Verilog for Combinational Circuits

Digital Design and Computer Architecture
Mohammad Sadrosadati
Frank K. Gürkaynak

http://safari.ethz.ch/ddca
What will we learn?

- Introduction to Verilog
- Combinational Logic in Verilog
- Structural Modeling
Hardware Description Languages

- In the beginning HDLs were developed as a ‘standard way’ of drawing circuit schematics.
- Modeled the interface of circuits, described how they were connected
- Allowed connections between these modules
- Supported some common logic functions
  - AND OR NOT XOR
  - Multiplexers
Convenient Way of Drawing Schematics
Convenient Way of Drawing Schematics

- **It is standard**
  - Everybody will interpret the schematic the same way

- **It is not proprietary**
  - HDLs are not tool specific

- **It is machine readable**
  - It is easier for computers to understand the circuit

- **Only later on additional benefits were discovered**
  - Simulation and Synthesis
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

```verilog
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*

```verilog
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:

```vhdl
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**

```verilog
// 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**

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

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

  // description of small
endmodule
```
**Structural HDL Example**

**Instantiate first module**

```verilog
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
```

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

    // description of small
endmodule
```

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**

```verilog
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
```

```verilog
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
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

<table>
<thead>
<tr>
<th>Now: 800 ns</th>
<th>0 ns</th>
<th>160 ns</th>
<th>320 ns</th>
<th>480 ns</th>
<th>640 ns</th>
<th>800 ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bitwise Operators

```verilog
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

dendmodule
```
Bitwise Operators: Synthesis Results
Reduction Operators

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

    assign y = &a;

    // &a is much easier to write than

endmodule
Reduction Operators: assign $y = \&a;$
### Conditional Assignment

```verilog
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: $y = s \ ? \ d1: d0;$
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
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?

\[N'B_{xx}\]

\[8'b0000_0001\]

- **(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

<table>
<thead>
<tr>
<th>Verilog</th>
<th>Stored Number</th>
<th>Verilog</th>
<th>Stored Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>4’ b1001</td>
<td>1001</td>
<td>4’ d5</td>
<td>0101</td>
</tr>
<tr>
<td>8’ b1001</td>
<td>0000 1001</td>
<td>12’ hFA3</td>
<td>1111 1010 0011</td>
</tr>
<tr>
<td>8’ b0000_1001</td>
<td>0000 1001</td>
<td>8’ o12</td>
<td>00 001 010</td>
</tr>
<tr>
<td>8’ bxX0X1zZ1</td>
<td>XX0X 1ZZ1</td>
<td>4’ h7</td>
<td>0111</td>
</tr>
<tr>
<td>‘b01</td>
<td>0000 .. 0001</td>
<td>12’ h0</td>
<td>0000 0000 0000</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Highest</th>
<th>Lowest</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>ternary operator</td>
</tr>
<tr>
<td>*, /, %</td>
<td></td>
</tr>
<tr>
<td>mult, div, mod</td>
<td></td>
</tr>
<tr>
<td>+, -</td>
<td></td>
</tr>
<tr>
<td>add, sub</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;&lt;, &gt;&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>arithmetic shift</td>
<td></td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td></td>
</tr>
<tr>
<td>comparison</td>
<td></td>
</tr>
<tr>
<td>==, !=</td>
<td></td>
</tr>
<tr>
<td>equal, not equal</td>
<td></td>
</tr>
<tr>
<td>&amp;, &amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>AND, NAND</td>
<td></td>
</tr>
<tr>
<td>^, ^^</td>
<td></td>
</tr>
<tr>
<td>XOR, XNOR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>, ~</td>
</tr>
<tr>
<td>OR, NOR</td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td></td>
</tr>
</tbody>
</table>
Example: Comparing two numbers

### An XNOR gate

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

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

endmodule
```

### An AND gate

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

    assign z = a & b; // AND

endmodule
```
Example: Comparing Two Numbers

```verilog
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

```verilog
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

```verilog
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

```verilog
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

```verilog
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

```verilog
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

```verilog
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 indispensable

- Try to stay closer to hardware
  - After all the goal is to design hardware
Parameterized Modules

We can pass parameters to a module
Parameterized Modules: Instantiating

module mux2
  #(parameter width = 8) // name and default value
  (input [width-1:0] d0, d1,
   input s,
   output [width-1:0] y);

  assign y = s ? d1 : d0;
endmodule

// If parameter is not given, default is assumed (here 8)
mux2 i_mux (d0, d1, s, out);

// The same module with 12-bit bus width:
mux2 #(12) i_mux_b (d0, d1, s, out);

// More verbose version:
mux2 #(.width(12)) i_mux_b (.d0(d0), .d1(d1),
   .s(s), .out(out));
Manipulating Bits

// You can assign partial busses
wire [15:0] longbus;
wire [7:0] shortbus;
assign shortbus = longbus[12:5];

// Concatenating is by {}
assign y = {a[2],a[1],a[0],a[0]};

// Possible to define multiple copies
assign x = {a[0], a[0], a[0], a[0]}
assign y = { 4{a[0]} }
Z floating output

module tristate(input [3:0] a,
               input en,
               output [3:0] y);

    assign y = en ? a : 4'bz;

endmodule

en

a[3:0]  [3:0]  [3:0]  [3:0]  y[3:0]
y_1[3:0]
### Truth Table for AND with Z and X

<table>
<thead>
<tr>
<th>&amp;</th>
<th>A</th>
<th>B</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Z</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
What About Timing?

- It is possible to define timing relations in Verilog
  - These are **ONLY** for Simulation
  - They **CAN NOT** be synthesized
  - They are used for modeling delays in simulation

```verilog
	timescale 1ns/1ps
module simple (input a, output z1, z2);

assign #5 z1 = ~a; // inverted output after 5ns
assign #9 z2 = a;  // output after 9ns
endmodule
```
Next Steps

- We have seen an overview of Verilog
- Discussed behavioral and structural modeling
- Showed combinational logic constructs

Still to come: (later)

- Sequential circuit description in Verilog
- Developing testbenches for simulation