiple Programs

Multiple programs can run simultaneously by sharing physical memory:

18.9.1 Each Program

- Has its own virtual address space
- Thinks it has entire memory to itself
- CPU switches between programs quickly
- Creates impression of simultaneous execution

18.9.2 Memory Sharing

- Physical memory contains active pages from all running programs
- Each program's virtual pages map to different physical frames
- Operating system ensures programs only access their own memory

18.9.3 Example

- Program 1 virtual address space: 8 virtual pages
- Program 2 virtual address space: 8 virtual pages
- Physical memory: Only 4 frames available
- Active pages from both programs share the 4 frames
- Same virtual page number from different programs can map to different physical frames

---

18.10 9. Page Table

The page table is a data structure stored in memory that contains address translations.

18.10.1 Purpose

- Stores virtual-to-physical address translations
- One page table per program
- Contains entries for ALL virtual pages (not just active ones)

18.10.2 Page Table Entry Contents

1. **Physical Page Number** (main component)
2. **Valid Bit**: Is the page currently in memory?
   - 1 = Page is in memory (translation valid)
   - 0 = Page not in memory (page fault)
3. **Dirty Bit**: Has page been modified?
   - 1 = Page modified, inconsistent with disk
   - 0 = Page not modified, consistent with disk
4. **Additional bits**: Access permissions, memory protection status

18.10.3 Finding Page Table

- Page tables stored at fixed locations in memory
- **Page Table Base Register (PTBR)**: Special CPU register storing starting address of active page table
- When CPU switches programs, OS updates PTBR to point to correct page table

---

18.11 10. Address Translation Process

Steps to access memory:

1. **CPU generates virtual address** (virtual page number + page offset)
2. **Access page table** using PTBR + virtual page number as index
3. **Read page table entry**:
    - If valid bit = 0: Page fault (handled by OS)
    - If valid bit = 1: Read physical page number
4. **Construct physical address**: Physical page number + page offset
5. **Access physical memory** with physical address
6. **Return data to CPU**

18.11.1 Memory Accesses Required

- One access for page table
- One access for actual data
- **Total**: Two memory accesses per data access

18.12 11. Page Table Size Calculation

18.12.1 Example: 4 GB Virtual, 1 GB Physical, 1 KB Pages

18.12.1.1 Number of Entries

- Virtual address: 32 bits
- Page offset: 10 bits (for 1 KB pages)
- Virtual page number: 22 bits
- **Number of entries**: 2^22 = ~4 million entries

18.12.1.2 Entry Size

- Physical address: 30 bits (for 1 GB)
- Page offset: 10 bits
- Physical page number: 20 bits
- Valid bit: 1 bit
- Dirty bit: 1 bit

- Total needed: 22 bits
- Actual storage: 32 bits (word-aligned)
- **Size per entry**: 4 bytes

18.12.1.3 Total Page Table Size

- 4 bytes × 2^22 entries = **16 MB**
- Significant memory overhead for each program

18.13 12. Write Policy for Virtual Memory

18.13.1 Write-Through: NOT USED

- Would require writing to disk on every write
- 1 million cycle penalty unacceptable
- Not a good design decision

18.13.2 Write-Back: USED (Standard Policy)

- Writes only update memory
- Dirty bit tracks modified pages
- Only write to disk when:
  - Page is evicted from memory
  - Page's dirty bit is 1
- Minimizes disk accesses

18.14 13. Placement Policy

18.14.1 Fully Associative Placement

- Any page from disk can go to any frame in memory
- Memory treated as one large set containing all frames
- No direct mapping or set restrictions
- Maximizes flexibility in page placement
- Reduces page faults

18.14.2 Why Fully Associative?

- Minimizes page faults (primary goal)
- Large page fault penalty (1 million cycles) justifies complexity
- Different from cache (doesn't use tag comparators in memory)
- Address translation through page table provides necessary mechanism

---

18.15 14. Page Fault Handling

18.15.1 What Operating System Must Do

18.15.1.1 1. Fetch Missing Page

- Access disk to retrieve page
- OS must know disk location of page
- OS maintains data structures tracking page locations

18.15.1.2 2. Find Unused Frame

- OS tracks which frames are currently used
- Can determine this through page tables
- If unused frame exists: Place page in unused frame

18.15.1.3 3. If Memory Full (No Unused Frames)

- Select active page to replace using replacement policy
- Common replacement policies:
  - Least Recently Used (LRU)
  - Pseudo-LRU (PLRU)
  - First-In-First-Out (FIFO)
  - Least Frequently Used (LFU)
- Goal: Exploit temporal locality (keep recently/frequently used pages)

18.15.1.4 4. Check Dirty Bit of Page to be Replaced

- If dirty bit = 1: Write page back to disk before replacement
- If dirty bit = 0: Can directly overwrite (data consistent with disk)
- Prevents data loss

18.15.1.5 5. Update Data Structures

- Update page table entry for new page
- Update page table entry for replaced page (set valid = 0)
- Place fetched page in frame

18.15.2 Optimization

- Many operations can occur in parallel during disk fetch
- While fetching data, OS can determine placement and handle replacement
- Use buffers for write-back operations

18.15.3 Why Software Handling?

- 1 million cycle penalty is so large that software overhead is negligible
- Complex replacement policies better suited to software

- Hardware optimization doesn't provide significant benefit

18.16 15. Translation Lookaside Buffer (TLB)

18.16.1 Purpose

- Avoid accessing memory twice for every data access
- Act as cache for page table entries
- Reduce address translation overhead

18.16.2 What is TLB?

- Hardware cache specifically for page table entries
- Stores recently used address translations
- Based on locality of page table entry accesses
- Exploits temporal and spatial locality of page accesses

