ECE 411: Computer Organization and Design

# MP 0: Introduction to SystemVerilog HDL and HVL

Version 0.0.0

# Contents

# 1 Introduction

Welcome to the first ECE 411 Machine Problem! In this MP you will verify and debug designs for several hardware components. Additionally, you will design a hardware component using test-driven design methodology. This document contains the design specifications required to complete these tasks.

The primary objective of these exercises is to introduce you to hardware verification methodologies using SystemVerilog which you will use for the more complicated hardware designs found in follow on MPs. Additionally, you will learn how to verify and/or design the following hardware components:

- An eight-bit multiplier
- A synchronous FIFO
- A content addresable memory (CAM)
- A parallel input, serial output sequencer

The remainder of this section describes some notation that you will encounter throughout this tutorial. Most of this notation should not be new to you; however, it will be worthwhile for you to reacquaint yourself with it before proceeding to the tutorial itself. Section 8 covers the simulation of the design using ModelSim. Section 9 contains the items you will need to submit for a grade. Also included are several appendices that contain additional useful information.

As a final note, *read each and every word of the tutorial* and follow it very carefully. Careful attention to how signal values are assigned and sampled is key to passing the verification assignments. If you do decide to work on the MP before finishing the documentation, make sure to see Sections 8 and 9 for instructions on running the simulation software and submitting your MP for grading, respectively.

## 1.1 Notation

The numbering and notation conventions used in this tutorial are described below:

- Indexing into arrays is based on the dimensions of the array. For example, in `bit [7:0] arr1`, 'bit 0' refers to the *rightmost* bit of `arr1`, while in `bit [0:7] arr2`, 'bit 0' refers to the *leftmost* bit of `arr2`. [1]
- Numbers beginning with `0x` are hexadecimal.
- For RTL descriptions, `pattern[x:y]` identifies a bit field consisting of bits x through y of a larger binary pattern. For example, `X[15:12]` identifies a field consisting of bits 15, 14, 13, and 12 from the value `X`.
- Commands to be typed at the terminal are shown as follows:

  ```
  $  command
  ```

  Do not type the dollar sign; this represents the prompt displayed by the shell (e.g., `[netid@linux-a2 ~]$`).
- Filenames are shown in *italics*.
- SystemVerilog keywords are displayed as `keyword`.
- Signal identifiers are shown in `fixed_width`.
- Signal identifiers are written in "snake case" (lower case with underscores seperating words). Additionally, identifiers will include one or more suffix:
  - Port input signals are suffixed with '`_i`'.
  - Port output signals are suffixed with '`_o`'.

[1] If these notations are new to you, see IEEE Standard 1800-2017 "IEEE Standard for SystemVerilog" section 7.4 "Packed and Unpacked Arrays" [1]

  - Port inout signals are suffixed with '`_io`'.
  - Signals which are synthesized into registers are suffixed with '`_r`'.
  - Signals which are used to wire components within a module are suffixed with '`_w`'.
  - Signals which are 'active low' are suffixed with '`_n`'.
  - SystemVerilog `interface` signals are suffixed appropriately to show the driver of the signal and, if there is only one consumer, the consumer. For example, if the 'Master' port is the driver of a signal, and the 'Slave' port is the consumer, then the signal will use the suffix '`_m_s`'.
  - A `parameter` identifier is suffixed with '`_p`'.
  - A `localparameter` identifier is suffixed with '`_lp`'
  - Signals may concatenate multiple suffixes as necessary.
- All SystemVerilog `function`, `task`, `module`, `interface`, `modport`, and `class` identifiers use either 'snake case' or 'camel case'.
- SystemVerilog macro identifiers are written in `UPPER_CASE` and do not have a prefixed underscore.
- Actions to take in the GUI are shown in **bold**.

## 2 Getting Started

### 2.1 Creating a Github Repository

To create your git repository, goto `https://edu.cs.illinois.edu/create-ghe-repo/ece411-fa19/` to create your repository.

Next, create a directory to contain your MP files and execute the following commands (replacing NETID with your netid):

```
$   git init
$   git remote add origin git@github-dev.cs.illinois.edu:ece411-fa19/NETID
        .git
$   git remote add release git@github-dev.cs.illinois.edu:ece411-fa19/
        _release.git
$   git fetch release
$   git merge --allow-unrelated-histories release/mp0 -m "Merging provided
         MP0 files"
$   git push --set-upstream origin master
```

Alternatively, download the *mp0* directory in the '_release' repository and copy the files manually to your repository.

## 3 What Is Verification

When designing a digital circuit in a hardware descriptor language (HDL), we are attempting to describe a hardware component whose behavior will comply with a high level description of an intended behavior (a specification). Hardware verification is a process which attempts to ensure that a design's behavior matches a specified behavior.

