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 2, 2015
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: 19
Additional pages in enclosures: 0

Controlled by:

Dato Sign

Intentionally left blank

Problem 1 Multiple choice (20 points)

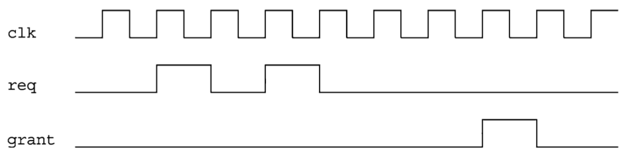
Answer by circling the answer alternative you believe is the correct answer. You are awarded 2 points for a correct answer and 0 points if you do not answer. If your answer is wrong or you circle more than one alternative, you will get -1 point.

a) (2 p) Ideally, verification is complete when:

1. Code and functional coverage reaches 100%.
2. Code coverage reaches 100%.
3. Functional coverage reaches 100%.
4. When the DUT passes 100% of the directed tests.

Correct answer: Alternative 1

b) (2 p) Choose the assertion that exactly matches the timing diagram:



1. `assert property(@(posedge clk) req |-> nexttime[3] grant);`
2. `assert property(@(posedge clk) req |-> nexttime[4] grant);`
3. `assert property(@(posedge clk) req |=> nexttime[4] grant);`
4. `assert property(@(posedge clk) req |-> nexttime[*4] grant);`

Correct answer: Alternative 2

c) (2 p) Which best describes the difference between `$rose` and `$posedge`?

1. `$posedge` returns an event, `$rose` returns a boolean
2. `$posedge` returns a boolean, `$rose` returns an event
3. `$posedge` is used for clocks, `$rose` is used for signals
4. `$posedge` is used for signals, `$rose` is used for clocks

Correct answer: Alternative 3

d) (2 p) In the SystemVerilog simulation engine, the *reactive region set* is responsible for:

1. Handling events from design code.
2. Executing statements from programs and checkers.
3. Sampling values used in concurrent assertions.
4. Finishing simulation tasks which do not include value changes or events.

Correct answer: Alternative 2

e) (2 p) Which of the following SVA snippets is equivalent to `|-> ##1`?

1. *## 0*
2. `|=>`
3. `|=> 1`
4. *## 1*

Correct answer: Alternative 2

f) (2 p) Which of the following is equivalent to the sequence `a ##1 b [*5] ##1 c`?

1. `a ##1 b [*1:$] ##1 c`
2. `a ##1 b [*5:$] ##1 c`
3. `a ##1 b ##1 b ##1 b ##1 b ##1 b ##1 c`
4. `a ##1 b ##1 c`

Correct answer: Alternative 3

g) (2 p) Which of the following properties is NOT true with respect to untimed TLM?

1. Bit-accurate behaviour and communication between modules.
2. Respect for dependences between processes using system synchronisation.
3. Sequential execution of independent processes.
4. Fast, clock-free simulation.

Correct answer: Alternative 3

h) (2 p) The following are SystemC primitive channels:

1. sc_semaphore, sc_mutex, sc_fifo

2. sc_semaphore, sc_event, sc_mutex
3. sc_signal, sc_semaphore, sc_mutex
4. sc_semaphore, sc_mutex, sc_event_queue

Correct answer: Alternative 1

i) (2 p) In SystemC, `notify()` and `wait()` are:

1. Used to start and stop the event simulation kernel.
2. Used to communicate and synchronise between processes.
3. Virtual functions that must be implemented in `SC_MODULE`.
4. None of the above.

Correct answer: Alternative 2

j) (2 p) In the SystemC simulation kernel, *elaboration* is:

1. The phase where class destructors are executed.
2. The phase where all simulation processes are invoked in unspecified deterministic order.
3. The phase where statements are executed after `sc_start()`.
4. The phase where statements are executed prior to `sc_start()`.

Correct answer: Alternative 4

Problem 2 SystemVerilog Assertions (10 points)

a) (4 p) What is the difference between *code coverage* and *functional coverage*? In the context of assertion-based verification, which is more important? Why?

Solution: "Code Coverage indicates the how much of RTL has been exercised. The Functional Coverage indicates which features or functions has been executed. Both of them are very important. With only Code Coverage, it may not present the real features coverage. On the other hand, the functional coverage may miss some unused RTL coverage."

Also "Coverage is used too check whether the testbench has exercised the design or not. Code coverage will give information about how many lines are executed, how many times expressions and branches are executed. This coverage is collected by the simulation tools. Users use this coverage to reach those corner cases which are not hit by the random testcases. Users have to write the directed testcases to reach the missing code coverage areas.

Functional coverage by the name itself is related to the functionality of the design and it is defined by the user. User will define the coverage points for the functions to be covered in DUT. This is completely under user control.

Both of them have equal importance in verification.100% functional coverage does not mean that the DUT is completely exercised and vice-versa. Verification engineers will consider both coverages to measure the verification progress."

b) (3 p) You are tasked with verifying a memory controller. Explain how you will apply constrained randomisation. Where is constrained randomisation useful? Where is it not useful?

Solution: Some examples:

- If the memory is smaller than the address space, we can constrain the address field to reduce test cases
- We can limit our tests to certain bit widths, for example 8-, 16-, or 32-bit
- For verification of write functionality, it is preferable to leave the data to be written unconstrained

Constrained randomisation is useful we want to exclude or constrain some set of values from the stimulus, where those values can be seen to be 1) not useful for verification or 2) cases which are incompatible with underlying protocols, packet types, etc. For example, a CRC field in a packet should be constrained to reflect the payload.

