
Computer Aided Digital Systems Design - EE 4743/6743

Sherif Abdelwahed

Structural Verilog

**Department of Electrical and Computer Engineering
Mississippi State University**

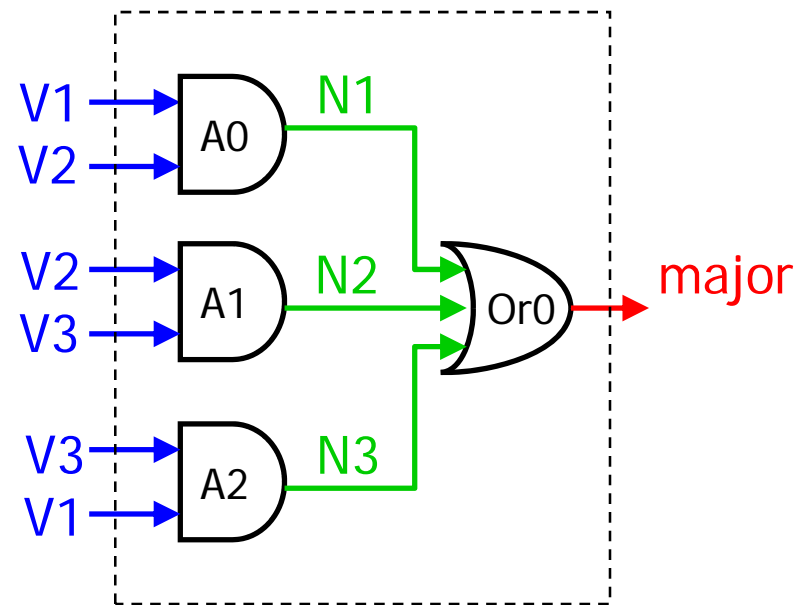
Structural Verilog

- Essentially a schematic in text form
 - Build up a circuit from gates/flip-flops
 - Gates are primitives (part of the language)
 - Flip-flops themselves described behaviorally
 - Structural design
 - Create module interface
 - Instantiate the gates in the circuit
 - Declare the internal wires needed to connect gates
 - Put the names of the wires in the correct port locations of the gates to make connections
 - For primitives, outputs always come first
-

Structural Example: Majority Detector

Structural models specify interconnections of primitives and modules. Synthesis tools may still optimize your design!

```
module majority (major, V1, V2, V3) ;  
  
    output major ;  
    input V1, V2, V3 ;  
  
    wire N1, N2, N3;  
  
    and A0 (N1, V1, V2),  
        A1 (N2, V2, V3),  
        A2 (N3, V3, V1);  
  
    or Or0 (major, N1, N2, N3);  
  
endmodule
```



Verilog Structure vs. Behavior

- **Structure**

- Gate level — built-in models for AND, OR, ...
- modules and instantiations and connecting wires

- **Behavior**

- C-like programs or Boolean algebra (with a few extra operators)
- assign statements
- always blocks — procedural statements

- If a module has an assign statement in it, is it behavior or structure?

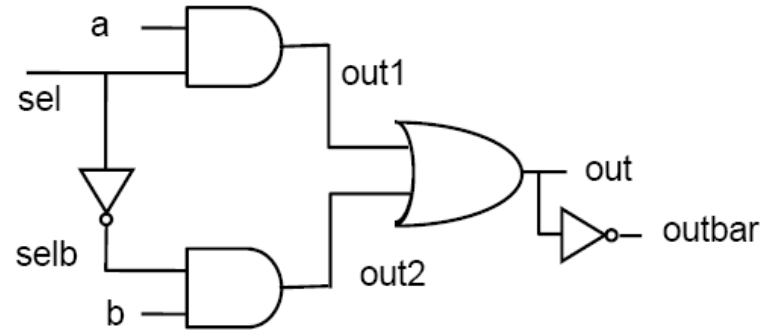
- On the outside, it appears as structure — it's wired in, takes up space (it's physical) — maybe it is an ALU slice
 - On the inside, it appears as behavior — we only know the translation of inputs to outputs, but without physical connotations
-

Structural Basics: Primitives

```
module muxgate (a, b, out, outbar, sel);
  input a, b, sel;
  output out, outbar;

  and a1 (out1, a, sel);
  not i1 (selb, sel);
  and a2 (out2, b, selb);
  or o1 (out, out1, out2);

  assign outbar = ~out;
endmodule
```