## 3.1 Verification Is Hard

Digital hardware verification is a hard[2] problem. For example, consider the collection of Boolean functions

$$B_n = \left\{ f \;\middle|\; f : \{0,1\}^n \to \{0,1\} \right\}.$$

These are the functions with $n$ binary inputs and a binary output.

How would you go about writing a program which takes as input an element of $B_n$ (the specification), and a SystemVerilog description of a digital circuit (the design), and outputs whether or not the design matches the specification? Can you come up with something significantly better than iterating through all $2^n$ possible function inputs and ensuring that the output of the design matches the output of the specification?[3]

## 3.2 Verification Is Necessary

Since verification is both hard and an integral part of hardware development, it figures that it must be important. Visit EDA Playground[4] and JSFiddle[5] and you'll see that while EDA Playground by default gives as much screenspace to testbench development as design development, JSFiddle gives zero screenspace to debugging.

We have all experienced buggy software where developers clearly prioritized getting a product to market over getting a product to work correctly.[6] What makes hardware so different from most software in that thorough verification is considered necessary in industry?

There are numerous reasons, including the following from Kropf:

- fabrication costs are much higher for hardware than for software;
- hardware bug fixes after delivery to customers are almost impossible;
- quality expectations are usually higher for hardware than for software;
- time to market severely affects potential revenue.

[3, pg.3] In this excerpt, "quality expectatoins are usually higher" often means "human safety is at risk if this hardware device does not work properly".

For you, an ECE 411 student, verification is also important because your GPA will depend on succesfully verifying digital designs.

## 3.3 Verification Is Not Validation

A similar but different process to verification is *validation*. Whereas verification is a process by which we ensure that a design matches **its** specification, validation is a process by which we ensure that a design matches **a** specification.

Consider the case where a truck is designed to meet a specification of being able to haul twenty tons of material. The truck designers at ACME Truck Co. must *verify* that their trucks can haul twenty tons. Likewise, ACE Hauling Co. requires a truck which can haul twenty-two tons. Thus the engineers and technicians at ACE Hauling Co must *validate* that the ACME Truck Co.'s truck can haul twenty-two tons.

[2] coNP-Complete

[3] If you can, please give ECE 411 a shout out as you claim your $1M Prize [2]

[4] A digital hardware design and verification `https://www.edaplayground.com/`

[5] A front-end web development equivalent of EDA Playground `https://www.jsfiddle.net/`

[6] *cough* EDA tools *cough*

## 3.4 How To Do Verification

There are three central tasks to verification[7]:

1. Stimulate a design by providing sequences of stimuli;
2. Check that the design outputs results in accordance with the specification;
3. Measure how much of a design's *execution state space* [8] has been stimulated and checked.

The way that you will complete these three tasks in this MP is by using *dynamic simulation*.[9] In this MP you will use specifications to generate (sometimes random) sequences of input stimuli, create software checkers which confirm that the output of the *design under test* (DUT) conform to the specification, and "scoreboard" DUT accuracy and coverage.

## 3.5 A Simple Verification Example

To demonstrate dynamic simulation we can use the simple example of a purely combinational circuit:[10]

Our task is to verify that `module purefunction`, shown in Listing 1 (the design) actually implements the truth-table its description comment says it does. The truth-table is an example of a specification which describes the intended behavior of the circuit.[11]

Listing 1: A purely combinational design

```
// Module implements the following truth-table:
/*
    abc || x
    000 || 0
    001 || 0
    011 || 1
    010 || 1
    110 || 0
    100 || 1
    101 || 0
    111 || 1
*/
module purefunction
(
  input logic a_i,
  input logic b_i,
  input logic c_i,
  output logic x_o
);
```

[7] [4, pg.23]

[8] The full space of all RTL state and input values.

[9] In *dynamic simulation*, the design is simulated in software using cycle or gate level simulators (e.g. ModelSim), stimuli consist of sequences of input signals to the device under test, and outputs are verified against the specification using assertions. This is in contrast to *formal verification* techniques which use mathematical representations of the design, along with assumptions about possible inputs and states, to constrain the test space to a subset of the execution state space which is actually reachable by the design (and assumptions). In effect, formal verification techniques partition the execution state space into a *reachable space* and *unreachable space*, often drastically reducing the size of the space needed to be tested, and then use automated proofing techniques to prove properties about the circuit.

[10] Although in this case, an eight-to-one MUX may be an appropriate implementation of `module purefunction`, consider a similar circuit but with 20 bits of input rather than 3, implementing a function $f : \{0,1\}^{20} \to \{0,1\}$. In this case, a $2^{20}$-to-one mux is likely unreasonable, and the circuit should be implemented differently.

