

CS2100

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

COMPUTER ORGANISATION

## Lecture #8

---

# MIPS

## Part II: More Instructions



**NUS**  
National University  
of Singapore

School of  
Computing

# Lecture #8: MIPS Part 2: More Instructions

## 1. Memory Organisation (General)

1.1 Memory: Transfer Unit

1.2 Memory: Word Alignment

## 2. MIPS Memory Instructions

2.1 Memory Instruction: Load Word

2.2 Memory Instruction: Store Word

2.3 Load and Store Instructions

2.4 Memory Instruction: Others

2.5 Example: Array

2.6 Common Questions

2.7 Example: Swapping Elements

# Lecture #8: MIPS Part 2: More Instructions

## 3. Making Decisions

3.1 Conditional Branch: beq and bne

3.2 Unconditional Jump: j

3.3 IF statement

3.4 Exercise #1: IF statement

## 4. Loops

4.1 Exercise #2: FOR loop

4.2 Inequalities

## 5. Array and Loop

## 6. Exercises

# 1. Memory Organisation (General)

- 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 right contains one byte (8 bits) in every location/address.
  - This is called byte addressing

Address	Content
0	8 bits
1	8 bits
2	8 bits
3	8 bits
4	8 bits
5	8 bits
6	8 bits
7	8 bits
8	8 bits
9	8 bits
10	8 bits
11	8 bits
	:

# 1.1 Memory: Transfer Unit

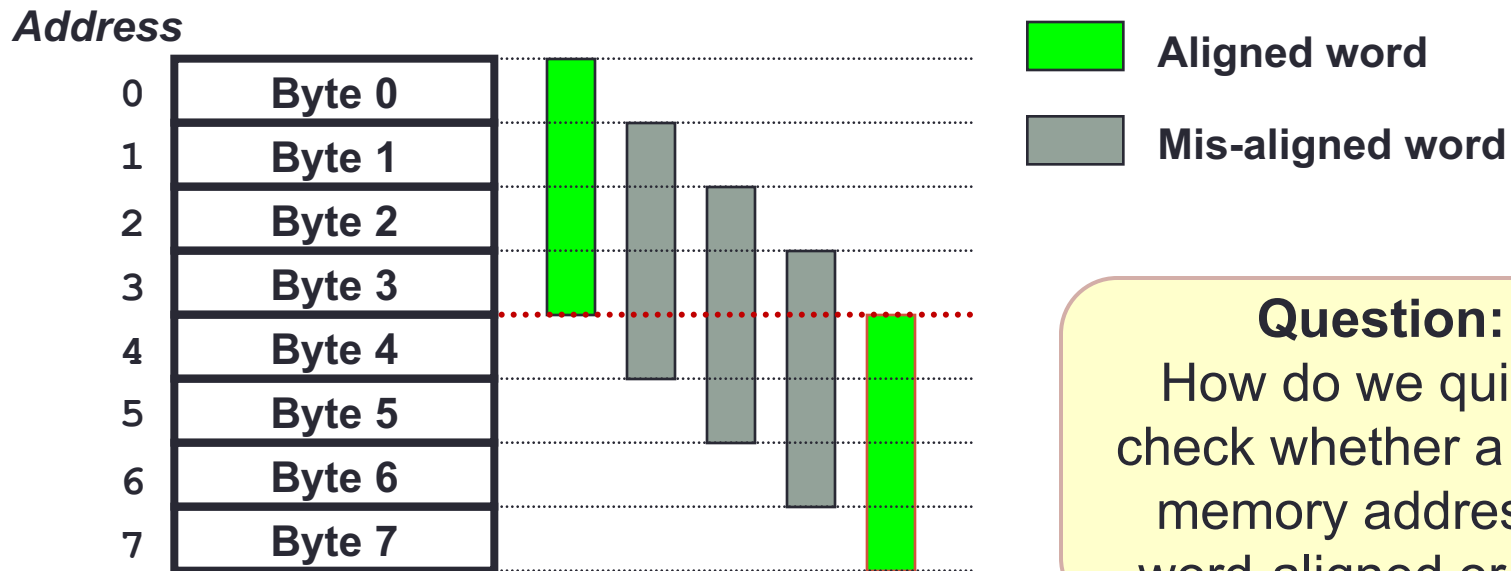
- Using distinct memory address, we can access:
  - a single **byte** (**byte addressable**) or
  - a single **word** (**word addressable**)
- **Word** is:
  - Usually  $2^n$  bytes
  - The common unit of transfer between processor and memory
  - Also commonly coincide with the register size, the integer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

# 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 a word.

Example: If a word consists of 4 bytes, then:



### Question:

How do we quickly check whether a given memory address is word-aligned or not?

