NTNU – Trondheim
Norwegian University of
Science and Technology

DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATIONS

EXAM IN COURSE TFE4171 DESIGN OF DIGITAL SYSTEMS II

Contact: Donn Morrison
Tel.: 455 48 895
Examination date: June 4, 2016
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: **50**.

The **final grade** is calculated by the sum of points from the exercises that count 40% and the exam results which count 60%.

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: 17
Additional pages in enclosures: 0

Controlled by:

_____

Dato Sign

Intentionally left blank

Problem 1 Multiple choice (20 points)

Answer by circling the answer alternative you believe is the correct answer. You are awarded 2 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 point.

a) (2 p) Which of the following statements is NOT true about the SystemVerilog simulation engine?

1. The *Active* region set handles events issued in the design code.
2. Concurrent assertions are evaluated in the *Observed* region.
3. The *Reactive* region set can schedule events to the *Active* region set.
4. Final assertions are executed in the *Preponed* region.

Correct answer: Alternative 4

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

b) (2 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 4

c) (2 p) Which of the following is NOT true regarding *deferred* assertions?

1. Deferred assertions must only be used when code has delay controls.
2. Deferred assertions are insensitive to simulation glitches.
3. Deferred assertions may be placed inside and outside procedural code.
4. Deferred assertion action blocks must contain at most one statement.

Correct answer: Alternative 1

d) (2 p) Which of the following assertions best matches the specification “a and b must not be active simultaneously”?

1. `assert property (@posedge clk) !($rose(a) && $rose(b));`
2. `assert property (@posedge clk) !($stable(a) && $stable(b));`
3. `assert property (@posedge clk) a |-> !b`
4. `assert property (@posedge clk) !(a && b);`

Correct answer: Alternative 4

```
class Bus;
    rand bit[15:0] addr;
    rand bit[31:0] data;
    constraint align {addr[3:0] == 4'b0;}
endclass

typedef enum {low, mid, high} AddrType;

class MyBus extends Bus;
    rand AddrType atype;
    constraint addr_range
    {
        (atype == low ) -> addr inside { [0 : 16383] };
        (atype == mid ) -> addr inside { [16384 : 32766] };
        (atype == high) -> addr inside { [32767 : 65534] };
    }
endclass
```

e) (2 p) Consider the above SystemVerilog code snippet. Upon instantiation of `MyBus`, which one of the following values can `addr` never take when *atype* == *mid*?

1. 0x4000
2. 0x4096
3. 0x6000
4. 0x7F00

Correct answer: Alternative 2

f) (2 p) The principle problem with the traditional hardware design cycle that SystemC and TLM address is:

1. Lack of formal equivalence checking between high- and RT-level models.

2. Late HW/SW partitioning can necessitate HW redesign.
3. Lack of early communication between hardware and software designers.
4. Slow cycle-accurate simulation.

Correct answer: Alternative 3

g) (2 p) Which of the following is NOT true regarding the execution of a SystemC application?

1. The *initialisation* phase includes a delta cycle.
2. The construction of the module hierarchy (e.g., modules created, ports bound to channels) takes place in the *elaboration* phase.
3. Delta notifications can be created during the *elaboration* phase.
4. The evaluate-update loop begins with a call to `sc_start()`.

Correct answer: Alternative 1

h) (2 p) The main objectives of timed TLM do NOT include which of the following?

1. Early software development on cycle-accurate models.
2. Benchmarking the performance of the micro-architecture based on timing annotations.
3. Fine-tuning the micro-architecture.
4. Optimising software for the micro-architecture to meet real-time constraints.

Correct answer: Alternative 1

i) (2 p) The difference between a SystemC *method* and *thread* is:

1. Only *methods* must be registered by the SystemC kernel.
2. A *thread* cannot have a return value.
3. A *method* can be invoked many times and cannot be suspended during execution.
4. A *thread* cannot have a sensitivity list, while a *method* can.

Correct answer: Alternative 3

j) (2 p) Which of the following statements relating to SystemC channels is true?

