CS2100 COMPUTER ORGANISATION

http://www.comp.nus.edu.sg/~cs2100/

Lecture #7

MIPS

Part I: Introduction

NUS National University of Singapore | School of Computing

Questions?

Ask at
https://sets.netlify.app/module/676ca3a07d7f5ffc1741dc65

OR

Scan and ask your questions here!
(May be obscured in some slides)

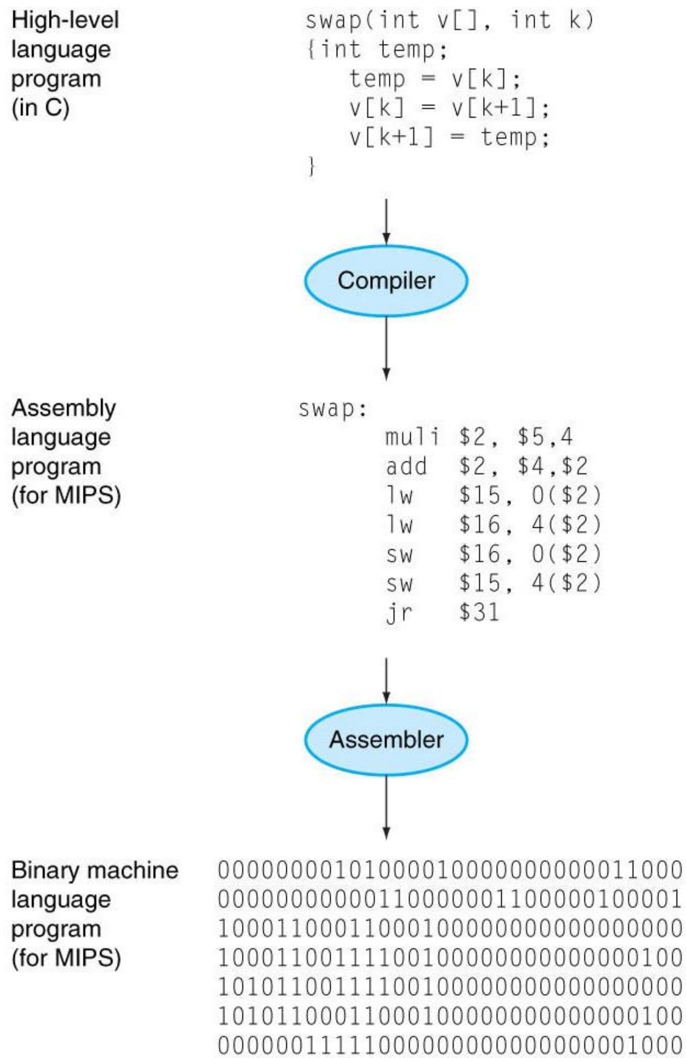
Lecture #7: MIPS Part 1: Introduction (1/2)

1. Instruction Set Architecture
2. Machine Code vs Assembly Language
3. Walkthrough
4. General Purpose Registers
5. MIPS Assembly Language
   - 5.1 General Instruction Syntax
   - 5.2 Arithmetic Operation: Addition
   - 5.3 Arithmetic Operation: Subtraction
   - 5.4 Complex Expression
   - 5.5 Constant/Immediate Operands
   - 5.6 Register Zero ($0 or $zero)

Lecture #7: MIPS Part 1: Introduction (1/2)

5. MIPS Assembly Language
   - 5.7 Logical Operations: Overview
   - 5.8 Logical Operations: Shifting
   - 5.9 Logical Operations: Bitwise AND
   - 5.10 Logical Operations: Bitwise OR
   - 5.11 Logical Operations: Bitwise NOR
   - 5.12 Logical Operations: Bitwise XOR
6. Large Constant: Case Study
7. MIPS Basic Instructions Checklist

Recap



- 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

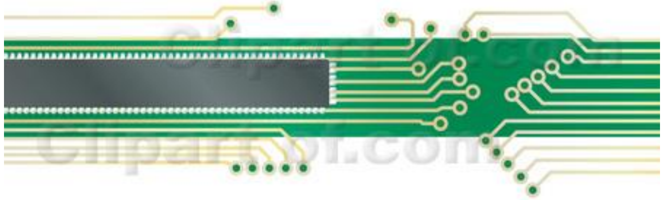
1. Instruction Set Architecture (1/2)

- **Instruction Set Architecture (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)



1. 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 be executed on any of these implementations

2. Machine Code vs Assembly Language

Machine Code	Assembly Language
Instructions in binary e.g.: `1000 1100 1010 0000` → Add two numbers	Human readable e.g.: `add A, B` → Add two numbers
Hard and tedious to code	Easier to write than machine code, symbolic version of machine code
`1000 1100 1010 0000` ← ASSEMBLER ← `add A, B`	
May also be written in hexadecimal for a more human-readable format	May provide '**pseudo-instructions**' as syntactic sugar
	When considering performance, only real instructions are counted

NOTE:
Syntactic "sugar" is basically a translation scheme from a language to the ***same*** language (*e.g.,* from C to C or in this case from MIPS to MIPS). The pseudo-instructions are then translated into one or more real instructions.
For example, in MIPS, we have `move $rd, $rs` being translated into `add $rd, $rs, $zero`.

3. Walkthrough: An Example Code (1/15)

- 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

compile to

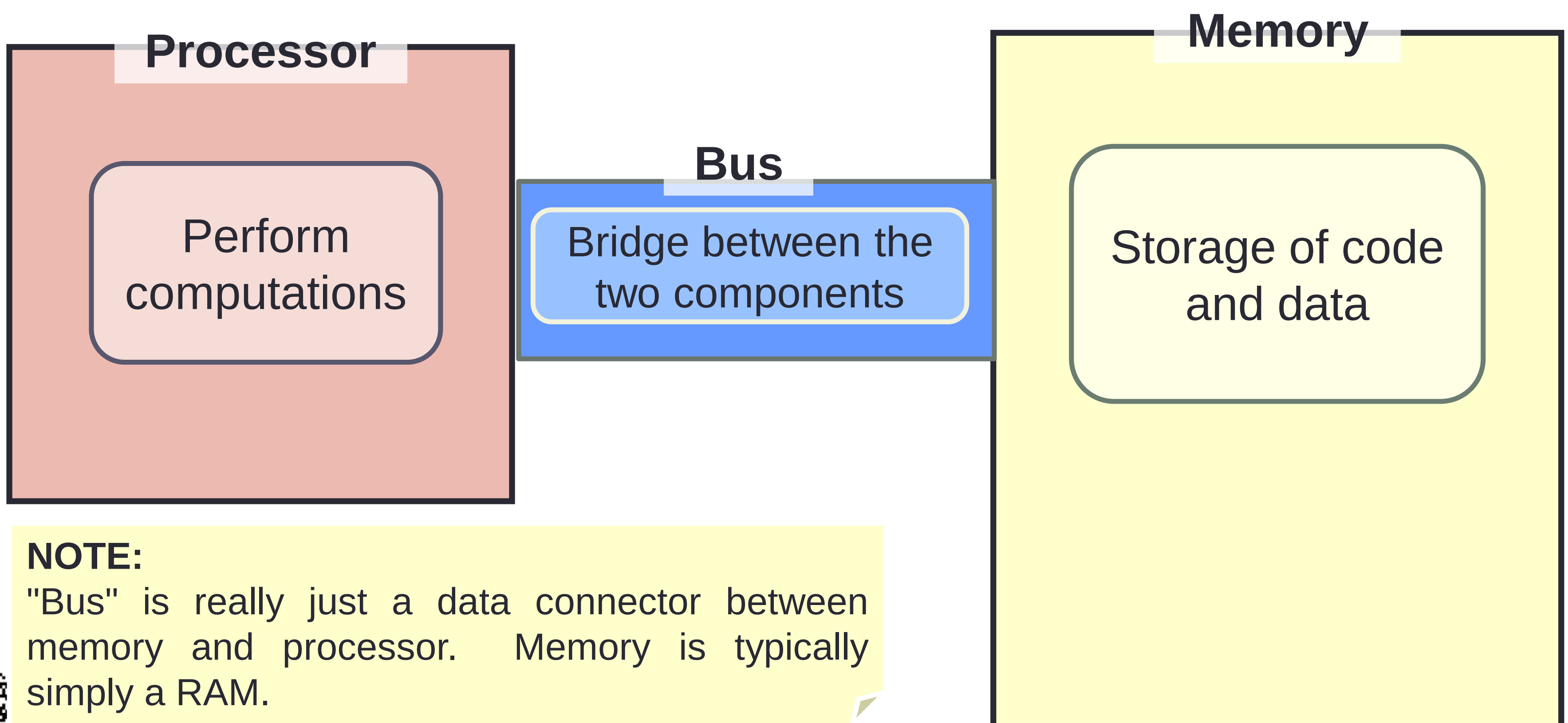
```
res ← res + i
i ← i + 1
if i < 10, repeat
```

“Assembly” Code

NOTE:
Not a real "assembly" language but hopefully instructive enough for our purpose.

