

Digital Design & Computer Arch.

Lecture 9: Von Neumann Model & Instruction Set Architectures

Prof. Onur Mutlu

ETH Zürich

Spring 2022

24 March 2022

Assignment: Lecture Video (April 1)

- Why study computer architecture? Why is it important?
- Future Computing Platforms: Challenges & Opportunities
- **Required Assignment**
 - ❑ **Watch one of** Prof. Mutlu's lectures and analyze either (or both)
 - ❑ <https://www.youtube.com/watch?v=kgiZISOcGFM> (May 2017)
 - ❑ <https://www.youtube.com/watch?v=mskTeNnf-i0> (Feb 2021)
- **Optional Assignment – for 1% extra credit**
 - ❑ **Write a 1-page summary** of one of the lectures and email us
 - What are your key takeaways?
 - What did you learn?
 - What did you like or dislike?
 - Submit your summary to [Moodle](#) by April 1

Extra Assignment: Moore's Law (I)

- **Paper review**
- G.E. Moore. "Cramming more components onto integrated circuits," Electronics magazine, 1965

- **Optional Assignment – for 1% extra credit**
 - **Write a 1-page review**
 - Upload PDF file to Moodle – Deadline: April 7

- I strongly recommend that you **follow my guidelines for (paper) review** (see next slide)

Extra Assignment 2: Moore's Law (II)

■ 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 2022 Lectures/Schedule

Week	Date	Livestream	Lecture	Readings
W1	24.02 Thu.	Live	L1: Introduction and Basics (PDF) (PPT)	Suggested Mentioned
	25.02 Fri.	Live	L2a: Tradeoffs, Metrics, Mindset (PDF) (PPT)	Required Suggested Mentioned
			L2b: Mysteries in Computer Architecture (PDF) (PPT)	Required Suggested Mentioned
W2	03.03 Thu.	Live	L3a: Mysteries in Computer Architecture (PDF) (PPT)	Required Suggested Mentioned
			L3b: Introduction to the Labs and FPGAs (PDF) (PPT)	Required Suggested Mentioned
	04.03 Fri.	Live	L4: Combinational Logic I (PDF) (PPT)	Video- Required Required Suggested Mentioned
W3	10.03 Thu.	Live	L5: Combinational Logic II (PDF) (PPT)	Required Suggested Mentioned
	11.03 Fri.	Live	L6: Sequential Logic Design (PDF) (PPT)	Required Suggested Mentioned
W4	17.03 Thu.	Live	L7: Hardware Description Languages and Verilog (PDF) (PPT)	Required Suggested Mentioned
	18.03 Fri.	Premiere	L8: Timing and Verification (PDF) (PPT)	Required Suggested Mentioned

Problem

Algorithm

Program/Language

System Software

SW/HW Interface

Micro-architecture

Logic

Devices

Electrons

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

Problem
Algorithm
Program/Language
System Software
SW/HW Interface
Micro-architecture
Logic
Devices
Electrons

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

Problem
Algorithm
Program/Language
System Software
SW/HW Interface
Micro-architecture
Logic
Devices
Electrons

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

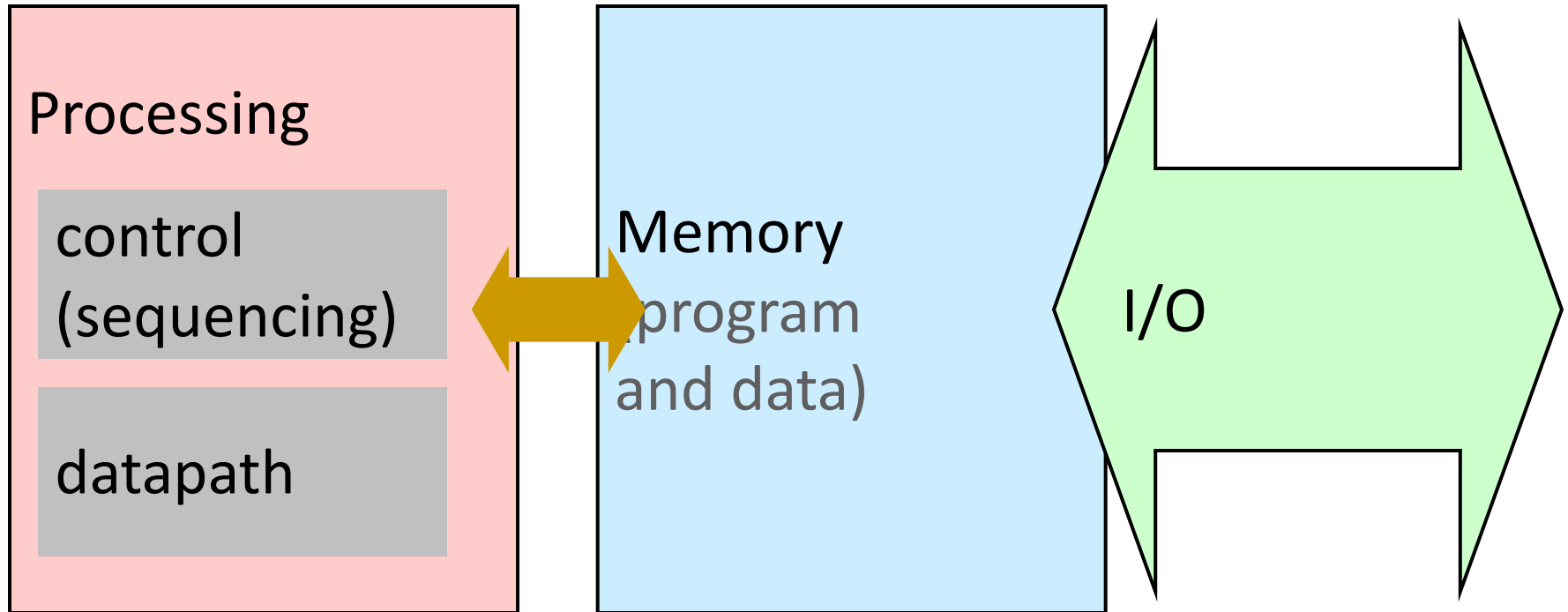
- Introduction to microarchitecture and single-cycle microarchitecture
 - H&H, Chapter 7.1-7.3
 - P&P, Appendices A and C
- Multi-cycle microarchitecture
 - H&H, Chapter 7.4
 - P&P, Appendices A and C

Building a Computing System

The von Neumann Model

Recall: What is A Computer?

- We will cover all three components



Building Up to A Basic Computer Model

- 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

Problem
Algorithm
Program/Language
System Software
SW/HW Interface
Micro architecture
Logic
Devices
Electrons

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

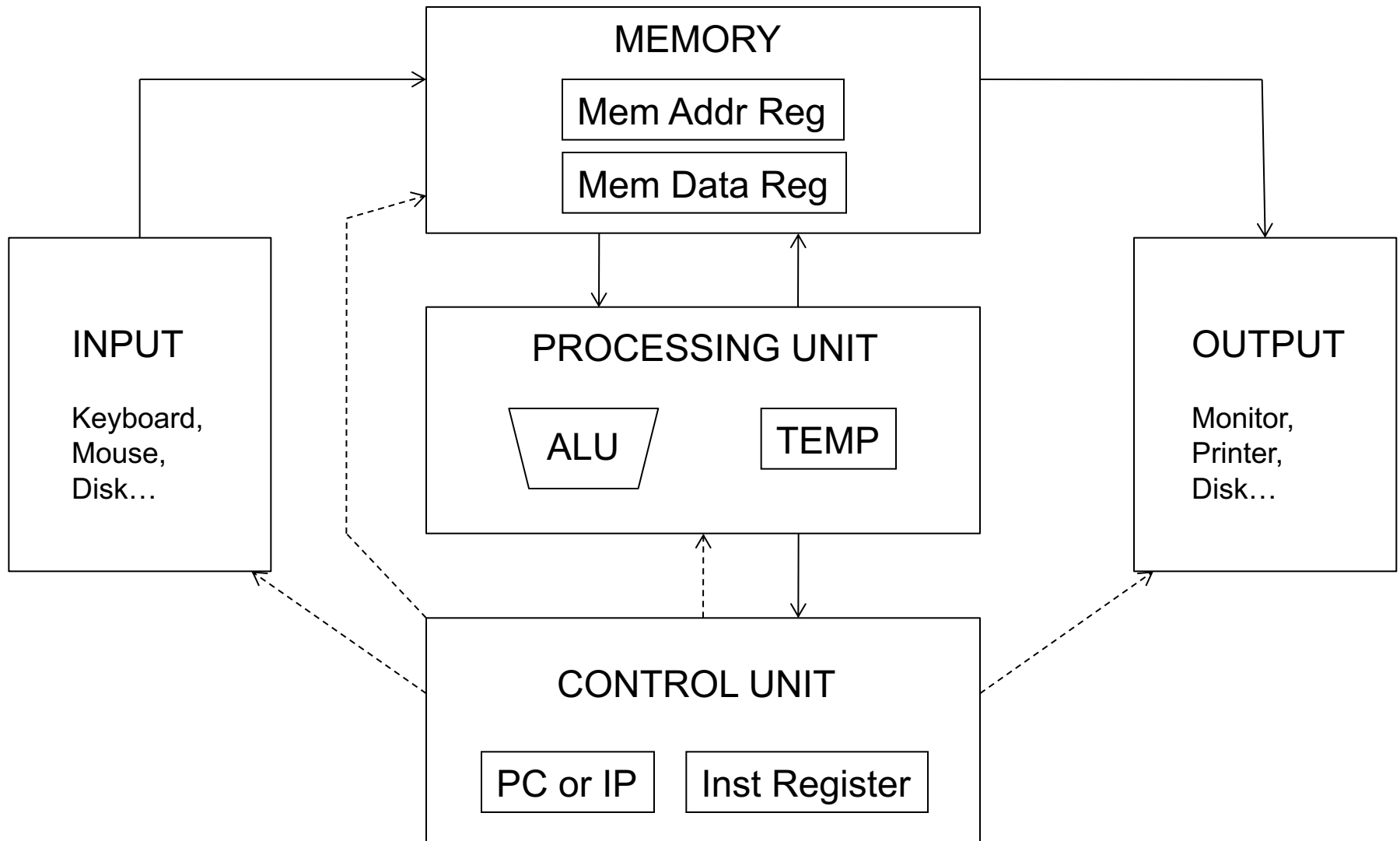
- In order to build a computer, we need an execution model for processing computer programs
- John von Neumann proposed a fundamental model in 1946
- The von Neumann Model consists of 5 components
 - ❑ Memory (stores the program and data)
 - ❑ Processing unit
 - ❑ Input
 - ❑ Output
 - ❑ Control unit (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



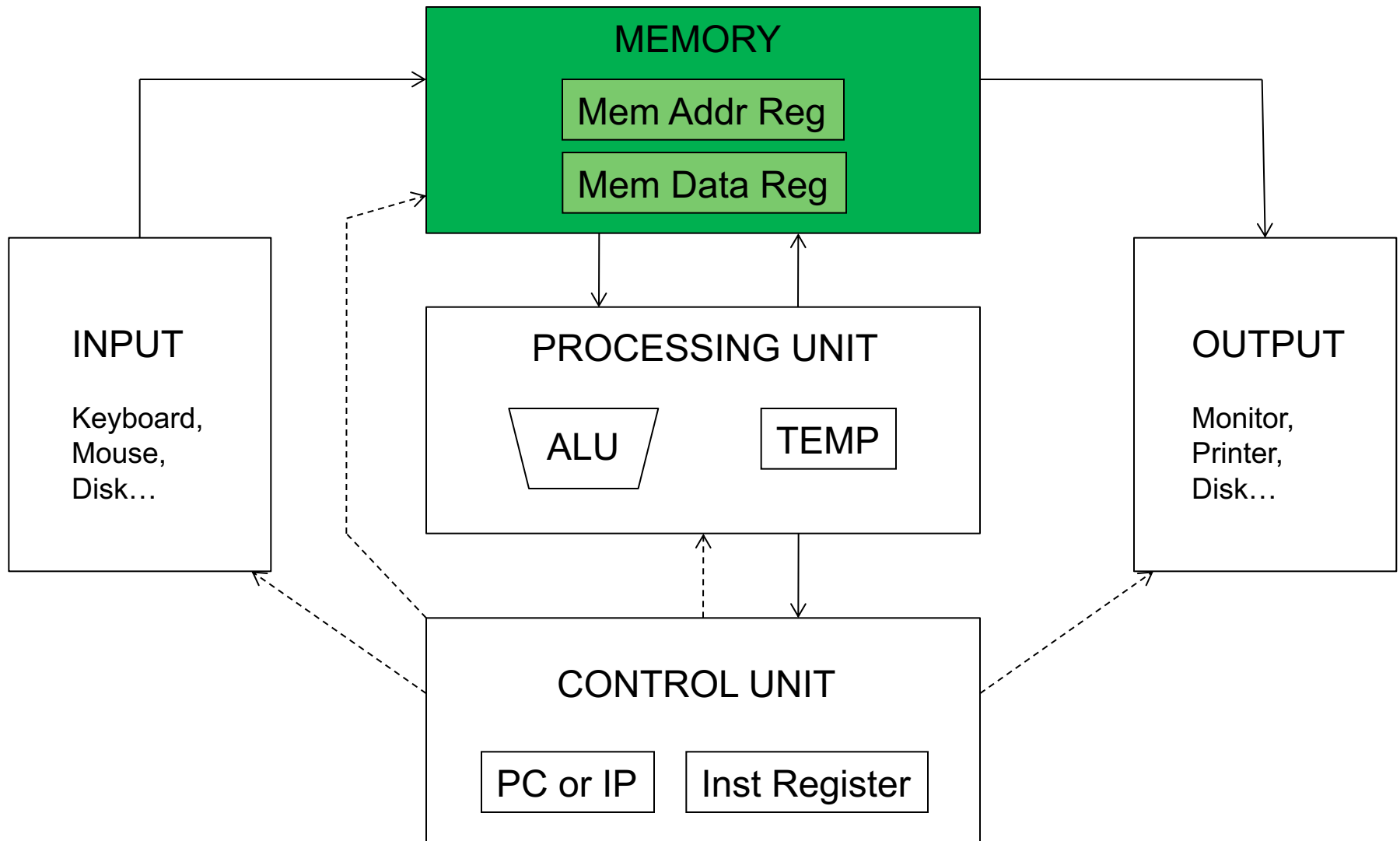
Burks, Goldstein, von Neumann,
“Preliminary discussion of the logical design
of an electronic computing instrument,” 1946.

All general-purpose computers today use the von Neumann model

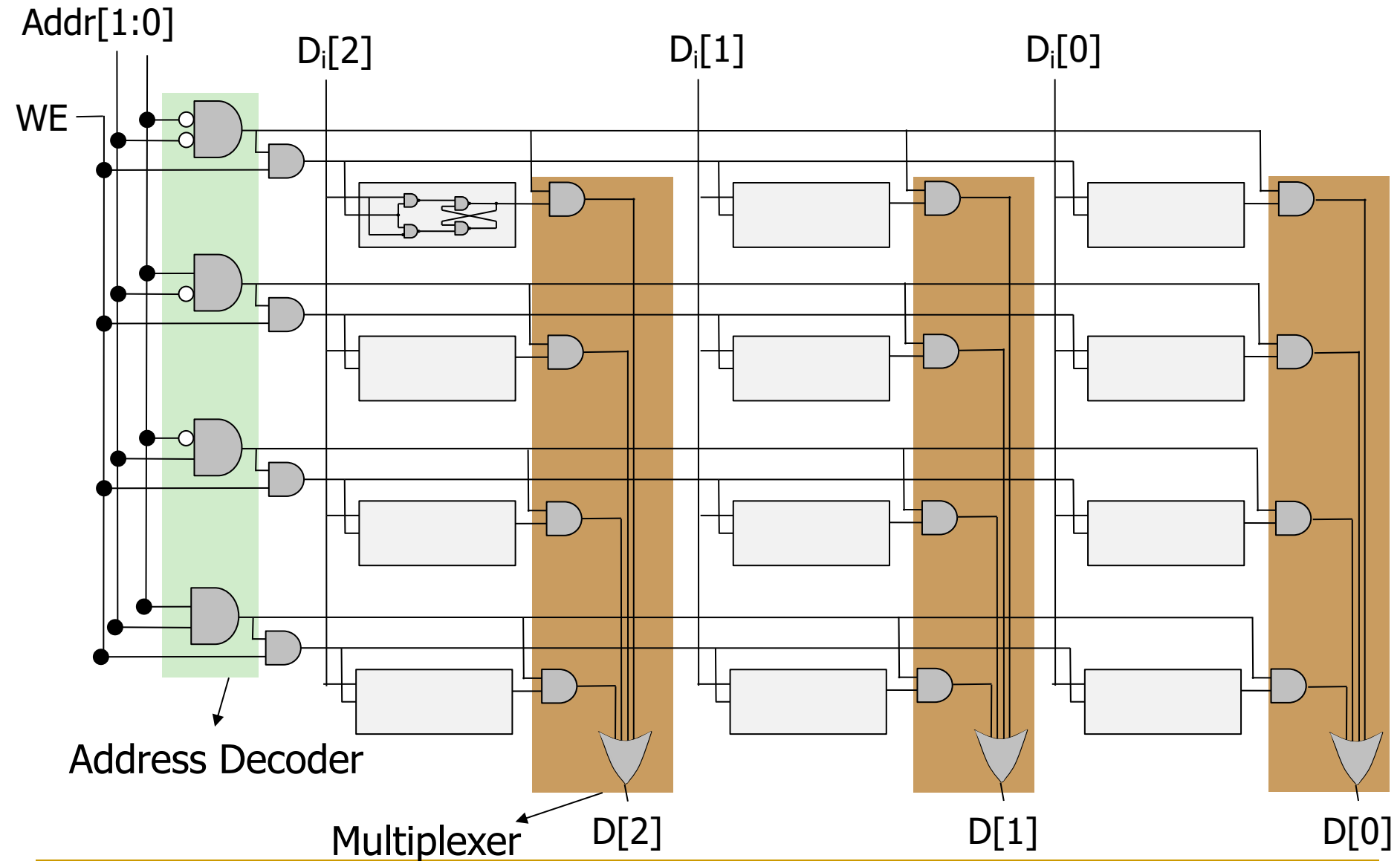
The von Neumann Model



The von Neumann Model



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

Address	Data Value
000	
001	
010	
011	
100	00000110
101	
110	00000100
111	

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**

Word Address	Data	MIPS memory
·	·	·
·	·	·
·	·	·
00000003	D 1 6 1 7 A 1 C	Word 3
00000002	1 3 C 8 1 7 5 5	Word 2
00000001	F 2 F 1 F 0 F 7	Word 1
00000000	8 9 A B C D E F	Word 0

Byte-Addressable Memory

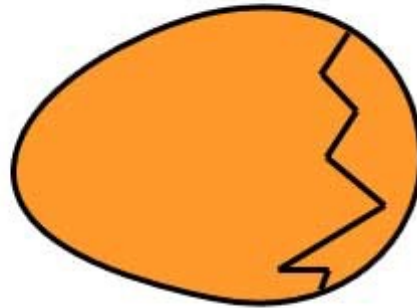
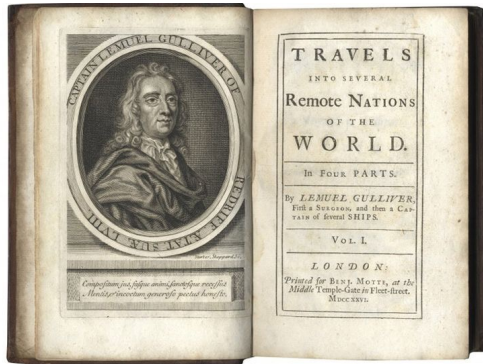
- Each **byte** has a **unique address**
 - MIPS is actually **byte-addressable**
 - LC-3b (updated version of LC-3) is also **byte-addressable**

Byte Address of the Word		Data				MIPS memory	
.		.				.	
.		.				.	
0000000C		D 1	6 1	7 A	1 C	Word 3	
00000008		1 3	C 8	1 7	5 5	Word 2	
00000004		F 2	F 1	F 0	F 7	Word 1	
00000000		How are these four bytes ordered?				Word 0	

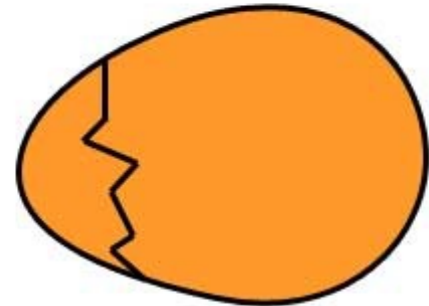
Which of the four bytes is most vs. least significant?

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



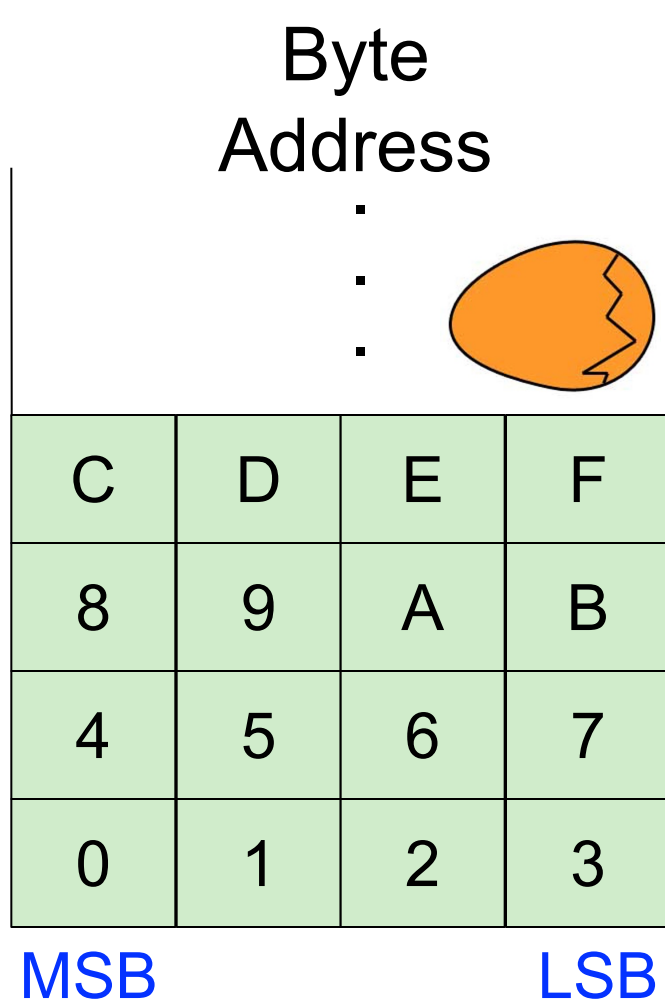
BIG ENDIAN - The way people always broke their eggs in the Lilliput land



LITTLE ENDIAN - The way the king then ordered the people to break their eggs

Big Endian vs. Little Endian

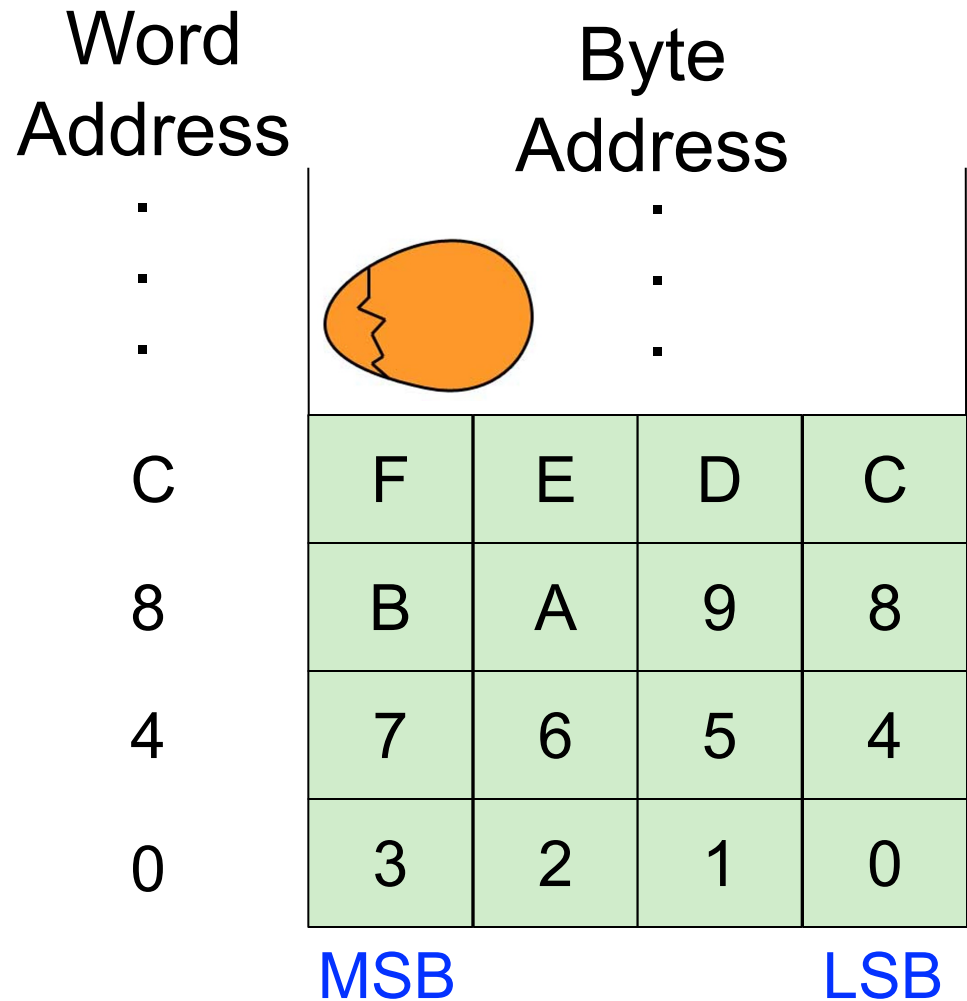
Big Endian



(Most Significant Byte) (Least Significant Byte)

LSB in higher byte address

Little Endian



LSB in lower byte address

Big Endian vs. Little Endian

Big Endian

Little Endian

Does this really matter?

