Lectures on Computer Architecture

By Isuru Nawinne

Lectures on Computer Architecture

By Isuru Nawinne

ISBN 978-624-92913-0-0

Downloadable ebook and supplementary material available at
https://cepdnaclk.github.io/Computer-Architecture-Web

Publisher:
Dr. Isuru Nawinne,
Department of Computer Engineering, Faculty of Engineering,
University of Peradeniya,
Peradeniya 20400,
Sri Lanka.
isurunawinne@eng.pdn.ac.lk https://people.ce.pdn.ac.lk/staff/academic/isuru-nawinne/

Content

Preface

Computer architecture sits at the heart of modern computing. It is the discipline that reveals how machines execute instructions, manage data, and achieve programmability—bridging the conceptual world of algorithms and the physical realities of hardware. Over many years of teaching this subject to undergraduate students, I have found that curiosity grows not only from understanding what computers do, but from discovering how they do it and why they're designed that way.

This book, *Lectures on Computer Architecture*, is built upon the lecture series delivered to undergraduate cohorts. Each section distills core ideas, clarifies subtle concepts, and connects theory to the practical systems. The video lectures grew from classroom sessions, refined through questions, discussions, and repeated teaching experience. The accompanying notes are designed to complement the videos rather than duplicate it, offering multiple modalities through which students can explore the subject.

My goal is to provide a learning resource that is rigorous yet approachable, structured yet flexible, and suitable for both guided instruction and independent study. Whether used as a primary course text, or a guide for self-study, I hope this book supports students in creating a solid cognitive model of how computers are built.

I am grateful to all my students over the years whose questions and feedback helped refine these explanations, and to everyone who encouraged the development of a resource that unifies both lecture and text. It is my hope that this book helps you to see computer architecture not merely as a subject to be completed, but as a foundation for understanding the modern machines that shape our world.

I would like to convey my sincere appreciation to Dr. Kisaru Liyanage and Dr. Swarnalatha Radhakrishnan for their valuable contributions in delivering selected lectures. My profound thanks go to Kanishka Gunawardana and Sanka Peeris, for helping me edit this book and setting up the interactive web version. Their careful attention to detail, thoughtful feedback, and commitment to ensuring the clarity and accuracy have contributed greatly to the quality and reliability of this work.

Isuru Nawinne
Senior Lecturer in Computer Engineering

Learning Methods

This book is designed to support active, independent, and flexible learning, aligning closely with flipped learning and self-directed study practices.

The flipped-learning approach encourages students to engage with key concepts before coming to class or attempting exercises. Each section in this book includes a corresponding video lecture that introduces the fundamental ideas, explains core mechanisms, and walks through examples. Watching the video beforehand allows learners to arrive at discussions or problem-solving sessions better prepared, able to ask informed questions, and ready to dive deeper.

Flipped-learning transforms the role of classroom or study time: instead of passively receiving information, students actively seek and apply it. With multiple modalities of the videos as the initial exposure and the book as a reference and reinforcement tool, learners can use their interactive time: whether in discussions; tutorials; labs; or group study; to focus on reasoning, analysis, and synthesis.

Computer architecture is a subject that rewards curiosity and exploration. To support self-directed learning, each chapter is structured so students can progress at their own pace. The notes are carefully layered. They begin with foundational principles and incrementally build toward more advanced ideas.

Students are encouraged to:

- Watch the video lectures as many times as needed to internalize concepts;
- Revisit diagrams and derivations to strengthen visual and mathematical intuition;
- Use end-of-section summaries and conceptual checkpoints to evaluate their understanding; and
- Make connections between topics—for example, how pipelining interacts with branching, or how memory hierarchy influences performance.

This style of learning builds autonomy, critical thinking, and long-term retention—key skills for an engineer.

Introduction

This book follows a gradual progression from fundamental concepts to advanced architectural mechanisms, mirroring the structure of the lecture series. The material is organized into twenty sections, each corresponding to a major topic typically covered in an undergraduate computer architecture course.

How the Book Is Organized

The first set of chapters: Computer Abstractions, Technology Trends, and Performance establish the context and quantitative foundation needed to reason about architectural decisions. These are followed by chapters on Assembly Language Programming, Number Representation, Branching, Function Calls, and Memory Access which build the low-level understanding of how instructions operate.

Midway through the book, the focus shifts to the execution engine itself: Microarchitecture, Datapath, Control, and the progression from Single-Cycle Execution to Pipelined Processors. The chapters such as Pipeline Analysis help students understand real-world engineering challenges.

The later sections explore the memory subsystem in depth: Memory Hierarchy, Caching, Direct Mapped and Associative Cache Control, Multi-Level Caches, and Virtual Memory, before extending the architectural view to Multiprocessors, Storage, and Interfacing.

Each chapter includes:

- A complete video lecture that introduces and explains concepts
- Written notes highlighting definitions, diagrams, examples, and reasoning steps
- Clarifications of common misconceptions
- Connections to earlier and later material
- Guidance on how the topic relates to real processors and modern systems

How to Use the Videos and Notes

The recommended learning sequence is:

1. Start with the video lecture to gain an intuitive, big-picture understanding.
2. Read the notes from the corresponding chapter to clarify details, solidify concepts, and explore more formal explanations.
3. Revisit the video or specific parts of the chapter if some ideas feel unclear—the two formats reinforce each other.
4. Use diagrams and worked examples as anchors for your understanding; architecture is highly

visual and spatial.

5. Progress through sections sequentially, as many topics build directly on earlier ones.

For review, you may find it helpful to skim chapter summaries and rewatch short segments of the videos rather than re-reading entire chapters.

Chapter 1

Fundamentals

1.1 Lecture 1: Computer Abstractions and Technology

By Isuru Nawinne

1.1.1 Introduction

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

1.1.2 The Big Picture of Computer Systems

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

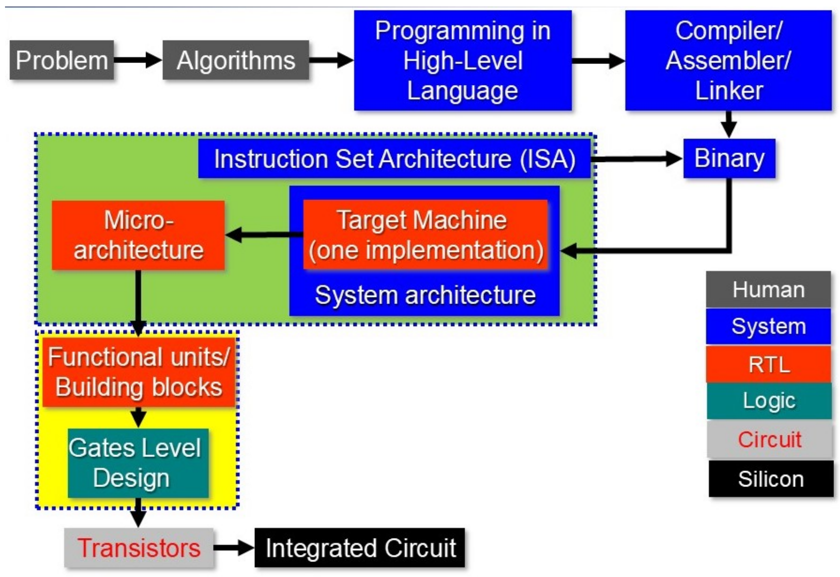

Problem
Algorithms
Programming in High-Level Language
Compiler/ Assembler/ Linker
Instruction Set Architecture (ISA)
Binary
Micro-architecture
Target Machine (one implementation)
System architecture
Functional units/ Building blocks
Gates Level Design
Transistors
Integrated Circuit
Human
System
RTL
Logic
Circuit
Silicon

