Problem 1 Systemverilog (15%)

a) (3%) Systemverilog extends Verilog with an `interface` declaration. What are interfaces and what purpose do they serve. Explain in short.

> Simple interfaces are declared "collections"/groups of signals that defines a common structure that can be included in the port declaration of one or more modules. This facilitates a unique declaration of eg. communication structures. In general, interfaces may also contain most structures that can be contained in modules such as data types, variables, functions, tasks, initial, final and always blocks. Also, sequences, properties and assertions can be contained in interfaces.

b) (3%) What are deferred assertions? Which two types of deferred assertions are there, and what is the difference. One of these can be used standalone in interfaces, but which one? What is the purpose of assertions in interfaces?

> They are assertions that are evaluated at the end of the current simulation time cycle, and that means that they will not be triggered by glitches. The simple deferred assertion (`assert #0 ...`) are evaluated at the end of the current active region, while the final deferred assertion (`assert final ...`) is evaluated at the end of the current re-active region. This means that final deferred assertions are also independent of glitches due to 0 delay changes in the program part

c) (3%) What is the purpose of a `cover` statement, and what makes it different from a `property` statement. Mention in short the different types of `cover` statements.

> A cover statement declares what is called functional coverage, the gathering of information during simulation of what has been executed in order to check degree of completeness of tests for example. There are as for assertions immediate, deferred and concurrent cover statements. Where an asserted property is meant to test a requirement of the system, what the system can do and can not do, the cover statements intyent is to report, check or count how the execution takes place.

d) (3%) Systemverilog also extends Verilog with the `program` declaration. What are programs for, and why are they different from modules.

> Programs declare test benches, they can not be turned into HW but are meant for designing the stimuli, monitoring and checking part of the design process. Programs can not contains always blocks as opposed to modules, but they will typically contain initial and final blocks to set up stimuli and check results.

e) (3%) Explain in short the main (top level) regions in the simulator that takes care of the different semantics ("meanings") of modules, assertions, and programs.

> The active region evaluates the expressions in the modules, after that comes the observed region which evaluates and checks expressions in assertions, and at last is the re-active region which evaluate the program expressions for stimuli and checking.

Problem 2 Systemverilog Assertions (30%)

a) (5%) Write the SVA code for a parameterized property with parameters a clock (`clk`) and a bit array (`d`) that tests for even parity of `d` as sampled on the positive edge of the clock `clk`. By even parity we mean that the number of 1's in the bit array is an even number. Show that it is possible to solve both by the Verilog xor operator (^) and by using built in functions of SVA.

Using the bitwise xor operator (^):

```
property even_par (bit clk, bit d[]);
    @ (posedge clk) ^ d;
end property;
```

or the count ones and % (modulo):

```
property even_par (bit clk, bit d[]);
    @ (posedge clk) ($count_ones(d) % 2) == 0;
end property;
```

b) (5%) Assume we shall assert some properties for a state machine where we know that the state variable is of the following declared type:

```
typedef enum {IDLE, LDI, LDD, ST} state_T;
state_T fsm_state;
```

We do not know the exact conditions for transitions between the states. However, we would like to ascertain that the FSM holds the following requirements: (1) From any other state apart from IDLE itself it should eventually return to IDLE. (2) From the specific states LDI and LDD the FSM should always return to IDLE within a specific but parameterizable time delay ret_del. Write up those two properties and possibly necessary sequences.

```
property return_idle;
    (fsm_state !== IDLE)
        |=> s_eventually (fsm_state == IDLE);
end property;

property return_idle2;
    (fsm_state !== IDLE)
        |-> ##[1:$] (fsm_state == IDLE);
end property;

property return_delay(int maxdel;
    fsm_state == LD1 || fsm_state == LD2
        |-> ##[1:maxdel] (fsm_state == IDLE);
end property;
```

c) (10%) We have a simple generator of waveforms for the two bit signals `a` and `b` as shown here:

```
module p2; bit clk,a,b;

always #10 clk = !clk;

initial begin {a,b}=2'b00; end

initial begin
    @(negedge clk); {a,b}=2'b10;
    @(negedge clk); {a,b}=2'b01;
    @(negedge clk); {a,b}=2'b11;
    @(negedge clk); {a,b}=2'b10;
    @(negedge clk); {a,b}=2'b01;
    @(negedge clk); {a,b}=2'b11;
    @(negedge clk); {a,b}=2'b11;
    @(negedge clk); {a,b}=2'b10;
    @(negedge clk); {a,b}=2'b10;
    @(negedge clk); {a,b}=2'b10;
end
```

We have the following properties (within some standard default clocking scheme):

```
property p1; @(posedge clk) a ##[2:3] b; endproperty

property p2; @(posedge clk) a |-> ##[2:3] b; endproperty

property p3; @(posedge clk) a[-> 2] ##0 b[-> 3]; endproperty

property p4; @(posedge clk) a[-> 2] within b[-> 3];
endproperty
```

