ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

COMPUTER ORGANISATION
(TỔ CHỨC MÁY TÍNH)

MIPS: Introduction

Acknowledgement

- The contents of these slides have origin from School of Computing, National University of Singapore.
- We greatly appreciate support from Mr. Aaron Tan Tuck Choy for kindly sharing these materials.

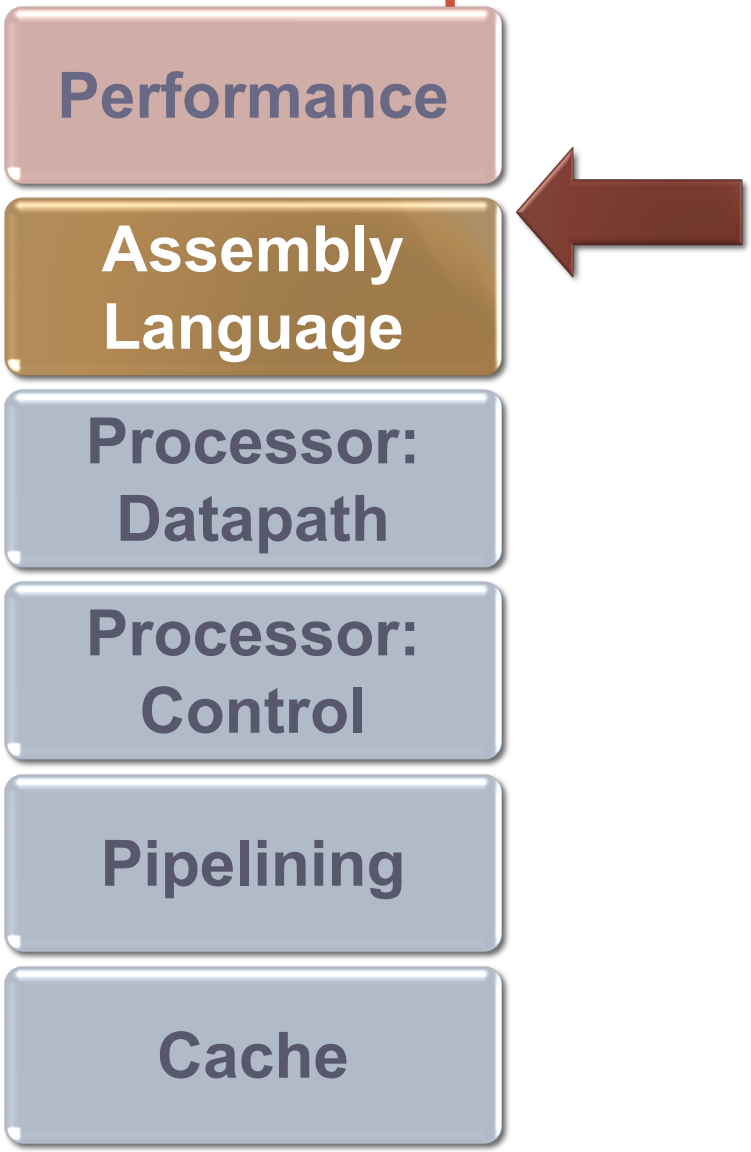
Policies for students

- These contents are only used for students PERSONALLY.
- Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

Recording of modifications

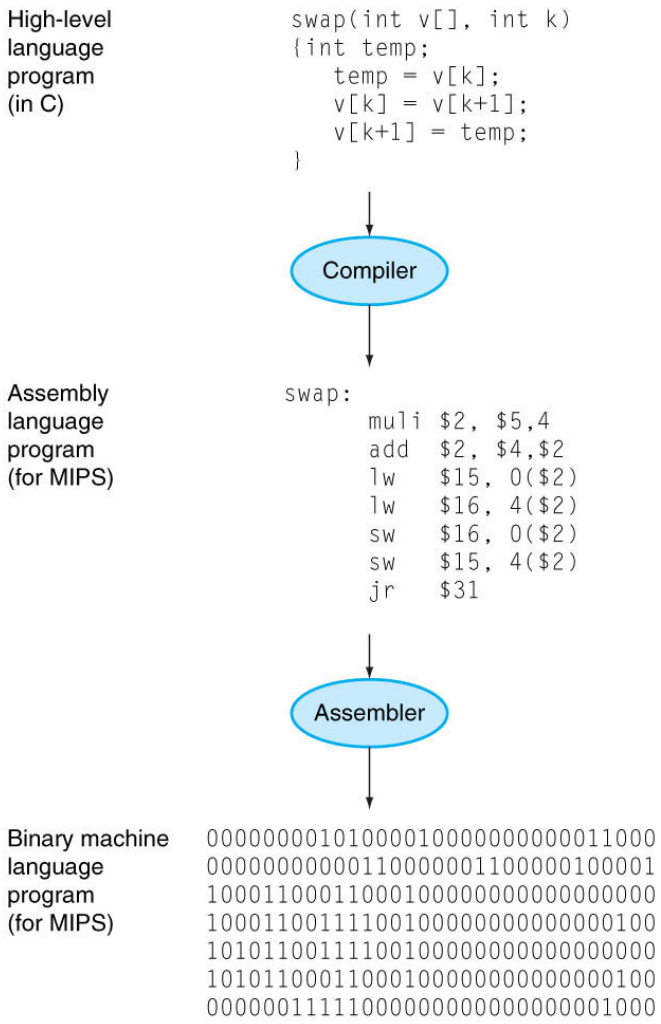
- The list of the text books was changed (1-st book)
- Minor changes on online materials

Road Map: **Part II**



- Introduction:
  - Execution Walkthrough
  - Simple MIPS instructions
    - Arithmetic Operations
    - Immediate Operands
    - Logical Operations

Recap: Controlling the hardware?



- You write programs in high level programming languages, e.g., C/C++, Java:

```
A + B
```

- Compiler translates this into assembly language statement:

```
add A, B
```

- Assembler translates this statement into machine language instructions that the processor can execute:

```
1000 1100 1010 0000
```

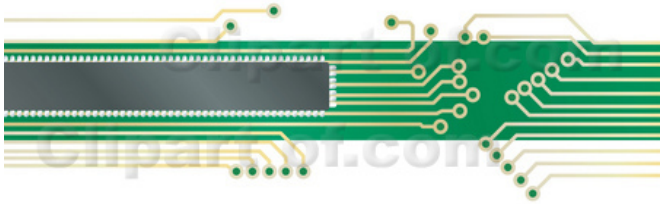
Instruction Set Architecture (1/2)

- **I**nstruction **S**et **A**rchitecture (**ISA**):
  - An abstraction on the interface between the hardware and the low-level software.

Software
(to be translated to the instruction set)

Instruction Set Architecture

Hardware
(implementing the instruction set)



Instruction Set Architecture (2/2)

- Instruction Set Architecture
  - Includes everything programmers need to know to make the machine code work correctly
  - Allows computer designers to talk about functions independently from the hardware that performs them
- This abstraction allows many implementations of varying cost and performance to run identical software.
  - Example: Intel x86/IA-32 ISA has been implemented by a range of processors starting from 80386 [1985] to Pentium 4 [2005]
  - Other companies such as AMD and Transmeta have implemented IA-32 ISA as well
  - A program compiled for IA-32 ISA can execute on any of these implementations

Machine Code vs Assembly Language

- **Machine code**
  - Instructions are represented in binary
  - `1000110010100000` is an instruction that tells one computer to add two numbers
  - Hard and tedious for programmer

- **Assembly language**
  - Symbolic version of machine code
  - Human readable
  - `add A, B` is equivalent to `1000110010100000`
  - **Assembler** translates from assembly language to machine code
  - Assembly can provide '**pseudo-instructions**' as syntactic sugar
  - When considering performance, only real instructions are counted.

