Guidelines on how to use Verilog to describe hardware.

Developed for: **COE-308 Students and Similar level**

By: **Dr. Abdelhafid Bouhraoua**

1. Starting with Verilog:

Verilog-HDL is a standard hardware description language that is easy to use and learn. The expected learning curve should be very fast.

A hardware description starts from defining the ***models*** you are going to use for your design. A model is a construct similar to a block (hardware) or a user-defined type (in software). Designers will define all their models, then they will use them to build the whole circuit by using one or several instance.

2. Verilog module

A model is called **`module.`** The module constitutes the basic container for any Verilog description. A module is defined as follows (key words are in bold and optional names are between < >)

```
module <module_name> (<port_list>);
...
endmodule
```

A module name should be unique for the same project. It is strongly recommended that you put every module in a separate file with the file name being the same as the module name with the extension .v:
The module ***counter*** should be put is the file ***counter.v***

3. Port list definition

The port list is the module interface to the external world of the module. It is in other terms, how the module communicates with the outside world. The port list contains only a list of names. The direction and wire width of each port will be defined after the module as shown in the example below:

```
module counter_4bits ( clk,
                       reset,
                       mode,
                       parallel_in,
                       cnt);

input       clk;
input       reset;
input       mode;
input  [3:0]   parallel_in;
output [3:0]   cnt;
```

endmodule

From the description above it is clear that:

- clk and reset are two inputs with a width of a single wire (1 bit)
- mode is a single wire input too
- parallel_in is a 4-bit input bus (4 wires)
- cnt is a 4-bit output bus

4. Structural and behavioral description

Once the ports are defined, the next step is to describe the behavior of the counter so that it can be simulated. Two choices are available to us:

1. Describe the counter in terms of gate connections
2. Describe the behavior of the counter using behavioral description constructs

The first option is called **structural** description while the second option is called **behavioral description**.

In the structural description there are **gates** that are built in the Verilog language itself and others that are defined by the users as regular modules (generally by circuit fabrication companies who will supply the designers with their own gate libraries)

4.1 Structural description

The structural description is given below:

```
module counter_4bits ( clk,
                       reset,
                       mode,
                       parallel_in,
                       cnt);

input       clk;
input       reset;
input       mode;
input  [3:0]   parallel_in;
output [3:0]   cnt;

// This is a comment
// Declare the internal wires
wire   [3:0]  cnt;      // output of the counter
                        // using the same name as the output port
                        // means that the wire is connected to the
                        // output
wire   [3:0]  cnt_in;   // input to the counter flip-flops

// modeling a synchronous counter with parallel input
// The mode input is a control signal:
// mode = 0: the counter counts up every clock cycle
// mode = 1: the counter loads the value on the parallel_in bus
//
// a synchronous counter means that bit[i] must change its value
// from 0->1 or 1->0 if all the previous bits are 1
// computing all the previous bits at 1 means using and gates
// and gates are built-in gates in Verilog

wire  [1:0]   at_one;

assign at_one[0] = cnt[0] & cnt[1];
assign at_one[1] = cnt[2] & at_one[0];

assign cnt_in[0] = (mode & parallel_in[0]) | (~mode & ~cnt[0]);
assign cnt_in[1] = (mode & parallel_in[1]) |
                   (~mode & ((cnt[0] & ~cnt[1])|(~cnt[0] & cnt[1])));
assign cnt_in[2] = (mode & parallel_in[2]) |
                   (~mode & ((at_one[0]&~cnt[2])|(~at_one[0]&cnt[2])));
assign cnt_in[3] = (mode & parallel_in[3]) |
                   (~mode & ((at_one[1]&~cnt[3])|(~at_one[1]&cnt[3])));
```

```
// Instantiating the 4 flip-flops
// The flip-flops are not part of the built in gates
// The flip-flop module is like:
// module DFF (q, ck, res, d);
// input  d, ck, res;
// output q
// ...
// endmodule

DFF dff_bit0 (.d   (cnt_in[0]),
              .q   (cnt[0]),
              .clk (clk),
              .res (reset));

DFF dff_bit1 (.d   (cnt_in[1]),
              .q   (cnt[1]),
              .clk (clk),
              .res (reset));

DFF dff_bit2 (.d   (cnt_in[2]),
              .q   (cnt[2]),
              .clk (clk),
              .res (reset));

DFF dff_bit3 (.d   (cnt_in[3]),
              .q   (cnt[3]),
              .clk (clk),
              .res (reset));
```