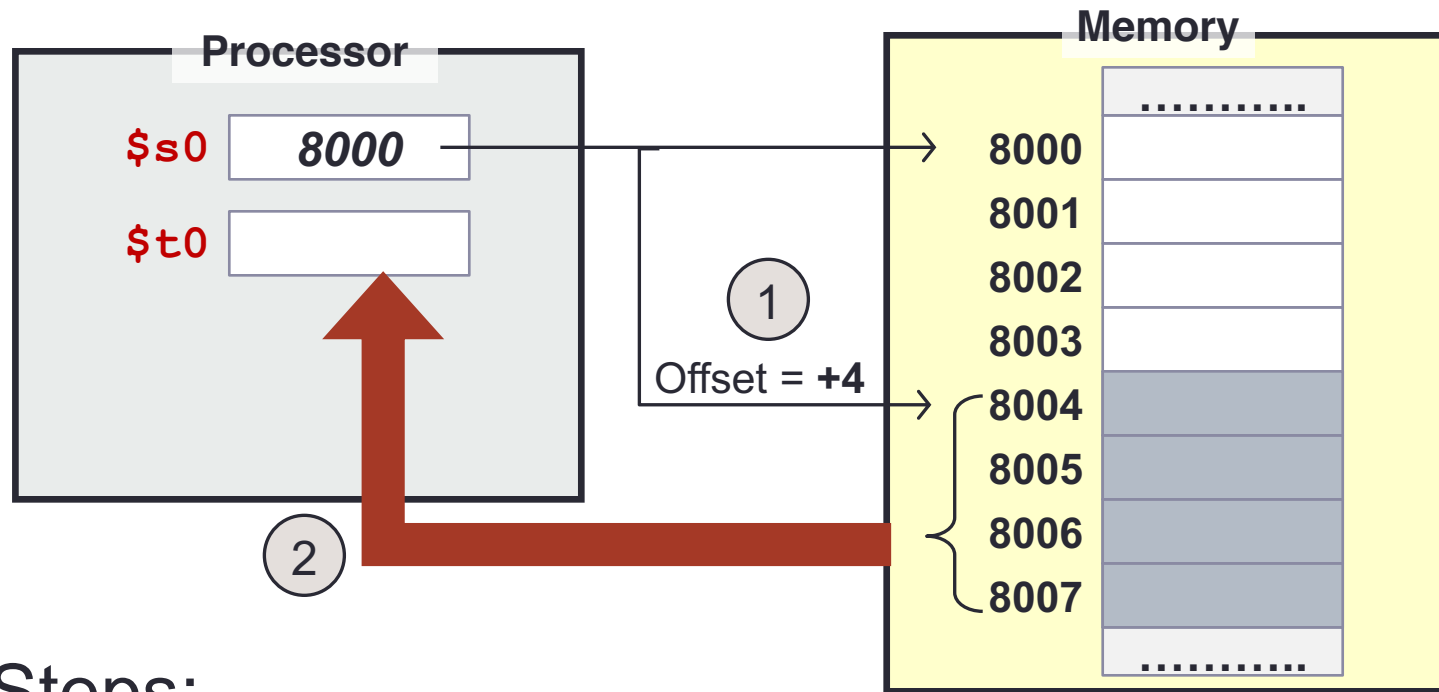
## 2. MIPS Memory Instructions

- MIPS is a load-store register architecture
  - 32 registers, each 32-bit (4-byte) long
  - Each word contains 32 bits (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. <b>In MIPS, data must be in registers to perform arithmetic.</b>
$2^{30}$ memory words	<b>Mem</b> [0] , <b>Mem</b> [4] , ... , <b>Mem</b> [4294967292]	Accessed only by data transfer instructions. <b>MIPS uses byte addresses, so consecutive words differ by 4.</b> Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

