

Lexical Conventions

- basic constructs and conventions in Verilog.
- Data types in Verilog model actual data storage and switch elements in hardware very closely.
- Verilog contains a stream of tokens. **Tokens** can be **comments, delimiters, numbers, strings, identifiers, and keywords.**
- Verilog HDL is a case-sensitive language.
- All keywords are in lowercase.
- **Whitespace**
 - Blank spaces (\b) , tabs (\t) and newlines (\n) comprise the whitespace.
 - Whitespace is ignored by Verilog except when it separates tokens.
 - Whitespace is not ignored in strings .
- **Comments**
 - Comments can be inserted in the code for readability and documentation.
 - A one-line comment starts with "//".
 - A multiple-line comment starts with "/*" and ends with "*/".

a = b && c; // This is a one-line comment

/ This is a multiple line comment */*

/ This is /* an illegal */ comment */ /* This is //a legal comment */*

Lexical Conventions – contd.

■ Operators

- Operators are of three types: **unary**, **binary**, and **ternary**.
- **Unary** operators precede the operand.
 - $a = \sim b$; *// \sim is a unary operator. b is the operand*
- **Binary** operators appear between two operands.
 - $a = b \&\& c$; *// $\&\&$ is a binary operator. b and c are operands*
- **Ternary** operators have two separate operators that separate three operands.
 - $a = b ? c : d$; *// $?:$ is a ternary operator. b , c and d are operands*

Lexical Conventions – contd.

■ Number Specification

- There are two types of number specification in Verilog: sized and unsized.

– Sized numbers

- Sized numbers are represented as `<size> '<base format> <number>`.

- `4'b1111`
- `12'habc`
- `16'd255`

– Unsized numbers

- Numbers that are specified without a `<base format>` are by default decimal numbers
- Numbers that are specified without a `<size>` are by default 32-bits

- `23456`
- `'hc3`
- `'o21`

– X or Z values

- X symbol represents unknown value
- Z symbol represents high impedance value

- `12'h13x`
- `6'hx`
- `32'bz`

Lexical Conventions – contd.

■ Number Specification

- *If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z.*

| | | |
|------------------|---|---------|
| Hexadecimal base | An x or z sets four bits for a number | 12'h13x |
| octal base | An x or z sets three bits for a number | 6'hx |
| Binary base | An x or z sets one bit for a number | 32'bz |

– Negative numbers

- Negative numbers can be specified by putting a minus sign before the size for a constant number.
 - -6'd3
 - -6'sd3

– Underscore characters and question marks

- An underscore character "_" is allowed anywhere in a number except the first character. (improves readability)
- A question mark "?" is the Verilog HDL alternative for z in the context of numbers.

Lexical Conventions – contd.

■ Strings

- *A string is a sequence of characters that are enclosed by double quotes.*
- *The restriction on a string is that it must be contained on a single line, that is, without a carriage return.*
- *It cannot be on multiple lines.*
 - `"Hello Verilog World"`
 - `"a / b"`

■ Identifiers and Keywords

- *Keywords are special identifiers reserved to define the language constructs.*
- *Lowercase*
- *Identifiers are names given to objects so that they can be referenced in the design.*
- *Identifiers are made up of alphanumeric characters, the underscore (`_`), or the dollar sign (`$`)*
- *Identifiers are case sensitive. Identifiers start with an alphabetic character or an underscore.*
- *They cannot start with a digit or a \$ sign*
 - `reg value;`
 - `input clk;`

Keywords


| | | | | | |
|-----------|-----------------|---------------|--------------|---------------------|----------|
| always | ifnone | rnmos | end | not | tri |
| and | incdir | rpmos | endcase | notif0 | tri0 |
| assign | include | rtran | endconfig | notif1 | tri1 |
| automatic | initial | rtranif0 | endfunction | or | triand |
| begin | inout | rtranif1 | endgenerate | output | trior |
| buf | input | scalared | endmodule | parameter | trireg |
| bufif0 | instance | showcancelled | endprimitive | pmos | unsigned |
| bufif1 | integer | signed | endspecify | posedge | use |
| case | join | small | endtable | primitive | vectored |
| casex | large | specify | endtask | pull0 | wait |
| casez | liblist | specparam | event | pull1 | wand |
| cell | library | strong0 | for | pulldown | weak0 |
| cmos | localparam | strong1 | force | pullup | weak1 |
| config | macromodule | supply0 | forever | pulsestyle_onevent | while |
| deassign | medium | supply1 | fork | pulsestyle_ondetect | wire |
| default | module | table | function | rcmos | wor |
| defparam | nand | task | generate | real | xnor |
| design | negedge | time | genvar | realtime | xor |
| disable | nmos | tran | highz0 | reg | |
| edge | nor | tranif0 | highz1 | release | |
| else | noshowcancelled | tranif1 | if | repeat | |

