ETH zürich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and Electrical Engineering

VLSI I:
From Architectures to VLSI Circuits and FPGAs

227-0116-00L

Exercise 5

Best Practices of RTL Design in SystemVerilog

Prof. L. Benini
F. Gürkaynak
M. Rogenmoser & M. Bertuletti

Last Changed: 2023-11-13 18:52:59 +0100

1 Introduction

In this exercise, you will learn by experience some of the most common mistakes that can be made writing SystemVerilog code. Learning how to write proper HDL code requires experience. If you end up using System Verilog in your professional career you will develop your own style. Nevertheless, when working in a team it is good to have a common ground. We will present some rules that proved to be useful in our group.

No matter what we will explain here, there is not a universally accepted correct way of writing SystemVerilog code. If you use SystemVerilog for describing hardware, your primary goal would be to achieve the correct functionality. We would generally consider a SystemVerilog code better if it allows you to:

- **Achieve the desired functionality faster**
  You will not get paid extra for writing beautiful looking code. The goal is to get a functional circuit with the least amount of effort possible. Structuring your code properly may help you in this regard.
- **Find eventual problems easier**
  The simple truth is that a designer spends most of their time trying to find faults in the description. Good SystemVerilog code should make it easier to isolate the problems in the code quickly.
- **Synthesize in a shorter time**
  Logic synthesizers sometimes have difficulty interpreting some constructs resulting in long run-times. It is possible to speed up the process by making use of hierarchy.
- **Get smaller circuits**
  Logic synthesizers are not perfect. They sometimes have problems with optimizing high-level constructs. Using lower-level constructs simplifies the task of the synthesizer, and reduces the chances that something gets misinterpreted.

Note that the size of the SystemVerilog code is not one of our criteria. Of course simpler and shorter statements will allow you to type less and make fewer mistakes, however, good SystemVerilog code is not necessarily short.

2 Preparation

To get access to materials required for the exercise please follow the task below:

Student Task 1 (Setup): Setup the working directory for this exercise by calling our install script and changing to the newly created directory:

```
sh> /home/vlsi1/ex5/install.sh
sh> cd ex5
```

3 SystemVerilog Coding made Simple

It is possible to write syntactically correct SystemVerilog code in many different ways. Most students familiar with high-level computer languages like C, will have a tendency to treat SystemVerilog like any other procedural language and will prefer certain constructs. While it is possible to get correct functionality in this way, SystemVerilog offers some more practical constructs for describing hardware.

3.1 Block diagrams

Block Diagrams should help you in every step of your design to have a clear picture of what the code you are writing means. In this task you are asked to implement in SystemVerilog a simple datapath starting from its block diagram.

Student Task 2 (Implementing a Block Diagram): Have a look at the sourcecode template located in `./part1/sourcecode/rgb_grayscaler.sv`. This is a starting point of your implementation, where the module description is already given.

