Computation II - 5EIB0
Lab 2: Creating a Moore machine v1.0

Diederik Markus Wouter Schoenmakers

The goal of this lab is to create a finite state machine. The finite state machine will be a Moore machine. The machine would simulate a coffee machine, like the one in the slides. The exercises will guide you through making the Moore machine itself, a debouncing circuit for the buttons and sending messages via the USB cable to your laptop with the Zynq processor. The coffee machine module will interact with the processor through the AXI4-Lite bus and directly with the GPIO ports for the buttons and leds.
The coffe vending machine accept two coins: 5 cent and 10 cent. When the total is more than or equal to 15c a cup of coffee is prepared. The coffee machine does not return change but it does remember how much coins you have left. So if you throw in 10 cents twice, you will get a cup of coffee and you only have to input 10 cents to get a second cup of coffee.

Contents

1 Creating a project

First you need to setup a new project in Vivado, in the exactly the same way as you did in Lab 1.

Task 1. Create a Vivado project.

1. First create a new folder in the file explorer. It is recommended that you place this in the SharedWork folder. **Do not use spaces in the folder name.**
2. Open Vivado, you can use the menu in the top left of the screen.
3. To create a project select "File" $\rightarrow$ "Project" $\rightarrow$ "New". A pop-up will be presented on the screen, go ahead and click "Next".
4. You can now choose a project name and location. On default Vivado will create a folder with the project in your home folder, it is recommended that you change this to the folder you just made. Make sure you have selected "Create project subdirectory". Click "Next" to continue.
5. Vivado has several different types of projects available, you want to make sure that you have "RTL Project" selected as well as "Do not specify sources at this time", you will add them to your project later on. Once again click "Next" to proceed.
6. The next thing Vivado wants to know is what kind of hardware you will be using. In your case this is the PYNQ z2 board. In the "Boards" tab, select the "pynq-z2" board. Click "Next" and then "Finish".

2 The IP block

The finite state machine itself will be made as an IP block. The IP will several physical inputs and outputs as well as an AXI4Lite bus.

2.1 The logic

First you will make the IP block itself.

Task 2. Creating your own IP block

1. Create an IP block on the default location, named `coffee_moore`. The IP should have one AXI4Lite slave interface with at least 4 registers.
2. Add a design source named `coffee_moore.v` and make this the only active file (so disable `coffee_moore_v1_0.v` and `coffee_moore_v1_0_S00_AXI.v`). Set `coffee_moore.v` as the top module as well.

You have now created the IP block and added the module to the block. The next step is to implement the FSM logic. The FSM diagram is depicted in figure 1. The FSM has an output `coffee` and an input `coins`. The output `coffee` is used to let the user know that their coffee is ready. `coins` represents the coins that the user inserted into the coffee machine. In Fig. 1, every state ellipse has the following info (wcxC,y,[z]): w is the cents in the machine, x is the coffee output (+ for true, - for false), y is the output `state_display` and z is output `coffee`. On every state transition arrow the input `coins` is written down.
A second and third input are not displayed in the state diagram but are needed none the less. The input `reset` will be used to reset your FSM and `insert` to let you know when a user inserts a coin. The latter input is needed to control the speed of your FSM. A standard FSM will make a transition every clock cycle, however as it is impossible to press a button for only one cycle. A single press will be read multiple times and cause multiple transitions. This can be avoided by only making a transition when the button for `insert` is pressed. So you can debug your FSM in the simulator and on the board you also want an output port for the current state. For this you will create a port called `state_display`.

Task 3. Implementation of the Moore coffee machine.

