CS232 Project 4: Programmable Lights

October 7th, 2024

Task: Light Show

The goal of this project was to design and implement a simple programmable light display using VHDL. The display operates as a state machine with an instruction set of eight commands, allowing us to manipulate an 8-bit light register (LR). This project was executed by creating two VHDL entities: lights.vhd, which manages the state machine, and lightrom.vhd, which serves as the read-only memory (ROM) that holds the program instructions. The system's state machine interacts with the ROM, fetching and executing a series of 16 instructions.

Design Description

The project consists of two main components:

- lightrom.vhd: This module implements the ROM, which holds the program to be executed. It takes a 4-bit address input representing the program counter (PC) and outputs a 3-bit instruction.
- lights.vhd: This entity defines the control circuit and state machine. It uses the instruction from the ROM and operates on the light register (LR), which holds an 8-bit value. The state machine cycles between fetching instructions and executing them, based on the provided instruction set.

The state machine operates with two main states:

- Fetch State (sFetch): In this state, the instruction register (IR) is loaded with the next instruction from the ROM, and the PC is incremented to move to the next instruction in the sequence. The system then transitions to the sExecute state.
- Execute State (sExecute): In this state, the fetched instruction is executed, updating the LR accordingly. Depending on the instruction, the LR may be shifted, rotated, inverted, incremented, or decremented. After the instruction is executed, the state machine returns to sFetch to process the next instruction.

The possible instructions and their effects on the LR are as follows:

- 000: Load 00000000 into LR.
- 001: Shift LR right by one position (left-fill with '0').
- 010: Shift LR left by one position (right-fill with '0').
- 011: Add 1 to LR.
- 100: Subtract 1 from LR.
- 101: Invert all bits in LR.
- 110: Rotate LR right (rightmost bit becomes leftmost bit).
- 111: Rotate LR left (leftmost bit becomes rightmost bit).

Internal Signals:

- PC (Program Counter): A 4-bit signal that keeps track of the current instruction address.
- IR (Instruction Register): A 3-bit register that holds the instruction to be executed.

- LR (Light Register): An 8-bit register that represents the output to the lights.
- ROMValue: The 3-bit output from the ROM, which holds the instruction for the current state.
- State: The control state of the system, either sFetch or sExecute.

The ROM data for testing purposes was pre-loaded with a basic set of instructions that manipulate the LR in various ways, allowing us to validate the functionality of the system.

Simulation(s):

Program 1: Given

Simulation: https://drive.google.com/file/d/1WvjE46TwMm5goG1Ivpp8bUJ-ZgBpO5Jo/view?usp=sharing

Program 2: Alternating Blink Sequence

This program creates a pattern that alternates between 01010101 and 10101010 on the LEDs, producing a visual "dance" effect. The operations applied to the Light Register (LR) gradually build the pattern, shift bits to the left, and toggle the pattern by inverting the bits.

Code:

```
 data <=

   "000" when addr = "0000" else -- Load "00000000" into LR

    "011" when addr = "0001" else -- Add 1 to LR, LR = 00000001

    "010" when addr = "0010" else -- Shift LR Left 1, LR = 00000010

    "010" when addr = "0011" else -- Shift LR Left 1, LR = 00000100

    "011" when addr = "0100" else -- Add 1 to LR, LR = 00000101

    "010" when addr = "0101" else -- Shift LR Left 1, LR = 00001010

    "010" when addr = "0110" else -- Shift LR Left 1, LR = 00010100

    "011" when addr = "0111" else -- Add 1 to LR, LR = 00010101

    "010" when addr = "1000" else -- Shift LR Left 1, LR = 00101010

    "010" when addr = "1001" else -- Shift LR Left 1, LR = 01010100

    "011" when addr = "1010" else -- Add 1 to LR, LR = 01010101

    "101" when addr = "1011" else -- Invert all bits, LR = 10101010 (dance)

    "101" when addr = "1100" else -- Invert all bits, LR = 01010101

    "101" when addr = "1101" else -- Invert all bits, LR = 10101010

    "101" when addr = "1110" else -- Invert all bits, LR = 01010101

    "101"; -- Invert all bits, LR = 01010101
```

Simulation: https://drive.google.com/file/d/1XzFrrabmFMFnn4uOcs-rSPY49GXg-LyF/view?usp=sharing

Program 3: Rolling Wave Pattern

This program creates a rolling wave pattern where the lit bits continuously rotate through the LR, giving the effect of a light wave moving across the display from right to left.

