Verilog Hardware Description Language (Verilog HDL)

Edited by Chu Yu

http://ece.niu.edu.tw/~chu/

(2007/2/26)
Verilog HDL

- Brief history of Verilog HDL
  - 1985: Verilog language and related simulator Verilog-XL were developed by Gateway Automation.
  - 1990: Open Verilog International formed.

- Features of Verilog HDL
  - Ability to mix different levels of abstract freely.
  - One language for all aspects of design, testing, and verification.
Verilog HDL

- **HDL – Hardware Description Language**
  - A programming language that can describe the functionality and timing of the hardware.

- **Why use an HDL?**
  - It is becoming very difficult to design directly on hardware.
  - It is easier and cheaper to different design options.
  - Reduce time and cost.
Programming Language V.S. Verilog HDL

- **Programming Language**

  ```c
  if (a>b)
  {
  ...
  }
  ```

  (C code)

  ```asm
  sub a, b
  ...
  ...
  ```

  (asm code)

  - **Compiler**
  - **Assembler**

  Computer (CPU)

- **Verilog HDL**

  ```verilog
  if (a>b)
  begin
  ...
  end
  ```

  (Verilog code)

  - **Synthesizer**

  (logic circuit)

  (cell library)
Verilog HDL

- Verilog-XL is an *event-driven* simulator that can emulate the hardware described by Verilog HDL.

- Verilog-HDL allows you to describe the design at various levels of abstractions within a design.
  - Behavioral Level
  - RTL Level
  - Gate Level
  - Switch Level
Time Wheel in Event-Driven Simulation

Event queues at each time stamp

An event $E_t$ at time $t$
Schedules another event at time $t + 2$

- Time advances only when every event scheduled at that time is executed.
Different Levels of Abstraction

- **Architecture / Algorithmic (Behavior)**
  - A model that implements a design algorithm in high-level language construct.
  - A behavioral representation describes how a particular design should respond to a given set of inputs.

- **Register Transfer Logic (RTL)**
  - A model that describes the flow of data between registers and how a design processes these data.

- **Gate Level (Structure)**
  - A model that describes the logic gates and the interconnections between them.

- **Switch Level**
  - A model that describes the transistors and the interconnections between them.
Three Levels of Verilog-HDL

- Behavioral Level (RTL)
  
  \[
  \text{assign } \{\text{Co, Sum}\} = A + B + Ci
  \]

- Gate Level
  
  \[
  \text{xor } u0(.z(\text{hs}), .a1(A), .a2(B)); \\
  \text{xor } u1(.z(\text{Sum}), .a1(Ci), .a2(\text{hs})); \\
  \text{and } u2(.z(\text{hc0}), .a1(A), .a2(B)); \\
  \text{and } u3(.z(\text{hc1}), .a1(Ci), .a2(\text{hs})); \\
  \text{or } u4(.z(\text{Co}), .a1(\text{hc0}), .a2(\text{hc1}));
  \]

- Switch Level

  // AND gate of u2

  \[
  \text{pmos } p0(\text{VDD, nand, A}), \\
  \text{p1(\text{VDD, nand, B});} \\
  \text{nmos } n0(\text{nand, wire1, A}), \\
  \text{n1(wire1, GND, B);} \\
  \text{pmos } p2(\text{VDD, hc0, nand);} \\
  \text{nmos } n2(\text{hc0, GND, nand);} \\
  \]

  \[\vdots\]
Top-Down Design Flow in ASIC

- Functionality
- performance
- data flow.

- Partitioning
- model the submodule at behavior level
- re-simulation

- re-model the submodule using
  library component
- re-simulation

- Pre-designed and tested
  library component
Top Down ASIC Design Flow

Idea and Specification

Behavioral Modeling

Behavior Model
*(Verilog HDL or C language)*

Verification
*(Verilog-XL)*

Partitioning and Re-modeling

Logic Blocks with Function Definition

RTL Modeling

RTL Model
*(Verilog HDL)*

Verification
*(Verilog-XL)*

...sum = a + b;
...

...always @(a or b or c)
{carry, sum} = a+b+c;
...
Top Down ASIC Design Flow (Con’t)

- RTL Model *(Verilog HDL)*
  
- Logic Synthesis *(Synopsys)*
  
- Gate Level Netlist *(Verilog HDL)*
  
- Physical Design *(CELL3 Ensemble)*
  
- Verification *(Verilog-XL)*
  
- GDS II

... xo03d1 u0(sum,a,b,c); an02d1 u1(g2,a,b);
...
Verilog-HDL Simulators

- **VCS (Synopsys)**
  - Platform
    - Windows NT/XP, SUN Solaris (UNIX), Linux.

- **Modelsim (Mentor)**
  - Platform
    - Windows NT/XP, SUN Solaris (UNIX), Linux.

- **NC-Verilog (Cadence)**
  - Platform
    - Windows NT/XP, SUN Solaris (UNIX), Linux.

- **Verilog-XL (Cadence)**
  - Platform
    - SUN Solaris (UNIX).

- **Other Simulators**
  - MAX+PLUS II, Quartus II (Altera)
  - Active HDL (Aldec), Silos (Silvaco), …
Overview of Verilog Module

- A Verilog module includes the following parts:

  module module_name (port_name);
  port declaration
  data type declaration
  Task & function declaration
  module functionality or declaration
  timing specification
  endmodule
Example of Adder

- A Full Adder

```verilog
module adder (carry, sum, a, b, cin);
output carry, sum;
input a, b, cin;
wire w0, w1, w2;
xor u0(sum, a, b, cin);
and    u1(w0, a, b);
and    u2(w1, b, cin);
and    u3(w2, cin, b);
or      u4(carry, w0, w1, w2);
endmodule
```

---

**module name**

- `module adder (carry, sum, a, b, cin);`

**I/O pins**

- `output carry, sum;`
- `input a, b, cin;`
- `wire w0, w1, w2;`

**I/O pin declaration**

- `xor u0(sum, a, b, cin);`
- `and u1(w0, a, b);`
- `and u2(w1, b, cin);`
- `and u3(w2, cin, b);`
- `or u4(carry, w0, w1, w2)`

**logic circuit description**

- `build-in module invoked`
- `instantiation`

---

**Example Diagram**

```
    a       b      cin
   -----> u0 -------- u4
            |        |
             w0      w2
             |        |
        u1 -------- u3
            |        |
             w1      w1
```