Walkthrough: The code example

- Let us take a journey with the execution of a simple code:
  - Discover the components in a typical computer
  - Learn the type of instructions required to control the processor
  - Simplified to highlight the important concepts ☺

```
//assume "res is 0" initially
for ( i = 1; i < 10; i++ )
{
    res = res + i;
}
```

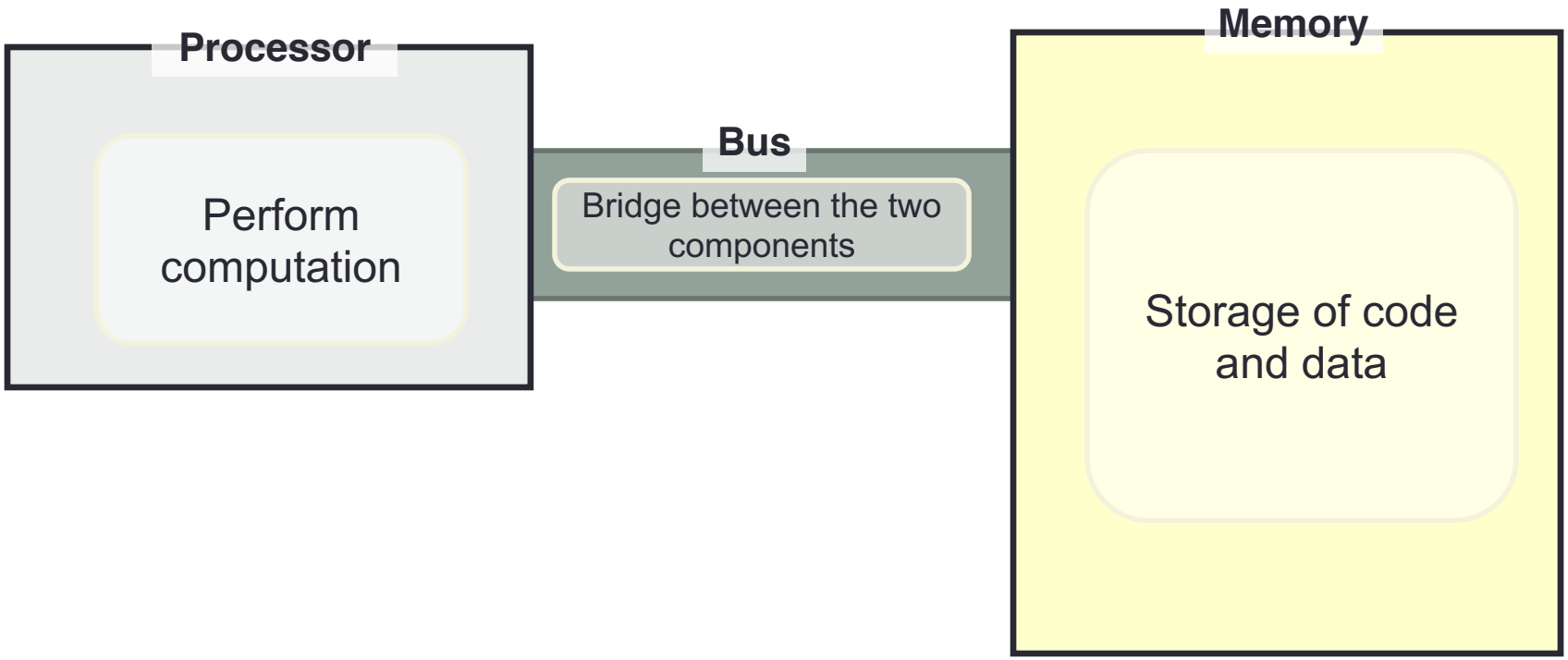
C-like code fragment


```
res ← res + i
i ← i + 1
If i < 10, repeat
```

"Assembly" Code

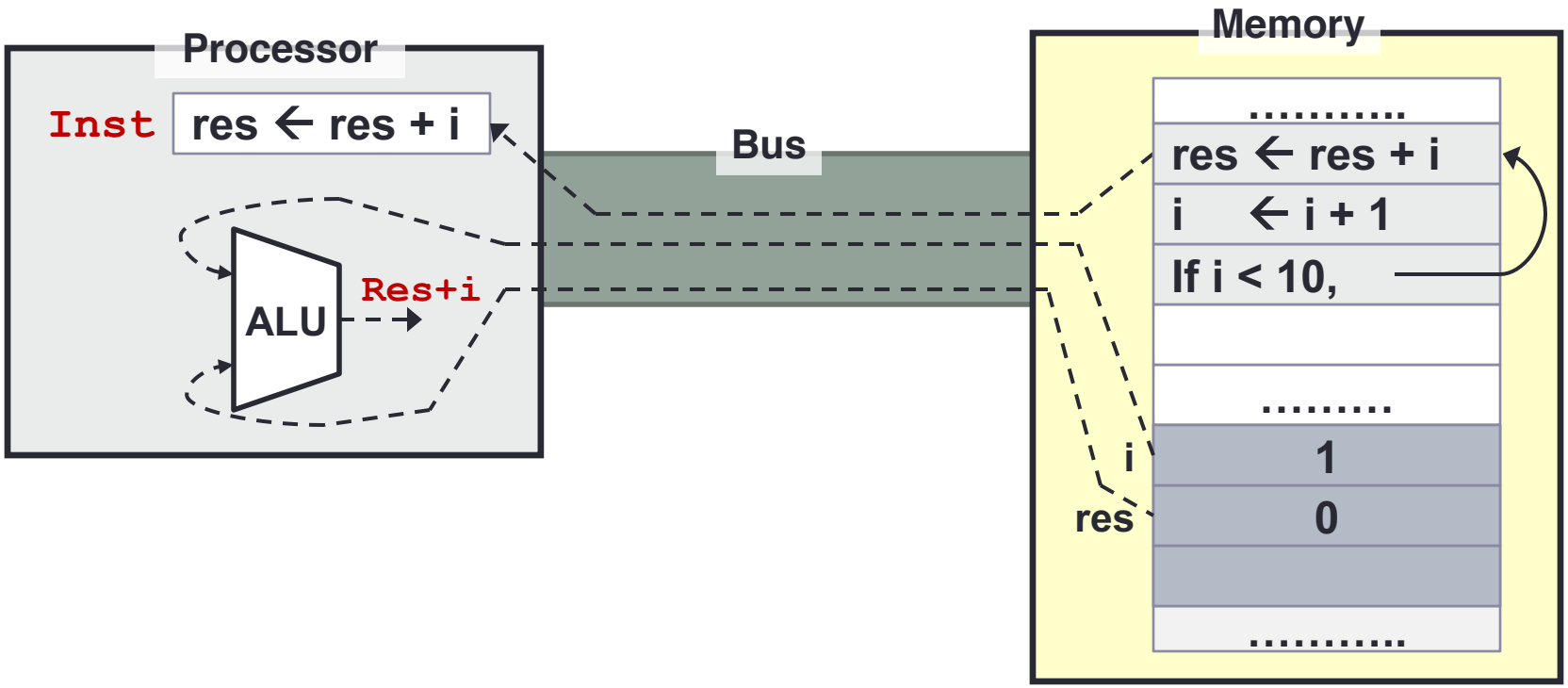
Walkthrough: The components

- The two major components in a computer
  - **Processor** and **Memory**
  - Input/Output devices omitted in this example



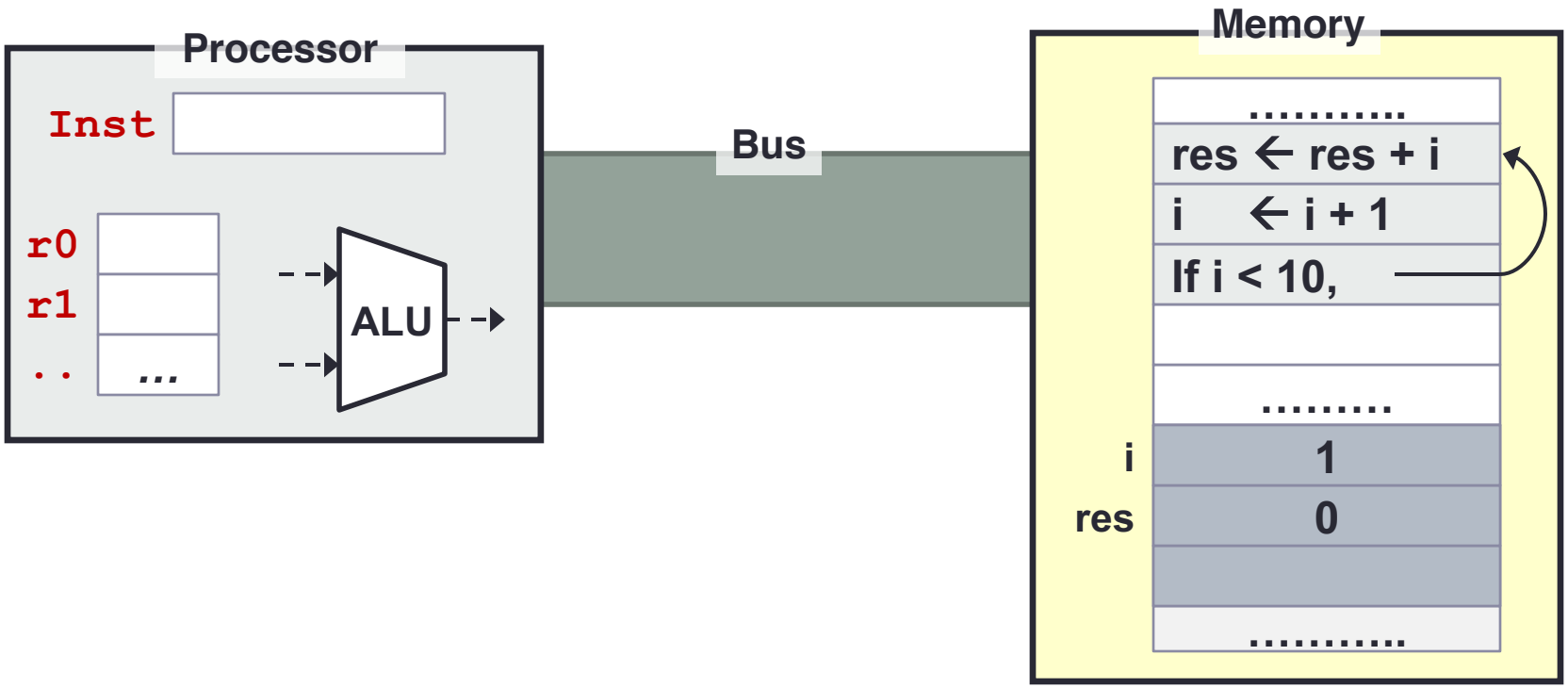
Walkthrough: The code in action

- The code and data reside in memory
  - Transferred into the processor during execution