Constrained randomisation is not useful when we are looking for full input coverage, for example where we would instead use a directed test to iterate a small number of memory addresses or fully random stimulus to distribute access to a large address space.

c) (3 p) Write the assertion for: “when request `req` is issued and thereafter the first data chunk is received as identified by `data` bit asserted, acknowledgement `ack` should be sent.” Ensure assertion *re-use* by using named sequences and properties.

Solution:

```
a1: assert property(first_match(req ##[+] data) |-> ack);
```

or

```
a2: assert property(req ##1 data[->1] |-> ack);
```

with re-use:

```
sequence s;
    first_match(req ##[+] data);
endsequence: s

property p;
    s |-> ack;
endproperty : p

a1 assert property( p );
```

Problem 3 Formal Verification (10 points)

a) (4 p) Figure 3 shows the finite state machine (FSM) model of a bus unit. We would like to prove that the unit only enables a data transfer after it has requested the transfer from an arbiter (not shown) and has received an acknowledge. The following property is written:

```
Assume:
        at t+2: transfer = 1
        during[t, t+2]: reset = 0

Prove:
        at t: request = 1
        at t+1: acknowledge = 1
```

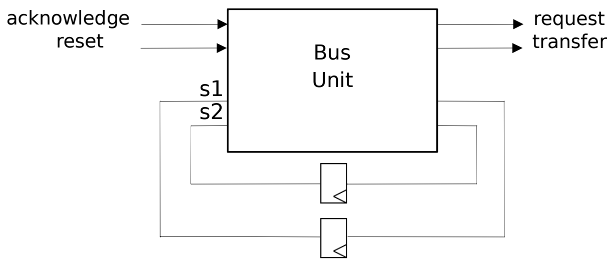


Figure 1: System under verification

Draw the block diagram of a model that can be used to prove this property by satisfiability solving.

The block diagram must show an appropriate unrolling of the FSM and the Boolean function which is checked for satisfiability. If the function you created is unsatisfiable, what does it mean for the validity of the considered property?

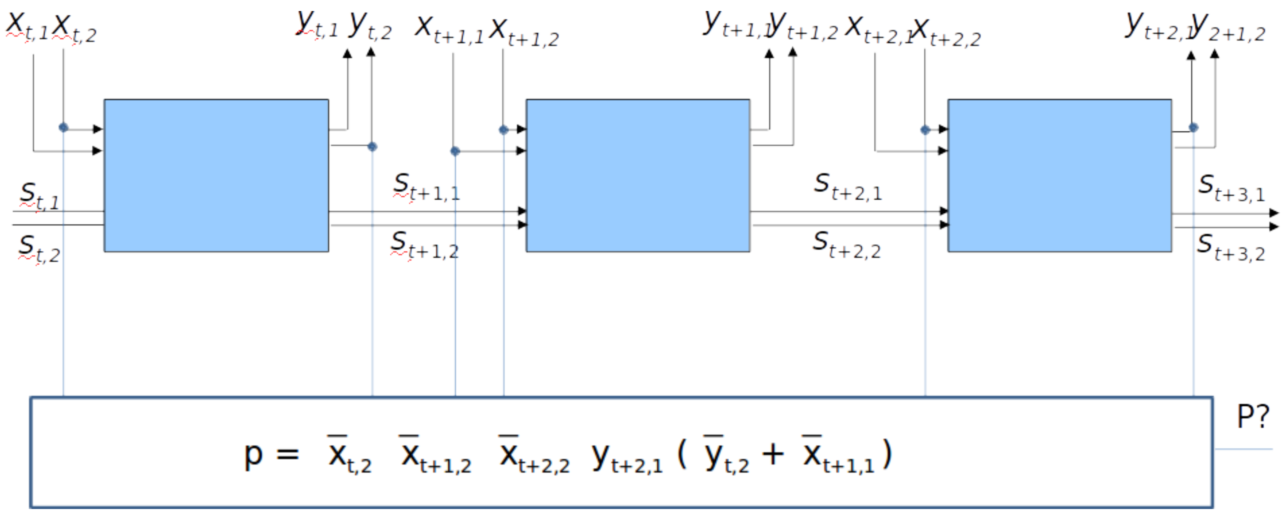
Solution: For readability of the block diagram let inputs and outputs be defined as

- x_1: acknowledge
- x_2: reset
- y_1: transfer
- y_2: request

The following Boolean function p is unsatisfiable if and only if the property holds.

$$p = \bar{x}_{t,2} \quad \bar{x}_{t+1,2} \quad \bar{x}_{t+2,2} \quad y_{t+2,1} \quad (\bar{y}_{t,2} + \bar{x}_{t+1,1})$$

The unrolled FSM with inserted function p is shown below.



b) (6 p) An Interval Property Checker is used to check the following three properties on a design represented by the FSM of Figure b with state vector s = (p, q, r, u). In the state diagram no inputs and outputs of the FSM are shown since they are not relevant for the following properties.

```
Property 1
Assume:
        at t: p = 1
Prove:
        at t+1: q = 1


Property 2
Assume:
        at t: p . q = 1
Prove:
        at t+1: q = 1


Property 3
Assume:
        at t: q = 1
Prove:
        at t+1: q = 1
or at t+2: q = 1
```

