Project Report

Faculty of Computer Science

Master Applied Computer Science

FPGA Programming

SS2021

Implementation of Car Wash Lane with VHDL

Contributors:

Ann Thomas (816175)

Florian Strohhammer (498623)

Furkan Ariöz (821968)

Manuel Gabler (452352)

Said Anshasi (815280)

Lecturer:

Prof. Dr. Martin Schramm

Submission Date

30.07.2021

Clarification of the independent work

Hereby, we assure that everyone in the team has put roughly the same amount of effort and resources into the project. Furthermore, we assure that we have not copied any works without mentioning them and that this work has not yet been submitted to any other examination.

Team members:	Ann Thomas
	Florian Strohhammer
	Furkan Ariöz
	Manuel Gabler
	Said Anshasi

Table of contents

1 Introduction 4
2 Design 4
3 User Interface..4
4 Implementation 7
4.1 Entities 7
e_my_car_wash_project 7
e_car_wash_FSM 8
e_modulo_counter 8
e_axis 8
e_option_selection 9
e_price_calculation 9
e_RAM 10
e_display_and_LED_handler 10
e_convert_to_7seg 10
4.2 Finite State Machine...11
5 Timing of the circuit..13
6 Simulation /Testbench 15
7 Conclusion 17
8 References 18

1 Introduction

The objective of the project includes the conceptual design, implementation, and verification of functionality through simulation of a VHDL based digital circuit as a team of 5 members. Our idea was to develop an automatic car wash where the end user will be given the privilege to select from different options like rim cleaning, waxing, polishing, and much more. The price will be calculated and shown on displays. Once the end user pays, the process starts automatically. This selection is stored into the RAM memory so that the user is not able to add some options for which he has not paid for during the already running process. In case of any errors or serious faulty situations an emergency stop is implemented. The system can also be reset and get back to the initial selection/state. The state machine is executed based on the selections. In addition, the current state is shown on displays and LEDs.

2 Design

During the design of the project, focus was given on the requirements and functionalities were added to fulfil the same.

The modules were mainly divided like implementation of state machine, selection, display, price calculation, memory, counter. Each module is later converted to one or more sub entities. The flow of data between these modules were also defined and a top level entity need to be created. Each worked on individual units and created testbench that checks these individual units followed by the merging and testbench for the whole digital circuit.

The option to be selected by end customer with the help of switches is predefined along with prices. The general washing and drying is the basic unit and costs 4 Euro. There are 5 options that could be added to the basic unit – rim cleaning, waxing, polishing, shine wash and undercarriage wash. The price range is from 4 Euro to 14 Euro maximum with all options selected and the calculated price is shown in 7 segment displays.

3 User Interface

SW [0]	– select option for rim cleaning
SW [1]	– select option for undercarriage wash
SW [2]	– select option for waxing
SW [3]	– select option for polishing
SW [4]	– select option for shine wash

KEY [0] reset button for Finite state machine entity, press KEY [0] whenever:

- You want to stop all operations and go back to initial state “IDLE” state in which you will only be able to store your selection to the RAM and acknowledge your selection.

KEY [1] is used to acknowledge whatever the user has selected from the program selection (undercarriage washing, waxing, polishing, rim cleaning and shining) in addition to the basic washing and drying functionality, once the user has acknowledged the selection, he/she will be brought to the pricing interface (state).

KEY [2] press this button wheneve3r you want to pay the total price of the selected option plus the base price of washing and drying, once the pay button is pressed the washing operations will start. It is not possible to change the sequence of operation until all operations have been done. Only after that the user can acknowledge the new selection and pay the amount for a different sequence of operations to start.

KEY [3] press this button if you no longer want to pay the amount and wash your car, in this case you must press the acknowledge button again, had you wished to pay the amount and start the washing operation.

SW [9] toggle this switch to 0 whenever you want to perform an emergency shut down to the machine axis, by toggling the switch to 0, you will stop all operations of the machine and go back to initial state, this includes stopping the movement of the axis, the axis base/end positions flags will be zero, which denotes that the axis haven't reached either point. In this case the technician needs to drive the axis back to start position if he/she wishes to start over with a new sequence of operations.

SW [8] toggle this switch to 1 had you wished to drive the axis back to its starting position.

HEX [0-5], LED [0-9]

- Seven segment displays will show info based on the state we are currently in in the finite state machine.
- LED's lit will depend on the state we are in and the activated operation of the washing sequence.

