Verilog Language Concepts

Adapted from Z. Navabi
Portions Copyright Z. Navabi, 2006

- Characterizing Hardware Languages
  - Timing
  - Concurrency
  - Timing and concurrency example

- Module Basics
  - Code format
  - Logic value system
  - Wires and variables
  - Modules
  - Module ports
  - Names
  - Numbers
Verilog Language Concepts

- Module Basics (cont.)
  - Arrays
  - Verilog operators
  - Verilog data types
  - Array indexing

- Verilog Simulation Model
  - Continuous assignments
  - Procedural assignments

Verilog Language Concepts

- Compiler Directives
  - `timescale`
  - `default-nettype`
  - `include`
  - `define`
  - `ifdef, else, endif`
  - `unconnected-drive`
  - `celldefine, endcelldefine`
  - `resetall`
Verilog Language Concepts

- System Tasks and Functions
  - Display tasks
  - File I/O tasks
  - Timescale tasks
  - Simulation control tasks
  - Timing check tasks
  - PLA modeling tasks
  - Conversion functions for reals
  - Other tasks and functions

- Summary

Characterizing Hardware Languages

- Timing and concurrency are the main characteristics of hardware description languages.

- **Timing** is associated with values that are assigned to hardware carriers.

- **Concurrency** refers to simultaneous operation of various hardware components.
Timing

- Transfer of data is done through wires or busses and some of delays are associated with transfer of data through wires.

- Variables in Verilog may be used for representation of wires and variable assignments can include timing specification.

```
assign w1 = a & b | c & ~b;
assign #6 n = ~b;
assign #3 m = a & b;
assign #3 p = n & c;
assign #2 w2 = m | p;
```

A more accurate assignment:

```
assign w = #3 a & b;
assign #3 m = a & b;
assign #3 p = n & c;
assign #2 w2 = m | p;
```

An assign statement drives the signal on the left-hand side of the equal sign with the boolean expression on its right.

- An AND-OR Circuit
Timing

When \( b \) changes from 1 to 0 the final value of \( \bar{w} \) remains at 1. However, because of the inverter delay in forwarding a 1 to the output, a 6 ns glitch appears on \( \bar{w} \).

- Output Glitch
  - \( \bar{w}_1 \), which is the result of the single assign statement, does not show the delay
  - \( \bar{w}_2 \) shows both propagation delays and glitches that may occur on the \( \bar{w} \) output of circuit

Concurrency

Characterizing Hardware Languages

Timing

Concurrency

Timing & Concurrency Example
Concurrent

* Concurrency is an essential feature of any language for description of hardware.

* Functionality of a hardware system is described by concurrent sub-components or by a program in a sequential manner.

* By using concurrent sub-components Verilog simulator makes us think that simulation of components is being done concurrently.

An AND-OR Circuit

```
assign #6 n = ~b;
assign #3 m = a & b;
assign #3 p = n & c;
assign #2 w2 = m | p;
```

Regarded as concurrent. The order in which these statements appear in a concurrent body of Verilog is not important.

perform their operations.
Timing & Concurrency Example

Characterizing Hardware Languages

Timing & Concurrency Example

Module header declares inputs and outputs of circuit.

timescale 1ns/100ps

module FullAdder (input a, b, ci, output co, s);
assign co = a & b | a & ci | b & ci);
assign s = a ^ b ^ ci);
endmodule

Full Adder Description

Propagation Delay of Assignment

Because of the delay values, if an input change causes both outputs to change, s changes before co does.
Timing & Concurrency Example

```
timescale 1ns/100ps
module FullAdderTester;
    reg a = 0, b = 0, ci = 0;
    wire co, s;
    parameter tlimit = 500;
    FullAdder MUT (a, b, ci, co, s);
................................
................................
................................
```

* Fulladder Tester Procedural Description

```
always begin
    if ($time >= tlimit) $stop;
    else begin
        #17;
        a = ~a;
        #13;
        ci = ~ci;
        #19;
        b = ~b;
    end
end
endmodule
```

* Fulladder Tester Procedural Description (Continued)
Now we are going to see:

- How modules are developed
- How names, numbers and operators are used
Code Format

- Verilog code is free format.
- Spaces and new lines are served as separators.
- It is case sensitive.
- Language keywords use lowercase characters.
- A comment designator start with // makes the rest of line comment.
- The symbols /* ... */ bracket the section of code which is in between as a comment.
* Bit type, or bits of vectors or arrays, of Verilog wires and variables take the *4-value logic* value system.

* Values in this system are 0, 1, Z and X.

* The values 0 and 1 have Three modes: **Forcing**, **Resistive** and **Capacitive**.

* The Z value represents an undriven, high impedance value.

* The X value represents a conflict in multiple driving values, an unknown or value of a variable not initialized.
### Logic Value System

<table>
<thead>
<tr>
<th>Value</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td><img src="0.png" alt="Diagram" /></td>
</tr>
<tr>
<td>1:</td>
<td><img src="1.png" alt="Diagram" /></td>
</tr>
<tr>
<td>X or x:</td>
<td><img src="X.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Z or z:</td>
<td><img src="Z.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

*Logic Values and Examples*

### Wires and Variables

- **Module Basics**
  - Code Format
  - Logic Value System

- **Wires and Variables**
  - Modules
  - Module Ports
  - Names
  - Numbers
  - Arrays
  - Verilog Operators
  - Verilog Data Types
  - Array Indexing
Wires and Variables

- Wires and Variables:
  - `net`: represents a wire driven by a hardware structure or output of a gate.
  - `reg`: represents a variable that can be assigned values in behavior description of a component in a Verilog procedural block.

