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 31, 2018
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: **100**.

The **final grade** is calculated by summing your points from this exam scaled to 60% with the exercises and the term project both scaled to 20% each.

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: 14
Additional pages in enclosures: 0

Controlled by:

Dato Sign

Informasjon om trykking av eksamensoppgave

Originalen er: **1-sidig** ☐ [✓ **2-sidig**] **sort/hvit** ☐ [✓ **farger**]

Skal ha flervalgskjema ☐
[✓ **MÅ IKKE NEDSKALERES**]

Intentionally left blank

Problem 1 Systemverilog assertions (25%)

a) (5%) Explain concurrent assertions and immediate assertions, how they differ, what they are used for, and the performance or efficiency difference between them.

Cerny 4.2–4.4. Immediate: simple, integrated in functional code, executed when reached, can only check conditions of current simulation time. Concurrent: Can describe and check complex behaviour in time (temporal conditions) and execute independently/in parallell/concurrently with the functional code. Immediate most used for checking invariant conditions, while concurrent can check complex temporal conditions and sequences over time. Concurrent assertions are much more performance demanding.

b) (5%) List and explain the simulation engine regions. Explain also how deferred assertions are handled in the regions.

Cerny 3.2–3.6. Deferred are delayed with respect to evaluation, they will be executed in the reactive (for deferred) or postponed (for final) so that all changes within the same time slot have settled meaning that they will only check the conditions at the end of the time slot. Non-deferred can react to glitches since they may be evaluated multiple times in a time slot if there are 0-delay updates.

c) (2%) What is the problem of shortcircuiting immediate assertions?

Since logical conditions such as "`if (f(a) && g(b))`" are lazily evaluated (meaning that if f(a) evaluates to 0, g(b) will not be evaluated at all since the result will be 0 anyway), if the logic functions contain assertions that might lead to the problem that any assertions in this case of g(b) will not be checked if f(a) returns 0. This is probably not what is intended, and is called short-circuiting (immediate) assertions and is not a good way to use assertions

d) (5%) Explain the repetition operator in sequences. In addition explain how goto repetition and non-consecutive repetition differs from standard repetition.

See Cerny 6.5, 6.7, 11.1.2–3

e) (3%) We have a simple req-ack protocol where req can be asserted when ack is deasserted, and then if ack is asserted within 5 steps, req should be deasserted and then followed by ack being deasserted. However, if the ack response has not been asserted after 5 steps, then an a_error is supposed to be asserted within the next 3 steps followed by deasserting req. Write a single sequence that captures this protocol, and show how it is asserted.

Solution:

```
!ack ##0 (req ##[1:5] ack ##1 !req ##1 !ack) or
         (req ##[6:9] a_error ##1 !req)
```

f) (5%) Explain what the following assertion means. Assume that command, read and write are signals. Mention also 3 other operators for combining sequences, and explain in short how they work.

```
double_end_a: assert property (@(posedge clk)
  command |-> write[->1] intersect read[->2])
```

After a `command` there must be one write asserted and two reads, and the write must be asserted at the same time as the second read (intersection means the sequences and at the same time)

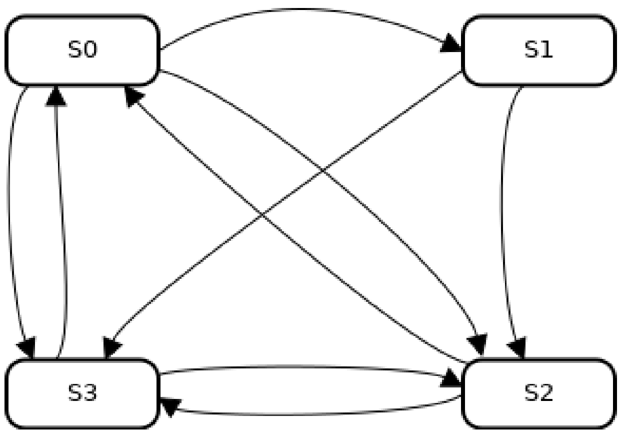
Problem 2 Systemverilog coverage (20%)

a) (5%) What is the purpose of constrained random testing? How is this done in Systemverilog? Show a simple example with a data structure of an add and subtract command and two 32 bit operands. How can we ensure that we test the corner and edge cases without necessarily simulating the massive set of all combinations?

