



COMPUTER ORGANIZATION (ECS 409/609)

VON NEUMANN, ISA, LC-3 AND MIPS

Dr. Sukarn Agarwal

EECS
Indian Institute of Science Education and Research Bhopal

Course Outline for Today and Next Few Lecture

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

Required Readings

■ This week

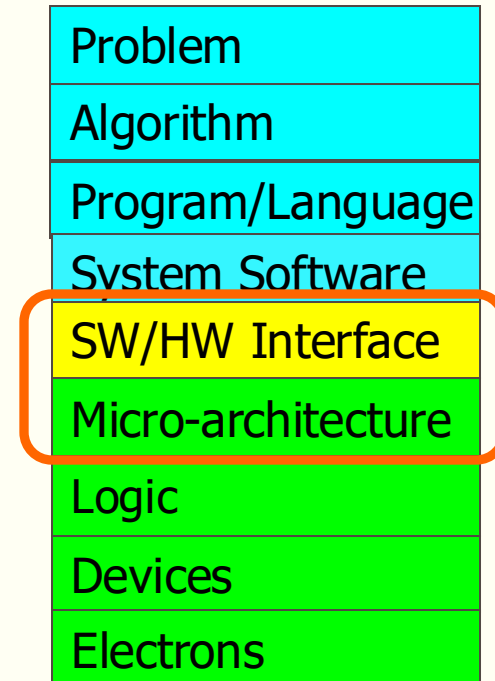
- Von Neumann Model, LC-3, and MIPS
 - P&P, Chapters 4, 5
 - H&H, Chapter 6
 - P&P, Appendices A and C (ISA and microarchitecture of LC-3)
 - H&H, Appendix B (MIPS instructions)
- **Recommended:** H&H Chapter 5, especially 5.1, 5.2, 5.4, 5.5

What We learn today

- The von Neumann model
 - LC-3: An example von Neumann machine
- Instruction Set Architectures: Collection of Instructions
 - Example LC-3 and MIPS
 - Operate instructions
 - Data movement instructions
 - Control instructions
- Instruction formats
- Addressing modes

Basic Elements of a Computer

- In Digital Design and Computer Organization you learned
 - Combinational Elements
 - Sequential Elements
- With them, we can build
 - Decision Units (e.g.: Finite State Machine)
 - Storage Units (e.g.: Register and Memory)
 - Execution Units (e.g. ALU)
 - Communication Units
- Basic elements of a computer
 - We raise our abstraction level today
 - Using logic structures to construct basic computing model



Basic Elements 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

Von Neumann Model

■ Let's start **building the computer**

■ In order to build a computer **we need a plan/model or Specifications.**

■ John von Neumann proposed **a fundamental model or Specifications** in 1946

■ It consists of 5 parts

❑ **Memory**: stores the program and data

❑ **Processing unit**: does the execution

❑ **Input**: To feed the information

❑ **Output**: To return the result

❑ **Control unit**: Orchestrate the components or control the order in which the instructions are carried out.



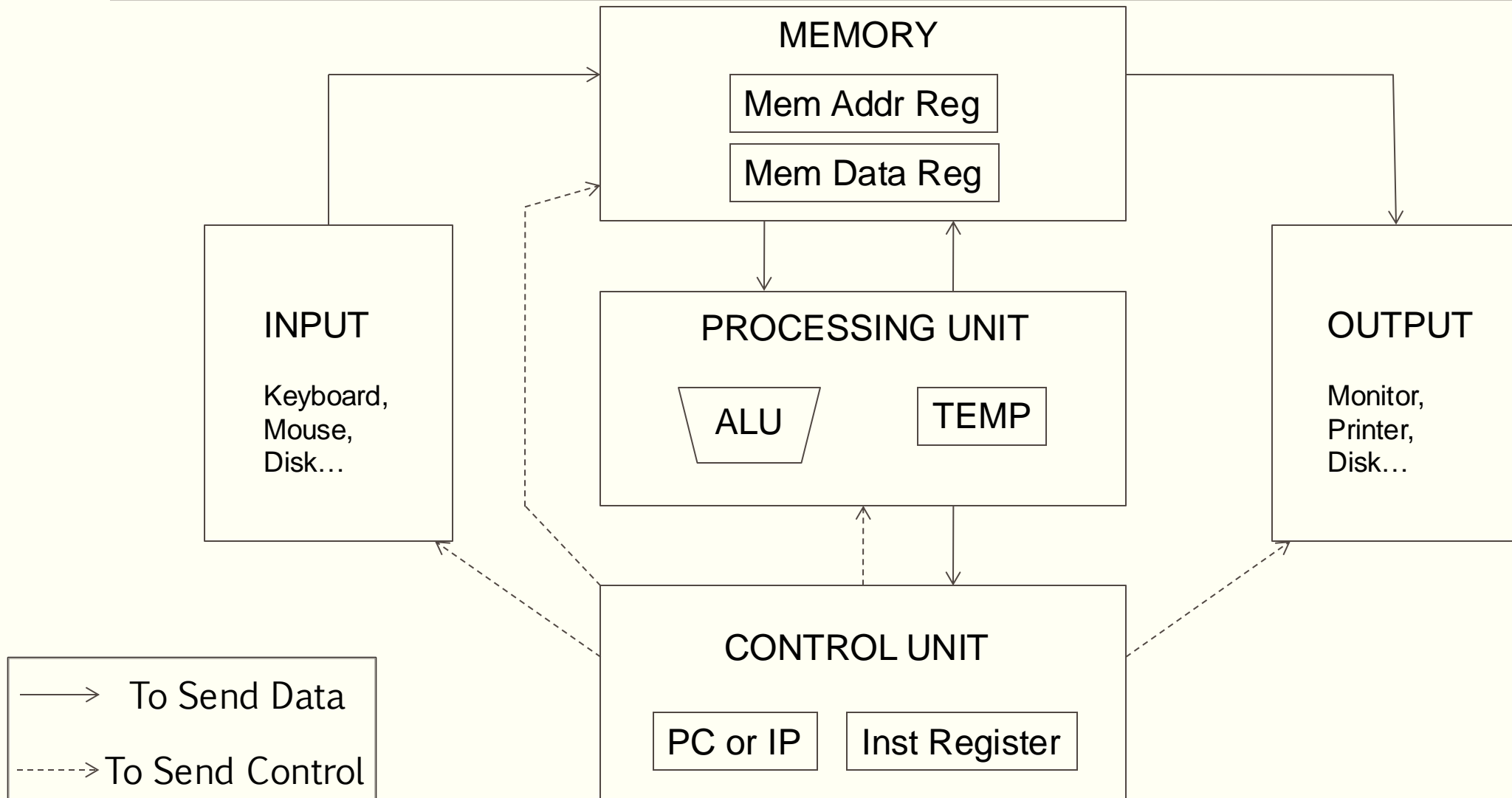
■ Throughout this lecture, we consider two isa examples of the von Neumann model

❑ LC-3 (or LC3-b, P&P)

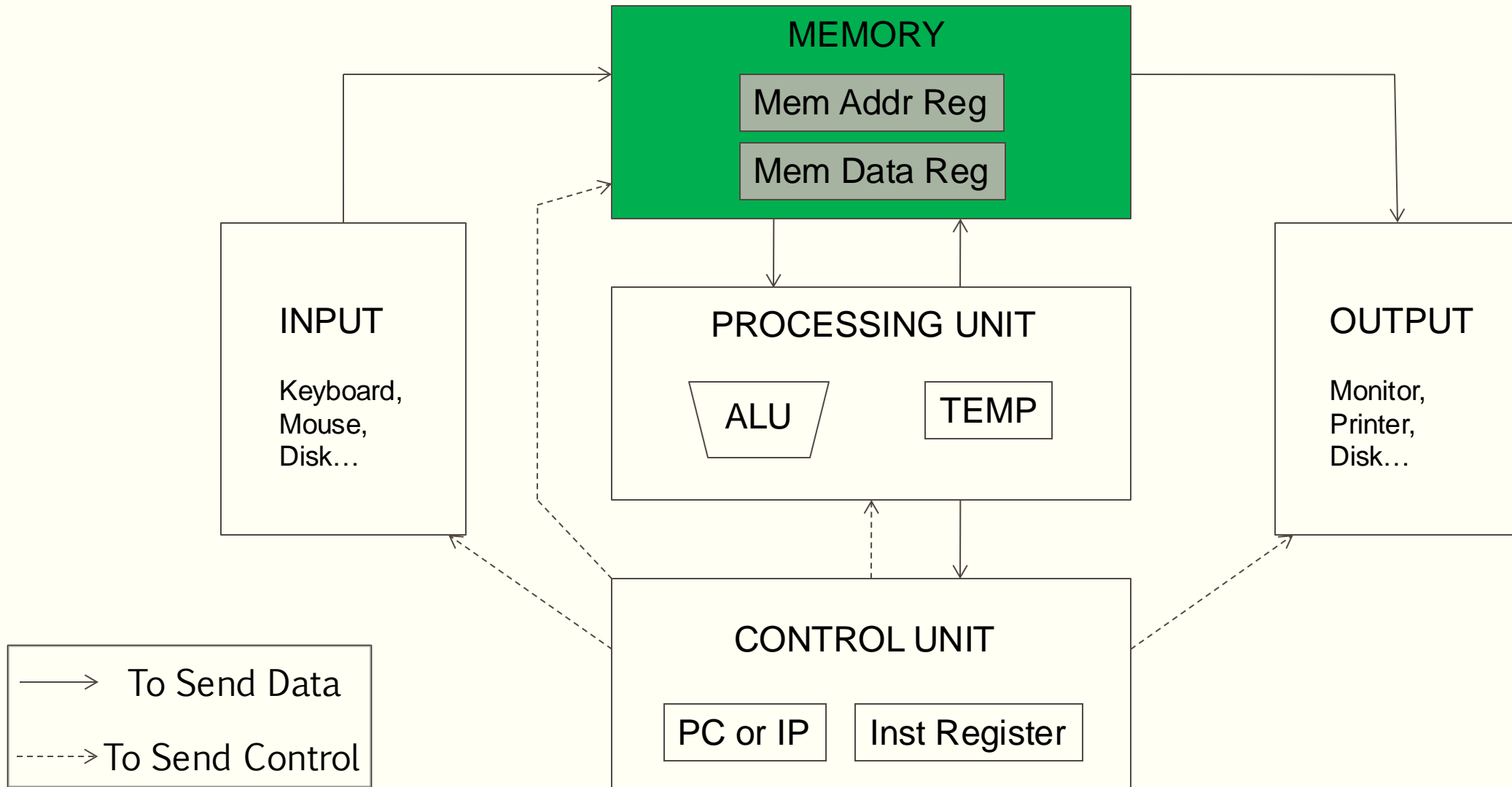
❑ MIPS (H&H)

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

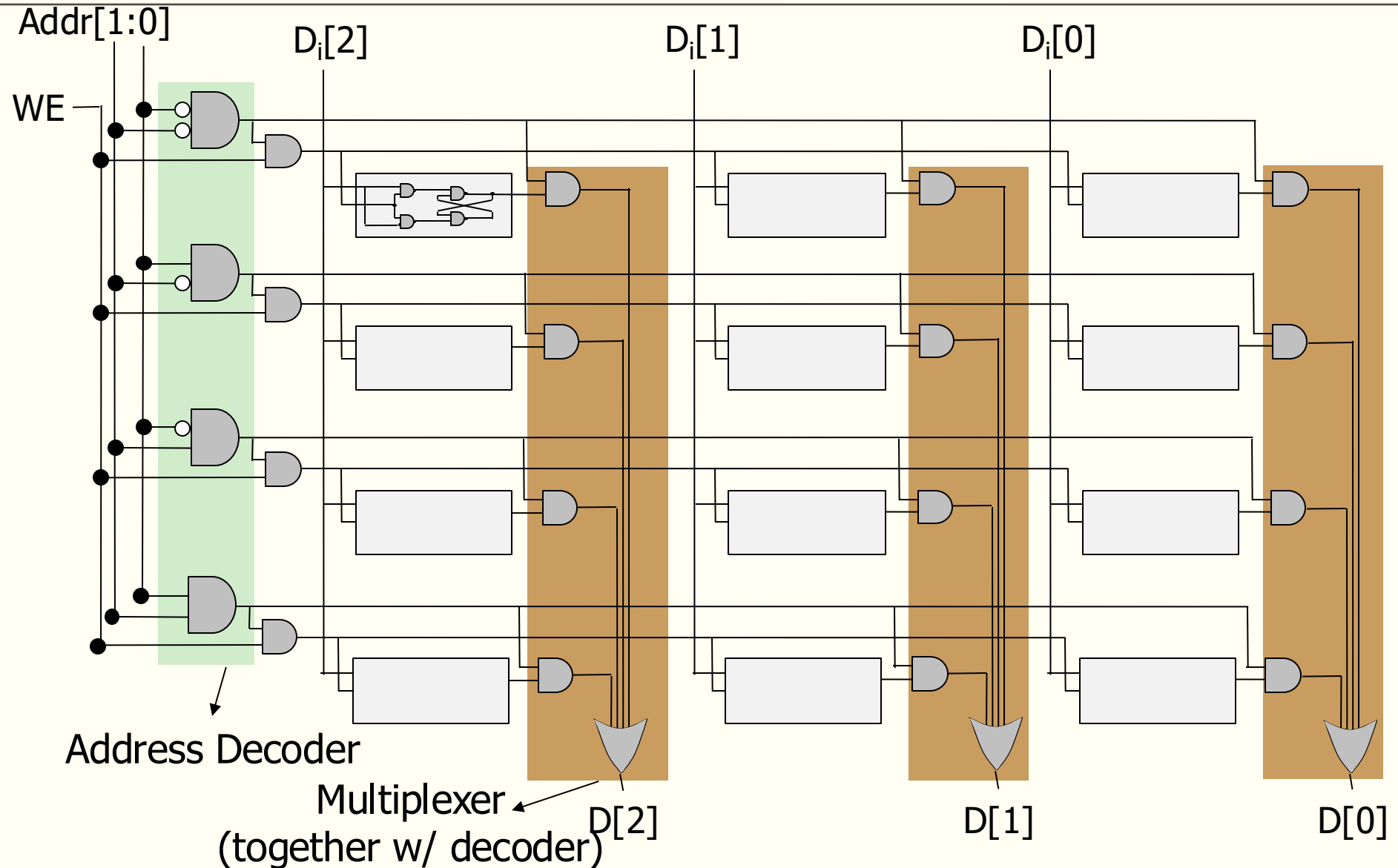
Von Neumann Model



Von Neumann Model



Recall: Memory Array (4 locations x 3 Bits) Underneath



Memory

- The memory stores

- Data
- Programs

- The memory contains **bits**

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

- How the bits are accessed (for reading and writing) determines the **addressability**

