NTNU – Trondheim
Norwegian University of Science and Technology

DEPARTMENT OF ELECTRONIC SYSTEMS

EXAM IN COURSE TFE4171 DESIGN OF DIGITAL SYSTEMS II

Contact: Kjetil Svarstad
Tel.: 458 54 333
Examination date: May 15, 2017
Examination time (from - to): 0900-1300

Permitted support material: C–Specified printed and hand-written support material is allowed. A specific basic calculator is allowed.

Other information: **Maximum number of points** per task and sub-task are given in the text.
Maximum number of points totally: **60**.

The **final grade** is calculated by summing your points from this exam with the exercises scaled to 20 points and the optional term project (if delivered and result contributes positively). Sum is then scaled to 100.

NB: This exam must be **passed** to pass in total. It is not sufficient that the total grade is a pass grade (E or better), the grade on the exam itself must also be E or better.

Language: English
Number of enumerated pages: 13
Additional pages in enclosures: 0

Controlled by:

Dato Sign

Intentionally left blank

Problem 1 Multiple choice (18 points)

Answer by circling the answer alternative you believe is the correct answer. You are awarded 3 points for a correct answer and 0 points if you do not answer. If your answer is wrong or you circle more than one alternative, you will get -1.5 point. If you need to change your answer, cross out your circled number and circle the right one.

a) (3 p) Which statement is NOT true

1. Cover statements can not contain a FAIL action
2. Cover statements can not contain a reset condition
3. Both `cover property` and `cover sequence` are legal statements.

Correct answer: Alternative 2

```
assign sig1 = 1;
assign sig2 = 1;
always_comb a1: assert #0 (sig1 == sig2);
```

b) (3 p) Consider the above SVA code snippet. Assuming `sig1` and `sig2` are initially zero, which of the following is a possible result?

1. Depending on execution order, the assertion `a1` may fail due to a simulation race.
2. The assertion `a1` will vacuously pass.
3. The assertion `a1` will never fail.
4. Depending on execution order, the assertion `a1` may fail due to a glitch.

Correct answer: Alternative 3, it's a deferred observed assertion, which is unsensitive to active region glitches

```
assert property (@(posedge clk) always s_eventually p);
```

c) (3 p) The assertion above means that the signal p must be:

1. asserted at every clock tick in the future
2. asserted at least once in the future
3. asserted at the current time and at least once in the future
4. asserted infinitely often in the future

Correct answer: Alternative 4, at every time point in the future, p must be asserted some time in the future. That means for for the strong assertion it must be infinitely often asserted, but not necessarily at every clock tick. Relevant for fairness and liveness for example.

d) (3 p) Which of the following statements is true for a given Kripke model:

1. CTL formulas can be associated with a set of paths in which they are valid.
2. CTL formulas can be associated with a set of states in which they are valid.
3. CTL formulas can be associated with a set of inputs for which the formula is valid.
4. CTL formulas can be associated with a set of outputs that fulfil the formula.
5. CTL formulas can always be associated with a set of state transitions in a Kripke model.

Correct answer: Alternative 2

e) (3 p) Let a, b, c be CTL formulas. Which of the following is NOT a CTL formula:

1. $a \vee \mathrm{EX}a$
2. $a \vee b$
3. true
4. $\mathrm{E}(a \mathrm{~U~} (\mathrm{AF}b))$
5. $\mathrm{AF}(a\mathrm{X}b)$

Correct answer: Alternative 5, very temporal operator must come together with a quantifier, this is not the case for "X" here

f) (3 p) Which of the following statements is true:
CTL formulas can be evaluated by

1. computing the set of reachable states starting from the initial state
2. using the fixed point characterizations of the CTL operators to iteratively compute the state sets associated with the formula
3. using a SAT-solver to jump out of local fixed points
4. ruling out false negatives obtained in the reachable state set

Correct answer: Alternative 2

Problem 2 SystemVerilog Assertions (12 points)

a) (3p) Describe in short at least 3 new features in System Verilog that is not a part of standard Verilog.