18.16.3 TLB Entry Structure

- **Tag**: Virtual address tag (or physical address tag)
- **Physical Page Number**: The address translation
- **Valid Bit**: Is this TLB entry valid?
  - Different from page table valid bit
  - Indicates if TLB entry contains valid translation
- **Dirty Bit**: Same meaning as in page table

18.16.4 TLB Parameters

18.16.4.1 Size

- **16-512 page table entries** (typical range)

18.16.4.2 Block Size

- **1-2 address translations**
- Small blocks because spatial locality between pages is larger
- Adjacent pages not as closely related as adjacent cache blocks

18.16.4.3 Placement Policy

- Fully associative or set associative
- Fully associative for smaller TLBs (~16 entries)
- Set associative for larger TLBs
- Goal: Keep miss rate below 1%

18.16.4.4 Hit Latency

- **Much less than 1 cycle**

18.16.4.5 Miss Penalty

- **10-100 cycles** (memory access required)

18.16.5 TLB Operation

18.16.5.1 Hit

- Address translation available in TLB
- Use translation directly without accessing memory
- Only one memory access needed (for data)

18.16.5.2 Miss

- Translation not in TLB
- Must access page table in memory
- Total: Two memory accesses (page table + data)

18.16.6 Why Low Miss Rate Essential?

- TLB misses double memory access time
- Must access page table (10-100 cycles) then data
- Miss rate typically kept below 1%
- 99% of translations served by TLB

18.17 16. Complete Memory Access with TLB

Two different approaches for handling memory access with TLB:

18.18 17. Approach 1: Virtually Addressed Cache

18.18.1 Process

1. **CPU generates virtual address**
2. **Access cache with virtual address** (parallel with TLB)
3. **Cache Hit**: Return data to CPU immediately

4. **Cache Miss**:

    a. **Check TLB for address translation**

    b. **TLB Hit**:

    - Get physical address
    - Access memory with physical address
    - Fetch missing block
    - Update cache
    - Send word to CPU

    c. **TLB Miss**:

    - Access page table in memory
    - **Page Hit**:
        - Get translation
        - Access memory for data
        - Update TLB
        - Update cache
        - Send word to CPU
    - **Page Fault**:
        - OS accesses disk
        - Fetch missing page
        - Find unused frame or replace page
        - If replaced page dirty: write back
        - Update page table
        - Update TLB
        - Update cache
        - Send word to CPU

18.18.2 Advantage

- TLB access overlapped with cache access
- Both happen in parallel
- No additional latency for TLB access on cache hit

18.19 18. Approach 2: Physically Addressed Cache

18.19.1 Process

1. **CPU generates virtual address**

2. **Access TLB for translation first**

3. **TLB Hit**:

    a. Get physical address

    b. **Access cache with physical address**

   c. **Cache Hit**: Return data to CPU

   d. **Cache Miss**:

      - Access memory with physical address
      - Fetch missing block
      - Update cache
      - Send word to CPU

4. **TLB Miss**:

   a. Access page table in memory

   b. **Page Hit**:

      - Get translation
      - Update TLB
      - Access cache with physical address
      - If cache hit: return data
      - If cache miss: fetch from memory, update cache, return data

   c. **Page Fault**:

      - OS handles as described above
      - Update page table, TLB, cache
      - Return data to CPU

18.19.2 Advantage

- Cache physically indexed and tagged
- Simpler cache design
- No aliasing issues

18.19.3 Key Difference

- **Approach 1**: Cache uses virtual addresses, TLB access parallel
- **Approach 2**: Cache uses physical addresses, TLB access sequential

Both approaches are valid, and the choice depends on cache indexing method (virtual vs physical).

18.20 Key Takeaways

1. Virtual memory provides memory abstraction and protection
2. Address translation is fundamental to virtual memory operation
3. Page tables map virtual addresses to physical addresses
4. TLB caches translations to avoid double memory access
5. Page faults are extremely expensive (~1 million cycles)
6. Write-back policy is essential for virtual memory
7. Fully associative placement minimizes page faults

8. Multiple programs can safely share physical memory
9. OS handles page faults in software
10. Virtual memory enables modern multitasking operating systems

---

18.21 Summary

Virtual memory represents a crucial abstraction in modern computing, enabling efficient and safe memory management across multiple concurrent programs while providing each program with the illusion of abundant, dedicated memory resources.

Chapter 19

Lecture 19: Multiprocessors

19.1 Introduction

Multiprocessor systems represent a fundamental paradigm shift in computer architecture, using multiple processors on the same chip to execute multiple programs or threads simultaneously when traditional performance improvement techniques—clock frequency scaling and instruction-level parallelism—reached physical and practical limits. This lecture explores the evolution toward multiprocessor architectures driven by power walls and parallelism walls, examines the critical challenge of cache coherence that arises when multiple processors maintain private caches of shared memory, and analyzes solutions including bus snooping protocols like MESI and scalable directory-based coherence schemes. We compare architectural organizations from uniform memory access (UMA) to non-uniform memory access (NUMA), understanding how different designs balance simplicity, performance, and scalability for systems ranging from dual-core smartphones to thousand-processor supercomputers.

19.2 1. Introduction to Multiprocessors

Multiprocessor systems address performance limitations encountered with single processor systems by employing multiple processors on the same chip to execute multiple programs or threads simultaneously.

19.3 2. Performance Evolution Background

19.3.1 Historical Performance Improvements

19.3.1.1 Early Methods: Clock Frequency Scaling

Approach:

- Increasing clock frequency (reducing clock cycle time)
- Goal: Spend less time per instruction

Limitations Encountered:

- Hit barrier at ~4 GHz: Power wall problem
- Excessive power dissipation caused overheating
- Cooling became inadequate
- Could not sustainably increase frequency further

19.3.1.2 Instruction Level Parallelism (ILP)

Techniques:

- **Pipelining**: Process multiple instructions simultaneously
- Utilize different hardware components at same time
- Example: Execute one instruction while fetching another
- **Advanced techniques**: Multiple issue, out-of-order execution
- Exploit parallelism inherent in programs

Limitations Encountered:

- Programs contain limited inherent parallelism
- Dependencies prevent unlimited parallel execution
- Hit “parallelism wall”
- Can’t exploit more parallelism beyond program’s inherent limits

19.3.2 Moore’s Law Context

Observation:

- Number of transistors doubles every 2 years
- Technology improves, more transistors available

Question: How to use abundant transistors?

