Last Lecture: Timing and Verification

- **Timing in combinational circuits**
  - Propagation delay and contamination delay
  - Glitches

- **Timing in sequential circuits**
  - Setup time and hold time
  - Determining how fast a circuit can operate

- **Circuit Verification & Testing**
  - How to know a circuit works correctly
  - Functional verification & testing
  - Timing verification & testing
  - Offline vs. online testing
Timing in a Single Sequential Component

- Clock cycle time is determined by the maximum logic delay across different possible combinational paths.
Timing in Multiple Sequential Components

- Clock cycle time is determined by the maximum logic delay across different sequential components
A Final Word on Timing

Meetings Timing Constraints can be done via Principled Design

Clock cycle time is determined by the maximum logic delay we can accommodate without violating timing constraints

Good design principles

- **Critical path design**: Minimize the maximum logic delay
  → Maximizes performance

- **Balanced design**: Balance maximum logic delays across different parts of a system (i.e., between different pairs of flip flops)
  → No bottlenecks + minimizes wasted time

- **Bread and butter design**: Optimize for the common case, but make sure non-common-cases do not overwhelm the design
  → Maximizes performance for common use cases
Making Sure A Design Works Correctly

- This is done via Verification and Testing methods

- Functional Verification & Testing
  - Make sure the circuit logically operates correctly

- Timing Verification & Testing
  - Make sure the circuit operates correctly when timing is considered

- Verification and Testing consume most of manufacture time
  - Performed at many stages of design, e.g., pre-silicon, post-silicon
  - It is very difficult to completely verify and test complex circuits
    - Recall billions of transistors on a chip
  - Even after so much V&T, errors still slip into the field
    - Online verification and testing is critical in modern systems
We have a new problem: cores that disobey instructions

CPU cores that
- repeatedly
- but not always
- mis-calculate
- certain computations
- without giving any obvious signal

“Mercurial cores” committing
"Corrupt Execution Errors"

Due to local silicon defects, not eg cosmic rays

Google

https://www.youtube.com/watch?v=QMF3rqhjYuM
Silent Data Corruptions at Scale

Harish Dattatraya Dixit
Facebook, Inc.
hdd@fb.com

Sneha Pendharkar
Facebook, Inc.
spendharkar@fb.com

Matt Beadon
Facebook, Inc.
mbeadon@fb.com

Chris Mason
Facebook, Inc.
clm@fb.com

Tejasvi Chakravarthy
Facebook, Inc.
teju@fb.com

Bharath Muthiah
Facebook, Inc.
bharathm@fb.com

Sriram Sankar
Facebook Inc.
sriramsankar@fb.com

Cores that don’t count

Peter H. Hochschild
Paul Turner
Jeffrey C. Mogul
Google
Sunnyvale, CA, US

Rama Govindaraju
Parthasarathy
Ranganathan
Google
Sunnyvale, CA, US

David E. Culler
Amin Vahdat
Google
Sunnyvale, CA, US

https://www.youtube.com/watch?v=QMF3rqhjYuM
1 Introduction

Imagine you are running a massive-scale data-analysis pipeline in production, and one day it starts to give you wrong answers – somewhere in the pipeline, a class of computations are yielding corrupt results. Investigation fingers a surprising cause: an innocuous change to a low-level library. The change itself was correct, but it caused servers to make heavier use of otherwise rarely-used instructions. Moreover, only a small subset of the server machines are repeatedly responsible for the errors.

This happened to us at Google. Deeper investigation revealed that these instructions malfunctioned due to manufacturing defects, in a way that could only be detected by checking the results of these instructions against the expected results; these are “silent” corrupt execution errors, or CEEs. Wider investigation found multiple different kinds of CEEs; that the detected incidence is much higher than software engineers expect; that they are not just incremental increases in the background rate of hardware errors; that these can manifest long after initial installation; and that they typically afflict specific cores on multi-core CPUs, rather than the entire chip. We refer to these cores as “mercurial.”

ABSTRACT

Silent Data Corruption (SDC) can have negative impact on large-scale infrastructure services. SDCs are not captured by error reporting mechanisms within a Central Processing Unit (CPU) and hence are not traceable at the hardware level. However, the data corruptions propagate across the stack and manifest as application-level problems. These types of errors can result in data loss and can require months of debug engineering time.

In this paper, we describe common defect types observed in silicon manufacturing that leads to SDCs. We discuss a real-world example of silent data corruption within a datacenter application. We provide the debug flow followed to root-cause and triage faulty instructions within a CPU using a case study, as an illustration on how to debug this class of errors. We provide a high-level overview of the mitigations to reduce the risk of silent data corruptions within a large production fleet.

In our large-scale infrastructure, we have run a vast library of silent error test scenarios across hundreds of thousands of machines in our fleet. This has resulted in hundreds of CPUs detected for these errors, showing that SDCs are a systemic issue across generations. We have monitored SDCs for a period longer than 18 months. Based on this experience, we determine that reducing silent data corruptions requires not only hardware resiliency and production detection mechanisms, but also robust fault-tolerant software architectures.
Recall: RowHammer

- One can predictably induce bit flips in commodity DRAM chips
  - All tested DRAM chips are vulnerable

- First example of how a simple hardware failure mechanism can create a widespread system security vulnerability
Recall: One Can Take Over an Otherwise-Secure System

Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors

Abstract. Memory isolation is a key property of a reliable and secure computing system — an access to one memory address should not have unintended side effects on data stored in other addresses. However, as DRAM process technology

Project Zero

Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors (Kim et al., ISCA 2014)

News and updates from the Project Zero team at Google

Exploiting the DRAM rowhammer bug to gain kernel privileges (Seaborn+, 2015)

Monday, March 9, 2015

Exploiting the DRAM rowhammer bug to gain kernel privileges

It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after.
Challenge and Opportunity

- How do we build fundamentally reliable, safe, secure systems?

- Verification and Testing, both offline and online, are critical

- You will get a small glimpse of V&T in your labs

- To prepare, please watch Lecture 6c (Verification & Testing)
  - Overview of verification and testing approaches & complexity
  - Examples of how to do testing in Verilog
Lecture 6c: Verification & Testing

Automatic Testbench

- The DUT output is compared against the golden model

Challenge:
- Need to generate inputs to the designs
  - Sequential values to cover the entire input space?
  - Random values?

Digital Design & Comp Arch - Lecture 6c: Verification & Testing (Spring 2023)

https://www.youtube.com/watch?v=qGO5w9KZiHQ
The von Neumann Model & Instruction Set Architectures
Extra Credit Assignment 1: Talk Analysis

- **Intelligent Architectures for Intelligent Machines**

- **Watch and analyze this short lecture (33 minutes)**
  - [https://www.youtube.com/watch?v=WxHribseelw](https://www.youtube.com/watch?v=WxHribseelw) (Oct 2022)

- **Assignment – for 1% extra credit**
  - **Write a good 1-page summary (following our guidelines)**
    - What are your key takeaways?
    - What did you learn?
    - What did you like or dislike?
  - Submit your summary to Moodle – deadline April 1
Extra Credit Assignment 2: Moore’s Law

- **Paper review**

- **Optional Assignment – for 1% extra credit**
  - Write a 1-page review
  - Upload PDF file to Moodle – Deadline: April 1

- I strongly recommend that you follow my guidelines for (paper) review (see next slide)
Guidelines on how to review papers critically

- Guideline slides: pdf, ppt
- Video: https://www.youtube.com/watch?v=tOL6FANAJ8c

Example reviews on “Main Memory Scaling: Challenges and Solution Directions” (link to the paper)
- Review 1
- Review 2

Example review on “Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems” (link to the paper)
- Review 1
## What Have We Learned So Far?

- We are mostly done with “Digital Design” part of this course

### Spring 2023 Lectures/Schedule

<table>
<thead>
<tr>
<th>Week</th>
<th>Date</th>
<th>Livestream</th>
<th>Lecture</th>
<th>Readings</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>23.02 Thu.</td>
<td>YouTube Live</td>
<td>L1: Introduction and Basics (PDF) (PPT)</td>
<td>Suggested Mentioned</td>
</tr>
<tr>
<td></td>
<td>24.02 Fri.</td>
<td>YouTube Live</td>
<td>L2a: Tradeoffs, Metrics, Mindset (PDF) (PPT)</td>
<td>Suggested Mentioned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>L2b: Combinational Logic I (PDF) (PPT)</td>
<td>Suggested Mentioned</td>
</tr>
<tr>
<td>W2</td>
<td>02.03 Thu.</td>
<td>YouTube Live</td>
<td>L3: Combinational Logic II (PDF) (PPT)</td>
<td>Suggested Mentioned</td>
</tr>
<tr>
<td></td>
<td>03.03 Fri.</td>
<td>YouTube Live</td>
<td>L4: Sequential Logic Design I (PDF) (PPT)</td>
<td>Suggested Mentioned</td>
</tr>
<tr>
<td>W3</td>
<td>09.03 Thu.</td>
<td>YouTube Live</td>
<td>L5a: Sequential Logic Design II (PDF) (PPT)</td>
<td>Suggested Mentioned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>L5b: Hardware Description Languages and Verilog (PDF) (PPT)</td>
<td>Suggested Mentioned</td>
</tr>
<tr>
<td></td>
<td>10.03 Fri.</td>
<td>YouTube Live</td>
<td>L6a: Hardware Description Languages and Verilog II (PDF) (PPT)</td>
<td>Suggested Mentioned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>L6b: Timing and Verification (PDF) (PPT)</td>
<td>Required Suggested</td>
</tr>
</tbody>
</table>

Agenda for Today & Next Few Lectures

- The von Neumann model
- LC-3: An example of von Neumann machine
- LC-3 and MIPS Instruction Set Architectures
- LC-3 and MIPS assembly and programming
- Introduction to microarchitecture and single-cycle microarchitecture
- Multi-cycle microarchitecture
What Will We Learn Today?

- Basic elements of a computer & the von Neumann model
  - LC-3: An example von Neumann machine

- Instruction Set Architectures: LC-3 and MIPS
  - Operate instructions
  - Data movement instructions
  - Control instructions

- Instruction formats

- Addressing modes
Readings

This week

- Von Neumann Model, ISA, LC-3, and MIPS
  - P&P, Chapters 4, 5 (we will follow these today & tomorrow)
  - H&H, Chapter 6 (until 6.5)
  - P&P, Appendices A and C (ISA and microarchitecture of LC-3)
  - H&H, Appendix B (MIPS instructions)

- Programming
  - P&P, Chapter 6 (we will follow this tomorrow)

  Recommended: H&H Chapter 5, especially 5.1, 5.2, 5.4, 5.5

Next week

- Introduction to microarchitecture and single-cycle microarchitecture
  - H&H, Chapter 7.1-7.3
  - P&P, Appendices A and C

- Multi-cycle microarchitecture
  - H&H, Chapter 7.4
  - P&P, Appendices A and C
Building a Computing System
The von Neumann Model
Recall: What is A Computer?