LF:

- Assertions (SVA) for dynamic (during simulation) verification
- Interfaces (standalone de-coupled definitions of such)
- Programs (reactive, for testbeds and other non-hw description)

b) (4p) Explain the problem of *vacuity/vacuous success/vacuous pass*. Use the following property as an example. Discuss how this problem can be solved by the verification engineer by changing the property description. Mention also how some tools remedy this problem.

```
property pr_r_q;
  @(posedge clk) req |-> ##2 gnt;
endproperty;

assert property (pr_r_q) $display($stime,,,"%m PASS");
  else $display($stime,,,"%m FAIL");
```

LF:

- Since an implication s |-> p (or |=>) is a logic function equal to (not s or p), this means that the implication itself is logically true if s and p are both true, or whenever s is not true. The last case, s is false, is vacuous pass/success since it does not signify any useful information. For the above assertion it means that the PASS condition will be executed whenever req is not asserted, which is of course not very interesting, however, it is not “wrong”.
- Can be handled by excluding the PASS condition block, and only report failure which is the interesting condition here. Also, the “followed-by” (#-#) could be used since that is non-vacuous. The assertion could be rewritten in other ways also.
- Simulators will typically have an option (or even default to) not reporting vacuous success in assertions based on implications.

c) (2p) Show a timing diagram or signal trace that matches the following sequence:

```
(a ##2 b) ##0 (c ##1 d)
```

LF:
a must be asserted from the start (0), while b must be asserted from 2. c must be asserted simultaneously with b (`##0` means overlap of end and start) and d from 3 (X here is don't-care):

```
a 111111....
b XX1111....
c XX1111....
d XXX111....
```

d) (3p) Show by timing diagram or signal trace matching sequences for the two properties p1 and p2. Explain why they are different or equivalent.

```
p1: assert property (@(posedge clk) a |-> ##1 b);
p2: assert property (@(posedge clk) a |=> ##2 b);
```

LF:
p1 and p2 are not equivalent. p1 is overlapping implication, and since there is a delay of 1 cycle before b, that means that if a is asserted, b must be asserted in the next cycle. p2 is non-overlapping and b is delayed 2 extra cycles which means that whenever a is asserted, b must be asserted 3 cycles later.

```
p1:
a 0010000...
b XXX1XXXX...
p2:
a 0010000...
b XXXXX1XXXX...
```

Problem 3 (6 points)

For the Kripke state diagrams in Figure 1:

a) (3p) Mark the states in which the properties hold.

LF:
See 1.

b) (3p) State whether or not the property holds for the system, explain your answer

LF:
Left: property holds since reset state is element of marked state set
Right: property does not hold since reset state is not marked

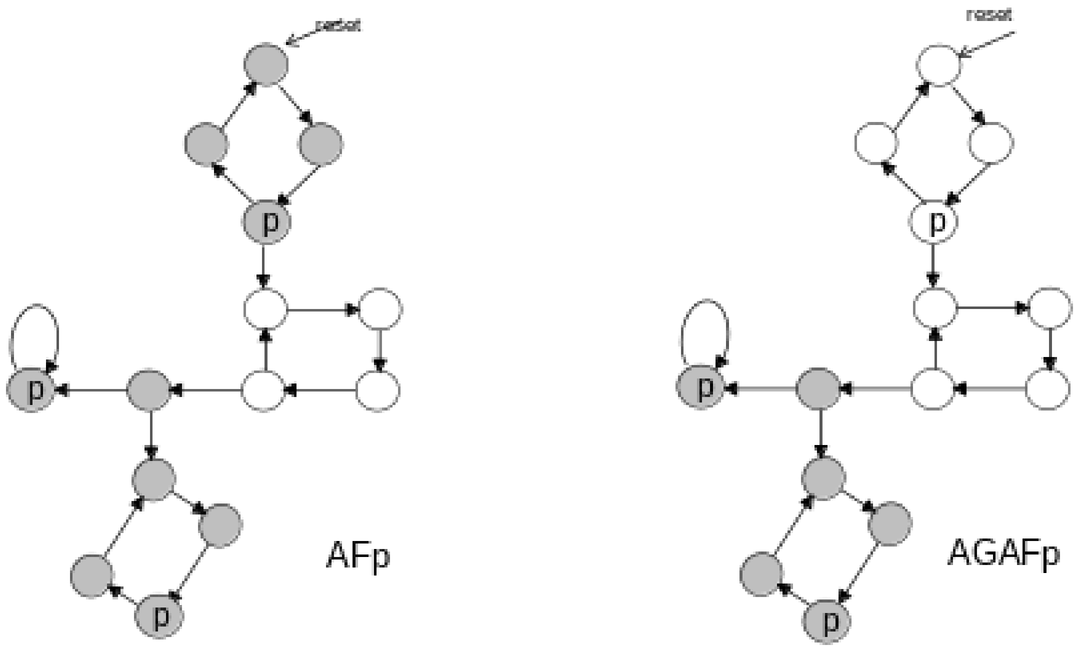

FIGURE 1 – Kripke state diagrams (marked solution)

