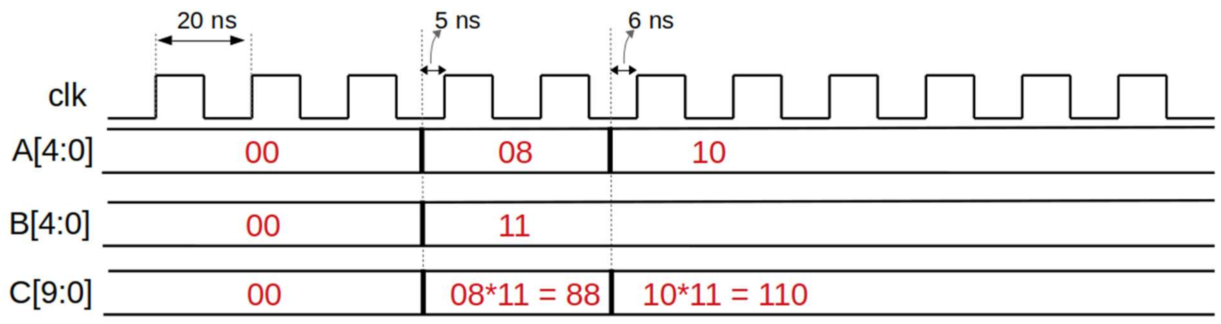
EXAM 2020

Proposed Solutions

1- SystemVerilog (15p)

As a verification engineer you are asked to generate the following waveform using SystemVerilog:



1a) Generate A, B, and C signals using a SystemVerilog task (clk is an input port). (12p)

It can be imiplemented in different ways. For example the following code shows a possible implementation using delay. One may use a combination of clock edges and delays.

```
task signal_generation( input logic clk,output logic[4:0] A, B,output logic[9:0] C);
@(posedge clk)
    A = 0;
    B = 0;
    C = 0;

   #55ns
  A = 8;
   B = 11;
   C = A*B;

   #39ns
  A = 10;
  C = A*B;
endtask
```

1b) Can you implement it using a SystemVerilog function? Explain your **Yes** or **No** answer. (3p)

No, functions cannot include delay statements in SystemVerilog.

2- SystemVerilog Assertions (25 p)

2a) Write a SystemVerilog assertion that checks the following design property in an arbiter with `request`, `ack`, and `grant` signals: (5p)

"When the `request` signal becomes active (high), the `ack` signal becomes active after two or more clock cycles, remains active until it becomes inactive (low) at the same cycle as the `grant` signal gets active".

```
property p_arb_ex;
    @(posedge clk) request |-> ##[2:$] ack [*1:$] (grant && !ack);
endproperty
A_arb_ex : assert property (p_arb_ex);
```

2b) Assume we have a finite state machine with the following states:

```
Typedef enum {IDLE, WAIT1, WAIT2, EXEC1, EXEC2, EXEC3, ACK}
```

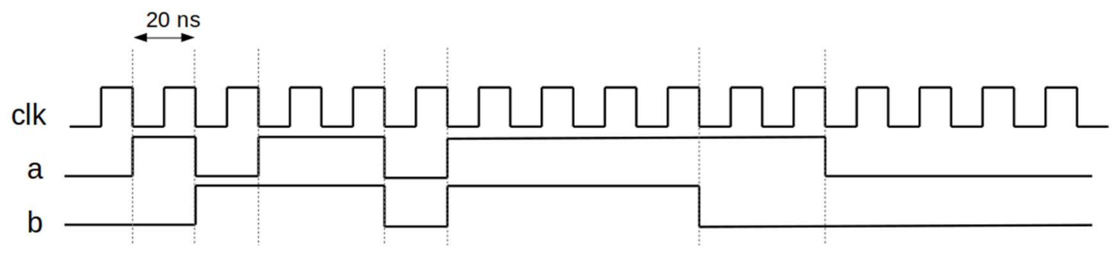
Write SystemVerilog assertions for checking the following design properties:

