32-Bit Microprocessor

ECE4710 – Final Project

Lukas Popovic & Cameron Vogeli
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
lukaspopovic@oakland.edu, cmvogeli@oakland.edu

***Abstract*—To gain a better understanding of one of the world's most popular digital systems, this project involves the design of a 32-bit multi cycle microprocessor. Machine instructions may be loaded using either a text file or the UART protocol, and specified memory locations are uploaded back to the computer through UART. Using a versatile instruction set, an expandable memory interface, and functionality similar to RISC architectures today, this microprocessor can execute basic programs at the 32-bit level.**

I. Introduction

The design of the 32-bit PicoBlaze microprocessor was guided by the goal of creating a compact yet versatile processor with a robust instruction set. In the initial stages of the project, the team outlined several core arithmetic and logic operations commonly used in industry-standard processors. This included basic operations like addition and subtraction, as well as logical functions like AND, OR, and XOR. The instruction set was designed to be flexible, allowing for operations using either immediate values or register-to-register interactions. Immediate-based instructions were clearly distinguished with a unique "I" suffix. To enhance the processor's functionality, shift and rotate instructions were incorporated, supporting single-bit operations and simple multiplication and division techniques. These instructions used a consistent format, shifting or rotating one bit at a time with various options for what to shift in (like 0, 1, MSB, or LSB). The inclusion of these instructions, along with other essential operations, provided a comprehensive and versatile foundation for the processor's instruction set. Beyond arithmetic and logic instructions, the project also focused on branching, comparison, and memory-related operations. The branching mechanism employed conditional and unconditional jumps, with support for subroutine calls and returns. To accommodate these operations, the processor's program counter was designed to handle a wide range of addresses, allowing for flexible control flow. Additionally, the call stack could hold up to 32 addresses, ensuring efficient subroutine management. Memory storage and data transfer instructions were also included, albeit with some limitations due to time constraints. Data could be stored and retrieved through registers, and memory utilization was minimized to maintain a compact design. Despite these limitations, the instruction and data memory were designed with expansion in mind, allowing for future scalability.The implementation of the microprocessor also involved a robust experimental setup, including simulation and testing using Xilinx's Vivadosimulator. This ensured that each component functioned correctly before integrating them into the final design. Although the project faced challenges,particularly with the UART loader, the overall result was successful, yielding a functional 32-bit microprocessor with a versatile instruction set and ample room for future enhancements.

I. Methodology

A. Architecture Overview

	Instruction Type	Instruction	Opcode (binary)	Function (Decimal)	Description
Arithmetic	**R**	ADD rd, rs	011001	-	rd ← rd + rs
	IM	ADDI rd, imm	011000	-	rd ← rd + imm
	R	ADDC rd, rs	011011	-	rd ← rd + rs + C
	IM	ADDCI rd, imm	011010	-	rd ← rd + rs + C
	R	SUB rd, rs	011101	-	rd ← rd - rs
	IM	SUBI rd, imm	011100	-	rd ← rd - imm
	R	SUBC rd, rs	011111	-	rd ← rd - rs - C
	IM	SUBCI rd, imm	011110	-	rd ← rd – imm - C
Logic	**R**	AND rd, rs	001011	-	rd ← rd and rs, C ← 0
	IM	ANDI rd, imm	001010	-	rd ← rd and imm, C ← 0
	R	OR rd, rs	001101	-	rd ← rd or rs, C ← 0
	IM	ORI rd, imm	001100	-	rd ← rd or imm, C ← 0
	R	XOR rd, rs	001111	-	rd ← rd xor rs, C ← 0
	IM	XORI rd, imm	001110	-	rd ← rd xor imm, C ← 0

Table 2.1: Arithmetic and Logic Instructions

At the beginning of the project, it was decided that this processor at minimum should include most arithmetic and logic instructions commonly used in the industry[2]. As shown in the table above, addition and subtraction, each with and without carry, were included. And, or and xor instructions are also available. For each of these operations, the user may use either registers or immediate values for operands. For clarity, instructions that make use of immediate operands have an "I" suffix to distinguish from register type instructions.

