1.Verilog Data Types

The data storage and transmission elements found in digital hardware are represented using a set of Verilog Hardware Description Language (HDL) data types. The purpose of Verilog HDL is to design digital hardware.

Data types in Verilog are divided into **NETS** and **Registers**. These data types differ in the way that they are assigned and hold values and also they represent different hardware structures.

The Verilog HDL value set consists of four basic values:

VALUE	DEFINITION
0	Logic zero or false
1	Logic one or true
x	Unknown logical value
z	High impedance of tristate gate

1. **NETS**
   The nets variables represent the physical connection between structural entities. These variables do not store values (except trireg); have the value of their drivers which changes continuously by the driving circuit. Some net data types are wire, tri, wor, trior, wand, triand, tri0, tri1, supply0, supply1 and trireg. **wire** is the most frequently used type. A net data type must be used when a signal is:
   - driven by the output of some device.
   - declared as an input or in-out port.
   - on the left-hand side of a continuous assignment.

2. **Registers**
   The register variables are used in procedural blocks which store values from one assignment to the next. An assignment statement in a procedure acts as a trigger that changes the value of the data storage element. Some register data types are: reg, integer, time and real.reg is the most frequently used type. **Reg** is used for describing logic, **integer** for loop variables and calculations, **real** in system modules, and **time** and **realtime** for storing simulation times in test benches.

2. Array declarations

reg [7:0] mema[0:255]; // declares a memory mema of 256 8-bit registers

reg arrayb[7:0][0:255]; // declare a two dimensional array of one bit registers

wire w_array[7:0][5:0]; // declare array of wires

integer inta[1:64]; // an array of 64 integer values

time chng_hist[1:1000] // an array of 1000 time values

Memory Differences

A memory of n 1-bit regs is different from an n-bit vector reg

reg [1:n] rega; // An n-bit register is not the same

reg mema [1:n]; // as a memory of n 1-bit registers

3. The continuous assignment statement

The continuous assignment statement shall place a continuous assignment on a net data type. This means that whenever an operand in the righthand side expression changes value, the whole right-hand side shall be evaluated and if the new value is different from the previous value, then the new value shall be assigned to the left-hand side.

wire mynet ;

assign mynet = 1b’1 ;

4. Procedural assignments

Procedural assignments occur within procedures such as **always**, **initial**, **task**, and **function** and can be thought of as triggered assignments. The trigger occurs when the flow of execution in the simulation reaches an assignment within a procedure. Reaching the assignment can be controlled by conditional statements. Event controls, delay controls, if statements, case statements, and looping statements can all be used to control whether assignments are evaluated.

Example 1: Declare a 4 bit reg and assign it the value 4.

reg[3:0] a = 4'h4;

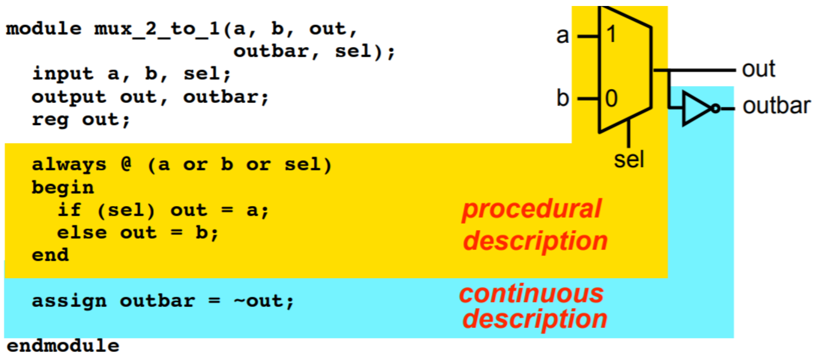
This is equivalent to writing:

reg[3:0] a;

initial a = 4'h4;

There is a significant difference between procedural assignments and continuous assignments:

- Continuous assignments drive nets and are evaluated and updated whenever an input operand changes value.
- Procedural assignments update the value of variables under the control of the procedural flow constructs that surround them.



```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input a, b, sel;
  output out, outbar;
  reg out;

  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end

  assign outbar = ~out;

endmodule
```

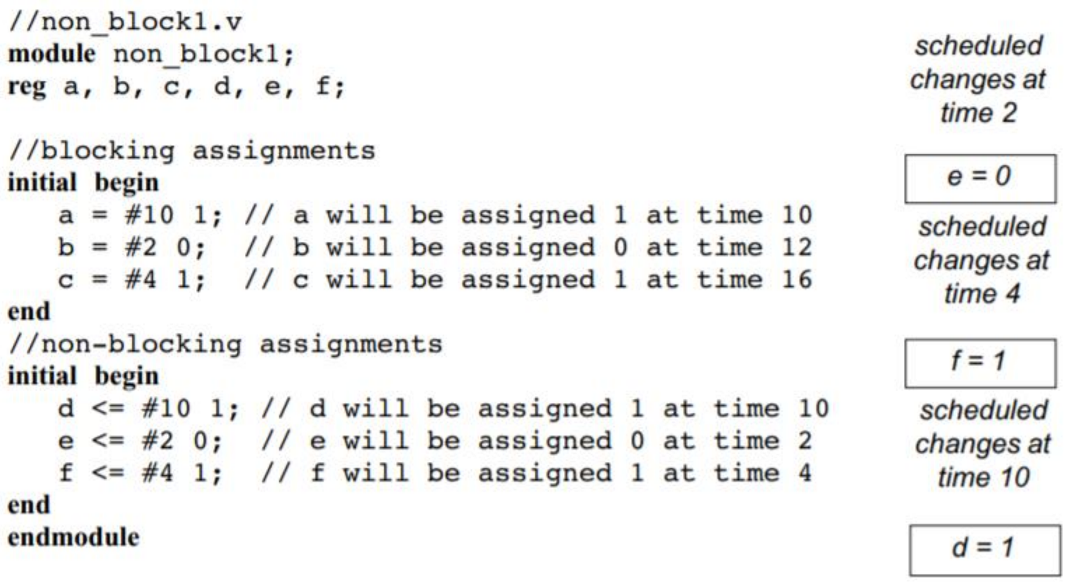
The Verilog HDL contains two types of procedural assignment statements:

Blocking procedural assignment statements: Evaluation and assignment are immediate

Non blocking procedural assignment statements: All assignments deferred until all right-hand sides have been evaluated (end of simulation timestep) It means that nonblocking statements resemble actual hardware more than blocking assignments.

In Verilog, if you want to create sequential logic use a clocked always block with Nonblocking assignments. If you want to create combinational logic use an always block with Blocking assignments. Try not to mix the two in the same always block.

Example 2:

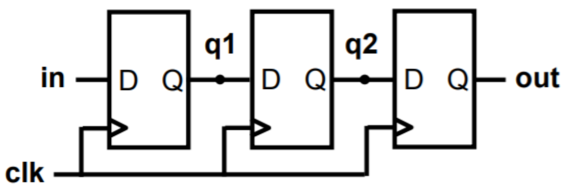


```
//non_block1.v
module non_block1;
reg a, b, c, d, e, f;

//blocking assignments
initial begin
    a = #10 1; // a will be assigned 1 at time 10
    b = #2 0;  // b will be assigned 0 at time 12
    c = #4 1;  // c will be assigned 1 at time 16
end
//non-blocking assignments
initial begin
    d <= #10 1; // d will be assigned 1 at time 10
    e <= #2 0;  // e will be assigned 0 at time 2
    f <= #4 1;  // f will be assigned 1 at time 4
end
endmodule
```

scheduled changes at time 2
e = 0
scheduled changes at time 4
f = 1
scheduled changes at time 10
d = 1

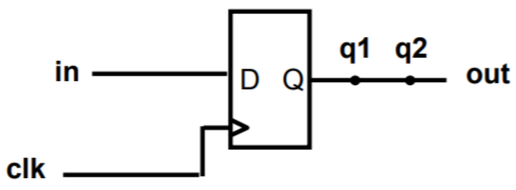
```
always @ (posedge clk)
begin
  q1 <= in;
  q2 <= q1;
  out <= q2;
end
```

"At each rising clock edge, *q1*, *q2*, and *out* simultaneously receive the old values of *in*, *q1*, and *q2*."



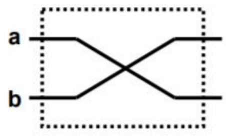
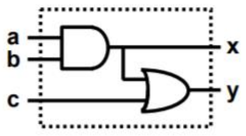
```
always @ (posedge clk)
begin
  q1 = in;
  q2 = q1;
  out = q2;
end
```

"At each rising clock edge, *q1* = *in*.
After that, *q2* = *q1* = *in*.
After that, *out* = *q2* = *q1* = *in*.
Therefore *out* = *in*."



Why two ways of assigning values?

Conceptual need for two kinds of assignment (in `always` blocks):

Blocking: Evaluation and assignment are immediate	~~`a = b`~~
	~~`b = a`~~
		`x = a & b`
		`y = x | c`

Non-Blocking: Assignment is postponed until all r.h.s. evaluations are done	`a <= b`
	`b <= a`
		~~`x <= a & b`~~
		~~`y <= x | c`~~

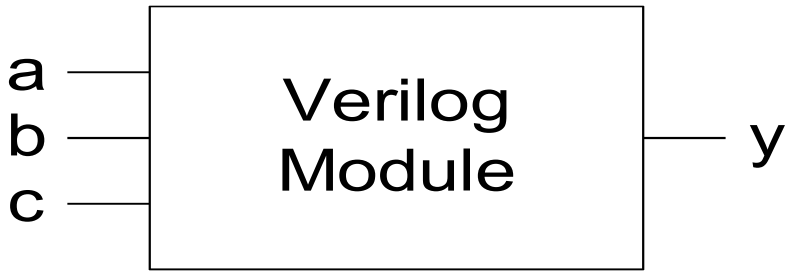
When to use: (only in always blocks!)	**Sequential Circuits**	**Combinational Circuits**

Two Hardware Description Languages

- **Verilog**
  - developed in 1984 by Gateway Design Automation
  - became an IEEE standard (1364) in 1995
  - More popular in US
- **VHDL (VHSIC Hardware Description Language)**
  - Developed in 1981 by the Department of Defense
  - Became an IEEE standard (1076) in 1987
  - More popular in Europe
- **In this course we will use Verilog**

