Verilog notes

Tommy Bui

01-29-2023

Contents

1 Verilog Tutorial

Reference to Chipverify

1.1 Lore

In the early days of integrated circuits, engineers had to physically draw transistors & their netlist on paper. As circuits became more complex and larger in scale, this process eventually tedious. Languages such as VHDL & Verilog were developed to simply the process of describing the functionality of IC and let tools convert the behavior into hardware using combinational & sequential logic.

1.2 How is Verilog useful

Verilog creates a level of abstraction that hides the details of its implementation & technology.

E.g. the design of a D flip-flop requires the knowledge of the transistor layout in order to achieve an edge-triggered FF, rise/setup time, fall/clk-Q times to latch value onto flop, etc.

2 Introduction to Verilog

Source

A digital element such as a FF can be represented using combinational gates such as NAND or NOR gates. The functionality of a FF is determined based on the layout of such gates. How the gates have to be connected is usually determined using K-maps

Below is an schemetic of a Data flip-flop & its corresponding truth table. The output, q is asserted only when rstn & d are both set.

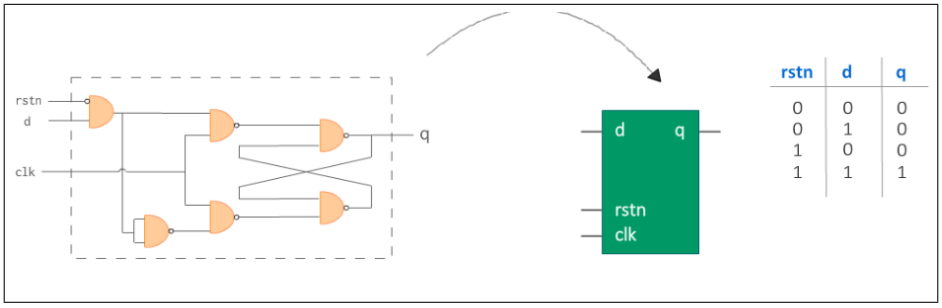


Figure 1: D flip flop

2.1 What is a hardware schematic?

A hardware schematic is a daigram that shows how the combinational gates should be connected to implemented a particular behaviour in hardware. From figure 1, a set of NAND gates are connected in order to create a D flip flop.

2.2 What is a Hardware Description Language?

It's easier to describe how a block of logic should behave & let software tools convert that behavior into an actual hardware scchematic. The langugage that describes the hardware functinality is classified as a Hardware Description Language.

2.3 Sections of Verilog Code

All behavior code should be described within the keywords module & endmodule.

2.3.1 Verilog section template

- Module definition & port list declaration
- List of input & output ports
- Declaration of Verilog data types
- Module instantiations
- Behavioural code

```
1  module [design_name] ( [port_list] );
2
3      [list_of_input_ports]
4      [list_of_output_ports]
5
6      [declaration_of_other_signals]
7
8      [other_module_instantiations_if_required]
9
10     [behavioral_code_for_this_module]
11 endmodule
```

Figure 2: Verilog example Template

3 Data Types

3.1 Verilog Syntax

Source (Lexical item. In lexicography [citation needed], a lexical item is a single word, a part of a word, or a chain of words (catena) that forms the basic elements of a language's lexicon (vocabulary)).

Lexical conventions in Verilog are similar to C in the sense that it contains a sense of tokens. A lexical token may consist of one or more characters and tokens can be comments, keywords, numbers, strings or white space. However, all lines are terminated by a semi-colon.

Verilog is case-sensitive, so variables, var_a & var_A differ.
Two types of comments:

- Single-line comment uses two forward slashes (e.g. //). Single line comments can be nested in a multiple line comment.
- Multiple-line comment starts with /* and ends with */ and cannot be nested

3.1.1 White space

White space refers to spaces, tabs, newlines, & formfeeds, and is usually ignored by Verilog except when it separates tokens. Be sure to take advantage of this when making readable code.

3.1.2 Operators

There are three types of operators: unary, binary, & ternary or conditional.

- Unary operators shall appear to the left of their operand
- Binary operators shall appear between their operands
- Conditional operators have 2 separate operates that separate 3 operands

```
1  x = ~y;              // ~ is a unary operator, and y is the operand
2  x = y | z;           // | is a binary operator, where y and z are its operands
3  x = (y > 5) ? w : z;  // ?: is a ternary operator, and the expression (y>5), w and z are its operands
```

Figure 3: Example of unary, binary, and ternary/conditional operators

If the expression (y¿5) is true, then variable x will get the value w, else y=z.

3.1.3 Number Format

```
16              // Number 16 in decimal
0x10            // Number 16 in hexadecimal
10000           // Number 16 in binary
20              // Number 16 in octal
```

Figure 4: number formats

By default, Verilog simulators treat numbers as decimals. In order to represent them in different radixes, there are certain rules which must be used.

