

# Computing with memory

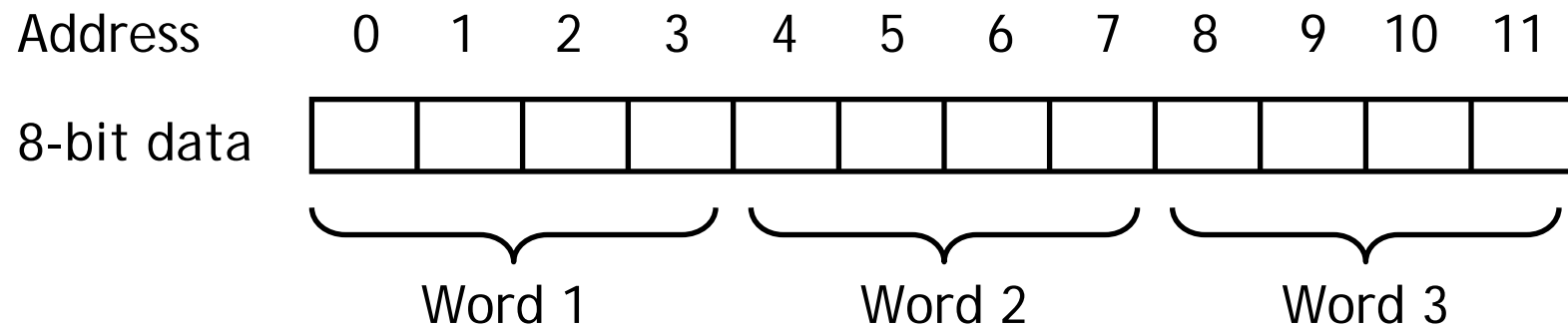
---

- ❑ So, to compute with memory-based data, you must:
  1. Load the data from memory to the register file.
  2. Do the computation, leaving the result in a register.
  3. Store that value back to memory if needed.
- ❑ For example, let's say that you wanted to do the same addition, but the values were in memory. How can we do the following using MIPS assembly language using as few registers as possible?

```
char A[4] = {1, 2, 3, 4};  
int result;  
  
result = A[0] + A[1] + A[2] + A[3];
```

# Memory alignment

- Keep in mind that memory is byte-addressable, so a 32-bit word actually occupies four contiguous locations (bytes) of main memory.



- The MIPS architecture requires words to be **aligned** in memory; 32-bit words must start at an address that is divisible by 4.
  - 0, 4, 8 and 12 are valid **word addresses**.
  - 1, 2, 3, 5, 6, 7, 9, 10 and 11 are *not* valid word addresses.
  - Unaligned memory accesses result in a **bus error**, which you may have unfortunately seen before.
- This restriction has relatively little effect on high-level languages and compilers, but it makes things easier and faster for the processor.

# Exercise

---

□ Can we figure out the code?

```
swap(int v[], int k);      swap:           ; $5=k $4=v[0]
{ int temp;                sll $2, $5, 2; $2←k×4
    temp = v[k]            add $2, $4, $2; $2←v[k]
    v[k] = v[k+1];        lw $15, 0($2) ; $15←v[k]
    v[k+1] = temp;        lw $16, 4($2) ; $16←v[k+1]
}                           sw $16, 0($2) ; v[k]←$16
                           sw $15, 4($2) ; v[k+1]←$15
                           jr $31
```

Assuming k is stored  
in \$5,  
and the starting address  
of v[] is in \$4.

# Pseudo-instructions

---

- ❑ MIPS assemblers support **pseudo-instructions** that give the illusion of a more expressive instruction set, but are actually translated into one or more simpler, “real” instructions.
- ❑ For example, you can use the **li** and **move** pseudo-instructions:

```
li      $a0, 2000      # Load immediate 2000 into $a0
move    $a1, $t0       # Copy $t0 into $a1
```

- ❑ They are probably clearer than their corresponding MIPS instructions:

```
addi    $a0, $0, 2000  # Initialize $a0 to 2000
add     $a1, $t0, $0    # Copy $t0 into $a1
```

- ❑ We'll see lots more pseudo-instructions this semester.
  - A core instruction set is given in “Green Card” of the text (1<sup>st</sup> page).
  - Unless otherwise stated, you can always use pseudo-instructions in your assignments and on exams.