Answer: **No**, it is a convention

Qualified answer: **No**, except when one **big-endian system** and **one little-endian system** have to **share or exchange data**

MSB

LSB

MSB

LSB

(Most Significant Byte)

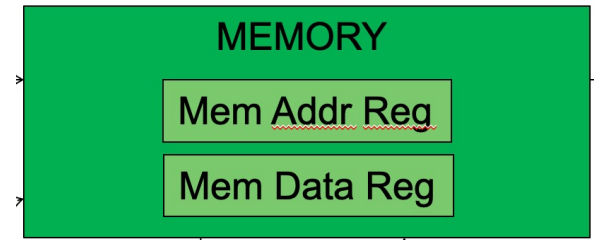
(Least Significant Byte)

LSB in higher byte address

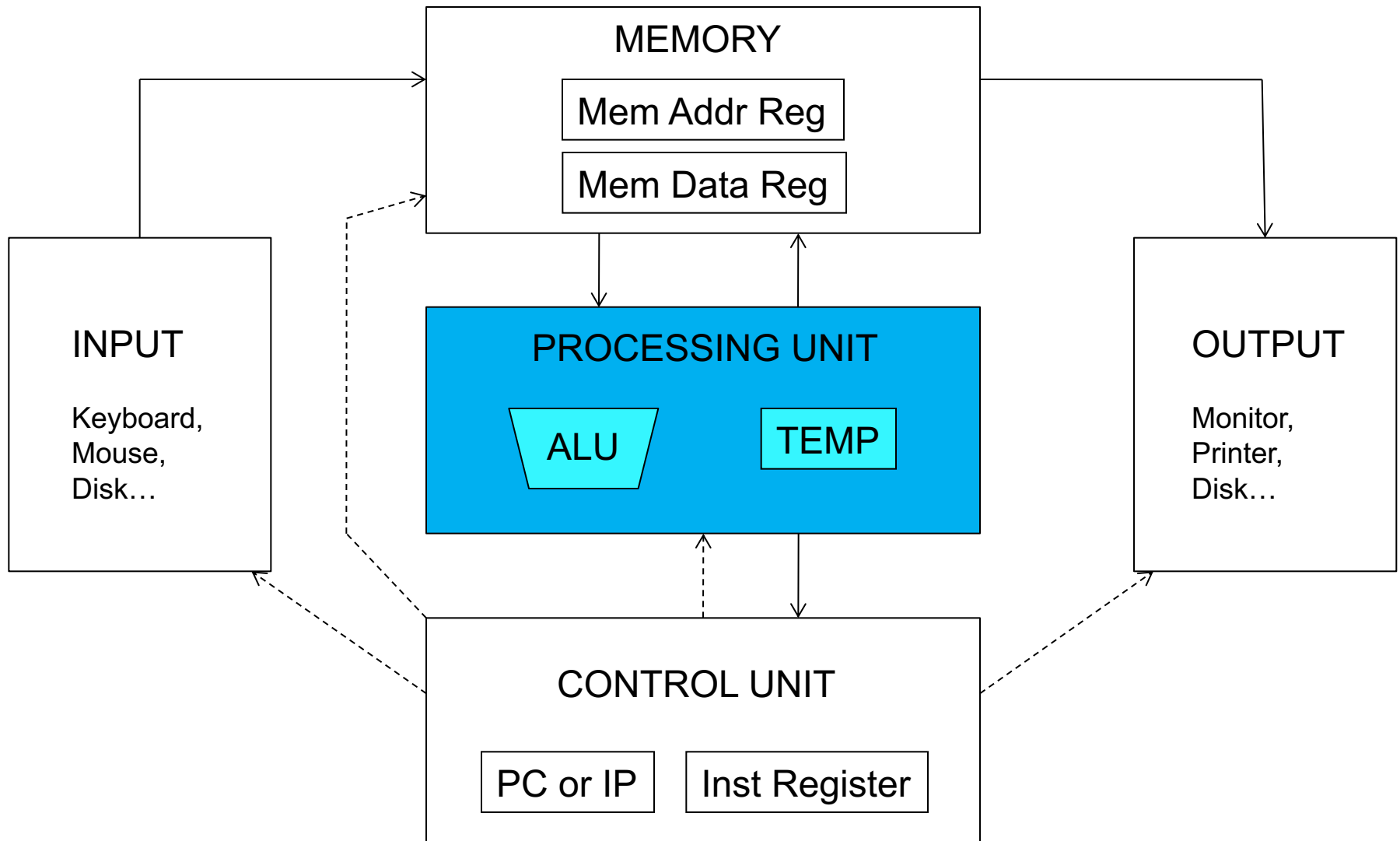
LSB in lower byte address

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



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:

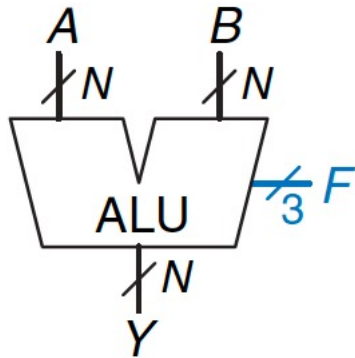


Figure 5.14 ALU symbol

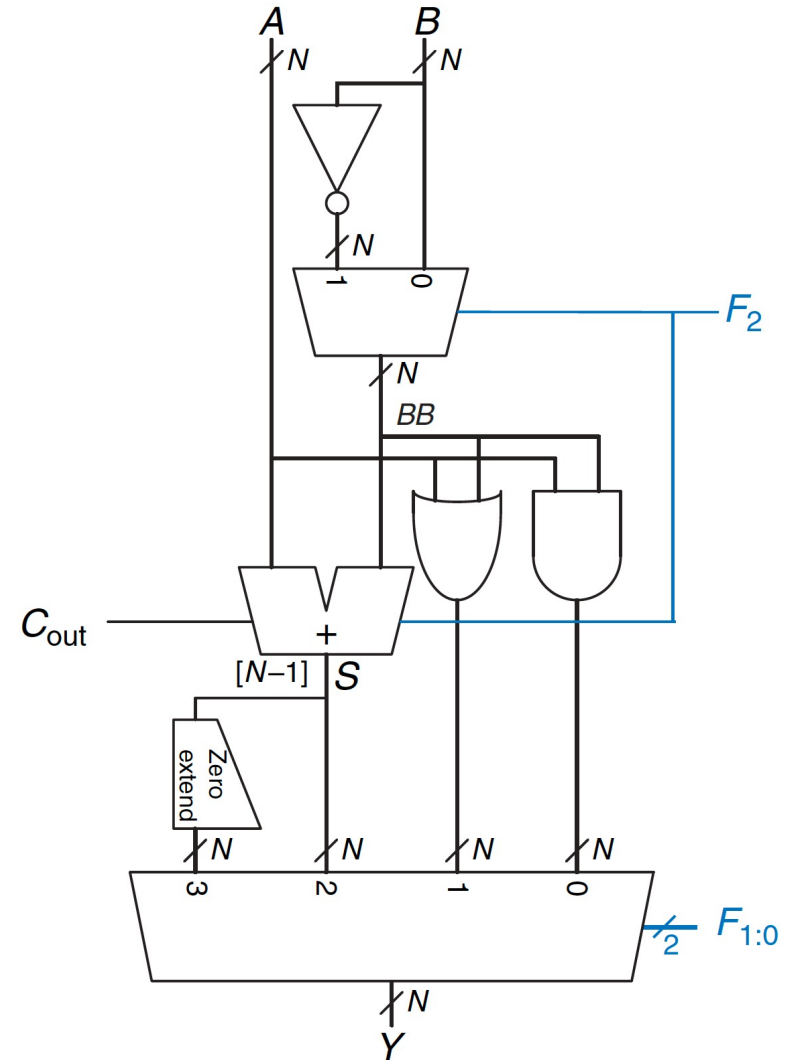
Table 5.1 ALU operations

$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND \bar{B}
101	A OR \bar{B}
110	A - B
111	SLT

Recall: Example ALU (Arithmetic Logic Unit)

Table 5.1 ALU operations

$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND \overline{B}
101	A OR \overline{B}
110	A – B
111	SLT

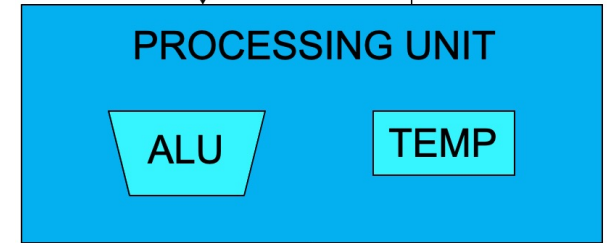


Processing Unit: Fast Temporary Storage

- 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 $((A+B)*C)/D$, 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 $((A+B)*C)$
- 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)**

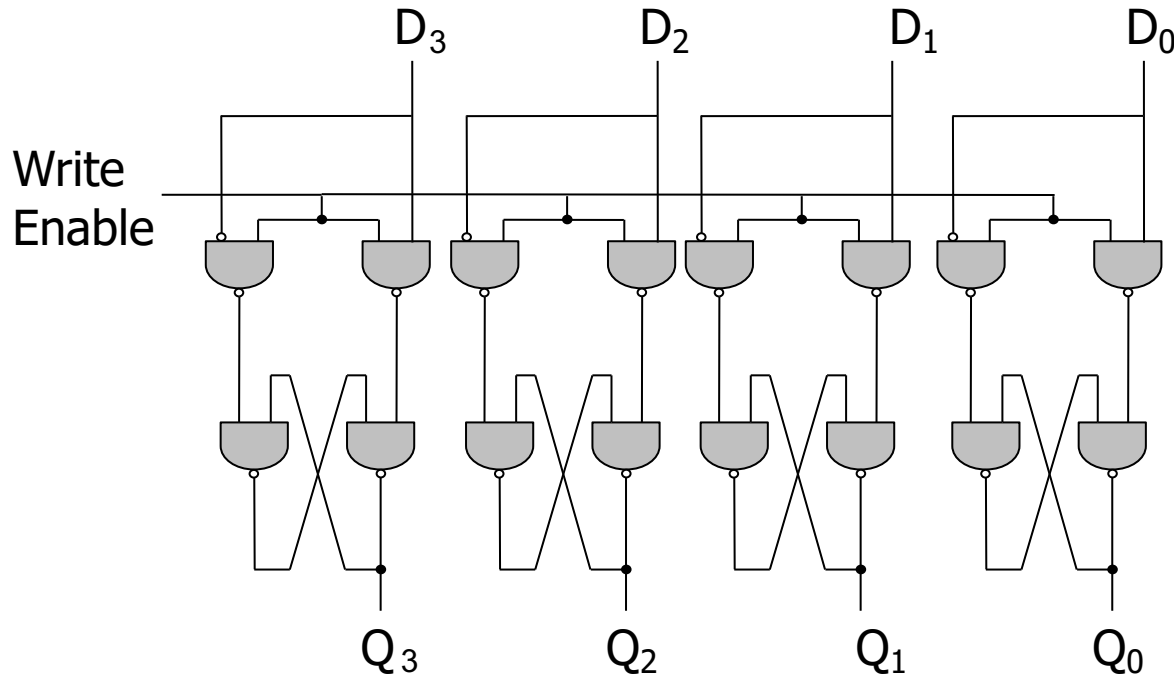
- **Register Set or Register File**

- **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



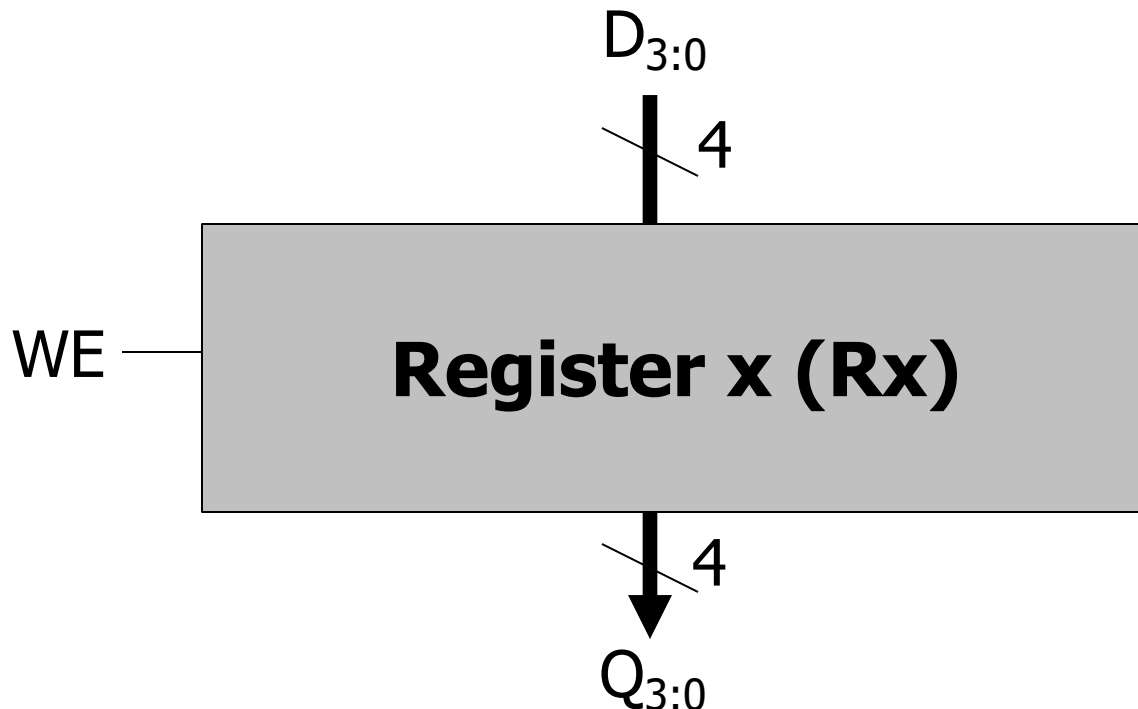
Here we have a **register**, or a structure that stores more than one bit and can be read from and written to

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

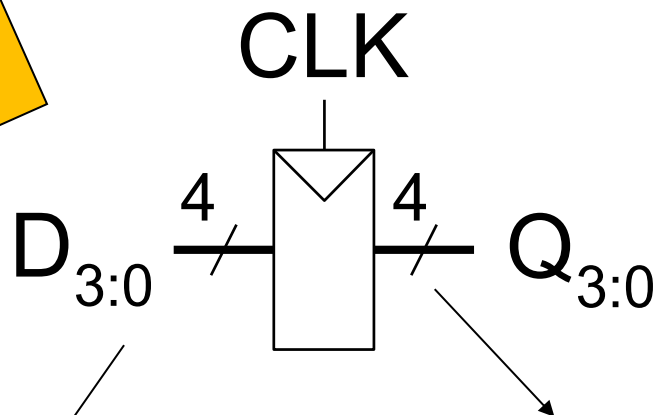
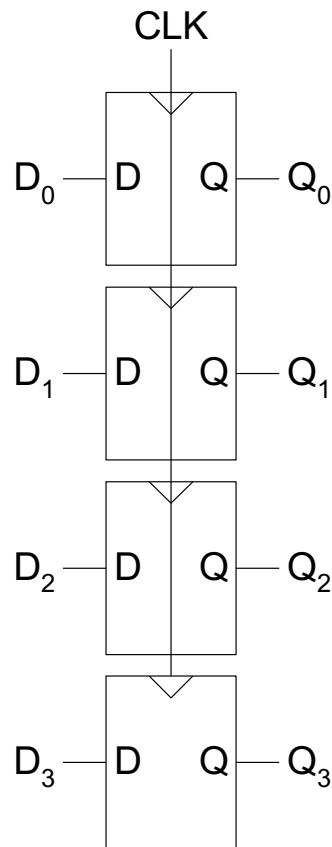


Here we have a **register**, or a structure that stores more than one bit and can be read from and written to

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

Recall: D Flip-Flop Based Register

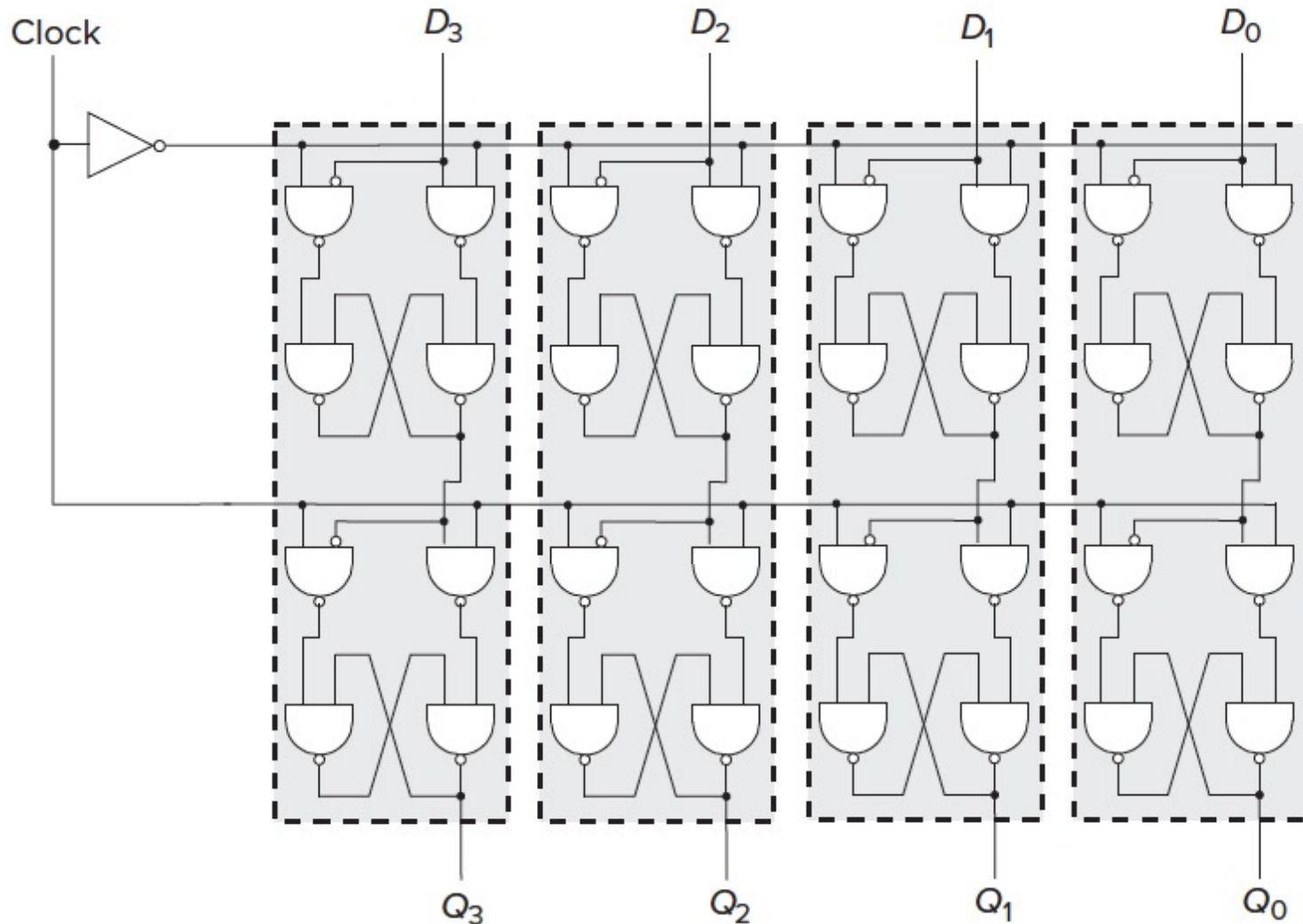
- Multiple parallel D flip-flops, each of which storing 1 bit



This line represents 4 wires

This register stores 4 bits

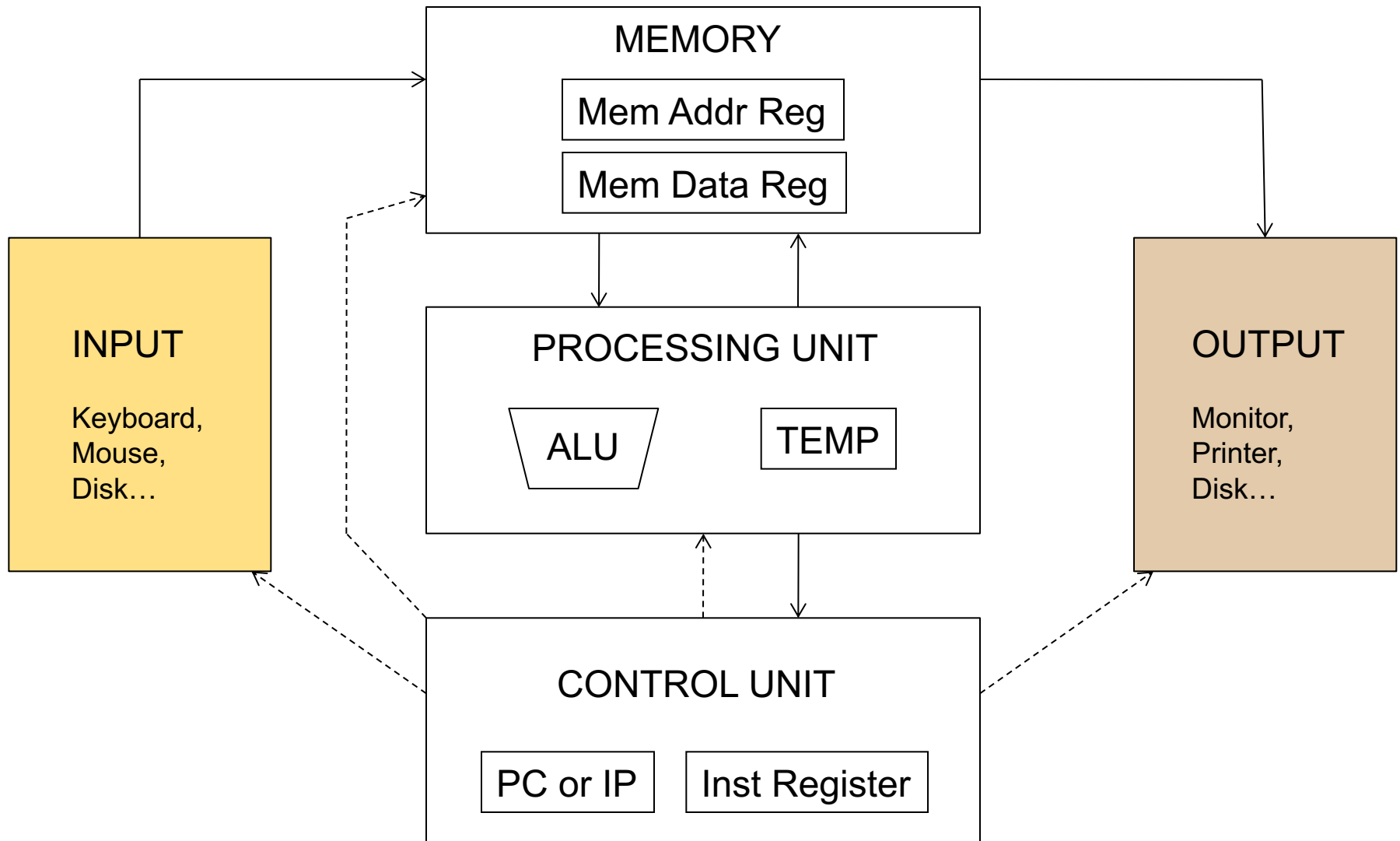
Recall: A 4-Bit D-Flip-Flop-Based Register (Internally)



