

## Lecture 16: Memory and a simple hardware design

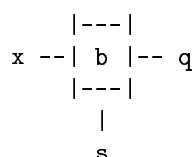
This lecture describes devices that exhibit memory. A simple example illustrates some of the issues in hardware design.

### Functions have no memory

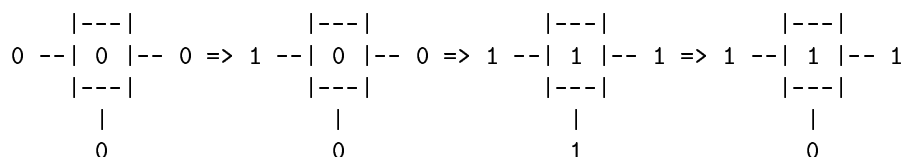
So far we have been describing circuits that compute Boolean functions. The circuits we have considered are combinational circuits, in which there is no path of wires and gates from the output of a gate back to its input. Such circuits are also called feedforward, in contrast to the feedback created by loops.

The output of a combinational circuit is strictly a function of the current inputs; it has no “memory” of past inputs. In addition, we need devices that “remember” or “store” a value for us, (somewhat) independently of the current values of the inputs.

In particular, we’d like a 1-bit memory that functions as follows. It has two inputs, which we call  $x$  and  $s$ , and one output  $q$ , together with some internal state  $b$  that stores a 0 or a 1. The input  $s$  functions as a “set” command; when  $s = 0$ , the output  $q$  is equal to the stored bit, regardless of the value of  $x$ . When  $s = 1$ , the value of  $x$  is copied into the internal state, and also appears on the output  $q$ . Our 1-bit memory element will be pictured as follows.



Some of its behavior can be illustrated by the following sequence of changes to its inputs.



In the first situation, the internal state is 0 and the output is 0 and two inputs are 0. Then the  $x$  input is changed to 1, and there is no change in the state or output. Then the  $s$  input is changed to 1, and the state and output are changed to the value of  $x$ , that is 1. Finally, the  $s$  input is changed to 0, and the state and output remain 1. Comparing the second with the fourth situations, the  $x$  input is 1 and the  $s$  input is 0 in both cases, but the outputs are different. This illustrates that this circuit is not simply a function of its current inputs ( $x$  and  $s$ ), but “remembers” the value of  $x$  at the last time  $s$  was 1.

### What is core memory?

A number of technologies have been used to implement memory. One that was quite important in the 1960’s and 1970’s and isn’t much used anymore is core memory. When the random access memory of a computer is referred to as “core,” it reflects the former dominance of this memory technology. (This usage is preserved in the name “core” for the annoying large file created in your directory when an application such as scm

crashes; it is short for “core dump,” a snapshot of the contents of random access memory intended to help you track down the causes of the mysterious crash. Free free to delete the file, especially since it may eat up a lot of your file quota.)

The basic physical element is a little ferrite core (doughnut-shape) with an electric wire passing through it. When sufficient electric current is passed through the core in one direction, the core is magnetized in one direction, and retains that magnetization until sufficient electric current is passed through the core in the other direction, magnetizing it in the other direction. This behavior allows us to store a value of 0 or 1 by which way the core is magnetized, but a write-only memory is not much use; we also need a way to read the value.

For that purpose, another wire (the sense wire) is threaded through the core, and then sufficient current is passed through the core on the set wire in the 0 direction to write it. If the core was holding the 1 value, then a pulse can be detected on the sense wire induced by the change in the magnetic field. If the core was holding the 0 value, the magnetic field does not change, and no pulse is detected.

Thus, the 0 or 1 value held in the core can be read, at the cost of forcing the core to the 0 state, and losing the previous information it held. This is an example of a “destructive read.” To overcome this, the memory circuitry was designed to write back the value read, thus implementing a “nondestructive read.”

