Verilog:
Examples of Basic Components

Dr. Bassam Jamil

Topics

- Combinational Designs
  - Mux/Dec/Encoder
- Sequential Designs
  - FFs/Latches/Counters/Shifters

Parameterized Mux Designs

default value

```
module mux4 #( parameter WIDTH = 1 )
            ( input[WIDTH-1:0]  a, b, c, d
              input [1:0] sel,
              output[WIDTH-1:0] out );

  wire [WIDTH-1:0] out, t0, t1;

  assign t0  = (sel[1]? c : a);
  assign t1  = (sel[1]? d : b);
  assign out = (sel[0]? t0: t1);
endmodule
```

Parameterization is a good practice for reusable modules

Instantiation Syntax

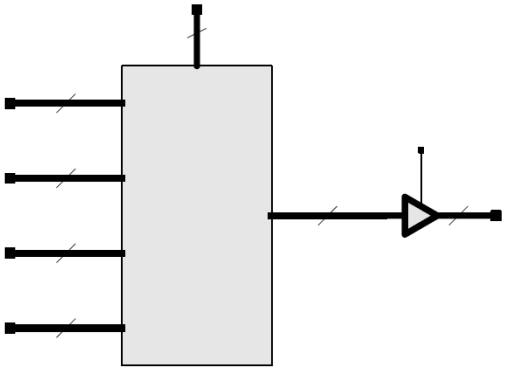
```
mux4#(32) alu_mux
(
   .a (op1),
   .b (op2),
   .c (op3),
   .d (op4),
   .sel (alu_mux_sel),
   .out (alu_mux_out)
```

Parameterized Comparator Design

```
module compare_32_CA (A_gt_B, A_lt_B, A_eq_B, A, B);
  parameter                          word_size = 32;
  input          [word_size-1: 0]    A, B;
  output                             A_gt_B, A_lt_B, A_eq_B;

  assign   A_gt_B = (A > B),         // Note: list of multiple assignments
           A_lt_B = (A < B),
           A_eq_B = (A == B);
endmodule
```

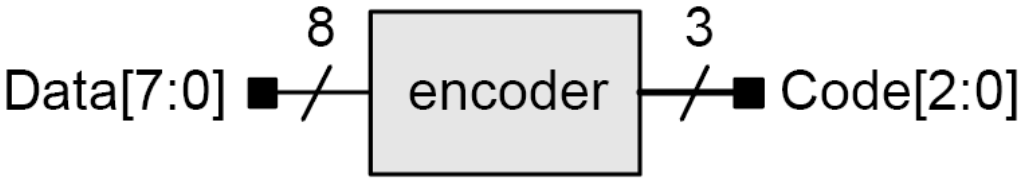
Mux With Tri-State Output

```
module Mux_4_32_case   (mux_out, data_3, data_2, data_1, data_0, select, enable);
      output  [31: 0]     mux_out;
      input   [31: 0]     data_3, data_2, data_1, data_0;
      input   [1: 0]      select;
      input               enable;
      reg     [31: 0]     mux_int;

      assign mux_out = enable ? mux_int : 32'bz;

      always @ ( data_3 or data_2 or data_1 or data_0 or select)
        case (select)
          0:          mux_int = data_0;
          1:          mux_int = data_1;
          2:          mux_int = data_2;
          3:          mux_int = data_3;
          default:    mux_int = 32'bx;          // May execute in simulation
        endcase
    endmodule
```



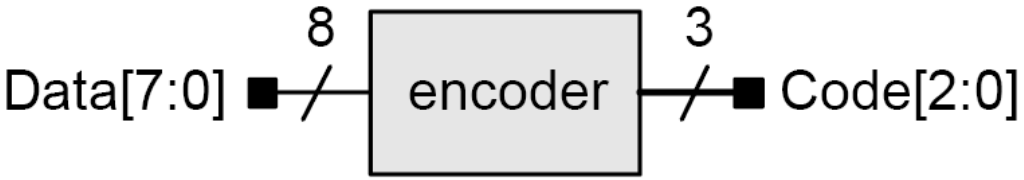
Encoder Design (1)

```
module encoder (Code, Data);
  output        [2: 0] Code;
  input         [7: 0] Data;
  reg           [2: 0] Code;

  always @  (Data)
    begin
      if (Data == 8'b00000001) Code = 0; else
      if (Data == 8'b00000010) Code = 1; else
      if (Data == 8'b00000100) Code = 2; else
      if (Data == 8'b00001000) Code = 3; else
      if (Data == 8'b00010000) Code = 4; else
      if (Data == 8'b00100000) Code = 5; else
      if (Data == 8'b01000000) Code = 6; else
      if (Data == 8'b10000000) Code = 7; else Code = 3'bx;
    end

  endmodule
```



