COMP2300-COMP6300-ENGN2219
Computer Organization &
Program Execution

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

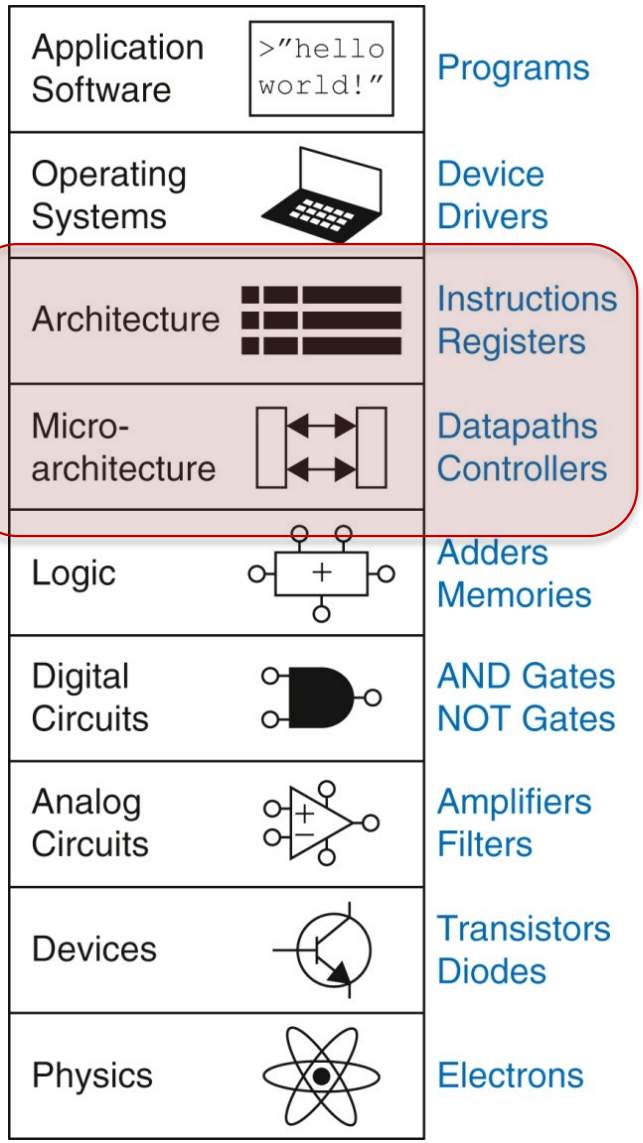
Australian
National
University

Our Status

- We are done with digital logic fundamentals that we need to understand and build a CPU

- We are now at
  - Architecture layer

- Next week
  - Microarchitecture layer



ISA then microarchitecture

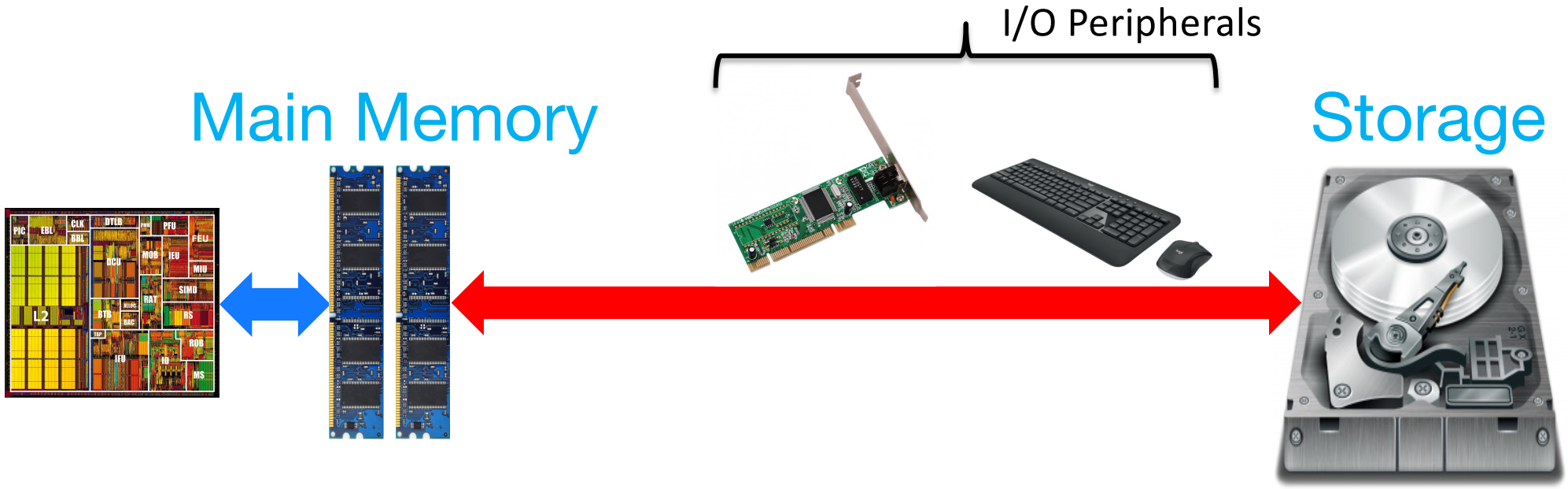
Admin

- Checkpoint (5% of grade) marking is under way
- Assignment 1 will be release today
  - We will be adjusting the due date slightly
- 25% of assignment 1 grade comes from work you are doing in Labs 4 – 6

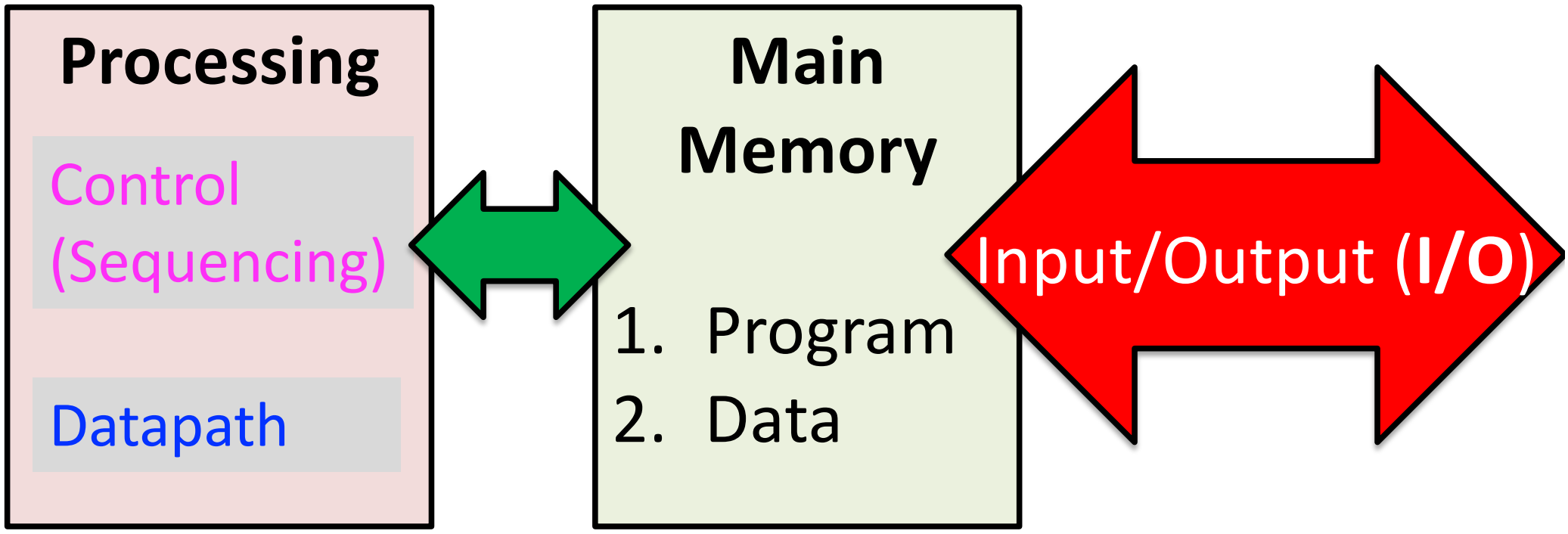
Von Neumann Model

Recall: A Computer System

- **Key resources:** CPU, memory, and Input/Output (I/O) devices
  - CPU (processor) does the actual processing (computation)
  - Memory stores temporary data and forms a hierarchy (Registers, SRAM, DRAM, ...)
  - Some fast (small capacity) memory is close to the CPU; rest is far
  - Storage disk is an I/O device (much slower than memory, stores persistent data)
  - Memory is volatile, while disk is non-volatile



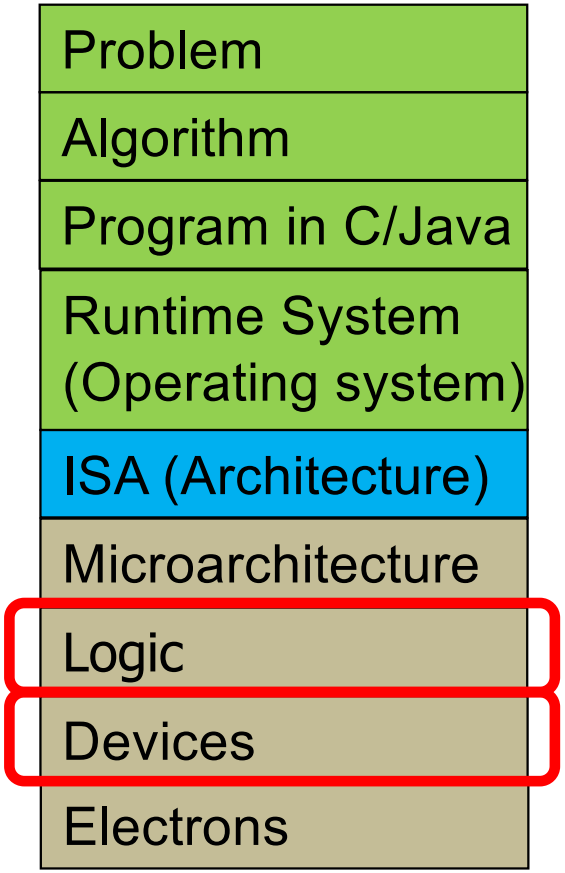
What is a Computer?



- We will cover all three components

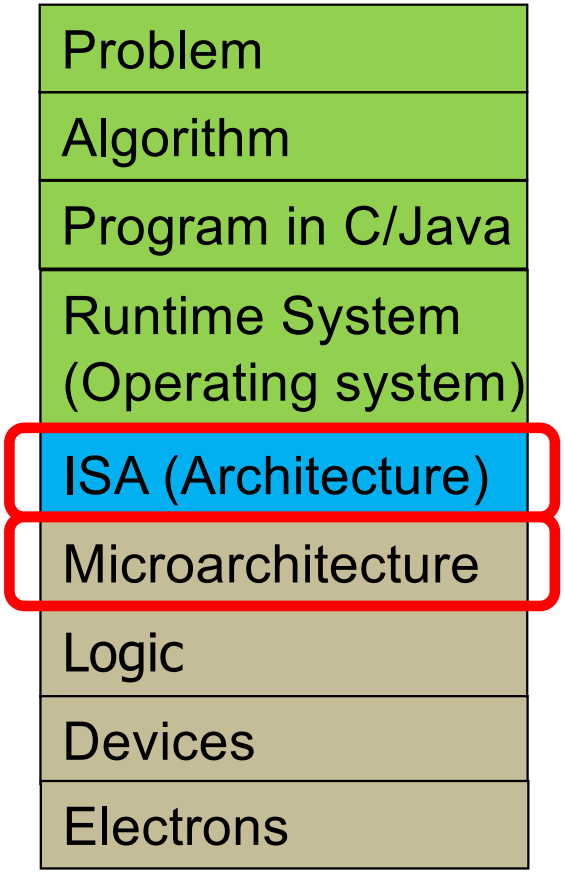
Building up a Basic Computer Model

- In past lectures, we learned how to design
  - Combinational logic structures
  - Sequential logic structures
- With logic structures, we can build
  - Execution units
  - Decision units
  - Memory/storage units
  - Communication units
- All are basic elements of a computer
  - We will raise our abstraction level today
  - Use logic structures to construct a basic computer model



Building up a Basic Computer Model

- **ISA:** Specification of the instructions computer can perform
- **Microarchitecture:** Circuit implementation of the specification



What is a Computer?

- To get a task done by a (general-purpose) computer, we need
  - A computer program
    - That specifies what the computer must do
  - The computer itself
    - To carry out the specified task

- Program: A set of instructions
  - Each instruction specifies a well-defined piece of work for the computer to carry out
  - Instruction: the smallest piece of specified work in a program

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

The Von Neumann Model

- In order to build a computer, we need an execution model for processing computer programs

- John von Neumann proposed a fundamental model in 1946

- The von Neumann Model consists of 5 components
  - Memory (stores the program and data)
  - Processing unit
  - Input
  - Output
  - Control unit (controls the order in which instructions are carried out)

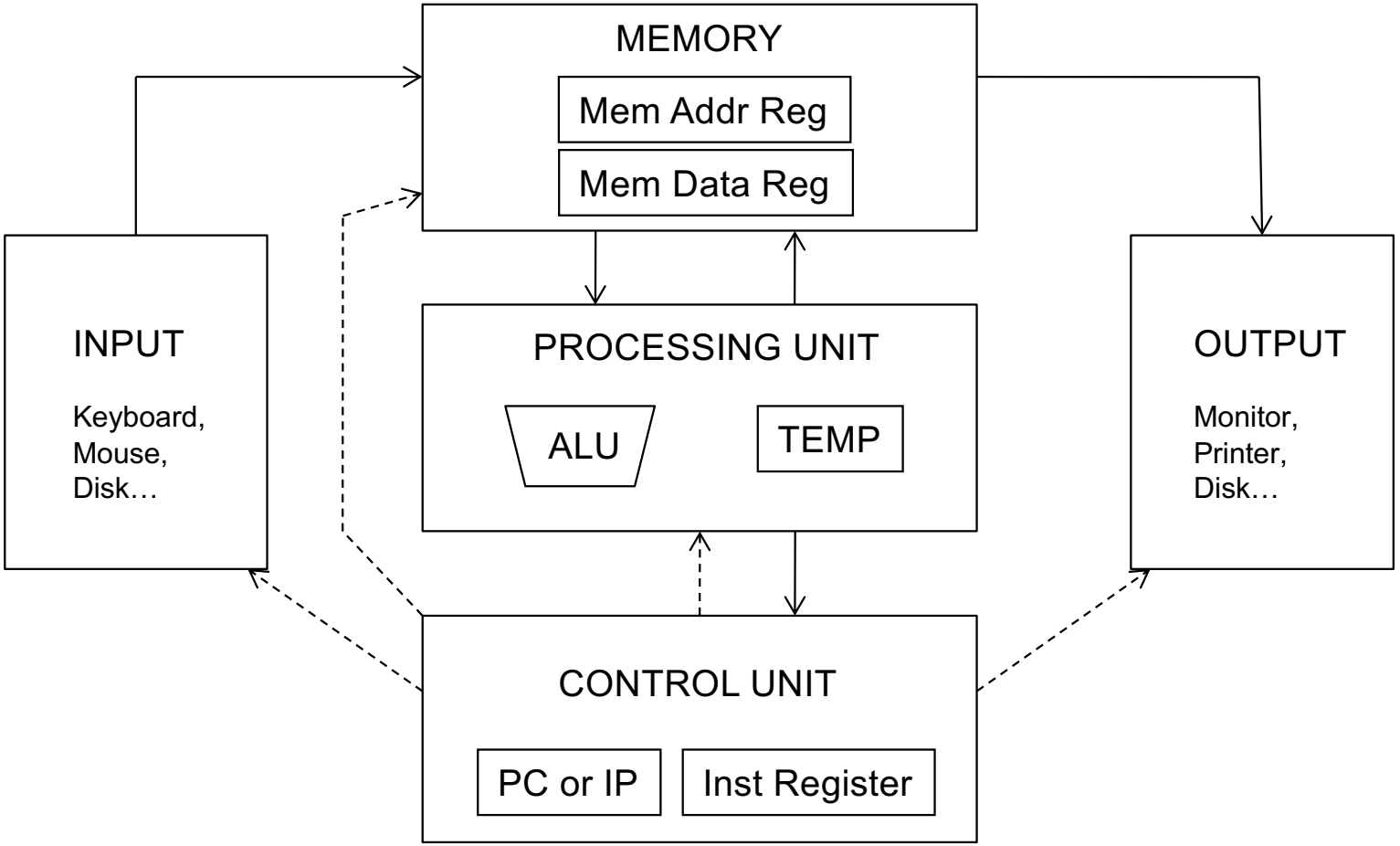
Burks, Goldstein, von Neumann,
"Preliminary discussion of the logical design of an electronic computing instrument," 1946.

All general-purpose computers today use the von Neumann model

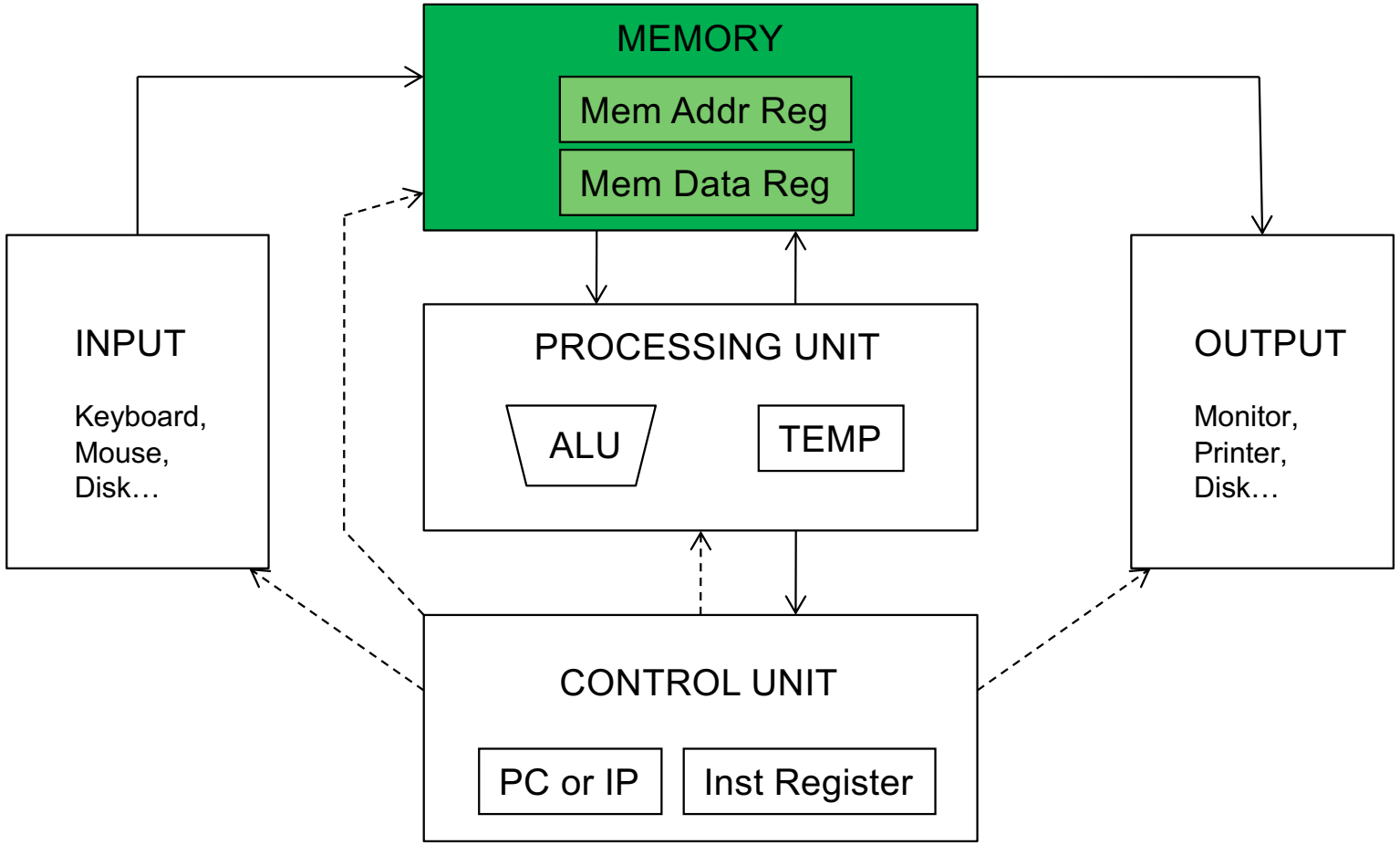
The Von Neumann Model

- Throughout this lecture, we will examine two examples of the von Neumann model
  - ARM
  - MIPS

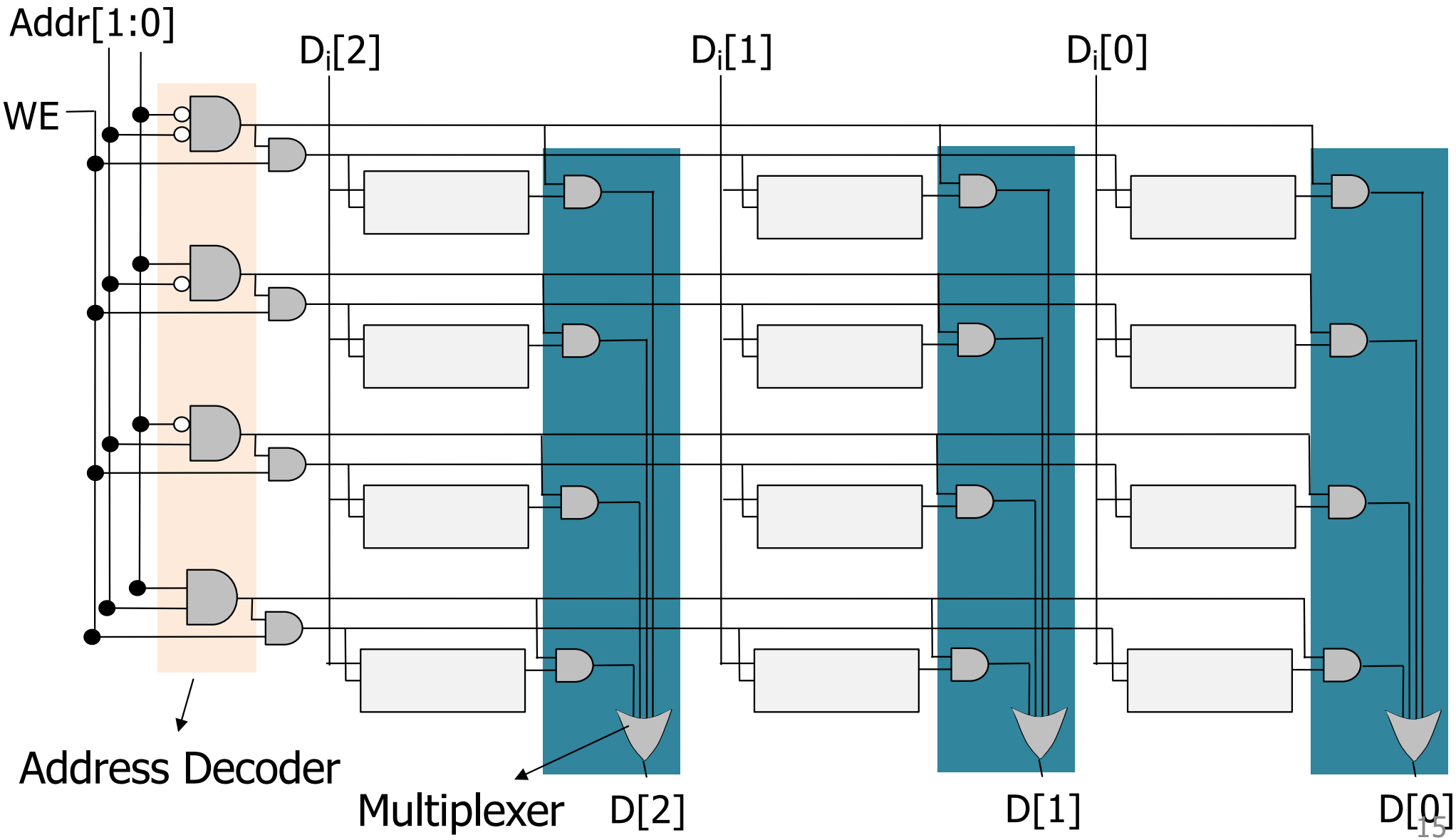
The Von Neumann Model



The Von Neumann Model

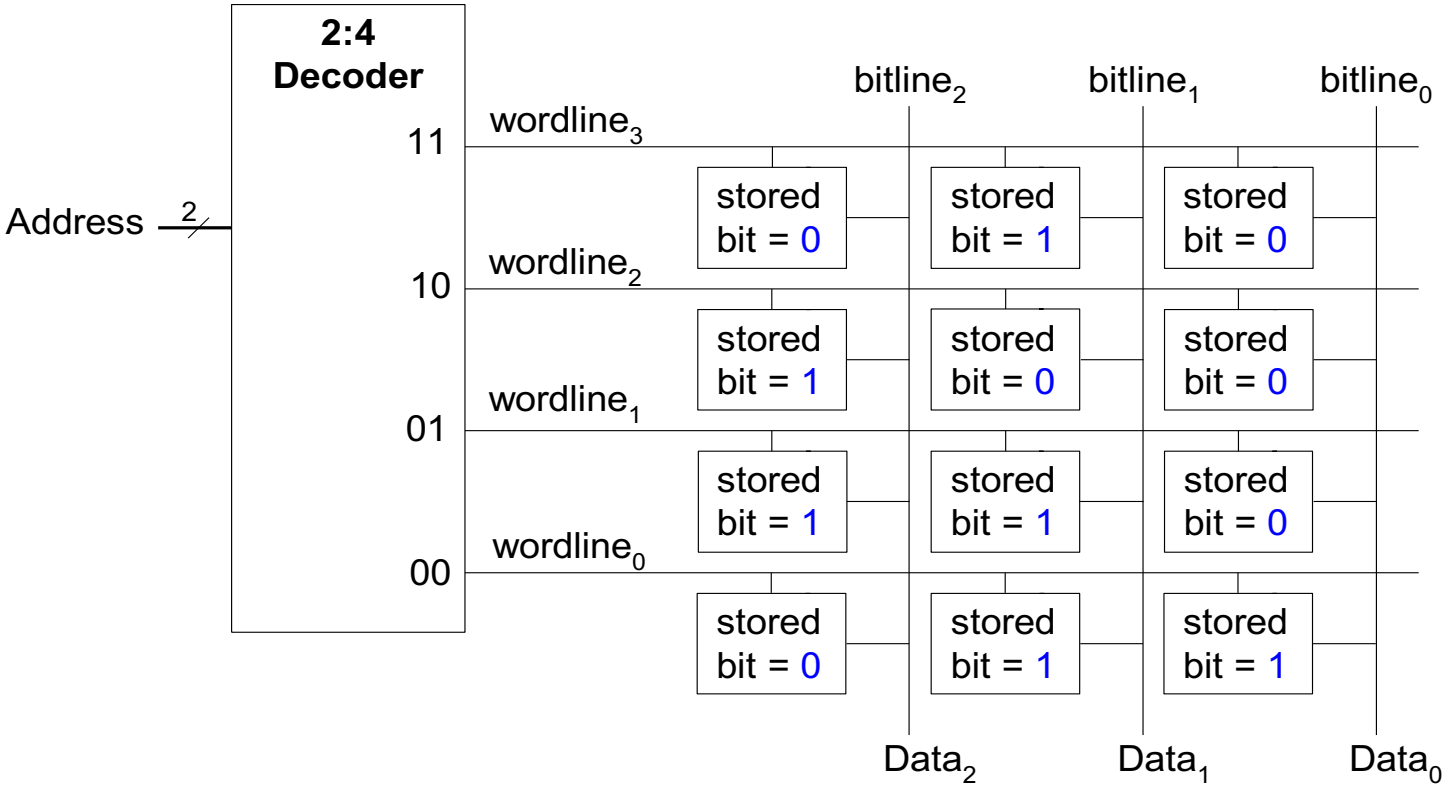


Recall: A Memory Array (4 locations X 3 bits)


Addr[1:0]
WE
$D_i[2]$
$D_i[1]$
$D_i[0]$
Address Decoder
Multiplexer
D[2]
D[1]
D[0]

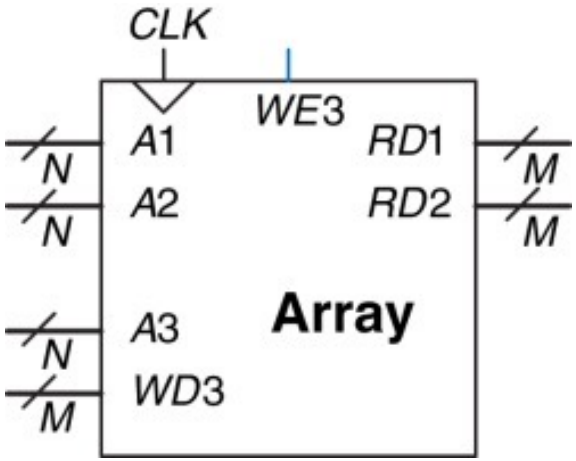
Recall: Memory Array Organization

- Decoder drives the wordline HIGH based on the address
- Data on the selected row appears on the bitlines



