Verilog® HDL Training Guide

教師：蘇慶龍
Outline

1. Introduction and Basic Concepts
2. Lexical Conventions and Data Types in Verilog
3. Support for Verification
4. Design Examples
5. Behavior Modeling
Outline

1. Introduction and Basic Concepts

2. Lexical Conventions and Data Types in Verilog

3. Support for Verification

4. Design Examples

5. Behavior Modeling
1. Introduction and Basic Concepts

Objectives

- Understand the basics of Hardware Description Language (HDL) and simulators.
- Understand the verilog® language and the verilog-XL™ software.
- Introduce verilog® structural and behavioral construct in a sample design.
- Starting the verilog-XL software.
Key Features of HDLs

- HDLs have high-level programming language constructs and constructs to describe the connectivity of the circuit.
- HDLs allow you to describe the design at various levels of abstractions.
- HDLs allow you to describe the functionality as well as the timing.
- Concurrency.
- Time.
Different Levels of Abstraction

- Architecture / Algorithm
- RTL
- Gate
- Switch
Verilog HDL and Verilog-XL

- Verilog® and Verilog-XL™ are CADENCE trademarks.
- Verilog HDL
  Hardware description language that allows you to describe circuits at different levels of abstractions and allow you to mix any level of abstraction in the design.
- Verilog-XL Software
  High speed event-driven simulator that reads Verilog HDL and simulates the description to emulate the behavior of real hardware.

National Chiao Tung University
Electronics Engineering
http://www.EE.NCTU.edu.tw

VLSI and Signal Processing Lab
http://soc.eecs.yuntech.edu.tw/
E-Mail:kevinsu@yuntech.edu.tw
Key Language Features

Verilog Module: basic building blocks

module SN74LS74
endmodule

module DFF
endmodule

module ALU
endmodule
module DFF (d, clk, clr, q, qb);
input d, clk, clr;
output q, qb;
endmodule
Key Language Features (continued)

Module Instance

```verilog
module REG4 (d, clk, clr, q, qb);

output [3:0] q, qb;
input [3:0] d;
input clk, clr;

DFF d0 (d[0], clk, clr, q[0], qb[0]);
DFF d1 (d[1], clk, clr, q[1], qb[1]);
DFF d2 (d[2], clk, clr, q[2], qb[2]);
DFF d3 (d[3], clk, clr, q[0], qb[3]);

endmodule
```
A Simple and Complete Example

Test Fixture Files

Device Under Test

Response

gr_waves
gr_regs
c_Waves

Input Vectors & Control

a
b

sel

a1

b1

out

a
b

VLSI and Signal Processing Lab

http://soc.eecs.yuntech.edu.tw/
E-Mail:kevinsu@yuntech.edu.tw
module MUX2_1 (out,a,b,sel);

output out;
input a,b,sel;

not (sel_,sel);
and (a1,a,sel_);
and (b1,b.sel);
or (out,a1,b1);
endmodule
module testfixture;
reg a, b, sel;
MUX2_1 mux (out, a, b, sel);
initial
begin
    a = 0; b = 1; sel = 0;
    #5 b = 0;
    #5 b = 1; sel = 1;
    #5 a = 1;
end
initial
    $monitor ($time, ,out, ,a, ,b, ,sel);
endmodule
Running and Results

% Verilog mux.v testfixture.v

Running

Results

VERILOG-XL 1.6a ........
* Copyright Cadence ..... 
* All Rights ................
* Condiditial and ..........
* property od ..............
Compiling source file “mux.v”
Compiling source file “testfixture.v”
Highest level modules :
testfixture
time out a b sel
  0 0 0 1 0
  5 0 0 0 0
  10 1 0 1 1
  15 1 1 1 1
28 simulation events + 12 accelerated events
CPU time : 0.4 secs to compile + 0.1 secs to ...
End of VERILOG-XL 1.6a 7.1 ........

http://soc.eecs.yuntech.edu.tw/
E-Mail: kevinsu@yuntech.edu.tw
Procedural Blocks

- initial procedural blocks, execute only once.
- initial always blocks, execute in a loop.
- all procedural blocks execute concurrently.
Starting the Verilog-XL Software

