TFE4171 Exercise 3 Report

Morten Paulsen
Magnus Ramsfjell

March 12, 2020

1 D flip-flop

The folder contains the following files

- dff.vhd - Implementation of a D flip-flop
- dff.sva - Verification file for the D flip-flop

1.1 Verifying the D flip-flop

The assertion file contains two assertions connected to the behaviour of the DFF:

- a_behavior1
- a_behavior2

After checking them we found that a_behavor1 worked as intented, but a_behavior2 did not. By analyzing the counterexample we found out that the bug was that a_behavior2 checks the value of `d_i` two clock cycles earlier (`$past(d_i, 2)`) instead of after one clock cycle as the specifications are.

To fix the bug the following code was changed:

```
15 property p_behavior2;
16     q_o == $past(d_i, 2);
17 endproperty
```

Listing 1: Before change.

```
15 property p_behavior2;
16     q_o == $past(d_i, 1);
17 endproperty
```

Listing 2: After change.

2 JK flip-flop

The folder contains the following files

- jkff.vhd - Implementation of a JK flip-flop
- jkff.sva - Verification file for the JK flip-flop

2.1 Verifying the JK flip-flop

To verify the JK flip-flop we implemented four sequences to represent the four states the JK flip-flop can have - `seq_j`, `seq_k`, `seq_jk` and `seq_jk_n`. And to verify its functionality we implemented four properties - `p_j`, `p_k`, `p_jk` and `p_jk_n` - and asserted them by checking them on every positive clockedge of `clk`.

```
sequence seq_j;
    j_i ##0 !j_k;
endsequence

sequence seq_k;
    !j_i ##0 j_k;
endsequence

sequence seq_jk;
    j_i ##0 j_k;
endsequence

sequence seq_jk_n;
    !j_i ##0 !j_k;
endsequence
```

Listing 3: The implemented sequences.

```
property p_j;
    seq_j |=> q_o;
endproperty

property p_k;
    seq_k |=> !q_o;
endproperty

property p_jk;
    seq_jk |=> q_o == !$past(q_o);
endproperty

property p_jk_n;
    seq_jk_n |=> $stable(q_o);
endproperty

// Make assertions on properties to be checked
a_only_j:  assert property (@(posedge clk) p_j);
a_only_k:  assert property (@(posedge clk) p_k);
a_both_jk: assert property (@(posedge clk) p_jk);
a_none_jk: assert property (@(posedge clk) p_jk_n);
```

Listing 4: The implemented properties and their assertions.

3 ATM - Asynchronous Transfer Mode

The folder contains the follow files

- atm.vhd - Implementation of an ATM
- atm.sva - Verification file for the ATM

3.1 Verifying the ATM

To verify each of the specifications listed below, we six sequences and five properties as shown in Listing 5 and Listing 6.

1. A cell is never corrected and dismissed at the same time.
2. An error-free cell is neither corrected or dismissed.
3. All cells with multiple-bit errors are dismissed.
4. A first erroneous cell coming in is corrected if the error is a single-bit error and not a multiple-bit error.
5. A second erroneous cell is always dismissed.

The assertions are similar to the ones in Listing 4.

In the last two properties (`p_correct_first` and `p_dismiss_second`) we use the keyword `implies` instead of `|->` or `|=>`, the difference between their functionality is that `implies` uses the same starting point in the consequent as in the antecedent while `|->` and `|=>` continues where the antecedent stopped.

For the last specification (A second erroneous cell is always dismissed.) we had to modify the implementation of the ATM. Before the modification the `dismiss_o` signal did not take it into consideration the state of the ATM which it had to do to know wether or not it was the second erroneous cell in a row.

```
dismiss_o <= error_i and multiple_i;
```

Listing 7: Before change.

```
dismiss_o <= (error_i and multiple_i) or (state_s and error_i);
```

Listing 8: After change.

```
sequence seq_not_correct_and_dismiss;
        !correct_o ##0 !dismiss_o;
endsequence

sequence seq_correct_and_dismiss;
        correct_o ##0 dismiss_o;
endsequence

sequence seq_error_free;
        !error_i ##0 !multiple_i;
endsequence

sequence seq_multiple;
        error_i ##0 multiple_i;
endsequence

sequence seq_first_error;
        !error_i ##1 error_i;
endsequence

sequence seq_second_error;
        error_i ##1 error_i;
endsequence
```

Listing 5: The implemented sequences.

```
property p_not_both;
    not seq_correct_and_dismiss;
endproperty

property p_error_free;
    seq_error_free |-> seq_not_correct_and_dismiss;
endproperty

property p_dismiss_mult;
    seq_multiple |-> dismiss_o;
endproperty

property p_correct_first;
    (seq_first_error ##0 !multiple_i) implies ##1 correct_o;
endproperty

property p_dismiss_second;
    (seq_second_error implies ##1 dismiss_o;
endproperty
```

Listing 6: The implemented properties.