Property P1: "The state machine eventually goes to `IDLE` state from `EXEC1`, `EXEC2`, `EXEC3`, and `ACK` states". (5p)

Property P2: "Within a specific but parameterized number of clock cycles, transition from states `EXEC1`, `EXEC2`, and `EXEC3` must happen to either `ACK` or `IDLE` state". (5p)

```
sequence s_is_exec;
fsm_state == EXEC1 || fsm_state == EXEC2 || fsm_state == EXEC3;
endsequence
property p1;
   @(posedge clk) s_is_exec || fsm_state == ACK |-> ##[1:$] IDLE;
endproperty
property p2 (int delay);
  @(posedge clk) s_is_exec |-> ##[1:delay] (fsm_state == IDLE || fsm_state == ACK);
endproperty
a_p1 : assert property (p1);
a_p2 : assert property (p2(the_specified_delay));
```

2c) Considering the following waveform, specify on which time points the assertions on the following properties pass or fail (ignore vacuous passes, and assume the first rising edge of `clk` happens at time 0 ns and it has a duty cycle of 50%). (10p)



- property p1; @(posedge clk) a |-> ##[1:4] b; end property;
- property p2; @(posedge clk) a[-> 2] ##1 b[-> 2]; end property;
- property p3; @(posedge clk) a[-> 2] within b[-> 3]; end property;

- (some might consider the end of the failed evaluation which is also correct)
- P1 Failed evaluation starts at : 180 ns, 200 ns, 220 ns
- P2 Failed evaluation starts at : 160 ns, 180 ns, 200 ns, 220 ns
- p3 Failed evaluation starts at : no fail

3- SystemVerilog Coverage (10):

Assume we have a finite state machine with the following states:

{S0, S1, S2, S3, S4, S5, S6, S7, S8}

This state machine drives two output ports (CMD and ADDRESS[8:0]) towards a memory block and can perform *read* and *write* operations in the memory. The CMD port can have two values Read and Write.

Considering these, define *coverage points*/*coverage bins* to check if:

3a) All states are covered. (2p)

3b) Both Read and Write commands are tested in states S2, S3, S4, S5, S6. (4p)

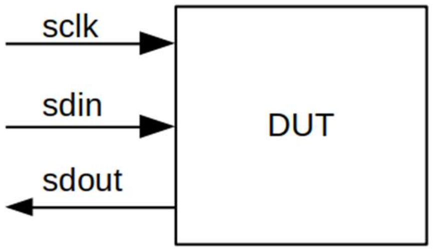
3c) Both Read and Write operations are tested on all 512 memory entries. (4p)

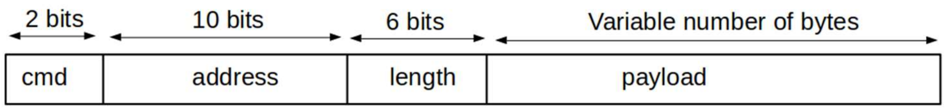
```
covergroup g_task3 @(posedge clk);
// TASK 3-a
// Assuming that the FSM has a variable FSM_STATE keeping track of its state.
C_all_states : coverpoint FSM_STATE;
// TASK 3-b
C_cmd : coverpoint CMD;
C_special_states : coverpoint FSM_STATE
{
Bins states [5] = {S2, S3, S4, S5, S6};
}
```

```
C_cmd_in_special : cross c_special_states, c_command;
// TASK 3-c
C_addr: coverpoint ADDRESS;
C_cmd_in_addr : cross c_addr, c_cmd;
Endgroup
```

4-UVM (10p)**:**

A Design Under Test (DUT) with its three single-bit ports (`sclk`, `sdin`, and `sdout`), its serial communication packet format, as well as some constraints are shown in the following figures:





