

Verilog Tutorial

By

Sachin B. Patkar

(adapted from some slides from
internet)

About the original source

- The web page from which I downloaded original slides itself does not indicate/claim the authorship of those slides
- You can check the original slides at the following URL (Caution: but you will see a lot of difference!)
- <http://cs.haifa.ac.il/courses/verilog/verilog.html>

VERILOG HDL

- Basic Unit – A module
 - Module describes a hardware component
 - Modules will be instantiated in the description of a hardware design
 - Module *similar to* “C++ class” or “a blue-print”,
 - module instances == hardware objects
- Module
 - Describes the functionality of the design
 - States the input and output ports
 - a “port” is an “interface” to rest of the universe

Module

- General definition

```
module module_name (  
    port_list );  
    port declarations;  
    ...  
    variable declaration;  
    ...  
    description of behavior  
    using a set of “concurrent  
    statements”  
endmodule
```

- Example

```
module HalfAdder (A, B, Sum  
    Carry);  
    input A, B; output Sum, Carry;  
    wire A,B, Sum, Carry;  
  
    // concurrent statements follow  
    assign Sum = A ^ B;  
    // ^ denotes XOR  
    assign Carry = A & B;  
    // & denotes AND  
endmodule
```

Concurrent Statements

- Hardware specified using one or more “concurrent statements”
- Each “concurrent statement” could be in “structural style” or in “dataflow style” or a “behavioural” statement
- Each concurrent statement describes a piece of hardware/logic with inputs and outputs
- These hardware/logic blocks work “forever” **and concurrently of each other (concurrent == in parallel)**

Lexical Conventions

(just a glimpse of usual gory details! Ignorable in the beginning)

- **Comments**

- // Single line comment

- /* Another single line comment */

- /* Begins multi-line (block) comment

- All text within is ignored line below ends multi-line comment

- */

- **Number** : decimal, hex, octal, binary, unsized decimal form
size base form

- include underlines, +,-

- **String**

- " Enclose between quotes on a single line" (not hardware concept)

Lexical Conventions (some more pain !)

- **Identifier**

A ... Z

a ... z

0 ... 9

Underscore

- Strings are limited to 1024 chars
- First char of identifier must not be a digit
- **Keywords:** See B & V text.
- **Operators:** See B & V text.

Important: Verilog is case sensitive

Description Styles

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

- Example:

```
not n1(sel_n, sel); // create instance of a “not” gate
```

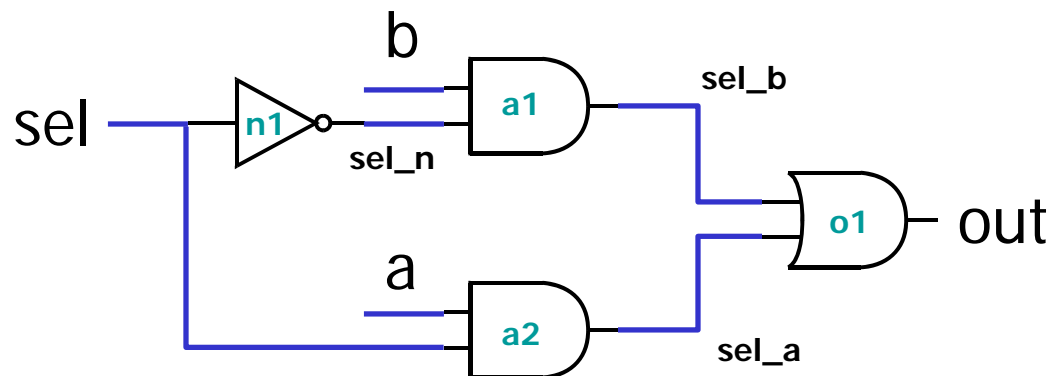
```
and a1(sel_b, b, sel_n); //create instance of “and” gate
```

```
and a2(sel_a, a, sel); // .....
```

```
or o1(out, sel_b, sel_a); // ////
```

