Digital Design and Computer Architecture LU

Lab Exercises I and II

Florian Huemer, Florian Kriebel, Jürgen Maier
{fhuemer, fkriebel, jmaier}@ecs.tuwien.ac.at
Department of Computer Engineering
University of Technology Vienna

Vienna, March 1, 2018

1 Introduction

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

- Exercise I: 23.03.2018, 23:55
- Exercise II: 20.04.2018, 23:55

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 improve the course.

Please note that this document is not the complete assignment. View the protocol template for all required measurements, screenshots and questions to be answered. Make sure that all required details can be seen on the screenshots you put into your report, otherwise these screenshots will be graded with zero points.

1.1 Coding Style

We highly recommend to implement state machines with the 2 or 3-process method discussed in the lecture. To avoid hard-to-find bugs we also recommend to use the "named association" method for creating instances. Please keep the entity and the associated architecture in the same file, this makes it easier for use to check and grade your solutions.

1.2 Software

As discussed in more detail in the Design Flow Tutorial, we are using Quartus and 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 ModelSim (ModelSim-Altera) from the Altera Website.[1] However, note that the simulation performance with ModelSim-Altera might be reduced, when compared to the full version of ModelSim provided in the lab.

To connect to the serial port on the lab computers use GTK term (graphical interface) or minicom (command line tool).

1.3 Allowed Warnings

Although your design might be correct, Quartus still outputs some warnings during the compilation process. Table 1.1 lists all warnings that your design is allowed to contain. There is no need in trying to fix these warnings from your side, since they won't have any 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 the assignment.

The last two warnings in Table**??** 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 and Quartus reports that it does not drive any logic, then there is certainly a problem. On the other hand, if you intentionally drive some output with a certain constant logical level (for example an unused seven segment display), then the "stuck at VCC or GND" warning is fine.

[1] `https://www.altera.com/products/design-software/fpga-design/quartus-prime/download.html`

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.`
276020	`Inferred RAM node [...] from synchronous design logic. Pass-through logic has been added to match the read-during-write behavior of the original design.`
15064	`PLL [...]|altpll:altpll_component|pll_altpll:auto_generated|pll1" output port clk[0] feeds output pin "nclk~output" via non-dedicated routing -- jitter performance depends on switching rate of other design elements. Use PLL dedicated clock outputs to ensure jitter performance`
169177	`[...] pins must meet Intel FPGA requirements for 3.3-, 3.0-, and 2.5-V interfaces. For more information, refer to AN 447: Interfacing Cyclone IV E Devices with 3.3/3.0/2.5-V LVTTL/LVCMOS I/O Systems.`
171167	`Found invalid Fitter assignments. See the Ignored Assignments panel in the Fitter Compilation Report for more information.`
15705	`Ignored locations or region assignments to the following nodes`
15714	`Some pins have incomplete I/O assignments. Refer to the I/O Assignment Warnings report for details`
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: 23.03.2018)

2.1 Overview

In first lab exercise you will make yourself acquainted with the tools used in this lab and design your first FPGA system with VHDL. A basic FPGA design flow consists of a simulator and tools for synthesis and place & route. The simulator is used for verifying and debugging the functionality and the timing of the circuits. The synthesis tools translate the behavioral and/or structural description into a gate-level netlist. This netlist can then be mapped to the FPGA's logic cells. Finally the produced bitstream file is used to configure the FPGA.

2.2 Required Reading

- Design flow tutorial
- VHDL introduction slides (Hardware Modeling)
- IP core documentation
- Logic Analyzer Resources

2.3 Task Descriptions

Task 1: Structural Modeling [2 Point(s)] Your task is to implement a top-level structural VHDL description of the system shown in Figures 2.1 and 2.2. The description must be done in VHDL and should contain only structural primitives (component instantiations and concurrent signal assignments). All information needed to wire the IP cores together is contained in the figures.

Create a new qrtus project (in `top/quartus/`) and add all needed IP cores and your top-level description to it. We already provide you with an empty entity description (`top/src/top.vhd`). The constants used for the generics in the figures are defined in Table 2.1. Make sure you use constants in VHDL. Do not set the values directly in the generic map.

Constant	Value
SYS_CLK_FREQ	50000000
SYNC_STAGES	2
PS2_BUFFER_DEPTH	8

Table 2.1: Constants

The PLL displayed in the figure is not supplied. You need to generate it using the corresponding wizard within Quartus (see the Design Flow Tutorial for further information). The frequency of the display clock (PLL output) should be 25 MHz.

You don't have to take care 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. The unused inputs (e.g. keys(3 downto 1) or switches) may stay unconnected (i.e. you can ignore the corresponding warning generated by Quartus). The hex{0-7} outputs should be set to constant '1' for now. Unused (i.e. unconnected) outputs of instances (e.g. current_color) should be marked with the "open" keyword.

When you have completely assembled the design, you should have a system that reads characters over the PS/2 interface from the keyboard, translates them into instructions for the graphics controller and executes these instructions. The scancodes received from the PS/2 keyboard are displayed on the green LEDs of the board. For a first check type in the following character sequences on the keyboard.

```
x
cffff
l00000031f1df
```

These commands should clear the screen and draw a white line form the top left to the lower right corner. A complete list of supported input commands can be found in the IP Cores Manual.

