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 20, 2022

1 Introduction

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

- Exercise I: 01.04.2022, 23:55
- Exercise II: 05.05.2022, 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 "Tetris" clone[1], which uses a Nintendo GameCube 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, you can download a free version of Quartus (Quartus Prime Lite Edition) and Questa/Modelsim (ModelSim-Intel) from the Intel website[4]. However, note that the simulation performance of ModelSim-Intel is considerably lower than the full version of Questa/Modelsim provided in the lab (especially for large designs).

We also provide 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_04032022.txz`. Extract the archive using e.g., `tar -xf ECS-EDA-Tools_vm_DDMMYYYY.txz`

The root/user password of the VM is `ecseda`, change it using the `passwd` command.

[1]https://en.wikipedia.org/wiki/Tetris
[2]https://en.wikipedia.org/wiki/GameCube_controller
[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 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.`
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`
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: 01.04.2022)

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 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, simpler questions can be answered by the staff and you don't 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 2.3-2.5.

Project Creation: Create a new Quartus project in the `top/quartus/` directory. 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 (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.

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 precomplied modules. This includes the precompiled_gc_cntrl, the precompiled_vga_gfx_cntrl, the audio_cntrl and the dbg_port. These modules always come with one or more `*.vhd` files as well as a `*.qxp` and 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_cntrl the files `audio_cntrl_top.qxp`, `audio_cntrl_2s.vhd` and `audio_cntrl_pkg.vhd` are needed.

Note that the tetris_game module internally uses IP cores from the `ram/`, `decimal_printer/` and `tetris_util/` directories. Thus, be sure to also add those source files 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.

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. However, 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_cntrl 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.

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 Tetris game system. The "heart" of the system is the tetris_game module which implements the actual game logic, processes controller input, sends instructions to the vga_gfx_cntrl and plays sounds using the audio_cntrl. 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_cntrl implements a simple synthetic sound generator that interfaces with the board's audio DAC (digital to analog converter) WM8731. The vga_gfx_cntrl processes graphics instructions from the tetris_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 frame buffers. 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.

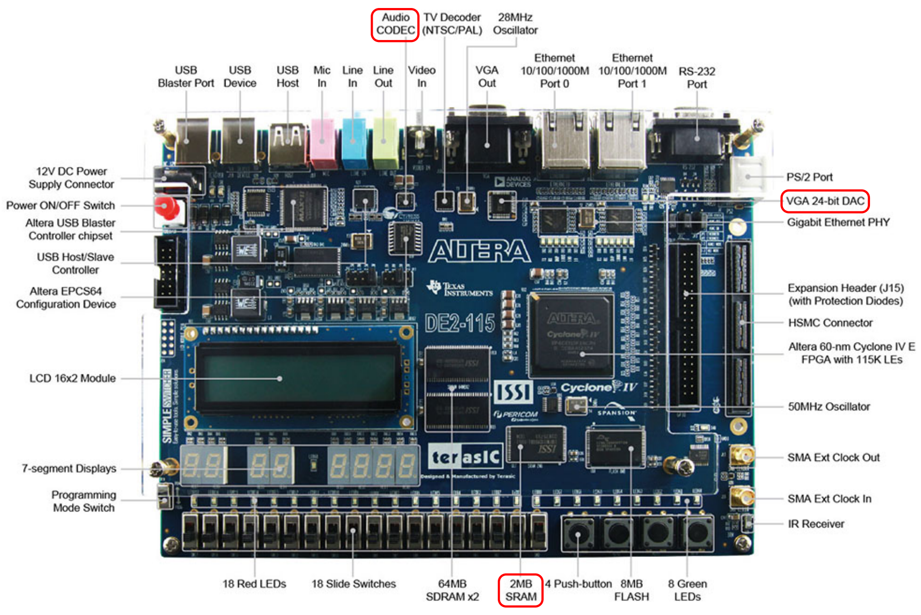


Figure 2.1: DE2-115 FPGA Development Board

The precompiled_gc_cntrl implements the interface to the GameCube controller. In the next Task, 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 a tetromino[5] that can be moved in a certain section of the screen using the arrow keys of the GameCube controller. The seven-segment displays should display the current x and y displacement of the joysticks, while the green LEDs show the state of (some of) the controller buttons. The state of the switches should be directly reflected by the red LEDs.

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

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

From Figure 2.5 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

[5]`https://en.wikipedia.org/wiki/Tetromino`

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 (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, of course you can also do this via SSH) to install the required Python packages.

```
pip3 install --user termcolor dataclasses docopt pyserial
```

Now you can execute the following commands:

```
remote.py -s # read the current state of the (hardware) switches
remote.py -s 0x2aaaa # set the software switches to an alternating pattern
remote.py -s # read back the value just set
```

After running these commands, the red LEDs 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 buttons
remote.py --swic 1 # manually enable the software buttons
remote.py --swic # read the current value of the software input control flag
```

Figure 2.5 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 GameCube controller, such that the design can also be operated in the Remote Lab. This is the reason for the emulated_gc_data signal of the top-level entity. The boards in the Remote Lab have this (inout) signal connected with gc_data using an external wire. The boards at the normal workplaces in the lab have the physical GameCube controller connected at gc_data. To control the state of the emulated controller `remote.py` offers the `-n` command line argument, which lets you access a 64-bit value. Individual buttons and analog inputs can be accessed using the -b command line option:

```
remote.py -n 0 # reset all buttons and analog values to zero
remote.py -n -b A 1 # press the A button
remote.py -n -b S 1 # press the Start button
remote.py -n -b Left 1 # press the left arrow button
remote.py -n -b LT 0xdd # set the left trigger to 0xdd
remote.py -n -b JX 0xca # set the x axis of the joy stick to 0xca
remote.py -n -b CY 0x22 # set the y axis of the c stick to 0x22
```

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 Exercise II, is the ability to send graphics instructions to the vga_gfx_cntrl. Figure 2.4 shows that the gfx_instr and gfx_instr_wr outputs of the tetris_game module are not directly connected to the vga_gfx_cntrl, but that there

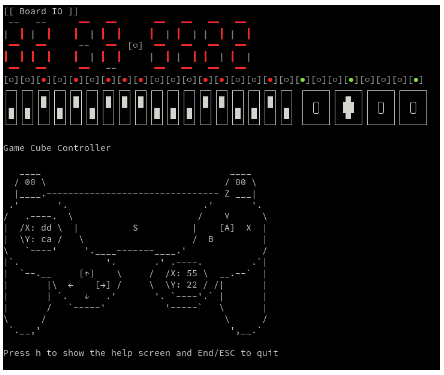


Figure 2.2: Screenshot of the interactive mode of the `remote.py` tool

are multiplexers in between. These multiplexers are controlled by the graphics instruction source control signal (gisc) coming from the dbg_port. If this signal is asserted, the multiplexers relay the signals dbg_gfx_instr and dbg_gfx_instr_wr of the dbg_port to the vga_gfx_cntrl. To issue instructions using the `remote.py` tool, use the `--gfx` command line option, which will automatically assert the gisc signal. Executing the following script will color the screen green and then draw a diagonal line in red.