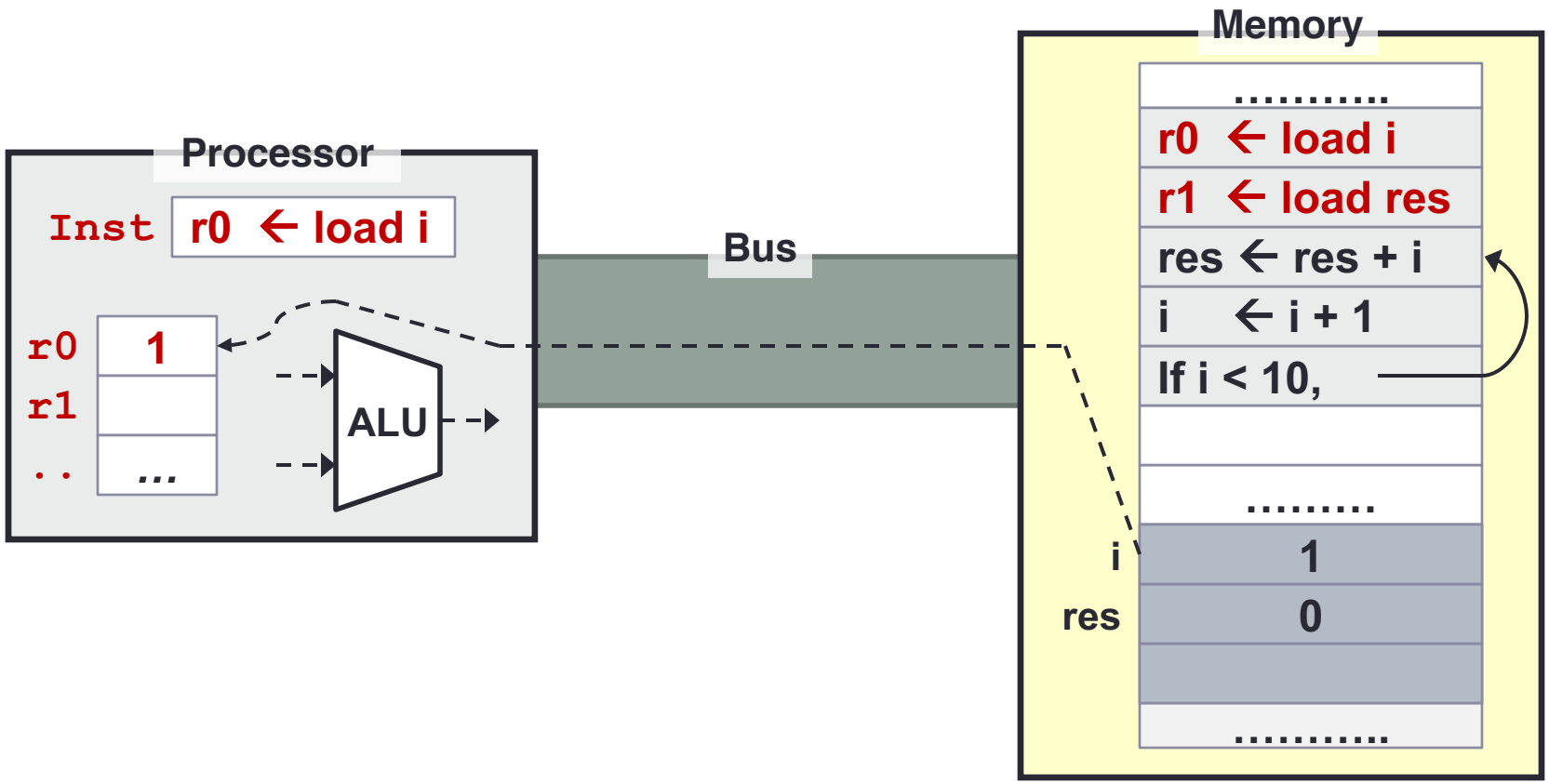
Walkthrough: Memory access is slow!

- To avoid frequent access of memory
  - Provide temporary storage for values in the processor (known as **register**)



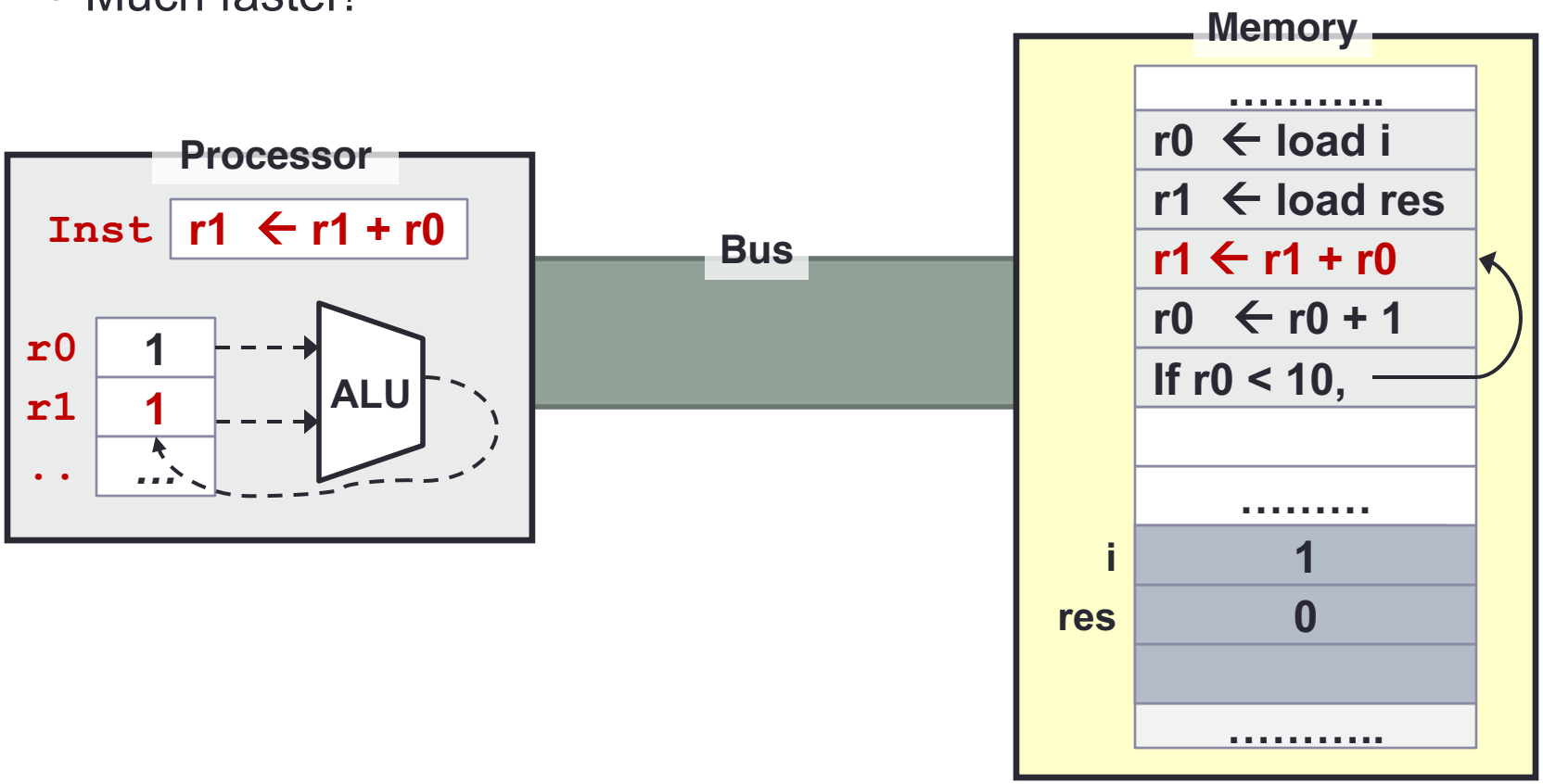
Walkthrough: Memory instruction

- Need instruction to move data into register
  - also out of register into memory later