List of operations:

- Washing jets activate
- Drying jets activate
- Shining activate
- Polishing activate
- Waxing activate
- Undercarriage washing activate
- Rims cleaning activate

State	7 segment displays HEX [0-5]	LED [0-9]
Idle	SELECT	“00000” & Selected options (waxing, shining, etc)
Price	Price tens price ones-EUR	“00000” & Stored options in the RAM
Wash starts	WASH	Axis moving flag & ”001000000”
Wash under-carriage start	WSH-UC	Axis moving flag & ”001000000”
Wash axis back	WASH	Axis moving flag & “000000000”
Rim cleaning	RIMS	Axis moving flag & “000000001”
Shine start	SHINE	Axis moving flag & ”000010000”
Shine axis back	SHINE	Axis moving flag & ”000010000”
Waxing start	WAXING	Axis moving flag & ”000000100”
Waxing axis back	WAXING	Axis moving flag & ”000000000”
Dry start	DRYING	Axis moving flag & ”000100000”
Dry axis back	DRYING	Axis moving flag & ”000100000”
Polish start	POLISH	Axis moving flag & ”000001000”
Polish axis back	POLISH	Axis moving flag & ”000000000”
Home	HOMING	“1010101010”
Done	DONE	“1111111111”
Undefined state		“0000000000”

Figure-1: Info displayed on seven segments display and LED’s

4 Implementation

4.1 Entities

Often a large FPGA design is broken into many entity/architect combinations where each entity contains the port map that defines all input and output signals for a particular entity.

In our design we have a top-level entity *e_my_car_wash_project.vhd* that connects to the other 8 sub entities.

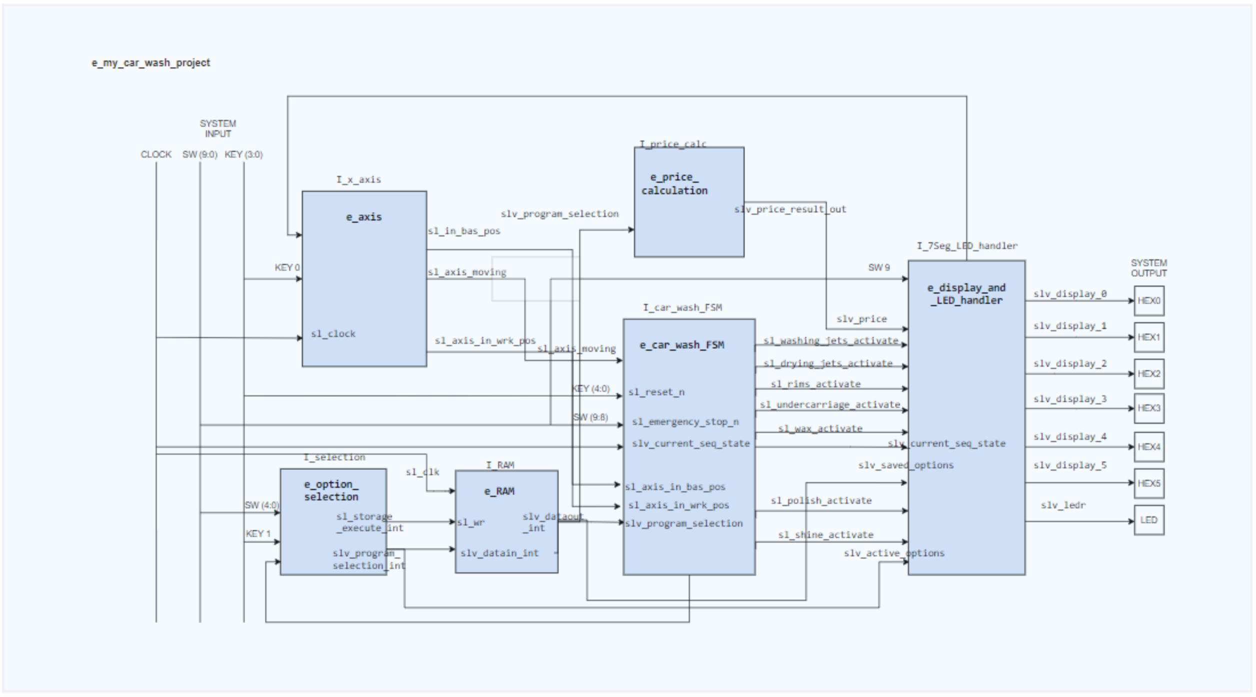


Figure-2: Block Diagram of carwash lane circuit

e_my_car_wash_project

The top-level entity of the design is e_my_car_wash_project. Its main task is serving the interfaces of all the other entities. Furthermore, it reads in the hardware inputs (keys, switches, clock signal) and passes them on to the entities that need them. After the sub-level entities have processed the signals, they also return some results, that the top-level entity either passes on to other entities or brings them onto the outputs (LEDs, 7 segment displays).

