1 Introduction

In this TP we are going to design a control system of a simple coffee machine and implement it on Gecko4Education. The basic operation of the machine is the following: 1) the user inserts a certain number of coins, 2) the user chooses the desired type of drink, 3) change is issued to the user if needed, 4) the drink is prepared and served. The exact details are slightly more complex than that and will be introduced gradually in the following sections.

General instructions

Please download the archive *templates.zip*. It contains the entity files for all the modules used in this TP (suffixed with *-entity.vhd*) and empty architecture files (suffixed with *-rtl.vhd*). To make sure that the automated grader recognizes the modules and interfaces correctly, please do not change any of the file, entity, architecture, or port names and write the architecture descriptions in the specified files only.

The solution is to be submitted for grading one module at a time, by uploading the architecture source file (*-rtl.vhd) to **https://digsys.epfl.ch**. The total number of uploads allowed is 45, which makes 3 attempts per module, on average. We will not impose a hard limit on the number of attempts per module. The only constraint is that the total does not exceed 45.

Hint: This will be a rather big project, but don't be terrified. We have already split it into small pieces of tasks, which you should be able to finish one by one. Don't worry if you cannot complete one of the tasks, and don't stuck at there giving up the following tasks. *Do* start early and save time for debugging. In this document, you will find 14 tasks, and in each task you will see *Hint* in this font telling you which file you should be working on for this task and which files you should have finished before putting your hand on this one. (Remember to also take a look in the corresponding *entity* file before you start, as some detailed descriptions are written there.) Feel free to not follow the order the tasks are listed, as long as the prerequisites are fulfilled. We also provide an estimated difficulty level for each task, so that you can try to finish the easier ones first and get as much points as possible ;)

1.1 Design Hierarchy

Our coffee machine, from the control perspective, consists of two main parts: the money storage unit, responsible for handling the coins during all trans-

actions and the drink preparation unit, responsible for preparing the drinks and keeping track of the available ingredients. A master automaton issues the appropriate commands to these two units, depending on the user inputs. This is shown in Figure 1.

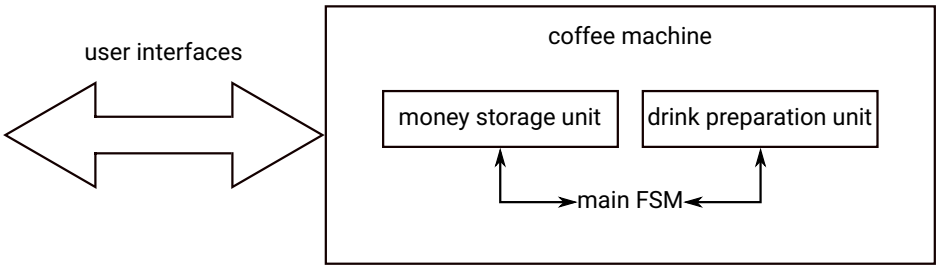

Figure 1: Basic design hierarchy.

1.2 Interfaces

To interact with the user, the machine has several interfaces, as shown in Figure 2.

1.3 Behavior

The behavior of the coffee machine is represented by the state diagram of Figure 3. When the user wants to buy a drink, they first need to insert the required number of coins. The accepted coin types are 5 franc, 2 franc, 1 franc, 1/2 franc and 20 cents. On Gecko4Education, coin insertion is emulated by pressing the appropriate button, as indicated in Figure 2. Once the coin insertion process is finished, the user should press the *accept* button.

Then, they can choose one of the five drinks, again by pressing the appropriate button. It may happen that there are no more ingredients for the chosen drink type. If that is the case, the user is asked whether they want to choose another drink instead. If the *yes* button is pressed, the user is allowed to choose again. If the *no* button is pressed, the coins that the user inserted are released and the machine returns to the idle state.

Once an available drink is chosen, the user is asked whether they want to use their own cup. To motivate them to protect the environment, if their answer is affirmative, the price of their drink is reduced by 20 cents. The answer is entered using the *yes* and *no* buttons and stored in the appropriate register. After that, the machine proceeds to calculate the change.

