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 6

IP Integrator

Prof. L. Benini
F. Gürkaynak
M. Korb

Last Changed: 2023-11-22 21:44:11 +0100

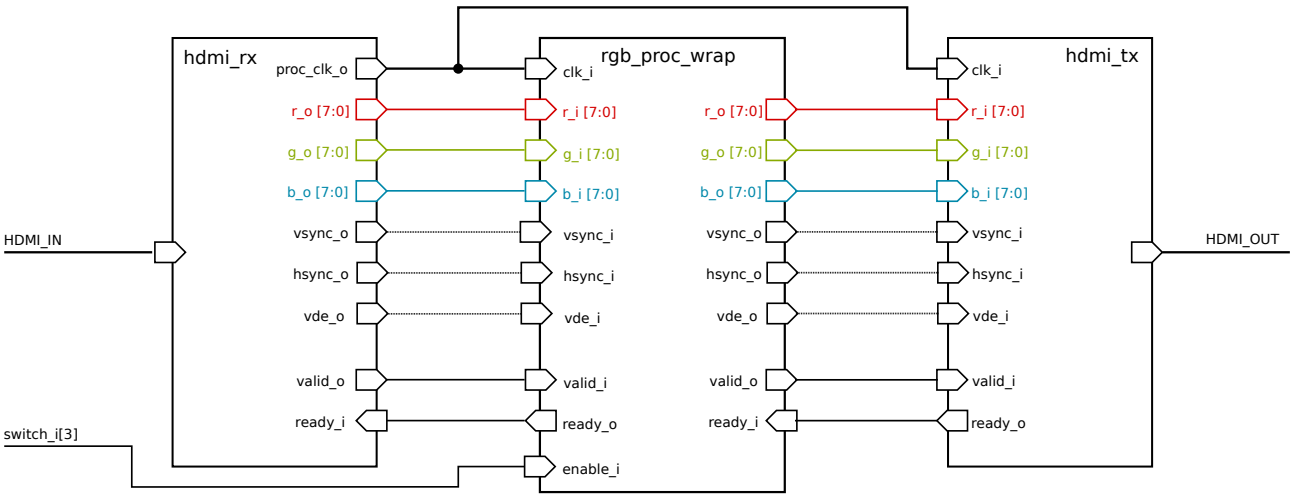

Figure 1: Overview of the HDMI image processing pipeline

1 Introduction

Over the last exercises, you have learned how to describe hardware in SystemVerilog. While hardware description languages allow you to implement any digital circuit precisely the way you want, they are sometimes impractical. Two important examples are (1) the rapid exploration of the architectural design space for a given algorithm and (2) the assembly of component modules into a system on a chip.

In this and the next exercise, you will work with tools that are better suited for these jobs: This exercise covers XILINX VIVADO IP Integrator, which makes (2) much more practical. The next exercise will introduce Vivado high-level synthesis (HLS), which aims to solve (1). Beware that neither tool replaces HDLs at their core task, i.e., precisely describing digital circuits, though. Rather, these tools complement HDLs so that you as hardware engineer can choose the right tool for a job.

This exercise depends on your knowledge from the prior exercises: First, we assume that you know the basics of Vivado from Exercise 1. You may want to use your notes and the material provided on our website for reference. Second, we will use the RGB processing block developed in the previous exercises as the core component for this exercise. While this exercise comes with its own basic implementation of the RGB processor, you can use your implementation with minor adaptations. We suggest you first use the implementation that comes with this exercise but replace it with your own implementation once you have the system working.

2 System Overview

We will use the familiar HDMI image processing pipeline that was already used in the previous exercises, shown in Fig. 1. However this time, we are going to assemble the HDMI receiver (`hdmi_rx`) and the HDMI transmitter (`hdmi_tx`) modules using the XILINX VIVADO IP Integrator.

As you know from the previous exercises, the Zybo Z7 development board features two HDMI connectors: one input (HDMI RX) and one output (HDMI TX). The signals on these interfaces can be read and written by the programmable logic inside the FPGA, respectively.

