The Verilog Primer

So you want to learn Verilog

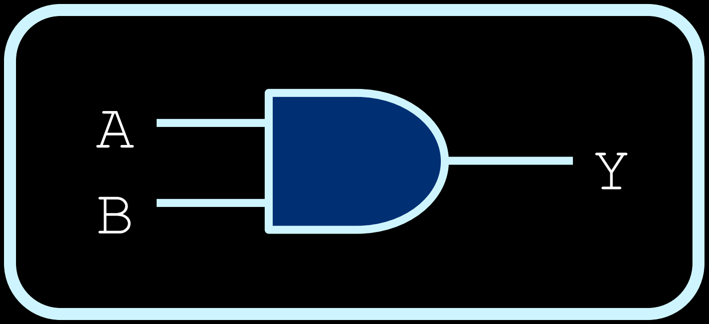
- Verilog is a new language that is used a lot in the CSC258 labs. It's not hard to learn, but it is different from other languages that you've seen before.
- This primer is meant to give you a sense of what Verilog is about, and what you need to know in order to get by in the labs.
- It's a living document, so if there's anything that's missing or unclear, let us know!

Understanding Verilog

- The first thing to realize about Verilog is that it is not a programming language, but is a hardware description language (HDL).
- It's used to describe what the layout needs to look like, start designing circuits too large or complicated t implement with actual chips and wires.

Basic Verilog Example

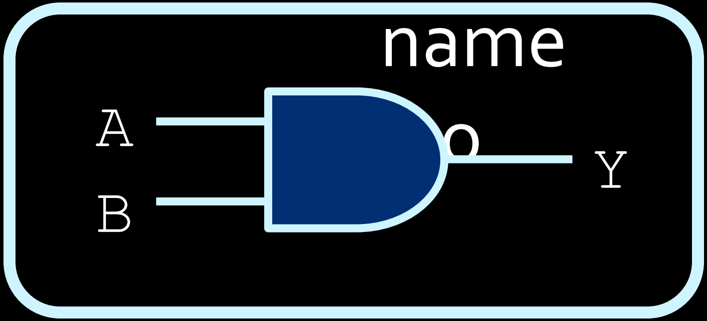
- For instance, this is a simple AND gate:



- If you had to create a software language that would allow you to specify an AND gate with these inputs and outputs, what would the specification look like?

Basic Verilog Example (cont’d)

- It would have to have a that you could use describe the gate.
  - e.g. **“and”**



- It would have to allow you to specify the inputs and outputs of the gate.
  - e.g. `and(Y, A, B)`
  - Since gates can have many inputs but only one output, the output is listed first, followed by all of the gate’s inputs.

The basics of Verilog

- Verilog is based off the idea that the designer of the circuit needs a simple way to describe the components of a circuit in software.
- There are several basic primitive gates that are built into Verilog.

➢ `and(out,in,in)`	➢ `or(out,in,in)`	➢ `not(out,in)`
➢ `nand(out,in,in)`	➢ `nor(out,in,in)`	➢ `buf(out,in)`
➢ `xor(out,in,in)`	➢ `xnor(out,in,in)`	

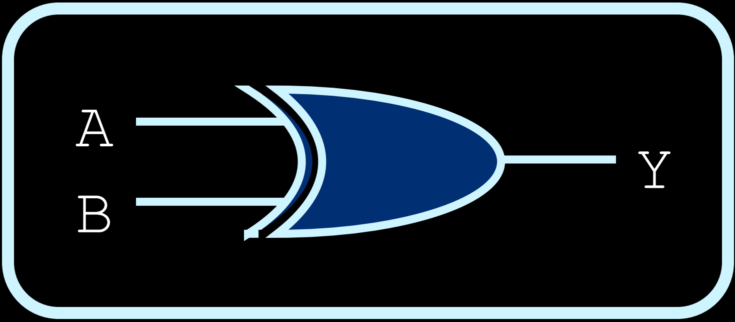
  - two-input gates shown here, but multi-input also possible.

Creating modules

- Using built-in gates is one thing, but what if you want to create logical units of your own?
- Modules help to specify a combination of gates with a set of overall input and output signals.
  - Specified similarly to C and Python functions.
  - Less like functions though, and more like specifying a part of a car.