For your report include a screenshot of the overall system from RTL netlist viewer in Quartus.

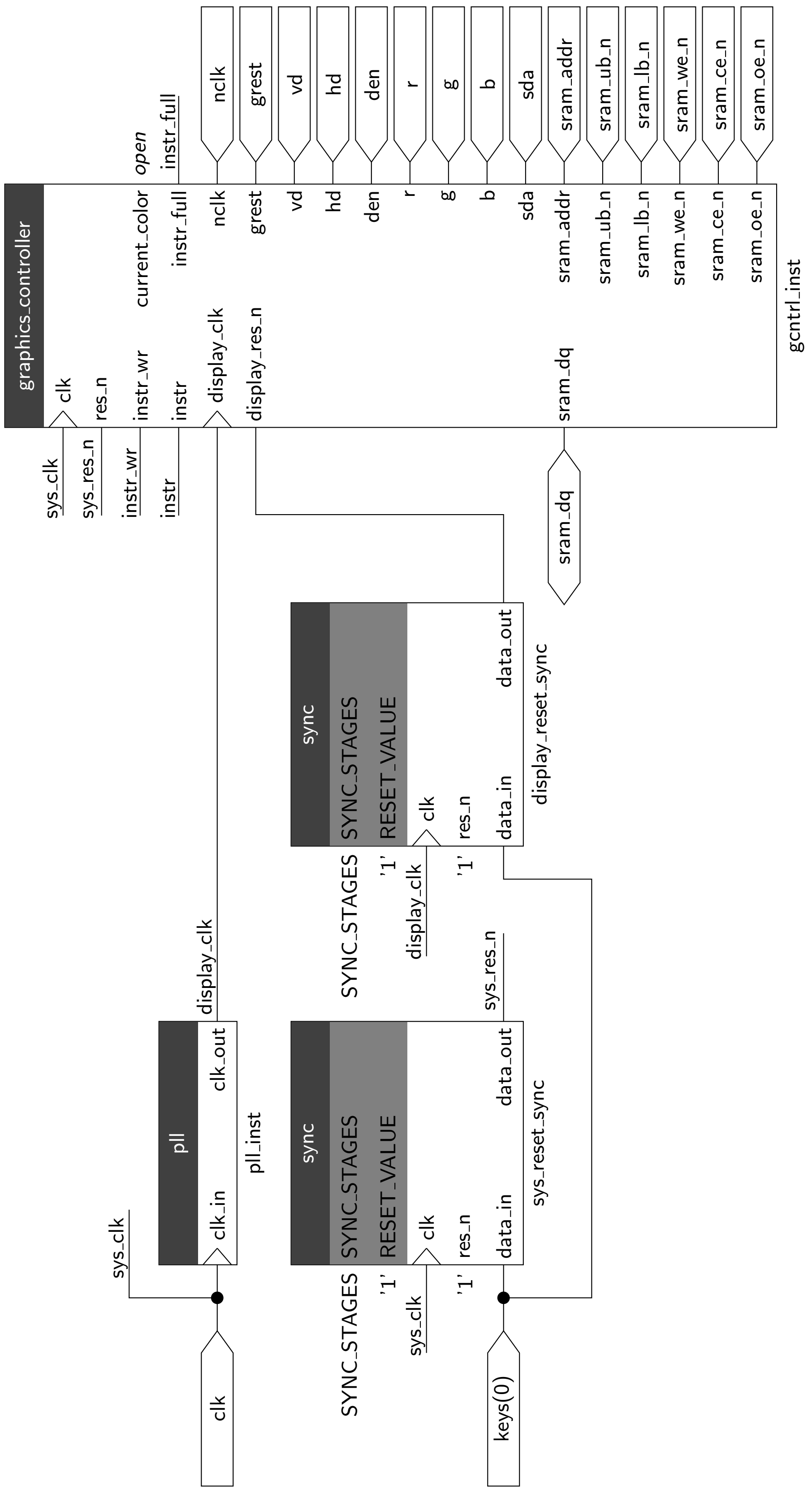

Figure 2.1: Structural system description (part 1)

