

Design of Digital Circuits

Lecture 9: Von Neumann Model, ISA, LC-3 and MIPS

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zurich

Spring 2018

22 March 2018

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
- LC3 and MIPS assembly and programming
- Introduction to microprogramming and single-cycle microarchitectures
- Multi-cycle microarchitecture
- Microprogramming

Readings

■ This week

- Von Neumann Model, LC-3, and MIPS
 - P&P, Chapter 4, 5
 - H&H, Chapter 6
 - P&P, Appendices A and C (ISA and microarchitecture of LC-3)
 - H&H, Appendix B (MIPS instructions)
- Digital Building Blocks
 - H&H, Chapter 5

■ Next week

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

What Will We Learn Today?

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

The Von Neumann Model

Basic Elements of a Computer

- In past lectures we learned
 - Combinational circuits
 - Sequential circuits
- With them, we can build
 - Decision elements
 - Storage elements
- Basic elements of a computer
- To get a task done by a computer we need
 - Computer
 - Data
 - Program: A set of instructions
 - Instruction: the smallest piece of work in a computer

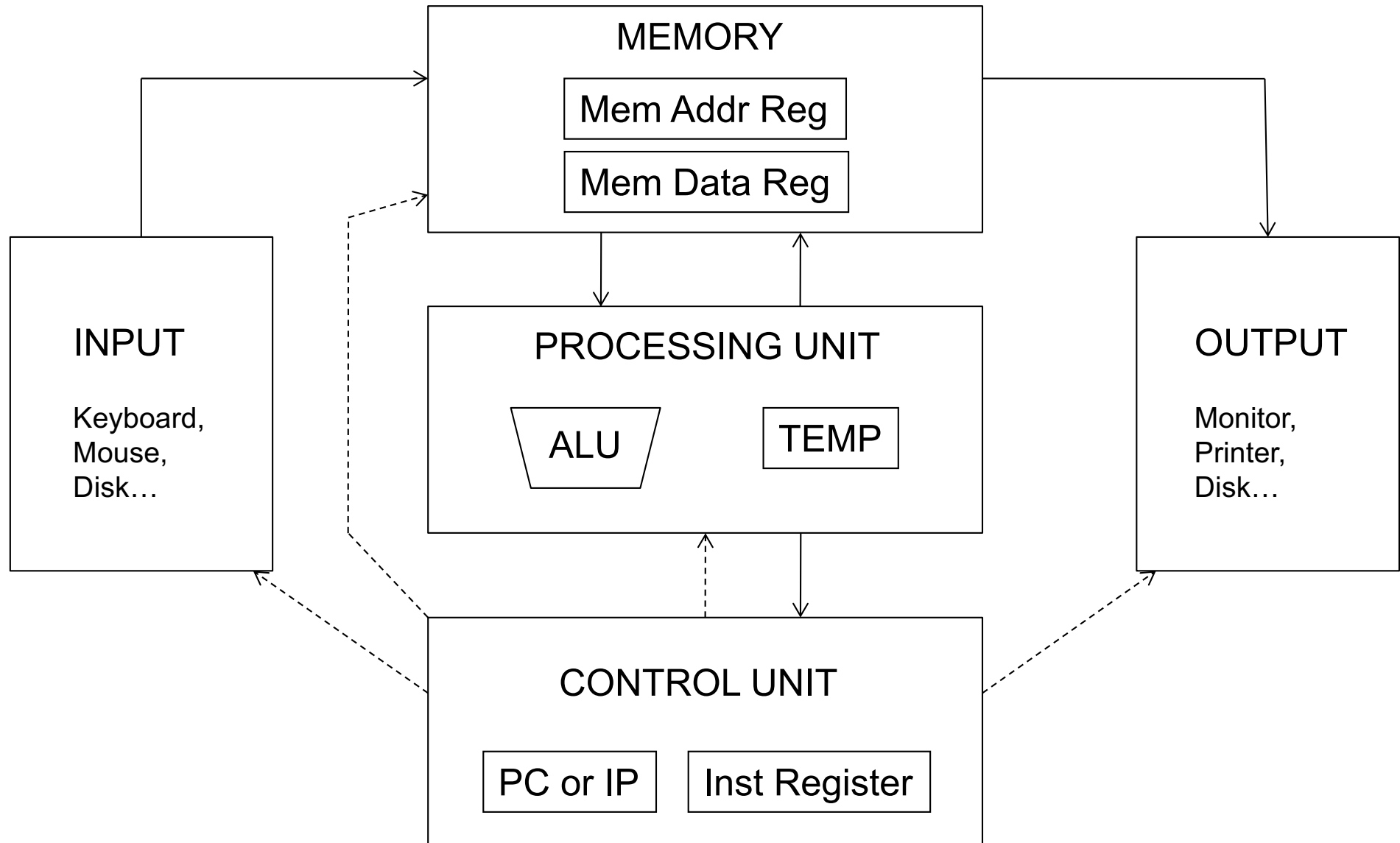
The Von Neumann Model

- Let's start **building the computer**
- In order to build a computer **we need a model**
- John von Neumann proposed **a fundamental model** in 1946
- It consists of 5 parts
 - Memory
 - Processing unit
 - Input
 - Output
 - Control unit
- Throughout this lecture, we consider two examples of the von Neumann model
 - **LC-3**
 - **MIPS**

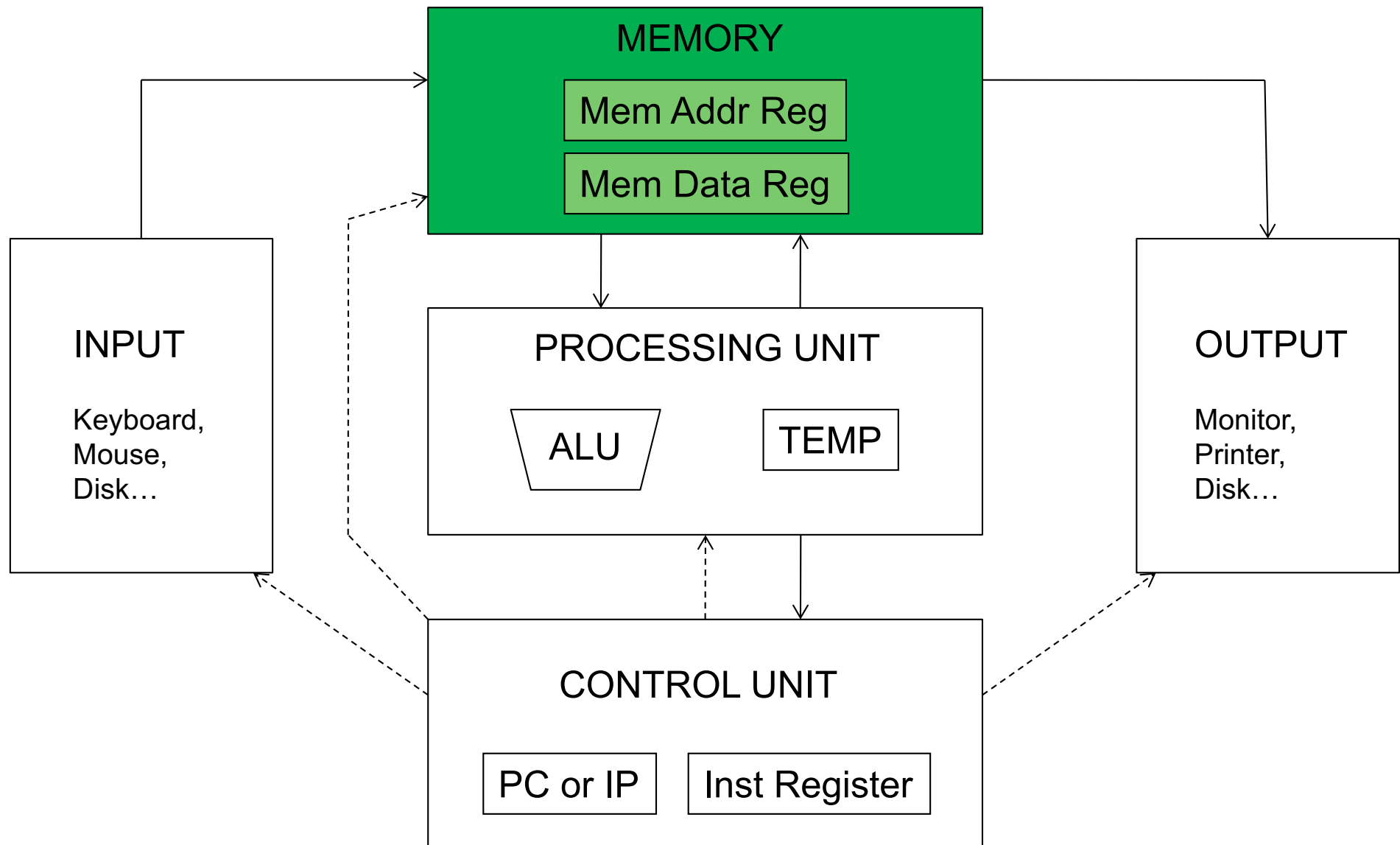


Burks, Goldstein, von Neumann,
“Preliminary discussion of the logical design
of an electronic computing instrument,” 1946.

The Von Neumann Model



The Von Neumann Model



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 determines the addressability
 - E.g., word-addressable
 - E.g., 8-bit addressable (or byte-addressable)
- The total number of addresses is the address space
 - In LC-3, the address space is 2^{16}
 - 16-bit addresses
 - In MIPS, the address space is 2^{32}
 - 32-bit addresses

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

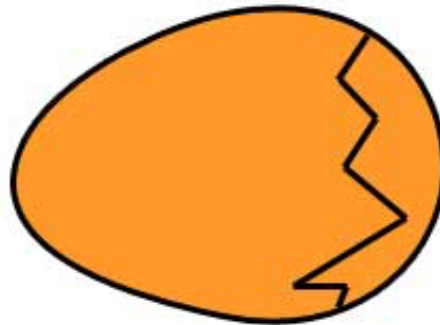
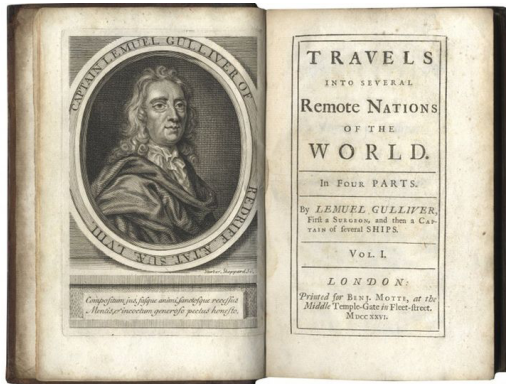
- Each **byte** has a **unique address**
 - Actually, MIPS is **byte-addressable**
 - LC-3b is **byte-addressable**, too

MIPS
memory

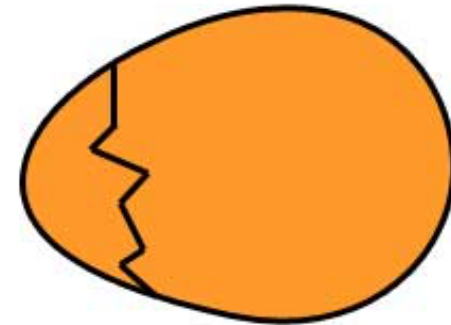
| Word Address | Data | | | | |
|--------------|-------------------------------------|-----|-----|-----|--------|
| . | . | . | . | . | |
| . | . | . | . | . | |
| . | . | . | . | . | |
| 0000000C | D 1 | 6 1 | 7 A | 1 C | Word 3 |
| 00000008 | 1 3 | C 8 | 1 7 | 5 5 | Word 2 |
| 00000004 | F 2 | F 1 | F 0 | F 7 | Word 1 |
| 00000000 | How are these four bytes addressed? | | | | Word 0 |

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

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 | | LSB | |

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 | |

Big Endian vs Little Endian

Big Endian

Little Endian

Byte

Word

Byte

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 data

MSB

LSB

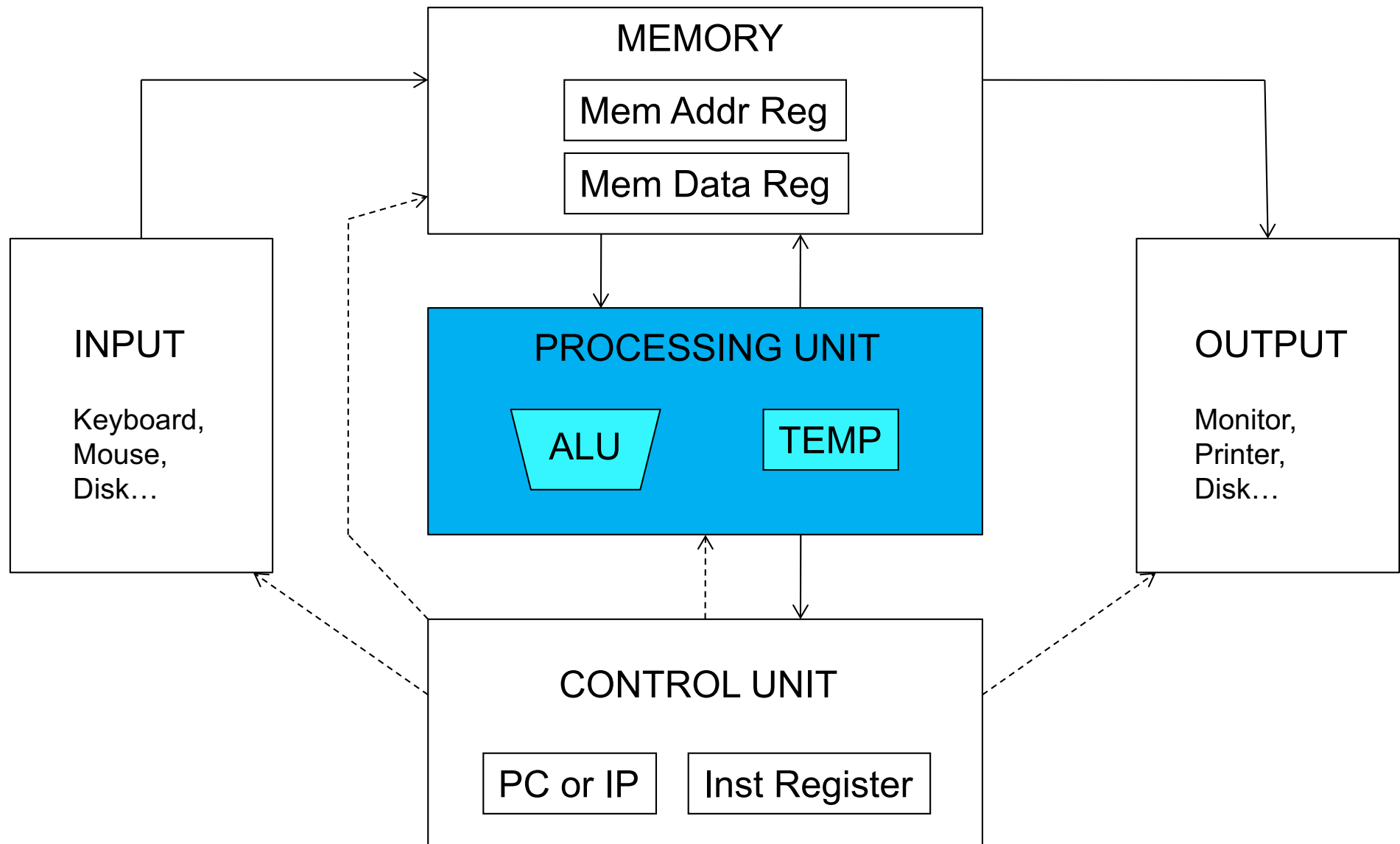
MSB

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

- The processing unit can consist of many **functional units**
- We start with a simple **Arithmetic and Logic Unit (ALU)**
 - **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
 - In MIPS it is 32 bits
- Temporary storage: **Registers**
 - E.g., to calculate $(A+B)*C$, the intermediate result of $A+B$ is stored in a register

