Verilog HDL:
Language Essentials

Pravin Zode

Outline

- Lexical Conventions
- Comments
- Number Specifications
- Data Types

Lexical Conventions

- Lexical conventions in Verilog HDL are similar to the C programming language.
- Verilog contains a stream of tokens such as:
  - Comments
  - Delimiters
  - Numbers
  - Strings
  - Identifiers
  - Keywords
- Verilog HDL is a case-sensitive language.
- All keywords in Verilog HDL are written in lowercase.

White Space

- Whitespace in Verilog includes
  - Blank spaces (\b)
  - Tabs (\t)
  - Newlines (\n)
- Whitespace is ignored by Verilog except when separating tokens
- Whitespace within strings is not ignored.

Comments

- Comments improve code readability and documentation
- Two types of comments in Verilog:
  - One-line comments: Start with // and extend to the end of the line.
  - Multiple-line comments: Start with /* and end with */
- Multiple-line comments cannot be nested
- One-line comments can be embedded within multiple-line comments

Operators

- Operators in Verilog are classified into three types:
  - Unary operators: Precede the operand
  - Binary operators: Appear between two operands
  - Ternary operators: Use two separate operators to work with three operands

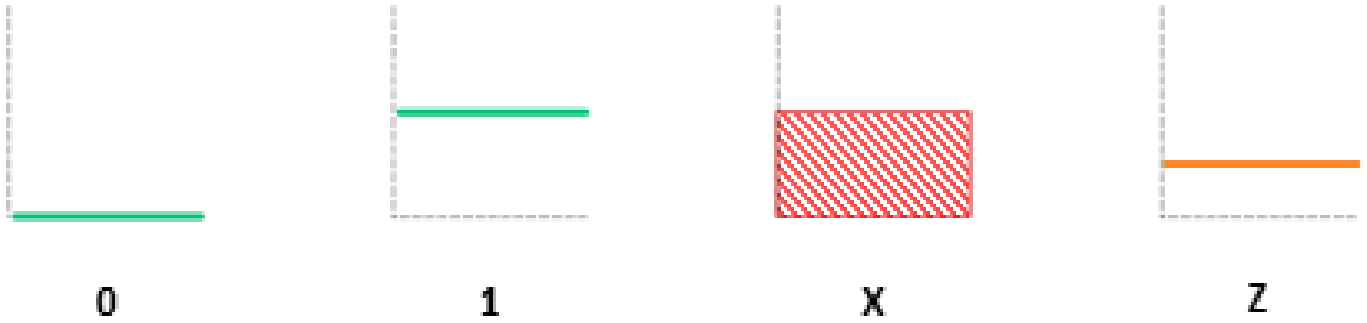
```
a = ~ b; // ~ is a unary operator. b is the operand
a = b && c; // && is a binary operator. b and c are operands
a = b ? c : d; // ?: is a ternary operator. b, c and d are operands
```

Number Specification

- Verilog supports two types of number specification
  - **Sized Numbers**: Size is explicitly specified, e.g., 4'b1010 (4-bit binary), 8'hA3 (8-bit hexadecimal)
  - **Unsized Numbers**: Size is not explicitly mentioned
  - Default size is 32 bits in most cases, e.g. 123 (interpreted as a 32-bit decimal number)