Problem 4 **(12 points)**

Consider the design in Figure 2, consisting of two finite state machines. One is a counter. After reset, it counts from 0 to 5 with every clock tick. Once x=5, the value does not change any more until the next reset. The other component implements the finite state machine of Moore type shown in Figure 3

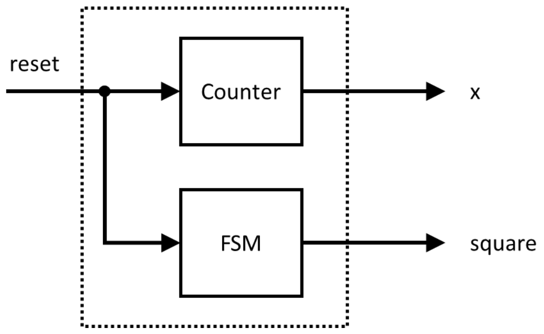

FIGURE 2 – System of 2 FSM's

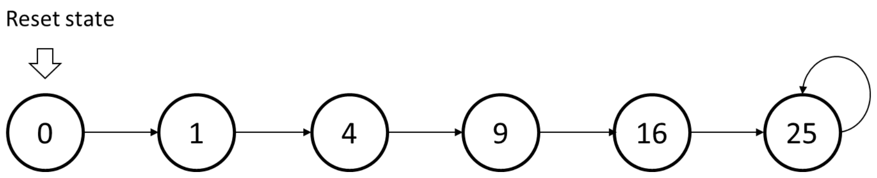

FIGURE 3 – Moore FSM

The numbers shown in the nodes are both, the state of the FSM and its output in that state, name "square". For the design, obviously, after reset, at any point in time the output "square" represents the square of the output "x".

a) (3p) Write an SVA property that expresses that, at all times, square = x^2.

LF:

```
property sva_squares;
  !reset |-> square == x*x;
endproperty
```

b) (4p) For the sequence of natural square numbers it holds that the distance between two consecutive squares grows by two. For example, (4-1)=3, (9-4)=5, (16-9)=7, The following SVA property checks this relationship.

```
property sva_squares_difference;
  !reset ##1 !reset ##1 !reset |->
    (square - $past(square, 1) ==
    $past(square, 1)-$past(square, 2) + 2);
endproperty
```

An interval property checker returns the counterexample in Figure 4 to this property.

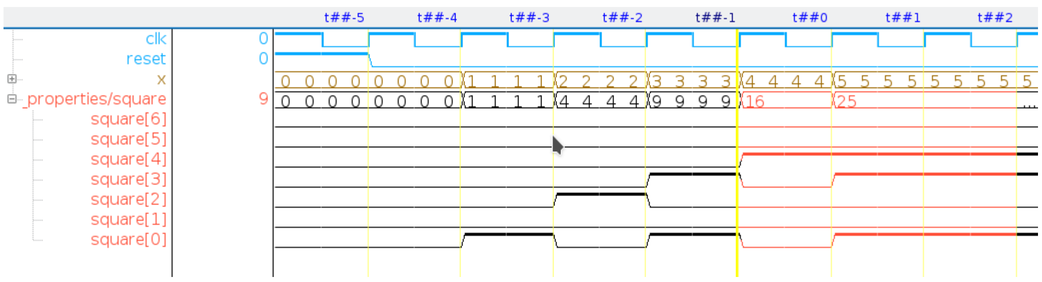

FIGURE 4 – Counterexample

- Is this a true or a false counterexample?
- How can the problem be fixed?

LF:
It is a true counterexample to the shown property. The property does not hold on the design because the design stays in state (x=5, square=25) once it reaches that state.

The problem can be fixed by correcting the property; for example, by restricting the range of the values of x and square for which the property is checked.

```
property sva_correct_squares_difference;
  !reset && (x <= 3) ##1 !reset ##1 !reset |->
  (square - $past(square, 1) ==
  $past(square,1)-$past(square, 2) + 2);
endproperty
```

c) (5p) The following operation property expresses that when the FSM begins in state 0, a sequence of numbers x and their squares will be produced by the design, as shown.

```
property square_sequence;
  t ##0 square == 0 implies
    t  ##0 x == 0 && square == 0 and
    t  ##1 x == 1 && square == 1 and
    t  ##2 x == 2 && square == 4 and
    t  ##3 x == 3 && square == 9 and
    t  ##4 x == 4 && square == 16 and
    t  ##5 x == 5 && square == 25;
endproperty
```

An interval property checker returns the counterexample in Figure 5 to this property.