Solution: Multiple processors on same chip

19.4 3. Multiprocessor Approach

19.4.1 Key Characteristics

- Multiple processor cores on same chip
- Execute multiple instruction streams simultaneously
- Run multiple programs/threads in real time (true parallelism)
- Different from single processor illusion of parallelism

19.4.2 Terminology

- **Processing Elements (PE)**: Common term for individual processors
- Each PE is a complete CPU with fetch, decode, execute units

19.4.3 Key Problem: Communication Between Processors

- Multiple processors executing simultaneously
- Programs often need to communicate/share data
- Splitting programs into threads requires coordination
- Communication is central design challenge

19.5 4. Shared Memory Multiprocessors (SMM)

19.5.1 Most Common Approach

Architecture:

- Communication through shared memory
- All processors access same physical address space
- Memory device connected via common bus/interconnect

19.5.2 Operating System Role

Responsibilities:

- OS code stored in shared memory
- OS shared between all processors
- Manages memory access arbitration
- Performs workload balancing
- Ensures processors access only authorized memory portions

19.5.3 Workload Balancing

Purpose:

- OS distributes tasks among processors
- Goal: All processors working in parallel
- Avoid idle processors
- Maximize overall system utilization

19.6 5. Memory Contention Problem

19.6.1 Inherent Issue

Challenge:

- Multiple processors accessing same memory device
- Competition for memory access
- Processors must wait for memory availability
- Synchronization overhead
- Access time increases with contention

19.6.2 Effect on Performance

Bottleneck:

- Memory becomes bottleneck
- Bus connects all processors to memory
- If one processor using memory, others must wait
- Can take hundreds of cycles
- Limits scalability

19.7 6. Uniform Memory Access (UMA)

19.7.1 Definition

Characteristics:

- Each processor sees memory in exact same way
- Same average memory access time for all processors
- Access time independent of which processor is accessing
- By design, no difference in access time (ignoring contention)

19.7.2 Also Known As

- **Symmetric Multiprocessors (SMP)**
- Both terms used interchangeably

19.7.3 Key Properties

- Shared address space
- Uniform view of memory by all processors
- All processors experience same average latency

19.8 7. Solution to Contention: Caches

19.8.1 Using Local Caches

Approach:

- Each processor has private cache
- Based on locality principles (temporal and spatial)
- Most memory accesses served at cache level
- Only small percentage (misses) go to main memory
- Reduces bus/memory contention significantly

19.8.2 Benefits

- Exploits locality in programs
- Minimizes memory accesses
- Reduces bottleneck effect
- Allows better scalability

19.8.3 New Problem: Cache Coherence

- Shared data blocks can be in multiple caches
- Updates in one cache not automatically reflected in others
- Need mechanism to maintain consistency

19.9 8. Cache Coherence Problem

19.9.1 The Issue

Scenario:

- Multiple caches have copies of same data block
- One processor writes to that block
- Other caches have stale (old) data
- Processors see different values for same address
- Data becomes incoherent

19.9.2 Example Sequence

1. **PE1 reads X (value = 1)** → Cached in PE1
2. **PE2 reads X (value = 1)** → Cached in PE2
3. **PE1 writes X = 0** → PE1 cache updated
4. Memory may or may not be updated (depends on write policy)
5. PE2 still sees X = 1 (stale data)
6. **Inconsistency**: Same address, different values

19.9.3 With Write-Through Policy

- PE1 writes X = 0 → Cache and memory updated
- Memory has correct value
- But PE2 cache still has old value (X = 1)
- Coherence still lost

19.9.4 With Write-Back Policy

- PE1 writes X = 0 → Only cache updated
- Memory still has old value (X = 1)
- PE2 cache still has old value (X = 1)
- Both memory and PE2 incoherent with PE1

19.9.5 Requirement

- Cache coherence MUST be maintained
- Otherwise parallel programs execute incorrectly
- Get wrong results
- Latest updates must be visible to all processors

19.10 9. Bus Snooping

Common technique for cache coherence in SMP systems.

19.10.1 What is Bus Snooping?

Mechanism:

- Dedicated bus for coherency control: **Snoop bus**
- Sole purpose: Control coherency of cache data
- Separate from memory bus
- Cache controllers communicate through snoop bus

19.10.2 How It Works

1. Cache controller performs write to address
2. Broadcasts address information on snoop bus
3. All cache controllers listen to snoop bus
4. Controllers check if they have same address cached
5. If yes, take action based on protocol

19.10.3 Key Feature

- All caches monitor (snoop on) the bus

- Detect writes by other processors
- Take appropriate action to maintain coherence

19.11 10. Write Invalidate Protocol

19.11.1 Approach

- When write detected, invalidate own copy
- Group of protocols using this approach
- Most common and easiest to implement

19.11.2 Mechanism

19.11.2.1 On Write by Processor

1. Update own cache
2. Broadcast write address on snoop bus

19.11.2.2 On Receiving Write Broadcast

1. Check if same address in own cache
2. If yes: Mark block as INVALID (clear valid bit)
3. Next access will be miss

19.11.3 With Write-Through Policy

- Memory always has up-to-date value
- On miss after invalidation: Fetch from memory
- Straightforward implementation

19.11.4 With Write-Back Policy

Challenge:

- Only writing cache has up-to-date value
- Memory has stale value
- On miss after invalidation: Cannot fetch from memory