As we will soon see, the machine stores the coins in two separate storage units: *main* and *temporary*. The temporary storage unit is the one into

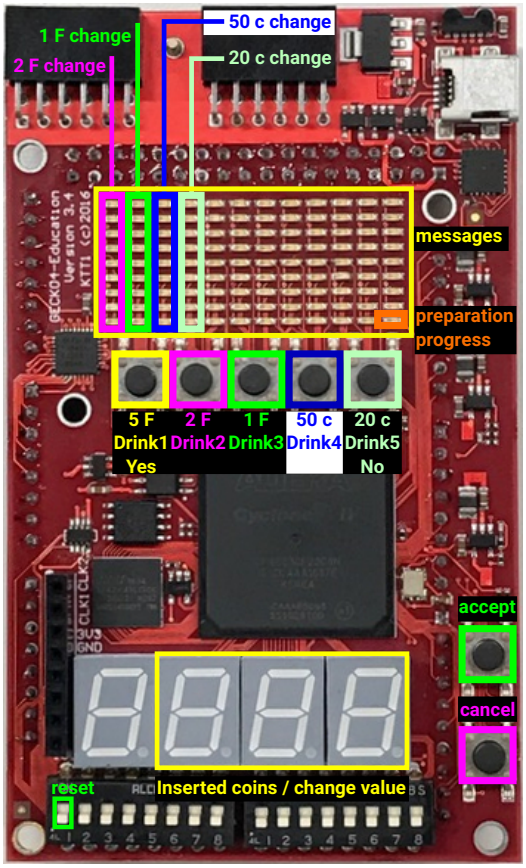

Figure 2: Coffee machine's interfaces on Gecko4Education.

which the coins inserted by the user fall. If the user decides to cancel the purchase at any moment before the drink has actually been served, the exact same coins that they inserted can be returned, by simply emptying the temporary storage. Once the entire transaction is complete, the money from the temporary storage is transferred to the main one. Any change that has to be returned to the user is taken exclusively from the main storage unit. Depending on the coins that the user inserted, the price of the drink, and the state of the main storage, it may not always be possible to return all of the change. Silently failing to return all change is not acceptable, of course. However, the user would often prefer to pay an extra 10 cents than to remain without coffee for the rest of the day, simply because the machine does not have the appropriate coins to return the change in full. For this reason, the machine calculates the maximum change that it can return, before the drink is actually served. It then asks the user if the calculated change is acceptable. The user again asserts their choice using the *yes* and *no* buttons.

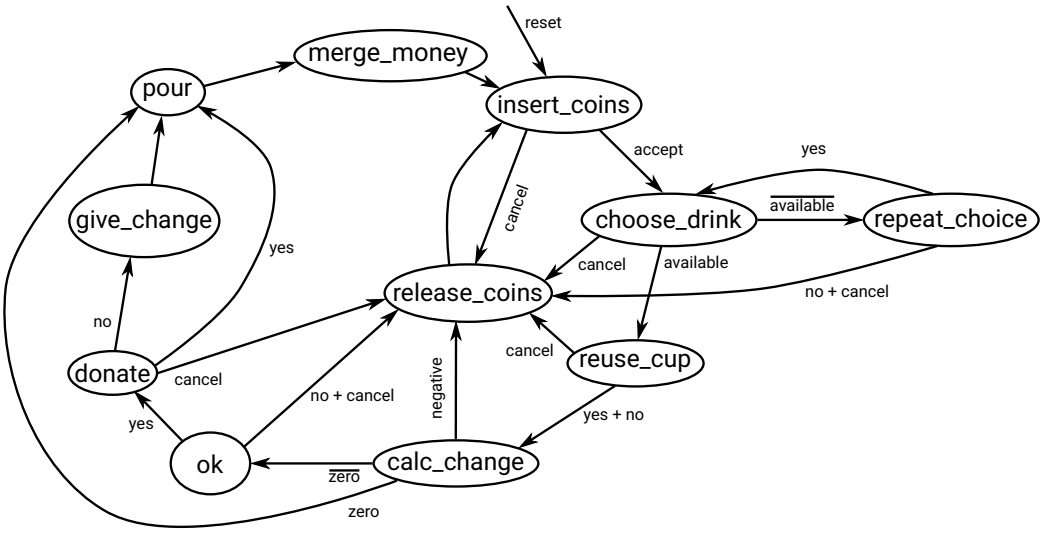

Figure 3: State diagram representing the behavior of the coffee machine.

Apart from stimulating the user to be more eco-friendly, by lowering the price of the drink if the need for wasting a new cup is avoided, the machine can be put to noble use in other ways as well. The price of the coffee it serves needs to be competitive, of course, as otherwise nobody will be willing to use it. However, many users will be happy to give up e.g., 20 cents of one coffee per day if they know that this money is going to be used, for instance, to provide food at school for children in the villages where the coffee beans are sourced from. Hence, before the change is actually returned to the user, they are asked if they want to donate it instead. If the answer is affirmative (*yes* button pressed), the machine proceeds to serve the drink. Otherwise, the change is issued, as calculated previously and agreed upon by the user, after which the machine serves the drink.

Finally, the money from the temporary storage is transferred to the main one and the machine returns to the idle state, waiting for the next user to start inserting the coins.

At several steps in the machine's operation, the user can press the *cancel* button which automatically cancels any previous choices and releases all inserted coins.

