SoC Chapter 7: Formal Verification

Modern SoC Design on Arm
End-of-Chapter Exercises

Q1 What is declarative proof?

Define the following classifications of programming languages and systems: declarative, functional, imperative, behavioural and logic. What class are the following languages: Prolog, SQL, Verilog, C++, Spec- man Elite, PSL and LISP?

Answer:

A declaration is a statement that holds for all time. Mathematical proofs are necessarily such declarations. The last two sentences and this sentence are declarations! “Today is Tuesday” is a declarative assertion, but it does not hold for all time! Hence, formal correctness is a matter of proving that declarative assertions universally hold. We can then make strong statements about the way a system or circuit will behave.

A computer scientist will know that a (pure) functional program makes no changes to the values of variables and iteration is achieved using tail recursion. A good example is the pure subset of LISP.

Imperative programming is widely used. Variables have their values changed as a program executes. Mainstream examples are Verilog and C++

In logic programming languages such as Prolog, or the query sub-language of SQL, the system searches for all possible satisfying combinations of parameters. No variables are mutated. Instead, the answer is a declaration of the form ‘the user’s query is satisfied with each of the following values’ where the values are generally tuples.

Specman Elite and PSL are both declarative.

Q2 Assertions over an RTL design.

The synchronous subsystem in Figure 1 has three inputs: clock, reset and start. It has one output called Q. It must generate two output pulses for each zero-to-one transition of the start input (unless it is already generating pulses). Give an RTL implementation of the component. Write a formal specification for it using PSL or SVA. Speculate whether your RTL implementation could have been synthesized from your formal specification.

Answer:

```
module MOD(input clk, input reset, input start, output Q);
  reg oldstart; reg[1:0] counter;
  always @(posedge clk) if (reset) begin
     oldstart <= 0; counter <= 0; end
  else begin
     if (start && !oldstart && counter == 0) counter <= 1
     else if (counter > 0) counter <= counter + 1;
```

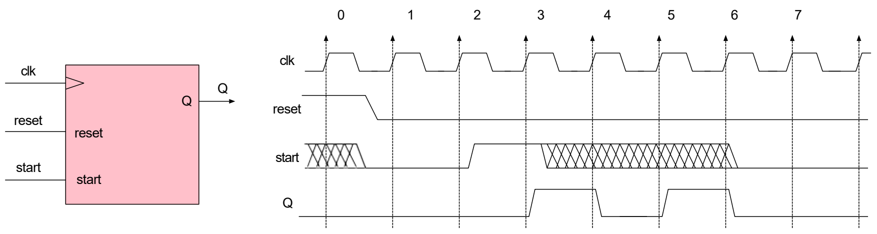


Figure 1: A pulse generator: schematic symbol and timing waveforms.

```
        oldstart <= start;
        end
    assign Q = (counter == 1) || (counter == 3);
    endmodule
```

Here is a suitable PSL (Verilog sugaring) specification of the core functionality.

```
assert always {!start; start} |=> { Q; !Q; Q; } @(posedge clk);
```

Note that the PSL specification above says nothing about the behaviour of output `Q` when the component is not generating pulses. Although we might expect the output to remain stable at this time, it has not been constrained to return to zero (as it does in the question and the RTL implementation) and it might just sit at a logic one!

The example stimulus does not specify what values `start` has in cycles 4 and 5, (or 6). If `start` were low in 4 and high in 5 then another start edge would be present. The verbal spec and implemented RTL ignore that, but the above PSL specification considers it a requirement on the output in periods 6, 7, and 8. Indeed, the specification is invalid in the general case since it places contradictory requirements on a net. Hence the antecedent (l.h.s of the |=> operator) needs further guarding to ignore those transitions, such as

```
event trigger = {!start; start};
assert always (trigger && !{1[1:3];trigger}) |=> { Q; !Q; Q; } @(posedge clk);
```

Speculate whether your RTL implementation could have been synthesized from your formal specification. Tools like FoCs from IBM can compile a large subset of PSL into state machines. Likewise, for RTL simulators that support temporal logic, the path and sequence expressions present in the assertions are most likely compiled into broadly the same intermediate code as used for RTL modelling.

Q3 Model checking a FIFO

Create a formal glue shim like the one in Figure 2 to check the correctness of a FIFO component.