[11] In this case, the specification is a *formal* specification, as it is written in a formal language with the expressivity of propositional logic. Often an initial specification will not be formalized so nicely.

```
  assign x_o = a_i ^ b_i ^ (a_i & c_i);

endmodule : purefunction
```

In a sense, combinational circuits are the simplest of digital circuits: they have no initial or intermediate state, the size of the input and the output are fixed, and the "runtime" is constant. To verify the design, we can simply[12] run though all possible inputs and verify that the DUT generates the proper outputs:

Listing 2: Generating Stimulus

```
initial begin
  for (int i = 0; i < 4'b1000; ++i) begin
    {a_i, b_i, c_i} = i[2:0];
    #1;[13]
  end
end
```

Now that we've managed to generate all possible inputs, we must create a model of the specified behavior:

Listing 3: Modeling the Proper Behavior

```
function logic spec_output(logic a, logic b, logic c);
    case ({a, b, c})
        3'b000: return 0;
        3'b001: return 0;
        3'b011: return 1;
        3'b010: return 1;
        3'b110: return 0;
        3'b100: return 1;
        3'b101: return 0;
        3'b111: return 1;
        default: $error("Invalid␣input␣to␣spec_output␣function");[14]
    endcase
endfunction
```

Here we directly implement the specified truth table in something which resembles a MUX. In the case of combinational logic with more inputs, we could instead load the truth table into a memory indexed by the inputs as our specification model.

Finally, we can rewrite the for loop which generates the input stimuli to check that the output of the DUT matches the output of the model:

Listing 4: Checking Outputs

```
initial begin
  for (int i = 0; i <= 4'b1000; ++i) begin
      {a_i, b_i, c_i} = i[2:0];
      #1;
      output_equiv:[15]assert (x_o == spec_output(a_i, b_i, c_i))
              else
                $error("{a,b,c}=%b,␣dut␣output:␣%b␣spec␣output:␣%b",
                        {a_i,b_i,c_i},x_o,spec_output(a_i,b_i,c_i));
```

[12]in time exponential to the number of inputs

[13]We must have some type of time delay in order to ensure that each input stimulus actually gets simulated. If there were no time delay, the input stimulus would immediately set to 7

[14]We have this default case since *logic* encodes four-states. Thus if the input to the function is mistakenly **x** or **z**, we can display an error showing our *test bench* is at fault, rather than our *DUT*.

[15]The label *output_equiv:* is used as a name for the *assertion*. This is NOT a label for flow control (in fact, SystemVerilog lacks a *goto* statement).

```
        end
        $finish;
    end
```

Putting this all together, we can write our testbench to verify `module purefunction`:

```
function logic spec_output(logic a, logic b, logic c);
    case ({a, b, c})
        3'b000: return 0;
        3'b001: return 0;
        3'b011: return 1;
        3'b010: return 1;
        3'b110: return 0;
        3'b100: return 1;
        3'b101: return 0;
        3'b111: return 1;
        default: $error("Invalid␣input␣to␣spec_output␣function");
    endcase
endfunction

module purefunction_tb;
    timeunit 1ns;
    timeprecision 1ns;

    logic a_i, b_i, c_i, x_o;

    purefunction dut(.*);

    initial begin
        reset = '1;
        // Generate sequence of inputs
        for (int i = 0; i <= 4'b0111; ++i) begin
            // Set input values to the dut, and let combinational logic
                settle
            {a_i, b_i, c_i} = i[2:0];
            #1;
            reset = '0;
            // Check dut output vs specification output
            output_equiv: assert (x_o == spec_output(a_i, b_i, c_i))
                    else $error("With␣{a,␣b,␣c}=%b,␣dut␣outputs:␣%b␣
                        while␣spec␣outputs:␣%b",
                    {a_i, b_i, c_i}, x_o, spec_output(a_i, b_i, c_i));
        end
        $finish;
    end
endmodule : purefunction_tb
```

Our testbench generates sequences of input stimuli, uses these stimuli to drive the DUT as well as a software model of the specification, and compares the outputs of the two. Further, although we don't explicitly measure it, our knowledge of the test stimuli generated and the execution state space ensures that we have full coverage of the design.

## 3.6 Verifying a Sequential Circuit

When verifying a circuit representation of a Boolean function, we can exhaust all possible inputs simply by iterating through each possible input combination. Consider the case of a sequential circuit, which takes arbitrarily large inputs serially. Clearly verifying the circuit by simply monitoring the input and output ports is insufficient, since the circuit can potentially process infinitely many different input "strings". We consider such an example:

Listing 5: A Sequential Circuit With Binary String Input

```
module div5(
    input logic clk,
    input logic rst,
    input logic serial_in,
    input logic run,
    output logic decision
);

logic [2:0] state;
localparam logic [2:0] initial_state = '1;

always_ff @(posedge clk) begin
    if (rst) begin
        state <= initial_state;
    end
    else if (run) begin
        case (state)
            initial_state,
            3'b000: state <= serial_in ? 3'b001 : 3'b000;
            3'b001: state <= serial_in ? 3'b011 : 3'b010;
            3'b010: state <= serial_in ? 3'b000 : 3'b100;
            3'b011: state <= serial_in ? 3'b010 : 3'b001;
            3'b100: state <= serial_in ? 3'b100 : 3'b011;
        endcase
    end
    else begin
        state <= initial_state;
    end
end

assign decision = state == 3'b000;

endmodule: div5
```

Listing 5 is a SystemVerilog representation of a DFA[16] which "decides" the language "DIV5". If the input string is divisible by five, then on completion of input processing, the output port `decision` should be high. Similarly, if the input string is NOT divisible by five, then on completion of input processing, the output port `decision` should be low.