- E.g., **word-addressable**: With one address, we access one word.
- E.g., 8-bit addressable (or **byte-addressable**): With one address, we access one byte.

- The total number of addresses (or total size of memory) is the **address space**

- In **LC-3**, the address space is 2^{16} (Word Addressable)
 - 16-bit addresses
- In **MIPS**, the address space is 2^{32} (Word Addressable)
 - 32-bit addresses
- In **x86-64**, the address space is (up to) 2^{48}
 - 48-bit addresses

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 (Let suppose that), 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

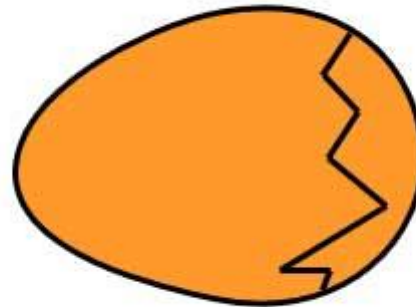
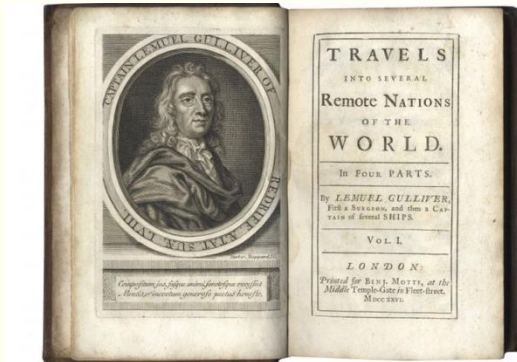
- Each **byte** has a **unique address**
 - Actually, MIPS is **byte-addressable**
 - LC-3b (updated version of LC-3) is **byte-addressable**, too

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

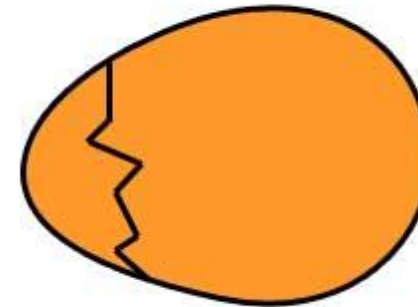
Which of the four bytes is most or least significant

Big Endian vs Little Endian

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



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



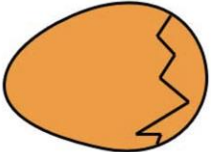
LITTLE ENDIAN - The way the king then ordered the people to break their eggs

Memory Types: Big Endian vs Little Endian

Big Endian

Byte
Address

·
·
·



C	D	E	F
8	9	A	B
4	5	6	7
0	1	2	3

MSB

(Most Significant Byte)

LSB

(Least Significant Byte)

LSB in higher byte address

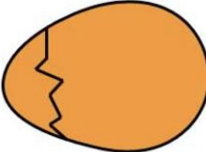
Word
Address

·
·
·
C
8
4
0

Little Endian

Byte
Address

·
·
·



F	E	D	C
B	A	9	8
7	6	5	4
3	2	1	0

MSB

LSB

LSB in lower byte address

Memory Types: 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 (SPARC)** and **one little-endian system (X86)** have to **share data**. Need to keep in mind about these issues.

MSB

(Most Significant Byte)

LSB

(Least Significant Byte)

MSB

LSB

Accessing Memory: MAR and MDR

- There are two ways of **accessing memory**

- Reading or loading

- Writing or storing

- **Two registers** are necessary to access memory

- Memory Address Register (**MAR**)

- Memory Data Register (**MDR**)

- **To read**

- Step 1: Load the **MAR with the address**

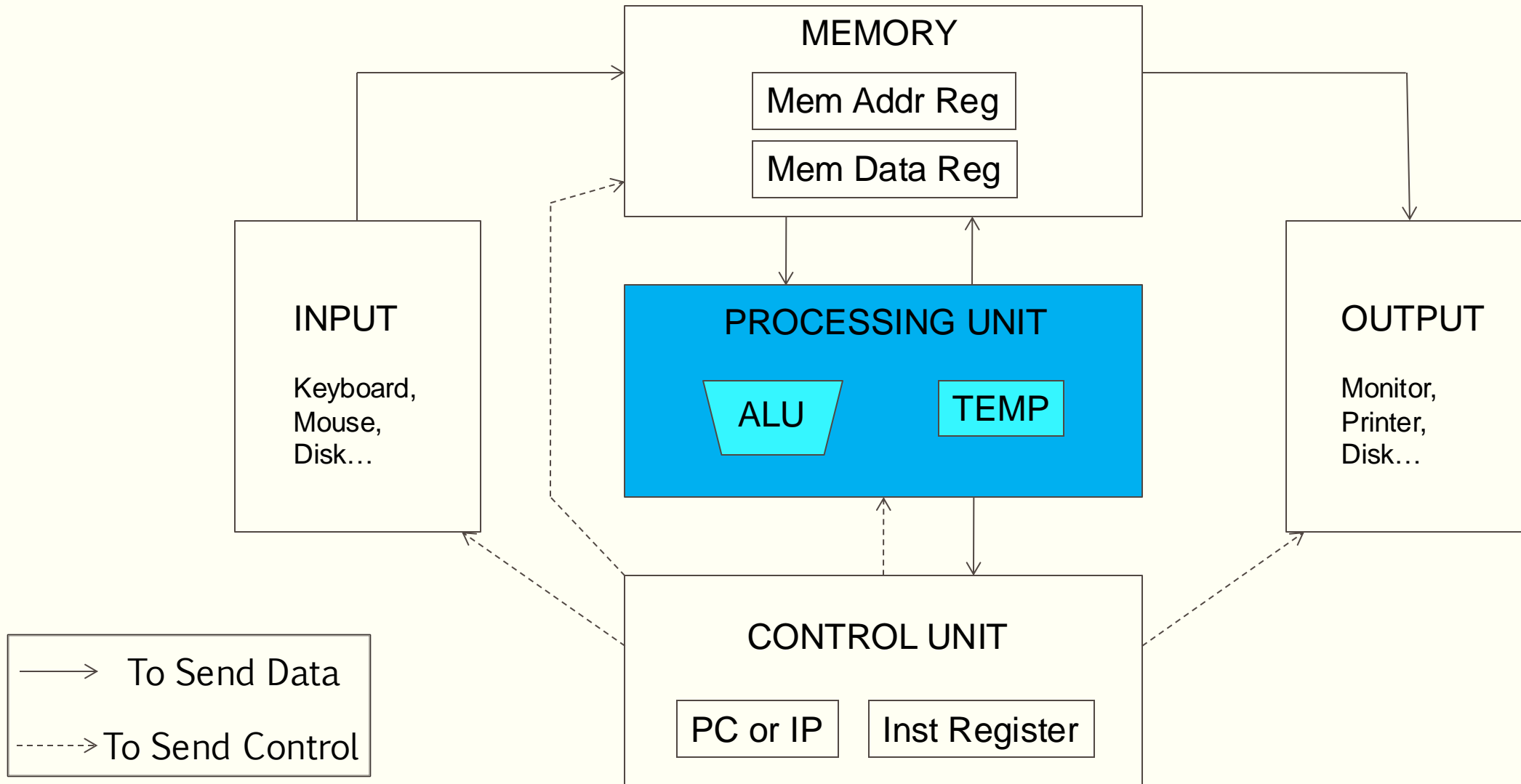
- Step 2: **Data** is placed **in MDR**

- **To write**

- Step 1: Load the **MAR with the address** and the **MDR with the data**

- Step 2: Activate **Write Enable** signal

The Von Neumann Model



Processing Unit

- Perform the actual computation.
- The processing unit can consist of many **functional units** like integer, floating point, vector processing units, etc.
- We start with a simple **Arithmetic and Logic Unit (ALU)**, which executes computations
 - **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 is of same size that of address)
 - In MIPS it is 32 bits

Recall: Arithmetic and Logic Unit (ALU)

- 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:

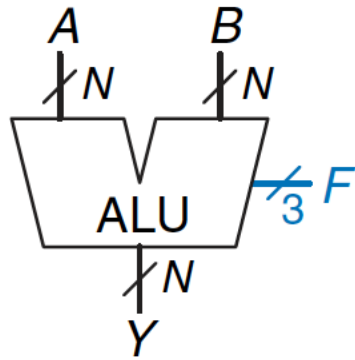


Figure 5.14 ALU symbol

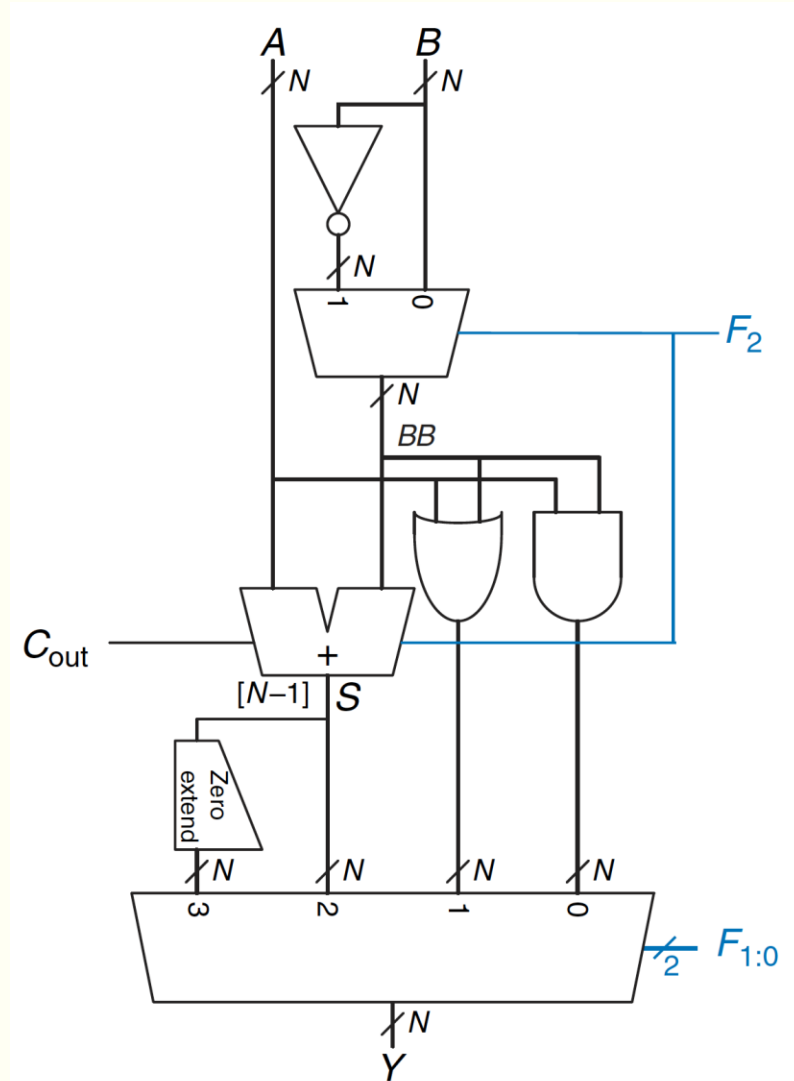
Table 5.1 ALU operations

$F_{2:0}$	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

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

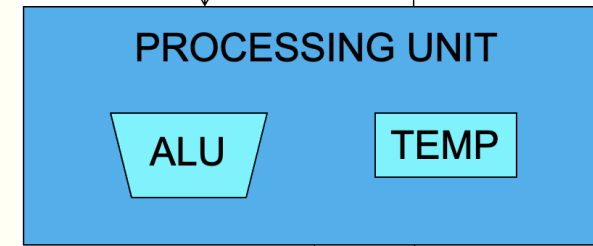


Processing Unit: Fast Temporary Storage

- It is almost always the case that a computer provides a small amount of storage very close to ALU
 - Purpose: to store temporary values (visible to programmer) 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 (programmer visible, saw, address or manipulate)

Registers: Why we use it?

