14/8/2018 Lesson 8: (Term 3, Week 4)

Review of full adder and multi-bit adder.

Installing build-essential (including gcc) with apt-get

“Hello World”program in C:

```
#include <stdio.h>
int main()
{
	printf(“Hello World.”);
}
```

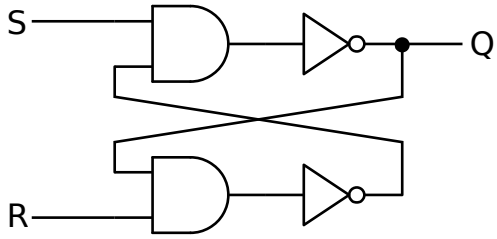
21/8/2018 Lesson 9: (Term 3, Week 5)

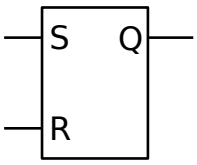
In past weeks, we have learned how computers do arithmetic: adding numbers together. Today we start to learn how computers remember things.

Sequential Logic and the RS Flip-Flop

All the circuits you have learned about so far (gates, adders, ...) are what we call **combinatorial logic**. This means that their output only depends on what the inputs are right now. The past does not affect them. In a circuit diagram, it means bits only go forwards, from the inputs to the outputs. (Draw a few circuits such as gates, XOR gate and full-adder. Show how the inputs go to the outputs, with no looping back.)

There is a second we way we can wire gates together, called **sequential logic**. In sequential logic, we wire some of the outputs back to the inputs. An example of a sequential circuit is shown below:





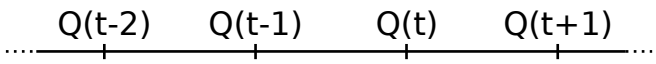
(*Draw an example RS flip-flop. When drawing the circuit, put the feedback paths in last. Draw the feedback path to input of the AND gate connected to the S input and explain how the circuit is still combinatorial, because it does not contain any loops. Draw the second feedback path, from the Q output, and explain how the circuit is now sequential, as we can trace a loop from the output to the input and keep looping around for ever.*)

In sequential logic circuits, the outputs don't just depend on the input as they are right now: they also depend on the inputs as they were in the past. Look at the circuit I drew. It has two inputs (R and S) and an output (Q). Let's look at its truth table. (*Draw blank table and fill it in with the students.*)

For the first three lines of the truth table, trace the logic though the diagram and fill in the output column. For last line trace truth table, as the question “What's the output of the AND gate?” The answer is that we don't know, as it is ambiguous. Arbitrarily let the Q output equal 0, and trace it through the circuit. We end up with Q=0 after we go around the loop, meaning the Q=0 output is stable and will not change. Now let Q=1 and trace it around the loop. Again, the loop is stable with Q=1. Thus for S=R=1, the circuit will stay in whatever state it is in: Q=1 or Q=0. In other words, the circuit remembers its state. In the truth table, we show this memorised state with the notation Q(t-1)

S(t)	R(t)	Q(t)
0	0	1
0	1	1
1	0	0
1	1	Q(t-1)

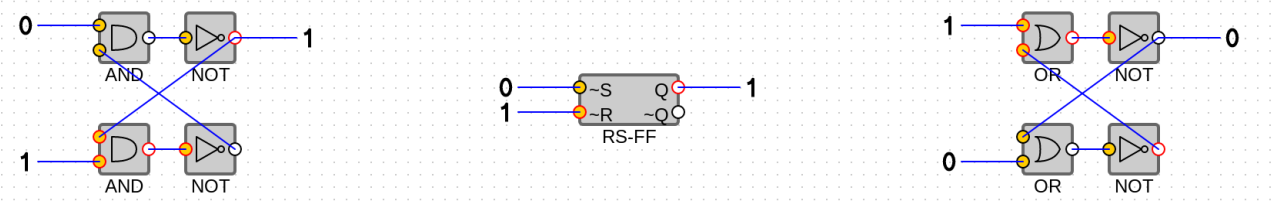
Let's explain the notation Q(t-1) by drawing a time line, which is really just a number line.



So we can tell the difference between what is happening now, and what happened in the past, we add a (t) or (t-1) to each variable. “t” means the time right now. t-1 means one time period in the past, t-2 means 2 time periods in the past, and so on. What might t+1 mean? Yes, it means one time period in the future. You hear this terminology being used in rocket launches: t-20 and counting, t-10, 9, 8, 7, ..., 2, 1, liftoff!

Note how when R=S=1, this circuit remembers it's past input. We have built a memory cell! This circuit is call a **reset-set flip-flop**, or a RS flip-flop and it can store a single bit of information. the label S stands for the word “set”, because it set the output to 1. The label R stands for “reset”, because it resets the output to 0. Q is just a traditional name for the output of a memory circuit. Can you see where the name flip-flop comes from? The circuit “flips” and “flops” between two states!