3. Walkthrough: The Components (2/15)

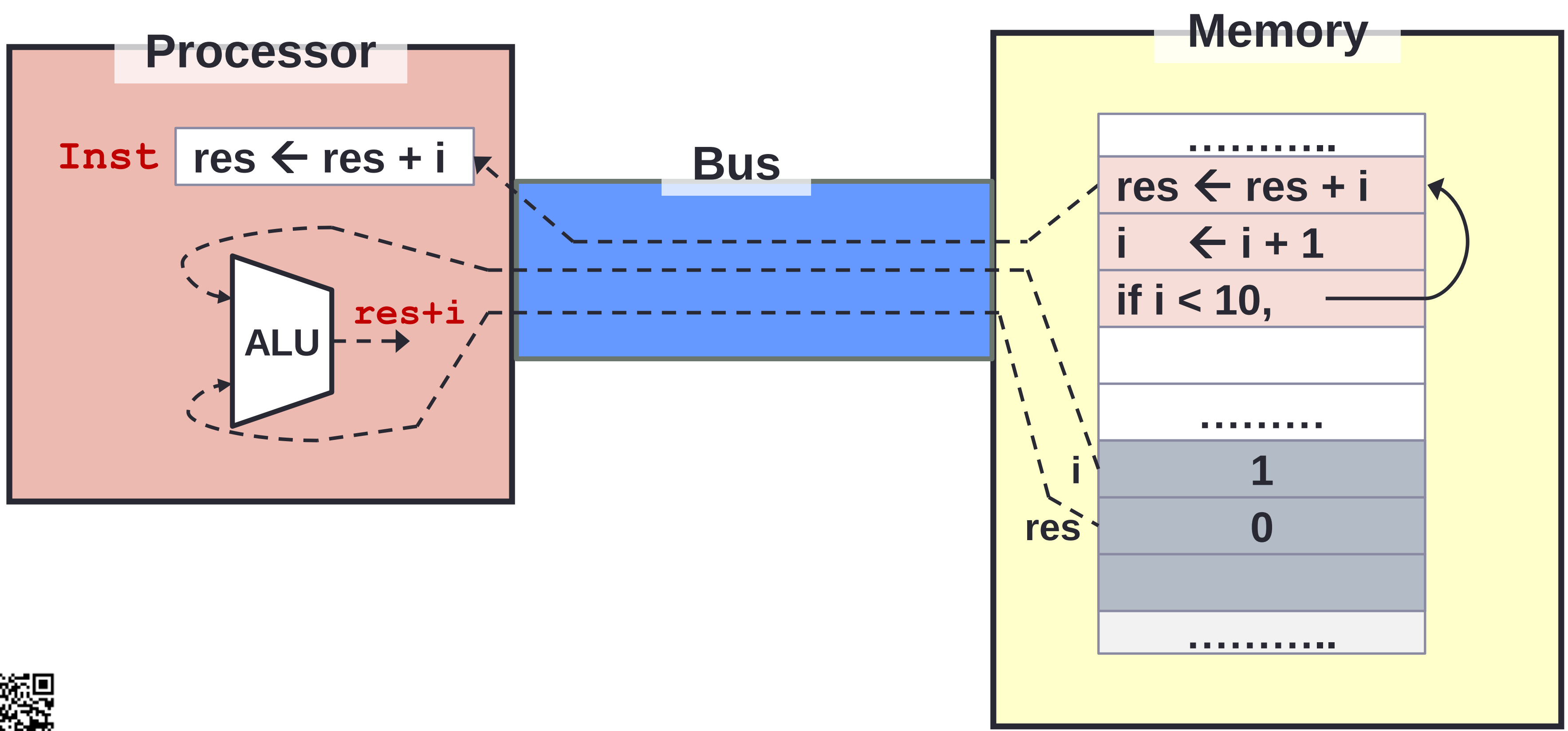
- The two major components in a computer
  - **Processor** and **Memory**
  - Input/Output devices omitted in this example



NOTE:
"Bus" is really just a data connector between memory and processor. Memory is typically simply a RAM.

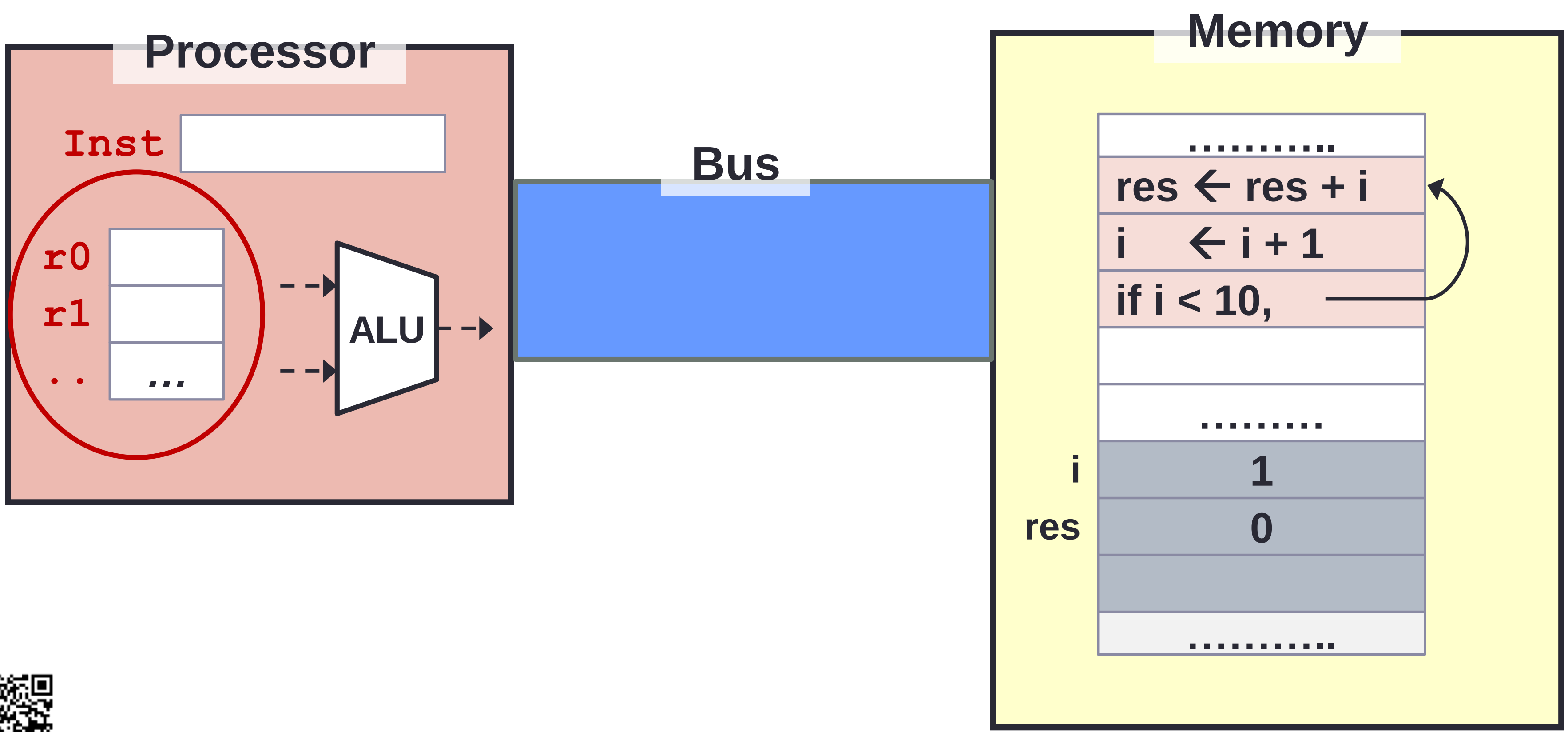
3. Walkthrough: The Code in Action (3/15)