2 Money Storage

In this section we will design the money storage unit. Its block diagram is shown in Figure 4. As we have already mentioned, the unit consists of two

identical storage units: 1) temporary, which serves to hold the coins inserted by the user and 2) main, which holds all the coins after coffee serving and is responsible for issuing the change. The maintenance staff makes sure that the main storage has enough coins of each type at the start of the day, so that change can be properly returned.

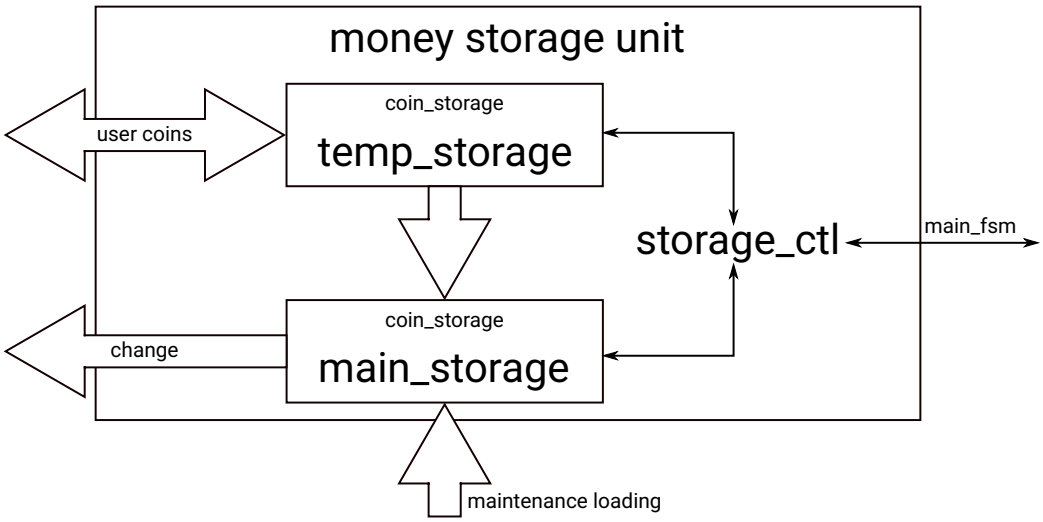

Figure 4: Block diagram of the money storage unit.

2.1 Individual Coin Storage Units

The coin storage units (*temp_storage* and *main_storage* in Figure 4) consist of five separate compartments: one for each coin type. The operations that the units can perform are: 1) addition of a coin to the chosen compartment, unless it is full and 2) removal of a coin from the chosen compartment, unless it is empty. The compartment is chosen by the one-hot-encoded signal *coin_type* where the MSB designates the 5 franc coin and the LSB the 20 cents coin.

If an attempt to add a coin to a full compartment or to remove a coin from an empty one is made, a fault is signaled. At all times, regardless of the command that is currently being issued (including none at all), the unit should output the number of coins in the selected compartment (zero, if no compartment is selected) as well as the total value of the coins in the storage unit; that is, the sum of the values stored in each of the compartments.

A block diagram of the coin storage unit is shown in Figure 5.

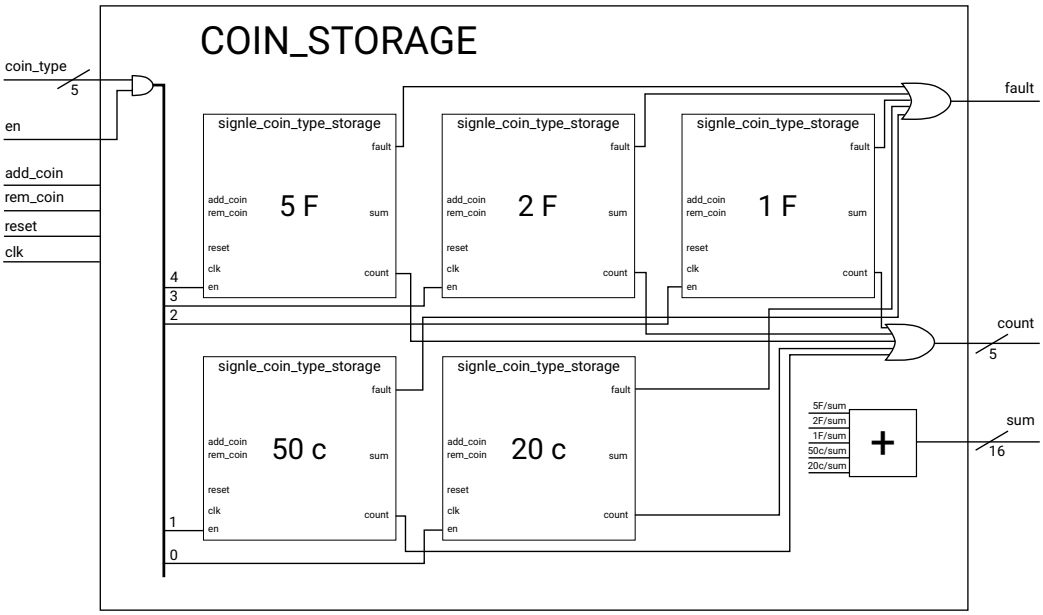

Figure 5: Block diagram of a coin storage unit.