Recall: Memory Ports

- Each memory port gives **read** or **write** access to one memory address
- Multiported memories can access **multiple** addresses **simultaneously**
- **Example of three-ported memory**
  - Port 1 reads the data from address A1 onto the read data output RD1
  - Port 2 reads the data from address A2 onto the read data output RD2
  - Port 3 writes the data from the write data input WD3 into address A3 on the rising clock edge if WE3 is **TRUE**



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 MIPS, the address space is 2^{32}
    - 32-bit addresses
  - In ARM, the address space is 2^{32}
    - 32-bit addresses
  - In x86-64, the address space is (up to) 2^{48}
    - 48-bit addresses
- Addressability: How many bits are stored in each location (address)
  - E.g., 8-bit addressable (or byte-addressable)
  - E.g., word-addressable
  - A given instruction can operate on a byte or a word

A Simple Example

- A representation of memory with 8 locations
- Each location contains 8 bits (one byte)
  - Byte addressable memory; address space of 8
  - Value 6 is stored in address 4 & value 4 is stored in address 6

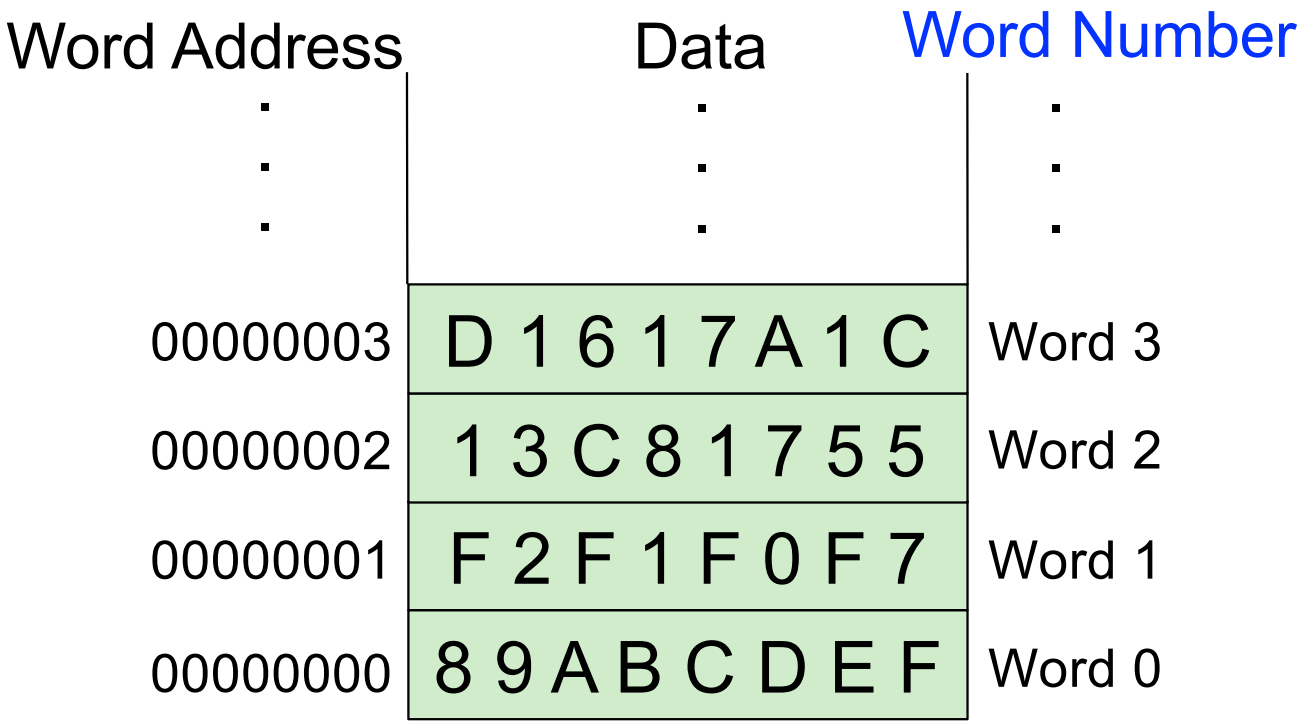
Address	Data Value
000	
001	
010	
011	
100	00000110
101	
110	00000100
111	

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

Answer:
64 locations
Each location stores 1 bit

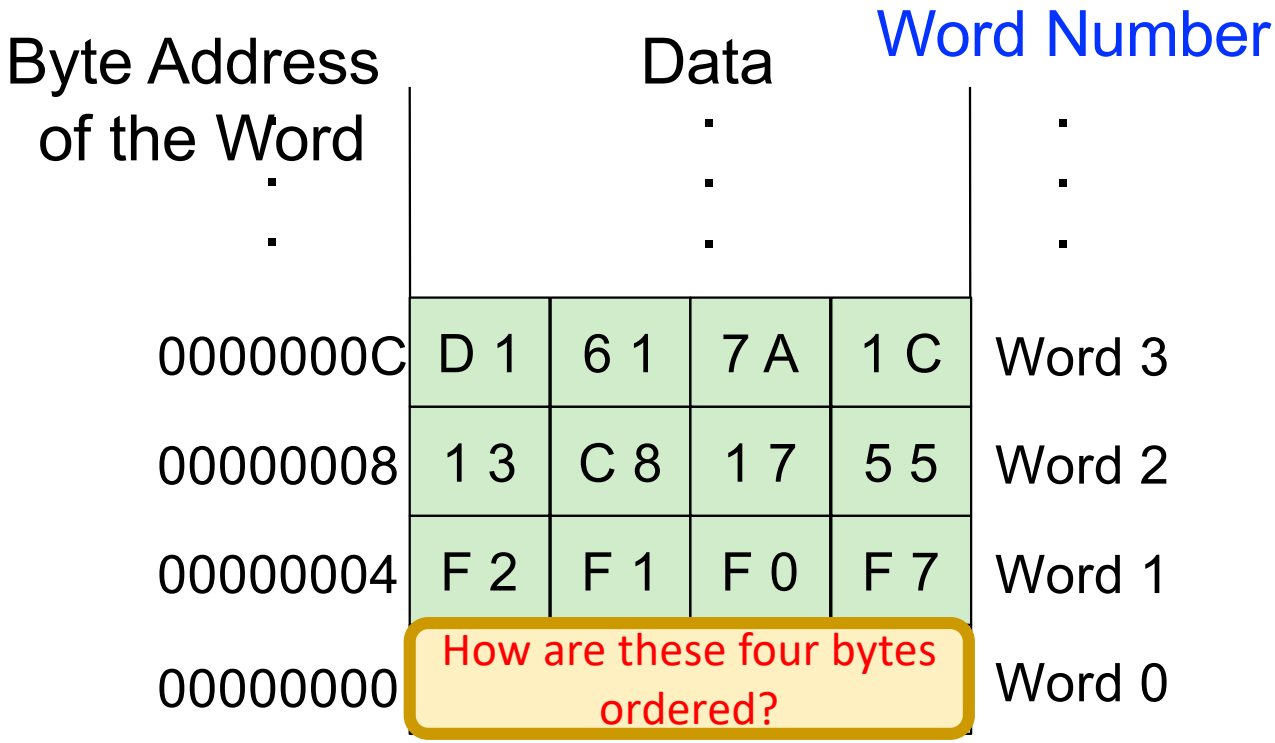
Word-Addressable Memory

- Each data word has a unique address
  - In MIPS, a unique address for each 32-bit data word
  - In **QuAC**, a unique address for each 16-bit data word



Byte-Addressable Memory

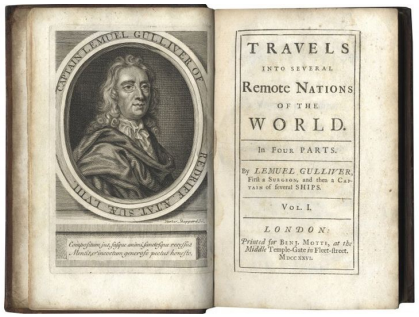
- Each byte has a unique address
  - MIPS is actually byte-addressable
  - ARM is also byte-addressable

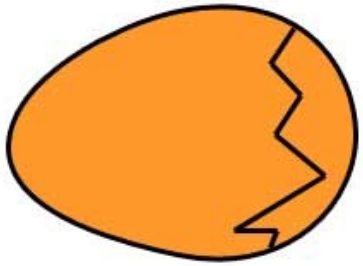


Which of the four bytes is most vs. least significant?

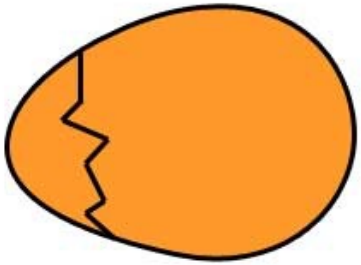
Big Endian vs. Little Endian

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



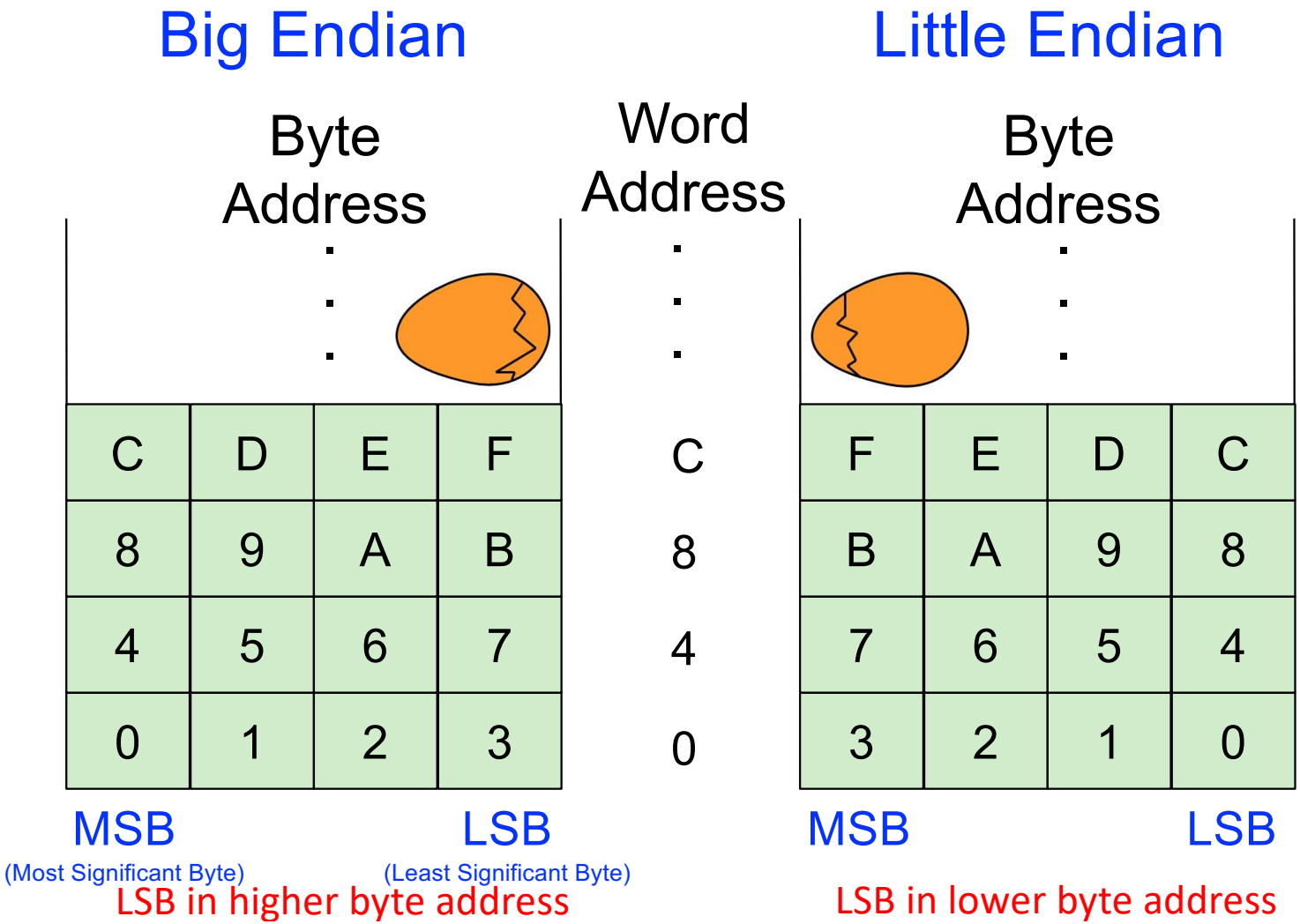


BIG ENDIAN - The way people always broke their eggs in the Lilliput land

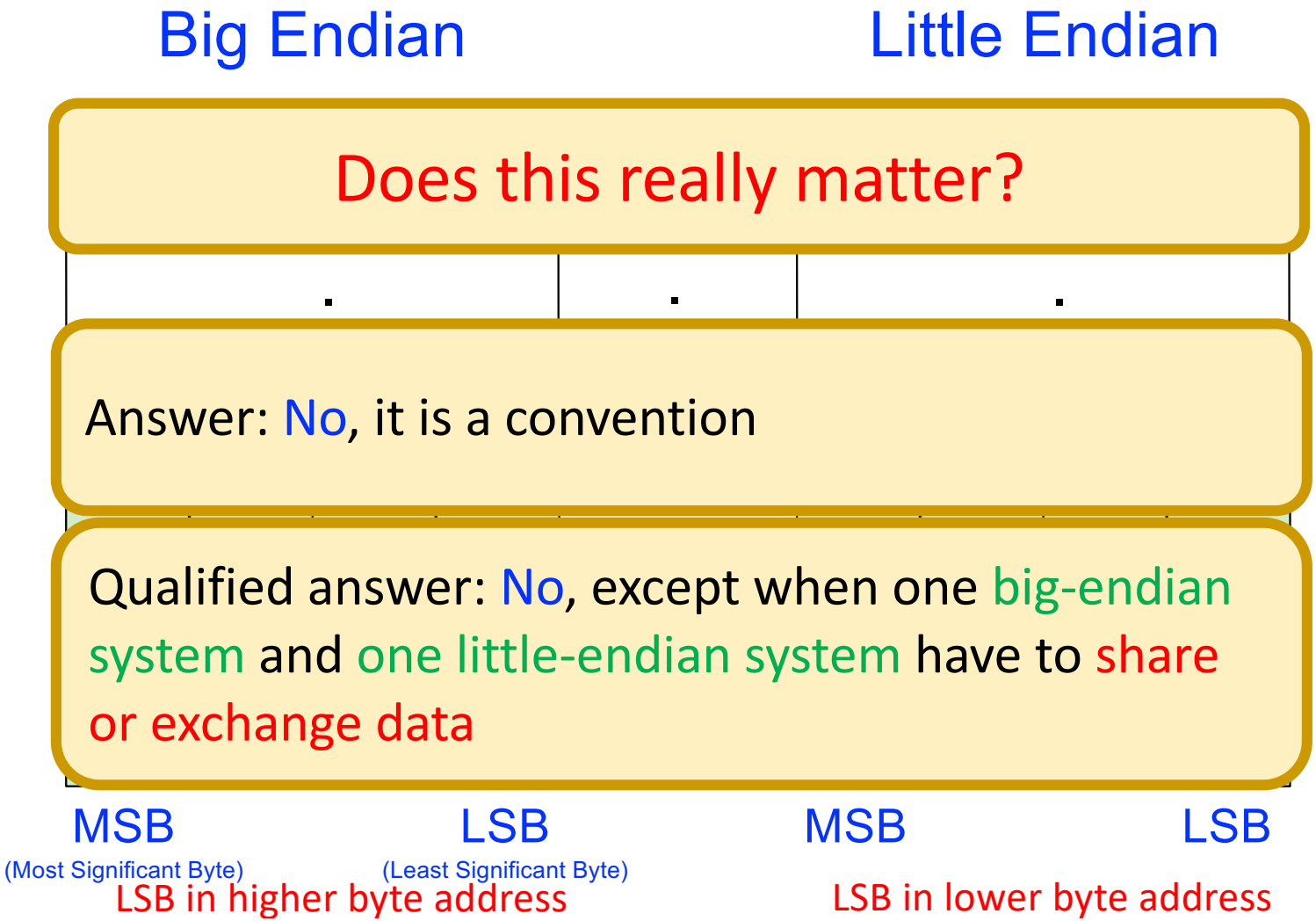


LITTLE ENDIAN - The way the king then ordered the people to break their eggs

Big Endian vs. Little Endian

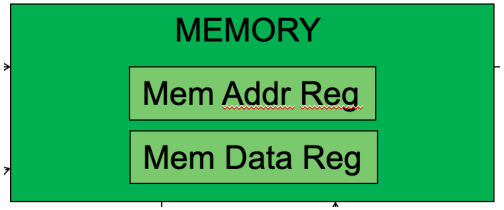


Big Endian vs. Little Endian



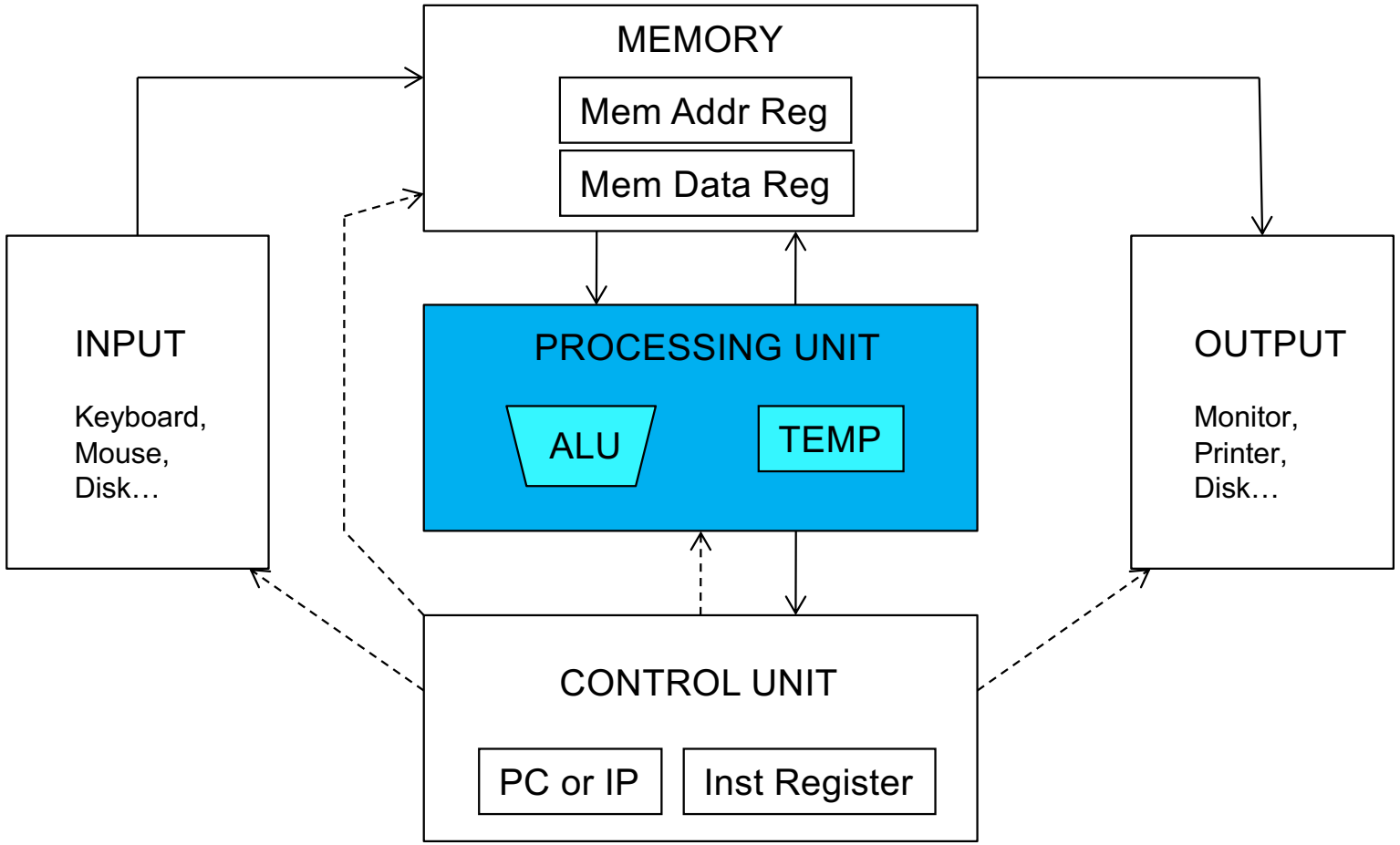
Accessing Memory: MAR and MDR

- There are two ways of accessing memory
  - Reading or loading data from a memory location
  - Writing or storing data to a memory location
- Two registers are usually used to access memory
  - Memory Address Register (MAR)
  - Memory Data Register (MDR)



- To read
  - Step 1: Load the MAR with the address we wish to read from
  - Step 2: Data in the corresponding location gets placed in MDR
- To write
  - Step 1: Load the MAR with the address and the MDR with the data we wish to write
  - Step 2: Activate Write Enable signal → value in MDR is written to address specified by MAR

The Von Neumann Model



Processing Unit

- Performs the actual computation(s)
- The processing unit can consist of many functional units
- We start with a simple Arithmetic and Logic Unit (ALU), which executes computation and logic operations
  - ARM: ADD, AND, NOT, SUB
  - MIPS: add, sub, mult, and, nor, sll, slr, slt…
- The ALU processes quantities that are referred to as words
  - Word length in ARM**v4** is 32 bits (**v8** is 64 bits)
  - Word length in MIPS is 32 bits

Recall: Arithmetic & 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:

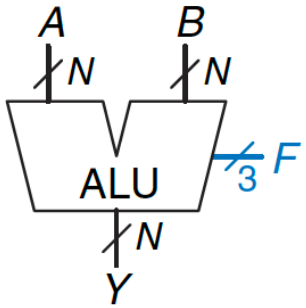


Figure 5.14 ALU symbol

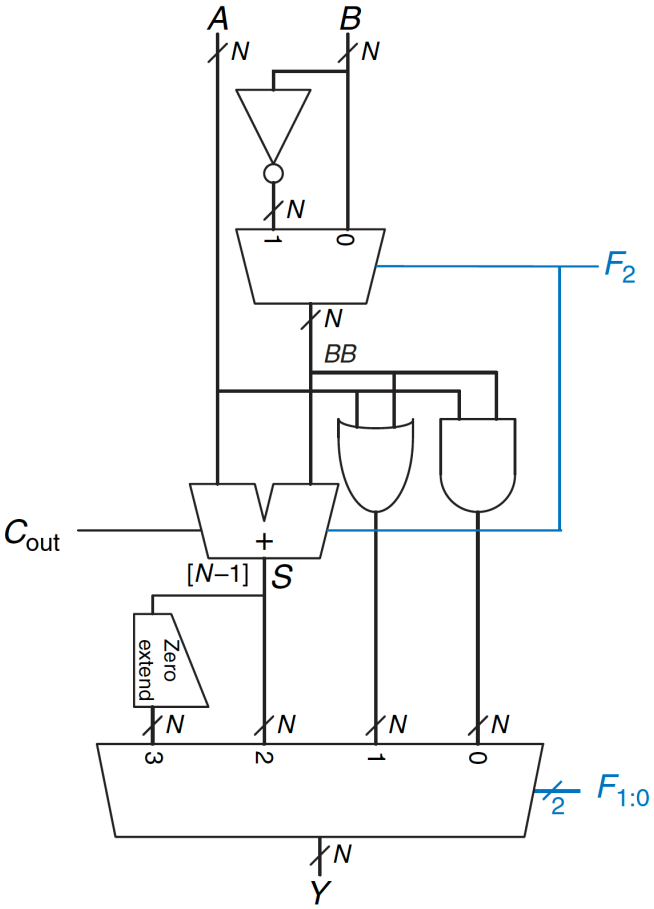
Table 5.1 ALU operations

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

Recall: Arithmetic & Logic Unit (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 $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT

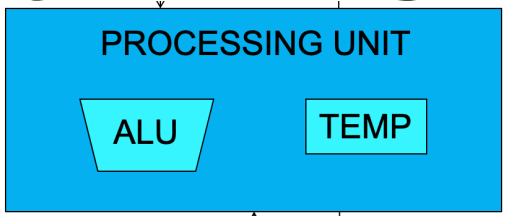


We Covered Until Here Last Week

Processing Unit: Fast Temporary Storage

- It is almost always the case that a computer provides a small amount of storage very close to ALU
  - Purpose: to store temporary values and quickly access them later
- E.g., to calculate ((A+B)*C)/D, the intermediate result of A+B can be stored in temporary storage
  - Why? It is too slow to store each ALU result in memory & then retrieve it again for future use
    - A memory access is much slower than an addition, multiplication or division
  - Ditto for the intermediate result of ((A+B)*C)
- This temporary storage is usually a set of registers
  - Called Register File

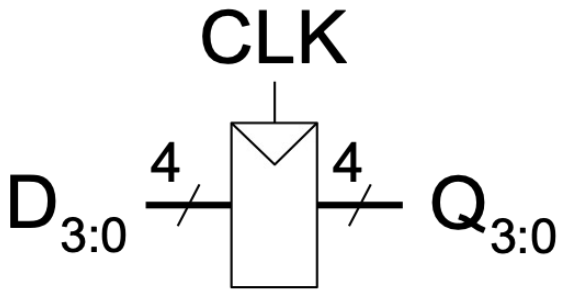
Registers: Fast Temporary Storage



- Memory is large but slow
- Registers in the Processing Unit
  - Ensure fast access to values to be processed in the ALU
  - Typically one register contains one word (same as word length)
- Register Set or Register File
  - Set of registers that can be manipulated by instructions
  - ARM has 16 general purpose registers (GPRs)
    - R0 to R15: 4-bit register number
    - Register size = Word length = 32 bits
  - MIPS has 32 general purpose registers
    - R0 to R31: 5-bit register number (or Register ID)
    - Register size = Word length = 32 bits

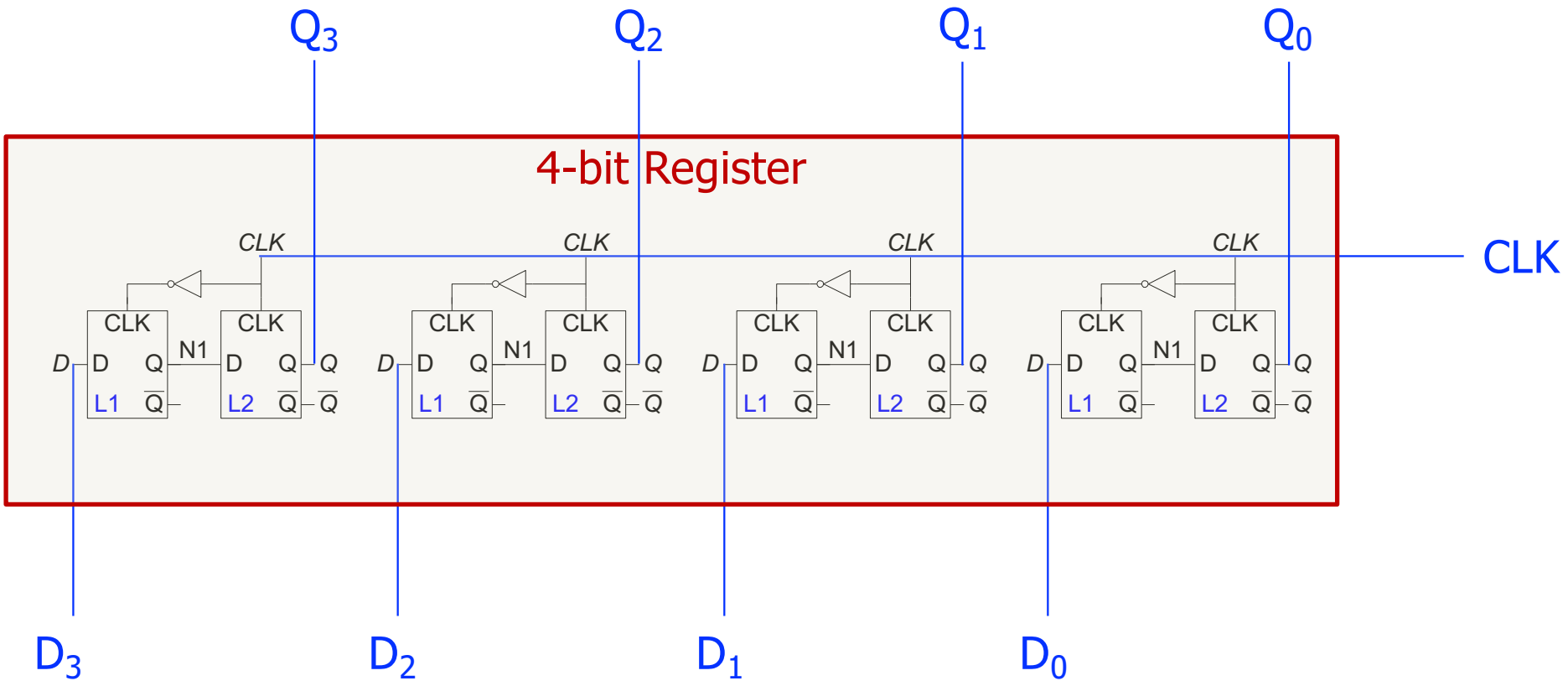
Recall: Register

- **How can we use flipflops to store more than one bit?**
  - **Principle of modularity:** Use more flipflops!
  - A single CLK to simultaneously write to all flipflops



