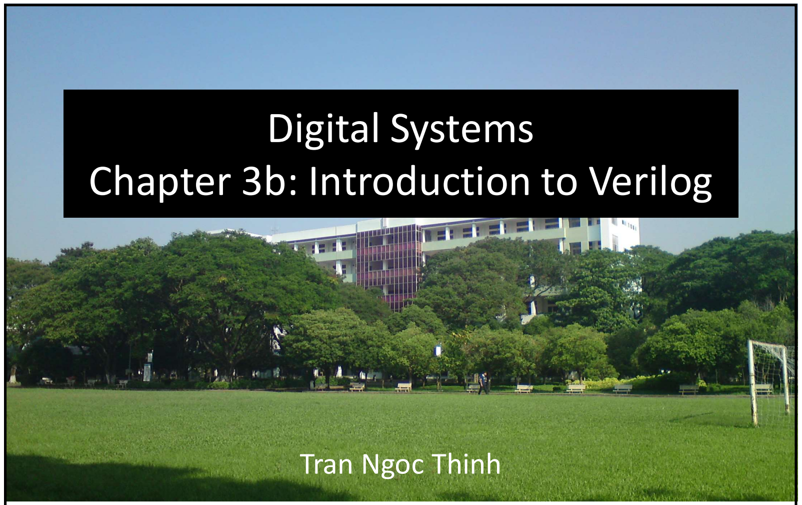


Digital Systems
Chapter 3b: Introduction to Verilog

Tran Ngoc Thinh

Adapted from Dr. Pham Quoc Cuong's slides (cuongpham@hcmut.edu.vn and Prof. Mike Schulte's slides (schulte@engr.wisc.edu)

Computer Engineering – CSE – HCMUT

Overview of HDLs

- Hardware description languages (HDLs)
  - Are computer-based hardware **description** languages
  - Allow modeling and simulating the functional behavior and timing of digital hardware
  - Synthesis tools take an HDL description and generate a technology-specific netlist
- Two main HDLs used by industry
  - Verilog HDL (C-based, industry-driven) (60% of US companies use it)
  - VHSIC HDL or VHDL (Ada-based, defense/industry/university-driven).

Synthesis of HDLs

- Takes a description of what a circuit DOES
- Creates the hardware to DO it

- HDLs may LOOK like software, **but they're not**!
  - NOT a program
  - Doesn't "run" on anything
    - Though we do *simulate* them on computers
  - Don't confuse them!

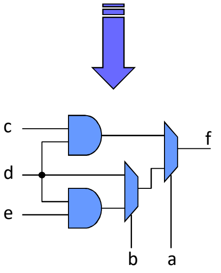
- Also use HDLs to test the hardware you create
  - This is more like software

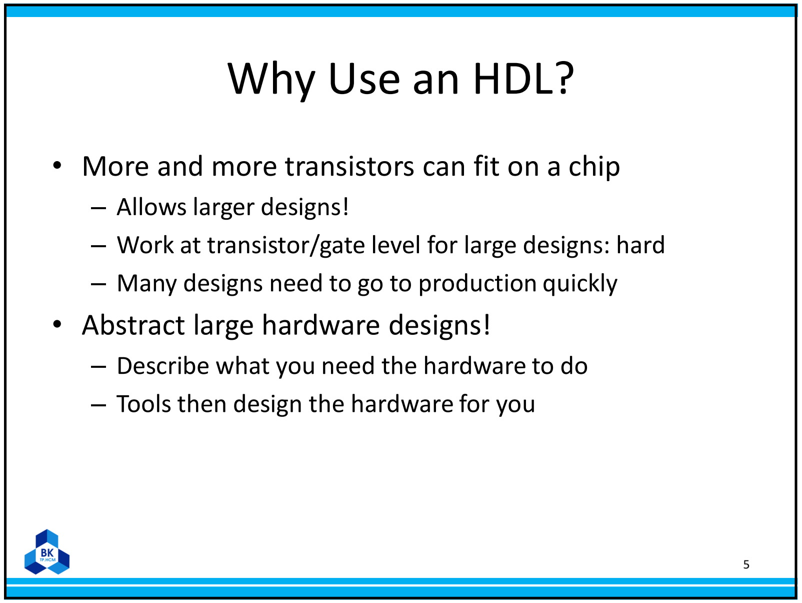
Describing Hardware!

