

Introduction to Computer Architecture

Lecture 7: Von Neumann Model, ISA, LC-3 and MIPS

Pooyan Jamshidi



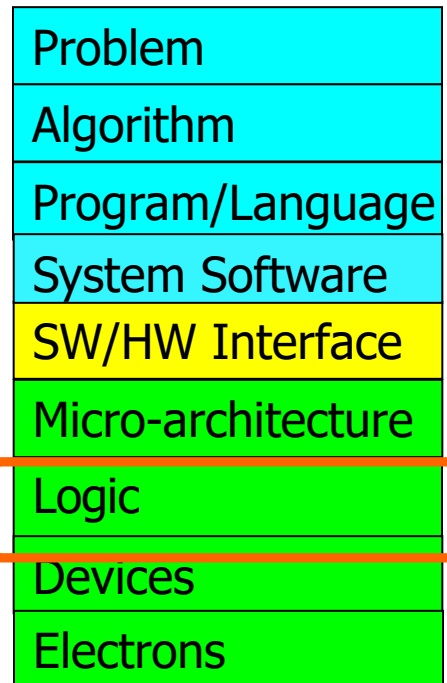
**Engineering
and Computing**

UNIVERSITY OF SOUTH CAROLINA

What Have We Learned So Far?

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

Date	Lecture	Readings
01/9 & 01/11	Lecture #1 : L1: Introduction and Basics [slides]	• D. Harris and S. Harris, Digital Design and Computer Architecture. Chapters 1 - 5
01/16	Lecture #2 : L2: Tradeoffs, Metrics, Mindset [slides]	• D. Harris and S. Harris, Digital Design and Computer Architecture. Chapters 1 - 5
01/18 & 01/23	Lecture #3 : L3: Combinational Logic Design [slides]	• D. Harris and S. Harris, Digital Design and Computer Architecture. Chapters 1 - 5
01/30 - 02/08	Lecture #4 : L4: Sequential Logic Design [slides]	• D. Harris and S. Harris, Digital Design and Computer Architecture. Chapters 1 - 5
02/13 - 02/15	Lecture #5 : L5: Hardware Description Languages and Verilog [slides]	• D. Harris and S. Harris, Digital Design and Computer Architecture. Chapters 1 - 5
02/27 - 02/29	Lecture #6 : L6: Timing and Verification [slides]	• D. Harris and S. Harris, Digital Design and Computer Architecture. Chapters 1 - 5



Agenda for Today & Next Few Lectures

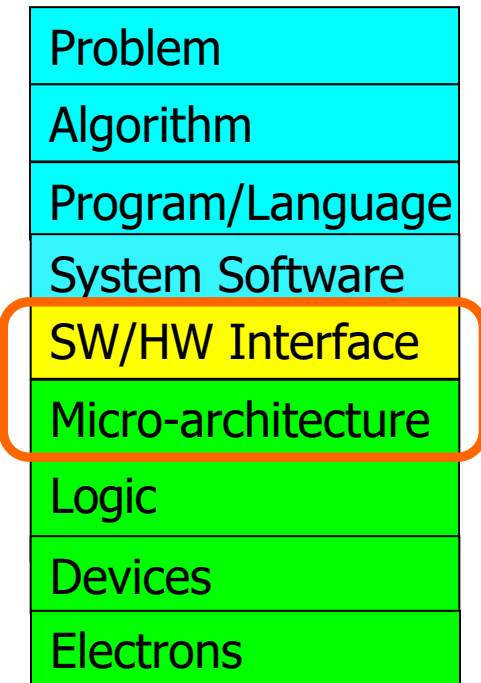
- The von Neumann model
- LC-3: An example of von Neumann machine

- LC-3 and MIPS Instruction Set Architectures

- LC-3 and MIPS assembly and programming

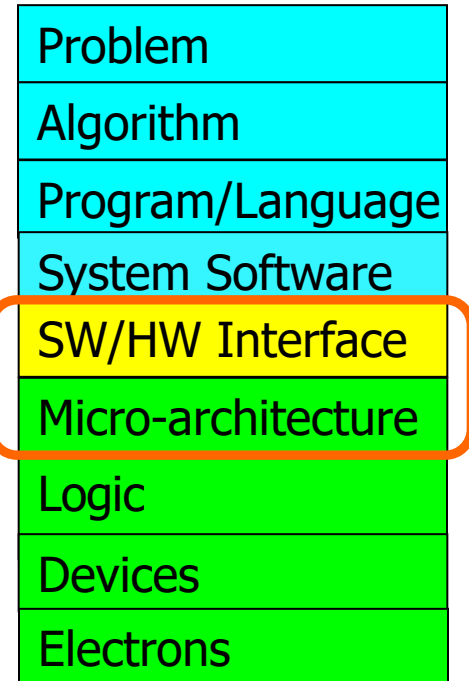
- Introduction to microarchitecture and single-cycle microarchitecture

- 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
 - 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
- **Recommended:** H&H Chapter 6, especially 6.1, 6.2, 6.4, 6.5

■ Next lecture

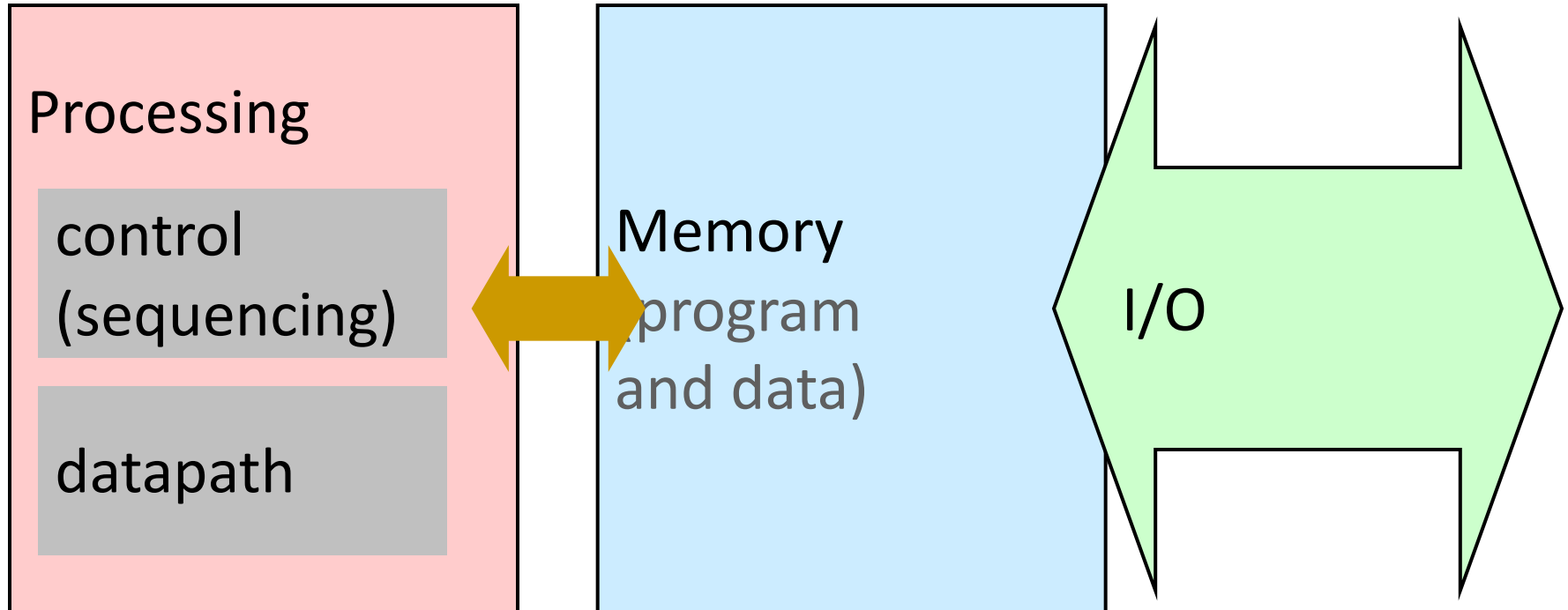
- Introduction to microarchitecture and single-cycle microarchitecture
 - H&H, Chapter 7.1-7.3
 - P&P, Appendices A and C
- Multi-cycle microarchitecture
 - H&H, Chapter 7.4
 - P&P, Appendices A and C

Building a Computing System

The Von Neumann Model

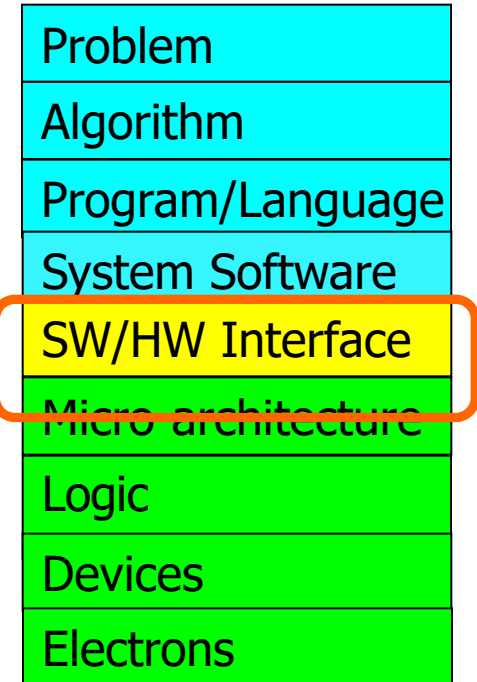
Recall: What is A Computer?

- 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



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

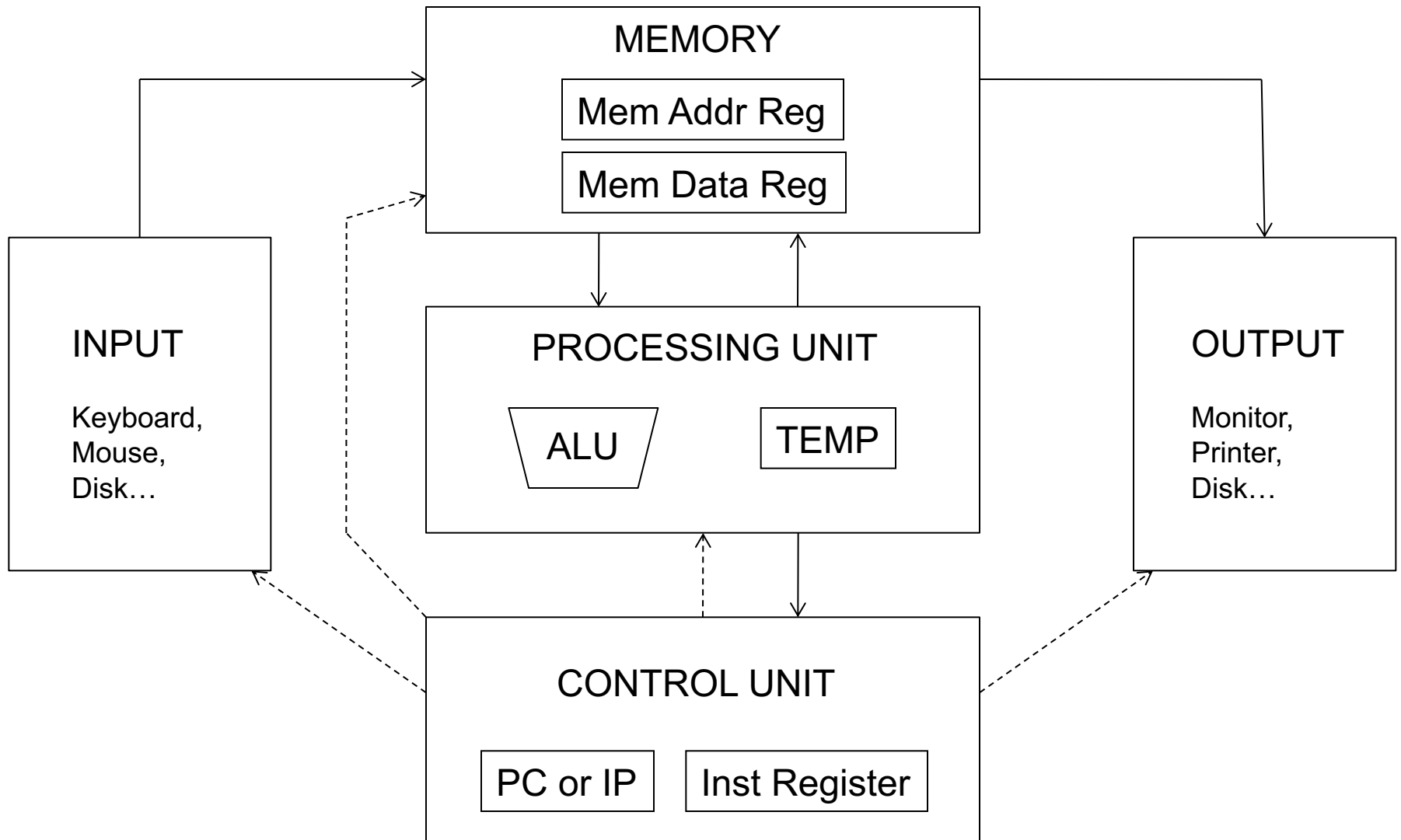
- In order to build a computer, we need an execution model for processing computer programs
- John von Neumann proposed a fundamental model in 1946
- The von Neumann Model consists of 5 components
 - Memory (stores the program and data)
 - Processing unit
 - Input
 - Output
 - Control unit (controls the order in which instructions are carried out)
- Throughout this lecture, we will examine two examples of the von Neumann model
 - LC-3
 - MIPS



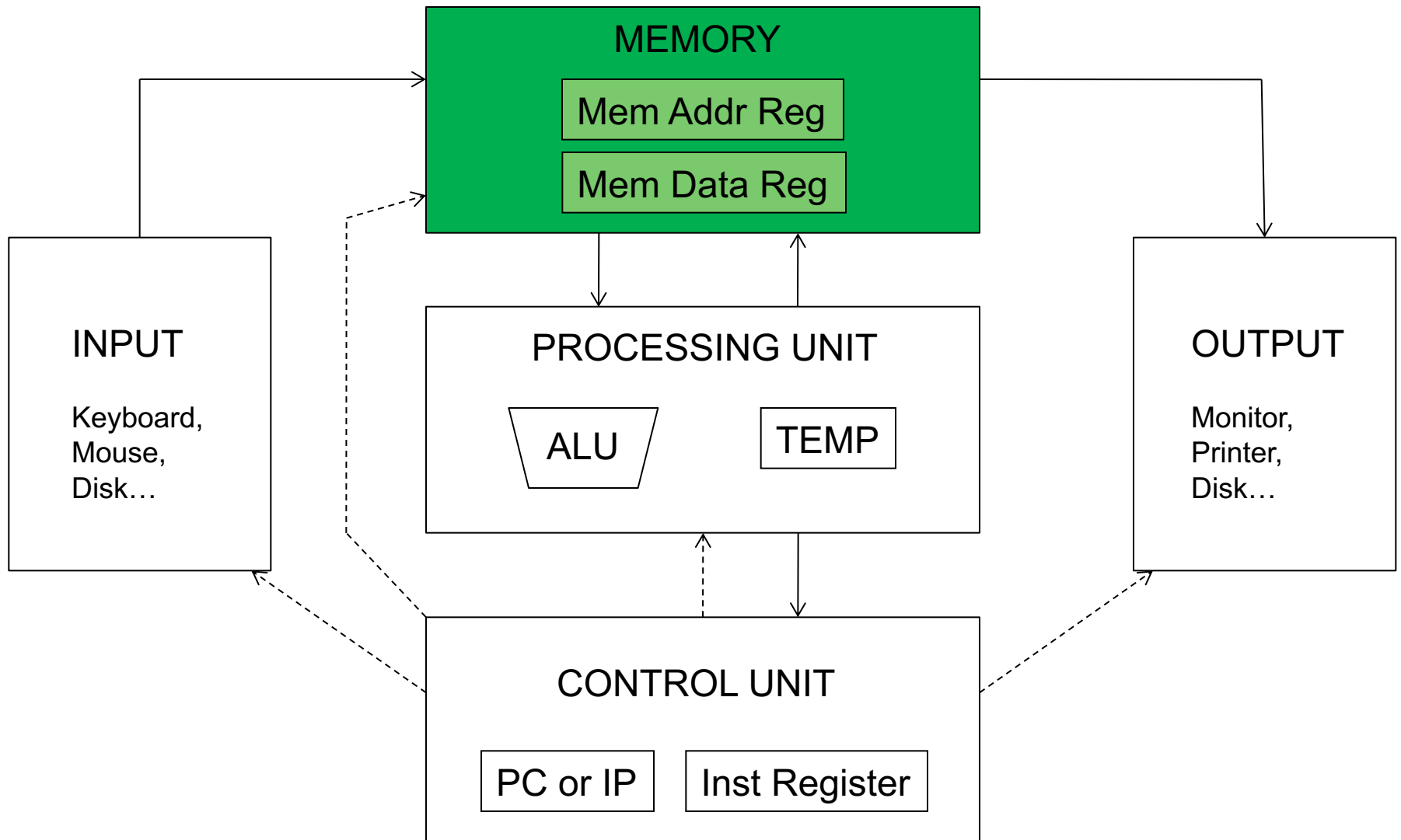
Burks, Goldstein, von Neumann,
“Preliminary discussion of the logical design
of an electronic computing instrument,” 1946.

All general-purpose computers today use the von Neumann model

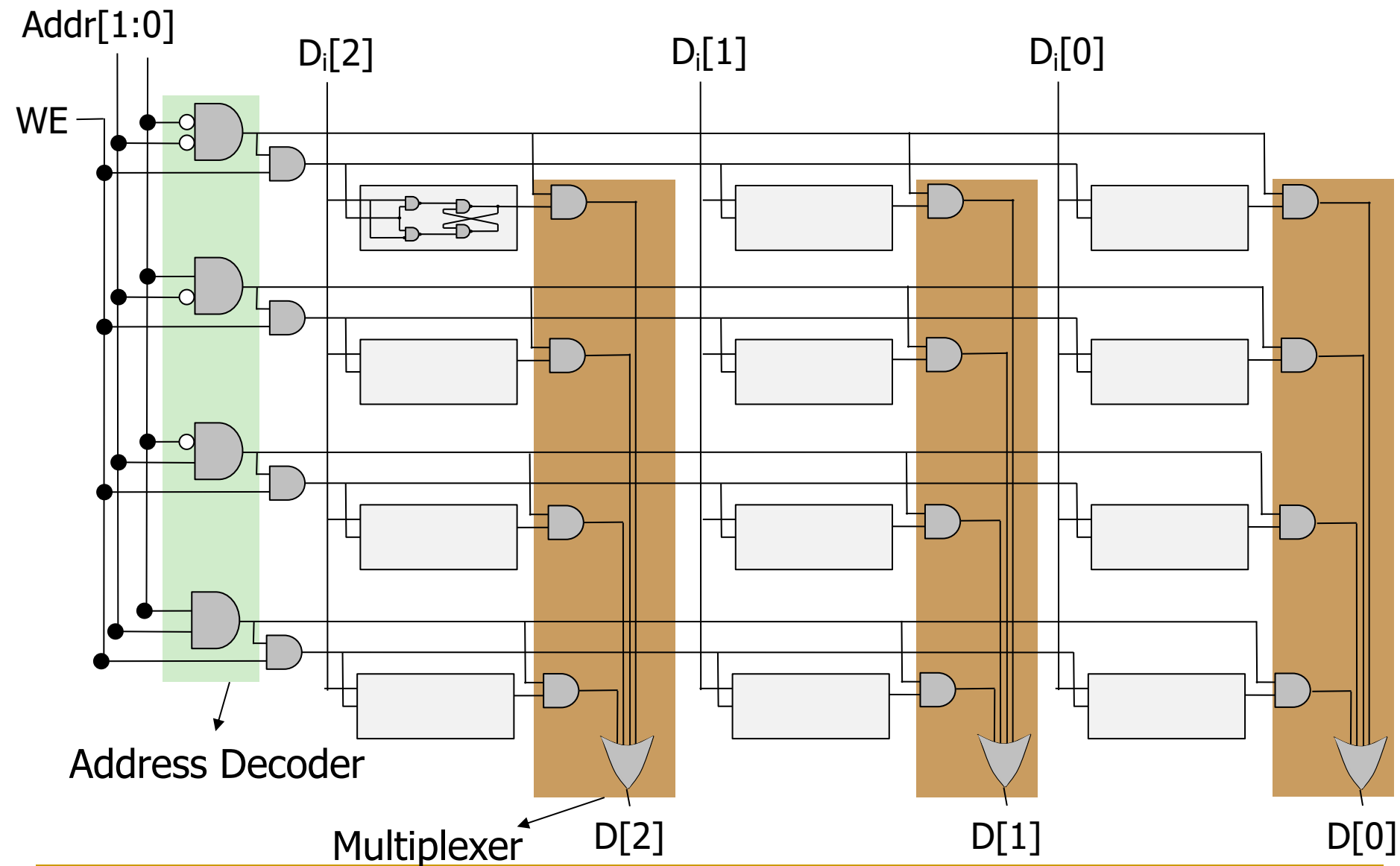
The von Neumann Model



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**

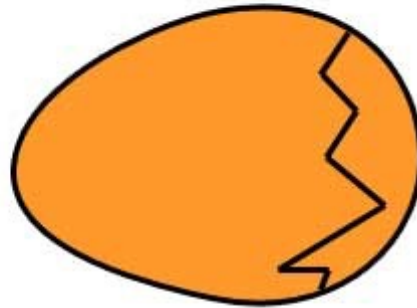
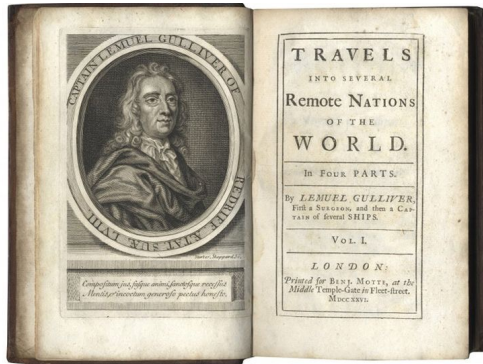
MIPS memory

Byte Address of the Word	Data				
⋮					⋮
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

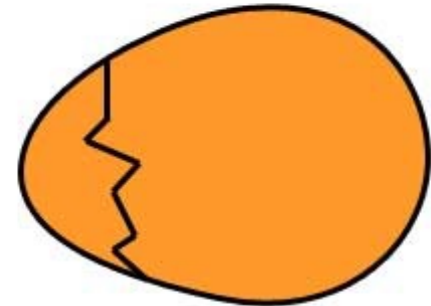
Which of the four bytes is most vs. least significant?

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



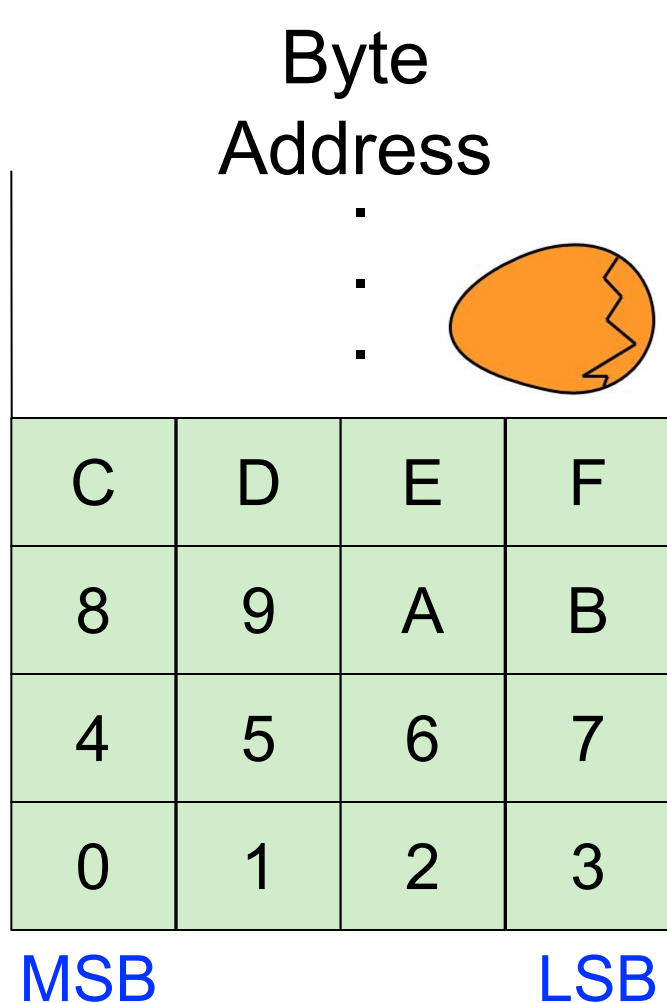
BIG ENDIAN - The way people always broke their eggs in the Lilliput land



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

Big Endian vs. Little Endian

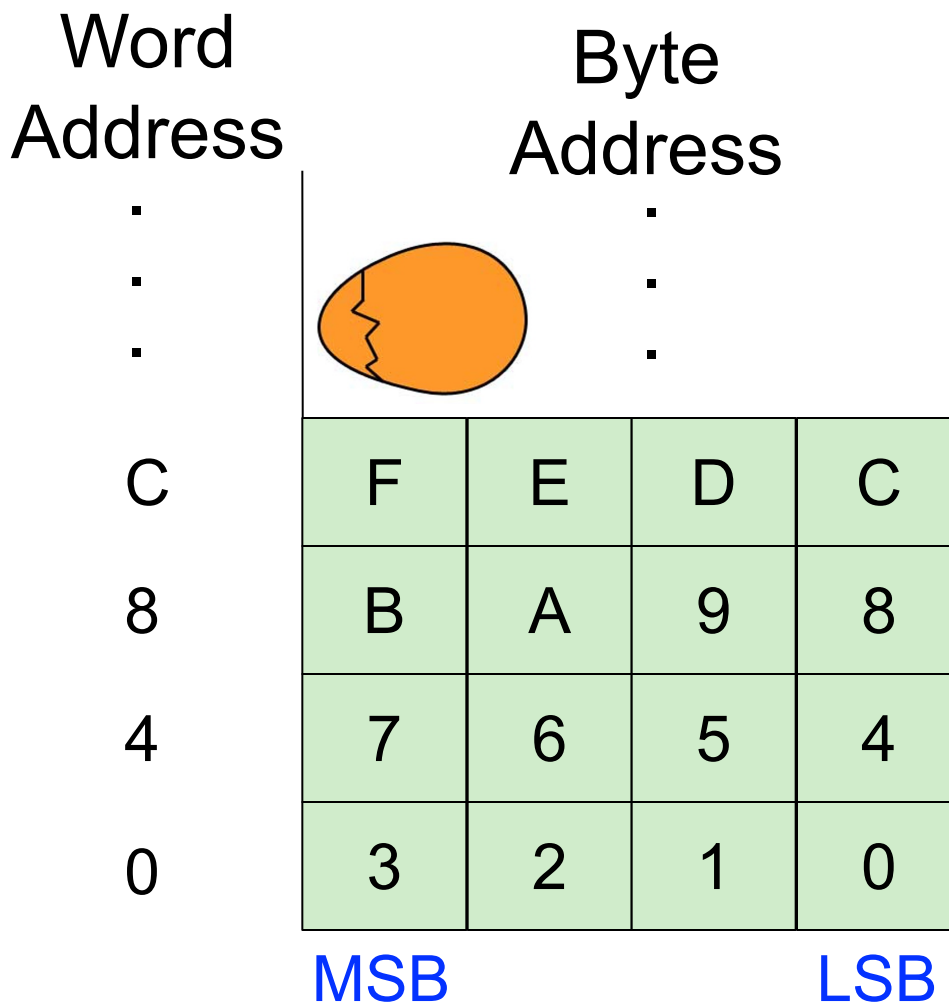
Big Endian



(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)

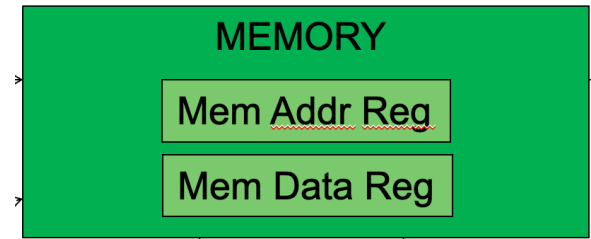
(Least Significant Byte)

