Chapter 6

(6.1)

“bar” is latched. Recode the design to remove the latch as follows.

```verilog
always @(foo or fred)
  if (foo == 2'h2) bar = fred;
  else bar = 1'bx;
```

(6.2)

“fred” and “mike” are latched. Recode the design to remove the latches as follows.

```verilog
always @(foo or hal or tron)
  begin
    fred = 1'bx;
    mike = 1'bx;
  case (foo)
    2'b00 : fred = hal;
    2'b01 : fred = tron;
    2'b10 : mike = hal;
    2'b11 : mike = tron;
  endcase
  end
```

(6.3)

```verilog
module counter (clock, in, latch, inc, ones);
  input clock;
  input [3:0] in;
  input latch;
  input inc;
  output ones;
  reg [3:0] value;
  wire ones;
  always @(posedge clock)
    if (latch)  value <= in;
    else if (inc && (!ones)) value <= value + 1'b1;
  assign ones = &value;
endmodule
```

(6.4)

a. See part b.
(6.5)

```verilog
module counter (clock, in, latch, dec, divide-by-two, zero);
input clock;  /* clock */
input [7:0] in;  /* input initial count */
input latch;  /* `latch input */
input dec;  /* decrement */
input divide-by-two;  /* divide by 2 when high */
output zero;  /* zero flag */
reg [7:0] value;  /* current count value */
wire zero;
always @(posedge clock)
begin
if (latch) value <= #1 in;
else if (dec && !zero) value <= #1 value - 1'b1;
else if (divide-by-two) value <= value >> 1;
end
assign zero = ~|value;
endmodule
```

(6.6)

```verilog
module count_compare (clock, in1, in2, in3, load1, load2, dec1, dec2,
count1, count2, ended);
input clock;
input [7:0] in1, in2, in3;
input load1;  // when load1 goes high, in1 is registered into count1
input load2;  // when load2 goes high, in2 is registered into count2
input dec1;  // decrement count1 by 1
input dec2;  // decrement count2 by in3
output [7:0] count1;  // contents of counter 1
output [7:0] count2;  // contents of counter 2
output ended;  // goes high when count1=count2, or either counter
  // experiences an unsigned overflow
```

b.

```
reg [4:0] addressA;
reg zero;
always @(posedge clock)
  AddressA = AddressA + 1'b1;
always @(posedge clock)  // RS flip-flop
  if (data == 32'b0) zero = 1;
  else if (AddressA=5'b0) zero = 0;
```
((6.6) continued)
// note: count1 and count2 will not change after same goes high until
// load1 or load2 forces one of the to be reloaded.

reg [7:0] count1, count2;
reg [7:0] next_count1, next_count2;
reg carry1, carry2;
reg carry1a, carry2a;
wire ended;

always @(posedge clock)
    if (load1) {carry1a, count1} <= {1'b0, in1};
    else if (dec1 & !ended) {carry1a, count1} <= {carry1, next_count1};

always @(count1)
    {carry1, next_count1} = count1 - 1'b1;

always @(posedge clock)
    if (load2) {carry2a, count2} <= {1'b0, in2};
    else if (dec2 & !ended) {carry2a, count2} <= {carry2, next_count2};

always @(count2)
    {carry2, next_count2} = count2 - in3;

assign ended = (count1 == count2) | carry1a | carry2a;
endmodule

module IntervalCounter (clock, Data, Interval, Overflow);

input clock;
input [7:0] Data;
output [7:0] Interval;
output Overflow;

reg [7:0] Interval;
reg Overflow;

always@ (posedge clock)
    begin
        Interval <= nextInterval;
        Overflow <= nextOverflow;
    end

always@ (Data or Interval)
    if (data[7:5] == 4'hF) {nextOverflow, nextInterval} = 9'h1;
    else {nextOverflow, nextInterval} = Interval + 1'b1;
endmodule
module DataPort (Clock, Reset, Clear, InData, Payload, Count, Error);

