KING'S College LONDON

Hardware Design

Lecture 1: Von Neumann Model & Instruction Set Architectures

Dr. Haiyu Mao

22.01.2026

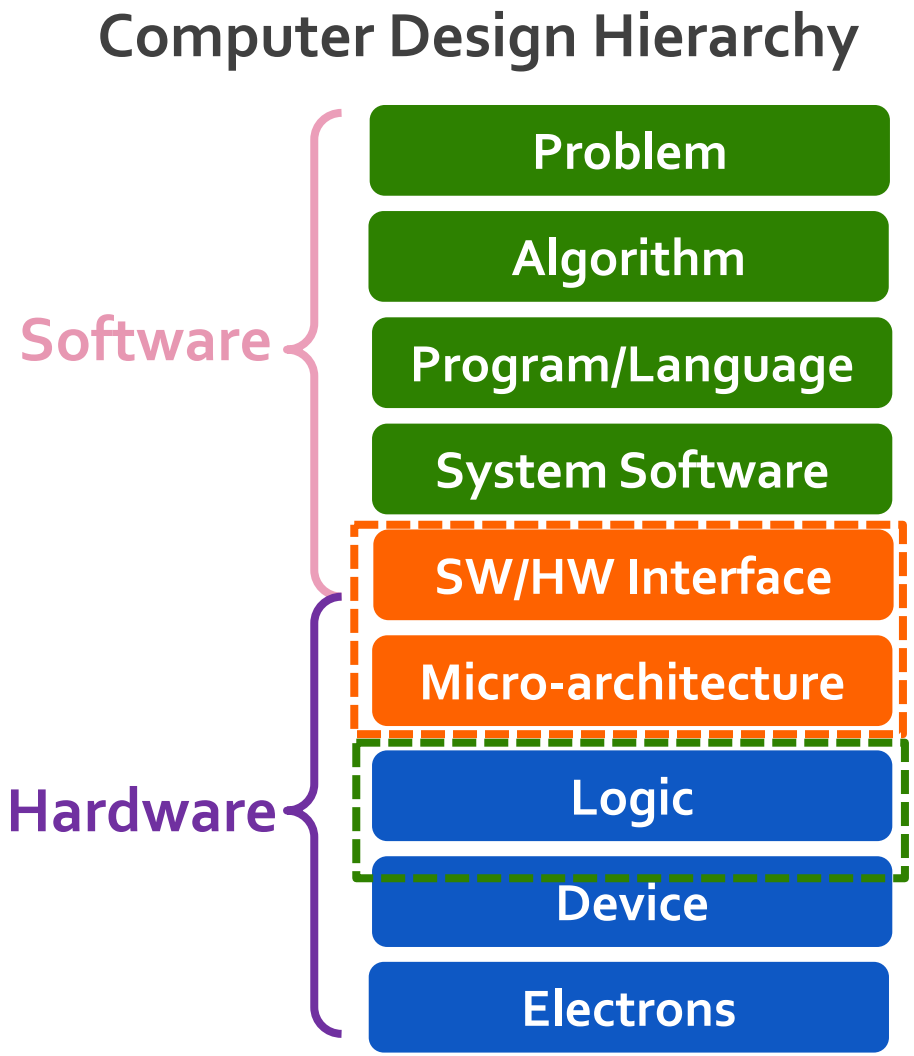
Recall: Digital Design in LOD

LOD is a prerequisite module for HAD.

- **Complementary Metal-Oxide-Semiconductor (CMOS)**: basic transistors
- **Logic Gates**: the CMOS building blocks of Boolean functions
- **Combinational Logic**: networks of logic gates
- **Sequential Logic**: combinational logic + memory, updated by clocks
- **Finite State Machine**: a state-based model to design/control sequential behavior (states, transitions, outputs)
- **Hardware Description Languages**: languages (e.g., Verilog) to describe, simulate, and synthesize digital circuits
- **Timing and Verification**: ensuring designs meet timing constraints and behave correctly via simulation, assertions, and testbenches

Agenda for Today & Next Few Lectures

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

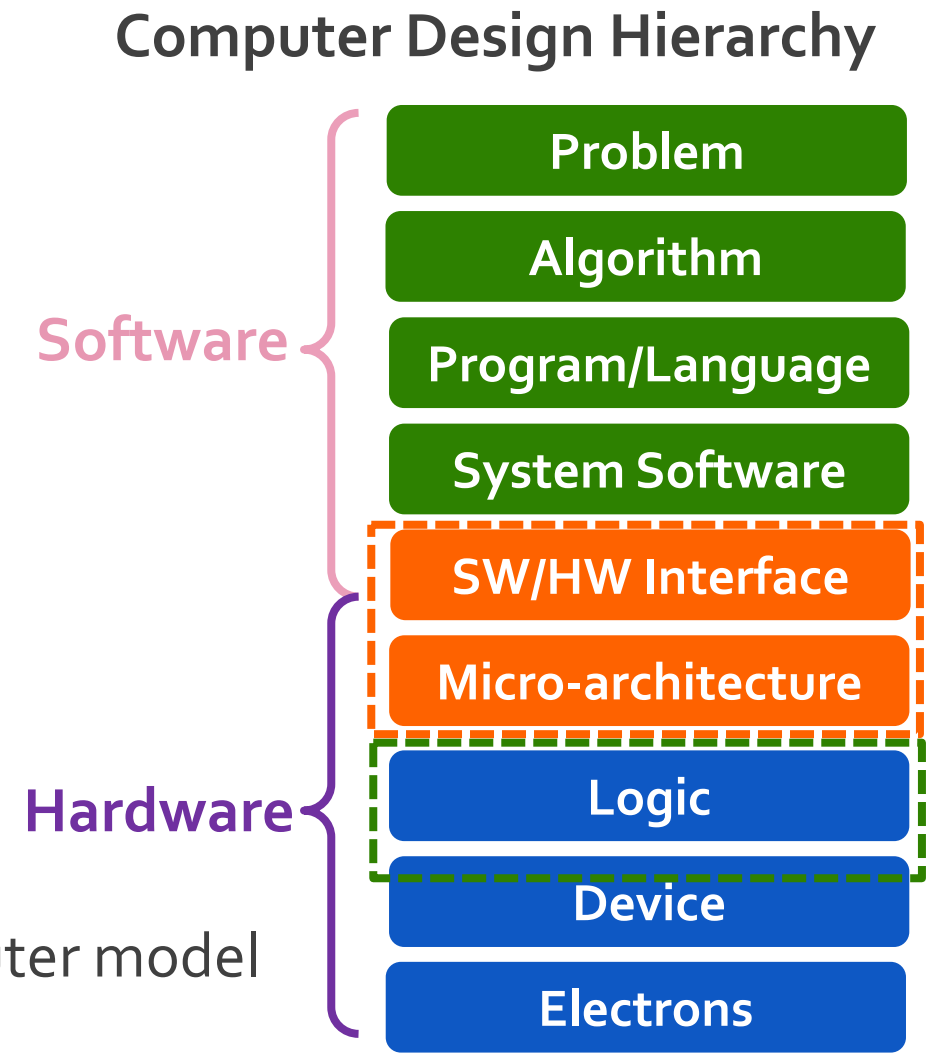


Hardware Design

The von Neumann Model

Building Up to A Basic Computer Model

- In LOD, 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

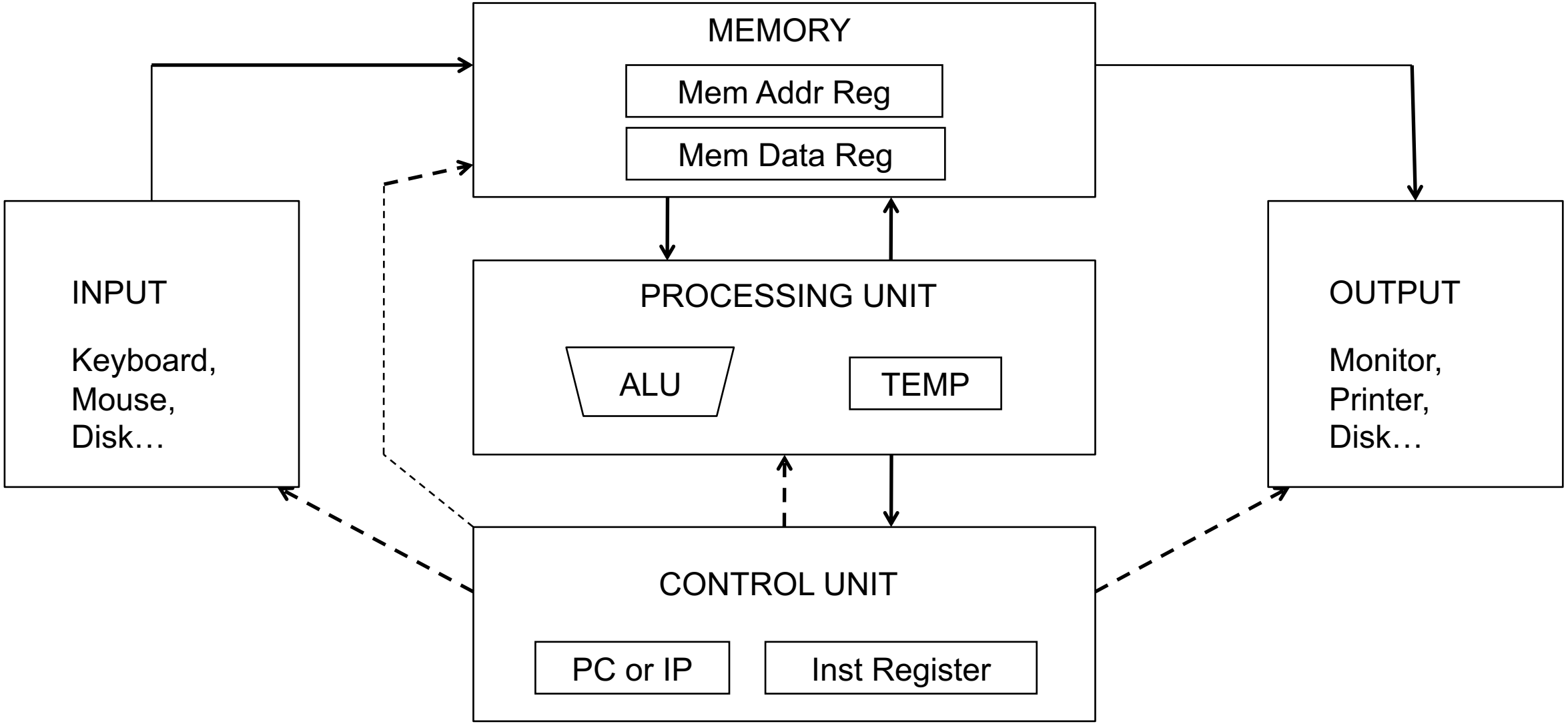
- Instruction set: All possible instructions that a computer is designed to be able to carry out

The von Neumann Model

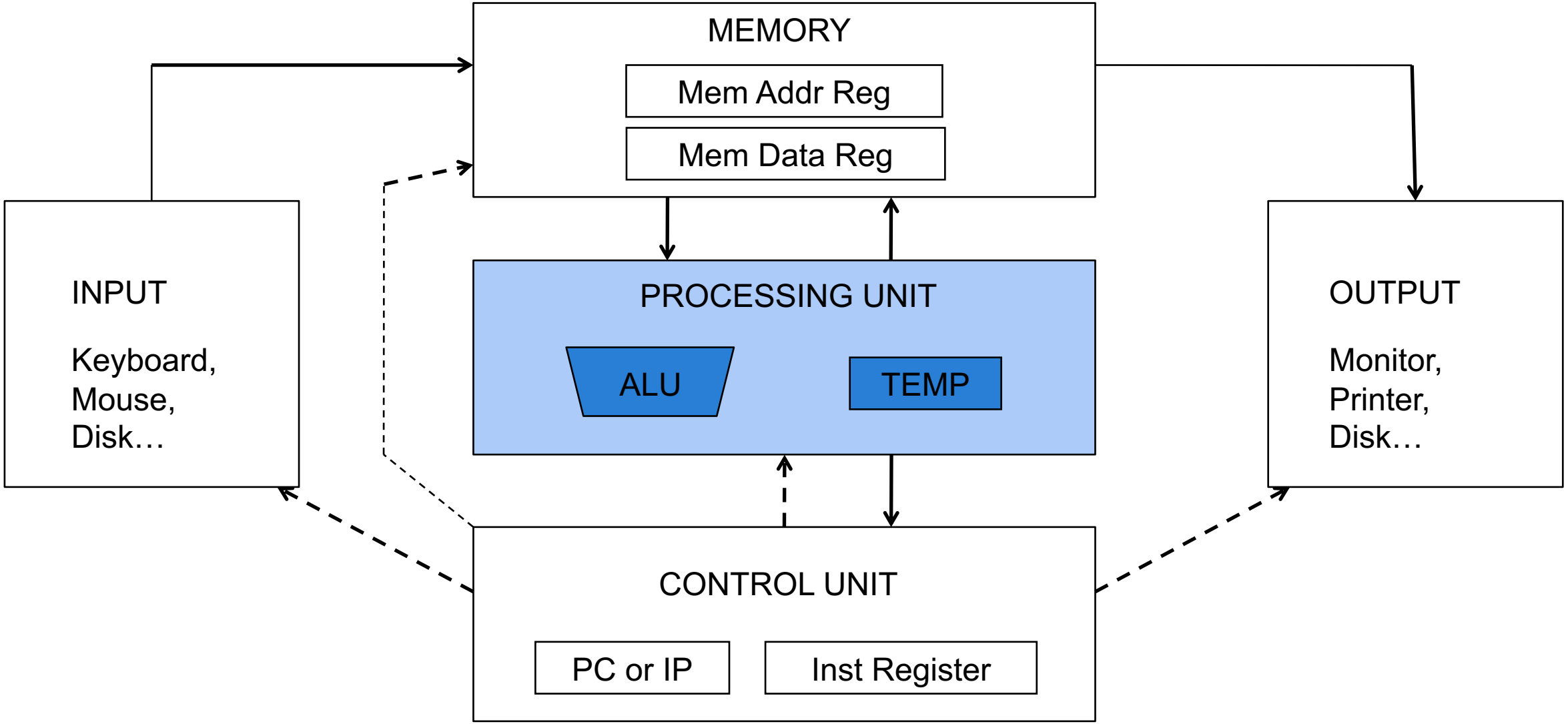
- To build a computer, we need an execution model for processing computer programs
- John von Neumann proposed a fundamental model in 1946
- The von Neumann Model consists of 5 components
  - Memory (stores the program and data)
  - Processing unit
  - Input
  - Output
  - Control unit (controls the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

All general-purpose computers today use the von Neumann model!