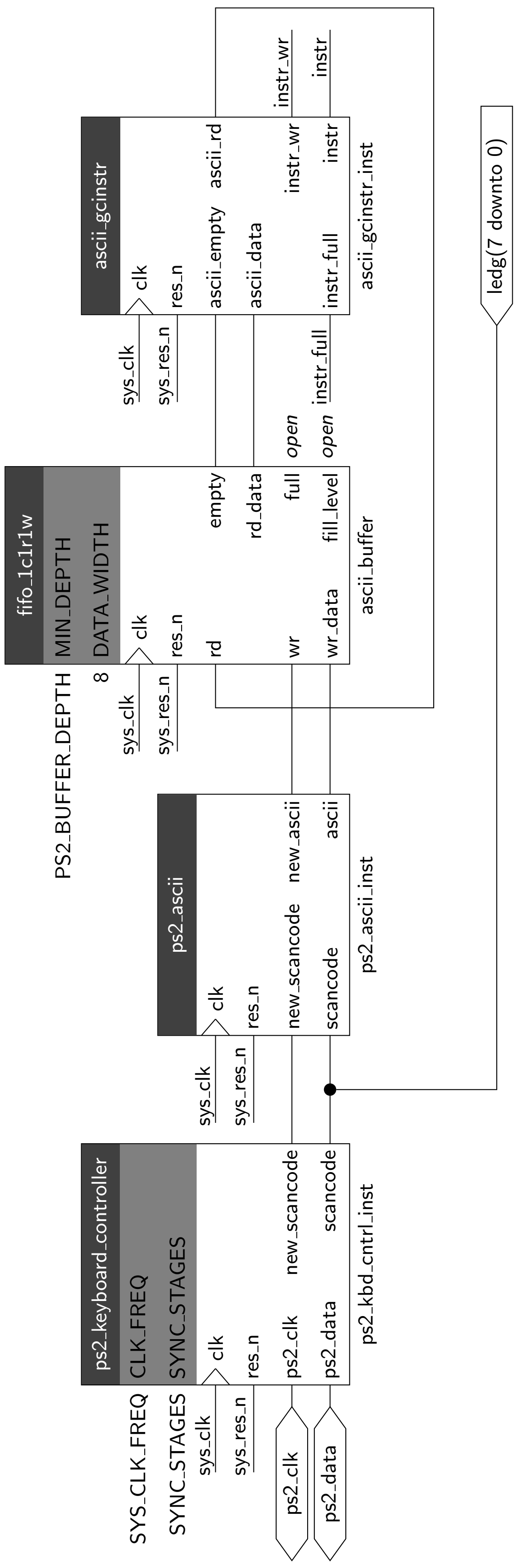

Figure 2.2: Structural system description (part 2)

Task 2: Seven Segment Display I [1 Point(s)] In this task you will extend our design with a simple combinational module that outputs the current drawing color that is used by the graphics controller on the seven segment display of the board. For this purpose create a new entity called seven_segment_display and place it in the `seven_segment_display/src` directory. Table 2.2 specifies the interface of this entity.

Name	Dir.	Width	Functionality
color	in	16	The 16 bit RGB color value
hex0	out	7	The bits[3:0] of the color input as hex digit
hex1	out	7	The bits[7:4] of the color input as hex digit
hex2	out	7	The bits[11:8] of the color input as hex digit
hex3	out	7	The bits[15:12] of the color input as hex digit

Table 2.2: seven_segment_display interface specification

Figure 2.3 shows the pattern that should be used for the number 0-9 and A-F.

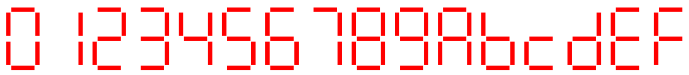

Figure 2.3: Seven segment hexadecimal number patterns

Consult the FPGA board manual for more information on how to control the individual segments of the seven segment display. After your module is complete, create a package for it and add an instance of the seven segment display to the top-level module. Connect the color input to the current_color output of the graphics controller and the hex{0-3} outputs to the corresponding outputs of the top level entity. The correct pin assignment should have already been configured in Task 1.

If you now execute a color change command (e.g. `c1234`) the seven segment display should reflect your input.

Task 3: Behavioral Simulation [2 Point(s)] Perform a behavioral simulation of the system. An appropriate testbench is provided in the `top/tb` directory. This testbench generates keyboard input and simulates the input `c1234x`, i.e. a color change command followed by a clear screen command.

Add all signals of the top-level entity to the waveform window and run the simulation long enough to trace the propagation of the color change command (`c1234`, i.e. the first five input characters over the PS/2 interface) through to the color showing up on the output of the *seven_segment_display*. Take screenshots showing the propagation of the last input character of the color change command (i.e. '4") through the system including the following (internal) signals:

- The outputs new_scancode and scancode of the *ps2_keyboard_controller* instance
- The outputs new_ascii and ascii of the *ps2_ascii* instance
- The outputs ascii_rd, instr_wr and instr and the inputs ascii_empty and instr_full of the the *ascii_gcinstr* instance
- The current_color output of the *graphics_controller* instance

Hence, show how the input scancode is converted to an ASCII character and finally to an instruction for the *graphics_controller*. The start of the time frame shown in the your simulation screenshot should be around the time when the scancode for the last character of the change color command (i.e. the scancode for the '4' key) is generated by the *ps2_keyboard_controller*. Furthermore, measure the time intervals requested in the protocol template with the help of markers. Take a screenshot showing the marker pair for one of the measurements and include it in your protocol.

In a second step simulate the complete execution of the clear screen command (i.e. the time the *graphics_controller* takes to execute the command). How can you determine when the command has finished execution by just observing the signals to the SRAM?

Lastly, use the testbenches provided for the *ps2_ascii* module to check it for bugs. If you find one, fix it and describe how it affects the design. The testbenches as well as a README file, explaining how to use them, are located in the *ps2_ascii* directory.

