CX1005
Digital Logic

Verilog for Combinational Circuits

So, Where are We?

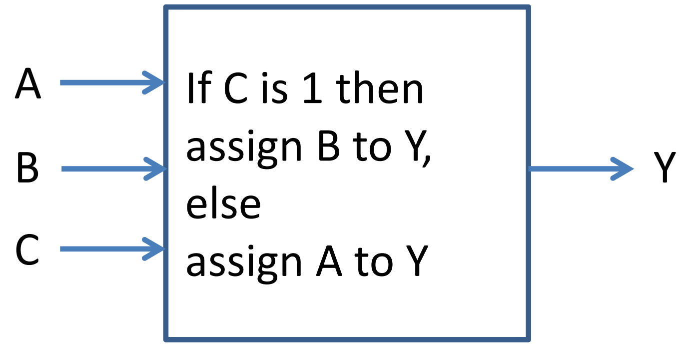
- We have covered: **modules**, **ports**, **wires**, **module instantiation**, **gates** and **assign statements**

```
module myModule (input a, b, data_i,
                 output flag, data_o, addr);
       wire or1out;

       get_Data IC1 (.din(data_i), .add(addr), .do(data_o));
       or g1 (or1out, a, b);
        assign flag = or1out & data_i;

endmodule
```

- So far we have only examined **Structural Verilog**.

Behavioral Modeling

- A more powerful level: we describe **how** the circuit **behaves**, not how it is *constructed*
- Synthesis tools work out how to make hardware that fulfills the description, taking into account the target architecture




```
module counter (clk,
reset, enable, count);
  input clk, reset,
enable;
  output [3:0] count;
  reg [3:0] count;

  always @ (posedge clk)
  if (reset == 1'b1)
      count <= 0;
else if ( enable == 1'b1)
      count <= count + 1;

endmodule
```

Tools
(e.g. Logic
Synthesizer)

Combinational always Block

- For ***behavioral design***, we use a type of procedural block called an **always** block
- An **always** block contains procedural statements that describe the behavior of the desired hardware

Sensitivity List

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

Procedural statements

- The **`always`** keyword starts a block
- The *sensitivity list* **must** contain the names of any signals that affect the output of the block

always @ *

- If you accidently leave a signal out you will not get the expected circuit
- Luckily, there's a shortcut: **`always @ *`** **– use it!**

```
always @ (a, b)
  begin
    x = a & b;
    y = a | b;
    z = a & c;
  end
```

This will not generate the expected output as it will not be triggered when input c changes

```
always @ *
  begin
    x = a & b;
    y = a | b;
    z = a & c;
  end
```

The “*” means all input signals inside the block.

This is OK. Whenever any signal changes (* means all input signals) a change in the output is forced.

Combinational always Block

- The statements can be assignments as in the example or more complex structures
- 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.

```
always @ *
  begin
    x = a & b;
    y = a | b;
    z = a & c;
  end
```

Reg

- Signals you assign to from within an ***always*** *block*
  - Must be declared as being of type: **`reg`**
- **`reg`** is synonymous with wire, but these signals can be assigned to from inside an always block, **`wire`**s 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
- Note: you *cannot* assign to a **`reg`** signal using an ***assign*** *statement*, or connect it to module instance outputs.

```
module temp  (input a, b,
                      output out);

   wire w1;

    assign w1 = a & b;
    assign out = w1;

endmodule
```

```
module temp  (input a, b,
                      output out);
   reg w1;

   always @ *
        w1 = a & b;

   assign out = w1;
endmodule
```

Reg

- Declare **reg** signals:
  - You can set an *initial value* when you declare **reg** signals

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

- You can also declare your module outputs as **reg** if you plan to assign to them directly from within an **always** block:

```
module sel_one (input [5:0] a, b, c, d,
                input [1:0] sel,
                output reg [5:0] sigsel);
```

So What is Synthesized?

- 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;
```

Exactly the same circuit will be synthesized

If Statement

- With always blocks, we can use some powerful constructs:
- **If** statement:

```
always @ *
begin
   if (x < 6)
      alarm = 1'b0;
   else
      alarm = 1'b1;
end
```

- You can have more than one statement in each branch. If so, you use **begin** and **end**

```
always @ *
begin
   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
```

A note on begin and end

- Verilog follows C standards for blocks
- The difference is that Verilog uses **begin** and **end**, rather than **{ }**
  - **Note { } is used for concatenation**
- If there is just one statement in the body of a statement (like if, for, or always block), then there is no need for **begin** and **end**
- If there is more than 1 statement, you must use **begin** and **end**

