LC-3

Antonio Iovine

7/12/24

Abstract

This document serves as both an overview and a tutorial on how to use the LC-3 model implemented in Verilog. It provides guidance on writing programs and correctly interpreting all associated files

Contents

List of Figures

List of Tables

1 Introduction

This side project, undertaken as part of my academic career at the University of Verona, introduced me to both the Verilog language and the LC-3 architecture. Inspired by this experience, I have undertaken the task of reimplementing the entire LC-3 processor from scratch using Verilog/SystemVerilog.

1.1 References

I drew inspiration and knowledge from several sources, including Appendix A from the University of Texas at Austin, the LC-3b ISA, Appendix C from Georgetown University, and the architectural lessons provided by the University of Verona

2 Verilog Code

The Verilog code I have written is organized into the following subdivisions:

- FSM.sv
- ALU.sv
- design.sv
- LogicNZP.sv
- Adder.sv
- Gate.sv
- RegFile.sv
- Ram.sv
- Mux.sv
- MuxC2.sv
- PcMux.sv
- SEXT11.sv
- SEXT9.sv
- SEXT6.sv
- SEXT5.sv
- ZEXT.sv
- Output.sv
- testbench.sv

2.1 FSM.sv

The Finite State Machine (FSM) is the core component of this project, functioning as a highly efficient coordinator. It generates all the necessary signals to control the gates and wires within the system. The FSM receives inputs from the Instruction Register (IR) and the clock, and it outputs a 29-bit signal that is distributed throughout the system to ensure proper coordination and operation.

The FSM is divided into two primary states: Fetch and Decode/Execute. The Decode/Execute state itself is further subdivided into 16 distinct substates, each responsible for determining a specific operation of the CPU. These states are implemented using a switch-case construct. Below is an example illustrating this implementation:

```
1'b0: begin // Fetch
        #delay gate <= pc; ld_mar <= on;
        #delay ld_mar <= off; MuxMDR <= off;  MuxMAR <= off;
            MuxPC <= 2'b00;
        #delay ld_mar <= off;
        #delay gate <= mdr;
        #delay read <= on;
        WMFC; //Wait for Memory Function to Complete
        #delay read <= off; ld_pc <= on;
        #delay ld_pc <= off; ld_mdr <= on;
        #delay ld_mdr <= off; ld_ir <= on;
        #delay ld_ir <= off;
      end
```

The Fetch state is one of the two primary states of the FSM and is further divided by clock delays. This division is useful as it allows sufficient time for signals to propagate and activate registers or other components. The Wait for Memory Completion (WMFC) task is employed to 'pause' the circuit until the memory returns a 'Ready' signal. In this context, ON and OFF are local parameters, where 1 represents 'ON' and 0 represents 'OFF'.

```
ADD: begin
        if (IR[5] == on) MuxSR2 <= on;
        else MuxSR2 <= off;
        #delay DR <= IR[11:9]; SR1 <= IR[8:6]; SR2 <= IR
            [2:0]; Alu <= 2'b00;
        #delay gate <= alu; ld_reg <= on; ld_cc <= on;
        #delay ld_reg <= off; ld_cc <= off;
     end
```

This is one of the 16 sub-states of the Decode/Execute state. It performs an addition operation, where the values of two registers are added together, and the result is stored in the register specified by the Destination Register (DR). The AND parameter is a local constant that corresponds to the opcode (operation code) associated with this particular state

2.2 ALU.sv

While the FSM serves as the heart of the project, the Arithmetic Logic Unit (ALU) functions as its brain. The ALU performs all arithmetic operations, such as addition, as well as logical operations like bitwise AND.

Below is the code illustrating the implementation of the ALU:

```
module ALU(
  input [15:0] data1,
  input [15:0] data2,
  input [1:0] operazione,
  output reg [15:0] risultato
);

    always @(*) begin
        case (operazione)
            2'b00: risultato = data1 + data2;
            2'b01: risultato = data1 & data2;
            2'b10: risultato = ~data1 + 1;
        endcase
    end

endmodule
```

2.3 design.sv

The design serves as the overarching structure of the system, integrating various modules such as the FSM, ALU, LogicNZP, and others. It initializes the Program Counter (PC) to the standard hexadecimal value 0x3000. The design also manages the signals from the FSM to update registers, including the Instruction Register (IR), Memory Address Register (MAR), Memory Data Register (MDR), and the flags N, Z, and P. Additionally, it controls the critical BUS wire that facilitates communication throughout the system.

The design also includes the wiring for all registers, ensuring proper connectivity and signal routing throughout the system.

2.4 LogicNZP.sv

The LogicNZP module is a crucial component of the LC-3 architecture. It monitors the data on the bus during operations such as AND, ADD, and other instructions like STORE. This module updates the condition codes—Negative (N), Zero (Z), and Positive (P)—based on the results of these operations. The LogicNZP unit plays a key role in comparison operations, helping to determine whether one number is greater than another and performing other related functions

2.5 Adder.sv

It is an adder utilized in the system to add two numbers, which can then be used to reference a specific location in memory. This adder can operate with either the Program Counter (PC) or a general-purpose register.