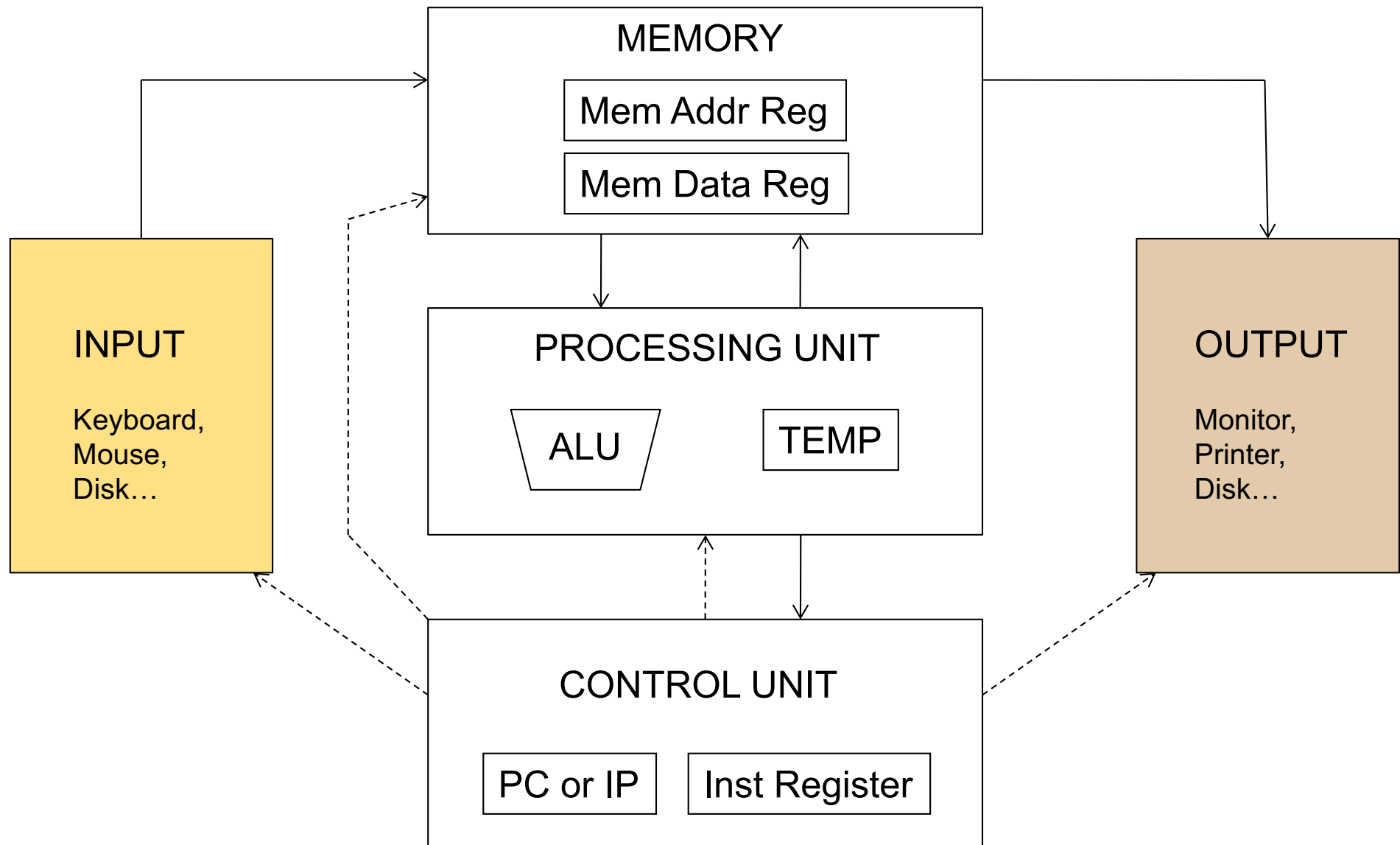
Registers

- Memory is big but slow
- Registers
 - Ensure fast access to operands
 - Typically one register contains one word
- Register set or file
 - 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

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 |

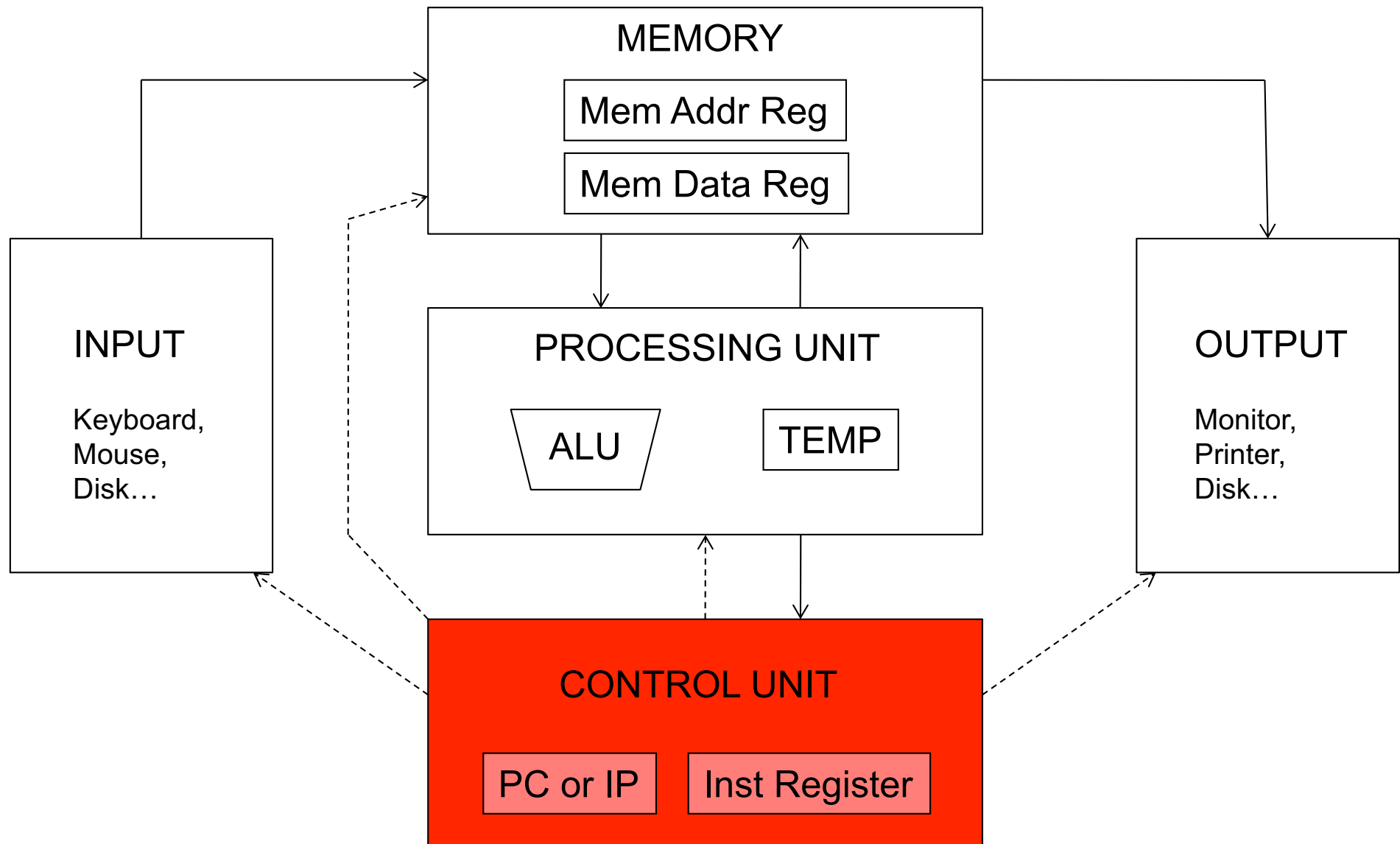
The Von Neumann Model



Input and Output

- 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

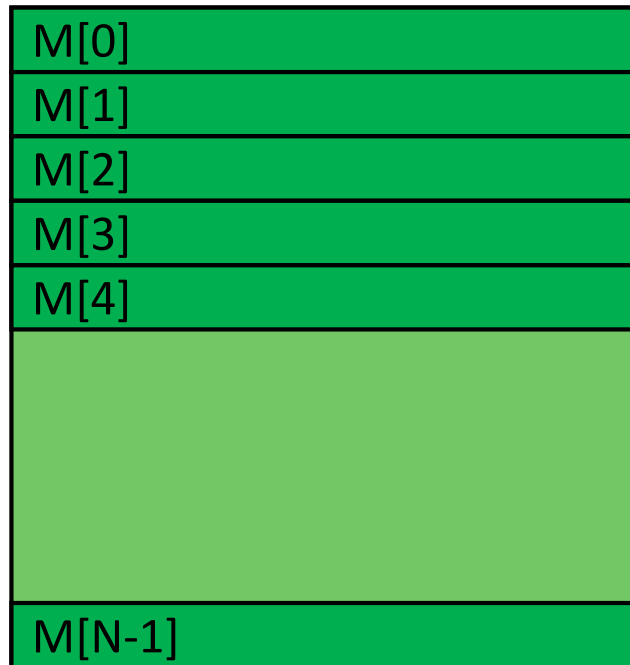
The Von Neumann Model



Control Unit

- The control unit is the conductor of the orchestra
- It conducts the step-by-step process of executing a program
- It keeps track of the instruction being executed with an instruction register (IR), which contains the instruction
- Another register contains the address of the next instruction to execute. It is called program counter (PC) or instruction pointer (IP)

Programmer Visible (Architectural) State



Memory
array of storage locations
indexed by an address



Registers

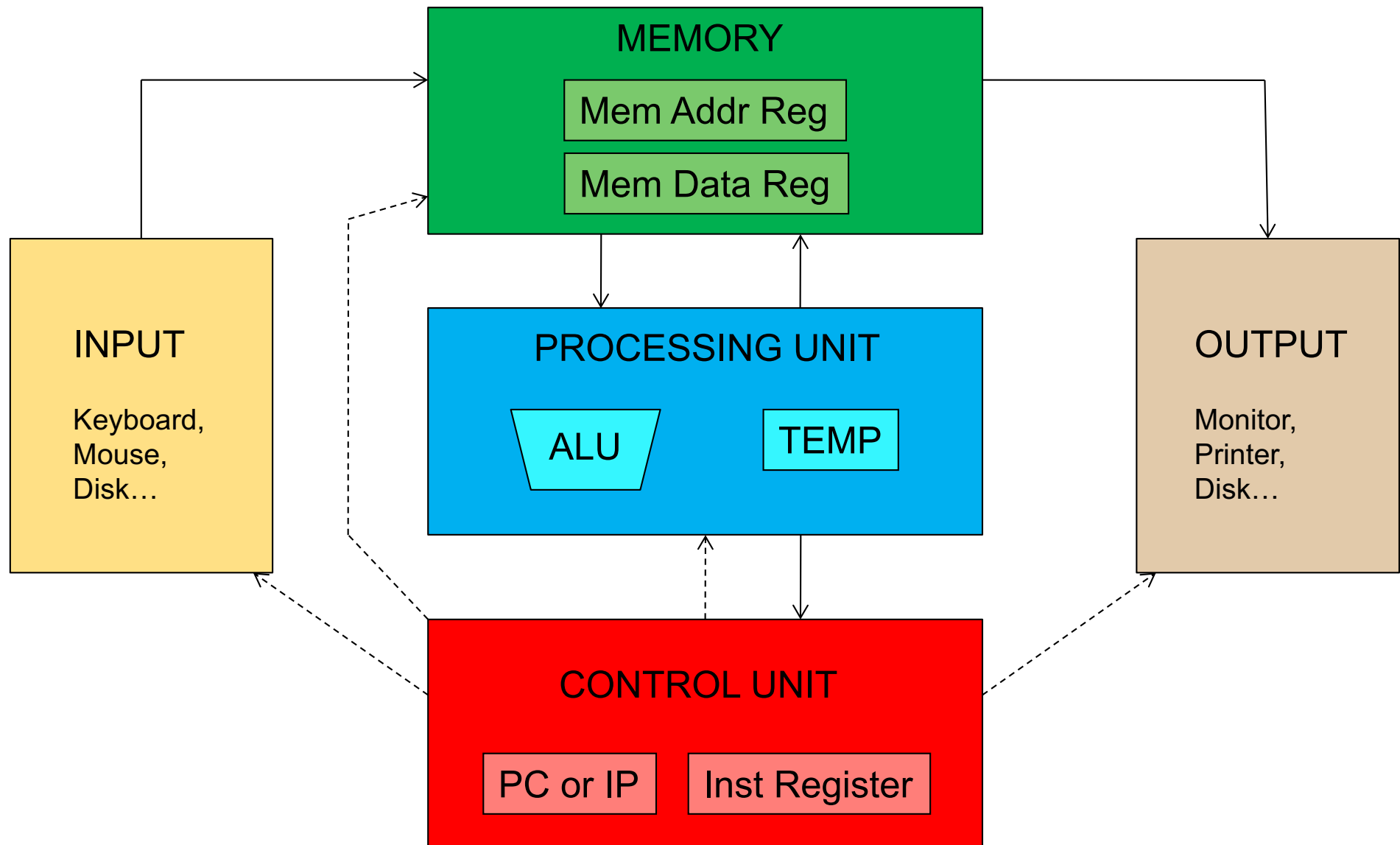
- given special names in the ISA (as opposed to addresses)
- general vs. special purpose

