8-Bit Microprocessor

ECE 2700 Final Project Report

Barnabas Kiss, James Lauritsen, Olumide Akeredolu, Rishi Tripathi
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
E-mails: kiss@oakland.edu, jlauritsen@oakland.edu, oakeredolu@oakland.edu, rishitripathi@oakland.edu

***Abstract*— The purpose of this project was to design and implement a microprocessor capable of the typical operations available for this type of device. The device components – an arithmetic logic unit (ALU), control circuit, registers, and others – process the data provided by the user. The device functionality was created using the software Vivado with VHDL. Microprocessors are the backbone of many real-life applications such as embedded systems and everyday electronics. Despite their complex appearance, microprocessors can be understood with proper explanation.**

I. Introduction

This project is about a microprocessor created on the Nexys Artix-7 FPGA board using VHDL. Microprocessors are the backbone of several real-life applications such as embedded systems, everyday electronic devices, and telecommunications. The inspiration behind this project comes from wanting to further develop knowledge of microprocessors and to understand the complexity behind their functionality. Thus, the scope of this project is oriented around a set of functions and arithmetic operations that together form an instruction set. An 8-bit input value and an 8-bit command – an instruction for the microprocessor inputted via switches – are provided by the user. After the user provides the number and command using the switches, a button on the FPGA board is pressed to execute the operation. We based our microprocessor on a previous implementation, and then added several upgrades including a higher number of instructions, more general registers, better output handling, and more.

The components of the microprocessor are the primary guides for understanding how the device works. They are individually easy to comprehend. These components are the inputs, the bus multiplexor, registers, the ALU, the control circuit, and the outputs of the circuit which are further described in the following section. Almost all of the components and their functionality have been taught in the ECE 2700 lectures; however, combining these functionalities into one larger project required further learning beyond the knowledge obtained from the lecture.

II. Methodology

A. Control Circuit

The control circuit is the heart of the microprocessor. It takes the incoming instructions and performs them according to a finite state machine. For this microprocessor, instructions are 8 bits wide. The first 4 bits are the operation code (opcode), while the rest are split into two 2 bit "parameters." These parameters can refer to a register (00 is Register 1, 01 is Register 2, and so on) or a 2-bit value.

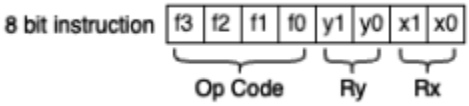


Figure 1: Operation Code and Register Parameters

With 4-bit opcodes, the CPU can support up to 16 unique operations. To maximize the usability of this machine, we targeted the most widely used and helpful operations. The operations and their functions are listed next.

Table 1: Machine Code Instructions

Code	Instruction	Description
0000	MOVE	Move contents of Ry into Rx
0001	LOAD	Move contents of input switches to Rx[1]
0010	INCR	Increments value in Rx by Ry+1[2]
0011	DECR	Decrements value in Rx by Ry+1
0100	ADD	Sets the value of ALU output to Rx+Ry
0101	SUB	Sets the value of ALU output to Rx-Ry
0110	COMP	Sets value of ALU output to not(Rx)
0111	MALU	Moves ALU output to Rx
1000	SHR	Shifts Rx to the right by Ry+1 spaces
1001	SHL	Shifts Rx to the left by Ry+1 spaces
1010	AND	Sets the value of ALU output to Rx AND Ry
1011	OR	Sets value of ALU output to Rx OR Ry
1100	XOR	Sets the value of ALU output to Rx XOR Ry
1101	MUL	Sets value of ALU output to bottom 8

[1] In some instructions, the parameter Ry is ignored.
[2] We add 1 to the parameter to maximize the usability of an operation where "00" as the parameter makes no impact.

		bits of Rx*Ry
1110	MULH	Sets the value of ALU output to the top 8 bits of Rx*Ry
1111	DISP	Sets value of output register to Rx, displaying the result on the 7 segment displays

When the *Begin* line is brought high while the CPU is in its waiting state, an instruction register captures the instruction and the FSM begins executing that instruction. As an example, the FSM diagram for the LOAD operation is given below.