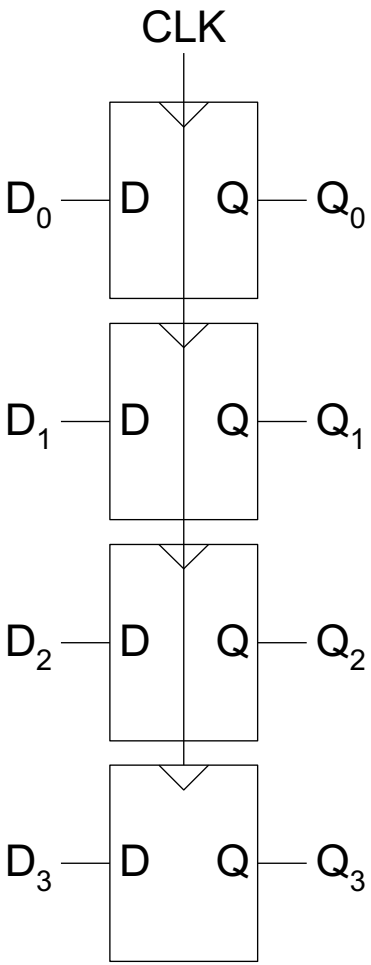
- **Register:** A structure that stores more than **one bit** of information and can be read from and written to
- This **register** holds 4 bits, and its data is referenced as Q[3:0]

Recall: 4-bit Register

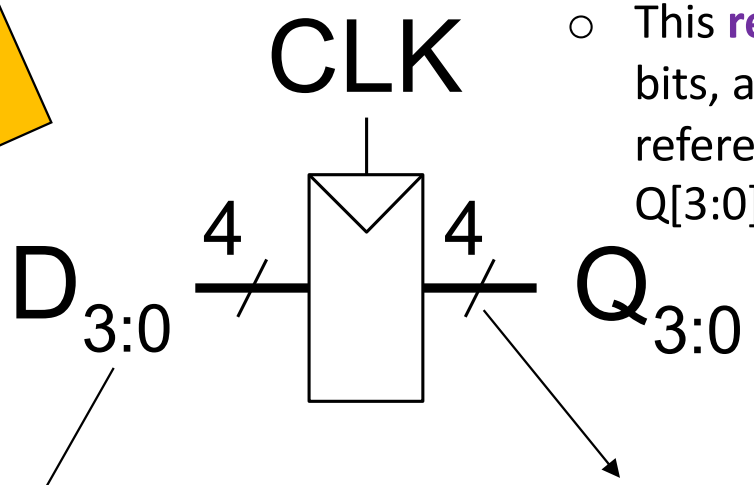


To build an **N-bit** register, use a bank of **N** flipflops with a shared **CLK**

Recall: 4-bit Register



Condensed



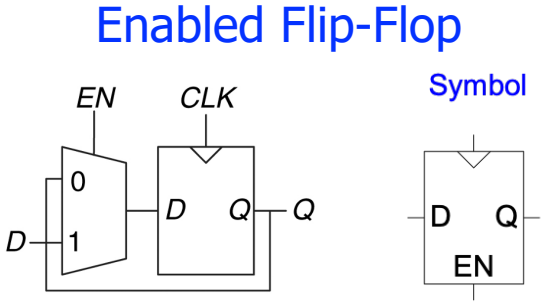
This line represents 4 wires

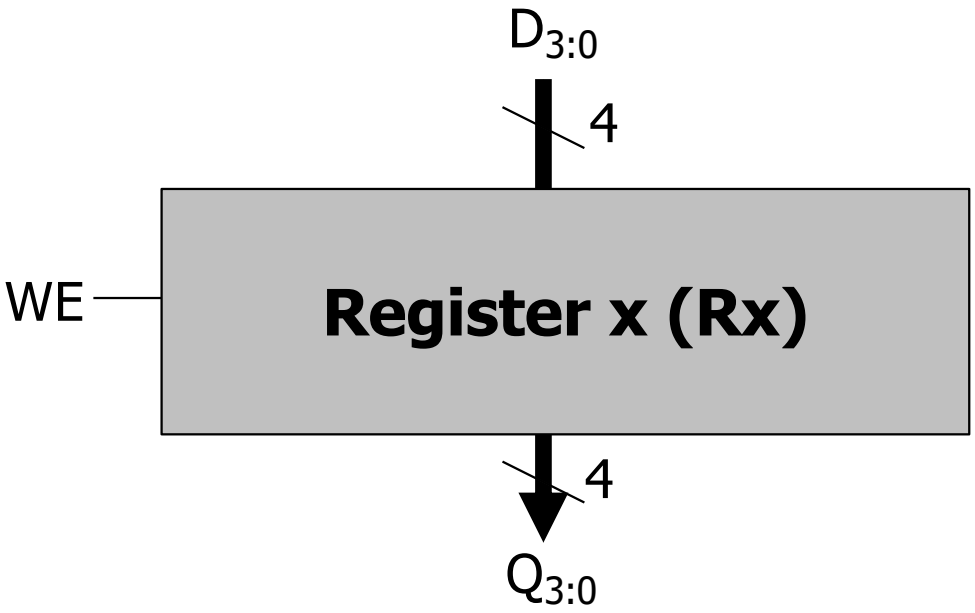
This register stores 4 bits

- 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]

More Realistic Register

- A single WE signal for all flip-flops 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]

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

ARM Register File

Table 6.1 ARM register set

Name	Use
R0	Argument / return value / temporary variable
R1–R3	Argument / temporary variables
R4–R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

QuAC Register File

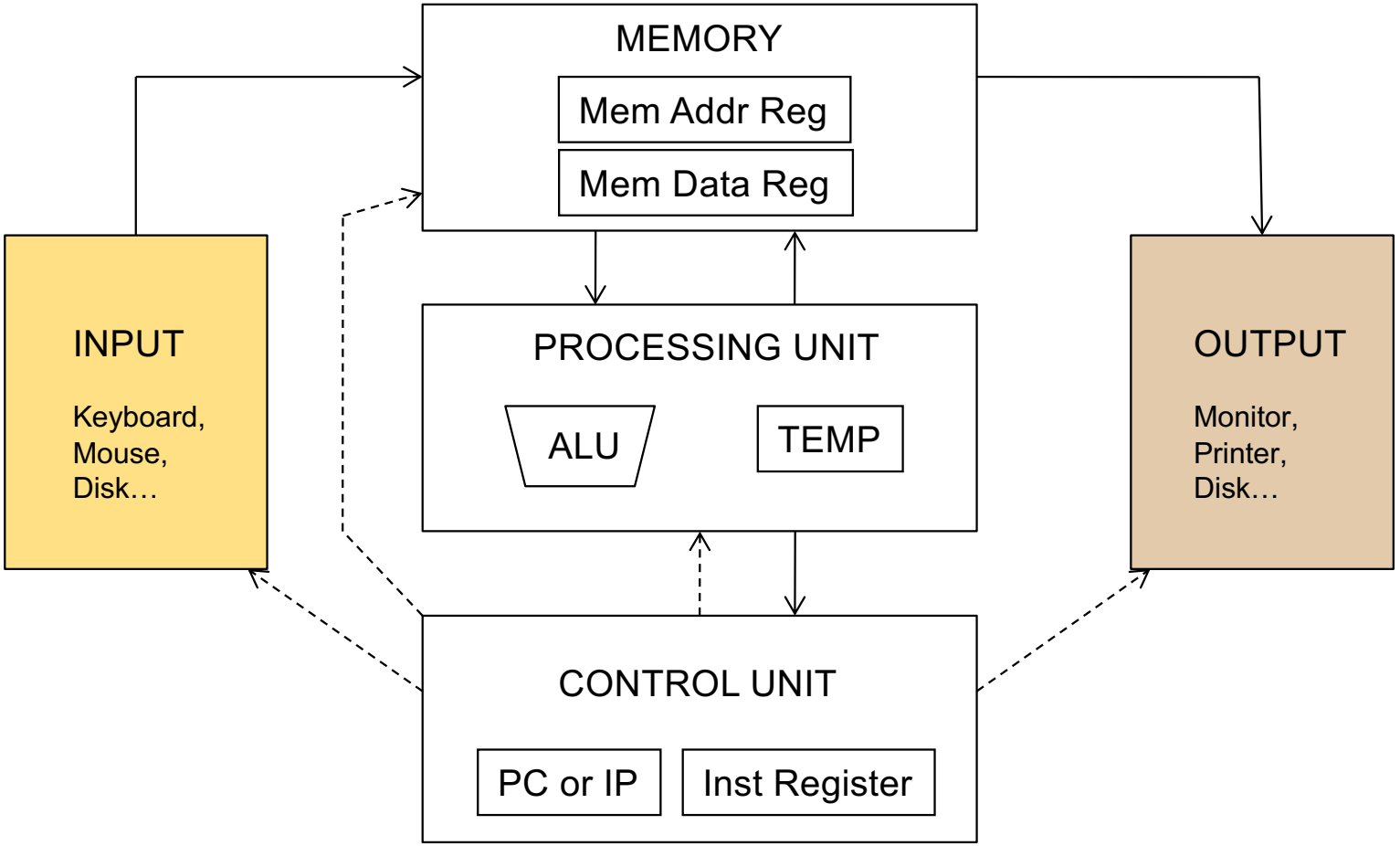
Registers

All registers start initalised to `0x0000`, and are 16-bits wide.

Code	Mnemonic	Meaning	Behaviour
000	`rz`	Zero Register	Always reads as zero, even after being written to.
001	`r1`	Register 1	General purpose register.
010	`r2`	Register 2	General purpose register.
011	`r3`	Register 3	General purpose register.
100	`r4`	Register 4	General purpose register.
101	`fl`	Flag register	See Flags.
110	-	Undefined	Any operation with this register is undefined.
111	`pc`	Program Counter	See Program Counter.

- `rz`, `fl`, and `pc` may also be described as `r0`, `r5`, and `r7` respectively.
- An instruction is allowed to write to `rz`, however the next time an instruction reads `rz` it will still read as `0`.
- `r1`, `r2`, `r3`, and `r4` are the general purpose registers. You may write to them, and they will store that value. Reading from a general purpose register returns the last value written to them.

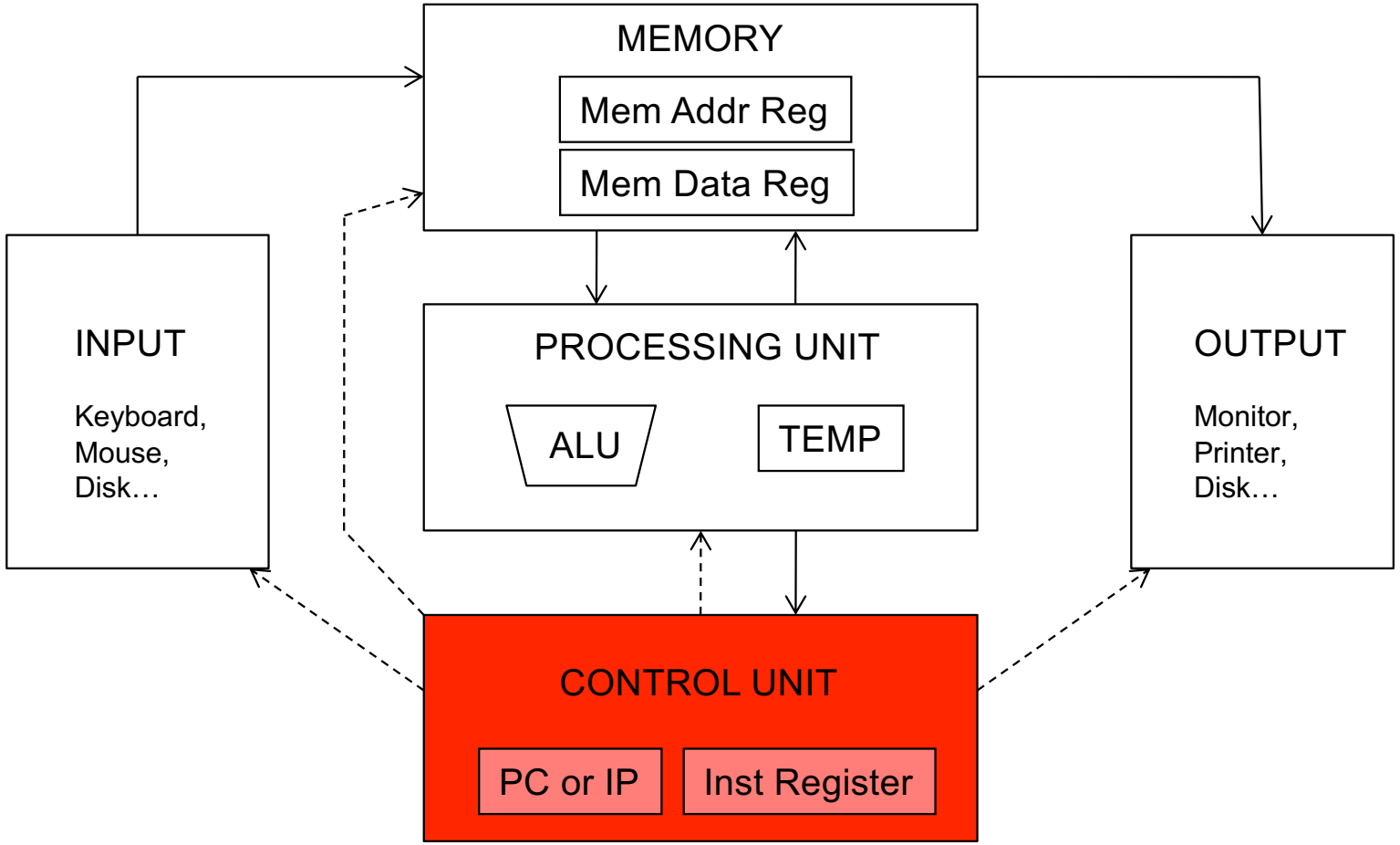
The Von Neumann Model



Input and Output

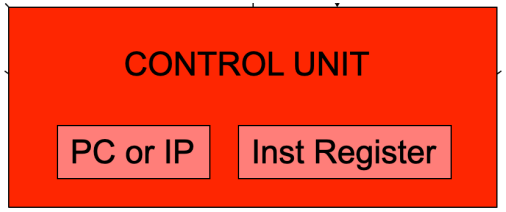
- Enable information to get into and out of a computer
- Many devices can be used for input and output
- They are called peripherals
  - Input
    - Keyboard
    - Mouse
    - Scanner
    - Disks
    - Etc.
  - Output
    - Monitor
    - Printer
    - Disks
    - Etc.

The Von Neumann Model



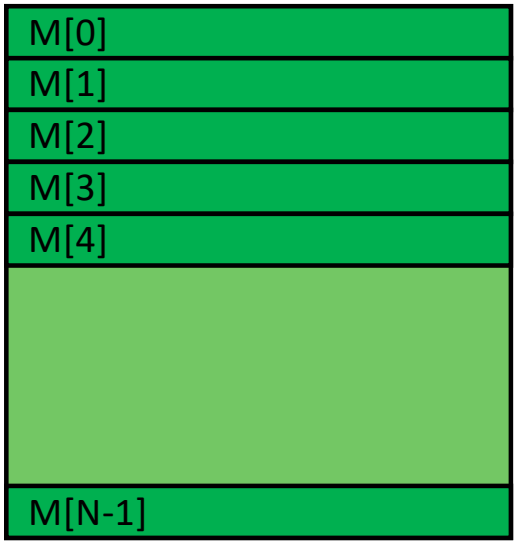
Control Unit

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



- It keeps track of which instruction being processed, via
  - Instruction Register (IR), which contains the instruction
- It also keeps track of which instruction to process next, via
  - Program Counter (PC) or Instruction Pointer (IP), another register that contains the address of the (next) instruction to process

Programmer Visible (Architectural) State



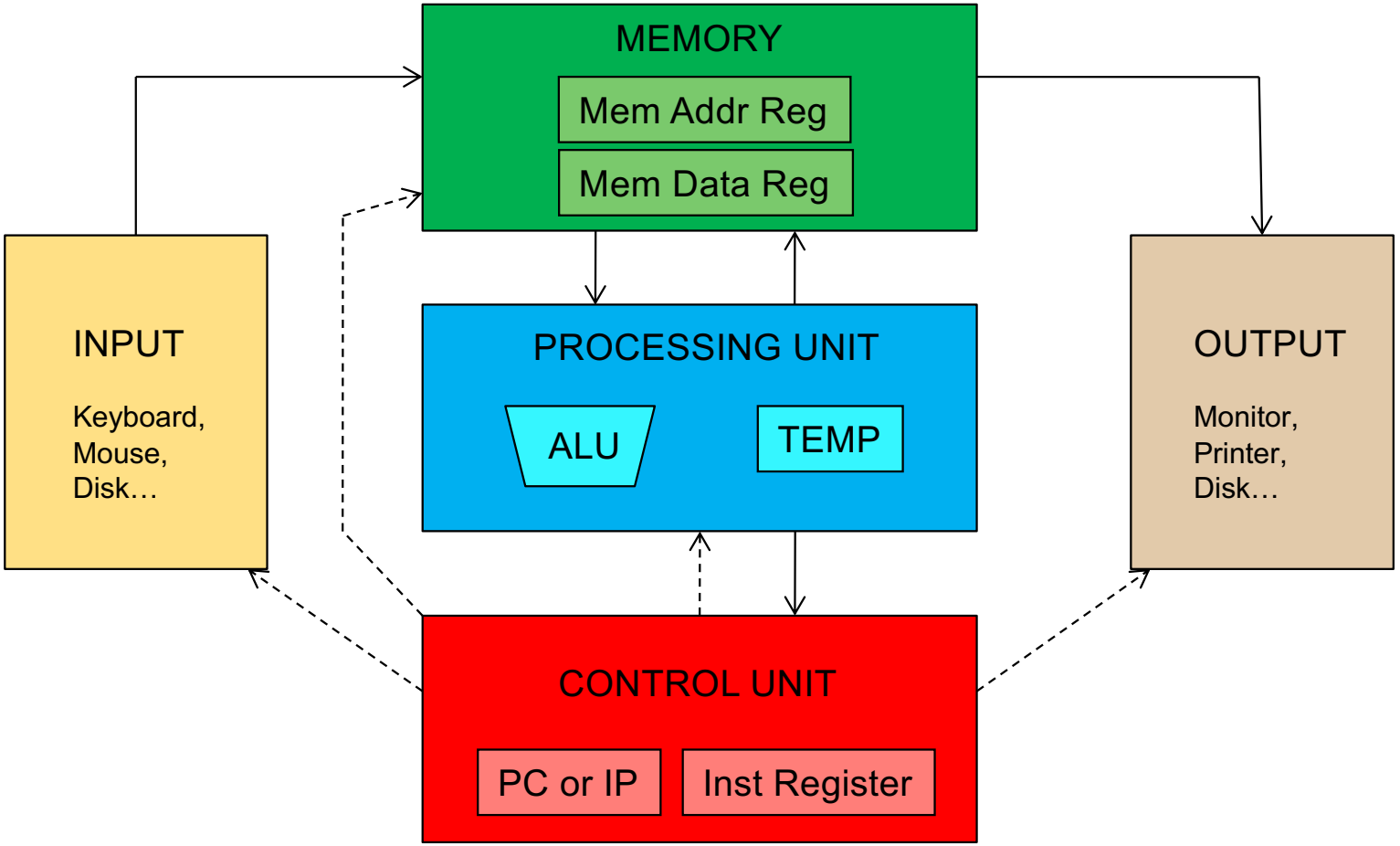
Memory
array of storage locations
indexed by an address

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

Program Counter
memory address
of the current (or next) instruction

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

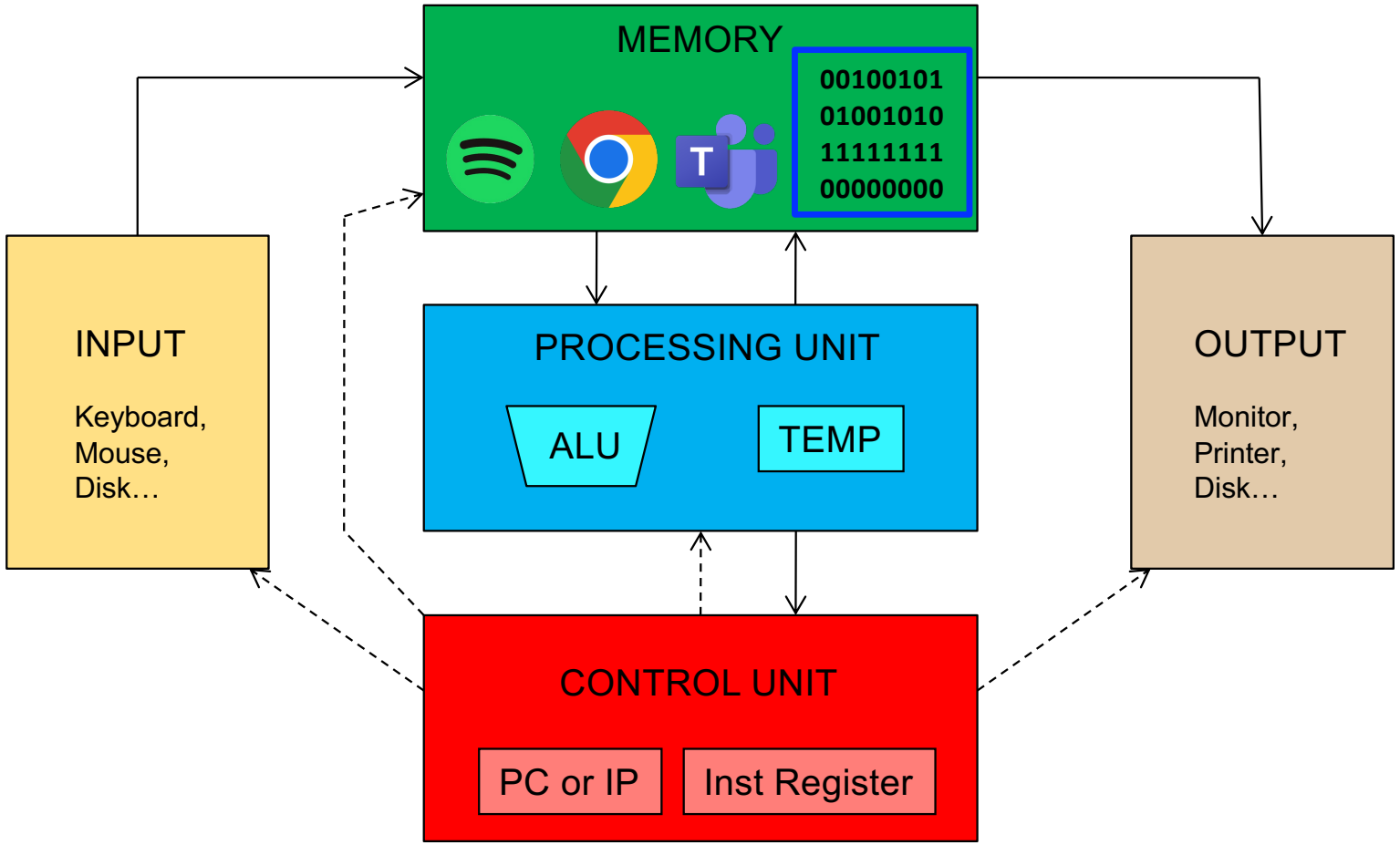
The Von Neumann Model



Von Neumann Model: Two Key Properties

- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:
- Stored program
  - Instructions stored in a linear memory array
  - Memory is unified between instructions and data
    - The interpretation of a stored value depends on the control signals
- Sequential instruction processing
  - One instruction processed (fetched, executed, completed) at a time
  - Program counter (instruction pointer) identifies the current instruction
  - Program counter is advanced sequentially except for control transfer instructions

The Von Neumann Model



Examples of

von Neumann Machines

LC-3: A von Neumann Machine

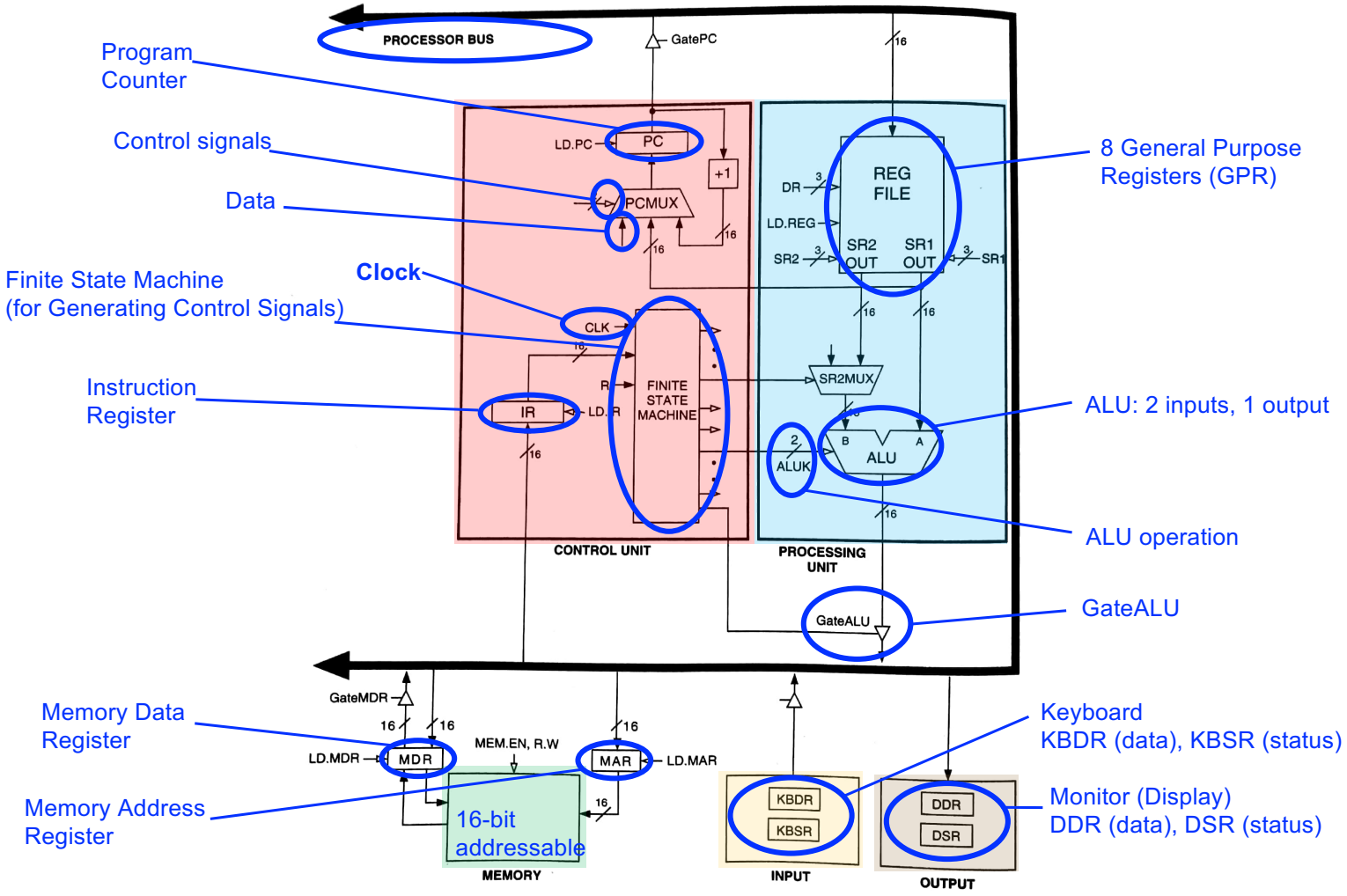
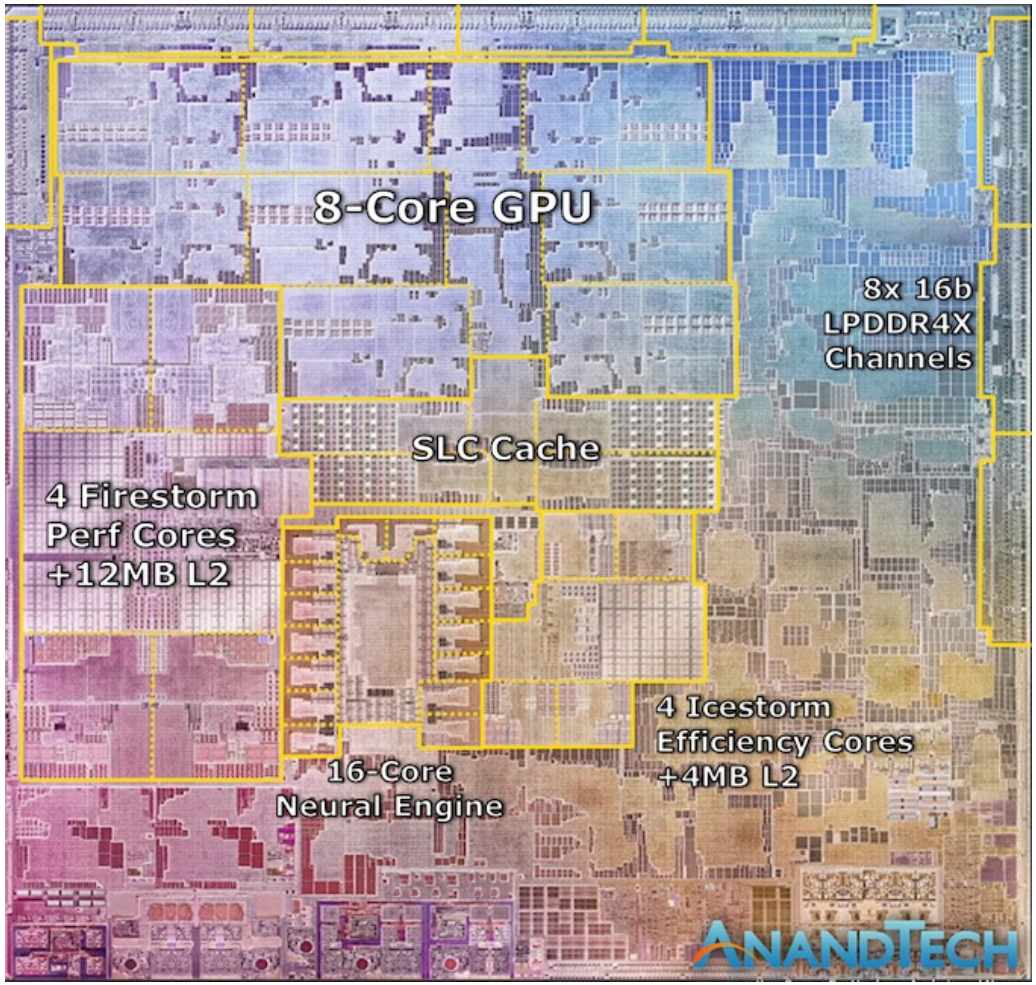