Sized

Sized numbers are represented as show below, where size is written only in decimal to specify the number of bits in the number:

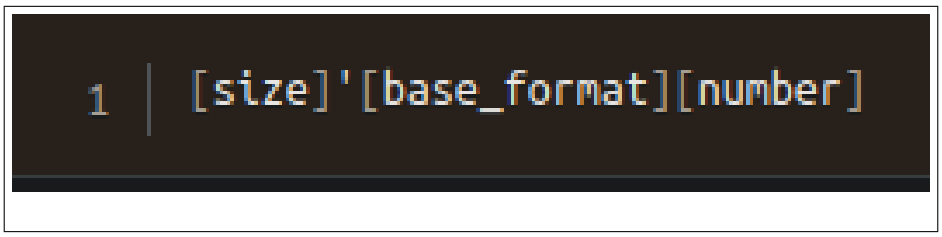


Figure 5: Template for sized numbers

- base_format can be decimal('d or 'D), hex('h or 'H), or octal('o or 'O) & specifies the base of the number
- number can be specified as any valid digit with respect to the specified based format

```
3'b010;      // size is 3, base format is binary ('b), and the number is 010 (indicates val
ue 2 in binary)
3'd2;        // size is 3, base format is decimal ('d) and the number is 2 (specified in de
cimals)
8'h70;       // size is 8, base format is hexadecimal ('h) and the number is 0x70 (in hex)
to represent decimal 112
9'h1FA;      // size is 9, base format is hexadecimal ('h) and the number is 0x1FA (in hex)
to represent decimal 506

4'hA = 4'd10 = 4'b1010 = 4'o12 // Decimal 10 can be represented in any of the four formats
8'd234 = 8'D234                  // Legal to use either lower case or upper case for base
format
32'hFACE_47B2;                   // Underscore (_) can be used to separate 16 bit numbers
for readability
```

Figure 6: ToDo

3.1.4 Unsized

Numbers without a size format specification have a default number of bits depending on the simulator & machine.

3.1.5 Negative

Negative numbers are specified with the - sign before the size of a number; It is illegal to have the - between base format & number.

```
1 integer a = 5423;       // base format is not specified, a gets a decimal value of 5423
2 integer a = 'h1AD7;     // size is not specified, because a is int (32 bits) value stored in a = 32'h0000_1AD7
```

Figure 7: ToDo

3.1.6 Strings

A sequence of characters enclosed in double quotes are strings. It cannot be split into multiple lines & every chracter in the string takes 1B to be stored:

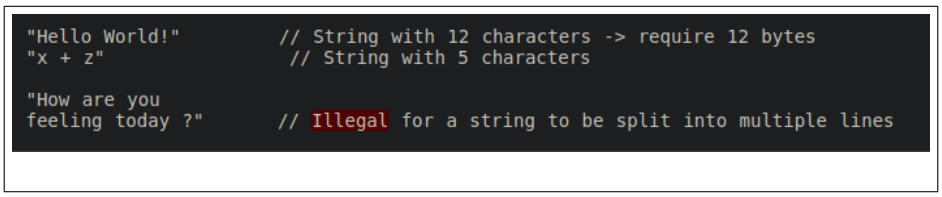


Figure 8: ToDo

3.1.7 Identifiers

Identifiers are names of variables; Can be made of alphanumeric characters and symbols. They cannot start with a digit nor a $:

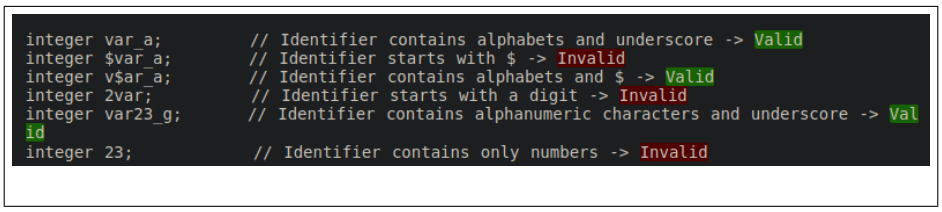


Figure 9: ToDo

3.1.8 Keywords

Keywords are special identifiers reserved to be define the language constructs & are lower case. A list of important keywords is as follows:

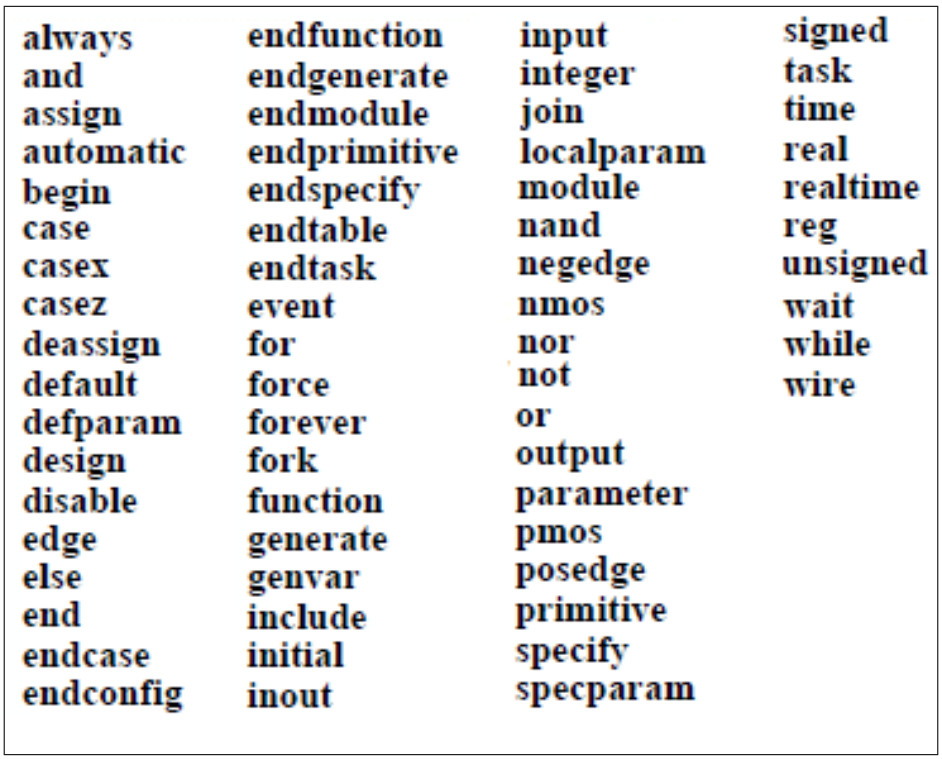


Figure 10: ToDo

3.1.9 Verilog Revisions

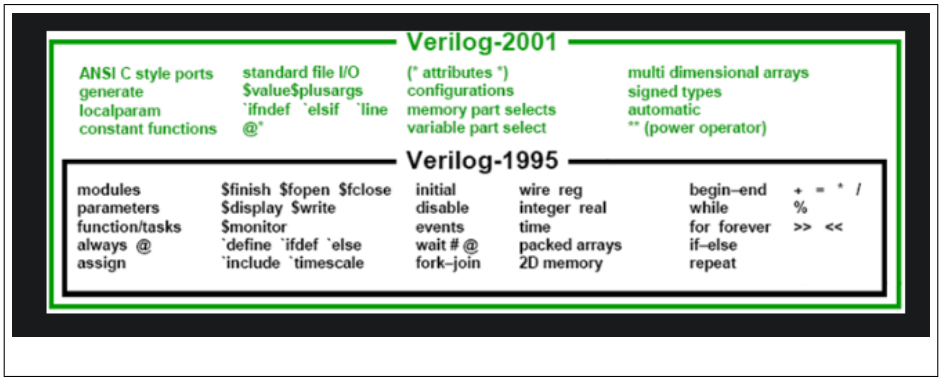


Figure 11: ToDo

3.2 Verilog Data Types

Source

Data-types in Verilog is meant to represent data storage elements like bits in a flip-flop & transmission elements like wires that connect between logic gates & sequential structures.

3.2.1 What values do variables hold?

Almost all data-types can have one of the four different values as given below except for real & event data types.

0	Represents logic zero or false condition
1	Refresents logic one or a true condition
x	Represents an unknown logic value (could be 0 or 1, metastability)
z	Represents high impedance state

The following image shows how these values are represented in timing diagrams & simulation waveforms. Most simulators use this convention where red stands for X and orange in the middle stands for high-impedance or Z.
:w

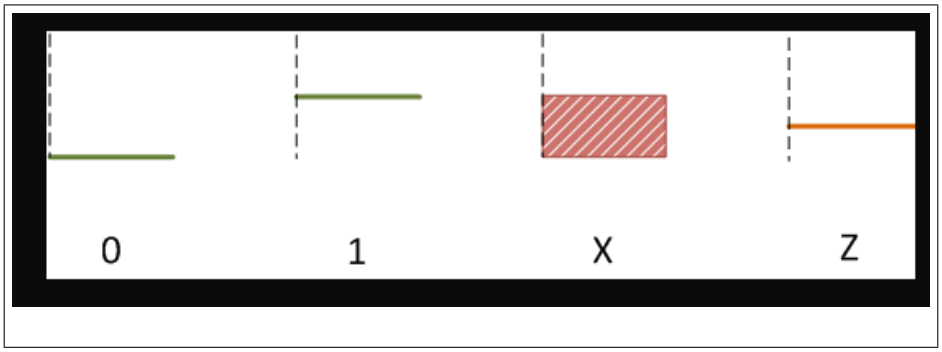


Figure 12: ToDo

3.2.2 What does the verilog value-set imply?

Since Verilog is essentially used to describe hardware elements like flip-flops & combinational logic like NAND & NOR, it has to model the value system found in hardware. A logic one would represent the voltage supply V_{dd}.

X or x means that the value is unknown at the time and could either be 1 or 0. This is an issue known as metastability. This differs from the boolean logic X, where it signifies don't care.

As with any incomplete eletric circuit, the wire that is not connected to anything will have high-impedance at that node which is represented by Z or z.

3.2.3 Nets & Variables

Nets & variables are the two main groups of data types which represent different hardware structures & differ in the way they are assigned & retain values.

Nets

Nets are used to connect between hardware entities like logic gates & don't store any values on their own. In the follow image, a net_11 connects between the output of the AND gate and the first input of the flip_floped called data_0. Similarly, the inputs of the AND gate are connected to nets, net_45 & net_67.