% verilog <command_line_options> <design_files>

<command_line_options>

<table>
<thead>
<tr>
<th>command line option</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>-f&lt;file_name&gt;</td>
<td>Read commands from the file &lt;file_name&gt;</td>
</tr>
<tr>
<td>-c</td>
<td>Only compile the description</td>
</tr>
<tr>
<td>-d</td>
<td>Decompile the Verilog description</td>
</tr>
</tbody>
</table>

<design_files> are files that contain Verilog descriptions.
Running and Results

% verilog –f run.f

fadder.v
testfadder.v

run.f
Outline

1. Introduction and Basic Concepts
2. Lexical Conventions and Data Types in Verilog
3. Support for Verification
4. Design Examples
5. Behavior Modeling
2. Lexical Conventions and Data Types in Verilog

Objectives

- Understand the lexical conventions used in the Verilog language.
- Learn to recognize special language tokens.
- Learn the various classes of data types in Verilog.
- Learn the Verilog logic value system.
- Learn the various Verilog strengths.
- Learn the declaration syntax.
module MUX2_1 (out,a,b,sel);
// Port declarations
output output;
input a,b,sel;
/*
   The netlist logic selections input “a” when sel=0 and it selects
   “b” when sel=1
*/
not (sel_,sel);
and (a1,a,sel_);
and (b1,b.sel);
or (out,a1,b1);
endmodule
**Integer and Real Numbers**

- Number can be integers or real numbers.
- Integers can be sized or unsized. Sized integers are represented as `<size>` `<base>` `<value>
  where `<size>` is the size in bits.
  `<base>` can be `b`(binary), `o`(octal), `d`(decimal) or `h`(hexadecimal)
  `<value>` is any legal number in the selected base and x, z, ?
- Real number can be represented in decimal or scientific format.

<table>
<thead>
<tr>
<th>Number</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>unsized decimal</td>
<td>6.3   - decimal notation</td>
</tr>
<tr>
<td>‘H83a</td>
<td>unsized hexadecimal</td>
<td>32e-4 - scientific notation for 0.0032</td>
</tr>
<tr>
<td>8’b1100_0001</td>
<td>8-bit binary</td>
<td>4.1E3 - scientific notation for 4100.0</td>
</tr>
<tr>
<td>64’hff01</td>
<td>64-bit hexadecimal</td>
<td></td>
</tr>
<tr>
<td>9’o17</td>
<td>9-bit octal</td>
<td></td>
</tr>
<tr>
<td>32’Bz</td>
<td>32-bit z (x and z automatically extended)</td>
<td></td>
</tr>
</tbody>
</table>
Strings

- Strings are enclosed in **double quotes** and must be specified on one line.
- Verilog recognizes normal C-escape characters.
  - \t = tab
  - \n = newline
  - \\ = backslash
  - \" = quote mark ( “ )
  - %%% = % sign
- A new line using a carriage return cannot be used in string.

" This is a normal string. “
" This is a string with a \t tab and a new line at the end\n “
" This is a formatted string : value = %b”
Identifiers are user-provided names for Verilog objects within a description.

Identifiers must begin with an alphabetical character (a~z, A~Z) or an underscore (_) and can contain any alphanumeric character, dollar signs ($), and the underscore.

Identifiers can be up to 1023 characters long.

Names of modules, ports and instances are identifiers.

Legal Identifiers:
- shift_reg_a
- busa_index
- _bus3

Illegal Identifiers:
- 34net  // does not begin with alphabet
- a*b   // contain a non alphanumeric
- n@238 // character
Case Sensitivity

- Verilog is a case-sensitive language.
- All keywords are lowercase.
- You can run Verilog in case-insensitive mode by specifying `-u` at the command line.
**Special Language Tokens**

- **System Tasks and Functions**
- `$<identifier>`
  - “$” sign denotes Verilog system tasks and functions
  - A number of system task and functions are available to perform different operations like
    - Finding the current simulation time ($time)
    - Displaying / monitoring the values of the signals ($display, monitor)
    - Stopping the simulation ($stop)
    - Finishing the simulation ($finish)

```verilog
$monitor ($time, "a=%b, b=%h", a, b);
```
Delay Specification