- All hardware created during synthesis
  - Even if `a` is true, still computing `d&e`

- Learn to understand how descriptions translated to hardware

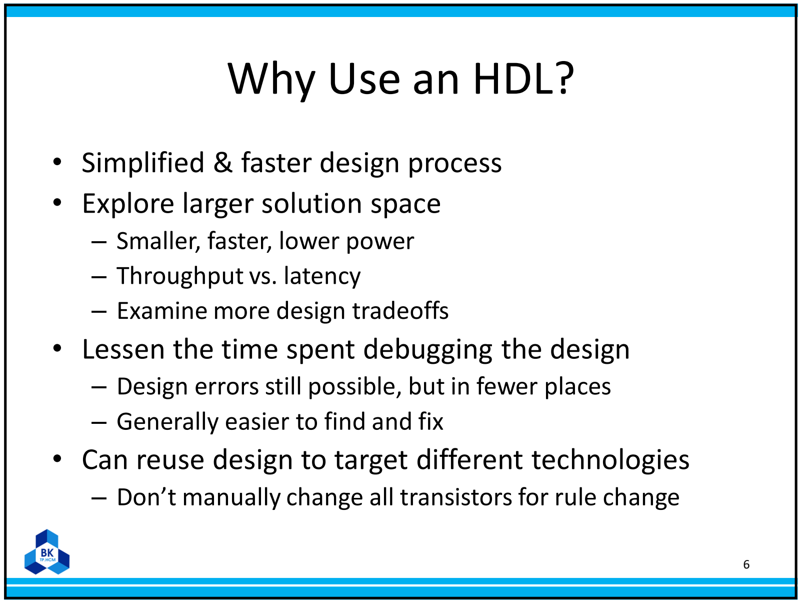
```
if (a) f = c & d;
else if (b) f = d;
else f = d & e;
```





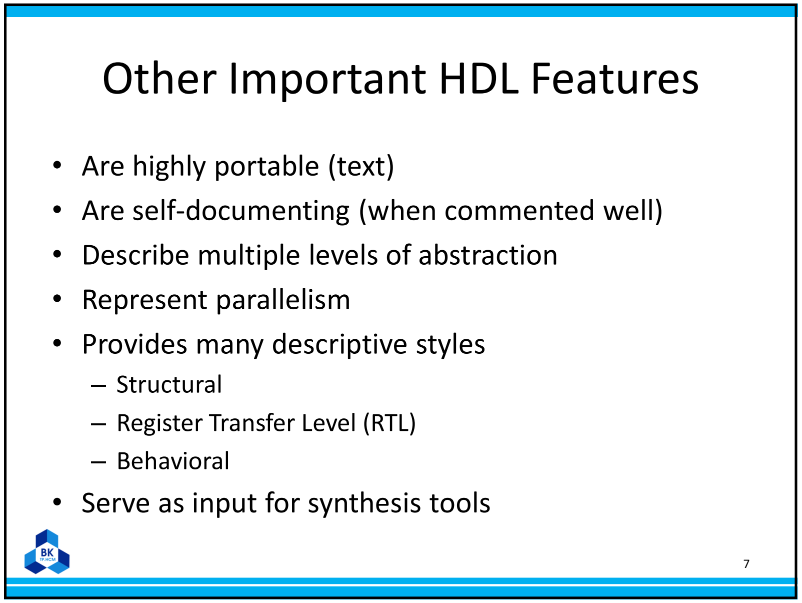
Why Use an HDL?

- More and more transistors can fit on a chip
  - Allows larger designs!
  - Work at transistor/gate level for large designs: hard
  - Many designs need to go to production quickly
- Abstract large hardware designs!
  - Describe what you need the hardware to do
  - Tools then design the hardware for you



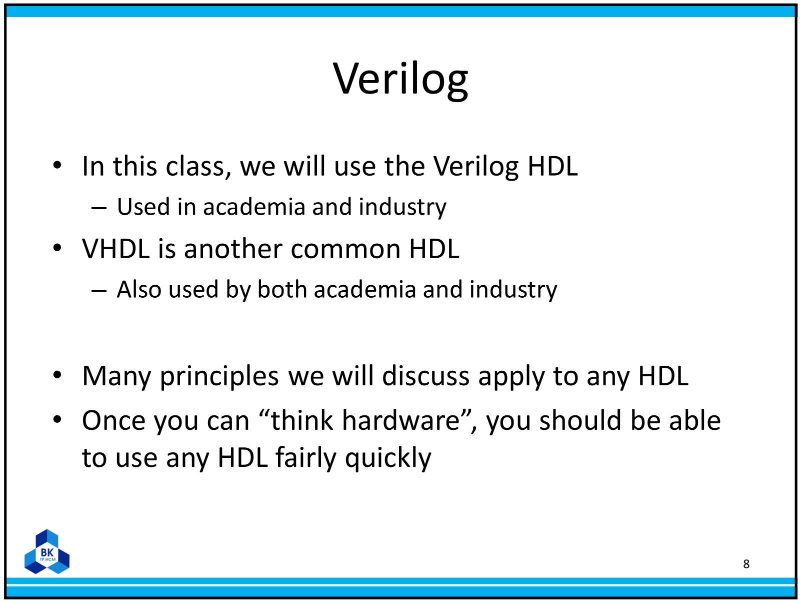
Why Use an HDL?