Figure 1.1: Computer System Abstraction Layers

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.*

Human-Related Level (Gray)

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

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

RTL (Register Transfer Level) - Red/Orange

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

Logic Level (Green)

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

Circuit Level (Light Gray)

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

Substrate Level (Black)

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

Purpose of Computer Systems

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

1.1.3 Instruction Set Architecture (ISA) - The Key Interface

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

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

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

1.1.4 From Problem to Execution - The Translation Chain

High-Level Process

Problem $\rightarrow$ Algorithm $\rightarrow$ Programming Language (C, Python, etc.)

$\downarrow$

Compiler (translates to assembly code)

$\downarrow$

Assembler (translates to machine code)

$\downarrow$

Linker (combines with libraries)

$\downarrow$

Machine Code / Binary Image

$\downarrow$

Runs on Microarchitecture (CPU)

Tool Chain Components

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

Assembler

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

Linker

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

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

1.1.5 Writing Programs at Different Levels

Machine Code (Binary)

Characteristics:

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

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

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

1.1.6 Microarchitecture Details

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

Hierarchy of Microarchitecture Components

Microarchitecture Level

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

Functional Units Level

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

Logic Gate Level

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

Transistor Level

- **Purpose**: Manipulates voltages and currents
- **Types**: BJT, CMOS
- Built using semiconductors

Semiconductor Level

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

1.1.7 Abstraction Concept

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

Hardware Abstraction Hierarchy (Bottom to Top)

1. Substrate (Silicon, Germanium)

- Base semiconductor material

2. Transistors

- Built using semiconductor substrate
- Deal with voltage levels

3. Logic Gates

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

4. Functional Units

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

5. Microarchitecture

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

Software Abstraction Hierarchy (Bottom to Top)

1. Machine Instructions (Binary)

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

2. Assembly Instructions

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

3. Programs / Source Code

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

4. Algorithms and Data Structures

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

Relationships Between Hardware and Software Abstractions

Voltage Levels $\leftrightarrow$ 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)

Logic Levels ↔ Numbers

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

Numbers ↔ Instructions

- Instructions represented as binary numbers
- Microarchitecture interprets these numbers

Summary of Relationships

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

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

1.1.8 Performance Theme

Throughout the Lecture Series

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

1.1.9 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 lecture series

1.1.10 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 lecture series, 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.

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

By Dr. Isuru Nawinne

1.2.1 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.

1.2.2 Moore's Law - Foundation of Computer Technology Evolution

Who Was Gordon Moore?

Background and Influence:

- Co-founder of Intel Corporation, 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

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

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

1. **World Wide Web and Internet Search:**

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

1. **Artificial Intelligence and Machine Learning:**

- Neural networks with billions of parameters
- Real-time image and speech recognition
- Autonomous systems and decision-making

1. **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

1.2.3 Technology Scaling - Historical Data

Transistor Count Growth (1970-2010)

Chart Analysis:

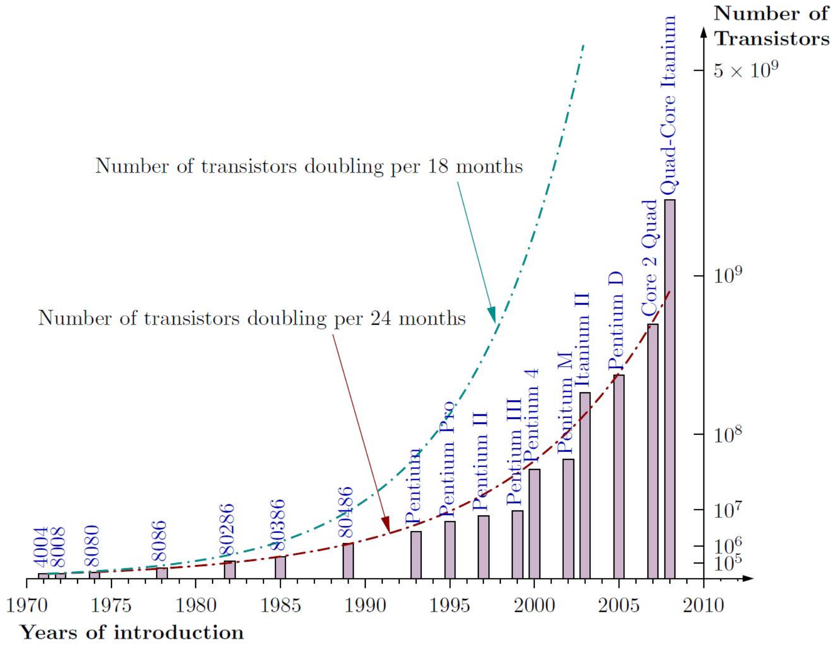


Figure 1.2: Moore's Law: Transistor Count Growth

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

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

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

1.2.4 Feature Size Scaling - Lithography Improvements

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
- **2013 Actual:** 22 nm achieved
- **2015 Target:** 16 nm achieved
- **2019 Target:** 12 nm achieved
- **2023 Target:** 7 nm achieved
- **2028 Target:** 5 nm exceeded
- **Future Roadmap:** 3nm and 2nm are currently in production, future is 1nm and sub-1nm

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

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

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

1.2.5 Technology Roadmaps - ITRS Predictions

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

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

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 Observations:

- **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

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

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

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)

1.2.6 Why Smaller Transistors Improve Performance

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

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 $\rightarrow$ Higher possible clock rate
- Higher clock rate $\rightarrow$ 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

1.2.7 Clock Rate Trends - The Power Wall

Clock Rate Increases (1982-2004)

Exponential Growth Era:

Processor clock frequencies increased dramatically for over two decades:

Historical Progression:

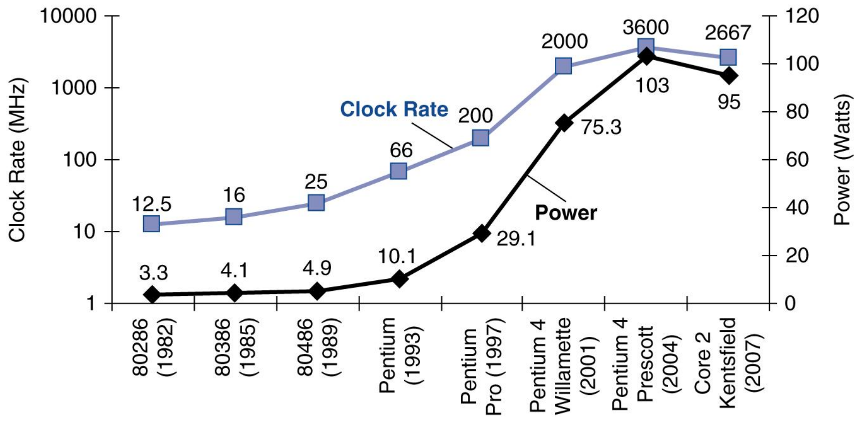


Figure 1.3: Clock Rate Trends and the Power Wall

- **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

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

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

1. **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

Dynamic Power Equation

The Physics of Power Consumption:
Dynamic power consumption follows this relationship:
Power = Capacitance Load $\times$ Voltage^2 $\times$ 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:

$$\begin{aligned}
\text{Power Scaling} &= (\text{Capacitance}) \times (\text{Voltage}^2) \times (\text{Frequency}) \\
&= (1\times) \times (\frac{1}{5})^2 \times (300\times) \\
&= (1\times) \times (\frac{1}{25}) \times (300\times) \\
&= 12 \times \text{ power increase}
\end{aligned}$$

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

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

1.2.8 Shift to Multi-Core Processors

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?

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)

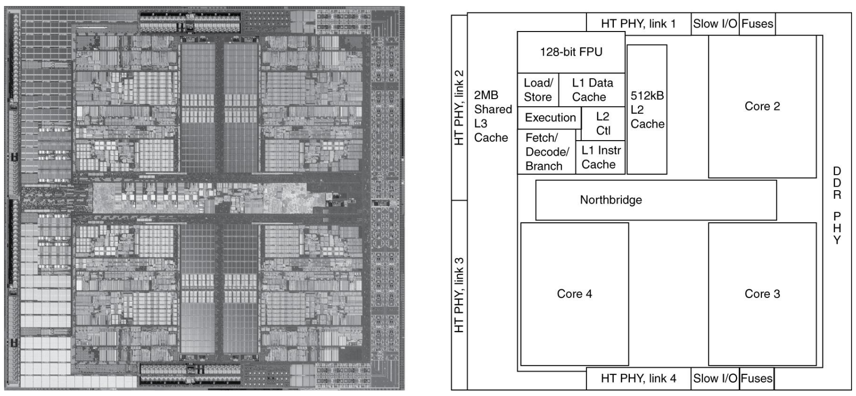


Figure 1.4: AMD Barcelona Quad-Core Processor

- 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

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

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

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

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)

1.2.9 Computer System Organization - Three Layers

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

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

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

1.2.10 From High-Level Code to Machine Code - The Translation Process

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]

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

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

1.2.11 Program Execution - Inside the CPU

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

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

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)

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 Lecture:

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

1.2.12 Real CPU Layout - AMD Barcelona Example

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

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

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

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

Additional Information

WikiChip Database: 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

1.2.13 Key Takeaways

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

1. **Smaller transistors enabled by improved lithography** - progression from 90nm $\rightarrow$ 45nm $\rightarrow$ 22nm $\rightarrow$ 7nm $\rightarrow$ 5nm through advancing manufacturing processes

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

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

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

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

1. **Dynamic power equation explains the crisis** - despite 25$\times$ power reduction from voltage scaling, 300$\times$ frequency increase overwhelmed the benefit

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

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

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

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

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

1. **3D chip technology changed scaling paradigm (2013-2015)** - industry shifted from pure 2D shrinking to vertical stacking of transistor layers

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

1. **Computer systems organized in three layers** - hardware (physical components), system software (OS, compilers, tools), application software (user programs)

1. **System software provides abstraction and protection** - OS prevents malicious programs from damaging hardware, hides complexity from application programmers

1. **Program translation is multi-stage process** - high-level language $\rightarrow$ assembly language $\rightarrow$ machine code through compiler, assembler, linker

1. **CPU contains datapath and control** - datapath performs computation by routing data through functional units; control coordinates execution and generates signals

1. **Cache memory critical for performance** - fast on-chip memory (L1, L2, L3) stores frequently accessed data/instructions, hiding main memory latency

1. **Real CPUs have complex layouts** - die photos reveal intricate organization with multiple cores, cache hierarchies, shared interconnects, memory controllers

1.2.14 Summary

This lecture provides a comprehensive examination of computer technology evolution from the 1970s to present day. Moore's Law, predicting transistor count doubling every two years, serves as the guiding principle for the semiconductor industry and enables 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.

1.3 Lecture 3: Understanding Performance

By Dr. Isuru Nawinne

1.3.1 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.

1.3.2 Defining and Measuring Performance

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

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

1.3.3 CPU Time and Performance Factors

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

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

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

1.3.4 Understanding CPI in Detail

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

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

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

1.3.5 Performance Optimization Principles

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

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)]

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.

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×.

1.3.6 Complete Performance Analysis

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

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/sec) = 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/sec) = 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.

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 × Voltage^2 × Frequency
- Mobile systems prioritize power over peak performance

1.3.7 Practical Performance Considerations

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

Performance Metrics in Practice

MIPS (Million Instructions Per Second)

MIPS = Instruction Count / (Execution Time $\times 10^6$) = Clock Rate / (CPI $\times 10^6$)

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

Power and Energy Considerations

Power Wall

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

Dynamic Power Equation
$\text{Power} = \text{Capacitance} \times \text{Voltage}^2 \times \text{Frequency}$
Energy Equation
$\text{Energy} = \text{Power} \times \text{Time}$
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

1.3.8 Key Takeaways

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

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

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

1. **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.

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

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

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

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

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

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

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

1.3.9 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 2

ARM Assembly Programming

2.1 Lecture 4: Introduction to ARM Assembly

By Dr. Kisaru Liyanage

2.1.1 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.

