Digital Design and Computer Architecture LU

Lab Exercises I and II

Florian Huemer, Jürgen Maier
{fhuemer, jmaier}@ecs.tuwien.ac.at
Department of Computer Engineering
TU Wien

Vienna, April 5, 2021

1 Introduction

This document contains the assignments for exercises I and II. The deadlines for these exercises are:

- Exercise I: 02.04.2021, 23:55
- Exercise II: 30.04.2021, 23:55

The combined points achieved in Exercises I and II count 25 % to the overall grade of the course. Please hand in your solutions via TUWEL. We would like to encourage you to fill out the feedback form in TUWEL after you submitted your solution. The feedback is anonymous and helps us to improve the course.

Please note that this document is only one part of the assignment. Take a look at the protocol template for all required measurements, screenshots and questions to be answered. Make sure that all necessary details can be seen in the figures you put into your report, otherwise they will be graded with zero points.

The application created in Exercises I and II is a simple "retro gaming system", which uses an NES controller [1] for user input and generates a video baseband signal (VBS, i.e., an unmodulated analog black and white TV signal[2]) as its main output. Additionally the large LCD attached to the board as well as the seven-segment displays and LEDs will be used.

In the game (implemented in Exercise II) the player controls a ball that is falling downwards and is blocked by upwards moving bricks. If the player hits the upper or lower border of the screen the game is over. To score points the player has to collect items placed on the bricks.

1.1 Coding Style

Refer to the "VHDL Coding and Design Guidelines" document before starting your solution. Moreover, we highly recommend to implement state machines with the 2 or 3-process method discussed in the Hardware Modeling lecture, since the 1-process method can easily lead to very hard-to-find bugs. We further recommend to use "named mapping" for connecting wires to an instantiated entity.

1.2 Software

As discussed in more detail in the Design Flow Tutorial, we are using Quartus and QuestaSim (formerly ModelSim) in the lab. If you want to work on your own computer you can download a free version of Quartus (Quartus Prime Lite Edition) and Questa/Modelsim (ModelSim-Intel) from the Intel website.[3] However, note that the simulation performance of ModelSim-Intel might be lower than the full version of Questa/Modelsim provided in the lab (especially for large designs).

We also proivde you with a (Virtual Box) VM image, which has the free version of these tools installed under CentOS 7 (the same operating system as used in the lab). You can download the VM using `scp` from `ssh.tilab.tuwien.ac.at:/opt/eda/vm/ECS-EDA-Tools_vm_02102020.txz`.

[1] `https://en.wikipedia.org/wiki/Nintendo_Entertainment_System#Controllers`
[2] `https://en.wikipedia.org/wiki/Composite_video`
[3] `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/download.html`

1.3 Submission

Do not change the latex template in any way. Most importantly do not delete, add or reorder any questions/subtasks (i.e., the “qa” environments). If you don’t answer a particular question, just leave it empty, but don’t delete it. Everything you enter into the lab protocol must be inside one of the “qa” environments, everything outside of these environments will not be considered for grading.

When including simulation screenshots, remove the window border and menus. Only show the relevant parts!

Further note that it is mandatory to put the files exactly in the required folders! The submission script will assist you to avoid mistakes.

1.4 Allowed Warnings

Although your design might be correct, Quartus still outputs some warnings during the compilation process. Table 1.1 lists all allowed warnings, i.e., warnings that won’t have a negative impact on your grade. All other warnings, however, indicate problems with your design and will hence reduce the total number of points you get for your solution.

The last two warnings in Table 1.1 may still indicate problems with your design. So thoroughly check which signals these warnings are reported for! If you have, for example, an input button that should trigger some action in your design but Quartus reports that it does not drive any logic, then there is certainly a problem. If you intentionally drive some output with a certain constant logic level (for example an unused seven segment display), then the “stuck at VCC or GND” warning is fine.

ID	Description
18236	`Number of processors has not been specified which may cause overloading on shared machines. Set the global assignment NUM_PARALLEL_PROCESSORS in your QSF to an appropriate value for best performance.`
13009	`TRI or OPNDRN buffers permanently enabled.`
276020	`Inferred RAM node [...] from synchronous design logic.  Pass-through logic has been added to match the read-during-write behavior of the original design.`
15064	`PLL [...]|altpll:altpll_component|pll_altpll:auto_generated|pll1" output port clk[0] feeds output pin "nclk~output" via non-dedicated routing -- jitter performance depends on switching rate of other design elements. Use PLL dedicated clock outputs to ensure jitter performance`
169177	`[...] pins must meet Intel FPGA requirements for 3.3-, 3.0-, and 2.5-V interfaces. For more information, refer to AN 447: Interfacing Cyclone IV E Devices with 3.3/3.0/2.5-V LVTTL/LVCMOS I/O Systems.`
171167	`Found invalid Fitter assignments. See the Ignored Assignments panel in the Fitter Compilation Report for more information.`
15705	`Ignored locations or region assignments to the following nodes`
15714	`Some pins have incomplete I/O assignments. Refer to the I/O Assignment Warnings report for details`
12240	`Synthesis found one or more imported partitions that will be treated as black boxes for timing analysis during synthesis`
13024	`Output pins are stuck at VCC or GND`
21074	`Design contains [...] input pin(s) that do not drive logic`
292013	`Feature LogicLock is only available with a valid subscription license. You can purchase a software subscription to gain full access to this feature.`

Table 1.1: Allowed warnings

2 Exercise I (Deadline: 02.04.2021)

2.1 Overview

In the first exercise you will already design your first FPGA application using VHDL. Prior to that it is, however, necessary that you make yourself acquainted with the tools and the remote working environment used in this lab course. A basic FPGA design flow consists of simulation, synthesis and place & route. The simulation is used to verify and debug functionality and timing of the circuits. During synthesis the behavioral and/or structural description is translated into a gate-level netlist. This netlist can then be mapped to the FPGA's logic cells. Finally the produced bitstream file is used to configure the FPGA.

Note that we provide you with a reference implementation in the form of an SOF (bitstream) file, that can be downloaded from TUWEL. If some explanation in this document is unclear, this implementation can be used as a guideline for how the finished system should behave. Nonetheless, don't hesitate to contact the teaching personal using the provided communication channels. Please note that the TU Chat can also be used to ask questions outside of the tutor lab slots. This way simpler questions can be answered immediately by the staff and you don't have to wait for the next tutor slot.

2.2 Required and Recommended Reading

All documents are available in TUWEL.

Essentials (read before you start!)

- Design flow tutorial
- VHDL introduction slides (Hardware Modeling)
- VHDL Coding and Design Guidelines

Consult as needed

- IP Cores Manual
- Datasheets and Manuals (e.g., for the board)

2.3 Task Descriptions

Task 1: Structural Modeling [8 Points]

Your task is to implement a top-level structural VHDL description of the system shown in Figures 2.3-2.5. The description must be done in VHDL and should contain only structural primitives (component instantiations and concurrent signal assignments). All information needed to wire the IP cores together is contained in the figures.

System Assembly: Create a new Quartus project in the `top/quartus/` directory of the template and add all needed IP cores and your top-level description to it. The name of this project shall be *top*. Quartus will create two files named `top.qpf` and `top.qsf`. Set the VHDL version of the project to VHDL-2008. We also provide you with a Makefile located in the `top/quartus/` directory, that allows you to start the synthesis process from the command line. This can be useful if you work on the lab computers over an SSH connection.

We already provide you with a template for the entity description (`top/src/top.vhd`). It already contains two instances and some connections. Leave them as they are. There are also some signals that are connected to an instance but which are not used in the design yet. These signals are marked with NC in the figures and will be used late in Exercise II. You may ignore the warnings that Quartus produces because of that.

The constants used for the generics in the figures are defined in Table 2.1. Make sure you use constants in VHDL. Do not set these values directly in the generic map sections.

Constant	Value	Description
SYNC_STAGES	2	The number of synchronizer stages used when (asynchronous) external signals are read.
WIDTH	400	The horizontal resolution of the game.
HEIGHT	240	The vertical resolution of the game.

Table 2.1: Constants

The ball_game module internally uses the module prng, which must also be added to the Quartus project. Note that some of the IP cores are only provided as precomplied modules. This includes the audio_cntrl, the prng, the dbg_port and the lcd_graphics_controller. These modules always come with one or more `*.vhd` files as well as a `*.qxp` and a `*.vho` file. For your Quartus project add all `*.vhd` as well as the `*.qxp` file (the `*.vho` is only required for simulations). For example, for the audio_cntrl the files `audio_cntrl_top.qxp`, `audio_cntrl_2s.vhd` and `audio_cntrl_pkg.vhd` are needed.

Unused (i.e., unconnected) outputs of instances (e.g., game_state or player_points) shall be marked with the "open" keyword. For your report include a screenshot of the overall system from the RTL netlist viewer in Quartus.

System Explanation: The shown design is the top-level module of our gaming system, which will be used and extended throughout exercises I and II.

Normally this top-level module provides connections to all required peripheral devices, such that a user is able to interact with the design. However, since this is a remote-only semester we had to make some slight adjustments, because using e.g., the board's buttons and switches (see Figure 2.1) is simply not possible in such a setup. Hence, we developed the dbg_port module which provides outputs for the buttons (dbg_keys) and switches (dbg_switches), which we will use instead of the "real" ones. In our remote lab environment these "virtual" buttons/switches can then be controlled using a simple python tool (`remote.py`). The dbg_port also outputs the dbg_nes_buttons signals, which we will use until Task 6, where the NES controller interface will be implemented. However, for the same reasons as for the board buttons/switches you will not be interfacing with a "real" controller. Instead the dbg_port will emulate the controller's behavior over a external loop back using the GPIO pins of the FPGA.

Additionally the dbg_port module is also able to read the state of the LEDs and the HEX displays, which can then be retrieved using `remote.py`. Another feature is the ability to send instruction to a graphics controller using the gfx_* signals. This will be relevant for Exercise II since there you will implement your own graphics controller.

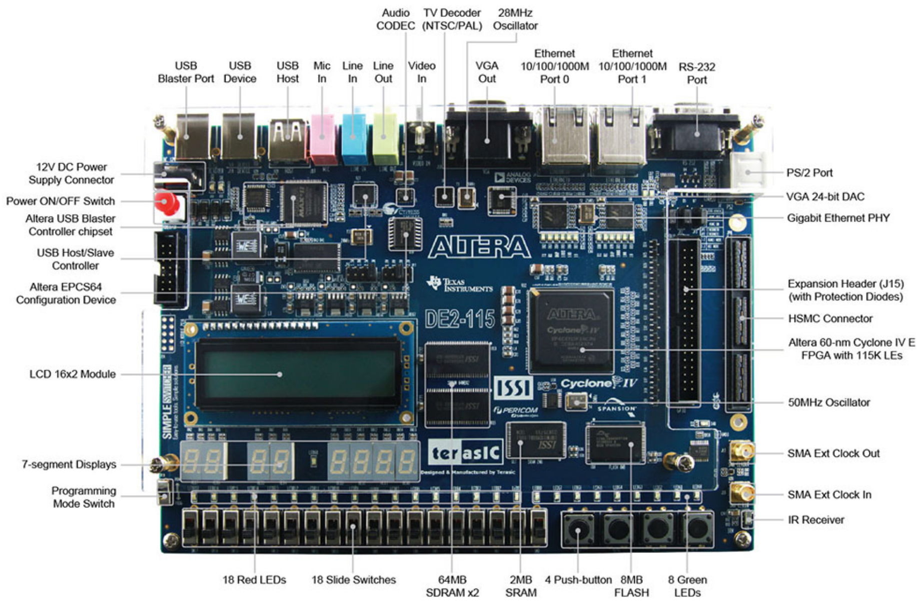


Figure 2.1: DE2-115 FPGA Development Board

The "heart" of the system is the ball_game module which implements the actual game logic (Exercise II), processes controller input, sends instructions to graphics controller and plays sounds using the audio_cntrl. The display_switch module selects which graphics controller the game should use and is controlled by dbg_port.