Modules

- Module Basics
  - Code Format
  - Logic Value System
- Wires and Variables
- Modules
- Module Ports
- Names
- Numbers
- Arrays
- Verilog Operators
- Verilog Data Types
- Array Indexing
* Module is the main structure of definition of hardware components and testbenches.
* Begins with `module` keyword and end with `endmodule`.
* Immediately following the `module` keyword, port list of the module appears enclosed in parenthesis.

```
`timescale 1ns/100ps

module FlipFlop (preset, reset, din, clk, qout);
  input preset, reset, din, clk;
  output qout;
  reg qout;
  always @(posedge clk) begin
    if (reset) qout <= #7 0;
    else if (preset) qout <= #7 1;
    else qout <= #8 din;
  end
endmodule
```

* Separate Port Declarations Statements

Ports are only listed in the port list and declared as separate input and output ports inside the body of the Flip-Flop module.
Module Ports

- Inputs and outputs of a model must be declared as:
  - input
  - output
  - inout

- By default, all declared ports are regarded as net and the default net type is used for the ports.

- Ports declared as output may be declared as reg. This way they can be assigned values in procedural blocks.

- An inout port can be used only as a net. To assign values to an inout port in procedural bodies, a reg corresponding to the port must be declared and used.

- For an output, a reg specification can follow the output keyword in the port list of the module.
Names

- A stream of characters starting with a letter or an underscore forms a Verilog identifier.
- The $ character and underscore are allowed in an identifier.

- Verilog uses **keywords** that are all formed by streams of **lowercase characters**.
- The names of **system tasks** and **functions** begin with a $ character.
- **Compiler directive names** are preceded by the ` (back single quote) character. Example: `timescale
Names

- The following are valid names for identifiers:
  
a_name, name1, _name, Name,
Name$, name55, _55name, setup,
$_name.

- The following are Verilog keywords or system tasks.
  
$display, default, $setup,
begin, tri1, small.
Numbers

- Constants in Verilog are integer or real.
- Specification of integers can include X and Z in addition to the standard 0 and 1 logic values.
- Integers may be
  - Sized: Begins with the number of equivalent bits
  - Unsized: Without the number of bits specification
- The general format for a sized integers is:
  \[ \text{number of bits} \, ^{\text{base identifier}} \, \text{digits} \]
  example: \(6'b101100\)
  
  The base specifier is a single lower or uppercase character \(b, d, o\) or \(h\) which respectively stand for binary, decimal, octal and hexadecimal bases.

Numbers

- Optionally, the base-identifier can be preceded by the single character \(s\) (or \(S\)) to indicate a signed quantity.
- A plus or minus operator can be used on the left of the number specification to change the sign of the number.
- The underscore character (_) can be used anywhere in a number for grouping its bits or digits for readability purposes.

- Real constants in Verilog use the standard format as described by IEEE std 754-1985, the IEEE standard for double precision floating-point numbers. Examples: 1.9, 2.6E9, 0.1e-6, 315.96-12.
### Numbers

#### Number Representation Examples

<table>
<thead>
<tr>
<th>Number representation</th>
<th>Binary Equivalent</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>4’d5</td>
<td>101</td>
<td>Decimal 5 is interpreted as a 4-bit</td>
</tr>
<tr>
<td>8’b101</td>
<td>101</td>
<td>Binary 101 is turned into an 8-bit</td>
</tr>
<tr>
<td>12’h5B_3</td>
<td>10110110011</td>
<td>Binary equivalent of hex;</td>
</tr>
<tr>
<td>10’o752</td>
<td>01111010100</td>
<td>This is the complement of the 8-bit 752 with a 0 padded to itself to make it a 10-bit number.</td>
</tr>
<tr>
<td>8’hF</td>
<td>0000111111</td>
<td>Hexadecimal F is expanded to 12 bits by padding zeros to its left.</td>
</tr>
<tr>
<td>12’hxA</td>
<td>xxxxxxxx10100</td>
<td>Hexadecimal XA is expanded to 12 bits by extending the left x.</td>
</tr>
<tr>
<td>12’shA6</td>
<td>111101010011</td>
<td>This is an 8-bit number treated as a signed number.</td>
</tr>
<tr>
<td>-4’shA</td>
<td>Signed 0110</td>
<td>The 2’s complement (because of the minus sign) 4-bit A is regarded as a signed number.</td>
</tr>
</tbody>
</table>

#### Variables

```verilog
module NumberTest;

reg [11:0] a = 8’shA6; // 111110100110
initial $displayb("a=", a);

reg [11:0] b = 8’sh6A; initial $displayb("b=", b);
// b=000001101010

reg [11:0] c = 'shA6; initial $displayb("c=", c);
// c=000101100110

reg [11:0] d = 'sh6A; initial $displayb("d=", d);
// d=000001101010

reg [11:0] e = -8’shA6; initial $displayb("e=", e);
// e=000001101110

..............................................
```

* Integer Constants
**Numbers**

```verilog
reg [11:0] f = -'shA6; initial $display("f=", f);
// f=1111010110
reg [11:0] g = 9'shA6; initial $display("g=", g);
// g=000010110110
reg [11:0] h = 9'sh6A; initial $display("h=", h);
// h=000001101010
reg [11:0] i = -9'shA6; initial $display("i=", i);
// i=111101011010
reg [11:0] j = -9'sh6A; initial $display("j=", j);
// j=111110010110
reg [11:0] k = 596; initial $display("k=", k);
// k=001001010100
reg [11:0] l = -596; initial $display("l=", l);
// l=110110101100
endmodule
```

