Second laboratory session

Digital Electronics

Programming an elevator microcontroller

Roberto Uceda Gómez

NIA: 100346538

Date of the session: 16-10-2018

GROUP 29 - GITI

Introduction

For this session, there are several tasks to accomplish, based on the previous session:

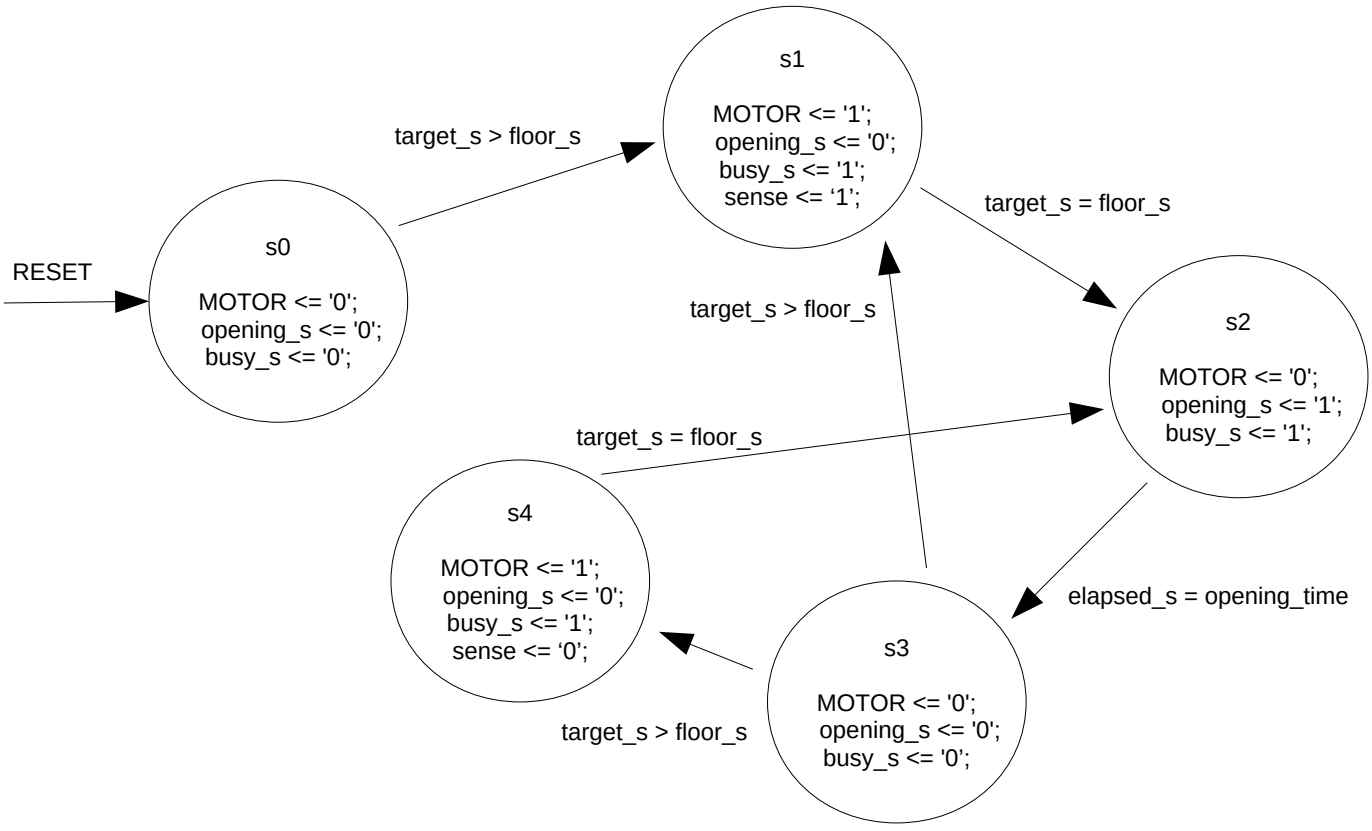
- Implement a timer, so as to allow the door to be opened and closed automatically.
- Redesign the system as a state machine
- Use the test bench to verify the design.