After everything is assembled correctly you should have a system that draws a ball on the board's LCD, that can be moved using the arrow keys of the virtual controller in the remote.py tool. Whenever the ball is moved a sound is played.

To use the remote.py tool in the TILab, please execute the following command (when logged in at a TILab computer).

```
pip3 install --user termcolor dataclasses docopt pyserial pyusb PyVISA PyVISA-py
```

PLL Generation: The PLL shown in Figure 2.3 is not supplied. You need to generate it using the corresponding wizard in Quartus (see the Design Flow Tutorial for further information). The frequency of the display clock (first PLL output) must be 8 MHz. The second clock output of the PLL is required by the audio controller and must be configured to 12 MHz. Place the PLL generated by the wizard in the top/src/ folder.

Create and add an SDC file as discussed in the Design Flow Tutorial. Additionally add the following lines to end of this file:

```
set_false_path -from [get_clocks {clk}] -to [get_clocks {PLL_INST_NAME|altpll_component
    |auto_generated|pll1|clk[0]}];
set_false_path -from [get_clocks {clk}] -to [get_clocks {PLL_INST_NAME|altpll_component
    |auto_generated|pll1|clk[1]}];
```

PLL_INST_NAME must be replaced by the name of your PLL instance in the top-level design architecture.

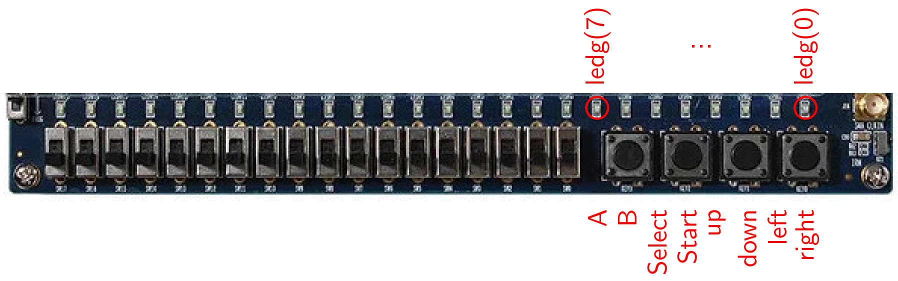


Figure 2.2: Controller button/LED mapping

Reset: Notice that the dbg_port module's reset is directly connected to keys(0). All other components are reset by the signals res_n, audio_res_n and/or display_res_n. These signals are generated using the (synchronized) output of the AND gate in Figure 2.3. Hence, the design can be reset by either a "virtual" button press on dbg_keys(0) or a "real" button press on the board (keys(0)). This feature is important for the top testbench used in Tasks 3 and 4, so be sure to implement it correctly.

Connecting the LEDs: The current state of the NES controller buttons (i.e., which button is pressed/not pressed) is represented using the record data type nes_buttons_t declared in the nes_controller_pkg package. This data type contains one flag (std_logic) for each of the controller's buttons. As long as a button on the controller is pressed, the corresponding flag in this record is set to one. Connect the individual button signals in accordance with Figure 2.2 (i.e., ledg(0) shall be connected to dbg_nes_buttons.btn_right and so on).

The ledg(8) output shall be set to constant `'0'`; The red LEDs (ledr) shall be directly connected to dbg_switches. The hex{0-7} outputs shall be set to constant `'1'` for now.

Pin Assignments: You don't have to take care of (most of) the pin assignments by yourself. Simply import the provided pinout file located in `top/quartus/top_pinout.csv`, as discussed in the Design Flow Tutorial. Now everything *except* for the 50 MHz clock signal is connected. Consult the FPGA board manual to find out its exact location (the signal is called CLOCK_50 in the manual) and assign it using the Pin Planner in Quartus. Be sure to select the correct I/O Standard (3.3-V LVTTL).

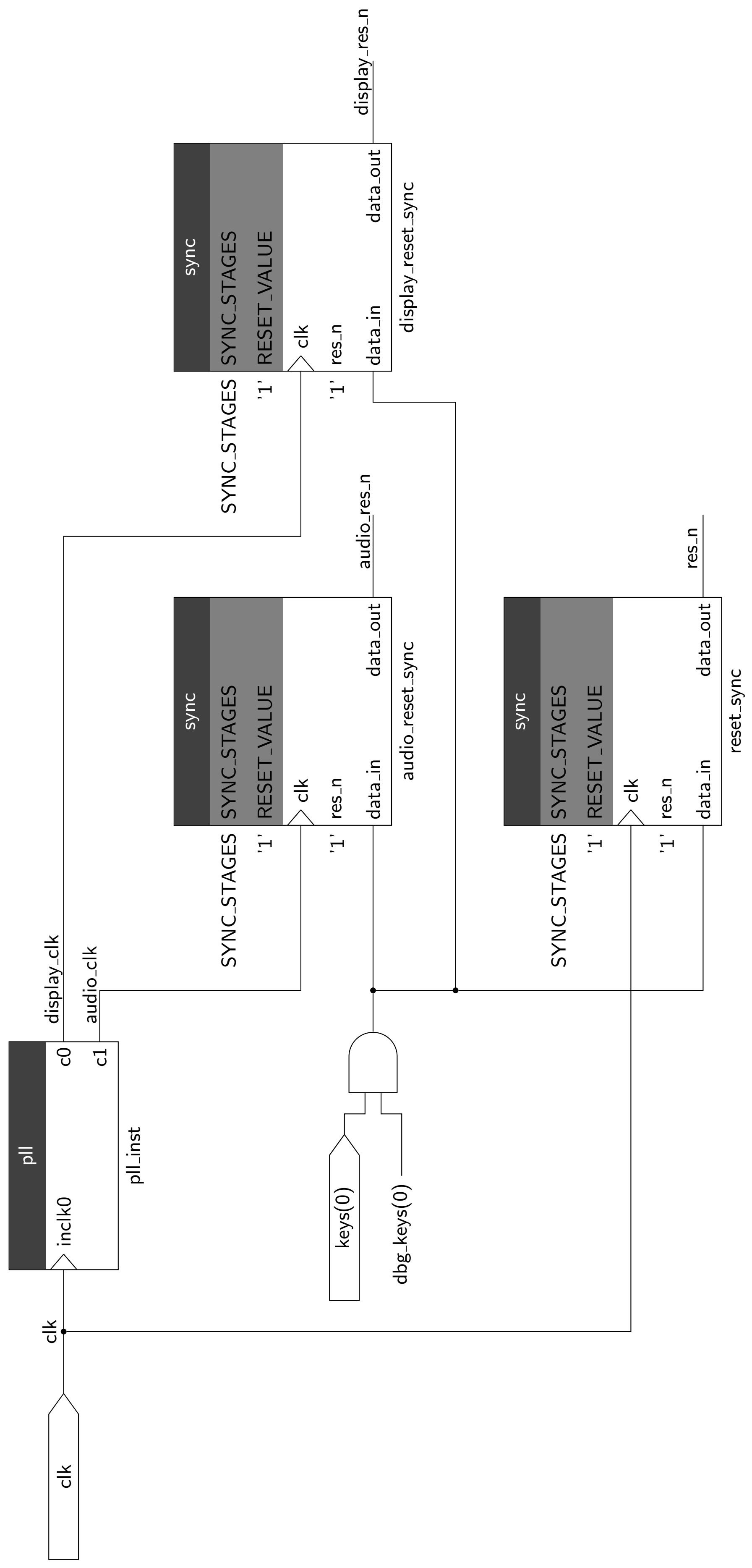


Figure 2.3: Structural system description (part 1)

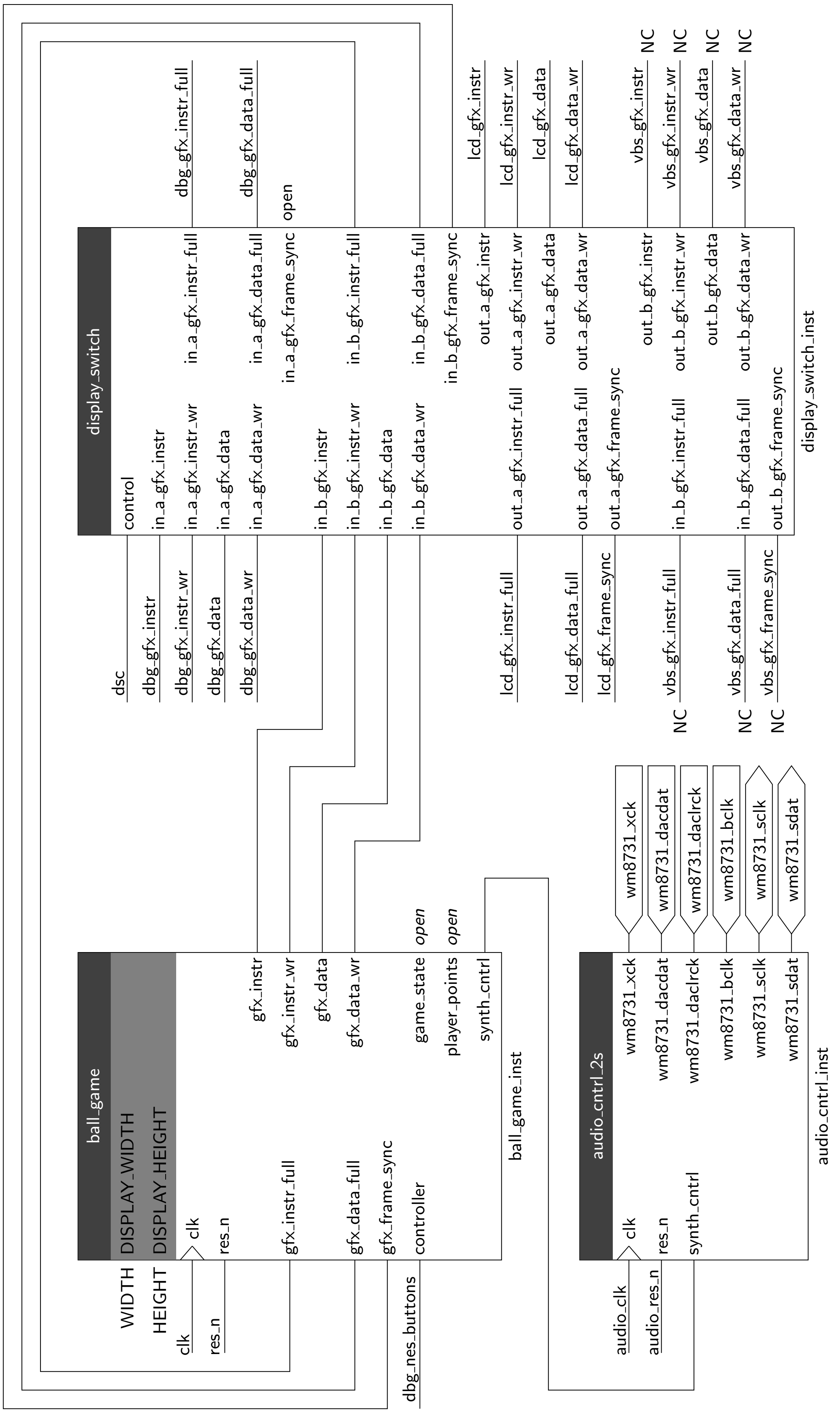


Figure 2.4: Structural system description (part 2)