1. `sc_event_queue` is a primitive channel.
2. From a resource access perspective, `sc_semaphore res(2);` is equivalent to `sc_mutex res();`.
3. `sc_buffer` is a hierarchical channel.
4. None of the above.

Correct answer: Alternative 4

Problem 2 SystemVerilog Assertions (10 points)

a) (2p) Explain how an FPGA can be used to check design assertions. What parts of the assertion block are synthesisable? What is the benefit over software simulation?

Solution:

- Looking for a description of emulation of the assertions and the design running side-by-side. (1p)
- Action blocks are not synthesisable. The rest of the assertion is. (0.5p)
- Benefit is faster verification because the FPGA can run faster than the software simulator. (0.5p)
- More: Section 1.4.2

b) (2p) Consider the testbench below. Assume that the module `router` consumes request packets and issues acknowledgement packets in the next clock cycle upon receiving a request packet. The program `test` tests the correctness of the received packets by generating 100 request packets with random IDs and random data. Complete the testbench with two assertions:

a1 : Check that the received packet is an acknowledgement packet.

a2 : Check that the ID of the acknowledgement packet matches the ID of the request packet sent in the previous cycle.

```
typedef enum logic {REQ = 1'b1, ACK = 1'b0} dirType;

typedef struct packed
{
    dirType rq;
    logic [6:0] id;
    logic [23:0] data;
} packetType;

program test ( input logic clk, packetType received,
               output packetType sent);
logic [6:0] sent_id;
```

```
initial begin
    repeat (100) begin
    @( posedge clk);
    sent_id = $random;
    sent = '{REQ, sent_id, $random};
    @( posedge clk);
// Assertion a1 here:




// Assertion a2 here:




    end
end
endprogram : test

module router ( input packetType inpkt, logic clk,
                output packetType outpkt);
// ... Details ommitted
endmodule : router

module top;
    logic clk = 1'b0;
    initial repeat (400) #5 clk = !clk;
    packetType inpkt, outpkt;
    test t(.clk(clk), .received(outpkt), .sent(inpkt));
    router r(.*);
endmodule : top
```

Solution:

(1p) each:

```
a1: assert final (received.rq == ACK)
    else $error("Corrupted packet");
a2: assert final (received.id == sent_id)
    else $error("Lost packet");
```

More information page 41, Book SVA The Power of Assertions in SystemVerilog.).

c) (3p) Consider the simulation of the design below. Assume that the initialisation phase is complete and the simulator is beginning to process the Active region at time $t = 0$.

```
1  module procReq(input logic req, gnt, clk);
2      logic allow;
3      wire proceed;
4      assign proceed = allow && gnt;
5      always @(posedge clk) allow <= req;
6      always @(posedge proceed) processData();
7      a1: assert property(@(posedge clk) req |=> proceed || !gnt);
8  endmodule : procReq
9
10 program test(output logic request, grant, sync);
11     logic oldreq = 1'b0;
12     assign grant = oldreq;
13     initial begin
14         request = 1'b0;
15         sync = 1'b0;
16         for (int i = 0; i < 100; i++) begin
17             #5 sync <= !sync;
18             if (i % 2) begin
19                 oldreq <= request;
20                 request <= $random;
21             end
22         end
23     end
24 endprogram : test
25
26 module top();
27     logic r, g, c;
28     procReq dut(r, g, c);
29     test tb(r, g, c);
30 endmodule : top
```

Recall that the region order is Preponed →Active →Inactive →Non-blocking assignment (NBA) →Observed →Reactive →Re-reactive →Re-NBA →Postponed

What are the other necessary regions and their processing order at $t = 0$? How many times is Line 4 executed in this time step and in which region(s)? Explain in brief.

