So far we've focused on two parts of the von Neumann architecture: memory and the CPU.

Now we look at the third part: input/output, or I/O.

Some examples of input are keyboard, mouse, network, etc.
Some examples of output are monitor, printer, etc.

For the CPU to read input and write output, there are two options:
  • include special commands in the ISA for performing these operations
  • set aside special locations in memory such that input goes into those locations and output goes out from those locations

The LC-3 uses the latter. This is known as **memory-mapped I/O**.

### Reading Keyboard Input

As a simple example of memory-mapped I/O, let's consider how the LC-3 reads input from the keyboard. Whenever the user strikes a key, its ASCII code is placed into a special location in memory. This location is referred to as the Keyboard Data Register (KBDR) and is at address xFE02.

For example, when the user strikes the 'b' key, the hardware will take its ASCII value (x62) and write it to address xFE02. Now the program just needs to read from that address and it will know which key was struck.

How can a program read from address xFE02? If we use PC-offset addressing, we would have to be within 256 bytes (more or less) of that address, which would severely limit the locations at which our code could run. Of course, to use indirect or base-offset addressing, we'd somehow need to get the full 16-bit value (i.e., xFE02) into some other memory address or register.

For example, we could do something like this:

```
LDI R0, KBDR_ADDR ; read from address xFE02 into R0
...
                   ; rest of program
KBDR_ADDR .FILL xFE02 ; holds the value of the KBDR address
```

Note that “LD R0, KBDR_ADDR” would not really serve our purposes here. That would simply read the value xFE02 into register R0. We don't want that value in R0, we want the value held at that address! So we need to use LDI here, not LD.

Okay, now we know how to read a value from the keyboard. However, there are two practical limitations to consider:
  • what if there is no key to read yet? That is, the instruction “LDI R0, KBDR_ADDR” doesn't *wait* for a key to be struck, it just reads whatever's there, even if it's garbage
on a related note, what if we want to read two keys? Even if we somehow knew that the first key had been struck, if we call “LDI R0, KBDR_ADDR” and then “LDI R1, KBDR_ADDR”, the second read will almost certainly happen before the user has time to enter the second key, and we'd end up reading the same key twice.

So, we need some way of knowing whether a key has been struck. To solve this problem, the LC-3 microarchitecture dictates that address xFE00 be used as the Keyboard Status Register (KBSR). It works as follows:

- when a key is pressed, the highest-order bit (i.e., the most significant bit) of the KBSR is automatically set to 1
- when the KBDR is read, the highest-order bit of the KBSR is automatically cleared to 0

This certainly is very handy! Now our program has a way of waiting until a key has been pressed. To accomplish this, it continuously loops and reads the value in the KBSR; when it sees that the highest-order bit has become 1, it knows that a key has been pressed, so it can move on and read from the KBDR. Like this:

```
POLL LDI R1, KBSR_ADDR   ; read from the KBSR
BRzp POLL           ; loop while the high bit is 0

; ; once we get out of the loop, the high bit must be 1
LDI R0, KBDR_ADDR   ; read from the KBDR
.
.
; ; rest of program

KBDR_ADDR .FILL xFE02         ; value of the KBDR address
KBSR_ADDR .FILL xFE00         ; value of the KBSR address
```

Why does the BR instruction use the Z and P flags? Because if the most significant bit in the KBSR is 0 (meaning that there is no key to read), then either the value stored there is zero or is positive (since 2's complement positive numbers start with 0). So in either of those cases, we continue to loop. However, when a key is struck and the most significant bit in the KBSR goes to 1, then the value there would be considered negative (since 2's complement negative numbers start with 1) and we fall out of the loop and read the KBDR.

This approach is known as polling. It means that the program constantly is checking (“polling”) the value in the KBSR and will keep doing that forever unless someone strikes a key. It is fairly wasteful, since the program spends a lot of time waiting and essentially doing nothing useful, but it is very easy to implement both in hardware and in software.

**Monitor Output**

We've already seen one example of memory-mapped output: the video output in the LC-3 simulator uses addresses xC000 through xFDFF to represent the colors of the pixels in the screen.

LC-3 also allows you to print ASCII characters to a separate display by writing the character's value to
address xFE06, which is known as the Display Data Register (DDR).

For instance, this program would write the letter 'k' to the display:

```
LD R0, K            ; reads the ASCII value into R0
STI R0, DDR_ADDR    ; write from R0 to address xFE06
. . .               ; rest of program

DDR_ADDR .FILL xFE06 ; holds the value of the DDR address
K .FILL x006B        ; ASCII value of the letter 'k'
```

Just as reading keyboard input requires waiting for a new key to arrive, we cannot simply write ASCII values to the display without waiting for it to be ready, since it takes much longer for the monitor to “draw” the character than it does for the CPU to execute instructions.

So, in LC-3, address xFE04 is used as the Display Data Register (DSR). As with the KBSR, the highest-order bit of the DSR will be set to 1 to indicate that the display is ready for the next character, and will be 0 while it's drawing the character (indicating that it's not ready for another).

So to write a single character to the display, we'd need to modify the above program like this:

```
LD R0, K            ; reads the ASCII value into R0
POLL LDI R1, DSR_ADDR ; read the DSR
BRzp POLL           ; if not ready, keep looping
STI R0, DDR_ADDR    ; write from R0 to address xFE06
. . .               ; rest of program

DDR_ADDR .FILL xFE06 ; value of DDR address
DSR_ADDR .FILL xFE04 ; value of DSR address
K .FILL x006B        ; ASCII value of the letter 'k'
```

Strings

In LC-3, as in most other programming language, a string is a collection of characters. For instance, “dog”, “ab123cd”, and “hello world” are all strings. And a string can be represented using the ASCII values for each character.