MIPS Register File

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

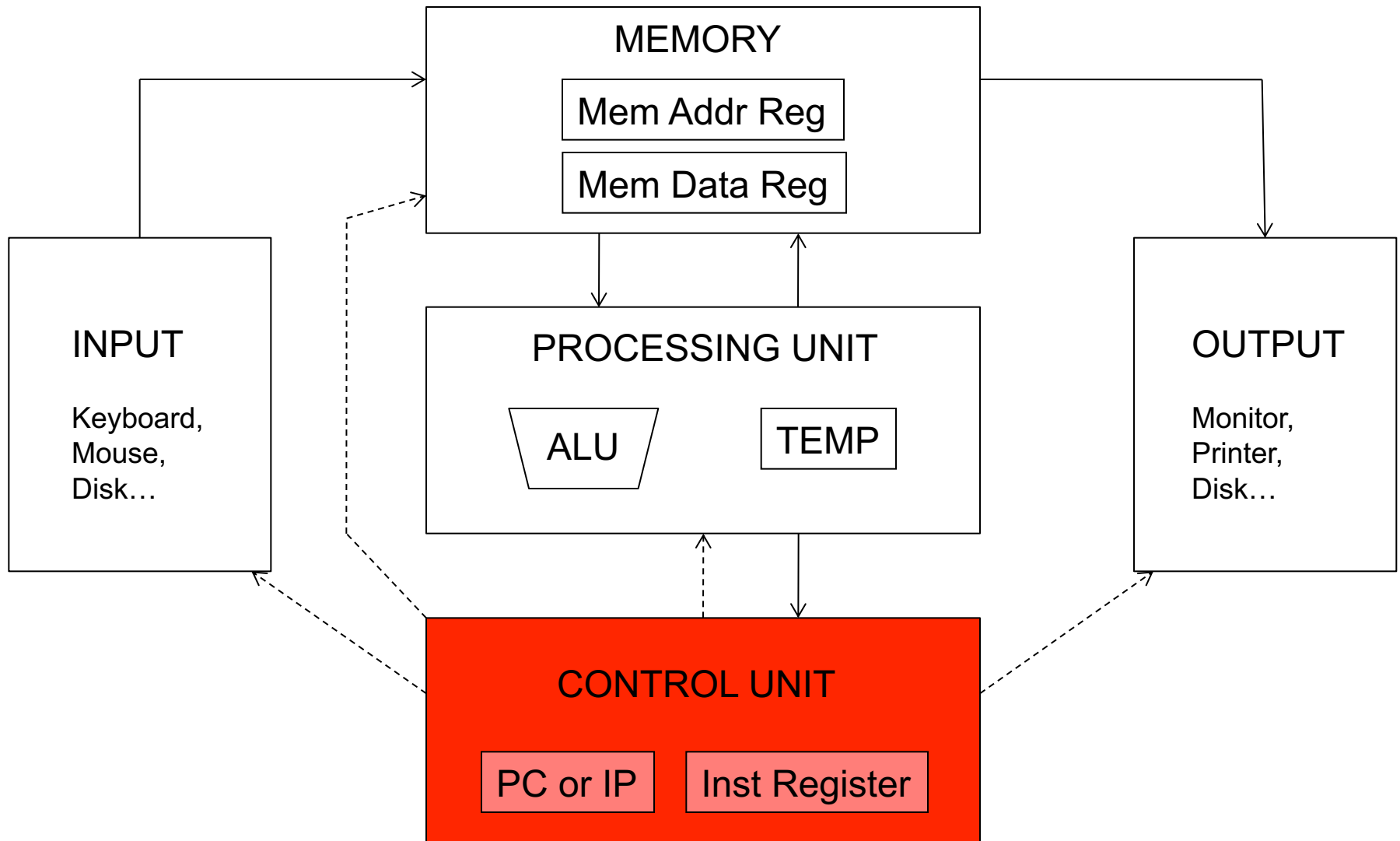
The Von Neumann Model



Input and Output

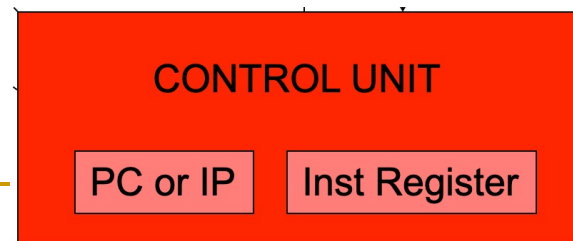
- Enable information to get into and out of a computer
- Many devices can be used for input and output
- They are called **peripherals**
 - **Input**
 - Keyboard
 - Mouse
 - Scanner
 - Disks
 - Etc.
 - **Output**
 - Monitor
 - Printer
 - Disks
 - Etc.
 - In LC-3, we consider keyboard and monitor

The Von Neumann Model

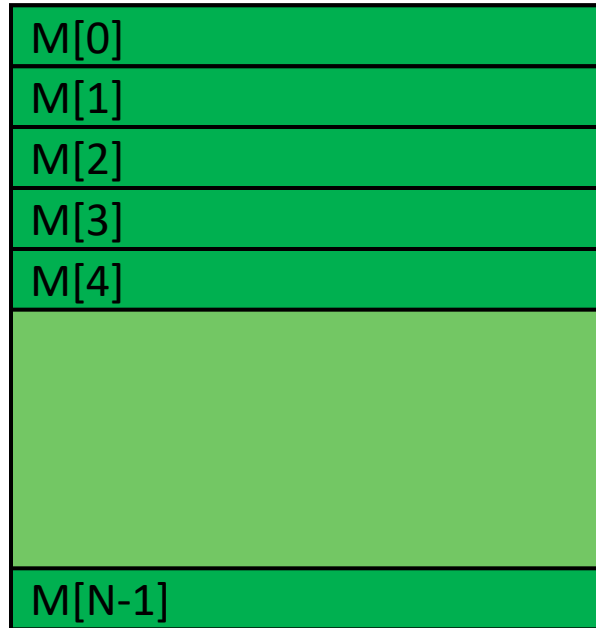


Control Unit

- The control unit is like the conductor of an orchestra
- It conducts the **step-by-step process of executing (every instruction in) a program**
- 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



Programmer Visible (Architectural) State



Memory

array of storage locations
indexed by an address



Registers

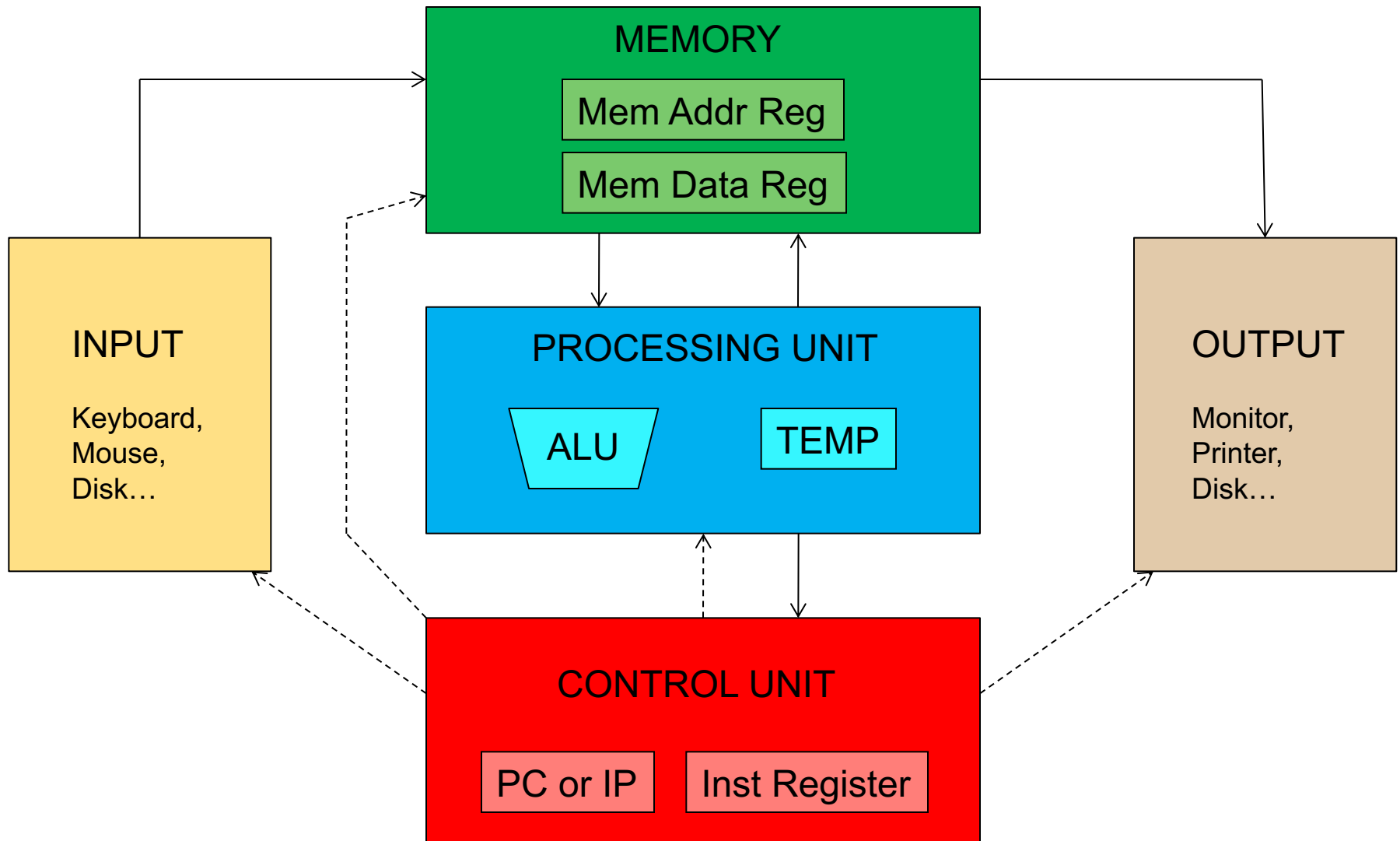
- given special names in the ISA (as opposed to addresses)
- general vs. special purpose

Program Counter

memory address
of the current (or next) instruction

Instructions (and programs) specify how to transform
the values of programmer visible state

The von Neumann Model

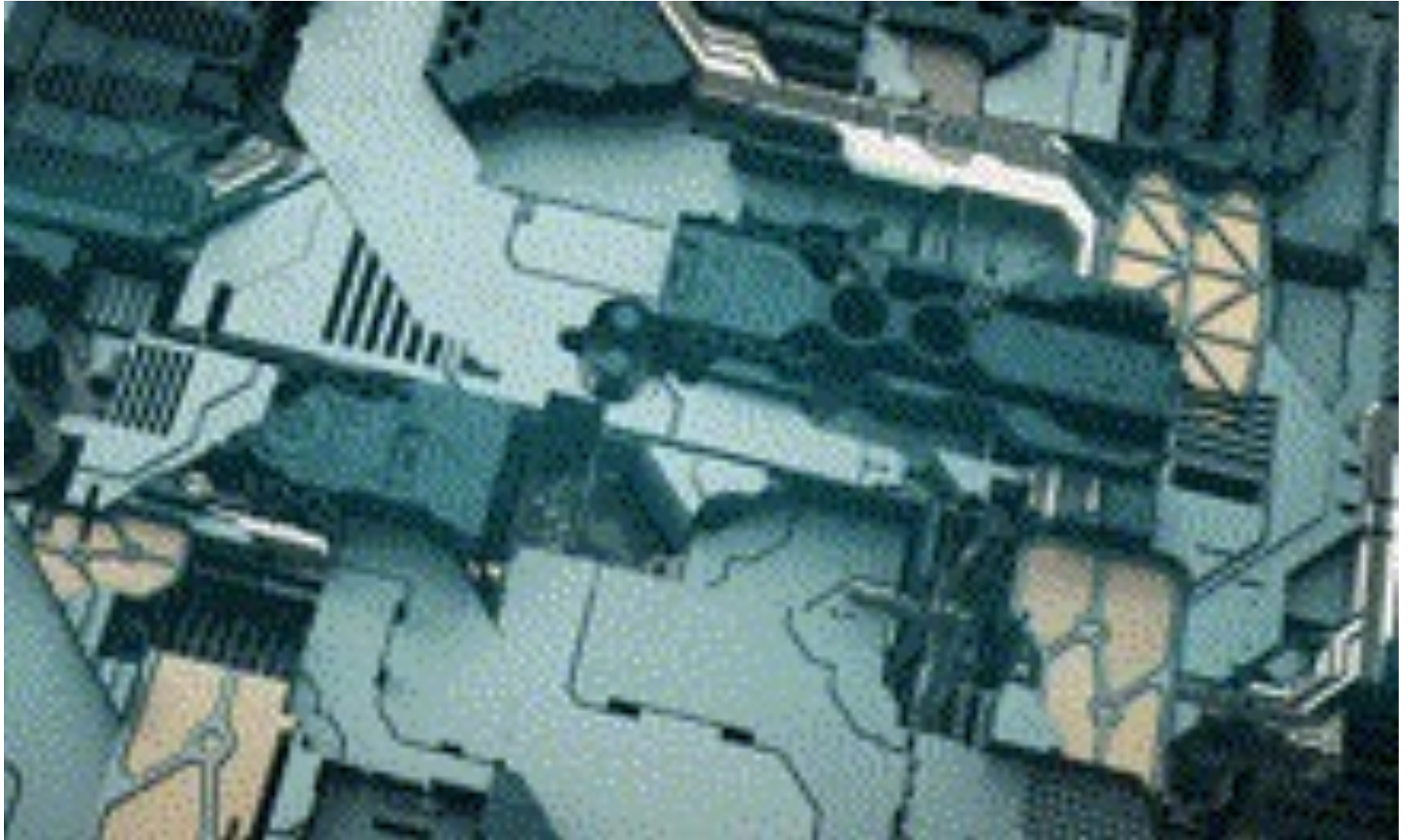


von Neumann Model: Two Key Properties

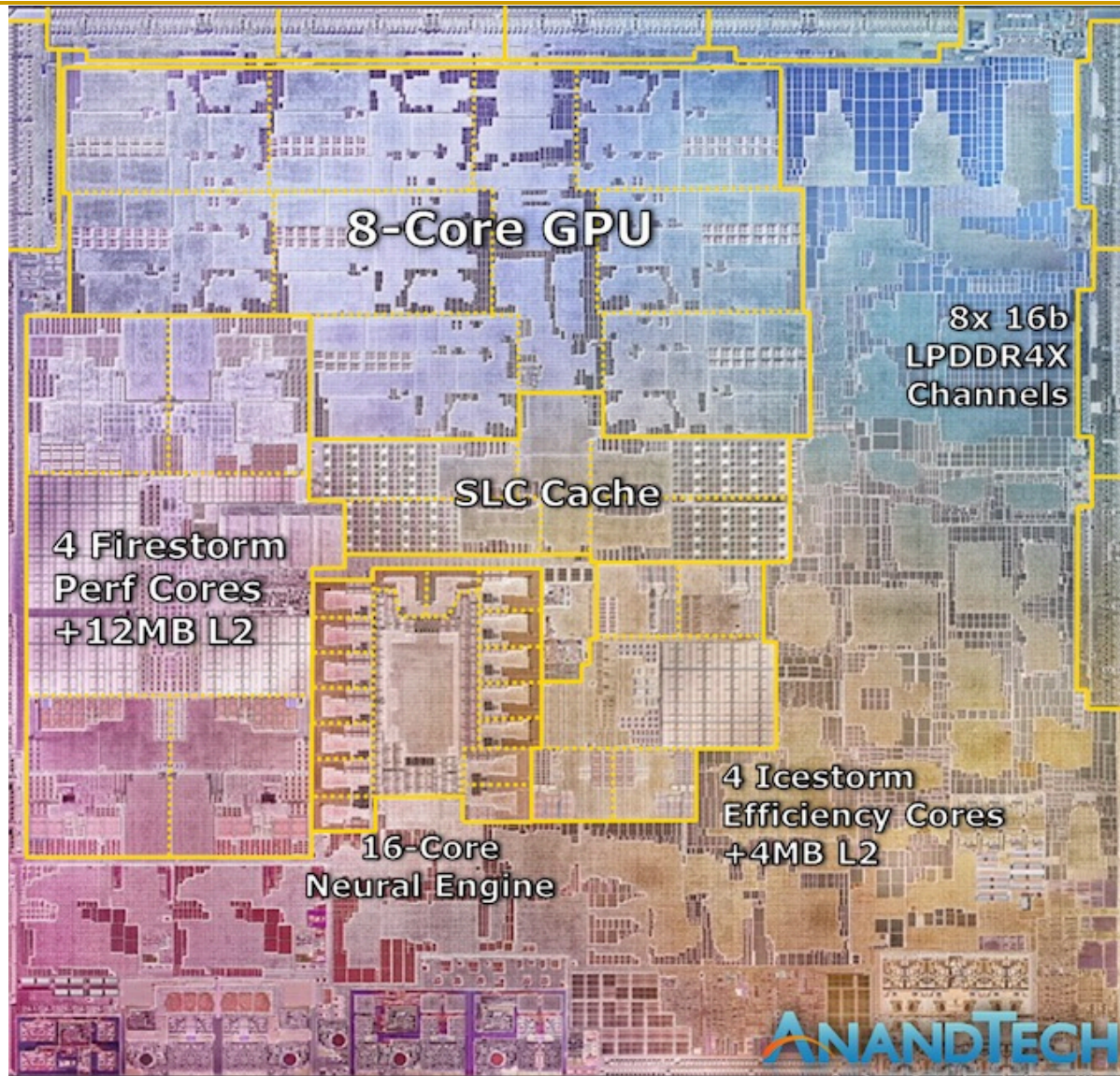
- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:
- **Stored program**
 - Instructions stored in a linear memory array
 - **Memory is unified** between instructions and data
 - The interpretation of a stored value depends on the control 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

LC-3: A von Neumann Machine



Another von Neumann Machine



Apple M1,
2021

Another von Neumann Machine

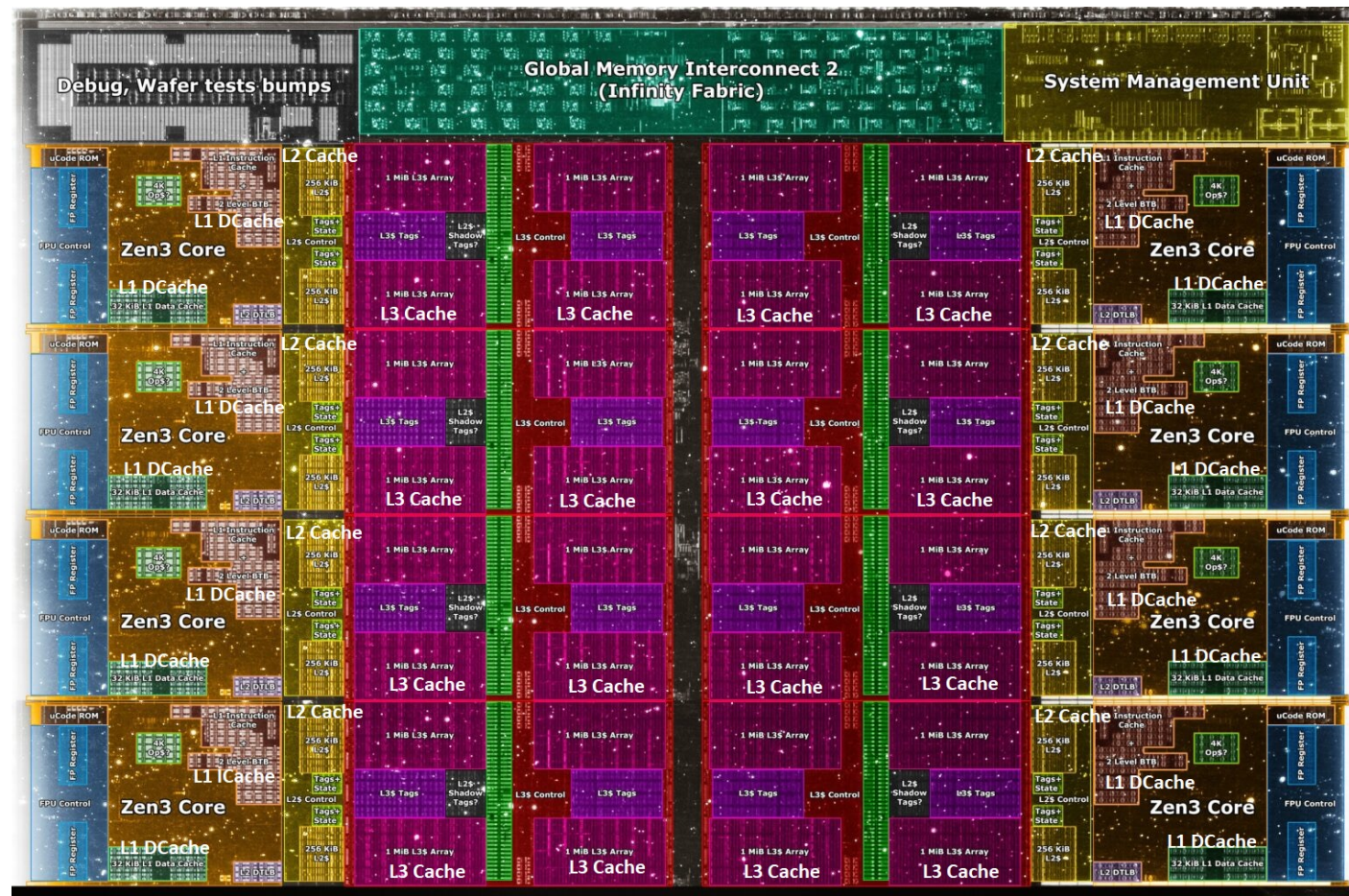


10nm ESF=Intel 7 Alder Lake die shot (~209mm²) from Intel: <https://www.intel.com/content/www/us/en/newsroom/news/12th-gen-core-processors.html>

Die shot interpretation by Locuza, October 2021

Intel Alder Lake,
2021

Another von Neumann Machine



Core Count:

8 cores/16 threads

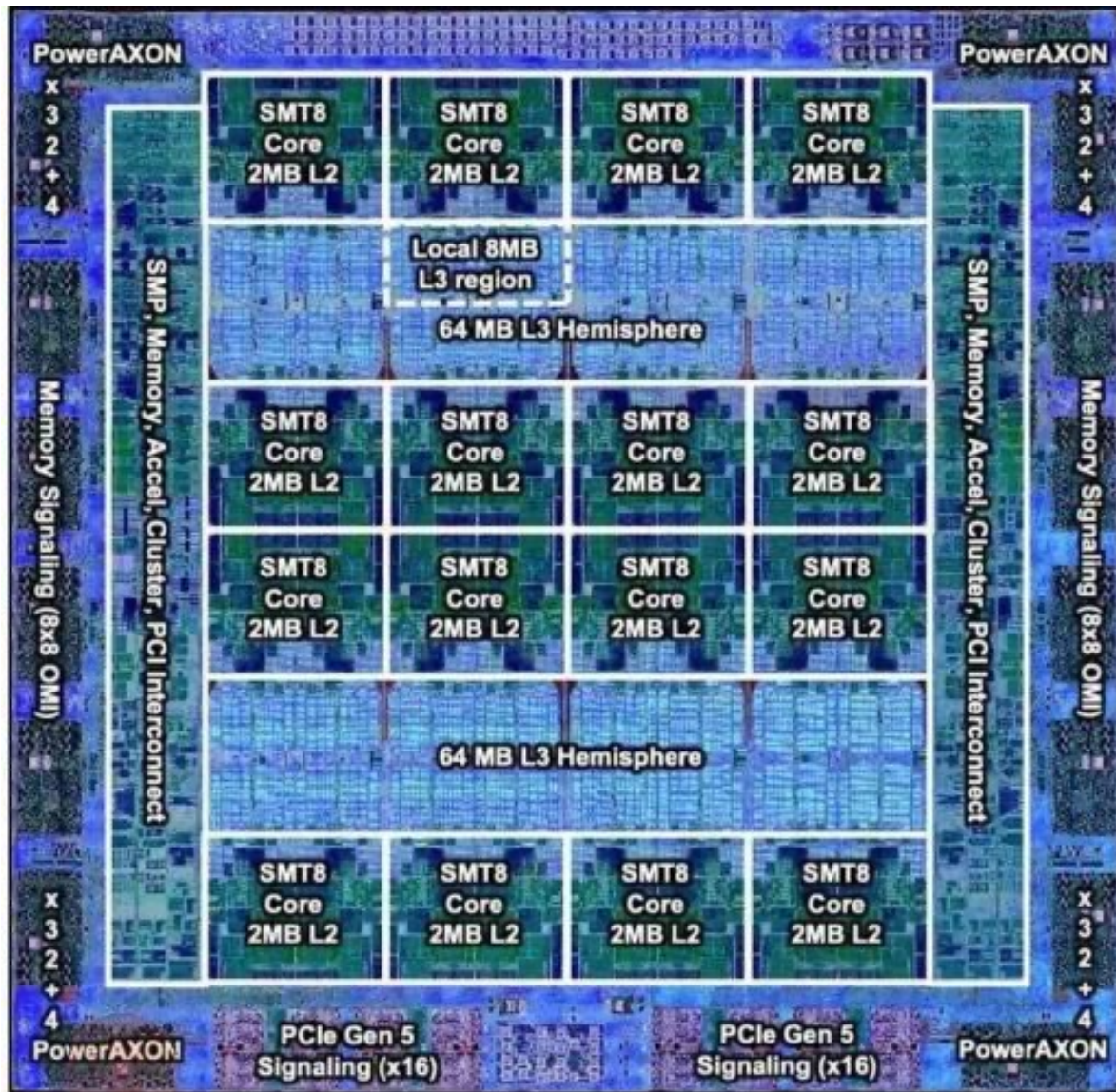
L1 Caches:
32 KB per core

L2 Caches:
512 KB per core

L3 Cache:
32 MB shared

AMD Ryzen 5000, 2020

Another 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

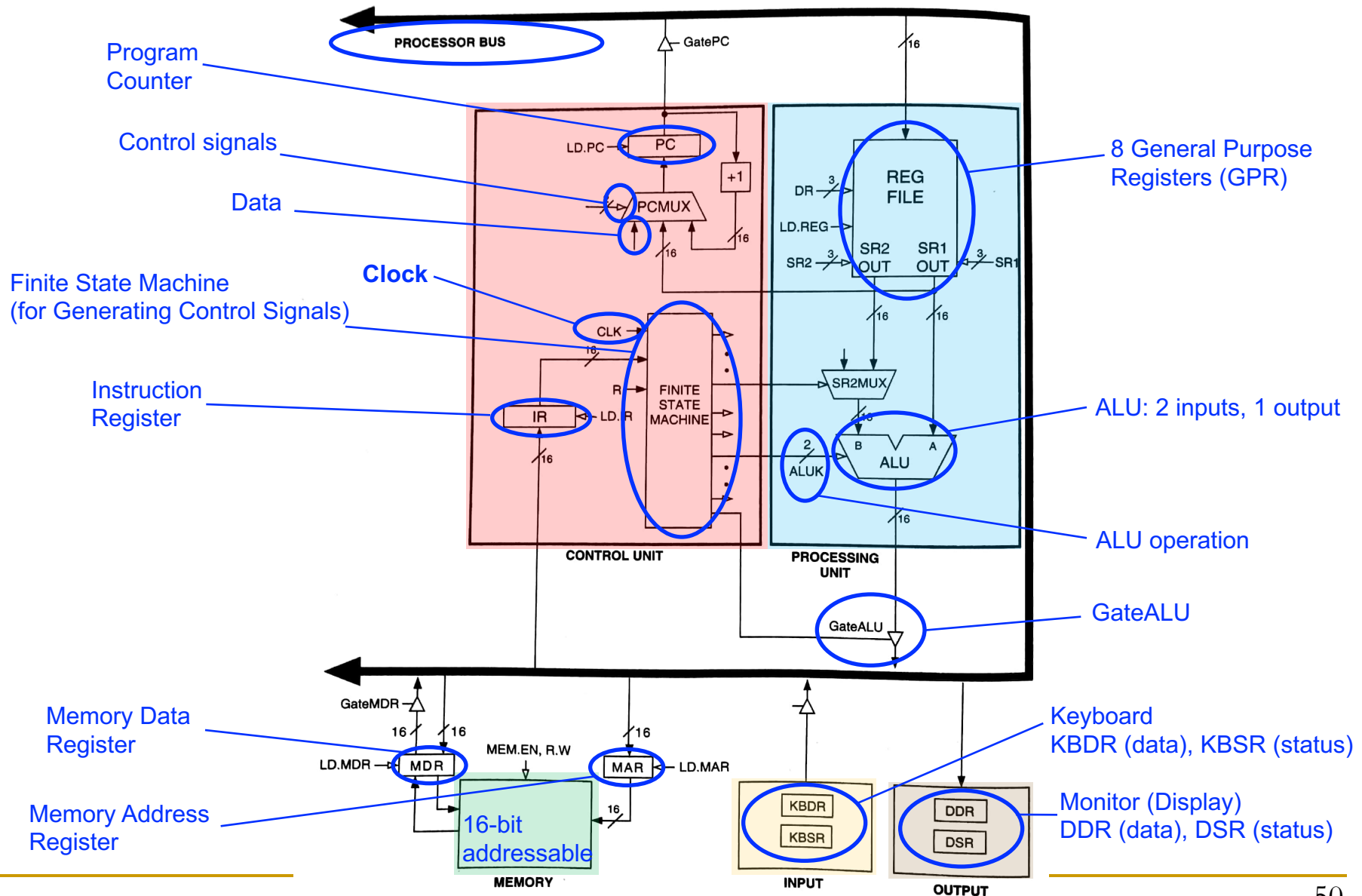


Figure 4.3 The LC-3 as an example of the von Neumann model

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

```
lw    $t2, 32($0)
add   $s0, $s1, $s2
addi  $t0, $s3, -12
sub   $t0, $t3, $t5
```

