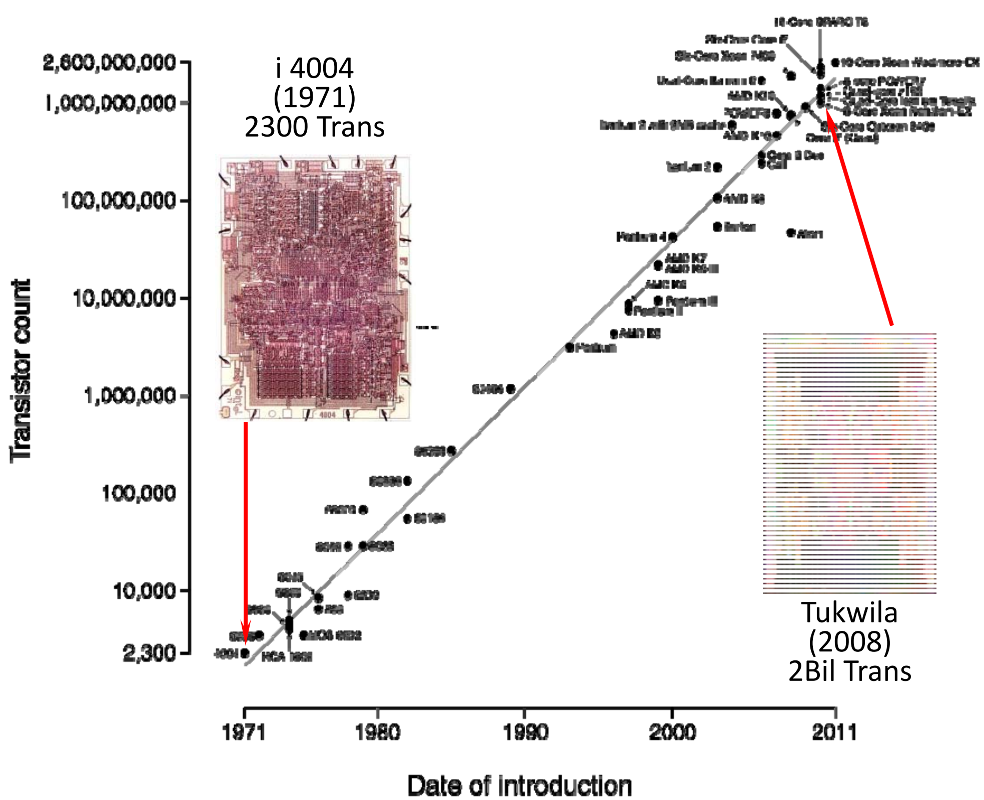
CX1005
Digital Logic

Introduction to Verilog

Where is this going?

- Designs are getting bigger and more complex
  - Designer productivity is an issue



So is working with individual gates going to allow you to design a 2 billion transistor CPU?

What about using Schematic Capture? Will that allow you to design a 2 billion transistor CPU?

So we need better Design Methods and Tools (EDA and CAD tools) to design large complex systems.

Verilog and the **FPGA tools** are starting you along this path…

Next, we will cover:

- Introduction to Verilog HDL
  - Verilog is a hardware description language (HDL) and has some similarities to programming languages
- Combinational circuits in Verilog
  - Combinational circuits are circuits where the current outputs depend only on the current inputs
  - So far you have only looked at gate level combinational circuits
- Sequential circuits
  - Sequential circuits are circuits where the outputs depend on both the present input signals and on the past history of the inputs. That is it contains memory.
  - We will only consider synchronous (clocked) circuits.
- State machines
  - A state machine is a sequential circuit with a finite number of states which can be used to control the operation of physical devices

Hardware Description Languages (HDLs)

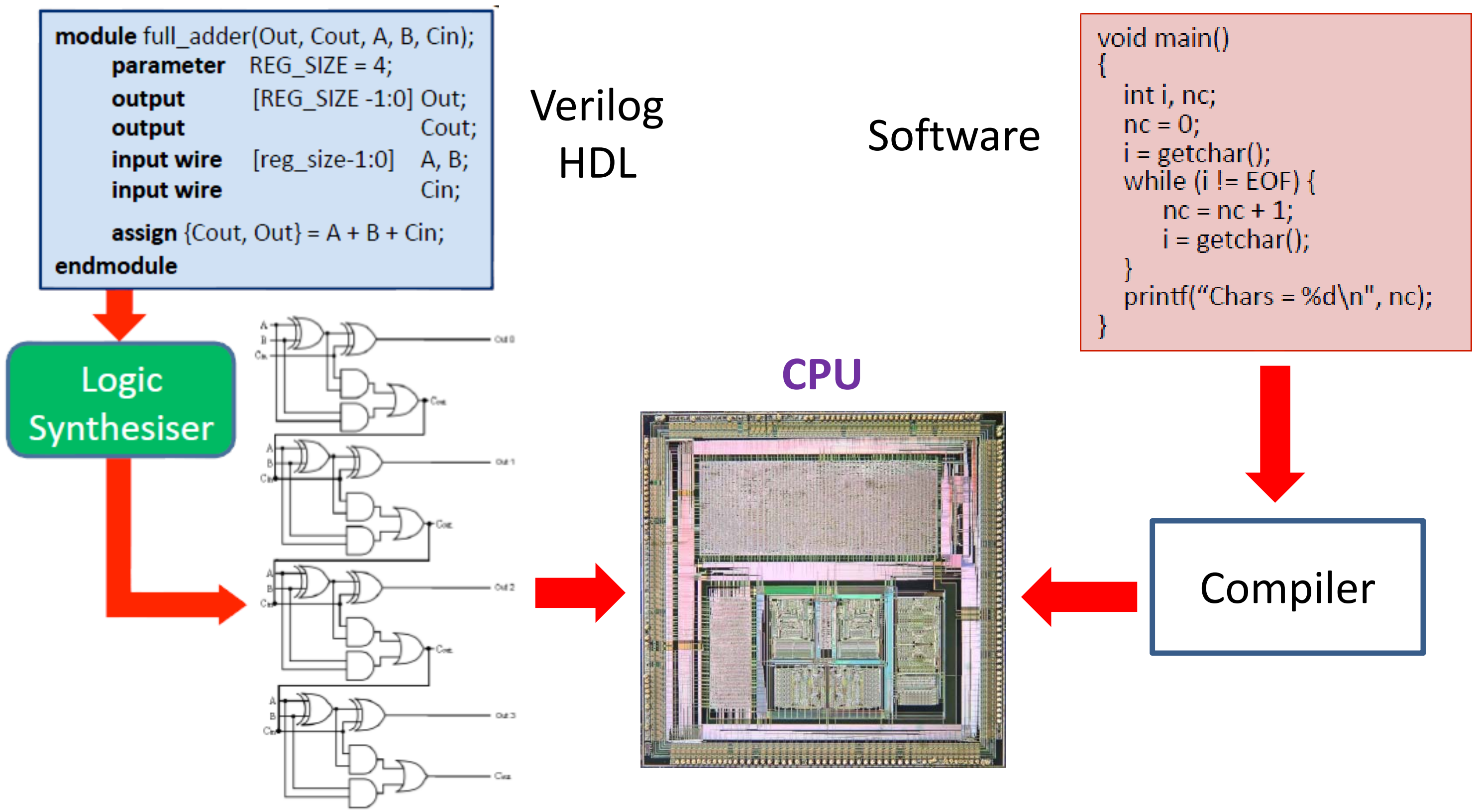
- Special languages with special constructs for describing hardware
- They can be used to describe hardware
  - At various levels of abstraction
  - Enables hierarchical design
- Sophisticated tools can then ensure the hardware generated from the description is efficient

```
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)

Hardware Description Languages (HDLs)

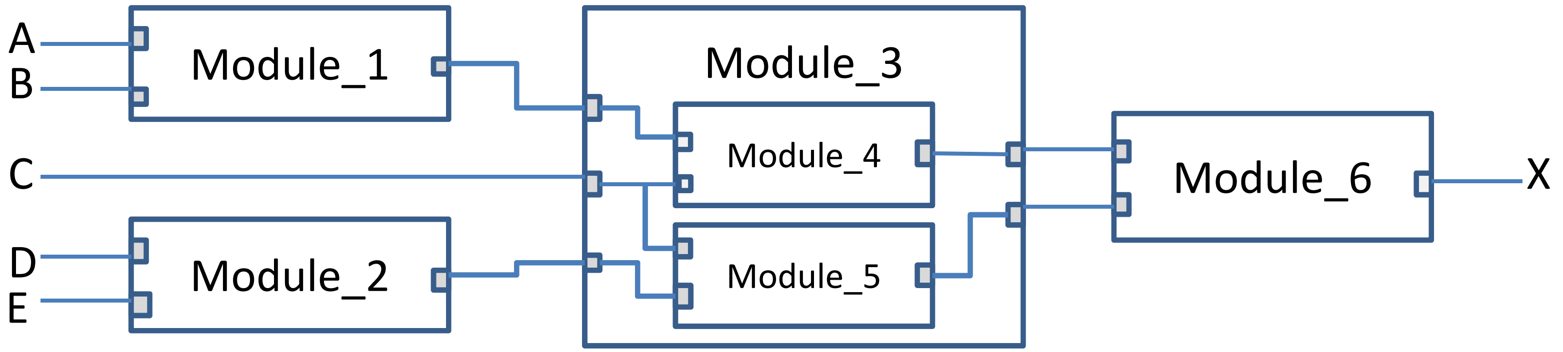
- HDLs are **programming-like** languages that are used to describe hardware
- HDLs are synthesized (and optimized) to hardware primitives
  - Note: Software is compiled to primitive instructions



MODULES

Module

- In Verilog, designs are broken down into ***modules***
- A ***module*** is a ***container*** the designer can use to ***encapsulate*** a unit of functionality
- Modules can contain code to describe hardware and also instances of other modules
- Good designs consist of sufficient (but not excessive) ***levels of hierarchy***, with modules containing instances of modules, that contain instances of other modules
- At each level in the hierarchy, a module instance is treated as a “black-box” – the internals are unknown



Verilog Module Declaration

- We declare a module in Verilog using the **module** keyword and a list of ports:

```
module somename (port1, port2, port3);
```

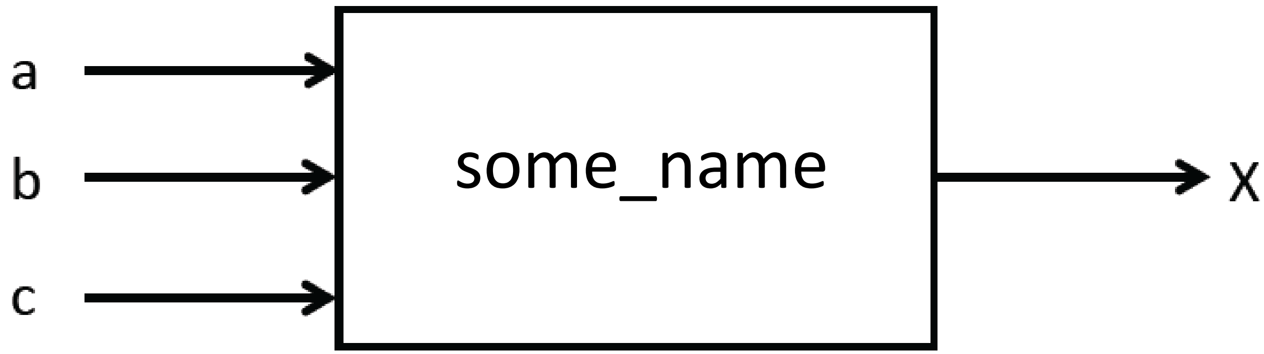
  - The above declaration describes a module with three **ports**, each a single wire

- We can indicate the direction of the ports using the keywords **input** and **output**:

```
module somename (input port1, port2,
                 output port3);
```

  - The above describes a module with two inputs and one output

Verilog Module Declaration




```
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

Structural Design using Verilog

- Our first experience with Verilog will be for ***structural design***
- This allows us to ***structurally*** describe any circuit we might otherwise use a circuit diagram for
- The principles learnt in the first half of the course allow us to describe any circuit in terms of these gates

X	Y	C_i	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Truth table for Full Adder

$$S = X \oplus Y \oplus C_i$$

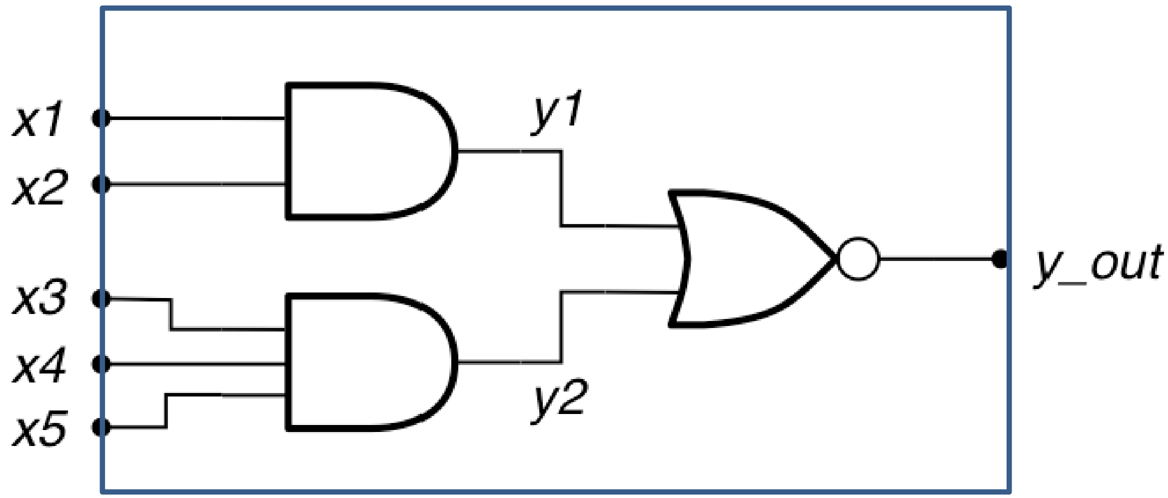
$$C_o = X.Y + X.C_i + Y.C_i$$

Minimize using Boolean Algebra or K-Maps

```
module full_adder (
      input X, Y, Ci,
      output S, Co);

  // Describe your structural
  // design for full adder here

endmodule
```

Structural Design using Verilog

- Describing a circuit by its internal structure
- To do this, we need:
  - A module definition – *done*
  - Some gates
  - Wires to connect those gates



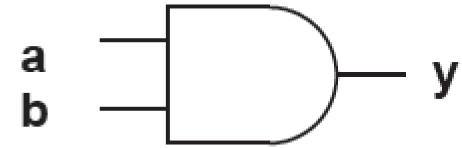
We are trying to 'draw' the circuit using codes

INSTANTIATING GATE-LEVEL PRIMITIVES

Gate-Level Primitives

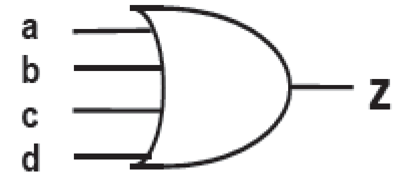
- Verilog provides us with basic ***primitives*** to model Boolean gates: and, nand, or, nor, not, xor, xnor

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



- Represents an ***and*** gate with inputs connected to wires a and b, and output connected to wire y
- Gate primitives allow more than two inputs:

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



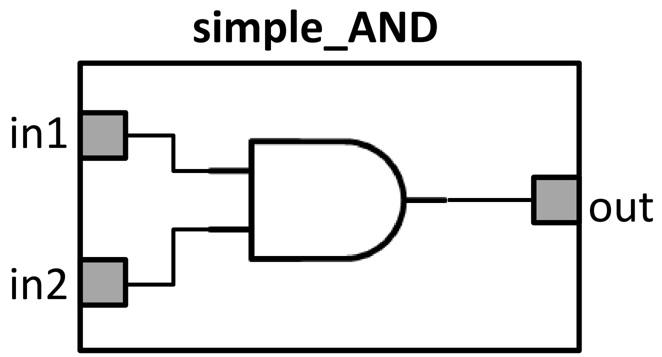
- This represents an ***or*** gate with inputs a, b, and c, d and output z

The (single) output is always the first argument

Verilog Module with Gate Level Primitives

- A simple module that contains just a two-input ***and*** gate would be written like this:

```
module simple_AND (
       input in1, in2,
       output out);

       and (out, in1, in2);

endmodule
```



- The module's ports act the same as wires (but you **cannot** connect the output of a gate to an input port!)