Packet Format

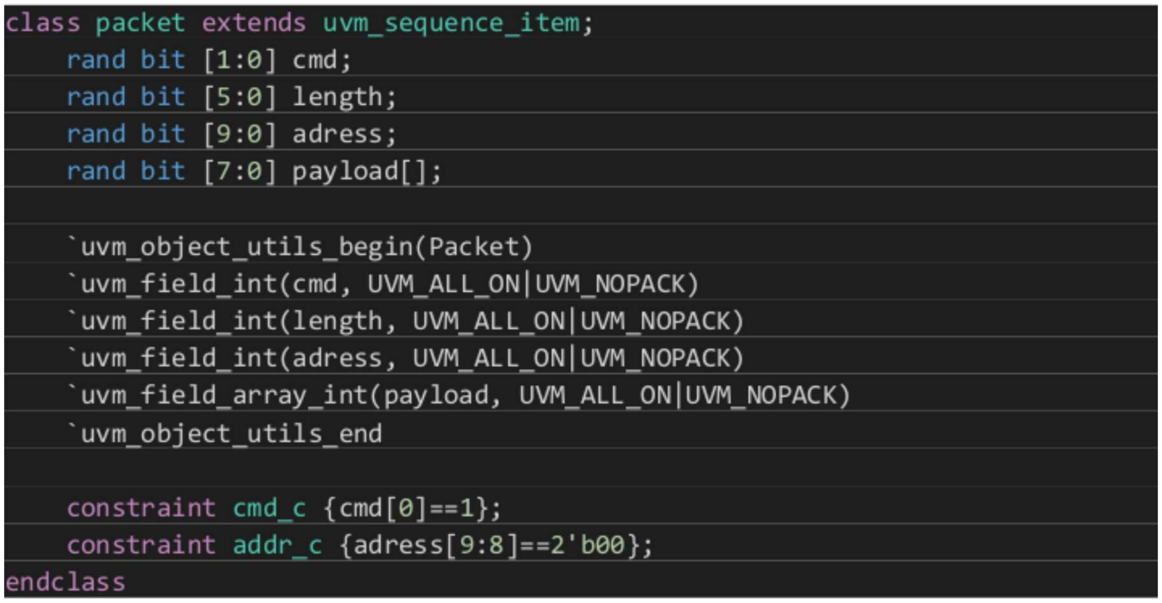
Constraints:

- Only `address[7:0]` bits are valid for the DUT
- `cmd[0]` is always '1'

You as the verification engineer are assigned to verify the DUT using UVM.

4a) Define a transaction class (sequence item class) that represents/encapsulates the packet, considering the provided constraints. The objects of this class can be used to generate random packets (with random `cmd`, `address`, `length`, and `payload`). (5p)