LSB in higher byte address

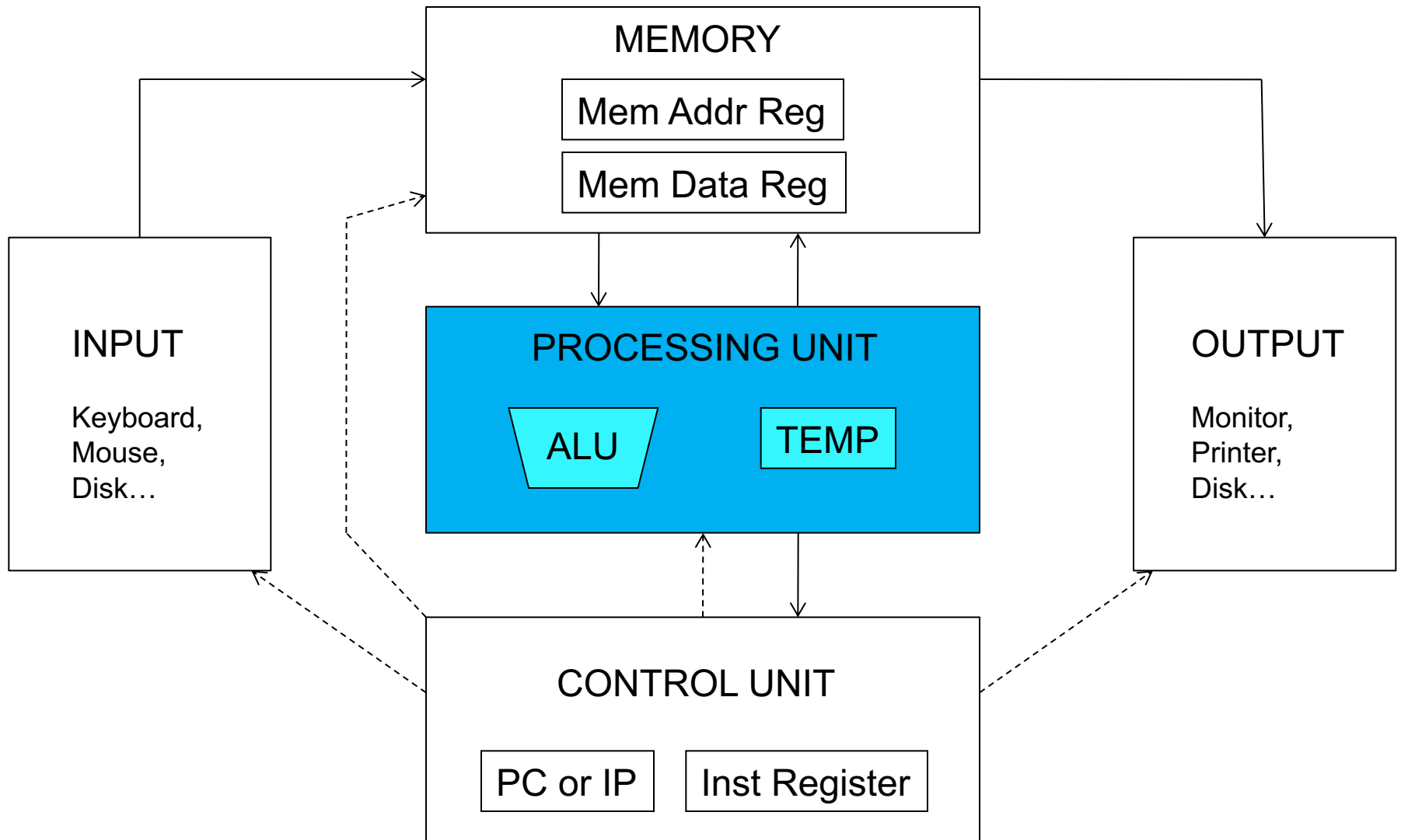
LSB in lower byte address

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:

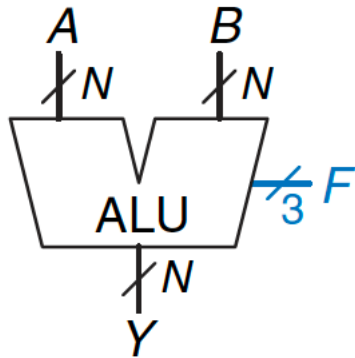


Figure 5.14 ALU symbol

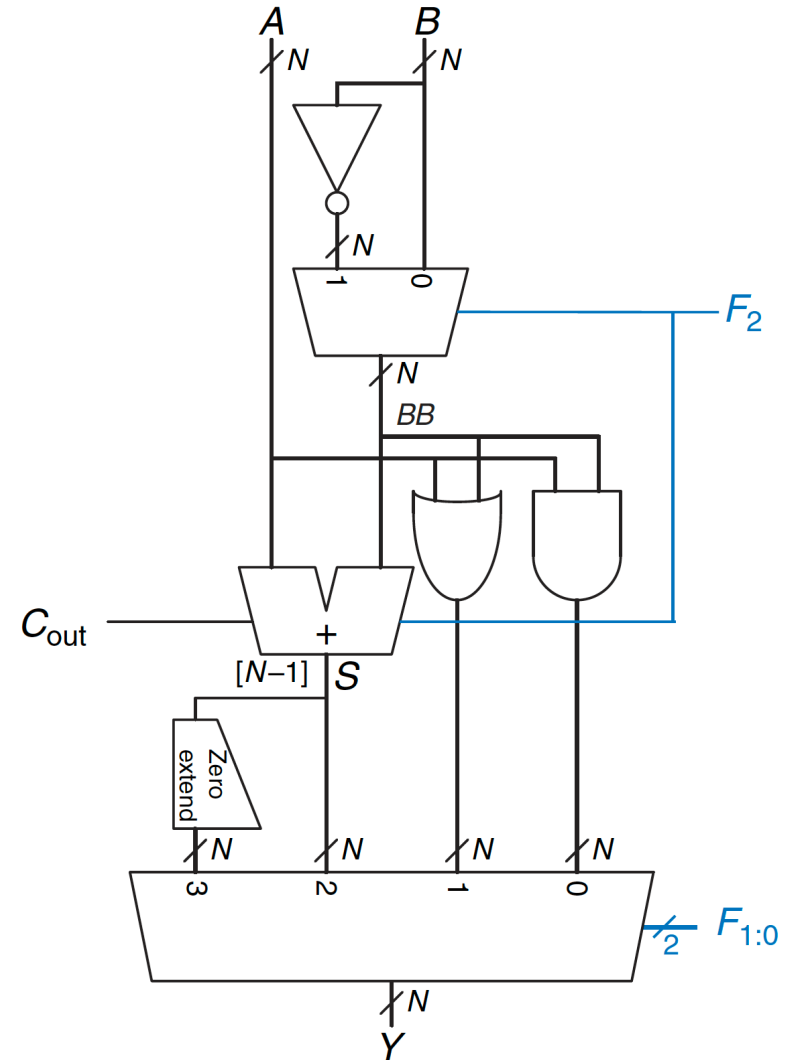
Table 5.1 ALU operations

$F_{2:0}$	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 \bar{B}
101	A OR \bar{B}
110	A - B
111	SLT

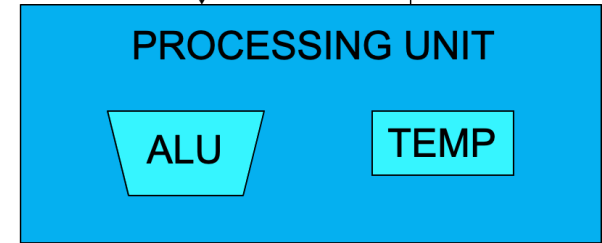


Processing Unit: Fast Temporary Storage

- 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

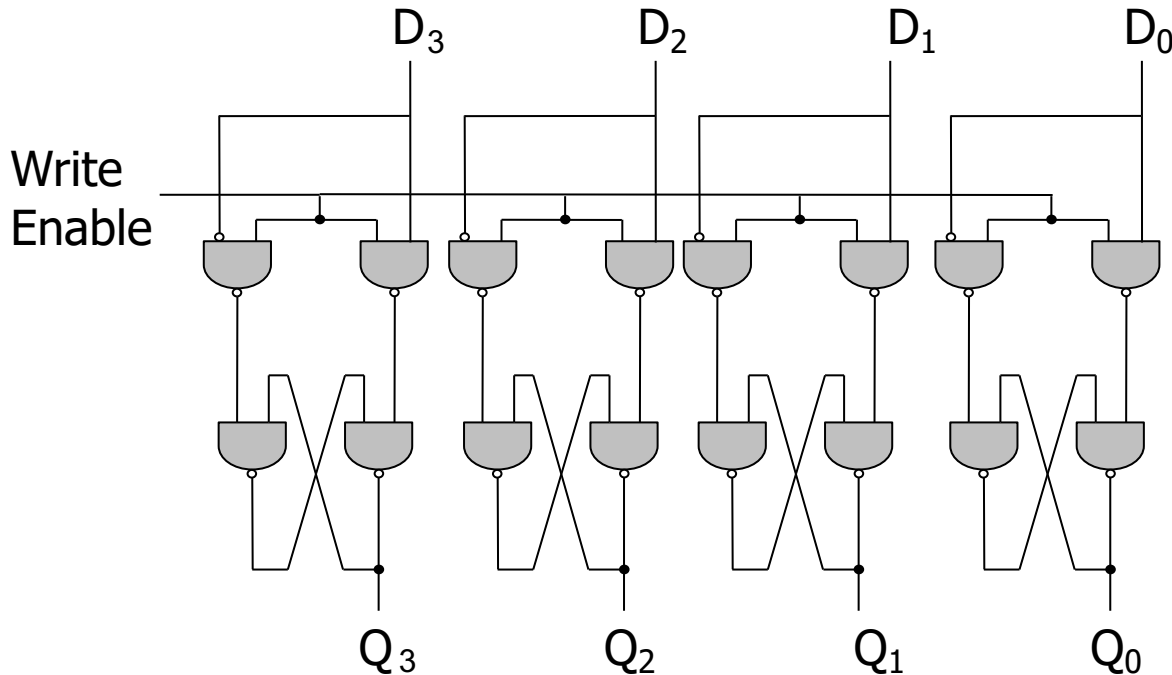
- **Memory** is large but slow
- **Registers** in the Processing Unit
 - Ensure fast access to values to be processed in the ALU
 - Typically one register contains **one word (same as word length)**
- **Register Set or Register File**
 - **Set of registers that can be manipulated by instructions**
 - 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



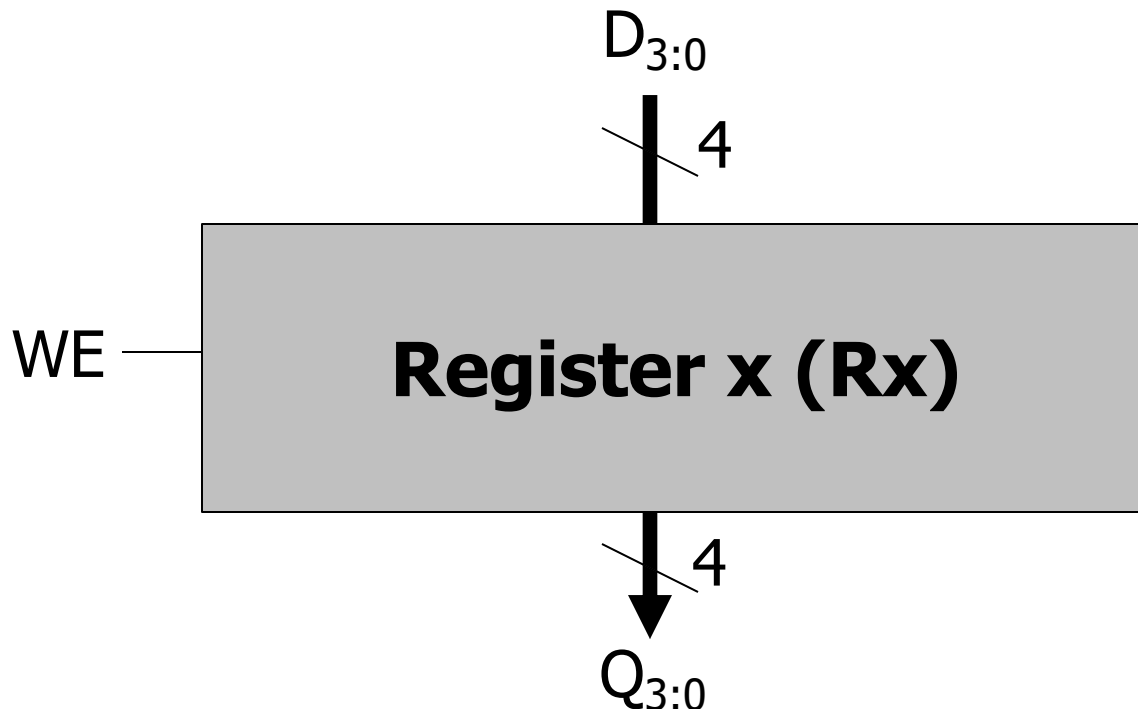
Here we have a **register**, or a structure that stores more than one bit and can be read from and written to

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

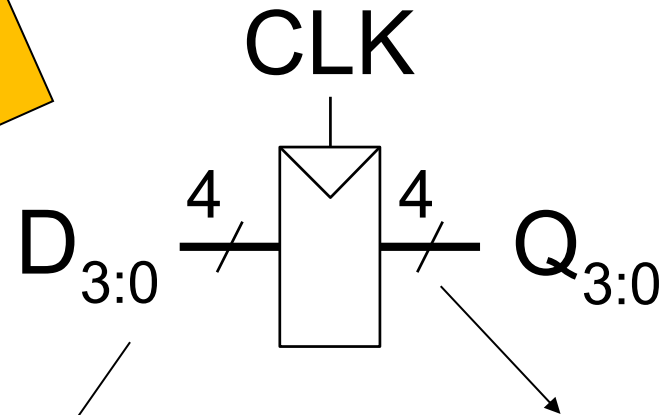
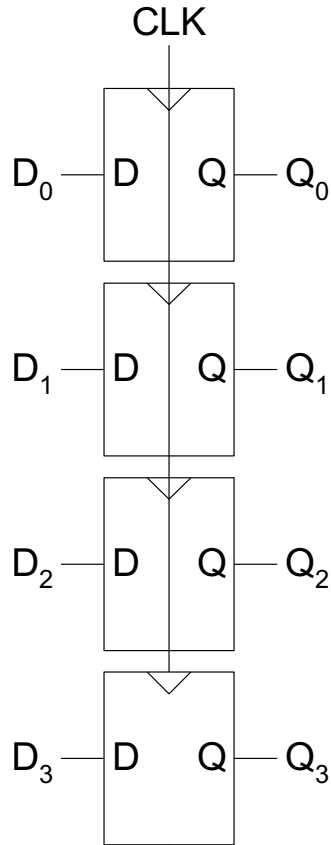


Here we have a **register**, or a structure that stores more than one bit and can be read from and written to

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

Recall: D Flip-Flop Based Register

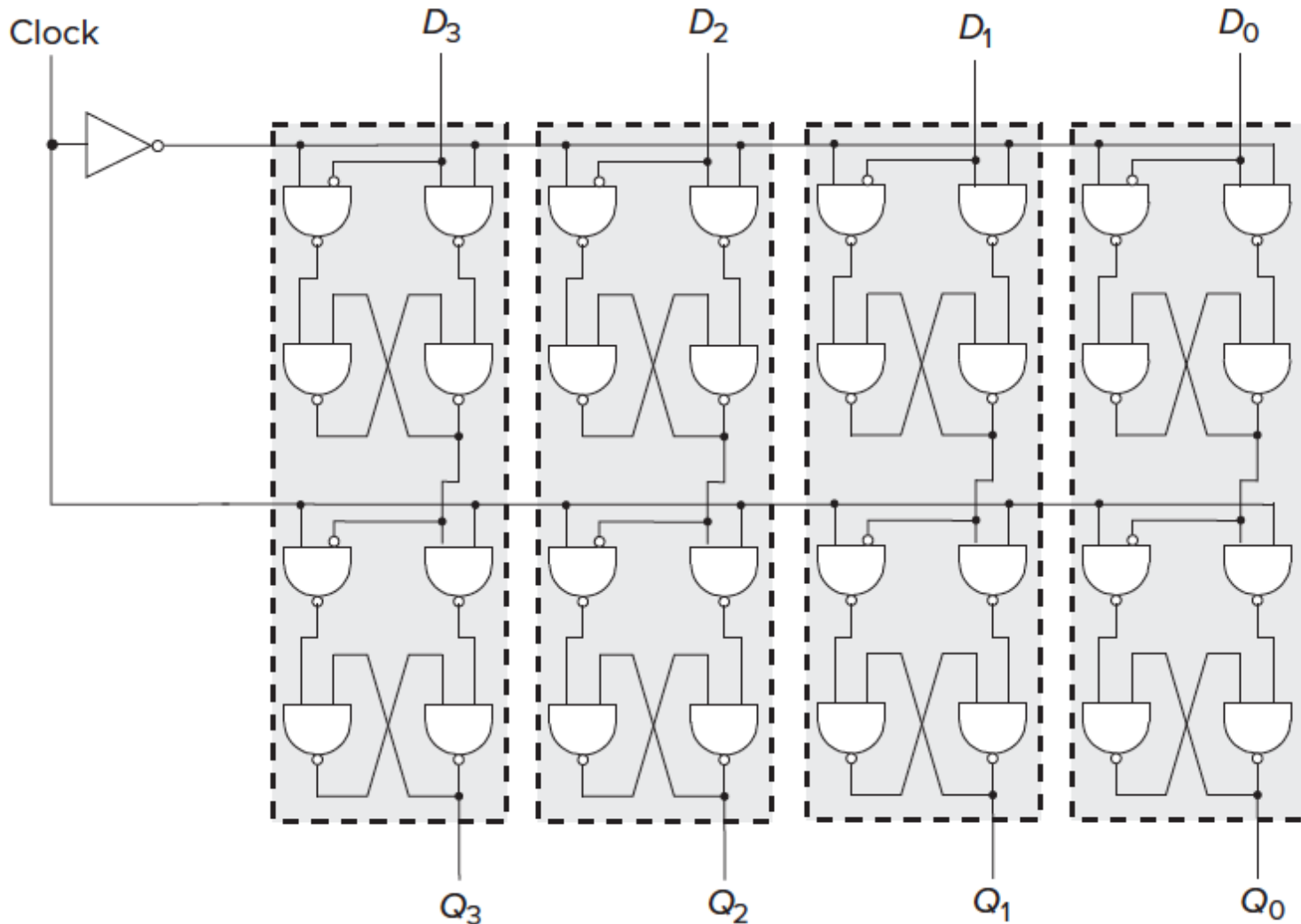
- Multiple parallel D flip-flops, each of which storing 1 bit



This line represents 4 wires

This register stores 4 bits

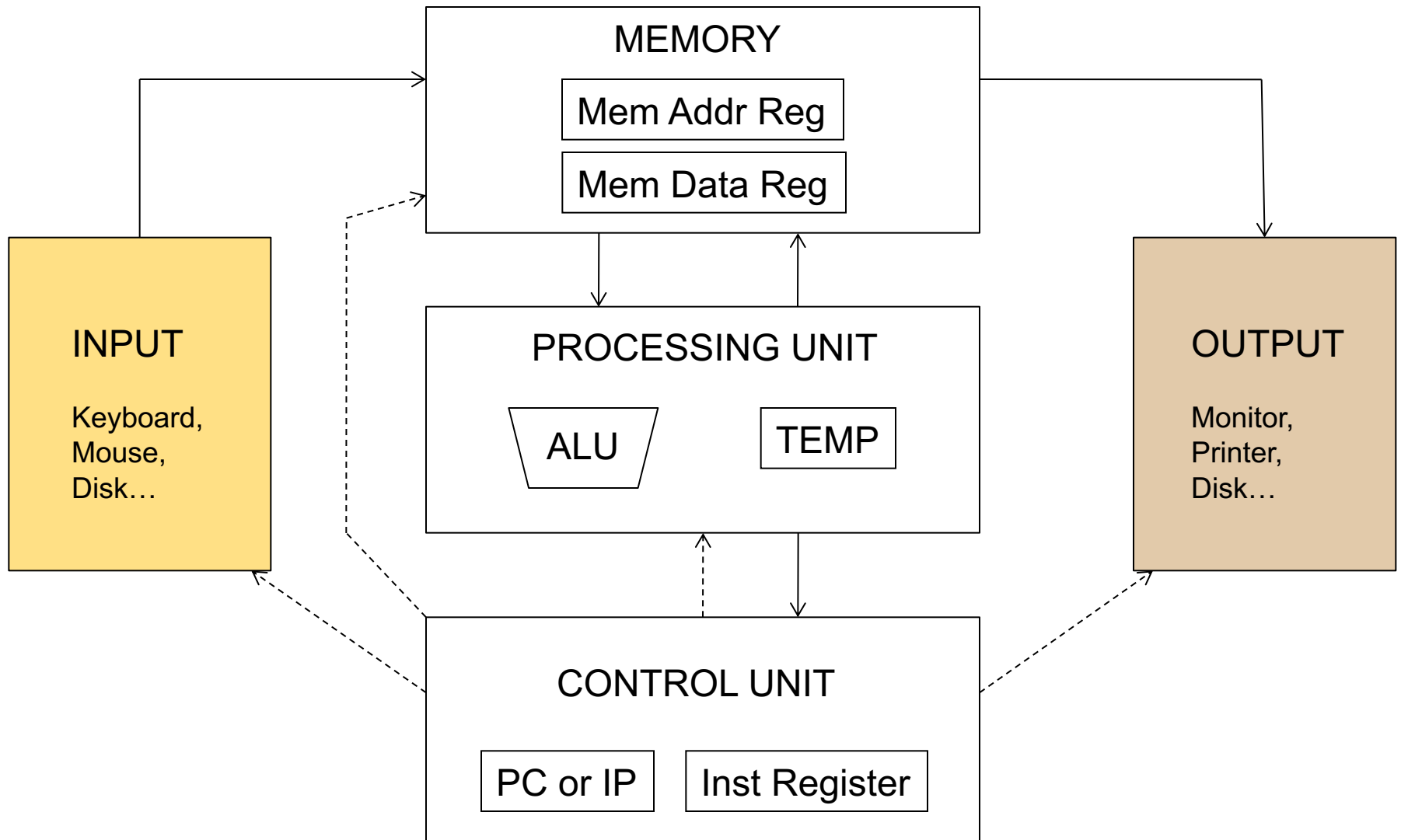
Recall: A 4-Bit D-Flip-Flop-Based Register (Internally)



MIPS Register File (Conventions)

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

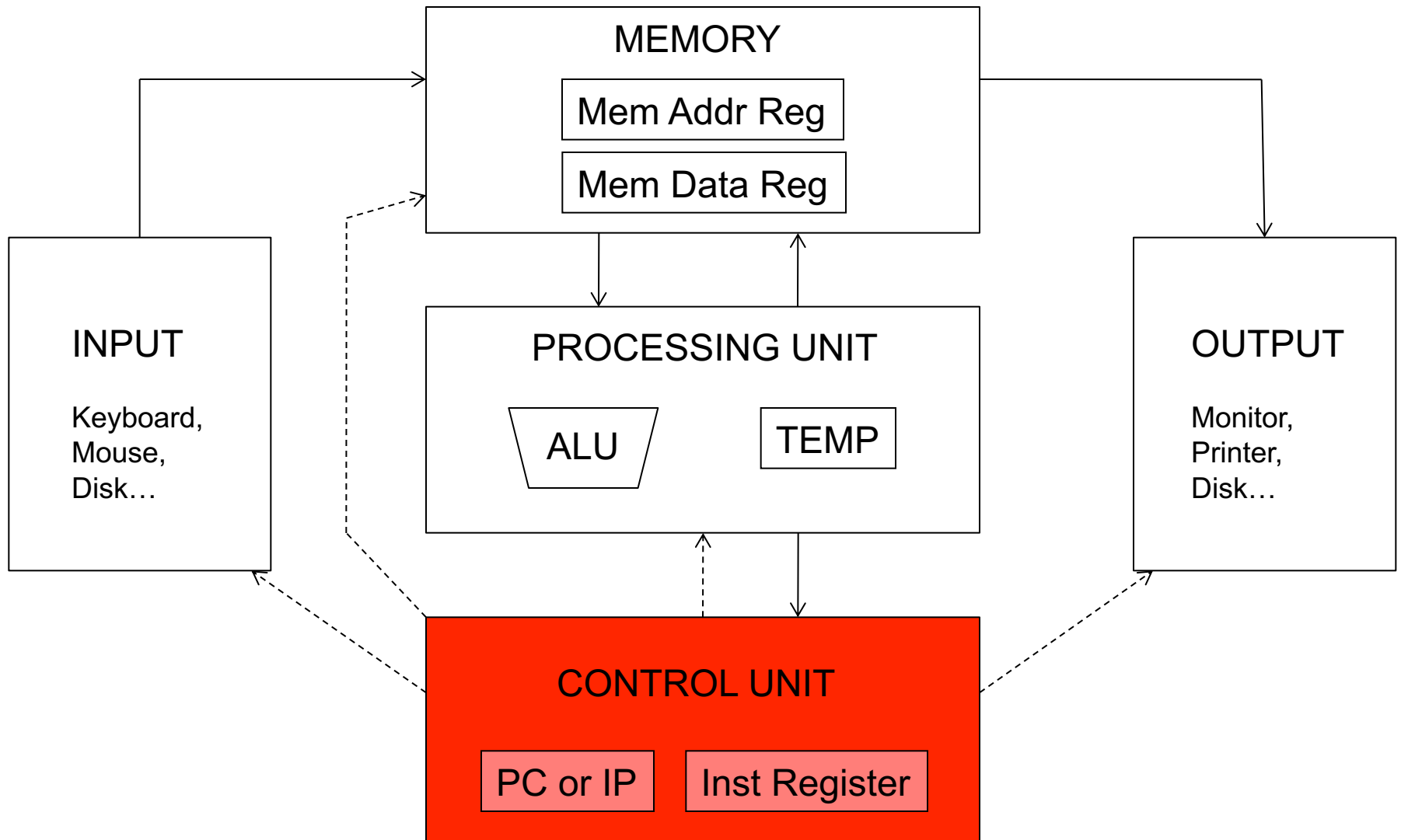
The Von Neumann Model



Input and Output

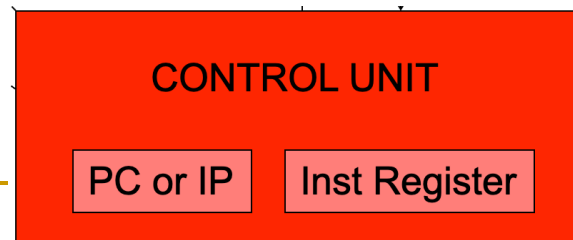
- Enable information to get into and out of a computer
- Many devices can be used for input and output
- They are called **peripherals**
 - **Input**
 - Keyboard
 - Mouse
 - Scanner
 - Disks
 - Etc.
 - **Output**
 - Monitor
 - Printer
 - Disks
 - Etc.
 - In LC-3, we consider keyboard and monitor

The Von Neumann Model

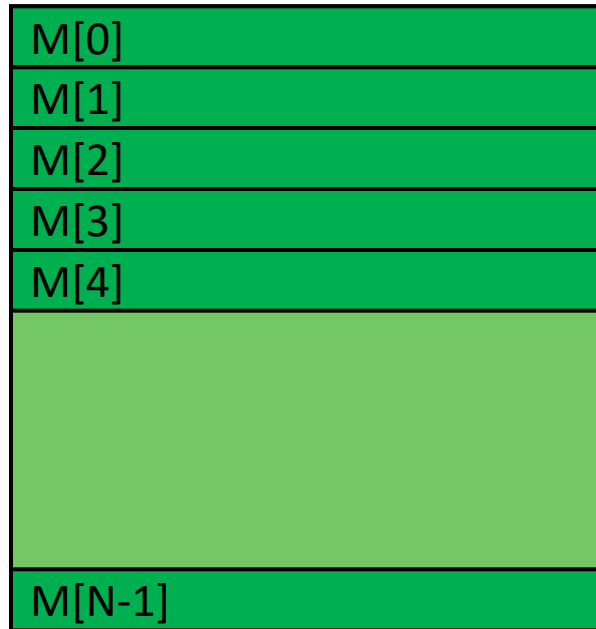


Control Unit

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



Programmer Visible (Architectural) State



Memory

array of storage locations
indexed by an address



Registers

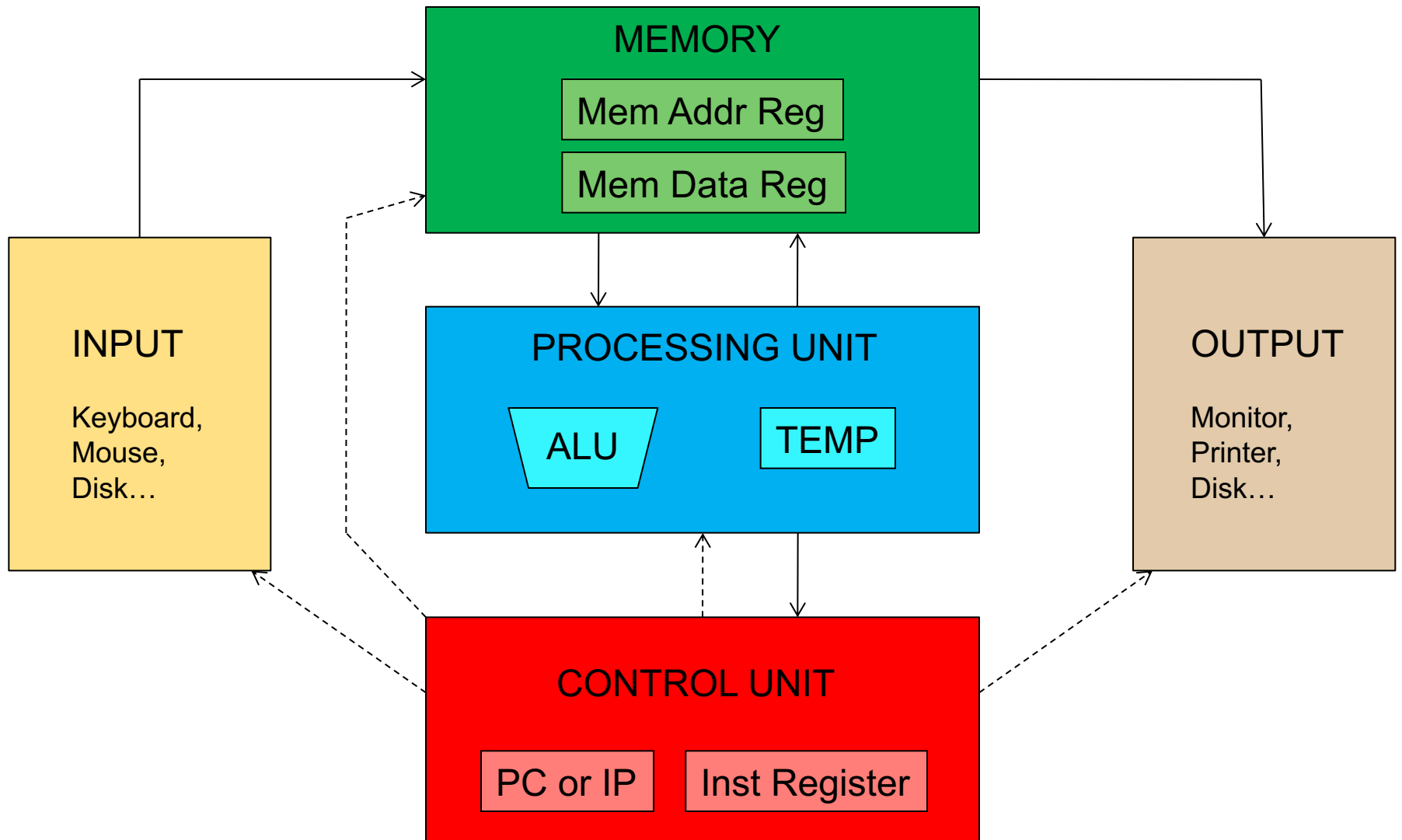
- given special names in the ISA (as opposed to addresses)
- general vs. special purpose