endmodule

This is illustrating the structural description. In fact assign statement are used to describe combinational logic equations attached to a wire and not gates. The real example of the structural description is the instantiation of the DFF blocks.

4.2 Behavioral description

The behavioral description is often very simple compared to the structural description as shown below:

```
module counter_4bits ( clk,
                       reset,
                       mode,
                       parallel_in,
                       cnt_out);

input       clk;
input       reset;
input       mode;
input  [3:0]   parallel_in;
output [3:0]   cnt_out;

// Definition of the flip-flops
reg    [3:0]   cnt;
```

```
// Description

always @(posedge clk or negedge reset)
begin
    if (reset == 1'b0)
        cnt <= 4'b0000;
    else
        if (mode == 1)
           cnt <= parallel_in;
        else
           cnt <= cnt + 1;
end

assign cnt_out = cnt;

endmodule
```

The **always** block is a statement that define a continuous monitoring process that is there to describe a hardware element. The **posedge** prefix means the positive edge or the rising edge of the signal clk. This refers to the transition from 0 to 1.

The block means: everytime there is a rising edge of the signal clk do what is between **begin** and **end**.

The '==' is a comparison. the 1'b0 is a constant description in Verilog. Constants can be described in various forms. The 1'b0 means 1-bit value expressed in binary (b) with the value being 0. As an example, let's show some of the various ways of representing a constant value of 27:

```
27                // regular, by default, decimal value
5'd27             // 5-bits decimal value
6'h11             // expressed in hexadecimal
5'b10001          // in binary
5'o33             // in octal as 3x8 = 24 + 3 = 27
```

The behavior description is a series of nested **if** statements which describe what a register (flip-flops) value will be after the rising edge of the signal clk. The register is cnt. It is a 4 bits register defined by the statement:

```
reg    [3:0]   cnt;
```

The register cnt is assigned 0 when the signal reset is low.

5. Using modules in other modules

Using modules within other modules is the core of structural description. It is used to build a system using several components interconnected together within several nested modules until we reach the top level system module that will contain the main components.

This hierarchical description allows the user to hide the details at the top level. In order to illustrate this, let's imagine a system that uses our counter and some other blocks. Let's build a real time clock circuit that displays hours, minutes and seconds on seven segment displays. Let's assume that the input clock frequency is the power line 50 Hz. To get a second, we should divide the frequency by 50 using the counters.

```
module get_second ( clk,
                    reset,
                    clk_sec);
```

```
input        clk;
input        reset;
output        clk_sec;

wire          cnt_1_carry;
wire          cnt_1_9;
wire          cnt_2_4;
wire [3:0]    sec_cnt1_out;
wire [3:0]    sec_cnt2_out;

// Lower digit counter counts from 0 → 9
counter_4bits sec_cnt1 (.clk         (clk),
                        .reset       (reset),
                        .mode        (cnt_1_9),
                        .parallel_in (4'b0000),
                        .cnt_out     (sec_cnt1_out));

assign cnt_1_9 = sec_cnt1_out[3] & sec_cnt1_out[0]; // 4'b1001 == 9
assign cnt_1_carry = ~sec_cnt1_out[3];
counter_4bits sec_cnt2 (.clk         (cnt_1_carry),
                        .reset       (reset),
                        .mode        (cnt_2_4),
                        .parallel_in (4'b0000),
                        .cnt_out     (sec_cnt2_out));

assign cnt_2_4 = sec_cnt2_out[2];
assign clk_sec = ~cnt_2_4;

endmodule
```

