My first program on D flip flop

```
// code for dff
module Dff(input d,input clk,output reg q);
  always @(posedge clk) // note: lines whithin the always block are executed sequententialy
  begin
  q&amp;lt;=d;
  end
endmodule
// code ends
```

D flip flop as we all know is a one bit storage device and its used to model sequential circuits.

REGISTER

Few d flip flops can be used as an array to create a register. Well u can directly use the data type "reg" but i provide this code as its explains the use of structural coding . You can alter the size of the register to increase the size for the code below

```
//code for
module reg4(input [3:0] d,input clk,output [3:0] q);
Dff r1(d[0],clk,q[0]); // this process of calling a module is called instantiating
Dff r2(d[1],clk,q[1]);
Dff r3(d[2],clk,q[2]);
Dff r4(d[3],clk,q[3]);
endmodule
// code ends
```

Traffic light controller using task

```
// code foe traffic light controller
module traffic_lights;
reg clock, red, amber, green;
parameter on = 1, off = 0, red_tics = 30,
amber_tics = 3, green_tics = 20;

initial red = off;
initial amber = off;
initial green = off;

always begin // sequence to control the lights.
red = on; // turn red light on
light(red, red_tics); // and wait.
```

```
green = on; // turn green light on
light(green, green_tics); // and wait.
amber = on; // turn amber light on
light(amber, amber_tics); // and wait.
end
// task to wait for tics positive edge clocks
// before turning color light off.
task light;
output color;
input [31:0] tics;
begin
repeat (tics) @ (posedge clock);
color = off; // turn light off.
end
endtask

always begin // waveform for the clock.
#10 clock = 0;
#10 clock = 1;
end
endmodule // traffic_lights.
// code ends
```

32 bit booth multiplier

```
module mul32(x,y,z);
    input signed[31:0] x;
    input signed[31:0] y;
    output  signed [63:0] z;
 reg  signed[63:0] z;
reg [1:0] temp;
 reg e;
 reg [31:0] y1;
 integer i;
 always@(x,y)
 begin
 e=1'd0;
 z=64'd0;
 for(i=0;i&lt;32 data-blogger-escaped-i="i+1)&lt;/font"&gt;
   begin
 temp={x[i],e};
 y1=-y;
```

```
 case(temp)
 2'd2:z[63:32]=z[63:32]+y1;
 2'd1:z[63:32]=z[63:32]+y;
 default:begin
 end

 endcase
 z=z &amp;gt;&amp;gt;1;
 z[63]=z[62];
 e=x[i];
 end
 if(y==32'd2147483648)
   z=-z;
    end
 endmodule
```

32 bit carry look ahead adder

```
// code starts here
module cla32( d1,d2,clk,cin,sum,cout);
    input [31:0] d1;
    input [31:0] d2;
    input clk;
    input cin;
    output cout;
    output [31:0] sum;

  wire c0,c1,c2,c3,c4,c5,c6;
  reg [31:0] b;
  always@(posedge clk)
  begin
    if(cin==1)
  b&amp;lt;=-d2-1;
  else
  b&amp;lt;=d2;
  end
 cla4 n1(d1[3:0],b[3:0],clk,cin,sum[3:0],c0);
 cla4 n2(d1[7:4],b[7:4],clk,c0,sum[7:4],c1);
 cla4 n3(d1[11:8],b[11:8],clk,c1,sum[11:8],c2);
 cla4 n4(d1[15:12],b[15:12],clk,c2,sum[15:12],c3);
 cla4 n5(d1[19:16],b[19:16],clk,c3,sum[19:16],c4);
 cla4 n6(d1[23:20],b[23:20],clk,c4,sum[23:20],c5);
```

```
 cla4 n7(d1[27:24],b[27:24],clk,c5,sum[27:24],c6);
 cla4 n8(d1[31:28],b[31:28],clk,c6,sum[31:28],cout);
 endmodule

//clad 4 module
module cla4(a,b,cin,s,cout);
input[3:0] a,b;
input cin;
output cout;
output[3:0] s;
wire[3:0] g,p;
wire[13:0] z;

xor21 x1 (.a1(a[0]),.a2(b[0]),.z(p[0]));
and21 x2 (.a1(a[0]),.a2(b[0]),.z(g[0]));
xor21 x3 (.a1(a[1]),.a2(b[1]),.z(p[1]));
and21 x4 (.a1(a[1]),.a2(b[1]),.z(g[1]));
xor21 x5 (.a1(a[2]),.a2(b[2]),.z(p[2]));
and21 x6 (.a1(a[2]),.a2(b[2]),.z(g[2]));
xor21 x7 (.a1(a[3]),.a2(b[3]),.z(p[3]));
and21 x8 (.a1(a[3]),.a2(b[3]),.z(g[3]));

xor21 x9 (.a1(cin),.a2(p[0]),.z(s[0]));
and21 x10 (.a1(cin),.a2(p[0]),.z(z[0]));
or21 x11 (.a1(z[0]),.a2(g[0]),.z(z[1]));
xor21 x12 (.a1(z[1]),.a2(p[1]),.z(s[1]));

and31 x13 (.a1(cin),.a2(p[0]),.a3(p[1]),.z(z[2]));
and21 x14 (.a1(g[0]),.a2(p[1]),.z(z[3]));
or31 x15 (.a1(z[2]),.a2(z[3]),.a3(g[1]),.z(z[4]));
xor21 x16 (.a1(z[4]),.a2(p[2]),.z(s[2]));

and41 x17 (.a1(cin),.a2(p[0]),.a3(p[1]),.a4(p[2]),.z(z[5]));
and31 x18 (.a1(g[0]),.a2(p[1]),.a3(p[2]),.z(z[6]));
and21 x19 (.a1(g[1]),.a2(p[2]),.z(z[7]));
or41 x20 (.a1(z[5]),.a2(z[6]),.a3(z[7]),.a4(g[2]),.z(z[8]));
xor21 x21 (.a1(z[8]),.a2(p[3]),.z(s[3]));

and41 x22 (.a1(cin),.a2(p[0]),.a3(p[1]),.a4(p[2]),.z(z[9]));
and31 x23 (.a1(g[0]),.a2(p[1]),.a3(p[2]),.z(z[10]));
and21 x24 (.a1(g[1]),.a2(p[2]),.z(z[11]));
or41 x25 (.a1(z[9]),.a2(z[10]),.a3(z[11]),.a4(g[2]),.z(z[12]));
and21 x26 (.a1(z[12]),.a2(p[3]),.z(z[13]));
```

```
or21 x27 (.a1(z[13]),.a2(g[3]),.z(cout));
endmodule

//xor
module xor21(a1,a2,z);
input a1,a2; output z;
reg z;
always@(a1,a2)
begin
z&amp;lt;=a1 ^ a2;
end
endmodule


//and
module and21(
a1,
a2,
z
    );
input a1;
input a2;
output z;
assign z=a1&amp;amp;a2;
endmodule

//or
 module or21(a1,a2,z);
input a1,a2; output z;
reg z;
always@(a1,a2)
begin
 z&amp;lt;=a1 | a2;
end
endmodule

module or31(a1,a2,a3,z);
input a1,a2,a3; output z;
reg z;
always@(a1,a2,a3)
begin
 z&amp;lt;=a1 | a2 | a3;
end
```

```
endmodule

module or41(a1,a2,a3,a4,z);
input a1,a2,a3,a4; output z;
reg z;
always@(a1,a2,a3,a4)
begin
z&amp;lt;=a1 | a2 | a3 | a4;
end
endmodule

//code ends here
```