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

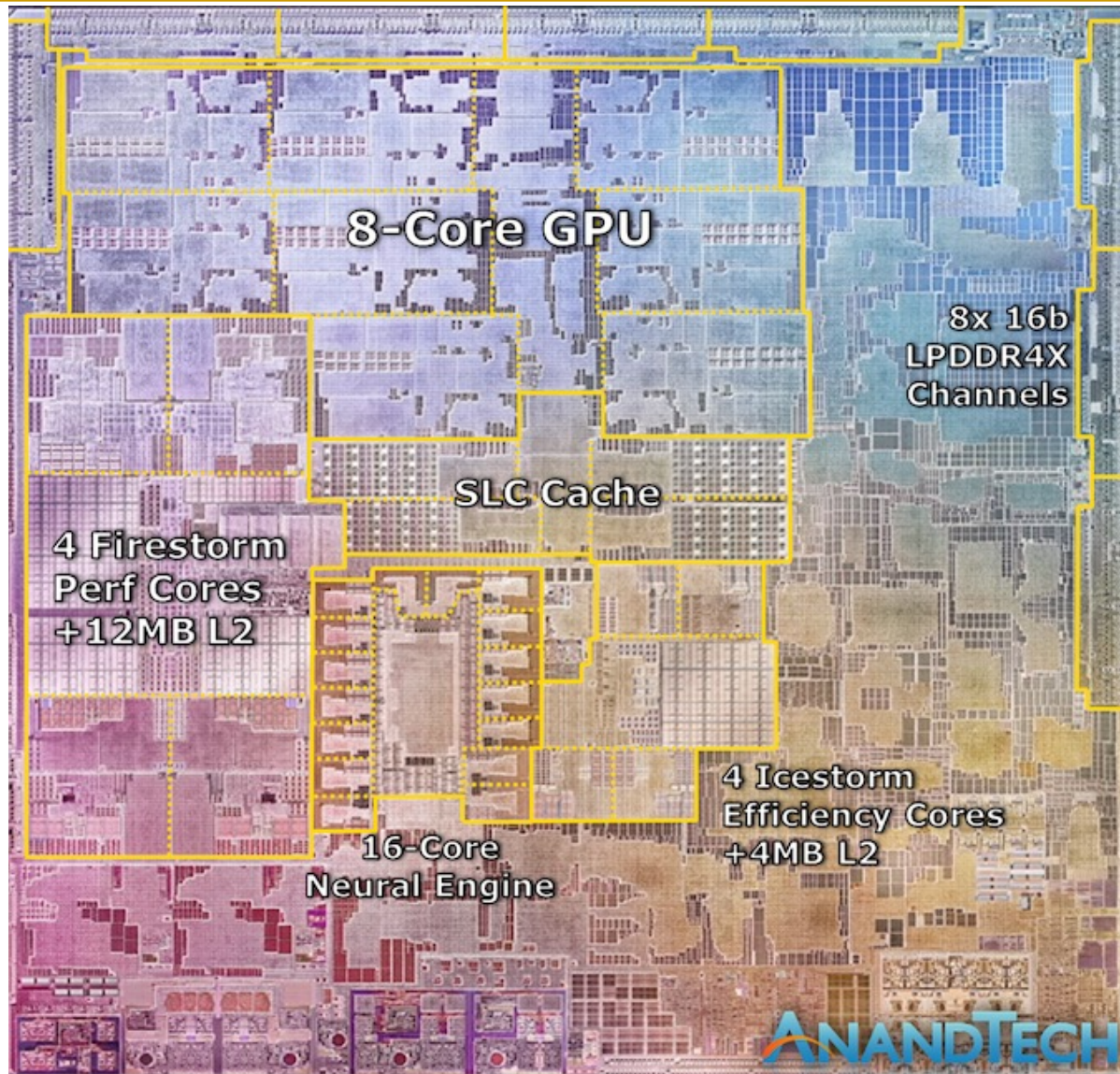


von Neumann Model: Two Key Properties

- 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

Another von Neumann Machine



Apple M1,
2021

Another von Neumann Machine

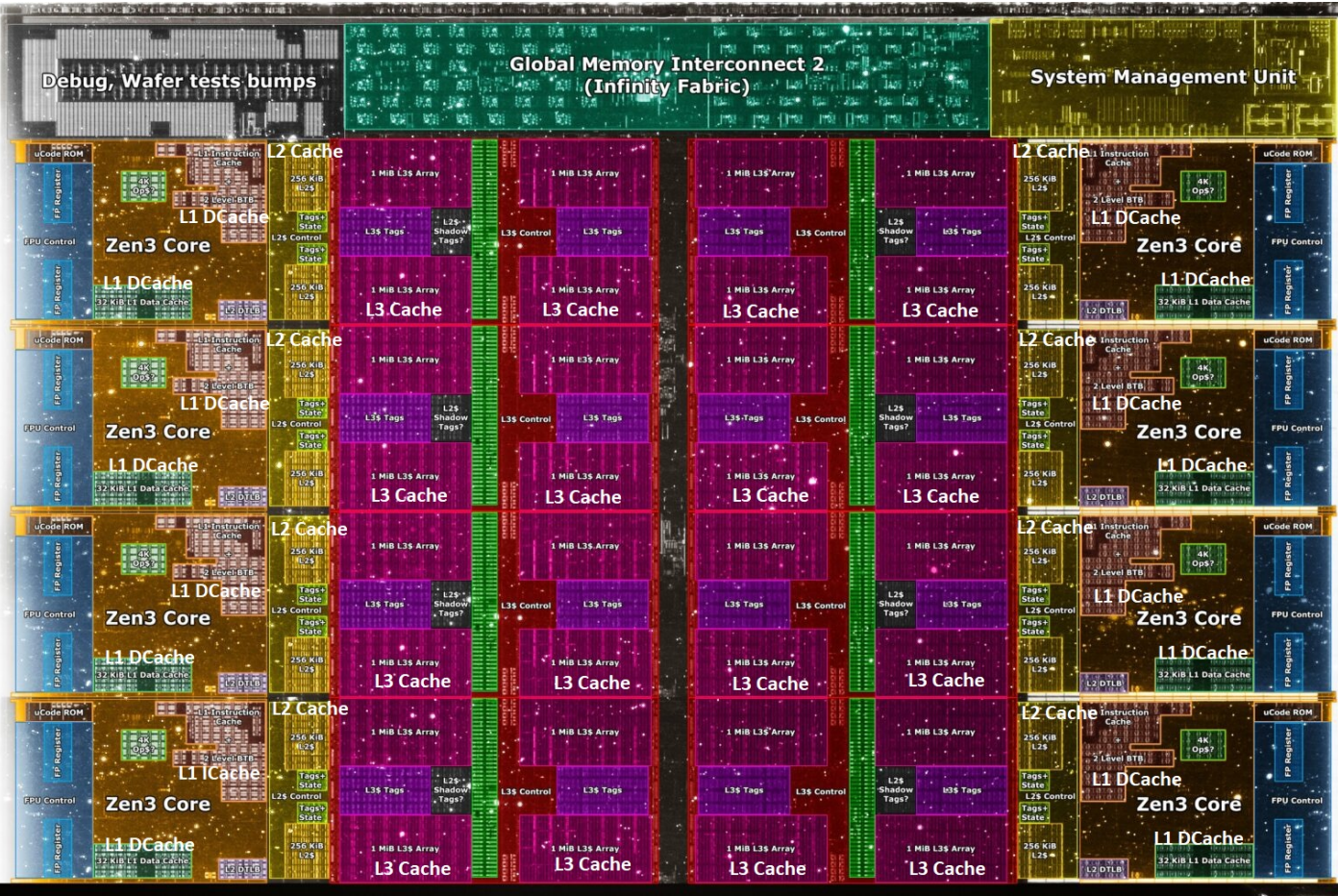


10nm ESF=Intel 7 Alder Lake die shot (~209mm²) from Intel: <https://www.intel.com/content/www/us/en/newsroom/news/12th-gen-core-processors.html>

Die shot interpretation by Locuza, October 2021

Intel Alder Lake,
2021

Another von Neumann Machine



Core Count:
8 cores/16 threads

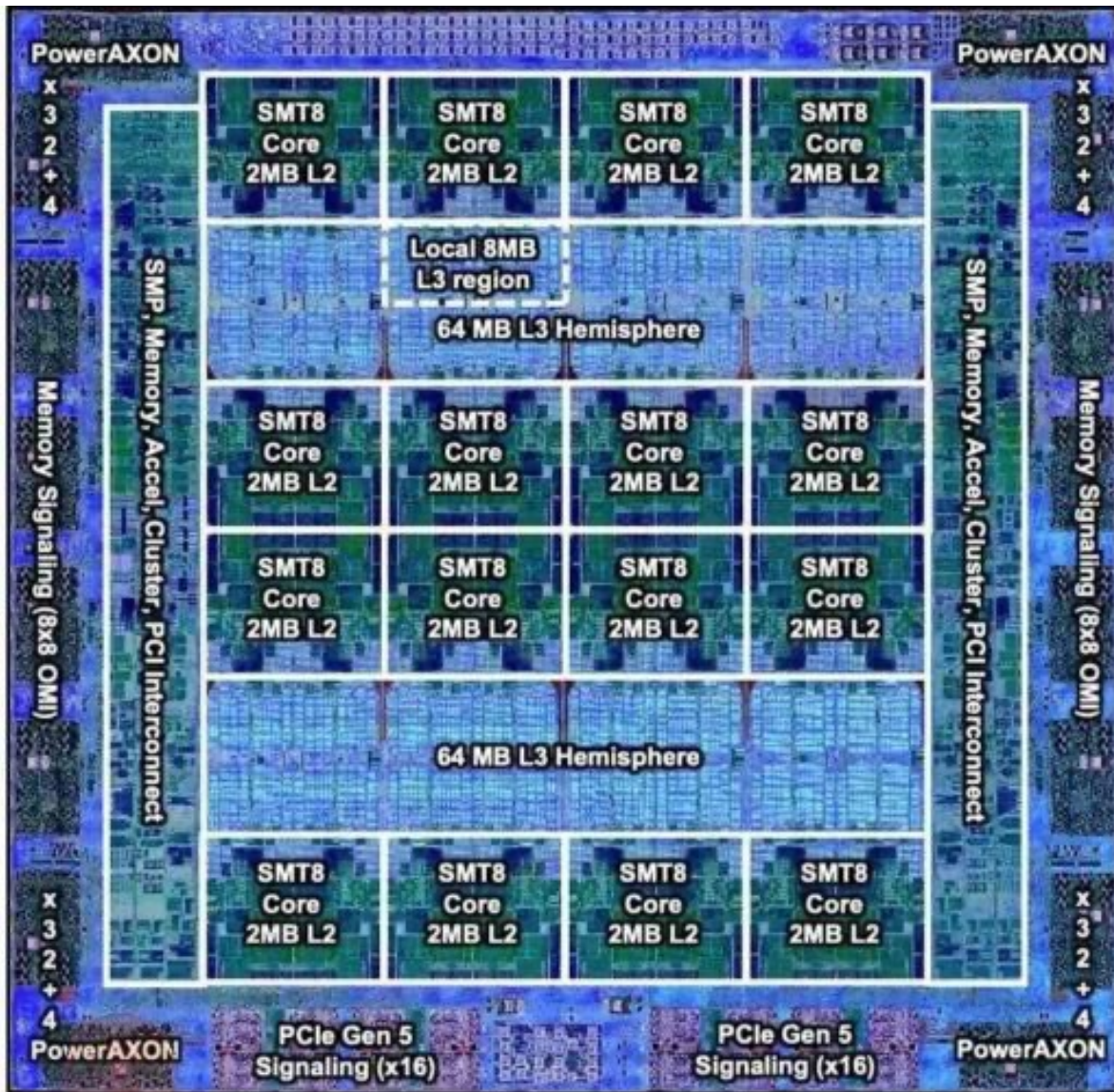
L1 Caches:
32 KB per core

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

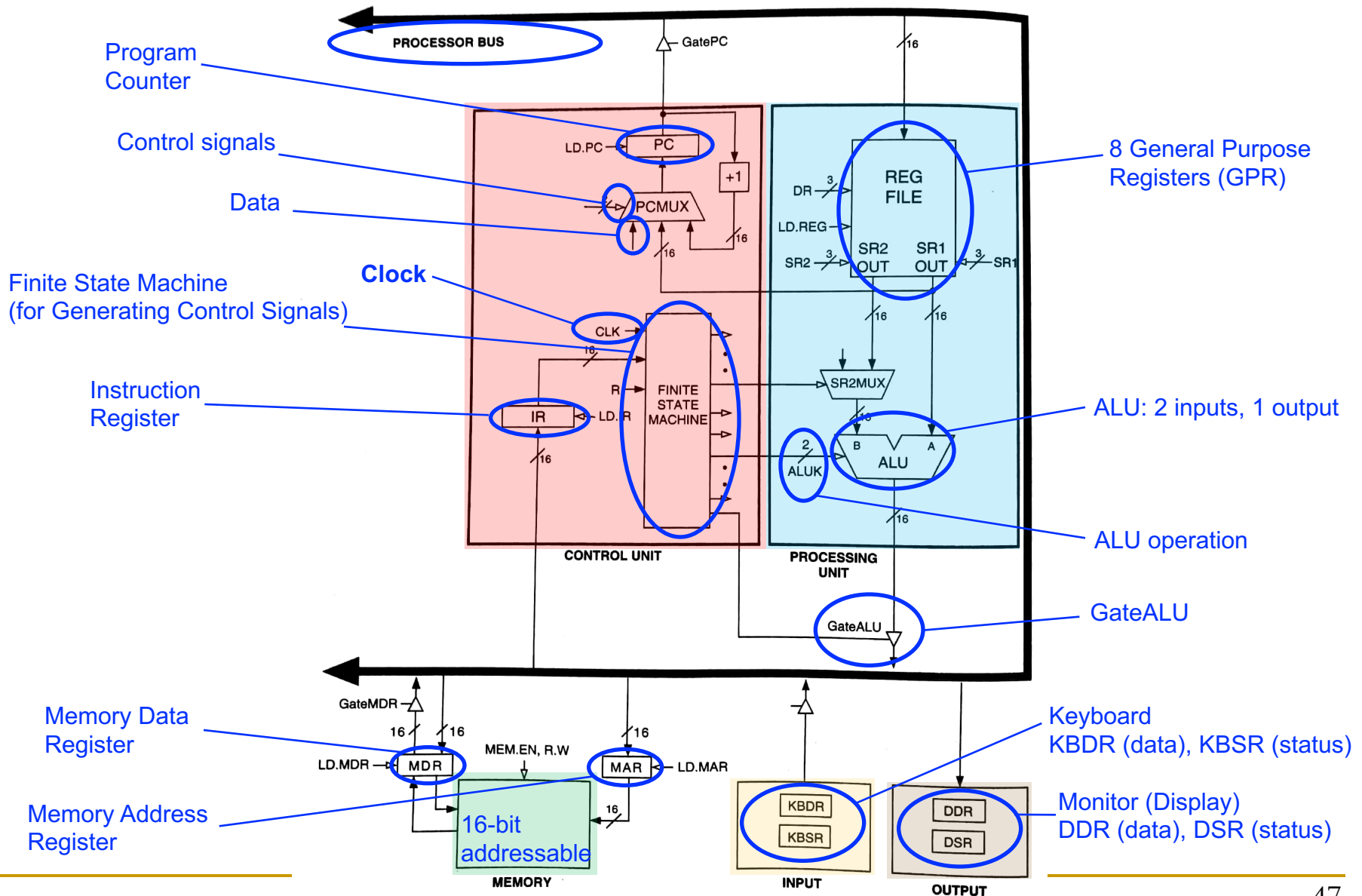


Figure 4.3 The LC-3 as an example of the von Neumann model

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)

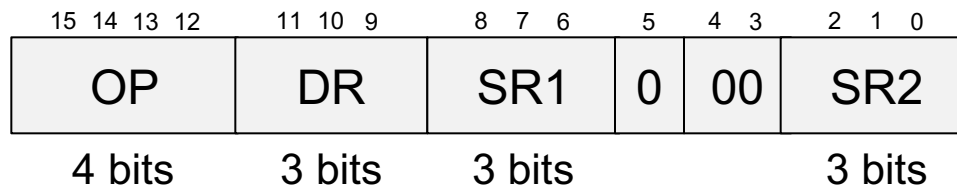
OP	DR	SR1			SR2
0001	000	001	0	00	010
15 14 13 12	11 10 9	8 7 6	5	4 3	2 1 0

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

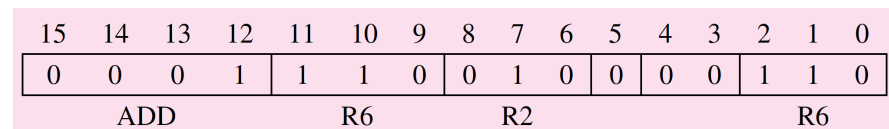
□ **Semantics:** $DR \leftarrow SR1 + SR2$

■ E.g., AND = 0101

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

□ SR1, SR2 = source registers

□ DR = destination register



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)

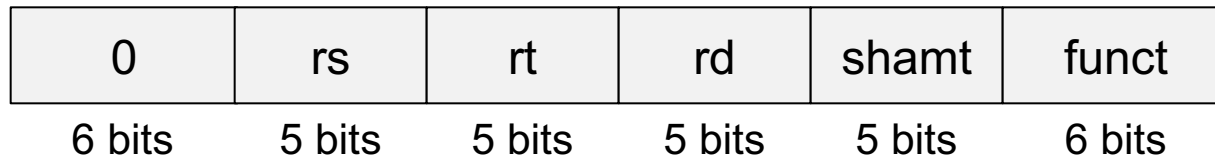
op	rs	rt	rd	shamt	funct
000000	10001	10010	10000	00000	100000

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

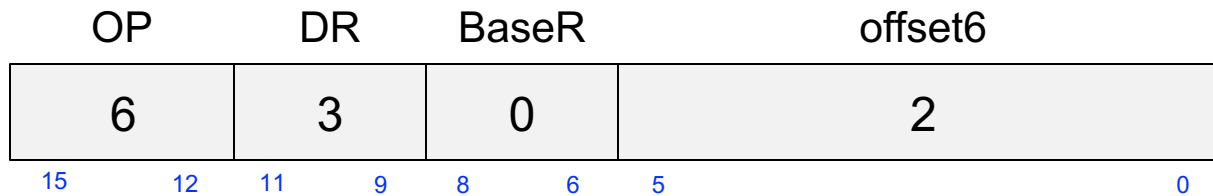
Instruction Format With Immediate

■ LC-3

LC-3 assembly

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

Field Values

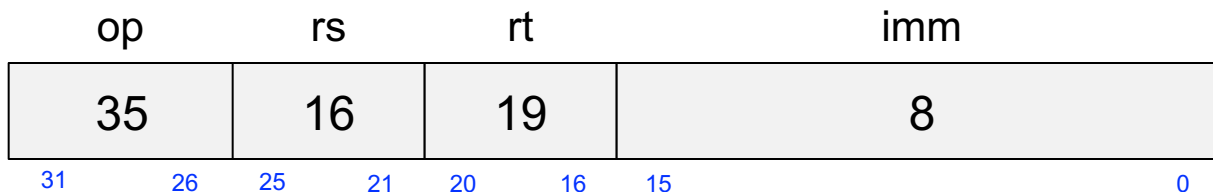


■ MIPS

MIPS assembly

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

Field Values



I-Type

Instruction (Processing) Cycle

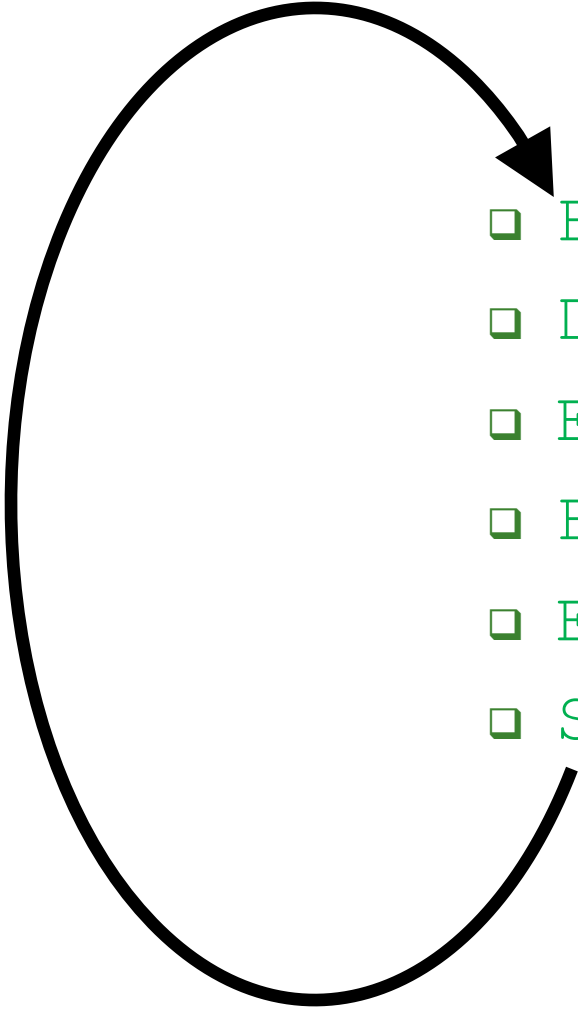
How Are These Instructions Executed?

- 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
 - ❑ DECODE
 - ❑ EVALUATE ADDRESS
 - ❑ FETCH OPERANDS
 - ❑ EXECUTE
 - ❑ STORE RESULT

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

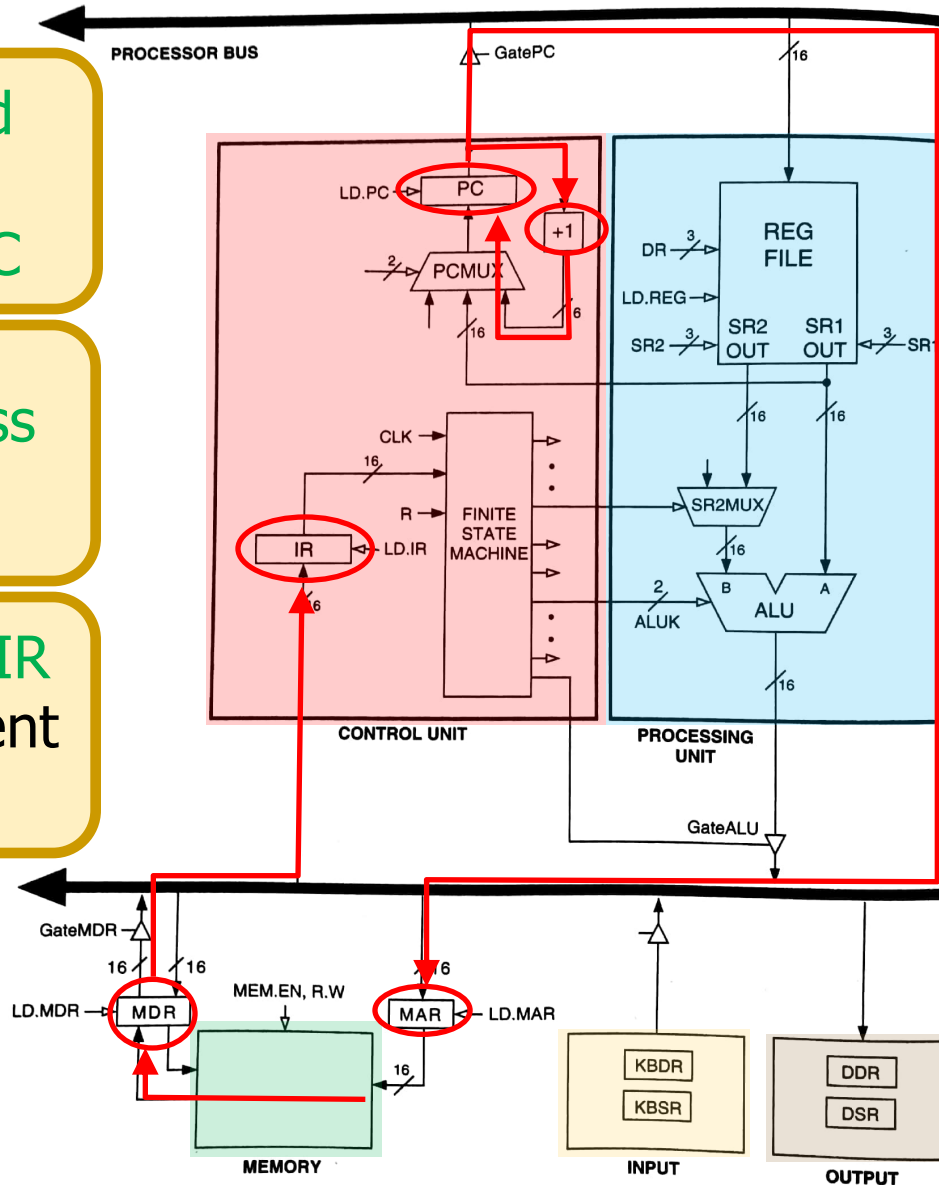


Figure 4.3 The LC-3 as an example of the von Neumann model

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

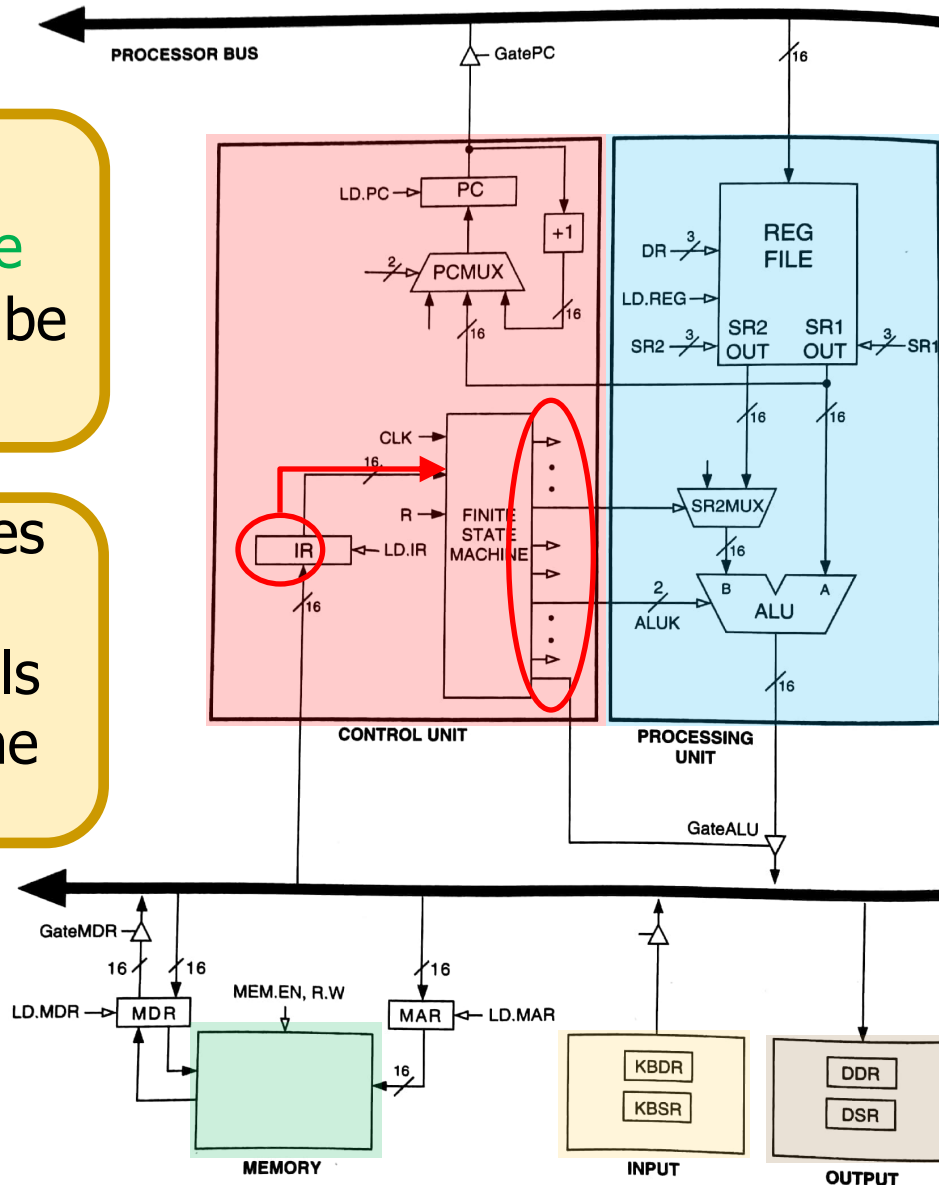
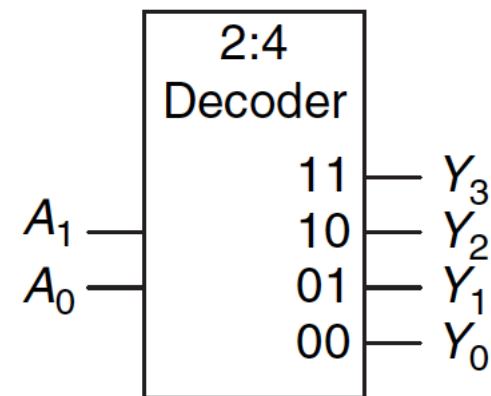