- We will cover all three components

Processing:
- control (sequencing)
- datapath

Memory:
- program and data

I/O
In past lectures, we learned how to design
- Combinational logic structures
- Sequential logic structures

With logic structures, we can build
- Execution units
- Decision units
- Memory/storage units
- Communication units

All are basic elements of a computer
- We will raise our abstraction level today
- Use logic structures to construct a basic computer model
Basic Components of a Computer

- To get a task done by a (general-purpose) computer, we need:
  - A computer program
    - That specifies what the computer must do
  - The computer itself
    - To carry out the specified task

- Program: A set of instructions
  - Each instruction specifies a well-defined piece of work for the computer to carry out
  - Instruction: the smallest piece of specified work in a program

- Instruction set: All possible instructions that a computer is designed to be able to carry out
The von Neumann Model

- In order to build a computer, we need an execution model for processing computer programs

- John von Neumann proposed a fundamental model in 1946

- The von Neumann Model consists of 5 components
  - Memory (stores the program and data)
  - Processing unit
  - Input
  - Output
  - Control unit (controls the order in which instructions are carried out)

- Throughout this lecture, we will examine two examples of the von Neumann model
  - LC-3
  - MIPS

Burks, Goldstein, von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument,” 1946.

All general-purpose computers today use the von Neumann model
The von Neumann Model

INPUT
Keyboard, Mouse, Disk...

OUTPUT
Monitor, Printer, Disk...

MEMORY
Mem Addr Reg
Mem Data Reg

PROCESSING UNIT
ALU
TEMP

CONTROL UNIT
PC or IP
Inst Register
The von Neumann Model

**INPUT**
- Keyboard,
- Mouse,
- Disk...

**OUTPUT**
- Monitor,
- Printer,
- Disk...

**CONTROL UNIT**
- PC or IP
- Inst Register

**PROCESSING UNIT**
- ALU
- TEMP

**MEMORY**
- Mem Addr Reg
- Mem Data Reg

**INPUT**

**OUTPUT**
Recall: A Memory Array (4 locations X 3 bits)
Memory

- Memory stores
  - Programs
  - Data

- Memory contains **bits**
  - Bits are logically grouped into **bytes** (8 bits) and **words** (e.g., 8, 16, 32 bits)

- **Address space:** Total number of uniquely identifiable locations in memory
  - In **LC-3**, the address space is $2^{16}$
    - 16-bit addresses
  - In **MIPS**, the address space is $2^{32}$
    - 32-bit addresses
  - In **x86-64**, the address space is (up to) $2^{48}$
    - 48-bit addresses

- **Addressability:** How many bits are stored in each location (address)
  - E.g., 8-bit addressable (or **byte-addressable**)
  - E.g., **word-addressable**
  - A given instruction can operate on a byte or a word
A Simple Example

- A representation of memory with 8 locations
- Each location contains 8 bits (one byte)
  - Byte addressable memory; address space of 8
  - Value 6 is stored in address 4 & value 4 is stored in address 6

<table>
<thead>
<tr>
<th>Address</th>
<th>Data Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>00000110</td>
</tr>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>00000100</td>
</tr>
<tr>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>

Question:
How can we make same-size memory bit addressable?

Answer:
64 locations
Each location stores 1 bit
**Word-Addressable Memory**

- Each **data word** has a **unique address**
  - In MIPS, a unique address for each 32-bit data word
  - In LC-3, a unique address for each 16-bit data word

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th>MIPS memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000003</td>
<td>D 1 6 1 7 A 1 C</td>
<td>Word 3</td>
</tr>
<tr>
<td>000000002</td>
<td>1 3 C 8 1 7 5 5</td>
<td>Word 2</td>
</tr>
<tr>
<td>000000001</td>
<td>F 2 F 1 F 0 F 7</td>
<td>Word 1</td>
</tr>
<tr>
<td>000000000</td>
<td>8 9 A B C D E F</td>
<td>Word 0</td>
</tr>
</tbody>
</table>
### Byte-Addressable Memory

- Each byte has a unique address
  - MIPS is actually byte-addressable
  - LC-3b (updated version of LC-3) is also byte-addressable

<table>
<thead>
<tr>
<th>Byte Address of the Word</th>
<th>Data</th>
<th>MIPS memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000C</td>
<td>D 1</td>
<td>000000000C</td>
</tr>
<tr>
<td></td>
<td>6 1</td>
<td>D 1 1 6 1 7 A 1 C</td>
</tr>
<tr>
<td></td>
<td>7 A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1  C</td>
<td></td>
</tr>
<tr>
<td>000000008</td>
<td>1 3</td>
<td>000000008</td>
</tr>
<tr>
<td></td>
<td>C 8</td>
<td>1 3 8 1 7 5 5</td>
</tr>
<tr>
<td>000000004</td>
<td>F 2</td>
<td>000000004</td>
</tr>
<tr>
<td></td>
<td>F 1</td>
<td>F 2 1 7 5 5</td>
</tr>
<tr>
<td>000000000</td>
<td>F 0</td>
<td>000000000</td>
</tr>
<tr>
<td></td>
<td>F 7</td>
<td>F 0 1 7 5 5</td>
</tr>
</tbody>
</table>

How are these four bytes ordered? Which of the four bytes is most vs. least significant?
Big Endian vs. Little Endian

- Jonathan Swift’s *Gulliver’s Travels*
  - **Big Endians** broke their eggs on the big end of the egg
  - **Little Endians** broke their eggs on the little end of the egg

**BIG ENDIAN** - The way people always broke their eggs in the Lilliput land

**LITTLE ENDIAN** - The way the king then ordered the people to break their eggs
## Big Endian vs. Little Endian

### Big Endian

<table>
<thead>
<tr>
<th>Byte Address</th>
<th>LSBE</th>
<th>MSBE</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

### Little Endian

<table>
<thead>
<tr>
<th>Byte Address</th>
<th>LSBE</th>
<th>MSBE</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>F</td>
<td>E</td>
</tr>
<tr>
<td>8</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

(LSB = Least Significant Byte) (MSB = Most Significant Byte)
**Big Endian vs. Little Endian**

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Does this really matter?</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Answer:</strong> No, it is a convention</td>
<td></td>
</tr>
<tr>
<td><strong>Qualified answer:</strong> No, except when one big-endian system and one little-endian system have to share or exchange data</td>
<td></td>
</tr>
</tbody>
</table>

(Most Significant Byte) (Least Significant Byte)

- LSB in higher byte address
- LSB in lower byte address
Accessing Memory: MAR and MDR

There are two ways of accessing memory
- Reading or loading data from a memory location
- Writing or storing data to a memory location

Two registers are usually used to access memory
- Memory Address Register (MAR)
- Memory Data Register (MDR)

To read
- Step 1: Load the MAR with the address we wish to read from
- Step 2: Data in the corresponding location gets placed in MDR

To write
- Step 1: Load the MAR with the address and the MDR with the data we wish to write
- Step 2: Activate Write Enable signal → value in MDR is written to address specified by MAR
The von Neumann Model

MEMORY
- Mem Addr Reg
- Mem Data Reg

PROCESSING UNIT
- ALU
- TEMP

CONTROL UNIT
- PC or IP
- Inst Register

INPUT
- Keyboard,
- Mouse,
- Disk...

OUTPUT
- Monitor,
- Printer,
- Disk...

INPUT

OUTPUT
Processing Unit

- Performs the actual computation(s)

- The processing unit can consist of many functional units

We start with a simple **Arithmetic and Logic Unit (ALU)**, which executes computation and logic operations
  - **LC-3**: ADD, AND, NOT (XOR in LC-3b)
  - **MIPS**: add, sub, mult, and, nor, sll, slr, slt...

- The ALU processes quantities that are referred to as **words**
  - **Word length** in LC-3 is 16 bits
  - **Word length** in MIPS is 32 bits
Recall: ALU (Arithmetic Logic Unit)

- Combines a variety of arithmetic and logical operations into a single unit (that performs only one function at a time)
- Usually denoted with this symbol:

```
<table>
<thead>
<tr>
<th>$F_{2:0}$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A AND B</td>
</tr>
<tr>
<td>001</td>
<td>A OR B</td>
</tr>
<tr>
<td>010</td>
<td>A + B</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>A AND $\overline{B}$</td>
</tr>
<tr>
<td>101</td>
<td>A OR $\overline{B}$</td>
</tr>
<tr>
<td>110</td>
<td>A – B</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>
```

Figure 5.14 ALU symbol
Recall: Example ALU (Arithmetic Logic Unit)

<table>
<thead>
<tr>
<th>$F_{2:0}$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A AND B</td>
</tr>
<tr>
<td>001</td>
<td>A OR B</td>
</tr>
<tr>
<td>010</td>
<td>A + B</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>A AND $\overline{B}$</td>
</tr>
<tr>
<td>101</td>
<td>A OR $\overline{B}$</td>
</tr>
<tr>
<td>110</td>
<td>A – B</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>

Table 5.1 ALU operations

![ALU Diagram](image)
Processing Unit: Fast Temporary Storage

- It is almost always the case that a computer provides a small amount of storage very close to ALU
  - Purpose: to store temporary values and quickly access them later

- E.g., to calculate \(((A+B)\times C)/D\), the intermediate result of \(A+B\) can be stored in temporary storage
  - Why? It is too slow to store each ALU result in memory & then retrieve it again for future use
    - A memory access is much slower than an addition, multiplication or division
  - Ditto for the intermediate result of \(((A+B)\times C)\)

- This temporary storage is usually a set of registers
  - Called Register File
Registers: Fast Temporary Storage

- **Memory** is large but slow

- ** Registers** in the Processing Unit
  - Ensure fast access to values to be processed in the ALU
  - Typically one register contains *one word (same as word length)*

- **Register Set or Register File**
  - Set of registers that can be manipulated by instructions
  - LC-3 has 8 *general purpose registers (GPRs)*
    - R0 to R7: 3-bit register number
    - Register size = Word length = 16 bits
  - MIPS has 32 *general purpose registers*
    - R0 to R31: 5-bit register number (or Register ID)
    - Register size = Word length = 32 bits
Recall: The Register

How can we use D latches to store more data?
- Use more D latches!
- A single WE signal for all latches for simultaneous writes

Here we have a register, or a structure that stores more than one bit and can be read from and written to

This register holds 4 bits, and its data is referenced as Q[3:0]
Recall: The Register

How can we use D latches to store more data?
• Use more D latches!
• A single WE signal for all latches for simultaneous writes

Here we have a register, or a structure that stores more than one bit and can be read from and written to

This register holds 4 bits, and its data is referenced as $Q[3:0]$
Recall: D Flip-Flop Based Register

- Multiple parallel D flip-flops, each of which storing 1 bit

This register stores 4 bits

This line represents 4 wires
Recall: A 4-Bit D-Flip-Flop-Based Register (Internally)

# MIPS Register File (Conventions)

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>assembler temporary</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>function return value</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>function arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporary variables</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved variables</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>temporary variables</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>26-27</td>
<td>OS temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>function return address</td>
</tr>
</tbody>
</table>
The Von Neumann Model

- **CONTROL UNIT**: PC or IP, Inst Register
- **PROCESSING UNIT**: ALU, TEMP
- **MEMORY**: Mem Addr Reg, Mem Data Reg
- **INPUT**: Keyboard, Mouse, Disk...
- **OUTPUT**: Monitor, Printer, Disk...
Input and Output

- Enable information to get into and out of a computer
- Many devices can be used for input and output

- They are called *peripherals*
  - **Input**
    - Keyboard
    - Mouse
    - Scanner
    - Disks
    - Etc.
  - **Output**
    - Monitor
    - Printer
    - Disks
    - Etc.

