**Synthesizable Verilog Code Examples**

**D Type Flip Flops:**

Two things to note about inferring flip flops:

- Non blocking signal assignment \((<=)\) should always be used
- The sensitivity list must have the keyword posedge or negedge. (also for resets)

```verilog
D-type flip flop
reg q;
always @ (posedge clk)
    q <= d;
```

```verilog
d
```

```verilog
clk
```

---

```verilog
D type flip flop with asynchronous reset
reg q;
always @ (posedge clk or posedge reset)
    if (reset)
        q <= 1'b0;
```

```verilog
reset
reset/clk
reset/q
reset/reset
reset/q
```

---

```verilog
reset/d
reset/clk
reset/reset
reset/q
```

---

```verilog
reset/d
reset/clk
reset/q
```
D type flip flop
with synchronous reset

reg q;
always @ (posedge clk)
  if (reset)
    q <= 1'b0;
  else
    q <= d;

D type flip flop
with gated clock

reg q;
wire gtd_clk = enable && clk;
always @ (posedge
Data enabled D type flip flop

```vhdl
reg q;
always @ (posedge clk)
  if (enable)
    q <= d;
```

Negative edge triggered D type flip flop

```vhdl
reg q;
always @ (negedge clk)
  q <= d;
```
Latches

Latch

```verilog
reg q;
always @ (q or enable)
  if (enable)
    q = d;
```

Multiplexers

Two input multiplexer (using if else)

```verilog
reg y;
always @ (a or b or select)
  if (select)
    y = a;
  else
    y = b;
```
Two input multiplexer (using ternary operator ?:)

wire t = (select ? a : b);

Two input multiplexer (using case statement)

reg w;

// mux version 3
always @ (a or b or select)
  case (select)
    1'b1 : w = a;
    default :

Two input multiplexer (using default assignment and if)

reg p;

// mux version 4
always @ (a or b or select)
begin
    p = b;
    if (select)
        p = a;
end

Three input priority encoded mux multiplexer (using if else)

reg q;
always @ (a or b or c or select2)
    if (select2 == 2'b00)
        q = a;
    else
        if (select2 == 2'b01)
            q = b;
        else
            q = c;

Three input priority encoded mux
multiplexer (using case)

reg r;

// Priority encoded mux, version 2
always @ (a or b or c or select2)
    begin
        r = c;
        case (select2)
            2'b00: r = a;
            2'b01: r = b;
        endcase
    end
Three input multiplexer with no priority (using case)

```verilog
reg s;
always @ (a or b or c or select2)
  begin
    case (select2) // synopsys parallel_case
    2'b00: s = a;
    2'b01: s = b;
    default: s = c;
    endcase
  end
```
Comparator (using assign)

module comparator1 (a,b,c);

input  a;
input  b;
output c;

assign c = (a == b);
endmodule

Comparator (using always)

module comparator2 (a,b,c);

input  a;
input  b;
output c;
reg    c;

always @ (a or b)
    if (a == b)
        c = 1'b1;
    else
        c = 1'b0;
endmodule
Finite State Machines
A full example of a state machine and associated test bench

The state machine

The following is a simple for state finite state machine.
The methodology for writing finite state machines is as follows:
1. Draw a state diagram. Label all conditions, label all output values. Label the state encoding.
2. Use parameter to encode the states as in the example.
3. Use two processes or always statements— one sequential and one combinational. See the example.
4. The state machine is normally resettable _ choose synchronous or asynchronous.
5. The combinational process normally has one big case statement in it. Put default values at the beginning.

There are a couple of neat things about the example.
We are using parameters is the test bench and passing them to the state machine using parameter passing
We are using tasks to control the flow of the test bench
We are using hierarchal naming to access the state variable in the state machine from the test bench.
Finally we are using test bench messages which allow us to monitor the current state from the simulation waveform viewer (assuming we change the bus radix of the 'message' to ascii.

// --------------------------------------------
// // STATE MACHINE
//
module state_machine(sm_in, sm_clock, reset, sm_out);

parameter idle = 2'b00;
parameter read = 2'b01;
parameter write = 2'b11;
parameter wait = 2'b10;

input sm_clock;
input reset;
input sm_in;
output sm_out;

reg [1:0] current_state, next_state;

always @(posedge sm_clock)
begin
if (reset == 1'b1)
  current_state <= 2'b00;
else
  current_state <= next_state;
end

always @(current_state or sm_in)
begin
  // default values
  sm_out = 1'b1;
  next_state = current_state;
  case (current_state)
  idle:
    sm_out = 1'b0;
    if (sm_in)
      next_state = 2'b11;
  write:
    sm_out = 1'b0;
    if (sm_in == 1'b0)
      next_state = 2'b10;
  read:
    if (sm_in == 1'b1)
      next_state = 2'b01;
  wait:
    if (sm_in == 1'b1)
      next_state = 2'b00;
  endcase
end
endmodule