	Instruction Type	Instruction	Opcode (binary)	Function (Decimal)	Description
Shift and Rotate	**SR**	RL rd	100000	0	rd ← rd[30..0] & rd[31], C ← rd[31]
	SR	RR rd		1	rd ← rd[0] & rd[31..1], C ← rd[0]
	SR	SL0 rd		2	rd ← rd[30..0] & '0', C ← rd[31]
	SR	SL1 rd		3	rd ← rd[30..0] & '1', C ← rd[31]
	SR	SLA rd		4	rd ← rd[30..0] & rd[0], C ← rd[31]
	SR	SLC rd		5	rd ← rd[30..0] & C, C ← rd[31]
	SR	SR0 rd		6	rd ← '0' & rd[31..1], C ← rd[0]
	SR	SR1 rd		7	rd ← '1' & rd[31..1], C ← rd[0]
	SR	SRA rd		8	rd ← rd[31] & rd[31..1], C ← rd[0]
	SR	SRC rd		9	rd ← C & rd[31..1], C ← rd[0]

Table 2.2: Shift and Rotate Instructions

As seen in the table above, basic shift and rotate instructions are also included to provide single bit shifts and rotations, multiplication and division. For simplicity, these instructions can only shift or rotate one bit per instruction. The user can either shift in a 0, 1, the MSB or LSB of the value in the specified register. Each of these instructions uses the single register format with the same opcode but with differing function values.

Compare/Test	**R**	CMP rd, rs	010101	-	C ← rd – rs, Z ← rd – rs
	IM	CMPI rd, imm	010100	-	C ← rd – imm, Z ← rd – imm
	R	TST rd, rs	010011	-	C ← rd xor rs
	IM	TSTI rd, imm	010010	-	C ← rd xor imm

Table 2.3: Compare and Test Instructions

Register and immediate compare and test instructions were the last logic instructions considered important to include in this instruction set. These instructions aid in the previously discussed instructions as well as conditional branch instructions. As further explained in the report, there are no explicit “branch if greater than”, less than, etc. Instead, the user may execute the CMP instruction in conjunction with a branch instruction and select the function necessary to achieve the same result.

Data Transfer	**R**	TFR rd, rs	000001	-	rd ← rs
	IM	TFRI rd, imm	000000	-	rd ← imm
	R	LDW rd, rs	000111	-	rd ← M(rs[5..0])
	IM	LDWI rd, imm	000110	-	rd ← M(imm[5..0])
	R	STW rd, rs	101111	-	M(rs[5..0]) ← rd
	IM	STWI rd, imm	101110	-	M(imm[5..0]) ← rd

Table 2.4: Data Transfer Instructions

Register to register transfer, memory storage and load instructions were also included in the final set. Due to time constraints, the only way to store data into the data memory is through the use of registers, thus there is no direct way of storing one immediate word into memory.

Jump and Call	**JBC**	CALL imm	110000	-	Go to subroutine at imm
	JBC	CALL Z, imm	110001	0	If Z = 1: Go to subroutine at imm
	JBC	CALL NZ, imm		1	If Z = 0: Go to subroutine at imm
	JBC	CALL V, imm		2	If V = 1: Go to subroutine at imm
	JBC	CALL NV, imm		3	If V = 0: Go to subroutine at imm
	JBC	CALL N, imm		4	If N = 1: Go to subroutine at imm
	JBC	CALL NN, imm		5	If N = 0: Go to subroutine at imm
	JBC	CALL C, imm		6	If C = 1: Go to subroutine at imm
	JBC	CALL NC, imm		7	If C = 0: Go to subroutine at imm
	JBC	JMP aaa	110100	-	Jump to instruction in address im
	JBC	JMP Z, aaa	110101	0	If Z = 1: Jump to instruction in address imm
	JBC	JMP NZ, aaa		1	If Z = 0: Jump to instruction in address imm
	JBC	JMP V, aaa		2	If V = 1: Jump to instruction in address imm
	JBC	JMP NV, aaa		3	If V = 0: Jump to instruction in address imm
	JBC	JMP N, aaa		4	If N = 1: Jump to instruction in address imm
	JBC	JMP NN, aaa		5	If N = 0: Jump to instruction in address imm
	JBC	JMP C, aaa		6	If C = 1: Jump to instruction in address imm
	JBC	JMP NC, aaa		7	If C = 0: Jump to instruction in address imm

