

Exercise 1

TFE4171 - DESIGN OF DIGITAL SYSTEMS 2

SIMON DARGAHI & REZA NOROOZI

Table of Contents

Part A	2
Task A 2	
Task B 3	
Task C 3	
Task D 4	
Task E 5	
Task F 6	
Part B	7
Task A 7	
Task B 7	
Task C 7	
Task D 8	
Part C	9
Task A 9	
Task B 9	
Part D	11
Task A 11	
Task B 11	

Part A

Task A

```
1 `ifdef check1
2 property reset_asserted;
3     @(posedge clk) rst |-> !data_out;
4 endproperty
5
6 reset_check: assert property(reset_asserted)
7     $display($time,, "\t\tRESET CHECK PASS:: rst_=%b data_out=
8     \n", rst, data_out);
9 else $display($time,, "\t\tRESET CHECK FAIL:: rst_=%b data_out=
10    \n",    rst, data_out);
11 `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.

```
1 #          15 rst=1 clk=1 validi=1 DIN=1 valido=0 DOUT=0
2 #          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

```
1 `ifdef check2
2
3 property validi_asserted;
4   @(posedge clk) disable iff (rst) validi[*3] | => valido;
5 endproperty
6
7 validi_check: assert property(validi_asserted)
8   $display($stime,, "\t validi_check TEST PASS");
9 else $display($stime,, "\t validi_check TEST FAIL");
10 `endif
11
```

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.

```
1 #          95  rst=0 clk=1 validi=0 DIN=8 valido=0 DOUT=0
2 #          105 rst=0 clk=1 validi=1 DIN=9 valido=0 DOUT=0
3 #          115 rst=0 clk=1 validi=1 DIN=10 valido=0 DOUT=0
4 #          125 rst=0 clk=1 validi=1 DIN=11 valido=0 DOUT=0
5 #          135 rst=0 clk=1 validi=0 DIN=12 valido=1 DOUT=101
6 #          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

```
1 `ifdef check3
2 property valido_asserted;
3   @(posedge clk) valido -> $past(validi, 1) && $past(validi, 2) && $past(validi, 3) /* && $past(validi, 3) */ ;
4 endproperty
5
6 valido_check: assert property(valido_asserted)
7 else $display($stime,, "\tVALIDO TEST FAIL:: Previous validi status =
8   and $past(validi, 1), $past(validi, 2), $past(validi, 3) );
9 `endif
10
```

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

```

1 # run -all
2 #      5  rst=0 clk=1 validi=1 DIN=1 valido=x DOUT=x
3 #      15 rst=1 clk=1 validi=1 DIN=1 valido=0 DOUT=0
4 #      25 rst=0 clk=1 validi=1 DIN=1 valido=0 DOUT=0
5 #      35 rst=0 clk=1 validi=0 DIN=2 valido=0 DOUT=0
6 #      45 rst=0 clk=1 validi=1 DIN=3 valido=0 DOUT=0
7 #      55 rst=0 clk=1 validi=0 DIN=4 valido=0 DOUT=0
8 #      65 rst=0 clk=1 validi=1 DIN=5 valido=0 DOUT=0
9 #      75 rst=0 clk=1 validi=1 DIN=6 valido=0 DOUT=0
10 #     85 rst=0 clk=1 validi=0 DIN=7 valido=0 DOUT=0
11 #     95 rst=0 clk=1 validi=0 DIN=8 valido=0 DOUT=0
12 #    105 rst=0 clk=1 validi=1 DIN=9 valido=0 DOUT=0
13 #    115 rst=0 clk=1 validi=1 DIN=10 valido=0 DOUT=0
14 #    125 rst=0 clk=1 validi=1 DIN=11 valido=0 DOUT=0
15 #    135 rst=0 clk=1 validi=0 DIN=12 valido=1 DOUT=101
16 #    145 rst=0 clk=1 validi=0 DIN=12 valido=0 DOUT=101
17 # ** Note: Data structure takes 4879708 bytes of memory

```

Because we do not get any instances where valido has been high without having three validi cycles. We get nothing. But we tested it by adding a fourth 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

```

1 `ifdef check4
2 property data_out_assert;
3   @(posedge clk) valido |-> data_out == ($past(data_in, 3) * $past(data_in, 2) + $past(data_in, 1));
4 endproperty
5 data_out_check: assert property(data_out_assert)
6   $display($time, "tDOUT TEST PASS:: data_out should be =
7   data_out($past(data_in, 3) * $past(data_in, 2) + $past(data_in, 1)), data_out);
8 else, $display($time, "tDOUT TEST FAIL:: data_out should be =
9   data_out($past(data_in, 3) * $past(data_in, 2) + $past(data_in, 1)), data_out);
10 `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 to get the past results.

