MIPS conditional set instructions:

```
slt   $t0, $s0, $s1  # $t0 = 1 if $s0 < $s1
     # $t0 = 0 otherwise
slti  $t0, $s0, <imm>  # $t0 = 1 if $s0 < imm
     # $t0 = 0 otherwise
```

These are useful for "remembering" the results of a Boolean comparison for later use.
MIPS unconditional branch instructions:

- `j` Label # PC = Label
- `b` Label # PC = Label
- `jr` $ra # PC = $ra

These are useful for building loops and conditional control structures.
Conditional Branch Instructions

Decision making instructions
- alter the control flow,
- i.e., change the "next" instruction to be executed

MIPS conditional branch instructions:

\[
\text{bne} \quad \$t0, \$t1, <\text{label}> \quad \# \text{branch on not-equal} \\
\text{# PC += 4 + Label if} \\
\text{# \$t0 \neq \$t1} \\
\text{beq} \quad \$t0, \$t1, <\text{label}> \quad \# \text{branch on equal}
\]

Labels are strings of alphanumeric characters, underscores and periods, not beginning with a digit. They are declared by placing them at the beginning of a line, followed by a colon character.

\[
\text{if} \ (i == j) \ \\
\text{h = i + j;}
\]

\[
\text{bne} \quad \$s0, \$s1, \text{Miss} \\
\text{add} \quad \$s3, \$s0, \$s1 \\
\text{Miss: \ ....}
\]
if ( i < j )
    goto A;
else
    goto B;

# $s3 == i, $s4 == j
slt $t1, $s3, $s4
beq $zero, $t1, B

A:    # code...
b      C
B:    # code...
C:
for Loop Example

```c
int Sum = 0;
for (int i = 1; i <= N; ++i) {
    Sum = Sum + i;
}
```

```mips
# $s0 == Sum, $s1 == N, $t0 == i
move $s0, $zero       # register assignment
lw $s1, N           # assume global symbol
li $t0, 1           # literal assignment
loop: beq $t0, $s1, done   # loop test
       add $s0, $s0, $t0    # Sum = Sum + i
       addi $t0, $t0, 1      # ++i
       b    loop             # restart loop
done:
```
```c
int Sum = 0;
for (int i = 1; i <= N; ++i) {
    Sum = Sum + i;
}
```

```mips
# $s0 == Sum, $s1 == N, $t0 == i
move $s0, $zero       # register assignment
lw $s1, N           # assume global symbol
beq $s1, $zero, done # check for trivial case
li $t0, 1           # literal assignment
loop: add $s0, $s0, $t0    # Sum = Sum + i
     addi $t0, $t0, 1      # ++i
     bne $t0, $s1, loop   # loop test
done:
```
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 would attempt to continue execution into the memory words that followed the final instructions of the program. That rarely produces graceful results.
MIPS programmers are expected to conform to the following conventions when using the 29 available 32-bit registers:

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results and expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>

Register 1 ($at) is reserved for the assembler, 26-27 ($k0, $k1) for operating system.
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 SPIM, 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 0x23A0FB17
#
# 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
```
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, 43690       # 1010101010101010
```

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

```
ori $t0, $t0, 39321  # 1001100110011001
```

low-order bits filled with zeros
Directives are special reserved identifiers used to communicate instructions to the assembler.

For example:
# MARS System Calls

## Table of Available Services

<table>
<thead>
<tr>
<th>Service</th>
<th>Code in $v0</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print integer</td>
<td>1</td>
<td>$a0 = integer to print</td>
<td></td>
</tr>
<tr>
<td>print float</td>
<td>2</td>
<td>$f12 = float to print</td>
<td></td>
</tr>
<tr>
<td>print double</td>
<td>3</td>
<td>$f12 = double to print</td>
<td></td>
</tr>
<tr>
<td>print string</td>
<td>4</td>
<td>$a0 = address of null-terminated string to print</td>
<td></td>
</tr>
<tr>
<td>read integer</td>
<td>5</td>
<td></td>
<td>$v0 contains integer read</td>
</tr>
<tr>
<td>read float</td>
<td>6</td>
<td></td>
<td>$f0 contains float read</td>
</tr>
<tr>
<td>read double</td>
<td>7</td>
<td></td>
<td>$f0 contains double read</td>
</tr>
<tr>
<td>read string</td>
<td>8</td>
<td>$a0 = address of input buffer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a1 = maximum number of characters to read</td>
<td></td>
</tr>
</tbody>
</table>
**Basic fact:** at the machine language level there are no explicit data types, only contents of memory locations. The concept of type is present only implicitly in how data is used.

**Declaration:** reserving space in memory, or deciding that a certain data item will reside in a certain register.

Directives are used to reserve or initialize memory:

- `.data` # mark beginning of a data segment
- `.ascii" a string"` # declare and initialize a string
- `.byte 13, 14, -3` # store values in successive bytes
- `.space 16` # alloc 16 bytes of space
- `.word 13, 14, -3` # store values in successive words

A complete listing of MIPS/MARS directives can be found in the MARS help feature.
Arrays

First step is to reserve sufficient space for the array.

Array elements are accessed via their addresses in memory, which is convenient if you’ve given the .space directive a suitable label.

```mips
.data
list:  .word  2, 3, 5, 7, 11, 13, 17, 19, 23, 29
size:  .word  10
...
la $t1, list  # get array address
li $t2, 0  # set loop counter
print_loop:
  beq $t2, $t3, print_loop_end  # check for array end
  lw $a0, ($t1)  # print value at the array pointer
  li $v0, 1
  syscall
  addi $t2, $t2, 1  # advance loop counter
  addi $t1, $t1, 4  # advance array pointer
  j print_loop  # repeat the loop
print_loop_end:
```
Array Example

This is part of the palindrome example from the course website:

```
.data
string_space: .space 1024
...
# prior to the loop, $t1 is set to the address of the first
# char in string_space, and $t2 is set to the last one
test_loop:
bge  $t1, $t2, is_palin  # if lower pointer >= upper
    # pointer, yes

lb  $t3, ($t1)  # grab the char at lower ptr
lb  $t4, ($t2)  # grab the char at upper ptr
bne $t3, $t4, not_palin  # if different, it's not

addu $t1, $t1, 1  # advance lower ptr
subu $t2, $t2, 1  # advance upper ptr
j test_loop  # repeat the loop
...
```