2.1 HDMI Interface

The HDMI, short for *High-Definition Multimedia Interface*, can be used to transmit digital video and audio data between a source device (a display controller for example) and a sink device (e.g. a screen). The video and audio signals are transmitted using *Transition-Minimized Differential Signaling* (TMDS), a protocol that encodes 8-bit data into 10-bit codewords and sends these over a differential serial link as shown in Fig. 2. The toggles between low and high are kept to a minimum and the number of ones and zeros are balanced in the encoding so that over time, the average DC level on the lines is constant.

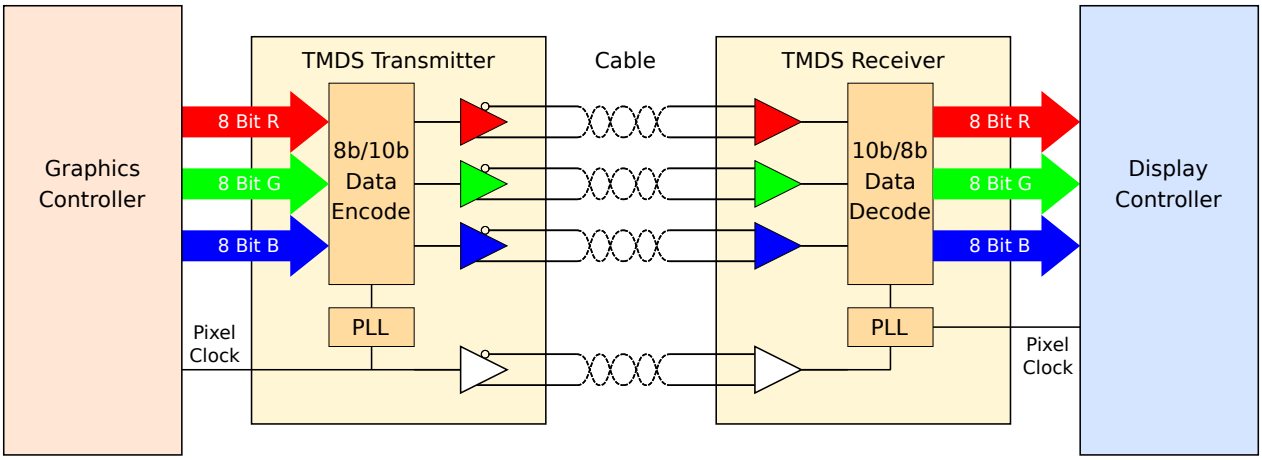

Figure 2: The TMDS link used in HDMI to transmit video data. Source: wikipedia.org

HDMI uses three TMDS channels to transmit red, green and blue color signals in parallel. The clock signal used to send the data, called the *pixel clock*, is also sent over a differential link and reconstructed at the sink device. The pixel clock toggles once per pixel, so 10 bits are transmitted during one cycle of the pixel clock. The sink device will try to re-synchronize the received data with the received clock signal.

All video data of one image row are sent in series over these channels at once. After each row as well as at the end of the frame, so-called blanking periods take place. During this time, additional data such as audio is sent over the links. So the data received over the R/G/B channels is not always pixel information.

There are also other communication channels in HDMI such as the *Display Data Channel* (DDC), which is basically an I^2C link used to read information about the sink device and its capabilities such as the supported audio/video formats. Newer versions of HDMI have additional channels such as Ethernet.

2.2 DVI to RGB

The video signal in HDMI is transported using the same 8b/10b-encoding as DVI (Digital Video Interface). To more easily access the image information in the familiar red-green-blue (RGB) format, XILINX VIVADO provides configurable IP[1] blocks that can convert DVI to RGB and vice-versa.

The decoder block will convert the TMDS signals into R/G/B values of 8-bit each, along with the recovered pixel clock. The encoder will take the R/G/B signals and send them out over TMDS with the proper encoding.

