TFE4171 exercise 2 report

Morten Paulsen
Magnus Ramsfjell

March 12, 2020

1 Lab 7

1.1 Task a - check 0 and 1

Running `./run_check 0` gave us an understanding of how the design works. `./run_check 1` produced a test that passed because the module reset correctly.

Code added:

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

1.2 Task b - check 2

This exercise the property files were not kitted out with boilerplate code for the assertions. We decided to define properties for every check with the sequences inlined (not defined as their own sequences).

`check 2` asserts that `valido` is asserted one cycle after `validi` has been asserted for three consecutive cycles. When running the check the simulation never fails and passes at time 135. The pass indicated a successful assertion of `valido`.

Code added:

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

1.3 Task c - check 3

`check 3` ensures that `valido` is not asserted on the cycle that `validi` fell if `validi` was not asserted for three consecutive clock cycles. The simulation passes at times 35, 55 and 85 when `valido` remains low when `validi` falls. It does not fail with the current simulation, as expected.

Code added:

```
property valido_not_asserted;
    @(posedge clk) disable iff (rst)
      ($rose(validi) || validi && $past(rst)) ##0 validi  [*0:2] ##1 $fell(validi) |-> !valido;
endproperty
```

1.4 Task d - check 4

`check 4` checks that the calculation of `data_out` is carried out correctly. We achieved this by checking past values of `data_in` after `valido` is asserted. The simulation passes at time 135 when `data_out` is correctly computed to be $9 * 10 + 11 = 101$.

Code added:

```
property data_out_asserted;
    @(posedge clk) disable iff(rst)
      $rose(validi) ##0 validi [*3] ##1 valido |-> data_out == ($past(data_in,3) * $past(data_in,2) + $
endproperty
```

1.5 Task e - check 2 with more stimuli

Uncommenting the extended stimuli and running `check 2` again resulted in several new fails. The new fails are at times 185, 195, 245, 305 and 315 and all occur because `valido` is de-asserted after the first three consecutive high values of `validi`.

1.6 Task f - Fixing the state machine

Our solution of task f should be prefaced by mentioning that we found the problem to be somewhat ambiguous. The desired behavior of the system was unclear to us, and thus we made the following assumption:

After `validi` has been asserted for three consecutive clock cycles and a computation has completed, the result of this computation as well as as `valido` should be asserted until `validi` is de-asserted. This implies that two calculations can not be computed in six clock cycles, because an additional cycle is required to de- and re-assert `validi`.

To correct the design to adhere to our assumptions we added an additional state. This new state is entered and held when an output is computed but `validi` is still asserted. The old and new state diagrams are shown in figure 1.6.

Running `check 5` with the new design produced only passes. This confirms that the new design adheres to the system specifications our assumptions implied (and to a certain degree confirmed our assumptions).

Code changed:

```
from
    enum {S0, S1, S2} state = S0, next = S0;
to
    enum {S0, S1, S2, S3} state = S0, next = S0;

from:
    S2: begin
            if (validi) begin
                a += data_in;
                data_out <= a;
                valido <= 1'b1;
            end
            next = S0;
        end
to:
    S2: begin
            if (validi) begin
                a += data_in;
                data_out <= a;
                valido <= 1'b1;
                next = S3;
            end
            else
                next = S0;
        end
```

Code addded:

```
    S3: begin
            if (validi) begin
                data_out <= a;
                valido <= 1'b1;
```

```
            next = S3;
        end
        else
            next = S0;
    end
```

2 Lab 8

2.1 Task 2 - Setting up alu_packet.sv

We followed the instructions and set up the file `alu_packet.sv`. It is bundled with this pdf file in the final delivery.

2.2 Task 3 - Setting up alu_tb.sv

Once again we followed the instructions and set up `alu_tb.sv`. The new file generated a few seemingly (pseudo-)random numbers correctly.

2.3 Task 4 - Adding coverage

Following the instructions and simulating the added covergroup reported a coverage of 44.4%. It never hits the 7th bin of opcode, neither of the `hunds[]` nor the `large`-bin, and only 33 of 64 of the bins in b. We assume these bins are implicitly created by the cross coverage of a and b. The cross coverage itself reports a coverage of 1.2%.

If we manually run the simulation for longer (100 000 ns), the cross coverage of a and b increases to 46.1%. This is the maximum achievable coverage using our constraints.

3 Lab 9

3.1 Task a - Connecting FSM and ALUs

To use the updated finite state machine from subsection 1.6 with the ALU design from section 2 we created a top level module called `top.v`. In this module we create two instances of the ALU to allow for synchronous computation of the addition and multiplication operations. These ALUs are controlled by the FSM as shown in figure 3. *This block diagram is not the prettiest because it was generated by Vivado's Schematic viewer, but it does show how the modules are connected.*

3.2 Task b - Validate with assertions

We used the assertions from lab 7 to validate the functionality of the top module. It passed according to the assumptions stated in subsection 1.6. The results from the multiplications and additions are occasionally larger than what the 8-bit output can display, and in these cases the assertions still pass. We assume this is because the assertion only evaluates 8 bits of its multiplication and addition result because the value it is comparing to (`data_out`) and the values it uses for its internal computation (`$past(data_in, 1 to 3)`) are 8-bit values.

