Learning verilog HDL

Documenting various beginner projects, syntax and common paradigms of verilog HDL. My set-up

- Ubuntu 18.04
- Icarus verilog for compilation
  - Version 12.0
- Visual studio code for editing
  - Verilog-HDL extension
- gtkwave for visualising waveforms

Command line interface

- Compiling
  - Using icarus, to compile the `<test.v>` file into `<output.out>` ,
  - `iverilog <test.v> -o <output.out>`
- Running
  - To run a testbench simulation, which was compiled into `<output.out>`
  - `./<output.out>`
- Viewing
  - To view the simulation outut of a testbench simulation, whose outputs ere dumped into `<waveform.vcd>` using $dumpfile and $dumpvars directives
  - `gtkwave <waveform.vcd>`
- Automation using makefiles

```
# Source files used
source          = and2.v

# Testbench code
testbench       = and2_tb.v

# Result of compilation
object          = and2.out

# Waveform file
wave            = and2.vcd

compile: $(object)
.PHONY: compile

$(object): $(source) $(testbench)
        iverilog $(testbench) -o $(object)

$(wave): $(object)
        ./$(object)

simulate: $(wave)
        gtkwave $(wave)
.PHONY: simulate

run: $(object)
        ./$(object)
.PHONY:run
```

- `make compile` : compiles to make the object file
- `make run` : runs the object file
- `make simulate` : visualises result stored using `gtkwave`

Basics of verilog

- Hardware description language - meaning, its used for describing hardware, and not for computation on conventional computers.
- Does not follow line-by-line execution. We are only describing hardware
- Designed with modularity in mind.
- Similar to C

Syntax features

- Verilog does not care about whitespaces! (in most cases)
- Is case sensitive

Comment

- Single line comments:

```
// This is a commment
```

- Multi-line comments

```
/*
This is a mult-line comment
*/
```

Modules

- Modules are used for modularity. They represent components.

Ports

- Ports are the interface used by the external world to interface with an instance of a module. They can have 3 types :
  - input
  - output
  - inout

Syntax

- This is the basic structure of a verilog module.

```
module test_module(port1, port2, port3, ..., portx);
    // Port definitions
    input port1, port2;
    output port3;
    inout portx;
    // Doing stuff
    ...
endmodule
```

- The port types can also be definied in the bracket.

```
module test_module(
    input port1, port2,
    output port3,
    ...
    inout portx
);
    // Doing stuff
    ...
endmodule
```

Module instantiation

```
test_module test_instance (net1, net2, net3, ... netx);
```

- We can also specify which nets connect to which ports

```
test_module test_instance (.port1(net1), .port2(net2), .port3(net3), ..., .portx(netx));
```

- Both of these connects
  - net1 -> port1
  - net2 -> port2
  - net3 -> port3

    ...
  - netx -> portx

Parameters

- Used to customize instances of the same module
- basic syntax to define parameters with :

```
parameter
  param_1=default_value_1,
  param_2=default_value_2,
  ...
  param_x=default_value_x;
```

- During instantiation, parameter values can be overriden by several methods:

```
// First method

test_module #(
  value_1,
  value_2,
  value_3)
test_instance(
  port1,
  ...
  portx
);

// Second method

test_module #(
  .param_1(value_1),
  .param_3(value_3),
  .param_2(value_2))
test_instance(
  port1,
  ...
  portx
);

// Third method

test_module test_instance(
  port1,
  ...
  portx
);
defparam
  test_instance.param_1=value_1,
  test_instance.param_2=value_2,
  test_instance.param_3=value_3;
```

- `localparam` can be used instead of `parameter` for parameters that cannot be over-ridden during instantiation.