Since there is no limit on how long input strings can be, if we test the functionality by looking only at inputs and outputs of the design module, then we can only give guarantees qualified by a certain input size (e.g. "all inputs of less than 16-bits produced the proper outputs"). Luckily, we *can* verify whether this design is functionally correct without qualifications. Instead of specifying that the design produces a certain output signal based on the sequence of input signals we instead specify that the design implements a specific DFA, which we prove decides the language DIV5. Thus we must simply verify that the design implements the DFA.

[16]Deterministic Finite Automaton

The DFA that we implement has six states, five of which are labeled 0 through 4 which represent the value of the in-process input string modulus 5. The sixth state is the initial state, labeled $s$. The next state transition function, $\delta$, which takes the current state $i$, and input bit $b$ as follows[17].

$$\delta(i,b) = \begin{cases} (2i+b) \mod 5 & i \in \{0,1,2,3,4\} \\ b & i = s \end{cases}.$$

An input string is divisble by five if and only if the DFA moves to state 0 upon processing the last (least significant) bit in the bit string. We consider this DFA to consume its input string from left-to-right (i.e. the most significant bit first).[18]

Thus to verify the design, we must move the design into every posible state it can enter, and then ensure the transitions from these states are correct.

### 3.6.1 Coverage Points

These "edges" — combinations of internal design state and input signals — are called "coverage points".[19] In this MP, you will be graded on your ability to write testbenches which reach these coverage points and ensure the correct behavior of the design at these points.

## 3.7 Testbench Components

In these prior examples, the verification steps of input stimulus generation, driving the DUT and model, comparing the results of the two are done using only the most basic buliding blocks of SystemVerilog: modules, arithmetic and logical operators, procedural flow-control, immediate assertions, functions, and the timestep delay operator (#). Additionally, it may be useful to seperate functionality of the verification proccess into multiple independent parts:

- A 'sequencer' whose only responsibility is generating input stimuli, independent of the bus or interface used by the DUT.[20]
- A 'driver' which generates the bus or interface control input stimuli and transfers the sequencer's data to the DUT.[21]
- A 'monitor', which acts like a mirrored image of the driver. Just as the driver transactionalizes input stimuli to send to the DUT, the monitor observes and collects the inputs and outputs of the DUT to identify when a transaction is complete and ready to be evaluated by the 'scoreboard'.
- A 'scoreboard' takes the output of the monitor and evaluates whether the DUT produced the appropriate value. In addition to evaluating correctness, the scoreboard can also measure testing coverage.

Performing dynamic simulation of more complicated designs will often suggest using other SystemVerilog language features, such as object-oriented programming, and interprocess communication[22] features . Further, other designs may have far too large of an execution state space to fully cover, and thus explicit cover points must be determined and tested for, while large portions of the execution state space may only be covered if randomized[23] stimulus happen to check those states.

[17] Please humor the abuse of notation which implicitly maps states to integers

[18] See Appendix A for proof of correctness.

[19] or "coverages" or "covers"

[20] Consider two 8-bit adders, one whose data inputs are sent in parallel through a 16 bit port in one clock cycle, and another whose data inputs are sent serially one bit per cycle. Since both have the same functionality — 8 bit adder — they both should be simulated with the same data stimuli (i.e. 3 + 5), while the interface protocol stimuli must be radically different.

[21] Similarly, we can reuse drivers across differing modules as long as those modules share the same bus protocol.

[22] Each 'initial' and 'always' block is treated as an individual process by SystemVerilog simulators. Additionally, the 'fork' ... 'join[_any | _none]?' constructs allow dynamic creation of additional procceses. SystemVerilog's 'mailbox' provides signals and message passing, while 'mutex' provides both blocking and non-blocking mutual exclusion primatives.

[23] ModelSim does support random number generation, but it does not support SystemVerilog's 'rand' modifier, or its constrained randomization features

In the ensuing exercises, you will see designs which you should be able to fully cover as we did for `module purefunction`, and designs whose execution state space is too large to fully cover.

# 4 Verifying a Multiplier

## 4.1 Overview

....

For this exercise, you will write a testbench to verify an unsigned integer add-shift multiplier for use in an 8-bit computer. The multiplier is described in *./multiplier/hdl/multiplier.sv*. In this exercise, you will design a test bench to verify this design.