Module Example

- Making an XOR gate.
  - An XOR gate can be represented with the following logic statement:

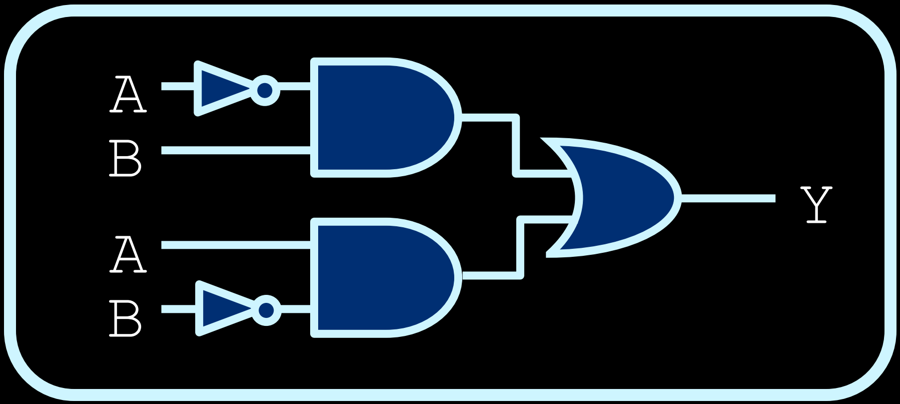


$$Y = A \cdot \overline{B} + \overline{A} \cdot B$$

  - How would we describe a logic equation like this in a hardware design language?

Module Example (cont'd)

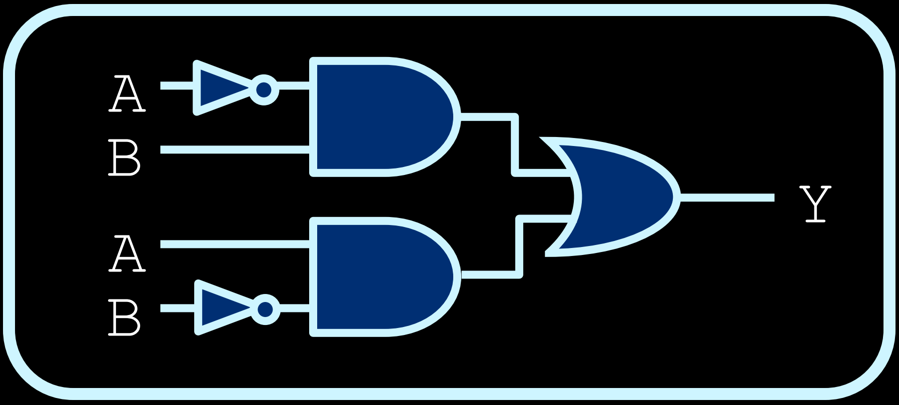
- Not to hard to represent it in logic gates.



- How would you specify the AND and OR gates?

```
and(__,B,__)
and(__,A,__)
or(Y,__,__)
```

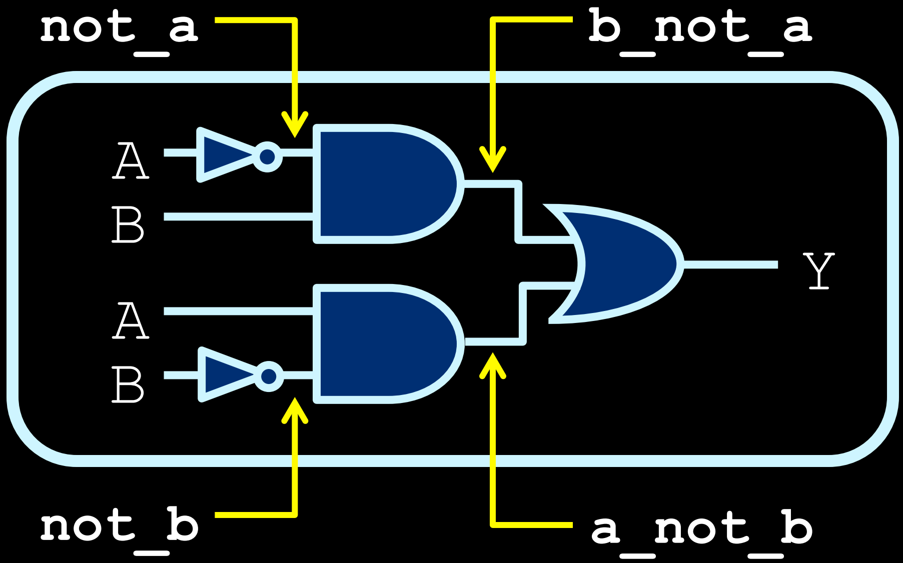
Module Example (cont’d)



```
and(__,B,__)
and(__,A,__)
or(Y,__,__)
```

- Wires that are used internally in the circuit to connect components together are declared and labeled using the `wire` keyword.
    - Label the output of each gate, so that you can refer to it when specifying the inputs of other gates.