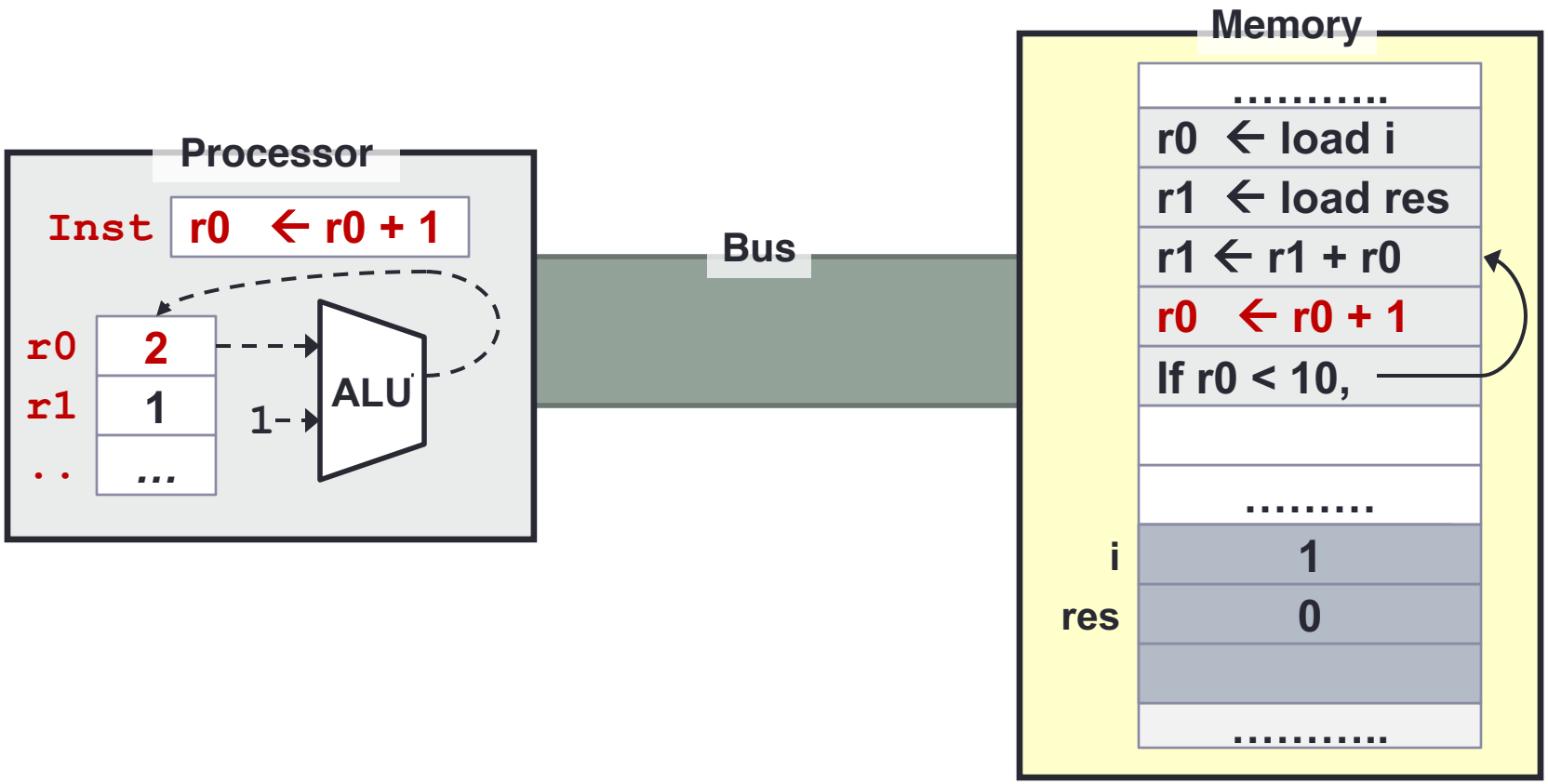
Walkthrough: Reg-to-Reg Arithmetic

- Arithmetic operation can now work directly on registers only:
  - Much faster!



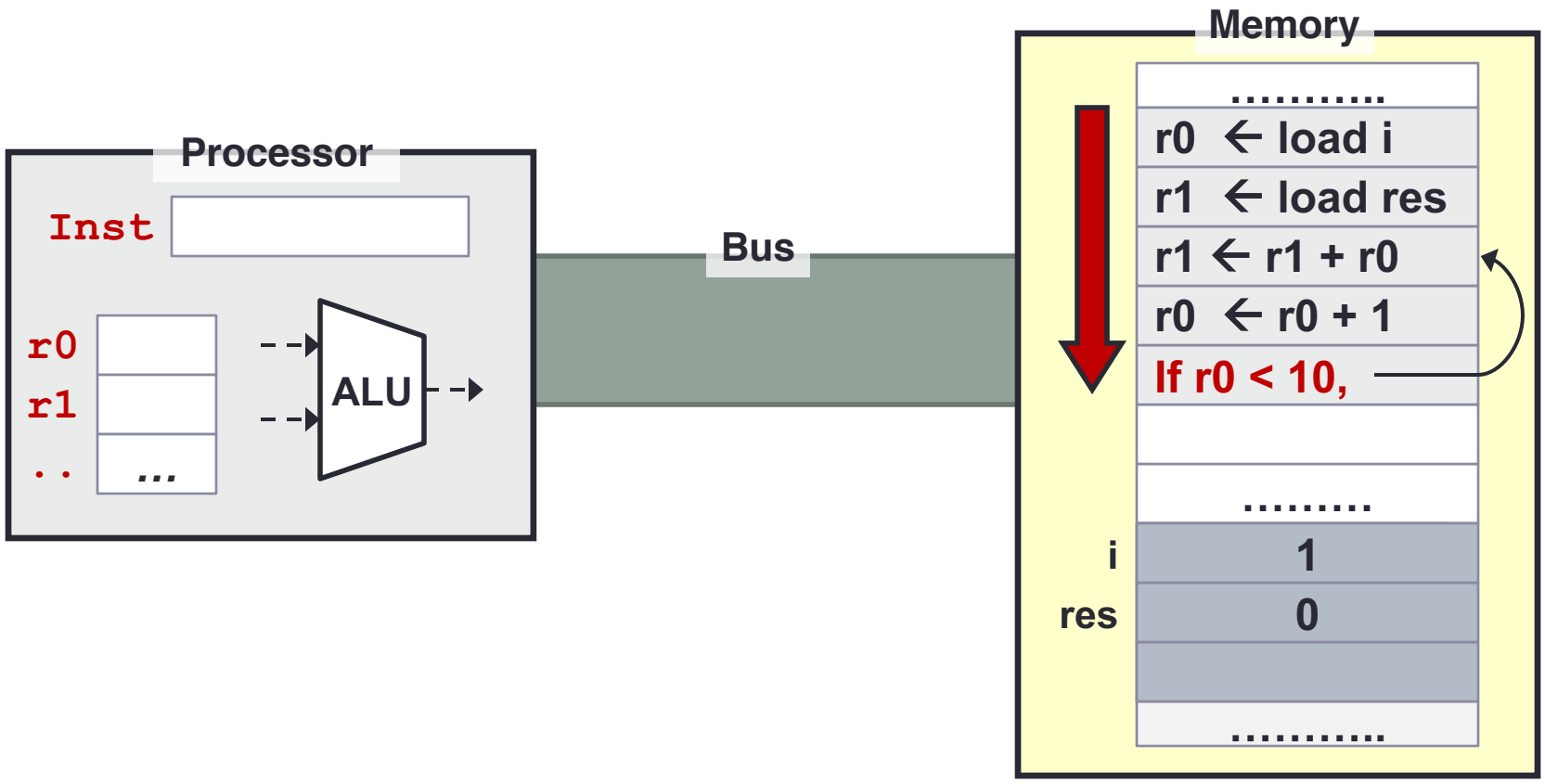
Walkthrough: Reg-to-Reg Arithmetic

- Sometimes, arithmetic operation uses a constant value instead of register value