## 4.2 Specification

The multiplier has the following port listing:

```
module add_shift_multiplier
(
    input logic clk_i,
    input logic reset_n_i,
    input operand_t multiplicand_i,
    input operand_t multiplier_i,
    input logic start_i,
    output logic ready_o,
    output result_t product_o,
    output logic done_o
);
```

- `clk_i` is the clock which drives the sequential logic in the multiplier
- `reset_n_i` is an active low, synchronous reset signal. If this signal is asserted at `@(posedge clk_i)`, the multiplier should halt any ongoing multiplication and reset its state to allow for the start of a new multiplication.
- `multiplicand_i` and `multiplier_i` are the input operands for the multiplication. When a multiplication begins, these signals are registered in the multiplier and thus are not required to be continuously asserted throughout the multiplication.
- `start_i` begins a new multiplication if it is asserted at `@(posedge clk_i)` and the multiplier is in a 'ready' state. If the multiplier is not in a 'ready' state, assertion of this signal has no effect.
- `ready_o` asserts that the multiplier is in a 'ready' state and can begin a new multiplication.
- `product_o` contains the `2 * width_p` bit output of the multiplication when the multiplier is in a 'done' state.
- `ready_o` is asserted when the multiplier is in a 'done' state. This occurs when multiplication is complete, meaning (`product_o` contains the product of the registered input operands OR a synchronous reset has occurred), AND a new multiplication has been started.

See Figure 1 for a timing diagram depiction of this protocol. We do not specify how many cycles the multiplier takes to complete the multiplication.

## 4.3 Coverages

Your testbench must cover at least the following:

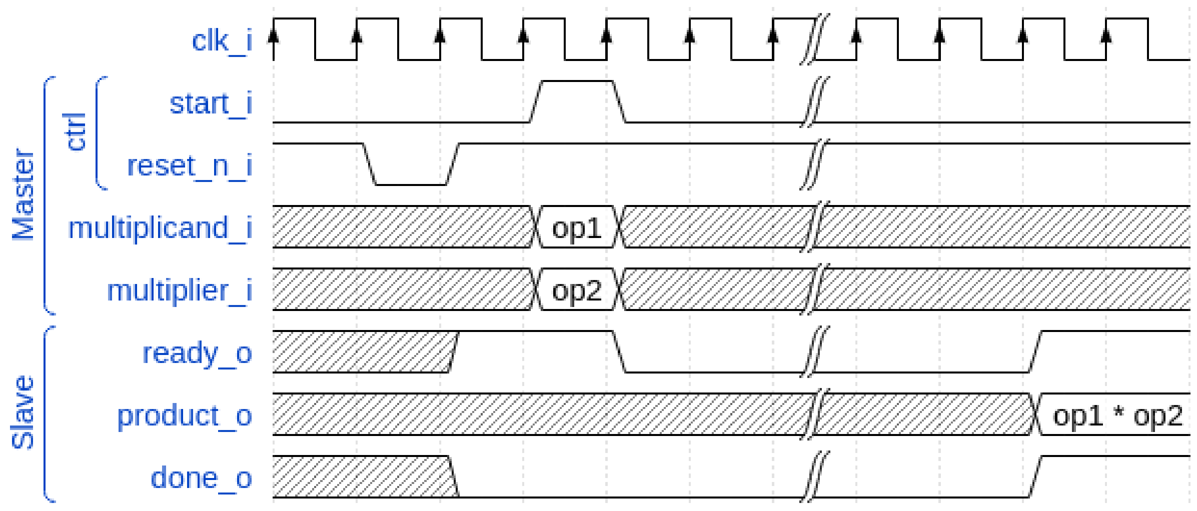

Figure 1: Multiplier Timing Diagram

- From a ‘ready’[24] state, assert `start_i` with every possible combination of multiplicand and multiplier, and without any resets until the multiplier enters a ‘done’ state (resets while the device is in a done state are acceptable);
- For each ‘run’ state *s*, assert the `start_i` signal while the multiplier is in state *s*;
- For each ‘run’ state *s*, assert the active-low `reset_n_i` signal while the multiplier is in state *s*;

## 4.4 Error Reporting

Your testbench must detect the following errors (defined in ‘types.sv’):

- Upon entering the ‘DONE’ state, the output signal `product_o` holds an incorrect product, report a `BAD_RPODUCT` error;
- If the `ready_o` signal is not asserted after a reset, report a `NOT_READY` error;
- If the `ready_o` signal is not asserted upon completion of a multiplication,report a `NOT_READY` error.

To report an error, pass the appropriate error type to `report_error` task defined in ‘testbench.sv’. An example is given in Listing 6.

Listing 6: Reporting a BAD_PRODUCT error

```
assert (/* your assertion here */)
  else begin
    $error ("%0d:␣%0t:␣BAD_PRODUCT␣error␣detected", `__LINE__, $time);
    report_error (BAD_PRODUCT);
  end
```

## 4.5 Clocking Blocks

In SystemVerilog, `clocking` blocks are an abstraction used to capture precise timing information and allow the verification engineer to write verification code at the ‘cycle’ level. The `clocking` blocks allow you to specify input and output skews, but in this MP, they are only used to specify clocks. When using a `default clocking` construct,

[24] see `ready_states` in *multiplierinclude/types.sv*

signals should be assigned using non-blocking assignments. Further, you can insert a delay of *N* cycles using the syntax `##(N)`. To delay until some condition holds, use the 'if and only if' keyword: `@(<clk> iff <conditon>);`.