2.2 Single Coin Type Compartments

As we mentioned in the previous section, the coin storage units are comprised of five distinct compartments, each holding coins of one of the five accepted types. A block diagram of the storage unit for any one coin type is shown in Figure 6. The unit can perform two operations, addition and removal of a coin, with the *fault* signal indicating that an attempt to insert a coin into a full unit or to remove a coin from an empty one was made. The operations are only executed when the enable signal *en* is asserted and they result in appropriately updating the coin count and stored coin value registers.

Because the individual compartments are composed in such a way that only the coin count of the selected one is being shown at the output of the complete storage unit, the *count* signal should be 00000 whenever the *en* signal is low. Then, we can use a set of simple OR-gates to multiplex the appropriate outputs, as shown in Figure 5.

Unlike the counts, the total sum of all compartment values is shown at the output of the complete storage unit. Hence, the *sum* signal always corresponds to the output of the sum register. The coin values are represented using a 16-bit fixed-point format, with 9 bits for the whole part and 7 for the fractional. It takes 9 bits to represent the maximum value stored in the main storage unit, when there are 31 coins of each type, while 7 bits provide reasonable guard for the accumulated error. Note here that 5, 2,

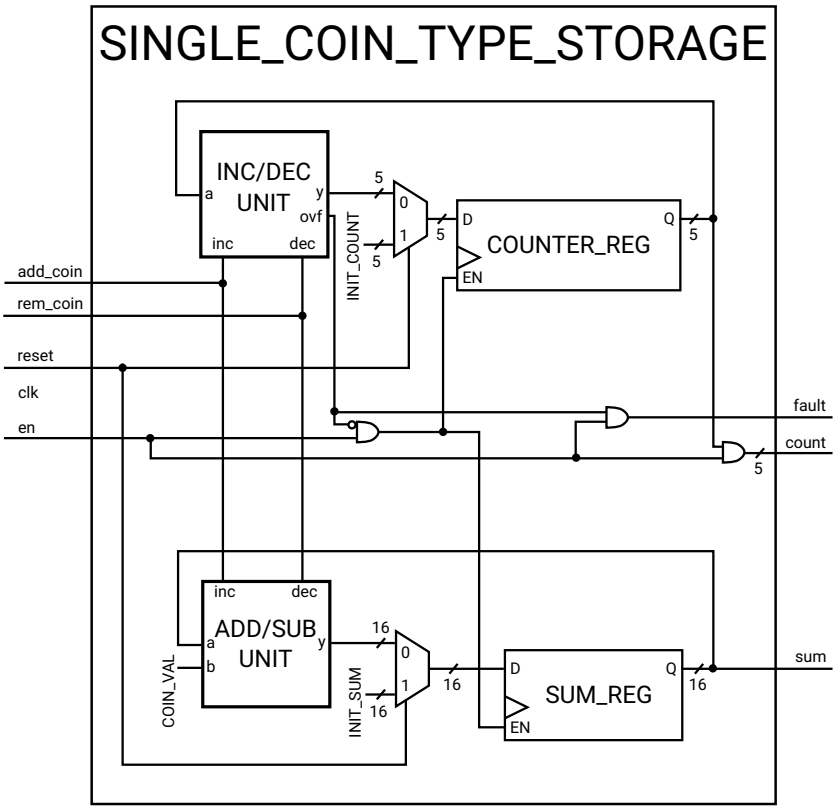

Figure 6: Block diagram of a single coin type storage unit.

and 1 franc coins can, of course, be represented exactly using a fixed-point format, as can 1/2 franc, but for the 20 cents coin, we would need infinite precision. A real-life machine would likely operate with custom arithmetic, but for the sake of simplification as well as to have an opportunity to practice using fixed-point numbers, we will employ the format just described. In the *fixed_point_pkg.vhd* file provided along with the template files, you may find some routines as well as constant and type definitions that can help you in the implementation process.

As was mentioned before, the maintenance staff can load the machine with a certain number of coins needed to return the change to the first morning users. This results in loading the registers of Figure 6 with the appropriate values, held by the two generic constants in the provided template files.

