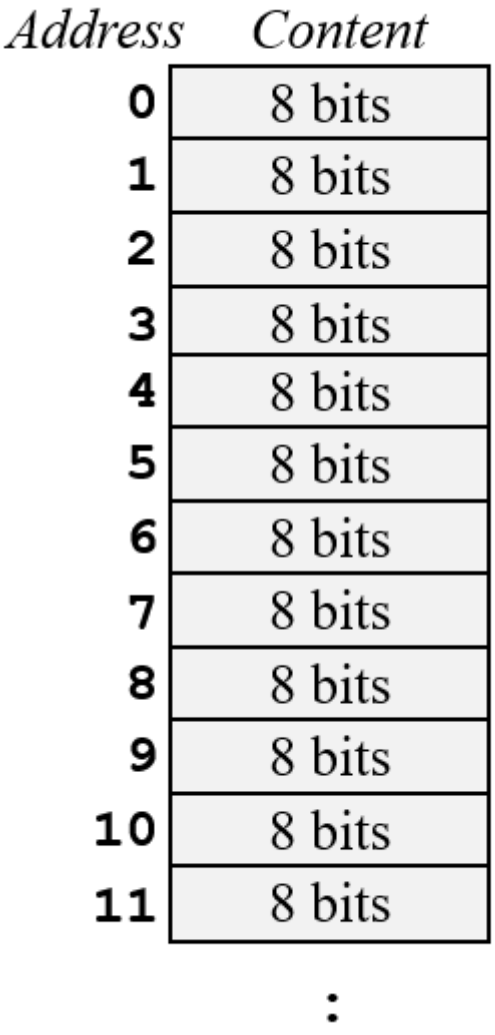
4 - MIPS II

4.1 Memory Organization

- The main memory can be viewed as a large, single-dimension array of memory locations
- Each location of the memory has an address, which is an index into the array
  - Given a `k`-bit address, the address space is of size 2^k
- The memory map on the below contains one byte (8 bits) in every location/address.
  - This is called **byte addressing**



Byte Addressing 字节寻址
Byte addressing（字节寻址）是指在计算机内存中，每个字节都有其独特的地址。这意味着，即使一个数据项（例如32位整数）需要多个字节来存储，我们仍然可以分别访问这些字节。

为了更加清晰地说明，我们可以拿一个32位整数来做例子：

假设我们有一个32位的整数，它需要4个字节来存储（因为32位等于4字节）。在byte addressing系统中，这4个字节会被存放在连续的内存地址中。假设该整数的第一个字节存储在地址0x1000处，那么：

- 第一个字节的地址是：0x1000

- 第二个字节的地址是：0x1001
- 第三个字节的地址是：0x1002
- 第四个字节的地址是：0x1003

这就是byte addressing的核心思想，即每个字节在内存中都有唯一的地址。这种方式使得硬件和软件可以非常灵活地访问内存，但与此同时，也需要在内存管理方面投入更多的精力，以确保有效地使用这种精细级别的寻址机制。

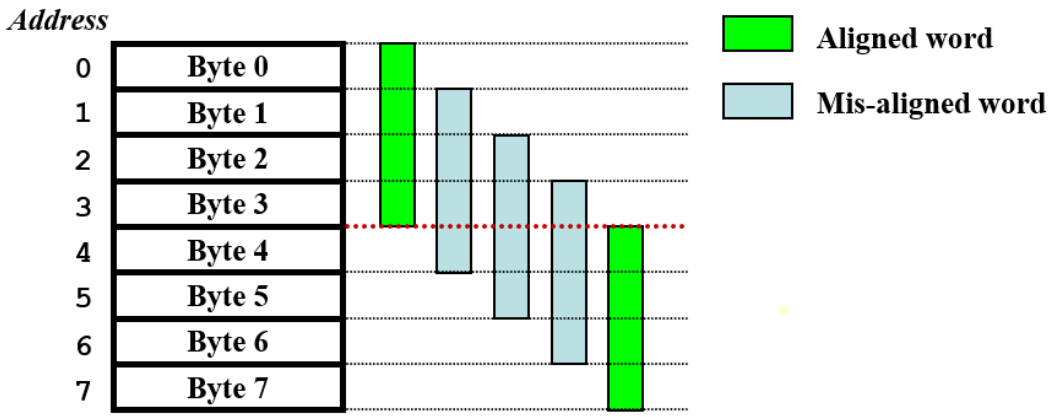
4.1.1 Memory: Transfer Unit

- Using distinct memory address, we can access:
  - a single byte (byte addressable) or
  - a single word (word addressable)
- Word is:
  - Usually 2^n bytes
  - The common unit of transfer between processor and memory
  - Also commonly coincide with the register size, the integer size and instruction size in most architecture