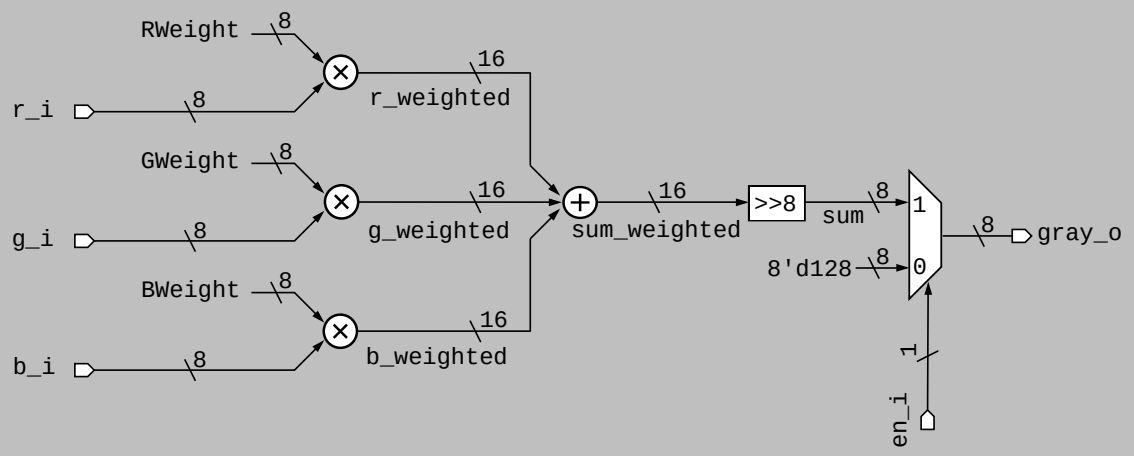

Figure 1: Sample datapath (Grayscaler).

Implement the module described by the block diagram in Figure 1. Compile and test it by running the testbench with the following command in the `./part1` directory:

```
sh>  make -C modelsim runsim TB_DUT=tb_rgb_grayscaler
```

Note 1 (Drawing Block Diagrams): Drawing a good block diagram can simplify writing System-Verilog code a great deal. The block diagram should have names as well as the bit widths for all the signals in the circuit. In fact, anybody who knows the basic syntax of SystemVerilog should have no difficulty translating the given block diagram into HDL code.

It is very important to learn how to draw block diagrams prior to writing SystemVerilog code. The block diagram is also indispensable during debugging.

3.2 Writing code that others can understand

IC design, like many other complex design processes, requires the collaboration of many individuals. It is important to learn to write well-documented, legible, and easy to understand code. Remember that frequently, other people will have to look into your code to enhance it, or fix mistakes. More often than not, you will also have to look at your own code to fix mistakes, and ensuring it is clearly written helps you understand your own work further down the line.

An important part of writing easy to understand code is to adopt a consistent naming scheme. If signals, processes, and entities are always named the same way, any inconsistency can be detected easier. If a design group shares the same naming conventions, all members would immediately *feel at home* with each other's code.

Members of the Integrated Systems Laboratory use the naming conventions suggested by the Microelectronics Design Center, adopted and expanded upon by lowRISC[1]. Note that, as long as conventions are applied consistently, their definition is not so important. However, we strongly encourage to follow this style. Below is a brief excerpt:

[1] The style guide can be found at https://github.com/lowRISC/style-guides/blob/master/VerilogCodingStyle.md

Note 2 (IIS Style Guide): This is an excerpt of the full style guide[a], containing the most relevant information for this course.

- With exceptions of netlist files, each `.sv` or `.v` file should contain only one module, and the name should be associated. For instance, file `foo.sv` should contain only the module `foo`. (This helps a lot with finding the file containing the module you need.)
- If a statement wraps at a block boundary, it must use `begin` and `end`. Only if a whole semicolon-terminated statement fits on a single line can `begin` and `end` be omitted. (This helps a lot with consistent grouping within the code.)
- Indent each new level of hierarchy with **2 spaces**. (This helps with legibility throughout.)
- Naming style: (This helps a lot with differentiating parameters, constants, signals, as well as modules, types, and enums, both when writing code and when investigating traces in the simulator.)

Construct	Style	Example
Declarations (module, package, interface)	`lower_snake_case`	`fifo`
Instance names	`i_lower_snake_case`	`i_fifo`
Signals	`lower_snake_case`	`clk`
Parameters	`UpperCamelCase`	`DataWidth`
Enumeration types	`lower_snake_case_e`	`state_e`
Other typedefs	`lower_snake_case_t`	`word_t`
Enumeration values	`UpperCamelCase`	`Idle`

- Signal suffix: (This helps a lot identifying and differentiating signals and their intended function throughout the design.)
  - Input, output, bi-directional ports shall end with `_i`, `_o` and `_io`
  - Active-low signals have the ending `_n` (`_ni`, `_no` `_nio` in the case of module ports)
  - Register next state and current state end with `_d` and `_q`
  - Pipelined signals end with `_q`, `_q1`, `_q2` etc. where the number implies the cycle latency of the signal
- All clock signals must begin with `clk`.
- Resets are **active-low and asynchronous**. The default name is `rst_n`.
- In `always_comb` blocks, you must use blocking-assignments (=) exclusively, and in `always_ff` or `always_latch` blocks only non-blocking assignments must be used (<=).

