# 1 Text of Examples

/*



## From the Authors:

This file contains copies of most of the examples in "The Verilog Hardware Description Language, Fourth Edition" by D. E. Thomas and P. R. Moorby. Note that corrections may have been made to the examples in this file and thus they may

differ from the examples in the book. Later printings of the book will include these corrections.

This file, itself, cannot be simulated because several module names may be duplicated in different examples. However, the examples may be extracted and simulated. A few of the examples are not included because they were not meant to be fully simulatable.

-always

DT, PM

*/

```
module binaryToESeg;
    wire    eSeg, p1, p2, p3, p4;
    reg     A, B, C, D;

    nand #1
        g1 (p1, C, ~D),
        g2 (p2, A, B),
        g3 (p3, ~B, ~D),
        g4 (p4, A, C),
        g5 (eSeg, p1, p2, p3, p4);
endmodule


module binaryToESegSim;
    wire    eSeg, p1, p2, p3, p4;
    reg     A, B, C, D;

    nand #1
        g1 (p1, C, ~D),
        g2 (p2, A, B),
        g3 (p3, ~B, ~D),
        g4 (p4, A, C),
        g5 (eSeg, p1, p2, p3, p4);

    initial         // two slashes introduce a single line comment
        begin
            $monitor ($time,,,
                "A = %b B = %b C = %b D = %b, eSeg = %b",
                A, B, C, D, eSeg);
            //waveform for simulating the binaryToESeg driver
            #10 A = 0; B = 0; C = 0; D = 0;
            #10 D = 1;
            #10 C = 1; D = 0;
            #10 $finish;
        end
endmodule


module binaryToESeg (eSeg, A, B, C, D);
    output  eSeg;
    input   A, B, C, D;

    nand #1
        g1 (p1, C, ~D),
        g2 (p2, A, B),
```

```
        g3 (p3, ~B, ~D),
        g4 (p4, A, C),
        g5 (eSeg, p1, p2, p3, p4);
endmodule

module testBench;
    wire      w1, w2, w3, w4, w5;

    binaryToESeg      d   (w1, w2, w3, w4, w5);
    test_bToESeg      t   (w1, w2, w3, w4, w5);
endmodule

module binaryToESeg (eSeg, A, B, C, D);
    input     A, B, C, D;
    output    eSeg;

    nand #1
        g1 (p1, C, ~D),
        g2 (p2, A, B),
        g3 (p3, ~B, ~D),
        g4 (p4, A, C),
        g5 (eSeg, p1, p2, p3, p4);
endmodule

module test_bToESeg (eSeg, A, B, C, D);
    input     eSeg;
    output    A, B, C, D;
    reg       A, B, C, D;

    initial         // two slashes introduce a single line comment
      begin
            $monitor ($time,,
                "A = %b B = %b C = %b D = %b, eSeg = %b",
                A, B, C, D, eSeg);
            //waveform for simulating the nand lip lop
            #10 A = 0; B = 0; C = 0; D = 0;
            #10 D = 1;
            #10 C = 1; D = 0;
            #10 $finish;
      end
endmodule


module binaryToESeg_Behavioral (eSeg, A, B, C, D);
    output    eSeg;
```

```
    input     A, B, C, D;
    reg       eSeg;

    always @(A or B or C or D) begin
        eSeg = 1;
        if (~A & D)
            eSeg = 0;
        if (~A & B & ~C)
            eSeg = 0;
        if (~B & ~C & D)
            eSeg = 0;
    end
endmodule


module fsm (out, in, clock, reset);
    output            out;
    input             in, clock, reset;
    reg               out;
    reg       [1:0]   currentState, nextState;

    always @(in or currentState) begin   // the combinational portion
        out = ~currentState[1] & currentState[0];
        nextState = 0;
        if (currentState == 0)
            if (in) nextState = 1;
        if (currentState == 1)
            if (in) nextState = 3;
        if (currentState == 3)
            if (in) nextState = 3;
            else nextState = 1;
    end

    always @(posedge clock or negedge reset) begin  // the sequential portion
        if (~reset)
            currentState <= 0;
        else
            currentState <= nextState;
    end
endmodule


module fsmNB (out, in, clock, reset);
    output            out;
    input             in, clock, reset;
```

```
    reg             out, cS1, cS0;

    always @(cS1 or cS0)    // the combinational portion
        out = ~cS1 & cS0;

    always @(posedge clock or negedge reset) begin  // the sequential portion
        if (~reset) begin
            cS1 <= 0;
            cS0 <= 0;
        end
        else begin
            cS1 <= in & cS0;
            cS0 <= in | cS1;
        end
    end
endmodule


module m16 (value, clock);
    output  [3:0]   value;
    reg     [3:0]   value;
    input           clock;

    always @(posedge clock)
        value <= value + 1;
endmodule



module m555 (clock);
    output  clock;
    reg     clock;

    initial
        #5 clock = 1;

    always
        #50 clock = ~ clock;
endmodule



module board;
    wire    [3:0]   count;
    wire            clock, eSeg;
```

```
    m16             counter     (count, clock);
    m555            clockGen    (clock);
    binaryToESeg    disp        (eSeg, count[3], count[2], count[1], count[0]);

    initial
        $monitor ($time,,,"count=%d, eSeg=%d", count, eSeg);
endmodule


module boardWithConcatenation;
    wire            clock, eSeg, w3, w2, w1, w0;

    m16             counter     ({w3, w2, w1, w0}, clock);
    m555            clockGen    (clock);
    binaryToESeg    disp        (eSeg, w3, w2, w1, w0);

    initial
        $monitor ($time,,,"count=%d, eSeg=%d", {w3,w2,w1,w0}, eSeg);
endmodule



module counterToESeg (eSeg, clock);
    output          eSeg;
    reg     [3:0]   value;
    input           clock;

    initial
        value = 0;

    always @(posedge clock)
        value <= value + 1;

    nand #1
        g1 (p1, value[1], ~value[0]),
        g2 (p2, value[3], value[2]),
        g3 (p3, ~value[2], ~value[0]),
        g4 (p4, value[3], value[1]),
        g5 (eSeg, p1, p2, p3, p4);
endmodule


module mixedUpESegDriver (eSeg, A, B, C, D);
    output          eSeg;
```

```
    reg             eSeg;
    input           A, B, C, D;

    nand #1
        g1 (p1, C, D),
        g2 (p2, A, ~B),
        g3 (p3, ~B, ~D),
        g4 (p4, A, C);

    always @(p1 or p2 or p3 or p4)
        eSeg = ~(p1 & p2 & p3 & p4);
endmodule


module register (out, in, clear, load, clock);
    parameter               WIDTH = 7;
    output  [WIDTH:0]  out;
    reg     [WIDTH:0]  out;
    input   [WIDTH:0]  in;
    input                   clear, load, clock;

    always @(posedge clock)
        if (~clear)
            out <= 0;
        else if (~load)
            out <= in;
endmodule


module adder (sum, a, b);
    parameter   WIDTH = 7;
    input   [WIDTH:0]  a, b;
    output  [WIDTH:0]  sum;

    assign sum = a + b;
endmodule


module compareLT (out, a, b);   // compares a < b
    parameter   WIDTH = 7;
    input       [WIDTH:0]  a, b;
    output                 out;

    assign out = a < b;
endmodule
```

```
module compareLEQ (out, a, b);// compares a <= b
    parameter    WIDTH = 7;
    input        [WIDTH:0]  a, b;
    output                  out;

    assign out = a <= b;
endmodule


module sillyComputation (yIn, y, x, ck, reset);
    parameter    WIDTH = 7;
    input                   ck, reset;
    input    [WIDTH:0]  yIn;
    output   [WIDTH:0]  y, x;
    wire     [WIDTH:0]  i, addiOut, addxOut;
    wire                    yLoad, yClear, xLoad, xClear, iLoad, iClear;

    register     #(WIDTH)   I(i, addiOut, iClear, iLoad, ck),
                            Y(y, yIn, yClear, yLoad, ck),
                            X(x, addxOut, xClear, xLoad, ck);

    adder        #(WIDTH)   addI(addiOut, 1, i),
                            addX(addxOut, y, x);

    compareLT       #(WIDTH)cmpX(xLT0, x, 0);
    compareLEQ      #(WIDTH)cmpI(iLEQ10, i, 10);

    fsm             ctl
    (xLT0, iLEQ10, yLoad, yClear, xLoad, xClear, iLoad, iClear, ck, reset);
endmodule


module fsm (LT, LEQ, yLoad, yClear, xLoad, xClear, iLoad, iClear, ck, reset);
input   LT, LEQ, ck, reset;
output  yLoad, yClear, xLoad, xClear, iLoad, iClear;
reg     yLoad, yClear, xLoad, xClear, iLoad, iClear;
reg     [2:0]    cState, nState;

always @(posedge ck or negedge reset)
    if (~reset)
            cState <= 0;
    else    cState <= nState;

always @(cState or LT or LEQ)
```

```
        case (cState)
            3'b000 :  begin       // state A
                          yLoad = 1; yClear = 1; xLoad = 1; xClear = 0;
                          iLoad = 1; iClear = 0; nState = 3'b001;
                      end
            3'b001 :  begin       // state B
                          yLoad = 1; yClear = 1; xLoad = 0; xClear = 1;
                          iLoad = 0; iClear = 1; nState = 3'b010;
                      end
            3'b010 :  begin       // state C
                          yLoad = 1; yClear = 1; xLoad = 1; xClear = 1;
                          iLoad = 1; iClear = 1;
                          if (LEQ) nState = 3'b001;
                          if (~LEQ & LT) nState = 3'b011;
                          if (~LEQ & ~LT) nState = 3'b100;
                      end
            3'b011 :  begin       // state D
                          yLoad = 1; yClear = 0; xLoad = 1; xClear = 1;
                          iLoad = 1; iClear = 1; nState = 3'b101;
                      end
            3'b100 :  begin       // state E
                          yLoad = 1; yClear = 1; xLoad = 1; xClear = 0;
                          iLoad = 1; iClear = 1; nState = 3'b101;
                      end
            default : begin  // required to satisfy combinational synthesis rules
                          yLoad = 1; yClear = 1; xLoad = 1; xClear = 1;
                          iLoad = 1; iClear = 1; nState = 3'b000;
