Basic Hardware Module Designs

Alex Yang
Engineering School, NPU

1. Encoder/Decoder

a) Encoder:

0000_0000_0000_0000 => 0000
0000_0000_0000_0001 => 0000
0000_0000_0000_0010 => 0001
0000_0000_0000_0100 => 0010
0000_0000_0000_1000 => 0011
... ...
1000_0000_0000_0000 => 1111

```
`define    kDInBitW      16
`define    kDInBitS      `kDInBitW-1:0
`define    kDOutBitW     4
`define    kDOutBitS     `kDOutBitW -1:0

module enc(
                                dataIn_i,
                                en_i,
                                dataOut_o
);
    input[`kDInBitS]        dataIn_i;
    input                   en_i;
    output[`kDOutBitS]      dataOut_o;

    reg[`kDOutBitS]         dataOut_o;

    always@(*)begin
        if(!en_i)           dataOut_o= 4'b0000;
        else begin
            case(dataIn_i)
                16'h0002:       dataOut_o=4'b0001;
                16'h0004:       dataOut_o=4'b0010;
                16'h0008:       dataOut_o=4'b0011;
                16'h0010:       dataOut_o=4'b0100;
                16'h0020:       dataOut_o=4'b0101;
                16'h0040:       dataOut_o=4'b0110;
                16'h0080:       dataOut_o=4'b0111;
                16'h0100:       dataOut_o=4'b1000;
                16'h0200:       dataOut_o=4'b1001;
                16'h0400:       dataOut_o=4'b1010;
                16'h0800:       dataOut_o=4'b1011;
                16'h1000:       dataOut_o=4'b1100;
                16'h2000:       dataOut_o=4'b1101;
                16'h4000:       dataOut_o=4'b1110;
                16'h8000: dataOut_o=4'b1111;
                default:    dataOut_o=4'bZZZZ;
```

```
                endcase
            end
        end
    endmodule
```

b) Decoder;

```
    0000 => 0000_0000_0000_0001
    0001 => 0000_0000_0000_0010
    0010 => 0000_0000_0000_0100
    0011 => 0000_0000_0000_1000
    0100 => 0000_0000_0001_0000
    ... ...
    1111 => 1000_0000_0000_0000
```

```
    `define   kDInBitW      4
    `define   kDInBitS      `kDInBitW-1:0
    `define   kDOutBitW     16
    `define   kDOutBitS     ` kDOutBitW -1:0

    module dec(
                            dataIn_i,
                            en_i,
                            dataOut_o
    );
        input[`kDInBitS]        dataIn_i;
        input                   en_i;
        output[`kDOutBitS]      dataOut_o;

        reg[`kDOutBitS]         dataOut_o;

        always@(*)begin
            if(!en_i)           dataOut_o= 0;
            else begin
                case(dataIn_i)
                    4'b0000: dataOut_o=`kDOutBitW'h0001;
                    4'b0001: dataOut_o=`kDOutBitW'h0002;
                    4'b0010: dataOut_o=`kDOutBitW'h0004;
                    4'b0011: dataOut_o=`kDOutBitW'h0008;
                    4'b0100: dataOut_o=`kDOutBitW'h0010;
                    4'b0101: dataOut_o=`kDOutBitW'h0020;
                    4'b0110: dataOut_o=`kDOutBitW'h0040;
                    4'b0111: dataOut_o=`kDOutBitW'h0080;
                    4'b1000: dataOut_o=`kDOutBitW'h0100;
                    4'b1001: dataOut_o=`kDOutBitW'h0200;
                    4'b1010: dataOut_o=`kDOutBitW'h0400;
                    4'b1011: dataOut_o=`kDOutBitW'h0800;
                    4'b1100: dataOut_o=`kDOutBitW'h1000;
                    4'b1101: dataOut_o=`kDOutBitW'h2000;
                    4'b1110: dataOut_o=`kDOutBitW'h4000;
                    4'b1111: dataOut_o=`kDOutBitW'h8000;
                    default:   dataOut_o=`kDOutBitW'hZZZZ;
                endcase
```

```
        end
    end
endmodule
```