Figure 4.3 The LC-3 as an example of the von Neumann model

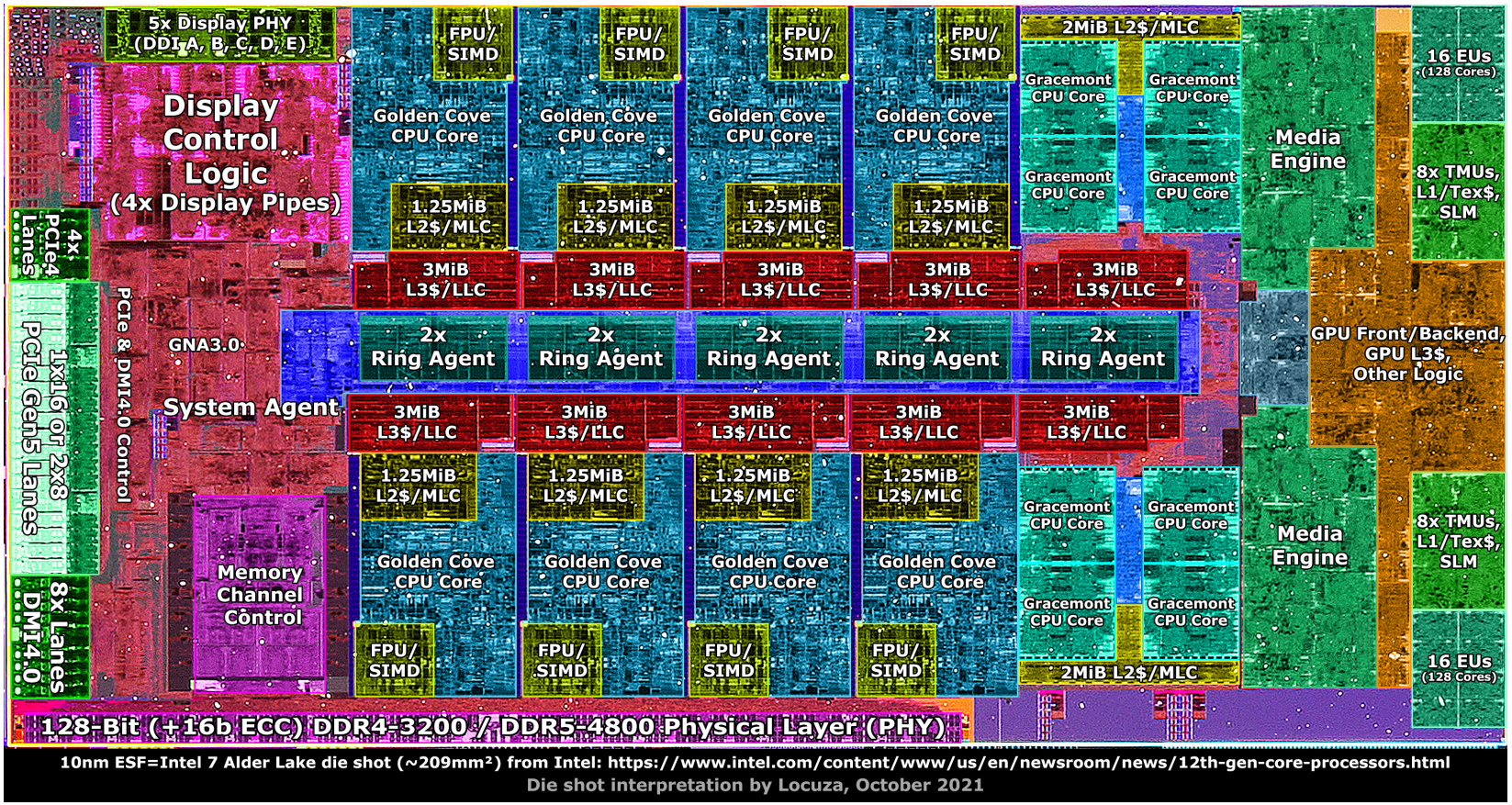
Another Von Neumann Machine



Apple M1, 2021

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested

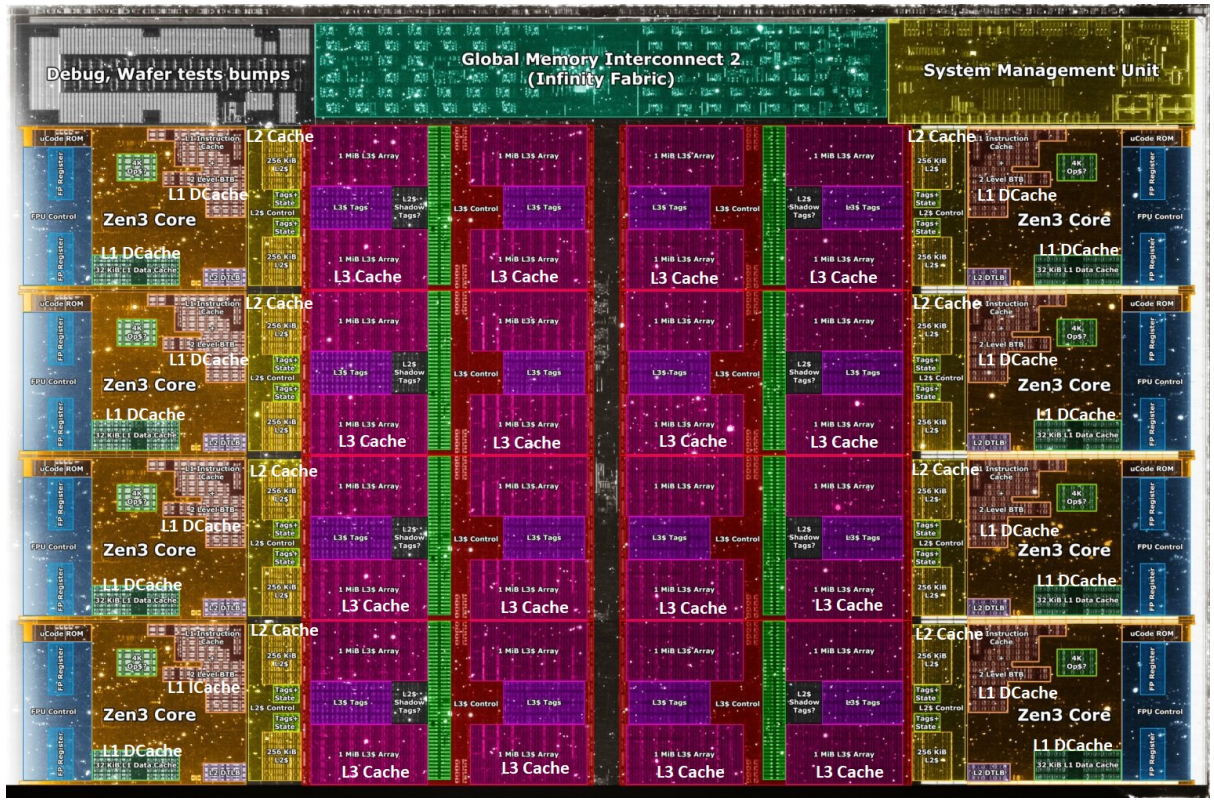
Another Von Neumann Machine



Intel Alder Lake, 2021

Source: https://twitter.com/Locuza_/status/1454152714930331652

Another Von Neumann Machine



AMD Ryzen 5000, 2020

https://wccftech.com/amd-ryzen-5000-zen-3-vermeer-undressed-high-res-die-shots-close-ups-pictured-detailed/

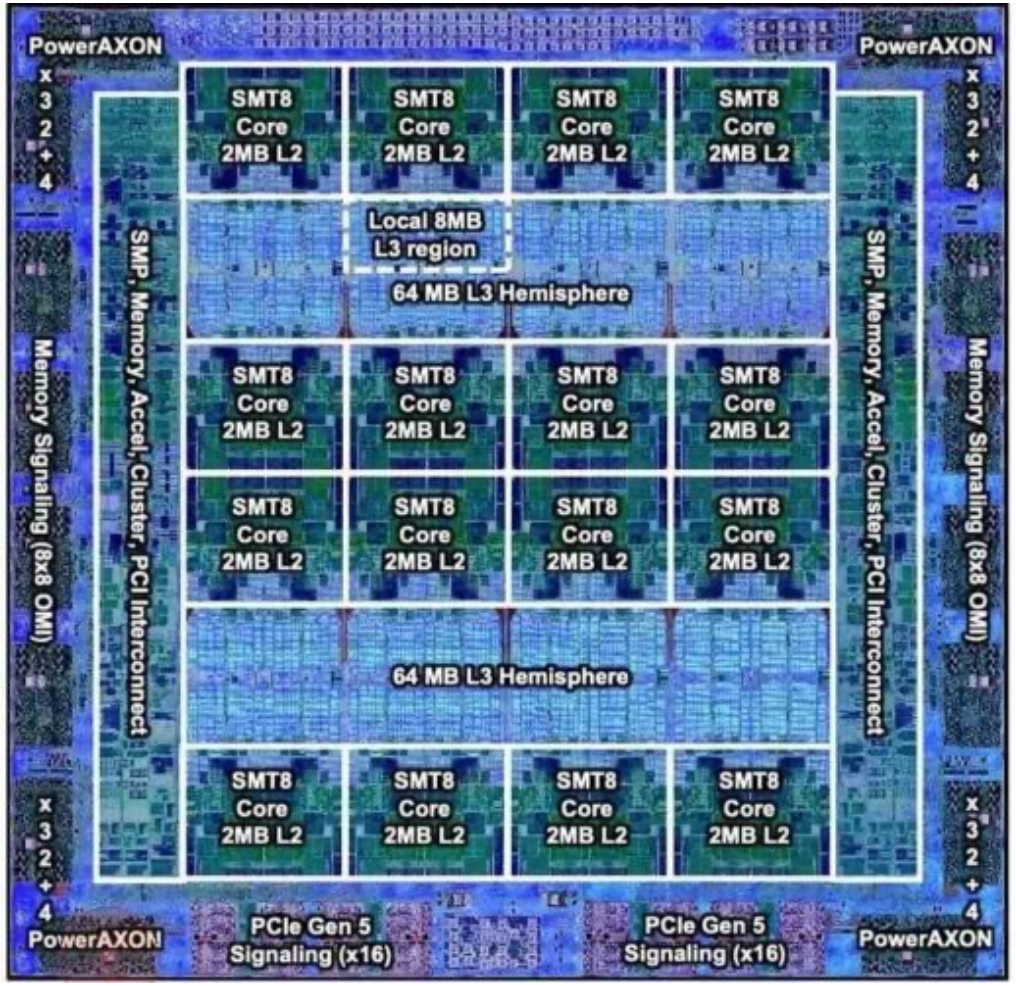
Core Count:
8 cores/16 threads

L1 Caches:
32 KB per core

L2 Caches:
512 KB per core

L3 Cache:
32 MB shared

Another Von Neumann Machine



IBM POWER10, 2020

Cores:

15-16 cores,
8 threads/core

L2 Caches:

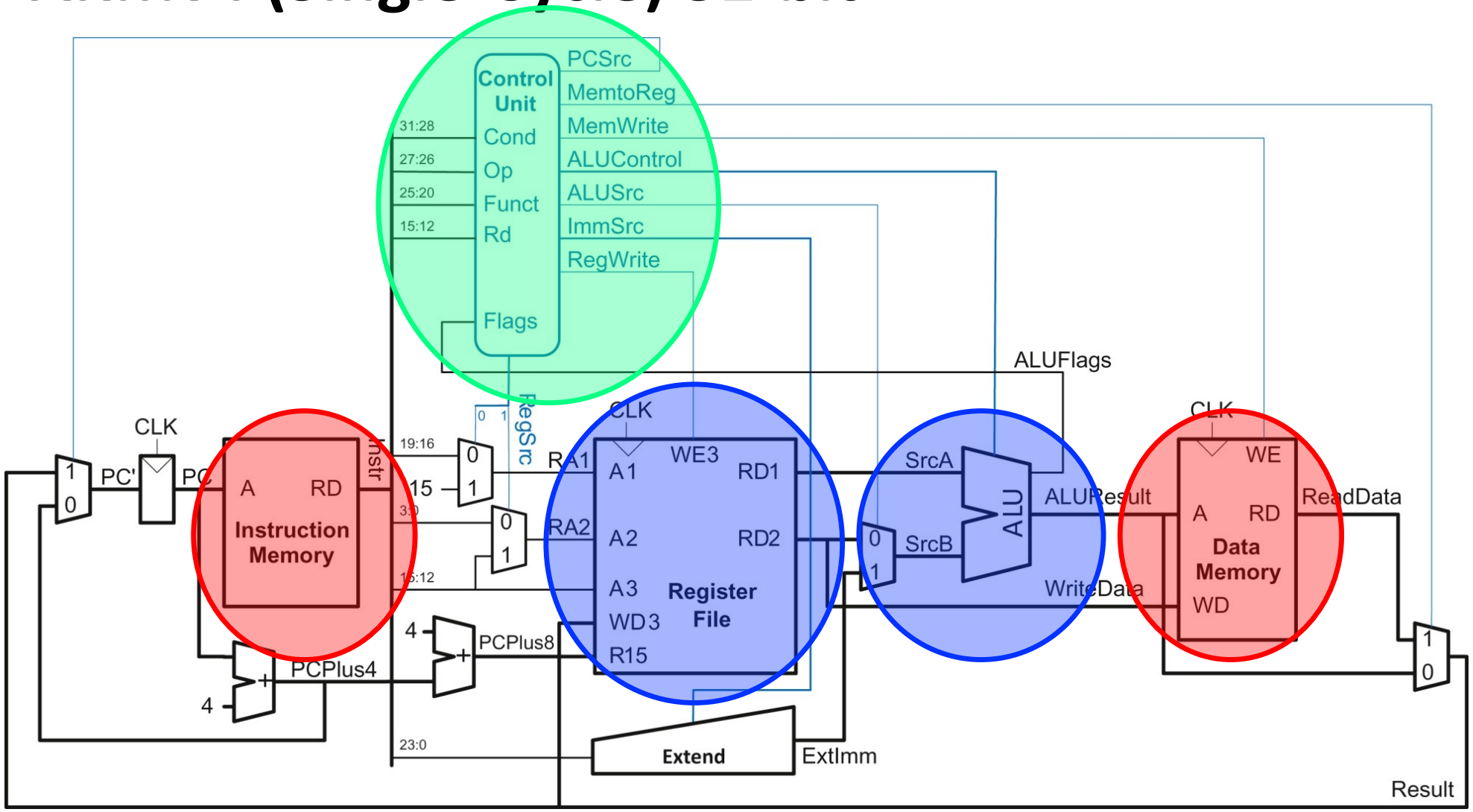
2 MB per core

L3 Cache:

120 MB shared

https://www.it-techblog.de/ibm-power10-prozessor-mehr-speicher-mehr-tempo-mehr-sicherheit/09/2020/

ARMv4 (Single-Cycle) 32-bit



ARMv4 (Multi-Cycle) 32-bit

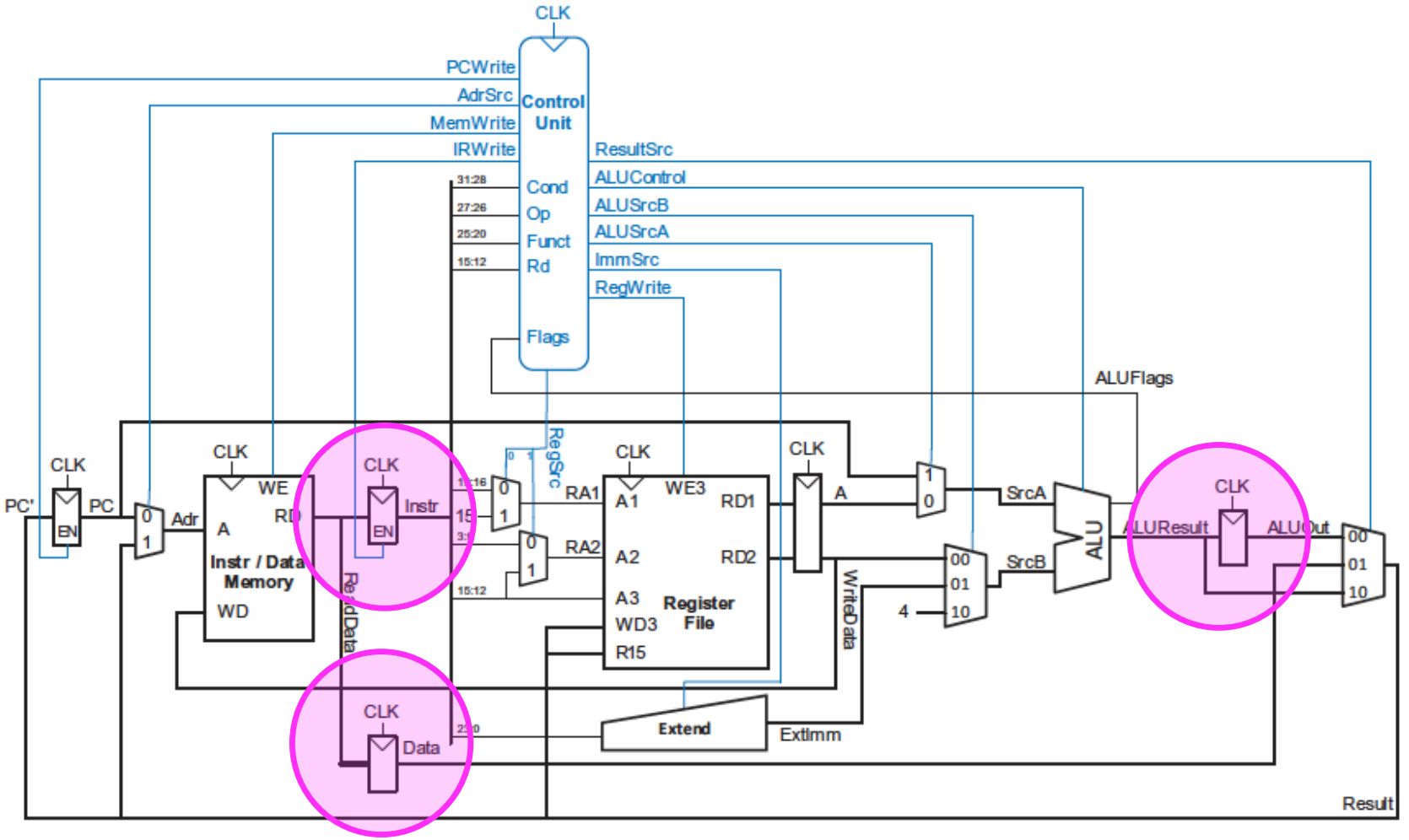
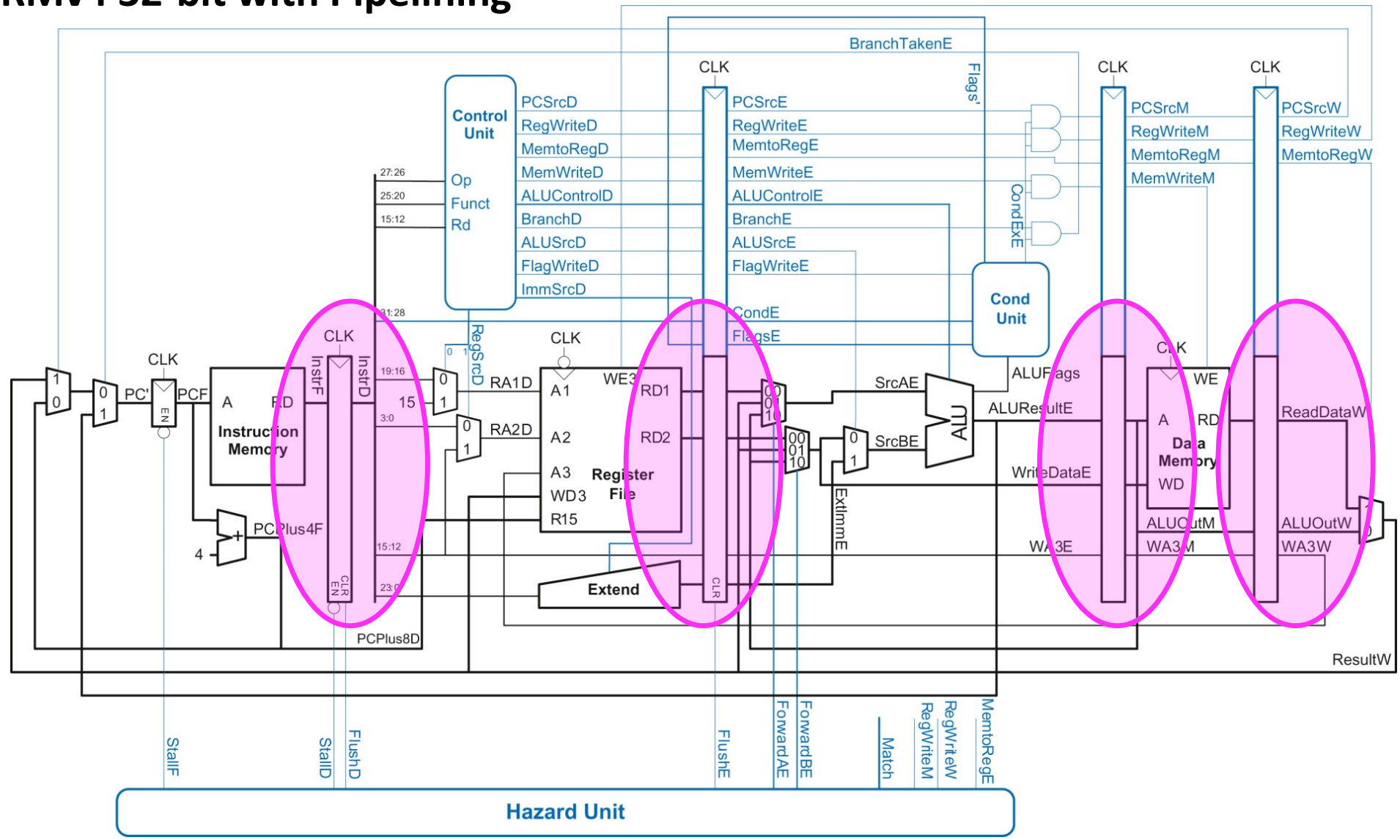
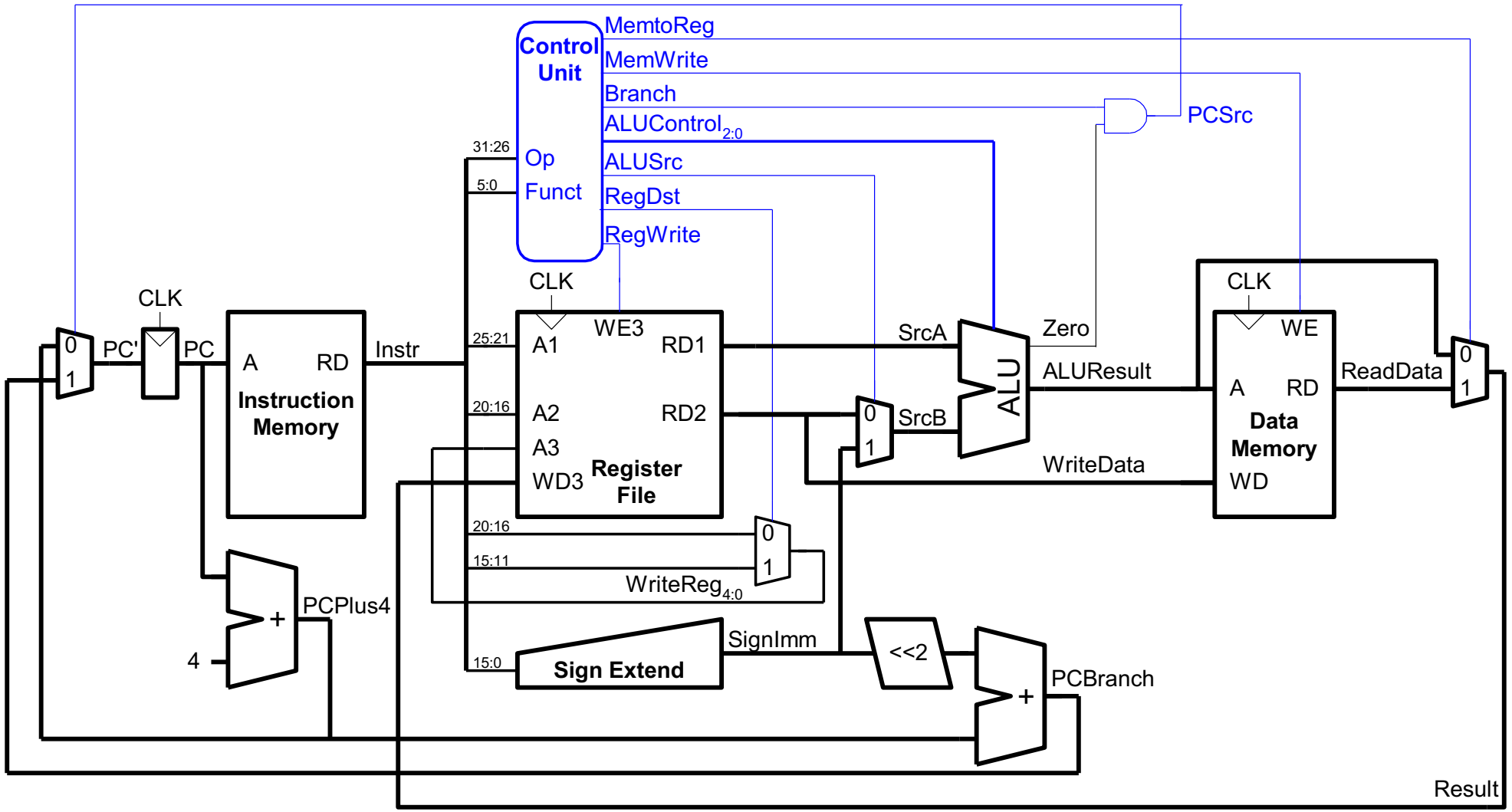


