Digital Design and Computer Architecture LU

Lab Exercises I and II

Florian Huemer, Florian Kriebel
fhuemer@ecs.tuwien.ac.at, florian.kriebel@tuwien.ac.at
Department of Computer Engineering
TU Wien

Vienna, April 12, 2023

1 Introduction

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

- Exercise I: ~~06.04.2023~~ 09.04.2023, 23:55
- Exercise II: 04.05.2023, 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 report 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 "Space Invaders" clone[1], which uses a Dualshock 2 controller[2] for user input and generates an RGB analog component video signal[3] on the VGA connector of the board.

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 subtle and, hence, very hard-to-find bugs.

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, we provide you with a (Virtual Box) VM image. The VM runs CentOS 7 (the same operating system as used in the lab) with the free versions of Quartus (Quartus Prime Lite Edition) and Questa/Modelsim (ModelSim-Intel) already preinstalled. You can download the VM using `scp` from `ssh.tilab.tuwien.ac.at:/opt/eda/vm/ECS-EDA-Tools_vm_09032023.txz`. Extract the archive using e.g., `tar -xf ECS-EDA-Tools_vm_09032023.txz`. The root/user password of the VM is `ecseda`, change it using the `passwd` command.

You are, of course, also free to download and install the tools yourself[4]. Unfortunately, we cannot provide you with any help or support for that (in contrast to the VM).

The simulation performance of ModelSim-Intel is lower than the full version of Questa/Modelsim provided in the lab (especially for large designs). However, for the purpose of this course, this should not be a big problem.

[1] https://en.wikipedia.org/wiki/Space_Invaders
[2] https://en.wikipedia.org/wiki/DualShock
[3] https://en.wikipedia.org/wiki/Component_video
[4] https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/download.html

1.3 Submission

Do not change the LaTeX report 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 report must be inside one of the “qa” environments, everything outside of these environments will not be considered for grading.

When including 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. However, all other warnings 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 (i.e., 13024 and 21074) 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.`
276027	`Inferred dual-clock RAM node [...] from synchronous design logic. The read-during-write behavior of a dual-clock RAM is undefined and may not match the behavior of the original design.`
15064	`PLL [...]|altpll:altpll_component|pll_altpll:auto_generated|[...]" output port clk[...] feeds output pin [...] 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`
292013	`Feature LogicLock is only available with a valid subscription license. You can purchase a software subscription to gain full access to this feature.`
13024	`Output pins are stuck at VCC or GND`
21074	`Design contains [...] input pin(s) that do not drive logic`

Table 1.1: Allowed warnings

2 Exercise I (Deadline: 09.04.2023)

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 software tools, the lab and the remote working environment if you plan to work remotely. 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 in VHDL is translated into a gate-level netlist. This netlist can then be mapped to the FPGA's logic cells. Finally, a bitstream (SOF) file is created, which is used to configure the FPGA.

Note that we provide you with a reference implementation in the form of a bitstream file (located in the TILab under `/opt/ddca/ref_ex1.sof`). 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 staff 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, simple questions can be answered by the staff immediately and you don't always have to wait for the next tutor slot.

2.2 Required and Recommended Reading

Essentials (read before you start!)

- Design Flow Tutorial
- VHDL Coding and Design Guidelines
- Hardware Modeling VHDL introduction slides (see TUWEL)

Consult as needed

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

2.3 Task Descriptions

Task 1: Introduction and Preparations [10 Points]

Your task is to create a Quartus project for the VHDL design that will be used throughout Exercises I and II, add some missing parts and program it onto the FPGA board. A structural overview of the top-level module (`top/src/top.vhd`) is shown in Figures 2.3-2.6. Note that inputs are always drawn on the left side of a module, while outputs are drawn on the right side.

Project Creation: Create a new Quartus project in the `top/quartus/` directory. The name of this project shall be <u>top</u>. Quartus will create two files named `top.qpf` and `top.qsf`. Set the VHDL version of the project to VHDL-2008 (otherwise it will not compile). 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 (using `make quartus`). This can be useful if you work on the lab computers over an SSH connection or if you prefer to work without the GUI.

Add the top-level VHDL source file (top/src/top.vhd) as well as the source files of the required IP cores in the vhdl/ directory to the project. Note that some of the cores are only provided as precompiled modules. This includes the precompiled_dualshock_ctrl, the vga_gfx_ctrl, the audio_ctrl and the dbg_port. These modules always come with one or more *.vhd files as well as a *.qxp and (sometimes) a *.vho file in their src/ directories. For your Quartus project add all *.vhd files as well as the *.qxp file (the *.vho is only required for simulations). For example, for the audio_ctrl the files audio_ctrl_top.qxp, audio_ctrl_2s.vhd and audio_ctrl_pkg.vhd are needed.

Note that the game module depends on the gfx_cmd_pkg package. Thus, be sure to also add gfx_cmd/src/gfx_cmd_pkg.vhd to the project. Furthermore, don't forget to add the math_pkg package (math/src/math_pkg.vhd) as many IP cores depend on it. You can also add the source files from the mem/src directory as we will need them in Exercise II.

Now, before you can synthesize the project, you also have to:

- create a PLL and instantiate it in your top-level source design
- configure the pin assignment of the FPGA

PLL Generation: The main system clock applied to the clk input of the top-level module has a frequency of 50 MHz. However, our system requires two additional clock signals (audio_clk and display_clk), which have to be generated out of the 50 MHz clock using a PLL. The PLL shown in Figure 2.3 is not supplied by the code base and there is also no instance for it present in the top-level design (top/src/top.vhd). You need to generate it using the corresponding wizard in Quartus (see the Design Flow Tutorial for further information) and then add it to the system. The first clock output of the PLL (which will be named c0 by the PLL generation utility) is required by the audio controller and must be configured to 12 MHz. The second output clock (c1) is needed for the vga_gfx_ctrl to generate the output video signal and must be set to 25 MHz. Place the VHDL files generated by the wizard for the PLL in the top/src/ folder.

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

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

PLL_INST_NAME must be replaced by the name of your PLL instance in the top-level design architecture. This command prevents the timing analyzer to report problems for signals crossing between the 50 MHz system clock domain and the 12 MHz audio clock domain. Don't add a similar rule for the 25 MHz, though!

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).

System Explanation and Download: The top-level module connects all the modules that make up our Space Invaders game system. The "heart" of the system is the game module which implements the actual game logic, processes controller input, sends commands to the vga_gfx_ctrl and plays sounds using the audio_ctrl. However, currently it only supports quite basic functionality.

Its main purpose now is to demonstrate some of the functions of the other modules in the system and how to interact with them. Implementing the actual game logic will be done in Exercise II.

The audio_ctrl implements a simple synthetic sound generator that interfaces with the board's audio DAC (digital to analog converter) WM8731. The vga_gfx_ctrl processes graphics commands from the game module or the dbg_port and generates the video signal for the VGA port of the board. For that purpose it uses the board's SRAM to store the required image data. Figure 2.1 highlights the external components on our FPGA board these modules interface with. More information about both of these components can be found in the IP Cores Manual.

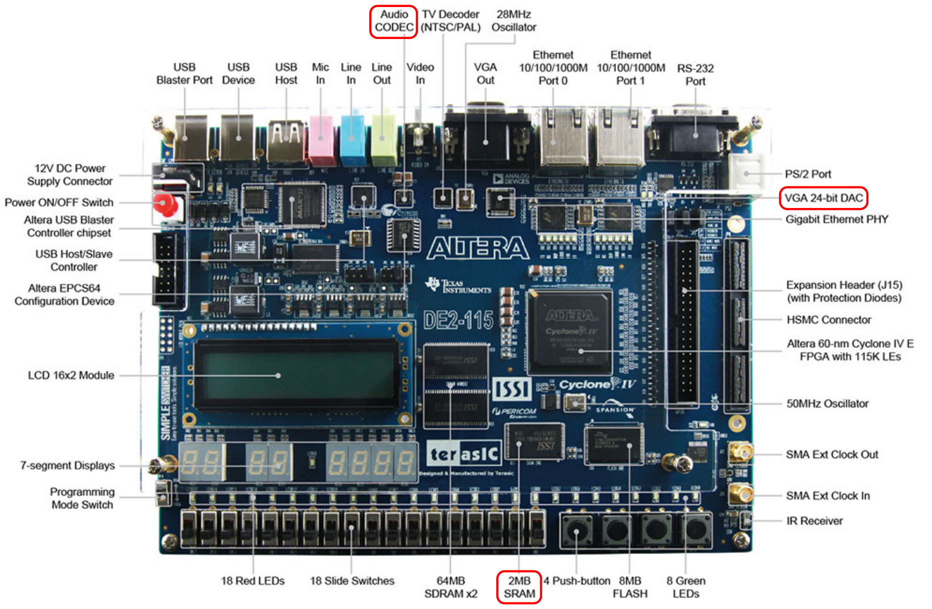


Figure 2.1: DE2-115 FPGA Development Board

The precompiled_dualshock_ctrl implements the interface to the DualShock controller, connected to the GPIO port of the FPGA board. In Exercise II, you will implement this module by yourself.

After creating the project, adding the PLL and adjusting the pin assignment, you can synthesize/compile the project and download it to the FPGA board. This can be done using the Quartus GUI or the provided Makefile in the `top/quartus` folder using the `download` target.

