Let's start with a review of the steps taken by the Control Unit (CU). We can break these down into three simple steps: fetch, decode, and execute.

In the fetch phase, the CU needs to get the instruction to execute. It does this by placing the value in the Program Counter (PC) into the MAR, since the PC holds the address of the next instruction to execute, and we want to read it from memory. When that value is put into the MAR, it then reads from that address and the data is put in the MDR, from where it is then moved into the Instruction Register (IR). And voila! The instruction to execute is now where we want it. Note that, as a side effect, the PC needs to be incremented in this stage to point to the next instruction, which presumably (but not necessarily) is in the next location in memory.

In the decode stage, the CU figures out which operation it needs to execute by decoding the 1s and 0s in the instruction. It can use a decoder (which we saw in Chapter 3) to select/activate one circuit and ignore the others.

In the execute stage, the CU passes the operands to the appropriate place:

- for an ALU operation, the data will be moved from the register(s) or from the instruction into the circuit, and when the output comes out, it will be moved into the destination register
- for a write to memory, the address to write to will be determined and then put in the MAR, and the data to write will be moved from a register to the MDR; then, the data in the MDR is written to the address held in the MAR
- for a read from memory, the address to read will be determined and then put in the MAR; the data held at that address will be put in the MDR, and then moved to the destination register

**LC-3 ALU Operations**

The LC-3 supports three ALU operations: NOT, AND, and ADD.

**NOT operation**
The NOT operation takes two registers as its operands: the source and the destination. The value in the source register is read, a logical “not” is applied to each bit, and the result is written to the destination register.

For instance, if the value 0011 1000 0001 1010 were stored in register R5, then the instruction:

\[ \text{NOT R2, R5} \]

would take the value in R5 (the source is the second operand), apply logical “not” to each bit, and write the value 1100 0111 1110 0101 to the destination register R2.

The instruction “NOT R2, R5” is known as an **assembly language instruction**. It is a simple, human readable format that directly corresponds to a single LC-3 instruction, i.e. one fetch-decode-execute cycle.

Of course, LC-3 can't understand “NOT R2, R5”, since that is human readable and the computer can only understand 1s and 0s. So we need some way of **encoding** that assembly language instruction as a sequence of 1s and 0s that means the same thing. The encoding, i.e. what each bit “means” is
determined by the Instruction Set Architecture, as we talked about last time.

For LC-3, the NOT instruction is encoded as follows:

- the first four bits (we'll call them bits 15 through 12, going from left-to-right) encode the operation: in this case 1001 means “NOT”. In fact, all LC-3 instructions use the first four bits to represent the operation (this is super-convenient, as you'll see as we go along)
- the next three bits (bits 11-9), represent the destination register as a binary number. In the above example, since the destination register is R2, these bits would be 010, which corresponds to binary 2
- the next three bits (bits 8-6) represent the source register; in this example, those would be 101 to represent R5
- that's all we need for NOT, so the remaining six bits (bits 5-0) are just 1s

Thus, the encoding for NOT R2, R5 is 1001010101111111.

What, then, does 1001110000111111 represent? Well, we'd need to first look at the first four bits to get the operation. Here, they are 1001, which means NOT.

Now that we know that, we know that for a NOT instruction, the next three bits will represent the destination register. Those bits are 110, meaning R6.

We also know that the next three bits are the source register. Those are 000, which is R0.

Therefore, 1001110000111111 represents NOT R6, R0.

**AND and ADD operations**
The AND and ADD operations are very similar in their structure, so we'll discuss them together.

In both cases, we have a destination register (where the result goes) and a source register (where one of the operands comes from). But we also have an option for the second operand; it can come from:

- a register; or
- a value directly stored in the instruction (known as an immediate operand)

For instance, AND R2, R4, R3 would take the value in R4, perform a bitwise logical “and” with the value in R3, and put the result in R2.

On the other hand, AND R1, R6, #10 would take the value in R6, perform a bitwise logical “and” with the 16-bit representation of the decimal value 10 (the # in front of a number means “decimal”), and put the result in R1.