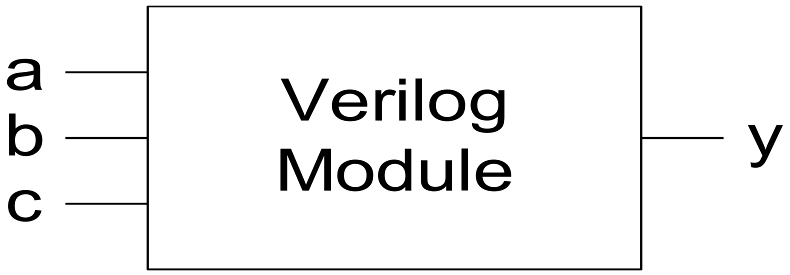
Defining a module

- **A module is the main building block in Verilog**
- **We first need to declare:**
  - Name of the module
  - Types of its connections (input, output)
  - Names of its connections



Defining a module

```
module example (a, b, c, y);
        input a;
        input b;
        input c;
        output y;

// here comes the circuit description

endmodule
```



A question of style

The following two codes are identical

```
module test ( a, b, y );
       input a;
       input b;
       output y;

endmodule
```

```
module test ( input a,
              input b,
              output y );

endmodule
```

What if we have busses?

- **You can also define multi-bit busses.**
  - [range_start : range_end]
- **Example:**

```
input  [31:0] a;    // a[31], a[30] .. a[0]
output [15:8] b1;   // b1[15], b1[14] .. b1[8]
output [7:0]  b2;   // b2[7], b2[6] .. b1[0]
input         clk; // single signal
```

Basic Syntax

- **Verilog is case sensitive:**
  - `SomeName` and `somename` are not the same!
- **Names cannot start with numbers:**
  - `2good` is not a valid name
- **Whitespace is ignored**

```
// Single line comments start with a //

/* Multiline comments
   are defined like this */
```

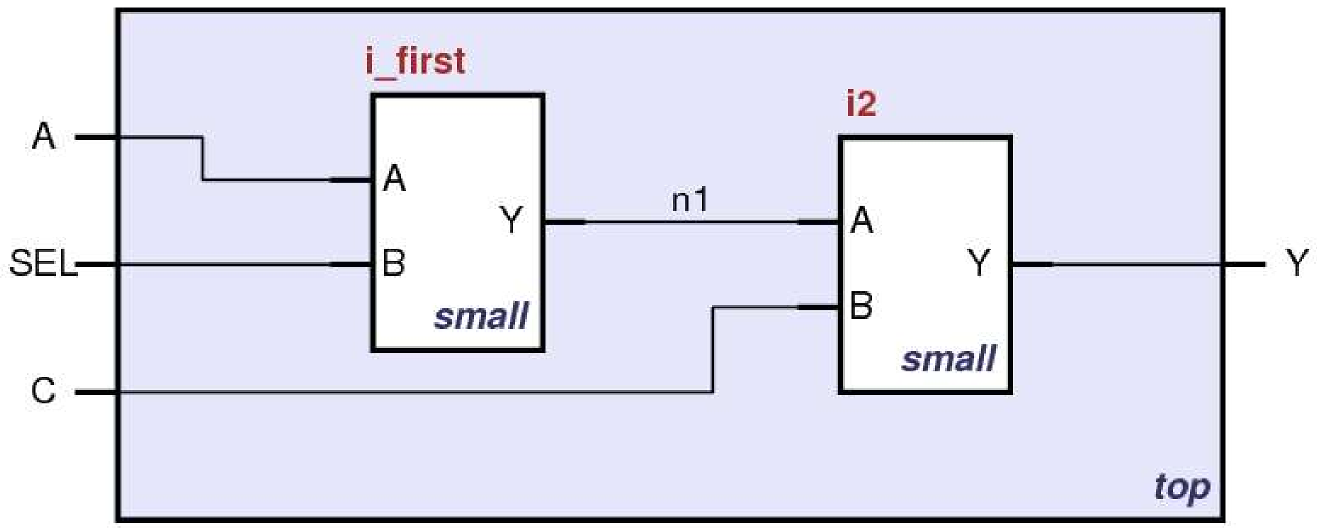
Good Practices

- **Develop/use a consistent naming style**
- **Use MSB to LSB ordering for busses (little-endian)**
  - Try using "`a[31:0]`" and not "`a[0:31]`"
- **Define one module per file**
  - Makes managing your design hierarchy easier
- **Use a file name that equals module name**
  - i.e. module TryThis is defined in a file called TryThis.v

There are Two Main Styles of HDL

- **Structural**
  - Describe how modules are interconnected
  - Each module contains other modules (instances)
  - … and interconnections between these modules
  - Describes a hierarchy
- **Behavioral**
  - The module body contains functional description of the circuit
  - Contains logical and mathematical operators
- **Practical circuits would use a combination of both**

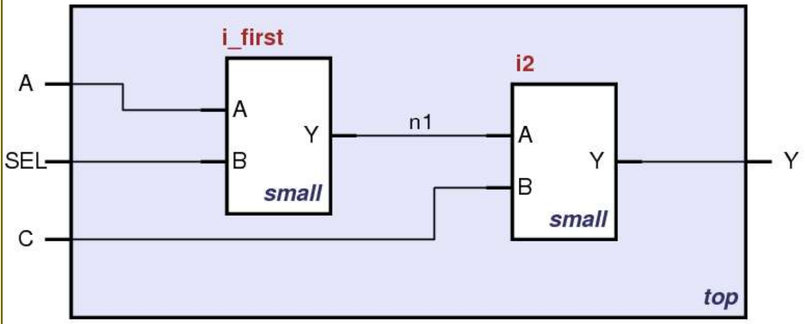
Structural HDL: Instantiating a Module



Structural HDL Example

Module Definitions

```
module top (A, SEL, C, Y);
  input A, SEL, C;
  output Y;
  wire n1;









endmodule
```