Program Counter

memory address
of the current instruction

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

The Von Neumann Model



LC-3: A Von Neumann Machine

LC-3: A Von Neumann Machine

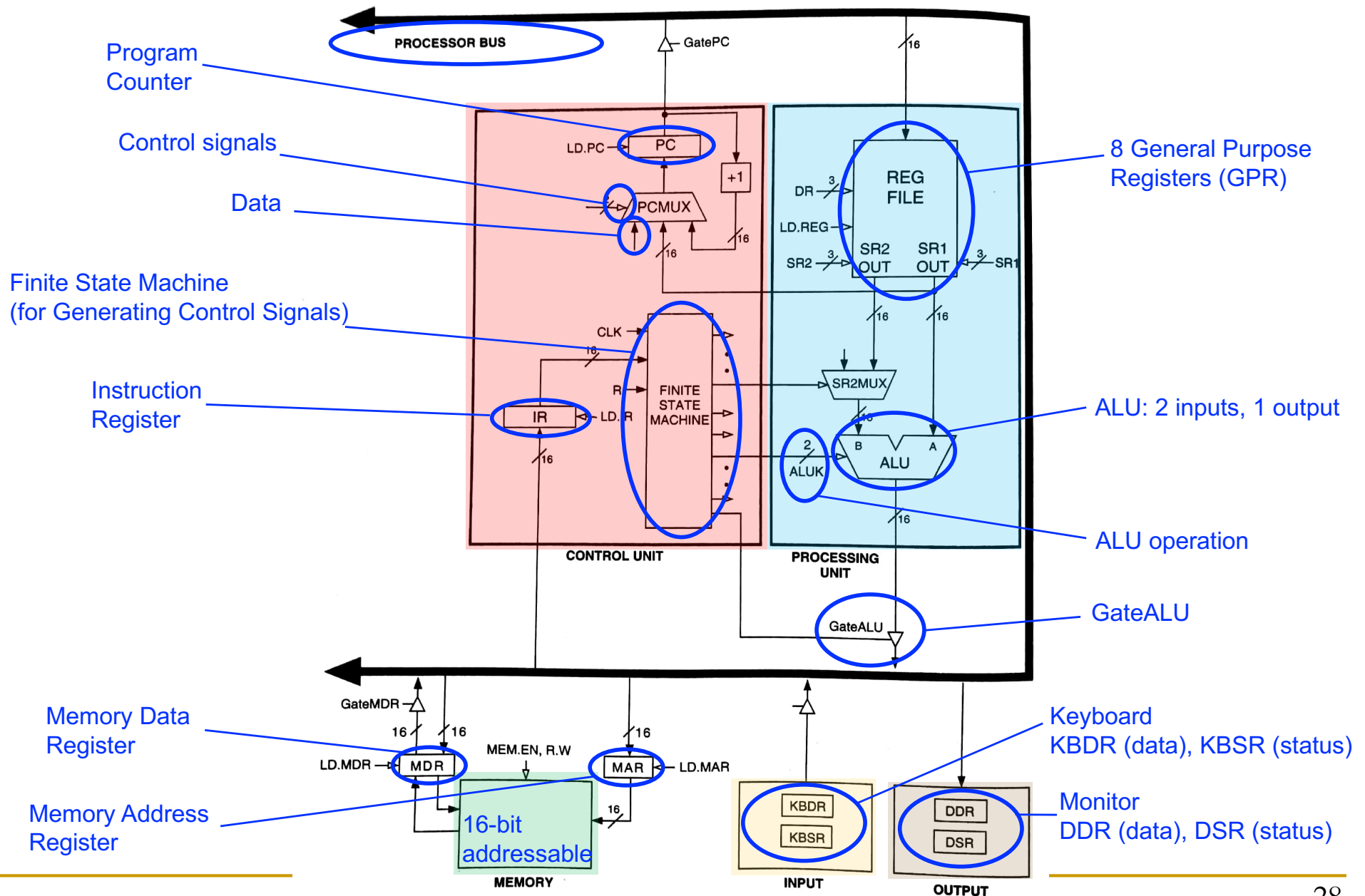


Figure 4.3 The LC-3 as an example of the von Neumann model

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 word length** (4 in MIPS)
 - In MIPS the OS typically sets the PC to **0x00400000**

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 learn **LC-3 instructions** and **MIPS instructions**
- Let us start with some examples of instructions

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

An Example of 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 |

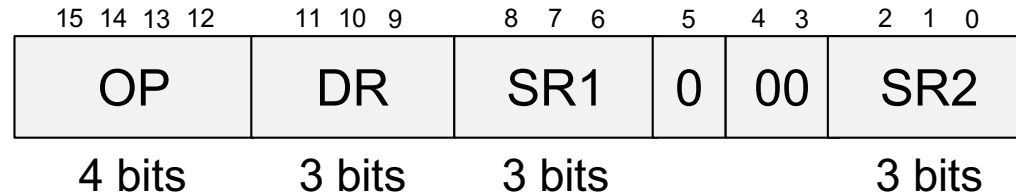
Machine Code

| 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

Instruction Format or Encoding

■ LC-3



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

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

$rd \leftarrow rs + rt$

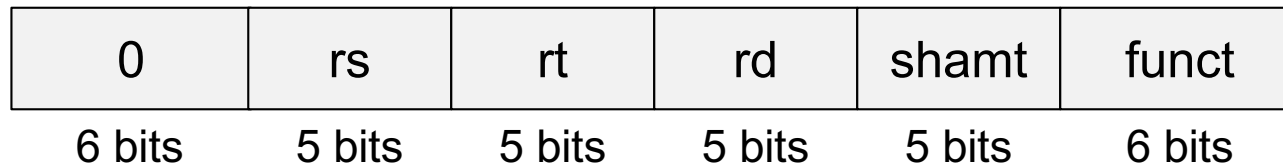
Machine Code

| 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 Formats: R-Type in MIPS

- R-type
 - 3 register operands
- MIPS



- 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 the **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**
- 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
- $a \leftarrow \text{Memory}[A + i]$