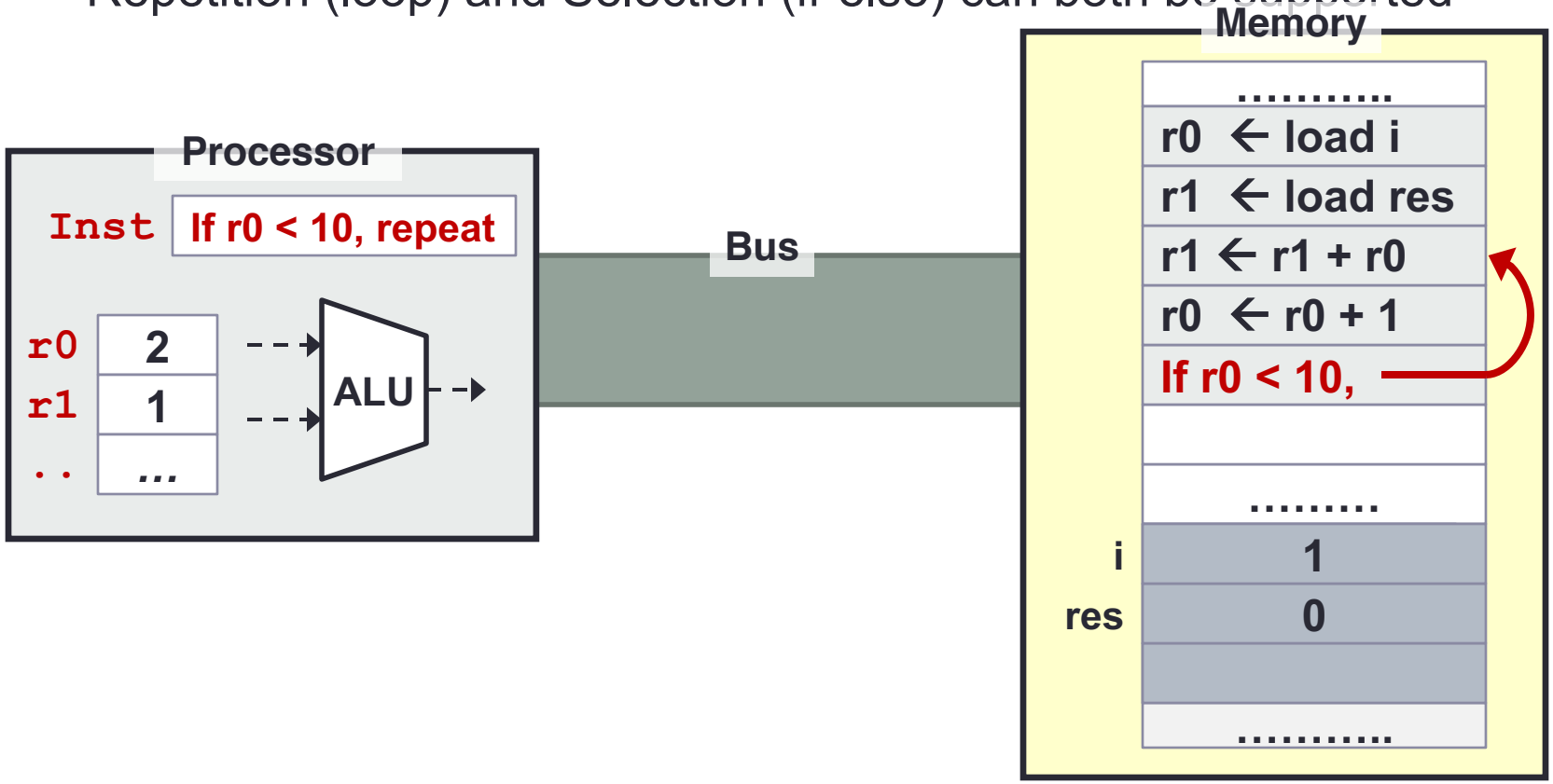
Walkthrough: Execution sequence

- Instruction is executed sequentially by default
  - How do we "repeat" or "make a choice"?



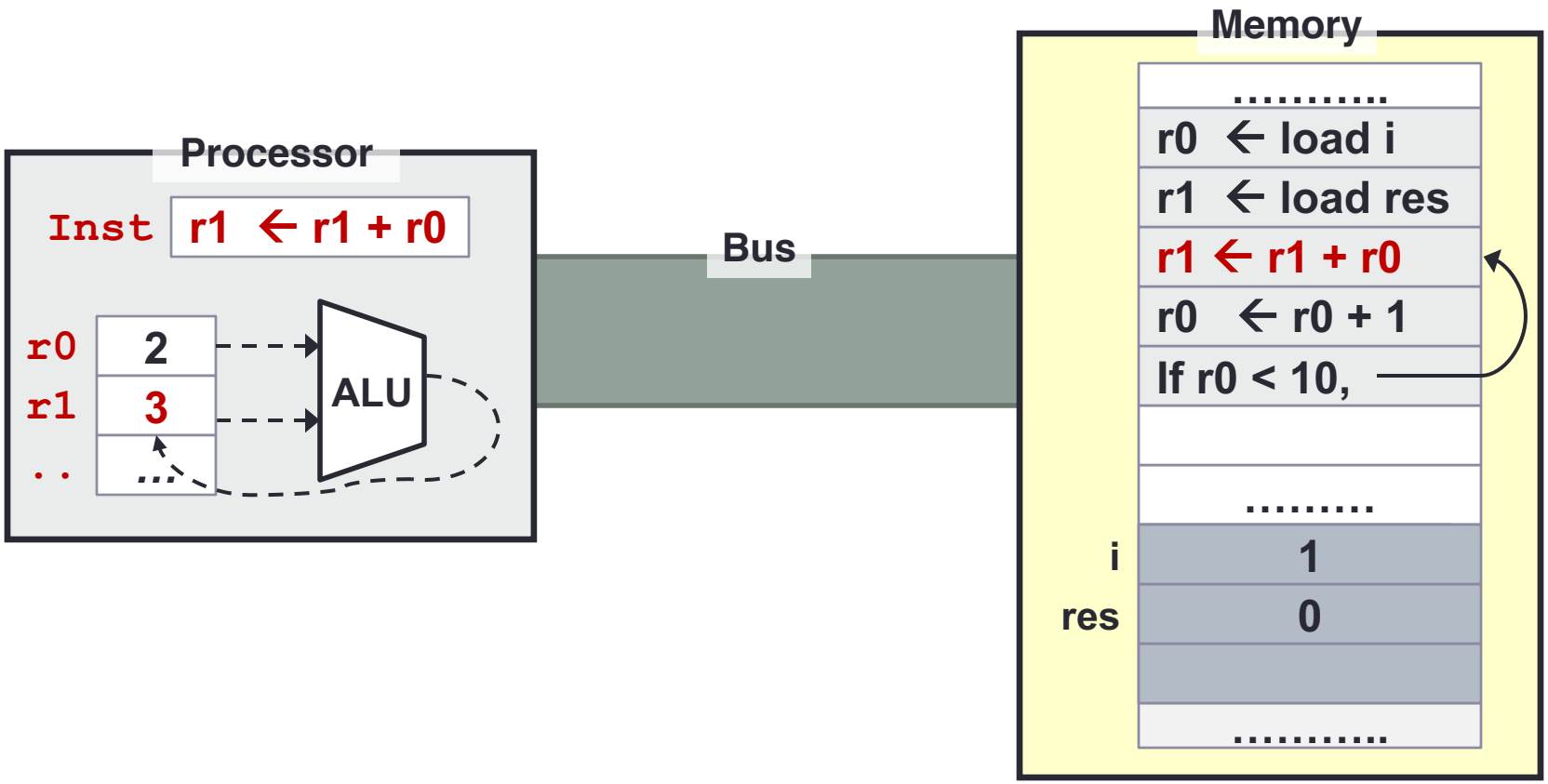
Walkthrough: Control flow instruction

- We need instruction to **change** the control flow based on **condition**:
  - Repetition (loop) and Selection (if-else) can both be supported



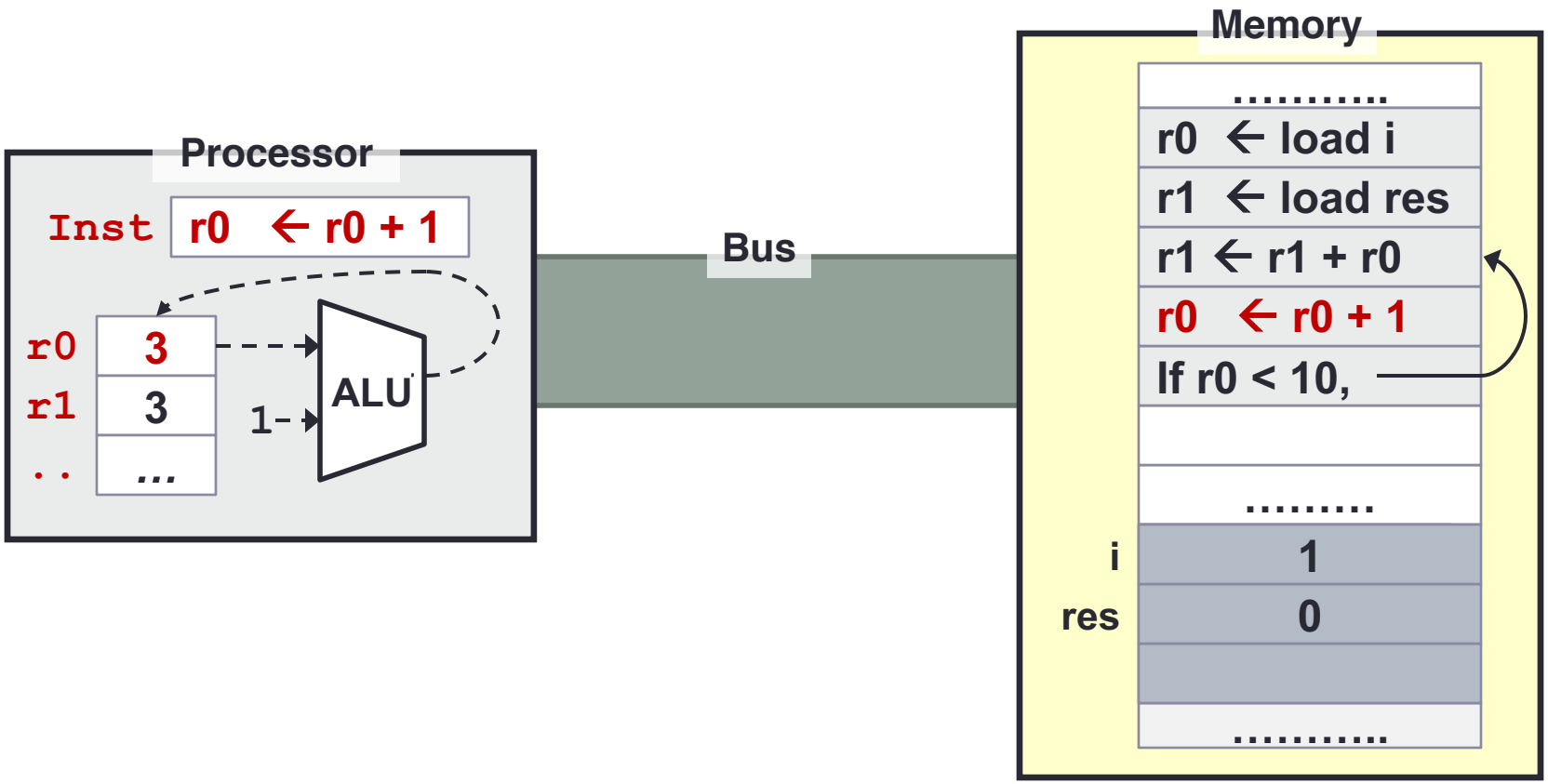
Walkthrough: Looping!