## 2.1 Memory Instruction: Load Word

- Example: `lw $t0, 4($s0)`



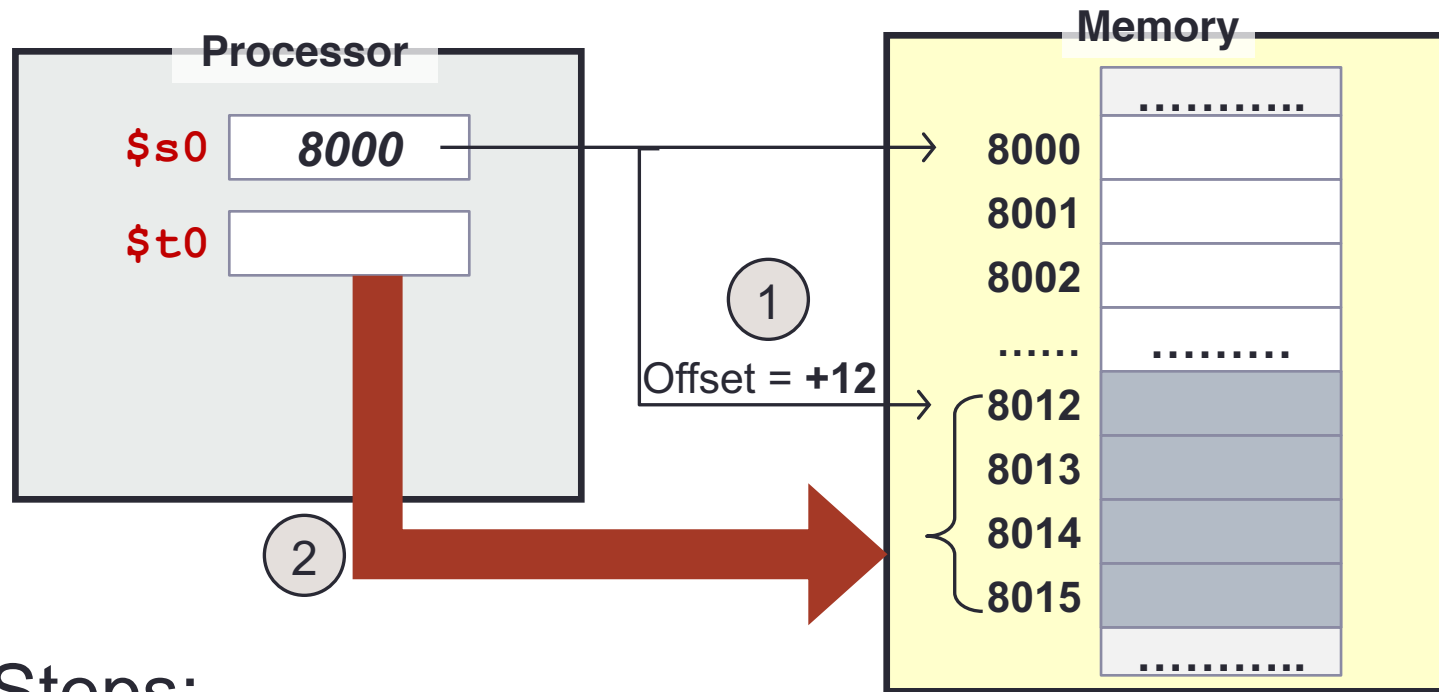
- Steps:

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



## 2.2 Memory Instruction: Store Word

- Example: **sw** **\$t0**, 12 (**\$s0**)



- Steps:

1. Memory Address = **\$s0** + 12 = 8000 + 12 = **8012**
2. Content of **\$t0** is stored into word at **Mem[8012]**

## 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
<code>A[7] = h + A[10];</code>	<code>lw    \$t0, 40(\$s3)</code> <code>add   \$t0, \$s2, \$t0</code> <code>sw    \$t0, 28(\$s3)</code>

- 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**.
- Remember arithmetic operands (for **add**) are registers, not memory!