The von Neumann Model



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, srl, 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

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:

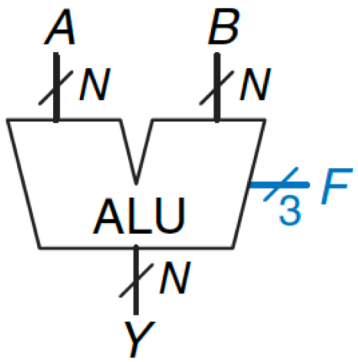


Figure 5.14 ALU symbol

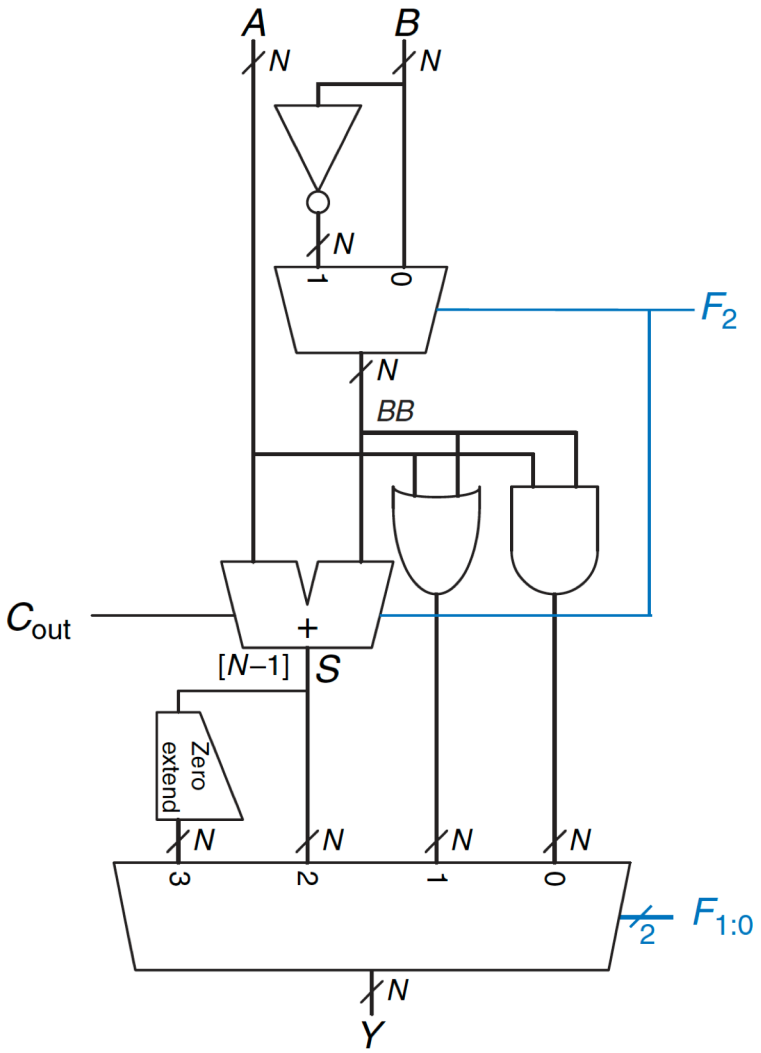
Table 5.1 ALU operations

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

Example ALU (Arithmetic Logic Unit)

Table 5.1 ALU operations

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



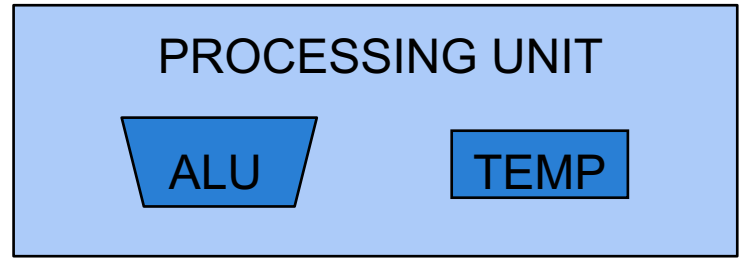
Processing Unit: Fast Temporary Storage

- It is almost always the case that a computer provides a small amount of storage very close to the 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

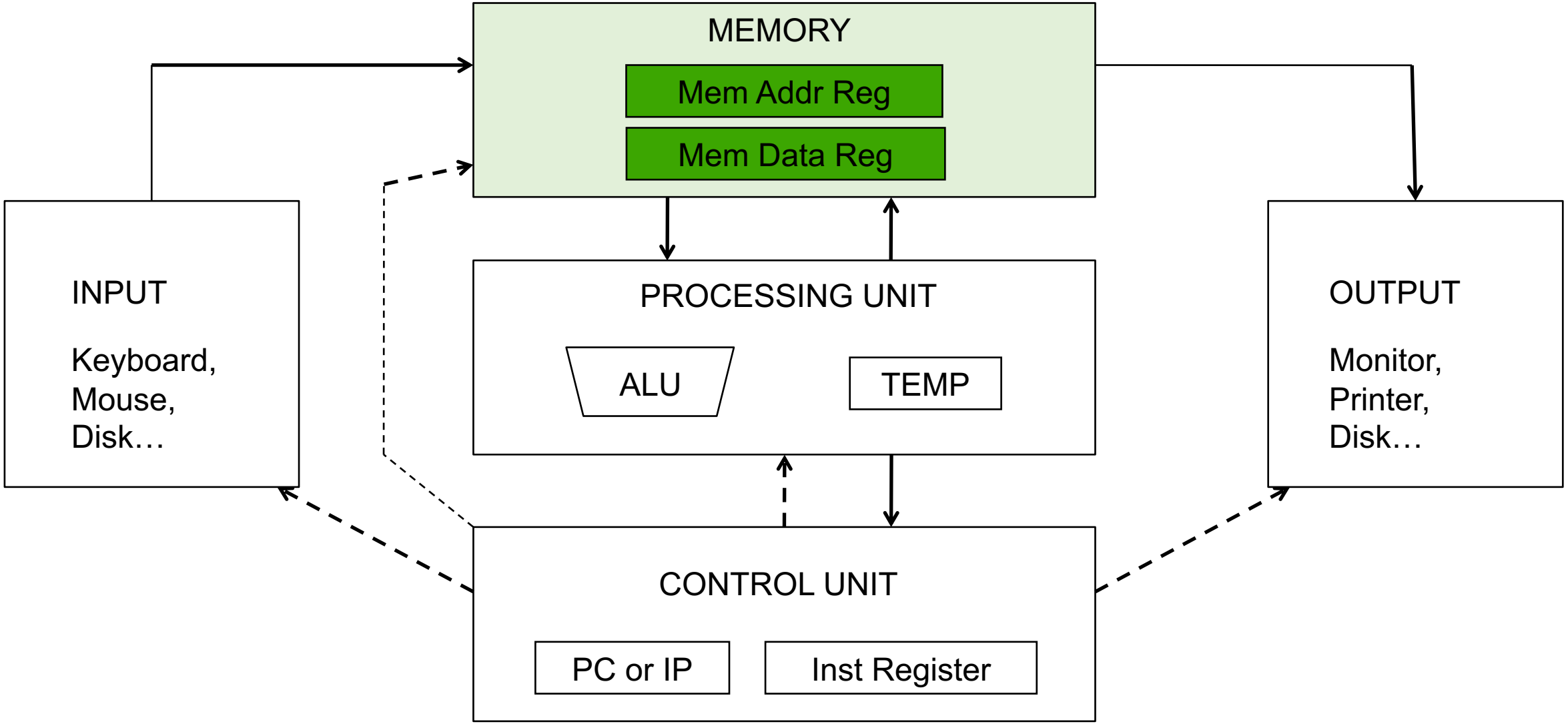


- Memory is large but slow
- Registers in the Processing Unit
  - Ensure fast access to values to be processed in the ALU
  - Typically, one register contains one word (same as word length)
- Register Set or Register File
  - A set of registers that can be manipulated by instructions
  - LC-3 has 8 general-purpose 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

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

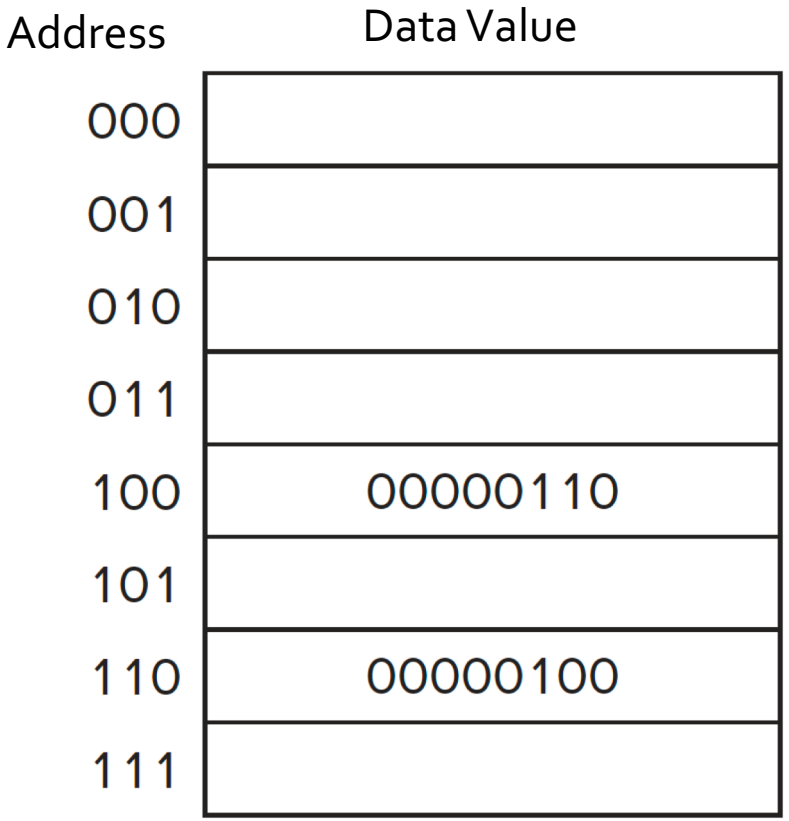


Memory

- 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

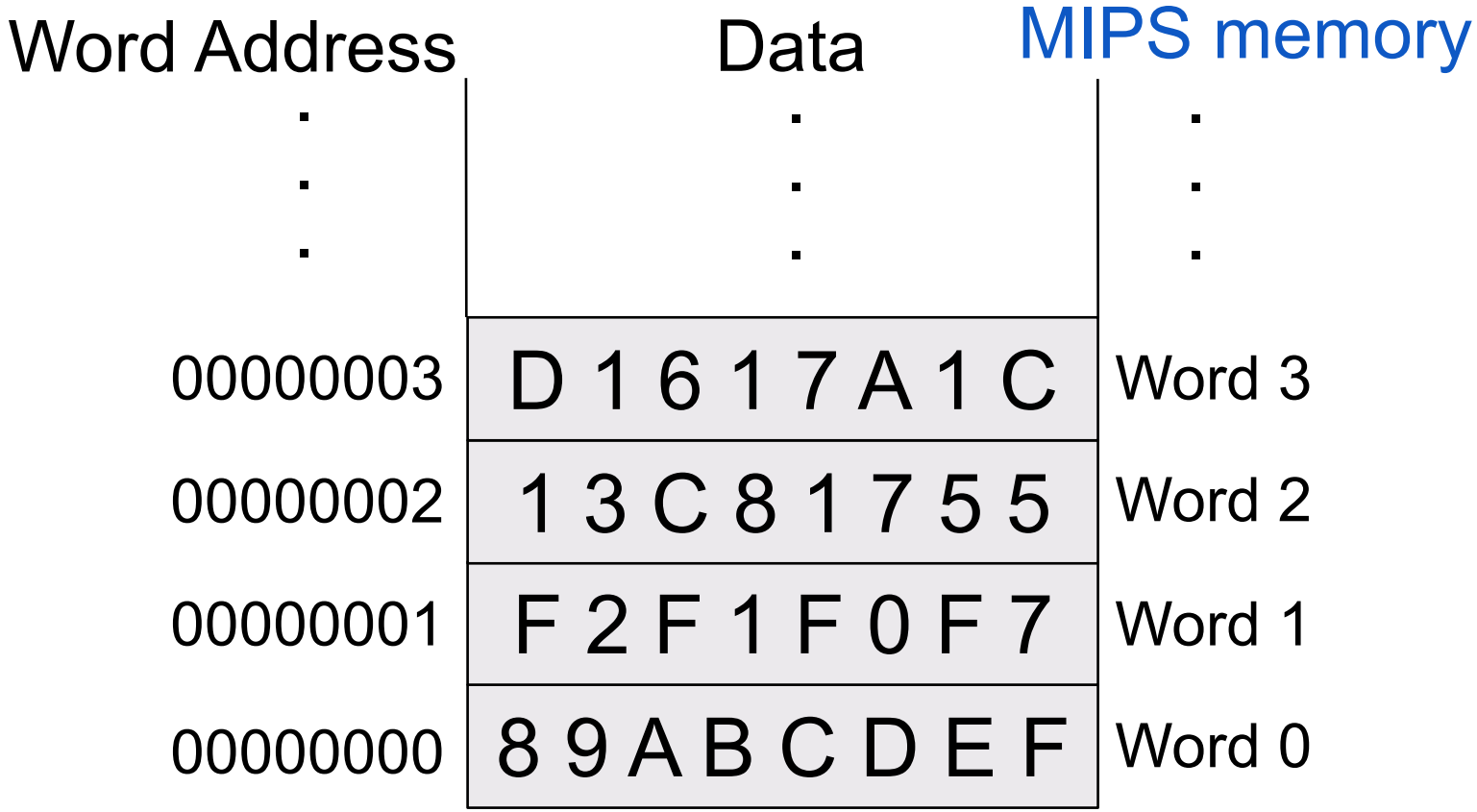


Question:
How can we make same-size memory bit addressable?