Figure 7.30 Complete multicycle processor

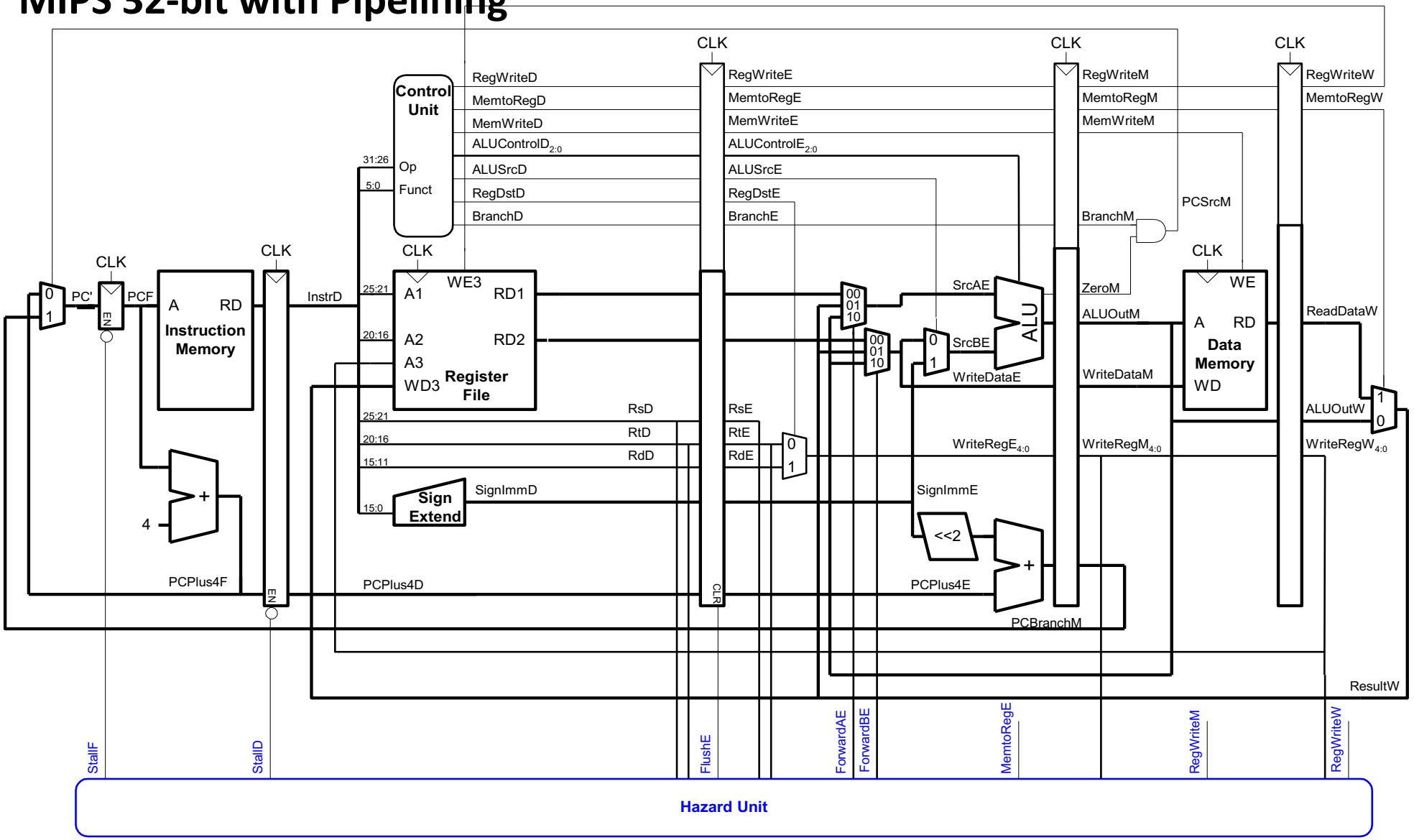
ARMv4 32-bit with Pipelining



MIPS (Single-Cycle) 32-bit



MIPS 32-bit with Pipelining



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
  - 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 QuAC)
  - If byte-addressable memory, the processor increments the PC by the instruction length in bytes (4 in MIPS and ARM)
    - In MIPS the OS typically sets the PC to 0x00400000 (start of a program)

A sample ARM program stored in memory

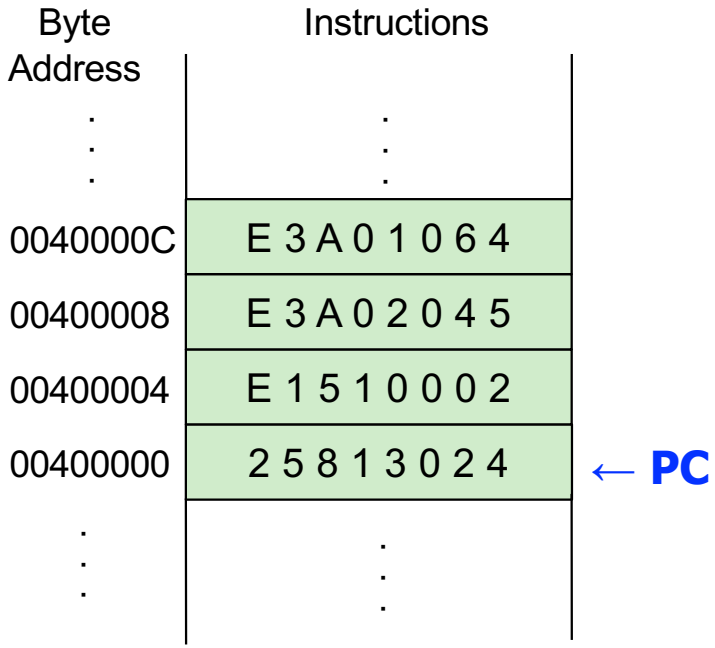
- A sample **ARM** program
  - 4 instructions stored in consecutive words in memory
    - No need to understand the program now. We will get back to it

ARM assembly code

```
MOV   R1,  #100
MOV   R2,  #69
CMP   R1,  R2
STRHS R3,  [R1, #0x24]
```

Machine code (encoded instructions)

```
0xE3A01064
0xE3A02045
0xE1510002
0x25813024
```



A sample program: MIPS Example

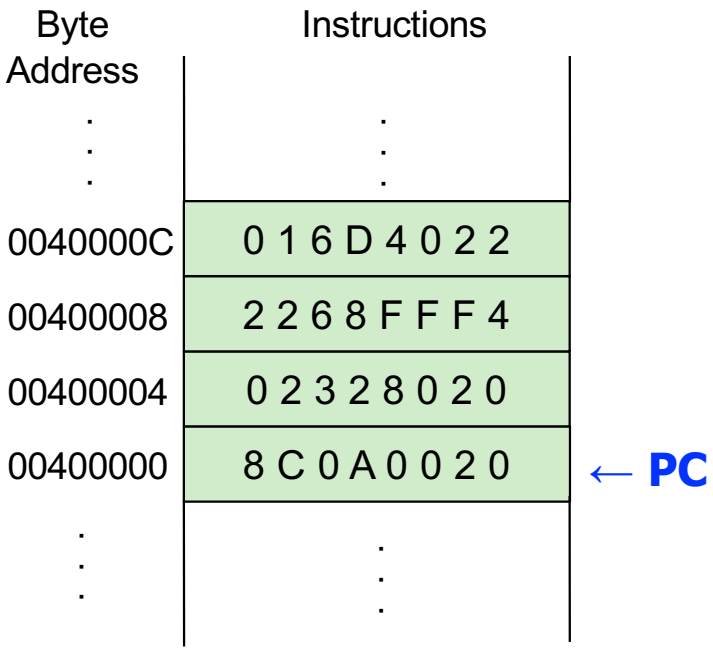
- A sample MIPS program
  - 4 instructions stored in consecutive words in memory
    - No need to understand the program now. We will get back to it

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 ARM (in detail) and (some) MIPS/QuAC/LC-3 instructions
  - Principles are similar in all ISAs (x86, SPARC, RISC-V, …)

The Instruction: Opcode & Operands

- An instruction is made up of two parts
  - Opcode and Operands

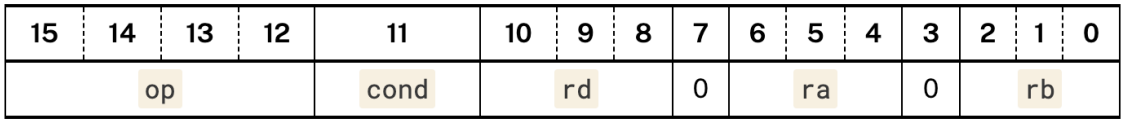
- Opcode specifies what the instruction does
- Operands specify who the instruction is to do it to

- Both are specified in instruction format (or instr. encoding)
  - A MIPS and ARM instructions consists of 32 bits (bits [31:0])
  - QuAC instructions consist of 16 bits (bits [15:0])

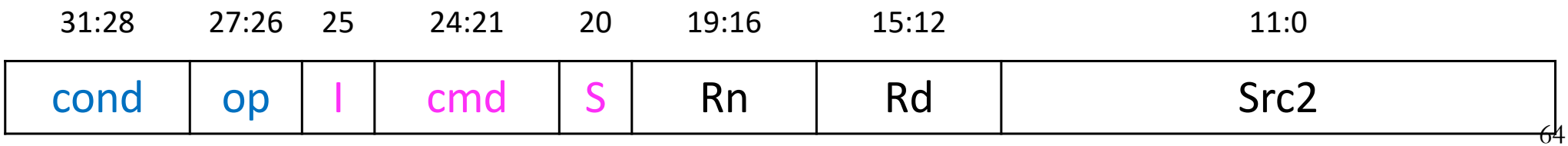
The Instruction: Examples

- **MIPS** example: Bits [31:26] specify the opcode → up to 64 distinct opcodes
  - Bits [25:11] are used to figure out where the **operands** are

- **QuAC** example: Bits [15:12] specify the opcode → up to 16 distinct opcodes
  - Bits [10:0] are used to figure out where the **operands** are



- **ARM** example: Bits [27:26] specify the opcode → up to 4 distinct opcodes
  - Bits [19:0] are used to figure out where the **operands** are



Instruction Types

- There are three main types of instructions
- Operate (data processing) instructions
  - Execute operations in the ALU
- Data movement (memory) instructions
  - Read from or write to memory
- Control flow (branch/jump) instructions
  - Change the sequence of execution (decision making)
- 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
```

ARM registers

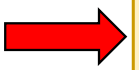
```
b = R1
c = R2
a = R0
```

Registers

All registers start initalised to `0x0000`, and are 16-bits wide.

Code	Mnemonic	Meaning	Behaviour
000	`rz`	Zero Register	Always reads as zero, even after being written to.
001	`r1`	Register 1	General purpose register.
010	`r2`	Register 2	General purpose register.
011	`r3`	Register 3	General purpose register.
100	`r4`	Register 4	General purpose register.
101	`fl`	Flag register	See Flags.
110	-	Undefined	Any operation with this register is undefined.
111	`pc`	Program Counter	See Program Counter.

- `rz`, `fl`, and `pc` may also be described as `r0`, `r5`, and `r7` respectively.
- An instruction is allowed to write to `rz`, however the next time an instruction reads `rz` it will still read as `0`.
- `r1`, `r2`, `r3`, and `r4` are the general purpose registers. You may write to them, and they will store that value. Reading from a general purpose register returns the last value written to them.



QuAC registers

```
b = r1
c = r2
a = r0
```

MIPS registers

```
b = $s1
c = $s2
a = $s0
```

From Assembly to QuAC Machine Code

- Addition

ARM assembly

```
add   r0, r1, r2
```

- Instruction Fields

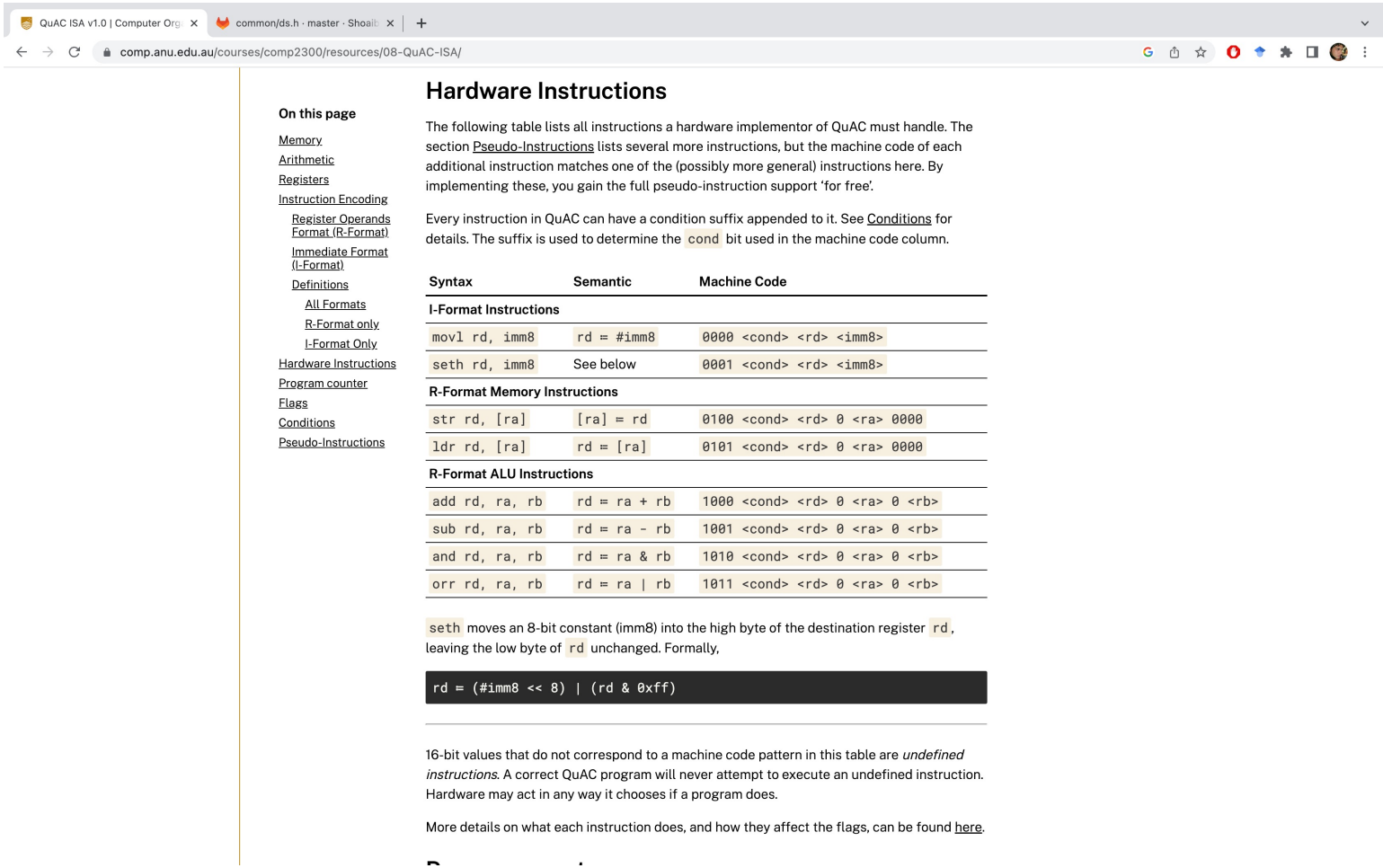
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op			8	cond	0	rd	0	0	ra		1	0	rb		2

- Machine code (Instruction Encoding)

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

- Machine code in short (hexadecimal)
  - 0x 8 0 1 2

QuAC Opcodes



QuAC ISA v1.0 | Computer Org

common/ds.h · master · Shoaib

comp.anu.edu.au/courses/comp2300/resources/08-QuAC-ISA/

On this page

- Memory
- Arithmetic
- Registers
- Instruction Encoding
  - Register Operands Format (R-Format)
  - Immediate Format (I-Format)
  - Definitions
    - All Formats
    - R-Format only
    - I-Format Only
- Hardware Instructions
- Program counter
- Flags
- Conditions
- Pseudo-Instructions

Hardware Instructions

The following table lists all instructions a hardware implementor of QuAC must handle. The section Pseudo-Instructions lists several more instructions, but the machine code of each additional instruction matches one of the (possibly more general) instructions here. By implementing these, you gain the full pseudo-instruction support 'for free'.

Every instruction in QuAC can have a condition suffix appended to it. See Conditions for details. The suffix is used to determine the `cond` bit used in the machine code column.

Syntax	Semantic	Machine Code
I-Format Instructions		
`movl rd, imm8`	`rd = #imm8`	`0000 <cond> <rd> <imm8>`
`seth rd, imm8`	See below	`0001 <cond> <rd> <imm8>`
R-Format Memory Instructions		
`str rd, [ra]`	`[ra] = rd`	`0100 <cond> <rd> 0 <ra> 0000`
`ldr rd, [ra]`	`rd = [ra]`	`0101 <cond> <rd> 0 <ra> 0000`
R-Format ALU Instructions		
`add rd, ra, rb`	`rd = ra + rb`	`1000 <cond> <rd> 0 <ra> 0 <rb>`
`sub rd, ra, rb`	`rd = ra - rb`	`1001 <cond> <rd> 0 <ra> 0 <rb>`
`and rd, ra, rb`	`rd = ra & rb`	`1010 <cond> <rd> 0 <ra> 0 <rb>`
`orr rd, ra, rb`	`rd = ra | rb`	`1011 <cond> <rd> 0 <ra> 0 <rb>`

`seth` moves an 8-bit constant (imm8) into the high byte of the destination register `rd`, leaving the low byte of `rd` unchanged. Formally,

```
rd = (#imm8 << 8) | (rd & 0xff)
```

16-bit values that do not correspond to a machine code pattern in this table are *undefined instructions*. A correct QuAC program will never attempt to execute an undefined instruction. Hardware may act in any way it chooses if a program does.

More details on what each instruction does, and how they affect the flags, can be found here.

From Assembly to ARM Machine Code

- Addition

ARM assembly

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

- Instruction Fields

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op	I	cmd	S	Rn	Rd	Src2

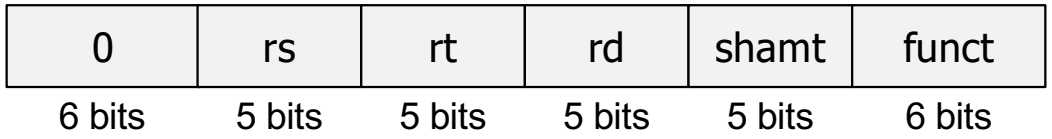
- Machine Code (Instruction Encoding)

31:28	27:26	25	24:21	20	19:16	15:12	11:0
1110	00	0	0100	1	0001	0000	000000000010

- Machine Code in short (hexadecimal)
  - 0x E 0 9 1 0 0 0 1

Instruction Format: R Type in MIPS

- MIPS R-type Instruction Format
  - 3 register operands



  - 0 = opcode
  - rs, rt = source registers
  - rd = destination register
  - shamt = shift amount (only shift operations)
  - funct = operation in R-type instructions

Instruction Format: DP Type in ARM

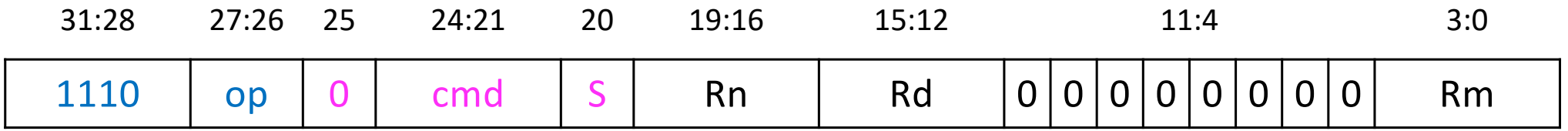
- op = opcode (what the instruction does?)
  - 00 means operate instruction; cmd = 0100 means ADD
  - Some bits are pre-set (**details later**)

- Format (Encoding)

ADD Rd, Rn Rm

ADD R0, R1, R3

Semantics:
Rd = Rn + Rm



- Rn and Rm are source registers
- Rd is destination register

DP – Format Instructions

Read 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

Load Word in MIPS and ARM

- ARM assembly

High-level code

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

ARM assembly

```
LDR  R3, [R0, #8]
```

R3 ← Memory[R0 + 2]

- MIPS assembly

High-level code

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

MIPS assembly

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

$s3 ← Memory[$s0 + 2]

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

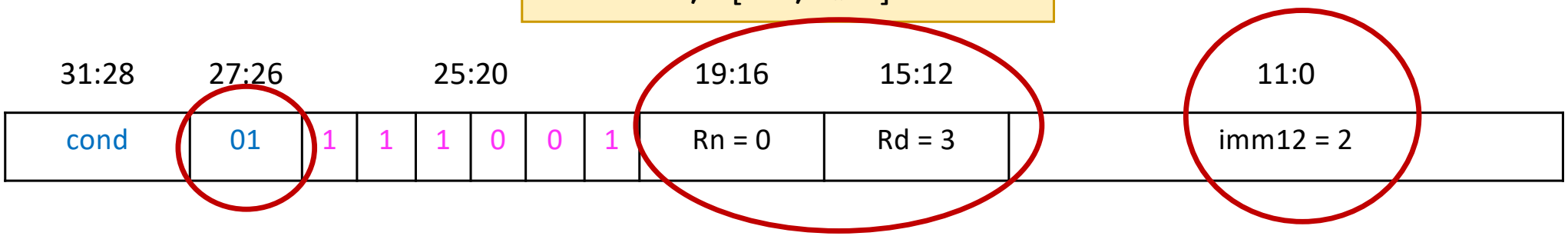
Instruction Format with Immediate

- ARM

Memory – Format Instructions

ARM assembly