Now let's describe the module that counts minutes and seconds as they both count from 00 to 59.

```
module minsec ( clk,
                reset,
                clk_minsec,
                minsec_out);

input        clk;
input        reset;
output       clk_minsec;
output [7:0] minsec_out;

wire         cnt_1_carry;
wire         cnt_1_9;
wire         cnt_2_5;
wire [3:0]   minsec_cnt1_out;
wire [3:0]   minsec_cnt2_out;

// Lower digit counter counts from 0 → 9
counter_4bits minsec_cnt1 (.clk         (clk),
                           .reset       (reset),
                           .mode        (cnt_1_9),
                           .parallel_in (4'b0000),
                           .cnt_out     (minsec_cnt1_out));

assign cnt_1_9 = minsec_cnt1_out[3] & minsec_cnt1_out[0]; // 4'b1001 ==
9
assign cnt_1_carry = ~minsec_cnt1_out[3];
```

```
counter_4bits minsec_cnt2 (.clk         (cnt_1_carry),
                           .reset       (reset),
                          .mode        (cnt_2_5),
                          .parallel_in (4'b0000),
                          .cnt_out     (minsec_cnt2_out));

assign cnt_2_5 = minsec_cnt2_out[2] & minsec_cnt2_out[0];
assign clk_minsec = ~cnt_2_5;
assign minsec_out = {minsec_cnt2_out,minsec_cnt1_out};
endmodule
```

The following module counts the hours. It counts from 00 to 23. It is a little more complicated as the generation of the mode signal for the two counters should be modified to take care of the last case (23).

```
module hours ( clk,
               reset,
               clk_day,
               hrs_out);

input        clk;
input        reset;
output       clk_day;
output [7:0] hrs_out;

wire         cnt_1_carry;
wire         cnt_1_9;
wire         cnt_1_3;
wire         cnt_2_2;
wire         mode_1;
wire         mode_2;
wire [3:0]   hr_cnt1_out;
wire [3:0]   hr_cnt2_out;

// Lower digit counter counts from 0 → 9
counter_4bits hr_cnt1 (.clk         (clk),
                           .reset       (reset),
                           .mode        (mode_1),
                           .parallel_in (4'b0000),
                           .cnt_out     (hr_cnt1_out));

assign cnt_1_9 = hr_cnt1_out[3] & hr_cnt1_out[0]; // 4'b1001 == 9
assign cnt_1_3 = hr_cnt1_out[1] & hr_cnt1_out[0]; // 4'b0011 == 3
assign mode_1 = cnt_1_9 | (cnt_2_2 & cnt_1_3);
assign cnt_1_carry = ~mode_1;
counter_4bits hr_cnt2 (.clk         (cnt_1_carry),
                       .reset       (reset),
                       .mode        (mode_2),
                       .parallel_in (4'b0000),
                       .cnt_out     (hr_cnt2_out));

assign cnt_2_2 = hr_cnt2_out[1];
assign #2 mode_2 = cnt_2_2 & cnt_1_3; // delay of #2 to make sure
                                      // mode is maintained after the
                                      // rising edge of the carry
assign clk_day = ~mode_2;
assign hrs_out = {hr_cnt2_out,hr_cnt1_out};
endmodule
```

Let’s describe a BCD to seven segment decoder.

```
module BCD7seg (bcd,
                dec_out);

input   [3:0]   bcd;
output  [6:0]   dec_out;

// Seven segments are:
//       a
//      ---
//   f | g | b
//      ---
//   e |   | c
//      ---
//       d
// -----------------------
// dec_out |6|5|4|3|2|1|0|
// -----------------------
// segment |g|f|e|d|c|b|a|
//-----------------------

always @(bcd)       // any of the bcd inputs change
begin
     case (bcd)
     4'b0000: dec_out = 7'b0111111;
     4'b0001: dec_out = 7'b0000110;
     4'b0010: dec_out = 7'b1011011;
     4'b0011: dec_out = 7'b1001111;
     4'b0100: dec_out = 7'b1100110;
     4'b0101: dec_out = 7'b1101101;
     4'b0110: dec_out = 7'b1111101;
     4'b0111: dec_out = 7'b0000111;
     4'b1000: dec_out = 7'b1111111;
     4'b1001: dec_out = 7'b1101111;
     default: dec_out = 7'b0000000;
     endcase
end
endmodule
```

