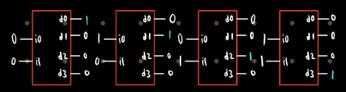
Combinational Circuits

- a function is a relation between a set of inputs and a set of permissible outputs with the property that each input is related to exactly one output.

Decoder

- Take a binary input number, and output a corresponding one-hot output
- Only one bit of the output is high, its position corresponds to the input value
- Output with is always 2 to the power of input width
  - N-input Decoder: 2^n outputs

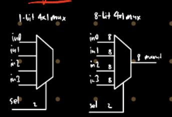
Input	Output
00	0001
01	0010
10	0100
11	1000

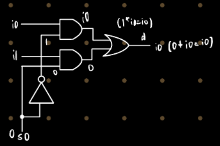


Multiplexer (mux)

- Consists of multiple inputs, and a single output
- A select input determines which input should be connected to the output

Diagrams





Timing Diagrams

- Timing diagrams show the behaviour of a circuit with progression of time

Verilog

Hardware Description Languages (HDL)
- HDLs are programming-like languages that are used to describe hardware
- HDLs are synthesized (and optimized) to hardware primitives

Modules
- A module is a container the designer can use to encapsulate a unit of functionality.

Verilog Module Declaration

① module somename (input port1, port2, output port 3)

②
```
module some_name(
    input a,b,c,
    output x);
  // Describe your circuit here
endmodule
```

- The endmodule keyword indicates the end of statements that comprise the module description.

Vectors

Verilog has a special construct for handling multi-bit signals (wires).
Formed by specifying a range:

```
module add16 (input [15:0] a,b,
              output [15:0] sum,
              output cout);
   // add module internals here
endmodule
```

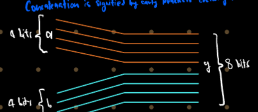
We can declare multi-bit signals (or buses as we have seen them called) in Verilog:
- By convention, we label the MSB using the higher number, and the LSB using 0.

e.g. 16-bit signal => [15:0]
8-bit signal => [7:0]
64-bit signal => [63:0]

We can select individual bits of the vector:
```
assign y = some[3]; // assign 4th bit of some to y
```
We can also select a range of the vector
```
assign z = some[4:3] // assign 5th/4th bit of some to two-bit signal z
```
We can assign to individual bits or a range
```
assign x[0] = y[1];
assign x[2:1] = y[4:3];
```
Be careful! Widths of vectors in assignments should match.

Concatenation

It is sometimes very useful to be able to concatenate a number of signals into a single signal.
Concatenation is signified by curly brackets enclosing a list




```
wire [3:0]
wire [7:0]
...
assign y = {a,b};
```

Concatenation => `assign b = {a[3:0], 4'b0000}; // b is 8-bit`

Replication uses braces with a preceding integer or variable representing an integer => `assign c = {{4{a[3]}}, a[3:0]}; // c is 8-bit`

The above example replicates the MSB 4 times

Parameters

A parameter is a constant that is local to a module
- Can be declared in the module header
- Can also be declared in the module body (we will not use this)
- Can have multiple parameters

```
module some_mod #(parameter SIZE=8, WIDTH=16) (
    input [SIZE-1:0] X, Y,
    output [WIDTH-1:0] Z);
```

Module Instantiation

We instantiate a module by invoking its name and giving an instance a name:

Named Instantiation

```
module add_half (input a,b
                 output sum, cout)
  xor g1(sum, a,b);
  and g2(cout, a,b);
endmodule
```

```
add_half M1(.a(A)
            .b(B)
            .sum(w1)
            .cout(w2));
```

Verilog Assignments

Verilog allows us to use combinational logic expressions through the assign keyword

Continuous Assignment: `assign y = a & b;`

Continuous Assignment Example

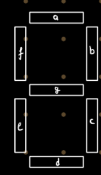
```
module full_adder (input A, B, Cin
                   output S, Cout);
  assign S = A^B^Cin;
  assign Cout = (A&B)|(B&Cin)|(A&Cin);
endmodule
```

Conditional Assignment => `assign y = sel ? x1 : x0;`

Seven Segment Decoder

Inputs X[3:0]

