CS232 Project 5: Programmable Lights 2

October 22nd, 2024

Task: Light Show 2

This project extends the fundamentals established from the previous project, but instead of only having operations, the ROM now takes into account conditional and unconditional branches, extending the duration of simulations without actually adding more bits.

Design Description

The top-level design of this project consists of three major components: pld2, pldrom, and pldbench. Each of these VHDL files interacts to create a programmable light display that follows specific instructions stored in ROM. Here's how the components work together:

1. pld2.vhd (Main Controller): This file acts as the central processing unit (CPU) of the programmable light display system. It fetches instructions from the ROM (defined in pldrom.vhd), decodes them, and performs the appropriate operations. The states of this controller include fetching an instruction (sFetch), determining the action to take (sExecute1), and executing the action (sExecute2).
    a. Key elements include:
        i. Instruction Register (IR): Holds the current instruction to be executed.
        ii. Program Counter (PC): Determines the current address in the ROM to fetch the next instruction.
        iii. Accumulator (ACC): An 8-bit register used to store and manipulate data.
        iv. Light Register (LR): An 8-bit register that controls the programmable lights.
        v. ROMValue: The 10-bit value fetched from the ROM.
        vi. Clocking and Reset: Ensures the program executes synchronously and can reset to its initial state.
    b. Instructions are fetched and executed in cycles. In sFetch, the instruction is loaded from ROM, and in sExecute1 and sExecute2, the actual data operations and branching are carried out.
2. pldrom.vhd (ROM Storage): This file defines the read-only memory (ROM) that stores the machine language program. The ROM has a 4-bit address input (addr) and a 10-bit output (data), which corresponds to the instruction fetched by pld2.
    a. The stored program in ROM controls how the lights behave. It may include moving values between registers, performing binary operations, or branching conditionally or unconditionally. For instance, the provided program moves values like 16 to the light register, performs shifts and arithmetic operations, and controls when to branch to different parts of the program.
    b. Each instruction in ROM is 10 bits, where specific fields encode the operation type, source, destination, and immediate values. For example:
        i. Move operations transfer data between registers.

ii. Binary operations include shifts, rotations, and logical operations (XOR, AND).
iii. Branch instructions alter the program flow, allowing for loops and conditional execution.

3. pldbench.vhd (Testbench): This file provides a test environment for simulating the functionality of pld2 and pldrom. It defines the input clock, reset signals, and monitors the behavior of the programmable lights as the program runs. The testbench verifies that the design behaves as expected by feeding the correct inputs and checking the output registers (ACC, LR) for expected values after each instruction is executed.

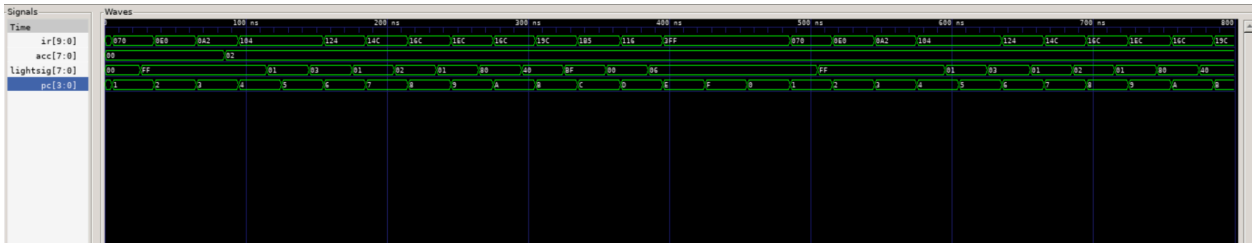
Component Interactions:

- The pld2.vhd fetches the program instructions from pldrom.vhd by sending the current program counter (PC) as the address to the ROM. The ROM responds with a 10-bit instruction.
- Depending on the type of instruction fetched, pld2.vhd executes data movement, binary operations, or branching. For example, if an instruction tells the system to move a value to the light register (LR), the light pattern will change accordingly.
- pldbench.vhd provides simulated input signals and checks if the lights behave as programmed, ensuring the design works before deploying it to real hardware.

GTKWaves:

Test 1:

Test 2:



Simulation(s):

Program 1: 16 down to 0, flash all 1s and 0s (Given)

Simulation: https://drive.google.com/file/d/1hhTuD_6WN41LocfRksmHDUJH4ihs515u/view?usp=sharing

Program 2: Randomness

This program utilizes all 8 LEDS by counting from 0 to 255, then flashing the first 4 and last 4 bits consecutively 8 times, followed by a final swipe of all LEDs right, to which it restarts.