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

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, #4
```

Field Values

| OP | | DR | | BaseR | | offset6 | |
|----|----|----|---|-------|---|---------|---|
| 6 | | 3 | | 0 | | 4 | |
| 15 | 12 | 11 | 9 | 8 | 6 | 5 | 0 |

■ MIPS

MIPS assembly

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

Field Values

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

I-Type

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

The Instruction Cycle

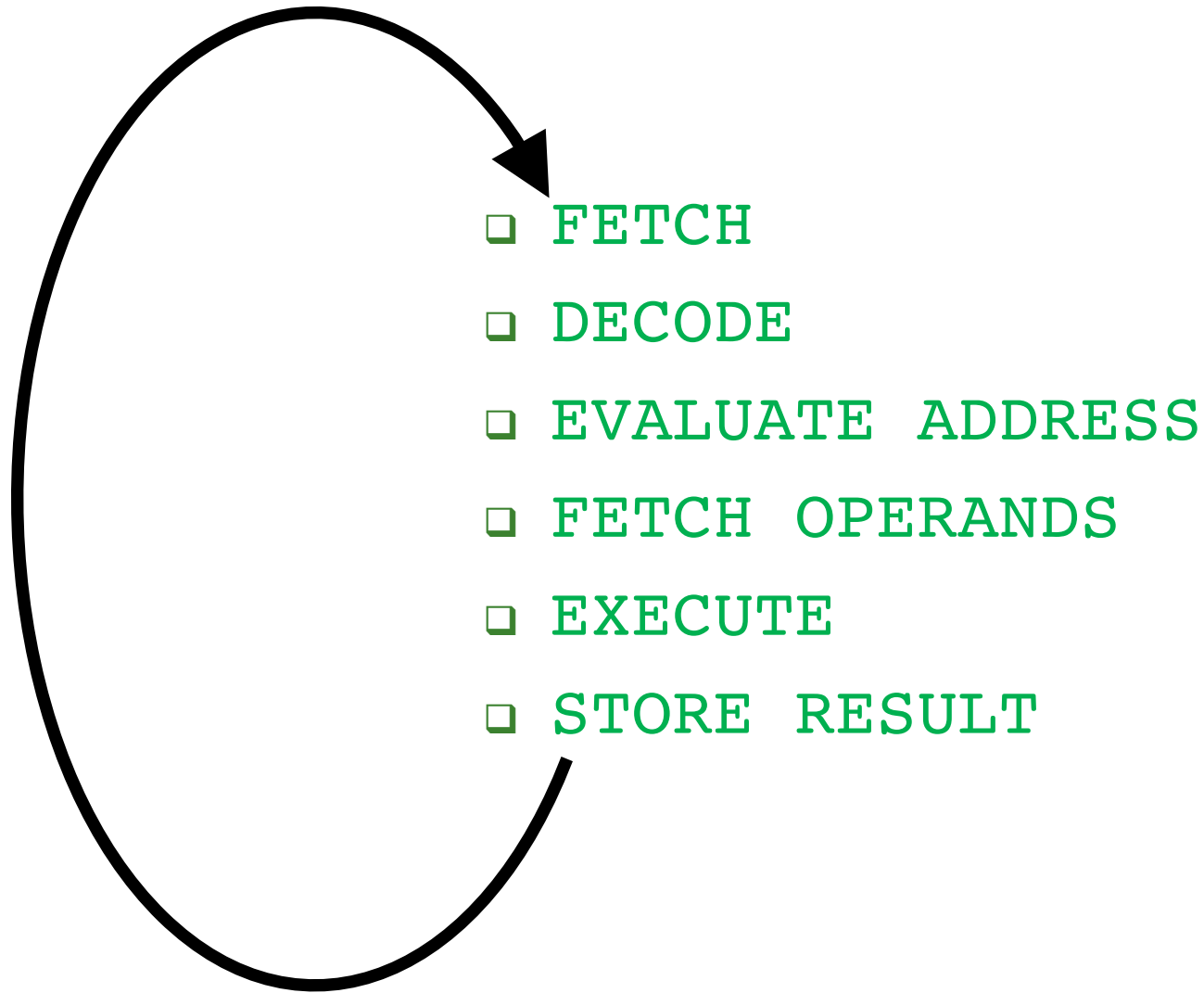
- The instruction cycle is a sequence of steps or **phases**, that an instruction goes through to be executed
 - **FETCH**
 - **DECODE**
 - **EVALUATE ADDRESS**
 - **FETCH OPERANDS**
 - **EXECUTE**
 - **STORE RESULT**

- **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

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**
- **Complete description**
 - Step 1: **Load the MAR with** the contents of the **PC**, and simultaneously **increment the PC**
 - Step 2: Interrogate memory. This results the **instruction to be placed in the MDR**
 - 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

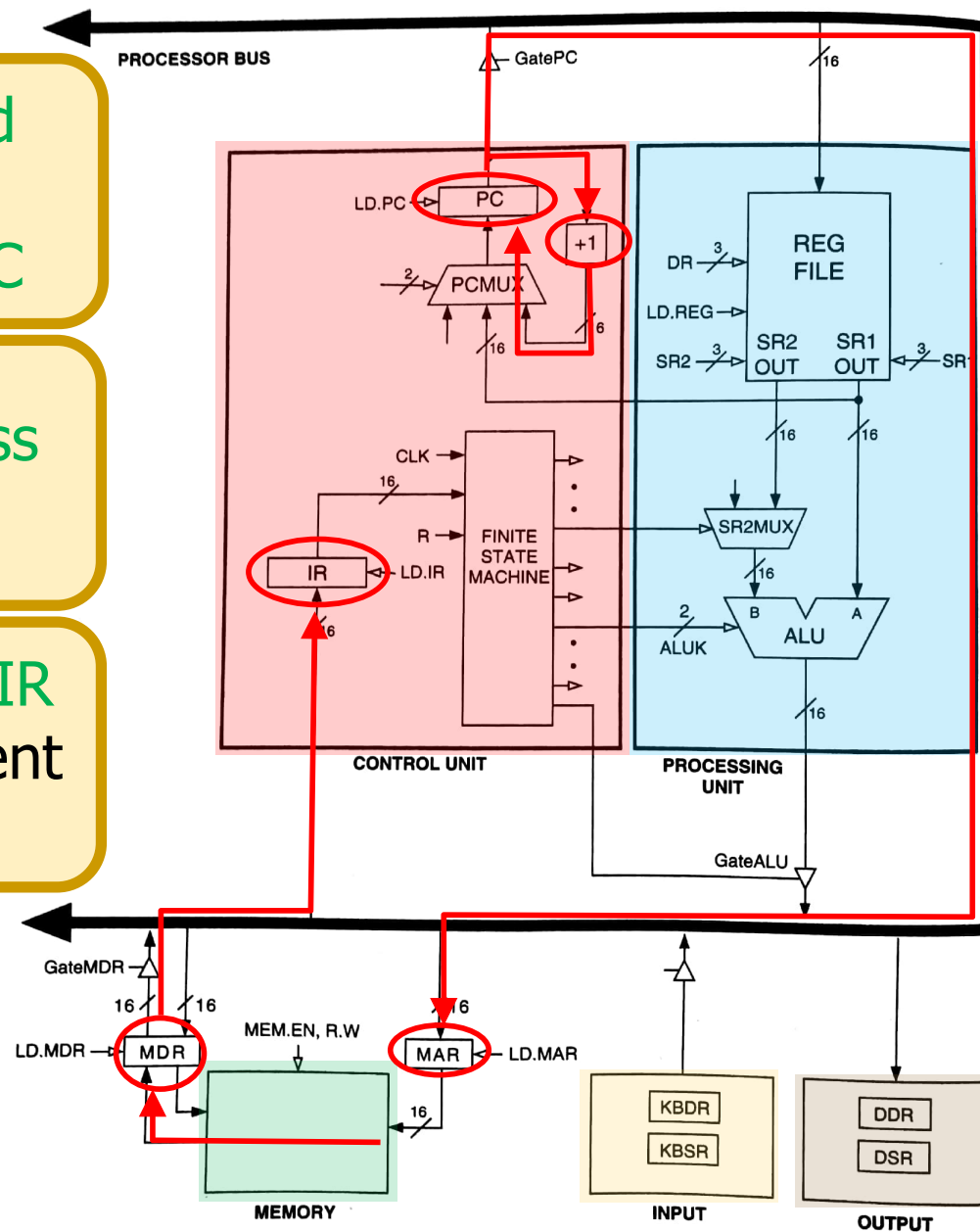


Figure 4.3 The LC-3 as an example of the von Neumann model

DECODE

- The DECODE phase identifies the instruction
- Recall the decoder (Lecture 6, Slides 26-27)
 - 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

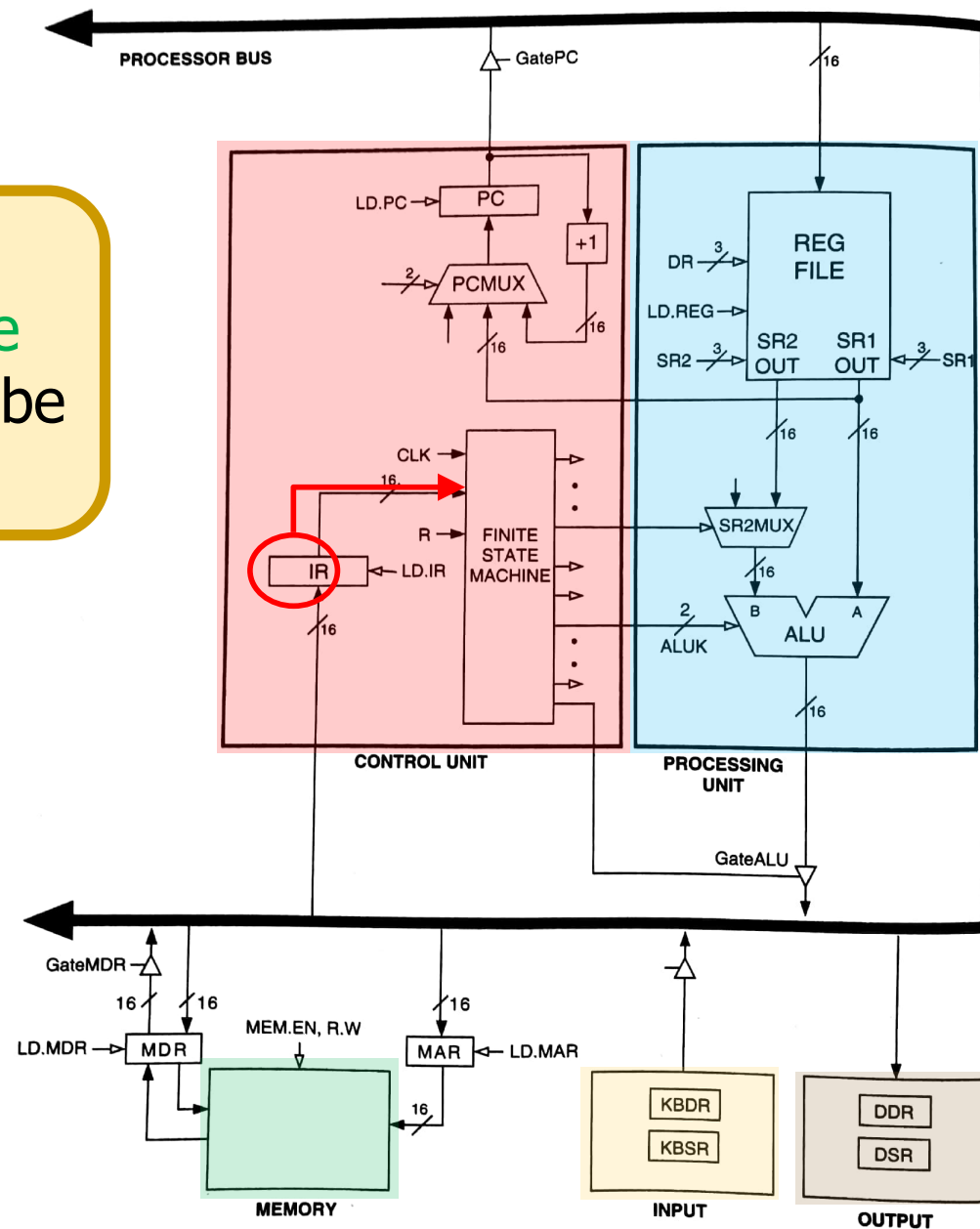


Figure 4.3 The LC-3 as an example of the von Neumann model

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

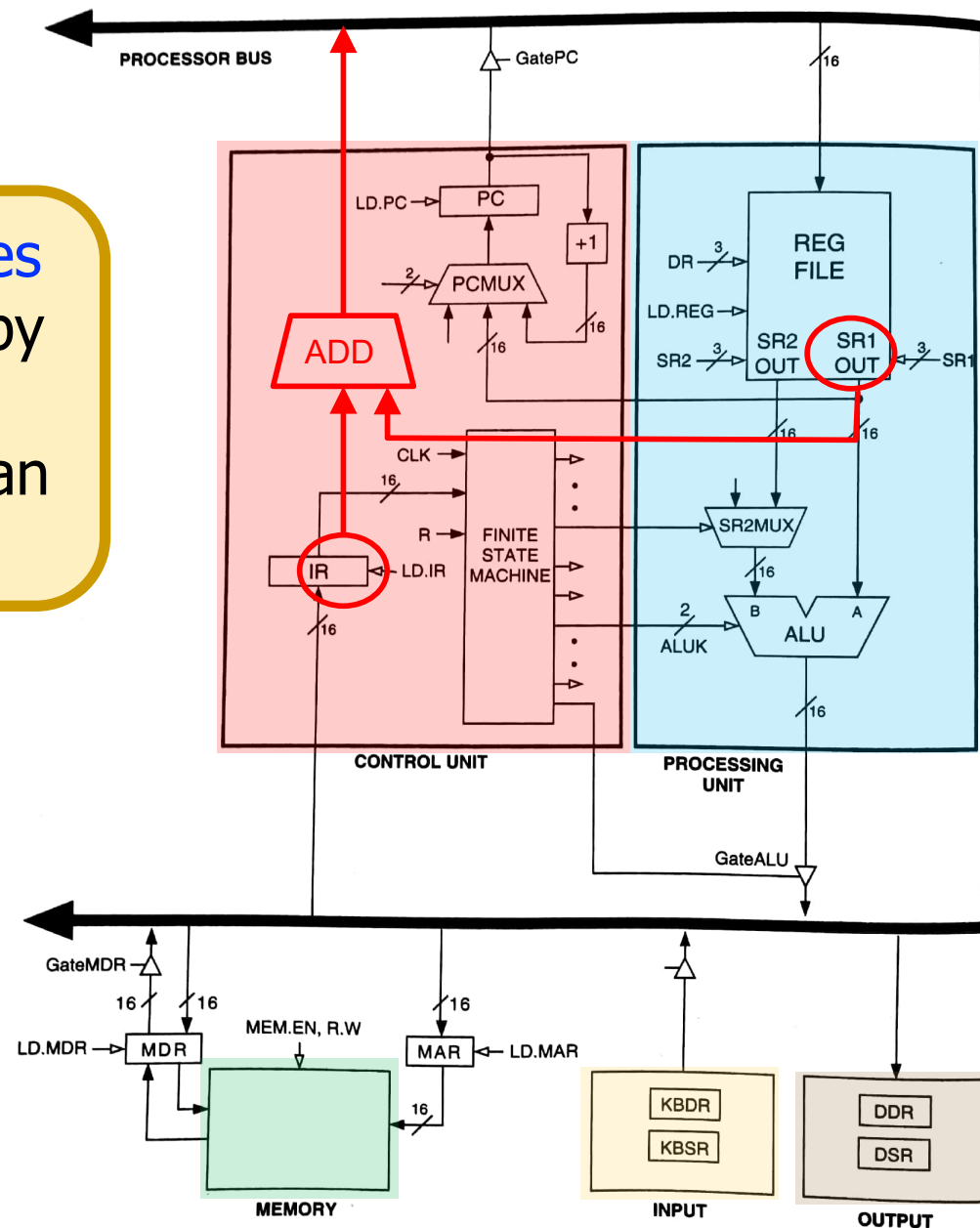


Figure 4.3 The LC-3 as an example of the von Neumann model

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 most current microprocessors, this phase 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)

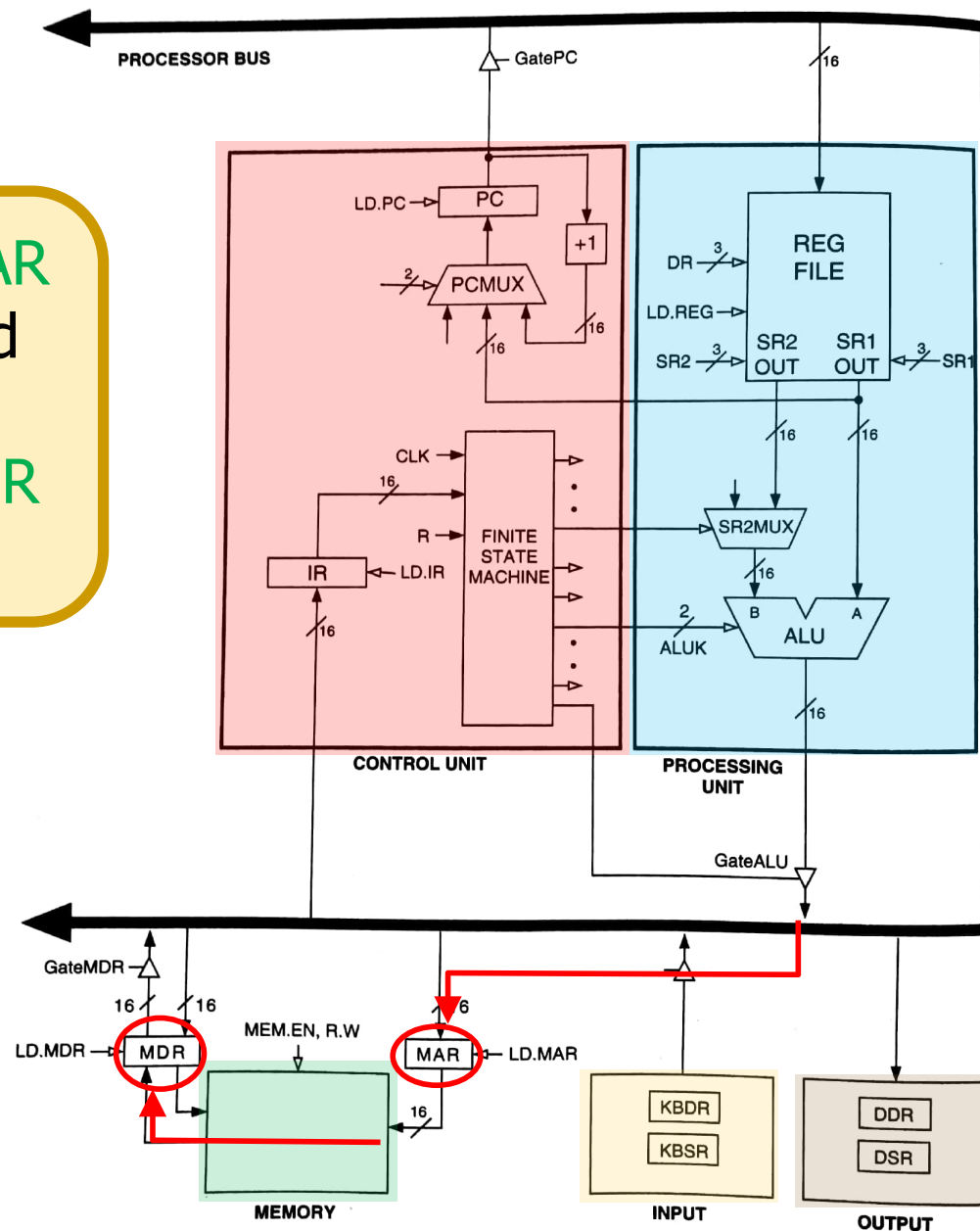


Figure 4.3 The LC-3 as an example of the von Neumann model

EXECUTE

- The EXECUTE phase **executes the instruction**
 - In ADD, it performs addition in the ALU

EXECUTE in LC-3

ADD adds SR1
and SR2

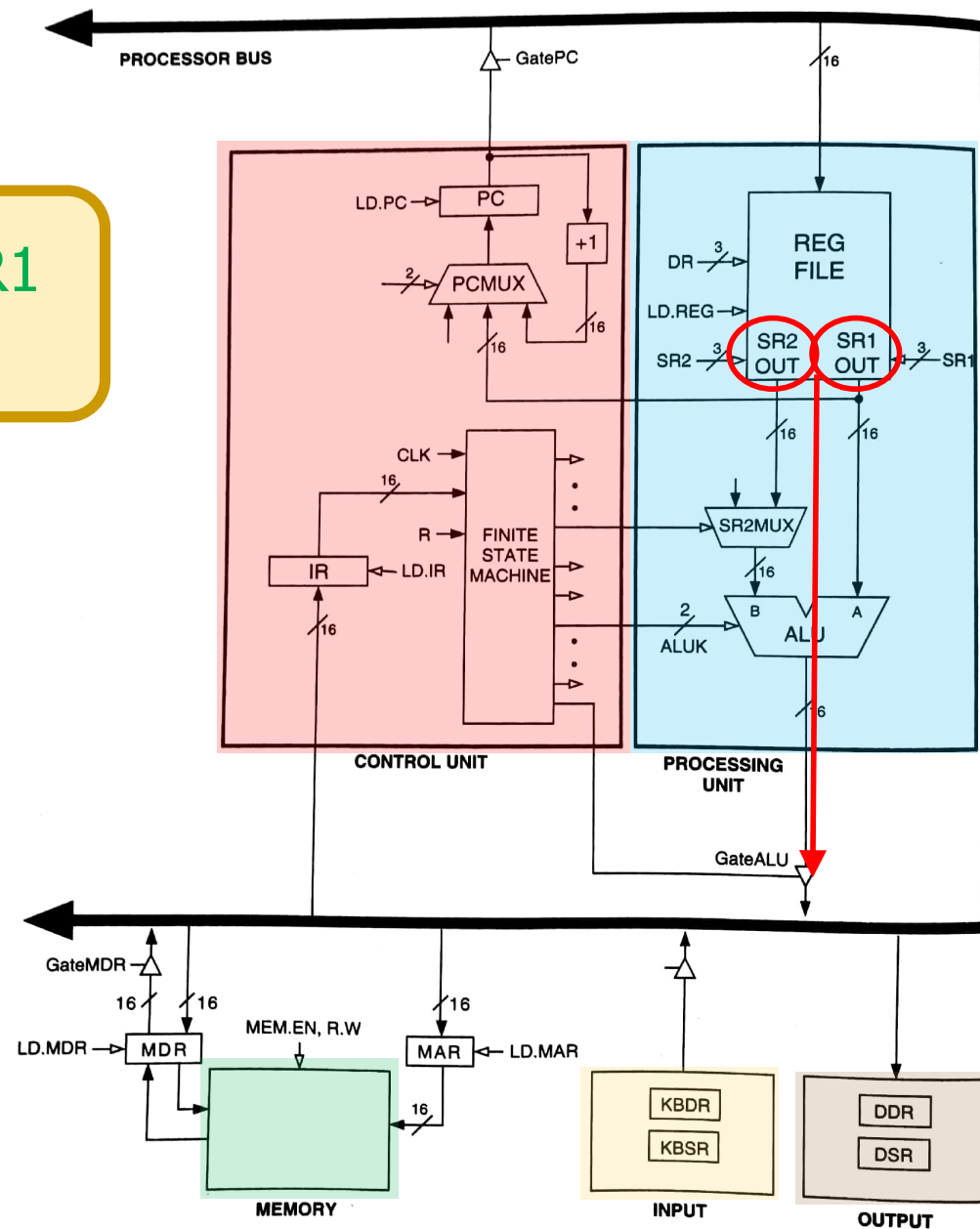


Figure 4.3 The LC-3 as an example of the von Neumann model

STORE RESULT

- The STORE RESULT phase writes to the designated destination
- Once STORE RESULT is completed, a new instruction cycle starts (with the FETCH phase)

STORE RESULT in LC-3

LDR loads MDR
into DR

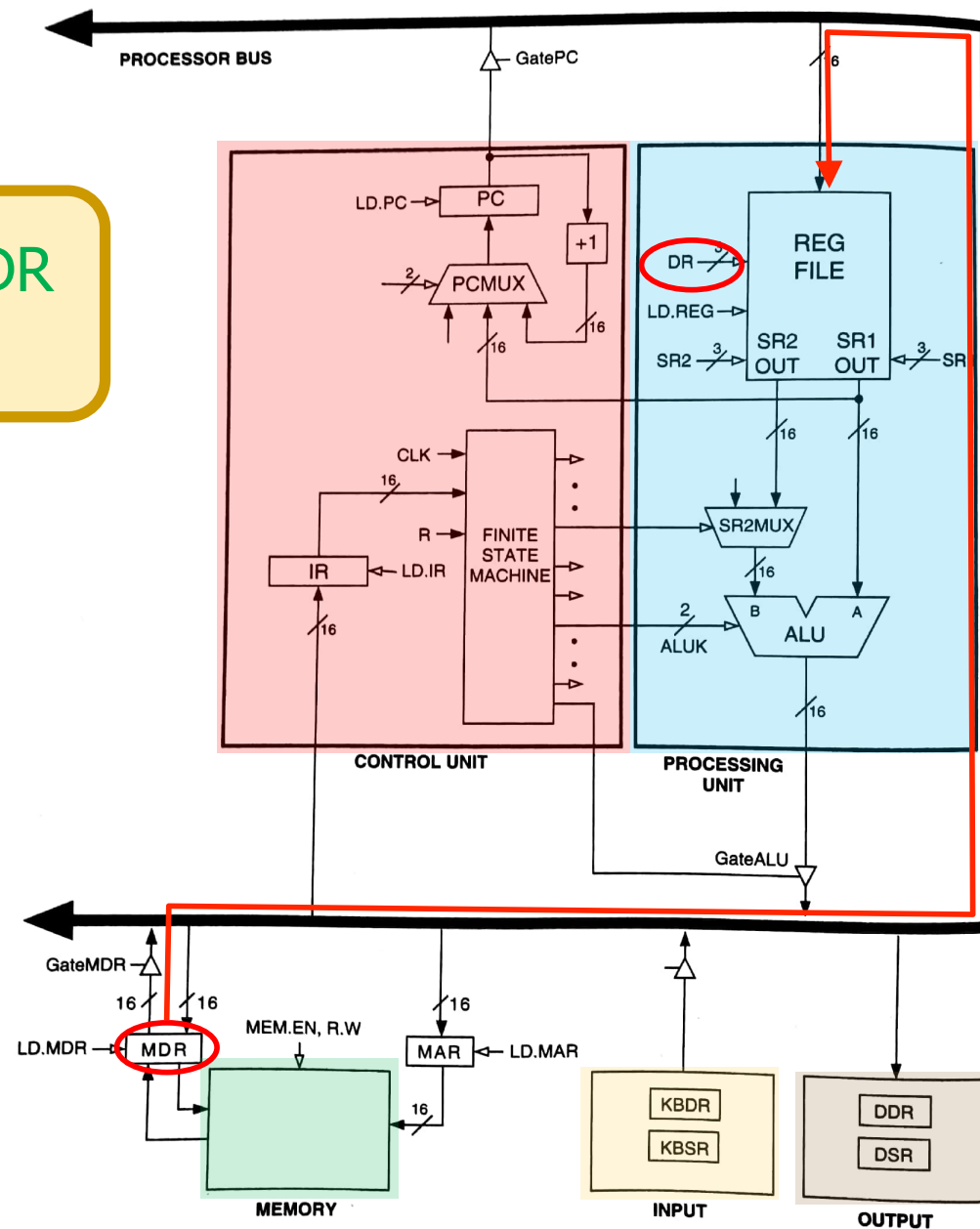
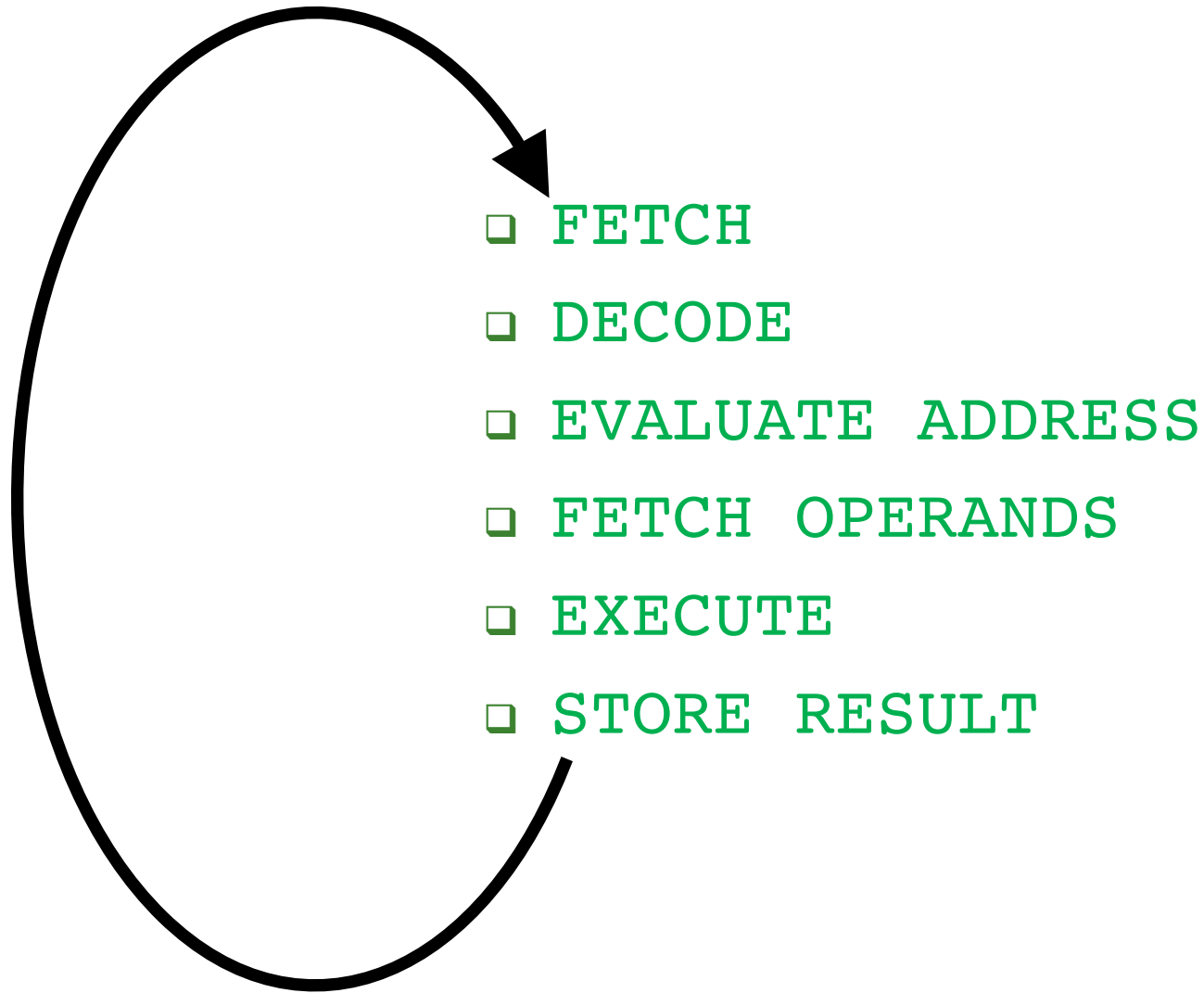


Figure 4.3 The LC-3 as an example of the von Neumann model

The Instruction Cycle



Changing the Sequence of Execution

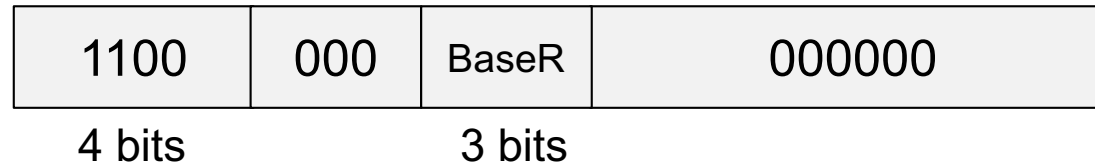
- A computer program **executes in sequence** (i.e., in program order)
 - First instruction, second instruction, third instruction and so on
- Unless we **change the sequence of execution**
- **Control instructions** allow a program to execute **out of sequence**
 - They can change the PC by loading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

This is register addressing mode

- Variations

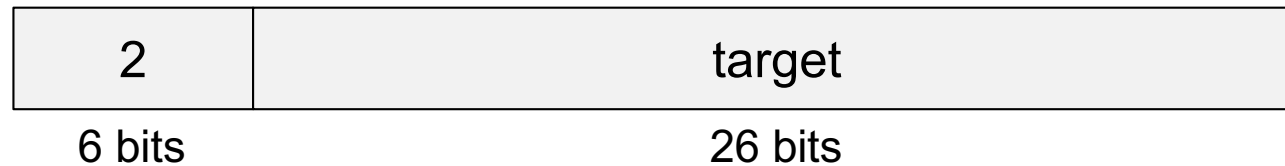
- RET: special case of JMP where BaseR = R7
- JSR, JSRR: jump to subroutine

Jump in MIPS

- Unconditional branch or jump

- MIPS

j target



J-Type

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

jr \$s0

j uses pseudo-direct addressing mode

jr uses register addressing mode

[†] This is the incremented PC

LC-3 Data Path

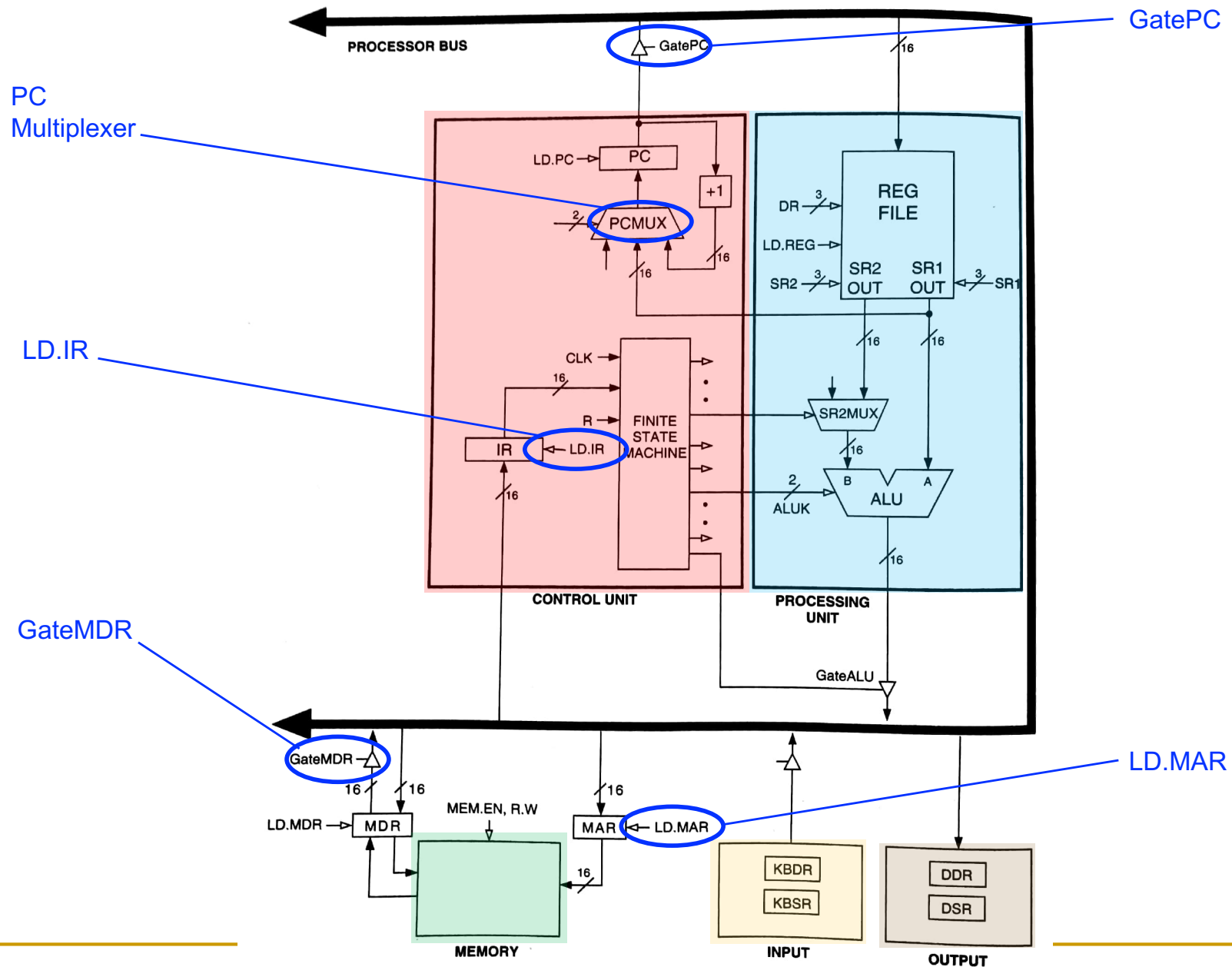
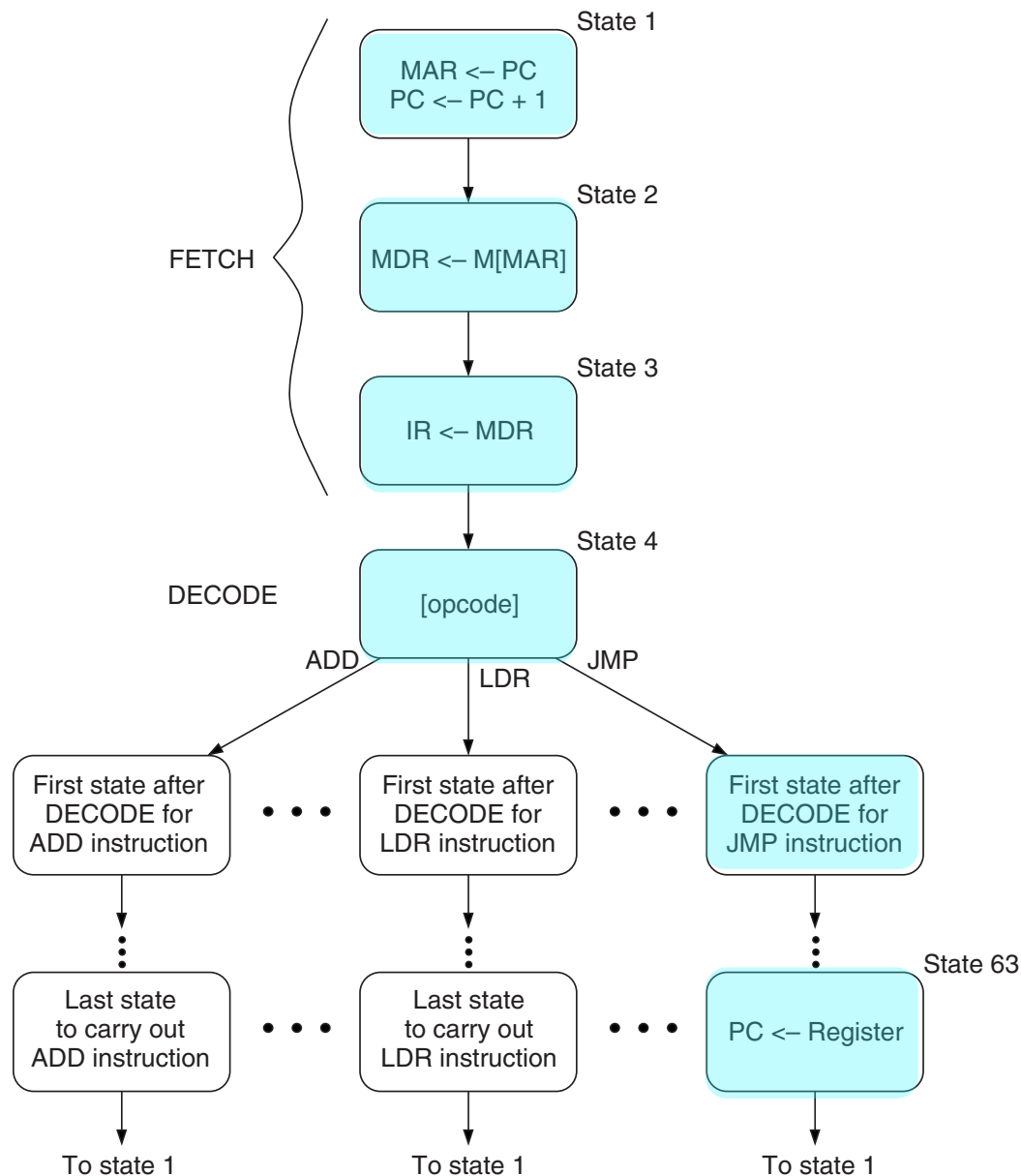