For each of these 4 properties, assume that they are asserted in the standard way with an action part for both `pass` and `fail` that just display a PASS or FAIL message. Please indicate each time they either PASS or FAIL.

```
#      10   p1 FAIL
#      50   p1 FAIL
#      70   p2 PASS
#      70   p1 PASS
#     110   p2 PASS
#     110   p4 PASS
#     110   p4 PASS
#     110   p4 PASS
#     110   p1 FAIL
#     110   p1 PASS
#     130   p2 PASS
#     130   p4 PASS
#     130   p3 PASS
#     130   p3 PASS
#     130   p1 PASS
#     150   p4 PASS
#     150   p4 PASS
#     150   p3 PASS
#     150   p3 PASS
#     190   p2 FAIL
#     190   p1 FAIL
#     210   p2 FAIL
#     210   p1 FAIL
```

d) (5%) Take the following simple assertion under the assumption of a standard default clocking scheme:

```
assert property (x |-> y until !x);
```

What happens when `x` is deasserted (false)? What do we call this, and is it a real problem in assertion based verification?

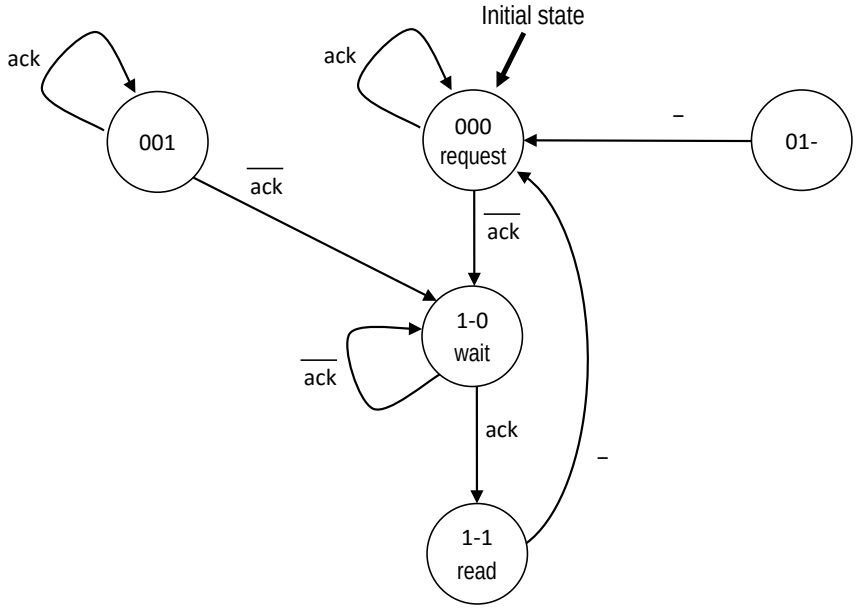
The challenge is if x is false and y is true. Then the consquent would be true for one sampling period when y and until !x is matched in the next sampling. And since an implication a -> b is logically true when the antecedent is false no matter what the value of the consequent, that means the assertion would logically pass! This is what is called vacuity. However, most simulators are able to detect this and filter out the vacuous passes. This example, though, can trick some of the simulators because of the tricky consequent and pass no matter what. Mentor Questa was able to detect it though... If you answered vacuity or vacuous passes and that simulators may filter them away then you are correct.

e) (5%) We are now verifying the interface (black box) of a multiply-accumulate core (MAC). What we know and expect on the interface level is that whenever the `mac_op` signal is asserted, there will follow 3 times that `read_op` is asserted and 2 times `write_op`. However, we do not know anything about the order of these reads and writes, we just know that the first op will be a read and the last a write. Create a property that can verify this protocol operation.

```
property rw_proto;
    read_op |->
        read_op[=2] within write_op[->2];
end property;
```

Problem 3 Reachable states, properties and invariants (20%)

The following questions relate to the finite state machine with state vector $s = (x, y, z)$ and input *ack*, as given by the state transition diagram below. ("–" denotes the don't care value.)



a) (5%) What are the reachable states of the system?

Possible answer: *request*, *wait* and *read*.
Also possible answer: $\{000, 1-0, 1-1\}$
Also possible answer: $\{000, 100, 110, 101, 111\}$

b) (5%) Consider the following property `operation1` to be proven on a design implementing the finite state machine given above.

```
property operation1 is
  assume:
    at t:      x='0' and y='0';
    at t:      ack='0';
    at t+1:    ack='1';

  prove:
    at t+2:    x='1' and z='1';
end property;
```

Does the property hold for the design?

> Yes.

In case the property does not hold: provide a counter example to the property. In case the property holds: can the Interval Property Checker prove this property without strengthening it by an invariant? Explain your answer.