Solution:

- The regions are Active -> Reactive -> Active (1p)
- Line 4 is processed twice (proceed = X (Active), proceed = 0 (Active)) (2p)

Annotated example page 53 Book SVA The Power of Assertions in SystemVerilog.).

d) (3p) Complete the covergroup with two coverpoints - one for the counting modes and the other for the memory address. For the second coverpoint, create two address bins - the first counting accesses between addresses 0x00 and 0xFF and the second counting accesses to all other memory locations.

```
enum {INC, DEC, NO_CHANGE} count_modes;
bit [31:0] address;

covergroup cg_ahb;
// Answer goes here:









endgroup
```

Solution:

1p for correct coverpoints, 2p for correct bins, syntax not strict

```
enum {INC, DEC, NO_CHANGE} count_modes;
bit [31:0] address;

covergroup cg_ahb;
cp_modes : coverpoint count_modes;
cp_address : coverpoint address {
    bins no_access = {[0:255]};
```

```
    bins other[8] = {[256:$]};
}
endgroup
```

Further reading: Chapter 18, SVA Power of Assertions, pg 92-94, 425 + slides.

http://www.verilab.com/files/svug_2007_fall_func_cov_in_sv.pdf

Problem 3 Formal Verification (10 points)

Part 1: For the following assume that Interval Property Checking (IPC) is used to prove an operational property p spanning over an interval of length n. Decide for each of the following statements whether or not it is true. Give a short explanation.

a) (2p) The computational complexity of IPC generally increases with the length n of a property.

Solution: Yes, the number of time frames that need to be unrolled increases with n. Larger values of n therefore lead to larger SAT instances.

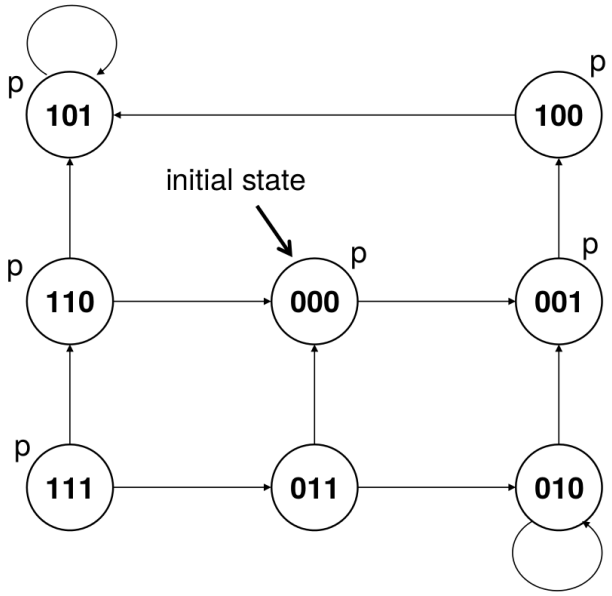
b) (2p) IPC may consider less design states than are actually reachable. This can lead to a spurious counter-example (false negative).

Solution: No, IPC considers more states than are actually reachable. Formal techniques never underestimate the search space. They may, however, over-approximate it.

c) (2p) To avoid spurious counter-examples (false negatives) it is possible to prove assertions by induction and to use them as invariants for IPC.

Solution: Yes, the base of the induction proves that the considered state set includes the initial state. The induction step proves that the state set is closed under reachability. Thus, the proven assertion represents a state set that is an invariant and includes the initial state.

Part 2: The figure depicts the state diagram of a finite state machine M with state vector s = (s1, s2, s3). The state diagram indicates in which states a property p is valid.



d) (2p) Consider the state set P = {000, 001, 100, 101, 110, 111}. Is P an invariant of this design? Explain.

Solution: P is not an invariant because it is not closed under reachability. ("Closed under reachability" means that all states that can be reached from P are already contained in P.) For example, state 111 reaches 011 which is not in P.

e) (2p) Determine the reachable state set R and an invariant W, W $\neq$ R, which is sufficient to prove that the design fulfills p.

Solution: $R = \{000, 001, 100, 101\}$. $W = \{110, 000, 001, 100, 101\}$. Obviously, W is closed under reachability, it is $W \supset R$ and p hold for all states in W. This is the only W that fulfills all these requirements and is therefore the only correct answer.

