Verilog I – II & Complexity Exercises

Paul D. Franzon
Verilog I

What is the timing produced by this test fixture?

```verilog
parameter CP=10;
initial
begin
    clock = 0;
    #(CP+1) In1 = 1'b0;
    #CP In2 = 1'b1;
    #(CP*5) $finish;
end
always #(CP/2) clock = ~clock;
always #(CP*2) In3=In1 & In2;
```
Verilog II

Review Question:
Convert the following into Verilog

Hint: \( \{\text{Cout, sum}\} = A + B; \)
Design the following controller for an Arithmetic Unit

```
assign {AddSub, And, Or} = Out;
always@(*) casex(IR[15:13])
  3'b001 : Out = 3'b100;
  3'b010 : Out = 3'b000;
  3'b011 : casex(In[2:0])
    3'b001 : Out = 3'b100;
    3'b010 : Out = 3'b010;
    3'b100 : Out = 3'b001;
    default : Out = x;
  default : Out = x;
endcase
```

<table>
<thead>
<tr>
<th>IR[15:13]</th>
<th>In[2:0]</th>
<th>{AddSub ; And, Or}</th>
</tr>
</thead>
<tbody>
<tr>
<td>3'b1</td>
<td>3'bx</td>
<td>100</td>
</tr>
<tr>
<td>3'b010</td>
<td>3'bx</td>
<td>000</td>
</tr>
<tr>
<td>3'b011</td>
<td>3'b001</td>
<td>100</td>
</tr>
<tr>
<td>3'b011</td>
<td>3'b010</td>
<td>010</td>
</tr>
<tr>
<td>3'b011</td>
<td>3'b100</td>
<td>001</td>
</tr>
<tr>
<td>All else</td>
<td>All else</td>
<td>xxx</td>
</tr>
</tbody>
</table>
Implement the following in Verilog95.

```
always@(A or rot or count)
casex (rot)
  000: B = A << count;
  001: B = A >> count;
  010: B = {A[15], A << count};
  011: B = {{16{A[15]}}, A} >> count;
  100: B = {A, A} >> count;
  101: B = {A, A} << count;
  default: B = x;
endcasex
count[3:0]
```

000: logical shift left
001: logical shift right
010: signed shift left
011: signed shift right
100: logical rotate left
101: logical rotate right
Example 2

Design and implement a simple Checksum error detector to work on a packet-based protocol. A packet of data usually contains a header, the data itself and a checksum. In this case the packet is organized as follows:

Byte 0 : 0101 0101 \This byte ONLY indicates that a correct packet is following Bytes 1-7 : Data
Byte 8 : Checksum, ie. sum of data rounded down to 8-bits

For example, the following packet has a correct checksum of 05 (01 + 01 + 01 + 01 + 01 + 01 + FF = 05 when truncated to 8 bits):

55 01 01 01 01 01 FF 05

while the following example has an error in the data, as the checksum is incorrect:

55 01 01 02 01 01 01 FF 05
... Example 2

Design and code in Verilog a synthesizable module that will detect the packet start and work out if the packet is correct or not. The I/O are:

module checksum (clock, reset, data, error);
    input clock;
    input reset;  \ reset active low
    input [7:0] data;
    output error;

    Error is to be set low by the global reset and is to remain low until an error is detected and then is to remain high until a new packet arrives.
    For example:

    reset 1 0 1
    data  0 0 0 0 55 01 01 02 01 01 FF 05 FF AA 55 01 01 02 01 01 01 FF 05
    error x x 0 0 0 0 1 1 1 0 0 0
Steps in design

1. Understand and draw IO
2. Work out strategy
3. Identify and name registers
4. Identify behavior of combinational logic blocks, and name any needed signals
5. Coding
   1. - note any named combinational outputs must be assigned in CL blocks or continuous assignment statements
Example 2 Solutions (UNVERIFIED)
Example 2 Solutions (unverified)
Example 3

You are going to build a serial demux unit. It will have one input port and seven output ports. All ports are 8-bits wide. Data will come in to the input port as packets with a variable length (up to 15 bytes), preceded by a header. A ‘newpacket’ signal will indicate arrival of a new packet. The high order 4-bits of the header byte will indicate the first port to be used as the output, while the lower 4-bits will indicate the packet length. Output ports will signal 0 when not in use.