My proposed solution is in several vhd files alongside this document.

I opted for a modular approach, so as to be able to recycle most of my work for the first session (code available at github.com). The timer and the main entity (elevator) had to be written from scratch.

For the new state machine paradigm, I tried several options, but decided to opt for a simpler machine with few states. It could have been implemented with many more, but I wanted to keep the code readable and not redundant.

A draft for the state diagram chosen:



Note that s0 is the initial state (after reset), with all variables set to zero; state s1 is moving upwards; state s4 is moving downwards; state s2 is opening the door and holding it open; and s3 is stand-by (waiting for next input).

Code Commentary

Having a clear idea of how the main entity was going to be, it was time to write.

First of all, I had to build a timer. The solution proposed comprises a clock cycle counter which stores the number in an integer type variable, and then sends it as output.

This, even though it is simple and fast, has a big disadvantage: the register for the value needs to be very big. For example: for a 3s timer, with a 1KHz (as specified in the lab instructions), the register needs to store an integer of value 3s / 1ms = 3000, so it would need a 12 bit register. But for this session, an optimized solution for this problem is not considered.

A quick review of the other blocks used in this session:

- Edge detector: sends a clock pulse for each rising edge of the input.
- Counter: 2 bit counter, counts rising edges in input. With sense and overflow bits (overflow not used in this case)
- Register: 2 bit register, with enable. Stores input value on clock rising edge
- Encoder: one-hot 4 bit encoder. With active output signal which is high if the input combination is accepted (only one bit active at a time).

The top-level entity for this project is the ‘elevator’, in the ‘elevator.vhd’ file.

In it reside the connections between blocks (instantiation and link through intermediary signals) and the state machine.

The code starts with the necessary library declarations. Then, the entity ‘elevator’ is declared as per the specifications:

Inputs: CLK, RESET, B, S. Outputs: SENSE, MOTOR, OPENING, BUSY, FLOOR, TARGET.

Next, the architecture:

- Type declaration for the states structure, and the corresponding signals.
- A constant for the opening time, which comes handy when having to change this value
- Declaration of the used blocks: register, edge detector, encoder, counter, timer. As per usual, it is almost the same code block as their respective entity declarations.
- Intermediate signals. These are important, as they are needed for internally linking the components, and doing more complex operations.

The actual architecture starts after all this.

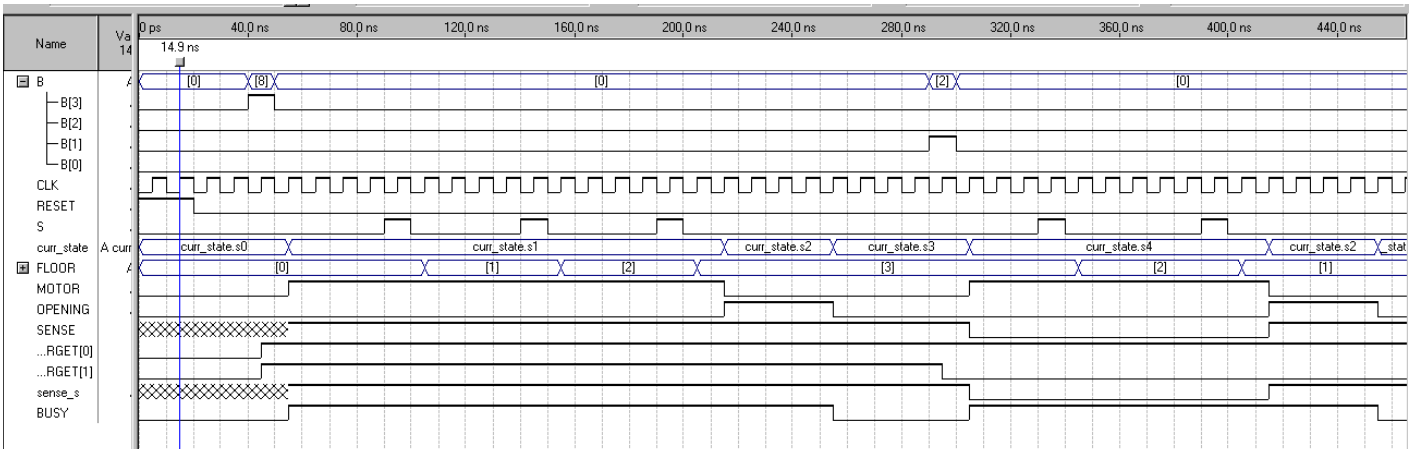
- First of all, it is needed to map the components each with its respective inputs or outputs. Given how each variable has a descriptive name (those with a '_s' appended are internally used only, those which are not, are actual pins), please refer to the source code for more details. It is important to point that there are two register instantiated, as there is need for two of them. For this case, tags prove useful to differentiate between them.
- Synchronous process: tagged 'sync_process'. Resets the whole machine to state s0 on RESET high, and schedules state changes for a rising edge of the clock. This synchronizes the system, along with each block clock input.
- State machine: tagged 'state_change'. Controls the flow of the program. As seen in the state diagram, I chose a Moore type machine. When either B, S or curr_state change (signals in the sensitivity list) change, a case-when block set the outputs accordingly, as well as setting next_state based on the main signals used: floor_s and target_s (the intermediate signals for the position and the destination selected for the elevator).
- Finally, outside a process, there are the physical signal assignation, which sends the internal signals to the actual pins.