- In LC-3, we consider keyboard and monitor
The Von Neumann Model

**INPUT**
- Keyboard
- Mouse
- Disk...

**MEMORY**
- Mem Addr Reg
- Mem Data Reg

**PROCESSING UNIT**
- ALU
- TEMP

**OUTPUT**
- Monitor
- Printer
- Disk...

**CONTROL UNIT**
- PC or IP
- Inst Register
Control Unit

- The control unit is like the conductor of an orchestra

- It conducts the step-by-step process of executing (every instruction in) a program (in sequence)

- It keeps track of which instruction being processed, via
  - Instruction Register (IR), which contains the instruction

- It also keeps track of which instruction to process next, via
  - Program Counter (PC) or Instruction Pointer (IP), another register that contains the address of the (next) instruction to process
Instructions (and programs) specify how to transform the values of programmer visible state.
The von Neumann Model
Von Neumann Model: Two Key Properties

- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:

  - **Stored program**
    - Instructions stored in a linear memory array
    - Memory is unified between instructions and data
      - The interpretation of a stored value depends on the control signals

  - **Sequential instruction processing**
    - One instruction processed (fetched, executed, completed) at a time
    - Program counter (instruction pointer) identifies the current instruction
    - Program counter is advanced sequentially except for control transfer instructions
LC-3: A von Neumann Machine
Another von Neumann Machine

Apple M1, 2021

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested
Another von Neumann Machine

Intel Alder Lake, 2021

Source: https://twitter.com/Locuza_/status/1454152714930331652
Another von Neumann Machine

AMD Ryzen 5000, 2020

Core Count: 8 cores/16 threads

L1 Caches: 32 KB per core

L2 Caches: 512 KB per core

L3 Cache: 32 MB shared

Another von Neumann Machine

IBM POWER10, 2020

Cores:
15-16 cores, 8 threads/core

L2 Caches:
2 MB per core

L3 Cache:
120 MB shared
LC-3: A von Neumann Machine

Figure 4.3 The LC-3 as an example of the von Neumann model
Stored Program & Sequential Execution

- Instructions and data are stored in memory
  - Typically, the instruction length is the word length

- The processor fetches instructions from memory sequentially
  - Fetches one instruction
  - Decodes and executes the instruction
  - Continues with the next instruction

- The address of the current instruction is stored in the program counter (PC)
  - If word-addressable memory, the processor increments the PC by 1 (in LC-3)
  - If byte-addressable memory, the processor increments the PC by the instruction length in bytes (4 in MIPS)
    - In MIPS, the OS typically sets the PC to 0x00400000 (start of a program)
A Sample Program Stored in Memory

- A sample MIPS program
  - 4 instructions stored in consecutive words in memory
    - No need to understand the program now. We will get back to it

MIPS assembly

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Byte Address</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $t2, 32($0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $s0, $s1, $s2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $t0, $s3, -12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub $t0, $t3, $t5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Machine code (encoded instructions)

<table>
<thead>
<tr>
<th>Byte Address</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>00400000C</td>
<td>016D4022</td>
</tr>
<tr>
<td>00400008</td>
<td>2268FF4</td>
</tr>
<tr>
<td>00400004</td>
<td>02328020</td>
</tr>
<tr>
<td>00400000</td>
<td>8C0A0020</td>
</tr>
</tbody>
</table>

← PC
The Instruction

- An instruction is the most basic unit of computer processing
  - Instructions are words in the language of a computer
  - Instruction Set Architecture (ISA) is the vocabulary

- The language of the computer can be written as
  - Machine language: Computer-readable representation (that is, 0’s and 1’s)
  - Assembly language: Human-readable representation

- We will study LC-3 instructions and MIPS instructions
  - Principles are similar in all ISAs (x86, ARM, RISC-V, ...)

66
The Instruction: Opcode & Operands

- An instruction is made up of two parts
  - Opcode and Operands

- Opcode specifies what the instruction does
- Operands specify who the instruction is to do it to

- Both are specified in instruction format (or instr. encoding)
  - An LC-3 instruction consists of 16 bits (bits [15:0])
  - Bits [15:12] specify the opcode → 16 distinct opcodes in LC-3
  - Bits [11:0] are used to figure out where the operands are

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- ADD
- R6
- R2
- R6
Instruction Types

- There are three main types of instructions
  - Operate instructions
    - Execute operations in the ALU
  - Data movement instructions
    - Read from or write to memory
  - Control flow instructions
    - Change the sequence of execution

- Let us start with some example instructions
An Example Operate Instruction

- **Addition**

<table>
<thead>
<tr>
<th>High-level code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = b + c;</code></td>
<td><code>add a, b, c</code></td>
</tr>
</tbody>
</table>

- **add**: mnemonic to indicate the operation to perform
- **b, c**: source operands
- **a**: destination operand
- **a ← b + c**
We map variables to registers

Assembly

\texttt{add a, b, c}

LC-3 registers

\begin{align*}
  b &= R1 \\
  c &= R2 \\
  a &= R0
\end{align*}

MIPS registers

\begin{align*}
  b &= $s1 \\
  c &= $s2 \\
  a &= $s0
\end{align*}
From Assembly to Machine Code in LC-3

- Addition

LC-3 assembly

```
ADD R0, R1, R2
```

Field Values

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR1</th>
<th>00</th>
<th>SR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Machine Code (Instruction Encoding)

```
0 0 0 1 0 0 0 0 0 1 0 1 0
```

0x1042

Machine Code, in short (hexadecimal)
Instruction Format (or Encoding)

- **LC-3 Operate Instruction Format**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OP</strong></td>
<td><strong>DR</strong></td>
<td><strong>SR1</strong></td>
<td>0</td>
<td><strong>00</strong></td>
<td><strong>SR2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 bits</td>
<td>3 bits</td>
<td>3 bits</td>
<td>3 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  - **OP** = opcode (what the instruction does)
    - E.g., **ADD** = 0001
      - Semantics: **DR** ← **SR1** + **SR2**
    - E.g., **AND** = 0101
      - Semantics: **DR** ← **SR1** AND **SR2**

  - **SR1**, **SR2** = source registers

  - **DR** = destination register
From Assembly to Machine Code in MIPS

- **Addition**

**MIPS assembly**

```mips
add $s0, $s1, $s2
```

**Field Values**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>16</td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

**Machine Code (Instruction Encoding)**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>10000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

```

```
0x02328020
```
Instruction Format: R-Type in MIPS

- **MIPS R-type Instruction Format**
  - 3 register operands

<table>
<thead>
<tr>
<th>0</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- 0 = opcode
- rs, rt = source registers
- rd = destination register
- shamt = shift amount (only shift operations)
- funct = operation in R-type instructions
Reading Operands from Memory

- With **operate instructions**, such as addition, we tell the computer to **execute arithmetic (or logic) computations** in the ALU.

- We also need instructions to **access the operands from memory**
  - Load them from memory to registers
  - Store them from registers to memory

- Next, we see how to **read (or load) from memory**

- **Writing (or storing)** is performed in a similar way, but we will talk about that later.
Reading Word-Addressable Memory

- **Load word**

<table>
<thead>
<tr>
<th>High-level code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = A[i]; )</td>
<td><code>load a, A, i</code></td>
</tr>
</tbody>
</table>

- **load**: mnemonic to indicate the load word operation

- **A**: base address

- **i**: offset
  - E.g., *immediate or literal* (a constant)

- **a**: destination operand

- **Semantics**: \( a \leftarrow \text{Memory}[A + i] \)
Load Word in LC-3 and MIPS

- LC-3 assembly

  High-level code
  \[ a = A[2]; \]

  LC-3 assembly
  \[ \text{LDR R3, R0, #2} \]
  \[ \text{R3} \leftarrow \text{Memory[R0 + 2]} \]

- MIPS assembly (assuming word-addressable)

  High-level code
  \[ a = A[2]; \]

  MIPS assembly
  \[ \text{lw }$s3, 2($s0)\]
  \[ \text{$s3} \leftarrow \text{Memory[$s0 + 2]} \]

These instructions use a particular addressing mode (i.e., the way the address is calculated), called **base+offset**
Load Word in Byte-Addressable MIPS

- **MIPS assembly**

  High-level code
  ```
  a = A[2];
  ```

  MIPS assembly
  ```
  lw $s3, 8($s0)
  ```

