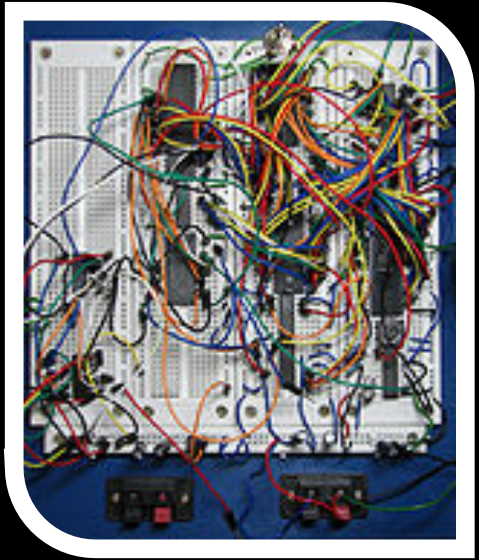
Week 2 Tutorial: The Verilog Primer

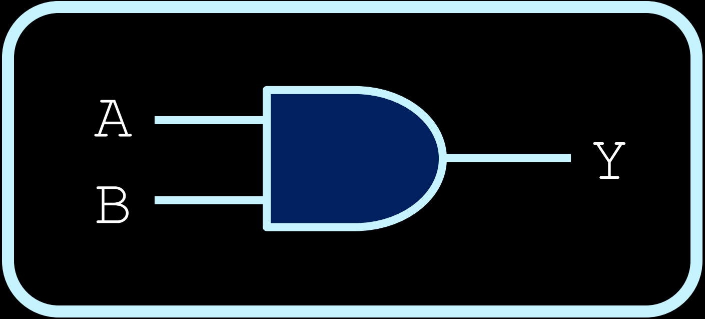
Understanding Verilog

- 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 circuit layout needs to look like, once we start designing circuits that are too large or complicated to implement with actual chips and wires.



Basic Verilog Example

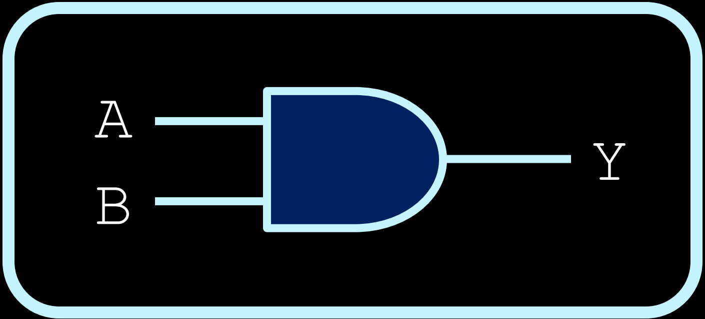
- For instance, this is a simple AND gate:



- 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 name that you could use to 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)`	

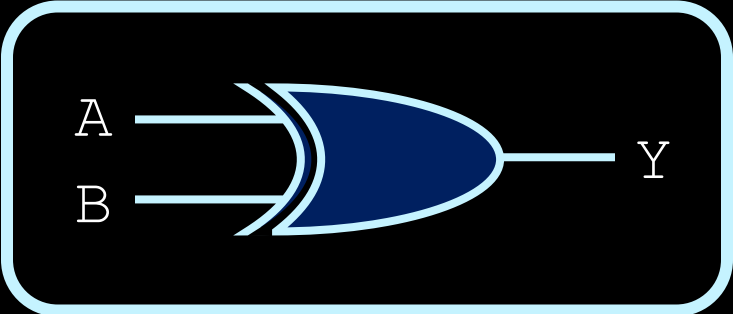
  - two-input gates shown here, but multi-input also possible.

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:

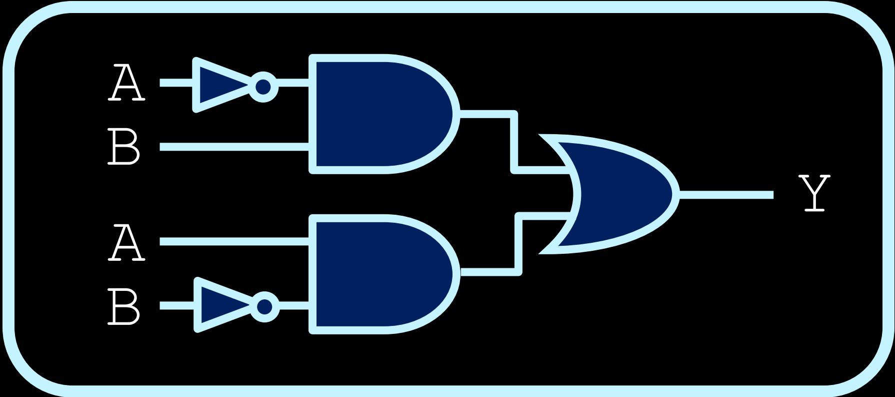


$$Y = A \cdot \overline{B} + \overline{A} \cdot B$$

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

Module Example (cont’d)