2. Mux/Demux

a) Mux: 4-to-1

```
module mux4to1(
                    a_i,
                    b_i,
                    c_i,
                    d_i,
                    sel_i,
                    dOut_o
);
    input           a_i;
    input           b_i;
    input           c_i;
    input           d_i;
    input[1:0]      sel_i;
    output          dOut_o;

    assign          dOut_o  =   (sel_i == 0)? a_i:
                                (sel_i == 1)? b_i:
                                (sel_i == 2)? c_i:
                                (sel_i == 0)? d_i: 1'bz;
end
```

b) Demux: 1-to-4

```
module demux4to1(
                        dIn_i,
                        sel_i,
                        dOut_0
);
    input               dIn_i;
    input[1:0]          sel_i;
    output[3:0]         dOut_0;

    reg[3:0]            dOut_0;

    always@(*)begin
        case(sel_i)
            2'b00:  dOut_0[0] = dIn_i;
            2'b01:  dOut_0[1] = dIn_i;
            2'b10:  dOut_0[2] = dIn_i;
            2'b11:  dOut_0[3] = dIn_i;
            default: dOut_0   = 4'bZZZZ;
        endcase
    end
endmodule
```

3. Counter

a) up-down counter with load

```
`define kBitW       8
`define kBitS       `kBitW-1:0
module counter(
                    clk,
                    rst,
                    en_i,
                    load_i,         // Enable starting counting value
                    dataload_i,
                    upDown_i,       // upDown_i = 0, count up
                    dataOut_o
);
    input               clk;
    input               rst;
    input               en_i;
    input               load_i;
    input               upDown_i;
    input[`kBitS]       dataload_i;
    output[`kBitS]      dataOut_o;

    reg[`kBitS]         dataOut_o;


    always@(posedge clk)begin
        if(rst)                         dataOut_o <= 0;
        else if(en_i)begin
            if(load_i)                  dataOut_o <= dataload_i;
            else if(upDown_i)           dataOut_o < = dataOut_o + 1;
            else                        dataOut_o < = dataOut_o - 1;
        end
        else                            dataOut_o <= 0;
    end
endmodule
```

b) Different base up-down counter with load

```
`define kBitW       4
`define kBitS       `kBitW-1:0
`define kBase       13

module counter(
                    clk,
                    rst,
                    en_i,
                    load_i,         // Enable starting counting value
                    dataload_i,
                    upDown_i,       // upDown_i = 1, count up
                    dataOut_o
);
    input               clk;
```

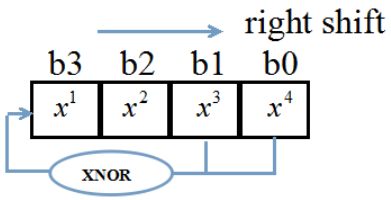
```
    input               rst;
    input               en_i;
    input               load_i;
    input               upDown_i;
    input[`kBitS]       dataload_i;
    output[`kBitS]      dataOut_o;

    reg[`kBitS]         dataOut_o;


    always@(posedge clk)begin
        if(rst)     dataOut_o <= 0;
        else if(en_i)begin
            if(dataOut_o == (`kBase-1))         dataOut_o <= 0;
            else begin
                if(load_i)                  dataOut_o <= dataload_i;
                else if(upDown_i)           dataOut_o < = dataOut_o + 1;
                else                        dataOut_o < = dataOut_o - 1;
            end
        end
        else        dataOut_o <= 0;
    end
endmodule
```

c) Linear Feedback Shift Register(LFSR)

i. Pseudo Random Number Generator

Bits	Feedback polynomial	Period
n		$2^n - 1$
2	$x^2 + x + 1$	3
3	$x^3 + x^2 + 1$	7
4	$x^4 + x^3 + 1$	15
5	$x^5 + x^3 + 1$	31
6	$x^6 + x^5 + 1$	63
7	$x^7 + x^6 + 1$	127
8	$x^8 + x^6 + x^5 + x^4 + 1$	255
9	$x^9 + x^5 + 1$	511
10	$x^{10} + x^7 + 1$	1023
11	$x^{11} + x^9 + 1$	2047
12	$x^{12} + x^{11} + x^{10} + x^4 + 1$	4095
13	$x^{13} + x^{12} + x^{11} + x^8 + 1$	8191