Answer:
64 locations
Each location stores 1 bit

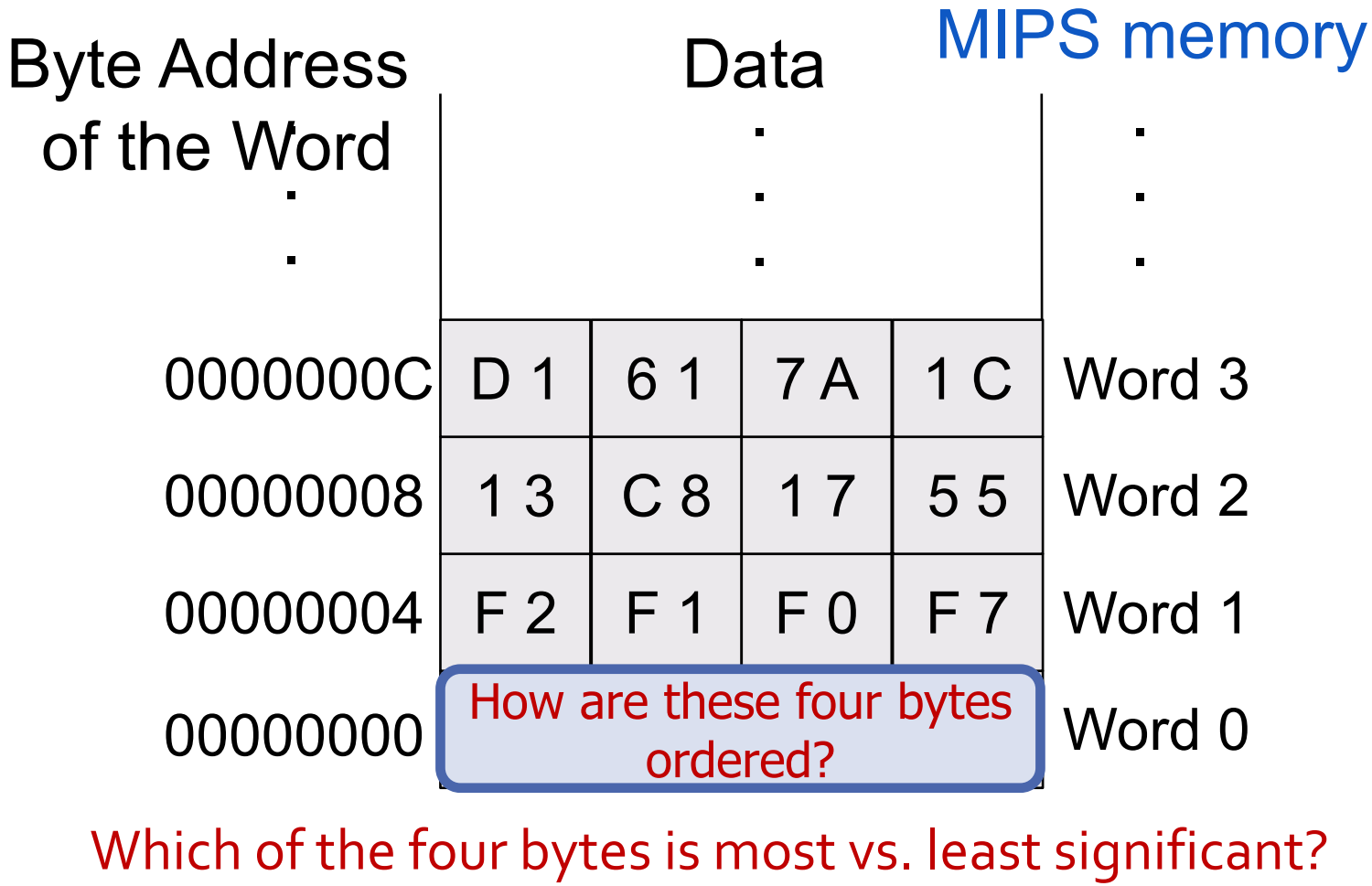
Word-Addressable Memory

- 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



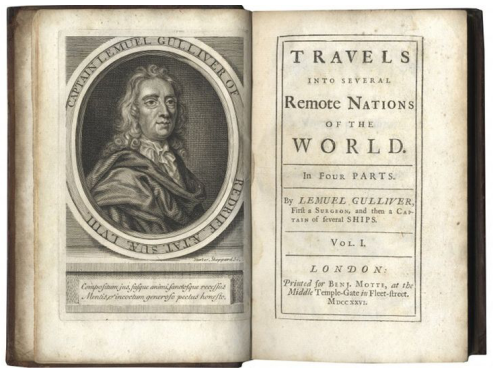
Byte-Addressable Memory

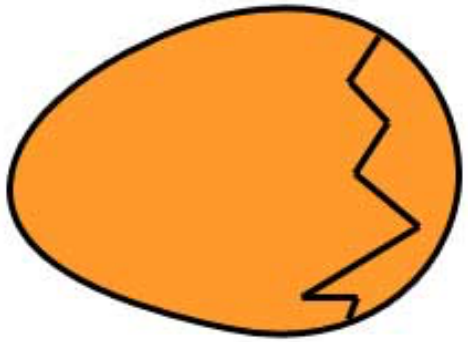
- Each byte has a unique address
  - MIPS is actually byte-addressable
  - LC-3b (updated version of LC-3) is also byte-addressable



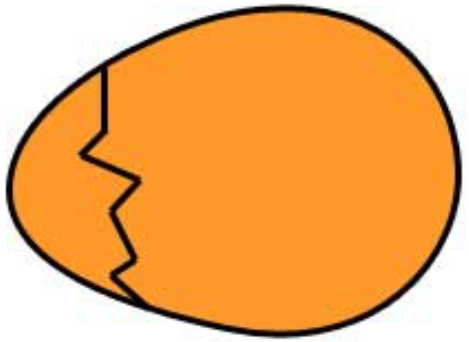
Big Endian vs. Little Endian

- 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



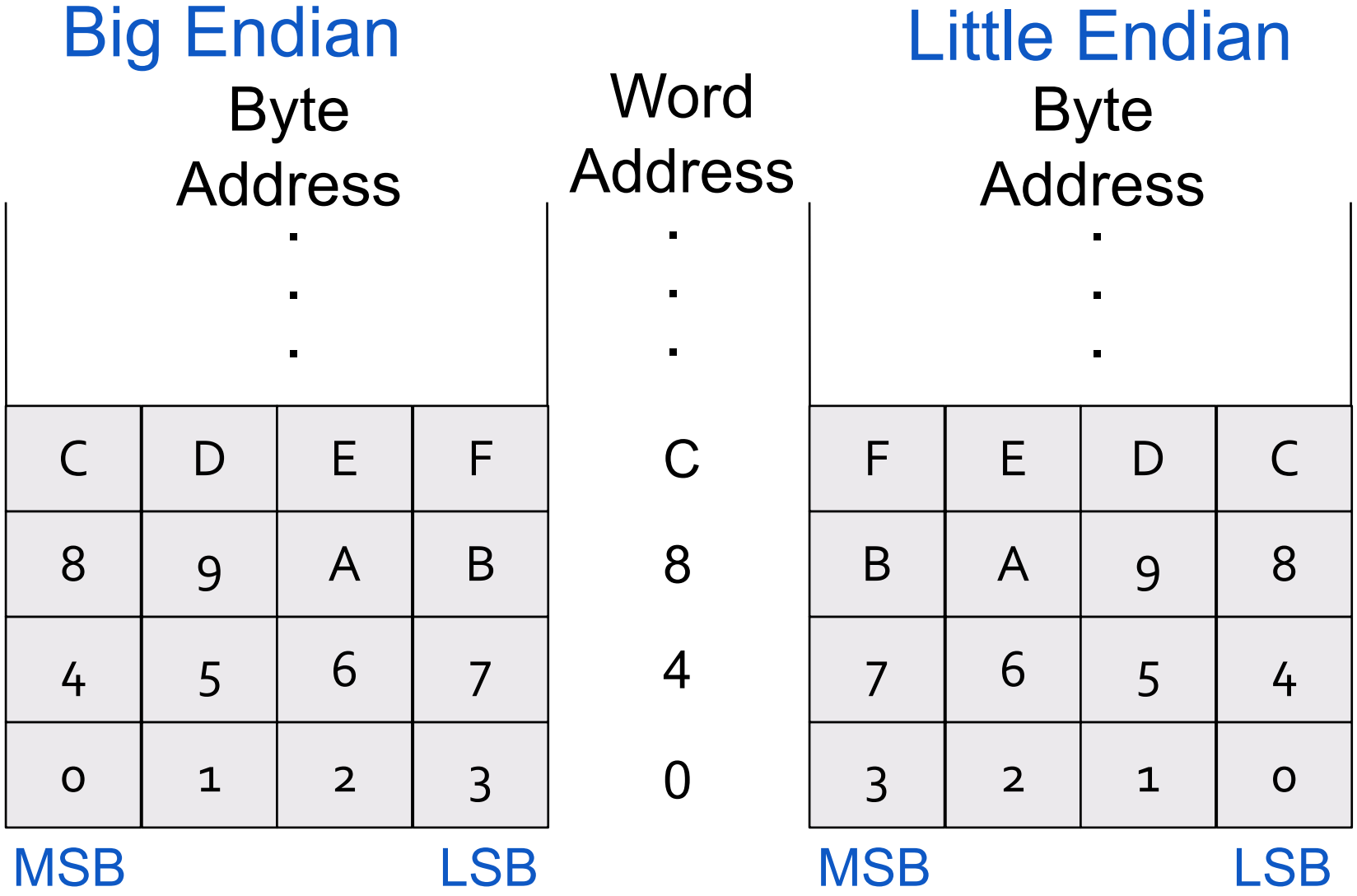


BIG ENDIAN - The way people always broke their eggs in the Lilliput land

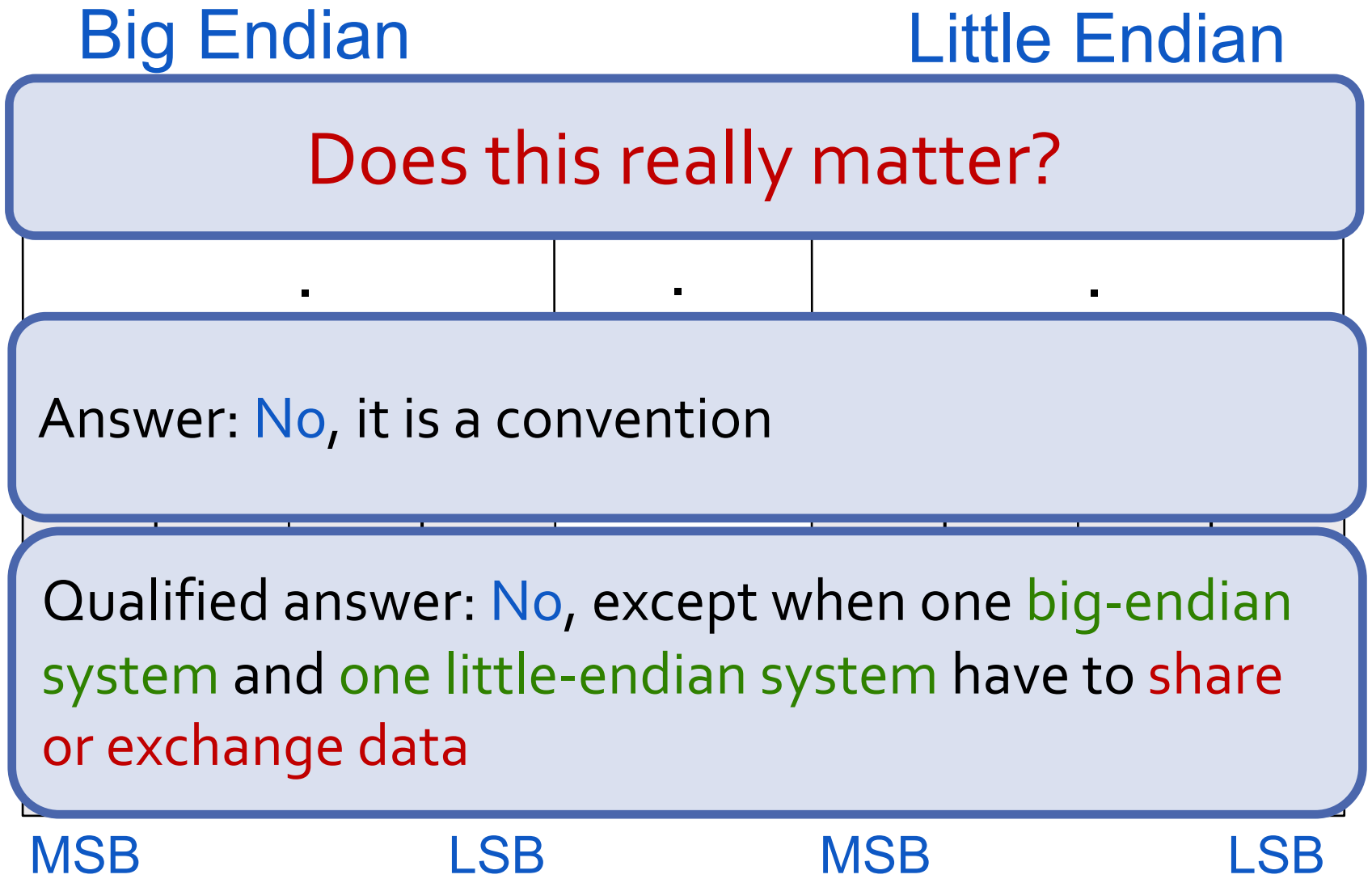


LITTLE ENDIAN - The way the king then ordered the people to break their eggs

Big Endian vs. Little Endian

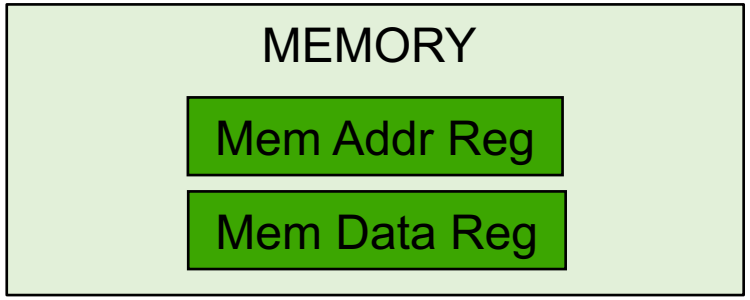


Big Endian vs. Little Endian

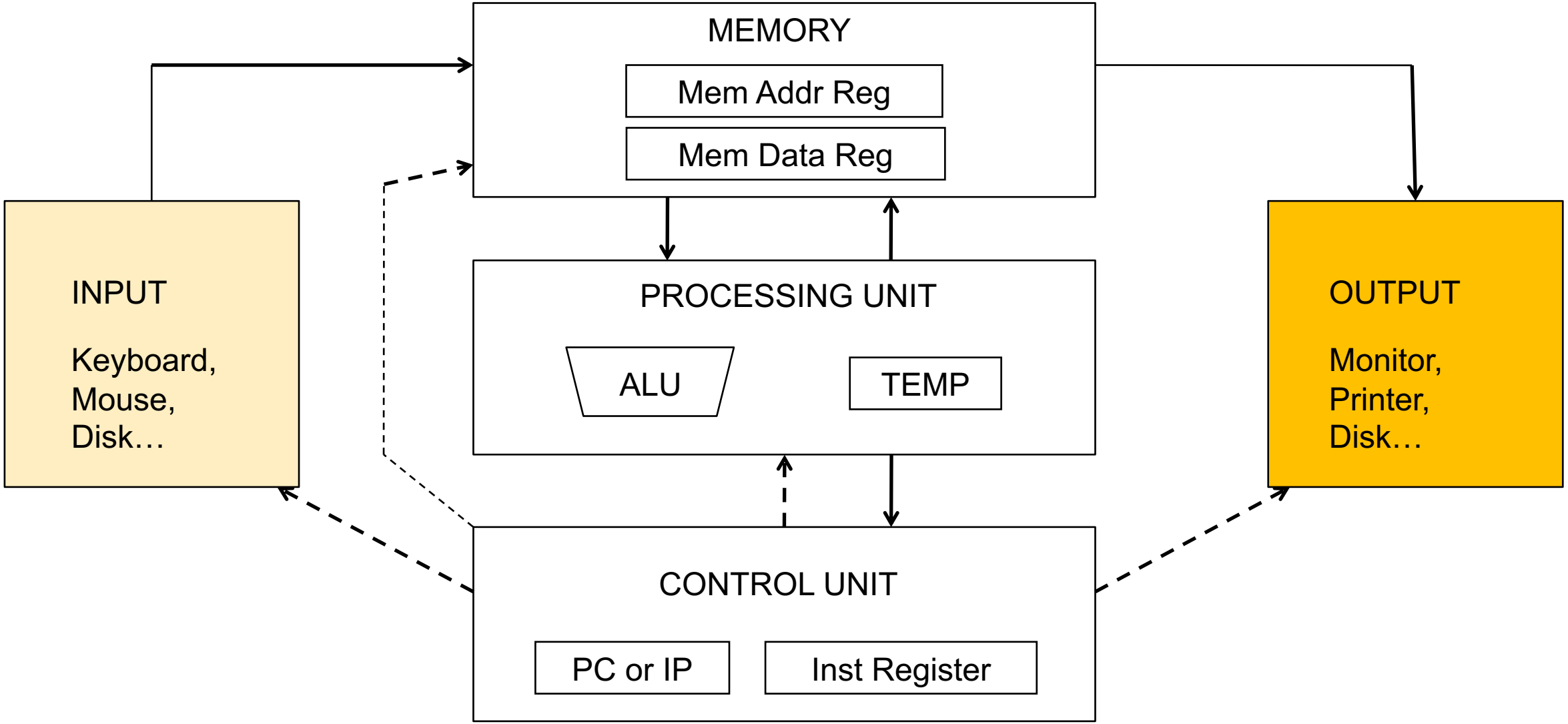


Accessing Memory: MAR and MDR

- There are two ways of accessing memory
  - Reading or loading data from a memory location
  - Writing or storing data in 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 the address specified by MAR