- Simplified & faster design process
- Explore larger solution space
  - Smaller, faster, lower power
  - Throughput vs. latency
  - Examine more design tradeoffs
- Lessen the time spent debugging the design
  - Design errors still possible, but in fewer places
  - Generally easier to find and fix
- Can reuse design to target different technologies
  - Don’t manually change all transistors for rule change



Other Important HDL Features

- Are highly portable (text)
- Are self-documenting (when commented well)
- Describe multiple levels of abstraction
- Represent parallelism
- Provides many descriptive styles
  - Structural
  - Register Transfer Level (RTL)
  - Behavioral
- Serve as input for synthesis tools



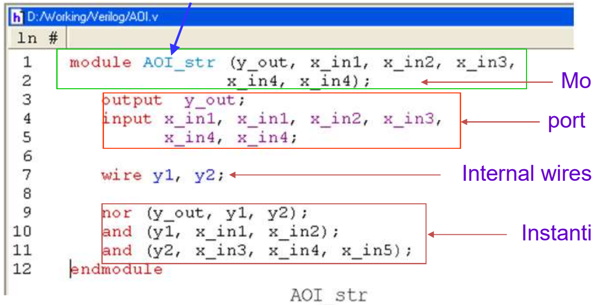
Verilog

- In this class, we will use the Verilog HDL
  - Used in academia and industry
- VHDL is another common HDL
  - Also used by both academia and industry

- Many principles we will discuss apply to any HDL
- Once you can “think hardware”, you should be able to use any HDL fairly quickly

Verilog Modules

module name



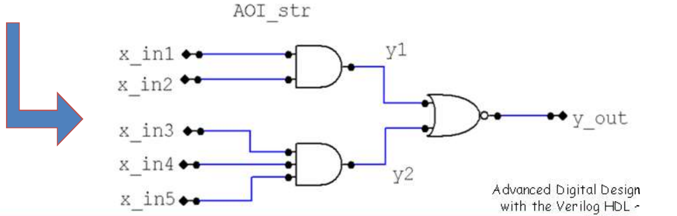
```
module AOI_str (y_out, x_in1, x_in2, x_in3,
                x_in4, x_in4);
    output  y_out;
    input x_in1, x_in1, x_in2, x_in3,
          x_in4, x_in4;

    wire y1, y2;

    nor (y_out, y1, y2);
    and (y1, x_in1, x_in2);
    and (y2, x_in3, x_in4, x_in5);
endmodule
```

Module ports

port modes

Internal wires

Instantiated primitives



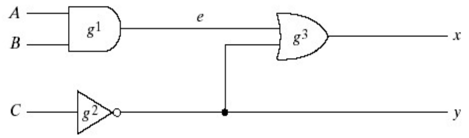
Advanced Digital Design with the Verilog HDL - chapter 4

Verilog Module

- In Verilog, a circuit is a **module**.
  - A *module* is the building block in Verilog.
  - It is declared by the keyword *module* and is always terminated by the keyword *endmodule*.
  - Each statement is terminated with a semicolon, but there is no semi-colon after *endmodule*.

```
module module_name ( port_list );
    port declarations;
    …
    variable declaration;
    …
    description of behavior
endmodule
```

Verilog Module



module name

ports names of module

port types

```
module smpl_circuit(A,B,C,x,y);
    input  A,B,C;
    output x,y;
    wire    e;
        and    g1(e,A,B);
        not    g2(y,C);
        or     g3(x,e,y);
endmodule
```

module contents

keywords underlined

Verilog Module (2)

