3 - MIPS Assembly I

3.0 Recap

3.1 Instruction Set Architecture (ISA)

> 指令集架构（Instruction Set Architecture, ISA）定义了一个计算机系统可以执行的低级机器语言指令集，也就是计算机硬件能够理解和执行的指令。
>
> 指令集架构涵盖了以下几个方面：
>
> 1. **操作和指令**：定义了计算机能够执行的基本操作，如加法、减法、乘法、逻辑操作等。
> 2. **寄存器**：描述了计算机中的数据存储位置，通常分为通用寄存器、状态寄存器等。
> 3. **地址模式**：定义了如何计算数据和指令的存储位置。
> 4. **数据类型**：描述了支持的数据的种类和大小，如整数、浮点数等。
> 5. **异常和中断处理**：定义了当某些事件（如算术溢出、缺页中断）发生时计算机应该如何响应。
>
> 不同的指令集架构会导致计算机的性能、功耗、代码密度等方面的差异。有一些著名的ISA，如x86（由Intel和AMD使用）、ARM（用于大多数移动设备）、MIPS等。
>
> ISA是计算机架构的一个层次，通常分为三个层次：
>
> 1. **高级语言层**：如Python、Java等。
> 2. **汇编语言和指令集架构层**：这里就是ISA所在的层次。
> 3. **微架构或实现层**：这是具体硬件的实现细节，比如Intel的Core、Pentium等或AMD的Ryzen系列。
>
> ISA定义了软件与硬件之间的接口，使得软件开发者可以不必关心底层硬件的具体实现细节，而只需要关心指令集来编写程序。

- Instruction Set Architecture (ISA) (指令集架构)
  - An abstraction on the interface between the hardware and the low-level software
    - Software: To be translated to the instruction set
    - Hardware: Implementing the instruction set
  - ISA Allows computer designers to talk about functions independently from the hardware that performs them
    - > 允许计算机设计师在不考虑特定硬件实现的情况下，讨论和设计计算机功能。以指令集架构为例，设计师可以定义一个指令来完成特定的操作，例如加法，而不需要指定这个加法是如何在硬件上实现的。这样，ISA就充当了软件和硬件之间的桥梁，为高级编程语言提供了一个稳定的接口。
  - This abstraction allows many implementations of varying cost and performance to run identical software

> - 由于存在上述的抽象，同一个指令集架构可以有多种不同的硬件实现。这些实现可能在成本和性能上有所不同。例如，高性能的服务器处理器和低功耗的移动设备处理器可能都遵循相同的ISA，但它们在微架构（即具体的硬件实现）上会有所不同。尽管如此，由于它们共享相同的ISA，它们仍然可以运行相同的软件。这种抽象确保了软件的兼容性和长期稳定性。

3.2 Machine Code vs. Assembly Language

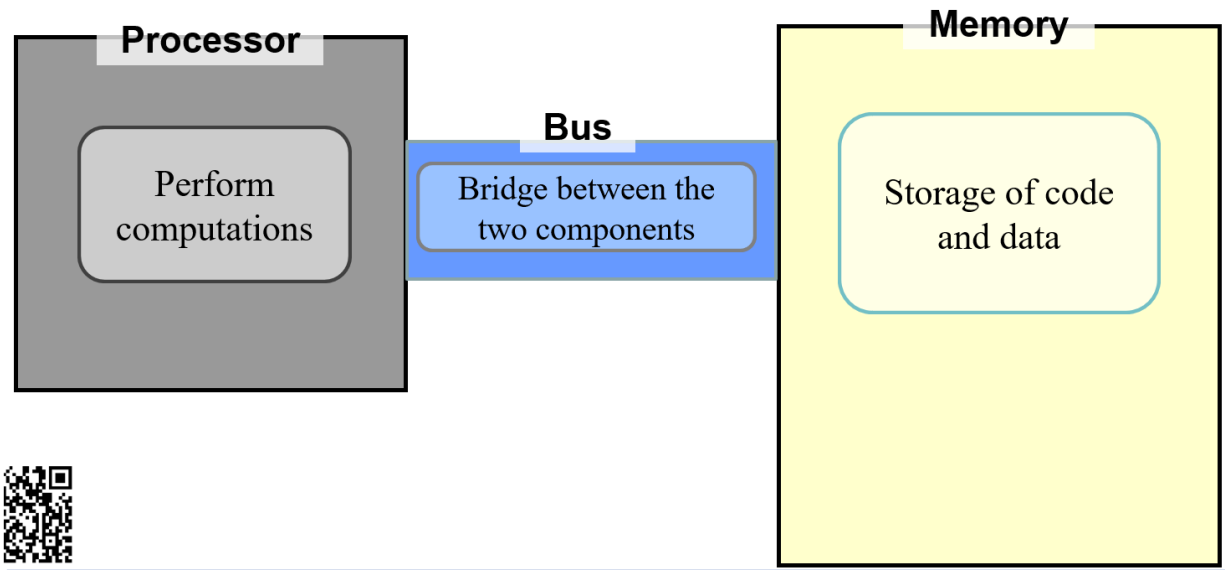
- Machine Code
  - Instructions are represented in **Binary**
  - Hard and tedious for programmer
- Assembly Language
  - Symbolic version of machine code
  - Human readable
    - **1000110010100000** in binary and `add A,B` in assembly language
  - **Assembler** translates from assembly language to machine code
  - Assembly can provide '**pseudo-instructions**' as **syntactic sugar**
    - "伪指令" (pseudo-instructions)
      - 在汇编语言中，伪指令不是实际的机器语言指令，但它们在汇编器中有特定的含义。汇编器在处理伪指令时会将它们转换为一个或多个实际的机器指令，或执行特定的操作。例如，某些伪指令可能用于数据分配或指定一个内存地址。

