CSE-308: Digital System Design

Lecture 3

VERILOG

Hardware Description Language

About Verilog

- Along with VHDL, Verilog is among the most widely used HDLs.
- Main differences:
  - VHDL was designed to support system-level design and specification.
  - Verilog was designed primarily for digital hardware designers developing FPGAs and ASICs.
- The differences become clear if someone analyzes the language features.

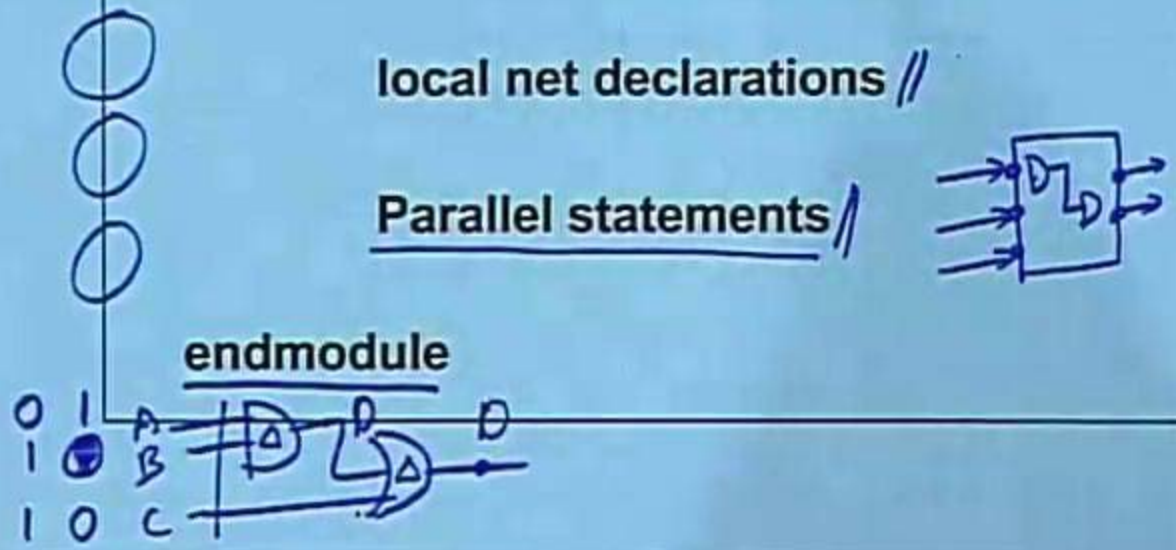
- VHDL
  - Provides some high-level constructs not available in Verilog (user defined types, configurations, etc.).
- Verilog
  - Provides comprehensive support for low-level digital design
  - Not available in native VHDL
    - Range of type definitions and supporting functions (called packages) needs to be included.

Concept of Verilog "Module"

- In Verilog, the basic unit of hardware is called a *module*.
  - Modules cannot contain definitions of other modules.
  - A module can, however, be instantiated within another module.
  - Allows the creation of a *hierarchy* in a Verilog description.

Basic Syntax of Module Definition

```
module module_name (list_of_ports);
    input/output declarations //
    local net declarations //
    Parallel statements //
endmodule
```

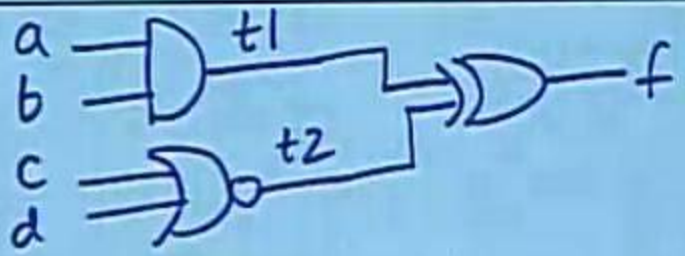


Example 1 :: simple AND gate


```
module  simpleand (f, x, y);
   input  x, y;
   output  f;
   assign  f = x & y;
endmodule
```

Example 2 :: two-level circuit


```
module two_level (a, b, c, d, f);
   input a, b, c, d;
   output f;
   wire t1, t2;
   assign t1 = a & b;
   assign t2 = ~ (c | d);
   assign f = t1 ^ t2;
endmodule
```



Example 3 :: a hierarchical design

```
module  add3 (s, cy3, cy_in, x, y);
     input [2:0] x, y;
     input cy_in;
     output [2:0] s;
     output cy3;
     wire [1:0]  cy_out;
       add B0 (cy_out[0], s[0], x[0], y[0], cy_in);
       add B1 (cy_out[1],s[1],x[1],y[1],cy_out[0]);
       add B2 (cy3, s[2], x[2], y[2], cy_out[1]);
endmodule
```

Specifying Connectivity

- There are two alternate ways of specifying connectivity:
  - ***Positional association***
    - The connections are listed in the same order

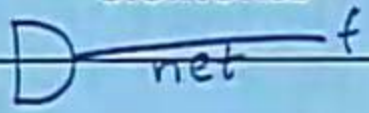
```
add  A1 (c_out, sum, a, b, c_in);
```

  - ***Explicit association***
    - May be listed in any order

```
add  A1 (.in1(a), .in2(b), .cin(c_in),
         .sum(sum), .cout(c_out));
```