  ```
  $s3 ← Memory[$s0 + 8]
  ```

- Byte address is calculated as: `word_address * bytes/word`
  - 4 bytes/word in MIPS
  - If LC-3 were byte-addressable (i.e., LC-3b), 2 bytes/word
Instruction Format With Immediate

- **LC-3**

  LC-3 assembly

  \[
  \text{LDR } R3, R0, \#2
  \]

  Field Values

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>BaseR</th>
<th>offset6</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

- **MIPS**

  MIPS assembly

  \[
  \text{lw } \$s3, 8(\$s0)
  \]

  Field Values

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>16</td>
<td>19</td>
<td>8</td>
</tr>
</tbody>
</table>

  **I-Type**
Instruction (Processing) Cycle
How Are These Instructions Executed?

- By using instructions, we can speak the language of the computer.

- Thus, we now know how to tell the computer to:
  - Execute computations in the ALU by using, for instance, an addition.
  - Access operands from memory by using the load word instruction.

- But, how are these instructions executed on the computer?
  - The process of executing an instruction is called the instruction cycle (or, instruction processing cycle).
The Instruction Cycle

- The instruction cycle is a sequence of steps or **phases**, that an instruction goes through to be executed
  - **FETCH**
  - **DECODE**
  - **EVALUATE ADDRESS**
  - **FETCH OPERANDS**
  - **EXECUTE**
  - **STORE RESULT**

- **Not all instructions require the six phases**
  - LDR does **not** require EXECUTE
  - ADD does **not** require EVALUATE ADDRESS
  - Intel x86 instruction **ADD [eax], edx** is an example of instruction with six phases
After STORE RESULT, a New FETCH

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT
The FETCH phase obtains the instruction from memory and loads it into the Instruction Register (IR).

This phase is common to every instruction type.

Complete description:

- Step 1: Load the MAR with the contents of the PC, and simultaneously increment the PC.
- Step 2: Interrogate memory. This results in the instruction being placed in the MDR by memory.
- Step 3: Load the IR with the contents of the MDR.
FETCH in LC-3

Step 1: Load MAR and increment PC

Step 2: Access memory

Step 3: Load IR with the content of MDR

Figure 4.3 The LC-3 as an example of the von Neumann model
The DECODE phase identifies the instruction
- Also generates the set of control signals to process the identified instruction in later phases of the instruction cycle

Recall the decoder (from Lecture 5)
- A 4-to-16 decoder identifies which of the 16 opcodes is going to be processed

- The input is the four bits IR[15:12]

- The remaining 12 bits identify what else is needed to process the instruction
DECODE in LC-3

DECODE identifies the instruction to be processed.

Also generates the set of control signals to process the instruction.

Figure 4.3  The LC-3 as an example of the von Neumann model
Recall: Decoder

- “Input pattern detector”
- $n$ inputs and $2^n$ outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The output that is logically 1 is the output corresponding to the input pattern that the logic circuit is expected to detect
- Example: 2-to-4 decoder

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_0$</th>
<th>$Y_3$</th>
<th>$Y_2$</th>
<th>$Y_1$</th>
<th>$Y_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

![2:4 Decoder Diagram]
Recall: Decoder (II)

- The decoder is useful in determining how to interpret a bit pattern
  - It could be the address of a location in memory, that the processor intends to read from
  - It could be an instruction in the program and the processor needs to decide what action to take (based on instruction opcode)
To Come: Full State Machine for LC-3b

Figure C.2: A state machine for the LC-3b

Decode State

The EVALUATE ADDRESS phase computes the address of the memory location that is needed to process the instruction.

This phase is necessary in LDR:
- It computes the address of the data word that is to be read from memory.
- By adding an offset to the content of a register.

But not necessary in ADD.
EVALUATE ADDRESS in LC-3

LDR calculates the address by adding a register and an immediate.
FETCH OPERANDS

- The FETCH OPERANDS phase obtains the source operands needed to process the instruction

- In LDR
  - Step 1: Load MAR with the address calculated in EVALUATE ADDRESS
  - Step 2: Read memory, placing source operand in MDR

- In ADD
  - Obtain the source operands from the register file
  - In some microprocessors, operand fetch from register file can be done at the same time the instruction is being decoded
FETCH OPERANDS in LC-3

LDR loads MAR (step 1), and places the results in MDR (step 2)
The EXECUTE phase executes the instruction

- In ADD, it performs addition in the ALU
- In XOR, it performs bitwise XOR in the ALU
- ...
EXECUTE in LC-3

ADD adds SR1 and SR2
STORE RESULT

- The STORE RESULT phase writes the result to the designated destination

- Once STORE RESULT is completed, a new instruction cycle starts (with the FETCH phase)
ADD loads ALU Result into DR

Figure 4.3 The LC-3 as an example of the von Neumann model
STORE RESULT in LC-3

LDR loads MDR into DR

Figure 4.3 The LC-3 as an example of the von Neumann model
The Instruction Cycle

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT
Changing the Sequence of Execution

- A computer program **executes in sequence** (i.e., in program order)
  - First instruction, second instruction, third instruction and so on

- Unless we **change the sequence of execution**

- **Control instructions** allow a program to execute **out of sequence**
  - They can change the PC by loading it during the EXECUTE phase
  - That wipes out the incremented PC (loaded during the FETCH phase)
Jump in LC-3

- Unconditional branch or jump

**LC-3**

![JMP R2](image)

<table>
<thead>
<tr>
<th>1100</th>
<th>000</th>
<th>BaseR</th>
<th>000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>3 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **BaseR** = Base register
- **PC ← R2** (Register identified by BaseR)

**Variations**
- RET: special case of JMP where BaseR = R7
- JSR, JSRR: jump to subroutine

This is register addressing mode
Jump in MIPS

- Unconditional branch or jump

- MIPS

  \[
  j \ target
  \]

  \[
  \begin{array}{|c|c|}
  \hline
  2 & \text{target} \\
  \hline
  \end{array}
  \]

  6 bits 26 bits

  - 2 = opcode
  - target = target address

  - \( \text{PC} \leftarrow \text{PC}^+\left[31:28\right] \mid \text{sign-extend(target)} \ast 4 \)

  - Variations
    - jal: jump and link (function calls)
    - jr: jump register

  \[
  \text{jr} \$s0
  \]

  \( j \) uses pseudo-direct addressing mode

  \( jr \) uses register addressing mode

\( ^\dagger \) This is the incremented PC
PC UPDATE in LC-3

JMP loads SR1 into PC
Control of the Instruction Cycle

State 1
- The FSM asserts GatePC and LD.MAR
- It selects input (+1) in PCMUX and asserts LD.PC

State 2
- MDR is loaded with the instruction

State 3
- The FSM asserts GateMDR and LD.IR

State 4
- The FSM goes to next state depending on opcode

State 63
- JMP loads register into PC

Full state diagram in Patt&Pattel, Appendix C
The Instruction Cycle

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT
LC-3 and MIPS
Instruction Set Architectures
Agenda for Today & Next Few Lectures

- The von Neumann model
- LC-3: An example of von Neumann machine
- LC-3 and MIPS Instruction Set Architectures
- LC-3 and MIPS assembly and programming
- Introduction to microarchitecture and single-cycle microarchitecture
- Multi-cycle microarchitecture
The Instruction Set

- It defines opcodes, data types, and addressing modes
- ADD and LDR have been our first examples

### ADD

<table>
<thead>
<tr>
<th></th>
<th>OP</th>
<th>DR</th>
<th>SR1</th>
<th>SR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Register mode

### LDR

<table>
<thead>
<tr>
<th></th>
<th>OP</th>
<th>DR</th>
<th>BaseR</th>
<th>offset6</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>6</td>
<td>3</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Base+offset mode
The Instruction Set Architecture

- The ISA is the **interface between** what the **software** commands and what the **hardware** carries out

- The ISA specifies
  - **The memory organization**
    - Address space (LC-3: $2^{16}$, MIPS: $2^{32}$)
    - Addressability (LC-3: 16 bits, MIPS: 8 bits)
      - Word- or Byte-addressable
  - **The register set**
    - 8 registers (R0 to R7) in LC-3
    - 32 registers in MIPS
  - **The instruction set**
    - **Opcodes**
    - **Data types**
    - **Addressing modes**
    - Length and format of instructions
Instructions (Opcodes)
Opcodes

- A large or small set of opcodes could be defined
  - E.g, HP Precision Architecture: an instruction for A*B+C
  - E.g, x86 ISA: multimedia extensions (MMX), later SSE and AVX
  - E.g, VAX ISA: opcode to save all information of one program prior to switching to another program

- Tradeoffs are involved. Examples:
  - Hardware complexity vs. software complexity
  - Latency of simple vs. complex instructions

- In LC-3 and in MIPS there are three types of opcodes
  - Operate
  - Data movement
  - Control
## Opcodes in LC-3

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD+</td>
<td>0001</td>
<td>DR, SR1, 0, 0, SR2</td>
</tr>
<tr>
<td>ADD+</td>
<td>0001</td>
<td>DR, SR1, 1, imm5</td>
</tr>
<tr>
<td>AND+</td>
<td>0101</td>
<td>DR, SR1, 0, 0, SR2</td>
</tr>
<tr>
<td>AND+</td>
<td>0101</td>
<td>DR, SR1, 1, imm5</td>
</tr>
<tr>
<td>BR</td>
<td>0000</td>
<td>n, z, p, PCoffset9</td>
</tr>
<tr>
<td>JMP</td>
<td>1100</td>
<td>000, BaseR, 000000</td>
</tr>
<tr>
<td>JSR</td>
<td>0100</td>
<td>1, PCoffset11</td>
</tr>
<tr>
<td>JSRR</td>
<td>0100</td>
<td>0, 0, BaseR, 000000</td>
</tr>
<tr>
<td>LD+</td>
<td>0010</td>
<td>DR, PCoffset9</td>
</tr>
<tr>
<td>LDI+</td>
<td>1010</td>
<td>DR, PCoffset9</td>
</tr>
<tr>
<td>LDR+</td>
<td>0110</td>
<td>DR, BaseR, offset6</td>
</tr>
<tr>
<td>LEA+</td>
<td>1110</td>
<td>DR, PCoffset9</td>
</tr>
<tr>
<td>NOT+</td>
<td>1001</td>
<td>DR, SR, 111111</td>
</tr>
<tr>
<td>RET</td>
<td>1100</td>
<td>000, 111, 000000</td>
</tr>
<tr>
<td>RTI</td>
<td>1000</td>
<td>000000000000</td>
</tr>
<tr>
<td>ST</td>
<td>0011</td>
<td>SR, PCoffset9</td>
</tr>
<tr>
<td>STI</td>
<td>1011</td>
<td>SR, PCoffset9</td>
</tr>
<tr>
<td>STR</td>
<td>0111</td>
<td>SR, BaseR, offset6</td>
</tr>
<tr>
<td>TRAP</td>
<td>1111</td>
<td>0000, trapvect8</td>
</tr>
<tr>
<td>reserved</td>
<td>1101</td>
<td></td>
</tr>
</tbody>
</table>
# Opcodes in LC-3b

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADD</strong></td>
<td>0001 DR SR1 A</td>
<td>op.spec</td>
</tr>
<tr>
<td><strong>AND</strong></td>
<td>0101 DR SR1 A</td>
<td>op.spec</td>
</tr>
<tr>
<td><strong>BR</strong></td>
<td>0000 n z p</td>
<td>PC.offset9</td>
</tr>
<tr>
<td><strong>JMP</strong></td>
<td>1100 000 BaseR 000000</td>
<td></td>
</tr>
<tr>
<td><strong>JSR(R)</strong></td>
<td>0100 A</td>
<td>operand specifier</td>
</tr>
<tr>
<td><strong>LDB</strong></td>
<td>0010 DR BaseR</td>
<td>boffset6</td>
</tr>
<tr>
<td><strong>LDW</strong></td>
<td>0110 DR BaseR</td>
<td>offset6</td>
</tr>
<tr>
<td><strong>LEA</strong></td>
<td>1110 DR</td>
<td>PC.offset9</td>
</tr>
<tr>
<td><strong>RTI</strong></td>
<td>1000</td>
<td>00000000000000</td>
</tr>
<tr>
<td><strong>SHF</strong></td>
<td>1101 DR SR A D</td>
<td>amount4</td>
</tr>
<tr>
<td><strong>STB</strong></td>
<td>0011 SR BaseR</td>
<td>boffset6</td>
</tr>
<tr>
<td><strong>STW</strong></td>
<td>0111 SR BaseR</td>
<td>offset6</td>
</tr>
<tr>
<td><strong>TRAP</strong></td>
<td>1111 0000</td>
<td>trapvec18</td>
</tr>
<tr>
<td><strong>XOR</strong></td>
<td>1001 DR SR1 A</td>
<td>op.spec</td>
</tr>
</tbody>
</table>

**not used**

| 1010 | |
| 1011 | |
MIPS Instruction Types

<table>
<thead>
<tr>
<th></th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>6-bit</td>
</tr>
</tbody>
</table>

**R-type**

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>16-bit</td>
</tr>
</tbody>
</table>

**I-type**

<table>
<thead>
<tr>
<th>opcode</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>26-bit</td>
</tr>
</tbody>
</table>

**J-type**
Funct in MIPS R-Type Instructions (I)

**Table B.2** R-type instructions, sorted by funct field

<table>
<thead>
<tr>
<th>Funct</th>
<th>Name</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>sll rd, rt, shamt</td>
<td>shift left logical</td>
<td>([rd] = [rt] \ll \text{shamt})</td>
</tr>
<tr>
<td>000010</td>
<td>srl rd, rt, shamt</td>
<td>shift right logical</td>
<td>([rd] = [rt] \gg \text{shamt})</td>
</tr>
<tr>
<td>000011</td>
<td>sra rd, rt, shamt</td>
<td>shift right arithmetic</td>
<td>([rd] = [rt] \ggg \text{shamt})</td>
</tr>
<tr>
<td>000100</td>
<td>sllv rd, rt, rs</td>
<td>shift left logical variable</td>
<td>([rd] = [rt] \ll [rs]_{4:0})</td>
</tr>
<tr>
<td>000110</td>
<td>srlv rd, rt, rs</td>
<td>shift right logical variable</td>
<td>([rd] = [rt] \gg [rs]_{4:0})</td>
</tr>
<tr>
<td>000111</td>
<td>srav rd, rt, rs</td>
<td>shift right arithmetic variable</td>
<td>([rd] = [rt] \ggg [rs]_{4:0})</td>
</tr>
<tr>
<td>001000</td>
<td>jr rs</td>
<td>jump register</td>
<td>(PC = [rs])</td>
</tr>
<tr>
<td>001001</td>
<td>jal rs</td>
<td>jump and link register</td>
<td>($ra = PC + 4, PC = [rs])</td>
</tr>
<tr>
<td>001100</td>
<td>syscall</td>
<td>system call</td>
<td>system call exception</td>
</tr>
<tr>
<td>001101</td>
<td>break</td>
<td>break</td>
<td>break exception</td>
</tr>
<tr>
<td>010000</td>
<td>mfhi rd</td>
<td>move from hi</td>
<td>([rd] = [hi])</td>
</tr>
<tr>
<td>010001</td>
<td>mthi rs</td>
<td>move to hi</td>
<td>([hi] = [rs])</td>
</tr>
<tr>
<td>010010</td>
<td>mflo rd</td>
<td>move from lo</td>
<td>([rd] = [lo])</td>
</tr>
<tr>
<td>010011</td>
<td>mtlo rs</td>
<td>move to lo</td>
<td>([lo] = [rs])</td>
</tr>
<tr>
<td>011000</td>
<td>mult rs, rt</td>
<td>multiply</td>
<td>([hi], [lo] = [rs] \times [rt])</td>
</tr>
<tr>
<td>011001</td>
<td>multu rs, rt</td>
<td>multiply unsigned</td>
<td>([hi], [lo] = [rs] \times [rt])</td>
</tr>
<tr>
<td>011010</td>
<td>div rs, rt</td>
<td>divide</td>
<td>([lo] = [rs]/[rt], [hi] = [rs]%[rt])</td>
</tr>
<tr>
<td>011011</td>
<td>divu rs, rt</td>
<td>divide unsigned</td>
<td>([lo] = [rs]/[rt], [hi] = [rs]%[rt])</td>
</tr>
</tbody>
</table>

Opcode is 0 in MIPS R-Type instructions. **Funct** defines the operation.

(continued)
More complete list of instructions are in H&H Appendix B
Data Types
Data Types

- An ISA supports one or several data types

- LC-3 only supports 2’s complement integers
  - Negative of a 2’s complement binary value $X = \text{NOT}(X) + 1$

- MIPS supports
  - 2’s complement integers
  - Unsigned integers
  - Floating point

- Tradeoffs are involved. Examples:
  - Hardware complexity vs. software complexity
  - Latency of operations on supported vs. unsupported data types
Why Have Different Data Types in ISA?

- An example of programmer vs. microarchitect tradeoff

- **Advantage of more data types:**
  - Enables better mapping of high-level programming constructs to hardware
  - Hardware can directly operate on data types present in programming languages → small number of instructions and code size
    - Matrix operations vs. individual multiply/add/load/store instructions
    - Graph operations vs. individual load/store/add/... instructions

- **Disadvantage:**
  - More work for the microarchitect
    - who needs to implement the data types and instructions that operate on data types
Data Types and Instruction Complexity

- Data types are coupled tightly to the semantic level, or complexity of instructions.

- Concept of **semantic gap**
  - how close instructions & data types are to high-level language

- **Complex instructions + data types → small semantic gap**
  - E.g., insert into a doubly linked list, multiply two matrices
  - VAX ISA: doubly-linked list, multi-dimensional arrays

- **Simple instructions + data types → large semantic gap**
  - E.g., primitive operations: load, store, multiply, add, nor
  - Early RISC machines: Only integer data type, simple operations
Semantic Gap

- How close instructions & data types are to high-level language (HLL)

Small Semantic Gap

Large Semantic Gap
- **Complex instruction**: An instruction **does a lot of work**, e.g. many operations
  - Insert in a doubly linked list
  - Compute FFT
  - String copy
  - Matrix multiply
  - ...

- **Simple instruction**: An instruction **does little work** -- it is a primitive using which complex operations can be built
  - Add
  - XOR
  - Multiply
  - ...
Complex vs. Simple Instructions + Data Types

- **Advantages of Complex Instructions + Data Types**
  + Denser encoding $\rightarrow$ smaller code size $\rightarrow$ better memory utilization, saves off-chip bandwidth, better cache hit rate (better packing of instructions)
  + Simpler compiler: no need to optimize small instructions as much

- **Disadvantages of Complex Instructions + Data Types**
  - Larger chunks of work $\rightarrow$ compiler has less opportunity to optimize (limited in fine-grained optimizations it can do)
  - More complex hardware $\rightarrow$ translation from a high level to control signals and optimization needs to be done by hardware
Aside: An Example: **Binary Coded Decimal**

- Each decimal digit is encoded with a fixed number of bits

![Binary Coded Decimal Diagram](http://commons.wikimedia.org/wiki/File:Binary_clock.svg?raw=true)

"Binary clock" by Alexander Jones & Eric Pierce - Own work, based on Wapcaplet's Binary clock.png on the English Wikipedia. Licensed under CC BY-SA 3.0 via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Binary_clock.svg#mediaviewer/File:Binary_clock.svg

Aside: An Example: **Binary Coded Decimal**

- Each decimal digit is encoded with a fixed number of bits
Addressing Modes
Addressing Modes

- An addressing mode is a mechanism for specifying where an operand is located

- There are five addressing modes in LC-3
  - Immediate or literal (constant)
    - The operand is in some bits of the instruction
  - Register
    - The operand is in one of R0 to R7 registers
  - Three memory addressing modes
    - PC-relative
    - Indirect
    - Base+offset

- MIPS has pseudo-direct addressing (for j and jal), additionally, but does not have indirect addressing
Why Have Different Addressing Modes?

- Another example of programmer vs. microarchitect tradeoff

- **Advantage of more addressing modes:**
  - Enables better mapping of high-level programming constructs to hardware
  - Some accesses are better expressed with a different mode \(\rightarrow\) reduced number of instructions and code size
    - Array indexing
    - Pointer-based accesses (indirection)
    - Sparse matrix accesses

- **Disadvantages:**
  - More work for the microarchitect
  - More options for the compiler to decide what to use
Semantic Gap Applies to Addressing Modes

- How close instructions & data types & addressing modes are to high-level language (HLL)

**Diagram:**
- **Small Semantic Gap**
  - HLL
  - ISA with Complex Inst & Data Types & Addressing Modes
  - HW Control Signals

- **Large Semantic Gap**
  - HLL
  - ISA with Simple Inst & Data Types & Addressing Modes
  - HW Control Signals
Many Tradeoffs in ISA Design...

- Execution model – sequencing model and processing style
- Instruction length
- Instruction format
- Instruction types
- Instruction complexity vs. simplicity
- Data types
- Number of registers
- Addressing mode types
- Memory organization (address space, addressability, endianness, ...)
- Memory access restrictions and permissions
- Support for multiple instructions to execute in parallel?
- ...
Operate Instructions
Operate Instructions

- In **LC-3**, there are three operate instructions
  - NOT is a **unary operation** (one source operand)
    - It executes bitwise NOT
  - ADD and AND are **binary operations** (two source operands)
    - ADD is 2’s complement addition
    - AND is bitwise SR1 & SR2

- In **MIPS**, there are many more
  - Most of **R-type** instructions (they are **binary operations**)
    - E.g., add, and, nor, xor...
  - **I-type** versions (i.e., with one immediate operand) of the R-type operate instructions
  - **F-type** operations, i.e., floating-point operations
NOT in LC-3

- NOT assembly and machine code

**LC-3 assembly**