3 Getting Started

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/ex6/install.sh
sh> cd ex6
```

Student Task 2 (Create the project in Vivado): Change to the `vivado` directory and start Vivado by entering

```
sh> make
```

[1] *IP* is short for Intellectual Property module and, in this context, refers to a standalone hardware component distributed in some form of code.

To give you a quick start, we provide you with an initalized project that already hosts many of the blocks we will need.

The `zybo_top` block diagram should open automatically in IP Integrator. If it does not, click on *Open Block Design* in the *IP Integrator* step of the *Flow Navigator* and double click on `zybo_top` in the *Sources* window.

Vivado IP Integrator has two main components: First, a graphical user interface that lets you visually compose a system on the FPGA by placing modules and connecting them. Second, a catalog of standard modules such as memories, FIFOs, or clocking infrastructure, but also buses and peripheral interfaces.

The prepared block diagram already contains all input and output ports and the following blocks:

- The `hdmi_rx` block, which you can expand by clicking the **+** symbol in the top left of the block. It contains:
  - A clock generator, `clk_gen`, which we have configured to derive two clocks from the input clock: a fast one, `clk_out1`, for the DVI/RGB IPs and a slower one, `clk_out2`, for the digital logic on the FPGA.
  - A system reset generator, `rst_gen`, which synchronizes the asynchronous reset input with the slowest clock and provides sequenced resets for busses, interconnects, and peripherals (and a processor, which we do not use here). There are both an active-high and active-low reset generated, as many FPGA-native blocks use the active-high variant.
  - The DVI-to-RGB decoder, `dvi2rgb`, which is an IP provided by Digilent, the vendor of the Zybo boards.
- The `hdmi_tx` block, which contains:
  - The RGB-to-DVI encoder, `rgb2dvi`, which is another IP provided by Digilent.

The `locked` output of the clock generator becomes high as soon as the clock outputs are stable, causing the reset generator to deassert the `rst_o`/`rst_no` pair (the former active-high, the latter active-low).

Note that blocks in Vivado mostly have inputs on their left side and outputs on the right side (except for bundled interface ports such as `hdmi_rx_ddc`).

The clock generator is a Xilinx-provided IP, which is highly configurable.

Note 1 (IP Documentation): If you are unsure about the functionality of an IP block or simply would like to know more about it, check out the IP documentation by right-clicking the IP block and selecting *IP Documentation→View Product Webpage*.

Student Task 3 (Get familiar with the clocks): Double-click on the `clk_gen` to open its settings and properties. What are the frequencies of the two output clocks?

We are now going to set up the digital video processing chain in three steps: First, we will assemble the HDMI receiver and convert the video data to RGB, then we'll assemble the transmitter and convert the signal back to DVI. Secondly, we will add clock domain crossings to move RGB data to and from the FPGA logic that uses the clock `clk_out2`. Lastly, we will insert the video processing pipeline into the path on the FPGA to modify the video data in flight.

4 HDMI Feed-Through on the FPGA

In the IP Integrator diagram view, individual connections between ports can be made by dragging and dropping their ends with the mouse. The mouse will turn into a pencil icon if you can start dragging a connection from that location. Individual blocks can also be dragged around to rearrange the block diagram.

Student Task 4 (Connect HDMI RX to DVI-to-RGB decoder): We start by feeding the HDMI signals from the connector on the board to the receiving side circuit.

- Rearrange the diagram so that `hdmi_rx` is to the left of `hdmi_tx` to maintain a logical 'left-to-right' data flow.
- Connect the clock and HDMI input pins (prefixed `hdmi_rx_`) to the `hdmi_rx` block. Also connect the `hdmi_rx_ddc` interface which contains both inputs and outputs to the `hdmi_rx` block.
- HDMI contains an active-high 'Hot Plug Detect' signal, `hdmi_rx_hpd_o`, which lets a source device (the computer) know that a running sink device (here the FPGA, but usually a monitor) has been plugged in. Make sure this signal is asserted whenever the circuit is not in reset.

Hint: Whenever the diagram layout gets too messy, try the *Optimize Routing* button in the diagram view (second from right). Your experience may vary, though.