Value Set

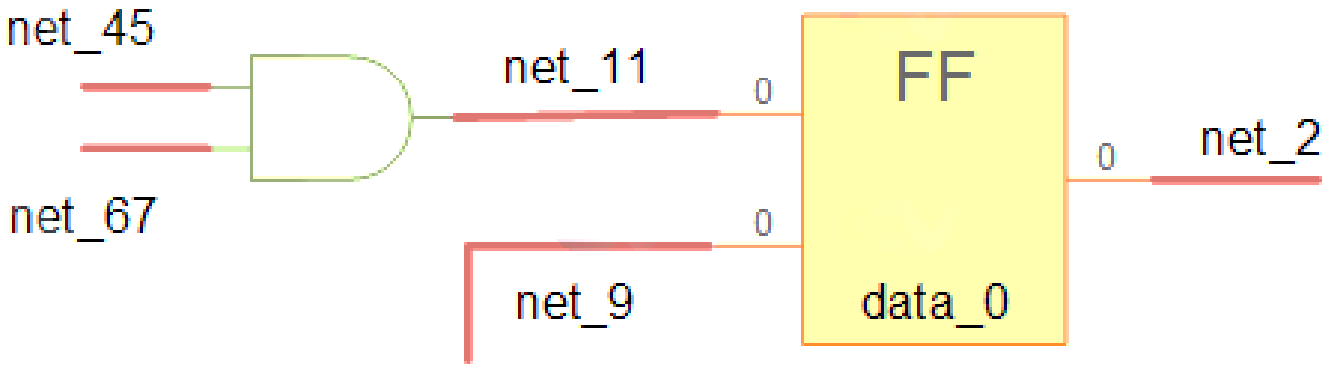
Value	Description
0	A logic zero, or false condition.
1	A logic one, or true condition.
x or X	Unknown or uninitialized.
z or Z	High impedance, tri-stated, or floating.



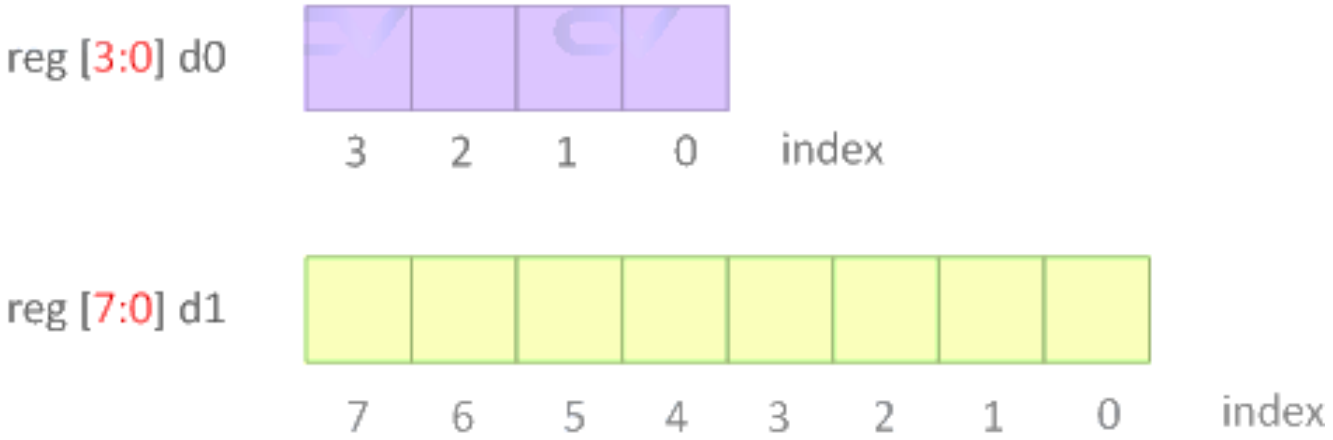
Net Data Type

- Every signal in Verilog must be associated with a data type.
- Verilog defines two primary data types:
  - **Nets** – used to represent physical connections (wires).
  - **Registers** – used to store values (i.e., memory elements).
- Purpose of these data types:
  - Nets connect logical/hardware elements within or across modules.
  - Registers hold values for later use in simulation or logic processing.
- The most common synthesizable net data type in Verilog is wire
- Nets can take values: 0, 1, X and Z.

Net Data Type



Nets /Wire



Reg

Variable Data Types

- Variable data types in Verilog model storage.
- They can take values: 0, 1, X (unknown), and Z (high impedance).
- Unlike nets, variable data types do not have an associated strength.
- They retain their assigned value until the next assignment.