Table 2.5: Jump and Call Instructions

The user can jump and call up to 65536 different addresses, thanks to the 16 bit address width. It should be noted that to keep memory utilization minimal, only 1024 addresses can be accessed due to the address space of the current instruction memory, having only 10 bits per address. Each of these instructions are also conditional; depending on the function chosen, the system can jump to or call a subroutine depending on the status of a specified ALU flag. Similarly, the user may return from a subroutine conditionally or unconditionally, as shown in the table below.

Return from Subroutine	**NOP**	RTS	101010	-	Return from Subroutine
	NOP	RTS Z	101011	0	If Z = 1: Return from Subroutine
	NOP	RTS NZ		1	If Z = 0: Return from Subroutine
	NOP	RTS V		2	If V = 1: Return from Subroutine
	NOP	RTS NV		3	If V = 0: Return from Subroutine
	NOP	RTS N		4	If N = 1: Return from Subroutine
	NOP	RTS NN		5	If N = 0: Return from Subroutine
	NOP	RTS C		6	If C = 1: Return from Subroutine
	NOP	RTS NC		7	If C = 0: Return from Subroutine

Table 2.6: RTS Instructions

Finally, conditional and unconditional branch instructions are also made available, and can be seen in the following table. Conditional branches depend entirely on the function bits provided in the instruction and correspond to the ALU flags set in the prior instruction.

	Instruction Type	Instruction	Opcode (binary)	Function (Decimal)	Description
Branch	**JBC**	BR	110010	-	Branch to offset
	JBC	BR Z, offset	110011	0	Branch to offset if Z = 1
	JBC	BR NZ, offset		1	Branch to offset if Z = 0
	JBC	BR V, offset		2	Branch to offset if V = 1
	JBC	BR NV, offset		3	Branch to offset if V = 0
	JBC	BR N, offset		4	Branch to offset if N = 1
	JBC	BR NN, offset		5	Branch to offset if N = 0
	JBC	BR C, offset		6	Branch to offset if C = 1
	JBC	BR NC, offset		7	Branch to offset if C = 0

Table 2.7: Branch Instructions

To achieve the bandwidth of a typical 32 bit processor, the system bus and each register inherently has a width of 32 bits. 32 general purpose registers are also included to avoid delays

in data manipulation. Other notable system registers are also described in the table below.

	Type	Name	Width (bits)	Description
Registers	User	r0 ... r31	32	32 General Purpose Registers
	System	PC	16	Program Counter
		SP	16	Stack Pointer
		SA	32	ALU Source A

Table 2.8: User and System Registers

As previously stated, it was opted to keep memory utilization at a minimum while at the same time maximizing the possibility of future memory upgrades if the user so chooses. To achieve greater bandwidth, the instruction and data memory have been placed in separate banks, each with their own bus to and from the datapath. In terms of data memory, the user has access to 64 32-bit words. The instruction memory can hold 1024 32-bit instructions. Lastly, the call stack can hold 32 addresses at a time, and uses its own register bank to do so.

	Type	Name	Data Width (bits)	Address Width (bits)	Depth
Memory	User	Data	32	6	64
	System	Instruction	32	10	1024
		Call Stack	16	5	32

Table 2.9: System memory

B. *Instruction Formats*

		31	26	25	21	20	16	15	0
Instruction Types	**R**	opcode		rd		rs		--	
	IM	opcode		rd		immediate			
	SR	opcode		rd		fn			
	JBC	opcode		fn	offset	address			
	NOP	opcode		fn					

Table 2.10: Instruction Formats

From the previously discussed instructions and the targets for the total amount of memory and registers, it was decided that five different instruction types must be included. Each of these instruction types are detailed in the table above. The unused 16 LSBs in the R-type instruction may have been used for an additional source register or a function field, but due to time constraints this was not considered.