Task 4: Postlayout Simulation [1 Point(s)] Use the netlist file (.vho) and the timing file (.sdo), which were generated during Task 2, for performing a post-layout simulation [2]. The testbench file used in the behavioral simulation can also be employed for post-layout simulation.

The timing file provides information on the real physical signal delays. Therefore, signals do not switch instantaneously after the clock edge, in contrast to a behavioral simulation. Every single bit of a signal vector switches individually depending on the propagation and routing delays of the corresponding circuitry. Run the simulation long enough in order to take a screenshot of the switching of hex{0-3} after the execution of the color change command. Zoom into the waveform until you can see the different delays of the signals and use two markers to measure the duration between the first and the last bit toggling.

Task 5: Seven Segment Display II [1 Point(s)] In this task you will extend the functionality of the *seven_segment_display* in the following way. Use two additional seven segment displays (hex4 and hex5) to also output the individual color channel components of the 16-bit color input. Initially the value for the red color channel should be shown. If an input button is pressed the display should switch from red to blue, from blue to green and from green back to red again.

To implement this behavior add the signals listed in Table 2.3 to your entity.

Name	Dir.	Width	Functionality
clk	in	1	The global clock input
res_n	in	1	The global reset input
button	in	1	The button input, that should trigger the change of the displayed color channel
hex4	out	7	The lower 4 bits of the currently selected color channel
hex5	out	7	The upper 4 bits of the currently selected color channel

Table 2.3: seven_segment_display interface specification (extension)

You should design a state machine with three states. Pressing the button should switch its state to the next one, according to the scheme described above.

Implement a testbench for your design (and place it in the `seven_segment_display/tb` directory) and create a simulation screenshot showing the state machine switching through all its states (i.e. simulate three button presses in your testbench). This simulation should contain all inputs and outputs of the *seven_segment_display* core, as well as the internal state variable. Use your matriculation number modulo 2^{16} as input value for the color port and argue why your output on the signals hex{4-5} is correct.

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

Note that after finishing this task your *seven_segment_display* core should still provide the output value at hex{0-3}, as implemented in Task 2.

On the top-level entity use keys(1) as input button. However, keep in mind that you need a synchronizer for this button. Hence don't directly connect keys(1) to the button input of the *seven_segment_display*, but rather use another instance of the synchronizer component.

Task 6: Serial Port Interface [3 Point(s)] In this task you will implement a serial port interface and integrate it in the top-level design. This should enable your design to process commands received from a PC. The generics and the port signals of the serial port interface are described in Table 2.4 and 2.5, respectively.

Name	Functionality
CLK_FREQ	Actual clock frequency of the *clk* signal given in Hz
BAUD_RATE	The baud rate used for the serial port
SYNC_STAGES	Number of stages used in the input synchronizer
TX_FIFO_DEPTH	Number of elements which can be stored within the transmitter FIFO
RX_FIFO_DEPTH	Number of elements which can be stored within the receiver FIFO

Table 2.4: Serial port generics description

Name	Dir.	Width	Functionality
clk	in	1	Global clock signal
res_n	in	1	Global reset signal (low active, not internally synchronized)
tx_data	in	8	Data which should be transmitted to the host
tx_wr	in	1	If 1, a new data byte is present on tx_data and should be copied to the internal buffer.
tx_full	out	1	If 1, the internal buffer is full and a new data byte may not be written into the buffer.
rx_data	out	8	The byte which was last read from the receiver FIFO.
rx_rd	in	1	If 1, the next byte is read from the receiver FIFO if it is available. The byte is available at the rx_data port. If no byte is available, the value of the rx_data port is undefined.
rx_full	out	1	If 1, the receiver FIFO buffer is full and characters may have been lost because they could not be stored in the buffer.
rx_empty	out	1	If 1, the receiver FIFO is empty, otherwise received bytes are available in the buffer.
rx	in	1	The receive signal of the UART (Host → Device).
tx	out	1	The transmit signal of the UART (Device → Host).

Table 2.5: Serial port signal description.

The interface protocol of the serial port is the same as for the FIFO buffers described in the IP core documentation. The transmitter port uses the write port of the transmit FIFO, while the receiver port uses the read port of the receive FIFO.

The tx and rx signals use the standard UART protocol with 8 data bits, 1 stop bit and no parity. The used baud rate is configured using the corresponding generics.

The serial port consists of five components, namely a synchronizer, two FIFO buffers and the two state machines which implement the receiver and the transmitter. Figure 2.4 shows how these components are connected together and how the input and output signals are assigned.

