



Norwegian University of Science and
Technology
Department of Electronics and
Telecommunication

TFE4171
Design of Digital Systems 2
Spring 2019

Exercise set 3

Delivery time: Friday, March 15, 16:00h.

About the exercises

The exercise sets 3 and 4 will be run on the **venus.tele.ntnu.no** computer, using the software OneSpin 360 DV, which is a property checker. It can be accessed from the computers on the DAKLAB (B-320) or you can **ssh -X** into it if you are on the NTNU network.

You can call the OneSpin software from the console by executing the **onespin** command from the console. Please, try it once and exit the software immediately, just to see if you have access. If not, it may be necessary to update your **.bashrc**. You can obtain a fresh copy by typing this command:

```
cp -r /home/courses/desdigsys2/2019s/dd2master/.bashrc .
```

Next, you need to copy the project files into your home directory.

```
cp -r /home/dstoffel/projects .
```

The **projects** directory holds several working directories containing the project files for the various lab tasks, namely: **dff** (for “D-flipflop”), **jkff** (for “JK-flipflop”), **atm**, **arbiter**, **processor** and **readserial**. The relevant working directory is denoted at the beginning of each lab description.

Lab 1

[Working directory: **dff**]

Task 1.1

Using the handout material, familiarize yourself with the general methodology of formal property checking with **onespin**. How is the hardware model generated? How are assertions entered into the verification system? How is the property checker started, what results are produced and how do you evaluate them?

Go through the example of the D-flipflop that was discussed in class.

- Start **onespin**, change your working directory to **dff** and load the VHDL design for the D-flipflop. Elaborate and compile the design and then switch to module verification mode.
- Load both assertions in the file **dff.sva** and check them.

- Analyze any counterexample that is found by the tool. Where is the bug?



Lab 2

[Working directory: [jkff](#)]

Task 2.1

A synchronous JK-flipflop (Fig. 1) has two control inputs **j_i** and **k_i** and one clock input **clk**. The control inputs are evaluated only with the rising edge of the clock (clock event). If the **j_i** input is set (**j_i** = '1') at the clock event, the output **q_o** is set to **q_o** = '1'. If the **k_i** input is set (**k_i** = '1') at the clock event the output **q_o** is set to **q_o** = '0'. If both control inputs are set at the clock event the output **q_o** toggles, i.e., it switches to the opposite logic value. Finally if **j_i** and **k_i** are both equal to '0' then the value of the output does not change.

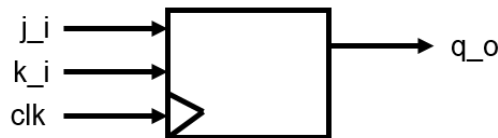


Figure 1: JK-flipflop

Write a set of assertions to capture the above specified behavior of JK-flipflop and prove them:

- Restart **onespin** and change your working directory to the subdirectory **jkff**.
- Edit the file **jkff.sva** to formulate your assertions.
- Run the property checker to prove the assertions.

Lab 3

[Working directory: [atm](#)]

In this assignment you will become more familiar with the use of the language SVA in formal property checking.

Given: the VHDL description of a **part** of an ATM controller (ATM = *Asynchronous Transfer Mode*, a telecommunication protocol). Its behavior depends not only on the current input but also on previous inputs. The controller's task is to classify incoming ATM messages (called *cells*) based on the results of a separate CRC checker (CRC = *Cyclic Redundancy Check*). ATM cells consist of a header, a CRC block and a data block. The error detection using CRC allows to detect and correct single-bit errors and also to detect multiple-bit errors.

The controller determines for each incoming ATM cell whether the cell needs to be corrected or whether it is to be dismissed. The controller receives information about the outcome of error detection from the CRC checker. (Note that the CRC checker is specified in a separate module which is not considered in this assignment.)

Design description (atm.vhd):

In every clock cycle a new ATM cell is processed and examined by the error checker.

Input signals (binary, “**bit**”):

- error_i**: set if the ATM cell is erroneous (single-bit or multiple-bit error)
multiple_i: set if and only if the error is a multiple-bit error

Note: In case of a multiple-bit error both input signals are set.