If the download succeeds, the monitor attached to the VGA connector of the board should show the player at the bottom of the screen and a monster moving from left to right at the top of the screen. The player can be moved using the directional buttons of the controller and fires a shot when the X button is pressed.

The seven-segment displays should display the current x and y displacement of the left and right analog sticks. The green and red LEDs show the state of the controller buttons (see Figure 2.6).

Debug Port: The dbg_port is a central system component that enables you to

- conveniently test and debug your system, even when you have physical access to the board and
- work completely remotely, i.e., without coming to the TILab physically.

From Figure 2.6 it can be seen that the keys and switches inputs of the top-level design are directly connected to the dbg_port. All other parts of the system only use the internal signals int_keys and int_switches produced by the dbg_port. After start-up of the design (or physical reset, by pressing KEY0) the dbg_port will simply relay the values of keys and switches to int_keys and int_switches completely unchanged. However, the dbg_port can also be switched into a mode where instead of the physical keys and switches signals, it outputs values that can be set using a simple software tool provided on the lab computers called `remote.py`.

When you are logged in to a computer with a board connected to it (i.e., either physically in the lab or using a Remote Lab computer) you can, hence, use this command to interact with your design. To use it, please first execute the following commands when logged in at a **TILab computer** to install the required Python packages. If you are working remotely, e.g., using the VM, you first have to use SSH to connect to a TILab computer (use `rpa_shell.py` for this purpose).

```
pip3 install --user --upgrade pip
pip3 install --user pyyaml termcolor dataclasses docopt pyserial aiohttp
```

Now you can execute the following commands:

```
remote.py -s # read the current state of the (hardware) switches
remote.py -s 0x2aa00 # set the software switches to an alternating pattern
remote.py -s # read back the value just set
```

After running these commands, the upper red LEDs (17 downto 8) should light up in an alternating pattern and changing the physical switches should have no effect. The `-s` command line argument lets you access the switches, the keys can be accessed analogously using the `-k` argument. Setting the value of the switches or the keys using the `remote.py` tools automatically disconnects the physical inputs from the switches and keys outputs of the dbg_port and connects the software configurable values. Internally this is achieved using a multiplexer which is controlled by the software input control (SWIC) flag, which can be directly accessed using the `--swic` command line argument:

```
remote.py --swic 0 # disable the software keys/switches
remote.py --swic 1 # manually enable the software keys/switches
remote.py --swic # read the current value of the software input control flag
```

Figure 2.6 also shows that the dbg_port is connected to the green and red LEDs (ledg, ledr) and the seven-segment displays (hex0-7). Those values can be read using the `-g`, `-r` and `-x` command line arguments.

The dbg_port also contains an emulator for the physical DualShock controller, such that the design can also be operated in the Remote Lab. This is the reason for the emulated_ds_* signals of the top-level entity. The boards in the Remote Lab have these signals connected with the ds_* ports using external wires. The boards at the normal workplaces in the lab have the physical DualShock controller connected at ds_*. To control the state of the emulated controller `remote.py` offers the `-n` command line argument, which lets you access a 48-bit value, containing the state of all buttons and analog sticks. Individual buttons and analog stick values can be accessed using the -b command line option[5]:

```
remote.py -n 0 # reset all buttons and analog values to zero
remote.py -n -b TRIANGLE 1 # press the triangle button
remote.py -n -b START 1 # press the start button
remote.py -n -b Left 1 # press the left arrow button
remote.py -n -b RY 0xdd # set the y axis value of the right stick to 0xdd
remote.py -n -b LX 0xca # set the x axis value of the left stick to 0xca
```

[5]Note that changing the state of the emulated controller does not have any effect if a physical controller is connected to the board.

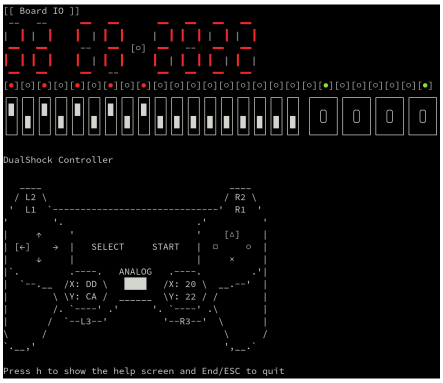


Figure 2.2: Screenshot of the interactive mode of the `remote.py` tool

However, a more convenient way to interact with the emulated controller (as well as the other I/O of the board) is the interactive mode of the `remote.py` tool. Using the `-i` command line argument brings up the user interface shown in Figure 2.2.

Another important feature, which will become important for Task 3, is the ability to send graphics commands to the vga_gfx_ctrl. Figure 2.4 shows that the gfx_cmd and gfx_cmd_wr outputs of the game module are not directly connected to the vga_gfx_ctrl, but that there are multiplexers in between. These multiplexers are controlled by the graphics command source control signal (gcsc) coming from the dbg_port. If this signal is asserted, the multiplexers relay the signals dbg_gfx_cmd and dbg_gfx_cmd_wr of the dbg_port to the vga_gfx_ctrl. To issue commands using the `remote.py` tool, use the `--gfx` command line option, which will automatically assert the gcsc signal. Executing the following script will color the screen red and then put a blue circle in the middle.

```
remote.py --gfx 0x9000 0 0 320 240 # define base address and dimensions bitmap 0
remote.py --gfx 0x9800 # activate bitmap 0 for drawing
remote.py --gfx 0xa000 # output bitmap 0
remote.py --gfx 0x8003 # set primary color to blue
remote.py --gfx 0x84e0 # set secondary color to red
remote.py --gfx 0x2400 # clear bitmap with secondary color
remote.py --gfx 0x0800 160 120 # move graphics pointer to the center of the screen
remote.py --gfx 0x4000 16 # draw a circle with radius 16 using the primary color
```

If you use the `--gfx` command line option without specifying a command, the data returned by a previous read command (`GET_PIXEL` or `VRAM_READ`) is read. For more information about the command format, refer to the IP Cores Manual. Note that all those commands could also have been issued using just a single call to `remote.py`. To deassert the gcsc signal run `remote.py --gcsc 0`.

Please refer to `remote.py -h` for a full documentation of all command line options and features.

Reset: Note that the dbg_port module's reset is directly connected to keys(0), hence it can only be reset by physically pressing the reset button on the board (or by reprogramming the FPGA altogether). All other components are reset by the signals res_n, audio_res_n and/or display_res_n, which are generated using the (synchronized) output of the AND gate in Figure 2.3. Because the reset signals in our system are low active, the AND gate ensures that the reset can be triggered by either of the AND gate's inputs. One input of this AND gate is connected to int_keys(0) (which, as explained above can be connected to the physical button or the "virtual" button of the debug interface) while the other one is connected to the sw_reset output of the dbg_port. The software reset can be issued using `remote.py --reset`.

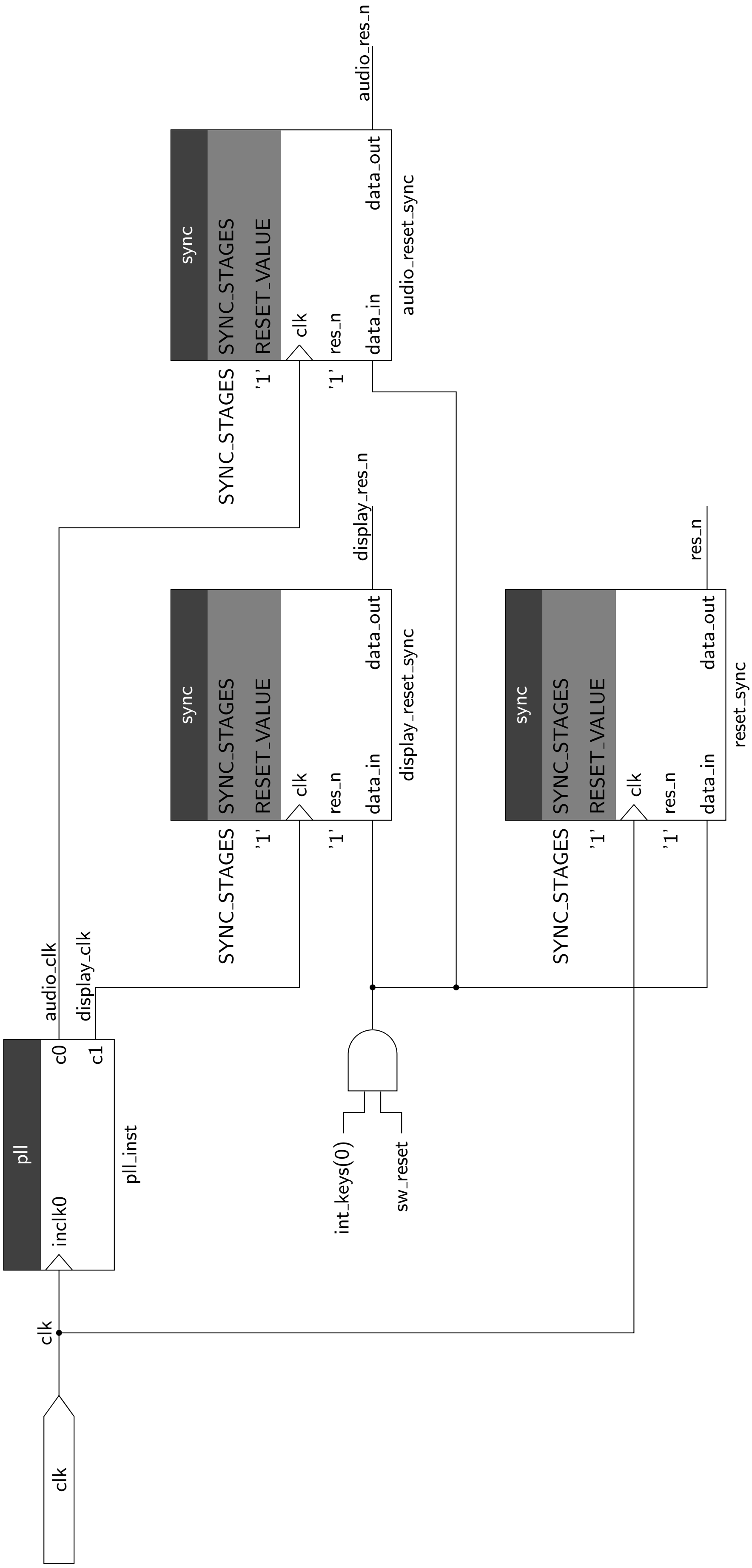


Figure 2.3: Structural system specification (clock and reset)

