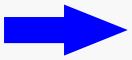
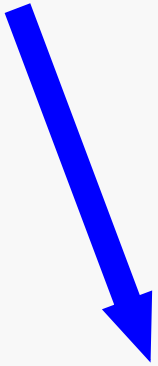
# Conditional Control Structure



```
if ( i == j )
   h = i + j;
```



```
# $s0 == i, $s1 == j, $s3 == h
        beq    $s0, $s1, doif # if-test
        b      skip           # skip if
doif:                         # if-body
        add    $s3, $s0, $s1
skip:   ....
```



```
# $s0 == i, $s1 == j, $s3 == h
        bne    $s0, $s1, skip  # test negation of C-test
        add    $s3, $s0, $s1   # if-body
skip:   ....
```

QTP: what's the advantage of the second version?

# Conditional Control Structure



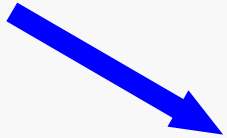
```
if ( i < j )
   i++;
else
   j++;
```

```
# $s3 == i, $s4 == j
        bge   $s3, $s4, doelse
        addi  $s3, $s3, 1   # if-body
        b     endelse       # skip else
doelse:
        addi  $s4, $s4, 1   # else-body
endelse:
```

# `while` Loop Example



```
int N = 100;
int i = 0;
while ( N > 0 ) {
   N = N / 2;    // N = N >> 1;
   i++;
}
```

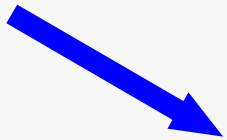


```
# $s0 == N, $t0 == i
        li    $s0, 100          # N = 100
        li    $t0, 0            # i = 0
loop:   ble   $s0, $zero, done  # loop test
        sra   $s0, $s0, 1       # calculate N / 2
        addi  $t0, $t0, 1       # i++
        b     loop              # restart loop
done:
```

# `while` Loop Alternative



```
int N = 100;
int i = 0;
while ( N > 0 ) {
   N = N / 2;
   i++;
}
```



```
# $s0 == N, $t0 == i
          li    $s0, 100          # N = 100
          li    $t0, 0            # i = 0
          ble   $s0, $zero, done # see if loop is necessary
loop:
          sra   $s0, $s0, 1       # calculate N / 2
          addi  $t0, $t0, 1       # i++
          bgt   $s0, $zero, loop # check whether to restart
done:
```

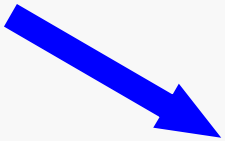
Converted while loop to a do-while

QTP: what's the advantage?

# `for` Loop Example



```
int Sum = 0, Limit = 100;
for (int i = 1; i <= Limit; ++i) {
   Sum = Sum + i*i;
}
```



```
# $s0 == Sum, $s1 == Limit, $t0 == i
          li    $s0, 0            # Sum = 0
          li    $s1, 100          # Limit = 0
          li    $t0, 1            # i = 1
loop:     bgt   $s0, $s1, done    # loop test
          mul   $t1, $t0, $t0     # calculate i^2
          add   $s0, $s0, $t1     # Sum = Sum + i^2
          addi  $t0, $t0, 1       # ++i
          b     loop              # restart loop
done:
```

QTP: convert this to the do-while pattern

# Program Termination

Unlike the high-level languages you are accustomed to, MIPS assembly does not include an instruction, or block syntax, to terminate the program execution.

MIPS programs can be terminated by making a system call:

```
    ## Exit
    li      $v0, 10  # load code for exit system call in $v0
    syscall          # make the system call to exit
```

Without such code, the system could attempt to continue execution into the memory words that followed the final instructions of the program. That rarely produces graceful results.

# Pseudo-Instructions

You may have noticed something is odd about a number of the MIPS instructions that have been covered so far. For example:

```
li   $t0, 0xFFFFFFFF
```

Now, logically there's nothing wrong with wanting to place a 32-bit value into one of the registers.

But there's certainly no way the instruction above could be translated into a 32-bit machine instruction, since the immediate value alone would require 32 bits.

This is an example of a *pseudo-instruction*. A MIPS assembler, or MARS, may be designed to support such extensions that make it easier to write complex programs.

In effect, the assembler supports an *extended MIPS architecture* that is more sophisticated than the actual MIPS architecture of the underlying hardware.

Of course, the assembler must be able to translate every pseudo-instruction into a sequence of valid MIPS assembly instructions.

# Pseudo-Instruction Examples



```
move   $t1, $t2   # $t1 <-- $t2
```

```
or   $t1, $t2, $zero  # recall:  x OR 0 == x
```

```
li   $t1, <imm>   # $t1 = 32-bit imm value
```

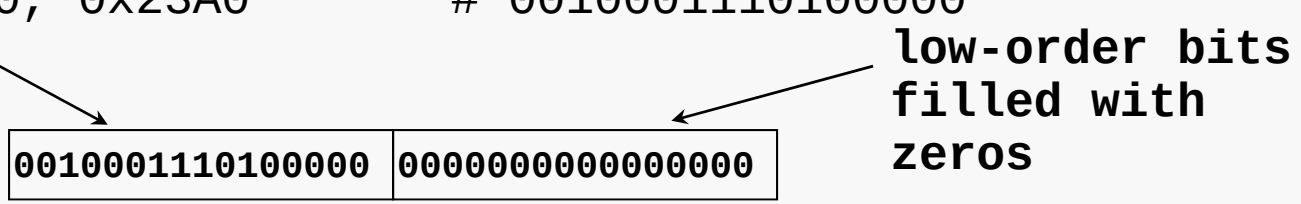
```
# e.g., suppose <imm> is 0x23A0FB18
#
# The assembler sometimes needs a register in which it can
# store temporary values.  The register $at is reserved for
# such use.

lui    $at, 0x23A0      # put upper byte in upper byte of reg,
                        #   and 0s in the lower byte

ori    $t1, $at, 0xFB18 # put lower byte into reg
```

# lui and ori Details

We'd like to be able to load a 32-bit constant into a register

Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 0x23A0        # 0010001110100000
```




Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, FB18  # 1111101100011000
```