Machine code (encoded instructions)

```
0x8C0A0020
0x02328020
0x2268FFF4
0x016D4022
```

Byte Address	Instructions
⋮	⋮
0040000C	0 1 6 D 4 0 2 2
00400008	2 2 6 8 F F F 4
00400004	0 2 3 2 8 0 2 0
00400000	8 C 0 A 0 0 2 0 ← 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, ...)

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

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

High-level code

```
a = b + c;
```

Assembly

```
add a, b, c
```

- **add**: mnemonic to indicate the operation to perform
- **b, c**: source operands
- **a**: destination operand
- $a \leftarrow b + c$

Registers

- We map variables to registers

Assembly

```
add a, b, c
```

LC-3 registers

```
b = R1
```

```
c = R2
```

```
a = R0
```

MIPS registers

```
b = $s1
```

```
c = $s2
```

```
a = $s0
```

From Assembly to Machine Code in LC-3

■ Addition

LC-3 assembly

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

Field Values

OP	DR	SR1	SR2		
1	0	1	0	00	2

Machine Code (Instruction Encoding)

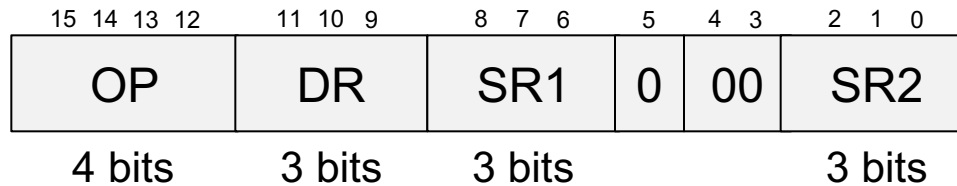
OP	DR	SR1	SR2		
0 0 0 1	0 0 0	0 0 1	0	0 0	0 1 0
15 14 13 12	11 10 9	8 7 6	5	4 3	2 1 0

0x1042

Machine Code, in short (hexadecimal)

Instruction Format (or Encoding)

■ LC-3 Operate Instruction Format



□ OP = **opcode** (what the instruction does)

■ E.g., ADD = 0001

□ **Semantics:** $DR \leftarrow SR1 + SR2$

■ E.g., AND = 0101

□ **Semantics:** $DR \leftarrow SR1 \text{ AND } SR2$

□ SR1, SR2 = source registers

□ DR = destination register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	1	0	0	0	0	1	1	0
ADD				R6			R2			R6					

From Assembly to Machine Code in MIPS

■ Addition

MIPS assembly

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

Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32

$rd \leftarrow rs + rt$

Machine Code (Instruction Encoding)

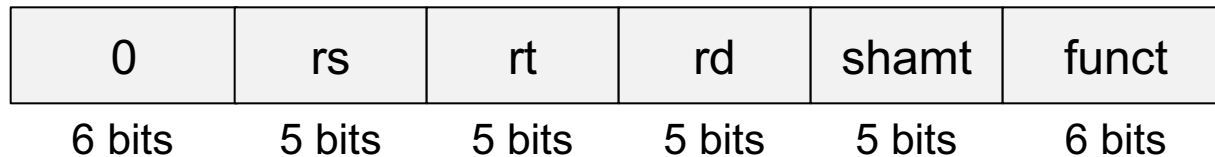
op		rs		rt		rd		shamt		funct	
000000		10001		10010		10000		00000		100000	
31	26	25	21	20	16	15	11	10	6	5	0

0x02328020

Instruction Format: R-Type in MIPS

■ MIPS R-type Instruction Format

- 3 register operands



- 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

High-level code

```
a = A[i];
```

Assembly

```
load a, A, i
```

- ❑ **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

```
LDR R3, R0, #2
```

$R3 \leftarrow \text{Memory}[R0 + 2]$

■ MIPS assembly (assuming word-addressable)

High-level code

```
a = A[2];
```

MIPS assembly

```
lw $s3, 2($s0)
```

$\$s3 \leftarrow \text{Memory}[\$s0 + 2]$

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)
```

$\$s3 \leftarrow \text{Memory}[\$s0 + 8]$

- Byte address is calculated as: $\text{word_address} * \text{bytes/word}$
 - 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