2.1.2 ARM Architecture Overview

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)

| Feature | RISC | CISC | |———|——|——| | **Instruction Format** | Simple, uniform format | Variable-length instructions | | **Instruction Complexity** | Simple instructions, more instructions per program | Complex operations | | **Memory Access** | Load/store architecture (only LOAD/STORE access memory) | Memory operands in arithmetic operations | | **Registers** | Large number of general-purpose registers | Fewer registers | | **Hardware Design** | Hardware simplicity for higher clock rates | More complex hardware | | **Pipelining** | Regular structure enables efficient pipelining | More difficult to pipeline |

ARM Design Principles

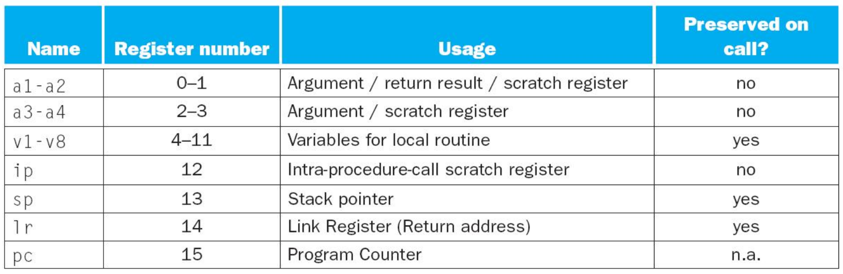
- Simplicity enables high performance
- Regular instruction encoding aids decoding
- Load/store architecture simplifies memory access
- Large register file reduces memory traffic
- Consistent design across instruction types

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



Name	Register number	Usage	Preserved on call?
a1-a2	0–1	Argument / return result / scratch register	no
a3-a4	2–3	Argument / scratch register	no
v1-v8	4–11	Variables for local routine	yes
ip	12	Intra-procedure-call scratch register	no
sp	13	Stack pointer	yes
lr	14	Link Register (Return address)	yes
pc	15	Program Counter	n.a.

Figure 2.1: ARM Register 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

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
```

textbfWord Alignment

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

textbfAddress Space

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

2.1.3 ARM Instruction Format

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)

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

textbfData Transfer Instructions

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

textbfControl Flow Instructions

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

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)
```

2.1.4 Basic ARM Instructions

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 x Rm (lower 32 bits)
```

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)
```

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 x 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
```

2.1.5 Memory Access Instructions

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)
```

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
```

Addressing Modes

textbfOffset Addressing

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

textbfPre-indexed Addressing

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

textbfPost-indexed Addressing

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

textbfRegister Offset

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

2.1.6 Assembly Program Structure

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
```

Labels

Purpose

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

textbfSyntax

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

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

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!"
```

2.1.7 ARM Development Tools

Toolchain Components

Cross-Compiler

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

textbfAssembler

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

textbfLinker

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

textbfEmulator

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

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
\
```

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
\
```

2.1.8 Programming in ARM Assembly

Translating C to ARM

C Code:

```
int a = 5;
int b = 10;
int c = a + b;
\\end{lstlisting}

\textbf{ARM Assembly:}

\begin{lstlisting}[language=assembly]
    MOV R0, #5        ; a = 5
    MOV R1, #10       ; b = 10
    ADD R2, R0, R1    ; c = a + b
\\end{lstlisting}

\textbf{C Code with Array:}

\begin{lstlisting}[language=c]
int arr[3] = {1, 2, 3};
int x = arr[1];
\\end{lstlisting}

\textbf{ARM Assembly:}

\begin{lstlisting}[language=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)
\\end{lstlisting}

\subsubsection{Common Patterns}

\textbf{Clearing a Register}

\begin{lstlisting}[language=assembly]
```

```
MOV R0, #0            ; Method 1
EOR R0, R0, R0        ; Method 2 (XOR with itself)
\\end{lstlisting}

\textbf{Negating a Value}

\begin{lstlisting}language=Assembler
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 x 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
```

2.1.9 Key Takeaways

- **ARM follows RISC principles** - simple instructions, load/store architecture, large register file, fixed instruction length
- **16 registers (R0-R15)** with special purposes: R13 (SP), R14 (LR), R15 (PC), and calling conventions for R0-R11
- **16 general-purpose registers** (R0-R15) with special roles for SP, LR, and PC
- **Three main instruction categories** - data processing (arithmetic/logic), data transfer (load/store), control flow (branches)
- **Fixed 32-bit instruction format** simplifies hardware and enables efficient pipelining
- **Little-endian byte ordering** - least significant byte stored at lowest address
- **Immediate values** indicated by # symbol, with encoding constraints due to fixed instruction size
- **Memory access only through LOAD/STORE** - arithmetic operations work on registers only (load/store architecture)
- **Rich addressing modes** - offset, pre-indexed, post-indexed, register offset with optional shifts
- **Cross-compilation toolchain** - arm-linux-gnueabi-gcc, as, ld, and qemu-arm for development on x86

- **Assembly programming requires understanding** of register allocation, instruction selection, and calling conventions

2.1.10 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.

2.2 Lecture 5: Number Representation and Instruction Encoding

By Dr. Kisaru Liyanage

2.2.1 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.

2.2.2 Number Representation Systems

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 * 2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0)
      = 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
```

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

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)
```

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
```

2.2.3 ARM Instruction Encoding

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)

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
```

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
```

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 $(2 \times rotationfield)$ 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
```

2.2.4 Logical Operations

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
```

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
```

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
```

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)
```

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
```

2.2.5 Practical Bit Manipulation Examples

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)
```

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
```

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
```

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
```

2.2.6 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.

2.2.7 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.

2.3 Lecture 6: Branching and Control Flow

By Dr. Kisaru Liyanage

2.3.1 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.

2.3.2 Fundamentals of Conditional Execution

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

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)

2.3.3 Comparison Instructions

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
```

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

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)
```

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

2.3.4 Conditional Branch Instructions

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
```

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
```

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
```

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
```

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

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
```

2.3.5 Labels in Assembly

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
```

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

2.3.6 Implementing Control Structures

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)
```

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...
```

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...
```

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...
```

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

2.3.7 Array Access in Loops

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 $\times$ 4 (shift left 2 = multiply by 4)
- Words are 4 bytes, so array element i is at base + (i $\times$ 4)
- Efficient: shift is faster than multiplication

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
```

2.3.8 PC-Relative Addressing

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

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
```

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

2.3.9 Conditional Execution (Alternative to Branching)

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
```

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)
```

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

2.3.10 Basic Blocks

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
```

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

2.3.11 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.

2.3.12 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.

2.4 Lecture 7: Function Call and Return

By Dr. Kisaru Liyanage

2.4.1 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.

2.4.2 Function Calling Fundamentals

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

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
}
```

2.4.3 ARM Register Conventions

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
```

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
```

2.4.4 Function Call Instructions

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

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
```

2.4.5 Parameter Passing

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
```

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
```

2.4.6 Return Values

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
```

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
```

2.4.7 The Stack

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
```