Task 2.1 Single coin type storage unit (8)

Implement the storage unit for a single coin type.
Hint: Complete the following architecture: *single_coin_type_storage*.
Difficulty: ☆☆☆

Task 2.2 Coin storage unit (5)

Implement the coin storage unit of Figure 5.
Hint: You need to have already completed the following architecture: *single_coin_type_storage*. Complete the following architecture: *coin_storage*.
Difficulty: ☆☆

2.3 Money Storage Control

In this section we will design the control unit of the money storage. The interfaces for communicating with the main FSM on one side, and the two coin storage units and the 7-segment displays on the other, are specified in the provided template files. The unit needs to perform the following five operations: 1) inserting coins into temporary storage, 2) calculating the change to be returned, 3) returning change to the user, 4) merging the coins from the temporary into the main storage, and 5) releasing the coins from the temporary storage back to the user. Which of the operations is to be performed is dictated by the main FSM of the coffee machine.

Tasks 1) and 5) are very straightforward and amount to asserting the correct inputs to the two coin storage units. We can assume that the mechanical steering of coins is catered for by the main FSM of the machine, so task 4) reduces to asserting the right coin storage inputs as well. To simplify things further, we can neglect the possibility of the main storage unit becoming full. What differentiates the task 1) from the rest is that the coin type is specified by the user, whereas in other tasks, it needs to be iteratively changed by the control unit of the money storage itself.

Tasks 2) and 3) are slightly more involved. Instead of truly maximizing the returnable change, the machine executes a simple greedy algorithm:

1. Starting from the coin of the largest value, return to the user the maximum number of such coins, without the value of the change returned thus far exceeding the total value to be returned.

2. When the addition of one more coin of the current type would cause the accumulated change to exceed the sum to be returned, proceed with the next, smaller coin type.

3. Stop when all coin types have been considered, or the accumulated change has met the required sum.

For example, if a change of 1.6 francs is to be returned to the user, the machine will return one 1 franc coin and one 50 cents coin, instead of one 1 franc coin and three 20 cents coins.

Remember that the change is not returned immediately, but only when the *give_change* command is issued by the main FSM, allowing the user to accept the change or cancel the purchase in the meantime. For this reason, the computed coin counts need to be stored in internal registers when performing task 2). In task 3), these registers are used to prevent emptying the main storage unit entirely, and to guarantee that the user gets back the amount they agreed on.

Since the coin storage units can accept or release at most one coin at each clock cycle, the operations of the money storage unit will take multiple cycles to complete. Moreover, the number of coins to be manipulated will differ from operation to operation, even if the operation type is the same. In other words, we can not know in advance how many clock cycles each operation will take to complete. We must guarantee, however, that the main FSM does not move to the next state, before the issued command is completed as this would lead to malfunctioning (e.g., asking user to confirm that they agree with the change to be returned, before it has been increased to its final value). For this reason, we implement a simple *four-phase handshake protocol* for communicating between the main FSM and the control unit of the money storage, as shown in Figure 7. Note that coin insertion is controlled by the main FSM itself, for the most part, so this operation requires no synchronization.

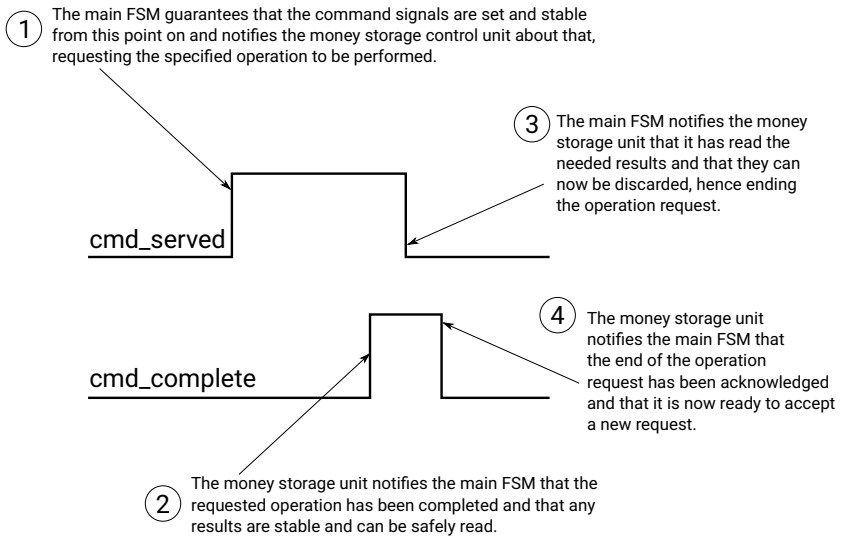

Figure 7: A four-phase handshake protocol for communication between the main FSM and the money storage control unit.

Task 2.3 Money storage control unit (34)

Implement the money storage control unit. Correctly releasing the coins from the temporary storage gives 10 points. Correctly merging the coins from the temporary into the main storage gives 2 points. Correctly calculating the change is worth 15 points, while correctly issuing it to the user is worth 5 points. Finally, having the entire money storage control unit completely functional gives additional 2 points. In all cases, the handshake protocol must function properly for the points to be recognized.