DSP Butterfly unit

```
//code starts here
module butterfly(
ar,
ai,
// ar+jai is the first number
br,
bi,
// br+jbi is the second number
wr,
wi,
// wr+jwi is the twiddle factor
z1r,
z1i,
z2r,
z2i,
clk,

z1r_c,
z1i_c,
z2r_c,
z2i_c

  );

input [31:0]ar,ai,br,bi;
input [32:0]wr,wi;
input clk;
```

```
output[31:0]z1r,z1i,z2r,z2i,z1r_c,z1i_c,z2r_c,z2i_c;

wire [31:0] x1,x2,x3,x4,z_t_i,z_t_i_c,z_t_r,z_t_r_c;

// multliper stage-4
mul32 s1(br,wr,x1);
mul32 s2(bi,wi,x2);
mul32 s3(br,wi,x3);
mul32 s4(bi,wr,x4);
//adder stage-6
cla32 s5(x3,x4,clk,0,z_t_i,z_t_i_c);//temp2
cla32 s6(x1,x2,clk,1,z_t_r,z_t_r_c);//temp1

cla32 s7(z_t_r,ar,clk,0,z1r,z1r_c);
cla32 s8(z_t_i,ai,clk,0,z1i,z1i_c);

cla32 s9(ar,z_t_r,clk,1,z2r,z2r_c);
cla32 s10(ai,z_t_i,clk,1,z2i,z2i_c);

endmodule
// code ends here
```

HALF ADDER

```
module
halfadder(a,b,sum,carry);
input a,b;
output sum, carry;
wire sum, carry;
assign sum = a^b; // sum bit
assign carry = (a&amp;amp;b) ;
//carry bit
endmodule
```

FULL ADDER

```
module fulladder(a,b,c,sum,carry);
input a,b,c;
output sum,carry;
```

```
wire sum,carry;
assign sum=a^b^c; // sum bit
assign carry=((a&amp;amp;b) | (b&amp;amp;c) | (a&amp;amp;c)); //carry bit
endmodule
```

JK FLIP FLOP

```
`define TICK #2 //Flip-flop time delay 2 units
module jkflop(j,k,clk,rst,q);
input j,k,clk,rst;
output q;
reg q;
always @(posedge clk)begin
if(j==1 &amp;amp; k==1 &amp;amp; rst==0)begin
q &amp;lt;=`TICK ~q; //Toggles
end
else if(j==1 &amp;amp; k==0 &amp;amp; rst==0)begin
q &amp;lt;= `TICK 1; //Set
end
else if(j==0 &amp;amp; k==1)begin
q &amp;lt;= `TICK 0; //Cleared
end
end
always @(posedge rst)begin
q &amp;lt;= 0; //The reset normally has negligible delay and hence ignored.
end

endmodule
```

VEDIC MULTIPLIER

Indians have always proved that it has a great history of great mathematicians . Vedic maths is one of the discovery of Indians. One i read one such algorithm i decided to code one such algorithm and check for its speed and compare it with normal conventional multipliers. Here is a code for a 2 bit vedic multiplier

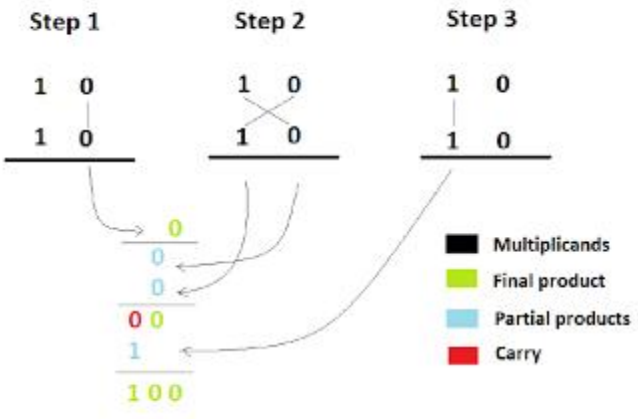
```
// code starts here
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:  Somshekhar R Puranmath
```

```
// Module Name:    vedic_2_x_2
// Target Devices:  spartan 6
// Tool versions: Xilinx 13.3
//////////////////////////////////////////////////////////////////////////////////
module vedic_2_x_2(
a,
b,
c

    );
input [1:0]a;// first input
input [1:0]b;// second input
output [3:0]c;// output
wire [3:0]c;
wire [3:0]temp;
// four multiplication operation of bits according to vedic logic
assign c[0]=a[0]&amp;amp;b[0];
assign temp[0]=a[1]&amp;amp;b[0];
assign temp[1]=a[0]&amp;amp;b[1];
assign temp[2]=a[1]&amp;amp;b[1];
// using two half adders
ha z1(temp[0],temp[1],c[1],temp[3]);
ha z2(temp[2],temp[3],c[2],c[3]);
endmodule

//code for Half adder
module ha(a, b, sum, carry);
// a and b are inputs
input a;
input b;
output sum;
output carry;
assign carry=a&amp;amp;b;
assign sum=a^b;
endmodule
// code ends
```

The above algorithm is according to the URDHVA TIRYAKBHYAM SUTRA. Here is a image illustrating the algorithm



2:4 decoder

```
// code starts here
module dec2_4 (a,b,en,y0,y1,y2,y3)
 input a, b, en;
 output y0,y1,y2,y3;
 assign y0= (~a) &amp;amp; (~b) &amp;amp; en;
 assign y1= (~a) &amp;amp; b &amp;amp; en;
 assign y2= a &amp;amp; (~ b) &amp;amp; en;
 assign y3= a &amp;amp; b &amp;amp; en;
 end module
//                                code                                ends
```

8:3 encoder with priority

```
//code starts here
module enc8_3 (I, en, y, v);
 input [7:0]I;
 input en;
 output v;
 output [2:0]y;
 sig y; sig v;
 always @ (en, I)
 begin
  if(en= =0)
```

```
      v=0;
  else
     v=1;
end
if ( I[7]= =1 &amp;amp; en= =1)                y=3'b111;
else if ( I[6]==1 &amp;amp; en==1)          y=3'b110;
else if ( I[5]==1 &amp;amp; en==1)          y=3'b101;
else if ( I[4]==1 &amp;amp; en==1)          y=3'b100;
else if ( I[3]==1 &amp;amp; en==1)          y=3'b011;
else if ( I[2]==1 &amp;amp; en==1)          y=3'b010;
else if ( I[1]==1 &amp;amp; en==1)          y=3'b001;
else if ( I[0]==1 &amp;amp; en==1)          y=3'b000;
else  y=3'b000;
 end
 endmodule
// code ends here
```