Output signals (binary, “**bit**”):

- correct_o**: ATM cell is to be corrected
dismiss_o: ATM cell is to be dismissed

Fig. 2 shows the state transition diagram of the ATM controller. The labels at the edges of the graph denote sets of input and output value pairs in the following format:

error_i multiple_i / correct_o dismiss_o

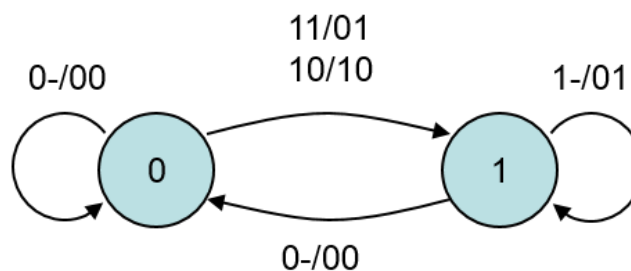


Figure 2: State transition graph of the ATM packet monitor

Specification of the design

1. A cell is never corrected and dismissed at the same time.
2. An error-free cell is neither corrected nor 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.

**Task 3.1**

For each item of the specification above, formulate an assertion and prove it using **onespin**. For the last two specifications (4th and 5th) describe the properties using the SVA sequence logical operator **implies** (how does **implies** differ from the temporal implications \rightarrow , \Rightarrow ?).



Analyze any counterexample found by the tool, and identify whether it is caused by an erroneous design or by an erroneous assertion; then correct the error.

Lab 4

[Working directory: [arbiter](#)]

In this task you will prove the correctness of the implementation of an *arbiter* design.

Design Description

Fig. 3 shows the arbiter within its environment. The environment of the arbiter consists of two *masters* and a *resource* (such as a memory or peripheral device). From time to time the masters request access to the resource. At any point in time only a single master may use the resource. The arbiter ensures this by granting access to the resource only to one master at a time.

The arbitration policy is as follows: whenever two masters request access simultaneously, the master who received the previous grant *is denied* access and the other master is granted access to the resource.

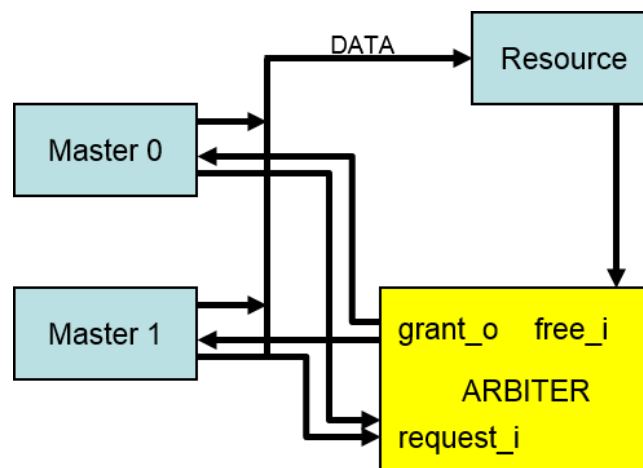


Figure 3: Arbiter with environment

Input signals:

clk: `std_logic` – clock signal
reset: `std_logic` – reset signal (active high)
free_i: `std_logic` – resource is available
request_i: `std_logic_vector(1 downto 0)` of request signals, one for each master

Output signals:

grant_o: `std_logic_vector(1 downto 0)` of grant signals, one for each master.

Specification:

Either master k may request access to the resource using its signal **request_i(k)**. If the resource is free and there is only one master requesting access, it is granted access at the next clock

event. If the resource is free and both masters request access (“competing request”), the one that did not receive the previous grant is granted access. If the resource is busy, i.e., not free, the masters have to wait.

If the first request occurring after reset is a competing request then master 0 is to be granted access.

For solving the tasks of this assignment, you may either use a design of your own conforming to this specification, or you may use the design in the **arbiter** subdirectory.