  Start your `always_comb` block with a set of default assignments to all the signals you are going to assign to within your conditional statements (**`if`**, **`case`**) **to prevent unintended latches**.

  Keep the logic within `always_ff` simple. Declare two signals for your registers: `somename_d` and `somename_q`. Put the calculation of the next state `_d` in a separate `always_comb` block and stick to the following structure for the sequential process:

```
always_ff @(posedge clk_i, negedge rst_ni) begin
  if (!rst_ni) begin
    state_q   <= Idle;
    counter_q <= '0;
  end else begin
    state_q   <= state_d;
    counter_q <= counter_d;
  end
end
```

[a] https://github.com/lowRISC/style-guides/blob/master/VerilogCodingStyle.md

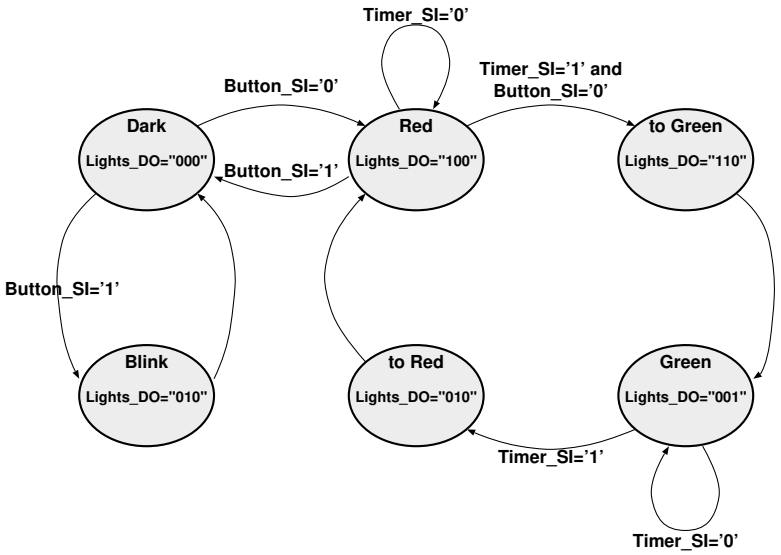

Figure 2: State diagram of a finite state machine, describing a traffic light controller.

In this exercise, you will be given two SystemVerilog codes, both of which implement the finite state machine which describes a traffic light controller illustrated in Figure 2. One of these codes is written with naming conventions and other good practices described in Note 2, while the other is written in a rather crude way. You will be asked to make small changes to both.

Student Task 3 (Modify Good and Bad code):

- Open the well written SystemVerilog code of the traffic light controller, located at **`./part1/sourcecode/traffic_ctrl_good.sv`**.
- Make yourself familiar with the design and ensure that you understand why the code describes the state machine illustrated in Figure 2. How have the states from the state machine in Figure 2 been named within the code?

  Dark = _____ Red = _____ to Green = _____

  Blink = _____ to Red = _____ Green = _____

  What is the state after a reset?_____

  Are there any errors in the implemented state transitions?_____
- Before making changes to the design, take the initial version, compile, and run it. You can launch the simulator using the following command from the **`./part1`** directory:

  ```
  sh>  make -C modelsim runsim TB_DUT=tb_traffic_ctrl
  ```

  You can check if the design works as expected by analyzing the waveforms, and by checking the relevant simulation reports in the command output.
- Now we would like you to make two adjustments to the behavior of the finite state machine:
  1. Change the state machine such that the lights change directly from red to green, without turning orange. Note that when changing from green to red, the lights should still change to orange first.

     Note: For these modifications, stimuli are also provided. Simply adapt the expected responses file name in the parameter of the **`sourcecode/tb/tb_traffic_ctrl.sv`** testbench or add `SOLFILE=red2green_expresp.asc` to the make command for the first change, or `SOLFILE=flash_all_expresp.asc` for the second.