14	$x^{14}+x^{13}+x^{12}+x^{2}+1$	16383
15	$x^{15}+x^{14}+1$	32767
16	$x^{16}+x^{14}+x^{13}+x^{11}+1$	65535
17	$x^{17}+x^{14}+1$	131071
18	$x^{18}+x^{11}+1$	262143
19	$x^{19}+x^{18}+x^{17}+x^{14}+1$	524287



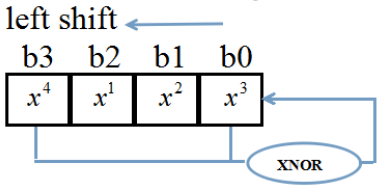
Note: XNOR could allow seed=0 (initial values), seed≠1;
XOR allows seed=1 (initial values), seed≠0

```
`define  kBitW       4
`define  kBitS       `kBitW-1:0
module   lfsrCnt(
                        clk,
                        rst,
                        en_i,
                        cnt_o,
                        bitStr_o        //Overflow bit
);
    input               clk, rst;
    input               en_i;
    output[`kBitS]      cnt_o;

    reg[`kBitS]         cnt_o;

    assign          bitStr_o = cnt_o[0];

    always@(posedge clk)begin
        if(rst)     cnt_o <= 0;
        else if(en_i) begin
                    cnt_o[0] <= cnt_o[1] ;
                    cnt_o[1] <= cnt_o[2];
                    cnt_o[2] <= cnt_o[3];
                    cnt_o[3] <= cnt_o[0] ~^ cnt_o[1];
        end
    end
endmodule
```

ii. Reverse random number generation by inverse LFSR

1) change shift direction from right to left

2) $x^1 - x^2 - x^3 - x^4$ rotate to the right as below




```
`define  kBitW       4
`define  kBitS       `kBitW-1:0
module   lfsrCnt(
                        clk,
                        rst,
                        en_i,
                        cnt_o,
                        bitStr_o        //Overflow bit
);
        input                   clk, rst;
        input                   en_i;
        output[`kBitS]          cnt_o;

        reg[`kBitS]             cnt_o;

        assign          bitStr_o = cnt_o[3];

        always@(posedge clk)begin
                if(rst)     cnt_o <= 0;
                else if(en_i) begin
                        cnt_o[0] <= cnt_o[3] ~^ cnt_o[0];
                        cnt_o[1] <= cnt_o[0];
                        cnt_o[2] <= cnt_o[1];
                        cnt_o[3] <= cnt_o[2] ;
                end
        end
endmodule
```

iii. up-down LFSR counter

```
`define    kBitW        4
`define    kBitS        `kBitW-1:0

module upDownLFSR(
                        clk,
                        rst,
                        en_i,
                        upDown_i,       //upDown_i = 1 -> up; othrwise down
                        cnt_o,
                        bitStr_o        //Overflow bit
);
        input                   clk;
        input                   rst;
        input                   en_i;
        input                   upDown_i;
```

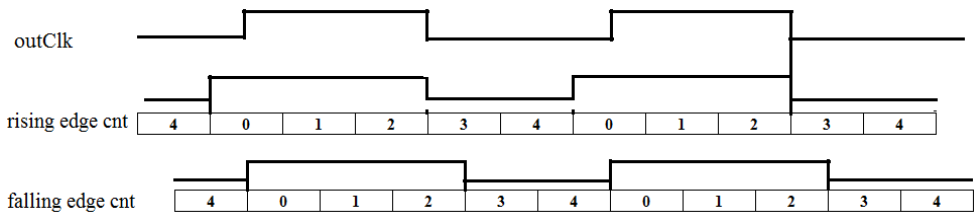
```
    output[`kBitS]          cnt_o;

    reg[`kBitS]             cnt_o;

    assign          bitStr_o = (!en_i)?         1'b0        :
                               (upDown_i)?      cnt_o[0]  : cnt_o[3] ;

    always@(posedge clk)begin
        if(rst)         cnt_o <= 0;
        else if(en_i)begin
            if(upDown_i)begin
                cnt_o[0] <= cnt_o[1] ;
                cnt_o[1] <= cnt_o[2];
                cnt_o[2] <= cnt_o[3];
                cnt_o[3] <= cnt_o[0] ~^ cnt_o[1];
            end
            else begin
                cnt_o[0] <= cnt_o[3] ~^ cnt_o[0];
                cnt_o[1] <= cnt_o[0];
                cnt_o[2] <= cnt_o[1];
                cnt_o[3] <= cnt_o[2] ;
            end
        end
        else        cnt_o <= 0;
    end
endmodule
```

