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 3

HDL Coding: Sequential Circuits

Prof. L. Benini
F. Gürkaynak
M. Korb

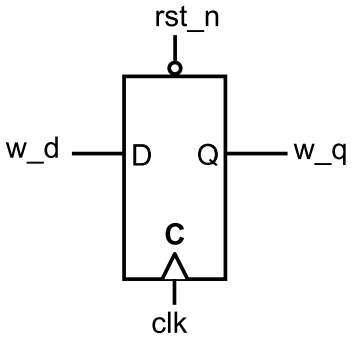
Last Changed: 2023-10-16 17:15:33 +0200

1 Introduction

In this exercise, you will learn the fundamentals of sequential circuits. Sequential circuits are the state-holding elements in a design. By combining them with combinational circuits, which you learned about in the previous exercise, we can build comprehensive digital designs. Sequential circuits can be broadly divided into two categories: *synchronous* and *asynchronous*. Synchronous state-holding elements update their status only during particular windows of time dictated by a global *clock* signal, common ('synchronous') to all storage elements in the design. An asynchronous sequential circuit, on the other hand, changes its state depending only on signals that are not shared globally. This kind of sequential circuit creates problems in terms of stability as you shall learn later during the course, hence our focus in this exercise is on synchronous sequential circuits.

Sequential circuits are further classified as *level-sensitive* or *edge-sensitive*, depending on how the window of time during which the state is updated is defined relative to the clock signal. In a level-sensitive design, updates can be made anytime the clock is sensed as logic high by the state holding elements. In this exercise, we will be dealing with edge-sensitive designs, which restrict the time for updates only when a value transition occurs on the synchronizing clock signal, thereby preventing unwanted updates that may creep in due to a larger update time window.

The fundamental edge-sensitive synchronous sequential circuit element is called a *D flip-flop*, which samples and updates the state at the event of a positive edge of the clock signal. The schematic of a D flip-flop with *synchronous active-low reset* and its corresponding hardware description in SystemVerilog is given in Figure 1. As shown in the figure, a D flip-flop has a clock input pin (`C`), symbolized by the small triangle, indicating that it is sensitive to positive edges of the clock input. The `D` pin samples the input at the positive edge of the `clk` signal applied to `C`. The `Q` pin makes the sampled value available as an output after the positive edge of the clock. Finally, the reset pin, connected to the `rst_n` signal, will reinstate the reset value synchronous to the clock when a low value is applied. The fact that the reset is triggered by a logical low signal is shown by the inverter circle at the reset pin.




```
logic w_d, w_q;
always_ff @(posedge clk) begin
    if(~rst_n) begin
        w_q <= 'd0;
    end else begin
        w_q <= w_d;
    end
end
```

Figure 1: D flip-flop. Schematic and code snippet

On the other hand, the fundamental level-sensitive synchronous sequential circuit element is the *latch*. As already mentioned, its output follows combinatorially the input while the clock is asserted, whereas it is kept stable when the clock is not asserted. We will not use this kind of device here.

Having covered the fundamentals of sequential design, let's now work on several example use cases where sequential circuits become useful. First, as a warm up, you will build a shift register design out of D flip-flops. Afterwards, the fun begins - for the remaining part of this exercise you will be working with some high-tech video processing applications to help out a forensic expert to analyze an obscure picture! In the process, you will (re-)learn sequential circuit design principles such as finite state machines (FSMs), digital edge detectors and digital counters.

The basics you learned about video processing principles and techniques in the previous exercise come in handy when you attempt later assignments in this exercise. It is recommended to follow our *SystemVerilog coding guidelines*[1] for the parts you are expected to write codes for the hardware descriptions in SystemVerilog. It is also recommended to create your own script file and add commands to automate Vivado project creation and compilation process.