Simulation: https://drive.google.com/file/d/1JMOcuL6IwZr6aHw9GSuJoT1f4dRrYsvi/view?usp=sharing

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

Simulation of these Extensions:

https://drive.google.com/file/u/1/d/1ScUB3WunGbnQglRBc-IlBtqa6e_iGOt2/view?usp=sharing

Extension 1: Hold/Free Button

(This was taken from the previous project) 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 2: Adjustable Speed

(This was taken from the previous project) 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.

Extension 3: Simple Assembler (converter.py)

For this project, I developed a simple Python-based assembler that translates human-readable instructions into machine code, specifically for the pldrom VHDL architecture. The purpose of this assembler is to streamline the process of writing programs that interact with the LEDs on a DE0 board by generating the appropriate machine instructions. This allows for high-level instructions like "MOVE," "ADD," and "XOR" to be converted into a binary format that is compatible with the PLD ROM.

The assembler converts human-readable operations, such as register moves, binary operations, and branches, into 10-bit machine code instructions, adhering to the provided instruction set architecture.

Opcode	Format	Description
00	[C1 C0] [Dest1 Dest0] [Src1 Src0] [Val3 Val2 Val1 Val0]	Move from SRC to DEST
	Dest10: 00 = ACC, 01 = LR, 10 = ACC low 4 bits, 11 = ACC high 4 bits	
	Src10: 00 = ACC, 01 = LR, 10 = IR low 4 bits sign extended, 11 = all 1s	
01	[C1 C0] [Op2 Op1 Op0] [Src1 Src0] [Dest0] [Val1 Val0]	Binary operator DEST = DEST op SRC
	Op210: 000 = add, 001 = sub, 010 = shift left, 011 = shift right maintain sign bit	
	Op210: 100 = xor, 101 = and, 110 = rotate left, 111 = rotate right	
	Src10: 00 = ACC, 01 = LR, 10 = IR low 2 bits sign extended, 11 = all 1s	
	Dest0: 0 = ACC, 1 = LR	
10	[C1 C0] [U3 U2 U1 U0] [Addr3 Addr2 Addr1 Addr0]	Branch to ADDR
11	[C1 C0] [Src0] [U2 U1 U0] [Addr3 Addr2 Addr1 Addr0]	Branch to ADDR if SRC is 0
	Src0: 0 = ACC, 1 = LR	

This was achieved by defining dictionaries that map high-level commands (MOVE, ADD, BRANCH) to their corresponding opcode, destination, source, and operand representations. The assembler reads a list of high-level instructions, processes each one by splitting it into components, and generates the corresponding 10-bit binary machine code.

```
# Define opcodes and instruction mappings
opcodes = {
    "MOVE": "00",       # Move instruction opcode
    "BINARY": "01",     # Binary operator instruction
    "BRANCH": "10",     # Unconditional branch
    "CONDITIONAL_BRANCH": "11" # Conditional branch
}

# Define destination and source mappings
registers = {
    "ACC": "00",          # ACC register
    "LR": "01",           # Light register (LR)
    "ACC_LOW": "10",      # ACC low 4 bits
    "ACC_HIGH": "11"      # ACC high 4 bits
}

srcs = {
    "ACC": "00",          # ACC as source
    "LR": "01",           # LR as source
    "IR_LOW": "10",       # IR low 4 bits, sign extended
    "ALL_ONES": "11"      # All 1s
}

binary_ops = {
    "ADD": "000",         # Add operation
    "SUB": "001",         # Subtract operation
    "SHL": "010",         # Shift left
    "SHR": "011",         # Shift right with sign bit
    "XOR": "100",         # XOR operation
    "AND": "101",         # AND operation
    "ROL": "110",         # Rotate left
    "ROR": "111"          # Rotate right
}
```

The assembler is divided into several key functions:

- Opcode Mapping: The assembler begins by mapping human-readable instructions to their respective opcodes. For instance, MOVE is mapped to "00", BINARY operations like ADD and XOR are

mapped to "01", and branching operations (BRANCH, CONDITIONAL_BRANCH) are mapped to "10" and "11" respectively.

