Advanced Computer Architecture Laboratory
School of Electrical Engineering, Tel Aviv University

# Verilog Tutorial

This document briefly describes Verilog language key principles and syntax.

## Contents

## 1. Introduction

Nowadays, digital hardware is designed similarly to software – by coding. Hardware Description Languages (HDL) such as VHDL, Verilog, System-Verilog etc. provide the ability to design sophisticated digital circuits without explicitly defining gates and wires. This high-level approach is often referred to as Register Transfer Level (RTL) design, which describes only which data travels from which register and when it happens. Again, in RTL design no explicit circuitry is described by the designer. In our lab, we will use Verilog as our primary HDL.

**Although Verilog might look like a C programming language due to a syntactic similarity, it is not a programming language. Verilog code describes a hardware module, hence when this module is instantiated, all its code lines are continuously evaluated in parallel. This is different from programing languages which are executed sequentially.**

Verilog code is not intended to run like a main() function of the C code, but to be synthesized and to “come to life” as a circuit. In our lab we will synthesize Verilog code and place it on an FPGA for it to become alive. However, synthesis and placement are lengthy processes, hence **behavioral simulations** can be performed to verify a correctness of a Verilog code. A simulation is a way of bringing your Verilog code to life in a virtual reality, and we will study how to design test-benches (aka scenarios) for simulations. Test-benches are Verilog code segments which resemble the sequential C code the most.

## 2. General syntax remarks

Verilog's syntax resembles the one of C programming language.

- A scope of a block of code is enclosed by the saved words "begin" and "end" instead of the { } braces of the C language. However, if only one command appears in the block of code, no begin-end required (similar to C).
- The comments are marked by **//single line comment** and /* **multi line comment */**
- Each command must be terminated by a semicolon **;**
- The names and saved words are **case sensitive**
- The Verilog source files are normally saved with **.v** extension. Header files have **.vh** extension.
- There are data types called integers, which are purely virtual. Their counterparts will be wires and regs which are synthesizable.

## 3. Hardware Data Types

Verilog provides two data types for hardware: **wire** and **reg**. For example:

`reg  r1;` – r1 keeps a value, which can be changed on another assignment
`wire w1;` – w1 is a net, which must be constantly driven by a signal

One can define wide registers and buses by specifying the MSB (leftmost) and LSB (rightmost) indices:

```
reg  [3:0] r1;
wire [3:0] w1;
```

Signed busses and multiple bit registers (two's complement)

```
reg  signed [3:0] r1
wire signed [3:0] w1
```

In the following sections, you will see that there some places where only wires can be used (continuous assignments) and there are places where only regs can be used (always@ blocks).

## 4. Virtual Data Types

Verilog also provides data types which will not constitute a part of the circuit, but can assist in managing some intermediate (temporary) computations.

- `integer` – 32-bit two's complement integer value
- `real` – 64-bit (double precision) IEEE floating point value
- `realtime` – used for storing time as floating point
- `time` – 64 bit unsigned value for the simulation time counting
- `genvar` - positive integer values. The index variable used in a generate loop must be declared as a genvar

## 5. Operators

| Verilog Operator | Name | Functional Group |
|---|---|---|
| ! | logical negation | logical |
| ~ | negation | bit-wise |
| & | reduction AND | reduction |
| \| | reduction OR | reduction |
| ~& | reduction NAND | reduction |
| ~\| | reduction NOR | reduction |
| ^ | reduction XOR | reduction |
| ~^ or ^~ | reduction XNOR | reduction |
| + | unary (sign) plus | arithmetic |
| - | unary (sign) minus | arithmetic |
| { } | concatenation | concatenation |
| {{ }} | replication | replication |
| * | multiply | arithmetic |
| / | divide | arithmetic |
| % | modulus | arithmetic |
| + | binary plus | arithmetic |
| - | binary minus | arithmetic |
| << | shift left | shift |
| >> | shift right | shift |
| > | greater than | relational |
| >= | greater than or equal to | relational |
| < | less than | relational |
| <= | less than or equal to | relational |
| == | case equality | equality |
| != | case inequality | equality |
| & | bit-wise AND | bit-wise |
| ^ | bit-wise XOR | bit-wise |
| \| | bit-wise OR | bit-wise |
| && | logical AND | logical |
| \|\| | logical OR | logical |
| ?: | conditional “ternary if-the-else” | conditional |

Examples:

- Reduction means “tree”:
  - `^A[2:0]` performs xor-tree of 3 bits and returns 1 bit. It is equivalent to `A[2]^A[1]^A[0]` and to `^{A[2],A[1],A[0]}`
- Bitwise:
  - `A & B` performs bitwise and between buses A and B
- Logical
  - `A == B && B==C` returns only one bit!
- Arithmetic
  - `A >>> 3` is arithmetic right shift of A by 3 bits
  - Comparison `A < B` respects whether A and B are signed.
- Concatenation operator: `w1 = {r1[1:0],r1[3:2]};` in this example above, the 4-bit bus w1 will be wired with `r1[3:2]` as its two LSBs, and with `r1[1:0]` as its two MSBs;

## 6. Signal Values

A single bit value can be logical (“**0**” or “**1**”), non-logical “**Z**” - meaning “high impedance” (floating wire), or “**X**” which means “unknown”. Integer numbers can be specified using a representation of your choice (decimal, binary, hexadecimal). To do so, use the following pattern: **<number of bits>’<radix><value>** where radix is specified by a single character h, d, b (hexadecimal, decimal, binary). Verilog internally represents negative numbers in 2’s complement format. An optional **signed** specifier can be added before the wire or the reg for signed arithmetic.

Example 1 – unsigned integer – all following assignments, assign the value 12 to X:

| | | | |
|---|---|---|---|
| `X=12;` | `X=4’hc;` | `X=4’b1100;` | `X=4’d12;` |

**Make sure to understand the following elementary principle: When you change the representation of a number, its value doesn’t change, it is only presented differently.**

Example 2 – signed integer – all following assignments, assign the value -1 to X:

| | | | |
|---|---|---|---|
| `X=-1;` | `X=4’hf;` | `X=4’b1111;` | `X=-4’d1;` |
| | `X=-4’h1` | `X=-4’b0001` | |

As you probably guessed, the signed representation is the well-known two’s complement.

## 7. Modules

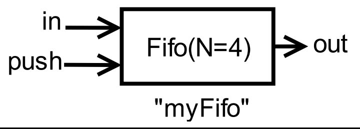
Module describes a synchronous/combinational circuit. You should try writing each module in its own .v source file.

Example: “Fifo” **module declaration**:

```
module Fifo(in,push,pop,out);
      parameter N=2;
      input in;
      input push;
      output out;
      //some logic goes here
endmodule
```

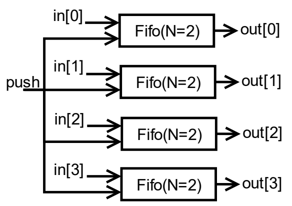
The saved words are in bold-blue. This module has 2 single bit inputs (in,push) and one single bit output (out). In addition, there is a parameter (N) which has a *default* value of 2 and its value cannot change during the operation of the module.

Once you have written a module, you can create as many **instances of it** as you wish. Each instance can be created with different parameter values (in the Fifo module mentioned above, N can have values different than 2). Design hierarchy is constructed by instantiating modules inside other modules. Below is an example

| Example: instantiation of a **single** Fifo(4) module: | Equivalent schematic |
|---|---|
| `reg in, push;`<br>`wire out;`<br>`Fifo #(4) myFifo(in, push, out);` |  |

Where Fifo is a module type, #(4) means that the parameter of the module, which is N, is set to 4 for this particular instance. “myFifo” is the name given for this particular instance, and the wires/registers in the parentheses are the higher-level module’s nets I chose to connect to the instance of Fifo.

One could also create an array of instances using a (i) special syntax or a (ii) generate block. Note that a conditional instantiation of modules (with if-else, loops and caese) can be done only using “generate”!

| | Example: instantiation of a an **array** of 4 Fifo(2) modules: | Equivalent schematic |
|---|---|---|
| **(i)** | `reg [3:0] in; reg push; wire [3:0] out;`<br>`Fifo fArr[3:0](in, {4*{push}}, out);` |   in[0]<br>in[1]<br>in[2]<br>in[3]<br>push<br>Fifo(N=2)<br>Fifo(N=2)<br>Fifo(N=2)<br>Fifo(N=2)<br>out[0]<br>out[1]<br>out[2]<br>out[3] |
| **(ii)** | `reg [3:0] in; reg push; wire [3:0] out;`<br>`genvar i;`<br>`generate for ( i = 0; i < 4; i = i+1 )`<br>`  begin : fArr`<br>`   Fifo f(in[i], push, out[i]);`<br>`  end`<br>`endgenerate` | |

## 8. Combinational Logic

Inside modules, you design the logic. The combinational logic can be written in two ways:

1. Continuous assignment – this way you can assign values to `wire` only. For example, if you have w1, and w2 declared as wires, then the continuous assignment can be written as:

| Example: two different continuous assignments (the 1st one uses an if-then-else ternary operator, the 2nd one assigns a constant value. |
|---|
| `assign w1 = (select) ? b : a;`<br>`assign w2 = 1’b1;` |

2. Behavioral always@ block – this way you can write a more complicated behavioral description of a combinational logic. The list of signals after the @ symbol is called **sensitivity list.** The block will be evaluated only when one of these signals changes. The logic description itself is written between the saved words “begin” and “end”.