Variable Data Types

Type	Description
reg	A variable that models logic storage. Can take on values 0, 1, X, and Z.
integer	A 32-bit, 2's complement variable representing whole numbers between $2,147,483,648_{10}$ and +2,147,483,647.
real	A 64-bit, floating point variable representing real numbers between $-(2.2 \times 10^{-308})10$ and $+(2.2 \times 10^{308})_{10}$.
time	An unsigned, 64-bit variable taking on values from 0_{10} to $+(9.2 \times 10^{18})$.
realtime	Same as time. Just used for readability.

Vector Data Types

- A vector is a one-dimensional array of elements.
- All net data types can be used to form vectors.
- The variable type reg can also be used to create vectors

<type> [<MSB_index> : <LSB_index>] vector_name

```
wire [7:0] Sum;     // This defines an 8-bit vector called "Sum" of type wire. The
                    // MSB is given the index 7 while the LSB is given the index 0.

reg [15:0] Q;       // This defines a 16-bit vector called "Q" of type reg.
```

Individual bit addressing

```
Sum[0];         // This is the least significant bit of the vector "Sum" defined above.
Q[15:8];        // This is the upper 8-bits of the 16-bit vector "Q" defined above.
```

Arrays

- An array is a multidimensional collection of elements
- It can be thought of as a "vector of vectors.“
- Vectors within the array all have the same dimensions.
- To declare an array:
  - First, define the element type and dimensions. Then, specify the array name and its dimensions.

Arrays

```
<element_type>  [<MSB_index>:<LSB_index>]  array_name  [<array_start_index>:
<array_end_index>];
```

Example:

```
reg[7:0] Mem[0:4095];  // Defines an array of 4096, 8-bit vectors of type reg
integer A[1:100];      // Defines an array of 100 integers.
```

```
Mem[2];          // This is the 3rd element within the array named "Mem".
                 // This syntax represents an 8-bit vector of type reg.

Mem[2][7];       // This is the MSB of the 3rd element within the array named "Mem".
                 // This syntax represents a single bit of type reg.

A[2];            // This is the 2nd element within the array named "A". Recall
                 //  that A was declared with a starting index of 1.
                 // This syntax represents a 32-bit, signed integer.
```

Expressing Number with different bases

Syntax	Description
'b	Unsigned binary.
'o	Unsigned octal.
'd	Unsigned decimal.
'h	Unsigned hexadecimal.
'sb	Signed binary.
'so	Signed octal.
'sd	Signed decimal.
'sh	Signed hexadecimal.

```
<size_in_bits>'<base><value>
```

- If a number is entered in Verilog without specifying its syntax, it is treated as an integer
- Verilog supports different number bases (binary, octal, decimal, hexadecimal)
- The underscore (_) can be inserted between numerals to improve readability.

Expressing Number with different bases

```
10              // This is treated as decimal 10, which is a 32-bit signed vector.
4'b1111         // A 4-bit number with the value 1111₂.
8'b1011_0000    // An 8-bit number with the value 10110000₂.
8'hFF           // An 8-bit number with the value 11111111₂.
8'hff           // An 8-bit number with the value 11111111₂.
6'hA            // A 6-bit number with the value 001010₂. Note that leading zeros
                //  were added to make the value 6-bits.
8'd7            // An 8-bit number with the value 00000111₂.
32'd0           // A 32-bit number with the value 0000_0000₁₆.
'b1111          // A 32-bit number with the value 0000_000F₁₆.
8'bZ            // An 8-bit number with the value ZZZZ_ZZZZ.
```

Assigning Between Different Types

- Verilog is a weakly typed (loosely typed) language, allowing assignments between different data types.
- In contrast, strongly typed languages (e.g., VHDL) only allow assignments between like types.
- **Reason:** Verilog treats all data types as groups of bits, making type conversion flexible.
- When assigning between different types: Verilog automatically truncates or adds leading bits as needed.
- Example scenario: Assume ABC_TB is declared as reg[2:0].

```
ABC_TB = 2'b00;     // ABC_TB will be assigned 3'b000. A leading bit is automatically
                       added.
ABC_TB = 5;         // ABC_TB will be assigned 3'b101.  The integer is truncated to
                       3-bits.
ABC_TB = 8;         // ABC_TB will be assigned 3'b000.  The integer is truncated to
                       3-bits.
```

Operators

- Verilog provides a variety of predefined operators for different operations.
- Operators are designed to work on specific data types (e.g., reg, wire, integer).
- Not all operators are synthesizable (some are for simulation purposes only).
- Understanding which operators are synthesizable is crucial for hardware design.