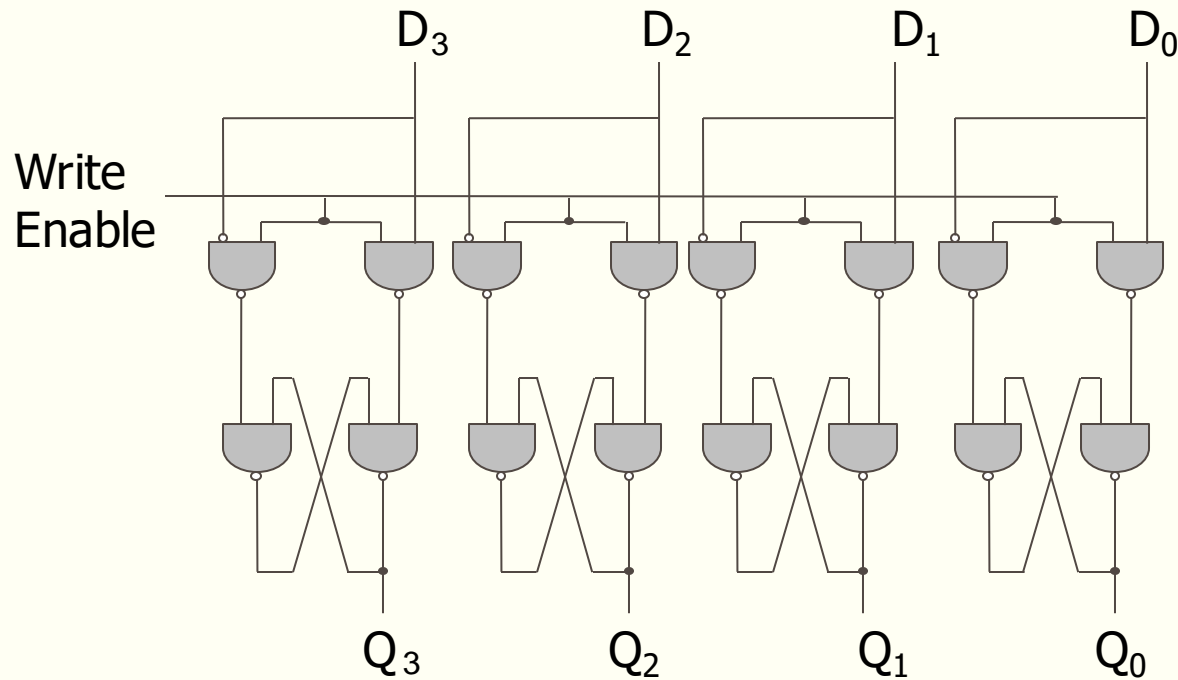
- **Memory** is big but slow
- **Registers** in the processing unit (small in Number)
 - Ensure fast access to value to processed in the ALU
 - Typically one register contains **one word** (same as the word length)
 - ALU operate on word length
- **Register set or file: Collection or Set of registers that the processing unit has or set of register that is manipulated by instructions**
 - LC-3 has 8 **general purpose registers** (GPR)
 - **R0 to R7**: 3-bit register number
 - Register size = Word length = 16 bits
 - MIPS has 32 registers
 - Register size = Word length = 32 bits



Recall: The Registers

How can we use D latches to store **more** data?

- Use **more** D latches!
- A single WE signal for all latches for simultaneous writes



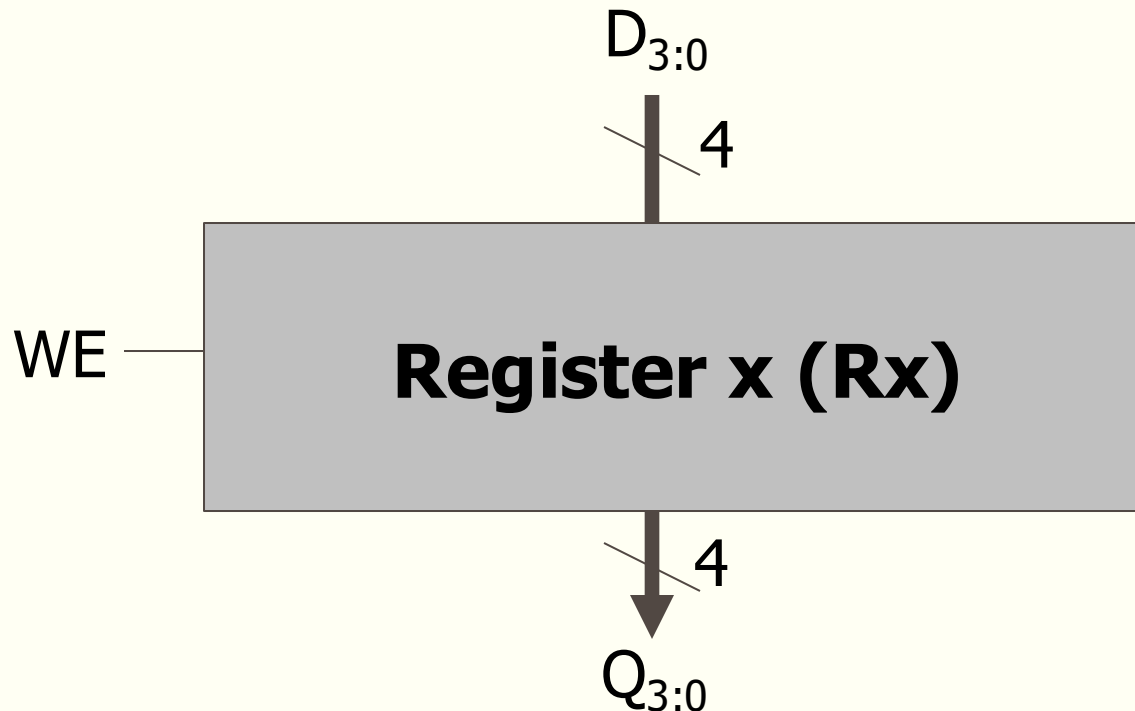
Here we have a **register**, or a structure that stores more than one bit and can be read from and written to

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

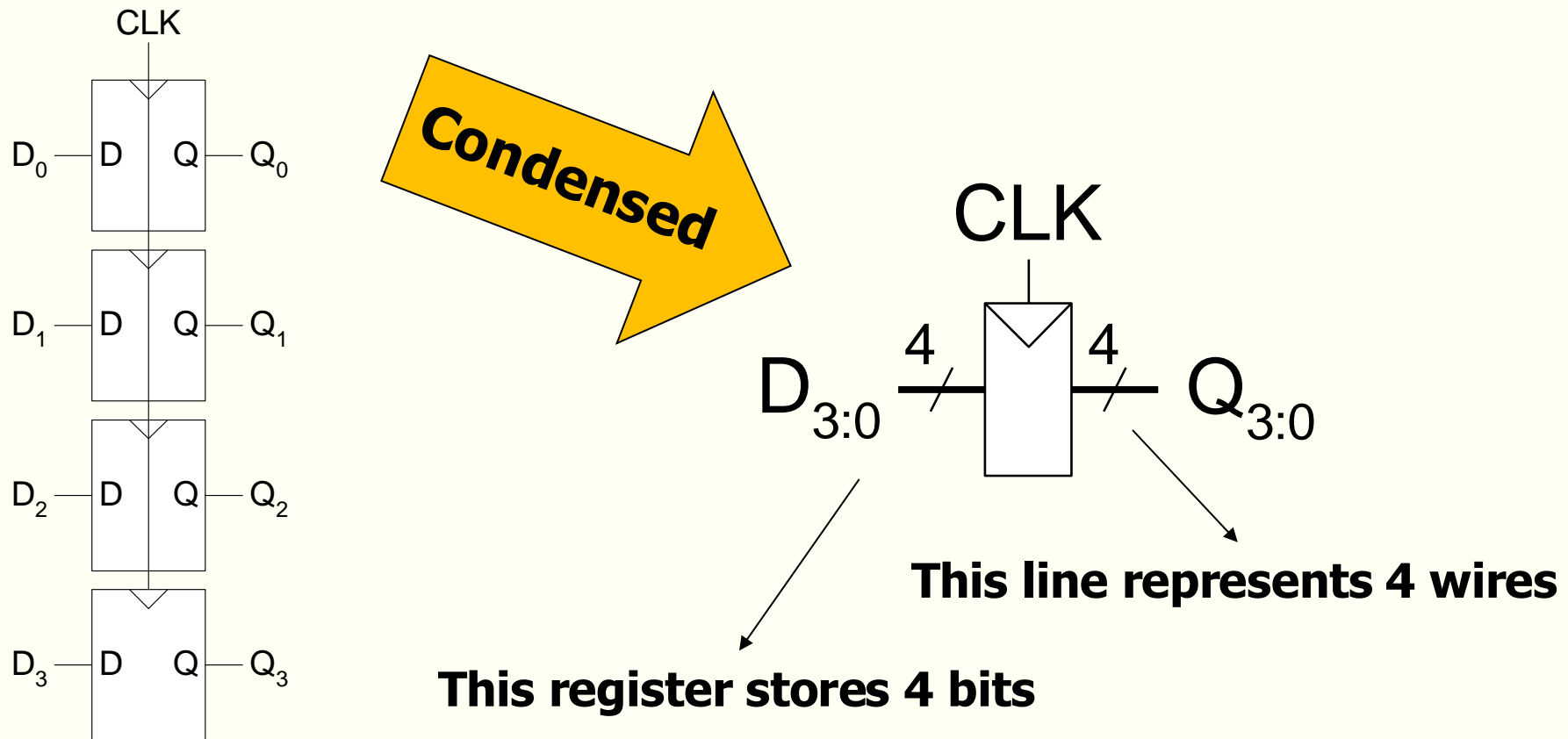


Here we have a **register**, or a structure that stores more than one bit and can be read from and written to

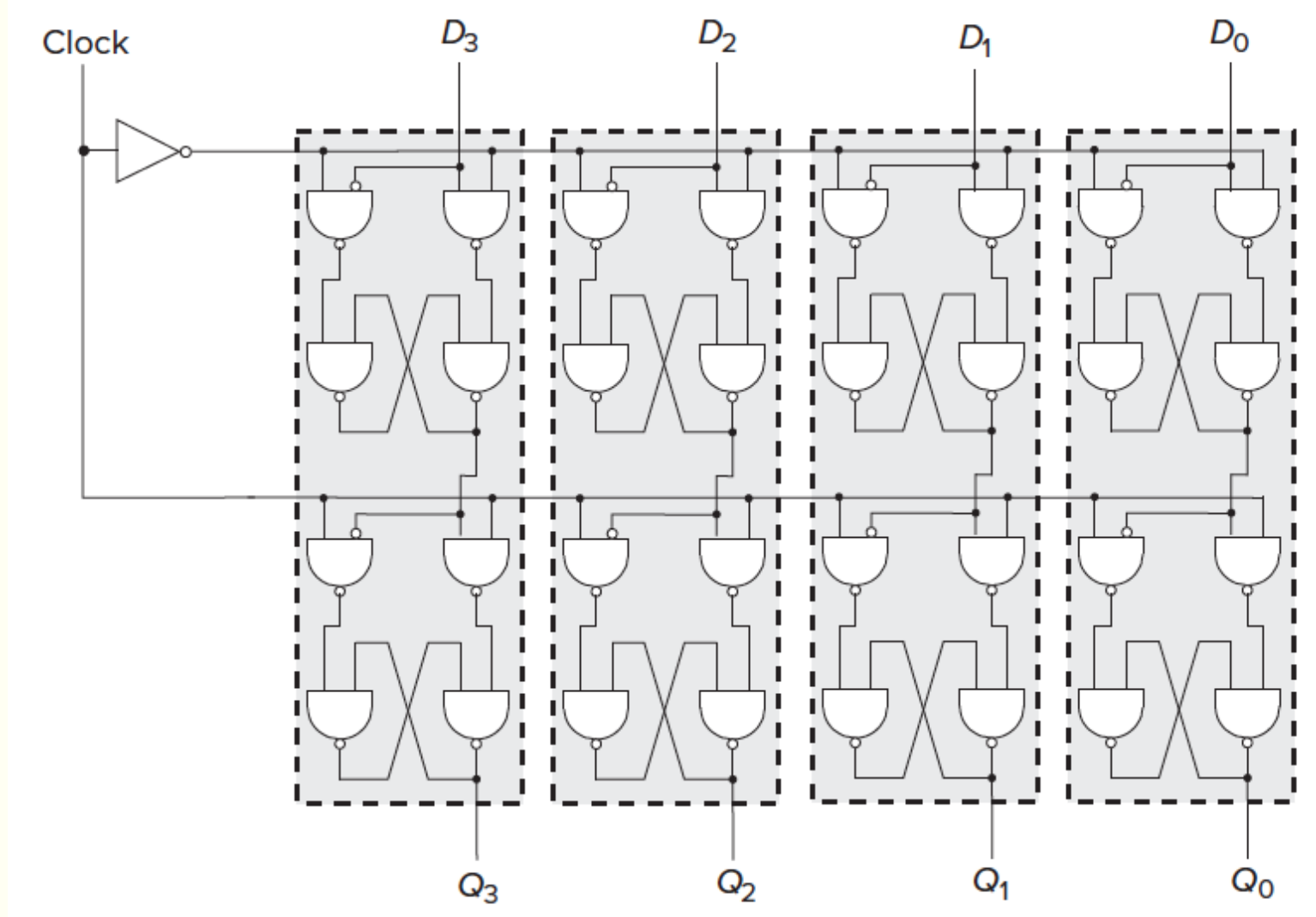
This **register** holds 4 bits, and its data is referenced as Q[3:0]