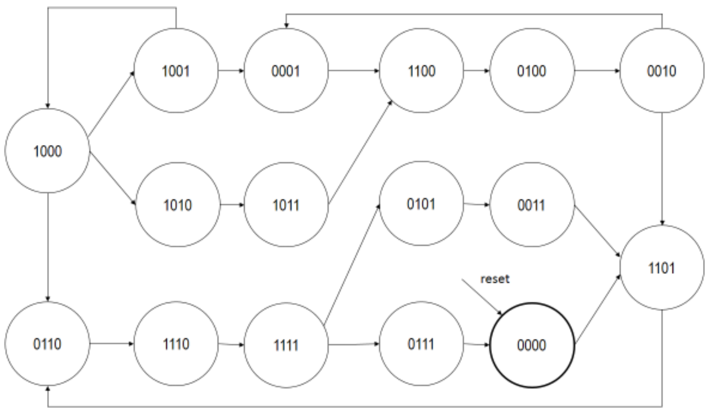


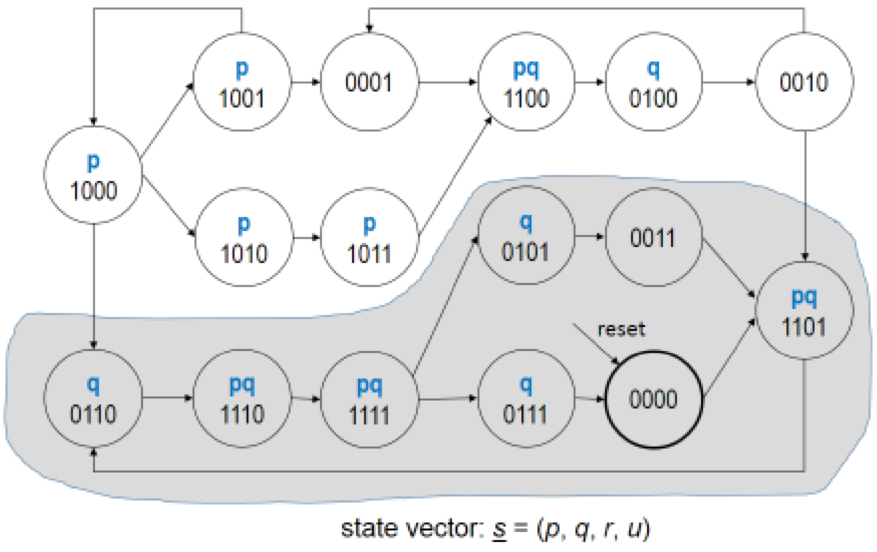
Figure 2: FSM with state vector $\underline{s} = (p, q, r, u)$

Hint: in the state diagram of Figure b, for your convenience when answering the following questions, label the states in which p holds with ‘p’ and the states in which q holds with ‘q’.

1) Which of the above properties hold in the design?

Explain your answer for each of the properties and provide a counter example in case the property fails.

Solution: The reachable state set is shaded grey in the below figure. The labeling of states with p and q is also shown.



Property 1: holds in the design. E.g. the sequence 1000 -> 1001 is a counter example but it does not lie within the reachable states.

Property 2: holds for all states (reachable and unreachable)

Property 3: holds in the design. E.g. the sequence 0100 -> 0010 -> 0001 is a counter example but it does not lie within the reachable states.

2) An IPC checker is used to prove the properties. It unrolls the FSM for the considered time interval and maps property checking to Boolean satisfiability checking. No invariant is used that restricts the state space. Which of the properties are proved to hold by the property checker?

Explain your answer for each of the properties and provide a counter example in case the property fails.

Solution: Property 1: fails since the IPC checker at time t does not distinguish reachable from unreachable states. It will produce spurious counter examples, e.g. the sequence 1000 -> 1001

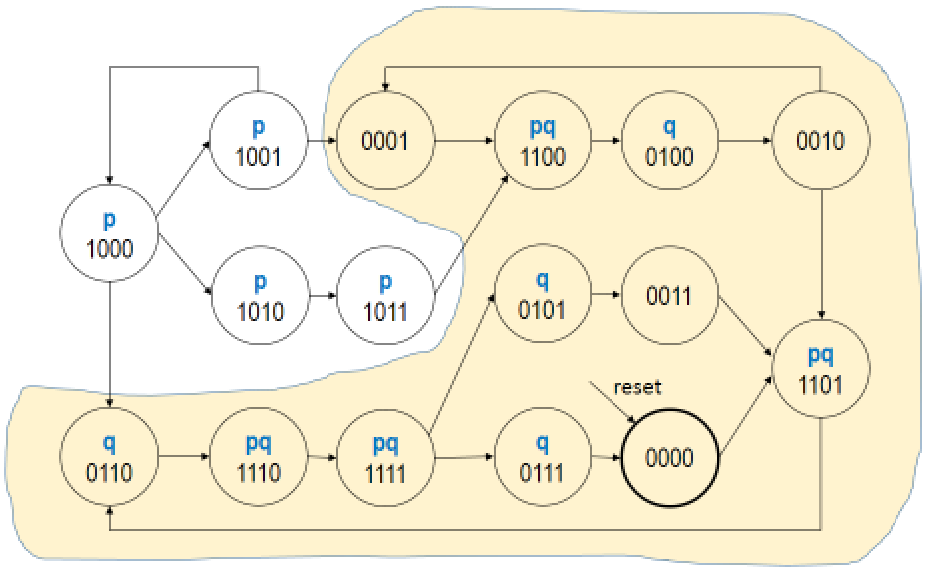
Property 2: can be proven since it holds in all states.

