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.

Comparison Instructions: Useful for computing conditions for branch instructions.

```
slt d_reg, s_reg1, s_reg2 # d_reg = 1 if (s_reg1 $<$ s_reg2), otherwise 0
```

`slt` stands for ***S**et if **L**ess **T**han*. There is also a ***S**et if **L**ess **T**han **I**mmediate* instruction (`slti`), for comparing against a constant, which we demonstrate below. The MIPS assembler supplies pseudo instructions for other compare operations like `sle` (***S**et if **L**ess than or **E**qual*).

Translating `for` loops with a constant number of iterations

Here is a simple `for` loop in C:

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

Below is the loop translated into MIPS assembly:

```
        add $t0, $zero, $zero   # i is initialized to 0, $t0 = 0
Loop:   // stuff
        addi $t0, $t0, 1        # i ++
        slti $t1, $t0, 4        # $t1 = 1 if i < 4
        bne $t1, $zero, Loop    # go to Loop if i < 4
```

1. Loop unrolling

   Vector operations are common in many applications, such as image and sound processing applications. Assume that we have three vectors `A`, `B` and `C`, each containing sixty-four 32-bit integers. We can represent these vectors with arrays, and perform a vector addition `A = B + C` by summing together the individual elements of `B` and `C`:

   ```
   for (i = 0; i < 64; i++) {
       A[i] = B[i] + C[i];
   }
   ```

   Assuming the values of `$t0`, `$t1`, and `$t2` are set to the starting addresses of arrays `a`, `b`, and `c` respectively, the loop can be translated into the following MIPS code:

   ```
           add $t4, $zero, $zero    # I1 i is initialized to 0, $t4 = 0
   Loop:   add $t5, $t4, $t1        # I2 temp reg $t5 = address of b[i]
           lw $t6, 0($t5)           # I3 temp reg $t6 = b[i]
           add $t5, $t4, $t2        # I4 temp reg $t5 = address of c[i]
           lw $t7, 0($t5)           # I5 temp reg $t7 = c[i]
           add $t6, $t6, $t7        # I6 temp reg $t6 = b[i] + c[i]
           add $t5, $t4, $t0        # I7 temp reg $t5 = address of a[i]
           sw $t6, 0($t5)           # I8 a[i] = b[i] + c[i]
           addi $t4, $t4, 4         # I9 i = i + 1
           slti $t5, $t4, 256       # I10 $t5 = 1 if $t4 < 256, i.e. i < 64
           bne $t5, $zero, Loop     # I11 go to Loop if $t4 < 256
   ```

   This program contains 11 instructions.

   (a) How many instructions are executed by the CPU to execute this code?

   (b) The above loop is not particularly execution efficient; much of the work in each iteration is spent computing the memory addresses and resolving control flow. One technique to reduce this overhead is *loop unrolling*. Since we know that the loop is going to be executed exactly 64 times, we can completely unroll the loop, resulting in the following C code:

   ```
   A[0] = B[0] + C[0];
   A[1] = B[1] + C[1];
       .
       .
       .
   A[63] = B[63] + C[63];
   ```

   Show how you can write these three additions in MIPS assembly language, using as few instructions as possible. Assume that vectors A, B and C are stored in main memory, and their addresses are in registers `$t0`, `$t1` and `$t2`, respectively.

(c) If you kept doing this, how many MIPS instructions would you have to write for the entire 64-element addition?

(d) Unrolling reduces the number of instructions executed (*the dynamic instruction count*) for a larger program (*static instruction count*). The two cases weve seen are only the end points of a continuum.

Write code for the loop that has been unrolled by a factor of two; that is:

```
for (i = 0; i < 64; i += 2) {
    A[i] = B[i] + C[i];
    A[i+1] = B[i+1] + C[i+1];
}
```

(e) Write equations that compute the static and dynamic instruction counts for the above loop that are parameterized by the unrolling factor. Your answer should handle unrolling factors of 1 (i.e., no unrolling), 2, 4, 8, 16, and 32.

Memory access in MIPS (example):

```
.data                       # data segment
A: .word 7, 3, 2, 5         # declare and initialize an array of words
length: .byte 4             # declare and initialize a byte variable

.text                       # code segment

main:
  lb   $t0, length          # load *value* at memory location length into $t0
  la   $a0, A               # load the *address* at memory location A into $a0
  add  $t1, $zero, $zero
loop:
  lw   $t2, 0($a0)          # load array element from memory
  addi $t2, $t2, 1          # increment element
  sw   $t2, 0($a0)          # write back to memory
  addi $a0, $a0, 4          # increment array pointer by 4 (word = 4 bytes)
  addi $t1, $t1, 1          # increment loop counter by 1
  blt  $t1, $t0, loop       # loop, if necessary
```