Assignment 1

Notes

The focus of this assignment is to revisit the basics of SystemVerilog design with the assist of formal assertion/property checking. The assignment also demonstrates the power of formal verification using abstract properties and shows how these properties can be used for design optimization.

An implementation of a 4-bit sequential multiplier is provided that performs the computation based on the *shift-and-add* algorithm (Fig. 1). The structure of the implementation is shown in Fig. 2. Note that a basic n-bit *shift-and-add* multiplier would require two n-bit registers for multiplicand and multiplier and one (2n+1)-bit accumulator register for the result (including carry bit). For this assignment, an optimized multiplier is used that reuses the multiplier register to serve as an accumulator for the lower half of the result. As the multiplier bits are shifted out, the lower bits of the accumulator are shifted into this register. As a result, a (n+1)-bit accumulator register is sufficient to store the result of the multiplication since the lower result bits are stored in the multiplier register.

After the reset, the design asserts **ready** signal indicating readiness for new computation. The computation starts when **start** signal is asserted by the environment. The following assertion of **ready** signal indicates that the computation is finished and the product is available. In this assignment, we mainly focus on the control path of the multiplier marked with red in Fig. 2. The *Control Flow Graph (CFG)* of the control path logic is shown in Fig. 3. Note that each state indicates which signals are asserted. Only the non-greyed-out signals should be considered for this assignment. An example of a multiplication procedure performed in the given hardware is shown in Fig. 4.

The provided verification suite for the multiplier contains two SystemVerilog Assertion (SVA) properties. The **reset** property verifies the reset behavior and the **compute_product** property verifies the correctness of the computation. This verification suite maintains high abstraction - it does not consider implementation details within the design and allows flexible timing as long as it fulfils the upper-bound.

For this assignment it is enough to know that the provided **compute_product** property requires the design to read **multiplicand** and **multiplier** when the **start** signal is asserted and at some point in the future (within the upper-bound described in the property) produce the correct **product** indicated by the **ready** signal. An optional, more detailed explanation of this property can be found below, if skipped - proceed to Task 1.

The **compute_product** property is written using OneSpin proprietary TiDAL library (Timing Diagram Assertion Library). This library provides an easy and intuitive semantics to describe design behaviors and allows usage of some advanced features like **Timepoints**. The use of the timepoints allows writing **compute_product** property in a way that it only focuses on the two important events:

- Reading data from the inputs when the **start** signal is asserted by the environment.
- Producing the correct result (**product**) when the **ready** signal is asserted.

The timing between these two events is not fixed, the only requirement is that it has to fall into the provided range described by the timepoint **t_ready**:

```
sequence t_ready; await_o(t, 1, ready, 12); endsequence
```

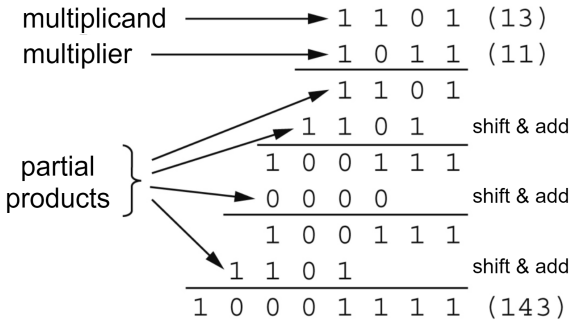

Figure 1: "Add and shift" binary multiplication.

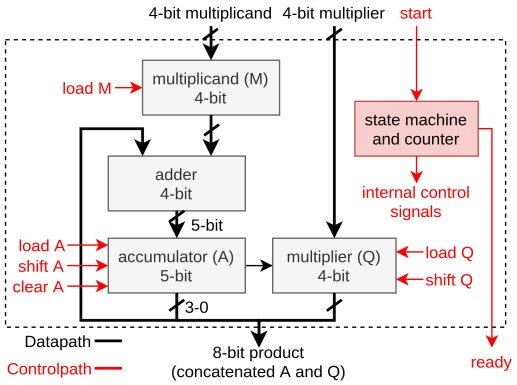

Figure 2: Structure of the sequential multiplier.

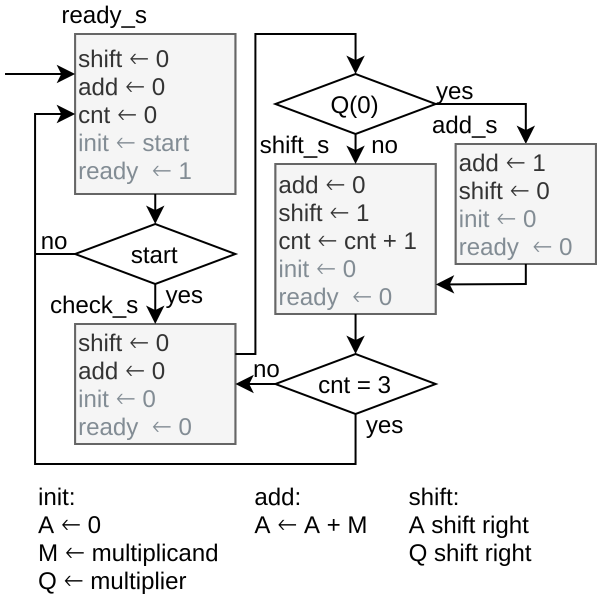

Figure 3: Control flow graph of the multiplier.

M	A	Q	cnt	operation
1101	00000	1011	0	init
	00000	1011	0	check
	+ 1101		0	add
	01101	1011	0	shift
	00110	1101	1	check
	+ 1101		1	add
	10011	1101	1	shift
	01001	1110	2	check
	00100	1111	2	shift
	00100	1111	3	check
	+ 1101		3	add
	10001	1111	3	shift
	01000	1111	0	ready
	product			

Figure 4: Example computation performed in the given hardware.

In other words, this timepoint description within the **compute_product** property requires the **ready** signal to be asserted between the 1st and the 12th clock cycle after the assertion of the **start** signal, otherwise the design is considered to be faulty.

[Working directory: **multiplier**]

Task 1

- Start **onespin**, change your working directory to **multiplier** and load all SystemVerilog files (*.sv) within it. Elaborate and compile the design and then switch to module verification mode.
- Load the verification suite in the file **multiplier.sva** and perform the checks.
- The design contains **two** bugs that can be detected using the given property suite. Analyze the counterexamples that are found by the tool, identify and fix the bugs. **Hint:** the bugs are within the modifiable section (main process starting with `process(clk)`). For debugging purposes the **multiplicand** and **multiplier** operand values can be fixed to 13 and 11 respectively by uncommenting `"and t ##0 define_operands"` line within **compute_product** property

in **multiplier.sva**. **Tip:** after modifying the design you can run **update.tcl** script to update the SystemVerilog files and rerun the verification. To run the script type "`source <path to the script>`".

Task 2

The new requirements for the multiplier state that it has to compute the result in no more than 8 clock cycles. You are asked to optimize the design to comply with the new requirements.

- Change the upper limit within the timepoint definition in **multiplier.sva** to the following:
  `sequence t_ready; await_o(t, 1, ready, 8); endsequence`
- Modify **multiplier.sv** so that it would perform computation in 8 or less clock cycles. Note that there are multiple ways of achieving this. Modifications should stay within the modifiable section in the code. The grayed-out control signals in Fig. 3 are directly derived from design states and should not be modified. **Hint:** it is known that the given register implementation **regn.sv** supports loading (e.g., sum of the addition) and shifting at the same time.
- Verify the optimized design with the modified verification suite.
- Challenge (optional): optimize **multiplier.sv** so that it would compute the result in 4 clock cycles.