Data types

- Physical data types
  - `wire` - Used for wiring different components together
  - `reg` - Used for temorarily storing data, like registers
- Abstract data types
  - `integer` - 32 bit signed value
  - `time` - 64 bit unsigned value from system task $time
  - `real` - Floating point value
  -

Values and literals

- 4 basic values
  - 0
  - 1
  - X (unknown/undefined)
  - Z (high-impedance)
- The last 2 are only for physical data types
- Literals are defined in the following formats

```
// <width>'<sign><radix><value>
4'b1000     // 4-bit unsigned binary 1000     : 1000
4'd15       // 4-bit unsigned decimal 15      : 1000
8'd32       // 8-bit unsigned decimal 32      : 00100000
8'sd32      // 8-bit signed decimal 32        : 00100000
8'b10x      // 8-bit unsigned binary 10x      : 0000010x

// '<radix><value>
'b10        // Unsigned Binary 10             : 10
'hf0        // Unsigned Hexadecimal f0        : 11110000
'o77        // Unsigned Octal 77              : 111111
'sb110      // Signed Binary 110              : 110       (sign extension with 1s)
'b110       // Unsigned Binary 110            : 110       (sign extension with 0s)
```

- The letters for sign and radix are **not** case sensitive.
- Signed notation

    - Does not change the bit pattern being converted, only affects its interpretation. A stack overflow question on this topic
    - For signed representation, use s. Else, dont use s.
    - Changes end result when the size is not specified and sign extension is required
    - Changes the effect of arithmetic right shift operator

```
integer a;

a = 8'sb110;   // Gives value 6  : 0b00000000000000000000000000000110
a = 'sb110;    // Gives value -2 : 0b11111111111111111111111111111110
```

- Radix abbreviations
    - b - Binary
    - d - Decimal
    - o - Octal
    - h - Hexadecimal

Scalars, Vectors

- In verilog, scalars are 1-bit wide data types, like a simple `reg` or `wire`.
- We can define buses using vectors. They are defined as follows :

```
wire[7:0] bus1;   // 8-bit wide little-endian bus
wire[0:7] bus2;   // 8-bit wide big-endian bus
```

- The indexing can be either from high:low (little endian) or low:high (big endian).
- Slices of the vectors can be taken as follows:

```
bus1[6:3] = 4'ha;   // Bits 5 to 3 (both inclusive) become 1010
```

Tasks and Functions

Tasks and functions are similar to procedures and functions in c. They are defined inside modules.

Tasks

- Can have multiple arguments of type
    - input

    - output
    - inout
- Does not return anything
- Tasks are defined as follows:

```
// Defining a task
task test_task;
  input [3:0] businp;
  output out;

  // Body of task
  begin
  ...
  end
endtask
```

- Tasks can be instantiated as follows:

```
test_task(inp, out);
```

Functions

- Should have at least one input type argument
- Cannot have output type arguments
- Return a single value
- They are defined as follows:

```
function [3:0] test_func;
  input [1:0] businp;

  test_func = ...; // Return value
endfunction
```

- Functions are instantiated as follows:

```
out = test_task(inp);
```

System tasks

System tasks are built-in tasks. All system tasks are preceeded with `$`

Printing to screen

- `$display` : Displays passed arguemnts to console
- `$strobe` : Same as `$display` , but the values are all printed only at the end of current timestep instead of instantly
- `$monitor` : Displays every time one of its parameters changes.

```
$display ("format_string", par_1, par_2, ... );
$strobe ("format_string", par_1, par_2, ... );
$monitor ("format_string", par_1, par_2, ... );
```

- The format string used here is similar to that in normal programming languages and can have the format characters :
  - `%d` : Decimal
  - `%h` : Hexadecimal
  - `%b` : Binary
  - `%c` : Character
  - `%s` : String
  - `%t` : Time
  - `%m` : Hierarchy level
  - As usual, we can specify number of spaces (eg : %5d)

Random

- `$random` : Generates signed 32-bit integers
- `$urandom` : Generates unsigned 32-bit integers
- `$urandom_range(a, b)` : Generates an unsigned integer within a speicified range

Time

- `$time` : Returns time as 64-bit integer
- `$stime` : Returns time ass 32-bit integer
- `$realtime` : Returns time as real number

Simulation control

- `$reset` : Resets time to 0
- `$stop` : Stops simulation and puts it in `interactive mode`
- `$finish` : Exits the simulator

Dumping to file

- These can dump variable changes to a simulation viewer like GTKWave.
- `$dumpfile("filename")` : Sets the file name to dump values into
- `$dumpvars(n, module)` : Dumps variables in module instantiated with name `module` and n levels below
  - Note that `$dumpvars` does not include array variables into the waveform.
  - To add array variables, they need to be manually specified

    ```
    $dumpvars(1, tud.mem[0]);
    // Where mem is defined in module tud as
    // reg[7:0] mem[0:1024];
    ```