- Since the condition succeeded, execution will repeat from the indicated position



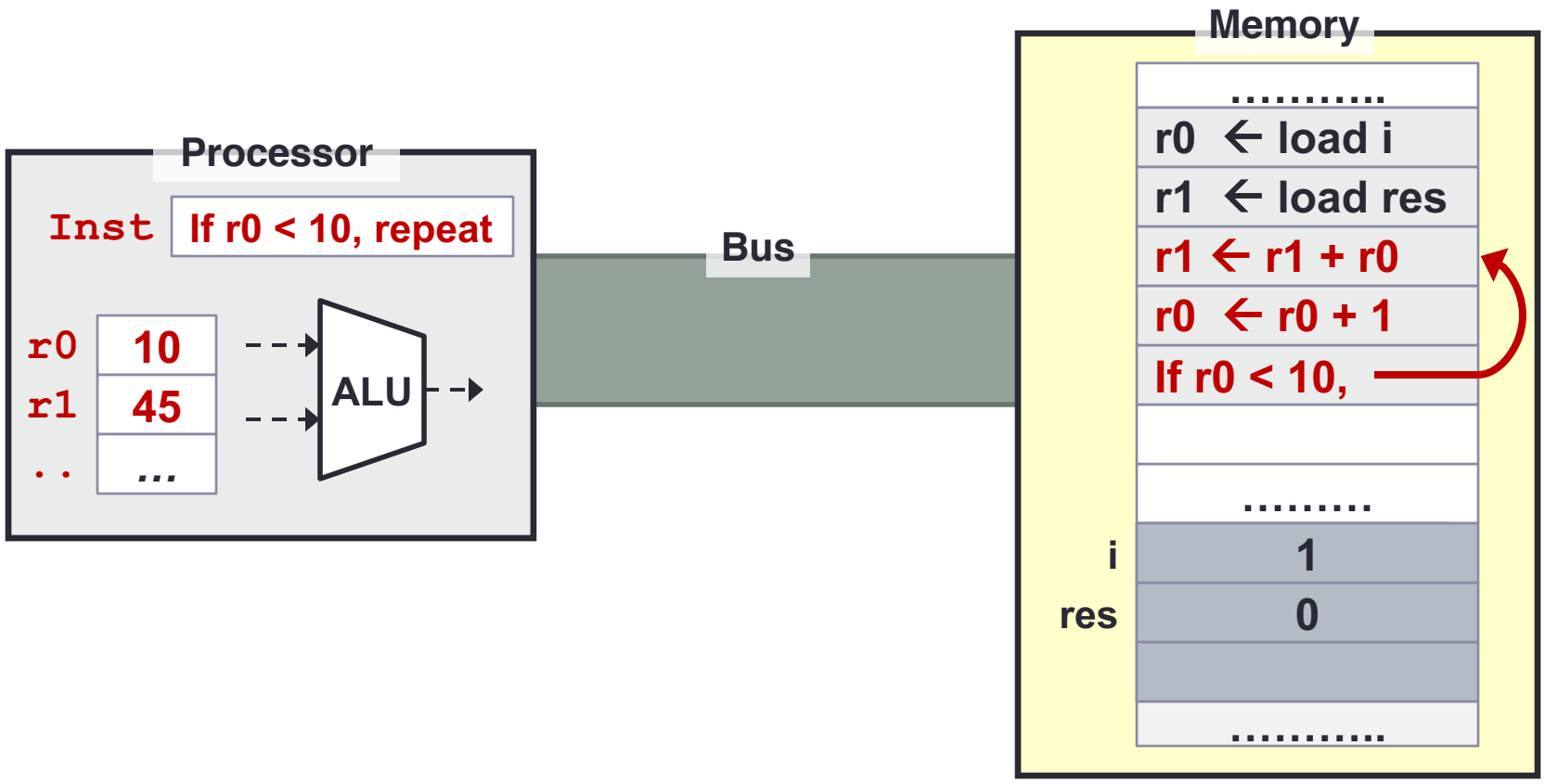
Walkthrough: Looping!

- Execution will continue sequentially:
  - Until we see another control flow instruction!



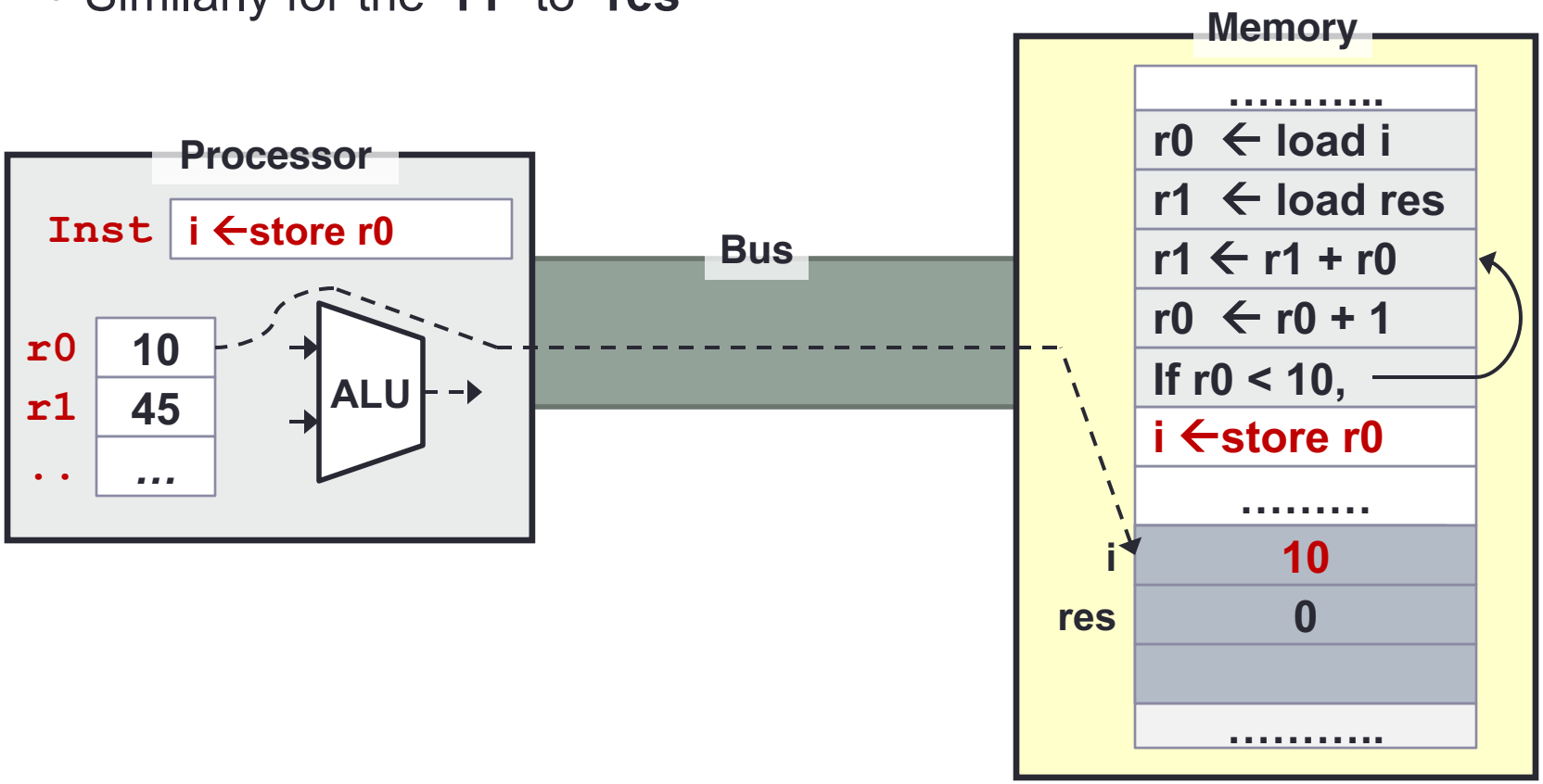
Walkthrough: Control flow instruction

- The three instructions will be repeated until the condition fails



Walkthrough: Memory instruction

- We can now move back the values from register to their "home" in memory
  - Similarly for the "**r1**" to "**res**"