```
NOT R3, R5
```

**Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

**Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1</td>
<td>0 1 1</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>

There is no NOT in MIPS. How is it implemented?
Operate Instructions

- We are already familiar with LC-3’s ADD and AND with register mode (R-type in MIPS)

- Now let us see the versions with one literal (i.e., immediate) operand

- We will use Subtraction as an example
  - How is it implemented in LC-3 and MIPS?
Recall: LC-3 Operate Instruction Format

LC-3 Operate Instruction Format (Register OP Register)

- OP = opcode (what the instruction does)
  - E.g., ADD = 0001
    - Semantics: DR ← SR1 + SR2
  - E.g., AND = 0101
    - Semantics: DR ← SR1 AND SR2

- SR1, SR2 = source registers

- DR = destination register
Operate Instr. with one Literal in LC-3

- **ADD and AND**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR1</th>
<th>1</th>
<th>imm5</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>3 bits</td>
<td>3 bits</td>
<td>5 bits</td>
<td></td>
</tr>
</tbody>
</table>

- **OP = operation**
  - E.g., **ADD = 0001** (same OP as the register-mode ADD)
    - DR ← SR1 + sign-extend(imm5)
  - E.g., **AND = 0101** (same OP as the register-mode AND)
    - DR ← SR1 AND sign-extend(imm5)

- **SR1 = source register**
- **DR = destination register**
- **imm5 = Literal or immediate (sign-extend to 16 bits)**
ADD with one Literal in LC-3

- **ADD assembly and machine code**

**LC-3 assembly**

ADD R1, R4, #−2

**Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR</th>
<th>imm5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-2</td>
<td></td>
</tr>
</tbody>
</table>

**Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR</th>
<th>imm5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1</td>
<td>0 0 1</td>
<td>1 0 0</td>
<td>1 1 1 1 0</td>
</tr>
</tbody>
</table>

![Diagram of Register file and Instruction register](image)
ADD with one Literal in LC-3 Data Path
Instructions with one Literal in MIPS

- I-type MIPS Instructions
  - 2 register operands and immediate
- Some operate and data movement instructions

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- opcode = operation
- rs = source register
- rt =
  - destination register in some instructions (e.g., addi, lw)
  - source register in others (e.g., sw)
- imm = Literal or immediate
ADD with one Literal in MIPS

- **Add immediate**

  **MIPS assembly**

