Two branch instructions:

- `beq $t0, $t1, label`  # if $t0 == $t1, jump to “label”
- `bne $t0, $t1, label`  # if $t0 != $t1, jump to “label”

For branch instructions, the constant field is not an address, but an offset from the current program counter (PC) to the target address.

```
beq $t0, $t1, EQ
add $t0, $t0, $t1
addi $t1, $t0, $0
EQ: add $v1, $v0, $v0
```

Since the branch target EQ is three instructions past the beq, the address field contains 3
J-type format

- In most programs, branch targets are less than 32,767 instructions away
  - branches are mostly used in loops and conditionals
  - programmers are taught to make loop bodies short

- For “far” jumps, use j and jal instructions (J-type instruction format)

<table>
<thead>
<tr>
<th>opcode</th>
<th>address (exact)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

  - address is always a multiple of 4 (32 bits per instruction)
  - only the top 26 bits actually stored (last two are always 0)

- For even longer jumps, the jump register (jr) instruction can be used.

  jr $ra  # Jump to 32-bit address in register $ra
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

There are also immediate versions of these branches, where the second source is a constant instead of a register.

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 `blt $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
  ```

- Immediate versions: `slti` used instead of `slt`

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

- 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
  - if you use `$at`, the assembler will warn you
Translating an if-statement

- The easiest way is often to invert the original condition:

  ```
  if(t0 < t1) {
      ...
  } // other stuff
  ```

- How about an if-else statement?

  ```
  if(t0 < t1) {
      // thing 1
  } else {
      // thing 2
  } // other stuff
  ```
A while-loop is essentially an iterated if-statement:

```plaintext
while(t0 < t1) {
    ... 
}
// other stuff
```

Translate into MIPS:

```plaintext
if(a0 % 4 == 0)  // MIPS instruction: rem
    if(a0 % 100 != 0)
        v0 = 1;       // leap year
    else if(a0 % 400 == 0)
        v0 = 1;
    else v0 = 0;
else v0 = 0;
else v0 = 0;
```

```plaintext
loop: bge $t0, $t1, other 
    ... 
j   loop
other:  # other stuff
```
Memory: Variables and Arrays

- For now, we have been translating C++ variables as MIPS registers
  - C++ programs can have many variables, complex data structures

- Let’s consider a simple C++ code snippet:
  ```
  char c = 0;     // 1 byte of memory, initialized to 0
  c++;             
  ```

  1. Load `c` from memory into a register
  2. Add 1 to that register
  3. Store the result back into `c`

- More specifically, a value is being loaded from address `&c` into a register, modified, and then stored back into address `&c`

  ```
  la    $t0, c       # load address &c into t0
  lb    $t1, 0($t0)  # (load byte)  t1 = Mem[t0 + 0]
  addi  $t1, $t1, 1  # t1++
  sb    $t1, 0($t0)  # (store byte) Mem[t0 + 0] = t1
  ```
Declaring variables

- Variables (labeled chunks of memory) are declared in the `.data` section.

```plaintext
.data:
c: .byte 0 # char c = 0;
a: .byte 0 1 2 # char a[3] = {0, 1, 2};
b: .space 50 # char b[50];
.text:
main: ...
```

- Both `lb` and `sb` are I-type instructions: can specify a 16-bit immediate.

```plaintext
la $t0, a # t0 = &a[0]
lb $t1, 0($t0) # t1 = a[0]
lb $t2, 1($t0) # t2 = a[1]
sb $t2, 2($t0) # a[2] = t2
```

- This technique cannot be used to loop through an array: `$t1($t0)`
## Looping through an array

- Consider this code:
  ```c
  for(i = 0; i < 10; i++)
  a[i] = 0;
  ```

- What’s going on here?
  Answer: Pointer arithmetic!

```c
for(char *p = &a[0]; p < (a + 10); p++)
  *p = 0;
```

```assembly
li $t0, 0  # t0 = i
la $t1, a  # t1 = &a[i]

loop:
  bge $t0, 10, done
  sb $0, 0($t1)  # a[i] = 0
  addi $t0, $t0, 1
  addi $t1, $t1, 1
  j loop
done:
```

```assembly
la $t0, a  # t0 = &a[i]
addi $t1, $t0, 10

loop:
  bge $t0, $t1, done
  sb $0, 0($t0)  # a[i] = 0
  addi $t0, $t0, 1
  j loop
done:
```
Accessing random array elements

/* a = array of char */
lo = 1, hi = 31;

while(lo <= hi) {
    mid = (lo+hi)/2
    if(a[mid]==17) break;
    if(a[mid]<17) lo = mid+1;
    else hi = mid-1;
}

- A complication: array of (32-bit) ints
  - &a[i] is actually &a[0] + 4*i

- Add 4 to pointer when looping through an array of ints
- Use lw/sw instead of lb/sb

```
la    $t0, a
li    $t1, 1
li    $t2, 31

loop:
    bgt   $t1, $t2, done
    add   $t3, $t1, $t2
    sra   $t3, $t3, 1
    add   $t4, $t3, $t0
    lb    $t4, 0($t4)
    beq   $t4, 17, done
    bge   $t4, 17, else
    addi  $t1, $t3, 1
    j     loop

else:
    addi  $t2, $t3, -1
    j     loop

done:
```
/* a = array of int */
for(i = 0; i < 10; i++) {
    if(a[i]==a[i+1]) break;
}

Try this on your own:

/* a = array of int */
for(i = 9; i >= 0; i--) {
    a[i+1]=a[i];
}
a[0] = 0;

```
la    $t0, a
addi  $t1, $t0, 40
loop:
    beq $t0, $t1, done
    lw   $t2, 0($t0)
    lw   $t3, 4($t0)
    beq $t2, $t3, done
    addi $t0, $t0, 4
    j     loop
done:
```
```