```
LDR R3, R0, #2
```

Field Values

OP	DR	BaseR	offset6
6	3	0	2
15 12	11 9	8 6 5	0

■ MIPS

MIPS assembly

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

Field Values

op	rs	rt	imm
35	16	19	8
31 26	25 21	20 16 15	0

I-Type

Instruction (Processing) Cycle

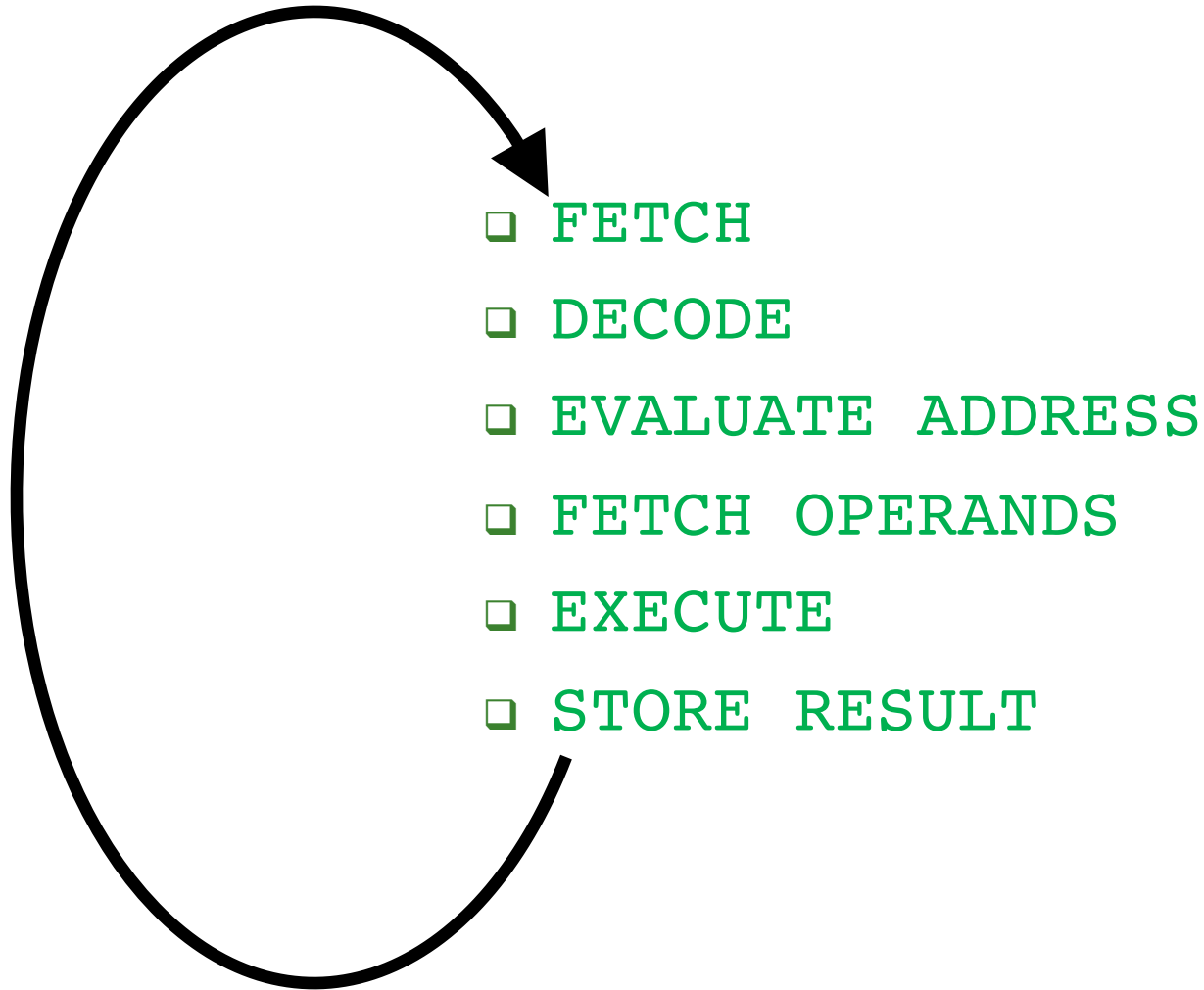
How Are These Instructions Executed?

- 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 is 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

- 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

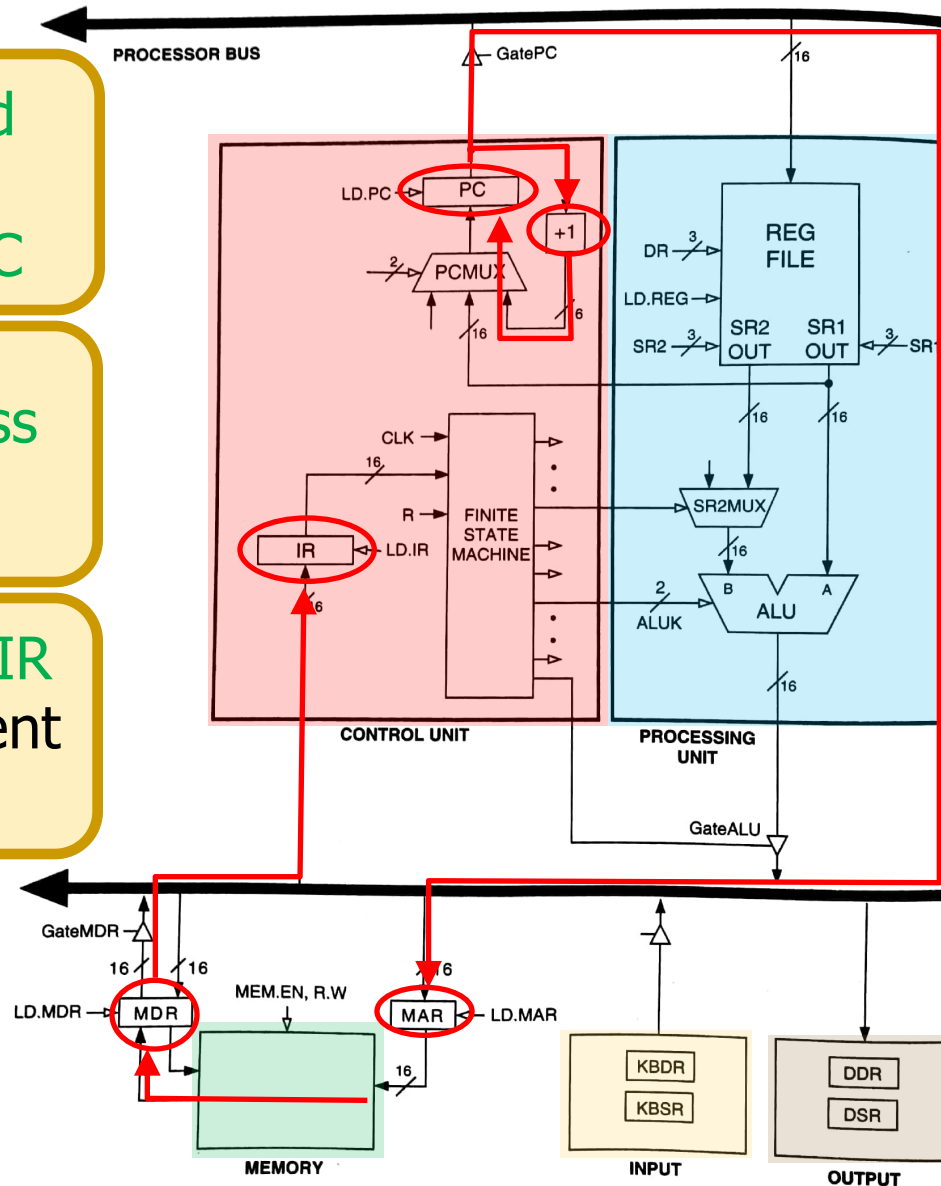


Figure 4.3 The LC-3 as an example of the von Neumann model

DECODE

- 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

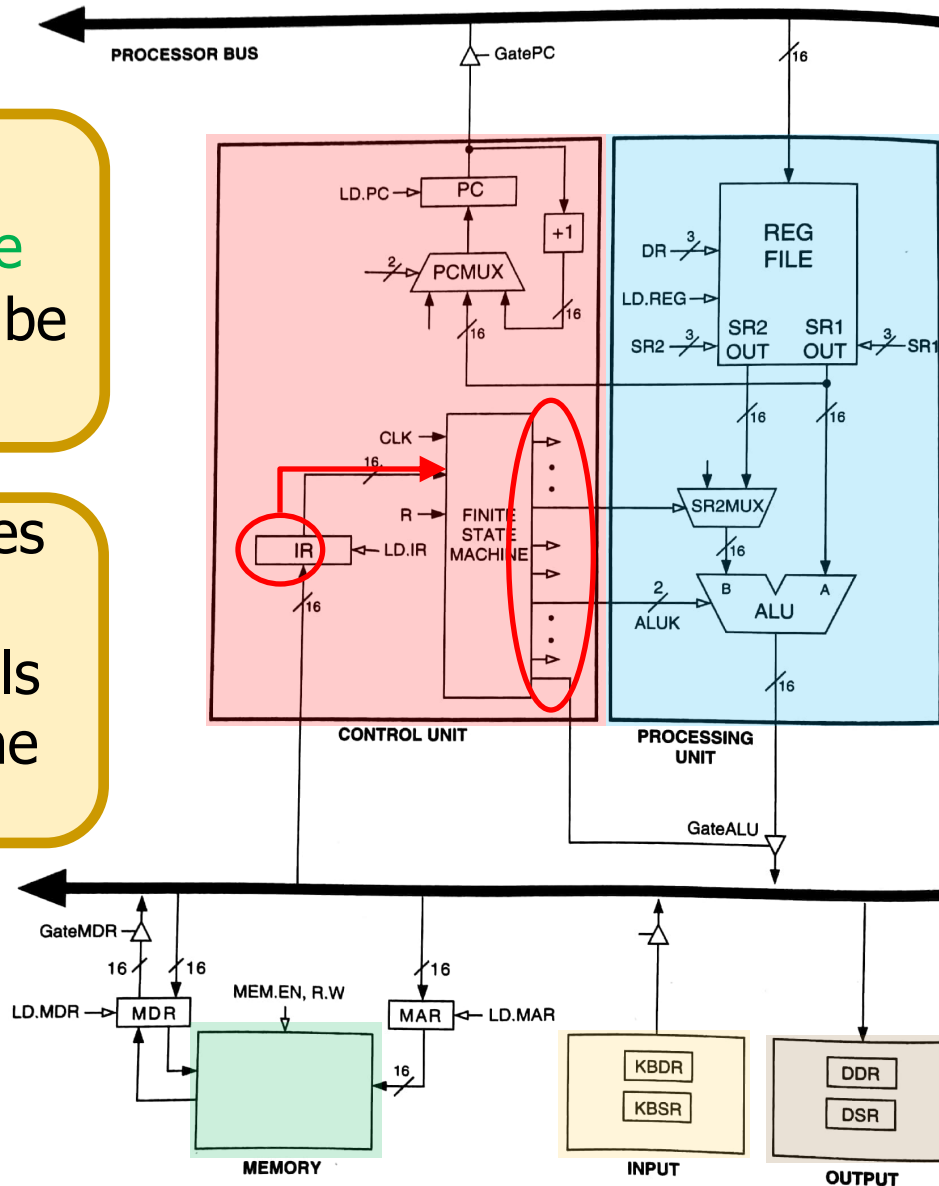
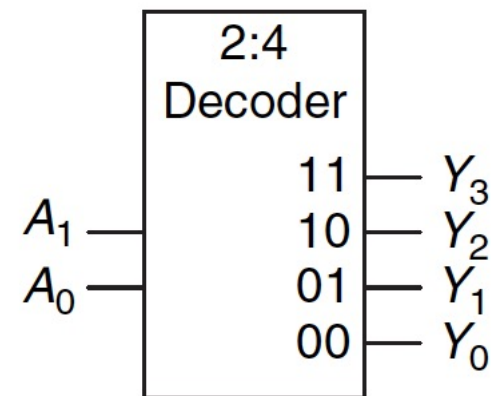


Figure 4.3 The LC-3 as an example of the von Neumann model

Recall: Decoder

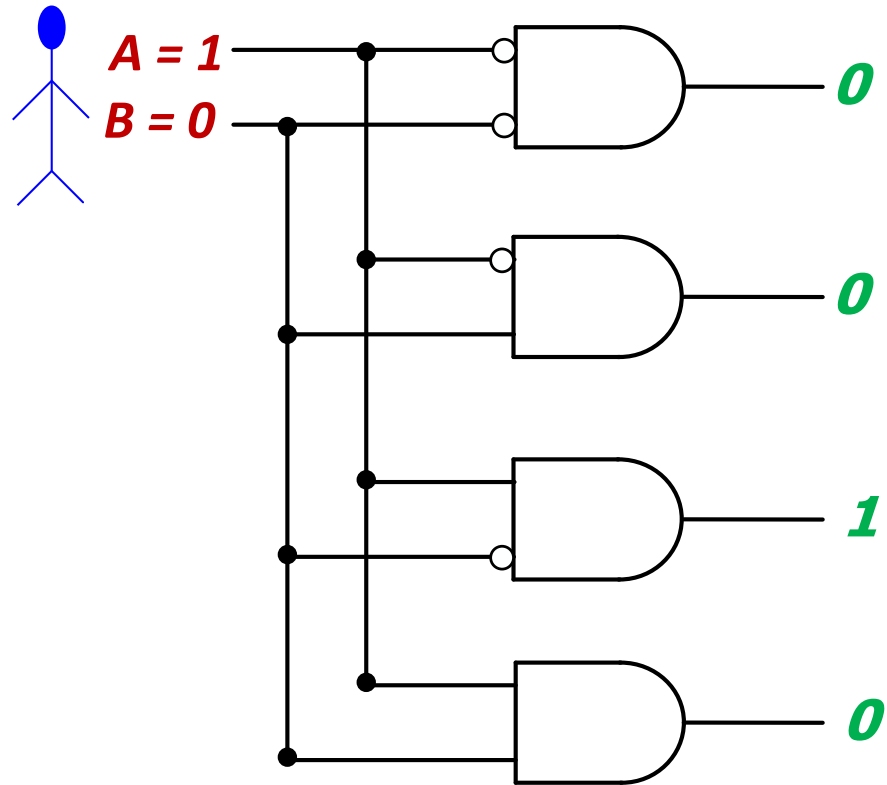
- “Input pattern detector”
- n inputs and 2^n outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The **output** that is logically 1 is the output corresponding to the input **pattern** that the logic circuit is expected to detect
- Example: 2-to-4 decoder

A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



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*)**



EVALUATE ADDRESS

- 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

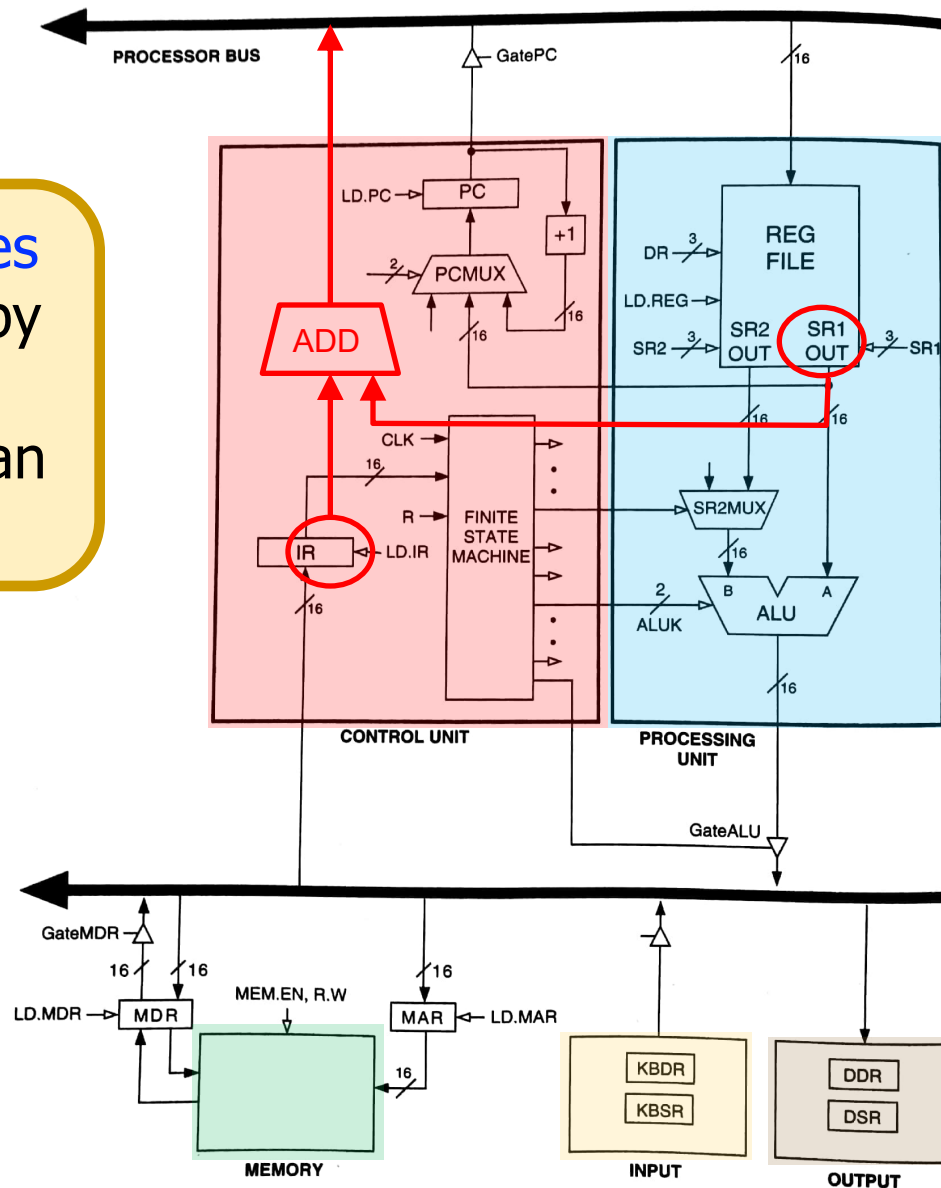


Figure 4.3 The LC-3 as an example of the von Neumann model

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)

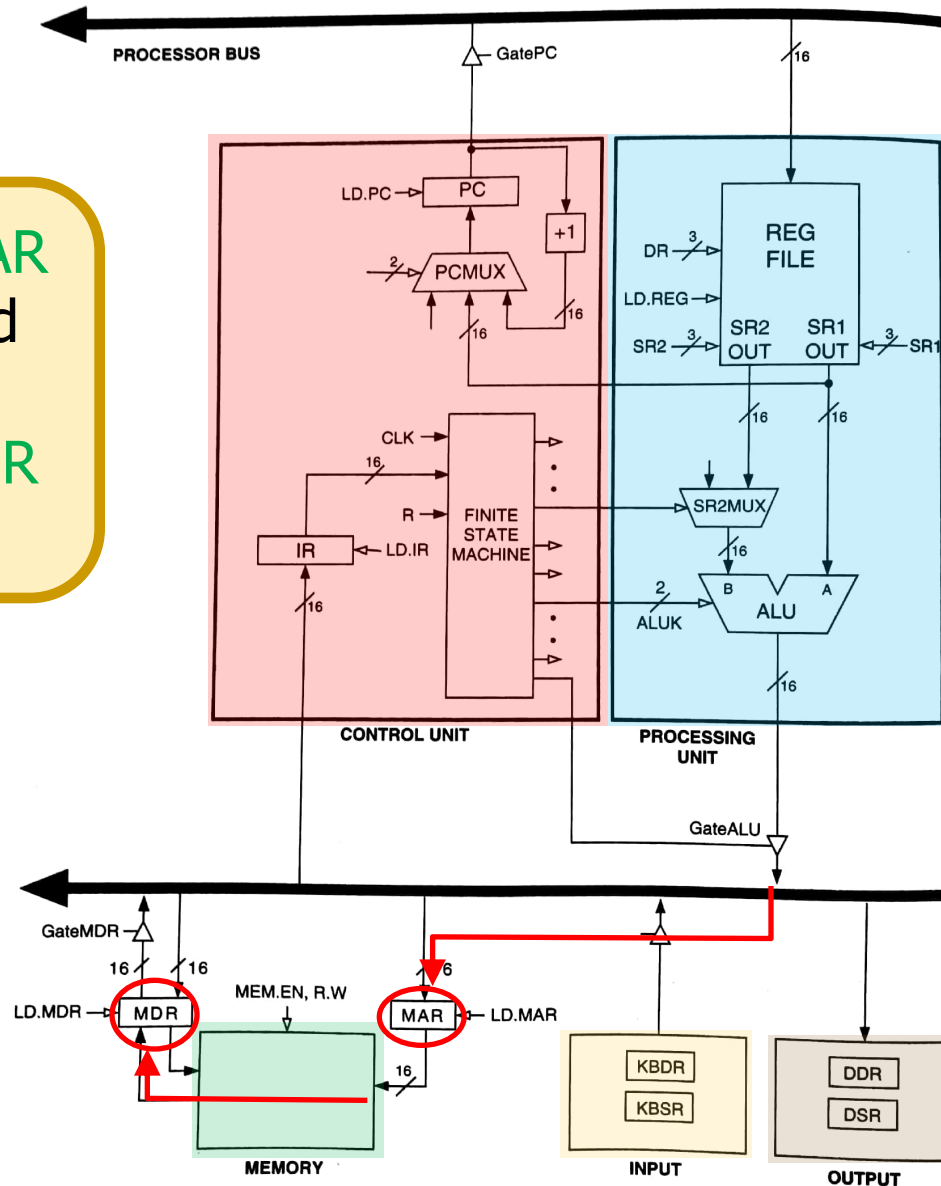


Figure 4.3 The LC-3 as an example of the von Neumann model

EXECUTE

- 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

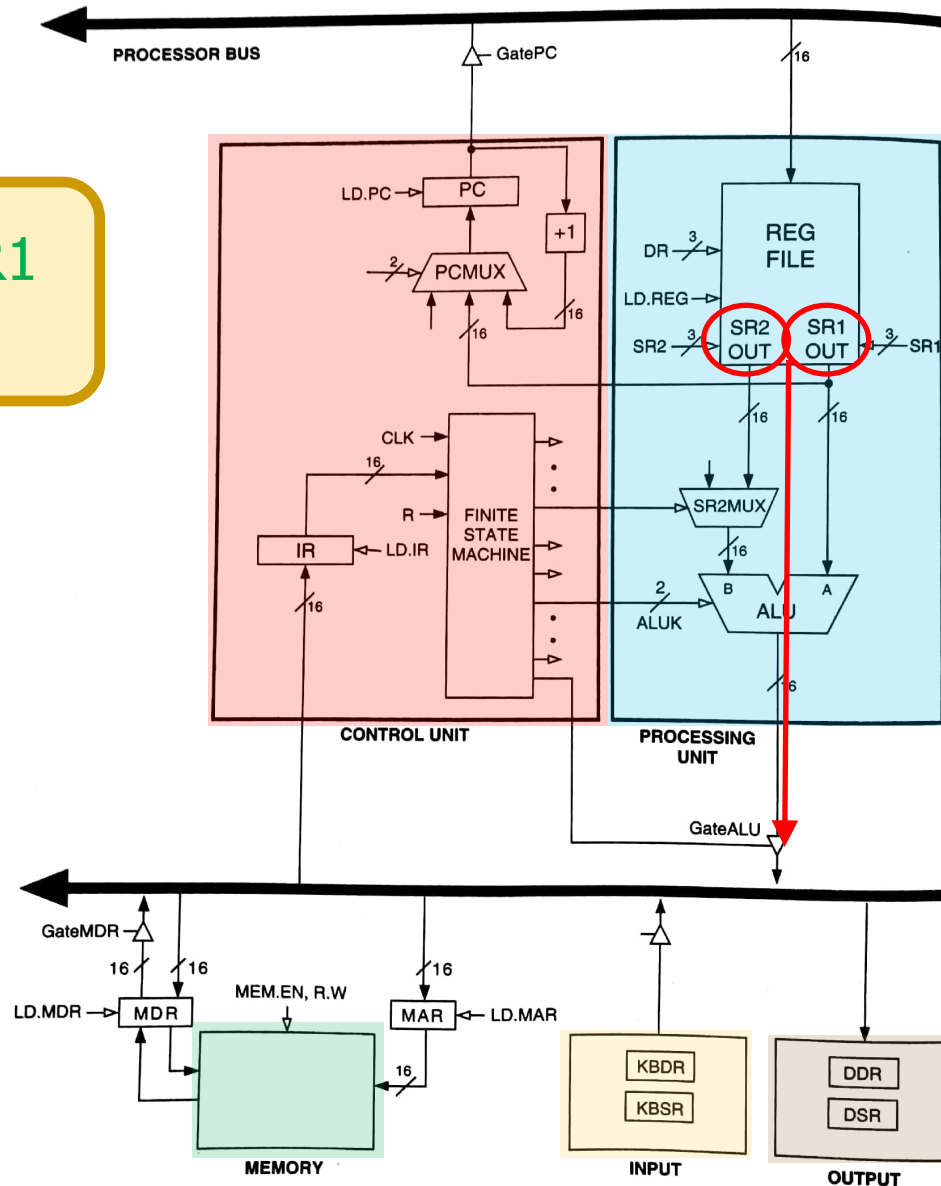


Figure 4.3 The LC-3 as an example of the von Neumann model

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)

STORE RESULT in LC-3

ADD loads ALU
Result into DR

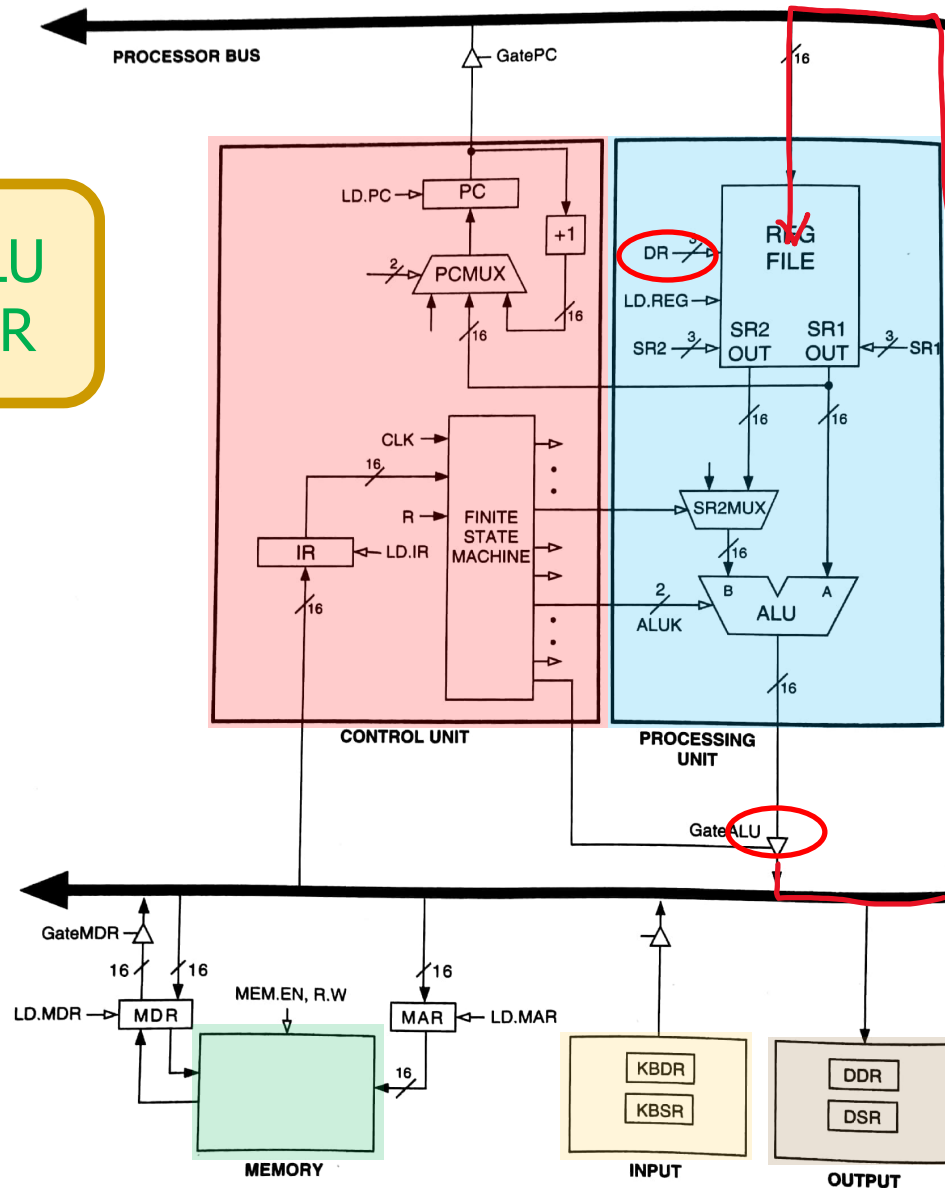


Figure 4.3 The LC-3 as an example of the von Neumann model

STORE RESULT in LC-3

LDR loads
MDR into DR

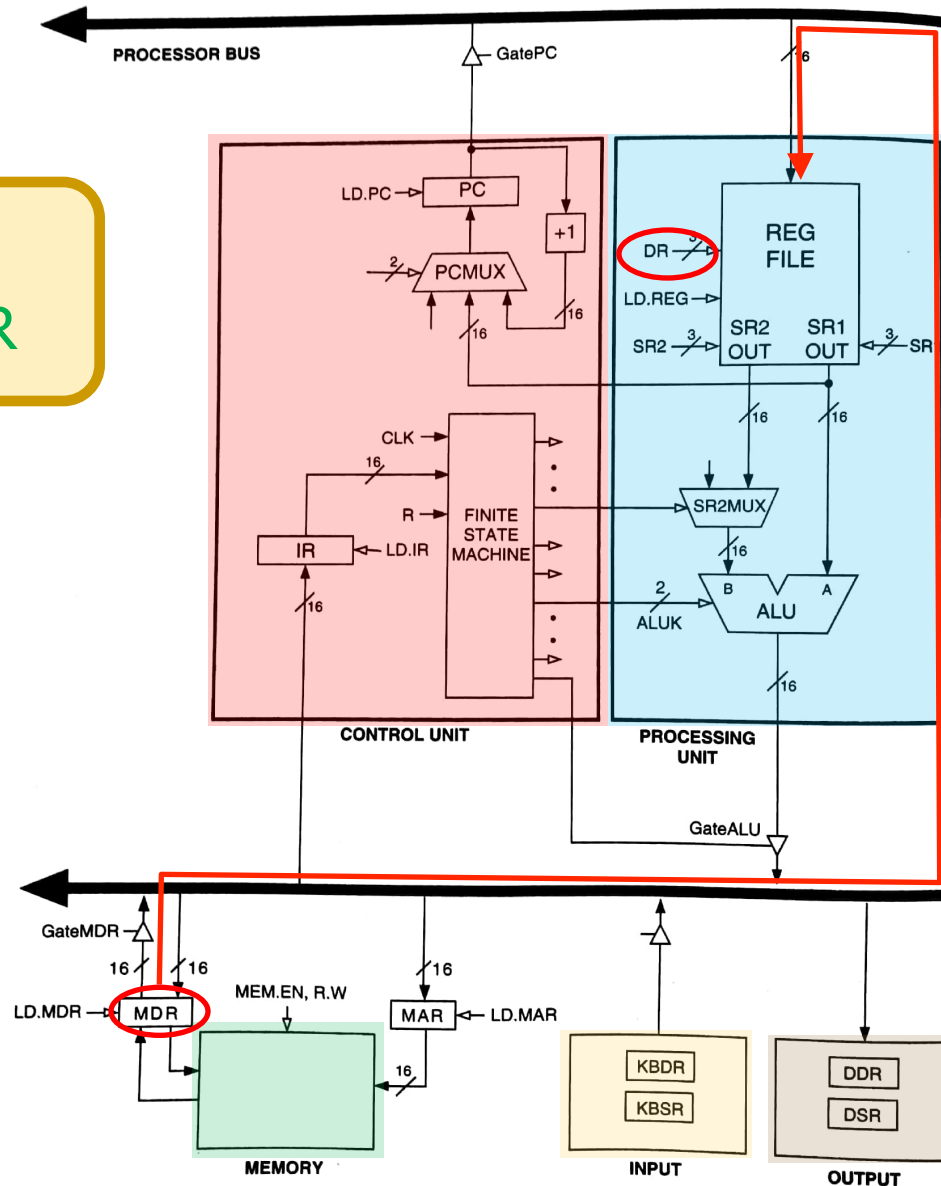
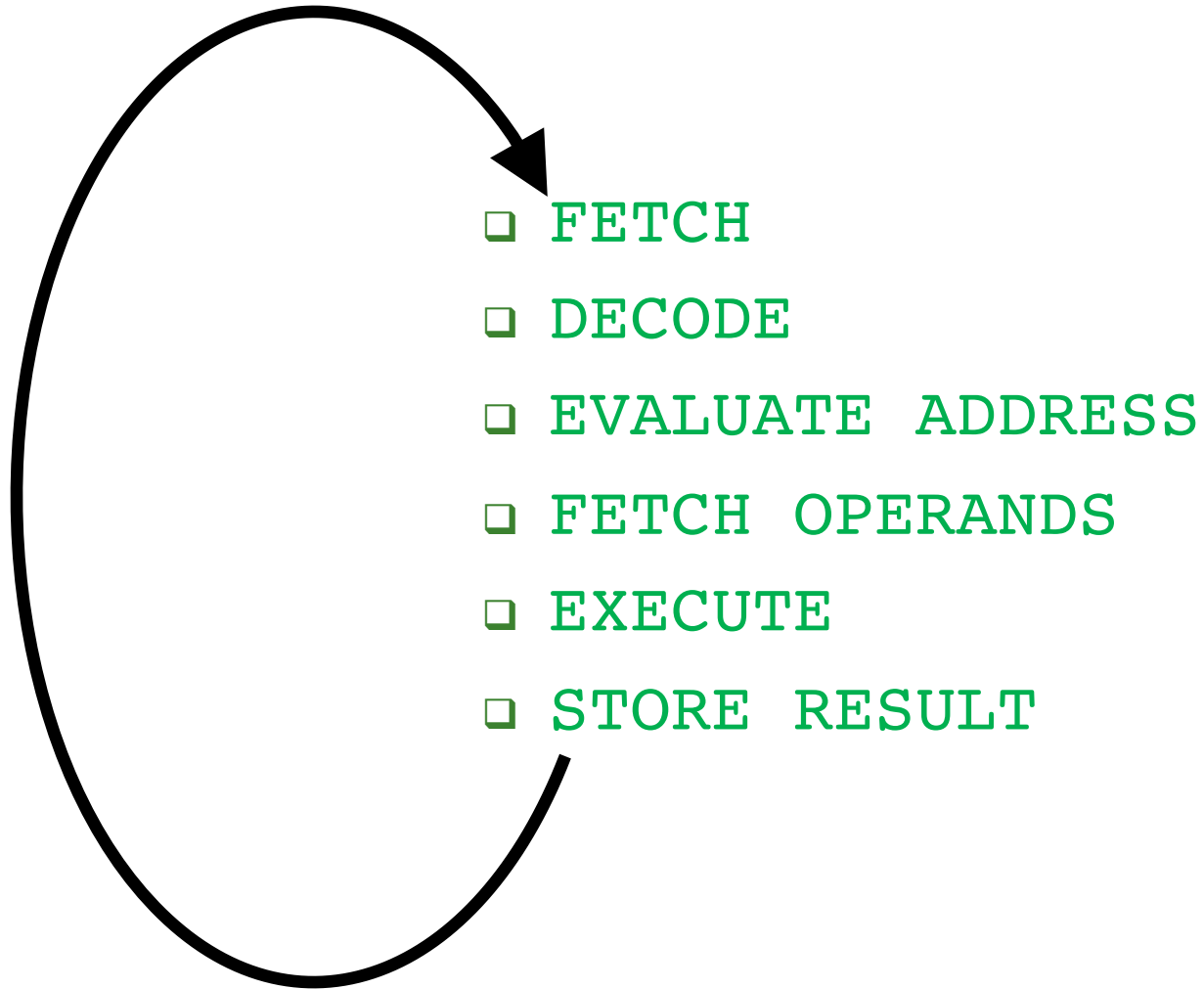


Figure 4.3 The LC-3 as an example of the von Neumann model

The Instruction Cycle



Changing the Sequence of Execution

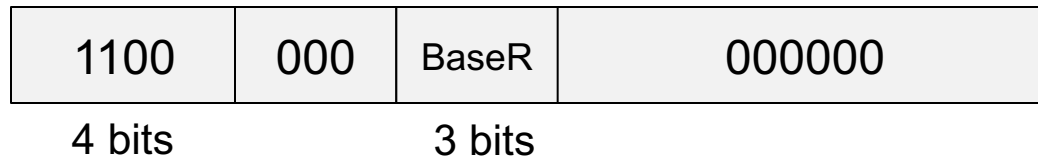
- A computer program **executes in sequence** (i.e., in program order)
 - First instruction, second instruction, third instruction and so on
- Unless we **change the sequence of execution**
- **Control instructions** allow a program to execute **out of sequence**
 - They can change the PC by loading 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



- ❑ BaseR = Base register
- ❑ $PC \leftarrow R2$ (Register identified by BaseR)
- ❑ Variations
 - RET: special case of JMP where BaseR = R7
 - JSR, JSRR: jump to subroutine

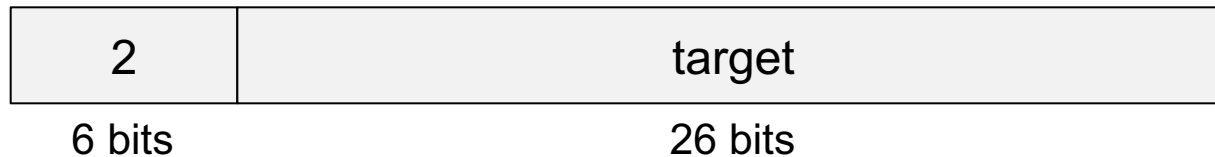
This is register addressing mode

Jump in MIPS

- Unconditional branch or jump

- MIPS

j target



J-Type

- 2 = opcode
- target = target address
- $PC \leftarrow PC^+ [31:28] \mid \text{sign-extend}(\text{target}) * 4$

- Variations

- jal: jump and link (function calls)

- jr: jump register

jr \$s0

j uses pseudo-direct addressing mode

jr uses register addressing mode

[†] This is the incremented PC

PC UPDATE in LC-3

JMP loads
SR1 into PC

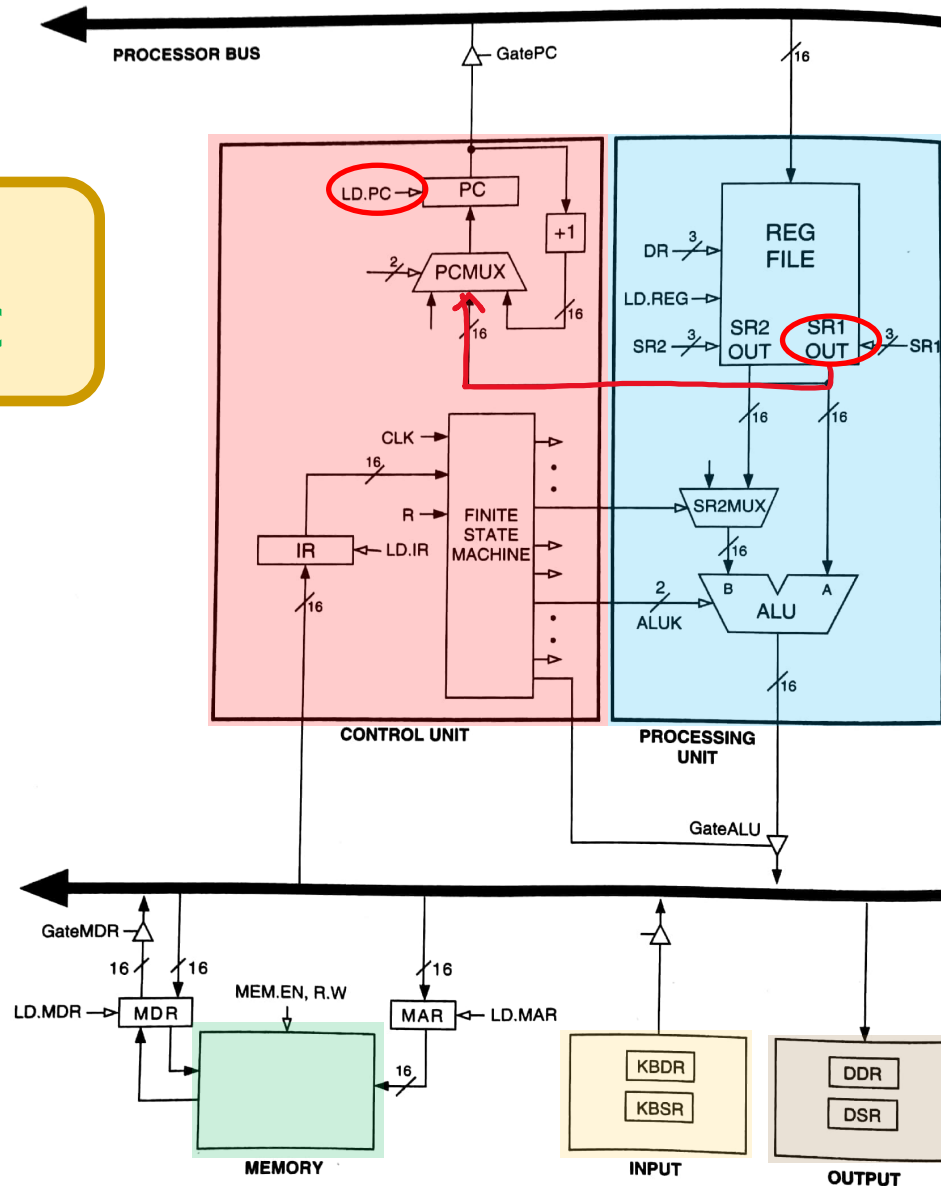
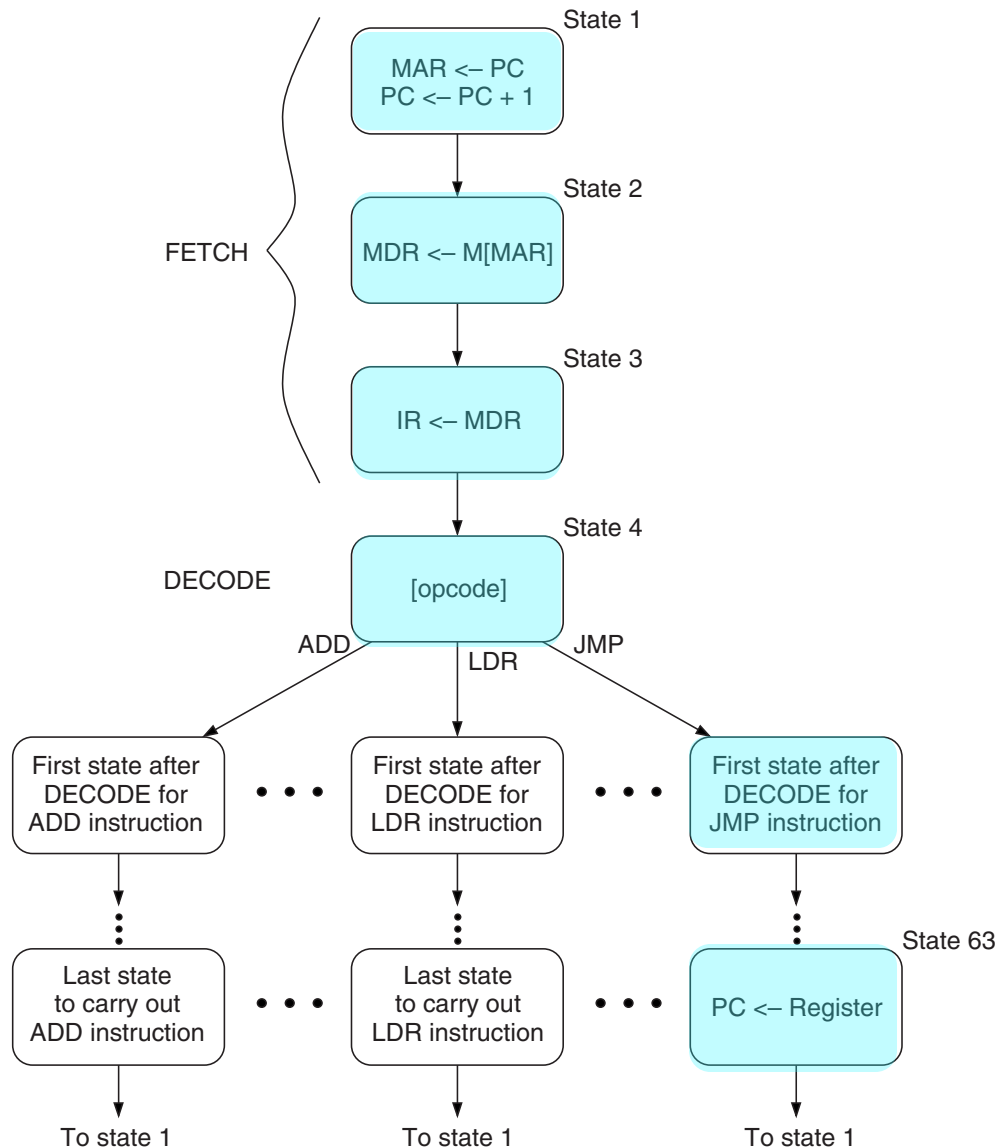


Figure 4.3 The LC-3 as an example of the von Neumann model

Control of the Instruction Cycle

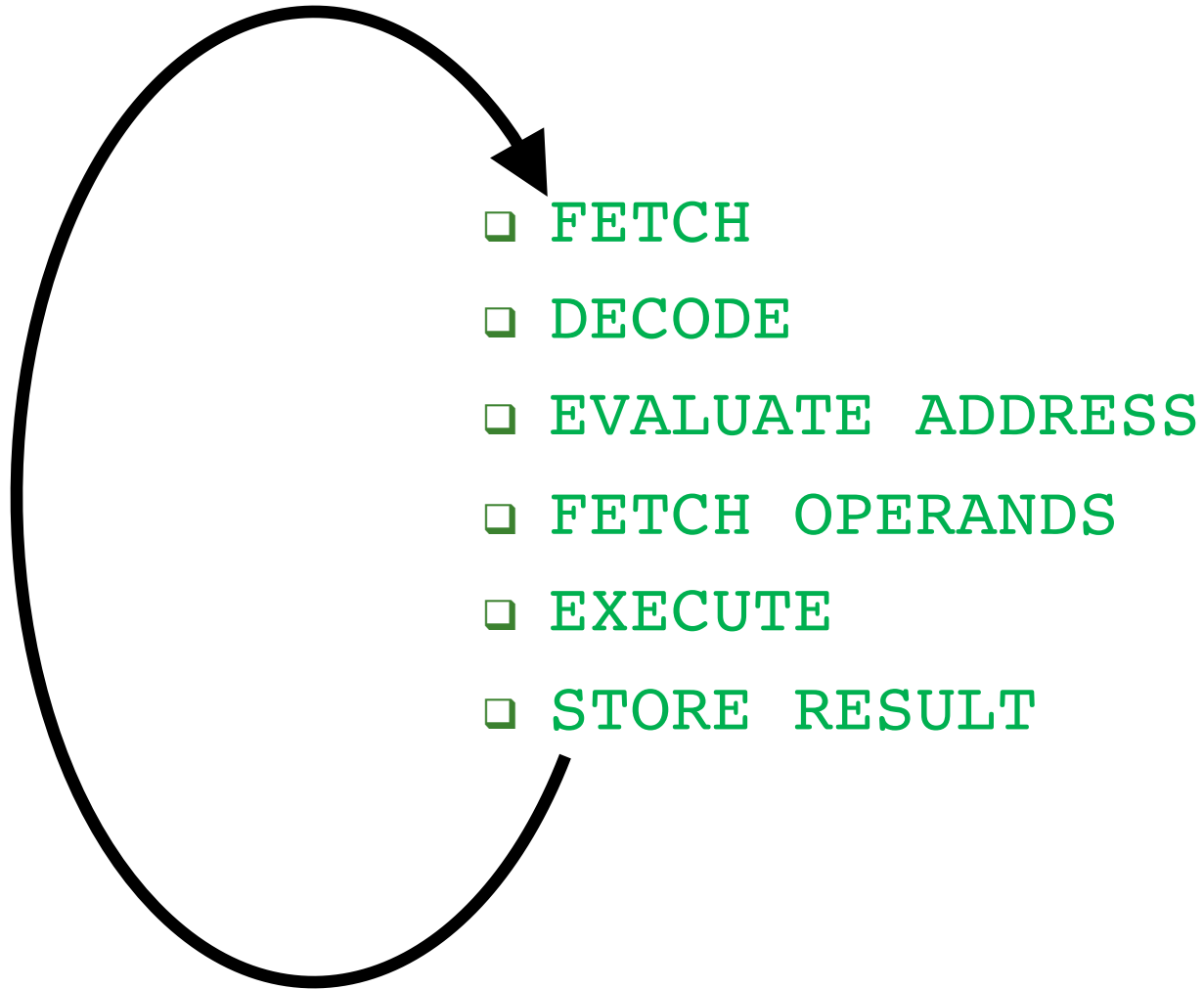


- State 1
 - The FSM asserts GatePC and LD.MAR
 - It selects input (+1) in PCMUX and asserts LD.PC
- State 2
 - MDR is loaded with the instruction
- State 3
 - The FSM asserts GateMDR and LD.IR
- State 4
 - The FSM goes to next state depending on opcode
- State 63
 - JMP loads register into PC
- Full state diagram in Patt&Pattel, Appendix C

Figure 4.4 An abbreviated state diagram of the LC-3

This is an FSM Controlling the LC-3 Processor

The Instruction Cycle



We Covered Until Here
in Lecture

Digital Design & Computer Arch.

Lecture 9: Von Neumann Model & Instruction Set Architectures

Prof. Onur Mutlu

ETH Zürich

Spring 2022

24 March 2022

LC-3 and MIPS

Instruction Set Architectures

The Instruction Set

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

ADD

OP	DR	SR1	SR2		
1	0	1	0	00	2

Register mode

LDR

OP	DR	BaseR	offset6
6	3	0	4

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**
 - R0 to R7 in LC-3
 - 32 registers in MIPS
 - The **instruction set**
 - Opcodes
 - Data types
 - Addressing modes
 - Length and format of instructions

Problem
Algorithm
Program
ISA
Microarchitecture
Circuits
Electrons

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
 - ❑ Hardware complexity vs. software complexity
- In LC-3 and in MIPS there are three **types of opcodes**
 - ❑ Operate
 - ❑ Data movement
 - ❑ Control

Opcodes in LC-3

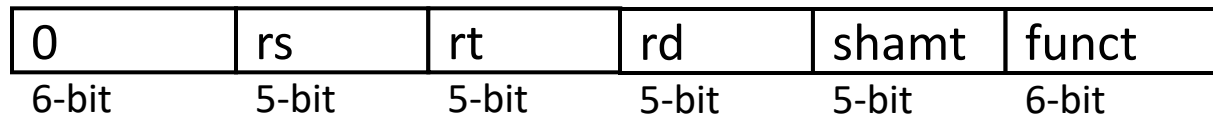
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD ⁺	0001				DR			SR1		0	00				SR2	
ADD ⁺	0001				DR			SR1		1				imm5		
AND ⁺	0101				DR			SR1		0	00				SR2	
AND ⁺	0101				DR			SR1		1				imm5		
BR	0000				n	z	p									PCOffset9
JMP	1100				000			BaseR								000000
JSR	0100				1											PCOffset11
JSRR	0100				0		00	BaseR								000000
LD ⁺	0010				DR											PCOffset9
LDI ⁺	1010				DR											PCOffset9
LDR ⁺	0110				DR			BaseR								offset6
LEA ⁺	1110				DR											PCOffset9
NOT ⁺	1001				DR			SR								111111
RET	1100				000			111								000000
RTI	1000															000000000000
ST	0011				SR											PCOffset9
STI	1011				SR											PCOffset9
STR	0111				SR			BaseR								offset6
TRAP	1111				0000											trapvect8
reserved	1101															

Figure 5.3 Formats of the entire LC-3 instruction set. NOTE: + indicates instructions that modify condition codes