  ```
  addi $s0, $s1, 5
  ```

  **Field Values**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>17</td>
<td>16</td>
<td>5</td>
</tr>
</tbody>
</table>

  \[ rt \leftarrow rs + \text{sign-extend}(imm) \]

  **Machine Code**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>10001</td>
<td>10010</td>
<td>0000 0000 0000 0101</td>
</tr>
</tbody>
</table>

  0x22300005
Subtraction in MIPS vs. LC-3

- **MIPS assembly**
  
  High-level code
  
  \[ a = b + c - d; \]
  
  MIPS assembly
  
  ```
  add $t0, $s0, $s1
  sub $s3, $t0, $s2
  ```

- **LC-3 assembly**
  
  High-level code
  
  \[ a = b + c - d; \]
  
  LC-3 assembly
  
  ```
  ADD R2, R0, R1
  NOT R4, R3
  ADD R5, R4, #1
  ADD R6, R2, R5
  ```

- **Tradeoff in LC-3**
  
  - More instructions
  - But, simpler control logic
Subtract Immediate

- **MIPS assembly**

  **High-level code**
  
  ```
  a = b - 3;
  ```

  **MIPS assembly**
  
  ```
  subi $s1, $s0, 3
  ```

  **Is subi necessary in MIPS?**

- **LC-3**

  **High-level code**
  
  ```
  a = b - 3;
  ```

  **LC-3 assembly**
  
  ```
  ADD R1, R0, #-3
  ```
Data Movement Instructions and Addressing Modes
Data Movement Instructions

- In **LC-3**, there are seven data movement instructions
  - LD, LDR, LDI, LEA, ST, STR, STI

- Format of load and store instructions
  - **Opcode** (bits [15:12])
  - **DR or SR** (bits [11:9])
  - **Address generation bits** (bits [8:0])
  - Four ways to interpret bits, called **addressing modes**
    - PC-Relative Mode
    - Indirect Mode
    - Base+Offset Mode
    - Immediate Mode

- In **MIPS**, there are only **Base+offset** and **Immediate modes** for load and store instructions
PC-Relative Addressing Mode

- **LD (Load) and ST (Store)**

  - **OP = opcode**
    - E.g., LD = 0010
    - E.g., ST = 0011
  
  - **DR = destination register in LD**
  
  - **SR = source register in ST**
  
  - **LD:** \( DR \leftarrow \text{Memory}[\text{PC}^\dagger + \text{sign-extend}(\text{PCoffset9})] \)
  
  - **ST:** \( \text{Memory}[\text{PC}^\dagger + \text{sign-extend}(\text{PCoffset9})] \leftarrow \text{SR} \)

\[\text{OP} \quad \text{DR/SR} \quad \text{PCoffset9}\]

- 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

  - 4 bits
  - 3 bits
  - 9 bits

\[^\dagger\text{This is the incremented PC}\]
**LD in LC-3**

- **LD assembly and machine code**

**LC-3 assembly**

```
LD R2, 0x1AF
```

**Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>0x1AF</td>
</tr>
</tbody>
</table>

**Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0</td>
<td>0 1 0</td>
<td>1 1 0 1 0 1 1 1 1</td>
</tr>
</tbody>
</table>

The memory address is **only +255 to -256 locations away of the LD or ST instruction**

**Limitation:** The PC-relative addressing mode cannot address far away from the instruction
Indirect Addressing Mode

- **LDI (Load Indirect) and STI (Store Indirect)**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR/SR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>3 bits</td>
<td>9 bits</td>
</tr>
</tbody>
</table>

- **OP = opcode**
  - E.g., LDI = 1010
  - E.g., STI = 1011

- **DR = destination register in LDI**
- **SR = source register in STI**

- **LDI:** \( DR \leftarrow \text{Memory[Memory[PC}^\dagger \text{ + sign-extend(PCoffset9)]]} \)

- **STI:** \( \text{Memory[Memory[PC}^\dagger \text{ + sign-extend(PCoffset9)]]} \leftarrow SR \)

\(^\dagger\text{This is the incremented PC}\)
LDI in LC-3

- LDI assembly and machine code

**LC-3 assembly**

LDI R3, 0x1CC

**Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>0x1CC</td>
</tr>
</tbody>
</table>

**Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>011</td>
<td>111001100</td>
</tr>
</tbody>
</table>

Now the address of the operand can be anywhere in the memory
Base+Offset Addressing Mode

- LDR (Load Register) and STR (Store Register)

<table>
<thead>
<tr>
<th>OP</th>
<th>DR/SR</th>
<th>BaseR</th>
<th>offset6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>3 bits</td>
<td>3 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- OP = opcode
  - E.g., LDR = 0110
  - E.g., STR = 0111

- DR = destination register in LDR
- SR = source register in STR

- LDR: \( DR \leftarrow \text{Memory[BaseR + sign-extend(offset6)]} \)

- STR: \( \text{Memory[BaseR + sign-extend(offset6)]} \leftarrow \text{SR} \)
LDR in LC-3

- LDR assembly and machine code

LC-3 assembly

LDR R1, R2, 0x1D

Field Values

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>BaseR</th>
<th>offset6</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>0x1D</td>
</tr>
</tbody>
</table>

Machine Code

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>BaseR</th>
<th>offset6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>001</td>
<td>010</td>
<td>011101</td>
</tr>
</tbody>
</table>

Again, the address of the operand can be anywhere in the memory.
Address Calculation in LC-3 Data Path

- Global bus
- MAR Multiplexer
- Adder
- Sign extension (Address)
In MIPS, `lw` and `sw` use base+offset mode (or base addressing mode).

High-level code

\[ A[2] = a; \]

MIPS assembly

\[ \text{sw } \$s3, \ 8(\$s0) \]

Memory[\$s0 + 8] \leftarrow \$s3

**Field Values**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>16</td>
<td>19</td>
<td>8</td>
</tr>
</tbody>
</table>

**imm** is the 16-bit offset, which is sign-extended to 32 bits.
An Example Program in MIPS and LC-3

<table>
<thead>
<tr>
<th>High-level code</th>
<th>MIPS registers</th>
<th>LC-3 registers</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = A[0];</code></td>
<td><code>A = $s0</code></td>
<td><code>A = R0</code></td>
</tr>
<tr>
<td><code>c = a + b - 5;</code></td>
<td><code>b = $s2</code></td>
<td><code>b = R2</code></td>
</tr>
<tr>
<td><code>B[0] = c;</code></td>
<td><code>B = $s1</code></td>
<td><code>B = R1</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MIPS assembly</th>
<th>LC-3 assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw $t0, 0($s0)</code></td>
<td><code>LDR R5, R0, #0</code></td>
</tr>
<tr>
<td><code>add $t1, $t0, $s2</code></td>
<td><code>ADD R6, R5, R2</code></td>
</tr>
<tr>
<td><code>addi $t2, $t1, -5</code></td>
<td><code>ADD R7, R6, #-5</code></td>
</tr>
<tr>
<td><code>sw $t2, 0($s1)</code></td>
<td><code>STR R7, R1, #0</code></td>
</tr>
</tbody>
</table>
Immediate Addressing Mode (in LC-3)

- **LEA (Load Effective Address)**

  - **OP = 1110**
  - **DR = destination register**
  - **LEA:** \( DR \leftarrow PC^\dagger + \text{sign-extend}(PC\text{offset9}) \)

What is the **difference from PC-Relative addressing mode**?

**Answer:** Instructions with **PC-Relative mode** load from memory, but **LEA does not → Hence the name Load Effective Address**

\(^\dagger\) This is the incremented PC
LEA in LC-3

LEA assembly and machine code

**LC-3 assembly**

LEA R5, #-3

**Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>5</td>
<td>0x1FD</td>
</tr>
</tbody>
</table>

**Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>101</td>
<td>1111111101</td>
</tr>
</tbody>
</table>
Address Calculation in LC-3 Data Path

- Global bus
- MAR Multiplexer
- Adder
- Sign extension (Address)
Immediate Addressing Mode in MIPS

- In MIPS, `lui` (load upper immediate) loads a 16-bit immediate into the upper half of a register and sets the lower half to 0.

- It is used to assign 32-bit constants to a register.

High-level code