b) (5%) What is coverage, and how can we use it to guide our testing and verification process? Show a simple coverage setup for the previous example. Explain the difference between property coverage and covergroups.

c) (5%) What is cross coverage, and how can it be used to help making sure that edge and corner cases are covered? Explain how is could be used for the previous example.

d) (5%) Below is the abstract definition of a state machine with 4 states. Imagine this was coded in Systemverilog with the states represented as an enumerated type with values S0, S1, S2 and S3. Show how we can use the state transition coverpoints to specifically find the coverage for specific paths in the simulation of the state machine. As an example show how to find coverage for all paths from S0 to S3 with a length (number of transitions) less than 5. And how could you find the coverage for the infinitely many paths 5 or more transitions long?



Problem 3 CTL Property Checking (20%)

a) (4%) Give a definition of the “Kripke model”.

(copied from lecture slides:)

Definition:

A Kripke model is a quintuple $K = (S, S_0, R, A, l)$ with

S:	a finite set of states
S_0:	a set of initial states with $S_0 \subseteq S$
R:	a transition relation, $R \subseteq S \times S$, describing all possible state transitions of the model
A:	a set of atomic formulas, $A = \{p, q, ...\}$ that can each assume the values *true* or *false*
l:	a valuation function, $l: A \rightarrow 2^S$. It specifies for every formula in A the set of all states in S for which the atomic formula is valid *(true)*

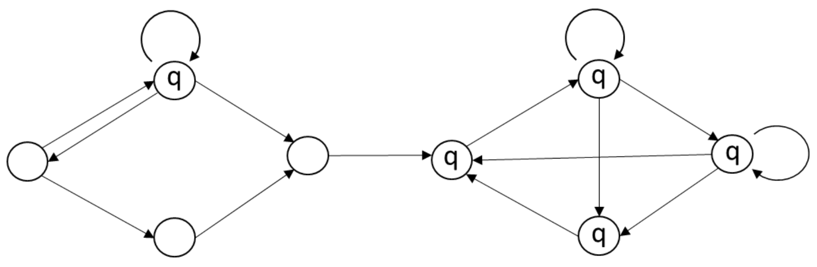
b) (4%) Provide the fixed point characterization of CTL formula $\mathsf{EG}\, p$.

$$y_{i+1} = p \wedge \mathsf{EX}\, y_i, \qquad y_0 = S = \mathit{true}$$

c) (4%) The CTL formulas $\neg p$, $p \wedge q$, $p \vee q$, $\mathsf{EX}\, p$, $\mathsf{E}(p\, \mathsf{U}\, Q)$, $\mathsf{EG}\, p$ and $\mathsf{EF}\, p$ can be used to express all other CTL formulas. Express $\mathsf{AG}\, p$ in terms of these formulas.

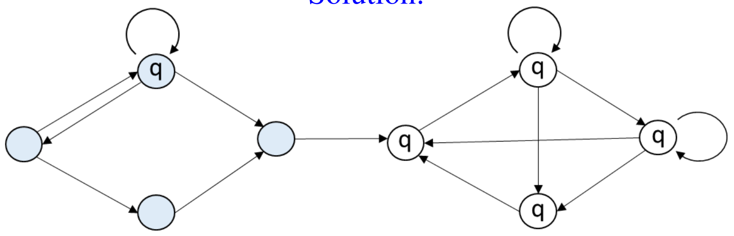
$$\mathsf{AG}\, p = \neg \mathsf{EF}\, \neg p$$

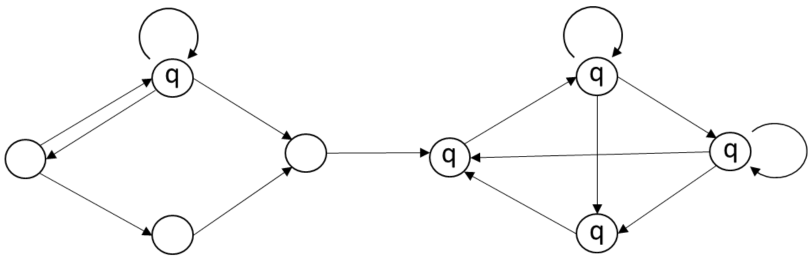
d) (4%) In the following Kripke model, mark the states which fulfill properties p_1 and p_2.