A private set wire for every bit of memory (the sense wire could be shared) was not a practical design, so most accounts you’ll see of core memory emphasize that the cores were arranged in rectangular arrays, with two set wires (one horizontal and one vertical) passing through each core. Then by sending half the “sufficient” current through a vertical wire and half through a horizontal wire only the single core through which both wires passed would be affected by the set signal. For example, in a 32 by 32 array of 1024 bits, this idea cuts the number of different set wires from 1024 (one per bit) to 64 (32 horizontal, 32 vertical.)

## Sequential circuits and memory

Sequential circuits have feedback loops; there is a path of wires from the output of some gate back to the input of that gate. Here is a simple example:

```

|-- |NOT>-- |
|         |
|-----|

```

What is this supposed to mean? The picture represents a NOT gate with its output fed back as its input. It is not clear how to reason about this configuration; if the input is 0, then the output is 1, so the input is 1 and the output is 0, so the input is 0, etc. It seems to be contradictory: the output is 0 if and only if the output is 1. The physical reality is that there is a time delay before the output changes, so this kind of configuration will exhibit an oscillatory behavior, with the value on the wire changing rapidly between 0 and 1. This kind of behavior is in fact useful, as it provides a regularly spaced timing signal that can be used to control the time-order of events in hardware.

Here is another simple example:

```

x--- |
    |OR>-- |
|-- |     |
|-----|

```

This is intended to depict an OR gate with one input  $x$  and the output fed back as the other input to the OR gate. Suppose we start with  $x = 0$  and the output of the OR gate 0. This is a consistent and stable state of the circuit, because the value computed by the OR gate is 0 when its two inputs are 0. Now, if we set  $x = 1$ , the output will change to 1 and this is also a consistent and stable state of the circuit, since both inputs to the OR gate are 1 and the output is 1. However, if we now set  $x = 0$ , the output of the circuit remains 1, because the inputs to the OR are 1 and 0, so the output is 1, another stable state of the circuit. Clearly, the output remains 1 indefinitely, independent of the value of  $x$ . This circuit exhibits a rudimentary kind of memory; it remembers forever whether the input  $x$  has ever been set to 1.

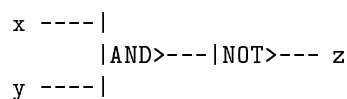
To implement the 1-bit memory element we seek, we'll need a more complicated sequential circuit. First, we'll take a slight detour on the NAND and NOR functions.

## NAND and NOR

The two input Boolean function NAND is a combination of NOT and AND. Its truth table is

| x | y | (x NAND y) |
|---|---|------------|
| 0 | 0 | 1          |
| 0 | 1 | 1          |
| 1 | 0 | 1          |
| 1 | 1 | 0          |

You can check that a Boolean expression for this function is  $(x \cdot y)'$ , and a circuit implementing it is the following.

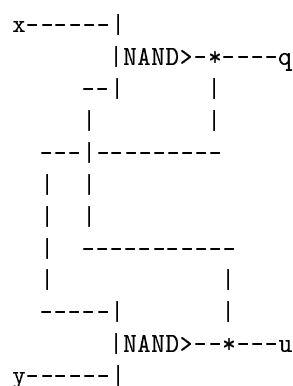


Various transistor technologies make it very convenient to implement NAND as a primitive operation. Its gate symbol is similar to the AND gate, with a little circle (signifying negation) at the start of the output wire. Thus implementing other Boolean functions purely out of NAND gates became a design goal. This is possible because, all by itself, the NAND gate is a complete Boolean basis. In particular,  $(x \text{ NAND } x)$  is the NOT function, so  $((x \text{ NAND } y) \text{ NAND } (x \text{ NAND } y))$  is the AND function, and we have previously seen that NOT and AND are a complete Boolean basis.

We should suspect a dual result for OR. The corresponding function is NOR, for NOT-OR, which can be represented by the Boolean expression  $(x + y)'$ . It also is a complete Boolean basis, and has a gate symbol similar to that for OR, with a little circle on the output wire. NOR is also called Sheffer's stroke.