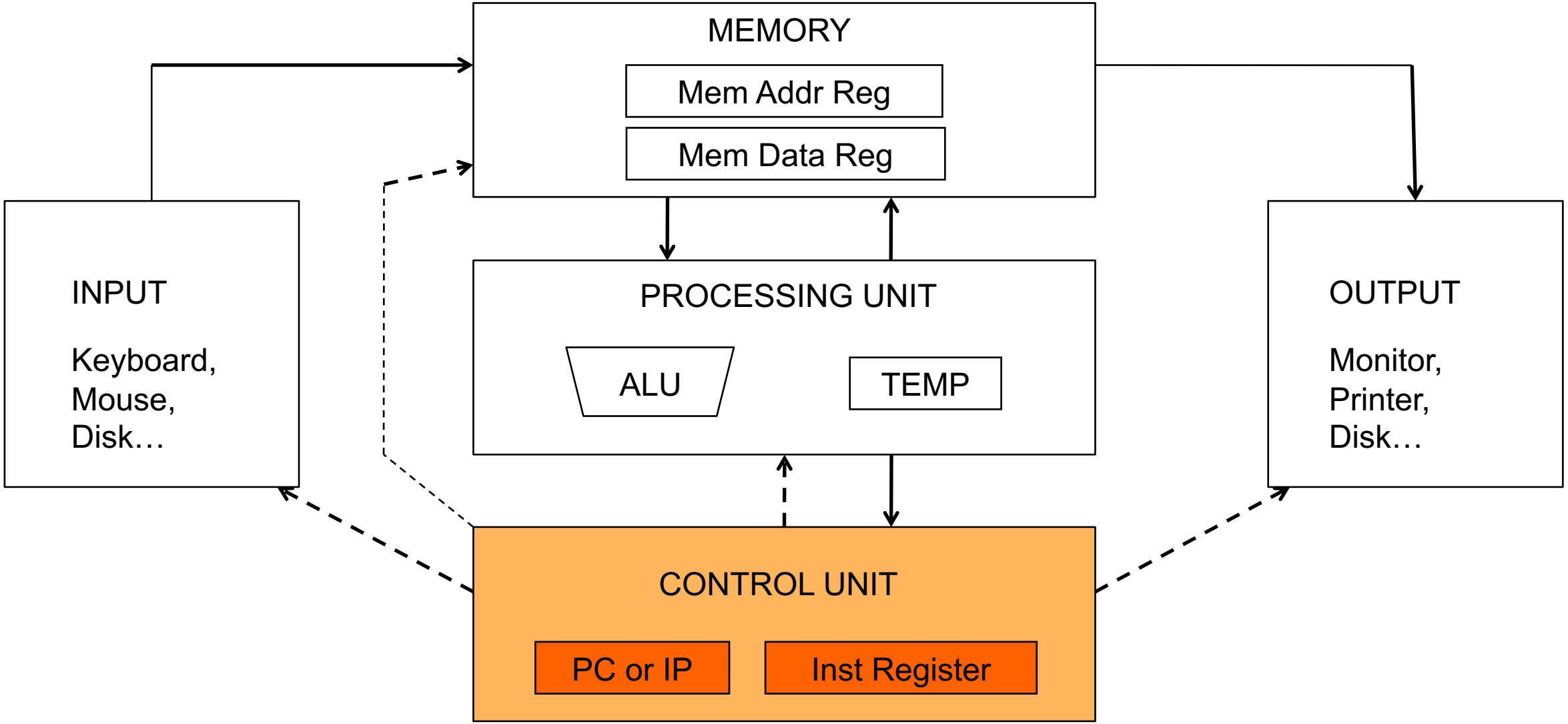
The von Neumann Model



Input and Output

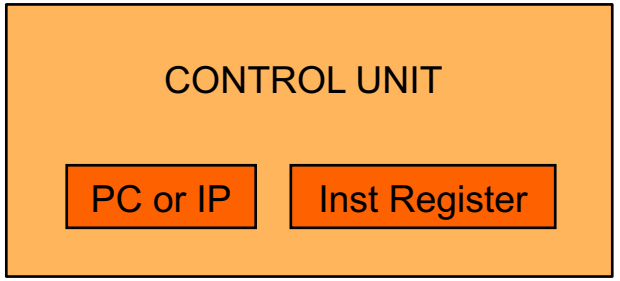
- Enable information to get into and out of a computer
- Many devices can be used for input and output
- They are called peripherals
  - Input
    - Keyboard
    - Mouse
    - Scanner
    - Disks
    - Etc.
  - Output
    - Monitor
    - Printer
    - Disks
    - Etc.
  - In LC-3, we consider the keyboard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

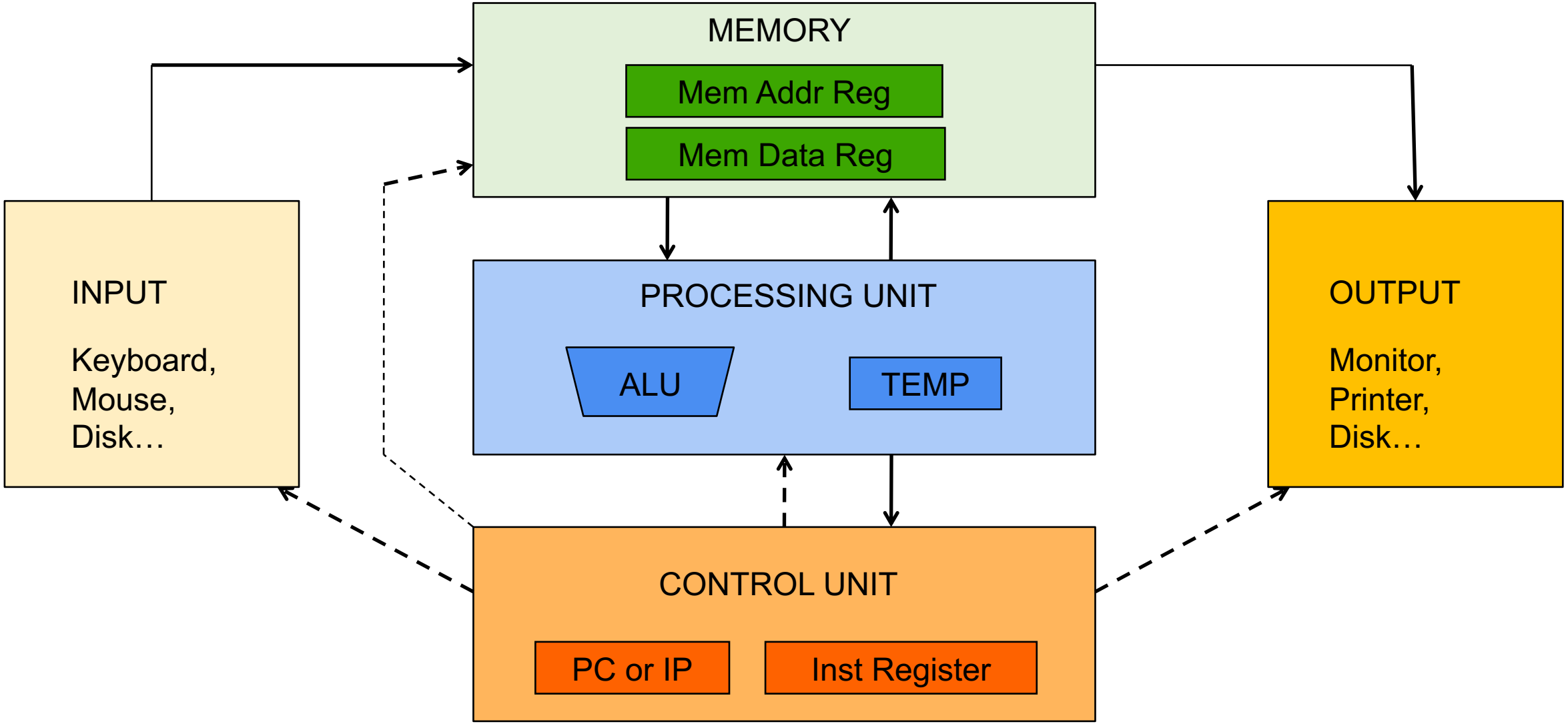
Program Counter

memory address

of the current (or next) instruction

Instructions (and programs) specify how to transform
The values of the programmer's 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

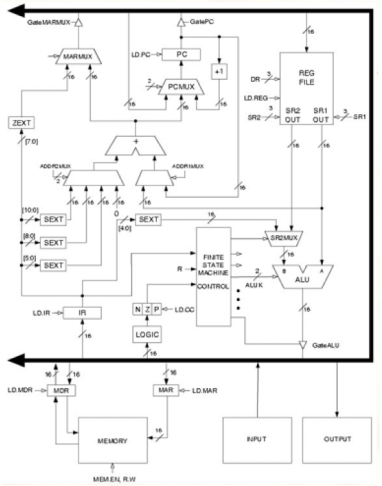
The von Neumann Model

Stored program

Sequential instruction processing

Hardware Design

LC-3: A von Neumann Machine



LC-3 (Little Computer 3): A von Neumann Machine

- A **simple educational 16-bit computer architecture**

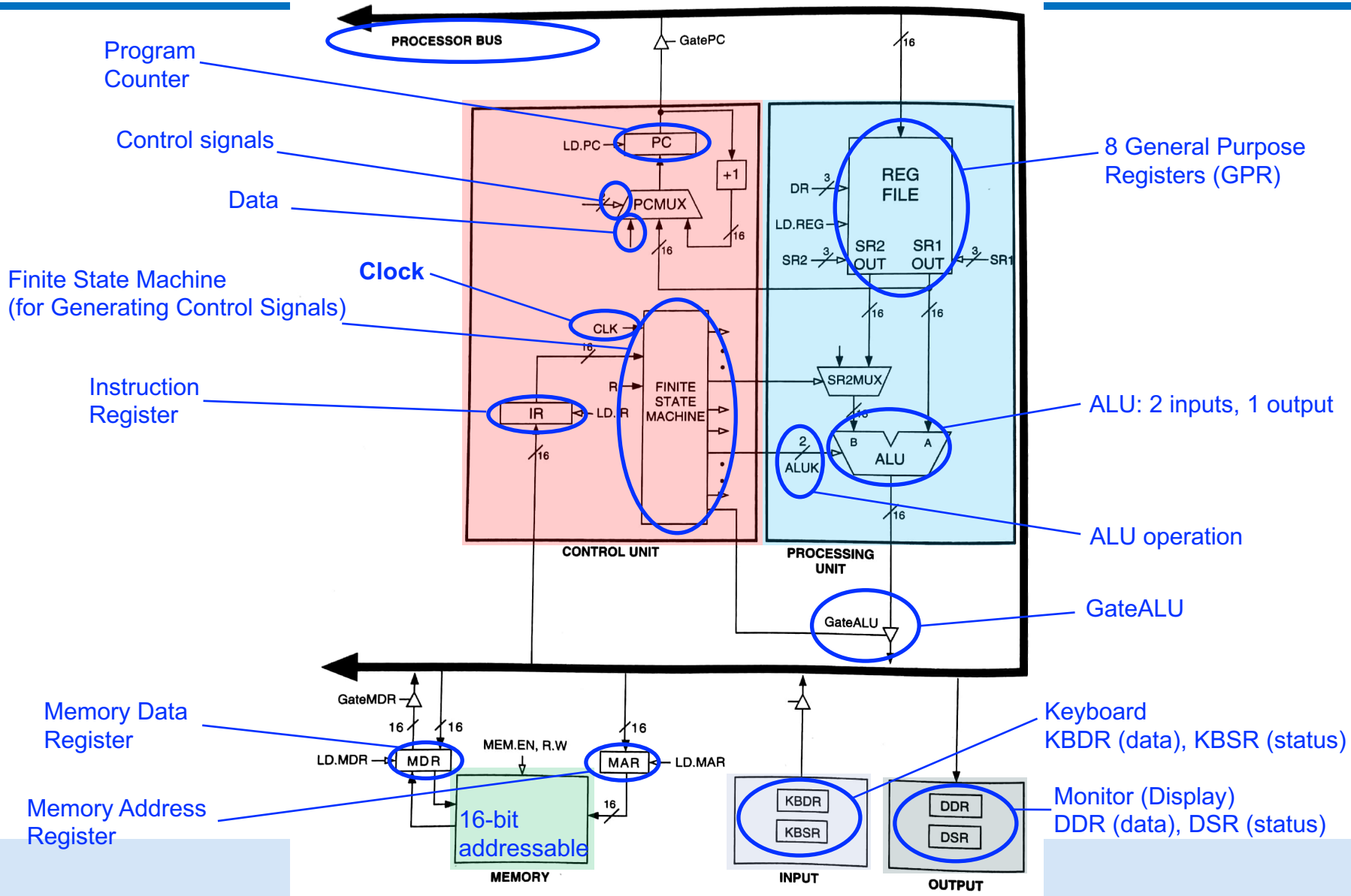


Figure 4.3 The LC-3 as an example of the von Neumann model

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

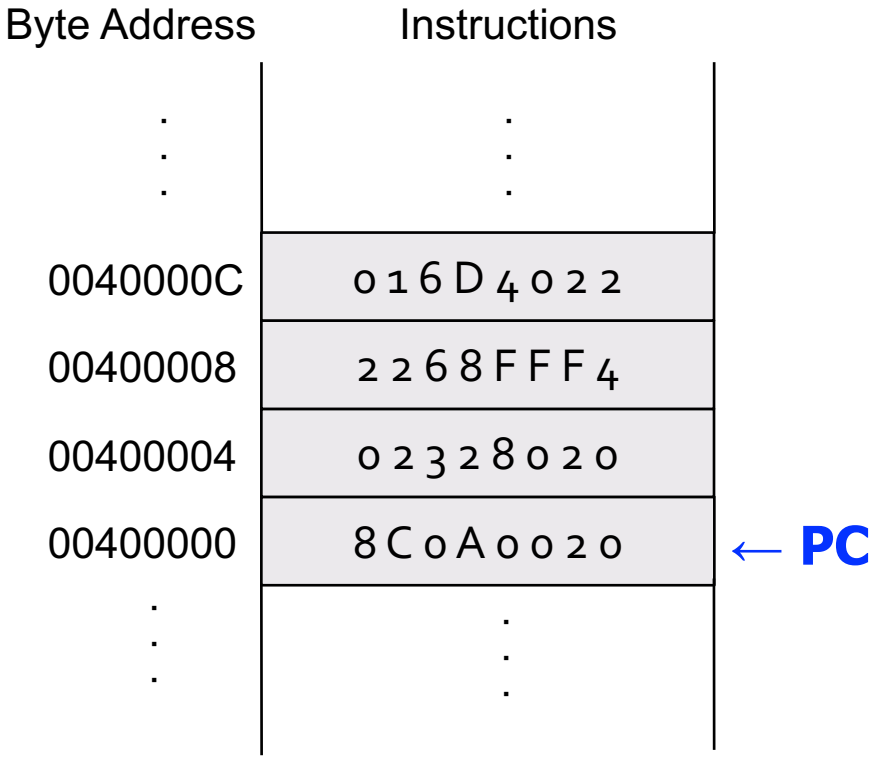
- A sample MIPS program
  - 4 instructions stored in consecutive words in memory
    - No need to understand the program now. We will get back to it