* Integer Constants (Continued)
Arrays

- Verilog allows declaration and usage of multidimensional arrays for nets or regs.
- The following declares \texttt{a\_array} as a two-dimensional array of 8-bit words:
  \begin{itemize}
    \item \texttt{reg [7:0] a\_array [0:1023][0:511];}
  \end{itemize}
- In an array declaration, the address range of the elements of the array comes after the name of the array.
- \textbf{Range specifications} are enclosed in \texttt{square brackets}.
- The size and range specification of the elements of an array come after the \texttt{net} type (e.g., \texttt{wire}) or \texttt{reg} keyword.
- In the absence of a range specification before the name of the array, an \texttt{element size of one bit} is assumed.

\begin{itemize}
  \item Arrays
  \end{itemize}

\begin{itemize}
  \item Arrays
    \begin{itemize}
      \item \texttt{// An 8-bit vector}
        \begin{itemize}
          \item \texttt{reg [7:0] Areg;}
        \end{itemize}
      \item \texttt{// A memory of 8 one-bit elements}
        \begin{itemize}
          \item \texttt{reg Amem [7:0];}
        \end{itemize}
    \end{itemize}
  \end{itemize}

- Array Structures
Arrays

// A two-dimensional memory of one-bit elements
reg Bmem [7:0] [0:3];

// A memory of four 8-bit words
reg [7:0] Cmem [0:3];

* Array Structures (Continued)

Arrays

// A two-dimensional memory of 3-bit elements
reg [2:0] Dmem [0:3] [0:4];

* Array Structures
Verilog Operators

Module Basics

Code Format

Logic Value System

Wires and Variables

Modules

Module Ports

Names

Numbers

Arrays

Verilog Operators

Verilog Data Types

Array Indexing

* Verilog Operations
Verilog Operators

Basic Operators

Equality Operators

Boolean Operators

Shift Operators

Concatenation Operators

Conditional Operators

Basic Operators

Verilog Operators

Basic Operators

Equality Operators

Boolean Operators

Shift Operators

Concatenation Operators

Conditional Operators
Basic Operators

- Arithmetic Operations in Verilog take bit, vector, integer and real operands.
- Basic operators of Verilog are +, -, *, / and **.
- An X or a Z value in a bit of either of the operands of a multiplication causes the entire result of the multiply operation to become X.
- Unary plus (+) and minus (−) are allowed in Verilog. These operators take precedence over other arithmetic operators.
- If any of the operands of a relational operator contain an X or a Z, then the result becomes X.

<table>
<thead>
<tr>
<th>Example</th>
<th>Results in</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 * 8’b6</td>
<td>150</td>
</tr>
<tr>
<td>25 + 8’b7</td>
<td>32</td>
</tr>
<tr>
<td>25 / 8’b6</td>
<td>4</td>
</tr>
<tr>
<td>22 % 7</td>
<td>1</td>
</tr>
<tr>
<td>8'b10110011 &gt; 8'b0011</td>
<td>1</td>
</tr>
<tr>
<td>4'b1011 &lt; 10</td>
<td>0</td>
</tr>
<tr>
<td>4'b1z10 &lt; 4'b1100</td>
<td>x</td>
</tr>
<tr>
<td>4'b1x10 &lt; 4'b1100</td>
<td>x</td>
</tr>
<tr>
<td>4'b1x10 &lt;= 4'b1x10</td>
<td>x</td>
</tr>
</tbody>
</table>

- Examples of Basic Operations
Equality Operators

Equality operators are categorized into two groups:

- **Logical**: Compare their operands for equality (==) or inequality (!=)
  - Return a one-bit result, 0, 1, or Z

- An X ambiguity arises when an X or a Z occurs in one of the operands.

  - **Case**: Consider X and Z values in comparing their operands.
  - The result is always 0 or 1.
### Equality Operators

<table>
<thead>
<tr>
<th>Example</th>
<th>Results in</th>
</tr>
</thead>
<tbody>
<tr>
<td>8'b10110011 == 8'b10110011</td>
<td>1</td>
</tr>
<tr>
<td>8'b1011 == 8'b00001011</td>
<td>1</td>
</tr>
<tr>
<td>4'b1100 == 4'b1210</td>
<td>0</td>
</tr>
<tr>
<td>4'b1100 != 8'b100X</td>
<td>1</td>
</tr>
<tr>
<td>8'b1011 != 8'b00001011</td>
<td>0</td>
</tr>
<tr>
<td>8'b101X == 8'b101X</td>
<td>1</td>
</tr>
</tbody>
</table>

* Examples of Equality Operations

### Boolean Operators

- **Verilog Operators**
  - **Basic Operators**
  - **Equality Operators**
  - **Boolean Operators**
  - **Shift Operators**
  - **Concatenation Operators**
  - **Conditional Operators**
Boolean Operators

* If an X or a Z appears in an operand of a logical operator, an X will result.
* The complement operator ~ results in 1 and 0 for 0 and 1 inputs and X for X and Z inputs.

<table>
<thead>
<tr>
<th>&amp;</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>x</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>x</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>

* Bit-by-bit Bitwise and Reduction Operators
**Boolean Operators**

- Complement reduction operations (~&, ~|, and ~^) perform reduction first and then complement the result.

<table>
<thead>
<tr>
<th>Example</th>
<th>Results in</th>
</tr>
</thead>
<tbody>
<tr>
<td>8'b01101110 &amp;&amp; 4'b0</td>
<td>0</td>
</tr>
<tr>
<td>8'b01101110</td>
<td></td>
</tr>
<tr>
<td>8'b01101110 &amp;&amp; 8'b10010001</td>
<td>1</td>
</tr>
<tr>
<td>! (8'b10010001)</td>
<td>1</td>
</tr>
<tr>
<td>8'b01101110 &amp; 8'bxxxx1100</td>
<td>8'b0xx01100</td>
</tr>
<tr>
<td>8'b01101110</td>
<td>8'bxxxx1100</td>
</tr>
<tr>
<td>~&amp; (4'b0xz1)</td>
<td>1</td>
</tr>
<tr>
<td>~</td>
<td>(4'b0xz1)</td>
</tr>
</tbody>
</table>

* Logical, Bit-wise, and Reduction

**Shift Operators**

- Basic Operators
- Equality Operators
- Boolean Operators
- Shift Operators
- Concatenation Operators
- Conditional Operators
Shift Operators

- Logical shift operators (\texttt{>>} and \texttt{<<} for shift right and left) fill the vacated bit positions with zeros.
- Fill values for arithmetic shift operators depend on the type of their results being signed or unsigned.

