Microprocessor Design

List of Authors (Connor Goetz, Matthew Hait, Ethan Postma)
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: connorgoetz@oakland.edu, mhait@oakland.edu, ethanpostma@oakland.edu

***Abstract*—This project's goal was to create a functioning microprocessor design utilizing a VHDL programmed Xilinx FPGA trainer board. In order to accomplish this, a 16-bit architecture was chosen. Also included in the design will be the ability to load machine code programs into the processor via text file, as well as an accessible data stack. The processor has the ability to understand and execute basic computer programs and functions.**

I. Introduction

The microprocessor constructed consists of an ALU, register array, program counter, and stack. An instruction set was developed alongside the microprocessor to tailor the limited instruction sets to the processor hardware. The machine code used by the microprocessor is 16 bits in length as well as the register and stack width. This microprocessor incorporates components covered in Computer Hardware Design such as stack memory, FPGA ram blocks, parametric VHDL, and generation of memory from text files [2].

II. Methodology

The final circuit design of the created microprocessor circuit is shown here:

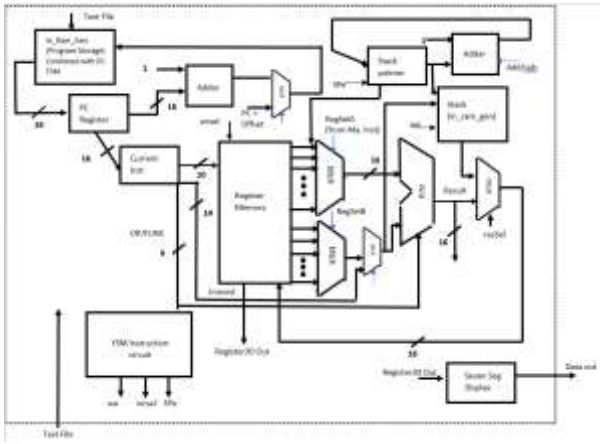

Fig. 1 Final circuit Design'

As demonstrated by the diagram, the creation of this microprocessor required several different components and design considerations. The following is the methodology related to each component and design consideration.

A. Machine Language Creation

When considering the creation of this new microprocessor, one of the first considerations was what type of instructions it should process. With instructions limited to an agreed upon 16-bits, it was decided that the processor should be able to handle basic logic and arithmetic instructions, with the addition of branching and jumping. This addition allows for more complex machine coding techniques such as the use of loops and conditionals.

With these desired deliverables, the machine code was broken down into four basic categories: logic instructions, arithmetic calculations, program movement/data memory manipulation, and immediate value loading. With four basic categories, a simple logarithmic calculation yields that two bits are necessary for indexing instruction. Therefore, the first two bits of each machine code instruction determines its category. The following documentation lists each category, its corresponding instructions, as well as their machine language instruction setup.

Logic Instructions:

OP CODE	Destination/ Source A	Source B	Unused	Function
0 0	_ _ _ _ _	_ _ _ _ _	X	_ _ _

Function	Outcome
000	A = Not A (Bitwise)
001	B = Not B (Bitwise)
010	A = A and B (Bitwise)
011	A = A or B (Bitwise)
100	A = A nand B (Bitwise)
101	A = A nor B (Bitwise)
110	A = A xor B (Bitwise)
111	A = A xnor B (Bitwise)

Fig. 2 Machine Language: Logic

Arithmetic Instructions:

OP CODE	Destination/ Source A	Source B	Function
0 1	_ _ _ _ _	_ _ _ _ _	_ _ _ _

Function	Outcome
0000	A = A + 1
0001	B = B + 1
0010	A = A - 1
0011	B = B - 1
0100	A = A + B (Signed)
0101	A = A – B (Signed)
0111	B = A
1 DIR _ N _ _	A = A shifted by N in DIR Direction, '0' = right, '1' = left (Signed)

Fig. 3 Machine Language: Arithmetic

Stack/Movement Instructions:

OP CODE	Offset Bits 7 to 2	Source B	Offset Bits 1 to 0	Function
1 0	_ _ _ _ _ _	_ _ _ _ _	_ _	_ _

Function	Outcome
0 0	PC = PC + Offset (Signed)
1 0	PC = PC + Offset (Signed) if Source B = 0
1 1	PC = PC + Offset (Signed) if Source B < 0

Fig. 4 Machine Language: Movement

OP CODE	Unused	Source B	Stack OP	Function
1 0	X X X X X	_ _ _ _ _	_ _	0 1

Stack OP	Outcome
0 0	SP = SP + 1
0 1	SP = SP - 1
1 0	Store B on Stack, SP + 1
1 1	Pull data from stack onto B, SP - 1

Fig. 5 Machine Language: Data Memory/Stack

Immediate Value Instructions:

OP CODE	Signed 14-bit Immediate Value
1 1	_ _ _ _ _ _ _ _ _ _ _ _ _ _

Fig. 6 Machine Language: Immediate

For the immediate category, it should be noted that some design compromises were made to maximize the size of values that could be loaded. As shown in the machine instruction above, there is no room for a destination register. When executing this instruction, the destination register is always 31, or address 1F in hex. This means that an immediate value must first be loaded, and then moved into its true register destination before operations begin. Once again, this compromise was made to maximize immediate value range.