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

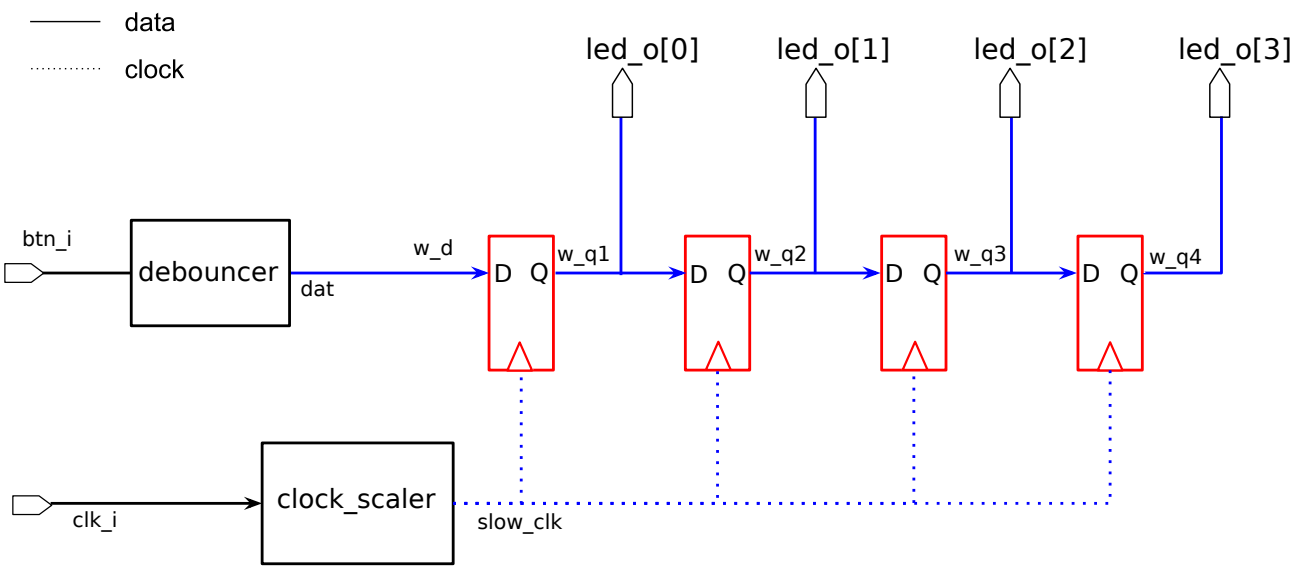


Figure 2: Top level block diagram of the assignment 1

Please take a moment to discuss your answers with one of the assistants every now and then.

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/ex3/install.sh
sh> cd ex3
```

3 Implementing a shift register

The goal of assignment 1 is to implement a shift register.

To complete this assignment, we will use the D flip-flop description given in Figure 1.

The schematic of the top module of the shift register is shown in Figure 2. It has four instances of D flip-flops connected in series. In addition to the system clock `clk_i`, it has one user input `btn_i` for the upstream data input of the first D flip-flop.

The top module also has peripheral modules which most probably have been used in previous exercises. The peripheral modules include:

1. A debounce circuit (`debouncer`), used to prevent unintended glitches from the user when using the push button to send inputs.
2. A clock converter circuit (`clock_prescaler`), to slow down the system clock by a sufficient factor before applying to the flip-flops. This is done so that we can actually watch the transitions on the outputs connected to the LEDs.

Student Task 2 (Project Setup):

1. Open a terminal in **`1_shiftreg/vivado`** directory and run the following command:

```
sh> make gui
```

2. Select the `Sources` window and check whether the following files are listed under `Design Sources`: **`zybo_top.sv`**, **`debouncer.sv`**, **`rst_gen.sv`**, and **`clock_prescaler.sv`** (note that the top level file is a dropdown).
   Otherwise add them from **`1_shiftreg/sourcecode`** directory.
3. Make sure **`tb.sv`** is set as top level of 'Simulation Sources'. Otherwise add **`tb.sv`** as a simulation source from **`1_shiftreg/sourcecode/tb`**, then right click on the file in "Sources" window under 'Simulation Sources' and click "Set as Top".
4. Make sure **`zybo-z7.xdc`** is listed under `Constraints` in the `Sources` window. Otherwise add it to the project from **`vivado/constraints`** directory.

Student Task 3 (Completing the Top-Level Design):

1. Open the top module SystemVerilog file **`zybo_top.sv`** in the text editor of your choice.
2. Declare the signals for the flip-flops (follow the names shown in Figure 2 when labeling the signals).
3. Define the behavior of the flip-flops with a `always_ff` block as shown in Figure 1. Trigger them on `slow_clk`.
4. Make the remaining missing signal assignments to the submodule IOs and primary IOs.

Now the design is ready for testing. We will first test it in the Vivado simulator.

Student Task 4 (Simulating the Design):

1. Click 'Run Simulation' on the left pane of the Vivado window. Then 'Run Behavioral Simulation'.
2. If your solution to the above task is producing incorrect outputs, the simulation will end with `output mismatch` message in the *Tcl Console*
3. If you wish to re-run the simulation, type the following in the Tcl console in the `Simulation` window: `restart` followed by `run -a`. Remember you can add signals to the wave window to help you debug.
4. If you encounter errors, revisit your solution or seek assistance from an instructor.

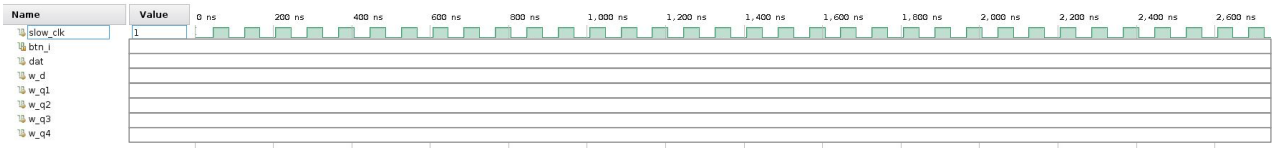


Figure 3: Simulation waveform (to be filled)

Student Task 5 (Investigating the simulation behavior):

1. If the simulation passes, check the output waves in the wave window. If you didn't add signals to the wave window you can do this now and re-run the simulation by entering `restart` and `run -a` in the Tcl console.
2. Plot the signals listed in Figure 3 from $200\,\mathrm{ns}$ to $2600\,\mathrm{ns}$. If they are not present in the `Waveform` window, please select the uut design to show the right signals in the `Object` window. Select them, right click and then choose 'Add to Wave Window'.
3. Does `w_d`, `w_q1`, `w_q2`, `w_q3`, `btn_i`, and `dat` behave as you expected?

   ---

4. Write down a boolean function to generate a single cycle pulse for each rising edge of `w_d`, i.e. when the signal pass from a LOW state to a HIGH state. Use the shift register signals of Figure 2.

Now it is time to test the design on the FPGA board. We have attached the push button `BTN0` of the board to the signal `btn_i`. The outputs of the flip-flops are attached to LEDs `LD0` to `LD3` on the board. These constraints are defined in **`zybo-z7.xdc`**.

Student Task 6 (Programming the FPGA):

1. Click on `Generate Bitstream` on the left side pane in Vivado and initiate the bitstream generation process. Note: If the display looks similar to the extended mode, change the display settings to single screen.
2. Once the bitstream is ready, open the `Hardware Manager`, click `Open Target`, followed by `Auto-Connect` and program the FPGA using the bitstream just created.

As previously mentioned, we have programmed `BTN0` on the board to be connected to the input signal `btn_i` which in turn goes through a debouncer. The output of the debouncer, called `dat` goes to the input of the Flip-flop chain. Refer to Figure 2.

Student Task 7 (On-board testing):

1. Manipulate `BTN0` on the board to observe the behavior of the outputs of the register chain.
2. Can you estimate the frequency of the output clock of `clock_prescaler` module using the FPGA setup?

   _____

3. Describe the method you used to estimate the above frequency

   _____

   _____

4 Implement a finite state machine to control a video filter chain