  - To add all values, use a for loop. There is no other way
- `$dumpon` : Initiates dump (only required if stopped manually)
- `$dumpoff` : Stops dump
- `$dumpall` : Dump all variables
- `$dumplimit(size)` : Sets limit on .vcd file size

Memory manipulation

- We can load values from files into memory elements.
- `$readmemb(filename, memname, startaddr, stopaddr)`
  - Reads data in binary from `filename`
  - Writes it into memory element `memname` from address `startaddr` to `stopaddr` (last 2 parameters are optional)
- `$readmemb(filename, memname, startaddr, stopaddr)`
  - Reads data in hexadecimal format from `filename`
- Memory is modeled as array of register vectors, like

  ```
  reg[7:0] memory[1023:0];
  ```

Log

- Log of 2 is very useful in verilog because we can get the width of a bus required to accomodate some maximum value.
- `clog2(inp)`
  - Logarithm of 2, ceiled (if result is not a perfect integer, the next highest integer is taken)

Operators

- Operators in verilog are almost identical to those used in C. They operate on data. The types are:

Arithmetic

- They are used to carry out arithmetics on operands. They can be `unary` or `binary`. `unary` operators have a single operand and `binary` have 2 operands.
- Unary operators are
  - `+` (plus sign)
  - `-` (minus sign)
- Binary operators are
  - `+` (add)
  - `-` (subtract)
  - `*` (multiply)
  - `/` (divide)
  - `%` (modulus)
  - `**` (exponentiation)

Bitwise

- Bitwise binary operations. Input and output size are the same
- Unary
  - `~` (negation)
- Binary
  - `|` (or)
  - `&` (and)
  - `^` (xor)
  - `~^` (xnor)

Reduction

- Input is a unary operant of multiple bits and output is a single bit. (eg. to find 8-bit AND of 8 bits in a register)
- Unary
  - `&` (and)
  - `~&` (nand)
  - `|` (or)

- `~|` (nor)
- `^` (xor)
- `~^` (xnor)

```
&(4'b0111);   // 0 (0 and 1 and 1 and 1)
&(4'b1111);   // 1 (1 and 1 and 1 and 1)
```

Relational

- Used to compare values, and results in a single bit (0 or 1)
- Binary
  - `<` (less than)
  - `>` (greater than)
  - `<=` (less than or equal to)
  - `>=` (greater than or equal to)
  - `===` (equal to, including `x` and `z` states)
  - `!==` (not equal to, including `x` and `z`)
  - `==` (equal to, excluding `x` and `z`)
  - `!=` (equal to, excluding `x` and `z`)

Logical

- These give output as a single bit (0 or 1). They are **not** bitwise operators
- Unary
  - `!` (not)
- Binary
  - `&&` (and)
  - `||` (or)

Shifting

- There are logical and arithmetic shift. Arithmetic right shift fills the additional bits with `1` if the number was singed and first bit was `1` , whereas logical shift will always fill them with `0` .
- Binary
  - `<< shift_amnt` Logical left shift
  - `>> shift_amnt` Logical right shift
  - `<<< shift_amnt` Arithmetic left shift
  - `>>> shift_amnt` Arithmetic right shift

```
// Arithmetic shifting example

8'b11111110  >>> 2;    // 00111111
8'b01111111  >>> 2;    // 00011111
8'b11111110  <<< 2;    // 11111000
8'b01111111  <<< 2;    // 11111100

8'sb11111110 >>> 2;    // 11111111
8'sb01111111 >>> 2;    // 00011111
8'sb11111110 <<< 2;    // 11111000
8'sb01111111 <<< 2;    // 11111100
```

Assignment

- Assign values
- Binary
  - `=` Blocking assignment - is evaluated and assigned immediately
  - `<=` Non-blocking assignment - rhs is evaluated immediately but lhs is assigned only after current timestep

```
// Swap upper, lower 8 bytes
always @(posedge clk)
{
  begin
    word[15:8] = word[7:0];
    word[7:0]  = word[15:8];
  end
}
```

This will not work since the statements are evaluated and assigned line-by-line

```
// Swap upper, lower 8 bytes
always @(posedge clk)
{
  begin
    word[15:8] <= word[7:0];
    word[7:0]  <= word[15:8];
  end
}
```

This will work since the statements are non-blocking and are assigned only after current time step.