In order to verify the complete behavior of the design one has to write several assertions that each checks part of the behavior. The set of assertions has to be set up such that all possible behaviors are covered by at least one assertion in this set. This can be ensured by referring to important states of the design. In our example, all transactions begin at an important starting state (e.g., IDLE) and also end at that state. This is reflected in the assertions. First, we prove that the circuit can be set into its starting state using the reset facility. Then, we cover all the circuit functionality by writing assertions with a cause-effect structure. The assertions begin and end in the starting state, verifying the correct input/output relationship along the intermediate state transitions. We also call this kind of assertion *operational property*. From now on we use word *property* to refer to the *operational property* and use *assertion* to refer to other kinds of assertions.

Note:

Employ the SVA sequence logical operator **implies** (cause-effect operation) for writing all properties of this assignment.

Task 4.1

Write a property that proves the reset behavior. In what state should the circuit be if the reset is activated? What are the outputs of the circuit after reset?



The file **arbiter.sva** may help you set up the proof module.

Task 4.2

Write a property to check that a first request immediately after reset is granted to master 0 in case the request is a competing one.

Note:

Maybe you need a constraint to the environment of arbiter for this task. The assume property statement may help you to establish a constraint. Formulate the constraint in a property, and use **assume property** (**@(posedge clk)** *<property name>*) to assume that a property is always true.

Task 4.3

Write a property stating that if there is no request there will be no grant.

Task 4.4

Prove the behavior of the arbiter for the case that the resource is free and there is a single request from one master. Assume that the circuit starts in the IDLE state and prove that it returns to the IDLE state again (after how many clock cycles?).



Task 4.5

Write a property that verifies the correct behavior if two masters request access simultaneously, and prove it using **onespin**.

Extend your property to show that arbitration switches between the two masters if two consecutive competing requests occur.

Lab 5

[Working directory: [processor](#)]

Your task is to verify a sequential implementation (i.e., an implementation without pipelining) of a simple microprocessor (CPU). The processor is a simplified version of the DLX processor by Hennessy and Patterson [Hennessy/Patterson: “Computer Architecture – A Quantitative Approach”, Morgan Kaufmann Publishers, ISBN 1-55860-372-7]. The DLX processor is also covered in Peter Ashenden’s book “The Designer’s Guide to VHDL”.

Description of the processor

Compared to the DLX, the instruction set of this processor is reduced to 8 instructions. Instead of 32 registers, this processor has only 8. Also, the width of the registers and the data bus is reduced to 8 bit. This reduction helps improving the run time of the Onespin proof engine.

Just like in the MIPS processor, execution of an instruction is carried out in the following steps.

IF phase	Instruction Fetch cycle
ID phase	Instruction Decode / Register Fetch cycle
EX phase	Execution / Effective Address cycle
MEM phase	Memory Access
WB phase	Write-Back cycle

Jumps and branches are executed in the ID phase.

Instruction set

The instruction set comprises the following 8 instructions.

Instruction	Transfer
ADD_REG	$rd := rs1 + rs2$
OR_REG	$rd := rs1 \text{ or } rs2$
ADD_IMM	$rd := rs1 + \text{sign_ext}(\text{Immediate})$
OR_IMM	$rd := rs1 \text{ or } \text{sign_ext}(\text{Immediate})$
LOAD	$rd := \text{MEM}[rs1 + \text{sign_ext}(\text{Immediate})]$
STORE	$\text{MEM}[rs1 + \text{sign_ext}(\text{Immediate})] := rs2$
JUMP	$PC := PC + 2 + \text{sign_ext}(\text{Offset})$
BRANCH	if $rs1 = 0$ then $PC := PC + 2 + \text{sign_ext}(\text{Offset})$ else $PC := PC + 2$

An instruction is encoded in 16 bits. The specific encoding of each instruction can be found in the following list.