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

2.4.8 Stack Operations

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

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
```

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
```

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

2.4.9 Register Preservation

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

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

2.4.10 Nested Function Calls (Non-Leaf Functions)

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
```

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
```

2.4.11 Recursion Example: Factorial

Factorial Function

C Code

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

Key Points

- Base case: $n \leq 1$, return 1
- Recursive case: return $n\times$ fact$(n-1)$
- Each call creates new stack frame
- Stack unwinds as recursion returns

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
```

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!

2.4.12 Memory Layout and Stack vs. Heap

Complete Memory Layout

```
High Address (0xFFFFFFFF)
  +---------------+
  |   Reserved    | OS and system
  +---------------+
  |    Stack      | <- SP (grows down)
  |      v        |  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)
```

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

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

2.4.13 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.

2.4.14 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.

2.5 Lecture 8: Memory Access and String Operations

By Dr. Kisaru Liyanage

2.5.1 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.

2.5.2 Character Data and Encoding

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
```

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)

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

2.5.3 Byte Load/Store Operations

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

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
```

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

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

2.5.4 Half-Word Load/Store Operations

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

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
```

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)
```

2.5.5 String Copy Example (strcpy)

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

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
```

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

2.5.6 Library Functions: scanf and printf

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
```

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
```

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

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)

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

2.5.7 Compilation, Linking, and Loading

Translation Overview

Complete Process

```
C Program (.c)
    | [Compiler]
    v
```

```
Assembly (.s)
    | [Assembler]
    v
Object Module (.o)
    | [Linker]
    v
Executable (a.out)
    | [Loader]
    v
Memory (running program)
```

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
```

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)

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

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)

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()

2.5.8 Exercises

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;
    }
}
```

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
```

Skills Required

- Character data handling (LDRB/STRB)
- String manipulation
- scanf for input
- printf for output
- Stack management
- Function calling conventions
- Loop implementation
- Array indexing

2.5.9 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.

2.5.10 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 3

Processor Design

3.1 Lecture 9: Microarchitecture and Datapath Design

By Dr. Isuru Nawinne

3.1.1 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.

3.1.2 Course Context and MIPS ISA

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

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

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)

3.1.3 Digital Logic Review

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

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: $Y = \text{function}(A, B, operation)$

Characteristics

- Output changes immediately with input (plus propagation delay)
- Can draw complete truth table
- Asynchronous operation (no clock needed)

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

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 x 10^-12) = 4 GHz
```

Register Operations

Basic Register

- Stores multi-bit value (e.g., 32 bits)
- Updates on clock edge: D (input) → Q (output state)

Timing Example

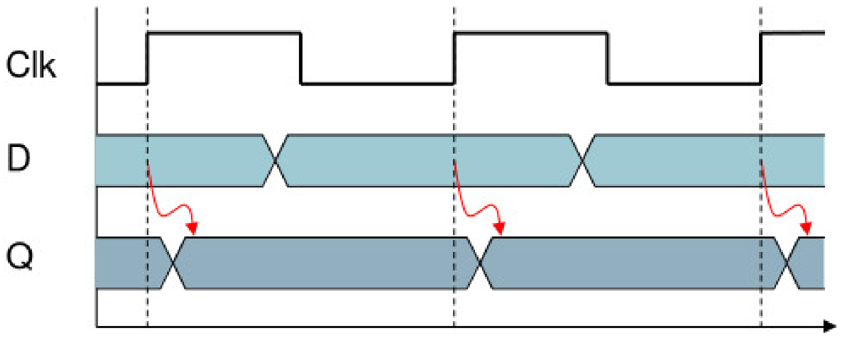


Figure 3.1: Register Timing Diagram

Register with Write Control

- Additional Write Enable signal
- Updates ONLY when clock edge AND Write Enable = 1
- Otherwise holds previous value

Timing Example

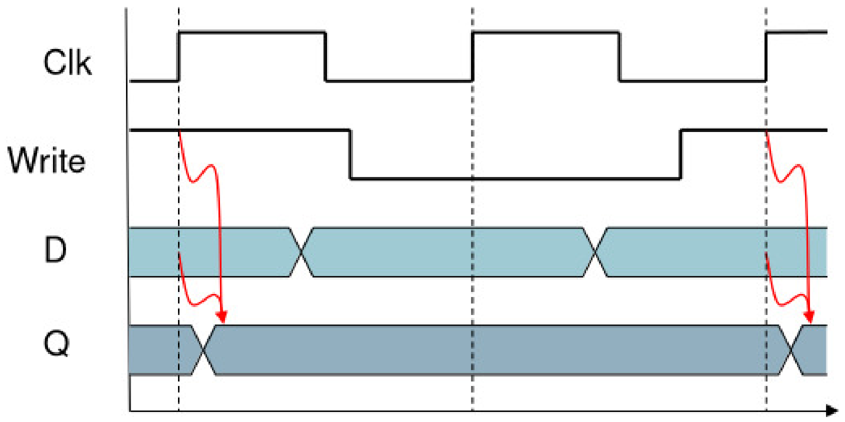


Figure 3.2: Register with Write Enable Timing Diagram

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!)

3.1.4 CPU Execution Stages

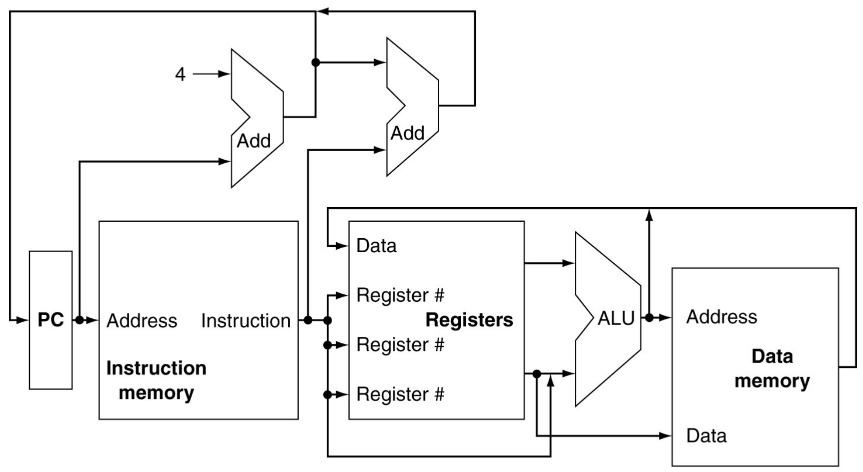


Figure 3.3: CPU Execution Stages Overview

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

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

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

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

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

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

3.1.5 R-Type Instruction Datapath

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)

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

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)

3.1.6 I-Type Instruction Datapath

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

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)

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)
```

3.1.7 Load/Store Instruction Datapath

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
```

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

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

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

3.1.8 Branch Instruction Datapath

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

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!)