- Verilog supports basic logic gates as primitives
 - **and, nand, or, nor, xor, xnor**, supporting multiple inputs
`nand nand3in(out, in1, in2, in3);`
 - **buf, not**: one or more outputs first followed by one input
`not N1(out1, out2, out3, out4, in);`
 - **bufif0, bufif1, notif0, notif1**: three-state drivers with output terminal first, then input, then control
`Bufif1 BF1 (outA, inA, ctrlA);`

Primitives

- No declarations - can only be instantiated
- Output port appears before input ports
- Optionally specify: instance name and/or delay

```
and N25 (Z, A, B, C);    // name specified
and #10 (Z, A, B, X),
        (X, C, D, E);   // delay specified, 2 gates
and #10 N30 (Z, A, B); // name and delay specified
```

Delays are not synthesizable. They are used for simulation only

Syntax For Structural Verilog

- First declare the interface to the module
 - Module keyword, module name
 - Port names/types/sizes
 - Next, declare any internal wires using “wire”
 - `wire [3:0] partialsum;`
 - Then instantiate the primitives/submodules
 - Indicate which signal is on which port
-

How to Wire Modules Together

- Real designs have many modules and gates

```
module putTogether ();  
    wire w1, w2, w3, w4;  
  
    bbb lucy(w1, w2, w3, w4);  
    aaa ricky(w3, w2, w1);  
    ...
```

what happens when
out1 is set to 1?

```
module bbb (i1, i2, o1, clk);  
    input i1, i2, clk;  
    output o1;  
  
    xor (o1, i2, ...);  
    ...
```

```
module aaa (in1, out1, out2);  
    input in1;  
    output out1, out2;  
  
    ...  
    nand #2 (out1, in1, b);  
    nand #6 (out2, in1, b);  
    ...
```

Each module has its own namespace. Wires connect elements of namespaces.

Implicit Wires

- How come there were no wires declared in some of these modules?
 - Gate instantiations implicitly declare wires for their outputs.
 - **All** other connections must be explicitly declared as wires — for instance, connections between module ports
 - **Output** and **input** declarations are wires

```
module putTogether ();  
  wire w1, w2, w3, w4;  
  
  mux inst1(w1, w2, w3, w4);  
  aaa duh(w3, w2, w1);  
  ...  
endmodule
```

wires explicitly declared

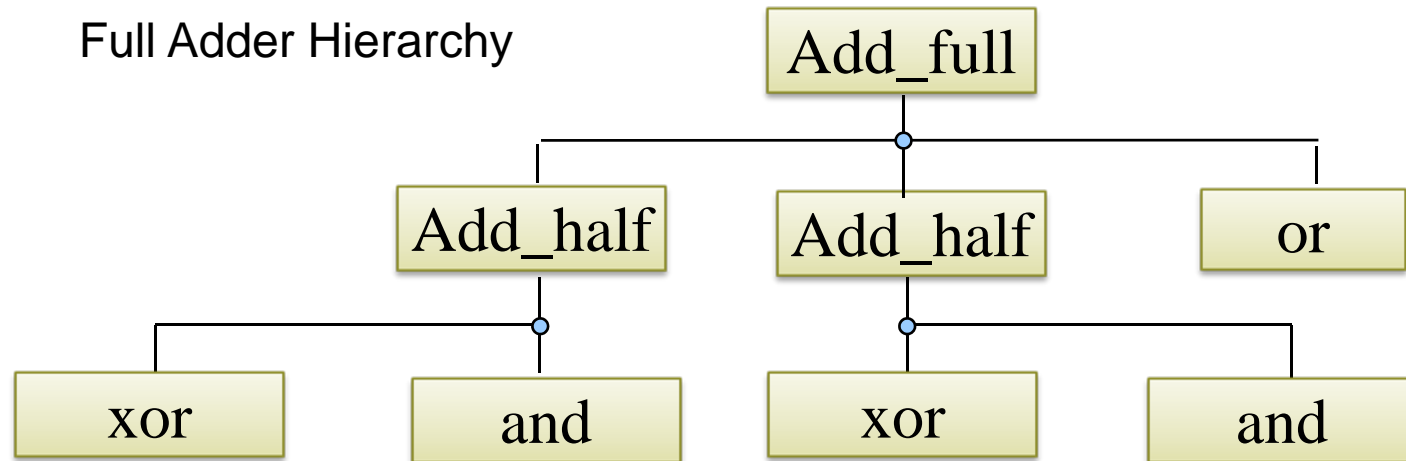
```
module mux (f, a, b, sel);  
  output f;  
  input a, b, sel;  
  
  and #5 g1 (f1, a, nsel),  
         g2 (f2, b, sel);  
  or #5 g3 (f, f1, f2);  
  not g4 (nsel, sel);  
  
endmodule
```

wires implicitly declared (f1, f2, nsel)

