Verilog basics

Comments

`//`	Single line comment
`/* */`	Multiple line comment

Definitions

input	clock
output	something
output reg	something_reg
inout	bidir
input [1:0]	data_bus

Signal and wire declaration

```
wire        wire_name    USE IN ASSIGN STATEMENT
wire [31:0] bus_wire

reg signal                      USE IN ALWAYS BLOCK
reg [31:0] bus_signal
reg [1:0]  bus = 2’b01           Initialized

parameter WIDTH = 8;
reg [WIDTH-1:0] my_bus;
```

Arithmetic operators

+	Addition
-	Subtraction
*	Multiplication
/	Divide
%	Modulus
**	Power Operator

Bitwise operators

~	Invert single bit or each bit vector
&	AND single bits or two bitvectors
|	OR two single bits or two bitvectors
^	XOR two single bits or two bitvect.

Logical operators

==	Inputs Equal
!=	Inputs Not Equal
<	Less-than
<=	Less-than or Equal
>	Greater-than
>=	Greater-than or Equal
!	Not True
&&	Both Inputs True
||	Either Input True

Shit operators

<<	Left shift
>>	Right shift

Bit select, replicate, concatenate and bit-reduction

```
q[3:1] = b[2:0];     Select some bits

q = {a, b, c};       Concatenate a, b and c
q = {3{a}};          Replicate a, 3 times
q = {{2{a}}, b};     Replicate a, 5 times and
                     concatenate to b
f= &a [2:0];         a[0] & a[1] & a[2]
f= |a [2:0];         a[0] | a[1] | a[2]
f= ^a [2:0];         a[0] ^ a[1] ^ a[2]
                      (this is parity)
```

Shifting with bit select & concatenate

```
reg [7:0] shift_l, shift_r;
always @ (posedge clock)
    shift_l <= {shift_l [6:0] , ser_in_l};

always @ (posedge clock)
    shift_r <= {ser_in _r, shift [7:1]};
```

Variable value assignment

```
wire a;
wire [31:0] b;

assign a= 1’b0                 Decimal 0
assign b= 32’b1                Decimal 1
assign b= 32’b00001            Decimal 1
assign b= 52                   Decimal 52
assign b= 32’d52               Decimal 52
assign b= 32’hFF               Decimal 255
assign b=’hFF                  Decimal 255
```

Multiple statements (begin - end)

Only one statement:

```
...
if (reset)
         signal_1 <=value;
else
...
```

Multiple statements:

```
...
if (reset)
      begin
         signal_1 <=value1;
         signal_2 <=value1; //Executed parallel
      end
else
...
```

Simple things

Rule of thumb 1:

Assignment	*Variable type*	*Operator*
assign	**wire**	**=**
always	**reg**	**<=**

Rule of thumb 2:

assign	Combinatorial (logic)
always @ (*****)	Combinatorial (logic)
always @ (**some wire**)	Combinatorial (logic)
always @ (**posedge** clock)	Sequential (register)

Combinatorial assignment

```
wire   wire_name;
assign wire_name = signal_or_value;
```

OR

```
reg  signal;
always @ ( * )
          signal <= signal_or_value;
```

Sequential assignment

```
reg  signal;
always @(posedge clock)
          if (reset)
                    signal <= 0;
          else
                    signal <= signal_or_value;
```

Conditional assignment

```
assign wire_name =
          (condition) ? input1 : input0;
```

OR

```
always @ ( * ) /  always @ (posedge clk)
   if        (condition0)
          statement0;
   else if   (condition1)
          statement1;
   else
          statement2;
```

OR

```
always @ ( * ) /  always @ (posedge clk)
          case (two_bit select)
             2'b00  : statement0;
             2'b01  : statement1;
             2'b10  : statement2;
             2'b11  : statement3;
             default: statement_def;
          endcase
```

Example module declaration

```
module something(
   input         clock,
   input         reset,

   input  [7:0] bus_in,
   output [7:0] bus_out,
);

always @ (posedge clock)
if (reset)
          bus_out <=0;
else
          bus_out <= bus_in;
endmodule
```

Example module instantiation

```
...

wire clock, reset;
wire local_bus_in, local_bus_out;

something inst_name (
  .clock    (clock),
          //module_port_name (local_name)
  .reset    (reset),
  .bus_in   (local_bus_in),
  .bus_out  (local_bus_out)
);
```

Tri state output

```
assign port =
          (enable_signal) ?  signal : 1'bz;
```

Created by lazanyi@mit.bme.hu
v.2.0

Coding examples

D Flip-flop

```
reg       ff;
always @(posedge clk)
if (reset)                    //sync. reset
         ff <= 1'b0;
else
         ff<= new_value;
end
```

Counter

```
reg [3:0] count;
always @(posedge clk)
if (reset)
          count <= 0;
else if (load)
          count<= default_value;
else if (enable)
          count<= count + 1;
```

Serial in, serial out shifter

```
reg [3:0] shift;
wire      ser_out, ser_in;
always @(posedge clk)
if (reset)
          shift<= 4’b0001;
else if (clk_enable)
          shift<={shift[2:0], ser_in};

assign ser_out = shift[3];
```

Parallel in, serial out shifter

```
reg [2:0] shift;
wire      out;

always @(posedge clock)
if (reset)
         shift <= 0;
else if (load)
        shift  <= load_input[2:0];
else if (shift_enable)
        shift <= { shift[1:0] , 1’b0};

assign out = shift[2];
```

Multiplexer

```
reg [3:0]          output;

always @( * )
      case ( select)
         2'b00: output = input1;
         2'b01: output = input2;
         2'b10: output = input3;
         2'b11: output = input4;
       default: output = input1; //Security
      endcase
```

Decoder

```
reg       [3:0] output;
wire      [1:0] select;
wire      enable;
always @( * )
   if (enable)
         case (select)
            2'b00   : output <= 4'b0001;
            2'b01   : output <= 4'b0010;
            2'b10   : output <= 4'b0100;
            2'b11   : output <= 4'b1000;
            default : output <= 4'b0000;
         endcase
   else
          output <=0;
```

OR

```
wire      [3:0] output;
wire      [1:0] select;
wire      enable;
assign output = enable << (select);
```

State Machine

```
parameter state1 = 2'b01;
parameter state2 = 2'b10;
reg  state = state1;

always@(posedge clock) begin
if (reset)                    //sync. reset
      state <= state1;
else
  case (state)
      state1 : if (condition)
                  state <= next_state2;
               else
                  state <= next_state1;

      state2 : if (condition)
                  state <= next_state1;
               else
                  state <= next_state2;

      default : state <= state1;
  endcase
```