```
LDR   R3, [R0, #2]
```

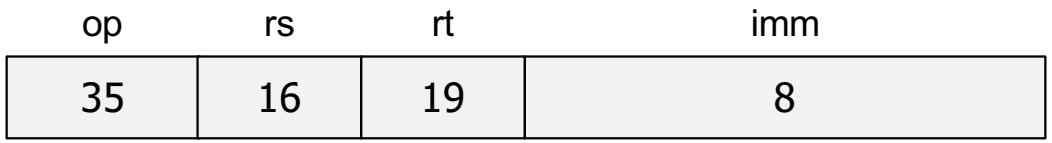


- MIPS

MIPS assembly

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

Field Values



I-Type

QuAC Instruction Formats

> **info**
>
> There's nothing enforcing future instructions fall into these two formats: R-Format and I-Format only *describe* the general pattern existing instructions follow. New instructions could follow an entirely different encoding format.

Register Operands Format (R-Format)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op				cond	rd			0	ra			0	rb		

Immediate Format (I-Format)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op				cond	rd			imm8							

Syntax	Semantic	Machine Code
I-Format Instructions		
`movl rd, imm8`	`rd = #imm8`	`0000 <cond> <rd> <imm8>`
`seth rd, imm8`	See below	`0001 <cond> <rd> <imm8>`
R-Format Memory Instructions		
`str rd, [ra]`	`[ra] = rd`	`0100 <cond> <rd> 0 <ra> 0000`
`ldr rd, [ra]`	`rd = [ra]`	`0101 <cond> <rd> 0 <ra> 0000`
R-Format ALU Instructions		
`add rd, ra, rb`	`rd = ra + rb`	`1000 <cond> <rd> 0 <ra> 0 <rb>`
`sub rd, ra, rb`	`rd = ra - rb`	`1001 <cond> <rd> 0 <ra> 0 <rb>`
`and rd, ra, rb`	`rd = ra & rb`	`1010 <cond> <rd> 0 <ra> 0 <rb>`
`orr rd, ra, rb`	`rd = ra | rb`	`1011 <cond> <rd> 0 <ra> 0 <rb>`

Instruction (Processing) Cycle

How are these instructions executed?

- By using instructions, we can speak the language of the computer
- Thus, we now know how to tell the computer to
  - Execute computations in the ALU by using, for instance, an addition
  - Access operands from memory by using the load word instruction
- But, how are these instructions executed on the computer?
  - The process of executing an instruction is called is the instruction cycle (or, instruction processing cycle)

The Instruction Cycle

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

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

LC-3 Assembly (1)

- We will use LC-3 architecture as an example
- Similar to ARM, MIPS, and QuAC

LC-3 assembly

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

Field Values

OP	DR	SR1			SR2
1	0	1	0	00	2

LC-3 Assembly (2)

- We will use LC-3 architecture as an example
- Similar to ARM, MIPS, and QuAC

High-level code

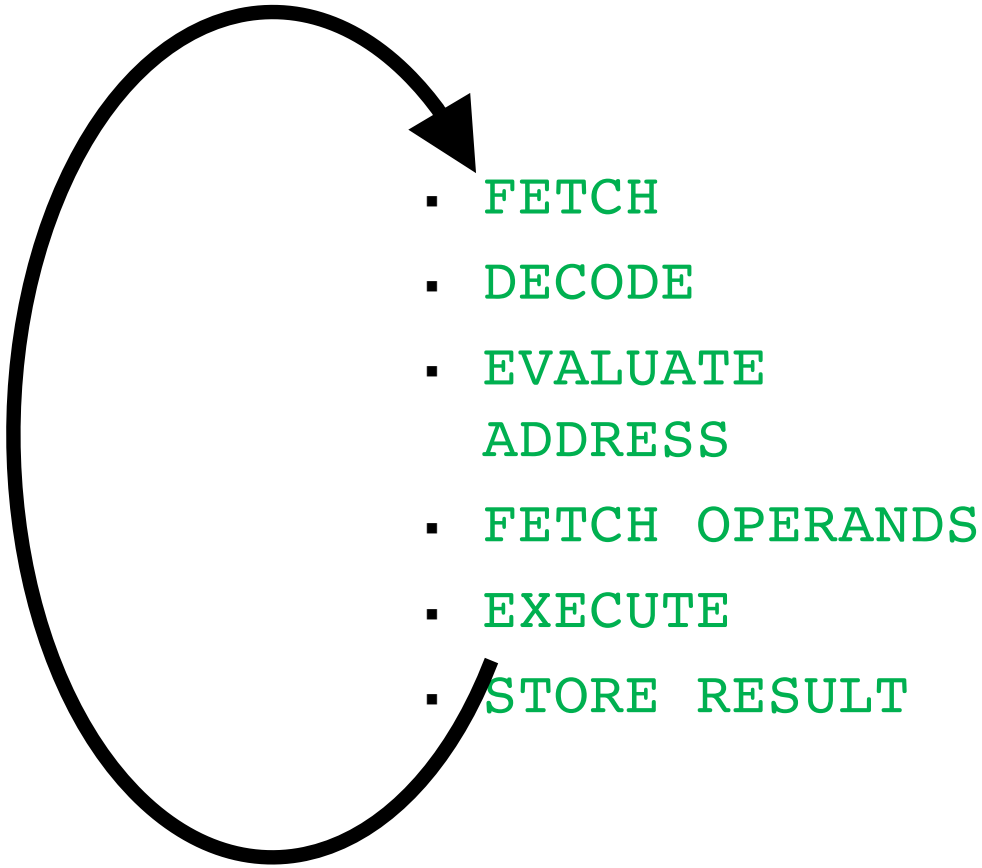
```
a = A[2];
```

LC-3 assembly

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

R3 ← Memory[R0 + 2]

After STORE RESULT, a NEW FETCH



The remaining slides on instruction processing cycle with LC-3 as an example are optional but it strongly recommended that you go through them and ask questions on the forum in the case of questions

Instruction (Processing) Cycle

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

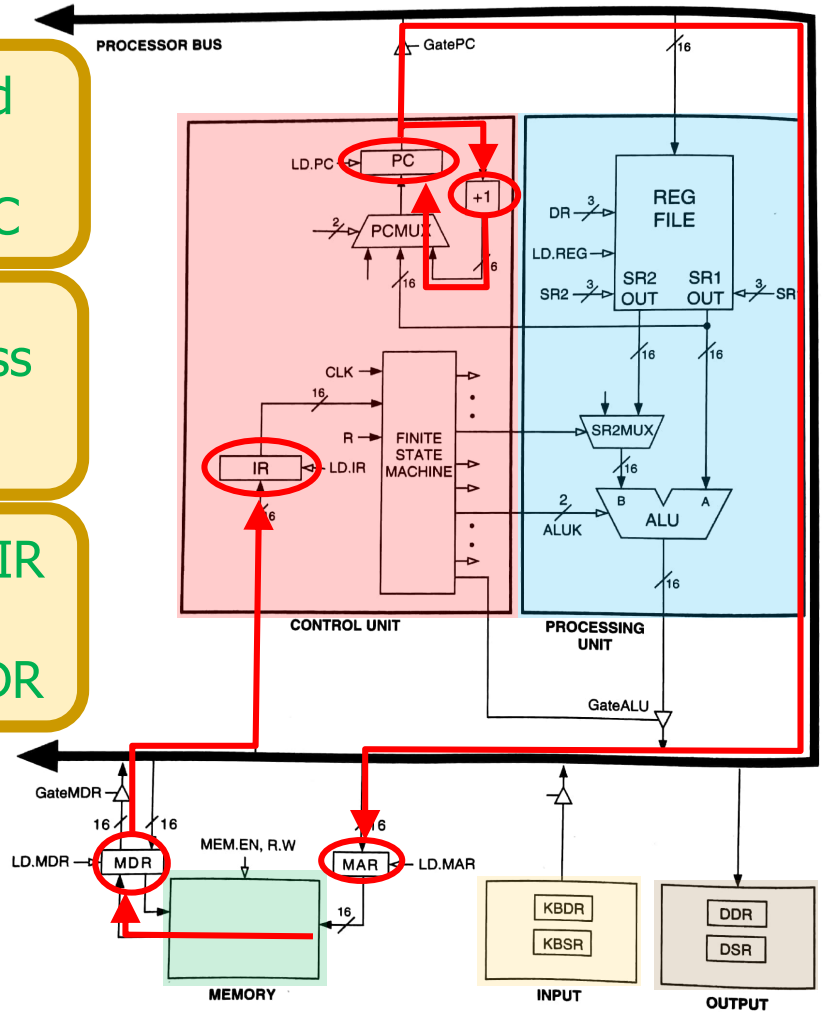


Figure 4.3 The LC-3 as an example of the von Neumann model

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
  - 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

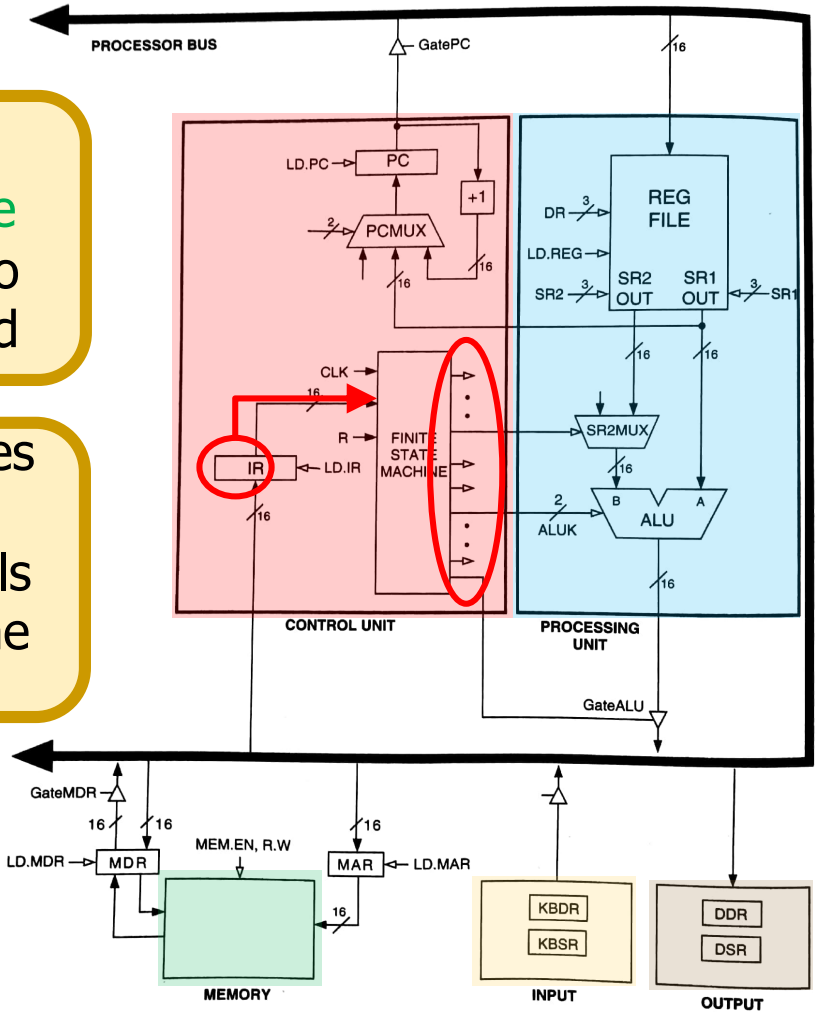


Figure 4.3 The LC-3 as an example of the von Neumann model

EVALUATE ADDRESS

- The EVALUATE ADDRESS phase computes the address of the memory location that is needed to process the instruction
- This phase is necessary in LDR
  - It computes the address of the data word that is to be read from memory
  - By adding an offset to the content of a register
- But not necessary in ADD

EVALUATE ADDRESS in LC-3

LDR calculates the address by adding a register and an immediate

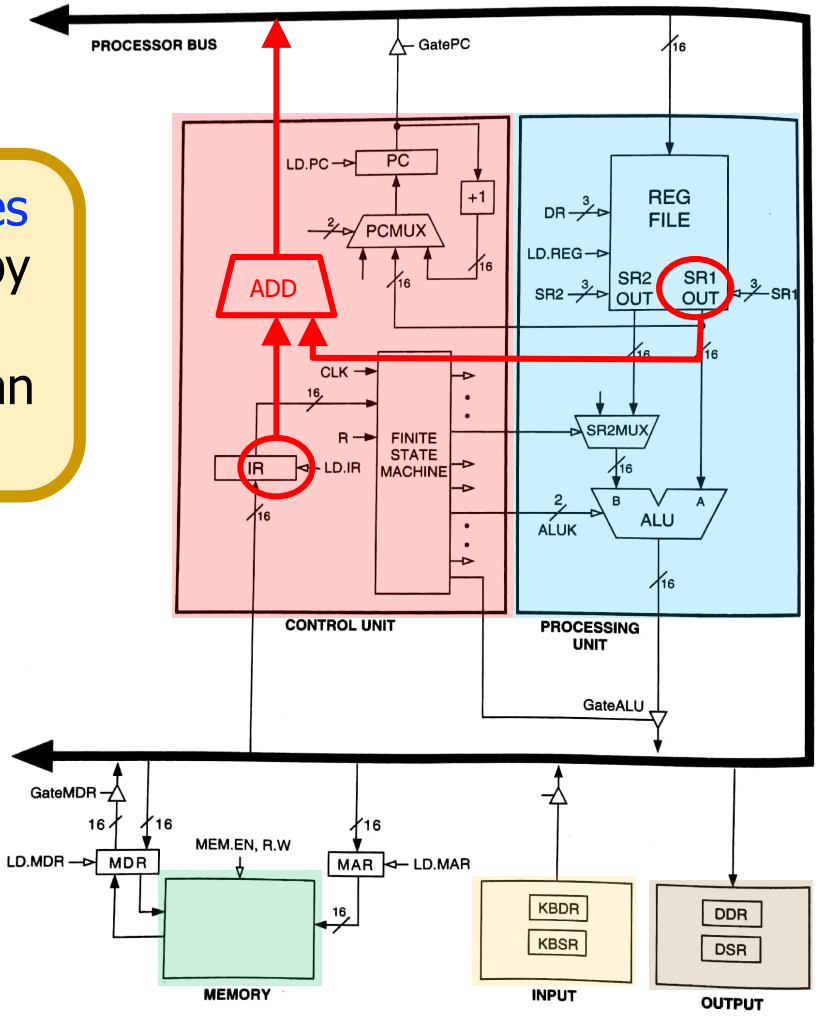


Figure 4.3 The LC-3 as an example of the von Neumann model

FETCH OPERANDS

- The FETCH OPERANDS phase obtains the source operands needed to process the instruction

- In LDR
  - Step 1: Load MAR with the address calculated in EVALUATE ADDRESS
  - Step 2: Read memory, placing source operand in MDR

- In ADD
  - Obtain the source operands from the register file
  - In some microprocessors, operand fetch from register file can be done at the same time the instruction is being decoded

FETCH OPERANDS in LC-3

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

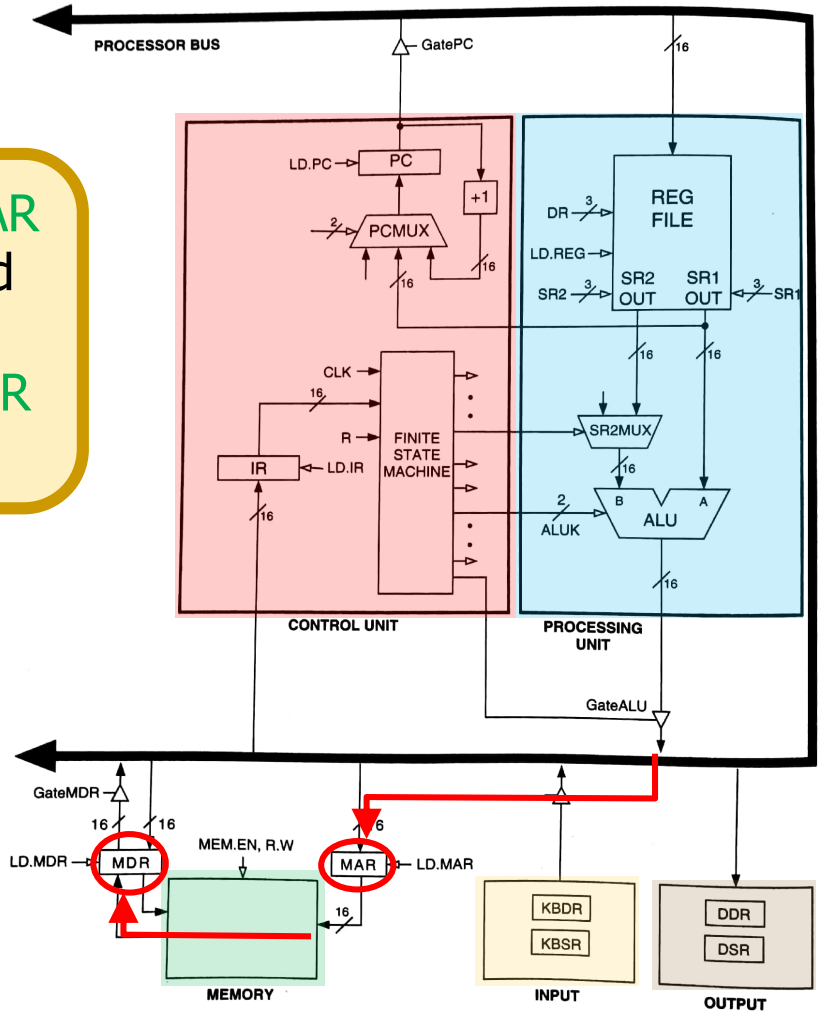


Figure 4.3 The LC-3 as an example of the von Neumann model

EXECUTE

- The EXECUTE phase executes the instruction
  - In ADD, it performs addition in the ALU
  - In XOR, it performs bitwise XOR in the ALU
  - …

EXECUTE in LC-3

ADD adds SR1 and SR2

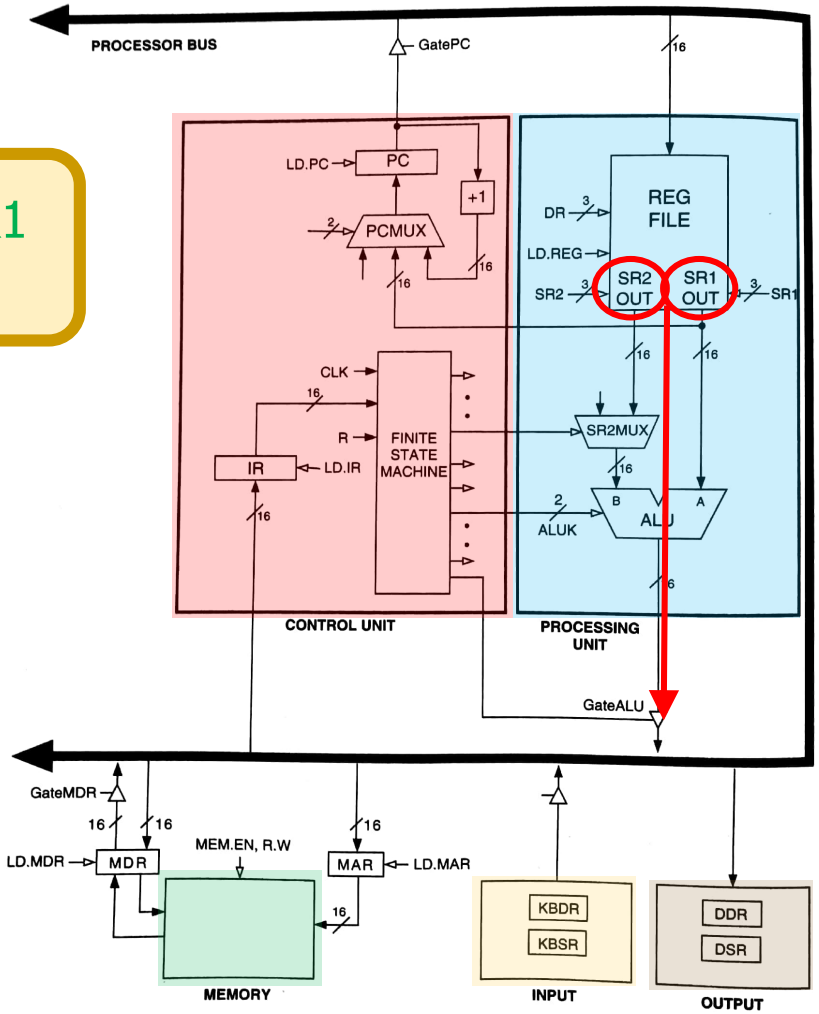


Figure 4.3 The LC-3 as an example of the von Neumann model

STORE RESULT

- The STORE RESULT phase writes the result to the designated destination

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

STORE RESULTS in LC-3

ADD loads ALU Result into DR

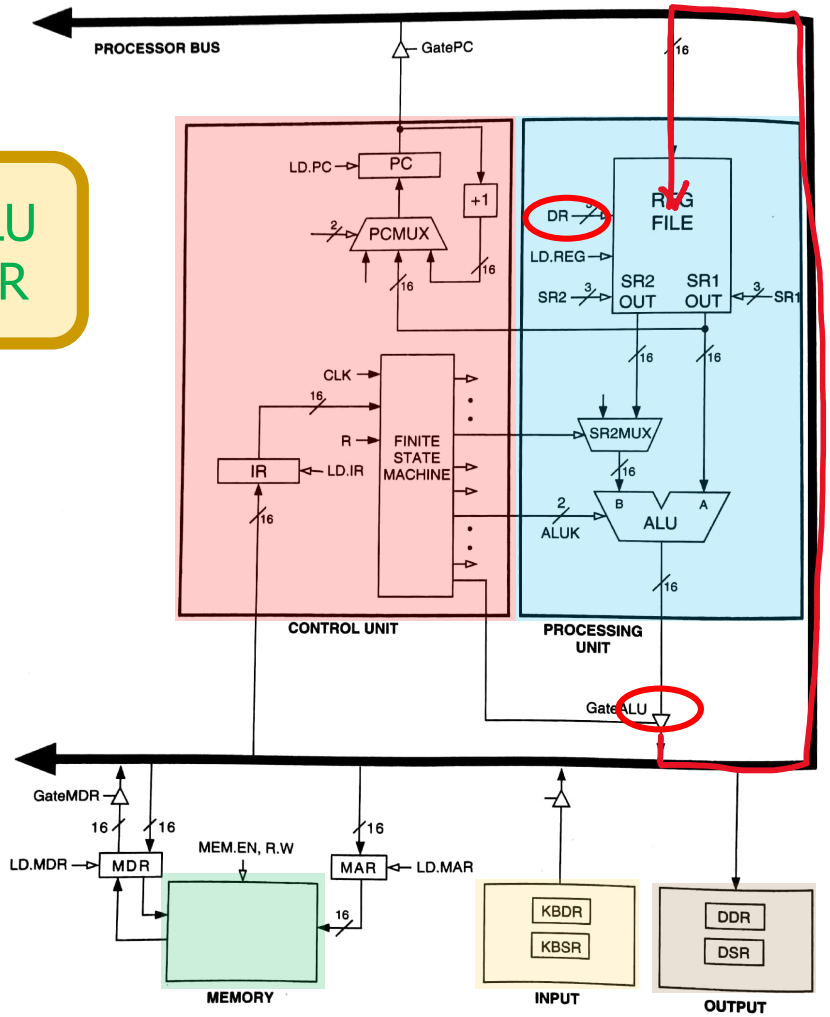


Figure 4.3 The LC-3 as an example of the von Neumann model

STORE RESULTS in LC-3

LDR loads
MDR into DR

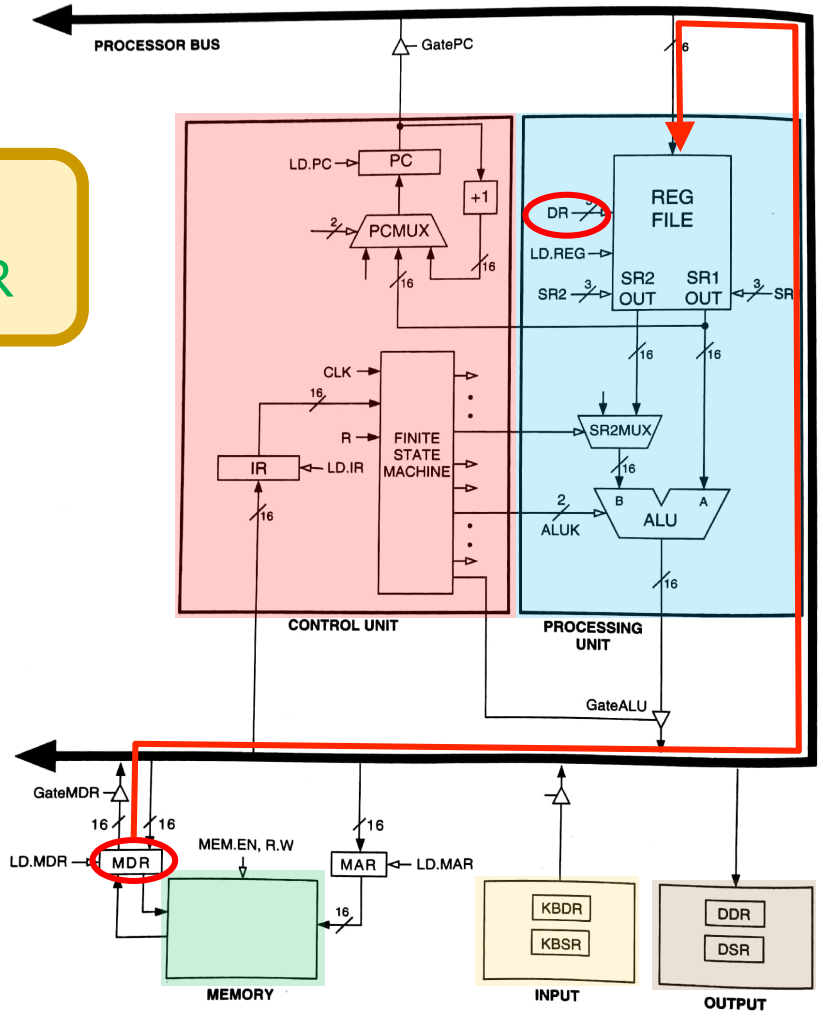


Figure 4.3 The LC-3 as an example of the von Neumann model

The Instruction Cycle

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

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 (Branch)

- **Unconditional** branch or jump (ARM)

```
B   TARGET
```

- **Conditional** branch or jump (ARM)

```
BEQ   TARGET
```

```
BNE   TARGET
```

- These instructions are encoded using a special branch format in ARM ISA
  - LC-3 has a jump instruction that can load a register into PC
- Let's see

PC UPDATE in LC-3

JMP loads
SR1 into PC

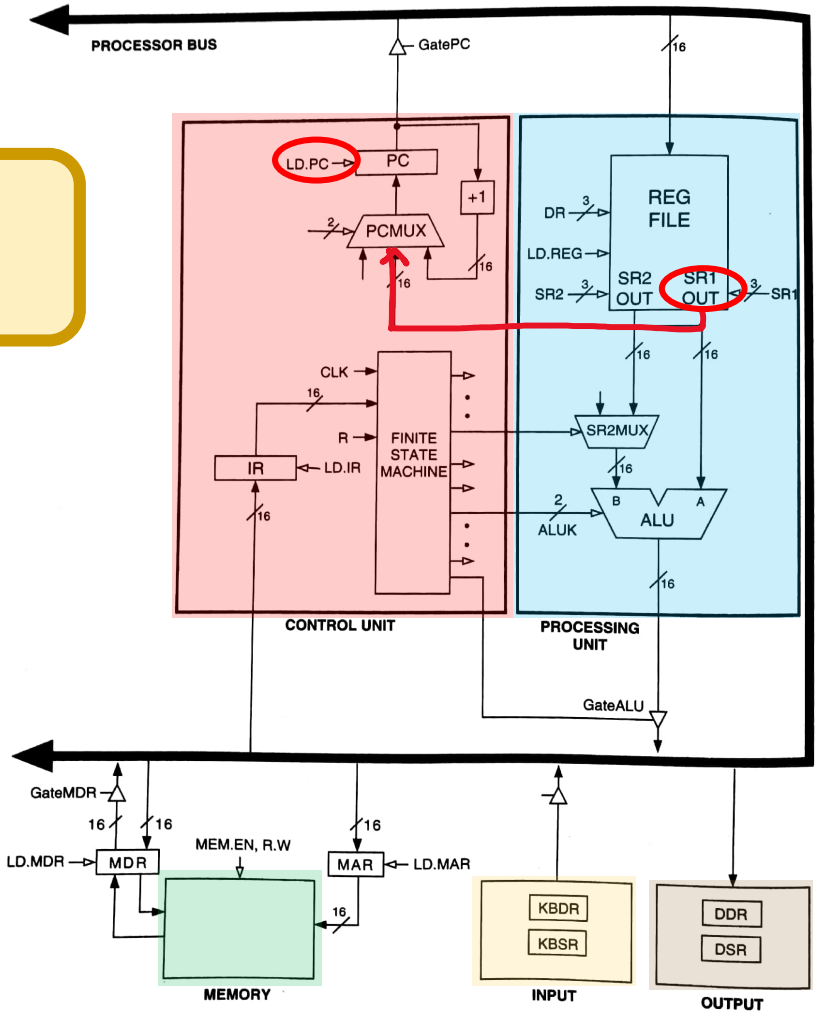


Figure 4.3 The LC-3 as an example of the von Neumann model

Control (FSM) of the Instruction Cycle

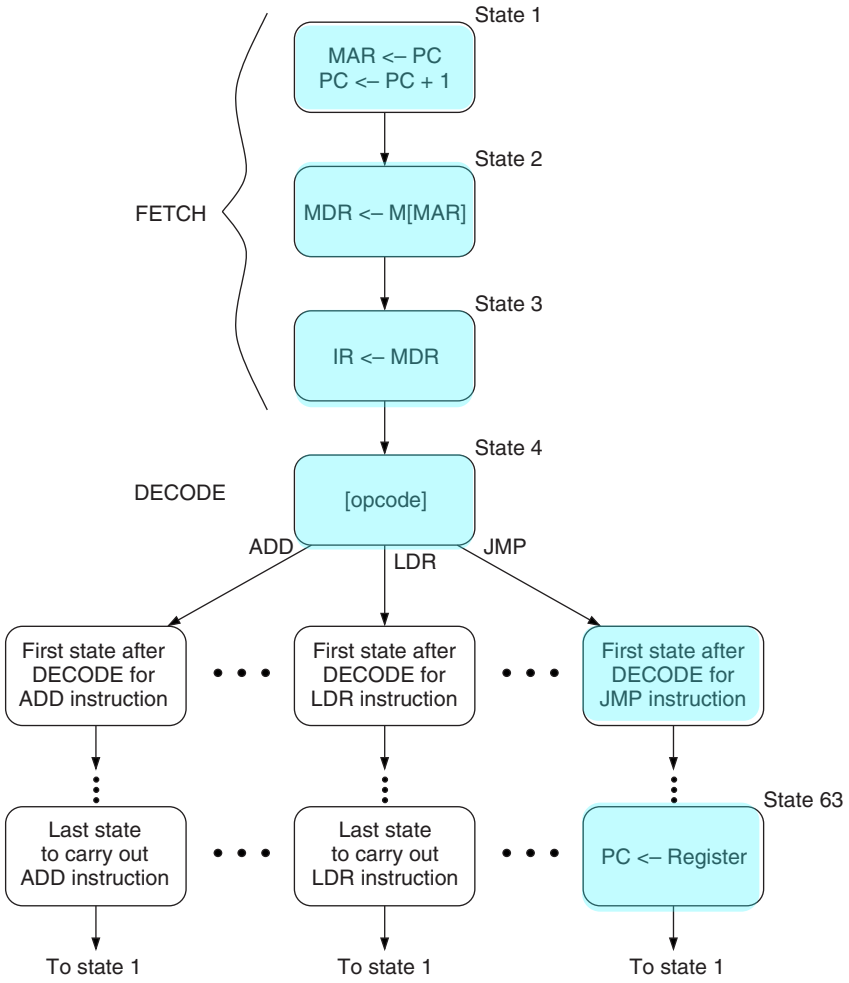


Figure 4.4 An abbreviated state diagram of the LC-3

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

This is an FSM Controlling the LC-3 Processor

The Instruction Cycle

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

The Instruction Cycle: Things to Note

- Not all instructions need all phases
- The ordering of phases in not set in stone
- Some phases can be grouped as one
- Some structures may not be needed in a different microarchitecture
- Microarchitecture “style” dictates many details (week 6)

The Instruction Cycle: Things to Note

- What we have seen is a multi-cycle CPU
  - Each instruction takes multiple cycles to complete

- Labs 4 – 6 + the first assignment asks you to build a single-cycle CPU
  - The entire instruction must finish in one cycle
  - Contrast with multi-cycle as you build

- We Will cover both single-cycle and multi-cycle ARM CPUs next week

We Were Here

ARM and QuAC Instruction Set Architectures (ISAs)

We **really** care about ARM (**Chapter 6** of H&H + Assignment 2) and QuAC (Assignment 1)

Recall

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 Instruction Set

- It defines opcodes, data types, and addressing modes

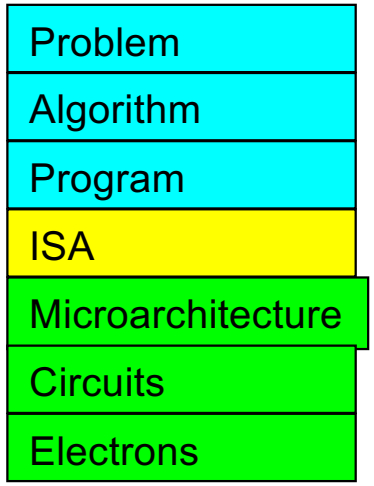
  = Formulas for figuring out operands
  Register, Immediate, Base + Offset

- ADD and LDR have been our first examples

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 (ARM: 2^{32}, MIPS: 2^{32})
    - Addressability (ARM: 8 bits, MIPS: 8 bits, QuAC: 16 bits)
    - Word- or Byte-addressable
  - The register set
    - R0 to R15 in ARM
    - 32 registers in MIPS
  - The instruction set
    - Opcodes
    - Operands
    - Addressing modes
    - Length and format of instructions



Recall: Instruction Types

- There are three main types of instructions
- Operate (data processing) instructions
  - Execute operations in the ALU
- Data movement (memory) instructions
  - Read from or write to memory
- Control flow (branch/jump) instructions
  - Change the sequence of execution (decision making)
- Let us start with some example instructions

Data Processing Instructions

ARM Data Processing (DP) Instructions

- a = b + c – d
  - We can use two ARM instructions to do the computation

```
ADD  t,  b,  c
SUB  a,  t,  d
```

- ADD and SUB are instruction **mnemonics**
- Instructions operate on operands `(a, b, c)`
- Computers operate on binary data not variable names
  - We need to specify the physical location of operands
    - We have registers, memory, constants in instructions

Registers as Operands

- Instructions need **fast access** to operands, but **memory** is **slow**
  - Keep a small set of registers close to the CPU in a register file
  - ARM architecture uses 16 registers
  - 32-bit architecture means 32-bit registers
- **a = b + c - d**
  - R0 = **a**, R1 = **b**, R2 = **c**, R3 = **d**, R4 = **t**

Mapping is chosen by human, or a tool called assembler that translates high-level code to assembly

```
ADD  t,  b,  c
SUB  a,  t,  d
```

```
ADD  R4,  R1,  R2
SUB  R0,  R4,  R3
```

Source/Destination Operand

- Instructions operate on one or more **source** operands and store the result after execution in a **destination** operand

```
ADD  R4,  R1,  R2
SUB  R0,  R4,  R3
```

- R1 and R2 are the **source operands** for the ADD instruction
- R4 is the **destination operand** for the ADD instruction

Another Example

- **a = b – c**
- **f = (g + h) – (i + j)**
  - Variables a – c are held in registers R0 – R2 and f – j are held in registers R3 – R7