Recall: D Flip Flop Based Register

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



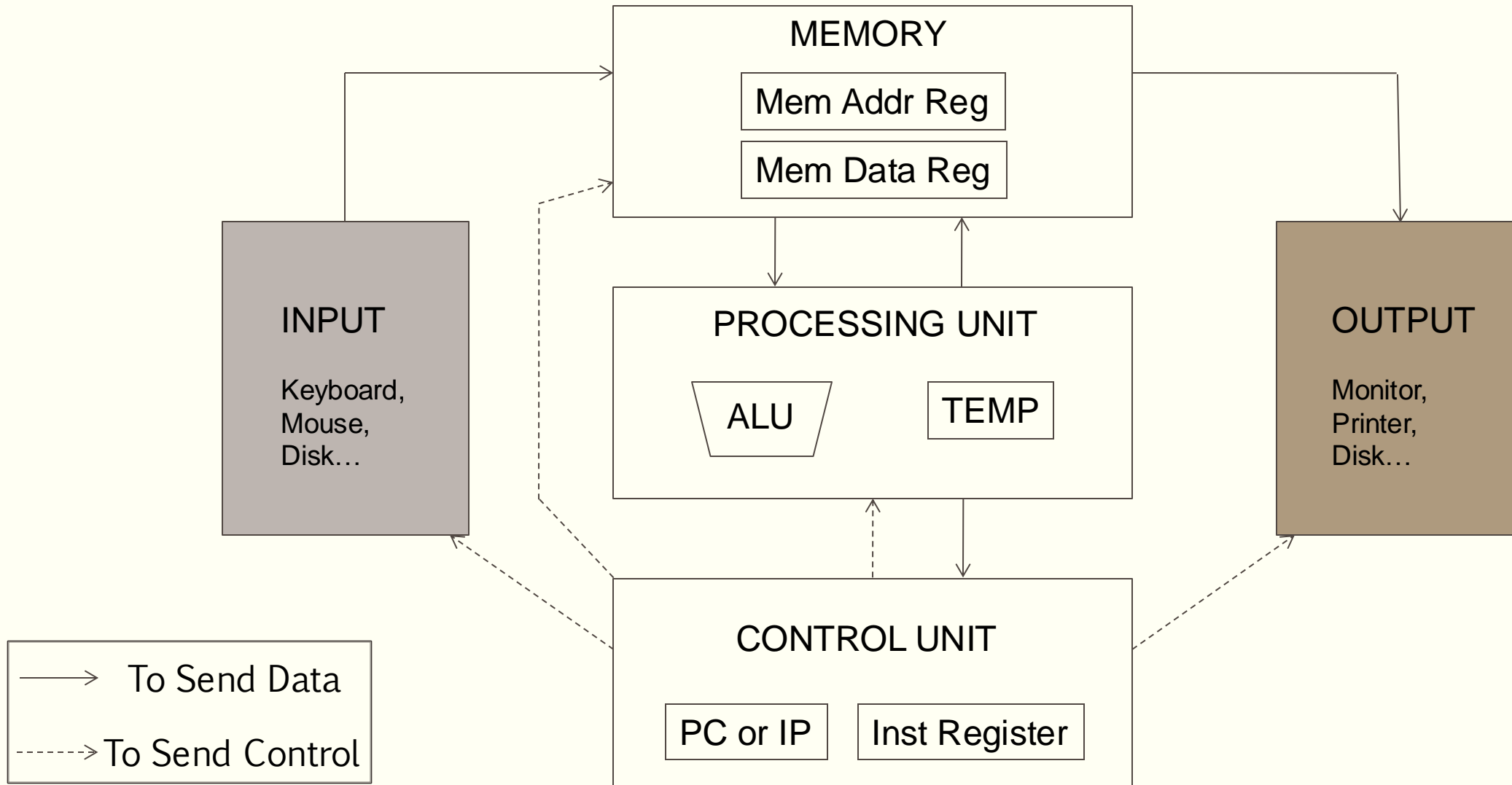
Recall: A 4-Bit D-Flip-Flop-Based Register (Internally)



MIPS Register File

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

The Von Neumann Model



Input and Output

- Interface through which we interact with the computer
- They are also called as **peripherals**
- Many devices can be used for input and output.

- **Input**

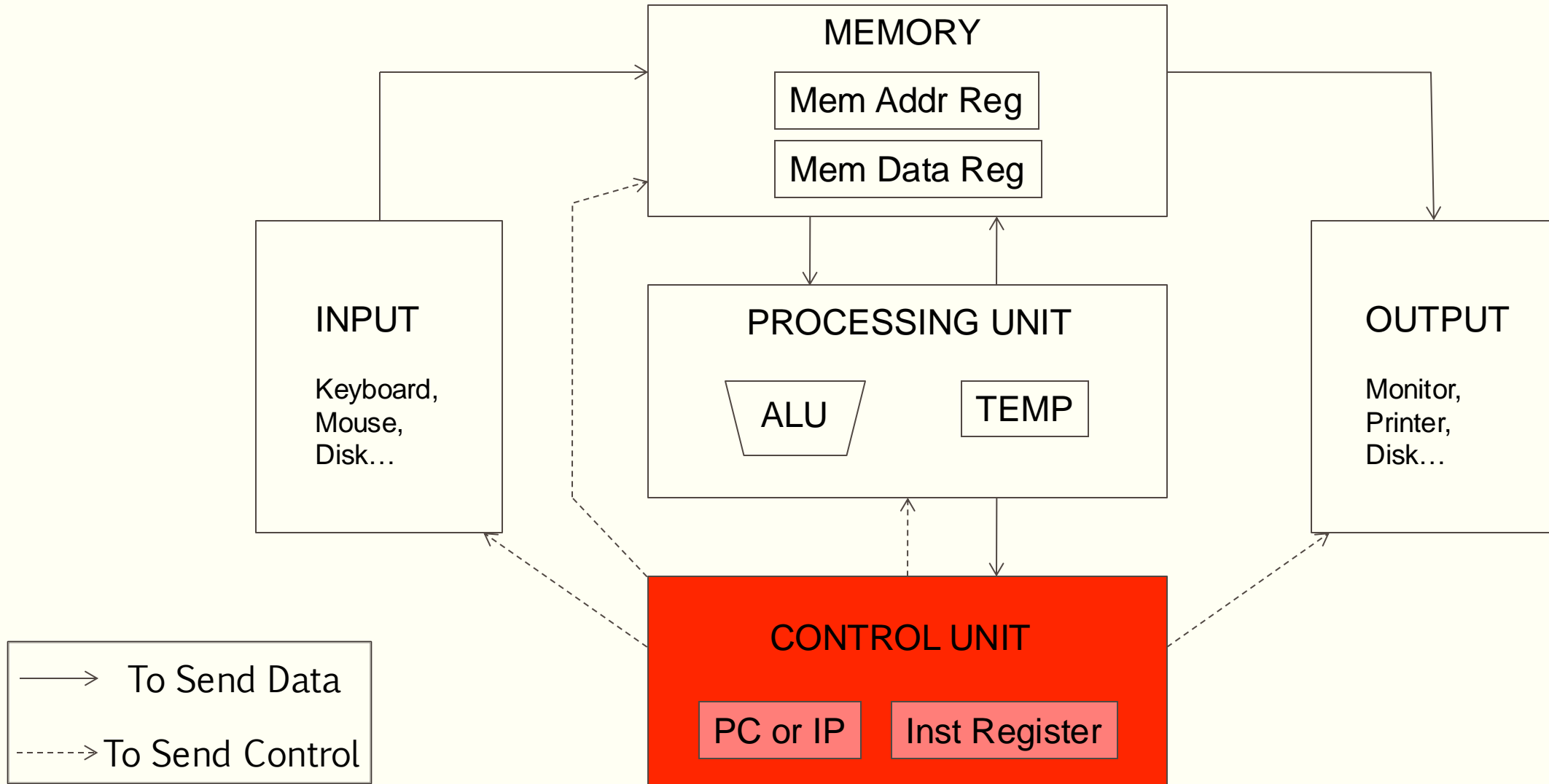
- **Keyboard**
- Mouse
- Scanner
- Disks
- Etc.

- **Output**

- **Monitor**
- Printer
- Disks
- Etc.

- In LC-3, we consider keyboard and monitor to understand the basic principles

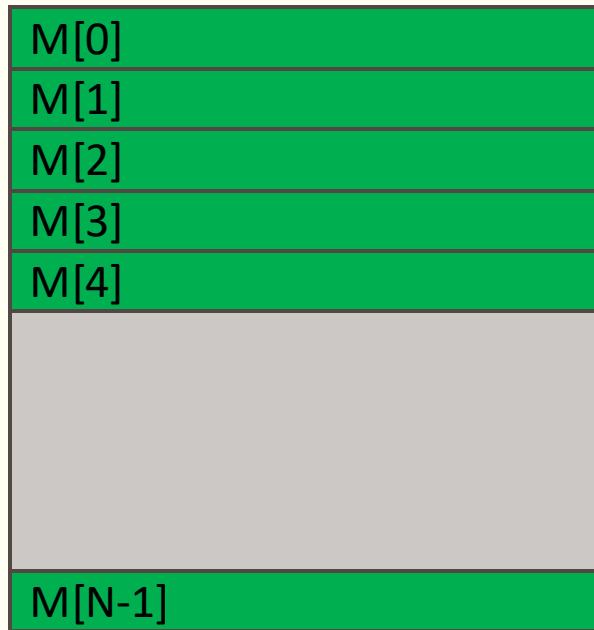
The Von Neumann Model



Control Unit

- The control unit is similar to the conductor of an orchestra (like the brain in the computer)
- It conducts the **step-by-step process of executing (every instruction in) a program**
- It keeps track of the instruction being executed with an **Instruction Register (IR)**, which contains the bit encoding of 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