Lexical Conventions – Data Types

■ Value Set

- Verilog supports *four values* and *eight strengths* to model the functionality of real hardware.
- In addition to logic values, strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits.

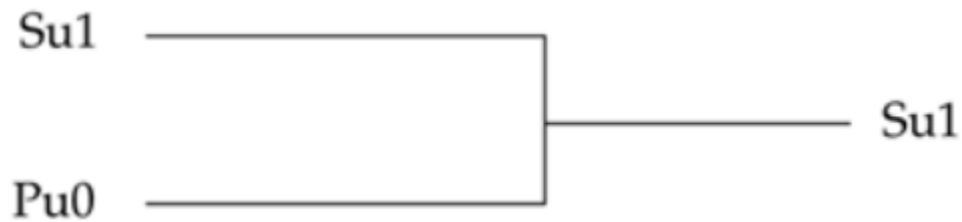
| Value Level | Condition in Hardware Circuits |
|-------------|--------------------------------|
| 0 | Logic zero, false condition |
| 1 | Logic one, true condition |
| x | Unknown logic value |
| z | High impedance, floating state |

| Strength Level | Type | Degree |
|----------------|----------------|--|
| supply | Driving | strongest |
| strong | Driving |  |
| pull | riving | |
| large | Storage | |
| weak | Driving | |
| medium | Storage | |
| small | Storage | |
| highz | High Impedance | weakest |

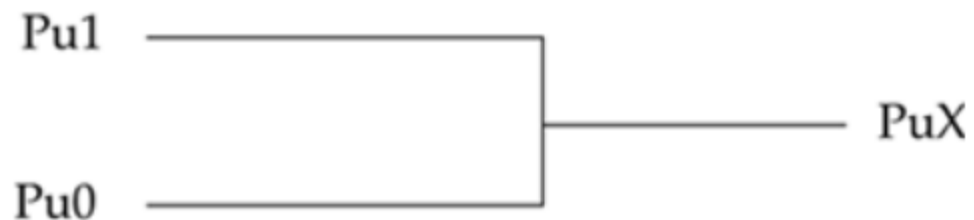
Lexical Conventions – Data Types



- Driving strengths** are used for signal values that are driven on a net.
- Storage strengths** are used to model charge storage in trireg type nets

Multiple Signals with Same Value and Different Strength



Multiple Signals with Opposite Value and Same Strength



| Strength Level | Abbreviation | Degree | Strength Type |
|----------------|--------------|--|----------------|
| supply1 | Su1 | strongest 1 | driving |
| strong1 | St1 |  | driving |
| pull1 | Pu1 | | driving |
| large1 | La1 | | storage |
| weak1 | We1 | | driving |
| medium1 | e1 | | storage |
| small1 | Sm1 | | storage |
| highz1 | HiZ1 | weakest1 | high impedance |
| highz | HiZ0 | weakest0 | high impedance |
| small0 | Sm0 |  | storage |
| medium0 | Me0 | | storage |
| weak0 | We0 | | driving |
| large0 | La0 | | storage |
| pull0 | Pu0 | | driving |
| strong0 | St0 | | driving |
| supply0 | Su0 | strongest0 | driving |

Lexical Conventions – Data Types

■ Nets

- connections between hardware elements
- nets have **values continuously driven** on them by the outputs of devices that they are connected to.
- Nets are declared primarily with the keyword **wire**
- Nets are **one-bit values** by default unless they are declared explicitly as vectors.
- The **default value** of a net is **z**