We already provide you with a working implementation of the transmitter state machine (*serial_port_tx_fsm*). Hence your task is to implement the receiver state machine (*serial_port_rx_fsm*)

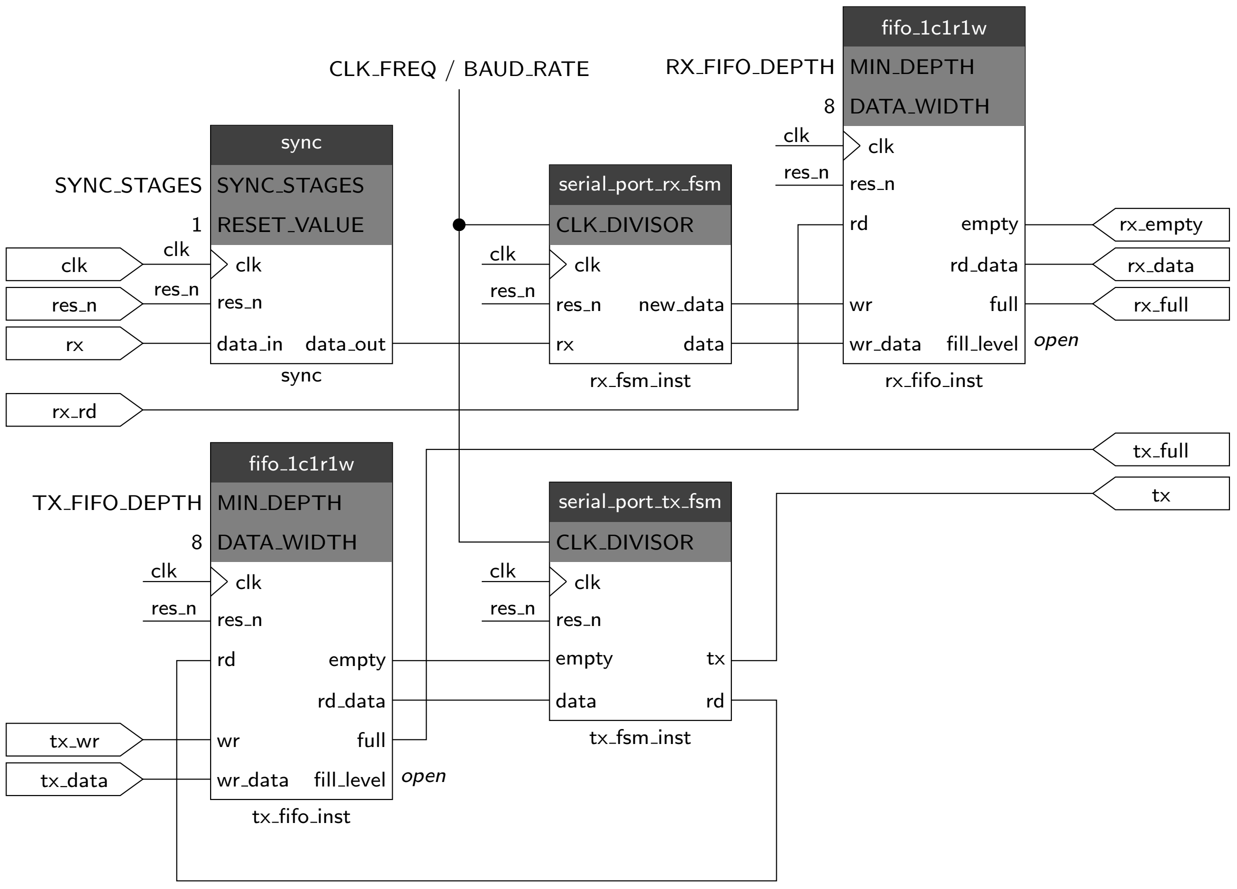

Figure 2.4: Serial port internal circuitry

Signal	Default Assignment
clk_cnt	keeps its data
bit_cnt	keeps its data
data_int	keeps its data
data_new	0
data_out	keeps its data

Table 2.6: Default assignments for the receiver FSM signals

and to do the overall structural modeling of the serial port (i.e. tie everything together). All source files of the serial port must be placed in the `serial_port/src` directory.

The operation of the receiver FSM is described by its state chart (see Figure 2.5). The arc labels show the conditions that must be fulfilled for a state transition. Every state also contains a number of (signal) assignments that must take place when the state machine is in a particular state. It can hence be seen that your state machine will need registers to keep track of the values for clk_cnt, bit_cnt and data_int. The default assignments for these signals (i.e. the values that should be assigned to these variables when there is no explicit assignment in the state chart) are shown in Table 2.6.

Start by implementing the receiver state machine and create behavioral simulations to validate the correctness of your implementation.

Afterwards create a structural VHDL design which connects all components (synchronizer, FIFO buffers and the state machines) corresponding to Figure 2.4. Create a testbench