Solution: Snoop Read:

- Cache with invalid block places snoop read request on bus
- Cache controllers listen to snoop read
- Controller with valid up-to-date copy responds
- Supplies data through snoop bus
- More efficient than going to memory
- Avoids slow memory access

19.11.5 Complexity

Trade-offs:

- More complex cache controller
- Snoop bus needs to carry data and addresses
- More hardware required
- Higher power consumption
- But better performance (less memory traffic)

19.12 11. Write Update Protocol

19.12.1 Alternative Approach

Concept:

- Update own copy instead of invalidating
- Also called Write Broadcast
- Different action when write detected

19.12.2 Mechanism

19.12.2.1 On Write by Processor

1. Update own cache
2. Broadcast BOTH address AND data on snoop bus

19.12.2.2 On Receiving Write Broadcast

1. Check if same address in own cache
2. If yes: Update own copy with new data
3. Keep block VALID

19.12.3 Benefits

- No miss on next access to same address
- Data already updated in all caches
- Don't need extra read operation
- Simpler cache controller (no snoop read needed)

19.12.4 Costs

- Snoop bus must carry data (wider bus)
- More hardware on snoop bus
- Higher power consumption
- More bus traffic

19.12.5 Comparison

- Simpler than write invalidate with write-back
- Fewer cache misses
- Higher bus bandwidth requirement

19.13 12. Real Protocol Implementations

19.13.1 Historical Protocols

19.13.1.1 Write Once Protocol

- **Type**: Write invalidate
- **Write policy**: Write-through on first write, write-back after
- One of first bus snooping protocols

19.13.1.2 Synapse N+1 Protocol

- **Type**: Write invalidate
- **Write policy**: Write-back
- Early implementation

19.13.1.3 Berkeley Protocol

- **Type**: Write invalidate
- **Write policy**: Write-back
- Used in Berkeley SPUR processor

19.13.1.4 Illinois Protocol (MESI)

- **Type**: Write invalidate
- **Write policy**: Write-back
- Used in SGI Power and Challenge systems
- Very popular, widely adopted

19.13.1.5 Firefly Protocol

- **Type**: Write update
- **Write policy**: Mixed (write-back for private data, write-through for shared data)
- Used in DEC Firefly and Sun SPARC systems

19.13.2 Most Common Combination

- Write invalidate protocols
- Write-back policy
- Reduces memory accesses (expensive in terms of time)

- Easier to implement than write update
- Good balance of performance and complexity

19.14 13. MESI Protocol Details

Named after four states: **Modified, Exclusive, Shared, Invalid**

Most popular cache coherency protocol, used in Intel Pentium and IBM PowerPC processors.

19.14.1 Four Block States (Requires 2 Bits)

19.14.1.1 1. INVALID (I)

- Data not valid
- Block cannot be used
- Must fetch from elsewhere

19.14.1.2 2. SHARED (S)

- Multiple caches have copies of this block
- All copies have same value
- Value consistent with memory
- Memory has up-to-date value

19.14.1.3 3. EXCLUSIVE (E)

- Only cached copy in entire system
- No other cache has this block
- Value consistent with memory
- Memory has up-to-date value

19.14.1.4 4. MODIFIED (M)

- Only cached copy in system
- Value INCONSISTENT with memory
- This cache has most recent value
- Memory has stale value
- Block is “dirty”

19.15 14. MESI Protocol State Transitions

19.15.1 Example with PE1, PE2, PE3

Initial State: Variable X = 1 in memory, all cache entries invalid

19.15.1.1 Step 1: PE1 Reads X

Actions:

- Check other caches (snoop read request)
- No other cache has X
- Fetch from memory
- **State transition**: Invalid → Exclusive

Result:

- PE1: X = 1 (Exclusive)

19.15.1.2 Step 2: PE3 Reads X

Actions:

- Check other caches (snoop read request)
- PE1 responds (has Exclusive copy)
- PE1 supplies data to PE3

State transitions:

- PE1: Exclusive → Shared
- PE3: Invalid → Shared

Result:

- Both PE1 and PE3: X = 1 (Shared)
- Consistent with memory

19.15.1.3 Step 3: PE3 Writes X = 0

Actions:

- Block in PE3 was Shared
- Update local cache
- Broadcast invalidate on snoop bus
- **State transition**: Shared → Modified

Result:

- PE3: X = 0 (Modified)
- PE1 receives invalidate:
  - State transition: Shared → Invalid
  - PE1: X = ? (Invalid)
- Memory still has X = 1 (stale)

19.15.1.4 Step 4: PE1 Reads X

Actions:

- Block in PE1 is Invalid (tag matches but invalid)
- Place snoop read request on bus
- PE3 has Modified copy (most up-to-date)
- PE3 responds to snoop read:
  - Supplies data to PE1 through snoop bus
  - Writes back to memory
  - State transition: Modified → Shared
- PE1 receives data:
  - State transition: Invalid → Shared

Result:

- PE1: X = 0 (Shared)
- PE3: X = 0 (Shared)
- Memory: X = 0 (updated)
- All consistent

19.15.2 Key Points

- Coherency maintained throughout
- Invalidations prevent stale data reads
- Modified state identifies most recent value
- Snoop reads fetch from other caches efficiently
- Write-backs occur when transitioning from Modified to Shared

---

19.16 15. Scalability of UMA Systems

19.16.1 Limitation

Challenges:

- Bus snooping doesn’t scale well
- Bus contention increases with more processors
- Snoop bus becomes bottleneck
- Memory bus also becomes bottleneck

19.16.2 Practical Limit

- Up to ~32 processing elements with bus-based design
- Not a hard threshold but approximate practical limit
- Beyond this, contention significantly degrades performance