2. Keeping the changes from the previous step, add the following: Currently, when the button is pressed, the center (orange) light starts blinking. Change it so that all lights are flashing at the same time.
3. *[BONUS] In keeping the previous changes, allow any state to jump to the Dark state when the button is pressed.*

- Now have a look at the bad code in **`./part1/sourcecode/traffic_ctrl_bad.sv`**. You can run it with following command:

```
sh>  make -C modelsim runsim TB_DUT=tb_traffic_ctrl DEFINE=+define+\
        USE_BAD
```

  Be sure to configure the initial expected responses in the testbench.

- Familiarize yourself with the design, and attempt to make the same changes as above. **Do NOT spend more than 10min on this part.**
- Do you agree that the good code is better?
- If so, why is it better?

Note that the bad code still has proper indentation, which makes reading the code much easier. However, it is not really feasible to make changes to the bad code. It would probably be faster (and easier) to rewrite the code from scratch.

4 Simple State Machines

In the previous example we adapted a simple state machine, which was clearly described with a corresponding state diagram (at least when working with the good code). In this task, we will look at a slightly larger example, and adapt the design in several steps. The example design is located in **`part1/sourcecode/fsm_counter.sv`** and contains a simple FSM which realizes a counter running from zero to fifteen. A corresponding testbench can be found in **`part1/sourcecode/tb/tb_fsm_counter.sv`**.

Student Task 4 (Counter FSM):

- Familiarize yourself with the design in **`part1/sourcecode/fsm_counter.sv`**. Note that there are several signals we are not using at the moment.
- Compile and run the simulation using the provided make target by running the following command from the **`./part1`** folder:

```
sh>  make -C modelsim runsim TB_DUT=tb_fsm_counter
```

  Ensure your design behaves as expected.

Now we want to adapt this FSM so the counting operation can be interrupted externally. We introduce a new signal `hold_i`. When the `hold_i` signal is set (equal to logic 1), the counter should hold its current value. Feel free to rethink how the FSM should be written.

Student Task 5 (Counter FSM Hold):

- Adapt the design such that, when the `hold_i` signal is set (`=1'b1`), the counter should hold its current value and only continue counting once `hold_i` is unset (`=1'b0`).
- In order to verify whether your changes are correct or not, modify the testbench parameter to reference a new file for its expected responses, provided in the file **`./part1/simvectors/counter_change1_expresp.asc`**, or add `SOLFILE=` to the command with the corresponding file name, as in the previous task. Furthermore, you can investigate the wave forms in the simulator. Run the simulation as above and verify your modification.

In a second step, we would like to adapt the FSM so that it can count both up and down, depending on the value of the signal `direction_i`. This will give the counter more flexibility. Feel free to rethink how the FSM should be written.

Student Task 6 (Counter FSM Direction):

- Adapt the design such that the `direction_i` signal determines the counting direction. If `direction_i` is set (`=1'b1`), the counter should be decremented, otherwise it should be incremented.
- In order to verify whether your changes are correct or not, modify the testbench parameter to reference a new file for its expected responses, provided in the file **`./part1/simvectors/counter_change2_expresp.asc`**. Furthermore, you can investigate the wave forms in the simulator. Run the simulation as above and verify your modification.

As a final step, we want to introduce a signal `init_i`, that will set the counter to 0 in the next clock cycle, regardless of where it is at the moment. Feel free to rethink how the FSM should be written.

Student Task 7 (Counter FSM Init):

- Adapt the design such that, when the `init_i` signal is set (`=1'b1`), the counter is set to 0 in the next clock cycle, regardless of the current values of the other control signals.
- In order to verify whether your changes are correct or not, modify the testbench parameter to reference a new file for its expected responses, provided in the file **`./part1/simvectors/counter_change3_expresp.asc`**. Furthermore, you can investigate the wave forms in the simulator. Run the simulation as above and verify your modification.