```
a = 0x6d5e4f3c;
```

MIPS assembly

```
# $s0 = a
lui  $s0, 0x6d5e
ori  $s0, 0x4f3c
```
### Addressing Example in LC-3

- What is the final value of R3?

#### P&P, Chapter 5.3.5

| Address | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| x30F6  | 1  | 1  | 1  | 0  | 0  | 0  | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |   |
| x30F7  | 0  | 0  | 0  | 1  | 0  | 1  | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |   |
| x30F8  | 0  | 0  | 1  | 1  | 0  | 1  | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |   |
| x30F9  | 0  | 1  | 0  | 1  | 0  | 1  | 0 | 1 | 0 | 0 | 0 | 0 | 0 |   |   |   |
| x30FA  | 0  | 0  | 0  | 1  | 0  | 1  | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |   |
| x30FB  | 0  | 1  | 1  | 1  | 0  | 1  | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |   |
| x30FC  | 1  | 0  | 1  | 0  | 0  | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |   |

\[
\begin{align*}
R1 & \leftarrow \text{PC} - 3 \\
R2 & \leftarrow R1 + 14 \\
M[x30F4] & \leftarrow R2 \\
R2 & \leftarrow 0 \\
R2 & \leftarrow R2 + 5 \\
M[R1 + 14] & \leftarrow R2 \\
R3 & \leftarrow M[M[x30F4]]
\end{align*}
\]
Addressing Example in LC-3

What is the final value of R3?

The final value of R3 is 5

P&P, Chapter 5.3.5
Control Flow Instructions
Control Flow Instructions

- Allow a program to execute **out of sequence**

- Conditional branches and unconditional jumps
  - Conditional branches are used to **make decisions**
    - E.g., if-else statement
  - In LC-3, three **condition codes** are used
  - **Jumps** are used to implement
    - Loops
    - Function calls
  - **JMP** in LC-3 and **j** in MIPS
    - We have already seen these
Conditional Control Flow (Conditional Branching)
Condition Codes in LC-3

- Each time one GPR (R0-R7) is written, three single-bit registers are updated.

- Each of these condition codes are either set (set to 1) or cleared (set to 0).
  - If the written value is negative:
    - N is set, Z and P are cleared.
  - If the written value is zero:
    - Z is set, N and P are cleared.
  - If the written value is positive:
    - P is set, N and Z are cleared.

- x86 and SPARC are examples of ISAs that use condition codes.
Conditional Branches in LC-3

- BRz (Branch if Zero)

  - \( \text{BRz \ PCoffset9} \)

  - 0000  n  z  p  PCoffset9
    
    4 bits  9 bits

  - \( n, z, p = \text{which condition code is tested} \ (N, Z, \text{and/or P}) \)
    
    - \( n, z, p: \text{instruction bits to identify the condition codes to be tested} \)
    
    - \( N, Z, P: \text{values of the corresponding condition codes} \)

  - \( \text{PCoffset9} = \text{immediate or constant value} \)

  - if \( ((n \ \text{AND} \ N) \ \text{OR} \ (p \ \text{AND} \ P) \ \text{OR} \ (z \ \text{AND} \ Z)) \)
    
    then \( \text{PC} \leftarrow \text{PC}^{\dagger} + \text{sign-extend(PCoffset9)} \)

  - \( \text{Variations: BRn, BRz, BRp, BRzp, BRnp, BRnz, BRnzp} \)

\( ^{\dagger} \text{This is the incremented PC} \)
Conditional Branches in LC-3

- **BRz**

BRz  0x0D9

What if \( n = z = p = 1? \)
(i.e., BRnzp)

And what if \( n = z = p = 0? \)

\( n, z, p \) are the instruction bits to identify the condition codes to be tested
Conditional Branches in MIPS

- **beq (Branch if Equal)**

  
  ```
  beq $s0, $s1, offset
  ```

<table>
<thead>
<tr>
<th>4</th>
<th>rs</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

  - 4 = opcode
  - rs, rt = source registers
  - offset = immediate or constant value
  - if rs == rt
    - then PC ← PC† + sign-extend(offset) * 4
  - Variations: beq, bne, blez, bgtz

† This is the incremented PC
This is an example of tradeoff in the instruction set

- The same functionality requires more instructions in LC-3
- But, the control logic requires more complexity in MIPS
What We Learned

- Basic elements of a computer & the von Neumann model
  - LC-3: An example von Neumann machine

- Instruction Set Architectures: LC-3 and MIPS
  - Operate instructions
  - Data movement instructions
  - Control instructions

- Instruction formats

- Addressing modes
There Is A Lot More to Cover on ISAs

https://www.youtube.com/onurmutlulectures
Many Different ISAs Over Decades

- x86
- PDP-x: Programmed Data Processor (PDP-11)
- VAX
- IBM 360
- CDC 6600
- SIMD ISAs: CRAY-1, Connection Machine
- VLIW ISAs: Multiflow, Cydrome, IA-64 (EPIC)
- PowerPC, POWER
- RISC ISAs: Alpha, MIPS, SPARC, ARM, RISC-V, ...

What are the fundamental differences?

- E.g., how instructions are specified and what they do
- E.g., how complex are instructions, data types, addr. modes
Complex vs. Simple Instructions + Data Types

- **Complex instruction**: An instruction does a lot of work, e.g. many operations
  - Insert in a doubly linked list
  - Compute FFT
  - String copy
  - Matrix multiply
  - ...

- **Simple instruction**: An instruction does little work -- it is a primitive using which complex operations can be built
  - Add
  - XOR
  - Multiply
  - ...
Complex vs. Simple Instructions + Data Types

- **Advantages of Complex Instructions + Data Types**
  + **Denser encoding** → smaller code size → better memory utilization, saves off-chip bandwidth, better cache hit rate (better packing of instructions)
  + **Simpler compiler**: no need to optimize small instructions as much

- **Disadvantages of Complex Instructions + Data Types**
  - Larger chunks of work → compiler has less opportunity to optimize (limited in fine-grained optimizations it can do)
  - More complex hardware → translation from a high level to control signals and optimization needs to be done by hardware
Semantic Gap

- How close instructions & data types & addressing modes are to high-level language (HLL)

Easier mapping of HLL to ISA
Less work for software designer
More work for hardware designer
Optimization burden on HW

Harder mapping of HLL to ISA
More work for software designer
Less work for hardware designer
Optimization burden on SW
How to Change the Semantic Gap Tradeoffs

- Translate from one ISA into a different “implementation” ISA

X86-64

ISA with
Complex Inst
& Data Types
& Addressing Modes

Small Semantic Gap

Software or Hardware Translator

ARM v8.4

HLL

Implementation ISA with
Simple Inst
& Data Types
& Addressing Modes

HW

Control
Signals
An Example: Rosetta 2 Binary Translator

Rosetta 2  [ edit ]

In 2020, Apple announced Rosetta 2 would be bundled with macOS Big Sur, to aid in the Mac transition to Apple silicon. The software permits many applications compiled exclusively for execution on x86-64-based processors to be translated for execution on Apple silicon.[2][8]

In addition to the just-in-time (JIT) translation support, Rosetta 2 offers ahead-of-time compilation (AOT), with the x86-64 code fully translated, just once, when an application without a universal binary is installed on an Apple silicon Mac.[9]

Rosetta 2's performance has been praised greatly.[10][11] In some benchmarks, x86-64-only programs performed better under Rosetta 2 on a Mac with an Apple M1 SOC than natively on a Mac with an Intel x86-64 processor. One of the key reasons why Rosetta 2 provides such high level of translation efficiency is the support of x86-64 memory ordering in Apple M1 SOC.[12]

Although Rosetta 2 works for most software, some software doesn't work at all[13] or is reported to be "sluggish".[14] A lot of software can be made compatible with the new Macs by the vendor recompiling the software, often a simple task; while for some software (such as software that includes assembly language code, or that generates machine code), the changes to make them work aren't simple and cannot be automated.

Similar to the first version, Rosetta 2 does not normally require user intervention. When a user attempts to launch an x86-64-only application for the first time, macOS prompts them to install Rosetta 2 if it is not already available. Subsequent launches of x86-64 programs will execute via translation automatically. An option also exists to force a universal binary to run as x86-64 code through Rosetta 2, even on an ARM-based machine.[15]
An Example: Rosetta 2 Binary Translator

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested
Another Example: Intel and AMD Processors

X86-64

ISA with
Complex Inst
& Data Types
& Addressing Modes

HLL

Small Semantic Gap

Hardware Translator

Implementation ISA with
Simple Inst
& Data Types
& Addressing Modes

Secret
Micro-operations

HW
Control
Signals
Another Example: Intel and AMD Processors

Source: https://twitter.com/Locuza_/status/1454152714930331652
Another Example: Intel and AMD Processors

Core Count:
8 cores/16 threads

L1 Caches:
32 KB per core

L2 Caches:
512 KB per core

L3 Cache:
32 MB shared

AMD Ryzen 5000, 2020

Another Example: NVIDIA Denver

The Secret of Denver: Binary Translation & Code Optimization

As we alluded to earlier, NVIDIA's decision to forgo a traditional out-of-order design for Denver means that much of Denver's potential is contained in its software rather than its hardware. The underlying chip itself, though by no means simple, is at its core a very large in-order processor. So it falls to the software stack to make Denver sing.

Accomplishing this task is NVIDIA's dynamic code optimizer (DCO). The purpose of the DCO is to accomplish two tasks: to translate ARM code to Denver's native format, and to optimize this code to make it run better on Denver. With no out-of-order hardware on Denver, it is the DCO's task to find instruction level parallelism within a thread to fill Denver's many execution units, and to reorder instructions around potential stalls, something that is no simple task.

https://www.anandtech.com/show/8701/the-google-nexus-9-review/4
https://www.toradex.com/computer-on-modules/apalis-arm-family/nvidia-tegra-k1
Transmeta: x86 to VLIW Translation

Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.


https://www.wikiwand.com/en/Transmeta_Efficeon
**ISA-level Tradeoffs: Number of Registers**

- **Affects:**
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - (uarch) Size, access time, power consumption of register file

- **Large number of registers:**
  - + Enables better register allocation (and optimizations) by compiler $\rightarrow$ fewer saves/restores
  - -- Larger instruction size
  - -- Larger register file size
There Is A Lot More to Cover on ISAs

https://www.youtube.com/onurmutlulectures
There Is A Lot More to Cover on ISAs


28,806 views • Jan 23, 2015

Carnegie Mellon Computer Architecture
22.8k subscribers

Lecture 4. ISA Tradeoffs (cont.) & MIPS ISA
Lecturer: Kevin Chang (http://users.ece.cmu.edu/~kevincha/)
Date: Jan 21th, 2015

https://www.youtube.com/onurmutlulectures
Detailed Lectures on ISAs & ISA Tradeoffs

- Computer Architecture, Spring 2015, Lecture 3
  - ISA Tradeoffs (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=QKdiZSfwg&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=3

- Computer Architecture, Spring 2015, Lecture 4
  - ISA Tradeoffs & MIPS ISA (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=RBgeCCW5Hjs&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=4

- Computer Architecture, Spring 2015, Lecture 2
  - Fundamental Concepts and ISA (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=NpC39uS4K4o&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=2

https://www.youtube.com/onurmutlulectures
ISA Design and Tradeoffs: More Critical Thinking
The Von Neumann Model/Architecture

Stored program

Sequential instruction processing
The von Neumann Model/Architecture

- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:
  
  - **Stored program**
    - Instructions stored in a linear memory array
    - Memory is unified between instructions and data