Figure 4.3 The LC-3 as an example of the von Neumann model

Recall: Decoder

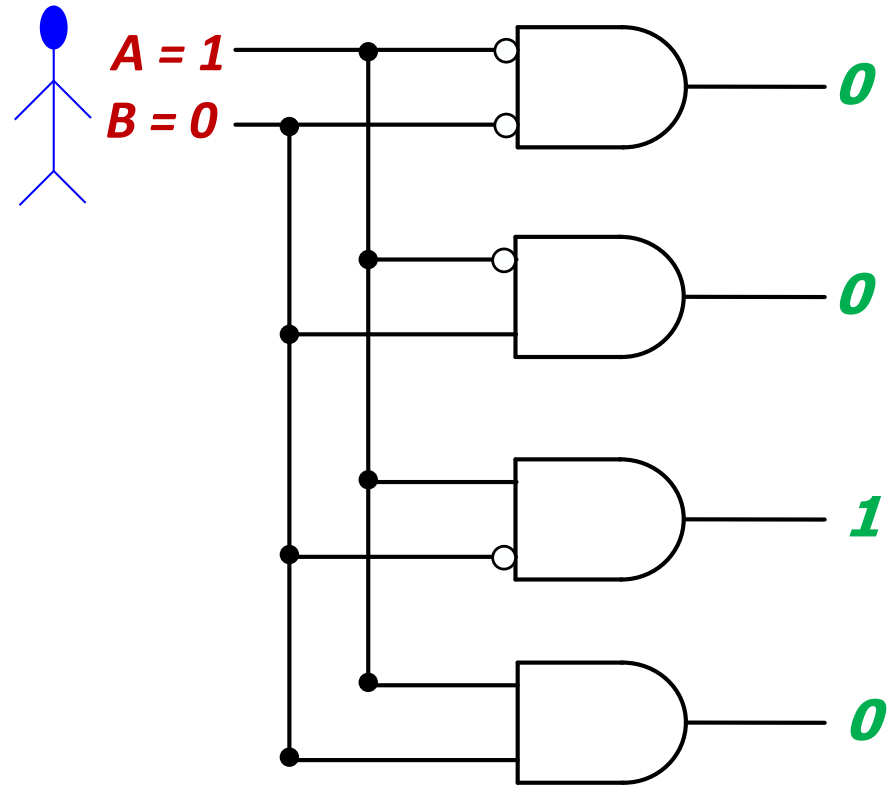
- “Input pattern detector”
- n inputs and 2^n outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The **output** that is logically 1 is the output corresponding to the input **pattern** that the logic circuit is expected to detect
- Example: 2-to-4 decoder

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*)**



To Come: Full State Machine for LC-3b

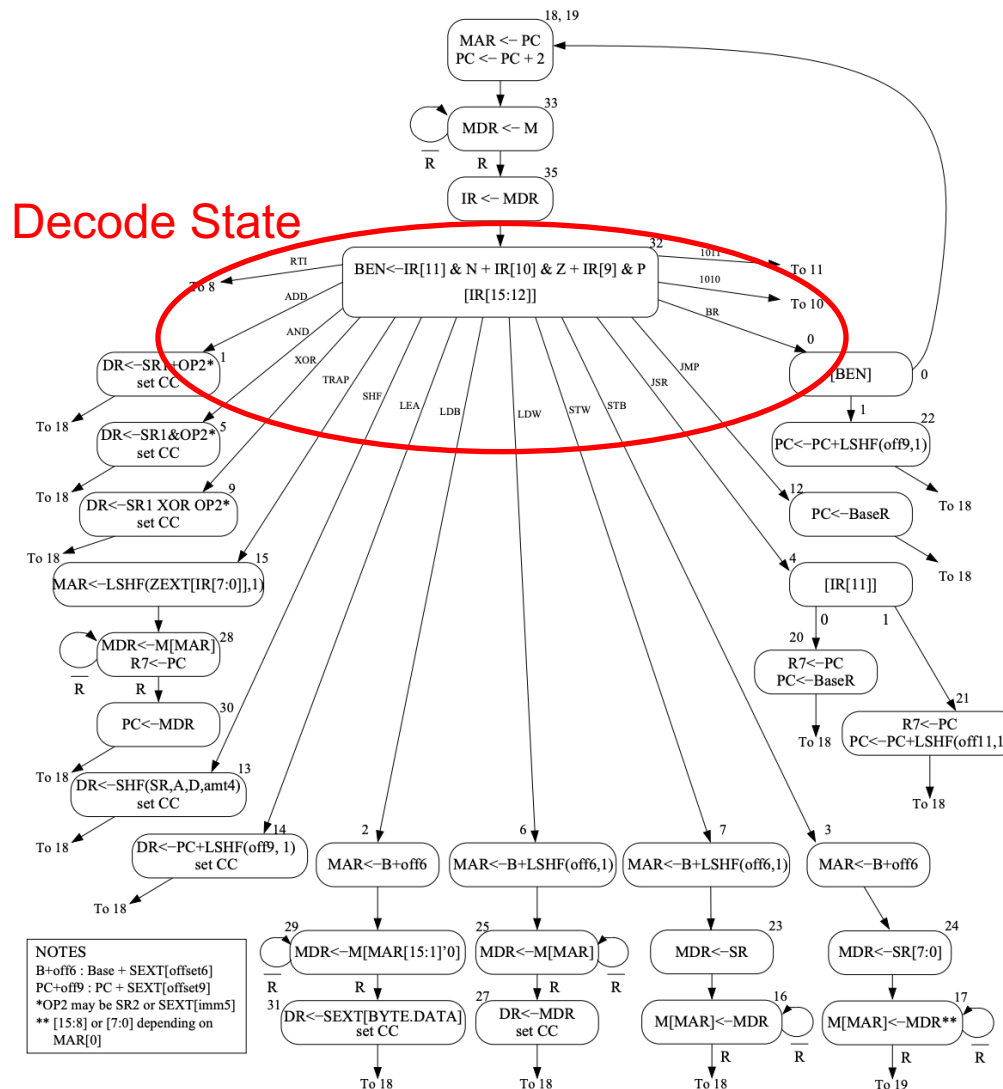


Figure C.2: A state machine for the LC-3b

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

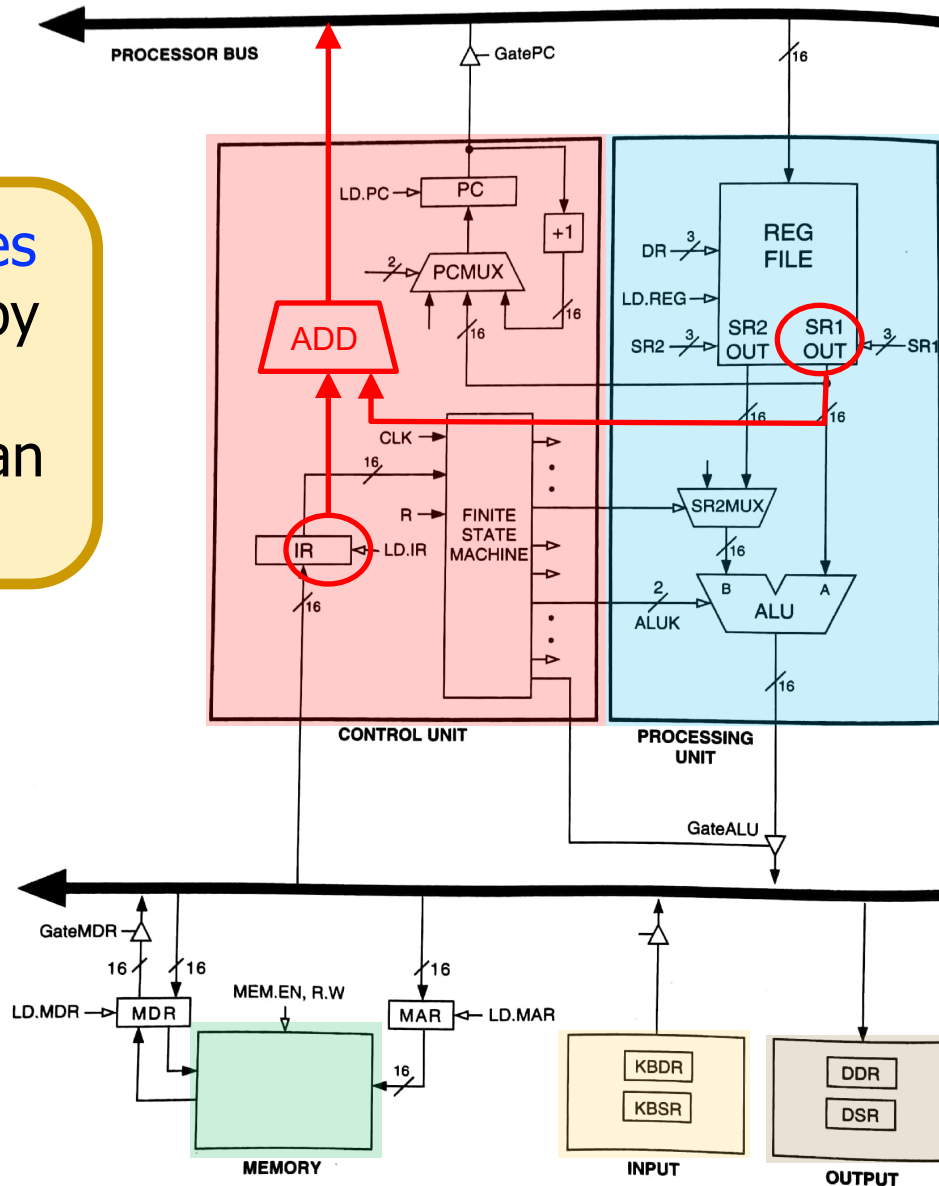


Figure 4.3 The LC-3 as an example of the von Neumann model

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)

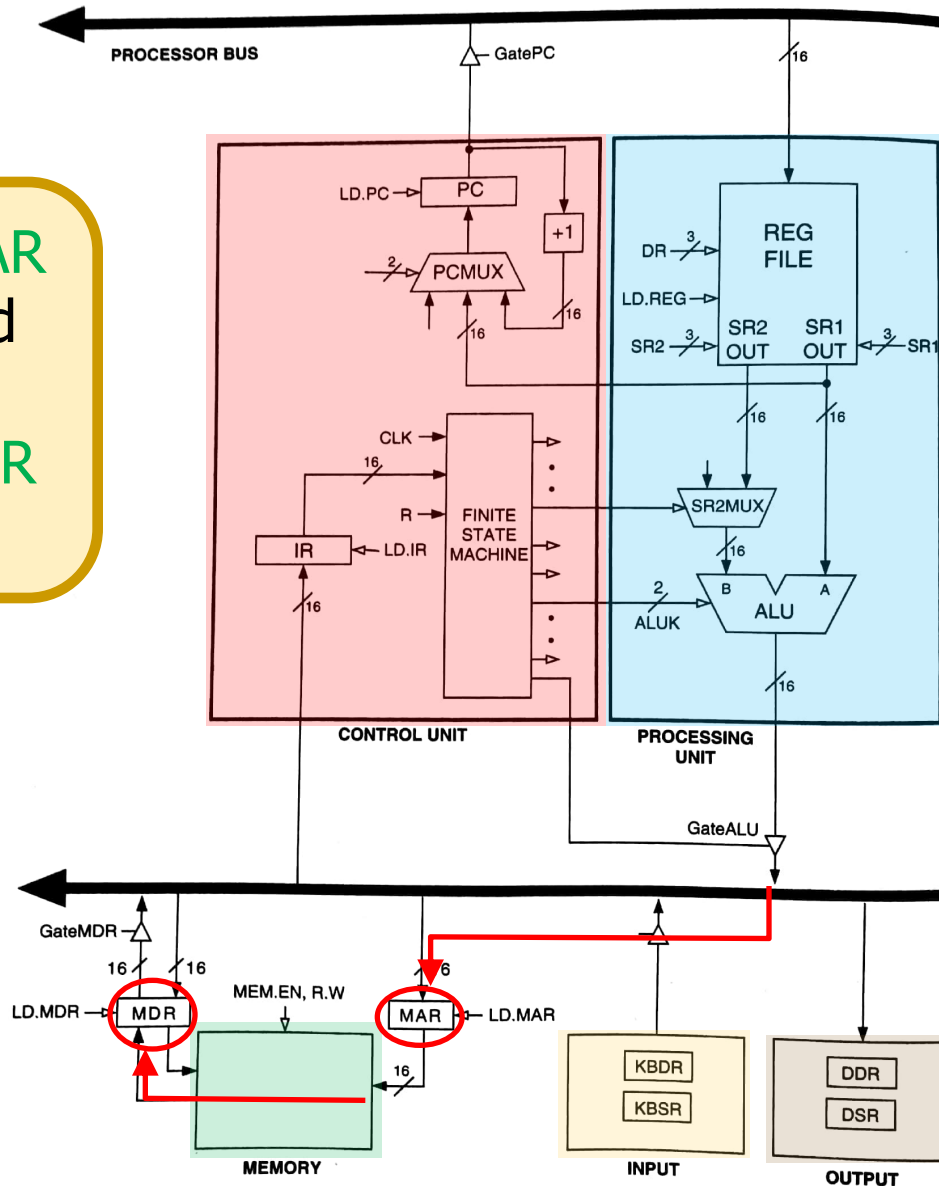


Figure 4.3 The LC-3 as an example of the von Neumann model

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

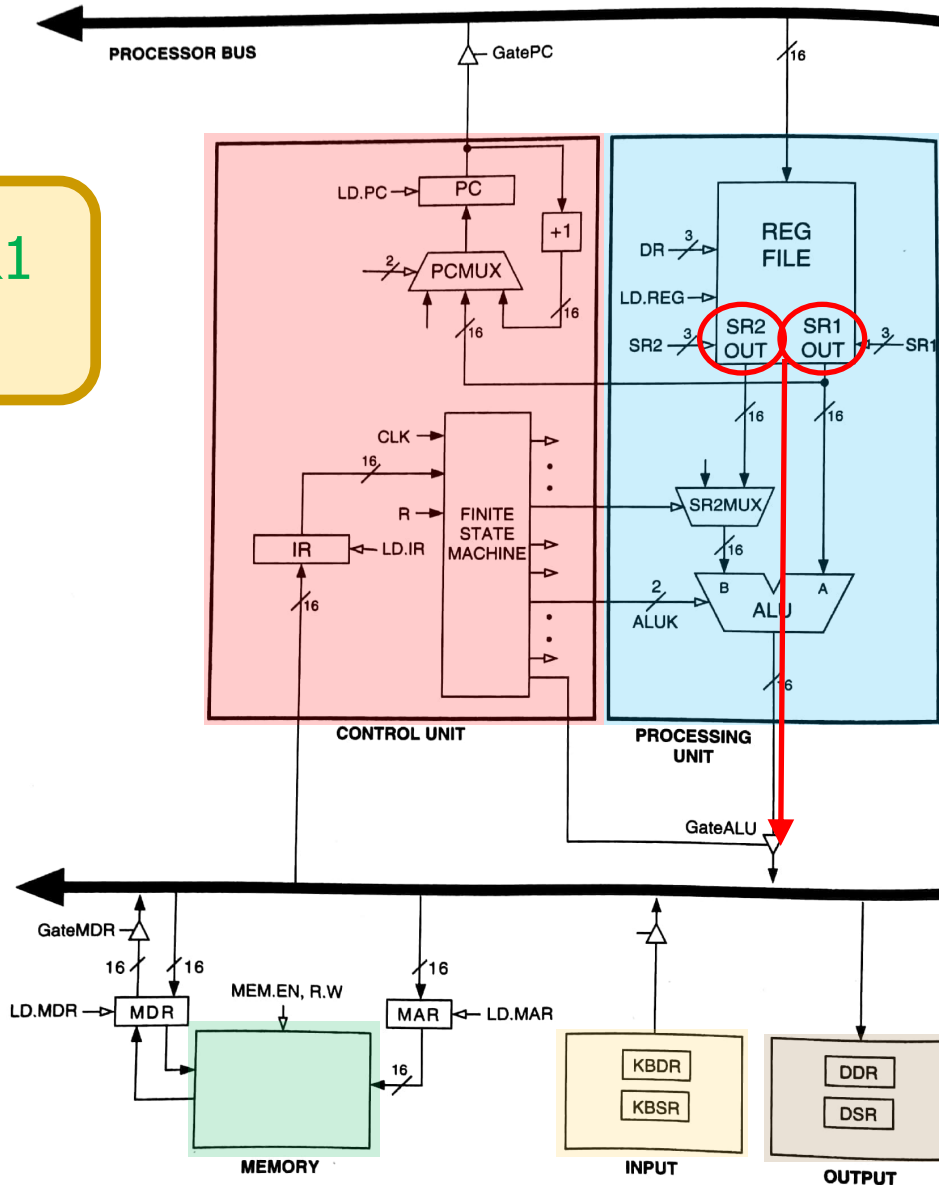


Figure 4.3 The LC-3 as an example of the von Neumann model

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

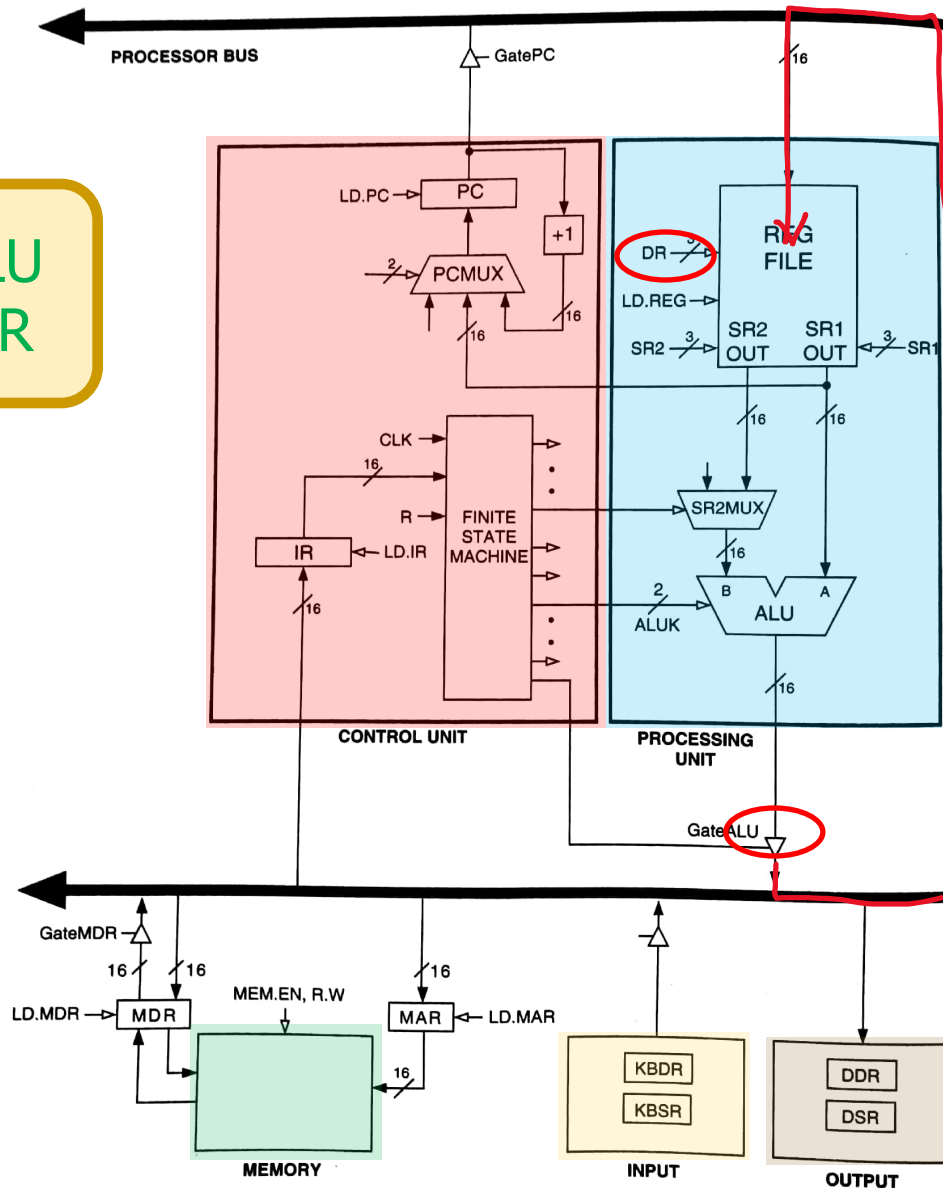


Figure 4.3 The LC-3 as an example of the von Neumann model

STORE RESULT in LC-3

LDR loads
MDR into DR

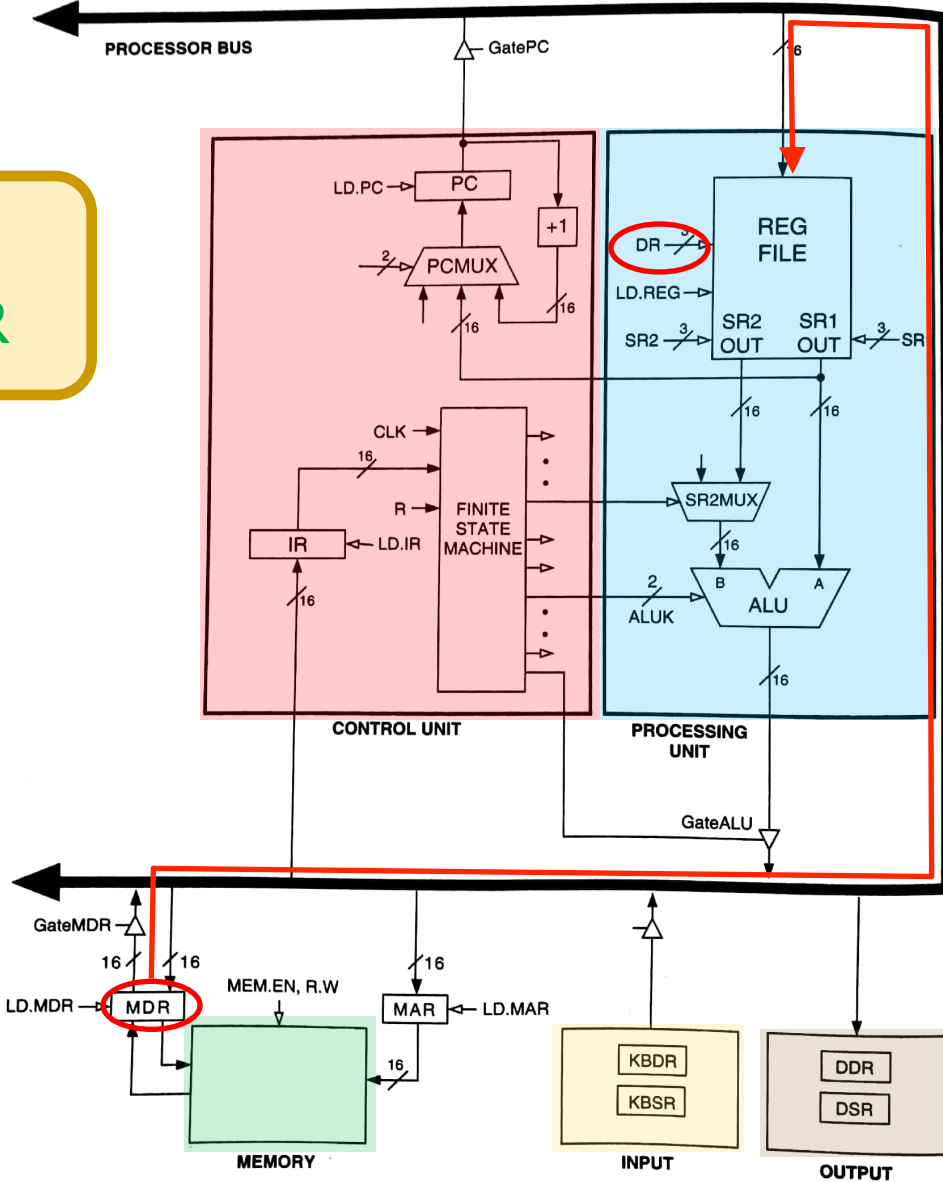
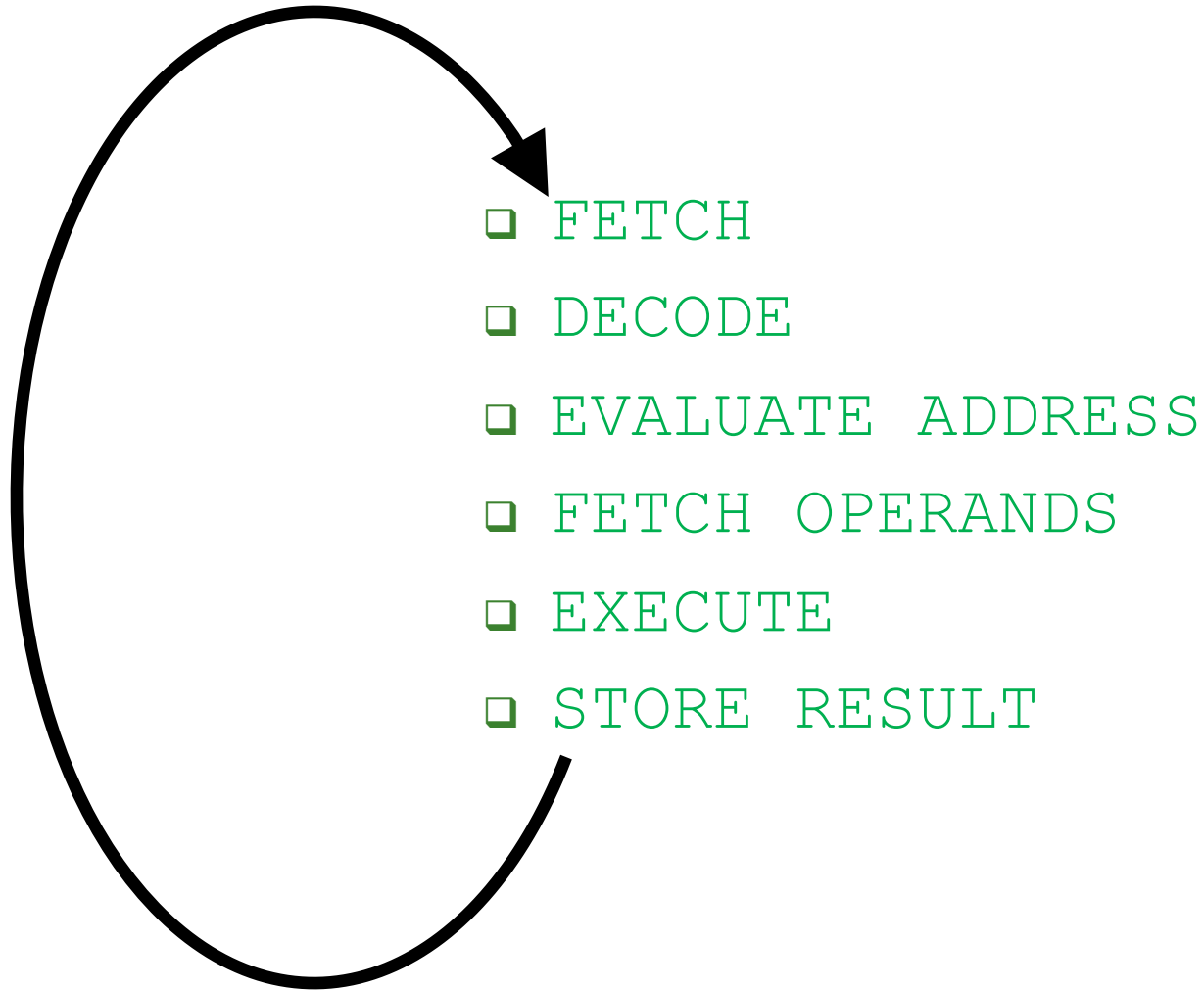