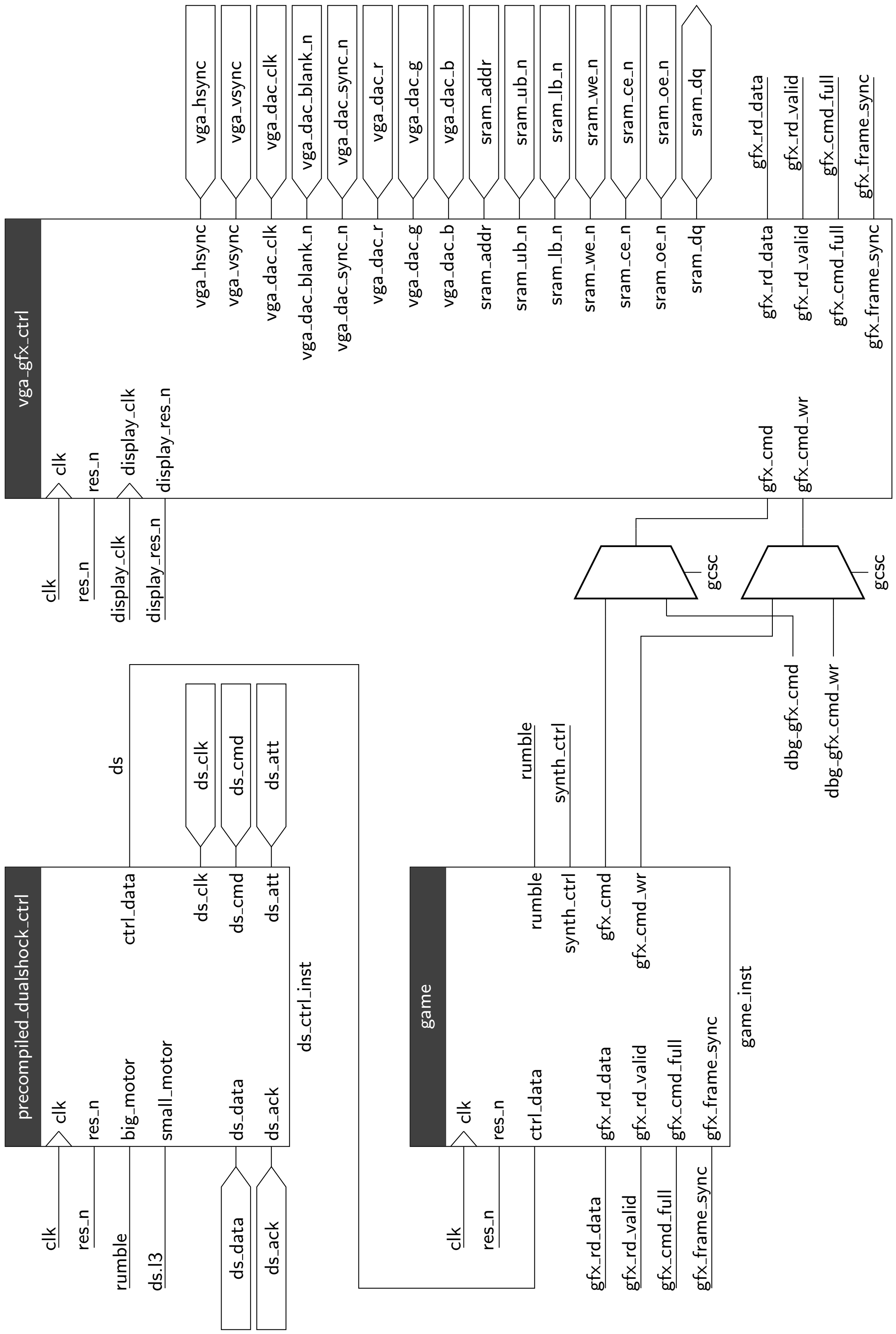


Figure 2.4: Structural system description (core system components)

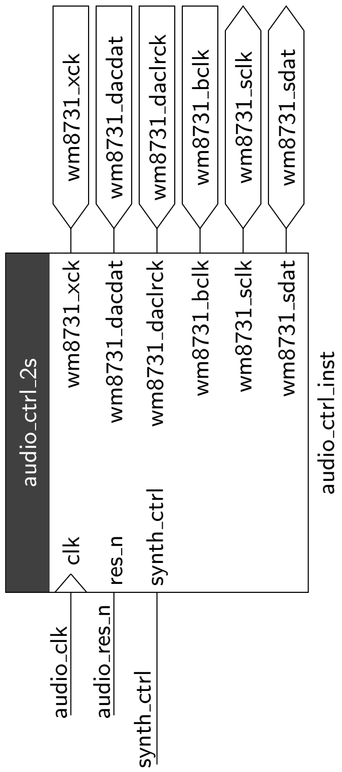


Figure 2.5: Structural system description (audio controller)

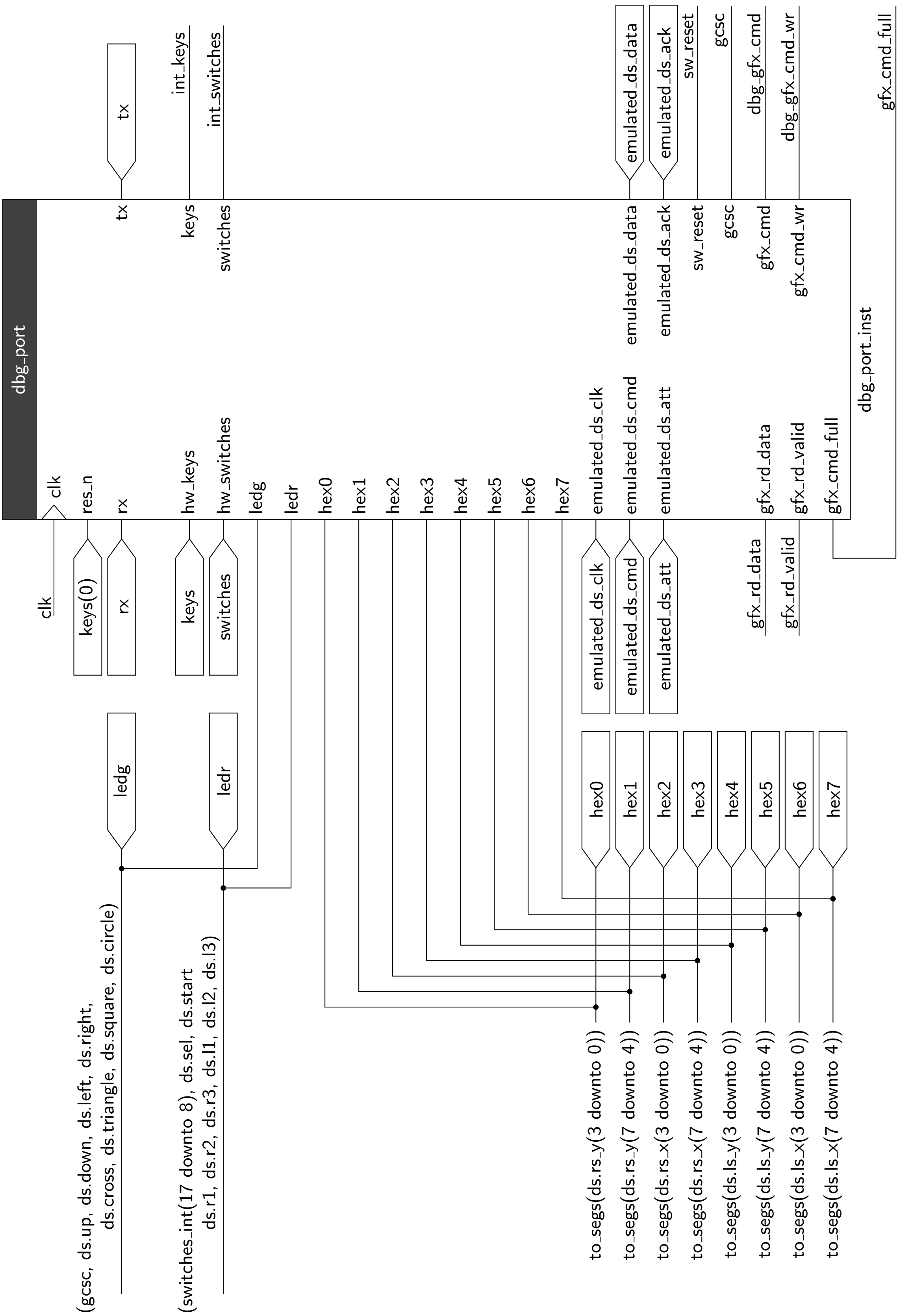


Figure 2.6: Structural system description (debug port)