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

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!

3.1.9 Complete Single-Cycle Datapath

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

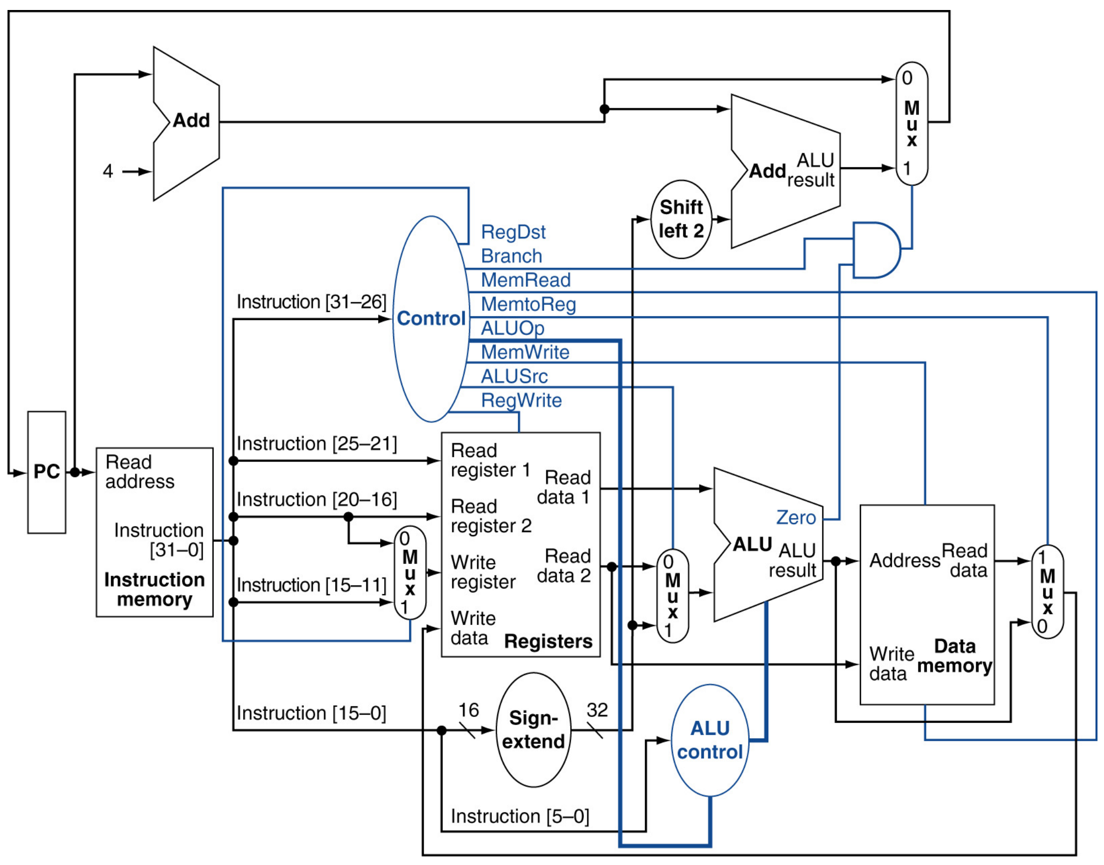


Figure 3.4: Complete Single-Cycle CPU Control and Datapath

- 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)

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

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)

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

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

3.1.10 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.

3.1.11 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.

3.2 Lecture 10: Processor Control

By Dr. Isuru Nawinne

3.2.1 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.

3.2.2 Control Unit Overview

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)

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

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

3.2.3 ALU Operations for Different Instructions

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)
```

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
```

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
```

3.2.4 ALU Control Signal

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

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

3.2.5 Two-Stage ALU Control Generation

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

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

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

Complete ALU Control Path

Flow Diagram:

```
Instruction Opcode (6 bits)
         |
         v
   [Main Control Unit]
         |
         v
    ALUOp (2 bits)  +  Funct Field (6 bits)
         |                    |
         v                    v
              [ALU Control Unit]
                     |
                     v
            ALU Control (4 bits)
                     |
                     v
                   [ALU]
```

Advantages:

- Modular design
- Simplified main control
- Localized R-type complexity
- Easier to verify

3.2.6 Main Control Signals

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

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
```

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)
```

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

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
```

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

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
```

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)
```

3.2.7 Control Signal Truth Table

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	

Table 3.1: Control Signal Truth Table

  Legend:

- **0**: Signal low/false/select input 0
- **1**: Signal high/true/select input 1
- **X**: Don't Care (not used, can be anything)

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)

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

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

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
```

3.2.8 Control Unit Implementation

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

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)
```

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)

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

3.2.9 Why Separate MemRead and MemWrite?

Initial Observation

Question: Seem mutually exclusive—why not one signal?

- Could use: 0 = Read, 1 = Write
- Appears redundant

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)
```

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

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

3.2.10 Complete Datapath with Control

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)

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
```

3.2.11 Key Takeaways

1. **Control unit generates signals based on instruction opcode**, orchestrating datapath operations.
2. **ALU control uses two-stage generation**: Opcode $\rightarrow$ ALUOp (2 bits) $\rightarrow$ 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.

3.2.12 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 $\rightarrow$ ALUOp $\rightarrow$ 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.

3.3 Lecture 11: Complete Single-Cycle MIPS Processor and Performance Analysis

By Dr. Isuru Nawinne

3.3.1 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.

3.3.2 Lecture Overview and Context

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

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

3.3.3 Control Unit Inputs and Outputs

Control Unit Inputs

Total Input Bits: 12 bits

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

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

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:

1. **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

3.3.4 R-Type Instruction Detailed Analysis

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

  #### 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

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

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 = $2value$
- Read Data 2 = $3value$

Step 4: ALU Operation

- ALUSrc = 0: Select RT value for Input B
- Input A = $2value, InputB =$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

3.3.5 Branch If Equal Instruction Detailed Analysis

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` $\times$ `4)`

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 $\times$ 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

Control Signal Values for BEQ

Exercise Example: BEQ $1, $2, 100
Detailed Explanations:

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

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

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 $\times$ 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 $\leftarrow$ Branch Target (1404) If PCSrc = 0: PC $\leftarrow$ PC + 4 (1004)

3.3.6 Load Word Instruction Detailed Analysis

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]

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

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

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!