The top level module will contain all of the different instances representing the counters and their decoders.

```
module clock_top (clk_50Hz,
                  reset,
                  seconds,
                  minutes,
                  hours);

input             clk_50Hz;
input             reset;
output  [13:0]    seconds;
output  [13:0]    minutes;
output  [13:0]    hours;

wire              clk_sec;
wire              clk_min;
wire              clk_hr;
```

```
wire    [7:0]     secs;
wire    [7:0]     mins;
wire    [7:0]     hrs;

// Instantiating the counters
get_second get_second1 (.clk      (clk_50Hz),
                        .reset    (reset),
                        .clk_sec  (clk_sec));

minsec seconds_1(.clk         (clk_sec),
                 .reset       (reset),
                 .clk_minsec  (clk_min),
                 .minsec_out  (secs));

minsec minutes_1(.clk         (clk_min),
                 .reset       (reset),
                 .clk_minsec  (clk_hr),
                 .minsec_out  (mins));

hours hours_1   (.clk         (clk_hr),
                 .reset       (reset),
                 .hrs_out     (hrs));

// Instantiating the BCD to 7 Segments Decoders
BCD7seg sec_dec_0 (.bcd       (secs[3:0]),
                   .dec_out   (seconds[6:0]));

BCD7seg sec_dec_1 (.bcd       (secs[7:4]),
                   .dec_out   (seconds[13:7]));

BCD7seg min_dec_0 (.bcd       (mins[3:0]),
                   .dec_out   (minutes[6:0]));

BCD7seg min_dec_1 (.bcd       (mins[7:4]),
                   .dec_out   (minutes[13:7]));

BCD7seg hrs_dec_0 (.bcd       (hrs[3:0]),
                   .dec_out   (hours[6:0]));

BCD7seg hrs_dec_1 (.bcd       (hrs[7:4]),
                   .dec_out   (hours[13:7]));

endmodule
```

6. Testing and simulating the design

After the design is complete, it needs to be verified to make sure it is actually behaving the way it is supposed to behave. In order to do that, stimuli need to be generated and applied to the device under test (DUT). The DUT (in fact the top level module of the circuit) is instantiated into another module that is called a testbench.

6.1 Building the testbench

The testbench is the module where all the stimuli generators and output analyzers are connected to the DUT. In our case, the testbench will contain:

- The instance the top level module of our design
- Apply the reset signal

- Interpret and display the outputs

The testbench module is shown below:

```
`timescale 1ns/1ps     // defines the precision of the simulation
module clock_tb;  // this is the testbench. It is the top module
                  // for the simulation and DOES NOT have any I/Os

// Define registers to generate stimuli
// ... and wires to capture the outputs
reg             clk;
reg             reset;
wire  [13:0]    sec_out;
wire  [13:0]    min_out;
wire  [13:0]    hr_out;

// Initialize the clk and reset signals
initial
begin
    clk = 1'b0;
    reset = 1'b1;        // start by initializing reset to inactive
                         // state of 1
    #12 reset = 1'b0     // after 12ns apply reset
    #15 reset = 1'b1     // release reset after that.
    #18000000 $finish    // simulate for 18000000 ns representing more
                         // than 24 hours in order to test all cases
                         // it is equal to 24 x 3600 x 50 x 4
end

// Generating the clock
always
   #2 clk = ~clk;

// Instantiate the design
clock_top clock_top_inst (.clk_50Hz     (clk),
                          .reset        (reset),
                          .seconds      (sec_out),
                          .minutes      (min_out),
                          .hours        (hr_out));

// analyzing the outputs
// everytime an output changes display the current time
// use local variables to display the content
reg  [7:0]  hrs, mins, secs;

always @(sec_out or min_out or hr_out)
begin
     // because the determination of the digits from the
     // decoded outputs is an identical task, we use the
     // verilog function construct to do that
     hrs[7:4] = seven_decode(hr_out[13:7]);
     hrs[3:0] = seven_decode(hr_out[6:0]);
     mins[7:4] = seven_decode(min_out[13:7]);
     mins[3:0] = seven_decode(min_out[6:0]);
     secs[7:4] = seven_decode(sec_out[13:7]);
     secs[3:0] = seven_decode(sec_out[6:0]);
     // display the results
     $display($time," %h:%h:%h\n",hrs,mins,secs);
```

```
end

function [3:0] seven_decode;
   input [6:0] segments;
begin
      case (segments)
      7'b0111111: seven_decode = 4'h0;
      7'b0000110: seven_decode = 4'h1;
      7'b1011011: seven_decode = 4'h2;
      7'b1001111: seven_decode = 4'h3;
      7'b1100110: seven_decode = 4'h4;
      7'b1101101: seven_decode = 4'h5;
      7'b1111101: seven_decode = 4'h6;
      7'b0000111: seven_decode = 4'h7;
      7'b1111111: seven_decode = 4'h8;
      7'b1101111: seven_decode = 4'h9;
      default: seven_decode = 4'hf;
      endcase
end
endfunction

endmodule
```

After completing the testbench the simulation is started and the results are checked in the logfile.

7. Describing memories

Memories are an integral part of many designs and simulating them efficiently is a major concern among designers. Verilog offers a versatile way of describing memories that allows the designers to create memorization elements that are accessed in a group like an array. The statement used to create a memory variable in Verilog is:

```
reg  [15:0] mem [0:1023] ;     // means a memory of 1024 positions of
                               // word size of 16-bits
```

Beside that, it is the responsibility of the designer to describe the memory the way it should be. Below are some examples of memory models commonly used in designs:

1 – Asynchronous RAM

The read is completely asynchronous while the write occurs at the falling edge of the write enable signal.

```
module async_ram (addr,
                  din,
                  dout,
                  web);

input  [9:0]   addr;
input  [15:0]  din;
output [15:0]  dout;
input          web;

reg    [15:0] mem [0:1023];

// writing
always @(posedge we)
begin
     mem[addr] <= din;     // writing din into the specified address
```

end

```
//reading
assign dout = mem[addr];
endmodule
```

2 – Asynchronous ROM

The asynchronous ROM is identical to the asynchronous RAM except that it does not accept writes. It also needs to be initialized. Initialization is generally done through the **$readmemb** or **$readmemh** system tasks.

The following has been extracted from the IEEE Verilog reference manual

Two system tasks **$readmemb** and **$readmemh** read and load data from a specified text file into a specified memory. Either task can be executed at any time during simulation. The text file to be read shall contain only the following:

White space (spaces, new lines, tabs, and form-feeds)
Comments (both types of comment are allowed)
Binary or hexadecimal numbers

The numbers shall have neither the length nor the base format specified. For **$readmemb**, each number shall be binary. For **$readmemh**, the numbers shall be hexadecimal. The unknown value (x or X), the high impedance value (z or Z), and the underscore (_) can be used in specifying a number as in a Verilog HDL source description. White space and/or comments shall be used to separate the numbers.

In the following discussion, the term address refers to an index into the array that models the memory. As the file is read, each number encountered is assigned to a successive word element of the memory. Addressing is controlled both by specifying start and/or finish addresses in the system task invocation and by specifying addresses in the data file.

When addresses appear in the data file, the format is an at character (@) followed by a hexadecimal number as follows:

@hh...h

Both uppercase and lowercase digits are allowed in the number. No white space is allowed between the @ and the number. As many address specifications as needed within the data file can be used. When the system task encounters an address specification, it loads subsequent data starting at that memory address. If no addressing information is specified within the system task, and no address specifications appear within the data file, then the default start address shall be the left-hand address given in the declaration of the memory.