B. Arithmetic Logic Unit Design

Once it was determined what calculations and logic instructions would be needed based upon the machine language, the next integral part of the microprocessor was to design its arithmetic logic unit (ALU). For the project, it was vital to ensure that the ALU made quick calculations with enough functionality to be comparable to a industry ALU [1]. Due to the nature of the machine language which operates utilizing four basic categories, the ALU was created to operate in a similar fashion. As inputs, the ALU requires to 16-bit data entries representing the values to be operated on. In addition, the ALU requires the first two bits of the machine language instruction, as well as its last four bits. By receiving this data directly from the machine input, the ALU does not require any FSM signals to complete tasks. This also means that the ALU is a purely combinational circuit.

Although the ALU only outputs one results, every possible logic and arithmetic results is combinatorial calculated each cycle. By using the machine code input to multiplex data inside of the ALU, the component can accurately produce the desired result of those calculated. This design has positives as well as drawbacks. From a timing perspective, this implementation style means that the ALU can operate combinatorial. Since each possible result is being calculated, it is not necessary for data to be latched at any point in the calculation process, meaning the output result is ready within one clock cycle. The primary drawback to this approach is the amount of required hardware. Since the circuit calculates each possible output, many instances of components such as adders are required. In turn, this utilizes more of the FPGA's resources.

On a final note regarding the ALU, by examining the its circuitry, one may find that besides handling arithmetic and logical calculations, it also emits necessary signals when it comes to branch determinations. For the two types of branch instructions, branch on zero and branch on negative, the ALU asynchronously calculates whether the register in question does in fact contain a negative, or zero value.

C. Register Controller and Data Memory

The register controller and data memory circuits have similar roles as it is necessary for them to latch data, but their implementations are vastly different. Starting with the register controller circuit, this circuit contains 31 registers, each with a capacity of 16 bits. Their address range for 0x01 to 0x1F. By utilizing two multiplexers, the circuit can determine which two registers are needed for any given operation based upon two 5-bit input bus lines whose sources come directly from the current input machine instruction. The circuit also has two output 16-bit data lines which travel towards the ALU. Finally, one input 16-bit data lines connects to every register so newly calculated data can be saved. This data's ability to be saved depends upon a 5-bit enable signal which emits from the FSM. This 5-bit signal determines which register is the destination for new data, and a decoder splits the enable signal into a solitary logic high for the given destination register. Finally regarding the data memory, it should be noted that although all 31 registers are accessible by the user, two are set aside for specific applications. As noted in the machine language section, register 31 (address 0x1F) is reserved for immediate value loading. Register 30 (address 0x1E) is set aside specifically for display data. Therefore, when a user requires data to be output to the seven segment displays, it must be copied onto register 30.

When deciding upon an approach for the data memory, several considerations had to be made. Like a lab performed in class, it would have been possible to again use a register-based approach. This was not chosen however due to its extreme hardware costs at scale as well as its very limited capacity. It was determined that utilizing the FPGA's built in blockRAM, a relatively large amount of space would in turn be available for storage. To use this however, two compromises had to be made. First, regarding timing, pulling data from this source type inherently requires two extra clock cycles, therefore costing some execution time.

Second, with such a large data space, it would be difficult to map each desired location within one 16-bit instruction. To solve this issue, and to simplify the end use design, a stack-like approach was utilized. Although not utilizing the reverse order system of a normal stack, the data memory address can only be altered by a single increase or decrease at a time. The inRAMgen circuit tested in class was used to implement this blockRAM data memory approach. It should also be noted that the blockRAM is also utilized by the program counter circuit to store a program. Due to this, the data memory begins at address 0x2000 to avoid the two data blocks overwriting each other.

D. Program Counter Circuit

The program counter consists of a 7-bit adder, finite state machine, and the inRAMgen block to handle indexing the ran address and jump functions. Internal to the finite state machine is a counter to index the ram address and adder to add a jump offset to the ram address. Keeping these two components internal to the finite state machine is a faster approach than incorporating physical components that would need to change in size as the amount of stored instructions changes. The finite state machine also reduces the delay of indexing and jumping down to one clock cycle.

The program counter takes advantage of using parametric VHDL to adjust signals and blocks. In synthesizing the block, the number of stored instructions is passed to the synthesizer. This is first used in the synthesis of inRAMgen by only allocating the minimum ram blocks needed to store the instructions. Taking the logarithm base two of the number of instructions, the signal width of the ram address is scaled to access all allocated ram blocks. Likewise, the counter and adder internal to the finite state machine also grow to accommodate all ram address.