19.16.3 Alternative Interconnects

19.16.3.1 Crossbar Switches

- Alternative to bus-based architecture
- Allows multiple simultaneous connections
- Better than simple bus

19.16.3.2 Multi-Stage Crossbar Switch Network

- Multiple crossbar switches in network topology
- Increased parallelism in interconnect
- Can connect multiple memory banks simultaneously
- Increases scalability

19.16.4 Improved Scalability

- With crossbar networks: Up to ~256 processing elements
- Still limited but much better than bus-based
- Trade-off: More complex hardware

19.17 16. Non-Uniform Memory Access (NUMA)

19.17.1 Designed for Even Higher Scalability

Goals:

- Target: Thousands of processing elements
- Beyond limits of UMA systems
- Still uses shared memory model
- Communication through shared address space

19.17.2 Key Difference from UMA

Non-Uniform Access Times:

- Memory access time DEPENDS on which processor is accessing
- Different processors experience different latencies
- Memory perspective is non-uniform

19.17.3 Architecture

Structure:

- Each processor has local memory
- Faster to access local memory
- Slower to access remote memory (other processors' local memory)

- But all memory accessible by all processors (shared address space)

19.17.4 Access Time Difference

- Remote memory access: 4-5 times more cycles than local
- Significant performance impact
- Programming must consider locality

19.17.5 Operating System Role

Optimization Responsibilities:

- Must use special algorithms for memory optimization
- Workload distribution affects performance
- Should relocate memory blocks for optimization
- Goal: Maximize local accesses, minimize remote accesses
- Global optimization problem

19.18 17. Two Types of NUMA

19.18.1 1. NC-NUMA (Non-Cached NUMA)

Characteristics:

- No caches shown in architecture
- Processors directly access memory
- Simpler but slower

19.18.2 2. CC-NUMA (Cache-Coherent NUMA)

Characteristics:

- Includes caches at each node
- Must maintain cache coherence
- More complex but better performance
- Cannot use bus snooping (not scalable enough)
- Solution: Directory-based coherence

19.19 18. Directory-Based Cache Coherence

Used in CC-NUMA systems for scalable cache coherence.

19.19.1 What is Directory?

Definition:

- Data structure tracking cache contents
- Distributed across system
- Stores information about which blocks are cached where
- Can be in memory or separate hardware

19.19.2 Purpose

Functionality:

- Cache controllers check directory to find block locations
- Determines if other caches have copies of block
- Enables coherence without bus snooping
- Scalable to thousands of processors

19.19.3 Organization

Distributed Structure:

- Directory can be distributed
- Each node has local directory
- Local directory tracks blocks from local memory address range
- Information about which caches have those blocks
- Blocks from other address ranges tracked in other directories

19.19.4 Operation

Access Process:

- Cache controller accesses appropriate directory
- Local directory if accessing local address range
- Remote directory if accessing remote address range
- Directory provides information about block locations
- Can then send invalidations or updates as needed

19.19.5 Write Policy

- Typically use write-through policy

19.20 Key Takeaways

1. Multiprocessors overcome single-processor performance limitations
2. Shared memory provides communication mechanism between processors
3. Cache coherence is essential for correct parallel program execution

4. Bus snooping works well for small-scale systems (up to ~32 processors)
5. MESI protocol is widely adopted for cache coherence
6. UMA systems provide uniform access but limited scalability
7. NUMA systems enable thousands of processors with non-uniform access
8. Directory-based coherence enables scalable cache coherence
9. Operating system plays crucial role in workload balancing and optimization
10. Trade-offs exist between simplicity, performance, and scalability

---

19.21 Summary

Multiprocessor systems have become the standard in modern computing, from smartphones to supercomputers, enabling the parallel processing power required for contemporary applications while managing the complex interactions between multiple processors sharing memory resources.

Chapter 20

Lecture 20: Storage and Input/Output Systems

20.1 Introduction

This lecture completes our exploration of computer architecture by examining storage devices and input/output (I/O) systems that enable computers to interact with external devices and provide persistent data storage beyond volatile main memory. We explore storage technologies from mechanical magnetic disks to solid-state flash memory, understanding their performance characteristics, reliability metrics, and cost tradeoffs. The lecture covers I/O communication methods including polling, interrupts, and direct memory access (DMA), analyzes RAID configurations that improve both performance and dependability, and examines how storage systems connect to processors through memory-mapped I/O or dedicated I/O instructions. Understanding these peripheral systems reveals how complete computer systems integrate computation, memory, and external interaction into cohesive platforms.

20.2 1. Introduction to I/O Devices and Storage

Chapter 6 covers storage devices and input/output systems that enable computers to interact with external devices and persistent storage.

20.3 2. I/O Device Characteristics

I/O devices can be characterized by three fundamental factors:

20.3.1 1. Behavior

Input Devices:

- Provide data to system
- Examples: keyboards, mice, sensors

Output Devices:

- Receive data from system
- Examples: displays, printers, speakers

Storage Devices:

- Store and retrieve data
- Examples: disks, flash drives

20.3.2 2. Partner

Human Devices:

- Communicate with humans
- Examples: keyboards, displays, audio

Machine Devices:

- Communicate with other machines
- Examples: networks, controllers

20.3.3 3. Data Rate

- Measured in bytes per second or transfers per second
- Wide variation across device types
- Affects system design and communication methods

20.4 3. I/O Bus Connections

20.4.1 Simplified System Architecture

20.4.1.1 Components

- **Processor (CPU)**
- **Cache**
- **Memory I/O Interconnect (Bus)**
- **Main Memory**
- **Multiple I/O Controllers**
- **Various I/O Devices**

20.4.1.2 Bus Structure

- Processor and cache connected to bus
- Main memory connected to bus
- I/O controllers connected to bus

- Each controller manages specific devices

20.4.1.3 Connections

- Processor receives interrupts from bus/devices
- **I/O Controller 1**: Connected to disk
- **I/O Controller 2**: Connected to graphic output
- **I/O Controller 3**: Connected to network channel

Multiple controllers allow parallel device operation while sharing common interconnect.