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
```



The Instruction

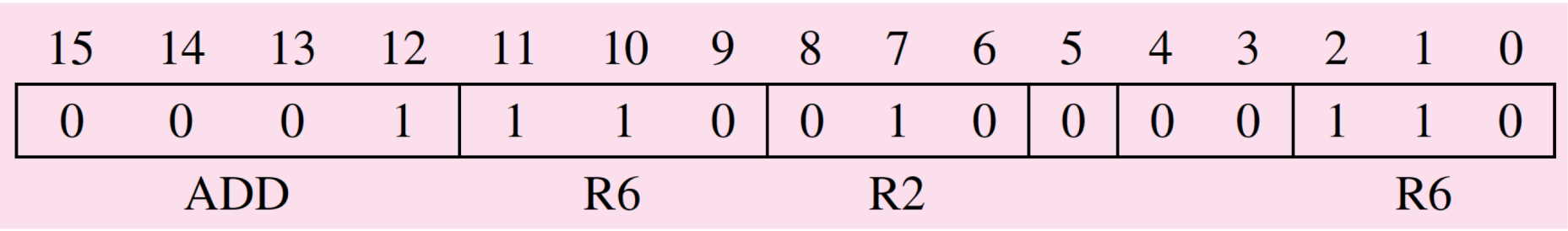
- An instruction is the most basic unit of computer processing
  - Instructions are words in the language of a computer
  - Instruction Set Architecture (ISA) is the vocabulary

- 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



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 ← 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
```

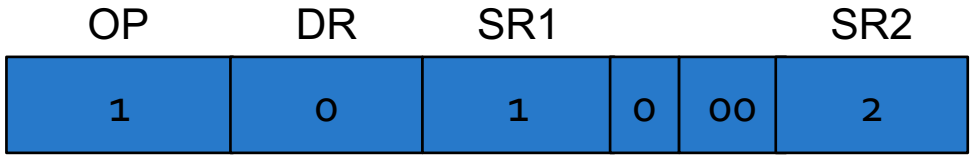
From Assembly to Machine Code in LC-3

❑ Addition

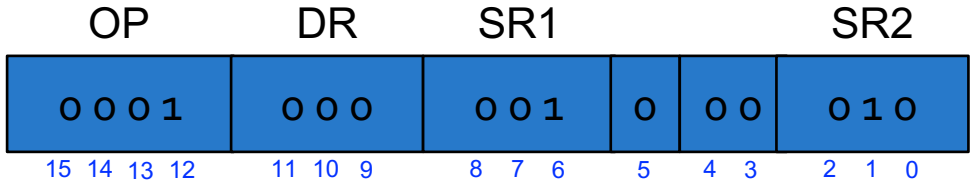
LC-3 assembly

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

Field Values



Machine Code (Instruction Encoding)

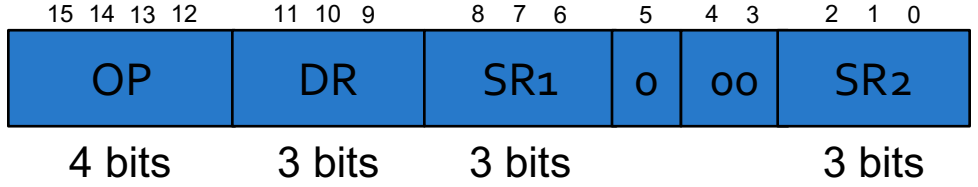


0x1042

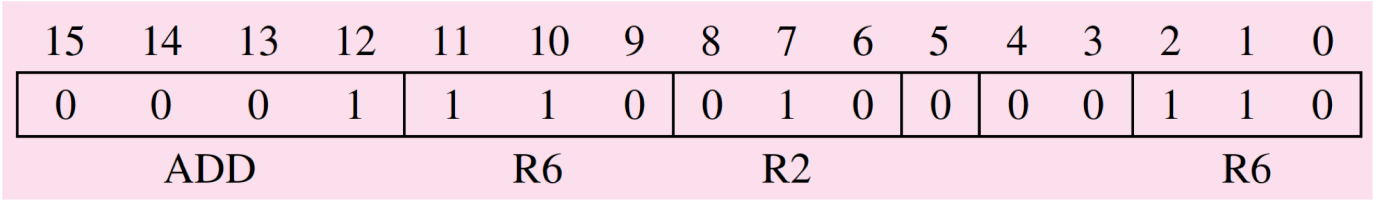
Machine Code, in short (hexadecimal)

Instruction Format (or Encoding)

LC-3 Operate Instruction Format



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



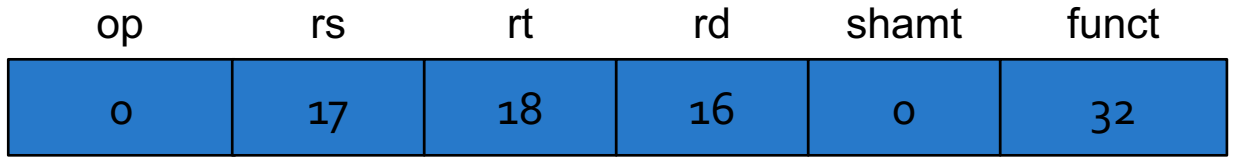
From Assembly to Machine Code in MIPS

❑ Addition

MIPS assembly

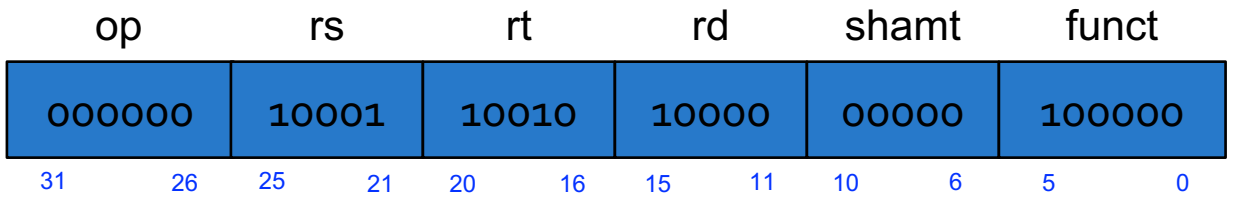
```
add  $s0, $s1, $s2
```

Field Values



rd ← rs + rt

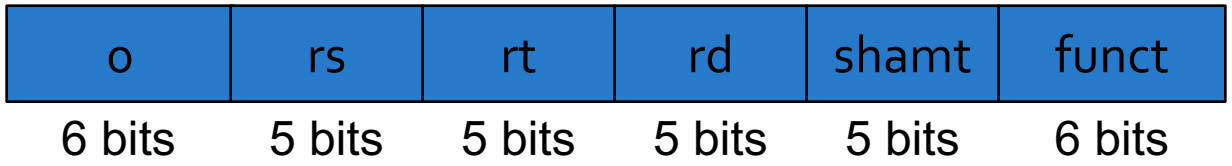
Machine Code (Instruction Encoding)



0x02328020

Instruction Format: R-Type in MIPS

- MIPS R-type Instruction Format
  - 3 register operands



  - o = 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	Assembly
`a = A[i];`	`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 ← 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 ← Memory[R0 + 2]

- MIPS assembly (assuming word-addressable)

High-level code

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

MIPS assembly

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

$s3 ← Memory[$s0 + 2]

These instructions use a particular addressing mode
(i.e., the way the address is calculated), called base+offset

Load Word in Byte-Addressable MIPS

- MIPS assembly

High-level code

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

MIPS assembly

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

$s3 ← Memory[$s0 + 8]

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

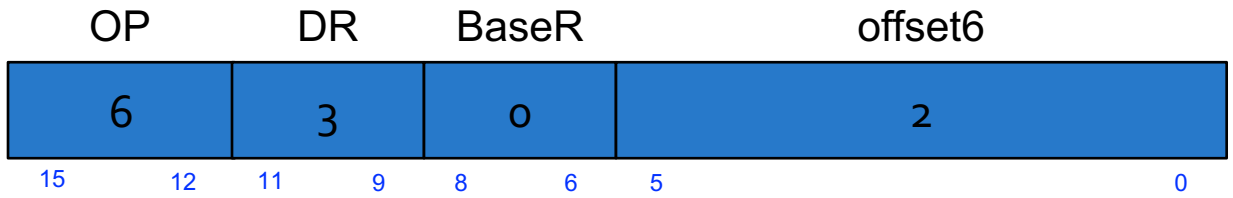
Instruction Format With Immediate

- LC-3

LC-3 assembly

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

Field Values

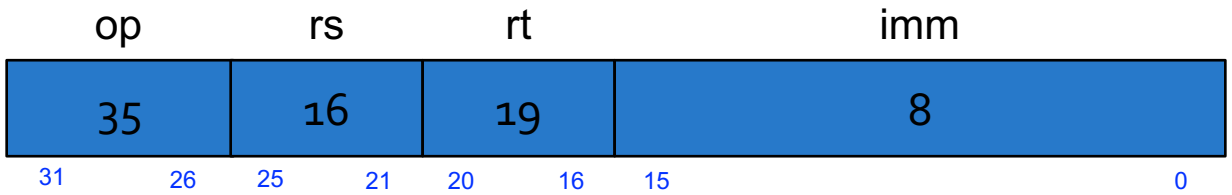


- MIPS

MIPS assembly

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

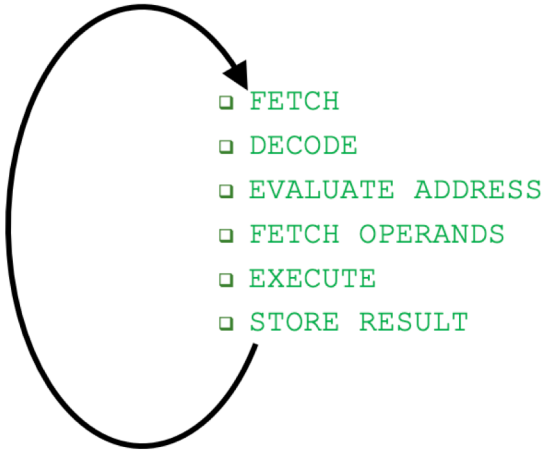
Field Values



I-Type

Hardware Design

Instruction (Processing) Cycle



How Are These Instructions Executed

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

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

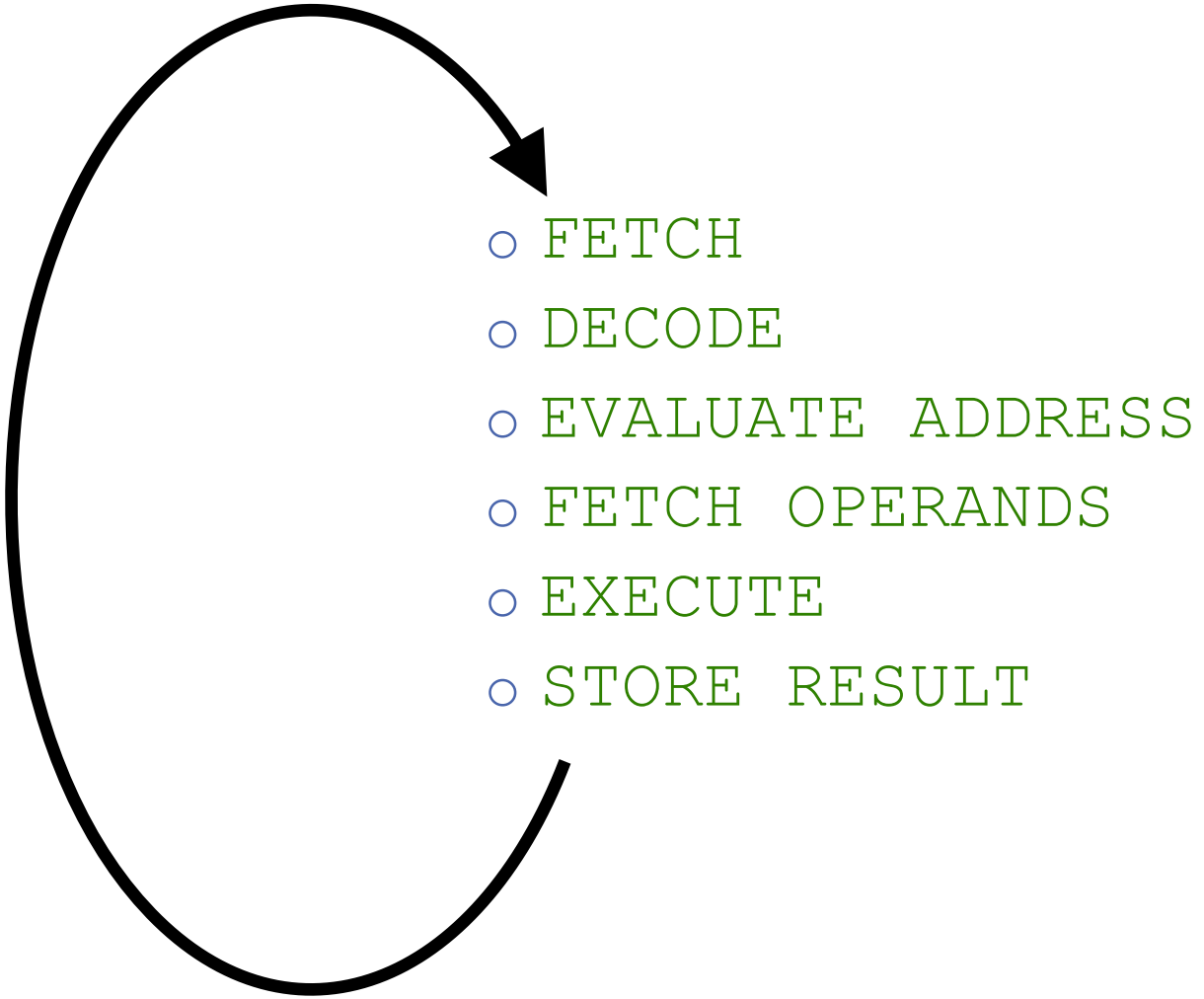
- But how are these instructions executed on the computer?
  - The process of executing an instruction is called the instruction cycle (or, instruction processing cycle)