**Verilog HDL**

Edited by Chu Yu
Three Levels of Abstraction

● A Full Adder

```verilog
module adder (carry, sum, a, b, cin);
output carry, sum;
input a, b, cin;
reg sum, carry;

always @(a or b or cin)
{carry, sum} = a + b + cin;
endmodule //behavioral level

module adder (carry, sum, a, b, cin);
output carry, sum;
input a, b, cin;
wire w0, w1, w2;
xor u0(sum, a, b, cin);
and u1(w0, a, b);
and u2(w1, b, cin);
and u3(w2, cin, b);
or u4(carry, w0, w1, w2);
endmodule //gate level
```

```verilog
module adder (carry, sum, a, b, cin);
output carry, sum;
input a, b, cin;
assign {carry, sum} = a + b + cin;
endmodule //RTL level
```
Identifiers of Verilog

- Identifiers are user-provided name for Verilog objects within a description.
- Legal characters in identifiers:
  - a-z, A-Z, 0-9, _, $
- The first character of an identifier must be an alphabetical character (a-z, A-Z) or an underscore (_).
- Identifiers can be up to 1024 characters long.

Example:

- Mux_2_1
- abc123
- ABC123
- Sel_
- A$b$10
Escaped Identifiers

- Escaped Identifiers start with a backslash (\) and end with a white space.
- They can contain any printable ASCII characters.
- Backslash and white space are not part of the identifiers.

Example:

```verilog
module \2:1mux(out, a, b, sel);

not u0(\~out, in);
```
Case Sensitivity

- Verilog is a case-sensitive language.
- You can run Verilog in case-insensitive mode by specifying \texttt{--u} command line option.

\textbf{Example:}

\begin{verbatim}
module inv(out, in); module Inv(out, in);
...
endmodule
\end{verbatim}

\begin{verbatim}
module Inv(out, in);
...
endmodule
\end{verbatim}

// Both \texttt{inv} and \texttt{Inv} are viewed as two different modules.
Verilog-HDL Structural Language

- **Verilog Module**
  - Modules are basic building blocks in hierarchy.
  - Every module description starts with module name(output_ports, input_ports), and ends with endmodule.

- **Module Ports**
  - Module ports are equivalent to the pins in hardware.
  - Declare ports to be *input, output, or inout* (bidirectional) in the module description.
Nets and Registers

- **Nets**: nets are continuously driven by the devices that drive them.
  - wire, wor, wand, ...
  - example: wire [7:0] w1, w2;
  - wire [0:7] w1;
  - if wire is not vector type, then it doesn’t need to declaration.

- **Registers**: registers are used extensively in behavioral modeling and in applying stimulus.
  - reg
  - example: reg [3:0] variable;
Registers

● More Examples

    reg mem1[127:0];  //128-bit memory with 1-bit wide
    reg mem2[63:0];
    reg [7:0] mem3[127:0];  //128-bit memory with 8-bit wide

    mem2=0;  // illegal syntax
    mem2[5] = mem1[125];
    mem2[10:8] = mem1[120:118];
    mem3[11]=0;  //8-bit zero value
Other Types of Nets

- Various net types are available for modeling design-specific and technology-specific functionality.

<table>
<thead>
<tr>
<th>Net Types</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>wire, tri</td>
<td>For multiple drivers that are Wired-OR</td>
</tr>
<tr>
<td>wand, triand</td>
<td>For multiple drivers that are Wired-AND</td>
</tr>
<tr>
<td>trireg</td>
<td>For nets with capacitive storage</td>
</tr>
<tr>
<td>tri1</td>
<td>For nets with weak pull up device</td>
</tr>
<tr>
<td>tri0</td>
<td>For nets with weak pull down device</td>
</tr>
<tr>
<td>supply1</td>
<td>Power net</td>
</tr>
<tr>
<td>supply0</td>
<td>Ground net</td>
</tr>
</tbody>
</table>
Example of Nets

• Example I

\[\text{tri } z ;\]
\[
\begin{array}{c}
\text{i} \\
\text{sel}
\end{array} \rightarrow \\
\begin{array}{c}
\text{i} \\
\text{sel}
\end{array} \\
\begin{array}{c}
\text{z} \\
\text{z}
\end{array}
\]

- if \(en = 1\), \(z = i\);
- if \(en = 0\), \(z = \text{high impedance}\);

\[\text{trireg } z ;\]
\[
\begin{array}{c}
\text{i} \\
\text{sel}
\end{array} \rightarrow \\
\begin{array}{c}
\text{i} \\
\text{sel}
\end{array} \\
\begin{array}{c}
\text{z} \\
\text{z}
\end{array}
\]

- if \(en = 1\), \(z = i\);
- if \(en = 0\), \(z = \text{stores its last value}\);

\[\text{tri0 } z ;\]
\[
\begin{array}{c}
\text{i} \\
\text{sel}
\end{array} \rightarrow \\
\begin{array}{c}
\text{i} \\
\text{sel}
\end{array} \\
\begin{array}{c}
\text{z} \\
\text{z}
\end{array}
\]

- if \(en = 1\), \(z = i\);
- if \(en = 0\), \(z = 0\);

\[\text{tril } z ;\]
\[
\begin{array}{c}
\text{i} \\
\text{sel}
\end{array} \rightarrow \\
\begin{array}{c}
\text{i} \\
\text{sel}
\end{array} \\
\begin{array}{c}
\text{z} \\
\text{z}
\end{array}
\]

- if \(en = 1\), \(z = i\);
- if \(en = 0\), \(z = 1\);
Example of Nets

- Example II

- TTL

Any floating input in a TTL logic gate acts like a logical 1 applied at that input.