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

```

1 #          95 rst=0 clk=1 validi=0 DIN=8 valido=0 DOUT=0
2 #         105 rst=0 clk=1 validi=1 DIN=9 valido=0 DOUT=0
3 #         115 rst=0 clk=1 validi=1 DIN=10 valido=0 DOUT=0
4 #         125 rst=0 clk=1 validi=1 DIN=11 valido=0 DOUT=0
5 #         135 rst=0 clk=1 validi=0 DIN=12 valido=1 DOUT=101
6 #         135 DOUT TEST PASS:: data_out should be =101 data_out is=101
7 #
8 #         145 rst=0 clk=1 validi=0 DIN=12 valido=0 DOUT=101

```

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

Task E

```

1 #          95 rst=0 clk=1 validi=0 DIN=8 valido=0 DOUT=0
2 #         105 rst=0 clk=1 validi=1 DIN=9 valido=0 DOUT=0
3 #         115 rst=0 clk=1 validi=1 DIN=10 valido=0 DOUT=0
4 #         125 rst=0 clk=1 validi=1 DIN=11 valido=0 DOUT=0
5 #         135 rst=0 clk=1 validi=0 DIN=12 valido=1 DOUT=101
6 #         135 validi_check TEST PASS
7 #         145 rst=0 clk=1 validi=1 DIN=13 valido=0 DOUT=101
8 #         155 rst=0 clk=1 validi=1 DIN=14 valido=0 DOUT=101
9 #         165 rst=0 clk=1 validi=1 DIN=15 valido=0 DOUT=101
10 #        175 rst=0 clk=1 validi=1 DIN=16 valido=1 DOUT=197
11 #        175 validi_check TEST PASS
12 #        185 rst=0 clk=1 validi=1 DIN=17 valido=0 DOUT=197
13 #        185 validi_check TEST FAIL
14 #        195 rst=0 clk=1 validi=0 DIN=18 valido=0 DOUT=197
15 #        195 validi_check TEST FAIL
16 #        205 rst=0 clk=1 validi=1 DIN=19 valido=0 DOUT=197
17 #        215 rst=0 clk=1 validi=1 DIN=20 valido=0 DOUT=197
18 #        225 rst=0 clk=1 validi=1 DIN=21 valido=0 DOUT=197
19 #        235 rst=0 clk=1 validi=1 DIN=22 valido=1 DOUT=401
20 #        235 validi_check TEST PASS
21 #        245 rst=0 clk=1 validi=0 DIN=23 valido=0 DOUT=401
22 #        245 validi_check TEST FAIL
23 #        255 rst=0 clk=1 validi=0 DIN=24 valido=0 DOUT=401
24 #        265 rst=0 clk=1 validi=1 DIN=25 valido=0 DOUT=401
25 #        275 rst=0 clk=1 validi=1 DIN=26 valido=0 DOUT=401
26 #        285 rst=0 clk=1 validi=1 DIN=27 valido=0 DOUT=401
27 #        295 rst=0 clk=1 validi=1 DIN=28 valido=1 DOUT=677
28 #        295 validi_check TEST PASS
29 #        305 rst=0 clk=1 validi=1 DIN=29 valido=0 DOUT=677
30 #        305 validi_check TEST FAIL
31 #        315 rst=0 clk=1 validi=1 DIN=30 valido=0 DOUT=677
32 #        315 validi_check TEST FAIL
33 #        325 rst=0 clk=1 validi=0 DIN=31 valido=1 DOUT=842
34 #        325 validi_check TEST PASS
35 #        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, valido 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 valido 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" variable, 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 of 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 defined 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 values.

We then wait another 20 clock cycles.

Part D

Task A

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

Typedef 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 being defined
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 $200 \leq \text{"value"}$

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 from alu_cg with name alu_cg_inst

Task D

```
56 //Sample covergroup here
57 always @(posedge clk) alu_cg_inst.sample();
58
59
```

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

But from testing the code, it look slike the @(posedge 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 added 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 encountered 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 were 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.