Your remaining assignments are about helping a forensic expert to analyze the content of an obscure picture, which is known to contain vital evidence related to a case he is currently working on. The picture he sent to you is available in the following location:
`2_fsm/resources/secret.jpg`
He believes that the picture can be more clearly identified by applying image processing filters on the picture. The forensic expert has supplied you with the SystemVerilog descriptions of the above filters for your convenience. You are assigned the task of properly implementing and controlling these filters on the provided FPGA hardware to see if you can reveal the picture's contents.

From the last exercise (Section 3, Video Processing) you are already familiar with the basic concepts of image processing such as the concept of a *pixel* and its representation using three 8-bit values for each of the **R**ed, **G**reen and **B**lue color channels. In this exercise, we make use of the already developed infrastructure which will stream a continuous sequence of pixels from the computer through the FPGA to the monitor. The pixels are streamed in frame order, meaning that first, all the pixels from the first frame are sent, then the pixels from the second frame, etc. Within a frame, the stream of pixels follows a *raster scan* order. This means the pixels from the topmost row are streamed first from left to right followed by the second row etc.

Our forensic expert wants you to try out different combinations of four kinds of filters which he suspects could be most helpful in the identification process. By applying two or more filters back to back it is possible to have an effect different to when filters applied individually.

Now you are expected to write a *finite state machine* (FSM) to drive the control signals that enable a certain set of filters at a given time in the 2nd assignment. A high level block diagram of the task is given in Figure Figure 4.

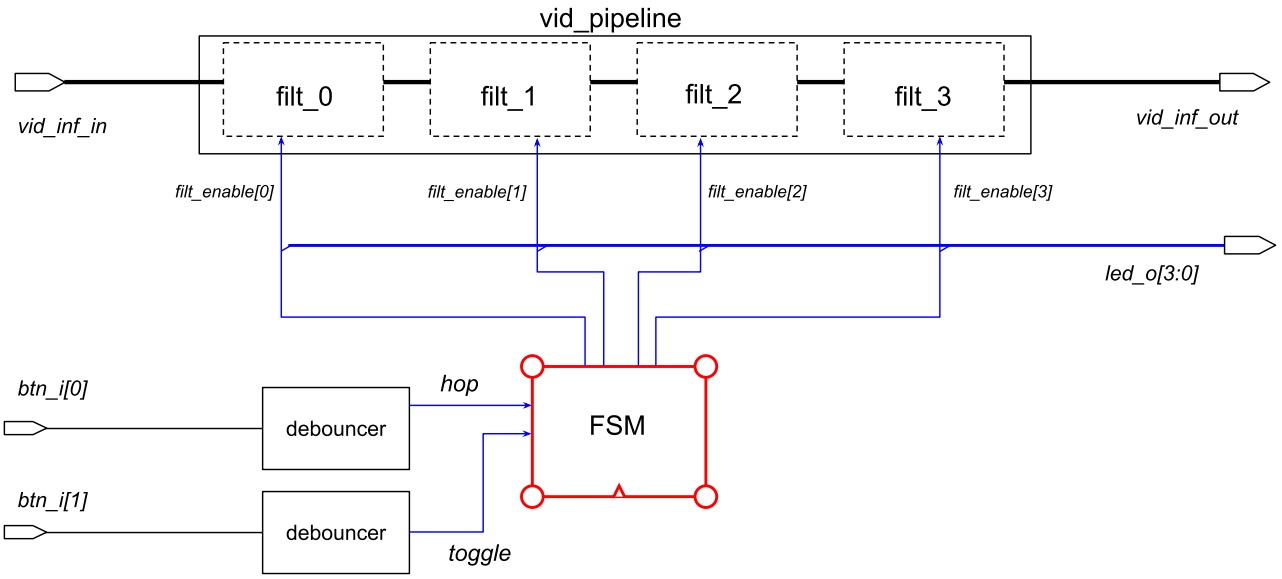


Figure 4: High level block diagram - assignment 2

As shown in Figure 4, the module `vid_pipeline`, defined in **`vid_pipeline.sv`**, is already instantiated in the top-level module `rgb_proc`. `vid_pipeline` contains the four different filter blocks, which can be separately enabled by driving any combination of the `enable[3:0]` inputs high. The `enable[3:0]` port is driven by the `filt_enable[3:0]` signal. The FSM you design should have outputs that drive the `filt_enable[3:0]` signal. The FSM will take two inputs from the user. Namely:

1. `hop`: During a LOW to HIGH transition of this input, the selected filter position will circular right shift by 1 (refer to Figure 2). During reset, the selection goes back to the left-most filter.
2. `toggle`: During a LOW to HIGH transition of this input the enable state of the selected filter should toggle. During reset, the default enable state of a filter is LOW.

The design of an FSM starts with the definition of the state variables and a state-space enumeration.

If you try to build a monolithic FSM that fulfills all the specifications, you will soon realize that this strategy brings to a overly complex design with an unmanageable state explosion. Since there are orthogonal requirements, you can split the FSM design in more sub-circuits. For example:

1. A circuit to detect LOW to HIGH transitions of the input signals, i.e. an edge-detector that generates a 1-cycle pulse when a rising edge is detected.
2. A circuit to decide the filter-enable values for the next clock cycle.

If you think about it, these circuits are FSMs. Imagining the black box *FSM* as a single unit or as a composition of sub-systems is only a matter of abstraction.

Student Task 8 (Defining states):

1. Try to think about a circuit to detect LOW to HIGH transitions of an input signal, for example the one that generates a pulse when the input `hop_i` is asserted. Consider the case where `hop_i` the output of a button, and the case where `hop_i` is the output of a flip-flop. How do you need to adapt your circuit?