module name

ports names of module

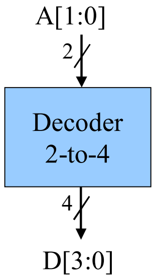
port types

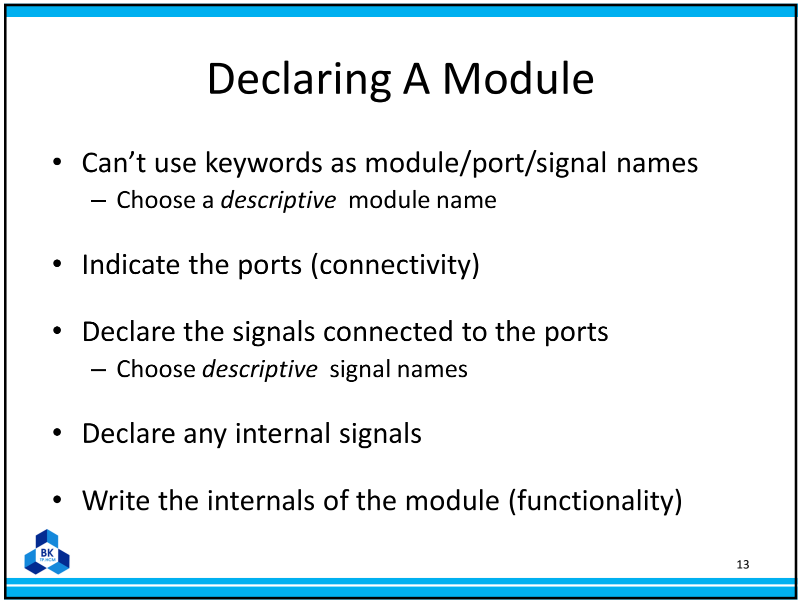
port sizes

```
module decoder_2_to_4 (A, D) ;
input [1:0] A ;
output [3:0] D ;

assign D =    (A == 2'b00) ? 4'b0001 :
              (A == 2'b01) ? 4'b0010 :
              (A == 2'b10) ? 4'b0100 :
              (A == 2'b11) ? 4'b1000 ;
endmodule
```

module contents

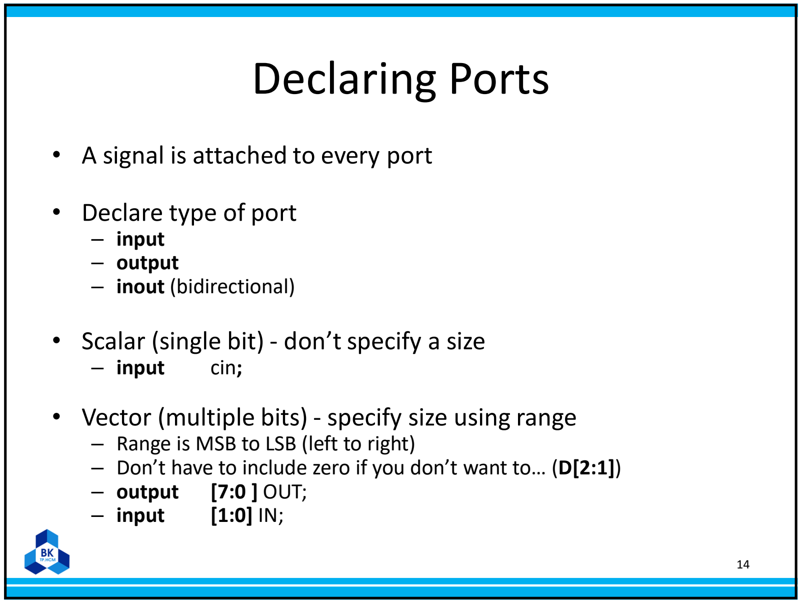
keywords underlined





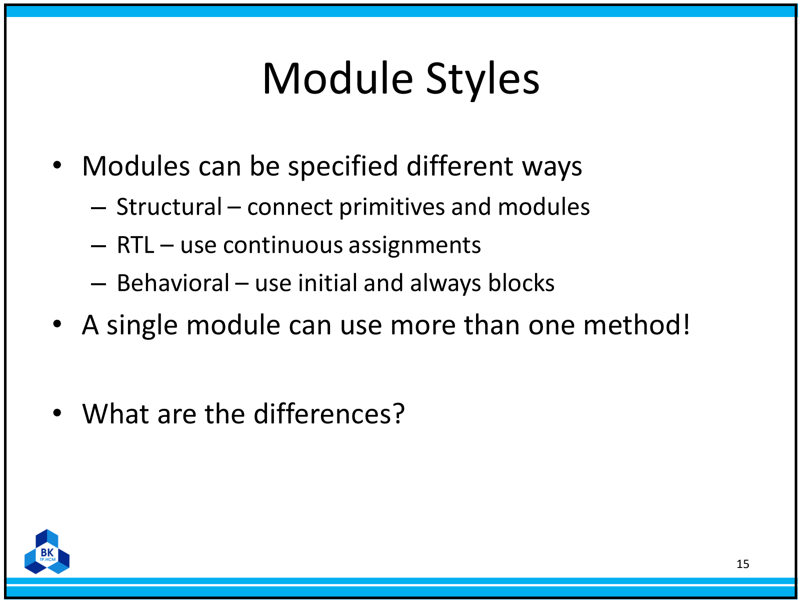
Declaring A Module

- Can’t use keywords as module/port/signal names
  - Choose a *descriptive* module name
- Indicate the ports (connectivity)
- Declare the signals connected to the ports
  - Choose *descriptive* signal names
- Declare any internal signals
- Write the internals of the module (functionality)