Module Example (cont’d)



- Note: the wire names are not built in or named according to any convention. The names of the wires is at the discretion of the designer.

Module Example (cont’d)

- The result is the circuit description on the right.
- The order of the five lines at the bottom doesn’t matter.
    - Remember: Verilog is a hardware description, not a programming language, so the result is the same.

```
wire not_a, not_b;
wire a_not_b, b_not_a;

and(b_not_a, B, not_a);
and(a_not_b, A, not_b);
or(Y, b_not_a, a_not_b);

not(not_a, A);
not(not_b, B);
```

Module Example (cont’d)

- The module is nearly done!
- Only missing three things:
    1. Semicolons at the end of each line.
    2. Statements describing the circuit’s input and outputs.

```
input A, B;
output Y;

wire not_a, not_b;
wire a_not_b, b_not_a;

and(b_not_a, B, not_a);
and(a_not_b, A, not_b);
or(Y, b_not_a, a_not_b);

not(not_a, A);
not(not_b, B);
```

Module Example (cont'd)

- Last missing feature:
  3. Keywords laying out the start and end of the module, as well as the input and output signals.

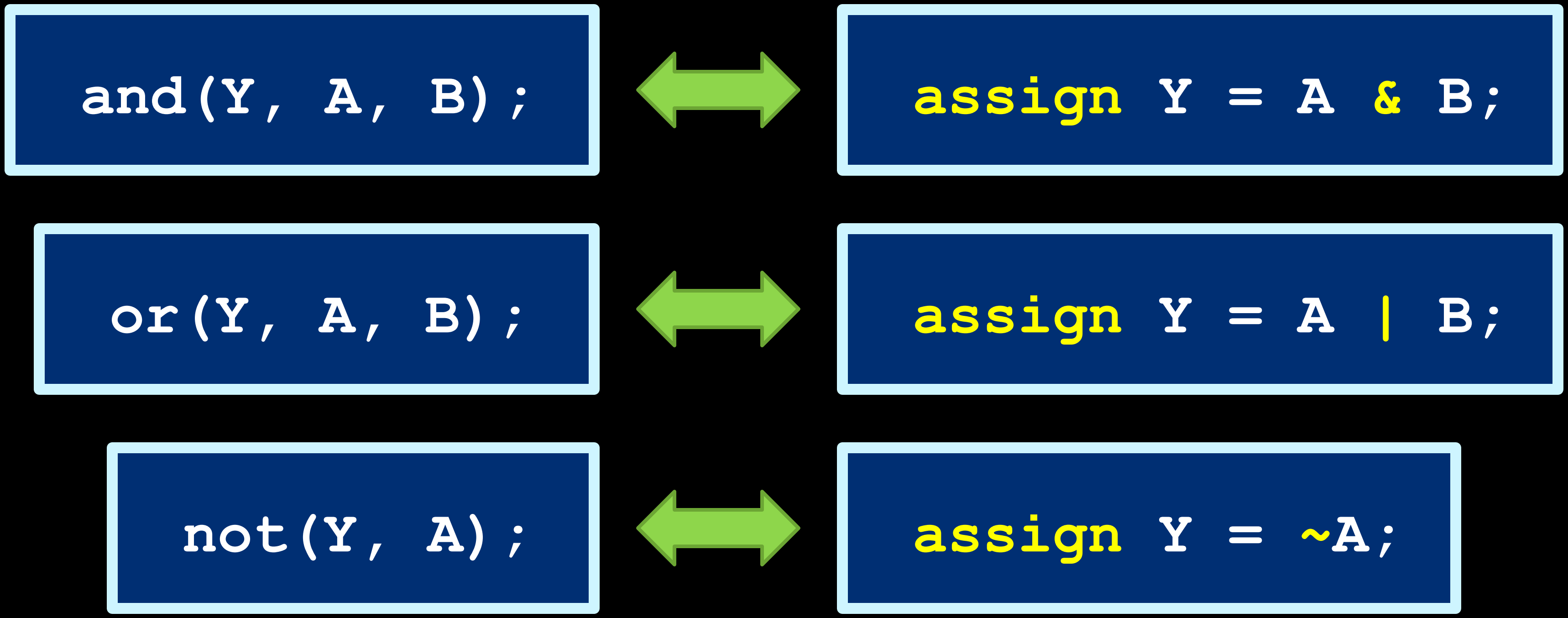
```
module xor_gate(A, B, Y);
   input A, B;
   output Y;

   wire not_a, not_b;
   wire a_not_b, b_not_a;

   and(b_not_a, B, not_a);
   and(a_not_b, A, not_b);
   or(Y, b_not_a, a_not_b);

   not(not_a, A);
   not(not_b, B);
endmodule
```

Module review

- Creating a module follows a few simple steps:
  1. Declare the module (along with its name, its input and output signals, and where it ends).
  2. Specify which of the module's external signals are inputs and which are outputs.
  3. Provide labels for the internal wires that will be needed in the circuit.
  4. Specify the components of the circuit and how they're connected together.