Variable Data Types

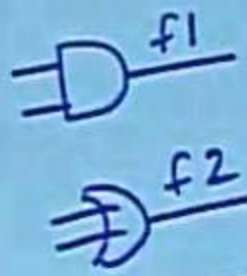
- A variable belongs to one of two data types:
  - *Net*
    - Must be continuously driven
    - Used to model connections between continuous assignments & instantiations
  - *Register*
    - Retains the last value assigned to it
    - Often used to represent storage elements

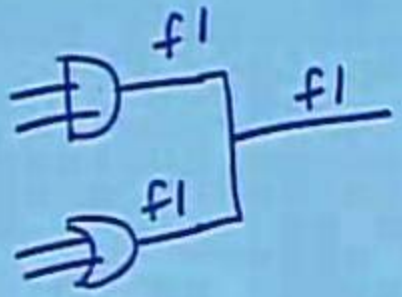


Net data type

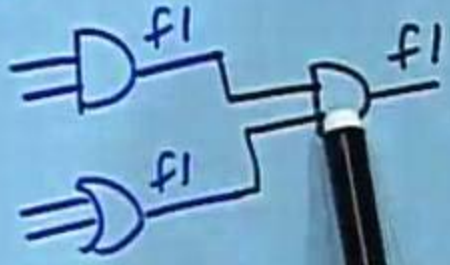
- Different 'net' types supported for synthesis:
  - *wire, wor, wand, tri, supply0, supply1*
- 'wire' and 'tri' are equivalent; when there are multiple drivers driving them, the outputs of the drivers are shorted together.
- 'wor' / 'wand' inserts an OR / AND gate at the connection.
- 'supply0' / 'supply1' model power supply connections.

wire f1, f2;



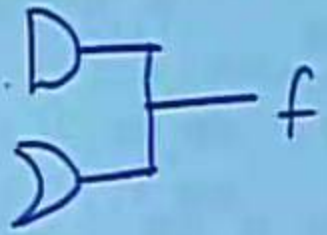


wand f1;



```
module  using_wire  (A, B, C, D, f);
   input    A, B, C, D;
   output   f;
   wire     f;          // net f declared as 'wire'

   assign   f = A & B;
   assign   f = C | D;
endmodule
```

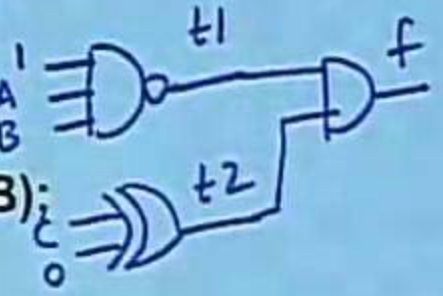


```
module using_wired_and (A, B, C, D, f);
   input  A, B, C, D;
   output f;
   wand   f;          // net f declared as 'wand'

   assign f = A & B;
   assign f = C | D;
endmodule
```

```
module using_supply_wire (A, B, C, f);
   input    A, B, C;
   output   f;
   supply0  gnd;
   supply1  vdd;
   nand   G1  (t1, vdd, A, B);
   xor    G2  (t2, C, gnd);
   and    G3  (f, t1, t2);
endmodule
```

wire t1, t2;



Register data type

- Different 'register' types supported for synthesis:
  - reg, integer
- The 'reg' declaration explicitly specifies the size.

```
reg  x, y;  // single-bit register variables
reg [15:0] bus; // 16-bit bus, bus[15] MSB
```

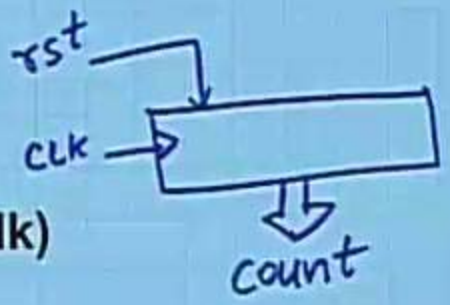
- For 'integer', it takes the default size, usually 32-bits.
  - Synthesizer tries to determine the size.

Other differences:

- In arithmetic expressions,
  - An 'integer' is treated as a 2's complement signed integer.
  - A 'reg' is treated as an unsigned quantity.
- General rule of thumb
  - 'reg' used to model actual hardware registers such as counters, accumulator, etc.
  - 'integer' used for situations like loop counting.

```
module simple_counter (clk, rst, count);
   input   clk, rst;
   output  count;
   reg [31:0]  count;

   always @(posedge clk)
   begin
      if (rst)
         count = 32'b0;
      else
         count = count + 1;
   end
endmodule
```




```
 01111111
        +1
 ---------
 10000000
```

- When 'integer' is used, the synthesis system often carries out a data flow analysis of the model to determine its actual size.

A ⇒ 10
B ⇒ 10
10+1

- Example:

```
wire [1:10]  A, B;
integer      C;
C = A + B;
```

→ The size of C can be determined to be equal to 11 (10 bits plus a carry).

Specifying Constant Values

- A value may be specified in either the 'sized' or the 'un-sized' form.
  - Syntax for 'sized' form:

    <size>'<base><number>

32'b0

- Examples:

```
8'b01110011   // 8-bit binary number
12'hA2D       // 1010 0010 1101 in binary
12'hCx5       // 1100 xxxx 0101 in binary
25            // signed number, 32 bits
1'b0          // logic 0
1'b1          // logic 1
```