8 TO 1 MULTIPLEXER

```
//code starts here

module mux8_1
input [7:0]I;
output [2:0]S;
output y;
input en;
reg y;
always @(en,S,I,y);
 begin
   if (en= =1)
  begin
 if (s= =000 y=I[0];
 else if (s==001) y=I[1];
 else if (s==001) y=I[2];
else if (s==001) y=I[3];
else if (s==001) y=I[4];
else if (s==001) y=I[5];
else if (s==001) y=I[6];
else if (s==001) y=I[7];
end
else y=0;
```

```
end
end
endmodule
//code                                                                    ends
```

4-BIT BINARY TO GRAY COUNTER CONVERTER

```
//code starts here
module b2g(b,g);
input [3:0] b;
output [3:0] g;
xor (g[0],b[0],b[1]),
        (g[1],b[1],b[2]),
        (g[2],b[2],b[3]);
assign g[3]=b[3];
 endmodule
//code ends
```

DE-MULTIPLEXER (1 TO 4)

```
                                        //code starts here
module demux (s2,s1,I,en,y0,y1,y2,y3)
input s2,s1,I,en;
output y0,y1,y2,y3;
assign y0=(~s2)&amp;amp;(~s1)&amp;amp; I&amp;amp; en;
assign y1=(~s2)&amp;amp; s1&amp;amp; I&amp;amp; en;
assign y2=s2&amp;amp;(~s1)&amp;amp;  I &amp;amp; en;
assign y3=s2&amp;amp; s1 &amp;amp; I &amp;amp; en;
                                                endmodule
//code ends
```

SR FLIPFLOP

```
module srff(s,r,clk,rst, q,qb);
   input s,r,clk,rst;
   output q,qb;
            reg q,qb;
           reg [1:0]sr;
          always@(posedge clk,posedge rst)
```

```
        begin
         sr={s,r};
         if(rst==0)
                  begin
                  case (sr)
                  2'd1:q=1'b0;
                  2'd2:q=1'b1;
           2'd3:q=1'b1;
                  default: begin end
                  endcase
                  end
        else
        begin
                  q=1'b0;
                  end

                  qb=~q;
                  end


endmodule
```

BINARY 4 bit COUNTER(UP/DOWN)

```
module bin_as(clk,clr,dir, temp);
   input clk,clr,dir;
   output reg[3:0] temp;
                          always@(posedge clk,posedge clr)
         begin
         if(clr==0)
         begin
         if(dir==0)
         temp=temp+1;
         else temp=temp-1;
         end
         else
         temp=4'd0;
         end
endmodule
```

Linear feed back shift register (8 bit)

```
module lfsr    (
out             ,  // Output of the counter
enable          ,  // Enable  for counter
clk             ,  // clock input
reset              // reset input
);

//----------Output Ports--------------
output [7:0] out;
//------------Input Ports--------------
input [7:0] data;
input enable, clk, reset;
//------------Internal Variables--------
reg [7:0] out;
wire        linear_feedback;

//-------------Code Starts Here-------
assign linear_feedback = !(out[7] ^ out[3]);

always @(posedge clk)
if (reset) begin // active high reset
  out &amp;lt;= 8'b0 ;
end else if (enable) begin
  out &amp;lt;= {out[6],out[5],
          out[4],out[3],
          out[2],out[1],
          out[0], linear_feedback};
end

endmodule // End Of Module counter
```

Grey counter

```
module gray_counter (
  out    , // counter out
  enable , // enable for counter
  clk    , // clock
  rst      // active hight reset
  );

  //------------Input Ports--------------
  input clk, rst, enable;
  //----------Output Ports---------------
  output [ 7:0] out;
  //------------Internal Variables--------
  wire [7:0] out;
  reg [7:0] count;
  //-------------Code Starts Here---------
  always @ (posedge clk)
  if (rst)
    count &amp;lt;= 0;
  else if (enable)
    count &amp;lt;= count + 1;

  assign out = { count[7], (count[7] ^ count[6]),(count[6] ^
```

```
                count[5]),(count[5] ^ count[4]), (count[4] ^
                count[3]),(count[3] ^ count[2]), (count[2] ^
                count[1]),(count[1] ^ count[0]) };

endmodule
```

UART

```
module uart (
reset           ,
txclk           ,
ld_tx_data      ,
tx_data         ,
tx_enable       ,
tx_out          ,
tx_empty        ,
rxclk           ,
uld_rx_data     ,
rx_data         ,
rx_enable       ,
rx_in           ,
rx_empty
);
// Port declarations
input         reset           ;
input         txclk           ;
input         ld_tx_data      ;
input  [7:0]  tx_data         ;
input         tx_enable       ;
output        tx_out          ;
output        tx_empty        ;
input         rxclk           ;
input         uld_rx_data     ;
output [7:0]  rx_data         ;
input         rx_enable       ;
input         rx_in           ;
output        rx_empty        ;

// Internal Variables
reg [7:0]     tx_reg          ;
reg           tx_empty        ;
reg           tx_over_run     ;
reg [3:0]     tx_cnt          ;
reg           tx_out          ;
reg [7:0]     rx_reg          ;
reg [7:0]     rx_data         ;
reg [3:0]     rx_sample_cnt   ;
reg [3:0]     rx_cnt          ;
reg           rx_frame_err    ;
reg           rx_over_run     ;
reg           rx_empty        ;
reg           rx_d1           ;
reg           rx_d2           ;
reg           rx_busy         ;
```

```
// UART RX Logic
always @ (posedge rxclk or posedge reset)
if (reset) begin
  rx_reg        &amp;lt;= 0;
  rx_data       &amp;lt;= 0;
  rx_sample_cnt &amp;lt;= 0;
  rx_cnt        &amp;lt;= 0;
  rx_frame_err  &amp;lt;= 0;
  rx_over_run   &amp;lt;= 0;
  rx_empty      &amp;lt;= 1;
  rx_d1         &amp;lt;= 1;
  rx_d2         &amp;lt;= 1;
  rx_busy       &amp;lt;= 0;
end else begin
  // Synchronize the asynch signal
  rx_d1 &amp;lt;= rx_in;
  rx_d2 &amp;lt;= rx_d1;
  // Uload the rx data
  if (uld_rx_data) begin
    rx_data  &amp;lt;= rx_reg;
    rx_empty &amp;lt;= 1;
  end
  // Receive data only when rx is enabled
  if (rx_enable) begin
    // Check if just received start of frame
    if (!rx_busy &amp;amp;&amp;amp; !rx_d2) begin
      rx_busy       &amp;lt;= 1;
      rx_sample_cnt &amp;lt;= 1;
      rx_cnt        &amp;lt;= 0;
    end
    // Start of frame detected, Proceed with rest of data
    if (rx_busy) begin
       rx_sample_cnt &amp;lt;= rx_sample_cnt + 1;
       // Logic to sample at middle of data
       if (rx_sample_cnt == 7) begin
          if ((rx_d2 == 1) &amp;amp;&amp;amp; (rx_cnt == 0)) begin
            rx_busy &amp;lt;= 0;
          end else begin
            rx_cnt &amp;lt;= rx_cnt + 1;
            // Start storing the rx data
            if (rx_cnt &amp;gt; 0 &amp;amp;&amp;amp; rx_cnt &amp;lt; 9) begin
              rx_reg[rx_cnt - 1] &amp;lt;= rx_d2;
            end
            if (rx_cnt == 9) begin
               rx_busy &amp;lt;= 0;
               // Check if End of frame received correctly
               if (rx_d2 == 0) begin
                 rx_frame_err &amp;lt;= 1;
               end else begin
                 rx_empty     &amp;lt;= 0;
                 rx_frame_err &amp;lt;= 0;
                 // Check if last rx data was not unloaded,
                 rx_over_run  &amp;lt;= (rx_empty) ? 0 : 1;
               end
            end
          end
```

```
      end
    end
  end
  if (!rx_enable) begin
    rx_busy &amp;lt;= 0;
  end
end

// UART TX Logic
always @ (posedge txclk or posedge reset)
if (reset) begin
  tx_reg        &amp;lt;= 0;
  tx_empty      &amp;lt;= 1;
  tx_over_run   &amp;lt;= 0;
  tx_out        &amp;lt;= 1;
  tx_cnt        &amp;lt;= 0;
end else begin
   if (ld_tx_data) begin
      if (!tx_empty) begin
        tx_over_run &amp;lt;= 0;
      end else begin
        tx_reg   &amp;lt;= tx_data;
        tx_empty &amp;lt;= 0;
      end
   end
   if (tx_enable &amp;amp;&amp;amp; !tx_empty) begin
     tx_cnt &amp;lt;= tx_cnt + 1;
     if (tx_cnt == 0) begin
       tx_out &amp;lt;= 0;
     end
     if (tx_cnt &amp;gt; 0 &amp;amp;&amp;amp; tx_cnt &amp;lt; 9) begin
        tx_out &amp;lt;= tx_reg[tx_cnt -1];
     end
     if (tx_cnt == 9) begin
       tx_out &amp;lt;= 1;
       tx_cnt &amp;lt;= 0;
       tx_empty &amp;lt;= 1;
     end
   end
   if (!tx_enable) begin
     tx_cnt &amp;lt;= 0;
   end
end

endmodule
```