# Control flow in high-level languages

---

- ❑ The instructions in a program usually execute one after another, but it's often necessary to alter the normal control flow.
- ❑ **Conditional statements** execute only if some test expression is true.

```
// Find the absolute value of *a0
v0 = *a0;
if (v0 < 0)
    v0 = -v0;           // This might not be executed
v1 = v0 + v0;
```

- ❑ **Loops** cause some statements to be executed many times.

```
// Sum the elements of a five-element array a0
v0 = 0;
t0 = 0;
while (t0 < 5) {
    v0 = v0 + a0[t0];   // These statements will
    t0++;              // be executed five times
}
```

# MIPS control instructions

---

- In this lecture, we introduced some of MIPS's control-flow instructions

<code>j immediate</code>	// for unconditional jumps
<code>bne</code> and <code>beq \$r1, \$r2, label</code>	// for conditional branches
<code>slt</code> and <code>slti \$r1, \$r2, \$r3</code>	// set if less than (w/ and w/o an immediate)

- And how to implement loops

- Today, we'll talk about

- MIPS's pseudo branches
- if/else
- case/switch

# Pseudo-branches

---

- ❑ The MIPS processor only supports two branch instructions, **beq** and **bne**, but to simplify your life the assembler provides the following other branches:

```
blt $t0, $t1, L1 // Branch if $t0 < $t1
ble $t0, $t1, L2 // Branch if $t0 <= $t1
bgt $t0, $t1, L3 // Branch if $t0 > $t1
bge $t0, $t1, L4 // Branch if $t0 >= $t1
```

- ❑ Later this quarter we'll see how supporting just **beq** and **bne** simplifies the processor design.

# Implementing pseudo-branches

- ❑ Most pseudo-branches are implemented using `slt`. For example, a branch-if-less-than instruction `blt $a0, $a1, Label` is translated into the following.

```
slt  $at, $a0, $a1      // $at = 1 if $a0 < $a1
bne  $at, $0, Label     // Branch if $at != 0
```

- ❑ This supports immediate branches, which are also pseudo-instructions. For example, `blti $a0, 5, Label` is translated into two instructions.

```
slti  $at, $a0, 5      // $at = 1 if $a0 < 5
bne   $at, $0, Label   // Branch if $a0 < 5
```

- ❑ All of the pseudo-branches need a register to save the result of `slt`, even though it's not needed afterwards.
  - MIPS assemblers use register `$1`, or `$at`, for temporary storage.
  - You should be careful in using `$at` in your own programs, as it may be overwritten by assembler-generated code.

# Translating an if-then statement

---

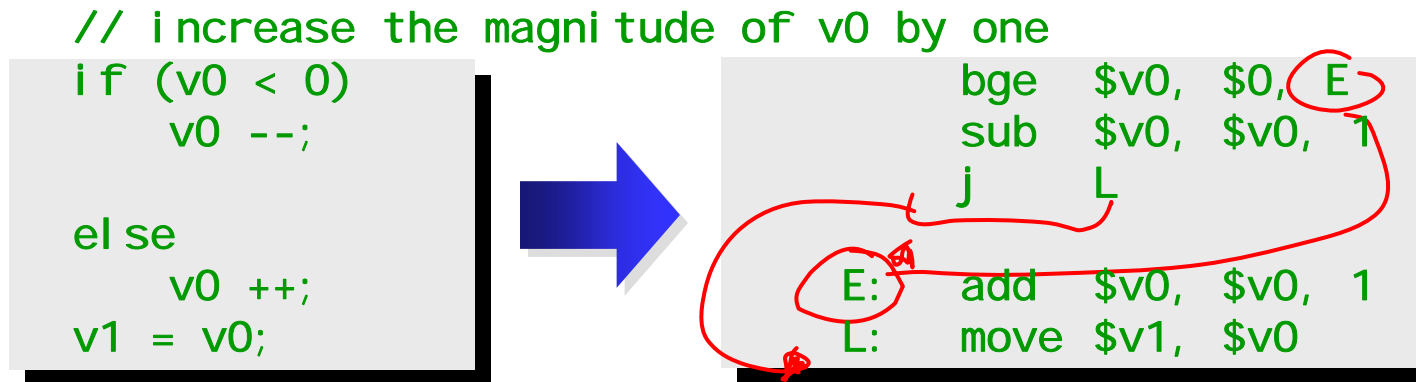
- We can use branch instructions to translate if-then statements into MIPS assembly code.



- Sometimes it's easier to *invert* the original condition.
  - In this case, we changed “continue if  $v0 < 0$ ” to “skip if  $v0 \geq 0$ ”.
  - This saves a few instructions in the resulting assembly code.

# Translating an if-then-else statements

- If there is an **else** clause, it is the target of the conditional branch
  - And the **then** clause needs a jump over the **else** clause



- Dealing with else-if code is similar, but the target of the first branch will be another if statement.
  - Drawing the control-flow graph can help you out.