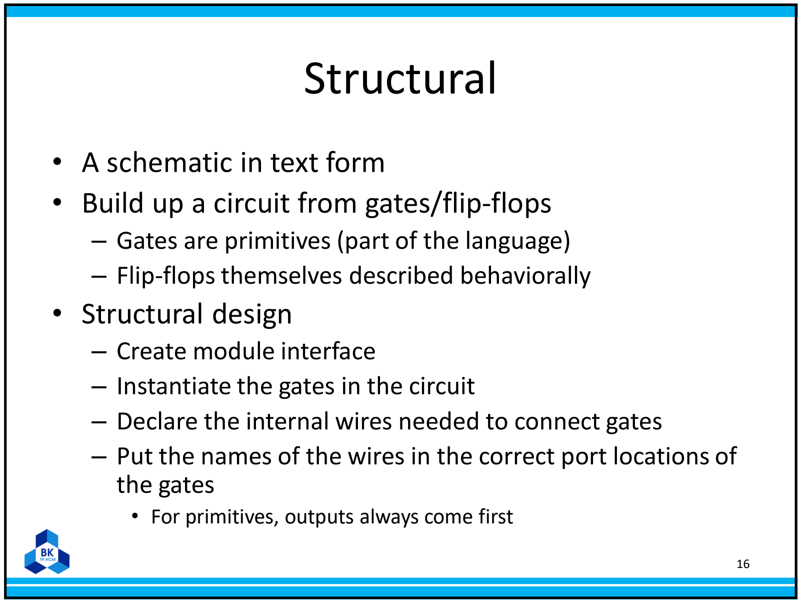
Declaring Ports

- A signal is attached to every port
- Declare type of port
  - **input**
  - **output**
  - **inout** (bidirectional)
- Scalar (single bit) - don’t specify a size
  - **input** cin**;**
- Vector (multiple bits) - specify size using range
  - Range is MSB to LSB (left to right)
  - Don’t have to include zero if you don’t want to… (**D[2:1]**)
  - **output** **[7:0]** OUT;
  - **input** **[1:0]** IN;



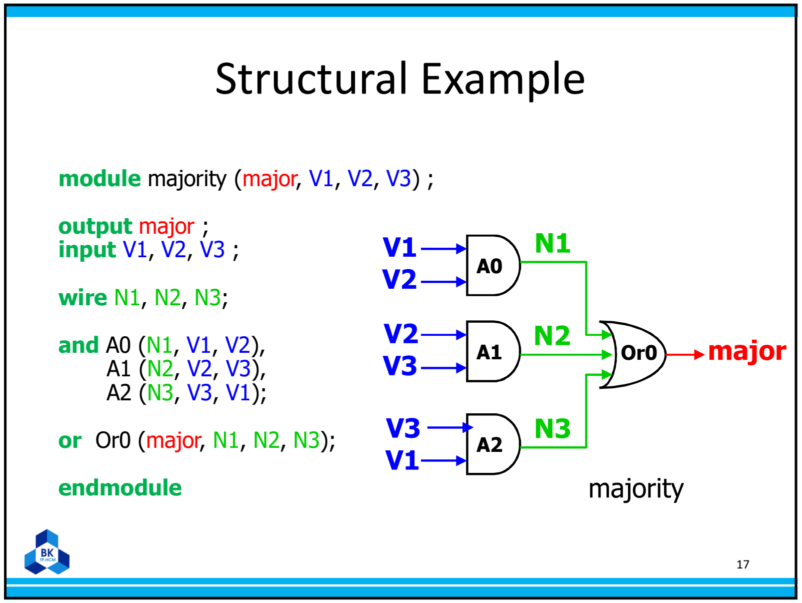
Module Styles

- Modules can be specified different ways
  - Structural – connect primitives and modules
  - RTL – use continuous assignments
  - Behavioral – use initial and always blocks
- A single module can use more than one method!
- What are the differences?