Summary of observations

- The stored-memory concept:
  - Both **instruction** and **data** are stored in memory
- The load-store model:
  - Limit memory operations and relies on registers for storage during execution
- The major types of assembly instruction:
  - **Memory:** Move values between memory and register
  - **Calculation:** Arithmetic and other operations
  - **Control flow:** Changes the sequential execution

MIPS ASSEMBLY LANGUAGE PART I

Overview

- Closer look at internal storage:
  - General Purpose Registers
- MIPS assembly language:
  - Basics
  - Arithmetic and logical instructions
  - Memory instructions (in Part II)
  - Control flow instructions (in Part II)

General Purpose Registers (1/2)

- Fast memories in the processor:
  - Data are transferred from memory to registers for faster processing.
- Limited in number:
  - A typical architecture has 16 to 32 registers
  - Compiler associates variables in program with registers.
- Registers have **no data type**
  - Unlike program variables!
  - Machine/Assembly instruction assumes the data stored in the register is the correct type

General Purpose Registers (2/2)

- There are **32 registers** in **MIPS** assembly language:
  - Can be referred by a number (`$0`, `$1`, …, `$31`) OR
  - Referred by a name (eg: `$a0`, `$t1`)

Name	Register number	Usage
`$zero`	0	Constant value 0
`$v0-$v1`	2-3	Values for results and expression evaluation
`$a0-$a3`	4-7	Arguments
`$t0-$t7`	8-15	Temporaries
`$s0-$s7`	16-23	Program variables

Name	Register number	Usage
`$t8-$t9`	24-25	More temporaries
`$gp`	28	Global pointer
`$sp`	29	Stack pointer
`$fp`	30	Frame pointer
`$ra`	31	Return address

$at (register 1) is reserved for the assembler.
$k0-$k1 (registers 26-27) are reserved for the operation system.

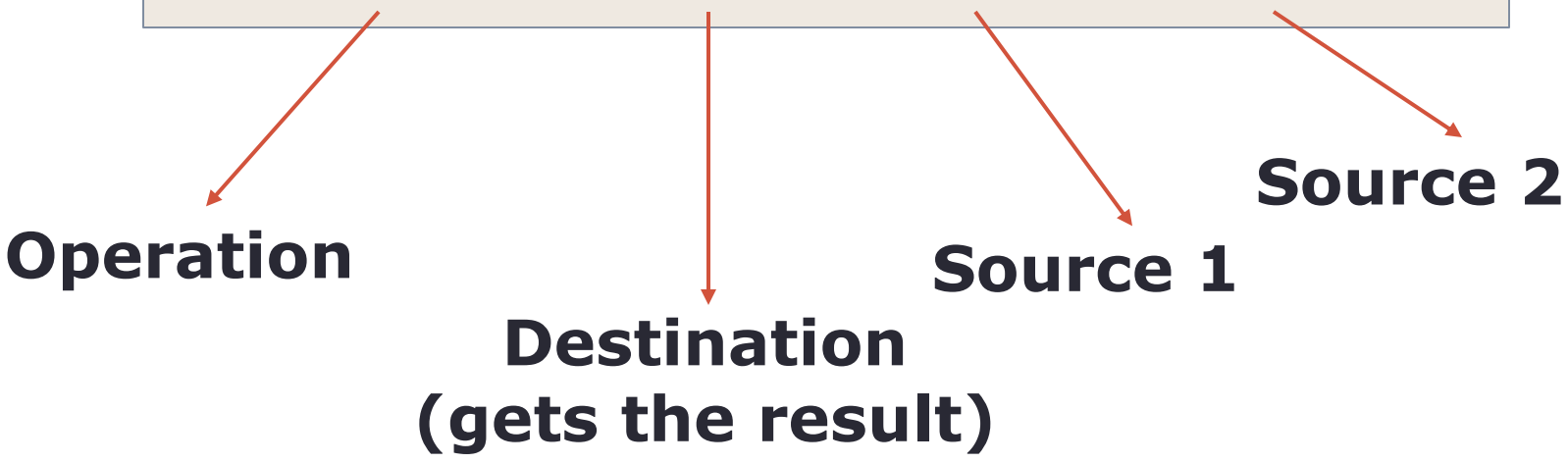
MIPS Assembly Language

- In MIPS assembly language:
  - Each instruction executes a simple command
    - Usually has a counterpart in high level programming languages like C/C++, Java etc
  - Each line of assembly code contains at most 1 instruction
  - **#** (hex-sign) is used for comments
    - Anything from **#** mark to end of line is a comment and will be ignored

```
add $t0, $s1, $s2   # $t0 ← $s1 + $s2
sub $s0, $t0, $s3   # $s0 ← $t0 - $s3
```

General Instruction Syntax

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




```
$s0 = $s1 + $s2
```

Naturally, most of the MIPS arithmetic/logic operations have three operands: **2 sources** + **1 destination**

Arithmetic Operation: **Addition**

C Statement	MIPS Assembly Code
`a = b + c;`	`add $s0, $s1, $s2`

- We assume the values of "**a**", "**b**" and "**c**" are loaded into registers "**$s0**", "**$s1**" and "**$s2**"
  - Known as **variable mapping**
  - Actual code to perform the loading will be shown later in ***memory instruction***
- Important concept:
  - MIPS arithmetic operations are mainly register-to-register

Arithmetic Operation: **Subtraction**

C Statement	MIPS Assembly Code

`a = b - c;`	`sub $s0, $s1, $s2`
	
	`$s0` → variable `a`
	`$s1` → variable `b`
	`$s2` → variable `c`

- Positions of `$s1` and `$s2` (i.e., source1 and source2) are important for subtraction

Complex Expression

C Statement	MIPS Assembly Code

`a = b + c - d;`	`??? ??? ???`
	`$s0` → variable `a`
	`$s1` → variable `b`
	`$s2` → variable `c`
	`$s3` → variable `d`

- A single MIPS instruction can handle at most two source operands

➔ Need to break complex statement into multiple MIPS instructions

MIPS Assembly Code
`add $t0, $s1, $s2  # tmp = b + c` `sub $s0, $t0, $s3  # a = tmp - d`

Use temporary registers **$t0** to **$t7** for intermediate results

Complex Expression: Example

C Statement	Variable Mappings
`f = (g + h) - (i + j);`	`$s0` → variable **f** `$s1` → variable **g** `$s2` → variable **h** `$s3` → variable **i** `$s4` → variable **j**

- Break it up into multiple instructions
  - Use two temporary registers **`$t0`**, **`$t1`**

```
add $t0, $s1, $s2   # tmp0 = g + h
add $t1, $s3, $s4   # tmp1 = i + j
sub $s0, $t0, $t1   # f = tmp0 - tmp1
```

Exercise: **Complex statement**

C Statement	Variable Mappings
`z = a + b + c + d;`	`$s0` → variable **a** `$s1` → variable **b** `$s2` → variable **c** `$s3` → variable **d** `$s4` → variable **z**

C Statement	Variable Mappings
`z = (a - b) + c;`	`$s0` → variable **a** `$s1` → variable **b** `$s2` → variable **c** `$s3` → variable **z**

Constant / Immediate Operands

C Statement	MIPS Assembly Code
`a = a + 4;`	`addi $s0, $s0, 4`

- Immediate values are numerical constants
  - Frequently used in operations
  - MIPS supplies a set of operations specially for them

- "Add immediate" (**`addi`**)
  - Syntax is similar to **`add`** instruction; but source2 is a constant instead of register
  - **The constant ranges from [-2^{15} to $2^{15}-1$]**

Register Zero

- The number zero (0), appears very often in code
  - Provide register zero (`$0` or `$zero`) which always have the **value 0**

C Statement	MIPS Assembly Code
`f = g;`	`add $s0, $s1, $zero` `$s0` → variable **f** `$s1` → variable **g**

- The above assignment is so common that MIPS has an equivalent **pseudo instruction (move)**:

MIPS Assembly Code
`move $s0, $s1`

Pseudo-Instruction
"Fake" instruction that get translated to corresponding MIPS instruction(s).
Provided for convenience in coding only.