The Instruction Cycle

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

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

After STORE RESULT, a New FETCH



FETCH

- The FETCH phase obtains the instruction from memory and loads it into the Instruction Register (IR)
- This phase is common to every instruction type
- Complete description
  - Step 1: Load the MAR with the contents of the PC, and simultaneously increment the PC
  - Step 2: Interrogate memory. This results in the instruction being placed in the MDR by memory
  - Step 3: Load the IR with the contents of the MDR

FETCH in LC-3

Step 1: Load MAR and increment PC

Step 2: Access memory

Step 3: Load IR with the content of MDR

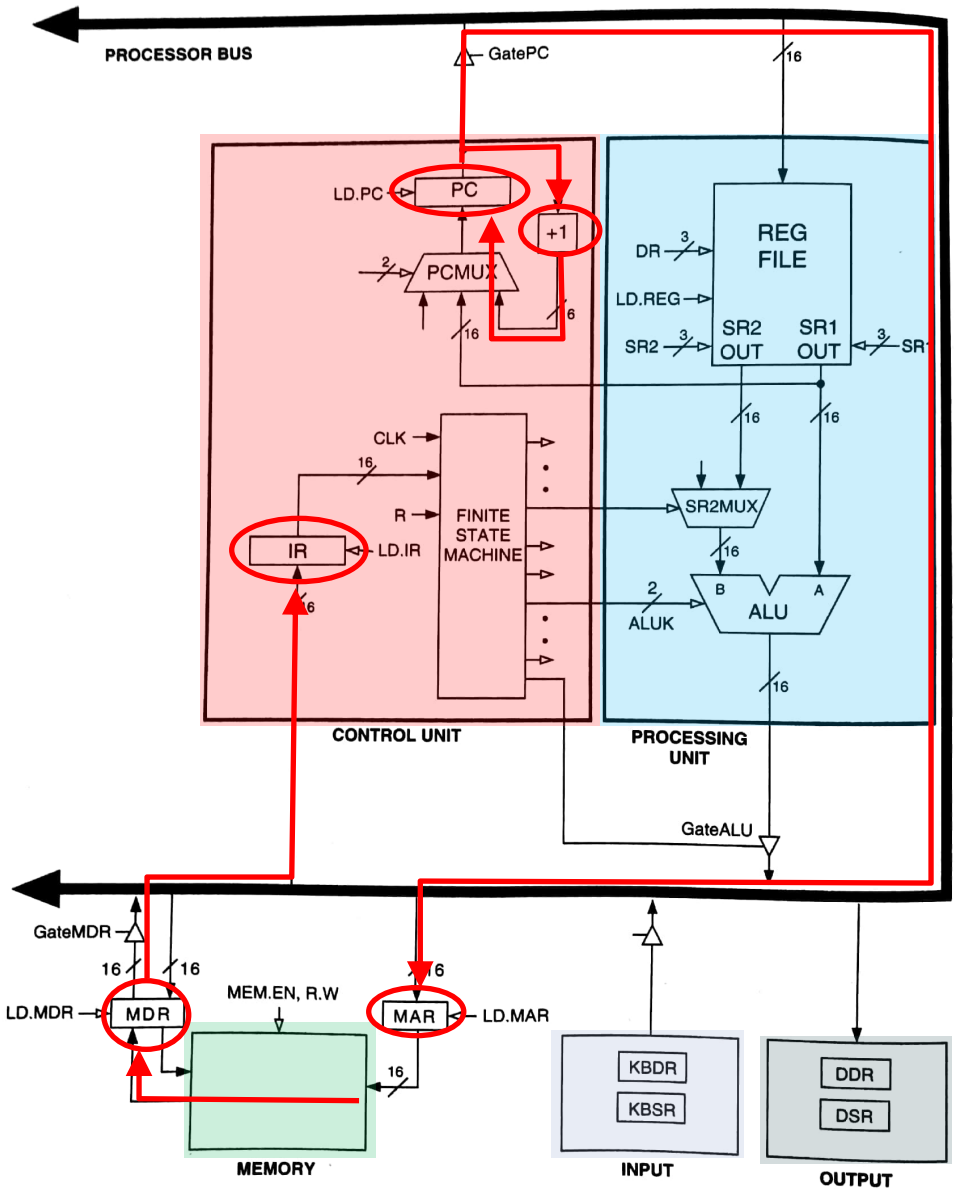


Figure 4.3 The LC-3 as an example of the von Neumann model

DECODE

- The DECODE phase identifies the instruction
  - Also generates the set of control signals to process the identified instruction in later phases of the instruction cycle
- Recall the decoder (from our Combinational Logic lectures)
  - 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

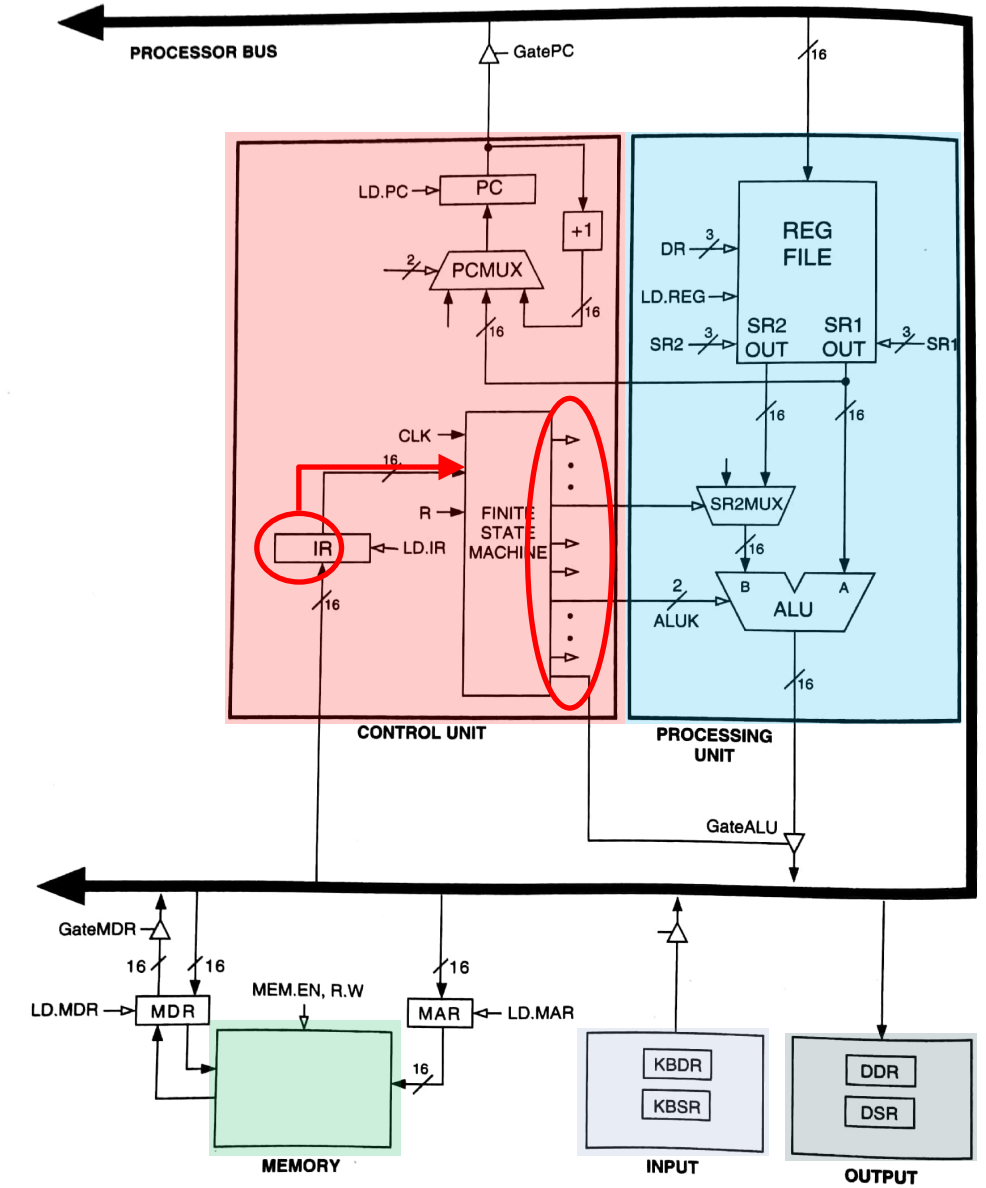
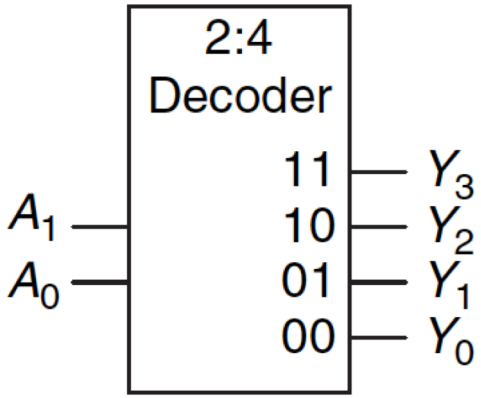


Figure 4.3 The LC-3 as an example of the von Neumann model

Recall: Decoder in LOD

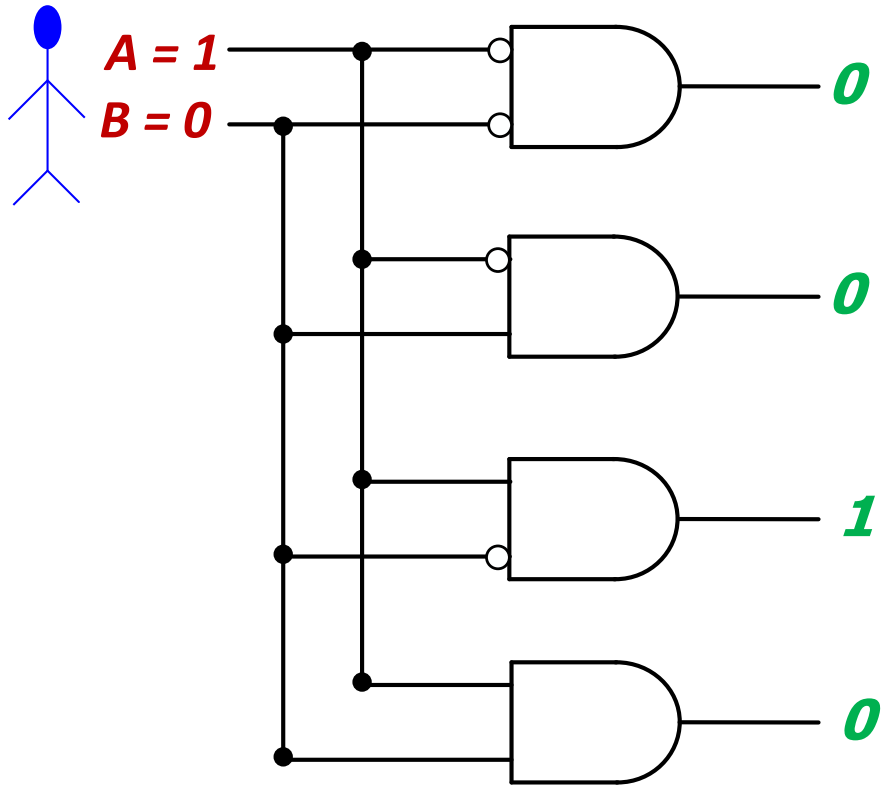
- "Input pattern detector"
- n inputs and 2^n outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The output that is logically 1 is the output corresponding to the input pattern that the logic circuit is expected to detect
- Example: 2-to-4 decoder

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

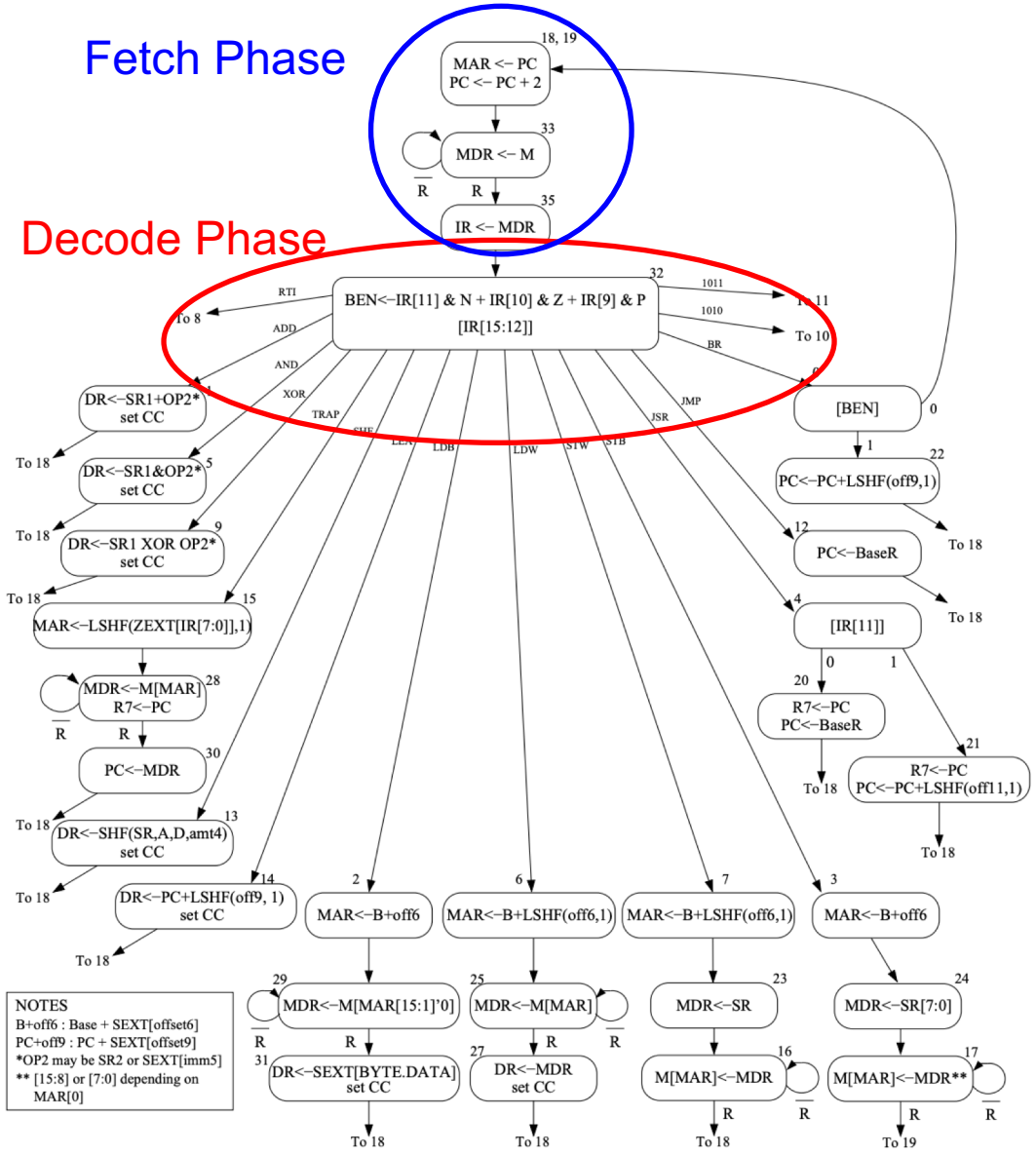


Recall: Decoder (II)

