Basic Logic Design with Verilog

TA: Chihhao Chao
chihhao@access.ee.ntu.edu.tw

Lecture note ver.1 by Chen-han Tsai
ver.2 revised by Chih-hao Chao

Lecture Note on Verilog, Course #90132300, EE, NTU, C.H. Chao
Outline

- Introduction to HDL/Verilog
- Gate Level Modeling
- Behavioral Level Modeling
- Test bench
- Summary and Notes
Introduction to HDL/Verilog
What is HDL/Verilog

Why use HDL (Hardware Description Language)?
- Design abstraction: HDL ↔ layout by human
- Hardware modeling
- Reduce cost and time to design hardware

Verilog is one of the most popular HDLs
- VHDL (another popular HDL)

Key features of Verilog
- Supports various levels of abstraction
  - Behavior level
  - Register transfer level
  - Gate level
  - Switch level
- Simulate design functions
An Example
1-bit Multiplexer

if (sel==0)
    out = in1;
else
    out = in2;

out = (sel’·in1) + (sel·in2)
Gate Level Description

Gate Level: you see only netlist (gates and wires) in the code

```verilog
module mux2(out,in1,in2,sel);
    output out;
    input  in1,in2,sel;

    and a1(a1_o,in1,sel);
    not n1(iv_sel,sel);
    and a2(a2_o,in2,iv_sel);
    or  o1(out,a1_o,a2_o);
endmodule
```
Behavioral Level/RTL Description

**Behavioral Level/RTL Description**

*Behavioral Level/RTL Description*  

**RTL:** you may see high level behavior in the code  

**Behavior:** event-driven behavior description construct

```verilog
module mux2(out, in1, in2, sel);
  output out;
  input in1, in2, sel;
  reg out;

  always@(in1 or in2 or sel)
    begin
      if (sel) out = in1;
      else    out = in2;
    end
endmodule
```

```verilog
module mux2(out, in1, in2, sel);
  output out;
  input in1, in2, sel;

  assign out = sel ? in1 : in2;
endmodule
```
Verilog HDL Syntax
A Simple Verilog Code

```
module mux2(out, in1, in2, sel);
output out;
input in1, in2, sel;
reg out;

always@(in1 or in2 or sel)
begin
  if (sel) out = in1;
  else out = in2;
end
endmodule
```
Module

- Basic building block in Verilog.

- Module
  1. Created by “declaration” (can’t be nested)
  2. Used by “instantiation“

- Interface is defined by ports
- May contain instances of other modules
- All modules run concurrently
Instances

A module provides a template from which you can create actual objects.

When a module is invoked, Verilog creates a unique object from the template.

Each object has its own name, variables, parameters and I/O interface.
Module Instantiation

```
module adder(out, in1, in2);
    output out;
    input in1, in2, sel;
    assign out = in1 + in2;
endmodule
```

```
module adder_tree(out0, out1, in1, in2, in3, in4);
    output out0, out1;
    input in1, in2, in3, in4;
    adder add_0(out0, in1, in2);
    adder add_1(out1, in3, in4);
endmodule
```
Analogy: module ↔ class

As module is to Verilog HDL, so class is to C++ programming language.

<table>
<thead>
<tr>
<th>Format</th>
<th>module m_Name( IO list );</th>
<th>class c_Name {</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>endmodule</td>
<td>}</td>
</tr>
<tr>
<td>Instantiation</td>
<td>m_Name ins_name ( port connection list );</td>
<td>c_Name obj_name;</td>
</tr>
<tr>
<td>Member</td>
<td>ins_name.member_signal</td>
<td>obj_name.member_data</td>
</tr>
<tr>
<td>Hierachy</td>
<td>instance.sub_instance.me mber_signal</td>
<td>object.sub_object.member_data</td>
</tr>
</tbody>
</table>
Analogy: module ↔ class

```cpp
class c_AND_gate {
    bool in_a;
    bool in_b;
    bool out;
    void evaluate() { out = in_a && in_b; }
};
```

```verilog
module m_AND_gate (in_a, in_b, out);
    input in_a;
    input in_b;
    output out;
    assign out = in_a & in_b;
endmodule
```