20.5 4. Dependability

Critical for I/O systems, especially storage devices.

20.5.1 Why Dependability Matters

- Storage devices hold data that must be reliable
- Users depend on devices being available
- Data loss is unacceptable
- Systems must continue functioning despite component failures

20.5.2 Dependability is Particularly Important For

- Storage devices (data integrity)
- Critical systems (servers, embedded systems)
- Systems with high availability requirements

20.6 5. Service States

20.6.1 Two Primary States

20.6.1.1 1. Service Accomplishment State

- Device is working correctly
- Providing expected service
- Normal operational state

20.6.1.2 2. Service Interruption State

- Device has failed
- Not providing service
- Requires repair/restoration

20.6.2 State Transitions

- **From Service Accomplishment to Service Interruption**: Due to failure
- **From Service Interruption to Service Accomplishment**: After restoration/repair

---

20.7 6. Fault Terminology

20.7.1 Fault Definition

Characteristics:

- Failure of a component
- May or may not affect the system
- May or may not lead to system failure
- System can continue running with faulty component
- May produce correct or wrong output

20.7.2 Distinction

- **Component failure ≠ System failure**
- Fault tolerance allows operation despite faults

20.8 7. Dependability Measures

20.8.1 Key Metrics

20.8.1.1 1. MTTF (Mean Time To Failure)

Definition:

- Reliability measure
- Average time device operates before failing
- Measures how long system stays in Service Accomplishment state
- Higher MTTF = more reliable

20.8.1.2 2. MTTR (Mean Time To Repair)

Definition:

- Service interruption measure
- Average time to restore service after failure
- How long device stays in Service Interruption state
- Lower MTTR = faster recovery

20.8.1.3 3. MTBF (Mean Time Between Failures)

Formula:

```
MTBF = MTTF + MTTR
```

Definition:

- Complete cycle: operation + repair
- Time from one failure to next failure
- Includes both operational and repair time

20.8.1.4 4. Availability

Formula:

```
Availability = MTTF / (MTTF + MTTR)
```

Definition:

- Proportion of time machine is available
- Ratio of operational time to total time
- Expressed as percentage or decimal

20.9 8. Improving Availability

20.9.1 Two Approaches

20.10 9. 1. Increase MTTF (Mean Time To Failure)

20.10.0.1 a) Fault Avoidance

Methods:

- Prevent faults before they occur
- Better design and manufacturing
- Quality components
- Proper operating conditions

20.10.0.2 b) Fault Tolerance

Methods:

- Design system to withstand faults
- Redundancy (duplicate components)
- Error correction mechanisms
- Graceful degradation

20.10.0.3 c) Fault Forecasting

Methods:

- Predict when faults will occur
- Preventive maintenance
- Monitor component health
- Replace before failure

20.11 10. 2. Reduce MTTR (Mean Time To Repair)

20.11.1 Methods

- Improve tools and processes for diagnosis
- Better diagnostic capabilities
- Easier repair procedures
- Quick replacement mechanisms
- Automated recovery systems
- Skilled maintenance personnel

20.11.2 Example Problems

- Book provides examples with specific MTTF and MTTR values
- Calculate availability
- Analyze improvement strategies
- Students should practice these calculations

20.12 11. Magnetic Disk Storage

Traditional secondary storage technology using magnetic recording.

20.12.1 Physical Structure

20.12.1.1 Disk Shape

- Circular/round shape
- Platter rotates on spindle

20.12.1.2 Tracks

- Concentric circles on disk surface
- From periphery (outer edge) to center
- Multiple tracks like ribbons arranged concentrically
- Similar to running tracks in sports (Olympics)

20.12.1.3 Sectors

- Tracks divided by radial lines (from center to periphery)
- Cross-sectional cuts across tracks
- Portion between two separation lines = one sector
- Smallest addressable unit on disk

20.12.2 Sector Contents

- **Sector ID** (identification)
- **Data** (512 bytes to 4096 bytes typical)
- **Error Correcting Code (ECC)**
  - Hides defects
  - Corrects recording errors
- **Gaps** between sectors (unused spaces)

20.13 12. Disk Access Process

20.13.1 Access Components and Timing

20.13.1.1 1. Queuing Delay

- If other accesses are pending
- Wait for previous operations to complete
- Managed by disk controller

20.13.1.2 2. Seek Time

- Moving head to correct track
- Head positioned on right sector
- Physical movement of read/write head
- Head placed diagonally on disc
- Time to “seek” the target sector
- Typically several milliseconds

20.13.1.3 3. Rotational Latency

- Rotating disk to position correct sector under head
- Disk spins to align sector with head
- Choose closest direction (shortest rotation)
- Sectors arranged diagonally on disk
- Multiple sectors per track
- Can rotate either direction (clockwise or counterclockwise)

20.13.1.4 4. Transfer Time

- Actual data read/write
- Depends on sector size and transfer rate
- Usually small compared to seek and rotation

20.13.1.5 5. Controller Overhead

- Processing by disk controller
- Command interpretation
- Error checking
- Generally small (fraction of millisecond)

20.13.2 Access Coordination

- Processor initiates access
- Memory Management Unit (MMU) handles translation
- Involves both hardware and operating system
- Reading page from disk to memory: millions of cycles
- Much slower than memory access

20.14 13. Disk Access Example Calculation

20.14.1 Given Parameters

- **Sector size**: 512 bytes
- **Rotational speed**: 15,000 RPM (rotations per minute)
- **Seek time**: 4 milliseconds
- **Transfer rate**: 100 MB/s
- **Controller overhead**: 0.2 milliseconds
- Assume idle disk (no queuing)