input Clock, Reset, Clear;
input [11:0] InData;
output [3:0] Payload;
output [7:0] Count, Error;

reg [3:0] Payload, nextPayload;
reg [7:0] Count, Error, nextCount, nextError;

wire parity;

always @(posedge Clock)
begin
    Payload <= nextPayload;
    Count <= nextCount;
    Error <= nextError;
end

always @(InData or Reset or Clear or Count)
if (!Reset || Clear)
begin
    nextPayload = 4'h0;
    nextCount = 8'h0;
    nextError = 8'h0;
end
else if (InData[11:8] == 4'h1)
begin
    nextPayload = InData[7:4];
    nextCount = Count + 1'b1;
    if (parity != InData[0]) nextError = Error + 1'b1;
    else nextError = Error;
end
else
begin
    nextPayload = Payload;
    nextCount = Count;
    nextError = Error;
end

assign parity = ~^InData[7:4];
endmodule

module parityCount(clock, data_in, go, odd, even);

input clock;
input [7:0] data_in;
input go;
output [7:0] odd;
output [7:0] even;

wire OddParity;
reg [7:0] odd, even, next_odd, next_even;
always@(posedge clock)
begin
    odd <= next_odd;
    even <= next_even;
end

always@(odd or even or OddParity or go)
if (go == 1'b0)
begin
    next_odd = 8'b0;
    next_even = 8'b0;
end
else
begin
    if (OddParity == 1'b1)
    begin
        next_even = even;
        next_odd = odd + 1;
    end
    else
    begin
        next_even = even + 1;
        next_odd = odd;
    end
end

assign OddParity = ^data_in;

endmodule
always@/(posedge clock)
if (!reset || clear) EvenParity <= 8’d0;
else EvenParity <= NextEvenParity;

always@/(posedge clock)
if (!reset || clear) GreyCode <= 8’d0;
else GreyCode <= NextGreyCode;

always@/(posedge clock)
if (!reset || clear) overflow <= 1’d0;
else overflow <= CarryOutParity | CarryOutGrey;

always@/(EvenParity or DataIn1 or DataIn2)
case ({~^DataIn1, ~^DataIn2}) // synopsys full_case parallel_case
2’b01 : {CarryOutParity, NextEvenParity} = EvenParity + 1’b1;
2’b10 : {CarryOutParity, NextEvenParity} = EvenParity + 1’b1;
2’b11 : {CarryOutParity, NextEvenParity} = EvenParity + 2’d2;
default : {CarryOutParity, NextEvenParity} = {1’b0,EvenParity} ;
endcase

always@/(GreyCode or DataIn1 or DataIn2)
{CarryOutGrey, NextGreyCode} = GreyCode +
((DataIn1 == 8’b10101010) | (DataIn1 == 8’b01010101)) +
((DataIn2 == 8’b10101010) | (DataIn2 == 8’b01010101));

endmodule

The testbench file is as follows:
`
#include “exercise610.v”
module test_fixture;
parameter TestCycles = 260; // # test cycles
parameter TC = 10; // # Clock period in ns
reg clock;
reg reset, clear;
reg [7:0] DataIn1Vectors [TestCycles-1:0];
reg [7:0] DataIn2Vectors [TestCycles-1:0];
reg [7:0] ExpectedEvenParity [TestCycles-1:0];
reg [7:0] ExpectedGreyCode [TestCycles-1:0];
reg [TestCycles-1:0] ExpectedOverflow;
reg [7:0] DataIn1;
reg [7:0] DataIn2;
wire overflow;
wire [7:0] EvenParity, GreyCode;
integer i,j;
initial // setup vectors
begin
  DataIn1Vectors [0] = 8’h03;
  DataIn1Vectors [1] = 8’hAA;
  DataIn1Vectors [2] = 8’h03;
  DataIn1Vectors [3] = 8’h03;
DataIn2Vectors [0] = 8'h03;
DataIn2Vectors [1] = 8'h01;
DataIn2Vectors [2] = 8'h01;
DataIn2Vectors [3] = 8'h01;