Hint: Complete the following architecture: *storage_ctl*.

Difficulty: ☆☆☆☆☆

Task 2.4 Money storage unit (5)

Assemble the modules designed so far into the complete money storage unit of Figure 4.

Hint: You need to have already completed the following architectures: *single_coin_type_storage*, *coin_storage*, *storage_control*. Complete the following architecture: *money_storage*.

Difficulty: ☆

3 Drink Preparation Unit

The block diagram of the drink preparation unit is shown in Figure 8. As soon as the user confirms their choice of the drink, this information is passed to the drink preparation unit by the one-hot-encoded signal *drink_type*. It then outputs the appropriate price, in the 16-bit fixed-point format presented before, and indicates whether the drink is available or not. Once the drink pouring command is actually issued by the main FSM, the timer inside the preparation unit begins to count the number of seconds necessary to prepare the purchased type of coffee. The timer is also driving an LED, making it blink with a frequency of 1 Hz, to indicate the preparation progress. Once the preparation time is up, this is signaled to the main FSM. By that time, the register bank holding the number of ingredient packages for each coffee type is updated accordingly. Like the money storage unit, the drink preparation unit communicates with the main FSM using the four-phase handshake protocol of Figure 7, because the time taken to prepare each drink depends on its type. Also like the money storage unit, the maintenance staff loads the drink preparation unit with the necessary ingredients each morning. This is reflected on the register bank upon reset.

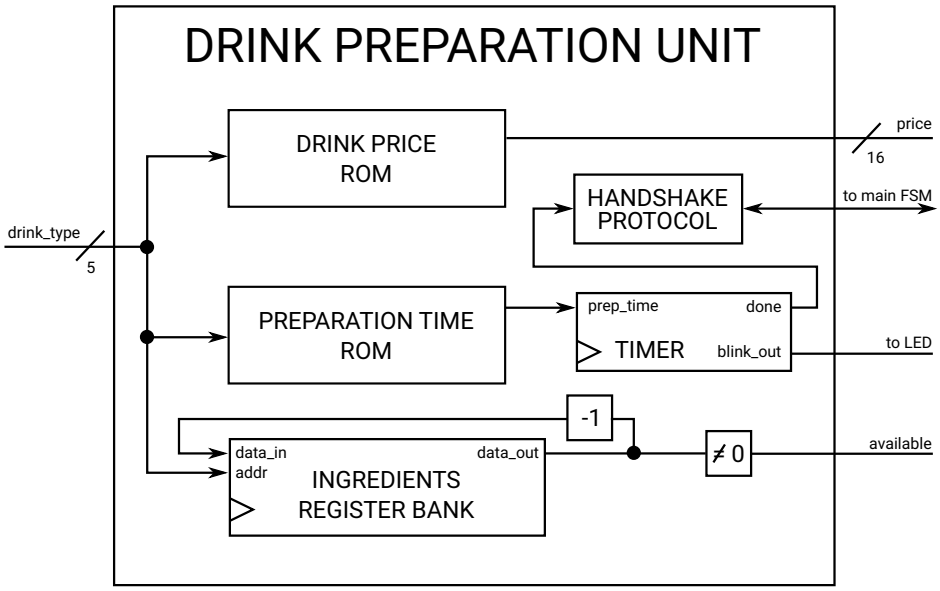

Figure 8: Block diagram of the drink preparation unit.

Task 3.5 Drink preparation unit (10)

Implement the drink preparation unit of Figure 8.
Hint: Complete the following architectures (in the suggested order): *timer*, *drink_preparation*.
Difficulties: ☆☆/☆☆☆☆

4 Interfaces

In this section we discuss the various interfaces used for interaction between the machine and the user, as shown in Figure 2.

4.1 7-Segment Displays

The three rightmost 7-segment displays are used to display the total value of the coins inserted by the user and the total value of the returnable change. Otherwise, all displays should remain turned off. The first of the displays outputs the whole part, which can never be larger than 5 francs, while the remaining two display the fractional part, truncated (not rounded) to two most significant digits.

The signal going out of the money storage control unit is the appropriate value encoded in the 16-bit fixed-point format presented before. The first step towards displaying this value on the 7-segment displays is to separate the whole and the fractional part, using the *sum_splitter* module. This module outputs the 9-bit whole part of the input sum and the two most significant digits of the fractional part, encoded in natural binary, using 7 bits.

Task 4.6 Sum splitter (10)

Complete the *sum_splitter* module. You are not allowed to use multiplication, division, modulo, or remainder operators.
Hint: Complete the following architecture: *sum_splitter*.
Difficulty: ☆☆

Once we have the whole and the fractional parts split, we need to convert these natural-binary-encoded numbers to BCD digits. This is done by the *bin_to_bcd* module that takes an 8-bit binary encoded number and outputs two BCD digits corresponding to the tens and the ones of the input number. The input number is guaranteed to be within $[0, 99]$.