      "语法糖" (syntactic sugar)
      - 语法糖是指在编程语言中，为了使代码更易读、更易写而添加的某种语法。这种语法并没有为语言增加新的功能，但它为编程师提供了一种更加方便、更加直观的方式来表示某个操作或结构。

      汇编语言提供的伪指令可以被视为一种语法糖，因为它们为程序员提供了一种更简单、更直观的方式来编写汇编代码，尽管这些伪指令在最终转换为机器代码时可能会被替换为实际的指令或执行一系列操作。简言之，伪指令使得汇编代码更易读和更易写，但它们并不直接对应于实际的硬件指令。
  - When considering performance, only read instructions are counted, pseudo-instructions are not counted

3.3 Walkthrough

The components

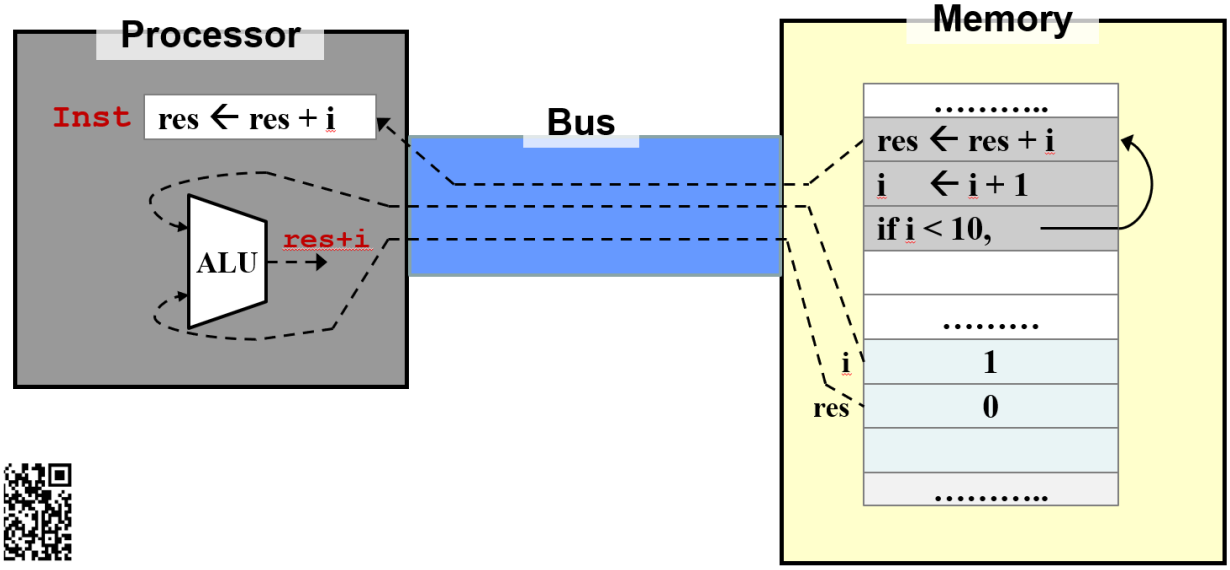


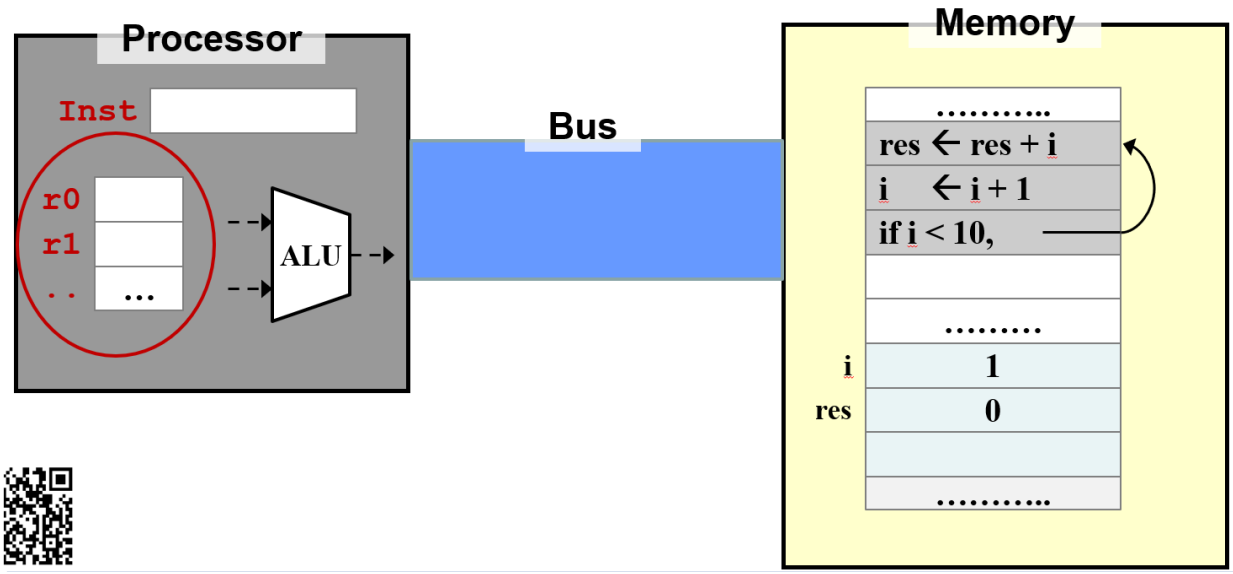
- Assume a simple computing-storage platform, including a processor, bus and a memory
  - Processor processing instructions
  - Bus transmit the data between memory and processor
  - Memory store the instructions and temp data