```
remote.py --gfx 0x7000 # disable double buffering
remote.py --gfx 0xf003 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0xf800 0x07e0  # load palette 3
remote.py --gfx 0xe003 # select palette 3
remote.py --gfx 0x4e00 # clear the screen
remote.py --gfx 0x1000 0 0 # set graphics pointer register to (0,0)
remote.py --gfx 0x3f00 319 239 # draw the line
```

For more information about the instruction format, refer to IP Cores Manual. Note that all those instructions could also have been issued using just a single call to `remote.py`. To deassert the gisc signal run `remote.py --gisc 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. 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 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`.

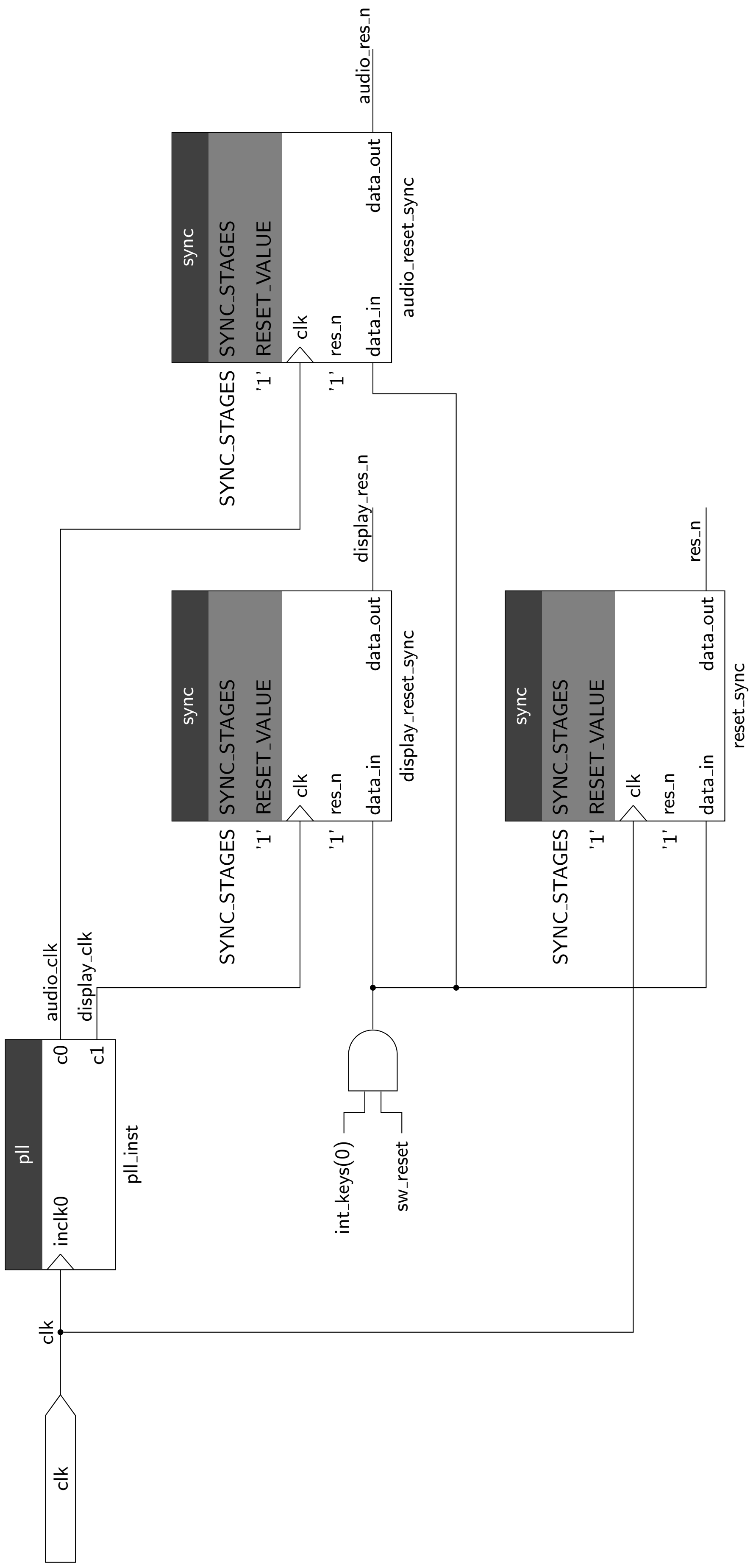


Figure 2.3: Structural system specification (clock and reset)

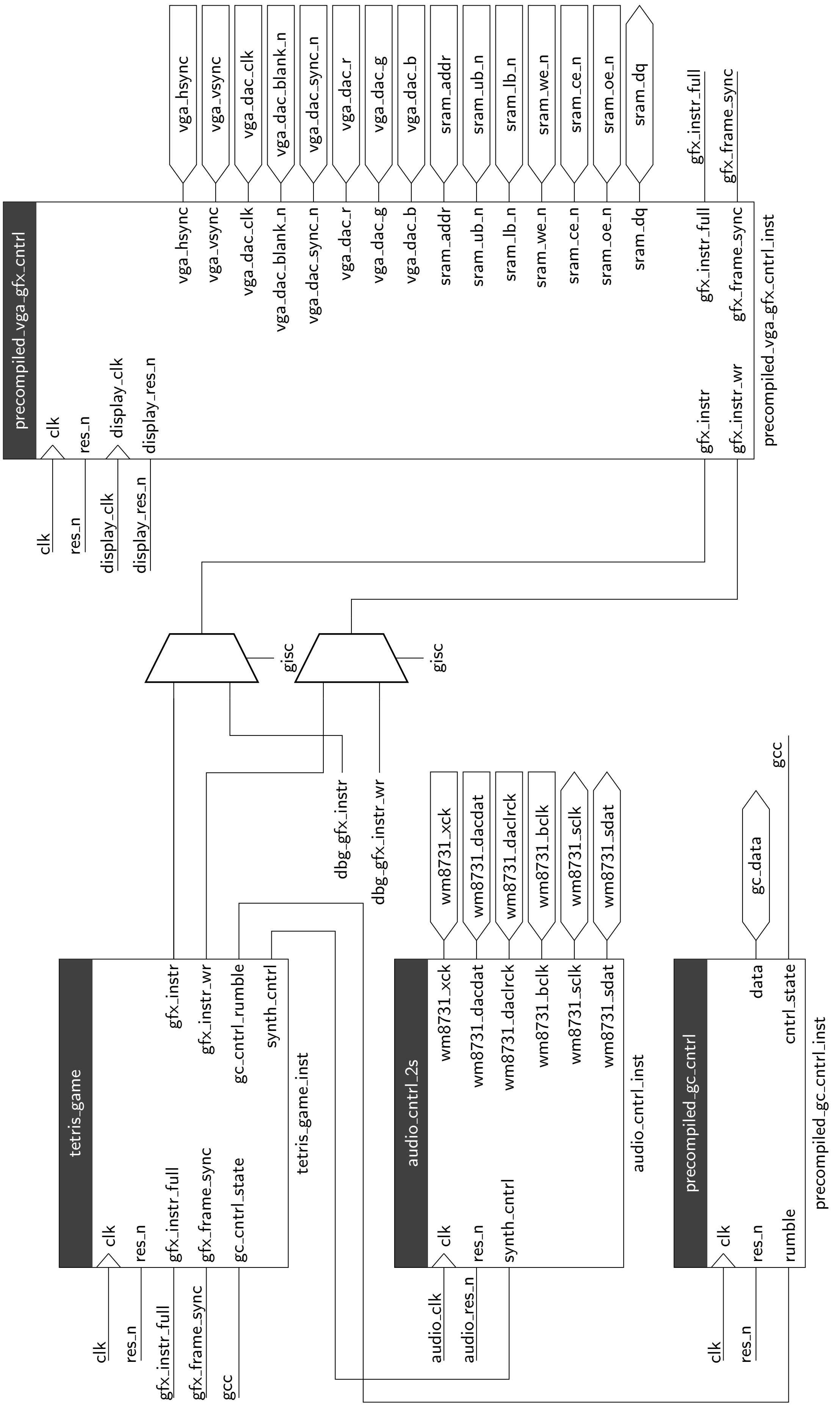


Figure 2.4: Structural system description (core system components)

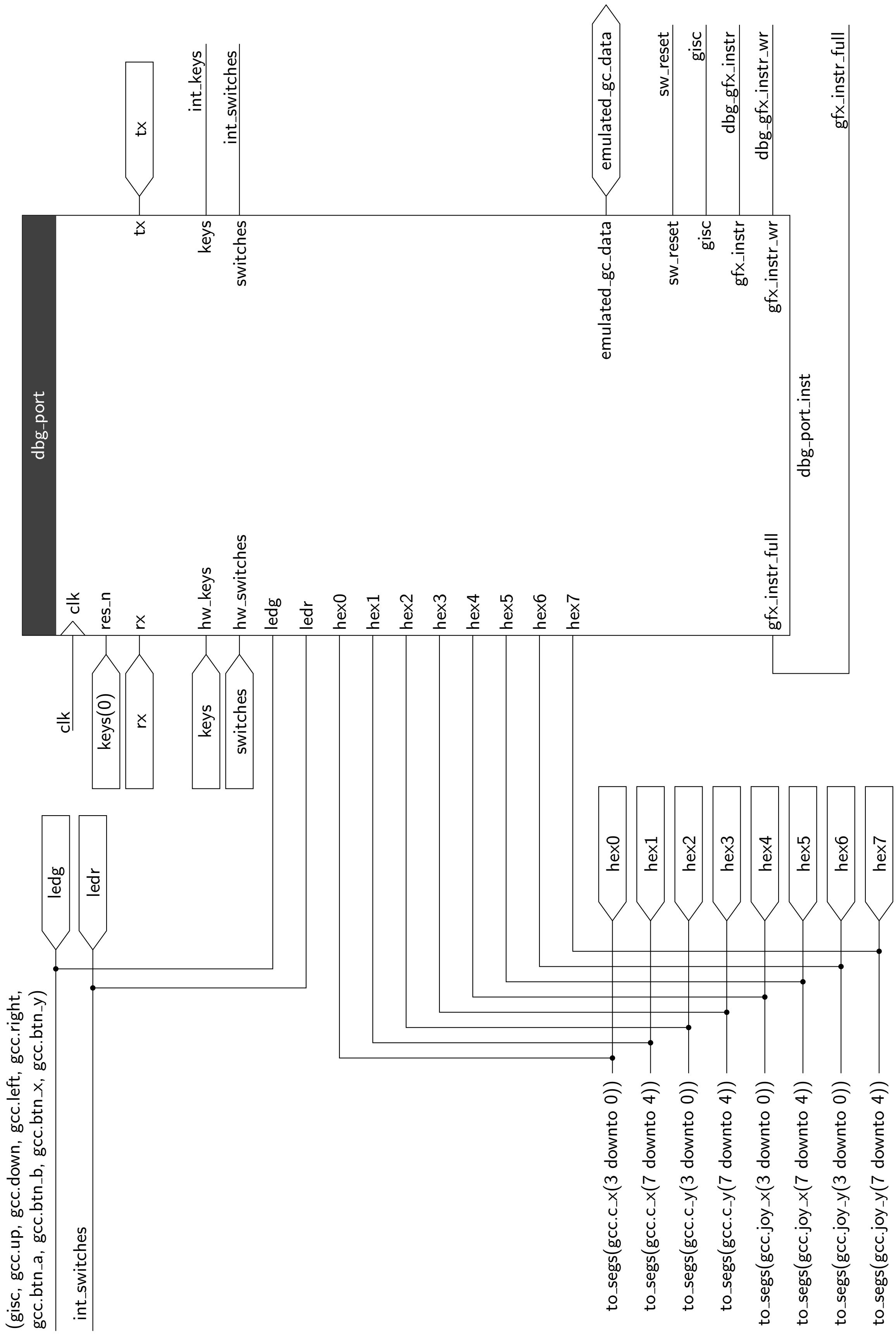

dbg_port
dbg_port_inst
clk
res_n
rx
hw_keys
hw_switches
ledg
ledr
hex0
hex1
hex2
hex3
hex4
hex5
hex6
hex7
gfx_instr_full
tx
keys
switches
emulated_gc_data
sw_reset
gisc
gfx_instr
gfx_instr_wr
int_keys
int_switches
dbg_gfx_instr
dbg_gfx_instr_wr
keys(0)
(gisc, gcc.up, gcc.down, gcc.left, gcc.right, gcc.btn_a, gcc.btn_b, gcc.btn_x, gcc.btn_y)
to_segs(gcc.c_x(3 downto 0))
to_segs(gcc.c_x(7 downto 4))
to_segs(gcc.c_y(3 downto 0))
to_segs(gcc.c_y(7 downto 4))
to_segs(gcc.joy_x(3 downto 0))
to_segs(gcc.joy_x(7 downto 4))
to_segs(gcc.joy_y(3 downto 0))
to_segs(gcc.joy_y(7 downto 4))

Figure 2.5: Structural system description (debug port)

Task 2: GameCube Controller [40 Points]

In this task you will implement the interface to the GameCube controller by yourself and replace the precompiled_gc_cntrl. The GameCube uses a simple serial interface to transfer the button states, analog triggers and joy stick values over a single (half-duplex) wire data. As shown in Figure 2.6 a pull-up resistor is used to connect this wire to the supply voltage (3.3 V)[6]. This means that unless no communication partner actively drives data low, the idle state of the wire is high. Although Figure 2.6 shows the pull-up resistor as an external component to the FPGA, it is actually an internal resistor integrated into the I/O cell of the chip that can be activated by the FPGA design. You don't have to take care for this resistor, since this has already been configured by the pinout file.

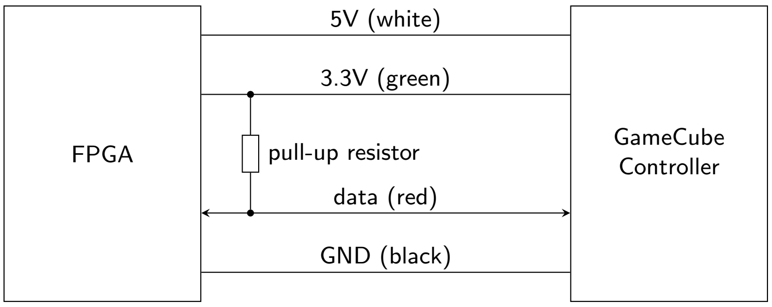


Figure 2.6: Physical connection between GameCube controller and FPGA

Depending on the current state of the communication protocol, the data wire is either driven low by your design (i.e., the FPGA) or the GameCube controller. Note that there is no need to ever actively drive this wire high, since this is already the idle state thanks to the pull-up resistor[7]. It is important that whenever one communication party drives the output, the receiver goes into the high-Z state, such that the transmitted value can be read.

To start a new transmission to read the current state (pressed/not pressed) of each button and the analog values of the triggers and joy sticks (x and y displacement), a 24-bit long polling command ("0100 00000000 0011 0000 000R") followed by a single (high) stop-bit must be sent to the controller . The individual bits are transmitted using a pulse width encoding. The transmission of a single bit takes 4 μs and always starts with a falling edge on data. A logic zero is represented by a rising edge after 3 μs (measured from the start of the bit, i.e., the falling edge), while for a logic one the rising edge must happen after 1 μs. Using this scheme the receiver can always synchronize to the falling edge of the data signal without the need for an explicit clock signal. Notice that one bit in the polling command is marked with R. This bit controls the rumble feature of the controller. Setting it to one will activate rumbling until the next transmission cycle. Figure 2.7 visualizes this protocol.