Arbiter implemented with 4 requests

```
module fsm_full(
```

```
clock , // Clock
reset , // Active high reset
req_0 , // Active high request from agent 0
req_1 , // Active high request from agent 1
req_2 , // Active high request from agent 2
req_3 , // Active high request from agent 3
gnt_0 , // Active high grant to agent 0
gnt_1 , // Active high grant to agent 1
gnt_2 , // Active high grant to agent 2
gnt_3   // Active high grant to agent 3
);
// Port declaration here
input clock ; // Clock
input reset ; // Active high reset
input req_0 ; // Active high request from agent 0
input req_1 ; // Active high request from agent 1
input req_2 ; // Active high request from agent 2
input req_3 ; // Active high request from agent 3
output gnt_0 ; // Active high grant to agent 0
output gnt_1 ; // Active high grant to agent 1
output gnt_2 ; // Active high grant to agent 2
output gnt_3 ; // Active high grant to agent

// Internal Variables
reg    gnt_0 ; // Active high grant to agent 0
reg    gnt_1 ; // Active high grant to agent 1
reg    gnt_2 ; // Active high grant to agent 2
reg    gnt_3 ; // Active high grant to agent

parameter  [2:0]  IDLE  = 3'b000;
parameter  [2:0]  GNT0  = 3'b001;
parameter  [2:0]  GNT1  = 3'b010;
parameter  [2:0]  GNT2  = 3'b011;
parameter  [2:0]  GNT3  = 3'b100;

reg [2:0] state, next_state;

always @ (state or req_0 or req_1 or req_2 or req_3)
begin
  next_state = 0;
  case(state)
    IDLE : if (req_0 == 1'b1) begin
        next_state = GNT0;
           end else if (req_1 == 1'b1) begin
        next_state= GNT1;
           end else if (req_2 == 1'b1) begin
        next_state= GNT2;
           end else if (req_3 == 1'b1) begin
        next_state= GNT3;
    end else begin
        next_state = IDLE;
           end
    GNT0 : if (req_0 == 1'b0) begin
        next_state = IDLE;
           end else begin
      next_state = GNT0;
   end
```

```
    GNT1 : if (req_1 == 1'b0) begin
        next_state = IDLE;
           end else begin
      next_state = GNT1;
   end
    GNT2 : if (req_2 == 1'b0) begin
        next_state = IDLE;
           end else begin
      next_state = GNT2;
   end
    GNT3 : if (req_3 == 1'b0) begin
        next_state = IDLE;
           end else begin
      next_state = GNT3;
   end
   default : next_state = IDLE;
  endcase
end

always @ (posedge clock)
begin : OUTPUT_LOGIC
  if (reset) begin
    gnt_0 &amp;lt;= #1 1'b0;
    gnt_1 &amp;lt;= #1 1'b0;
    gnt_2 &amp;lt;= #1 1'b0;
    gnt_3 &amp;lt;= #1 1'b0;
    state &amp;lt;= #1 IDLE;
  end else begin
    state &amp;lt;= #1 next_state;
    case(state)
 IDLE : begin
                gnt_0 &amp;lt;= #1 1'b0;
                gnt_1 &amp;lt;= #1 1'b0;
                gnt_2 &amp;lt;= #1 1'b0;
                gnt_3 &amp;lt;= #1 1'b0;
        end
   GNT0 : begin
            gnt_0 &amp;lt;= #1 1'b1;
          end
        GNT1 : begin
                  gnt_1 &amp;lt;= #1 1'b1;
                end
        GNT2 : begin
                  gnt_2 &amp;lt;= #1 1'b1;
                end
        GNT3 : begin
                  gnt_3 &amp;lt;= #1 1'b1;
                end
     default : begin
                  state &amp;lt;= #1 IDLE;
                end
    endcase
  end
end

endmodule
```

Memory design -ram/rom

Ram

```
module ram1(clk,enable,rdwr,address,dtin,dtout);

//parameter N=10;

input clk,enable,rdwr;

input[9:0] address;

input[15:0] dtin;

output [15:0] dtout;

reg[15:0] dtout;

reg[15:0] memory[0:1023];

//reg[15:0] mem;

always@(posedge clk)

begin

 if(enable==1'b1)

begin

if(rdwr==1'b0)

begin

memory[address]&amp;amp;lt;=dtin;

end
```

```
else

dtout&amp;lt;= memory[address];

end

else

dtout&amp;lt;=16'bz;

end

endmodule
```