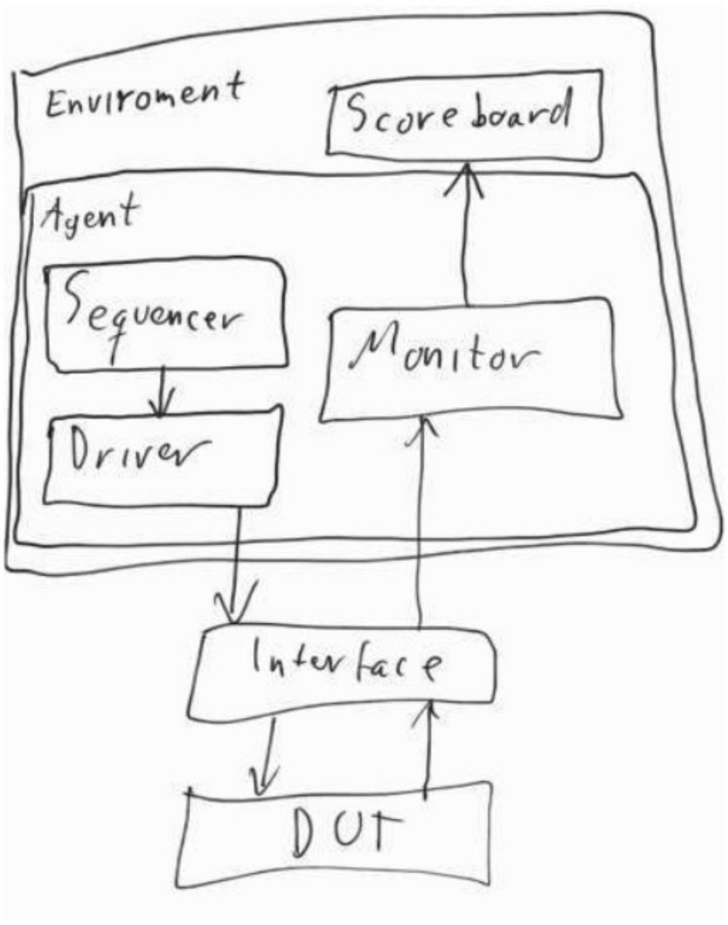
4b) Developing an *agent* for the communication protocol increases the reusability of the verification logic. Without showing/describing the details of implementation, by using a block-diagram show how the components inside an ***active*** agent for this protocol, test sequences, scoreboard, test environment, virtual interfaces, testbench, and the DUT are connected. (5p)



```
class packet extends uvm_sequence_item;
    rand bit [1:0] cmd;
    rand bit [5:0] length;
    rand bit [9:0] adress;
    rand bit [7:0] payload[];

    `uvm_object_utils_begin(Packet)
    `uvm_field_int(cmd, UVM_ALL_ON|UVM_NOPACK)
    `uvm_field_int(length, UVM_ALL_ON|UVM_NOPACK)
    `uvm_field_int(adress, UVM_ALL_ON|UVM_NOPACK)
    `uvm_field_array_int(payload, UVM_ALL_ON|UVM_NOPACK)
    `uvm_object_utils_end

    constraint cmd_c {cmd[0]==1};
    constraint addr_c {adress[9:8]==2'b00};
endclass
```



5 – SoC (20 p)

5a) What is SoC? Describe SoC components and present a block diagram of a common Soc structure with functional components and interfaces. (2p)

a.

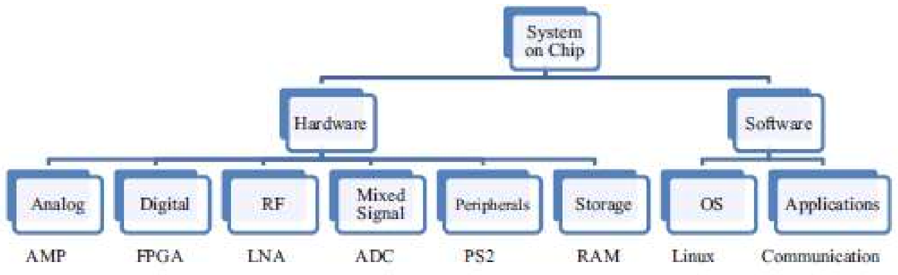
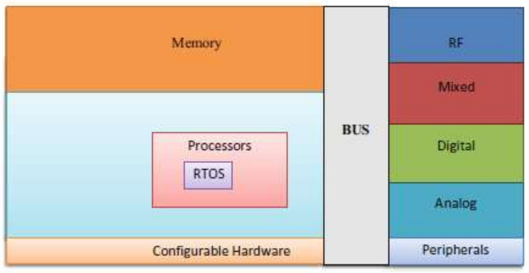


Fig. 1.2 SoC components: it contains hardware and software. Not all software fits on hardware, we have to check the compatibility



The SoC Hardware includes:

– Embedded processor
– ASIC Logics and analog circuitry
– Embedded memory
– Peripherals