## 2.4 Memory Instructions: Others (1/2)

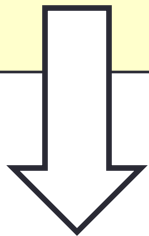
- Other than load word (**lw**) and store word (**sw**), there are other variants, example:
  - load byte (**lb**)
  - store byte (**sb**)
- Similar in format:  
**lb**    **\$t1**, 12 (**\$s3**)  
**sb**    **\$t2**, 13 (**\$s3**)
- Similar in working except that one byte, instead of one word, is loaded or stored
  - Note that the offset no longer needs to be a multiple of 4

## 2.4 Memory Instructions: Others (2/2)

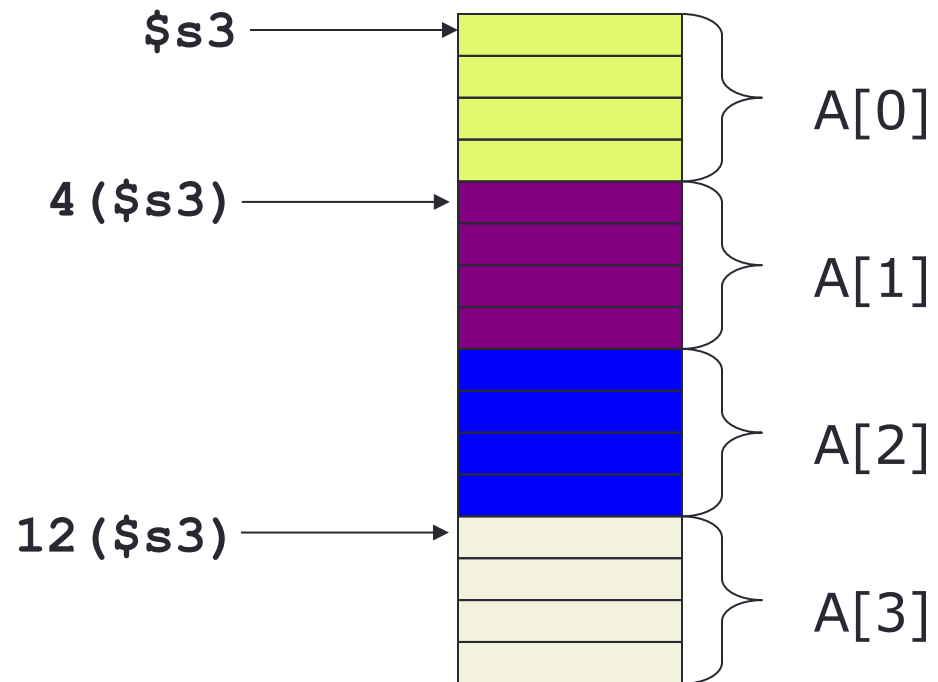
- MIPS disallows loading/storing unaligned word using **lw/sw**:
  - Pseudo-Instructions ***unaligned load word*** (**ulw**) and ***unaligned store word*** (**usw**) are provided for this purpose
- Other memory instructions:
  - **lh** and **sh**: load halfword and store halfword
  - **lwl**, **lwr**, **swl**, **swr**: load word left / right, store word left / right.
  - etc...

## 2.5 Example: Array (assume 4 bytes per element)

C Statement to translate	Variables Mapping
<code>A[3] = h + A[1];</code>	<code>h</code> $\rightarrow$ <code>\$s2</code> base of <code>A[]</code> $\rightarrow$ <code>\$s3</code>



```
lw  $t0, 4($s3)
add $t0, $s2, $t0
sw  $t0, 12($s3)
```



## 2.6 Common Questions: Address vs Value

**Key concept:**

**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 uses it
- Examples:
  - `add $t2, $t1, $t0`
    - ➔ `$t0` and `$t1` should contain data values
  - `lw $t2, 0($t0)`
    - ➔ `$t0` should contain a memory address

## 2.6 Common Questions: **Byte** vs **Word**

### **Important:**

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)