Examples

```
wire a; // Declare net a for the above circuit  
wire b,c; // Declare two wires b,c for the above circuit  
wire d = 1'b0; // Net d is fixed to logic value 0 at declaration
```

Lexical Conventions – Data Types

■ Registers

- Keyword **reg**
- default value for a **reg** data type is **x**
- a variable that can hold a value
- data storage elements
- retain value until another value is placed onto them.
- Signed registers are declared with keyword **integer**, **reg signed**

Examples

Example of Register

```
reg reset; // declare a variable reset that can hold its value  
initial  
  
begin  
reset = 1'b1; //initialize reset to 1 to reset the digital circuit.  
#100 reset = 1'b0; // after 100 time units reset is deasserted  
end
```

Signed Register Declaration

```
reg signed [63:0] m; // 64 bit signed value  
integer i; // 32 bit signed value
```

Lexical Conventions – Data Types

■ Integer , Real, and Time Register

– Integer

- general purpose register data type
- declared by the keyword **integer**
- integers store values as signed quantities
- default width for an integer is 32-bits

– Real

- Real number constants and real register data types
- keyword **real**
- specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6, which is 3×10^6)
- Real numbers cannot have a range declaration and **default** value is **0**
- a real value is assigned to an integer, the real number is rounded off to the nearest integer.

– Time

- simulation is done with respect to simulation time.
- special time register data type is used in Verilog to store simulation time
- A time variable is declared with the keyword **time**
- The system function **\$time** is invoked to get the **current simulation time**

Examples - integer

integer counter; //general purpose variable
used as a counter.

initial

counter = -1; // A negative one is stored in the
counter

Examples - real

real delta; // Define a real variable called delta

initial

begin

delta = 4e10; // delta is assigned in scientific notation

delta = 2.13; // delta is assigned a value 2.13

end

integer i; // Define an integer i

initial

i = delta; // i gets the value 2 (rounded value of 2.13)

Examples - time

time save_sim_time; // Define a time variable

save_sim_time

initial

save_sim_time = \$time; // Save the current simulation time

Lexical Conventions – Data Types

■ Arrays

- Arrays are allowed in Verilog for **reg**, **integer**, **time**, **real**, **realtime** and **vector register** data types.
- Arrays are accessed by <array_name>[<subscript>]
- Multi-dimensional arrays can also be declared with any number of dimensions

Examples

integer count[0:7]; // An array of 8 count variables

reg bool[31:0]; // Array of 32 one-bit boolean register variables

time chk_point[1:100]; // Array of 100 time checkpoint variables

reg [4:0] port_id[0:7]; // Array of 8 port_ids; each port_id is 5 bits wide

integer matrix[4:0][0:255]; // Two dimensional array of integers

reg [63:0] array_4d [15:0][7:0][7:0][255:0]; //Four dimensional array

wire [7:0] w_array2 [5:0]; // Declare an array of 8 bit vector wire

wire w_array1[7:0][5:0]; // Declare an array of single bit wires

Lexical Conventions – Memories and Parameters

■ Memories

- model register files, RAMs, and ROMs
- Memories are modeled in Verilog simply as a one-dimensional array of registers
- A particular word in memory is obtained by using the address as a memory array subscript

Examples

```
reg mem1bit[0:1023]; // Memory mem1bit with 1K 1-bit words  
reg [7:0] membyte[0:1023]; // Memory membyte with 1K 8-bit words(bytes)  
membyte[511] // Fetches 1 byte word whose address is 511
```

■ Parameters

- constants to be defined in a module by the keyword **parameter**
- Parameters cannot be used as variables
- Parameter values for each module instance can be overridden individually at compile time

Examples

```
parameter port_id = 5; // Defines a constant port_id  
parameter cache_line_width = 256; // Constant defines width of cache line  
parameter signed [15:0] WIDTH; // Fixed sign and range for parameter // WIDTH
```

Lexical Conventions – Strings

■ Strings

- Strings can be stored in reg
- Each character in the string takes up 8 bits (1 byte)
- If the width of the register is greater than the size of the string, Verilog fills bits to the left of the string with zeros.
- If the register width is smaller than the string width, Verilog truncates the leftmost bits of the string.