```
always @ *
begin

if (x < 6)
  alarm = 1'b0;
else
  alarm = 1'b1;

end
```

```
always @ *
if (x < 6)
begin
    alarm = 1'b0;
end
else
begin
    alarm = 1'b1;
end
```

These are the same

```
always @ *
if (x < 6)
    alarm = 1'b0;
else
    alarm = 1'b1;
```

Case Statement

- We can use **case** statements:

```
always @ *
case (sel)
    2'b00   : y = a;
    2'b01   : y = b;
    2'b10   : y = c;
    default : y = 4'b1010;
endcase
```

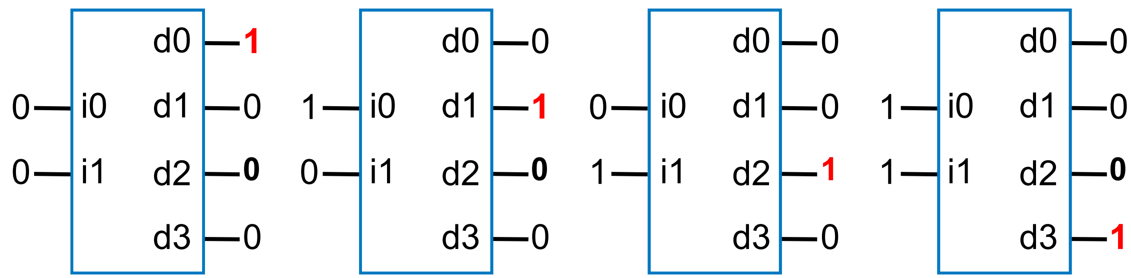
```
always @ *
case (sel)
    2'd0    : y = a;
    2'd1    : begin
                  y = b;
                  z = c;
              end
    2'd2    : y = c;
    default : y = 4'b1010;
endcase
```

- There is no need for a break in each branch (unlike some software languages)

Implementing Decoder with Verilog

<u>Input</u> <u>Output</u>

00 → 0001
01 → 0010
10 → 0100
11 → 1000



- Example: 3-to-8 decoder:

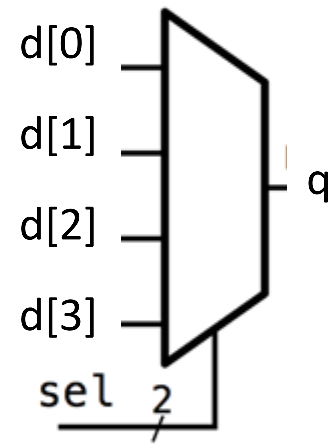
<u>Input</u> <u>Output</u>

000 → 00000001
001 → 00000010
010 → 00000100
011 → 00001000
…
111 → 10000000

```
module decoder3_8(output reg [7:0] d_out,
                  input  [2:0] ival);
always @ *
  case(ival)
     3'b000  :  d_out = 8'b00000001;
     3'b001  :  d_out = 8'b00000010;
     3'b010  :  d_out = 8'b00000100;
     3'b011  :  d_out = 8'b00001000;
     3'b100  :  d_out = 8'b00010000;
     3'b101  :  d_out = 8'b00100000;
     3'b110  :  d_out = 8'b01000000;
     3'b111  :  d_out = 8'b10000000;
  endcase
endmodule
```

Implementing Multiplexer with Verilog

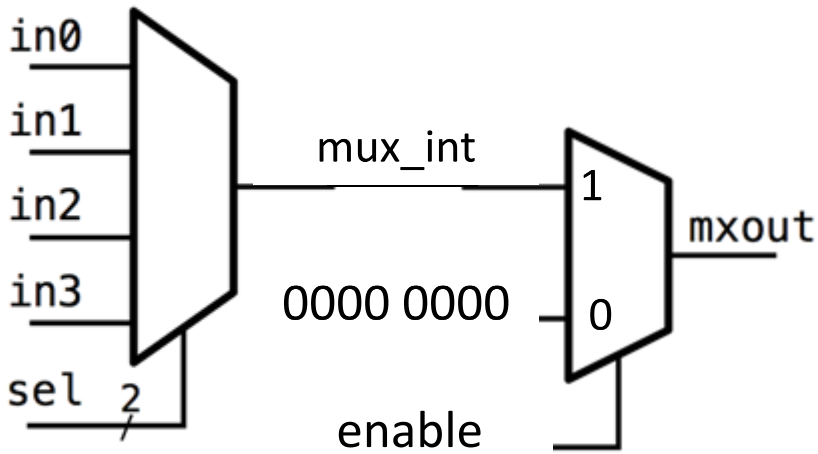
1-bit 4x1 mux