Designing the program counter in this way makes the microprocessor flexible to a wider range of applications. By decreasing the quantity of stored instructions, excess ram blocks inside the FPGA are not allocated and may be used in the stack or other processes.

Whilst the processor functioned normally in the behavioral simulation, during post-process timing simulation it was discovered that the program counter was indexing ram address inconsistently. Inspecting the post-synthesis nets shoed that two ram address net signals were synthesized unconnected. Due to the time restraints of the project, a full analysis into the failure of the program counter could not be completed. The program counter was the switched to use a look up table loaded from a text file.

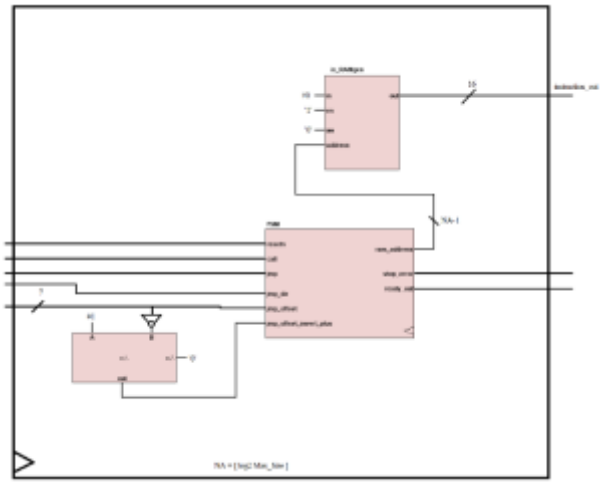

Fig. 7 Program Counter Circuit

1. E. Finite State Machines

This project utilizes two different FSMs to produce necessary outputs and results. The first FSM lies within the instruction entry/data path circuit. This refers to the portion of the microprocessor excluding the program counter circuit as well as the output data serializer. The second FSM emits necessary signals for the program counter circuit.

The first FSM controlling the instruction entry circuit emits necessary signals for multiplexing data channels, the data memory, the stack pointer, as well as the register memory. Due to the nature of the machine language and its corresponding four main categories, the FSM approach creates a simpler design than a combinatorial instruction decoder. Since the four main categories all contain similar instruction types, the FSM can emit signals based on the machine language input. While this requires some extra clock cycles, a large combinatorial circuit which emits outputs based upon every single operation and function code is not necessary. The following FSM flowchart depicts how this FSM moves through states dependent upon the current machine language instruction:

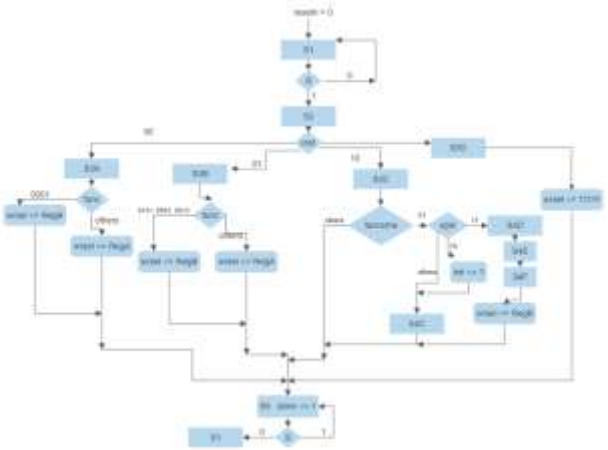

Fig. 8 FSM Flow Chart

The second FSM is utilized in the program counter. The FSM handles indexing the ram address to the stored instructions for each processor cycle. If the previous stored instruction was a branch or jump instruction, the FSM will also add the offset to the ram address to jump to that instruction. To prevent the scenario where the ram address exceeds the stored instructions and pulls random data from the ram blocks, the FSM checks if the current ram address is the last ram address and halts the processor if it is. Incorporating the ram counter in the FSM reduces the number of clock cycles to index that ram address and provide the processor with the next instruction.

