2023

Exercise 2

TFE4171 - DESIGN OF DIGITAL SYSTEMS 2

SIMON DARGAHI & REZA NOROOZI

Table of Contents

Part A

Task A

```
`ifdef check1
property reset_asserted;
   @(posedge clk) rst |-> !data_out;
endproperty

reset_check: assert property(reset_asserted)
  $display($stime,,,"\t\tRESET CHECK PASS:: rst_=%b data_out=
 \n", rst, data_out);
else $display($stime,,,"\t\RESET CHECK FAIL:: rst_=%b data_out=
 \n",    rst, data_out);
`endif
```

Check 1 checks when positive clock edge, from low to high, that if rst is high, check that data_out is low. If true, do what's behind the property assert. If not, do the else.

```
#           15  rst=1 clk=1 validi=1 DIN=1 valido=0 DOUT=0
#           15        RESET CHECK PASS:: rst_=1 data_out=0
```

At time 15, we can see that rst is high, therefore it will at the same clock cycle check DOUT (also know as data_out). Because this is low, it will therefore pass the assert and print the pass message.

Task B

```
`ifdef check2

property validi_asserted;
    @(posedge clk) disable iff (rst) validi[*3] |=> valido;
endproperty

validi_check: assert property(validi_asserted)
  $display($stime,,,"\t validi_check TEST PASS");
else $display($stime,,,"\t validi_check TEST FAIL");
`endif

```

The validi_assert property first checks if validi is high for three clock cycles. If it is, it then checks that valido is high on the fourth cycle.

```
#         95  rst=0 clk=1 validi=0 DIN=8 valido=0 DOUT=0
#        105  rst=0 clk=1 validi=1 DIN=9 valido=0 DOUT=0
#        115  rst=0 clk=1 validi=1 DIN=10 valido=0 DOUT=0
#        125  rst=0 clk=1 validi=1 DIN=11 valido=0 DOUT=0
#        135  rst=0 clk=1 validi=0 DIN=12 valido=1 DOUT=101
#        135     validi_check TEST PASS
```

We get a passed because validi has been high for three clock cycles, then valido is high on the fourth.

Task C

```
`ifdef check3
property valido_asserted;
   @(posedge clk) valido |-> $past(validi, 1) && $past(validi, 2)  && $past(validi, 3) /* && $past(validi, 3) */ ;
endproperty

valido_check: assert property(valido_asserted)
else $display($stime,,,"\tVALIDO TEST FAIL:: Previus validi status =
 and     $past(validi, 1), $past(validi, 2), $past(validi, 3) );
`endif
",
```

It checks that if valido is high, then validi has been high the three past clock cyckles. It does this by using $past("variable", "how many clock cycles ago"). And checking each of them with &&.

```
# run -all
#          5  rst=0 clk=1 validi=1 DIN=1 valido=x DOUT=x
#         15  rst=1 clk=1 validi=1 DIN=1 valido=0 DOUT=0
#         25  rst=0 clk=1 validi=1 DIN=1 valido=0 DOUT=0
#         35  rst=0 clk=1 validi=0 DIN=2 valido=0 DOUT=0
#         45  rst=0 clk=1 validi=1 DIN=3 valido=0 DOUT=0
#         55  rst=0 clk=1 validi=0 DIN=4 valido=0 DOUT=0
#         65  rst=0 clk=1 validi=1 DIN=5 valido=0 DOUT=0
#         75  rst=0 clk=1 validi=1 DIN=6 valido=0 DOUT=0
#         85  rst=0 clk=1 validi=0 DIN=7 valido=0 DOUT=0
#         95  rst=0 clk=1 validi=0 DIN=8 valido=0 DOUT=0
#        105  rst=0 clk=1 validi=1 DIN=9 valido=0 DOUT=0
#        115  rst=0 clk=1 validi=1 DIN=10 valido=0 DOUT=0
#        125  rst=0 clk=1 validi=1 DIN=11 valido=0 DOUT=0
#        135  rst=0 clk=1 validi=0 DIN=12 valido=1 DOUT=101
#        145  rst=0 clk=1 validi=0 DIN=12 valido=0 DOUT=101
# ** Note: Data structure takes 4879708 bytes of memory
```

Because we do not get any instaces where valido has been high without having three validi cycles. We get nothing. But we tested it by adding a fourt statement, checking if validi has been high for four cycles. By doing this, we got an error at 135. By doing this, we know that the code works.