## 4.6 Driving Signals

In order to facilitate autograding, your testbench should set signal values only at time 0 (the begining of an `initial` block) or using the `tb_clk` clock as described in section 4.5. Additionally, at time 0, your testbench must assert the `reset_n_i` signal.

## 4.7 Sampling Signals

Additionally, all time delaying constructs should be associated with this default clock. That is, they should either be of the form `##(n)`, which waits for *n* cycles with respect to the clocking block, or `@(tb_clk [iff <predicate >])` which delays for a single cycle, or delays until 'predicate' is evaluated true with samples taken with respect to the clocking block. Using the default clocking block in these ways is vital to getting an accurate assessment.

For example, the following are appropriate procedural blocks for your testbench

```
initial reset_n = 0;   // initialize reset signal
initial begin
  ##(5);               // Ensure DUT is reset
  reset_n <= 1;
  multiplicand_i <= 16;
  multiplier_i <= 32;  // NBA: signals still have their initial values
  @(tb_clk);           // Wait for clock signal (could use '##(1)')
                       // Now, when the values get assigned
end

always @() begin
  $display("SystemVerilog␣Functions␣cannot␣block");
end
```

and the following are inappropriate

```
initial begin
  reset_n_i = 1'b1;    // reset not initialized to active low 0
  @(posedge clk);      // Using clk rather than tb_clk
  multiplier_i = 32;   // signal value set at rising edge of clock
end

always @(negedge clk) begin
  reset_n_i = 1'b0;
  multiplicand_i = 16;
  multiplier_i = 32;  // Only use NON Blocking Assignments
                      // with a clocking block
  @(tb_clk);
end
```

# 5 Verifying a FIFO

## 5.1 Overview

For this exercise, you will write a testbench to verify a syncronous FIFO with a single enqueuer and a single dequeuer. A FIFO is called ‘synchronous’ when the enqueue clock and the dequeue clock are the same.[25] The FIFO is described in ./fifo/hdl/fifo.sv. In this exercise, you will design a test bench to verify this design.

## 5.2 Specification

The FIFO implements a valid-ready enqueue protocol, and a valid-yumi dequeue protocol, and has the following port listing:

```
module fifo_synch_1r1w
(
    input logic clk_i,
    input logic reset_n_i,

    // valid-ready input protocol
    input word_t data_i,
    input logic valid_i,
    output logic ready_o,

    // valid-yumi output protocol
    output logic valid_o,
    output word_t data_o,
    input logic yumi_i
);
```

- `clk_i` is the clock which drives the sequential logic in the fifo;
- `reset_n_i` is an active low, synchronous reset signal. If this signal is asserted at `@(posedge clk_i)`, the FIFO sets itself to ‘empty’;
- The valid-ready protocol is:
  - `data_i` contains the enqueued data word;
  - `valid_i` is asserted by the enqueuer to enqueue `data_i` into the FIFO;
  - `ready_o` asserts that the FIFO is not full and has capacity to enqueue a word. The behavior when `valid_i` is asserted while the FIFO is full is undefined and should be avoided.
- The valid-yumi protocol is:
  - `valid_o` asserts that the FIFO is not empty and that the value on `data_o` is the oldest word stored in the FIFO;
  - `yumi_i` is asserted by the dequeuer to signal to the FIFO that the word in `data_o` must be removed from the FIFO

See Figure 2 for a timing diagram depiction of these protocols.

[25] If the clocks are distinct, then it is an *asynchrnous* FIFO, and much more complicated.