ADD_REG

adds the contents of two source registers and stores the result in the destination register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	rs1			rs2			rd			0	0	1

Transfer: $rd := rs1 + rs2$

OR_REG

computes the bitwise logical OR of the contents of two source registers and stores the result in the destination register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	rs1			rs2			rd			0	1	0

Transfer: $rd := rs1 \text{ or } rs2$

ADD_IMM

adds an immediate 6-bit value (specified in the instruction code itself) to the contents of a source register and stores the result in the destination register. Before being used, the immediate 6-bit value is sign-extended to 8 bits.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	rs1			rd			Immediate					

Transfer: $rd := rs1 + \text{sign_ext}(\text{Immediate})$

OR_IMM

computes the bitwise logical OR of the contents of a source register and an immediate 6-bit value (specified in the instruction code itself) and stores the result in the destination register. Before being used, the immediate 6-bit value is sign-extended to 8 bits.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	rs1			rd			Immediate					

Transfer: $rd := rs1 \text{ or } \text{sign_ext}(\text{Immediate})$

LOAD

loads one byte from memory and stores it in the destination register. The memory address is computed by adding an immediate 6-bit value (specified in the instruction code itself) to the contents of a source register. Before being used, the immediate 6-bit value is sign-extended to 8 bits.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	rs1			rd			Immediate					

Transfer: $rd := \text{MEM}[rs1 + \text{sign_ext}(\text{Immediate})]$

STORE

stores the contents of the second source register in memory. The memory address is computed by adding an immediate 6-bit value (specified in the instruction code itself) to the contents of the first source register. Before being used, the immediate 6-bit value is sign-extended to 8 bits.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	rs1			rs2			Immediate					

Transfer: $\text{MEM}[\text{rs1} + \text{sign_ext}(\text{Immediate})] := \text{rs2}$

JUMP

performs an unconditional branch. The destination address is computed by adding the constant 2 and the offset (specified in the instruction code itself) to the program counter (PC). Before being used, the 12-bit offset is sign-extended to 16 bits.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	Offset											

Transfer: $\text{PC} := \text{PC} + 2 + \text{sign_ext}(\text{Offset})$

BRANCH

performs a conditional branch. The branch is taken if the content of the source register is 0. The destination address is computed by adding the constant 2 and the immediate value (specified in the instruction code itself) to the program counter (PC). Before being used, the 9-bit offset is sign-extended to 16 bits.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	rs1			Offset								

Transfer: if $\text{rs1} = 0$
 then $\text{PC} := \text{PC} + 2 + \text{sign_ext}(\text{Offset})$
 else $\text{PC} := \text{PC} + 2$

External Bus Interface

The external bus consists of the following signals:

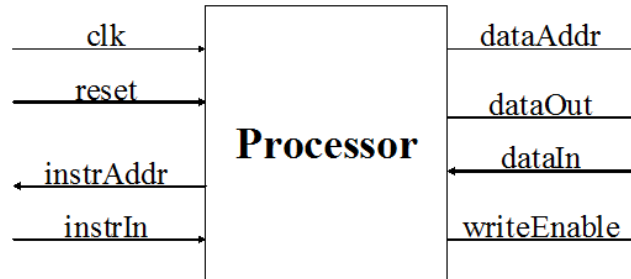


Figure 4: Processor interface

clk

The system clock (input).

reset

Asynchronous reset (input).

instrAddr

The memory address of the next instruction to be read (output, during the IF phase).

instrIn

The instruction read (during the IF phase, input).

dataAddr

The memory address of data in LOAD and STORE instructions (output, during MEM phase).

dataOut

The data sent to memory in LOAD and STORE instructions (output, during MEM phase).

dataIn

The data read from memory in LOAD and STORE instructions (output, during MEM phase).

writeEnable

Active during the MEM phase of STORE instructions.

Internal Signals

In the following we describe some of the internal registers of the processor implementation.

REG_FILE

The set of registers available in the CPU.

Note: There are in total 8 registers in the register file, however, only registers 1 through 7 are actually used to hold arbitrary 8-bit contents. Register 0 of the register file is not used by the implementation and therefore its content is unknown. R0 as an operand to an instruction always yields the constant 0. (It does not yield the content of register file at position 0, see the VHDL code.)

PC

The program counter.

IR

The instruction register.

A

During the EX phase, register A holds the contents of the first source register.

B

During the EX phase, register B holds the contents of the second source register.

CONTROL_STATE

A state variable containing the current phase being executed.