In addition, the top-level entity is generic. All values in every sub-level entity that will have to be changed for the purpose of easier simulation can be set out of the top-level entity. If the circuit will be loaded into hardware, the default values will be loaded. If the circuit should be simulated with a testbench, it is possible to parametrize all the values that are necessary for a clearly arranged simulation directly in the testbench by adjusting the e_my_car_wash_project entity with its generic map.

e_car_wash_FSM

The entity e_car_wash_FSM contains the finite state machine (FSM) of the circuit. It contains basically three processes that are handling the behaviour of the FSM.

The first process determines the next state to jump into in the next clock cycle. For that it contains a Case-When statement that checks in which state the machine is in the recent cycle. In all states there are transitions that determine the next step, programmed with if-else-statements. In the lowest else-branch the step always will stay the same. The transitions are reacting for example on push buttons, switches, or on the recent position of the car wash axis. Also relevant for the behaviour of the sequencer are the options that the user choosed at the beginning of the process. The information about these values come from the e_option_selection entity. There are also some states that are active for a decent amount of time and will be leaved when the time passed. For this purpose, the FSM entity also contains an instance of the e_modulo_counter.

On the following rising edge of the clock the second process only changes the state into the next state that was determined by the first process.

The third process checks again in which state the machine is in the recent cycle with a When-case statement and depending on that it writes the outputs individually for each step. The outputs are signals for some cleaning hardware, for example washing or drying jets, and furthermore the state machine gives commands to the e_axis entity. It determines in which direction the axis should move, and with what speed.

e_modulo_counter

This entity is used to count to a predefined integer number, it takes *reset*, *clock* and *rollover value* as an input; rollover value represents the number to which the modulo counter will count to. The entity will raise a *rollover* flag whenever it reaches that count, it will output the current count *Q* cyclically through a sequential process.

e_axis

This entity deals with the positioning of the washing machine axis and the speed at which the axis will be moving based on the current operation being performed, we have seven different operations (washing, undercarriage washing, rims cleaning, shining, waxing, drying and polishing).

As an input to the entity, it takes commands to move the axis to base position, commands to move the axis to working position and speed at which the axis will be moving, these inputs stem from the finite state machine entity and depends solely on the state in which we are in, we have states that represent every operation, and each state will drive the axis at a specific speed and to a specific position.

It outputs the position of the axis at any point of time, and flags that give an indication whether the axis is moving, at base position or at working position.

Two modulo counters have been used in this entity to simulate the distance crossed by the axis, in a way that describes how many milli seconds it takes to cross one milli meter. One modulo counter have been avoided to simulate this use case, since passing the distance equation as “to count to“ value to the modulo counter entity will increase the timing of our circuit.

e_option_selection

In this part, parameter selections are defined. These selections are about price calculation of operations, storage control, and selection enable. At first, there are several operations that are responsible for Car Wash Machine.

Before the car enters the machine, certain operations happen regarding the amount of coins that are entered; Rim Cleaning, Undercarriage Washing, Waxing, Polishing, and Shining-Washing. These operations are transferred to each bit of “slv_selected_params_int(4 downto 0)”. After that it is used in “e_price_calculation“ entity. Secondly, the “sl_storage_execute” is responsible for storing defined selections. Whenever the “sl_trigger_storage_int” is True, the data will be stored into ram. Followingly, “sl_selection_enable” and “sl_store_params” indicates that data will be either transferred to RAM or not. In short, the “sl_storage_execute” parameter enable/disables the data storing to “ e_RAM” entity. Thirdly, “sl_selection_enable” is already mentioned in RAM case, also it shows the parameters are either selected or not. Therefore, it is understood that these parameters control some entities before they operate.

e_price_calculation

Price calculation part takes the input “slv_params” that comes from “e_option_selection” entity. These parameters are defined as 5 bits. Selections are determined when the car enters the machine. Consequently, some prices are specified before the operations starts. The operations are Rim Cleaning, Undercarriage Washing, Waxing, Polishing, and Shining-Washing. These options are parametralized in the order by “slv_params”.

These parameters have also corresponding prices separately; Basic Price (4€), Rim Cleaning (1€), Undercarriage Washing (2€), Waxing (3€), Polishing (4€), Shining-Washing (1€).

Note that each price is defined as constant and concurrent entries are possible. After getting price from the user and some calculations, the entity results with an output value which is “slv_price_result_out”. Subsequently, the results would be ready to be shown in 7-Segment Display entity.