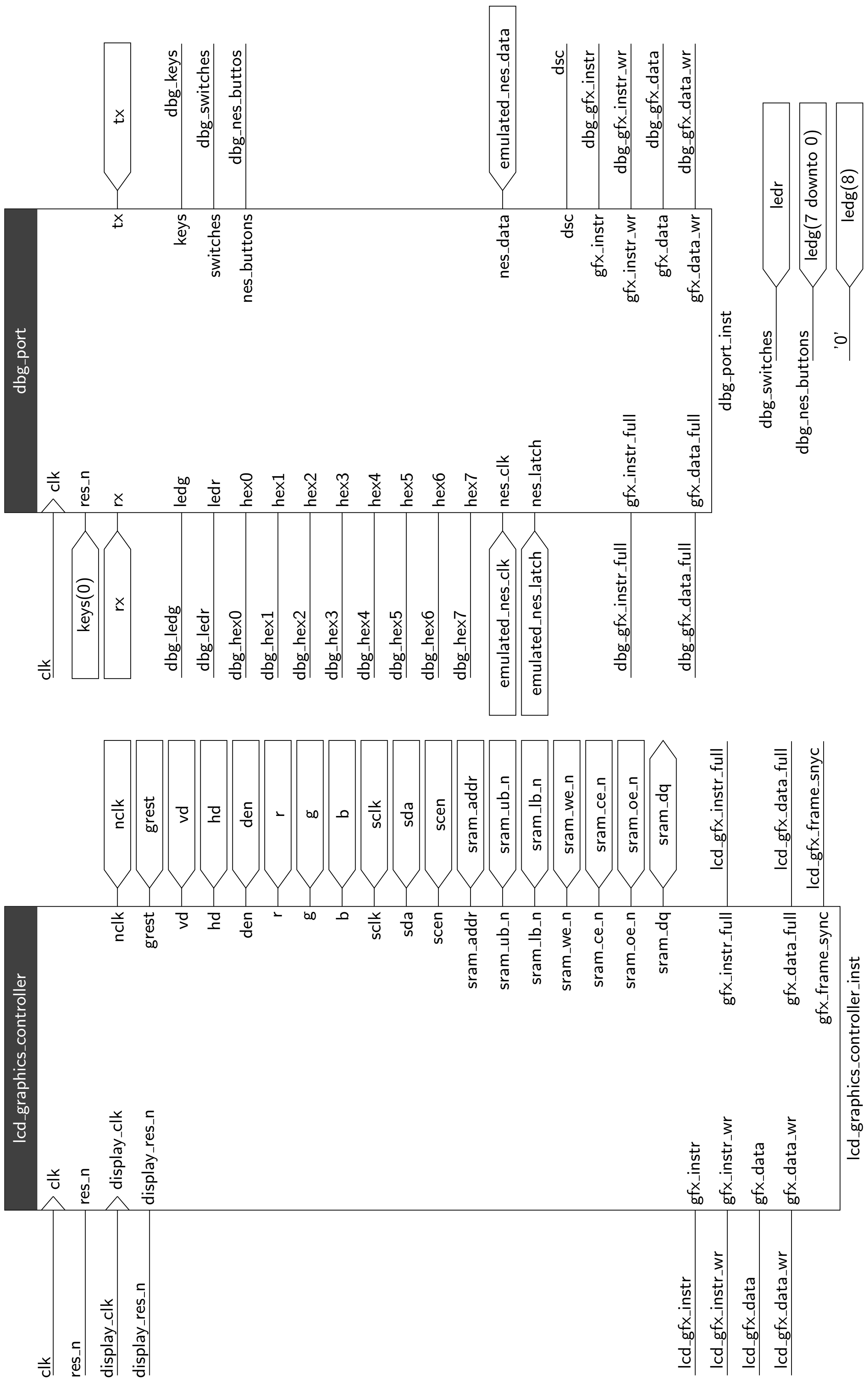


Figure 2.5: Structural system description (part 3)

Task 2: Seven Segment Display I [14 Points]

In this task you will extend your design with a simple combinational module that outputs the current state of the game on the seven segment display of the board. Moreover, it will also display whether the left or right key of the controller is pressed. For this purpose create a new entity called ssd_controller and place it in the file `ssd_controller/src/ssd_controller.vhd`. Table 2.2 specifies the interface of this entity. The clk and res_n inputs are not required for this task, but will be used for Task 7.

Name	Dir.	Width	Functionality
clk	in	1	Global clock signal
res_n	in	1	Global reset signal (low active, not internally synchronized)
game_state	in	game_state_t	The current game state
controller	in	nes_buttons_t	The NES controller input
player_points	in	16	The current player points (unused for now, see Task 7)
hex0	out	7	The right-most seven segment display: unused (switch off)
hex1	out	7	The 2. display: unused (switch off)
hex2	out	7	The 3. display: unused (switch off)
hex3	out	7	The 4. display: unused (switch off)
hex4	out	7	The 5. display: current direction
hex5	out	7	The 6. display: current direction
hex6	out	7	The 7. display: game state
hex7	out	7	The left-most seven segment display: game State

Table 2.2: ssd_controller interface specification

The game_state output of the ball_game module is an enumerated data type, comprised of the values IDLE, RUNNING, PAUSED, GAME_OVER (it is defined in the ball_game_pkg). Since the actual game logic is not yet implemented, the game_state output can be changed using the NES controller inputs as listed in Table 2.3. Note that if neither of the buttons is pressed, the output game state will be determined by a PRNG in the ball_game entity. Table 2.4 shows the patterns that shall be displayed on the outputs hex{6-7} for the different game states.

To display the current direction (i.e., whether the left or right key of the controller is pressed) use the pattern shown in Figure 2.5 for the outputs hex{4-5}. Note that, when the left and right buttons are pressed simultaneously the error code shall be displayed. When neither of the two buttons is pressed both segments shall be switched off.

The outputs hex{0-3} should be permanently switched off for now.

Don't use the all keyword for this task, but create explicit sensitivity lists for your processes (if you have any). Consult the FPGA board manual for more information on how to control

Input pattern	Game state
$\overline{\text{Start}} \wedge \overline{\text{Select}} \wedge \overline{\text{A}} \wedge \overline{\text{B}}$	Random State
Start	IDLE
Select	RUNNING
A	PAUSED
B	GAME_OVER

Table 2.3: Input patterns to activate the different game states

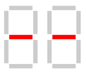
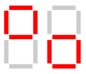
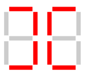
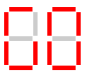
IDLE	
RUNNING	
PAUSED	
GAME_OVER	

Table 2.4: Game state display

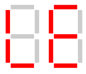
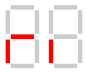
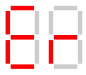
Left button pressed	
Right button pressed	
Left and Right buttons pressed simultaneously	

Table 2.5: Direction Patterns

the individual segments of the seven segment display. After your module is complete, create a package for it (`ssd_controller/src/ssd_controller_pkg.vhd`) and add an instance to the top-level module. Connect the game_state input to the game_state output of the ball_game module and connect the controller input to dbg_nes_buttons. Finally connect the hex{0-7} outputs to the corresponding outputs of the top level entity. The correct pin assignment should have already been configured in Task 1.

Task 3: Behavioral Simulation [8 Points]

In this task you will simulate the top-level entity with the provided testbench in `top/tb/top_tb.vhd`. To automate the compilation and simulation process use the makefile example, provided in the `ram/` directory, to create your own makefile-based simulation flow for the top-level design. The makefile for the top-level entity shall be placed in `top/Makefile`.

To get better acquainted with the tools, you can also create a Questa/Modelsim project using the GUI as outlined in the Design Flow Tutorial. However, this is not needed for the submission or the grading. Your makefile should support at least the targets `compile`, `sim_gui` and `clean`. The `compile` target should compile all required source files (i.e., vhd and vho files) using the Questa/Modelsim compiler (`vcom`). The simulation target (`sim_gui`) should start the graphical user interface of Questa/Modelsim, load an appropriate waveform viewer configuration script to add the relevant signals to the waveform viewer `top/scripts/wave.do` and run the simulation for a few microseconds. Make a simulation showing the signals controlling the SRAM (sram_we_n, sram_ub_n, sram_oe_n, sram_lb_n, sram_ce_n, sram_dq, and sram_addr) as well as the signals of the serial interface of the LCD driver IC (scen, sda, sclk). Answer the questions in the report and provide the required screenshots. The `clean` target should delete all files generated during the compilation and simulation process.

Note that the example makefile of the ram module is just a suggestion. You are free to change the makefile in any way you like or create a completely different implementation altogether. The only requirement is that the targets `compile`, `sim_gui` and `clean` work as specified.

Task 4: Postlayout Simulation [8 Points]

Use the netlist file (.vho) and the timing file (.sdo), which were generated during Task 2 by Quartus[4], for performing a post-layout simulation on the top-level entity. The testbench file used in the behavioral simulation can also be employed for post-layout simulation.

The timing file provides information on the real physical signal delays. Therefore, in contrast to a behavioral simulation, signals do not switch instantaneously after the clock edge. Every single bit of a signal vector switches individually depending on the propagation and routing delays of the corresponding circuitry. Run the simulation long enough in order to take a screenshot of the switching of hex{6-7}. Zoom into the waveform until you can see the different delays of the signals and use two markers to measure

- the duration between the first and the last bit toggling and
- the time between the (last) active clock edge (of clk) and the point in time when the hex{6-7} outputs have stabilized.

Note that the markers must be visible in the screenshot (You may use a single screenshot for both values). You don't have to provide a simulation script for this task, the screenshot is sufficient.

Important: The time resolution for the post-layout simulation must be set to pico seconds. If not set correctly Questa/Modelsim produces an error. Using the command line interface this is achieved using the `-t` argument. In the GUI the time resolution is set in the *"Start Simulation"* window.

