Verilog Coding Guidelines
for
High Performance

QUICK REFERENCE GUIDE
Version 1.0

Synopsys, Inc.
700 E. Middlefield Road
Mt.View, CA 94043

Contents

General Purpose Functions

```
function even;
input [31:0] num;
     begin
       even = ~num[0];
     end
endfunction

function mux_2_1;
input sel;
input [1:0]data;
     begin
       if (sel)
         mux_2_1 = data[0];
       else
         mux_2_1 = data[1];
     end
endfunction

function [N-1:0] log2N;
input [31:0] num;

     integer I;
     reg[31:0] M;

     begin
       M = 1;
       for (I=0; I<=31; I=I+1)
       begin : iterate
         if (M >= num)
         begin
           log2N = I;
           disable iterate;
         end
         else
           M = M << 1;
       end
     end
endfunction
```

If Statement
Priority encoded

Original Design

```
module mult_if(a, b_is_late_arriving, c, d, e, sel, z);
input  [3:0] sel;
input    a,
         b_is_late_arriving,
         c,
         d,
         e;
output   z;
reg      z;

     always @ (sel or a or b_is_late_arriving or c or d
                    or e)
     begin
       z = e;
       if (sel[0])
         z = a;
       if (sel[1])
         z = b_is_late_arriving;
       if (sel[2])
         z = c;
       if (sel[3])
         z = d;
     end
endmodule
```

Improved Design

```
module mult_if(a, b_is_late_arriving, c, d, e, sel, z);
input  [3:0] sel;
input    a,
         b_is_late_arriving,
         c,
         d,
         e;
output   z;
reg      z;

     wire high_priority_cond;
     assign high_priority_cond = (sel[2] | sel[3]);

     always @ (sel or a or b_is_late_arriving or c or d
                   or e or high_priority_cond)
     begin
       if (sel[1] & ~high_priority_cond)
         z = b_is_late_arriving;
       else
       begin
         z = e;
         if (sel[0])
           z = a;
         if (sel[2])
           z = c;
         if (sel[3])
           z = d;
       end
     end
endmodule
```

late arrival signal must be moved as closest at the output as possible

If Statement
Parallel encoded

Original Design

```
module single_if(A, C, CTRL_is_late_arriving, Z);
input    [6:1] A;
input    [5:1] C;
input          CTRL_is_late_arriving;
output   Z;
//
reg       Z;

    always @(C or A or CTRL_is_late_arriving)
    begin
      if (C[1] == 1'b1)
        Z = A[1];
      else if (C[2] == 1'b0)
        Z = A[2];
      else if (C[3] == 1'b1)
        Z = A[3];
      else if (C[4] == 1'b1 &&
               CTRL_is_late_arriving == 1'b0)
        Z = A[4];
      else if (C[5] == 1'b0)
        Z = A[5];
      else
        Z = A[6];
    end
endmodule
```

Improved Design

```
module single_if(A, C, CTRL_is_late_arriving, Z);
input    [6:1]   A;
input    [5:1]   C;
input    CTRL_is_late_arriving;
output   Z;
reg      Z;
//
     reg    Z1;
     wire   Z2;
     wire   prev_cond;
//
     always @(C or A)
     begin
       if (C[1] == 1'b1)
         Z1 = A[1];
       else if (C[2] == 1'b0)
         Z1 = A[2];
       else if (C[3] == 1'b1)
         Z1 = A[3];
       else if (C[5] == 1'b0)
         Z1 = A[5];
       else
         Z1 = A[6];
     end

     assign Z2 = A[4];
     assign prev_cond = (C[1] == 1'b1) ||
                        (C[2] == 1'b0) || (C[3] == 1'b1);

     always @(C or A or prev_cond or
              CTRL_is_late_arriving or Z1 or Z2)
     begin
       if (C[4] == 1'b1 &&
                          CTRL_is_late_arriving == 1'b0)
         if (prev_cond)
           Z = Z1;
         else
           Z = Z2;
       else
          Z = Z1;
     end
endmodule
```