After sending the polling command, the GameCube controller immediately answers with a 64-bit response using the same encoding, again followed by a stop bit. In order to receive this data, the FPGA has to switch off its output driver (i.e., output 'Z') immediately after the end of the polling command. Table 2.1 lists the meaning of the individual bits of the controller's response.

Implementation: Create an entity named gc_cntrl and place it in the `gc_cntrl/src/gc_cntrl.vhd` file. The required generics and the port signals are described in Tables 2.2 and 2.3, respectively.

[6] The additional 5 V supply line is required to drive the rumble motor in the controller.

[7] Driving a signal high that is driven low by another communication partner leads to a short-circuit that can possibly damage one or both of the communication partners!

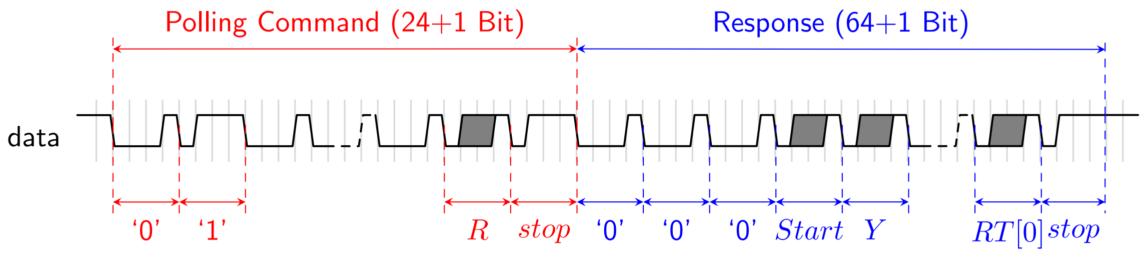


Figure 2.7: GameCube controller serial interface protocol

Bit	Content
0-2	unused (always 0)
3	Start
4	Y
5	X
6	B
7	A
8	unused (always 1)
9	L
10	R
11	Z
12	Up
13	Down
14	Right
15	Left
16-23	Joy Stick X-Axis (16 MSB)
24-31	Joy Stick Y-Axis (24 MSB)
32-39	C Stick X-Axis (32 MSB)
40-47	C Stick Y-Axis (40 MSB)
48-55	Left Trigger (48 MSB)
56-63	Right Trigger (56 MSB)
64	Stop bit (always 1)

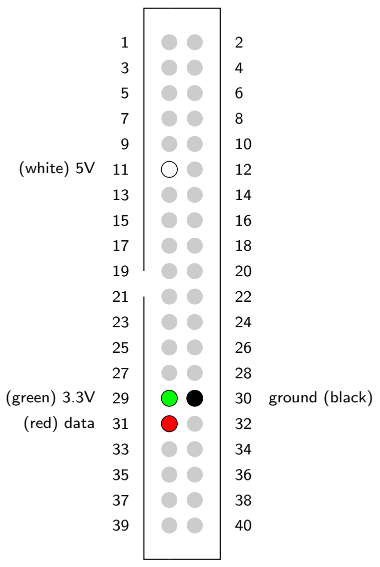
Table 2.1: Response of the GameCube controller

Name	Functionality
SYNC_STAGES	Number of flip flop stages used for the synchronization of the data signal
CLK_FREQ	Actual clock frequency of the *clk* signal given in Hz
REFRESH_TIMEOUT	The timeout in clk cycles the controller should wait in between controller state readouts. Set this generic to the equivalent of 6 ms.

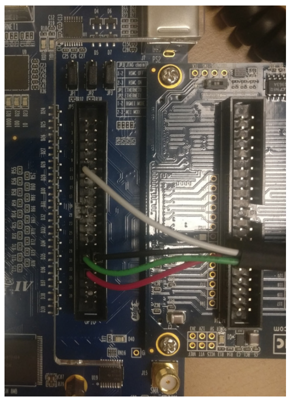
Table 2.2: gc_cntrl generics description

The state diagram of the gc_cntrl is shown in Figure 2.9. Implement a state machine according to this specification. The edges in this graph are labeled with the relevant condition that needs to be satisfied for the state change. Additionally the edge labels also contain assignments (indicated by the := operator) that must be executed when the condition is fulfilled. The initial state is WAIT_TIMEOUT.

Since, from the point of view of the FPGA the data signal is asynchronous, it must be syn-



(a) GPIO board connector pinout (top view)



(b) Connected wires for the GameCube controller

Figure 2.8: Physical controller/board interface

Name	Dir.	Width/Type	Functionality
clk	in	1	Global clock signal
res_n	in	1	Global reset signal (low active, not internally synchronized)
data	inout	1	The serial half-duplex data wire to the controller
rumble	in	1	The rumble-control signal.
cntrl_state	out	gc_cntrl_state_t	The actual state of the controller, represented by record type defined in the gc_cntrl_pkg package.

Table 2.3: gc_cntrl port signals description

chronized before it can be processed in the FSM. Hence you have to instantiate a synchronizer for this signal. In Figure 2.9 the output of this synchronizer is referred to as data_synced. The signal data_synced_old holds to the value of data_synced from the previous cycle (you will need a register for that).

The 25-bit register poll_cmd must be initialized to the value of the polling command (and the stop bit) discussed above. Note that the polling command is transmitted MSB first. The FSM always rotates the bits in the poll_cmd register to the left, such that next bit to transmit is always at poll_cmd(24).

Note that the cntrl_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, don't use a register for the data output. Don't forget to initialize all used registers during reset.

The ONE_US constant used in Figure 2.9 is the duration of a μs in clock cycles of clk. You have to calculate this constant from the generics.

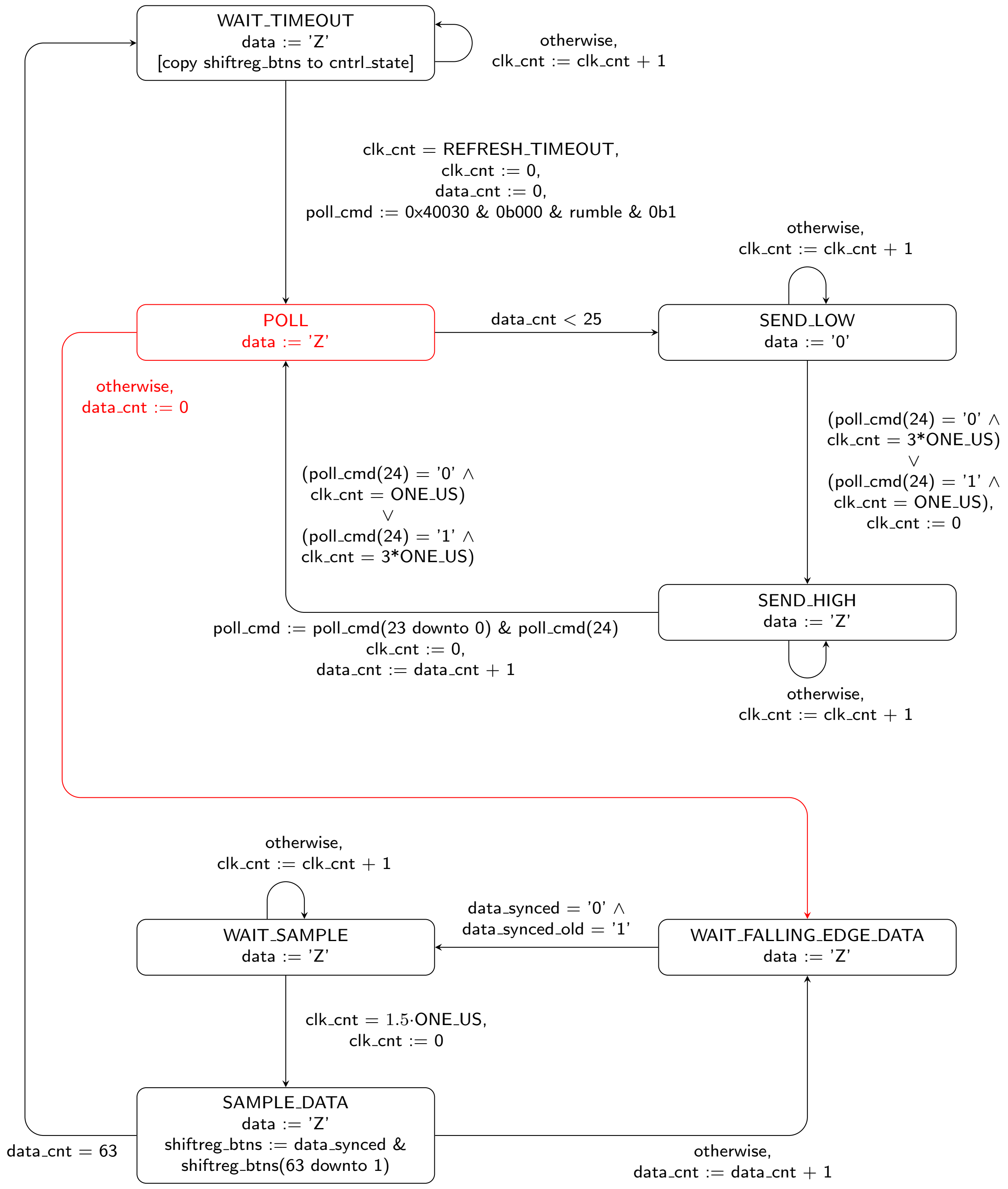