Note 3: We sincerely hope that you did not keep the structure of the FSM as provided in the example, but have devised a much simpler way of implementing this circuit. In this case, the problem was that the initial code identified each one of the count values as a separate state. However, the behavior of the FSM did not really differ from one state to another. Each state was doing exactly the same (perhaps the first and last states could be considered slightly special). In other words, the FSM actually only had a single state, i.e. the one of counting.

It is not always trivial to identify the states of an FSM. The initial solution which uses 16 different states simulates correctly, but is especially cumbersome if you want to make modifications to the functionality of the FSM.

A simple way to implement the above mentioned circuits is to define a 4-bit register that will hold the present count value. In this case, the states would be encoded into the count value. The resulting circuit would still be an FSM with 16 states. However, a much simpler description, similar to the one below could be used.

```
module fms_counter (
  input  logic       clk_i,
  input  logic       rst_ni,
  output logic [3:0] count_o
)
  logic [3:0] count_d, count_q;

  assign count_o = count_q;

  assign count_d = count_q + 1;

  always_ff @(posedge clk_i or negedge rst_ni) begin : count_update
    if (!rst_ni) begin
      count_q <= 4'b0000;
    end else begin
      count_q <= count_d;
    end
  end
endmodule
```

Note that this description still has all three parts, output assignment, next state calculation, and the memorizing FF block. We cannot stress this enough: it is a really good idea to keep the combinational part separate from the FF block, as it clarifies for the synthesis tool what section is a flip-flop. The next task should clarify why this is useful.

Student Task 8 (Separating combinational logic from FF):

- To illustrate why it is good practice to separate combinational logic into and `always_comb` block and the flip-flop state update into an `always_ff` block, have a look at the following files: **`./part1/sourcecode/fsm_good.sv`** and **`./part1/sourcecode/fsm_bad.sv`**. Note that while they look similar, they are not meant to be functionally equivalent. Count the number of bits stored in flip-flops for each circuit. The goal is to make a *quick* assumption, do not spend a lot of time trying to decipher the code.

  – `fsm_good`: _____ – `fsm_bad`: _____

- Let us now have a look how the synthesizer implements the two curcuits. Run the following command from the **`./part1`** directory to synthesize both examples:

  ```
  sh> make -C vivado sep_comb_ff
  ```

  Investigate the synthesis report. This can be found in the GUI in the bottom menu at `Reports→ \ Synthesis→ synth_1→ Vivado Synthesis Report`, or in **`./part1/vivado/sep_comb_ff.runs/synth_1/fsm_all.vds`**. How many flip-flops (registers) are found in the design for each component?

  – `fsm_good`: _____ – `fsm_bad`: _____

- Did you get the correct number of memory elements as estimated before?

  _____

Note 4: When you separate the memorizing process, you clearly state which signals you want to have registered. If everything is correct, during synthesis you will see memory elements for only these signals, not more, not less. Register Transfer Level (RTL) design is based on defining what happens between two *registered* states of the circuit that are stored in memory elements. That is why it is important to have complete control over what is stored and what is not stored in the SystemVerilog code.

When you combine the memorizing process with the next state logic, you create a memory element, by *deliberately* not assigning a value in a conditional statement. The problem is that the same happens if you simply make a mistake and *forget* assigning the value. While you would detect missing memory elements during functional simulation, it is possible that superfluous memory elements go unnoticed. But the real problem is that it is more difficult to separate these processes into smaller ones.

You are done with the first part of the exercise.
Please discuss your answers and any open questions with an assistant.

5 Structuring SystemVerilog code

Modern digital circuits are very complex systems which at times can have millions of components. Therefore, successful digital design is also an exercise in managing complexity. The most important tool for managing complexity for digital designs is known as divide and conquer. Larger functional blocks are sub-divided into smaller functional blocks, and the process is repeated until sub-blocks reach a manageable size. This creates a hierarchy of designs in a tree-like structure.

