

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, then, and else:

```
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

```
jr    $r1
```

□ 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 2nd 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, $t0, $t1
lw $t2, 0($t1)
/* jump */
jr $t2
```

Example of a Loop Structure

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



Assume: addresses of
x[1000] and x[0]
are in \$s1 and \$s5
respectively; h is in
\$s2;

```
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]
```

Homework 1

- Let's write a program to count how many bits are zero in a 32-bit word.
 - Assigned 1/17. Due 1/24 before class

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.

Nested functions

- ❑ What happens when you call a function that then calls another function?
- ❑ Let's say A calls B, which calls C.
 - The arguments for the call to C would be placed in \$a0-\$a3, thus *overwriting* the original arguments for B.
 - Similarly, **jal C** overwrites the return address that was saved in \$ra by the earlier **jal B**.

```
A:  ...
    # Put B's args in $a0-$a3
    jal B      # $ra = A2
A2:  ...
```

```
B:  ...
    # Put C's args in $a0-$a3,
    # erasing B's args!
    jal C      # $ra = B2
B2:  ...
    jr $ra    # Where does
             # this go???
```

```
C:  ...
    jr $ra
```