Introduction to TiDAL (Timing Diagram Assertion Library)

In the last assignment you may have experienced that writing operational properties using plain SVA can easily become complicated. In the following we use extended library (TiDAL) to SVA which aids in writing structured easy-to-read operational properties that clearly document cause and effect in operations. In addition, TiDAL, as its name suggests, can directly translate timing diagrams from the specification into formal properties. Because it is a SVA library complying to the standard it can be used with many tools supporting SVA.

Structure of TiDAL properties

In this section we will explain the syntax and semantics of TiDAL based on an example. A TiDAL property has a cause-effect structure, where the cause part and the effect part are separated by the keyword “**implies**” (introduced in the 2009 SystemVerilog standard). Note that this implication operator differs from the implicators “**|->**” and “**|=>**” in that “**implies**” evaluates the antecedent sequence and the consequent sequence starting from the same time point.

```
1: module proc_property_suite(reset, clk, instrIn, REG_FILE, CONTROL_STATE);
2:   input logic reset;
3:   input logic clk;
4:   input logic [15:0] instrIn;
5:   input logic [7:0] [7:0] REG_FILE;
6:   input logic [2:0] CONTROL_STATE;
7:   ...
8:   parameter c_IF = 3'b001;
9:   parameter c_ID = 3'b010;
10:  parameter c_EX = 3'b011;
11:  parameter c_MEM = 3'b100;
12:  parameter c_OR_IMM = 4'b0011;
13:  ...
14:  `include "tidal.sv"
15:  `begin_tda(ops)
16:    ...
17:  property or_imm;
18:    logic [2:0] rs1;
19:    logic [2:0] rd;
```

```

20: logic [5:0] imm;
21: logic [7:0] content_rs1;
22:
23: t ##0 set_freeze(rs1, instrIn[11:9]) and
24: t ##0 set_freeze(rd, instrIn[ 8:6]) and
25: t ##0 set_freeze(imm, instrIn[ 5:0]) and
26: t ##0 set_freeze(content_rs1_t, REG_FILE[rs1_t]) and
27:
28: t ##0 CONTROL_STATE == c_IF and
29: t ##0 instrIn[15:12] == c_OR_IMM
30:
31: implies
31:
32: t ##1 CONTROL_STATE == c_ID and
33: t ##2 CONTROL_STATE == c_EX and
34: ...;
35: endproperty
36: ...
37: a_or_imm: assert property(@(posedge clk) disable iff (reset==1) or_imm);
38: `end_tda
39: ...
40: endmodule
41: ...
42: bind proc proc_property_suite inst_proc_property_suite(.*)

```

As you can see in the above example, the module containing TiDAL properties and the corresponding bind statement are specified just like in plain SVA. The formal arguments of this module, parameters, clocking and disable conditions are also the same as in plain SVA. Here are some explanations to the above code.

Line 14: Before you use TiDAL expressions you need to include the TiDAL library.

Line 15 and line 38: TiDAL properties need to be enclosed in `'begin_tda` and `'end_tda` macros. The `'begin_tda` macro takes a string as parameter, `ops` in our example. This is done to mark properties as TiDAL “operational properties” which enables some optimizations for the property checker. Standard SVA properties can also be used inside a TiDAL block, however they should be marked by `'begin_sva` and `'end_sva` macros.

Lines 17, 35: Properties are declared and ended in the same way as in SVA.

Lines 18 to 21: Declaration of temporal variables which are local to this property. Later these variables are used as *freeze variables* (explained below).

Lines 23 to line 29: The TiDAL keyword `t` represents an arbitrary time point used as a reference to specify temporal relationships. The notation `t ##N` is used to refer to a time point at an offset of N clock ticks from the reference time point t , in other words, “at time point $t+N$ ”. N can be any natural number including zero. In the TiDAL property notation, every occurrence of `t##N` starts a new temporal/Boolean expression which is, syntactically, a sequence. It is ended by the keyword `and`. Thinking in terms of timing diagrams, every such expression describes a signal value at a certain time point in the waveform.