An open collector TTL output operates wired AND function when two or more gates are wired together.
## True Tables for tri, triand, and trior Nets

<table>
<thead>
<tr>
<th>wire/tri</th>
<th>0</th>
<th>1</th>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>wand/triand</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>1</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>1</td>
<td>x</td>
<td>z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>wor/trior</th>
<th>0</th>
<th>1</th>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>z</td>
</tr>
</tbody>
</table>
# Logic Level Modeling

## Built-in primitive functions

<table>
<thead>
<tr>
<th>Gates</th>
<th>MOS Switches and Bidirectional Transistors</th>
<th>Nets</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>buf</td>
<td>wire</td>
</tr>
<tr>
<td>nand</td>
<td>bufif0</td>
<td>supply0</td>
</tr>
<tr>
<td>nor</td>
<td>bufif1</td>
<td>wand</td>
</tr>
<tr>
<td>or</td>
<td>notif0</td>
<td>wor</td>
</tr>
<tr>
<td>xor</td>
<td>notif1</td>
<td>tri</td>
</tr>
<tr>
<td>xnor</td>
<td>pullup</td>
<td>triand</td>
</tr>
<tr>
<td>not</td>
<td>pulldown</td>
<td>trior</td>
</tr>
</tbody>
</table>

- bufif1 (z, i, cn)
- bufif0 (z, i, cn)
- notif1 (zn, i, cn)
- notif0 (zn, i, cn)
Switch Level Modeling

nMOS (unidirectional)

pMOS (unidirectional)

cMOS (unidirectional)

nmos(out, in, ctl);

pmos(out, in, ctl);

cmos(out, in, nctl, pctl);
Operators Used in Verilog (Cont.)

- **Verilog Language Operators**

<table>
<thead>
<tr>
<th>Operator Type</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 Used in Verilog (Cont.)

**Precedence Rules for Operators**

<table>
<thead>
<tr>
<th>Operator Precedence Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>! ~</td>
</tr>
<tr>
<td>* / %</td>
</tr>
<tr>
<td>+ -</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt; &gt;=</td>
</tr>
<tr>
<td>== != === !==</td>
</tr>
<tr>
<td>&amp;</td>
</tr>
<tr>
<td>^ ^~</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>?: (ternary operator)</td>
</tr>
</tbody>
</table>

highest precedence

lowest precedence
Operators Used in Verilog (Cont.)

- The Relational Operators Defined

<table>
<thead>
<tr>
<th>Relational Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b</td>
</tr>
<tr>
<td>a &gt; b</td>
</tr>
<tr>
<td>a &lt;= b</td>
</tr>
<tr>
<td>a &gt;= b</td>
</tr>
</tbody>
</table>

- The Equality Operators Defined

<table>
<thead>
<tr>
<th>Equality Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>a === b</td>
</tr>
<tr>
<td>a !== b</td>
</tr>
<tr>
<td>a == b</td>
</tr>
<tr>
<td>a != b</td>
</tr>
</tbody>
</table>
Equality and Identity Operators

- equality operator

<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>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>z</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

- identity operator

<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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

```verilog
data_type a = 'b0x;
data_type b = 'b0x;
if(a == b) begin
    $display("a is equal to b");
    result = a is not equal to b;
end else begin
    $display("a is not equal to b");
    result = a is not equal to b;
end
```

```verilog
da = 'b0x;
b = 'b0x;
if(a == b) begin
    $display("a is identity to b");
    result = a is not identity to b;
end else begin
    $display("a is not identity to b");
    result = a is not identity to b;
end
```
- **Operators**
  - **Unary Operator**
    ```verilog
global assign a = ~b;
```
  - **Binary Operator**
    ```verilog
global assign a = b & c;
```
  - **Ternary Operator**
    ```verilog
global assign out = sel ? a : b; // 2-to-1 multiplexer
```

- **Comments**
  - **One Line Comment**
    ```verilog
    // this is an example of one line comment
    ```
  - **Multiple Line Comment**
    ```verilog
    /* this is an example of multiple line comment */
    ```
  - **Error Comment Remarks**
    ```verilog
    /* Error comment remark */ */
    ```
Operators Used in Verilog

● **Index:**
  - example: a[11:6], b[2], ...

● **Concatenation:** `{n{<exp> <, <exp>>*}}`
  ```
  adder4 a1(sum,carry,{a[2],a[2:0]},b[3:0]);
  assign {carry, sum} = a+b+ci;
  sign = {4{in[3]}, in};
  temp = 2’b01;
  out = {2{2’b10}, 2’b11, temp};  //out=8’b1010_1101
  ```

● **Arithmetic operation:** +, -, *
  - example: a=b+c;
    ```
    x=y*z;
    ```

● **Condition:** = =, !=, >, <, >=, <=, ...
  - example: assign b = (a = = 0) ;
Literal Numbers

- Literal integers are interpreted as decimal numbers in the machine word size (32 bits) by default.
- Size and base may be explicitly specified

```
<size>'<base><value>
```

- `<size>`: size in bits as a decimal number.
- `<base>`: b(binary), o(octal), h(hex), d(decimal).
- `<value>`: 0-9, a-f, x, z, ? (must be legal number in `<base>`)

- Four types of logic value
  - 0 (logical 0), 1 (logical 1), x (unknown), z (high impedance)
Literal Numbers (cont.)

**Examples**

- 12  
  32-bit decimal
- 8'd45  
  8-bit decimal
- 10'hF1  
  10-bit hex (left-extended with zero)
- 1'B1  
  1-bit binary
- 32'bz  
  32-bit Z
- 6'b001_010  
  6-bit binary with underscore for readability.

Underscores are ignored.
X and Z values are automatically extended.
A question mark ? in <value> is interpreted as a Z.
Block Statement

- **Block statement are used to group two or more statements together.**

- **Two Types of Blocks**
  - Sequential Block
    - Enclosed by keyword `begin` and `end`.
  - Parallel Block
    - Enclosed by keyword `fork` and `join`.

<table>
<thead>
<tr>
<th>initial</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>end</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>join</td>
<td></td>
</tr>
</tbody>
</table>

<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>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>join</td>
<td></td>
</tr>
</tbody>
</table>
Procedural Timing Controls

- **Three Types of Timing Controls**
  - `#<delay>`: Simple delay.
  - `@(.sig)`: Event control with edge-trigger and level-sensitive controls.
  - `wait(expr)`: Level-sensitive control.

- **Edge-Trigger Control**
  - `posedge`: positive edge. EX: always `@(posedge clk)`
  - `negedge`: negative edge. EX: always `@(negedge clk)`

- **Examples**

```verilog
always @(posedge clk)
begin
  #5 q=d;
  #1 qb=~d;
end
```

![Timing Diagram](image)
Procedural Timing Controls

**Examples**