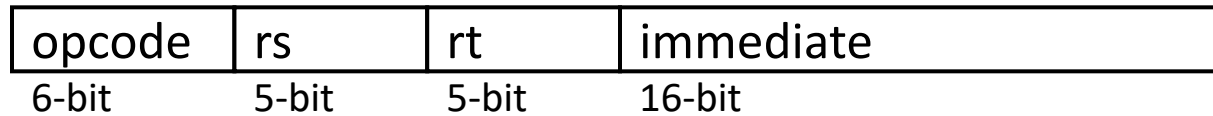
Opcodes in LC-3b

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD ⁺	0001				DR			SR1			A	op.spec				
AND ⁺	0101				DR			SR1			A	op.spec				
BR	0000				n	z	p	PCOffset9								
JMP	1100				000			BaseR			000000					
JSR(R)	0100				A	operand.specifier										
LDB ⁺	0010				DR			BaseR			boffset6					
LDW ⁺	0110				DR			BaseR			offset6					
LEA ⁺	1110				DR			PCOffset9								
RTI	1000				000000000000											
SHF ⁺	1101				DR			SR			A	D	amount4			
STB	0011				SR			BaseR			boffset6					
STW	0111				SR			BaseR			offset6					
TRAP	1111				0000			trapvect8								
XOR ⁺	1001				DR			SR1			A	op.spec				
not used	1010															
not used	1011															

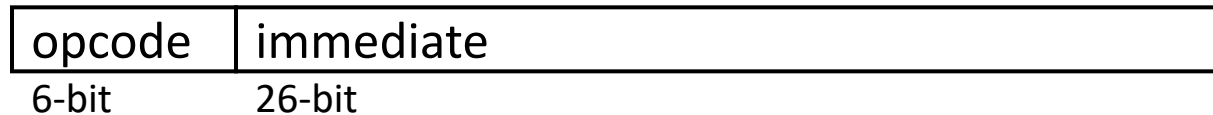
MIPS Instruction Types



R-type



I-type



J-type

Funct in MIPS R-Type Instructions (I)

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

Funct	Name	Description	Operation
000000 (0)	sll rd, rt, shamt	shift left logical	[rd] = [rt] << shamt
000010 (2)	srl rd, rt, shamt	shift right logical	[rd] = [rt] >> shamt
000011 (3)	sra rd, rt, shamt	shift right arithmetic	[rd] = [rt] >>> shamt
000100 (4)	sllv rd, rt, rs	shift left logical variable	[rd] = [rt] << [rs] _{4:0}
000110 (6)	srlv rd, rt, rs	shift right logical variable	[rd] = [rt] >> [rs] _{4:0}
000111 (7)	srav rd, rt, rs	shift right arithmetic variable	[rd] = [rt] >>> [rs] _{4:0}
001000 (8)	jr rs	jump register	PC = [rs]
001001 (9)	jalr rs	jump and link register	\$ra = PC + 4, PC = [rs]
001100 (12)	syscall	system call	system call exception
001101 (13)	break	break	break exception
010000 (16)	mfhi rd	move from hi	[rd] = [hi]
010001 (17)	mthi rs	move to hi	[hi] = [rs]
010010 (18)	mflo rd	move from lo	[rd] = [lo]
010011 (19)	mtlo rs	move to lo	[lo] = [rs]
011000 (24)	mult rs, rt	multiply	{[hi], [lo]} = [rs] × [rt]
011001 (25)	multu rs, rt	multiply unsigned	{[hi], [lo]} = [rs] × [rt]
011010 (26)	div rs, rt	divide	[lo] = [rs]/[rt], [hi] = [rs]%[rt]
011011 (27)	divu rs, rt	divide unsigned	[lo] = [rs]/[rt], [hi] = [rs]%[rt]

(continued)

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

Funct in MIPS R-Type Instructions (II)

Table B.2 R-type instructions, sorted by funct field—Cont'd

Funct	Name	Description	Operation
100000 (32)	add rd, rs, rt	add	[rd] = [rs] + [rt]
100001 (33)	addu rd, rs, rt	add unsigned	[rd] = [rs] + [rt]
100010 (34)	sub rd, rs, rt	subtract	[rd] = [rs] - [rt]
100011 (35)	subu rd, rs, rt	subtract unsigned	[rd] = [rs] - [rt]
100100 (36)	and rd, rs, rt	and	[rd] = [rs] & [rt]
100101 (37)	or rd, rs, rt	or	[rd] = [rs] [rt]
100110 (38)	xor rd, rs, rt	xor	[rd] = [rs] ^ [rt]
100111 (39)	nor rd, rs, rt	nor	[rd] = ~([rs] [rt])
101010 (42)	slt rd, rs, rt	set less than	[rs] < [rt] ? [rd] = 1 : [rd] = 0
101011 (43)	sltu rd, rs, rt	set less than unsigned	[rs] < [rt] ? [rd] = 1 : [rd] = 0

- Find the complete list of instructions in the H&H Appendix B

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
- Again, **tradeoffs** are involved
 - What data types should be supported and what should not be?

Data Type Tradeoffs

- What is the benefit of **having more or high-level data types** in the ISA?
- What is the disadvantage?
- Think compiler/programmer vs. microarchitect
- Concept of **semantic gap**
 - Data types coupled tightly to the semantic level, or complexity of instructions → how close are instrs. to high-level languages
- Example: Early RISC architectures vs. Intel 432
 - Early RISC machines: Only integer data type
 - Intel 432: Object data type, capability based machine
 - VAX: Complex types, e.g., doubly-linked list

Aside: An Example: **Binary**Coded**D**ecimal

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



"Binary clock"
Wikipedia.

on the English

http://commons.wikimedia.org/wiki/File:Binary_clock.svg#mediaviewer/File:Binary_clock.svg

"Digital-BCD-clock" by Julo - Own work. Licensed under Public Domain via Wikimedia Commons -
<http://commons.wikimedia.org/wiki/File:Digital-BCD-clock.jpg#mediaviewer/File:Digital-BCD-clock.jpg>

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 → reduced number of instructions and code size
 - Array indexing
 - Pointer-based accesses (indirection)
 - Sparse matrix accesses
- **Disadvantages:**
 - More work for the compiler
 - More work for the microarchitect

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

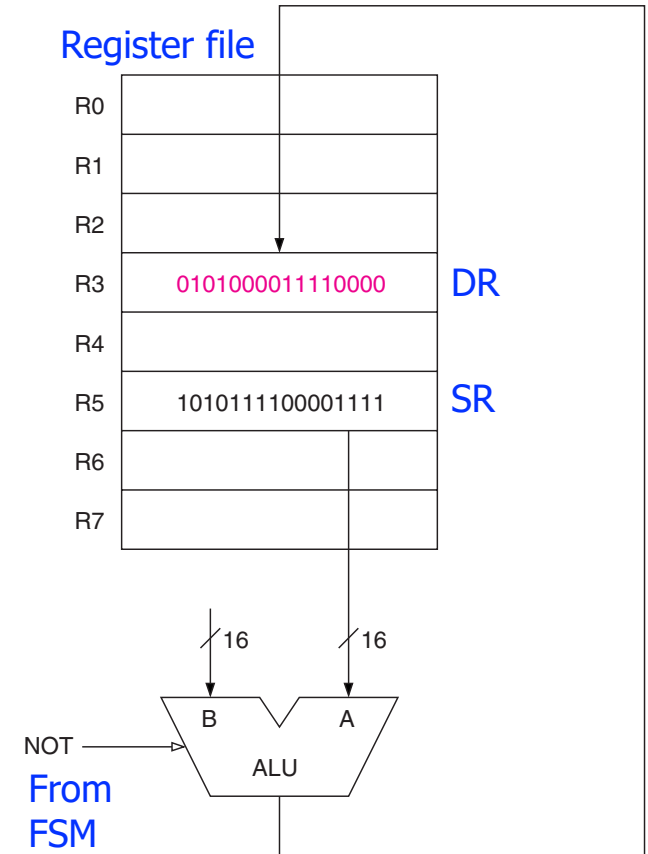
```
NOT R3, R5
```

Field Values

OP	DR	SR	
9	3	5	1 1 1 1 1 1

Machine Code

OP	DR	SR	
1 0 0 1	0 1 1	0 0 1	1 1 1 1 1 1
15	12	11 9	8 6 5 0



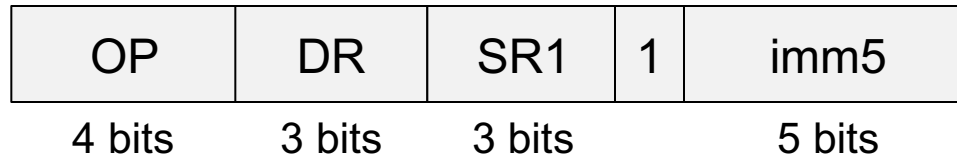
There is **no NOT in MIPS**. How is it implemented?

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
- Subtraction is another necessary operation
 - How is it implemented in LC-3 and MIPS?

Operate Instr. with one Literal in LC-3

■ ADD and AND



- OP = operation
 - E.g., **ADD** = 0001 (same OP as the register-mode ADD)
 - $DR \leftarrow SR1 + \text{sign-extend}(\text{imm5})$
 - E.g., **AND** = 0101 (same OP as the register-mode AND)
 - $DR \leftarrow SR1 \text{ AND } \text{sign-extend}(\text{imm5})$
- 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

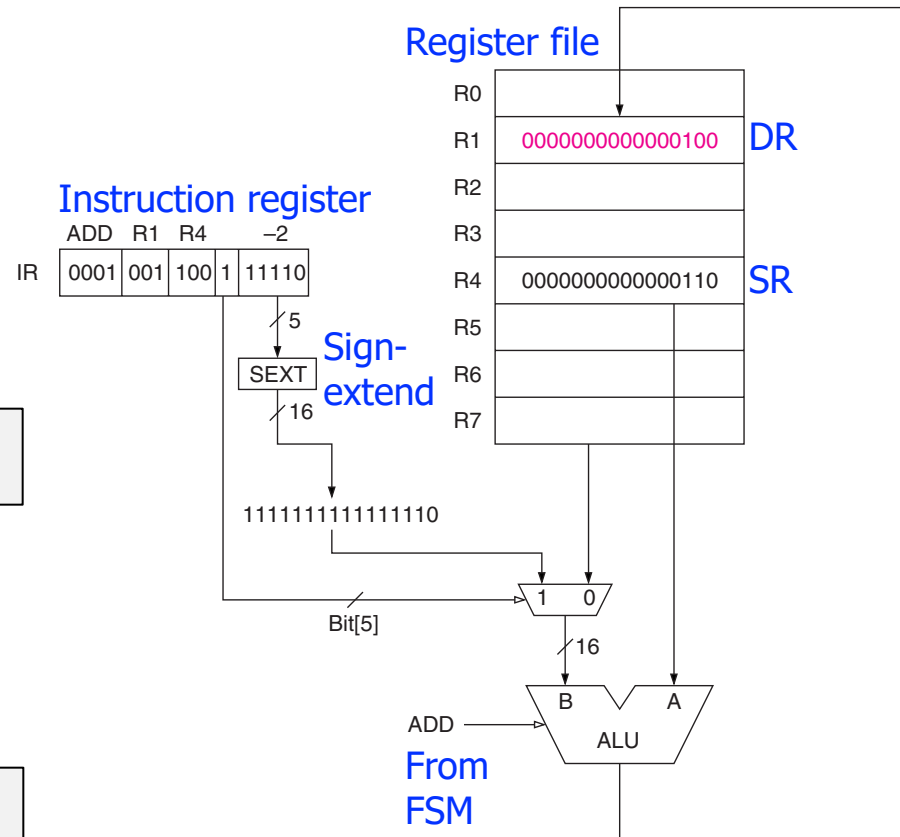
ADD R1, R4, #-2

Field Values

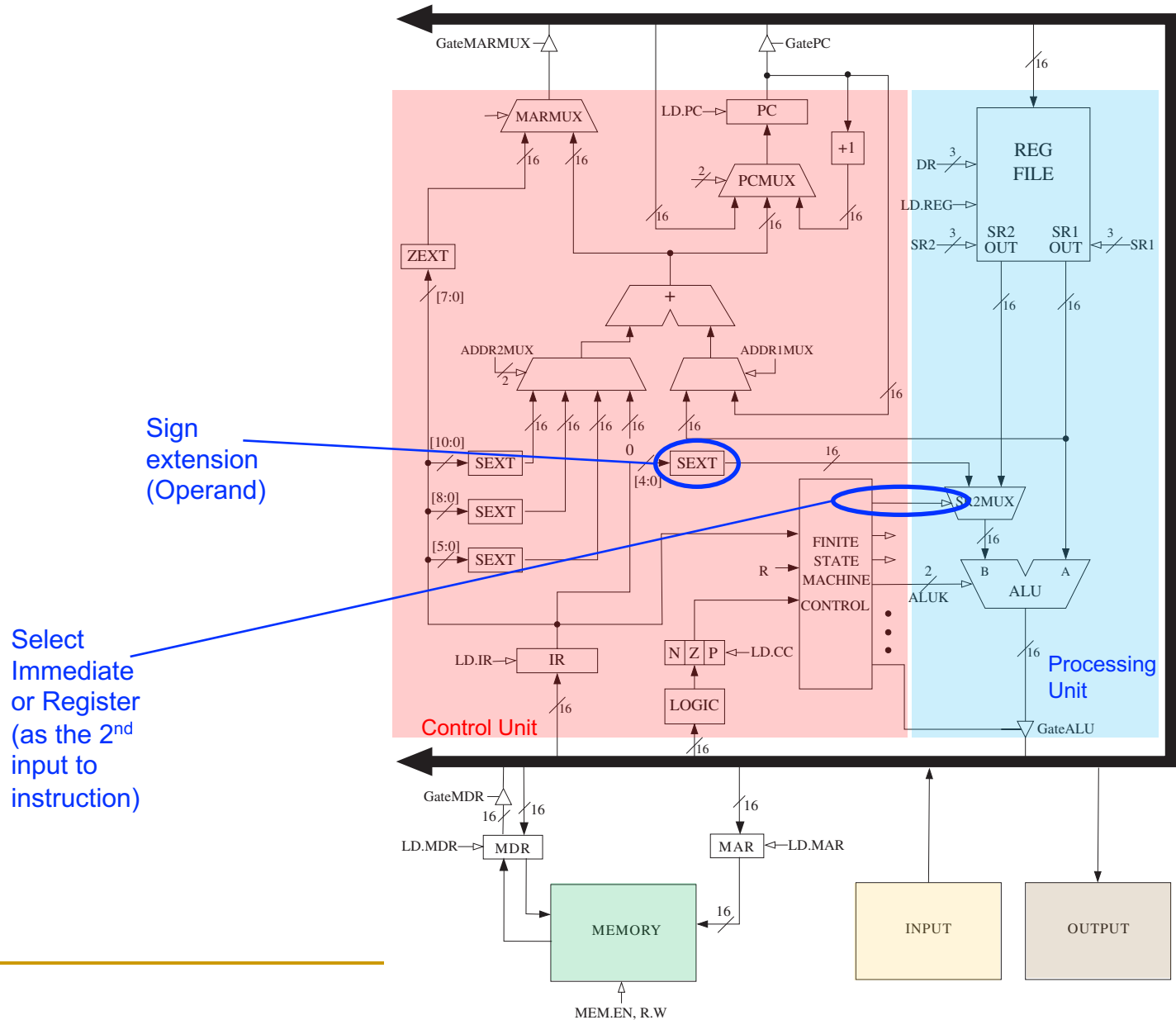
OP	DR	SR		imm5
1	1	4	1	-2

Machine Code

OP				DR		SR		imm5						
0001				001		100		1	11110					
15		12		11		9		8	6		5	4	0	

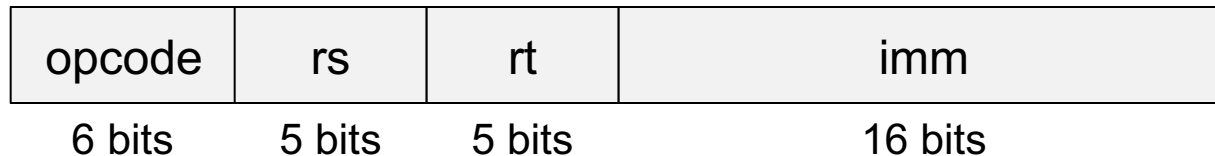


ADD with one Literal in LC-3 Data Path



Instructions with one Literal in MIPS

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



- 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

op	rs	rt	imm
0	17	16	5

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

Machine Code

op	rs	rt	imm
001000	10001	10010	0000 0000 0000 0101

0x22300005

Subtract in 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
```