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



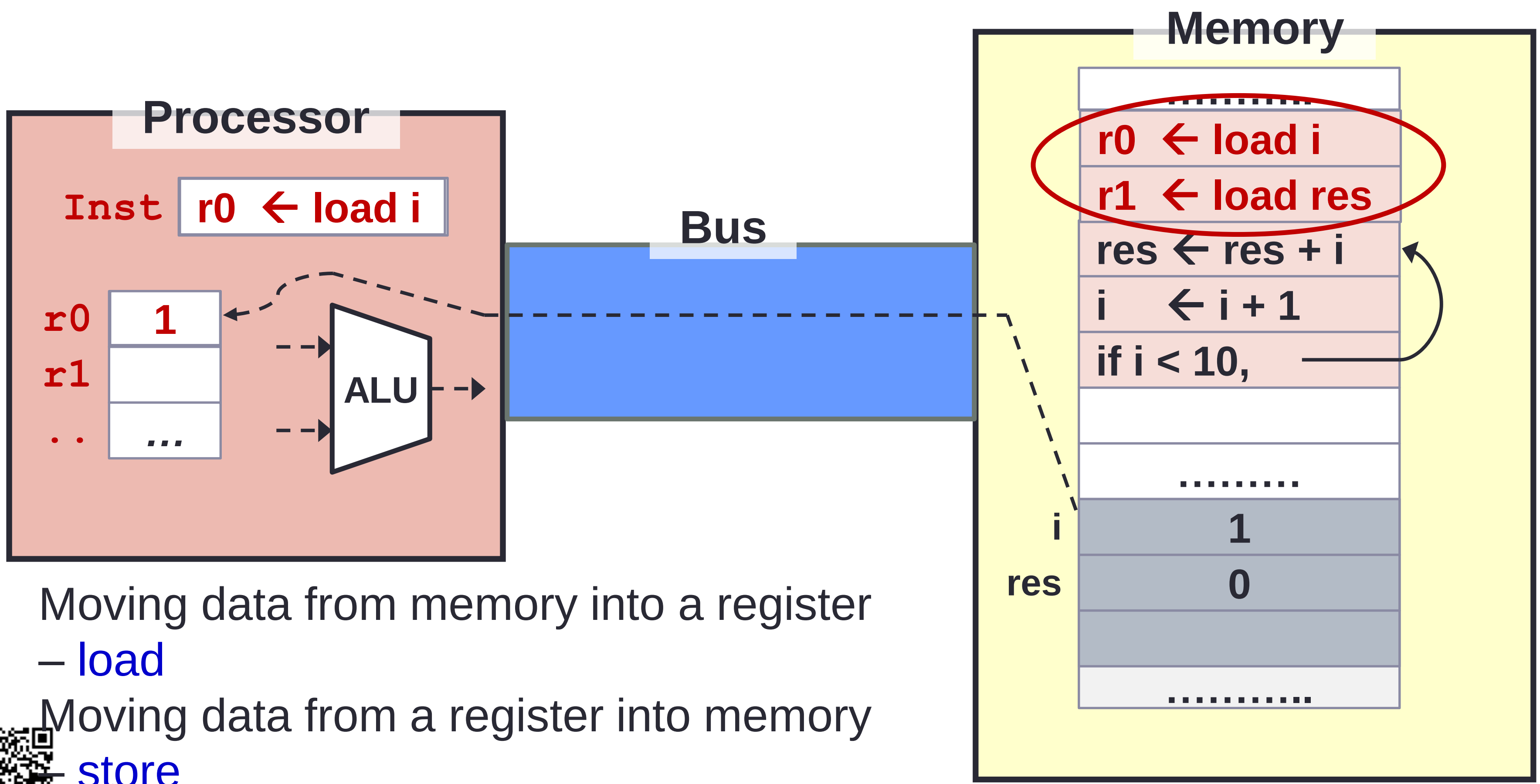
3. Walkthrough: Memory Access is Slow! (4/15)

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



3. Walkthrough: Memory Instruction (5/15)

- Need instruction to move data into registers
  - Also to move data from registers to memory later

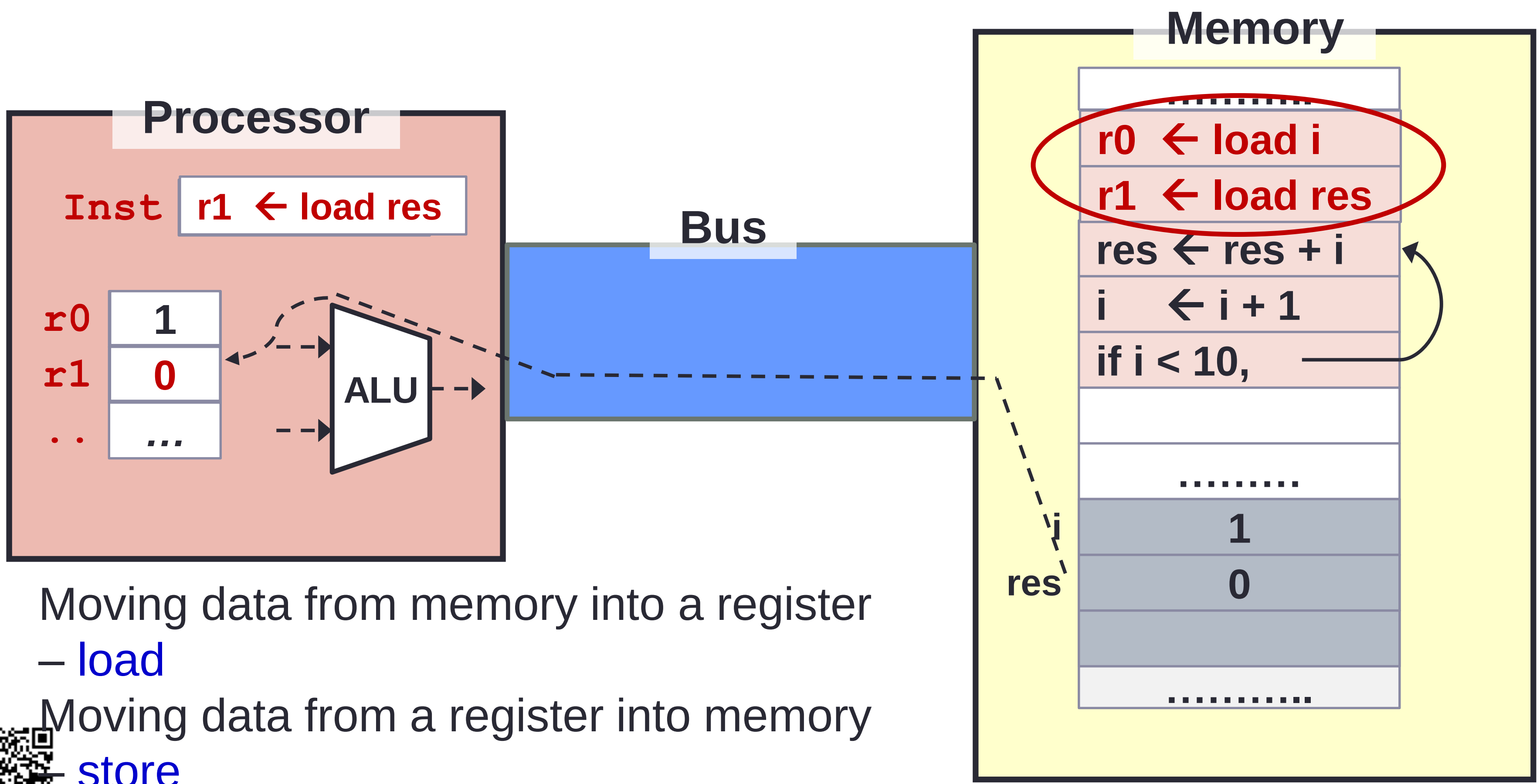


Moving data from memory into a register – load

Moving data from a register into memory – store

3. Walkthrough: Memory Instruction (6/15)

- Need instruction to move data into registers
  - Also to move data from registers to memory later