- `<delay specification>`

The "#" character denotes the delay specification for both gate instances and procedural statements.

```verilog
module testfixture;
reg a,b,sel;
MUX2_1 mux (out,a,b,sel);
initial
begin
  a=0; b=1; sel=0;
  #5 b=0;
  #5 b=1; sel=1;
  #5 a=1;
end
initial
$monitor ($time, ,out, ,a, ,b, ,sel);
endmodule
```
Compiler Directives

You indicate compiler directives with a grave accent (`).
Text Substitution

- The `define` compiler directive provides a sample text-substitution facility.

```
`define <name> <macro_text>


`define not_delay #1
......
.......

not `not_delay not1(sel_,sel);

endmodule
```
Text Inclusion

- Using the `include` compiler directive to insert the contents of an entire file.

```
#include "global.v"
#include "parts/count.v"
#include "../library/mux.v"
```

- Search directories for the file to be included can be specified using the `+incdir` command-line option.

```
+incdir+<directory1>+<directory2>+ ... +<directoryN>
```
Timescale in Verilog

- The `timescale` compiler directive declares the time unit and its precision.

```
timescale <time_unit> / <time_precision>
```

- The `timescale` compiler directive cannot appear inside a module boundary.

```
`timescale 1 ns / 100 ps
module MUX2_1 (out, a, b, sel);
    .......
    ...........
    ...........
endmodule
```
The smallest precision of all the timescale directives determines the time unit of the simulation.

```
`timescale 1 ns / 10 ps
module1 ( ... );
    ......
endmodule

........
........

`timescale 1 ps / 100 fs
moduleN ( ... );
    ......
endmodule
```

Simulation takes place in units of 100 fs
4-Value Logic System in Verilog

Zero, Low, False, Logic Low, Ground, VSS, Negative Assertion

One, High, True, Logic High, Power, VDD, VCC, Positive Assertion
4-Value Logic System in Verilog (continued)

- **X**, Unknown: Occurs at Logical Conflict Which Cannot be Resolved
- **HiZ**, High Impedance, Tri-Stated, Disabled Drived (Unknown)

Diagram:

- Symbol bufif1
- Symbol X
- Symbol Z
- Symbol 0
Major Data Type Classes

- Nets
- Registers
- Parameters
Nets are continuously driven by the devices that driven them.

Verilog automatically propagates a new value onto a net when the drivers on the net change value.
A register holds its value until a new value is assigned to it.

Register are used extensively in behavioral modeling and in applying stimuli.

Values are applied to registers using behavioral constructs.
### Types of Registers

The register class consists of four data types.

<table>
<thead>
<tr>
<th>Register Types</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg</td>
<td>Unsigned integer variable of varying bit width</td>
</tr>
<tr>
<td>integer</td>
<td>Signed integer variable, 32-bits wide. Arithmetic operations procedure 2’s complement results</td>
</tr>
<tr>
<td>real</td>
<td>Signed floating-point variable, double precision</td>
</tr>
<tr>
<td>time</td>
<td>Unsigned integer variable, 64-bits wide (verilog-XL stores simulation time as a 64-bit positive value)</td>
</tr>
</tbody>
</table>
Examples

- `reg a; // a scalar register`
- `reg [3:0] v; // a 4-bit vector register from msb to lsb`
- `wire [0:31] w1, w2; // Two 32-bit wires with msb=0`
- `reg [7:0] mema [0:255] // 256*8 memory (8 bits/word)`
- `mema[1]=0 // Assign 0 to the first element`
Choosing the Correct Data Type

Module Boundary

net/register → net → net/register

net

net/register → net → net/register
Use parameters to declare runtime constants.