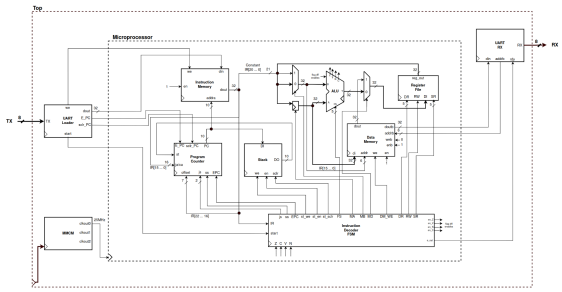

Figure 2.1: Top Block Diagram

Above is a top-level diagram of the microprocessor, including the I/O and clock divider peripherals. The control circuit includes the instruction decoder, program counter and the stack. The datapath is essentially the other components.

C. *Loader*

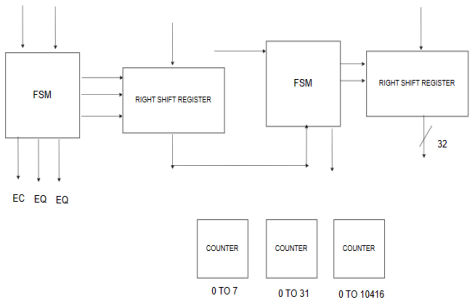

Figure 2.2: Loader Block Diagram

This VHDL code defines a UART transmitter control module with a finite state machine (FSM) that manages the transitions of different states during UART transmission. The code implements various components, including counters (my_genpulse_sclr), shift registers (my_pashiftreg, my_pashiftreg_sclr), and two FSM processes to control state transitions and output behaviors. The first FSM (Transitions) regulates the flow of data through UART by monitoring and controlling signals like TXD, zC, and zQ. It transitions between three states, S1, S2, and S3, depending on the values of these signals and manages FSM outputs to ensure correct timing and data transmission. The second FSM (Transitions2) focuses on operations related to 32-bit data shifting, including clearing signals and generating bit-related outputs. The FSMs handle data preparation, transmission, and the synchronization required for UART operation. Various outputs, like EC, ER, sclrC, EZ, and others, are set based on the current state to ensure proper UART data shifting, timing, and control flow.

D. *Instruction and Data memory*

To save FPGA resources, block RAMs were used in place of registers for these two memories. The key difference between the two is the instruction memory is a single port RAM, while the data memory is a true dual port RAM. The first port of the data memory is used by the datapath, while the second is used by the output circuit. Since the output circuit does not need to write to this memory, that port was set as read only. The

minimum read/write time of each of these memories is one clock cycle. Since each instruction takes two clock cycles to complete, there has been no issues in terms of data storage or retrieval.

E. *Instruction Decoder*

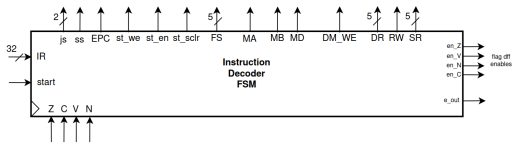

Figure 2.3: Instruction Decoder

The instruction decoder is a single FSM that sends control signals to the datapath, and only receives one signal from the loader circuit. Since each instruction takes two clock cycles, the FSM only needed three states to operate. The first state initializes the stack to 31 and clears the ALU flags. The second state executes the first part of the instruction, and updates the program counter. Because the datapath operates on a single bus, the source register of an instruction must be latched on the ALU operand A register. Since the program counter is updated at this point, JBC instructions must be completed in this state. The third state finishes the instruction, if necessary. Instructions that utilize a source operand are loaded onto the B operand of the ALU, and the destination register, if necessary, latches the new data from the bus. Since all instructions are finalized in this state, the ALU flag flip flops are enabled to accept their new values. Below is a detailed ASM diagram of the FSM.

