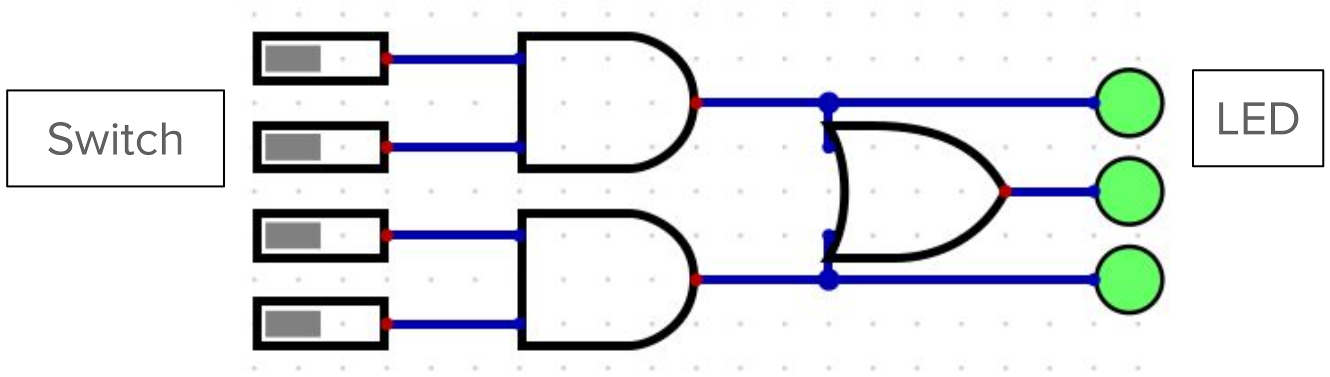
COMBINATIONAL VS SEQUENTIAL CIRCUITS

Combinational Circuits

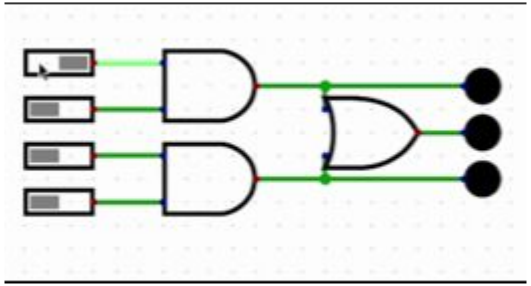
Combinational Circuits: Circuits which depend only on the current inputs. There is no memory of the previous inputs.

What does that mean?

Say we have this circuit



Combinational Circuits



It is a direct mapping from input to output with no memory

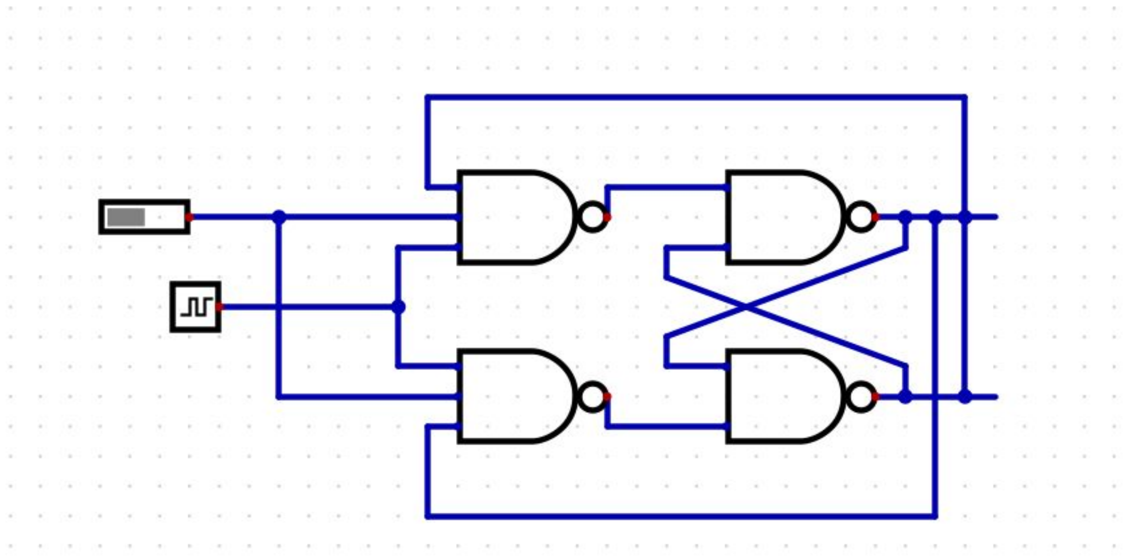
Observe the circuit’s behaviour

Regardless of what the previous input was the output for a given input is the same.