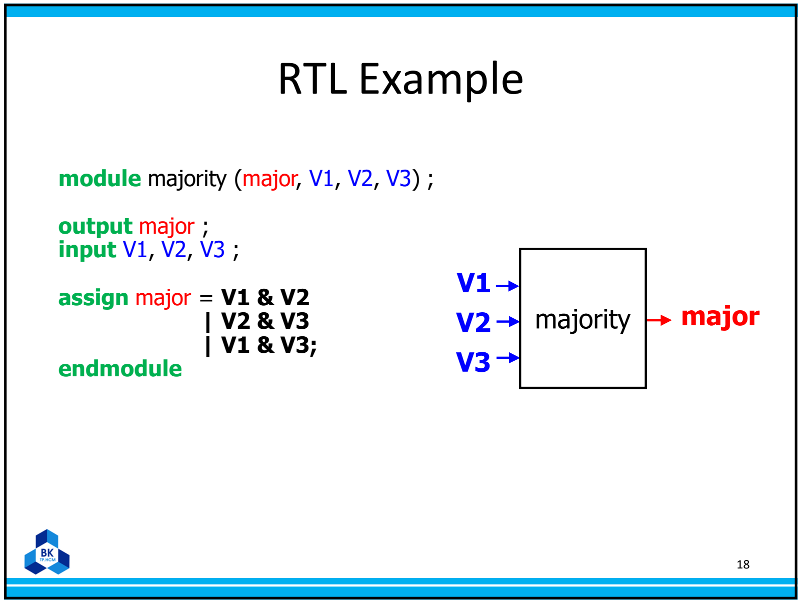
Structural

- 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
    - For primitives, outputs always come first



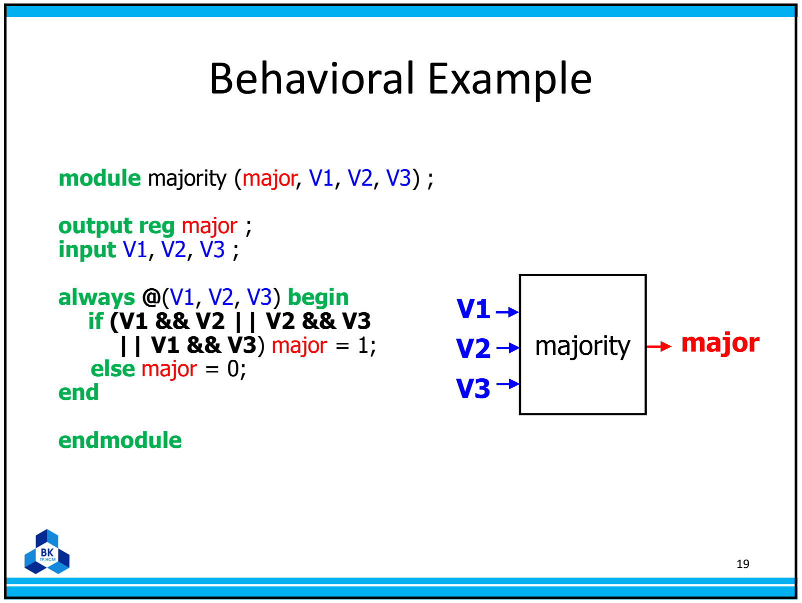
Structural Example

```
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
```



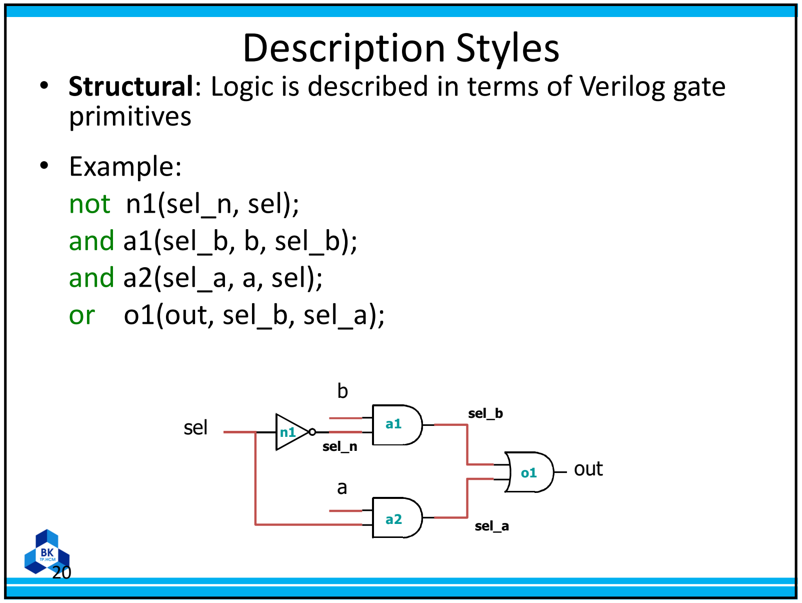
RTL Example

```
module majority (major, V1, V2, V3) ;

output major ;
input V1, V2, V3 ;

assign major = V1 & V2
             | V2 & V3
             | V1 & V3;
endmodule
```