- 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



EVALUATE ADDRESS

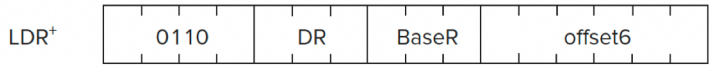
- The EVALUATE ADDRESS phase computes the address of the memory location that is needed to process the instruction

- 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



DR ← Memory[BaseR + sign-extend(offset6)]

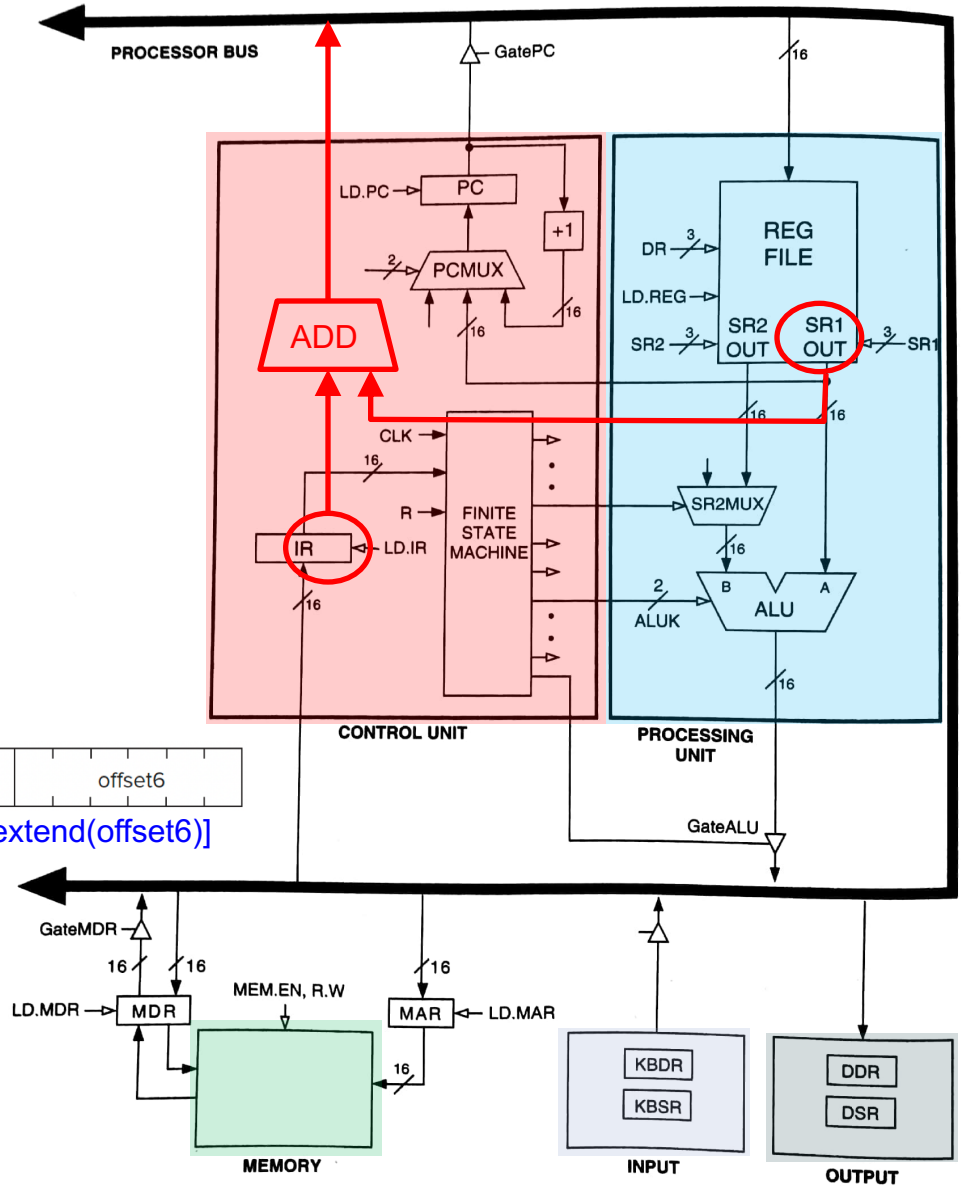


Figure 4.3 The LC-3 as an example of the von Neumann model

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 the source operand in MDR

- In ADD
  - Obtain the source operands from the register file
  - In some microprocessors, operand fetch from the 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 result in MDR (step 2)