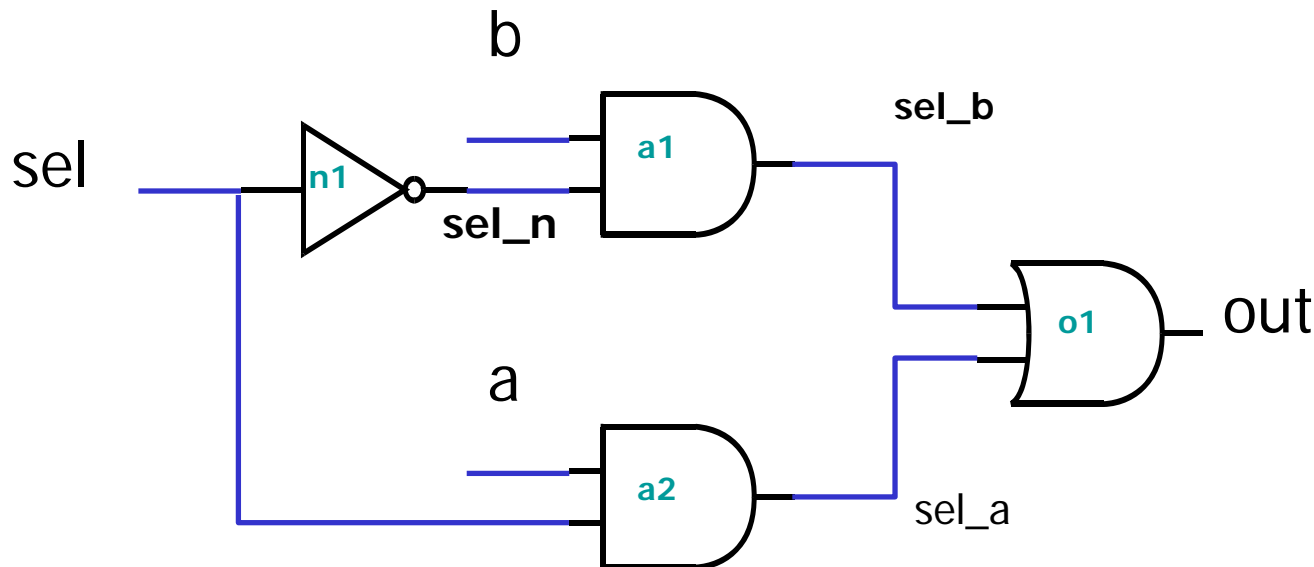
These hardware objects are wired together using internal wires
sel_n, sel_b, sel_a

Do you recognize this
logic circuit?



- **Dataflow style concurrent statement:** Specify output signals as a combination of input signals
 - A “continuous assignment”
 - specifies a combinational logic block

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



Description Styles (cont.)

- **Behavioral concurrent statement:** Algorithmically specify the behavior of the design (uses “always” block)
 - **Why “always”?** : Hardware is “always” (forever) running!!!

```
always @(a or b or select) begin
```

```
    if (select == 0) begin
```

```
        out <= b;
```

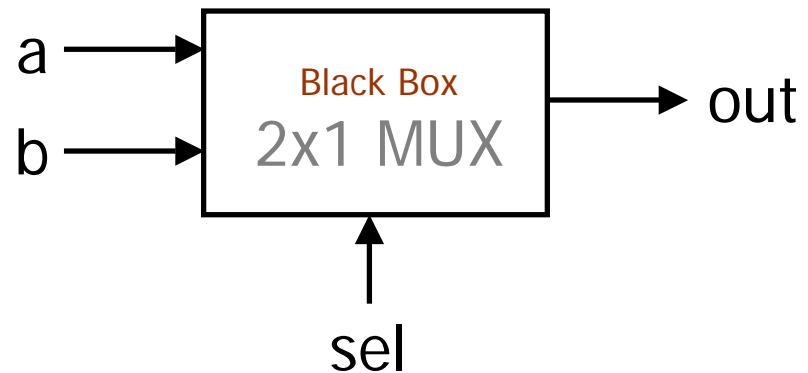
```
    end
```

```
    else if (select == 1) begin
```

```
        out <= a;
```

```
    end
```

```
end
```



Note that “out” is specified to be updated whenever any change occurs on “a”, “b” or “select” (what if we had missed out the “select == 1” condition?)