4. Frequency Divider (Don't use in RTL design)

Case1: divided by integer number A, such as 5

Step1: rising/falling edge counter: 0~4 (A-1 = 5-1)

Step2: observe 0~2 (a little bit bigger than 5/2 cycles) and compare two waveforms as follows

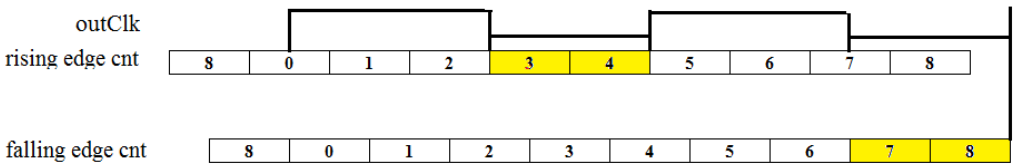


outClk = (rEdgeCnt <3 && fEdgeCnt < 3)? 1'b1: 1'b0

Case2: divided by x.5, such as 4.5 and hi-voltage duty 2.5 cycles /low 2 cycles

Step1: rising/falling edge counter: 0~8 (8= 4.5x2 - 1)

Step2: observe 0.5~2 in rEdgeCnt for hi-voltage duty and low in cycle 3~4 and cycle 7~8 in fEdgeCnt, and then compare two waveforms as follows



outClk = ((rEdgeCnt >2 && rEdgeCnt < 5) || fEdgeCnt >6) ? 1'b0: 1'b1

a) Divided by 3

```
module Div3(
                clk,
                rst,
                clkOut_o
);
    input       clk;
    input       rst;
    output      clkOut_o;

    reg[1:0]    rEdgeCnt_r;
    reg[1:0]    fEdgeCnt_r;

    assign      clkOut_o = ((rEdgeCnt_r != 2) && (fEdgeCnt_r != 2));

    always@(posedge clk)begin
        if(rst)                 rEdgeCnt_r <= 0;
        else if(rEdgeCnt_r == 2)rEdgeCnt_r <= 0;
        else                    rEdgeCnt_r <= rEdgeCnt_r + 1;
    end

    always@(negedge clk)begin
        if(rst)                 fEdgeCnt_r <= 0;
        else if(fEdgeCnt_r == 2)fEdgeCnt_r <= 0;
        else                    fEdgeCnt_r <= fEdgeCnt_r+ 1;
    end
endmodule
```

b) Divided by 5

```
module Div5(
                clk,
                rst,
                clkOut_o
);
    input       clk;
    input       rst;
    output      clkOut_o;

    reg[2:0]    rEdgeCnt_r;
```

```
        reg[2:0]        fEdgeCnt_r;

        assign          outClk = (rEdgeCnt <3 && fEdgeCnt < 3)? 1'b1: 1'b0;

        always@(posedge clk)begin
            if(rst)                     rEdgeCnt_r <= 0;
            else if(rEdgeCnt_r == 4)    rEdgeCnt_r <= 0;
            else                        rEdgeCnt_r <= rEdgeCnt_r + 1;
        end

        always@(negedge clk)begin
            if(rst)                     fEdgeCnt_r <= 0;
            else if(fEdgeCnt_r == 4)    fEdgeCnt_r <= 0;
            else                        fEdgeCnt_r <= fEdgeCnt_r+ 1;
        end
    endmodule
```