  Of course, we wouldn't create a module for a single *and* gate since the primitive already exists

Wires

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

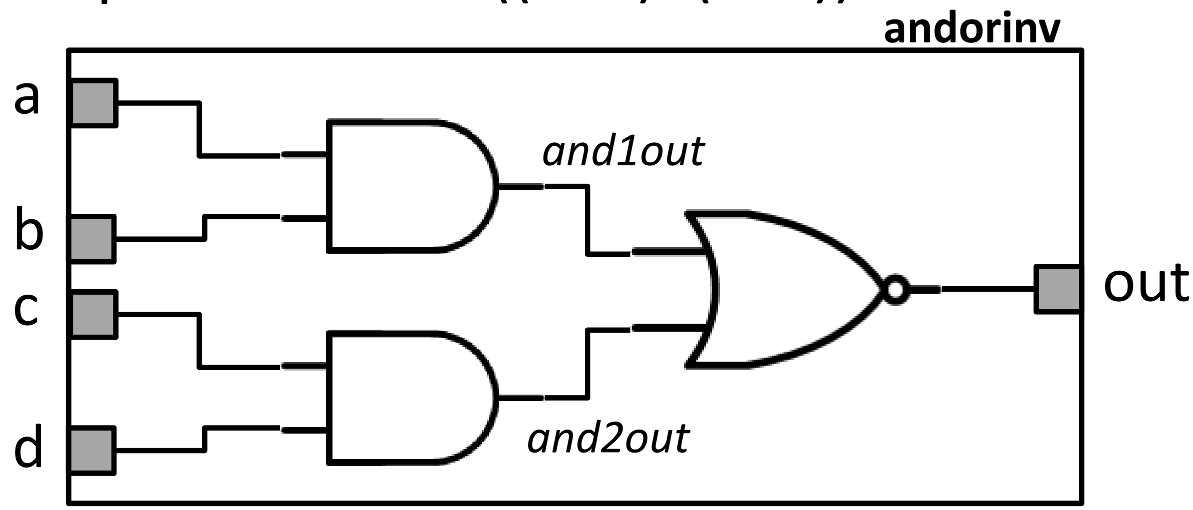
```
wire int_signal;
```

int_signal

- This creates a named (1-bit) wire that can be used to connect gates together
  - Note: we will see later that it is not always necessary to declare 1-bit wires

Verilog Module with Gate Level Primitives

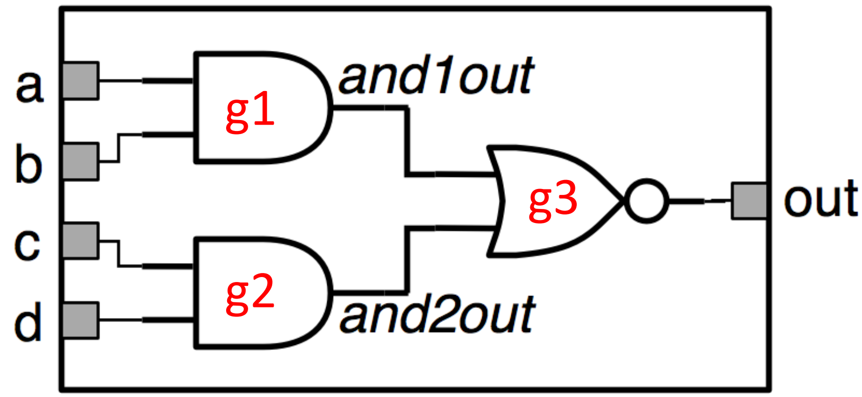
- Let us consider an ***and-or-inverter***
- Boolean equation: out = ((a•b)+(c•d))’




```
module andorinv (input a, b, c, d,
                 output out);
      wire and1out, and2out;
      and (and1out, a, b);
      and (and2out, c, d);
      nor (out, and1out, and2out);
endmodule
```

Verilog Module with Gate Level Primitives

- Note you can also use of a unique ***identifier*** for each gate (this can help in testing):



Gate Identifiers

```
module andorinv (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
```

Verilog Module with Gate Level Primitives

- When declaring module ports of the same type and direction, no need for separate declarations, can just use a comma between names:

```
module hmm (input in1,
            input in2,
            input in3,
            output out);
```

```
module hmm (input in1,
            in2, in3,
            output out);
```

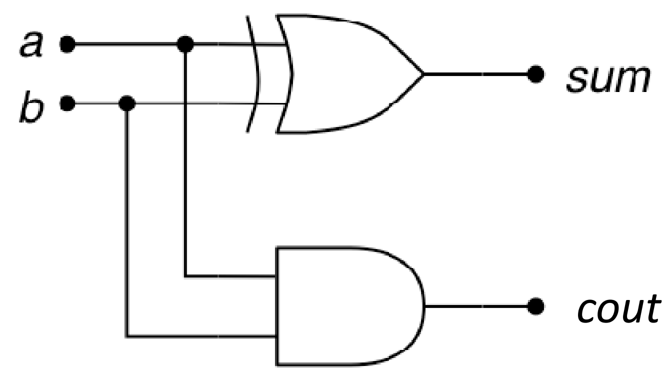
- Also, when declaring multiple wires of the same type, there is no need for separate declarations, use a comma between names:

```
wire red;
wire blue;
wire green;
```

```
wire red, blue, green;
```

Example: Half Adder in Verilog

- Lets re-look at the binary adder
- A ***half-adder*** takes two 1-bit inputs and produces a 1-bit **sum** output and a 1-bit carry out (**cout**)
- We can see that **sum** is just an ***xor*** function and **cout** is just an ***and*** function



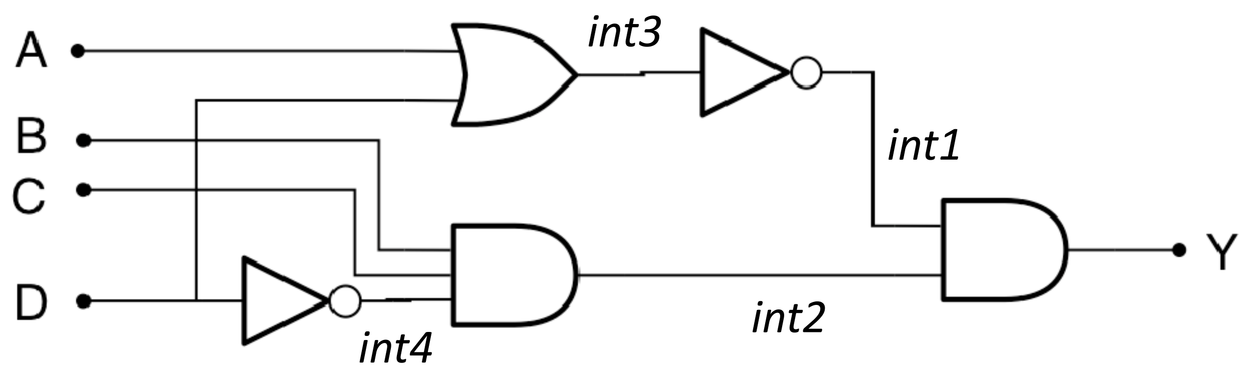
a	b	sum	cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

```
module add_half (
  input a, b,
  output sum, cout);

  xor g1 (sum, a, b);
  and g2 (cout, a, b);
endmodule
```