Structural Modeling

- Execution: Concurrent
- Format (Primitive Gates):
`and G2(Carry, A, B);`
- First parameter (Carry) – Output
- Other Inputs (A, B) – Inputs
- Could also specify instance of user-defined modules/gates, for example,
`HalfAdder ha0(in0, in1, sum, carry);`

Dataflow Modeling

- Uses **continuous assignment** statement
 - Format: **assign** net = expression;
 - Example: **assign** sum = a ^ b;
 - note “=” symbol (different from “<=” which we use inside “always” statements)
- each “continuous assignment” statement specifies a combinational logic block that executes concurrently with other logic blocks specified by other “concurrent” statements
- Order of the statement does not impact the design

Dataflow Modeling (cont.)

- Example:

```
module HalfAdder (A, B, Sum, Carry);  
    input A, B;  output Sum, Carry;  
    wire A, B, Sum, Carry; // not necessary to declare  
  
    // two concurrent (dataflow style) statements to model  
    // two blocks of hardware (for Sum and for Carry)  
    assign Sum = A ^ B; // one logic block to create Sum  
    assign Carry = A & B; // another logic block for Carry  
endmodule
```

Behavioral Modeling

```
module mux_2x1(a, b, sel, out);
```

```
  input a, a, sel; output out;
```

```
  reg out; // “out” must be declared as a reg! What is the rule?
```

```
  always @(a or b or sel)
```

```
  begin
```

```
    if (sel == 1)
```

```
      out <= a;
```

```
    else out <= b;
```

```
  end
```

```
endmodule
```

Sensitivity List

Behavioral Modeling (cont.)

- **always** statement : is a “**concurrent**” statement!
- When is it (at the run-time of the hardware) *specified to be executed*?
 - **Occurrence of an event in the sensitivity list**
 - **Event :: Change in the logical value**

In the last example, we are **specifying** a logic block, that “**does something**” whenever any of its inputs, ie. a, b or select change value.

Furthermore, it specifies that **if** at any such event “select” happens to be “1” then “out” is updated to take the value from “a” **otherwise** “out” takes value from (driven by) “b”

Synthesis (verilog compiler!) tool recognizes that a MUX is to be generated!

Initial and always

- Two behavioural constructs
 - **initial** Statement
 - **always** Statement
- **initial** Statement : Executes only once (NOT HARDWARE!)
- **always** Statement : hardware **executes forever** whenever specified events occur

- Example:

```
initial begin
    Sum <= 0;
    Carry <= 0;
end
...
```

```
...
always @(A or B) begin
    Sum <= A ^ B;
    Carry <= A & B;
end
...
```


Event Control

Edge Triggered Event Control

always @ (posedge CLK) //Positive Edge of CLK

Curr_State <= Next_state;

*Note that Curr_State need not be updated even if Next_State changes (thus Curr_State is **NOT** **combinationally driven** by Next_State and CLK, Curr_State needs to be **backed by memory element**)*

- Level Triggered Event Control

always @ (A or B) //change in levels of A or B (either edge!)

Out <= A & B;

no memory needed here! (purely combinational!)

Conditional (IF) Statements (used within “always” block)

if (condition)

.....

else if (condition)

.....

else

.....

- always @(D or enable)

if (enable) Q <= D

This is a D latch (HOW ?)

Conditional (case) Statements (used within “always” block only)

- Example 1:

case (X)

2'b00: Y <= A + B;

2'b01: Y <= A - B;

2'b10: Y <= A / B;

endcase

- Example 2:

case (3'b101 << 2)

3'b100: A <= B + C;

4'b0100: A <= B - C;

5'b10100: A <= B / C; //This statement is executed

endcase

Conditional Statements (cont.)

