CS201
Computer Organization B. Tech. II (CSE) Sem-3

Introduction

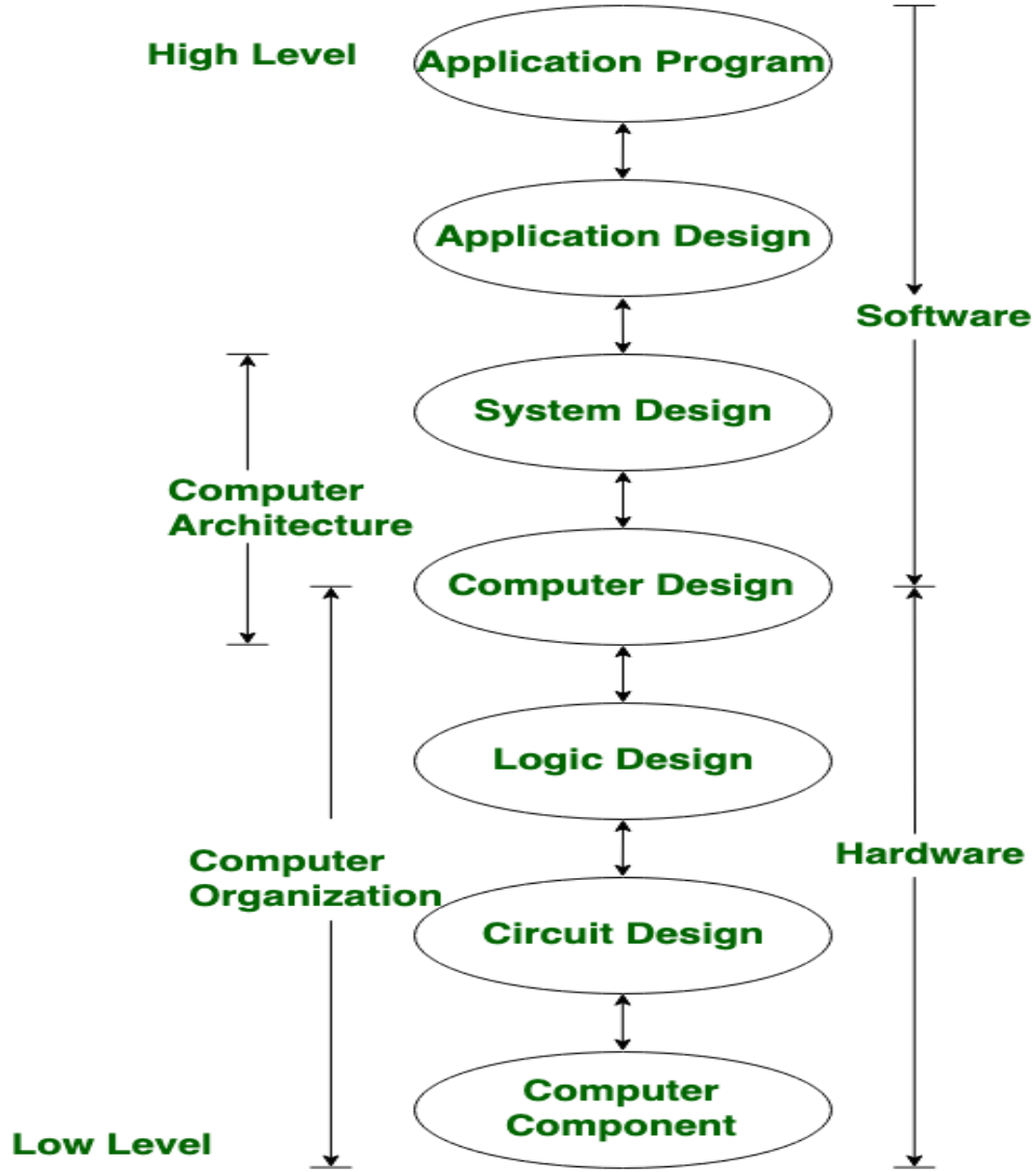
What Is Computer Organization ?

Let us first understand the meaning of term organization in the context of computers. The term organization is defined as arranging , classifying things together logically to maximize the functional convenience.

The computer organization is based on the computer architecture. The computer organization implements the system architecture.

In simple words , the computer organization is all about organizing various system hardware components and how these components are interconnected.

- **Computer Architecture** is a functional description of requirements and design implementation for the various parts of a computer. It deals with the functional behavior of computer systems. It comes before the computer organization while designing a computer.


High Level
Application Program
Application Design
System Design
Computer Design
Logic Design
Circuit Design
Computer Component
Low Level
Software
Hardware
Computer Architecture
Computer Organization
Computer System

S. No.	Computer Architecture	Computer Organization
1.	Architecture describes what the computer does.	The Organization describes how it does it.
2.	Computer Architecture deals with the functional behavior of computer systems.	Computer Organization deals with a structural relationship.
3.	In the above figure, it's clear that it deals with high-level design issues.	In the above figure, it's also clear that it deals with low-level design issues.
4.	Architecture indicates its hardware.	Where Organization indicates its performance.
5.	As a programmer, you can view architecture as a series of instructions, addressing modes, and registers. For designing a computer, its architecture is fixed first.	The implementation of the architecture is called organization. For designing a computer, an organization is decided after its architecture.

Computer Organization

- Concern with ‘How does a computer work?’
- Considers all physical aspects of computer systems
- The way in which various circuits and structural components come together to make up fully functional computer systems is the way the system is organized
- Logical aspects of system as seen by the programmer
- e.g., instruction sets, instruction formats, data types, addressing modes
- While the Computer Architecture considers design and components

What Do You Learn In Computer Organization?

- Understanding computer architecture is essential. The relation between instruction set architecture design, datapath design, and algorithm design should be evaluated.
- Understanding the performance tradeoffs involved in designing the memory subsystem, including cache, main memory, and virtual memory, will help you make better decisions.

Why Do We Need To Study Computer Organization?

- By knowing what's inside and how it works, you will be able to design, develop, and implement applications that are faster, cheaper, more efficient, and easier to use, since you will be able to make informed decisions instead of relying on assumptions and guestimates.

What Is The Purpose Of Computer Architecture And Organization?

- A computer architecture describes the design of computers, data storage devices, and networking components that store and run programs, transmit data, and drive interactions between computers, across networks, and with users, as well as the way in which they communicate.

Subject Overview

- Goal is to Provide the Knowledge of :
- Computer system's functional components, their characteristics, their performance, and their interactions
- Computer architecture in order to structure a program so that it runs more efficiently on a real machine

Study of:

- The laws of computer organization and design
- Performance Measures
- Interfaces between hardware and software
- Instruction Set Design, Datapath and control path
- Influence of instruction set on performance
- Computer arithmetic
- Design of a processor with pipelining is analyzed

- PreRequisites
- Data Structures and Algorithms
  - Arrays, pointers,
- Logic Design
  - Number system, basic computer arithmetic • Logic circuits

Can help in

- System design tools
  - Application of design theories that is used at the lowest level of system design at higher levels
  - Example: The interface between a processor and its memory chips are used to design the addressing scheme of an IP network
- Software design tools
  - To optimized/simplify the logic portions of software to run faster
- Improved troubleshooting skills
  - To isolate a problem quicker and with greater accuracy
- Interconnectivity
  - Writing software to control the hardware
- Marketability
  - The software engineer with experience in hardware design has a significant advantage over hardware engineers in this market
- To do a particular task,
  - Design a software program on a processor • Design a hardware component to do so

Schedule

Credit: 5

- Lectures : 3
- Tutorial : 1
- Practical : 2

Tutorial & Practical

Tutorial

- Test
- Designing, Calculation and Analysis using examples

- **Online quizzes**
  - Format
  - Objective type: Select the best choice
  - Questions on material already discussed in class
- **Practical**
  - Related to the Design and Implementation

Books

- John L. Hannessy, David A. Patterson- “Computer organization and Design”,3/E,Morgan Kaufmaan, reprint -2003 OR
- Computer Organization – HW/SW Interface, Patterson and Hennessy, 5th edition
- Stallings,” Computer Organization & Architecture : Designing For Performance”, 4/E,PHI EEE ed, 1997
- Tanenbaum – “Structured Computer Organization “, PHI EEE, reprint 1995
- Morris Mano – “Computer Systems Architecture”, 3/E, PHI, reprint 1997
- Hamacher – “Computer Organization”, McGraw-Hill IS ed, 1994

Functional Unit

A computer in its simplest form comprises five functional units namely input unit, output unit memory unit, arithmetic & logic unit and control unit. Figure 2 depicts the functional units of a computer system.

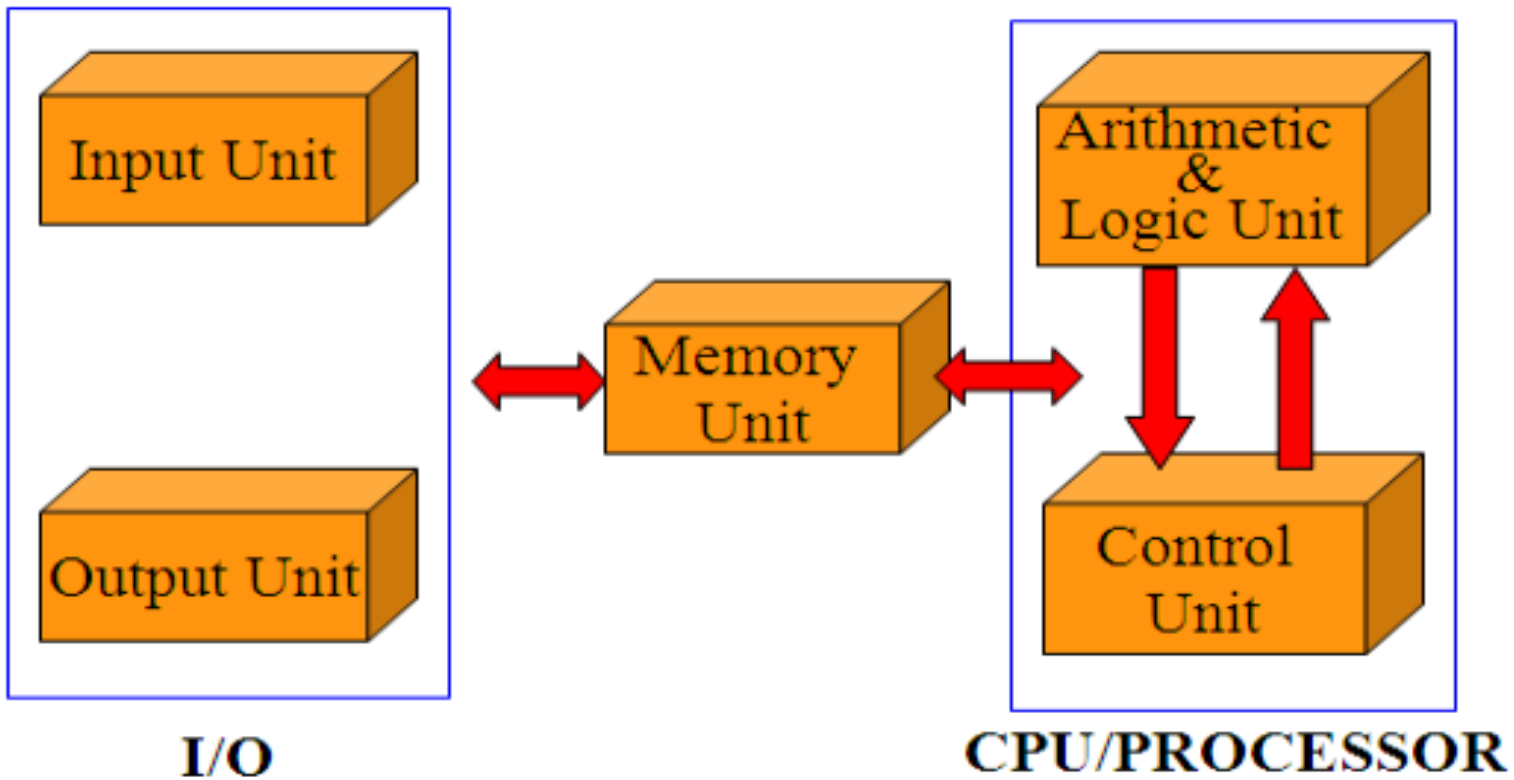


Figure 2: Basic functional units of a computer

The operations of a computer can be summarized as follows:

1. A set of instructions called a program reside in the main memory of computer.
2. The CPU fetches those instructions sequentially one-by-one from the main memory, decodes them and performs the specified operation on associated data operands in ALU.
3. Processed data and results will be displayed on an output unit.
4. All activities pertaining to processing and data movement inside the computer machine are governed by control unit.

Registers

- A processor register (CPU register) is one of a small set of data holding places that are part of the computer processor.
- A register may hold an instruction, a storage address, or any kind of data (such as a bit sequence or individual characters).
- Some instructions specify registers as part of the instruction. For example, an instruction may specify that the contents of two defined registers be added together and then placed in a specified register.

Register	Symbol	Number of bits	Function
Data register	DR	16	Holds memory operand
Address register	AR	12	Holds address for the memory
Accumulator	AC	16	Processor register
Instruction register	IR	16	Holds instruction code
Program counter	PC	12	Holds address of the instruction
Temporary register	TR	16	Holds temporary data
Input register	INPR	8	Carries input character
Output register	OUTR	8	Carries output character

Basic Operational Concepts

- The primary function of a computer system is to execute a program, sequence of instructions. These instructions are stored in computer memory.
- These instructions are executed to process data which are already loaded in the computer memory through some input devices.
- After **processing the data**, the result is either stored in the memory for further reference, or it is sent to the outside world through some output port.
- To perform the execution of an instruction, in addition to the arithmetic logic unit, and control unit, the processor contains **a number of registers** used for temporary storage of data and some special function registers.
- The special function registers include program counters (PC), instruction registers (IR), memory address registers (MAR) and memory and memory data registers (MDR).
- The Program counter is one of the most critical registers in CPU.
- **The Program counter** monitors the execution of instructions. It keeps track on which instruction is being executed and what the next instruction will be.
- **The instruction register IR** is used to hold the instruction that is currently being executed.
- The contents of IR are available to the control unit, which generate the timing signals that control, the various processing elements involved in executing the instruction.
- The two **registers MAR and MDR** are used to handle the data transfer between the main memory and the processor.
- **The MAR** holds the address of the main memory to or from which data is to be transferred.
- **The MDR** contains the data to be written into or read from the addressed word of the main memory.

Stored Program Concept

- When we talk about the **stored program concept,** we need to think with regard to the internal layout and architecture of a computer.
- Early computers such as the IBM had no form of internal storage - all instructions and data were held on punch cards, which could be fed into the IBM to process data
- Nowadays, computers typically use **Von-Neumann Architecture**, which reflects the idea of the stored program concept.
- The Von Neumann Architecture uses the stored program concept where *instructions and data are loaded from main memory into the processor to be executed*.
- The Von-Neumann Architecture, and stored-program concept, works where *machine code instructions and data are stored, and loaded from memory into the processor to be executed in sequential order*.
- Von-Neumann Architecture is used for general purpose machines, where instructions and data are held in the same memory location - this is our **main memory**, or RAM.

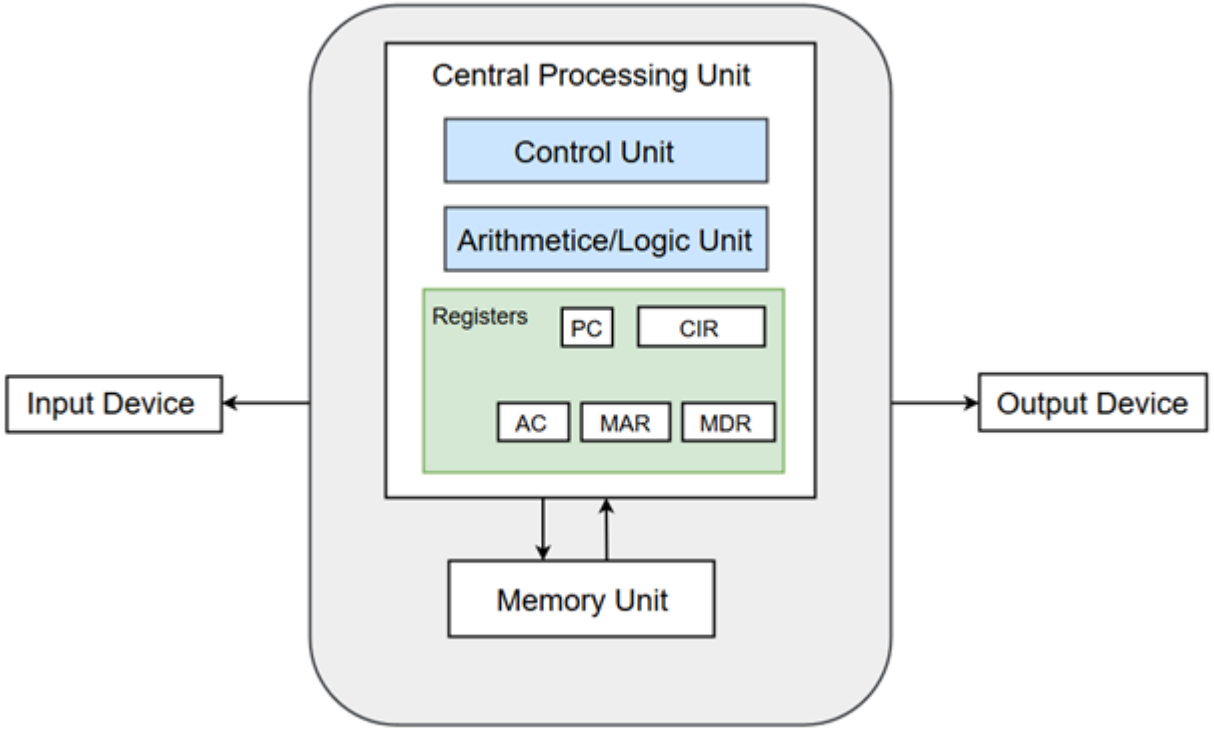
Stored Program Concept - a program must be in main memory in order for it to be executed. The instructions are fetched, decoded and executed one at a time

- Building on the Von Neumann architecture we get the idea of how the stored program concept works. If you have ever loaded a game on a console you might notice that:
  - you need to insert a disc
  - the disc spins
  - the game says loading
  - the game plays
- This is the stored program concept in motion! Let's take apart what is happening:
  - You insert an optical disk (secondary storage) with the code on
  - The code is loaded into main memory
  - The processor fetches, decodes and executes instructions from main memory to play game

Von-Neumann Model

- Von Neumann architecture is based on the stored-program computer concept, where instruction data and program data are stored in the same memory. This design is still used in most computers produced today.
- A Von Neumann-based computer:
  - Uses a single processor
  - Uses one memory for both instructions and data.
  - Executes programs following the fetch-decode-execute cycle

Von-Neumann Model



Components of Von-Neumann Model:

- Central Processing Unit
- Buses
- Memory Unit

Central Processing Unit

The part of the Computer that performs the bulk of data processing operations is called the Central Processing Unit and is referred to as the CPU.

Von-Neumann Architecture comprised of three major bus systems for data transfer.

Bus	Description
Address Bus	Address Bus carries the address of data (but not the data) between the processor and the memory.
Data Bus	Data Bus carries data between the processor, the memory unit and the input/output devices.
Control Bus	Control Bus carries signals/commands from the CPU.