Figure 4.3 The LC-3 as an example of the von Neumann model

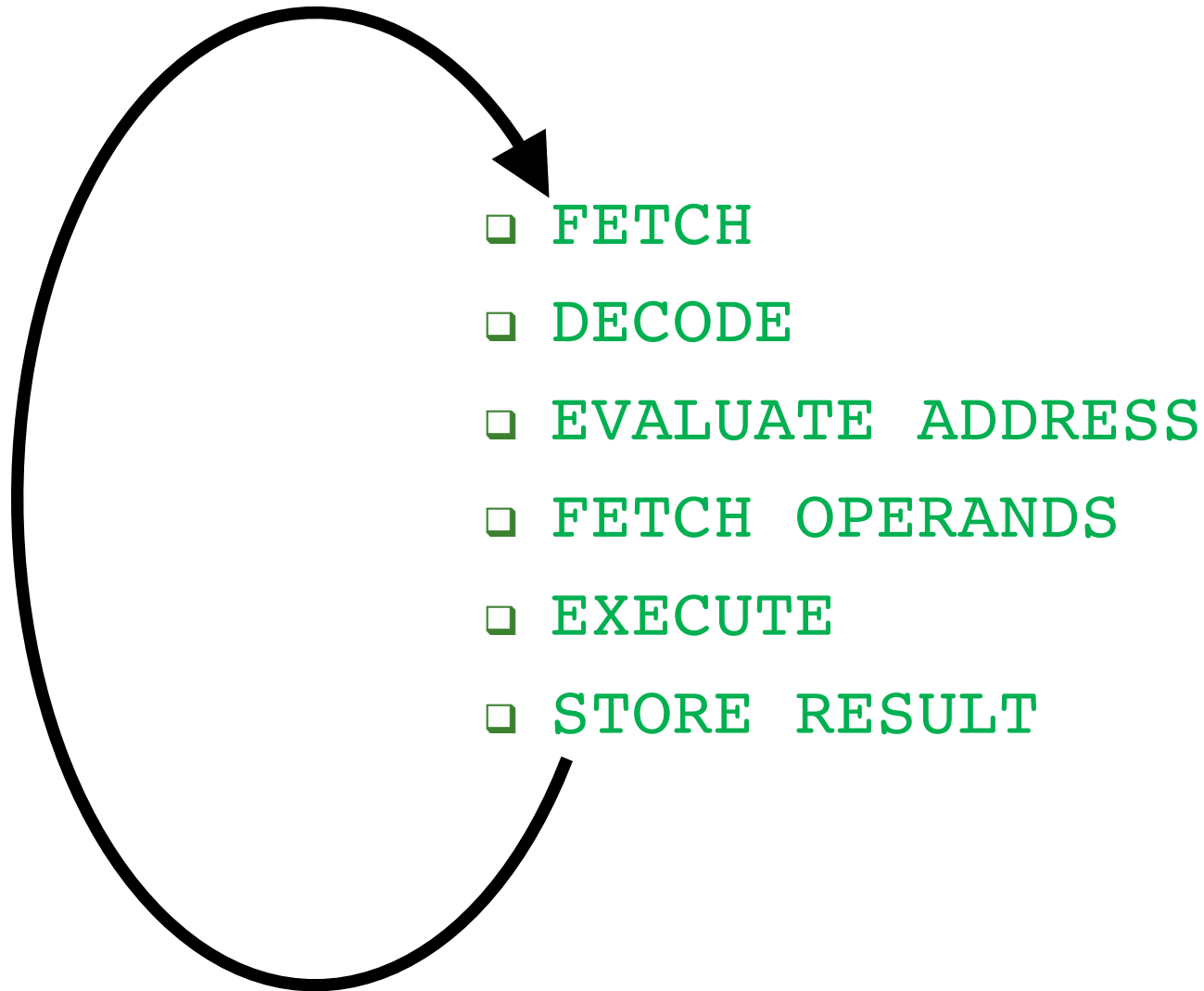
Control of the Instruction Cycle



- 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&Pattell, Appendix C

Figure 4.4 An abbreviated state diagram of the LC-3

The Instruction Cycle



LC-3 and MIPS

Instruction Set Architectures

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: 32 bits)
 - Word- or Byte-addressable
 - The **register set**
 - R0 to R7 in LC-3
 - 32 registers in MIPS
 - The **instruction set**
 - Opcodes
 - Data types
 - Addressing modes

| |
|-------------------|
| Problem |
| Algorithm |
| Program |
| ISA |
| Microarchitecture |
| Circuits |
| 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

Opcodes

- Large or small **sets of opcodes** could be defined
 - E.g, HP Precision Architecture: an instruction for $A*B+C$
 - E.g, x86: multimedia extensions
 - E.g, VAX: opcode to save all information of one program prior to switching to another program
- **Tradeoffs** are involved
 - Hardware complexity vs. software complexity
- In LC-3 and in MIPS there are three **types of opcodes**
 - Operate
 - Data movement
 - Control

Opcodes in LC-3

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------------------|------|----|----|----|--------------|------------|---|-----------|---|---|---------|------|---|-----|---|---|
| ADD ⁺ | 0001 | | | | DR | | | SR1 | | | 0 | 00 | | SR2 | | |
| ADD ⁺ | 0001 | | | | DR | | | SR1 | | | 1 | imm5 | | | | |
| AND ⁺ | 0101 | | | | DR | | | SR1 | | | 0 | 00 | | SR2 | | |
| AND ⁺ | 0101 | | | | DR | | | SR1 | | | 1 | imm5 | | | | |
| BR | 0000 | | | | n | z | p | PCoffset9 | | | | | | | | |
| JMP | 1100 | | | | 000 | | | BaseR | | | 000000 | | | | | |
| JSR | 0100 | | | | 1 | PCoffset11 | | | | | | | | | | |
| JSRR | 0100 | | | | 0 | 00 | | BaseR | | | 000000 | | | | | |
| LD ⁺ | 0010 | | | | DR | | | PCoffset9 | | | | | | | | |
| LDI ⁺ | 1010 | | | | DR | | | PCoffset9 | | | | | | | | |
| LDR ⁺ | 0110 | | | | DR | | | BaseR | | | offset6 | | | | | |
| LEA ⁺ | 1110 | | | | DR | | | PCoffset9 | | | | | | | | |
| NOT ⁺ | 1001 | | | | DR | | | SR | | | 111111 | | | | | |
| RET | 1100 | | | | 000 | | | 111 | | | 000000 | | | | | |
| RTI | 1000 | | | | 000000000000 | | | | | | | | | | | |
| ST | 0011 | | | | SR | | | PCoffset9 | | | | | | | | |
| STI | 1011 | | | | SR | | | PCoffset9 | | | | | | | | |
| STR | 0111 | | | | SR | | | BaseR | | | offset6 | | | | | |
| TRAP | 1111 | | | | 0000 | | | trapvect8 | | | | | | | | |
| reserved | 1101 | | | | | | | | | | | | | | | |

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

Opcodes in LC-3b

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

Funct in MIPS R-Type Instructions (I)

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

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

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

(continued)

Funct in MIPS R-Type Instructions (II)

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

| Funct | Name | Description | Operation |
|-------------|-----------------|------------------------|-------------------------------------|
| 100000 (32) | add rd, rs, rt | add | $[rd] = [rs] + [rt]$ |
| 100001 (33) | addu rd, rs, rt | add unsigned | $[rd] = [rs] + [rt]$ |
| 100010 (34) | sub rd, rs, rt | subtract | $[rd] = [rs] - [rt]$ |
| 100011 (35) | subu rd, rs, rt | subtract unsigned | $[rd] = [rs] - [rt]$ |
| 100100 (36) | and rd, rs, rt | and | $[rd] = [rs] \& [rt]$ |