- Variants of **case** Statements:
 - **casex** and **casez**
- **casez** – z is considered as a don't care
- **casex** – both x and z are considered as don't cares
- Example:
casez (X)
 2'b1z: A <= B + C;
 2'b11: A <= B / C;
endcase

Data Types

- Net Types: Physical Connection between structural elements
 - A **net** needs to be driven “at all times” (in this sense it is like a physical **wire** only)
- Register Type: Represents an abstract storage element
 - A **reg** need not be updated “at all times”.
 - That is a value assigned to a “reg” sticks to it even when the driver has got “detached” (modeling behaviour of latches and flipflops)

Default Values

- Net Types : z (ie. unconnected)
- Register Type : z (ie. unconnected)

Data Types

- Net Type: Wire

`wire [msb : lsb] wire1, wire2, ...`

– Example

`wire Reset; // A 1-bit wire`

`wire [6:0] Clear; // A 7-bit wire(-bus)`

- Register Type: Reg

`reg [msb : lsb] reg1, reg2, ...`

– Example

`reg [3: 0] A; // 4-bit register`

`reg B; // 1-bit register`

Restrictions on Data Types

- Data Flow and Structural Modeling
 - Can use only *wire* data type
 - Cannot use *reg* data type
- Behavioral Modeling
 - Can use only *reg* data type (within initial and always constructs)
 - Cannot use *wire* data type
 - Sometimes “dataflow” style is also loosely regarded as “behavioural”

Memories

- An array of registers

```
reg [ msb : lsb ] memory1 [ upper : lower ];
```

- Example

```
reg [ 0 : 3 ] mem [ 0 : 63 ];
```

```
// An array of 64 4-bit registers
```

```
reg mem [ 0 : 4 ];
```

```
// An array of 5 1-bit registers
```


System Tasks (simulation only)

- Display tasks
 - \$display : Displays the entire list of specified signals at the time when statement is encountered
 - \$monitor : Whenever there is a change in any argument, displays the entire list at end of time step
- Simulation Control Task
 - \$finish : makes the simulator to exit
 - \$stop : suspends the simulation
- Time
 - \$time: gives the simulation

Type of Port Connections

- Connection by Position

```
module child_mod (sig_a, sig_b,  
sig_c, sig_d);  
  input    sig_a, sig_b;  
  output   sig_c, sig_d;
```

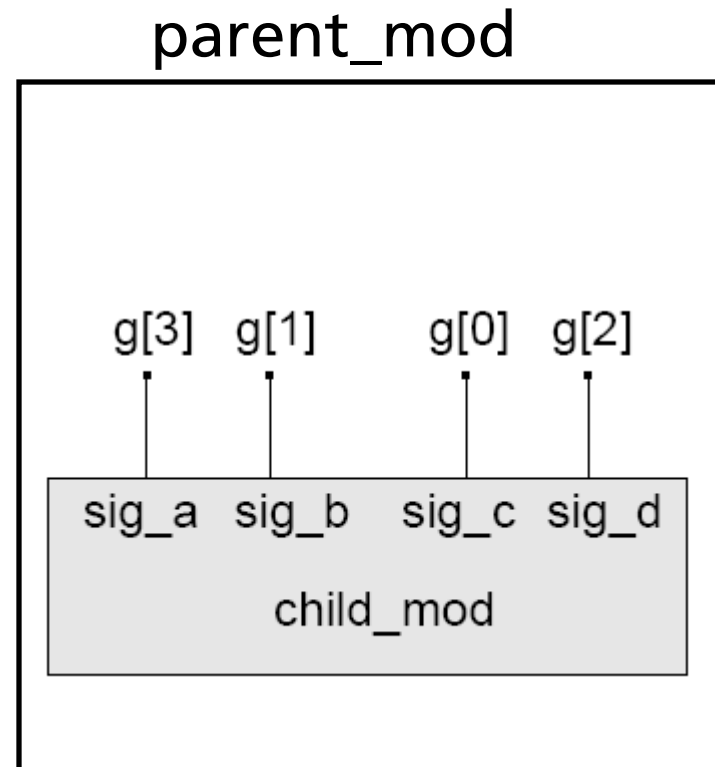
```
  // module description goes here.
```

```
endmodule
```

```
module parent_mod;  
  wire [4:0] g;
```

```
  child_mod G1 (g[3], g[1], g[0], g[2]);
```

```
  // Listed order is significant  
endmodule
```



Type of Port Connections (cont.)