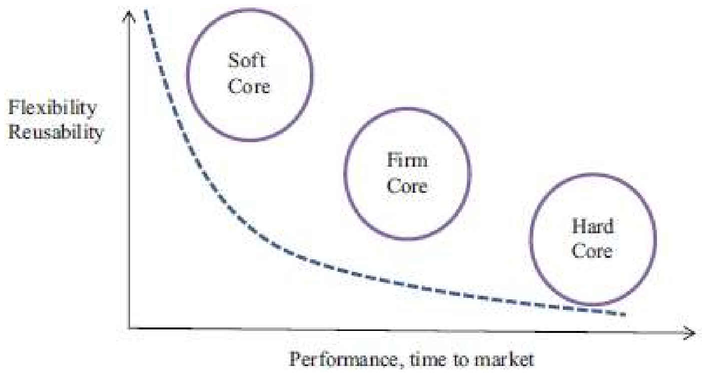
The SoC Software includes:

– OS/RTOS (Middleware, Device Drivers)
– Applications (C/C++, assembly)

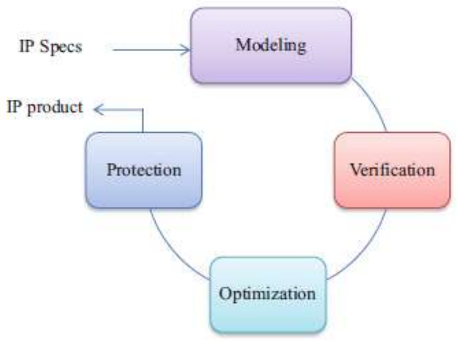
5b) A simple Xilinx SoC system is presented in the figure below. Describe functional components and

b. Processor system (2 ARM cores),
DDR and Fixed_IO external connections to interface ports for accessing DDR memory and to fixed Processing system peripherals
FPGA components: TIMER, external peripherals LEDS AND BUTTONs connected through AXI GPIO interface
Processing system is connected through the AXI interconnect with the components on PL side, where PS is a master and axi_timer, axi_gpio_0 and axi_gpio_1 are slaves.
Interrupt support timer and GPIO component connected to button peripheral

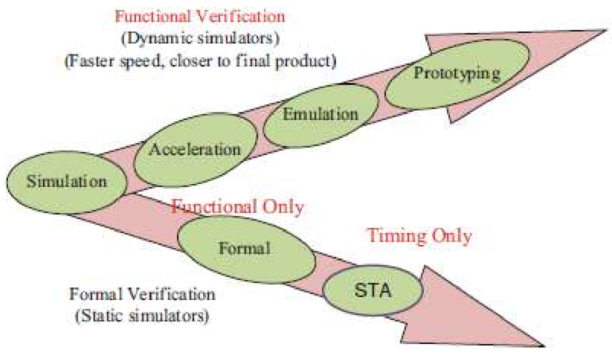
5c) Describe classification of hardware IP cores. Elaborate flexibility-performance trade-off for each type of the IP core. (3p)



5d) Describe each phase of FPGA-based IP core life cycle process. (6p)



1.) FPGA-based Modeling: defined by fixed functionality and connectivity of hardware elements.
2.) FPGA-Based/Processor-Based IP Verification
    a. Function-based verification
        i. Simulation-based
        ii. Accelerator-based
        iii. Emulation-based
        iv. FPGA prototyping
    b. Formal-based verification
        i. Assertion-based