| 100101 (37) | or rd, rs, rt | or | $[rd] = [rs] \mid [rt]$ |
| 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 | $[rs] < [rt] ? [rd] = 1 : [rd] = 0$ |

- Find the complete list of instructions in the appendix

Data Types

- An ISA supports one or several data types
- LC-3 only supports 2's complement integers
- MIPS supports
 - 2's complement integers
 - Unsigned integers
 - Floating point
- Again, tradeoffs are involved

Data Type Tradeoffs

- What is the benefit of **having more or high-level data types** in the ISA?
- What is the disadvantage?
- Think compiler/programmer vs. microarchitect
- Concept of **semantic gap**
 - Data types coupled tightly to the semantic level, or complexity of instructions
- Example: Early RISC architectures vs. Intel 432
 - Early RISC (e.g., MIPS): Only integer data type
 - Intel 432: Object data type, capability based machine

Addressing Modes

- An addressing mode is a mechanism for specifying where an operand is located
- There 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 of them are memory addressing modes
 - PC-relative
 - Indirect
 - Base+offset
- In addition, MIPS has pseudo-direct addressing (for j and jal), but does not have indirect addressing

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 of the R-type operate instructions
 - **F-type** operations, i.e., floating-point operations

NOT in LC-3

■ NOT assembly and machine code

LC-3 assembly

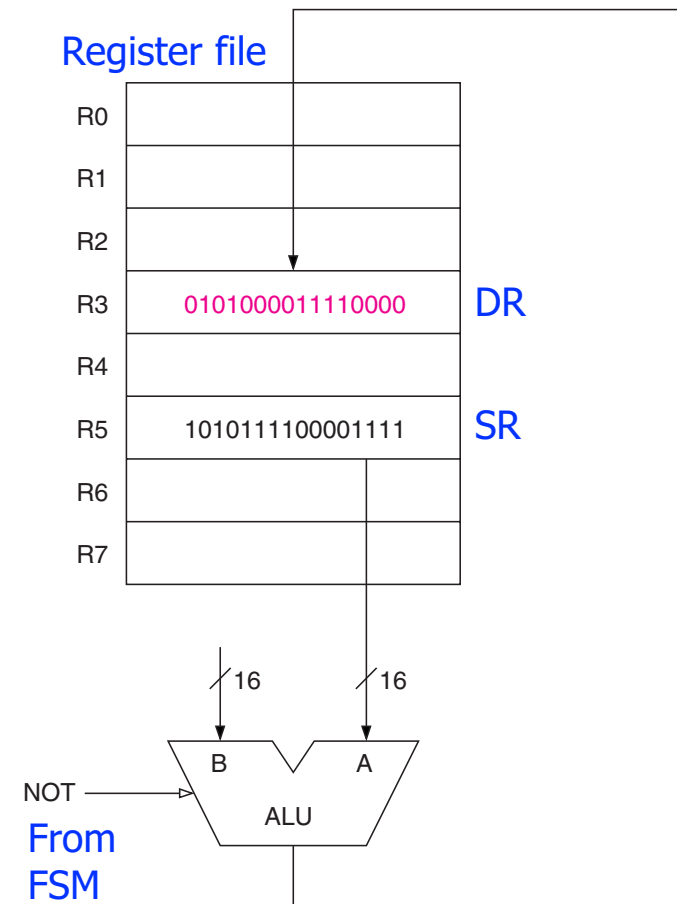
NOT R3, R5

Field Values

| OP | DR | SR | |
|----|----|----|-------------|
| 9 | 3 | 5 | 1 1 1 1 1 1 |

Machine Code

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



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

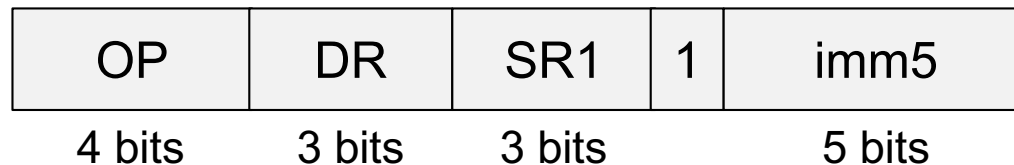
Operate Instructions

- We are already familiar with LC-3's ADD and AND with register mode (R-type in MIPS)
- Now let us see the versions with one literal (i.e., immediate) operand
- Subtraction is another necessary operation
 - How is it implemented in LC-3 and MIPS?

We did not cover the following slides in lecture.
These are for your preparation for the next lecture

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** + sign-extend(imm5)
 - E.g., **AND** = **0101** (same OP as the register-mode AND)
 - **DR** \leftarrow **SR1** AND sign-extend(imm5)
- SR1 = source register
- DR = destination register
- **imm5** = Literal or immediate (sign-extend to 16 bits)

ADD with one Literal in LC-3

■ ADD assembly and machine code

LC-3 assembly

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

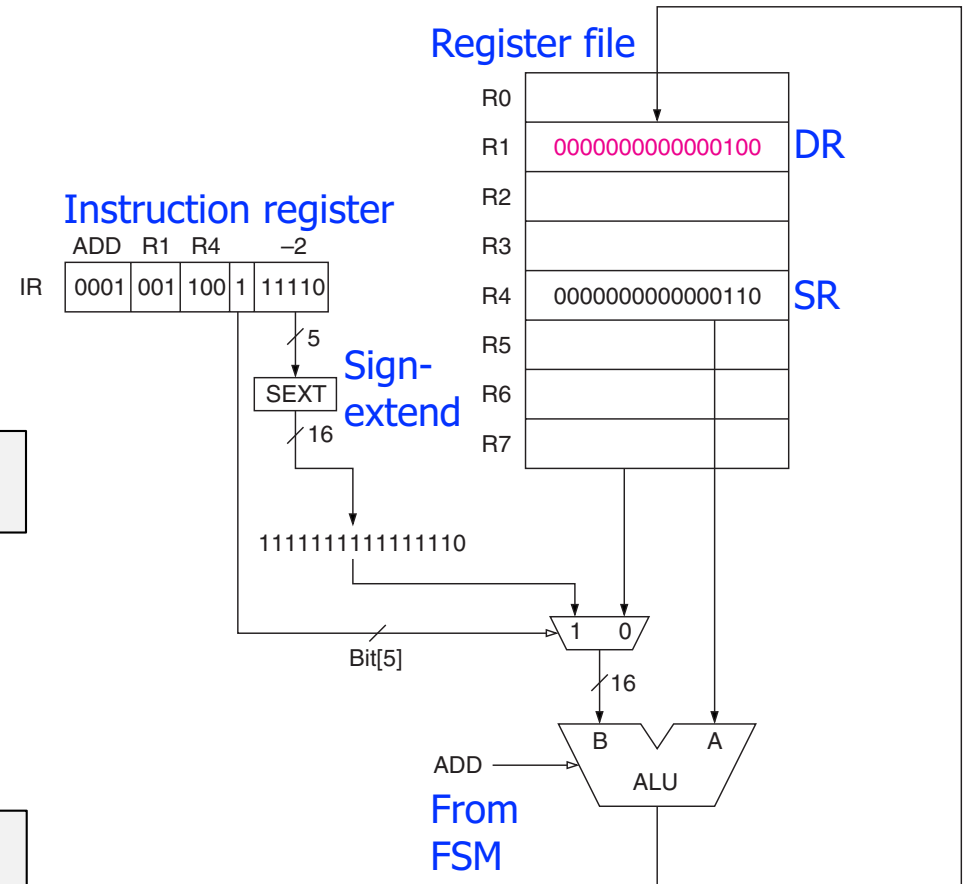
Field Values

| OP | DR | SR | imm5 |
|----|----|----|------|
| 1 | 1 | 4 | 1 |
| | | | -2 |

Machine Code

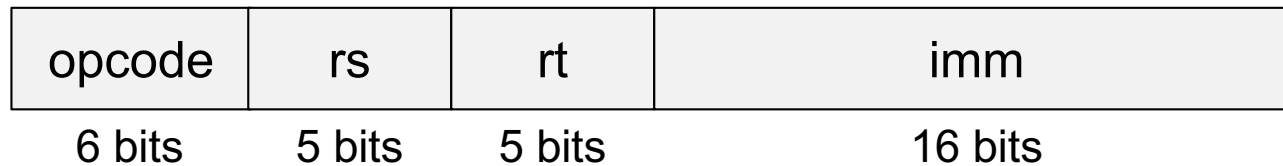
| OP | DR | SR | imm5 |
|------|-----|-----|-------|
| 0001 | 001 | 100 | 1 |
| | | | 11110 |

15 12 11 9 8 6 5 4 0



Instructions with one Literal in MIPS

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



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

Add with one Literal in MIPS

- Add immediate

MIPS assembly

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

Field Values

| op | rs | rt | imm |
|----|----|----|-----|
| 0 | 17 | 16 | 5 |

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

Machine Code

| op | rs | rt | imm |
|--------|-------|-------|---------------------|
| 001000 | 10001 | 10010 | 0000 0000 0000 0101 |

0x22300005

Subtract in LC-3

■ MIPS assembly

High-level code