<table>
<thead>
<tr>
<th>Example</th>
<th>Results in</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{8'b0110_0111 \ll 3}</td>
<td>\texttt{8'b0011_1000}</td>
</tr>
<tr>
<td>\texttt{8'b0110_0111 \ll \texttt{1'bz}}</td>
<td>\texttt{8'bxxxx_xxxx}</td>
</tr>
<tr>
<td>\texttt{Signed_LHS = 8'b1100_0000\gg\gg2}</td>
<td>\texttt{8'b1111_0000}</td>
</tr>
</tbody>
</table>

- Shift Operators

Concatenation Operators
Concatenation Operators

- The notation used for this operator is a pair of curly brackets ( {... } ) enclosing all scalars and vectors that are being concatenated.

- If a is a 4-bit reg and aa is a 6-bit reg, the following assignment places 1101 in a and 001001 in aa:
  \[ \{ a, \text{aa} \} = 10\text{'b}1101001 \]

- If the a and aa registers have the values assigned to them above, and aaa is a 16-bit reg data type, then the assignment,
  \[ \text{aaa} = \{ \text{aa}, \{ 2 \text{(a)} \}, 2\text{'b}11 \} \]
  puts 001001_1101_1101_11 in aaa.

  - \{ a, 2\{ b,c \}, 3\{ d \} \} is equivalent to: \{ a, b, c, b, c, d, d, d \}
  - \{ 2\text{'b}00, 3\{ 2\text{'01} \}, 2\text{'b}11 \} results in: 10\text{'b}0001010111
Conditional Operators

- expression1 ? expression2 : expression3

If expression1 is true, then expression2 is selected as the result of the operation; otherwise expression3 is selected.

- If expression1 is X or Z, both expressions 2 and 3 will be evaluated, and the result becomes the bit-by-bit combination of these two expressions.

assign a = (b == c)? 1 : 0;

...
Conditional Operators

- Conditional Operators

<table>
<thead>
<tr>
<th>Example</th>
<th>Results in</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ? 4'b1100 : 4'bZX0</td>
<td>4'b1100</td>
</tr>
<tr>
<td>0 ? 4'b1100 : 4'b1ZX0</td>
<td>4'b1ZX0</td>
</tr>
<tr>
<td>x ? 4'b1100 : 4'b1ZX0</td>
<td>4'b1XX0</td>
</tr>
</tbody>
</table>

Precedence of Operators

- Operator Precedence
**Precedence of Operators**

- Precedence Examples

```plaintext
W & X + Y    A & B && C + D
              &    &
```

**Verilog Data Types**

- Module Basics
- Code Format
- Logic Value System
- Wires and Variables
- Modules
- Module Ports
- Names
- Numbers
- Arrays
- Verilog Operators
- Verilog Data Types
- Array Indexing
Net Declarations

- Types `wire` and `tri`, `wand` and `triand`, and `wor` and `trior` are equivalent.

- Types `supply0` and `supply1` are used for declaring signal names for supply voltages.

- The `tristate` net type declares three-state capacitive signals.

- Other net types (`wire`, `wand` and `wor` or their equivalents, `tri`, `triand` and `trior`) declare state signals that allow multiple driving sources.

This statement declares wires used between gates or Boolean expressions representing logic structures.

```verilog
wire w, n, m, p;
```

- By default, ports of a module are net of `wire` type.

- The `Z` value is the weakest and is overridden by non-Z values from other driving sources.
**Net Declarations**

- Driving a **wire** with multiple 0 and 1 conflicting values resolves in the X value for the **wire**.

- The **wand** and **wor** type nets signify wired-and and wired-or functions, respectively.

- For **wand** type, a 0 value on a driving source overrides all other source values and value Z is treated as null and is overridden by any other value driving a **wand** net.

- In **wor** operation, logic value 1 on one source overrides all other source values. As in **wand**, the Z value is the weakest and is overridden by 0, 1 and X values.

---

**Net Declarations**

<table>
<thead>
<tr>
<th>NET TYPES</th>
<th>PROPERTIES</th>
<th>INITIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>supply0</td>
<td>Driver: 0</td>
<td>0</td>
</tr>
<tr>
<td>supply1</td>
<td>Driver: 1</td>
<td>1</td>
</tr>
<tr>
<td>Three-state</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wire (tri)</td>
<td>Tri-state wired logic</td>
<td>Driver: X Not Driven: Z</td>
</tr>
<tr>
<td>wand (triand)</td>
<td>Wired-and logic</td>
<td>Driver: X Not Driven: Z</td>
</tr>
<tr>
<td>wor (trior)</td>
<td>Wired-or logic</td>
<td>Driver: X Not Driven: Z</td>
</tr>
<tr>
<td>Capacitive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tristatereg</td>
<td>Hold old value</td>
<td>X</td>
</tr>
</tbody>
</table>

- **net** Types and Properties
Net Declarations

\[ \begin{array}{c|c|c|c}
\text{Value} & 0 & 1 & X \\
\hline
\text{Value} & 0 & X & X \\
\text{Value} & 1 & X & X \\
\text{Value} & X & X & X \\
\text{Value} & Z & 0 & X \\
\end{array} \]