In fact, all prices are individually added on “Basic Price”. Logically, if one of these parameters is “True”, their price is added each other so that they bring out the total result that is “u_total_price_int”. If every parameter is “False”, only the basic price is added to total sum.

e_RAM

FPGA designs require a certain amount of memory. In our circuit the selection parameters are stored into the RAM memory after the selection and the payment. This could be later used in case if an emergency stop is applied and set it back to the previous selections. The entity e_RAM is defined for this purpose. The selection parameter is a 5 bit signal. So, the addr_width with the total number of elements to store is set to 1 and the data_width is specified to 5 (number of bits). The inputs are the clock signal ‘sl_clk’ and the write enable signal ‘sl_wr’ which is set.

Based on the status of the ‘storage_execute_int’, the data will be written if this is enabled and at the rising edges of the clock.

The address ‘slv_address_int’ is predefined to “0001”. The data input to write to memory ‘slv_datain_int’ will be program selection signal ‘slv_program_selction_int’. The output is set to the saved options.

e_display_and_LED_handler

Throughout the car wash process, the 7-segment display and LEDs should show the customer different information. That is the task of this entity. For the correct output, it is mainly decisive in which state the sequencer is currently (slv_current_seq_state). The 7-segment displays have hard-coded values or letters for each state except S_PRICE. With an interface to the entity e_price_calculation (slv_price), the calculated total price for the car wash is variably displayed to the customer. The emergency stop is a special case because it is not a real state of the sequencer (sl_emergency_stop_n).

On the other hand, the LEDs either show the additional options selected (slv_active_options) or purchased (slv_saved_options) by the customer or indicate in a kind of a loading bar how advanced the washing process is in relation to the individual steps.

Inside this entity, the e_convert_to_7seg entity is called and instantiated 6 times (6 x 7-segment display).

e_convert_to_7seg

This entity is a simple decoder for 7-segment displays. Depending on the input vector (slv_hex), a corresponding output vector (slv_display) is generated, which serves as input for a 7-segment display in order to display the desired number or letter.

4.2 Finite State Machine

State machines are often the backbone of FPGA development. State machines are logical constructs that transition among a finite number of states is therefore also called finite-state machine (FSM). Based on the current state and a given input the machine performs state transitions and produces outputs.

In the circuit of the car wash machine, a Moore state machine is used. It is easier to handle than the Mealy state machine and sufficient for the application. The Moore FSM changes its states depending on the input, but the outputs are only depending on the recent state of the FSM. So, if the sequencer stays in one state, the output will remain the same, although the inputs are changing.

The several states of the car wash state machine and the transitions are shown in the algorithmic state machine chart on the next page. The rectangles symbolize the different states, the rhombuses are if statements that are deciding which transition the sequencer will take. If the containing statement is true the sequencer will take the path of the “1”, if the statement is false, it will take the path of “0”. The sequencer starts with the state S_IDLE. If it has reached the S_DONE state and the activated timer elapsed, the FSM jumps back to S_IDLE and starts from the beginning.