Examples

```
reg [8*18:1] string_value; // Declare a variable that is 18 bytes wide
```

```
initial
```

```
string_value = "Hello Verilog World"; // String can be stored in variable
```

Operator Types

| Operator Type | Operator Symbol | Operation Performed | Number of Operands |
|---------------|-----------------|-----------------------|--------------------|
| Arithmetic | * | multiply | two |
| | / | divide | two |
| | + | add | two |
| | - | subtract | two |
| | % | modulus | two |
| | ** | power (exponent) | two |
| Logical | ! | logical negation | one |
| | && | logical and | two |
| | | logical or | two |
| Relational | > | greater than | two |
| | < | less than | two |
| | >= | greater than or equal | two |
| | <= | less than or equal | two |
| Equality | == | equality | two |
| | != | inequality | two |
| | === | case equality | two |
| | !== | case inequality | two |

Operator Types

| | | | |
|---------------|----------|------------------------|------------|
| Bitwise | ~ | bitwise negation | one |
| | & | bitwise and | two |
| | | bitwise or | two |
| | ^ | bitwise xor | two |
| | ^~ or ~^ | bitwise xnor | two |
| Reduction | & | reduction and | one |
| | ~& | reduction nand | one |
| | | reduction or | one |
| | ~ | reduction nor | one |
| | ^ | reduction xor | one |
| Shift | ^~ or ~^ | reduction xnor | one |
| | >> | Right shift | Two |
| | << | Left shift | Two |
| | >>> | Arithmetic right shift | Two |
| Concatenation | <<< | Arithmetic left shift | Two |
| | { } | Concatenation | Any number |
| | { { } } | Replication | Any number |
| | ?: | Conditional | Three |

Operator Types – Arithmetic Operators

Two types of arithmetic operators

- Unary Operators (+, -)
 - Has single operand
 - The operators + and - can also work as unary operators.
 - specify the positive or negative sign of the operand.
 - Unary + or ? operators have higher precedence than the binary + or ? operators.

Example

-4 // Negative 4

+5 // Positive 5

- Binary Operators

- Has two operands
- multiply (*)
- divide (/)
- add (+)
- subtract (-)
- power (**)
- modulus (%)

Example

A = 4'b0011; B = 4'b0100; // A and B are register vectors
D = 6; E = 4; F=2 // D and E are integers

A * B // Multiply A and B. Evaluates to 4'b1100

D / E // Divide D by E. Evaluates to 1. Truncates any fractional part.

A + B // Add A and B. Evaluates to 4'b0111

B - A // Subtract A from B. Evaluates to 4'b0001

F = E ** F; //E to the power F, yields 16

13 % 3 // Evaluates to 1

16 % 4 // Evaluates to 0

Operator Types – Logical Operators

Logical operators

- logical-and (&&) – binary operator
- logical-or (||) – binary operator
- logical-not (!) – unary operator
 - Logical operators always evaluate to a 1-bit value
 - 0 (false)
 - 1 (true)
 - X (ambiguous)
 - Logical operators take variables or expressions as operands

Example

// Logical operations

A = 3; B = 0;

A && B // Evaluates to 0. Equivalent to (logical-1 && logical-0)

A || B // Evaluates to 1. Equivalent to (logical-1 || logical-0)

!A // Evaluates to 0. Equivalent to not(logical-1)

!B // Evaluates to 1. Equivalent to not(logical-0)

// Unknowns

A = 2'b0x; B = 2'b10;

A && B // Evaluates to x. Equivalent to (x && logical 1)

// Expressions

(a == 2) && (b == 3) // Evaluates to 1 if both a == 2 and b == 3 are true.