- \text{wire type}

\[ \begin{array}{c|c|c|c}
\text{Value} & 0 & 1 & X \\
\hline
\text{Value} & 0 & 0 & 0 \\
\text{Value} & 0 & 1 & X \\
\text{Value} & X & X & X \\
\text{Value} & Z & 0 & 1 \\
\end{array} \]

- \text{wand type}

\[ \begin{array}{c|c|c|c}
\text{Value} & 0 & 1 & X \\
\hline
\text{Value} & 0 & 1 & X \\
\text{Value} & 1 & 1 & 1 \\
\text{Value} & X & 1 & X \\
\text{Value} & Z & 0 & 1 \\
\end{array} \]

- \text{wor type}

\text{(a) \text{wand net} Type, (b) \text{wor net} Type}
Net Declarations

- The `trig` type net behaves as a capacitive wire and holds its old value when a new resolved value is to become Z. As long as there is at least one driver with 0, 1, or X value, `trig` behaves the same as wire. When all drivers are turned off (Z), a `trig` net retains its previous value.

- The amount of time a `trig` net holds a value is specified by a delay parameter in its declaration. Delay parameters will be discussed next. Chapter 7 shows examples of using `trig` for CMOS flip-flop modeling.

- Three delay values for net switching to 1, to 0, and to Z are specified in a set of parenthesis that are followed by a # sign after the net type keyword. A simpler format contains a single delay value.

```verilog
wire #3 w, n, m, p;
```
**Net Declarations**

- *`trireg`* net types may also be declared with three delay parameters. Unlike the case with other nets, in this case the third timing parameter is not delay for the Z transition. Instead, this specifies the time that a declared `trireg` net holds an old value when driven by Z.

- The initial value for all net types except `supply0` and `supply1` with at least one driver is X. A net with no driver assumes the Z value, except for `trireg`, which has the initial value X.

**Reg Declarations**

- **Verilog Data Types**
  - **Net Declarations**
  - **Signed Data**
  - **Parameters**

- **Reg Declarations**
Reg Declarations

* `reg` is a variable for holding intermediate signal values or nonhardware parameters and function values.

* The `reg` declaration shown below declares `a`, `b` and `ci` as `reg` types with 0 initial values.
  ```
  reg a=0, b=0, ci=0;
  ```

* The default initial value of a declared `reg` is (X).

Reg Declarations

* Other `reg` types are integer and time. An integer declaration declares a signed 2s-complement number, and a time declaration declares an unsigned `reg` variable of at least 64 bits.

* Verilog also allows declaration of real and realtime variables. These variables are similar in use to integer and time variables, but do not have direct bit-to-bit correspondence with `reg` type registers.
Signed Data

Verilog Data Types

Net Declarations

Reg Declarations

Signed Data

Parameters

- Verilog net and reg types can be declared as signed. In below example areg is declared as a signed reg.

  ```verilog
  reg signed [15:0] areg;
  ```

- A signed reg that is shifted right by the `>>>` operator is sign filled, whereas an unsigned reg shifted by this operator is zero-filled.

- If the right-hand side of an assignment is determined as signed, it is sign extended to the size of its left hand side and is placed on the left hand side reg or net.
Parameters

Parameters in Verilog do not belong to either the variable or the net group. Parameters are constants and cannot be changed at runtime. Parameters can be declared as signed, real, integer, time or realtime.

<table>
<thead>
<tr>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter p1=5, p2=6;</td>
<td>32 bit parameters</td>
</tr>
<tr>
<td>parameter [4:0] p1=5, p2=6;</td>
<td>5 bit parameters</td>
</tr>
<tr>
<td>parameter integer p1=5;</td>
<td>32 bit parameters</td>
</tr>
<tr>
<td>parameter signed [4:0] p1=5;</td>
<td>5 bit signed parameters</td>
</tr>
</tbody>
</table>

Parameter Examples
Array Indexing

Module Basics

Code Format

Logic Value System

Wires and Variables

Modules

Module Ports

Names

Numbers

Arrays

Verilog Operators

Verilog Data Types

Array Indexing

Array Indexing

Array Indexing

Bit Selection

Part Selection

Standard Memory

Multi Dimensional Memories
Bit Selection

- Bit-select and part-select operators are used for extracting a bit or a group of bits from a declared array.
  ```verilog
  reg [7:0] Areg;
  reg Amem [7:0];
  reg Dmem [7:0][0:3];
  reg [7:0] Cmem [0:3];
  reg [2:0] Dmem [0:3][0:4];
  ```

- Bit-selection is done by using the addressed bit number in a set of square brackets. For example Areg[5] selects bit 5 of the Areg array.
Verilog allows constant and indexed part-select. For example, `Areg[7:3]` selects the upper five bits of `Areg`.

- `Areg [5:3] //selects bits 5, 4 and 3`
- `Areg [5-:4] //selects bits 5, 4, 3 and 2`
The standard format for declaring a memory in Verilog is to declare it as an array of a vector.

```verilog
reg [7:0] Emem [0:1023];
```

```verilog
Cmem [Areg [7:6]]
```

```verilog
Emem [0]
```

```verilog
Emem [355] [3:0]
```

```verilog
Emem [355] [3-:4]
```

```verilog
Emem [355:358]
```

- Extracts `Emem` word addressed by `Emem[0]`
- Extracts `Emem` word addressed by `Emem[0]`
- Extracts `Emem` word addressed by `Emem[0]`
- Extracts `Emem` word addressed by `Emem[0]`
- Extracts `Emem` word addressed by `Emem[0]`
- Illegal; Does not address a 4-word block

---

Standard Memory

- The standard format for declaring a memory
- Standard Memory
- Bit Selection
- Part Selection
- Multi Dimensional Memories
- Array Indexing