Task D

```
`ifdef check4
property data_out_assert;
   @(posedge clk) valido |-> data_out == ($past(data_in, 3) * $past(data_in, 2) + $past(data_in, 1));
endproperty
data_out_check: assert property(data_out_assert)
  $display($stime,,,"\tDOUT TEST PASS:: data_out should be =
 data_out($past(data_in, 3) * $past(data_in, 2) + $past(data_in, 1)), data_out);
else $display($stime,,,"\tDOUT TEST FAIL:: data_out should be =
 data_out($past(data_in, 3) * $past(data_in, 2) + $past(data_in, 1)), data_out);
`endif
```

Data_out_assert checks that if valido is high, check that data_out has the right value. Data out is calculated using the formula A*B+C where A is data_in value three clock cycles ago, B is two and C is one. We use the Past command do get the past results.

To make debugging easier, we also added the values if what data_out should be, and what it is in the assert.

```
#         95  rst=0 clk=1 validi=0 DIN=8 valido=0 DOUT=0
#        105  rst=0 clk=1 validi=1 DIN=9 valido=0 DOUT=0
#        115  rst=0 clk=1 validi=1 DIN=10 valido=0 DOUT=0
#        125  rst=0 clk=1 validi=1 DIN=11 valido=0 DOUT=0
#        135  rst=0 clk=1 validi=0 DIN=12 valido=1 DOUT=101
#        135    DOUT TEST PASS:: data_out should be =101 data_out is=101
#
#        145  rst=0 clk=1 validi=0 DIN=12 valido=0 DOUT=101
```

At 135 valido is high, it therefor checks that DOUT = DIN(105) * DIN(115) + DIN(125) <=> 101 = 9*10+11 = 101. Therefor we get a PASS.

Task E

```
#         95  rst=0 clk=1 validi=0 DIN=8 valido=0 DOUT=0
#        105  rst=0 clk=1 validi=1 DIN=9 valido=0 DOUT=0
#        115  rst=0 clk=1 validi=1 DIN=10 valido=0 DOUT=0
#        125  rst=0 clk=1 validi=1 DIN=11 valido=0 DOUT=0
#        135  rst=0 clk=1 validi=0 DIN=12 valido=1 DOUT=101
#        135    validi_check TEST PASS
#        145  rst=0 clk=1 validi=1 DIN=13 valido=0 DOUT=101
#        155  rst=0 clk=1 validi=1 DIN=14 valido=0 DOUT=101
#        165  rst=0 clk=1 validi=1 DIN=15 valido=0 DOUT=101
#        175  rst=0 clk=1 validi=1 DIN=16 valido=1 DOUT=197
#        175    validi_check TEST PASS
#        185  rst=0 clk=1 validi=1 DIN=17 valido=0 DOUT=197
#        185    validi_check TEST FAIL
#        195  rst=0 clk=1 validi=0 DIN=18 valido=0 DOUT=197
#        195    validi_check TEST FAIL
#        205  rst=0 clk=1 validi=1 DIN=19 valido=0 DOUT=197
#        215  rst=0 clk=1 validi=1 DIN=20 valido=0 DOUT=197
#        225  rst=0 clk=1 validi=1 DIN=21 valido=0 DOUT=197
#        235  rst=0 clk=1 validi=1 DIN=22 valido=1 DOUT=401
#        235    validi_check TEST PASS
#        245  rst=0 clk=1 validi=0 DIN=23 valido=0 DOUT=401
#        245    validi_check TEST FAIL
#        255  rst=0 clk=1 validi=0 DIN=24 valido=0 DOUT=401
#        265  rst=0 clk=1 validi=1 DIN=25 valido=0 DOUT=401
#        275  rst=0 clk=1 validi=1 DIN=26 valido=0 DOUT=401
#        285  rst=0 clk=1 validi=1 DIN=27 valido=0 DOUT=401
#        295  rst=0 clk=1 validi=1 DIN=28 valido=1 DOUT=677
#        295    validi_check TEST PASS
#        305  rst=0 clk=1 validi=1 DIN=29 valido=0 DOUT=677
#        305    validi_check TEST FAIL
#        315  rst=0 clk=1 validi=1 DIN=30 valido=0 DOUT=677
#        315    validi_check TEST FAIL
#        325  rst=0 clk=1 validi=0 DIN=31 valido=1 DOUT=842
#        325    validi_check TEST PASS
#        335  rst=0 clk=1 validi=0 DIN=31 valido=0 DOUT=842
```

Check 2 does one thing. It checks that if validi is high at any cycle, high the next cycle, and high the third cycle, THEN it will check valido is high for the past message, and low for the fail message.