$$p_1 = \mathsf{EF}\, \neg q$$

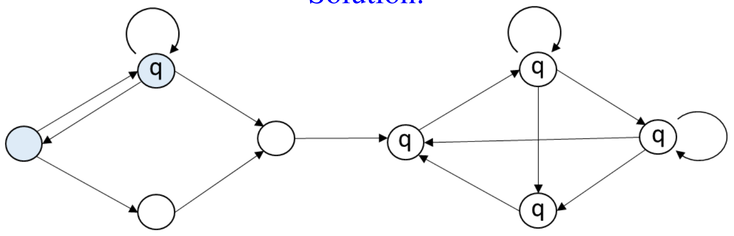
Solution:





$p_2 = \mathsf{EGEF}\, \neg q$

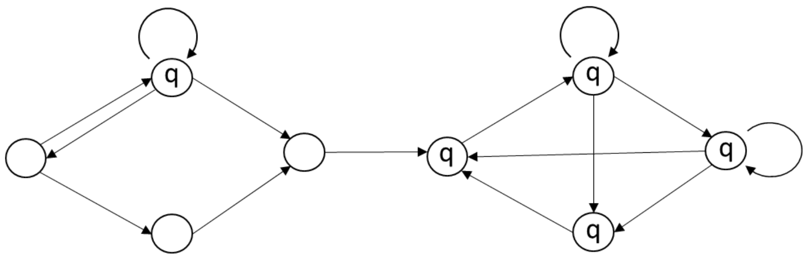
Solution:



e) (4%) Formulate a CTL property p_3 which checks that a property q is valid for all possible behaviors of the system with the possible exception of the behavior during a finite initialization period.

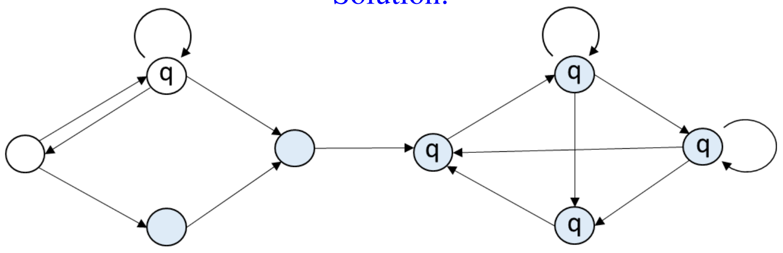
$p_3 = \mathsf{AFAG}\, q$

Mark the states which fulfil the property p_3.



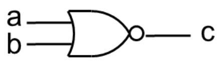
p_3

Solution:

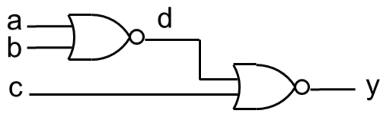


Problem 4 Satisfiability Problem (10%)

a) (5%) Provide a CNF for the following circuits using the Tseitin construction.

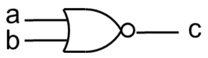


$\varphi_c =$



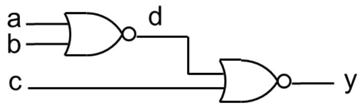
$\varphi_y =$

Solution:



$\varphi_c = \neg(c \oplus \neg(a + b))$ (since $c = \neg(a+b)$)

$= (a + b + c)(\neg a + \neg c)(\neg b + \neg c)$



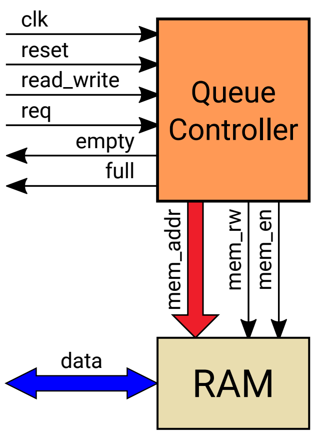
$\varphi_y = (a + b + d)(\neg a + \neg d)(\neg b + \neg d)(c + d + y)(\neg d + \neg y)(\neg c + \neg y)$

b) (5%) What advantage does this Tseitin CNF provide when compared to the classical CNF, as it can be derived from the truth table?