## A more complex sequential circuit

We now consider two NAND gates, with the output of each fed back to be one input of the other. In ASCII diagram, we have the following.



The inputs to this circuit are  $x$  and  $y$ , and the outputs are  $q$  and  $u$ . There are loops in this circuit – if we trace the output wire of the first NAND gate around to the input of the second NAND gate, to the output of the second NAND gate, to the input of the first NAND gate, to the output of the first NAND gate, we are back where we started: a loop. Thus, this circuit is sequential.

One way to figure out what is happening in this circuit is to consider the equations:

$$q = (x \cdot u)'$$

and

$$u = (y \cdot q)'$$

A solution of these equations consists of an assignment of 0's and 1's to the variables  $x$ ,  $y$ ,  $q$ , and  $u$  such that both equations are true; this represents a possible stable configuration of the circuit. There are 16 possible combinations to test, and we find that there are just 5 combinations that give a solution for both equations:

| x | y | q | u |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |

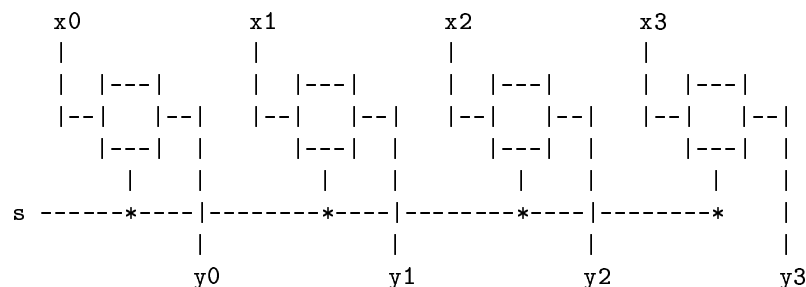
The last two lines show that neither  $q$  nor  $u$  is a function of the values of  $x$  and  $y$ , because we can have  $x = y = 1$  and  $q = 1$  or  $q = 0$ , and similarly for  $u$ . The values of  $q$  and  $u$  are not uniquely determined by the values of  $x$  and  $y$ ; the circuit has state, or memory.

We exploit the state to store information as follows. We AVOID setting  $x$  and  $y$  to 0 simultaneously, and thereby avoid the first line of the table. Then there are two states of the circuit: where  $q = 1$  and  $u = 0$ , and where  $q = 0$  and  $u = 1$ . If the inputs are  $x = 1$  and  $y = 1$ , the circuit stays in whichever state it is currently in. If the inputs are  $x = 0$  and  $y = 1$ , the circuit goes from whichever state it is currently in to the state where  $q = 1$  and  $u = 0$ . If the inputs are  $x = 1$  and  $y = 0$ , the circuit goes from its current state to the state where  $q = 0$  and  $u = 1$ .

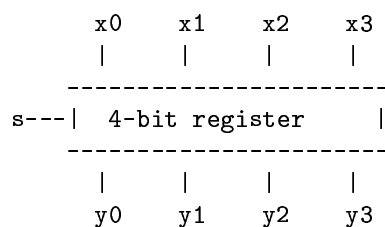
This use of the circuit allows us to store one bit of information. That is,  $q = 1$  signifies that the last time the inputs were not both 1, it was  $x$  that was 0. Correspondingly,  $q = 0$  signifies that the last time the inputs were not both 1, it was  $y$  that was 0. A further elaboration of this circuit gives an implementation of our desired 1-bit memory. Rather than pursue that further elaboration, we turn to possible uses of the 1-bit element.

## Registers

A register is a device that can store some fixed number of bits in memory. For example, suppose we connect four 1-bit memories with a common set line as follows.