Word的大小通常与寄存器大小相同，例如32位架构中，通常由32位的寄存器（4字节），一个word就是32位

4.1.2 Memory: Word Alignment

- **Word alignment**:
  - Words are aligned in memory if they begin at a byte address that is a multiple of the number of bytes in the word
- Example: If a word consists of 4 bytes, then:



"Word Alignment"（或简称 "Alignment"）是计算机存储和内存管理中的一个概念，它指的是数据项在内存中的开始地址应该是其大小（通常是数据项大小或特定架构的word大小）的某个倍数。

为什么需要对齐？

1. **性能**：在许多架构上，访问对齐的数据比非对齐的数据要快。当数据对齐时，数据可能完全位于一个或多个缓存行内，从而减少了需要访问的缓存行数量。

2. **硬件要求**：一些处理器不支持非对齐的数据访问，或者在尝试这样做时可能导致性能损失或异常。

以32位系统为例，其中word的大小为4字节（32位）：

- 如果一个32位的整数地址为0x1004或0x1008，那么这个整数是对齐的，因为这些地址都是4的倍数。
- 但是，如果这个32位的整数的地址为0x1005或0x1006，那么它就不是对齐的，因为这些地址不是4的倍数。

为了确保对齐，编译器和内存分配器通常会自动处理数据对齐的问题，为变量分配适当对齐的地址。但在低级编程或嵌入式系统开发中，程序员可能需要更加关注对齐的问题，因为它可能会影响性能或正确性。

如果是64位系统，则应该是8的倍数。

4.2 MIPS Memory Instructions

- MIPS is a load-store register architecture
  - 32 registers, each 32-bit (4 bytes) long
  - Each word contains 32-bit (4 bytes)
  - Memory addresses are 32-bit long

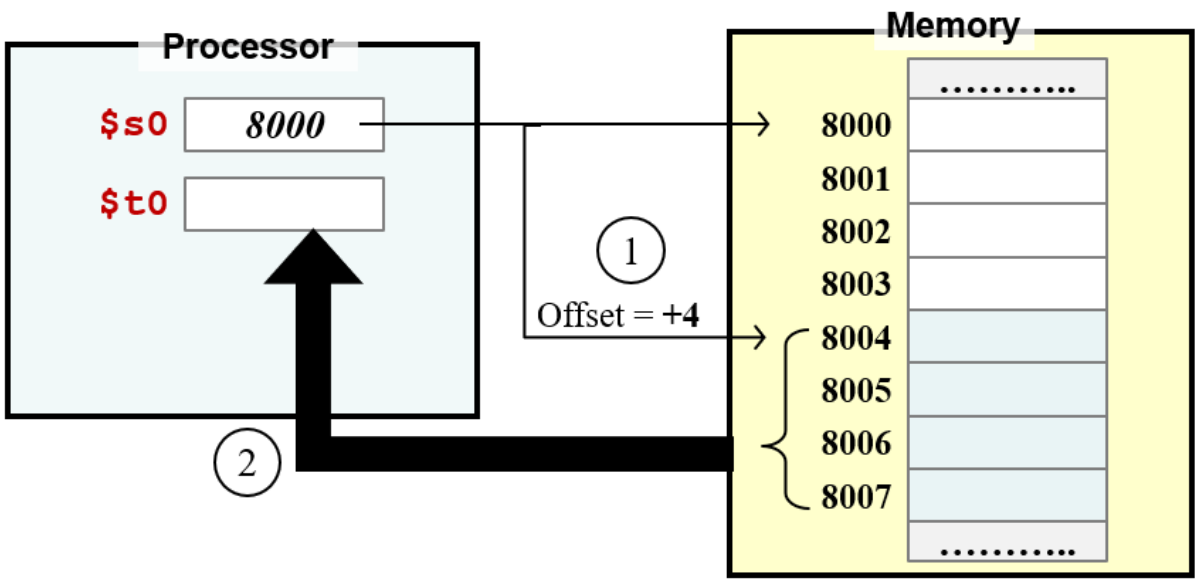
Name	Examples	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Fast processor storage for data. In MIPS, data must be in registers to perform arithmetic.
2^{30} memory words	Mem[0] , Mem[1] , …, Mem[4294967292]	Accessed only by data transfer instructions. MIPS uses **byte addresses**, so consecutive words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls

- 32 registers:
  1. $0 或 $zero ：这个寄存器始终包含值0，任何尝试向其写入的操作都会被忽略。
  2. $1 或 $at ：为汇编器预留的临时寄存器。
  3. $2-$3 或 $v0-$v1 ：用于返回函数值的寄存器。
  4. $4-$7 或 $a0-$a3 ：用于传递函数参数的寄存器。
  5. $8-$15 、 $24-$25 或 $t0-$t7 和 $t8-$t9 ：临时寄存器，函数调用不会保存它们。
  6. $16-$23 或 $s0-$s7 ：保存的寄存器，函数调用会保存它们。
  7. $26-$27 或 $k0-$k1 ：为操作系统预留的寄存器。
  8. $28 或 $gp ：全局指针。
  9. $29 或 $sp ：堆栈指针。
  10. $30 或 $fp ：帧指针（在某些约定中使用）。
  11. $31 或 $ra ：返回地址。

- memory words
  - 指内存中的数据单元。一个 "word" 在 MIPS 中通常指代一个固定大小的数据块，其大小在传统的 MIPS 架构中为 32 位，或 4 字节。
  - 如果 MIPS 系统有 2^{30} 个 "memory words"，这意味着这个系统有 2^{30} 个独立的 32 位数据块。换句话说，该系统的总内存大小是 2^{30}×32 位，或 2^{30}×4 字节 (4GB)。
- Words and Memory words
  - **Word**: 是指数据的大小。在 32 位 MIPS 架构中，一个 word 是 32 位或 4 字节。
  - **Memory Words**: 是指内存中的 word 单元的数量。如果一个系统有 2^{30} 个 memory words，那么它具有 2^{30} 个独立的 32 位数据单元。

4.2.1 Memory Instruction: Load Word

- `lw $t0, 4($t0)`



- Steps:
  1. Memory Address = `$s0 + 4 = 8000 + 4 = 8004`
  2. Memory word at `Mem[8004]` is loaded into `$t0`

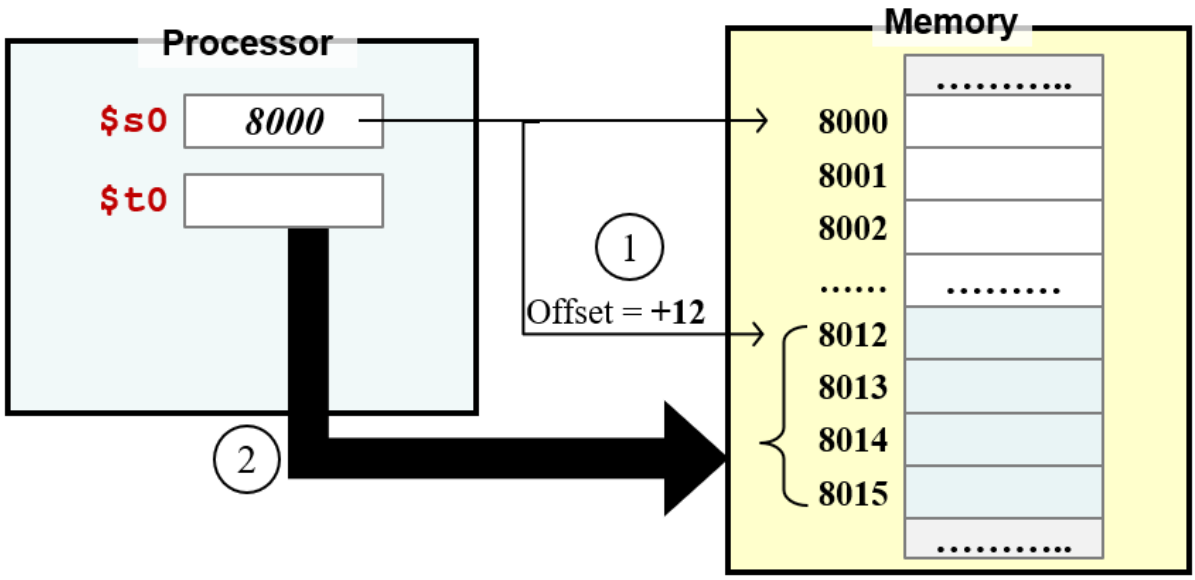
1. `lw` : 这是 "load word" 的指令，意味着从内存中加载一个 32 位的数据块。
2. `$t0` : 这是目标寄存器，指示数据从内存加载到哪个寄存器中。
3. `4($t0)` : 这是源操作数，表示内存的地址。这个地址是通过取 `$t0` 寄存器中的值，并加上偏移量 `4` 来计算的。这里的偏移量是以字节为单位的，因此 `4` 实际上是 4 字节的偏移量。

