Norwegian University of Science and Technology
Department of Electronics and Telecommunication

TFE4171
Design of Digital Systems 2
Spring 2023

Lab 3

Delivery time: Sunday, March 19, 23:59 h.

General information

This lab and the following one will be run using the software OneSpin 360 DV, which is a formal property checker.

The tool is available on the server **venus.tele.ntnu.no** for remote connection. To use it on your own computer/laptop, you need to run an X-server software (e.g., MobaXterm) and connect to **venus.tele.ntnu.no** using secure-shell with X tunneling (**ssh -X venus.tele.ntnu.no**). You can start the tool on the remote machine by typing **onespin** into the command line. The tool connects to the X server on your local machine and opens the graphical user interface.

Before you can start working on the assignments, you need to copy the project files into your home directory. On **venus**, type or copy/paste:

```
cp -r /home/courses/desdigsys2/2023s/dd2master/projects .
```

The copied directory **projects** holds a number of subdirectories containing the project files for the individual lab tasks, namely: **dff** (for "D-flipflop"), **jkff** (for "JK-flipflop"), **atm**, **busarbiter**, **processor** and **readserial**. At the beginning of each lab description, there is a note indicating what project subdirectory is to be used as working directory.

Lab 3.1

[Working directory: **dff**]

Task 3.1.1

Using the handout material, familiarize yourself with the general workflow of formal property checking with **onespin**. Understand how to enter the RTL code for the design and how to run elaborate and compile. Learn how to write and check properties.

- Start **onespin**, change your working directory to **dff** and load the SystemVerilog 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.
- Question 3.1.1: What outputs does the tool produce?
- Analyze any counterexample that is found by the tool.

  Question 3.1.2: Where is the bug?

Lab 3.2

[Working directory: **jkff**]

Task 3.2.1

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.

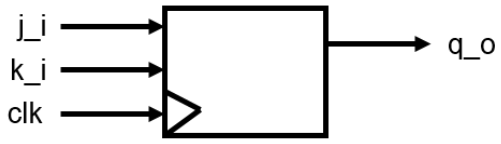


Figure 1: JK-flipflop

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.

Question 3.2.1: Is the set of properties *complete* in the sense that it covers all possible input/output behaviors of the design? If not, which behavior is missing?

Lab 3.3

[Working directory: **atm**]

> **Note:** When reading-in the SVA file (in MV mode), make sure to select SVA standard 2009 or later. Otherwise, the SVA keyword **implies** will not be recognized by OneSpin.

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

Given: the SystemVerilog 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.sv):

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

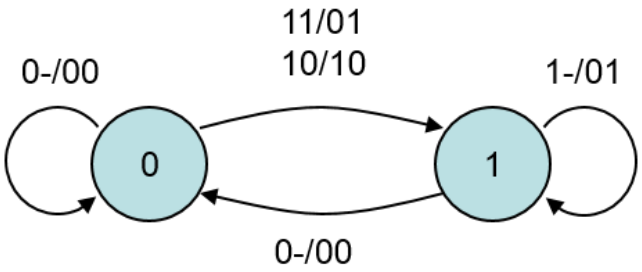


Figure 2: State transition graph of the ATM packet monitor

After reset, the design is in state 0.

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.

Question 3.3.1: Is this a Mealy or a Moore design?

Task 3.3.1

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**.

Question 3.3.2: How does **implies** differ from the temporal implications |->, |=>?

Question 3.3.3: Which of the above five specification items are not fulfilled by the design?

Question 3.3.4: Can you successfully verify all five specification items for the design? Explain your answer. In case your answer is no, describe possible remedies.

Lab 3.4

[Working directory: **busarbiter**]

In this task you will prove the correctness of the implementation of an *arbiter* in a bus system.

In order to get started, change the working directory in OneSpin to **busarbiter**. Open the "Read Verilog" dialog. **First, select and read-in in the package file busarbiter-package.sv.** Then read-in the remaining SystemVerilog source files.

Design Specification

Fig. 3 shows the bus system with three masters, one slave and the arbiter to be verified. Masters may request bus access using their individual request signal. Only one master at a time is granted access by the arbiter. The bus transaction begins with the grant signal. The master deasserts its request signal after receiving the grant signal. The slave signals completion of the transaction by asserting its **ack** signal for the duration of one clock cycle. (After assertion, the **ack** signal goes low for at least one clock cycle.) The grant signal stays stable during the whole transaction. Another transaction can begin in the immediate clock cycle after the **ack** signal was asserted.