c) Divided by 4.5

```
    module Div45(
                    clk,
                    rst,
                    clkOut_o
    );
        input       clk;
        input       rst;
        output      clkOut_o;

        reg[3:0]    rEdgeCnt_r;
        reg[3:0]    fEdgeCnt_r;

        assign      outClk = ((rEdgeCnt >2 && rEdgeCnt < 5) || fEdgeCnt >6 ) ? 1'b0: 1'b1

        always@(posedge clk)begin
            if(rst)                     rEdgeCnt_r <= 0;
            else if(rEdgeCnt_r == 8)    rEdgeCnt_r <= 0;
            else                        rEdgeCnt_r <= rEdgeCnt_r + 1;
        end

        always@(negedge clk)begin
            if(rst)                     fEdgeCnt_r <= 0;
            else if(fEdgeCnt_r == 8)    fEdgeCnt_r <= 0;
            else                        fEdgeCnt_r <= fEdgeCnt_r+ 1;
        end
endmodule
```

5. Edge Detectors

a) Rising edge detector

```
    module      risingDet(
                        clk,
                        rst,
                        dIn_i,              //Synchronized data in
                        risingPulse_o
    );
```

```
    input       clk;
    input       rst;
    input       dIn_i;
    output      risingPulse_o;

    reg         oneDelayDin_r;

    assign      risingPulse_o = dIn_i & ~oneDelayDin_r;

    always@(posedge clk)begin
        if(rst)     oneDelayDin_r <= 0;
        else        oneDelayDin_r <= dIn_i;
    end

endmodule
```

b) Falling edge detector

```
module      fallingDet(
                        clk,
                        rst,
                        dIn_i,
                        fallingPulse_o
);
    input       clk;
    input       rst;
    input       dIn_i;
    output      fallingPulse_o;

    reg         oneDelayDin_r;

    assign      fallingPulse_o = ~dIn_i & oneDelayDin_r;

    always@(posedge clk)begin
        if(rst)     oneDelayDin_r <= 0;
        else        oneDelayDin_r <= dIn_i;
    end
endmodule
```

c) Raising/Falling edge detectors

```
module      edgeDet(
                        clk,
                        rst,
                        dataIn_i,
                        risingPulse_o,
                        fallingPulse_o
);
    input       clk;
    input       rst;
    input       dIn_i;
    output      risingPulse_o;
    output      fallingPulse_o;

    reg         oneDelayDin_r;
```

```
    assign      risingPulse_o =    dIn_i & ~oneDelayDin_r;
    assign      fallingPulse_o = ~dIn_i & oneDelayDin_r;

    always@(posedge clk)begin
        if(rst)     oneDelayDin_r <= 0;
        else        oneDelayDin_r <= dIn_i;
    end
endmodule
```