2's complement of R3

■ Tradeoff in LC-3

- ❑ More instructions
- ❑ But, simpler control logic

Subtract Immediate

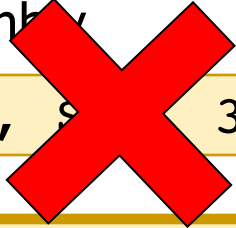
■ MIPS assembly

High-level code

```
a = b - 3;
```

MIPS assembly

```
subi $s1, $s0, 3
```



Is **subi** necessary in MIPS?

MIPS assembly

```
addi $s1, $s0, -3
```

■ LC-3

High-level code

```
a = b - 3;
```

LC-3 assembly

```
ADD R1, R0, #-3
```

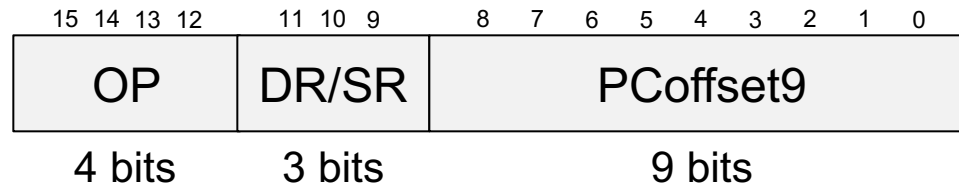
Data Movement Instructions and Addressing Modes

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

[†] This is the incremented PC

LD in LC-3

LD assembly and machine code

LC-3 assembly

LD R2, 0x1AF

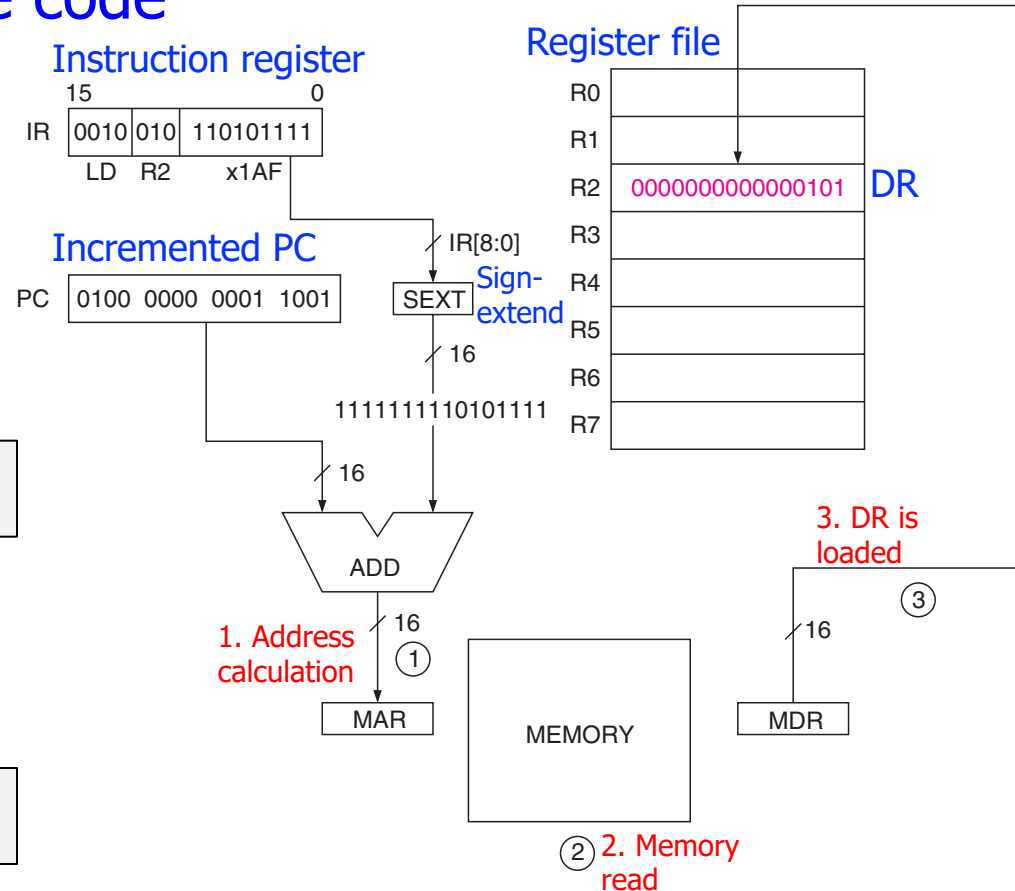
Field Values

OP	DR	PCOffset9
2	2	0x1AF

Machine Code

OP	DR	PCOffset9
0010	010	110101111
15	12	11 9 8 0

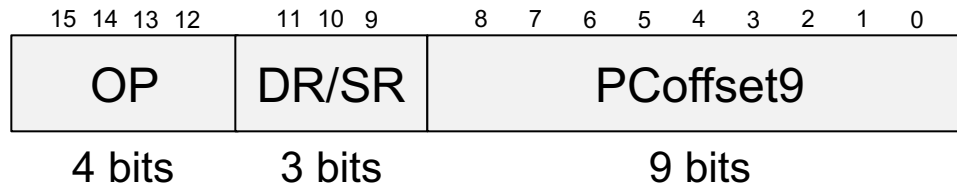
The memory address is **only +255 to -256** locations away of the **LD or ST instruction**



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

Indirect Addressing Mode

■ LDI (Load Indirect) and STI (Store Indirect)



- OP = opcode
 - E.g., LDI = 1010
 - E.g., STI = 1011
- DR = destination register in LDI
- SR = source register in STI
- LDI: $DR \leftarrow \text{Memory}[\text{Memory}[PC^{\dagger} + \text{sign-extend}(\text{PCOffset9})]]$
- STI: $\text{Memory}[\text{Memory}[PC^{\dagger} + \text{sign-extend}(\text{PCOffset9})]] \leftarrow SR$

[†] This is the incremented PC

LDI in LC-3

LDI assembly and machine code

LC-3 assembly

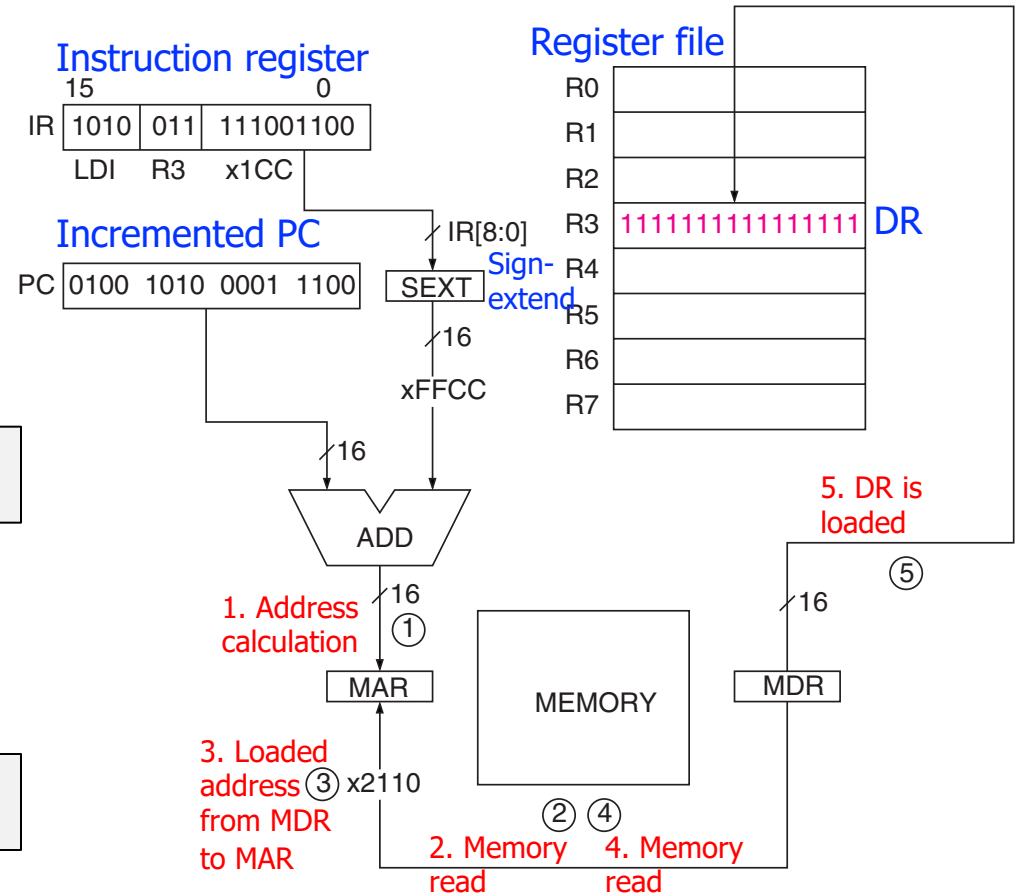
```
LDI R3, 0x1CC
```

Field Values

OP	DR	PCOffset9
A	3	0x1CC

Machine Code

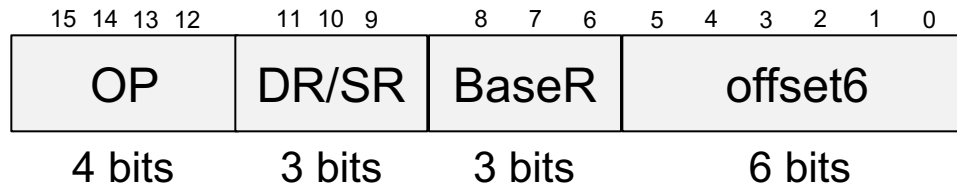
OP	DR	PCOffset9
1 0 1 0	0 1 1	1 1 1 0 0 1 1 0 0
15	12	11 9 8 0



Now the address of the operand can be anywhere in the memory

Base+Offset Addressing Mode

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



- OP = opcode
 - E.g., LDR = 0110
 - E.g., STR = 0111
- DR = destination register in LDR
- SR = source register in STR
- LDR: $DR \leftarrow \text{Memory}[\text{BaseR} + \text{sign-extend}(\text{offset6})]$
- STR: $\text{Memory}[\text{BaseR} + \text{sign-extend}(\text{offset6})] \leftarrow SR$

LDR in LC-3

LDR assembly and machine code

LC-3 assembly

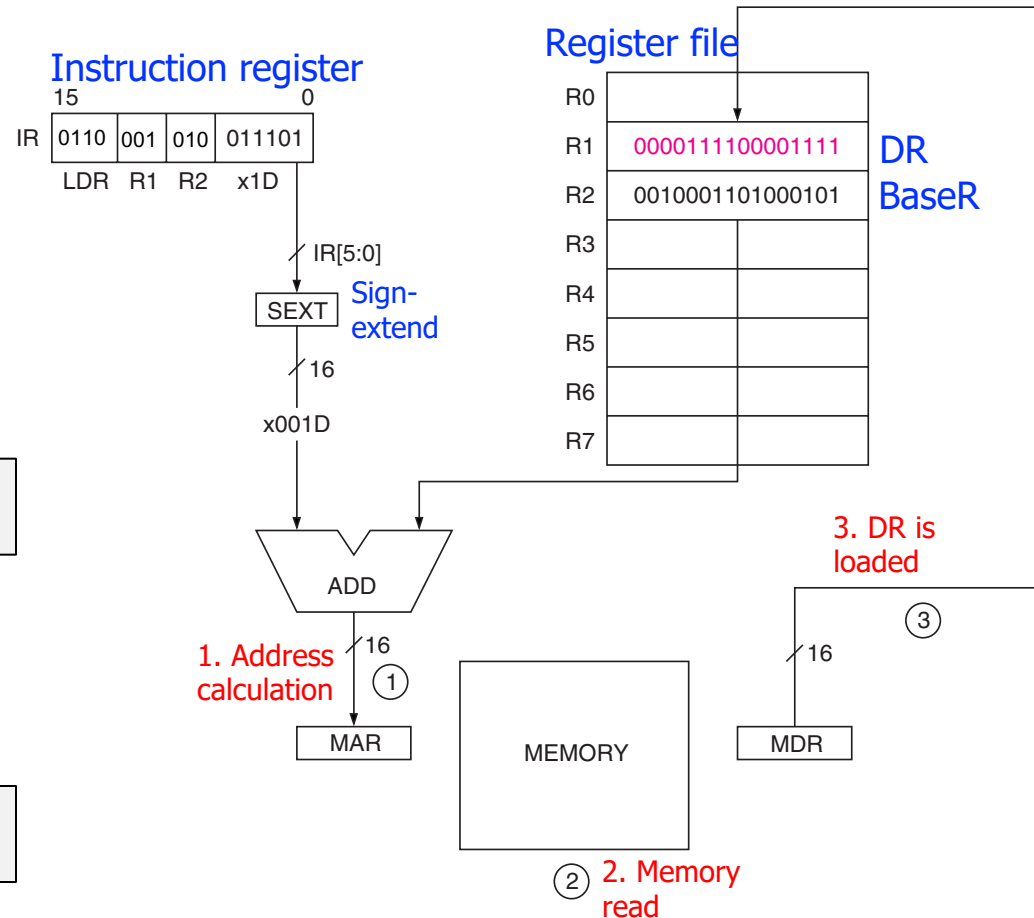
```
LDR R1, R2, 0x1D
```

Field Values

OP	DR	BaseR	offset6
6	1	2	0x1D

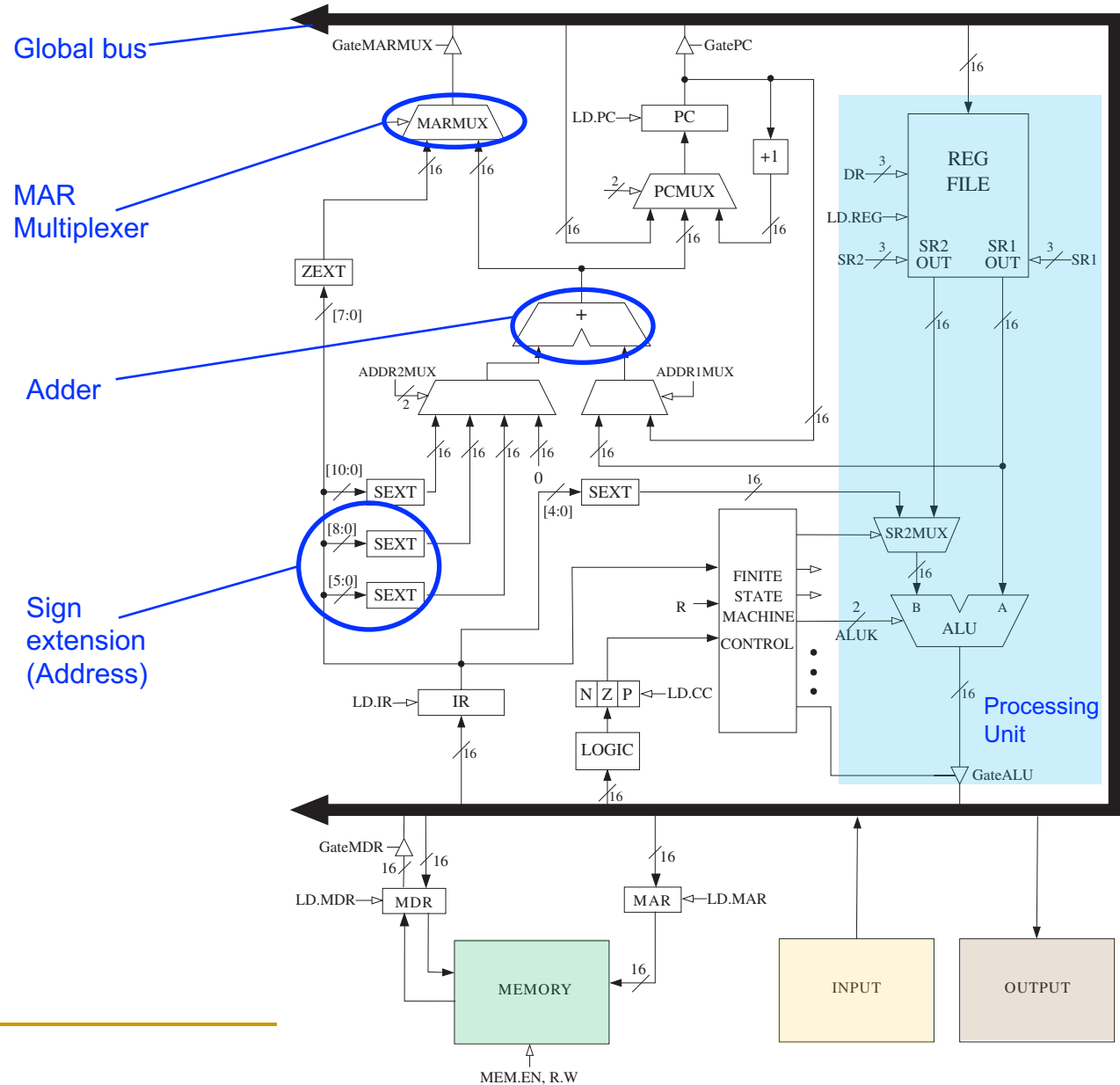
Machine Code

OP				DR		BaseR		offset6							
0 1 1 0				0 0 1		0 1 0		0 1 1 1 0 1							
15		12		11		9		8		6		5		0	



Again, the address of the operand can be **anywhere in the memory**

Address Calculation in LC-3 Data Path



Base+Offset Addressing Mode in MIPS

- In MIPS, **lw** and **sw** use base+offset mode (or **base addressing mode**)

High-level code

```
A[ 2 ] = a;
```

MIPS assembly

```
sw    $s3, 8( $s0 )
```

Memory[\$s0 + 8] ← \$s3

Field Values

op	rs	rt	imm
43	16	19	8

- **imm** is the 16-bit offset, which is **sign-extended to 32 bits**

An Example Program in MIPS and LC-3

High-level code

```
a      = A[0];  
c      = a + b - 5;  
B[0]   = c;
```

MIPS registers

```
A = $s0  
b = $s2  
B = $s1
```

LC-3 registers

```
A = R0  
b = R2  
B = R1
```

MIPS assembly

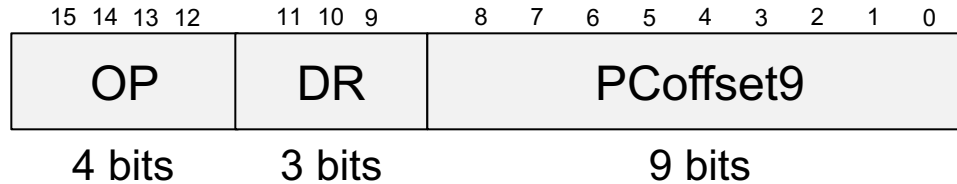
```
lw    $t0, 0($s0)  
add   $t1, $t0, $s2  
addi  $t2, $t1, -5  
sw    $t2, 0($s1)
```

LC-3 assembly

```
LDR   R5, R0, #0  
ADD   R6, R5, R2  
ADD   R7, R6, #-5  
STR   R7, R1, #0
```

Immediate Addressing Mode

■ LEA (Load Effective Address)



- OP = 1110
- DR = destination register
- LEA: $DR \leftarrow PC^{\dagger} + \text{sign-extend}(\text{PCoffset9})$

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*

[†] This is the incremented PC

LEA in LC-3

LEA assembly and machine code

LC-3 assembly

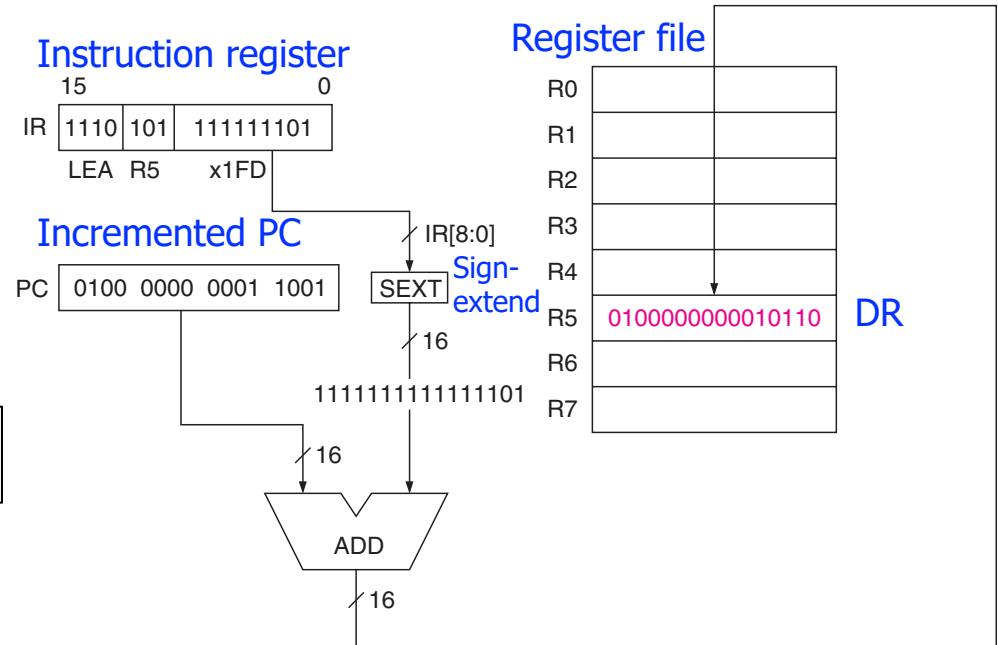
```
LEA R5, #-3
```

Field Values

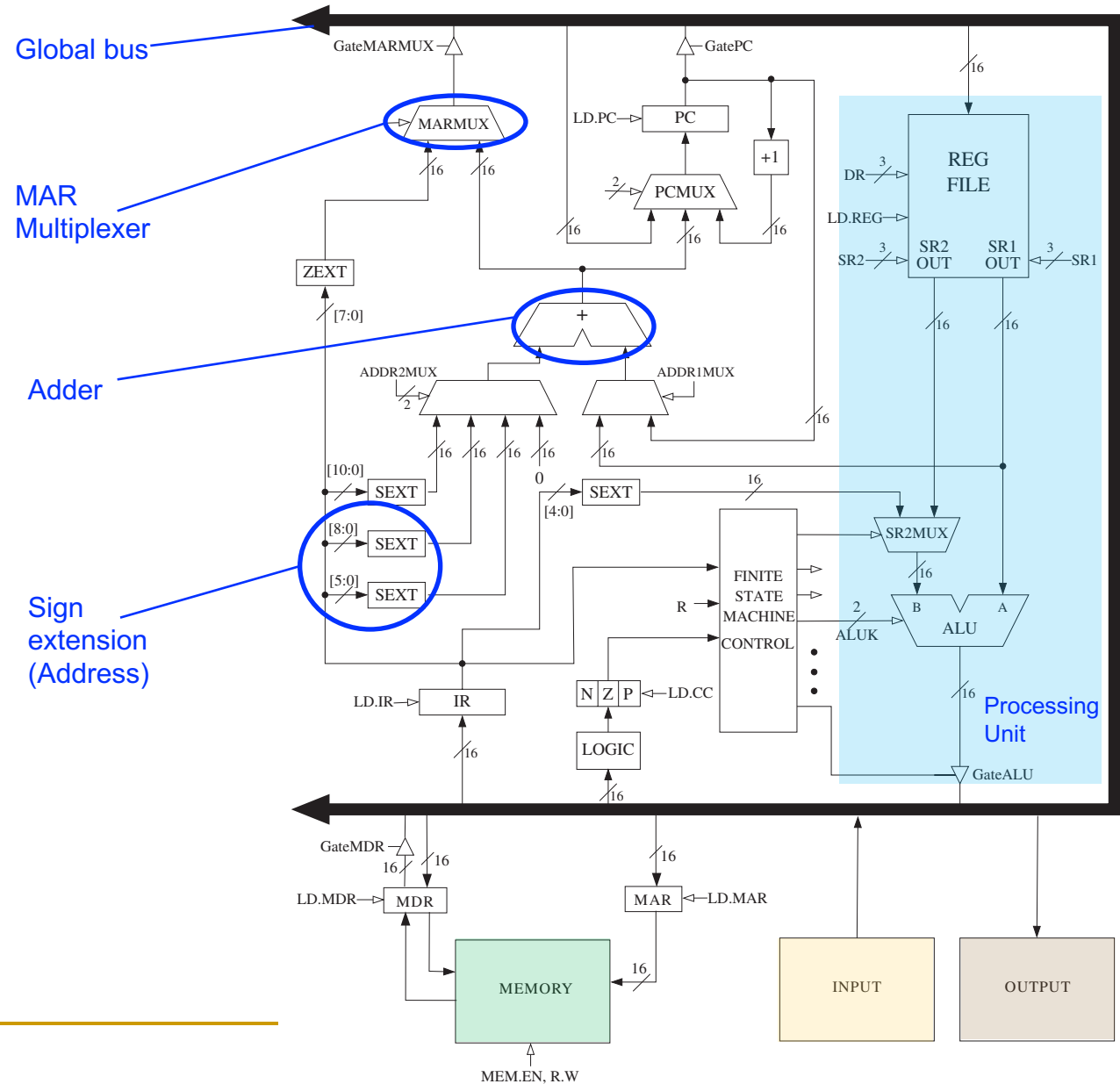
OP	DR	PCOffset9
E	5	0x1FD

Machine Code

OP	DR	PCOffset9
1 1 1 0	1 0 1	1 1 1 1 1 1 1 0 1
15	12	11 9 8 0



Address Calculation in LC-3 Data Path



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	1	0	1	R1 ← PC - 3
x30F7	0	0	0	1	0	1	0	0	0	1	1	0	1	1	1	0	R2 ← R1 + 14
x30F8	0	0	1	1	0	1	0	1	1	1	1	1	1	0	1	1	M[x30F4] ← R2
x30F9	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 ← 0
x30FA	0	0	0	1	0	1	0	0	1	0	1	0	0	1	0	1	R2 ← R2 + 5
x30FB	0	1	1	1	0	1	0	0	0	1	0	0	1	1	1	0	M[R1 + 14] ← R2
x30FC	1	0	1	0	0	1	1	1	1	1	1	1	0	1	1	1	R3 ← M[M[x30F4]]