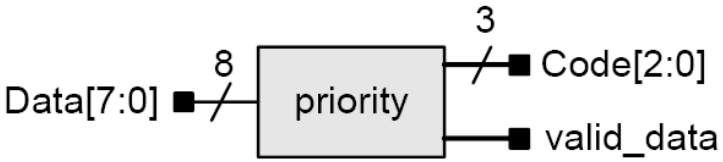
Encoder Design (2)

```
module encoder (Code, Data);
  output        [2: 0] Code;
  input         [7: 0] Data;
  reg           [2: 0] Code;

  always @  (Data)
    case (Data)
        8'b00000001     : Code = 0;
        8'b00000010     : Code = 1;
        8'b00000100     : Code = 2;
        8'b00001000     : Code = 3;
        8'b00010000     : Code = 4;
        8'b00100000     : Code = 5;
        8'b01000000     : Code = 6;
        8'b10000000     : Code = 7;
        default         : Code = 3'bx;
    endcase

  endmodule
```



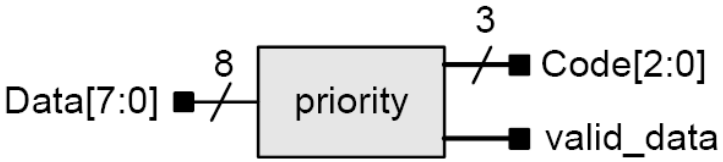
Priority Encoder Design (1)

```
module priority (Code, valid_data, Data);
  output        [2: 0] Code;
  output               valid_data;
  input         [7: 0] Data;
  reg           [2: 0] Code;

  assign        valid_data = |Data; // "reduction or" oper...

  always @  (Data)
   begin
    if (Data[7]) Code = 7; else
    if (Data[6]) Code = 6; else
    if (Data[5]) Code = 5; else
    if (Data[4]) Code = 4; else
    if (Data[3]) Code = 3; else
    if (Data[2]) Code = 2; else
    if (Data[1]) Code = 1; else
    if (Data[0]) Code = 0; else
                 Code = 3'bx;
   end

  endmodule
```



Priority Encoder Design (2)

```
module priority (Code, valid_data, Data);
  output        [2: 0] Code;
  output               valid_data;
  input         [7: 0] Data;
  reg           [2: 0] Code;

  assign        valid_data = |Data; // "reduction or" operator

  always @  (Data)
    casex (Data)
        8'b1xxxxxxx     : Code = 7;
        8'b01xxxxxx     : Code = 6;
        8'b001xxxxx     : Code = 5;
        8'b0001xxxx     : Code = 4;
        8'b00001xxx     : Code = 3;
        8'b000001xx     : Code = 2;
        8'b0000001x     : Code = 1;
        8'b00000001     : Code = 0;
        default         : Code = 3'bx;
    endcase

  endmodule
```



Decoder Design (1)

```
module decoder (Data, Code);
 output        [7: 0] Data;
 input         [2: 0] Code;
 reg           [7: 0] Data;

 always @  (Code)
  begin
   if (Code == 0) Data = 8'b00000001; else
   if (Code == 1) Data = 8'b00000010; else
   if (Code == 2) Data = 8'b00000100; else
   if (Code == 3) Data = 8'b00001000; else
   if (Code == 4) Data = 8'b00010000; else
   if (Code == 5) Data = 8'b00100000; else
   if (Code == 6) Data = 8'b01000000; else
   if (Code == 7) Data = 8'b10000000; else
                  Data = 8'bx;
  end

 endmodule
```

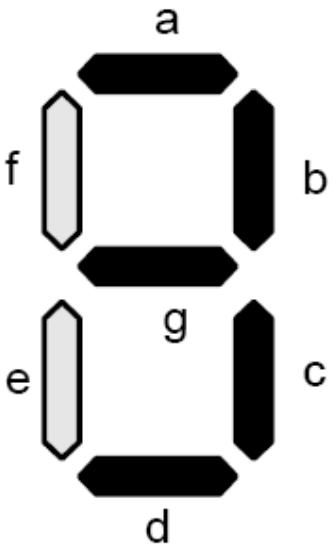
Design (1)

```
module decoder (Data, Code);
 output        [7: 0] Data;
 input         [2: 0] Code;
 reg           [7: 0] Data;

 always @  (Code)
  case (Code)
   0        : Data = 8'b00000001;
   1        : Data = 8'b00000010;
   2        : Data = 8'b00000100;
   3        : Data = 8'b00001000;
   4        : Data = 8'b00010000;
   5        : Data = 8'b00100000;
   6        : Data = 8'b01000000;
   7        : Data = 8'b10000000;
   default  : Data = 8'bx;
  endcase

endmodule
```