ExpectedEvenParity [0] = 8'h02;
ExpectedEvenParity [1] = 8'h03;
ExpectedEvenParity [2] = 8'h04;
ExpectedEvenParity [3] = 8'h05;

ExpectedGreyCode [0] = 8'h0;
ExpectedGreyCode [1] = 8'h1;
ExpectedGreyCode [2] = 8'h1;
ExpectedGreyCode [3] = 8'h1;

ExpectedOverflow [0] = 8'h0;
ExpectedOverflow [1] = 8'h0;
ExpectedOverflow [2] = 8'h0;
ExpectedOverflow [3] = 8'h0;

for (i=4; i<=TestCycles-1; i = i+1)
  begin
    DataIn1Vectors [i] = 8'h03;
    DataIn2Vectors [i] = 8'h01;
    {ExpectedOverflow[i], ExpectedEvenParity [i]} = ExpectedEvenParity[i-1] + 1'b1;
  end
end

initial //the following block executed only once
begin
  $shm_open("waves.shm");
  $shm_probe("AS");

  // insert monitor statement here
  $monitor("Time =%d;   Clock =  %b;   DataIn1 =  %h;   Evenparity =  %h; GreyCode = %h",
    $stime, clock, DataIn1, EvenParity, GreyCode);

  clock = 0;
  reset = 1;
  clear = 0;
  #6 DataIn1 = 8'h01;
  DataIn2 = 8'h01;
  #TC clear = 1;
  #TC clear = 0;
  #TC for (j=0; j<=TestCycles-1; j = j+1)
    begin
      DataIn1 = DataIn1Vectors[j];
      DataIn2 = DataIn2Vectors[j];
      #TC
      if (EvenParity != ExpectedEvenParity[j])
        $display("Error in Parity cycle %d
", j);
      if (GreyCode != ExpectedGreyCode[j])
        $display("Error in Grey Code
");
      if (overflow != ExpectedOverflow[j])
        $display("Error in Overflow 
");
    end
  $shm_close(); //closes data base
  $finish;
end

always #5 clock=¬clock; //10ns clock
AccumStats u1 (clock, reset, clear, DataIn1, DataIn2, EvenParity, GreyCode, overflow);

endmodule /* test_fixture */
module SaturatingAccumulator (clock, go, in, out);

input clock;
input go; // Accumulate inputs while go is high, clear when low
input [7:0] in;
output [7:0] out;
wire Cout, zero, saturate;
wire [7:0] Sum;
reg [7:0] out, next_out;
reg [2:0] count, next_count;

assign {Cout, Sum} = out + in;

always@(posedge clock)
out <= next_out;
always@(in or go or zero or Cout or Sum)
casex({go, zero, Cout, saturate})
  4'b0xxx : next_out = 8'b0;
  4'b11xx : next_out = in;
  4'b101x : next_out = 8'hFF;
  4'b10x1 : next_out = 8'hFF;
  default : next_out = Sum;
endcase
always@(posedge clock)
count <= next_count;
always@(go or count)
if (go==0) next_count = 3'h0;
else next_count = count + 1'b1;

assign zero = (count == 0);
assign saturate = (out == 8'hFF);
endmodule

(6.12)

Latches are accidently inferred; incomplete sensitivity list; “casex” should be used.

reg [1:0] Sel2;
reg in1, in2, out1, out2;

always@ (in1 or in2 or Sel2)
begin
out1 = in2;
out2 = in1;
casex (Sel2)
  2'b1x : begin out1 = in1; out2 = in2; end
default : ;
endcase
end
Chapter 7

(7.1) Omitted

(7.2) Omitted

(7.3) Omitted

(7.4) Omitted

(7.5) Omitted