Logical Operations

- Arithmetic instructions view the content of a register as a single quantity (signed or unsigned integer)
- **New perspective:**
  - View register as 32 raw bits rather than as a single 32-bit number

  ➔ Possible to operate on individual bytes or bits within a word

Logical operation	C operator	Java operator	MIPS instruction
Shift Left	<<	<<	`sll`
Shift right	>>	>>, >>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

Logical Operation: **Shifting** (1/2)

Opcode: `sll` (shift left logical)

Move all the bits in a word to the left by a number of positions; fill the emptied positions with zeroes.

- E.g. Shift bits in **`$s0`** to the left by 4 positions

`$s0` `0000 1000 0000 0000 0000 0000 0000 1001`

```
sll $t2, $s0, 4   # $t2 = $s0<<4
```

`$t2`

Logical Operation: **Shifting** (2/2)

Opcode: `srl` (shift right logical)

Shifts right and fills emptied positions with zeroes

- What is the equivalent math operations for shift left/right *n* bits? Answer:
- Shifting is faster than multiplication/division
  - ➔Good compiler translates such multiplication/division into shift instructions

C Statement	MIPS Assembly Code
`a = a * 8;`	`sll $s0, $s0, 3`

Logical Operation: **Bitwise AND**

Opcode: `and` (bitwise AND)

Bitwise operation that leaves a 1 only if both the bits of the operands are 1

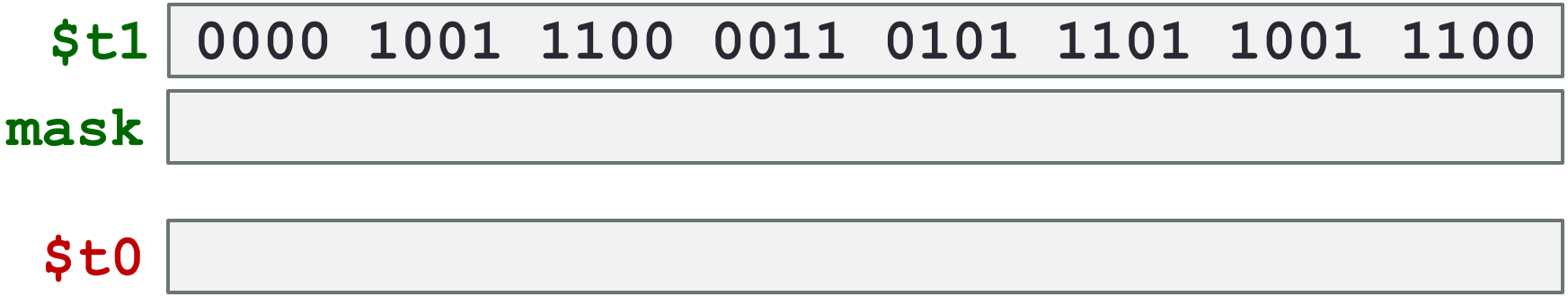
- E.g.: `and $t0, $t1, $t2`

`$t1`	0000 0000 0000 0000 0000 1101 0000 0000
`$t2`	0000 0000 0000 0000 0011 1100 0000 0000
`$t0`	0000 0000 0000 0000 0000 1100 0000 0000

- `and` can be used for masking operation:
  - Place **0s** into the positions to be ignored ➔ bits will turn into 0s
  - Place **1s** for interested positions ➔ bits will remain the same as the original.

Exercise: **Logical Operation**

- We are interested in the last 12 bits of the word in register **`$t1`**
  - **Q:** What's the mask to use?



Notes:
`and` instruction has an immediate version, i.e. **`andi`**

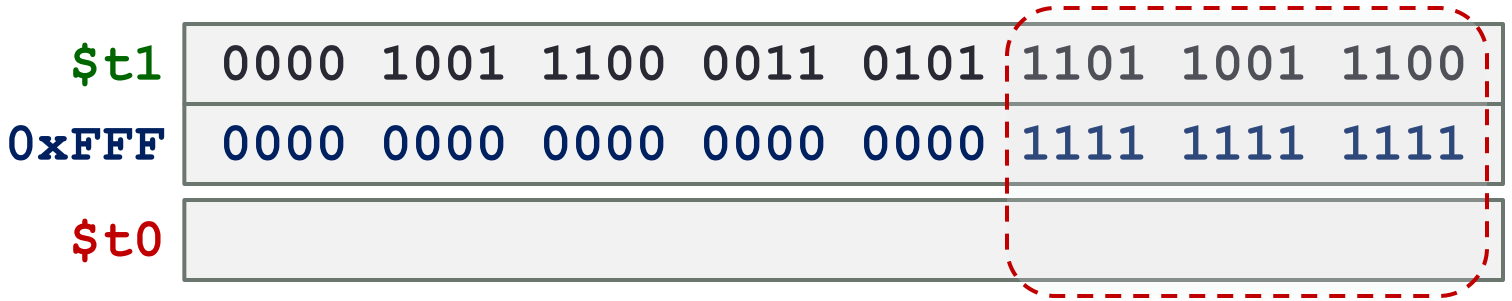
Logical Operation: **Bitwise OR**

Opcode: `or` (bitwise OR)

Bitwise operation that that places a 1 in the result if either operand bit is 1

Example: `or $t0, $t1, $t2`

- Can be used to force certain bits to 1s
- E.g.: `ori $t0, $t1, 0xFFF`



Logical Operation: **Bitwise NOR**

- **Strange fact 1:**
  - There is no **NOT** operation in MIPS to toggle the bits (1 → 0, 0 → 1)
  - However, **NOR** operation is provided…..

Opcode:	`nor` (bitwise NOR)
Example:	`nor $t0, $t1, $t2`

- **Question:** How do we get NOT operation?
- **Question:** Why do you think is the reason for not providing NOT operation?

Logical Operation: **Bitwise XOR**

Opcode:	`xor` (bitwise XOR)
Example:	`xor $t0, $t1, $t2`

- **Question:** Can we also get **NOT** operation from **XOR**?

- **Strange Fact 2:**
  - There is no **NORI**, but there is **XORI** in MIPS
  - Why?

Large Constant: Case Study

- **Question:** How to load a 32-bit constant into a register? e.g `10101010 10101010 11110000 11110000`

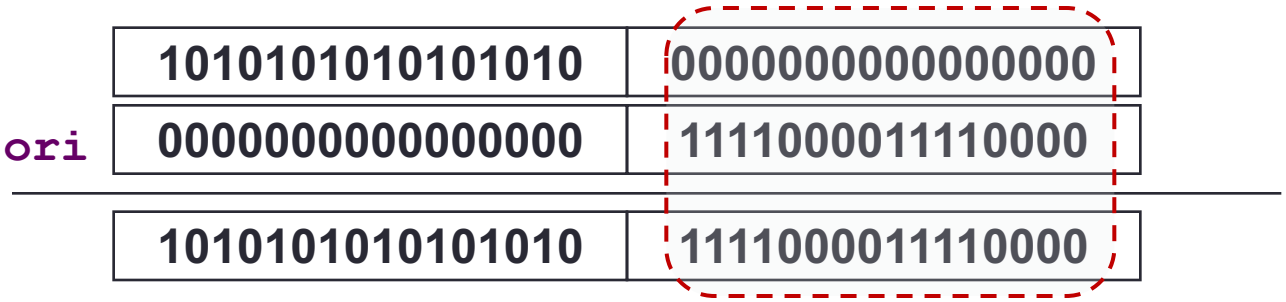
---

1. Use "load upper immediate" (`lui`) to set the upper 16-bit:

```
lui   $t0, 0xAAAA        #1010101010101010
```

2. Use "or immediate" (`ori`) to set the lower-order bits:

```
ori   $t0, $t0, 0xF0F0 #1111000011110000
```



MIPS Basic Instructions Checklist

Operation	Opcode in MIPS	Immediate Version (if applicable)
Addition	`add $s0, $s1, $s2`	`addi $s0, $s1, C16`$_{2s}$ `C16`$_{2s}$ is [-2^{15} to $2^{15}-1$]
Subtraction	`sub $s0, $s1, $s2`	
Shift left logical	`sll $s0, $s1, C5` `C5` is [$0..2^5-1$]	
Shift right logical	`srl $s0, $s1, C5`	
AND bitwise	`and $s0, $s1, $s2`	`andi $s0, $s1, C16` `C16` is a 16-bit pattern
OR bitwise	`or $s0, $s1, $s2`	`ori $s0, $s1, C16`
NOR bitwise	`nor $s0, $s1, $s2`	
XOR bitwise	`xor $s0, $s1, $s2`	`xori $s0, $s1, C16`

Reading Assignment

- Instructions: Language of the Computer
  - COD Chapter 2, pg 46-53, 58-71, 95-96. (3rd edition)
  - COD Chapter 2, pg 74-81, 86-87, 94-104. (4th edition)

Q&A