Model AND gate with C++          Model AND gate with Verilog HDL

assign and evaluate() is simulated/called at each $T_{i+1} = T_i + t_{\text{resolution}}$
Port Connection

- Connect module port by order list
  - FA1 fa1(c_o, sum, a, b, c_i);

- Not fully connected
  - FA1 fa3(c_o,, a, b, c_i);

- Connect module port by name .PortName( NetName )
  - FA1 fa2(.A(a), .B(b), .CO(c_o),.Cl(c_i), .S(sum));
  - Recommended
Verilog Language Rule

- Case sensitive
- Identifiers:
  - Digits 0...9
  - Underscore _
  - Upper and lower case letters from the alphabet

- Terminate statement/declaration with semicolon “;”

- Comments:
  - Single line: // it’s a single line comment example
  - Multi-line: /* when the comment exceeds single line, multiline comment is necessary*/
Register and Net

Registers
- Keyword: `reg`, `integer`, `time`, `real`
- Event-driven modeling
- Storage element (modeling sequential circuit)
- Assignment in “always” block

Nets
- Keyword: `wire`, `wand`, `wor`, `tri`
- `triand`, `trior`, `supply0`, `supply1`
- Doesn’t store value, just a connection
- Input, output, inout are default “wire”
- Can’t appear in “always” block assignment
Four-valued Logic

Verilog’s nets and registers hold four-valued data

- **0**: represent a logic zero or false condition
- **1**: represent a logic zero or false condition
- **z**: Output of an undriven tri-state driver – high-impedance value
  - Models case where nothing is setting a wire’s value
- **x**: Models when the simulator can’t decide the value – uninitialized or unknown logic value
  - Initial state of registers
  - When a wire is being driven to 0 and 1 simultaneously
  - Output of a gate with z inputs
Logic System

- Four values: 0, 1, x or X, z or Z // Not case sensitive here
  - The logic value x denotes an unknown (ambiguous) value
  - The logic value z denotes a high impedance

- Primitives have built-in logic
- Simulators describe 4-value logic (see Appendix A in text)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Number Representation

Format: <size>’<base_format><number>

- `<size>` - decimal specification of number of bits
  - Default is unsized and machine-dependent but at least 32 bits

- `<base format>` - ' followed by arithmetic base of number
  - `<d>` `<D>` - decimal - default if no `<base_format>` given
  - `<h>` `<H>` - hexadecimal
  - `<o>` `<O>` - octal
  - `<b>` `<B>` - binary

- `<number>` - value given in base of `<base_format>
  - _ can be used for reading clarity
  - `x`, `z` is automatically extented
Number Representation

Examples:
- 6'b010_111 gives 010111
- 8'b0110 gives 00000110
- 4'bx01 gives xx01
- 16'H3AB gives 0000001110101011
- 24 gives 0...0011000
- 5'O36 gives 11110
- 16'Hx gives xxxxxxxxxxxxxxxxxxx
- 8'hz gives zzzzzzzzzz
Value and Number Expressions: Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>659</td>
<td>// unsized decimal</td>
<td>27_195_000</td>
<td>// underline usage</td>
</tr>
<tr>
<td>‘h 837ff</td>
<td>// unsized hexadecimal</td>
<td>16'b0001_0101_0001_1111</td>
<td></td>
</tr>
<tr>
<td>‘o 7460</td>
<td>// unsized octal</td>
<td>32’h12ab_f001</td>
<td></td>
</tr>
<tr>
<td>4af</td>
<td>// illegal syntax</td>
<td></td>
<td>// X and Z is sign-extended</td>
</tr>
<tr>
<td>4’b1001</td>
<td>// 4-bit binary</td>
<td></td>
<td>reg [11:0] a;</td>
</tr>
<tr>
<td>5’D 3</td>
<td>// 5-bit decimal</td>
<td></td>
<td>initial</td>
</tr>
<tr>
<td>3’b01x</td>
<td>// 3-bit number with unknown LSB</td>
<td></td>
<td>begin</td>
</tr>
<tr>
<td>12’hx</td>
<td>// 12-bit unknown</td>
<td></td>
<td>a = ‘hx;</td>
</tr>
<tr>
<td>8’d -6</td>
<td>// illegal syntax</td>
<td></td>
<td>// yields xxx</td>
</tr>
<tr>
<td>-8’d 6</td>
<td>// phrase as - (8’d 6)</td>
<td></td>
<td>a = ‘h3x;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>// yields 03x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a = ‘h0x;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>// yields 00x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>
Net Concatenations:
An Easy Way to Group Nets