Task 2: Seven-Segment Display Controller [45 Points]

In this task you will implement a controller for the seven-segment displays of our system. The entity declaration of this module, which we will refer to as the ssd_ctrl is shown below.

```vhdl
entity ssd_ctrl is
  port (
    clk : in std_logic;
    res_n : in std_logic;

    -- controller input
    ctrl_data : in dualshock_t;

    -- button/switch inputs
    sw_enable : in std_logic;
    sw_stick_selector : in std_logic;
    sw_axis_selector : in std_logic;
    btn_change_sign_mode_n : in std_logic;

    -- seven-segment display outputs
    hex0 : out std_logic_vector(6 downto 0);
    hex1 : out std_logic_vector(6 downto 0);
    hex2 : out std_logic_vector(6 downto 0);
    hex3 : out std_logic_vector(6 downto 0);
    hex4 : out std_logic_vector(6 downto 0);
    hex5 : out std_logic_vector(6 downto 0);
    hex6 : out std_logic_vector(6 downto 0);
    hex7 : out std_logic_vector(6 downto 0)
  );
end entity;
```

The purpose of the ssd_ctrl is to display the current state of the analog stick values of the controller in a few different ways. As can be seen in Figure 2.6 the top-level design already outputs these signals as hex values. Note that for both sticks (i.e., the left and the right one) there are two 8 bit numbers representing the current x and y displacement. The center location of a stick (i.e., its idle position) corresponds to value 0x80 for both the x and y axis[6].

Specification: As long as the input sw_enable is low, the ssd_ctrl should produce the same output as is currently done by top-level design (see Table 2.1).

hex7, hex6	ctrl_data.ls_x as a hexadecimal number
hex5, hex4	ctrl_data.ls_y as a hexadecimal number
hex3, hex2	ctrl_data.rs_x as a hexadecimal number
hex1, hex0	ctrl_data.rs_y as a hexadecimal number

Table 2.1: hex* outputs when sw_enable is low

If sw_enable is high the ssd_ctrl should switch to a mode where it only shows the x/y displacement for <u>one</u> of the sticks. Which of the sticks is chosen is determined by the input sw_stick_selector. If sw_stick_selector is low (high) the right (left) stick is selected. In this mode hex7-6 (hex5-4) show the x (y) displacement of the selected stick as a hexadecimal number. The displays hex3-0 shall output the value of one of the axes of the selected stick as a <u>decimal number</u>. Which axis is selected is determined by the sw_axis_selector input, where a low (high) signal level selects the y (x) axis. After reset the core shall interpret the displacement values as unsigned 8 bit integers

[6]Depending on the actual controller this value might be slightly off.

and, hence, display values between 0 and 255. Pressing the (low-active) button attached to the input btn_change_sign_mode_n switches the ssd_ctrl into signed mode, where it shall output values between -128 and 127. A button press can be detected by observing a signal change from high to low on btn_change_sign_mode_n between two clock cycles. Pressing the button again switches the core back to unsigned mode.

Figure 2.7-2.10 show some example outputs for the ssd_ctrl for the controller input values of ctrl_data.ls_x=0xdd, ctrl_data.ls_y=0xca, ctrl_data.rs_x=0x20 and ctrl_data.rs_y=0x23.

Figure 2.7: sw_enable = '0'

Figure 2.8: sw_enable = '1', sw_stick_selector='0' (right), sw_axis_selector='0' (y)

Figure 2.9: sw_enable = '1', sw_stick_selector='1' (left), sw_axis_selector='1' (x), unsigned mode

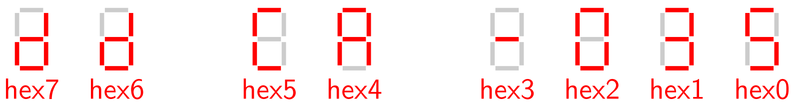


Figure 2.10: sw_enable = '1', sw_stick_selector='1' (left), sw_axis_selector='1' (x), signed mode

FSM Implementation: Your task is to design and implement a finite state machine that implements the behavior described above. To display decimal numbers you have to convert the respective values to BCD[7] (binary coded decimal) values. This conversion **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 100 until the value is smaller or equal to 99. By counting the number of times 100 could be subtracted, the hundreds digit is obtained. Now repeat the process by subtracting 10 to obtain the tens digit. The rest corresponds to the ones digit. Since we are dealing with 8-bit numbers the highest (absolute) value that can appear is 255. Hence, 100 will have to be subtracted at most twice. To deal with negative numbers[8] for the signed mode initially check, whether the value that needs to be converted is smaller than 0. If this is the case store the sign somewhere, convert the value to a positive number and continue normally with the conversion process.

To simplify things, the (minus) sign is always displayed using hex3. If a positive number is displayed this display should be empty. Furthermore, when generating the symbols for hex{0-2}, draw the leading zeros (see Figure 2.8).

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

[8] Recall that negative numbers are represented using the two's complement, see `https://en.wikipedia.org/wiki/Two%27s_complement` for details.

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 unnecessary signals!

Testbench and Simulation: Implement a testbench for your design (place it in `ssd_ctrl/tb/ssd_ctrl_tb.vhd`). The testbench shall run at least 4 test cases (i.e., different sets of input values) on your core and check the outputs using assertions.

To automate the compilation and simulation process use the makefile example provided in the `mem/` directory, to create your own makefile-based simulation flow. The makefile for the ssd_ctrl shall be placed in `ssd_ctrl/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`, `sim_gui` and `clean`. The `compile` target should compile all required source files using the Questa/Modelsim compiler (vcom). The simulation target `sim_gui` should start the graphical user interface of Questa/Modelsim and load an appropriate waveform viewer configuration script to add the relevant signals to the waveform viewer (`ssd_ctrl/scripts/wave.do`). Be sure to include all inputs and outputs of the ssd_ctrl, as well as the internal state variables (this also includes counters).

The `sim` target shall not start a graphical simulation but just run the simulation (i.e., execute the testbench). If everything works correctly, the testbench shall run through without any errors being reported. In case the unit under test (UUT), i.e., the ssd_ctrl, does not work as expected, the problem shall be reported through the text output of the simulator (in the terminal). Make sure that in both cases the simulator terminates.

The `clean` target should delete <u>all</u> (temporary) files generated during the compilation and simulation process.

System Integration: Add an instance of your ssd_ctrl to the top-level design. Connect its hex* outputs to the respective outputs of the top entity. Don't forget to remove the signal assignments to the hex* signals from the top entity's architecture first. Otherwise you have a driver conflict. Table 2.2 shows how the inputs of the core shall be connected.

ssd_ctrl input	top-level signal
clk	clk
res_n	res_n
ctrl_data	ds
sw_enable	switches_int(0)
sw_stick_selector	switches_int(1)
sw_axis_selector	switches_int(2)
btn_change_sign_mode_n	keys_int(3)

Table 2.2: ssd_ctrl connection in the top-level design

Be sure to add synchronizes for the inputs you feed into the ssd_ctrl. Finally, synthesize and run your design to test it in hardware.

Task 3: Game Testbench [45+(10) Points]

In this task you will implement a testbench for the game module. This work will come in handy in Exercise II when you implement the actual game logic.

As can be seen in Figure 2.4 and 2.5 the game module sits at the center of our system. It takes controller input and outputs commands for the graphics controller as well as controls the audio_ctrl.

In the finished game a lot of data is passed around between the game module and the vga_gfx_ctrl. Just for the initialization alone hundreds of 16 bit words are transferred. Finding bugs by just looking at this stream of data in a waveform viewer is very tedious and many times not really effective or even necessary. Hence, when running simulations on the game module it would be nice to have a graphical representation of the image that is produced by the vga_gfx_ctrl based on the commands it gets fed. For this purpose first implement the gfx_cmd_interpreter, which you can then integrate in your testbench for the game module.

Graphics Command Interpreter: The entity declaration gfx_cmd_interpreter is shown below:

```vhdl
entity gfx_cmd_interpreter is
  generic (
    OUTPUT_DIR : string := "./"
  );
  port (
    clk   : in std_logic;
    gfx_cmd       : in std_logic_vector(GFX_CMD_WIDTH-1 downto 0);
    gfx_cmd_wr    : in std_logic;
    gfx_frame_sync  : out std_logic;
    gfx_rd_data     : out std_logic_vector(15 downto 0);
    gfx_rd_valid    : out std_logic
  );
end entity;
```

Before you start reading the assignment text to this task, it makes sense to first consult the IP Cores Manual to get an overview of the vga_gfx_ctrl.

A template is already provided in the file `gfx_cmd/tb/gfx_cmd_interpreter.vhd`. As can be derived from its name, this module should implement a simulation model that offers the same (internal) interface as the vga_gfx_ctrl. This means that this module is **not** meant to be synthesizeable. Hence, there is no need to implement it as a classical state machine, as you would for a "real" hardware module. It should just receive graphics commands and immediately execute them by performing the appropriate graphical operations on an internal data structure representing the VRAM.