                          $display ("Oops, unknown state: %b", cState);
                      end
        endcase
    endmodule


    module simpleTutorial (clock, y, x);
        input             clock;
        output   [7:0]    x, y;
        reg      [7:0]    x, y, i;

        always begin
            @(posedge clock) x <= 0;
            i = 0;
            while (i <= 10) begin
                  @(posedge clock);
                  x <= x + y;
                  i = i + 1;
```

```
        end
        @(posedge clock);
        if (x < 0)
            y <= 0;
        else  x <= 0;
    end
endmodule


`define DvLen 15
`define DdLen 31
`define QLen 15
`define HiDdMin 16

module divide (ddInput, dvInput, quotient, go, done);
    input   [`DdLen:0]   ddInput, dvInput;
    output  [`QLen:0]    quotient;
    input                go;
    output               done;

    reg     [`DdLen:0]   dividend;
    reg     [`QLen:0]    quotient;
    reg     [`DvLen:0]   divisor;
    reg                  done, negDivisor, negDividend;

    always begin
        done = 0;
        wait (go);
        divisor = dvInput;
        dividend = ddInput;
        quotient = 0;
        if (divisor) begin
            negDivisor = divisor[`DvLen];
            if (negDivisor)      divisor = - divisor;
            negDividend = dividend[`DdLen];
            if (negDividend)    dividend = - dividend;
                repeat (`DvLen + 1)  begin
                    quotient = quotient << 1;
                    dividend = dividend << 1;
                    dividend[`DdLen:`HiDdMin] =
                        dividend[`DdLen:`HiDdMin] - divisor;
                    if (! dividend [`DdLen])   quotient = quotient + 1;
                    else
                        dividend[`DdLen:`HiDdMin] =
                            dividend[`DdLen:`HiDdMin] + divisor;
```

```
                end
                if (negDivisor != negDividend)   quotient = - quotient;
        end
        done = 1;
        wait (~go);
    end
endmodule
```

```
module mark1;
reg [31:0]    m [0:8191];   // 8192 x 32 bit memory
reg [12:0]    pc;           // 13 bit program counter
reg [31:0]    acc;          // 32 bit accumulator
reg [15:0]    ir;           // 16 bit instruction register
reg           ck;           // a clock signal

    always
        begin
            @(posedge ck)
                ir = m [pc];     // fetch an instruction
            @(posedge ck)
                if (ir[15:13] == 3'b000)          // begin decoding
                    pc = m [ir [12:0]];           // and executing
                else if (ir[15:13] == 3'b001)
                    pc = pc + m [ir [12:0]];
                else if (ir[15:13] == 3'b010)
                    acc = -m [ir [12:0]];
                else if (ir[15:13] == 3'b011)
                    m [ir [12:0]] = acc;
                else if ((ir[15:13] == 3'b101) || (ir[15:13] == 3'b100))
                    acc = acc - m [ir [12:0]];
                else if (ir[15:13] == 3'b110)
                    if (acc < 0) pc = pc + 1;
            pc = pc + 1;          //increment program counter
        end
endmodule
```

```
module mark1Case;
reg [31:0]    m [0:8191];   // 8192 x 32 bit memory
reg [12:0]    pc;           // 13 bit program counter
```

```
reg [31:0]    acc;          // 32 bit accumulator
reg [15:0]    ir;           // 16 bit instruction register
reg           ck;           // a clock signal

    always
        begin
            @(posedge ck)
                ir = m [pc];
            @(posedge ck)
                case (ir [15:13])
                    3'b000 :    pc = m [ir [12:0]];
                    3'b001 :    pc = pc + m [ir [12:0]];
                    3'b010 :    acc = -m [ir [12:0]];
                    3'b011 :    m [ir [12:0]] = acc;
                    3'b100,
                    3'b101 :    acc = acc - m [ir [12:0]];
                    3'b110 :    if (acc < 0) pc = pc + 1;
                endcase
            pc = pc + 1;
        end
endmodule



module mark1Mult;
reg [31:0]    m [0:8191];   // 8192 x 32 bit memory
reg [12:0]    pc;           // 13 bit program counter
reg [31:0]    acc;          // 32 bit accumulator
reg [15:0]    ir;           // 16 bit instruction register
reg           ck;           // a clock signal

    always
        begin
            @(posedge ck)
                ir = m [pc];
            @(posedge ck)
                case (ir [15:13])
                    3'b000 :    pc = m [ir [12:0]];
                    3'b001 :    pc = pc + m [ir [12:0]];
                    3'b010 :    acc = -m [ir [12:0]];
                    3'b011 :    m [ir [12:0]] = acc;
                    3'b100,
                    3'b101 :    acc = acc - m [ir [12:0]];
                    3'b110 :    if (acc < 0) pc = pc + 1;
                    3'b111 :    acc = acc * m [ir [12:0]];    //multiply
```

```
                endcase
            pc = pc + 1;
        end
endmodule


module mark1Task;
    reg [15:0]    m [0:8191];    // 8192 x 16 bit memory
    reg [12:0]    pc;            // 13 bit program counter
    reg [12:0]    acc;           // 13 bit accumulator
    reg           ck;            // a clock signal

    always
        begin: executeInstructions
            reg  [15:0]     ir;   // 16 bit instruction register

            @(posedge ck)
                ir = m [pc];
            @(posedge ck)
                case (ir [15:13])
                    // other case expressions as before
                    3'b111 :    multiply (acc, m [ir [12:0]]);
                endcase
            pc = pc + 1;
        end

    task multiply;
    inout    [12:0]  a;
    input    [15:0]  b;

        begin: serialMult
            reg      [5:0]   mcnd, mpy;//multiplicand and multiplier
            reg      [12:0] prod;//product

            mpy = b[5:0];
            mcnd = a[5:0];
            prod = 0;
            repeat (6)
                begin
                    if (mpy[0])
                        prod = prod + {mcnd, 6'b000000};
                    prod = prod >> 1;
                    mpy = mpy >> 1;
                end
            a = prod;
```

```
        end
    endtask
endmodule



module mark1Fun;
    reg [15:0]    m [0:8191];    // 8192 x 16 bit memory
    reg [12:0]    pc;            // 13 bit program counter
    reg [12:0]    acc;           // 13 bit accumulator
    reg           ck;            // a clock signal

    always
        begin: executeInstructions
            reg  [15:0]     ir;   // 16 bit instruction register

            @(posedge ck)
                ir = m [pc];
            @(posedge ck)
                case (ir [15:13])
                    //case expressions, as before
                    3'b111:     acc = multiply(acc, m [ir [12:0]]);
                endcase
            pc = pc + 1;
        end

    function [12:0] multiply;
    input    [12:0]  a;
    input    [15:0]  b;

        begin: serialMult
            reg      [5:0]        mcnd, mpy;

            mpy = b[5:0];
            mcnd = a[5:0];
            multiply = 0;
            repeat (6)
                begin
                    if (mpy[0])
                        multiply = multiply + {mcnd, 6'b000000};
                    multiply = multiply >> 1;
                    mpy = mpy >> 1;
                end
        end
    endfunction
```

```
endmodule

module mark1Mod;
    reg [15:0]    m [0:8191];    // 8192 x 16 bit memory
    reg [12:0]    pc;            // 13 bit program counter
    reg [12:0]    acc;           // 13 bit accumulator
    reg [15:0]    ir;            // 16 bit instruction register
    reg           ck;            // a clock signal

    reg       [31:0]  mcnd;
    reg               go;
    wire      [31:0]  prod;
    wire              done;
    multiply          mul (prod, acc, mcnd, go, done);

    always
        begin
            @(posedge ck)
                go = 0;
                ir = m [pc];
            @(posedge ck)
                case (ir [15:13])
                    //other case expressions
                    3'b111:begin
                               wait (~done) mcnd = m [ir [12:0]];
                               go = 1;
                               wait (done);
                               acc = prod;
                           end
                endcase
            pc = pc + 1;
        end
endmodule

module multiply (prod, mpy, mcnd, go, done);
    output    [12:0]  prod;
    input     [5:0]   mpy, mcnd;
    input             go;
    output            done;

    reg       [12:0]  prod;
    reg       [5:0]   myMpy;
    reg               done;

    always
```

```
        begin
            done = 0;
            wait (go);
            myMpy = mpy;
            prod = 0;
            repeat (6)
                begin
                    if (myMpy[0])
                        prod = prod + {mcnd, 6'b000000};
                    prod = prod >> 1;
                    myMpy = myMpy >> 1;
                end
            done = 1;
            wait (~go);
        end
endmodule



module topFib;
    wire [15:0]   number, numberOut;

    numberGen     ng     (number);
    fibNumCalc    fnc    (number, numberOut);
endmodule

module numberGen(number);
output [15:0]     number;
reg    [15:0]     number;
event             ready;          //declare the event

    initial
        number = 0;

    always
        begin
            #50 number = number + 1;
            #50 -> ready;         //generate event signal
        end
endmodule

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

```
reg     [15:0]    count, fibNum, oldNum, temp;