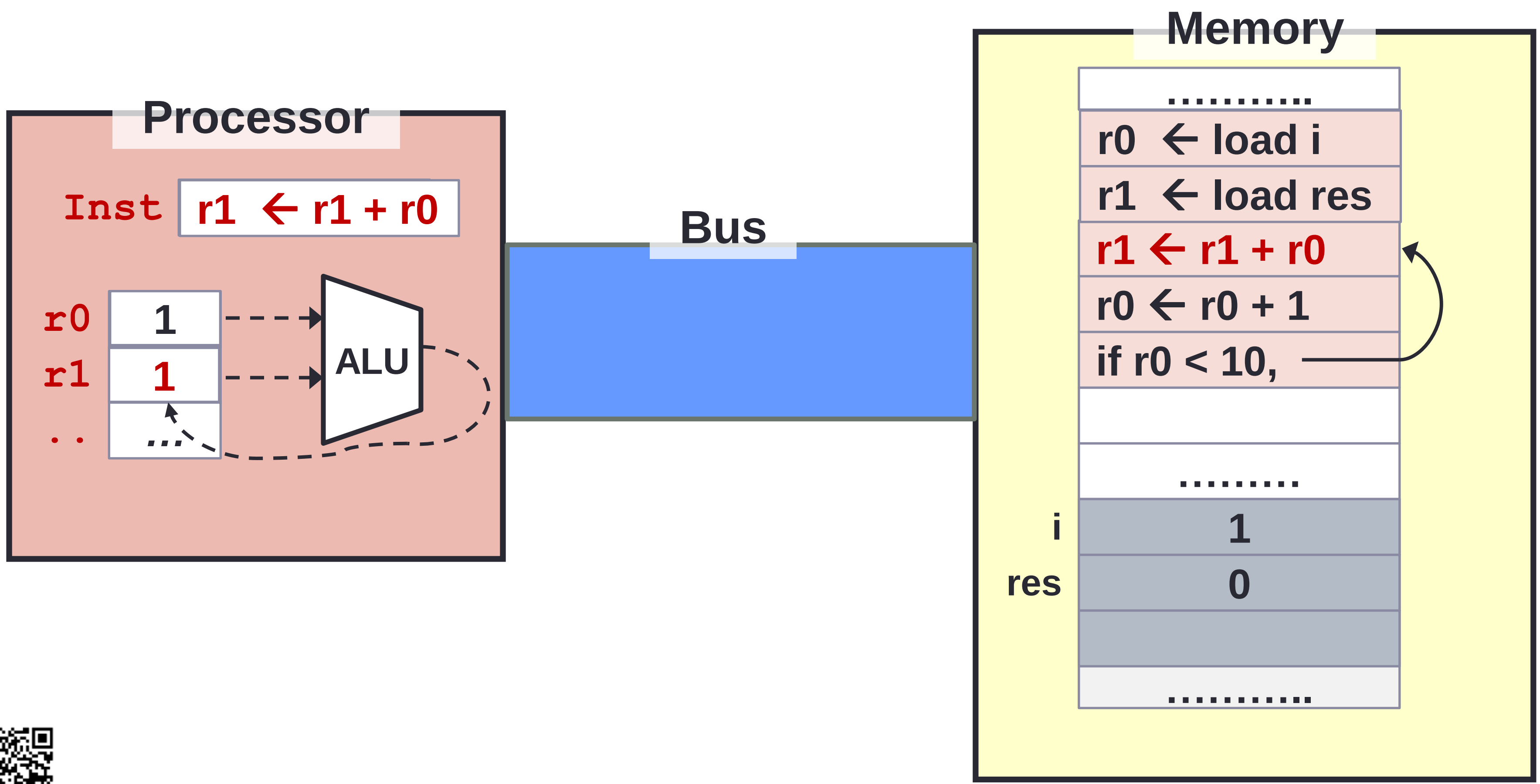


Moving data from memory into a register – load

Moving data from a register into memory – store

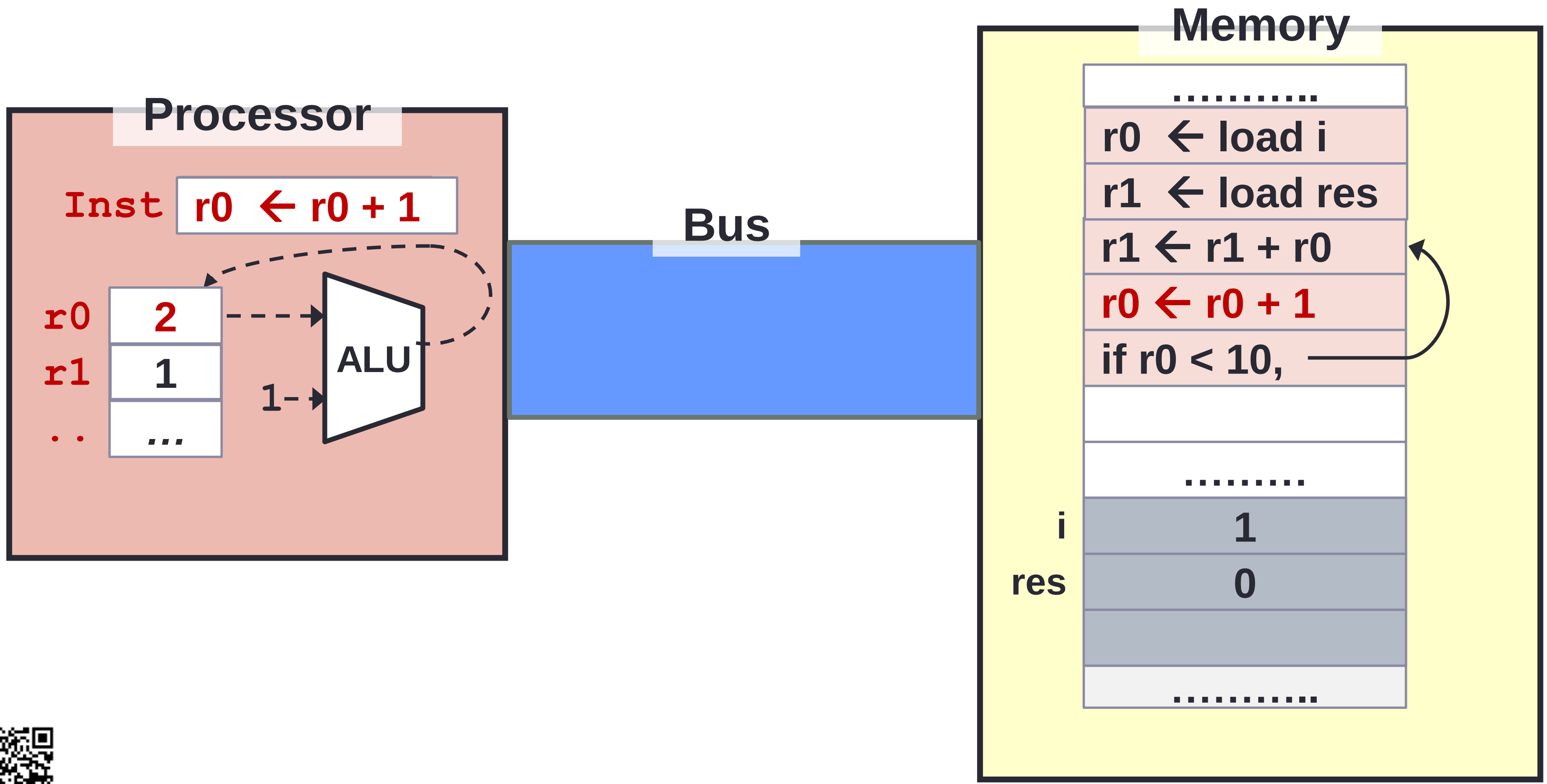
3. Walkthrough: Reg-to-Reg Arithmetic (7/15)

- Arithmetic operations can now work directly on registers only (much faster!)