Memory Unit

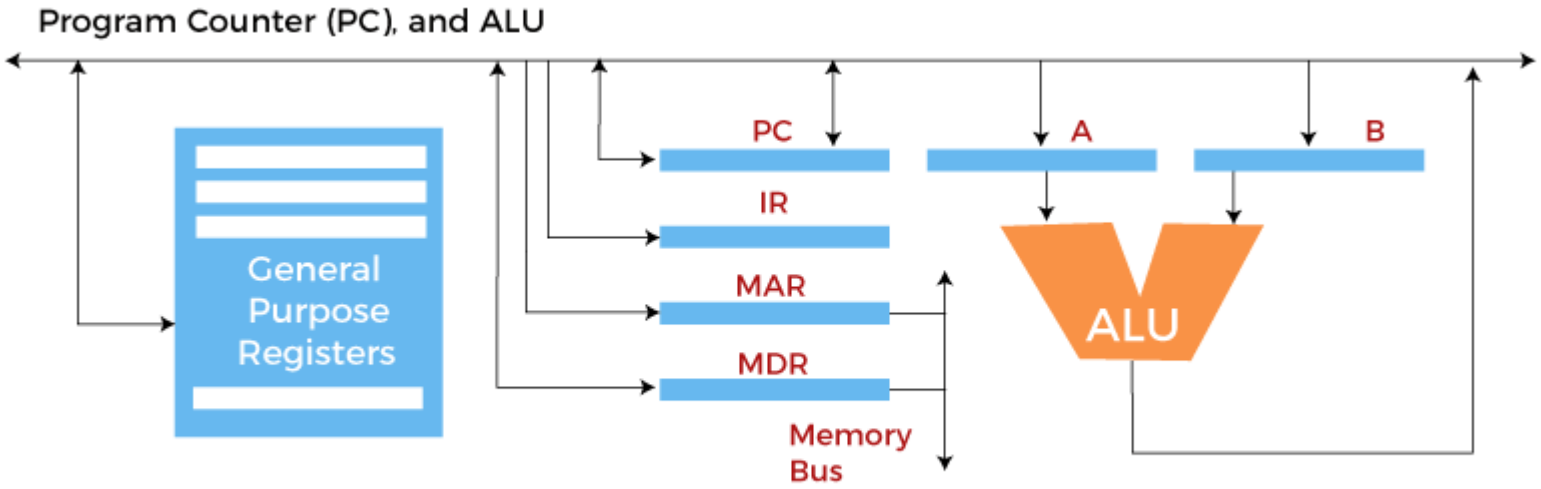
A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of the storage.

Data Paths

- Suppose any data processing operation should be performed in the CPU like transferring the content of register from one place to another, from one register to another register, performing the addition of two numbers in ALU, copying something in memory and copying to register. So, any data processing operation happens in the CPU, and then for that data operation, data follows a specific path, called ***"data path".***
- A data path is a collection of functional units such as ***arithmetic logic units*** or multipliers that perform data processing operations, ***registers,*** and ***buses.*** Along with the control unit, it composes the central processing unit (CPU). A larger data path can be made by joining more than one data path using multiplexers.
- A data path is the ***ALU,*** the set of ***registers,*** and the CPU's internal buses that allow data to flow between them.
- The bus can be used for a single purpose or multiple purposes. When we would have different kinds of buses, different types of ***bus organizations*** will take place, such as:

One Bus organization

In one bus organization, a single bus is used for multiple purposes. A set of general-purpose registers, program counters, instruction registers, memory address registers (MAR), and memory data registers (MDR) are connected with the single bus. Memory read/write can be done with MAR and MDR.



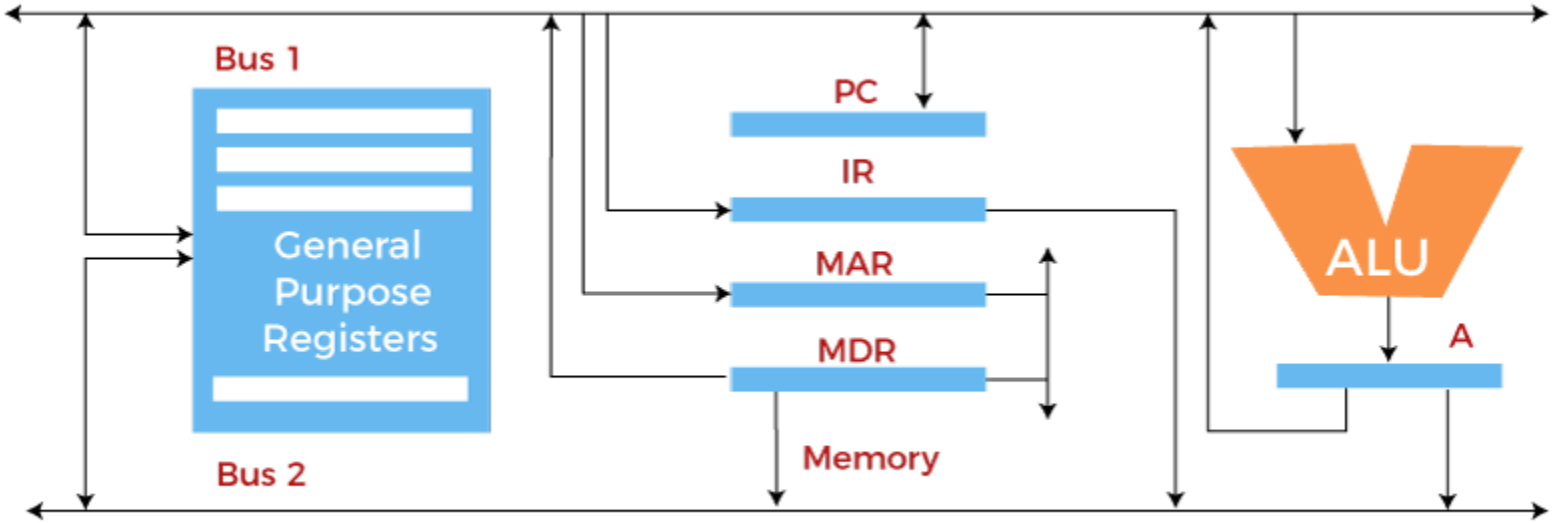
One-bus datapath

The program counter points to the memory location from where the next instruction is to be fetched. The instruction register will hold a copy of the current instruction. In one bus organization, only one operand can be read from the bus at a time.

If the requirement is to read two operands for the operation, then the read operation needs to be carried twice. So that's why it is making the process a little longer.
One of the **advantages** of one bus organization is that it is one of the simplest and also this is very cheap to implement.
At the same time, there is a **disadvantage** that it has only one bus. This "one bus" is accessed by all general-purpose registers, program counter, instruction register, MAR, and MDR, making every operation sequential. No one recommends this architecture nowadays.

Two Bus organizations

This overcame the disadvantage of one bus organization, and another architecture was developed known as two bus organization. In two bus organizations, there are two buses, and the general-purpose register can read/write from both the buses. In this case, two operands can be fetched at the same time because of the two buses.

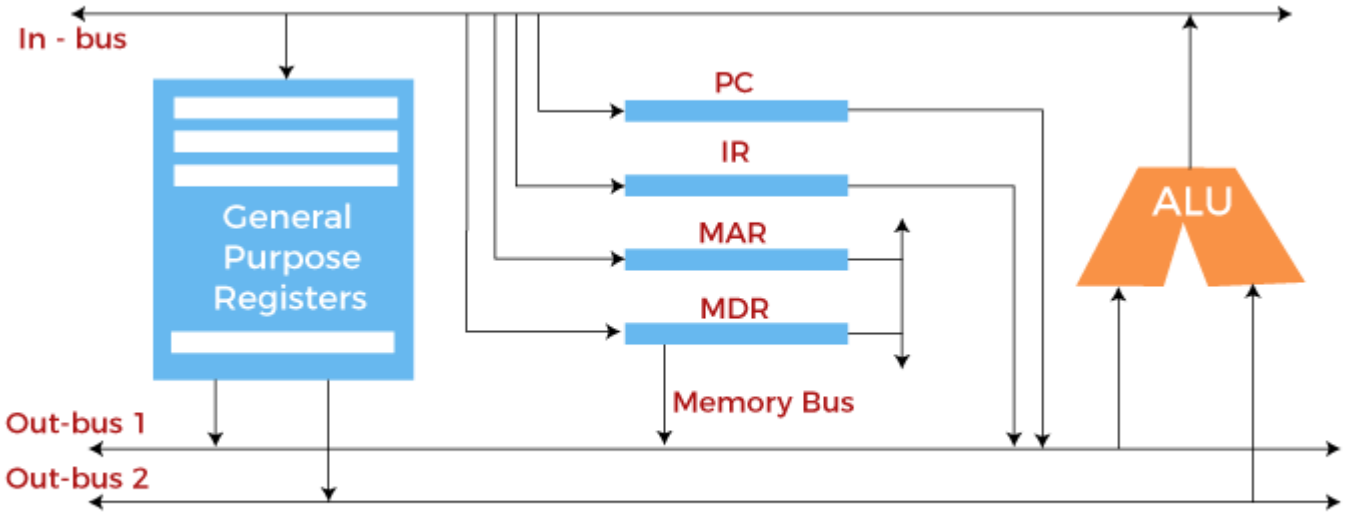


Two Bus Datapath

One bus fetch operand for ALU and another bus fetch for register. The situation arises when both buses are busy fetching operands, the output can be stored in a temporary register. When the buses are free, then the particular output can be dropped on the buses.

There are two versions of two bus organizations, i.e., in-bus and out-bus. From in-bus, the general-purpose register can read data, and to the out bus, the general-purpose registers can write data. Here buses get dedicated.

Three Bus organization



Three Bus Datapath

In three bus organizations, we have three buses, OUT bus1, OUT bus2, and an IN bus. From the out buses, we can get the operand that can come from the general-purpose register and evaluated in ALU, and the output is dropped on In Bus to be sent to respective registers.

This implementation is complex but faster because two operands can flow into ALU and out of ALU in parallel. It was developed to overcome the "busy waiting" problem of two bus organizations.

In this structure, after execution, the output can be dropped on the bus without waiting because of an extra bus.
The main ***advantages*** of multiple bus organizations over the single bus are:

- Increase in size of the registers.
- Reduction in the number of cycles for execution.
- Increases the speed of execution, or we can say faster execution.

Instruction Set Architecture(ISA)

- An Instruction Set Architecture (ISA) is **part of the abstract model of a computer that defines how the CPU is controlled by the software**.
- The ISA acts as an interface between the hardware and the software, specifying both what the processor is capable of doing as well as how it gets done.

Contents

- How a program executes

- **MIPS instructions**
- **MIPS memory organization**

- **Data operations**
  - Add, sub, etc.
- **Data transfers**
  - Store/load data to/from memory
- **Sequencing**
  - Jump, branch if equal

Programming a processor

C Program

```
while (i != 2) {
  i = i + 1;
}
```

Compiler

Assembly Code

```
load r0 mem[7]
loop:
  r1 = r0 - 2
  j_zero r1 done
  r0 = r0 + 1
  jump loop
done:
```

OS Loader

Memory

0	load r0 mem[7]
1	r1 = r0 - 2
2	j_zero r1 *5 (done)*
3	r0 = r0 + 1
4	jump *1 (loop)*
5	
6	
7	0

Programming a processor

C Program

```
while (i != 2) {
  i = i + 1;
}
```

Compiler

Assembly Code

```
load r0 mem[7]
loop:
  r1 = r0 - 2
  j_zero r1 done
  r0 = r0 + 1
  jump loop
done:
```

Compiler put variable i in register 0.

OS Loader

Memory

0	load r0 mem[7]
1	r1 = r0 - 2
2	j_zero r1 *5 (done)*
3	r0 = r0 + 1
4	jump *1 (loop)*
5	
6	
7	0

Programming a processor

C Program

```
while (i != 2) {
  i = i + 1;
}
```

Compiler

Assembly Code

```
load r0 mem[7]
loop:
  r1 = r0 - 2
  j_zero r1 done
  r0 = r0 + 1
  jump loop
done:
```

Check if i==2 by sub 2 and check if zero.

OS Loader

Memory

0	load r0 mem[7]
1	r1 = r0 - 2
2	j_zero r1 *5 (done)*
3	r0 = r0 + 1
4	jump *1 (loop)*
5	
6	
7	0

Programming a processor

C Program

```
while (i != 2) {
  i = i + 1;
}
```

Compiler

Assembly Code

```
load r0 mem[7]
loop:
  r1 = r0 - 2
  j_zero r1 done
  r0 = r0 + 1
  jump loop
done:
```

Check if i==2 by sub 2 and check if zero.

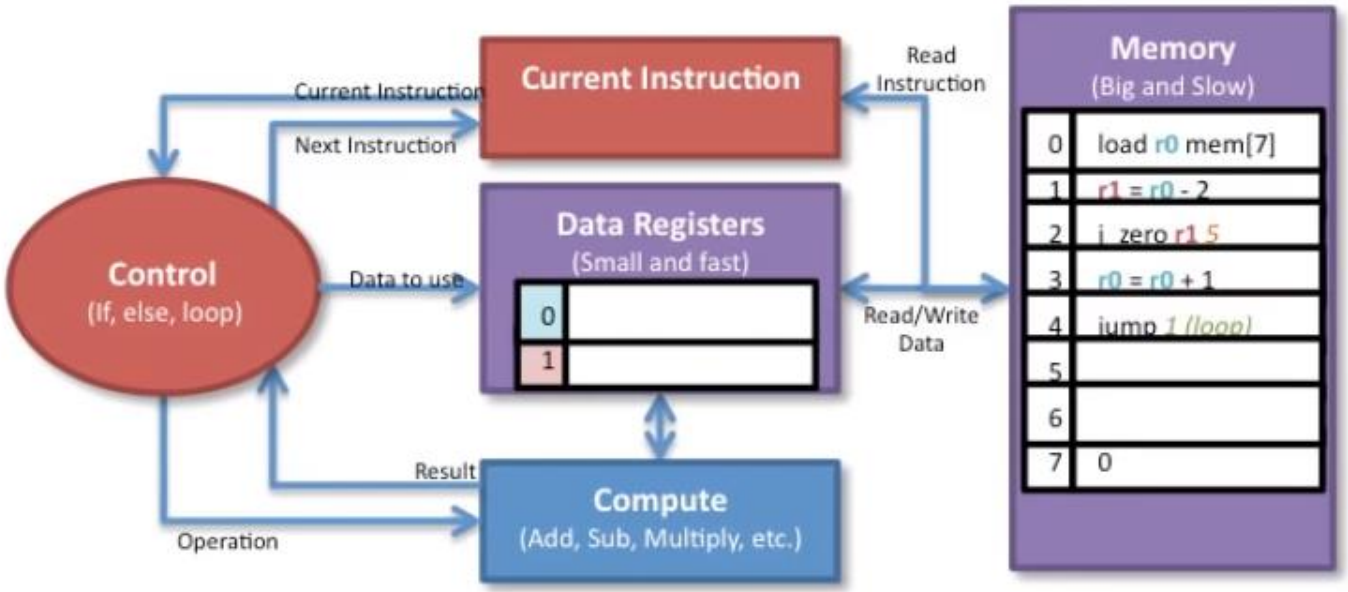
OS Loader

Memory

0	load r0 mem[7]
1	r1 = r0 - 2
2	j_zero r1 *5 (done)*
3	r0 = r0 + 1
4	jump *1 (loop)*
5	
6	
7	0

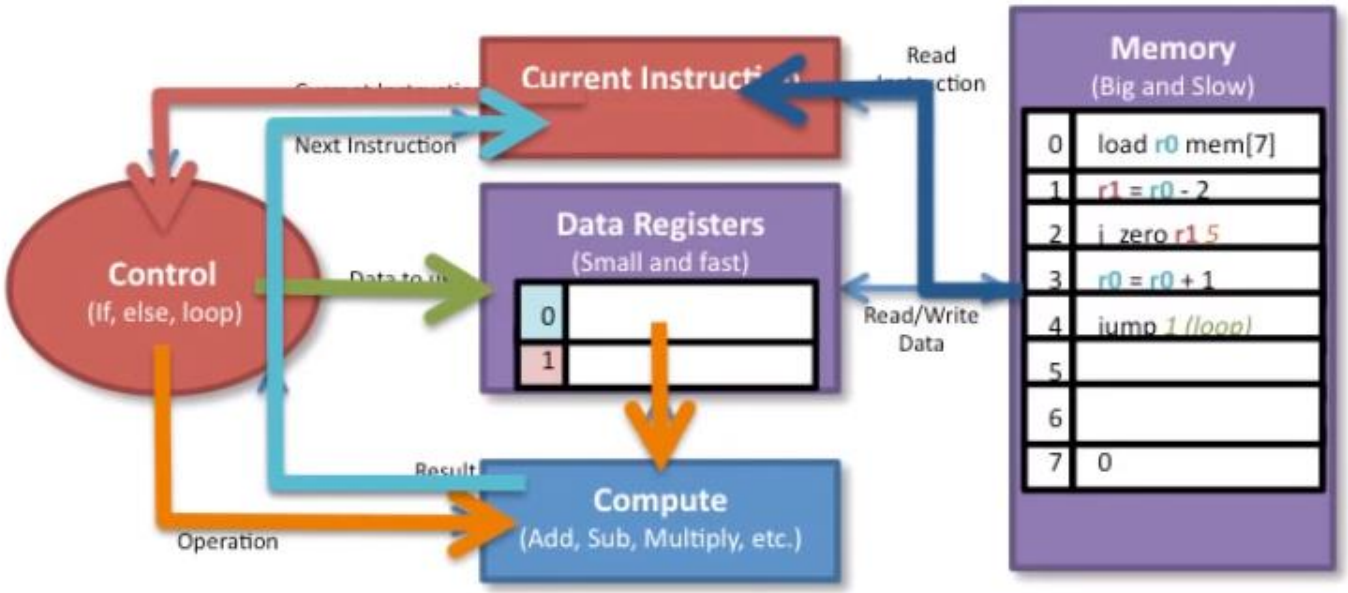
Walking through program execution

- What will the processor do?



Walking through program execution

- What will the processor do?
  - 1. Load the instruction
  - 2. Figure out what operation to do
  - 3. Figure out what data to use
  - 4. Do the computation
  - 5. Figure out next instruction
- Repeat this over and over and over…



MIP Instructions

Example MIPS instructions

- General 3-operand format:
  - *op* dest, src1, src2 — dest ← src1 *op* src2

dest, src1, src2 are registers

- Addition
  - add a, b, c — a ← b + c
  - addi a, b, 12 — a ← b + 12

The "i" in addi is for immediate

- Subtraction
  - sub a, b, c — a ← b - c

- Complex: f = (g + h) - (i + j)
  - add t0, g, h — t0 ← g + h
  - add t1, i, j — t1 ← i + j
  - sub f, t0, t1 — f ← t0 - t1

Complex operations generate many instructions with temporary values.

Types of instructions

- Data operations
- Data transfer
- Sequencing

```
load r0 mem[7]
loop:
  r1 = r0 - 2
  j_zero r1 done
  r0 = r0 + 1
  jump loop
done:
```

Function	Instruction	Effect
add	add R1, R2, R3	R1 = R2 + R3
sub	sub R1, R2, R3	R1 = R2 – R3
add immediate	addi R1, R2, 145	R1 = R2 + 145
multiply	mult R2, R3	hi, lo = R1 * R2
divide	div R2, R3	low = R2/R3, hi = remainder
and	and R1, R2, R3	R1 = R2 & R3
or	or R1, R2, R3	R1 = R2 | R3
and immediate	andi R1, R2, 145	R1 = R2 & 143
or immediate	ori R1, R2, 145	R1 = R2 | 145
shift left logical	sll R1, R2, 7	R1 = R2 << 7
shift right logical	srl R1, R2, 7	R1 = R2 >> 7
load word	lw R1, 145(R2)	R1 = memory[R2 + 145]
store word	sw R1, 145(R2)	memory[R2 + 145] = R1
load upper immediate	lui R1, 145	R1 = 145 << 16
branch on equal	beq R1, R2, 145	if (R1 == R2) go to PC + 4 + 145*4
branch on not equal	bne R1, R2, 145	if (R1 != R2) go to PC + 4 + 145*4
set on less than	slt R1, R2, R3	if (R2 < R3) R1 = 1, else R1 = 0
set less than immediate	slti R1, R2, 145	if (R2 < 145) R1 = 1, else R1 = 0
jump	j 145	go to 145
jump register	jr R31	go to R31
jump and link	jal 145	R31 = PC + 4; go to 145

(Complete table is printed in the book for reference.)