This data structure is implemented by the protected type[9] vram_t defined in the package vram_pkg. Protected types in VHDL are comparable to classes in other programming languages. However, you don't really need to fully understand this code or the concept of protected types in order to use it. The template already contains a process that shows how to integrate it into your gfx_cmd_interpreter.

The protected type vram_t features functions to read and write bytes and words from and to VRAM. Furthermore, it contains a function to dump an image in VRAM to a PPM image file[10]. Note that since PPM is a text based format the resulting images can be quite large.

Your gfx_cmd_interpreter should support all commands supported by the vga_gfx_ctrl as documented in the IP Cores Manual. The only exception is the `DRAW_CIRCLE` command. You don't have to draw anything when a `DRAW_CIRCLE` command is encountered. However, you still need to advance the graphics pointer correctly.

Whenever the command interpreter reads a `DISPLAY_BMP` command it should dump the bitmap identified by the `bmpidx` field to a file. The limitation to bitmaps with a size of 320x240 pixels

[9] `https://fpgatutorial.com/vhdl-shared-variable-protected-type/`

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

does not apply to the gfx_cmd_interpreter. It can output images of arbitrary dimensions. The file names of the dumped images shall be [N].ppm, where [N] is a number increased with every frame dump, starting at 0 (i.e., the first dumped image should be named 0.ppm). The image files should be placed in the directory specified by the generic OUTPUT_DIRECTORY. If the fs flag is set, the command interpreter shall assert gfx_frame_sync for a single clock cycle.

If you are unsure how certain commands should behave, you can use the graphics command interface of the dbg_port to test them on the "real" vga_gfx_ctrl in hardware. The GUI-based Remote Lab interface (rpa_gui.py) also offers a very convienient way to compose graphics commands. Be sure to have a look at the gfx_cmd_pkg before you start your implementation as it contains important constants and functions. Use those constants when you access certain flags or fields in the commands. Don't hard-code constants in your code!

Before you integrate your gfx_cmd_interpreter into a testbench for the game module, create a simple testbench to see if all the graphics commands are executed correctly (the game module in its current form does not use all of the commands). The file gfx_cmd/tb/gfx_cmd_interpreter_tb.vhd already contains a template. To start the simulation, create a Makefile (gfx_cmd/Makefile) with the targets compile, sim and clean, which should behave as described in the previous task. Your testbench shall execute each command at least once (except DRAW_CIRCLE). The generated image(s) shall be placed in the gfx_cmd/ directory.

Hint:

- Try to make use of functions and procedures, as this will significantly simplify your code.
- Develop the testbench and the command interpreter in parallel, i.e., whenever you implement a specific command, add testcode to your testbench to check if it works correctly.
- Start out by implementing the VRAM_WRITE* and VRAM_READ commands. The effects of a write to VRAM can be checked by the read operation.
- When this works implement DEFINE_BMP, ACTIVATE_BMP and DISPLAY_BMP.
- Then you can implement the simple drawing commands (SET_PIXEL, CLEAR, etc.) and the commands used to access the graphics pointer.
- Finally take care of the bit blit commands.

Testbench for the game module: If you think that your gfx_cmd_interpreter works sufficiently well create a testbench for the game module and place it in the file game/tb/game_tb.vhd. Create an instance of the game module and the command interpreter and connect them correctly (the gfx_cmd_full input of the game module can simply be set to '0'). To start the simulation, create a Makefile (game/Makefile) with the targets compile, sim and clean, with their usual behaviors. Your testbench shall wait for 8 images to be generated and then terminate. The images shall be stored in the game/ directory. Apply appropriate input to ctrl_state in order to fire a shot and move the player. Check if the images you generate are consistent with the output on the monitor attached to the FPGA board.

Note: The first frame output by the game module is completely black.

Bonus Task (10 Points): Implement the `DRAW_CIRCLE` command for the `gfx_cmd_interpreter`. For that purpose you can implement a version of Bresenham's algorithm for circles[11]. If you implement the Bonus Task, add a circle to the image generated by the command interpreter testbench. Bonus points will only be awarded if the rest of the `gfx_cmd_interpreter` is fully implemented and works (mostly) correctly.