In order to describe relationships between signal values at *different* time points you can either use the `$past()` system function or the TiDAL function `set_freeze()`. With it you can create (“freeze”) a fixed reference to a value of an expression at a particular time point and store it in a “freeze variable”. A freeze variable has the same value throughout all time points of a property. The syntax is `set_freeze(temp_variable, expression)`, where “temp_variable”

is a local variable like the ones defined in lines 18 to 21. The **expression** can be any valid SVA expression. For example, in line 23 we freeze the value of a slice of the signal `instrIn` at time point $t + 0$ and store it in variable `rs1`. (It resembles the opcode field “source register 1” of the instruction word input to the processor.)

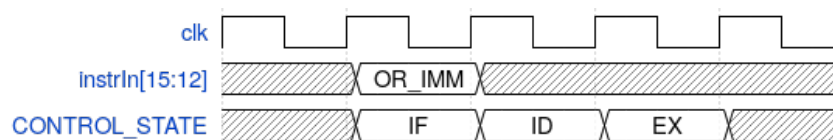
Remember: Temporal statements must be separated by the keyword **and**.

Line 31: The keyword **implies** separates the cause part and the effect part of the property.

Line 28: A temporal statement in the cause part of the property (i.e., the antecedent of the implication) is called an **assumption**. Line 28 specifies that we assume that at time point $t + 0$ the signal `CONTROL_STATE` has the value `c_IF`.

Line 33: A temporal statement in the effect part of the property (i.e., the consequent of the implication) is called a **commitment**, something that has to be proven for all scenarios where the assumption is true. Line 33 specifies that time point $t + 2$ we want to prove that the signal `CONTROL_STATE` has value `c_EX`.

The above code describes the situation displayed in the following timing diagram.



Notes

Working directory **processor** contains the following files:

proc-package.vhd some type and constant declarations (use these in your properties!)

proc.vhd contains the **entity**

proc-seq.vhd contains the **architecture**

proc.tda a start-up file for your project

Task 5.1

Verify the instructions `OR_IMM`, `OR_REG`, `ADD_IMM` and `ADD_REG` using TiDAL. Set up a property for each instruction, verifying a complete execution cycle. What execution phase should be specified for the *assumption* part of the property, i.e., the part before the **implies** statement? What execution phase should be specified for the last time point of the *commitment* part of the property, i.e., the part after the **implies** statement?

In order to verify a particular instruction you have to make assumptions about certain fields in the instruction word as shown in the ISA specification above. (The instruction is fed into the processor through the input `instrIn`, see above.) Show that after the execution of the instruction, the correct result of the ALU operation is in the destination register. Make sure your property handles the special register `R0` as well as correct sign extension of the *immediate* operands.

Some SVA tips:

- In SVA, the operator for the bitwise OR is “|”, and the arithmetic addition operator is “+”.

- Bit vectors can be concatenated by placing them in curly braces { }, and separating them by commas. Example:

`{1'b1, 1'b1, 1'b0, 1'b0}` is the same as `4'b1100`.

For shorter notation, you can specify the repetition of (sub-)vectors. For example,

`{ {3{2'b10}}, 2'b01 }` is the same as `3'b10101001`.

Note that for the repetition operator, `{n{vec}}`, the enclosing parentheses cannot be omitted.

- The following conditional statement allows to formulate an “if-then-else” structure:

`boolean_condition? expr1: expr2`

The value of this expression is `expr1` when `boolean_condition` is true, otherwise it is `expr2`.


Task 5.2

Verify the LOAD and STORE instructions using TiDAL:

1. For LOAD, show that the memory contents end up in the destination register.
2. For STORE, show that right data is sent to memory correctly at the right time point.

For both instructions, make sure to verify that the correct memory address is generated and the writeEnable signal is properly set during the whole instruction execution cycle.

Task 5.3

Verify the control flow instructions (JUMP, BRANCH). How many clock cycles does the execution of these instructions take? For each instruction, show that after execution, the next instruction is fetched according to the ISA specification. 

Note: The properties may not hold if you use an expression like `PC == prev_PC + 2`. (Note the warnings you may see in this case when reading in your SVA file.) Instead, if you write your expression like this: `PC == prev_PC + 16'd2`, it may work. Why? 