Verilog Reference Card

1. Module

```
module module_name (list of ports);
    input / output / inout declarations
    net / reg declarations
    integer declarations
    parameter declarations

    gate / switch / hierarchical instances
    parallel statements
endmodule
```

2. Parallel Statements

Following statements start executing simultaneously inside module

```
initial begin
    {sequential statements} end
always begin
    {sequential statements}
end
assign wire_name ={[expression]};
```

3. Basic Data Types

a. Nets : e.g. **wire, wand, tri, wor**

- Continuously driven
- Gets new value when driver changes
- LHS of continuous assignment

```
tri [15:0] data;                    // unconditional
assign data[15:0] = data_in;    // conditional
assign data[15:0] = enable ? data_in : 16'bz;
```

b. Registers : e.g. **reg**

- Represents storage
- Always stores last assigned value
- LHS of an assignment in procedural block.

```
reg signal;
@(posedge clock) signal = 1'b1; // positive edge
@(reset) signal = 1'b0;  // event (both edges)
```

4. Sequential Statements

```
► if (reset == 0) begin
        data = 8'b00;
  end
► case (operator)
      2'd0 : z = x + y;
      2'd1 : z = x – y;
      2'd2 : z = x * y;
    default : $display ("Invalid Operation");
  endcase
► initial begin            // 50 MHz clock
        clock = 0;
        forever #10 clock = ~clock;
  end                      // precission 1 ns
► repeat (2) @(posedge clk) data;
► bus <= repeat (5) @ (posedge clk) data
      // evaluate data when the assignment is
      // encountered and assign to bus after 5
      // clocks.
► repeat (flag) begin // looping
        .... action ....
► while (i < 10) begin
        .... action ....
  end
► for (i = 0; i < 9; i = i + 1) begin
        .... action ....
  end
► wait (!oe) #5 data = d_in;
► @(negedge clock) q = d;
► begin       // finishes at time #25
        #10 x = y;
        #15 a = b;
  end
► fork
        #10 x = y;
        #15 a = b;
  join
```

5. Gate Primitives

and	(out, in_1, ..., in_n);	**nand**	(out, in_1, ..., in_n);
or	(out, in_1, ..., in_n);	**nor**	(out, in_1, ..., in_n);
xor	(out, in_1, ..., in_n);	**xnor**	(out, in_1, ..., in_n);
buf	(out_1, ..., out_n, in);	**not**	(out_1, ..., out_n, in);
bufif0	(out, in, control);	**bufif1**	(out, in, control);
notif0	(out, in, control);	**notif0**	(out, in, control);
pullup	(out);	**pulldown**	(out);

6. Blocking and Non-blocking Statements

```
// These blocking statements  exhibit  race condition.
always @(posedge clock)   a = b;
always @(posedge clock)   b = a;
// This Non-blocking statement removes above race
// condition and gives true swapping operation
always @(posedge clock)   a <= b;
always @(posedge clock)   b <= a;
```

7. Memory Instantiation

```
module mem_test;
reg [7: 0] memory [0: 10]; // memory declaration
    integer i;
    initial begin
        // reading the memory content file
        $readmemh ("contents.dat", memory);
        // display contents of initialized memory
        for (i = 0; i < 9, i = i + 1)
        $display ("Memory [%d] = %h", i, memory[i]);
    end
    endmodule
```

8. Functions and Tasks

```
function calc_parity;
input [31: 0] address;
  begin   calc_parity = ^address;   end
endfunction
```

Functions must not contain any delay, event, or timing control statements. Functions must have at least one (or more) input argument. Functions always return a single value. They cannot have output or inout argument.