Figure 2.9: gc_cntrl state diagram

Simulation: After implementation use the template in `gc_cntrl/tb/gc_cntrl_tb.vhd` to create a testbench for the gc_cntrl entity.

Your testbench shall simulate 8 controller state transmissions. For that purpose the testbench shall wait for the polling command and then generate appropriate input data on the data input of the controller. Your testbench shall be able to detect when the polling command is not correct and raise an error. After the transmission is complete and the cntrl_state output of the gc_cntrl is updated, your testbench shall again check if the received data matches the transmitted data. If this is not the case for any of the entries in cntrl_state, an error should be raised. To raise an error use a `report` statement with the severity `error` (e.g., `report "polling command wrong" severity error;`). You don't have to use the 6 ms value for the REFRESH_TIMEOUT generic in the testbench. You may set it to a value that seems appropriate for your simulation.

The test vectors that your testbench uses for the data transmission, shall be generated randomly. The testbench template already contains code, which lets you create random gc_cntrl_state_t values. Initialize the seed variables (seed1 and seed2) to your matriculation number. Before you change anything in the template, try running a simulation with it.

Be careful when processing and writing the data signal. Since the the pull-up resistor is not simulated, 'Z' must be interpreted as high for the polling command. For the same reason you have to actively drive the data input (i.e., set it to '1') when you generate your response.

To automate the compilation and simulation process use the makefile example, provided in the `ram/` directory, to create your own makefile-based simulation flow. The makefile for the gc_cntrl entity shall be placed in `gc_cntrl/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, load an appropriate waveform viewer configuration script to add the relevant signals to the waveform viewer (`gc_cntrl/scripts/wave.do`) and run the simulation for long enough such that all 8 state transmissions are visible.

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 will run through without any errors being reported. In case the unit under test (UUT), i.e., the gc_cntrl, does not work as expected, the problem will 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 *all* files generated during the compilation and simulation process.

System Integration: Add an instance of the gc_cntrl to your top-level design (`top/src/top.vhd`) and remove the precompiled_gc_cntrl. Connect it to the relevant signals and configure the REFRESH_TIMEOUT generic to the equivalent of 6 ms. If everything works, the overall design should behave exactly as before.

Task 3: Decimal Printer [40+(10) Points]

In this task you will design and implement the state machine for the decimal_printer module that will eventually be used to display the player points in the finished game.

Hence, the input to this module will be an unsigned (16-bit) number alongside a screen position. The output shall be a sequence of graphics instructions (see the IP Cores Manual for details) that can be fed into the vga_gfx_cntrl and that will cause it to print the decimal representation of the number to the screen at the desired position.

Implementation: The file `decimal_printer/src/decimal_printer.vhd` already contains a template for your module. Don't change the interface of the entity in any way!

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

    gfx_instr : out std_logic_vector(GFX_INSTR_WIDTH-1 downto 0);
    gfx_instr_wr : out std_logic;
    gfx_instr_full : in std_logic;

    start : in std_logic;
    busy : out std_logic;
    x : in std_logic_vector(15 downto 0);
    y : in std_logic_vector(15 downto 0);
    number : in std_logic_vector(15 downto 0)
  );
end entity;
```