Program Visible (Architectural) State

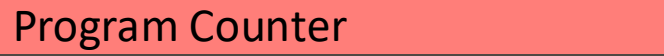


Memory
array of storage locations
indexed by an address



Registers

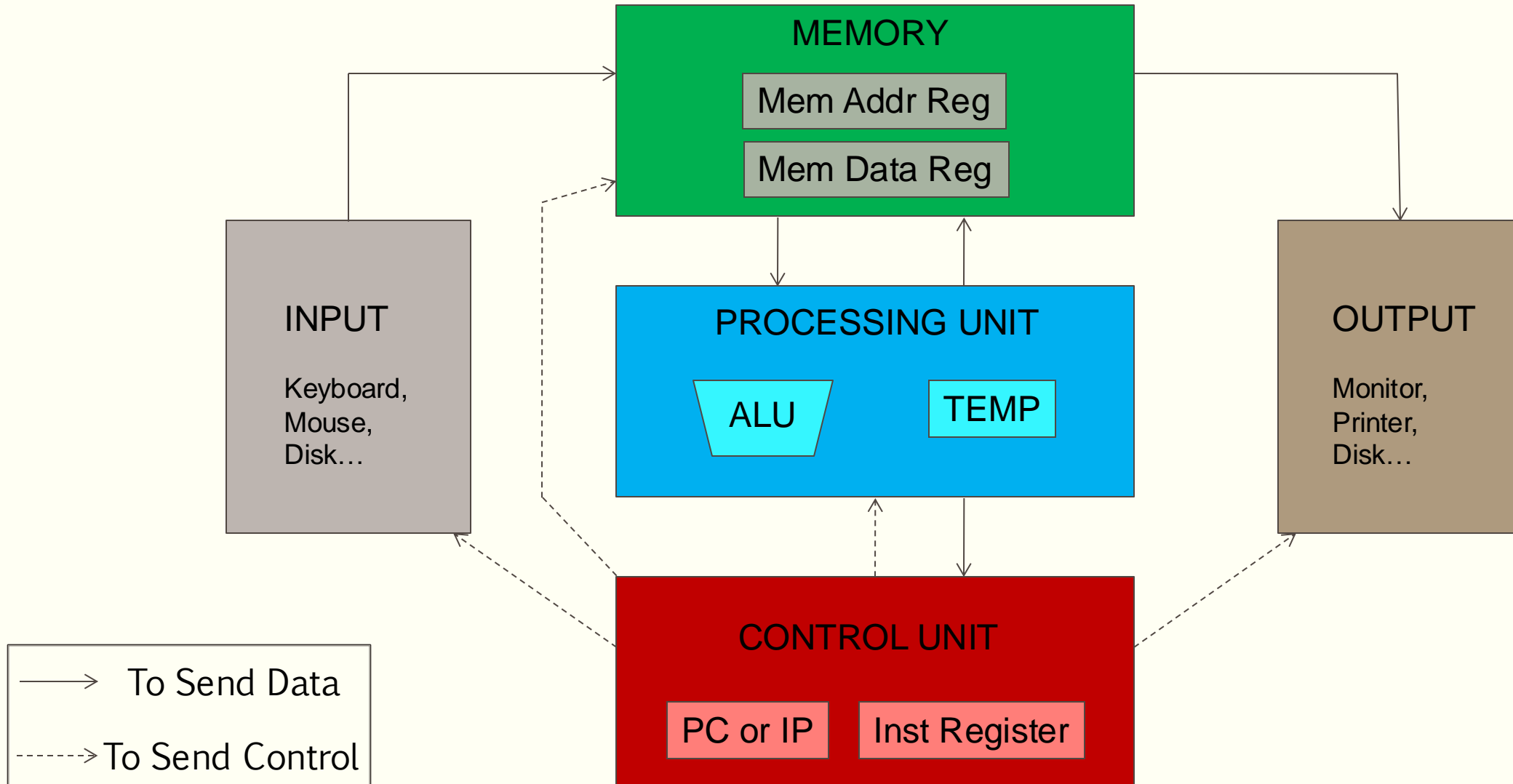
- given special names in the ISA
(as opposed to addresses)
- general vs. special purpose



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

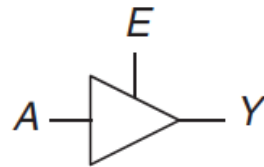
- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:
 - **Stored program**
 - Instructions stored in a linear memory array
 - **Memory is unified** between instructions and data
 - The interpretation of a stored values (both instruction and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

Tri State Buffer

A tri-state buffer enables gating of different signals onto a wire

**Tristate
Buffer**



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

**A tri-state buffer
acts like a switch**

Figure 2.40 Tristate buffer

Floating signal (Z): Signal that is not driven by any circuit Open circuit, floating wire

Example: Use of Tri-State Buffer

- Imagine a wire connecting the CPU and memory
 - At any time only the CPU or the memory can place a value on the wire, both not both
 - You can have two tri-state buffers: one driven by CPU, the other memory; and ensure at most one is enabled at any time

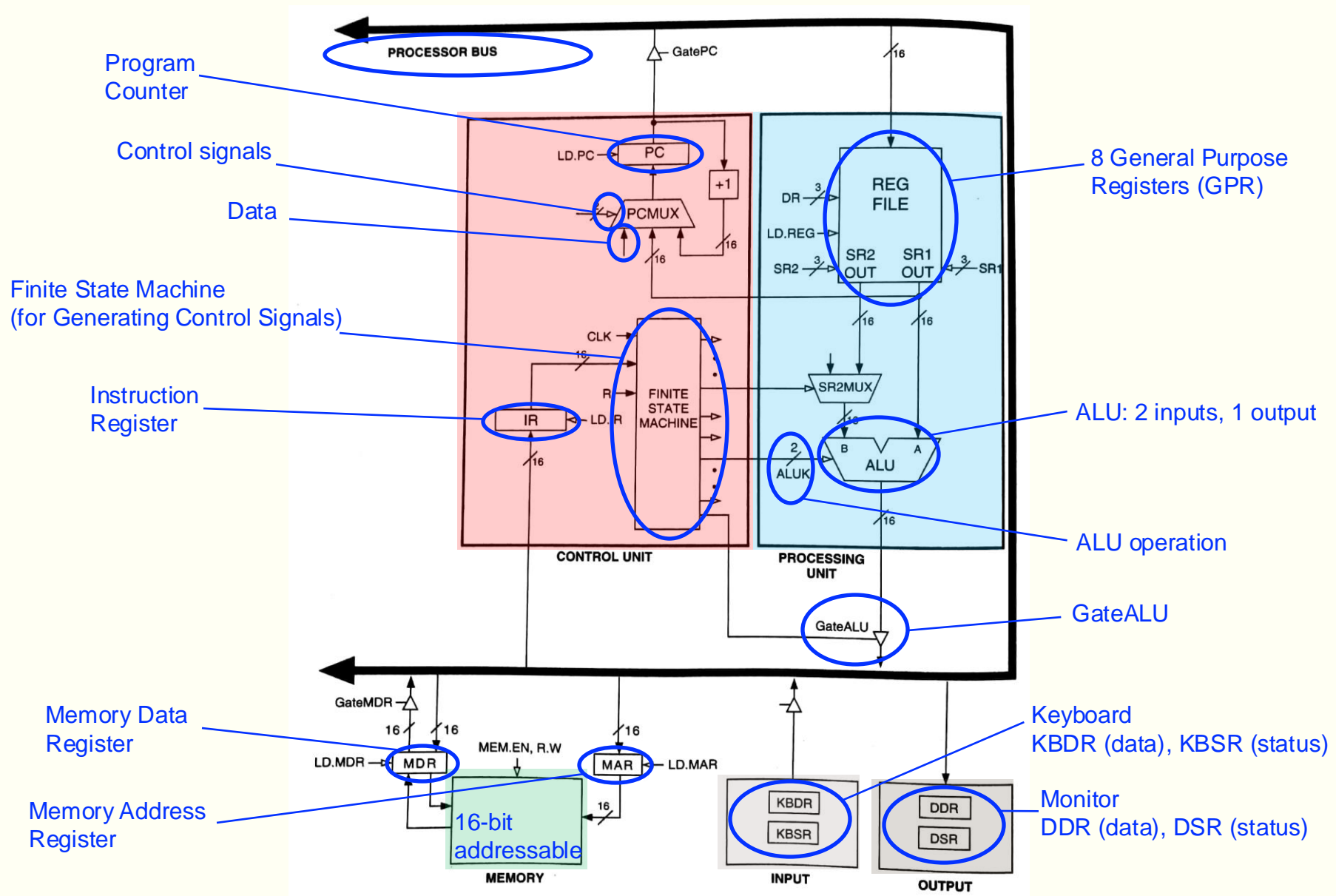


Figure 4.3 The LC-3 as an example of the von Neumann model

Stored Program and 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 and write back the result
 - Continues with the next instruction

(called as **Instruction cycle**)

- 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 word length** (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

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

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 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 look at **LC-3 instructions** and **MIPS instructions**

- Let us start with some example instructions

The Instruction: Opcode and Operand

- 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 instructions in the ALU
- **Data movement instructions**
 - Read from or write to memory
- **Control flow instructions**
 - Change the sequence of execution

Operate Instruction Example

- 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
- $a \leftarrow b + c$ (Semantic)
- A generic representations is presented here

Representation With LC-3 and MIPS

- The variables are mapped to different register values.

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

Instruction Encoding Format:

OP	DR	SR1			SR2
1	0	1	0	00	2

Machine Code

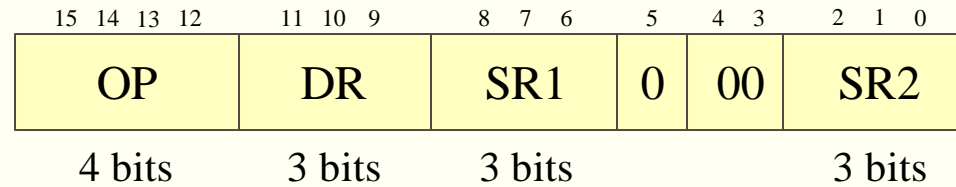
OP	DR	SR1			SR2
0001	000	001	0	00	010
15 14 13 12	11 10 9	8 7 6	5	4 3	2 1 0

0x1042

Machine Code, in short (hexadecimal)

Instruction Format (or Encoding)

- LC-3



- OP = **opcode** (what the instruction does, different for different instructions)
 - 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

Instruction Format:

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

16 (s0), 17 (s1) and 18 (s2) Represent the Register number

$rd \leftarrow rs + rt$

Machine Code

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

31 26 25 21 20 16 15 11 10 6 5 0

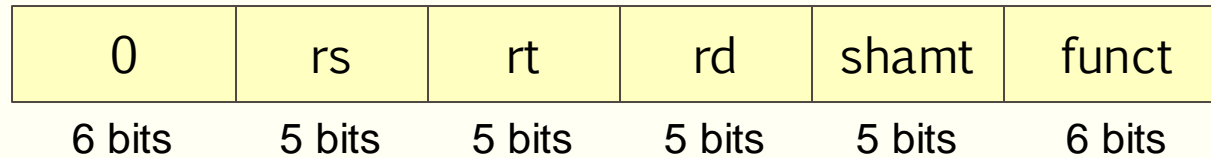
0x02328020

Instruction Format: R-type in MIPS

■ R-type : that uses register as an operands (one type of operate inst. format)

□ 3 register operands

■ MIPS



□ 0 = opcode (only for R-type)

□ rs, rt = source registers

□ rd = destination register

□ shamt = shift amount (only shift operations)

□ funct = operation in R-type instructions (extended opcode)

Reading Operands from Memory

- With the **operate instructions**, such as addition, we tell the processor to **execute arithmetic (or logic) computations** in the ALU
- We also need instructions to **access the operands from memory**. (MIPS and LC3 are register to register architectures X86 allows memory operations)
 - **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];
```

Generic Assembly

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

- **load**: mnemonic to indicate the load word operation
- **A**: base address
- **i**: offset, determine where the element of array is located
 - 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
```

LDR: Load Data Register

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

- MIPS assembly (Assuming MIPS is 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** (useful when matrix or array need to be operated)