Figure 4.3 The LC-3 as an example of the von Neumann model

The Instruction Cycle



Changing the Sequence of Execution

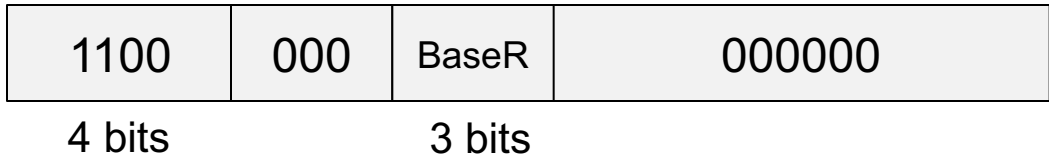
- A computer program **executes in sequence** (i.e., in program order)
 - First instruction, second instruction, third instruction and so on
- Unless we **change the sequence of execution**
- **Control instructions** allow a program to execute **out of sequence**
 - They can change the PC by loading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



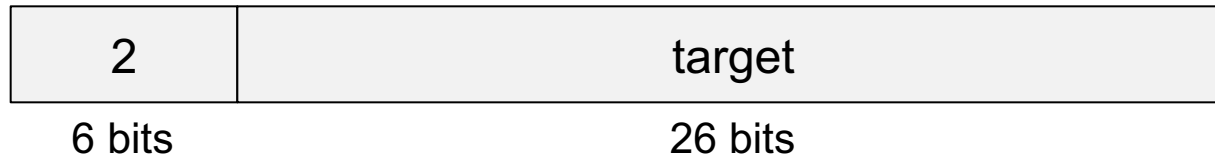
- BaseR = Base register
- $PC \leftarrow R2$ (Register identified by BaseR)
- Variations
 - RET: special case of JMP where BaseR = R7
 - JSR, JSRR: jump to subroutine

This is register addressing mode

Jump in MIPS

- Unconditional branch or jump

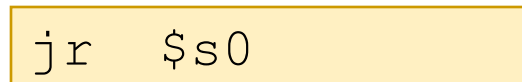
- MIPS



J-Type

- 2 = opcode
- target = target address
- $PC \leftarrow PC^{\dagger} [31:28] \mid \text{sign-extend}(\text{target}) * 4$
- Variations
 - jal: jump and link (function calls)

- jr: jump register



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

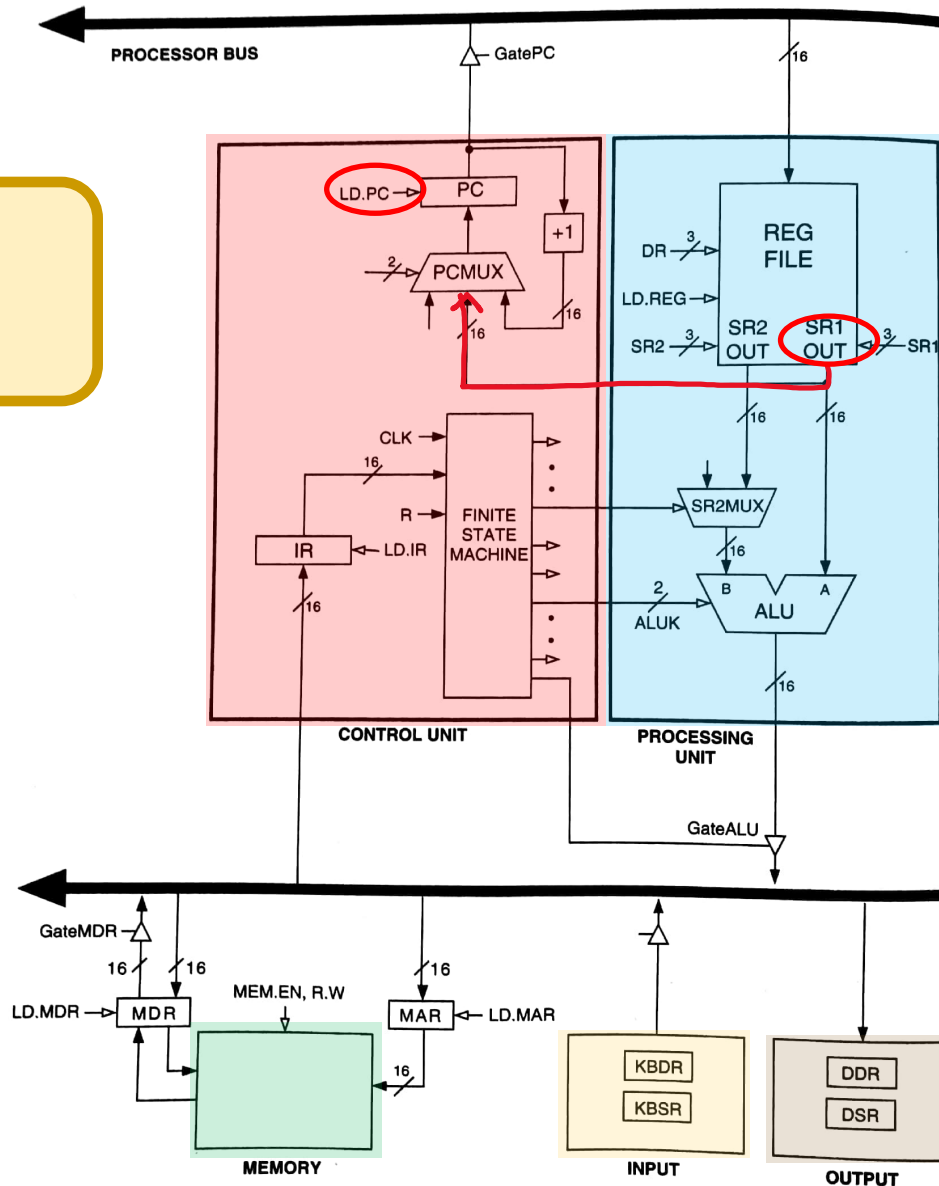
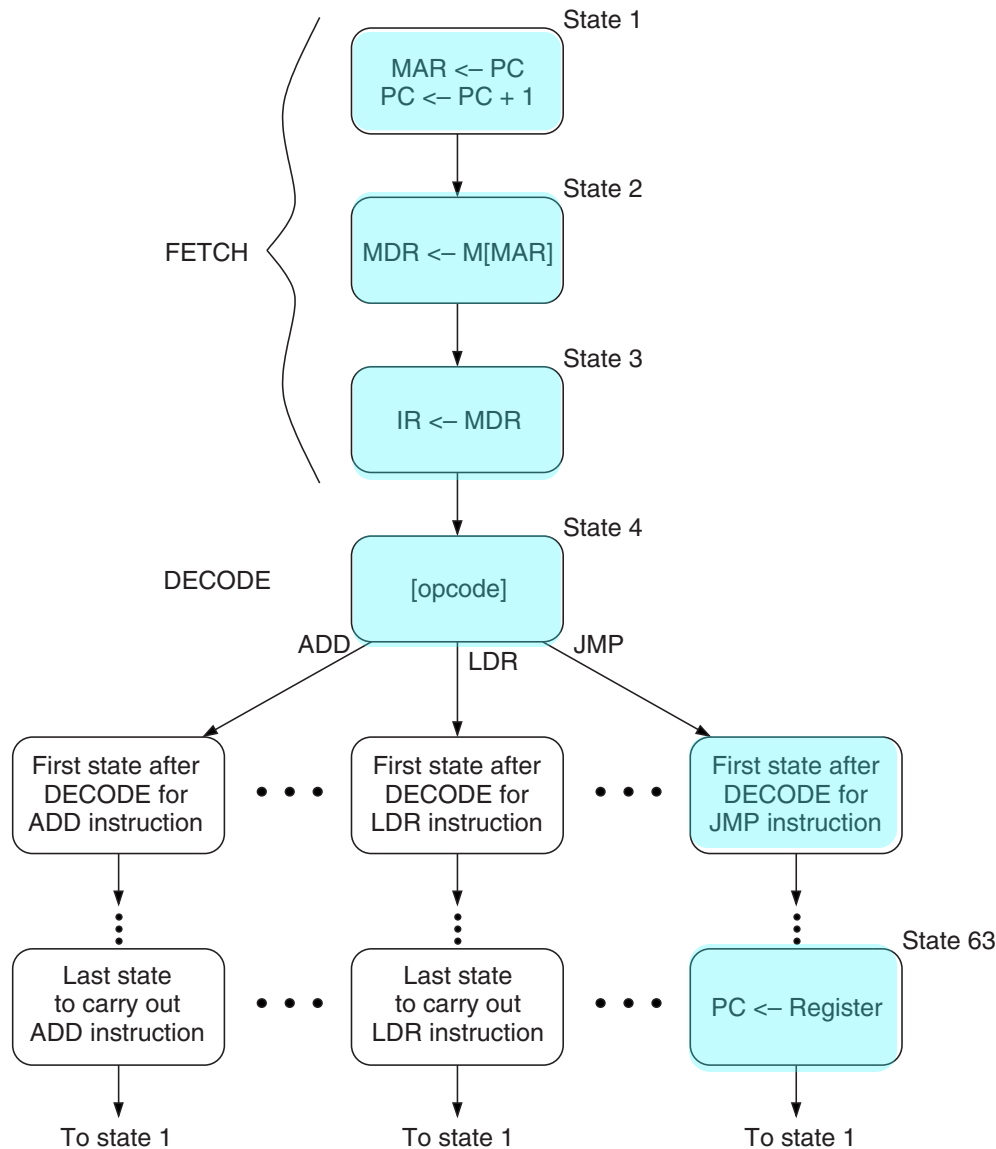


Figure 4.3 The LC-3 as an example of the von Neumann model

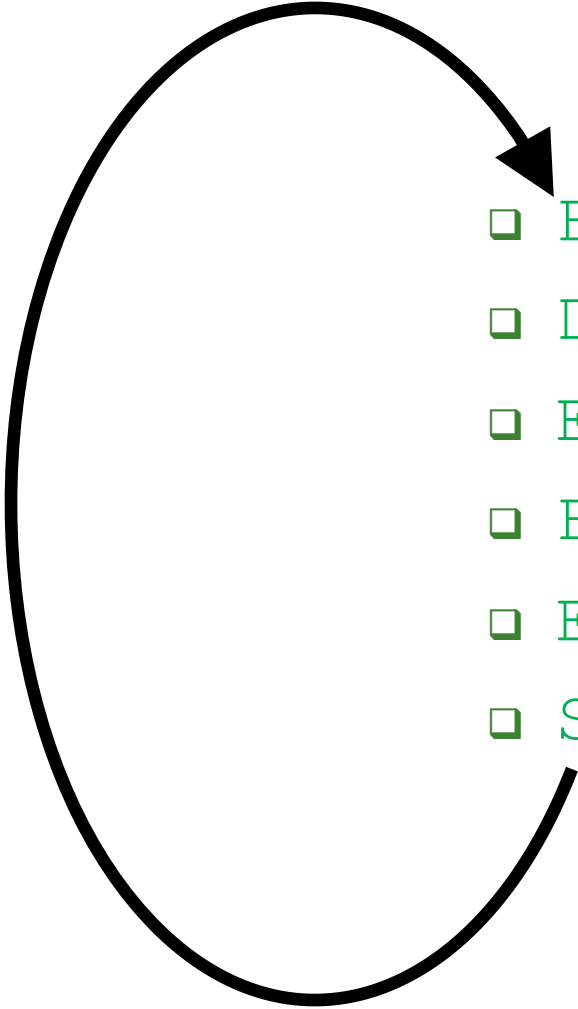
Control of the Instruction Cycle



- **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

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

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

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**
 - 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

Problem
Algorithm
Program
ISA
Microarchitecture
Circuits
Electrons

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

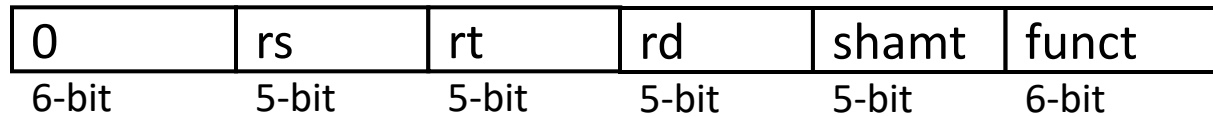
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD ⁺	0001			DR	SR1	0	00	SR2								
ADD ⁺	0001			DR	SR1	1	imm5									
AND ⁺	0101			DR	SR1	0	00	SR2								
AND ⁺	0101			DR	SR1	1	imm5									
BR	0000			n	z	p	PCoffset9									
JMP	1100			000		BaseR			000000							
JSR	0100			1	PCoffset11											
JSRR	0100			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

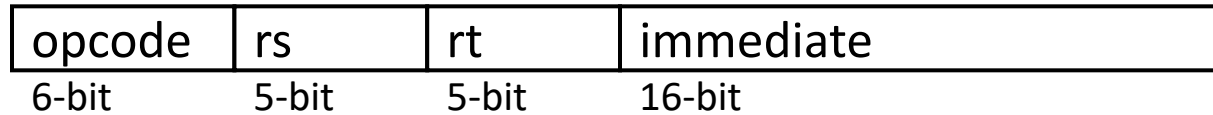
Opcodes in LC-3b

	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															

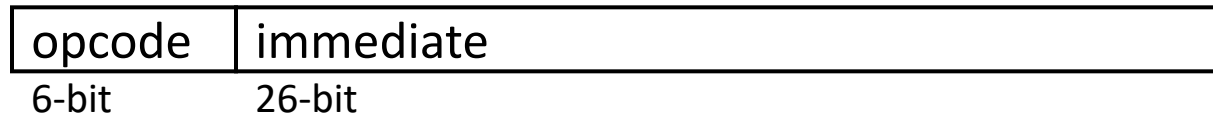
MIPS Instruction Types



R-type



I-type



J-type

Func in MIPS R-Type Instructions (I)

Opcode is 0
in MIPS
R-Type
instructions.

Func defines
the operation

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

Func	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)

Funcnt in MIPS R-Type Instructions (II)

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

Funcnt	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	$[rd] = [rs] \mid [rt]$
100110 (38)	xor rd, rs, rt	xor	$[rd] = [rs] \wedge [rt]$
100111 (39)	nor rd, rs, rt	nor	$[rd] = \sim([rs] \mid [rt])$
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	$[rs] < [rt] ? [rd] = 1 : [rd] = 0$

- 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?

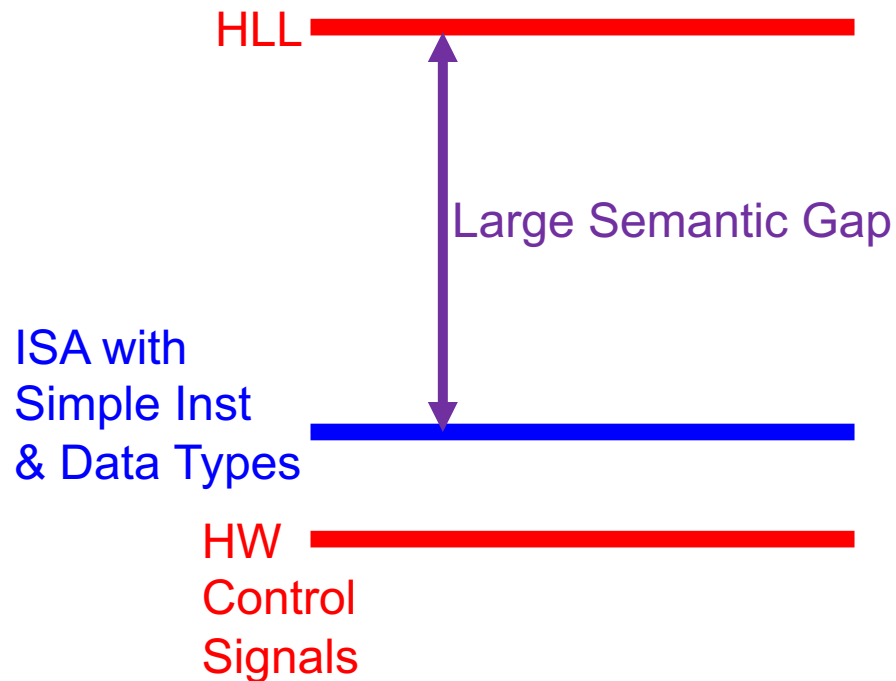
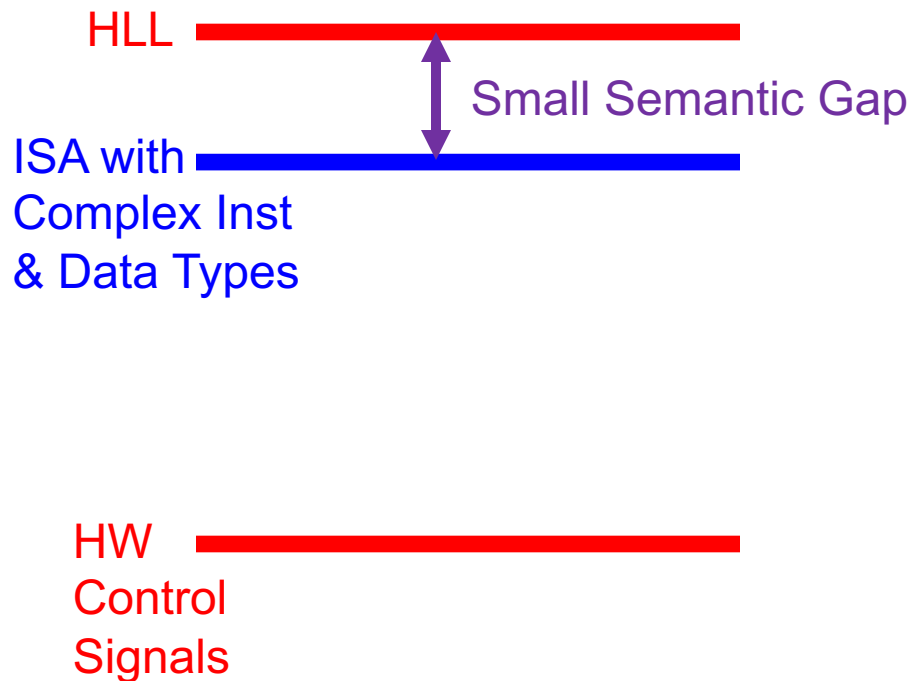
- An example of programmer vs. microarchitect tradeoff
- **Advantage of more data types:**
 - Enables **better mapping of high-level programming constructs to hardware**
 - Hardware can directly operate on data types present in programming languages → **small number of instructions and code size**
 - 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)



Complex vs. Simple Instructions+Data Types

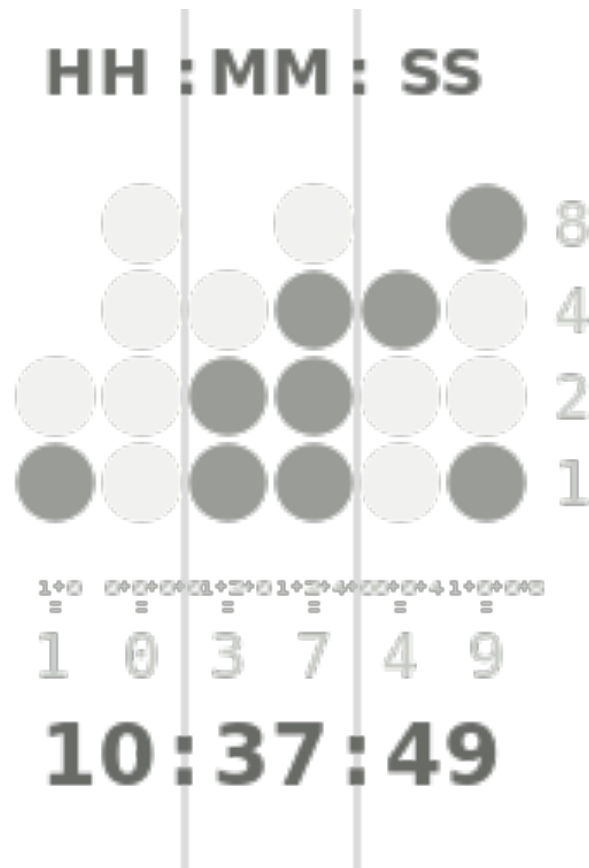
- **Complex instruction:** An instruction **does a lot of work**, e.g. many operations
 - ❑ Insert in a doubly linked list
 - ❑ Compute FFT
 - ❑ String copy
 - ❑ Matrix multiply
 - ❑ ...
- **Simple instruction:** An instruction **does little work** -- it is a primitive using which complex operations 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

Aside: An Example: BinaryCodedDecimal

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



"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

"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>

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

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



"Binary clock" on the English Wikipedia.

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

"Digital-BCD-clock" by Julu - 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

Addressing Modes

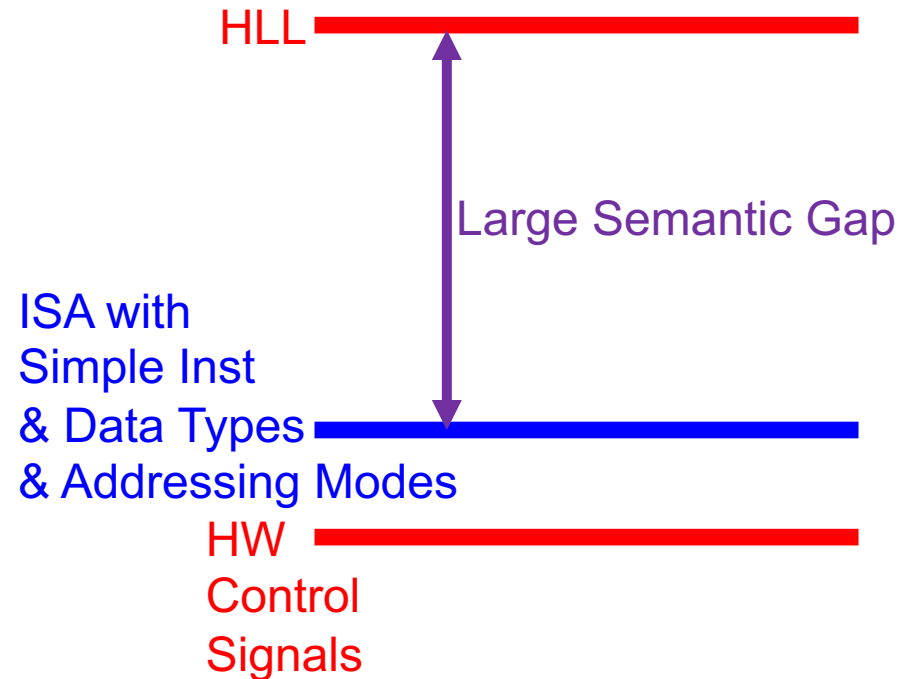
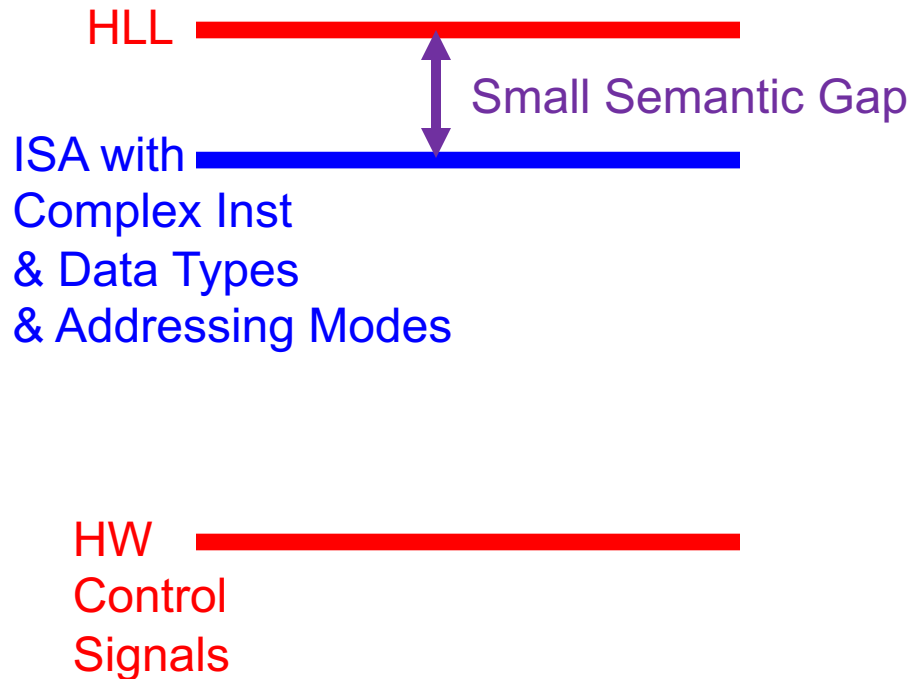
- An addressing mode is a mechanism for specifying where an operand is located
- There are five addressing modes in LC-3
 - Immediate or literal (constant)
 - The operand is in some bits of the instruction
 - Register
 - The operand is in one of R0 to R7 registers
 - Three memory addressing modes
 - 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 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)



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

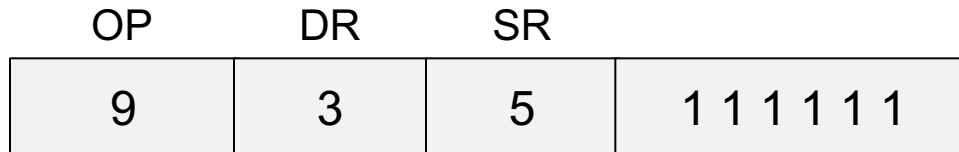
NOT in LC-3

NOT assembly and machine code

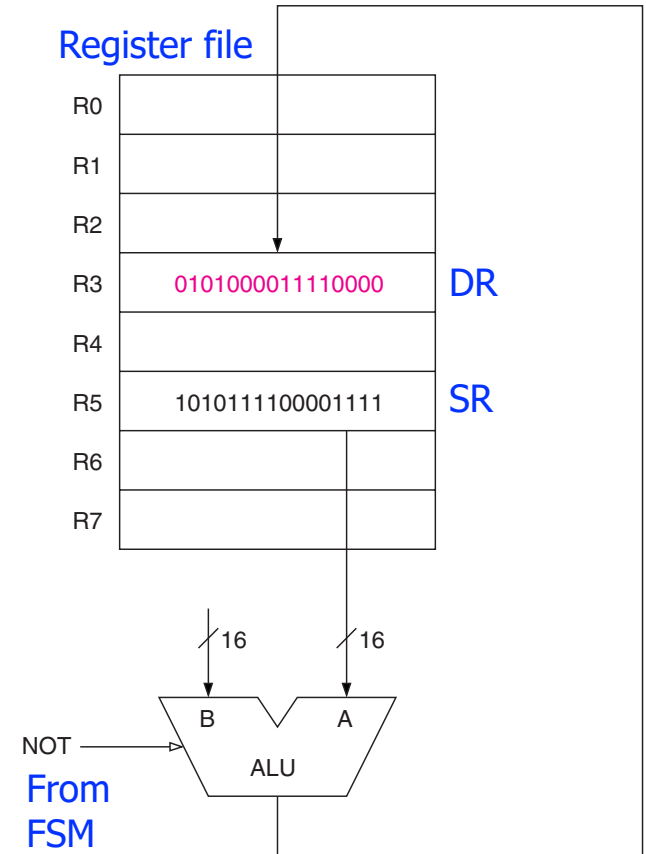
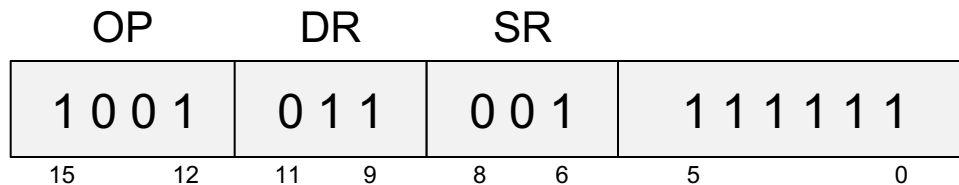
LC-3 assembly

```
NOT R3, R5
```

Field Values



Machine Code



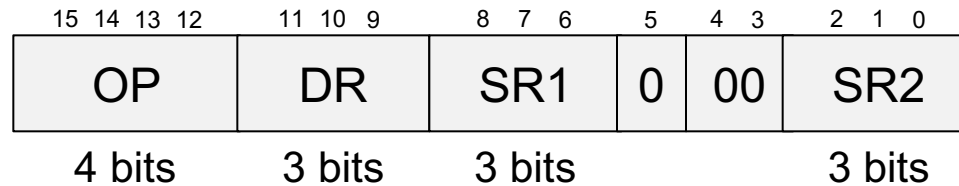
There is **no NOT in MIPS**. How is it implemented?