Types of instructions

- **Data operations**
  - Arithmetic (add, subtract, ...)
  - Logical (and, or, not, xor, ...)
- **Data transfer**
- **Sequencing**

```
load r0 mem[7]
loop:
  r1 = r0 - 2
  j_zero r1 done
  r0 = r0 + 1
  jump loop
done:
```

Function	Instruction	Effect
add	add R1, R2, R3	R1 = R2 + R3
sub	sub R1, R2, R3	R1 = R2 - R3
add immediate	addi R1, R2, 145	R1 = R2 + 145
multiply	mult R2, R3	hi, lo = R1 * R2
divide	div R2, R3	low = R2/R3, hi = remainder
and	and R1, R2, R3	R1 = R2 & R3
or	or R1, R2, R3	R1 = R2 | R3
and immediate	andi R1, R2, 145	R1 = R2 & 143
or immediate	ori R1, R2, 145	R1 = R2 | 145
shift left logical	sll R1, R2, 7	R1 = R2 << 7
shift right logical	srl R1, R2, 7	R1 = R2 >> 7
load word	lw R1, 145(R2)	R1 = memory[R2 + 145]
store word	sw R1, 145(R2)	memory[R2 + 145] = R1
load upper immediate	lui R1, 145	R1 = 145 << 16
branch on equal	beq R1, R2, 145	if (R1 == R2) go to PC + 4 + 145*4
branch on not equal	bne R1, R2, 145	if (R1 != R2) go to PC + 4 + 145*4
set on less than	slt R1, R2, R3	if (R2 < R3) R1 = 1, else R1 = 0
set less than immediate	slti R1, R2, 145	if (R2 < 145) R1 = 1, else R1 = 0
jump	j 145	go to 145
jump register	jr R31	go to R31
jump and link	jal 145	R31 = PC + 4; go to 145

(Complete table is printed in the book for reference.)

Types of instructions

- **Data operations**
  - Arithmetic (add, subtract, …)
  - Logical (and, or, not, xor, …)
- **Data transfer**
  - Load (memory → register)
  - Store (register → memory)
- **Sequencing**

```
load r0 mem[7]
loop:
  r1 = r0 - 2
  j_zero r1 done
  r0 = r0 + 1
  jump loop
done:
```

Function	Instruction	Effect
add	add R1, R2, R3	R1 = R2 + R3
sub	sub R1, R2, R3	R1 = R2 - R3
add immediate	addi R1, R2, 145	R1 = R2 + 145
multiply	mult R2, R3	hi, lo = R1 * R2
divide	div R2, R3	low = R2/R3, hi = remainder
and	and R1, R2, R3	R1 = R2 & R3
or	or R1, R2, R3	R1 = R2 | R3
and immediate	andi R1, R2, 145	R1 = R2 & 143
or immediate	ori R1, R2, 145	R1 = R2 | 145
shift left logical	sll R1, R2, 7	R1 = R2 << 7
shift right logical	srl R1, R2, 7	R1 = R2 >> 7
load word	lw R1, 145(R2)	R1 = memory[R2 + 145]
store word	sw R1, 145(R2)	memory[R2 + 145] = R1
load upper immediate	lui R1, 145	R1 = 145 << 16
branch on equal	beq R1, R2, 145	if (R1 == R2) go to PC + 4 + 145*4
branch on not equal	bne R1, R2, 145	if (R1 != R2) go to PC + 4 + 145*4
set on less than	slt R1, R2, R3	if (R2 < R3) R1 = 1, else R1 = 0
set less than immediate	slti R1, R2, 145	if (R2 < 145) R1 = 1, else R1 = 0
jump	j 145	go to 145
jump register	jr R31	go to R31
jump and link	jal 145	R31 = PC + 4; go to 145

(Complete table is printed in the book for reference.)

Types of instructions

- **Data operations**
  - Arithmetic (add, subtract, ...)
  - Logical (and, or, not, xor, ...)
- **Data transfer**
  - Load (memory → register)
  - Store (register → memory)
- **Sequencing**
  - Branch (conditional, e.g., <, >, ==)
  - Jump (unconditional, e.g., goto)

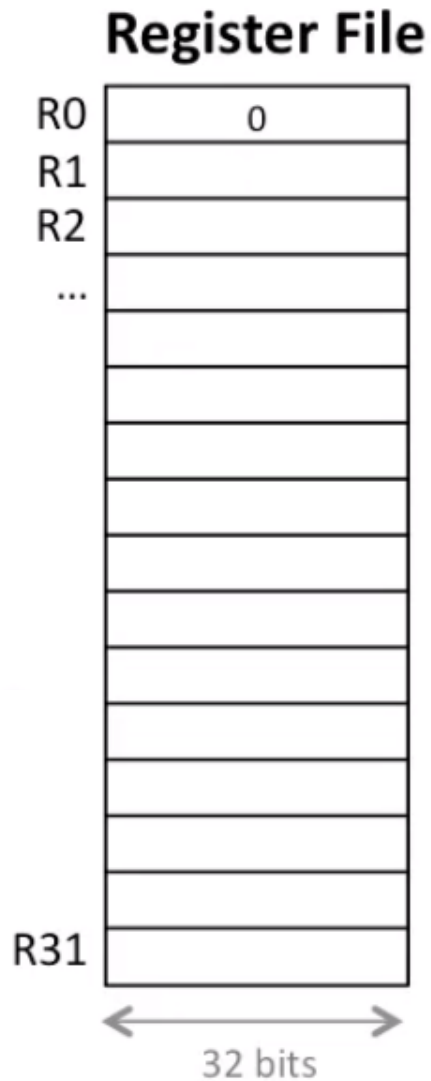
```
load r0 mem[7]
loop:
  r1 = r0 - 2
  j_zero r1 done
  r0 = r0 + 1
  jump loop
done:
```

Function	Instruction	Effect
add	add R1, R2, R3	R1 = R2 + R3
sub	sub R1, R2, R3	R1 = R2 - R3
add immediate	addi R1, R2, 145	R1 = R2 + 145
multiply	mult R2, R3	hi, lo = R1 * R2
divide	div R2, R3	low = R2/R3, hi = remainder
and	and R1, R2, R3	R1 = R2 & R3
or	or R1, R2, R3	R1 = R2 | R3
and immediate	andi R1, R2, 145	R1 = R2 & 143
or immediate	ori R1, R2, 145	R1 = R2 | 145
shift left logical	sll R1, R2, 7	R1 = R2 << 7
shift right logical	srl R1, R2, 7	R1 = R2 >> 7
load word	lw R1, 145(R2)	R1 = memory[R2 + 145]
store word	sw R1, 145(R2)	memory[R2 + 145] = R1
load upper immediate	lui R1, 145	R1 = 145 << 16
branch on equal	beq R1, R2, 145	if (R1 == R2) go to PC + 4 + 145*4
branch on not equal	bne R1, R2, 145	if (R1 != R2) go to PC + 4 + 145*4
set on less than	slt R1, R2, R3	if (R2 < R3) R1 = 1, else R1 = 0
set less than immediate	slti R1, R2, 145	if (R2 < 145) R1 = 1, else R1 = 0
jump	j 145	go to 145
jump register	jr R31	go to R31
jump and link	jal 145	R31 = PC + 4; go to 145

(Complete table is printed in the book for reference.)

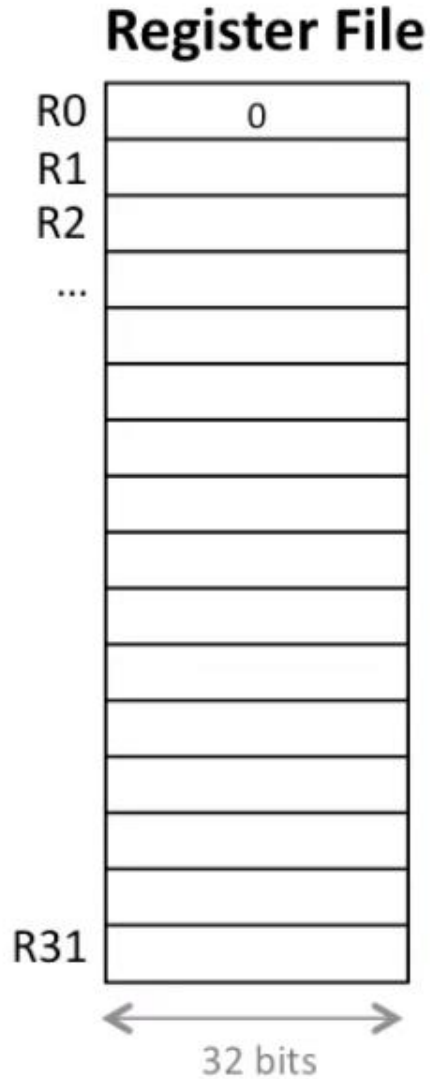
Registers in MIPS

- **32 General Purpose Registers**
  - R0…R31 or $0…$31
  - Each is 32 bits wide
  - Values for instructions must come from registers
- **Some are special**
  - **R0** is always **zero**
  - **R29/R31** are used for function calls
- **A few special registers**
  - **PC (Program Counter)**: current instruction
  - Hi & Lo results of multiplication
  - Floating point registers
  - Control registers (for errors and status)



Registers in MIPS

- **32 General Purpose Registers**
  - R0…R31 or $0…$31
  - Each is 32 bits wide
  - Values for instructions must come from registers
- **Some are special**
  - **R0** is always zero
  - **R29/R31** are used for function calls
- **A few special registers**
  - PC (Program Counter): current instruction
  - Hi & Lo results of multiplication
  - Floating point registers
  - Control registers (for errors and status)

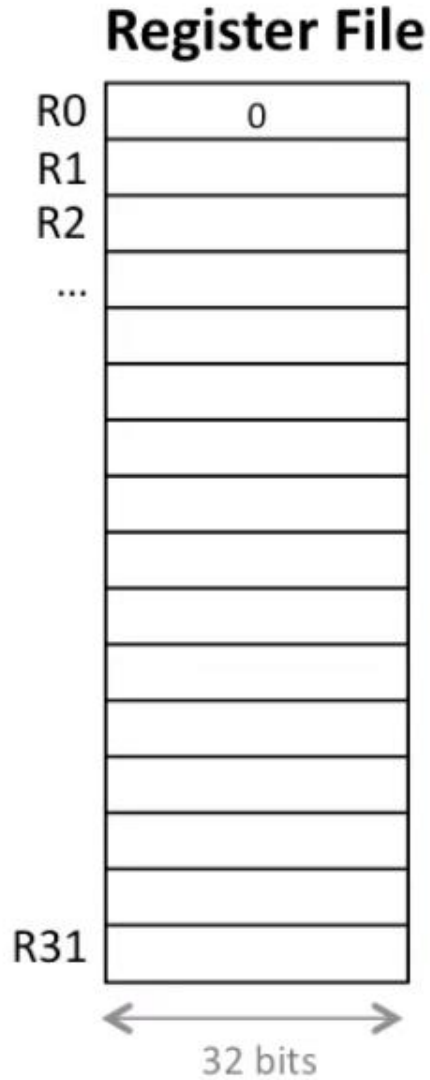


Q: What does add r3, r1, r0 do?

- Put r0+r3 → r1
- Put r1+r3 → r0
- Move r1 → r3

Registers in MIPS

- **32 General Purpose Registers**
  - R0…R31 or $0…$31
  - Each is 32 bits wide
  - Values for instructions must come from registers
- **Some are special**
  - **R0** is always zero
  - **R29/R31** are used for function calls
- **A few special registers**
  - PC (Program Counter): current instruction
  - Hi & Lo results of multiplication
  - Floating point registers
  - Control registers (for errors and status)



Q: What does add r3, r1, r0 do?

- Put r0+r3 → r1
- Put r1+r3 → r0
- Move r1 → r3

A: Move r1 → r3

add r3, r1, r0 has two sources (r1 and r0) and one destination (r3).
So r3 ← r1 + r0

But r0 is always 0.
This is the same as r3 ← r1+0
or Move r1 → r3.

Question: register order

Q: How do I store the results of a+b into c?
R1 = a; R2 = b; R3 = c

- add R1, R2, R0
- add R3, R2, R1
- add R2, R1, R3

Question: register order

Q: How do I store the results of a+b into c?
R1 = a; R2 = b; R3 = c

- add R1, R2, R0
- add R3, R2, R1
- add R2, R1, R3

op dest, src1, src2
dest ← src1 *op* src2

add R3, R1, R2
same as
add R3, R2, R1

A: add R3, R2, R1
We want the destination to be c (R3)
We want the sources to be a and b (R2, R1)

(Note that the order of the sources does not matter for addition because it is commutative.)

The first register is the destination (R3) and the other two registers re the sources (R2, R1).

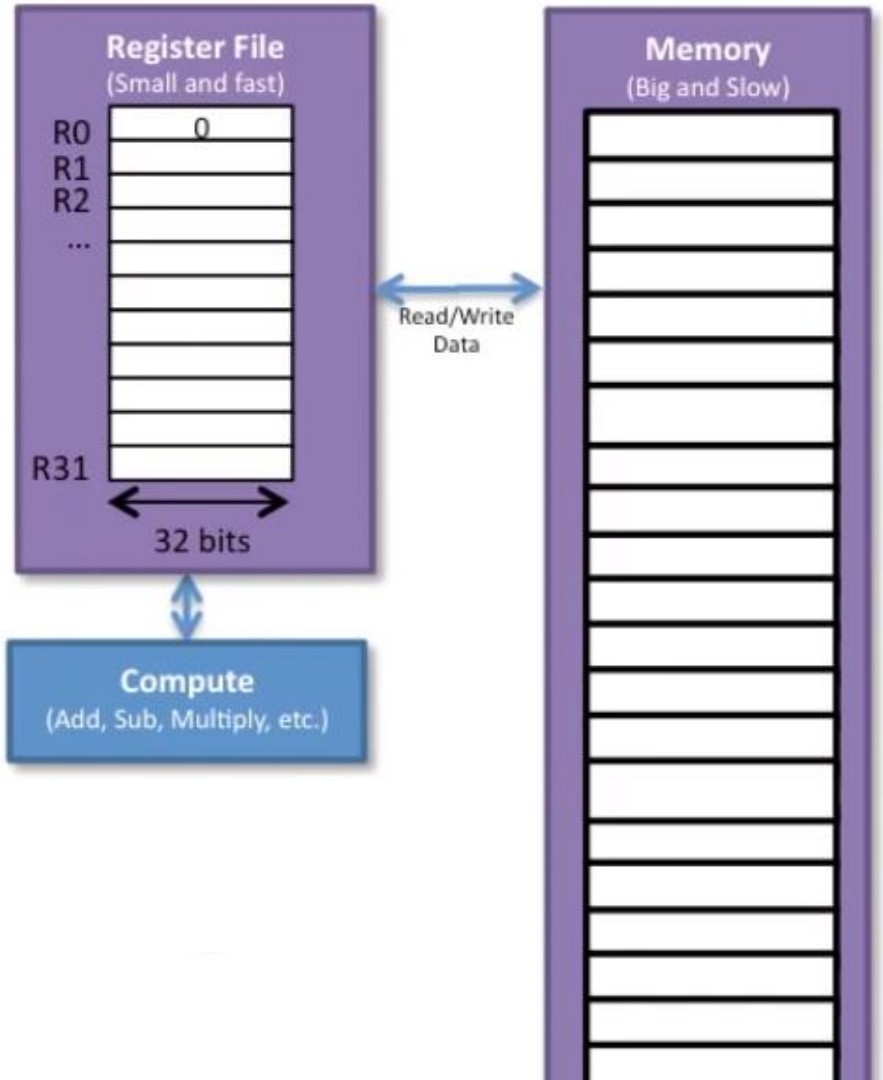
So add R3, R2, R1 gives us R3 ← R2+R1, which is c=b+a, which is the same as c=a+b.

MIPS memory organization

Register file and main memory

Memory vs. registers

- MIPS is a **Load/Store Register File** machine
  - Instructions **compute only on data in the Register File**
  - Example:
    - **add R3, R2, R1**
    - all data needs to be in the Register File
  - But we only have 32 registers in the Register File
    - Clearly not enough for a big program

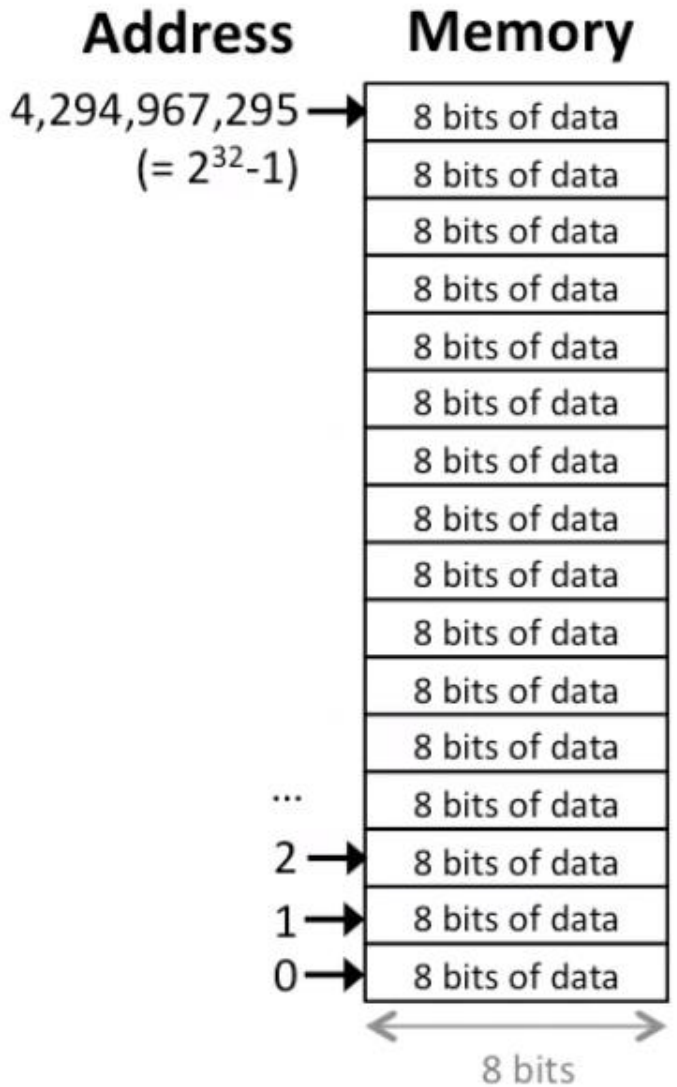


- Most data is stored in **memory** (large, but slow)
- Need to **transfer the data to the Register File** to use it
  - **Load**: load data from memory to the Register File (lw instruction)
  - **Store**: store data to the memory from the Register File (sw instruction)