> The property holds in the reachable states as well as in the unreachable states covered by the property. Therefore, the property does not need to be strengthened by an invariant. (Note: it is not enough to say that the property holds for the reachable state set.)

c) (5%) Now consider the following property `operation2`.

```
property operation2 is
  assume:
    at t:    x = '0';
    at t:    ack = '0';
    at t+1:  ack = '1';
  prove:
    at t+2:  x = '1' and z = '1';
end property;
```

This property holds on the design. However, without strengthening the property by an invariant the Interval Property Checker produces a spurious counterexample. Which counterexample does it produce?

> Counterexample:
> t = 0: s = (01-) with ack = 0
> t = 1: s = (000) with ack = 1
> t = 2: s = (000)

d) (5%) For each of the following Boolean formulas, determine whether it represents an invariant that is strong enough to prove the property. Briefly explain your answer.

$\overline{\mathbf{x}} \cdot \overline{\mathbf{y}} + \overline{\mathbf{x}} \cdot \mathbf{y}$

No, this is not an invariant.

$\overline{\mathbf{x}} \cdot \overline{\mathbf{y}} \cdot \overline{\mathbf{z}} + \mathbf{x} \cdot \overline{\mathbf{z}} + \mathbf{x} \cdot \mathbf{z}$

Yes, this is the reachable state set.

$\mathbf{y} \rightarrow \mathbf{x}$

Yes. It excludes state $(0, 1, -)$ and no other state. It defines a superset of the reachable state set in which the property holds.

$\overline{\mathbf{y}} \rightarrow \mathbf{x}$

No, this is not an invariant and it excludes the initial state.

Problem 4 Use of formal verification (10%)

a) (5%) Are the following statements true or false? Explain your answer.

In the Spectre and Meltdown attacks a subtle violation of the security protocol leads to a leakage of confidential information.

False. No protocol is violated, the designs are functionally correct.

In the Spectre and Meltdown attacks secret data like an encryption key can flow from the protected part of the memory into a register which the attacker can read.

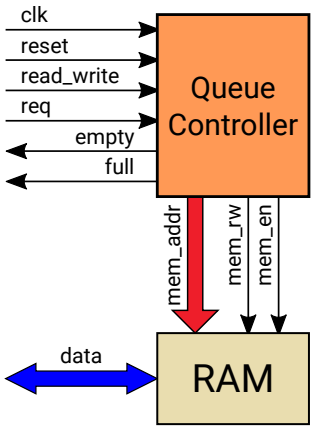
False. Spectre and Meltdown are side channel attacks. Information is not leaked directly.

b) (5%) Explain the notion of a timing-based microarchitectural side channel.

The execution of a program leaves a microarchitectural footprint which can depend on secret data. The attacker analyzes this footprint to deduce the secret information. In case of a timing-based side channel, the attacker measures the time needed to execute certain instruction sequences. The instruction sequence is constructed such that its execution time depends on previously executed instructions involving secret data.

Problem 5 Interval Property Checking (20%)

In the following, we consider the design of a FIFO memory (FIFO = "first-in, first-out"), based on a RAM and a queue controller. The FIFO can hold up to N data items. The data is stored in the RAM. The queue controller generates addresses and control signals for the RAM.



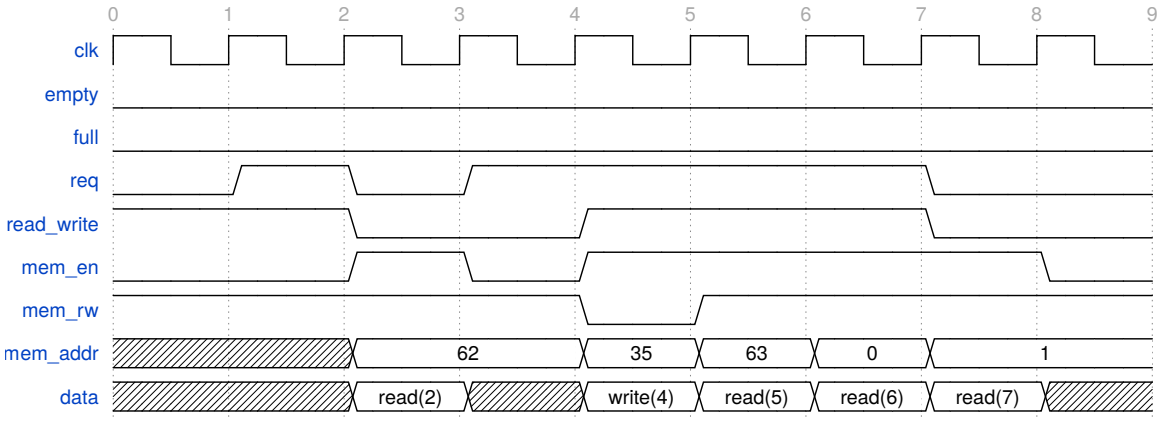
The user interface consists of the signals **reset** (active high), the request signal **req** and the signal **read_write** which indicates a read transaction when it is asserted and a write transaction when not asserted. The controller provides also two flags to the user: Signal **empty** is asserted whenever there are no data stored in the FIFO. Signal **full** is asserted, when the number of stored data items is at its maximum, N. (The two flags can never be asserted at the same time.) After reset, the FIFO is empty.