## 2.7 Example: Swapping Elements

C Statement to translate	Variables Mapping
<pre> swap( int v[], int k ) {     int temp;     temp = v[k]     v[k] = v[k+1];     v[k+1] = temp; } </pre>	<p> <math>k \rightarrow \\$5</math>  Base address of <math>v[] \rightarrow \\$4</math>  <math>temp \rightarrow \\$15</math> </p> <p> Example: <math>k = 3</math>; to swap <math>v[3]</math> with <math>v[4]</math>.  Assume base address of <math>v</math> is 2000. </p> <p> <math>\\$5 (k) \leftarrow 3</math>  <math>\\$4 (\text{base addr. of } v) \leftarrow 2000</math> </p> <p> <math>\\$2 \leftarrow 12</math>  <math>\\$2 \leftarrow 2012</math>  <math>\\$15 \leftarrow \text{content of mem. addr. } 2012 (v[3])</math>  <math>\\$16 \leftarrow \text{content of mem. addr. } 2016 (v[4])</math>  <math>\text{content of mem. addr. } 2012 (v[3]) \leftarrow \\$16</math>  <math>\text{content of mem. addr. } 2016 (v[4]) \leftarrow \\$15</math> </p>
<pre> 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) </pre>	

Note: This is simplified and may not be a direct translation of the C code.



# Reading

- **Instructions: Language of the Computer**
  - Read up COD Chapter 2, pages 52-57. (3<sup>rd</sup> edition)
  - Read up COD Section 2.3 (4<sup>th</sup> edition)



### 3. Making Decisions (1/2)

- We cover only sequential execution so far:
  - Instruction is executed in program order
- To perform general computing tasks, we need to:
  - **Make decisions**
  - **Perform iterations** (in later section)
- Decisions making in high-level language:
  - **if** and **goto** statements
  - MIPS decision making instructions are similar to **if** statement with a **goto**
    - **goto** is discouraged in high-level languages but necessary in assembly 😊

## 3. Making Decisions (2/2)

- 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!

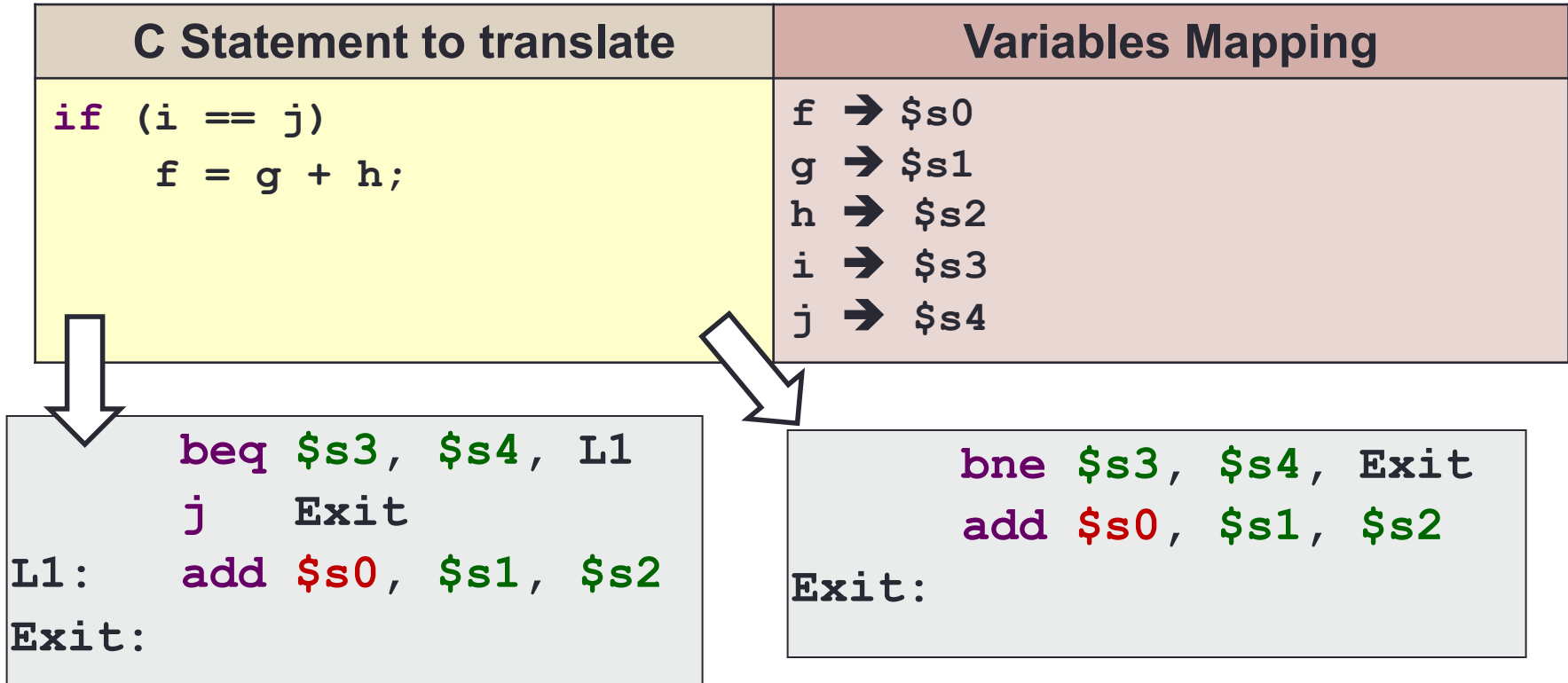
## 3.1 Conditional Branch: **beq** and **bne**