Task 5: Testbench Design [10+12+(5) Points]

In this task you will design testbenches for two of the cores provided in the exercise template. Both simulations are purely text-based, i.e., the waveform viewer is not used to examine the results of the simulation, although you can use it to design and debug your testbenches. Furthermore, the simulations shall be controlled by the Makefiles (`prng/Makefile` and `lcd_graphics_controller/Makefile`) similar to Task 3. Both makefiles should support the targets `compile sim` and `clean`. You can, however, add further targets for your use during development. Before you start your work on the testbenches consult the IP Cores Manual to learn about the features and interfaces of the two cores.

(A) PRNG (Pseudo Random Number Generator) [10 Points]
Create a testbench for the prng named prng_tb and place it in the `prng/tb/prng_tb.vhd` file. This testbench should apply a clock (the frequency can be chosen freely) and monitor the output sequence of the prdata signal. Your task is to determine the period of the output sequence generated by the PRNG for different values of seed and record the maximum and minimum periods. After all required seed values (defined later) have been tried the simulation shall be stopped, i.e., the clock shall be switched off and the results shall be reported (and the simulator program shall exit).

As documented in the IP Cores Manual, the PRNG internally uses a 16-bit linear feedback shift register (LFSR), whose initial state and feedback polynomial can be controlled by the seed input. Unfortunately nothing is known about how this seed value is processed and how it affects

[4]Depending on the settings, the Quartus timing analyzer might produce multiple sets of vho and sdo files: with fast/slow timings. For this exercise use the conservative (slow) timing estimates.

the aforementioned parameters. Hence the period of the PRNG (i.e., its internal LFSR) can only be determined by observing the prdata output. To do this in the testbench attach a shift register to the prdata output of the PRNG (see Figure 2.6) and observe its value. Note that the shift register must be 16 bits wide[5], to allow it to capture the complete internal state of the LFSR. Recall that the maximum period of a 16-bit LFSR is $2^{16} - 1 = 65535$.

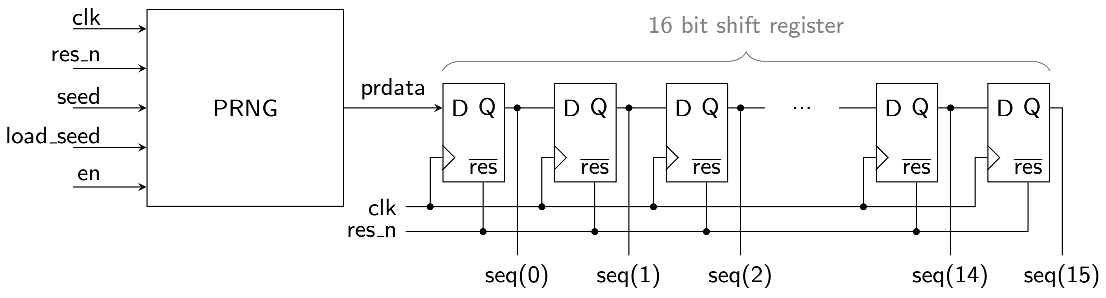


Figure 2.6: PRNG with shift register attached at prdata

To measure the period of the PRNG take a snapshot of the shift register's output (seq in the figure) and count the number of clock cycles until this value appears again. The value or point in time of the initial snapshot does not matter, you only have to make sure that the shift register is completely filled with data from the PRNG when you take the snapshot. This means that you have to wait a certain number of clock cycles after a reset or when you change the seed before taking a snapshot.

Figure 2.7 shows an example simulation for the PRNG initialized with the seed 0x00 (the default seed after startup). It can be seen that in this case the period is 7.

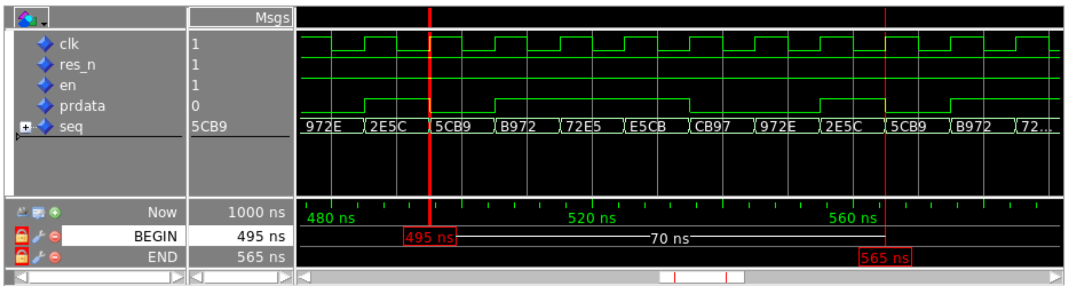


Figure 2.7: Example simulation of the PRNG

The range $[n_a, n_b]$ of seed values to check in your simulation is based on your matriculation number.

$$n_a = (([\text{Your Matriculation Number}] \mod 15) + 1) << 4, n_b = n_a + 15$$

For each of the 16 seed values your testbench should produce an output line reporting the seed value itself and the measured period.

When `make sim` is executed in the `prng/` directory, the simulation shall be performed and the cycle count should be reported. After that the simulator should exit. The graphical user interface of Questa/Modelsim should not be opened during this process. You can either use the textio package or simple report statements for your output, which should look something like this:

```
seed: [SEED0], period: [PERIOD0]
seed: [SEED1], period: [PERIOD1]
```

[5]In fact it could also be larger, but then it would record the additional bits unnecessarily.

```
[...]
seed: [SEED15], period: [PERIOD15]
min period: [MIN_PERIOD], max period: [MAX_PERIOD]
```

If report statements are used additional information will be printed (e.g., the simulation time step when the report occurred), this is completely fine for your submission.

(B) LCD Graphics Controller [12 Points]
Create a testbench for the lcd_graphics_controller named lcd_graphics_controller_tb and place it in `lcd_graphics_controller/tb/lcd_graphics_controller_tb.vhd`. This testbench shall generate the necessary clock signals with the right frequencies, set a user pattern (you are free to design one) and use this user pattern to draw two nxn rectangles at $(x, y) = (0, 0)$ and $(x, y) = (400-n, 240-n)$ with $n = 16$.

From the IP Cores Manual we know that the lcd_graphics_controller uses the external SRAM of the board to store its frame buffer(s). Hence implement a simple model of this SRAM chip (the datasheet is available in TUWEL) that can be attached to the SRAM interface signals of the lcd_graphics_controller in the testbench. We already prepared an entity named sram in the file `lcd_graphics_controller/tb/sram.vhd`. Be sure to not change the interface of this entity in any way!

Your SRAM model shall have an internal memory reflecting the size of the real SRAM chip on the FPGA board, where data written by the graphics controller is recorded. You don't have to support every write operation mode listed in the datasheet. The only constraint is that your SRAM model has to work with the lcd_graphics_controller. For the sake of simplicity initialize every location of the internal memory to zero. Note, however, that this is not the case with real SRAM, which powers up to a random state.

In order to visualize the data stored in your SRAM, the entity of the SRAM module features the write_file input. Whenever the sram sees a rising edge on this signal it shall dump the image with the resolution defined by the inputs width and height which is stored at the location specified by the input base_address to an image file. For that purpose we use the ASCII version of the Portable PixMap[6] format (magic number: P3). The file names of the dumped images shall be `sram_dump_[N].ppm`, where `[N]` is a number increased with every image dump, starting at 0. The image files should be placed in the directory specified by the generic OUTPUT_DIRECTORY.

The layout of the image in memory is line based. The address of a pixel in memory can be calculated with the following formula:

$$\text{address\_of}(x, y) = \textsf{base\_address} + y * \textsf{width} + x$$

The pixel $(x, y) = (0, 0)$ is stored at base_address, while pixel $(1, 0)$ is stored at base_address$+1$. The location base_address+width refers to the pixel $(0, 1)$.

Figure 2.8 shows the expected output of the simulation. For better visibility, we used larger rectangles ($n = 64$) and a different primary color.

The lcd_graphics_controller_tb shall wait for the issued graphics instructions to be completed and then start a memory dump. Make sure that the image is saved in the same directory where the Makefile of the lcd_graphics_controller module resides (the name of this image shall be `sram_dump_0.ppm`). The testbench shall consider the graphics controller to be finished when there was no write operation for 100 clock cycles. After that the simulation shall be stopped and the simulator shall exit. The whole simulation process should again be started by executing the `make sim` command (without the Questa/Modelsim GUI opening).

[6]`https://en.wikipedia.org/wiki/Netpbm`

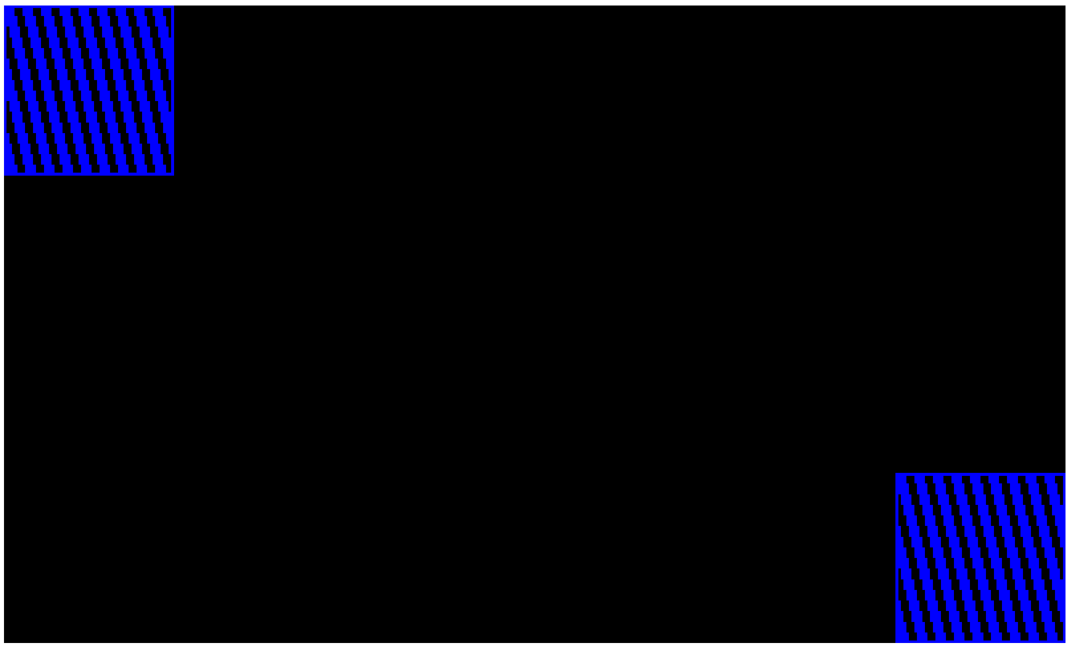


Figure 2.8: Sample SRAM dump ($n = 64$ and blue primary color for better visibility)

While solving this task, it may be beneficial to write only a few pixels to the SRAM for testing purposes and only change to the required image after the SRAM model works. The netlist of the lcd_graphics_controller is quite large, which leads to quite long simulation times with the free version of ModelSim. Hence it is also a good idea to use the TILab computers with the licensed (i.e., faster) version of Questa/Modelsim.