Answer:

This stable oracle style of shim/harness uses a floating input to nominally select one word where its transit through the system is checked, but since the input is undetermined, the system

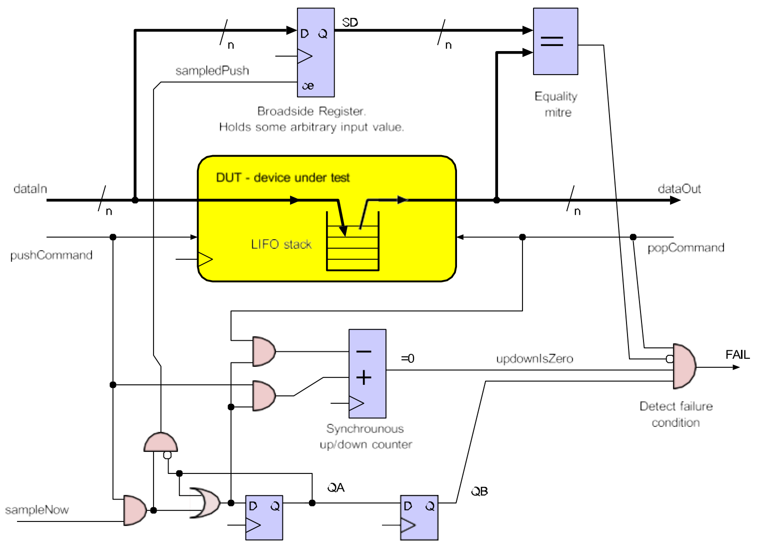


Figure 2: A harness around a data path component (a LIFO stack).

must work correctly for every word.

The difference between a FIFO and a LIFO just regards the order in which words are clock in and out. For the LIFO the word entered at a given fill depth must be emitted when the device is being read at that depth. For FIFO, a word is emitted after all of those present in the FIFO at entry time have been dequeued.

The harness diagram must retain its current occupancy counter but is (nominally) augmented with a second counter that has a parallel load of the occupancy at the time the word of interest is entered in SD: i.e. it is load enabled with the samplePush net. It must then count down on any dequeue cycle, which is when QA && popCommand holds. The equality mitre must be checked when this new counter has the value zero.

A neater implementation is possible, based on the realisation that only one of these two counters is live at once. (In dataflow analysis, a register is 'live' if its current value can affect the output.) The optimisation is to stick with a single counter and simply suppress its increments after the word of interest has been encountered.

This enables symbolic verification using a basic model checker. All inputs on the left are unconstrained, as is the pop input on the right. No stimulus pattern should make the FAIL output hold.

Q4 Model checking a RAM

Create a similar formal proof of the correctness of a RAM, showing that writes to different locations do not interfere with each other.

Answer:

Again, a shim/harness will be created in RTL and put around the RAM under test and the RAM will be subject to an unconstrained sequence of random reads and writes. One subtlety is that data that is read out from uninitialised locations must not be checked since it is allowed to be anything. This is easy to handle by nominally selecting a given write and to check that the next read to that location gives the same value back if it has not been written since.

Rather than drawing schematics, we'll answer using RTL.

This can be our SRAM

```
module SSRAM32x256(input clk, input wen, input[7:0] addr, input [31:0] wdata, output reg [31:0] rdata);
   reg [31:0] data [255:0];
   always @(posedge clk)
       if (wen) data[addr] <= wdata;
       else rdata <= data[addr];
   endmodule
```

Here is the shim for model checking:

```
module TEST(input clk, input wen, input[7:0] addr, input [31:0] wdata, input sampleNow, output reg FAIL);
   wire [31:0] rdata;
   SSRAM32x256 ram_dut (clk, wen, addr, wdata, rdata);
   reg QA, QB;
   always @(posedge clk)
       if (reset) begin QA <= 0; QB <= 0; FAIL <= 0; end
       else begin
           if (!QA && sampleNow && wen) begin
               QA <= 1;
               saved_data <= wdata;
               saved_addr  <= addr;
               end

           if (QA && wen && addr == saved_addr) begin
               saved_data <= wdata;
               end

           QB <= (QA && !wen && addr == saved_addr);
           if (QB) begin
               FAIL <= (rdata != saved_data);
               end
           end
   endmodule
```

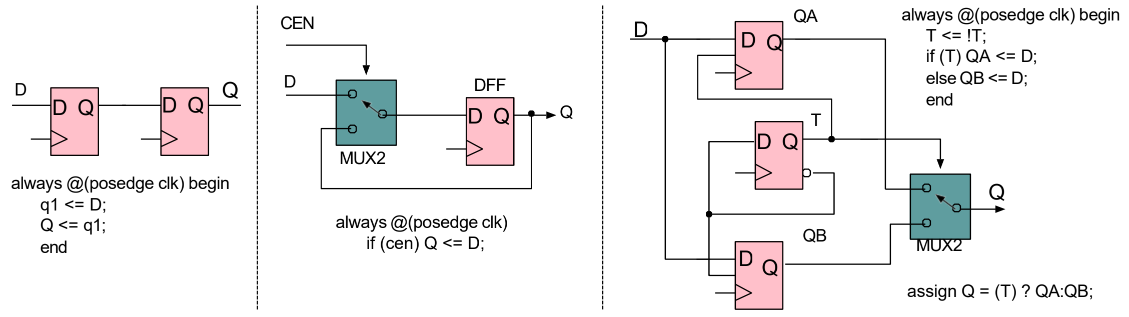
The address of the first write that happens when QA is zero and `sampleNow` is asserted is captured. The data stored is captured at every write to that address. After each read of the address, the saved data is compared. This is done one cycle later owing to the synchronous nature of the SRAM.