Rom

```
module rom1(clk,romrd,address,dtout);

parameter N=8; input[N-2:0] address;

 input clk,romrd; output[15:0] dtout;

reg[15:0] dtout; reg[15:0] romarray[0:(2**N-1)-1];

always@(posedge clk)

begin romarray[0]=16'b0111111111111111;

 romarray[1]=16'b0111011001000010;

 romarray[2]=16'b0101101010000010;

 romarray[3]=16'b0011000011111100;

 romarray[4]=16'b0000000000000000;
```

romarray[5]=16'b1100111100000100;

romarray[6]=16'b1010010101111110;

romarray[7]=16'b1000100110111110; //real twiddle factors

if(romrd==1)

begin

dtout<=romarray[address];

end else

dtout<=16'bz;

end

endmodule

Fibonacci number generator

```
module fibNumberGenNE(startingValue, fibNum);
    input  [15:0] startingValue;
    output [15:0] fibNum;

    reg [15:0] myValue;
    reg [15:0] fibNum;

    always
        begin
            @ng.ready            //accept event signal
                myValue = startingValue;
            for (fibNum = 0; myValue != 0; myValue = myValue - 1)
                fibNum = fibNum + myValue;
            $display ("%d, fibNum=%d", $time, fibNum);
        end
endmodule
```

RS232 transmitter

```
module RS232_Transmitter (
    clock,
  reset_neg,
    tx_datain_ready,
  Present_Processing_Completed,
    tx_datain,
    tx_transmitter,
    tx_transmitter_valid
    );

    parameter HIGH = 1'b1;
    parameter LOW  = 1'b0;

    parameter CLOCK_FREQ = 100000000; // 100MHz
//    parameter CLOCK_FREQ = 33000000; // 33MHz

    parameter BAUD_RATE = 115200; // ;9600

    parameter REG_INPUT = 1; // in REG_INPUT mode, the input doesn't have to
stay valid while the character is been transmitted

    parameter BAUD_ACC_WIDTH = 16;

    input reset_neg;
    input clock;
    input tx_datain_ready;
    input Present_Processing_Completed;
    input [7:0] tx_datain;
    output tx_transmitter;
    output tx_transmitter_valid;

    reg tx_transmitter;

    // Baud generator
    wire [BAUD_ACC_WIDTH:0] Baun_Inc;
    reg [BAUD_ACC_WIDTH:0] Baud_Acc;

assign Baun_Inc = ((BAUD_RATE << (BAUD_ACC_WIDTH - 4)) + (CLOCK_FREQ >> 5))
                   / (CLOCK_FREQ >> 4);

wire Baud_Pulse = Baud_Acc[BAUD_ACC_WIDTH];

always @ (posedge clock or negedge reset_neg)
begin
    if (reset_neg == LOW)
    begin
    Baud_Acc <= {(BAUD_ACC_WIDTH + 1){LOW}};
    end
    else if (Present_Processing_Completed == 1'b1) Baud_Acc <=
{(BAUD_ACC_WIDTH + 1){LOW}};
    else if (tx_transmitter_valid)
    begin
    Baud_Acc <= Baud_Acc[BAUD_ACC_WIDTH - 1:0] + Baun_Inc;
    end
end
```

```
// Transmitter State machine
reg [3:0] State;
wire tx_Xfer_Ready = (State==0);
assign tx_transmitter_valid = ~tx_Xfer_Ready;

reg [7:0] tx_data_reg;
always @ (posedge clock or negedge reset_neg)
begin
    if (reset_neg == LOW)
    begin
    tx_data_reg <= 8'hff;
    end
    else if (Present_Processing_Completed == 1'b1) tx_data_reg <= 8'hff;
    else if (tx_Xfer_Ready & tx_datain_ready)
    begin
    tx_data_reg <= tx_datain;
    end
end

    wire [7:0] Tx_Data_Byte;

assign Tx_Data_Byte = REG_INPUT ? tx_data_reg : tx_datain;

always @ (posedge clock or negedge reset_neg)
begin
    if (reset_neg == LOW)
    begin
    State <= 4'b0000;
    end
    else if (Present_Processing_Completed == 1'b1) State <= 4'b0000;
    else
    begin
    case(State)
        4'b0000: if(tx_datain_ready) State <= 4'b0100;
        // 4'b0001: if(Baud_Pulse) State <= 4'b0100; // registered input
        4'b0100: if(Baud_Pulse) State <= 4'b1000;  // start
        4'b1000: if(Baud_Pulse) State <= 4'b1001;  // bit 0
        4'b1001: if(Baud_Pulse) State <= 4'b1010;  // bit 1
        4'b1010: if(Baud_Pulse) State <= 4'b1011;  // bit 2
        4'b1011: if(Baud_Pulse) State <= 4'b1100;  // bit 3
        4'b1100: if(Baud_Pulse) State <= 4'b1101;  // bit 4
        4'b1101: if(Baud_Pulse) State <= 4'b1110;  // bit 5
        4'b1110: if(Baud_Pulse) State <= 4'b1111;  // bit 6
        4'b1111: if(Baud_Pulse) State <= 4'b0010;  // bit 7
        4'b0010: if(Baud_Pulse) State <= 4'b0000;  // stop1
        // 4'b0011: if(Baud_Pulse) State <= 4'b0000;  // stop2
        default: if(Baud_Pulse) State <= 4'b0000;
    endcase
    end
end

// Output mux
    reg MuxBit;

always @ (State or Tx_Data_Byte)
begin
    case (State[2:0])
```

```
        3'd0: MuxBit <= Tx_Data_Byte[0];
        3'd1: MuxBit <= Tx_Data_Byte[1];
        3'd2: MuxBit <= Tx_Data_Byte[2];
        3'd3: MuxBit <= Tx_Data_Byte[3];
        3'd4: MuxBit <= Tx_Data_Byte[4];
        3'd5: MuxBit <= Tx_Data_Byte[5];
        3'd6: MuxBit <= Tx_Data_Byte[6];
        3'd7: MuxBit <= Tx_Data_Byte[7];
    endcase
end

// Put together the start, data and stop bits
always @ (posedge clock or negedge reset_neg)
begin
    if (reset_neg == LOW)
    begin
    tx_transmitter <= HIGH;
    end
    else if (Present_Processing_Completed == 1'b1) tx_transmitter <= HIGH;
    else
    begin
    tx_transmitter <= (State < 4) | (State[3] & MuxBit); // register the
output to make it glitch free
    end
end

endmodule
```