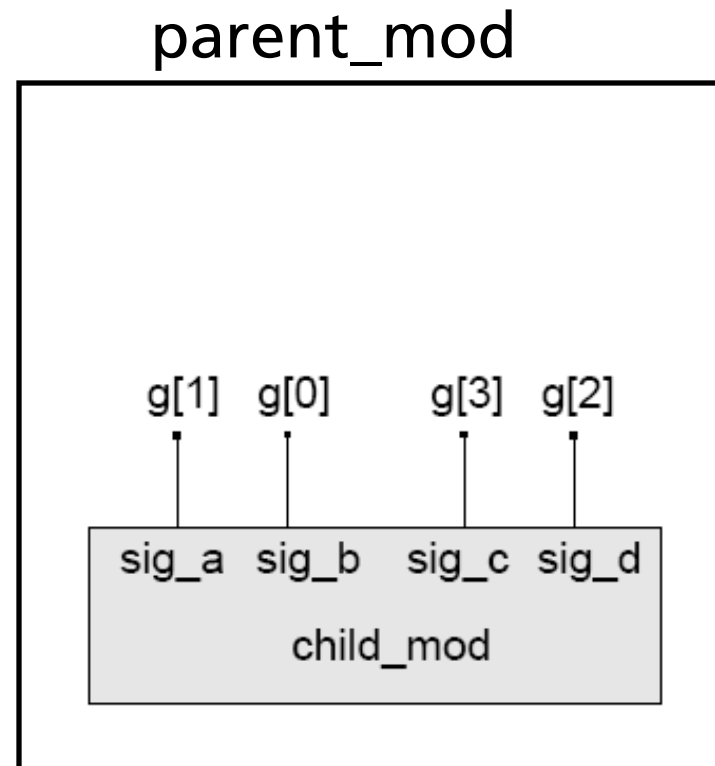
- Connection by Name

```
module child_mod (sig_a, sig_b,  
sig_c, sig_d);  
  input    sig_a, sig_b;  
  output   sig_c, sig_d;
```

```
  // module description goes here.
```

```
endmodule
```

```
module parent_mod;  
  wire [3:0] g;  
  // Listed order not significant  
  
  child_mod G1 ( .sig_c(g[3]),  
                 .sig_d(g[2]),  
                 .sig_b(g[0]),  
                 .sig_a(g[1]));  
  
endmodule
```

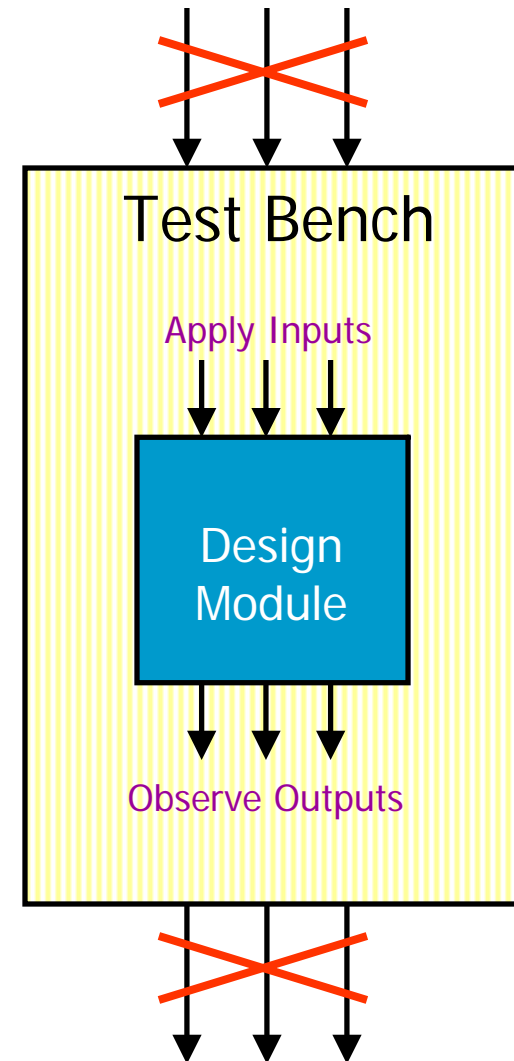


Test Bench

```
module Top; // a testbench for a halfadder
  reg PA, PB;
  wire PSum, PCarry;

  HalfAdder G1(PA, PB, PSum, PCarry);

  initial begin
    reg [1:0] i;
    for (i=0; i <=3; i=i+1) begin
      {PA, PB} <= i;
      #5 $display ("PA=%b PB=%b PSum=%b
PCarry=%b", PA, PB, PSum, PCarry);
    end // for
  end // initial
endmodule
```