Design (2)

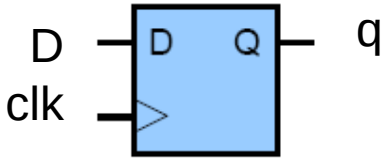
Seven Segment Display

```
module Seven_Seg_Display (Display, BCD);
  output   [6: 0]Display;
  input    [3: 0]BCD;
  reg      [6: 0]Display;
  //                                abc_defg
  parameter   BLANK   = 7'b111_1111;
  parameter   ZERO    = 7'b000_0001;          // h01
  parameter   ONE     = 7'b100_1111;          // h4f
  parameter   TWO     = 7'b001_0010;          // h12
  parameter   THREE   = 7'b000_0110;          // h06
  parameter   FOUR    = 7'b100_1100;          // h4c
  parameter   FIVE    = 7'b010_0100;          // h24
  parameter   SIX     = 7'b010_0000;          // h20
  parameter   SEVEN   = 7'b000_1111;          // h0f
  parameter   EIGHT   = 7'b000_0000;          // h00
  parameter   NINE    = 7'b000_0100;          // h04

  always @ (BCD or)
    case (BCD)
      0:          Display = ZERO;
      1:          Display = ONE;
      2:          Display = TWO;
      3:          Display = THREE;
      4:          Display = FOUR;
      5:          Display = FIVE;
      6:          Display = SIX;
      7:          Display = SEVEN;
      8:          Display = EIGHT;
      9:          Display = NINE;
      default:    Display = BLANK;
    endcase
endmodule
```

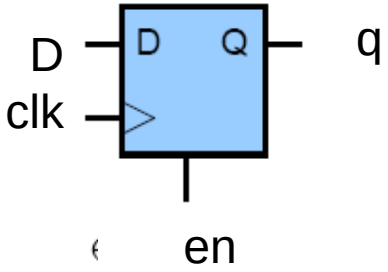


Flip-Flops(1)

```
module FF0 (input clk, input  d,
            output q);
always @( posedge clk )
  begin
     q <= d;
  end
endmodule
```




```
module FF (input clk, input  d,
           input  en, output q);
always @( posedge clk )
  begin
    if ( en )
      q <= d;
  end
endmodule
```



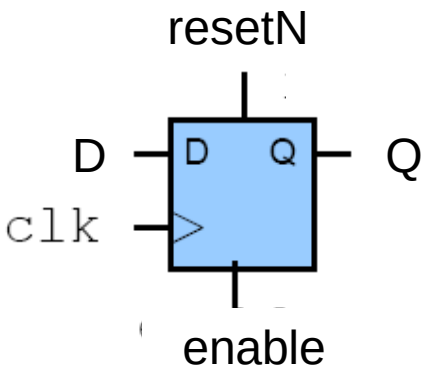
Flip-Flops (2)

```
always @( posedge clk )
begin
  if (~resetN)
    Q <= 0;
  else if ( enable )
    Q <= D;
end
```

synchronous reset

```
always @( posedge clk or
          negedge resetN)
begin
  if (~resetN)
    Q <= 0;
  else if ( enable )
    Q <= D;
end
```

asynchronous reset



Flip-Flip (3)

```
module df_behav (q, q_bar, data, set, reset, clk);
input          data, set, clk, reset;
  output       q, q_bar;
  reg          q;

  assign q_bar = ~ q;

  always @  (posedge clk)  // Flip-flop with synchr
  begin
    if (reset == 0) q <= 0;          // <= is the nonbloc
      else if (set ==0) q <= 1;
        else q <= data;
  end
endmodule
```

```
module asynch_df_behav (q, q_bar, data, set, clk, reset );
  input          data, set, reset, clk;
  output         q, q_bar;
  reg            q;

  assign q_bar = ~q;

  always @  (negedge set or negedge reset or posedge clk)
    begin
      if (reset == 0) q <= 0;
      else if (set == 0) q <= 1;
        else q <= data;              // synchronized activity
    end
endmodule
```