2.6 Gate.sv

It is controlled by a signal from the FSM, which determines whether the Program Counter (PC), ALU, Memory Data Register (MDR), or Memory Address Register (MAR) should drive the bus. Only one of these components can write to the bus at any given time, although all can read from it simultaneously.

2.7 RegFile.sv

This file contains the registers responsible for storing information. The registers are organized as R0 through R7, with R0 serving a special function as the output to the video display. The registers are designed to be compact yet efficient.

2.8 Ram.sv

The RAM in the LC-3 serves as the primary memory, storing all information including executable code, data, and characters. At initialization, the RAM is preloaded with three system call routines: input, output, and kill.

2.9 Mux.sv

This is a multiplexer used in various contexts within the system. It is a 32-bit multiplexer with a 1-bit control signal that determines which of the two 15-bit input wires should be selected for output.

2.10 MuxC2.sv

This is a multiplexer that selects among the SEXT6, SEXT9, and SEXT11 signals to determine which one should be passed as the second operand to the adder.

2.11 PcMux.sv

This is a multiplexer that selects whether the Program Counter (PC) should be updated with PC + 1, the result from the adder, or the information contained on the bus

2.12 SEXT Modules

The SEXT modules are sign extension units that take inputs of varying bit-widths and extend them to a larger bit-width while preserving the sign of the original value. Specifically, SEXT11 handles 11-bit inputs, SEXT9 handles 9-bit inputs, and SEXT6 handles 6-bit inputs. The SEXT5 module, unlike the others, is not connected to MUXC2.sv and performs sign extension on the immediate number encoded in the instruction provided by the Instruction Register (IR).

2.13 ZEXT.sv

The ZEXT module is a zero extension unit that extends the width of its input by padding with zeros, as opposed to sign extension which preserves the original sign.

2.14 Output.sv

This module retrieves the information from register R0 and outputs that value to the video display (console).

2.15 testbench.sv

This module provides the clock signal to all other modules and also monitors for errors. It issues a signal indicating completion if an issue arises or if the kill command is missing at the end of execution.

3 Scheme and Implementation of Wiring

This is the scheme and placement of the wiring as defined by the Vivado.

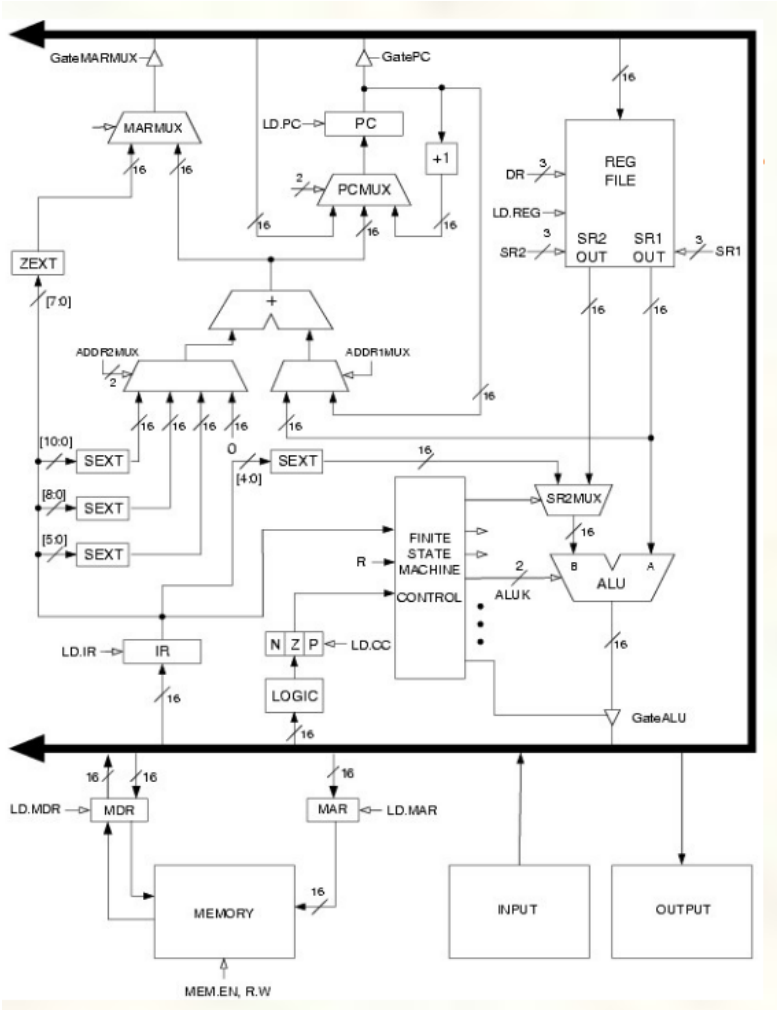

Figure 1: Caption for Scheme and Implementation of Wiring

4 EPWAVE

The EPWAVE file, generated by GTKWave, is used to test various operations of the LC-3. Although it does not cover all possible operations, it provides comprehensive testing for many of the LC-3's functionalities