Since both “register mode” and “immediate operand mode” use the same opcode, there must be some way of indicating what all the bits mean (i.e., whether they're representing a register or an immediate value) in the encoding. In AND and ADD, this is done using a “mode bit”.

For AND, the encoding is as follows:

- the opcode is 0101, regardless of whether the second operand is a register or an immediate value
• bits 11-9 represent the destination register
• bits 8-6 represent the first source register
• bit 5 is the mode bit: 0 means “source register”, “1” means “immediate operand”
• for the remaining bits:
  • if source register mode, then bits 4-3 are 0s and the last three bits represent the second
    source register
  • if immediate operand mode, then the last five bits represent the value in 2’s complement

For example, to encode AND R2, R4, R3:
• the first four bits are 0101 for the opcode for AND
• the next three bits are 010 for R2
• the next three bits are 100 for R4
• the next bit is 0 to indicate source register mode
• the next two bits are 0s (unused)
• the last three bits are 011 for R3
So the encoding is 0101010100000011.

To encode AND R1, R6, #10:
• the first four bits are still 0101
• the next three are 001
• then 110
• then 1 to indicate immediate operand mode
• then 01010 to represent decimal 10
So the encoding is 0101001110101010.

Note that, for immediate operand mode, you only have five bits and have to represent a 2’s complement
number. So the biggest value you could represent is 01111, or +15. The smallest (most negative) value
is 10000, or -16.

As mentioned, ADD works the same way in terms of having two modes; the only difference is that its
opcode is 0001.

**LC-3 programs**
There is no subtract operation in LC-3. So does that mean that it’s impossible to do subtraction? No, it
just means that we need a series of instructions that will accomplish the same thing. Such a series of
instructions is known as a program (but you kind of knew that already, right?).

Let’s say we want to subtract the value in R1 from the value in R5, and put the result in R4. Something
like R4 = R5 − R1. To do this, we could use the ADD operation to calculate R4 = R5 + (-R1). In order
to get the negative of R1, we could calculate its 2’s complement negative representation: flip the bits
(using NOT) and add 1.

So we could write our LC-3 program like this:
```
NOT R1, R1
ADD R1, R1, #1
ADD R4, R5, R1
```
The first instruction flips the bits in R1, and the second adds 1, so that at that point, R1 holds the negative value of what was originally there. In the third instruction, we add it to R5 and put the result in R4.

How does an LC-3 program get executed? First, a program called an **assembler** converts our human-readable LC-3 assembly language program into its machine language (binary) representation. As we'll see later, this can be a somewhat tricky proposition, but for the program above, it's quite straightforward because each line can be encoded independently.

```
NOT R1, R1       →       1001 001 001 111111
ADD R1, R1, #1   →       0001 001 001 1 00001
ADD R4, R5, R1   →       0001 100 101 0 00 001
```

The encoded instructions are then loaded into memory starting at an address that is specified in the program, e.g. address x3000. Then, the PC is set to the first of those addresses, and the Control Unit's fetch-decode-execute cycle begins there.

That is, the PC reads the first instruction (1001001001111111) from address x3000, decodes it by figuring out that it's a NOT instruction, and then executes it by moving the data from R1 (the source) into the ALU, taking the result (which is NOT R1, since only the NOT circuit was selected/activated by the decoder), and putting it in R1 (the destination).

At that point, the PC is set to x3001. So the next instruction (0001001001100001) is read, decoded, and executed.

Then, when the PC is set to x3002, the last instruction (0001100101000001) is read, decoded, and executed, and our program is finished.

In this class, we will write LC-3 programs in files that look like this:

```
.ORIG x3000
NOT R1, R1 ; flip the bits in R1
ADD R1, R1, #1 ; now R1 has been negated
ADD R4, R5, R1 ; add to R5 and store in R4
HALT ; special keyword to end the program
.END
```

Everything after a semicolon is a comment and is ignored by the assembler. It's just there to help the human who's reading the code.

Note that “.ORIG x3000” is not encoded into an instruction; rather, it tells the assembler that the first instruction of this program should be loaded into address x3000.

Likewise, the “.END” at the end is not an instruction, but just tells the assembler that it's reached the end of the program.