```
SUB  R0,  R1,  R2
ADD  R8,  R4,  R5
ADD  R9,  R6,  R7
SUB  R3,  R8,  R9
```

Design Principle

- Regularity leads to simpler hardware
  - Instructions with a consistent number of operands (2 sources, 1 destination) are easier to encode and handle in hardware.

Design Principle

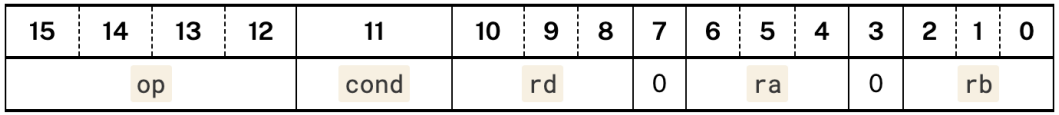
- Regularity leads to simpler hardware
  - Instructions with a consistent number of operands (2 sources, 1 destination) are easier to encode and handle in hardware.

Design Principle

- Regularity leads to simpler hardware
  - Instru of operands (2 source code and handl

> **info**
>
> There's nothing enforcing future instructions fall into these two formats: R-Format and I-Format only *describe* the general pattern existing instructions follow. New instructions could follow an entirely different encoding format.

Register Operands Format (R-Format)



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op				cond	rd			0	ra			0	rb		

The Register Set (File)

- ARM defines 16 *architectural* registers
- The register set is part of the ISA specification
- R0 – R12 are used for storing variables
- R13 – R15 have special uses

Design Principle

- Smaller is Faster
  - Reading data from a small register file is faster than reading from a large file

Constant & Immediate in Instruction

- ARM instructions can use constant or immediate operands

  Fact: 98% of all the constants in a program would fit in 13 bits

- The value is available immediately from the instruction
  - Advantage: No register or memory access
  - Disadvantage: Immediate can be 8 – 12 bits because **limited bits in the encoding (instruction format)**

- In the following example, assume R7 = **a**, R8 = **b**

High-Level code

```
a = a + 4
b = a – 12
```

ARM Assembly Code

```
ADD   R7,   R7,   #4
SUB   R8,   R7,   #0xC
```

Design Principle

- Good design demands good compromises
  - To encode immediate in instructions in QuAC we need a new format
  - Same with ARM (encoding is more complex)

Design Principle

- Good design demands good compr
  - To enc e need to move w format.

> **info**
>
> There's nothing enforcing future instructions fall into these two formats: R-Format and I-Format only *describe* the general pattern existing instructions follow. New instructions could follow an entirely different encoding format.

Register Operands Format (R-Format)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op				cond	rd			0	ra			0	rb		

Immediate Format (I-Format)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op				cond	rd			imm8							

MOV Instruction

- **MOV** is a useful instruction for initializing register values

- MOV can also take a register source operand
  - MOV R1, R7 copies the contents of register R7 into R1

  - In the following example, assume R4 = **i**, R5 = **x**

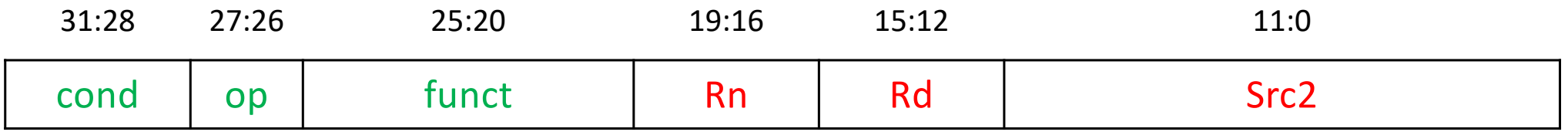
High-Level code

```
i = 0;
x = 4080;
```

ARM Assembly Code

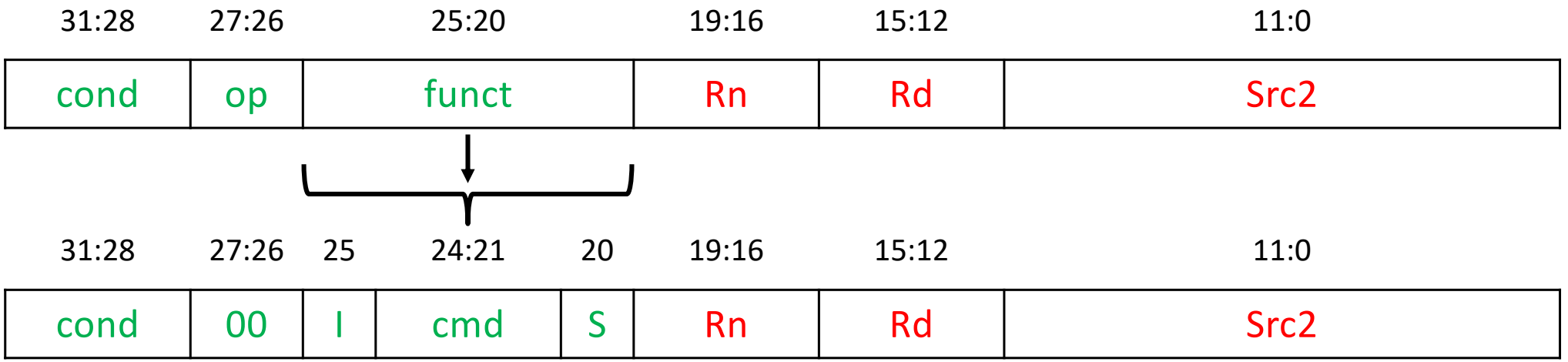
```
MOV  R4,   #0
MOV  R5,   #0xFF0
```

Instruction Format – 1: DP



- **Operands**
  - **Rn [19:16]:** first source operand register (0000, 0001, ..., 1111)
  - **Src2 [11:0]:** second source register or immediate
  - **Rd [15:12]:** destination register
- **Control fields**
  - **cond [31:28]:** specifies conditional execution (1110 for unconditional)
  - **op [27:26]:** the operation code or opcode (00 for data processing)
  - **funct [25:20]:** the function/operation to perform

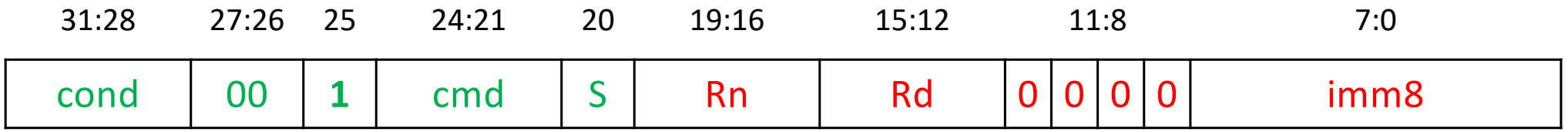
Breaking down funct Field



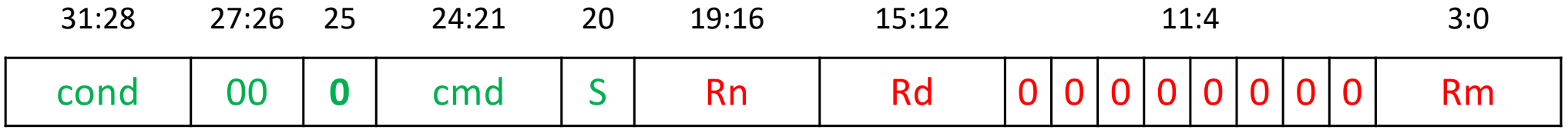
- **cmd [24:21]:** specifies the specific DP instruction (0100 for ADD; 0010 for SUB)
- **I-bit [25]:** informs the control unit about Src2
  - I = 0: Src2 is a register
  - I = 1: Src2 is an immediate
- **S-bit [20]:** 1 if the instruction sets the condition flags

Two DP Formats (Src2 Variations)

Immediate (assume 11:8 are 0 for now)



Register (assume 11:4 are 0 for now)

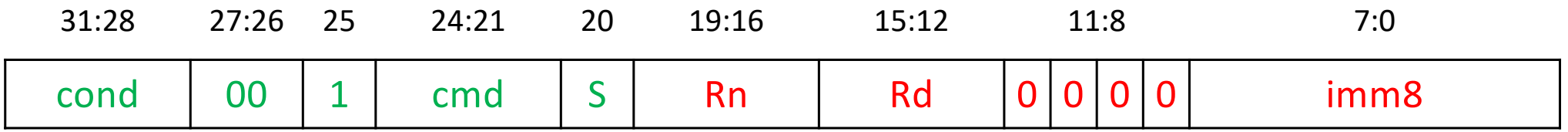


DP with Src2 as Immediate

- Bit 25 (I) informs the CPU how to interpret Src2
  - I = 1, CPU interprets Src2[7:0] as an **unsigned** 8-bit constant

- Format (Src2 = immediate)

```
ADD  R0,   R1,   #16
      |     |     |
      v     v     v
ADD  Rd,   Rn,   #imm8
```

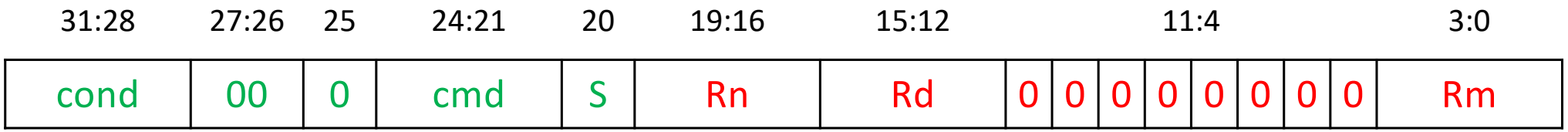


DP with Src2 as Register

- Bit 25 (I) informs the CPU how to interpret Src2
  - I = 0, CPU interprets Src2[3:0] as a register

- Format (Src2 = Register)

```
ADD  R0,   R1,   R3
      ↓    ↓    ↓
ADD  Rd,   Rn,   Rm
```



More Data Processing Insts.

- AND
- ORR (OR)
- EOR (XOR)

- BIC (Bit Clear)

- MVN (MoVe and Not)

The Bit Clear Instruction

- `Bit Clear (BIC)`
  - Used for bit masking bits and forcing **unwanted** bits to `0`

- `BIC R6, R1, R2`
  - R2 is the mask
    - The bits we want to CLEAR or ZERO in `R1` are set to TRUE in `R2`
  - The instruction stores the result of `R1 AND (NOT R2)` in `R6`

Example of Data Processing

Source registers

R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code

```
AND  R3, R1, R2
ORR  R4, R1, R2
EOR  R5, R1, R2
BIC  R6, R1, R2
MVN  R7, R2
```

Result

R3	0100 0110	1010 0001	0000 0000	0000 0000
R4	1111 1111	1111 1111	1111 0001	1011 0111
R5	1011 1001	0101 1110	1111 0001	1011 0111
R6	0000 0000	0000 0000	1111 0001	1011 0111
R7	0000 0000	0000 0000	1111 1111	1111 1111

Design Principle

- Make the common case fast
  - ARM architecture includes only simple commonly used instructions
  - The number of instructions is kept small, so hardware required for decoding is simple, small, and fast
  - More elaborate operations are performed using sequences of multiple simple instructions

RISC vs. CISC Architectures

- Reduced Instruction Set Computer (RISC)
  - Provide a **small set** of simple instructions
  - Minimizes hardware complexity (high clock rate, power-efficient)
  - Requires many instructions to solve a complex problem
  - **Examples:** ARM, MIPS, QuAC, RISC-V

- Complex Instruction Set Computer (CISC)
  - Provides many complex instructions
  - Complex hardware (longer critical paths, lower clock frequency)
  - Each instruction is more complex so fewer instructions to solve a problem
  - **Example:** Intel x86

Another RISC ISA: QuAC

- **Fixed width** instructions make decoding easy and simple
- A small number of **crucial** instructions (fewer opcodes save instruction real-estate)

Register Operands Format (R-Format)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op				cond	rd			0	ra			0	rb		

Immediate Format (I-Format)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op				cond	rd			imm8							

- Two formats and **regularity** in the ISA (across formats)
  - `rd` in same place ($\texttt{Instr}_{10:8}$)
  - `opcode` in the same place
- `seth`: somewhat complex

- Few general-purpose registers
- Space for constants in the ISA
- Easy to convert to hexadecimal
- Conditional execution + general-purpose `PC` = Conditional branch instructions

`seth` moves an 8-bit constant (imm8) into the high byte of the destination register `rd`, leaving the low byte of `rd` unchanged. Formally,

```
rd = (#imm8 << 8) | (rd & 0xff)
```

Data Movement Instructions

Motivation

Main Memory: **Outside the CPU**

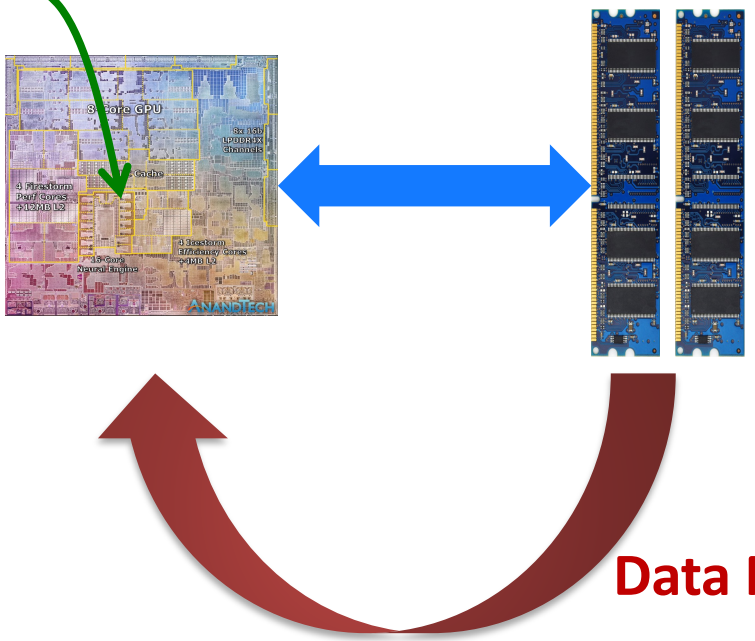
Physically separated from CPU (external)

Large and Slow

Registers: Inside the CPU

Tight integration with ALU

Small and Fast



Data Movement Instructions

- Real programs need to operate on more data than can fit in the register file
  - Most data resides in (slow) memory
  - Fetched from memory into the register file when needed

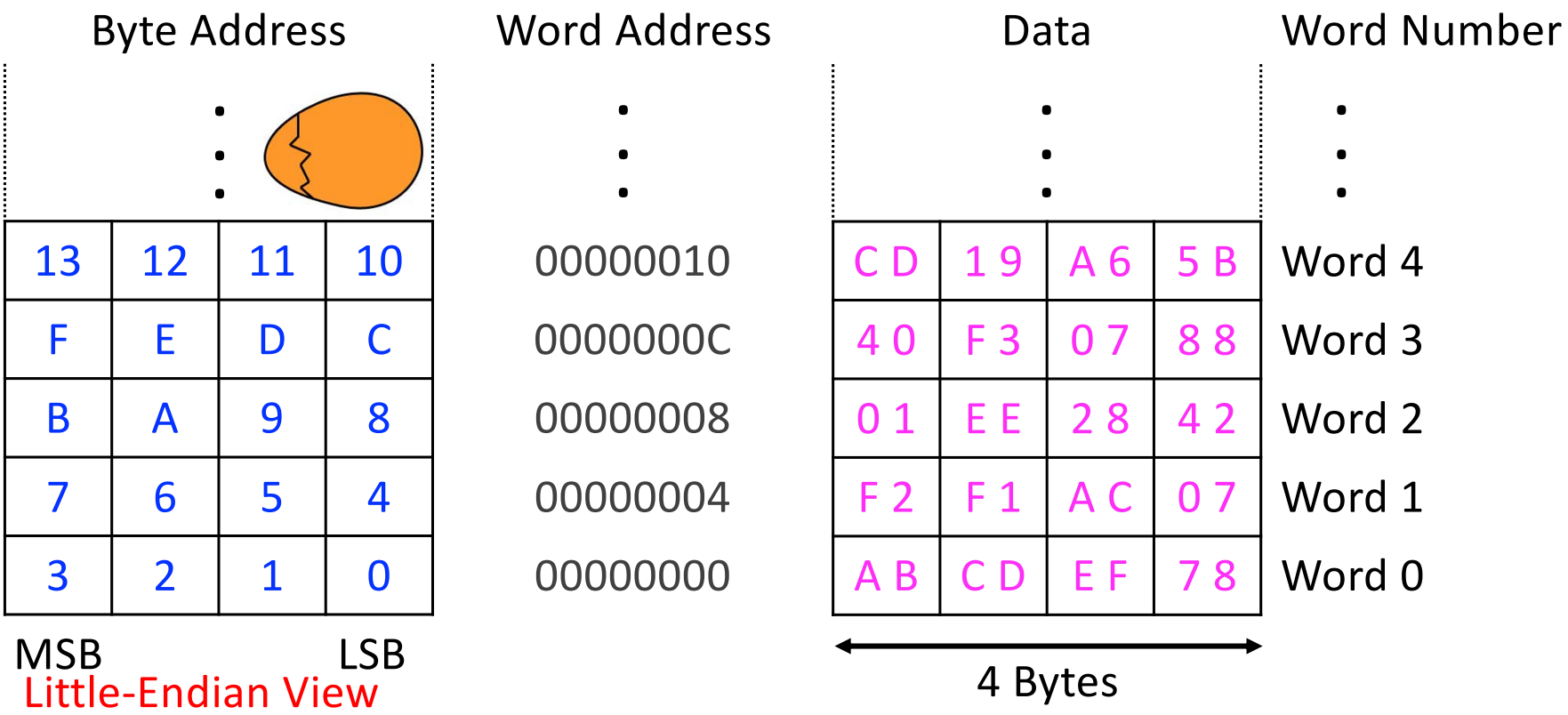
- **Two instructions to facilitate data movement**

  - **The LDR instruction:** Bring data word from memory into the register file
    - **L**oa**D** **R**egister

  - **The STR instruction:** Store data word from the register file into memory
    - **ST**ore **R**egister

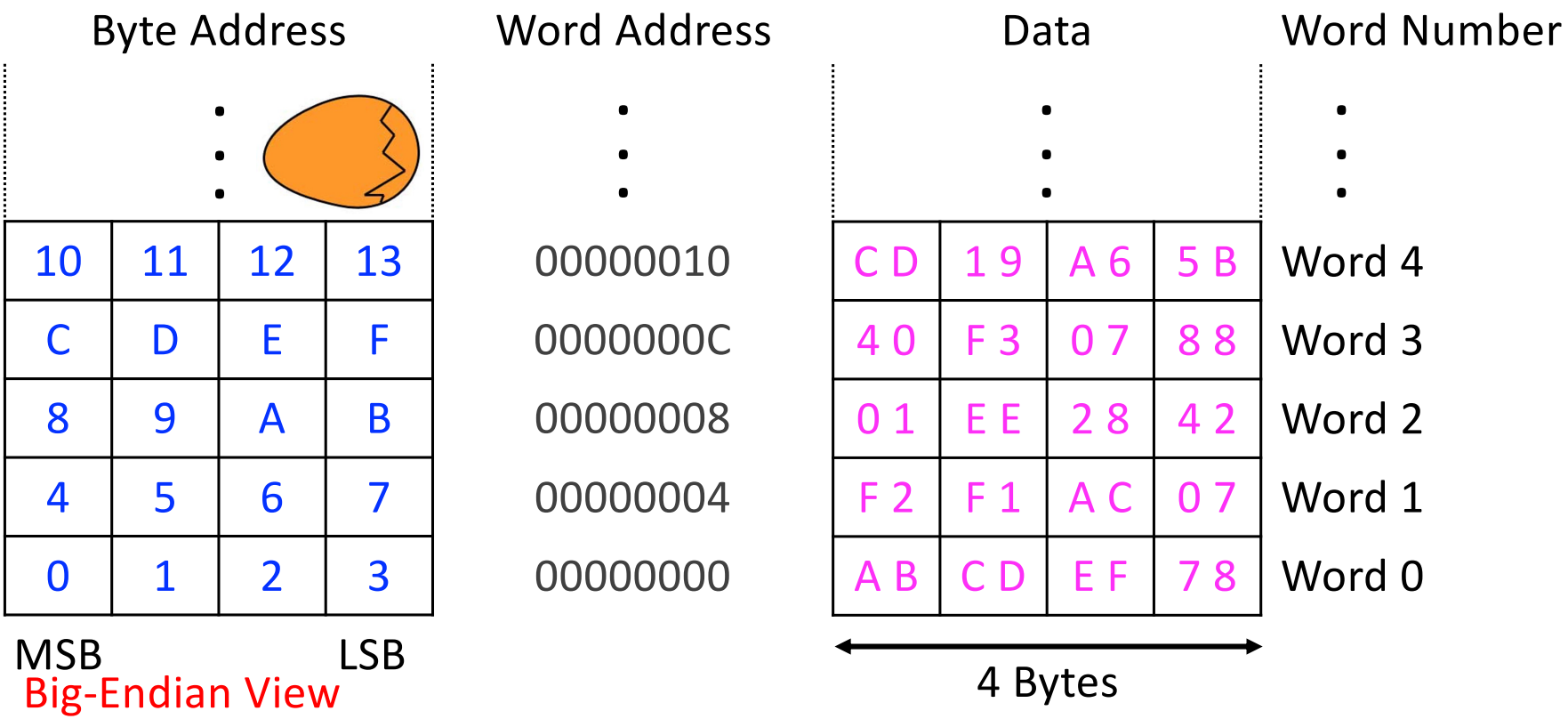
Memory View (32 bits = 4 bytes)

- Byte-addressable memory (each box is a byte; each row is a word)
- Byte addresses (left) and 8-bit byte data (right, 1 byte = 2 Hex digits)



Memory View (32 bits = 4 bytes)

- Byte-addressable memory (each box is a byte; each row is a word)
- Byte addresses (left) and 8-bit byte data (right, 1 byte = 2 Hex digits)



Reading from Memory

- **Format** of **L**oa**D** **R**egister instruction

```
LDR R0, [R1,  #12]
```

- **Address calculation (base + offset addressing)**
  - Add **base** address (contents of `R1`) to the **offset** (`12`)
  - Address = (`R1 + 12`)
  - Use any register for base address
  - `R1` is a source (register) operand

- **Result**
  - `R0` holds the data at memory address (`R1 + 12`) after the instruction is executed
  - `R0` is a destination (register) operand

LDR Example

- Read a 32-bit word of data at memory (byte) address 8 into R3. Use R2 as the base register. Show the contents of R3.
  - Let's initialize R2 to 0, and add 8 as the offset

```
MOV    R2,  #0
LDR    R3,  [R2, #8]
```

R3	0x 01 EE 28 42

Word Address	Data				Word Number
00000010	CD	19	A6	5B	Word 4
0000000C	40	F3	07	88	Word 3
00000008	01	EE	28	42	Word 2
00000004	F2	F1	AC	07	Word 1
00000000	AB	CD	EF	78	Word 0

Address vs. Value

- Square brackets signify **address** (also called **pointer** in C)

```
LDR    R3, [R2, #8]
```

  - If you [add the contents of register R2 (a binary number) to constant #8, you will get the **address** with which to **access** memory]

- When presented with an address, memory obliges by returning the value stored at address given (8 in this example)

- In a 32-bit computer
  - Width of address bus = 32 bits (address space = 2^{32} locations)
  - Although memory is byte-addressable, it returns a 32-bit word to fill the entire register

Writing to Memory

- **Format** of **ST**ore **R**egister instruction

  STR R0, [R1, #12]

- **Address calculation**
  - Add base address (R1) to the offset (12)
  - Address = (R1 + 12)
  - R0 and R1 are both source (register) operands

- **Result**
  - Memory address (R1 + 12) contains the value in R0 after the instruction is executed
  - The second operand is the destination, or the destination operand is memory address

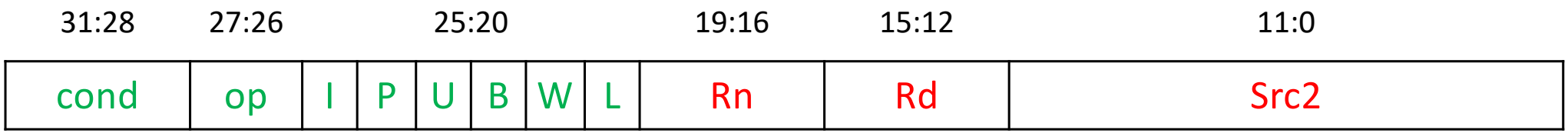
STR Example

- Store the value held in R7 into memory word 21
  - Let's initialize R5 to 0, and add 84 (21 X 4) as the offset

```
MOV     R5,     #0
STR     R7,     [R5,  #0x54]
```

- The offset can be written in decimal or hexadecimal: 84 (decimal) is 0x54 (Hex)

Instruction Format – 2: Memory



- op = 01
- Rn = base register (base address)
- Rd = destination (load), source (store)
- Src2 = offset (register, shifted register, immediate)
- funct [25:20] = 6 control bits
  - I (Bit 25): Encoding of Src2
  - L (Bit 20): Load or Store

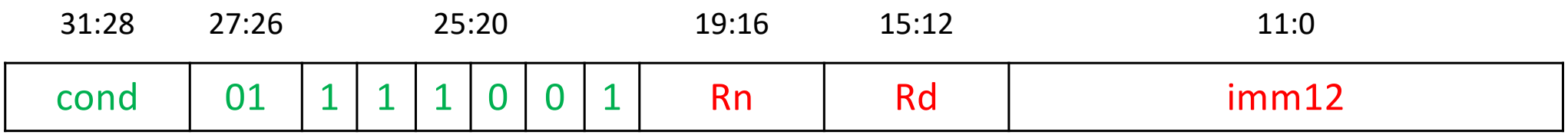
LDR with Src2 as Immediate

- I (Bit 25) = 1: Src2 = imm12 where imm2 is a 12-bit unsigned offset added to the value in the base register (Rn)

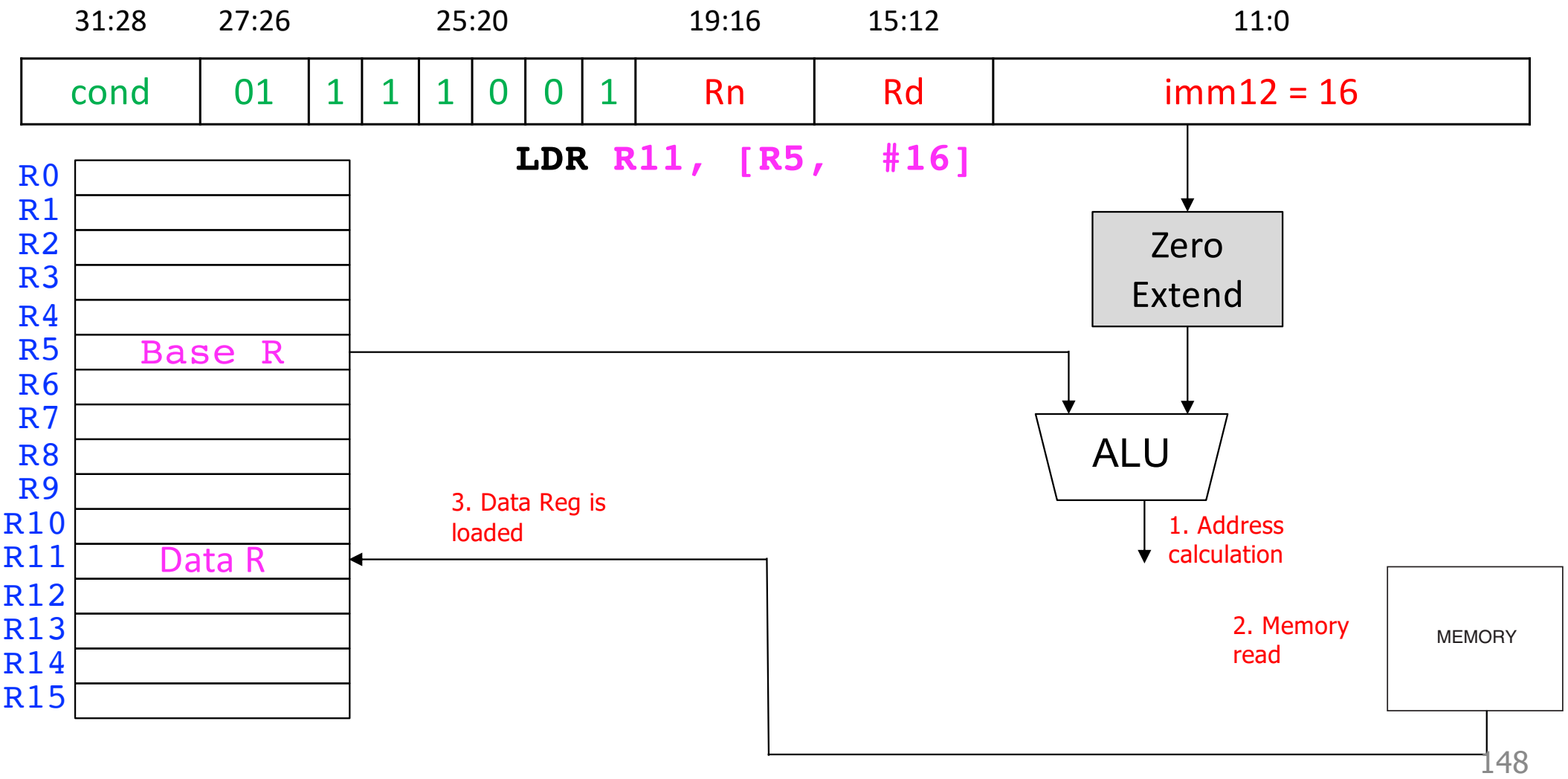
- Format of **L**oa**D** **R**egister instruction