---

Standard Memory

- The standard format for declaring a memory
- Standard Memory
For accessing such memories (e.g. Dmem declared before), simple indexing are allowed for specifying a word in the memory, and bit-select and part-select are allowed for accessing bit or bits of the addressed word.
Multi-Dimensional Memories

// declaration: reg [7:0] Areg;
Areg [7:5]

// declaration: reg Amem [7:0];
Amem [3]

// declaration: reg Bmem [7:0];
Bmem [2] [1]

// declaration: reg Bmem [7:0];
Bmem [2] [1]

// declaration: reg Cmem [7:0];
Cmem [2] [1]

// declaration: reg Dmem [7:0];
Dmem [2] [1]

* Array Addressing & Selection

Verilog Simulation Model

Verilog Simulation Model

Continuous Assignments

Procedural Assignments
Continuous Assignments

Verilog Simulation Model

Continuous Assignments

Procedural Assignments

Simple Assignments

Delay Specification

Strength Specification

Net Declaration Assignments

Multiple Drives
Simple Assignments

- A continuous assignment in Verilog is used only in concurrent Verilog bodies.

- This assignment represents a net driven by a gate output or a logic function.

```
assign w = m | p;
```
Delay Specification

```
* assign #2 w = m | p;
```

This assignment becomes active when m or p changes. At this time, the new value of the m | p expression is evaluated, and after a wait time of 2 time units, this new value is assigned to w.
Delay Specification

```verilog
`timescale 1ns/100ps

module Mux2to1 (input a, b, c, output w);
  wire n, m, p;
  assign #3 m = a & b;
  assign #3 p = n & c;
  assign #6 n = ~b;
  assign #2 w = m | p;
endmodule
```

- Concurrent Continuous Assignments

The simulation of the previous circuit results in a glitch due to a 1-hazard on w. The event driven simulation of concurrent statements makes this simulation to correspond to events in the actual circuit.
Net strengths are specified by a pair of strength values bracketed by a set of parenthesis, as shown below.

```verilog
assign (strong0, strong1) w = m | p;
```

- One strength value is for logic 1 and one is for logic 0, and the order in which the strength values appear in the set of parenthesis is not important.

- Strength value names for logic 1 end with a 1 (supply1, strong1, pull1, weak1, …) and those for logic 0 end with a 0 (supply0, strong0, pull0, weak0, …).
For wire and tri type nets, strength values are used, and for storage nets charge strength is used. Default values for these nets are strong0 and strong1 for logic 0 and logic 1 respectively. Three strength values, large, medium, and small, are used for trireg net types, and the default is medium.

**Net Types and Their Strengths**

<table>
<thead>
<tr>
<th>Net Type</th>
<th>Strength Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wire (tr), tri (triw)</td>
<td>Supply 0: Level 7, Strength value: Large</td>
</tr>
<tr>
<td>Wire (tr), tri (triw)</td>
<td>Strong 0: Level 6, Strength value: Medium (0)</td>
</tr>
<tr>
<td>Wire (tr), tri (triw)</td>
<td>Pull 0: Level 5, Strength value: Small (0)</td>
</tr>
<tr>
<td>Wire (tr), tri (triw)</td>
<td>Weak 0: Level 3, Strength value: Small (1)</td>
</tr>
<tr>
<td>Wire (tr), tri (triw)</td>
<td>Highz 0: Level 1, Strength value: Medium (1)</td>
</tr>
<tr>
<td>Wire (tr), tri (triw)</td>
<td>Highz 1: Level 0, Strength value: Large (1)</td>
</tr>
<tr>
<td>Wire (tr), tri (triw)</td>
<td>Weak 1: Level 2, Strength value: Large (0)</td>
</tr>
<tr>
<td>Wire (tr), tri (triw)</td>
<td>Pull 1: Level 4, Strength value: Medium (0)</td>
</tr>
<tr>
<td>Wire (tr), tri (triw)</td>
<td>Strong 1: Level 6, Strength value: Large (1)</td>
</tr>
<tr>
<td>Wire (tr), tri (triw)</td>
<td>Supply 1: Level 7, Strength value: Large</td>
</tr>
</tbody>
</table>

**Net Declaration Assignments**

- Continuous Assignments
- Simple Assignments
- Delay Specification
- Strength Specification
- Multiple Drives
- Net Declaration Assignments
Net Declaration Assignments

```verilog
`timescale 1ns/100ps

module Mux2to1 (input a, b, c, output w);
    wire #3
        m = a & b,
        p = n & c,
        n = ~b,
        w = m | p;
endmodule
```

* Using `net_declaration_assignment`

In this code, all the continuous assignments of previous code are replaced by a list of net declaration assignments providing drivers for \( w, n, m \) and \( p \) signals.

Multiple Drives

Continuous Assignments

- Simple Assignments
- Delay Specification
- Strength Specification
- Net Declaration Assignments

Multiple Drives
Multiple Drives

```
module Mux2to1 (input a, b, c, output w);
    wire n;
    assign #2 w = a & b;
    assign #3 w = n & c;
    assign #6 n = ~b;
endmodule
```

* A net with Multiple Drivers

A value assigned to \( w \) is first delayed by continuous assignment delay. Before this value appears on \( w \), it is further delayed by 2 ns specified in \( \text{wor} \) declaration.