RS232 receiver

```
module RS232_Receiver (
    reset_neg,
    clock,
    rx_receiver,
    rx_dataout_ready,
    rx_dataout,

    // optional pins for identifying long break in data reception
    rx_endofpacket,
    rx_Idle,
    Exe_LogicImp
    );

    parameter HIGH = 1'b1;
    parameter LOW  = 1'b0;

//`ifdef ML401
    parameter CLOCK_FREQ = 100000000; // 100MHz
//`else // ifdef ML461
//    parameter CLOCK_FREQ = 33000000; // 33MHz
//`endif

    //parameter BAUD_RATE = 9600; // 115200;
    parameter BAUD RATE = 115200; // 115200;

    // BAUD_RATE generator (Uses 8 times oversampling)
    parameter BAUD_RATE_8X = BAUD_RATE*8;
```

```
    parameter BAUD_8X_ACC_WIDTH = 16;

    input reset_neg;
    input clock;
    input rx_receiver;
    input Exe_LogicImp;
    output rx_dataout_ready;
    output [7:0] rx_dataout;

    // optional pins for identifying long break in data reception
    // We also detect if a gap occurs in the received stream of characters
    // That can be useful if multiple characters are sent in burst
    //  so that multiple characters can be treated as a "packet"
    output rx_endofpacket;  // one clock pulse, when no more data
             // is received (rx_Idle is going high)
    output rx_Idle;  // no data is being received

    wire [BAUD_8X_ACC_WIDTH:0] Baud_8X_Incr;
    reg [BAUD_8X_ACC_WIDTH:0] Baud_8X_Acc;

assign Baud_8X_Incr = ((BAUD_RATE_8X << (BAUD_8X_ACC_WIDTH - 7)) +
(CLOCK_FREQ >> 8))
                          / (CLOCK_FREQ >> 7);

always @(posedge clock or negedge reset_neg)
begin
    if (reset_neg == LOW)
    begin
    Baud_8X_Acc <= {(BAUD_8X_ACC_WIDTH + 1){LOW}};
    end
    else if (Exe_LogicImp == 1'b1) Baud_8X_Acc <= {(BAUD_8X_ACC_WIDTH +
1){LOW}};
    else
    begin
    Baud_8X_Acc <= Baud_8X_Acc[BAUD_8X_ACC_WIDTH - 1:0]
                               + Baud_8X_Incr;
    end
end
wire Baud_Pulse_8x = Baud_8X_Acc[BAUD_8X_ACC_WIDTH];

////////////////////////////
    reg [1:0] Rx_Sync;

always @(posedge clock or negedge reset_neg)
begin
    if (reset_neg == LOW)
    begin
        Rx_Sync <= 2'b11;
    end
    else if (Exe_LogicImp == 1'b1) Rx_Sync <= 2'b11;
    else if (Baud_Pulse_8x)
    begin
    Rx_Sync <= {Rx_Sync[0], rx_receiver};
    end
end

    reg [1:0] Rx_Count;
```

```
    reg Rx_Bit;

always @(posedge clock or negedge reset_neg)
begin
    if (reset_neg == LOW)
    begin
     Rx_Count <= 2'b11;
     Rx_Bit <= HIGH;
    end
    else if (Exe_LogicImp == 1'b1)
    begin
     Rx_Count <= 2'b11;
     Rx_Bit <= HIGH;
    end
    else if(Baud_Pulse_8x)
    begin
        if( Rx_Sync[1] && Rx_Count!=2'b11) Rx_Count <= Rx_Count + 2'h1;
        else
        if(~Rx_Sync[1] && Rx_Count!=2'b00) Rx_Count <= Rx_Count - 2'h1;

        if(Rx_Count==2'b00) Rx_Bit <= 1'b0;
        else
        if(Rx_Count==2'b11) Rx_Bit <= 1'b1;
    end
end

    reg [3:0] State;
    reg [3:0] Bit_Spacing;

    // "next_bit" controls when the data sampling occurs
    // depending on how noisy the rx_receiver is, different values might work
better
    // with a clean connection, values from 8 to 11 work
    wire next_bit = (Bit_Spacing==4'd10);

always @(posedge clock or negedge reset_neg)
begin
    if (reset_neg == LOW)
    begin
    Bit_Spacing <= 4'b0000;
    end
    else if (Exe_LogicImp == 1'b1)  Bit_Spacing <= 4'b0000;
    else if(State==0)
    begin
    Bit_Spacing <= 4'b0000;
    end
    else if(Baud_Pulse_8x)
    begin
    Bit_Spacing <= {Bit_Spacing[2:0] + 4'b0001} | {Bit_Spacing[3], 3'b000};
    end
end

always @(posedge clock or negedge reset_neg)
begin
    if (reset_neg == LOW)
    begin
    State <= 4'b0000;
```

```
    end
    else if (Exe_LogicImp == 1'b1)  State <= 4'b0000;
    else if(Baud_Pulse_8x)
    begin
    case(State)
        4'b0000: if(~Rx_Bit) State <= 4'b1000;  // start bit found?
        4'b1000: if(next_bit) State <= 4'b1001;  // bit 0
        4'b1001: if(next_bit) State <= 4'b1010;  // bit 1
        4'b1010: if(next_bit) State <= 4'b1011;  // bit 2
        4'b1011: if(next_bit) State <= 4'b1100;  // bit 3
        4'b1100: if(next_bit) State <= 4'b1101;  // bit 4
        4'b1101: if(next_bit) State <= 4'b1110;  // bit 5
        4'b1110: if(next_bit) State <= 4'b1111;  // bit 6
        4'b1111: if(next_bit) State <= 4'b0001;  // bit 7
        4'b0001: if(next_bit) State <= 4'b0000;  // stop bit
        default: State <= 4'b0000;
    endcase
    end
end

    reg [7:0] rx_dataout;

always @(posedge clock or negedge reset_neg)
begin
    if (reset_neg == LOW)
    begin
    rx_dataout <= 8'h00;
    end
    else if (Exe_LogicImp == 1'b1) rx_dataout <= 8'h00;
    else if(Baud_Pulse_8x && next_bit && State[3])
    begin
    rx_dataout <= {Rx_Bit, rx_dataout[7:1]};
    end
end

    reg rx_dataout_ready, RxD_data_error;

always @(posedge clock or negedge reset_neg)
begin
    if (reset_neg == LOW)
    begin
    rx_dataout_ready <= LOW;
    RxD_data_error <= LOW;
    end
    else if (Exe_LogicImp == 1'b1)
    begin
    rx_dataout_ready <= LOW;
    RxD_data_error <= LOW;
    end
    else
    begin
    rx_dataout_ready <= (Baud_Pulse_8x && next_bit && State==4'b0001 &&
Rx_Bit);  // ready only if the stop bit is received
    RxD_data_error <= (Baud_Pulse_8x && next_bit && State==4'b0001 &&
~Rx_Bit);  // error if the stop bit is not received
    end
end
```

```
// Optional functionality for detection of gap if it occurs in the
// received stream of characters.
    reg [4:0] gap_count;

always @ (posedge clock or negedge reset_neg)
begin
    if (reset_neg == LOW)
    begin
    gap_count<=5'h00;
    end
    else if (Exe_LogicImp == 1'b1) gap_count<=5'h00;
    else if (State!=0)
    begin
    gap_count<=5'h00;
    end
    else if (Baud_Pulse_8x & ~gap_count[4])
    begin
    gap_count <= gap_count + 5'h01;
    end
end

assign rx_Idle = gap_count[4];

    reg rx_endofpacket;

always @(posedge clock or negedge reset_neg)
begin
    if (reset_neg == LOW)
    begin
    rx_endofpacket <= LOW;
    end
    else if (Exe_LogicImp == 1'b1) rx_endofpacket <= LOW;
    else
    begin
    rx_endofpacket <= Baud_Pulse_8x & (gap_count==5'h0F);
    end
end

endmodule
```