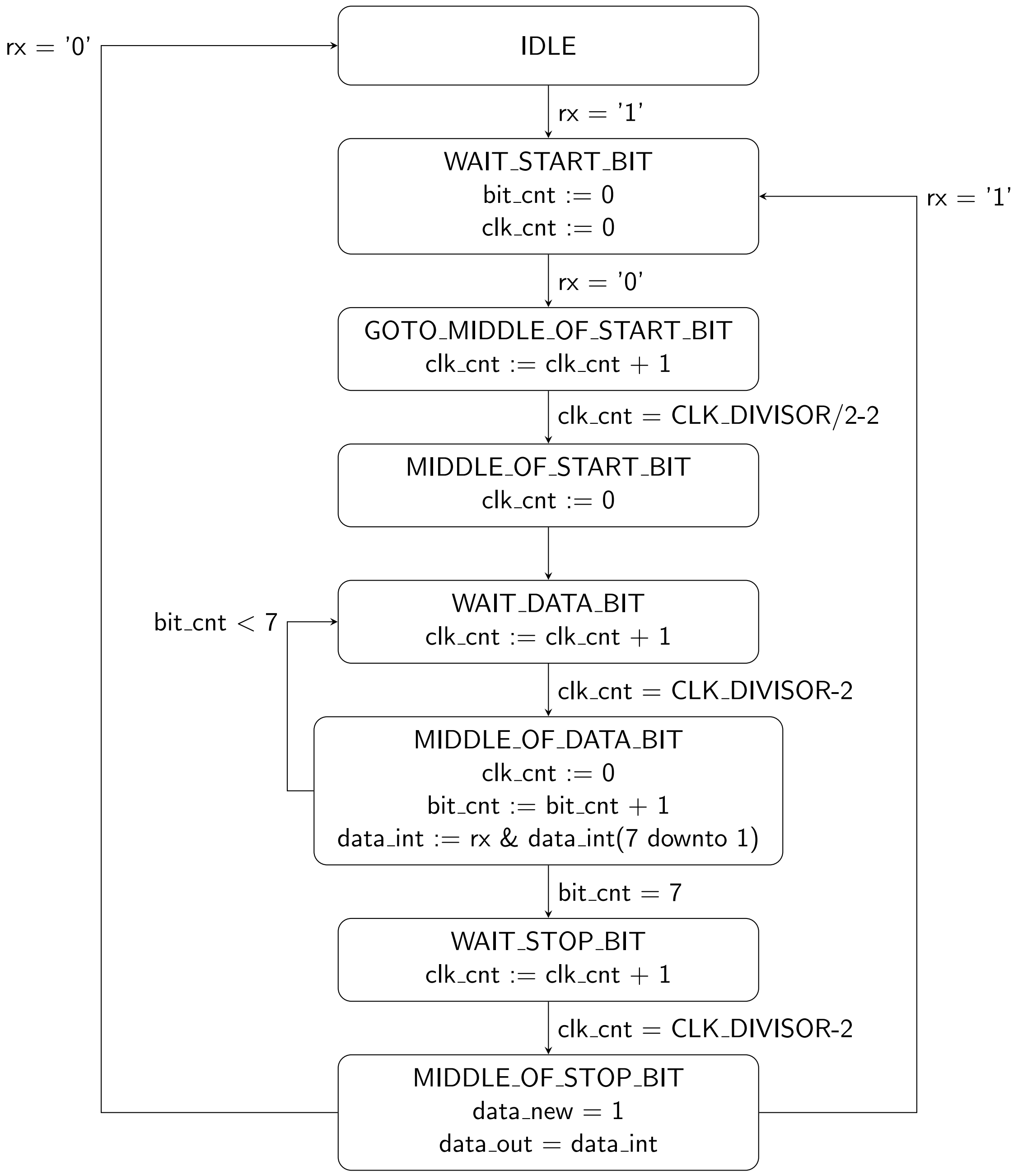

Figure 2.5: Serial port receiver state machine

(`serial_port/tb/serial_port_tb.vhd`) and run a behavioral simulation to validate its functionality. For the protocol, take a screenshot of a simulation showing the reception of a whole UART frame and include markers measuring the duration of the frame including the start-bit. Include the states of the receiver and transmitter FSMs and all signals connecting the components shown in Figure 2.4.

To demonstrate the correct functionality of your serial port design when really implemented in hardware, add the serial port component to your implementation of the previous task. For the rx and tx signals you have to create new ports in your top level entity and map them to the corresponding FPGA pins. It should be possible to use the serial port receiver interface as a second form of input for drawing commands. Hence you need to create a second instance of the ASCII to Graphics Controller Instructions Converter (*ascii_gcinstr*) and connect it to the signals rx_rd, rx_data and rx_empty of the serial port (rx_full can be left open). In order for the graphics controller to process instructions from two sources (i.e. the serial port and the PS/2 keyboard) use the *merge_fifo* IP core (see IP Cores Manual).

The UART transmitter port (i.e. the signals tx_wr and tx_data) should in turn be connected to the output of the PS/2 to ASCII converter (tx_full can be left open). Hence, every input over the PS/2 keyboard should also be transmitted over the serial port.

After compilation and testing, find the resource usage of the serial module including all submodules and add it to the protocol.

Task 7: Logic Analyzer Measurement [2 Point(s)] In the VHDL system description, add copies of the signals hd, vd, r(7), g(7), b(7) and tx to the top level entity and map them to appropriate pins on the FPGA board's extension connector (GPIO). Connect the logic analyzer to the chosen pins and trace the signals and execute the following tasks:

- Measure the length of a whole UART frame including the start bit sent from the FPGA to the computer, as well as the bit time. Include markers for measuring these duration in the screenshot. Transmit an appropriate character such that you can perform this measurement.
- Measure the hsync to hsync and vsync to vsync intervals. The first interval is the time it takes to transmit one horizontal line of pixels to the display, while the second one basically yields the refresh rate of the display. Include a screenshot of the first measurement, where the markers can be seen. Include the hd, vd, r(7), g(7), b(7) signals and markers for measuring the time interval in the screenshot.
- Implement an advanced trigger for the tx signal, that is only activated if the first data bit (i.e. the first bit *after* the start bit) of the transmitted UART frame is 1. Hence the logic analyzer should only start recording data if ASCII symbols are transmitted where the LSB is set to one. This includes ASCII characters like 'a', 'c', 'e', 'g', 'i' '1', '3', '5', etc. but excludes characters like 'b', 'd', 'f', '0', '2', '4'. Include a screenshot of the trigger settings in the protocol and include the trigger configuration file in your submission.