Property 3: fails since the IPC checker at time t does not distinguish reachable from unreachable states. It will produce spurious counter examples, e.g. the sequence 0100 -> 0010 -> 0001

3) As in 2) but the property is strengthened with the invariant $\neg p + q$. Which of the properties are now proved to hold by the property checker?

Explain your answer for each of the properties and provide a counter example in case the property fails.

Solution: The state set defined by this invariant is shown in yellow below.



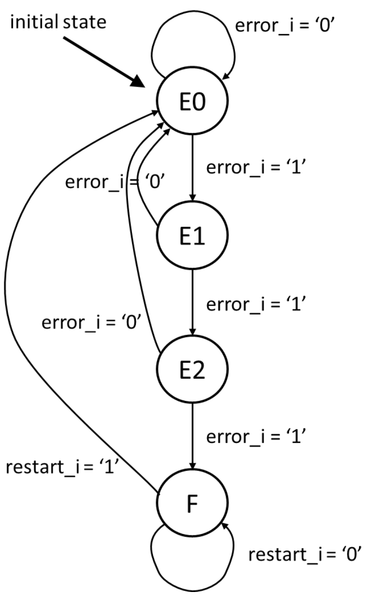
state vector: $\underline{s} = (p, q, r, u)$

Property 1: can now be proven since it holds within the yellow region.

Property 2: can be proven since it holds in all states.

Property 3: fails since the invariant is too weak for this property and does not exclude the spurious counter example 0100 -> 0010 -> 0001

c) (6 p) Consider the following FSM for tracking of an error level:



The machine has three inputs:

- reset_i An asynchronous reset that takes the machine to the initial state E0
- error_i An input from external error detection logic; asserted when an error has occurred.
- restart_i An input to take the machine out of the burst error state F.
- The FSM is a Moore machine with three outputs (not shown in the state transition graph above):
- correct_o Asserted only in state E1; indicates that a first error occurred which is to be corrected.
- dismiss_o Asserted only in state E2; indicates that a second error occurred and that the data should be dismissed.
- fatal_o Asserted only in state F; indicates that a burst of three or more errors occurred.

The FSM states represent four levels of error: E0, E1, E2, and F. Whenever the error_i input is asserted the machine moves to the next error level. Whenever the error_i input is deasserted the machine goes back to error level E0, except for when the FSM is in the "fatal" state F. Once the machine reaches this state, it will remain there until the input restart_i is asserted.

The following SVA module for formal property checking has been written (so far). Note that there is no representation of the internal state variables of the design. Properties are written solely in terms of the parameters of the SVA module, i.e., the inputs and outputs of the design.

```
module errortracker_properties(clk, reset,
error_i, restart_i, correct_o, dismiss_o, fatal_o);

input logic clk;
input logic reset;
input logic error_i;
input logic restart_i;

input logic correct_o;
input logic dismiss_o;
input logic fatal_o;

sequence reset_sequence;
  reset == 1'b1;
endsequence

property p_reset;
        reset_sequence |=> ready;
endproperty

sequence ready;
        // Your solution to question 1 goes here.
        // This sequence matches whenever the FSM is in state E0.
endsequence;

property p_single_error;
        ready
        ##0 error_i
        ##1 !error_i
        implies
        ##1 correct_o && !dismiss_o && !fatal_o
        ##1 ready;
endproperty;

property p_double_error;
        ready
        ##0 error_i
        ##1 error_i
        ##1 !error_i
        implies
        ##1 correct_o && !dismiss_o && !fatal_o
        ##1 !correct_o && dismiss_o && !fatal_o
        ##1 ready;
endproperty;

property p_burst_error;
        // Your solution to question 3 goes here
endproperty;

// The following property is considered in question 2.
property p_lock_burst_error;
        fatal_o |=> fatal_o;
endproperty;

property p_restart;
        fatal_o && restart_i |=> ready;
endproperty;

a_reset: assert property (@(posedge clk) p_reset);
a_single_error: assert property (@(posedge clk) disable iff(reset) p_single_error);
a_double_error: assert property (@(posedge clk) disable iff(reset) p_double_error);
a_burst_error: assert property (@(posedge clk) disable iff(reset) p_burst_error);
a_lock_burst_error: assert property (@(posedge clk) disable iff(reset) p_lock_burst_error);
a_lock_restart: assert property (@(posedge clk) disable iff(reset) p_restart);

endmodule

bind errortracker errortracker_properties inst1_errortracker(.*);
```

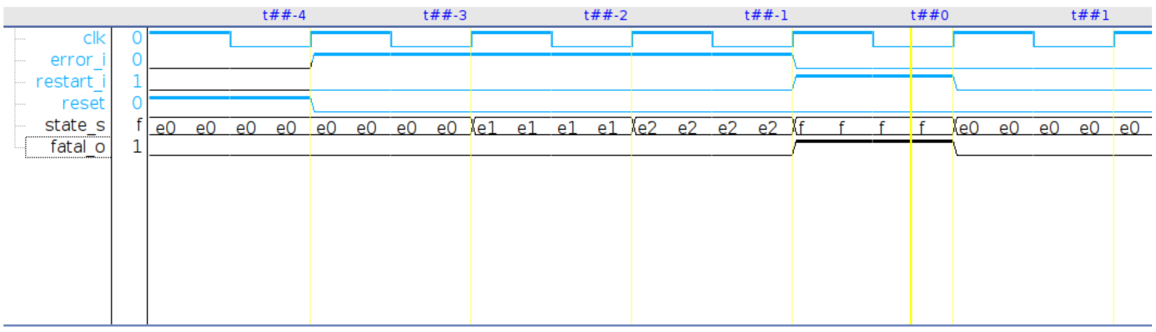
1) Write the body of the definition of the SVA sequence ready. This sequence is used in several properties of the verification module. It matches whenever the design is in state E0. Note that you cannot use the state variables of the design.