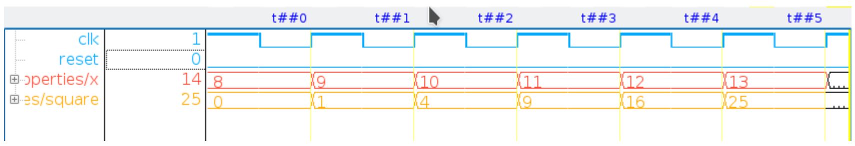

FIGURE 5 – Second counterexample

- Is this a true or false counterexample?
- How can the problem be fixed?

LF:
It is a false counterexample, caused by missing reachability information in the Interval Property Check. The design is correct: After reset, the counter and the FSM are synchronized so that square = x^2 at all times.

Reachability information must be added to the proofs in the form of invariants. A standard way is to formulate sequences that represent reachability information. For example:

```
sequence fsm_counter_sync;
  (!(square == 0)) || (x == 0);
endsequence

property square_sequence;
  t ##0 square == 0 and t ##0 fsm_counter_sync implies
  t ##0 x == 0 && square == 0 and
  t ##1 x == 1 && square == 1 and
  t ##2 x == 2 && square == 4 and
  t ##3 x == 3 && square == 9 and
  t ##4 x == 4 && square == 16 and
  t ##5 x == 5 && square == 25;
endproperty
```

(For the proof to be valid, the sequence must be verified in an independent assertion check.)

Another correct solution to the question is to modify the property such that it starts at the reset state. This way, reachability is explicitly contained in the property.

```
property square_sequence;
  t ##0 $past(reset)
implies
  t ##0 x == 0 && square == 0 and
  t ##1 x == 1 && square == 1 and
  t ##2 x == 2 && square == 4 and
  t ##3 x == 3 && square == 9 and
  t ##4 x == 4 && square == 16 and
  t ##5 x == 5 && square == 25;
endproperty
```

A third solution is to assume that the operation starts in a state where x=0 and square=0.

```
property square_sequence;
  t ##0 square == 0 && x == 0
implies
  t ##0 x == 0 && square == 0 and
  t ##1 x == 1 && square == 1 and
  t ##2 x == 2 && square == 4 and
  t ##3 x == 3 && square == 9 and
  t ##4 x == 4 && square == 16 and
  t ##5 x == 5 && square == 25;
endproperty
```

Problem 5 SystemC (12 points)

a) (2p) How many times will the `countme` method in the SystemC module code below be executed, and what will be the output? Explain your answer.

```
class count : sc_module {
  SC_CTOR(count) {
    c = 0;
    SC_METHOD(countme);
  }
  void countme () {
    c++;
    cout << c << " ";
  }
  int c;
};
int sc_main () {
  count count_inst;
  sc_start (100);
}
```

LF:
Only 1 time, at time 0. This is due to the lack of a sensitivity list which means it will never be run after initialisation.

b) (6p) Explain how the use of event modelling (use of `sc_event`), dynamic sensitivity lists, interface methods, and Transaction Level Modelling with “local time” can improve system modelling, HW/SW codesign, and simulation performance.

LF:

The use of `sc_event` means that it is possible to disconnect data from the simulation and instead use explicit events to model time control which can lead to both more abstract data modeling and better simulation performance.

Dynamic sensitivity lists enables the tuning of sensitivity of methods so that a computation that is in a state where only a limited set of events are interesting can limit the simulation to listen to just these events for a perticular period of time (or set of states). This enables higher level of abstraction in computation and also better simulation performance.

Interface methods abstract communication in the sense that methods act as a kind of abstract command bus. This allows more abstract communication modeling, less signalling and higher performance, and also an interface for lower level SW emulation drivers for early HW/SW validation.

TLM is putting all this into one concept and in addition allowing modules to compute on "slack time" in the sense that communicating modules run local time computation without involving the simualtor until a max point is reached and the simulator consolidates all modules. This means more consistent communication modeling with an abstract bus/channel system, SW emulation and validation, and possibly more that 1000 times higher simulation performance.

c) (4p) SystemC like many Hardware Description Languages uses Discrete Event Simulation as the semantics ("meaning") of the modelling code. Such a simulator can be either *preemptive* or *non-preemptive*. Which one is the case for SystemC, and explain what that means.

LF:

SystemC (like VHDL, Verilog and System Verilog) uses a preemptive scheduling simulation which means that all events (and that includes all types of signals, channels, etc) can only have one unique scheduled event in the future at any time. Thus if several events are scheduled for the same, only the closest in time will be kept, the others will be pre-empted—thrown away, not scheduled, and will not have any effect or being observed.