It is applicable to a multi-level gate netlist and grows linearly with the number of gates in the netlist. In contrast, the classical CNF derived from the truth table can grow exponentially with the number of gates in the multi-level circuit representation.

Problem 5 Interval property checking (20%)

In the following, we consider the design of a FIFO (first-in, first-out), based on a RAM and a queue controller.



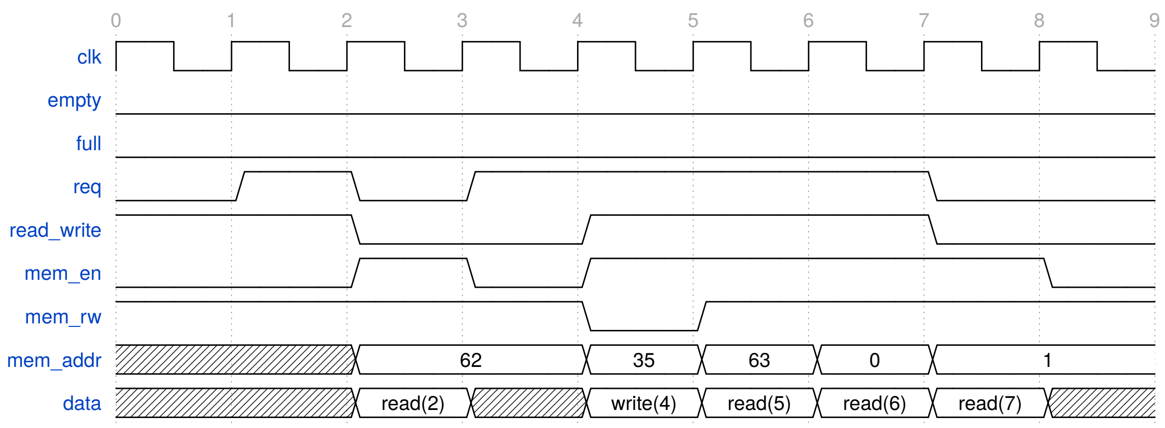
The controller generates addresses and two control signals for the RAM, one "enable" signal, **mem_en**, and one signal controlling the data direction: *read* when **mem_rw** is asserted, and *write* otherwise.

The design, internally maintains two address pointers, a *read pointer* and a *write pointer*. Depending on the transaction, the respective pointer is presented at the **mem_addr** output. The generated addresses range from 0 to $N - 1$, where N is the FIFO size, i.e., the maximum number of data elements that can be stored. In this design, $N = 64$.

The controller presents at its user interface two flags. Signal **empty** is asserted whenever there are no data stored in the FIFO. Signal **full** is asserted, when the number of data stored reaches its maximum, N. After reset, the FIFO is empty. The two flags can never be asserted at the same time.

The environment uses the FIFO by sending read or write requests to the queue controller. A request is indicated by asserting the **req** signal. The environment must read or write the data in the immediate clock cycle after the request. Transactions can be pipelined, i.e., a new request can be sent when the data for the current request is exchanged. A request to read from an empty FIFO is simply ignored, as is a request to write into a full FIFO.

The following timing diagram illustrates a few read and write transactions.



The environment sends a read request at $t = 2$. The controller generates the address of the datum at the head of the queue and enables the RAM. The environment reads the datum at $t = 3$ and then sends a new request, a write, at $t = 4$. It provides the data to be written at $t = 5$. At the same time it sends another request. Observe how, at $t = 6$, the read address wraps around at the boundary given by the FIFO size. Also note the different address values corresponding to the read and the write pointers in the respective transactions.

For the following problems, assume you are using a SAT-based Interval Property Checker.

a) (8%) Write an SVA property specifying the behavior after reset, assuming that the signal **reset** is active high. You do not need to specify anything about the values of **mem_addr**.