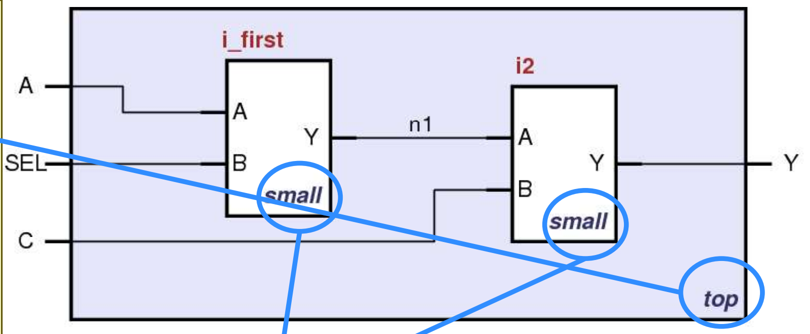


```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

Structural HDL Example

Module Definitions

```
module top (A, SEL, C, Y);
  input A, SEL, C;
  output Y;
  wire n1;









endmodule
```

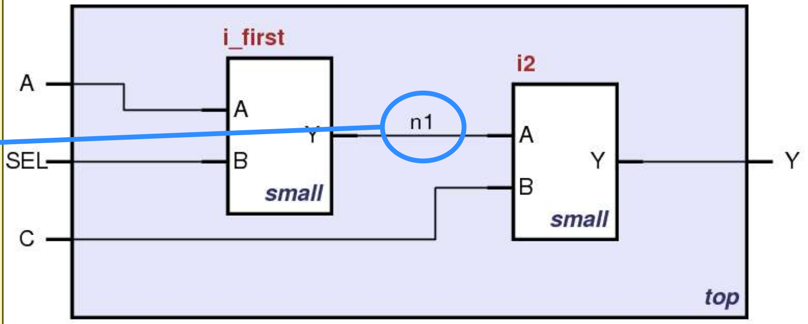


```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

Structural HDL Example

Wire definitions

```
module top (A, SEL, C, Y);
  input A, SEL, C;
  output Y;
  wire n1;









endmodule
```

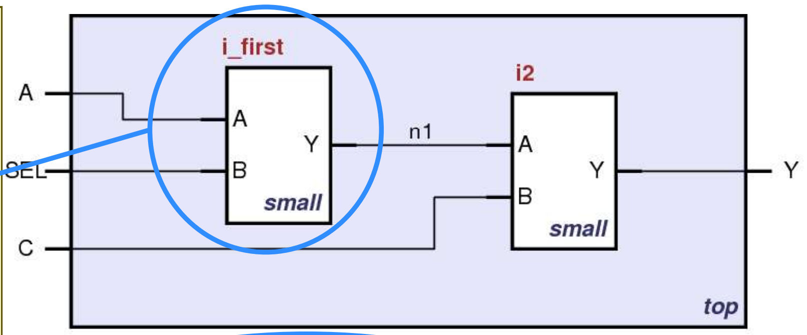


```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

Structural HDL Example

Instantiate first module

```
module top (A, SEL, C, Y);
  input A, SEL, C;
  output Y;
  wire n1;

// instantiate small once
small i_first ( .A(A),
                .B(SEL),
                .Y(n1)   );




endmodule
```

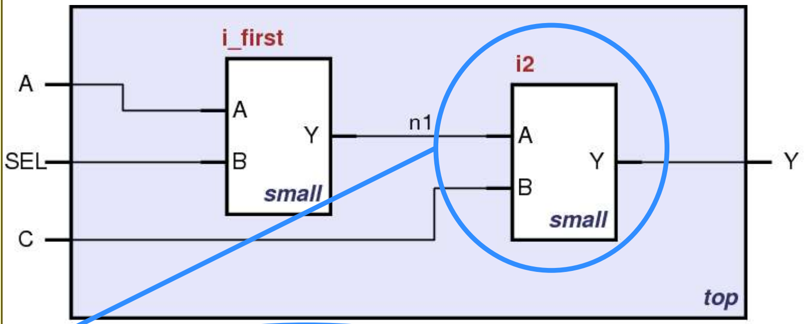


```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

Structural HDL Example

Instantiate second module

```
module top (A, SEL, C, Y);
  input A, SEL, C;
  output Y;
  wire n1;

// instantiate small once
small i_first ( .A(A),
                .B(SEL),
                .Y(n1)   );

// instantiate small second time
small i2 ( .A(n1),
           .B(C),
           .Y(Y) );

endmodule
```