Test Bench - Generating Stimulus

- Example: A sequence of values

initial begin

Clock <= 0;

#50 Clock <= 1;

#30 Clock <= 0;

#20 Clock <= 1;

end



Test Bench - Generating Clock

- Repetitive Signals (clock)



- A Simple Solution:

`wire` Clock;

`assign #10 Clock <= ~ Clock; // single inverter based oscillator`

- Caution:
 - Initial value of Clock (*wire* data type) = z
 - $\sim z = x$ and $\sim x = x$
 - **SO YOU MUST INITIALIZE**

Test Bench - Generating Clock (cont.)

- Initialize the Clock signal

```
initial begin  
    Clock <= 0;  
end
```

- Caution: Clock is of data type *wire*, cannot be used in an *initial* statement
- Solution:

```
reg Clock;  
  
...  
initial begin  
    Clock <= 0;  
end  
  
...  
always begin  
    #10 Clock <= ~ Clock;  
end
```

forever loop can
also be used to
generate clock

VERILOG: Synthesis - Combinational Logic

- Combination logic function can be expressed as:

$$\text{logic_output}(t) = f(\text{logic_inputs}(t))$$



- Rules
 - Avoid technology dependent modeling
 - The combinational logic must not have feedback.
 - Specify the output of a combinational behavior for all possible cases of its inputs.
 - Logic that is not combinational will be synthesized as sequential.

Styles for Synthesizable Combinational Logic

- Synthesizable combinational can have following styles
 - Netlist of gate instances and Verilog primitive gates (Fully structural)
 - Continuous Assignments
 - Behavioral statements
 - There are a few some more styles

Synthesis of Combinational Logic – Gate Netlist (cont.)

- General Steps:
 - Logic gates are translated to Boolean equations (**synthesis**), which are optimized optionally.
 - Optimized Boolean equations are covered by library gates (**technology mapping**).
 - Complex behavior that is modeled by gates is mapped to complex library cells (e.g. adder, multiplier)

Synthesis of Multiplexors

- **Conditional Operator**

```
module mux_4bits(y, a, b, c, d, sel);
```

```
  input [3:0] a, b, c, d;
```

```
  input [1:0] sel;
```

```
  output [3:0] y;
```

```
  assign y =
```

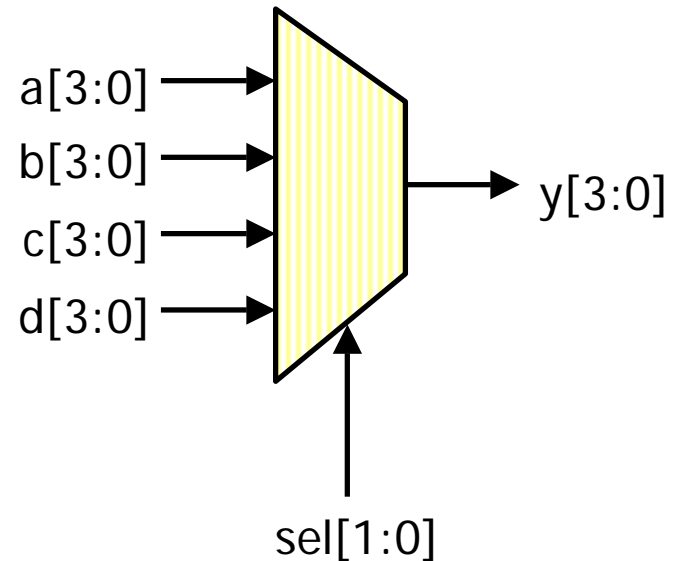
```
    (sel == 0) ? a :
```

```
    (sel == 1) ? b :
```

```
    (sel == 2) ? c :
```

```
    (sel == 3) ? d : 4'bx;
```

```
endmodule
```



Synthesis of Multiplexors (cont.)