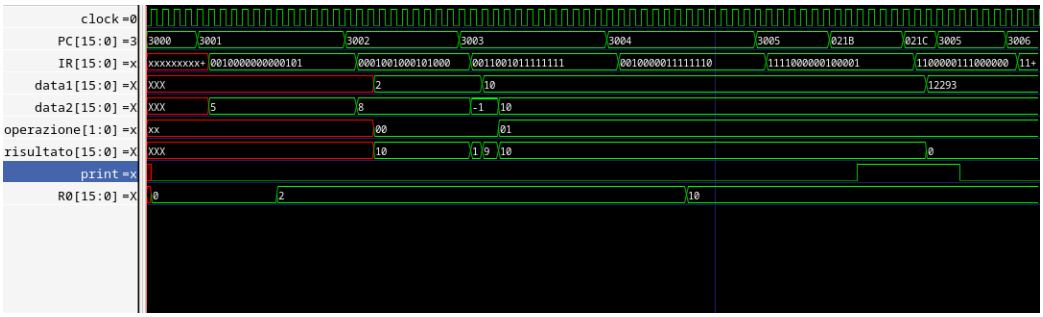

Figure 2: Caption for EPWAVE

In this figure, the clock provides the timing signal, while the program counter (PC) directs the memory on which instruction to fetch. The instruction register (IR) holds the current instruction being executed. Data1 and Data2 are the two operands used in the arithmetic logic unit (ALU) operation. The operation field specifies the arithmetic operation performed by the ALU, and the result (*risultato*) is the output of this operation. The *print* signal, when set to 1, outputs the content of register R0 to the console. Finally, R0 is one of the general-purpose registers.

```
//Istruzioni da fare partendo dalla cella di memoria HEX 3000-------------------------------------
memory[16'h3000] = 16'b0010_000_000000101; //LD R0, #2
memory[16'h3001] = 16'b0001_001_000_1_01000; //ADD R0, R0, #8
memory[16'h3002] = 16'b0011_001_011111111;   //ST R1, NUMBER
memory[16'h3003] = 16'b0010_000_011111110;   //LD R0, NUMBER
memory[16'h3004] = 16'b1111_0000_00100001; //TRAP output
memory[16'h3005] = 16'b1111_000000100101; //kill
memory[16'h3006] = 16'h0002;
```

Figure 3: Binary code in ram

This simple code performs the following steps:

1. Store the number 2 in R0.
2. Add 8 to R0 (which now contains 2) and store the result in R1.
3. Store the value of R1 in memory.
4. Load the value from memory into R0.
5. Call the Trap output (print the value in R0).
6. Terminate the simulation (Kill).

Note: The number 2 is stored in memory location hex 3006 and is used in the first instruction.

4.1 Tests and debugging

Tests and debugging are conducted by students from the University of Verona, including Antonio Iovine, Marzio Begolazzio, and Hitbel.

5 Instructions for Using the LC-3 Model

To use this LC-3 model, follow these steps:

1. **Load the Code:** Input the code in binary format into RAM, starting at the memory address `0x3000` and continuing forward. Ensure that the `kill` command is included at the end of the code.
2. **Compile and Run:** Use `iverilog` to compile and run the LC-3 model. This process can be facilitated by using the provided Makefile, which automates the compilation and execution of the LC-3.

6 Credits

6.1 Conclusion

In conclusion, this project has successfully implemented a detailed LC-3 model using Verilog/SystemVerilog, demonstrating the practical application

of digital design concepts. Through the design and integration of key components such as the FSM, ALU, and various registers, we have built a robust framework for understanding and executing LC-3 instructions.

The implementation includes comprehensive modules for arithmetic and logic operations, memory management, and input/output handling. The EPWAVE file, generated by GTKWave, has been instrumental in validating the functionality of the system through rigorous testing.

6.2 Special Thanks

We extend our heartfelt thanks to the following individuals and institutions:

- **Students:** Tommi Bimbato and Giovanni Martinelli, whose dedication and expertise were crucial in the testing and debugging phases of this project.
- **Professors and Advisors:** Michele Lora and Franco Fummi, for their invaluable advice and assistance throughout this project.
- **University of Verona:** For providing the academic environment and resources that supported the development and execution of this project.

This work highlights the importance of careful design and testing in digital systems and sets a solid foundation for future enhancements. Future work may involve expanding the functionality of the LC-3 model, optimizing performance, and integrating additional features. Continuous refinement and validation will contribute to the broader understanding and application of digital design principles.

We hope that this project serves as a valuable reference for students and professionals interested in digital system design and Verilog/SystemVerilog implementation.