Consecutive words shall be loaded until either the memory is full or the data file is completely read. If the start address is specified in the task without the finish address, then loading shall start at the specified start address and shall continue towards the right-hand address given in the declaration of the memory. If both start and finish addresses are specified as parameters to the task, then loading shall begin at the start address and shall continue toward the finish address, regardless of how the addresses are specified in the memory declaration.

When addressing information is specified both in the system task and in the data file, the addresses in the data file shall be within the address range specified by the system task parameters; otherwise, an error message is issued and the load operation is terminated.

A warning message shall be issued if the number of data words in the file differs from the number of words
in the range implied by the start through finish addresses.

Example:

reg [7:0] mem[1:256];

Given this declaration, each of the following statements load data into mem in a different manner:

initial $readmemh("mem.data", mem);
initial $readmemh("mem.data", mem, 16);
initial $readmemh("mem.data", mem, 128, 1);

The first statement loads up the memory at simulation time 0 starting at the memory address 1. The second statement begins loading at address 16 and continue on towards address 256. For the third and final statement, loading begins at address 128 and continue down towards address 1. In the third case, when loading is complete, a final check is performed to ensure that exactly 128 numbers are contained in the file. If the check fails, a warning message is issued.

Let's describe the ROM model now:

```
module async_rom (addr,
                  dout);

input  [9:0]   addr;
output [15:0]  dout;

reg    [15:0] mem [0:1023];

// initialization
initial
      $readmemh(“mem.dat”,mem);

//reading
assign dout = mem[addr];

endmodule
```

The mem.dat (by the way, there is no restriction on the file name extension) can be:

```
// starting at address 000
53ab    // first 16bits word
2232    // etc...
1aa5
2006
@1e3    // jumping to address 1e3 = 483
5555
5528
5527
```

There can be various combinations of the initialization file as explained in the excerpt .

3 – Synchronous RAM

The description of synchronous RAMs is very similar to the asynchronous ones except that everything is governed by the clock signal

An example of two RAMs is given below. First, the description of a memory that is synchronous for both read and write operations. Second the description of a memory that is asynchronous for reads and synchronous for writes.

```
module synch_mem_rw (clk,
                     addr,
                     din,
                     dout,
                     web);

input          clk;
input  [9:0]   addr;
input  [15:0]  din;
output [15:0]  dout;
input          web;

reg    [15:0]  mem [0:1023];
reg    [15:0]  dout;

always @(posedge clk)
begin
    if(web == 1'b0)
       mem[addr] <= din;
    else
       dout <= mem[addr];
end

endmodule
```

The memory that is asynchronous for reads and synchronous for writes:

```
module synch_mem_w  (clk,
                     addr,
                     din,
                     dout,
                     web);

input          clk;
input  [9:0]   addr;
input  [15:0]  din;
output [15:0]  dout;
input          web;

reg    [15:0]  mem [0:1023];

always @(posedge clk)
begin
    if(web == 1'b0)
       mem[addr] <= din;
end

assign dout = mem[addr];
endmodule
```

4 – Multiple ports memories

Whether for describing register files or other types of memories, multiple port memories can be described with relative ease if in any case a single port can write. Multiple port memories where many ports can write are a little more complex. However, these are not used often compared to the first category.

Let's describe a memory that has two read ports and a single write port. The two read ports are asynchronous while the write port is synchronized with the clock.

```
module dpr_mem      (clk,
                     raddr_a,
                     raddr_b,
                     waddr,
                     rdata_a,
                     rdata_b,
                     wdata,
                     web);

input           clk;
input  [9:0]    raddr_a;
input  [9:0]    raddr_b;
input  [9:0]    waddr;
output [15:0]   rdata_a;
output [15:0]   rdata_b;
input  [15:0]   wdata;
input           web;

reg    [15:0]   mem [0:1023];

always @(posedge clk)
begin
    if(web == 1'b0)
       mem[waddr] <= wdata;
end

assign rdata_a = mem[raddr_a];
assign rdata_b = mem[raddr_b];

endmodule
```