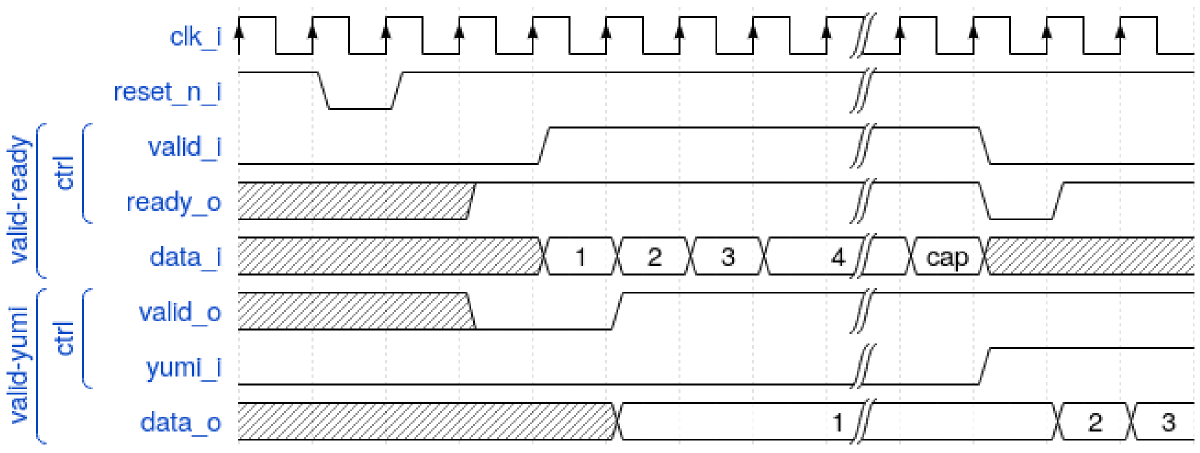

Figure 2: FIFO Timing Diagram

## 5.3 Coverages

Your testbench must cover at least the following for the FIFO with capacity `cap_p`:

- You must enqueue words while the FIFO has size in [0, cap_p-1];
- You must dequeue words while the FIFO has size in [1, cap_p];
- You must simultaneously enqueue and dequeue while the FIFO has size in [1, cap_p-1].

## 5.4 Error Reporting

Your testbench must detect the following errors (defined in *fifo/include/types.sv*):

- Asserting `reset_n_i` at `@(tb_clk)` should result in `ready_o` being high at `@(posedge clk_i)`. If it is not, report the appropriate error.
- When asserting `yumi_i` at `@(tb_clk)` when data is ready, the value on `data_o` is the CORRECT value. If not, report the appropriate error. Recall that asserting `yumi_i` when the FIFO is empty results in undefined behavior, so avoid doing this.

To report an error, pass the appropriate error type to `report_error` task defined in *fifo/hvl/testbench.sv*. An example is given in Listing 7.

Listing 7: Reporting a error

```
assert (/* your assertion here */)
  else begin
    $error ("%0d:␣%0t:␣%s␣error␣detected", `__LINE__, $time, err.name);
    report_error (err);
  end
```

## 5.5 Driving Signals

Once again, only drive signals at time 0 or using non-blocking assignments syncrhonized using the default `tb_clk`. Only sample signals as described in Subsection 4.7.

# 6 Verifying a CAM

## 6.1 Overview

For this exercise, you will write a testbench to verify a content addressable memory, or CAM. A CAM can be thought of as similar to an associative array in computer programming, with the distinction that a CAM is of fixed size. A CAM, then, is a collection of key-value pairs, and supports read and write operations. When reading a CAM, a key is provided, and the CAM responds with the appropriate value, or a signal indicating that their is no value associated with the key in the CAM. On a write, both a key and a value are provided, and these get stored into the CAM.

Since a CAM has a fixed number of entries[26], some type of ‘replacement policy’ must be used when writing to a full CAM. The replacement policy used by the CAM in this MP is the ‘least recently used’ (LRU)[27] policy, which evicts (removes) the entry whose key was least recently used by a read or write.

The CAM is described across several files in *cam/hdl*. In this exercise, you will design a test bench to verify this design.

## 6.2 Specification

The CAM has the following port listing:

```
module cam
(
    input clk_i,
    input reset_n_i,
    input rw_n_i,
    input valid_i,
    input key_t key_i,
    input val_t val_i,
    output val_t val_o,
    output logic valid_o
);
```

- `clk_i` is the clock which drives the sequential logic in the CAM;
- `reset_n_i` is an active low, synchronous reset signal. If this signal is asserted at `@(posedge clk_i)`, the CAM resets itself to ‘empty’;
- `rw_n_i` decides whether the operation is a read (if set to `1'b1`) or a write (if set to `1'b0`). This value has no effect on the CAM unless `valid_i` is asserted;
- `valid_i` is asserted when a read or write operation is performed;
- `key_i` is the key input used by both read and write operations;
- `val_i` is the value input used by write operations;
- `val_o` is the output value on reads;
- `valid_o` is asserted by the CAM on reads to assert that the value in `val_o` is correct (that is, the CAM found a value associated with `key_i`).

Write and read operations are serviced at the rising edge of `clk_i`. That is, the CAM updates its internal state (both key-value pairs as well as LRU metadata sequentially. Additionally, the CAM guarantees that `val_o` and `valid_o` show the correct value on a read at the rising edge of `clk_i`.

[26] Eight, in this MP
[27] https://en.wikipedia.org/wiki/Cache_replacement_policies

### 6.3 Coverages

Your testbench must cover at least the following:

- The CAM must evict a key-value pair from each of its eight indices;
- The CAM must record a “read-hit” from each of its eight indices;
- You must perform writes to the same key on consecutive clock cycles;
- You must perform a write then a read to the same key on consecutive clock cycles.

### 6.4 Error Reporting

Your testbench must detect the following errors

- Assert a read error when the value read from the CAM is incorrect.