Bonus Task: SRAM reads [5 Points]
Extend your SRAM implementation to also support read operations, i.e., if the lcd_graphics_controller performs a read operation supply the correct data from your internal memory. Add a screenshot of a read access to your lab protocol.

Task 6: NES Controller [20 Points]

In this task you will implement the NES controller interface to be able to process inputs from a controller attached to the board. The original NES as well as the SNES controller are based on a simple parallel-in/serial-out shift register (like the CMOS 4021 shift register). The controller uses a serial interface consisting of three signals, as shown in the timing diagram in Figure 2.9.

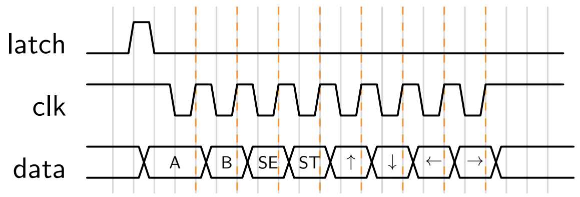


Figure 2.9: NES controller serial interface protocol

To start a new transmission reading the state (pressed/not pressed) of each button, a pulse must be generated at the latch signal. This pulse causes the shift register inside the controller to *latch* the current state of every button (parallel load). Now a clock signal can be applied at the clk input to serially shift out the button state over the data signal. Note that the data signal changes with the rising edge of clk. However, to be on the safe side we only sample it at the next rising edge indicated by the orange lines. This way the input signal at nes_data has the maximum time to stabilize (setup constraint).

Implementation Create an entity named nes_controller and place it in the file `nes_controller/src/nes_controller.vhd`. The required generics and the port signals are described in Tables 2.6 and 2.7, respectively.

Name	Functionality
CLK_FREQ	Actual clock frequency of the *clk* signal given in Hz
CLK_OUT_FREQ	The desired clock frequency that should be generated for the nes_clk signal in Hz. Don't use frequencies higher than 1 MHz.
REFRESH_TIMEOUT	The timeout in clk cycles the controller should wait in between button readouts. Set this generic to the equivalent of 8 ms.

Table 2.6: nes_controller generics description

Name	Dir.	Width	Functionality
clk	in	1	Global clock signal
res_n	in	1	Global reset signal (low active, not internally synchronized)
nes_latch	out	1	The latch signal used to load the shift register in the NES controller with the current state of all buttons.
nes_clk	out	1	The clock signal for the shift register in the NES controller.
nes_data	in	1	The data from the shift register in the NES controller.
button_state	out	nes_buttons_t	This is a record type defined in the nes_controller_pkg package.

Table 2.7: nes_controller signal description

The state diagram of the nes_controller is shown in Figure 2.10. Implement a state machine according to this specification. The initial state is WAIT_TIMEOUT.

Note that the button_state output is updated only in the WAIT_TIMEOUT state. Otherwise it must hold the value of the last known button state. This implies that a register must be used to buffer this output value. However, you don't need (and shall not use) registers for the nes_clk and nes_latch outputs. Don't forget to initialize all used registers during reset.

The BIT_TIME constant used in Figure 2.10 is the duration of a single bit transmitted on the nes_data line in clock cycles of clk. You have to calculate this constant from the generics.

Simulation: After implementation create a testbench for the nes_controller entity and place it in `nes_controller/tb/nes_controller_tb.vhd`. Add a makefile to control the simulation process (again implement the targets `compile`, `clean` and `sim_gui`).

The `sim_gui` target shall start a graphical (behavioral) simulation that shows the transmission of two button state information frames. The waveform viewer shall show all inputs and outputs of the controller as well as all internal state signals (FSM state, counters). Add a screenshot to your lab protocol. Be sure to expand the button_state output in the waveform viewer such that the individual record elements are visible in the screenshot.

Your testbench shall simulate two button state transmissions, i.e., it has to wait for an appropriate trigger on the nes_latch signal and then generate the appropriate input data on the nes_data

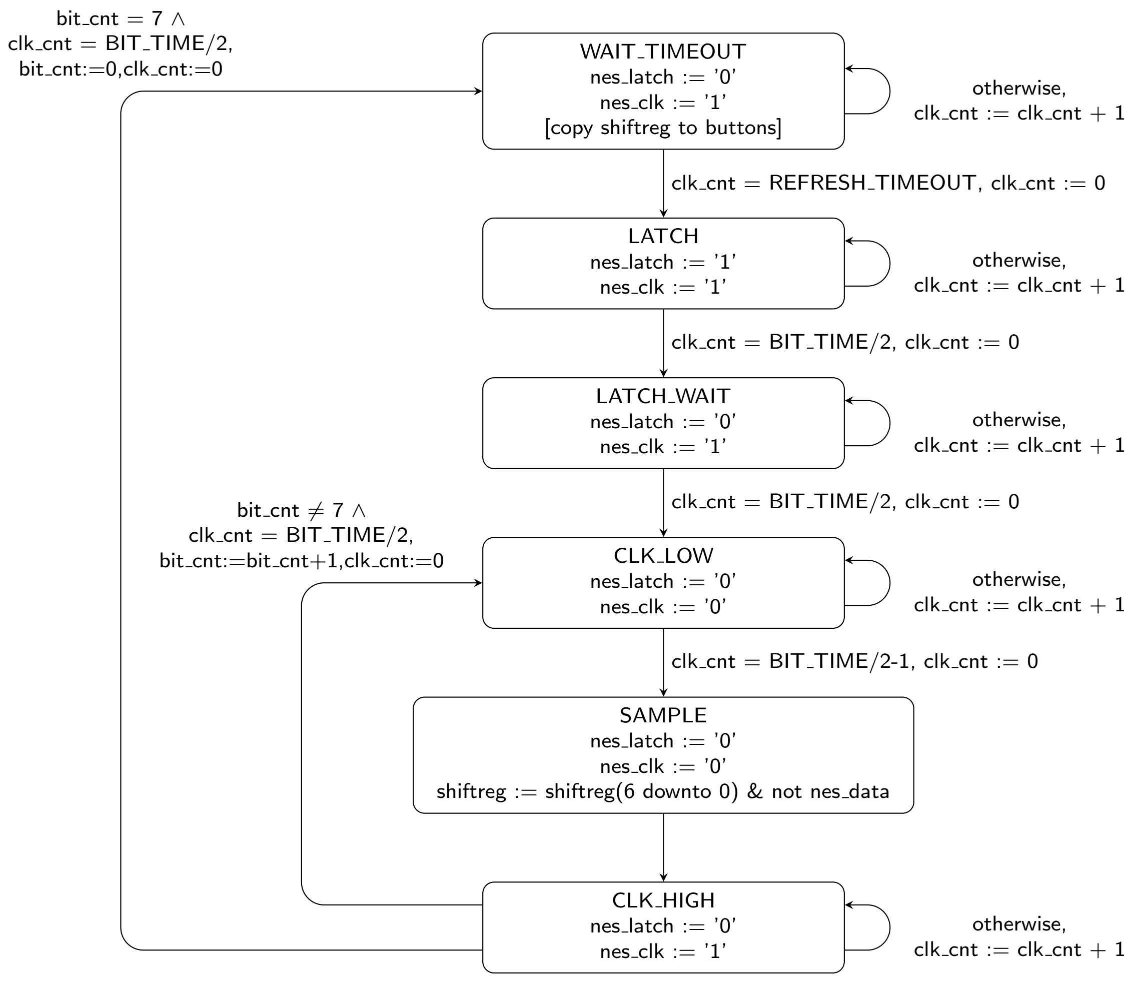


Figure 2.10: nes_controller state diagram

signal. Both transmissions shall be visible in the screenshot. Hence set the REFRESH_TIMEOUT to a rather low value.

To calculate the two button state values b_0 and b_1 in your testbench take the binary representation of your matriculation number modulo 2^{16} and use the lower 8 bits for b_0 and the upper 8 bits for b_1. Should one of these values only contain zeros or ones, take an alternating pattern instead. The values b_0 and b_1 should be mapped to the vector given by the individual buttons signals (A,B,Select,Start,↑,↓,←,→), where A is the MSB.

Example:

$$mn = 123456$$

$$123456 \mod 2^{16} = 57920 = 0\text{xE}240 \Rightarrow b_1 = 0\text{xE}2, b_0 = 0\text{x}40$$

System Integration If your simulation shows that your design works, add an instance of nes_controller to your top-level design and test it in hardware. Because of this year's remote working environment you will, of course, not be able to use a "real" controller with your design. Moreover, there isn't even a real controller connected to the boards. You might already have noticed the NES related signals that connect the dbg_port to inputs/outputs of the top-level entity. These inputs/outputs are externally fed back into other GPIO pins of the FPGA (where normally the real NES controller would be attached). Using these signals the dbg_port is able to emulate the behavior of a real controller

Figure 2.11 shows how the controller is connected to the board's GPIO connector. Consult the FPGA board manual for the pin locations and use the Pin Planner in Quartus the make the appropriate configurations. Be sure to select the correct I/O Standard (3.3-V LVTTL).

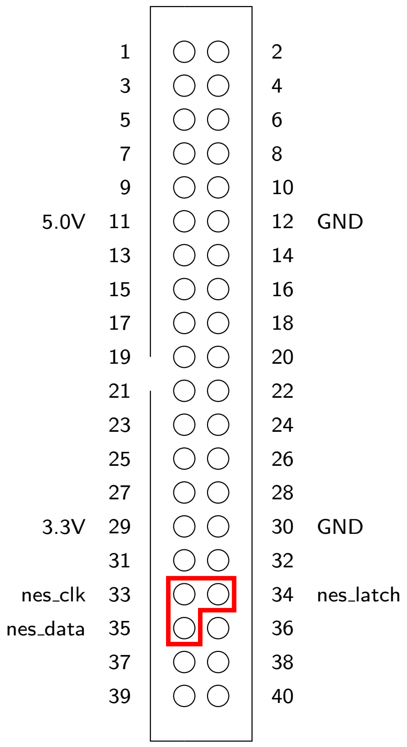


Figure 2.11: Physical controller/board interface

Reading the state of the emulated controller will return the value currently output by the dbg_port at the nes_buttons output. Hence after integration of your nes_controller simply replace the dbg_nes_buttons signals with a signal connecting to the nes_buttons output of your controller. Additionally set the nes_buttons output of the dbg_port instance to open.

Task 7: Seven Segment Display II [20+(5) Points]

In this task you will extend the ssd_controller, such that it is able to show a decimal representation of the current player points on hex{0-3}. The points shall be read from the player_points input.

For that purpose design and implement a state machine that converts the player_points from their binary representation to a BCD value [7] and then convert this BCD value to the corresponding seven segment symbols (see Figure 2.12). Initially the FSM should start in state where it **waits** for a change at the player_points input and then starts the conversion process. After completion of the conversion the state machine either directly returns to this initial state or lets the new output value first blink for a certain number of times and then returns to the initial state. The decision which path to take is based on the difference between the new and the old player_points value. If the new value exceeds the old one by 25 points or more, the hex display should blink. While the hex display is blinking no new player_points value shall be read or converted.