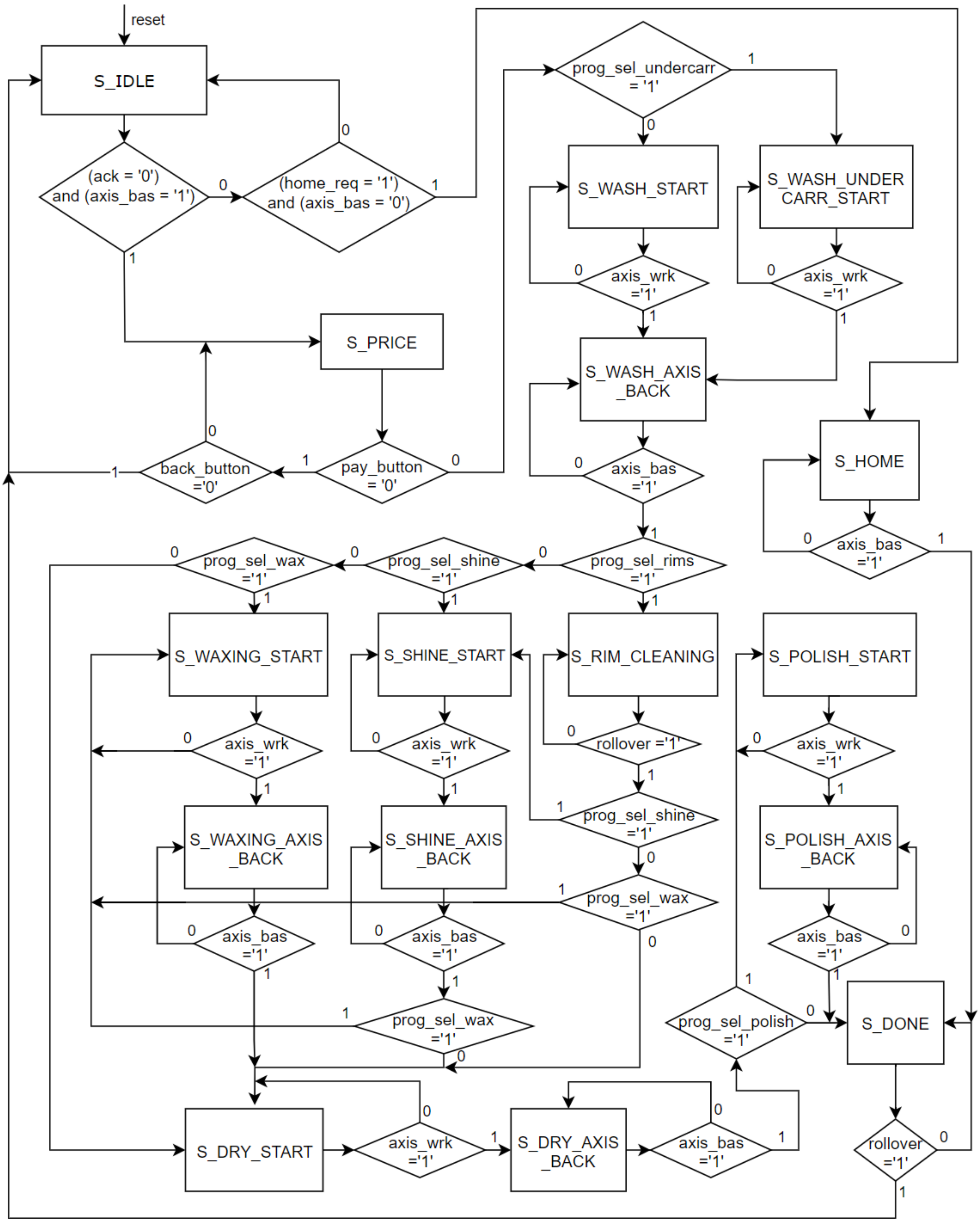
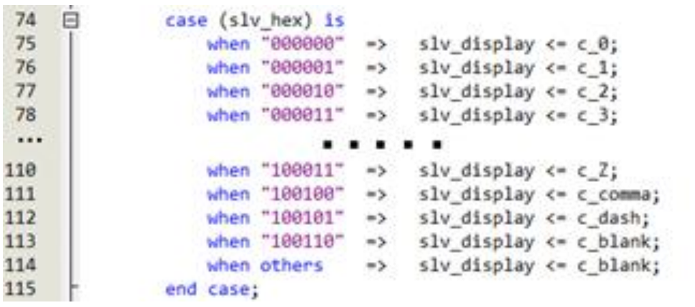


Figure-3: Algorithmic State Machine (ASM)

5 Timing of the circuit

To make the best possible use of the circuit´s potential, it is necessary to consider its timing. If only one part of the design is programmed in an inefficient way, the maximum possible frequency of the whole system will reduce. Therefore it´s very important to pay attention to the effects on the timing while programming each part of the system. In a software language it´s no problem to use for example long chains of “if…elsif…” statements, or additions of many variables in one turn (sum <= a+b+c+d+e+f+…). But as VHDL is a hardware description language, this way of programming would lower the maximum frequency and thus cause timing problems. It will result in a nesting of logic gates which creates a priority encoder, so the single statements would have to be processed in a daisy chain. VHDL code is not processed sequentially in a processor, but it´s a text description of a physical design. The programmed statements will be synthesized to a digital circuit whose components should be able to run in parallel as much as possible. Therefore, long constructs of “if…elsif…” cause a long data path that needs much time for the signals to stride through, because each elseif-part depends directly on the result of the previous elsif-statement. So generally, to take advantage of the strengths of FPGAs, in VHDL and other hardware description languages, it should be attempted to program the code in a way that as many logics as possible can be processed in parallel.