7. Determines minimum clock period
8. All other instructions must wait for this worst case
9. Major performance bottleneck

3.3.7 Store Word Instruction Detailed Analysis

Instruction Format

Encoding Structure (32 bits):

10. **Bits 26-31**: Opcode = 101011 (43) - SW
11. **Bits 21-25**: RS (5 bits) - Base address register
12. **Bits 16-20**: RT (5 bits) - Source data register
13. **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")

Datapath Elements Used

Active Elements:

14. Instruction Memory
15. Program Counter PC+4 Adder
16. Register File: Read RS (base) AND RT (data source)
17. Sign Extender
18. Multiplexer (ALUSrc): Select immediate
19. ALU: Add base + offset
20. Data Memory: Write RT data at calculated address
    Inactive Elements:
21. Register Write: No register write
22. Memory Read: Writing, not reading
23. MemtoReg multiplexer: Output not used
    Key Difference from Load:
24. TWO register reads: RS for base, RT for data
25. Memory write instead of read
26. NO register write operation

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):

27. RegWrite = 0: No register write
28. Write address irrelevant
29. Could be 0 or 1, doesn't matter
30. Using X simplifies logic design

CRITICAL: RegWrite = 0:

31. Must prevent register file write
32. **If RegWrite = 1:** Disaster!
33. Some register address fed to write port
34. Either ALU result (address) or memory data (garbage, MemRead=0)
35. Would corrupt random register
36. Data integrity violated

Why It Matters:

37. Hardware operates in parallel
38. Multiplexers produce outputs even if not used
39. Without RegWrite = 0:
40. RegDst mux outputs some address
41. MemtoReg mux outputs some data
42. If RegWrite = 1: This garbage written to register!
43. Control signal correctness essential

MemRead = 0, MemWrite = 1:

44. Writing to memory, not reading
45. MemRead = 0: Memory read output undefined
46. MemWrite = 1: Memory accepts write data

47. Opposite of Load Word
MemtoReg = X (Don't Care):
48. RegWrite = 0: Write data source irrelevant
49. Output not used
50. Even if wrong data selected, RegWrite prevents write
ALUOp = 00:
51. Same as Load Word
52. Address calculation: ADD operation
ALUSrc = 1:
53. Need immediate offset
54. Same as Load Word

Important Lesson: Don't Care vs Zero

Student Confusion: "$RegDst = 0 isnotwrong, butbestansweris X$"
Clarification:

55. **Functionally:** 0 works (doesn't cause error)
56. **Logically:** X is correct (truly doesn't matter)
57. **Design perspective:** X simplifies Boolean expressions
58. **Karnaugh map minimization:** X allows more groupings
However:
59. RegWrite MUST be 0 (not X!)
60. MemWrite MUST be correct (not X!)
61. Read/Write enables are critical for data integrity

3.3.8 Jump Instruction Integration

Instruction Format

Encoding Structure (32 bits):

62. **Bits 26-31**: Opcode = 000010 (2) - J
63. **Bits 0-25**: Address (26 bits) - Jump target (word address)
Alternative: JAL (Jump and Link)
64. Opcode = 000011 (3)
65. Used for function calls
66. 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`

Jump Target Address Calculation

Word Address to Byte Address:

67. Target field: 26 bits (word address)
68. Shift left by 2: Append 2 zero bits
69. Result: 28-bit byte address

Upper 4 Bits:

70. Take from PC+4 current value
71. Bits 31:28 of next sequential instruction
72. Preserves region (256 MB regions)
73. 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:

74. PC = 0x10000000
75. PC+4 = 0x10000004
76. Target = 100 = 0x000064
77. Shift left 2: 0x000190
78. Upper 4 bits: 0x1
79. Jump Address: 0x10000190

Limitation:

80. Can only jump within 28-bit range (256 MB)
81. Upper 4 bits fixed by current PC region
82. For larger jumps: Use jump register (JR) instruction

Additional Datapath Hardware

New Components:

Shift Left 2 (for jump):

83. Input: 26-bit target field
84. Output: 28-bit byte offset
85. Implementation: Wire routing (no actual shifter!)

Concatenation Logic:

86. Input 1: PC+4 bits [31:28] (4 bits)

87. Input 2: Shifted target (28 bits)
88. Output: 32-bit jump address
89. Implementation: Wire concatenation
    New Multiplexer:
90. **Input 0**: Output from branch/sequential mux
91. Could be PC+4 or branch target
92. **Input 1**: Jump target address (32 bits)
93. **Select**: Jump control signal
94. **Output**: Next PC value
    Original PC Source Mux:
95. Input 0: PC + 4
96. Input 1: Branch target
97. Select: PCSrc (Branch AND Zero)
    New Jump Mux (outer):
98. Input 0: Original mux output (PC+4 or branch target)
99. Input 1: Jump target
100. Select: Jump signal
101. Output: Final next PC value

Jump Control Signal

Jump Signal:

102. 10th control output bit
103. Generated by Control Unit
104. Based on opcode = 2 (J) or 3 (JAL)
     Values:
105. Jump = 1: Select jump target
106. Jump = 0: Select sequential/branch
     Other Control Signals for Jump:
     Note: JAL (Jump and Link) different:
107. RegWrite = 1 (saves return address)
108. RegDst = ? (special: write to 31)
108. Additional logic needed for return address

Complete Datapath with Jump

All Instruction Types Supported:

109. **R-type**: Arithmetic, logic, shift

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

110. **I-type**: Load, Store, Branch, Immediate arithmetic
111. **J-type**: Jump, Jump and Link
Coverage:
112. 95
113. Complete single-cycle implementation
114. Additional variants (BNE, shifts, etc.) need minor additions
Datapath Completeness:
115. Two memories: Instruction and Data
116. One ALU for computation
117. Multiple adders: PC+4, Branch target
118. Many multiplexers for data routing
119. Sign extender
120. Shift left 2 circuits (wire routing)
121. Control unit with 10 control signal bits

3.3.9 Timing Analysis with Concrete Delays

Assumed Component Delays

Delay Values (in nanoseconds):
Assumptions:
122. Simplified for analysis
123. Real delays depend on technology, circuit design
124. Memory accesses typically slowest
125. Combinational logic relatively fast

Critical Path Analysis

Definition:
126. Longest delay path from clock edge to clock edge

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	$\sim$0 ns	Negligible (wire replication)
Multiplexers	$\sim$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

127. Determines minimum clock period
128. All combinational logic between sequential elements

Single-Cycle Constraint:

129. Entire instruction must complete in one clock cycle
130. Clock period $\geq$ Critical path delay
131. All instructions take same time (worst case)

Load Word Instruction Timing

Step-by-Step Delay Calculation:

Step 1: Instruction Fetch (2 ns)

132. Clock edge: PC updated
133. PC $\rightarrow$ Instruction Memory
134. Instruction Memory reads and outputs instruction
135. Delay: 2 ns
136. Running total: 2 ns

Step 2: Register Read (1 ns)