```
initial
begin
  read=0;
  wait(en1|en2) read=1;
  #5 read=0;
end
```

```
always wait(set)
begin
  @(negedge clk);
  #3 q=1;
  #1 q=0;
  wait(!set);
end
```
Syntax of Verilog

**C-like structural language**
- statement: `begin ... end`, `if ... else...`, and so on.
- free writing format: statement ended with `;`
- remark: between `/* */` or `//` until the line feed.
- hierarchical modules.
High-level Programming Language Constructs

**Looping Controls**

- **forever** loop
  ```verilog
testbench:
  forever #100 clk=~clk;
  always #100 clk=~clk;
  ```

- **repeat** loop
  ```verilog
repeat_example:
  repeat(mem_depth)
  begin
    mem[address]=0;
    address=address+1;
  end
  ```

- **while** loop
  ```verilog
while_example:
  while(val[index]==1'b0)
    index=index-1;
  ```

- **for** loop
  ```verilog
for_example:
  for(index=0;index<size;
    index=index+1)
  if(val[index]==1'bx)
    $display("found an x");
  ```
High-Level Programming Language Constructs

- **Decision-making controls**

  **if statement**
  
  ```verilog
  example
  if (set == 1) out = 1;
  if (clear == 0) q = 0;
  else q = d;
  ```

  **case statement**
  
  ```verilog
  example
  case(instruction)
  2'b00: out = a + b;
  2'b01: out = a - b;
  default: out=0;
  endcase
  ```
### Continuous Assignment

- Continuous assignment provide a means to abstractly model combinational hardware driving values onto nets. An alternate version of the 1-bit full adder is shown below:

```verilog
module FA(Cout, Sum, a, b, Cin);
output Cout, Sum;
input a, b, Cin;
assign Sum = a ^ b ^ Cin,
    Cout = (a & b) | (b & Cin) | (a & Cin);
endmodule
```

- Logic loop of Continuous Assignment

```verilog
assign a = b+a;
```

\[ b \rightarrow \oplus \rightarrow a \]
Procedural Assignments

- Assignments made within procedural blocks are known as procedural assignments.
- The left-hand side of procedural assignment must be a data type in the `register` class.

Example

```
initial
begin
  out=0;
  #10 en1=~net23;
  #5 set=(r1|en1)&net4;
end
```
Intra-Assignment Timing Control

- Previously described timing control.
  
  #100 clk = ~ clk;
  @(posedge clock) q = d;

- Intra-assignment timing control.
  
  clk = #100 ~ clk;
  q = @(posedge clock) d;

- Simulators perform two steps when encounter an intra assignment timing control statement.
  - Evaluate the RHS immediately.
  - Execute the assignment after a proper delay.
Intra-Assignment Timing Control

- Intra-assignment timing control can be accomplished by using the following constructs:

<table>
<thead>
<tr>
<th>With intra-assignment construct</th>
<th>With intra-assignment construct</th>
</tr>
</thead>
</table>
| \(a = \#10 \ b;\)               | \begin{align*}
begnin \\
temp &= b; \\
\#10 \ a &= temp; \\
end \end{align*} |
| \(a = @\text{(posedge clk)} \ b;\) | \begin{align*}
begnin \\
temp &= b; \\
@ \text{(posedge clk)} \ a &= temp; \\
end \end{align*} |
| \(a = \text{repeat}\,(3)@\text{(posedge clk)} \ b;\) | \begin{align*}
begnin \\
temp &= b; \\
@ \text{(posedge clk)} \\
@ \text{(posedge clk)} \\
@ \text{(posedge clk)} \ a &= temp; \\
end \end{align*} |
Non-Blocking Procedural Assignment

- **Blocking procedural assignment.**
  
  ```
  rega = #100 regb;
  rega = @(posedge clk) regb;
  ```

- **Non-Blocking procedural assignment.**
  
  ```
  rega <= #100 regb;
  rega <= @(posedge clk) regb;
  ```

- **Schedule the assignment without blocking the procedural flow.**

- **Simulators perform two steps when encounter an non-blocking procedural assignment statement.**
  - Evaluate the RHS immediately.
  - Schedule the assignment at a proper time.
Blocking Procedural Assignment

initial begin
  a = #10 1;
  $display(“current time = %t a = %b”, $time, a);
  → evaluate at time = 10, a = 1
end
Non-Blocking Procedural Assignment

initialize begin
    a <= #10 1;
    $display("current time = %t a = %b ", $time, a);
    
end

$display(…)
Evaluates RHS (RHS = 1)
event queues at each time stamp
assign logic 1 to a

Evaluate RHS (RHS = 1)

Current Simulation Time

$t = 0$
$t = 1$
$t = 10$
$t = 11$
Hierarchical Design

Top-Down Design Methodology

module CPA4b(Cout, Sum, a, b, Cin);
output Cout;
output [3:0] Sum;
input [3:0] a, b;
inout Cin;
wire [2:0] c;
adder fa0(c[0], Sum[0], a[0], b[0], Cin);  //by position mapping
adder fa1(.a(a[1]), .b(b[1]), .cin(c[0]), .carry(c[1]), .sum(Sum[1]));  //by name mapping
adder fa2(c[2], Sum[2], a[2], b[2], c[1]);
adder fa3(Cout, Sum[3], a[3], b[3], c[2]);
endmodule

module adder (carry, sum, a, b, cin);
output carry, sum;
inout a, b, cin;
    assign {carry, sum} = a + b + cin;
endmodule
Delay Specification in Primitives

Delay specification defines the propagation delay of that primitive gate.

not #10 u0(out, in);
Delay Specification in Primitives

- Support (rise, fall, turn-off) delay specification.

```verilog
and #(3, 2) (out, in1, in2);

bufif1 #(3, 4, 7) (out, in, ctrl);
```
Delay and Time Scales

- **Delays**
  - **Gate Description**
    
    ```
    buf #<delay> buf0(X,A);
    
    where <delay> is:
    
    <delay time> or
    
    (<minimum delay>:<typical delay>:<maximum delay>)
    ```

    **example:**
    ```
    buf #(3:4:5) buf0(X,A);
    or #1 u0(out, in0, in1);
    ```

  - **Modeling Separate Rise and Fall Delays**
    
    ```
    not #<delay> not0(X,A);
    ```

    where <delay> is

    ```
    (<rise delay>,<fall delay>)
    ```

    **example:**
    ```
    not #(2.23:2.33:2.66,3.33:3.47:3.9) not0(X,A);
    ```
Delay and Time Scales (cont.)

- Three-state drivers: include rise, fall, and **turn off** delays

  *example*: bufif1 #(2.2:2.3:2.6, 3.3:3.4:3.9, 0.2:0.2:0.2) u0(out, in);

- **Timescales**

  The `timescale` compiler directive can be used to specify delays in explicit time units.
  
  Syntax of the `timescale` compiler directive:
  
  ```
  `timescale <unit>/<precision>
  ```

  *example*: `timescale 1ns/10ps
  
  then the design will be simulated in units of 10 ps.

  *example*: `not #(2.337,3.472) not1(X, A); 2.337ns will be scaled to 2.34 ten pico-second units for simulation purposes.`
Delay and Time Scales (cont.)

- The smallest precision of all the `timescale determines the time unit of simulation.