The code in Action

```
for (i=1; i<10; i++) {
    res = res + i;
}
```

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

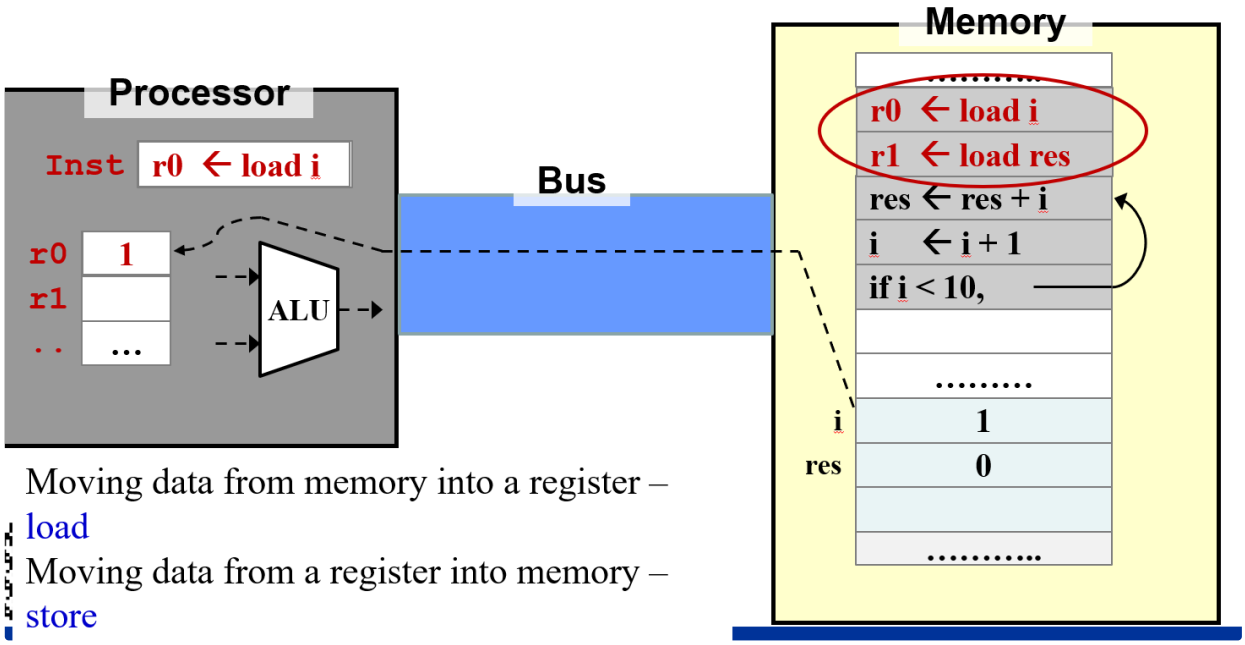




- The instructions and variables are stored in the **memory**
- The variable, `i` and `res` has been transmitted to **processor** through **bus**
  - The data of two variables and the instruction has been stored in **CPU register**, in order to get faster speed
  - > - The register (寄存器) is inside in the CPU, which is using to store the processing data and instruction
    > - The register can provide data and instruction quickly to the ALU
    > - The size of register determines the bit of the CPU, like 32 bit or 64 bit CPU
    > - THE REGISTER IS NOT L1, L2, and L3 CAHCE