To report an error, pass the appropriate error type to `itf.tb_report_dut_error` task defined in *cam/include/-cam_itf.sv*. An example is given in Listing 8.

Listing 8: Reporting a error

```
@(clk);
assert (itf.val_o == val) else  begin
    itf.tb_report_dut_error(READ_ERROR);
    $error("%0t␣TB:␣Read␣%0d,␣expected␣%0d", $time, itf.val_o, val);
end
```

### 6.5 Driving Signals

Once again, only drive signals at time 0 or using non-blocking assignments syncrhonized using the default `tb_clk`. Only sample signals as described in Subsection 4.7.

## 7 Design a Serializer

In this assignment, you will design a Parallel-to-Serial converter which is able to pass the testbench file provided in *piso/hvl/testbench.sv*. A skeleton design file is provided in *piso/hdl/piso.sv*.

The recommended way of approaching this is to start by writing only enough HDL code to pass the first test. Once the first test passes, add HDL code to pass the second test, etc. This portion of the MP is completely self-grading: if your design passes all of the tests in the testbench, it will receive full credit.

## 8 ModelSim

In the base directory of each of the verification assignments, there is a Tcl[28] file named *run.do*. To test your design, from the command line execute the following:

- Load ModelSim in module

```
$ module load altera/18.1-std
```

- run ModelSim from its CLI

```
$ vsim -c
```

- run the Tcl file

```
> do run.do
```

[28] "Tool Command Language"

## 9 Submission

The 'master' branch of your repository is graded nightly. Ensure that any additional files you use are `` `include ``'ed in each testbench. Nightly autograder runs submit results into '_grade' branch in your git repository.

## A DIV5 DFA

Binary strings are defined recursively as either

- The empty string, denoted as $\epsilon$, or
- $s0$ — a binary string, $s$ followed by the symbol '0', or
- $s1$ — a binary string, $s$ followed by the symbol '1'.

We define the length of a binary string $w$, notated as $|w|$ as

- $|w| = 0$ for $w = \epsilon$
- $|w| = 1 + |s|$ for $w = s0$ or $w = s1$.

To prove this DFA is correct, we will actually prove a stronger property: for any non-empty string, $w$, the DFA will halt in state $k$ where $k = w \mod 5$ where $w$ is interpreted as a binary number.

*Proof.* Let $w$ be an arbitrary binary string. Assume, for every string $x$ such that $1 \le |x| < |w|$ that the DFA described above halts in state $k$ where $k = x \mod 5$. There are four cases to consider.

- Suppose $w = 0$. Then the DFA halts in state $0 = 0 \mod 5$.
- Suppose $w = 1$. Then the DFA halts in state $1 = 1 \mod 5$.
- Suppose $w = x0$ for some binary string $x$. Since $|x| < |w|$, by the inductive assumption, the DFA is in state $k = x \mod 5$. Thus, after processing $w = x0$ from left-to-right, the DFA is in state

$$\begin{aligned}\delta(k,0) &= (2k+0) \mod 5\\ &= (2(x \mod 5)) \mod 5\\ &= 2x \mod 5\\ &= w \mod 5.\end{aligned}$$

  Thus the DFA halts in state $w \mod 5$.

- Suppose $w = x1$ for some binary string $x$. Since $|x| < |w|$, by the inductive assumption, the DFA is in state $k = x \mod 5$. Thus, after processing $w = x1$ from left-to-right, the DFA is in state

$$\begin{aligned}\delta(k,0) &= (2k+1) \mod 5\\ &= (2(x \mod 5)+1) \mod 5\\ &= (2x+1) \mod 5\\ &= w \mod 5.\end{aligned}$$

  Thus the DFA halts in state $w \mod 5$.

Therefore the DFA halts in state $w \mod 5$ for any non-empty binary string $w$. □

The above proof combined with the facts that the DFA only accepts in state 0, and that the start state is not accepting prove that the DFA accepts a binary string if and only if the binary number represented by that string is divisible by five.

## References

[1] “Ieee standard for systemverilog–unified hardware design, specification, and verification language”, *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, Feb. 2018. DOI: 10.1109/IEEESTD.2018.8299595.

[2] C. M. Institute. (2019). P vs np problem, [Online]. Available: http://www.claymath.org/millennium-problems/p-vs-np-problem (visited on 06/05/2019).

[3] T. Kropf, *Introduction to Formal Hardware Verification*. Springer, 1999, ISBN: 3-540-65445-3.

[4] E. Seligman, T. Schubert, and K. M V Achutha Kiran, *Formal Verification: An Essential Toolkit for Modern VLSI Design*, 1st. Elsevier, ISBN: 978-0-12-800727-3. [Online]. Available: http://elibrary.nusamandiri.ac.id/ebook/2015_Formal_Verification_An_Essential_Toolkit_for_Modern_VLSI_Design.pdf.