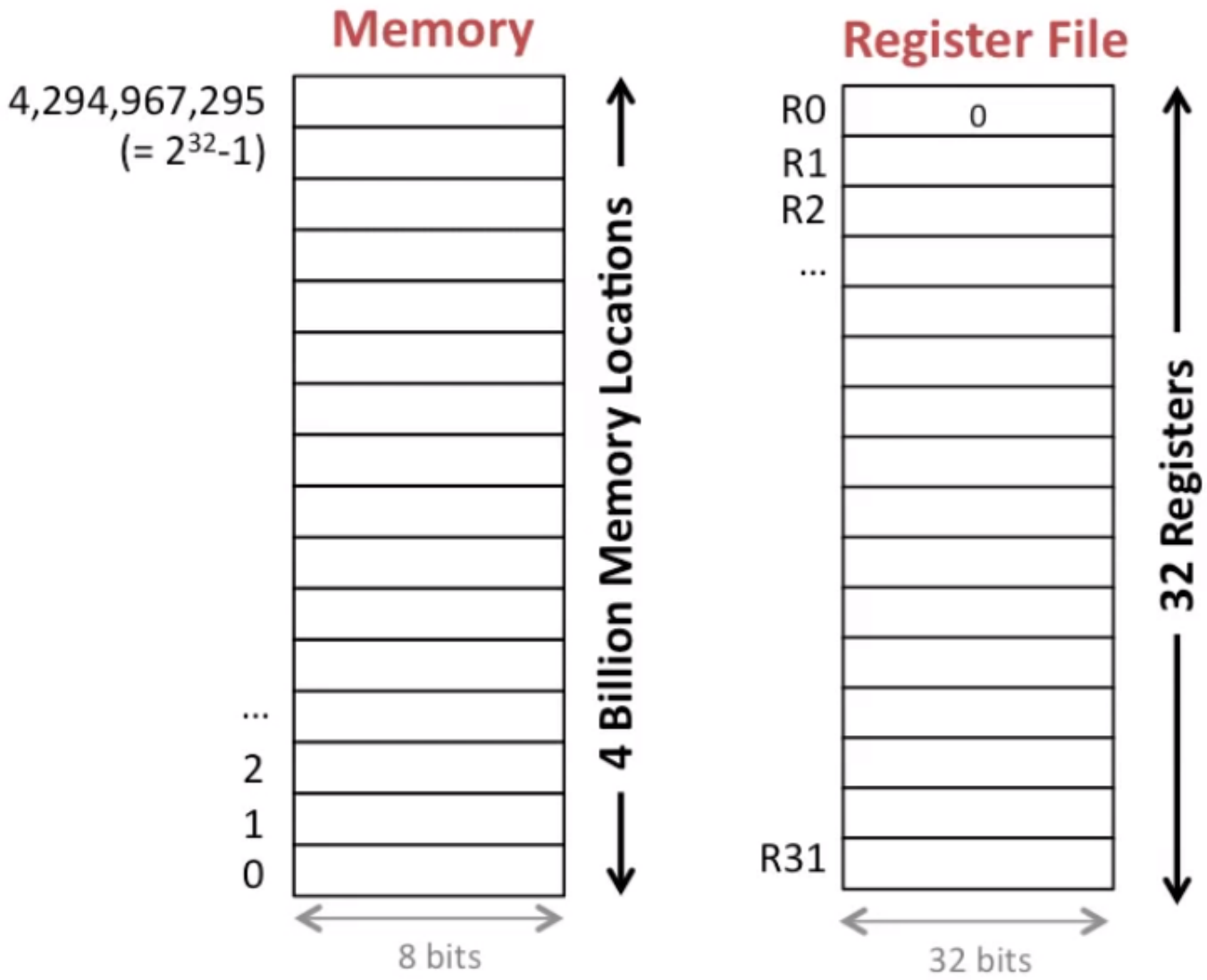
- Hence, MIPS is a **Load/Store Register File** machine

Memory organization

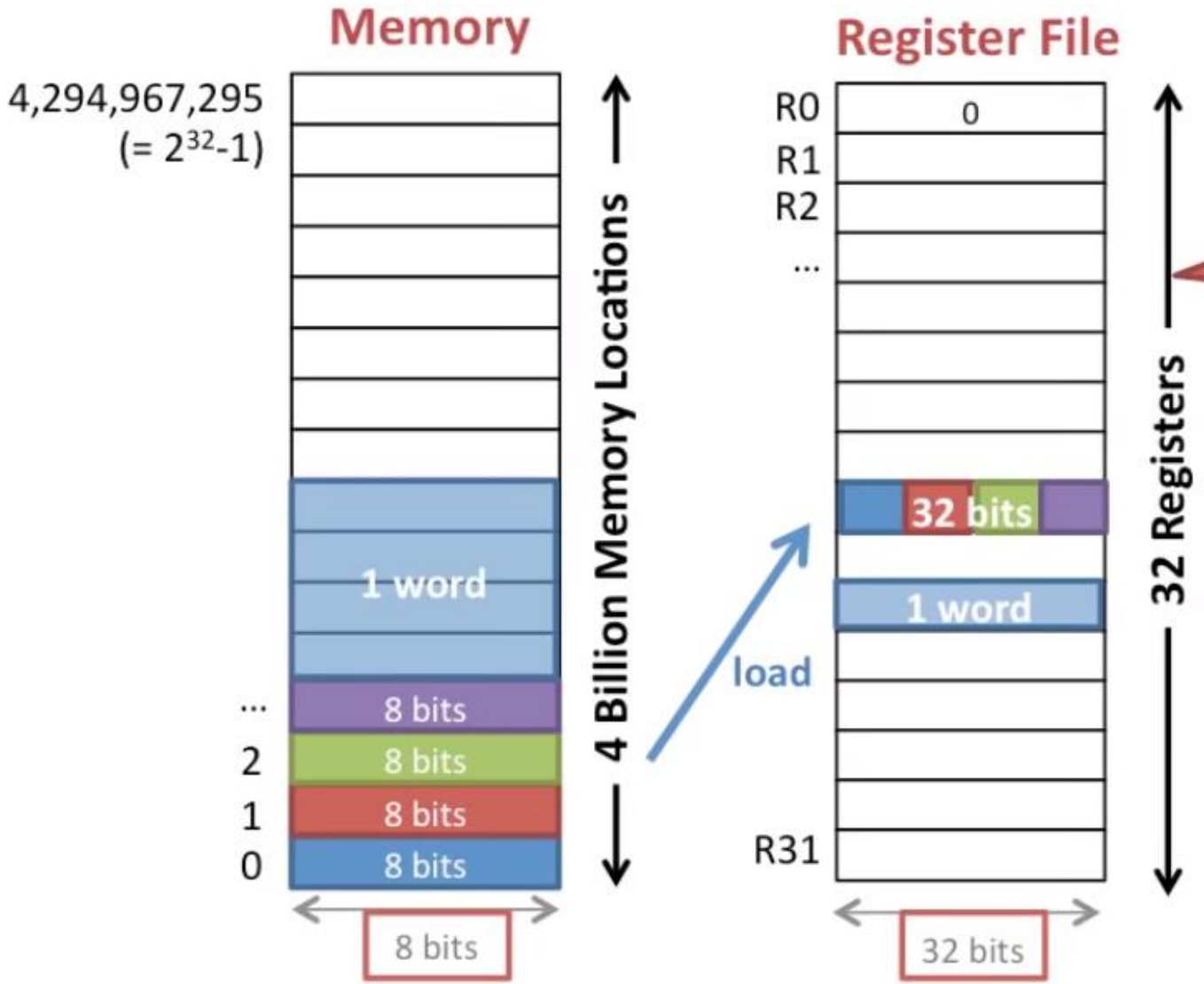
- Memory is a large 1-dimension array
- Each location is **one byte** (8 bits)
- A **memory address** indexes into the array
- For a 32-bit computer, there are 2^{32} memory locations (4GB)
- For a 64-bit computer, there are 2^{64} memory locations (16EB)



Memory and register file



Memory and register file



Q: How many memory locations do we need to fill a register in the Register File?

- 1
- 2
- 4

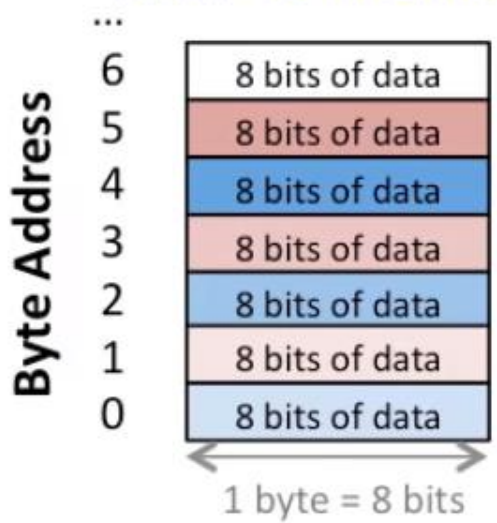
A: 4

Each entry in the register file is 32-bits. So we need 4 8-bit locations from the memory to make 32-bits for the register file.

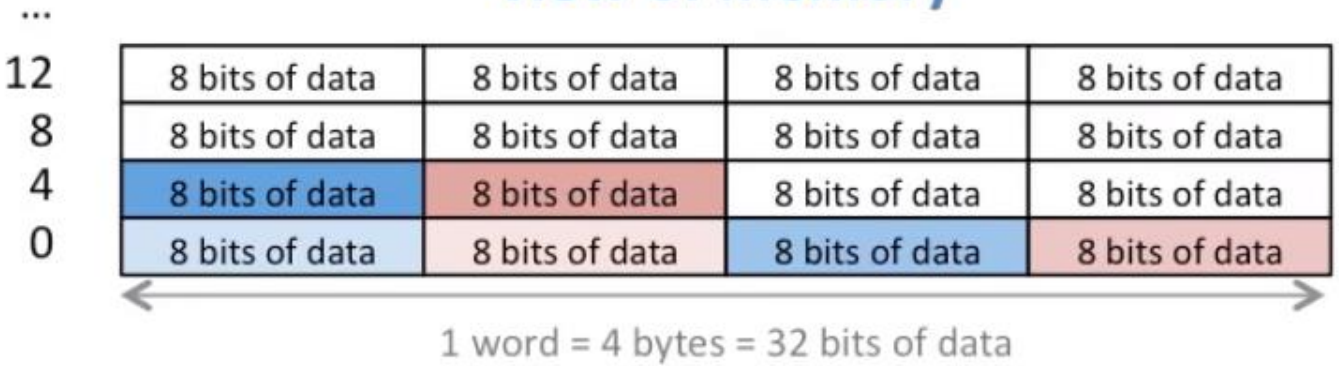
This is inconvenient.
In MIPS we mostly deal with data in 4-byte chunks (32-bits) and we call this a **word**.

Viewing memory as bytes or words

Byte-addressable view of memory



Word-aligned view of memory



Registers hold 32 bits of data (1 word)
Addresses are 32 bits of data (1 word)

- Most data in MIPS is handled in **words** not bytes
  - A **word** is 32 bits or 4 bytes
  - A **word** is the size of a register

Access alignment

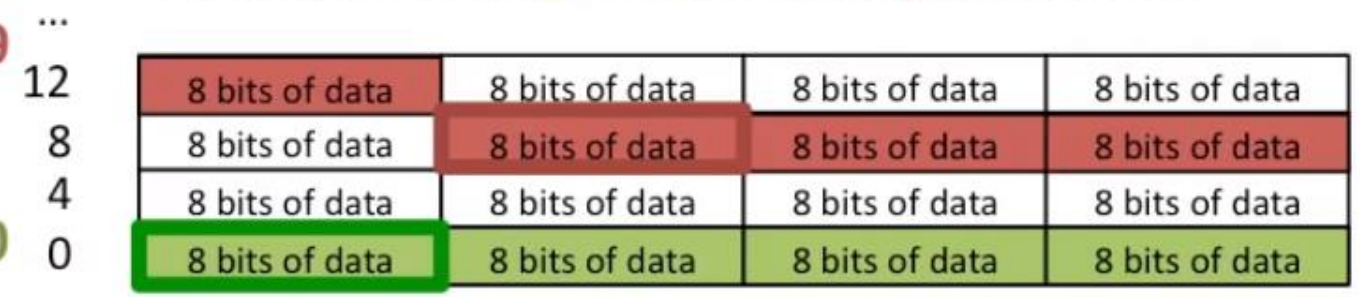
- **Aligned addresses** fall on 4 byte (word) boundaries (e.g., 0, 4, 8, 12…)
- **Unaligned addresses** do not (e.g., 1, 3, 7, 63)

- Some machines support unaligned accesses (not MIPS)

Example of aligned vs. unaligned access

Load Word at addr=9
Not Aligned

Load Word at addr=0
Aligned



Instruction execution model

How to think about instructions

Two key concepts

- Processor execution model
- Program storage model

Processor execution model

- Processor promises that the instruction execution will ***appear to be sequential and atomic***
  - **Sequential**: execute instructions in-order
  - **Atomic**: execute each instruction all at once

Why do we care?
If a program is not sequential and atomic we can't figure out what it should do. Impossible to debug and program.

- **Sequential**
  - Program says: R2 = R1 + R2 then R3 = R1 + R2
  - Processor may not do: R3 = R1 + R2 then R2 = R1 + R2 ← wrong result
- **Atomic**
  - Program says: R2 = R1 + R2 then R3 = R1 + R2
  - Processor has to **finish** R2= R1 + R2 **before starting** R3 = R1 + R2

Stored program computers

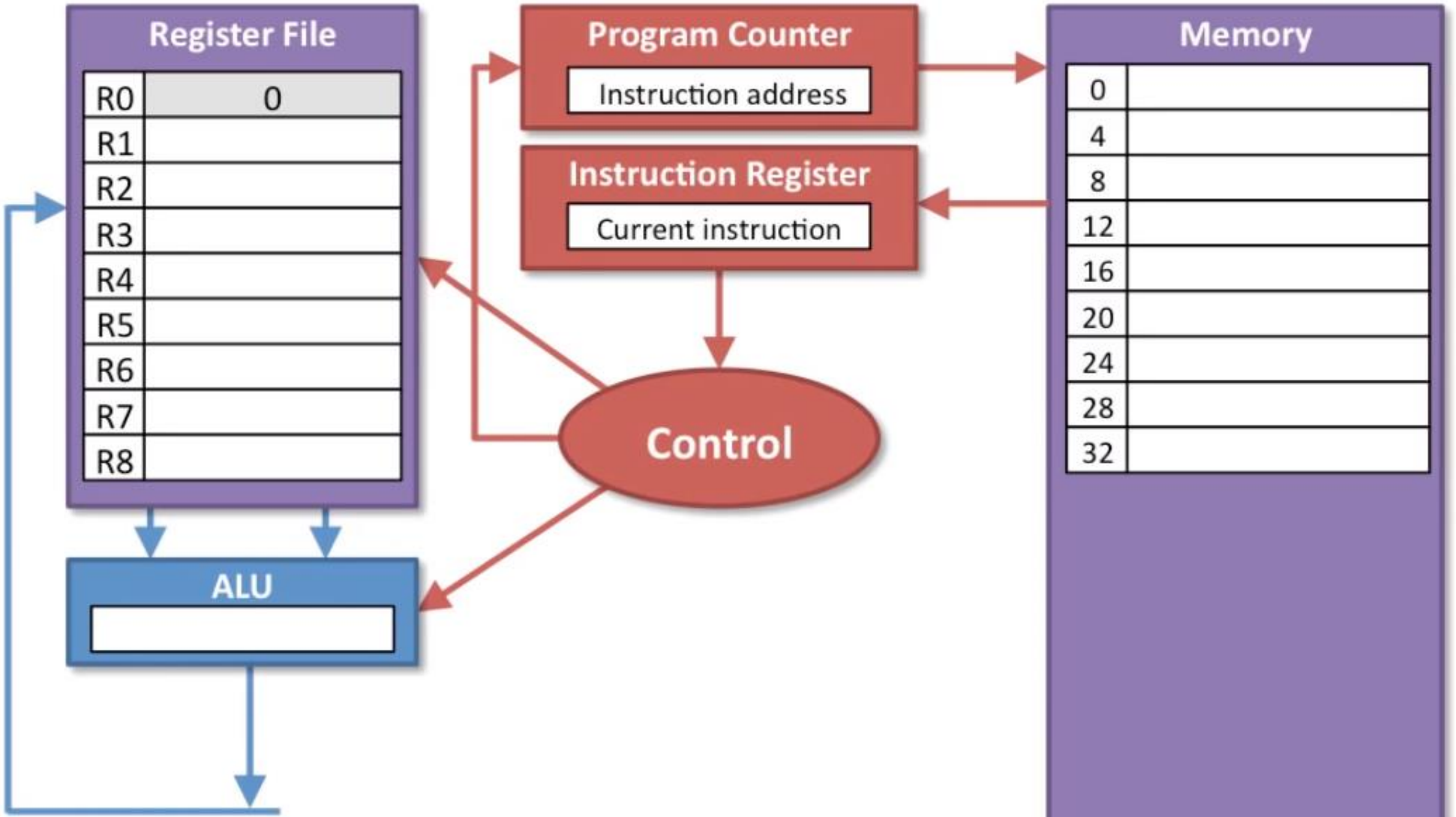
- **Program and data are stored in memory**
  - Instructions have to be *fetched (loaded)* from memory for execution
  - Data has to be *fetched (loaded)* from memory for computation

Data operations

Doing computations

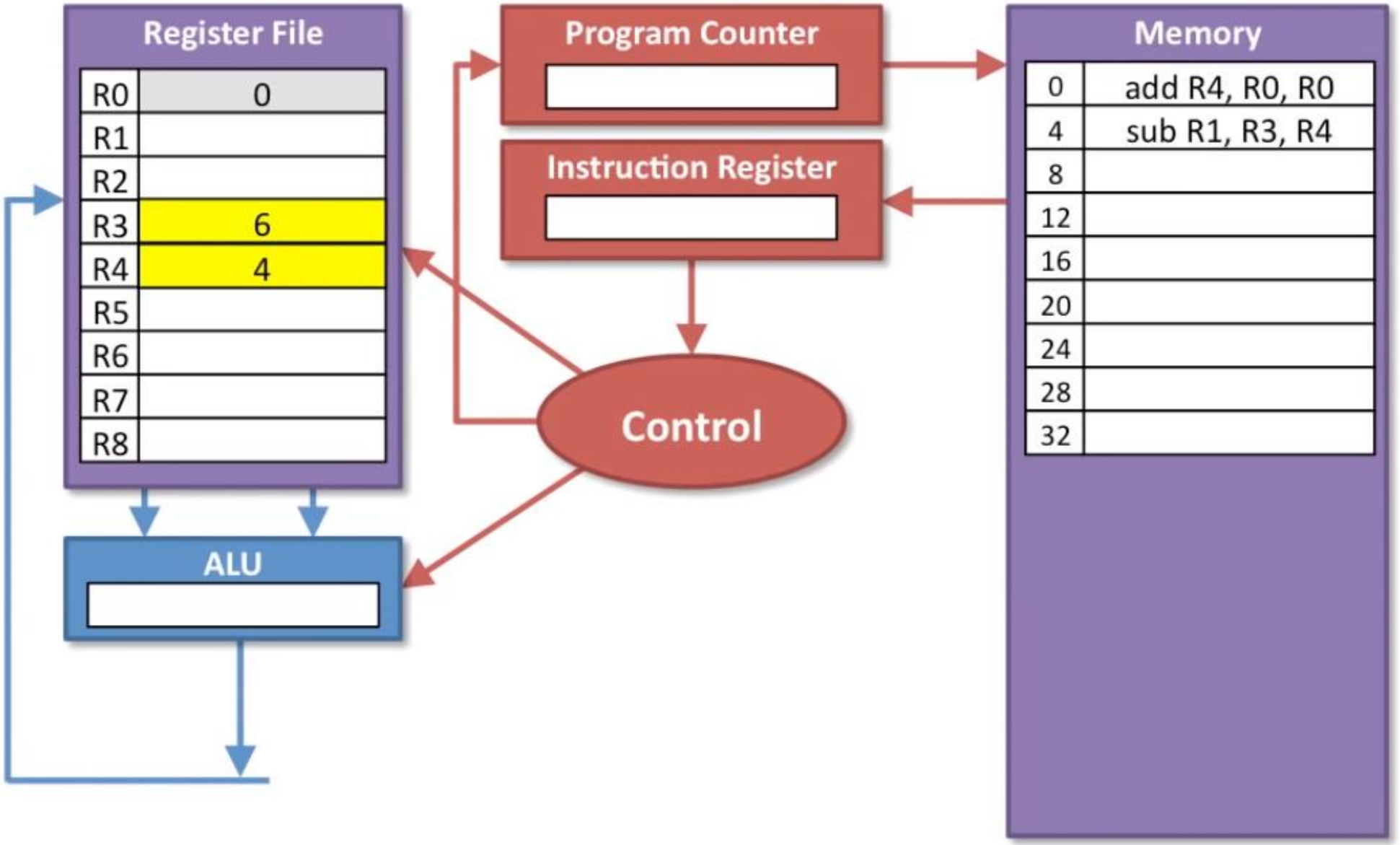
Data operations in detail

1. **Program Counter** holds the instruction address.
2. Instructions are ***fetched*** from memory into the **Instruction Register**.
3. **Control** logic ***decodes*** the instruction and tells the **ALU** and **Register File** what to do.
4. **ALU** ***executes*** the instruction and results flow back to the **Register File**.
5. The **Control** logic ***updates*** the **Program Counter** for the next instruction.

