

# 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.
- ❑ In addition to the **la** (load address) we saw on last lecture, 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 section, we introduced some of MIPS's control-flow instructions

<code>j immediate</code>	// for unconditional jumps
<code>bne and beq \$r1, \$r2, label</code>	// for conditional branches
<code>slt and 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 semester 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 `b<lt $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, `b<lti $a0, 5, Label` is translated into two instructions.

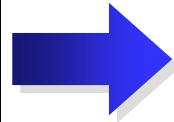
```
slti   $at, $a0, 5     // $at = 1 if $a0 < 5
bne    $at, $0, Label  // Branch if $a0 < 5
```

- ❑ 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.

```
v0 = *a0;  
if (v0 < 0)  
    v0 = -v0;  
v1 = v0 + v0;
```

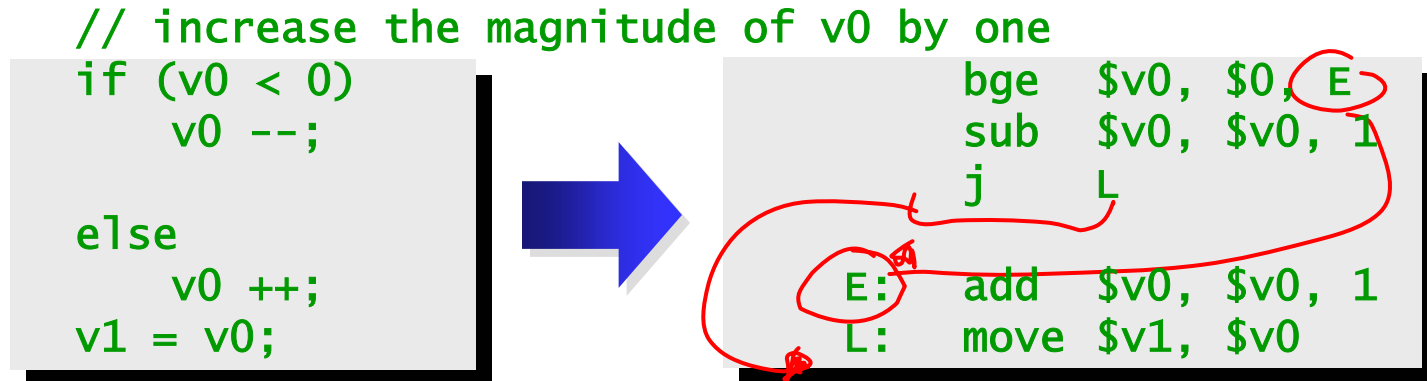


```
lw $t0, 0($a0)  
bge $t0, $zero, label1  
sub $t0, $zero, $t0  
label1: add $t1, $t0, $t0
```

- 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.

# Case/Switch statement

---

- ❑ Many high-level languages support **multi-way branches**, e.g.

```
switch (two_bits) {  
    case 0:    break;  
    case 1:    /* fall through */  
    case 2:    count ++;    break;  
    case 3:    count += 2;  break;  
}
```

- ❑ We could just translate the code to if, thens, and elses:

```
if ((two_bits == 1) || (two_bits == 2)) {  
    count ++;  
} else if (two_bits == 3) {  
    count += 2;  
}
```

- ❑ This isn't very efficient if there are many, many **cases**.

# Case/Switch statement

---

```
switch (two_bits) {  
case 0:    break;  
case 1:    /* fall through */  
case 2:    count ++;    break;  
case 3:    count += 2;  break;  
}
```

## □ Alternatively, we can:

1. Create an array of jump targets – jump table
2. Load the entry indexed by the variable `two_bits`
3. Jump to that address using the jump register, or **jr**, instruction

## □ This is much easier to show than to tell.

# Coding with jump table (sketch)

□ Suppose the jump table is stored in the memory. It's starting address is in \$t0.

- If `two_bits==1`, the branch should jump to the 2<sup>nd</sup> entry in the table, i.e., our target address is `$t0+4`.