Example: combinational behavioral always block, equivalent to the 1st continuous assignment of the previous example.

```
always @(a or b or select)
  begin
    if(select == 1)
      result_r = b;
    else
      result_r = a;
  end
```

Notes:

a. Any signal fed by an always block, must be defined as `reg`. However, no flip-flops are synthesized as we wish to describe a combinational circuit.
b. Every if statement must have explicit options for the “else” case, otherwise a **latch will be inferred** to store the previous value of the reg. This is an unwanted phenomenon.
c. All the input signals of the always block must be specified in the sensitivity list. Otherwise, a simulation of this block will not be accurate.
d. Don’t write `assign` within an `always@` block. It’s a common syntax error. If you need to write a combinational logic expression, pick one way of doing it – either with an always@ block or with a continuous assignment, but not both!

## 9. Synchronous Logic

Synchronous logic blocks are described by always@ blocks which respond to clock rising/falling edges rather than to some data signals (like it was in the case of combinational logic). There are two important aspects we need to emphasize here.

**The first issue is blocking vs. non-blocking assignments**:

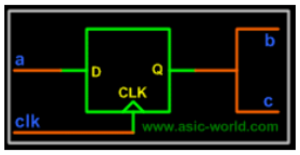
Example: blocking assignment

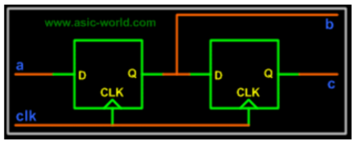
```
module blocking (clk,a,c);
      input clk,a;
      output c;
      reg c,b;
      always @ (posedge clk)

      begin
            b = a;
            c = b;
      end
endmodule
```

Example: non-blocking assignment

```
module nonblocking (clk,a,c);
      input clk,a;
      output c;
      reg c,b;
      always @ (posedge clk)

      begin
            b <= a;
            c <= b;
      end
endmodule
```

The lefthand side module employs a blocking assignment (using “=”) which implies that the lines of the verilog code are evaluated sequentially. The righthand side module on the other hand, employs a non-blocking assignemnt (using “<=”) which means that the assignments happen simultaneously, regardless of the line order. The synthesized circuits of the two assignments will look as follows:





**The second issue is synchronous vs. asynchronous reset:**

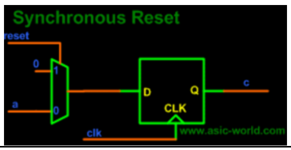
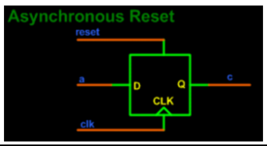
Example: sync. reset

```
always @(posedge clk)
begin
if(reset == 1)
      begin
      //initializations
      end
else
      begin
      //actual stuff
      end
end
```

Example: asynchronous reset

```
always @(posedge clk, posedge reset)
begin
if(reset == 1)
      begin
      //initializations
      end
else
      begin
      //actual stuff
      end
end
```

The lefthand side code is sensitive to the rising edge of the clock, hence it employs synchronous reset. It means that the reset signal may affect the circuit only when the clock rises. On the other hand, the righthand side code is equally sensitive to the clock and to the reset signals. Accordingly, whenever the reset signal rises, it will cause initailizations. This sort of reset is called asynchronous. The table below presents the synthesized circuits of the two reset techniques.

| | |
|---|---|
|   Synchronous Reset<br>reset<br>0<br>1<br>a<br>D<br>Q<br>CLK<br>c<br>clk<br>www.asic-world.com |   Asynchronous Reset<br>reset<br>a<br>D<br>Q<br>CLK<br>c<br>clk |

## 10. Test bench

A test bench is a special form of a module which is not intended for instantiation. Instead, it is a scenario for testing other module. In Verilog, we write test-benches in .v files. A general structure of a test-bench module is (i) instantiation of unit-under-test (UUT) module (ii) stimulation inputs to the UUT (iii) observation of UUT outputs and comparing them against expected values. Test-bench describes a sequence of inputs fed into a UUT during the test time. The sequence of inputs is listed inside a special block called **initial** block, which is similar to the always@ block, however it is performed only once.