1. Add the following ports to your `coffee_moore.v` module: 1 bit wide input port `clk`, 1 bit wide input port `insert`, 1 bit wide input port `reset`, 2 bits wide input port `coins`, 1 bit wide output port `coffee` and 3 bits wide output port `state_display`.
2. Write the implementation of the finite state machine as provided in Fig. 1. Make a transition when the `insert` button is pressed and reset the FSM when `reset` is high. Only update the output at the positive edge of the clock and only use edge triggering of the clock signal. To make sure that you only make a transition when the `insert` is **pressed**, check whether the current value is diffrent from its previous value. **Do not use: always @ (posedge insert).**

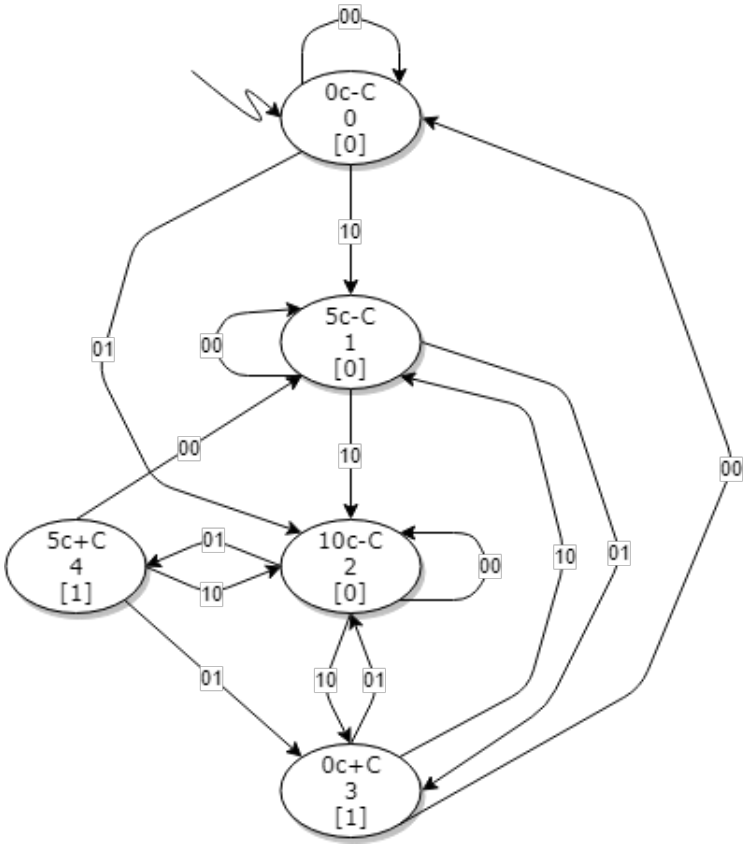

Figure 1: State diagram of the Moore coffee machine.

2.2 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 finite state machine.

Task 4. Write a testbench that generates all possible transitions in your finite state machine. It should also try the reset option in every state. Simulate you testbench to verify that your vending machine works correctly.

2.3 The AXI4Lite interface

Your finite state machine 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. Later on you will write a piece of code that sends this information to the terminal of the SDK.

Task 5. The template for the AXI4Lite bus is already present in your IP block. Since the input and output to your Moore machine is driving from outside your module, but not over the AXI4Lite bus, you have to add a few external ports to your IP block.

1. Enable the automatic generated files you disabled in section 2.
2. Somewhere in the bottom of the file `coffee_moore_v1_0_S00_AXI.v` there should be a line stating "// Add user logic here". Below this line make an instance of your module `coffee_moore.v`.
3. Add a 1 bit wide input wire called `reset` to both `coffee_moore_v1_0.v` and `coffee_moore_v1_0_S00_AXI.v` where it says "// Users to add ports here".
4. Connect the input wire you just created to the `reset` input of your FSM, such that the signal of `reset` is connected through to the output of your IP. **Keep the clocking in mind.**
5. Repeat previous two steps for the input signals `insert` and `coins` as well as for the output signals `coffee` and `state_display`.

Task 6. The clock of Moore machine is provided by AXI interface. Thus the clock of Moore machine is required to be connected to AXI clock. In addition, note that we want to be able to read the remaining Moore machine ports via AXI Bus. Thus, it is required to connect them to the AXI registers, also.

1. Connect `clk` to the clock of the `S_AXI_ACLK`.
2. Modify the generated always block that assigns to the slave registers to connect the external signal ports `reset`, `coins`, `coffee` and `state_display` to the lowest bits of `slv_reg0`, `slv_reg1`, `slv_reg2` and `slv_reg3` respectively. Connect `insert` to `slv_reg1` as well, use the first two bits for the signal `coins` and the third bit for `insert`. You do not need to be able to write to these registers.
3. Change the top module back to `coffee_moore_v1_0.v`.