[7] `https://en.wikipedia.org/wiki/Binary-coded_decimal`

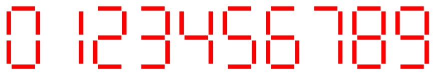


Figure 2.12: Seven segment decimal number patterns

Add the generics shown in Table 2.8 and use them in your implementation. Configure the instance in the top module, such that BLINK_COUNT is set to three and BLINK_INTERVAL is set to a value corresponding to 0.25 seconds.

Generic	Description
BLINK_INTERVAL	The amount of time in clock cycles the hex display should be on/off.
BLINK_COUNT	The number of times the hex display should blink. A value of one indicates that after the conversion the hex display shall on for BLINK_INTERVAL and then off for BLINK_INTERVAL. After that it should stay on.

Table 2.8: Constants

The conversion itself **must not** be implemented using a division operation, but by successively subtracting decimal powers (i.e., once every clock cycle) from the binary value. Start by subtracting 1000 until the value is smaller or equal to 999. By counting the number of times 1000 could be subtracted, the thousands digit is obtained. Now repeat the process by subtracting 100 to obtain the hundreds digit and finally 10 to obtain the tens digit. The rest corresponds to the ones digit. Since we are dealing with 16-bit unsigned number the highest value that could appear at the input is is 65535. However, the highest value we can display using 4 decimal places is 9999. Hence, if the input exceeds this number simply display four dashes, as shown in Figure 2.13.

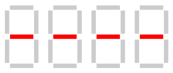


Figure 2.13: Output pattern for number greater 9999

Again, don't use the all keyword for this task, but create explicit sensitivity lists for your processes. Make sure that all required signals are contained in these lists and don't add superfluous signals!

Implement a testbench for your design (place it in `ssd_controller/tb/ssd_controller_tb.vhd`) and add a makefile to control the simulation process (again implement the targets `compile`, `clean` and `sim_gui`). The `sim_gui` target should open the GUI of Questa/Modelsim and load an appropriate waveform file showing all inputs and outputs of the ssd_controller core, as well as the internal state variables (this also includes counters). In the testbench use the following value for player_points.

$$\text{player\_points} = 1234 + ([\text{Your Matriculation Number}] \mod 500).$$

Include a simulation screenshot of the conversion process in your report. Make sure that the values of all intermediate states are visible in the simulation screenshot. You can make multiple screenshots if this is not possible with one.

The ball_game module increments the player_points output every second. When the A button on the controller is presses, 50 is added to the player_points.

Note that after finishing this task your ssd_controller core should still display the game state at hex{6-7}, and the direction at hex{4-5} as implemented in Task 2.

Bonus Task: Animation [5 Points]
Animate the outputs hex6 to hex7 whenever the game is in the RUNNING state. Use the patterns shown in Figure 2.14. Add a generic to your entity that specifies how many cycles the individual animation steps should be shown by your core. In the top-level design, configure this value to the equivalent of one second. Include a screenshot in your lab report showing a simulation run of your core generating the outputs of one complete animation cycle. Set the generic to 4 clock cycles for this simulation.

Note that the points for this task are only awarded if the other parts of your ssd_controller core are fully implemented.

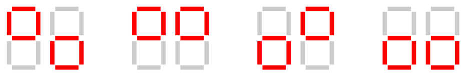


Figure 2.14: Animation steps for the RUNNING state (Bonus Task)

2.4 Submission

To create an archive for submission in TUWEL execute the submission_exercise1 makefile target of the template we provided you with.

```
cd path/to/your/project
make submission_exercise1
```

The makefile creates a file named submission.tar.gz which should contain the following information.

- Your lab protocol as PDF
- The source code of **all** IP cores
- The source code of the PLL
- The source code of your top-level module
- The source code and testbenches of your IP cores
- Your Quartus project (don't forget a cleanup!)
- The SDC file containing the clock definition
- Your makefiles to start the individual simulations

Most of these points are automatically checked by the submission script. If the script reports an error, no archive will be created. Carefully check the warnings that are generated. The created archive should have the following structure.

```
submission.tar.gz
├─ report.pdf ........................................ Your lab report
└─ vhdl ................................. The source code of all IP cores
   ├─ top
   ├─ nes_controller
   ├─ ssd_controller
   └─ [... all other IP cores]
```

Make sure the submitted Quartus project compiles and that your makefiles are working. All submissions which can not be compiled will be graded with zero points! **Don't create the archive manually**. If you have problems running the makefile target consult a tutor.

3 Exercise II (Deadline: 30.04.2021)

3.1 Overview

In this exercise you have to extend the existing design such that (i) the complete game logic is implemented and (ii) the VBS graphics controller can display the game's output using a regular analog baseband TV signal.

Please note that Tasks 1 and 2 do not depend on each other, and thus enable you to work on them in any order or even in parallel. Nevertheless, before you start we highly recommend (i) to read the *whole* assignment and (ii) to run the reference solution, to get an intuition how the game is supposed to behave.

3.2 Required and Recommended Reading

All documents are available in TUWEL.

Essentials (read before you start!)

- Design flow tutorial
- VHDL introduction slides (Hardware Modeling)
- VHDL Coding and Design Guidelines

Consult as needed

- IP Cores Manual
- Datasheets and manuals
- SignalTap manuals

3.3 Task Descriptions

Task 1: VBS Graphics Controller [50 Points]

In this task you will implement a module named vbs_graphics_controller that interfaces with the board's ADV7123 digital-to-analog converter (DAC) to generate a video baseband signal (VBS). A VBS is an analog unmodulated black and white TV signal, which will be used as an alternative to the video output on the LCD (controlled by the lcd_graphics_controller). For that purpose the module has to provide the exact same interface to the remaining system as the lcd_graphics_controller. It is also supposed to support the same resolution (400x240 pixels), however, with a lower color depth (see below). The template already provides

- an entity specification in `vbs_graphics_controller.vhd`
- and a package template in `vbs_graphics_controller_pkg.vhd`.

Don't change the interface or the name of the entity. Tables 3.1 and 3.2 show the generics and signals of the vbs_graphics_controller entity.

To implement this module you can use a similar overall structure as described for the lcd_graphics_controller. Use the cores provided by the gfx_util_pkg package to implement the

Name	Functionality
CLK_FREQ	The frequency of the input clock.

Table 3.1: vbs_graphics_controller generics description

Name	Dir.	Width	Functionality
clk	in	1	Input clock signal
res_n	in	1	Low-active reset signal
gfx_frame_sync	out	1	The frame synchronization signal
gfx_instr	in	GFX_INSTR_WIDTH	The actual instruction
gfx_instr_wr	in	1	The write signal of the instruction FIFO
gfx_instr_full	out	1	The full signal of the instruction FIFO
gfx_data	in	GFX_DATA_WIDTH	The data associated with the instruction (coordinates, colors, etc.)
gfx_data_wr	in	1	The write signal of the data FIFO
gfx_data_full	out	1	The full signal of the data FIFO
vga_clk	out	1	The clock signal used for the communication with the DAC
vga_r	out	8	Data for the red output channel (unused, drive with constant 0)
vga_g	out	8	Data for the green output channel
vga_b	out	8	Data for the blue output channel (unused, drive with constant 0)
vga_sync_n	out	1	Control signal to set the output signal to the synchronization level
vga_blank_n	out	1	Control signal to set the output signal to the blank (black) level

Table 3.2: vbs_graphics_controller signal description

rasterizer sub module. The template also already provides a frame_reader module in the `frame_reader.vhd` file which you can use for your implementation.

Internal Interface The internal interface must match that of the lcd_graphics_controller, which is specified in detail in the IP Cores Manual. The lcd_graphics_controller implements a simple instruction interface, with an 8-bit instruction signal (gfx_instr) and a 16-bit data signal (gfx_data). Your graphics controller must support all of the instructions and features specified in the IP Cores Manual. The only difference to the lcd_graphics_controller is that the vbs_graphics_controller only supports 4 shades of gray and hence only needs 2 bits for the "color" information. This means that for the data associated with the `SET_COLOR` instruction only bits 0 and 1 are used. The values 00 and 11 correspond to the colors black and white, respectively, while 01 and 10 should yield some shade of gray of your choosing.

Use instances of the FIFO from the ram_pkg package to buffer incoming instructions and data and expose their write ports to the port signals gfx_instr_* and gfx_data_* of your graphics core. Internally you can then read the instructions and data from these FIFOs and perform the appropriate actions.

Video Memory The lcd_graphics_controller uses the external SRAM on the FPGA board to store its frame buffers. For the sake of simplicity the vbs_graphics_controller uses only the embedded memory of the FPGA. Because this resource is quite limited we can only support 2 bits per pixel. Use the dual-port RAM from the ram_pkg package to implement the video RAM.

Using this type of memory comes with the added benefit that no arbiter is required to manage simultaneous write operations from the rasterizer and read operation from the frame_reader. Both components use completely separate ports of the video RAM and can, hence, access it completely independent from each other.

External Interface The external interface has to communicate with the ADV7123 DAC to generate the appropriate video signal. As can be surmised from the datasheet and its utilization on the DE2-115 FPGA board this DAC is actually intended to implement a VGA interface. However, it has all the necessary features to generate an analog TV signal as well. For that purpose, we will simply use one of the three output channels of the DAC (namely the green one, i.e., vga_g) and set the other channels (i.e., vga_b and vga_r) to constant zero. The I/O mapping for all signals connected to the ADV7123 is already preconfigured in the template.

A video baseband signal (also called composite video signal[8] is usually transmitted over a single wire pair using an RCA[9] connector. The analog signal operates in a voltage range of 0 V to ≈ 1 V, where 1 V corresponds to the white level and 0.3 V to the black level. Voltages below 0.3 V are only used for synchronization information.

Video information is transmitted frame by frame with a rate of 25 frames per second (FPS) [10] As shown in Figure 3.1, a frame consists of 625 scanlines and can be further subdivided into two so-called fields. Since frames are transmitted in an interlaced fashion, first all even lines are transmitted in the first field and then the odd lines are transmitted in the second field. Note that only 288 lines per field contain actual (visible) image information, the other lines are used solely for synchronization. This is the reason why PAL is often said to have a "resolution" of 576i.

A single scanline has a length of 64 µs and always starts with a synchronization pulse which goes down to 0 V (synchronization level). Depending on the type of the line this synchronization pulse can have a length of 2.35 µs (e.g., line 4), 4.7 µs (e.g., line 6) or 27.3 µs (e.g., line 1). A line always ends on the black voltage level (≈ 0.3 V). A visible scanline (i.e., a scanline containing actual image information) always starts with a (horizontal) synchronization pulse (4.7 µs) followed by 5.7 µs of the black voltage level (back porch), which is then followed by an analog waveform between 0.3 and 1V encoding the brightness information of the line. Finally the signal level returns to the black level for 1.65 µs (front porch) to end the line.