Then as long as  $s$  is 0, the outputs  $y_0$ ,  $y_1$ ,  $y_2$ , and  $y_3$  are equal to the bits stored in the four 1-bit memories, respectively. When  $s$  is set to 1, the values of the inputs  $x_0$ ,  $x_1$ ,  $x_2$ , and  $x_3$  are copied into the corresponding 1-bit memories and also appear on the corresponding output wires. This allows us to read and set the four bits in the register. Larger or smaller registers can be constructed by similarly connected larger or smaller numbers of 1-bit memories. We abstract this circuit and picture it as a rectangle.



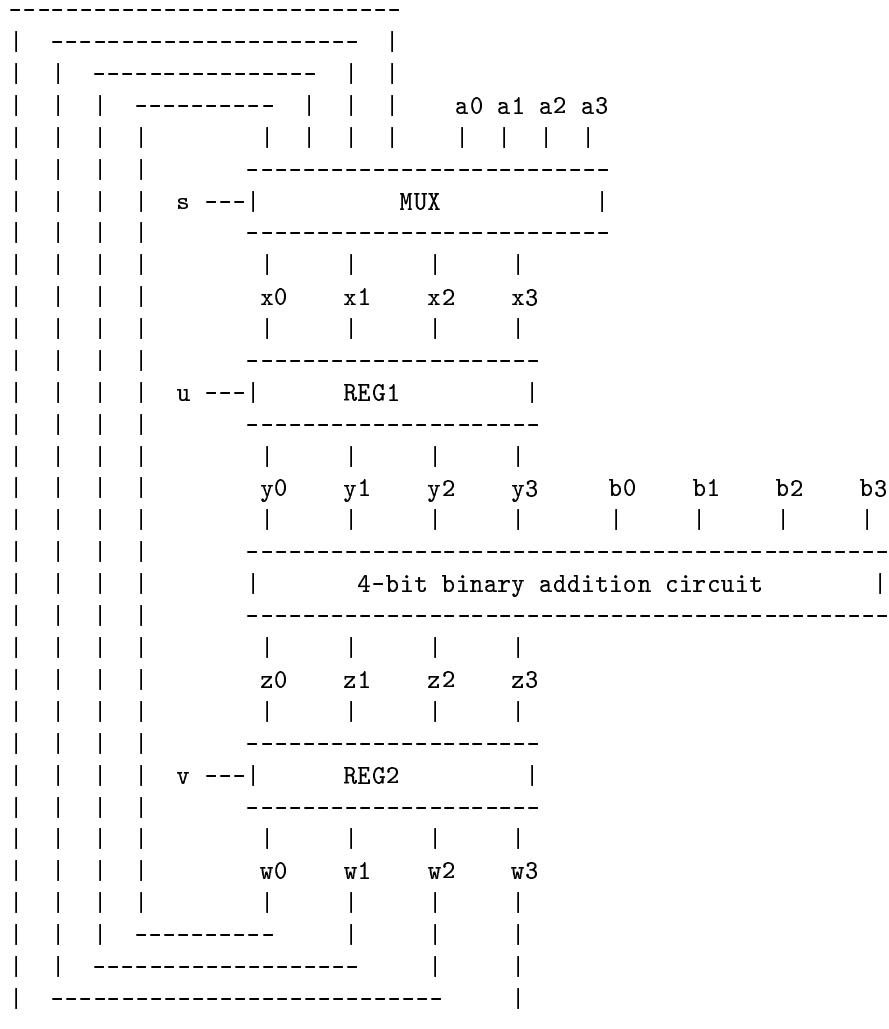
Note that both registers and combinational circuits are depicted with rectangles, so some context is necessary to decide which kind of circuit is intended.

## A small circuit with registers

To illustrate some of the issues and ideas that come up in the design of hardware using both combinational and sequential circuits, we look at the design of a small circuit incorporating two registers, a multiplexor, and a binary addition circuit.

The circuit will have two 4-bit quantities and three single bits as inputs, and one four bit quantity as output. We will imagine the inputs provided by us as switches that can be toggled between 0 and 1, and the output presented to us as 4 small lamps that may be lighted or not. We will describe operating the circuit by means of the switches and lamps in order to suggest the control signals that must be generated by a computer's central processing unit to control a much more complicated circuit.

