Computation II - 5EIB0
Lab 4: Creating a Elevator v1.0

Diederik Markus Wouter Schoenmakers

In previous labs, you have created a Moore and a Mealy machine. In this lab your are going to make another FSM machine. Since the outputs of both machines are the same, you may use your preferred machine to implement it. This lab has the same setup as the earlier lab assignments. First you will develop your FSM itself, then adapt the AXI4Lite interface, followed by making the block diagram and programming the board.

Contents

1 Create the project

The first thing to do is creating a new empty project.

Task 1. Create a new project.

The next step is to create a new IP-block, which will contain your elevator FSM machine.

Task 2. Create a new IP-block with 4 registers. Name the IP block `ElevatorFSM` and add a design source named `ElevatorFSM.v`.

2 The FSM

Now the basic part of your project is set-up. The next step, before implementing is choosing which kind of FSM you will make. The follow-up is making a state diagram, before your start programming in verilog. The elevator FSM will consist of three floors. The elevator can move up and down between the three floors. On each floor there is little display above the elevator doors to provide information to the waiting people. The following information is provided: at which floor the elevator is and whether it moved up, down or did not move at all.

2.1 Preparation

Task 3. Choose whether you implement a Mealy or a Moore machine. Make a state diagram of your FSM. The requirements are:

- The FSM has three floors: 0,1,2.
- The elevator can move up on floor 0 and 1, can move down on floor 1 and 2. On all floors the elevator can stand still.
- The FSM has two (`motor_control`) input switches: one to go up, one to go down. The up signal is the most significant bit.
- Both input switches can not be high at the same time.
- If the elevator is at the lowest floor (floor 0), it can not move down. If an input signal for this invalid move is present at `motor_control`, the signal will be considered as a "staying at the same floor"-signal.
- If the elevator is at the highest floor (floor 2), it can not move up. If an input signal for this invalid move is present at `motor_control`, the signal will be considered as a "staying at the same floor"-signal.
- The FSM has a two bit output `movement` to show the last movement of the elevator. The most significant bit of `movement` represent if the last movement was going up.
- No led is lighted when the elevator stayed at the same floor.
- Transitions happens at the posedge of signal `update`. Use level triggering for this.
- The number of internal states may be as many as you want.
- When the machine is reset, it will start at floor 0, with no leds lighted. In other words, `movement` is zero.

2.2 The logic

You will now have a clear schematic what you are going to implement in verilog. This schematic will be basis for the next task.

Task 4. Implemation of the elevator FSM.

1. Add the following input ports, with the lowest possible number of bits to `elevatorFSM.v`: `clk`, `motor_control`, `update` and `reset`.
2. Add the following output ports, with the lowest possible number of bits: `floor`, `movement`.
3. Write the implementation of the final state machine as drawn by yourself in the previous task. If it turns out that your machine does not have the right behaviour, go back to the previous task.

2.3 Logic Testbench

To make sure that the logic works as intended you are going to create a testbench. This testbench should try all possible transitions in your final state machine just line in the previous lab.

Task 5. Write a testbench that tries all possible transitions in the state diagram and tries `reset` in every possible state. Simulate you testbench to verify that your elevator works correctly.

2.4 The AXI4Lite interface

Your elevator FSM is now ready and should be connected to the AXI4Lite bus. You are going to use the AXI bus to read the current state of your FSM as well as the value of the inputs.

Task 6. Connecting the AXI4Lite interface.

1. Include your module in the AXI4Lite interface.
2. Connect `clk` to the clock of the AXI4-Lite bus.
3. Connect the signals in the following order to the AXI4Lite registers: `motor_control`, `floor`,`movement`.
4. Connect the signals `reset` and `update` to the last register. The signal `reset` should be connected to the least significant bit.
5. Make all registers read only.

Task 7. Add the following external input ports to the IP block `update`, `reset` and `motor_control`. The following external output ports should also be added: `floor` and `movement`.

2.5 AXI4Lite Testbench

Since section 2.4 is practicably the same as for Lab 2, we expect you to be able to adapt it without the need to test it. When you are not sure, if you did it correctly, you can recap to lab 1 on how to write the testbench for the AXI4Lite interface.

Task 8. Synthesize your IP. In the bar on the left click on "Run Synthesis". Make sure that you resolve all errors and **critical warnings**. Vivado can synthesize a design with critical warnings however they often fail during implementation which means you will not be able to program the board. When Vivado is done synthesizing open the synthesized design do not run implementation.

Task 9. Package your IP block. Review Lab 1, if you do not know how to do it. When you are done switch back to the original project that you created in section 1;

3 Debounce module

Just like in Lab 2 you need to debounce the buttons before you can use them, you can copy the module you wrote for the moore machine.

Task 10. Copy `debounce.v` from the project of Lab 2 to your current project.

4 Testing on the board

In this part of the lab you are going to build the block diagram, setting up the constraint file, write the software and eventually program it on the Pynq board.

Task 11. Create a block diagram with the Zynq processor, your custom IP block, the debounce module and external ports. **Be aware of the high/low reset line**. Generate a HDL wrapper when your block design is ready.

Task 12. Create a constraint file `constraints.xdc`. Copy the template for the pynq board in this file and edit the template in such a way that: the first two leds are connected to `movement` and the last two leds are connected to `floor`. The switches should be connected to `motor_control`, the first button to `reset` and the second button to `update`.

Task 13. Generate the bitstream. This will probably generate a lot of warnings, this is not unusual so can be ignored.

1. Click on "Generate Bitstream".
2. If you get a pop-up message asking to run synthesis and implementation click on "Yes".
3. In the next pop-up keep all the default settings and click "OK"

You can now write the C code for the ZYNQ processor. The code should read the input and output of the FSM and write them to the UART peripheral. A Universal Asynchronous Receiver-Transmitter is a piece of hardware that can be used to send messages to other chips or in this case to your laptop via the USB cable.

Task 14. Launch the SDK with a hello world project. Write a program that continuously reads the inputs and outputs of the FSM Machine and writes them to the UART. In the file `helloworld.c` you will find `int main()`. Replace the line `print("Hello World\n\r");` with your own code. Test it on the board by uploading it to the FPGA. Review Lab 1 if you forgot how to do this.