Q5 Sequential Equivalence Checking

Prove the equivalence of the two designs in Figure 3 by naming each state in each design and defining a minimal FSM whose states are each labelled with the list of states in each input design that they model.

Answer:

The left-hand machine has four states that we shall name LA through LD. The right-hand machine has 8 states that we will call R0 to R7.



```
always @(posedge clk) begin
    q1 <= D;
    Q <= q1;
    end
```

```
always @(posedge clk)
    if (cen) Q <= D;
```

```
always @(posedge clk) begin
    T <= !T;
    if (T) QA <= D;
    else QB <= D;
    end
```

```
assign Q = (T) ? QA:QB;
```

Figure 3: A two-bit shift register (left) with a conventional design. By using a clock-enabled flip-flop (centre), an alternative implementation is possible (right). The state encoding is totally different, but the observable black-box behaviour is identical.

We have no Mealy outputs and one Moore output. Using a standard bisimulation methodology, the first step is to partition the states where the observable outputs are identical. This gives us P0 and P1.

Q1	Q	Name
0	0	LA
0	1	LB
1	0	LC
1	1	LD

T	QA	QB	Q	NAME
0	0	0	0	R0
0	0	1	1	R1
0	1	0	0	R2
0	1	1	1	R3
1	0	0	0	R4
1	0	1	0	R5
1	1	0	1	R6
1	1	1	1	R7

```
P0: States where Q is 0
   [ LA LC  R0 R2 R4 R5 ]

P1: States where Q is 1
   [ LB LD  R1 R3 R6 R7 ]
```

A systematic process is then to find any partition whose members' successor partitions are not the same for any given input. If one is found, the partition is split into further partitions (two for a binary input) according to the value of that input and the search starts again. If no divergence is found, the construction is complete. If every member of the final partitioning has at least one state from each machine, the machines are equivalent. [NB: There are many other algorithms and it can also be done by inspection for simple problems.]

So, we tabulate the successor states for each member of P0 as follows:

P0 member	Successor with Din=0	Current partition of successor
LA	LA	P0
LC	LB	P1
R0	R4	P0
R2	R6	P1
R4	R0	P0
R5	R1	P1

Since there are two values in the current partition of the successor column for a fixed input, P0 needs to be split into a modified P0 and a fresh P2: P0=[LA R0 R4], P2=[LC R2 R5] and the analysis repeated. This will refine P1 and create a fresh P3: P1=[LB R1 R6], P3=[LD R7 R3] and the analysis repeated. No further partition sub-division is needed for any value of the input values (i.e. the two values of the one binary input) and so the partition is completed. Each partition holds at least one state from each machine, so the two systems have black-box sequential equivalence. They are said to bisimulate each other. QED.

The left-hand input machine had no redundant state and hence the minimal FSM is identical in size to it. In larger examples, both input machines might have redundancies.

Q6 Dynamic Validation using Formal VIP

In the book it says 'Implement the checker described in the `bus-checker` folder of the additional material' and that content is pasted below on this exercise sheet.

1a: Design at the gate-level an arbiter for three customers and a single resource and say what basic type of arbiter it is. (You do not need to include details of the resource or customers.)

Answer:
We shall use the suggested RTL. The missing line has been added as follows:

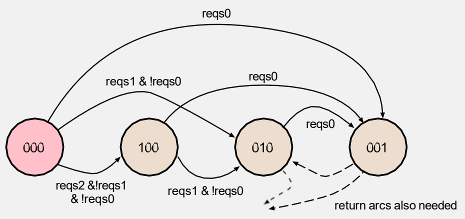
```
   module arbiter(input clk, input reset, input [2:0] reqs, output reg [2:0] grants);
     always @(posedge clk) if (reset) grants <= 0;
 else begin
    grants[0] <= reqs[0]; // Highest static priority
    grants[1] <= reqs[1] && !reqs[0]; // Missing line now added
    grants[2] <= reqs[2] && !(reqs[0] || reqs[1]);
 end
```

This is a synchronous arbiter with static priority and preemption. At the gate level it consists of three D-types, two AND gates and a few inverters.

1b: Each customer will interact with the arbiter using a protocol, typically using one request and one grant signal. Give a formal specification of this protocol using a state transition diagram. For a synchronous protocol, explain how the concept of the clock is embodied in the diagram.