Solution:

```
sequence ready;
        correct_o==1'b0 && dismiss_o==1'b0 && fatal_o==1'b0;
endsequence;
```

(In this particular Moore FSM, the states can be uniquely identified by the outputs.)

2) The property p_lock_burst_error checks that when the design is in state F it will remain there. When checked by the property checker, the verification fails and the following counterexample is returned:



What is the problem?

Write a corrected version of the property.

Solution:

The counterexample shows how the design moves into state F after three consecutive errors. At time t+1 the property requires “fatal_o” to be asserted, however in the counterexample it is deasserted. The FSM has moved back to state E0, because the restart_i input was given at time t+0. Write a corrected version of the property.

```
property p_lock_burst_error;
        fatal_o && !restart_i |=> fatal_o;
endproperty;
```

This property states that the design stays in state F unless the restart_i input is asserted.

3) Write the body of property p_burst_error. This property verifies the input/output behavior of the design for the following operation: The design begins in state E0, and three consecutive errors occur.

Solution:

```
property p_burst_error;
        ready
        ##0 error_i
        ##1 error_i
        ##1 error_i
        implies
        ##1 correct_o && !dismiss_o && !fatal_o
        ##1 !correct_o && dismiss_o && !fatal_o
        ##1 !correct_o && !dismiss_o && fatal_o;
endproperty;
```

4) Considering all properties: Is the complete design behavior of the design verified by the property suite, i.e., would a formal completeness check succeed? Explain your answer.

Solution: Answer: No.

- There is no property verifying the “no_error” operation, i.e., the situation when the FSM is in E0 and error_i is not asserted.
- The properties considering the operations in state F verify only the correctness of the output “fatal_o”, not of the other two outputs “correct_o” and “dismiss_o”.

Hence: Not all state transitions are covered, not all output behaviors are covered. A formal completeness check would fail for the property suite.

Problem 4 SystemC (10 points)

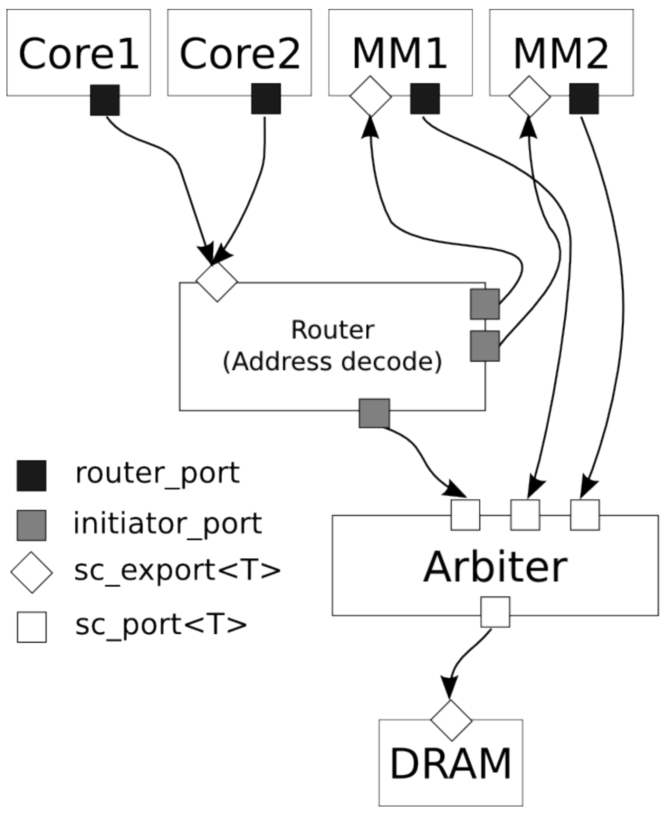
a) (2 p) Describe the principles of SystemC and TLM and how it helps in the design cycle.

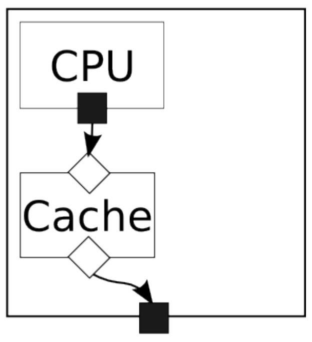
Solution:

- SystemC attempts to span a wider range of the design cycle, from specification through RTL.
- TLM's main objectives are 1) speed of simulation and 2) interoperability and these are achieved through common interfaces and different abstraction levels (functional vs. micro-architectural)
- A high-level executable architectural model can be used in verification, avoiding the need to first write a high-level model and then translate that model to RTL. SystemC is able to handle this through high-level synthesis functionality present in EDA tools
- Software development and testing is able to start much earlier than in a traditional design cycle (e.g., using VHDL), beacuse a functional model for the hardware is available on which the software can be tested soon after the specification is complete

b) (4 p) Sketch a block diagram for a SoC containing at least two identical processor cores, two multimedia processing cores, a DRAM controller and some amount of on-chip SRAM. Mark each end of each connection with a suitable port style to be used as part of a TLM model (e.g., blocking, non-blocking, master, slave).