```
`timescale 1ns/10ps
module m1(...);
...
`timescale 100ns/1ns
module m2(...);
...
`timescale 1ps/1ps
module m3(...);
...
```
Compiler Directives

- All Verilog compiler directives are preceded by the accent sign (`).
- Compiler directives remain active until they are overridden or deactivated.
- The `resetall compiler directive resets all the compiler directives to their default values (only if there is a default value).

Example:

`define s0 2'b00
`include “lib.v”
`timescale 10ns/100ps
Compiler Directives

- The `define compiler directive provides a simple text-substitution facility.
  
  **Syntax:** `define <macro_name> <text_string>
  
  `<text_string>` will substitute `<macro_name>` at compile time.

- Typically use `define` to make the description more readable.

**Example:**

```verilog
`define false 0
...
assign a = false ;   ==> assign a = 0;
```
Compiler Directives

- Use `include` compiler directive to insert the contents of an entire file.
  
  `include “lib.v”`

  `include “dir/lib.v”`

- You can use `include` to
  
  - include global or common definitions.
  - include tasks without encapsulating repeated code within module boundaries.
Parameters

- Parameters are constants rather than variables.
- Typically parameters are used to specify delays and width of variable.

**Example:**

```verilog
module varmux(out, I0, I1, sel);
parameter width=2, delay=1;
output [width-1:0] out;
input [width-1:0] I0, I1;
input sel;

assign #delay out = sel? I1:I0;

endmodule
```
Overriding the Value of Parameters

Use *defparam* and hierarchical name of that parameter to change the value of parameter.

**Example:**

```verilog
module top;
  ...
  ...
  varmux u0(out0, a0, a1, sel);
  varmux u1(out1, a2, a3, sel);
  defparam u1.width=4, u1.delay=2;
  ...
endmodule
```

```text
Top.u0.width=2
Top.u0.delay=1
Top.u1.width=4
Top.u1.delay=2
```
System Tasks and Functions

- **$<identifier>**
  - Sign ‘$’ denotes Verilog system tasks and functions.
  - A number of system tasks and functions are available to perform different operations such as
    - Finding the current simulation time ($time).
    - Displaying/monitoring the values of signals ($display, $monitor).
    - Stopping the simulation ($stop).
    - Finishing the simulation ($finish).

- **For more information on system tasks and functions, see the module “Support for Verification”**.
Examples of Verilog-HDL

● 2-to-1 Multiplexer

module mux(out, I0, I1, sel);
  output [3:0] out;
  input [3:0] I0, I1;
  input sel;
  reg [3:0] out;
  
  always @(I0 or I1 or sel)
    if (sel==1’b1) out=I1;
    else out=I0;
endmodule

//The output value will be changed when one of I0, I1, and sel input signals is changing.
Example of Verilog-HDL

● **2-to-1 Multiplexer**

```verilog
module mux(out, I0, I1, sel);
    output [3:0] out;
    input [3:0] I0, I1;
    input sel;
    assign out = sel?I1:I0;
endmodule
```

![Diagram of 2-to-1 Multiplexer]
Example of Verilog-HDL

4-to-1 Multiplexer

module mux(out, I3, I2, I1, I0, sel);
output [3:0] out;
input [3:0] I3, I2, I1, I0;
input [1:0] sel;
reg [3:0] out;

always @ (I3 or I2 or I1 or I0 or sel)
case (sel)
   2’b00: out=I0;
   2’b01: out=I1;
   2’b10: out=I2;
   2’b11: out=I3;
   default: out=4’bx;
endcase
endmodule
Example of Verilog-HDL

- **4-to-1 Multiplexer**

```verilog
module mux(out, I3, I2, I1, I0, sel);
    output [3:0] out;
    input [3:0] I3, I2, I1, I0;
    input [1:0] sel;
    assign out = (sel == 2'b00)?I0:
                   (sel == 2'b01)?I1:
                   (sel == 2'b10)?I2:
                   (sel == 2'b11)?I3:
                   4'bx;
endmodule
```
Example of Verilog-HDL

● 3-to-1 Multiplexer

module mux(out, I3, I1, I0, sel);
output [3:0] out;
input [3:0] I3, I1, I0;
input [1:0] sel;
reg [3:0] out;

always @(I3 or I1 or I0 or sel)
  case (sel) //synopsys full_case
    2’b00: out=I0;
    2’b01: out=I1;
    2’b11: out=I3;
  endcase
endmodule // Require latches to synthesis the priority encoder circuit
if the remark ‘synopsys full_case’ is not assigned.
Example of Verilog-HDL

**Compound Logic**

```verilog
module complogic1(F, x, y, z);
    output F;
    input x, y, z;
    assign F = (x&~y)|(y|z);
endmodule

module complogic2(F, x, y, z);
    output F;
    input x, y, z;
    not u1(yp, y);
    and u2(xy, x, yp);
    or  u3(yz, y, z);
    or  u4(F, xy, yz);
endmodule
```

![Diagram of complogic1 and complogic2 circuits]
Example of Verilog-HDL

● casex I

```verilog
define casex I

module top(R1, R2, R3, I2, I1, I0);
  output R1, R2, R3;
  input I2, I1, I0;
  reg R1, R2, R3;

  always @(I2 or I1 or I0)
    casex ({I2, I1, I0})
      3'b1???: {R1, R2, R3}=3'b100;
      3'b?1?: {R1, R2, R3}=3'b010;  // ? = {0, 1}
      3'b??1: {R1, R2, R3}=3'b001;  // ? != {x, z}
  endcase
endmodule
```
Example of Verilog-HDL

- **casex II**