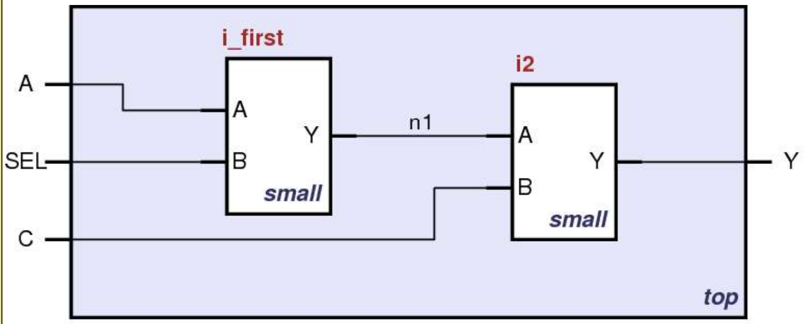


```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

Structural HDL Example

Short Instantiation

```
module top (A, SEL, C, Y);
  input A, SEL, C;
  output Y;
  wire n1;

// alternative
small i_first ( A, SEL, n1 );

/* Shorter instantiation,
   pin order very important */

// any pin order, safer choice
small i2 ( .B(C),
           .Y(Y),
           .A(n1) );

endmodule
```



```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

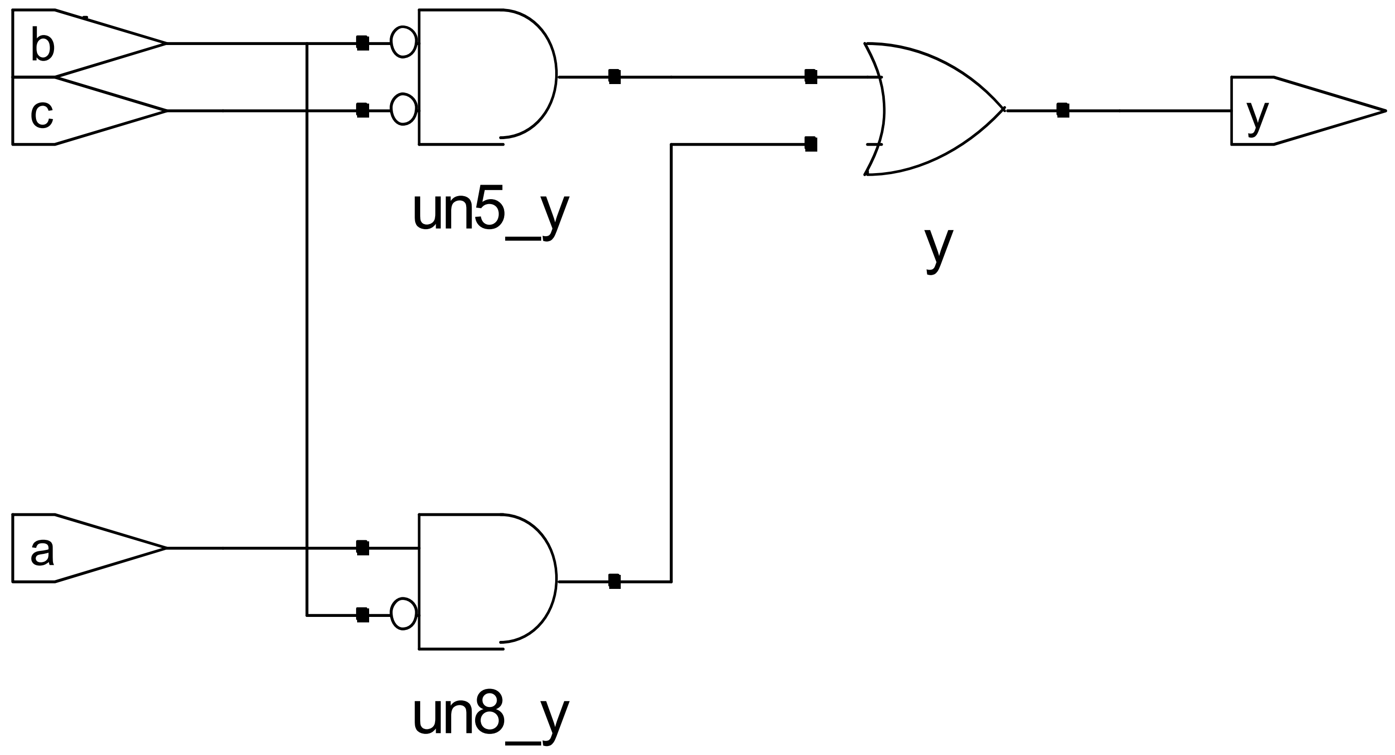
What Happens with HDL code?

- **Automatic Synthesis**
  - Modern tools are able to map a behavioral HDL code into gate-level schematics
  - They can perform many optimizations
  - … however they can not guarantee that a solution is optimal
  - Most common way of Digital Design these days
- **Simulation**
  - Allows the behavior of the circuit to be verified without actually manufacturing the circuit
  - Simulators can work on behavioral or gate-level schematics