    > - L1, L2 and L3 cache is bigger, but slower than register. It provides a buffer between the memory and the processor register, mainly to decrease the distance between these two, from the physically abstraction.

Memory Instructions

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

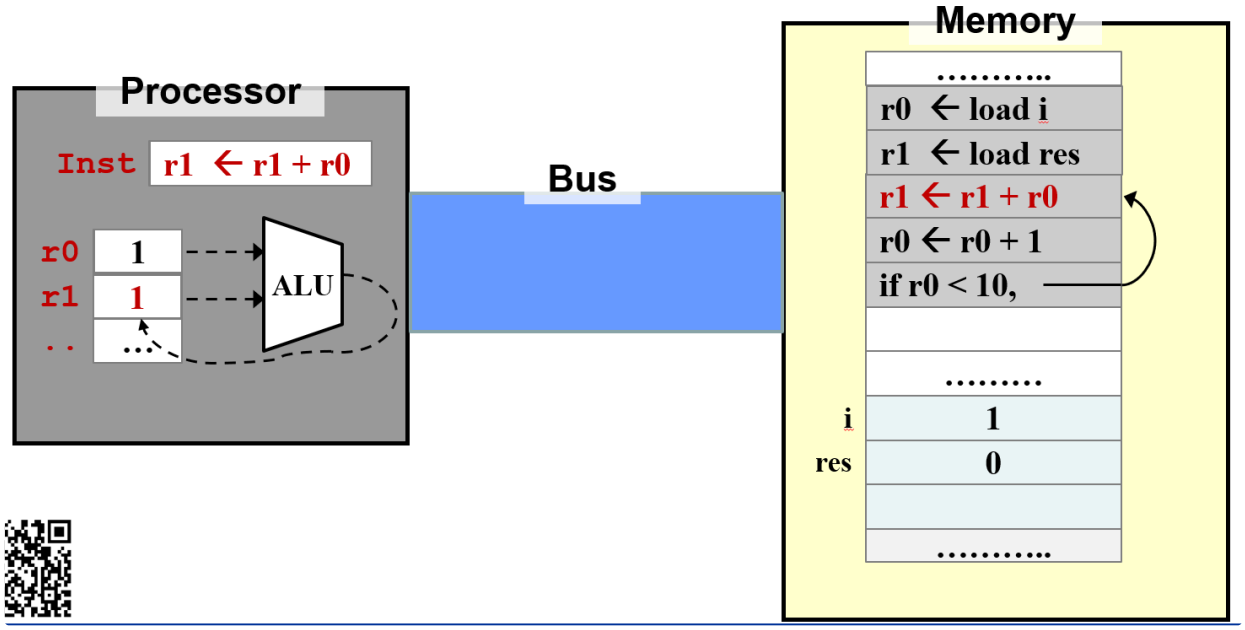


Moving data from memory into a register – load

Moving data from a register into memory – store

- The machine needs instruction to move data from memory to the register
  - `r0 <- load i` is in the memory
  - This instruction has been transmitted to the processor through bus
  - Then the register `0` is **loaded** the data of variable `i`
- Then the same process for variable `res`