Code:

```
data <=
  "000" when addr = "0000" else -- Load "00000000" into LR
   "011" when addr = "0001" else -- Add 1 to LR, LR = 00000001
   "111" when addr = "0010" else -- Rotate left, LR = 00000010
   "111" when addr = "0011" else -- Rotate left, LR = 00000100
   "111" when addr = "0100" else -- Rotate left, LR = 00001000
   "111" when addr = "0101" else -- Rotate left, LR = 00010000
   "111" when addr = "0110" else -- Rotate left, LR = 00100000
   "111" when addr = "0111" else -- Rotate left, LR = 01000000
   "111" when addr = "1000" else -- Rotate left, LR = 10000000
   "111" when addr = "1001" else -- Rotate left, LR = 00000001
   "111" when addr = "1010" else -- Rotate left, LR = 00000010
   "111" when addr = "1011" else -- Rotate left, LR = 00000100
   "111" when addr = "1100" else -- Rotate left, LR = 00001000
   "111" when addr = "1101" else -- Rotate left, LR = 00010000
   "111" when addr = "1110" else -- Rotate left, LR = 00100000
   "111";                        -- Rotate left, LR = 01000000
```

Simulation: https://drive.google.com/file/d/149njnW_N-wgHKeXE4e3OmYsMipP7vO8e/view?usp=sharing

Explanation:

For testing the programmable light display on the DE0 board, I mapped the lights output signal to the on-board LEDs and used a physical button for the reset control. Using the Quartus II Pin Planner, I mapped the lightsig output signal (which controls the 8-bit light register) to the DE0's green LEDs (LEDG[0] to LEDG[7]). This way, the light register values were directly visualized on the board's LEDs, making it easy to see how the instructions manipulated the light patterns. For the reset functionality, I mapped one of the DE0 board's push buttons to the reset input of the circuit. Pressing the button manually triggered a reset, setting the program counter and light register to zero, allowing for easy restart of the sequence. I used the on-board clock signal (pin G21) to drive the clock input of the circuit. Since the clock runs at a high frequency, I added a counter to divide the clock signal, effectively slowing it down to a human-visible rate. This allowed me to observe each instruction execution as it manipulated the lights on the LEDs, ensuring that the program executed correctly step by step.

The following Extensions take place in 1 project: betterlights, and the extensions are showcased in the order listed below:

Simulation of all 3 Extensions:
https://drive.google.com/file/d/1xfbWVqvCBiyd62suOdcwoEENytRxm3JT/view?usp=sharing

Extension 1: Longer Program

Using the third program originally made for the base project as a base, I expanded the program to use 5-bit addresses, allowing for 32 instructions. This additional instruction space enabled the creation of a wave-like pattern in which: 1 light moves from left to right, 2 lights move in sync, 3 lights move together, and finally, all the lights flash at the end.

Extension 2: Hold/Free Button

A new button was added as a hold/free (pause) control. Because the 3 buttons on the DE0 board were being occupied as reset, speed up and slow down buttons, I resolved to make it a pause switch. When the switch is pressed, the program execution halts, effectively freezing the display. Releasing the button resumes the program where it left off.

Extension 3: Adjustable Speed

To avoid constantly having to recompile to change the speed of the program, the speed of the clock was made adjustable with two buttons: one to speed up the clock and another to slow it down. Debouncing was accounted for to prevent issues with button press signals inputting more than it should. This allows the user to control how fast the light sequence plays out.

Acknowledgements

As always, the TAs and Professor Li helped troubleshoot the base code! Since no one's around for fall break, I worked on the extensions alone this time!