In lecture, so far we have only talked about how to write *straight line* MIPS code, i.e. sequences of instructions that execute one after another. To implement anything interesting, we need to introduce *control flow* (i.e., loops and conditionals). Here, we'll introduce some MIPS control-flow instructions and discuss how to translate simple `for` loops into MIPS assembly code.

Simple Conditions and Branches

Unconditional branch: always taken, much like a `goto` statement in C.

```
j   Loop
```

Example: Infinite Loop

```
Loop:    j    Loop  # goto Loop
```

The label `Loop` lets us identify which assembly instruction should be executed after the branch. The label could be anything, as long as the MIPS assembler doeesn't misinterpret it as an instruction.

Conditional branch: branch only if condition is satisfied.

```
bne reg1, reg2, target  # branch if reg1 != reg2
```

`bne` stands for ***B**ranch if **N**ot **E**qual*, so the branch is taken if the values in the two registers are not equal; otherwise, execution continues to the following instruction (sometimes called the *fallthrough path*). There is also a ***B**ranch if **EQ**ual* instruction `beq`.

What if you want to branch if one register is less than another? We need to synthesize that out of two instructions: one that compares two values and then a branch instruction based on the resulting comparison. MIPS provides the `slt` (***S**et if **L**ess **T**han*) instruction and its "immediate" version `slti`, as illustrated below.

Translating `for` loops

Here is a simple `for` loop in C, translated into MIPS alongside:

```
for(i = 0; i < 4; ++i) {
  // stuff
}
```

```
  li   $t0, 0       # t0 = i = 0
for_loop:
  slti $t1, $t0, 4  # t1 = 1 if and only if t0 < 4
  beq  $t1, $0, for_loop_done
  #    stuff
  addi $t0, $t0, 1
  j    for_loop
for_loop_done:
```

Note that the translation for `while`-loops is similar.

1. `do-while` **loops**

   Translate the following snippet of C code into MIPS:

```
i = 0;
do {
  // stuff
  i++;
} while(i < 4);
```

2. `while`**-style or** `do-while`**-style?**

   In the following snippet of C code, the value of variable `n` is unknown at compile-time. In order to minimize the number of instructions executed at run-time, should a C-to-MIPS compiler translate the loop using a `while`-style or a `do-while`-style loop?

```
for(i = 0; i < n; ++i) {
  // stuff
}
```

3. **Loop through an array**

   If `$a0` holds the *base address* of an array A of bytes, the following MIPS code reads $A[0]$ into register `$t0`: `lb $t0, 0($a0)`

   To read $A[i]$ into register `$t0` requires two steps: (1) compute the address `&`$A[i]$, and (2) read the byte from that address. Assuming i is stored in register `$t1`, the MIPS code to read $A[i]$ into `$t0` is:

```
add  $t2, $a0, $t1   # t2 = &A[t1]
lb   $t0, 0($t2)     # t0 = A[t1]
```

   Now translate this code into MIPS:

```
int sum = 0;
for(int i = 0; i < n; ++i)
  sum += A[i];
```