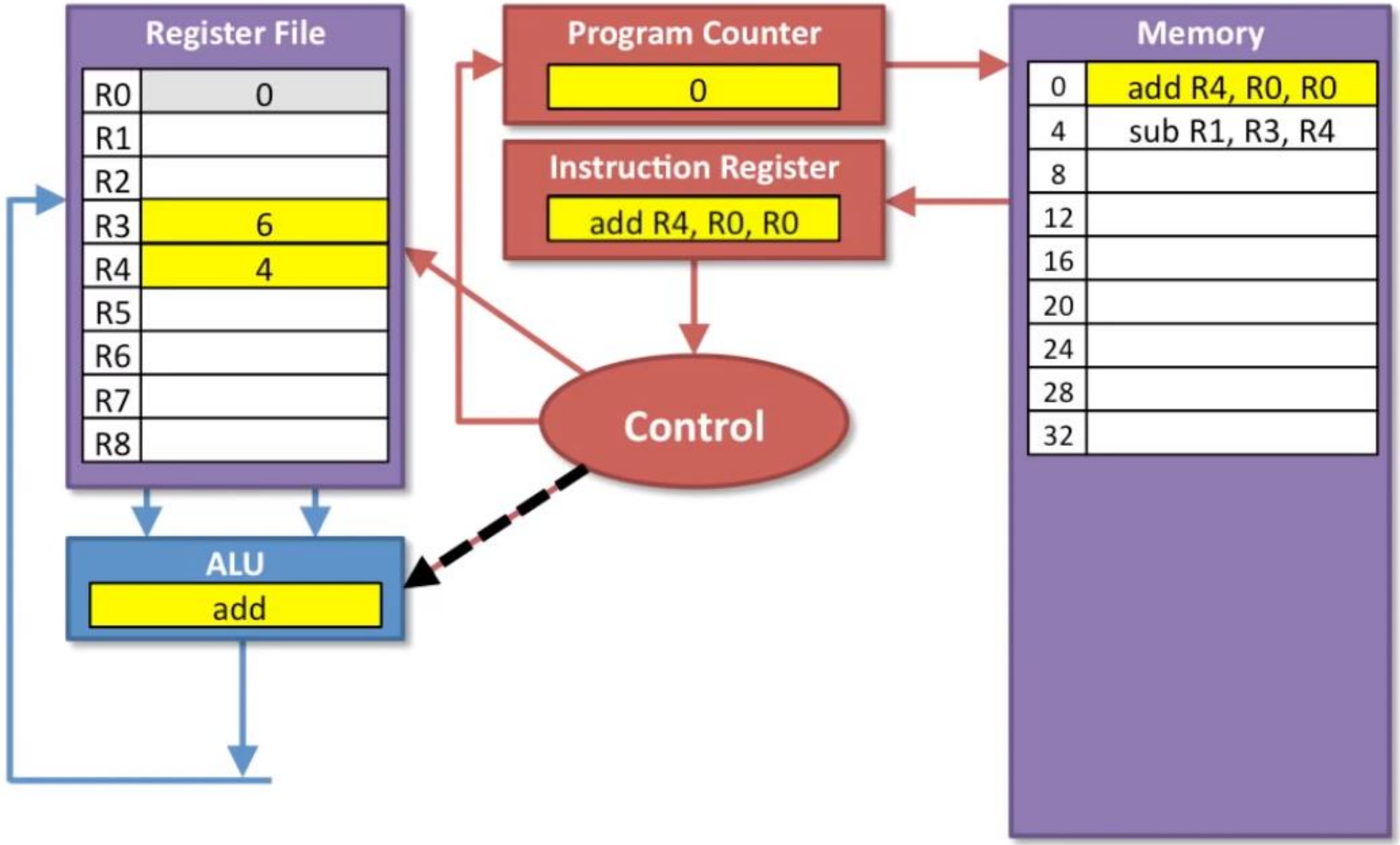


ALU = Arithmetic Logic Unit (Compute)

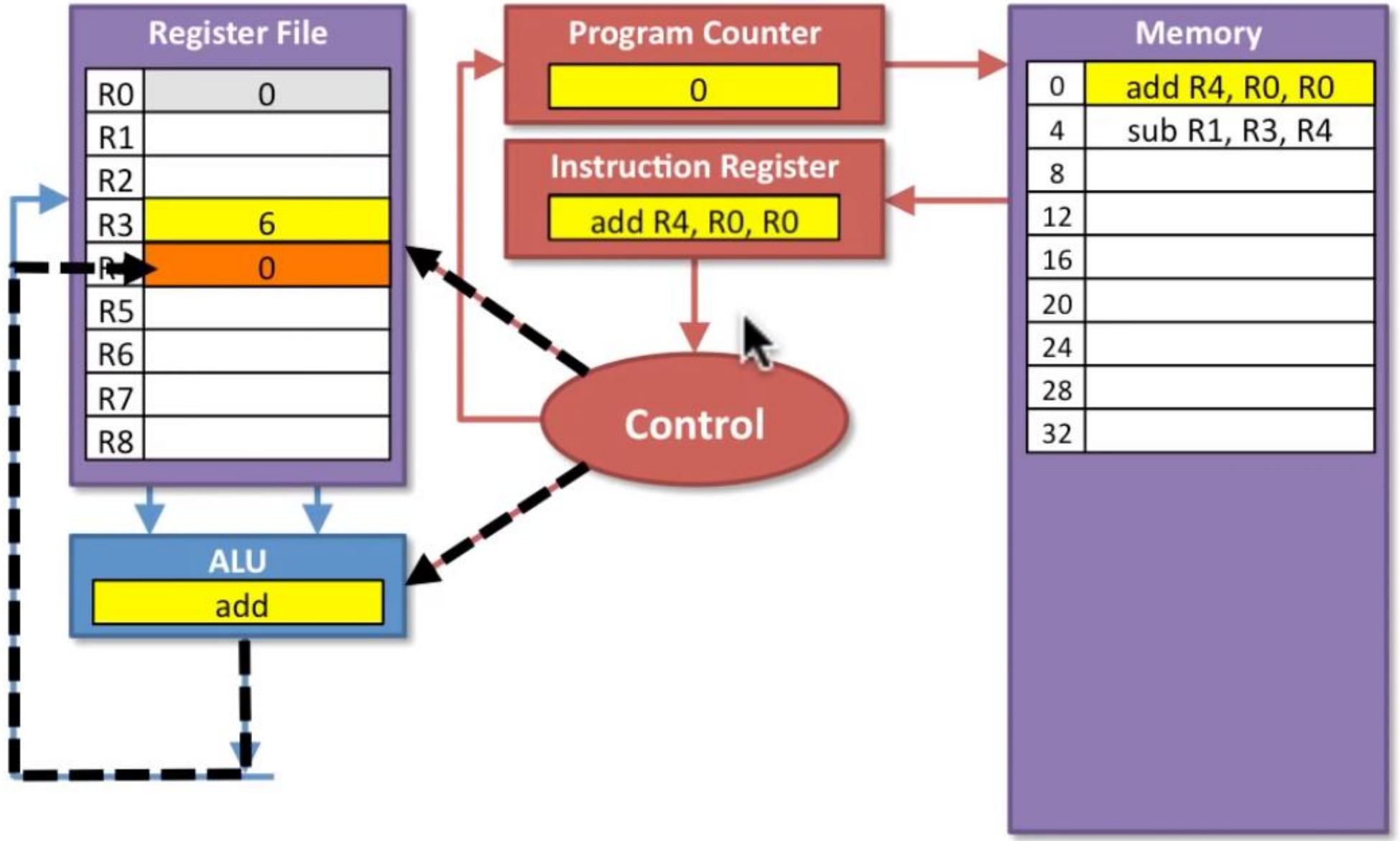
Add/sub example (1 of 2)



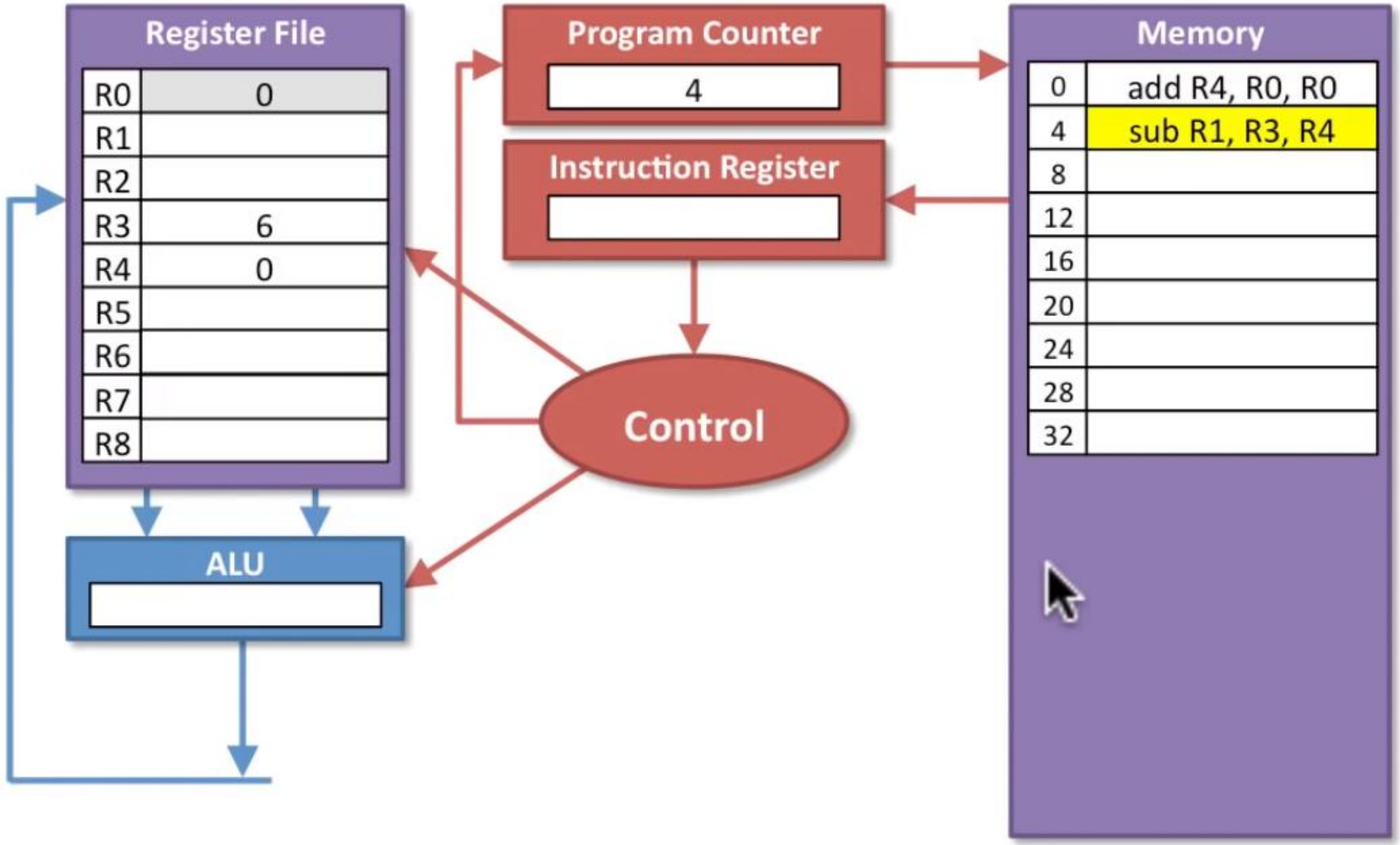
Add/sub example (1 of 2)



Add/sub example (1 of 2)



Add/sub example (2 of 2)



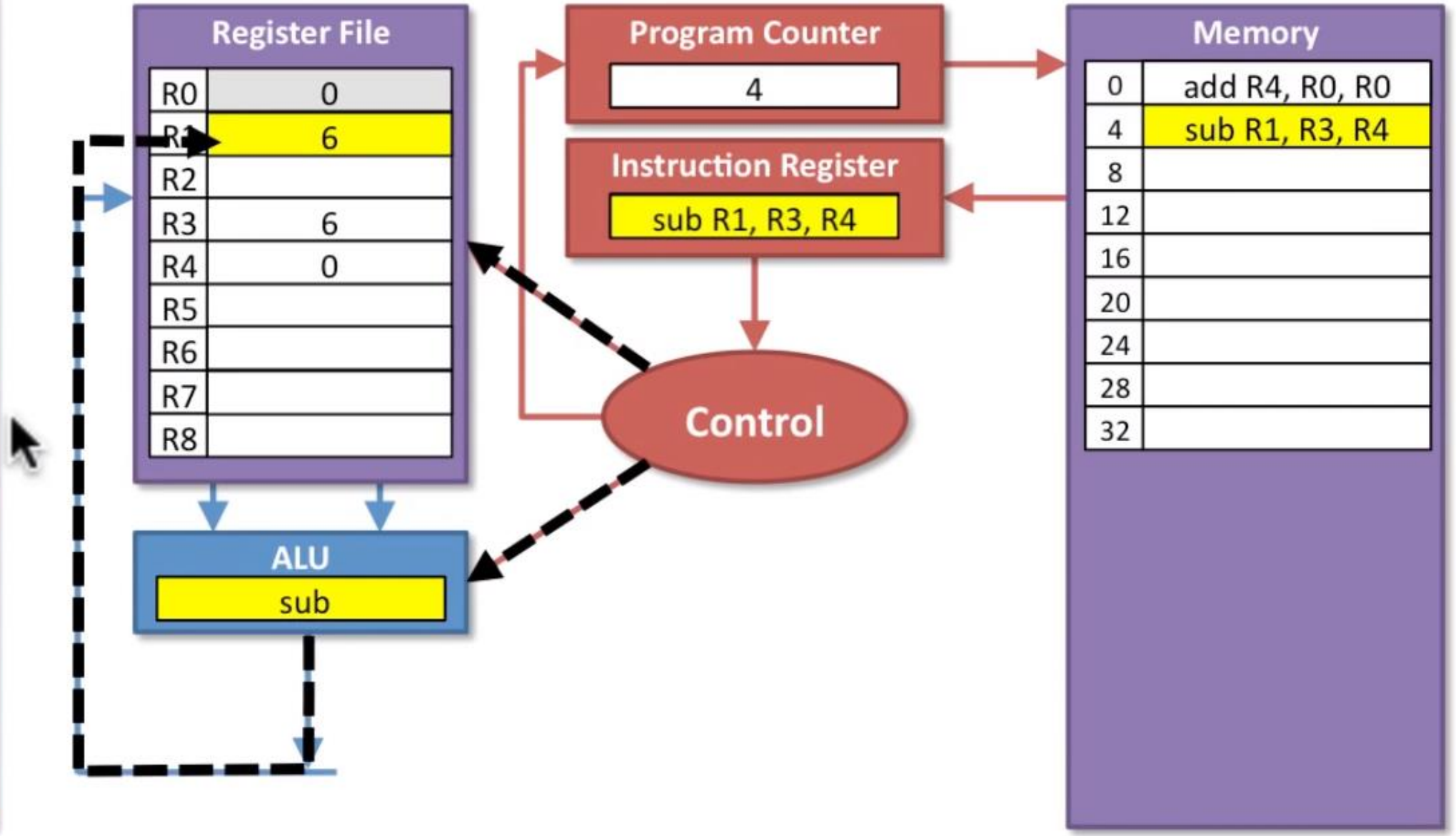
Add/sub example (2 of 2)

Program Counter (PC) specifies the address of the instruction to fetch.

The **Instruction Register** holds the current instruction.

Control tells the **ALU** and **Register File** what to do.

ALU computes the result and writes it back into the **Register File.**

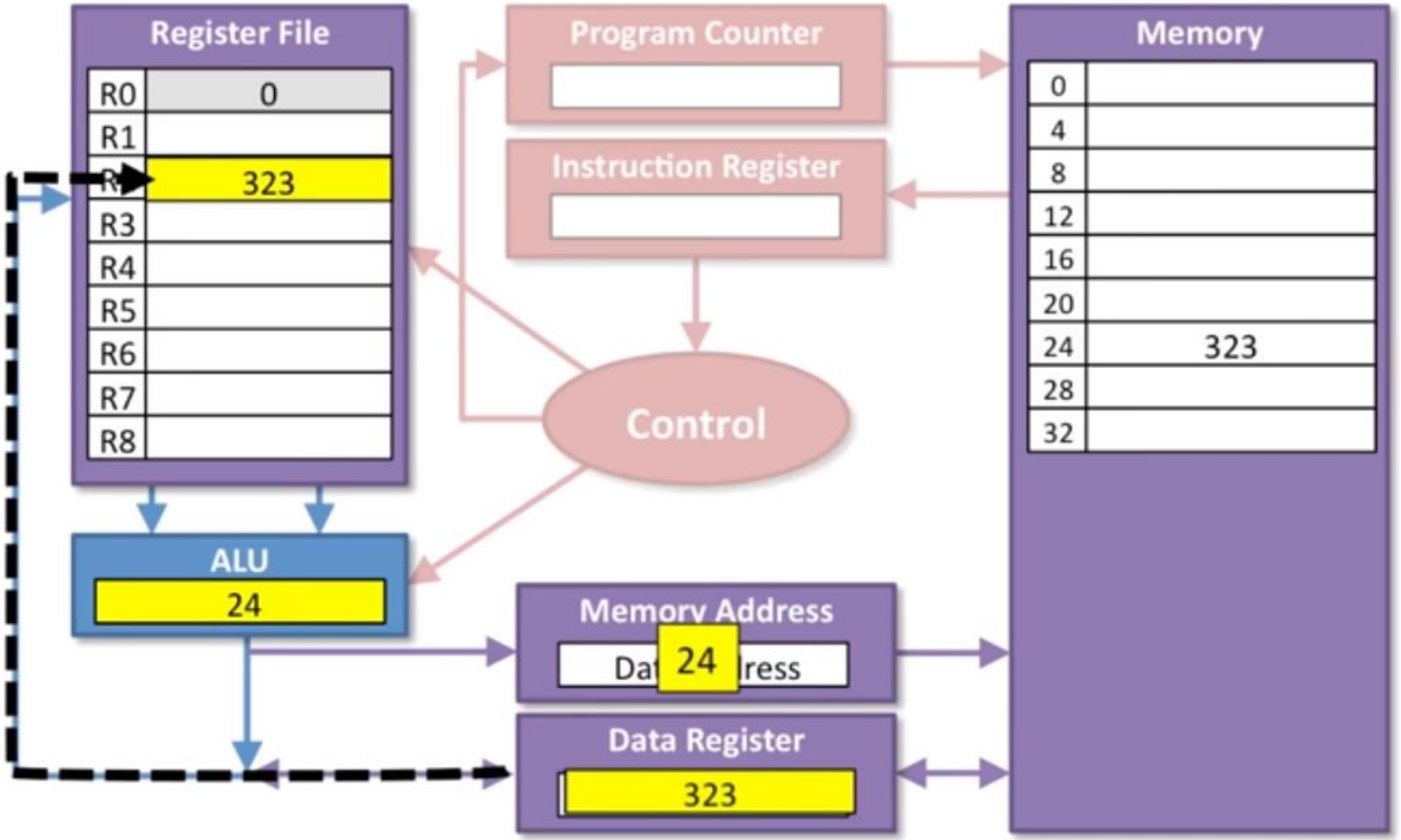


Data transfer instructions

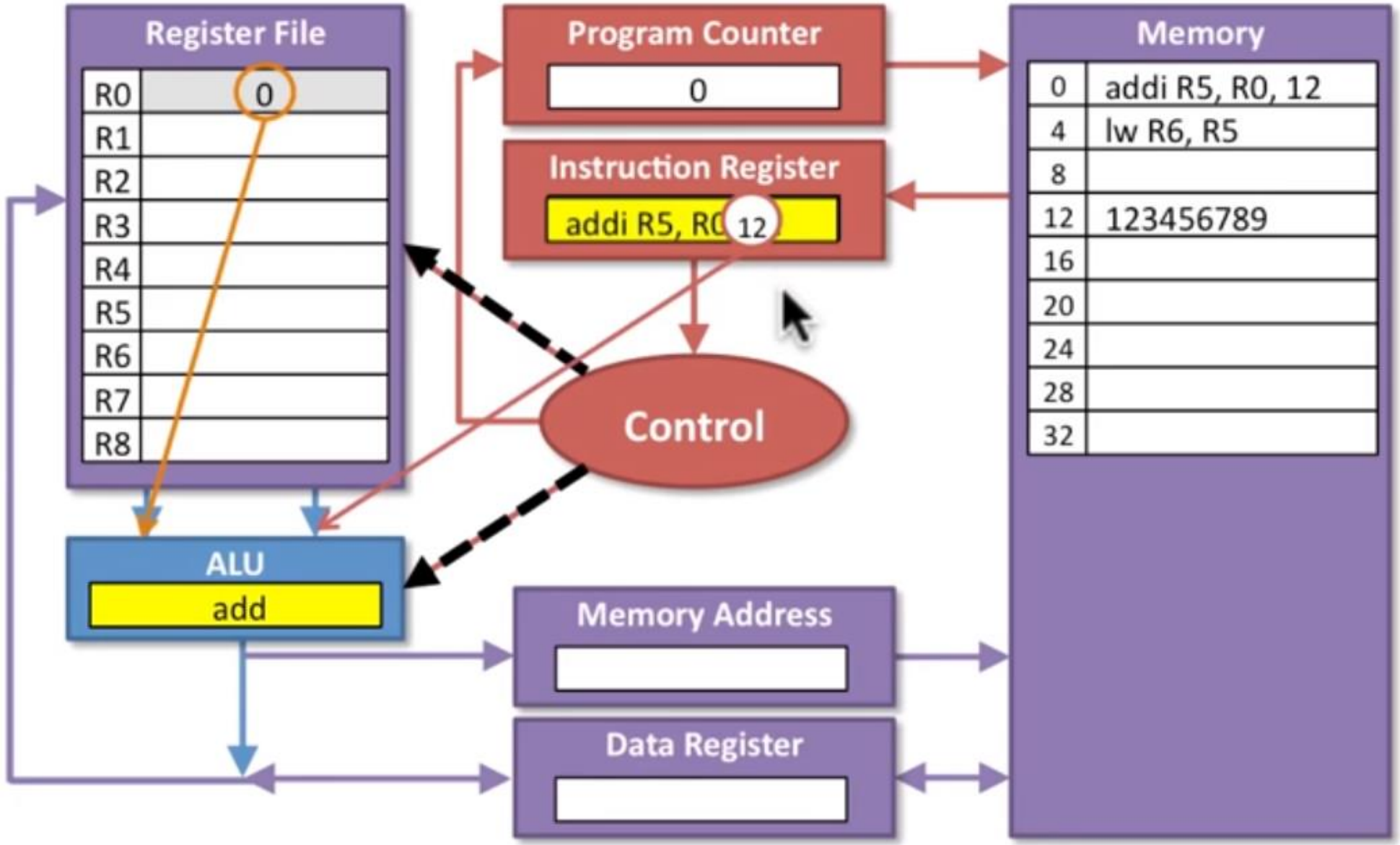
Moving data between memory and the register file