137. Instruction decoded
138. RS field extracted
139. RS $\rightarrow$ Register File Read Address 1
140. Register File outputs base address
141. Delay: 1 ns
142. Running total: $2 + 1 = 3$ ns

Step 3: Sign Extension ($\sim$0 ns)

143. Immediate field extracted
144. Sign extended to 32 bits
145. Delay: Negligible

146. Running total: 3 ns

Step 4: ALU Address Calculation (2 ns)

147. Base address + offset
148. ALU performs addition
149. Output: Memory address
150. Delay: 2 ns
151. Running total: $3 + 2 = 5$ ns

Step 5: Memory Read (2 ns)

152. ALU result $\rightarrow$ Data Memory address
153. MemRead = 1 asserted
154. Data Memory reads and outputs data
155. Delay: 2 ns
156. Running total: $5 + 2 = 7$ ns

Step 6: Register Write Setup (~0 ns)

157. Memory data $\rightarrow$ Register Write Data input
158. RT $\rightarrow$ Register Write Address
159. Ready for clock edge
160. Delay: Setup time negligible
161. Running total: 7 ns

Clock Edge: Register Write (next cycle)

162. At next positive clock edge
163. Data written to register
164. 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!

165. Longest instruction in single-cycle design
166. Determines clock period for ALL instructions

Store Word Instruction Timing

Step-by-Step Delay:

(a) **Instruction Fetch:** 2 ns (total: 2 ns)
(b) **Register Read:** 1 ns (total: 3 ns)
(c) Read RS (base) AND RT (data)
(d) **Sign Extension:** 0 ns (total: 3 ns)
(e) **ALU Address Calculation:** 2 ns (total: 5 ns)

(f) **Memory Write Setup:** 0 ns (total: 5 ns)

(g) Address and data ready at memory inputs
Clock Edge: Memory Write (end of cycle)

(h) Data written to memory at clock edge

(i) Takes 2 ns but next instruction fetch also 2 ns

(j) Next instruction register read starts after 3 ns total

(k) Memory write completes before register read needs data

(l) No conflict
Minimum Time Required: 5 nanoseconds
Note:

(m) Faster than Load Word (no memory read delay)

(n) But must use 7 ns clock period anyway (single-cycle)

(o) Wastes 2 ns per Store instruction

Arithmetic Instruction Timing (ADD, SUB, AND, OR)

Step-by-Step Delay:

i. **Instruction Fetch:** 2 ns (total: 2 ns)

ii. **Register Read:** 1 ns (total: 3 ns)

iii. Read RS and RT

iv. **ALU Operation:** 2 ns (total: 5 ns)

v. Perform arithmetic/logic operation

vi. **Register Write Setup:** 0 ns (total: 5 ns)

vii. ALU result ready at register write data input
Clock Edge: Register Write

viii. Result written to RD
Minimum Time Required: 5 nanoseconds
Efficiency Loss:

ix. Could run at 5 ns clock period

x. Forced to wait 7 ns (Load Word limitation)

xi. Wastes 2 ns = 28.6

Branch Instruction Timing

Step-by-Step Delay:

A. **Instruction Fetch:** 2 ns (total: 2 ns)

B. **Register Read:** 1 ns (total: 3 ns)

C. Read RS and RT for comparison

D. **ALU Comparison:** 2 ns (total: 5 ns)

E. Subtract RS - RT

F. Generate Zero flag

G. **Branch Target Calculation:** 2 ns (parallel with ALU)

H. Sign extend offset: 0 ns

I. Shift left 2: 0 ns (wire routing)

J. Add to PC+4: 2 ns

K. Can happen in parallel with ALU operation!

L. **PC Update Setup:** 0 ns (total: 5 ns)

M. Zero flag + Branch → PCSrc

N. Multiplexer selects next PC

O. Ready for clock edge
Minimum Time Required: 5 nanoseconds
Key Insight:

P. Branch target calculation parallel to ALU

Q. PC+4 already available from fetch stage

R. No memory access needed

S. Fast like R-type

Jump Instruction Timing

Step-by-Step Delay:

T. **Instruction Fetch:** 2 ns (total: 2 ns)

U. Also calculates PC+4 in parallel

V. **Jump Target Calculation:** 0 ns

W. Extract 26-bit target

X. Shift left 2: Wire routing, 0 ns

Y. Concatenate with PC+4[31:28]: Wire connection, 0 ns

Z. No ALU, no memory, no registers!

. **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!

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

3.3.10 Performance Analysis

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

Average Time Calculation

Variable Time (Ideal):
Average = $(0.48 \times 5) + (0.22 \times 7) + (0.11 \times 5) + (0.19 \times 5) = 2.40 + 1.54 + 0.55 + 0.95 = 5.44$ 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

Critical Path Problem

Critical Path Determination:

- Load Word uses most datapath elements
- Sequential dependencies:

- Instruction Memory
- Register File
- ALU
- Data Memory
- (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

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%

3.3.11 Path to Better Performance: Multi-Cycle Design

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

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

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)

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 = $(0.48 \times 8) + (0.22 \times 10) + (0.11 \times 8) + (0.19 \times 6) = 3.84 + 2.20 + 0.88 + 1.14 = 8.06$ 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 = $(0.48 \times 5.6) + (0.22 \times 7.0) + (0.11 \times 5.6) + (0.19 \times 4.2)$
$= 2.69 + 1.54 + 0.62 + 0.80 = 5.65$ ns per instruction
Speedup = $7 / 5.65 = 1.24\times$ faster

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)

3.3.12 Preview: Pipelining

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

Coming Next

Topics:

- Pipelined datapath design
- Hazard detection and resolution
- Forwarding (bypassing)
- Branch prediction
- Performance analysis
- MIPS pipeline implementation

3.3.13 Key Takeaways

- **Single-cycle design executes each instruction in one clock cycle**, with clock period determined by the slowest instruction (Load Word at 7 ns).
- **Control unit generates signals based on opcode**, orchestrating datapath operations for R-type, Load, Store, Branch, and Jump instructions.
- **Load Word is the critical path** (Instruction Fetch → Register Read → ALU → Memory Read → Register Write), determining minimum clock period.
- **Jump instruction uses PC[31:28]** concatenated with shifted immediate to form 32-bit target address, enabling 256 MB jump range.

. **Control signals must prevent data corruption**, with RegWrite=0 for Store and Branch to avoid unintended register modifications.

. **"Don't care" values (X) simplify control logic**, allowing optimization when signals don't affect instruction outcome.

. **Hardware operates concurrently**, not sequentially—multiple operations happen simultaneously within each clock cycle.

. **Performance inefficiency drives design evolution**, as most instructions finish early but must wait for full clock period.

. **Resource utilization varies dramatically**, with arithmetic instructions using 43
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.

3.3.14 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—multicycle 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.