Answer:

Although there are three flip-flops, there are only four reachable states. This state transition diagram follows one arc on each active clock edge. On any clock edge, the system preempts anything currently holding the resource and transitions to the highest priority (lowest numerically) current requestor. The diagram below should have a similar number of reverse edges that are not shown for clarity.



[Step 2 - Simple protocol safety checking using hardware monitors.]

2a: (easy) Give a completely separate RTL or gate-level design that is a monitor (or checker) for the following safety property: 'At all times, never is the resource granted to more than one requester'. This checker should be a component (e.g., separate RTL module) that has as many inputs as is needed to monitor the necessary nets in the system (e.g., all connections to the arbiter) and an output that is asserted in any state where the assertion is violated.

Answer:

```
module monitor_1_vip(input clk, input [2:0] grants, output reg bad);
    always @(posedge clk) begin
        bad = (grants != 3'b000 && grants != 3'b001 && grants != 3'b010 && grants != 3'b100);
        if (bad) $display("%m: %1t: Multiple grants at once.", $time);
        end
endmodule
```

Alternatively, the `$onehot0()` function could be used — much more concise!

2b: (harder) Similarly, design an RTL or gate-level protocol checker that could be instantiated for each connection between a customer and the arbiter that checks each instance of the request/grant protocol is being properly followed. Do you have, or can you envision, a request/grant hardware protocol that has no illegal behaviours? What is allowed to happen in your system if a customer wants to give up waiting for the resource (known as 'baulking')?

Answer:

Here is a minimal per-client monitor suitable for the simple arbiter semantics we are using in this example: it simply checks that grant is deasserted if request was not active the cycle before.

```
module monitor_2_vip(input clk, input req, input grant, output reg bad);
    reg last_req;
    always @(posedge clk) begin
        last_req <= req;
        bad = (grant && !last_req); // The "checked predicate". // Notice blocking assign.
        if (bad) $display("%m: %1t: Grant without request.", $time);
        end
endmodule
```

An arbiter that made a grant when no request had been received might be behaving illegally depending on what's allowed. On the other hand, systems which assert grant to a preferred customer in case they need to use it rapidly exist. The request signal for that customer might then be better considered as a 'do not preempt me' signal.

Depending on the type of arbiter in use, the per-client monitor could check a number of properties. For instance, if the arbiter was not allowed to preempt, it could check that grant is not deasserted until request is deasserted. Checking for fairness/liveness would also be nice but is not practical unless there hard real-time specification exists for the maximum waiting time; otherwise, the checker would not know when to declare a liveness failure.

The simple arbiter presented above intrinsically supports baulking: if a client de-asserts request, without a grant having been issued, there are no consequences. Other protocols could infer a request from a single clock cycle request pulse, which would remain outstanding until finally granted: then there is no protocol for baulking.

2c: For your particular request/grant protocol design, if you extended 2a to also check that no grant is issued without a request would this be a state or path property checker?

Answer:

This is precisely what we did in the previous answer. If `last_req` were to be viewed as part of the system state, then the checked predicate is a simple predicate over the current state of the system: a state predicate. But that variable was added as part of the checker and might not exist in the system under test. Any path checker must keep a track of the 'path so far' as part of its own behaviour, so this checker is a path checker.

Step 3 - Liveness Checking Machine

3. Give a completely separate RTL or gate-level design that is a monitor/checker of the following liveness property: "Whenever reset is not asserted, when a request is made for the resource, it will eventually be granted".

Answer:

Liveness cannot be checked very thoroughly under dynamic validation. All we can do is follow the advice in §7.5 of the book: 'The monitor can indicate whether it has been tested at least once and also whether there is a pending antecedent that is yet to be satisfied.'

```
module monitor_2_vip(input clk, input reset, input req, input grant, output reg ok_once, output reg currently_unsatisfied);
    reg pending;
    always @(posedge clk) if (reset) begin
        ok_once <= 0; currently_unsatisfied<= 0; pending = 0;
        end
    else begin
        if (req) begin
             pending = 1; // Notice blocking assign.
             currently_unsatisfied = 1;
             end
        if (pending && grant) begin
             pending = 0; // Notice blocking assign.
             ok_once <= 1;
             currently_unsatisfied = 0;
             end
        end
    endmodule
```

Step 4 - Formal Logic Implementations

4a. Using PSL, SVA or a similar assertion language, give an assertion that checks the safety property of step 2a above.

Answer:

```
assert always (grants == 3'b000 || grants == 3'b001 || grants == 3'b010 || grants == 3'b100) @(posedge clk);
```

4b. Give a similar temporal logic assertion that asserts that the liveness property of step 3 above is never violated.

Answer:
In contrast to the dynamic validation of above, now that formal proof is being used, liveness can be properly asserted and proven. Rather than using the `eventually!` keyword, an implication whose consequent contains an arbitrary delay is more natural:

```
assert always (request |-> (1[*] ; grant)) @(posedge clk);
```