Student Task 5 (RGB decoder outputs to block outputs): Now we need to take care of the decoded data coming from the DVI-to-RGB IP (expand the `hdmi_rx` block by clicking on the **+** to find the DVI-to-RGB IP). The image data as well as synchronization signals are packed into the interface `RGB`.

- Expand the `RGB` interface by clicking on the **+** next to its name. You will find the individual signals in the interface.
- We would like to treat all these signals as one large logic vector. Instantiate an IP that concatenates signals by right-clicking an empty portion of the diagram and selecting *Add IP...*. Find the `Concat` IP and place it inside the `hdmi_rx` block.
- Give the IP a simple, meaningful name by clicking on the block and editing the name field in the *Block Properties* window on the left. Make this a habit. We suggest to name module instances according to the underlying entity/IP, with *meaningful* prefixes and postfixes (numbers are inexpressive in most cases) if there are multiple instances of the same entity.
- Double-click the `Concat` IP and configure and connect it to arrange the data into a logic vector as follows (in SystemVerilog syntax):
  `dout[26:0] = {vid_pVDE, vid_pHSync, vid_pVSync, vid_pData}`.
- In order to select a bit slice from a vector, the `Slice` IP is used. Slice off the individual color channels (packed into the video data as `pData[23:0] = {r, g, b}`) as well as the synchronization signals from the output of the `Concat` IP and connect them to the `hdmi_rx` block outputs. You can copy-paste IPs if you need them multiple times.

  (***Hint****: If you copy/paste modules in the full view, they will land outside the rx/tx blocks. If you double-click a block to open a new tab specified for that block, and do the copy/paste there, they will end up inside the block*)

  Note: While it technically wouldn't be necessary at this point to pack the synchronization signals with the video data to then slice them off, this will be needed for the last tasks of this exercise.
- Connect the clock `proc_clk_o` to the slower clock generated by the clock generator.
- Finally connect the pixel clock and clock-lock signals to the `hdmi_` clock and reset outputs, respectively.

Now, the incoming HDMI signals should all properly connect to the edge of the `hdmi_rx` block. There are still some missing connections not directly related to the HDMI interface which we're going to take care of later. Now, we'll turn to the HDMI transmitter block `hdmi_tx`, which in essence has the same functionality as the receiver but in reverse. Note that the clock and reset signals generated in the receiver block can be reused here and don't need to be regenerated.

Student Task 6 (HDMI TX block): Complete and connect the `hdmi_tx` block. Namely, also make sure to:

- First concatenate the input RGB signals as well as the HDMI synchronization signals as done in the previous block.
- Then split them into the components used in the DVI encoder's `RGB` interface.
- The modules are already instantiated, you just need to connect them.

While it technically wouldn't be necessary to pack the synchronization signals with the video data at this point, this will be needed at a later stage.

The only thing remaining now is connecting the TX block to the outputs as well as bridge the two blocks to create a feed-through path.

Student Task 7 (HDMI feed-through):

- Connect the outputs of the `hdmi_tx` block to the HDMI output pins of the Zybo board.
- Attach the proper clock and reset signals to the `hdmi_tx` block.
- In order to create a feed-through path, connect the video data outputs as well as the synchronization signals from the RX to the TX block.

It's now time to verify that our HDMI feed-through pipeline is working so far.

Student Task 8 (Deploy the design):

- Synthesize your code, implement it and deploy it to the FPGA. The `ready_*`/`valid_*` signals can be left unconnected for now.
- Unplug the DVI cable between the PC and the display at the display side.
- Plug the first HDMI-to-DVI cable to the second display port of your PC and to the HDMI RX port of the FPGA.
- Plug the second HDMI-to-DVI cable to the HDMI TX port of the FPGA and to the DVI port of the Display.