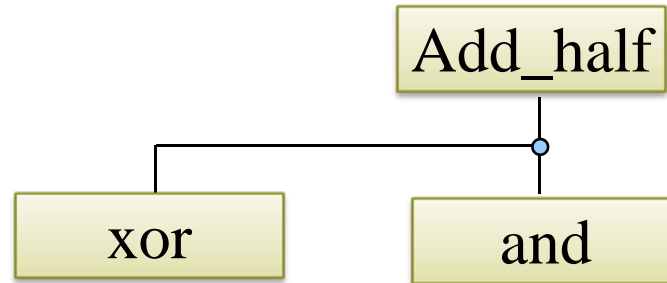
Hierarchical Design

- Build up a module from smaller pieces
 - Primitives
 - Other modules (which may contain other modules)
 - Building blocks for hierarchical design requires:
 - Interface (“black box” representation)
 - Module name, ports
 - Definition
 - Describe functionality of the block. Includes interface
 - Instantiation
 - Use the module inside another module
 - Instantiation & hierarchy control complexity.
 - No one designs 1M+ random gates — they use hierarchy.
 - What are the software analogies?
-

Example: Full Adder

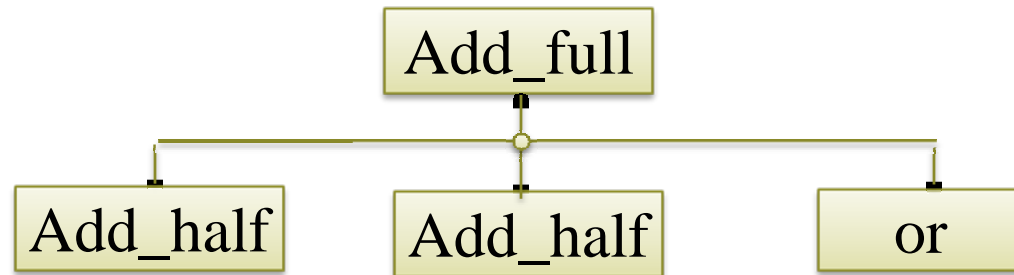


Add_half Module



```
module Add_half(c_out, sum, a, b);  
  output sum, c_out;  
  input a, b;  
  
  xor sum_bit(sum, a, b);  
  and carry_bit(c_out, a, b);  
endmodule
```

Add_full Module



```
module Add_full(c_out, sum, a, b, c_in) ;  
    output sum, c_out;  
    input a, b, c_in;  
  
    wire w1, w2, w3;  
  
    Add_half AH1(.sum(w1), .c_out(w2), .a(a), .b(b));  
    Add_half AH2(.sum(sum), .c_out(w3), .a(c_in), .b(w1));  
  
    or carry_bit(c_out, w2, w3);  
endmodule
```

Structural Verilog: Connections

- **“Positional”** or **“Implicit”** port connections
 - Used for primitives (first port is output, others inputs)
 - Can be okay in some situations
 - Designs with very few ports
 - Interchangeable input ports (and/or/xor gate inputs)
 - Gets confusing for large #s of ports
- Can specify the connecting ports by **name**
 - Helps avoid “misconnections”
 - Don’t have to remember port order
 - Can be easier to read
 - Syntax

```
.<port name>(<signal name>)
```

Connection Examples

- Variables – defined in upper level module
 - `wire [3:2] X; wire W_n; wire [3:0] word;`
 - By position
 - `module dec_2_4_en (A, E_n, D);`
 - `dec_2_4_en DX (X[3:2], W_n, word);`
 - By name
 - `module dec_2_4_en (A, E_n, D);`
 - `dec_2_4_en DX (.E_n(W_n), .A(X[3:2]), .D(word));`
 - In both cases,
`A = X[3:2], E_n = W_n, D = word`
-

Empty Port Connections

- Example: `module dec_2_4_en(A, E_n, D);`
 - `dec_2_4_en DX (X[3:2], , word); //E_n is high imp (z)`
 - `dec_2_4_en DX (X[3:2], W_n,); //Outputs D[3:0] unused.`
 - General rules
 - Empty input ports => high impedance state (z)
 - Empty output ports => output not used
 - Specify all input ports anyway!
 - Usually don't want z as input
 - Clearer to understand and find problems
 - Helps if no connection to name port, but leave empty:
 - `dec_2_4_en DX(.A(X3:2), .E_n(W_n), .D());`
-

Hierarchy and Scope

- Parent cannot access “internal” signals of child
- If you need a signal, must make a port!