4 Busarbiter

The folder contains the follow files

- busarbiter-package.vhd - Type definitions and constants
- busarbiter.vhd - Implementation of a busarbiter
- bus-system.vhd - Interconnects between the arbiter and its masters and slave
- busmaster.vhd - Implementation of a simple busmaster
- busslave.vhd - Implementation of a simple busslave
- busarbite.sva - Verification file for the busarbiter

4.1 p_reset

When the busarbiter is reset it should be ready to accept new requests from the masters and it should not be granting any masters access to the bus, so `state_s == READY` and `bus_grant == NO_GRANT`.

```
property p_reset;
    reset |-> bus_grant == NO_GRANT ##0 state_s == READY;
endproperty
```

Listing 9: The implementation of the property p_reset.

4.2 p_at_most_one_grant

```
27 property p_at_most_one_grant;
28     $onehot0(bus_grant);
29 endproperty
```

Listing 10: The implementation of the property p_at_most_one_grant.

4.3 p_grant_stable

```
31 property p_grant_stable;
32     ($rose(bus_grant[0]) || $rose(bus_grant[1]) || $rose(bus_grant[2])) |=> $stable(bus_grant) [*0:$] ##0 bus_ack;
33 endproperty
```

Listing 11: The implementation of the property p_grant_stable.

4.4 p_arbitration_master

```
35 property p_arbitration_master0;
36     bus_req[0] ##0 state_s == READY |=> bus_grant[0];
37 endproperty
```

Listing 12: The implementation of the property p_arbitration_master0.

```
39 property p_arbitration_master1;
40     bus_req[1:0] == 2'b10 ##0 state_s == READY |=> bus_grant[1];
41 endproperty
```

Listing 13: The implementation of the property p_arbitration_master1.

```
43 property p_arbitration_master2;
44     bus_req[2:0] == 3'b100 ##0 state_s == READY |=> bus_grant[2];
45 endproperty
```

Listing 14: The implementation of the property p_arbitration_master2.

4.5 p_grant_master

```
47 property p_grant_master1;
48     $rose(bus_grant[1]) implies $past(bus_req[1:0]) == 2'b10;
49 endproperty
```

Listing 15: The implementation of the property p_grant_master1.

```
51 property p_grant_master2;
52     $rose(bus_grant[2]) implies $past(bus_req[2:0]) == 3'b100;
53 endproperty
```

Listing 16: The implementation of the property p_grant_master2.

5 Processor

The folder contains the following files

- proc-package.vhd - Type definitions, constants and helper functions
- proc.vhd - Entity declaration of the processor module
- proc-seq.vhd - Architecture of the processor module
- proc.tda - Verification file for the processor

This part of the exercise uses TiDAL as a framework for creating sequences in a easy-to-read format as the sequences are quite complex.

5.1 ADD_IMM, OR_IMM, ADD_REG & OR_REG

Since all of the processor instructions `ADD_IMM`, `OR_IMM`, `ADD_REG` and `OR_REG` uses all five execution phases we wanted to freeze the input in phase 0 (Instruction Fetch) and check that the functionality was correct in the next cycles. We checked that it iterated through all the phases and that the contents of register RD was correct in the cycle after Write-Back phase.

```
49  property p_or_imm;
50      logic [2:0] rs1;
51      logic [2:0] rd;
52      logic [7:0] imm_ext;
53      logic [7:0] contents_rs1;
54      logic [7:0] contents_rd;
55
56      t ##0 set_freeze(rs1, instrIn[11:9]) and
57      t ##0 set_freeze(rd, instrIn[8:6]) and
58      t ##0 set_freeze(imm_ext, {instrIn[5], instrIn[5], instrIn[5:0]}) and
59      t ##0 set_freeze(contents_rs1, REG_FILE[rs1]) and
60      t ##0 set_freeze(contents_rd, (rd==3'b000)? (REG_FILE[rd]) : (rs1==3'b000)? imm_ext : (contents_rs1 | imm_ext)) and
61
62      t ##0 CONTROL_STATE == c_IF and
63      t ##0 instrIn[15:12] == c_OR_IMM
64
65      implies
66
67      t ##1 CONTROL_STATE == c_ID and
68      t ##2 CONTROL_STATE == c_EX and
69      t ##3 CONTROL_STATE == c_MEM and
70      t ##4 CONTROL_STATE == c_WB and
71      t ##5 REG_FILE[rd] == contents_rd;
72  endproperty
```