Solution:


Core1
Core2
MM1
MM2
Router
(Address decode)
Arbiter
DRAM
router_port
initiator_port
sc_export<T>
sc_port<T>


CPU
Cache

c) (4 p) Example 1 shows example SystemC code. Show the result of the simulation (duration 10 ns), and briefly explain the result.

```
#include <systemc.h>
#include <iostream>
using std::cout;
using std::endl;

char* simulation_name = "clock_gen";

SC_MODULE(clock_gen) {
  sc_port<sc_signal_out_if<bool> >  clk1_p;
  sc_export<sc_signal_in_if<bool> > clk2_p;
  sc_clock clk1;
  sc_clock clk2;
  SC_CTOR(clock_gen)
  : clk1("clk1",4,SC_NS)
  , clk2("clk2",6,SC_NS)
  {
    SC_METHOD(clk1_method);
    sensitive << clk1;
    clk2_p(clk2);
  }
  void clk1_method() {
    clk1_p->write(clk1);
  }
};

SC_MODULE(monitor) {
  sc_in<bool> clk1_p;
  sc_in<bool> clk2_p;
  SC_CTOR(monitor) {
    SC_METHOD(clk1_method);
    sensitive << clk1_p;
    SC_METHOD(clk2_method);
    sensitive << clk2_p;
  }
  void clk1_method() {
    cout << "INFO: "<< name()
         << " clk1=" << clk1_p->read()
         << " at " << sc_time_stamp() << endl
            ;
  }
  void clk2_method() {
    cout << "INFO: "<< name()
         << " clk2=" << clk2_p->read()
         << " at " << sc_time_stamp() << endl
            ;
  }
};

int sc_main(int argc, char* argv[]) {
  sc_set_time_resolution(1,SC_PS);
  sc_set_default_time_unit(1,SC_NS);
  sc_signal<bool> clk1;
  clock_gen clock_gen_i("clock_gen_i");
  clock_gen_i.clk1_p(clk1);
  monitor monitor_i("monitor_i");
  monitor_i.clk1_p(clk1);
  monitor_i.clk2_p(clock_gen_i.clk2_p);
  cout << "INFO: Simulating "<<
      simulation_name << endl;
  sc_start(10,SC_NS);
  return 0;
}
```

Example 1

Solution: INFO: Elaborating clock_gen

```
INFO: Simulating clock_gen
INFO: monitor_i clk1=0 at 0 s
INFO: monitor_i clk2=0 at 0 s
INFO: monitor_i clk2=1 at 0 s
INFO: monitor_i clk1=1 at 0 s
INFO: monitor_i clk1=0 at 2 ns
INFO: monitor_i clk2=0 at 3 ns
INFO: monitor_i clk1=1 at 4 ns
INFO: monitor_i clk2=1 at 6 ns
INFO: monitor_i clk1=0 at 6 ns
INFO: monitor_i clk1=1 at 8 ns
INFO: monitor_i clk2=0 at 9 ns
INFO: Post-processing clock_gen
```

```
INFO: Simulation clock_gen PASSED with 0 errors
```

Transfer
Goorseweg 5
7475 BB Markelo
Tel +31 547 367 367

SystemC Quickreference Card

For Training: www.Transfer.nl
email: Training@Transfer.nl

sc_main

```
#include "systemc.h"
// include module declarations

int sc_main(int argc, char *agv[ ])
{
// Create channels
    sc_signal<type> signal_name, signal_name, ...;
// Create clock
    sc_clock clock_name ("name", period, duty_cycle, start_time, positive_first );
// Module instantiations
    module_name instance_name("name") ;
// Module port bindings
// By name binding, do for each port
    instance_name.port_name (signal_name) ;
// By order port binding
    instance_name ( signal_name, signal_name, … ) ;
// By order using stream
    instance_name << signal_name << signal_name, …;
// Clock generation
    sc_start(value);

    return 0;
}
```

Clock syntax

sc_clock clock_name ("name**", period, duty_cycle, start_time, positive_first**) ;

name:	*name*	type: char *	
period:	*clock period*	type: variable of type sc_time or constant of type uint64	
duty_cycle:	*clock duty cycle*	type: double	default value: 0.5
start_time:	*time of first edge*	type: variable of type sc_time or constant of type uint64	
default value: 0			
positive_first:	*first edge positive*	type: bool	default value: true

✍

Clock object methods:

clock_name.**name()**	returns the "name"
clock_name**.period()**	returns the clock period
clock_name.**duty_cycle()**	returns the clock duty cycle
clock_name.**pos()**	Gives a reference to the positive edge of clk usage: sensitive << clock_name**.pos()**
clock_name.**neg()**	Gives a reference to the negative edge of clk usage: sensitive << clock_name**.neg()**

Clock functions

sc_start()	**Generate the waveforms for all sc_clock objects**
sc_stop()	Stops simulations
sc_time_stamp()	Returns the current simulation time as sc_time
sc_simulation_time()	Returns the current simulation time as double

Data Types

Scalar

sc_int<length> variable_name, variable_name, ...**;**
sc_uint<length> variable_name , variable_name, ...**;**
sc_bigint<length> variable_name , variable_name, ...**;**
sc_biguint<length> variable_name , variable_name, ...**;**
✍length: specifies the number of elements in the array.
✍Rightmost is LSB(0), Leftmost is MSB (length-1).
sc_bit variable_name, variable_name, ... **;**
✍Values: '0' , '1'
sc_bv<length> variable_name, variable_name, ... **;**
✍length: specifies the number of elements in the array.
✍Values: '0' , '1'. More than one bit represented by "0011".
sc_logic variable_name, variable_name, … ;
✍Values: '0' , '1', 'X', 'Z'
sc_lv<length> variable_name, variable_name, ... **;**
✍length: specifies the number of elements in the array.
✍Values: '0' , '1', 'X', 'Z' . More than one bit represented by "0011XXZZ".