Initially the FSM should start in a state where it **waits** for a logic one at the start input and then begins the conversion process. The unsigned 16-bit value at the number input shall be converted from its binary representation to a BCD value[8]. For each digit generated this way, a BIT_BLIT instruction[9] shall be used to print the digit to the screen. As long as your core is processing and/or generating graphics instructions, the busy signal must be asserted. Figure 2.10 visualizes this interface protocol.

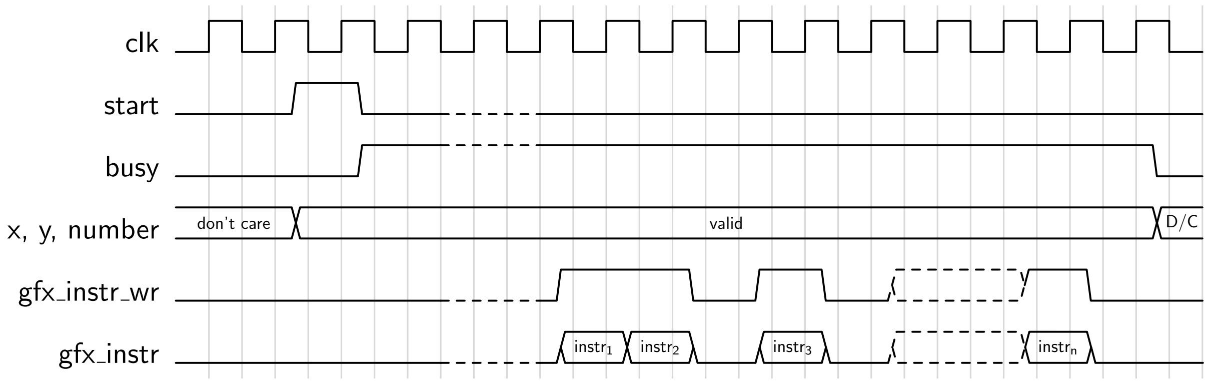


Figure 2.10: Example timing diagram for the decimal_printer

Note that the inputs number, x and y must be applied at the same cycle as the start signal is asserted and must be kept stable until the busy signal is deasserted by the core. Be sure that your FSM respects the gfx_instr_full signal, not shown in the timing diagram. This means that it must never be the case that gfx_instr_wr gets asserted while gfx_instr_full is asserted. There is no constraint on the processing time of the state machine or the number or type of graphics instructions to be used.

The conversion itself **must not** be implemented using a division operation, but by successively subtracting decimal powers (i.e., once per clock cycle) from the binary value. Start by subtracting 10^4 until the value is smaller or equal to 9999. By counting the number of times 10^4 could be subtracted, the ten-thousands digit is obtained. Now repeat the process by subtracting 10^3 to obtain the thousands digit and so on.

The decimal number shall be positioned at the screen coordinate defined by the inputs x and y. These coordinates mark the position of the upper left corner of the ten-thousands digit. Hence,

[8]https://en.wikipedia.org/wiki/Binary-coded_decimal
[9]Of course also the compact version of the BIT_BLIT instruction can be used.

the first graphics instruction generated by your core will have to be a MOVE_GP instruction, that loads these coordinates into the graphics pointer register. Don't print leading zeros. Figure 2.11 shows the result of printing the numbers 12345 and 123 to the position $(x, y)=(10, 10)$.

Don't use the all keyword for the implementation of the decimal_printer. Create explicit sensitivity lists for your processes.

Simulation: To test your decimal_printer in simulation, implement the gfx_instr_interpreter whose entity declaration is shown below:

```vhdl
entity gfx_instr_interpreter is
  generic (
    OUTPUT_DIR : string := "./";
    BB_ROM : bb_rom_t := MEMORY_CONTENTS
  );
  port (
    clk : in std_logic;
    gfx_instr : in std_logic_vector(GFX_INSTR_WIDTH-1 downto 0);
    gfx_instr_wr : in std_logic;
    gfx_frame_sync : out std_logic;
    dump_frame : in std_logic
  );
end entity;
```

A template is already provided in the file decimal_printer/tb/gfx_instr_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_cntrl. This means that this module is **not** meant to be synthesizeable. It should just receive graphics instructions and immediately execute them by performing the appropriate graphic operations on an internal image data structure.

In order to visualize the data stored in this data structure, the gfx_instr_interpreter features the dump_frame input. Whenever this signal exhibits a rising edge, an image shall be dumped that corresponds to what the vga_gfx_cntrl would currently display. For that purpose we use the ASCII version of the Portable PixMap[10] format (magic number: P3). The file names of the dumped images shall be frame_[N].ppm, where [N] is a number increased with every frame dump, starting at 0. The image files should be placed in the directory specified by the generic OUTPUT_DIRECTORY.

Your gfx_instr_interpreter should support *all* instructions supported by the vga_gfx_cntrl as documented in the IP Cores Manual. The only exceptions are the DRAW_LINE and the CFG instructions. You don't have to draw anything when a DRAW_LINE instruction is encountered. However, you still need to advance the graphics pointer correctly. Moreover, you can treat GFG instructions as NOP instructions, i.e., you don't need to support double buffering or switching the output to the test pattern. If you are unsure how certain instructions should behave, you can use the graphics instruction interface of the dbg_port to test them on the "real" vga_gfx_cntrl in hardware. Be sure to have a look at the vga_gfx_cntrl_pkg as it contains constants and functions that can be helpful for your implementation.

Use the decimal_printer/tb/decimal_printer_tb.vhd as a template for your testbench for the decimal_printer. This file already contains instances for both the decimal_printer and the gfx_instr_interpreter. Before triggering the decimal_printer via the start signal, load an appropriate color palette and clear the screen. You can use the signals init_gfx_instr and init_gfx_instr_wr for that purpose. Then set up the decimal_printer in such a way that it prints your matriculation number modulo 2^{16} at the position $(x, y) = (10, 10)$. Then dump the generated frame to decimal_printer/frame_0.ppm. The generated image may also contain other output created to

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

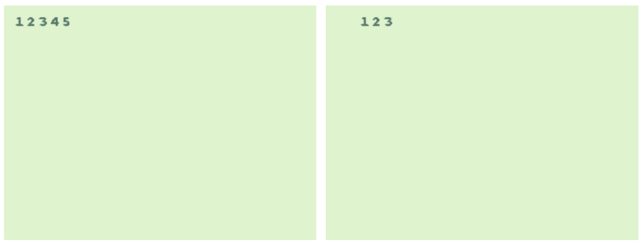


Figure 2.11: Sample output for 12345 and 123

test other features of the gfx_instr_interpreter not used by the decimal_printer. Figure 2.11 shows how a valid result can look like for number values of 12345 and 123.