Additionally, in LC-3 (and in most other programming languages), the end of the string is represented using the ASCII value for “null”, which is zero. Thus, we say that LC-3 strings are null-terminated because the last character will be zero. This will come in very handy later on when we learn C.

Let's see an LC-3 program that would display an entire string in the display. This program is kind of long but we'll explain it below (note that line numbers are not part of the program, of course):
.ORIG x3000

1    LEA R0, START    ; put addr of first char in R0
2    LDR R3, R0, #0   ; put first char in R3
3    POLL LDI R1, DSR_ADDR   ; wait for display to be ready
4    BRzp POLL
5    STI R3, DDR_ADDR   ; write char to display
6    ADD R0, R0, #1     ; update R0 to addr of next char
7    LDR R3, R0, #0     ; read next char
8    BRnp POLL         ; if not zero, keep going
9    HALT

10   START .STRINGZ  "hello"
11   DSR_ADDR .FILL     xFE04
12   DDR_ADDR .FILL     xFE06

.END

Note on line 10 that we have the string “hello” in our program, indicated by the .STRINGZ pseudo-op. This is sort of like .FILL, except that it populates more than one address (whereas .FILL only fills one spot). With .STRINGZ, the ASCII value of each character in the string will be placed into subsequent memory addresses, followed by a zero, indicating the end of our null-terminated string. Note here that the label START represents the address of the first character in the string, in this case 'h'.

Now let's go through the program:

On line 1, we read the address of the first character of the string and put it in R0. Why not just read the first character using LD? That will become apparent later on. So at this point, R0 will hold the value x3009, which is the address of the character 'h'.

On line 2, we use base-offset mode and read from address x3009, which holds the ASCII value for the character 'h' (that value is x0068).

On lines 3-4, we wait for the display to be ready, as described above.

On line 5, we write the value in R3 (which is x0068) to the DDR, and the character 'h' appears in the display.

On line 6, we increment the value in R0. Previously it was x3009, i.e. the address of the first character in the string. Now it becomes x300A, which is the address of the second character of the string, 'e'.

On line 7, we read from that address, and the ASCII value of 'e' (which is x0065) is placed into R3.
On line 8, we have a branch statement. If the N or P flag is set, we loop back up to line 3 to wait for the monitor to be ready so that we can display the letter 'e' (which is now in R3 because of line 7).

However, if the Z flag is set when we get to line 8, then we don’t loop. Why? Because that means the value we read on line 7 is zero, which is the ASCII value for null, indicating the end of our string.

If you step through lines 3-8 a few more times, you will see that R0 holds the address of the most recent character read, and R3 holds its ASCII value, and that we continue until we hit the null at the end of the string.

So going back to our first question, why didn’t we just use “LD R3, START” at the beginning of our program instead of lines 1-2? Because although that would accomplish the same thing, we need some way of reading from address START+1, START+2, etc. and thus we need that address (x3009) in one of the registers so that we can increment it and then read from that location.

Below is the encoding of this program, along with the addresses at which each instruction would be stored. As described last time, the offsets are calculated by the assembler using the symbol table. Note that the address labeled DSR_ADDR is not the one right after the address labeled START. That's because START refers to the address holding the first character of the string, but subsequent addresses will hold all the other characters (and the null) before getting to the next piece of data.

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>1110000000010000</td>
<td>LEA R0, START</td>
</tr>
<tr>
<td>x3001</td>
<td>0110011000000000</td>
<td>LDR R3, R0, #0</td>
</tr>
<tr>
<td>x3002</td>
<td>101001000001100</td>
<td>LDI R1, DSR_ADDR – note the offset of DSR_ADDR</td>
</tr>
<tr>
<td>x3003</td>
<td>0001111111111110</td>
<td>BRzp POLL</td>
</tr>
<tr>
<td>x3004</td>
<td>1011011000001011</td>
<td>STI R3, DDR_ADDR – note the offset of DDR_ADDR</td>
</tr>
<tr>
<td>x3005</td>
<td>0001000000100001</td>
<td>ADD R0, R0, #1</td>
</tr>
<tr>
<td>x3006</td>
<td>0110011000000000</td>
<td>LDR R3, R0, #0</td>
</tr>
<tr>
<td>x3007</td>
<td>000010111111101</td>
<td>BRnp POLL</td>
</tr>
<tr>
<td>x3008</td>
<td>1111000000100101</td>
<td>HALT</td>
</tr>
<tr>
<td>x3009</td>
<td>0000000001101000</td>
<td>ASCII value 'h'</td>
</tr>
<tr>
<td>x300A</td>
<td>0000000001100101</td>
<td>ASCII value 'e'</td>
</tr>
<tr>
<td>x300B</td>
<td>0000000001101100</td>
<td>ASCII value 'l'</td>
</tr>
<tr>
<td>x300C</td>
<td>0000000001101100</td>
<td>ASCII value 'l'</td>
</tr>
<tr>
<td>x300D</td>
<td>0000000001101111</td>
<td>ASCII value 'o'</td>
</tr>
<tr>
<td>x300E</td>
<td>0000000000000000</td>
<td>ASCII value null</td>
</tr>
<tr>
<td>x300F</td>
<td>1111111000000010</td>
<td>xFE04 for DSR_ADDR</td>
</tr>
<tr>
<td>x3010</td>
<td>1111111000000010</td>
<td>xFE06 for DDR_ADDR</td>
</tr>
</tbody>
</table>