Bcd to binary conversion

```
module bcd_to_binary (
  clk_i,
  ce_i,
  rst i,
  start_i,
  dat_bcd_i,
  dat_binary_o,
  done_o
  );
parameter BCD_DIGITS_IN_PP   = 2;  // # of digits of BCD input
parameter BITS_OUT_PP        = 7; // # of bits of binary output
parameter BIT_COUNT_WIDTH_PP = 3;  // Width of bit counter
```

```
// I/O declarations
input  clk_i;                       // clock signal
input  ce_i;                        // clock enable input
input  rst_i;                       // synchronous reset
input  start_i;                     // initiates a conversion
input  [4*BCD_DIGITS_IN_PP-1:0] dat_bcd_i;  // input bus
output [BITS_OUT_PP-1:0] dat_binary_o;      // output bus
output done_o;                      // indicates conversion is done

reg [BITS_OUT_PP-1:0] dat_binary_o;

// Internal signal declarations

reg  [BITS_OUT_PP-1:0] bin_reg;
reg  [4*BCD_DIGITS_IN_PP-1:0] bcd_reg;
wire [BITS_OUT_PP-1:0] bin_next;
reg  [4*BCD_DIGITS_IN_PP-1:0] bcd_next;
reg  busy_bit;
reg  [BIT_COUNT_WIDTH_PP-1:0] bit_count;
wire bit_count_done;

//----------------------------------------------------------------------------
// Functions & Tasks
//----------------------------------------------------------------------------

function [4*BCD_DIGITS_IN_PP-1:0] bcd_asr;
  input [4*BCD_DIGITS_IN_PP-1:0] din;
  integer k;
  reg cin;
  reg [3:0] digit;
  reg [3:0] digit_more;

  begin
    cin = 1'b0;
    for (k=BCD_DIGITS_IN_PP-1; k>=0; k=k-1) // From MS digit to LS digit
    begin
      digit[3] = 1'b0;
      digit[2] = din[4*k+3];
      digit[1] = din[4*k+2];
      digit[0] = din[4*k+1];
      digit_more = digit + 5;
      if (cin)
      begin
        bcd_asr[4*k+3] = digit_more[3];
        bcd_asr[4*k+2] = digit_more[2];
        bcd_asr[4*k+1] = digit_more[1];
        bcd_asr[4*k+0] = digit_more[0];
      end
      else
      begin
        bcd_asr[4*k+3] = digit[3];
        bcd_asr[4*k+2] = digit[2];
        bcd_asr[4*k+1] = digit[1];
        bcd_asr[4*k+0] = digit[0];
      end
      cin = din[4*k+0];
```

```
    end  // end of for loop
  end

endfunction

//-----------------------------------------------------------------------------
// Module code
//-----------------------------------------------------------------------------

// Perform proper shifting, binary ASL and BCD ASL
assign bin_next = {bcd_reg[0],bin_reg[BITS_OUT_PP-1:1]};
always @(bcd_reg)
begin
  bcd_next <= bcd_asr(bcd_reg);
//  bcd_next <= bcd_reg >> 1;  Just for testing...
end

// Busy bit, input and output registers
always @(posedge clk_i)
begin
  if (~rst_i)
  begin
    busy_bit <= 0;  // Synchronous reset
    dat_binary_o <= 0;
  end
  else if (start_i && ~busy_bit)
  begin
    busy_bit <= 1;
    bcd_reg <= dat_bcd_i;
    bin_reg <= 0;
  end
  else if (busy_bit && ce_i && bit_count_done && ~start_i)
  begin
    busy_bit <= 0;
    dat_binary_o <= bin_next;
  end
  else if (busy_bit && ce_i & ~bit_count_done)
  begin
    bin_reg <= bin_next;
    bcd_reg <= bcd_next;
  end
end
assign done_o = ~busy_bit;

// Bit counter
always @(posedge clk_i)
begin
  if (~busy_bit) bit_count <= 0;
  else if (ce_i && ~bit_count_done) bit_count <= bit_count + 1;
end
assign bit_count_done = (bit_count == (BITS_OUT_PP-1));

endmodule
```

CRC-serial /parallel implementation

Crc serial:

```
module crc(clk,din,en,rst,out
    );
input clk,din,en,rst;
output reg[4:0]out;
wire c1;
assign c1= din ^out[4];
always @(posedge clk)
begin
if(~rst)
out<=0;
else if(en)
begin
out[4]<=out[3];
out[3]<=out[2];
out[2]<=c1^out[1];
out[1]<=out[0];
out[0]<=c1;
end end
endmodule
```

Crc parallel :

```
module crcpar(data,crc_en,rst,clk,crc_out);
input crc_en,rst,clk;
input [7:0]data;
output [4:0]crc_out;
reg [4:0]lfsr_q;
reg [4:0]lfsr_c;
assign crc out=lfsr q;
always@(posedge clk)
begin
lfsr_c[0]=lfsr_q[1]^lfsr_q[4]^data[0]^data[3];
lfsr_c[1]=lfsr_q[2]^data[1];
lfsr c[2]=lfsr q[1]^lfsr q[3]^lfsr q[4]^data[0]^data[2]^data[3];
lfsr_c[3]=lfsr_q[2]^lfsr_q[4]^data[1]^data[3];
lfsr_c[4]=lfsr_q[0]^lfsr_q[3]^data[2];
end
always@(posedge clk)
begin
if(~rst)
begin
lfsr_q<=5'b00000;
end
else if(crc_en)
begin
lfsr_q<=lfsr_c;
//lfsr_q<=crc_en?lfsr_c:lfsr_q;
end
end

endmodule
```