```
LDR  R0,  [R1,  #12]
      |     |     |
      v     v     v
LDR  Rd,  [Rn,  #imm12]
```

- L (Bit 20) = 1: CPU performs an LDR



LDR Datapath



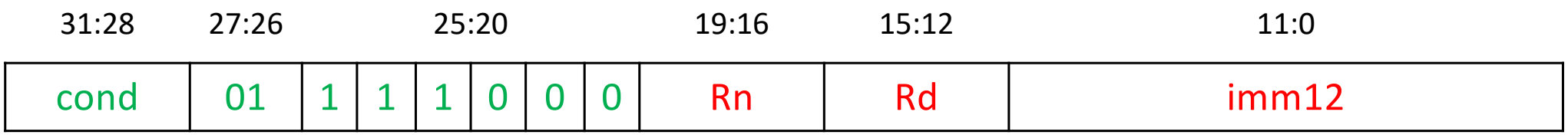
STR with Src2 as Immediate

- I (Bit 25) = 1: Src2 = imm12 where imm2 is a 12-bit unsigned offset added to the value in the base register (Rn)

- Format of **ST**ore **R**egister instruction

```
STR R0, [R1,  #12]
    |   |     |
    v   v     v
STR Rd, [Rn,  #imm12]
```

- L (Bit 20) = 0: CPU performs an STR



Conditional Execution

Conditional Execution

- We can execute instructions conditionally based on a specific condition being TRUE or FALSE

- ARM allows conditional execution in two steps
  - **Step # 1**: Instruction sets the condition flags (Negative, Zero, Carry, Overflow)
  - **Step # 2**: Subsequent instructions execute based on the state of the condition flags

- Condition (status) flags are set by ALU
  - They are contained in a register called the **C**urrent **P**rogram **S**tatus **R**egister (**CPSR**)

Setting the Condition Flags

- **Method 1:** Use the **CMP**ARE instruction

```
CMP  R5,  R6
```

  - The instruction subtracts the second source operand from the first operand (R5 – R6)
  - The instruction does not save any result

- Flags are set as follows
  - Is 0, Z = 1
  - Is negative, N = 1
  - Causes a carry out, C = 1
  - Causes a signed overflow, V = 1

Setting the Condition Flags

- **Method 2:** Append the instruction mnemonic with **S**

```
ADDS  R1,  R2,  R3
```

- The instruction adds source operands R2 and R3
- It sets the flags (**S**)
- It saves the result in R1

Condition Mnemonics

- We can execute instructions conditionally based on the status of the flags register

- Condition for execution is encoded as a *condition mnemonic* appended to the instruction mnemonic

```
CMP    R1,  R2
SUBNE  R3,  R5,  R8
ADDEQ  R1,  R2,  R3
```

- **NE** and **EQ** are condition mnemonics
- SUB executes only if R1 is not equal to R2 (meaning Z = 0)

Condition Mnemonics

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\bar{Z}$
0010	CS / HS	Carry set / Unsigned higher or same	C
0011	CC / LO	Carry clear / Unsigned lower	$\bar{C}$
0100	MI	Minus / Negative	N
0101	PL	Plus / Positive of zero	$\bar{N}$
0110	VS	Overflow / Overflow set	V
0111	VC	No overflow / Overflow clear	$\bar{V}$
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z\ OR\ \bar{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	$Z\ OR\ (N \oplus V)$
1110	AL (or none)	Always / unconditional	ignored

Instructions that affect condition flags

Type	Instructions	Condition Flags
Add	ADDS, ADCS	*N*, *Z*, *C*, *V*
Subtract	SUBS, SBCS, RSBS, RSCS	*N*, *Z*, *C*, *V*
Compare	CMP, CMN	*N*, *Z*, *C*, *V*
Shifts	ASRS, LSLS, LSRS, RORS, RRXS	*N*, *Z*, *C*
Logical	ANDS, ORRS, EORS, BICS	*N*, *Z*, *C*
Test	TEQ, TST	*N*, *Z*, *C*
Move	MOVS, MVNS	*N*, *Z*, *C*
Multiply	MULS, MLAS, SMLALS, SMULLS, UMLALS, UMULLS	*N*, *Z*

Example

- `R5 = 17 and R9 = 23`

- Will the **SUBEQ** and **ORRMI** instructions execute?

  - **NZCV** = **?**

```
CMP     R5,   R9
SUBEQ   R1,   R2,   R3
ORRMI   R4,   R0,   R9
```

Another Example

- `R2 = 0x80000000 and R3 = 0x00000001`

- Which instructions will execute?

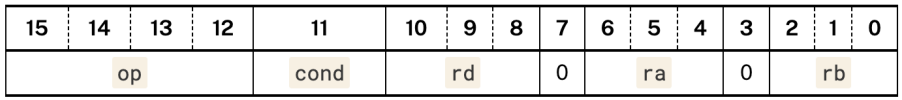
  - **Flags: N Z C V = ?**

```
CMP     R2,   R3
ADDEQ   R4,   R5,   #78
ANDHS   R7,   R8,   R9
ORRMI   R10,  R11,  R12
EORLT   R12,  R7,   R10
```

Conditional Execution in QuAC

- Bit 11 is associated with a condition code

Register Operands Format (R-Format)



- ALU instructions set the flags (a.k.a. condition codes). See Flags in QuAC ISA
  - The CPU uses that information to determine whether to execute the current instruction or not (e.g., store result into register file or memory)

Name	Suffix	Encoding	Condition	Meaning
Always	-	`0`	-	Always executes
Equals	`eq`	`1`	`Z == 1`	Execute if latest ALU result was zero

- If **cond field** ($Instr_{11}$) is TRUE, then
  - Execute the instruction only if he last ALU instruction set the **Z** flag to TRUE
  - Otherwise, do not execute the instruction (depart from the usual control flow)
- The default encoding of the **cond** field is 0 (execute the instruction)
  - `add r1, r2, r3` (**cond** = FALSE)
  - `addeq r1, r2, r3` (**cond** = TRUE)

Branch Instructions

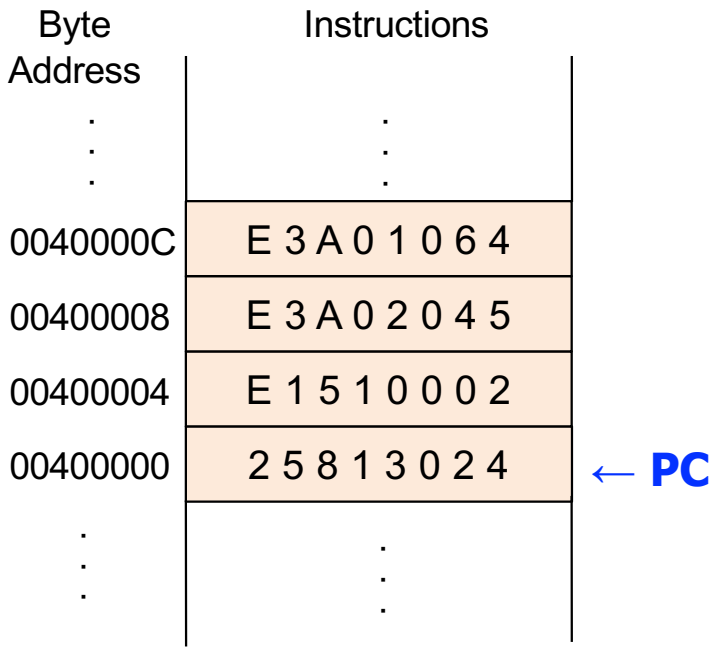
Branch Instructions

- Typically, a computer program is executed in sequence
  - First instruction is executed, then the second, then the third, and so on
- **Decision making** is an important advantage of computers
  - `if` and `if-else` statements
  - `for` and `while` loops
  - `switch-case` statements
- ARM provides branch instructions to **skip** and **repeat** code

Recall: Program Counter (PC)

- Program Counter (**PC**): Contains the **address** of (or **points** to) the next instruction to be executed
- Incremented by 4 (4 bytes or 32 bits) in the FETCH phase

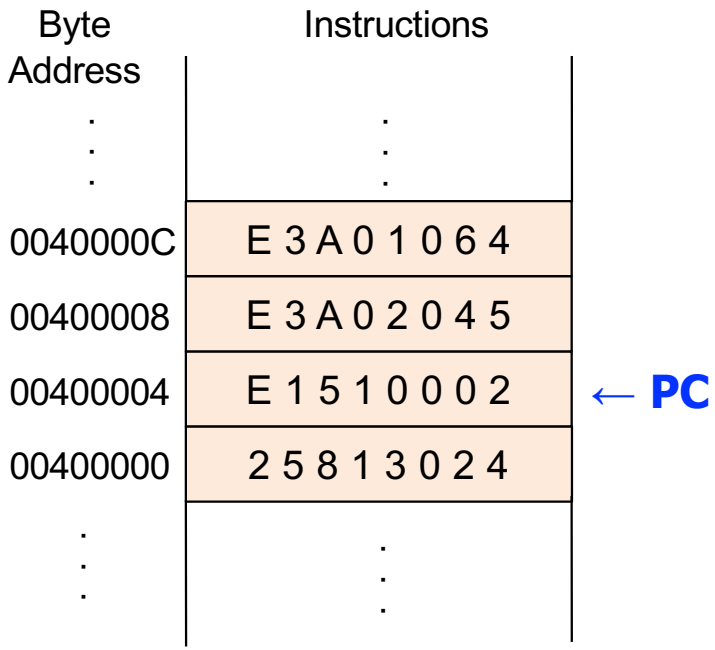
- PC = PC + 4 to execute the next **sequential instruction** in memory



Recall: Program Counter (PC)

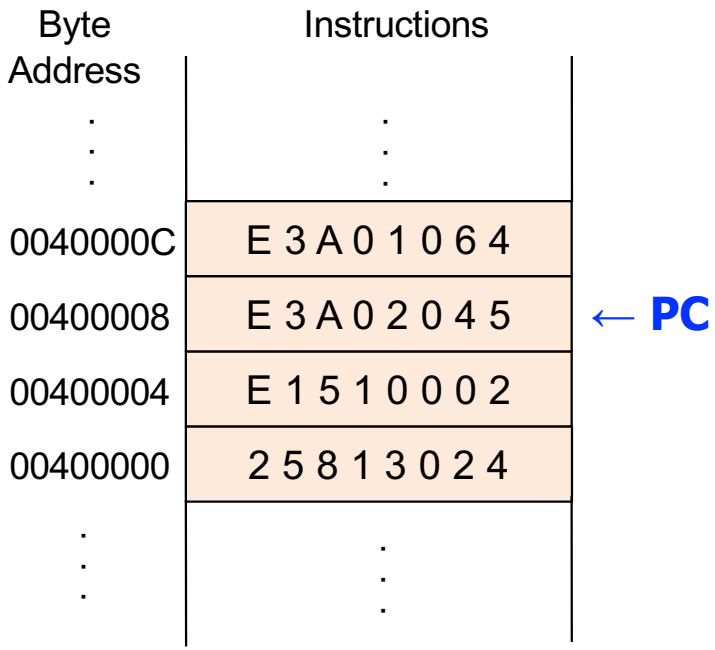
- Program Counter (**PC**): Contains the **address** of (or **points** to) the next instruction to be executed
- Incremented by 4 (4 bytes or 32 bits) in the FETCH phase

- PC = PC + 4 to execute the next **sequential instruction** in memory



Recall: Program Counter (PC)

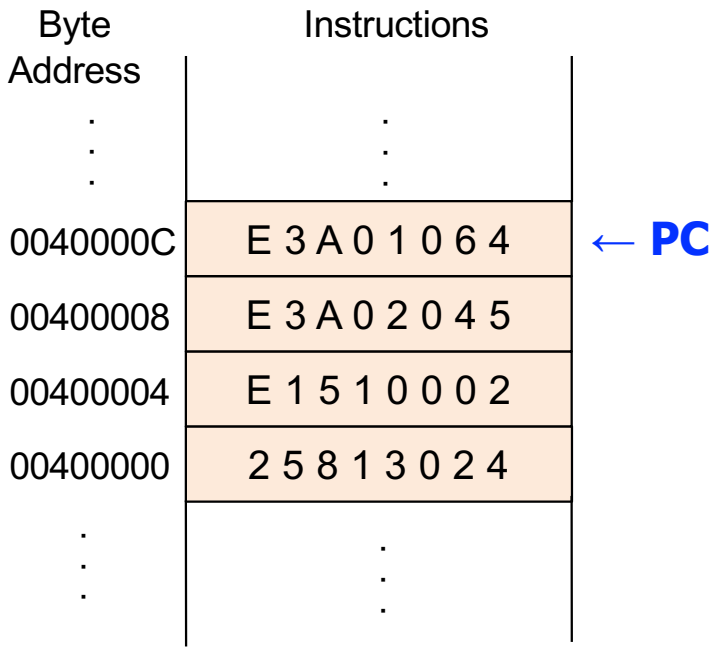
- Program Counter (**PC**): Contains the **address** of (or **points** to) the next instruction to be executed
- Incremented by 4 (4 bytes or 32 bits) in the FETCH phase



- PC = PC + 4 to execute the next **sequential instruction** in memory

Recall: Program Counter (PC)

- Program Counter (**PC**): Contains the **address** of (or **points** to) the next instruction to be executed
- Incremented by 4 (4 bytes or 32 bits) in the FETCH phase

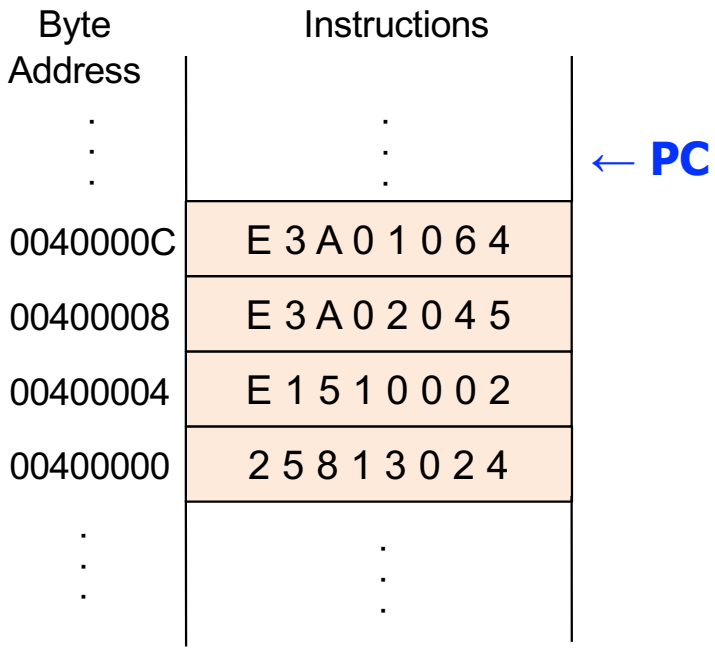


- PC = PC + 4 to execute the next **sequential instruction** in memory

Recall: Program Counter (PC)

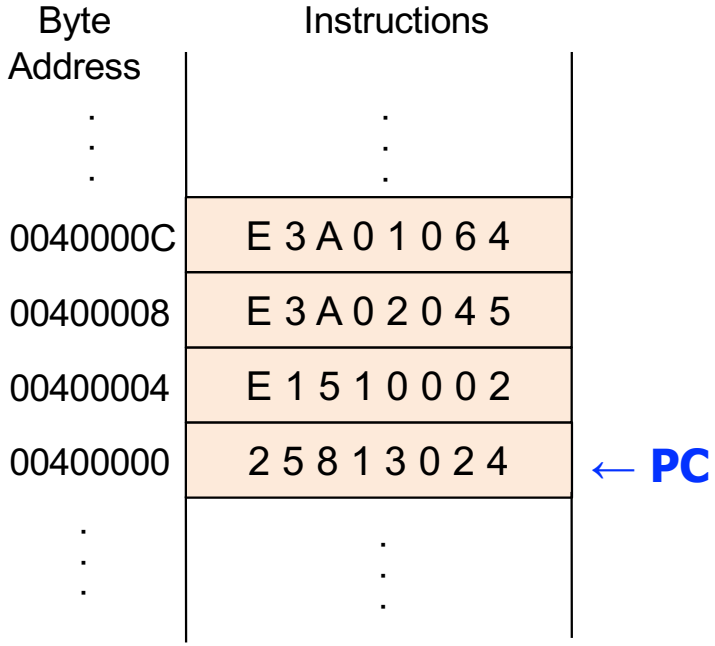
- Program Counter (**PC**): Contains the **address** of (or **points** to) the next instruction to be executed
- Incremented by 4 (4 bytes or 32 bits) in the FETCH phase

- PC = PC + 4 to execute the next **sequential instruction** in memory



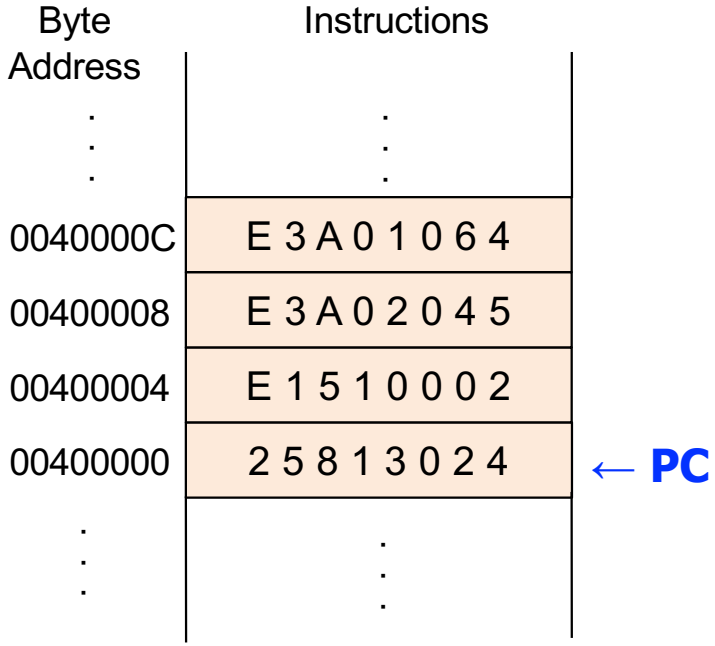
Branch Instructions and PC

- Branch instructions change the **PC** to point to a different instruction than the next **sequential instruction** in memory
- Updated by a different address in the EXECUTE phase
  - New address PC points to is determined by formula (addressing mode)



Branch Instructions and PC

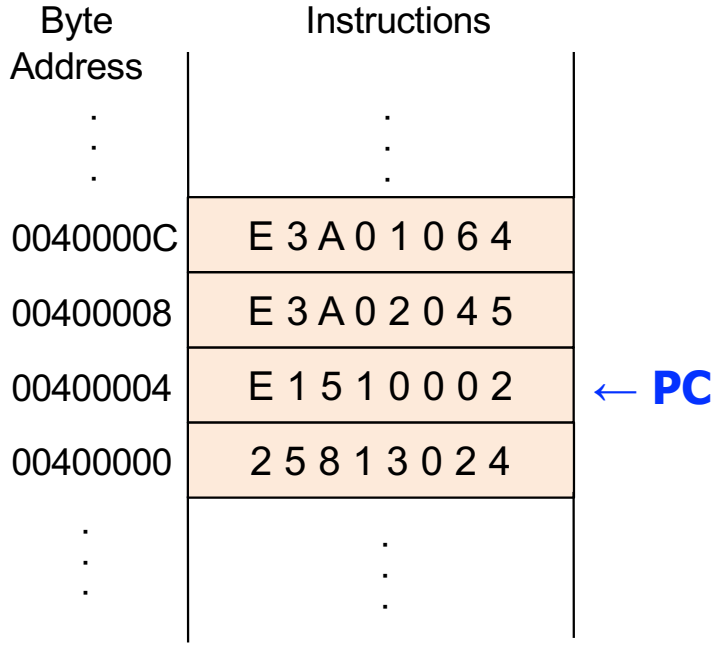
- Branch instructions change the **PC** to point to a different instruction than the next **sequential instruction** in memory
- Updated by a different address in the EXECUTE phase
  - New address PC points to is determined by formula (addressing mode)



- PC = PC + 4 to execute the next **sequential instruction** in memory

Branch Instructions and PC

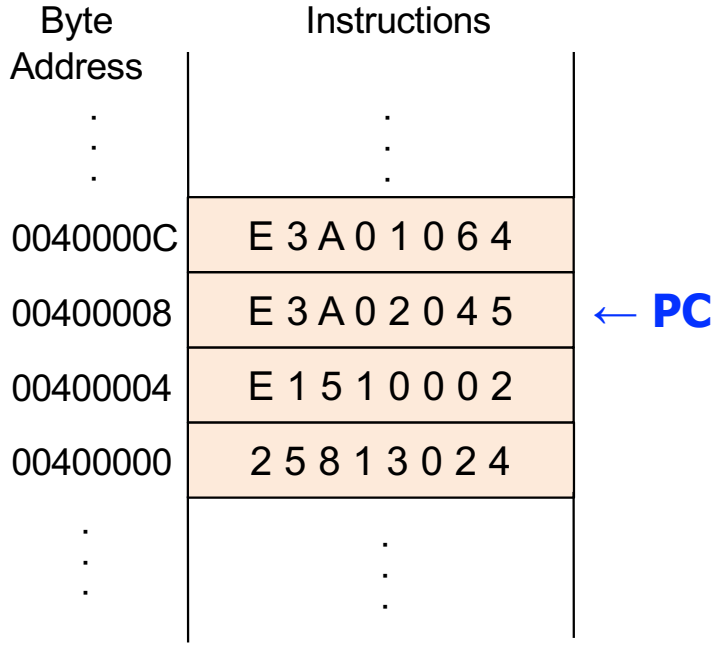
- Branch instructions change the **PC** to point to a different instruction than the next **sequential instruction** in memory
- Updated by a different address in the EXECUTE phase
  - New address PC points to is determined by formula (addressing mode)



- PC = PC + 4 to execute the next **sequential instruction** in memory

Branch Instructions and PC

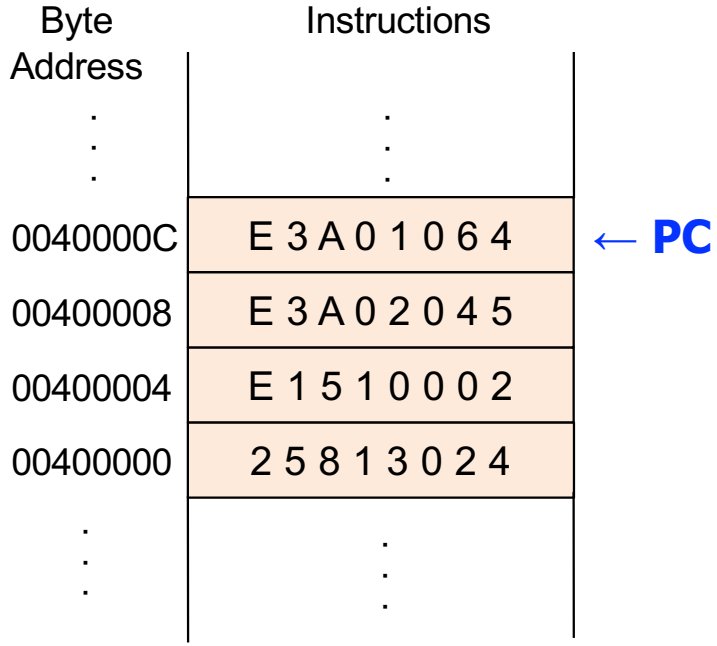
- Branch instructions change the **PC** to point to a different instruction than the next **sequential instruction** in memory
- Updated by a different address in the EXECUTE phase
  - New address PC points to is determined by formula (addressing mode)



- PC = PC + 4 to execute the next **sequential instruction** in memory

Branch Instructions and PC

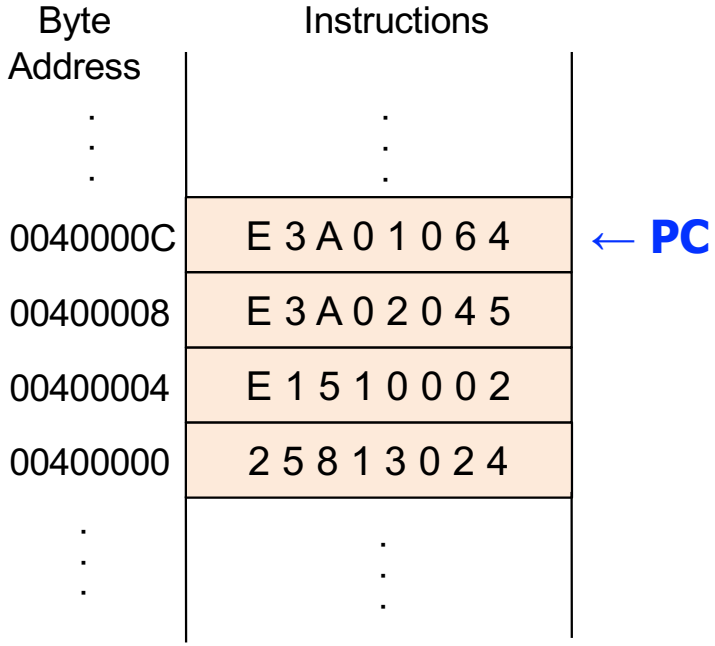
- Branch instructions change the **PC** to point to a different instruction than the next **sequential instruction** in memory
- Updated by a different address in the EXECUTE phase
  - New address PC points to is determined by formula (addressing mode)



- PC = PC + 4 to execute the next **sequential instruction** in memory

Branch Instructions and PC

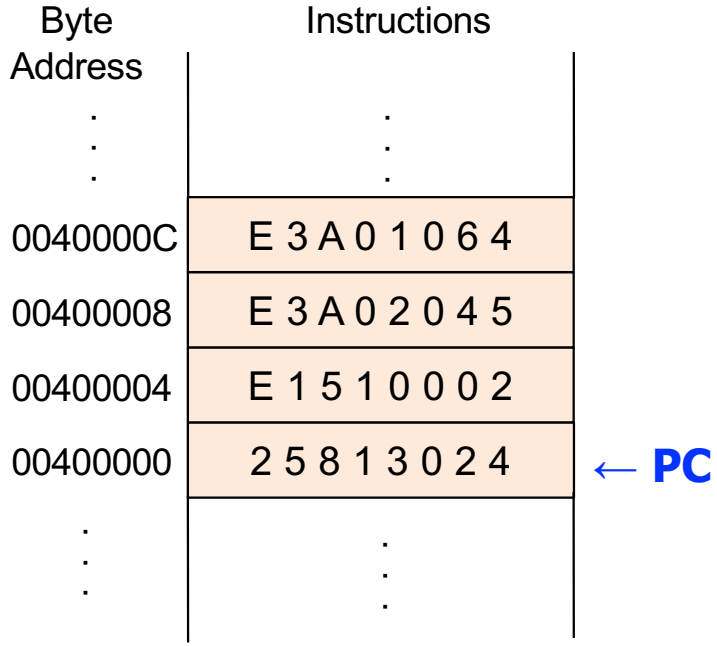
- Branch instructions change the **PC** to point to a different instruction than the next **sequential instruction** in memory
- Updated by a different address in the EXECUTE phase
  - New address PC points to is determined by formula (addressing mode)



- Update PC to **re-**execute the four **instruction** sequence again (**for loop**)

Branch Instructions and PC

- Branch instructions change the **PC** to point to a different instruction than the next **sequential instruction** in memory
- Updated by a different address in the EXECUTE phase
  - New address PC points to is determined by formula (addressing mode)



- Branch instruction updates PC to **re-**execute the **four-instruction** sequence again (`loop`)

Type of Branches

- Branch (B)
  - Branches to another **TARGET** instruction
  - Unconditional branch: always executes the target instruction
  - Conditional branch: either executes the TARGET instruction or the next sequential instruction in memory based on a condition
    - **BEQ** (Branch if the Zero flag is set)
    - **BNE** (Branch if the Zero flag is not set)

- Branch and Link (BL)
  - A special branch to provide support for functions in C++ or Java
    - Architectural support for high-level language demands

Unconditional Branch

- The **B**ranch in this example is unconditional and **always** TAKEN (T)

```
Assembly code:
    ADD   R1,   R2,   #17
    B     TARGET
    ORR   R1,   R1,   R3
    AND   R3,   R1,   #0xFF
TARGET
    SUB   R1,   R1,   #78
```

- After encountering **B**, the CPU executes SUB instead of ORR

- The **label** TARGET is a **memory address** in human readable form
  - TARGET is transformed into a **memory address** by a tool called **assembler**
  - Assemblers transform assembly code into machine code (0's and 1's)