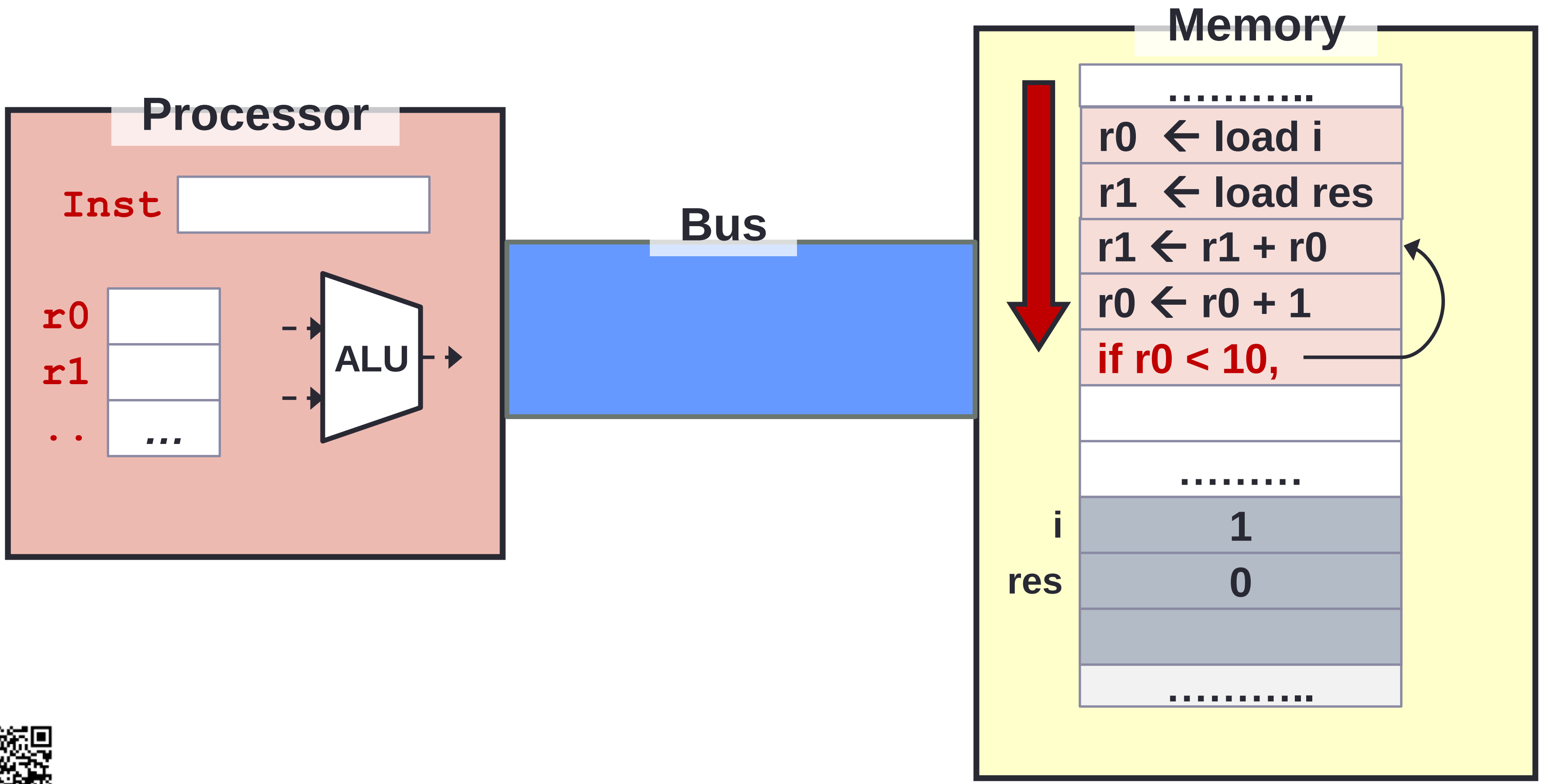
3. Walkthrough: Reg-to-Reg Arithmetic (8/15)

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



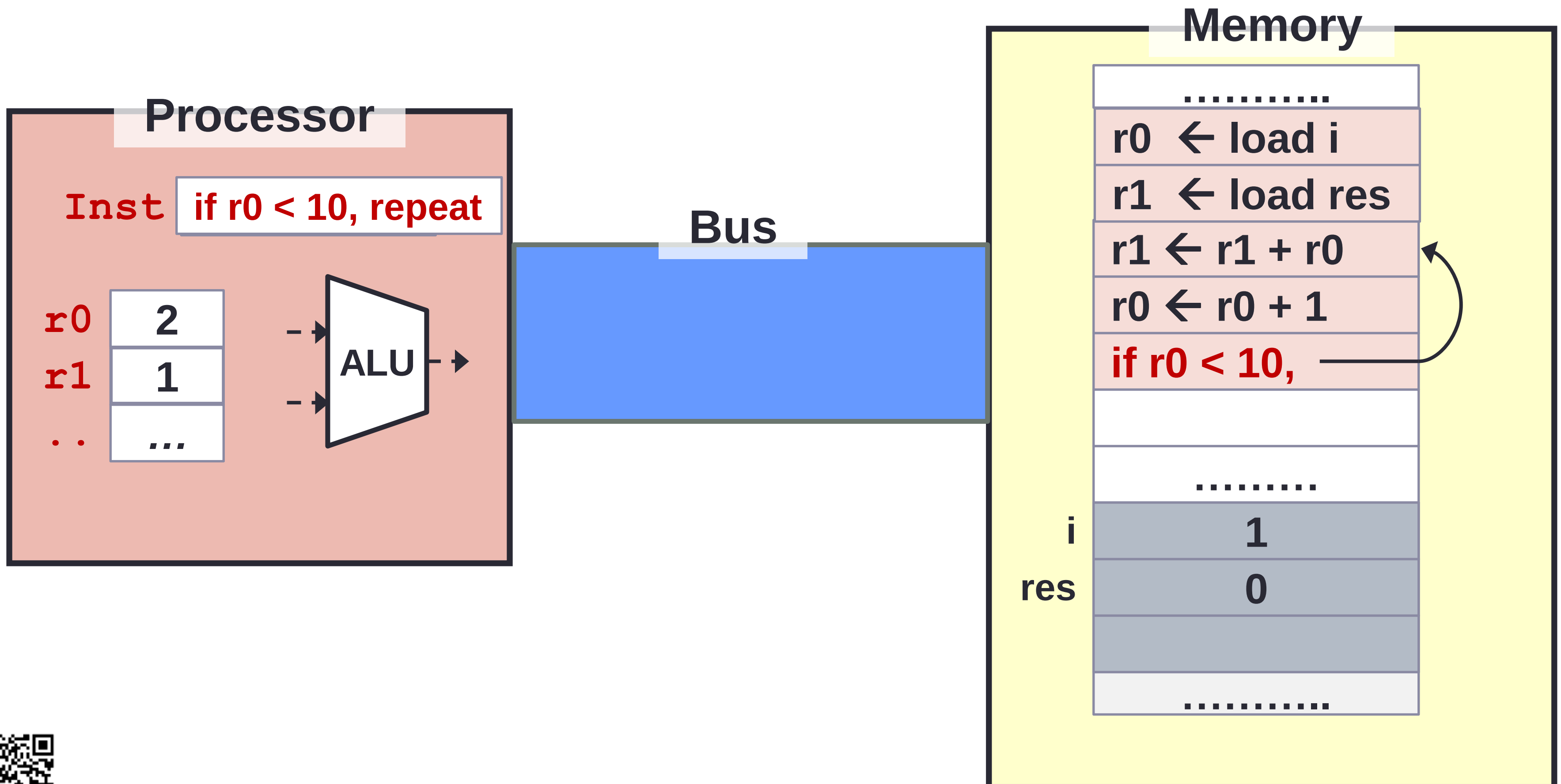
3. Walkthrough: Execution Sequence (9/15)