- CASE Statement**

```
module mux_4bits (y, a, b, c, d, sel);
```

```
  input [3:0] a, b, c, d;
```

```
  input [1:0] sel
```

```
  output [3:0] y;
```

```
  reg [3:0] y;
```

```
  always @ (a or b or c or d or sel)
```

```
    case (sel)
```

```
      0: y <= a;
```

```
      1: y <= b;
```

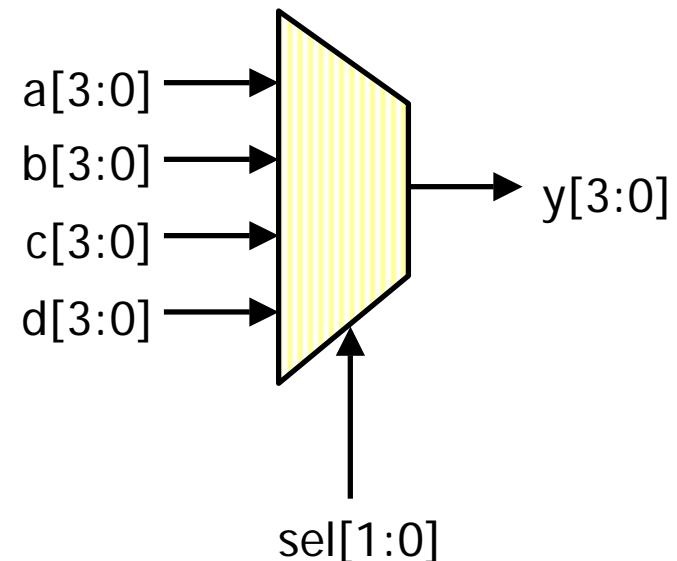
```
      2: y <= c;
```

```
      3: y <= d;
```

```
      default: y <= 4'bx;
```

```
    endcase
```

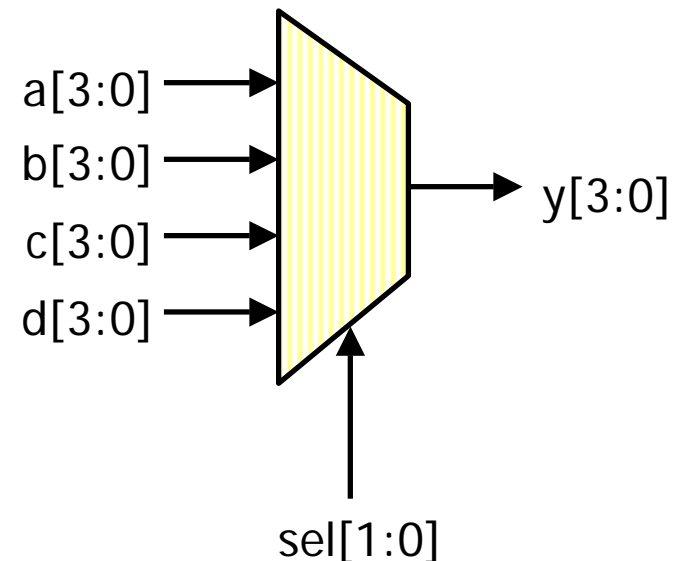
```
endmodule
```



Synthesis of Multiplexors (cont.)

- if .. else Statement

```
module mux_4bits (y, a, b, c, d, sel);  
  input [3:0] a, b, c, d;  
  input [1:0] sel  
  output [3:0] y;  
  reg [3:0] y;  
  always @ (a or b or c or d or sel)  
    if (sel == 0) y <= a; else  
    if (sel == 1) y <= b; else  
    if (sel == 2) y <= c; else  
    if (sel == 3) y <= d;  
    else y <= 4'bX;  
endmodule
```



Note: *CASE* statement and *if/else* statements are more preferred and recommended styles for inferring MUX