The arbiter follows a *static priority* arbitration policy. Master 0 has the highest priority, Master 2 the lowest. Whenever more than one master requests bus access, the one with the highest priority wins arbitration.

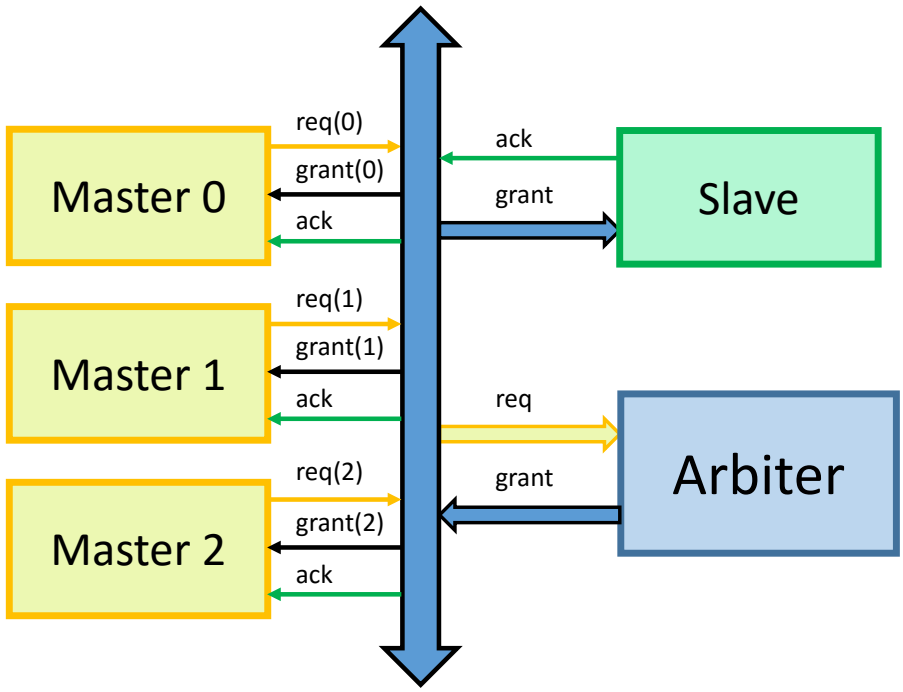


Figure 3: Bus system with three masters, one slave and an arbiter

Arbiter input signals:

clk:	**std_logic**	– clock signal
reset:	**std_logic**	– reset signal (active high)
bus_req:	**std_logic_vector(2 downto 0)**	– requests by masters
ack:	**std_logic**	– from slave, signaling completion of transaction

Arbiter output signals:
bus_grant: **std_logic_vector(2 downto 0)** – grant signals, one for each master.

Task 3.4.1

Write a property **p_reset** that verifies the reset behavior of the arbiter.

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

Question 3.4.1: What are the outputs of the circuit after the reset?

Task 3.4.2

Write a property **p_at_most_one_grant** that proves that at any point in time, not more than one master is being granted access to the bus.

Question 3.4.2: Can you use the built-in SVA function **$onehot()** in this property?

Question 3.4.3: What is the difference between the two SVA built-in functions **$onehot()** and **$onehot0()**?

Task 3.4.3

As stated above, a bus transaction begins with the assertion of a grant signal. Write a property **p_grant_stable** that proves that during a transaction the grant signal remains stable.

(Hint: Do not use SVA constructs other than **$rose(), $fell(), $stable() and implies**.)

Task 3.4.4

Write a property **p_arbitration_master0** that verifies correct arbitration when Master 0 requests and wins arbitration. In the same way, write two properties **p_arbitration_master1** and **p_arbitration_master2** verifying correct arbitration when Masters 1 and 2 request and win arbitration, respectively.

Task 3.4.5

Write a property **p_grant_master1** proving that if Master 1 is being granted then Master 1 has requested and Master 0 has not requested.

(Hint: Use the SVA construct **implies** to formulate an implication between two sequences.)

Analogously, write a property **p_grant_master2** verifying the correct preconditions for Master 2 being granted access.

Question 3.4.4: In this property, we check that whenever something is true in the present then something else must have been true in the past. This way of "looking backwards in time" is no problem with a formal verification tool. However, is it possible to verify such a property also with a simulation-based verification tool that can, obviously, simulate only forward in time? Explain your answer.