所以，整体上，这条指令的意思是: 从地址为 `$t0 + 4` 的位置加载一个 word (32 位的数据块) 到 `$t0` 寄存器中。

例如，假设 `$t0` 中原来的值是 `0x8000` ，那么这条指令会从内存地址 `0x8004` 加载一个 word 到 `$t0` 寄存器中。

4.2.2 Memory Instruction: Store Word

- `sw $t0, 12($s0)`



- Steps:
  1. Memory Address = `$t0 + 12 = 8000 + 12 = 8012`
  2. Content of `$t0` is stored into word at `Mem[8012]`

1. `sw`：这是 "store word" 的指令，意味着将一个 32 位的数据块存储到内存中。
2. `$t0`：这是源寄存器，它表示要存储到内存中的数据来源于哪个寄存器。
3. `12($s0)`：这是目标操作数，表示内存的地址。这个地址是通过取 `$s0` 寄存器中的值，并加上偏移量 `12` 来计算的。这里的偏移量是以字节为单位的，所以 `12` 实际上是 12 字节的偏移量。

因此，整体上，这条指令的意思是：将 `$t0` 寄存器中的 word（32 位的数据块）存储到地址为 `$s0 + 12` 的内存位置。

例如，假设 `$s0` 中的值是 `0x8000`，那么这条指令会将 `$t0` 寄存器中的内容存储到内存地址 `0x8012` 的位置。

4.2.3 Load and Store Instructions

- Only `load` and `store` instructions can access data in memory
- Example: Each array element occupies a word

C Code	MIPS Code

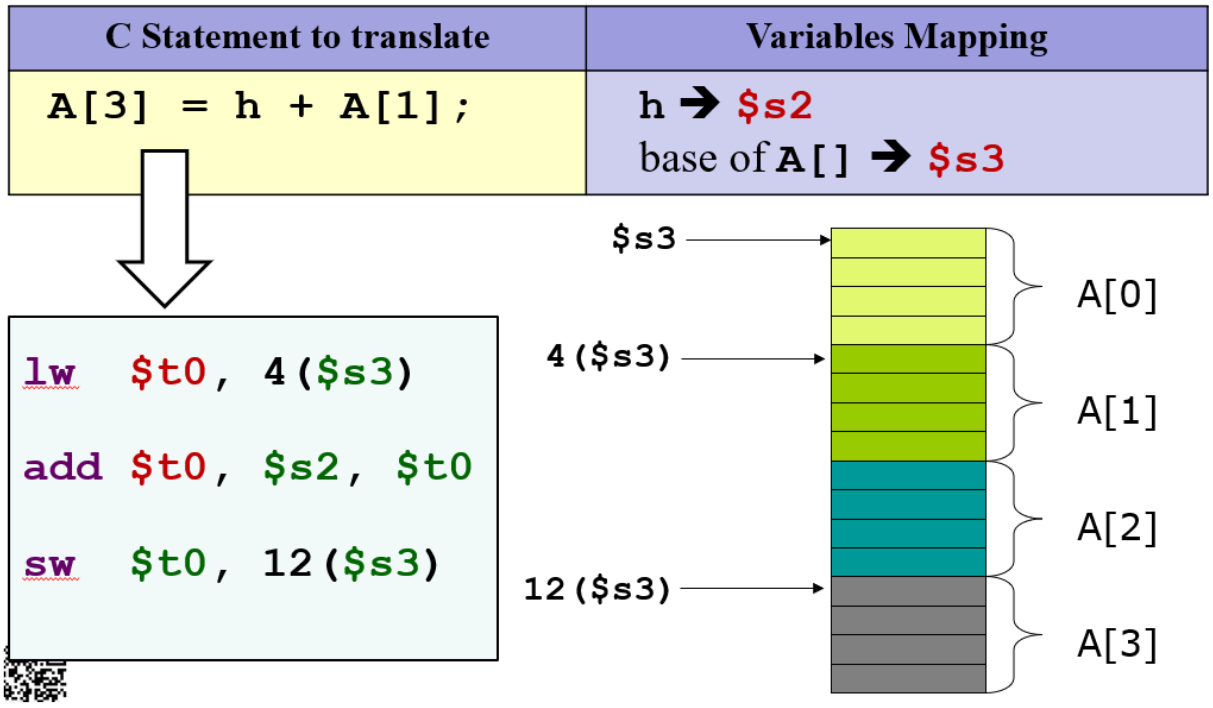
`A[7] = h + A[10]`	`lw $t0, 40($s3)`
	`add $t0, $s2, $t0`
	`sw $t0, 28($s3)`

