Now let's see how an LC-3 program can read data from (“load”) and write data to (“store”) memory.

Whenever we load/read from memory, we always have a destination register. And when we write/store to memory, we always have a source register.

That means that our load and store instructions have four bits for the opcode and three bits for the destination or source register, leaving us just nine bits left over. But keep in mind that we have a 16 bit address space! Does that mean that we can only read addresses that are nine bit numbers? Or that we can only access a range of $2^9$ addresses?

Fortunately not. In LC-3, there are a variety of addressing modes that deal with these concerns.

**PC-relative mode**
The simplest addressing mode treats the remaining nine bits of the instruction as an “offset”, and not a specific address. It takes that value, adds it to the value in the PC, and then uses that calculated value as the address from which to read or write.

For instance, the instruction:
LD R1, x4B
means “add the hexadecimal value x4B to the value in the PC, treat that value as an address, read from that address, and load the result into register R1”.

Important note! Keep in mind that when this instruction is executed, the PC will already have been incremented to the next instruction. So if this instruction is at address x2002, the value that is calculated is x2002 + 1 + x4B = x204E, not x204D. Thus, x204E is put into the MAR, the value at that address is moved into the MDR, and then into R1.

To encode this:
- the first four bits are 0010 for the opcode for LD
- the next three bits are the destination register (001 in this case)
- the remaining nine bits represent the PC-offset as a two's complement number (in this case, the offset x4B would be represented as 0 0100 1011).

There is a corresponding store instruction, too. In assembly language, it is “ST” and the opcode is 0011.

The implication of this mode, of course, is that you can only have a nine-bit offset which is a 2's-complement number. So the biggest value would be 0 1111 1111 = xFF = $2^8$-1, and the smallest (most negative) offset is -$2^8$. Which is still a pretty big chunk of memory, but is certainly not close to all of it.

**Indirect mode**
In order to address all of memory, we need some place to hold an arbitrary 16-bit address, which (as we just discussed) cannot be encoded in our 16-bit instruction.

So, one option is to read the address itself out of memory. That is, we figure out some address, read the data there, and then treat that data as an address, thus giving us access to all of memory. This is known
as indirect addressing, and is a critical part of understanding some aspects of the C programming language (which we’ll get to in a few weeks!).

For instance, consider the instruction:
LDI R0, x50
and let's say that the instruction is located at address x2020.

This instruction will do the following:
- add the already-incremented PC (x2021) to the offset (x50), giving us a value of x2071
- put that value in the MAR, get the data stored at address x2071 (let's say it's x4B01), and put it in the MDR
- rather than moving x4B01 into R0 (as would happen in PC-relative mode), we put it back in the MAR, thus treating it as an address
- when we put x4B01 in the MAR, we read the data held at that address (let's say the data is xFC00), and put it in the MDR
- finally, the value we just read (xFc00) is loaded into the destination register R0

Note that, for the LDI instruction, we make two reads from memory:
- the initial read based on adding the PC and the offset, in this case from address x2071
- the second read based on the result of the first read, in this case address x4B01

The encoding for LDI is similar as it is for LD, except that the opcode is 1010.

There is also a store instruction for indirect mode: in assembly language, it's STI, and the opcode is 1011. Note that STI makes one read from memory (to figure out which address to write to) and then one write (the actual store from the source register).

Base-offset mode
Another way to access all of memory is to hold the address in a register, instead of in memory (as we just saw in indirect mode).

That is, in base-offset mode (or sometimes I call it “register-offset mode”), we have an address held in a register, and then we read from or write to that address. The difference here is that, since we have some bits left over in the instruction, we can add/subtract an offset to the address.

For instance, let's say the value x45A2 is stored in register R3. Then the instruction:
LDR R4, R3, #2
would do the following: “take the value in R3, add the value 2 to it, and treat that as an address (giving us x45A2 + 2 = x45A4); read from that address and load the data from there into the destination register R4”.

Note that the PC is not used at all in base-offset mode. Rather, we just add the offset to whatever value is in the “base register” and use that as our address.

The above instruction would be encoded as follows:
- the opcode is 0110
- bits 11-9 are 100 for the destination register
- bits 8-6 are 011 for the base register
• the remaining six bits represent the offset as a 2's complement number; here the would be 000010

There is also an STR instruction; its opcode is 0111.