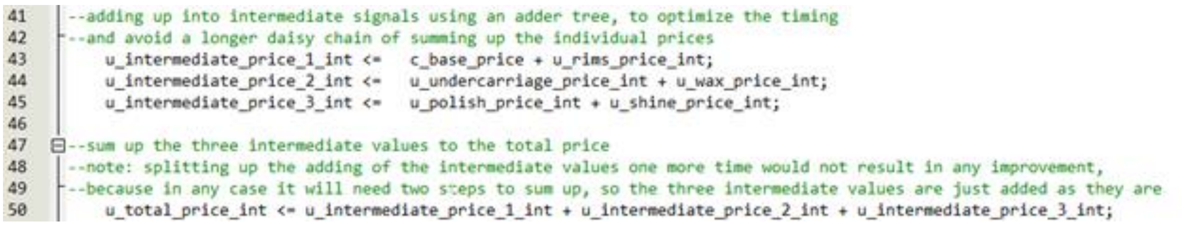
In the entity “e_convert_to_7seg” for example it would take much more time if there would be used an if…elseif… construct instead of the case…when… block.



```
        case (slv_hex) is
            when "000000"  =>   slv_display <= c_0;
            when "000001"  =>   slv_display <= c_1;
            when "000010"  =>   slv_display <= c_2;
            when "000011"  =>   slv_display <= c_3;
                          ▪ ▪ ▪ ▪ ▪
            when "100011"  =>   slv_display <= c_Z;
            when "100100"  =>   slv_display <= c_comma;
            when "100101"  =>   slv_display <= c_dash;
            when "100110"  =>   slv_display <= c_blank;
            when others    =>   slv_display <= c_blank;
        end case;
```

Figure 4: Case-When statement for decoding BCD into 7 segment display

Another example is the addition in “e_price_calculation”, where the sum of six prices must be calculated. If the single prices would be just summed up in one line (sum <= a+b+c+d+e+f+e) the resulting circuit would be a chain of five single addition blocks where each result depends on the result of the previous addition block. By calculating intermediate sums and adding those up the chain of additions can be reduced to an adding tree with a depth of three. The three intermediate values are summed up in one line because in either way it requires a depth of two operation to calculate the result. If there would be one more intermediate value, it would be better to split up this addition one more time. This way the depth of three would be possible for up to eight summands.



```
--adding up into intermediate signals using an adder tree, to optimize the timing
--and avoid a longer daisy chain of summing up the individual prices
    u_intermediate_price_1_int <=   c_base_price + u_rims_price_int;
    u_intermediate_price_2_int <=   u_undercarriage_price_int + u_wax_price_int;
    u_intermediate_price_3_int <=   u_polish_price_int + u_shine_price_int;

--sum up the three intermediate values to the total price
--note: splitting up the adding of the intermediate values one more time would not result in any improvement,
--because in any case it will need two steps to sum up, so the three intermediate values are just added as they are
    u_total_price_int <= u_intermediate_price_1_int + u_intermediate_price_2_int + u_intermediate_price_3_int;
```

Figure 5: Adder tree for summing up the price

Optimizing the timing of a hardware circuit to reach the highest possible frequency requires a decent amount of experience. The FPGA lectures only covered the basics of timing analysis. Based on those lessons a timing analysis with Quartus TimeQuest Analyzer was performed. With the default settings of the Clock, during a full compilation of the circuit, warnings emerged that the timing requirements are not met. To solve that the timing constraints, they have to be specified in an sdc-file. By reporting the timing of CLOCK_50 it´s possible to lookup the ten slowest paths throughout the circuit to be able to adapt some of those paths in the design to optimize the maximum frequency.

The circuit of the car washing machine has a maximum possible frequency of 231.21 MHz for the slow 85°C model. To retain some air to breath the frequency is only set to 200MHz in the sdc-file. This way the slack on the worst path, between when the data is required and when the data arrives, has the value ~0.7 ns.