Synchronous Set/Reset FF

Asynchronous Set/Reset FF

Latch Design

```
module Latch_CA (q_out, data_in, enable);
  output        q_out;
  input         data_in, enable;

  assign q_out = enable ? data_in : q_out;
endmodule
```

- Transparent latch does not change the output if the enable (or clk) is off

Register Design

```
module register#(parameter WIDTH = 1)
(
  input  clk,
  input  [WIDTH-1:0] d,
  input  en,
  output [WIDTH-1:0] q
);

  always @( posedge clk )
  begin
    if (en)
      q <= d;
  end

endmodule
```

```
module register2
(
  input  clk,
  input  [1:0] d,
  input  en,
  output [1:0] q
);

 FF ff0 (.clk(clk), .d(d[0]), .en(en),
         .q(q[0]));

 FF ff1 (.clk(clk), .d(d[1]), .en(en),
         .q(q[1]));

endmodule
```

Register with Generate Statement

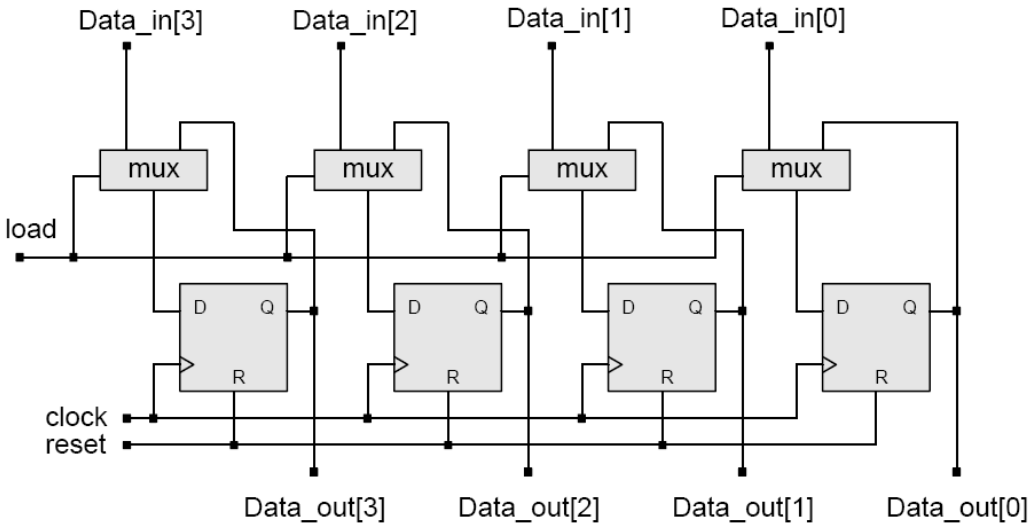
```
module register#(parameter WIDTH = 1)
(
  input  clk,
  input  [WIDTH-1:0] d,
  input  en,
  output [WIDTH-1:0] q
);

 genvar i;
 generate
 for (i =0; i < WIDTH; i = i + 1)
   begin: regE
    FF ff(.clk(clk), .d(d[i]), .en(en), .q(q[i]));
   end
 endgenerate
endmodule
```

genvars disappear after static elaboration

Generated names will have `regE[i].` prefix

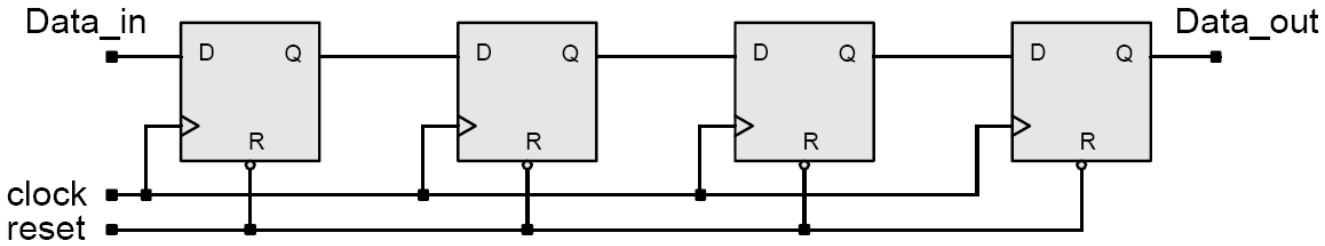
Register Design: Parallel Load

```
module Par_load_reg4 (Data_out, Data_in, load, clock, reset);
  input    [3: 0]     Data_in;
  input               load, clock, reset;
  output   [3: 0]     Data_out;      // Port size
  reg                 Data_out;      // Data type

  always @  (posedge reset or posedge clock)
    begin
      if (reset == 1'b1)              Data_out <= 4'b0;
      else if (load == 1'b1) Data_out <= Data_in;
    end
endmodule
```