□ Assume `two_bits` is in \$t1:

```
/* test the range of two_bits */
blt $t1, $zero, Exit
bge $t1, $a0, Exit          /* $a0==4 */
/* multiply two_bits by 4, to get byte addr */
sll $t1, $t1, 2
/* get the target address */
add $t1, $t1, $t0
lw $t2, 0($t1)
/* jump */
jr $t2
```

# Example of a Loop Structure

---

```
for (i=1000; i>0; i--)  
    x[i] = x[i] + h;
```



```
Loop: lw  $s0, 0($s1)           ;$s1=x[1000]  
      add $s3, $s0, $s2        ;$s2=h  
      sw  $s3, 0($s1)  
      addi $s1, $s1, # - 4  
      bne $s1, $s5, Loop       ;$s5=x[0]
```

# Exercise – do it on your own

---

- ❑ Let's write a program to count how many bits are set in a 32-bit word.

# Policy of Use Conventions

---

Name	Register number	Usage
\$zero	0	the 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	saved
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

# Functions calls in MIPS

---

- **We'll talk about the 3 steps in handling function calls:**
  1. The program's flow of control must be changed.
  2. Arguments and return values are passed back and forth.
  3. Local variables can be allocated and destroyed.
  
- **And how they are handled in MIPS:**
  - New instructions for calling functions.
  - Conventions for sharing registers between functions.
  - Use of a stack.

# Control flow in C

- ❑ Invoking a function changes the control flow of a program twice.
  1. **Calling** the function
  2. **Returning** from the function
- ❑ In this example the **main** function calls **fact** twice, and **fact** returns twice—but to *different* locations in **main**.
- ❑ Each time **fact** is called, the CPU has to remember the appropriate **return address**.
- ❑ Notice that **main** itself is also a function! It is called by the operating system when you run the program.

```
int main()
{
    ...
    t1 = fact(8);
    t2 = fact(3);
    t3 = t1 + t2;
    ...
}

int fact(int n)
{
    int i, f = 1;
    for (i = n; i > 1; i--)
        f = f * i;
    return f;
}
```

# Control flow in MIPS

---

- ❑ MIPS uses the jump-and-link instruction **jal** to call functions.
  - The jal saves the return address (the address of the *next* instruction) in the dedicated register **\$ra**, before jumping to the function.
  - jal is the only MIPS instruction that can access the value of the program counter, so it can store the return address PC+4 in \$ra.

jal Fact

- ❑ To transfer control back to the caller, the function just has to jump to the address that was stored in \$ra.

jr \$ra

- ❑ Let's now add the jal and jr instructions that are necessary for our factorial example.

# Changing the control flow in MIPS

```
int main()
{
    ...
    jal Fact;
    ...
    jal Fact;
    ...
    t3 = t1 + t2;
    ...
}

int fact(int n)
{
    int i, f = 1;
    for (i = n; i > 1; i--)
        f = f * i;
    jr $ra;
}
```

# Data flow in C

---

- ❑ Functions accept **arguments** and produce **return values**.
- ❑ The **black** parts of the program show the actual and formal arguments of the fact function.
- ❑ The **purple** parts of the code deal with returning and using a result.

```
int main()
{
    ...
    t1 = fact(8);
    t2 = fact(3);
    t3 = t1 + t2;
    ...
}

int fact(int n)
{
    int i, f = 1;
    for (i = n; i > 1; i--)
        f = f * i;
    return f;
}
```

# Data flow in MIPS

---

- ❑ MIPS uses the following conventions for function arguments and results.
  - Up to four function arguments can be “passed” by placing them in argument registers **\$a0-\$a3** before calling the function with jal.
  - A function can “return” up to two values by placing them in registers **\$v0-\$v1**, before returning via jr.
- ❑ These conventions are not enforced by the hardware or assembler, but programmers agree to them so functions written by different people can interface with each other.
- ❑ Later we’ll talk about handling additional arguments or return values.