For our purpose in this exercise we are not going to deal the hassle of interlaced video transmission and handle each field as if it was a separate frame. This means that we output 50 FPS, i.e., the frame buffer in the video RAM is read 50 times per second. The vertical resolution supported by the vbs_graphics_controller is 240 lines. Since one field has 288 visible lines we are only going to use a portion of that, the unused lines shall simply stay black. This means that we use lines 47-286 (field 1) and lines 359-598 (field 2) for our application. Note that it does not matter if you are off by one with the start of the frame in your solution, just make sure that the image is vertically centered. Use the signal pattern of line 6 for the visible lines that are not used in this exercise (e.g., lines 23-47).

To produce the horizontal resolution of 400 pixels divide the display area (gray area in Figure 3.1) into 400 equally sized sections, where you keep the output level (produced by DAC) constant. You may also add black sections at the left and right side of the image to simplify this division.

[8] `https://en.wikipedia.org/wiki/Composite_video`

[9] `https://en.wikipedia.org/wiki/RCA_connector`

[10] This actually depends on the transmission standard. PAL systems use 25 FPS (except for PAL-M) where NTSC uses ≈ 30 FPS. All information presented from here on is compatible with (25 FPS) PAL systems.

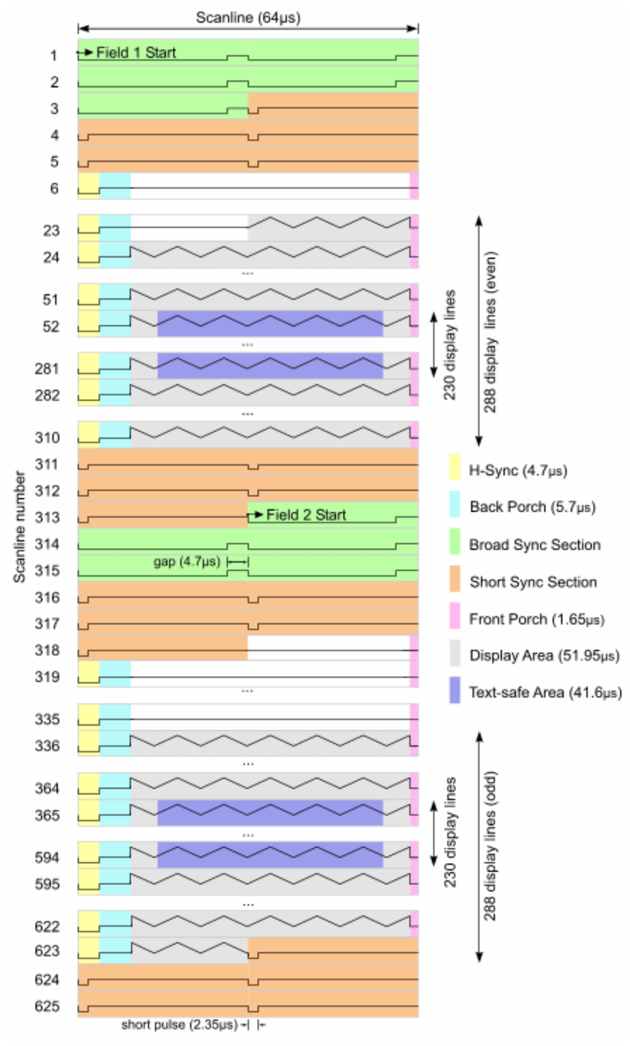


Figure 3.1: VBS frame format (Source: `http://www.batsocks.co.uk/readme/video_timing.htm`)

Use the full 50 MHz to drive the vga_clk signal of the DAC (i.e., don't use an additional PLL to generate a different clock signal). To generate the 0 V and 0.3 V output levels (required e.g., for the synchronization scanlines and the front and back porch sections of visible lines) the signals vga_blank_n and vga_sync_n must be used. See the DAC's datasheet for details.

The template provides a test pattern generator (tpg) in the `tpg.vhd` file, which you can use to test your VBS signal without the need to implement the rest of the graphics controller. To interface with the tpg only two signals (pix_rd and pix_data) are necessary. Whenever pix_rd is asserted, pix_data will output the next pixel value in the following clock cycle starting with the pixel at $(x, y) = (0, 0)$(see Figure 3.2). When $400 * 240 = 96000$ pixels have been read the tpg starts at $(x, y) = (0, 0)$ again. Note that the signals pix_rd and pix_data of the frame_reader core behave in exactly the same way.

Every board/remote work place in the lab is equipped with a video grabber connected to the FPGA board. You can access it using the `signal` stream. Figure 3.3 shows how the test pattern should look like. The outer white frame is a rectangle located at $(x, y) = (0, 0)$ with a width of 400 and a height of 240 pixels. It marks the visible bounds of the generated image. In the center are 4 squares with a width of 16, 32, 48 and 64 pixels showing the different brightness levels.

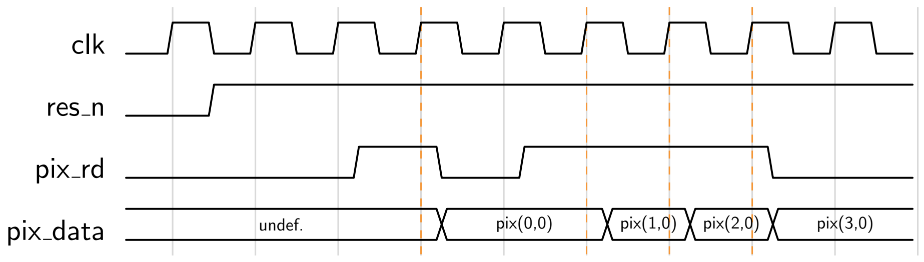


Figure 3.2: test pattern generator interface protocol

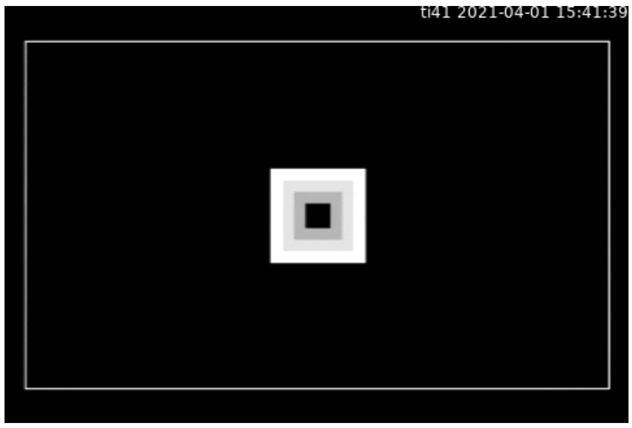


Figure 3.3: VBS test pattern as generated by the test pattern generator

Testing To test all features of your graphics controller you can use the `--gfx` command line argument of the `remote.py` script. The commands below demonstrate how to generate the test pattern of Figure 3.3 using graphics instructions instead of the `tpg`.

```
remote.py --gfx 0x01 # issue a clear screen instruction
remote.py --gfx 0x51 3 # set the primary color to "11"
remote.py --gfx 0x98 0 0 400 240 # draw the white frame
remote.py --gfx 0x9f 168 88 64 64 # draw the white box in the center
remote.py --gfx 0x51 2 # set the primary color to "10"
remote.py --gfx 0x9f 176 96 48 48 # draw the gray box in the center
remote.py --gfx 0x51 1 # set the primary color to "01"
remote.py --gfx 0x9f 184 104 32 32 # draw the gray box in the center
remote.py --gfx 0x51 0 # set the primary color to "00"
remote.py --gfx 0x9f 192 112 16 16 # draw the black box in the center
```

Note that all values supplied to the `remote.py` script can make use of the bit-wise OR operator (i.e., |-operator). Hence the following two commands are equivalent.

```
remote.py --gfx 0x51 "0x2 | 0x1"
remote.py --gfx "0x50 | 0x1" 3
```

Note, that after the FPGA board has been programmed, the display_switch is configured such that the ball_game module is connected to the lcd_graphics_controller and the dbg_port is connected to vbs_graphics_controller. If you want to test how the lcd_graphics_controller reacts to certain commands you can reverse this assignment by using the `--dsc` command line argument of the `remote.py` script.

```
remote.py --dsc 0 # ball_game -> vbs_graphics_controller
remote.py --dsc 1 # ball_game -> lcd_graphics_controller
```

Oscilloscope Measurement Perform oscilloscope measurements to demonstrate the correctness of the VBS signal you are generating. For that purpose, configure the hardware to provide the test pattern presented in Figure 3.3 (either by using the `tpg` or by issuing the appropriate commands to your design using the `remote.py` script). The output of the DAC is connected to channel 1 of the oscilloscopes located at the hosts `ti40` and `ti41`, whereat the `oscilloscope_ui.py` script can be used to interact with it.

Make a measurement of scanline 3 and use the cursors of the oscilloscope to validate the timing. Additionally, make a measurement of one of the scanlines in the center of the test pattern, showing all 4 brightness levels. Consult the report template to learn about the exact screenshots that are required.

You can use the TV trigger to easily step through all the lines of your VBS signal. See the help of the `oscilloscope_ui.py` tool for further details.

Task 2: Fully implement the Ball Game module [50 Points]

For Exercise I we supplied you with an architecture for the ball_game module (`ball_game_ex1.vhd`), which only implemented very basic functions, like moving a ball using the controller, generating data for the player_points and game_state outputs as well as playing simple sounds when certain controller buttons are pressed.

In this task you will implement your own ball_game architecture which shall provide the actual game logic. The architecture must be called ball_game_ex2 and must be placed in the file `ball_game/src/ball_game_ex2.vhd`.

The general concept of the game you should implement is the following: The player controls a ball that “falls” down a well and is blocked by upwards moving bricks (i.e., rectangular blocks). It is only possible to control the sideways movement of the ball. Figure 3.4 shows a screenshot of the game.

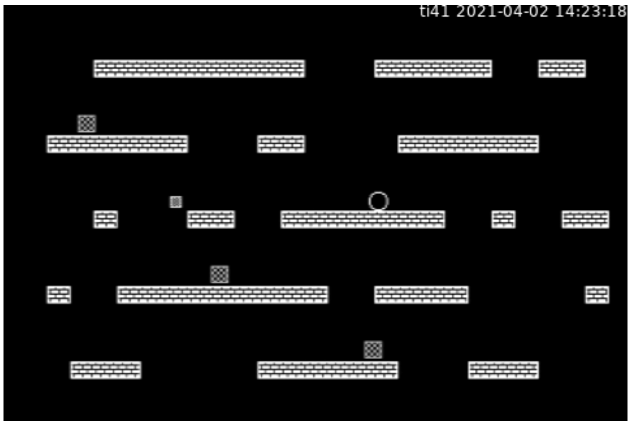


Figure 3.4: Ball game screenshot

The width and horizontal position of bricks are randomly generated (see below). However, they always appear in rows with a constant distance in between them. The player has to avoid touching the upper and lower edge of the screen. Doing so ends the game (game over). The left and right screen border simply block the player, touching them has no effect. To score points while moving downwards, the player has to collect items that appear on the bricks. There are 3 different types of items, which each yield a different amount of points. After specific time intervals the game’s pace is increased, which gradually makes it more difficult.