The following input and output ports will be used:

```vhdl
input  clock;
input [7:0] DataIn;
input  NewPacket;
output [7:0] DataOut0, DataOut1, DataOut2, DataOut3, DataOut4,
            DataOut5, DataOut6, DataOut7;
```

A sample data stream is as follows:

<table>
<thead>
<tr>
<th>clock</th>
<th>0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataIn</td>
<td>00 73 11 22 33 44 00 02 88 99</td>
</tr>
<tr>
<td>NewPacket</td>
<td>0 1 0 0 0 0 0 0 1 0 0</td>
</tr>
<tr>
<td>DataOut0</td>
<td>00 00 00 00 00 22 00 00 00 00 88 00 00</td>
</tr>
<tr>
<td>DataOut1</td>
<td>00 00 00 00 00 33 00 00 00 00 99 00 00</td>
</tr>
<tr>
<td>DataOut2</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>DataOut3</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>DataOut4</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>DataOut5</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>DataOut6</td>
<td>00 00 00 00 00 11 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>DataOut7</td>
<td>00 00 00 11 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>
Example 3 Solutions (UNVERIFIED)
Example 3 Solutions (UNVERIFIED)

```verilog
always@(*)
begin
    DataOut0 = (go & decode[0]) ? DataInt : 8'h0;
    DataOut1 = (go & decode[1]) ? DataInt : 8'h0;
    DataOut2 = (go & decode[2]) ? DataInt : 8'h0;
    DataOut3 = (go & decode[3]) ? DataInt : 8'h0;
    DataOut4 = (go & decode[4]) ? DataInt : 8'h0;
    DataOut5 = (go & decode[5]) ? DataInt : 8'h0;
    DataOut6 = (go & decode[6]) ? DataInt : 8'h0;
    DataOut7 = (go & decode[7]) ? DataInt : 8'h0;
end

always@(posedge clock)
DataInt <= Data;

always@(posedge clock)
if (NewPacket == 1)
    port <= Data[6:4];
else
    port <= port + 1'b1;

always@(posedge clock)
if (NewPacket == 1)
    count <= Data[3:0];
else if (count>0) count <= count - 1'b1;

assign decode = 8'h01 << port;
assign go = (count != 0);
endmodule;
```
module MB (clock, A, B, Y);
  input  clock;
  input [6:0] A, B;
  output [12:0] Y;

  wire [5:0] Am, Bm;
  wire [5:0] PP1;
  wire [6:0] PP2;
  wire [7:0] PP3;
  wire [8:0] PP4;
  wire [9:0] PP5;
  wire [10:0] PP6;

  wire [7:0] PP12;
  wire [9:0] PP34;
  wire [9:0] PP1234;
  wire [11:0] PP56;

  reg [12:0] Y;

always@(posedge clock)

assign Am = A[5:0];
assign Bm = B[5:0];
assign PP1 = Bm[0] ? Am : 6'b0;
assign PP2 = Bm[1] ? {Am, 1'b0} : 7'b0;
assign PP3 = Bm[2] ? {Am, 2'b0} : 8'b0;
assign PP4 = Bm[3] ? {Am, 3'b0} : 9'b0;
assign PP5 = Bm[4] ? {Am, 4'b0} : 10'b0;
assign PP6 = Bm[5] ? {Am, 5'b0} : 11'b0;

assign PP12 = PP1 + PP2; // Uses the carry out
assign PP34 = PP3 + PP4; // Uses the carry out
assign PP56 = PP5 + PP6; // Uses the carry out
assign PP1234 = PP12 + PP34; // Carry out not needed

always@(posedge clock)
  Y[11:0] = PP1234 + PP56;
endmodule
Speed up the design by pipelining it
Complexity Notes

You most likely recall the concept of a Fibonacci sequence,

\[ F = \sum_{i=1}^{N} i \]

(e.g. If \( N=3 \), \( F=1+2+3=6 \)). Design and implement a module that whenever its input \( N \) changes, it produces \( F \), at latest \( N \) clock cycles later. \( N \) will be any 4-bit number (meaning that \( F \) has to be 7 bits long. \( N \) will not change while a new \( F \) is being calculated.

1. Sketch your design. Explicitly Separate control from datapath.
2. Code your design using Synthesizable Verilog.
module Fib (clock, N, Fib);
input clock; input [3:0]  N;output [6:0] Fib;
reg [2:0] count; reg state, next_state; // control lines
reg [1:0] Fmux, Cmux; // status lines
wire zero;

// Datapath
always@(posedge clock)
case(Fmux)
  2'h0 : Fib <= N;
  2'h1 : Fib <= N + count;
endcase
always@(posedge clock)
case (Cmux)
  2'h0 : count <= N-1;
  2'h1 : count <= count - 1;
endcase
assign zero = (count == 0);

// Controller
always@(posedge clock)
Nprev <= N;
always@(posedge clock)
state <= next_state;
always@(state or N or Nprev or zero)
case (state) // synopsys full_case parallel_case
  waitNp : begin
    Fmux =0; Cmux=0;if  (N == Nprev) next_state = waitNp;
    else   next_state = waitZ;
  end
  waitZ : begin
    Fmux = 1; Cmux = 1;
    if (!zero) next_state = waitZ;
    else next_state = waitNp;
  end
endcase
endmodule