To start the simulation, create a Makefile with the targets `compile`, `sim` and `clean`, which should behave as described in the previous task.

System Integration: The tetris_game module already contains an instance of the decimal_printer, hence no action from your side is required. If you implemented the core correctly, the design should output the 16-bit value formed by the concatenation of the left and right triggers of the GameCube controller on the screen.

Bonus Task (10 Points): Fully implement the `DRAW_LINE` instruction for the gfx_instr_interpreter. For that purpose you can implement a version of Bresenham's line algorithm[11]. If you implement the Bonus Task, modify the testbench for the decimal_printer such that it draws a diagonal line from (0,239) to (319,0). Bonus points will only be awarded if the rest of the gfx_instr_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_ss2022/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
    ├── gc_cntrl
    ├── decimal_printer
    └── [... 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://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

3 Exercise II (Deadline: 05.05.2022)

3.1 Overview

In this exercise you have to implement (i) the complete game logic of the Tetris game and (ii) your own version of the VGA graphics controller.

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

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 (FPGA board, DAC, see TUWEL)
- SignalTap manuals

3.3 Task Descriptions

Task 1: VGA Graphics Controller [50 Points]

In this task you will implement your own version of the vga_gfx_cntrl, which was supplied as a precompiled module for Exercise I. Don't change its interface, i.e., its entity declaration in any way. You can add new instructions, however, don't change the predefined ones.

To implement this module you can use the overall structure described in the IP Cores Manual. For that purpose we already provide you with some IP cores that you can use for your implementation:

- Use the cores provided by the gfx_util_pkg package to implement the rasterizer sub module.
- The sram_cntrl lets you access the external SRAM on the FPGA board to store the frame buffers.
- To continuously read the SRAM we provide the frame_reader module in `vga_gfx_cntrl/src/frame_reader.vhd`.
- The test pattern is generated using the tpg provided in `vga_gfx_cntrl/src/tpg.vhd`.

Internal Interface: As described in the IP Cores Manual, the vga_gfx_cntrl implements a simple 16 bit wide FIFO-like instruction interface. Your graphics controller must support **all** of the instructions and features specified in the IP Cores Manual. This means that when presented with the same stream of graphics instructions, your core should produce the exact same output as the precompiled_vga_gfx_cntrl.

Use a FIFO provided by the ram_pkg package to buffer incoming instructions and expose its write port to the port signals gfx_instr, gfx_instr_wr and gfx_instr_full of your vga_gfx_cntrl. Internally you can then read the commands and operands from this FIFO and perform the appropriate actions.

To store the color palettes you can use the dp_ram_1c1r1w from the ram_pkg package.

Whenever the gfx_bb core produces a set of output coordinates on pixel_x and pixel_y and an associated color value on pixel_color, you have to take this color value and use it to access the palette memory in the next clock cycle. This will then give you the actual 16-bit color value to write to the frame buffer. There are essentially two ways how you can implement this. You can setup a processing pipeline that is fed by the gfx_bb core, performs the color lookup and outputs the final color value. The other possibility is to implement the lookup in a state machine and stall the gfx_bb core after each set of coordinates.

Video Memory: The vga_gfx_cntrl uses the external SRAM on the FPGA board to store its frame buffers. To interface with this memory, you can use the sram_cntrl documented in the IP Cores Manual.

To continuously read a frame from the external memory, in order to display it using the VGA controller (discussed in the next section), we provide you with the frame_reader module. This core can be directly attached to the read port of the sram_cntrl and handles data buffering, upscaling and implements the clock-domain crossing to the 25 MHz display clock domain used by the VGA Controller. As already explained in the IP Cores Manual, upscaling is required because the VGA Controller outputs a video signal with a resolution of 640x480. Hence every pixel of the 320x240 frame buffer is output four times.

Figure 3.1 shows an example timing diagram demonstrating how to interact with the frame_reader. Initially the core waits for a rising transition on the prepare_frame_start input, which signals the core that it should get ready to output data for a new frame and hence start pre-fetching pixel data from SRAM. Although this signal is sampled in the 50 MHz clock domain, it can also be driven by signals originating in the 25 MHz display clock domain (no synchronizer is necessary here). However, due to the time it takes to initially pre-fetch the first pixels of a frame, this transition must be applied **at least** 32 clock cycles before the first pixel can be read at pix_data_*. Note that prepare_frame_start may stay high for an arbitrary amount of cycles. You only have to make sure to deassert it **before** pix_ack goes high for the first time within a frame.

In response to the signal transition at prepare_frame_start the frame_reader produces a one-clock-cycle-long pulse (in the 50 MHz clock domain) on the frame_start output signal, which is intended to be used by the rasterizer to synchronize itself to the start of a frame. In the cycle following this pulse the core samples the vram_base_addr input, which (as the name suggests) is used as the base address in the SRAM to fetch the pixel data from. The frame_reader expects a certain memory layout. The (16-bit) color information for the pixel with the coordinates x and y must be located at the memory address

$$B + y * 320 + x,$$

where B is the base address configured using the vram_base_addr input.

After (at most) 32 cycles the frame_reader outputs the color information for the first pixel of the frame (i.e., the pixel at the coordinates (x, y)=(0,0)) at the 8-bit outputs pix_data_r, pix_data_g and

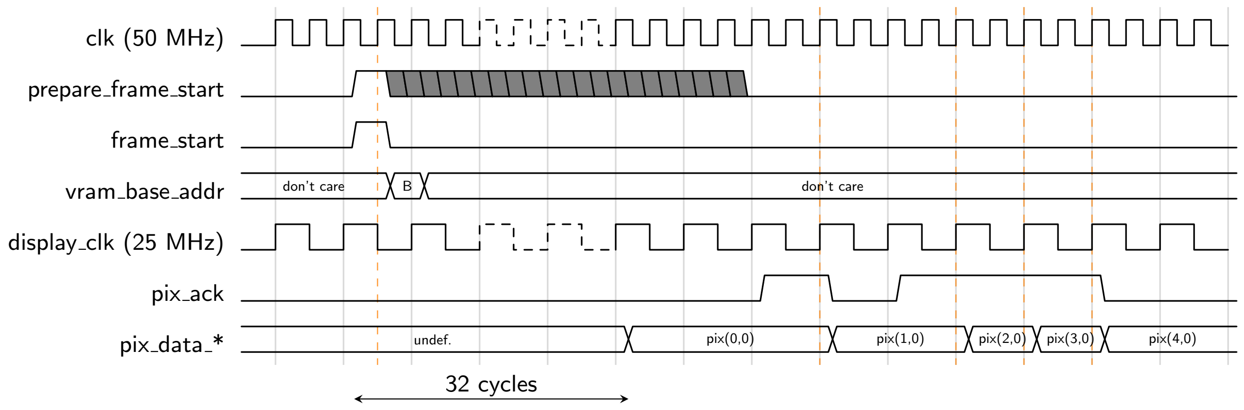


Figure 3.1: Frame Reader interface protocol

pix_data_b. It automatically converts the 5-6-5 pixel format stored in the SRAM into three separate 8-bit color channels. To get the color of the next pixel, the current one must be acknowledged. This is achieved by asserting pix_ack for one (25 MHz) clock cycle.

This interface is essentially the read port of a FIFO with first-word-fall-trough behavior (see IP Cores Manual). However, the valid signal is omitted here, because it is guaranteed that this FIFO will never run empty during a frame.

Note that **all** pixels of a frame must be read before prepare_frame_start can be asserted again. This means that the pix_ack signal must be asserted for exactly 640*480 (25 MHz) cycles during one frame.