Fixedpoint

sc_fixed<wl, iwl, q_mode, o_mode, n_bits> object_name, object_name, ... **;**
sc_ufixed<wl, iwl, q_mode, o_mode, n_bits> object_name, object_name, ... **;**
sc_fixed_fast<wl, iwl, q_mode, o_mode, n_bits> object_name, object_name, ..**;**
sc_ufixed_fast<wl, iwl, q_mode, o_mode, n_bits> object_name, object_name. **;**
wl: total word length, number of bits used in the type
iwl: integer word length, number of bits to the left of the binary point (.)
q_mode: quantization mode
o_mode: overflow mode
n_bits: number of saturated bits, used for overflow mode
sc_fix object_name **(list of options) ;**
sc_fix_fast object_name **(list of options) ;**
sc_ufix object_name **(list of options) ;**
sc_ufix_fast object_name **(list of options)**

q_mode: SC_RND, SC_RND_ZERO, SC_RND_MIN_INF, SC_RND_INF, SC_RND_CONV, SC_TRN, SC_TRN_ZERO
o_mode: SC_SAT, SC_SAT_ZERO, SC_SAT_SYM, SC_WRAP, SC_WRAP_SM

Data Operations/Functions

Type / **Operation**	sc_bit sc_bc sc_lv	sc_bc sc_lv	sc_int, sc_uint sc_bigint, sc_biguint	sc_fixed, sc_ufixed, sc_fix, sc_ufix
Bitwise	~ & ^ |	~ & ^ | << >>	~ & ^ | << >>	~ & ^ |
Arithmetic			+ - * / %	+ - * / % >> <<
Logical				
Equality	== !=	== !=	== !=	== !=
Relational			> < <= >=	
Assignment	= &= |= ^=	= &= |= ^=	= += -= *= /= %= &= |= ^=	= += -= *= /= %= &= |= ^=
Increment Decrement			++ --	++ --
Arithmetic if				
Concatenation	,	,	,	,
Bitselect		[x]	[x]	
Partselect		range()	range()	
Reduction		and_reduce or_reduce xor_reduce		

Channels

Name	**Methods**
sc_signal	**read(), write(), event()**
sc_signal_rv	**read(), event(), write()**
For vectors,, allows multiple writers	
sc_signal_resolved	**read(), event(), write()**
For non vectors, allows multiple writers	
sc_fifo	**read(), nb_read(), num_available(), write(), nb_write(), num_free()**
Point to point communication , one reader, one writer per fifo	
sc_mutex	**kind(), lock(), trylock(), unlock()**
Multipoint communication, only one writer/reader at the time	
sc_semaphore	**kind(), wait(), trywait(), get_value(), post()**
Limited concurrent access, specify number of concurrent users	
sc_buffer	**kind()**
Like sc_signal, value_change_event() and default_event() are triggered on each write	

Resolved ports/signals

Syntax:

```
SC_MODULE ( module_name) {
// ports
   sc_in_rv<N> port_name, port_name,...;
   sc_out_rv<N> port_name, port_name,...;
   sc_inout_rv<N> port_name, port_name,...;
   sc_signal_rv<N> signal_name,signal_name,. ;
//  rest of module
} ;      // N is the number of bits
         // Every bit can have either a 0, 1, X or  Z value
```

sc_signal channel methods

read()	retunrs value of signal or port
write()	assigns value to signal or port
event()	returns true or false if event on signal or port
default_event()	any change of value
value_changed_event()	any change of value
posedge()	returns true if 0 -> 1 transition
negedge()	returns true if 1 -> 0 transition

Modules

```
// Header file
SC_MODULE(module_name) {
   // module port declarations
   // signal variable declarations
   // data variable declarations
   // process declarations
   // other method declarations
   // module instantiations
SC_CTOR(module_name){
   // process registration & declarations of sensitivity lists
   // module instantiations & port connection declarations
   // global watching registration
            }
} ;

// Implementation file
void module_name::process_or_method_name() {
   // process implementation
   // SC_THREAD and SC_CTHREAD has
   // while(true) loop
}
```

Transfer
Goorseweg 5
7475 BB Markelo
Tel +31 547 367 367

SystemC Quickreference Card

For Training: www.Transfer.nl
email: Training@Transfer.nl

Scalar Syntax:

```
SC_MODULE(module_name) {
// ports
   sc_in<port_type>  port_name, port_name,... ;
   sc_out<port_type> port_name, port_name,... ;
   sc_inout<port_type> port_name, port_name,... ;
   sc_port<channel_type<port_type>, connections > port_name, port_name,... ;
   sc_port<channel_type<port_type>, connections > port_name, port_name,... ;
   sc_port<channel_type<port_type>, conections > port_name, port_name,... ;
// clock input (for SystemC 2.0 it is recommended to use sc_in<bool>)
   sc_in_clk clock_name;
// clock output (for SystemC 2.0 is is recommended to use sc_out<bool>)
   sc_out_clk clock_name;
// signals
   sc_signal<signal_type> signal_name, signal_name, ...;
//  variables
   type variable_name, variable_name...;
//  rest of module};
```