```verilog
module top(R1, R2, R3, I2, I1, I0);
output  R1, R2, R3;
input   I2, I1, I0;
reg     R1, R2, R3;

always @(I2 or I1 or I0)
  casex (1'b1)
    I0: {R1, R2, R3}=3'b100;
    I1: {R1, R2, R3}=3'b010;
    I2: {R1, R2, R3}=3'b001;
  endcase
endmodule
```
Example of Verilog-HDL

- **casez**

```verilog
casez (I2, I1, I0)
  3'b1??: {R1, R2, R3}=3'b100;
  3'b?1?: {R1, R2, R3}=3'b010;
  3'b??1: {R1, R2, R3}=3'b001;
endcase
```

```verilog
download içine
```
Example of Verilog-HDL

 Comparator

```verilog
module comparator(large, equal, less, a, b);
output large, equal, less;
input [3:0] a, b;
assign large = (a > b);
assign equal = (a == b);
assign less = (a < b);
endmodule
```
Hierarchical Modules

- **4-bit carry-propagate adder**

```verilog
module CPA(Cout, Sum, a, b, Cin);
output Cout;
output [3:0] Sum;
input [3:0] a, b;
input Cin;
wire [2:0] c;
adder fa0(c[0], Sum[0], a[0], b[0], Cin);
adder fa1(c[1], Sum[1], a[1], b[1], c[0]);
adder fa2(c[2], Sum[2], a[2], b[2], c[1]);
adder fa3(Cout, Sum[3], a[3], b[3], c[2]);
endmodule

module adder (carry, sum, a, b, cin);
output carry, sum;
input [3:0] a, b;
input cin;
assign {carry, sum} = a + b + cin;
endmodule
```

![Diagram of a 4-bit carry-propagate adder](image-url)
### Example of Verilog-HDL

#### Resource Sharing

<table>
<thead>
<tr>
<th>module top1(out, a, b, c, sel);</th>
<th>module top2(out, a, b, c, sel);</th>
</tr>
</thead>
<tbody>
<tr>
<td>output [4:0] out;</td>
<td>output [4:0] out;</td>
</tr>
<tr>
<td>input [3:0] a, b, c;</td>
<td>input [3:0] a, b, c;</td>
</tr>
<tr>
<td>input sel;</td>
<td>input sel;</td>
</tr>
<tr>
<td>reg [4:0] out;</td>
<td>reg [4:0] out;</td>
</tr>
<tr>
<td>always @ (a or b or c or sel)</td>
<td>always @ (a or b or c or sel)</td>
</tr>
<tr>
<td>if (sel) out=a+b;</td>
<td>if (sel) out=a+b;</td>
</tr>
<tr>
<td>else out=a-c;</td>
<td>else out=a+c;</td>
</tr>
<tr>
<td>endmodule //can’t be shared</td>
<td>endmodule //can be shared</td>
</tr>
</tbody>
</table>
Example of Verilog-HDL

- 1-bit latch

```verilog
module Latch(q, d, rst, enable);
    output q;
    input d, enable, rst;
    assign q=(rst==0)?0:
               (enable==1)?d:q;
endmodule
```

![1-bit latch diagram]
Example of Verilog-HDL

- **1-bit register with a synchronous reset**

```verilog
module D_FF(q, d, clk, load, rst);
  output q;
  input d, clk, load, rst;
  reg q;
  always @(posedge clk)
    if (rst == 1'b0) q = 0;
    else if (load == 1'b1) q = d;
endmodule
```

![Diagram of DFF](image-url)
Example of Verilog-HDL

- 1-bit register with a asynchronous reset

```verilog
module D_FF(q, d, clk, load, rst);
  output q;
  input  d, clk, load, rst;
  reg    q;
  always @(posedge clk or negedge rst)
    if (rst==1’b0) q=0;
    else if (load==1’b1) q=d;
  endmodule
```

![DFF Diagram]
Example of Verilog-HDL

- 4-bit up counter with load and enable signals

```verilog
module counter(q, in, load, enable, rst, clk);
    output [3:0] q;
    input [3:0] in;
    input clk, load, rst, enable;
    reg [3:0] q;
    always @(posedge clk or negedge rst)
        if (rst == 1'b0) q = 0;
        else if (load == 1'b1) q = in;
        else if (enable == 1'b1) q = q + 1;
    endmodule
```

4-bit up counter with load and enable signals.
Example of Verilog-HDL

- **Register with Combination Logic**

```verilog
module DFFE(Out, A, B, CLR, CLK);
  output Out;
  input A, B, CLR, CLK;
  reg Out;

  always @(posedge CLK or negedge CLR)
    if (CLR == 1'b0) Out = 0;
    else Out = (A & Out) | B;
endmodule
```
Mealy Machine (Finite State Machine)

```verilog
module mealy(out, in, rst, clk);
    output out;
    input in;
    input clk, rst;
    reg out;
    reg [1:0] state;
    parameter s0=2’d0, s1=2’d1, s2=2’d2, s3=2’d3;
    always @(posedge clk or negedge rst)
        if (rst==0) begin state=s0; out=0; end
        else begin
            case (state)
                s0: if (in==0) begin out=0; state=s1; end
                    else begin out=0; state=s0; end
                s1: if (in==0) begin out=0; state=s1; end
                    else begin out=0; state=s2; end
                s2: if (in==0) begin out=0; state=s3; end
                    else begin out=0; state=s0; end
                s3: if (in==0) begin out=0; state=s1; end
                    else begin out=1; state=s2; end
                default: state=s0;
            endcase
        end
    endmodule
```
Moore Machine

module moore(out, in, rst, clk);
output out;
input in;
input clk, rst;
reg out;
reg [1:0] state;
parameter s0=2’d0, s1=2’d1, s2=2’d2, s3=2’d3;

always @(posedge clk or negedge rst)
  if (rst==0) begin state=s0; out=0; end
  else begin
    case (state)
      s0: begin out=0; if (in==0) state=s1; else state=s0; end
      s1: begin out=0; if (in==0) state=s1; else state=s2; end
      s2: begin out=0; if (in==0) state=s3; else state=s0; end
      s3: begin out=1; if (in==0) state=s1; else state=s2; end
      default: state=s0;
    endcase
  end
endmodule
### RAM