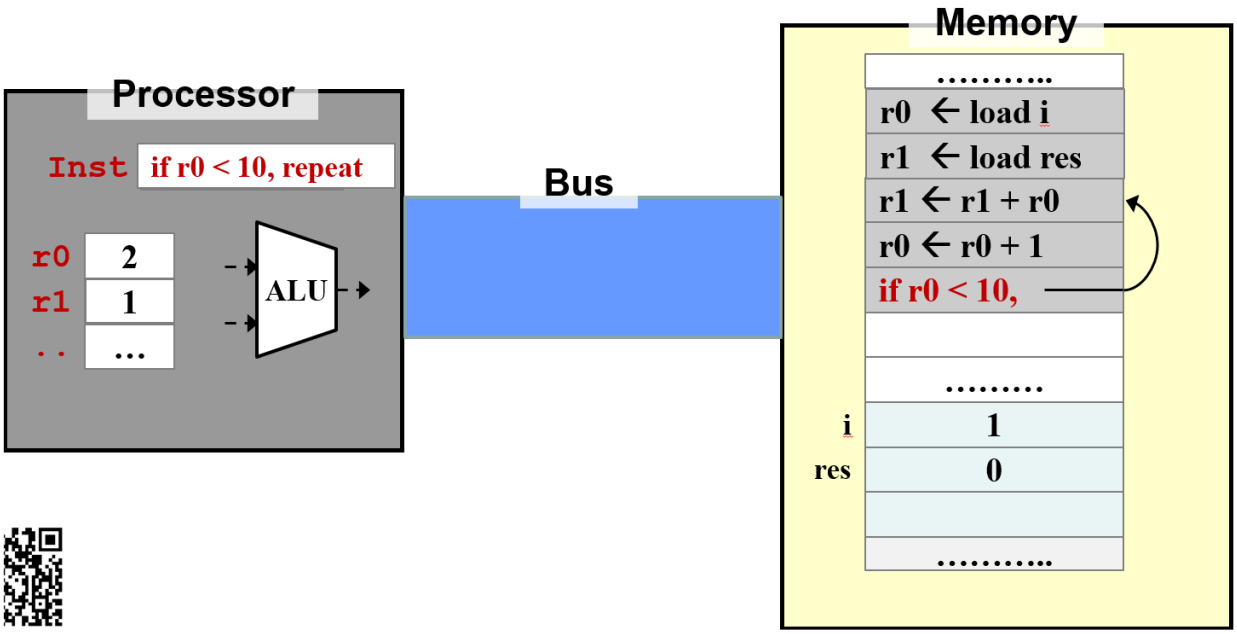
Reg-to-Reg Arithmetic



- After moving all needed variables into the register, the ALU can load all need variables from the register, but not memory. Which is much faster

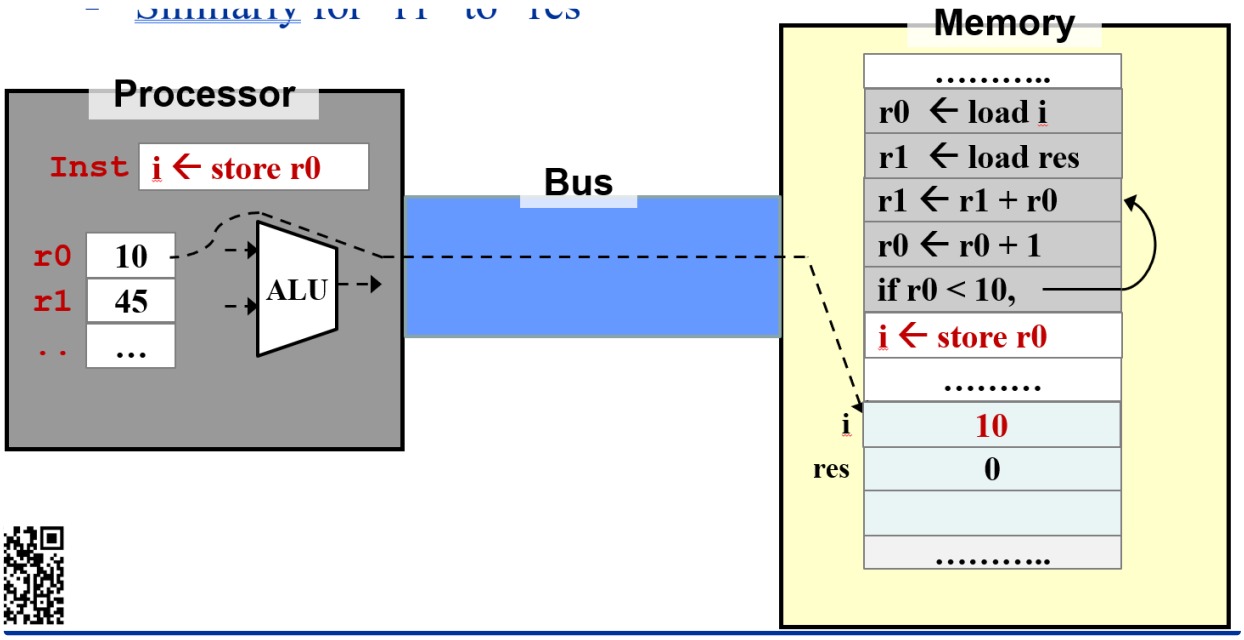
- Sometimes the arithmetic operation uses a constant value instead of register value
  - `r0 <- r0 + 1`, `1` is the constant value
  - > 常数被称为"立即数"（Immediate Value）。立即数是直接编码在机器指令中的常数值。
    >
    > 所以，当ALU要执行加法操作时，它不需要从寄存器或内存中加载立即数，因为这个值已经直接包含在指令中了。这意味着CPU可以在执行指令的同时，直接从指令本身获取这个常数值，并在ALU中与寄存器中的值进行计算。