Question 3.4.5: Is it possible to relate two sequences A and B with A **implies** B, if sequence B begins at an *earlier* timepoint than sequence A? Explain your answer, and describe a workaround if your answer is "No".

Lab 3.5

[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 or rs2
ADD_IMM	rd := rs1 + sign_ext(Immediate)
OR_IMM	rd := rs1 or sign_ext(Immediate)
LOAD	rd := MEM[rs1 + sign_ext(Immediate)]
STORE	MEM[rs1 + sign_ext(Immediate)] := rs2
JUMP	PC := PC + 2 + sign_ext(Offset)
BRANCH	if rs1 = 0 then PC := PC + 2 + sign_ext(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

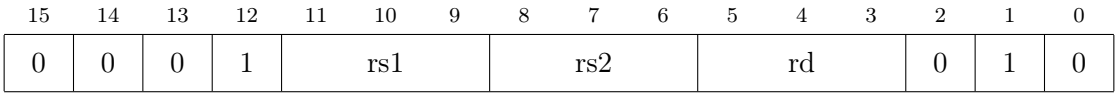
adds the contents of two source registers and stores the result in the destination register.

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

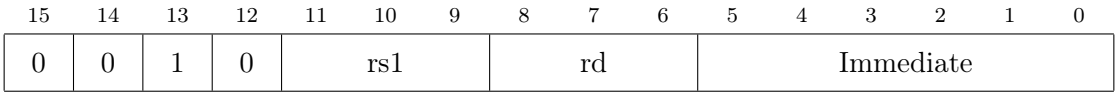
computes the bitwise logical OR of the contents of two source registers and stores the result in the destination register.



Transfer: rd := rs1 or rs2

ADD_IMM

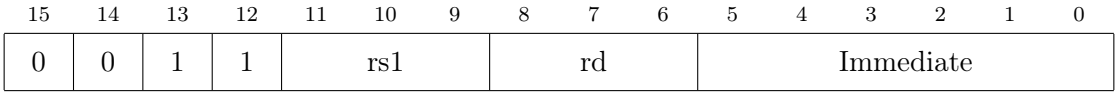
adds an immediate 6-bit value (specified in the instruction code itself) to the contents of a source register and stores the result in the destination register. Before being used, the immediate 6-bit value is sign-extended to 8 bits.



Transfer: rd := rs1 + sign_ext(Immediate)

OR_IMM

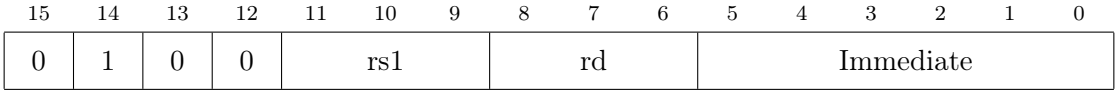
computes the bitwise logical OR of the contents of a source register and an immediate 6-bit value (specified in the instruction code itself) and stores the result in the destination register. Before being used, the immediate 6-bit value is sign-extended to 8 bits.



Transfer: rd := rs1 or sign_ext(Immediate)

LOAD

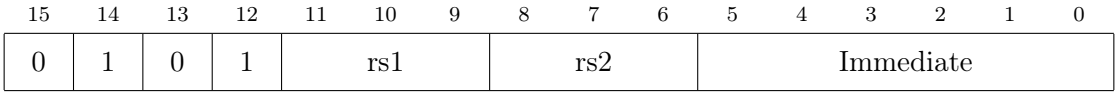
loads one byte from memory and stores it in the destination register. The memory address is computed by adding an immediate 6-bit value (specified in the instruction code itself) to the contents of a source register. Before being used, the immediate 6-bit value is sign-extended to 8 bits.



Transfer: rd := MEM[rs1 + sign_ext(Immediate)]

STORE

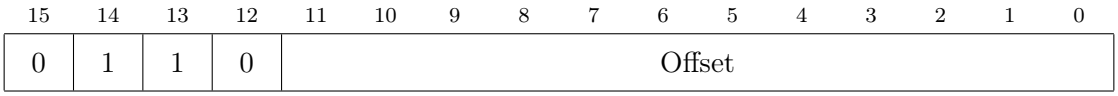
stores the contents of the second source register in memory. The memory address is computed by adding an immediate 6-bit value (specified in the instruction code itself) to the contents of the first source register. Before being used, the immediate 6-bit value is sign-extended to 8 bits.