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

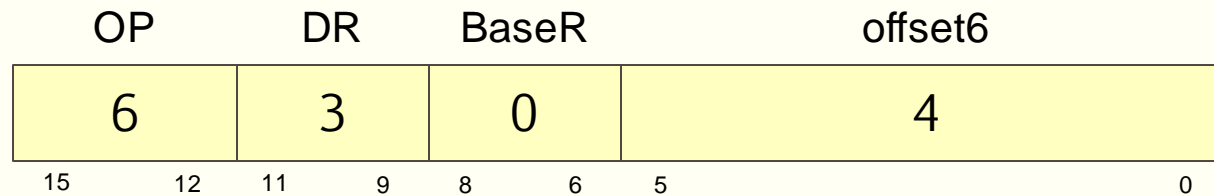
Instruction Format with Immediate

- LC-3

LC-3 assembly

```
LDR R3, R0, #4
```

Field Values

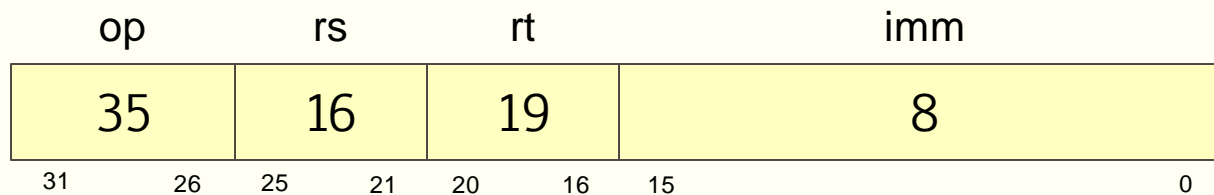


- MIPS

MIPS assembly

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

Field Values



I-Type

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**

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 have the six phases

- LDR does not require EXECUTE

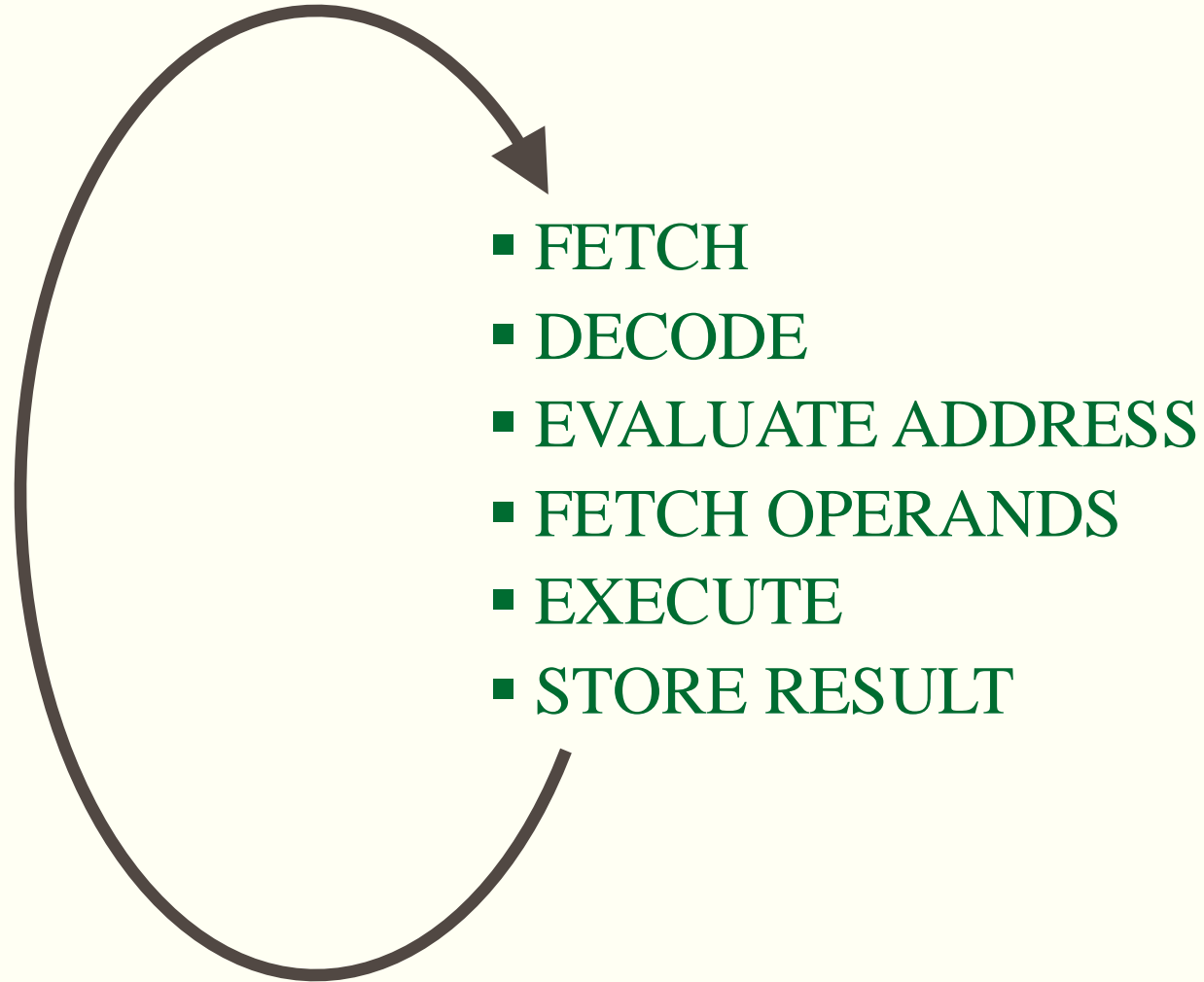
- ADD does not require EVALUATE ADDRESS

- Intel x86 instruction **ADD [eax], edx** is an example of instruction with six phases

- eax is the register that contain the address.

- $edx \leftarrow \text{Mem}[\text{eax}] + \text{edx}$

The Instruction Cycle (After Store Result, a new Fetch)



Fetch

- The FETCH phase obtains the instruction from memory and loads it into the **instruction register**

- This phase is **common to every instruction type. No difference between the operate, data movement and control instructions**
 - To execute every instruction, we need to get the instruction bits.

- **Complete description**
 - Step 1: **Load the MAR with** the contents of the **PC**, and simultaneously **increment the PC**

 - Step 2: Look up the memory for the address stored in the MAR. This results the **instruction to be placed in the MDR**

 - Step 3: **Load the IR** with the contents of the **MDR**

- Step 1: Load MAR and increment PC
- Step 2: Access memory and Load MDR
- Step 3: Load IR with the content of MDR

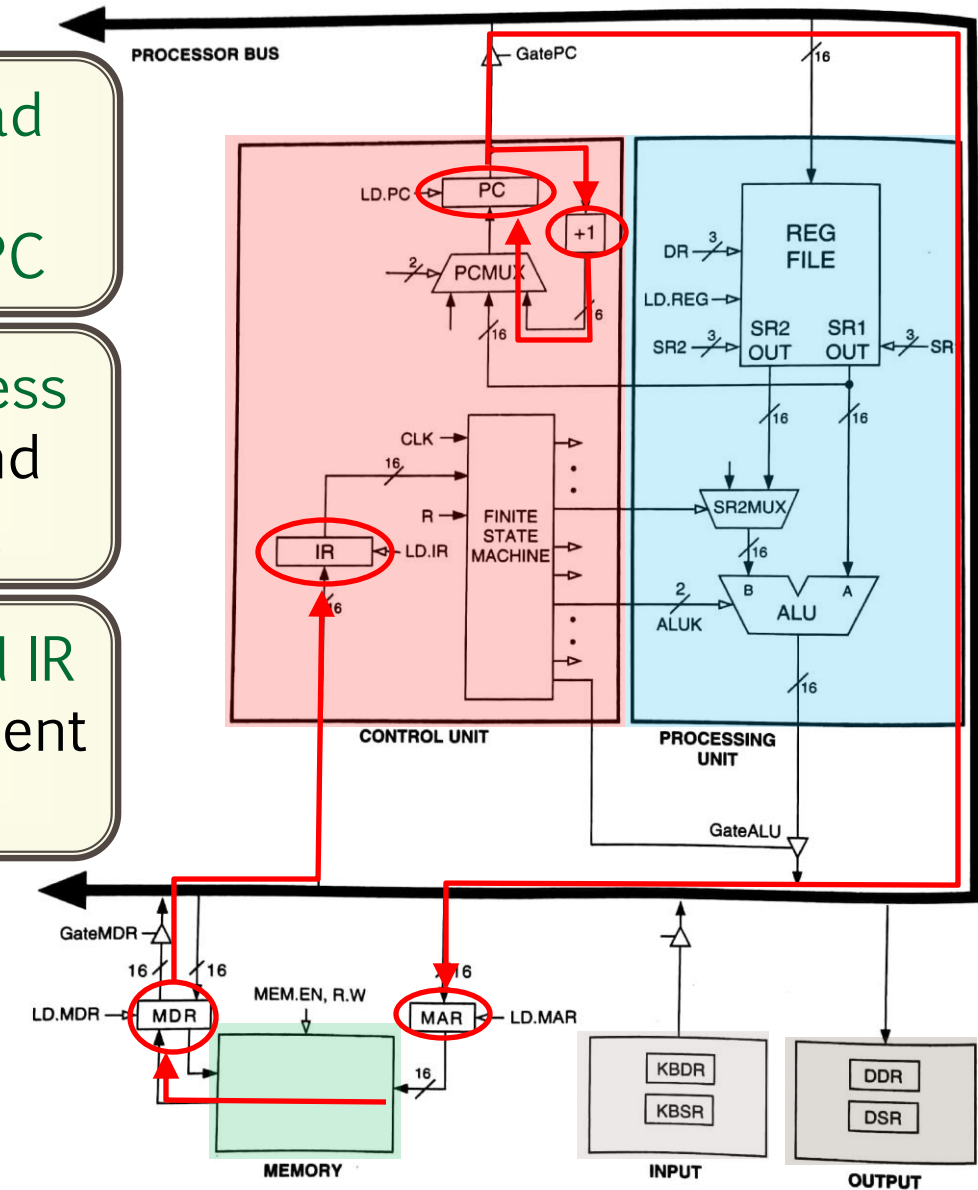


Figure 4.3 The LC-3 as an example of the von Neumann model

Decode

- The DECODE phase **identifies the instruction**
- The decoding of an instruction is done in the control unit
 - 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] (Most Significant Bit of the instruction)**
- The remaining 12 bits identify what else is needed to process the instruction

DECODE
identifies the
instruction to
be processed

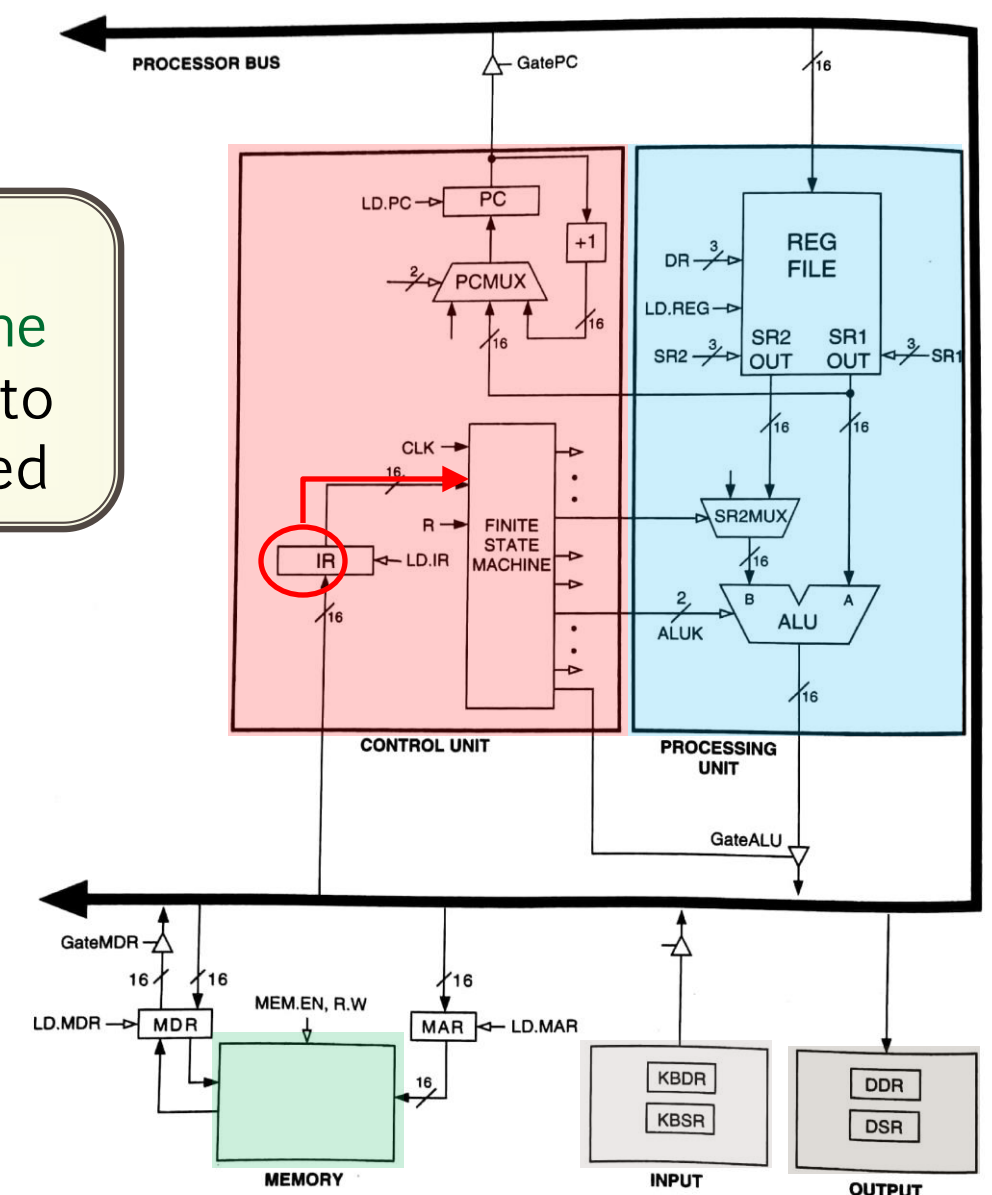
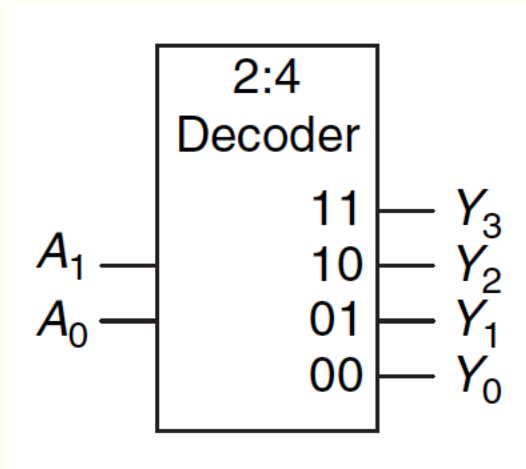