Execution Sequence (Loop)



- What happened when the machine want to process the loop?
  - ```
    res <- res + i
    i <- i + 1
    if i < 10, repeat
    ```
- We need instructions to change the **control flow** based on condition:
  - Repetition (loop) and Selection (if-else) can both support
- Since the condition succeeded, execution will repeat from the indicated position
  - The execution will continue sequentially
  - Until we see another control flow instruction
    ```

Memory Instructions (Finish Computing, output the result)



- Finally, move the computed value out from the register, store it into the memory

Summary

- 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

3.4 General Purpose Registers

General Purpose Registers（GPR，通用寄存器）是中央处理器（CPU）内部的一组寄存器，这些寄存器不是为某个特定的任务或操作而设计，而是为了存储临时数据或在指令执行过程中用作操作数。通常，汇编语言编程或机器语言编程中的指令可以直接访问和操作这些寄存器。

GPR的特点和用途

1. **多功能**：与专用寄存器（如浮点寄存器或状态寄存器）相比，通用寄存器可以用于多种任务，如算术运算、数据移动、逻辑操作等。
2. **速度**：访问通用寄存器的速度非常快，因为它们是CPU内部的存储单元。这使得它们成为临时存储操作数和结果的理想选择。
3. **数量**：现代处理器通常具有多个GPR。例如，x86架构提供了EAX、EBX、ECX、EDX等寄存器（在64位模式下，这些寄存器分别被称为RAX、RBX、RCX、RDX）；而ARM架构提供了R0到R15的寄存器。
4. **扩展性**：随着处理器架构的发展，GPR的数量和大小可能会发生变化。例如，早期的x86处理器使用16位的AX、BX、CX和DX作为其GPR，而现代的x86-64处理器则使用64位的RAX、RBX、RCX和RDX。

5. **任务**：尽管这些寄存器被称为"通用"，但在某些指令或情境下，它们可能有特定的用途或约定。例如，在某些架构中，某些GPR可能首选或专用于函数调用的返回值或作为参数传递。

- Not all CPU register are GPR
  - CPU寄存器是一个广泛的类别，涵盖了处理器内的所有寄存器。这包括GPR，但也包括其他专用或特定功能的寄存器。
  - GPR是处理器中的寄存器，可以用于多种通用计算和数据传输任务。它们并没有为特定功能（如浮点运算）专门设计。
- Data are transferred from memory to registers for faster processing
- A typical architecture has 16 to 32 registers
- GPR has no data type

- There are **32 registers** in MIPS assembly language
  - Can be referred by a number ($0, $1,…, $31) OR
  - referred by name ($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 (register 26-27) are reserved for the operation system

3.5 MIPS Assembly Language

MIPS assembly language 是 MIPS 架构的汇编语言。MIPS (Microprocessor without Interlocked Pipeline Stages) 是一种基于RISC (Reduced Instruction Set Computer) 原则的微处理器架构，旨在简化指令集以提高性能。以下是有关MIPS汇编语言的一些关键点：

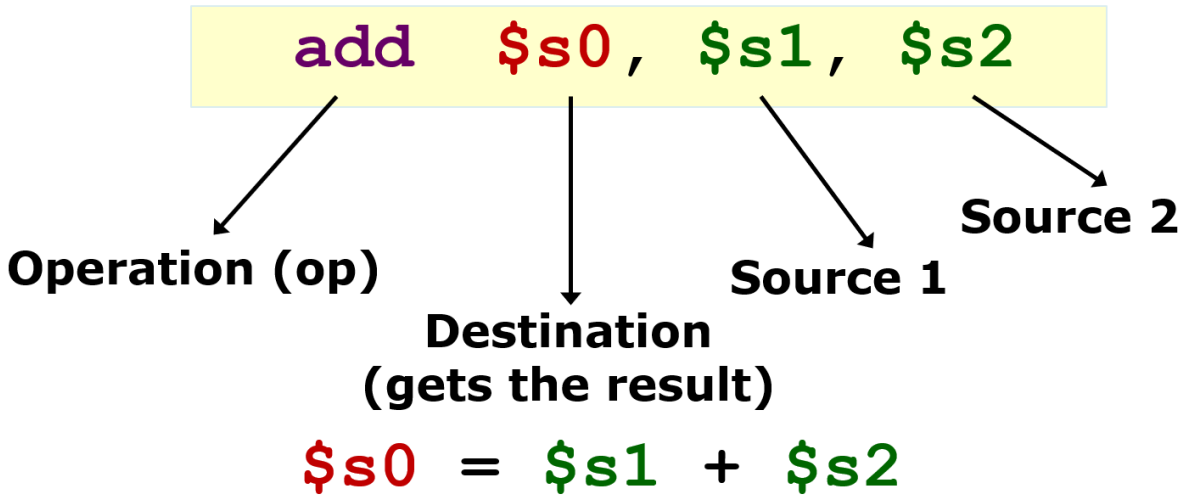
1. **指令集**：MIPS 指令集被设计得相对简单，并与其硬件管道设计紧密相连。这使得MIPS能高效地执行指令，同时简化了硬件实现。
2. **寄存器**：MIPS架构具有32个通用寄存器，标记为 `$0` 到 `$31` 。其中， `$0` 寄存器总是存储值0，其他寄存器用于不同的目的。例如， `$sp` 是堆栈指针， `$ra` 是返回地址寄存器。
3. **指令格式**：MIPS指令有几种格式，最常见的是R型、I型和J型。不同的格式支持不同的操作，例如算术运算、数据传输和跳转。
4. **指令示例**：以下是一些常见的MIPS汇编指令示例：

> - `add $t0, $t1, $t2` ：将 `$t1` 和 `$t2` 中的值加在一起，并将结果存储在 `$t0` 中。
> - `lw $t0, 4($sp)` ：从堆栈指针 `$sp` 加上偏移量 4 的位置加载一个字（word）到 `$t0` 。
> - `beq $t0, $t1, label` ：如果 `$t0` 和 `$t1` 的值相等，则跳转到 `label` 。

- Each instruction executes a simple command
  - Usually has a counterpart in high level programming language like C/C++, Java
- 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
```