If Statement
Cascaded Select Logic

Original Design

```
module ctrl_cascade(sel_is_late_arriving,
                    A,
                    B,
                    C,
                    D,
                    Z);
input    sel_is_late_arriving;
input    A, B, C, D;
output   Z;
reg      Z;

    reg i_sel;

    always @(sel_is_late_arriving or A or B or C or D or
                                          i_sel)
    begin
      if (sel_is_late_arriving)
        i_sel = C;
      else
        i_sel = D;
      if (i_sel)
        Z = A;
      else
        Z = B;
    end
endmodule
```

Improved Design

```
module ctrl_cascade(sel_is_late_arriving,
                    A,
                    B,
                    C,
                    D,
                    Z);
input    sel_is_late_arriving;
input    A, B, C, D;
output   Z;
reg      Z;

    reg C_Z, D_Z;

    always @(sel_is_late_arriving or A or B or C or D
                                     or C_Z or D_Z)
    begin
      if (C)
        C_Z = A;
      else
        C_Z = B;

      if (D)
        D_Z = A;
      else
        D_Z = B;

      if (sel_is_late_arriving)
        Z = C_Z;
      else
        Z = D_Z;
    end
endmodule
```

Logic Replication
Pre-computing Values

Original Design

```
module reorder(addr, sel_is_late_arriving, P1, P2, Z);
parameter       N = 8;
input  [N-1:0]  addr;
input           sel_is_late_arriving;
input  [N-1:0]  P1, P2;
output [N-1:0]  Z;

    reg[N-1:0] temp;

    always @(P1 or P2 or sel_is_late_arriving)
    begin
    if (sel_is_late_arriving)
      temp = P1;
    else
      temp = P2;
    end

    comp U1(addr, temp, Z);

endmodule
```

Improved Design

```
module reorder(addr, sel_is_late_arriving, P1, P2, Z);
parameter       N = 8;
input  [N-1:0]  addr;
input           sel_is_late_arriving;
input  [N-1:0]  P1, P2;
output [N-1:0]  Z;

    wire[N-1:0] Z1, Z2;

    comp U1(addr, P1, Z1);
    comp U2(addr, P2, Z2);

    assign Z = sel_is_late_arriving ? Z1 : Z2;

endmodule
```

Case Statement
Embedded If

Original Design

```
module case_with_if(sel,
                    X,
                    A,
                    B,
                    C_is_late_arriving,
                    D_is_late_arriving,
                    Z);
input  [2:0]  sel;
input         X;
input         A,
              B,
              C_is_late_arriving,
              D_is_late_arriving;
output        Z;
reg           Z;

    always @(sel or X or A or B or C_is_late_arriving or
                    D_is_late_arriving)
    begin
      case (sel)
        3'b000:    Z = A;
        3'b001:    Z = B;
        3'b010:
          if (X)
            Z = C_is_late_arriving;
          else
            Z = D_is_late_arriving;
        3'b100:    Z = A ^ B;
        3'b101:    Z = A & B;
        3'b111:    Z = ~A;
        default:   Z = ~B;
      endcase
    end
endmodule
```

Improved Design

```
module case_with_if(sel,
                    X,
                    A,
                    B,
                    C_is_late_arriving,
                    D_is_late_arriving,
                    Z);
input  [2:0]  sel;
input         X;
input         A,
              B,
              C_is_late_arriving,
              D_is_late_arriving;
output        Z;
reg           Z;

    reg  Z1;

    always @(sel or A or B)
    begin
      case (sel)
        3'b000:  Z1 = A;
        3'b001:  Z1 = B;
        3'b100:  Z1 = A ^ B;
        3'b101:  Z1 = A & B;
        3'b111:  Z1 = ~A;
        default: Z1 = ~B;
      endcase
    end

    always @(Z1 or sel or X or C_is_late_arriving or
             D_is_late_arriving)
    begin
      if (sel == 3'b010)
        if (X)
          Z = C_is_late_arriving;
        else
          Z = D_is_late_arriving;
      else
        Z = Z1;
    end
endmodule
```