You can use a parameter anywhere that you can use a literal.

```verilog
module mod ( out, in1, in2 ) ;
................
parameter p1=8 ,
    real_constant=2.039 ,
    x_word=16’bx ;

................
wire [ p1:0] w1
........
endmodule
```
Array of Instances

- A range following an instance name creates an array of instances. 
  `<name> <instance_name> <range> (<ports>);`

- The following two modules are equivalent:

```verilog
module driver (out, in, en);
    output [2:0] out;
    input  [2:0] in;
    input   en;
    buffif0 u2 (out[2], in[2], en);
    buffif0 u1 (out[1], in[1], en);
    buffif0 u0 (out[0], in[0], en);
endmodule
```

```verilog
module driver (out, in, en);
    output [2:0] out;
    input  [2:0] in;
    input   en;
    buffif0 u[2:0] (out, in, en);
endmodule
```
# Verilog Operators

<table>
<thead>
<tr>
<th>Type of Operators</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td>! ~ &amp;</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>* / % + -</td>
</tr>
<tr>
<td>Logical Shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>Relational</td>
<td>&gt; &lt; &gt;= &lt;=</td>
</tr>
<tr>
<td>Equality</td>
<td>== === != !==</td>
</tr>
<tr>
<td>Binary Bit-wise</td>
<td>&amp;</td>
</tr>
<tr>
<td>Binary Logical</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>Conditional</td>
<td>?:</td>
</tr>
</tbody>
</table>
Outline

1. Introduction and Basic Concepts
2. Lexical Conventions and Data Types in Verilog
3. Support for Verification
4. Design Examples
5. Behavior Modeling
3. Support for Verification

Objectives

- Understand textural and graphic outputs from Verilog.
- Understand different system function to read simulation time.
- Understand file I/O in Verilog.
- Understand dumping the recorded signal value (*.vcd) file
Support for Verification

- Verilog has system functions to read the current simulation
  - $time
  - $stime
  - $realtime

- Verilog has system tasks to support textual output
  - $display
  - $strobe
  - $write
  - $monitor

- Verilog has system tasks to support graphic output
  - $gr_waves
  - $gr_regs
  - cWaves
The $time, $realtime, $stime functions return the current simulating time.

$time returns time as a 64-bit integer.

$stime returns time as a 32-bit integer.

$realtime returns time as a real number.
Displaying Signal Values

- $\text{display}$ prints out the current values of the signals in the argument list.
- $\text{display}$ automatically prints a new line.
- $\text{display}$ supports different bases.

\[
\text{display} \left( \$time, \ "\%b \ \%h \ \%d \ \%o\" , \text{sig1, sig2, sig3, sig4} \right);
\]
$\text{write}$ is identical to $\text{display}$ except that it does not print a new line character.

$\text{strobe}$ is identical to $\text{display}$ except that the argument evaluation is delayed just prior to advance of the simulation time.
$\text{gr\_waves}$ displays the argument list in a graphic window.

Example:
$\text{gr\_waves ("data \%b", data, "clk", clk, "load", load, "cnt", cnt);}$

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>0100</td>
<td></td>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cnt</td>
<td>00 01 02</td>
<td></td>
<td>1d 1e</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Changing default setting

Default in Hex.
Verilog provides a set of system tasks to recode signal value changed in the standard VCD (value change dump) format for other tools (Debussy, SIMVISION, ...) to display all the signal waves.

```
$dumpfile("filename.vcd"); Open a VCD data base for recording
$dumpvars(); Select all signals for recording
```
module fadder_test;
    reg a,b,cin;
    wire sum,cout;
    ...
    #10 a=0;b=1;cin=0;
    ...
    end
    initial
    begin
    $dumpfile("filename.vcd");
    $dumpvars();
    end
endmodule
Outline

1. Introduction and Basic Concepts
2. Lexical Conventions and Data Types in Verilog
3. Support for Verification
4. Design Examples
5. Behavior Modeling
module fadder (sum, cout, a, b, cin);
    // port declaration
    output sum, cout;
    input   a, b, cin;

    // netlist declaration
    xor u0 (sum, a, b, cin);
    and u1 (net1, a, b);
    and u2 (net2, b, cin);
    and u3 (net3, cin, a);
    or    u4 (cout, net1, net2, net3);
endmodule
Behavior Level Verilog for a Full Adder

module fadder (sum, cout, a, b, cin);
  // port declaration
  output sum, cout;
  input   a, b, cin;
  reg     sum, cout;