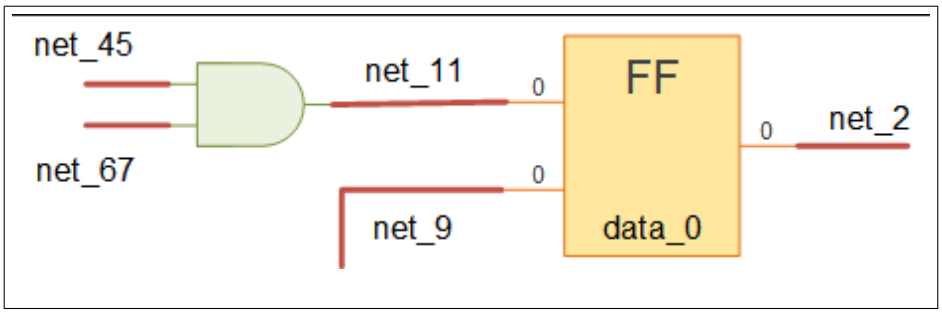


Figure 13: ToDo

There are different types of nets with various characterisitcs, the most commonly used net in digital design is of type wire. Wire is a Verilog data-type used to connect elements & to connect nets that are driven by a single gate or continuous assignment.

When there is a requirement for multiple nets, they can be bunched together to form a single wire. In the code below, we can have a 4-bit wire that sends 4 separate values on each one of these wires. A group of such entities is consider a vector.

$$wire[3:0]n0; //4-bitwire; exampleofavector$$

It is illegal to redeclare a variable name already declared by a net, parameter, or variable.

Variables

A variable is an abstraction of a data storage element & can hold values e.g. A flipflop.

Verilog datatype, reg can be used to model hardware registers since it can hold values between assignments. Note that a reg does not always represent a flipflop as it can also represent combinational logic.

3.2.4 Other DataTypes

Integer

Properties of datatypes in SV (hopefully same in Verilog)

An integer is a general purpose variable of 32-bits wide which is signed that can be used for other purposes while modeling hardware & stores integer values.

Time

Time is an unsigned & 64bits wide & can be used to store simulation time quantities for debugging purposes. A realtime variable simply stores time as a floating point quantity.

Real

A real variable can store floating point values & can be assigned the same way as integer & reg (See Don Mills' book for properties).

3.2.5 Verilog Strings

Strings are stored in reg, and the width of the reg variable has to be large enough to hold the string. Each character in a string represents an ASCII value & requires 1 byte. If the size of the variable is smaller than the string, then Verilog truncates teh leftmost bits of the string. If the size of the variable is larger then Verilog appends zeros to the beginning of the string.

```
// "Hello World" requires 11 bytes

reg [8*11:1] str = "Hello World";         // Variable can store 11 bytes, str = "Hello World"
reg [8*5:1]  str = "Hello World";         // Variable stores only 5 bytes (rest is truncated), str = "World"
reg [8*20:1] str = "Hello World";         // Variable can store 20 bytes (rest is padded with zeros), str = "
```

Figure 14: ToDo

3.3 Verilog Scalar & Vector

Reference A single bit can be represented as a flip-flop. However, a 16-bit sequential element is a register that can hold 16 bits, thus Verilog uses scalar & vector net and variables.

3.3.1 Scalar & Vector

A net or reg declaration without a range specification is considered a 1-bit wide scalar. If a range is specified, then the net or reg becomes a multibit entity known as a vector.

The most significant bit of the vector should be specified as the left hand value in the range while the least significant bit of the vector should be specified on the right.

```
1  wire  [msb:lsb]   name;
2  integer           my_msb;
3
4  wire [15:0]        priority;      // msb = 15, lsb = 0
5  wire [my_msb: 2]   prior;         // illegal
```

Figure 15: ToDo

The msb & lsb should be a constant expression and cannot be substituted by a variable (if you need to vary the width of vector, consider using parameters). But they can be any integer value - positive, negative, or zero; and the lsb value can be greater than, equal to, or less than the MSB.

3.3.2 Part-selects

A range of contiguous bits can be selected through part-select. There are two types of part-selects, one with a constant part-select and another with an indexed part-select.