Problem 4 SystemC (10 points)

```
class timer : public sc_module
{
public:
  SC_HAS_PROCESS(timer);
  timer (sc_module_name name) : sc_module (name) {
    SC_THREAD (time_thread);
    SC_THREAD (print_thread);
  }
  void time_thread ();
  void print_thread ();
  sc_event ev;
};

void timer::time_thread () {
  sc_time tm (1,SC_NS);
  while (true) {
    for (int i = 1 ; i < 10; ++i) {
      ev.notify (tm);
      cout << "notify (" << tm + sc_time_stamp () << ")" << endl;
      tm = tm * 2;
    }
    wait ();
  }
}

void timer::print_thread () {
  while (true) {
    wait (ev);
    cout << "event @ " << sc_time_stamp () << endl;
  }
}

int sc_main (int argc , char *argv[])  {
  timer t ("timer");
  sc_start (1000, SC_NS);
  return 0;
}
```

a) (3p) Timer version 1. Study the above model and write the expected output from simulation. Explain the result.

Solution:

```
            SystemC 2.3.0-ASI --- Mar 10 2013 22:00:49
        Copyright (c) 1996-2012 by all Contributors,
        ALL RIGHTS RESERVED

notify (1 ns)
notify (2 ns)
notify (4 ns)
notify (8 ns)
notify (16 ns)
notify (32 ns)
notify (64 ns)
notify (128 ns)
notify (256 ns)
event @ 1 ns
```

b) (4p) The `time_thread` process from above has been modified and now produces the following simulation output (no other part of the above code has been changed). Your task is to recover this modification using only the simulation output. Write the full process body in the space below, copying what you need from the version above.

```
            SystemC 2.3.0-ASI --- Mar 10 2013 22:00:49
        Copyright (c) 1996-2012 by all Contributors,
        ALL RIGHTS RESERVED

notify (1 ns)
event @ 1 ns
notify (12 ns)
event @ 12 ns
notify (24 ns)
event @ 24 ns
notify (38 ns)
event @ 38 ns
notify (56 ns)
notify (82 ns)
event @ 56 ns
```

```
    notify (124 ns)
    notify (198 ns)
    notify (336 ns)
    event @ 124 ns


void timer::time_thread () {
// Your code here









} // time_thread
```

Solution:

```
void timer::time_thread () {
  sc_time tm (1,SC_NS);
  while (true) {
    for (int i = 1 ; i < 10; ++i) {
      ev.notify (tm);
      cout << "notify (" << tm + sc_time_stamp () << ")" << endl;
      wait (10, SC_NS);
      tm = tm * 2;
    }
    wait ();
  }
}
```

c) (3p) Explain the difference between static and dynamic processes. When is it desired to use dynamic processes in a SystemC model? How is this achieved?

Solution:

- Static processes are established during elaboration (SC_THREAD and SC_METHOD), and dynamic processes are spawned during simulation using `sc_spawn`. (1p)
- It is desirable to use dynamic processes in testbench scenarios to track transaction completion (e.g., performing temporal checks) or to spawn traffic generators dynamically. Without dynamic processes it would be necessary to pre-allocate a number of static processes accommodating the maximum number of possible outstanding requests. (1p)
- We must first define `#define SC_INCLUDE_DYNAMIC_PROCESSES`, define a spawnable process, and then call `sc_spawn` (1p):

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include <systemc>
...
void spawned_thread() {// This will be spawned
    cout << "INFO: spawned_thread "
        << sc_get_current_process_handle().name()
        << " @ " << sc_time_stamp() << endl;
    wait(10,SC_NS);
    cout << "INFO: Exiting" << endl;
}
void simple_spawn::main_thread() {
    wait(15,SC_NS);
    // Unused handle discarded
    sc_spawn(sc_bind(&spawned_thread));
    cout << "INFO: main_thread " << name()
        << " @ " << sc_time_stamp() << endl;
    wait(15,SC_NS);
    cout << "INFO: main_thread stopping "
        << " @ " << sc_time_stamp() << endl;
}
```

Further information, Chapter 7 in the textbook SystemC From the Ground Up (2nd Ed.).