Case Statement
Late-arriving Selector

Original Design

```
module split_case(sel0, sel1_is_late_arriving, sel2,
                  X, A, B, C, D, Z);
input  sel0;
input  sel1_is_late_arriving;
input  sel2;
input  X;
input  A,
       B,
       C,
       D;
output Z;
reg    Z;

    wire[2:0]sel = {sel2, sel1_is_late_arriving, sel0};

    always @(sel or X or A or B or C or D)
    begin
      case (sel)
        3'b000:  Z = A;
        3'b001:  Z = B;
        3'b010:
          if (X)
            Z = C;
          else
            Z = D;
        3'b100:  Z = A ^ B;
        3'b101:  Z = A | B;
        3'b111:  Z = ~A;
        default:  Z = ~B;
      endcase
    end
endmodule
```

Improved Design

```
module split_case(sel0, sel1_is_late_arriving, sel2,
                  X, A, B, C, D, Z);
input  sel0;
input  sel1_is_late_arriving;
input  sel2;
input  X;
input  A,
       B,
       C,
       D;
output Z;
reg    Z;

    wire [1:0]  sel = {sel2, sel0};
    reg         Z1, Z2;

    always @(sel or sel1_is_late_arriving or X or A or B
                 or C or D)
    begin
      case (sel)   // sel1_is_late_arriving == 1'b0
        3'b00:  Z1 = A;
        3'b01:  Z1 = B;
        3'b10:  Z1 = A ^ B;
        3'b11:  Z1 = A | B;
        default:  Z1 = ~B;
      endcase

      case (sel)   // sel1_is_late_arriving == 1'b1
        3'b00:
          if (X)
            Z2 = C;
          else
            Z2 = D;
        3'b11:  Z2 = ~A;
        default:  Z2 = ~B;
      endcase

      if (sel1_is_late_arriving)
        Z = Z2;
      else
        Z = Z1;
    end
endmodule
```

Operand Reordering

Original Design

```
module reorder(A_is_late_arriving, B, C, D, Z);
parameter  N = 8;
input  [N-1:0] A_is_late_arriving, B;
input  [N-1:0] C, D;
output [N-1:0] Z;
reg    [N-1:0] Z;

     always @(A_is_late_arriving or B or C or D)
     begin
     if (A_is_late_arriving + B < 24)
       Z = C;
     else
       Z = D;
     end
endmodule
```

Improved Design

```
module reorder(A_is_late_arriving, B, C, D, Z);
parameter  N = 8;
input  [N-1:0] A_is_late_arriving, B;
input  [N-1:0] C, D;
output [N-1:0] Z;
reg    [N-1:0] Z;

    always @(A_is_late_arriving or B or C or D)
    begin
      if (A_is_late_arriving < 24 - B)
        Z = C;
      else
        Z = D;
    end
endmodule
```

Carry-in Speedup

Original Design

```
module carry(A, B, Cin_is_late_arriving, Z);
parameter       N = 8;
input  [N-1:0]  A, B;
input           Cin_is_late_arriving;
output [N-1:0]  Z;
reg    [N-1:0]  Z;

    always @(A or B or Cin_is_late_arriving)
    begin
      Z = A + B + Cin_is_late_arriving;
    end
endmodule
```

Improved Design

```
module carry(A, B, Cin_is_late_arriving, Z);
parameter       N = 8;
input  [N-1:0]  A, B;
input           Cin_is_late_arriving;
output [N-1:0]  Z;
reg    [N-1:0]  Z;

     always @(A or B or Cin_is_late_arriving)
     begin
       if (Cin_is_late_arriving)
         Z = A + B + 1;
       else
         Z = A + B;
     end
endmodule
```

common subexpression sharing

Eliminating Common Subexpressions Reducing Area

Original Design

```
module expressions(A, B, Q, R, S, C1, C2, C3, Z);
parameter      N = 8;
input  [N-1:0] A, B;
input  [N-1:0] Q, R, S;
input  [N-1:0] C1, C2, C3;
output [N-1:0] Z;
reg    [N-1:0] Z;

    always @(A or B or Q or R or S or C1 or C2 or C3)
    begin
      if (C1 > A + B + Q)
        Z = R - S;
      if (C2 > A + B + R)
        Z = S - Q;
      if (C3 > A + B + S)
        Z = Q - R;
    end
endmodule
```