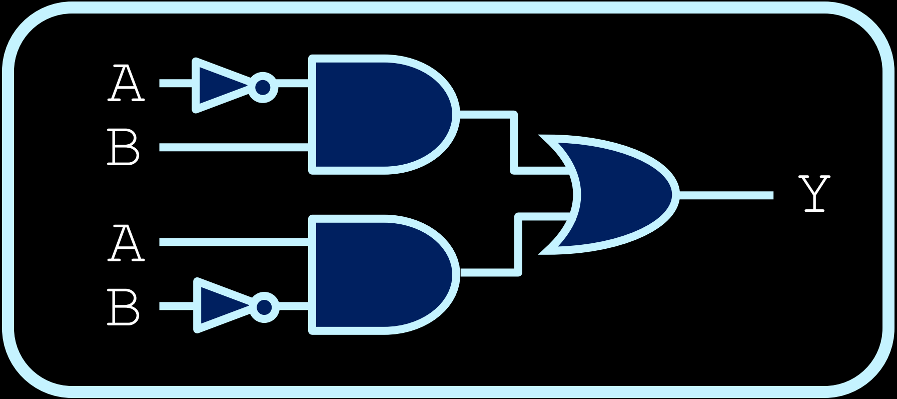
- Not to hard to represent it in logic gates.



- How would you specify the AND and OR gates?

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

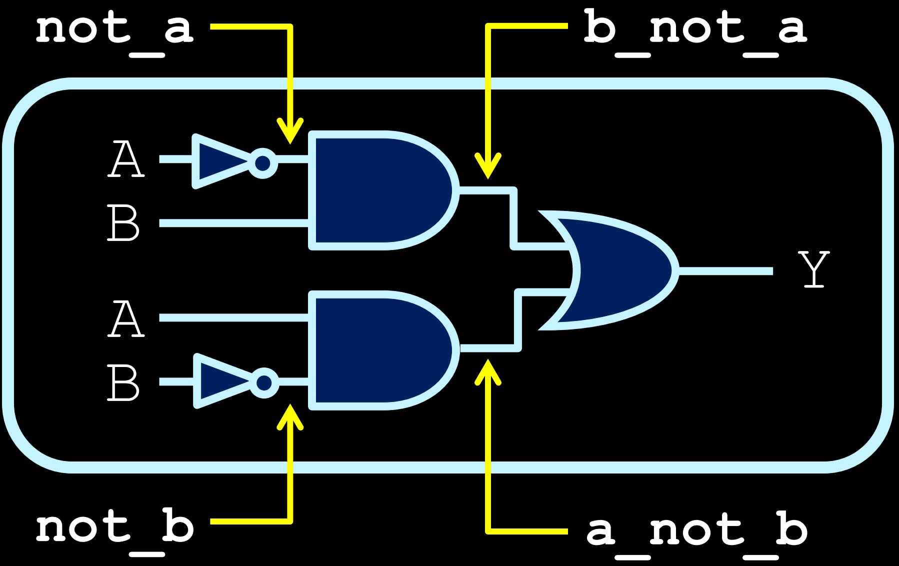
Module Example (cont'd)



```
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)



- <u>Note:</u> 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

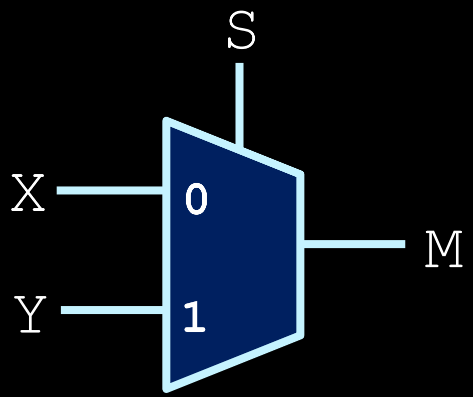
- Once a module is created, it can be used as a component of other modules that you create.
  - Example: half adder circuit.
  - C = X AND Y   S = X XOR Y

```
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

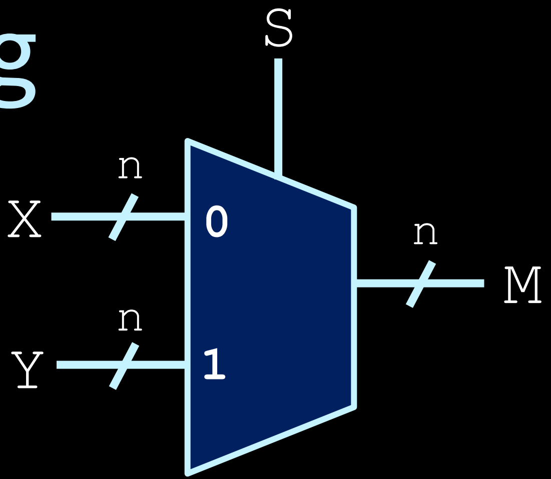
$$M = X \cdot \overline{S} + Y \cdot S$$



```
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 handled 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]`.