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:** $DR \leftarrow SR1 + SR2$

■ E.g., AND = 0101

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

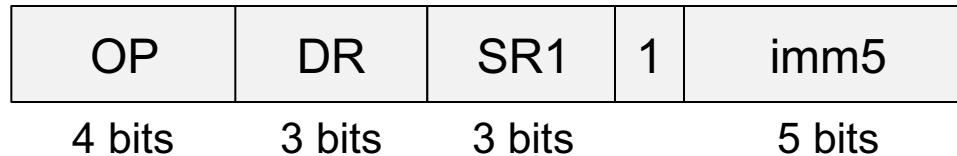
□ SR1, SR2 = source registers

□ 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					

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)

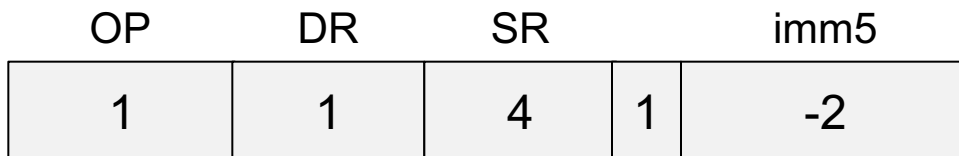
ADD with one Literal in LC-3

ADD assembly and machine code

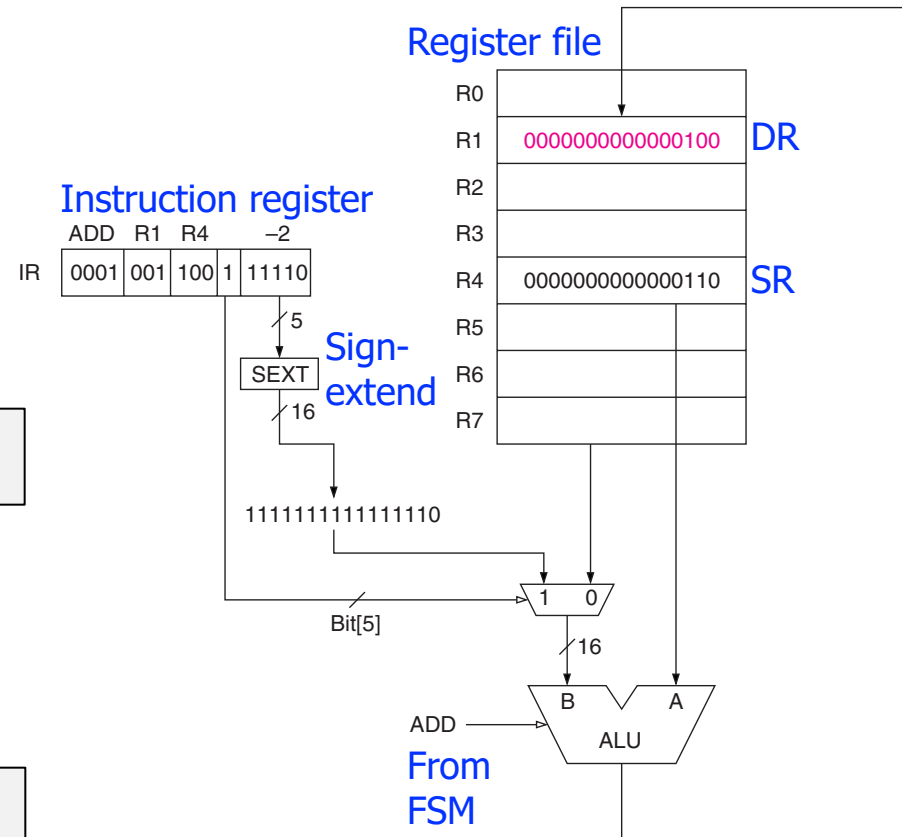
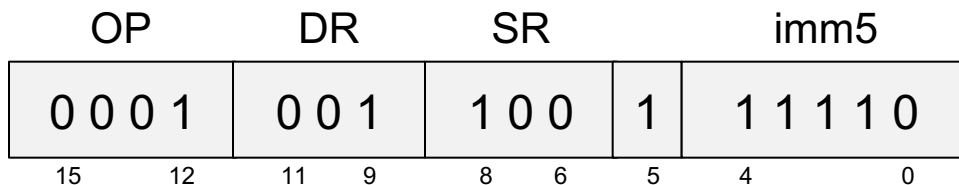
LC-3 assembly

```
ADD R1, R4, #-2
```

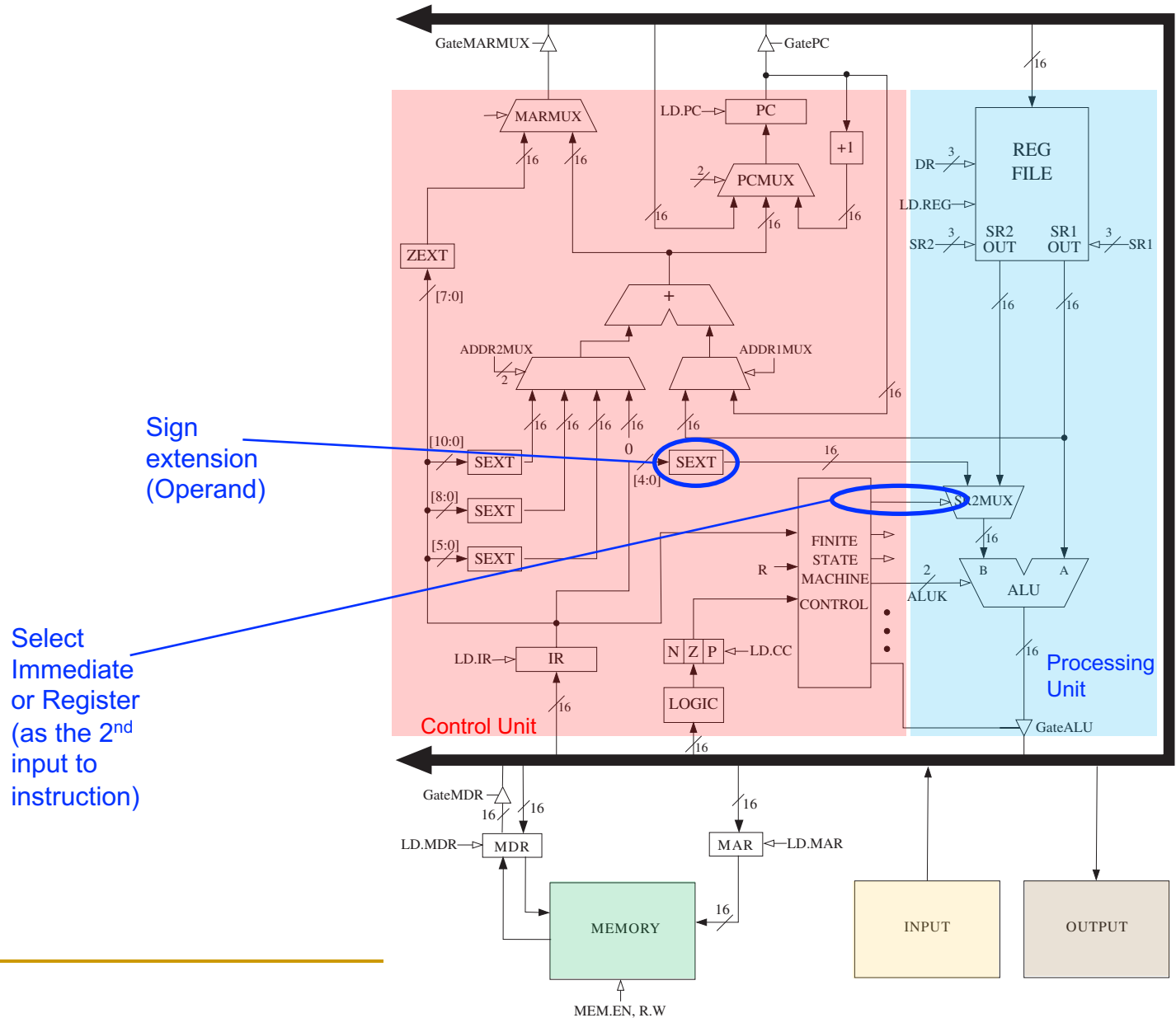
Field Values



Machine Code

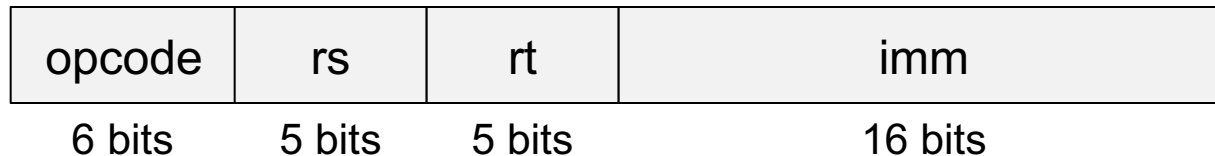


ADD with one Literal in LC-3 Data Path



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
8	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

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
```

2's complement of R3

■ Tradeoff in LC-3

- More instructions
- But, simpler control logic

Subtract Immediate

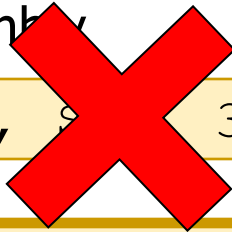
- MIPS assembly

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
```

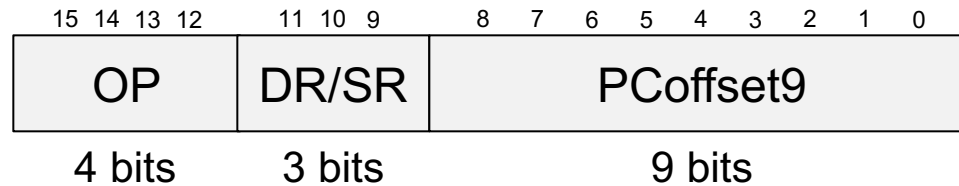
Data Movement Instructions and Addressing Modes

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

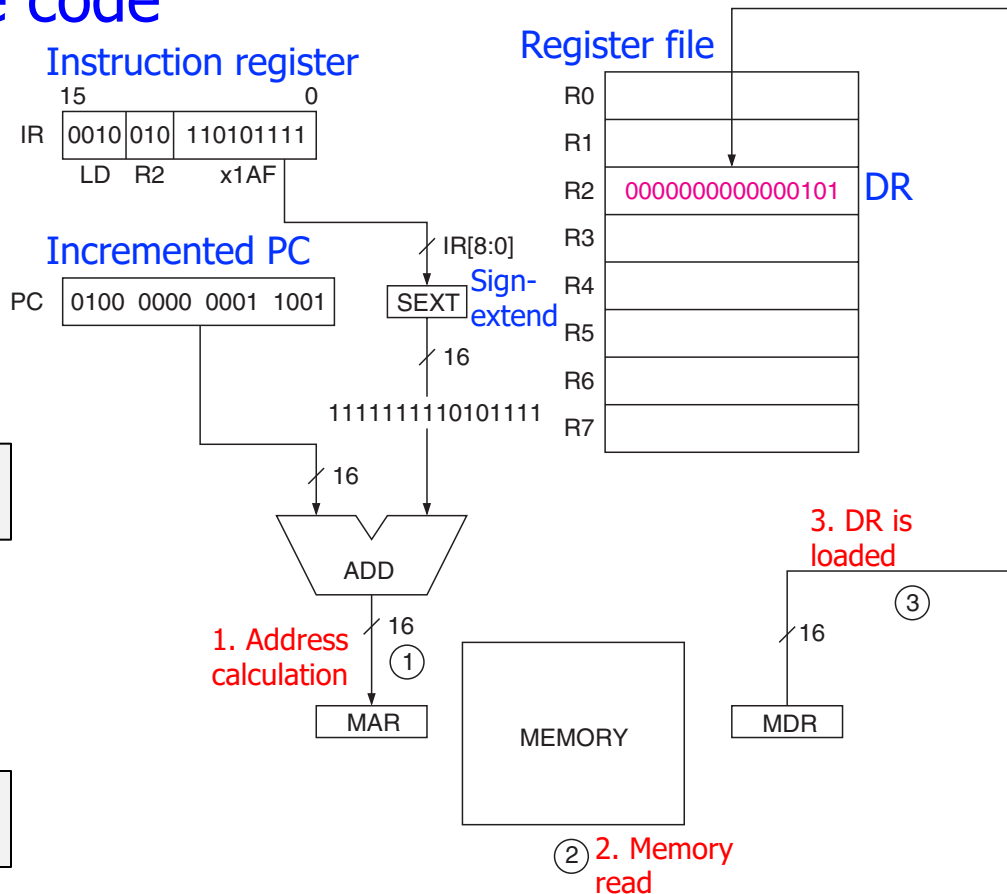
Field Values

OP	DR	PCoffset9
2	2	0x1AF

Machine Code

OP	DR	PCoffset9
0010	010	110101111

15 12 11 9 8 0

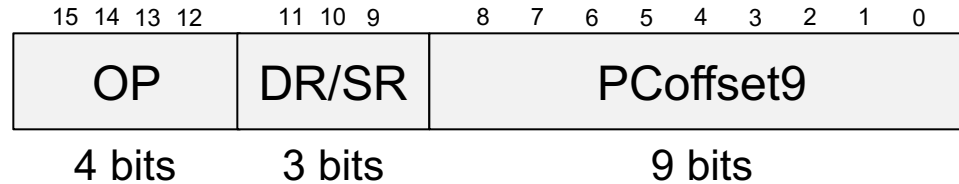


The memory address is **only +255 to -256** locations away of the **LD or ST instruction**

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: $DR \leftarrow \text{Memory}[\text{Memory}[\text{PC}^\dagger + \text{sign-extend}(\text{PCOffset9})]]$
- STI: $\text{Memory}[\text{Memory}[\text{PC}^\dagger + \text{sign-extend}(\text{PCOffset9})]] \leftarrow SR$

[†] This is the incremented PC

LDI in LC-3

LDI assembly and machine code

LC-3 assembly

```
LDI R3, 0x1CC
```

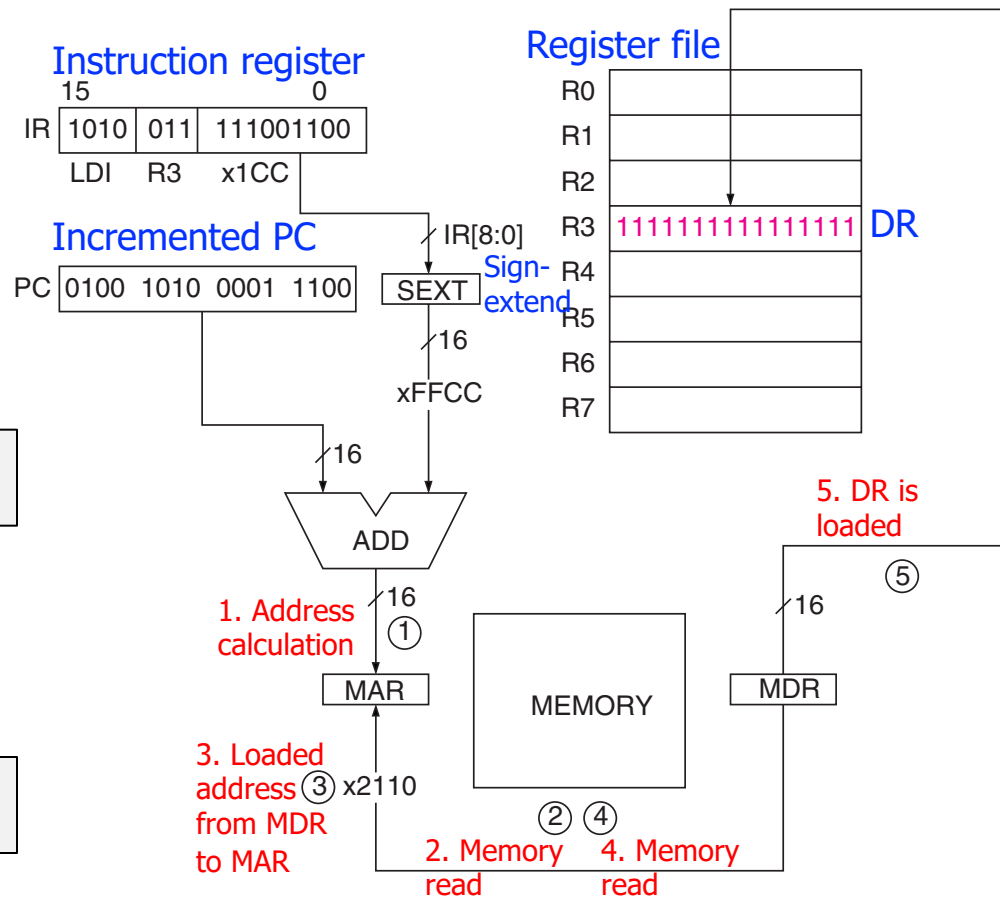
Field Values

OP	DR	PCoffset9
A	3	0x1CC

Machine Code

OP	DR	PCoffset9
1010	011	111001100

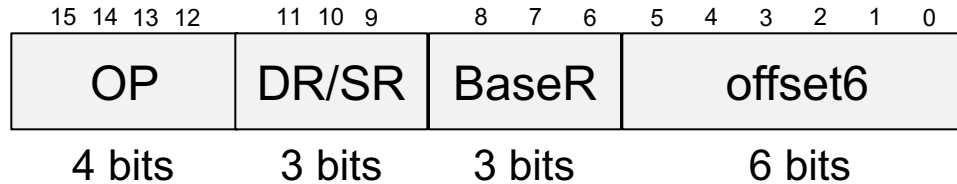
15 12 11 9 8 0



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
```

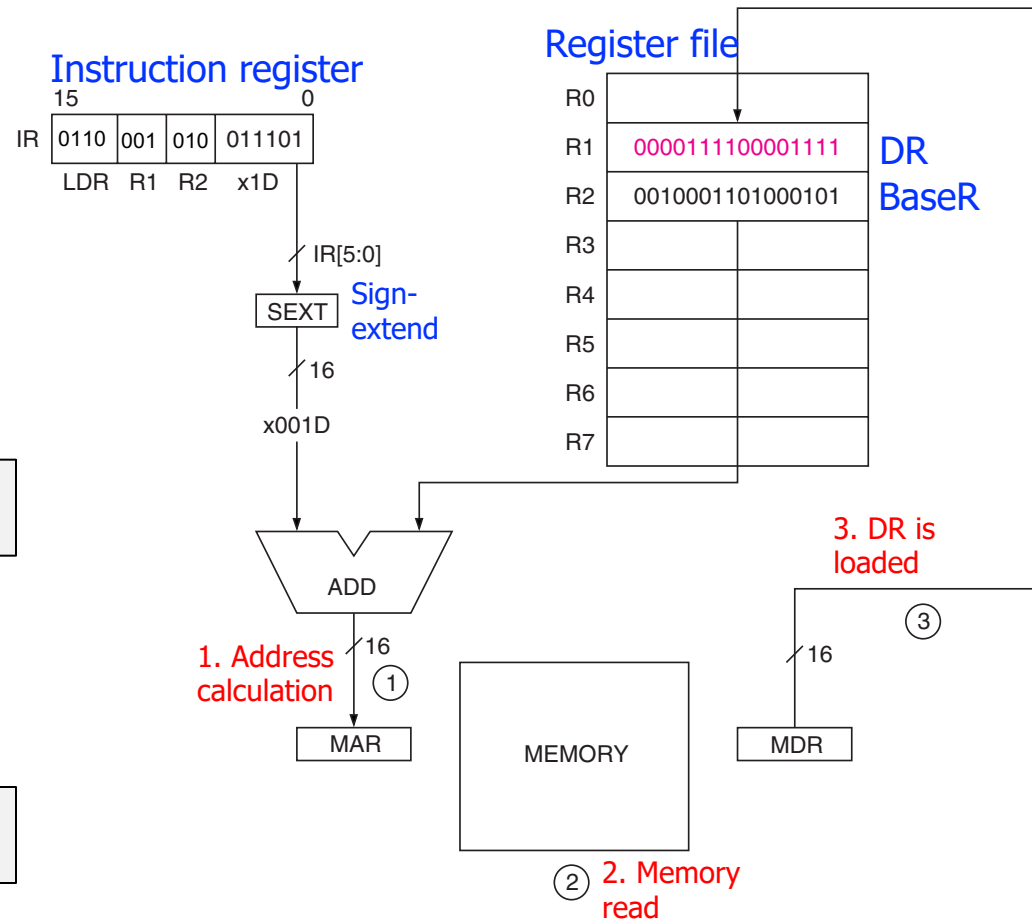
Field Values

OP	DR	BaseR	offset6
6	1	2	0x1D

Machine Code

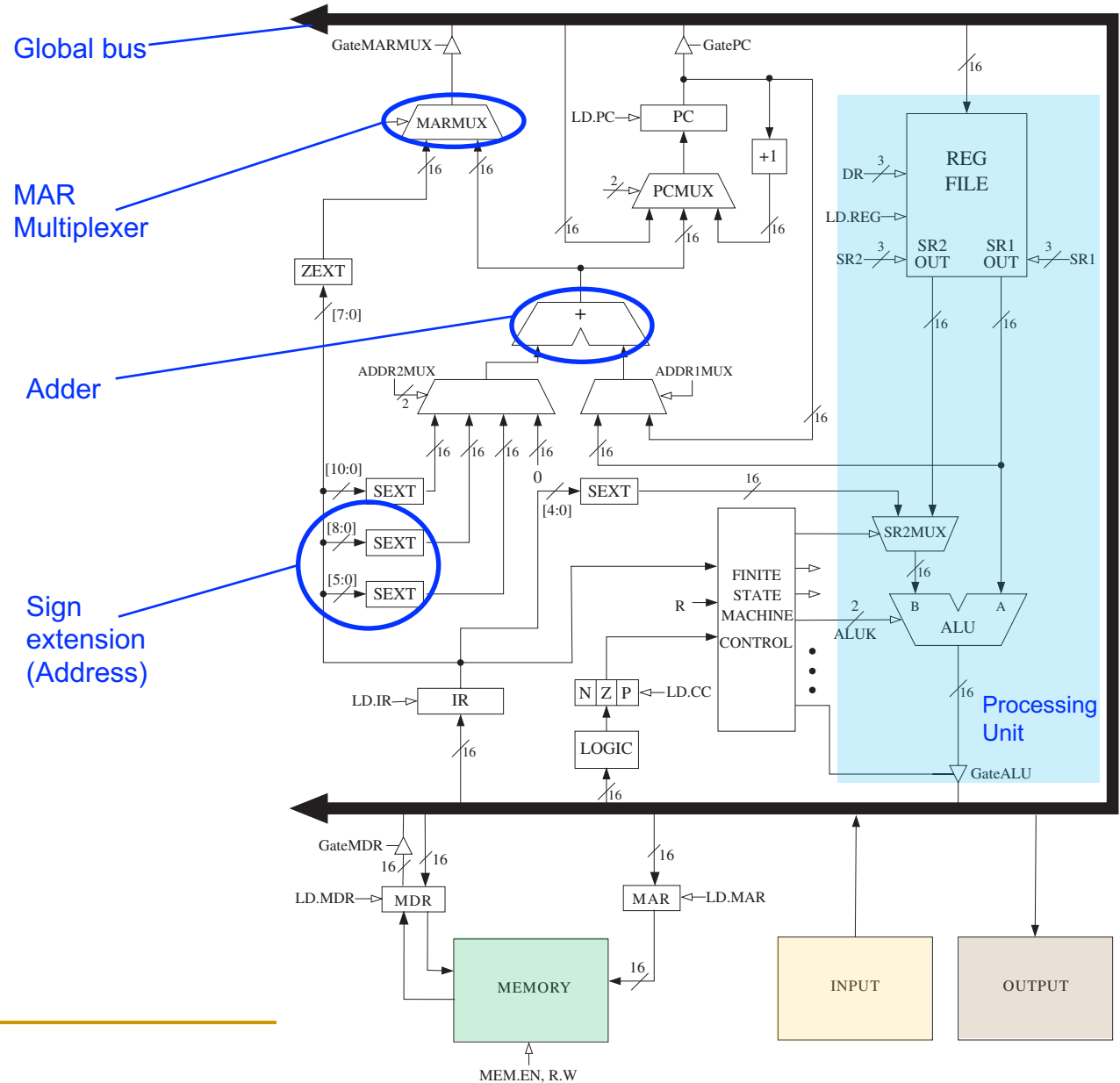
OP	DR	BaseR	offset6
0110	001	010	011101

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

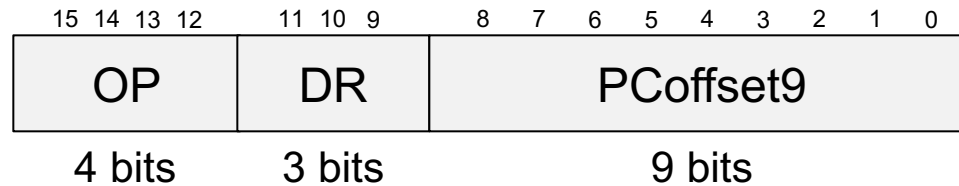
```
lw    $t0, 0($s0)  
add   $t1, $t0, $s2  
addi  $t2, $t1, -5  
sw    $t2, 0($s1)
```

LC-3 assembly

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

Immediate Addressing Mode (in LC-3)

■ 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

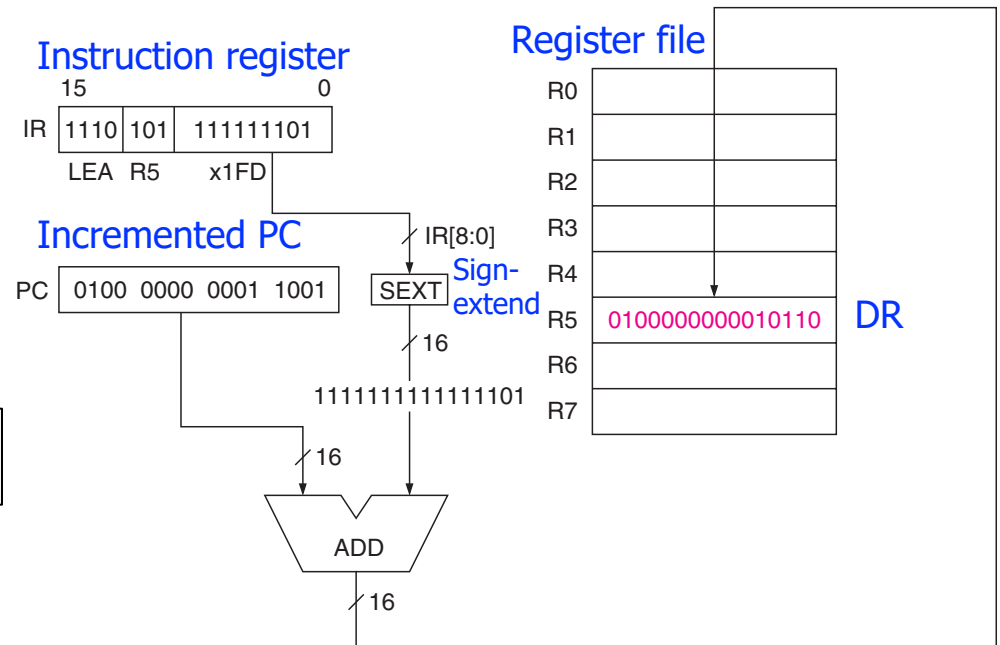
```
LEA R5, #-3
```

Field Values

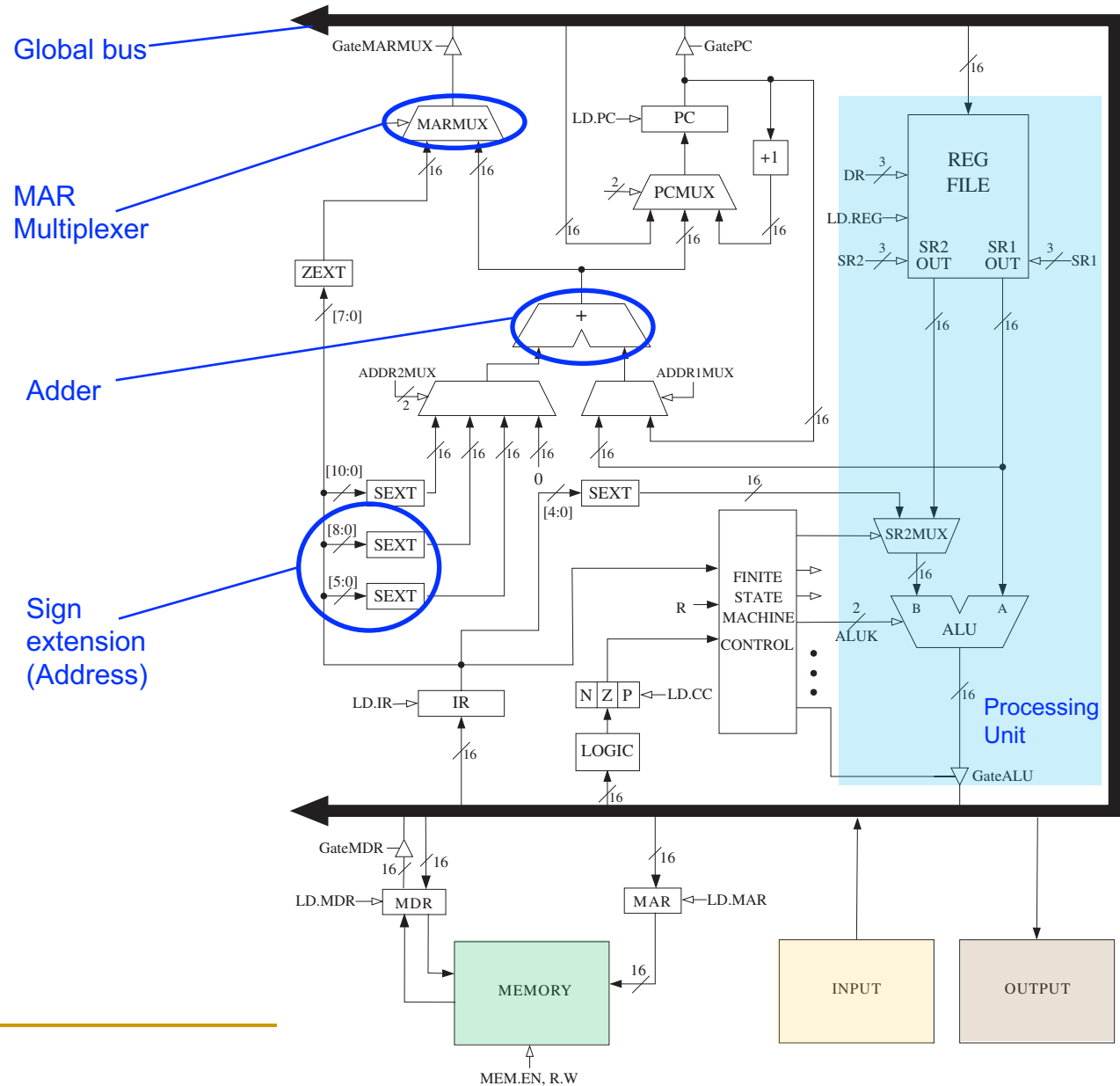
OP	DR	PCoffset9
E	5	0x1FD

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	LEA	1	1	0	0	0	1	3	1	1	1	1	1	1	1	0	R1 = PC - 3 = 0x30F7 - 3 = 0x30F4
x30F7	ADD	0	0	1	0	1	0	0	0	1	1	0	1	1	0	1	R2 = R1 + 14 = 0x30F4 + 14 = 0x3102
x30F8	ST	0	1	1	0	1	0	5	1	1	1	1	1	0	1	0	M[PC - 5] = M[0x030F4] = 0x3102
x30F9	AND	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 = 0
x30FA	ADD	0	0	1	0	1	0	0	1	0	1	5	0	1	0	1	R2 = R2 + 5 = 5
x30FB	STR	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	LDI	0	1	0	0	1	1	9	1	1	1	1	1	1	1	1	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
 - We have already seen these

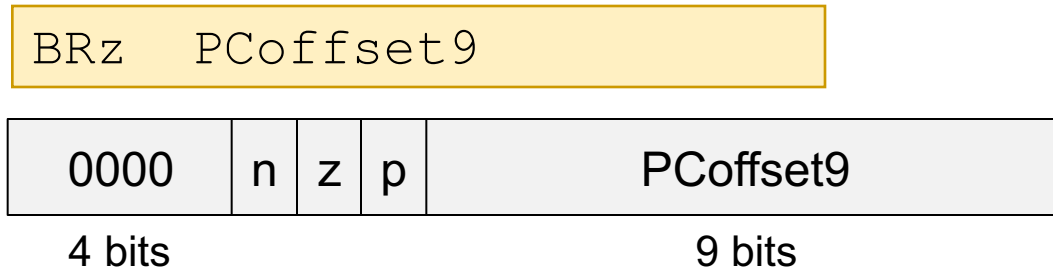
Conditional Control Flow (Conditional Branching)

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)



- $n, z, p =$ **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

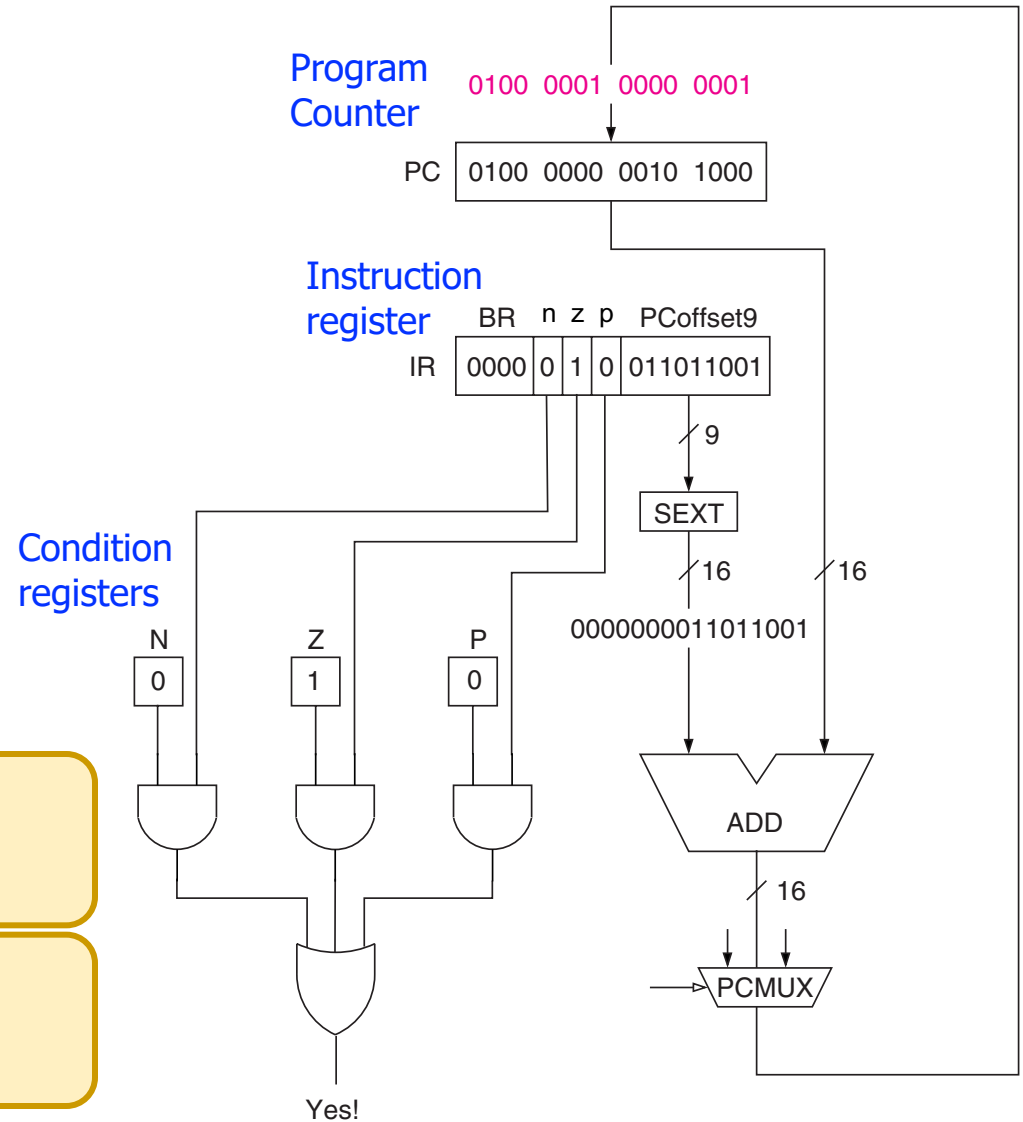
Conditional Branches in LC-3

BRz

BRz 0x0D9

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

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

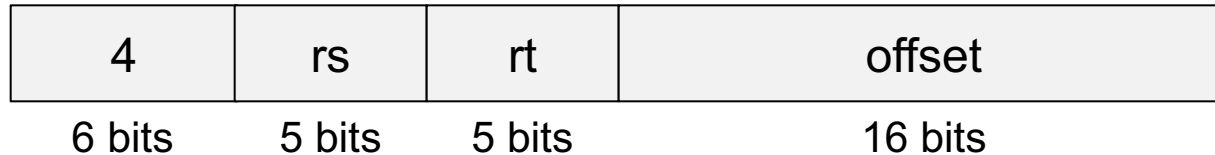


*n, z, p are the instruction bits to identify the condition codes to be tested

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
```