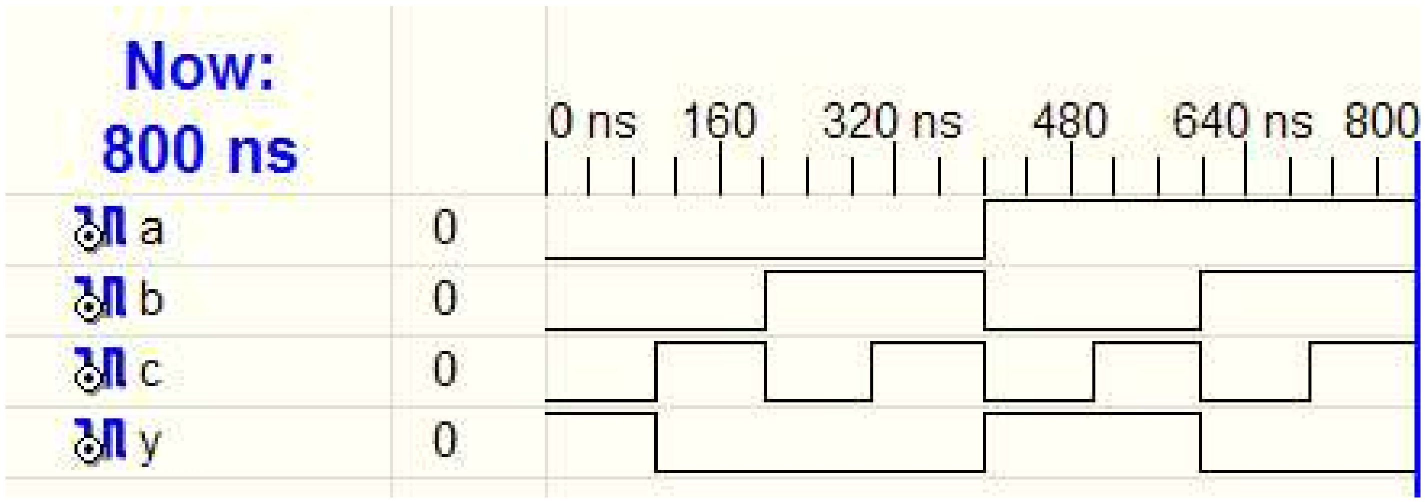
Behavioral HDL: Defining Functionality

```
module example (a, b, c, y);
       input a;
       input b;
       input c;
       output y;

// here comes the circuit description
assign y = ~a & ~b & ~c |
            a & ~b & ~c |
            a & ~b &  c;

endmodule
```

Behavioral HDL: Synthesis Results



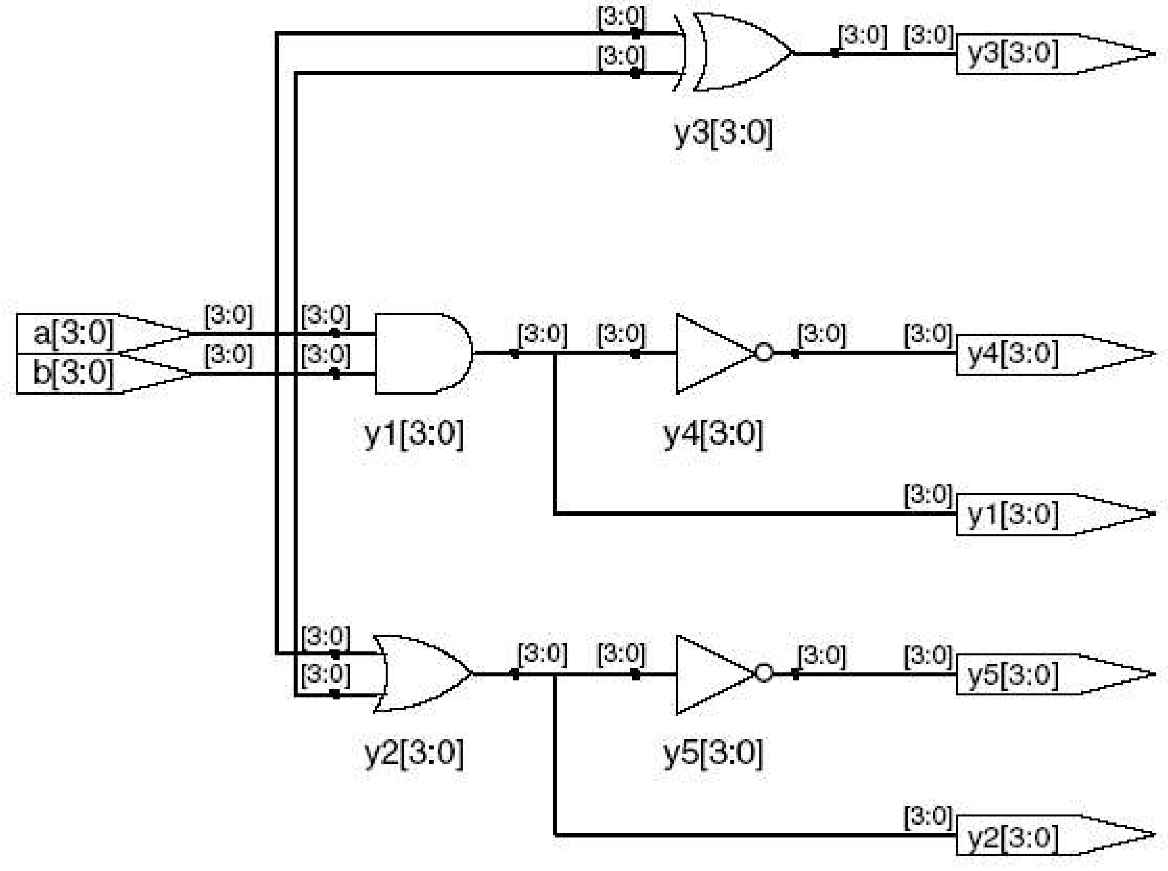
Behavioral HDL: Simulating the Circuit



Bitwise Operators

```
module gates(input  [3:0]  a, b,
             output [3:0] y1, y2, y3, y4, y5);

   /* Five different two-input logic
      gates acting on 4 bit busses */

   assign y1 = a & b;     // AND
   assign y2 = a | b;     // OR
   assign y3 = a ^ b;     // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR

endmodule
```