Others

- `{a,b,..c}`

    - Concatenation
    - Concatenation can also be in lhs

```
{2'b11, 3'b010};    // 11010

{cout, sum} = a + b + cin;    // For an adder
```

- `{m{n}}`
    - Replication operator
    - Is equivalent to {n, n, ... n} (concatenation m times)

```
{4{2'b10}};         // 10101010
```

- `a?b:c`
    - Conditional operator
    - Similar to conditional operator in C
    - if(a) b else c
- `[n]`
    - Bit selection
- `[m:n]`
    - Slicing
    - This is only possible for constant `m` and `n`
    - Includes both indices m and n
- `[i+:k]`
    - Inlcudes indices i, i+1, ... i+k
    - Can be used for variable i
- Using reg or wire with `x` or `z` will lead to whole result being `x`.

Assignments

Continuous assignments

- For continously assigning values to a net. Only net type variables can be assigned. Reg type variables cannot be assigned.
- Assignments happen outside procedural blocks, in assign statements.

```
// Format
assign #delay <net_expression> = <expression>;Assignment

// Example - a 10-bit adder
wire cout;
wire[10:0] sum;

assign #5 {cout, sum} = in1 + in2 + cin;
```

- The delay is said to be `inertial`, because if there are 2 changes within the specified delay, only the 2nd change is considered. The first change is discarded.

Procedural assignments

- Assignment is done only when control is transferred to it. This is used for reg type variables.
- Assignments occur inside procedural blocks (`always`, `initial`).
- `initial` block
  - Executes once and becomes inactive after that
  - There can be multiple initial blocks that all start at time 0
- `always` bock
  - always statement continuously repeats itself throughout the simulation.
  - If there are
    multiple always statements, they all start to execute concurrently at time 0.
  - May be triggered by events using an event recognizing list @( ).
- `always` and `initial` statements can only execute a single statement. Use `begin .... end` blocks for multiple statements.
- Blocking and non-blocking assignments become relevant here
- Blocking assinments wait for other blocking assignments of the same time and are executed sequentially.
  `register = expression;`
- Non-blocking assignments do not wait for other non-blocking assignments of the same time. They are all executed concurrently. `register <= expression;`
- There can be intra-assignment delay (immediate evaluation, delayed assignment)
  `register = #delay expression;`

```
a <= #5 in1;      // Time 5 : a = in1
b <= #5 in2;      // Time 5 : b = in2


a = #5 in1;       // Time 5 : a = in1
b = #5 in2;       // Time 10 : b = in2
```

- Recmmended not to use blocking and non-blocking in same procedural blcok.

Quasi-continuous assignments

- LHS must be reg type, and assignment inside procedural blocks.
- We can assign registers to be continously assigned. After doing this, normal procedural assignments on the registers is useless.
- Doing another quasi-continous assignment overrides previous assignment.
- We need to use `deassign` statement to de-assign before we can use procedural assignments again.

```
begin
  ...
  assign register = expression1;  // Activate quasi-continuous
  ...
  register = expression2;         // No effect
  ...
  assign register = expression3;  // Overrides previous quasi-continuous
  ...
  deassign register;              // Disable quasi-continuous
  ...
  register = expression4;         // Executed.
  ...
end
```

- There cannot be delays. Only initialisation can be delayed.

Blocks

- Sequential blocks
  - `begin` and `end`
  - Statements are executed line-by-line
- Parallel blocks
  - `fork` and `join`
  - Statements are all executed at the same time. Order is irrelevant
- Blocks can be named by appending `: name`

```
begin : sample_name
  ...
end
```

Timing control

Delay-baseed

- `#` is used to specify delays.
- `#delay statement;` will have the statement executed after delay of `delay` periods.
- Intra-assignment delay can be used in procedural assignments.
    - `register = #delay expression;`

Event-based

- `@(event)` can be used to block sequential flow till an event occurs
- `@(signal) statement;` will wait for signal to change and when there is a change, executes statement.
- `always @(signal)` can be used to run procedural statements whenever there is a change in signal.
- `@(posedge signal)` triggers at the positive edge of signal.
    - positive edge is defined as `0 -> x/z -> 1`
- `@(negedge signal)` triggers at the negative edge of the signal.
    - negative edge is defined as `1 -> x/z -> 0`
- `@(*)` will be triggered when any of the input variables change.
- Multiple events can be used to trigger by using `or`

```
always @(posedge clk or reset or my_event)
  $display("hello");
```