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



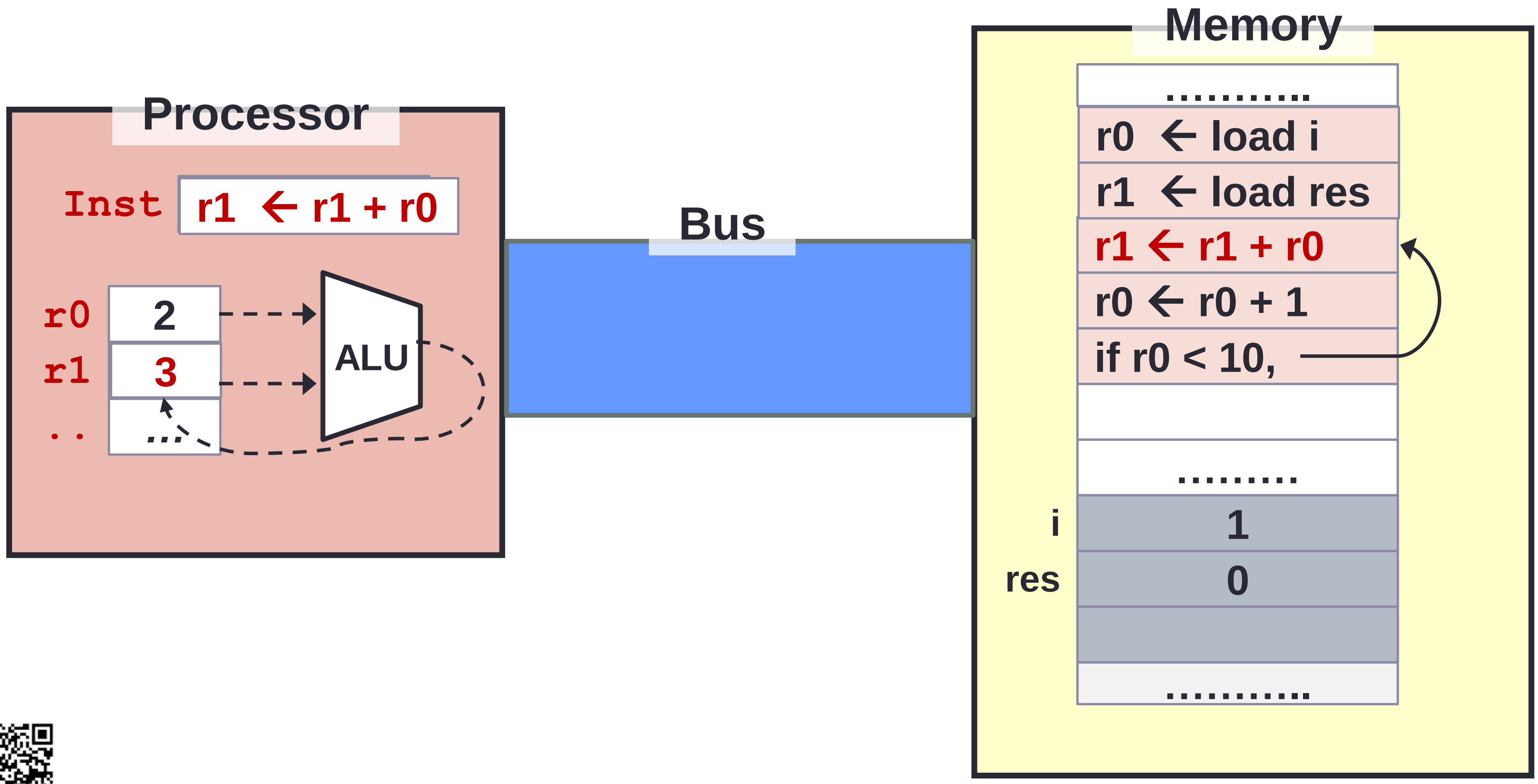
3. Walkthrough: Control Flow (10/15)

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



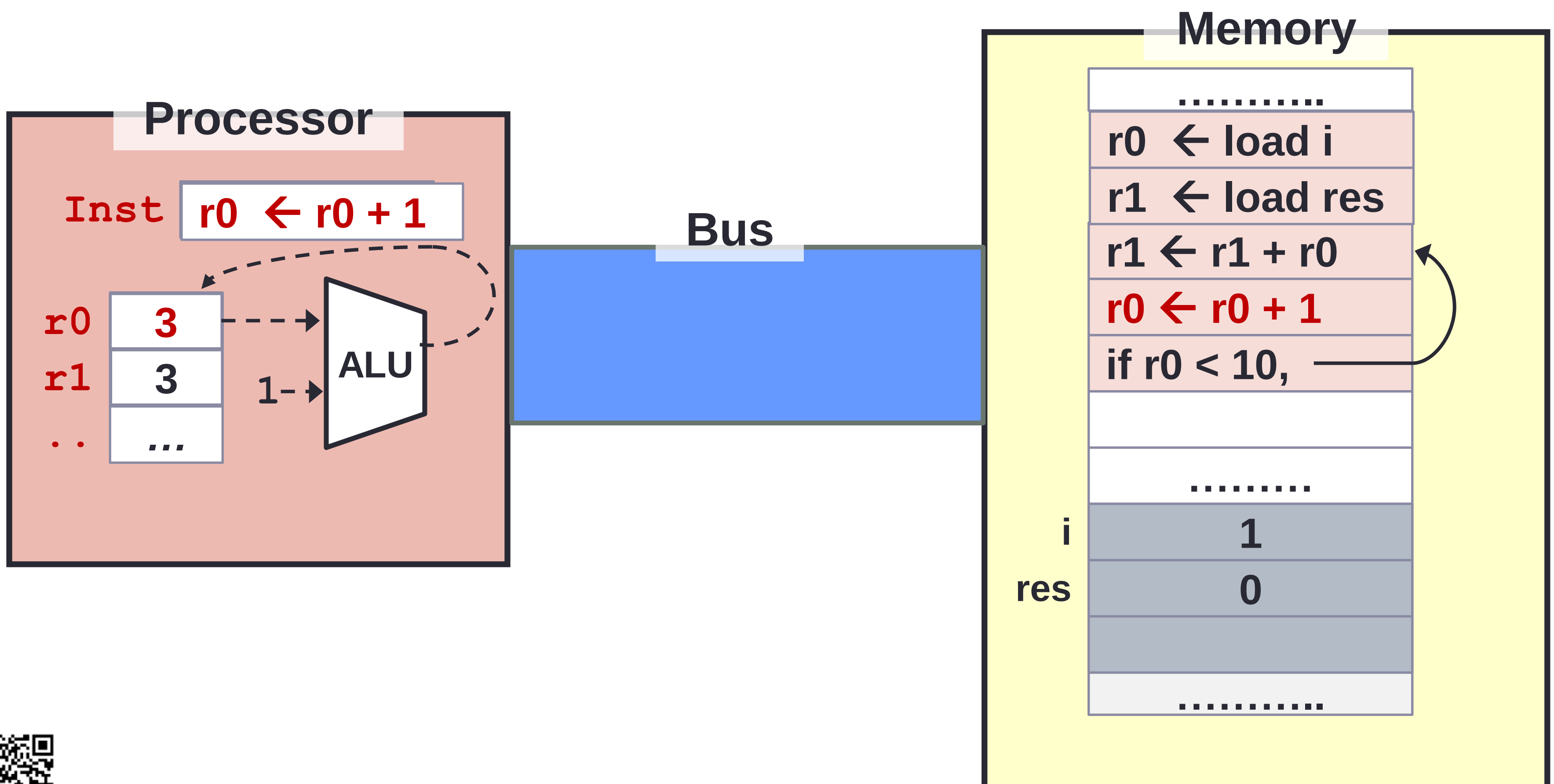
3. Walkthrough: Looping! (11/15)

