COMSM1302 Lab Sheet 3

Latches to flip-flops

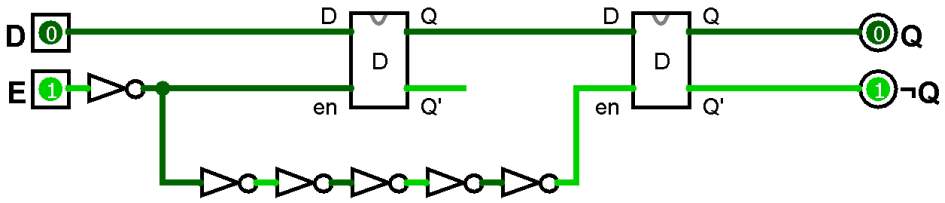
The active low R-S latch design is explicitly covered in lecture 3-1: sequential logic and the R-S latch, and the active high version can be creating by negating both the R and S inputs i.e. adding a NAND gate with copied inputs directly after each pin. For implementation see subcircuits *R-S latch (active low)* and *R-S latch (active high)*.

The active high D latch design is explicitly covered in lecture 3-2: from latches to flip-flops although the version shown in the solutions uses one less NAND gate. This uses a clever trick to transform the R' input value from $\neg(en \wedge \neg D)$ to $\neg(en \wedge \neg(en \wedge D))$ - try writing out the truth table for these expressions if you want to prove that they're logically equivalent! For implementation see subcircuit *D latch (active high)*. The active low version needs to activate when the *en* signal is 0, which can be achieved by negating the *en* input, using a NAND gate with copied inputs. For implementation see subcircuit *D latch (active low)*.

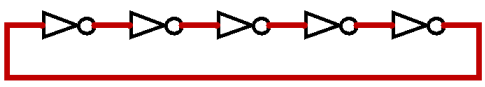
The negative edge D flip-flop design is explicitly covered in lecture 3-2: from latches to flip-flops, and the positive edge version can be created by negating the *en* signal. This is achieved either by adding a NAND gate with copied inputs to the *en* signal before it branches, or by moving the NAND gate that negated the *en* signal to the follower D latch to the *en* signal to the leader D latch instead. For implementation see subcircuits *D flip-flop (negative edge)* and *D flip-flop (positive edge)*.

Treachery of Logisim

You should observe that the output is now updating to D on both rising and falling edges of E, not just rising edges. By adding four more NAND gates (each acting as a NOT gate), we've increased the propagation delay of the falling edge of E by so much that the *en* input of the leader latch goes high and the D input propagates through to the follower latch before the *en* input of the follower latch goes low, causing the output to change.



For the next part, you should see that the lights representing the outputs of the NAND gates are on at a reduced brightness compared to a normal 1 output. What's going on here is again a matter of propagation delay - the circuit is oscillating between one state in which the five lines are 1, 0, 1, 0, and 1, and another state in which the five lines are 0, 1, 0, 1, and 0. This oscillation is happening too fast to see, but on average the LEDs are receiving half as much voltage as normal, so they appear dimmer.



This circuit is called a *ring oscillator*. In principle you could use it to output a clock signal, but the frequency is very inconsistent and depends on e.g. the temperature of the circuit boards - in fact, the variations in frequency are sometimes used as a source of hardware-based randomness.

Memory

To build a RAM circuit in Logisim, we use registers to store data and multiplexers/demultiplexers to control where data is written and read. Each register acts as one memory location. The data input signal *in* is connected to all registers simultaneously, but a register will only update its contents when its *load* (write-enable) signal is high at the same time as a clock's rising edge.

The *address* signal determines which register we want to access. During a **write** operation, the *address* is fed into a demultiplexer, which uses it to activate the load signal for exactly one register. This means only the addressed register will update its contents on the clock edge; all other registers will ignore the data input. During a **read** operation, the *address* is again used, this time with a multiplexer on the outputs of the registers. The selected register's stored data is routed to the RAM's output, while all other register outputs are ignored.

This same design can be scaled up to create larger RAM blocks by using smaller RAM units as subcircuits. For example, an 8-word RAM circuit can be reused as a building block to construct a 64-word RAM. The key idea is to split the *address* bits:

- The three most significant bits (MSBs) of the *address* choose which sub-RAM (one of the 8-word blocks) will be active.
- The three remaining least significant bits (LSBs) are passed down to that chosen sub-RAM to select the specific word inside it.

For implementation see subcircuits *RAM8*, *RAM64* and *RAM16K* - notice that RAM with 32KB of memory is listed as 16K, as the word size is 16 bits (2 bytes) and so it is made up of 16K registers.

Other flip-flops

A T flip-flop can be built from a D flip-flop by feeding the inverted output ($\neg Q$) back into the D input. On each rising clock edge, the flip-flop loads the opposite of its current state:

- If $Q = 0$, $\neg Q = 1$ is loaded so Q becomes 1
- If $Q = 1$, $\neg Q = 0$ is loaded so Q becomes 0

This causes the output to toggle between 0 and 1 on every clock cycle, which is the defining behaviour of a T flip-flop. For implementation see subcircuit *T flip-flop*.

A JK flip-flop can also be built from a D flip-flop. The provided truth table for a JK flip-flop can be expanded to include Q as an additional input, as when $J = K$ the D input on the D flip-flop depends on this. In particular:

- When $J = K = 0$, $D = Q$
- when $J = K = 1$, $D = \neg Q$

For implementation see subcircuit *JK flip-flop*.

Q \ JK	00	01	11	10
0	0	0	1	1
1	1	0	0	1

$C \equiv J \wedge \neg Q \vee \neg K \wedge Q$