```
a = b + c - d;
```

MIPS assembly

```
add    $t0, $s0, $s1  
sub    $s3, $t0, $s2
```

■ LC-3 assembly

High-level code

```
a = b + c - d;
```

LC-3 assembly

```
ADD    R2, R0, R1  
NOT    R4, R3  
ADD    R5, R4, #1  
ADD    R6, R2, R5
```

2's
complement
of R4

■ Tradeoff in LC-3

- ❑ More instructions
- ❑ But, simpler control logic

Subtract Immediate

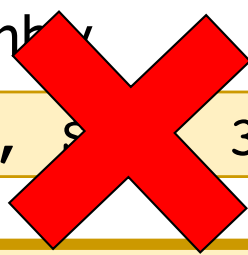
- MIPS assembly

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
```

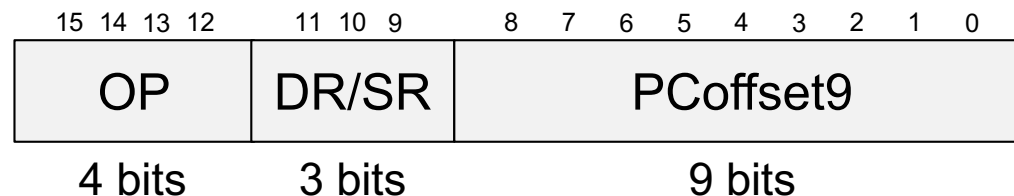
Data Movement Instructions and Addressing Modes