Each of the FAIL messages comes because valido is low, when the three previous validi values have been high. Same with the pass. But with the pass message, vaildo has been high.

Each of these tests are also untreated to each other. So if we have four cycles of validi, it will run two checks of vaildo because there are.

The run_no_implication script sends a flag with the name "no_implication". This means that this part of the script in dut_properties.sv is ran

Task F

```
// S2
S2: begin
   if (validi) begin
  c = data_in;

  data_out <= a * b + c;

  a = b;
  b = c;

  valido <= 1'b1;


   end
   else begin
   valido <= 1'b0;

   next = S0;
   end
end
```

by adding "b" and "c" vaiable, it can change out A*B+C as mentioned that dout should be.

The only problem, is that I wanted to try something cool for Task D. this was to calculate the amount that would be needed to get a DOUT pass. The calculation that prints what number is needed, does no longer work correctly. And I cant find the reason at first glance. But seeing how I did it for fun, ill let it be in there and look at it at a later time to see if I can fix it after turning the lab in.

Part B

Task A

```
typedef struct
{
	rand bit[0:7] a;
	rand bit[0:7] b;
	rand bit[0:2] op;

} data_t;
```

We use typedef to indicate “bigger” data pools

Struct it just to define it’s a struct

At the end is the name to call the struct

Rand gives random value

Bit[x:y] behind rand, gives a random bit value between x and y

The values are gotten from alu_tb.sv

Name after bit size declaration is the name if the variable

Task B

```
	rand data_t data;
```

rand initializes it as randomized, with the name data. Struct is data_t

Task C

```
	task get(ref bit [0:7] a, ref bit [0:7] b, ref bit [0:2] op);
		a	= data.a;
		b	= data.b;
		op	= data.op;
	endtask
```

The get function is a pointer. Ref, refers to what being pointed at. Length behind the bit. Now we have new variables a, b and op that are defind within the task scope. These have the same values as data.a/b/op. but are allocated a different place in the machine storage.

Task D

```
        constraint c1
        {
                data.a inside{[0:127]};
        }

        constraint c2
        {
                data.b inside{[0:255]};
        }

        constraint c3
        {
                data.op inside{[0:6]};
        }
```

As the name implies, these are constraints that look that data. Values are within a given INSIDE a given value. This is for the randomize functuion. It makes it so that the function randomizes INSIDE the given value.

Part C

Task A

```
alu_data test_data[NUMBERS-1];
```

Spawning class(es) of alu_data from alu_packet.sv, with name test_data, [] makes it an array, with the length of “Numbers”. We use -1 because the array is zero index.

Task B

```
initial begin : data_gen
    #20

    for (int i = 0; i < NUMBERS; i++)
    begin
        test_data[i] = new();
        test_data[i].randomize();
        test_data[i].get(a, b, op);

        #20;
    end

end
```

Initial begin, begins initial with name data_gen

#20, wait 20 clock cycles.

For loop, start with int I being zero, while statement I smaller than numbers is true, and increment i by one each cycle.

Test data, with array index “i” is constructed with the “new” command.

The values within test_data with on this index is then randomized.

Then we use the get command to pull out the calues.

We then wait another 20 clock cycles.

Part D

Task A

```
typedef enum {ADD, SUB, NOT, NAND, NOR, AND, OR, XOR} opcode;
```

Typefed makes the enum values

Task B

```
//Make your covergroup here
covergroup alu_cg @(posedge clk);
    op_cg : coverpoint op
    {
                bins op_value[8] = {ADD, SUB, NOT, NAND, NOR, AND, OR, XOR};

    }

    a_cg : coverpoint a {
        bins zero = {0};
        bins little = {[1:50]}; //cant call it small due to small beeing defind
somewhere in the system
        bins hunds[2] ={100,200};
        bins big = {[200:$]}; //cant call it large
    }

    b_cg : coverpoint b;

    aXb_cg : cross b_cg, a;
endgroup
```

This is a covergroup that does checks each time the clock raises. Op_cg compares op_value array with 0-7 values with the logic gates made with the enum.

a_cg covers:

zero checks that the value of a is zero

little checks that the value is between 1 and 50

hund[0] checks that it is 100

hund[1] checks that it 200

big checks that it is 200 <= “value”

a coverpoint is made from b used in the cross between a and b

edit:

did some changes, and saw that there was no necessity to have b_cg. Did this instead

```
    aXb_cg : cross a, b;
```

Task C

```
alu_cg alu_cg_inst = new();
```

initialize a new covergroup form alu_cg with name alu_cg_inst

Task D

```
//Sample covergroup here
always @(posedge clk) alu_cg_inst.sample();
```

Sample the covergroup by using the sample task in always block.