A note about Step #4

- There are alternate ways to express the internal logic of a module.
  - `assign` statements.



```
and(Y, A, B);
assign Y = A & B;
```

```
or(Y, A, B);
assign Y = A | B;
```

```
not(Y, A);
assign Y = ~A;
```

Verilog operators

- C and Python have operators, such as:
  - +, −, <, ==, etc.
- Verilog operators
  - “Bitwise” operations take multi-bit input values, and perform the operation on the corresponding bits of each value.
  - More operators exist, but this is enough for now.

Operator	Operation
~	Bitwise NOT (1’s complement)
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
!	NOT
&&	AND
||	OR
==	Test equality

Module Example, revisited

```
module xor_gate(A, B, Y);
  input A, B;
  output Y;

  assign Y = A & ~B | B & ~A;
endmodule
```

- This also works, but can be easier to express.

Using modules

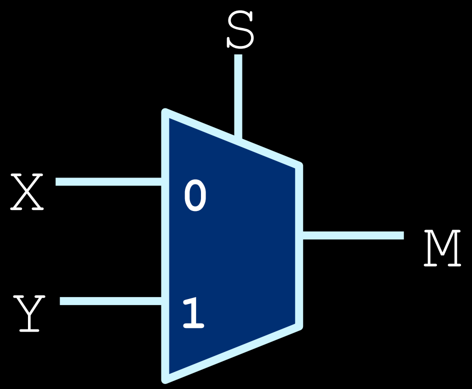
- Once a module is created, it can be used as a component of other modules that you create.
    - Example: half adder circuit.

```
module half_adder(X, Y, C, S);
   input X, Y;
   output C, S;

   and(C, X, Y);
   xor_gate(S, X, Y);
endmodule
```

Making a mux in Verilog

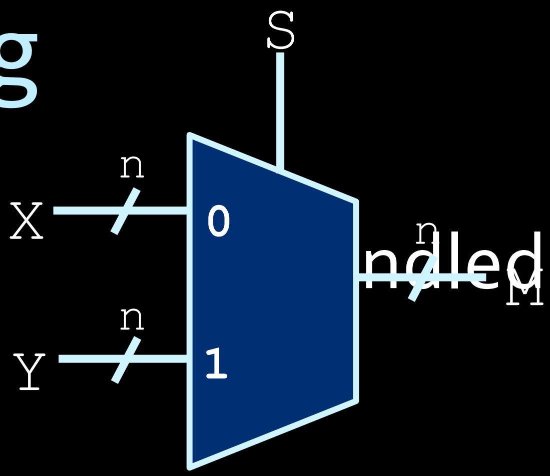
- Remember how a mux works:

$$M = X \cdot \overline{S} + Y \cdot S$$



```
module mux(X, Y, S, M);
   input X, Y, S;
   output M;

   assign M = X & ~S | Y & S;
endmodule
```

3-bit mux in Verilog



- How are multiple inputs ndled by Verilog?
  - e.g. 3-input multiplexers.
- Use square bracket characters to indicate a range of values for that signal.

```
module mux(X, Y, S, M);
  input [2:0] X, Y;   // 3-bit input
  input S;            // 1-bit input
  output [2:0] M;     // 3-bit output
  ...
```

3-bit mux in Verilog

- Continuing 3-bit mux example:

```
module mux(X, Y, S, M);
  input [2:0] X, Y;  // 3-bit input
  input S;          // 1-bit input
  output [2:0] M;  // 3-bit output

  assign M[0] = X[0] & ~S | Y[0] & S;
  assign M[1] = X[1] & ~S | Y[1] & S;
  assign M[2] = X[2] & ~S | Y[2] & S;
endmodule
```

A note about ranges

- When indicating that a labeled signal represents several input wires, the notation for the range can vary:
  - e.g.