- Each array element occupies a word(4 bytes)
- `$s3` contains the base address (address of first element, `A[0]`) of array A. Variable `h` is mapped to `$s2`

4.2.4 Memory Instructions: Others

- Other than load word (`lw`) and store word (`sw`), there are other variants, example:
  - load byte (`lb`)
  - store byte (`sb`)
- Similar in format:
  - `lb $t1, 12($s3)`
  - `lb $t2, 13($s3)`
- Similar in working except that one byte, instead of one word, is loaded or stored
  - Note that the offset no longer needs to be a multiple of 4
- MIPS disallows loading/storing unaligned word using `lw` / `sw`
  - Pseudo-Instructions unaligned load word `ulw` and unaligned store word `usw` are provided for this purpose
- Other memory instructions:
  - `lh` and `sh` : load and store halfword
  - `lwl` , `lwr` , `swl` , `swr` : load word left/right, store word left/right

4.2.5 Example: Array



1. **C Statement to translate**: 我们要转换的 C 语句是 `A[3] = h + A[1];` 。这个语句表示要将变量 `h` 与数组 `A` 的第二个元素（索引为1的元素）相加，然后将结果存储在数组 `A` 的第四个元素中（索引为3的元素）。
2. **Variables Mapping**: 这部分为我们提供了 C 语句中变量与 MIPS 寄存器之间的映射关系。即变量 `h` 映射到寄存器 `$s2` ，而数组 `A` 的基地址（第一个元素的地址）映射到寄存器 `$s3` 。
3. **MIPS Instructions**:

    - `lw $t0, 4($s3)`：这条指令从数组 `A` 中加载第二个元素（由于每个元素占 4 字节，所以索引为1的元素的偏移是 4 字节）到临时寄存器 `$t0` 中。
    - `add $t0, $s2, $t0`：这条指令将寄存器 `$s2`（存储变量 `h` 的值）与寄存器 `$t0` 中的值相加，并将结果存储在 `$t0` 中。
    - `sw $t0, 12($s3)`：这条指令将 `$t0` 中的值存储到数组 `A` 的第四个元素（由于每个元素占 4 字节，所以索引为3的元素的偏移是 12 字节）。
4. **Memory Representation**：这部分展示了数组 `A` 在内存中的表示方式。从基地址 `$s3` 开始，每个格子表示数组的一个元素。每个元素都是一个 word，这里假设一个 word 的大小是 4 字节。

4.2.6 Common Questions

Address vs Value

> Registers do NOT have types

- A register can hold any 32-bit number:
    - The number has no implicit data type and is interpreted according to the instruction that use it
- Examples:
    - `add $t2, $t1, $t0`
        - `$t0` and `$t1` should contain data values
    - `lw $t2, 0($t0)`
        - `$t0` should contain a memory address

Byte vs Word

> Consecutive word addresses in machines with byte-addressing do not differ by 1

- Common error:
    - Assume that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes
- For both `lw` and `sw`:
    - The sum of base address and offset must be a multiple of 4 (i.e. to adhere to word boundary)