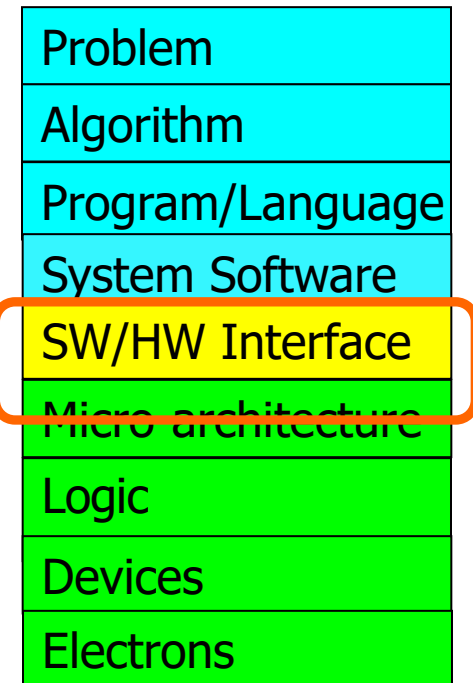
```
BRz offset
```

**Subtract
(R0 - R1)**

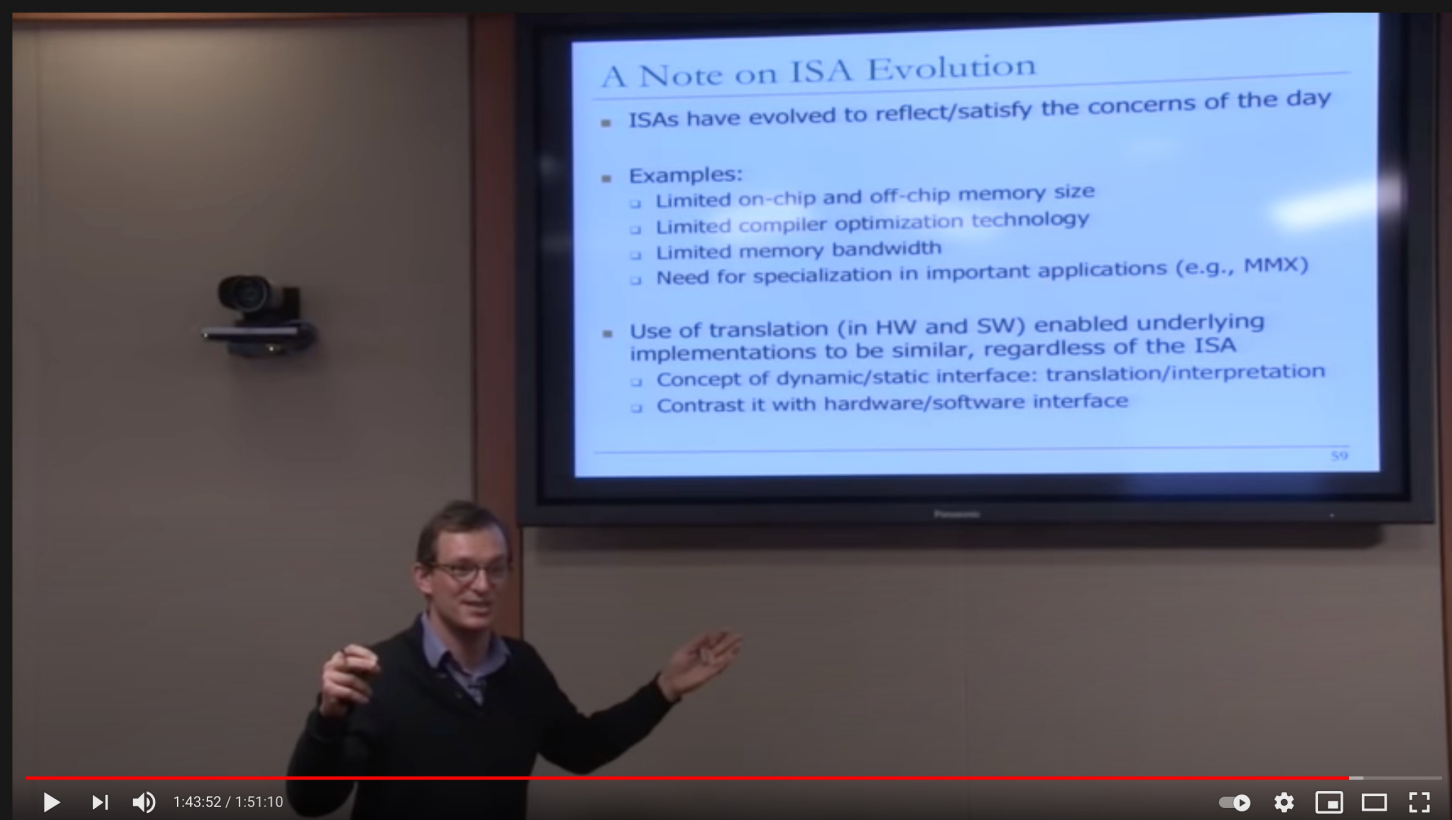
- This is an example of **tradeoff** in the instruction set
 - The same functionality requires **more instructions in LC-3**
 - But, the **control logic** requires **more complexity in MIPS**

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

44,973 views • Jan 24, 2015

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

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

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

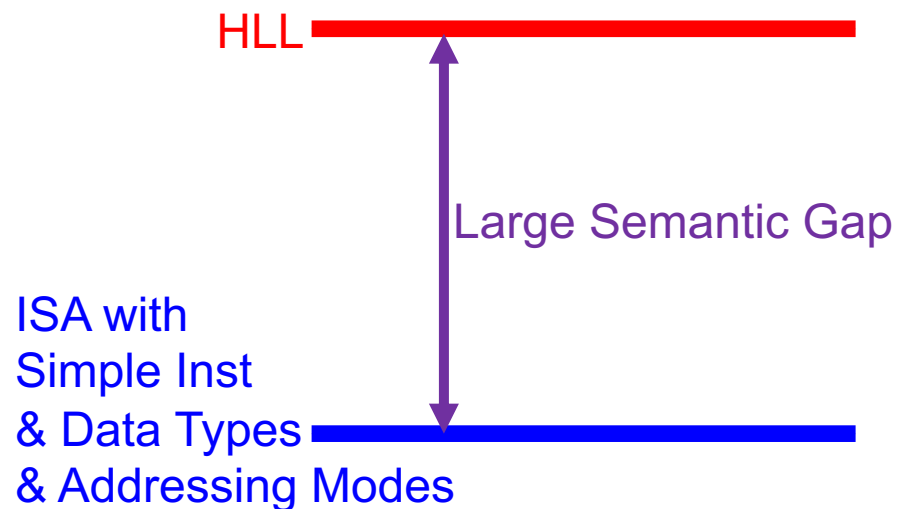
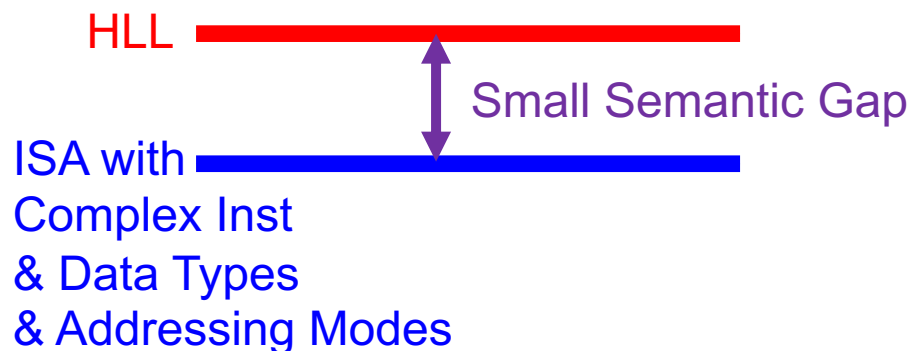
- **Complex instruction:** An instruction **does a lot of work**, e.g. many operations
 - ❑ Insert in a doubly linked list
 - ❑ Compute FFT
 - ❑ String copy
 - ❑ Matrix multiply
 - ❑ ...
- **Simple instruction:** An instruction **does little work** -- it is a primitive using which complex operations 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)



HW 
Control
Signals

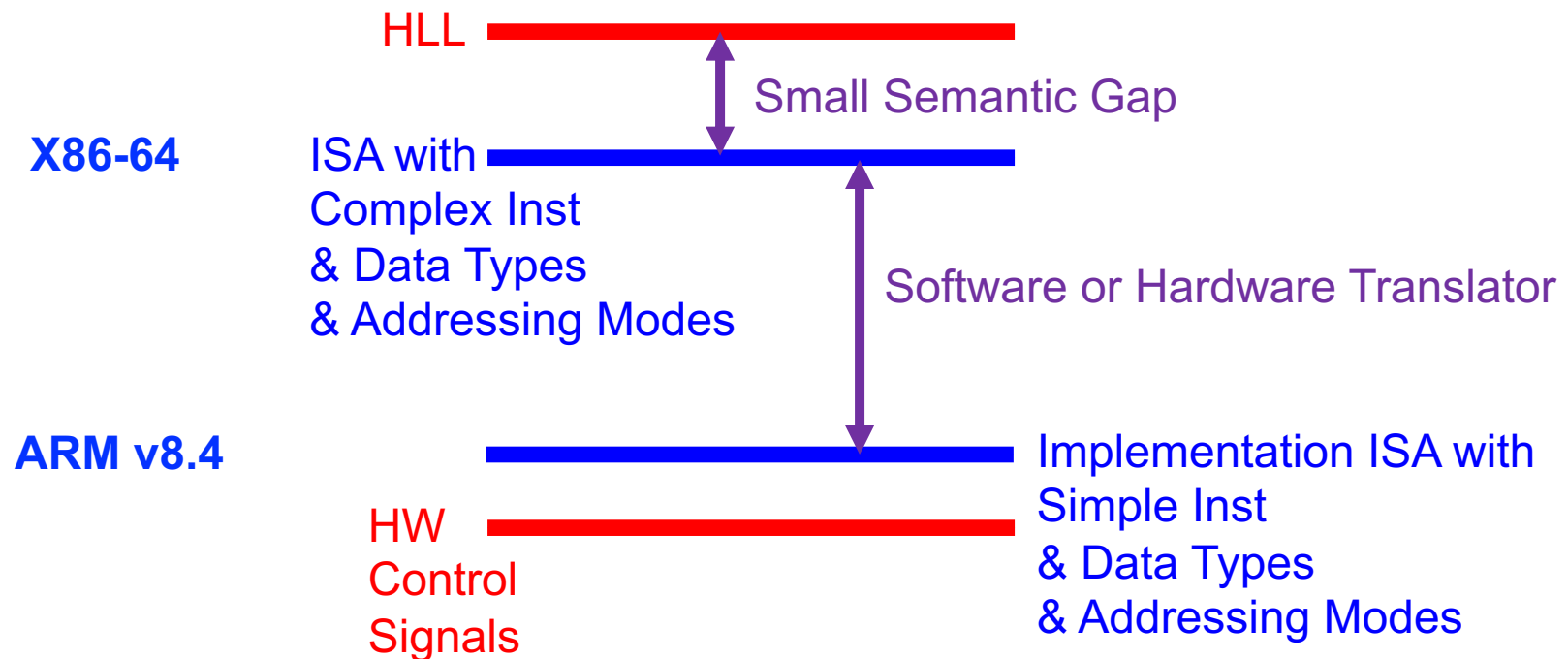
HW 
Control
Signals

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



An Example: Rosetta 2 Binary Translator

Rosetta 2 [\[edit \]](#)

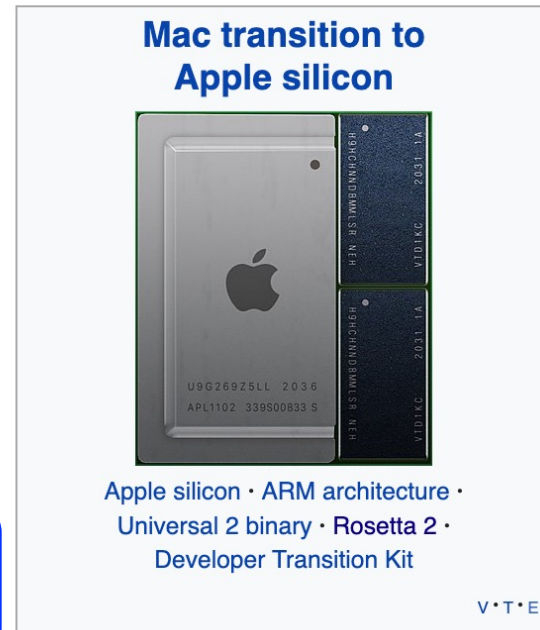
In 2020, Apple announced Rosetta 2 would be bundled with macOS Big Sur, to aid in the [Mac transition to Apple silicon](#). The software permits many applications compiled exclusively for execution on x86-64-based processors to be translated for execution on Apple silicon.^{[2][8]}

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]

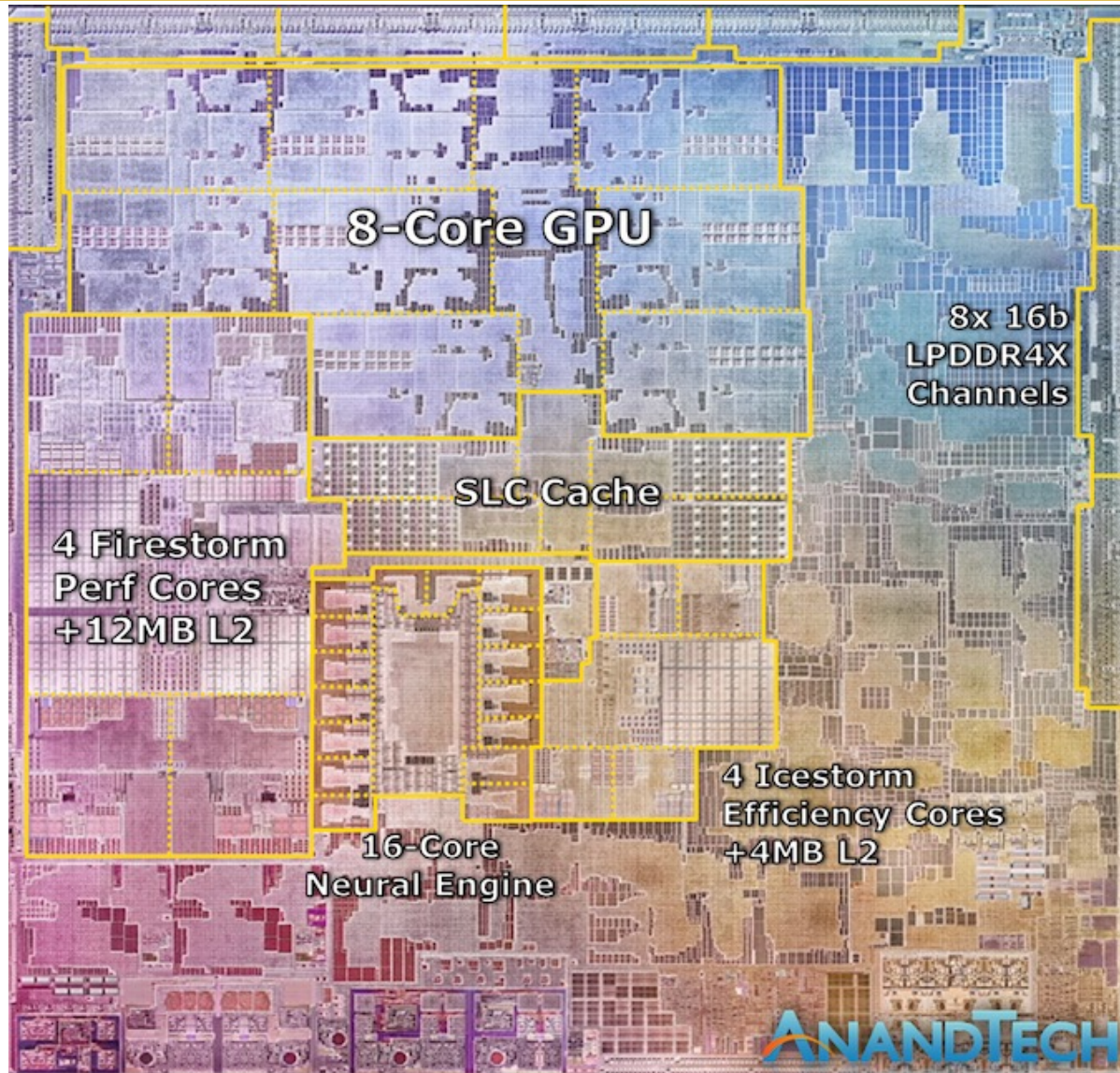
Rosetta 2's performance has been praised greatly.^{[10][11]} In some benchmarks, x86-64-only programs performed better under Rosetta 2 on a Mac with an Apple M1 SOC than natively on a Mac with an Intel x86-64 processor. One of the key reasons why Rosetta 2 provides such high level of translation efficiency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

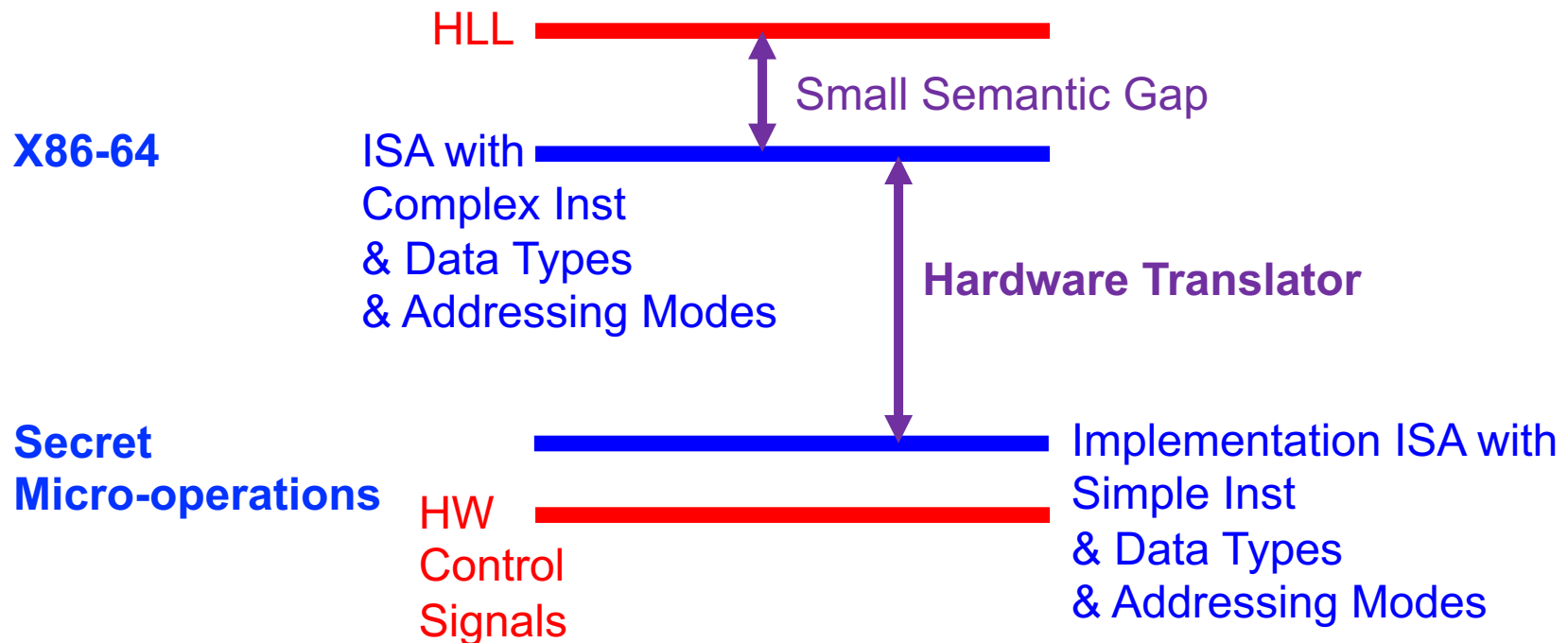


An Example: Rosetta 2 Binary Translator



Apple M1,
2021

Another Example: Intel and AMD Processors



Another Example: Intel and AMD Processors



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 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

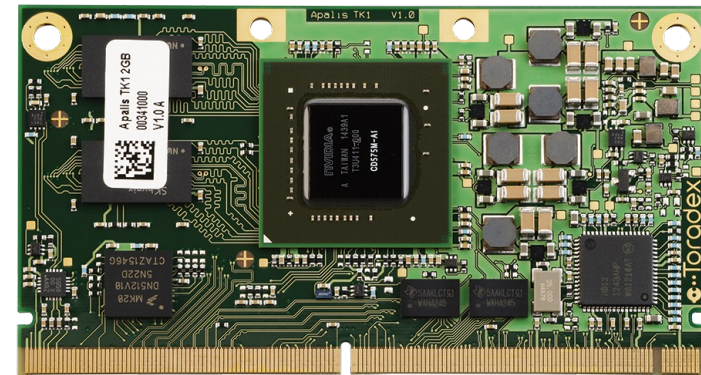
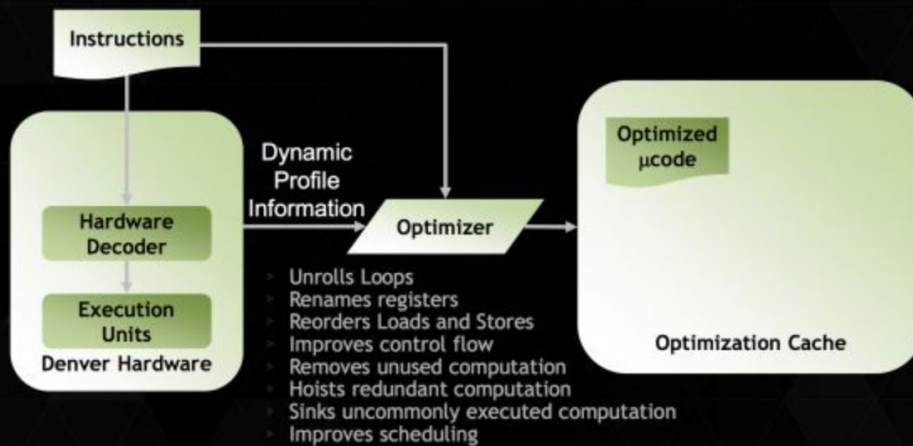
Another Example: NVIDIA Denver

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.