```verilog
module RAM(out, in, addr, RW, CS);
    output [7:0] out;
    input [7:0] in;
    input [3:0] addr;
    input RW, CS;
    reg [7:0] out;
    reg [7:0] DATA[15:0];
    always @(negedge CS)
        begin
            if (RW==1'b0) //READ
                out=DATA[addr];
            else
                if (RW==1'b1) //WRITE
                    DATA[addr]=in;
                else
                    out=8'b0;
        end
endmodule
```
ROM

module ROM_Qe(out, addr, CS);
output [15:0] out;
input [3:0] addr;
input CS;
reg [15:0] out;
reg [15:0] ROM[15:0];
always @(negedge CS)
begin
    ROM[0]=16'h5601;   ROM[1]=16'h3401;
    ROM[2]=16'h1801;   ROM[3]=16'h0a01;
    ROM[8]=16'h4801;   ROM[9]=16'h3801;
    ROM[12]=16'h1c01;  ROM[13]=16'h1601;
    out=ROM[addr];
end
endmodule
I/O read/write

```verilog
reg clk, rst, start;
reg buffer[255:0];

initial
begin
    clk=0; rst=0; start=0;
    $readmemb("c:/temp/in.dat",buffer);
    #10 rst=1;
    #30 start=1;
end

reg clk, rst;
integer f2;
parameter D=10;

initial
begin
    clk=0;rst=0;
    #D #D #D rst=1;
    f2=$fopen("c:/lss.dat");
    $fdisplay(f2,"%d", out);
    $fclose(f2);
    $stop; $finish;
end
endmodule
```
User Defined Primitives

- UDPs permit the user to augment the set of pre-defined primitive elements.
- Use of UDPs may reduce the amount of memory required for simulation.
- Both level-sensitive and edge-sensitive behavior is supported.
## UDP Table Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Interpretation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Logic 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Logic 1</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>Iteration of 0, 1, and x</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Iteration of 0 and 1</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>No change</td>
<td></td>
</tr>
<tr>
<td>(vw)</td>
<td>Change of value from v to w</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Same as (??)</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>Same as (01)</td>
<td>Any value change on input</td>
</tr>
<tr>
<td>f</td>
<td>Same as (10)</td>
<td>Rising edge on input</td>
</tr>
<tr>
<td>p</td>
<td>Iteration of (01), (0x), and (x1)</td>
<td>Falling edge on input</td>
</tr>
<tr>
<td>n</td>
<td>Iteration of (10), (1x), and (x0)</td>
<td>Positive edge including x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Negative edge including x</td>
</tr>
</tbody>
</table>
UDP Definition

- Pure combinational Logic

```verilog
primitive mux(o,a,b,s);
output o;
input a,b,s;

table
// a b s : o
 0 ? 1 : 0;
 1 ? 1 : 1;
? 0 0 : 0;
? 1 0 : 1;
0 0 x : 0;
1 1 x : 1;
endtable
endprimitive
```

- The output port must be the first port.
- UDP definitions occur outside of a module
- All UDP ports must be declared as scalar inputs or outputs. UDP ports cannot be inout.
- Table columns are inputs in order declared in primitive statement-colon, output, followed by a semicolon.
- Any combination of inputs which is not specified in the table will produce an 'x' at the output.
UDP Definition (cont.)

- **Level-sensitive Sequential Behavior**

```verilog
class latch(q,clock, data);
output q;
reg q;
input clock, data;

table
// clock data : state_output : next_state
0   1 : ?   : 1;
0   0 : ?   : 0;
1   ? : ?   : -;
endtable
endclass
```

- The '?' is used to represent don't care condition in either inputs or current state.
- The '-' in the output field indicates 'no change'.
UDP Definition (cont.)

● **Edge-sensitive Sequential Behavior**

```verilog
primitive d_edge_ff(q,clock,data);
output q;
reg q;
input clock,data;

// obtain output on rising edge of clock
// clock  data  state  next
// ignore negative edge of clock
(01)  0 : ? : 0;
(01)  1 : ? : 1;
(0x)  1 : 1 : 1;
(0x)  0 : 0 : 0;
// ignore data changes on steady clock
? (??) : ? : -;
endtable
endprimitive

Note that the table now has edgeterms representing transitions on inputs.
```
Logic Strength Modeling

- Verilog provides multiple levels of logic strengths for accurate modeling of signal contention.
- Logic strength modeling resolves combinations of signals into known or unknown values to represent the behavior of the hardware with maximum accuracy.

![Diagram showing logic gates with W/L ratios and input conditions](image)

without strength modeling,
if $A=0$, $B=1$,
then $Y=X$ (unknown).

however, the real situation is
if $A=0$, $B=1$,
then $Y=0$. 
Logic Strength Modeling

- Adding logic strength properties to Verilog primitive.

When $A=0$ and $B=1$, Verilog-XL will resolve $Y$ to logic 0.

Strength

<table>
<thead>
<tr>
<th>0 Strength</th>
<th>1 Strength</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 Su0</td>
<td>0 HiZ0</td>
</tr>
<tr>
<td>6 St0</td>
<td>1 Sm1</td>
</tr>
<tr>
<td>5 Pu0</td>
<td>2 Me1</td>
</tr>
<tr>
<td>4 La0</td>
<td>3 We1</td>
</tr>
<tr>
<td>3 We0</td>
<td>4 La1</td>
</tr>
<tr>
<td>2 Me0</td>
<td>5 Pul</td>
</tr>
<tr>
<td>1 Sm0</td>
<td>6 St1</td>
</tr>
<tr>
<td>0 HiZ0</td>
<td>7 Su1</td>
</tr>
</tbody>
</table>

strong → weak → strong
Logic Strength Modeling

- **Signal Strength Value System.**

<table>
<thead>
<tr>
<th>Strength</th>
<th>Values</th>
<th>%v formats</th>
<th>Specification mnemonics</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 Supply</td>
<td>Drive</td>
<td>Su0</td>
<td>supply0</td>
</tr>
<tr>
<td>6 Strong</td>
<td>Drive (default)</td>
<td>St0</td>
<td>strong0</td>
</tr>
<tr>
<td>5 Pull</td>
<td>Drive</td>
<td>Pu0</td>
<td>pull0</td>
</tr>
<tr>
<td>4 Large</td>
<td>Capacitive</td>
<td>La0</td>
<td>large</td>
</tr>
<tr>
<td>3 Weak</td>
<td>Drive</td>
<td>We0</td>
<td>weak0</td>
</tr>
<tr>
<td>2 Medium</td>
<td>Capacitive</td>
<td>Me0</td>
<td>medium</td>
</tr>
<tr>
<td>1 Small</td>
<td>Capacitive</td>
<td>Sm0</td>
<td>small</td>
</tr>
<tr>
<td>0 High Z</td>
<td>Impedance</td>
<td>HiZ0</td>
<td>highZ0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HiZ1</td>
<td>highZ1</td>
</tr>
</tbody>
</table>
Logic Strength Modeling

- If two values of unequal strength combines in a wired net configuration, the stronger signal is the result.

*Example:*