Build an RS flip-flop in the simulator. Do the same with the RS-FF component. For each circuit, very that it behaves like the above truth table. Put both inputs equal to one. Take the S input to 0 then back to 1. The output will be set to 1. Take the R input to 0 then back to 1. The output will be reset to 0. When both inputs are high, the output does not change.



Try building a flip-flop with OR gates in place of the AND gates, as on the right above. What changes? With OR gates, the flip-flop remembers its state when its inputs are 0 and you toggle the inputs to 1 to set and reset the output.

The NAND gate and NOR gate

Remember the basic gates: AND, OR NOT? Remember when connected them together to build a **compound logic** circuit: the XOR gate? The word compound just means we joined simple gates together to make more complicated gate. Here is a new compound gate: the NAND gate. NAND is short for NOT-AND. You can see this combinator of a NOT gate and AND gate in our RS flip-flop.

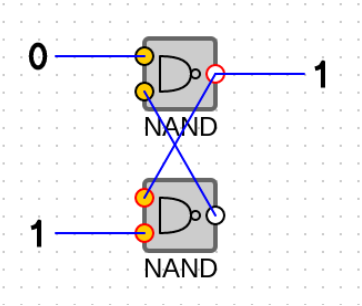
The symbol for a NAND gate is shown on the right. It's the same as the symbol with an AND gate, but with a bubble on the output. Bubbles in logic symbols usually stand for a NOT function. This is the reason the NOT gate symbol has a bubble on its output.

The truth table for the NAND gate is below. It's just the truth table for the AND gate, with the output column inverted. To jog your memory, we've shown the output for an AND gate in the third column, so you can compare it with the output of a NAND gate in the fourth column.

A	B	A AND B	A NAND B
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

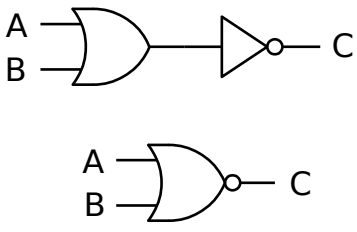
Use and AND gate and NOT gate to make a NAND gate in the simulator and compare it behaviour with the above truth table. Also place a NAND symbol from the simulator's toolbar and verify its behaviour.

Redraw our RS flip-flop with NAND gates and verify that it behaviour is the same as before.



What do you think a NOR gate looks like? That's right, it is a NOT-OR gate. It is the same as a NAND gate, but with OR in place of the AND. The truth table, circuit and symbol are below.

A	B	A OR B	A NOR B
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

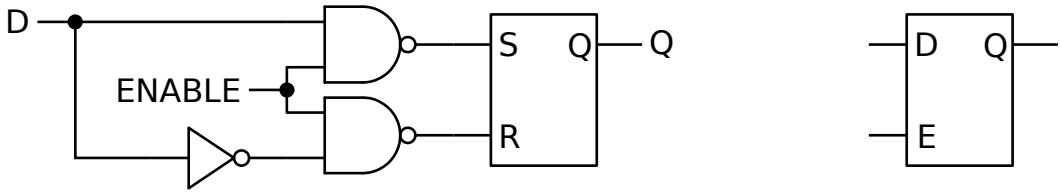


Make a NOR gate in the simulator and compare its behaviour with the truth table. Also place a NOR gate from the simulator's toolbar and verify its behaviour.

The Latch (Clocked RS Flip-Flop)

The RS flip-flop has the disadvantage that its output changes any time that one of its inputs changes. What happens if we want to pick and choose the bit that it stores, and remember that bit for a long time, even if the inputs change again? We can add a signal called an “enable”, to make a circuit called a **latch**. The RS flip-flop will only be updated when the enable signal is active. The enable signal is often called “E” for short.

Here is the circuit and symbol for a latch:



In this circuit, we put a pair of NAND gates on the input to the circuit. We have also added a NOT gate, so we input a single data (D) bit rather than having separate set and reset inputs. When the enable input is 0, the outputs of both NAND gates will be 1, meaning that the output of the RS flip-flop will not change. When the enable is 1, the RS flip-flop will be either set or reset, depending on the state of the input data bit. If the data bit is 1, the top NAND gate will drive the set (S) input of the RS flip-flip to 0 and the Q output will be set to 1. If the data bit is 0, the bottom NAND gate will drive the reset (R) input of the RS flip-flip to 0 and the Q output will be set to 1.

Here is the resulting truth table for a clocked RS flip-flop:

ENABLE	D	Q(t)
0	0	Q(t-1)
0	1	Q(t-1)
1	0	0
1	1	1

In plain English: when the enable is high, the output of the clocked RS flip-flop will be set to its input, and when the enable is low the output is remembered.

In a truth table, we can use an “X” to say that we don't care whether a bit it high or low. Using this notation, we can rewrite the above truth table as:

ENABLE	D	Q(t)
0	X	Q(t-1)
1	0	0
1	1	1

This just says that when the enable is low, the data (D) input of the latch will be ignored and the Q output will be remembered.