Behavioral Example

```
module majority (major, V1, V2, V3) ;

output reg major ;
input V1, V2, V3 ;

always @(V1, V2, V3) begin
    if (V1 && V2 || V2 && V3
        || V1 && V3) major = 1;
    else major = 0;
end

endmodule
```



Description Styles

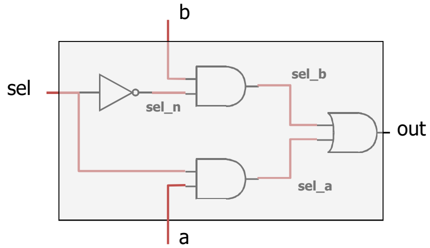
- **Structural**: Logic is described in terms of Verilog gate primitives
- Example:

```
not  n1(sel_n, sel);
and a1(sel_b, b, sel_b);
and a2(sel_a, a, sel);
or   o1(out, sel_b, sel_a);
```

Description Styles (cont.)

- **Dataflow (RTL)**: Specify output signals in terms of input signals
- Example:

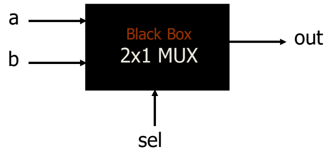
```
assign out = (sel & a) | (~sel & b);
```



Description Styles (cont.)

- **Behavioral**: Algorithmically specify the behavior of the design
- Example:

```
if (select == 0) begin
      out = b;
end
else if (select == 1) begin
      out = a;
end
```



Structural Basics: Primitives

- Build design up from the gate/flip-flop/latch level
  - Flip-flops actually constructed using Behavioral
- Verilog provides a set of gate primitives
  - and, nand, or, nor, xor, xnor, not, buf, bufif1, etc.
  - Combinational building blocks for structural design
  - Known “behavior”
  - Cannot access “inside” description
- Can also model at the transistor level
  - Most people don’t, we won’t

Primitives

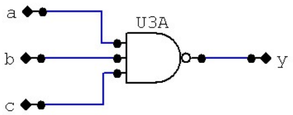
- No declarations - can only be instantiated
- Output port appears before input ports
- Optionally specify: instance name and/or delay (discuss delay later)

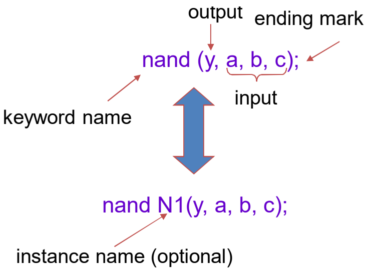
```
and N25 (Z, A, B, C);     // name specified
and #10 (Z, A, B, X),
        (X, C, D, E);     // delay specified, 2 gates
and #10 N30 (Z, A, B); // name and delay specified
```

Verilog Primitives

- 26 pre-defined primitives
- Output is the first port

n-input	n-output 3-states
and	buf
nand	not
or	bufif0
nor	bufif1
xor	notif0
xnor	notif1



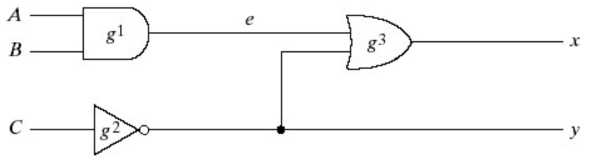

output
ending mark
nand (y, a, b, c);
keyword name
input
nand N1(y, a, b, c);
instance name (optional)

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

Verilog - Module

- A *module* is the building block in Verilog.
- It is declared by the keyword *module* and is always terminated by the keyword *endmodule*.
- Each statement is terminated with a semicolon, but there is no semi-colon after *endmodule*.