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	3	1	1	1	1	1	1	1	0	$R1 = PC - 3 = 0x30F7 - 3 = 0x30F4$
x30F7	0	0	0	1	0	1	0	0	0	1	1	0	1	1	1	0	$R2 = R1 + 14 = 0x30F4 + 14 = 0x3102$
x30F8	0	0	1	1	0	1	0	5	1	1	1	1	1	0	1	0	$M[PC - 5] = M[0x030F4] = 0x3102$
x30F9	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	$R2 = 0$
x30FA	0	0	0	1	0	1	0	0	1	0	1	5	0	1	0	1	$R2 = R2 + 5 = 5$
x30FB	0	1	1	1	0	1	0	0	0	1	0	0	0	0	0	0	$M[R1 + 14] = M[0x30F4 + 14] = M[0x3102] = 5$
x30FC	1	0	1	0	0	1	1	9	1	1	1	1	0	1	1	0	$R3 = M[M[PC - 9]] = M[M[0x30FD - 9]] = M[M[0x30F4]] = M[0x3102] = 5$

- The final value of R3 is 5

Control Flow Instructions

Control Flow Instructions

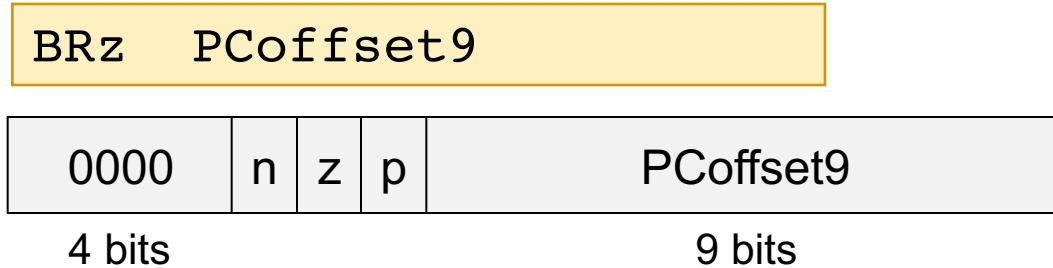
- Allow a program to execute **out of sequence**
- Conditional branches and unconditional jumps
 - **Conditional branches** are used to **make decisions**
 - E.g., if-else statement
 - In LC-3, three **condition codes** are used
 - **Jumps** are used to implement
 - **Loops**
 - **Function calls**
 - **JMP** in LC-3 and **j** in MIPS

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)



- ❑ $n, z, p =$ **which condition code is tested** (N, Z, and/or P)
 - n, z, p : instruction bits to identify the condition codes to be tested
 - N, Z, P : values of the corresponding condition codes
- ❑ $PCoffset9 =$ immediate or constant value
- ❑ if $((n \text{ AND } N) \text{ OR } (p \text{ AND } P) \text{ OR } (z \text{ AND } Z))$
 - then $PC \leftarrow PC^\dagger + \text{sign-extend}(PCoffset9)$
- ❑ Variations: BRn, BRz, BRp, BRzp, BRnp, BRnz, BRnzp

[†] This is the incremented PC

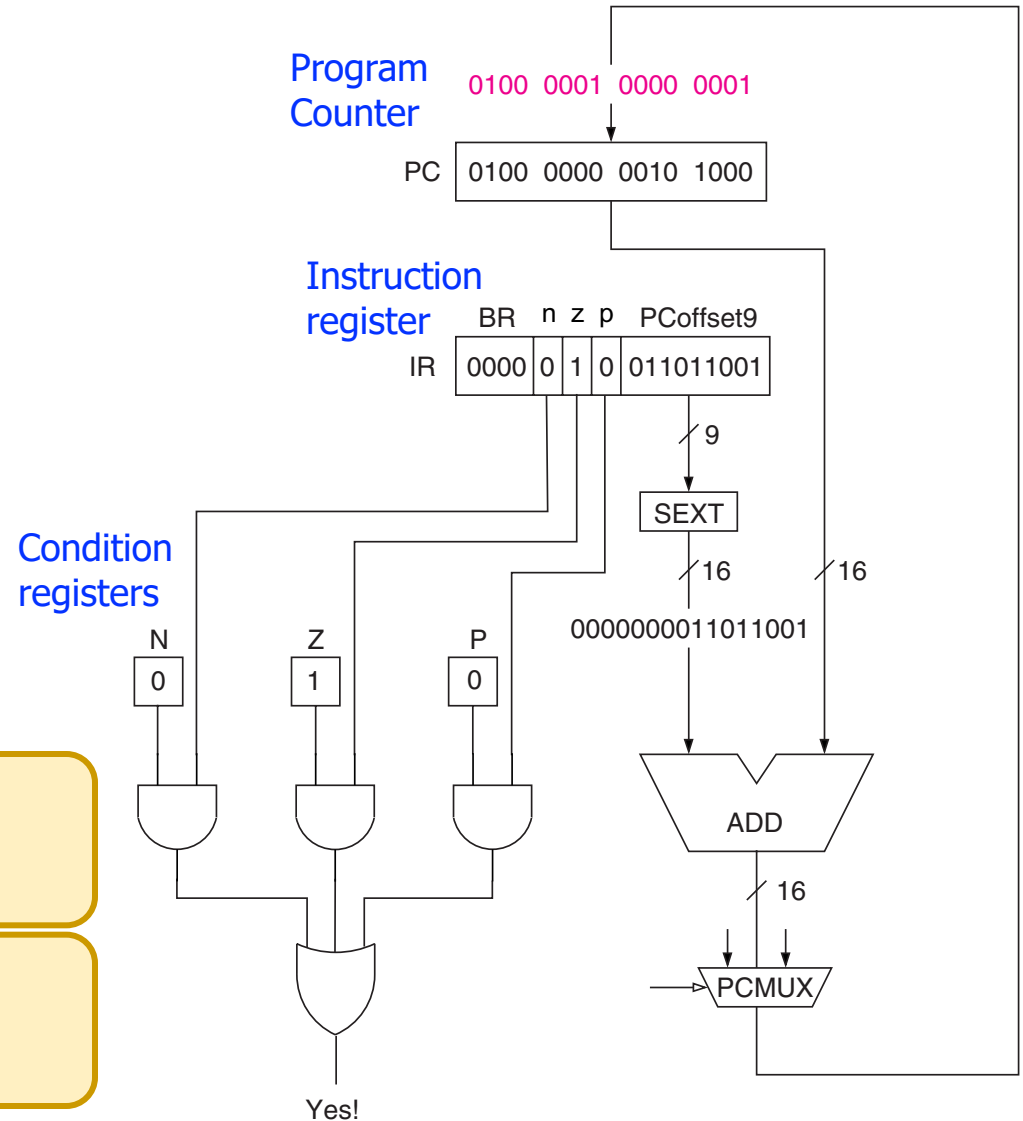
Conditional Branches in LC-3

■ BRz

BRz 0x0D9

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

And what if $n = z = p = 0$?

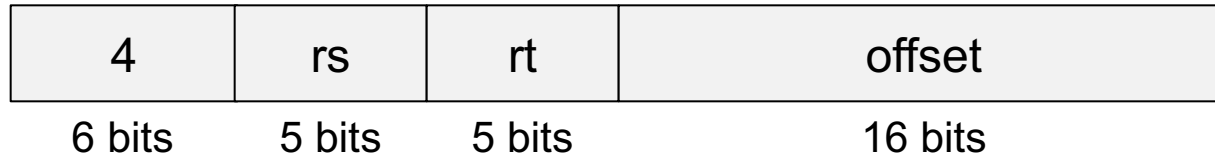


*n, z, p are the instruction bits to identify the condition codes to be tested

Conditional Branches in MIPS

■ beq (Branch if Equal)

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



- 4 = opcode
- rs, rt = source registers
- offset = immediate or constant value
- if $rs == rt$
 - then $PC \leftarrow PC^{\dagger} + \text{sign-extend}(\text{offset}) * 4$
- Variations: beq, bne, blez, bgtz

[†] This is the incremented PC

Branch If Equal in MIPS and LC-3

MIPS assembly

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

LC-3 assembly

```
NOT  R2, R1
```

```
ADD  R3, R2, #1
```

```
ADD  R4, R3, R0
```

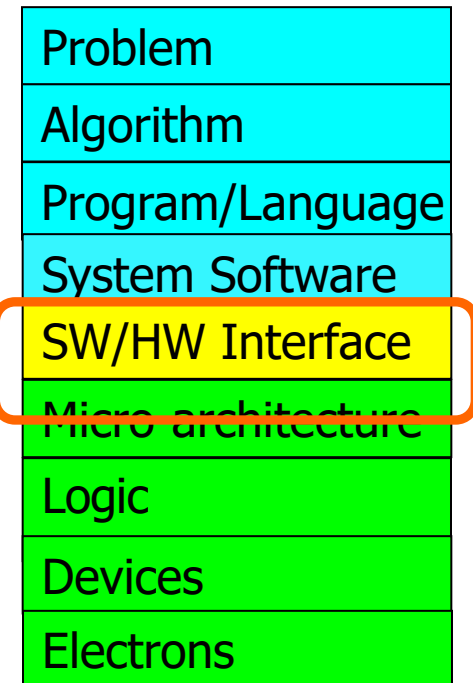
```
BRz  offset
```

**Subtract
(R0 - R1)**

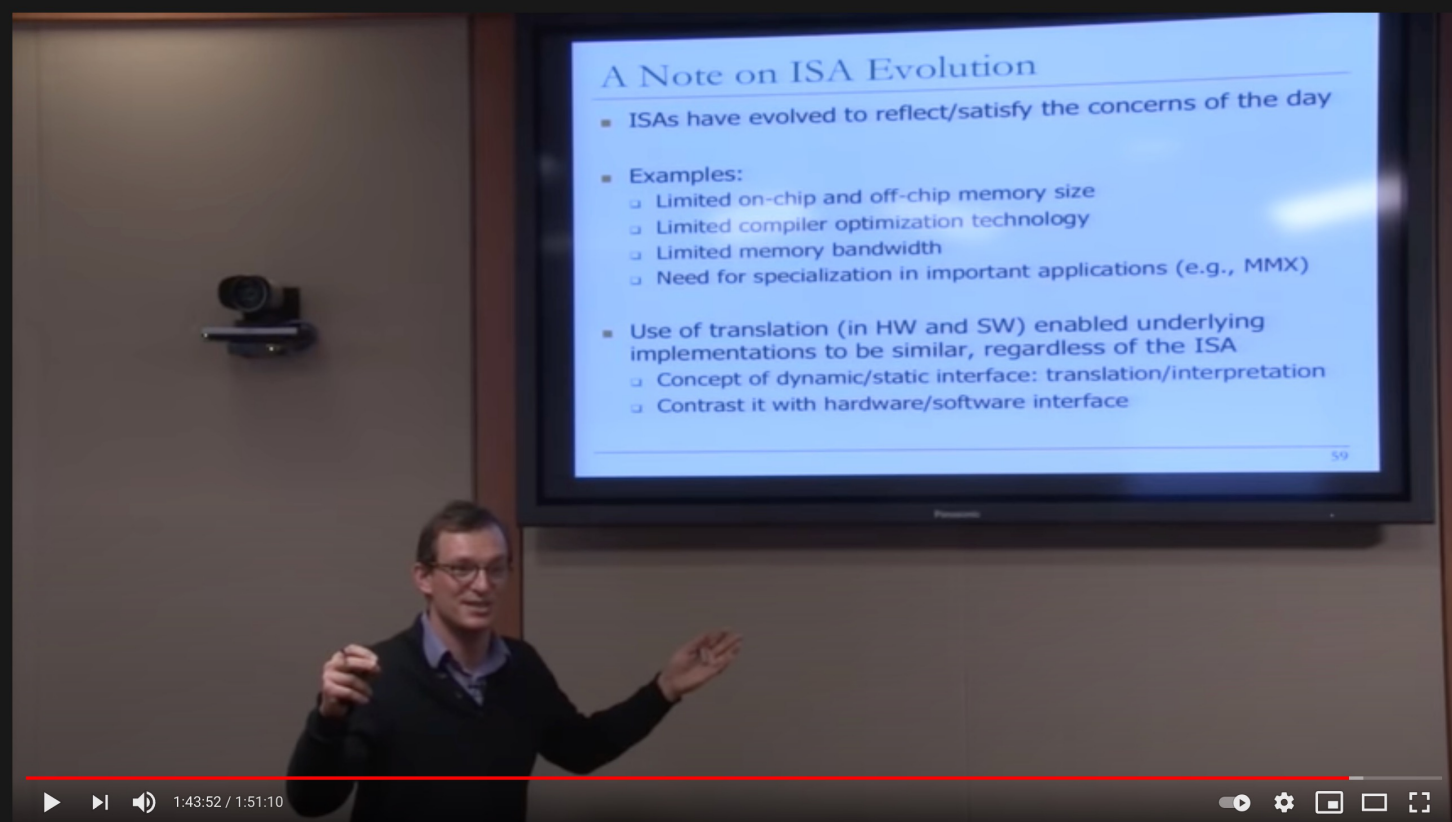
- This is an example of **tradeoff** in the instruction set
 - ❑ The same functionality requires **more instructions in LC-3**
 - ❑ But, the **control logic** requires **more complexity in MIPS**

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



A Note on ISA Evolution

- ISAs have evolved to reflect/satisfy the concerns of the day
- Examples:
 - Limited on-chip and off-chip memory size
 - Limited compiler optimization technology
 - Limited memory bandwidth
 - Need for specialization in important applications (e.g., MMX)
- Use of translation (in HW and SW) enabled underlying implementations to be similar, regardless of the ISA
 - Concept of dynamic/static interface: translation/interpretation
 - Contrast it with hardware/software interface

59

1:43:52 / 1:51:10

Lecture 3. ISA Tradeoffs - Carnegie Mellon - Computer Architecture 2015 - Onur Mutlu

44,973 views • Jan 24, 2015

Carnegie Mellon Computer Architecture
22.8K subscribers

Lecture 3. ISA Tradeoffs
Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)
Date: Jan 16th, 2015

276 5 SHARE SAVE ...

ANALYTICS EDIT VIDEO

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 the instructions

Complex vs. Simple Instructions

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

■ Advantages of Complex instructions

- + **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

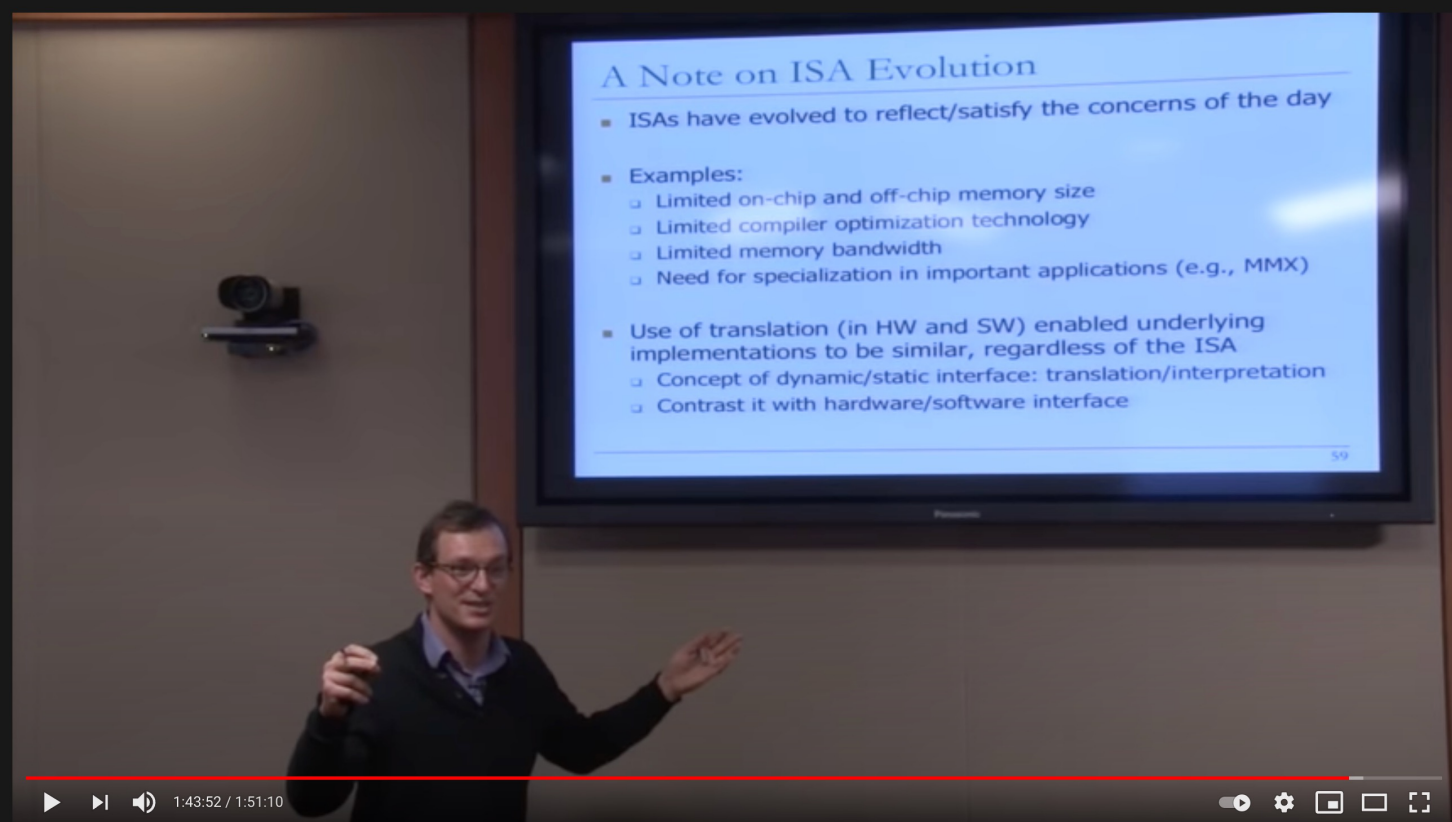
■ Disadvantages of Complex Instructions

- **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

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 → fewer saves/restores
 - Larger instruction size
 - Larger register file size

There Is A Lot More to Cover on ISAs



A Note on ISA Evolution

- ISAs have evolved to reflect/satisfy the concerns of the day
- Examples:
 - Limited on-chip and off-chip memory size
 - Limited compiler optimization technology
 - Limited memory bandwidth
 - Need for specialization in important applications (e.g., MMX)
- Use of translation (in HW and SW) enabled underlying implementations to be similar, regardless of the ISA
 - Concept of dynamic/static interface: translation/interpretation
 - Contrast it with hardware/software interface

59

Lecture 3. ISA Tradeoffs - Carnegie Mellon - Computer Architecture 2015 - Onur Mutlu

44,973 views • Jan 24, 2015

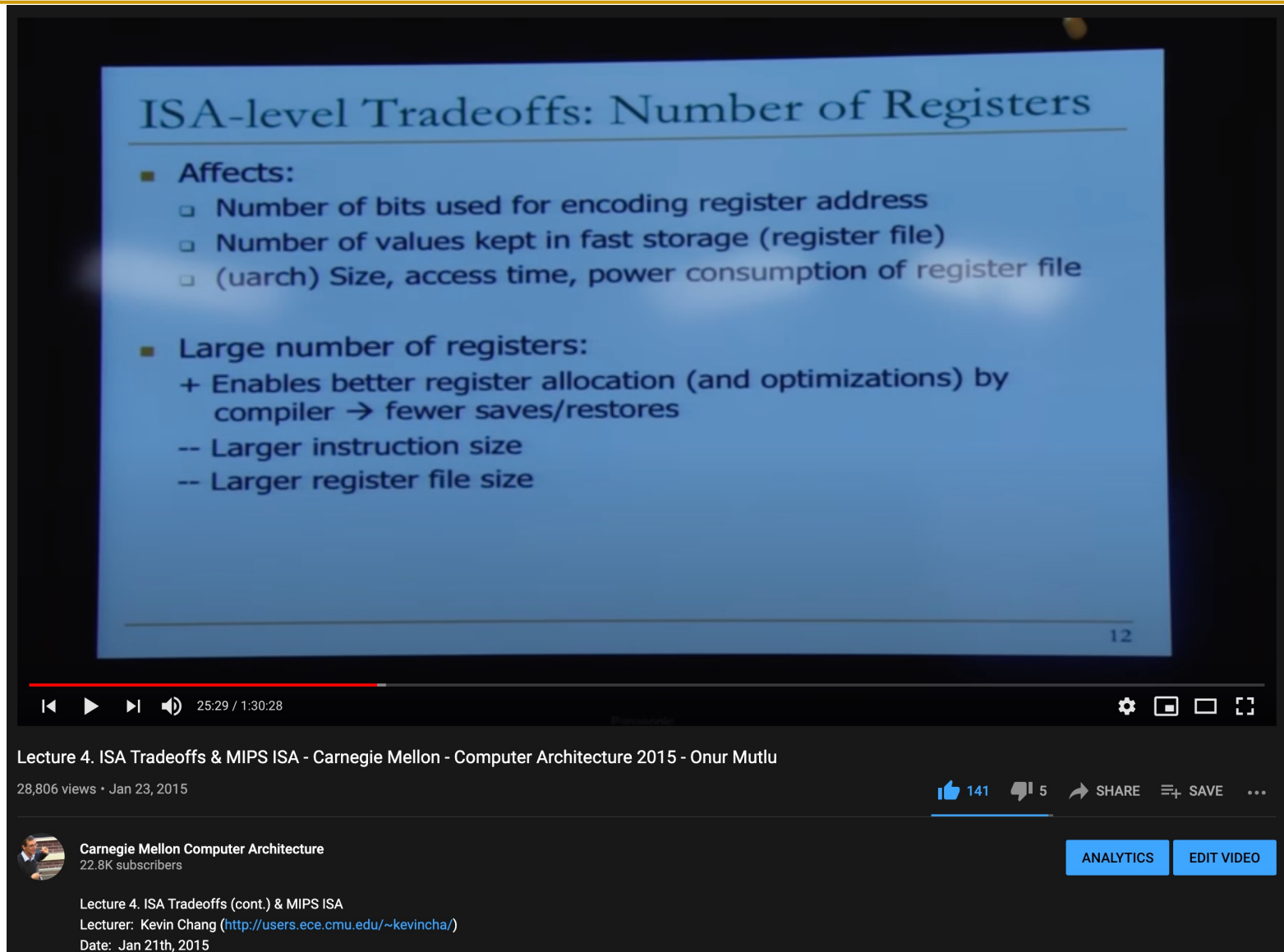
Carnegie Mellon Computer Architecture
22.8K subscribers

Lecture 3. ISA Tradeoffs
Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)
Date: Jan 16th, 2015

276 5 SHARE SAVE ...

ANALYTICS EDIT VIDEO

There Is A Lot More to Cover on ISAs



The video player displays a slide with the following content:

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 → fewer saves/restores
 - Larger instruction size
 - Larger register file size

12

Lecture 4. ISA Tradeoffs & MIPS ISA - Carnegie Mellon - Computer Architecture 2015 - Onur Mutlu

28,806 views • Jan 23, 2015

141 5 SHARE SAVE ...

 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

ANALYTICS EDIT VIDEO

Detailed Lectures on ISAs & ISA Tradeoffs

■ Computer Architecture, Spring 2015, Lecture 3

- ISA Tradeoffs (CMU, Spring 2015)
- <https://www.youtube.com/watch?v=QKdiZSfwg-g&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>