The circuit is as follows.



Here the inputs are the two 4-bit quantities  $a_0, a_1, a_2, a_3$ , and  $b_0, b_1, b_2, b_3$ , and the three 1-bit quantities,  $s, u$ , and  $v$ . The other variables are labels on wires so that we can talk about the values on them. We imagine that there are 4 lamps controlled by the values on the wires  $w_0, w_1, w_2$ , and  $w_3$  so we could directly observe those values.

The rectangle labelled MUX is a multiplexor that selects, based on the value of  $s$ , which of the two 4-bit quantities,  $w_0, w_1, w_2, w_3$  or  $a_0, a_1, a_2, a_3$ , will appear on the output wires  $x_0, x_1, x_2, x_3$ . This circuit can be constructed out of 4 copies of the 1-bit multiplexor we saw previously, with a common select input. The rectangles labelled REG1 and REG2 are 4-bit registers, as described above, where  $u$  is the set input for REG1 and  $v$  is the set input for REG2. The rectangle labelled as a 4-bit binary addition is a combinational circuit to perform addition of two 4-bit binary numbers, as we also saw previously. Note that the circuit has feedback, in that the outputs of register REG2 can be traced back in a loop to its inputs.

To describe the operation of this circuit, imagine that we start with the value 0000 in both registers and all the inputs set to 0. Then every wire will have a 0 value. Now suppose we set the  $b$  inputs to the pattern 0010, signifying the binary number 2. The binary addition circuit will add the 0000 in REG1 to the 0010

of the  $b$  inputs to produce 0010 on the  $z$  wires. However, since the set input,  $v$ , for register REG2 is still 0, these values are not loaded into the register, which retains its 0000 contents.

Now suppose we set the  $a$  inputs to the pattern 0101, or 5 in binary. Nothing else changes, because the selector input to the multiplexor MUX is still 0, selecting the  $w$  values. If we now set  $s$  to 1, then the  $x$  outputs become equal to the  $a$  inputs, that is, 0101. Nothing changes in register REG1, however, because its set input,  $u$ , is still 0.

If we change the value of  $u$  to 1, then the values on the  $x$  wires are copied into REG1, and the  $y$  outputs become 0101. At this point, the outputs of the binary addition circuit change to be the sum of the two input quantities (0101 and 0010), so the  $z$  values become 0111, representing 7 in binary. The values in REG2 do not change, however, because its set input,  $v$ , is still 0.

If we set  $v$  to 1, then these values will be loaded into REG2, and the pattern 0111 will appear on the wires  $w$ , which we are assuming connected to lamps that will show us this pattern. Because of the feedback, this changes the values input to the multiplexor MUX, but the outputs do not change because the selector  $s$  is 1, selecting the  $a$  inputs.

Now we would like to repeat this sequence of operations to add 0010 to the result 0111. However, we cannot just change  $s$  to 0, because the values would chase each other endlessly around the loop. To avoid this, we first change  $v$  to 0, which means that the output quantity 0111 will be held in REG2 until we change  $v$  back to 1. Then we change  $s$  to 0, which will cause the  $x$  values to change to 0111, and these values will be copied into REG1 (because  $u$  is still 1), appear on the wires  $y$ , causing the output of the addition circuit to become 1001, or 9 in binary. This is not copied into REG2 because we have set  $v$  to 0.

Before we set  $v$  to 0, we set  $u$  to 0, to hold the value 0111 in REG1. Then we can safely set  $v$  to 1, which copies the values 1001 into the register REG2. These values then appear on the wires  $w$ , the output lamps, and on the outputs  $x$  of the multiplexor. However, they are not copied into REG1, because  $u$  is 0.

This is indeed a pretty tedious way to compute the successive odd numbers starting with 5, but it illustrates the ideas in hardware of designing sequences of control signals to keep “too much” from happening at once.