20.14.2 Average Read Time Calculation

20.14.2.1 1. Seek Time

4 ms (given)

20.14.2.2 2. Rotational Latency

- Average = Half rotation time
- Full rotation = 60 seconds / 15,000 RPM = 4 ms
- Average = 4 ms / 2 = **2 ms**
- Why half? Can choose closest direction

20.14.2.3 3. Transfer Time

- Size / Rate = 512 bytes / 100 MB/s
- = **0.005 ms**

20.14.2.4 4. Controller Delay

0.2 ms (given)

20.14.3 Total Average Read Time

```
Total = 4 + 2 + 0.005 + 0.2 = 6.2 milliseconds
```

20.14.4 Real Case Variation

- Actual average seek time might be 1 ms (not 4 ms)
- Depends on:
  - Which sector being accessed
  - Current head position
  - Distance head must travel
- With 1 ms seek: Total = **3.2 ms**
- Significant variation based on access patterns

20.14.5 Additional Examples

- Book provides more practice problems
- Students should try different scenarios
- Understand impact of each component on total time

20.15 14. Flash Storage

Modern non-volatile semiconductor storage technology.

20.15.1 Characteristics

20.15.1.1 Advantages

- Non-volatile (retains data without power)
- 1000x faster than magnetic disk
- Smaller physical size
- Lower power consumption
- More robust (no moving parts)
- Can be carried around easily
- Shock resistant

20.15.1.2 Disadvantages

- More expensive than magnetic disk
- Limited write cycles (wears out over time)
- Technology cost higher

20.16 15. Types of Flash Storage

20.16.1 1. NOR Flash

20.16.1.1 Structure

- Bit cell like NOR gate
- Random read/write access
- Can access individual bytes

20.16.1.2 Characteristics

- Byte-level access
- Faster read access
- More expensive

20.16.1.3 Applications

- Instruction memory in embedded systems
- Code storage
- Execute-in-place applications

20.16.2 2. NAND Flash

20.16.2.1 Structure

- Bit cell like NAND gate
- Block-at-a-time access
- Cannot access individual bytes directly

20.16.2.2 Characteristics

- Denser (more storage per area)
- Block-level access
- Reading and writing done in blocks
- Cheaper per GB

20.16.2.3 Applications

- USB keys/drives
- Media storage (photos, videos)

- Solid-state drives (SSDs)
- Memory cards

Note: Values in lecture slides may be outdated as flash storage technology rapidly evolves.

20.17 16. Memory-Mapped I/O

Method of accessing I/O devices using memory addresses.

20.17.1 Concept

- Reserve some address space for I/O devices
- I/O device registers appear as memory locations
- Same address space as memory
- Address decoder distinguishes between memory and I/O

20.17.2 Example with 8 Address Lines

- **Total addressable locations**: 256 (2^8)
- **Reserve 128 locations for memory**
- **Reserve 128 locations for I/O devices**
- Same load/store instructions access both

20.17.3 Access Mechanism

- Use load/store instructions for both memory and I/O
- Operating system controls access
- Uses address translation mechanism
- Can make I/O addresses accessible only to kernel
- Protection mechanism prevents user programs from direct access

20.17.4 Advantages

- Unified programming model
- Same instructions for memory and I/O
- Simpler instruction set

20.17.5 Disadvantages

- Reduces available memory address space
- Must reserve addresses for I/O

20.18 17. I/O Instructions

Alternative to memory-mapped I/O: separate I/O instructions.

20.18.1 Characteristics

- Separate instructions specifically for I/O operations
- Distinct from load/store (memory) instructions
- Can duplicate addresses:
  - Same address can refer to memory location
  - Same address can refer to I/O device
  - Instruction type determines which is accessed

20.18.2 Access Control

- I/O instructions can only execute in kernel mode
- User programs cannot directly access I/O
- Protection mechanism
- Operating system mediates I/O access

20.18.3 Example Architecture

- **x86 (Intel/AMD processors)**
- Has special IN and OUT instructions for I/O
- Separate I/O address space

20.18.4 Advantages

- Full memory address space available
- No address space conflict
- Clear distinction between memory and I/O

20.18.5 Disadvantages

- More complex instruction set
- Additional instructions needed

20.19 18. Polling

Method for processor to communicate with I/O devices.

20.19.1 How Polling Works

20.19.1.1 1. Periodically Check I/O Status Register

- Processor repeatedly reads device status
- Check if device is ready
- Continuous monitoring in loop

20.19.1.2 2. If Device Ready

- Perform requested operation
- Read data or write data
- Continue with next task

20.19.1.3 3. If Error Detected

- Take appropriate action
- Error handling
- Retry or report error

20.19.2 Characteristics

20.19.2.1 When Used

- Small or low-performance systems
- Real-time embedded systems
- Simple applications

20.19.2.2 Advantages

Predictable Timing:

- Know exactly when device checked
- Deterministic behavior
- Important for real-time systems

Low Hardware Cost:

- Software handles communication
- No additional hardware needed
- Simple implementation

20.19.2.3 Disadvantages

Wastes CPU Time:

- CPU continuously loops checking device
- Can’t do other work while polling
- Inefficient for high-performance systems

Not Suitable for Complex Systems:

- Multiple devices difficult to manage
- CPU time wasted on idle devices

20.19.3 Programming Model

- Can write program to:
  - Read status bit from device
  - Check if device free
  - Make decisions based on status
- Simple control flow

20.20 19. Interrupts

Alternative to polling: device-initiated communication.