Hexadecimal digits (binary)	a	b	c	d	e	f	g
0 (0000)	1	1	1	1	1	1	0
1 (0001)	0	1	1	0	0	0	0
2 (0010)	1	1	0	1	1	0	1
3 (0011)	1	1	1	1	0	0	1
4 (0100)	0	1	1	0	0	1	1
5 (0101)	1	0	1	1	0	1	1
6 (0110)	1	0	1	1	1	1	1
7 (0111)	1	1	1	0	0	0	0
8 (1000)	1	1	1	1	1	1	1
9 (1001)	1	1	1	1	0	1	1
A (1010)	1	1	1	0	1	1	1
B (1011)	0	0	1	1	1	1	1
C (1100)	1	0	0	1	1	1	0
D (1101)	0	0	1	1	1	0	1
E (1110)	1	0	0	1	1	1	1
F (1111)	1	0	0	0	1	1	1



K maps

a

x_0x_1 \ x_2x_3	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	1	0	1	1
10	1	1	0	1

b

x_0x_1 \ x_2x_3	00	01	11	10
00	1	1	1	1
01	1	0	1	0
11	0	0	0	0
10	1	1	0	1

c

x_0x_1 \ x_2x_3	00	01	11	10
00	1	1	1	0
01	1	1	1	1
11	0	0	0	1
10	1	1	1	1

d

x_0x_1 \ x_2x_3	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	1	1	0	1
10	1	1	1	0

e

x_0x_1 \ x_2x_3	00	01	11	10
00	1	0	0	1
01	0	0	0	1
11	1	1	1	1
10	1	0	1	1

f

x_0x_1 \ x_2x_3	00	01	11	10
00	1	0	0	0
01	1	1	0	1
11	1	0	1	1
10	1	1	1	1

g

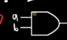
x_0x_1 \ x_2x_3	00	01	11	10
00	0	0	1	1
01	1	1	0	1
11	0	1	1	
10	1	1	1	1

Gate-Level Primitives

- Verilog provides us with basic primitives to model Boolean gates: and, nand, or, nor, xor, xnor

`or(z, a,b,c,d);` =>

`and(y, a,b);` =>

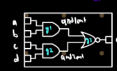


The (single) output is always the first argument

Wires

We can declare internal wires in a module, using the wire keyword:

`wire int_signal;` This creates a named (1-bit) wire that can be used to connect gates together.




```
module andornor (input a,b,c,d,
                 output out);
  wire and1out, and2out;
  and g1(and1out, a,b);
  and g2(and2out, c,d);
  nor g3(out, and1out, and2out);
endmodule
```

Note that the order of statements is unimportant

- Identifiers in Verilog
  - Must start with a letter or underscore
  - Can contain letters, numbers, underscore, and dollar sign ($)
  - Must NOT clash with keywords (e.g. module, input, etc.)

Range of operators

OPERATOR	NAME
~	Bitwise NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~&	Bitwise NAND
~|	Bitwise NOR

☆ The signal y will be connected to x1 if sel is 1, else is is connected to x0.

`assign max = (a>b) ? a:b;`

Number Literals

Verilog allows us to use number literals
- <size>'<radix><value>
  - <size> is the width in bits
  - <radix>: b for binary, o for octal, h for hex, d for decimal
  - <value>: the number you want, with as many optional underscores as needed (for readability)

Examples:
- 4'b0000 (4 binary bits "0 0 0 0")
- 8'h4F (= 8'b01001111)
- 8'b0100_1111 (same as above. Note the use of the underscore)
- 1'b1 (a single "1")

Underscores can help split long strings:
- 16'b0010_1110_0110_1001 (easier than 0010111001101001)

Verilog Combinational Circuits

Behavioral Modelling:
A more powerful: we describe how the circuit behaves, not how it is constructed

Combinational always Block

- For behavioural design, we use a type of procedural block called an always block.
- An always block contains procedural statements that describe the behaviour of the desired hardware.

- The always keyword starts with a block
- The sensitivity list must contain the names of any signals that affect the output of the block
- Luckily, there's a shortcut: always @ * - use it!
- You do not use the assign keyword inside an always block, EVER
- If there is more than one statement, use begin and end to define the start and end of the block.

Reg

- Signals you assign to from within an always block
  - must be declared as being of type: reg
- reg is synchronous with wire, but these signals can be assigned to from inside an always block, wires cannot!
- A wire is simply a connection, it holds no value of its own.
- The reg type is more like a variable in programming, as we will see.
- Declare reg signals:
  - You can set an initial value when you declare reg signals

```
reg a,b;
reg [3:0] x = 4'b0000;
```

```
module temp (input a,b
             output out);
  reg w1;
  always @ *
    w1 = a&b;
  assign out w1;
endmodule
```

sensitivity list
```
always @ (a,b)
  begin
    x = a & b;  }  procedural
    y = a | b;  }  statements
  end
```

☆ Assignments in an always block are exactly the same as assignments using an assign:

```
always @ *
begin
   x = a & b;
   y = a | b;
end
```
(=)
```
assign x = a&b;
assign y = a|b;
```

- If there is just one statement in the body of the statement (like if, for, or always block), then there is no need for begin and end.
- If there is more than one statement, you MUST use begin and end.