Operators

Common categories of Verilog operators

- Arithmetic operators (+, -, *, /, %)
- Bitwise operators (&, |, ^, ~)
- Logical operators (&&, ||, !)
- Relational operators (==, !=, >, <, >=, <=)
- Shift operators (<<, >>)
- Reduction operators (&, |, ^, ~&, ~|, ~^)
- Concatenation and replication ({}, {{}})

Assignment Operator

- Assignment Operator (=)
- Used to assign values to signals.
- Left-hand side (LHS) → Target signal.
- Right-hand side (RHS) → Input (can be signals, constants, or expressions).

Example :

```
F1 = A;      // Assign signal A to F1
F2 = 8'hAA;  // Assign 8-bit value 10101010 to F2
```

Continuous Assignment

- Uses the assign keyword for continuous signal assignment
- LHS must be a net type (e.g., wire)
- RHS can contain nets, registers, constants, and operators.
- Models combinational logic: any change in RHS updates LHS immediately.
- Multiple assignments to the same net → Higher drive strength takes priority.

```
assign F1 = A;       // F1 updates whenever A changes
assign F2 = 1'b0;    // F2 is assigned constant 0
assign F3 = 4'hA;    // F3 gets 4-bit hexadecimal value A (1010)
```

Concurrent Execution

- Each assign statement is executed simultaneously
- Synthesized as separate logic circuits
- Unlike traditional programming, Verilog does not execute assignments sequentially

```
assign X = A;
assign Y = B;
assign Z = C;
```

- These assignments occur at the same time, like three separate wires.

Bitwise Logical Operator

Syntax	Operation
~	Negation
&	AND
|	OR
^	XOR
~^ or ^~	XNOR
<<	Logical shift left (fill empty LSB location with zero)
>>	Logical shift right (fill empty MSB location with zero)

```
wire [3:0] A = 4'b1101;
wire [3:0] B = 4'b1010;
wire [3:0] X;

assign X = A & B;  // X = 1000 (Bitwise AND)
assign X = A | B;  // X = 1111 (Bitwise OR)
assign X = A ^ B;  // X = 0111 (Bitwise XOR)
assign X = ~A;     // X = 0010 (Bitwise NOT)
```

Reduction Operator

- Reduction operators perform a logical operation on all bits of a vector.
- The entire vector is treated as multiple inputs to a single logic operation.
- The result is always a single-bit output.

Syntax	Operation
&	AND all bits in the vector together (1-bit result)
~&	NAND all bits in the vector together (1-bit result)
|	OR all bits in the vector together (1-bit result)
~|	NOR all bits in the vector together (1-bit result)
^	XOR all bits in the vector together (1-bit result)
~^ or ^~	XNOR all bits in the vector together (1-bit result)

Boolean Logic Operators

- Boolean logic operators return TRUE (1) or FALSE (0) based on a logical operation.
- Used in decision-making statements such as if, while, and case.
- Operate on single-bit inputs or evaluate entire expressions as Boolean conditions

Syntax	Operation
!	Negation
&&	AND
||	OR

```
!X        // TRUE if all values in X are 0, FALSE otherwise
X && Y    // TRUE if the bitwise AND of X and Y results in all ones, FALSE otherwise
X || Y    // TRUE if the bitwise OR of X and Y results in all ones, FALSE otherwise
```

Relational Operators

- Relational operators compare two values and return TRUE (1) or FALSE (0)
- Used in conditional statements like if, while, and case
- Operate on scalars and vectors, performing bitwise or numerical comparisons.