But from testing the code, it look slike the @(postedge clk) used in the covergroup does the same

Part E

```
Coverpoint op_cg                    100.00%    100     -    Covered
    covered/total bins:                   8      8     -
    missing/total bins:                   0      8     -
    % Hit:                          100.00%    100     -
    bin op_value[0]                      12      1     -    Covered
    bin op_value[1]                       8      1     -    Covered
    bin op_value[2]                      28      1     -    Covered
    bin op_value[3]                      24      1     -    Covered
    bin op_value[4]                       8      1     -    Covered
    bin op_value[5]                       2      1     -    Covered
    bin op_value[6]                       8      1     -    Covered
    bin op_value[7]                      10      1     -    Covered
```

This is true, because in the transcript we see that every value are covered. Transcript op = “value”. The value goes from 0 to 7

```
Coverpoint a_cg                     40.00%    100     -    Uncovered
    covered/total bins:                  2      5     -
    missing/total bins:                  3      5     -
    % Hit:                          40.00%    100     -
    bin zero                             2      1     -    Covered
    bin little                          44      1     -    Covered
    bin hunds[0]                         0      1     -    ZERO
    bin hunds[1]                         0      1     -    ZERO
    bin big                              0      1     -    ZERO
```

```
    bin auto[100:103]                    2      1     -    Covered
```

```
61    clk=1 a=00110010
```

We hit what we expect from the transcript, just not 100. Im not sure why, when debugging and setting constrain to being value 100. We get it true, but when it shows at time 61 in the transcript that it came when the boundaries where bigger. We did not get it covered.

```
clk=1 a=00000000 b=00000000
```

```
Cross aXb_cg                                  1.22%        100
    covered/total bins:                          50       4096
    missing/total bins:                        4046       4096
    % Hit:                                    1.22%        100
```

Here we get the first init value to cross. That's why we get 1.22 prosentage.

Part D

Task A

```
module ex1_1 (
          input           clk, rst, validi,
          input [31:0]    data_in,
          output logic    valido,
          output logic [31:0] data_out,

          input     logic [31:0] alu_r_1, alu_r_2,
          output    logic [31:0] alu_a_1, alu_a_2,
          output    logic [31:0] alu_b_1, alu_b_2,
          output    logic [3:0] alu_op_1, alu_op_2

          );
```

```
      if (rst) begin
     data_out <= 32'b0;
     valido <= 1'b0;
     state = S0;

        alu_op_1 = 3'b010;
        alu_op_2 = 3'b000;
      end
```

```
        alu_a_1 = a;
         alu_b_1 = b;
         alu_a_2 = alu_r_1;
         alu_b_2 = data_out;


         data_out <= alu_r_2;
```

We addede signal for the alu

Then we set the op value, that is + and *

We then make it so that r1 = (a * b) and r2 = r1 + c with the pins

Task B

Somewhere along the way, we encounterd a problem we did not know how to solve. The dout did not show right.

```
#    125  rst=0 clk=1 validi=1 DIN=11 valido=0 DOUT=0
#    135  rst=0 clk=1 validi=0 DIN=12 valido=1 DOUT=z
#    135   DOUT TEST FAIL:: data_out should be =101 data_out is=z
#
#    145  rst=0 clk=1 validi=1 DIN=13 valido=0 DOUT=z
#    155  rst=0 clk=1 validi=1 DIN=14 valido=0 DOUT=z
#    165  rst=0 clk=1 validi=1 DIN=15 valido=0 DOUT=z
#    175  rst=0 clk=1 validi=1 DIN=16 valido=1 DOUT=z
```

Task C

I think it would take more time to compile in the simulation. But in real life it could take shorter. This is because the simulation both our version with the alu and partA v4 takes the same ns of time to compile. If the ALU where to work, it would take more. But in real life, it would be logicgates and not software that does this calculation. Or in other words, it would be more hardware doing the work. This way it would take shorter and use less power.