Testing

So as to ensure smooth operation when linking all the blocks together, it was necessary to test everyone of them before joining all in the main elevator entity. As it is outside the scope of this practice, it will not be shown in this report, but these tests can be found in the repository for the first session (link above).

The most common compilation errors I had to correct were syntax-related. Although, I had to try with several types for various signals, as there is no compilation-time type conversion. Every type had to be the same when declaring in the entity, when port mapping, and when comparing and assigning in the main entity. This lead to some troubleshooting: for example, I had to discard using a signal of type ‘time’ for the timer output, as it was not easy to work with, difficult to set the units, and even more for conversion.

Once solved, I could test the whole system. I started with a simple vwf time diagram, with the most significant I/O variables, to have a glance at the general behaviour of the machine.



This is the time diagram used.

I tested for an arbitrary clock period, as it does not affect the final result. Also, internal delays produced by the hardware were not considered.

The test case is two button presses, both with their corresponding floor pass signals, so that it shows the elevator can go up or down.

Even though it shows a clock cycle delay, the system works. The state changes when a discrepancy between the target and the present floor is found, and with it, the motor, sense, and busy signals.

The busy signal stays on during the opening period, too. This worked satisfactorily, proving the timer is functioning as expected.

Note: for this test, the opening time was set to 3 clock cycles.

A test bench was written, and can be found among the files attached (elevator_test.vht, auto-generated in Quartus, and elevator_test_2.vht, which was written from a bare-bones template). But, it could not be used in Multisim due to a compile-time error, of which the culprit could not be found ("*Actual expression (prefix expression) of formal "reset" is not globally static*").

Conclusion

The biggest difficulty found in for this practice was designing the state diagram. I could opt either for a machine that considered every situation possible as a state (situations such as position of the elevator, and from there, whether if going up or down), or rather a more concise one, with only the necessary inputs and states. For the sake of not repeating the code many times, and to keep it as short as possible, I chose the later.

Still, the model is not ideal, as there are cases not considered. For example, if the elevator is in stand-by (state 3) and the button for the same floor it is on is pressed, nothing will happen. This should be corrected, as in a real application this feature is pretty much needed.

On the other hand, coordinating the blocks through the main entity presented little to no problem at all. Though the declaration and mapping is quite verbose, once it was proof-read, with no syntax error and the mapping done correctly, it worked as expected, with not much of a headache.

On an ending note, even if the program is well written and the simulations are correct, testing with real hardware is on another level, and so debugging on a real FPGA may not be easy or intuitive, as there are many variables beyond the programmer's reach.

However, all things considered, learning HDL breaks with many molds found in traditional programming languages, and so has been an enriching and mind broadening experience.