You should now see the same image as before except for a little change of the resolution. The Zybo board processes data at a slightly higher resolution than the monitors, which the monitors cannot display perfectly. Due to the resolution change, Vivado present an "Internal Exception", which you can safely ignore and hit `Continue`. The monitor also presents a warning message, which you can discard by pressing any button on the monitor. If anything else does not work as expected, please ask an assistant for help.

Note 2 (Parallel synthesis): Make sure Vivado uses the maximum number of jobs available on your machine. Synthesizing independent modules in parallel significantly reduces overall run time.

The number of parallel jobs can be configured in the *Launch Runs* dialog that appears when you launch a new job. If you have deactivated that dialog, you can re-enable it by ticking *Show Launch Runs dialog when launching run* under *Tools→Settings→Window Behavior→Notifications*. You can deactivate that dialog after you have set the number of jobs.

5 Clock Domain Crossings

Next, we want to use the FPGA logic of the Zynq to process the video data from the receiver before sending it out to the transmitter. However, the FPGA logic is being clocked by a different clock signal than the HDMI blocks. This is a problem as both circuits could run at a different frequency[2] and signals from one clock domain will become asynchronous to the other.

[2] Matching the frequency alone does not mitigate the clock domain crossing problem, as the phase relation of the two clocks must also be controlled.

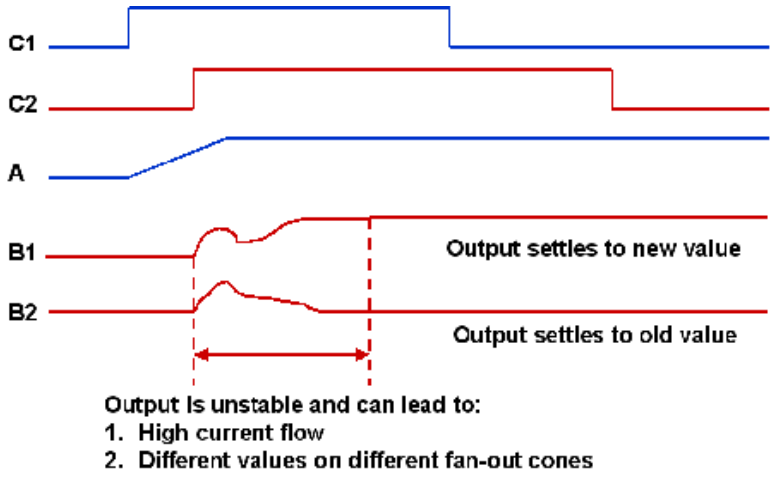

Figure 3: Some possible outcomes of a signal `A` crossing a clock boundary. `A` is sampled by `C2` and produces `B1`/`B2`. Source: eetimes.com.

This can lead to timing violations, corrupted data and even metastability in registers if signals that cross the boundary are not properly handled, as can be seen in Fig. 3. Furthermore, individual bits in multi-bit signals will likely no longer be in sync with each other as the difference in routing lengths could lead to some being sampled before, and some after a clock edge in the destination domain.

A safe way of crossing clock domain crossings with multi-bit data is using a FIFO buffer with two independently clocked ports, one to read and one to write.

In our video processing pipeline, there will be two clock domain crossings: one between `hdmi_rx` and the FPGA, where the information about the pixels is produced at `PixelClk` by the `dvi2rgb` and the FPGA logic clocked by `clk_out2` - and one between the FPGA logic and `hdmi_tx`, where the `rgb2dvi` is clocked by `PixelClk`. We need to add clock domain crossing FIFOs for both of them.

Student Task 9: Draw a diagram of the video processing datapath, from the input of the `dvi2rgb` block through the FPGA fabric and to the output of the `rgb2dvi` block. Make sure the two clocks and the two clock domain crossings are explicitly drawn. Also add the FIFOs on the clock domain crossings.

Note 3: Discuss your solution with an assistant before proceeding with the exercise.

We are now able to implement this clock domain crossings in the IP Integrator.

Student Task 10 (Add input and output FIFOs for video data):

- Add two FIFO IPs for the receiver and transmitter data to the block diagram by right-clicking at an empty place and selecting *FIFO Generator* from the *Add IP* menu.
- Give them meaningful names as `rx_dc_fifo` and `tx_dc_fifo`.
- Configure them (in the double-click menu) as follows:
  - Select *Independent Clocks Builtin FIFO* as *Implementation*.
  - Set the *Read Mode* to *Standard FIFO*.
  - Set both the *Write Width* and the *Read Width* to *27* (i.e., the width of the combined video data and synchronization signals).
  - Set the *Write Depth* to *512*.
  - Set the *Read Clock Frequency* and *Write Clock Frequency* correctly. (***Hint**: The frequency information could be quickly found by 'Block Pin Properties' window by clicking on the clock pin.*)
- Place each FIFO in its respective block.
- Connect the data ports of the receiver FIFO to the outputs of dvi2rgb to sample the video data. Then, connect the write clock `wr_clk` to the correct clock source for data inputs.
- Use the `aPixelClkLckd` port from dvi2rgb, which indicates the clock signal is stable, as the write enable input of the FIFO.
- Use the `ready_i` port of the `hdmi_rx` as the read enable of the FIFO.
- Connect the transmitter FIFO analogously, but this time the `valid_i` signal is used to write the FIFO.
- Use the `hdmi_rst_ni` port of the `hdmi_tx` as the read enable of the FIFO, it is practically the same signal as the write enable signal as the receiver FIFO, to balance both FIFO.