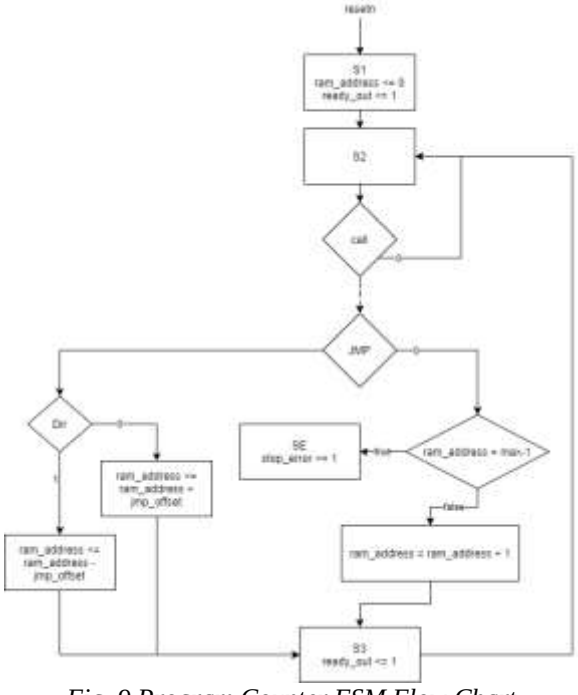

Fig. 9 Program Counter FSM Flow Chart

2. F. Data Output Serializer

The data output serializer utilized basic parametric code to let a dedicated register on the board, Register 29, display on the 7 segment display. A signal is output from the Register Controller to the Serializer with the value of Register 29 in it. The 16-bit signal is then broken down into four 4-bit signals for the four utilized 7-segment compartments. The figure below shows the inputs and decoding of the input signal in order to produce the correct output for the powered compartment of the 7-segment display.

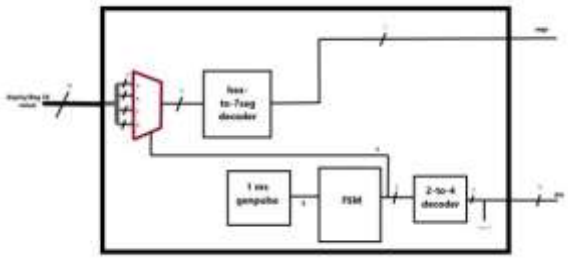

Fig. 10 Serializer Diagram

III. Experimental Setup

In order to test the microprocessor, small cases were designed to test individual parts like the program counter, register controller and the stack. The small cases and instructions helped to confirm that the separate blocks of the microprocessor all work as intended. To implement the design on the board, a counter was designed in machine code to increment the output register every second. Using a series of registers and jumps, the delay is calculated and set for implementation to give a one second counter on the 7 segment display. The instructions used for the program are in the figure below, as well as the description of what each instruction accomplishes.

```
4200;   R1 = R1 + 1                     PC: 000
43E7;   R29(output) = R1                PC: 001
C006;   IM = 6(larger for 1 sec delay)  PC: 002
7E37;   R3 = IM                         PC: 003
C005;   IM = 5                          PC: 004
7E27;   R2 = IM                         PC: 005
4402;   R2 = R2 - 1                     PC: 006
802A;   Branch to 009 if R2 = 0        PC: 007
BE38;   JMP to 006                      PC: 008
4602;   R3 = R3 - 1                     PC: 009
BA3A;   Branch to 000 if R3 = 0         PC: 00A
BC34;   JMP to 004                      PC: 00B
```

Fig. 11 Test Program Instructions

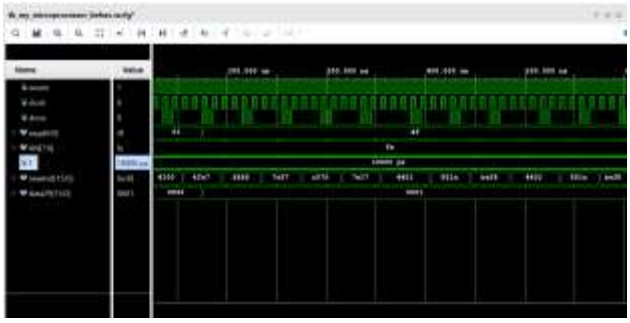

Fig. 12 Simulation of Previous Program

IV. Results

After performing both simulations and real world tests, the results of the microprocessor circuit are as expected. To utilize the circuit, a machine language program is uploaded to the FPGA board's block RAM. Based on the given instructions, the processor then makes necessary calculations. Results are then displayed via the onboard seven segment displays as expected.

V. Conclusions

Overall, our microprocessor utilizes a FPGA board to execute instructions and programs efficiently. The experimental results show that the microprocessor is capable of things similar to that of a standard simple microprocessor. We felt we met our project goal of creating a functional microprocessor with an accessible stack, a range of logical, arithmetic, and branching instructions, as well as being able to execute simple computer programs. We were also able to utilize our microprocessor's program storage to load programs from a text file, making the microprocessor practical in design.

VI. References

[1] Arithmetic Logic Units (ALU): An Introduction. [Online]. Available: https://arith-matic.com/notebook/arithmetic-logic-unit-introduction. [Accessed: 20-Apr-2021].

[2] Llamocca, Daniel. "Unit 6- Microprocessor Design" VHDL Coding for FPGAs. http://www.secs.oakland.edu/~llamocca/Courses/ECE4710/Notes%20-%20Unit%206.pdf