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
```

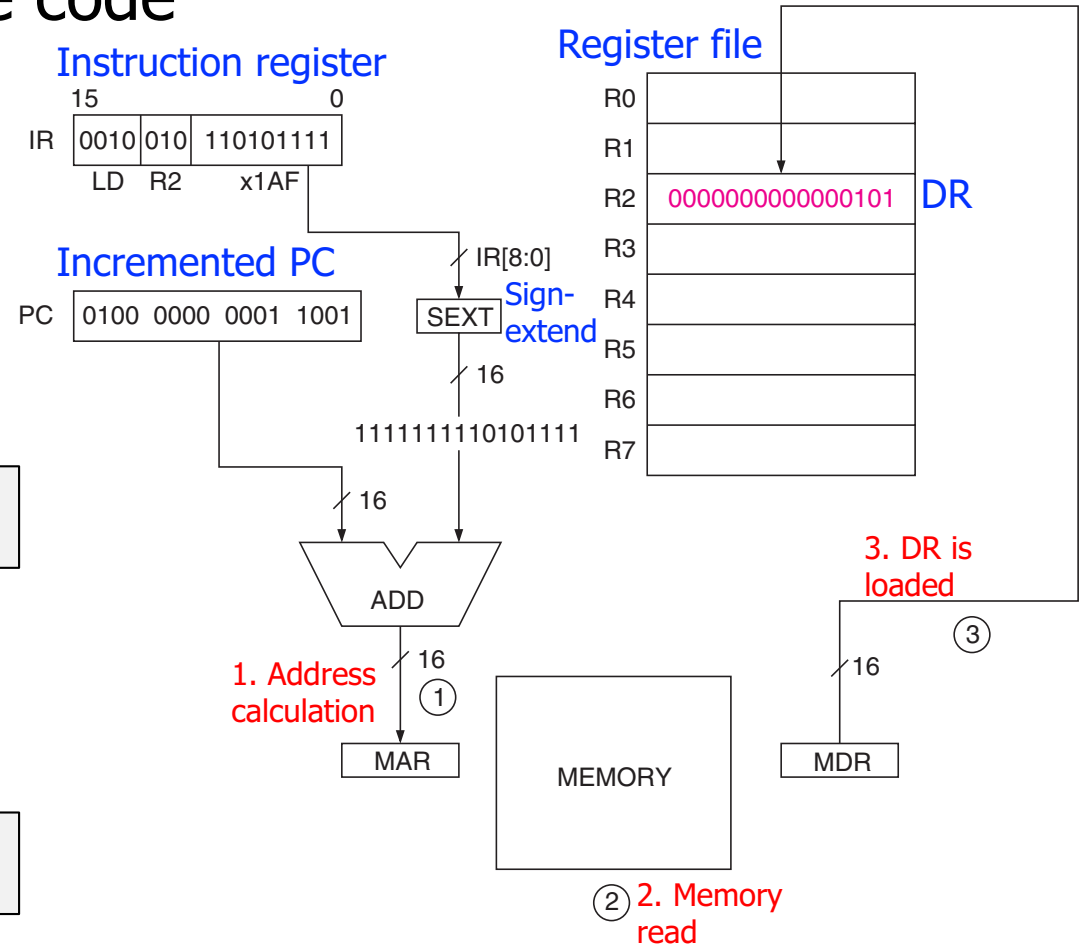
Field Values

| OP | DR | PCOffset9 |
|----|----|-----------|
| 2 | 2 | 0x1AF |

Machine Code

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

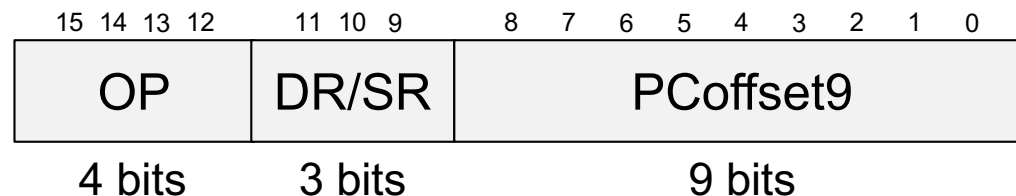
The memory address is **only +256 to -255** locations away of the **LD or ST instruction**



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

[†] This is the incremented PC

LDI in LC-3

LDI assembly and machine code

LC-3 assembly

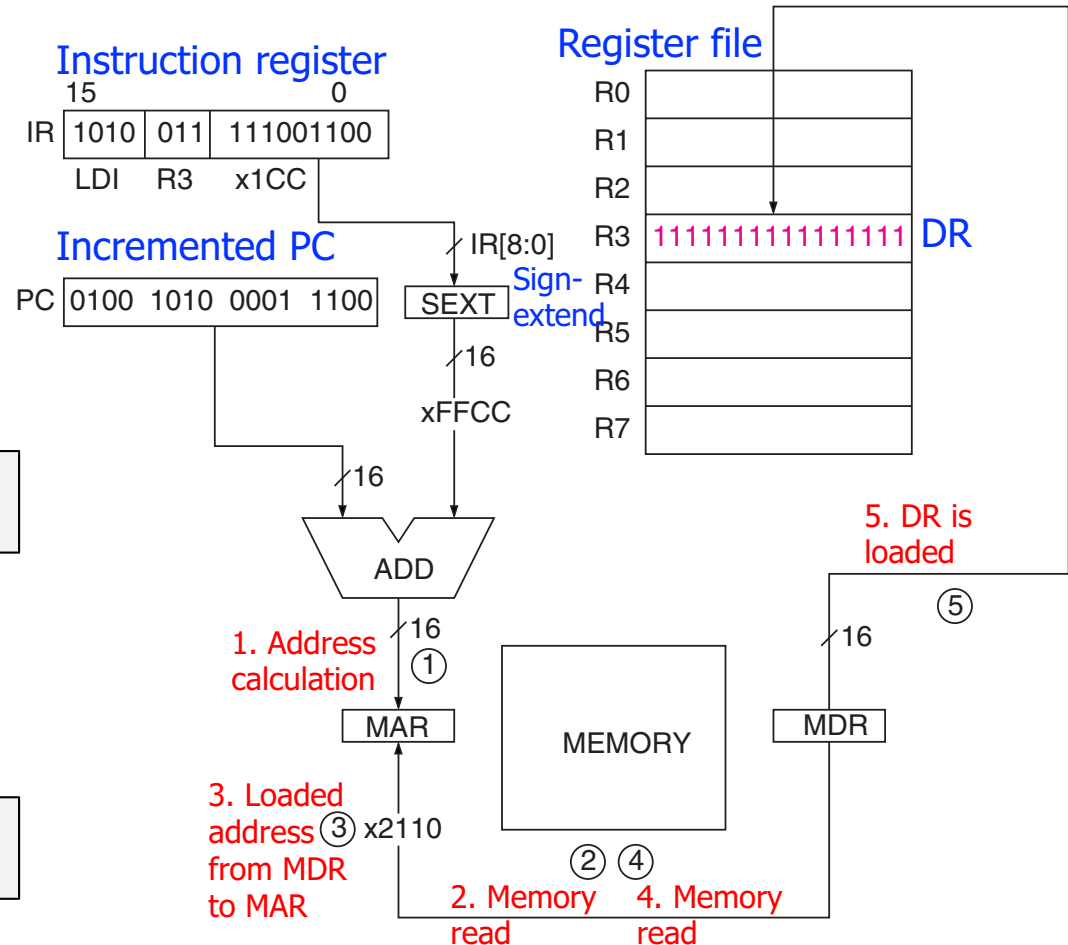
```
LDI R3, 0x1CC
```

Field Values

| OP | DR | PCOffset9 |
|----|----|-----------|
| A | 3 | 0x1CC |

Machine Code

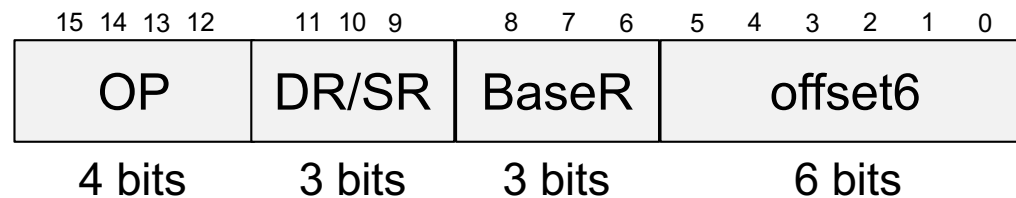
| OP | DR | PCOffset9 |
|---------|-------|-------------------|
| 1 0 1 0 | 0 1 1 | 1 1 1 0 0 1 1 0 0 |
| 15 | 12 | 11 9 8 0 |



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

Base+Offset Addressing Mode

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



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

LDR in LC-3

■ LDR assembly and machine code

LC-3 assembly

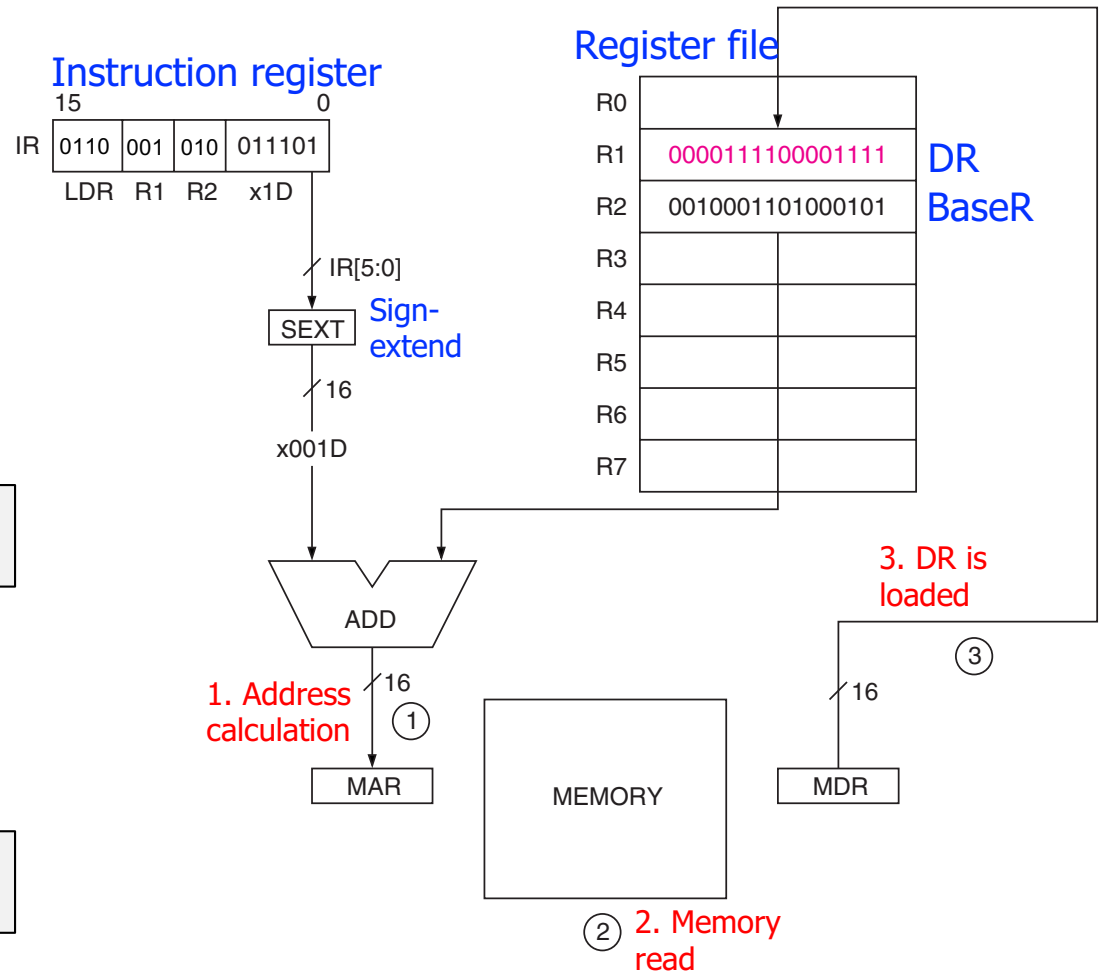
```
LDR R1, R2, 0x1D
```

Field Values

| OP | DR | BaseR | offset6 |
|----|----|-------|---------|
| 6 | 1 | 2 | 0x1D |

Machine Code

| OP | DR | BaseR | offset6 |
|------|-----|-------|---------|
| 0110 | 001 | 010 | 011101 |
| 15 | 12 | 11 | 9 |
| 8 | 6 | 5 | 0 |



The address of the operand can also be **anywhere in the memory**

Base+Offset Addressing Mode in MIPS

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

High-level code

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

MIPS assembly

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

Memory[\$s0 + 8] ← \$s3

Field Values

| op | rs | rt | imm |
|----|----|----|-----|
| 43 | 16 | 19 | 8 |

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

An Example Program in MIPS and LC-3

High-level code

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

MIPS registers

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

LC-3 registers

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

MIPS assembly

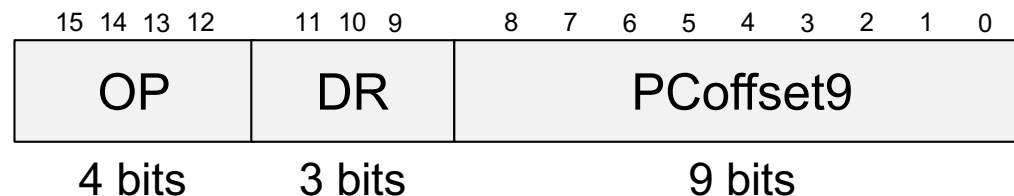
```
lw    $t0, 0($s0)  
add   $t1, $t0, $s2  
addi  $t2, $t1, -5  
sw    $t2, 0($s1)
```

LC-3 assembly

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

Immediate Addressing Mode

■ LEA (Load Effective Address)



- OP = 1110
- DR = destination register
- LEA: $DR \leftarrow PC^{\dagger} + \text{sign-extend}(\text{PCOffset9})$

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

Answer: Instructions with **PC-Relative** mode **access memory**,
but **LEA does not**

[†] This is the incremented PC

LEA in LC-3

LEA assembly and machine code

LC-3 assembly

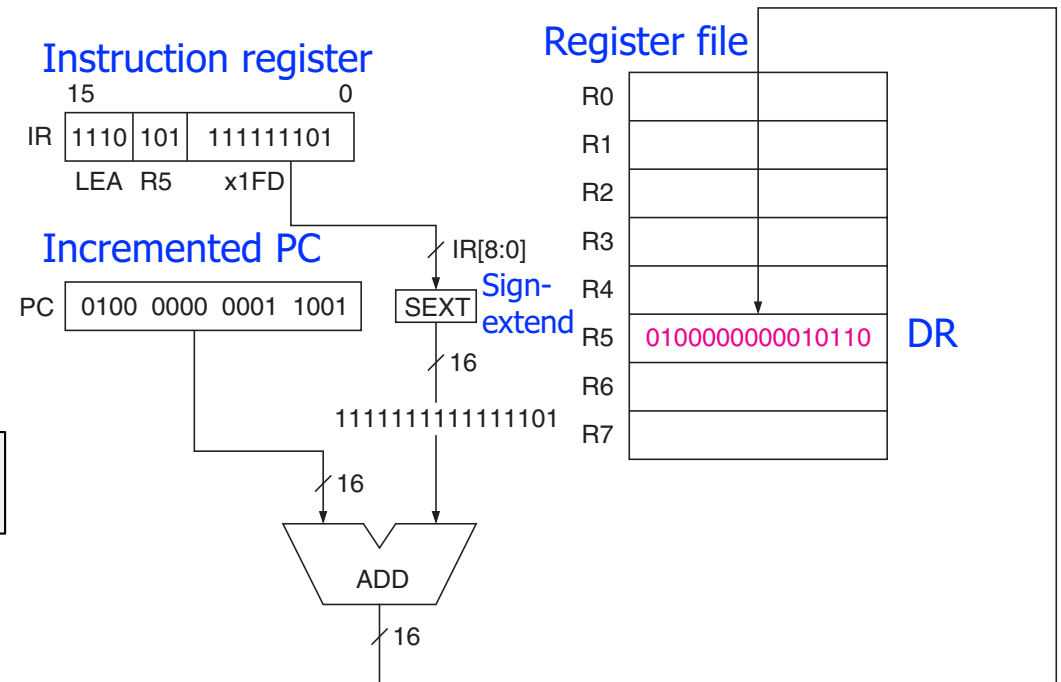
```
LEA R5, #-3
```

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 |



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?

| 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 | 0 | 1 | 1 | 1 | 1 | R3 ← M[M[x30F4]] |

Addressing Example in LC-3

- What is the final value of R3?

| 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 | 0 | 1 | $R1 = PC - 3 = 0x30F7 - 3 = 0x30F4$ |
| x30F7 | ADD | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | $R2 = R1 + 14 = 0x30F4 + 14 = 0x3102$ |
| x30F8 | ST | 0 | 1 | 1 | 0 | 1 | 0 | 5 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | $M[PC - 5] = M[0x30F4] = 0x3102$ |
| 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 | 1 | 1 | 1 | 0 | $M[R1 + 14] = M[0x30F4 + 14] = M[0x3102] = 5$ |
| x30FC | LDI | 0 | 1 | 0 | 0 | 1 | 1 | 9 | 1 | 1 | 1 | 1 | 0 | 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 jumps
 - **Conditional branches** are used to **make decisions**
 - E.g., if-else statement
 - In LC-3, three **condition codes** are used
 - **Jumps** are used to implement
 - **Loops**
 - **Function calls**
 - **JMP** in LC-3 and **j** in MIPS

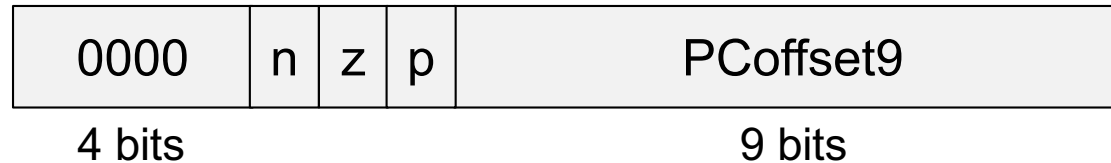
Condition Codes in LC-3

- Each time one GPR (R0-R7) is written, **three single-bit registers** are updated
- Each of these **condition codes** are either set (set to 1) or cleared (set to 0)
 - If the written value is **negative**
 - **N** is set, Z and P are cleared
 - If the written value is **zero**
 - **Z** is set, N and P are cleared
 - If the written value is **positive**
 - **P** is set, N and P are cleared
- SPARC and x86 are examples of ISAs that use condition codes

Conditional Branches in LC-3

■ BRz

BRz PCoffset9



- n, z, p = which N, Z, and/or P is tested
- PCoffset9 = immediate or constant value
- if $((n \text{ AND } N) \text{ OR } (p \text{ AND } P) \text{ OR } (z \text{ AND } Z))$
 - then $PC \leftarrow PC^{\dagger} + \text{sign-extend}(\text{PCoffset9})$
- Variations: BRn, BRz, BRp, BRzp, BRnp, BRnz, BRnzp

[†] This is the incremented PC

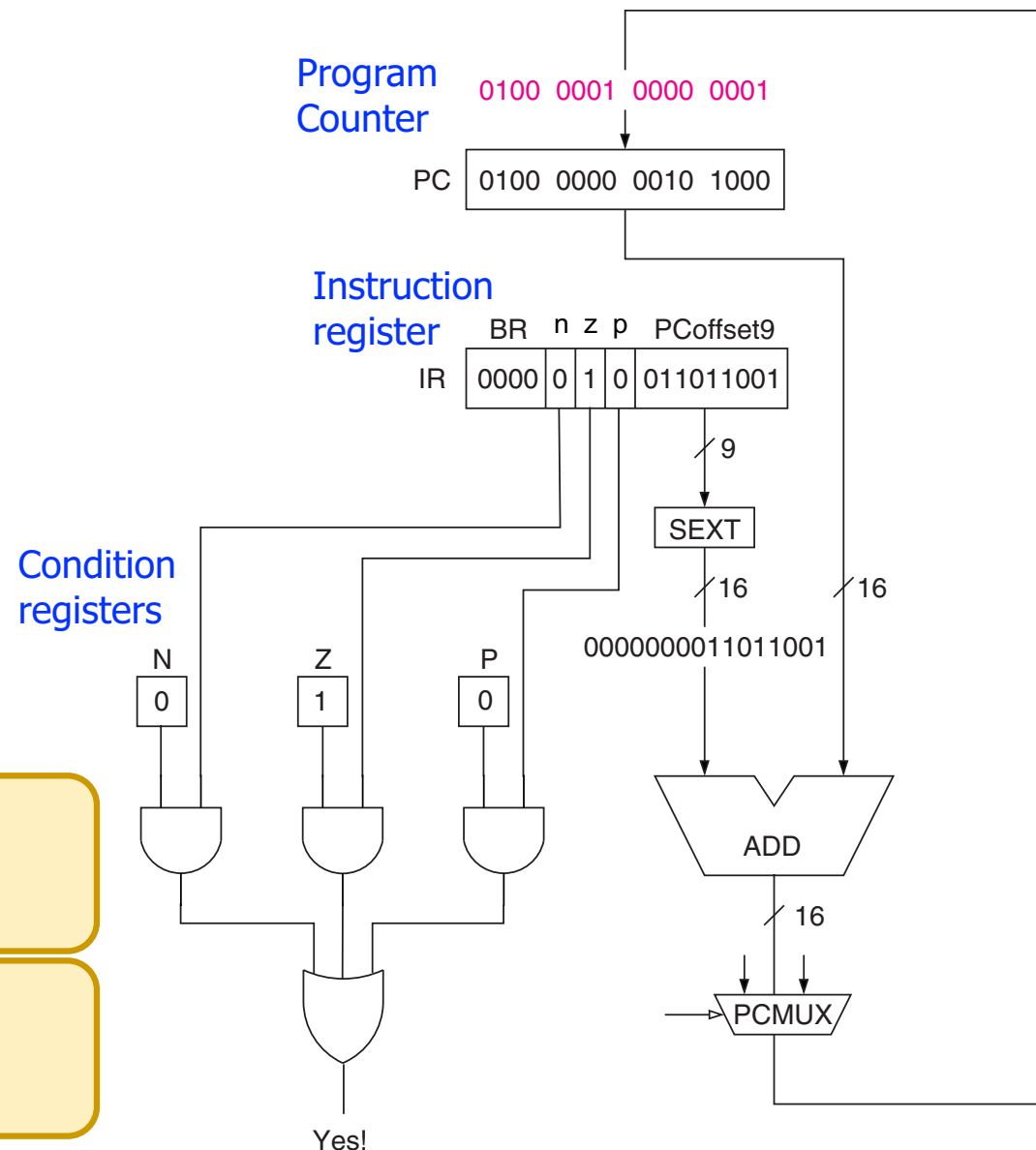
Conditional Branches in LC-3

■ BRz

BRz 0x0D9

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

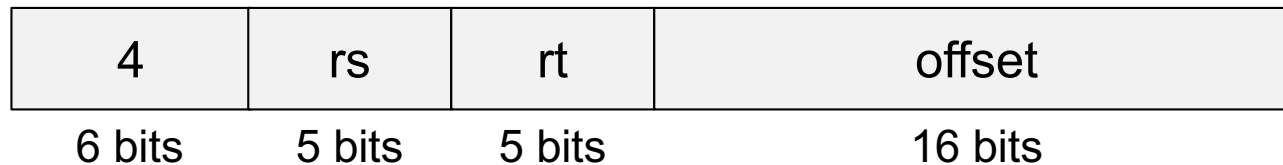
And what if $n = z = p = 0$?



Conditional Branches in MIPS

■ 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**

Lecture Summary

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

Design of Digital Circuits

Lecture 9: Von Neumann Model, ISA, LC-3 and MIPS

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zurich

Spring 2018

22 March 2018