Named event

- Define an event using
- `event my_event`
- Then, we can use the event as `@(my_event)` for timing.
- Call the event using `-> my_event`

Conditional and loop statements

- `if-else`

```
if ( expr )       statement;
else if ( expr )  statement;
else if ( expr )  statement;
else              statement;
```

- `case`

```
case ( expr )
  value1          : statement;
  value2          : statement;
  value3, value4  : statement;  // Multiple cases for same statement
  ...
  default         : statement;
endcase
```

- 2 variants
  - `casez` : Considers `z` in case values as dont cares
  - `casex` : Considers `x` and `z` in case values as dont cares

```
casex (sel)
  4'b1xxx : num = 0;    // Matches 4'b10xx
  4'bx1xx : num = 1;    // Matches 4'b01zx
  4'bxx1x : num = 2;
  4'bxxx1 : num = 3;
  default : num = -1;
endcase
```

- `while`

```
while ( expr )
  statement;
```

- `loop`

```
for ( init ; expr ; step)
  statement;
```

- `repeat`

```
repeat ( no_of_times )
  statement;
```

  - If argument is a variable/signal, it is evaluated only when repeat() statment is called

- `forever`

```
forever
  statement;
```

- They all use single statements. To execcute multiple statements sequentially, use `begin ... end` blocks.

Compiler directives

Like in `#` directives in c, there are compiler directives in verilog, which are preceeded by `` ` ``. Some compiler directives are

- `` `include ``
  - Similar to #include in c, used for inserting contents of another verilog file
- `` `define ``
  - Similar to c, used for defining macros
- `` `undef ``
  - Used to discard macros defined using `` `define ``
- `` `ifdef ``
  - Used to define areas of code that should be included if some macro has been defined. The area to be checked will be preceeded by the `` `ifdef `` tag and succeeded by the `` `endif `` tag, similar to `#ifdef` and `#endif` tags.
  - Some directives related to this are
    - `` `endif ``
    - `` `else ``
    - `` `ifndef ``
    - `` `elseif ``

Generate block

- Generate block can be used to dynamically create hardware definitions from iterative and conditional constructs (for, while, if, case, etc) to avoid having to repeat same code several times.
- Can be used for dynamically(still at compile-time, you cant create hardware from thin air!) instantiating
  - Modules
  - User-defined primitives
  - Gates
  - Continous assignment statements

    - Procedural assignment blocks
- We need special variables of type `genvar`
    - Only used and defined inside generate blocks
- As an example, look at my implementation of adders :
    - Ripple carry adder using generate

```
genvar i;

generate for (i = 0; i < WIDTH; i = i + 1)
  full_adder fa(a[i], b[i], c[i], s[i], c[i+1]);
endgenerate
```

    - Look-ahead adder using generate

```
genvar i,j;

generate
  for (i = 0; i < WIDTH ; i = i + 1)
  begin
    // Example
    // c[3] = g[2]      + g[1].p[2]     + g[0].p[1].p[2]    + c[0].p[0].p[1].p[2
    //        temp[3]     temp[2]         temp[1]             temp[0]
    wire[i+1:0] temp;

    assign temp[0] = &{c[0], p[i:0]};
    assign temp[i+1] = g[i];

    for (j = 1; j < i+1; j = j + 1)
        assign temp[j] = &{g[j-1], p[i:j]};

    assign c[i+1] = |(temp);
  end
endgenerate
```

User Defined Primitives

- They should have one and only one output.
- Output can be `1` , `0` , `x` or `z` .
- Inputs with `z` are automatically changed to `x`
- We define possible inputs and their corresponding outputs using `table` and `endtable`
- The symbols that can be used in the table are
    - `0` : Logic 0
    - `1` : Logic 1

- `x` : Unknown value
- `?` : Can be 0/1/x (input only)
- `-` : No change. (output only)
- `ab` : Change from a to b eg) `01` , `?1`
- `*` : Any change in input. Same as `??`
- `r` : Rising edge. Same as `01`
- `f` : Falling edge. Same as `10`
- `p` : Potential positive edge. Either `01` or `0x` or `x1`
- `n` : Potential negative edge. Either `10` or `1x` or `x0`