Example:

Detecting overflow

Overflow =

cout XOR cout6

Must output

overflow or cout6!

```
module add8bit(cout, sum, a, b);
    output [7:0] sum;
    output cout;
    input [7:0] a, b;
    wire cout0, cout1,... cout6;
    FA A0(cout0, sum[0], a[0], b[0], 1'b0);
    FA A1(cout1, sum[1], a[1], b[1], cout0);
    ...
    FA A7(cout, sum[7], a[7], b[7], cout6);
endmodule
```

Can Mix Styles In Hierarchy

```
module Add_half_bhv(c_out, sum, a, b);
  output reg sum, c_out;
  input a, b;

  always @(a, b) begin
    sum = a ^ b;
    c_out = a & b;
  end
endmodule
```

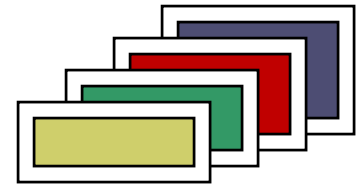
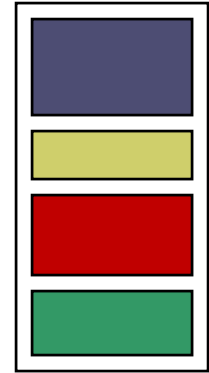
```
module Add_full_mix(c_out, sum, a, b, c_in) ;
  output sum, c_out;
  input a, b, c_in;
  wire w1, w2, w3;

  Add_half_bhv AH1(.sum(w1), .c_out(w2), .a(a), .b(b));
  Add_half_bhv AH2(.sum(sum), .c_out(w3), .a(c_in), .b(w1));
  assign c_out = w2 | w3;
endmodule
```

Hierarchy and Source Code

- All modules in a single file
 - Module order doesn't matter!
 - Good for small designs
 - Not so good for bigger ones
 - Not so good for module reuse (cut & paste)

- Break up modules into multiple files
 - Helps with organization
 - Lets you find a specific module easily
 - Great for module reuse (add file to project)



Design Readability

- Choose hierarchical blocks that have:
 - Minimal routing between blocks
 - Logical data flow between blocks
 - Choose descriptive labels for blocks and signals
 - Keep clock domains separated
 - Makes the interaction between clocks very clear
 - Each file should be less than 400 lines of HDL code
 - Easier to read, synthesize, and debug
-

Reusability/Extensibility of Modules

```
module xor_array(y_out, a, b);  
  parameter SIZE = 8, DELAY = 15; // parameter defaults  
  output [SIZE-1:0] y_out;  
  input [SIZE-1:0] a,b;  
  wire #DELAY y_out = a ^ b;  
endmodule
```

```
xor_array G1 (y1, a1, b1); // use defaults  
xor_array #(4, 5) G2(y2, a2, b2); // override defaults  
// SIZE = 4, DELAY = 5
```

- Module instantiations cannot specify delays without **parameters**

Overriding Parameters

- Parameters can be overridden
 - Generally done to “resize” module
 - Implicitly: override in order of appearance
 - `xor_array #(4, 5) G2(y2, a2, b2);`
 - Explicitly: name association
 - `xor_array #(.SIZE(4), .DELAY(5)) G3(y2, a2, b2);`
 - Explicitly: **defparam**
 - `defparam G4.SIZE = 4, G4.DELAY = 15;`
 - `xor_array G4(y2, a2, b2);`
 - **localparam** parameters in a module can't be overridden
 - `localparam SIZE = 8, DELAY = 15;`
-

Generated Instantiation

- **Generate statements:** control over the instantiation (creation) of
 - Modules, gate primitives, continuous assignments, **initial** blocks, **always** blocks
 - Generate instantiations resolved during “elaboration” (compile time)
 - When module instantiations are linked to module definitions
 - **Before** the design is simulated or synthesized
 - *Think of it as having the code help write itself*
-

Generated Instantiation

- Index variables used in generate statements declared using **genvar** (e.g., **genvar i**)
 - Often useful when developing parameterized modules
 - Can override the parameters to create different-sized structures
 - Easier than creating different structures for all different possible bitwidths
-