Note: g1 and g2 are gate identifiers

Exercise

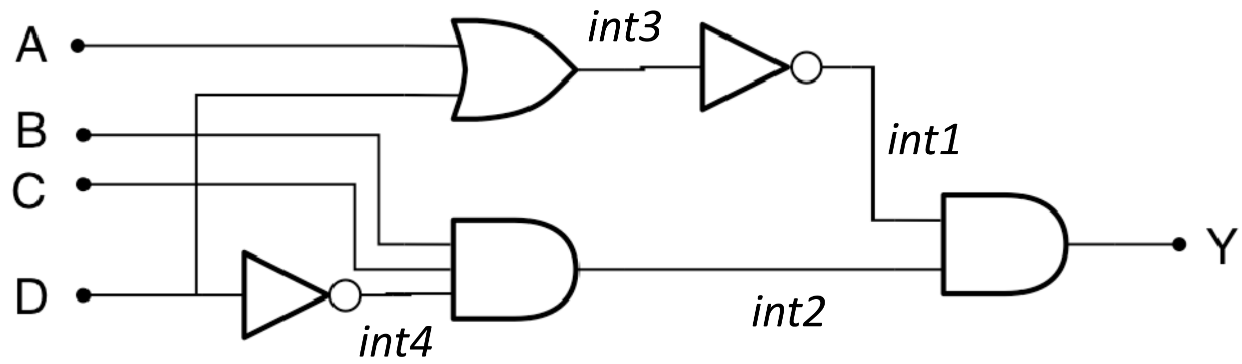
- Implement a Verilog module using structural gate-level primitives for the following circuit:




```
module simple_circ (input A, B, C, D, output Y);
   wire int1, int2, int3, int4;

   and (Y, int1, int2);
   not (int1, int3);
   or  (int3, A, D);
   and (int2, B, C, int4);
   not (int4, D);
endmodule
```

Exercise

- Implement a Verilog module using structural gate-level primitives for the following circuit:



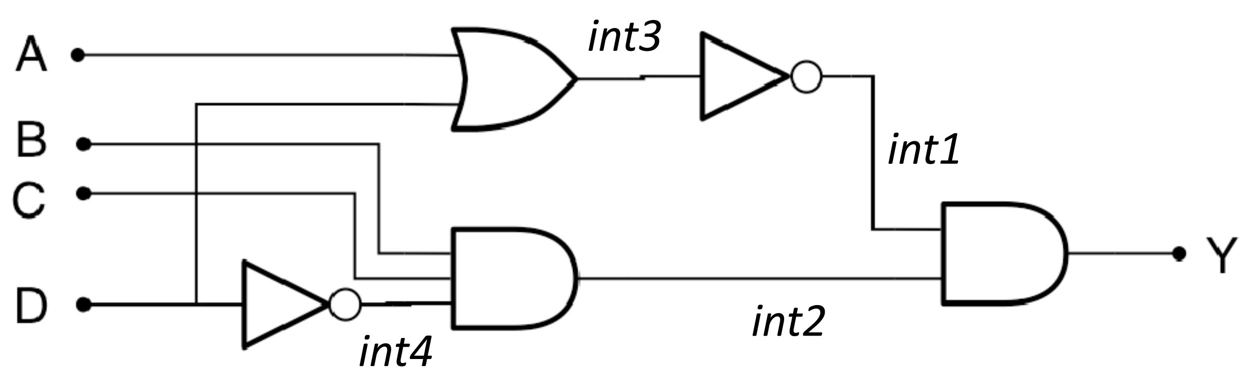

```
module simple_circ (input A, B, C, D,
                    output Y);

   wire int1, int2, int3, int4;

   and (Y, int1, int2);
   not (int1, int3);
   or  (int3, A, D);
   and (int2, B, C, int4);
   not (int4, D);
endmodule
```

```
module simple_circ (input A, B, C, D,
                    output Y);

   wire int1, int2, int3, int4;

   not (int1, int3);
   not (int4, D);
   and (int2, B, C, int4);
   and (Y, int1, int2);
   or  (int3, A, D);
endmodule
```

These two implementations will produce the same circuit

Verilog Module with Gate Level Primitives

- Note that the **order** of statements is **unimportant**
- Each statement describes a piece of hardware
- There is no sequence of steps when doing structural Verilog
- Usually we declare internal signals first, and then instantiate gates



A Note on Wires

- There is no need to declare 1-bit wires for connecting modules
- Tools will infer these wires automatically by matching names
- But **only** for 1-bit wires in instantiation:

```
module simple_circ (input A, B, C, D,
                    output Y);

   wire int1, int2, int3, int4;

   and (Y, int1, int2);
   not (int1, int3);
   or  (int3, A, D);
   and (int2, B, C, int4);
   not (int4, D);
endmodule
```

So, instead of this:

```
module simple_circ (input A, B, C, D,
                    output Y);

   and (Y, int1, int2);
   not (int1, int3);
   or  (int3, A, D);
   and (int2, B, C, int4);
   not (int4, D);

endmodule
```

This is OK!

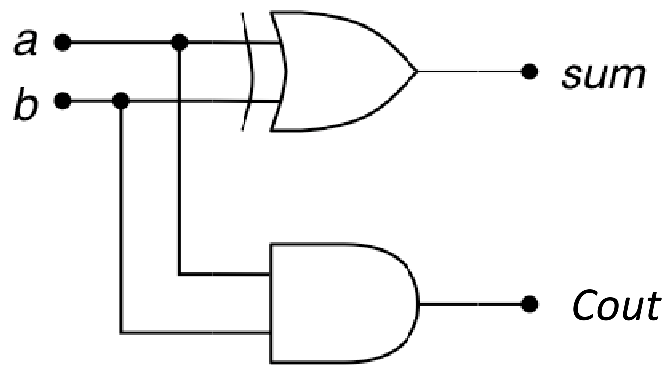
Additional Notes

- Verilog is **case-sensitive!**
  - FIFO_out, fifo_out and Fifo_Out refer to different things
- Statements are terminated with a semi-colon (;)
- Comments are ***C-style***:
  - // starts a single line comment
  - /* is used for block comments */ – cannot be nested!
- Whitespace is ignored – But you **should** use it to make your code readable
- ***Identifiers*** in Verilog:
  - Must start with **a letter** or **underscore** (_)
  - Can contain **letters**, **numbers**, **underscore**, and **dollar** ($)
  - Must not clash with keywords (e.g**. module** , **input**, etc.)

```
module and_or_inv (input a, b, c, d,
                   output out);

  wire and1out, and2out;

  and g1 (and1out, a, b); // AND gate
  and     g2 (and2out, c, d);
  nor g3 (out, and1out,    // NOR gate
          and2out);

endmodule
```

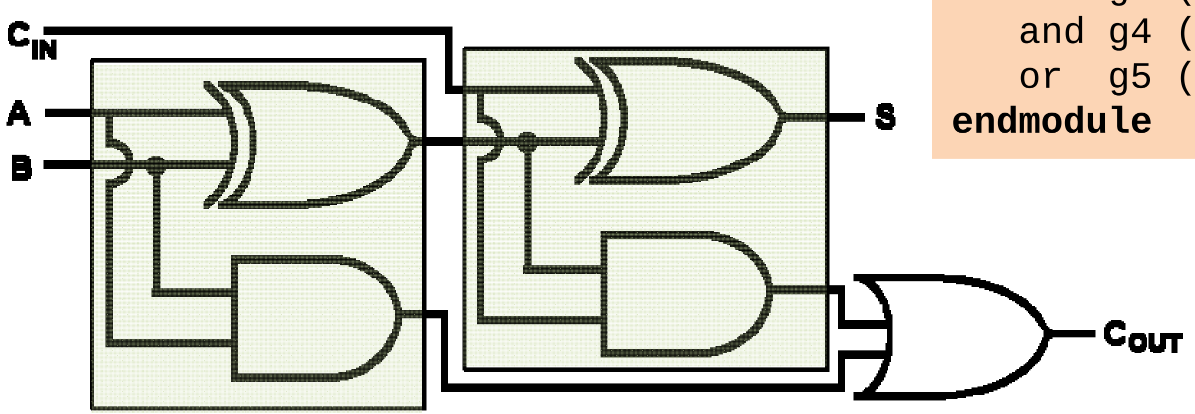
MODULE INSTANTIATION

Half Adder and Full Adder




```
module add_half (
   input a, b,
   output sum, Cout);

   xor g1 (sum, a, b);
   and g2 (Cout, a, b);
endmodule
```