Task 7. You have now connected all ports of `coffee_moore.v` to read and write registers. Edit the code for the AXI4Lite bus such that **all** ports are read only, under all circumstances. In this way writing to `slv_regX` registers from the AXI bus does not affect the FSM.

2.4 AXI4Lite Testbench

Since section 2.3 is similar to the previous lab assignment. If you are not sure about the correctness of your solution, you can adapt Lab1's testbench for the AXI interface and test it.
For this lab you should run synthesis instead. It is possible to write Verilog code that can be simulated but not synthesized. In this case Vivado knows what you want to do but can not build it.

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. The instructions are the same as Lab 1. When you are done switch back to the original project that you created in section 1;

3 Debounce module

Before making the block diagram of the board, we first need to make another Verilog module. You want to connect `insert` to a button however when buttons are pressed the internal contacts often bounce away from each other when they collide. This breaks the contact causing several spikes in the signal from the buttons, see figure 2. To be able to use the buttons correctly you first need to debounce them. This can be done with a capacitor parallel to the button or in with logic in the FPGA. Since there is no such capacitor on the board, debouncing should be implemented in the FPGA.

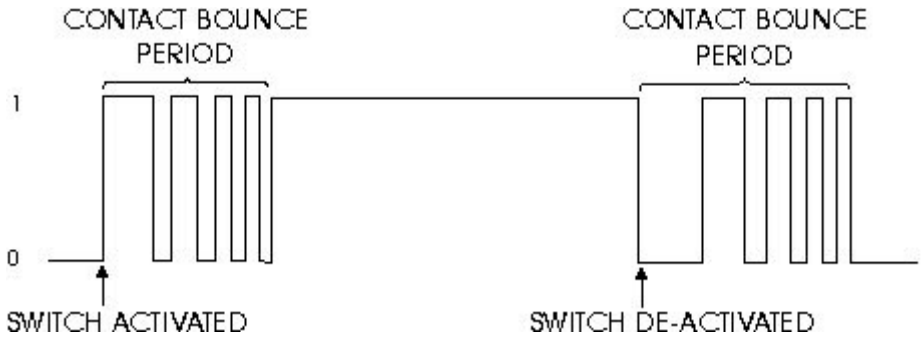

Figure 2: Waveform of switching a button.

Task 10. Writing a debounce module

1. Create a new source file "debounce.v".
2. Create the following inputs: 1 bit wide `clk`, 1 bit wide `reset` and a 1 bit wide `button_in`.
3. Create the following output: 1 bit wide `button_out`.
4. Write the debouncing logic. It does not matter if the signal is slightly delayed. Use `reset` to reset your module.

Hint: For the debounce module count for how many cycles the signal is stable. Use a 20 bit register for counting. Flip the state of the output, the cycle after the 20st bit is flipped. This will result in a counter of roughly 10 ms.

Note: If later on you are having problems with inconsistent transitions you might want to write a testbench for this module to make sure the buttons are properly debounced.

4 Testing on the board

For this lab you are going to skip simulating with the microblaze and go directly to programming the board however to do this you first need to create the block diagram and write some C code for the ZYNQ processor.

Task 11. Create a block diagram. The resulting structure should resemble figure 3.