6. Memory Design (Only for simulation, non-synthesizable)

a) Single port

```
module syncRAM (
                    clk,        // Clock Input
                    address,    // Address Input
                    data,       // Data bi-directional
                    cs,         // Chip Select
                    we,         // Write Enable/Read Enable
                    oe          // Output Enable
);

    parameter pBitW = 8;
    parameter pAdrW= 8 ;
    parameter pRAMDep = 1 << pAdrW;

    input                   clk;
    input [pAdrW-1:0]           address;
    input                   cs;
    input                   we;
    input                   oe;

    inout [pBitW-1:0]   data;

    reg [pBitW-1:0]     dOut_r;
    reg [pBitW-1:0]     mem [0:pRAMDep-1];
    reg                 oe_r;

    // Tri-State Buffer control
    // output : When we = 0, oe = 1, cs = 1
    assign              data = (cs && oe && !we) ? dOut_r : 8'bz;

    // Memory Write Block
    // Write Operation : When we = 1, cs = 1
    always @ (posedge clk)begin
        if ( cs && we ) begin
            mem[address] = data;
        end
    end

    // Memory Read Block
    // Read Operation : When we = 0, oe = 1, cs = 1
    always @ (posedge clk)begin
```

```
        if (cs && !we && oe) begin
            dOut_r = mem[address];
            oe_r = 1;
        end else begin
            oe_r = 0;
        end
    end
endmodule

module asyncRAM (
                clk,
                address,        // Address Input
                data,           // Data bi-directional
                cs,             // Chip Select
                we,             // Write Enable/Read Enable
                oe              // Output Enable
);
    parameter pBitW = 8;
    parameter pAdrW= 8 ;
    parameter pRAMDep = 1 << pAdrW;

    input                   clk;
    input [pAdrW-1:0]       address;
    input                   cs;
    input                   we;
    input                   oe;

    inout [pBitW-1:0]       data;

    reg [pBitW-1:0]         dOut_r;
    reg [pBitW-1:0]         mem [0 : pRAMDep-1];


    // Tri-State Buffer control
    // output : When we = 0, oe = 1, cs = 1
    assign                  data = (cs && oe && !we) ? dOut_r : 8'bz;

    // Memory Write Block
    // Write Operation : When we = 1, cs = 1
    always @ (posedge clk)begin
        if ( cs && we ) begin
            mem[address] = data;
        end
    end

    // Memory Read Block
    // Read Operation : When we = 0, oe = 1, cs = 1
    always @ (address or cs or we or oe)begin
        if (cs && !we && oe)    begin
            dOut_r = mem[address];
        end
    end
endmodule
```

b) Dual port

```
module dualSyncRAM(
                        clk,            // Clock Input
                        address0,       // address0 Input
                        data0,          // data0 bi-directional
                        cs0,            // Chip Select
                        we0,            // Write Enable/Read Enable
                        oe0,            // Output Enable
                        address1,       // address1 Input
                        data1,          // data1 bi-directional
                        cs1,            // Chip Select
                        we1,            // Write Enable/Read Enable
                        oe1             // Output Enable
);

        parameter pDataW = 8 ;
        parameter pAdrW    = 8 ;
        parameter pRAMDep = 1 << pAdrW;

        input [pAdrW-1:0]       address0 ;
        input                   cs0 ;
        input                   we0 ;
        input                   oe0 ;
        input [pAdrW-1:0]       address1 ;
        input                   cs1 ;
        input                   we1 ;
        input                   oe1 ;

        inout [pDataW -1:0]     data0 ;
        inout [pDataW -1:0]     data1 ;

        reg [pDataW -1:0]       dOut0_r ;
        reg [pDataW -1:0]       dOut1_r ;
        reg [pDataW -1:0]       mem [0 : pRAMDep-1];

        // Memory Write Block
        // Write Operation : When we0 = 1, cs0 = 1
        always @ (posedge clk)begin
              if ( cs0 && we0 ) begin
                    mem[address0] <= data0;
              end else if (cs1 && we1) begin
                    mem[address1] <= data1;
              end
        end

        // Tri-State Buffer control
        // output : When we0 = 0, oe0 = 1, cs0 = 1
        assign      data0 = (cs0 && oe0 && !we0) ? dOut0_r : 8'bz;

        // Memory Read Block
        // Read Operation : When we0 = 0, oe0 = 1, cs0 = 1
        always @ (posedge clk)begin
```

```
            if (cs0 && !we0 && oe0) begin
                dOut0_r <= mem[address0];
            end else begin
                dOut0_r <= 0;
            end
        end

        //Second Port of RAM
        // Tri-State Buffer control
        // output : When we0 = 0, oe0 = 1, cs0 = 1

        assign data1 = (cs1 && oe1 && !we1) ? dOut1_r : 8'bz;

        // Memory Read Block 1
        // Read Operation : When we1 = 0, oe1 = 1, cs1 = 1
        always @ (posedge clk)begin
          if (cs1 && !we1 && oe1) begin
            dOut1_r <= mem[address1];
          end else begin
            dOut1_r <= 0;
          end
        end
endmodule

module dualAsyncRAM (
                        address0,       // address0 Input
                        data0,          // data0 bi-directional
                        cs0,            // Chip Select
                        we0,            // Write Enable/Read Enable
                        oe0,            // Output Enable
                        address1,       // address1 Input
                        data1,          // data1 bi-directional
                        cs1,            // Chip Select
                        we1,            // Write Enable/Read Enable
                        oe1             // Output Enable
);

    parameter pDataW = 8 ;
    parameter pAdrW = 8 ;
    parameter pRAMDep = 1 << pAdrW;

    input [pDataW-1:0]      address0 ;
    input                   cs0 ;
    input                   we0 ;
    input                   oe0 ;
    input [pDataW-1:0]      address1 ;
    input                   cs1 ;
    input                   we1 ;
    input                   oe1 ;

    inout [pDataW-1:0]      data0 ;
    inout [pDataW-1:0]      data1 ;
```

```
    reg [pDataW-1:0]        dOut0_r ;
    reg [pDataW-1:0]        dOut1_r ;
    reg [pDataW-1:0]        mem [0 : pRAMDep-1];

    // Memory Write Block
    // Write Operation : When we0 = 1, cs0 = 1
    always @ (address0 or cs0 or we0 or data0 or address1 or cs1 or we1 or data1)begin
        if ( cs0 && we0 ) begin
            mem[address0] <= data0;
        end else if    (cs1 && we1) begin
            mem[address1] <= data1;
        end
    end

    // Tri-State Buffer control
    // output : When we0 = 0, oe0 = 1, cs0 = 1

    assign data0 = (cs0 && oe0 && !we0) ? dOut0_r : 8'bz;

    // Memory Read Block
    // Read Operation : When we0 = 0, oe0 = 1, cs0 = 1
    always @ (address0 or cs0 or we1 or oe0)begin
        if (cs0 && !we0 && oe0) begin
            dOut0_r <= mem[address0];
        end else begin
            dOut0_r <= 0;
        end
    end

    //Second Port of RAM
    // Tri-State Buffer control
    // output : When we0 = 0, oe0 = 1, cs0 = 1

    assign data1 = (cs1 && oe1 && !we1) ? dOut1_r : 8'bz;

    // Memory Read Block 1
    // Read Operation : When we1 = 0, oe1 = 1, cs1 = 1
    always @ (address1 or cs1 or we1 or oe1)begin
        if (cs1 && !we1 && oe1) begin
            dOut1_r <= mem[address1];
        end else begin
            dOut1_r <= 0;
        end
    end
endmodule
```