In SystemVerilog, the design hierarchy is established by instantiation. To instantiate a sub-module you have to specify its name and the name of its instance inside the larger design where it is introduced. Port connections have to be specified with a dot followed by the port name inside the sub-module and by the signal that it is connected to this port in round brackets. Dividing your design into several sub-modules (a.k.a. structuring) can serve three different purposes:

1. Create SystemVerilog code that can be reused
2. Simplify SystemVerilog code
3. Help/Accelerate Synthesis

In this part of the exercise we will examinate the advantages and disadvantages of including sub-modules in your design. We will show **how to** do structuring and **how not to** do it.

5.1 Reusable Code

Once you start designing digital circuits, you will realize that many sub-components will be used in slightly different variations over and over again. Some of these would be very small like multiplexers or registers, but others could be quite large like signal processing accelerators, or complete CPUs. If you could write some of your blocks so that they can be adapted for different purposes, your code could be made more reusable. At least this is the theory.

One common problem in reusing components is that at times some parameters may change from design to design. If the sub-component is within a single file, it might still be feasible to make all changes within this one SystemVerilog file. However, if multiple files rely on the same parameters, a more elegant way might be to use packages.

Unfortunately, many designers get carried away with using packages, defining too many constants, subtypes and component instantiations that will be only used in a single SystemVerilog file. While this is not strictly harmful, it makes reading the code more difficult and does not make coding any easier, and thus should be avoided. In this part we will investigate an example that over-uses the package.

Student Task 9 (Overuse of packages):

This example is a simple datapath defined in the file `./part2/sourcecode/shiftreg.sv` that instantiates four similar stages (called stage5, 7, 9, and 11) one after another. The first stage accepts a 5 bit value, and outputs a 7 bit value, and the next stage similarly takes a 7-bit value and outputs a 9-bit value, each stage increasing the data width by 2 bits. The functionality of the design is not really useful, this is just a toy example to illustrate you a problem.

At first, examine the sourcecode of the design. Next, compile and test it by running the testbench with the following command from the `./part2` directory:

```
sh>  make -C modelsim runsim TB_DUT=tb_shiftreg
```

When you are done, answer the following questions:

- Which file uses the definitions of the package **shiftreg_pkg.sv**?

- What is the advantage of having the definitions in a separate file. Are there any disadvantages?

- What is the purpose of the type definitions?

- What would happen if we would not use them?

Rewrite **shiftreg.sv** without using the package **shiftreg_pkg.sv** and taking your answers into account. To check the correctness of your design compile and simulate it.

This is a classical example of overusing the packages. Note that if the contents of the package are only going to be used by one other SystemVerilog code, there is no need to use a separate SystemVerilog package, one could easily add all the definitions to the main SystemVerilog code itself.

Student Task 10 (Reusing modules):

You may have noticed that the four stage variations are essentially the same, but they have different bit-widths. Copy one of the **stageN.sv** files to **stage.sv** and modify it so that it uses a single parameter which defines the bit-width. Later modify **shiftreg.sv** so that it instantiates four times the new **stage.sv** with different parameters. To check the correctness of your design compile and simulate it from the **./part2** directory:

```
sh>  make -C modelsim runsim TB_DUT=tb_shiftreg
```

By using generics it is possible to have one main code for several variations of a component that are essentially the same. In our example, having one source for the stage components has an important advantage: If we need to modify this file (to correct a mistake, or add more functionality), it would be much easier for us to modify one main file, instead of changing four files simultaneously.

5.2 Hazards of parameters

Similar to using packages is it also easy to get carried away when using parameters. Technically you should check that every time you parametrize a model it keeps the same functional specifications for all the possible values of the parameter. This becomes impractical when you have too many parameters.

Student Task 11 (Parameters in synthesis):

In this exercise we will try to apply different values to the parameter of a sample circuit. The design can be found in **./part2/sourcecode/bad_generics.sv**. The initial value of the parameter `Value` is 8. Open an example Vivado project from the **./part2** directory with the following command:

```
sh>  make -C vivado bad_generics
```

Then synthetize the module from the GUI (`Flow/Run_Synthesis`). In the pop-up, do not run the implementation. Simply press cancel. Check the synthesis report, which can be found in the GUI in the bottom menu at `Reports→ Synthesis→ synth_1→ Vivado Synthesis Report`, or in **`./part2/vivado/bad_generics.runs/synth_1/bad_generics.vds`**. Alternatively, you can also check the `Utilization` report. Note down the number of instantiated cells used by the design for different parameters. You can complete the following table by changing the value of the parameter and repeating the synthesis.