With respect to the RAM, the controller generates an address to a data item, **mem_addr**, as well as two control signals: one "enable" signal, **mem_en**, and one signal controlling the data direction: *read* when **mem_rw** is asserted, and *write* otherwise.

Internally, the queue controller maintains two address pointers, a *read pointer* and a *write pointer* (not visible in the figure). In a *read* transaction, the read pointer value is presented at the **mem_addr** output. In a *write* transaction, the write pointer value is presented at the **mem_addr** output. The generated addresses range from 0 to $N - 1$.

The environment sends 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 in the same clock cycle 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.

For illustration, the following timing diagram demonstrates a few read and write transactions. The FIFO capacity in this design is $N = 64$.



In this example, 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 capacity, $N = 64$. 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.

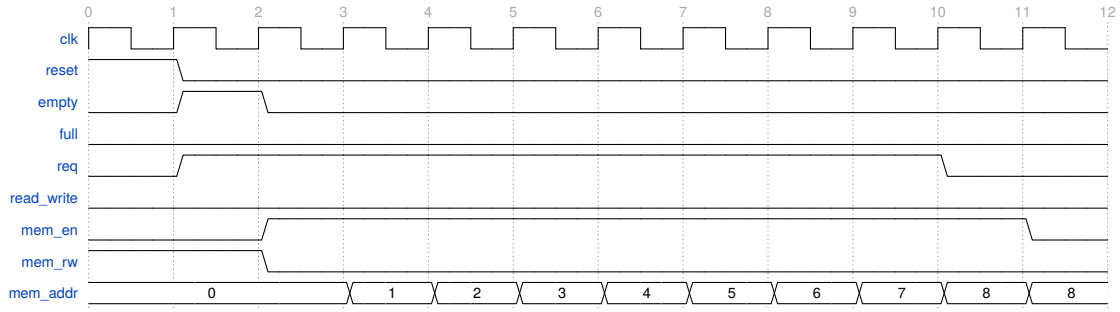
a) (6%)

The following property considers a write transaction into a FIFO that is not full. It specifies that the generated memory addresses correctly wrap around at the largest address, N, which is equal to the FIFO capacity. **In this design, the FIFO capacity is:** $N = 8$.

```
property write_request_wrapping;
        !full && req && !read_write
    ##1 !full && req && !read_write && mem_addr == N-1
  implies
    ##1  mem_en && !mem_rw
    ##1  mem_en && !mem_rw && mem_addr == 0;
endproperty;

a_write_request_wrapping:
assert property (@(posedge clk)
  disable iff (reset) write_request_wrapping);
```

When checking this property, the verification tool returns the following counterexample:



Is this a true or a false counterexample? Please, explain your answer.

Correct answer:
The counterexample must be a true counterexample because it begins with reset.

(Note: An Interval Property Checker, by construction, may return false (unreachable) counterexamples, because it allows the design to be in an arbitrary state at the beginning of the property. However, this counterexample begins with reset so it is reachable.)

A wrong answer would be:
"The counterexample is a true one because the address does not wrap around in clock cycle 11."

b) (6%) Write an SVA property specifying that at all times, the signals **full** and **empty** are never asserted at the same time.

```
property p_not_full_and_empty;
   !(full && empty);
endproperty;

a_not_full_and_empty:
assert property (@(posedge clk)
   disable iff (reset)
   p_not_full_and_empty);
```

c) (8%) Write an SVA property specifying the correct generation of the signals **mem_en** and **mem_rw** for the case of a read request when the FIFO is not empty. (You do not have to specify the generation of memory addresses.)

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

a_read_when_not_empty:
assert property (@(posedge clk)
   disable iff (reset)
   p_read_when_not_empty);
```

Problem 6 UVM–Universal Verification Methodology (5%)

Explain in short the different types and purpose of *agents* in the UVM architecture.

> In a UVM test environment there are at least three types of active agents: Sequencers, drivers and monitors. Sequencers will control the set of test sequences and force them on the driver at the right test time. The driver is responsible for translating the sequence item into a format (signal values) that conforms to the interface port of the DUT. The monitor will read both the inputs and outputs from the DUT, translate them into sequences and hand them over to scoreboards and checkers for controlling the result and keeping track of the functional coverage.