General Instruction Syntax



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

Arithmetic Operation: Addition

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

- The value `a` , `b` and `c` are loaded into the CPU register `\$s0` , `\$s1` and `\$s2`
  - This procession is called **variable mapping**

> MIPS arithmetic operations are mainly **reg-to-reg**

Arithmetic Operation: Subtraction

C Statement	MIPS Assembly Code

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

- Similar variable mapping processing than addition
  - The position of `\$s1` and `\$s2` is 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 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

- A single MIPS instruction can only handle at most two source operands.
  - In this case, we should break this expression into two separate expressions, one is addition and another is subtraction.

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
```

Constant/Immediate Operands

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

- Immediate value are also called: "constant value"
- The instruction is "Add immediate" (`addi`), differ from the default addition `add`

Register zero ($0)

C Statement	MIPS Assembly Code

`f = g;`	`add $s0, $s1, $zero`
	
	`$s0` → variable `f`
	`$s1` → variable `g`

- The number zero (0) is **constantly** assigned at register zero (`\$0` or `\$zero`)
- The above assignment is equivalent to the pseudo instruction (move)
  - `add \$s0, \$s1, \$zero`
  - `move \$s0, \$s1`

3.6 Logical Operations

Overview

- Arithmetic instructions view the content of a register as a single quantity (singed or unsigned integer)
- The logical operations allows the ALU to view register as 32 raw bits rather than a single 32-bit number

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, xori

- Truth table of logical operations

AND