- `Value = 8`: Cells = _____
- `Value = 16`: Cells = _____
- `Value = 24`: Cells = _____
- `Value = 32`: Cells = _____
- Which circuit uses the largest number of cells?

  _____
- Why?

  _____

As you can see the difference is huge. You might think that this is a far-fetched example, however we have encountered such problems in actual designs. You might add a parameter and know that the circuit would be meaningful only for specific values of the parameter. However if not properly documented, this knowledge may get lost, and future users of this reusable component may inadvertently use parameter values which the design was never intended for.

5.3 Simplification through structure

One of the most common reasons for using a structured approach is to make it easier to maintain the SystemVerilog code. Maintaining, and navigating overly long SystemVerilog sourcecode files could be challenging. Breaking such long code into several pieces may make it easier to manage the overall project. It is also generally good practice to isolate related sub-functions in separate entities. While it is difficult to describe universal guidelines on how to structure the sourcecode, you should certainly realize some benefit from the division.

Student Task 12 (Using Hierarchy):

Consider the example in **`./part2/sourcecode/lutchain.sv`**. This is a fairly long code that consists of several identical stages each consisting of a small look-up table and an XOR. In this part we will see why this is not a good idea. Consider that you need to make a simple change to the look-up table. *Note*: Do not forget to add your new `lut.sv` to the `file.list` file.

How much effort do you need to spend in order to apply the similar changes to all stages?

_____

It would be much simpler to write a separate entity that contains one stage, and then instantiate this entity multiple times. The current look-up table realizes the values:

`9,0,4,11,13,12,3,15,1,10,2,6,7,5,8,14`

You can compile and simulate this design using the following command from the **`./part2`** directory:

```
sh>  make -C modelsim runsim TB_DUT=tb_lutchain
```

Change the design so that the new look-up table realizes the values:

`3,12,6,13,5,7,1,9,15,2,0,4,11,10,14,8`

To check the correctness of your design compile and simulate it. Don't forget to change the test-bench, so that it sources the correct responses from **`./part2/simvectors/exp_resp_lutchain2.txt`**.

Note 5: Using a hierarchy can sometimes simplify writing SystemVerilog. If you use the same structure over and over again, it is simpler to put the structure in a separate entity and instantiate it when needed. In this way, any changes only need to be applied on a single location within your design.

A very common structural division in SystemVerilog design is to separate the controlling FSM from the datapath. In this example we will investigate such an example. The file **`./part2/sourcecode/simple_fsm.sv`** contains a circuit that instantiates two components described in two separate files called **`datapath.sv`** and **`fsm.sv`**, within the same directory. The exact functionality of the datapath and the FSM is not of importance for this exercise.

Student Task 13 (FSM and Datapath): Proceed as follows in order to make the changes to the sample design:

- Familiarize yourself with the design, and look at the three source files of the design.
- Complete the state diagram given in Figure 3. The state diagram will be quite helpful during making the changes to the design throughout the next steps.

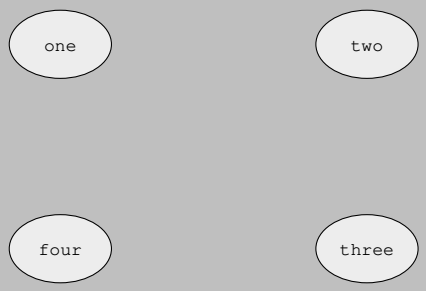

Figure 3: State diagram of the simple FSM.