- Use the active-high reset for all FIFOs.
- Finally, clock the FIFOs such that they form the clock domain crossings described above.

Ultimately, we will connect the remaining handshake and data signals on the FIFOs to a filter on each channel. However, we first want to be sure that our clock domain crossings work without any processing in between. For this purpose, we will first connect the input FIFOs directly to the output FIFOs.

Student Task 11 (Directly connect input to output FIFOs):

- Find out how to connect the associated handshake signals so that one FIFO can feed the other (use the `full` and `empty` signals).

  _____

  _____

  _____

  _____

- Implement your solution. It is not necessary to implement anything in HDL, as you can add a *Utility Vector Logic* IP instance and configure it to perform basic logic operations. Also, don't forget to connect the input/output pairs of handshake signals together.

If something goes wrong while passing data through the FIFOs, you will very likely see (or not see) it when testing the implementation. Pinpointing the source of error, however, can then be quite difficult. An option to increase the observability of your implementation is to monitor the fill level of the FIFOs via the LEDs.

Optional Student Task 12 (Display the fill status of the FIFOs on the LEDs): Let us have the LED on the left light in case the receiving FIFO is full and the rightmost LED in case the output FIFO is full.

- Connect the `full` output of the FIFOs to the respective bits of `led_o`.
- Use a `Constant` block to tie the unused bits of `led_o` to zero.

Student Task 13 (Deploy and test the digital signal feed-through): Finally, generate a new bitstream and program the new bistream to the device. You should be able to see the image on the screen just like you did before.

6 Processing the video stream

At this point, we are ready to actually process the video signal. We will add the RGB processor that implements various filters from Exercise 2. In a second step, you can change the processing block or use your own implementation from last exercise.

Student Task 14 (Insert the RGB processor): In the IP integrator, you are not limited to using the IPs from the catalog. If you add HDL to the project, it can be instantiated by right-clicking and selecting *Add Module...*. We already included the relevant source files in the project for you.

- Add one instance of the `rgb_proc_wrap` module that comes with this exercise.
- Clock it with `clk_out2` and reset it with the active-low reset.
- Connect data and handshake signals such that the module is between the input and the output FIFOs.
- Use `SW3` on the board to control the enable for the processor and feed the rest into its switch inputs.
- Synthesize and implement the design, make sure that there are no errors, and check the timing of the implemented design.
  ***Hint**: You may get timing violations on the* Worst Pulse Width Slack[a]*. They are unrelated to your design and may be safely ignored in this project. Any other timing violation may* not *be ignored, though!*
- Finally, generate a new bitstream and program it to the FPGA.
- Verify whether the RGB processor is working now.

[a] Worst Pulse Width Slack (WPWS): Corresponds to the worst slack of the min low/high pulse width, min/max period and max skew checks when using both min and max delays; see https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug906-vivado-design-analysis.pdf.

Congratulations, you have successfully implemented an HDMI video processing pipeline! At this point, you can try to use your own implementation from Exercise 5 and/or modify the given design.

Optional Student Task 15 (Replace the design with your own implementation):

- Make sure your implementation passes the testbench of Exercise 2 without any errors.
- Copy the source files of your implementation to the **`sourcecode`** directory of this exercise, overwriting the file we provided.
- Switch to the block diagram in Vivado. If your implementation matches the provided one with respect to all interfaces, you should get a message bar at the top of the block diagram notifying you that the sources for one module have changed and asking you to upgrade it. Confirm the upgrade.
- Generate a new bitstream (with the usual checks) and program it to the FPGA.
- Verify that your own implementation works as you expect.

Congratulations, you have successfully implemented *your own* HDMI video processing pipeline!

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