4.2.7 Example: Swapping Elements

C Statement to translate	Variables Mapping

`swap( int v[], int k )`
`{`
`  int temp;`
`  temp = v[k]`
`  v[k] = v[k+1];`
`  v[k+1] = temp;`
`}`
	k ➔ **$5**
	Base address of **v[]** ➔ **$4**
	temp ➔ **$15**

```
swap:
    sll   $2, $5, 2
    add   $2, $4, $2
    lw    $15, 0($2)
    lw    $16, 4($2)
    sw    $16, 0($2)
    sw    $15, 4($2)
```

Example: k = 3; to swap v[3] with v[4].
Assume base address of v is 2000.

$5 (k) ← 3
$4 (base addr. of v) ← 2000

$2 ← 12
$2 ← 2012

$15 ← content of mem. addr. 2012 (v[3])
$16 ← content of mem. addr. 2016 (v[4])

content of mem. addr. 2012 (v[3]) ← **$16**
content of mem. addr. 2016 (v[4]) ← **$15**

交换数组中两个连续元素的值（C和MIPS）

- 我们要转换的C函数是swap，函数的主要逻辑是交换数组 `v` 中索引为 `k` 和 `k+1` 的两个连续元素。
- **Variables Mapping**：这部分为我们提供了 C 函数中变量与 MIPS 寄存器之间的映射关系。即参数 `k` 映射到寄存器 `$5` ，数组 `v` 的基地址映射到寄存器 `$4` ，局部变量 `temp` 映射到寄存器 `$15` 。
- **Example**：提供了一个具体的例子，即当 `k=3` 时，交换数组 `v` 的第四和第五个元素（ `v[3]` 和 `v[4]` ）。假设数组的基地址是 2000。
- **MIPS Instructions**：
  - `sll $2, $5, 2` ：这条指令是将 `k` （存储在 `$5` 中）乘以 4（因为每个整数大小是4字节），结果保存在 `$2` 。这是为了计算数组中索引为 `k` 的元素的偏移量。
  - `add $2, $4, $2` ：将基地址（存储在 `$4` ）和偏移量（存储在 `$2` ）相加，计算出 `v[k]` 的地址，并将其保存在 `$2` 中。
  - `lw $15, 0($2)` ：加载 `v[k]` 的值到 `$15` 。
  - `lw $16, 4($2)` ：加载 `v[k+1]` 的值到 `$16` 。
  - `sw $16, 0($2)` ：将 `v[k+1]` 的值存储到 `v[k]` 的位置。
  - `sw $15, 4($2)` ：将 `v[k]` 的值存储到 `v[k+1]` 的位置。

4.3 Making Decisions

- Decision make in high-level language:
  - `if` and `goto` statement
  - MIPS decision making instructions are similar to `if` statement with a `goto`
- Decision making instructions
  - Alter the control flow of the program
  - Change the next instruction to be executed
- Two types of decision-making statements in MIPS
  - Conditional (branch)
    ```
    bne $t0, $t1, label
    beq $t0, $t1, label
    ```
  - Unconditional (jump)
    ```
    j label
    ```
- A label is an anchor in the assembly code to indicate point of interest, usually as branch target
  - Labels are NOT instructions

> 1. **Decision make in high-level language**:
>   - 当我们在高级编程语言（如 C、Java 或 Python）中进行决策时，通常使用的是 `if` 语句和 `goto` 语句。
>   - MIPS 的决策制定指令与高级编程语言中的 `if` 语句相似，并经常与 `goto` 语句一起使用，以决定程序的执行流程。
> 2. **Decision making instructions**:
>   - 这些指令用于改变程序的控制流程。
>   - 它们会改变下一条要执行的指令，从而使程序可能跳转到不同的部分执行。

4.3.1 Conditional Branch: `beq` and `bne`

- Processor follows the branch only when the condition is satisfied (True)
- `beq $r1, $r2, L1`
  - Go to statement labeled `L1` if the value in register `$r1` equals the value in register `$r2`
  - `beq` is “branch if equal”
  - C code: `if (a==b) goto L1`
- `bne $r1, $r2, L1`
  - Go to statement labeled `L1` if the value in register `$r1` does not equal the value in register `$r2`
  - `bne` is “branch if not equal”
  - C code: `if (a != b) goto L1`