IMPORTANT CONSIDERATIONS

If you assign to a signal from inside an always block, you must NEVER assign it from ANYWHERE ELSE!

☆ Each always block represents the logic that generates a certain signal or a set of signals
- Hence, you should only assign to those signal(s) from inside that always block

Statement Order in always block
When we read the statements in an always block, we read them in order, and ORDER MATTERS.

Tutorial 8

Q1. Using a behavioural description, write a Verilog module that has three 8-bit inputs, a, b and c, and outputs the largest (max) and smallest (min) of them. Add a third output (diff) that outputs the difference between the two. You should use if statement(s). You can ignore equal inputs.

PSEUDOCODE

Inputs	Output
a (8-bit)	max = max(a,b,c)
b (8-bit)	min = min(a,b,c)
c (8-bit)	diff = max - min

```
if (a>=b && a>=c) max = a
else if (b>=a && b>=c) max = b;
else if (c>=a && c>=b) max = c;

if (a<=b && a<=c) min = a;
else if (b<=a && b<=c) min = b;
else if (c<=a && c<=b) min = c;

diff = max - min;
```

max

Verilog

```
module agglmao (input a[7:0], b[7:0], c[7:0]
                output max[7:0], min[7:0], diff[7:0])

  if (a>=b && a>=c) max = a
  else if (b>=a && b>=c) max = b;
  else if (c>=a && c>=b) max = c;

  if (a<=b && a<=c) min = a;
  else if (b<=a && b<=c) min = b;
  else if (c<=a && c<=b) min = c;

  diff = max - min;
```

SEQUENTIAL CIRCUITS

Sequential circuits introduce the idea of state
- The state of a circuit encodes sufficient information about the past to determine how it will react with inputs to produce an output

If Statement

always @ *

Case Statement

always @ *
case (sel)

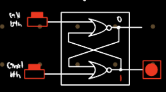
Multiple always blocks

The order of multiple always blocks in a module doesn't matter
... concurrently ... way to fix this is to use a default

The output of a sequential circuit depends on both the current inputs and previous inputs

Set-Reset (SR) Latch

The MOST BASIC circuit for storing a bit

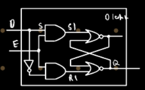


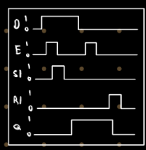
Transparent D-Latch

The D latch, aka the transparent latch removes the possibility of invalid inputs.

- Inverter ensures R is always opposite of S
- A single input controls both states of the latch.

The D latch effectively passes its input through to its output whenever the enable input is high.





Truth table

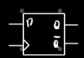
E	D	Q+	Function
0	X	Q	Store
1	0	0	Transparent
1	1	1	Transparent

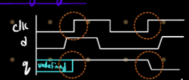
We can use an edge triggered flip-flop to overcome the limitation of D-latches

Master-Slave implies 2 latch stages:

- Master stage is enabled when control is low, and samples the current input, while slave is disabled, latching in the middle
- When the control goes high, the master is disabled, and the slave uses the value from the middle to determine its output
- Hence, the only time an output will change is as the rising edge
- For a negative-edge triggered setup, the master responds to high enable, and the slave to low enable

D-flip flop Timing diagram





- Note transitions on the output only occur at the RISING clock edge
- Latches can be level-sensitive, i.e. their outputs change when the control (enable) input is HIGH!
- A flip-flop is a circuit that changes its outputs only at the control signal's edges
  - Positive edge-triggered: output changes when control goes from 0 to 1
  - Negative edge-triggered: output changes when control goes from 1 to 0.

Synchronous Counter

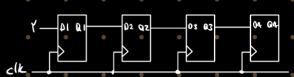
At each clock edge, the incremented value, inc, derived from the current output, is passed to q.

```
module simplecount (input clk, rst,
                    output reg [3:0] q);

  always@(posedge clk)
    begin
      if (rst) q <= 4'b0000;
      else q <= q + 1'b1;
    end
```

Shift Registers in Verilog

Shift registers take a single input and pass it through a chain of flip-flops

At each clock cycle, the input progresses one stage

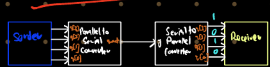


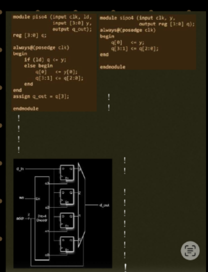
Memories

- Consists of an array of storage elements
- Can think as: Register → 1 storage elem.
- Each storage element has a unique address