Say we represent the input with four bits
If we input 0000 after inputting 1111 or after inputting 0101 the result is “always” the same

This is the key feature of combinational circuits

Sequential circuits

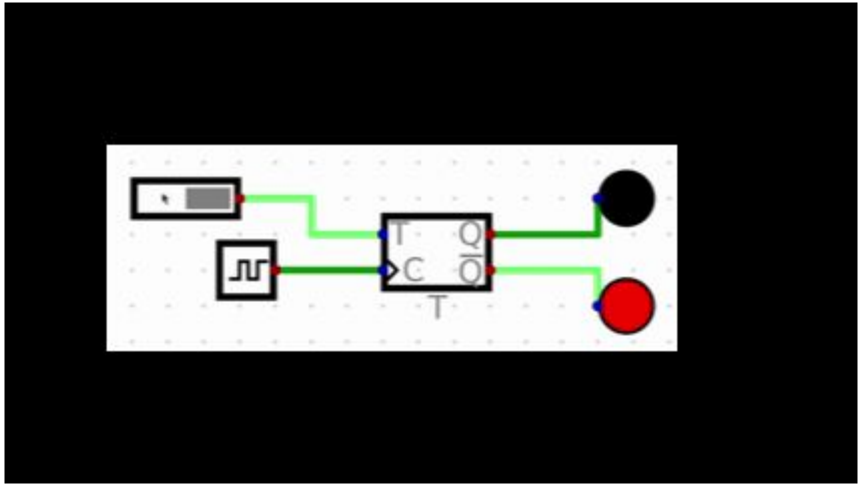


This is an example of a sequential circuit

This is the underlying circuit of a T-Flip flop

A T flip flop, flips the previous output (when the clock input is high) to get the new input

Sequential circuits



The previous circuit is too too complicated. Let us simply the T flip flop to be a black box that takes an input, clock and gives us an output

It is clear to see that when the input, clock are high the output becomes a 1c if it was a 0 previously or a 1 if it was a 0

Why does the difference matter?

The ability of sequential circuits to depend on the previous input enables them to “remember” the previous input.

This along with the clock’s ability to “freeze” the system for a cycle enables us to build systems where the action performed is dependant on the previous input and the current input

Thus sequential circuits enable us to solve more complicated problems by enabling us to remember

Why does the difference matter?

Combinational circuits can be used to model raw logic/arithmetic

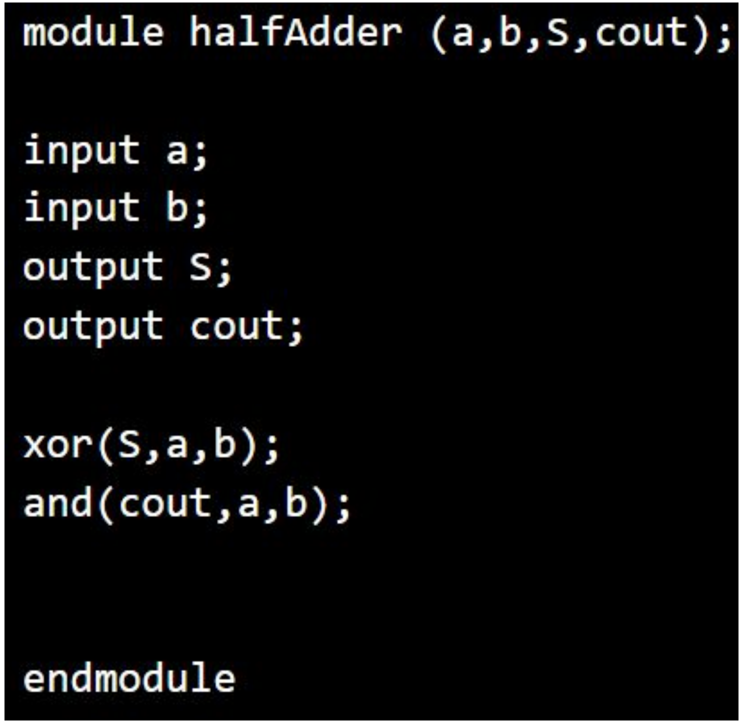
In some sense one provides us with the ability to calculate and another with the ability to remember. By combining and understanding where the applications of these circuits lie we are able to model/solve complex problems

Combinational in verilog

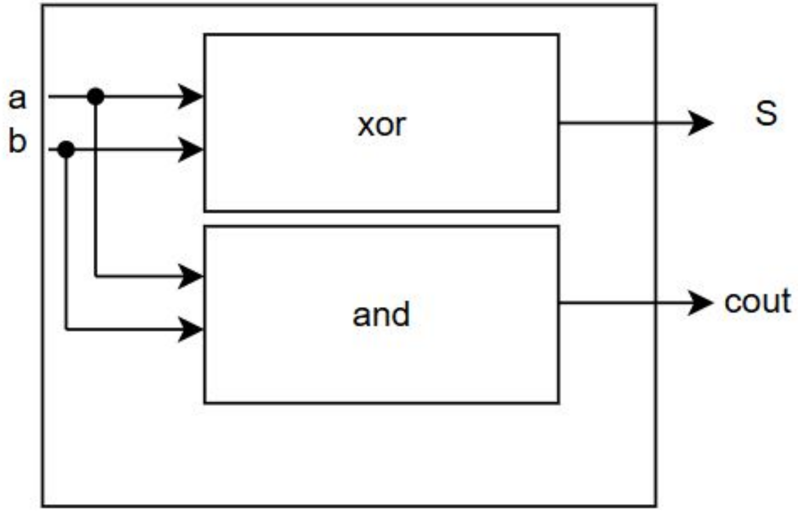
In verilog each module created is like a black box with its own logic