Figure 4.3 The LC-3 as an example of the von Neumann model

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

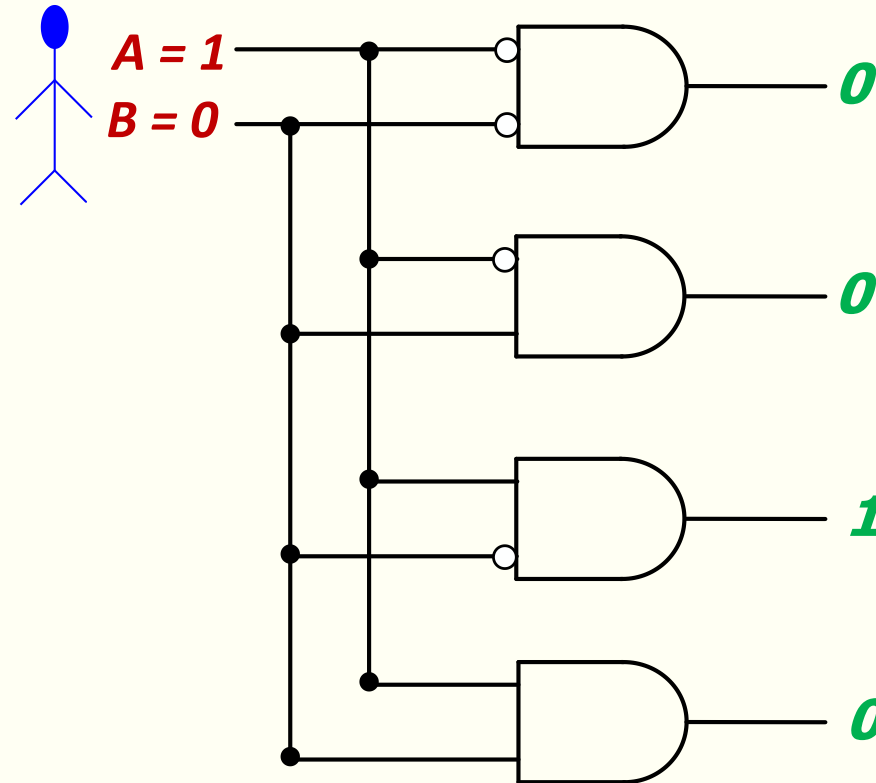


Recall: Decoder (II)

- 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

Decode State

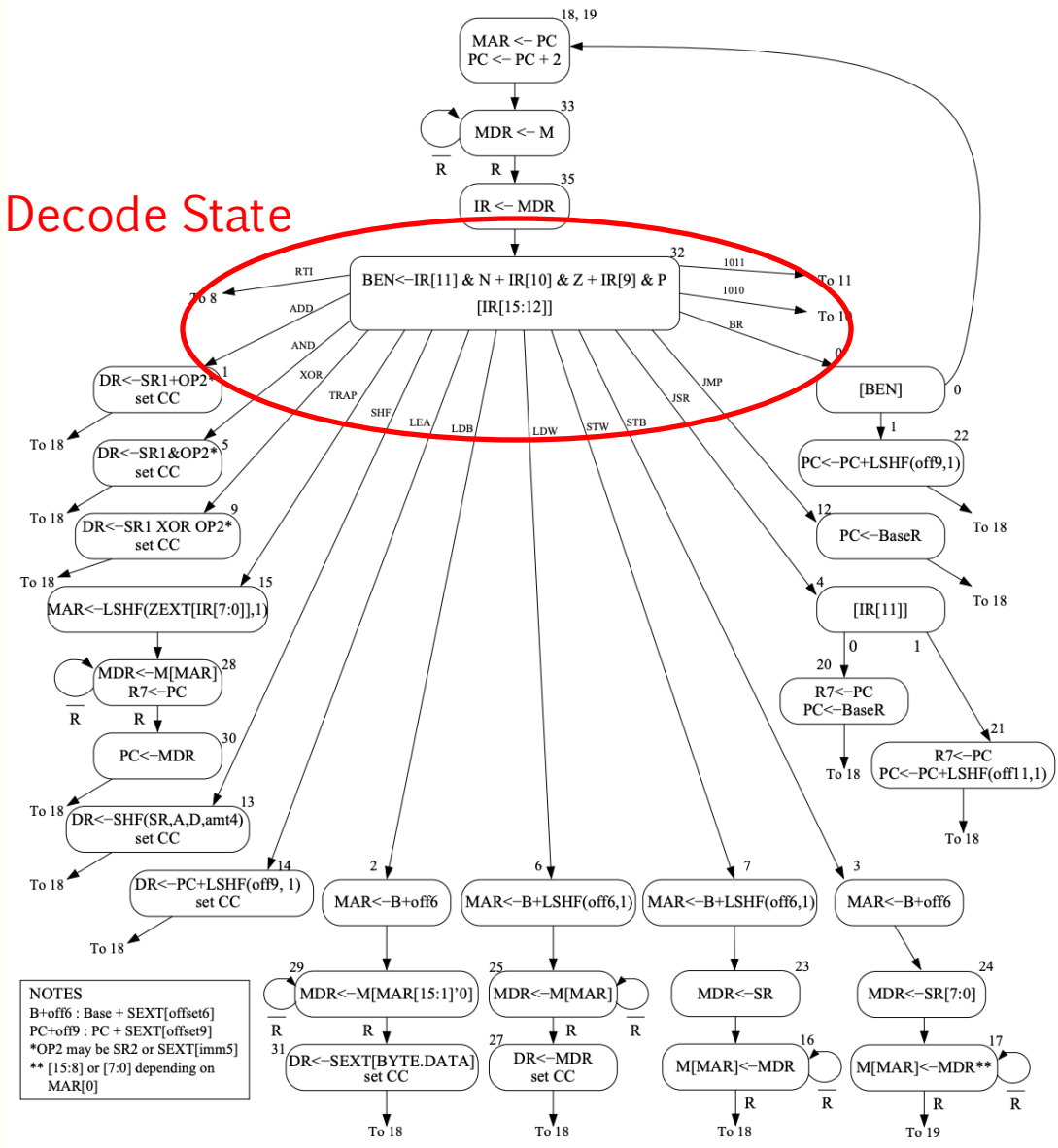


Figure C.2: A state machine for the LC-3b

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 that generate effective address.
- But not necessary in ADD (does not need the memory for the operand fetch)

LDR calculates the address by adding a register and an immediate

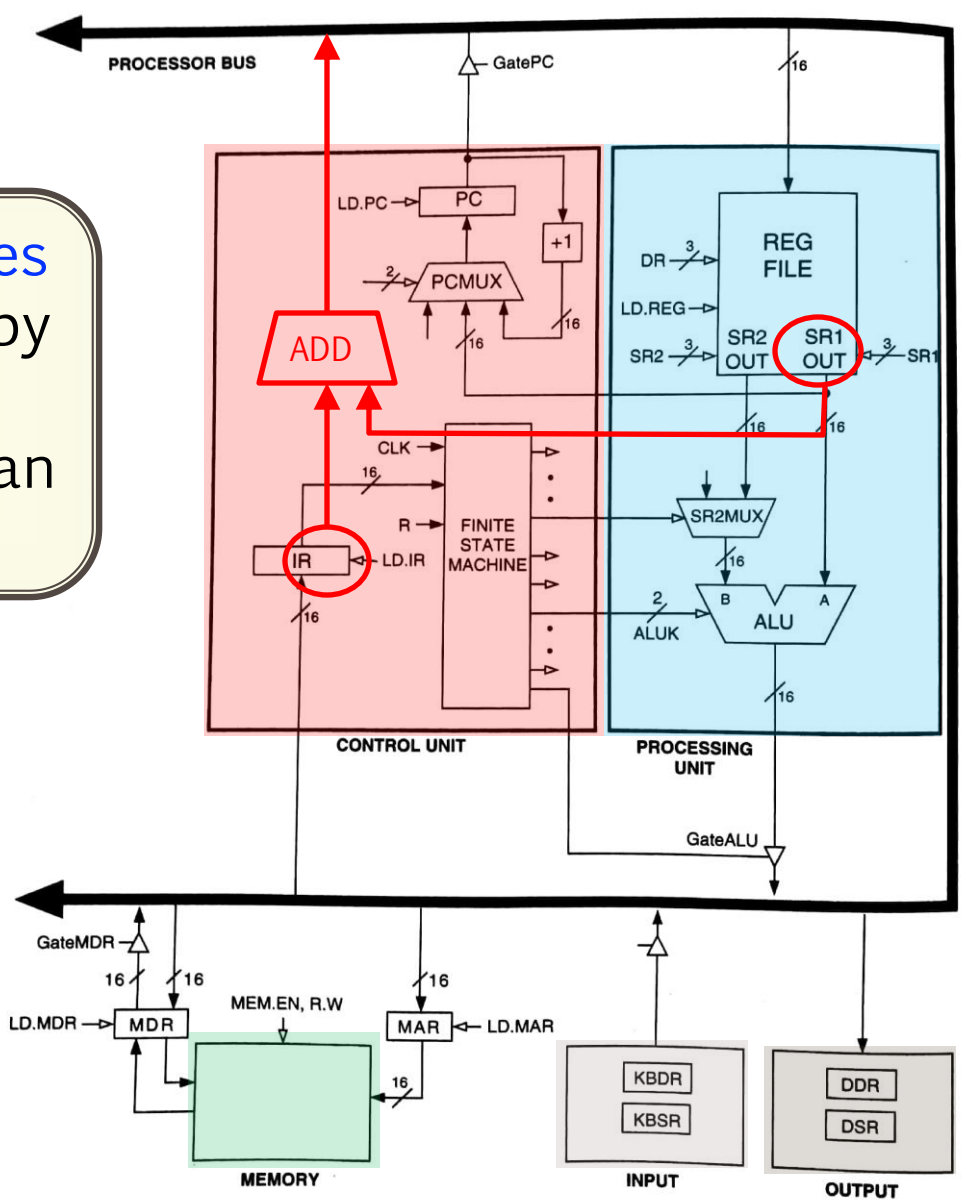


Figure 4.3 The LC-3 as an example of the von Neumann model

Fetch Operands

- The FETCH OPERANDS phase obtains the source operands needed to process the instruction
- Applied to both Operate instruction and Data Movement instruction (but functioning is different).
- In LDR
 - Step 1: Load MAR with the address calculated in EVALUATE ADDRESS
 - Step 2: Read memory, placing source operand (as it load this data in the destination register) in MDR
- In ADD
 - Obtain the source operands from the register file
 - In most current microprocessors, this phase can be done at the same time the instruction is being decoded, but the basic principle is same

LDR loads MAR (step 1), and places the results in MDR (step 2)

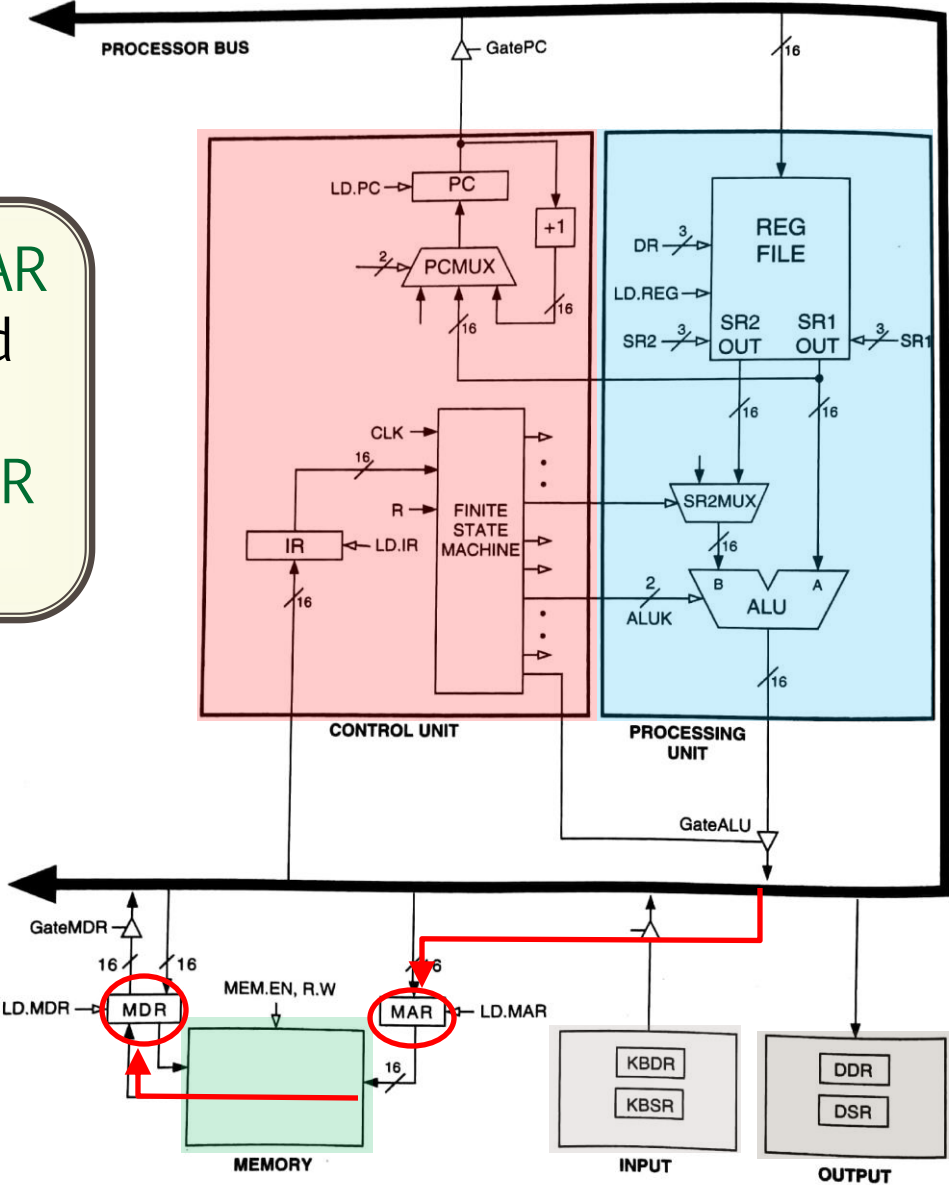


Figure 4.3 The LC-3 as an example of the von Neumann model

Execute

- The EXECUTE phase **executes the instruction based on the type of opcode**
 - In ADD, it performs addition in the ALU
 - In XOR, it performs bitwise XOR in the ALU
- Instruction(s) like LDR, STORE, etc does not have this phase.