Array Syntax:

```
SC_MODULE ( module_name) {
// ports
   sc_in<port_type>  port_name[size], port_name[size], ... ;
   sc_out<port_type> port_name[size], port_name[size], ... ;
   sc_inout<port_type> port_name[size], port_name[size], ...;
   sc_port<channel_type <port_type>, connections >port_name[size], port_name[size], ... ;
   sc_port<channel_type <port_type>, connections >port_name[size], port_name[size], ... ;
   sc_port<channel_type <port_type>, connections >port_name[size], port_name[size], ...;
// signals
   sc_signal<signal_type> signal_name [size], signal_name [size], ...
//  variables
   type variable_name[size], variable_name[size],...;;
//  rest of module
} ;
```

Module inheritance

```
SC_MODULE( base_module )
{
...
// constructor
SC_CTOR( base_module )
   { ... }
};
   class derived_module : public base_module
{
// process(es)
   void proc_a();
   SC_HAS_PROCESS( derived_module );
// parameter(s)
   int some_parameter;
// constructor
   derived_module( sc_module_name name_, int some_value )
   : base_module( name_ ), some_parameter( some_value )
{
   SC_THREAD( proc_a );
}
};
```

Processes

```
// Header file
SC_MODULE(module_name) {
// module port declarations
// signal variable declarations
// data variable declarations
// process declarations
   void process_name_A();
   void process_name_B();
   void process_name_C();
// other method declarations
// module instantiations
   SC_CTOR(module_name){
// process registration
   SC_METHOD(process_name_A);
// Sensitivity list
   SC_THREAD(process_name_B);
// Sensitivity list
   SC_CTHREAD(process_name_C, clock_edge_reference);
   //clock_name.pos() or clock_name.neg()
// global watching registration
// no sensitivity list
/ module instantiations & port connection declarations
}
} ;
```

Sensitivity list

Sensitive to any change on port(s) or signal(s)
sensitive(port_or_signal)
sensitive << port_or_signal **<<** port_or_signal …;
Sensitive to the positive edge of boolean port(s) or signal(s)
sensitive_pos(port_or_signal)
sensitive_pos << port_or_signal **<<** port_or_signal …;
Sensitive to the negative edge of boolean port(s) or signal(s)
sensitive_neg(port_or_signal)
sensitive_neg << port_or_signal **<<** port_or_signal …;

Module instantiation

Style 1

```
// Header file
SC_MODULE(module_name) {
   // module port declarations
   // signal variable declarations
   // data variable declarations
   // process declarations
   // other method declarations
module_name_A instance_name_A; // module instantiation..
module_name_N instance_name_N; // module instantiation

SC_CTOR(module_name):
instance_name_A("name_A"),
instance_name_N("name_N")
{
// by name port binding
   instance_name_A.port_1(signal_or_port);
// by order port binding
   instance_name_N(signal_or_port, signal_or_port,...);
// process registration & declarations of sensitivity lists
// global watching registration
}
} ;
```

Style 2

```
// Header file
SC_MODULE(module_name) {
   // module port declarations
   // signal variable declarations
   // data variable declarations
   // process declarations
   // other method declarations
module_name_A *instance_name_A; // module instantiation..
module_name_N *instance_name_N; // module instantiation
SC_CTOR(module_name)
{
   instance_name_A = new module_name_A("name_A"),
   instance_name_N = new module_name_N("name_N")
   instance_name_A->port_1(signal_or_port);
   instance_name_A->port_2(signal_or_port);
(*instance_name_N)(signal_or_port, signal_or_port,...);
// process registration & declarations of sensitivity lists
// global watching registration
}
} ;
```

Watching

```
// Header file
SC_MODULE(module_name) {
   // module port declarations
   // signal variable declarations
   // data variable declarations
   // process declarations
void process_name();// other method declarations
// module instantiations
SC_CTOR(module_name){
SC_CTHREAD(process_name, clock_edge_reference  // global watching registration
            watching (reset.delayed() = = 1);  // delayed() method required
}
```

Event

```
sc_event my_event;    // event
sc_time t_zero (0,sc_ns);
sc_time t(10, sc_ms); // variable t of type sc_time

Immediate:
   my_event.notify();
   notify(my_event);
Delayed:
   my_event.notify(t_zero);       // next delta cycle
   notify(t_zero, my_event);      // next delta cycle
Timed:
   my_event.notify(t);        // 10 ms delay
   notify(t, my_event);       // 10 ms delay
```

Dynamic sensitivity

```
wait for an event in a list of events:
   wait(e1);
   wait(e1 | e2 | e3);
   wait( e1 & e2 & e3);
wait for specific amount of time:
   wait(200, sc_ns);
wait on events with timeout:
   wait(200, sc_ns, e1 | e2 | e3);
wait for number of clock cycles:
   wait(200); // wait for 200 clock cycles, only for SC_CTHREAD
wait for one delta cycle:
   wait( 0, sc_ns ); // wait one delta cycle.
   wait( SC_ZERO_TIME ); // wait one delta cycle.
```