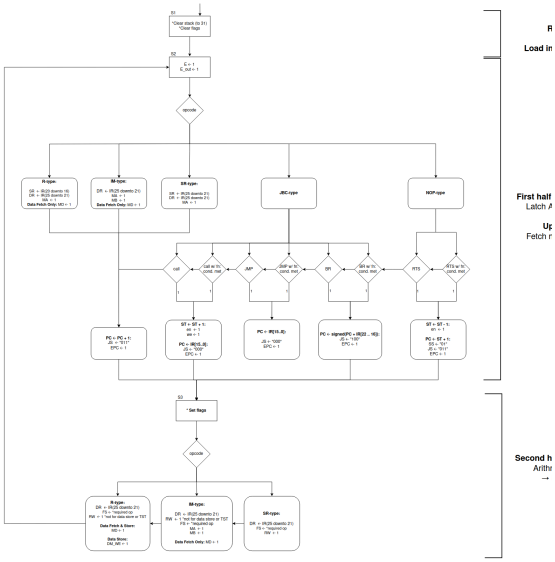

Figure 2.4: Loader Block Diagram

F. *Datapath*

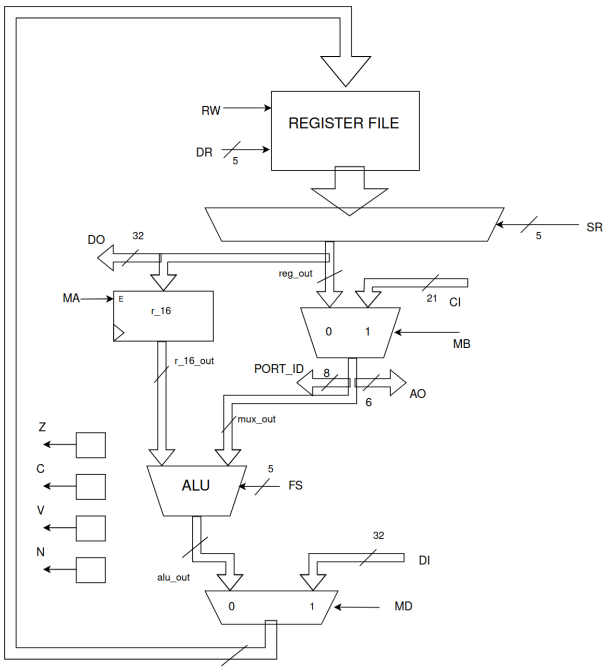

Figure 2.5: Datapath Block Diagram

The datapath, as depicted in the preceding diagram, comprises several components. Firstly, there is a Register File containing 32 registers, many of which are 16 bits wide. Additionally, there is an Arithmetic Logic Unit (ALU) that retains flags such as C (carry) and Z (zero), using flip flops. Lastly, there is an I/O interface incorporated into the datapath. This configuration allows the datapath to execute the microoperations necessary for an instruction based on the Control signals it receives from the Instruction Decoder (ID). Notably, certain functionalities are facilitated through specific control signals.

G. *ALU*

The ALU described here operates on two input operands, A and B, of N bits each, along with control signals for selecting the operation to be performed. The ALU architecture comprises various components such as adders, shifters, and flip-flops, each serving specific functions within the computation process. Notably, the adder components handle addition and subtraction operations, while the shifters facilitate shifting operations in both left and right directions. Additionally, flip-flops are utilized for storing and synchronizing control signals and status flags like carry, overflow, zero, and negative. The code incorporates multiplexers to select the appropriate result based on the control signals provided. These

results include the outcome of arithmetic and logical operations between A and B, as well as the shifted versions of A. Moreover, the status flags are updated accordingly to reflect the outcome of the performed operations, aiding in subsequent decision-making processes within the digital system.