    ```
    input [2:0]  X, Y;   or
    input [0:2]  X, Y;
    ```
- Both are legal; the first means that the first bits of the inputs are referred to as `X[2]` and `Y[2]`. The second means that the first bits of the inputs are referred to as `X[0]` and `Y[0]`.

End of combinational design

- Everything so far has been about specifying combinational circuits. The next section will discuss the elements of Verilog that arise when you incorporate sequential circuits.
- If you’ve been reading all these slides in one sitting, this is a good time to stand up, get something to eat or drink, and stretch while you absorb the things you’ve just read.

try some delicious and healthy snacks
at our concession stand
Intermission

Reflections on Verilog

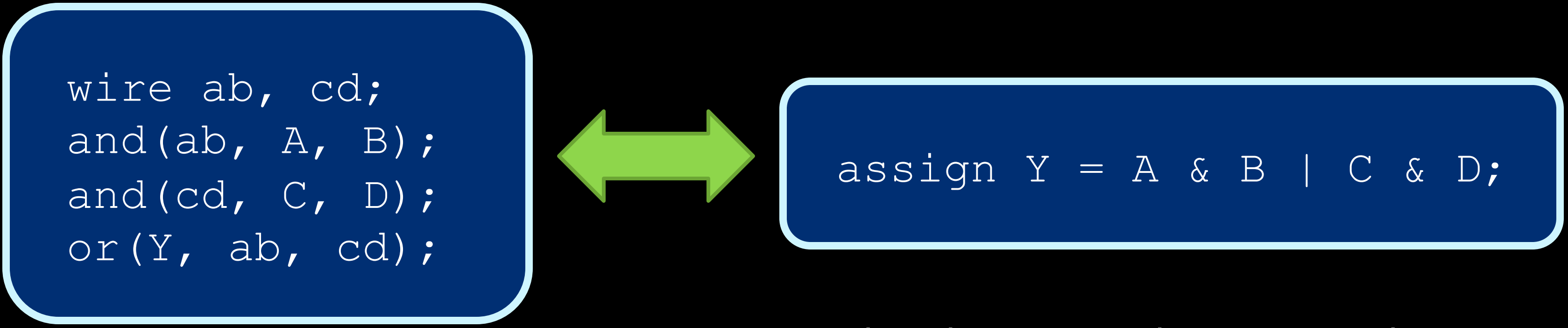
- By now, you’ve seen several elements of the Verilog language, but it’s good to put them into perspective again.
- **Verilog is used to specify circuit diagrams.**
  - If you had to describe a circuit diagram in text form, the result would strongly resemble Verilog syntax.

“I need two AND gates, one with inputs A and B, the other with inputs C and D, with an OR gate whose inputs are the outputs of the two AND gates and with an output called Y…”

```
wire ab, cd;
and(ab, A, B);
and(cd, C, D);
or(Y, ab, cd);
```

More reflections on Verilog

- If you can describe a circuit diagram, you have the ability to specify your circuit in Verilog!
- Verilog also tries to provide shorter syntax for describing certain circuits.
  - Example: Simple boolean circuits:



```
wire ab, cd;
and(ab, A, B);
and(cd, C, D);
or(Y, ab, cd);
```

```
assign Y = A & B | C & D;
```

**Quartus makes these equivalent at compile time!*

Other simplification syntax

- What about sequential circuits?
  - i.e. circuits involving flip-flops.
- One way to specify them is using gate design:

```
 // A gated D latch
module d_latch (D, Clk, Q);
   input D, Clk;
   output Q;

   wire R, S, Qa, Qb /* synthesis keep */ ;

   nand (R, D, Clk);
   nand (S, ~D, Clk);
   nand (Qa, R, Qb);
   nand (Qb, S, Qa);

   assign Q = Qa;
endmodule
```