How does an address get into a register in the first place? To do that, we could use the “load effective address” instruction like this:
LEA R4, 0x2A
This instruction would take the (already-incremented) PC, add the offset x2A to it, and store that value in R4. Note that, although this looks like PC-offset mode, it is not reading from memory! Rather, it is doing the same calculation (based on the PC and offset), but putting the result right into the destination register, and not into the MAR.

**LC-3 Programs with Memory**
Last time we saw a simple LC-3 program that just used registers. How can we use memory now?

Your program is expected to “allocate” spaces for the memory it is going to use. That is, the program itself should have some indication that there are some addresses that you want to access.

Let's say that we want to write a program that will read values from addresses x3005 and x3006, add them, and put the result in address x3007. We could write it like this:

```assembly
.ORIG x3000
LD R0, 4     ; the offset will be 4 to read from x3005
LD R1, 4     ; here the offset is 4 to read from x3006
ADD R2, R1, R0
ST R2, 3     ; the offset is 3 to write to x3007
HALT

.FILL xA1B0  ; we want this address to hold this data
.FILL xBFF2  ; same here
.FILL x0000  ; we're going to overwrite this anyway
.END
```

There are a few things you probably noticed about this program.

First, the offsets for the first two instructions are the same, even though we're reading from different places. Of course, the reason for that is that the address from which we're reading is relative to the address of the instruction, so in this case, the offsets are the same because we've moved up an instruction.

Even though it's five addresses from the first instruction (which is at address x3000) to the place where it's reading from (x3005), the offset is 4 and not 5 because, when that instruction is executed, the PC has already been incremented to x3001.
The “.FILL” lines at the end are where we're allocating memory. We're just telling the assembler to include these spaces in the assembled program, and to populate them not with the encoding for some instruction, but with the binary representation of the value that we've specified.

So, when this program is assembled and then loaded into memory, here are the values that will be at each address:

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>0010000000000100</td>
<td>this is the encoding for “LD R0, 4”</td>
</tr>
<tr>
<td>x3001</td>
<td>0010001000000100</td>
<td>this is the encoding for “LD R1, 4”</td>
</tr>
<tr>
<td>x3002</td>
<td>0001010001000000</td>
<td>this is the encoding for “ADD R2, R1, R0”</td>
</tr>
<tr>
<td>x3003</td>
<td>0011010000000111</td>
<td>this is the encoding for “ST R2, 3”</td>
</tr>
<tr>
<td>x3004</td>
<td>1111000000010010</td>
<td>this is the encoding for “HALT”</td>
</tr>
<tr>
<td>x3005</td>
<td>1010000110110000</td>
<td>this is the value xA1B0</td>
</tr>
<tr>
<td>x3006</td>
<td>1011111111111100</td>
<td>this is the value xBFF2</td>
</tr>
<tr>
<td>x3007</td>
<td>0000000000000000</td>
<td>this is the value x0000</td>
</tr>
</tbody>
</table>

After the program is loaded, the PC will be set to x3000 and the first instruction will be fetched, decoded, and executed. As a result of decoding, the Control Unit will notice that the opcode is 0010, meaning LD. This means that the last nine bits are the offset. In calculating the address to read from, the address will be PC + 1 + offset = x3000 + 1 + 4 = x3005. That will be put into the MAR, and the data at that address (which, as you can see above, is xA1B0) will be put in the MDR and then loaded into the destination register R0.

Then, on the next instruction, the value at address x3006 (which is xBFF2) will be loaded into R1.

Those values are added by the third instruction, and the result will be put in R2 (you may note that the addition causes overflow; LC-3 will ignore it).

Then, in the fourth instruction, we have a store to address x3003 + 1 + 3 = x3007, and the value at that address (which is x0000 when the program is loaded) is overwritten with the value held in R2. Finally, when we hit the “HALT” instruction, the program stops.

Programs like the one above can be error-prone if we're relying on a human to calculate the offsets. Also, if we insert any code or move it around, then all the offsets would change. So instead of hand-calculating the offsets, we use **labels** for the addresses and let the assembler calculate the offsets.

For instance, we could rewrite the above program like this:

```assembly
.ORIG x3000
LD R0, A     ; read from the address labeled A
LD R1, B     ; read from the address labeled B
ADD R2, R1, R0
```
Note the labels on the three lines representing our data. We could label any address (as we'll see next time), but it makes sense to label these three.

Now, the programmer doesn't have to do any calculation; they can just include the label of the address to read from or write to. So when the assembler is going through and encoding this program, it can calculate the offsets for us.

How does the assembler know what the offsets are? We'll see that next time....