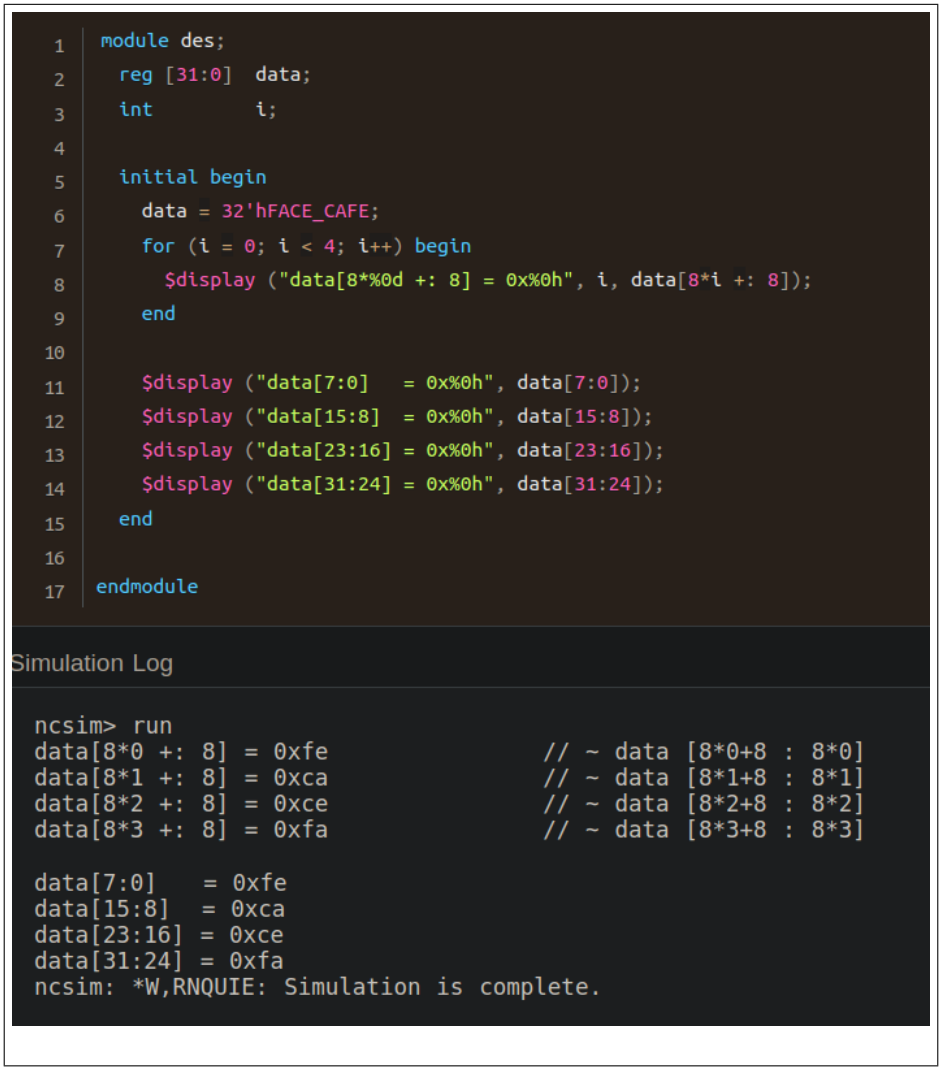
```
1  reg [31:0]    addr;
2
3  addr [23:16] = 8'h23;        // bits 23 to 16 will be replaced by the new value 'h23 -> constant part-select
```

Figure 16: ToDo

Using a variable part-select makes it effect to index parts of a vector in loops. Although the starting bit can be varied, the width has to be constant:

```
[<start_bit> +: <width>]      // part-select increments from start-bit
[<start_bit> -: <width>]      // part-select decrements from start-bit
```

Figure 17: ToDo



```
module des;
  reg [31:0]  data;
  int         i;

  initial begin
    data = 32'hFACE_CAFE;
    for (i = 0; i < 4; i++) begin
      $display ("data[8*%0d +: 8] = 0x%0h", i, data[8*i +: 8]);
    end

    $display ("data[7:0]   = 0x%0h", data[7:0]);
    $display ("data[15:8]  = 0x%0h", data[15:8]);
    $display ("data[23:16] = 0x%0h", data[23:16]);
    $display ("data[31:24] = 0x%0h", data[31:24]);
  end

endmodule
```

Simulation Log

```
ncsim> run
data[8*0 +: 8] = 0xfe                // ~ data [8*0+8 : 8*0]
data[8*1 +: 8] = 0xca                // ~ data [8*1+8 : 8*1]
data[8*2 +: 8] = 0xce                // ~ data [8*2+8 : 8*2]
data[8*3 +: 8] = 0xfa                // ~ data [8*3+8 : 8*3]

data[7:0]   = 0xfe
data[15:8]  = 0xca
data[23:16] = 0xce
data[31:24] = 0xfa
ncsim: *W,RNQUIE: Simulation is complete.
```

Figure 18: ToDo

3.4 Verilog Arrays & Memories

An array declaration of a net or variable can either be scalar or a vector. Any number of dimensions can be created by specifying an address range after the identifier name. Verilog allows the creation of arrays using reg, wire, integer, and real data types.

```
reg        y1 [11:0];        // y is an scalar reg array of depth=12, each 1-bit wide
wire [0:7] y2 [3:0]          // y is an 8-bit vector net with a depth of 4
reg  [7:0] y3 [0:1][0:3];    // y is a 2D array rows=2,cols=4 each 8-bit wide
```

Figure 19: ToDo

An index for every dimension has to be specified to access a particular element of an array & can be an expression of other variables. An array can be formed for any of the different data-types supported in Verilog (note that a memory of N wide 1-bit regs is not the same as an n-bit vector reg).