A ***full adder*** can be implemented using 2 Half-Adders and an OR gate (see slides on Digital Arithmetic)

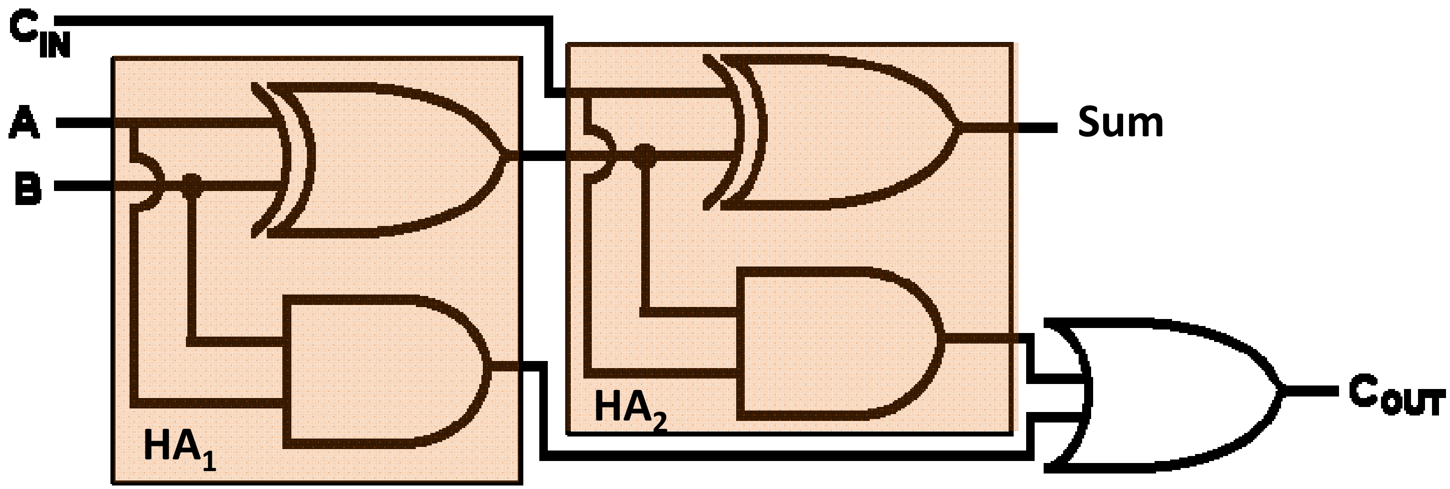


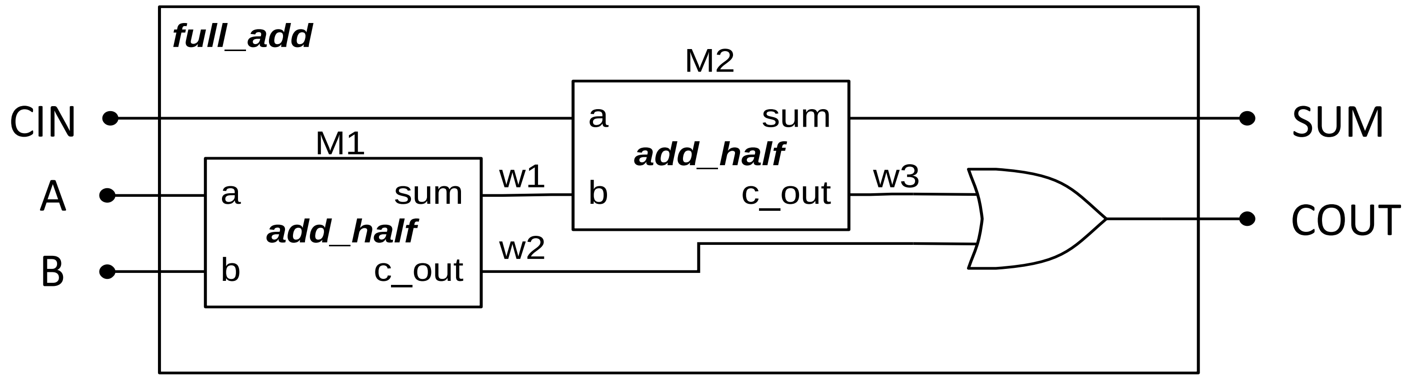

```
module full_add (input A, B, Cin
                 output sum, Cout);
   wire w1, w2, w3;

   xor g1 (w1, A, B);
   and g2 (w2, A, B);
   xor g3 (sum, Cin, w1);
   and g4 (w3, Cin, w1);
   or  g5 (Cout, w2, w3);
endmodule
```

This is tedious!

Instantiation in Verilog

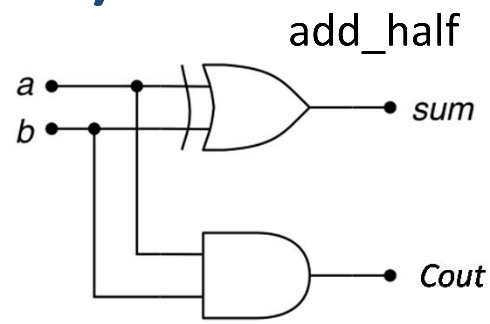
- In Verilog, we can use existing modules within new modules through ***instantiation***
- For example, we can ***instantiate*** the half adder module we previously designed **within** a new module

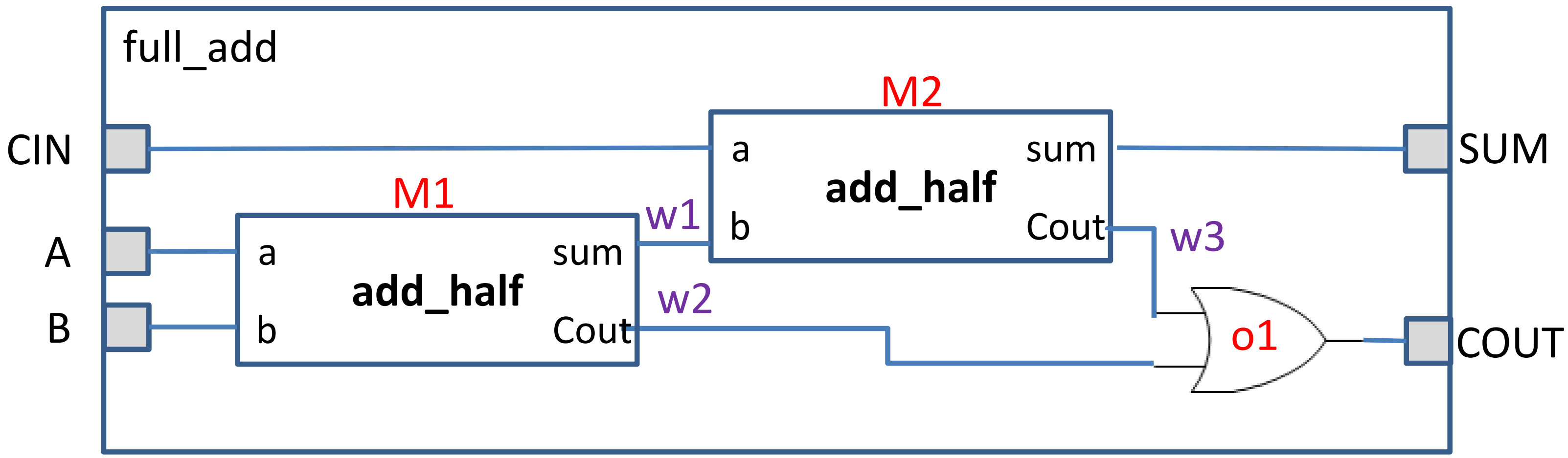




Instantiation in Verilog (Full Adder)