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/ddca_ss2023/dd
make submission_exercise1
```

The makefile creates a file named `submission.tar.gz` which contains all the required files. The submission script automatically checks if all the 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`
    - `gfx_cmd`
    - `game`
    - `[... all other 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.

[11] `https://de.wikipedia.org/wiki/Bresenham-Algorithmus` contains a C version of this algorithm that you can use as a basis for your implementation

3 Exercise II (Deadline: 04.05.2023)

3.1 Overview

In this exercise you have to implement (i) the complete game logic of the Space Invaders game and (ii) your own version of the dualshock_ctrl.

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 to read the *whole* assignment and to run the reference solution, to get an intuition on how the game is supposed to behave.

3.2 Required and Recommended Reading

Consult as needed

- IP Cores Manual
- SignalTap manuals (TUWEL)

3.3 Task Descriptions

Task 1: Space Invaders Game [50 Points]

For Exercise I we supplied you with an architecture for the game module (`game_ex1.vhd`). This design only implemented very basic functions, like initializing the vga_gfx_ctrl, moving the player/space invader and firing a shot.

In this task you will implement your own game architecture, which shall provide the actual game logic. The architecture must be called ex2 and must be placed in the file `game/src/game_ex2.vhd`.

Before you start your own implementation try to understand the architecture ex1 provided in the `game_ex1.vhd` file (note that some parts have been updated since Exercise I). We highly recommend using this file as a basis for your own implementation.

Figure 3.1 shows a screenshot of how the final game shall look like and what information it shall display. However, your solution does not need to be a pixel accurate copy of the reference solution, as long as all game mechanics are implemented correctly.

To avoid describing the mechanics of the game in every detail, we refer to the reference solution located in `/opt/ddca/ref_ex2.sof` in the TILab. The controls for the game are as follows:

Controller Button	Function
Left	Move the player to the left.
Right	Move the player to the right.
Cross	Fire a shot
Start	Pause/Resume the current game or start a new game when the previous game is over

General overview: The goal of the game is to score points by destroying the space invaders by firing shots at them. The game starts out with a field of 16×5 space invaders[12], which move from

[12]This number varies between different versions of the game. However, in this course we use the specified field size.

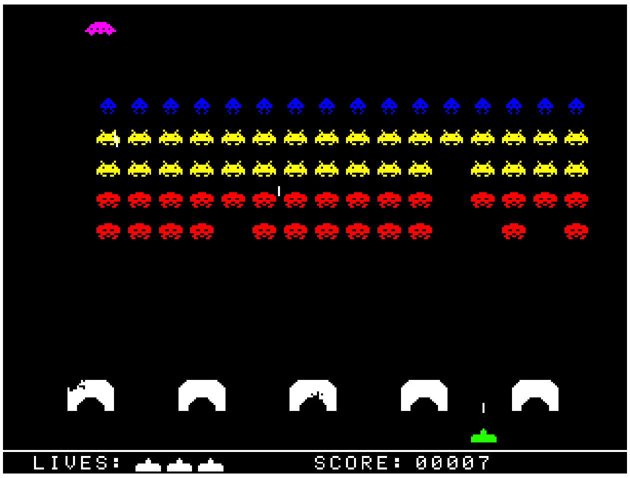


Figure 3.1: Space Invaders screenshot

side to side of the screen. Whenever either the left or the right screen border is reached, they move downwards and reverse direction. The movement speed of the space invaders depends on their number (less space invaders → faster movement). When the first space invader reaches a certain vertical position, the game is immediately over.

The player has to destroy all space invaders before that happens. If s/he manages to do so, a new round starts with a new field of space invaders. Hitting a space invader (with a shot) yields a certain number of points for the player (1 for a red one, 2 for a yellow one and 4 for a blue one).

Space invaders also fire shots at the player, which can (unless absorbed by a bunker) destroy the player. The player starts out with three lives – every hit costs a live. If no lives are left the game is over. Hence, note that there are two possibilities for how a game can end.

From time to time a single special flying-saucer-style enemy appears at the very top of the screen and moves from left to right. It does not fire shots. Hitting it yields 10 points for the player.

Initialization: After a reset the game module initializes the vga_gfx_ctrl by feeding it the commands defined in gfx_init_pkg. Those commands define a series of bitmaps (0-5) and fill some of them with data (see Table 3.1 and Figure 3.2).

Bitmaps 0 and 1 are the two frame buffers used for implementing double buffering. Those Bitmaps are only defined but not initialized (i.e., cleared). Bitmap 2 defines a font that can be used to print text via the `BB_CHAR` command. Bitmap 3 defines the main game assets (i.e., the space invaders, the player and patterns used to "damage" the bunkers). For the left-most three space invader types Bitmap 4 defines a second animation step. Finally, Bitmap 5 holds the five bunkers.

Note that in Figure 3.2 all bitmaps are drawn with an 8×8 pixels grid-overlay, which is, of course, not part of the actual image.

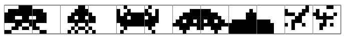
After initialization the primary color is set to black (i.e., 0x0) and the secondary color is set to white (i.e., 0xff).

Space Invader Field: We recommend using the sifield module of the game_util_pkg to manage the 16×5 field of space invaders (see IP Cores Manual). When the field moves horizontally across the screen, it moves rightwards/leftwards until the first space invader would hit the border of the screen before it moves downwards and reverses its direction. Hence, the distance traveled changes

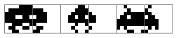
bmpidx	Dimensions (w×h)	Description	Figure
0	320×240	frame buffer 1	N/A
1	320×240	frame buffer 2	N/A
2	344×8	monospace font with 8×8 pixel characters	3.2a
3	96×8	game assets	3.2b
4	48×8	game assets for animations	3.2c
5	320×16	bunker bitmap	3.2d

Table 3.1: Bitmaps defined during initialization

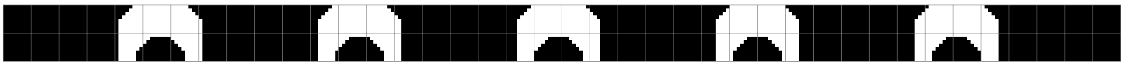
(a) 344×8 pixels, `bmpidx=2`



(b) 96×8 pixels, `bmpidx=3`



(c) 48×8 pixels, `bmpidx=4`



(d) 320×16 pixels, `bmpidx=5`

Figure 3.2: Predefined bitmaps

depending on the locations in the field actually occupied by space invaders, which, of course, changes throughout the course of a game. In the extreme case where there is only a single one remaining, this one space invader would have to move from the left border of the screen to the right border and back – no matter their original position in the field. The sifield module provides functionality to determine the right-, left- and bottom-most space invader in the field as well as the total number of space invaders remaining.

The field of space invaders shall move once every x frames, where x is given by the number of space invaders[13] remaining divided by four plus 10. Hence, initially x is 30 and decrements down to 10 throughout the game. Every time the field moves, the bitmap used to draw the space invaders should switch between Bitmaps 3 and 4.

For the horizontal motion the field shall move by one pixel at a time. When the field reaches a border, it shall move downwards by one row, i.e., 8 pixels.

Player Shots: We recommend using the shot_ctrl of the game_util_pkg to track shots (see IP Cores Manual).

The player can only fire one shot at a time. A new shot can only be fired when the previous one disappeared, i.e., hit something or moved out of the bounds of the screen. Hence, at any point in time there can be at most one active player shot.

A shot is fired by placing it right above the center of the player and letting it move upwards at a constant speed (i.e., by decrementing its y coordinate by a certain amount between two frames).

[13]The flying-saucer does not count towards this number

Don't move a shot by more that its length at a time as this causes issues with the collision detection!

Player shots collide with the bunkers and space invaders, however, not with other shots fired by space invaders.

Space Invader Shots: Every time the field of space invaders is moved, there is a chance of 1/2 that a shot is fired at the player. At any given time there can be at most four shots[14].

When a shot is fired by the field of space invaders you need to randomly select a shooter. Use the function `get_random_location` to get a random location in the field. However, you also need to check whether this random location actually (still) contains a space invader. If this is not the case simply select the next possible location until you find an occupied position.

Shots fired by space invaders don't collide with space invaders, only with the bunkers and the player.

Special Enemy: Every time the field of space invaders is moved, there is a chance of 1/32 that a flying-saucer-style special enemy appears. This enemy shall move from left to right at the top edge of the screen. However, if you want you can also vary the direction randomly and make some of them fly from right to left. It always flies above the field of space invaders at a constant height.

The special enemy does not need to be able to fire shots (however, you can add this feature if you want to). It also does not count to the total number of space invaders remaining, i.e., if all space invaders of the field are destroyed the round is over, regardless if there is an active special enemy or not.

If the special enemy is not hit by the player, it simply moves out of the visible area of the screen and disappears.

Bunkers: The bunkers protect the player by absorbing shots (both from the player and the space invaders). Whenever a bunker is hit, it is damaged. The bunkers are not "repaired" between rounds (only when a new game starts).

Bunker damage can be implemented by drawing a damage pattern (see the two right-most 8×8 sections in Figure 3.2b) over the bunker bitmap centered at the point of collision with the shot causing the damage. To get a nice, dynamic, non-repeating effect always select a random damage pattern and draw it with a random rotation. Use 2 bits of the prng_value of the template to get a random rotation and another bit to switch between the two patterns.

Drawing Text: Text can be drawn efficiently using `BB_CHAR` instructions with Bitmap 3 (see Figure 3.2a) as the source. However, be sure to not introduce separate states in your state machine for each letter you need to draw. It is much more efficient and less error-prone to store multiple instructions in an array and have some state iterating over this array to transmit the commands to the vga_gfx_ctrl.

To output decimal numbers you can use the decimal_printer (see IP Cores Manual). However, you can also create your own version of this core by adapting your ssd_ctrl form Exercise I.

Shots shall be stopped by the white border line at the bottom of the screen. Don't draw shots over the text.

[14]Please consider this value as a lower bound. Feel free to add more shots to make the game more difficult. You can also increase the number of shots with the number of rounds played. However, your game shall support at least four (active) shots at the same time.

Audio Output and Controller Rumble: The game shall play sounds at the following occasions:

- A space invader is destroyed
  Play a single tone for ≈ 0.25 s.
- A bunker is damaged
  Play a single tone for ≈ 0.25 s with a lower frequency than for the case a space invader is destroyed.
- The game is over or the player lost a live
  Play two successive single tones (≈ 0.25 s each) with increasing frequency

The controller shall vibrate each time the player gets hit or when the game is finally lost (Game Over). If you are motivated, you can, of course, add further effects, or try to implement the classic Space Invaders game music.

Pause and Game Over Screen: The game can be paused at any time by pressing the pressing the start button. Pressing it again resumes the game. While the game is paused it should simply display the word "PAUSED" on a black screen.

When the game is over the final score shall be displayed on a black screen alongside the word "GAME OVER". Pressing the start button, starts a new game.

Architecture: We recommend that you use the `ex1` architecture of the game module as a basis for your implementation. Moreover, you may use the sifield and the shot_ctrl modules as is, but you don't have to. Feel free to modify them, change their interface, or just pick specific parts of it. However, please don't change the original files in `game_util` folder, i.e., if you want to modify them make a copy of the particular module, rename it (e.g., my_sifield), and place it in the game folder.

You may, of course, also create new entities that implement parts of the described game behavior and instantiate them in the game module. If you do so, put all additional entities in the `game/src/` directory.

Suggested implementation sequence:

- Drawing of the space invader field (include sifield)
- Movement of the space invader field (left, right, downwards)
- Player shots (include shot_ctrl), destruction of space invaders
- Scoring and text display (include decimal_printer)
- Game Over detection
- Pausing
- Space invader shots
- Bunkers and damage
- Music effects and controller rumble

Suggested state machine structure (sequence of operations per frame)

- Clear the screen
- Move the player (add a player shot if necessary)
- Move the space invaders (add a new space invader shot and/or special enemy if necessary)
- Move special enemy
- Draw the player, space invaders, bunkers, text and the special enemy (if necessary)
- Move the shots (player and space invaders)
- Perform collision checks (apply bunker damage to the bunker bitmap, add points, destroy space invaders)
- Draw the shots

Hint:

- To schedule an event that happens with some specific chance of $1/2^n$, simply test whether prng_value(n-1 downto 0) is equal to the all-zero vector.
- To store the shots you can use the shot_vector_t provided by the game_util package. However, you can e.g., also use a FIFO.
- For testing, it can be very useful to include certain debug features to quickly bring the game into certain states. Since the controller has a lot of buttons that we don't need for our game, you can use those for this purpose. You don't have to remove your debug features for the final submission. Only make sure that the button used in the game (i.e., left, right, cross and start) work as specified.
- Sometimes it can be helpful to add report statements to your gfx_cmd_interpreter (or game module) to better trace the sequence of commands being executed.

Task 2: DualShock Controller [50 Points]

In this task you will implement the interface to the DualShock controller, i.e., the dualshock_ctrl module, that shall then replace the precompiled_dualshock_ctrl in the top-level architecture.

```vhdl
entity dualshock_ctrl is
  generic (
    CLK_FREQ : natural := 50_000_000;
    DS_CLK_FREQ : natural := 250_000;
    REFRESH_TIMEOUT : natural
  );
  port (
    clk   : in std_logic;
    res_n : in std_logic;

    -- external interface to the DualShock controller
    ds_clk  : out std_logic;
    ds_cmd  : out std_logic;
    ds_data : in  std_logic;
```

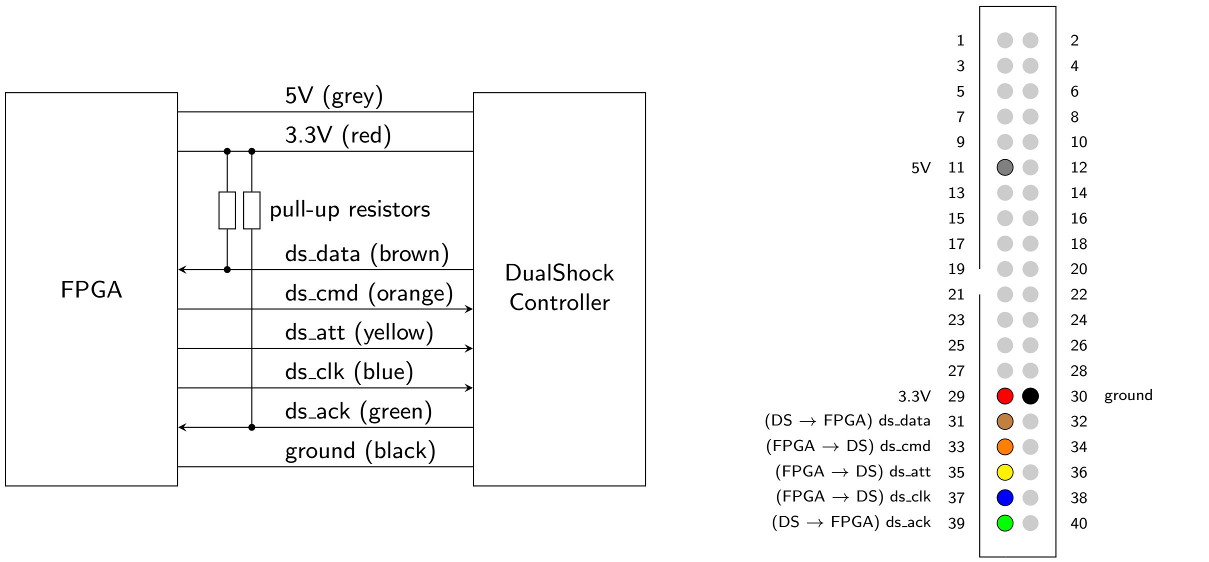
```vhdl
    ds_att   : out std_logic;
    ds_ack   : in  std_logic;