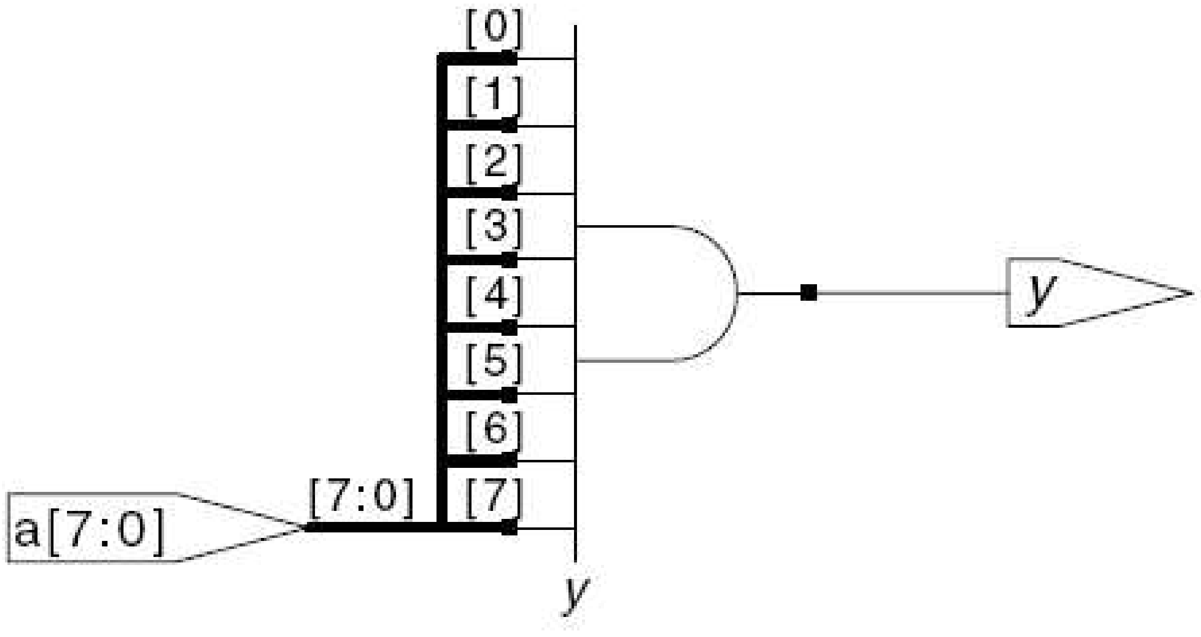
Bitwise Operators: Synthesis Results



Reduction Operators

```
module and8(input  [7:0] a,
            output       y);

   assign y = &a;

   // &a is much easier to write than
   // assign y = a[7] & a[6] & a[5] & a[4] &
   //            a[3] & a[2] & a[1] & a[0];

endmodule
```

Reduction Operators: assign y = &a;

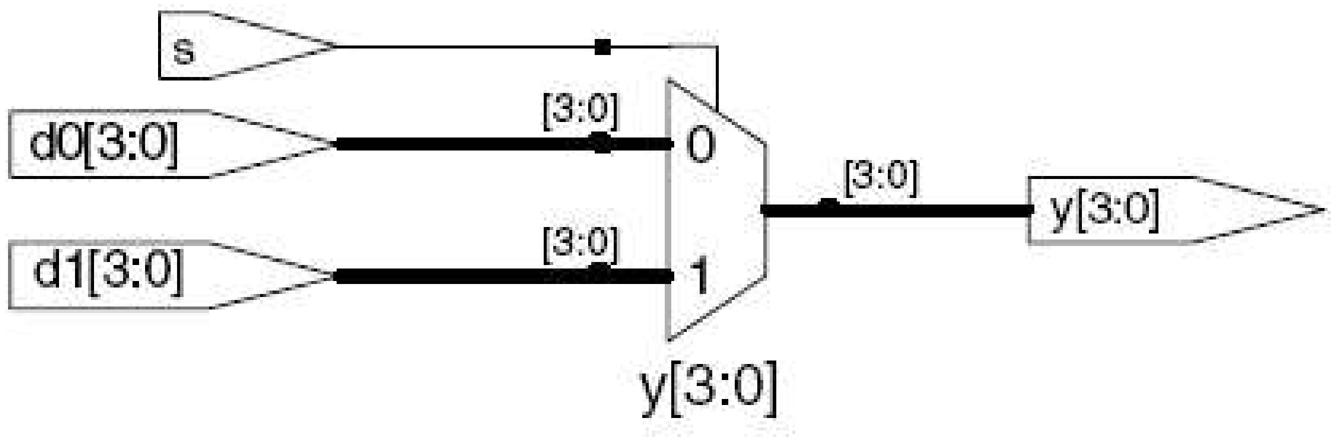


Conditional Assignment

```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);

   assign y = s ? d1 : d0;
   // if (s) then y=d1 else y=d0;

endmodule
```

- **? : is also called a ternary operator as it operates on three inputs:**
  - s
  - d1
  - d0.