1. Click on the left side of the screen under the top module "IP Integrator" on "Create block design". This will make a block design that you can use to connect the counter IP to the rest of the system. A new window is opened. Name the design `design_1`, and keep the rest of the settings untouched. Click "OK" to generate the block design. An empty window should be opened.
2. Click on the + to add IP blocks to the block design. Add your FSM (`coffee_moore_v1_0`) and the ZYNQ processor.
3. Make the ports `state_display`, `coffee` external. Select the wires and click on "Make External", make sure you have only selected the port and not the whole IP block.
4. Vivado will let you add Verilog modules to a block diagram as well as IP blocks. Right click on the module `debounce.v` and select "Add Module to Block Design".
5. The FSM block is reset by pushing a push button in the Pynq board. Thus, Connect the `reset` of FSM block to the output of a denounce module which will be connected to a push button as shown in Fig.3. Similarly connect `insert` and `coins` to push button and switches as shown in Fig.3.
6. Let Vivado automaticity setup and connect the ZYNQ processor. Click on "Run Block Automation" and "Run Connection Automation".
7. In the case that debounce modules reset were not connected to proper reset line by Vivado connection automation tool, correct it taking the below hint into account.
8. Generate a wrapper for the block diagram. Right click the block diagram in the sources view and select "Create HDL Wrapper". Let Vivado automatically update it.

Hint: There are two types of reset line, one type is active low reset, the other active high. Think about how you implemented yours in debounce module. Is it active high or active low ? To which kind should you connect your reset line? The active low reset line is most of the named something like `...resetn...` and the active high reset line is most of the time something like `...reset...`.

Hint: Use a `concat` module to merge coins[0] and coins[1] into a single wire.

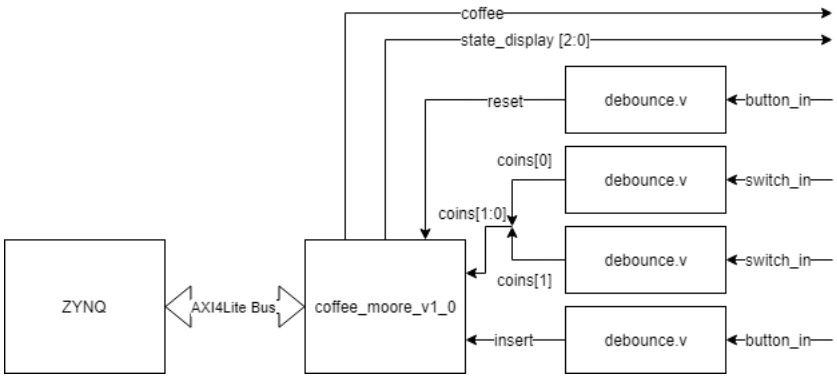

Figure 3: General structure of the block design.

The next step is to generate the bitstream but before you can do that you need to write the constraints file.

Task 12. Connecting the external ports to physical pins.

1. Create a new **constraint** source ("file" → "Add sources"), select "Add or create constraints". Name this file `constraints.xdc` and place it local to the project.
2. Edit the constraints template from Lab 1. The outputs `coffee` and `state_display` should be connected to the 4 leds. Connect the input of the debounce module which drives `coins` to the switches. Connect the inputs of the debounce modules which drive `insert` and `reset` to button 0 and button 1 respectively.

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.

Writing to the UART is fairly simple with the function `xil_printf(...)`, this function works in the same way as `printf`. The hello world project template comes with everything you need to be able to use `xil_printf(...)`.

Task 14. Launch the SDK ,create a project and test the bitstream.

1. Open the SDK by going to "File" → "Launch SDK".
2. In the newly opened SDK click on "File" → "New" → "Application Project". Set the name of your project to `fsm_code` and click on "Next", **not "Finish" like in Lab 1!**.
3. Select the hello world project and click on "Finish".
4. In the file `helloworld.c` you will find `int main()`. Delete the line `print("Hello World\n\r");`.
5. Program the FPGA and run file with the empty code. Your FSM should work on the leds right now.

Hint: If you want to split one register in two different values depended on the position of the bits, you can use bitwise operations. See `https://en.wikipedia.org/wiki/Bitwise_operations_in_C`.

Task 15. Write a program that continously reads the inputs and outputs of the Moore Machine and writes them to the UART. Write to the UART only once every second, you can use `usleep(..)` to do this. The output should be as following: `"State:..., Coffee:..., Reset:..., Insert:..., Coins:...\n"`. Test it on the board by uploading it to the FPGA and running the code. You can read the print statements in the SDK with the "SDK Terminal" on the bottom of the screen. Click on the green "+" and select `/dev/ttyUSB#`, leave the other settings at their default values.