    -- internal interface
    ctrl_data : out dualshock_t;
    big_motor   : in std_logic_vector(7 downto 0);
    small_motor : in std_logic
  );
end entity;
```

Physical Connection: The DualShock Controller uses an 8-pin interface as shown in Figure 3.3a. It consists of two supply voltages, a ground connection and 5 signal wires. For the actual communication on the signal wires 3.3V are used. The second supply voltage is only used to drive the rumble motors inside the controller. The PlayStation uses a slightly higher voltage but the 5V provided by our FPGA board also suffice.

The 5 signal wires are referred to as clock (ds_clk), command (ds_cmd), attention (ds_att), data (ds_data) and acknowledge (ds_ack). The two output signals of the controller (i.e., ds_data and ds_ack) are open drain outputs and require external pullup resistors. You don't need to worry about those resistors as we already integrated them into the connector. For this application it is, unfortunately, not possible to use the internal pullups integrated into the I/O cells of the FPGA because their resistance is too large. The resistance should be around 1kΩ – DualShock controllers are very particular about this value.

Figure 3.3b shows how this 8-pin interface is mapped onto the GPIO connector of the FPGA board. In the lab everything should already be connected, there is no need to disconnect any wires. If you suspect that a controller might not work correctly you can test it with another board or with the PlayStation provided in the lab. If you suspect that the connection from the board to the controller has problems, please report this to a tutor and/or our TU Chat channel.



(a) 8-pin DualShock controller interface

(b) Pin assignment to the FPGA board's GPIO connector (40-pin expansion header)

Figure 3.3: Physical DualShock controller connection

Bit-Level Communication Protocol: The DualShock controller uses a full duplex serial synchronous communication interface – very similar to the Serial Peripheral Interface (SPI)[15] discussed in the lecture. In fact the only differences to SPI are the use of open drain outputs and the additional acknowledge signal (ds_ack).

- Synchronous: A clock signal (ds_clk) is used and transmitted. Data wires (ds_cmd and ds_data) must only be sampled at specific clock edges. Per default this clock shall have a frequency of 250kHz.
- Serial: Data is transferred one bit at a time. In this case the transmission happens LSB first.
- Full duplex: Data transfer happens in both directions (i.e., controller to host and host to controller) at the same time. There are dedicated data wires for both directions.

Communication is always initiated by the host (i.e., the FPGA or the PlayStation), which also has to provide the clock. Data is transferred in the form of packets, which consist of multiple bytes. Figure 3.4 shows an example data transmission for such a packet.

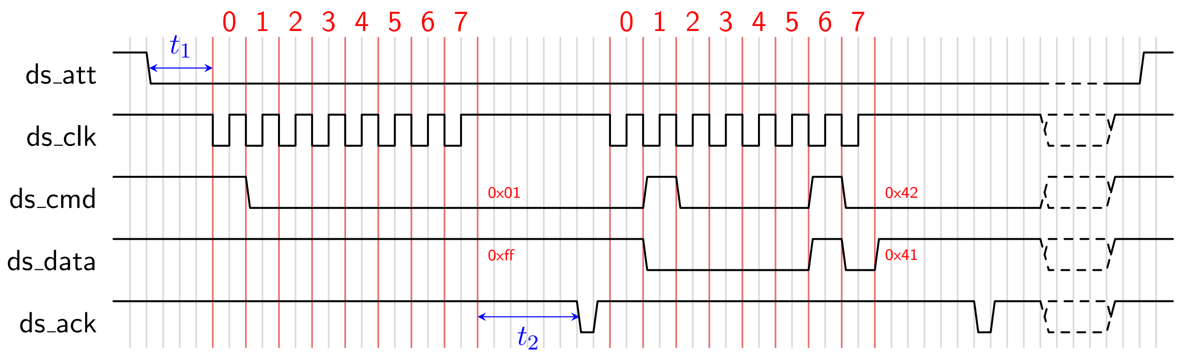


Figure 3.4: DualShock Timing

First, ds_att goes low, to indicate the start of a packet. This signal can be considered a low-active chip-select signal for the controller.

Then the data is transferred byte by byte. For that purpose the host generates 8 clock pulses, to transmit and receive one byte to/from the controller using the signals ds_cmd and ds_data, respectively. Note that the idle level of ds_clk is high. The controller and the host always apply the next data bit on falling clock edges and sample the data at rising ones. The time between the falling edge of ds_att and the first falling edge of ds_clk (i.e., t_1) shall be 8 ds_clk cycles (this is not drawn to scale in the figrue). After the transmission of a byte the controller pulls the ds_ack to low for half a clock cycle to indicate that data has been received. This happens approximately 12 us after the last data bit (t_2). The absence of the acknowledge pulse indicates a disconnected controller. Note, however, that the controller omits the acknowledge pulse for the last byte of a packet!

After the pulse on the ds_ack signal the next data byte can be transmitted immediately. If the last byte of a packet has been transmitted ds_att goes high.

Packet-level Communication Protocol: In this section we will only present the part of the protocol that is relevant for the course. If you are interested in further details, there are several online resources[16].

[15] https://en.wikipedia.org/wiki/Serial_Peripheral_Interface

[16] e.g., https://store.curiousinventor.com/guides/PS2

Packets always start with a 3-byte header, which is then followed by 2, 6, or 18 additional bytes[17] depending on the current operation mode of the controller. There are three operation modes of the controller referred to as digital, analog and configuration mode.

The header is used to indicate to the controller what data we want to read and which mode we want it to switch to.

- Byte 1:
  - `command`: always `0x01`
  - `data`: always `0xff`
- Byte 2:
  - `command`: This is the main command byte that controls what the packet actually does. Possible values are:
    - `0x42`: Main polling command (if the controller is in digital mode this value reads out the 16 buttons, in analog mode the joy stick values are returned as well)
    - `0x43`: Enter/exit configuration mode
    - `0x44`: Switch between analog and digital mode (only possible in configuration mode)
    - `0x4d`: Activate motor control (only possible in configuration mode)
  - `data`: This byte returns the current mode in the upper 4 bits (digital mode: `0x4`, analog mode: `0x7`, configuration mode: `0xf`) and the number of 16-bit words that follow the header in the lower 4 bits. Typical responses are:
    - `0x41`: The controller is in digital mode and 2 bytes follow the header (containing the information about the 16 digital buttons of the controller).
    - `0x73`: The controller is in analog mode and 6 bytes follow the header (containing the information about the 16 digital buttons as well as the 8-bit X/Y displacement values for the left and right stick).
    - `0xf3`: The controller is in configuration mode and 6 bytes follow the header (the actual information in these bytes can be ignored, as it is not relevant for our purposes).
- Byte 3:
  - `command`: always `0x00`
  - `data`: always `0x5a`

When the controller is first connected, it is in digital mode and the motors are disabled. Issuing the main polling command to the controller, thus, returns the state of the 16 digital buttons in the data bytes 4 and 5 (i.e., the values D_1 and D_2). The exact values transmitted and received are listed in Table 3.2. The command bytes 4 and 5 are zero.

byte	1	2	3	4	5
command	0x01	0x42	0x00	0x00	0x00
data	0xff	0x41	0x5a	D_1	D_2

Table 3.2: Main polling command (controller is in digital mode)

[17] 3+18 bytes packets are not used in this exercise

Table 3.3 lists how the individual bits of the bytes D_1 and D_2 map to the digital buttons of the controller. A low bit indicates a pressed a button, i.e., D_1=D_2=`0xff` corresponds to a controller state with no buttons pressed. The buttons R3 and L3 are activated when the analog sticks are pressed.

	0	1	2	3	4	5	6	7
D_1	Select	L3	R3	Start	Up	Right	Down	Left
D_2	L2	R2	L1	R1	Triangle	Circle	Cross	Square

Table 3.3: Mapping of the digital controllers buttons to the bits in D_1 and D_2