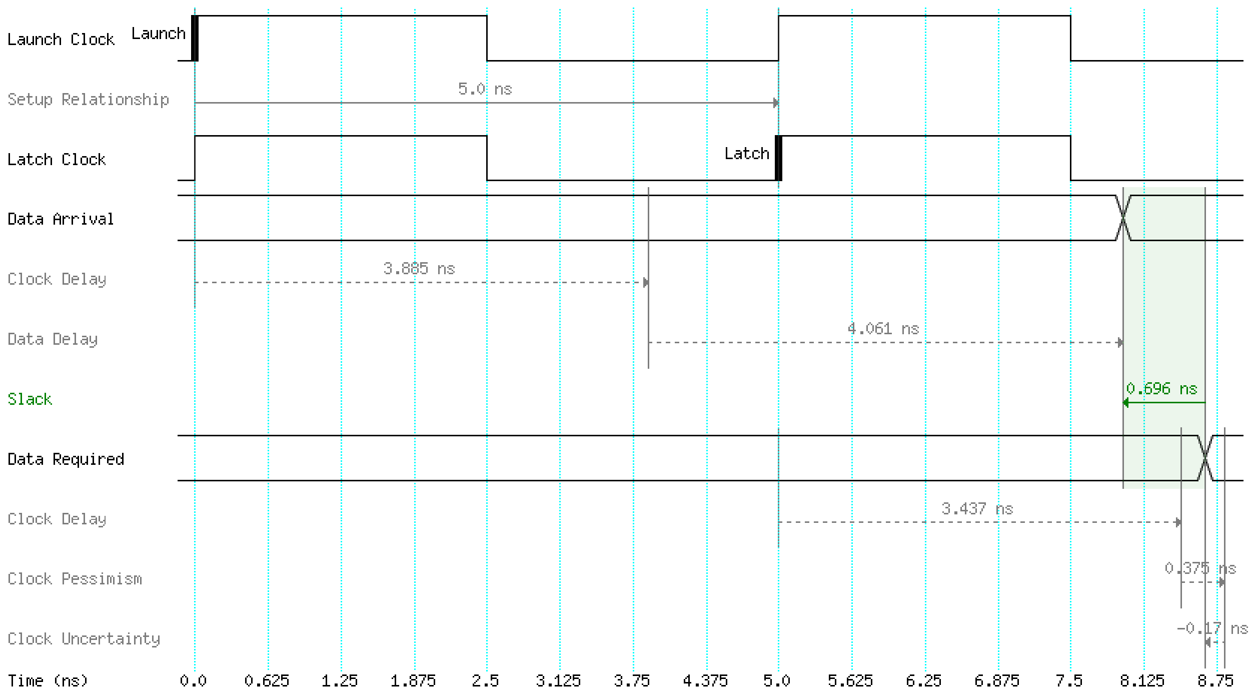
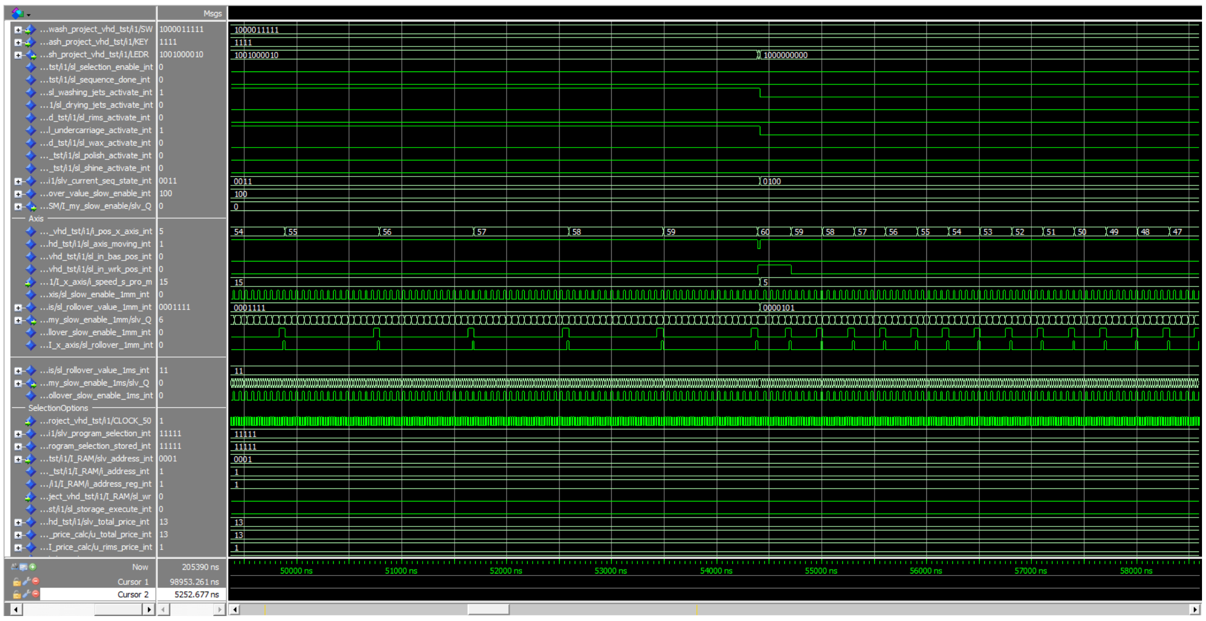


Figure 6: Overview of the slack values of the slowest path in the design

6 Simulation /Testbench

Simulation is a critical step in designing code that allows us the ability to look at FPGA design and ensure that it does what we expect it to. A testbench is a piece of code that provides the stimulus that drives the simulation. It could be considered container where the design is placed and driven with different input stimulus.