Data transfers in detail

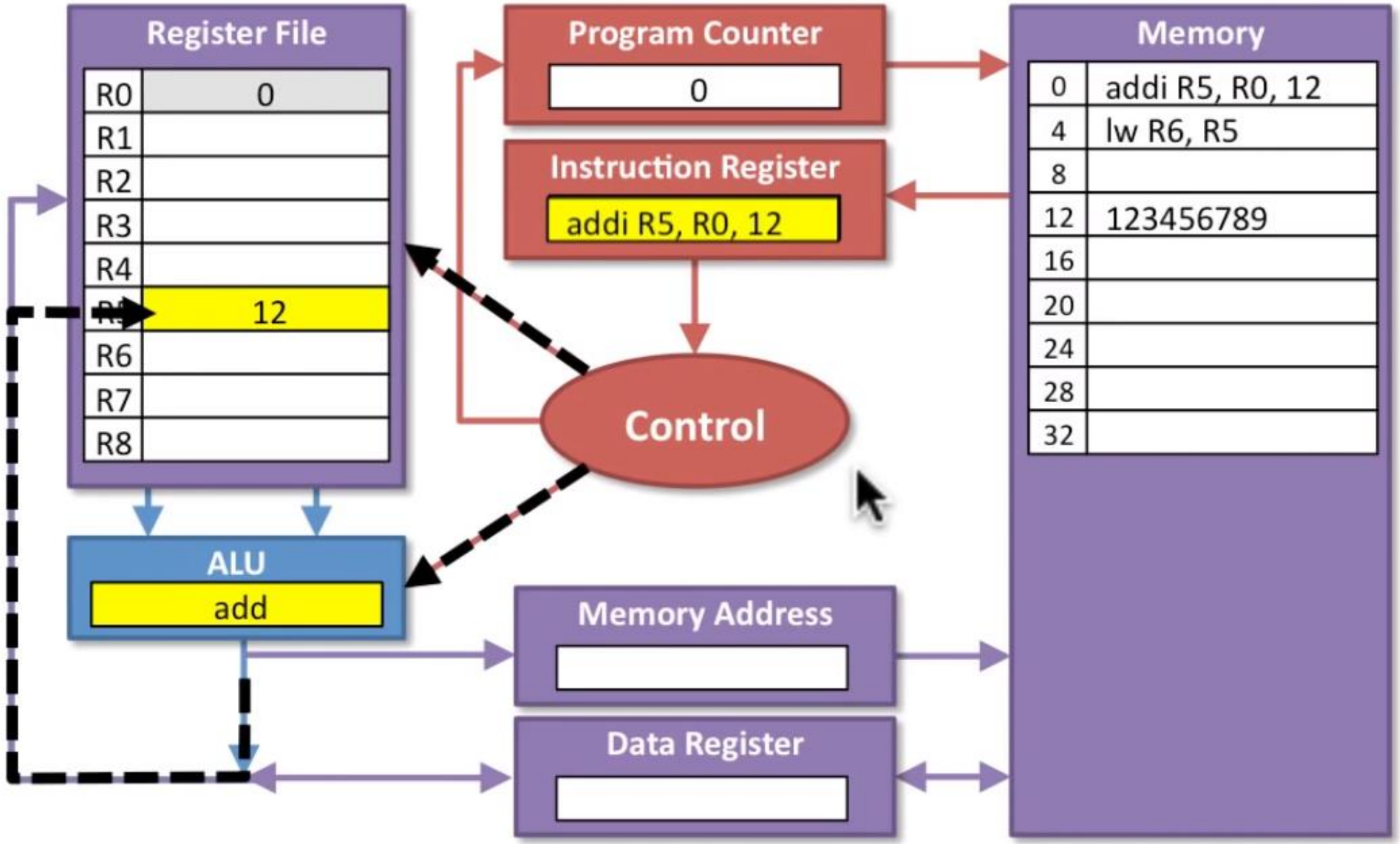
1. ALU *generates* address
2. Address goes to the Memory Address Register
3. Results to/from memory are stored in the Memory Data Register
4. Data from memory can now be stored back into the Register File or to memory can be written.



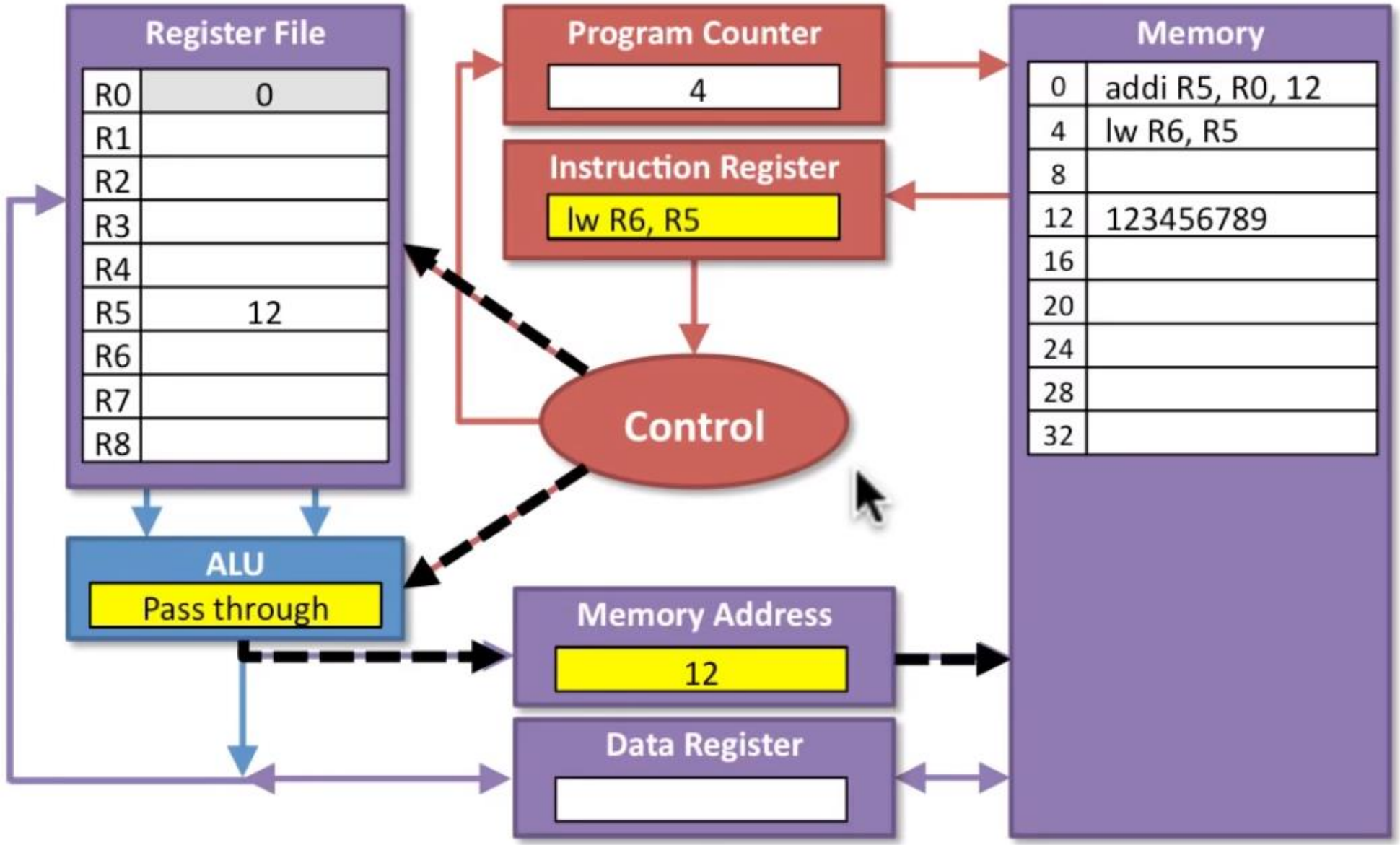
Load word example (1 of 2)



Load word example (1 of 2)

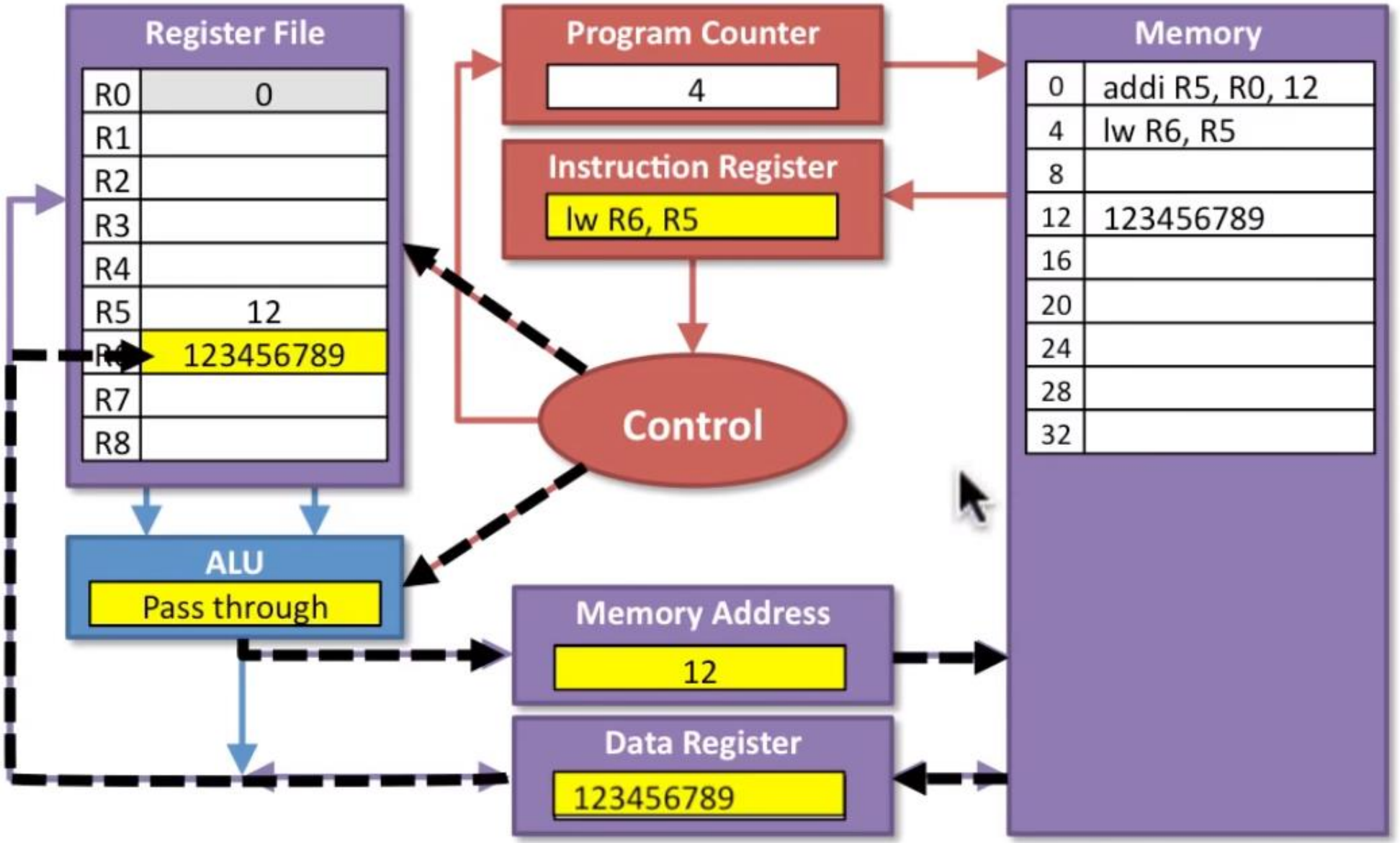


Load word example (2 of 2)



Load word example (2 of 2)

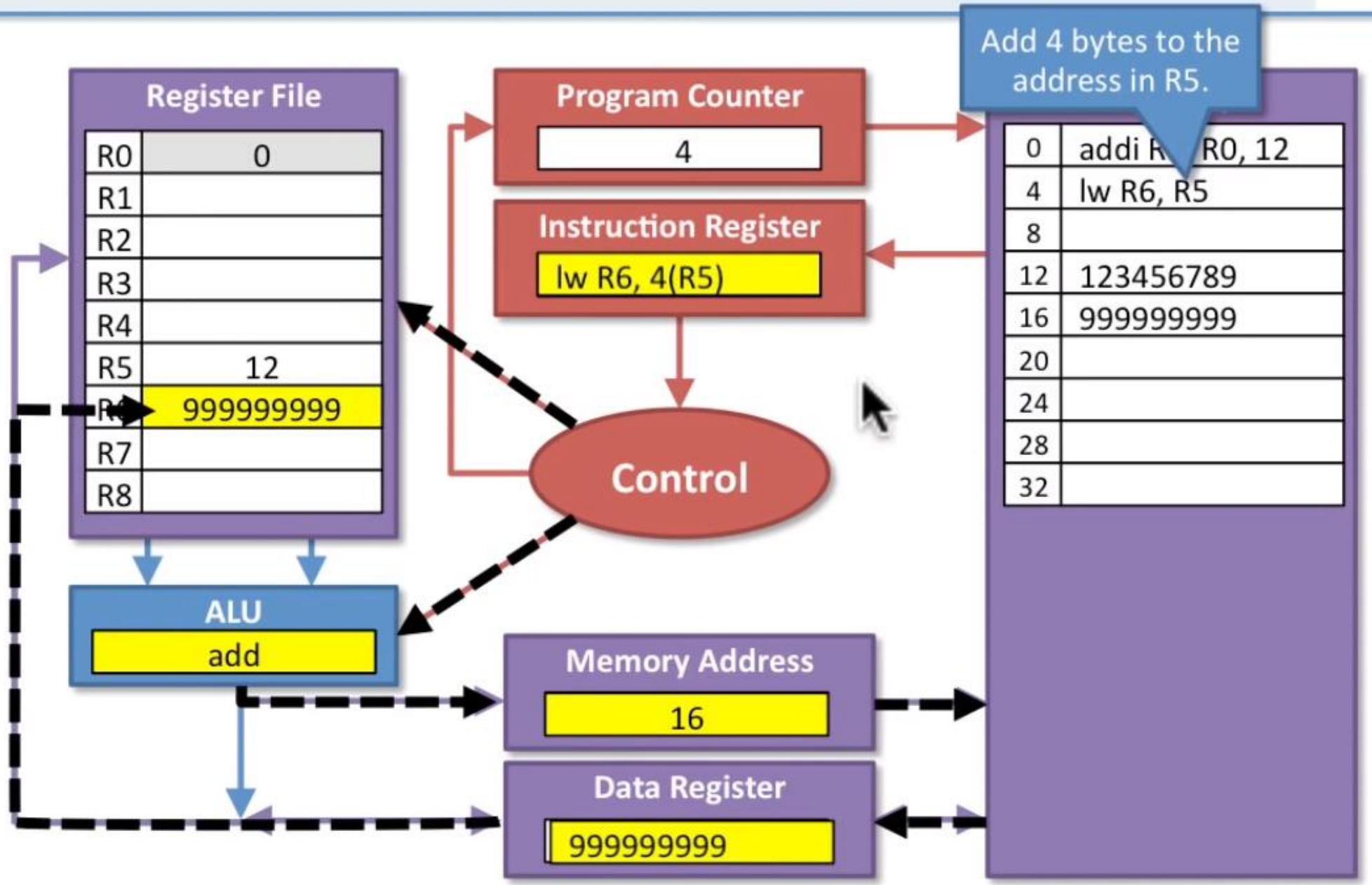
We used the address from the register file to load data from memory.



