Lecture 13: Memory continued

This lecture describes a one-bit memory with a load signal, and uses it to construct regusters and a simple computer memory. It also begins the description of the simulated ICS-201a machine.

A one-bit memory

Last lecture we saw that a sequential circuit containing two NAND gates can store one bit. We now add two more NAND gates to get a one-bit memory with a load signal.

```
s ---------|
           |NAND>-------|
       ----|            |NAND>-*------- q
       |              *-|      |
       |              |        |
       |              ---------|---
c -----*                       |  |
       |            ------------  |
       |            |             |
       |            ----|         |
       ----|            |NAND>----*---- q'
           |NAND>-------|
r ---------|
```

Note that the original sequential configuration of two NAND gates is reproduced in the righthand two NAND gates. The inputs are denoted s (for set), r (for reset), and c (for clock), which is the load signal. The outputs are denoted q and q' (because the way we will use this circuit, q' will always be the complement of q.) In this case, we AVOID the situation in which $s = r = 0$.

To figure out what this circuit will do, we consider two cases. In the case that the the load signal $c = 0$. It is a property of NAND that if one of its inputs is 0, its output is 0, no matter what the other input is. In this case, the outputs of the lefthand two NAND gates will both be 1. ¿From last lecture, we know that this means that the state of the righthand two NAND gates will remain what it was, either $q = 1$ and $q' = 0$, or $q = 0$ and $q' = 1$.

In the case that $c = 1$, if we have $s = 1$ and $r = 0$, then the signals on the outputs of the two lefthand NAND gates will be 0 and 1, and the outputs will be $q = 1$ and $q' = 0$. When $c = 1$ and $s = 0$ and $r = 1$, we have the opposite situation, and $q = 0$ and $q' = 1$.

Thus, we can consider the value of q to be a one-bit memory. To set the bit to 1, we set $s = 1$ and $r = 0$ and then set $c = 1$. When c goes back to 0, the value of q will remain 1 regardless of the values of s and r. To reset the bit to 0, we set $s = 0$ and $r = 1$ and then set $c = 0$. When c goes back to 0, the value of q will remain 0.

At a higher level, we abstract this circuit as:

```
     s
     |
   -----
   |   |
c--|   |
   -----
     |
     q
```

Here, we only show the s and c inputs (we assume the r input is set to s'), and we only show the q output. This represents a one-bit memory: to copy the value of s into the memory, we set $c = 1$ and then set c back to 0. Then the value of q does not change until the next time c is set to 1.

Registers

A register to store several bits may be constructed of several of these one-bit memories in parallel, with a common c input, thus:

```
       x3       x2       x1       x0
       |        |        |        |
     -----    -----    -----    -----
     |   |    |   |    |   |    |   |
c -*-|   |  --|   |  --|   |  --|   |
   | -----  | -----  | -----  | -----
   |   |    |   |    |   |    |   |
   ---------*--------*--------*   |
       |        |        |        |
       y3       y2       y1       y0
```

In this circuit, all four one-bit memories share a common load signal c. When $c = 1$, the values on the inputs x_0, x_1, x_2, x_3 are copied into the corresponding one-bit memories. The output values, y_0, y_1, y_2, y_3, are equal to the values being stored, and remain so after c is set to 0. While $c = 0$, the x inputs may change without changing what is stored in the four one-bit memories (and displayed on the output y lines.)

This, in turn, can be abstracted as a four-bit register:

```
       x3       x2       x1       x0
       |        |        |        |
     --------------------------------
     |                              |
c ---|                              |
     --------------------------------
       |        |        |        |
       y3       y2       y1       y0
```

The behavior of this register is that when c is set to 1, the values on the x lines are copied to the four bits of memory and displayed on the output y lines. When $c = 0$, the bits of memory and the output lines remain unchanged, regardless of changes in the input x values. Such a register can store $2^4 = 16$ possible bit patterns. There is an obvious generalization to construct a register with any fixed number of bits.

A simple memory

In turn, we may consider a collection of registers, and some control logic to access values in the registers, as a memory. For the simulated machine of homework 4, we assume that we have $2^{12} = 4096$ registers, each capable of storing 16 bits. We picture this collection of registers as a sequence of boxes called words, where each word has an address (from 0 to 4095) and is capable of holding a pattern of 16 bits:

```
address   contents

  0       0000 0000 0000 0010
  1       1000 0000 0000 0111
  2       0101 1010 0110 1001
  3       1111 0000 1111 1111
  .                 .
  .                 .
  .                 .
  4095    1110 1101 1011 0111
```

In this picture, the memory word with address 2 has as its contents the bit pattern `0101 1010 0110 1001`. (The groups of 4 bits are separated for human readability.)

In addition to the memory words and their contents, there are two more registers, the memory address register (MAR) and the memory data register (MDR). The MAR is able to hold any address as an unsigned binary number, so in our example it is a register capable of holding 12 bits. The MDR is able to hold the contents of a memory word, so in our example it is a 16-bit register.

The memory control logic implements fetch and store to the memory. When we issue a fetch signal, the MAR is interpreted as an unsigned binary number giving the address of a word, and the content of that memory word are copied to the MDR. For example, if the memory is as above and the MAR and MDR contain:

```
  MAR                 MDR
  0000 0000 0011      0011 0010 1101 0001
```

Then a fetch signal will cause the contents of memory word 3 to be copied to the MDR:

```
  MAR                 MDR
  0000 0000 0011      1111 0000 1111 1111
```

The contents of the MAR and the words of memory are unchanged by a fetch operation.

When we issue a store signal, the MAR is again interpreted as an unsigned binary number giving the address of a word of the memory, and the contents of the MDR are copied into that word of the memory, overwriting the previous contents. For example, if the contents of the memory are as above and MAR and MDR contain:

```
  MAR                 MDR
  0000 0000 0011      0011 0010 1101 0001
```

Then a store signal will cause the contents of memory word 3 to be replaced by the contents of the MDR. The memory will now contain:

```
address   contents

  0       0000 0000 0000 0010
  1       1000 0000 0000 0111
  2       0101 1010 0110 1001
  3       0011 0010 1101 0001
  .                 .
  .                 .
  .                 .
  4095    1110 1101 1011 0111
```

The contents of the MAR and MDR are unchanged by the store. Your textbook goes into more detail about how the memory access logic works. At this point, we are somewhat in the situation of Eeyore with the "useful pot" – able to put something into it, and take it back out again, but we'd like to be able to manipulate data values in more complicated ways.