DYNAMIC CODE OPTIMIZATION OPTIMIZE ONCE, USE MANY TIMES



Transmeta: x86 to VLIW Translation

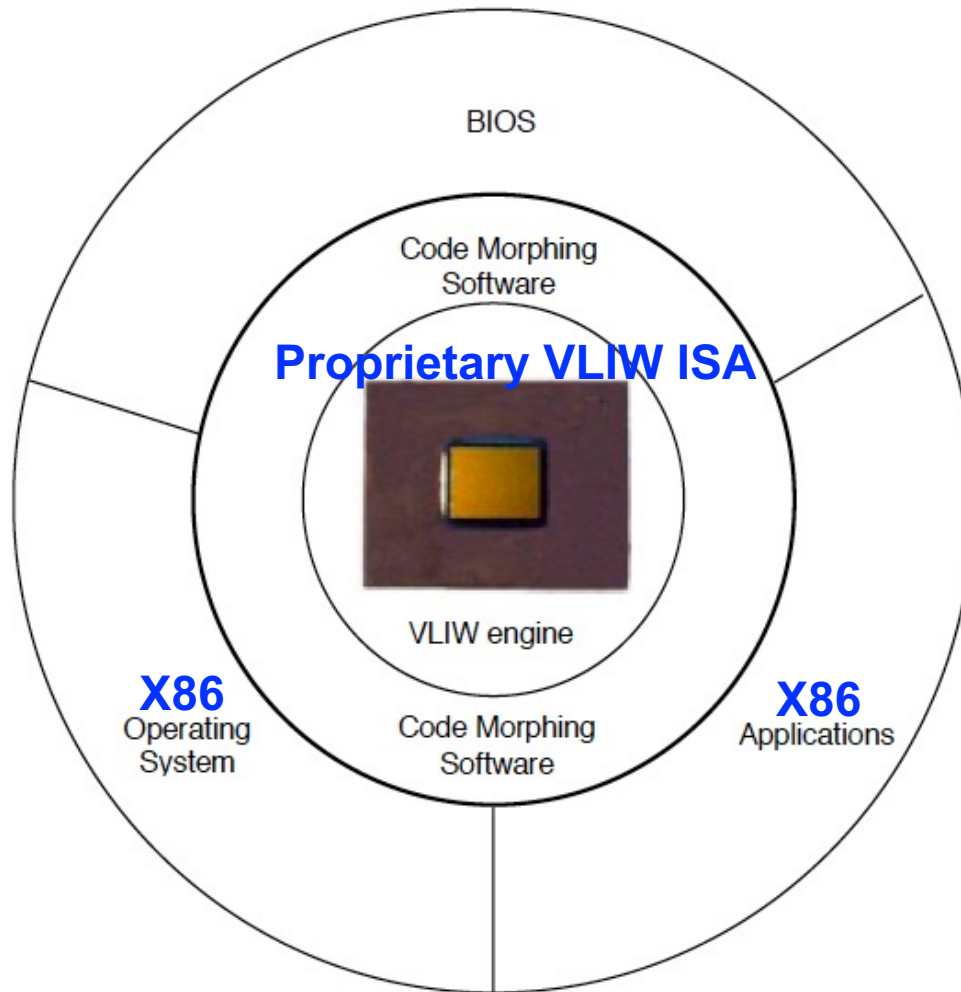


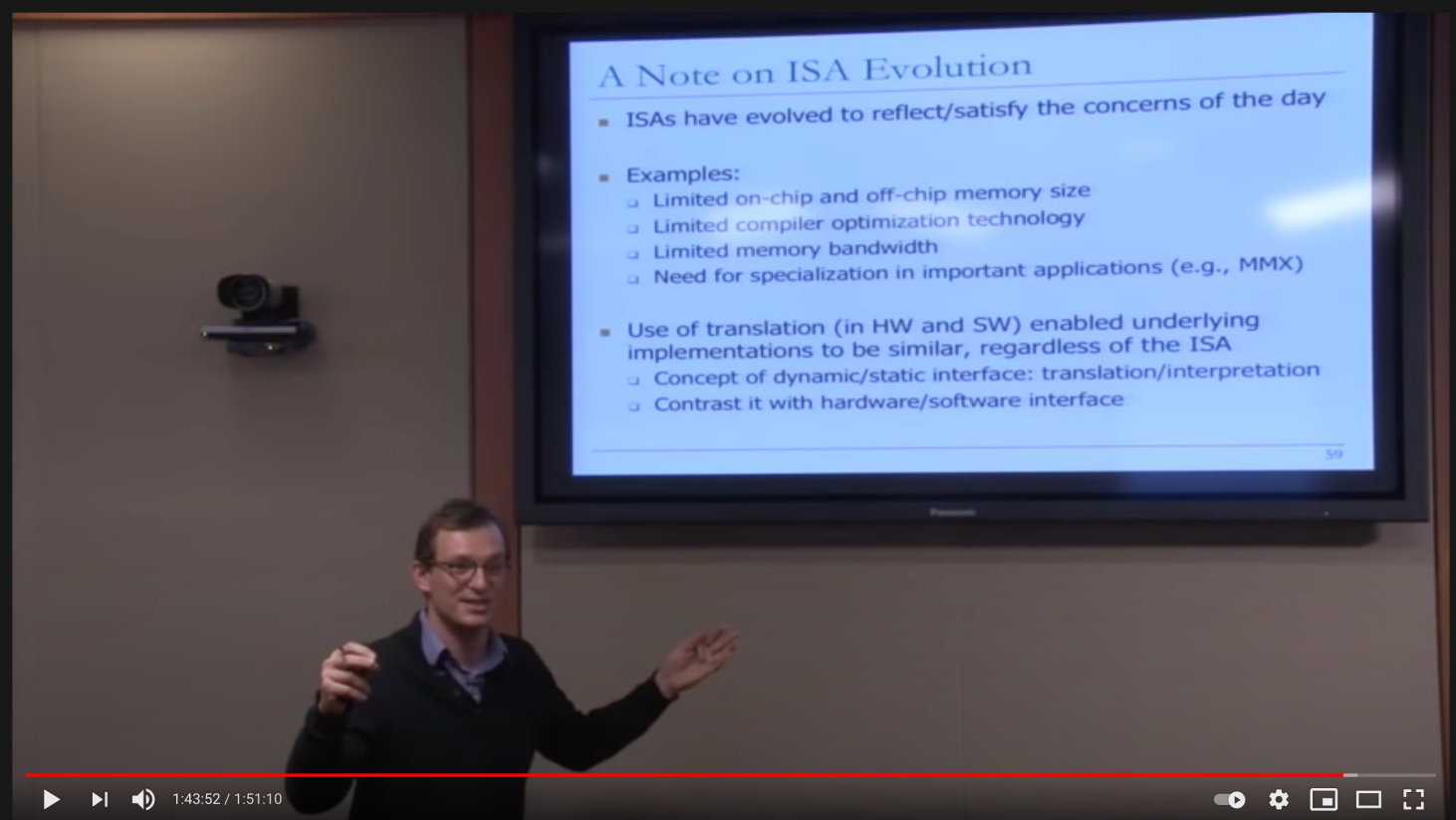
Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.

Klaiber, “[The Technology Behind Crusoe Processors](#),” Transmeta White Paper 2000.

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

276 likes 5 dislikes SHARE SAVE ...

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

ANALYTICS EDIT VIDEO

There Is A Lot More to Cover on ISAs

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

25:29 / 1:30:28



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

ANALYTICS EDIT VIDEO

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-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>

ISA Design and Tradeoffs: More Critical Thinking

The Von Neumann Model/Architecture

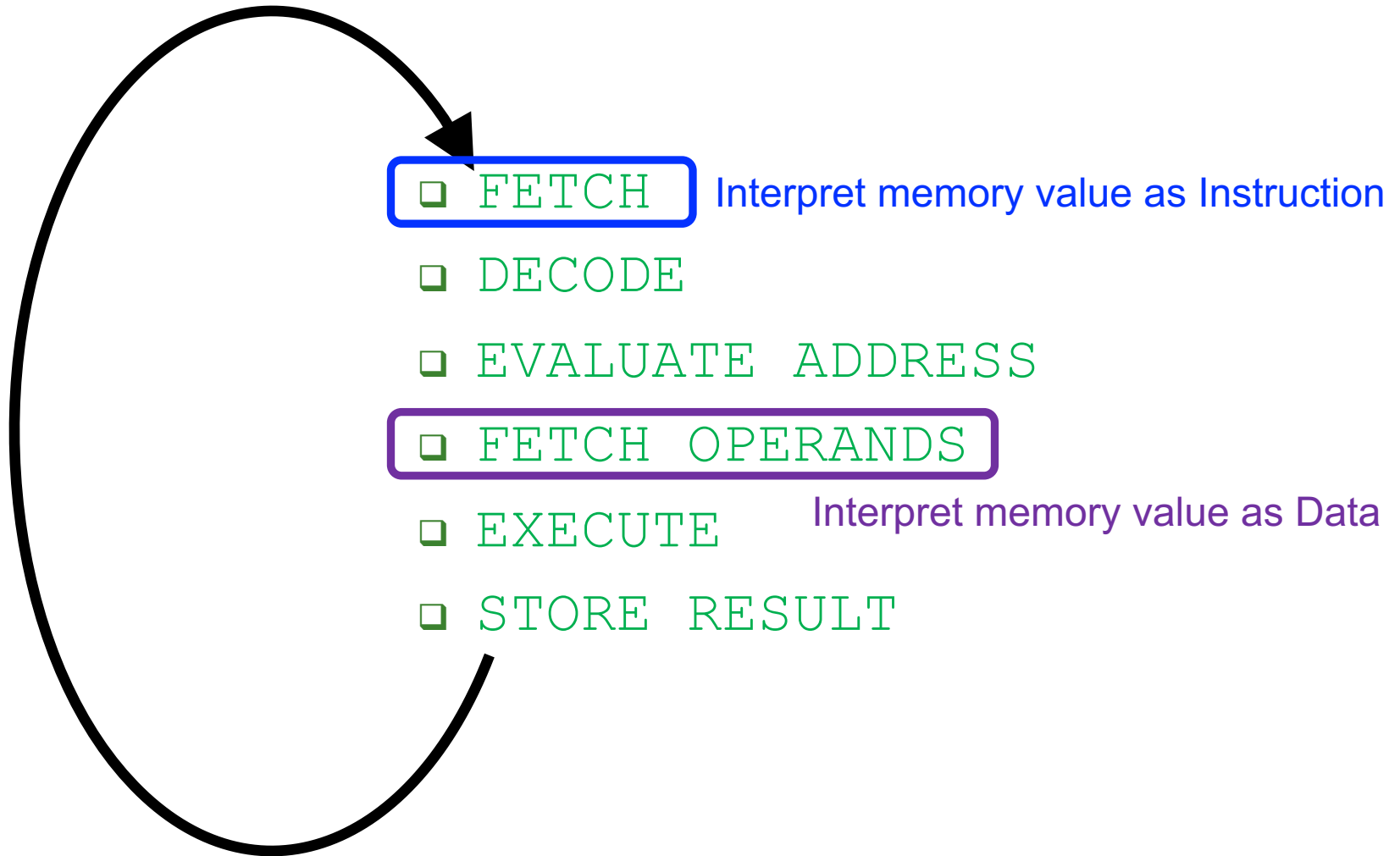
Stored program

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



Whether a value fetched from memory is interpreted as an instruction depends on **when** that value is **fetched** in the instruction processing cycle.

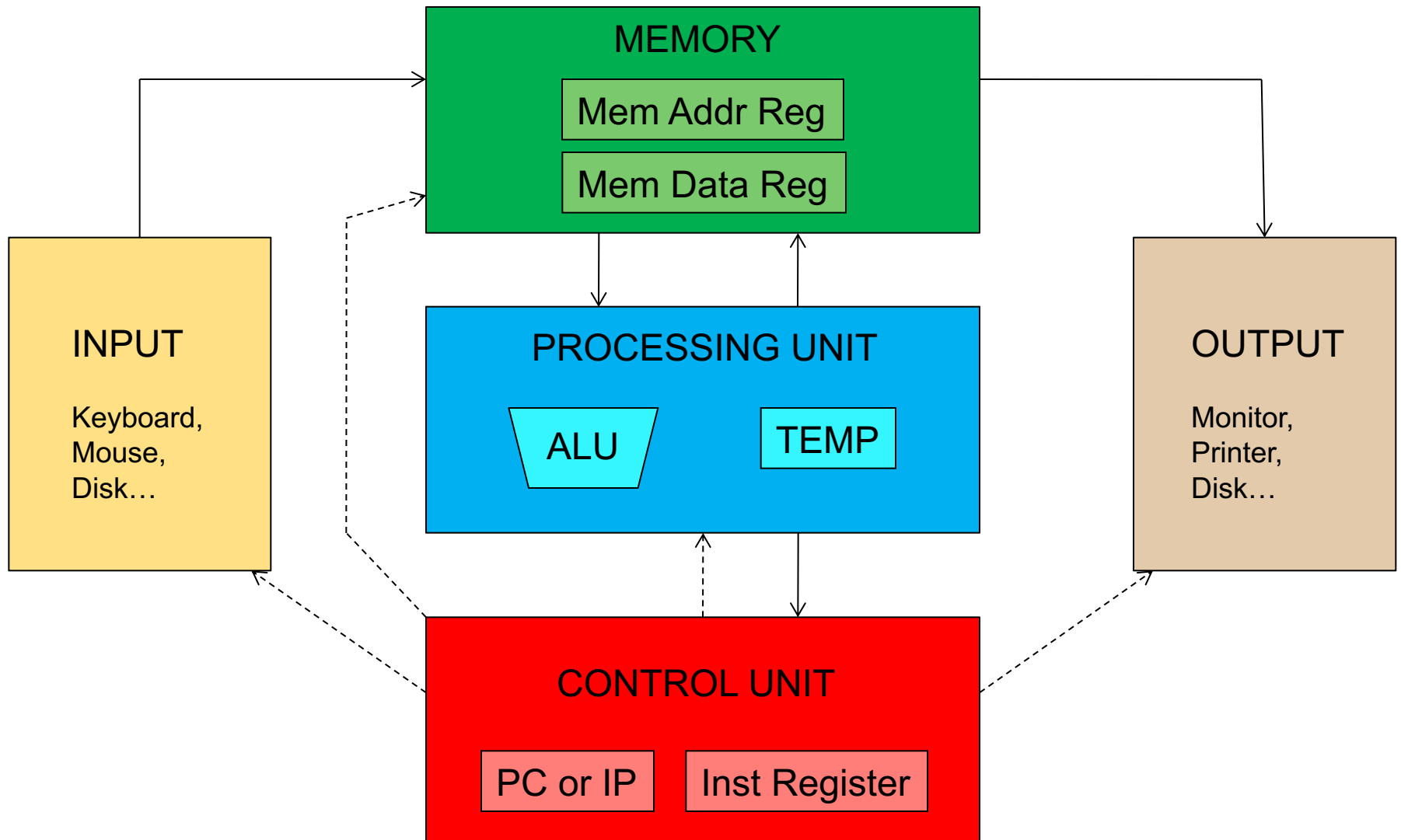
The von Neumann Model/Architecture

- 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**
When is a value interpreted as an instruction?
- **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

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)



The Von Neumann Model (of a Computer)

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

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

14

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

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

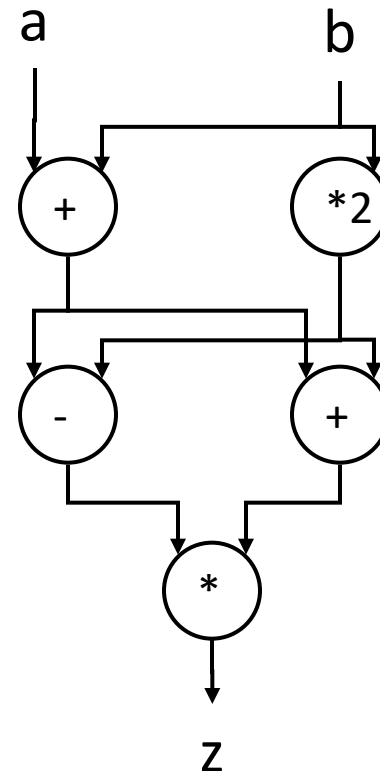
Von Neumann vs. Dataflow

- Consider a Von Neumann program
 - What is the significance of the program order?
 - What is the significance of the storage locations?

v = a + b;
w = b * 2;
x = v - w
y = v + w
z = x * y

Sequential

a, b are the only inputs
z is the only output

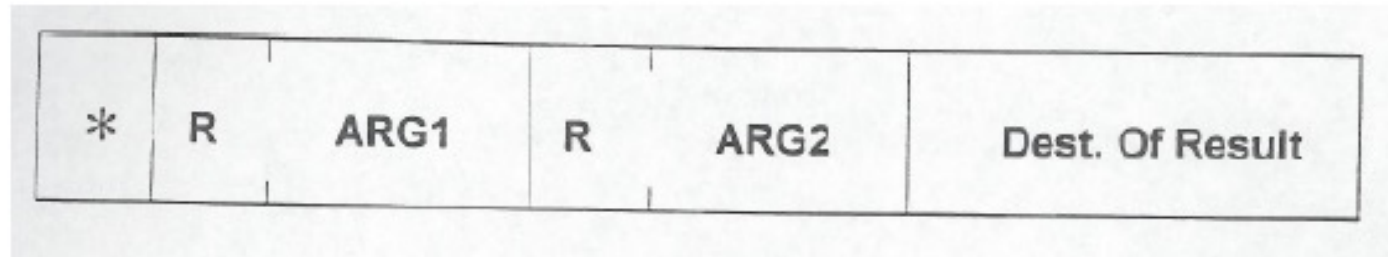
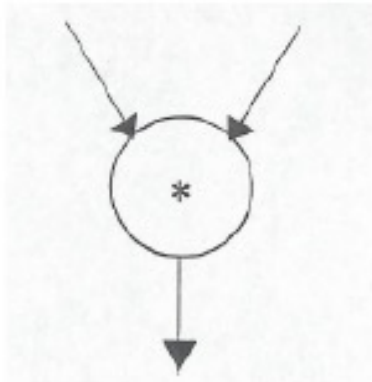


Dataflow

Which model is more natural to you as a programmer?

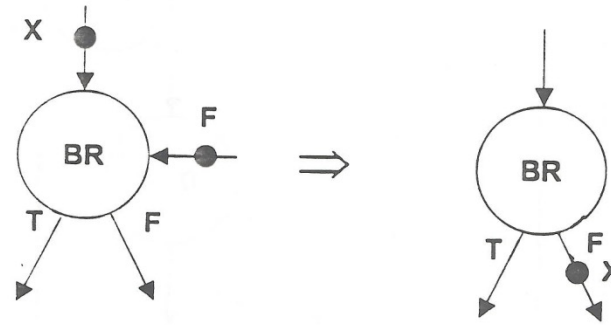
More on Dataflow

- 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

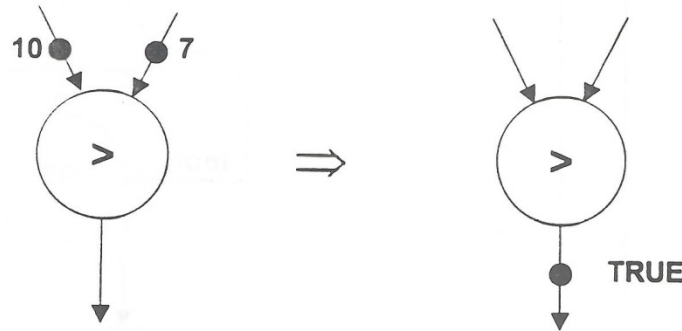


Example Dataflow Nodes

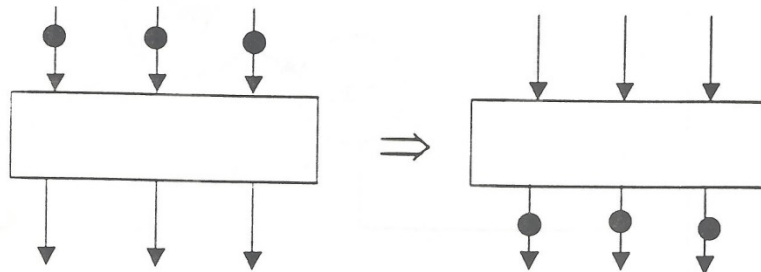
***Conditional**



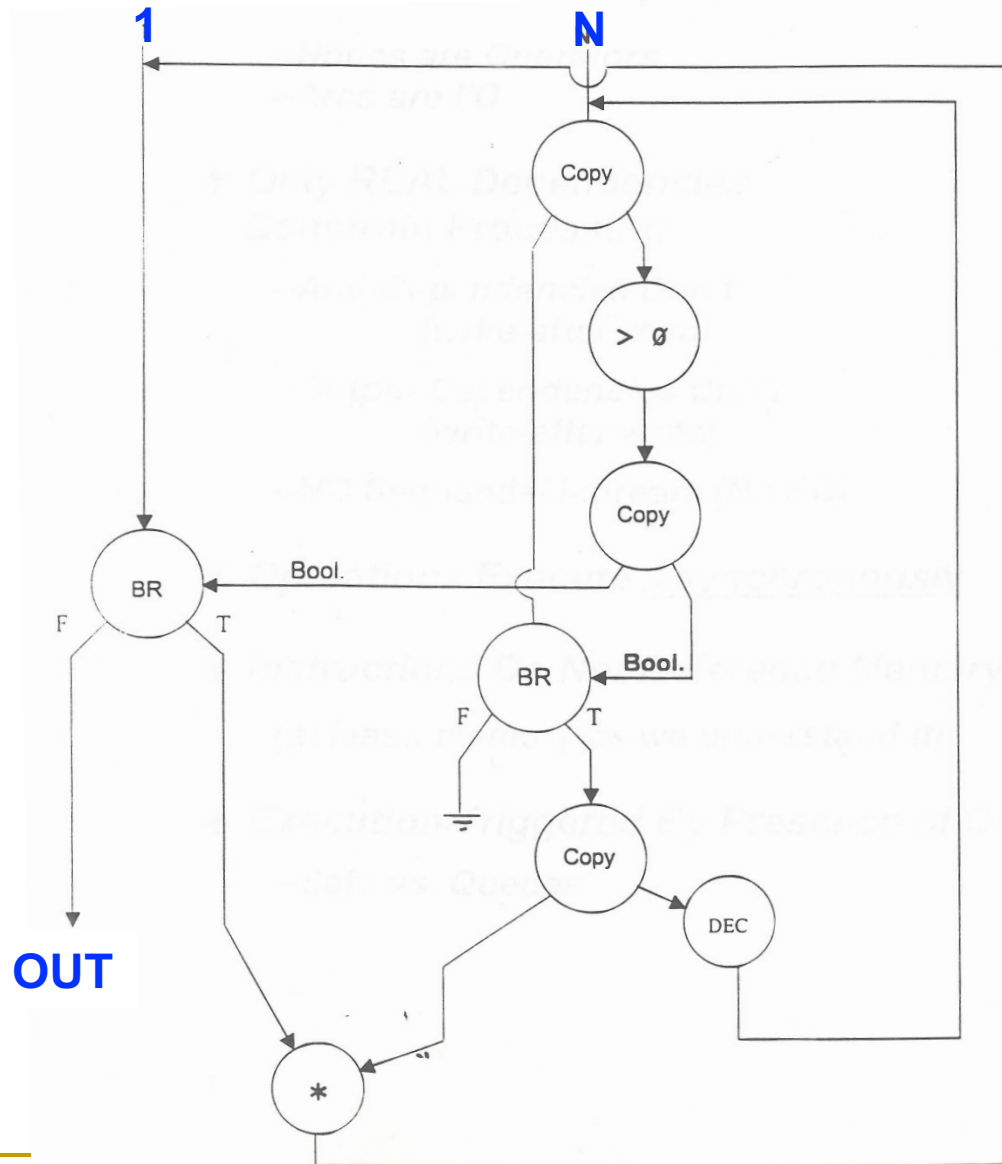
***Relational**



***Barrier Synchron**



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...
- Dennis and Misunas, "A preliminary architecture for a basic data-flow processor," ISCA 1974.
- Gurd et al., "The Manchester prototype dataflow computer," CACM 1985.
- A later lecture
- If you are really impatient:
 - <http://www.youtube.com/watch?v=D2uue7izU2c>
 - <http://www.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=onur-740-fall13-module5.2.1-dataflow-part1.ppt>

Lecture Video on Dataflow Architectures

Loops and Function Calls Summary

Figure 11. Interface for a procedure call. On the left a call of procedure P whose graph is on the right. P has one parameter and one return value. The actual parameter receives a new tag and is sent to the input node of P and concurrently a token containing address A is sent to the output node. This SEND-TO-DESTINATION node transmits the other input tokens to a node of which the address is contained in the first token. The effect is that, when the return value of the procedure becomes available, the output node sends the result to node A, which restores the tag belonging to the calling expression.

```
while i <= 10  
do x := x + 1  
  y := y * 100  
od
```

Figure 12. An implementation of a loop using tagged tokens. At the start of the loop a new tag area is allocated. Tokens belonging to consecutive iterations arrive consecutive tags within this area. The tag from the loop is restored on tokens that exit from the loop.

```
while i <= 10  
do x := x + 1  
  y := y * 100  
od
```

Handwritten Chalkboard:

$$S + A[j] \times B[j]$$
$$\langle 10, 3, L, 100 \rangle$$
$$\langle 100, 3, R, 1000 \rangle$$

42:27 / 1:25:00

Carnegie Mellon - Parallel Computer Architecture 2012-Onur Mutlu - Lec 22 - Dataflow I

3,627 views · Apr 21, 2013

24 0 SHARE SAVE

Carnegie Mellon Computer Architecture
1.79K subscribers

SUBSCRIBED

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

Problem
Algorithm
Program
ISA
Microarchitecture
Circuits
Electrons

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** ...
 - ...

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

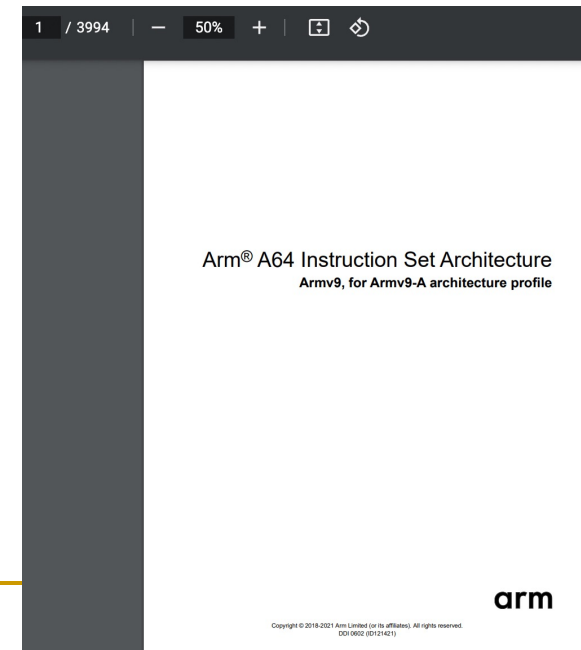
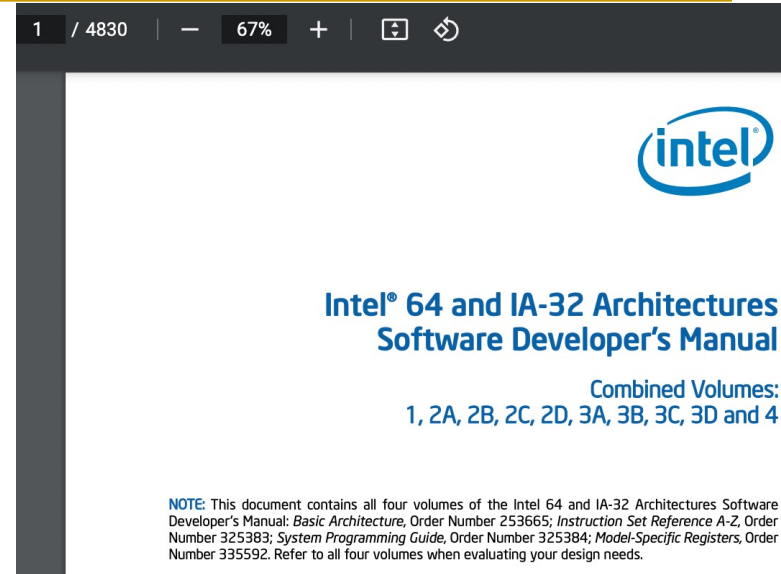
■ I/O: memory-mapped vs. instructions

■ 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

Document	Description
Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4	<p>This document contains the following:</p> <p>Volume 1: Describes the architecture and programming environment of processors supporting IA-32 and Intel® 64 architectures.</p> <p>Volume 2: Includes the full instruction set reference, A-Z. Describes the format of the instruction and provides reference pages for instructions.</p> <p>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/</p> <p>Volume 4: Describes the model-specific registers of processors supporting IA-32 and Intel® 64 architectures.</p>
Intel® 64 and IA-32 Architectures Software Developer's Manual Documentation Changes	<p>Describes bug fixes made to the Intel® 64 and IA-32 architectures software developer's manual between versions.</p> <p>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).</p>

ISA Manuals: Some Good Bedtime Reading



About RISC-V ▾ Membership ▾ RISC-V Exchange ▾ **Technical** ▾ News & Events ▾ Community ▾ 🔍

Specifications

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