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



module ProducerConsumer;
    reg             consReady, prodReady;
    reg     [7:0]   dataInCopy, dataOut;

    always      // The consumer process
        begin
            consReady = 1;      // indicate consumer ready
            forever
                begin
                    wait (prodReady)
                        dataInCopy = dataOut;
                    consReady = 0;// indicate value consumed
                    //...munch on data
                    wait (!prodReady) // complete handshake
                        consReady = 1;
                end
        end

    always      // The producer process
        begin
            prodReady = 0;      // indicate nothing to transfer
            forever
                begin
                    // …produce data and put into “dataOut”
                    wait (consReady) // wait for consumer ready
                        dataOut = $random;
```

```
                    prodReady = 1; //indicate ready to transfer
                    wait (!consReady) //inish handshake
                        prodReady = 0;
                    end
                end
endmodule


module endlessLoop (inputA);
    input       inputA;
    reg[15:0]   count;

    always
        begin
            count = 0;
            while (inputA)
                count = count + 1; // wait for inputA to change to false
            $display ("This will never print if inputA is true!");
        end
endmodule


`define READ 0
`define WRITE 1

module sbus;
parameter   tClock =    20;

reg             clock;
reg  [15:0]     m[0:31]; //32 16-bit words
reg  [15:0]     data;
// registers names xLine model the bus lines using global registers
reg             rwLine;     //write = 1, read = 0
reg   [4:0]     addressLines;
reg   [15:0]    dataLines;


initial
    begin
        $readmemh ("memory.data", m);
        clock = 0;
        $monitor ("rw=%d, data=%d, addr=%d at time %d",
rwLine, dataLines, addressLines, $time);
    end
```

```
always
    #tClock clock = !clock;

initial    // bus master end
    begin
        #1
        wiggleBusLines ( `READ, 2, data);
        wiggleBusLines ( `READ, 3, data);
        data = 5;
        wiggleBusLines ( `WRITE, 2, data);
        data = 7;
        wiggleBusLines ( `WRITE, 3, data);
        wiggleBusLines ( `READ, 2, data);
        wiggleBusLines ( `READ, 3, data);
        $finish;
    end

    task wiggleBusLines;
    input          readWrite;
    input [5:0]    addr;
    inout [15:0]   data;

    begin
        rwLine <= readWrite;
        if (readWrite) begin       // write value
            addressLines <= addr;
            dataLines <= data;
        end
        else begin                 //read value
            addressLines <= addr;
            @ (negedge clock);
        end
        @(negedge clock);
        if (~readWrite)
            data <= dataLines;  // value returned during read cycle
    end
endtask

always   // bus slave end
    begin
        @(negedge clock);
        if (~rwLine) begin         //read
            dataLines <= m[addressLines];
            @(negedge clock);
        end
```

```
        else        //write
            m[addressLines] <= dataLines;
    end
endmodule


module mark1PipeStage;
    reg [15:0]    m [0:8191];    // 8192 x 16 bit memory
    reg [12:0]    pc, pctemp;    // 13 bit program counter and temporary
    reg [12:0]    acc;           // 13 bit accumulator
    reg [15:0]    ir;            // 16 bit instruction register
    reg           ck, skip;

    always @(posedge ck) begin      //fetch process
            if (skip)
                pc = pctemp;
            ir <= m [pc];
            pc <= pc + 1;
    end


    always @(posedge ck) begin      //execute process
        if (skip)
            skip <= 0;
        else
            case (ir [15:13])
                3'b000 :   begin
                               pctemp <= m [ir [12:0]];
                               skip <= 1;
                           end
                3'b001 :   begin
                               pctemp <= pc + m [ir [12:0]];
                               skip <= 1;
                           end
                3'b010 :   acc <= -m [ir [12:0]];
                3'b011 :   m [ir [12:0]] <= acc;
                3'b100,
                3'b101 :   acc <= acc - m [ir [12:0]];
                3'b110 :   if (acc < 0) begin
                               pctemp <= pc + 1;
                               skip <= 1;
                           end
            endcase
    end
endmodule
```

```
module simpleTutorialWithReset (clock, reset, y, x);
    input           clock, reset;
    output  [7:0]   x, y;
    reg     [7:0]   x, y, i;

    initial
        forever begin
            @(negedge reset)
                disable main;
        end

    always begin: main
        wait (reset);
        @(posedge clock) x <= 0;
        i = 0;
        while (i <= 10) begin
            @(posedge clock);
            x <= x + y;
            i = i + 1;
        end
        @(posedge clock);
        if (x < 0)
            y <= 0;
        else  x <= 0;
    end
endmodule



module dFlop (preset, clear, q, clock, d);
    input   preset, clear, clock, d;
    output  q;
    reg     q;

    always
        @(clear or preset)
            begin
                if (!clear)
                    #10   assign q = 0;
                else if (!preset)
                    #10   assign q = 1;
                else
                    #10   deassign q;
            end
```

```
    always
        @(negedge clock)
            q = #10 d;
endmodule


module simpleTutorialWithReset (clock, reset, y, x);
    input           clock, reset;
    output  [7:0]   x, y;
    reg     [7:0]   x, y, i;

    always fork: main
        @(negedge reset)
            disable main;
        begin
            wait (reset);
            @(posedge clock) x <= 0;
            i = 0;
            while (i <= 10) begin
                @(posedge clock);
                x <= x + y;
                i = i + 1;
            end
            @(posedge clock);
            if (x < 0)
                y <= 0;
            else x <= 0;
        end
    join
endmodule



module fullAdder(cOut, sum, aIn, bIn, cIn);
    output  cOut, sum;
    input   aIn, bIn, cIn;

    wire    x2;

    nand    (x2, aIn, bIn),
            (cOut, x2, x8);
    xnor    (x9, x5, x6);
    nor     (x5, x1, x3),
            (x1, aIn, bIn);
```

```
    or      (x8, x1, x7);
    not     (sum, x9),
            (x3, x2),
            (x6, x4),
            (x4, cIn),
            (x7, x6);
endmodule



module andOfComplements (a, b, c, d);
    input   a, b;
    output  c, d;

    wand    c;
    wire    d;

    not (c, a);
    not (c, b);

    not (d, a);
    not (d, b);
endmodule



module testHam();
    reg     [1:8]    original;
    wire    [1:8]   regenerated;
    wire    [1:12]  encoded,
                    messedUp;
    integer         seed;

    initial begin
        seed = 1;
        forever begin
            original = $random (seed);
            #1
            $display ("original=%h, encoded=%h, messed=%h,regen=%h",
                    original, encoded, messedUp, regenerated);
        end
    end

    hamEncode   hIn (original, encoded);
    hamDecode   hOut (messedUp, regenerated);
```

```
    assign messedUp = encoded ^ 12'b 0000_0010_0000;
endmodule


module hamEncode (vIn, valueOut);
    input   [1:8]   vIn;
    output  [1:12]  valueOut;

    wire    h1, h2, h4, h8;

    xor     (h1, vIn[1], vIn[2], vIn[4], vIn[5], vIn[7]),
            (h2, vIn[1], vIn[3], vIn[4], vIn[6], vIn[7]),
            (h4, vIn[2], vIn[3], vIn[4], vIn[8]),
            (h8, vIn[5], vIn[6], vIn[7], vIn[8]);

    assign valueOut = {h1, h2, vIn[1], h4, vIn[2:4], h8, vIn[5:8]};

endmodule



module hamDecode (vIn, valueOut);
    input   [1:12]  vIn;
    output  [1:8]   valueOut;
    wire            c1, c2, c4, c8;
    wire    [1:8]    bitFlippers;
    xor     (c1, vIn[1], vIn[3], vIn[5], vIn[7], vIn[9], vIn[11]),
            (c2, vIn[2], vIn[3], vIn[6], vIn[7], vIn[10], vIn[11]),
            (c4, vIn[4], vIn[5], vIn[6], vIn[7], vIn[12]),
            (c8, vIn[8], vIn[9], vIn[10], vIn[11], vIn[12]);

    deMux mux1 (bitFlippers, c1, c2, c4, c8, 1'b1);
    xor8 x1 (valueOut, bitFlippers, {vIn[3], vIn[5], vIn[6], vIn[7], vIn[9],
                                     vIn[10], vIn[11], vIn[12]});
endmodule

module deMux (outVector, A, B, C, D, enable);
    output   [1:8]    outVector;
    input             A, B, C, D, enable;
    and     v (m12, D, C, ~B, ~A, enable),
            h (m11, D, ~C, B, A, enable),
            d (m10, D, ~C, B, ~A, enable),
            l (m9, D, ~C, ~B, A, enable),
            s (m7, ~D, C, B, A, enable),
```

```
            u (m6, ~D, C, B, ~A, enable),
            c (m5, ~D, C, ~B, A, enable),
            ks (m3, ~D, ~C, B, A, enable);

    assign outVector = {m3, m5, m6, m7, m9, m10, m11, m12};
endmodule

module xor8 (xout, xin1, xin2);
    output  [1:8]   xout;
    input   [1:8]   xin1, xin2;
    xor     (xout[8], xin1[8], xin2[8]),
            (xout[7], xin1[7], xin2[7]),
            (xout[6], xin1[6], xin2[6]),
            (xout[5], xin1[5], xin2[5]),
            (xout[4], xin1[4], xin2[4]),
            (xout[3], xin1[3], xin2[3]),
            (xout[2], xin1[2], xin2[2]),
            (xout[1], xin1[1], xin2[1]);
endmodule



module xor8 (xout, xin1, xin2);
output  [1:8]       xout;
input   [1:8]       xin1, xin2;

    xor a[1:8] (xout, xin1, xin2);
endmodule



module xor8 (xout, xin1, xin2);
output  [1:8]       xout;
input   [1:8]       xin1, xin2;
    xor     (xout[1], xin1[1], xin2[1]),
            (xout[2], xin1[2], xin2[2]),
            (xout[3], xin1[3], xin2[3]),
            (xout[4], xin1[4], xin2[4]),
            (xout[5], xin1[5], xin2[5]),
            (xout[6], xin1[6], xin2[6]),
            (xout[7], xin1[7], xin2[7]),
            (xout[8], xin1[8], xin2[8]);
endmodule
```