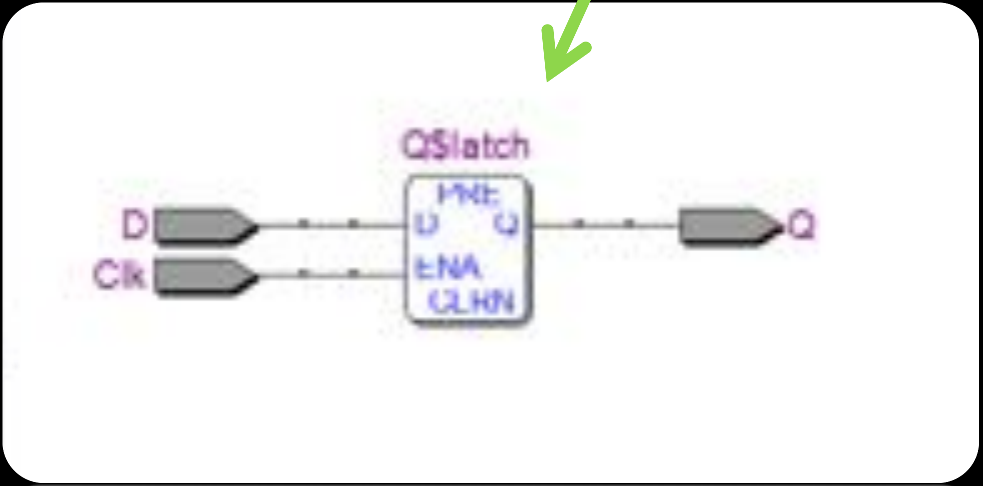
Other simplification syntax

- Another way of specifying this latch is:
  - *“Set Q to the value of D whenever Clk is high.”*
    - This is based on the concept of polling, where the software keeps checking the values of D and Clk to see if the output Q needs to be updated.
    - The better version is based on interrupts, where the circuit reacts when certain signal values change…
  - *“ Whenever D or Clk change, do the following.”*

```
always @ (D, Clk)
   begin
   ...
   end
```

Using `always`

- The `always` keyword provides even more abstraction than boolean equations, but in the end, everything is translated into simple gates.
- Other related high-level keywords:
  - `if/else`
  - `for`
  - `case`

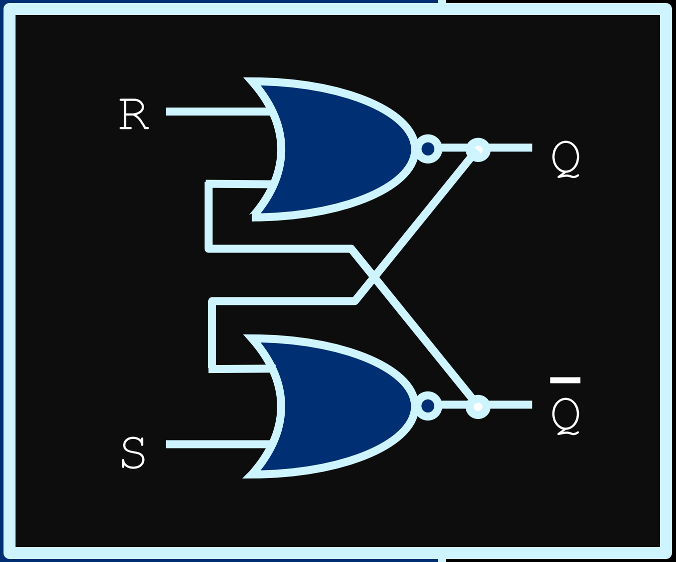


Using `always`

- Best to show `always` through examples.
  - Example #1: An RS latch.
    - If `R` and `S` are `0` and `0`, output is unchanged.
    - If `R` and `S` are `1` and `0`, `Q` is set high and $\overline{\texttt{Q}}$ is set low.
    - If `R` and `S` are `0` and `1`, `Q` is set low and $\overline{\texttt{Q}}$ is set high.
    - If `R` and `S` are `1` and `1`, `Q` and $\overline{\texttt{Q}}$ are both set low.

Always example: SR latch

```
module sr_latch(R, S, Clk, Q, Q_not);
   input R, S, Clk;
   output reg Q, Q_not;

   always @ (R, S, Clk)
      if (Clk & (R | S))
         begin
         if (R & ~S)
            Q_not = 1;
         else
            Q_not = 0;
         if (S & ~R)
            Q = 1;
         else
            Q = 0;
         end

endmodule
```



Using if, else & else if

- The `if` and `else` keywords are most useful within an `always` block.
  - Using them outside the `always` block doesn’t work.
  - The `if/else` statements are meant to turn signals on or off in response to certain conditions. The `always` statement is needed to tell the Verilog compiler when to test for those conditions.
- The `if`, `else if` and `else` statements can be nested, just like any other language.

if/else, begin and end

- Just like in other languages, the `if` condition can be followed by a single statement.
- If you want the circuit to perform multiple statements in response to a condition, use the `begin` and `end` keywords, like braces in C or Java.
  - Note: If you forget to use `begin` and `end`, the compiler will assume that only the next line belongs in the `if` statement.

```
if (S & ~R)
   Q = 1;
else
   Q = 0;
```

```
if (S & ~R)
   begin
   Q = 1;
   Q_not = 0;
   end
```