Assignment

```
y1 = 0;                        // Illegal - All elements can't be assigned in a single go

y2[0] = 8'ha2;          // Assign 0xa2 to index=0
y2[2] = 8'h1c;          // Assign 0x1c to index=2
y3[1][2] = 8'hdd;   // Assign 0xdd to rows=1 cols=2
y3[0][0] = 8'haa;   // Assign 0xaa to rows=0 cols=0
```

Figure 20: ToDo

3.4.1 Example

mem1 is an 8-bit vector, mem2 is an 8-bit array with a depth of 4 (specified by range of [0:3]) & mem3 is a 16-bit vector 2D array with 4 rows & 2 columns:

```
1  module des ();
2    reg [7:0]  mem1;                          // reg vector 8-bit wide
3    reg [7:0]  mem2 [0:3];                 // 8-bit wide vector array with depth=4
4    reg [15:0] mem3 [0:3][0:1];   // 16-bit wide vector 2D array with rows=4,cols=2
5
6    initial begin
7      int i;
8
9      mem1 = 8'ha9;
10     $display ("mem1 = 0x%0h", mem1);
11
12     mem2[0] = 8'haa;
13     mem2[1] = 8'hbb;
14     mem2[2] = 8'hcc;
15     mem2[3] = 8'hdd;
16     for(i = 0; i < 4; i = i+1) begin
17       $display("mem2[%0d] = 0x%0h", i, mem2[i]);
18     end
19
20     for(int i = 0; i < 4; i += 1) begin
21       for(int j = 0; j < 2; j += 1) begin
22         mem3[i][j] = i + j;
23         $display("mem3[%0d][%0d] = 0x%0h", i, j, mem3[i][j]);
24       end
25     end
26   end
27 endmodule
```

Simulation Log

```
ncsim> run
mem1 = 0xa9
mem2[0] = 0xaa
mem2[1] = 0xbb
mem2[2] = 0xcc
mem2[3] = 0xdd
mem3[0][0] = 0x0
mem3[0][1] = 0x1
mem3[1][0] = 0x1
mem3[1][1] = 0x2
mem3[2][0] = 0x2
mem3[2][1] = 0x3
mem3[3][0] = 0x3
mem3[3][1] = 0x4
ncsim: *W,RNQUIE: Simulation is complete.
```

Figure 21: ToDo

Notice how the inner loop completes before the outer loop.

3.4.2 Memories

Memories are digital storage elements that help store data & information in digital circuits. RAMs & ROMs are good examples of such memory elements. Storage elements can be modeled using one-dimensional arrays of type reg and is called a memory. Each element in the memory may represent a word and is referenced using a single array index.

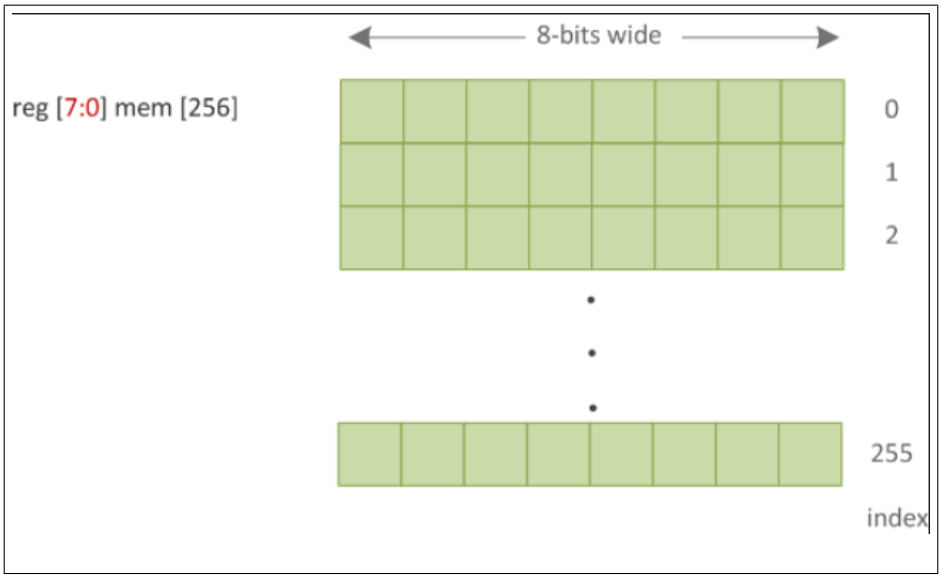


Figure 22: ToDo

3.4.3 Register Vector

Verilog vectors are declared using a size range on the left side of the variable name & these get realized into flops that match the size of the variable. In the code show below, the design module accepts a clock, reset, and some other input signals to RD/WR data into a block.