Task 4.7 Binary to BCD converter (8)

Complete the *bin_to_bcd* module. You are not allowed to use multiplication, division, modulo, or remainder operators.
Hint: Complete the following architecture: *bin_to_bcd*.
Difficulty: ☆☆

Finally, we need to convert the BCD digits to the appropriate signals driving the 7-segment displays. This is done by the *bcd_to_7seg* module.

Task 4.8 BCD to 7-segment converter (2)

Complete the *bcd_to_7seg* module and assemble the display driver module *disp_driver*. Note how much easier it is to describe the converter at the level of abstraction provided by VHDL than the one you used in the first part of the course.
Hint: Complete the following architecture: *bcd_to_7seg*.
Difficulty: ☆
Hint: You need to have already completed the following architectures: *bin_to_bcd*, *bcd_to_7seg*, *sum_splitter*. Complete the following architecture: *disp_driver*.
Difficulty: ☆

4.2 LEDs

The LEDs are used to display the messages to the user from the main FSM, the change composition by the money storage control unit, and the coffee preparation progress, by the drink preparation unit. The LED matrix is stored in a row-major order, in the *leds_out* output signal of the *led_driver* module. In the provided template for this module the message "ok" is included. The rest of the messages that are to be displayed to the user are shown in the gallery of Figure 9. The coin counts for the returnable change decomposition are displayed by thermometer encoding, in the four leftmost columns of the matrix, with the rightmost of these corresponding to the 20 cents coins.

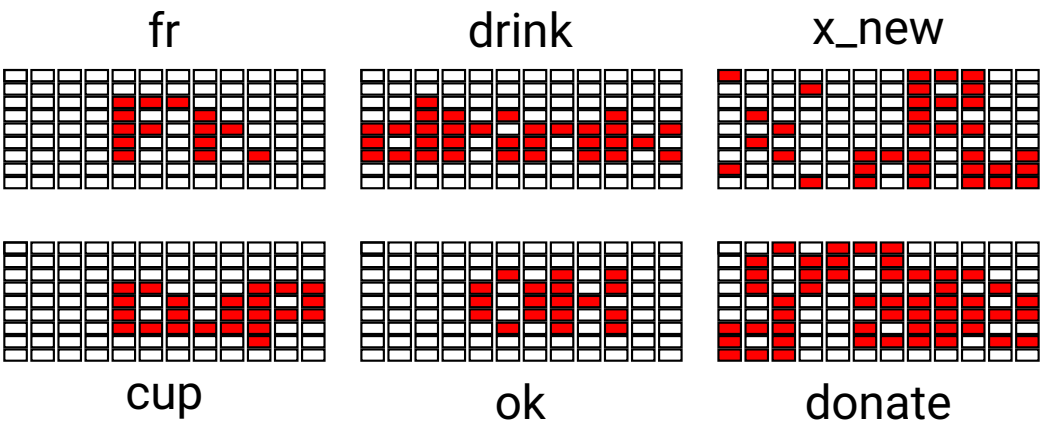

Figure 9: Messages displayed on the LED matrix by the main FSM.

Task 4.9 LED driver (3)

Complete the *led_driver* module.
Hint: Complete the following architecture: *led_driver*.
Difficulty: ☆☆☆

4.3 Buttons

There are two problems with the physical buttons that we must address here: 1) the user is free to press the button at any moment they desire, which may fall into the $[T_{su}, T_h]$ window of some flip-flop, leading it to metastability and 2) due to imperfection of the metal surfaces of the switch contacts and/or tilting of the contact surfaces on the spring, it can happen that while the user is applying force to the button, the current flow gets established and broken multiple times, resulting in multiple transitions of the signal at the output; this phenomenon which is called *switch bouncing* is illustrated in Figure 10.

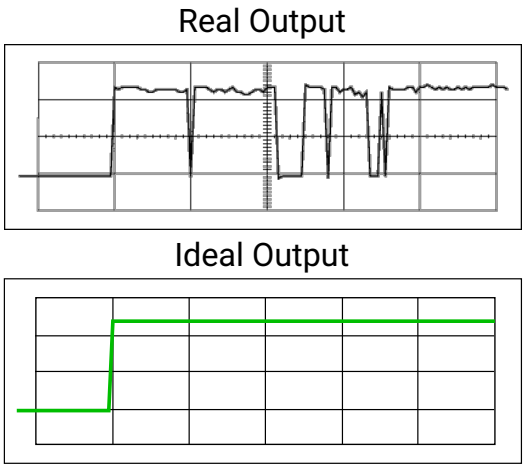

Figure 10: Output of a real mechanical switch illustrating the phenomenon of *switch bouncing*. Courtesy of Maxim Integrated (https://www.maximintegrated.com/en/design/technical-documents/app-notes/2/287.html).