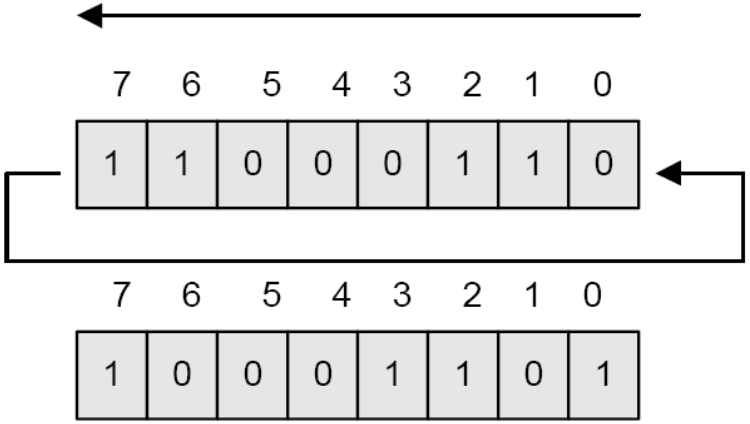
Register Design: Shift Register

```
module Shift_reg4 (Data_out, Data_in, clock, reset);
  output                Data_out;
  input                 Data_in, clock, reset;
  reg       [3: 0]      Data_reg;

  assign   Data_out = Data_reg[0];

  always @  (negedge reset or posedge clock)
    begin
      if (reset == 1'b0)        Data_reg <= 4'b0;
      else                      Data_reg <= {Data_in, Data_reg[3:1]};
    end
endmodule
```



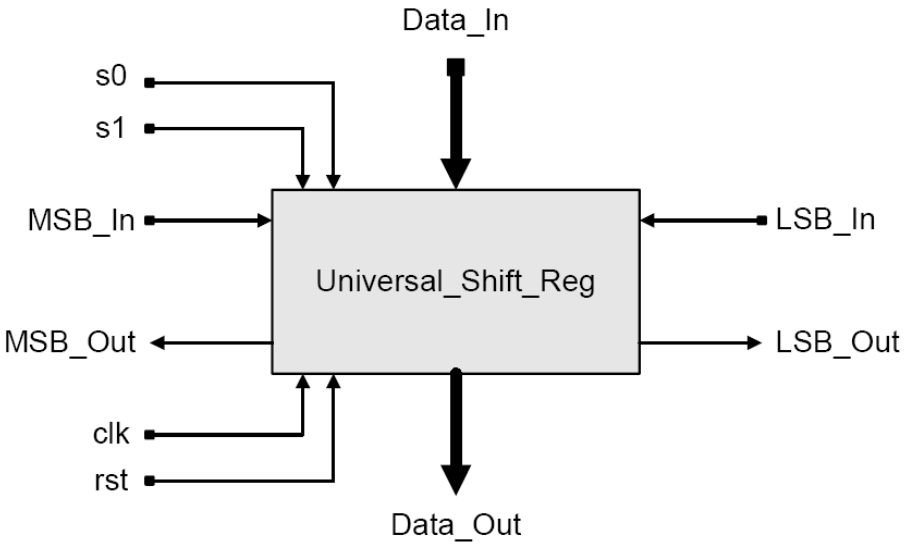
Barrel Shift Register

```
module barrel_shifter (Data_out, Data_in, load, clock, reset);
  output  [7: 0]      Data_out;
  input   [7: 0]      Data_in;
  input               load, clock, reset;
  reg     [7: 0]      Data_out;

  always @  (posedge reset or posedge clock)
    begin
      if (reset == 1'b1)          Data_out <= 8'b0;
      else if (load == 1'b1)      Data_out <= Data_in;
      else                        Data_out <= {Data_out[6: 0], Data_out[7]};
    end
endmodule
```