   _____

   _____

2. Think about this small circuit as an FSM. Is this a Moore or a Mealy FSM? Which is the difference?

Draw both a Moore and Mealy FSM for two edge detector circuits.

Now that you have transformed the two inputs into their pulsed version, you can design the last FSM. Even if it's possible to split again this FSM into two separated ones, we will stick to a unique instance of it.

An example definition of state variables and their enumerations are provided in the state diagram in Figure 6. It defines two types of states: First state variable `pos` state with four distinct states `Pos0`, `Pos1`, `Pos2`, `Pos3` indicated by bigger purple circles in the figure. This state identifies the current selection. We will use the FPGA push buttons to supply the `hop` and `toggle` inputs. To limit the number of hops during a single push, the state transitions on `pos` state happen on positive edges of `hop` indicated by `p_hop` arrows in the figure. The second state variable is the `enable` state, of which each filter has one instance. This state variable will have two states called `LOW` and `HIGH` shown by smaller green circles in the figure. A transition on the `enable` state occurs on positive edges of `toggle` as indicated by `p_toggle` signal in the figure. The green circles represents only the allowed state-changes, e.g. if the `pos` present-state is `pos2`, `p_toggle` will only influence the `enable` next-state acting on `enable[2]`.

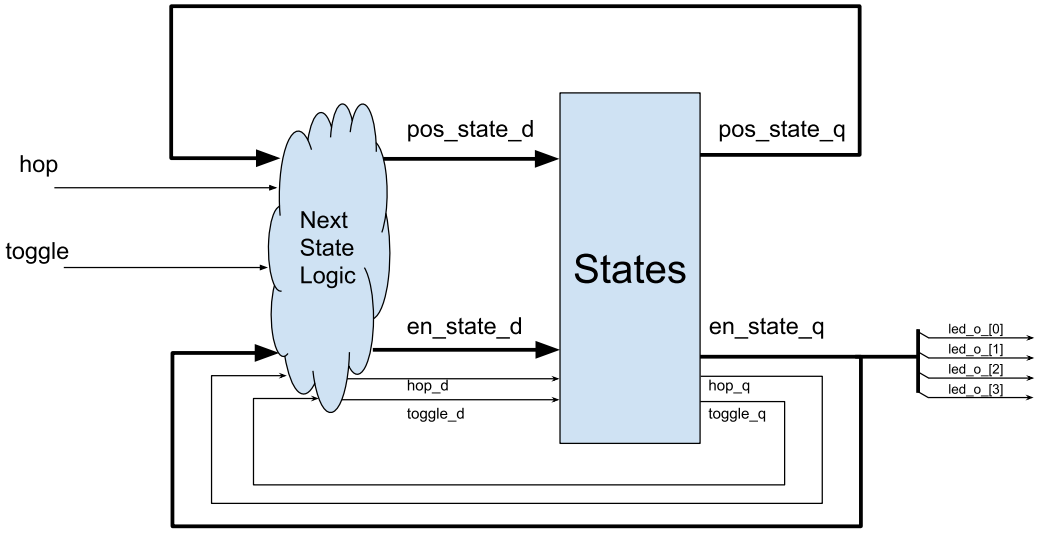


Figure 5: Schematic diagram - assignment 2

The next stage of the FSM design is coding the FSM in SystemVerilog. This is done in three steps. In step 1, state variables and their enumerations should be defined. In step 2, the next state logic should be defined. In step 3, the state assignment should be defined.

Student Task 9 (Coding the FSM):

1. Open **`filt_fsm.sv`** in the **`2_fsm/sourcecode`** folder using a text editor
2. Declare state variable types using the `typedef enum` keyword combination. You need to declare only the `position` state variable in this way, as each enable bit has only two states that can be directly represented by `1'b0` and `1'b1`. The syntax to introduce a new 4-states state variable type is:
   `typedef enum logic[1:0] {Pos0, Pos1, Pos2, Pos3} state_t;`
   Then, you will be allowed to use the `state_t` type to declare the related state variable signals.
3. Define next and current instances of state variables using the variable types defined above.
4. New sequential and combinational signals `p_hop` and `p_toggle` should be defined that sense positive transitions of `hop` and `toggle` signals.

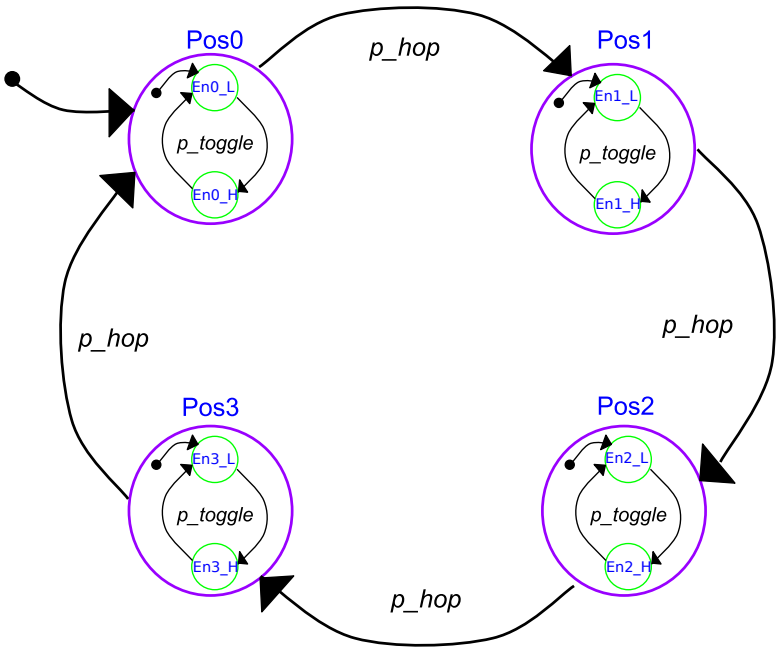


Figure 6: State diagram for filter selection - assignment 2

5. Write next state logic for each state using `always_comb` blocks. Refer to the cloud-like region in Figure 5 to get an idea about the inputs and outputs of the `always_comb` block. The **`case`** construct could be used to define behavior in different states. Make sure to have a default assignment for the combinational signals to avoid inference of unintended latches. Optionally, output assignments could also be included in the same `always_comb` blocks as required. Refer to the "SystemVerilog Examples" EDA Wiki page[a] to familiarize yourself with the right syntax. **In particular, please look at the Finite State Machine code snippet.**
6. Write sequential assignments using an `always_ff` block. Refer to the square region in Figure 5 to get an idea about the inputs and outputs of the `always_ff` block. Furthermore, add an active-low reset condition and assign correct values on reset for the state variables. Assign zeros to the other variables.

[a] http://eda.ee.ethz.ch/index.php/SystemVerilog_Examples

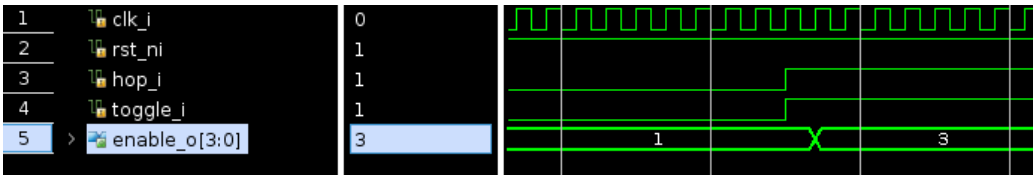


Figure 7: Expected waveform for filter selection - assignment 2

It is now the time to compile the code you just wrote.

Student Task 10 (Create Vivado project):

1. Open a terminal in **`2_fsm/vivado`** directory and run the following command:

```
sh> make gui
```

2. Make sure **`rgb_proc_tb.sv`** is set as top level of your `Simulation Sources`. Otherwise add **`rgb_proc_tb.sv`** as a simulation source from **`2_fsm/sourcecode`**, then right click on the file in the `Sources` window under `Simulation Sources` and click "Set as Top".

3. Make sure **zybo-z7.xdc** is listed under 'Constraints' in Sources window. Otherwise add it to the project from **vivado/constraints** directory.
4. Check if there are any critical warnings or errors in `Messages` windowpane of Vivado. If so, it means there are syntax errors in the code you wrote. The error messages provide hints as to the location and the kind of error generated. Fix these errors, if any, before moving to the next part of the exercise.

Once syntax errors are fixed, we can simulate the design.

Student Task 11 (Simulating the design):

1. Click `Run Simulation` on the left pane of the Vivado window. Then `Run Behavioral Simulation`. Note: make sure that the simulation is complete (output: "simulation done") if not, click on the blue play button on top of the window to run all.
2. Check if the simulation runs without errors. If there are any errors that prevents simulation from starting, these errors should be first fixed. Have a look at the Tcl Console and the `Messages` window pane to get hints about the errors and how to fix them. Seek assistance from an instructor if you are having trouble fixing errors.
3. Run the simulation till then end.
4. If your solution to the above task produces incorrect outputs, the simulation will end with an '**output mismatch**' message in the Tcl Console. If you see this error, please make sure your timings are coherent with the ones in Figure 7
5. If you encounter errors, revisit your solution or seek assistance from an instructor.

Having ensured the correctness of our design in the simulation, we can test the FSM design on the FPGA board.

Student Task 12 (Programming the FPGA):

1. Click on `Generate Bitstream` on the left side pane in Vivado window and initiate bitstream generation process.
2. Once the bitstream is ready, open the `Hardware Manager` and program the FPGA using the bitstream just created.

We have attached the push button `BTN0` of the board to the signal `btn_i` of the top-level design defined in **rgb_proc.sv**. This signal is routed to the FSM's input port `hop_i`. Similarly, `BTN1` is wired to the FSM input `toggle_i`. The `enable_filt` signals in `rgb_proc` are attached to LEDs `LD0` to `LD3` on the board. See Figure 4. These constraints are defined in **zybo-z7.xdc**.

Plug the HDMI-DVI cables to create a link between your PC, the FPGA and the screen. The video signal generated by the PC will pass through your FPGA and then will reach the screen. We have also added an optional switch `SW3` to switch between original stream and filtered stream in case you want to use the monitor for normal work without re-plugging the original cable to the monitor. To view the filtered video stream, set the `SW3` into HIGH mode.

Student Task 13 (On board testing):

1. Manipulate `BTN0` and `BTN1` on board to observe the behavior of all 16 combinations of filters on the picture **secret.jpg**. Use LEDs to keep track of the filter currently being applied.
2. Which configuration of filters gives the most realistic output?

   _____

   _____
3. Comment on the function of each filter by observing its behavior when all other filters are ON.

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

5 Using counters to control the filter chain

In the previous assignment the application of filters was manually controlled using push buttons. The filter that allows the best visualization of the picture was found only after applying the filters sequentially. If the quality difference of two filter combinations is only marginal then it may not be possible to choose between the two unless both filters can be viewed at the same time.

In order to view two or more filter combinations at the same time on the screen, one has to make sure that the filter enable signals are switched the required number of times within the duration of a frame, such that different regions of the image can be filtered differently. However, it is not possible to perform this task using push buttons as the frame rate of 50fps is too fast to manipulate the signals in real time using push buttons.

To overcome this problem we propose the use of *counters*. A counter is another classic use case of sequential circuits. The basic function of a counter is to generate an interrupt after a certain number of events happen. The counter resets at this point and restarts counting the events until the specified limit is reached again, upon which another interrupt is generated.

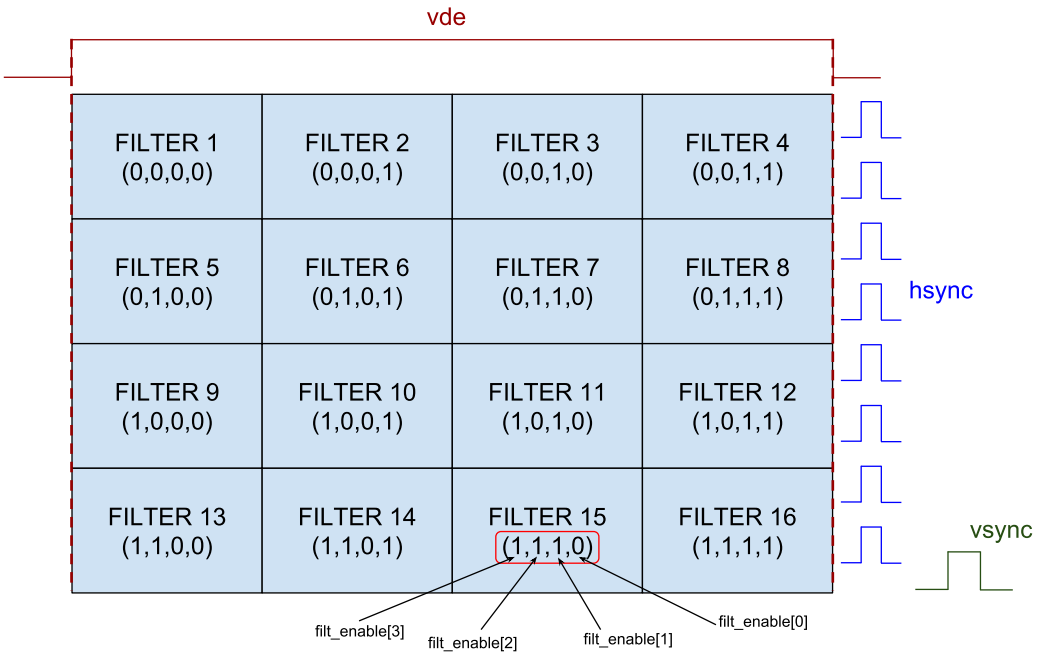


Figure 8: Filter distribution on screen - assignment 3

Our goal is to apply all 16 different filter combinations on different regions of the screen as shown in Figure 8. In order to apply regional filters, pixel data needs to be identified by the region that it belongs to. Synchronization control signals `hsync`, `vsync` and `vde` defined in the HDMI specification can be used for this purpose. The function of each of the control signals is given below:

1. `vde`: asserts when valid pixel data is present on the pixel data bus. Deasserts during 'blanking periods'.
2. `hsync`: asserts and deasserts once at the end of each row of pixels. When the signal is asserted, it can remain high for more than one cycle, i.e. creating one single- or multi-cycle pulse per row of pixel.