- Example
  - 2x1 multiplexer (combinational ex)

```
primitive mux(out, sel, a, b);
  output out;
  input sel, a, b;

  table
    //sel a   b   : out
      0   1   ?   : 1;
      0   0   ?   : 0;
      1   ?   1   : 1;
      1   ?   0   : 0;
      x   0   0   : 0;
      x   1   1   : 1;
  endtable
endprimitive
```

  - D latch (sequential level-triggered ex)

```
primitive dlatch (d, en, clr, q);
  input d, en, clr;
  output reg q;

  initial q = 0;

  table
  //d   en  clr : q : q(new)
    ?   0   0   : ? : -;
    ?   ?   1   : ? : 0;
    1   1   0   : ? : 1;
    0   1   0   : ? : 0;
  endtable
endprimitive
```

  - T flip-flop (sequential edge-triggered example)

```
primitive tflipflop (clk, clr, q);
  input d, clk, clr;
  output reg q;

  initial q = 0;

  table
  //clk clr : q : q(new)
    ?   1   : ? : 0;
    0?  0   : ? : -;
    10  0   : 1 : 0;
    10  0   : 0 : 1;
  endtable
endprimitive
```

Synthesis rules

Some rules of thumb to see how behavioral statements are usually synthesised.

- `assign` statements generally generates combinational logic. However, sequential logic can be formed similar to how gates can be used to form sequential building blocks.
- Conditional statements generate n-bit wide 2-to-1 multiplexers.
- Variable indexing on the right produces a multiplexer.
- Variable indexing on the left produces a demultiplexer.

```
// n-bit wide 2-to-1 mux
assign out1 = selb ? in2 : in1;

// Mutiplexer
assign outb = in1[sel];

// Demultiplexer
assign out1[sel] = inb;

// D latch
assign q = en ? d : q;
```

For synthesizing combinational circuits

- Do not rely on delays for timing (they are for simulation)
- There must not be feedback in combinational circuits

- For `if ... else` or `case` constructs, output of combinational ckts must be provided for all input cases.
- Else, circuit can be synthesized as sequential

Styles for synthesis

- Netlist of verilog built-in primitives

```
module half_adder(a, b, s, cout);
  input   a, b;
  output  s, cout;

  xor x1(s, a, b);
  and a1(cout, a, b);
endmodule
```

- Using user-defined primitives (UDPs)
- Continuous assignments

```
module carry(cout, a, b, c);
  output  cout;
  input   a, b, cl

  assign cout = (a & b) | (b & c) | (a & c);
endmodule;
```

- Using procedural blocking assignments

```
module mux2to1(f, in0, in1, sel);
  input in0, in1, sel;
  output reg f;

  always @(in0 or in1 or sel)
    if(sel)   f = in1;
    else      f = in0;
endmodule
```

- Functions

```
module full_adder(s, cout, a, b, cin);
  input a, b, cin;
  output s, cout;

  assign s    = sum(a, b, cin);
  assign cout = carry(a, b, cin);
endmodule

function sum;
  input x, y, z;

  sum = x ^ y ^ z;
endfunction

function carry;
  input x, y, z;

  carry = (x & y) | (x & z) | (y & z);
endfunction
```

- Tasks (without event or delay control)

```
  module full_adder(s, cout, a, b, cin);
  input a, b, cin;
  output reg s, cout;

  always @(*)
    FA(s, cout, a, b, cin);

  task FA;
    output sum, carry;
    input A, B, C;

    begin
      sum = A ^ B ^ C;
      carry = (A & B) | (A & C) | (B & C);
    end
  endtask
endmodule
```

- Behavioral statements
- Interconnected modules of above

Shortcuts for coding

- Declaring output and reg in same statement

```
output reg[7:0] data;

// Instead of
output[7:0]     data;
reg[7:0]        data;
```

- Declaring reg type variables with initial value

```
reg data = 0;

// Instead of
reg data;
initial data = 0;
```

References

1. Summary of verilog syntax
2. asicworld.com verilog tutorial
3. chipverify.com verilog tutorial
4. nptel course playlist on youtube - IIT KGP
5. fpga4fun for projects and examples