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





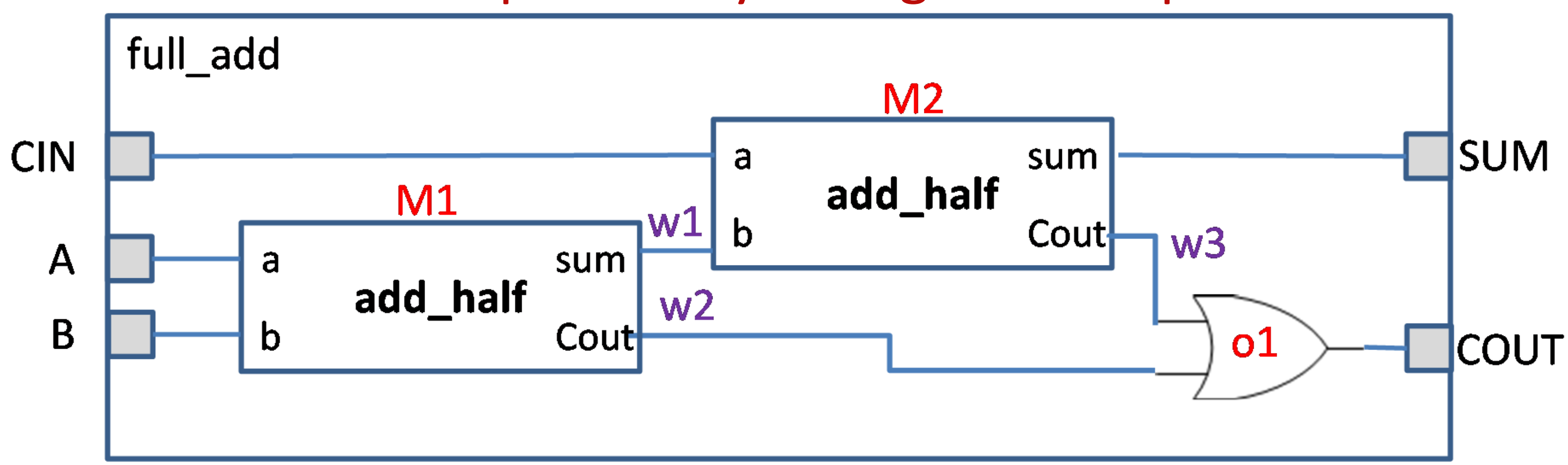

```
module full_add (input A, B, CIN,
                 output SUM, COUT);
    wire  w1, w2, w3;

    add_half M1 (A, B, w1, w2);
    add_half M2 (CIN, w1, SUM, w3);
    or o1 (COUT, w3, w2);
endmodule
```

Instantiation in Verilog

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

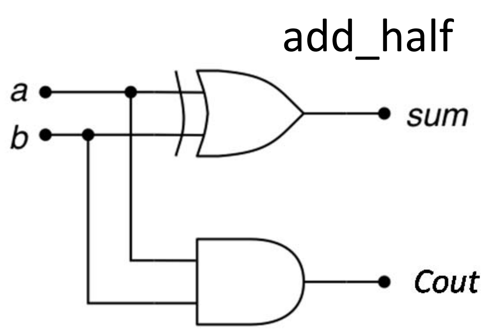
```
add_half M1 (A, B, w1, w2);
```

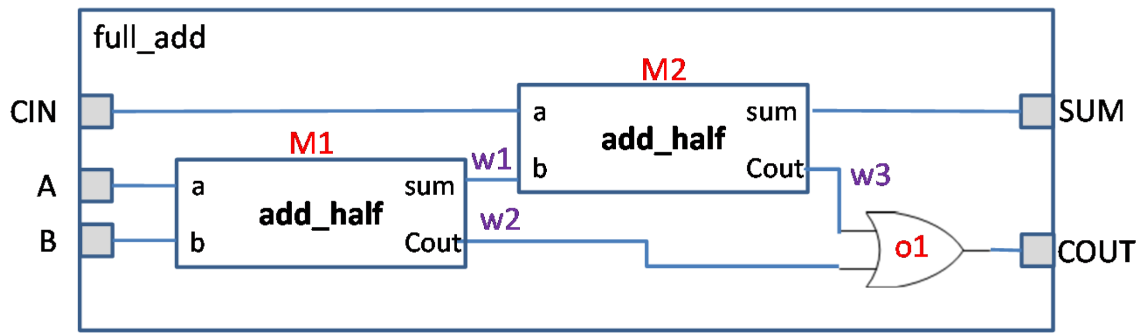
- Creates an ***instance*** called **M1** of module **add_half**
- It then **connects the signals and ports** referenced in the parentheses with the corresponding ports of the instantiated module
  - Just like we did previously with gate-level primitives



Ordered Instantiation

- The **order** determines connections
- Looking at the original **add_half** module declaration:
- Its first port is its ***a*** input, the second its ***b*** input, the third its ***sum*** output and the fourth its ***Cout*** output
- Connects a wire called ***A*** in the outer module with the ***a*** port, ***B*** with the ***b*** port, the ***w1*** wire to its ***sum*** output and the ***w2*** wire to its ***Cout*** port






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

```
add_half M1 (A, B, w1, w2);
```

```
add_half M2 (CIN, w1, SUM, w3);
```

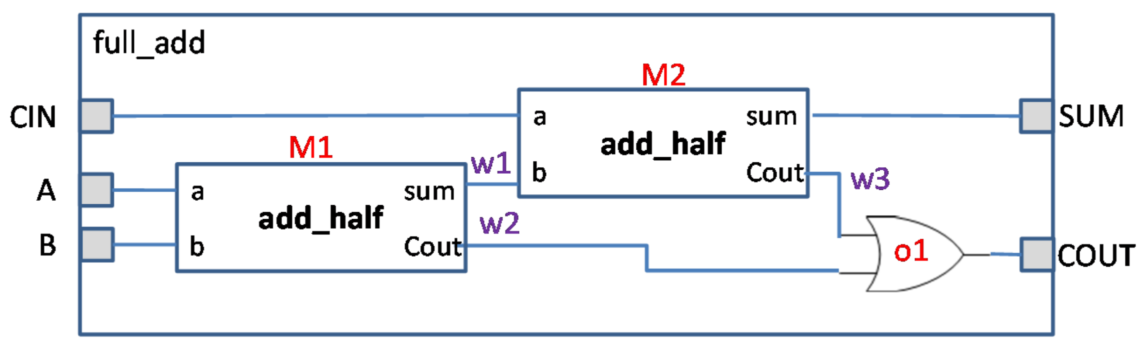
Named Instantiation

- Connecting instantiated modules to wires and ports in the manner just described can be **error-prone**:
  - Must remember the order of the ports
  - If we change the port order or add/remove ports, we have to change the instantiation
- Hence, we generally use a ***named connection***:

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

```
add_half M2 (.a(CIN),
             .b(w1),
             .sum(SUM),
             .Cout(w3));
```

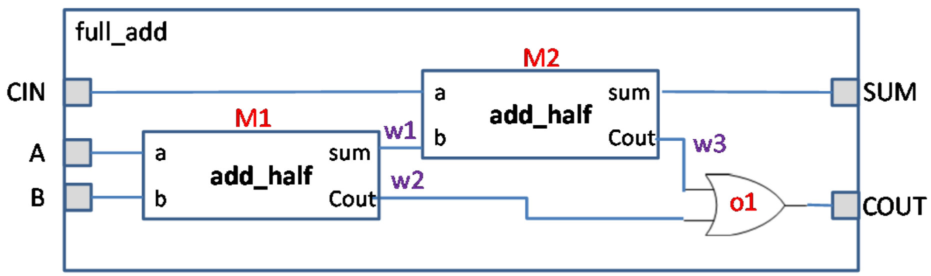
The port name is preceded by a dot, and the connected signal is placed in the brackets



Instantiation in Verilog

Ordered Instantiation

```
module full_add  (input A, B, CIN,
                  output SUM, COUT);
    wire  w1, w2, w3;

    add_half M1 (A, B, w1, w2);
    add_half M2 (CIN, w1, SUM, w3);
    or O1 (COUT, w3, w2);

endmodule


module add_half  (input a, b,
                  output sum, Cout);
    xor G1 (sum, a, b);
    and G2 (Cout, a, b);
endmodule
```



Named Instantiation

```
module full_add  (input A, B, CIN,
                  output SUM, COUT);
     wire  w1, w2, w3;

     add_half M1 (.a(A), .b(B),
                  .sum(w1),
                  .Cout(w2));

     add_half M2 (.a(CIN), .b(w1),
                  .sum(SUM),
                  .Cout(w3));

     or O1 (COUT, w3, w2);
endmodule

module add_half  (input a, b,
                  output sum, Cout);
     xor G1 (sum, a, b);
     and G2 (Cout, a, b);
endmodule
```

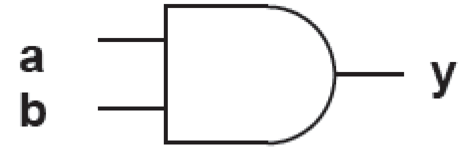
VERILOG ASSIGNMENTS

Verilog Assignments

- Implementing larger circuits using individual gates can be tedious.