Improved Design

```
module expressions(A, B, Q, R, S, C1, C2, C3, Z);
parameter       N = 8;
input  [N-1:0]  A, B;
input  [N-1:0]  Q, R, S;
input  [N-1:0]  C1, C2, C3;
output [N-1:0]  Z;
reg    [N-1:0]  Z;

     reg [N-1:0]  temp;
     always @(A or B)
     begin
       temp = A + B;
     end

     always @(Q or R or S or C1 or C2 or C3 or temp)
     begin
       if (C1 > temp + Q)
         Z = R - S;
       if (C2 > temp + R)
         Z = S - Q;
       if (C3 > temp + S)
         Z = Q - R;
     end
endmodule
```

Linear Structures
Reduce XOR function

Original Design

```
module xor_chain(A, Z);
parameter       N = 7;
input  [N-1:0]  A;
output          Z;
reg             Z;

    reg      result;
    integer  I;

    always @(A or result)
    begin
      result = A[N-1];
      for (I=N-2; I>=0; I=I-1)
      begin
        result = result ^ A[I];
      end
      Z = result;
    end
endmodule
```

Improved Design

```
module xor_tree(A, Z);
parameter       N = 7;
input  [N-1:0]  A;
output          Z;
reg             Z;

`define logN log2N(N)

    integer      I, J, K, NUM;
    reg  [N-1:0] temp, result;

    always @(A)
    begin
      temp[N-1:0] = A[N-1:0];
      NUM = N;
      for (K=`logN-1; K>=0; K=K-1)
      begin
        J = (NUM+1)/2;
        J = J-1;

        if (even(NUM))
          for (I=NUM-1; I>=0; I=I-2)
          begin
            result[J] = temp[I] ^ temp[I-1];
            J = J-1;
          end
        else
        begin
          for (I=NUM-1; I>=1; I=I-2)
          begin
            result[J] = temp[I] ^ temp[I-1];
            J = J-1;
          end
          result[0] = temp[0];
        end
        temp[N-1:0] = result[N-1:0];
        NUM = (NUM+1)/2;
      end
      Z = result[0];
    end
endmodule
```

Original Design

```
module mux_chain(data, sel, Z);
parameter       N = 9;
input  [N-1:0]  data;
input  [N-2:0]  sel;
output          Z;
reg             Z;

    integer     I;

    always @(sel or data)
    begin
      Z = data[N-1];
      for (I=N-2; I>=0; I=I-1)
      begin
        if (sel[I])
          Z = data[I];
      end
    end
endmodule
```

Improved Design

```
module mux_tree(data, sel, Z);
parameter       N = 9;
input  [N-1:0]  data;
input  [N-2:0]  sel;
output          Z;
reg             Z;

`define logN log2N(N)

    integer    I, J, K, S;
    integer    DEPTH, SEL_LEN, DATA_LEN;

    reg [N-1:0]  i_data, result, i_sel, temp_sel;

    always @(sel or data)
    begin
      i_data[N-1:0] = data[N-1:0];
      i_sel[N-2:0]  = sel[N-2:0];
      i_sel[N-1]= 1'b0;

      DATA_LEN = N;

      for (DEPTH=`logN-1; DEPTH>=0; DEPTH=DEPTH-1)
      begin
        SEL_LEN  = (DATA_LEN+1)/2;
        S = SEL_LEN-1;
        J = (DATA_LEN+1)/2;
        J = J-1;

        if (even(DATA_LEN))
          for (I=DATA_LEN-1; I>=1; I=I-2)
          begin
            result[J] = mux_2_1(i_sel[I-1],
                                {i_data[I], i_data[I-1]});
            temp_sel[S] = |{i_sel[I-1], i_sel[I]};
            J = J-1; S = S-1;
          end
        else
        begin
          for (I=DATA_LEN-1; I>=2; I=I-2)
          begin
            result[J] = mux_2_1(i_sel[I-1],
                                {i_data[I], i_data[I-1]});
            temp_sel[S] = |{i_sel[I-1], i_sel[I]};
            J = J-1; S = S-1;
          end
          result[0] = i_data[0];
          temp_sel[0] = i_sel[0];
        end
        i_data[N-1:0] = result[N-1:0];
        i_sel[N-1:0]  = temp_sel[N-1:0];
        DATA_LEN = (DATA_LEN+1)/2;
      end
      Z = result[0];
    end
endmodule
```