Only the ports (input + output) is exposed for the world to use. By instantiating and using a module we are essentially making a bigger module whose circuit uses another module

These are Identical!



```
module halfAdder (a,b,S,cout);

input a;
input b;
output S;
output cout;

xor(S,a,b);
and(cout,a,b);


endmodule
```



halfAdder

Sequential blocks in verilog

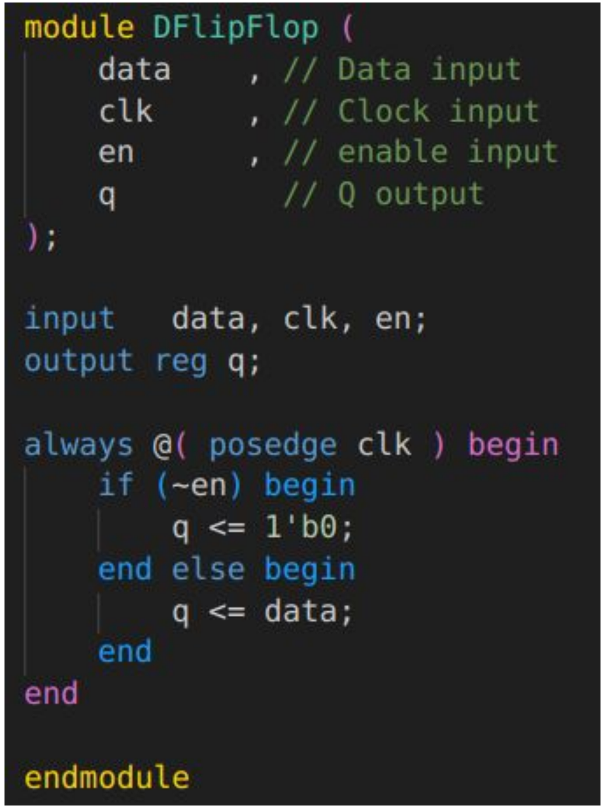
There are two types of sequential blocks in verilog:

An *initial* block. All the statements in this block are executed just once. But an *initial* block is non-synthesizable. Thus, it is only used for simulations.

An *always* block. You give a condition along with the *always* block and whenever the condition is satisfied, the statements inside it will be executed.

In verilog, when making sequential circuits, you kind of abstract away the circuit and just write the behaviour. An example of D flip flop is given on the next slide

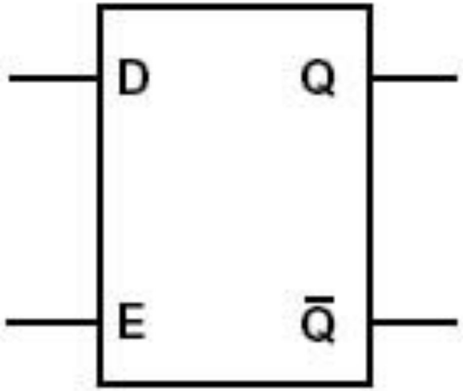
These are identical

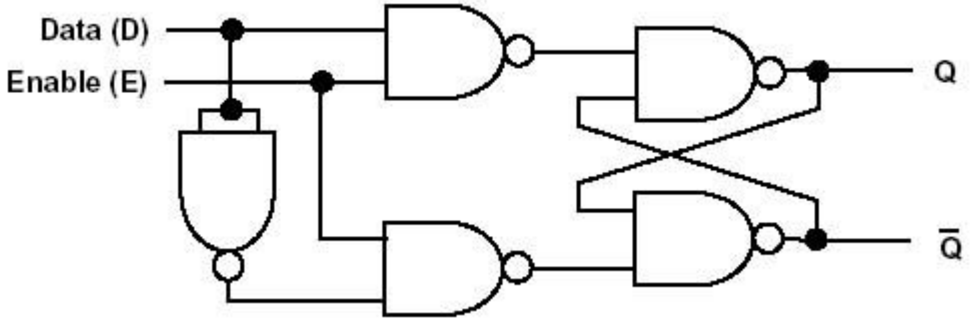


```
module DFlipFlop (
    data    , // Data input
    clk     , // Clock input
    en      , // enable input
    q         // Q output
);

input   data, clk, en;
output reg q;

always @( posedge clk ) begin
    if (~en) begin
        q <= 1'b0;
    end else begin
        q <= data;
    end
end

endmodule
```

When *enable*, q gets the value of data,

Else q = 0




Data (D)
Enable (E)
Q
$\bar{Q}$