```
module mux4 (output reg  q,
             input [3:0] d,
             input [1:0] sel);

   always @ * begin
      case (sel)
         2'b00 : q = d[0];
         2'b01 : q = d[1];
         2'b10 : q = d[2];
         2'b11 : q = d[3];
      endcase
   end
endmodule
```

Note the begin and end. These are not needed as a case is a single statement. But makes it easier to read!!

Implementing Multiplexer with Verilog




```
module mux_4_32(output [7:0] mxout,
                input  [7:0] in3, in2,
                input  [7:0] in1, in0,
                input  [1:0] sel,
                input        enable);

   reg [7:0] mux_int;

   assign mxout = enable ? mux_int : 8'd0;

   always @ *
     begin
       case(sel)
         2'b00   :  mux_int = in0;
         2'b01   :  mux_int = in1;
         2'b10   :  mux_int = in2;
         2'b11   :  mux_int = in3;
       endcase
     end
endmodule
```

IMPORTANT CONSIDERATIONS IN BEHAVIORAL VERILOG

Signal Assignments - always Block

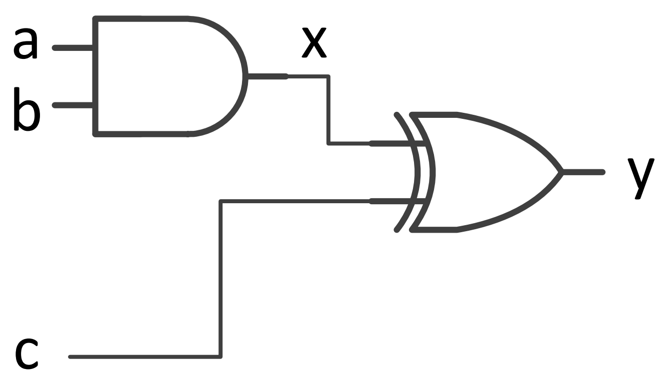
- Notice that we can assign to ***multiple*** signals from inside one **always** block
- If you assign to a signal from inside an **always** block, you must **never** assign to it from **anywhere else!**
- That is, not from:
  - **assign** statements
  - Other **always** blocks
- Each **always** block represents the logic that generates a certain signal or set of signals
  - Hence, you should only assign to those signal(s) from inside that **always** block
- Only group signals into a single **always** block if it makes sense to do so – generally when they follow similar logic

```
assign x = . . .;

always @ *
begin
  x = a & b;
  y = a | b;
end

always @ *
begin
     y = . . .;
end
```

Statement Order in always block

- The issue of statement order can be a little confusing
- When we read the statements in an **always** block, we read them in order, and **order matters**
- The synthesis tools turn the combined behavior described into a piece of combinational hardware
- Hence, if we assign to a signal more than once, in the end, the last assignment wins, since that's what the tools will use

```
always @ *
begin
      x = a & b;
      y = x | c;
      y = x ^ c;
end
```



Multiple always blocks

- The order of multiple **always** blocks in a module doesn't matter
  - They occur concurrently
- You can have as many **always** blocks as you want, and they each implement a piece of concurrent hardware

```
always @ *
begin
    a = … ;
    b = … ;
end

assign c = … ;

always @ *
begin
    y = … ;
    x = … ;
end

assign z = … ;
```

Order is important

Order is important

Order is not important

```
assign z = … ;
assign c = … ;

always @ *
begin
    y = … ;
    x = … ;
end

always @ *
begin
    a = … ;
    b = … ;
end
```

Avoiding Latches

- What happens to *y* in the **else** branch?

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

- This would imply *y* should keep its value – This is the behavior of a latch not combinational logic!

Avoiding Latches

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

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

Avoiding Latches

```
always @ *
  case (sel)
    2'b00   : y = a;
    2'b01   : begin
               y = b;
               z = c;
              end
    2'b10   : y = c;
    default : y = 4'b1010;
endcase
```

```
always @ *
begin
  z = a;          // default
  y = 4'b1010; // default
  case (sel)
    2'b00 : y = a;
    2'b01 : begin
             y = b;
             z = c;
            end
    2'b10 : y = c;
  endcase
end
```

But, as we only assign to z in one instance, this implies that you are storing the data. **Must not do this as it synthesises storage.**

Summary

- You must always include *any* signal that affects the output in the *sensitivity list*
  - else the synthesizer will produce something unexpected!!
  - *So,* easier to just use **always @ ***
- You must assign to the output signal in **all** possible cases
  - As you are designing combinational logic, it makes no sense to sometimes assign and other times not – that implies storing a value – That is not combinational logic
  - If you forget to do this, you will end up with a latch in your circuit, which is not wanted!!!
  - More on this when you start to look at **Sequential Circuits**