External Interface: The external interface, which we refer to as the VGA Controller, has to communicate with the ADV7123 digital-to-analog converter (DAC) to generate the appropriate RGB analog component video signal, which is then output at the VGA connector of the board. For the sake of compatibility we use a video signal resolution of 640x480 pixels instead of the 320x240 pixels used for the frame buffer. Although we could also generate a video signal with the native resolution, not all monitors can deal with it and display it correctly. However, this is nothing you really need to worry about, since the upscaling is already handled by the frame_reader.

Figure 3.2 shows the physical video interface of the FPGA board. It can be seen that it essentially consists of five signals. The digital signals vga_hsync and vga_vsync are used to transport horizontal and vertical synchronization information.

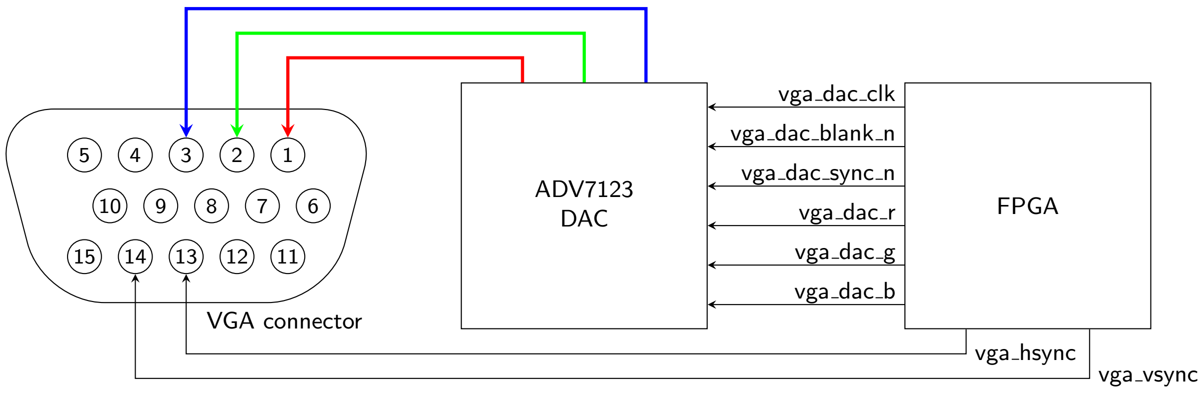


Figure 3.2: Video output circuity of the FPGA board

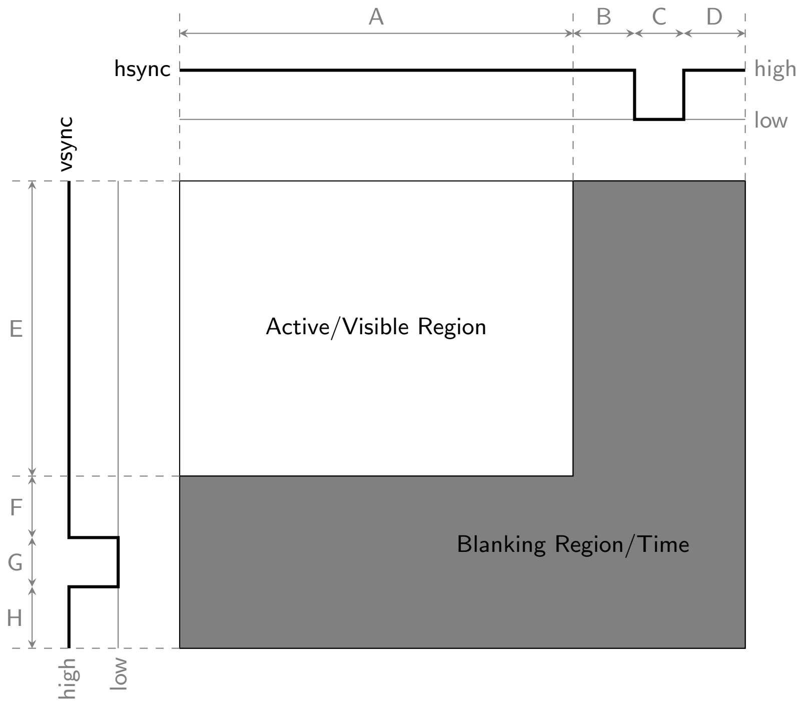


Figure 3.3: Synchronization signal timing

The actual color information for the individual pixels is transmitted using three analog signals (one for each color channel). Since the FPGA cannot directly output analog voltages, an external DAC is required, which is connected to the FPGA via the `vga_dac_*` signals.

Video information is transmitted frame by frame with a rate of approximately 60 frames per second. A frame is transmitted using a series of horizontal lines (in our case 525) starting at the top of the frame. Only a subset of these lines (in our case 480) actually represent the visible image, the rest is only needed for synchronization. A line has a duration of 800 25 MHz clock cycles (i.e., the frequency of `display_clk`)[12]. Pixel information is only transmitted in the first 640 cycles. Figure 3.3 shows the timing of a frame[13]. The names and the values of the symbols A-H can be found in Table 3.1.

Symbol	Parameter	Value	Unit
A	Horizontal Visible Area	640	cycles
B	Horizontal Front Porch	16	cycles
C	Horizontal Sync. Pulse Width	96	cycles
D	Horizontal Back Porch	48	cycles
E	Vertical Visible Area	480	lines
F	Vertical Front Porch	10	lines
G	Vertical Sync. Pulse Width	2	lines
H	Vertical Back Porch	33	lines

Table 3.1: Signal timings for 640x480 VGA

[12]Actually, the video mode we are using requires a pixel clock frequency of 25.175 MHz. However, for the sake of simplicity we use 25 MHz, as most displays are fine with this slightly off-spec frequency.

[13]We define the start of a frame with the visible portion of the frame. However, some explanations also use other points in time.

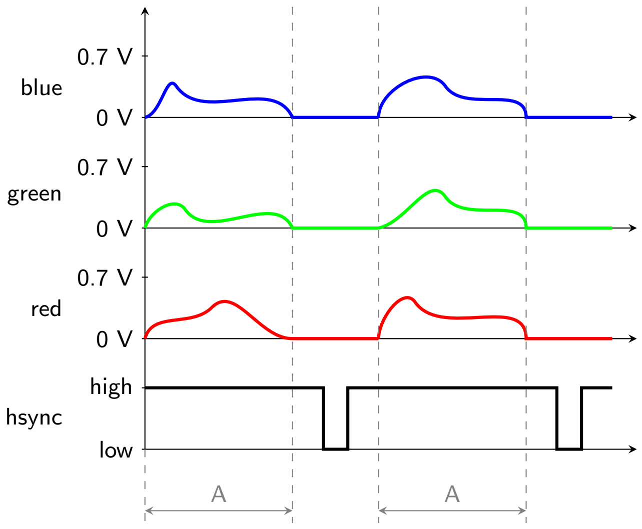


Figure 3.4: Timing diagram showing

Figure 3.4 shows the transmission of two successive visible lines. During the horizontal and vertical blanking region, the analog signals must stay at 0 V.

When interfacing with the ADV7123 DAC, directly connect display_clk to the output vga_dac_clk. The vga_dac_sync_n is not needed for this exercise and can be permanently set to `'1'`. Be sure to accommodate the fact that the digital-to-analog conversion is not instantaneous and delay your synchronization signals appropriately. The datasheet for the DAC is available in TUWEL.

The template provides a test pattern generator (tpg) in the file `vga_gfx_cntrl/src/tpg.vhd`, which you can use to test your VGA signal without the need to implement the rest of the graphics controller. The interface of the tpg is exactly the same as for the frame_reader. The test pattern is shown in Figure 3.5. What is not visible in this picture is that the test pattern has a white border, i.e., all pixels with an x coordinate of 0 or 639 or a y coordinate of 0 of 479 are white.

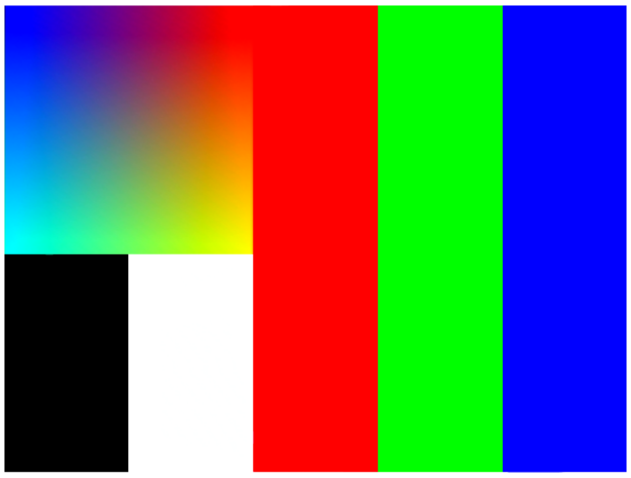


Figure 3.5: VGA test pattern as generated by the test pattern generator

Testing: To test all features of your graphics controller you can use the `--gfx` command line argument of the `remote.py` script. You can use this command in the Remote Lab and locally in the TILab.