Syntax	Description
==	Equality
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

Conditional Operator

- The conditional operator (? :) is a shorthand for if-else statements.
- Used for concise and readable assignments in combinational logic.

<target_net> = <Boolean_condition> ? <true_assignment> : <false_assignment>;

```
F = (A == 1'b0) ? 1'b1 : 1'b0;                  // If A is a zero, F=1, otherwise F=0.
                                                //      This models an inverter.

F = (sel == 1'b0) ? A : B;                      // If sel is a zero, F=A, otherwise F=B.
                                                //      This models a selectable switch.

F = ((A == 1'b0) && (B == 1'b0)) ? 1'b'0 :      // Nested conditional statements.
    ((A == 1'b0) && (B == 1'b1)) ? 1'b'1 :      //     This models an XOR gate.
    ((A == 1'b1) && (B == 1'b0)) ? 1'b'1 :
    ((A == 1'b1) && (B == 1'b1)) ? 1'b'0;

F = ( !C && (!A || B) ) ? 1'b1 : 1'b0;          // This models the logic expression
                                                //     F = C'·(A'+B).
```

Concatenation Operator

- Curly brackets {} are used to combine multiple signals into a single vector
- The target signal must have a bit width equal to the sum of the input signal sizes..

```
Bus1[7:0] = {Bus2[7:4], Bus3[3:0]};  // Assuming Bus1, Bus2, and Bus3 are all 8-bit
                                     // vectors, this operation takes the upper
                                        4-bits of
                                     // Bus2, concatenates them with the lower
                                        4-bits of
                                     // Bus3, and assigns the 8-bit combination
                                        to Bus1.

BusC = {BusA, BusB};                 // If BusA and BusB are 4-bits, then BusC
                                     //  must be 8-bits.

BusC[7:0] = {4'b0000, BusA};         // This pads the 4-bit vector BusA with
                                        4x leading
                                     // zeros and assigns to the 8-bit vector BusC.
```

Replication Operator

- The replication operator ({{}}) allows a vector to be repeated multiple times.
- Useful for extending bit-widths and pattern generation
- Replicates a vector multiple times, reducing redundant code
- Commonly used for zero-padding, sign extension, and test pattern generation.
- Can be combined with concatenation for flexible signal assignment

```
{<number_of_replications>{<vector_name_to_be_replicated>}}
```

```
BusX = {4{Bus1}};          // This is equivalent to: BusX = {Bus1, Bus1, Bus1, Bus1};
BusY = {2{A,B}};           // This is equivalent to: BusY = {A, B, A, B};
BusZ = {Bus1, {2{Bus2}}}; // This is equivalent to: BusZ = {Bus1, Bus2, Bus2};
```

Numerical Operator

- Verilog supports several numerical operators for arithmetic operations.
- These operators work on integers, real numbers, and bit-vector data types.

Syntax	Operation
+	Addition
–	Subtraction (when placed between arguments)
–	2's complement negation (when placed in front of an argument)
*	Multiplication
/	Division
%	Modulus
**	Raise to the power
<<<	Shift to the left, fill with zeros
<<<	Shift to the right, fill with sign bit

Operator Precedence

- Operators with higher precedence execute first unless overridden by parentheses
- Always use parentheses () for clarity in complex expressions.
- Assignment (=) has the lowest precedence, ensuring other operations execute before assignment

Operators	Precedence	Notes
! ~ + –	Highest	Bitwise/Unary
{} {{}}		Concatenation/Replication
()	↓	No operation, just parenthesis
**		Power
* / %		Binary Multiply/Divide/Modulo
+ –	↓	Binary Addition/Subtraction
<< >> <<< >>>		Shift Operators
< <= > >=		Greater/Less than Comparisons
== !=	↓	Equality/Inequality Comparisons
& ~&		AND/NAND Operators
^ ~^		XOR/XNOR Operators
| ~|	↓	OR/NOR Operators
&&		Boolean AND
||		Boolean OR
?:	Lowest	Conditional Operator

Thank you !

Happy Learning