Transfer: MEM[rs1 + sign_ext(Immediate)] := rs2

JUMP

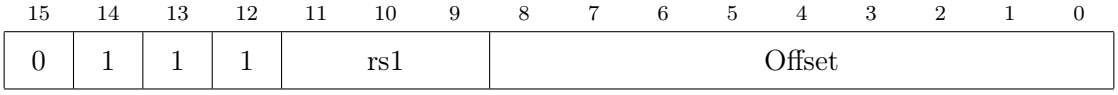
performs an unconditional branch. The destination address is computed by adding the constant 2 and the offset (specified in the instruction code itself) to the program counter (PC). Before being used, the 12-bit offset is sign-extended to 16 bits.



Transfer: PC := PC + 2 + sign_ext(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.



Transfer: if rs1 = 0
then PC := PC + 2 + sign_ext(Offset)
else PC := PC + 2

External Bus Interface

The external bus consists of the following signals:

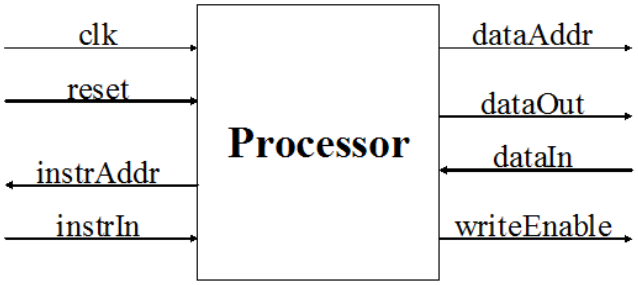


Figure 4: Processor interface

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 STORE instructions (output, during MEM phase).

dataIn

The data read from memory in LOAD instructions (input, 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 SystemVerilog code.) Instructions are allowed to use R0 as the destination register. For example, an ADDI instruction writing to R0 has no effect and can serve as a NOP.

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 quickly become complicated. In the following we use a SystemVerilog extension library called *TiDAL* (Timing Diagram Assertion Library) 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 the library is based on standard SystemVerilog it can also be used with other tools supporting SystemVerilog and SVA (e.g., simulation-based verification tools).

Structure of TiDAL properties

In this section we will explain the syntax and semantics of TiDAL based on an example. The additional TiDAL keywords used in our experiments are only a few: "`t`" for referencing time points, and "`set_freeze()`" for storing signal values at certain time points for future reference.

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*.

The following example explains the syntax and general structure of an operation property written using the TiDAL extensions.

```
 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, REG_FILE[rs1]) 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(.*);
```

The general organization of a checker module definition (formal arguments, parameters, clocking and disable conditions, etc.) is the same as for plain SVA. The differences are explained in the following.

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

Lines 15, 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.

There are two ways to relate signal values at *different* time points in a logic expression. You can either use the standard SVA system function `$past()` which refers to time points *relative* to the current one (given by `t ##N`), by specifying an offset. Or you can use 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" (see above). A freeze variable has the same value throughout all time points of a property (including the time points prior to the one where it was defined). 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.)

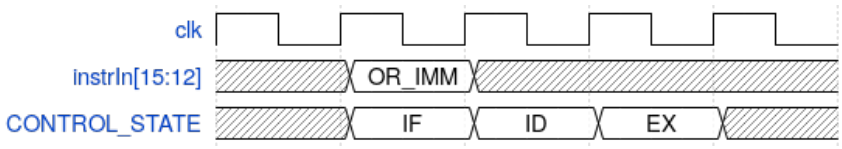
Remember: Temporal statements must be separated by the keyword `and`.

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.sv some type and constant declarations (use these in your properties!)

proc-seq.sv contains the **processor design**

proc.tda a start-up file for your project

Task 3.5.1

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.

Question 3.5.1: What execution phase should be specified for the *assumption* part of the property, i.e., the part before the `implies` statement?

Question 3.5.2: 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 `8'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 3.5.2

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 3.5.3

Verify the control flow instructions (JUMP, BRANCH). 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, a if you write your expression like this: `PC == prev_PC + 16'd2`, it may work.

Question 3.5.3: How many clock cycles does the execution of the JUMP instruction take?

Question 3.5.4: How many clock cycles does the execution of the BRANCH instruction take?

Question 3.5.5: Concerning the note above: What is the difference between the expressions: `PC == prev_PC + 2` and `PC == prev_PC + 16'd2` ?