Lab – Digital Circuits

1. Start Firefox, go to the Hello World web page, choose *Links*, and click on the link Logic circuit simulator from *The Analytical Engine*. (Click *Trust* if your browser asks you whether to trust the web site.)

2. Experiment with the simulator until you're comfortable with how it works (click the Help link for instructions). Test out the behavior of each logic gate by connecting switches to the gate's input terminals and a light bulb to the gate's output terminal. Verify the truth tables for AND, OR, XOR, and NOT, shown below:

a	***b***	***a* AND *b***	***a* OR *b***	***a* XOR *b***
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

a	**NOT *a***
0	1
1	0

3. Build a circuit that behaves like NOR *without* using the NOR gate provided by the simulator. Since NOR is just the negation of OR, all you need is an OR gate followed by a NOT gate. Also, build a circuit for NAND, which is just the negation of AND. The truth tables for NOR and NAND are shown below. Verify that your circuits work according to the table.

a	***b***	***a* NOR *b***	***a* NAND *b***
0	0	1	1
0	1	0	1
1	0	0	1
1	1	0	0

4. Build a circuit that corresponds to the logical expression **(*a* AND *b*) OR (NOT *b*)**. To do this, first connect an AND gate to two switches (representing the circuit's inputs *a* and *b*). Next, connect a NOT gate to the *b* switch. Next, connect the output of the AND gate and the output of the NOT gate to an OR gate. Finally, connect the output of the OR gate to a light bulb, which will serve as the output of the circuit. What is the output of this circuit for each possible combination of *a* and *b* input values? Fill in the truth table below with the appropriate output values.

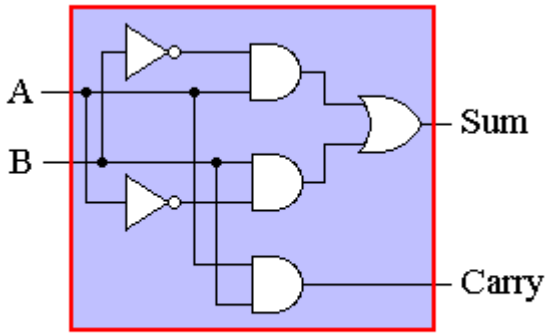
a	***b***	**(*a* AND *b*) OR (NOT *b*)**
0	0	
0	1	
1	0	
1	1	

5. Next, try building an XOR circuit. Remember that for inputs ***a*** and ***b***, the XOR function can be written in terms of AND, OR, and NOT as **[*a* AND (NOT *b*)] OR [(NOT *a*) AND *b*]** . Test your circuit to make sure its output matches the XOR truth table shown in Part 2 for all possible combinations of ***a*** and ***b*** values.

6. An important capability for a computer is to be able to compare two values to see if they are equal. Notice that the XOR circuit does this in reverse. Namely, it gives 1 when the two input values are different. Add an extra NOT gate to your XOR circuit to make a *Comparator* circuit, which returns 1 when its two inputs are identical.

7. Another way of implementing *Comparator* is as follows: **[*a* AND *b*] OR [(NOT *a*) AND (NOT *b*)]**
Build this circuit and verify that it gives the same output as your XOR-based circuit from Part 6.

8. Now try building a "half-adder" circuit, which takes as input the two binary digits you are adding and produces two output values: the result of the "ones" column (*i.e.*, the sum without the carry), and the result of the "twos" column (*i.e.*, the carry value). A diagram of a half-adder circuit is shown below. Convince yourself that the five gates in the upper portion of the diagram (the two NOTs, the upper two ANDs, and the OR) together are equivalent to a single XOR gate, as described earlier. Thus you can save yourself some work by building a half-adder circuit out of just an XOR gate and an AND gate, instead of using all six gates shown in the diagram. Once you've built your half-adder, test out your circuit to make sure it works.



1-bit Half Adder circuit

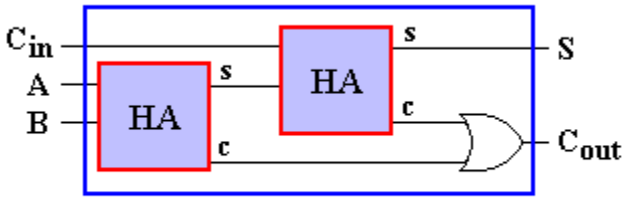
9. What a computer requires is not a half-adder but a *full-adder*, which is a half-adder that's been given the capacity to handle a possible carry-in from earlier columns in the addition. To handle a carry, you first do the addition problem without the carry, then consider whether adding in the carry will force another carry into the left column. For example, say we're adding up the "twos" column of the following binary addition problem:

```
  111
+ 101
```

1+0 gives us 1, so there's no carry, but then we add our answer (1) to the carry from the "ones" column, giving us 1+1, which will indeed force us to carry into the "fours" column to the left.

So, to build a full-adder, we use two half-adders. The first half-adder adds the two digits we are adding in the current column. The purpose of the second adder is to add the carry from the previous column to the sum output from the first half-adder. If either the first or the second half-adder has a carry, then the carry output from the full-adder will be 1.

See if you can build a full-adder circuit in the simulator. A circuit diagram for a full-adder is shown below. In the diagram, each HA component corresponds to a complete half-adder circuit.



1-bit Full Adder circuit

10. Now let's test out a simple multiplexor circuit. First you'll need to download the circuits in *Notes->week08* from the class web page. Save these files on your Desktop. Then, in the simulator, click *Open,* navigate to the Desktop, and choose **2_way_MUX_circuit**. The two switches on the left are the multiplexor's inputs; the switch at the bottom selects one of these two inputs as the output of the circuit. With the select switch set to 1, the top input is selected; with select = 0, the bottom input is selected. Test out this circuit until you understand how it works. Then, on a piece of paper, draw the truth table for this circuit. Label the columns of your table ***a*** (for the top input), ***b*** (for the bottom input), ***select*** (for the selector switch), and ***out*** for the output signal.

11. Now load in the **latch_circuit**. When the bottom switch (the *set* signal) is 1, the output of the latch is set to the value of the top switch (the *data* input). Flipping the bottom switch to 0 "locks in" the current output value of the circuit. When *set* = 0, flipping the *data* switch has no effect on the circuit's output, so this circuit in effect "remembers" its data value. Note also that when first activated, the circuit is unstable until the *set* switch is turned on. Test out this circuit until you understand how it works.

12. Finally, load in the **1_bit_memory_circuit** and test it out. The top two switches act just like the *data* and *set* switches in the latch circuit (in fact, the memory circuit *is* a latch circuit with a couple of extra AND gates and an extra switch thrown in). However, instead of calling the second switch *set*, we refer to it as *read/write*. *Read* mode corresponds to position 0 of the switch, and *write* mode corresponds to position 1. In *read* mode, all we can do is observe the output of the memory circuit; toggling the *data* switch has no effect. In *write* mode, the memory circuit is set to the value of the *data* switch. Finally, the bottom switch is a master "on/off" switch for the whole circuit. Turning off this switch deactivates the memory circuit, so that neither toggling *data* nor *read/write* has any effect. However, the memory still remembers its data value, even when it is inactive. Test out this circuit until you understand how it works.

13. Implement a majority-rules circuit (see problem 19 on page 185 of your textbook).

14. Implement an odd-parity circuit (see problem 20 on page 185).

15. Implement a 4-way multiplexor circuit (see problem 23 on page 185).

16. Implement a 3-to-8 decoder circuit (see problem 24 on page 185).