Instantiating gate primitive

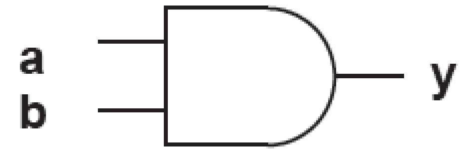
```
and (y, a, b);
```



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

Continuous assignment

```
assign y = a & b;
```



- This is called a ***continuous assignment***
  - As it is always permanently assigned
- It allows us to assign the result of a Boolean expression to a signal

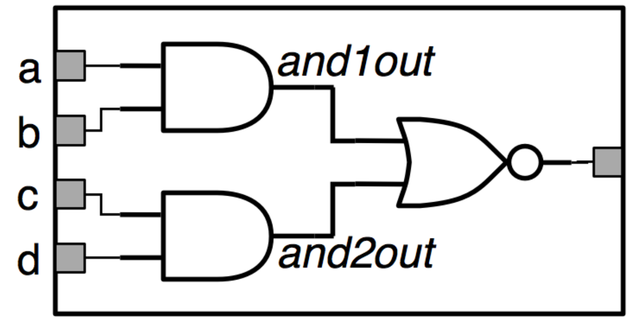
This is much simpler than using gate primitives

Verilog Assignments

- We can use a range of operators:
  - **&&** : and (bitwise is: **&**)
  - **||** : or (bitwise is: **|**)
  - **!** : not (bitwise is: **~**)
  - **^** : xor (bitwise is: **^**)

Operator	Name
~	Bitwise NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~&	Bitwise NAND
~|	Bitwise NOR

- The and-or-inverter example previously shown:
  - The ***assign statement*** allows the use of more complex operators and operands – more on this later



Instantiating gate primitives

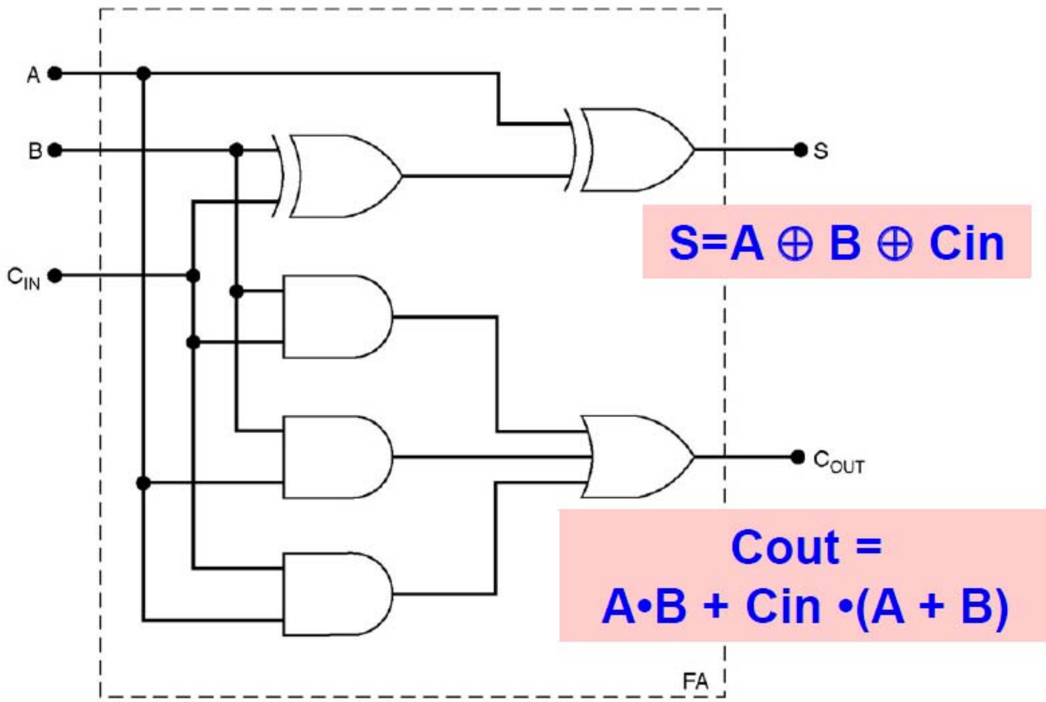
```
wire and1out, and2out;

nor g3 (out, and1out, and2out);
and g2 (and2out, c, d);
and g1 (and1out, a, b);
```

```
assign out = ~((a&b) | (c&d));
```

Continuous assignment

Continuous Assignment Example

- Full Adder (using assign statements)

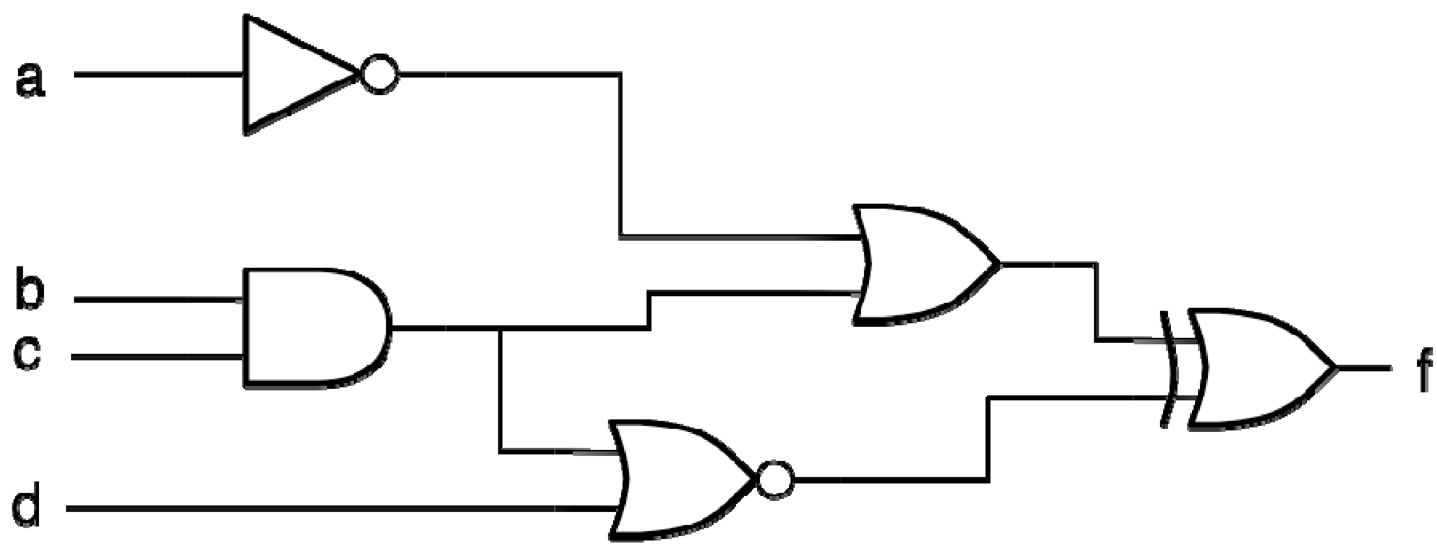


Remember that all assign statements happen concurrently (at the same time)

```
module full_adder (input A, B, Cin,
                   output S, Cout);

   assign S    = A ^ B ^ Cin;
   assign Cout = (A & B) | (B & Cin) | (A & Cin);

endmodule
```