Operator Types – Relational Operators

Relational operators

- greater-than ($>$) – binary operator
- less-than ($<$) – binary operator
- greater-than-or-equal-to ($>=$) – binary operator
- less-than-or-equal-to ($<=$) – binary operator
- If used in expression it returns
 - 0 if expression is false
 - 1 if expression is true
 - X if there are x or z values

Example

A = 4, B = 3

X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx

A $<=$ B // Evaluates to a logical 0

A $>$ B // Evaluates to a logical 1

Y $>=$ X // Evaluates to a logical 1

Y $<$ Z // Evaluates to an x

Equality operators **Operator Types – Equality Operators**

- logical equality (==)
- logical inequality (!=)
- case equality (===)
- case inequality (!==)

| Expression | Description | Possible Logical Value |
|------------|--|------------------------|
| a == b | a equal to b, result unknown if x or z in a or b | 0, 1, x |
| a != b | a not equal to b, result unknown if x or z in a or b | 0, 1, x |
| a === b | a equal to b, including x and z | 0, 1 |
| a !== b | a not equal to b, including x and z | 0, 1 |

Example

A = 4, B = 3

X = 4'b1010, Y = 4'b1101

Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx

A == B // Results in logical 0

X != Y // Results in logical 1

X == Z // Results in x

Z === M // Results in logical 1 (all bits match, including x and z)

Z === N // Results in logical 0 (least significant bit does not match)

M !== N // Results in logical 1

- These operators compare the two operands bit by bit, with zero filling if the operands are of unequal length
- If used in expression it returns
 - 0 if expression is false
 - 1 if expression is true
 - X if there are x or z values

Operator Types – Bitwise Operators

Bitwise operators

- negation (~)
- and(&)
- or (|)
- xor (^)
- xnor (^~, ~^)

| bitwise and | 0 | 1 | x |
|-------------|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | x |
| x | 0 | x | x |

| bitwise xor | 0 | 1 | x |
|-------------|---|---|---|
| 0 | 0 | 1 | x |
| 1 | 1 | 0 | x |
| x | x | x | x |

| bitwise or | 0 | 1 | x |
|------------|---|---|---|
| 0 | 0 | 1 | x |
| 1 | 1 | 1 | 1 |
| x | x | 1 | x |

| bitwise xnor | 0 | 1 | x |
|--------------|---|---|---|
| 0 | 1 | 0 | x |
| 1 | 0 | 1 | x |
| x | x | x | x |

Examples

X = 4'b1010, Y = 4'b1101

Z = 4'b10x1

~X // Negation. Result is 4'b0101

X & Y // Bitwise and. Result is 4'b1000

X | Y // Bitwise or. Result is 4'b1111

X ^ Y // Bitwise xor. Result is 4'b0111

X ^~ Y // Bitwise xnor. Result is 4'b1000

X & Z // Result is 4'b10x0

| bitwise negation | result |
|------------------|--------|
| 0 | 1 |
| 1 | 0 |
| x | x |

Operator Types – Reduction Operators

Reduction operators

- and (&)
- nand (~&)
- or (|)
- nor (~|)
- xor (^)
- xnor (~^, ^~)
 - Reduction operators take **only one operand**.
 - Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result.

Examples

X = 4'b1010

&X //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0

|X //Equivalent to 1 | 0 | 1 | 0. Results in 1'b1

^X //Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0

//A reduction xor or xnor can be used for even or odd parity
//generation of a vector.

Operator Types – Shift Operators

Shift operators

- right shift (>>)
- left shift (<<)
- arithmetic right shift (>>>)
- arithmetic left shift (<<<)

Examples

$X = 4'b1100$

$Y = X \gg 1$; //Y is 4'b0110. Shift right 1 bit. 0 filled in MSB position.

$Y = X \ll 1$; //Y is 4'b1000. Shift left 1 bit. 0 filled in LSB position.

$Y = X \ll 2$; //Y is 4'b0000. Shift left 2 bits.

Operator Types – Concatenation and Replication Operators

Concatenation operator

- concatenation operator ({ , })
 - append multiple operands
 - operands must be sized

Examples

A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110

Y = {B , C} // Result Y is 4'b0010

Y = {A , B , C , D , 3'b001} // Result Y is 11'b10010110001

Y = {A , B[0], C[1]} // Result Y is 3'b101

Replication operator

- Replication operator ({ })
 - Repetitive concatenation of the same number can be expressed by using a replication constant.