  // behavior declaration
  always @ (a or b or cin)
    begin
      sum = a ^ b ^ cin;
      cout = (a & b) | (b & cin) | (cin & a);
    end
endmodule
Hierarchical Structure – Connection by Order List

module top (Out1, Out2, In1, In2);
  output Out1, Out2;
  input  In1, In2;
  comp c1(Out1, Out2, In1, In2)
endmodule

module comp (o1, o2, i1, i2)
  output  o1, o2;
  input    i1, i2;
  ...
endmodule

module top (Out1, In1, In2);
  output Out1;
  input  In1, In2;
  comp c1(Out1, In1, In2)
endmodule
Hierarchical Structure – Connection by Name

module top (Out1, Out2, In1, In2);
  output Out1, Out2;
  input  In1, In2;
  comp c1 (.i2(In2), .o1(Out1), .o2(Out2), .i1(In1));
endmodule

module comp (o1, o2, i1, i2)
  output  o1, o2;
  input    i1, i2;
  ...
endmodule
Outline

1. Introduction and Basic Concepts
2. Lexical Conventions and Data Types in Verilog
3. Support for Verification
4. Design Examples
5. Behavior Modeling
4. Behavior Modeling

Objectives

- Learn the basics of behavioral modeling.
- Learn the high-level programming language constructs in Verilog.
Behavioral modeling enables you to describe the system at a high level of abstraction.

Behavioral modeling in Verilog is described by specifying a set of concurrently active procedural blocks.

High-level programming language constructs are available in Verilog for behavioral modeling.
Behavioral Modeling

- Procedural blocks are the basis for behavioral modeling.
- Procedural blocks have the following components.

### Conditions of Execution

<table>
<thead>
<tr>
<th>initial</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>always</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>
Timing Control in Procedural Blocks

- **Simple Delay**
  
  ```plaintext
  #10 rega = regb;
  #(cycle/2) clk = ~clk; // cycle is declared as a parameter
  ```

- **Edge-Triggered Timing Control**
  
  ```plaintext
  @(r or q) rega = regb; // controlled by in “r” or “q”
  @(posedge clk) rega = regb; // controlled by positive edge
  @(negedge clk) rega = regb; // controlled by negative edge
  ```

- **Level-Triggered Timing Control**
  
  ```plaintext
  wait (!enable) rega = regb; // will wait until enable = 0
  ```
always wait (set)
begin @(posedge clk)
  #3 q=1;
  #10 q=0;
  wait (set);
end
### Sequential Block vs. Parallel Block

- **Sequential Block**: begin-end block
- **Parallel Block**: fork-join block

<table>
<thead>
<tr>
<th>always</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>always</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>initial</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>fork</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>initial</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>initial</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

VLSI and Signal Processing Lab

http://soc.eecs.yuntech.edu.tw/
E-Mail: kevinsu@yuntech.edu.tw
### Sequential Block vs. Parallel Block

<table>
<thead>
<tr>
<th>Sequential Block</th>
<th>Parallel Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>fork</td>
</tr>
<tr>
<td>regA = 8'h00;</td>
<td>regA = 8'h00;</td>
</tr>
<tr>
<td>#50 regA = 8'h35;</td>
<td>#50 regA = 8'h35;</td>
</tr>
<tr>
<td>#50 regA = 8'hFF;</td>
<td>#100 regA = 8'hFF;</td>
</tr>
<tr>
<td>#50 regA = 8'hE7;</td>
<td>#150 regA = 8'hE7;</td>
</tr>
<tr>
<td>end</td>
<td>join</td>
</tr>
<tr>
<td></td>
<td>regA = 8'h7;</td>
</tr>
<tr>
<td></td>
<td>regA = 8'hE7;</td>
</tr>
</tbody>
</table>

---

**VLSI and Signal Processing Lab**


E-Mail: kevinsu@yuntech.edu.tw
Blocking vs. Non-Blocking Assignment

- **Block**: “=”
- **Non-Block**: “<=”

//sequentially assign
a=0;
b=0;

//concurrently assign
a<=0;
b<=0;
Blocking vs. Non-Blocking Assignment

module swap;
    reg a, b, clk;
    initial
        begin
            a=0;
            b=1;
            clk=0;
        end
    always #5 clk=~clk;
    always @(posedge clk)
        begin
            a<=b;
            b<=a;
        end
endmodule
References

- CADENCE Verilog-XL User Guide
- CIC training course : Verilog-XL Training Manual