- Source and Destination Mapping: Registers and values such as ACC, LR, and IR_LOW are mapped to binary equivalents (e.g., ACC = "00", LR = "01"). These mappings help in translating register operations into the corresponding bits used in the machine code.
- Instruction Parsing and Encoding: For each instruction, the assembler breaks it down into its components (e.g., operation type, source, destination, and value) and uses the predefined mappings to generate a 10-bit binary code for the instruction.
- Output Formatting: The bottom print statement outputs the machine code for each instruction in a format that can be directly integrated into the VHDL pldrom architecture, ensuring compatibility with the logic for LED control and display on the DE0 board.
- General formats are as follows:
    - MOVE SOURCE DESTINATION VALUE
    - OPERATION DESTINATION SOURCE VALUE
    - CONDITIONAL_BRANCH ADDR/LR ADDRESS
    - BRANCH ADDRESS

After assembling the instructions, the generated machine code was compared to an already verified solution set to verify the correctness of the operations. The sequence of machine instructions is designed to control the LED lights on the DE0 board, based on the sequence asked of from the first given program (count down from 16)

Original VHDL Code:

```
data <=
-- Load 0 into LR
"0011100001" when addr = "0000" else -- Move 16 (binary 00010000) to ACC (Upper 4 bits)

-- Initialize ACC to 16
"0001000000" when addr = "0001" else -- Move ACC to LR

-- Start countdown loop
"0100110001" when addr = "0010" else -- Subtract 1 from ACC (ACC = ACC - 1)
"0001000000" when addr = "0011" else -- Move ACC to LR (display ACC on LEDs)
"1100000110" when addr = "0100" else -- Branch to addr=0110 if ACC = 0 (start flashing loop)
"1000000010" when addr = "0101" else -- Unconditional branch to addr=0010(continue counting down)

-- Flashing loop - 8 flashes
"0010101000" when addr = "0110" else -- Move 8 (binary 00001000) to ACC (flash counter)
"0001110000" when addr = "0111" else -- Move all 1s (11111111) to LR
"0110011100" when addr = "1000" else -- XOR the LR with all 1s and write it back to the LR (bit inversion)
"0100110001" when addr = "1001" else -- Subtract 1 from ACC (ACC = ACC - 1)
"1100001100" when addr = "1010" else -- Branch to addr=1100 if ACC = 0 (end)
"1000000111" when addr = "1011" else -- Unconditional branch to addr=0111(continue flashing)

"1000000000" when addr = "1100" else -- Unconditional branch to addr=0000 (restart the whole process)
"1111111111"; -- garbage (terminate)
```

Translated to Words:

```
instructions = [
    "MOVE IR_LOW ACC_HIGH 1",
    "MOVE ACC LR",
    "SUB ACC IR_LOW 1",
    "MOVE ACC LR",
    "CONDITIONAL_BRANCH ACC 6",
    "BRANCH 2",
    "MOVE IR_LOW ACC 8",
    "MOVE ALL_ONES LR",
    "XOR LR ALL_ONES",
    "SUB ACC IR_LOW 1",
    "CONDITIONAL_BRANCH ACC 12",
    "BRANCH 7",
    "BRANCH 0",
]
```

Output:

```
data <=
"0011100001" when addr = "0000" else -- MOVE IR_LOW ACC_HIGH 1
"0001000000" when addr = "0001" else -- MOVE ACC LR
"0100110001" when addr = "0010" else -- SUB ACC IR_LOW 1
"0001000000" when addr = "0011" else -- MOVE ACC LR
"1100000110" when addr = "0100" else -- CONDITIONAL_BRANCH ACC 6
"1000000010" when addr = "0101" else -- BRANCH 2
"0010101000" when addr = "0110" else -- MOVE IR_LOW ACC_LOW 8
"0001110000" when addr = "0111" else -- MOVE ALL_ONES LR
"0110011100" when addr = "1000" else -- XOR LR ALL_ONES
"0100110001" when addr = "1001" else -- SUB ACC IR_LOW 1
"1100001100" when addr = "1010" else -- CONDITIONAL_BRANCH ACC 12
"1000000111" when addr = "1011" else -- BRANCH 7
"1000000000" when addr = "1100" else -- BRANCH 0
"1111111111";
end rtl;
```

This output demonstrates the assembler's ability to translate high-level instructions into the machine code format required by the pldrom. The assembler was able to handle complex operations such as conditional branches, binary operations, and register moves seamlessly.

Acknowledgements

As always, the TAs and Professor Li helped troubleshoot the base code!