Examples

reg A;

reg [1:0] B, C;

reg [2:0] D;

A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;

Y = { 4{A} } // Result Y is 4'b1111

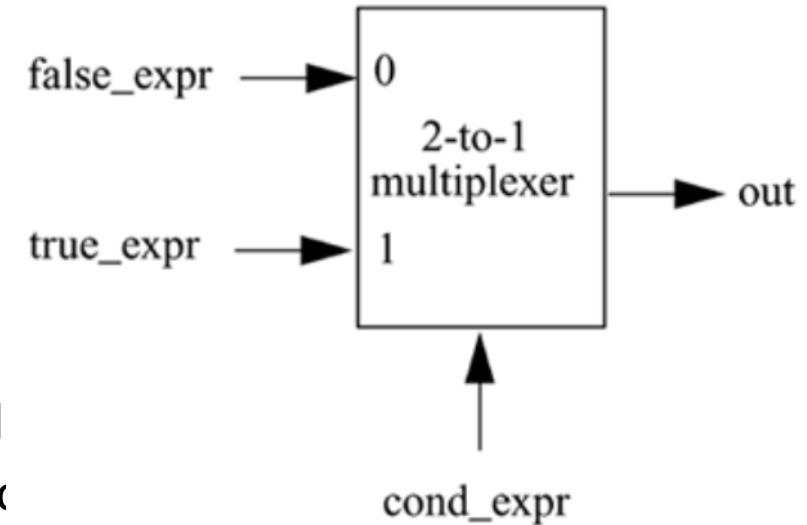
Y = { 4{A} , 2{B} } // Result Y is 8'b11110000

Y = { 4{A} , 2{B} , C } // Result Y is 8'b1111000010

Operator Types – Conditional Operator

Conditional operator

- conditional operator(?:)
 - takes three operands
 - Syntax: `condition_expr ? true_expr : false_expr ;`
 - condition expression (`condition_expr`) is first evaluated
 - If the result is true (logical 1), then the `true_expr` is evaluated
 - If the result is false (logical 0), then the `false_expr` is evaluated



Example

```
//model functionality of a 2-to-1 mux  
assign out = control ? in1 : in0;
```

Operator Precedence

| Operators | Operator Symbols | Precedence |
|---------------------------|-------------------------------|--------------------|
| Unary | + - ! ~ | Highest precedence |
| Multiply, Divide, Modulus | * / % | |
| Add, Subtract | + - | |
| Shift | << >> | |
| Relational | < <= > >= | |
| Equality | == != === !== | |
| Reduction | &, ~& ^ ^~ , ~ | |
| Logical | && | |
| Conditional | ?: | Lowest precedence |

System Tasks

System Tasks

- Verilog provides standard system tasks for certain routine operations
- All system tasks appear in the form **\$<keyword>**
- Operations such as **displaying** on the screen, **monitoring** values of nets, **stopping**, and **finishing** are done by system tasks

Examples

```
$display(p1, p2, p3,....., pn);
$monitor(p1,p2,p3,....,pn);
$stop;
$finish;
```

| Format | Display |
|----------|-----------------------------|
| %d or %D | Display variable in decimal |
| %b or %B | Display variable in binary |
| %s or %S | Display string |
| %h or %H | Display variable in hex |

| | |
|----------|--|
| %c or %C | Display ASCII character |
| %m or %M | Display hierarchical name (no argument required) |
| %v or %V | Display strength |
| %o or %O | Display variable in octal |
| %t or %T | Display in current time format |
| %e or %E | Display real number in scientific format (e.g., 3e10) |
| %f or %F | Display real number in decimal format (e.g., 2.13) |
| %g or %G | Display real number in scientific or decimal, whichever is shorter |

Compiler Directives

Compiler Directives

- All compiler directives are defined by using the **<keyword>**
 - **`define** (directive is used to define text macros in Verilog)
 - **`include** (The **`include** directive allows you to include entire contents of a Verilog source file in another Verilog file during compilation)

Examples – **`define**

```
//define a text macro that defines default word size  
//Used as 'WORD_SIZE in the code  
'define WORD_SIZE 32  
//define an alias. A $stop will be substituted wherever 'S  
appears  
'define S $stop;  
//define a frequently used text string
```

Examples – **`include**

```
// Include the file header.v, which contains declarations in the  
main verilog file design.v.  
'include header.v  
...  
...  
<Verilog code in file design.v>
```