7. Parity & CRC

a) Parity

i Even parity bit = 0 if # of 1's in input data is even; Even parity bit = 1 if # of 1's is odd.

ii Odd parity bit = 0 if # of 1's is odd; Even parity bit = 1 if # of 1's is even.

iii Even/Odd Parity Bit Generation Verilog Code:

```
`define         kBitW       8
`define         kBitS       `kBitW-1:0

module    parityBitGen(
                              dataIn_i,
                              dataOut_o
);
        input[`kBitW-2:0]     dataIn_i;
        output[`kBitS]        dataOut_o;

        assign                dataOut_o = (^dataIn_i)? {1, dataIn_i} : {0, dataIn_i}; //Even parity
        //assign              dataOut_o = (^dataIn_i)? {0, dataIn_i} : {1, dataIn_i}; //Odd parity

endmodule
```

b) CRC (cyclic redundancy check)

i. Serial CRC encoder:

e.g. dataIn = 1010

If g(x) = x^3+x+1, dataOut(with CRC) = 1010 xxx;

x^3+x+1 => b3b2b1b0 = 1011

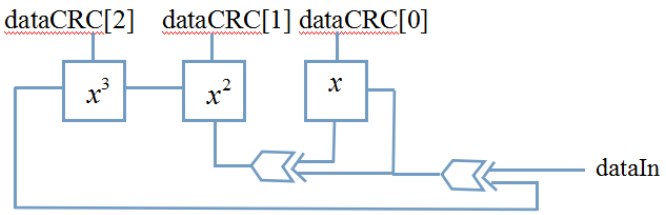
xxx(dataCRC) = 1010 000 % 1011 = 011

dataOut = 1010 011

Hardware implementation:

dataIn = {1, 0, 1, 0}, after 4 clock cycles, dataCRC = 011;

g(x) = x^3+x+1 => in shift register, x^3(b2) is output and then xor dataIn to x; x(b0) is output and then xor (dataIn xor x^3) to b1(x^2);



e.g. g(x) = x^4+x^2+x+1 => in DFF, x^4(b3) is output and then xor dataIn to x; x(b0) is output and then xor (dataIn xor x^4) to x^2; x^2(b1) is output and then xor (dataIn xor x^4) to x^3;