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



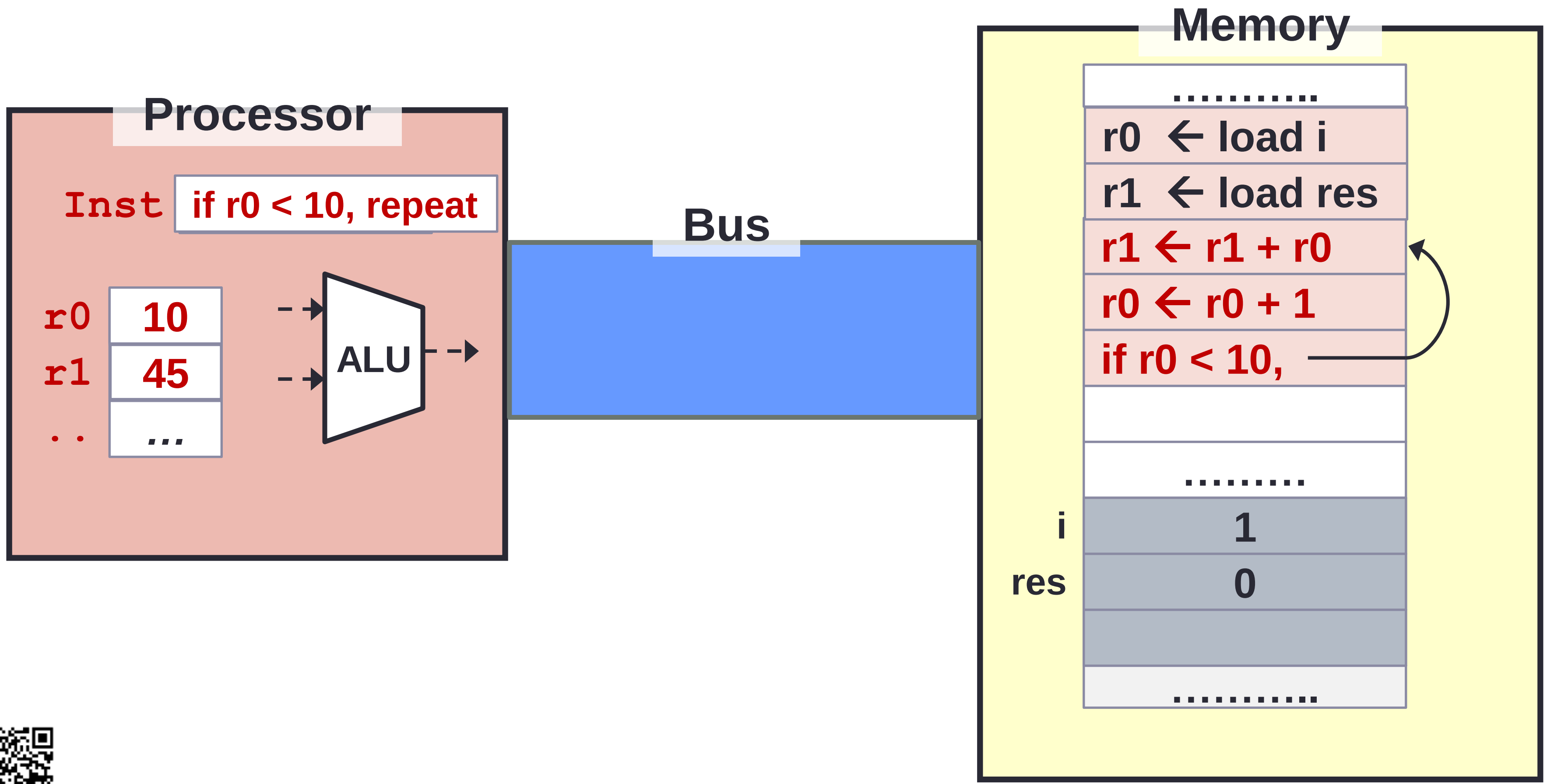
3. Walkthrough: Looping! (12/15)

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



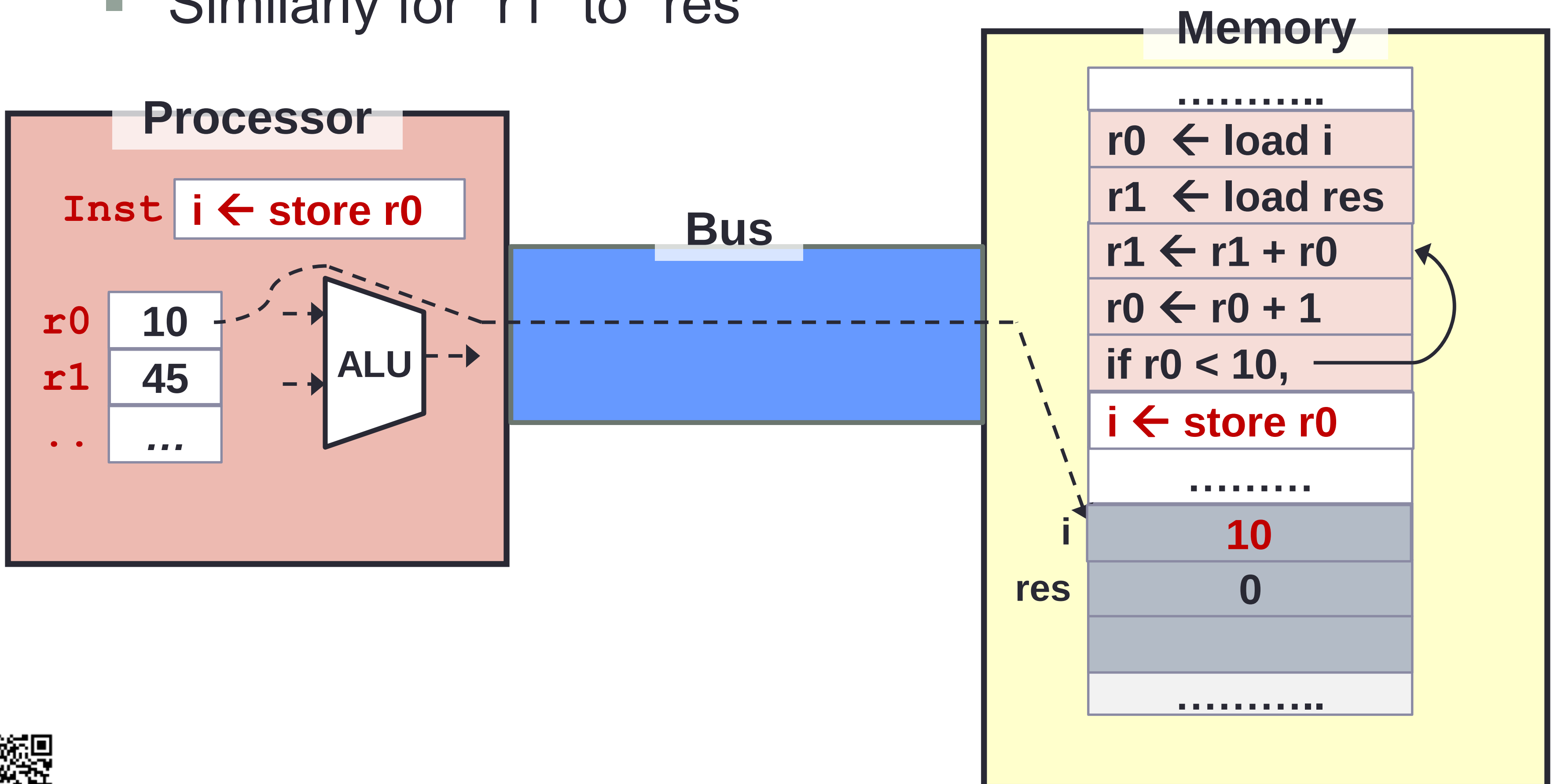
3. Walkthrough: Looping! (13/15)

- The three instructions will be repeated until the condition fails



3. Walkthrough: Memory Instruction (14/15)

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



3. Walkthrough: Summary (15/15)

- 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 registers
  - **Calculation:** Arithmetic and other operations
  - **Control flow:** Change the sequential execution

NOTE:
A typical assembly code structure is: (1) load, (2) compute, (3) store.
We will assume in this module that we have enough register to store all variables in our program.