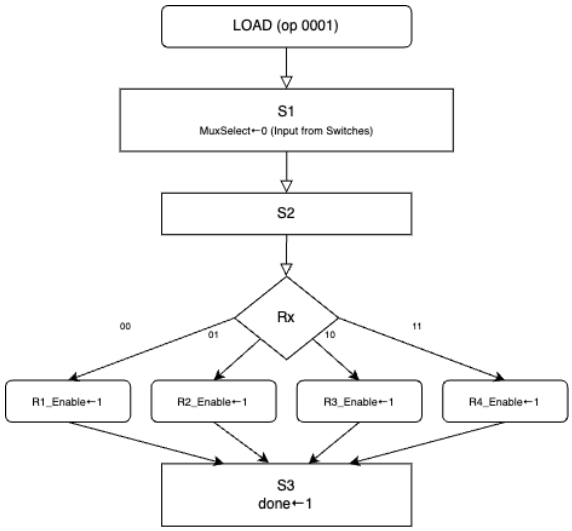


Figure 2: Load Operation FSM Diagram

Other operations are also performed in a similar manner and with the same methodology. For brevity, a full state machine diagram has been omitted, as it contains significant amounts of repeated/similar instructions. However, the logic is fairly straightforward and made as readable as possible in the control unit VHDL file.

B. Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is a component in the microprocessor that handles the logical and mathematical computations necessary for the microprocessor to complete an instruction set. There is a register, ALU Register A, which handles the first input to the ALU. This register ensures that two different numbers can be placed at the inputs of the ALU to execute a mathematical operation. Without the presence of the register and using the bus directly as the input for both inputs, the microprocessor would only be capable of computing operations for the same number in each input port – even a simple operation such as 1+2 could not be performed since the bus would only be able to provide one of these values at a time. Additionally, the ALU output register allows for storage of the ALU output and separation between the ALU and Main MUX. With these registers in place, the ALU accepts two 8-bit unsigned inputs and computes all operations specified in Figure 3. The ALU continuously updates the output of each operation, but these outputs never reach the Main MUX unless the ALU Output Register is enabled to capture the data. Additionally, the signal 'op' instructs the ALU which operation to output to the ALU Output Register by acting as a select inside the ALU block. Given that the number of instructions does not use the full 4 bits, it is sensible to implement the select in the top ALU design rather than in a MUX sub-component. The operations available are displayed in Figure 1 below.

IR Code	Function
0000	A + B
0001	|A - B|
0010	Shift A Left B Times
0011	Shift A Right B Times
0100	A AND B
0101	A OR B
0110	A XOR B
0111	NOT(A)
1000	Top 8 Bits of A * B
1001	Bottom 8 Bits of A * B
1010 to 1111	No Operation

Figure 3: Available ALU operations

This figure shows the instruction set of the microprocessor.

The ALU instruction set defined in Figure 3 is essential for the functionality of the microprocessor. Each operation was implemented uniquely using circuits developed in the laboratory and course lectures. The first operation "A+B" was implemented using an 8-bit adder composed of 8 serially connected full adders. The second operation "|A-B|" requires a sign extension and two 8-bit adders as shown in Figure 2.

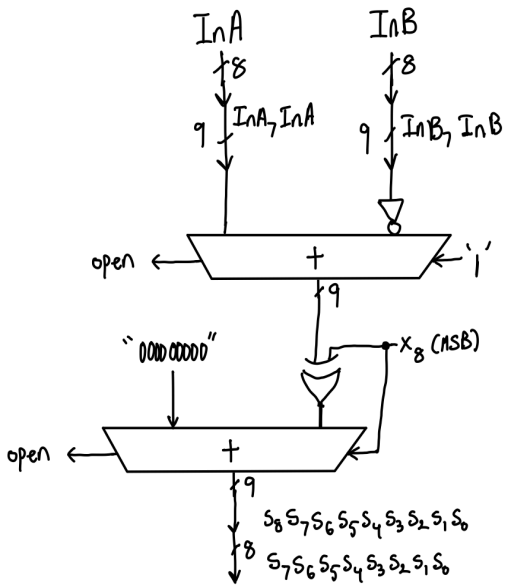


Figure 2: |A-B| Operation Circuit

This figure shows the combinational circuit required for the second operation |A-B|.

The shifting operation was implemented such that shift registers and a clock signal would not be necessary for shifting the value A by B to the left or right; instead, a shift bit is input to the shift subcomponent in a specific direction – '0' for left, '1' for right. The next several operations are purely simple combinational circuits, such as A AND B, A OR B, and NOT(A). The final operation is the unsigned multiplication of A and B; the maximum width that can come from A * B is 16 bits, so there is an operation for viewing the top 8 bits output from the multiplication and there is an operation for viewing the bottom 8 bits output from the multiplication so the whole result can be seen. Overall, the ALU could be expanded to have a full 16-bit IR code set for further functionality, however, they would be difficult to use as the CPU itself can't support more instructions.