GENERATOR POLYNOMIALS OF SOME CRC CODES

Common Name	r	Generator	
		Polynomial	Hex
CRC-12	12	$x^{12}+x^{11}+x^3+x^2+x+1$	80F
CRC-16	16	$x^{16}+x^{15}+x^2+1$	8005
CRC-CCITT	16	$x^{16}+x^{12}+x^5+1$	1021
CRC-32	32	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+$ $x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$	04C1 1DB7

Note: CCITT(International Telegraph and Telephone Consultative Committee)

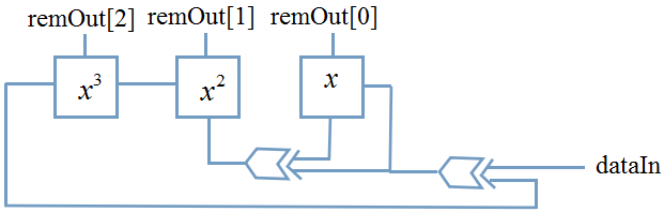
Other CRC generation polynomials, for example, are as follows.

Common Name	r	Generation Polynomial
CRC-3	3	x^3+x+1
CRC-4	4	x^4+x^2+x+1
CRC-5	5	x^5+x^2+1
CRC-6	6	x^6+x+1
CRC-8	8	$x^8+x^5+x^4+1$

```
module serialCRC3(
                        clk,
                        rst,
                        en_i,
                        dIn_i,          // 4 bits
                        dOut_o          // 3 bits CRC code
);
    input               clk;
    input               rst;
    input               en_i;
    input               dIn_i;
    output[2:0]         dOut_o;

    reg[2:0]            lfsr_r;

    assign              dOut_o = lfsr_r;

    // Generation polynomial:  x^3+x+1
    always@(posedge clk)begin
        if(rst)         lfsr_r <= 3'b000;
        else if(en_i)begin
                        lfsr_r[2] <= lfsr_r[1];
                        lfsr_r[1] <= lfsr_r[0] ^ lfsr_r[2] ^ dIn_i;
                        lfsr_r[0] <= lfsr_r[2] ^ dIn_i;
        end
    end
endmodule
```

ii. Serial CRC decoder:

dataIn={1,0,1,0,0,1,1} after 7 clock cycles, if remOut = 000, there are not any errors. If remOut is equal to zero, there is at least 1 error.



iii. Parallel CRC encoder

```
module crc5Parallel(
                        input [3:0]         dataIn,
                        output reg[4:0]     crc5,
                        input               rst,
                        input               clk
);
// LFSR for USB CRC5  g(x) = x^5 + x^2 + 1
    function [4:0] crc5Serial;
        input [4:0]     crc;
        input           data;
        begin
            crc5Serial[0] = crc[4] ^ data;
            crc5Serial[1] = crc[0];
            crc5Serial[2] = crc[1] ^ crc[4] ^ data;
            crc5Serial[3] = crc[2];
            crc5Serial[4] = crc[3];
        end
    endfunction
// 4 iterations of USB CRC5 LFSR
    function [4:0] crcIteration;
        input [4:0]     crc;
        input [3:0]     data;
        integer         i;
        begin
            crcIteration = crc;
            for(i=0; i<4; i=i+1)
                crcIteration = crc5Serial(crcIteration, data[3-i]);
            end
    endfunction
    always @(posedge clk) begin
        if(rst)     crc5 <= 5'h00;
        else        crc5 <= crcIteration(crc5,dataIn);
    end
endmodule
```

iv. Parallel CRC decoder

1. Hardware structure is the same as encoder, but number of input bits is different.