```
module reggae (Q, D, clock, clear);
    output  [7:0]   Q;
    input   [7:0]   D;
    input           clock, clear;

    dff     r[7:0]  (Q, D, clear, clock);
endmodule


module regExpanded (Q, D, clock, clear);
    output  [7:0]   Q;
    input   [7:0]   D;
    input           clock, clear;

    dff     r7  (Q[7], D[7], clear, clock),
                r6  (Q[6], D[6], clear, clock),
                r5  (Q[5], D[5], clear, clock),
                r4  (Q[4], D[4], clear, clock),
                r3  (Q[3], D[3], clear, clock),
                r2  (Q[2], D[2], clear, clock),
                r1  (Q[1], D[1], clear, clock),
                r0  (Q[0], D[0], clear, clock);
endmodule


module regFromTwoBusses (Q, busHigh, busLow, clock, clear);
    output  [7:0]   Q;
    input   [3:0]   busHigh, busLow;
    input           clock, clear;

    dff     r[7:0]  (Q, {busHigh, busLow}, clear, clock);
endmodule


module shiftRegister (in, out, clock, clear);
input   [7:0]   in;
output  [7:0]   out;
input           clock, clear;
wire    [15:0]  w;

    reggae  stage[2:0]  ({out, w}, {w, in}, clock, clear);
endmodule
```

```
module oneBitFullAdder(cOut, sum, aIn, bIn, cIn);
    output   cOut, sum;
    input    aIn, bIn, cIn;

    assign       sum = aIn ^ bIn ^ cIn,
                 cOut = (aIn & bIn) | (bIn & cIn) | (aIn & cIn);

endmodule



module multiplexor(a, b, c, d, select, e);
    input        a, b, c, d;
    input    [1:0]select;
    output       e;

    assign   e = mux (a, b, c, d, select);

function mux;
    input        a, b, c, d;
    input    [1:0]select;

    case (select)
        2'b00:   mux = a;
        2'b01:   mux = b;
        2'b10:   mux = c;
        2'b11:   mux = d;
        default: mux = 'bx;
    endcase
endfunction
endmodule



module modXor (AXorB, a, b);
    output   [7:0]   AXorB;
    input    [7:0]   a, b;

    wire     [7:0]   #5 AXorB = a ^ b;
endmodule
```

```
module wandOfAssigns (a, b, c);
input    a, b;
output  c;

      wand     #10      c;

      assign    #5   c = ~a;
      assign    #3   c = ~b;
endmodule


module bufferDriver (busLine, bufferedVal, bufInput, busEnable);
      inout     busLine;
      input     bufInput, busEnable;
      output    bufferedVal;

      assign    bufferedVal = busLine,
                busLine = (busEnable) ? bufInput : 1'bz;
endmodule



module Memory_64Kx8 (dataBus, addrBus, we, re, clock);
      input     [15:0]   addrBus;
      inout     [7:0]    dataBus;
      input              we, re, clock;
      reg       [7:0]    out;
      reg       [7:0]    Mem [65535:0];

      assign dataBus = (~re)? out: 16'bz;        /* drive the tristate output */

      always @(negedge re or addrBus)
          out = Mem[addrBus];

      always @(posedge clock)
          if (we == 0)
                Mem[addrBus] <= dataBus;
endmodule



module xor8 (xout, xin1, xin2);
output  [1:8]       xout;
input    [1:8]      xin1, xin2;
```

```
    xor     (xout[8], xin1[8], xin2[8]),
            (xout[7], xin1[7], xin2[7]),
            (xout[6], xin1[6], xin2[6]),
            (xout[5], xin1[5], xin2[5]),
            (xout[4], xin1[4], xin2[4]),
            (xout[3], xin1[3], xin2[3]),
            (xout[2], xin1[2], xin2[2]),
            (xout[1], xin1[1], xin2[1]);
endmodule


module xorx (xout, xin1, xin2);
parameter   width = 4,
            delay = 10;
output  [1:width] xout;
input   [1:width] xin1, xin2;

    assign #(delay) xout = xin1 ^ xin2;
endmodule


module xorsAreUs (a1, a2);
    output  [3:0]   a1, a2;
    reg [3:0]       b1, c1, b2, c2;

    xorx    a(a1, b1, c1),
            b(a2, b2, c2);
endmodule

module xorx (xout, xin1, xin2);
    parameter   width = 4,
                delay = 10;
    output  [1:width]   xout;
    input   [1:width]   xin1, xin2;

    assign #delay xout = xin1 ^ xin2;
endmodule

module annotate;
    defparam
        xorsAreUs.b.delay = 5;
endmodule
```

```
`define READ 0
`define WRITE 1

module sbus;
    parameter
        Tclock = 20,
        Asize = 4,
        Dsize = 15,
        Msize = 31;

    reg clock;

    wire                rw;
    wire    [Asize:0]   addr;
    wire    [Dsize:0]   data;

    master  #(Asize, Dsize)      m1  (rw, addr, data, clock);
    slave   #(Asize, Dsize, Msize)s1    (rw, addr, data, clock);

    initial
        begin
            clock = 0;
            $monitor ("rw=%d, data=%d, addr=%d at time %d",
                                rw, data, addr, $time);
        end

    always
        #Tclock clock = !clock;
endmodule

module busDriver(busLine, valueToGo, driveEnable);
    parameter   Bsize = 15;
    inout   [Bsize:0]   busLine;
    input   [Bsize:0]   valueToGo;
    input               driveEnable;

    assign busLine = (driveEnable) ? valueToGo: 'bz;
endmodule
module slave (rw, addressLines, dataLines, clock);
    parameter
        Asize = 4,
        Dsize = 15,
        Msize = 31;
    input   rw, clock;
    input   [Asize:0]   addressLines;
```

```
    inout    [Dsize:0]  dataLines;

    reg      [Dsize:0]  m[0:Msize];
    reg      [Dsize:0]  internalData;
    reg      enable;

    busDriver    #(Dsize)   bSlave (dataLines, internalData, enable);

    initial
      begin
        $readmemh ("memory.data", m);
        enable = 0;
      end

    always    // bus slave end
      begin
        @(negedge clock);
        if (~rw) begin  //read
          internalData <= m[addressLines];
          enable <= 1;
          @(negedge clock);
          enable <= 0;
        end
        else      //write
          m[addressLines] <= dataLines;
      end
endmodule

module master (rw, addressLines, dataLines, clock);
    parameter
      Asize = 4,
      Dsize = 15;
    input                clock;
    output               rw;
    output   [Asize:0]   addressLines;
    inout    [Dsize:0]   dataLines;


    reg                  rw, enable;
    reg      [Dsize:0]   internalData;
    reg      [Asize:0]   addressLines;

    busDriver #(Dsize) bMaster (dataLines, internalData, enable);

    initial    enable = 0;
```

```
    always    // bus master end
        begin
            #1
            wiggleBusLines (`READ, 2, 0);
            wiggleBusLines (`READ, 3, 0);
            wiggleBusLines (`WRITE, 2, 5);
            wiggleBusLines (`WRITE, 3, 7);
            wiggleBusLines (`READ, 2, 0);
            wiggleBusLines (`READ, 3, 0);
            $finish;
    end

task wiggleBusLines;
    input               readWrite;
    input   [Asize:0]   addr;
    input   [Dsize:0]   data;

    begin
        rw <= readWrite;
        if (readWrite) begin// write value
            addressLines <= addr;
            internalData <= data;
            enable <= 1;
        end
        else begin      //read value
            addressLines <= addr;
            @ (negedge clock);
        end
        @(negedge clock);
        enable <= 0;
    end
endtask
endmodule


module triStateLatch (qOut, nQOut, clock, data, enable);
    output  qOut, nQOut;
    input   clock, data, enable;
    tri     qOut, nQOut;

    not     #5          (ndata, data);
    nand    #(3,5)      d(wa, data, clock),
                        nd(wb, ndata, clock);
    nand    #(12, 15)   qQ(q, nq, wa),
```

```
                                nQ(nq, q, wb);
    bufif1   #(3, 7, 13)        qDrive (qOut, q, enable),
                                nQDrive(nQOut, nq, enable);
endmodule



module IOBuffer (bus, in, out, dir);
    inout    bus;
    input    in, dir;
    output   out;

    parameter
        R_Min = 3, R_Typ = 4, R_Max = 5,
        F_Min = 3, F_Typ = 5, F_Max = 7,
        Z_Min = 12, Z_Typ = 15, Z_Max = 17;

    bufif1   #(R_Min: R_Typ: R_Max,
                    F_Min: F_Typ: F_Max,
                    Z_Min: Z_Typ: Z_Max)
             (bus, out, dir);

    buf      #(R_Min: R_Typ: R_Max,
                    F_Min: F_Typ: F_Max)
             (in, bus);
endmodule


module nandLatch (q, qBar, set, reset);
    output   q, qBar;
    input    set, reset;

    nand #2
        (q, qBar, set),
        (qBar, q, reset);
endmodule


module DFF(q, d, clock);
    output   q;
    input    d, clock;
    reg          q;

    always
        @ (posedge clock)
```

```
            #5 q = d;
endmodule


module behavioralNand (out, in1, in2, in3);
    output      out;
    input       in1, in2, in3;
    reg             out;
    parameter   delay = 5;

    always
        @ (in1 or in2 or in3)
            #delay out = ~(in1 & in2 & in3);
endmodule



module twoPhiLatch (phi1, phi2, q, d);
    input   phi1, phi2, d;
    output  q;
    reg     q, qInternal;

    always begin
        @ (posedge phi1)
            qInternal = d;
        @ (posedge phi2)
            q = qInternal;
    end
endmodule



module twoPhiLatchWithDelay (phi1, phi2, q, d);
    input   phi1, phi2, d;
    output  q;
    reg     q, qInternal;

    always begin
        @ (posedge phi1)
            #2 qInternal = d;
        @ (posedge phi2)
            #2 q = qInternal;
    end
endmodule
```