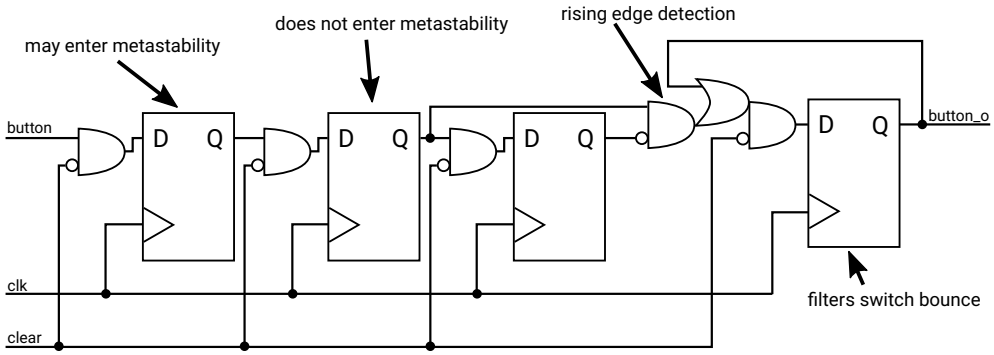

Figure 11: Switch synchronization and debouncing circuit.

The circuit of Figure 11 solves both of these issues. Try to understand how. For the coffee machine to function at all, we must use this circuit at the output of each button.

Task 4.10 Debouncing circuit (1)

Implement the debouncing circuit of Figure 11.
Hint: Complete the following architecture: *debouncer*.
Difficulty: ☆

Even after synchronization and debouncing, all problems with the buttons are not solved. The machine operates at 12 MHz clock frequency, leaving little time for the user to lift their finger from the button before the current press is detected again. Admittedly, we could reduce this frequency, but that would cause the multicycle operations of coin manipulation to take unnecessarily long time, making user experience poorer than it must be. Hence, we should find a different approach.

Task 4.11 Press delaying (2)

Integrate the timer that you developed in Section 3 with the debouncer of the previous task, so that when the *clear* input is asserted by the main FSM, it remains active for another second, thus giving the user enough time to lift the finger before making another press.
Hint: You need to have already completed the following architectures: *timer*, *debouncer*. Complete the following architecture: *timed_button*.
Difficulty: ☆

5 The Main FSM

Now that we understand all the details that the main FSM whose state diagram was presented in Figure 3 needs to take care of, we can actually implement it.

Task 5.12 Main FSM (8)

Implement the main FSM of Figure 3.
Hint: Complete the following architecture: *main_fsm*.
Difficulty: ☆☆☆☆

6 The Entire Machine

With all the pieces in place, the only thing that remains is to put them all together.

Task 6.13 Complete machine (4)

Assemble the entire machine.
Hint: You need to have already completed all the other architectures. Complete the following architecture: *machine*.
Difficulty: ☆☆

7 Hardware Testing

We have completed the design process and tested that all individual components, as well as the entire machine, work in simulation. However, in the example of switch bouncing, we have seen that simulation can only go so far. The real test of our machine comes when we implement it in hardware—in our case on the Gecko4Education FPGA board—and demonstrate that it works by interacting with it in the way a real user would.

Task 7.14 Hardware demonstration (0)

Implement the machine on Gecko4Education and demonstrate that it really works as expected.

A Special Note

As we are away from campus and you do not have access to the FPGA boards, this last step is not entirely practicable. This is why it is not graded, as it normally would have been. However, you can complete almost the entire process of mapping your design to the FPGA; the only missing part being downloading the generated bitstream to the board itself. We strongly advise you to do so as you will have at least two more courses expecting you to use Gecko4Education and there it will be largely assumed that you know how to complete the synthesis process in Quartus.

Besides, by now you gained solid understanding of the principles governing the functioning of digital circuits, you acquired skills in both the

classical design techniques such as hand-minimization of Boolean functions, and the modern, language-based ones. Wouldn't it be cool to make this one more step and be able to actually implement your designs in real hardware? If nothing else, it could be a great asset for the next LauzHack or it could enable you to make the World a little better, for instance, by letting people drink slightly more equitable coffee :).

We are here to help you during the TP sessions, so please use that possibility. After the grading deadline is over, we will open another submission series, where interested students will be able to upload their bitstream, which will then be tested on the board by the TAs. Due to our limited bandwidth, we will allow only two such submissions per student. These will, of course, not be graded, but they will give important feedback on whether the steps performed in mapping the design to the board were done correctly.