It contains a 16-bit storage element called register, which simply gets updated during writes and returns the current value during reads. The register is written when sel & wr are high on the same clock edge. It returns the current data when sel is high & wr is low:

```
1  module des (    input          clk,
2                  input          rstn,
3                  input          wr,
4                  input          sel,
5                  input [15:0]   wdata,
6                  output [15:0]  rdata);
7
8      reg [15:0] register;
9
10     always @ (posedge clk) begin
11     if (!rstn)
12         register <= 0;
13     else begin
14         if (sel & wr)
15         register <= wdata;
16         else
17         register <= register;
18     end
19     end
20
21     assign rdata = (sel & ~wr) ? register : 0;
22 endmodule
```

Figure 23: ToDo

The hardware schematic shows that a 16-bit flop is updated when control logic for writes are active & the current value is returned when control logic is configured for reads:

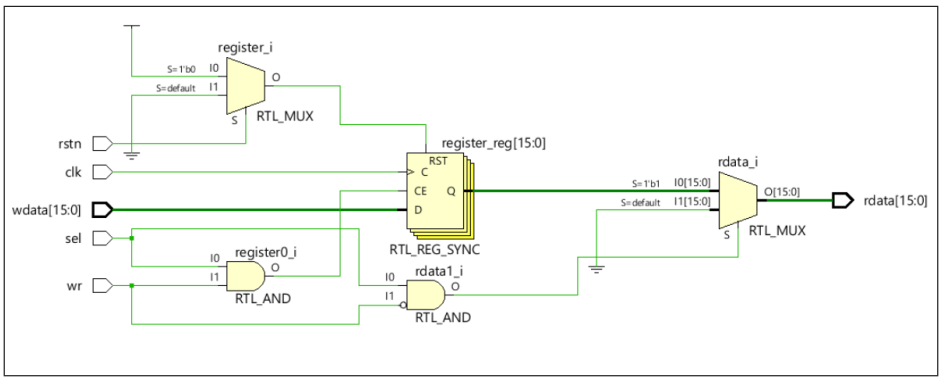


Figure 24: ToDo

3.4.4 Array

In this example, register is an array that has four locations with each having a width of 16-bits. The design module accepts an additional input signal which is called addr to access a particular index in the array.

```
1  module des (    input          clk,
2                  input          rstn,
3                  input  [1:0]   addr,
4                  input          wr,
5                  input          sel,
6                  input [15:0]   wdata,
7                  output [15:0]  rdata);
8
9  reg [15:0] register [0:3];
10 integer i;
11
12 always @ (posedge clk) begin
13     if (!rstn) begin
14         for (i = 0; i < 4; i = i+1) begin
15             register[i] <= 0;
16         end
17     end else begin
18         if (sel & wr)
19             register[addr] <= wdata;
20         else
21             register[addr] <= register[addr];
22     end
23 end
24
25 assign rdata = (sel & ~wr) ? register[addr] : 0;
26 endmodule
```

Figure 25: ToDo

It can be seen in the hardware schematic that each index of the array is a 16-bit flop and the input address is used to access a particular set of flops.

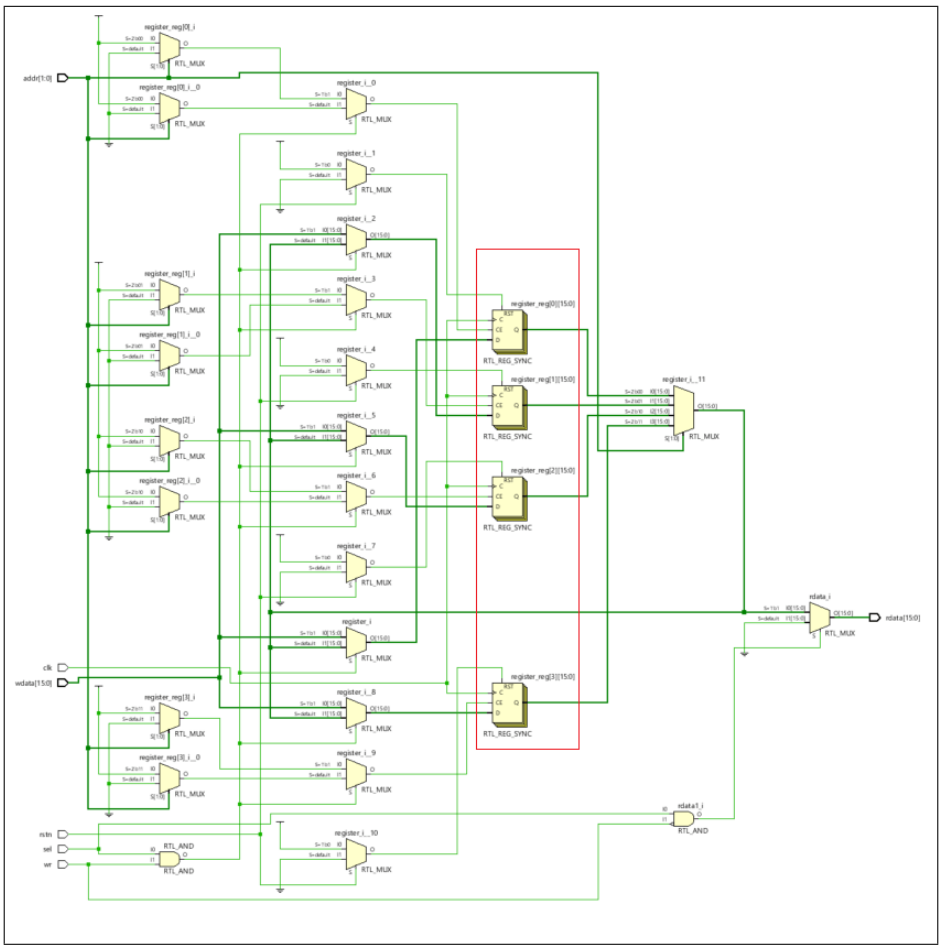


Figure 26: ToDo

4 Building Blocks

4.1 Verilog Module

A module is a block of Verilog code that implements a certain functionality. Modules can be embedded within other modules and a higher level module can communicate with its lower level modules using their input & output ports.