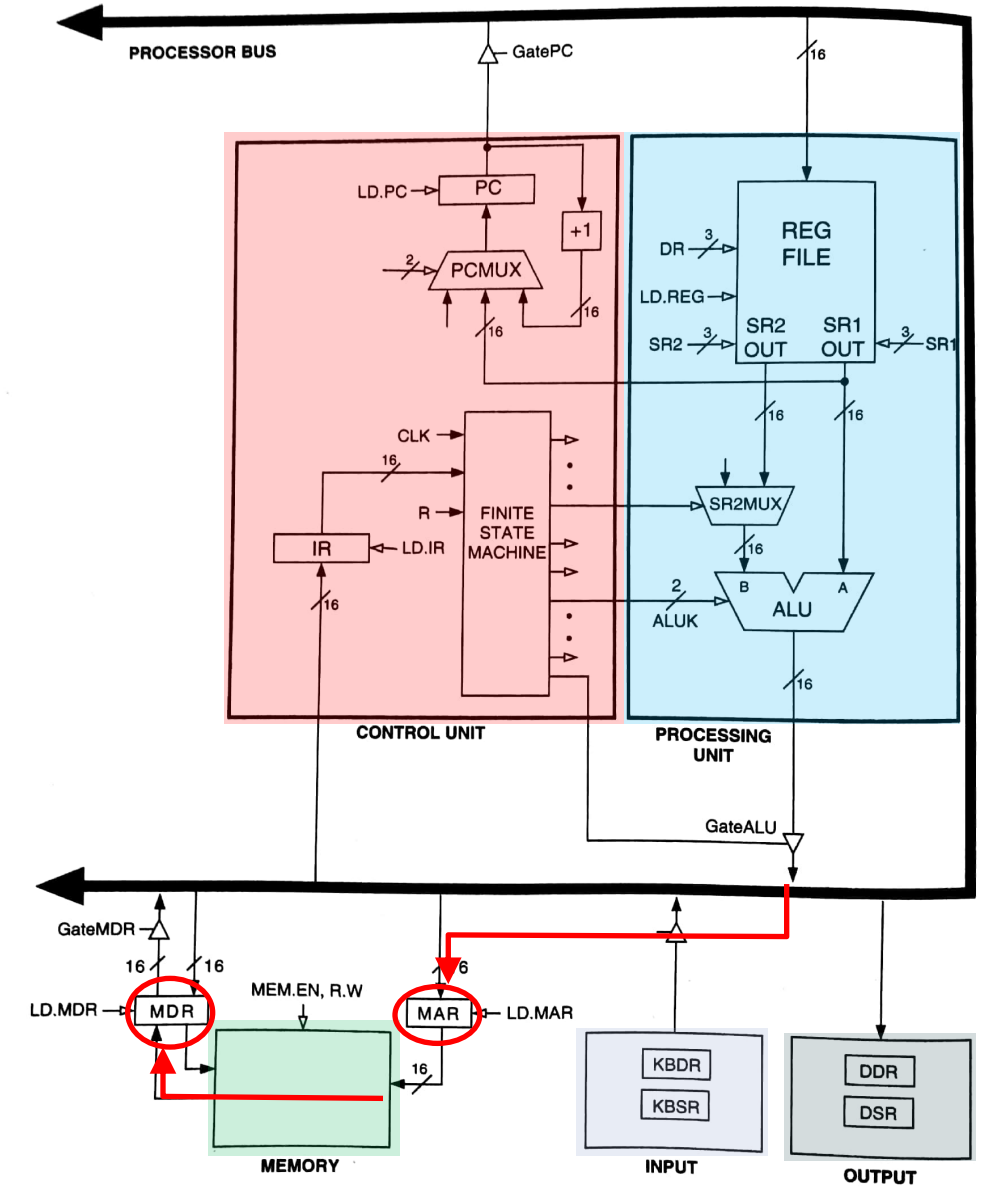


Figure 4.3 The LC-3 as an example of the von Neumann model

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

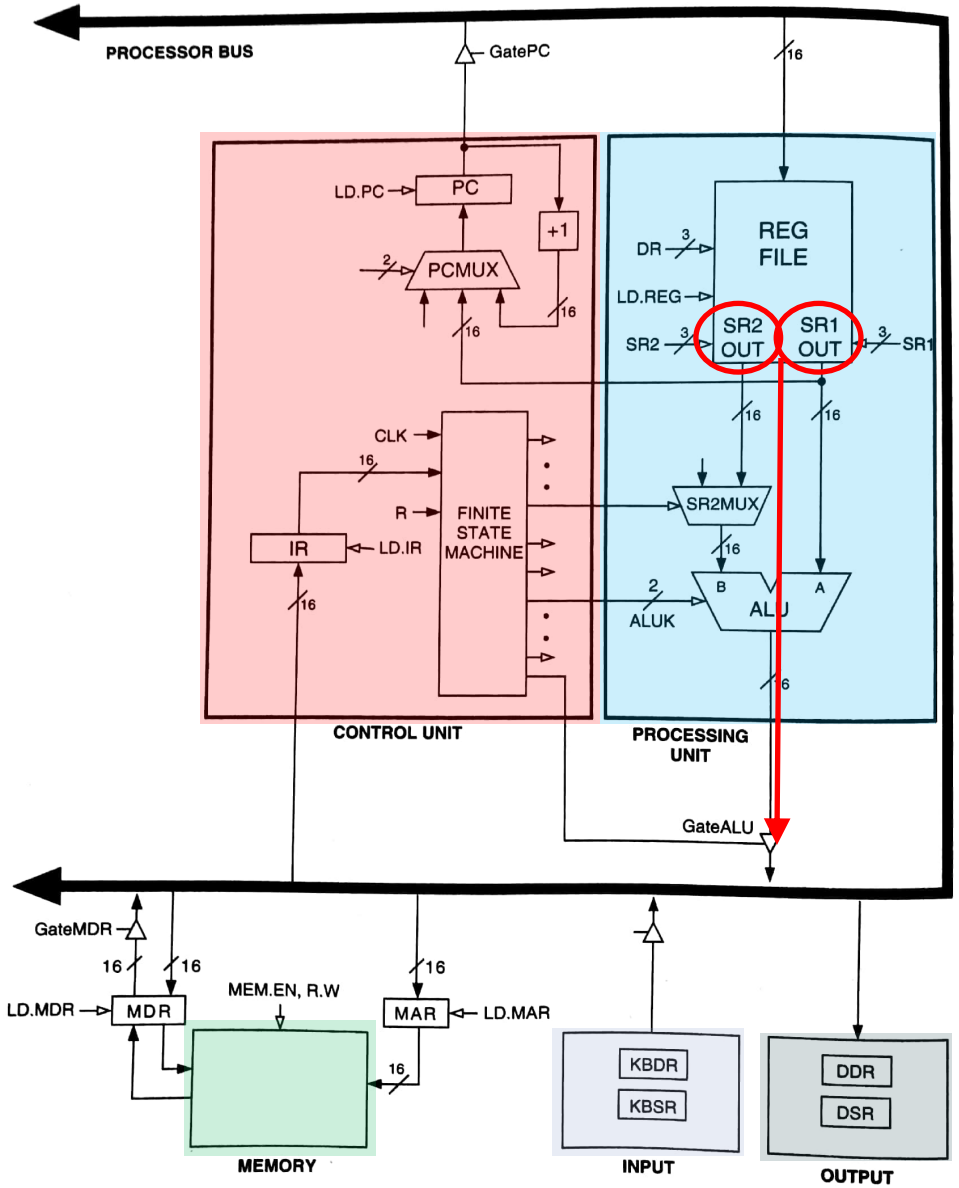


Figure 4.3 The LC-3 as an example of the von Neumann model

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

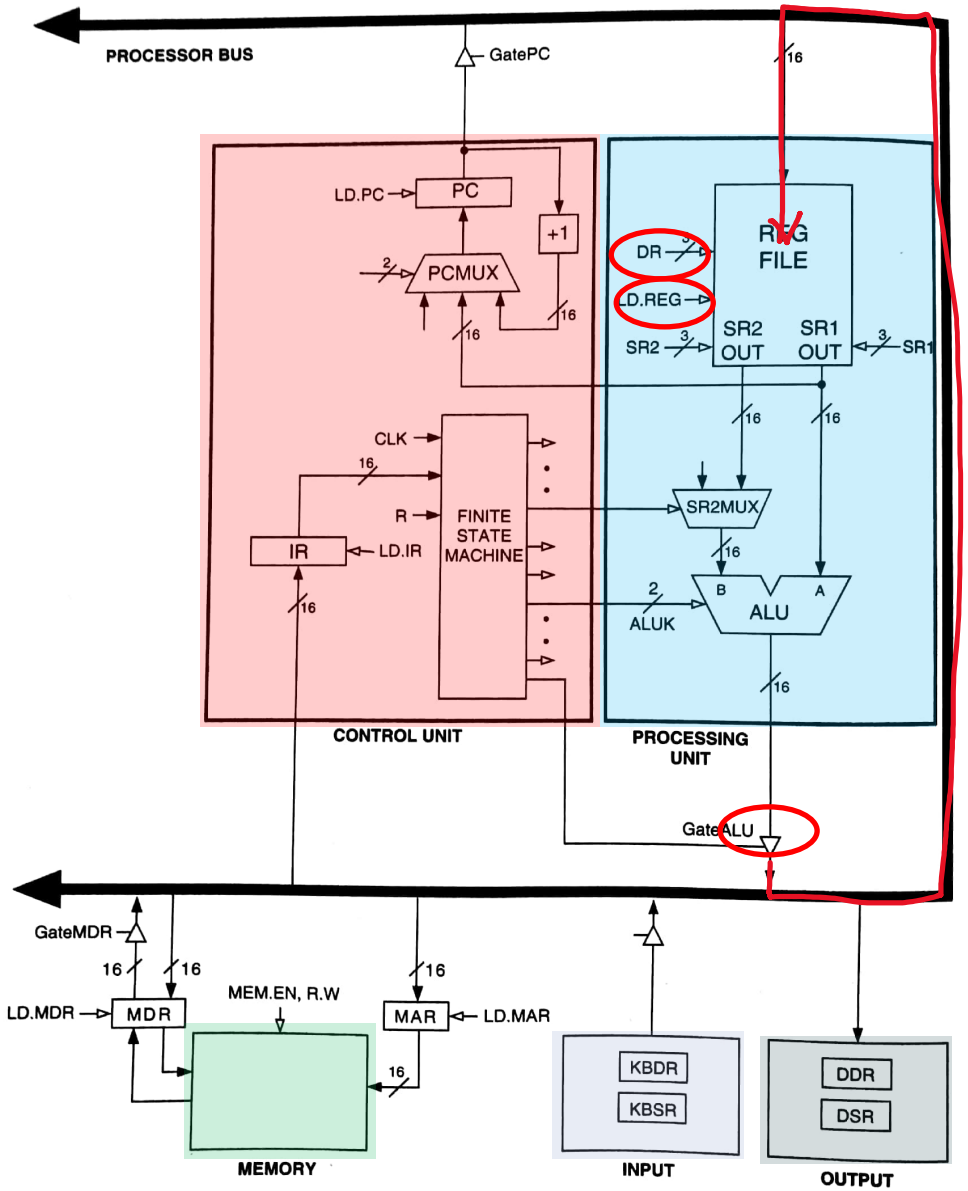


Figure 4.3 The LC-3 as an example of the von Neumann model

STORE RESULT in LC-3

LDR loads MDR into DR

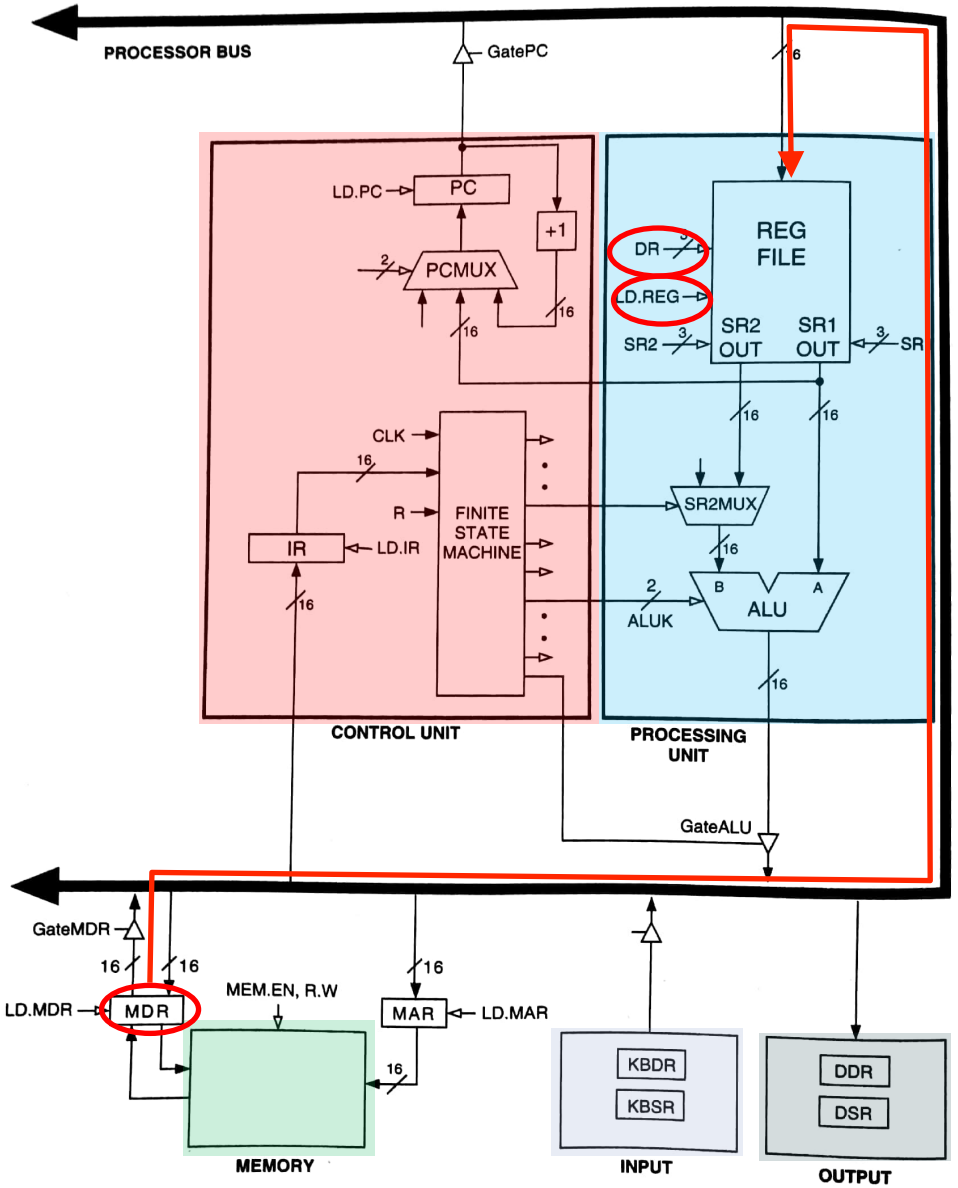
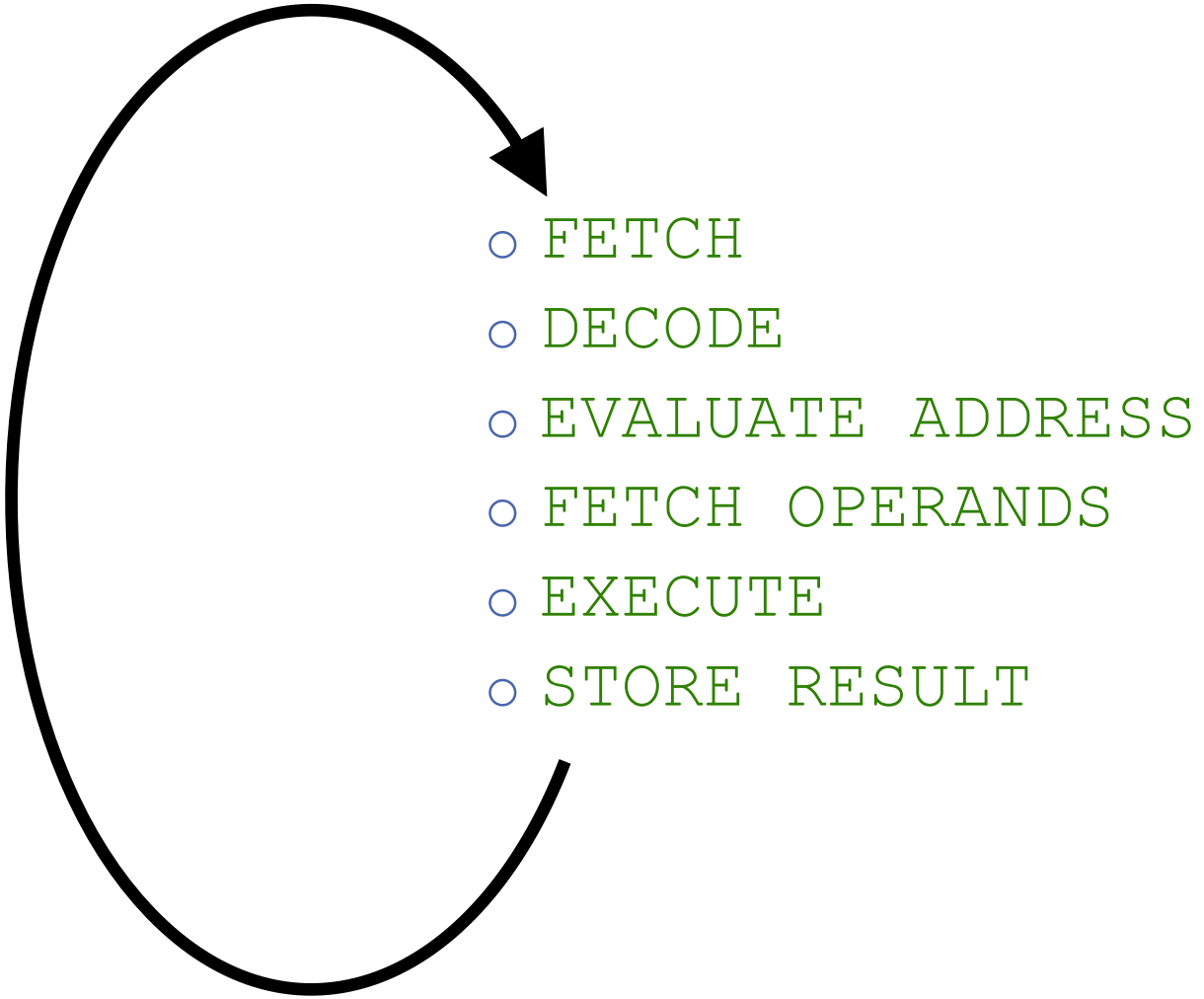


Figure 4.3 The LC-3 as an example of the von Neumann model

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

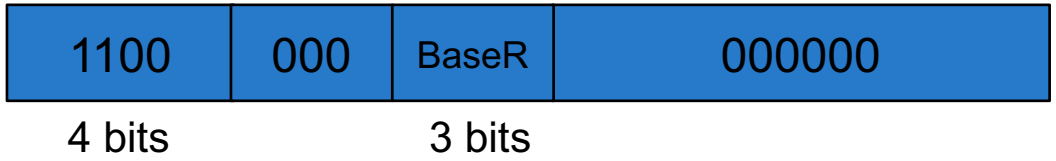
- Control instructions allow a program to execute out of sequence
  - They can change the PC by loading it during the EXECUTE phase
  - That wipes out the incremented PC (loaded during the FETCH phase)

Jump in LC-3

- Unconditional branch or jump

- LC-3

```
JMP   R2
```



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

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

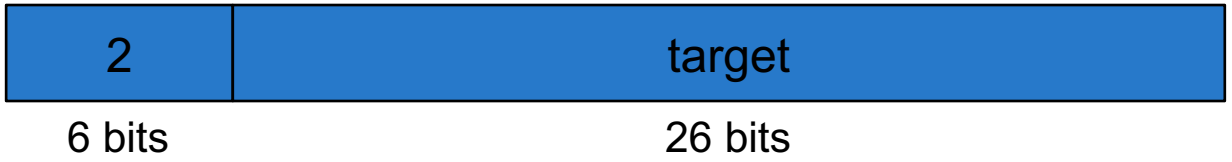
This is register addressing mode

Jump in MIPS

- Unconditional branch or jump

- MIPS

```
j   target
```



J-Type

  - 2 = opcode
  - target = target address

  - PC ← PC † [31:28] | sign-extend(target) * 4

j uses pseudo-direct addressing mode

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

```
jr   $s0
```

jr uses register addressing mode

† This is the incremented PC

PC UPDATE in LC-3

JMP loads SR1 into PC

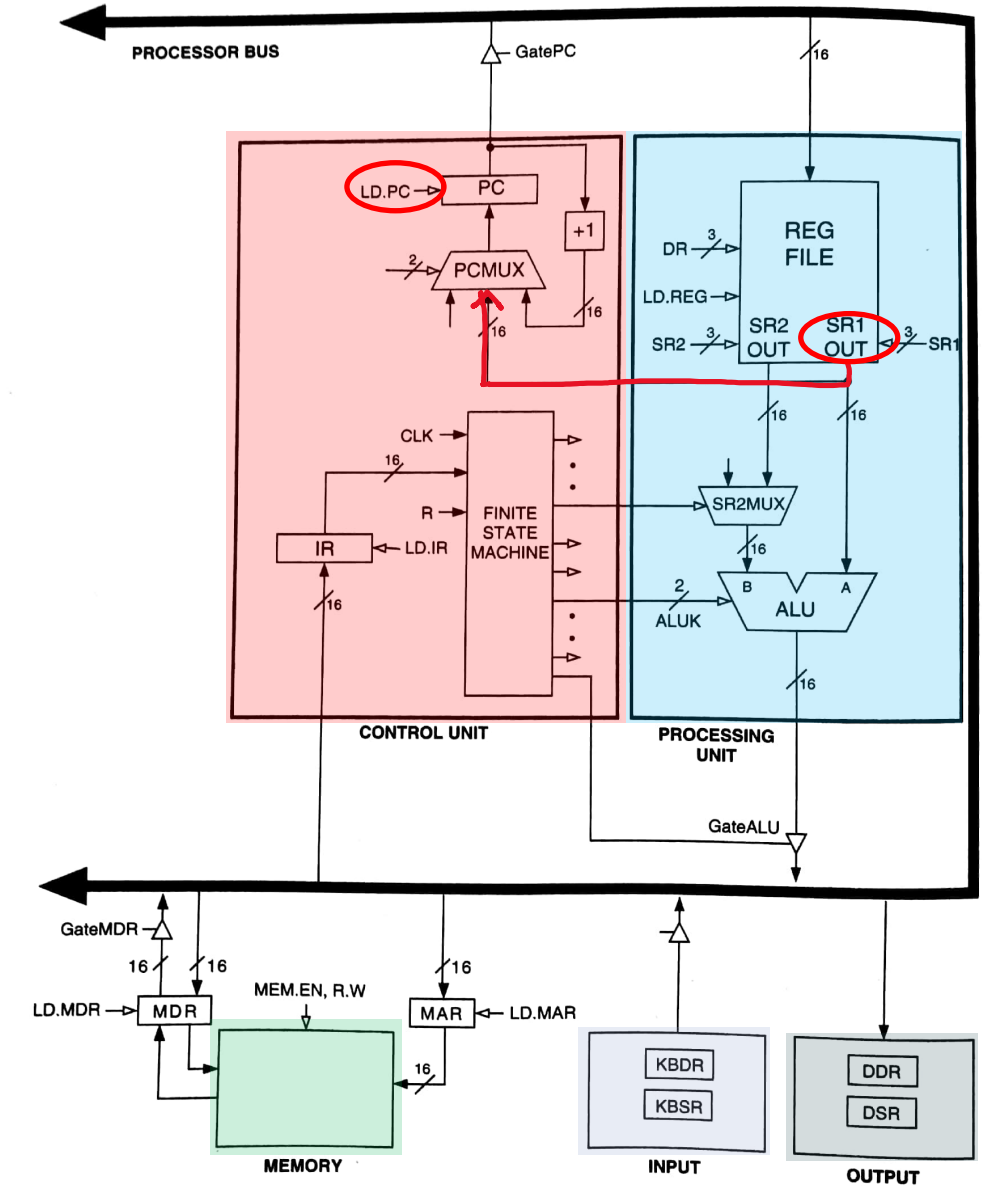


Figure 4.3 The LC-3 as an example of the von Neumann model

KING'S College LONDON

Hardware Design

Lecture 1: Von Neumann Model & Instruction Set Architectures

Dr. Haiyu Mao

22.01.2026