Generate-Loop

- A generate-loop permits making one or more instantiations (pre-synthesis) using a for-loop.

```
module gray2bin1 (bin, gray);
  parameter SIZE = 8; // this module is parameterizable
  output [SIZE-1:0] bin; input [SIZE-1:0] gray;
  genvar i;
  generate
    for (i=0; i<SIZE; i=i+1) begin: bit
      assign bin[i] = ^gray[SIZE-1:i]; // reduction or
    end
  endgenerate
endmodule
```

Generate-Conditional

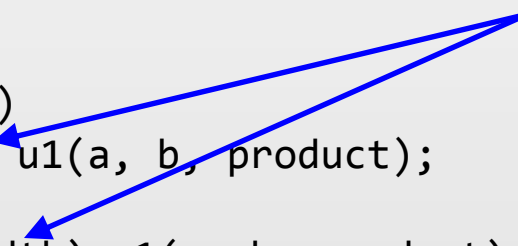
- A generate-conditional allows conditional (pre-synthesis) instantiation using **if-else-if** constructs

```
module multiplier(a ,b ,product);
  parameter a_width = 8, b_width = 8;
  localparam product_width = a_width+b_width;

  input [a_width-1:0] a;
  input [b_width-1:0] b;
  output [product_width-1:0] product;

  generate
    if ((a_width < 8) || (b_width < 8))
      CLA_multiplier #(a_width,b_width) u1(a, b, product);
    else
      WALLACE_multiplier #(a_width,b_width) u1(a, b, product);
  endgenerate
endmodule
```

These are parameters, not variables!



Generate-Case

- A generate-case allows conditional (pre-synthesis) instantiation using case constructs

```
module adder (output co, sum, input a, b, ci);
  parameter WIDTH = 8;

  generate
    case (WIDTH)
      1: adder_1bit x1(co, sum, a, b, ci); // 1-bit adder
      2: adder_2bit x1(co, sum, a, b, ci); // 2-bit adder
      default: adder_cla #(WIDTH) x1(co, sum, a, b, ci);
    endcase
  endgenerate

endmodule
```

Generate a Pipeline [Part 1]

```
module pipeline(out, in, clk, rst);
  parameter BITS = 8;
  parameter STAGES = 4;

  input [BITS-1:0] in;
  output [BITS-1:0] out;

  wire [BITS-1:0] stagein [0:STAGES-1]; // value from previous stage
  reg [BITS-1:0] stage [0:STAGES-1]; // pipeline registers

  assign stagein[0] = in;

  generate
    genvar s;
    for (s = 1; s < STAGES; s = s + 1) begin : stageinput
      assign stagein[s] = stage[s-1];
    end
  endgenerate

  // continued on next slide
```

Generate a Pipeline [Part 2]

```
// continued from previous slide

assign out = stage[STAGES-1];

generate
  genvar j;
  for (j = 0; j < STAGES; j = j + 1) begin : pipe
    always @(posedge clk) begin
      if (rst) stage[j] <= 0;
      else stage[j] <= stagein[j];
    end
  end
endgenerate
Endmodule
```

``define` Compiler Directives

- Macro for text substitutions - Used within and outside of module declarations

```
`define text_macro_name[(arguments)] macro-text
```

- Example 1:

```
`define address_register_length 16  
reg [`address_register_length :1] address;
```

- Example 2:

```
`define nord(dly) nor #dly  
`nord(4) g1 (N3, A, B); // becomes nor #4 g1(N3, A, B);  
`undef nord
```

- How does this differ from parameter?
 - Careful of conflicting directives
 - ``defines` in force until you indicate otherwise!
-

Conditional Compilation Directives

- Conditional compilation directives include:
 - ``ifdef`, ``ifndef`, ``else`, ``elsif`, ``endif`
- Useful when
 - Selecting different stimulus
 - Choosing different timing or structural information
 - Selecting different representations of modules

```
module and_op (output a, input b, c);  
  `ifdef RTL  
    assign a = b & c;           // continuous assign  
  `else  
    and a1 (a, b, c);          // primitive  
  `endif  
endmodule
```

`Include Compiler Directives

- ``include filename`
- Inserts entire contents of another file at compilation
- Can be placed anywhere in Verilog source
- Useful for shared tasks/functions!
- Example 1:

```
module use_adder8(...);  
    `include "adder8.v"; // include the task for adder8
```

- Example 2:

```
module use_incrementer(...);  
    `include "/home/schulte/components/incrementer.v"
```

- Can provide either relative or absolute path names
 - Useful for including tasks and functions in multiple modules
-