- 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**

## 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

## 3.3 IF statement (1/2)



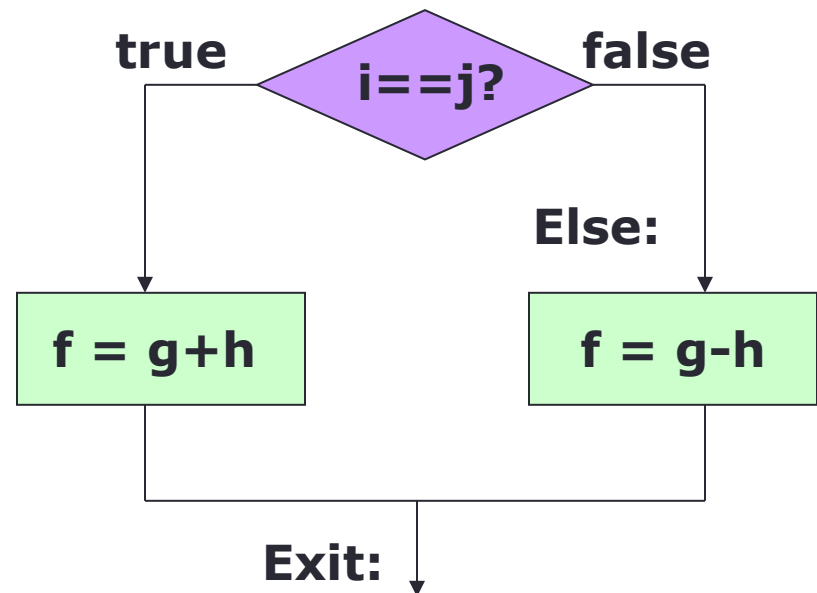
- Two equivalent translations:
  - The one on the right is more efficient
- Common technique: Invert the condition for shorter code

## 3.3 IF statement (2/2)

C Statement to translate	Variables Mapping
<pre>if (i == j)     f = g + h; else     f = g - h;</pre>	<pre>f → \$s0 g → \$s1 h → \$s2 i → \$s3 j → \$s4</pre>

↓

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



- Question: Rewrite with **bneq**?

## 3.4 Exercise #1: IF statement

MIPS code to translate into C	Variables Mapping
<pre>beq \$s1, \$s2, Exit add \$s0, \$zero, \$zero Exit:</pre>	<pre>f → \$s0 i → \$s1 j → \$s2</pre>

- What is the corresponding high-level statement?

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



## 4. Loops (1/2)

- 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. Loops (2/2)

C Statement to translate	Variables Mapping
<pre>Loop:  if (j != k)         goto Exit;         i = i+1;         goto Loop;  Exit:</pre>	<pre>i → \$s3 j → \$s4 k → \$s5</pre>

- What is the corresponding MIPS code?

```
Loop: bne  $s4, $s5, Exit  # if (j!= k) Exit
      addi $s3, $s3, 1
      j    Loop           # repeat loop
Exit:
```

## 4.1 Exercise #2: FOR loop

- Write the following loop statement in MIPS

C Statement to translate	Variables Mapping
<pre>for ( i=0; i&lt;10; i++)     a = a + 5;</pre>	<pre>i → \$s0 a → \$s2</pre>

```
        add    $s0, $zero, $zero  
        addi   $s1, $zero, 10  
Loop:   beq    $s0, $s1, Exit  
        addi   $s2, $s2, 5  
        addi   $s0, $s0, 1  
        j      Loop  
Exit:
```