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 0x1234
remote.py --gfx "0x1000 | 0x200 | 0x03 | 0x4"
```

Note, that when you issue graphics instructions using the `--gfx` command line argument, the dbg_port automatically sets the gisc signal.

Oscilloscope Measurement: Perform an oscilloscope measurement to demonstrate the correctness of the VGA signal you are generating. For that purpose, configure the design to output the test pattern generated by the tpg.

If you are working in the TILab, connect the oscilloscope to an FPGA board using the provided cable. Connect the three color channels and the horizontal synchronization signal to the four inputs of the device.

If you want to use the Remote Lab you have to use the host `ti40`. The board located at this host is already connected to an oscilloscope that can be controlled using the `rpa_gui.py` tool. Channels 1 to 3 of this oscilloscope are connected to the R, G and B channels of the VGA signal, while channel 4 is connected to the horizontal synchronization signal.

Make a measurement showing one of the visible lines that goes through the gradient area of the test image. Use markers to measure the length of the whole line as well as the length of the synchronization pulse. Add screenshots of those measurements to your report. The oscilloscope in the TILab is able to write to USB flash drives.

System Integration: When your core is working, replace the precompiled_vga_gfx_cntrl in the top-level entity with your own version. Note, that if you want to change the contents of the BB ROM, you can use the script `create_bb_rom_pkg.py` located in `vga_gfx_cntrl/tools/`. Furthermore, we provide the `convert_palette.py` script, that prints the color of each pixel of the image file passed to it, in the RGB 5-6-5 format used by the graphics controller.

Task 2: Tetris Game [50 Points]

For Exercise I we supplied you with an architecture for the tetris_game module (`tetris_game_ex1.vhd`), which only implemented very basic functions, like initializing the vga_gfx_cntrl and moving a tetromino using controller input within certain bounds.

In this task you will implement your own tetris_game architecture which shall provide the actual game logic. The architecture must be called ex2 and must be placed in the file `tetris_game/src/tetris_game_ex2.vhd`.

Figure 3.6 shows a screenshot of how the final game shall look like and what information it shall display. 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
Up	Rotate the current tetromino by 90 degrees (clockwise)
Left	Move the current tetromino on the map one block to the left.
Right	Move the current tetromino on the map one block to the right.
Down	Move the current tetromino on the map down one block.
A	Move the current tetromino downwards until it collides with something.
Start	Restart the game when the previous game is over.

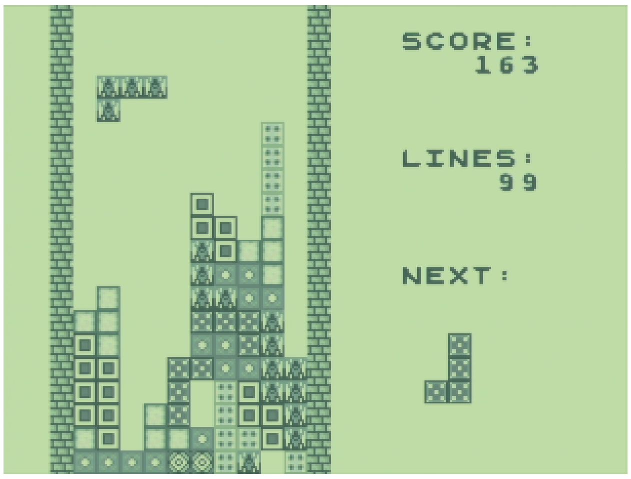


Figure 3.6: Tetris screenshot

Your implementation of the game shall behave similarly and use the exact same control scheme as the reference solution. You can choose an appropriate speed for the game (i.e., the speed with which the tetrominoes fall down the map). Note that, unlike the "real" Tetris game our version does not get faster as the game progresses.

Collision Detection: To assist you with the implementation of the collision detection between the currently "falling" tetromino and the blocks on the map as well as the walls, we provide you with the tetromino_collider. The `ex1` architecture for the tetris_game module also already contains an example on how to use it. See the IP Cores Manual for further details on this core.

When you detect a collision between the current tetromino and the map, you will have to convert the current tetromino to solid blocks on the map (i.e., set certain locations in the on-chip memory that represents the map to a value that indicates that the respective position is solid). Here the function `is_tetromino_solid_at` provided by the tetris_util_pkg will be useful. Whenever a tetromino is placed on the map, you need to check if lines are complete and remove them if necessary. Placing a tetromino on the map can lead to the removal of up to four rows.

Scoring: The games keeps track of two values: the total number of lines removed during a game and the score of the player. Both of these values start at zero.

The score is increased depending on how many rows are removed simultaneously.

rows removed	score
1	1
2	3
3	7
4	10

Tetromino Generation: Make sure that upon reset of the tetris_game module the exact same sequence of tetrominoes appears as in the reference solution. The reference solution always starts with a Z-type tetromino followed by a I-type one. Whenever a new (random) tetromino is needed in the `ex1` architecture of the tetris_game the prng_en is asserted for exactly one clock cycle. You can directly use this implementation as it will generate the required sequence of tetrominoes.

New tetrominoes must always be placed in the center of the upper end of the map. If placing a new tetromino immediately leads to a collision with map blocks, the game is over. To signify this condition to the player, the newly added tetromino (that caused the collision) should blink. Pressing the start button on the controller starts a new game, all other input (i.e., arrow keys and the A button) is ignored.

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

- The player completed a row (and it is removed)
  Play a single tone for ≈ 0.25 s with a frequency that depends on how many rows have been removed simultaneously. The more rows have been removed the higher the frequency.
- The game is over
  Play two successive single tones (≈ 0.25 s each) with increasing frequency

The controller rumble should be activated in both of these cases for ≈ 0.5 s.

Architecture: We recommend that you use the `ex1` architecture of the tetris_game module as a basis for your implementation. Moreover, you may use the tetrmino_collider and the tetrmino_drawer modules as is, but you don't have to. Feel free to modify them, change their 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 tetris_game module (similar to what is already done with the decimal_printer module in the arch_ex1 architecture). If you do so, put all additional entities in the `tetris_game/src/` directory.

Suggested implementation sequence:

- Start by implementing the rendering of the map and its borders.
- Add functionality that converts the current tetromino to solid blocks on the map.
- Add detection for complete lines and implement their removal from the map.
- Implement the automatic downwards movement.
- Add the score/lines display (use the decimal_printer for that) and the sound effects. Note that the tetromino_drawer has the same interface protocol as the decimal_printer implemented in Exercise I.

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 tetris_game module:

- gfx_instr
- gfx_instr_wr
- gfx_instr_full

Trigger on the first instruction that is issued to the graphics controller for some frame (probably a `CLEAR` 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 (command and operands) issued to the graphics controller. Make sure that the value of the vector signal gfx_instr is 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 template we provided you with.

```
cd path/to/ddca_ss2022/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_cntrl
    ├── tetris_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**
2.3	20.04.2022	FH	Added logic levels to Figure 3.3
2.2	14.04.2022	FH	Fixed signal name in the Internal Interface section of the vga_gfx_cntrl, improved wording of the tetromino collision detection section
2.1	07.04.2022	FH	Fixed paths in the submission instruction sections
2.0	07.04.2022	FH	Added Exercise II assignment
1.3	22.03.2022	FH	Figure 2.9: fixed data assignment in POLL state, changed successor state of POLL from WAIT_SAMPLE to WAIT_FALLING_EDGE_DATA.
1.2	19.03.2022	FH	Fixed generic name in GameCube Controller ⇒ System Integration section.
1.1	08.03.2022	FH	Fixed inconsistent statements about the bit order of the polling command of the GameCube controller (Figure 2.9).
1.0	06.03.2022	FH	Initial version

Author Abbreviations:

FH	Florian Huemer
FK	Florian Kriebel