2.4 Submission

Upload a ZIP or TAR.GZ file containing the following information to TUWEL:

- Your lab protocol as PDF
- The source code of **all** IP cores
- The source code of the PLL
- The SDC file containing the clock definition
- The source code of your top-level module
- The source code and testbenches of your IP cores
- Your Quartus project (don't forget a cleanup!) or a TCL script creating the project (including the pin mappings!)
- Your Modelsim projects (don't forget a cleanup!) or the scripts used for simulation
- The advanced trigger configuration file

Make sure the submitted Quartus project is compilable. All submissions which can not be compiled will be graded with zero points! The submitted archive should have the following structure.

```
submission.tar.gz
 ├─ report.pdf ............................................ Include your lab report here
 ├─ advanced_trigger.xml .......... The configuration file for your advanced trigger (Task 7)
 └─ vhdl ............................................... The source code of all IP cores
     ├─ top
     ├─ serial_port
     ├─ seven_segment_display
     └─ [...  all other IP cores]
```

3 Exercise II (Deadline: 20.04.2018)

3.1 Overview

In this exercise you have to design your own state machines for (a) controlling the touch screen which is available on the display extension board and (b) for interfacing with the graphics controller. Designing decent state machines is a key issue for many hardware circuits. Furthermore, you will use the SignalTap Logic Analyzer to make some measurements on your design. This tool can be very helpful, while debugging your code.

Read the **whole** assignment before you start working on your solution.

3.2 Required Reading

- Design flow tutorial
- IP cores documentation
- DE2-115 manual
- LCD module manuals
- AD7843 data sheet
- SignalTap documentation

3.3 Task Description

Task 1: External Interface to AD7843 [5 Point(s)] The first task is to design and implement a state machine that interfaces with the AD7843 Touch Screen Digitizer, with which it is possible to read out the exact position on the screen that has been touched. Your entity should be named *touch_controller* and be placed in the `touch_controller/src/touch_controller.vhd` file.

The (SPI) interface to the AD7843 consists of the signals shown in Table 3.1. Further details on the meaning and behavior of these signals can be found in the data sheet.

Signal	FPGA pin	Description
ADC_PENIRQ_n	PIN_Y28	ADC pen Interrupt
ADC_DOUT	PIN_T22	ADC serial interface data out
ADC_BUSY	PIN_Y27	ADC serial interface busy
ADC_DIN	PIN_T21	ADC serial interface data in
ADC_DCLK	PIN_V22	ADC serial interface clock
SCEN	PIN_J25	ADC chip enable

Table 3.1: Touch Screen Digitizer Interface

The ADC used by the device can be configured in different modes. For our exercise we use the following settings:

- 12 bit conversion mode (MODE bit)
- Use $+REF = V_{REF}$ and $-REF = GND$ (Channel Addressing Bits A2-A0, SER/$\overline{\text{DFR}}$ Bit)

- Obviously we need to read out the X and Y channels (Channel Addressing Bits A2-A0)

Design an appropriate interface (protocol) to forward the read coordinates to the next component and describe it in the lab report (which signals are used, what is their meaning). Implement a testbench to test your *touch_controller*.

Task 2: Internal Interface to the graphics controller [5 Point(s)] In this task you will design a state machine that uses the output of the *touch_controller* implemented in the previous task to generate instructions for the graphics controller. This component is named *input_manager* and must be placed in the `input_manager/src/input_manager.vhd` file. Table 3.2 specifies the port signals of the Input Manager.

Name	Dir.	Width	Functionality
clk	in	1	Global clock signal
res_n	in	1	Global reset signal (Low active)
switches	in	16	Input switches to select the drawing color
change_color_button	in	1	Button input that should trigger a color change
change_mode_button	in	1	Button input that should trigger a drawing mode change
hex	7	out	Output of the current operation mode for the seven segment display (you may also use two displays if you want)
instr_wr	1	out	Instruction write signal of the graphics controller
instr	out	GCNTRL_INSTR_WIDTH	The actual graphics controller instruction
instr_full	1	in	The full signal of the graphics controller, if '1' no new instructions can be processed
...	...	...	Your custom interface to the Touch Controller

Table 3.2: Input Manager Interface Specification

If the button connected to change_color_button is pressed, your state machine should generate a change color instruction. The argument of this instruction is taken from the switches.

If the button connected to change_mode_button is pressed, you should change your drawing (i.e. operation) mode. You have to support the following operation modes.

