Chapter 2

Instructions: Language of the Computer

Instruction for making decisions

- Decision making is commonly represented in programming languages using if statement and goto statement

- The following instruction **(branch if equal)** means go to the statement labeled L1, if the value in register1 **equal** the value in register2

```
beq register1, register2, L1
```

- The following instruction **(branch if not equal)** means go to the statement labeled L1, if the value in register1 **does not equal** the value in register2

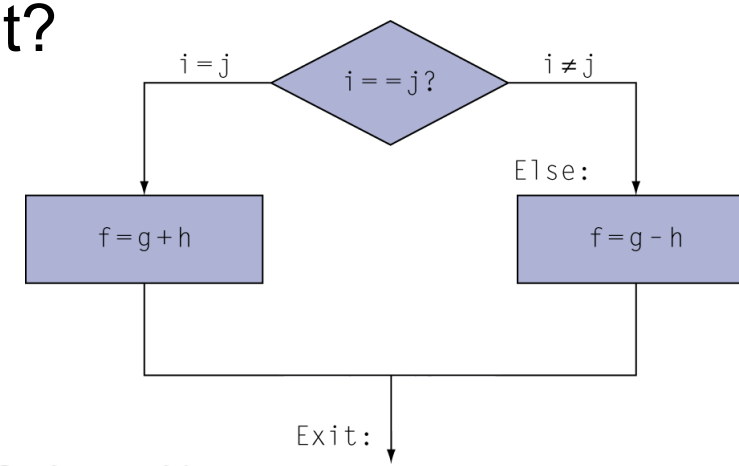
```
bne register1, register2, L1
```

- These two instructions are called **conditional branches**

Compiling If Statements

- If the five variables f through j correspond to the five registers $s0 through $s4, what is the compiled MIPS code for the following C if statement?

```
if (i == j) f = g + h; else f = g - h;
```




```
bne $s3,$s4,Else   # go to Else if i ≠ j
add $s0,$s1,$s2    # f = g + h (skipped if i ≠ j)
j Exit             # go to Exit
Else:sub $s0,$s1,$s2  # f = g - h (skipped if i = j)
Exit:
```

Unconditional branch

Compiling Loop Statements

- Assume that i and k correspond to registers $s3 and $s5 and the base of the array save is in $s6. What is the MIPS assembly code corresponding to the following C segment?

```
while (save[i] == k)
    i += 1;
```

```
Loop: sll  $t1,$s3,2     # Temp reg $t1 = i * 4
```

To get the address of `save[i]`, we need to add `$t1` and the base of save in `$s6`:

```
add $t1,$t1,$s6      # $t1 = address of save[i]
```

Now we can use that address to load `save[i]` into a temporary register:

```
lw $t0,0($t1)        # Temp reg $t0 = save[i]
```

The next instruction performs the loop test, exiting if `save[i] ≠ k`:

```
bne $t0,$s5, Exit    # go to Exit if save[i] ≠ k
```

Compiling Loop Statements

- Assume that i and k correspond to registers $s3 and $s5 and the base of the array save is in $s6. What is the MIPS assembly code corresponding to the following C segment?

```
while (save[i] == k)
        i += 1;
```

The next instruction adds 1 to `i`:

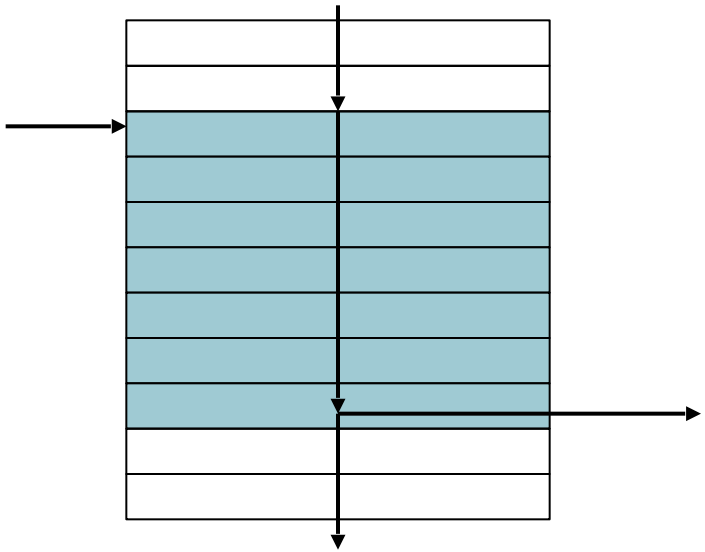
```
addi $s3,$s3,1        # i = i + 1
```

The end of the loop branches back to the *while* test at the top of the loop. We just add the `Exit` label after it, and we're done:

```
    j      Loop           # go to Loop
Exit:
```

Basic Blocks

- A basic block is a sequence of instructions with
  - No branches (except at end)
  - No branch targets or branch label (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

More Conditional Operations

- Sometime it is useful to see if a variable is less than another variable. For example, a *for* loop may want to test to see if the index variable is less than 0.