`always and initial`

- The `always` keyword tells the compiler to listen for conditions in certain signals, and make the circuit respond in those cases.
- The `initial` keyword does something similar (makes the circuit perform a set of operations), but only when the circuit starts.
  - Only occurs once, at beginning.
  - Useful for initiating circuit elements.

for loops

- The `initial` keyword and `for` loops go well together, as a result.
  - `for` loop syntax is similar to C and Java:

```
reg [15:0] memory [255:0];
integer index;

initial
   begin
   for(index = 0; index < 256; index = index + 1)
      begin
      memory [index] = 0;
      end
   end
```

case statements

- One more important control statement!
- The case statement works similarly to its Python, C and Java counterparts:

```
case (<variable name>)
   < case1 > : < statement >
   < case2 > : < statement >
   ...
   default : < statement >
endcase
```

- Good for creating decoders, or reacting to inputs that could have many possible values.

case example: 7-seg decoder

- Easier to specify decoders like the seven-segment decoder using `case` syntax.
  - Result is the same as the version obtained through K-maps and reduction.

```
module SevenSegmentDisplayDecoder(ss_out, bcd_in);
  output reg [6:0] ss_out;
  input [3:0] bcd_in;

  always @(bcd_in)
    case (bcd_in)
      4'h0: ss_out = 7'b0111111;
      4'h1: ss_out = 7'b0000110;
      4'h2: ss_out = 7'b1011011;
      4'h3: ss_out = 7'b1001111;
      4'h4: ss_out = 7'b1100110;
      4'h5: ss_out = 7'b1101101;
      4'h6: ss_out = 7'b1111101;
      4'h7: ss_out = 7'b0000111;
      4'h8: ss_out = 7'b1111111;
      4'h9: ss_out = 7'b1100111;
      4'hA: ss_out = 7'b1110111;
      4'hB: ss_out = 7'b1111100;
      4'hC: ss_out = 7'b0111001;
      4'hD: ss_out = 7'b1011110;
      4'hE: ss_out = 7'b1111001;
      4'hF: ss_out = 7'b1110001;
      default: ss_out = 7'b1111001;
    endcase
endmodule
```

Reflection time

- Remember: This course is about circuits, not Verilog ☺
- However…designing and testing these circuits is easier with tools or specification languages.
    - Designing complex circuits by wiring chips and gates together can take forever and be full of bugs.
    - Designing gate-by-gate in Verilog is faster, reproducible, and yields fewer bugs.
    - More powerful keywords like `always`, `if/else` and `case` create the same logic with less work, by simply specifying the behaviour that you want instead of the specific circuit.
        - Like specifying string and dictionary behaviour in Python or C, instead of creating your own objects from scratch.

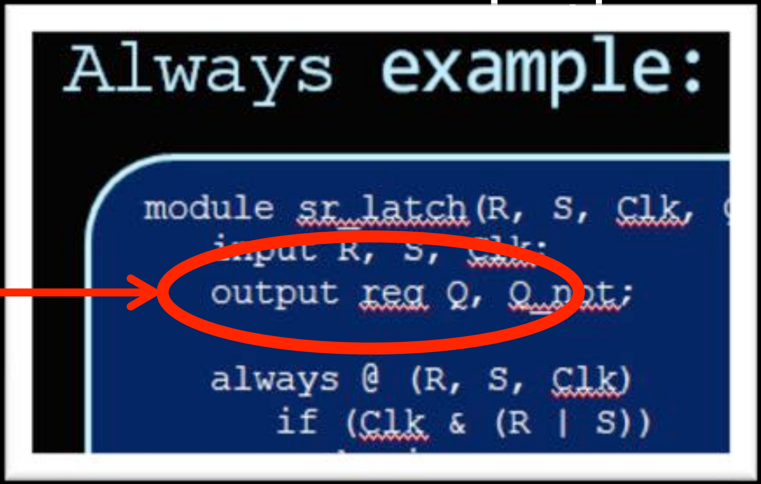
Reflection time

- Take a moment to think about this! People often get unduly distressed about higher-level Verilog statements.
- Really though, these make life once you of using say be.

Storing and assigning data

- Flip-flops and sequential circuits allow us to design circuits that can store values.
  - Some are implicit, in latch-style feedback circuits.
  - Others can be made explicit, using `reg` values.
- We've used reg before,

  SR latch example.