```
module stupidVerilogTricks (f, a, b);
    input     a, b;
    output    f;
    reg       f, q;

    initial
        f = 0;

    always
        @ (posedge a)
            #10 q = b;

    not       (qBar, q);

    always
        @ q
            f = qBar;

endmodule


module goesBothWays (Q, clock);
    input               clock;
    output    [2:1]     Q;

    wire      q1, q2;

    assign Q = {q2, q1};

    dff       a (q1, ~q1, clock),
              b (q2, q1, clock);
endmodule

module dff (q, d, clock);
    input     d, clock;
    output    q;
    reg             q;

    always
        @(posedge clock)
            #3 q = d;
endmodule
```

```
module suspend;
    reg     a;
    wire    b = a;

    initial begin
        a = 1;
        $display ("a = %b, b = %b", a, b);
    end
endmodule



module fsm (cS1, cS0, in, clock);
    output          cS1, cS0;
    input           in, clock;
    reg             cS1, cS0;

    always @(posedge clock) begin
        cS1 <= in & cS0;
        cS0 <= in | cS1;
    end
endmodule



module inertialNand (doneIt, lisa, michael);
    output      doneIt;
    input       lisa,
                michael;
    reg         doneIt;
    parameter   pDelay = 5;

    always
        @(lisa or michael)
            doneIt <= #pDelay ~(lisa & michael);
endmodule



module pipeMult (product, mPlier, mCand, go, clock);
    input           go, clock;
    input   [9:0]   mPlier, mCand;
    output  [19:0]  product;
    reg     [19:0]  product;
```

```
    always
        @(posedge go)
            product <= repeat (4) @(posedge clock) mPlier * mCand;
endmodule



module sMux (f, a, b, select);
    input   a, b, select;
    output  f;

    nand    #8
        (f, aSelect, bSelect),
        (aSelect, select, a),
        (bSelect, notSelect, b);
    not
        (notSelect, select);
endmodule


module bMux (f, a, b, select);
    input   a, b, select;
    output  f;
    reg     f;

    always
        @select
            #8 f = (select) ? a : b;
endmodule


module beenThere;
    reg     [15:0]  q;
    wire            h;
    wire    [15:0]  addit;

    doneThat   dT (q, h, addit);
    initial q = 20;

    always begin
        @ (posedge h);
        if (addit == 1)
            q = q + 5;
        else  q = q - 3;
    end
```

```
endmodule


module doneThat(que, f, add);
    input    [15:0]  que;
    output           f;
    reg              f;
    output   [15:0]  add;
    reg      [15:0]  add;
    always
        #10 f = ~ f;

    initial begin
        f = 0;
        add = 0;
        #14 add = que + 1;
        #14 add = 0;
    end
endmodule


module nbSchedule (q2);
    wire     q1;
    output   q2;
    reg      c, a;

    xor      (d, a, q1),
             (clk, 1'b1, c);
             // holy doodoo, Batman, a gated clock!
    dff      s1 (q1, d, clk),
             s2 (q2, q1, clk);
    initial begin
        c = 1;
        a = 0;
        #8 a = 1;
    end

    always
        #20 c = ~c;
endmodule

module dff (q, d, c);
    input    d, c;
    output   q;
    reg      q;
```

```
    initial q = 0;

    always
        @(posedge c) q <= d;
endmodule


module synGate (f, a, b, c);
    output  f;
    input   a, b, c;

    and     A (a1, a, b, c);
    and     B (a2, a, ~b, ~c);
    and     C (a3, ~a, o1);
    or      D (o1, b, c);
    or      E (f, a1, a2, a3);
endmodule


module synAssign (f, a, b, c);
    output  f;
    input   a, b, c;

    assign  f = (a & b & c) | (a & ~b & ~c) | (~a & (b | c));
endmodule


module addWithAssign (carry, sum, A, B, Cin);
    parameter WIDTH = 4;
    output  [WIDTH:0]  sum;
    input   [WIDTH:0]  A, B;
    input              Cin;
    output             carry;

    assign  {carry, sum} = A + B + Cin;
endmodule

module muxWithAssign (out, A, B, sel);
    output      out;
    input       A, B, sel;

    assign  out = (sel) ? A: B;
endmodule
```

```
module synCombinationalAlways (f, a, b, c);
    output   f;
    input    a, b, c;
    reg      f;

    always @ (a or b or c)
        if (a == 1)
            f = b;
        else
            f = c;
endmodule



module synInferredLatch (f, a, b, c);
    output   f;
    input    a, b, c;
    reg      f;

    always @(a or b or c)
        if (a == 1)
            f = b & c;
endmodule



module synCase (f, a, b, c);
    output   f;
    input    a, b, c;
    reg      f;

    always @(a or b or c)
        case ({a, b, c})
            3'b000:    f = 1'b0;
            3'b001:    f = 1'b1;
            3'b010:    f = 1'b1;
            3'b011:    f = 1'b1;
            3'b100:    f = 1'b1;
            3'b101:    f = 1'b0;
            3'b110:    f = 1'b0;
            3'b111:    f = 1'b1;
        endcase
endmodule
```

```
module synCaseWithDefault (f, a, b, c);
    output  f;
    input   a, b, c;
    reg     f;

    always @(a or b or c)
        case ({a, b, c})
            3'b000:     f = 1'b0;
            3'b101:     f = 1'b0;
            3'b110:     f = 1'b0;
            default:    f = 1'b1;
        endcase
endmodule



module synCaseWithDC (f, a, b, c);
    output  f;
    input   a, b, c;
    reg     f;

    always @(a or b or c)
        case ({a, b, c})
            3'b001:     f = 1'b1;
            3'b010:     f = 1'b1;
            3'b011:     f = 1'b1;
            3'b100:     f = 1'b1;
            3'b110:     f = 1'b0;
            3'b111:     f = 1'b1;
            default:    f = 1'bx;
        endcase
endmodule


module synUsingDC (f, a, b);
    output  f;
    input   a, b;
    reg     f;

    always @(a or b)
        casex ({a, b})
            2'b0?:  f = 1;
            2'b10:  f = 0;
            2'b11:  f = 1;
```

```
        endcase
endmodule


module synXor8 (xout, xin1, xin2);
    output  [1:8]   xout;
    input   [1:8]   xin1, xin2;
    reg     [1:8]   xout, i;

    always @(xin1 or xin2)
        for (i = 1; i <= 8; i = i + 1)
            xout[i] = xin1[i] ^ xin2[i];
endmodule



module synLatchReset (Q, g, d, reset);
    output  Q;
    input   g, d, reset;
    reg     Q;

    always @(g or d or reset)
        if (~reset)
            Q = 0;
        else if (g)
            Q = d;
endmodule



module synDFF (q, d, clock);
    output  q;
    input   clock, d;
    reg     q;

    always @(negedge clock)
        q <= d;
endmodule



module synDFFwithSetReset (q, d, reset, set, clock);
    input   d, reset, set, clock;
    output  q;
    reg     q;
```

```
    always @(posedge clock or negedge reset or posedge set) begin
        if (~reset)
            q <= 0;
        else if (set)
            q <= 1;
        else  q <= d;
    end
endmodule



module synTriState (bus, in, driveEnable);
    input    in, driveEnable;
    output   bus;
    reg      bus;

    always @(in or driveEnable)
        if (driveEnable)
            bus = in;
        else  bus = 1'bz;
endmodule



module fsm (i, clock, reset, out);
    input            i, clock, reset;
    output  [2:0]    out;
    reg     [2:0]    out;

    reg     [2:0]    currentState, nextState;

    parameter [2:0]  A = 0,      // The state labels and their assignments
                     B = 1,
                     C = 2,
                     D = 3,
                     E = 4,
                     F = 5;

    always @(i or currentState)  // The combinational logic
        case (currentState)
            A:  begin
                    nextState = (i == 0) ? A : B;
                    out = (i == 0) ? 3'b000 : 3'b100;
                end
```

```
                B:  begin
                        nextState = (i == 0) ? A : C;
                        out = (i == 0) ? 3'b000 : 3'b100;
                    end
                C:  begin
                        nextState = (i == 0) ? A : D;
                        out = (i == 0) ? 3'b000 : 3'b101;
                    end
                D:  begin
                        nextState = (i == 0) ? D : E;
                        out = (i == 0) ? 3'b010 : 3'b110;
                    end
                E:  begin
                        nextState = (i == 0) ? D : F;
                        out = (i == 0) ? 3'b010 : 3'b110;
                    end
                F:  begin
                        nextState = D;
                        out = (i == 0) ? 3'b000 : 3'b101;
                    end
                default: begin  // oops, undeined states. Go to state A
                        nextState = A;
                        out = (i == 0) ? 3'bxxx : 3'bxxx;
                    end
        endcase
            always @(posedge clock or negedge reset) // The state register
            if (~reset)
                    currentState <= A;
            else
                    currentState <= nextState;
endmodule



module synImplicit (dataIn, dataOut, c1, c2, clock);
    input   [7:0]   dataIn, c1, c2;
    input           clock;
    output  [7:0]   dataOut;
    reg     [7:0]   dataOut, temp;

    always begin
        @ (posedge clock)
            temp = dataIn + c1;
        @ (posedge clock)
            temp = temp & c2;
```