Listing 17: The implementation of the property p_or_imm.

```
property p_add_imm;
    logic [2:0] rs1;
    logic [2:0] rd;
    logic [7:0] imm_ext;
    logic [7:0] contents_rs1;
    logic [7:0] contents_rd;

    t ##0 set_freeze(rs1, instrIn[11:9]) and
    t ##0 set_freeze(rd, instrIn[8:6]) and
    t ##0 set_freeze(imm_ext, {instrIn[5], instrIn[5], instrIn[5:0]}) and
    t ##0 set_freeze(contents_rs1, REG_FILE[rs1]) and
    t ##0 set_freeze(contents_rd, (rd==3'b000)? (REG_FILE[rd]) : (rs1==3'b000)? imm_ext : (contents_rs1 + imm_ext)) and

    t ##0 CONTROL_STATE == c_IF and
    t ##0 instrIn[15:12] == c_ADD_IMM

    implies

    t ##1 CONTROL_STATE == c_ID and
    t ##2 CONTROL_STATE == c_EX and
    t ##3 CONTROL_STATE == c_MEM and
    t ##4 CONTROL_STATE == c_WB and
    t ##5 REG_FILE[rd] == contents_rd;
endproperty
```

Listing 18: The implementation of the property p_add_imm.

```
property p_or_reg;
    logic [2:0] rs1;
    logic [2:0] rs2;
    logic [2:0] rd;
    logic [7:0] contents_rs1;
    logic [7:0] contents_rs2;
    logic [7:0] contents_rd;

    t ##0 set_freeze(rs1, instrIn[11:9]) and
    t ##0 set_freeze(rs2, instrIn[8:6]) and
    t ##0 set_freeze(rd, instrIn[5:3]) and
    t ##0 set_freeze(contents_rs1, (rs1==3'b000)? (8'b00000000) : (REG_FILE[rs1])) and
    t ##0 set_freeze(contents_rs2, (rs2==3'b000)? (8'b00000000) : (REG_FILE[rs2])) and
    t ##0 set_freeze(contents_rd, (rd==3'b000)? (REG_FILE[rd]) : (contents_rs1 | contents_rs2)) and

    t ##0 CONTROL_STATE == c_IF and
    t ##0 instrIn[15:12] == c_ALU_REG and
    t ##0 instrIn[2:0] == c_OR

    implies

    t ##1 CONTROL_STATE == c_ID and
    t ##2 CONTROL_STATE == c_EX and
    t ##3 CONTROL_STATE == c_MEM and
    t ##4 CONTROL_STATE == c_WB and
    t ##5 REG_FILE[rd] == contents_rd;
endproperty
```

Listing 19: The implementation of the property p_or_reg.

```
property p_add_reg;
    logic [2:0] rs1;
    logic [2:0] rs2;
    logic [2:0] rd;
    logic [7:0] contents_rs1;
    logic [7:0] contents_rs2;
    logic [7:0] contents_rd;

    t ##0 set_freeze(rs1, instrIn[11:9]) and
    t ##0 set_freeze(rs2, instrIn[8:6]) and
    t ##0 set_freeze(rd, instrIn[5:3]) and
    t ##0 set_freeze(contents_rs1, (rs1==3'b000)? (8'b00000000) : (REG_FILE[rs1])) and
    t ##0 set_freeze(contents_rs2, (rs2==3'b000)? (8'b00000000) : (REG_FILE[rs2])) and
    t ##0 set_freeze(contents_rd, (rd==3'b000)? (REG_FILE[rd]) : (contents_rs1 + contents_rs2)) and

    t ##0 CONTROL_STATE == c_IF and
    t ##0 instrIn[15:12] == c_ALU_REG and
    t ##0 instrIn[2:0] == c_ADD

    implies

    t ##1 CONTROL_STATE == c_ID and
    t ##2 CONTROL_STATE == c_EX and
    t ##3 CONTROL_STATE == c_MEM and
    t ##4 CONTROL_STATE == c_WB and
    t ##5 REG_FILE[rd] == contents_rd;
endproperty
```

Listing 20: The implementation of the property p_add_reg.