- We should be able to select which register to store to using an address
- We should be able to read the value out of any register, again using the address

Serial Data Transfer





```
module piso4 (input clk, ld,
              input [3:0] y,
              output q_out);
reg [3:0] q;

always@(posedge clk)
begin
    if (ld) q <= y;
    else begin
        q[0]   <= y[0];
        q[3:1] <= q[2:0];
    end
end
assign q_out = q[3];

endmodule
```

```
module sipo4 (input clk, y,
              output reg [3:0] q);
always@(posedge clk)
begin
    q[0]   <= y;
    q[3:1] <= q[2:0];
end

endmodule
```

```
    if (alarm == 1'b1)
      begin
        if (after_hours)
          siren = 1'b0;
        else
          siren = 1'b1;
        light = 1'b1;
      end
    else
      begin
        siren = 1'b0;
        light = 1'b0;
      end
  end
```

```
    2'd0 : y = a;
    2'd1 : begin
             y = b;
             z = c;
           end
    2'd2 : y = c;
    default: y = 4'b1010;
  endcase
```

- They occur concurrently

Avoiding Latches

```
always @*
begin
  if (valid) begin
    x = a|b;
    y = c;
  end
  else
    x = a; //y?
end
```

=>

One way to fix this is to use a default assignment at the top of the always block

```
always @*
begin
  if (valid) begin
    x = a|b;
    y = c;
  end
  else
    c = a;
end
```

Multiple always Blocks

The order of multiple always blocks in a module doesn't matter

- They occur concurrently

Avoiding Latches

missing y → - called a 'latch'

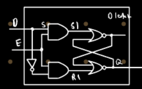
```
always @*
begin
  if (valid) begin
    x = a|b;
    y = c;
  end
  else
    x = a; //y?
end
```

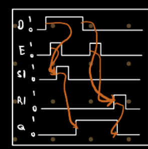
=>

One way to fix this is to use a default assignment at the top of the always block

```
always@*
  y = x;
begin
  if (valid) begin
    x = a|b;
    y = c;
  end
  else
    x = a;
end
```

- The default is overwritten by any subsequent assignment.
- Assignments at the top of an always block are hence a great way of remembering to assign to all your signals





Truth table

E	D	Q+	Function
0	X	Q	Store
1	0	0	Transparent
1	1	1	Transparent

Clock

- A signal that continuously toggles between 0 and 1 at a fixed rate.

clk ⎍⎍⎍⎍⎍⎍⎍⎍⎍

Period => the time to complete one complete cycle - we call this a cycle

clk (One clock period)

Frequency => inverse of period

Register

When we combine multiple D flip-flops together to store multiple bits, we call this a register

Verilog Sequential Circuits

- General sequential blocks in Verilog is generally used to describe synchronous circuits, i.e. edge-triggered components
- Synthesis tools will generally convert designs to D-type flip-flops/registers

Registers in Verilog


```
module simplereg (input d, clk,
                  output reg q);
always@(posedge clk)
  q <= d;
endmodule
```

This creates a 1-bit register/D flip-flop with input d and output q.

Note the new always block format

- For combinational, we list signals, or use always@*
- For synchronous, we use always@(posedge clk)

↑

This tells the synthesis tools that the block's behaviour should only happen at the clock rising edge, hence creating a flip-flop/register.

★ Falling-edge triggered
always@(negedge clk)

Reset the value in a register

- Asynchronous => asserted, contents of register set to the reset value
- Synchronous => ^, but only at a RISING EDGE!

Assignments in Always Blocks

Combinational Always Blocks => (=) blocking assignment ORDER MATTERS

Synchronous Always Blocks => (<=) non-blocking assignment ORDER DOES NOT MATTER

Clock and Reset

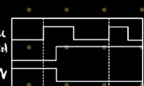
The signal name should be whatever the clock signal is => i_clk, gen_clk, clock, clk2400

If we're naming it, we often just use clk

Remember to add the clock input to your module port list

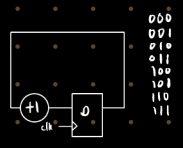
ALL synchronous always blocks should use the same clock signal

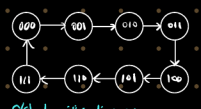
```
module simplereg (input [7:0] d,
                  input clk, rst,
                  output reg [7:0] q);

  always@(posedge clk or posedge rst)
  begin
    if (rst) q <= 8'b0000_0000;
    else q <= d;
  end
endmodule
```