```
        @ (posedge clock)
            dataOut = temp - c1;
    end
endmodule


module synPipe (dataIn, dataOut, c1, c2, clock);
    input   [7:0]   dataIn, c1, c2;
    input           clock;
    output  [7:0]   dataOut;
    reg     [7:0]   dataOut;

    reg     [7:0]   stageOne;
    reg     [7:0]   stageTwo;

    always  @ (posedge clock)
        stageOne <= dataIn + c1;

    always  @ (posedge clock)
        stageTwo <= stageOne & c2;

    always  @ (posedge clock)
        dataOut <= stageTwo + stageOne;
endmodule


module accumulate (qout, r, s, clock, reset);
    output  [11:0]  qout;
    input   [11:0]  r, s;
    input           clock, reset;
    reg     [11:0]  qout, q;

    initial
        forever @(negedge reset) begin
            disable main;
            qout <= 0;
        end

    always begin : main
        wait (reset);
        @ (posedge clock)
            q <= r + s;
        @ (posedge clock)
            qout <= q + qout;
    end
```

```
endmodule


module synSwitchFilter (Clock, reset, in, switch, out);
    input           Clock, reset, switch;
    input   [7:0]   in;
    output  [7:0]   out;
    reg     [7:0]   out, x1, x2, x3, y, yold, delta;

    initial forever @(negedge reset) begin
        disable main;
        out = 0;
         y = 1;
         x2 = 2;
         x3 = 3;
     end

    always begin :main
        wait (reset);
        @(posedge Clock)
        x1 = in;
        out <= y;
        yold = y;
        y = x1 + x2 + x3;
        delta = y - yold;
        delta = delta >> 1;
        if (switch == 1) begin
              delta = delta >> 1;
              @(posedge Clock) out <= out + delta;
              @(posedge Clock) out <= out + delta;
        end
        @(posedge Clock) out <= out + delta;
        x3 = x2;
        x2 = x1;
    end
endmodule



module firFilt (clock, reset, x, y);
    input           clock, reset;
    input   [7:0]   x;
    output  [7:0]   y;

    reg     [7:0]   coef_array [7:0];
```

```
    reg     [7:0]   x_array [7:0];
    reg     [7:0]   acc, y;

    reg     [2:0]   index, start_pos;
                    //important: these roll over from 7 to 0

    initial
      forever @ (negedge reset) begin
          disable firmain;
          start_pos = 0;
      end

    always begin: firmain
      wait (reset);
      @ (posedge clock);                    // State A;
      x_array[start_pos] = x;
      acc = x * coef_array[start_pos];
      index = start_pos + 1;
      begin :loop1
          forever begin
              @ (posedge clock);     // State B;
              acc = acc + x_array[index] * coef_array[index];
              index = index + 1;
              if (index == start_pos) disable loop1;
           end
      end // loop1
      y <= acc;
      start_pos = start_pos + 1;
    end
endmodule


module firFiltMealy (clock, reset, x, y);
    input           clock, reset;
    input   [7:0]   x;
    output  [7:0]   y;

    reg     [7:0]   coef_array [7:0];
    reg     [7:0]   x_array [7:0];
    reg     [7:0]   acc, y;
    reg     [2:0]   index, start_pos;

    initial
      forever @ (negedge reset) begin
          disable firmain;
```

```
            start_pos = 0;
            index = 0;
        end

    always begin: firmain
        wait (reset);
        begin: loop1
            forever begin
                @ (posedge clock);      // State 1 — the only state
                if (index == start_pos) begin
                    x_array[index] = x;
                    acc = x * coef_array[index];
                    index = index + 1;
                end
                else begin
                    acc = acc + x_array[index] * coef_array[index];
                    index = index + 1;
                    if (index == start_pos) disable loop1;
                end
            end
        end
        y <= acc;
        start_pos = start_pos + 1;
         index = start_pos;
    end
endmodule


primitive carry(carryOut, carryIn, aIn, bIn);
output  carryOut;
input   carryIn,
        aIn,
        bIn;

    table
        0     00   :   0;
        0     01   :   0;
        0     10   :   0;
        0     11   :   1;
        1     00   :   0;
        1     01   :   1;
        1     10   :   1;
        1     11   :   1;
    endtable
endprimitive
```

```
primitive carryX(carryOut, carryIn, aIn, bIn);
output  carryOut;
input   aIn,
        bIn,
        carryIn;

    table
        0     00  :  0;
        0     01  :  0;
        0     10  :  0;
        0     11  :  1;
        1     00  :  0;
        1     01  :  1;
        1     10  :  1;
        1     11  :  1;
        0     0x  :  0;
        0     x0  :  0;
        x     00  :  0;
        1     1x  :  1;
        1     x1  :  1;
        x     11  :  1;
    endtable
endprimitive


primitive carryAbbrev (carryOut, carryIn, aIn, bIn);
output  carryOut;
input   aIn,
        bIn,
        carryIn;

    table
        0     0?  :  0;
        0     ?0  :  0;
        ?     00  :  0;
        ?     11  :  1;
        1     ?1  :  1;
        1     1?  :  1;
    endtable
endprimitive
```

```
primitive latch (q, clock, data);
output  q;
reg     q;
input   clock, data;
    table
//              clock   data  state  output
                0       1   : ? :     1;
                0       0   : ? :     0;
                1       ?   : ? :     -;
    endtable
endprimitive
```

```
primitive dEdgeFF (q, clock, data);
output  q;
reg     q;
input   clock, data;

    table
//      clock     data    state    output
        (01)      0       : ? :    0;
        (01)      1       : ? :    1;
        (0x)      1       : 1 :    1;
        (0x)      0       : 0 :    0;
        (?0)      ?       : ? :    -;
         ?        (??)    : ? :    -;
    endtable
endprimitive
```

```
primitive dEdgeFFShort (q, clock, data);
output  q;
reg     q;
input   clock, data;

    table
//      clock         data  state  output
        r           0   : ? :    0;
        r           1   : ? :    1;
        (0x)        0   : 1 :    1;
        (0x)        1   : 1 :    1;
        (?0)        ?   : ? :    -;
```

```
        ?       *   : ? :   -;
    endtable
endprimitive


primitive jkEdgeFF (q, clock, j, k, preset, clear);
output  q;
reg     q;
input   clock, j, k, preset, clear;

    table
    //clock  jk   pc     state   output
    // preset logic
        ?     ??   01     : ? :     1;
        ?     ??   *1     : 1 :     1;

    // clear logic
        ?     ??   10     : ? :     0;
        ?     ??   1*     : 0 :     0;

    // normal clocking cases
        r     00   11     : ? :     -;
        r     01   11     : ? :     0;
        r     10   11     : ? :     1;
        r     11   11     : 0 :     1;
        r     11   11     : 1 :     0;
        f     ??   ??     : ? :     -;

    // j and k transition cases
        b     *?   ??     : ? :     -;
        b     ?*   ??     : ? :     -;

    //cases reducing pessimism
        p     00   11     : ? :     -;
        p     0?   1?     : 0 :     -;
        p     ?0   ?1     : 1 :     -;
        (x0)  ??   ??     : ? :     -;
        (1x)  00   11     : ? :     -;
        (1x)  0?   1?     : 0 :     -;
        (1x)  ?0   ?1     : 1 :     -;
        x     *0   ?1     : 1 :     -;
        x     0*   1?     : 0 :     -;
    endtable
endprimitive
```

```
module shreg (out, in, phase1, phase2);
/* IO port declarations, where 'out' is the inverse
of 'in' controlled by the dual-phased clock */

    output   out;     //shift register output
    input    in,      //shift register input
             phase1, //clocks
             phase2;

    tri      wb1, wb2, out;      //tri nets pulled up to VDD
    pullup   (wb1), (wb2), (out);//depletion mode pullup devices

    trireg   (medium) wa1, wa2, wa3;//charge storage nodes

    supply0      gnd;            //ground supply

    nmos #3      //pass devices and their interconnections
             a1(wa1, in, phase1), b1(wb1, gnd, wa1),
             a2(wa2, wb1, phase2), b2(wb2, gnd, wa2),
             a3(wa3, wb2, phase1), gout(out, gnd, wa3);
endmodule


module waveShReg;
    wire     shiftout;       //net to receive circuit output value
    reg      shiftin;        //register to drive value into circuit
    reg      phase1, phase2; //clock driving values

    parameter    d = 100;  //deine the waveform time step

    shreg    cct (shiftout, shiftin, phase1, phase2);

    initial
      begin :main
        shiftin = 0;     //initialize waveform input stimulus
        phase1 = 0;
        phase2 = 0;
        setmon;          // setup the monitoring information
        repeat(2)        //shift data in
            clockcct;
      end

    task setmon;     //display header and setup monitoring
```

```
        begin
            $display("              time   clks   in out wa1-3 wb1-2");
            $monitor ($time,,,,phase1, phase2,,,,,,shiftin,,,, shiftout,,,,,
                cct.wa1, cct.wa2, cct.wa3,,,,,cct.wb1, cct.wb2);
        end
    endtask

    task clockcct;     //produce dual-phased clock pulse
        begin
            #d phase1 = 1; //time step deined by parameter d
            #d phase1 = 0;
            #d phase2 = 1;
            #d phase2 = 0;
        end
    endtask
endmodule


module sram (dataOut, address, dataIn, write);
    output   dataOut;
    input    address, dataIn, write;
    tri      w1, w3, w4, w43;

    bufif1                g1(w1, dataIn, write);
    tranif1               g2(w4, w1, address);
    not (pull0, pull1)    g3(w3, w4), g4(w4, w3);
    buf                   g5(dataOut, w1);
endmodule

module wave_sram;      //waveform for testing the static RAM cell
    wire     dataOut;
    reg      address, dataIn, write;

    sram cell (dataOut, address, dataIn, write);

    parameter d = 100;
    initial begin
        #d dis;
        #d address = 1;      #d dis;
        #d dataIn = 1;       #d dis;
        #d write = 1;        #d dis;
        #d write = 0;        #d dis;
        #d write = 'bx;      #d dis;
        #d address = 'bx;    #d dis;
        #d address = 1;      #d dis;
```