We have created a testbench that attempts to verify all possible functionalities of the circuit. Each unit was tested manually and with testbench before the integration.



```
                                       Msgs
...wash_project_vhd_tst/i1/SW          1000011111
...ash_project_vhd_tst/i1/KEY          1111
...sh_project_vhd_tst/i1/LEDR          1001000010
...tst/i1/sl_selection_enable_int      0
...tst/i1/sl_sequence_done_int         0
...sl_washing_jets_activate_int        1
...1/sl_drying_jets_activate_int       0
...d_tst/i1/sl_rims_activate_int       0
...l_undercarriage_activate_int        1
...d_tst/i1/sl_wax_activate_int        0
..._tst/i1/sl_polish_activate_int      0
..._tst/i1/sl_shine_activate_int       0
...i1/slv_current_seq_state_int        0011
...over_value_slow_enable_int          100
...SM/I_my_slow_enable/slv_Q           0
Axis
..._vhd_tst/i1/i_pos_x_axis_int        5
...hd_tst/i1/sl_axis_moving_int        1
...vhd_tst/i1/sl_in_bas_pos_int        0
...vhd_tst/i1/sl_in_wrk_pos_int        0
...1/I_x_axis/i_speed_s_pro_m          15
...xis/sl_slow_enable_1mm_int          0
...is/sl_rollover_value_1mm_int        0001111
...my_slow_enable_1mm/slv_Q            6
...llover_slow_enable_1mm_int          0
...I_x_axis/sl_rollover_1mm_int        0

...is/sl_rollover_value_1ms_int        11
...my_slow_enable_1ms/slv_Q            0
...ollover_slow_enable_1ms_int         0
SelectionOptions
...roject_vhd_tst/i1/CLOCK_50          1
...i1/slv_program_selection_int        11111
...rogram_selection_stored_int         11111
...tst/i1/I_RAM/slv_address_int        0001
..._tst/i1/I_RAM/i_address_int         1
.../i1/I_RAM/i_address_reg_int         1
...ject_vhd_tst/i1/I_RAM/sl_wr         0
...st/i1/sl_storage_execute_int        0
...hd_tst/i1/slv_total_price_int       13
..._price_calc/u_total_price_int       13
...I_price_calc/u_rims_price_int       1

Now                                    205390 ns
Cursor 1                               98953.261 ns
Cursor 2                               5252.677 ns
```

Figure 7: Simulation

Used use cases and tests for the entire and fully integrated system are listed below:

- **Test 1**: start process and interrupt with emergency stop, afterwards driving to home position, then starting process again
- **Test 2**: acknowledge selected options and try to change options before start, without pressing back button
- **Test 3**: acknowledge selected options, push back button, deselect all options, acknowledge and start base program
- **Test 4**: start base program and push ack, back and pay buttons multiple times --> must not affect the system
- **Test 5**: start base program and push reset button --> system (FSM, counters, etc.) must be reset except axis position
- **Test 6**: try to start car wash lane while emergency stop is active --> system must not start

- **Test 7**: press pay button after program selection without pressing ack button
  --> system must remain in initial state (S_IDLE)
- **Test 8**: press back button after program selection without pressing ack button
  --> system must remain in initial state (S_IDLE)
- **Test 9**: select option, press ack button, press pay button and interrupt system with home request
  --> home request must not interrupt current process
- **Test 10**: calculate and display all possible options
- **Test 11**: press request-home button after program selection without pressing ack button
  --> system must remain in initial state (S_IDLE)

For more details about single sequences/tests and individual used procedures, check out the testbench file e_my_car_wash_project.vht.

7 Conclusion

The project helped all of us to make best use of our knowledge about programming FPGA in VHDL acquired from the FPGA Programming classes. The design of the digital circuit comprising the requirements while at the same time considering the interests of the team members was challenging. Once the concept and design were finalized, the team members started working in individual units on different parts of the whole system. Each team member had to test their written entities on their own before bringing it to the overall system. This spared a lot of effort for testing the whole system when bringing the parts together. We defined different use cases in final test bench for analysing the performance of the circuit in various conditions. Exchange of ideas and collaboration between members helped to realize the project in 4 weeks timespan.

In future projects the circuit could be extended by more detailed programming of the car washing hardware, for example some sensors that measure the distance to the car while the x-axis is moving, or some control of the washing water system. But for that it would be better if there were more hardware, for example some analogue sensors, more hardware to display values. Maybe even some kind of hardware for a stepper motor.

8 References

Reichardt, Jürgen and Schwarz, Bernd. VHDL-Synthese, München Oldenbourg Wissenschaftsverlag, 2013.

Bryan Mealy, Fabrizio Tappero (February 2012). "Free Range VHDL"

Associated GitHub repository for the project:
https://mygit.th-deg.de/fs16623/fpga_car_wash_lane.git