Load word with offset

An offset can be added to addresses as part of the lw/sw instructions.

lw R6, 4(R5)

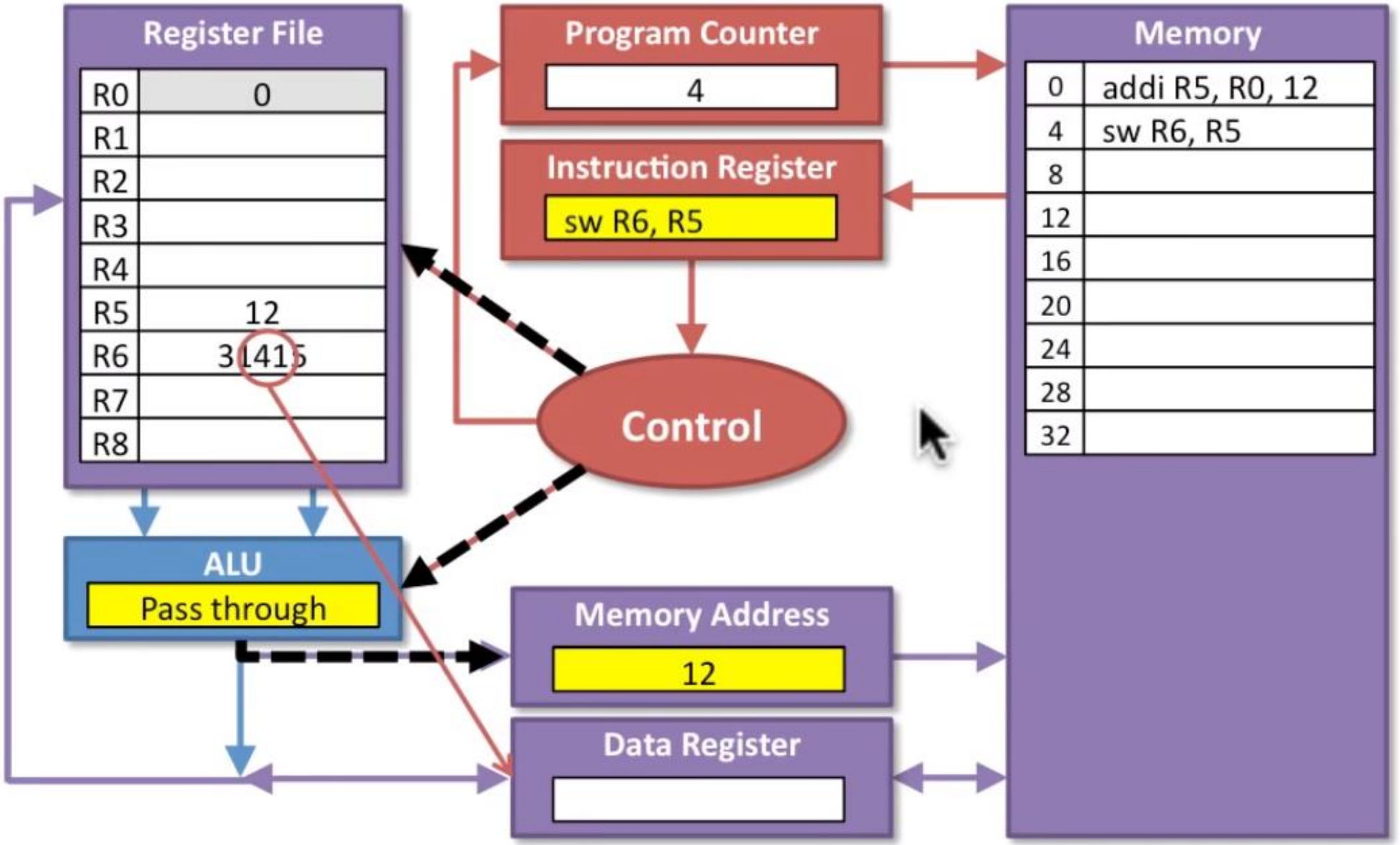


Store word example

For stores we need:

the address (from ALU)

the data (from register)

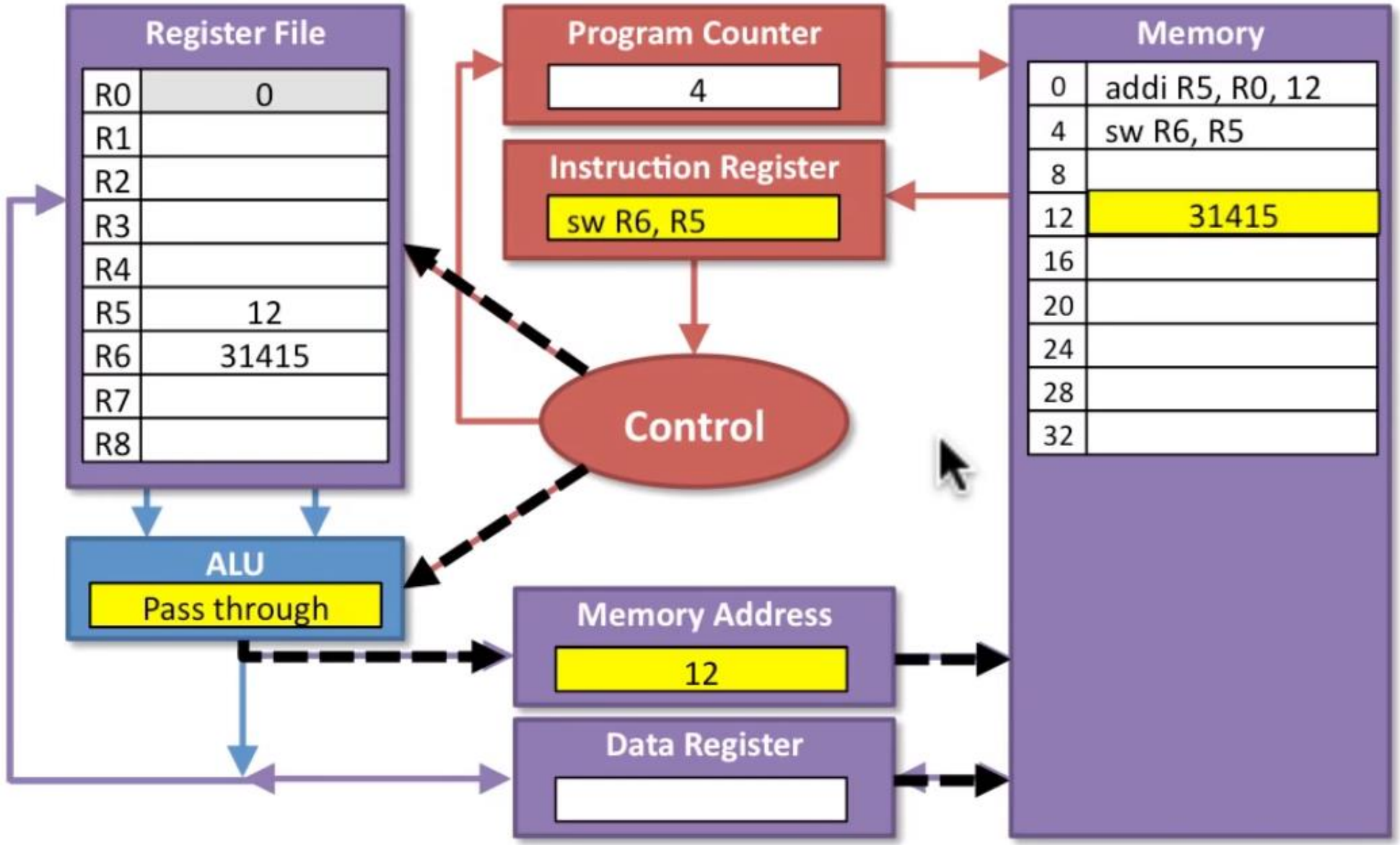


Store word example

For stores we need:

the address (from ALU)

the data (from register)



Sequencing Instructions

Branches, if-then-else, and loops

Sequencing (control instructions)

- **Sequencing instructions make decisions**
  - What instruction to execute **next**?
  - They change the "**control flow**" of the program

- **MIPS conditional branch instructions**
  - `bne R0, R1, Label` branch if *not equal* to label
  - `beq R3, R4, Label` branch if *equal* to label

- **Example:**

```
                    R1 = i; R2 = j; R3 = h

if (i==j)           bne R1, R2, Skip
  h = i+j;            add R3, R1, R2
...                 Skip:
                      ...
```

Sequencing: unconditional jump

- MIPS unconditional branch instruction: jump
  - `j Label` jump to label
- Example:

```
                        R1 = i; R2 = j; R3 = h

if (i==j)               bne R1, R2, DoElse
  h = i+j;               add R3, R1, R2
                         j SkipElse
else                    DoElse:
  h = i-j;               sub R3, R1, R2
                        SkipElse:
...                     ...
```

Branch to here if R1!=R2

Always skip the subtraction if we did the addition.

Branch Instructions

- **Change the flow** of the program → **change the Program Counter**
  - `j` jump — goto label no matter what
  - `bne` branch not equal — goto label only if registers are not equal
- Example: `if (a==b) c=1; else c=2;`

```
R5 = a; R6 = b; R7 = c
```

	Instruction	Comment
`if (a==b)`	`bne R5, R6, Else`	`; if (a!=b) goto Else`
`  c=1;`	`addi R7, R0, 1`	`; c <-- 1+0`
	`j SkipElse`	`; goto SkipElse`
`else c=2;`	`Else: addi R7, R0, 2`	`; c <-- 2+0`
`...`	`SkipElse: ...`	

Always skip setting to 2 if we set it to 1.

Sequencing: Loops


```
for (j=0; j<10; j++) {
  b = b + j;
}
...
```

R5 = j; R6 = b;

Remember that bne/beq only compare **registers** to **registers**. We need to put 10 in a **register** first!

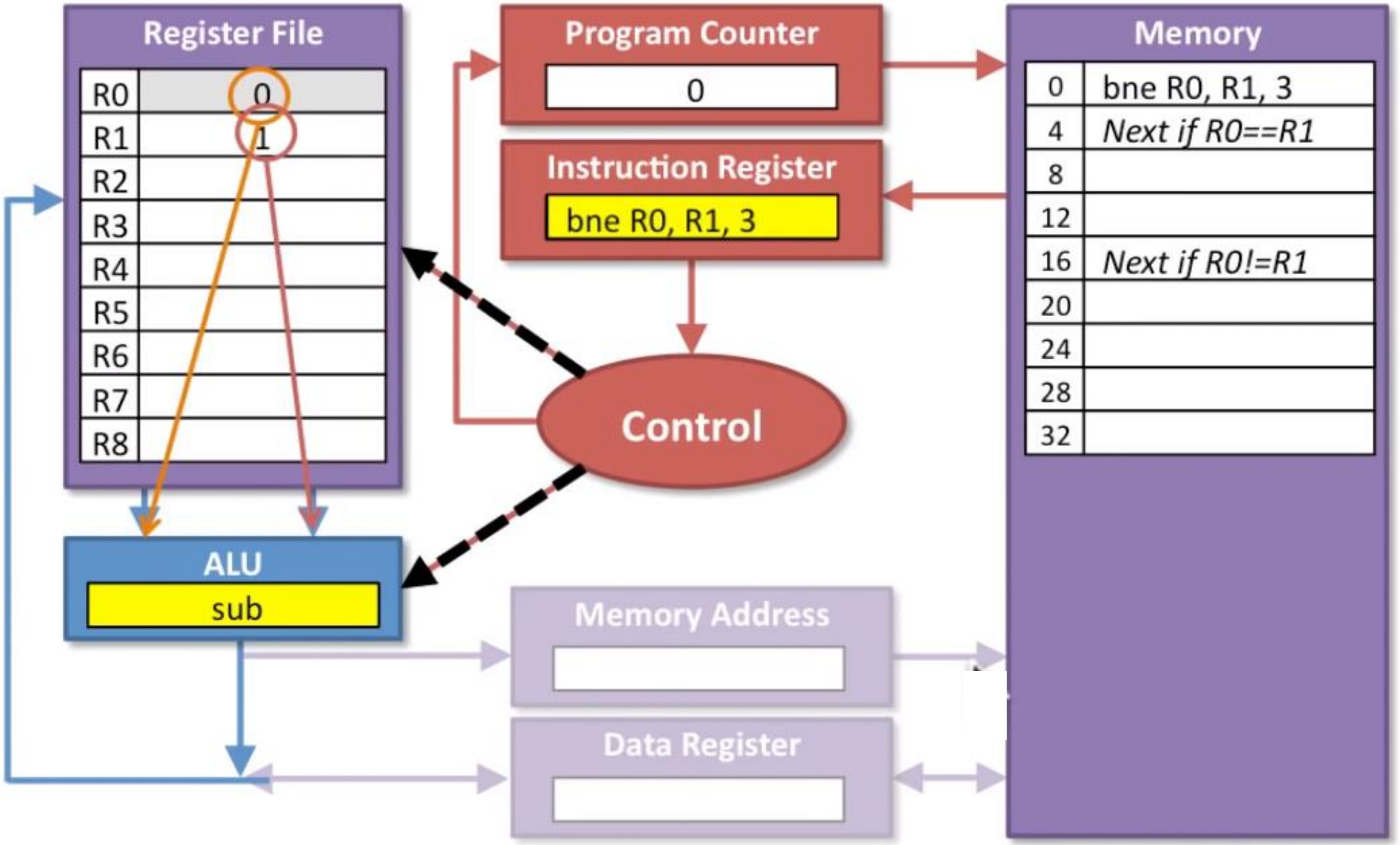
We need the **constant** 10 for the loop comparison, so put it in **register** R1.

```
          Instruction              Comment
          addi R5, R0, 0           ; j ← 0 + 0
          addi R1, R0, 10          ; R1 ← 0 + 10
Loop:     beq  R5, R1, Exit        ; if ( j == 10) goto Exit
          add  R6, R6, R5          ; b ← b + j
          addi R5, R5, 1           ; j ← j + 1
          j    Loop                ; goto Loop
Exit:     ...                      ; pop out of loop, continue
```

Sequencing example

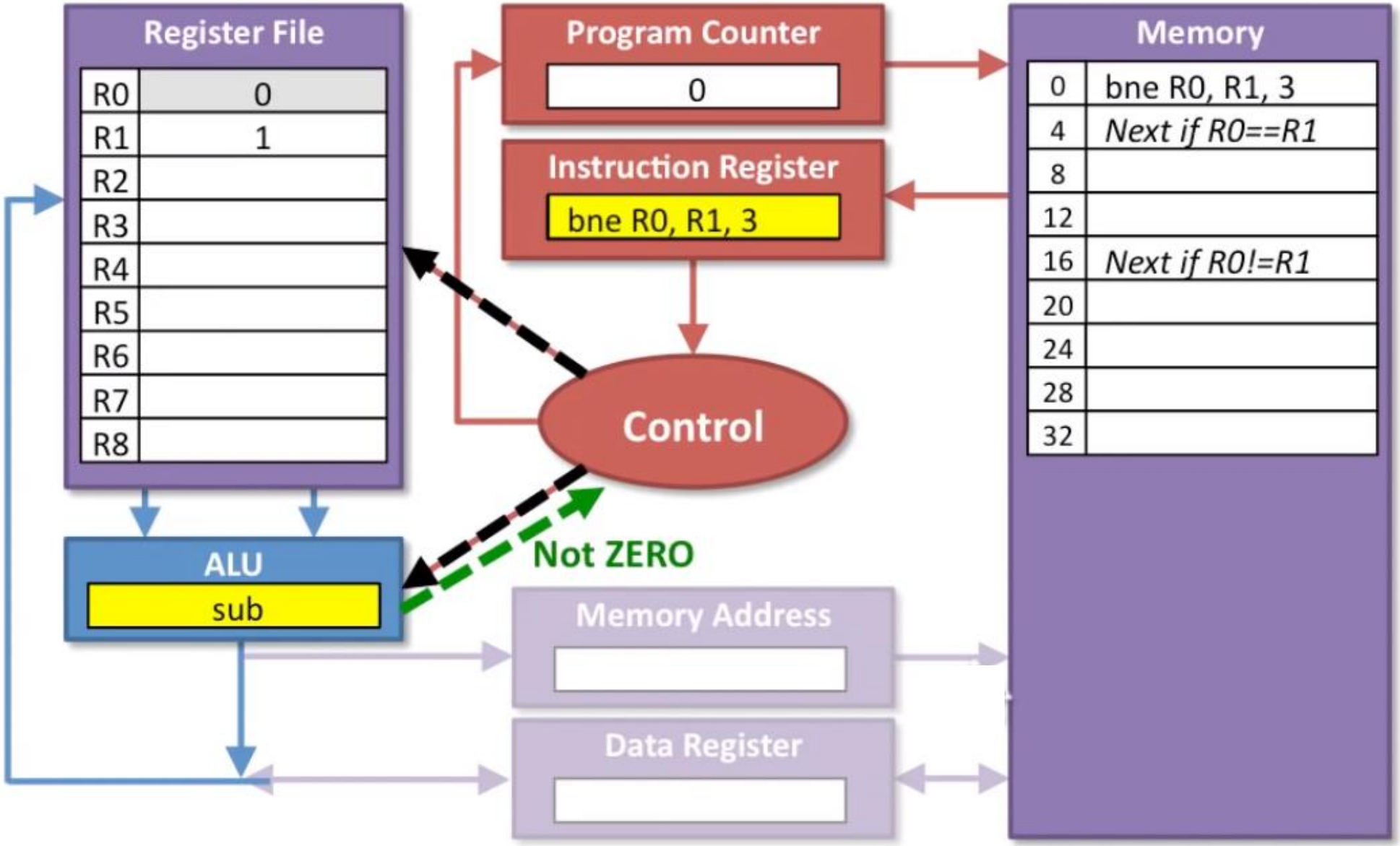
Sequencing in detail

1. **ALU** *compares* registers



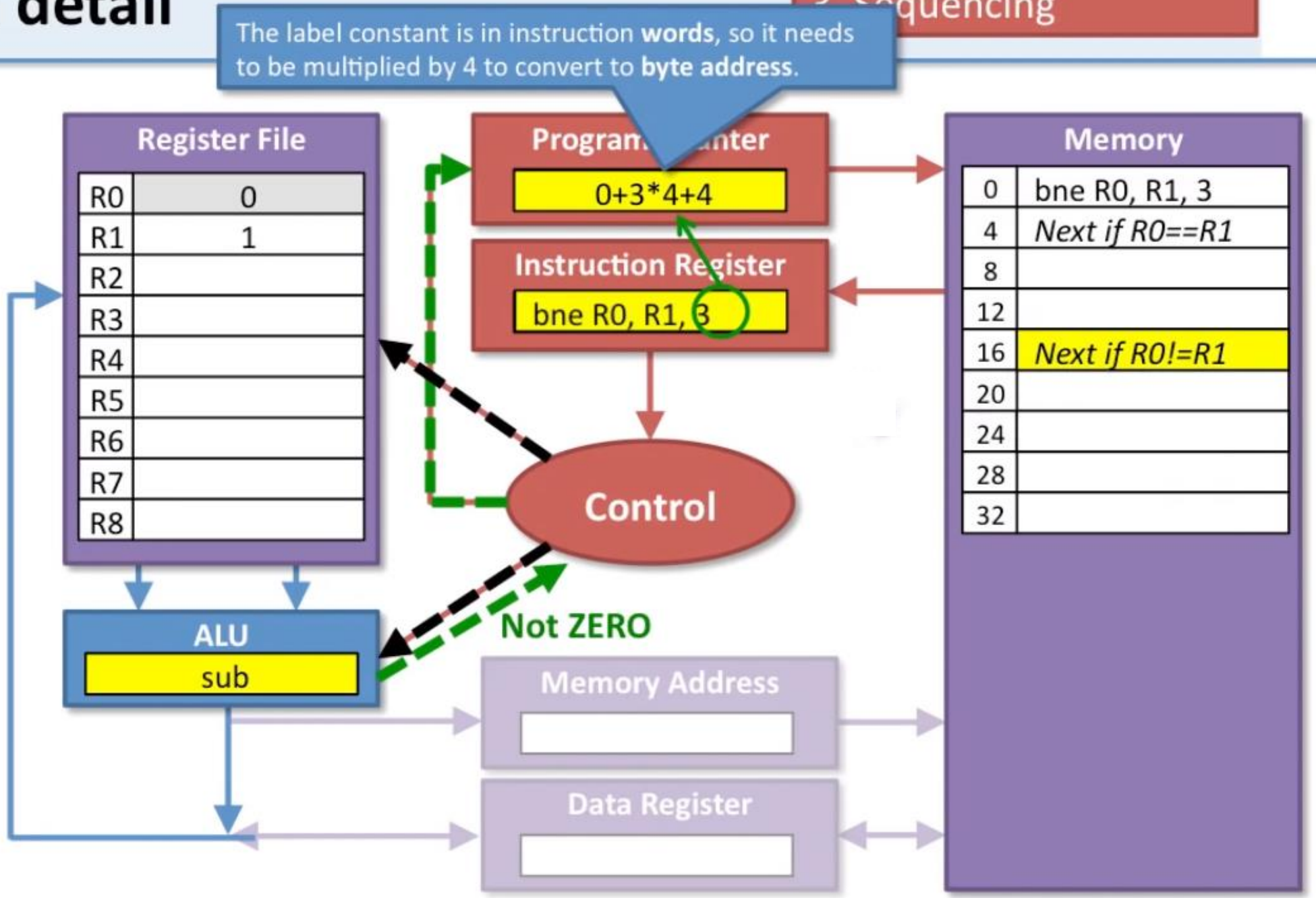
Sequencing in detail

1. **ALU** *compares* registers



Sequencing in detail

1. **ALU** *compares* registers
2. Result tells the **Control** whether to branch
3. If the branch is *taken*, then the **Control** **adds** a constant from the instruction to the **Program Counter**
4. The **Control** always adds 4 to the **Program Counter**



Other ISAs

- We've looked at MIPS in detail, but there are many other ISAs:
  - x86 (Intel/AMD)
  - ARM (ARM/Samsung/Apple)
  - JVM (Java)
  - PTX (Nvidia)
  - PPC (IBM, Motorola)
  - SPARC (Oracle, Fujitsu)
  - etc.