C. Bus Multiplexer (MUX)

The multiplexer is a component that allows multiple input signals to be routed to a single output line based on the control signal. It acts as a digital switch that enables the microprocessor to select one of many inputs for processing or transmission, using a minimal number of lines or pins. In a microprocessor, multiplexers can be used for various things such as data selections, bus sharing, address decoding, or control signal routing. We will be using a 6-to-1 MUX in conjunction with various other components to facilitate the bus on our microprocessor.

D. Registers

Registers in microprocessors are high-speed storage locations used to hold data temporarily during processing. They are essential for the operations of the microprocessor in that they facilitate the storage and transfer of data between different parts of the processor, such as the ALU, control unit, and memory. There are various ways registers can be used such as storing operands and intermediate results, program counters, instruction registers, general purpose, stack pointers and shift registers to name a few. We will be using general registers to manage the storage of our microprocessor.

III. Experimental Setup

To verify that the microprocessor was capable of executing the instruction sets provided to it, a behavioral simulation using a test bench to check the functionality of the microprocessor was necessary. The unit under test (UUT) was the top file with all the individual components – control circuit, ALU, main MUX, registers, and 7-segment decoder – connected in the top design file itself.

There were many layers of complexity involving the control circuit and its FSM; the code for the microprocessor was frequently synthesized using Vivado's built-in 'Synthesis' function to find simple syntax errors such as mismatching logic vector assignments; additionally, warnings were closely analyzed.

We quickly found it unrealistic to write complex test benches in VHDL. As the process became more tedious, we worked on automating it. We created a Python-based "compiler" that took text input and generated machine code as well as a test bench. The text input was based on basic assembly, although it was significantly more limiting. For example, the compiler could not generate loops or do conditional logic. However, as these tasks were not within the scope of this project, we did think that this was a shortcoming.

This compiler allowed us to create and run more complex programs. We demonstrated this ability by having the processor calculate the highest possible Fibonacci number which could fit within 8 unsigned bits. Upon running the Python program to generate the test bench, we could simply copy the file into Vivado and run a normal simulation. We could then view the results by inspecting the value of the output register. The code for the compiler, as well as the code for the Fibonacci program, can be found in the Compiler.zip file submitted alongside this report. The output of the testbench can be found in the Appendix.

To demonstrate the functionality of the microprocessor, a simple operation was completed as shown in the behavioral simulation snapshots available in the Appendix. The microprocessor behaved as expected. Additionally, a test bench was created to simulate calculating the 13th Fibonacci number which is the largest bit-size number this microprocessor can compute; the result was expected to be 233 which matches its final result.

IV. Results

The results of the test bench and behavior simulation indicate the full functionality of the microprocessor – these results can be viewed in the Appendix. The provided inputs are loaded into the system onto a register and the system is capable of executing the instructions provided to it as shown by the constant LED outputs CA-CG and AN[7:0] – the outputs to the two 7-segment displays used.

To show the physical implementation of the microprocessor working, a simple operation of 4+4 was performed on an Artix-7 100T board. The following instructions were given to the board and executed by pressing a push button on the board to begin the instruction. The instructions are shown below:

- LOAD 0x4 R1
- LOAD 0x4 R2
- ADD R1 R2
- MALU R2
- DISP R2

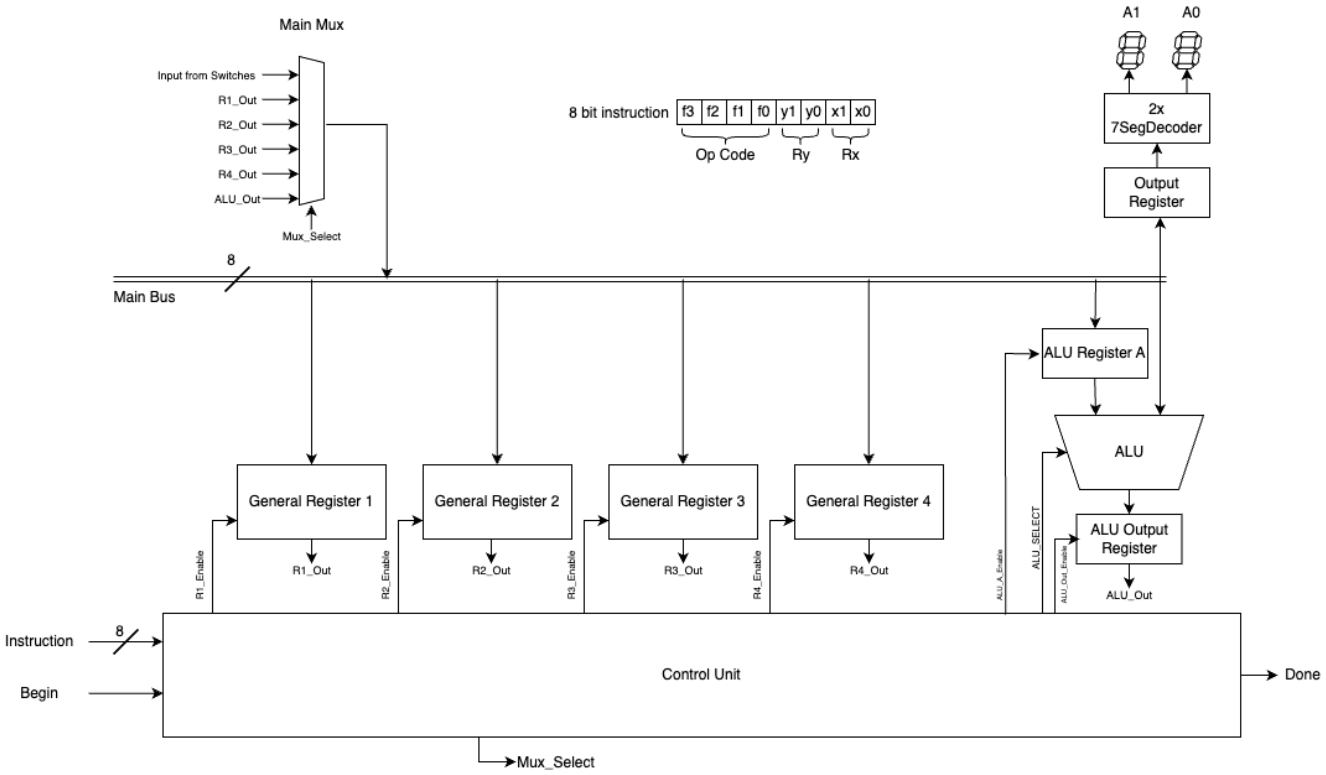
This operation results in an output of 8 on the 7-segment display, as shown in the video below.
https://drive.google.com/file/d/1Qg_sPso8u_3EUJRID-ZDKCj5DVGtO3fQ/view?usp=drive_link

Conclusions

Microprocessors are in many electronics around us, from cars to computers and cell phones. The ability to compute calculations faster than any human ever could have been one of the many cornerstones in the advancement of our everyday technology. The project's main focus is to build a simple processor to do some basic arithmetic. This didn't get accomplished without its fair share of headaches, however. One problem that appeared was the registers not being consistent in holding on to the data that was given to them. Another issue that arose was the display constantly showing zeros for everything. After extensive debugging, we are confident that some of our issues can be attributed to board errors. However, through perseverance, our goals were met, and overall are satisfied with our results. We were able to demonstrate that our microprocessor worked on our board. We also were able to show the ability of the microprocessor to run more complex programs, such as the Fibonacci calculation. Microprocessors are only going to get much more powerful in the future and this project is a great foundation for using what was learned in class and elevating it, to grasp the concept of how to create a CPU with digital logic design.

Appendix

1. Behavioral Simulation Snapshots, inc. Fibonacci sequence
https://drive.google.com/drive/folders/1D20k-mf32_3GCoJEtrjzXFerjtakaZog?usp=sharing



References

[1] D. Llamocca, “Laboratory 1, ECE-4710/5710.” Electrical and Computer Engineering Department, Oakland University, 2024

[2] [2] D. Llamocca, Reconfigurable Computing Research Laboratory, https://www.secs.oakland.edu/~llamocca/index.html (accessed Oct. 1, 2024).

[3] J. J. Jensen, “How to create a clocked process in VHDL,” VHDLwhiz, https://vhdlwhiz.com/clocked-process/ (accessed Nov. 20, 2024).

[4] In One Lesson, “How a CPU Works,” YouTube, https://www.youtube.com/watch?v=cNN_tTXABUA (accessed Nov. 2, 2024).