```
        #d write = 0;          #d dis;
    end

    task dis;       //display the circuit state
        $display($time,, "addr=%v d_In=%v write=%v d_out=%v",
            address, dataIn, write, dataOut,
            " (134)=%b%b%b", cell.w1, cell.w3, cell.w4,
            " w134=%v %v %v", cell.w1, cell.w3, cell.w4);
    endtask
endmodule



module miniSim;

// element types being modeled
`define Nand 0
`define DEdgeFF 1
`define Wire 2

// literal values with strength:
//   format is 8 0-strength bits in decreasing strength order
//   followed by 8 1-strength bits in decreasing strength order
`define Strong0 16'b01000000_00000000
`define Strong1 16'b00000000_01000000
`define StrongX 16'b01111111_01111111
`define Pull0   16'b00100000_00000000
`define Pull1   16'b00000000_00100000
`define Highz0  16'b00000001_00000000
`define Highz1  16'b00000000_00000001

// three-valued logic set
`define Val0 3'd0
`define Val1 3'd1
`define ValX 3'd2

parameter//set  DebugFlags to 1 for message
   DebugFlags =          'b11000,
//                         |||||
// loading              <------+||||
//  event changes       <-------+|||
//  wire calc           <--------+||
//  evaluation          <---------+|
//  scheduling          <----------+
```

```
    IndexSize = 16,     //maximum size for index pointers
    MaxElements = 50,   //maximum number of elements
    TypeSize = 12;      //maximum number of types

reg [IndexSize-1:0]
    eventElement,           //output value change element
    evalElement,            //element on fanout
    fo0Index[1:MaxElements],  //irst fanout index of eventElement
    fo1Index[1:MaxElements],  //second fanout index of eventElement
    currentList,            //current time scheduled event list
    nextList,               //unit delay scheduled event list
    schedList[1:MaxElements]; //scheduled event list index
reg [TypeSize-1:0]
    eleType[1:MaxElements]; //element type
reg
    fo0TermNum[1:MaxElements],   //irst fanout input terminal number
    fo1TermNum[1:MaxElements],   //second fanout input terminal number
    schedPresent[1:MaxElements]; //element is in scheduled event list lags
reg [15:0]
    eleStrength[1:MaxElements], //element strength indication
    outVal[1:MaxElements],      //element output value
    in0Val[1:MaxElements],      //element irst input value
    in1Val[1:MaxElements],      //element second input value
    in0, in1, out, oldIn0;      //temporary value storage

integer pattern, simTime; //time keepers

initial
  begin
    // initialize variables
    pattern = 0;
    currentList = 0;
    nextList = 0;

    $display("Loading toggle circuit");
    loadElement(1, `DEdgeFF, 0, `Strong1,0,0, 4,0,0,0);
    loadElement(2, `DEdgeFF, 0, `Strong1,0,0, 3,0,0,0);
    loadElement(3, `Nand, (`Strong0|`Strong1),
       `Strong0,`Strong1,`Strong1, 4,0,1,0);
    loadElement(4, `DEdgeFF, (`Strong0|`Strong1),
       `Strong1,`Strong1,`Strong0, 3,0,1,0);

    // apply stimulus and simulate
    $display("Applying 2 clocks to input element 1");
    applyClock(2, 1);
```

```
    $display("Changing element 2 to value 0 and applying 1 clock");
    setupStim(2, `Strong0);
    applyClock(1, 1);

    $display("\nLoading open-collector and pullup circuit");
    loadElement(1, `DEdgeFF, 0, `Strong1,0,0, 3,0,0,0);
    loadElement(2, `DEdgeFF, 0, `Strong0,0,0, 4,0,0,0);
    loadElement(3, `Nand, (`Strong0|`Highz1),
       `Strong0,`Strong1,`Strong1, 5,0,0,0);
    loadElement(4, `Nand, (`Strong0|`Highz1),
       `Highz1,`Strong0,`Strong1, 5,0,1,0);
    loadElement(5, `Wire, 0,
       `Strong0,`Strong0,`Highz1, 7,0,1,0);
    loadElement(6, `DEdgeFF, 0, `Pull1,0,0, 7,0,0,0);
    loadElement(7, `Wire, 0,
       `Strong0,`Pull1,`Strong0, 0,0,0,0);

    // apply stimulus and simulate
    $display("Changing element 1 to value 0");
    pattern = pattern + 1;
    setupStim(1, `Strong0);
    executeEvents;
    $display("Changing element 2 to value 1");
    pattern = pattern + 1;
    setupStim(2, `Strong1);
    executeEvents;
    $display("Changing element 2 to value X");
    pattern = pattern + 1;
    setupStim(2, `StrongX);
    executeEvents;
  end

// Initialize data structure for a given element.
task loadElement;
input [IndexSize-1:0] loadAtIndex; //element index being loaded
input [TypeSize-1:0] type;         //type of element
input [15:0] strengthCoercion;     //strength speciication of element
input [15:0] oVal, i0Val, i1Val;   //output and input values
input [IndexSize-1:0] fo0, fo1;    //fanout element indexes
input fo0Term, fo1Term;            //fanout element input terminal indicators
  begin
    if (DebugFlags[4])
       $display(
          "Loading element %0d, type %0s, with initial value %s(%b_%b)",
          loadAtIndex, typeString(type),
```

```
            valString(oVal), oVal[15:8], oVal[7:0]);
    eleType[loadAtIndex] = type;
    eleStrength[loadAtIndex] = strengthCoercion;
    outVal[loadAtIndex] = oVal;
    in0Val[loadAtIndex] = i0Val;
    in1Val[loadAtIndex] = i1Val;
    fo0Index[loadAtIndex] = fo0;
    fo1Index[loadAtIndex] = fo1;
    fo0TermNum[loadAtIndex] = fo0Term;
    fo1TermNum[loadAtIndex] = fo1Term;
    schedPresent[loadAtIndex] = 0;
  end
endtask

// Given a type number, return a type string
function [32*8:1] typeString;
input [TypeSize-1:0] type;
    case (type)
      `Nand: typeString = "Nand";
      `DEdgeFF: typeString = "DEdgeFF";
      `Wire: typeString = "Wire";
      default: typeString = "*** Unknown element type";
    endcase
endfunction

// Setup a value change on an element.
task setupStim;
input [IndexSize-1:0] vcElement; //element index
input [15:0] newVal;          //new element value
  begin
    if (! schedPresent[vcElement])
      begin
        schedList[vcElement] = currentList;
        currentList = vcElement;
        schedPresent[vcElement] = 1;
      end
    outVal[vcElement] = newVal;
  end
endtask

// Setup and simulate a given number of clock pulses to a given element.
task applyClock;
input [7:0] nClocks;
input [IndexSize-1:0] vcElement;
    repeat(nClocks)
```

```
      begin
        pattern = pattern + 1;
        setupStim(vcElement, `Strong0);
        executeEvents;
        pattern = pattern + 1;
        setupStim(vcElement, `Strong1);
        executeEvents;
      end
endtask

// Execute all events in the current event list.
// Then move the events in the next event list to the current event
// list and loop back to execute these events. Continue this loop
// until no more events to execute.
// For each event executed, evaluate the two fanout elements if present.
task executeEvents;
reg [15:0] newVal;
  begin
    simTime = 0;
    while (currentList)
      begin
        eventElement = currentList;
        currentList = schedList[eventElement];
        schedPresent[eventElement] = 0;
        newVal = outVal[eventElement];
        if (DebugFlags[3])
           $display(
              "At %0d,%0d Element %0d, type %0s, changes to %s(%b_%b)",
              pattern, simTime,
              eventElement, typeString(eleType[eventElement]),
              valString(newVal), newVal[15:8], newVal[7:0]);
        if (fo0Index[eventElement]) evalFo(0);
        if (fo1Index[eventElement]) evalFo(1);
        if (! currentList) // if empty move to next time unit
          begin
            currentList = nextList;
            nextList = 0;
            simTime = simTime + 1;
          end
      end
  end
endtask

// Evaluate a fanout element by testing its type and calling the
// appropriate evaluation routine.
```

```
task evalFo;
input fanout; //irst or second fanout indicator
  begin
    evalElement = fanout ? fo1Index[eventElement] :
                           fo0Index[eventElement];
    if (DebugFlags[1])
       $display("Evaluating Element %0d type is %0s",
          evalElement, typeString(eleType[evalElement]));
    case (eleType[evalElement])
      `Nand: evalNand(fanout);
      `DEdgeFF: evalDEdgeFF(fanout);
      `Wire: evalWire(fanout);
    endcase
  end
endtask