```
Pull ---- Strong0
|        |
|        |
|        |
Supply1 ---- Supply1
|        |
|        |
|        |
Large1
```
Logic Strength Modeling

● Syntax
  
  `<GATETYPE><drive_strength>?<delay><gate_instance>;`
  
  where `<drive_strength>` may be `(strength0, strength1)` or `(strength1, strength0)`

● Example
  
  not (weak1, strong0)  u0(Y, A);
  not (strong0, strong1) u1(Y, A);

● You can use `%v` format specificity to display the strength of a net.

  `$monitor("at time=%t, the strength of Y is %v", $time, Y);`
Specify Blocks

• What is a specify block?
  - Specify block let us add timing specifications to paths across a module.

![Full Adder Diagram]

\[
\begin{align*}
Tlh_a_{\text{to Sum}} &= 1.2 & Tlh_b_{\text{to Sum}} &= 1.5 \\
Thl_a_{\text{to Sum}} &= 2.0 & Thl_b_{\text{to Sum}} &= 2.2 \\
\text{...} & & \text{...}
\end{align*}
\]
module DFF (q, d, clk);
input d, clk;
output q;
reg notifier;

UDP_DFF u0(q, d, clk notifier);

specify
specparam
InCap$d = 0.024, Tsetup$d_cp = 0.41, Thold$d_cp = 0.2;
...
(clk => q) = (0.390, 0.390);
...
$setup(d, posedge clk, Tsetup$d_cp, notifier);
$hold(posedge clk, d, Thold$d_cp, notifier);
}
perform timing check

endspecify
endmodule
Starting the Verilog-XL Simulation

- **UNIX Environment**
  
  verolog <command_line_options> <design file>

  *Example 1:*
  
  unix> verilog adder.v

  *Example 2:*
  
  unix> verilog file1.v file2.v file3.v
  
  or
  
  unix> verilog –f file4

  file4 content in the text mode:

  file1.v
  file2.v
  file3.v
Testing and Verification of Full Adder

- Test the full adder’s Verilog model by applying test patterns and observing its output responses.
  - Stimulus and control: Changes on device inputs, simulation finish time, ... etc.
  - Device under test: Behavior, gate, or switch level modules.
  - Response generation and verification: Which signals to save/display, verification of generated response.
### Circuit Description

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

endmodule

### Testfixture

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

endmodule

---

0.00 ns  in = 0  out = x
16.00 ns  in = 0  out = 1
100.00 ns in = 1  out = 1

... ...
Example with a Test Fixture

- A Full Adder

```verilog
module testfixture;
reg     a, b, cin;
wire    sum, carry;

adder   u0 (carry, sum, a, b, cin);

initial begin
$monitor($time, "a=%b b=%b cin=%b sum=%b carry=%b", a, b, cin, sum, carry);

   a=0; b=0; cin=0;
   #10 a=0; b=0; cin=1;
   #10 a=0; b=1; cin=0;
   #10 a=0; b=1; cin=1;
   #10 a=1; b=0; cin=0;
   #10 a=1; b=0; cin=1;
   #10 a=1; b=1; cin=0;
   #10 a=1; b=1; cin=1;
   #10 $stop; #10 $finish;
end
endmodule
```

```verilog
module adder (carry, sum, a, b, cin);
output carry, sum;
input a, b, cin;
wire w0, w1, w2;

xor    u0(sum, a, b, cin);
and    u1(w0, a, b);
and    u2(w1, b, cin);
and    u3(w2, cin, b);
or     u4(carry, w0, w1, w2);
endmodule
```

This will generate some text outputs as

```
0  a=0 b=0 c=0 sum=0 carry=0
10 a=0 b=0 c=1 sum=1 carry=0
...
```
Useful System Tasks

- **Always and Initial**
  - `always`
  - `initial`
    - `$stop`: Stopping the simulation.
    - `$finish`: Finishing the simulation.

- **Monitoring Commands**
  - **Text Format Output**
    - `$monitor($time,"a=%d, b=%b,...\n",a,b);`
  - **Graphic Output**
    - `$gr_waves("<signal_label>",<signal>, ...);`
    - `$SimWave: $shm_open(“<file_name>”), $shm_probe( )`
Monitor System Task

- Any expression parameter that has no corresponding format specification is displayed using the default decimal format.

<table>
<thead>
<tr>
<th>Format</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>%h</td>
<td>display in hexadecimal format</td>
</tr>
<tr>
<td>%d</td>
<td>display in decimal format</td>
</tr>
<tr>
<td>%o</td>
<td>display in octal format</td>
</tr>
<tr>
<td>%b</td>
<td>display in binary format</td>
</tr>
<tr>
<td>%c</td>
<td>display in ASCII character format</td>
</tr>
<tr>
<td>%v</td>
<td>display net signal strength</td>
</tr>
<tr>
<td>%n</td>
<td>display net normalized voltage in Switch-RC</td>
</tr>
<tr>
<td>%m</td>
<td>display hierarchical name</td>
</tr>
<tr>
<td>%s</td>
<td>display as a string</td>
</tr>
<tr>
<td>%t</td>
<td>display in current time format</td>
</tr>
</tbody>
</table>
SimWave

- Using system tasks to save the circuit state into waveform database.
- You can use SimWave to view the signal waveforms after Verilog-XL simulation.

Example

```verilog
module testfixture;

    ...

    initial begin
        $shm_open("adder.shm");
        $shm_probe("A");
        ...
        #10 $finish; end

endmodule
```
SimWave Window
Trouble Shooting

- If \( a = b \) is triggered by some event, \( a \) must be declared as \text{reg}.
- A bus signal must be declared as \text{wire}.
- The negative value \text{should be sign-extended}.
- The port size and number of a module should match anywhere it is referred.