---

Multiple Drives

<table>
<thead>
<tr>
<th>Name</th>
<th>V_</th>
<th>0s</th>
<th>5ns</th>
<th>10ns</th>
<th>15ns</th>
<th>20ns</th>
<th>25ns</th>
<th>30ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \mu )</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \mu )</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \mu )</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Simulation Run of Assignment Statements
Procedural Assignments

- Procedural assignments in Verilog take place in the initial and always procedural constructs, which are regarded as procedural bodies.
Procedural Assignments

- Procedural Flow Control
- Procedural Blocking Assignments
- Procedural Non-blocking Assignments
- Multiple Assignments
- Procedural Continuous Assignments
- Force and Release

Procedural Flow Control

- Procedural Assignments
- Procedural Flow Control
- Procedural Blocking Assignments
- Procedural Non-blocking Assignments
- Multiple Assignments
- Procedural Continuous Assignments
- Force and Release
Procedural Flow Control

- Statements in a procedural body are executed when program flow reaches them.

- Flow control statements are classified as delay control and event control.

- An event or delay control statement in a procedural body causes program flow to be put on hold temporarily.
A blocking assignment uses a `reg` data type on the left-hand side and an expression on the right-hand side of an equal sign.

The syntax of intra-assignment control constructs is similar to that of procedural flow control statements, but these constructs appear on the right-hand side of an equal sign in a procedural assignment.

```verilog
... #200 a = #100 b;
```

The procedural assignment that is delayed by 200 time units by a delay control statement and by 100 time units by an intra-assignment delay control.
Procedural Blocking Assignments

```verilog
initial begin : Blocking_Assignment_to_b
    b = 1;
    #100
    b = #80 0; b = #120 1;
    #100
    $display ("Initial Block with Blocking Assignment to b Ends at:", $time);
end
```

* Blocking Procedural Assignments

Procedural Non-blocking Assignments

- Procedural Assignments
  - Procedural Flow Control
  - Procedural Non-blocking Assignments
  - Procedural Continuous Assignments
  - Procedural Blocking Assignments
  - Multiple Assignments
  - Force and Release
Procedural Non-blocking Assignments

- A non-blocking assignment uses the left arrow notation <= (left angular bracket followed by the equal sign) instead of the equal sign used in blocking assignments.

- When flow reaches a non-blocking assignment, the right-hand side of the assignment is evaluated and will be scheduled for the left-hand side reg to take place when the intra-assignment control is satisfied.

```verilog
initial begin
    #100
    a = 1;
    a <= #80 0; a <= #120 1;
    $display("Initial Block with Non-blocking Assignment to a Ends at:", $time);
end
```

* Non-blocking Procedural Assignments
Procedural Non-blocking Assignments

* Comparing Blocking and Non-blocking Procedural Assignments

Multiple Assignments

- Procedural Assignments
- Procedural Flow Control
- Procedural Non-blocking Assignments
- Procedural Continuous Assignments
- Force and Release
- Multiple Assignments
Multiple Assignments

* If several assignments appear at the same real time in a procedural body, the last assignment overrides all others.

* If program flow in two procedural bodies reaches assignments to the same reg at exactly the same time, the outcome of the value assigned to the left-hand side of the assignment will not be known.

```verilog
initial begin
  clk = 0;
end

always begin
  clk = ~clk;
  #17;
end
```

* Multiple reg Assignments

This code works properly only if complementing of the clk is delayed until the clk is initialized to 0 in the initial block.
Multiple Assignments

```verilog
initial begin
  clk = 0;
end

always begin
  #0;
  clk = ~clk;
  #17;
end
```

* Multiple `reg` Assignments; Delay Used for Deterministic Results

One way to correct this problem is to delay complementing the clock by one simulation cycle. This can be done by inserting `#0`.

---

Procedural Continuous Assignments

- Procedural Assignments
  - Procedural Flow Control
  - Procedural Non-blocking Assignments
  - Procedural Continuous Assignments
  - Procedural Blocking Assignments
- Multiple Assignments
- Force and Release
Procedural Continuous Assignments

* Using a procedural continuous assignment construct, an assignment to a `reg` type variable can be made to stop all other assignments to this variable from taking place.

```verilog
reg qout;
always @(reset) begin
  if (reset) assign qout <= 0;
  else deassign qout;
end
```

* Unlike `assign` and `deassign`, which apply to `reg` type variables, `force` and `release` constructs apply to net and `reg` types.

```
force qout;
release qout;
```

Procedural Continuous Assignments

```verilog
timescale 1ns/100ps

module FlipflopAssign (input reset, din, clk, output qout);
  reg qout;
  always @(reset) begin
    if (reset) assign qout <= 0;
    else deassign qout;
  end
  always @(posedge clk) begin
    qout <= din;
  end
endmodule
```

* Procedural Continuous Assignments
Force and Release

- Unlike assign and deassign, which apply to reg type variables, force, and release constructs apply to net and reg types.

- Forcing a value on a net overrides all values assigned to the net through continuous assignments or connected to it through gate outputs.
Compiler Directives

- `ifdef`, `else`, `endif`
- `unconnected`, `-drive`
- `celldefine`, `endcelldefine`
- `resetall`
- `timescale`
`timescale

- Including the `timescale 1ns/100 ps directive before a module header causes all time-related numbers to be interpreted as having a 1-ns time unit.

`default-nettype

- Compiler Directives

  `timescale  `default_nettype  `include  `define

  ifndef,  `else,  `endif

  `unconnected  `drive

  `celldefine,  `endcelldefine

  `resetall
`default-nettype

* The default wire type can be changed by the `default_nettype. For example,
  
  `default_nettype wor

  at the beginning of a module causes undeclared nets in constructs such as the terminal list of a module instance to be assumed to be wor type nets.

`include

Compiler Directives

`timescale `default_nettype `include `define

`ifdef, `else, `endif `unconnected _drive `celldesign `endcelldefine `resetall
Because Verilog does not provide a common library of parts and utilities, a shared code must be explicitly inserted in modules that use the code.