3.) FPGA-Based IP Optimization

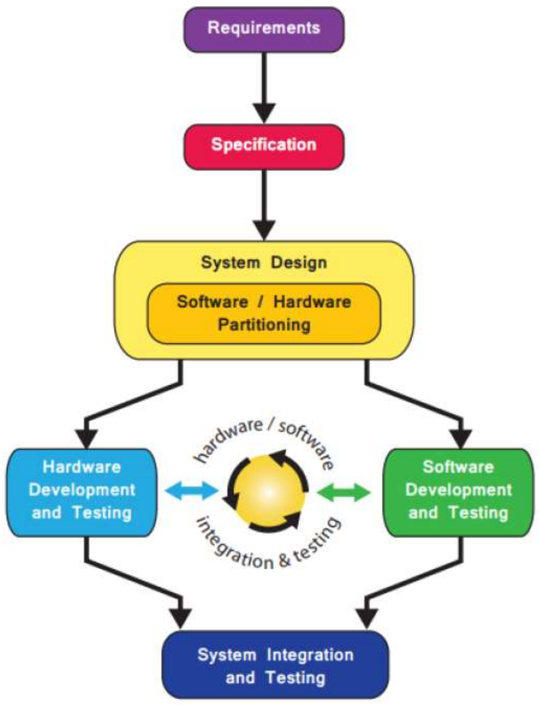
The optimization objective is to reduce area, delay, latency, and power and to increase performance and speed to meet the requirement.

   a) Compilation time optimization.
   b) Maximum frequency optimization.
   c) Following some RTL design tips.

4.) FPGA-Based/Processor-Based IP Protection

   a) Obfuscation
   b) Encryption
   c) Watermarking for soft IPs

5e) Describe the phases of a SoC codesign flow. (4p)


Requirements
Specification
System Design
Software / Hardware Partitioning
Hardware Development and Testing
hardware / software
integration & testing
Software Development and Testing
System Integration and Testing

5f) What are the trade-offs to consider when performing functional partition on a SoC system? (2p)

5. Trade-off parameters:

Time
Cost – chip area
Power consumption

Optional:

Reliability
Configurability

6 – Formal verification (20 p)

6a) (6 p) Describe the role of invariants in Interval Property Checking (IPC). What characterizes a "good" invariant and what a "bad" invariant when proving a property? Will the property always fail if the chosen invariant covers unreachable states?

Invariants are state sets that are closed under reachability: Any (forward) transition from a state of the invariant will lead to a next state that is also element of the invariant.
An invariant for IPC typically is a *superset* of the reachable state set, i.e., it also includes unreachable states. They are used to compensate the problem that the interval considered in IPC starts from *any* state. This may lead to "spurious" counterexamples that correspond to behavior that is unreachable in the concrete system (false alarms). By invariants we add some information on the reachable state space to the underlying computational model.
Invariants are always supersets never subsets of the reachable state set. Supersets may cause false alarms but never false correctness proofs.
The reason why invariants (in form of over-approximations of the reachable state set) are used is that such invariants may be more easy to compute than the exact reachable state set. The most simple and most "weak" invariant is the set of all states.
A "good" invariant is an invariant that is easy to compute but which is still "strong" enough to prove the property. This means that the property must also hold in the unreachable states covered by the invariant. Since a property often does not hold only in the reachable state set but also in some (or even all) unreachable states, this property will not fail for a well-chosen (or computed) invariant.

6b) (8 p) A property checker can produce a *counterexample* or a *witness* for a given property. Please, define these terms precisely.

Counterexample: A trace / waveform / sequence of signal values/assignments that show a behavior where the property does not hold.
Witness: A trace / waveform / sequence of signal values/assignments that show a behavior where the property holds.

Suppose you are running a property on a formal property checker.

I) Is it possible that for the same property, both a counterexample and a witness exist? Explain your answer.

It is possible. For example, the design may have a bug that is observable only in certain situations (counterexamples), and not in others (witnesses).

II) Is it possible that, for a given property, there exists no counterexample? Explain your answer.
III) Is it possible that a property holds on a design but the property checker returns a counterexample?

It is possible. It means the property holds for the design.

6c) (6 p) Without repeating one of the examples from the class, describe in general words the nature of hardware security vulnerabilities by timing-based microarchitectural side channels. What is new in Spectre and Meltdown style attacks?

See question 1: It is possible, if the counterexample is spurious. That means that the counterexample is not reachable from reset. (If the property holds for the design, then no reachable counterexample can be computed, only spurious ones.)