Below is an example of a test-bench module which tests a combinational module full-adder (FA). It first instantiates the FA module under a name “uut”, then it begins the initial block where it loops through all possible inputs and checks the outputs of uut. If all the outputs agreed with expected values “Test Passed – FA_tb” string will be printed to the screen:

Example: Test-bench for a combinational module

```
module FA_tb();
reg a, b, ci, correct, loop_skipped;
wire sum, co;
integer ai, bi, cii;
FA uut (a, b, ci, sum, co);
initial begin
correct = 1; loop_skipped = 1;
#1 for( ai=0; ai<=1; ai=ai+1 ) begin
  for( bi=0; bi<=1; bi=bi+1 ) begin
    for( cii=0; cii<=1; cii=cii+1 ) begin
      loop_skipped = 0;
      a = ai[0]; b = bi[0]; ci = cii[0];
      #5 correct = correct & (a + b + ci == {co,sum});
    end
  end
end
if (correct && ~loop_skipped)
  $display("Test Passed - %m");
else
  $display("Test Failed - %m");
$finish;
end
endmodule
```

In the example above, we can notice a usage of “virtual” data types, **integer**s, which are neither registers nor wires. These data types only help us run loops and browse numerical values. Another feature we see is the **#5** notation, which stands for “adding a delay of 5 time-units”. Note that the #5 notation must be either followed by a Verilog command or terminated by a semicolon “;”. A concrete value of a time unit (in seconds) can be defined by a special command **`timescale 1 ns / 1 ps** which sets the value of a time-unit to 1 nanosecond and sets the resolution of the simulation to be 1 picosecond. You can also notice the usage of special Verilog simulation-functions: **$display** and **$finish** for printing messages to the screen and for terminating the simulation respectively. By the way, **%m** means printing the current module name.

Test-benches for synchronous modules look very similar. The only additions are (i) the periodic stimulation of the clock signal; (ii) reseting; (iii) input stimulation and and output checking at specific times. Remember, in synchronous test-benches the timing is the key.

Example: Test-bench for a synchronous module

```
module Arbiter_tb();
reg clk, reset, req_0, req_1, correct;
wire gnt_0, gnt_1;
Arbiter uut (clk, reset, req_0, req_1, gnt_0, gnt_1);
initial begin
clk = 1; reset = 1; correct = 1;
#1;
#20 reset = 0; req_0 = 1; req _1 = 0;
#10 correct = correct & (gnt_0 == 1 && gnt_1 == 0);
    req_0 = 0; req_1 = 0;
#10 correct = correct & (gnt_0 == 0 && gnt_1 == 0);
    req_0 = 0; req_1 = 1;
#10 correct = correct & (gnt_0 == 0 && gnt_1 == 1);
    req_0 = 0; req_1 = 0;
#10 correct = correct & (gnt_0 == 0 && gnt_1 == 0);
    req_0 = 1; req_1 = 1;
#10 correct = correct & (gnt_0 == 1 && gnt_1 == 0);
    req_0 = 0; req_1 = 0;
#10 correct = correct & (gnt_0 == 0 && gnt_1 == 0);
#10 correct = correct & (gnt_0 == 0 && gnt_1 == 0);
if (correct==1)
  $display("Test Passed");
else
  $display("Test Failed");
$finish;
end
always #5 clk = ~clk;
endmodule
```

In the example above, the purple lines set the initial values of the clock, reset and data inputs, and perform a resetting that lasts for about two clock periods. The red line periodically generates a clock with a 10 time-units period time and 50% duty-cycle (The tilde "~" stands for "not"). The blue lines assert the test inputs one time-unit after the clock rising edge (thanks to the "#1" time shift trick!), such that these inputs will be sampled in the next rising edge. The green lines query the current outputs of the UUT. Notice the insignificant difference between the "&" which is a bitwise-and operator, and the "&&" which is a logical-and operator between two logical expressions. Generally, one can mix between the bitwise and the logical operators because anyway 0 means false and any non-zero value means true, however it is not elegant to do so.

## 11. Functions

Verilog allows writing functions, which are basically implementations of Boolean functions. A Verilog function can have arbitrary number of inputs but only one output. It can invoke other functions. The return value is performed by assigning into a function name.

Example: Defining a function

```
function myfunction;
      input a, b, c, d;
      begin
         myfunction = ((a+b) + (c-d));
      end
  endfunction
```

Example: calling a function

```
   W1 = myfunction (a,b,c,d);
```

## 12. Tasks

Tasks resemble small modules. A Verilog task can have arbitrary number of inputs and outputs. It can invoke other functions and tasks. When calling for a task, you simply attach wires to it – similar to an instantiation, but there is no actual instance you can refer to.

Example: Defining a task

```
task convert;
      input [7:0] temp_in;
      output [7:0] temp_out;
      begin
        temp_out = (9/5) *( temp_in + 32);
      end
  endtask
```

Example: calling a task

```
   convert (temp_a ,temp_b);
```