CME 341: Block Diagrams, Primitives, and Assignments

Brian Berscheid

Department of Electrical and Computer Engineering
University of Saskatchewan

Today's agenda

1. Interpreting Block Diagrams
2. Building Logic with Verilog: Explicit Structural Method
3. Building Logic with Verilog: Implicit Structural Method

Interpreting Block Diagrams

Module Instantiations: Review

- A module can be instantiated inside another module
- The module prototype is like a blueprint or template that is used by Verilog to create objects (instances)
- Each instantiated module needs a unique instance name
- Need to connect signals (generally wires) to the ports of the instance
  - ▷ Signals should be declared before they are connected
  - ▷ Much more on signal types later...
- The hierarchy can be nested as deeply or as shallowly as you like

Goto Chapter 1 Notes, Pages 9-13 on Block Diagrams

Declaring Signals

- Signals must be declared before they are used
- There are two main data types of interest in CME 341: reg and wire
- For now, stick with wire (reg will be discussed in detail later)
- To declare a signal, just use:

```
wire my_wire_name; // declares a 1-bit wire
reg  my_reg_name;  // declares a 1-bit reg
```

- Note that Verilog is case sensitive!

Vectors

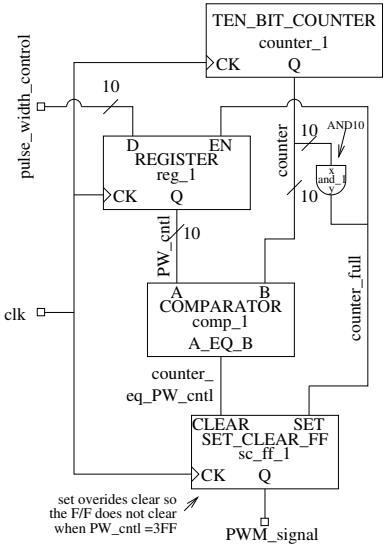
- It is convenient to declare multi-bit signals ("vectors") to represent data buses
- To do so, just specify the bit width in the signal declaration:

```
wire [3:0]  my_wire_name; // declares a 4-bit wire
reg  [7:0]  my_reg_name;  // declares an 8-bit reg
```

- Any time the name of a vector is used, it refers to the full set of bits
- Can access individual bits, ie. `my_wire_name[3] // MSB of vector`
- The notation [3:0] indicates that bit [3] of the vector is the MSB and bit [0] is the LSB
- It is possible to define vectors in the opposite orientation (with bit [0] as the MSB) as below, but this is very strongly discouraged

```
wire [0:3] my_wire_name; // DON'T DO THIS !!!
```

Challenge: Write the Verilog to Represent this PWM Diagram



Building Logic with Verilog: Explicit Structural Method

Logic Construction Approaches

- There are three main methods of building logic inside a module:
  - ▷ Explicit structural approach: manually instantiating and wiring primitives
  - ▷ Implicit structural approach: using operators to implicitly instantiate primitives
  - ▷ Behavioral approach: describing the behavior of the circuit under certain conditions
- The list above goes from lower to higher levels of abstraction (and ease of use)
- It is possible (and common) to mix these approaches within a module
- Behavioral methods will be discussed in detail in Chapter 3; for now we will focus on the structural approaches

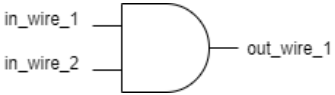
Verilog Built-In Primitives

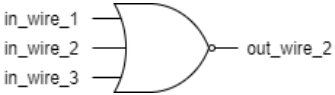
- Verilog has a set of built-in logic primitives that can be directly instantiated in your code
- The most used primitives correspond to the basic logic gates studied in EE 232:
  `and`, `nand`, `or`, `nor`, `xor`, `xnor`, `not`
- All (except for `not`) accept a variable number of 1-bit inputs and produce a 1-bit output
- `not` accepts a single 1-bit input and produces a 1-bit output
- Generally need to handle individual bits of vectors separately
- As always, instances of primitives require unique instance names
- The port names are omitted in primitive instantiations
  - ▷ Output port comes first

Primitive Instantiation Examples:

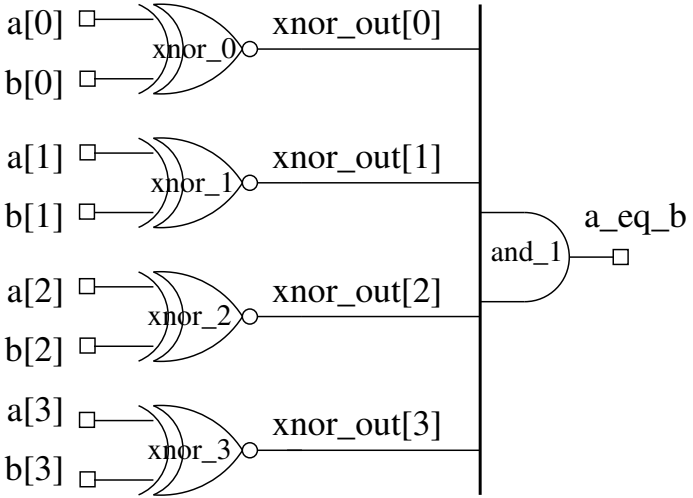
```
// declare signals
wire in_wire_1, in_wire_2, in_wire_3;
wire out_wire_1, out_wire_2, out_wire_3;

// instantiate primitives
and and_1 (out_wire_1, in_wire_1, in_wire_2);
nor my_nor (out_wire_2, in_wire_1, in_wire_2, in_wire_3);
not any_name_you_like (out_wire_3, in_wire_3);
```





Challenge: Write Explicit Structural Verilog to Implement this Comparator



Building Logic with Verilog: Implicit Structural Method

The assign statement

- Implicit structural logic is constructed using the `assign` statement as follows:

```
assign target_wire = expression;
```

- `target_wire` (LHS of statement) can be any wire or vector you have previously declared
- The expression (RHS of statement) is built by combining signals and operators
- Can cascade multiple operators to build complex expressions
  (use parentheses to make order of operations clear)
- If vectors are used, the operations and assignment are performed on a per-bit basis (sizes MUST[1] match)

[1] Technically not a MUST, but doing otherwise is DANGEROUS and DISCOURAGED. This will be discussed in more detail later in the course.

Operators used with assign

- There are two ways operators can be used:
  - ▷ Bitwise operation: requires two operands, performs bitwise operation between corresponding bits of operands
  - ▷ Reduction operation: requires one operand, performs operation between all bits of operand (single-bit output)

- Not all operands can be used in both modes

```
~      inverter / not (one operand; bitwise negation)
&      and (bitwise or reduction)
~&     nand (reduction only)
|      or (bitwise or reduction)
~|     nor (reduction only)
^      xor (bitwise or reduction)
~^     xnor (bitwise or reduction)
```

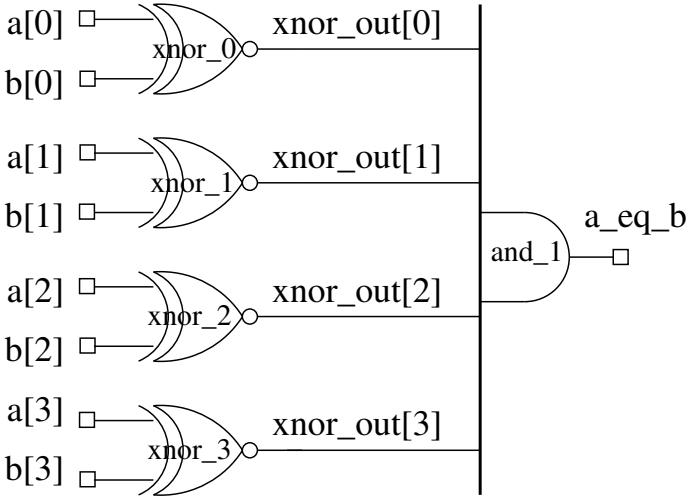
assign statement examples

```
// declare signals
wire in_wire_1, in_wire_2, in_wire_3;
wire [3:0] vec_1, vec_2, vec_3;
wire out_wire_1, out_wire_2, out_wire_3, out_wire_4;

// instantiate primitives
// examples from primitive section
assign out_wire_1 = (in_wire_1 & in_wire_2);
assign out_wire_2 = ~(in_wire_1 | in_wire_2 | in_wire_3);
assign out_wire_3 = ~in_wire_3;

// vector examples
assign vec_3 = (vec_1 & vec_2); // bitwise AND
assign out_wire_4 = & vec_1;    // reduction operator, single bit output
```

Challenge: Write Implicit Structural Verilog to Implement this Comparator



Thank you!
Have a great day!