- Free-Hand Mode:
  Read a single coordinate from the touch controller and set one pixel accordingly.
- Draw Line Mode:
  Draw a line between two points on the screen. The first point should be the first coordinate that is read by your *touch_controller*, immediately after the screen was touched for the first time. The second point should be the last coordinate that is read by the touch controller before the touch screen was released.
- Draw Rectangle Mode:
  Draw a rectangle defined by the two endpoints of its diagonal. The input behavior should be exactly the same as for the Draw Line Mode.
- Draw Circle Mode:
  Draw a circle with its center at the first position read by the touch controller after the screen

was pressed. The radius should be given by the absolute x distance to the coordinate where the screen was released again.

- Fill Rectangle Mode:
  Fill a rectangle defined by two endpoints of its diagonal (same input behavior as for the Draw Rectangle Mode).

Note that most of the commands can be directly translated to a graphics controller instruction. However, for the Fill Rectangle Mode, there is no single instruction available, that performs this operation. Hence you have to find some way to implement the required behavior by executing multiple instructions of the graphics controller.

The currently selected operation mode shall be reflected by a seven segment display via the hex output. You may freely choose what number, letter or pattern you want to assign to the different modes.

For the sake of simplicity you may ignore change_mode_button and change_color_button changes while your state machine is not in its idle state. Create a testbench that simulates the Fill Rectangle Mode and take a screenshot of the simulation output (also show the internal signals of your state machine). Use a small rectangle (at least 3x3), such that the execution of the whole command fits into the screenshot.

Note that at some point in your design you will have to convert the 12-bit values you get from the touch controller to a value range that reflects the resolution of the display (i.e. [0;799] and [0;479]). Don't use a division for that purpose! A division by a constant can easily be implemented by a few additions.[3] It may also be the case that there is a static offset in the ADC values. This value can be subtracted before scaling. Note that your solution doesn't have to work perfectly on every board. However, outline how you did the conversion in the lab report.

Listing 1 shows an example implementation of an approximate division function implemented in Python. Although it is not perfect and even yields wrong results for some values (e.g. 4095) it has sufficient precision for our purpose to convert the x coordinate of the ADC to screen coordinates. Note, however that here we did not take a static offset into account.

```python
def approx_div(x):
  # binary representation of 799/4095 = 0.001100011111...
  x = x << 4
  x =  (x>>3) + (x>>4) + (x>>8) + (x>>9) + (x>>10) + (x>>11) + (x>>12)
  return x >> 4
```

Listing 1: Approximate Division Example

Task 3: Integration [1 Point(s)] Add the new input manager and the touch controller to the top-level design of the project of Lab Exercise I. Connect the instr_wr, the instr and the instr_full inputs to the free port of the merge FIFO, which should already be present in the design. Don't forget to use synchronizes for all required input signals (for the switches the synchronizes may be omitted)! Synthesize the whole design and download it onto the FPGA board for testing.

Task 4: SignalTap Measurement [2 Point(s)] Use a SignalTap II Logic Analyzer to analyze the behavior of your design during run-time. For this purpose trace the followings signals of the input manager

- instr_wr
- instr_full

[3] see http://www.hackersdelight.org/divcMore.pdf for details

- instr.

and issue a **fill** rectangle command to your design (using the touchscreen). Trigger on a change of the instr_wr command. The measurement screenshot should contain at least one transition on instr_full.

Task 5: Draw Triangle Mode (Bonus) [2 Point(s)] For the bonus task you should implement an additional operation mode for the input manager, namely the "Draw Triangle Mode". This mode also reads two input points P_0 and P_1 in the same way as the "Draw Line/Rectangle Mode". As shown in Figure 3.1, these two points should form one side of an isosceles right-angled triangle. Hence your task is to calculate the third point/coordinate P_2 and generate appropriate instructions for the graphics controller. Beware however that the third point may lie "outside" of the display area. In such a case the input should be ignored and nothing should be drawn.

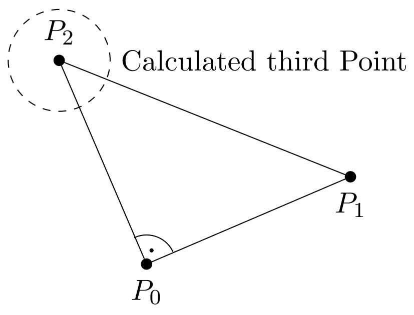

Figure 3.1: Triangle

3.4 Submission

Upload a ZIP or TAR.GZ file containing the following information to TUWEL:

- Your lab protocol as PDF
- The source code of **all** IP cores
- The source code of the PLL
- The SDC file containing the clock definition
- The source code of your top-level module
- The source code and testbenches of your IP cores
- Your Quartus project (don't forget a cleanup!) or a TCL script creating the project (including the pin mappings!)
- Your Modelsim projects (don't forget a cleanup!) or the scripts used for simulation

Make sure the submitted Quartus project is compilable. All submissions which can not be compiled will be graded with zero points! The submitted archive should have the following structure.

```
submission.tar.gz
└─ report.pdf ........................................ Include your lab report here
```

```
|
└── vhdl ........................................ The source code of all IP cores
    ├── top
    ├── serial_port
    ├── seven_segment_display
    ├── touch_controller
    ├── input_manager
    └── [... all other IP cores]
```