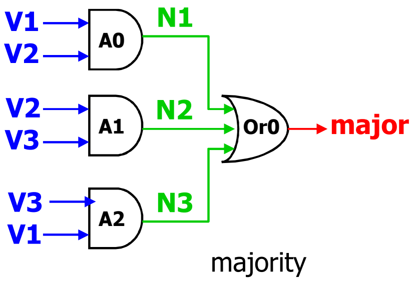
Verilog – Module (2)

HDL Example

```
module smpl_circuit(A,B,C,x,y);
  input  A,B,C;
  output x,y;
  wire    e;
  and  g1(e,A,B);
  not  g2(y,C);
  or   g3(x,e,y);
endmodule
```

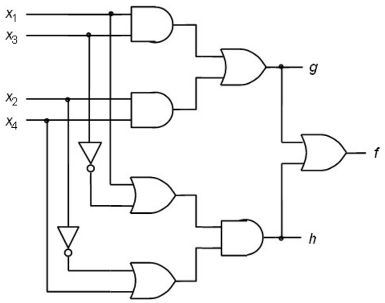
Again: Structural Example

```
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
```



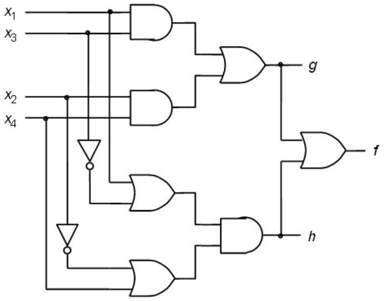
Example

- Using the Verilog structural style, describe the following circuit



Example (1)

```
module example2 (x1, x2, x3, x4, f, g, h);
        input x1, x2, x3, x4;
        output f, g, h;

        and (z1, x1, x3);
        and (z2, x2, x4);
        or (g, z1, z2);
        or (z3, x1, ~x3);
        or (z4, ~x2, x4);
        and (h, z3, z4);
        or (f, g, h);

endmodule
```



Example: Combinational Gray code

$$S_2' = \overline{Rst}\ S_2\ S_0 + \overline{Rst}\ S_1\ \overline{S_0}$$

$$S_1' = \overline{Rst}\ \overline{S_2}\ S_0 + \overline{Rst}\ S_1\ \overline{S_0}$$

$$S_0' = \overline{Rst}\ \overline{S_2}\ \overline{S_1} + \overline{Rst}\ S_2\ S_1$$

Datatypes

- Two categories
  - Nets
  - “Registers”
- Only dealing with nets in structural Verilog
- “Register” datatype doesn’t actually imply an actual register…
  - Will discuss this when we discuss Behavioral Verilog

Net Types

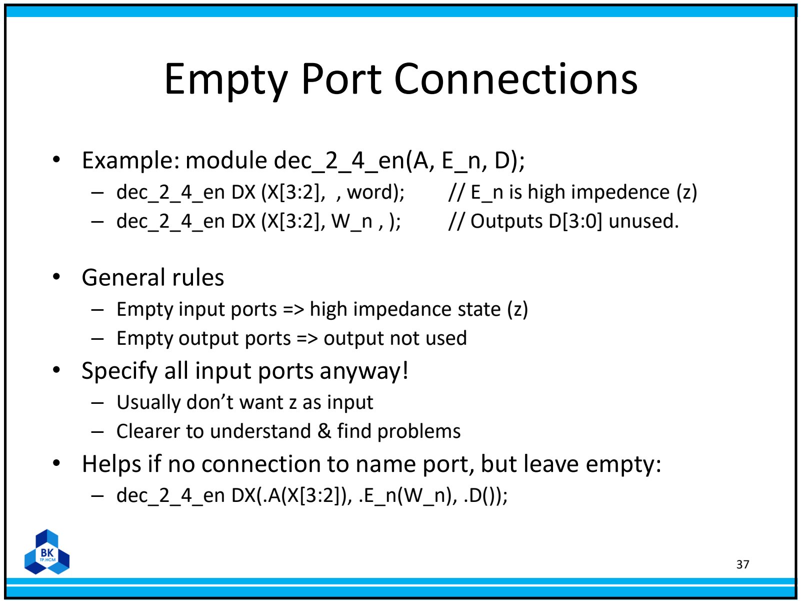
- **Wire**: most common, establishes connections
  - Default value for all signals
- **Tri**: indicates will be output of a tri-state
  - Basically same as “wire”
- **supply0**, **supply1**: ground & power connections
  - Can imply this by saying “0” or “1” instead
  - xor xorgate(out, a, 1’b1);
- **wand**, **wor**, **triand**, **trior**, **tri0**, **tri1**, **trireg**
  - Perform some signal resolution or logical operation
  - Not used in this course

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

Connections 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 impedence (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 & find problems
- Helps if no connection to name port, but leave empty:
  - dec_2_4_en DX(.A(X[3:2]), .E_n(W_n), .D());