3.3 Task c - Coverage of top module using predefined stimuli

When running the coverage analysis from lab 8 on the top module with the test pattern stimuli we expect the coverage to decrease dramatically. This test uses fewer numbers and the numbers are not random.

Running the coverage analysis confirmed our expectations. The new coverage was 12.5% of `dut.alu1.op` and 12.5% of `dut.alu2.op`. The coverage did not work properly for the other signals, and we have no idea why.

We would expect the coverage to be much worse than the random stimuli used in lab 8 since the stimuli we used is much shorter and uses a lot of the same values over and over again.

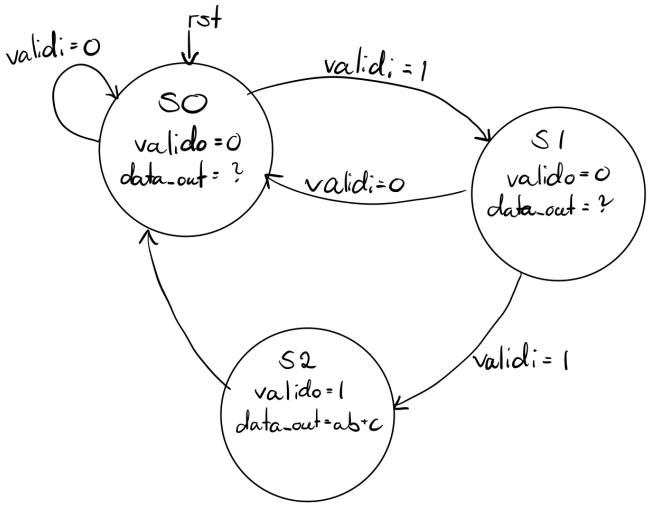

Figure 1: State diagram for the old design

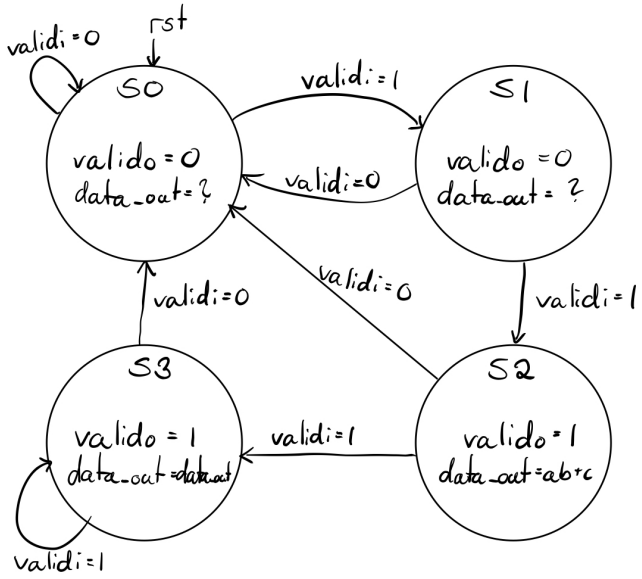

Figure 2: State diagram for the new design

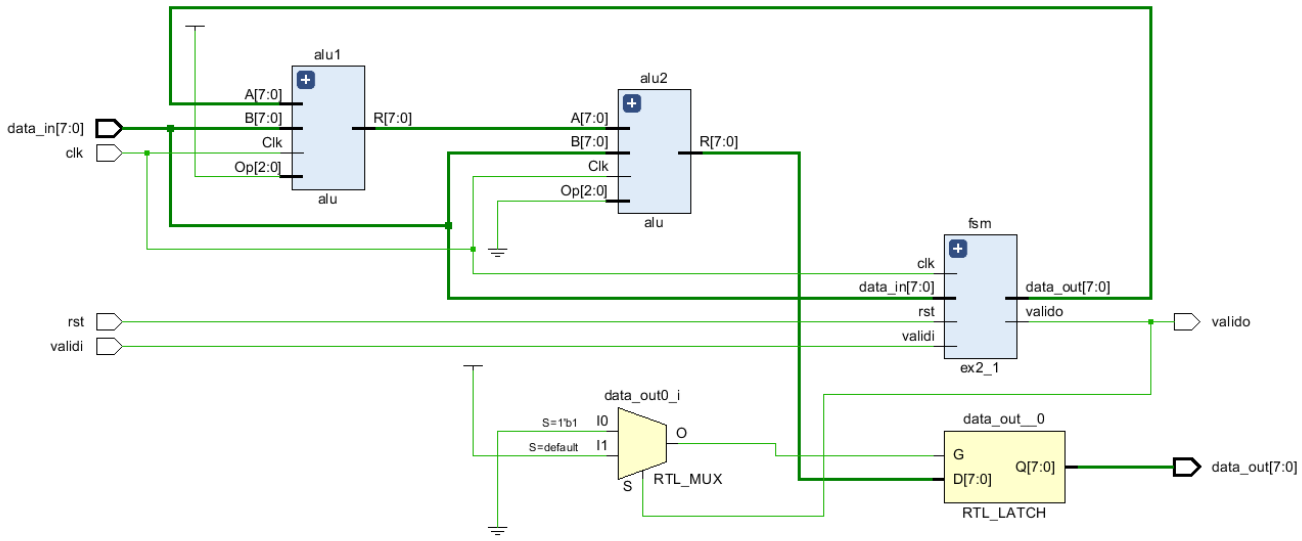

Figure 3: Block diagram for the top module