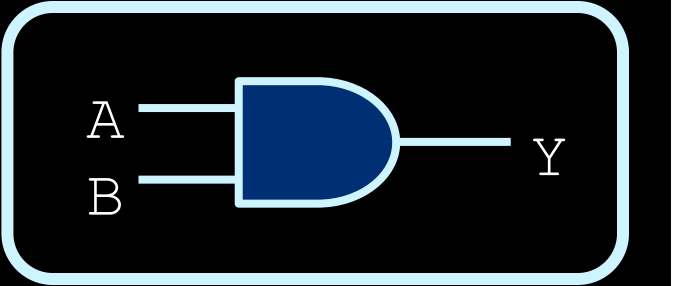


Always example:

```
module sr_latch(R, S, Clk,
   input R, S, Clk;
   output reg Q, Q_not;

   always @ (R, S, Clk)
      if (Clk & (R | S))
```

Storing and assigning data

- Labels can be given to both `wire` and `reg` values.
  - Main difference: `reg` values can drive output; `wire` values can't.



```
module wire_example( a, b, y);
  input a, b;
  output y;
  wire a, b, y;

  assign y = a & b;

endmodule
```

```
module reg_example( a, b, y);
input a, b;
output y;

reg   y;
wire a, b;

always @ ( a or b)
begin
  y = a & b;
end

endmodule
```

Reg versus wire

- "Driving output"? What does that mean?
  - Remember that Verilog is used to specify a circuit layout. The `wire` keyword specifies where an actual wire would be, and connecting two points with a wire (or `wire`) just makes them logically equivalent.
  - A `reg` (short for register) can actually store a value and drive an output signal. Any `always` or `initial` blocks that need to set a value must do so by setting `reg` values.
    - Similarly, wires can't be reassigned to connect new points together partway through a circuit's operation.

Literal values

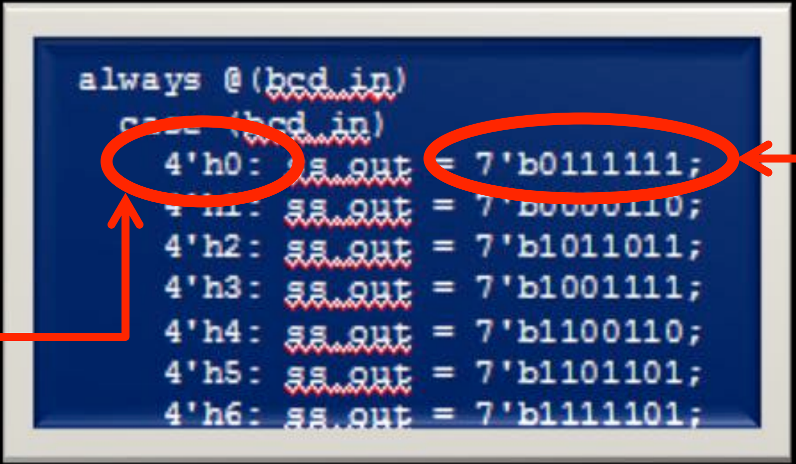
- Sometimes you need to assign a set of wire or reg values at the same time.
  - Example: the seven-segment display.
  - Possible to assign each segment one at a time, but easier to assign them all at once.
- Literal values in Python and C are values like `42`, `3.14` and “`Hello`”.
- Verilog has literal values as well.
  - But exactly what does `101` represent in Verilog?

Literal values

- 101 can represent different values, depending on what type of number it is.
  - Decimal: 101 = one hundred and one (base 10).
  - Binary: 101 base 2 = 5 base 10
  - Hexadecimal: 101 base 16 = 257 base 10
  - (Octal: 101 base 8 = 65 base 10)
- In Verilog:
  - Binary: `‘b101`
  - Decimal: `‘d101`
  - Hexadecimal: `‘h101`

Example: seven-seg decoder

- The instructions in the seg decoder translate to the following:



```
always @(bcd_in)
  case (bcd_in)
    4'h0: ss_out = 7'b0111111;
    4'h1: ss_out = 7'b0000110;
    4'h2: ss_out = 7'b1011011;
    4'h3: ss_out = 7'b1001111;
    4'h4: ss_out = 7'b1100110;
    4'h5: ss_out = 7'b1101101;
    4'h6: ss_out = 7'b1111101;
```

  - “in the case where the four-bit BCD input has a hexadecimal value of `0`…”
  - “…set the value `ss_out` to a 7-digit binary value of `0111111`.”

References

- Verilog Language Reference Guide:
  - http://verilog.renerta.com/
- Verilog Page:
  - http://www.asic-world.com/verilog/
- Verilog Tutorial:
  - http://www.fullchipdesign.com/verilog_tutorial.htm