4.3.2 Unconditional Jump: `j`

- Processor always follows the branch
- `j L1`
  - Jump to label `L1` unconditionally
  - C code: `goto L1`
- Technically equivalent to such statement
  `beq $s0, $s0, L1`

4.3.3 IF statement

C Statement to translate	Variables Mapping
`if (i == j)` `  f = g + h;`	f → $s0 g → $s1 h → $s2 i → $s3 j → $s4

```
      beq $s3, $s4, L1
      j   Exit
L1:   add $s0, $s1, $s2
Exit:
```

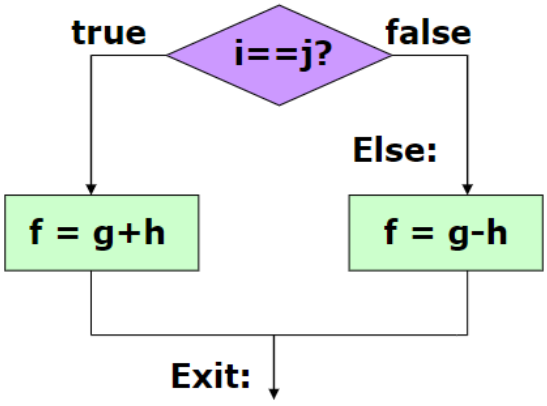
```
      bne $s3, $s4, Exit
      add $s0, $s1, $s2
Exit:
```

> The right one is more efficient

C Statement to translate	Variables Mapping

`if (i == j)`
`  f = g + h;`
`else`
`  f = g - h;`
	f → $s0
	g → $s1
	h → $s2
	i → $s3
	j → $s4

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit:
```



> Re-write to `beq`

```
1  beq $s3, $s4, Else
2  sub $s0, $s1, $s2
3  j Exit
4  Else: add $s0, $s1, $s2
5  Exit:
```

4.3.4 Exercise #1: IF statement

MIPS code to translate into C	Variables Mapping

`beq $s1, $s2, Exit`
`add $s0, $zero, $zero`
`Exit:`
	f ➔ $s0
	i ➔ $s1
	j ➔ $s2

- What is the corresponding high-level statement?

```
if (i != j) {
  f = 0;
}
```

4.4 Loops

- C while-loop:

```
while (j == k)
   i = i + 1;
```

- Rewritten with goto

```
Loop:  if (j != k)
          goto Exit;
       i = i+1;
       goto Loop;

Exit:
```

Key concept:

Any form of loop can be written in assembly with the help of conditional branches and jumps.

4.4.1 Exercise #2: FOR loop

4.4.2 Inequalities

- We have `beq` and `bne`, what about branch-if-less-than?
  - There is no real `blt` instruction in MIPS
- Use `slt` (set on less than) or `slti`

```
slt $t0, $s1, $s2
```

```
if ($s1 < $s2)
    $t0 = 1;
else
    $t0 = 0;
```

- To build a `blt $s1, $s2, L` in instruction

```
slt $t0, $s1, $s2
bne $t0, $zero, L
```

```
if ($t1 < $t2)
    goto L;
```

- This is another example of pseudo-instruction
  - Assembler translates (`blt`) instruction in an assembly program into the equivalent MIPS(two) instructions

4.5 Array and Loop

- Typical example of accessing array elements in a loop:

> Count the number of zeros in an Array A
>
> - A is word array with 40 elements
> - Address of A[] -> $t0, Result -> $t8
>
> **C code:**
>
> ```
> result = 0
> i = 0
> while (i<40) {
>     if (A[i] == 0)
>         result++;
>     i++
> }
> ```

```
addi $t8, $zero, 0              # Assign variable "result" with value 0
addi $t1, $zero, 0              # Assign variable "i" with value 0
addi $t2, $zero, 160            # Assign a temp anoymous variable with
value 160, point to endpoint of array
loop: bge $t1, $t2, end         # Comparing address
      lw  $t3, 0($t1)           # $t3 <- A[i]
      bne $t3, $zero, skip      # if A[i] != 0, then skip
      addi $t8, $t8, 1          # result++
skip: addi $t1, $t1, 4          # move to the next item
      j loop
end:
```

4.6 Exercises