4. 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 of the correct type

4. 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 operating system.

5. 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 **#** to end of line is a comment and will be ignored by the assembler

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

5.1 General Instruction Syntax

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

Operation (op)

Destination (gets the result)

Source 1

Source 2

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

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

5.2 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

NOTE:
The variable-to-register mapping deals with step (1) (*i.e., load*) and step (3) (*i.e., store*). All computations are assumed to be in register for this set of instructions. We will talk about load/store in the next lecture.

5.3 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

NOTE:

`sub $s0, $s1, $s2`

is basically

`$s0 = $s1 - $s2`

5.4 Complex Expression (1/3)

C Statement	MIPS Assembly Code

`a = b + c - d;`
 	
`t0 = b  + c;`
`a  = t0 - 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 a 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

5.4 Complex Expression: Example (2/3)

C Statement	Variable Mappings

`f = (g + h) - (i + j);`
 	
`t0 = g + h;`
`t1 = i + j;`
`f = t0 - t1;`
	`$s0` → variable `f`
	`$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
```

5.4 Complex Expression: Exercise (3/3)

C Statement	Variable Mappings

`z = a + b + c + d;`
`add $s4, $s0, $s1`
`add $s4, $s4, $s2`
`add $s4, $s4, $s3`
	`$s0` → variable `a`
	`$s1` → variable `b`
	`$s2` → variable `c`
	`$s3` → variable `d`
	`$s4` → variable `z`
	
	`z = a + b;`
	`z = z + c;`
	`z = z + d;`

C Statement	Variable Mappings

`z = (a – b) + c;`
`sub $s3, $s0, $s1`
`add $s3, $s3, $s2`
	`$s0` → variable `a`
	`$s1` → variable `b`
	`$s2` → variable `c`
	`$s3` → variable `z`
	
	`z = a - b;`
	`z = z + c;`

5.5 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 $2^{15}-1$]**

Can you guess what number system is used?

Answer: 16-bit 2s complement number system

There's no **`subi`**. Why?

Answer: Use **`addi`** with negative constant

5.6 Register Zero ($0 or $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;`
`f = g + 0`
	`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 gets translated to corresponding MIPS instruction(s).
Provided for convenience in coding only.

5.7 Logical Operations: Overview (1/2)

- 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 bits or bytes 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`
Bitwise XOR	`^`	`^`	`xor`

*with some tricks **this is "arithmetic" shift in both GCC and Clang (compiler dependent)

5.7 Logical Operations: Overview (2/2)

- Truth tables of logical operations
  - 0 represents false; 1 represents true

AND

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

NOTE: 1 if BOTH a and b are 1. Otherwise 0.

OR

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

NOTE: 0 if BOTH a and b are 0. Otherwise 1.

NOR

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

NOTE: 1 if BOTH a and b are 1. Otherwise 0.

XOR

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

NOTE: 1 if a is NOT the same as b.

5.8 Logical Operations: **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  1011 1000 0000 0000 0000 0000 0000 1001

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

$t2  1000 0000 0000 0000 0000 0000 1001 0000
```

NOTE:
The emptied positions are filled with 0s

5.8 Logical Operations: **Shifting** (2/2)

Opcode: `srl` (shift right logical)

Shifts right and fills emptied positions with zeroes.

- What is the equivalent math operations for shifting left/right *n* bits? Answer: Multiply/divide by 2^n
- 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`

NOTE:
Since 8 = 2^3, we can use a = a << 3.

5.9 Logical Operations: **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`

	`$t1`	0110 0011 0010 1111 0000 1101 0101 1001
mask	`$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.

NOTE:
Bit-mask is setting the irrelevant part to 0 (*i.e., masked*).

5.9 Exercise: **Bitwise AND**

- We are interested in the last 12 bits of the word in register **`$t1`**. Result to be stored in **`$t0`**.
  - **Q:** What's the mask to use?

$t1	0000 1001 1100 0011 0101 1101 1001 1100
mask	0000 0000 0000 0000 0000 1111 1111 1111
$t0	0000 0000 0000 0000 0000 1101 1001 1100

NOTE:
Keep last 12-bits as 1. This is equivalent to `andi $t1, $t1, 0xFFF`.

Notes:
The **`and`** instruction has an immediate version, **`andi`**

5.10 Logical Operations: **Bitwise OR**

Opcode: `or` (bitwise OR)

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

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

- The `or` instruction has an immediate version `ori`
- Can be used to force certain bits to 1s
- E.g.: `ori $t0, $t1, 0xFFF`

`$t1`	0000	1001	1100	0011	0101	1101	1001	1100
`0xFFF`	0000	0000	0000	0000	0000	1111	1111	1111
`$t0`	0000	1001	1100	0011	0101	1111	1111	1111

For `ori $t0, $t1, 0xFFFF` will the upper 16-bits be all 0s or all 1s?

Answer: all 0s (***in other words, this is not sign-extended***)

5.11 Logical Operations: **Bitwise NOR**

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

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

a	b	a NOR b
0	*0*	*1*
0	1	0
1	*0*	*0*
1	1	0

- Question: How do we get a NOT operation?

```
nor $t0, $t0, $zero
```

- Question: Why do you think is the reason for not providing a NOT instruction?

One of design principles: Keep the instruction set small.

5.12 Logical Operations: **Bitwise XOR**

Opcode: `xor` (bitwise XOR)

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

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

Yes, let `$t2` contain all 1s.
`xor $t0, $t0, $t2`

a	b	a XOR b
0	0	0
0	*1*	*1*
1	0	1
1	*1*	*0*

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

NOTE:
A possible reason is that there is not much need for NORI. So there is no reason to add this capability to keep the processor design simple.

6. 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
```

1010101010101010	0000000000000000

Lower-order bits filled with zeros.

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

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

Higher-order bits filled with zeros.

	1010101010101010	0000000000000000
ori	0000000000000000	1111000011110000
	1010101010101010	1111000011110000

7. MIPS Basic Instructions Checklist

Operation	Opcode in MIPS	Meaning
Addition	add $rd, $rs, $rt	$rd = $rs + $rt
	addi $rt, $rs, $C16_{2s}$	$rt = $rs + $C16_{2s}$
Subtraction	sub $rd, $rs, $rt	$rd = $rs - $rt
Shift left logical	sll $rd, $rt, C5	$rd = $rt << C5
Shift right logical	srl $rd, $rt, C5	$rd = $rt >> C5
AND bitwise	and $rd, $rs, $rt	$rd = $rs & $rt
	andi $rt, $rs, C16	$rt = $rs & C16
OR bitwise	or $rd, $rs, $rt	$rd = $rs | $rt
	ori $rt, $rs, C16	$rt = $rs | C16
NOR bitwise	nor $rd, $rs, $rt	$rd = $rs ↓ $rt
XOR bitwise	xor $rd, $rs, $rt	$rd = $rs ^ $rt
	xori $rt, $rs, C16	$rt = $rs ^ C16

C5 is $[0 \text{ to } 2^5-1]$ $C16_{2s}$ is $[-2^{15} \text{ to } 2^{15}-1]$ C16 is a 16-bit pattern

Additional Notes

- `sll` and `srl` only need 5 bits (i.e., `C5`) because shifting by 32-bits empties the register (i.e., set to 0).

- `C16` are NOT sign-extended.
- A possible reason is because it is used for logical operations which typically concern with the bits as it is (*plus, it is treated as raw bits and not number*).

- $\texttt{C16}_{\texttt{2s}}$ are sign-extended.
- Otherwise, addi will not work properly as the processor can only work with 32-bits.

Additional Notes

- You may wonder why we learn C to learn MIPS. The reason is simply because in C, we control the memory. So, the C code match closely to the corresponding MIPS code.
- All other language are too far removed from the underlying memory structure to be useful UNLESS we are only using a subset of those language.
  - But in C, we are forced to use these simpler subset.
  - This will hopefully make more sense once you start "compiling" from C to MIPS on your own.

End of File