Representations | Meanings
--- | ---
{b[3:0],c[2:0]} | {b[3],b[2],b[1],b[0],c[2],c[1],c[0]}
{a,b[3:0],w,3'b101} | {a,b[3],b[2],b[1],b[0],w,1'b1,1'b0,1'b1}
{4{w}} | {w,w,w,w}
{b,{3{a,b}}} | {b,1'b0,1'b1,1'b0,1'b1}

Module A

Module B

Module C
### Operators

<table>
<thead>
<tr>
<th>Category</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic Operators</td>
<td>+, -, *, /, %</td>
</tr>
<tr>
<td>Relational Operators</td>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
</tr>
<tr>
<td>Equality Operators</td>
<td>==, !=, ===, !==</td>
</tr>
<tr>
<td>Logical Operators</td>
<td>!, &amp;&amp;,</td>
</tr>
<tr>
<td>Bit-Wise Operators</td>
<td>~, &amp;,</td>
</tr>
<tr>
<td>Unary Reduction</td>
<td>&amp;!, ~&amp;!,</td>
</tr>
<tr>
<td>Shift Operators</td>
<td>&gt;&gt;, &lt;&lt;</td>
</tr>
<tr>
<td>Conditional Operators</td>
<td>?!</td>
</tr>
<tr>
<td>Concatenations</td>
<td>{}</td>
</tr>
</tbody>
</table>
Operators (cont.)

- Example

<table>
<thead>
<tr>
<th>opa = 0010</th>
<th>opb = 1100</th>
<th>opc = 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>all bits are 0 $\rightarrow$ logic false</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Unary reduction
  - $\&$ opa = 0
  - $0 \& 0 \& 1 \& 0 = 0$

- Logical operation
  - opa $\&\&$ opc = 0
  - opa = 0010 $\rightarrow$ true
  - opc = 0000 $\rightarrow$ false
  - true $\&\&$ false = false

- Bit-wise operation
  - opa $\&$ opb = 0000
  - 0000
  - $\&$ 1100
  - 0000

- Logical operation
  - opa $\&\&$ opb = 1
  - opa = 0010 $\rightarrow$ true
  - opb = 1100 $\rightarrow$ true
  - true $\&\&$ true = true

- Logical operation
  - $\sim$ opa = 1101
  - ! opa = 0
Compiler Directives

`define
- `define RAM_SIZE 16
- Defining a name and gives a constant value to it.

`include
- `include adder.v
- Including the entire contents of other verilog source file.

`timescale
- `timescale 100ns/1ns
- Setting the reference time unit and time precision of your simulation.
System Tasks

$monitor

$monitor ($time,"%d %d %d", address, sinout, cosout);
Displays the values of the argument list whenever any of the arguments change except $time.

$display

$display ("%d %d %d", address, sinout, cosout);
Prints out the current values of the signals in the argument list

$finish

$finish
Terminate the simulation
Gate Level Modeling

Gate Level Modeling
Case Study
Gate Level Modeling

Steps

- Develop the boolean function of output
- Draw the circuit with logic gates/primitives
- Connect gates/primitives with net (usually wire)

HDL: Hardware Description Language

- Figure out architecture first, then write code.
Primitives

- Primitives are modules ready to be instanced
- Smallest modeling block for simulator
- Verilog build-in primitive gate
  - and, or, not, buf, xor, nand, nor, xnor
  - `prim_name inst_name( output, in0, in1,.... );`
- User defined primitive (UDP)
  - building block defined by designer
Case Study
1-bit Full Adder