- Let's take a look at a few issues:
  - Machine types
  - ISA classes
  - Addressing modes
  - Instruction width
  - CISC vs. RISC

Basic machine types

- **Memory-to-Memory** machines
  - Instructions can directly manipulate memory
    - Mem[0] = Mem[1] + Mem[2]
  - Problems:
    - Need to store temporary values in memory
    - Memory is slow
    - Memory is big → need lots of bits for addresses

- **Register** machines
  - Hold temporary variables in a Register File
  - Register File is smaller → far faster than memory → faster programs
  - Fewer addresses in code → smaller programs

- **But it's never that simple…**
  - x86 has a few registers *and* supports memory operations (Register-Memory)
  - ARM has many addressing modes that complicate register operations
  - When you run out of registers you have to use memory anyway

Basic ISA classes

- **Accumulator** (1 register)
  - 1 address add A acc ← acc + mem[A]
- **General purpose register file** (load/store)
  - 3 addresses add Ra Rb Rc Ra ← Rb + Rc
    load Ra Rb Ra ← Mem[Rb]

MIPS

- **General purpose register file** (Register-Memory)
  - 2 address add Ra addressB Ra ← Mem[addressB]

x86

- **Stack** (not a register file but an operand stack)
  - 0 address add tos ← tos + next
    tos = top of stack

Java

- **Comparison**:
  - Bytes per instruction? Number of instructions? Speed?

Comparing number of instructions

Code for **C = A + B**

Stack	Accumulator	Register (Register-memory)	Register (Load-store)
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1, A Load R2, B Add R3, R2, R1 Store C, R3
JVM	PDP-8, 8008, 8051	x86	MIPS, PPC, ARM, SPARC

Comparing number of instructions

Code for **C = A + B**

Stack	Accumulator	Register (Register-memory)	Register (Load-store)
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1, A Load R2, B Add R3, R2, R1 Store C, R3
JVM	PDP-8, 8008, 8051	x86	MIPS, PPC, ARM, SPARC

Many very small instructions (no registers)

Lots of simple instructions.

Good unless you need temporaries...

Small code, but hard to build in hardware

Addressing modes (not all are in MIPS)

Addressing mode	Example	Meaning
Register	add R4, R3	R4←R4+R3
Immediate	add R4, 23	R4←R4+23
Displacement	add R4, 100(R1)	R4←R4+Mem[100+R1]
Register indirect	add R4, (R1)	R4←R4+Mem[R1]
Indexed/Base	add R3,(R1+R2)	R3←R3+Mem[R1+R2]
Direct or absolute	add R1,(1001)	R1←R1+Mem[1001]
Memory indirect	add R1,@(R3)	R1←R1+Mem[Mem[R3]]
Auto-increment	add R1,(R2)+	R1←R1+Mem[R2]; R2←R2+d
Auto-decrement	add R1,-(R2)	R2←R2-d; R1←R1+Mem[R2]
Scaled	add R1, 100(R2)[R3]	R1←R1+Mem[100+R2+R3*d]

MIPS

Very helpful for processing arrays of data.

Instruction widths (number of bits)

- **Variable width**
  - Different widths for different instructions
  - x86: **2-6 bytes for add**, **2-4 bytes for load**
  - Better for generating **compact code**
  - **Hard for hardware** to know where instructions start/stop

- **Fixed width**
  - Same width for every instruction
  - MIPS: **4 bytes for add**, **4 bytes for load**
  - **Larger code** size
  - **Easy for hardware** to decode

- **Multiple widths**
  - ARM and MIPS support both 32-bit and 16-bit instructions

General purpose register machines dominate

- **Literally all machines use general purpose registers**

- **Advantages**
  - **Faster** than memory
    (way faster than memory: 200x-600x)
  - Can hold **temporary variables**
    (easier to break up complex operations)
  - Easier for **compilers** to use
    (regular structure and uniform use)
  - Improved **code density**
    (fewer bits to select a register than a memory address)

Question: register-memory vs. load-store

Q: Which kind of processor takes more instructions to compute C=A+B where A, B, and C are all in memory?

- Register-Memory (can specify memory addresses in the compute instructions)
- Load-Store (all instructions have to compute on registers)
- Same
- Can't tell because we don't know how many registers we have

A: Load-Store

Load-store machines require you to first load the data from memory to the register file then compute on it then write it back. Register-memory machines can specify the memory address in the instruction, which reduces the number of instructions needed (but increases the number of bits needed per instruction).

Register
(Register-memory)

```
Load R1, A
Add R1, B
Store C, R1
```

Register
(Load-store)

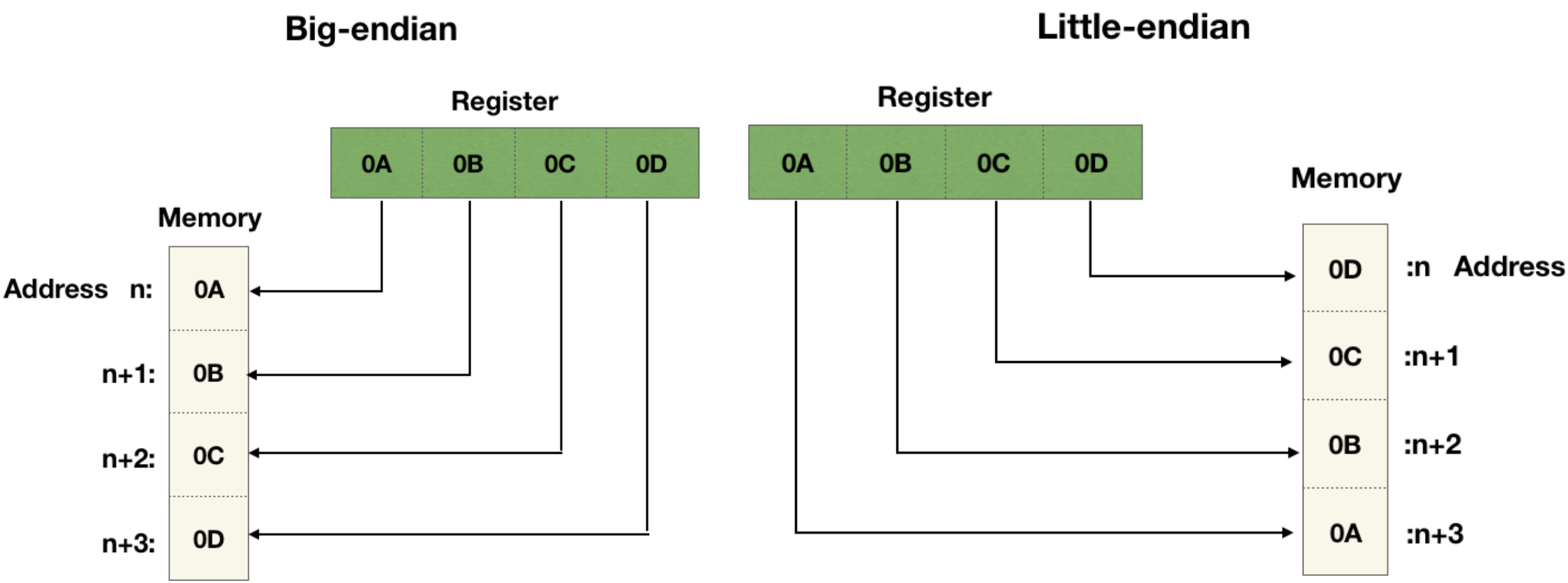
```
Load R1, A
Load R2, B
Add R3, R2, R1
Store C, R3
```

RISC vs CISC

RISC	**CISC**
Reduced Instruction Set Computer	Complex Instruction Set Computer
There are Fixed format instructions.	There are Variable format instructions.
There are few instructions.	There are many instructions.
Heavy use of RAM	More efficient use of RAM
There are multiple register sets are used.	A single register set is used.
Addressing modes-Few	Addressing modes-Many
It can include simple instructions and takes one cycle.	It can include complex instructions and takes multiple cycles.
Eg: It is used in Tablets, smartphones	Eg: Desktops, Laptops

Big Endian and Little Endian Byte Ordering

- The terms endian and endianness refer to the convention that decides the ordering of bytes when processor stores a word data from its register to memory or the other way around, loads a word data from memory to its register.
- The endianness convention consists of two different ways to decides the ordering of bytes when transferring a word data between register and memory.
- The first one is called **Big-endian** and the second one is called **Little-endian.**


Big-endian
Register
0A
0B
0C
0D
Memory
Address n:
0A
n+1:
0B
n+2:
0C
n+3:
0D
Little-endian
Register
0A
0B
0C
0D
Memory
0D
:n Address
0C
:n+1
0B
:n+2
0A
:n+3

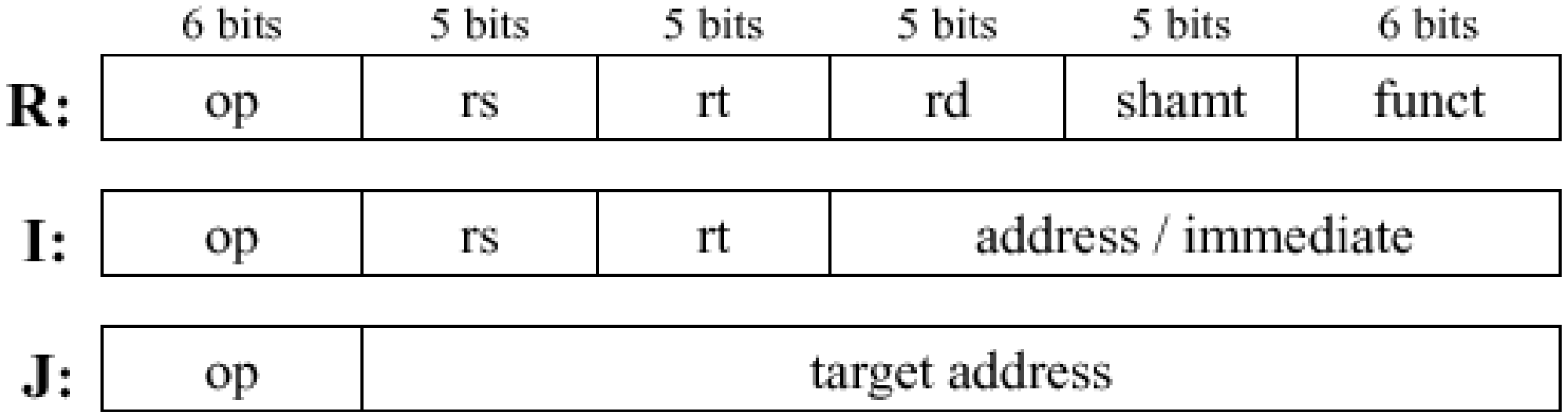
Machine Language Instruction Format

- MIPS instructions have three different formats:

1.R format - Arithmetic instructions

2.I format - Branch, transfer, and immediate instructions

3.J format - Jump instructions



op: basic operation of the instruction (opcode)
rs: first source operand register
rt: second source operand register
rd: destination operand register
shamt: shift amount
funct: selects the specific variant of the opcode (function code)
address: offset for load/store instructions (+/-2^{15})
immediate: constants for immediate instructions

Example 1. Consider an example of an **R-format** instruction **add $t0 $s1 $s2**, which means that we add two numbers stored in C registers 1 and 2 ($s1 and $s2), then put the result in temporary register 0 ($t0).

Here, **$t0 has the address 8**, and the **C-registers have the addresses 17 and 18**. The "$" sign means *contents-of*. For example $18 means *contents of register whose address is 18*. Also, note that the order of the operands in the machine language instruction (ones and zeros) is different than in the assembly-language version.

add $t0, $s1, $s2 (add $8, $17, $18)

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

sll $s2, $t3, 2 . 000000 01011 00000 10010 00010 xxxxxx

Name	Register Number	Usage
$zero	0	the constant value 0
$at	1	reserved for the assembler
$v0-$v1	2-3	value for results and expressions
$a0-$a3	4-7	arguments (procedures/functions)
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	reserved for the operating system
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Consider an example of an **I-format** instruction that involves loading a word - **lw $t0 52($s3)** , which means that we load a word into register $t0, which has address 8, by fetching the data from memory location $s3, which has address 19, with an offset of 52.

lw $t0, 52($s3) lw $8, 52($19)

6 bits	5 bits	5 bits	16 bits
35	19	8	52
100011	10011	01000	0000 0000 0011 0100

Jump Instruction

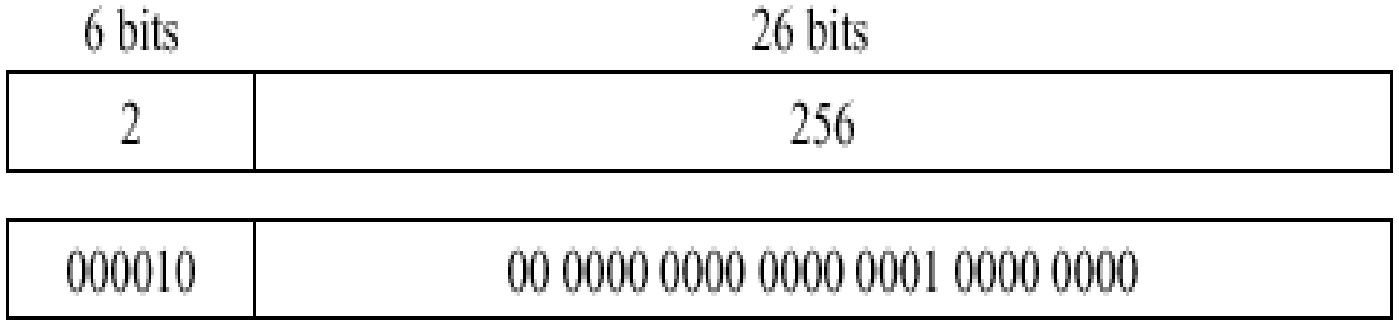
The MIPS *jump* instruction functions like the *go to* instruction in C, that is, control is transferred to a specific destination address in memory. The instruction format is either of the following two forms:

j **label** j **destination_address**

Here, *label* is a string and *destination_address* is a numberical constant within the range of permissible MIPS address values.

Example 3. Suppose we have the following assembly language code:

j 256



Concept of Machine Level Programming, Assembly Level Programming and High level Programming

- Each programming language contains a unique set of keywords and syntax, which are used to create a set of instructions. Thousands of programming languages have been developed till now, but each language has its specific purpose.
- These languages vary in the level of abstraction they provide from the hardware. Some programming languages provide less or no abstraction while some provide higher abstraction. Based on the levels of abstraction, they can be classified into two categories:
- Low-level language
- High-level language

Low-level language

- The low-level language is a programming language that provides no abstraction from the hardware, and it is represented in 0 or 1 forms, which are the machine instructions. The languages that come under this category are the Machine level language and Assembly language.

Machine-level language

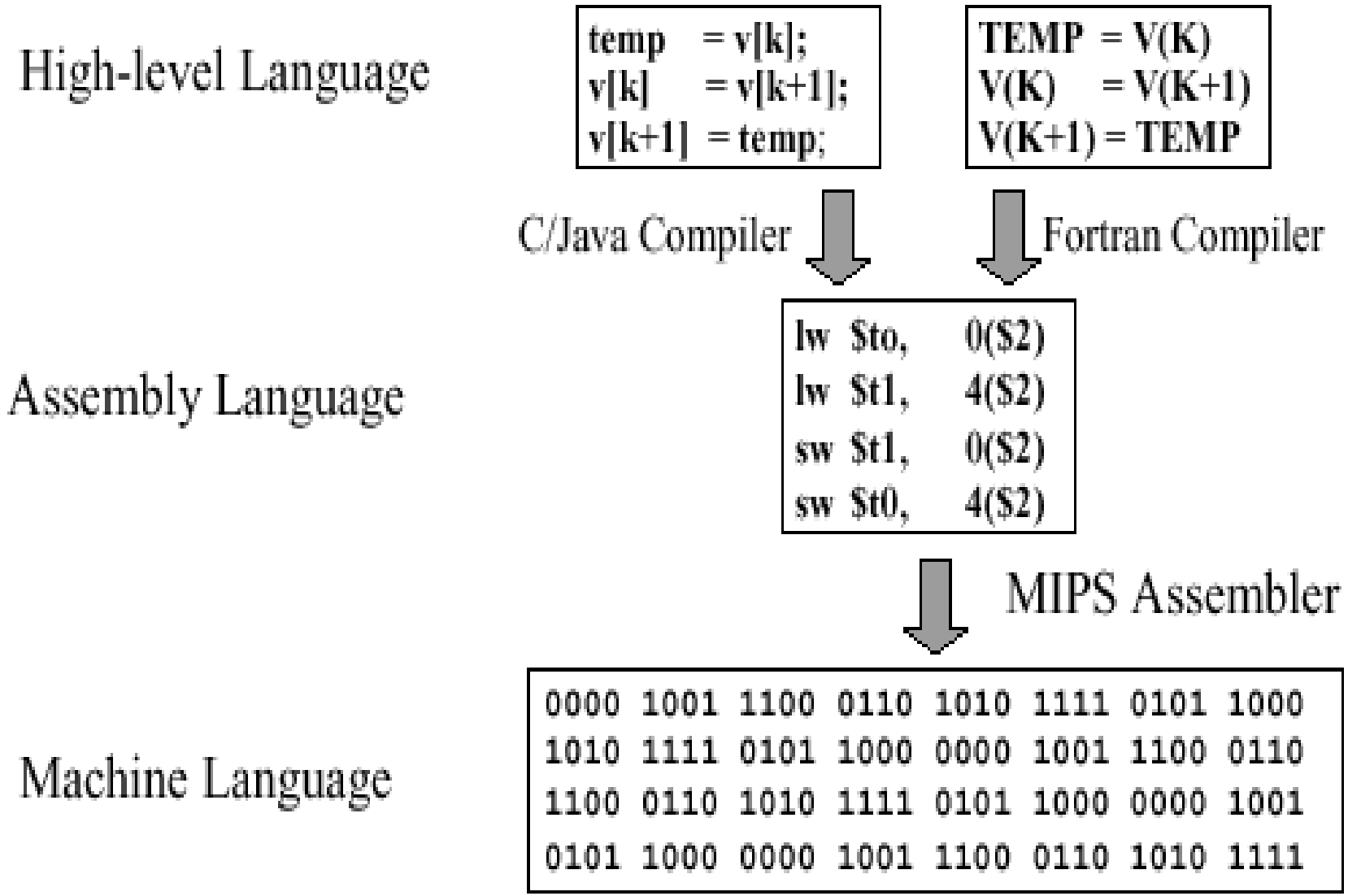
- The machine-level language is a language that consists of a set of instructions that are in the binary form 0 or 1. As we know that computers can understand only machine instructions, which are in binary digits, i.e., 0 and 1, so the instructions given to the computer can be only in binary codes.
- Creating a program in a machine-level language is a very difficult task as it is not easy for the programmers to write the program in machine instructions. It is error-prone as it is not easy to understand, and its maintenance is also very high. A machine-level language is not portable as each computer has its machine instructions, so if we write a program in one computer will no longer be valid in another computer.
- The different processor architectures use different machine codes, for example, a PowerPC processor contains RISC architecture, which requires different code than intel x86 processor, which has a CISC architecture.

Assembly Language

- The assembly language contains some human-readable commands such as mov, add, sub, etc. The problems which we were facing in machine-level language are reduced to some extent by using an extended form of machine-level language known as assembly language. Since assembly language instructions are written in English words like mov, add, sub, so it is easier to write and understand.
- As we know that computers can only understand the machine-level instructions, so we require a translator that converts the assembly code into machine code. The translator used for translating the code is known as an assembler.
- The assembly language code is not portable because the data is stored in computer registers, and the computer has to know the different sets of registers.

High-Level Language

The high-level language is a programming language that allows a programmer to write the programs which are independent of a particular type of computer. The high-level languages are considered as high-level because they are closer to human languages than machine-level languages.


High-level Language
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
TEMP = V(K)
V(K) = V(K+1)
V(K+1) = TEMP
C/Java Compiler
Fortran Compiler
Assembly Language
lw $to, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
MIPS Assembler
Machine Language
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111