So far our LC-3 programs have been “straight lines”: we execute one instruction after the next and then the program stops when it hits the end.

What if we want to jump ahead to a different part of the program? Or jump back? How can we do that?

In LC-3, there are four different types of control operations:

- unconditionally jump to some other instruction (this is called a jump)
- jump to some other instruction depending on the result of the previous instruction (this is called a branch)
- jump to some other part of your own code, and then come back when you're done (this is called a subroutine)
- jump to some other part of some other code, and then come back when you're done (this is called a trap)

Today we will look at the first two; we'll see the other two next week.

**Jump instruction**

The jump instruction is the simplest one. The instruction reads a value from a register and sets the PC to that value. Thus, on the next fetch-decode-execute cycle, the instruction at that address will be read and executed.

For instance:

JMP R4

means “read the value from register R4 and set the PC to that value”. So if R4 holds the value x2013, then the next instruction to be executed (after this one) will be the one at address x2013.

The encoding for JMP R4 is as follows:

- the opcode is 1100
- bits 11-9 are unused, so they are all zeroes
- bits 8-6 hold the register; in this case it would be 100
- the last six bits are all zeroes

**Branch instruction**

The JMP instruction is “unconditional”: it modifies the value of the PC no matter what.

However, we often need the ability to jump only under certain circumstances, and to continue going on to the next instruction in others. That's where the branch instruction comes in.

Before we see the branch instruction, let's discuss three one-bit registers (sometimes referred to as “flags”) that are in the CPU. These registers are known as N, Z, and P, and they indicate whether the result of the last write to a register was negative, zero, or positive, respectively.

For instance, after the instruction AND R4, R2, #0 is executed, the Z flag is set to 1 (since the result written to R4 would necessarily be 0), but N and P would be 0. Note that, if the next instruction is not a write to a register (e.g. if it's a write to memory), the N, Z, and P flags would not change.
The branch instruction works by querying those flags and then deciding whether or not to branch based on which flags are set.

For instance, the instruction

```
BRnz #3
```

would work as follows:

- inspect the N and Z flags
- if either is set to 1 (because the last write to a register was negative or zero), then add the offset 3 to the already-incremented PC
- if neither N nor Z is set to 1 (because the last write to a register was positive), then leave the PC alone and carry on to the next instruction

The encoding of BRnz #3 would be as follows:

- the opcode for BR is 0000
- the next three bits represent whether we want to check the N, Z, and P flags respectively; in this case, they would be 110 because we want to see if N or Z is set to 1, but not P
- the remaining nine bits are for the offset; here it would be 000000011

Note that, when you're writing an LC-3 program, you would usually use a label for the address to which you want to branch, rather than hand-calculating and hard-coding the offset (see the previous lecture's notes for a discussion of address labels).

**Branches and Loops**

As you probably know, a “loop” is a piece of code that executes numerous times. It generally uses some counter to control how many times to execute the loop.

In the example below, the value in R0 acts as the counter. When we first enter the loop, it is 10. After completing the loop, it is decremented down to 9. At the branch instruction, we jump back to the start of the loop as long as the counter (i.e., the value in R0) is positive.

```
AND R0, R0, #0 ; set R0 = 0
ADD R0, R0, #10; now R0 = 10, this is our counter

LOOP first instruction of loop
second instruction of loop
and so on. . .

ADD R0, R0, #-1 ; decrement the counter
BRp LOOP ; go back if R0 is still positive
;; otherwise, R0 counted down to zero, so continue from here
```

**Example**

In this example, we will combine a lot of what we've seen so far: ALU operations, memory operations, and control operations.

Let's say that we have some data in memory addresses that we'll label X and Y. Our program will read the values at those addresses and compare their values. If the value at address X is bigger than the value
at address Y, we will write the value at X to address Z; otherwise, we'll write the value at Y to Z.

In many programming languages, such a program would be written as something like:

```plaintext
define program
if (X > Y)
    Z = X
else
    Z = Y
end if
```

or something along those lines.

Here is the program in its entirety. Note that the line numbers on the side are not part of the program, of course; they are just there for purposes of discussion.

```
.ORIG x3000
0      LD R0, X       ; read from address X and put value in R0
1      LD R1, Y       ; read from address Y and put value in R1
2      NOT R2, R1     ; flip the bits in the value from Y
3      ADD R2, R2, 1 ; now R2 holds the negative of Y
4      ADD R2, R0, R2 ; now R2 holds X - Y
     ;; at this point, the P flag will be set if X > Y
5      BRnz NEXT      ; if X - Y was zero or negative, 
     ;; then X < Y, so branch
     ;; if we get here, we didn't branch, so X > Y
6      ST R0, Z       ; put the value of X (which is in R0) in Z
7      BRnzp DONE     ; jump to the end of the program
     ;; if we get here, we branched, so X <= Y
8      NEXT ST R1, Z       ; put the value of Y (which is in R1) in Z
9      DONE HALT       ; the end!
10     X .FILL x1234
11     Y .FILL xABCD
12     Z .FILL x0
.END
```

Be sure you can follow the logic of this program before proceeding. Note that:

- to determine whether X > Y, we can calculate X – Y; this means we need to get the negative of the value held at Y, which is done on lines 2 and 3
- on line 4, we add X + (-Y) to see whether X > Y
- if X ≤ Y, the write to register R2 on line 4 will be either zero (if they're equal) or a negative value since X – Y will be zero or negative; thus on line 5, we want to branch (because either the N or Z flag will be set) down to line 8, where we take the value from Y (which is now in R1) and write it to Z; then we go to line 9 and we're done
• if, however, X > Y, the write to register R2 on line 4 will be a positive value, since X – Y will be positive. Thus, on line 5, we don’t branch but rather go to line 6, in which we take the value from X (which is now in R0) and write it to Z; on line 7, we then jump (BRnzp will always branch, since one of those flags will always be set) down to line 9 and we're done

The LC-3 Assembler
Recall from last time (and as mentioned above) that determining the offsets for the memory and control instructions can be tedious and error-prone. So, in converting (or “assembling”) an LC-3 program from human-readable assembly language to machine-readable binary code, we use a program called an assembler and let it do the work.

The goal of the assembler is to calculate all the offsets for the different instructions, and then encode each instruction. To do this, it must make two passes through the code, i.e. read the code line-by-line twice.

On the first pass, the assembler creates a symbol table. This is a mapping of labels to their offset from the first instruction (which we assume has an offset of zero). You can also think of it as a mapping of labels to their memory addresses, since in an LC-3 program we know exactly where the program will be loaded, but in general we wouldn't know that.

In the program above, the assembler would create a symbol table that looks like this:

<table>
<thead>
<tr>
<th>label</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEXT</td>
<td>8</td>
</tr>
<tr>
<td>DONE</td>
<td>9</td>
</tr>
<tr>
<td>X</td>
<td>#10</td>
</tr>
<tr>
<td>Y</td>
<td>#11</td>
</tr>
<tr>
<td>Z</td>
<td>#12</td>
</tr>
</tbody>
</table>

That is, the instruction labeled “NEXT” is 8 addresses after the first one, “DONE” is 9 addresses after the first one, and so on.

In the second pass, the assembler encodes each instruction one at a time. For instructions that require an offset, it determines the difference between the line it's currently encoding and the offset (relative to the start of the program) of the label that's being used. Of course, it has to account for the fact that the PC has already been incremented.

For instance, to encode the first instruction “LD R0, X”, it would determine that the offset of that instruction is 0 (since it's the first instruction), the offset of the label X is #10 (from the symbol table) and thus the offset in the encoded instruction needs to be #10 – (0 + 1), with the extra 1 coming from the incremented PC. Thus, this instruction is encoded as:

• 0010 for LD
• followed by 000 for R0
• followed by 000001001 for the offset of 9
The encoding for the next instruction “LD R1, Y” works in a similar manner. Note that the offset here is also 9. Why is that? Because the distance from this instruction to the address labeled Y is 9.

The encoding for the next three instructions is pretty straightforward since none of them use any labels.

What about for the “BRnz NEXT” instruction? Here, it takes of the offset of NEXT (which is 8) and subtracts the current offset (which is 5) plus one, giving us $8 - (5+1) = 2$. So the encoding is

1. **0000** for BR
2. **110** for the N and Z flags
3. **00000010** for the offset of 2

Here's the entire encoding of the program, along with the addresses at which each encoded value would be stored when the program is loaded (note, of course, that the value at the address labeled Z would change as a result of executing this program):

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>0010000000001001</td>
<td>“LD R0, X” as described above</td>
</tr>
<tr>
<td>x3001</td>
<td>0010001000010001</td>
<td>“LD R1, Y” as described above</td>
</tr>
<tr>
<td>x3002</td>
<td>1001010001100000</td>
<td>“NOT R2, R1”; doesn't use symbol table</td>
</tr>
<tr>
<td>x3003</td>
<td>0001010010100001</td>
<td>“ADD R2, R2, #1”</td>
</tr>
<tr>
<td>x3004</td>
<td>0001010000000010</td>
<td>“ADD R2, R0, R2”</td>
</tr>
<tr>
<td>x3005</td>
<td>00001100000000010</td>
<td>“BRnz NEXT” as described above</td>
</tr>
<tr>
<td>x3006</td>
<td>0011000000000010</td>
<td>“ST R0, Z”; note that the offset is 5</td>
</tr>
<tr>
<td>x3007</td>
<td>0000111000000001</td>
<td>“BRnzp DONE”; here the offset is 1</td>
</tr>
<tr>
<td>x3008</td>
<td>0011001000000011</td>
<td>“ST R1, Z”; here the offset is 3</td>
</tr>
<tr>
<td>x3009</td>
<td>1111000000100101</td>
<td>“TRAP”</td>
</tr>
<tr>
<td>x300A</td>
<td>00010001000110100</td>
<td>the filled value x1234</td>
</tr>
<tr>
<td>x300B</td>
<td>1010101111001101</td>
<td>the filled value xABCD</td>
</tr>
<tr>
<td>x300C</td>
<td>0000000000000000</td>
<td>the filled value 0</td>
</tr>
</tbody>
</table>