Conditional Assignment: y = s ? d1: d0;



More Conditional Assignments

```
module mux4(input  [3:0] d0, d1, d2, d3
            input  [1:0] s,
            output [3:0] y);

   assign y = s[1] ? ( s[0] ? d3 : d2)
                   : ( s[0] ? d1 : d0);
  // if (s1) then
  //      if (s0) then y=d3 else y=d2
  // else
  //      if (s0) then y=d1 else y=d0

endmodule
```

Even More Conditional Assignments

```
module mux4(input  [3:0] d0, d1, d2, d3
            input  [1:0] s,
            output [3:0] y);

   assign y = (s == 2'b11) ? d3 :
              (s == 2'b10) ? d2 :
              (s == 2'b01) ? d1 :
              d0;
// if      (s = "11" ) then y= d3
// else if (s = "10" ) then y= d2
// else if (s = "01" ) then y= d1
// else                     y= d0

endmodule
```

How to Express numbers ?

N’ Bxx

`8’b0000_0001`

- **(N) Number of bits**
  - Expresses how many bits will be used to store the value
- **(B) Base**
  - Can be b (binary), h (hexadecimal), d (decimal), o (octal)
- **(xx) Number**
  - The value expressed in base, apart from numbers it can also have X and Z as values.
  - Underscore _ can be used to improve readability

Number Representation in Verilog

Verilog	Stored Number	Verilog	Stored Number
4’b1001	1001	4’d5	0101
8’b1001	0000 1001	12’hFA3	1111 1010 0011
8’b0000_1001	0000 1001	8’o12	00 001 010
8’bxX0X1zZ1	XX0X 1ZZ1	4’h7	0111
‘b01	0000 .. 0001	12’h0	0000 0000 0000

What have seen so far:

- **Describing structural hierarchy with Verilog**
  - Instantiate modules in an other module
- **Writing simple logic equations**
  - We can write AND, OR, XOR etc
- **Multiplexer functionality**
  - If … then … else
- **We can describe constants**
- **But there is more:**

Precedence of operations in Verilog

Highest

~	NOT
*, /, %	mult, div, mod
+, -	add,sub
<<, >>	shift
<<<, >>>	arithmetic shift
<, <=, >, >=	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
|, ~|	OR, NOR
?:	ternary operator

Lowest

Example: Comparing two numbers

An XNOR gate

```
module MyXnor (input a, b,
               output z);

   assign z = ~(a ^ b); //not XOR

endmodule
```

An AND gate

```
module MyAnd (input a, b,
              output z);

   assign z = a & b;    // AND

endmodule
```

Example: Comparing Two Numbers

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
                output eq);
       wire c0, c1, c2, c3, c01, c23;

MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
MyAnd haha (.A(c0), .B(c1), .Z(c01) ); // AND
MyAnd hoho (.A(c2), .B(c3), .Z(c23) ); // AND
MyAnd bubu (.A(c01), .B(c23), .Z(eq) ); // AND

endmodule
```

Example: Comparing Two Numbers

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
                output eq);
       wire c0, c1, c2, c3, c01, c23;

MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
assign c01 = c0 & c1;
assign c23 = c2 & c3;
assign eq  = c01 & c23;

endmodule
```

Example: Comparing Two Numbers

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
                output eq);
       wire c0, c1, c2, c3;

MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR

assign eq  = c0 & c1 & c2 & c3;


endmodule
```

Example: Comparing Two Numbers

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
                output eq);
       wire [3:0] c; // bus definition

MyXnor i0 (.A(a0), .B(b0), .Z(c[0]) ); // XNOR
MyXnor i1 (.A(a1), .B(b1), .Z(c[1]) ); // XNOR
MyXnor i2 (.A(a2), .B(b2), .Z(c[2]) ); // XNOR
MyXnor i3 (.A(a3), .B(b3), .Z(c[3]) ); // XNOR

assign eq  = &c; // short format


endmodule
```

Example: Comparing Two Numbers

```
module compare (input [3:0] a, input [3:0] b,
                output eq);
       wire [3:0] c; // bus definition

MyXnor i0 (.A(a[0]), .B(b[0]), .Z(c[0]) ); // XNOR
MyXnor i1 (.A(a[1]), .B(b[1]), .Z(c[1]) ); // XNOR
MyXnor i2 (.A(a[2]), .B(b[2]), .Z(c[2]) ); // XNOR
MyXnor i3 (.A(a[3]), .B(b[3]), .Z(c[3]) ); // XNOR

assign eq  = &c; // short format


endmodule
```

Example: Comparing Two Numbers

```
module compare (input [3:0] a, input [3:0] b,
                output eq);
       wire [3:0] c; // bus definition



assign c = ~(a ^ b); // XNOR


assign eq  = &c; // short format


endmodule
```

Example: Comparing Two Numbers

```
module compare (input [3:0] a, input [3:0] b,
                output eq);




assign eq = (a == b) ? 1 : 0; // really short





endmodule
```

What is the BEST way of writing Verilog

- **Quite simply IT DOES NOT EXIST!**
- **Code should be easy to understand**
  - Sometimes longer code is easier to comprehend
- **Hierarchy is very useful**
  - In the previous example it did not look like that, but for larger designs it is indispensible
- **Try to stay closer to hardware**
  - After all the goal is to design hardware