// Store output value of event element into
// input value of evaluation element.
task storeInVal;
input fanout; //irst or second fanout indicator
  begin
    // store new input value
    if (fanout ? fo1TermNum[eventElement] : fo0TermNum[eventElement])
       in1Val[evalElement] = outVal[eventElement];
    else
       in0Val[evalElement] = outVal[eventElement];
  end
endtask

// Convert a given full strength value to three-valued logic (0, 1 or X)
function [1:0] log3;
input [15:0] inVal;
    casez (inVal)
      16'b00000000_00000000: log3 = `ValX;
      16'b???????0_00000000: log3 = `Val0;
      16'b00000000_???????0: log3 = `Val1;
      default:               log3 = `ValX;
    endcase
endfunction

// Convert a given full strength value to four-valued logic (0, 1, X or Z),
// returning a 1 character string
function [8:1] valString;
input [15:0] inVal;
    case (log3(inVal))
```

```
      `Val0: valString = "0";
      `Val1: valString = "1";
      `ValX: valString = (inVal & 16'b11111110_11111110) ? "X" : "Z";
    endcase
endfunction

// Coerce a three-valued logic output value to a full output strength value
// for the scheduling of the evaluation element
function [15:0] strengthVal;
input [1:0] logVal;
    case (logVal)
      `Val0: strengthVal = eleStrength[evalElement] & 16'b11111111_00000000;
      `Val1: strengthVal = eleStrength[evalElement] & 16'b00000000_11111111;
      `ValX: strengthVal = illBits(eleStrength[evalElement]);
    endcase
endfunction

// Given an incomplete strength value, ill the missing strength bits.
// The illing is only necessary when the value is unknown.
function [15:0] illBits;
input [15:0] val;
  begin
    illBits = val;
    if (log3(val) == `ValX)
      begin
        casez (val)
  16'b1???????_????????: illBits = illBits | 16'b11111111_00000001;
  16'b01??????_????????: illBits = illBits | 16'b01111111_00000001;
  16'b001?????_????????: illBits = illBits | 16'b00111111_00000001;
  16'b0001????_????????: illBits = illBits | 16'b00011111_00000001;
  16'b00001???_????????: illBits = illBits | 16'b00001111_00000001;
  16'b000001??_????????: illBits = illBits | 16'b00000111_00000001;
  16'b0000001?_????????: illBits = illBits | 16'b00000011_00000001;
        endcase
        casez (val)
  16'b????????_1???????: illBits = illBits | 16'b00000001_11111111;
  16'b????????_01??????: illBits = illBits | 16'b00000001_01111111;
  16'b????????_001?????: illBits = illBits | 16'b00000001_00111111;
  16'b????????_0001????: illBits = illBits | 16'b00000001_00011111;
  16'b????????_00001???: illBits = illBits | 16'b00000001_00001111;
  16'b????????_000001??: illBits = illBits | 16'b00000001_00000111;
  16'b????????_0000001?: illBits = illBits | 16'b00000001_00000011;
        endcase
      end
  end
```

```
endfunction

// Evaluate a 'Nand' gate primitive.
task evalNand;
input fanout; //irst or second fanout indicator
  begin
    storeInVal(fanout);
    // calculate new output value
    in0 = log3(in0Val[evalElement]);
    in1 = log3(in1Val[evalElement]);
    out = ((in0 == `Val0) || (in1 == `Val0)) ?
        strengthVal(`Val1) :
        ((in0 == `ValX) || (in1 == `ValX)) ?
            strengthVal(`ValX):
            strengthVal(`Val0);
    // schedule if output value is diSerent
    if (out != outVal[evalElement])
        schedule(out);
  end
endtask

// Evaluate a D positive edge-triggered lip lop
task evalDEdgeFF;
input fanout; //irst or second fanout indicator
    // check value change is on clock input
    if (fanout ? (fo1TermNum[eventElement] == 0) :
                 (fo0TermNum[eventElement] == 0))
      begin
        // get old clock value
        oldIn0 = log3(in0Val[evalElement]);
        storeInVal(fanout);
        in0 = log3(in0Val[evalElement]);
        // test for positive edge on clock input
        if ((oldIn0 == `Val0) && (in0 == `Val1))
          begin
            out = strengthVal(log3(in1Val[evalElement]));
            if (out != outVal[evalElement])
              schedule(out);
          end
      end
    else
        storeInVal(fanout); // store data input value
endtask

// Evaluate a wire with full strength values
```

```
task evalWire;
input fanout;
reg [7:0] mask;
  begin
    storeInVal(fanout);

    in0 = in0Val[evalElement];
    in1 = in1Val[evalElement];
    mask = getMask(in0[15:8]) & getMask(in0[7:0]) &
          getMask(in1[15:8]) & getMask(in1[7:0]);
    out = illBits((in0 | in1) & {mask, mask});

    if (out != outVal[evalElement])
       schedule(out);

    if (DebugFlags[2])
       $display("in0 = %b_%b\nin1 = %b_%b\nmask= %b %b\nout = %b_%b",
          in0[15:8],in0[7:0], in1[15:8],in1[7:0],
          mask,mask, out[15:8],out[7:0]);
  end
endtask

// Given either a 0-strength or 1-strength half of a strength value
// return a masking pattern for use in a wire evaluation.
function [7:0] getMask;
input [7:0] halfVal; //half a full strength value
   casez (halfVal)
     8'b???????1: getMask = 8'b11111111;
     8'b??????10: getMask = 8'b11111110;
     8'b?????100: getMask = 8'b11111100;
     8'b????1000: getMask = 8'b11111000;
     8'b???10000: getMask = 8'b11110000;
     8'b??100000: getMask = 8'b11100000;
     8'b?1000000: getMask = 8'b11000000;
     8'b10000000: getMask = 8'b10000000;
     8'b00000000: getMask = 8'b11111111;
   endcase
endfunction

// Schedule the evaluation element to change to a new value.
// If the element is already scheduled then just insert the new value.
task schedule;
input [15:0] newVal; // new value to change to
  begin
    if (DebugFlags[0])
```

```
        $display(
            "Element %0d, type %0s, scheduled to change to %s(%b_%b)",
            evalElement, typeString(eleType[evalElement]),
            valString(newVal), newVal[15:8], newVal[7:0]);
    if (! schedPresent[evalElement])
      begin
       schedList[evalElement] = nextList;
      schedPresent[evalElement] = 1;
       nextList = evalElement;
      end
    outVal[evalElement] = newVal;
  end
endtask
endmodule



module top;
      wire      d, e, f;
      reg       a, b, c;

      not       #1   g1(d, b);
      and       #2   g2(e, a, d);
      or        #2   g3(f, e, c);       // (AB)+C

      initial begin
          $monitor($time,, "a=%b, b=%b, c=%b, d=%b, e=%b, f=%b\n",
a, b, c , d, e, f );

          a = 1;                  // initialization
          b = 0;
          c = 0;
          #20 b = 1;              // irst change of input
          #20 a = 0;              // second change of input
          #20 c = 1;              // third change of input

          #20 $finish;
      end
endmodule


module halfadder (cOut, sum, a, b);
      output        cOut, sum;
      input         a, b;
```

```
    xor     #1  (sum, a, b);
    and     #2  (cOut, a, b);
endmodule


module system;
    wire    CarryOut, SumOut, in1, in2;

    halfadder   AddUnit (CarryOut, SumOut, in1, in2);
    testadder   TestUnit (in1, in2, CarryOut, SumOut);
endmodule


module testadder ( x, y, c, s );
    output      x, y;
    reg         x, y;
    input       c, s;

    initial begin
        $monitor($time,,
        "x = %b, y = %b, Sum = %b, Carry = %b", x, y, s, c);
                x = 0;
                y = 0;
        #10     x = 1;
        #10     y = 1;
        #10     x = 0;
        #10     $finish;
    end
endmodule


module andOr(f, a, b, c);
    input   a, b, c;
    output  f;
    reg     f;

    always @(a or b or c)
        if (c + (a & ~b))
            f = 1;
        else  f = 0;
endmodule


module BCDtoSevenSeg ( led, bcd);
    output  [7:0]   led;
```

```
    input    [3:0]   bcd;
    reg      [7:0]   led;

    always @(bcd)
        case (bcd)
            0 :   led = 'h81;
            1 :   led = 'hcf;
            2 :   led = 'h92;
            3 :   led = 'h86;
            4 :   led = 'hcc;
            5 :   led = 'ha4;
            6 :   led = 'ha0;
            7 :   led = 'h8f;
            8 :   led = 'h80;
            9 :   led = 'h8c;
            default:led = 'bxxxxxxxx;
        endcase
endmodule


module counter_2_bit (up, clk, rst, count);          //Answer to problem A.14
    input            up, clk, rst;// Declarations
    output   [1:0]   count;
    reg      [1:0]   count, nextCount;

    always @(up or count)
        case (count)
            0: begin
                if (up) nextCount = 1;
                else nextCount = 3;
            end
            1: begin
                if (up) nextCount = 2;
                else nextCount = 0;
            end
            2: begin
                if (up) nextCount = 3;
                else nextCount = 1;
            end
            3: begin
                if (up) nextCount = 0;
                else nextCount = 2;
            end
            default:
                nextCount = 0;
```

```
        endcase

    always @(posedge clk or negedge rst)
        if(~rst)
            count <= 0;
        else
            count <= nextCount;
endmodule


module dEdgeFF (q, clock, data);
    input   clock, data;
    output  q;
    reg     reset;
    wire    q, qBar, r, s, r1, s1;

    initial begin
        reset = 1;
        #20 reset = 0;
    end

    nor #10
        a (q, qBar, r, reset);
    nor
        b (qBar, q, s),
        c (s, r, clock, s1),
        d (s1, s, data),
        e (r, r1, clock),
        f (r1, s1, r);

endmodule
```