- Next, compile the design and run the simulation. Verify whether your drawn state diagram matches the behaviour of the design or not. You can compile the design and run the test-bench with the following command from the **`./part2`** directory:

```
sh>  make -C modelsim runsim TB_DUT=tb_simple_fsm
```

- Now, you will need to modify the system such that `accu_d` is set to zero when the FSM is in state `one` (in the initial version of the design, `accu_d` is constantly set to `add_sub`). Make the necessary changes by adding a new control signal called `clr_accu` and verify your result using the same test bench as before. Don't forget to change the test-bench, so that it sources the correct responses from **`./part2/simvectors/fsm_change1_expresp.asc`**.

In order to make this change, you will have to add new ports to both instances, and change the components and instantiations. While you are debugging the system, such changes to FSMs are actually fairly common.

Student Task 14 (FSM and Datapath): You now need to make a second change. When the FSM is in state `Four`, until now, the state moved automatically to state `One`. Introduce a change so that the state moves to state `Two` if the `add_cnst` value is equal to all zeroes. Otherwise it should move back to state `One`. Use the provided expected response file **`./part2/simvectors/fsm_change2_expresp.asc`** in order to check whether your changes have been successful or not.

Although it is difficult to generalize, there is generally not much benefit from separating the FSM from its related datapath. Normally, the datapath and its FSM are tightly coupled which requires many connections between

the two entities. If both the datapath and the FSM are in the same entity, the changes like the ones we have asked you to do, become much easier. You can see for yourself whether or not you agree with this statement in the following task.

Student Task 15 (FSM and Datapath): Open the file **`./part2/sourcecode/simpler_fsm.sv`** which contains the same datapath and FSM within a single entity. Make the same two changes from the previous tasks. A test-bench is available for this design: **`./part2/sourcecode/tb/tb_simpler_fsm.sv`**. You can compile the design and run the simulation from the **`./part2`** directory as follows (For each task remember to change the expected response file in the test-bench or add the `SOLFILE=` variable to the command):

```
sh>  make -C modelsim runsim TB_DUT=tb_simpler_fsm
```

Which approach was simpler? Why?

Note 6: As mentioned earlier, the goal of using a structured approach is to simplify your life. It requires some experience to judge what to separate and what to keep together. However, do not blindly generate additional entities. Technically, these will not be a problem, you can generate as many instances as you want. But if they do not make your life easier, there is really no point in using them.

6 Inferred Latches

Unintended latches occur when a combinational block does not specify the value of an output for all possible input conditions. This scenario can effectively be prevented from happening by assigning a *default* value to all the signals within our block that appear on the left-hand side of an assignment at the beginning of the `always_comb` process. Assignments in an `always_comb` block are executed sequentially. The default assignment is only used in the situations where none of the other assignments are reached.

In general, it is not easy to detect latches in your design prior to synthesis. Therefore, let us try to synthesize a design which may infer a latch.

Student Task 16:

- Run the following command from the **`./part2`** directory. It synthesizes a single design file **`panel_mux_prim.sv`** with vivado.

```
sh>  make -C vivado inferred_latch
```

- In the pop-up, do not run the implementation. Simply press cancel.
- Investigate the synthesis log. This can be found in the `Log` panel in the bottom menu of the GUI, or in the file **`./part2/vivado/inferred_latch.runs/synth_1/runme.log`**.
- Search for the Synthesis Warning **'WARNING: [Synth 8-327] inferring latch for variable...'** in the synthesis log. This will help you understand if there are unintended latches in your design. Alternatively you can open the synthesized design in the left menu in the GUI and use the following TCL command in the bottom `Command` pane in the GUI to show all infered latches in the currently opened design:

```
show_objects -name find_1 [get_cells -hierarchical -filter { \
   PRIMITIVE_TYPE =~ FLOP_LATCH.latch.* } ]
```

- If you observe latches please fix them by assigning an appropriate default statement. Investigate the file **`panel_mux_prim.sv`**.
- After fixing the issue, repeat the synthesis using the `Run Synthesis` in the left pane of the GUI or by rerunning the above `make` command. Make sure the vivado synthesis log no longer warns about any inferred latches. When re-executing the search command, make sure to reload the design to incorporate your fix.

$\mathcal{E}$ **You have reached the end of this exercise.**
Please discuss your answers and any open questions with an assistant. $\mathcal{E}$