## 4.2 Inequalities (1/2)

- 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;
```

## 4.2 Inequalities (1/2)

- To build a “**blt** \$s1, \$s2, L” instruction:

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

==

```
if ($s1 < $s2)  
    goto L;
```

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

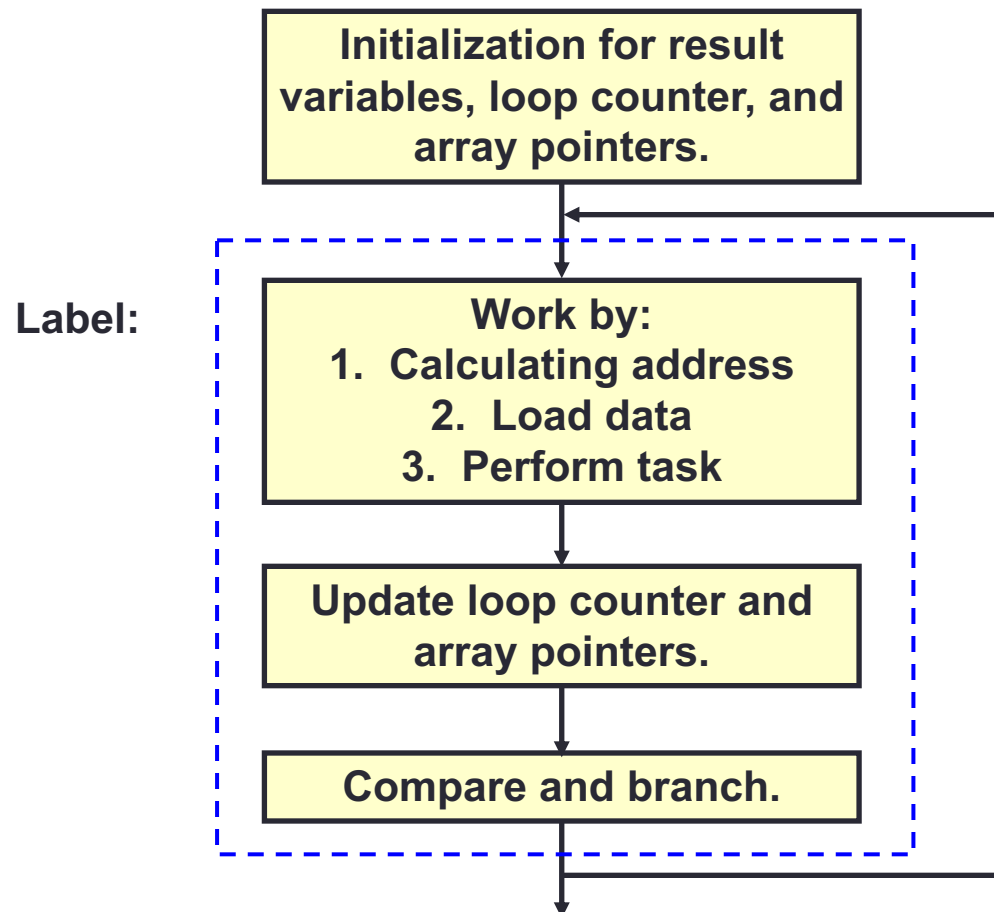
# Reading

- **Instructions: Language of the Computer**
  - Section 2.6 Instructions for Making Decisions. (3<sup>rd</sup> edition)
  - Section 2.7 Instructions for Making Decisions. (4<sup>th</sup> edition)



## 5. Array and Loop

- Typical example of accessing array elements in a loop:



## 5. Array and Loop: Question

Count the number of zeros in an Array **A**

- **A** is word array with 40 elements
- Address of A[] → **\$t0**, Result → **\$t8**

### Simple C Code