<table>
<thead>
<tr>
<th>Ci</th>
<th>A</th>
<th>B</th>
<th>Co</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Case Study
1-bit Full Adder

\[ co = (a \cdot b) + (b \cdot ci) + (ci \cdot a); \]
Case Study

1-bit Full Adder

\[ \text{sum} = a \oplus b \oplus ci \]

```
module FA_sum (sum, a, b, ci);
  input a, b, ci;
  output sum, co;
  xor g1( sum, a, b, ci );
endmodule
```
Case Study
1-bit Full Adder

Full Adder Connection
- Instance ins_c from FA_co
- Instance ins_s from FA_sum

module FA_gatelevel( sum, co, a, b, ci);
input  a, b, ci;
output sum, co;

FA_co ins_c( co, a, b, ci );
FA_sum ins_s( sum, a, b, ci );
endmodule
RT-Level & Behavioral Level Modeling

RT-Level & Behavioral Level Modeling
Case Study
RT-Level & Behavioral Level Modeling

High level description
- User friendly
- Concise code
- Faster simulation speed (event driven)

Widely used for some common operations
- +, -, *, &

Two main formats
- always block (for behavior level)
- assign (for RT level)
Case Study
1-bit Full Adder

\[ \{C_0, S\} = A + B + C_i \]
Case Study

1-bit Full Adder

RT-level modeling of combinational circuit

Describe boolean function with operators and use continuous assignment `assign`

```
module FA_rtlevel( sum, co, a, b, ci );

  input  a, b, ci;
  output sum, co;

  assign { co, sum } = a + b + cin;

endmodule
```
Case Study
1-bit Full Adder

Behavior-level modeling of combinational circuit:
- Use event-driven construct: `always` block
- Event: `@( sensitive_list )`

```verilog
define module FA_behavior( sum, co, a, b, ci );
    input a, b, ci;
    output sum, co;
    reg sum, co;

    always@ ( a or b or ci )
        { co, sum } = a + b + ci;
endmodule
```
Test bench
Test Methodology

- Systematically verify the functionality of a model.

Simulation:
1. detect syntax violations in source code
2. simulate behavior
3. monitor results

Diagram:
- Test bench
  - data_i
  - input ports
  - Design Top Module
  - answer_o
  - output ports
  - Equal?
  - data_o
Verilog Simulator

Circuit Description

module add4 (sum, carry, A, B, cin);
output [3:0] sum;
    . . . .
endmodule

Testfixture

module testfixture;
reg [3:0] A, B;
    . . . .
endmodule

Graphical Simulation Result

Text Mode Simulation Result

<table>
<thead>
<tr>
<th>Time</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00 ns</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>16.00 ns</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>100.00 ns</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>. . . .</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>
Testbench for Full Adder

```verilog
module t_full_add();
reg   a, b, cin; // for stimulus waveforms
wire  sum, c_out;

full_add M1 (sum, c_out, a, b, cin); //DUT

initial #200 $finish; // Stopwatch

initial begin // Stimulus patterns
#10 a = 0; b = 0; cin = 0; // Statements execute in sequence
#10 a = 0; b = 1; cin = 0;
#10 a = 1; b = 0; cin = 0;
#10 a = 1; b = 1; cin = 0;
#10 a = 0; b = 0; cin = 1;
#10 a = 0; b = 1; cin = 1;
#10 a = 1; b = 0; cin = 1;
#10 a = 1; b = 1; cin = 1;
end
endmodule
```
Summary

Design module
- Gate-level or RT-level
- Real hardware
  - Instance of modules exist all the time
- Each module has architecture figure
  - Plot architecture figures before you write verilog codes

Test bench
- Feed input data and compare output values versus time
- Usually behavior level
- Not real hardware, just like C/C++
Note

- **Verilog is a platform**
  - Support hardware design (design module)
  - Also support C/C++ like coding (test bench)

- **How to write verilog well**
  - Know basic concepts and syntax
  - Get a good reference (a person or some code files)
  - Form a good coding habit
    - Naming rule, comments, format partition (assign or always block)

- **Hardware**
  - Combinational circuits (today’s topic)
    - 畫圖 (architecture), then 連連看 (coding)
  - Sequential circuits (we won’t model them in this course)
    - register: element to store data