Binary to Bcd

```
module bin_bcd(
  clk_i,
  ce_i,
  rst_i,
  start_i,
  dat_binary_i,
  dat_bcd_o,
  done_o
  );
parameter BITS_IN_PP         = 7; // # of bits of binary input
parameter BCD_DIGITS_OUT_PP  = 2;  // # of digits of BCD output
parameter BIT_COUNT_WIDTH_PP = 3;  // Width of bit counter

// I/O declarations
input  clk_i;                      // clock signal
input  ce_i;                       // clock enable input
input  rst_i;                      // synchronous reset
input  start_i;                    // initiates a conversion
input  [BITS_IN_PP-1:0] dat_binary_i;        // input bus
output [4*BCD_DIGITS_OUT_PP-1:0] dat_bcd_o;  // output bus
output done_o;                     // indicates conversion is done

reg [4*BCD_DIGITS_OUT_PP-1:0] dat_bcd_o;

// Internal signal declarations

reg  [BITS_IN_PP-1:0] bin_reg;
reg  [4*BCD_DIGITS_OUT_PP-1:0] bcd_reg;
wire [BITS_IN_PP-1:0] bin_next;
reg  [4*BCD_DIGITS_OUT_PP-1:0] bcd_next;
reg  busy_bit;
reg  [BIT_COUNT_WIDTH_PP-1:0] bit_count;
wire bit_count_done;

//----------------------------------------------------------------------------
// Functions & Tasks
//----------------------------------------------------------------------------

function [4*BCD_DIGITS_OUT_PP-1:0] bcd_asl;
  input [4*BCD_DIGITS_OUT_PP-1:0] din;
  input newbit;
  integer k;
  reg cin;
  reg [3:0] digit;
  reg [3:0] digit_less;
  begin
    cin = newbit;
    for (k=0; k<BCD_DIGITS_OUT_PP; k=k+1)
    begin
      digit[3] = din[4*k+3];
      digit[2] = din[4*k+2];
      digit[1] = din[4*k+1];
      digit[0] = din[4*k];
      digit_less = digit - 5;
      if (digit > 4'b0100)
```

```
      begin
        bcd_asl[4*k+3] = digit_less[2];
        bcd_asl[4*k+2] = digit_less[1];
        bcd_asl[4*k+1] = digit_less[0];
        bcd_asl[4*k+0] = cin;
        cin = 1'b1;
      end
      else
      begin
        bcd_asl[4*k+3] = digit[2];
        bcd_asl[4*k+2] = digit[1];
        bcd_asl[4*k+1] = digit[0];
        bcd_asl[4*k+0] = cin;
        cin = 1'b0;
      end

    end // end of for loop
  end
endfunction

//-----------------------------------------------------------------------------
// Module code
//-----------------------------------------------------------------------------

// Perform proper shifting, binary ASL and BCD ASL
assign bin_next = {bin_reg,1'b0};
always @(bcd_reg or bin_reg)
begin
  bcd_next <= bcd_asl(bcd_reg,bin_reg[BITS_IN_PP-1]);
end

// Busy bit, input and output registers
always @(posedge clk_i)
begin
  if (~rst_i)
  begin
    busy_bit <= 0;  // Synchronous reset
    dat_bcd_o <= 0;
  end
  else if (start_i && ~busy_bit)
  begin
    busy_bit <= 1;
    bin_reg <= dat_binary_i;
    bcd_reg <= 0;
  end
  else if (busy_bit && ce_i && bit_count_done && ~start_i)
  begin
    busy_bit <= 0;
    dat_bcd_o <= bcd_next;
  end
  else if (busy_bit && ce_i && ~bit_count_done)
  begin
    bcd_reg <= bcd_next;
    bin_reg <= bin_next;
  end
end
assign done_o = ~busy_bit;
```

```
// Bit counter
always @(posedge clk_i)
begin
  if (~busy_bit) bit_count <= 0;
  else if (ce_i && ~bit_count_done) bit_count <= bit_count + 1;
end
assign bit_count_done = (bit_count == (BITS_IN_PP-1));
endmodule
```

Hamming code(h,k) encoder /decoder

Encoder:

```
module hamen(clk,d,c
    );
input clk;
input [3:0] d;
output reg[6:0] c;
always@(posedge clk)
begin
  c[6]=d[3];
  c[5]=d[2];
  c[4]=d[1];
  c[3]=d[1]^d[2]^d[3];
  c[2]=d[0];
  c[1]=d[0]^d[2]^d[3];
  c[0]=d[0]^d[1]^d[3];
  end
endmodule
```

Decoder:

```
module hamd(c,clk,s,c2,d
    );
input clk;
input[6:0] c;
output reg[2:0]s;

 output reg[6:0] c2;
 output reg[3:0]d;

always@(posedge clk)
begin
s[2]=c[0]^c[4]^c[5]^c[6];
s[1]=c[1]^c[2]^c[5]^c[6];
s[0]=c[0]^c[2]^c[4]^c[6];

c2=c;
if(s)
c2[s-1]=~c[s-1];

end
always@(c2)
begin
d[0]=c2[2];
d[1]=c2[4];
```

```
d[2]=c2[5];
d[3]=c2[6];
end

endmodule
```

Sequence detector using FSM flow (with output and RTL)

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////
// Company:
// Engineer:
//
// Create Date:    14:58:32 08/16/2013
// Design Name:
// Module Name:    sequence_0111
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////
module sequence_0111(clock, reset, in_bit, out_bit
    );

  input clock, reset, in_bit;
  output out_bit;

  reg [2:0] state_reg, next_state;

  // State declaration
  parameter    reset_state  =  3'b000;
  parameter    read_zero =   3'b001;
  parameter    read_0_one =   3'b010;
  parameter    read_zero_one_one = 3'b011;
  parameter    read_zero_one_one_one= 3'b100;

  // state register
  always @ (posedge clock or posedge reset)
     if (reset == 1)
        state_reg <= reset_state;
     else
        state_reg <= next_state;

  // next-state logic
```

```
always @ (state_reg or in_bit)
    case (state_reg)
        reset_state:
              if (in_bit == 0)
                  next_state = read_zero;
              else if (in_bit == 1)
                  next_state = reset_state;
              else next_state = reset_state;
        read_zero:
              if (in_bit == 0)
                  next_state = read_zero;
              else if (in_bit == 1)
                  next_state = read_0_one;
              else next_state = reset_state;
        read_0_one:
              if (in_bit == 0)
                  next_state = read_zero;
              else if (in_bit == 1)
                  next_state =  read_zero_one_one;
              else next_state = reset_state;
 read_zero_one_one:
            if (in_bit == 0)
                next_state = read_zero;
            else if (in_bit == 1)
                next_state = read_zero_one_one_one;
            else  next_state = reset_state;
      read_zero_one_one_one:
              if (in_bit == 0)
                  next_state = read_zero;
              else if (in_bit == 1)
                  next_state = reset_state;
              else next_state = reset_state;
        default: next_state = reset_state;
  endcase
assign out_bit = (state_reg == read_zero_one_one_one)? 1 : 0;
endmodule

//test bench

`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
///
// Company:
// Engineer:
//
// Create Date:   15:12:38 08/16/2013
// Design Name:   sequence_0111
// Module Name:   D:/verilogggggg/sequence_counter_0111/test_ma_sequence.v
// Project Name:  sequence_counter_0111
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: sequence_0111
//
// Dependencies:
```

```
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
///

module test_ma_sequence;

	// Inputs
	reg clock;
	reg reset;
	reg in_bit;

	// Outputs
	wire out_bit;

	// Instantiate the Unit Under Test (UUT)
	sequence_0111 uut (
		.clock(clock),
		.reset(reset),
		.in_bit(in_bit),
		.out_bit(out_bit)
	);

	initial begin
		// Initialize Inputs
		clock = 0;
		reset = 0;
		in_bit = 0;

		// Wait 100 ns for global reset to finish
		#10;
		reset = 1;
		in_bit = 0;
		#10;

		reset = 0;
		in_bit = 0;
		#10;

		  reset = 0;
		in_bit = 1;
		#10;
		reset = 0;
		in_bit = 1;
		#10;
		reset = 0;
		in_bit = 1;
		#10;
		// Add stimulus here

		reset = 1;
		in_bit = 1;
		#10;
	reset = 0;
```

```
                in_bit = 0;
                #10;
reset = 0;
                in_bit = 1;
                #10;
reset = 0;
                in_bit = 1;
                #10;
reset = 0;
                in_bit = 1;
                #10;
        end
 always begin #5 clock=~clock; end
endmodule

//output waveform
```