      - The interpretation of a stored value depends on the control signals
        When is a value interpreted as an instruction?
  
  - **Sequential instruction processing**
Recall: The Instruction Cycle

- **FETCH**: Interpret memory value as Instruction
- **DECODE**
- **EVALUATE ADDRESS**
- **FETCH OPERANDS**: Interpret memory value as Data
- **EXECUTE**
- **STORE RESULT**

Whether a value fetched from memory is interpreted as an instruction depends on **when** that value is **fetched** in the instruction processing cycle.
The von Neumann Model/Architecture

- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:

- **Stored program**
  - Instructions stored in a linear memory array
  - *Memory is unified* between instructions and data
    - The interpretation of a stored value depends on the control signals

- **Sequential instruction processing**
  - One instruction processed (fetched, executed, completed) at a time
  - *Program counter* (instruction pointer) identifies the current instruction
  - *Program counter is advanced sequentially* except for control transfer instructions

When is a value interpreted as an instruction?
The von Neumann Model/Architecture

- **Recommended reading**
  - Burks, Goldstein, von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument,” 1946.

- **Important reading**
  - Patt and Patel book, Chapter 4, “The von Neumann Model”

- **Stored program**

- **Sequential instruction processing**
The Von Neumann Model (of a Computer)

INPUT
- Keyboard,
- Mouse,
- Disk...

OUTPUT
- Monitor,
- Printer,
- Disk...

MEMORY
- Mem Addr Reg
- Mem Data Reg

PROCESSING UNIT
- ALU
- TEMP

CONTROL UNIT
- PC or IP
- Inst Register
The Von Neumann Model (of a Computer)

Q: Is this the only way that a computer can process computer programs?

A: No.

Qualified Answer: No. But, it has been the dominant way
- i.e., the dominant paradigm for computing
- for N decades

Let's examine a completely different model for processing computer programs
The Dataflow Execution Model of a Computer
The Dataflow Model (of a Computer)

- **Von Neumann model**: An instruction is fetched and executed in **control flow order**
  - As specified by the program counter (instruction pointer)
  - Sequential unless explicit control flow instruction

- **Dataflow model**: An instruction is fetched and executed in **data flow order**
  - i.e., when its operands are ready
  - i.e., there is **no program counter (instruction pointer)**
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies “who” should receive the result
    - An instruction can “fire” whenever all operands are received
  - Potentially many instructions can execute at the same time
    - Inherently more parallel
Consider a Von Neumann program

- What is the significance of the program order?
- What is the significance of the storage locations?

\[
\begin{align*}
v &= a + b; \\
w &= b \times 2; \\
x &= v - w \\
y &= v + w \\
z &= x \times y
\end{align*}
\]

Sequential

Dataflow

Which model is more natural to you as a programmer?
More on Dataflow

- In a dataflow machine, a program consists of dataflow nodes
  - A dataflow node fires (fetched and executed) when all its inputs are ready
    - i.e. when all inputs have tokens

- Dataflow node and its ISA representation
Example Dataflow Nodes

*Conditional

*Relational

*Barrier Synch
A Simple Example Dataflow Program

N is a non-negative integer

What is the value of OUT?
ISA-level Tradeoff: Program Counter

- Do we want a Program Counter (PC or IP) in the ISA?
  - Yes: Control-driven, sequential execution
    - An instruction is executed when the PC points to it
    - PC automatically changes sequentially (except for control flow instructions) → sequential
  - No: Data-driven, parallel execution
    - An instruction is executed when all its operand values are available → dataflow

- Tradeoffs: MANY high-level ones
  - Ease of programming (for average programmers)?
  - Ease of compilation?
  - Performance: Extraction of parallelism?
  - Hardware complexity?
ISA vs. Microarchitecture Level Tradeoff

- A similar tradeoff (control vs. data-driven execution) can be made at the microarchitecture level

- **ISA:** Specifies how the **programmer sees** the instructions to be executed
  - Programmer sees a sequential, control-flow execution order vs.
  - Programmer sees a dataflow execution order

- **Microarchitecture:** How the **underlying implementation actually executes** instructions
  - Microarchitecture can execute instructions in any order as long as it obeys the semantics specified by the ISA when making the instruction results visible to software
  - Programmer should see the order specified by the ISA
Let’s Get Back to the von Neumann Model

- But, if you want to learn more about dataflow...


- A later lecture

- If you are really impatient:
  - http://www.youtube.com/watch?v=D2uue7izU2c
Lecture Video on Dataflow Architectures

http://www.youtube.com/watch?v=D2uue7izU2c
The von Neumann Model

- All major *instruction set architectures* today use this model
  - x86, ARM, MIPS, SPARC, Alpha, POWER, RISC-V, ...

- Underneath (at the microarchitecture level), the execution model of almost all *implementations (or, microarchitectures)* is very different
  - Pipelined instruction execution: *Intel 80486 uarch*
  - Multiple instructions at a time: *Intel Pentium uarch*
  - Out-of-order execution: *Intel Pentium Pro uarch*
  - Separate instruction and data caches

- But, what happens underneath that is *not consistent* with the von Neumann model is *not exposed* to software
  - Difference between ISA and microarchitecture
What is Computer Architecture?

- **ISA+implementation definition:** The science and art of designing, selecting, and interconnecting hardware components and designing the hardware/software interface to create a computing system that meets functional, performance, energy consumption, cost, and other specific goals.

- **Traditional (ISA-only) definition:** “The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior *as distinct from* the organization of the dataflow and controls, the logic design, and the physical implementation.”

  *Gene Amdahl*, IBM Journal of R&D, April 1964
ISA vs. Microarchitecture

- **ISA**
  - Agreed upon interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write and debug system/user programs

- **Microarchitecture**
  - Specific implementation of an ISA
  - Not visible to the software

- **Microprocessor**
  - **ISA, uarch, circuits**
  - “Architecture” = ISA + microarchitecture
Microarchitecture

- A specific **implementation** of the ISA

- How do we implement the ISA?
  - We will discuss this for many lectures

- There can be many implementations of the same ISA
  - **MIPS** R2000, R3000, R4000, R6000, R8000, R10000, ...
  - **x86**: Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, Ice Lake, Golden Cove, Sapphire Rapids, ..., AMD K5, K7, K9, Bulldozer, BobCat, Ryzen X, ...
  - **POWER** 4, 5, 6, 7, 8, 9, 10 (IBM), ..., **PowerPC** 604, 605, 620, ...
  - **ARM** Cortex-M*, ARM Cortex-A*, NVIDIA Denver, Apple A*, M1, ...
  - **Alpha** 21064, 21164, 21264, 21364, ...
  - **RISC-V** ...
  - ...

208
ISA vs. Microarchitecture

What is part of ISA vs. Uarch?
- Gas pedal: interface for “acceleration”
- Internals of the engine: implement “acceleration”

Implementation (uarch) can be various as long as it satisfies the specification (ISA)
- Add instruction vs. Adder implementation
  - Bit serial, ripple carry, carry lookahead adders are all part of microarchitecture (see H&H Chapter 5.2.1)
- x86 ISA has many implementations:
  - Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, Ice Lake, Golden Cover, Sapphire Rapids, ..., AMD K5, K7, K9, Bulldozer, BobCat, Ryzen X, ...

Microarchitecture usually changes faster than ISA
- Few ISAs (x86, ARM, SPARC, MIPS, Alpha, RISC-V) but many uarchs
- Why?
ISA: What Does It Specify?

- **Instructions**
  - Opcodes, Addressing Modes, Data Types
  - Instruction Types and Formats
  - Registers, Condition Codes

- **Memory**
  - Address space, Addressability, Alignment
  - Virtual memory management

- **Call, Interrupt/Exception Handling**

- **Access Control, Priority/Privilege**

- **I/O: memory-mapped vs. instructions**

- **Task/thread Management**

- **Power & Thermal Management**

- **Multithreading & Multiprocessor support**

...
# ISA Manuals: Some Good Bedtime Reading

## Combined Volume Set of Intel® 64 and IA-32 Architectures Software Developer’s Manuals

<table>
<thead>
<tr>
<th>Document</th>
<th>Description</th>
</tr>
</thead>
</table>
| Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4 | This document contains the following:  
**Volume 1**: Describes the architecture and programming environment of processors supporting IA-32 and Intel® 64 architectures.  
**Volume 2**: Includes the full instruction set reference, A-Z. Describes the format of the instruction and provides reference pages for instructions.  
**Volume 3**: Includes the full system programming guide, parts 1, 2, 3, and 4. Describes the operating-system support environment of Intel® 64 and IA-32 architectures, including: memory management, protection, task management, interrupt and exception handling, multi-processor support, thermal and power management features, debugging, performance monitoring, system management mode, virtual machine extensions (VMX) instructions, Intel® Virtualization Technology (Intel® VT), and Intel® Software Guard Extensions (Intel® SGX). NOTE: Performance monitoring events can be found here: [https://perfmon-events.intel.com/](https://perfmon-events.intel.com/)  
**Volume 4**: Describes the model-specific registers of processors supporting IA-32 and Intel® 64 architectures. |
| Intel® 64 and IA-32 Architectures Software Developer's Manual Documentation Changes | Describes bug fixes made to the Intel® 64 and IA-32 architectures software developer's manual between versions.  
NOTE: This change document applies to all Intel® 64 and IA-32 architectures software developer's manual sets (combined volume set, 4 volume set, and 10 volume set). |


211
ISA Manuals: Some Good Bedtime Reading

The RISC-V instruction set architecture (ISA) and related specifications are developed, ratified and maintained by RISC-V International contributing members within the RISC-V International Technical Working Groups. Work on the specification is performed on GitHub, and the GitHub issue mechanism can be used to provide input into the specification.

If you would like more information on becoming a member, please see the membership page.

ISA Specification
The specifications shown below represent the current, ratified releases. Work is being done on GitHub.

- Volume 1, Unprivileged Spec v. 20191213 [PDF]
- Volume 2, Privileged Spec v. 20211203 [PDF]
- Recently ratified, but not yet integrated, extension specifications

Debug Specification
This is the currently ratified specification:

- External Debug Support v. 0.13.2 [PDF] [GitHub]

This is the current stable draft:

- External Debug Support v. 1.0.0-STABLE [PDF]

Trace Specification
The processor trace specification was approved on March 20, 2020.

- Trace Specification v. 1.0 [PDF] [GitHub]

Compatibility Test Framework
The RISC-V Architectural Compatibility Test Framework Version 2 is now available. This framework compares arbitrary models against a reference signature, and currently covers RV[32|64]IMC unprivileged specifications only. Tests for the not-yet-ratified Crypto Scalar extension and RV32EMC extensions are also available.

Work on Version 3.0 framework (RISCOF) is

https://riscv.org/technical/specifications/