```
result = 0;
i = 0;
while ( i < 40 ) {
    if ( A[i] == 0 )
        result++;
    i++;
}
```

- Think about:
  - How to perform the right comparison
  - How to translate A[i] correctly



## 5. Array and Loop: Version 1.0

Address of A[] → \$t0 Result → \$t8 i → \$t1	Comments
<pre>        addi \$t8, \$zero, 0         addi \$t1, \$zero, 0         addi \$t2, \$zero, 40 loop:   bge  \$t1, \$t2, end         sll  \$t3, \$t1, 2         add  \$t4, \$t0, \$t3         lw   \$t5, 0(\$t4)         bne  \$t5, \$zero, skip         addi \$t8, \$t8, 1 skip:   addi \$t1, \$t1, 1         j    loop end:</pre>	<pre># end point # i * 4 # &amp;A[i] # \$t3 ← A[i] # result++ # i++</pre>

## 5. Array and Loop: Version 2.0

Address of A[] → \$t0 Result → \$t8 &A[i] → \$t1	Comments
<pre>        addi \$t8, \$zero, 0         addi \$t1, \$t0, 0         addi \$t2, \$t0, 160 loop:   bge  \$t1, \$t2, end         lw   \$t3, 0(\$t1)         bne  \$t3, \$zero, skip         addi \$t8, \$t8, 1 skip:   addi \$t1, \$t1, 4         j    loop end:</pre>	<pre># addr of current item # &amp;A[40] # comparing address! # \$t3 ← A[i] # result++ # move to next item</pre>

- Use of “pointers” can produce more efficient code!

## 6.1 Exercise #3: Simple Loop

- Given the following MIPS code:

```
        addi $t1, $zero, 10
        add  $t1, $t1, $t1
        addi $t2, $zero, 10
Loop:   addi $t2, $t2, 10
        addi $t1, $t1, -1
        beq  $t1, $zero, Loop
```

- How many instructions are executed? **Answer: (a)**  
(a) 6    (b) 30    (c) 33    (d) 36    (e) None of the above
- What is the final value in **\$t2**? **Answer: (b)**  
(a) 10    (b) 20    (c) 300    (d) 310    (e) None of the above

## 6.2 Exercise #4: Simple Loop II

- Given the following MIPS code:

```
        add    $t0, $zero, $zero
        add    $t1, $t0, $t0
        addi   $t2, $t1, 4
Again:  add    $t1, $t1, $t0
        addi   $t0, $t0, 1
        bne   $t2, $t0, Again
```

- How many instructions are executed? **Answer: (c)**  
(a) 6    (b) 12    (c) 15    (d) 18    (e) None of the above
- What is the final value in **\$t1**? **Answer: (c)**  
(a) 0    (b) 4    (c) 6    (d) 10    (e) None of the above

## 6.3 Exercise #5: Simple Loop III (1/2)

- Given the following MIPS code accessing a word array of elements in memory with the starting address in \$t0.

```
        addi $t1, $t0, 10  
        add  $t2, $zero, $zero  
Loop:   ulw  $t3, 0($t1) # ulw: unaligned lw  
        add  $t2, $t2, $t3  
        addi $t1, $t1, -1  
        bne  $t1, $t0, Loop
```

- How many times is the **bne** instruction executed?  
(a) 1    (b) 3    (c) 9    (d) 10    (e) 11    **Answer: (d)**
- How many times does the **bne** instruction actually branch to the label **Loop**?  
(a) 1    (b) 8    (c) 9    (d) 10    (e) 11    **Answer: (c)**

## 6.3 Exercise #5: Simple Loop III (2/2)

- Given the following MIPS code accessing a word array of elements in memory with the starting address in \$t0.

```
        addi $t1, $t0, 10
        add  $t2, $zero, $zero
Loop:   ulw  $t3, 0($t1) # ulw: unaligned lw
        add  $t2, $t2, $t3
        addi $t1, $t1, -1
        bne  $t1, $t0, Loop
```

iii. How many instructions are executed?

(a) 6    (b) 12    (c) 41    (d) 42    (e) 46    **Answer: (d)**

iv. How many **unique** bytes of data are read from the memory?

(a) 4    (b) 10    (c) 11    (d) 13    (e) 40    **Answer: (d)**

# Summary: Focus of CS2100

- Basic MIPS programming
  - Arithmetic: among registers only
  - Handling of large constants
  - Memory accesses: load/store
  - Control flow: branch and jump
  - Accessing array elements
  - System calls (covered in labs)
- Things we are not going to cover
  - Support for procedures
  - Linkers, loaders, memory layout
  - Stacks, frames, recursion
  - Interrupts and exceptions

# End of File