20.20.1 How Interrupts Work

20.20.1.1 1. Device Initialization

- Device sends signal/request to processor
- Request for service
- Happens when device ready or error occurs

20.20.1.2 2. Controller Interrupts CPU

- Device controller signals processor
- Processor stops current work
- Handles interrupt

20.20.1.3 3. Handler Execution

- Special interrupt handler routine runs
- Services device request
- Returns to original program

20.20.2 Characteristics

20.20.2.1 Asynchronous

- Not synchronized to instruction execution
- Unlike exceptions (which are synchronous)
- Can occur between any two instructions
- Handler invoked between instructions

20.20.2.2 Fast Identification

- Interrupt often identifies device
- Know which device needs service
- Can be handled quickly

20.20.2.3 Priority System

- Not all devices have same urgency
- Devices categorized by priority levels
- Devices needing urgent attention get higher priority
- High-priority interrupts can preempt low-priority handlers

20.20.3 Advantages

Efficient CPU Use:

- No wasted time polling
- CPU does other work until interrupt

Good for Multiple Devices:

- Each device interrupts when ready
- No continuous checking needed

Responsive:

- Quick response to device events

20.20.4 Disadvantages

More Complex Hardware:

- Interrupt controller needed
- Priority management

Context Switching Overhead:

- Save/restore processor state
- Handler invocation takes time

20.20.5 Execution Model

- Main program running
- Instruction completes
- Interrupt checked
- If interrupt pending:
  - Current state saved
  - Interrupt handler runs
  - State restored
  - Resume main program at next instruction

20.21 20. I/O Data Transfer Methods

Three approaches for transferring data between memory and I/O:

20.22 21. 1. Polling-Driven I/O

20.22.1 Process

- CPU polls device repeatedly
- When ready, CPU transfers data
- CPU moves data between memory and I/O registers

20.22.2 Issues

- Time consuming
- CPU fully involved in transfer
- Inefficient for high-speed devices
- Wastes CPU cycles

20.23 22. 2. Interrupt-Driven I/O

20.23.1 Process

- Device interrupts when ready
- CPU services interrupt
- CPU transfers data between memory and I/O registers

20.23.2 Issues

- Still CPU-intensive for data transfer
- CPU must move every byte
- Better than polling but still inefficient for bulk transfers

20.24 23. 3. Direct Memory Access (DMA)

20.24.1 Process

Setup:

- DMA controller handles transfer
- Removes CPU from data movement
- Processor hands off transfer job to DMA controller
- DMA controller transfers data autonomously

20.24.2 DMA Operation

CPU Provides:

- Starting address in memory
- Transfer size
- Direction (memory→device or device→memory)

DMA Controller:

- Transfers data independently
- Operates in parallel with CPU
- No CPU intervention during transfer

Controller Interrupts CPU On:

- Completion of transfer
- Error occurrence

20.24.3 Advantages

- CPU free to do other work
- Efficient bulk data transfers
- Essential for high-speed devices
- Reduces CPU overhead significantly

20.24.4 When Used

- High-speed devices (disks, network)
- Large data transfers
- When CPU time is valuable

20.24.5 Comparison

- **Polling**: Simple, predictable, inefficient
- **Interrupts**: Responsive, better than polling, CPU still involved in transfer
- **DMA**: Most efficient, essential for high-performance I/O

20.25 24. RAID (Redundant Array of Independent Disks)

Technology to improve storage performance and dependability.

20.25.1 Purpose

- Improve performance through parallelism
- Improve dependability through redundancy
- Use multiple disks together as single logical unit

20.25.2 Benefits

20.25.2.1 Performance Improvement

- Parallel access to multiple disks
- Higher throughput
- Faster data access

20.25.2.2 Dependability Improvement

- Redundancy protects against disk failure
- Data not lost if one disk fails
- Improved reliability

Note: Lecture mentions RAID but doesn't go into detailed levels or configurations. This is a complex topic covered more thoroughly in other courses.

20.26 25. Conclusion and Summary

20.26.1 I/O System Performance Measures

- **Throughput**: Amount of data transferred per unit time
- **Response time**: Time from request to completion
- **Dependability**: Reliability and availability
- **Cost**: Important consideration in system design

20.27 26. Key Points

20.27.1 System Architecture

- Buses connect CPU, memory, and I/O controllers
- Multiple controllers manage different device types
- Shared interconnect with arbitration

20.27.2 Communication Mechanisms

- **Polling**: Simple, predictable, inefficient
- **Interrupts**: Responsive, efficient CPU use

- **DMA**: Most efficient for bulk transfers

20.27.3 Dependability

- Critical for storage systems
- Measured by MTTF, MTTR, MTBF, availability
- Improved through fault tolerance and faster repair

20.27.4 Storage Technologies

- **Magnetic disk**: Traditional, slower, cheaper per GB, mechanical
- **Flash storage**: Modern, faster, more expensive, no moving parts

20.27.5 Access Methods

- **Memory-mapped I/O**: I/O uses memory address space
- **I/O instructions**: Separate instruction set for I/O

20.27.6 Performance Optimization

- **RAID**: Improve performance and dependability
- **DMA**: Reduce CPU overhead
- **Interrupts**: Improve responsiveness

20.27.7 Exercises

- Book contains practice problems
- Calculate dependability metrics
- Analyze disk access times
- Compare different I/O mechanisms

20.28 Key Takeaways

1. I/O systems connect computers to external devices and storage
2. Dependability is critical for storage systems
3. MTTF, MTTR, and availability are key metrics
4. Magnetic disks use mechanical components with millisecond access times
5. Flash storage is faster but more expensive than magnetic storage
6. Memory-mapped I/O and separate I/O instructions are two access methods
7. Polling is simple but inefficient
8. Interrupts improve CPU efficiency
9. DMA is essential for high-speed bulk data transfers
10. RAID improves both performance and reliability

20.29 Summary

This concludes the processor and memory sections of the course, covering the complete spectrum from CPU design through memory hierarchy to I/O systems. We have explored how computers are designed from the ground up, from basic arithmetic operations through pipelined execution, memory hierarchies, multiprocessor systems, and finally to storage and I/O mechanisms that enable computers to interact with the external world.