Compiler Directives

- `timescale`
- `default_nettype`
- `include`
- `define`
- `ifdef`
- `else`
- `endif`
- `unconnected _drive`
- `celldesign`
- `endcelldesign`
- `resetall`
`define

- `define word_length 32
- `define begin_fetch_state 3'b101

'undef directive undefines a previously defined text macro

`ifdef, `else, `endif

Compiler Directives

- `timescale
- `default_nettype
- `include
- `define

- `ifdef, `else, `endif
- `unconnected_drive
- `celldel, `endcelldefine
- `resetall
`ifdef, `else, `endif

- Because Verilog does not provide a common library of parts and utilities, a shared code must be explicitly inserted in modules that use the code.

- If the next macro has been defined, the group of lines bracketed between `ifdef and `else is compiled.
- If the text macro has not been defined, the group of lines bracketed between `else and `endif is compiled.
Changes port value left open in the connection list of a module instantiation which is assumed to have the default net value.

The only arguments allowed with this directive are pull0 or pull1 for unconnected values 0 and 1, respectively.
`celldefine, `endcelldefine

- The `celldefine and `endcelldefine directives bracket modules that are to be considered as cells.
Using this directive at the beginning of every module guarantees that no previous setting affects compilation of modules and that all defaults are set.
System Tasks and Functions

- The names of system tasks and functions begin with a dollar sign, $, followed by a task specifier.
Display Tasks

- Display tasks include those for monitoring and outputting variable values as they change (the $monitor group of tasks) and those for displaying variables at a selected time (the $display tasks).

- Display tasks can display in binary, hexadecimal, or octal formats. The character b, h, or o at the end of the task name specifies the data type a task handles.
The `fopen` function opens a file and assigns an integer file description. The file descriptor will be used as an argument for all file I/O tasks.

There are string write tasks (`$fwrite`) that write their formatted outputs to a string.

Verilog also provides tasks for inputting data from files or strings. Examples of these tasks are `fgetc`, `fscanf`, and `sscanf` for getting character from file, reading formatted data from file, and reading formatted data from string, respectively.
File I/O Tasks

- Other input tasks exist for reading memory data directly into a declared memory. Examples of such tasks are $read and $readmemh.

- File positioning tasks, $seek and $rewind are available for positioning file pointer for read or write.

Timescale Tasks

System Tasks and Functions

- Display Tasks
- File I/O Tasks
- Timescale Tasks
- Simulation Control Tasks
- Timing Check Tasks
- PLA Modeling Tasks
- Conversion Functions For Reals
- Other Tasks and Functions
Timescale Tasks

- The `printtimescale` task displays the timescale and precision of the module whose hierarchical name is being passed to it as its argument.

- The `timeformat` task formats time for display by file IO and display tasks.

Simulation Control Tasks

- System Tasks and Functions
  - Display Tasks
  - Timescale Tasks
  - Timing Check Tasks
  - Conversion Functions For Reals
- File I/O Tasks
- Simulation Control Tasks
  - Simulation Control Tasks
  - PLA Modeling Tasks
  - Other Tasks and Functions
Simulation Control Tasks

- The $finish task ends the simulation and exits.
- The $stop task suspends the simulation and does not exit the simulation environment.

Timing Check Tasks

System Tasks and Functions

- Display Tasks
- File I/O Tasks
- Timescale Tasks
- Simulation Control Tasks
- Timing Check Tasks
- PLA Modeling Tasks
- Conversion Functions For Reals
- Other Tasks and Functions
Timing Check Tasks

* In general, timing check tasks check the timing on one signal or the relative timing of several signals for certain conditions to hold.

```
$nochange (posedge clock, d_input, 3, 5);
```

Uses the $nochange timing check task to report a violation if $d_input$ changes in the period of 3 time units before and 5 time units after the positive edge of the $clock$.

PLA Modeling Tasks

System Tasks and Functions

- Display Tasks
- File I/O Tasks
- Timescale Tasks
- Simulation Control Tasks
- Timing Check Tasks
- PLA Modeling Tasks
- Conversion Functions For Reals
- Other Tasks and Functions
PLA Modeling Tasks

* The general format that we use for describing these tasks is:
  * $sync\_async$ can be either sync or async
  * $and\_or$ can be and, or, nand, or nor
  * $array\_plane$: in place of array plane, array or plane can be used.

* $async\ $nand\ $array

(bool \_8by4,
{a1, a2, a3, a4, a5, a6, a7, a8},
{b1, b2, b3, b4})

An asynchronous PLA with a nand logical function, a1 to a7 inputs, and b1 to b4 outputs. PLA nand-plane fuses are determined by the contents of the bool\_8by4 declared memory

PLA Modeling Tasks

b1 = ~(a3 & a4 & a8)
b2 = ~(a1 & a2 & a5)
b3 = ~(a5 & a6)
b4 = ~(a2 & a4)

PLA Output Equations

* (a) Contents of bool\_8by4, (b) Corresponding PLA NAND Plane
Conversion Functions for Reals

- Verilog provides four system functions for converting from real to integer or bit, and for converting between bit or integer and real.

- The functions are `$bitstoreal`, `$realtobits`, `$itor`, and `$rtoi`.
- $\texttt{random}$ is a useful function for random data generation.

- There are three time functions, $\texttt{realtime}$, $\texttt{time}$, and $\texttt{stime}$, that return the simulation time in various formats.
Summary

This chapter presented:

- General timing and concurrency concepts that are particular to hardware description languages.
- Utilities found in Verilog for describing hardware and hardware test environments.
- General syntax of the language its operators, names, and data types.
- Simulation of hardware described in Verilog using language constructs and utilities of this language.
- Tasks and compiler directives, that are part of the language utilities for hardware and testbench modeling, but are secondary to those discussed in Section 3.2.
- Most of Verilog without presenting a lot of examples and specific applications of the language constructs.