ADD adds SR1
and SR2

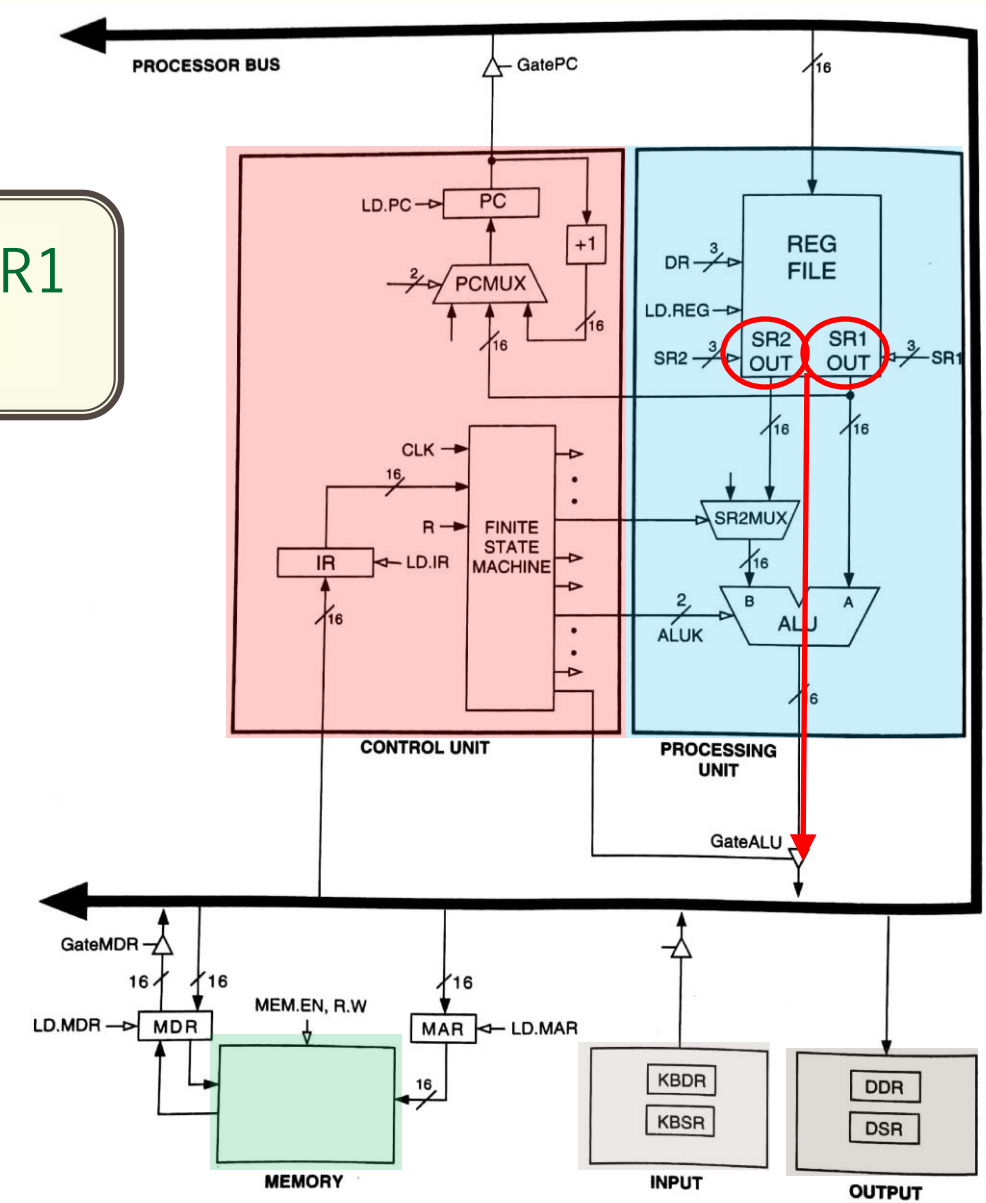


Figure 4.3 The LC-3 as an example of the von Neumann model

Store Result

- The STORE RESULT phase writes to the designated destination (depending on the instruction, the destination may be either register or memory)

- Once STORE RESULT is completed, a new instruction cycle starts (with the FETCH phase)

ADD loads ALU
Result into DR

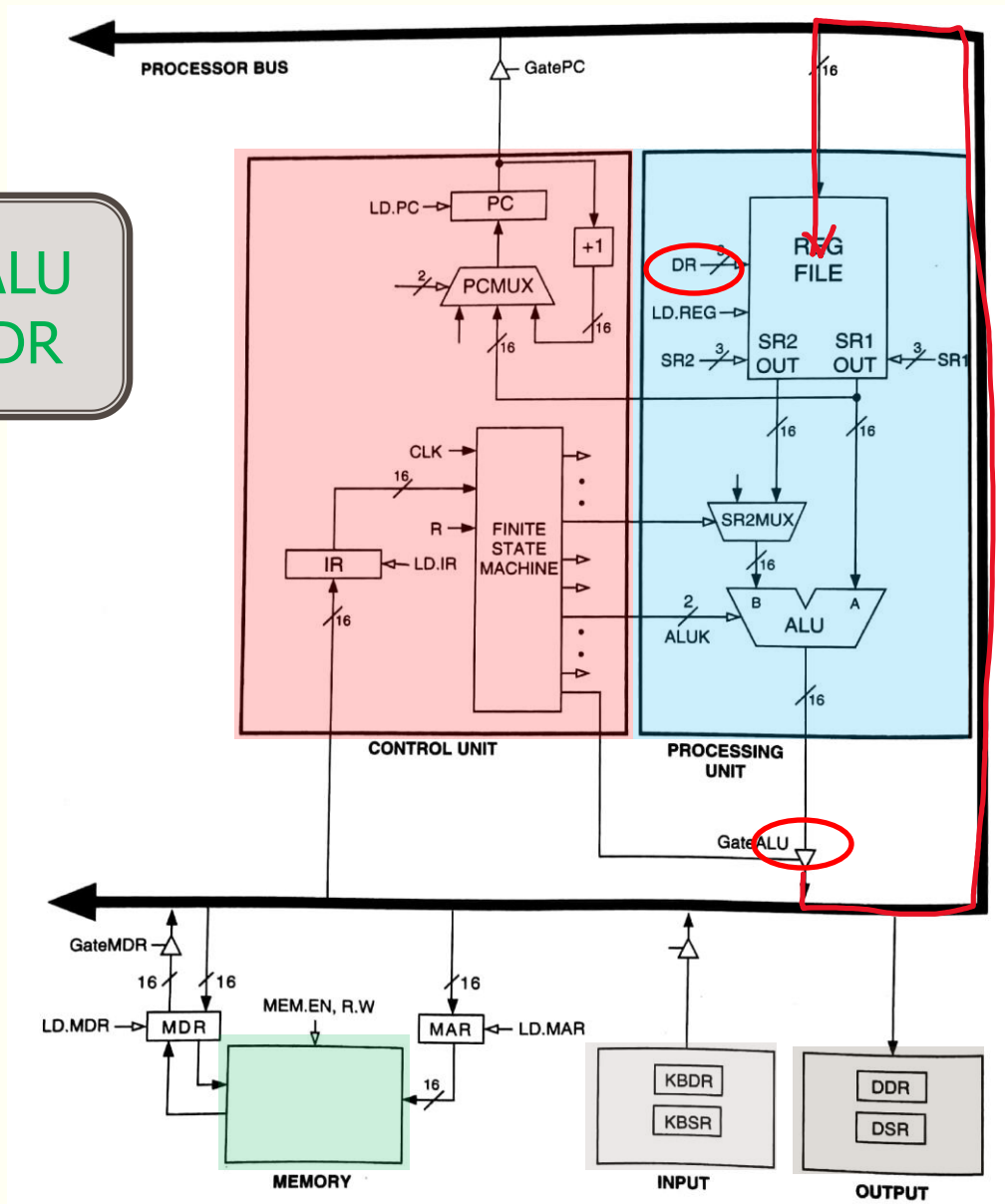


Figure 4.3 The LC-3 as an example of the von Neumann model

LDR loads MDR into DR

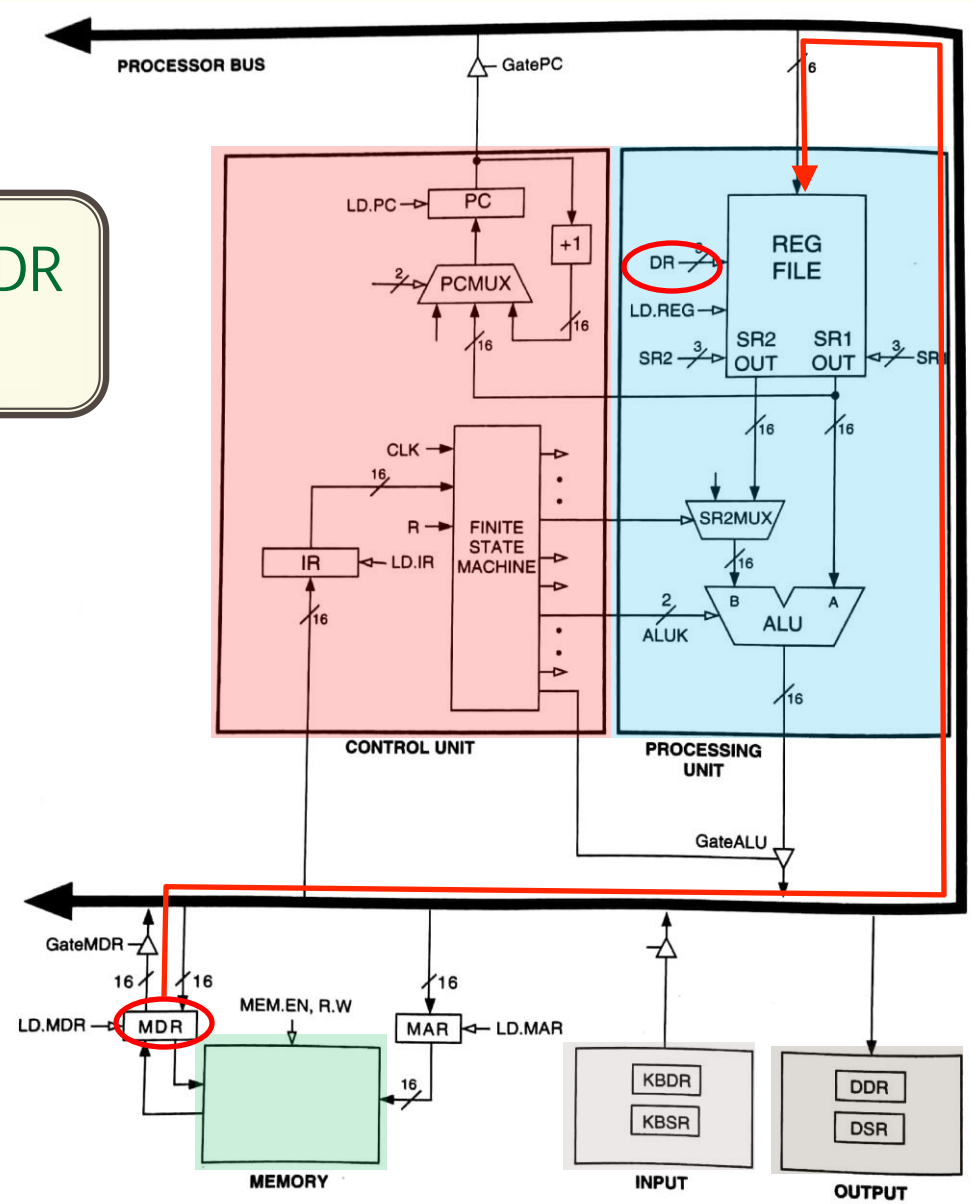


Figure 4.3 The LC-3 as an example of the von Neumann model

The Instruction Cycle