5.2 LOAD & STORE

For the `LOAD` and `STORE` instructions we want to assert that the inputted instruction is executed correctly by checking that `writeEnable` is kept deasserted for `LOAD` and is asserted only in memory phase for `STORE`. For the `LOAD` instruction we check that value of RD in the cycle after the Write-Back phase is the same as `dataIn` in the memory phase. Instead of checking the incoming address we check that the outgoing data and address is correct in the memory phase in `STORE`.

```
property p_load;
    logic [2:0] rs1;
    logic [2:0] rd;
    logic [7:0] contents_rs1;
    logic [7:0] contents_rd;
    logic [7:0] imm_ext;
    logic [7:0] addr;
    logic [7:0] din;

    t ##0 set_freeze(rs1, instrIn[11:9]) and
    t ##0 set_freeze(rd, instrIn[8:6]) and
    t ##0 set_freeze(imm_ext, {instrIn[5], instrIn[5], instrIn[5:0]}) and
    t ##0 set_freeze(contents_rs1, (rs1==3'b000)? (8'b00000000) : (REG_FILE[rs1])) and
    t ##0 set_freeze(addr, contents_rs1 + imm_ext) and

    t ##0 CONTROL_STATE == c_IF and
    t ##0 instrIn[15:12] == c_LOAD and

    t ##3 set_freeze(contents_rd, (rd==3'b000)? (REG_FILE[rd]) : (dataIn))

    implies

    t ##0 writeEnable == 1'b0 and
    t ##1 CONTROL_STATE == c_ID and
    t ##1 writeEnable == 1'b0 and
    t ##2 CONTROL_STATE == c_EX and
    t ##2 writeEnable == 1'b0 and
    t ##3 CONTROL_STATE == c_MEM and
    t ##3 writeEnable == 1'b0 and
    t ##3 dataAddr == addr and
    t ##4 CONTROL_STATE == c_WB and
    t ##4 writeEnable == 1'b0 and
    t ##5 REG_FILE[rd] == contents_rd;
endproperty
```

Listing 21: The implementation of the property p_load.

```
property p_store;
    logic [2:0] rs1;
    logic [2:0] rs2;
    logic [7:0] contents_rs1;
    logic [7:0] contents_rs2;
    logic [7:0] imm_ext;
    logic [7:0] addr;
    logic [7:0] din;

    t ##0 set_freeze(rs1, instrIn[11:9]) and
    t ##0 set_freeze(rs2, instrIn[8:6]) and
    t ##0 set_freeze(imm_ext, {instrIn[5], instrIn[5], instrIn[5:0]}) and
    t ##0 set_freeze(contents_rs1, (rs1==3'b000)? (8'b00000000) : (REG_FILE[rs1])) and
    t ##0 set_freeze(contents_rs2, (rs2==3'b000)? (8'b00000000) : (REG_FILE[rs2])) and
    t ##0 set_freeze(addr, contents_rs1 + imm_ext) and

    t ##0 CONTROL_STATE == c_IF and
    t ##0 instrIn[15:12] == c_STORE

    implies

    t ##0 writeEnable == 1'b0 and
    t ##1 CONTROL_STATE == c_ID and
    t ##1 writeEnable == 1'b0 and
    t ##2 CONTROL_STATE == c_EX and
    t ##2 writeEnable == 1'b0 and
    t ##3 CONTROL_STATE == c_MEM and
    t ##3 writeEnable == 1'b1 and
    t ##3 dataOut == contents_rs2 and
    t ##3 dataAddr == addr and
    t ##4 CONTROL_STATE == c_WB and
    t ##4 writeEnable == 1'b0;
endproperty
```

Listing 22: The implementation of the property p_store.

5.3 JUMP & BRANCH

The instructions `BRANCH` and `JUMP` are relatively similar in the fact that they both only execute the Instruction Fetch and Instruction Decode phases and that they modify the PC (`instrAddr`). The difference between them is that where `JUMP` always modifies the PC, `BRANCH` only modifies the PC when the contents of the register in the instruction is zero.

We use `instrAddr == instrAddr + 16'd2` instead of `instrAddr == instrAddr + 2` because `2` is a 32bit value and `16'd2` is 16bit so that both addends have the same size.

```
property p_jump;
    logic [15:0] offset;
    logic [15:0] prev_pc;

    t ##0 set_freeze(offset, {{4{instrIn[11]}}, instrIn[11:0]}) and
    t ##0 set_freeze(prev_pc, instrAddr) and

    t ##0 CONTROL_STATE == c_IF and
    t ##0 instrIn[15:12] == c_JUMP

    implies

    t ##1 CONTROL_STATE == c_ID and
    t ##2 CONTROL_STATE == c_IF and
    t ##2 instrAddr == prev_pc + offset + 16'd2;
endproperty
```

Listing 23: The implementation of the property p_jump.

```
property p_branch;
    logic [15:0] offset;
    logic [15:0] new_pc;
    logic [2:0] rs1;
    logic [7:0] contents_rs1;

    t ##0 set_freeze(rs1, instrIn[11:9]) and
    t ##0 set_freeze(contents_rs1, (rs1==3'b000)? (8'b00000000) : (REG_FILE[rs1])) and
    t ##0 set_freeze(offset, {{7{instrIn[8]}}, instrIn[8:0]}) and
    t ##0 set_freeze(new_pc, (contents_rs1==8'b00000000)? (instrAddr + offset + 16'd2) : (instrAddr + 16'd2)) and

    t ##0 CONTROL_STATE == c_IF and
    t ##0 instrIn[15:12] == c_BRANCH

    implies

    t ##1 CONTROL_STATE == c_ID and
    t ##2 CONTROL_STATE == c_IF and
    t ##2 instrAddr == new_pc;
endproperty
```

Listing 24: The implementation of the property p_branch.