Game Behavior Figure 3.5 shows a state diagram describing the overall behavior of the game logic and the different states it can be in. The current state should be reflected by the game_state output of the ball_game module. Note that the game states don't necessarily directly correspond to a state machine in your code with exactly these states.

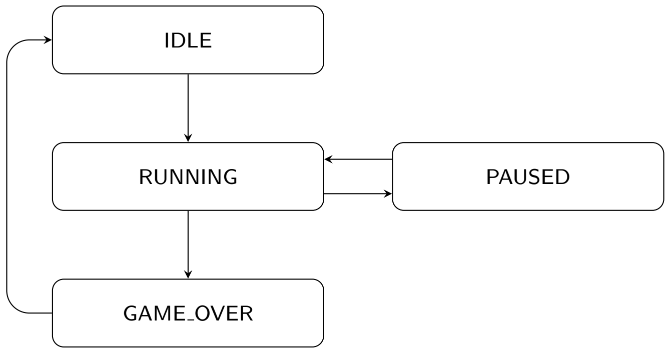


Figure 3.5: Ball game states

After the reset the game starts in the IDLE state, where it should display a set of randomly placed bricks arranged in four rows. In this state the bricks don't move upwards, but stay static. Define and use a user pattern to draw the bricks (the design is completely up to you). Figure 3.6 shows a possible example. The player_points output shall be reset to 0 in this state.

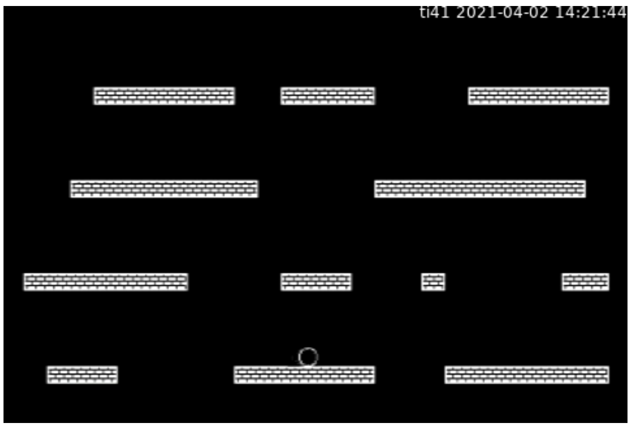


Figure 3.6: Ball game screenshot (IDLE state)

In the IDLE state the ball is *not* controlled by the player but shall move randomly through the randomly generated brick arrangement. This means that on each frame the ball should start/stop moving and/or change direction with a certain probability. Eventually, when the ball hits the lower screen border it should be re-positioned at the top again. Simply reset the player's y coordinate to zero in this case. Pressing Start on the NES controller switches the game into the RUNNING state.

When the RUNNING state is entered the screen is cleared, a single brick row is added at the bottom of the screen and the player is placed at the top of the screen. The player will immediately fall down towards the brick row, while the row itself moves upwards. The sideways movement of the ball can now be controlled using the Left and Right buttons of the NES controller. New brick rows are always added at the lower edge of the screen. After the currently lowest brick row moved upwards sufficiently far a new row is added. Choose a distance such that 5 rows fit on the screen. As soon as the top edge of a brick collides with the upper edge of the screen the brick is removed from the game.

Whenever a new row is added, one item is placed somewhere on that row (it does not matter if it is placed on a brick or if it "hovers" in the space between the bricks). There are three different items types, that are each worth a different number of points. You are free to chose appropriate point values but make sure that the highest value is 25 such that the ssd_controller will let the hex displays blink when the respective item is collected. The items shall be visually distinguishable, hence, use different patterns and sizes and/or colors. When the player collides with an item, the item shall immediately be removed from the game and the points shall be added to the player_points output. The item with the highest value shall appear with a probability of $\approx 1/16$, while the item with the second highest value shall appear with a probability of $\approx 1/8$.

After a certain amount of time the game should get faster to make it more difficult (i.e., the upwards movement of the bricks as well as the downwards movement of the player shall be increased).

Pressing the Start button when the game is in the RUNNING state switches the game into the PAUSED state. In this state the movement of the player and the bricks is stopped and arrow key inputs are ignored. Pressing Start again resumes the game. Note that when the game is paused, the timer that controls when to increase the speed of the game must also be paused.

When the player collides with the top or bottom edge of the screen the game shall transition into the GAME_OVER state. In this state the visual output shall be frozen, to show the last frame with the collision. Pressing the Start button on the controller shall put the game back into the IDLE state.

Use constants for all the values that determine the game play (e.g., row distance, movement speed, item points, speed increase interval, etc.).

It is a good idea to run the reference solution of Exercise 2 now, such that you fully understand the game mechanics before starting with your implementation.

Collision Detection: To assist you with the implementation of the collision detection between the player and the bricks/items in the game, we provide you with the object_collider. See the IP Cores Manual for details on how to interface with it. Besides the manual, we also provide you with a testbench for this module to better visualize its operation (run `make sim_gui` in the `object_collider` directory to start the simulation)

To demonstrate its usage in a real design we added another architecture for the ball_game module to the template. This architecture is called arch_ex2_demo and is located in the file `ball_game_ex2_demo.vhd`. You can use this architecture as a basis for your implementation of the game. However, keep in mind that your implementation must be placed in the file `ball_game_ex2.vhd`. Note that although the player is represented by a ball, for the collision detection it is treated as a rectangle by the object_collider. Further note that collision with the left and right screen borders is already handled by the object_collider.

During the game you will need to keep track of a large (and varying) number of game objects (i.e., bricks and items). One way to store them in your design is via a FIFO (from the ram_pkg package). Whenever you have to go through the game objects, because either the object_collider requests it or you have to update their positions (upwards movement of the bricks) you can simply read this FIFO. After the game object has been processed you can then either write it back to the FIFO or discard it (e.g., if an item has been collected by the player or a brick moved out of the screen). You can use the functions go_to_slv and slv_to_go provided by the object_collider_pkg package to convert between the game_object_t and std_logic_vector types. To keep track of the number of game objects you can either use a counter or simply add a special item to the FIFO that marks the end of the list (e.g., a game object with zero width and height). If you use a sufficiently large FIFO (e.g., containing 128 game objects) it is fine if you don't check its full flag (i.e., you don't have to worry about overflows).

Brick Generation and Placement: As already mentioned above, brick rows shall be generated randomly. Listing 1 shows a Python version of the algorithm you shall implement in your VHDL design. The code of this listing can be found in the file `ball_game/gen_bricks.py`.

```python
#!/bin/env python3

import random

BLOCK_WIDTH = 16
DISPLAY_WIDTH = 400

cur_x = random.randrange(0,3)*BLOCK_WIDTH
while (True):
  width = random.randrange(1,10)*BLOCK_WIDTH
  if(cur_x + width >= DISPLAY_WIDTH):
    width = DISPLAY_WIDTH - cur_x

  print("Brick: x=" + str(cur_x) + ", width=" + str(width))

  cur_x += width + random.randrange(2,5)*BLOCK_WIDTH
  if(cur_x >= DISPLAY_WIDTH):
    break
```

Listing 1: Brick generation algorithm

Note that, the arch_ex2_demo architecture also contains example code that shows how to generate (pseudo) random numbers that lie within a certain range. The height of the bricks shall be fixed at ≈12 pixels.

Audio Output: The game shall play sounds at the following occasions (when it is in the RUNNING state):

- The player collects an item
  Play a single tone for ≈ 0.2 s
- The game is over
  Play two successive single tones (≈ 0.25 s each) with increasing frequency

Architecture: You may use the object_collider module as is, but you don't have to. Feel free to modify it, change its interface, or just pick specific parts of it.

You may also create new entities that implement parts of the described game behavior and instantiate them in the ball_game module (similar to what is already done with the object_collider module in the arch_ex2_demo architecture). If you do so, put all additional entities in the `ball_game/src/` directory. It may, for example, be beneficial/useful to create a separate entity for the game object FIFO described above.

Testing: You might have already noticed that the `remote.py` script has a quite long latency when generating input for the NES controller in the interactive mode. To fix this problem we provide you with a simpler tool called `nes_cntrl.py`, that is optimized for low latency input and that does not interact with the other I/O devices (switches, LEDs, etc.). To use the tool, you first need to execute the following command to install the required packages (in the TILab):

```
pip3 install --user pynput dataclasses
```

The tool also lets you directly control the display_switch, to make it easy to change the output graphics controller of your design (run `nes_cntrl.py --help` for details). If you don't use `rpa_shell.py` to connect to the TILab computers, please make sure that you activate X-forwarding (`-X` command line argument of the `ssh` command).

Task 3: Bonus: SignalTap Measurement [12 Points]

Use a SignalTap II Logic Analyzer to analyze the behavior of your design during run-time. For this purpose trace the followings port signals of the ball_game module:

- gfx_instr
- gfx_instr_wr
- gfx_instr_full
- gfx_data
- gfx_data_wr
- gfx_data_full

Bring the game into the RUNNING state and trigger on the first instruction that is issued to the graphics controller for some frame (probably a `CLEAR_SCREEN` instruction). Include a screenshot of the trigger condition in your lab protocol. Furthermore, add a screenshot to your lab protocol showing at least the first 4 instructions issued to the graphics controller (you can use either the lcd_graphics_controller or the vbs_graphics_controller, i.e., it does not matter how the display_switch is configured). Make sure that the value of the vector signals (i.e., gfx_instr and gfx_data) are visible in the figure (split the screenshot into multiple images if necessary). Decode the first four instructions using the table provided in the template.

3.4 Submission

To create an archive for submission in TUWEL execute the `submission_exercise2` makefile target of the provided template.

```
cd path/to/your/project
make submission_exercise2
```

The makefile creates a file named `submission.tar.gz`.

A submission script checks whether all required files are present and in the right location. If the script reports an error, no archive will be created. Carefully check the warnings that are generated. The created archive should have the following structure.

```
submission.tar.gz
 ├─ report.pdf ........................................ Your lab report
 └─ vhdl ................................ The source code of all IP cores
     ├─ top
     ├─ vbs_graphics_controller
     ├─ ball_game
     └─ [...  all other IP cores]
```

All submissions which can not be compiled will be graded with zero points! Don't manually create the archive. If you have problems running the makefile target consult a tutor.

Revision History

Revision	Date	Author(s)	Description
2.0	05.04.2020	FH	Added Exercise 2.
1.4	20.03.2020	FH	Fixed I/O signal names of the display_switch module in Figure 2.4.
1.3	17.03.2020	FH	Added player_points input to Table 2.2.
1.2	16.03.2020	FH	Fixed clock name in Task 4 (nclk → clk).
1.1	10.03.2020	FH	Fixed description of signals hex{0-3} in Table 2.2, added missing outputs (sclk, scen) to lcd_graphics_controller in Figure 2.5
1.0	08.03.2020	FH	Initial version

Author Abbreviations:

FH	Florian Huemer
JM	Jürgen Maier