- Such comparisons are accomplished in MIPS assembly language with an instruction that **compares two registers and sets a third register to 1 if the first is less than the second; otherwise, it is set to 0.**

- The MIPS instruction is called ***set on less than,*** **or slt**. For example,

```
slt    $t0, $s3, $s4    # $t0 = 1 if $s3 < $s4
```

- To test if register $s2 is less than the constant 10, we can just write

```
slti    $t0,$s2,10      # $t0 = 1 if $s2 < 10
```

More Conditional Operations

- **Signed versus Unsigned**
- Comparison instructions must deal with the between signed and unsigned numbers.

- ***Set on less than* (slt)** and ***set on less than immediate* (slti)** work with signed integers.

- Unsigned integers are compared using ***set on less than unsigned* (sltu)** and ***set on less than immediate unsigned* (sltiu).**

More Conditional Operations

- **Signed versus Unsigned - example**

Suppose register $s0 has the binary number

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$$

and that register $s1 has the binary number

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two}$$

What are the values of registers $t0 and $t1 after these two instructions?

```
slt     $t0, $s0, $s1 # signed comparison
sltu    $t1, $s0, $s1 # unsigned comparison
```

The value in register $s0 represents -1_{ten} if it is an integer and $4{,}294{,}967{,}295_{ten}$ if it is an unsigned integer. The value in register $s1 represents 1_{ten} in either case. Then register $t0 has the value 1, since $-1_{ten} < 1_{ten}$, and register $t1 has the value 0, since $4{,}294{,}967{,}295_{ten} > 1_{ten}$.

Procedure

- Procedure: a store subroutine that performs a specific task based on the parameters with with which is provided
- Steps required
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call

Register Usage for procedure

- MIPS software follows the following convention for procedure calling in allocating its 32 registers:

- **$a0–$a3:** four argument registers in which to pass parameters
- **$v0–$v1:** two value registers in which to return values
- **$ra:** one return address register to return to the point of origin

Instruction for procedure

- MIPS assembly language includes an instruction just for the procedures: **it jumps to an address and** simultaneously **saves the address of the following instruction in register $ra**. The **jump-and-link instruction** (jal) is simply written

```
jal ProcedureAddress
```

- **Return address:** A link to the calling site that allows a procedure to return to the proper address; in MIPS it is stored in register $ra.

- MIPS use ***jump register*** **instruction (jr) for an unconditional jump to the address specified in a register:**

```
jr    $ra
```

- The jump register instruction jumps to the address stored in register $ra

Instruction for procedure

- The calling program, or **caller**, **puts the parameter values in $a0–$a3** and **uses jal X to jump to procedure X** (sometimes named the **callee**).
- The **callee** then performs the calculations, **places the results in $v0 and $v1**, **and returns control to the caller using jr $ra.**

Instruction for procedure

- **Caller**: The program that instigates a procedure and provides the necessary parameter values.
- **Callee**: A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller
- **Program counter (PC):** The register containing the address of the instruction in the program being executed.

Instruction for procedure

- **Stack**: A data structure for spilling registers organized as a last-in- first-out queue.
-
- **Stack Pointer**: A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found. In MIPS, it is register $sp.
- **Push:** Add element to stack
- **Pop:** Remove element from stack
- Any registers needed by the caller must be restored to the values that they contained *before* the procedure was invoked.

Example:Compiling a C Procedure That Doesn't Call Another Procedure

```
f = (g + h) - (i + j);
```

The variables f, g, h, i, and j are assigned to the registers $s0, $s1, $s2, $s3, and $s4, respectively. What is the compiled MIPS code?

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 - $t1, which is (g + h)-(i + j)
```

Example:Compiling a C Procedure That Doesn't Call Another Procedure

```
int leaf_example (int g, int h, int i, int j)
{
     int f;

     f = (g + h) - (i + j);
     return f;
}
```

What is the compiled MIPS assembly code?

- The parameter variables g, h, i, and j correspond to the argument registers $a0, $a1, $a2, and $a3, and f corresponds to $s0.
- we need to save three registers: $s0, $t0, and $t1. We "push" the old values onto the stack by creating space for three words (12 bytes) on the stack and then store them:

```
addi $sp, $sp, -12  # adjust stack to make room for 3 items
sw  $t1, 8($sp)     # save register $t1 for use afterwards
sw  $t0, 4($sp)     # save register $t0 for use afterwards
sw  $s0, 0($sp)     # save register $s0 for use afterwards
```

Example:Compiling a C Procedure That Doesn't Call Another Procedure

- The next three statements correspond to the body of the procedure
-

```
add $t0,$a0,$a1 # register  $t0 contains g + h
add $t1,$a2,$a3 # register  $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, which is (g + h)-(i + j)
```

- To return the value of f, we copy it into a return value register

```
add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

- Before returning, we restore the three old values of the registers we saved by "popping" them from the stack:

```
lw $s0, 0($sp)  # restore register $s0 for caller
lw $t0, 4($sp)  # restore register $t0 for caller
lw $t1, 8($sp)  # restore register $t1 for caller
addi $sp,$sp,12 # adjust stack to delete 3 items
```

- The procedure ends with a jump register using the return address:

```
jr   $ra    # jump back to calling routine
```

Reading assignment

- Read 2.7 and 2.8 of the text book