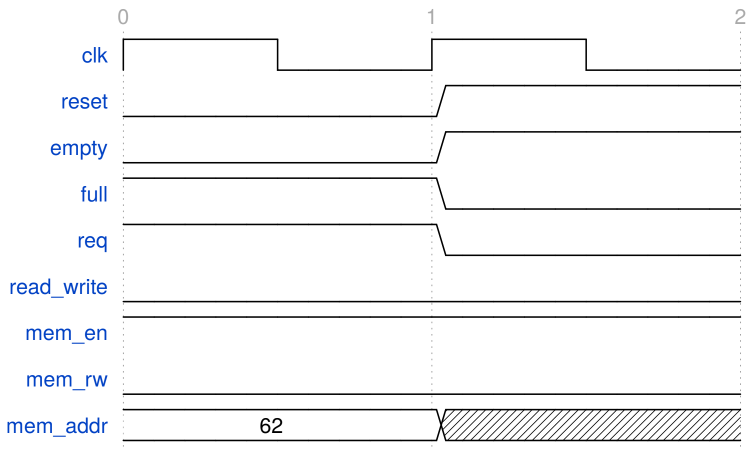
Without looking into the HDL code we don't know whether the design has an asynchronous or a synchronous reset. A property for either case can be considered a correct answer.
One possible answer, for a synchronous reset:

```
property p_reset;
   $fell(reset)
   |->
   empty && !full && !mem_en;
endproperty;

a_reset:  assert property (@(posedge clk) p_reset);
```

b) (6%) The following property considers the situation of a write transaction when the FIFO is full.

```
property write_when_full;
  !reset && full && req && !read_write |=>
  !mem_en && !mem_rw && full;
endproperty;

a_write_when_full:
  assert property (@(posedge clk) write_when_full);
```

The property checker returns the following counterexample to the property:



What is the problem?

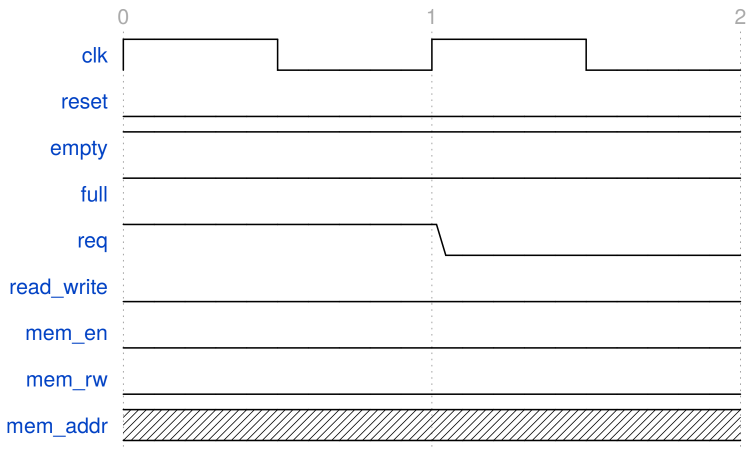
This is a true counterexample to the property which has a bug. It displays the situation when in the second clock cycle reset is asserted, as allowed by the property.
In order to rule out this example, the property should be "disabled" for reset situations, e.g., like this:

```
property write_when_full;
   !reset && full && req && !read_write |=>
   !mem_en && !mem_rw && full;
endproperty;

a_write_when_full:  assert property (@(posedge clk)
   disable iff (reset) write_when_full);
```

c) (6%) The following property considers the situation of a write transaction when the FIFO is empty.

```
property write_when_empty;
  !reset && empty && req && !read_write |=>
  mem_en && !mem_rw && !empty;
endproperty;

a_write_when_empty: assert property (@(posedge clk)
  disable iff (reset) write_when_empty);
```

The property checker returns the following counterexample to the property:



What is the problem?

There could be a bug in the design, of course. However, the shown counterexample could also be unreachable, as it seems to be the case here:
In the counterexample, both signals, **empty** and **full** are asserted. If the design is correct, then such a state can never be reached. (See design description.)

Problem 6 UVM (5%)

Explain in short the main parts in the UVM test environment, and point out how Systemverilog interfaces make this particularly powerful and efficient.