3. `vsync`: asserts and deasserts once at the end of each frame of pixels. When the signal is asserted, it can remain high for more than one cycle, i.e. creating one single- or multi-cycle pulse per frame of pixels.

An approximate timing diagram showing the above signals `hsync`, `vsync` and `vde` is given in Figure 9. In this figure N number of pixel rows are shown to be contained in a frame. In between the rows, *horizontal blanking period* occurs. Exactly one `hsync` pulse occurs during the horizontal blanking period. At the end of a frame exactly one `vsync` pulse arrives during *vertical blanking period* as shown in the figure. As depicted, `vsync` and `hsync` can also last for more than one cycle.

In addition to the synchronization control signals we need two signals that control the handshaking of data in the interface.

1. `valid`: This signal is provided by the source. A HIGH value on `valid` confirms the presence of correct data on the interface.
2. `ready`: This signal is provided by the target. A HIGH value on the `ready` signal specifies that the target is willing to accept data on the interface.

A data token (which includes RGB data and synchronization control signals in this case) is said to be transferred correctly on the interface during cycles in which both `valid` and `ready` are HIGH.

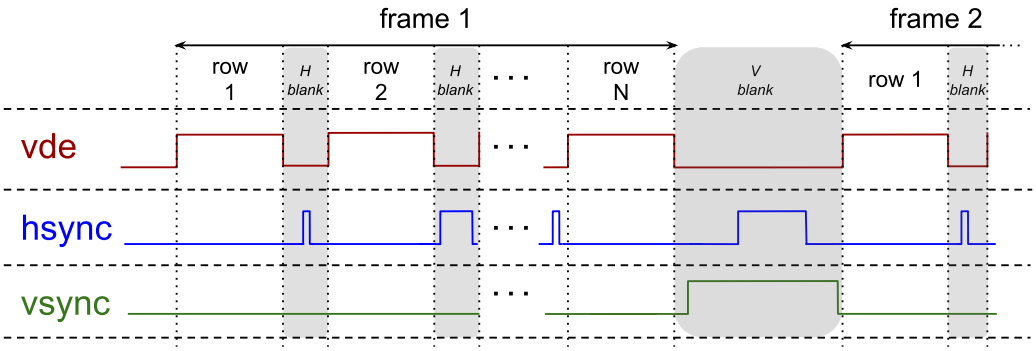


Figure 9: Timing diagram of control signals - assignment 3

The top level block diagram of the modified design that applies all 16 different filter combinations in a single frame is given in Figure 10

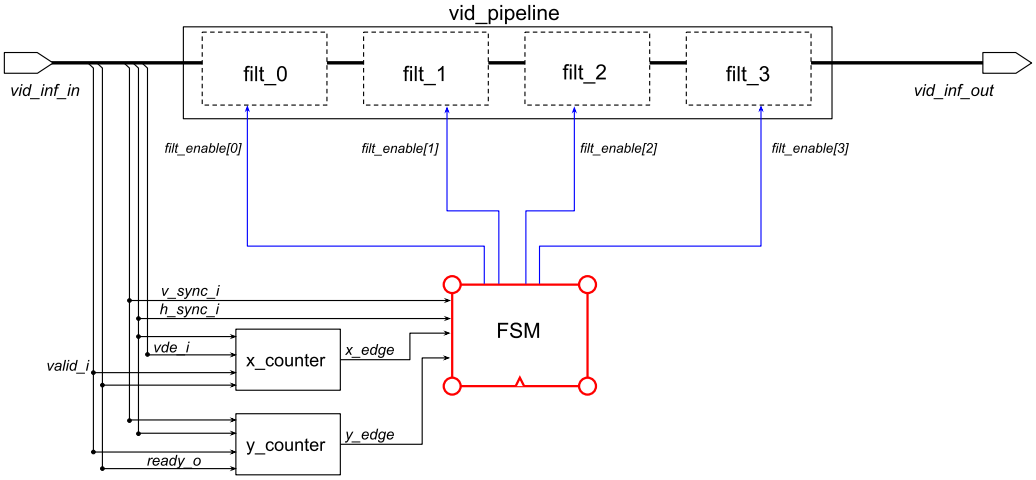


Figure 10: Top level block diagram - assignment 3

We have two new modules namely: `x_counter` and `y_counter`. `x_counter` keeps track of horizontal location of the pixel stream while `y_counter` keeps track of the vertical location of the pixel stream. Your first task is to write SystemVerilog code for the behavior of these modules.

Student Task 14 (Coding the Counters):

1. Open **`x_counter.sv`** file in location **`3_counter/sourcecode`** using a text editor
2. Write next state logic for `x_count_d` and `x_edge_d` using `always_comb` blocks. Before you start coding the counters there are a few things to consider
   a) ALL the changes to `x_counter` and the assertion of `x_edge` should be qualified by a handshake between `valid` and `ready`, i.e. `x_counter` should not change and `x_edge` should not be asserted if one between `valid` and `ready` was not asserted.[a] You must check this condition before evaluating the others.

b) The behavior of the `x_counter` should be such that it resets the cycle after `hsync_i` signal is asserted (`hsync_i` acts as a sync reset), otherwise up-counts during cycles when `vde_i` is HIGH until the counter reaches quarter of the horizontal resolution (note: this should be parametrizable). The horizontal resolution is passed into the module via the parameter `XResolution`.

c) Then `x_counter` resumes counting from zero. When `x_counter` returns to zero in this way, `x_edge` should be asserted for one cycle.

3. Write sequential assignments for `x_count_q` and `x_edge_q` using a `always_ff` block. Furthermore, add an active low reset condition and assign zeros for state variables.
4. Open **`y_counter.sv`** file in location **`3_counter/sourcecode`** using a text editor
5. Write next state logic for `y_count_d` and `y_edge_d` using `always_comb` block.

a) The same handshake rules pointed out before are valid also here: all the changes to `y_counter` and assertion of `y_edge` should be qualified by a handshake.

b) The behavior of the `y_counter` should be such that it resets in the presence of `vsync_i` signal, otherwise up-counts whenever `hsync_i` has a positive transition [b] until the counter reaches one quarter of the vertical resolution. The vertical resolution is passed into the module via the parameter `YResolution`.

c) Then `y_counter` resumes counting back from zero. When `y_counter` returns to zero in this way, `y_edge` should be asserted for one cycle.

6. Write sequential assignments for `y_count_q` and `y_edge_q` using a `always_ff` block. Furthermore, add an active low reset condition and assign zeros for state variables.

[a] This is valid for the always_comb block, as the synchronous reset in the always_ff block should work even without the handshake!

[b] Pay attention to this detail. hsync_i can also last for more than one cycle, and you don't want to keep on counting during that period. But you are now an expert of detecting edges from signals!

Student Task 15 (Connect the counters to their environment):

1. Instantiate `x_counter` module and `y_counter` module in `rgb_proc.sv` module found in **`3_counter/sourcecode`** directory. Use the already defined wires when connecting the ports of the instances. Also, pass the parameter values from the top level when instantiating. Hint: You will need to detect a transition between two **valid** `hsync` to detect new lines (see next question).
2. Each counter receives a pair of `valid_i` and `ready_i` signals to verify when a correct handshake was performed. Who is performing this handshake? Which is its purpose?

   _____

   _____
3. Check if you have connected the right handshake signals to the counters. Explain why you made such connections. Hint: Refer to Fig-10 for signal connections.

   _____

   _____

It is now the time to compile the code you just wrote.

Student Task 16 (Create Vivado project):

1. Open a terminal in **`3_counter/vivado`** directory and run the following command:

```
sh> make gui
```

2. Make sure **rgb_proc_tb.sv** is set as top level of Simulation Sources.
   Otherwise add **rgb_proc_tb.sv** as a simulation source from **3_counter/sourcecode**, then right click on the file in the Sources window under Simulation Sources and click "Set as Top".
3. Make sure **zybo-z7.xdc** is listed under Constraints in the Sources window. Otherwise add it to the project from the **vivado/constraints** directory.
4. Check if there are any critical warnings or errors in the Messages window pane of Vivado (except for timing requirements). If so, it means there are syntax errors in the code you wrote. The error messages provide hints as to the location and the kind of error generated. Fix these errors, if any, before moving to the next part of the exercise.

Once the syntax errors are fixed, we can simulate the design.

Student Task 17 (Simulating the design):

1. Click Run Simulation on the left pane of the Vivado window. Then Run Behavioral Simulation.
2. Check if the simulation runs without errors. If there are any errors that prevent the simulation from starting, these errors should be first fixed. Have a look at the 'Tcl Console' and 'Messages' window pane to get hints about the errors and how to fix them. Seek assistance from an instructor if you are having trouble fixing errors.
3. Run the simulation till the end. The resolution parameters are overridden to have smaller values in the testbench to reduce simulation time.
4. If your solution to the above task is wrong, the simulation will end with a '**output mismatch**' message in the 'Tcl Console'.
5. If you encounter errors, revisit your solution or seek assistance from an instructor.

Once it is made sure that the simulation is passing, we can test the counter design on the FPGA board. Generate the bitstream and program the FPGA as before.

Similar to the previous exercise, plug in the HDMI-DVI cables in the correct order. SW3 can be used to switch between the original stream and filtered stream in case you want to use the monitor for normal work without re-plugging the original cable to the monitor. To view the filtered video stream, set the SW3 into HIGH mode. You can use the picture **ex3/2_fsm/resources/secret.jpg** to test your implementation.

Student Task 18 (On board testing):

1. How many filters do you see on the screen?
2. If you don't see 16 filters then can you reason why this can happen? Discuss it with the TAs.
   Hint: See which filters are distinct according to filt_enable signal in Fig-8.
3. This task is for the students whose screen is flickering or the filter bounderies are unequal.
   a) Is the output you get on the screen flickering? Would you expect flickering on the screen if the counters work correctly?

   b) Are filter boundaries equally spaced? Would you expect filter boundaries to be equally spaced if the counters work correctly? Hint: If they are not equally spaced, verify that the XResolution of your screen matches with the one used in the RTL.

$\mathcal{E}$ **You have reached the end of this exercise.**
Please discuss your answers and any open questions with an assistant.
Finally, please provide us with feedback by filling out the form on our website. Thank you! $\mathcal{E}$