Exercise

- You should now be able to go from circuit to structural Verilog description, or assign statement, and vice versa.
- Try this (use a single assign statement):




```
assign f = (~a | (b&c)) ^ ((b&c) ~| d);
```

Conditional Assignment

- We have already seen the continuous assignment statement:

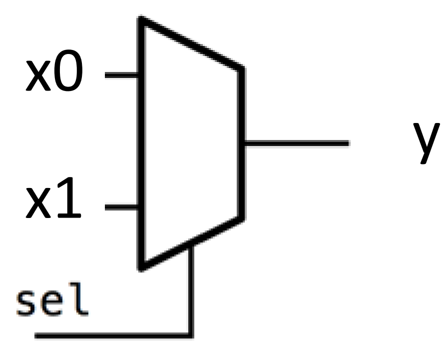
```
assign y = ~((a & b) | (c & d));
```

- It is also possible to have a conditional assignment (as in C):

```
assign y = sel ? x1 : x0; // a multiplexer
```

The signal **y** will be connected to **x1** **if sel is 1, else** it is connected to **x0**

1-bit 2x1 mux



Basic Combinational Arithmetic

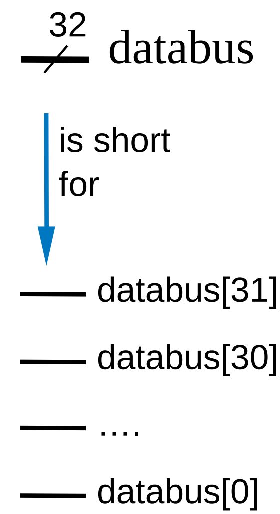
- We will not cover the details of arithmetic in Verilog in the course, but a few things are worth knowing
- Verilog supports many basic *arithmetic operations*:
  - Arithmetic: +, –, *, /   (division is not generally synthesizable)
  - Comparisons: <, <=, >, >=, ==, !=
    - The comparisons are evaluated as true (1) or false (0)
- These can be useful in statements like these:

```
assign sum = a + b;
assign diffval = curr – prev;
assign max = (a>b) ? a : b;
```

VECTORS IN VERILOG

So far we have used 1-bit signals

- What happens if I want to use multi-bit signals
  - After all it is very difficult do anything with 1 binary bit
- Verilog has a special construct for handling multi-bit signals (wires). **Formed by specifying a range:**




```
wire [31:0] databus;
```

- Also for specifying multi-bit module ports

```
module add16 (input [15:0] a, b,
              output [15:0]sum,
              output cout);

    // add module internals here

endmodule
```

Vectors in Verilog

- We can declare *multi-bit* signals (or busses as we have seen them called) in Verilog:

```
…, output [3:0] y, …
wire [7:0] something;
```

- By convention, we label the *most significant bit* (MSB) using the higher number, and the *least significant bit* (LSB) using zero
- So a *16-bit* signal will be [15:0], an *8-bit* signal will be [7:0], and a *64-bit* signal will be [63:0]

Example: Adder

- Consider the 32-bit adder below (with Carry in & out)

```
module adder (
      input Cin,
      input [31:0] A, B,
      output Cout,
      output [31:0] Sum);

      wire [32:0] w1;

      assign w1   = A + B + Cin;
      assign Sum  = w1[31:0];
      assign Cout = w1[32];

endmodule
```

- We must declare multi-bit signals

Vectors in Verilog

- *Vectors* are really easy to work with
- We can select *individual bits* of the vector:

```
assign y = some[3]; //assign 4th bit of
                    //some to y
```

- We can 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];
```

Vectors in Verilog

- But be careful! Widths of vectors in assignments *should match.*
- Verilog will let you do some really bad things:
  - Like in the previous example, **so be VERY careful!!!!**

```
assign x[2:0] = y[1];
assign x[2:1] = a;     // a 1-bit
```

- With arithmetic, this can be **catastrophic!**
- Always check you are assigning signals of equal width, and remember **the LSB index is zero**!

```
Always carefully check the Synthesis warnings!
```

Number Literals

- What about assigning a fixed bit pattern to a signal?
- 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")

Number Literals

- Underscores can help split long strings:
  - 16'b0010_1110_0110_1001 (easier than 0010111001101001)
    - In this example, it's better to use hex instead (16'h2E69)
  - 32'h0F4B_C009
- If you assign a literal to a larger signal, it is *zero-padded at the MSB*:

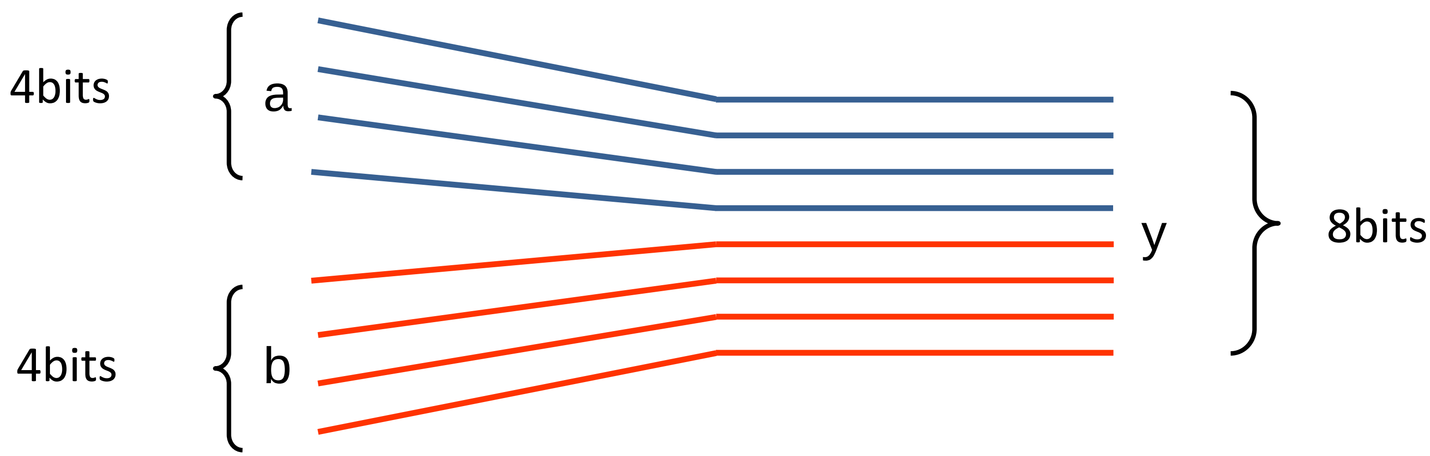
```
wire [5:0] x = 4'b1001   //x=001001
```

- If you assign a literal to a smaller signal, it is *truncated at the MSB*:

```
wire [3:0] x = 6'b110011   //x=0011
```

Concatenation

- It is sometime 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] a, b;
wire [7:0] y;
…
assign y = {a, b};
```

Concatenation and Replication

- 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 most significant bit 4 times

PARAMETERS

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)

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

- Can have multiple parameters

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

Parameters

- It is also possible to redefine a **parameter**
  - Consider the following 8-bit module

```
module submod #(parameter SIZE=8) (
    input    [SIZE-1:0] X, Y, output [SIZE-1:0] Z);
    // some statements in here
endmodule
```

- Then, note the change in the parameter in the submodule instantiation below
  - Now using two 16-bit submodules

```
module top_mod #(parameter SIZE=16) (
    input    [SIZE-1:0] a, b, c,  output [SIZE-1:0] D, E);

    submod #(.SIZE(SIZE)) U1 (.X(a), .Y(b), .Z(D));
    submod #(.SIZE(SIZE)) U2 (.X(c), .Y(b), .Z(E));

endmodule
```

In practice, it may be better to call them different names to avoid confusion

Example: Adder

- Consider the 32-bit adder below (with Carry in & out)

```
module adder #(parameter SIZE=32)(
        input Cin,
        input [SIZE-1:0] A, B,
        output Cout,
        output [SIZE-1:0] Sum);

        assign {Cout, Sum} = A+B+Cin;

endmodule
```

Note: 1. How the concatenation operator handles the 33-bit result produced by A+B+Cin
2. The use of the arithmetic addition operator (+) to generate the sum.