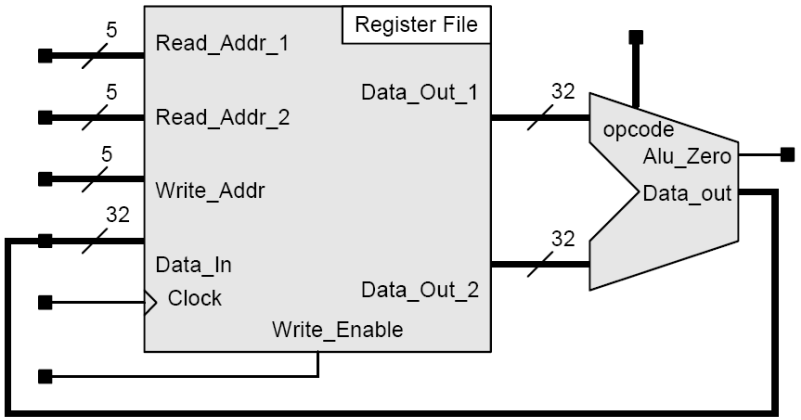


Universal Shift Register

```
module Universal_Shift_Reg
  (Data_Out, MSB_Out, LSB_Out, Data_In, MSB_In, LSB_In, s1, s0, clk, rst);
 output  [3: 0]      Data_Out;
 output              MSB_Out, LSB_Out;
 input   [3: 0]      Data_In;
 input               MSB_In, LSB_In;
 input               s1, s0, clk, rst;
 reg     [3: 0]      Data_Out;

 assign MSB_Out = Data_Out[3];
 assign LSB_Out = Data_Out[0];

 always @ (posedge clk) begin
  if (rst) Data_Out <= 0;
  else  case ({s1, s0})
   0:     Data_Out <= Data_Out;                  // Hold
   1:     Data_Out <= {MSB_In, Data_Out[3:1]};   // Serial shift from MSB
   2:     Data_Out <= {Data_Out[2: 0], LSB_In};  // Serial shift from LSB
   3:     Data_Out <= Data_In;                   // Parallel Load
  endcase
 end
endmodule
```



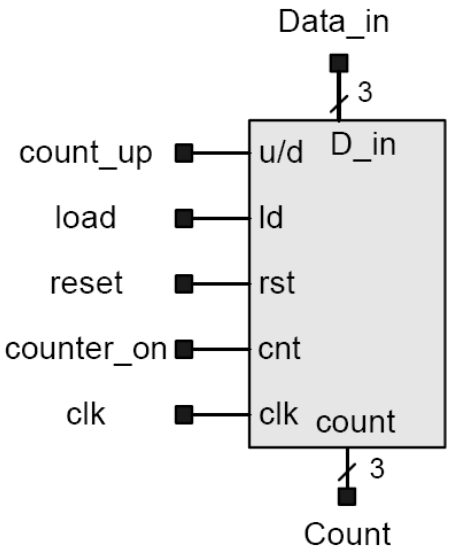
Register File




```
module Register_File  (Data_Out_1, Data_Out_2, Data_in, Read_Addr_1, Read_Addr_2,
  Write_Addr, Write_Enable, Clock);
  output   [31: 0]     Data_Out_1, Data_Out_2;
  input    [31: 0]     Data_in;
  input    [4: 0]      Read_Addr_1, Read_Addr_2, Write_Addr;


  input                Write_Enable, Clock;
  reg      [31: 0]     Reg_File [31: 0];     // 32bit  x32 word memory declaration

  assign Data_Out_1 = Reg_File[Read_Addr_1];
  assign Data_Out_2 = Reg_File[Read_Addr_2];

  always @ (posedge Clock) begin
    if (Write_Enable) Reg_File [Write_Addr] <= Data_in;
  end
endmodule
```

Up-Down Counter with Parallel Load

```
module up_down_counter (Count, Data_in, load, count_up, counter_on, clk, reset);
  output      [2: 0]      Count;
  input                   load, count_up, counter_on, clk, reset,;
  input       [2: 0]      Data_in;
  reg         [2: 0]      Count;

  always @  (posedge reset or posedge clk)
    if (reset == 1'b1) Count = 3'b0; else
      if (load == 1'b1) Count = Data_in; else
        if (counter_on == 1'b1) begin
          if (count_up   == 1'b1) Count = Count +1;
            else Count = Count –1;
      end
endmodule
```



Ring Counter

```
module ring_counter (count, enable, clock, reset);
  output        [7: 0]      count;
  input                     enable, reset, clock;
  reg           [7: 0]      count;

  always @  (posedge reset or posedge clock)
    if (reset == 1'b1)  count <= 8'b0000_0001; else
      if (enable == 1'b1) count <= {count[6: 0], count[7]};
        // Concatenation operator
endmodule
```

count [7:0]

t

0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1