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

OR

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

NOR

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

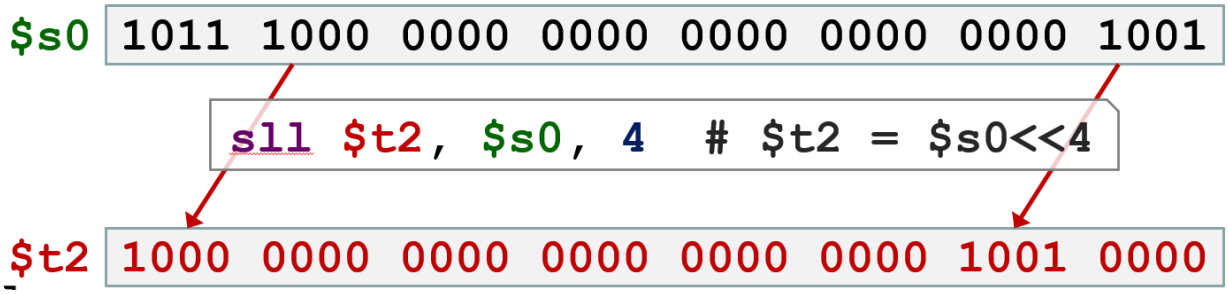
XOR

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

Shifting

Opcode: srl (shift right logical)

Shifts right and fills emptied positions with zeroes.



- Left shift: `sll`
- The above code let the value in register `\$s0` **left shift** 4 bits, then store into the register `\$t2`

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

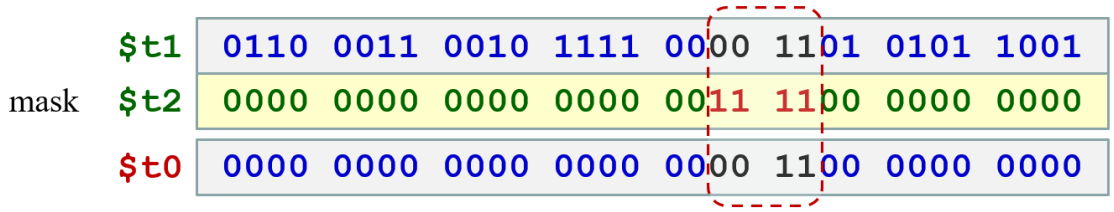
- Right shift: `srl`

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`



- Bitwise AND: `and`
  - Do AND operation on the register `\$t1` and `\$t2` **bit by bit**, then store the result to register `\$t0`
  - It can be used for masking operation

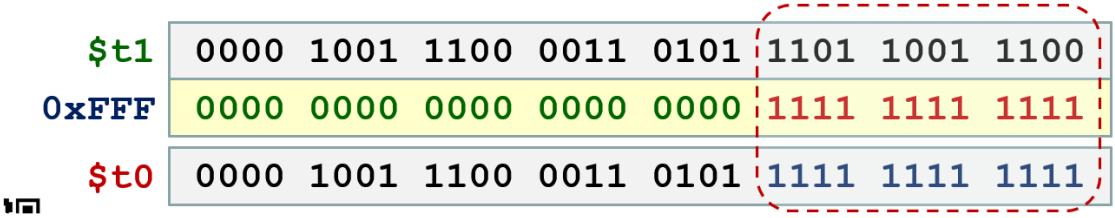
Bitwise OR

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`

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



- Bitwise OR: `or`
- Similar usage with the Bitwise AND operator
  - Also can be used to mask

Bitwise NOR

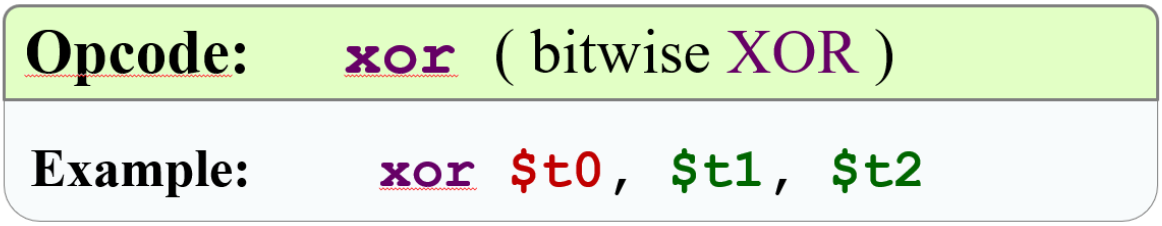
- Strange fact 1:
  - There is no **NOT** instruction in MIPS to toggle the bits (1 → 0, 0 → 1)
  - However, a **NOR** instruction is provided:

Opcode: `nor` (bitwise NOR)

Example: `nor $t0, $t1, $t2`

- Bitwise NOR: `nor`
- Similar usage with the Bitwise AND, OR operator
- There is no NOT instructions in MIPS to toggle bits 1->0, 0->1
  - However, the `nor` instruction can achieve the NOT operation
    - `nor \$t0, \$t0, \$zero`
    - This instruction turn all 0 to 1, and all 1 to 0 in register `\$t0`
- There DO NOT exist `nori` instruction

Bitwise XOR



- Bitwise XOR: `xor`
- To get `not` operation through `xor`:
  - `xor \$t0, \$t0, \$t2`
- There DO exist `xori`

3.7 Large Constant: Case Study