A module should be enclosed within module & endmodule keywords. The name of the module should be given after the module keyword as well as a list of optional ports can be declared.

4.1.1 Example

The module dff represents a D flip-flop which has 3 input ports d, clk, and rstn and one output port, q. The contents within the module block describe how a D flip flop should behave for different combinations of inputs. Here, input d is always assigned to output q at posedge of clk if rstn is high.

```
1  // Module called "dff" has 3 inputs and 1 output port
2  module dff (    input           d,
3                              input           clk,
4                              input           rstn,
5                              output reg  q);
6
7      // Contents of the module
8      always @ (posedge clk) begin
9          if (!rstn)
10             q <= 0;
11         else
12             q <= d;
13     end
14 endmodule
```

Figure 27: ToDo

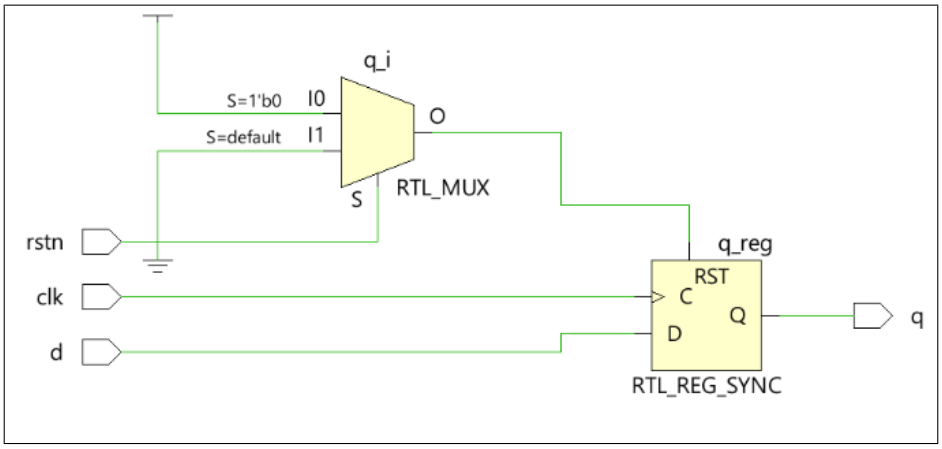


Figure 28: ToDo

4.1.2 What is the purpose of a module?

A module represents a design unit that implements certain behavioral characteristics and will get converted into a digital ciruit during synthesis. Any combination of inputs can be given to the module and it will providing the corresponding output. This allows such a module to be reused to form bigger modules that implement more complex designs.

4.2 Verilog Ports

Source

Ports are a set of signals that define the direction such signals with respect to the module.

4.2.1 Types of Ports

Port	Description
Input	The design module recieves values using its input port(s)
Output	The design module sends values using its output port(s)
Inout	The design module can either send or recieve values using its inout port(s)

Ports by default are considered as nets of type wire.

4.2.2 Signed Ports

The signed attribute can be attached to a port declaration or a net/reg declaration or both. Implicit nets are defaulted to unsigned.

4.3 Port Variations

Veriog 1995

Module declaraions had to frst list the names of its ports withinbrackets and then specify the direction of those ports within the body of the module.

```
module test (a, b, c);

    input   [7:0] a;            // inputs "a" and "b" are wires
    input   [7:0] b;
    output  [7:0] c;            // output "c" by default is a wire

    // Still, you can declare them again as wires to avoid confusion
    wire    [7:0] a;
    wire    [7:0] b;
    wire    [7:0] c;
endmodule


module test (a, b, c);

    input  [7:0] a, b;
    output [7:0] c;           // By default c is of type wire

    // port "c" is changed to a reg type
    reg    [7:0] c;
endmodule
```

Figure 29: ToDo

Veriog 2001

ANSI-C style port naming was introduced in 2001 and allowed the direction to be specified inside the port list.

If a port declaration is declared as a net or variabled type, then it is illegal to redeclare the same port with another declaration.

```
module test ( input      [7:0] a,       // a, e are implicitly declared of type wire
              output reg [7:0] e );

  wire signed [7:0] a;     // illegal - declaration of a is already complete -> simulator dependent
  wire        [7:0] e;     // illegal - declaration of e is already complete

  // Rest of the design code
endmodule
```

Figure 30: ToDo

If the port declaration does not include a net or variable type, then the port

can be declared in a net or variable type declaration again.

```
1  module test ( input      [7:0] a,
2                output     [7:0] e);
3
4      reg [7:0] e;              // Okay - net_type was not declared before
5
6      // Rest of the design code
7  endmodule
```

Figure 31: ToDo