To activate the analog mode and the motor control, the controller first needs to be switched into configuration mode. Table 3.4 shows the packet that needs to be send to achieve this[18]. This command essentially behaves like the main polling command from before, as it also returns the buttons states. The entries marked blue show the differences to a "normal" polling command.

byte	1	2	3	4	5
command	0x01	0x43	0x00	0x01	0x00
data	0xff	0x41	0x5a	D_1	D_2

Table 3.4: Switch controller to configuration mode (controller is in digital mode)

Now that the controller is in configuration mode the following 3 packets must be issued.

The packet shown in Table 3.5 activates the analog mode after exiting the configuration mode. The value `0x01` for the fourth byte actually selects the analog mode. Using a value of `0x00` would allow to switch the controller from analog mode back to digital mode. The value `0x03` for the fifth byte disables the analog button on the controller. If this button is not disabled the user could manually switch the controller back to digital mode.

byte	1	2	3	4	5	6	7	8	9
command	0x01	0x44	0x00	0x01	0x03	0x00	0x00	0x00	0x00
data	0xff	0xf3	0x5a	0x00	0x00	0x00	0x00	0x00	0x00

Table 3.5: Activate analog mode

Next, the packet shown in Table 3.6 enables the motors inside the controllers. Once enabled they can be controlled using the fourth and fifth byte of the main polling command (see below).

byte	1	2	3	4	5	6	7	8	9
command	0x01	0x4d	0x00	0x00	0x01	0xff	0xff	0xff	0xff
data	0xff	0xf3	0x5a	0x00	0x01	0xff	0xff	0xff	0xff

Table 3.6: Enable motor control

Finally, the configuration mode is left (see Table 3.7). The controller should now be in analog mode with the motor control enabled.

[18] If the controller would already be in analog mode, the packet would be 9 bytes long.

byte	1	2	3	4	5	6	7	8	9
command	0x01	0x43	0x00	0x00	0x5a	0x5a	0x5a	0x5a	0x5a
data	0xff	0xf3	0x5a	0x00	0x00	0x00	0x00	0x00	0x00

Table 3.7: Exit configuration mode

After the controller is configured correctly, the main polling command can be issued with a specific rate to continuously readout the state of the buttons and the analog sticks. Table 3.8 shows how this packet looks like. The values M_S and M_L control the small and large motor, respectively. The small motor is activated (i.e., made to spin) by a value of `0xff`, `0x00` disables it (don't use any other values for this byte). For the large motor (almost) the full 8-bit value range can be used, where `0xff` corresponds to strongest rumble. Values below `0x40` don't active the motor.

The bytes X_R, Y_R, X_L and Y_L return the X/Y displacement of the right and left analog sticks.

byte	1	2	3	4	5	6	7	8	9
command	0x01	0x42	0x00	M_S	M_L	0x00	0x00	0x00	0x00
data	0xff	0x73	0x5a	D_1	D_2	X_R	Y_R	X_L	Y_L

Table 3.8: Main Polling Command (analog mode)

General Module Behavior and System Integration: Your dualshock_ctrl shall essentially behave exactly like the precompiled_dualshock_ctrl used for Exercise 1. When a controller is connected it shall be configured as discussed in the previous section and then continuously polled to update the ctrl_data output. Note that the controller may be disconnected and reconnected at any given time during operation. The values for the bytes M_S and M_L of the polling command are taken from the inputs small_motor and big_motor, respectively.

The frequency used for the ds_clk signal of the controller interface is configured by the generic DS_CLK_FREQ. The generic CLK_FREQ specifies the frequency of the clock at the clk input. You will need this information, in order to determine how many clk cycles a ds_clk cycle entails. After polling the controller and updating ctrl_data wait for REFRESH_TIMEOUT clk cycles before you issue the next polling command.

Don't forget to use synchronizers for input signals!

Use the file `dualshock_ctrl.vhd` for your implementation. If you want to create sub-modules, put them into the same directory. Add an instance of your dualshock_ctrl to your top-level design and remove the precompiled_dualshock_ctrl. Connect it to the relevant signals and configure the REFRESH_TIMEOUT generic to the equivalent of 10 ms. If everything works, the overall design should behave exactly as before.

Hint: As a first step, try to poll the controller in digital mode (i.e., only issue the command shown in Table 3.2). Only if that works, configure the controller for the analog mode.

Measurement and Debugging: There is a PlayStation 2 in the lab with a MSO-X 6004A mixed-signal oscilloscope attached to its controller port. You can use this setup to analyze how the

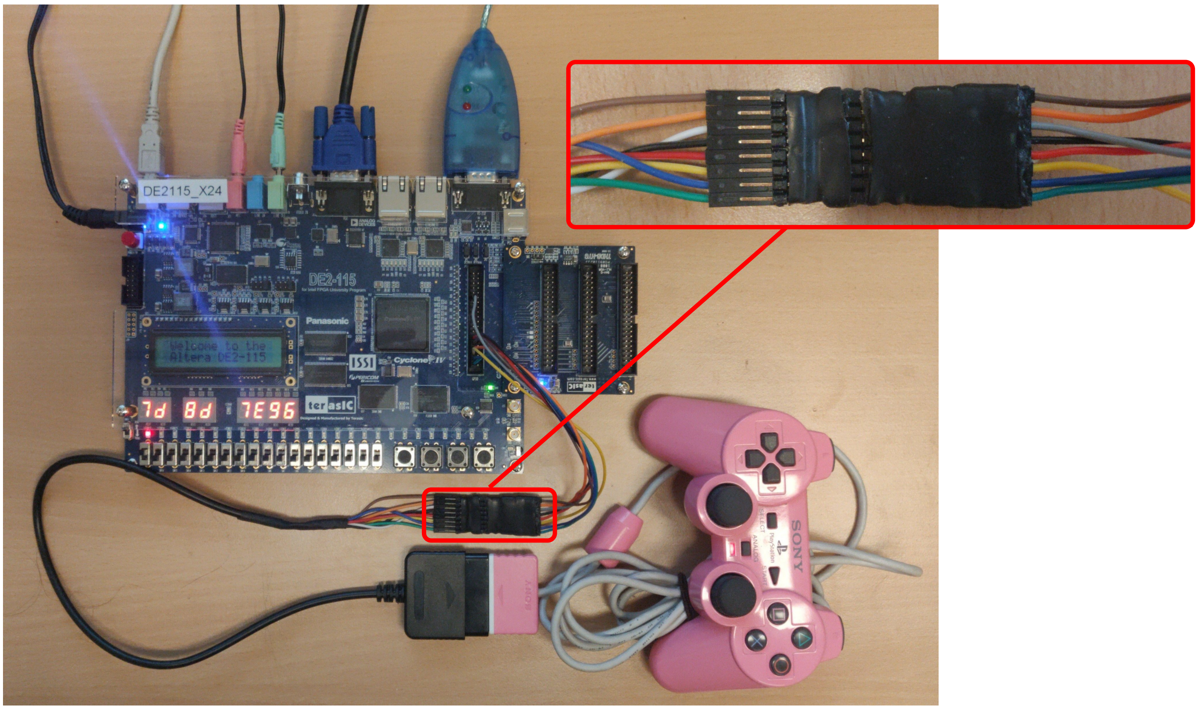


Figure 3.5: Connection points for the oscilloscope

PlayStation interacts with the controller and e.g., check if your signal timings are correct. Please, do not disconnect the PlayStation from the scope or the monitor and don't move it around.

You can also connect an oscilloscope to one of the FPGA boards to observe and analyze your design in operation. Figure 3.5 shows the connector where the scope can be attached to. Use the digital probes of the scope and connect it to the signals ds_clk, ds_cmd, ds_data, ds_att and ds_ack. The signals can be identified using the colors listed in Figure 3.3. If you are unsure about how to connect the scope please consult a tutor, as a wrongly connected scope can damage the devices involved!

For the lab report make a measurement showing the main polling command in the analog operation mode (i.e., the whole 9 byte packet). Press some of the buttons and move the sticks into some random position for this measurement. Make screenshots and decode the transmitted bytes – the report contains an appropriate template.

Please note that in the Remote Lab the (16) digital inputs of the scope are attached to the aux output of the top-level architecture. Hence, in order to make a measurement there, you have to connect the relevant signals in the top-level architecture with these outputs.

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 following port signals of the game module:

- gfx_cmd
- gfx_cmd_wr
- gfx_cmd_full

- gfx_rd_data
- gfx_rd_valid

Setup a trigger that is activated when a GET_PIXEL command is issued. Include a screenshot of the trigger condition in your lab report. Measure the time (in clock cycles) it takes from the time a GET_PIXEL command is issued until the gfx_rd_valid signal is asserted (include a screenshot of the signal traces in your lab report). Repeat this measurement 8 times and record the respone times in the table provided in the report template. Give an answer for the question why this value is not constant and what contributes to the response time.

Bonus points will only be awarded if at least one of the other tasks is implemented (mostly) correctly.

3.4 Submission

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

```
cd path/to/ddca_ss2023/dd
make submission_exercise2
```

The makefile creates a file named `submission.tar.gz` which contains all the required files. The submission script automatically checks if all the 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`
    - `vga_gfx_ctrl`
    - `game`
    - `[... 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.

Revision History

Revision	Date	Author(s)	Description
2.1	12.04.2023	FH	Added Task 2 and Bonus Task of Exercise II
2.0	07.04.2023	FH	Added Task 1 of Exercise II
1.2	19.03.2023	FH	Added pyyaml to the package list, improved installation instructions
1.1	13.03.2023	FH	Updated pip install commands
1.0	09.03.2023	FH	Initial version

Author Abbreviations:

FH	Florian Huemer
FK	Florian Kriebel