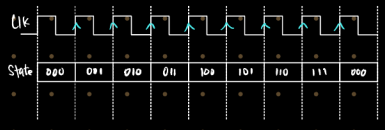
Sequences & States

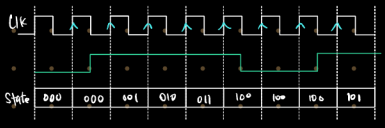
- For this counter, the sequence is easy to remember about
- At each point in time, the system is in a **state**, that determines what the output should be
- At each transition point, the circuit moves from its **current state** to the **next state**, corresponding to the one with the next target output.
- We can think of circuits as being **state machines**



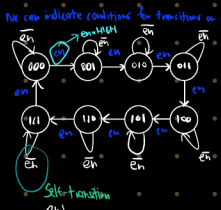


State transition diagram
- Each node is a possible state of the system
- It shows the movement from one state to the next.
- In this case, the states are simply labeled with their output values.





We can indicate conditions for transitions on the transition arrows

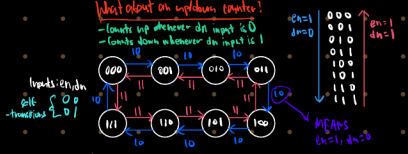


SHIFT REGISTERS

1. SISO Serial In Serial Out
2. SIPO Serial In Parallel Out
3. PISO Parallel In Serial Out
4. PIPO Parallel In Parallel Out

What about an up/down counter?

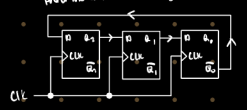
- Counts up whenever dn input is 0
- Counts down whenever dn input is 1



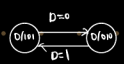
OTHER INPUTS RESULT IN NO TRANSITION!!

PRACTICE:

Draw the state transition diagram for the following circuit.

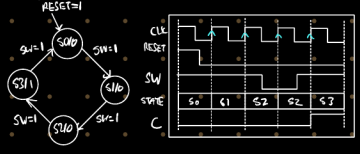


STATE TRANSITION DIAGRAM



PRACTICE:

The following FSM has 4 STATES S0, S1, S2, S3 and 2 inputs. (RESET and SW) It also has an output C. Complete the following timing diagram of the FSM by entering the name/sum for the STATE and output C.



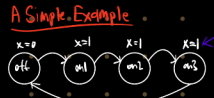
FINITE STATE MACHINES (FSM)

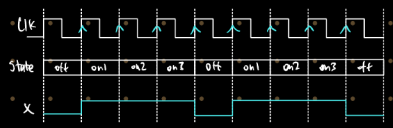
Describes a system using a finite number of states and the associated transitions

- In **Synchronous design**, an FSM is in one state for the duration of the clock cycle.
- At every rising clock edge, the FSM may transition to another state
- Whether it transitions or not depends on the input values at the rising edge
- Once it makes the transition, it remains in that state for the next cycle.

- We can label the states in a finite-state machine with any meaningful name
  - In the counter sequences we saw, we just used the counter value, but often we want more meaningful information.
- They are called "Finite" because FSMs have a finite number of states (unlike some similar representations in other domains)
- "Machine" refers to a general object that can execute an abstract language.

A Simple Example





We can indicate the outputs above, below, or beside the corresponding states.

It is also possible to put the outputs inside the state bubbles (can be on a separate line to state name).

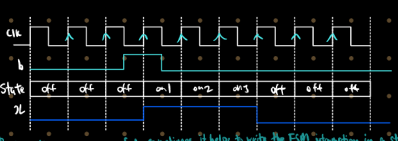
Include a legend so it is clear what the output signal is coded.

Adding Inputs

- What if we want the previous FSM to only output the three cycle high pulse when an input signal is high.

Input: b
Output: X

- Now, the FSM remains in state off until the b input goes high
- At that point it outputs an 1 as it passes through on1, on2 and on3, before returning to off.



While diagrams are useful, sometimes it helps to write the FSM information in a state transition table.

STATE TRANSITION TABLE

Current State	Next State b=0	Next State b=1	Output
off	off	on1	0
on1	on2	on2	1
on2	on3	on3	1
on3	off	off	1

another variant omitting state transition table

Current State	b	Next State	Output X
off	0	off	0
off	1	on1	0
on1	0	on2	1
on1	1	on2	1
on2	0	on3	1
on2	1	on3	1
on3	0	off	1
on3	1	off	1