H. *Program Counter*

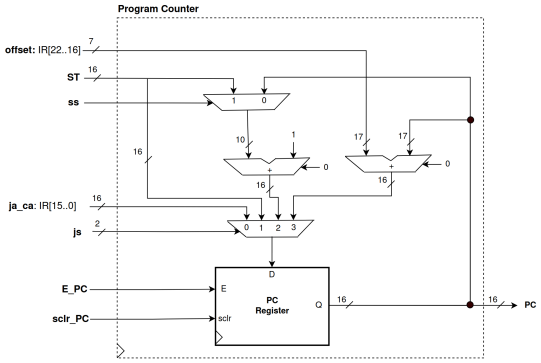

Figure 2.5: Loader Block Diagram

The program counter is able to handle conditional and unconditional branches, jumps and subroutine calls. The offset used in the branch instructions is a 7-bit signed number, which is first sign extended then added to the current program counter value, which has also been sign extended for the addition. When a branch instruction has been issued, the offset is added to the current PC. On a subroutine call, the current PC is saved on the stack, and the PC becomes set to the address specified by the ja_ca signal. Jump instructions are similar, except the PC isn't saved to the stack. On return from subroutine, the previous PC is loaded from the stack and 1 is added to it. The correct operation is set by the instruction decoder, specifically using the js, ss, and E_PC signal. To be clear, the instruction memory uses only 10 bits for the address, so the 6 MSBs of the ST and PC signals are discarded upon reaching the instruction memory.

I. *Call Stack*

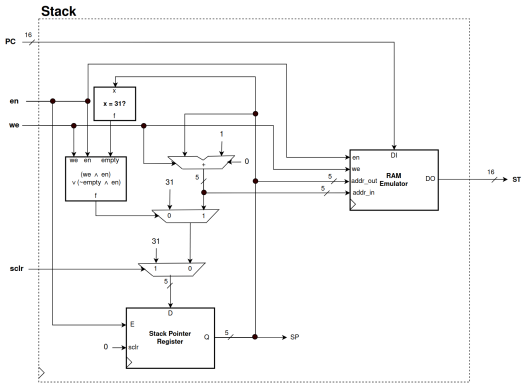

Figure 2.6: Call Stack Block Diagram

The call stack is essentially a stack data structure that holds the program counter value upon a subroutine call. When a subroutine is called, the current program counter is pushed onto the stack by setting the 'en' and 'we' signals to 1 from the instruction decoder. The current stack pointer decrements and at the same time stores the current PC at the register corresponding to the updated stack pointer. On a return from subroutine instruction, the stack pointer increments by having the 'en' signal set high by the instruction decoder. In either case, the current address specified at the current stack pointer value is always available on the 'ST' output. If the stack is full, the stack pointer will cycle back to 32, overwriting that PC value. If the stack is empty, the stack pointer will not increment beyond its maximum value of 31.

J. *Output*

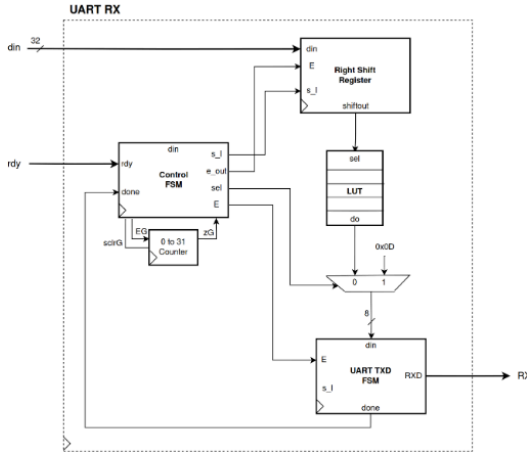

Figure 2.7: Output Block Diagram

At its core, the architecture includes a finite state machine named uart_output_fsm,

responsible for coordinating the entire transmission process. It manages tasks such as data serialization and signaling when transmission is complete. Accompanying this FSM are components like uart_rx, which handles data reception and acknowledgment signaling, ensuring data integrity during transmission. Additionally, custom components like my_genpulse_sclr generate pulses to control timing and synchronization, crucial for accurate data transmission. The architecture also features a shift register component, my_pashiftreg, aiding in the serialization of data bits for transmission. Through signal routing and multiplexers, the code orchestrates data manipulation and routing, ensuring smooth and reliable communication between the FPGA and external devices[3].

II. Experimental Setup

To verify the functioning of the processor, each component as well as the overall top level circuit was simulated using Xilinx's Vivado 2019.1 behavioral simulator. Obtaining correct functionality of each component was essential before simulating the processor in the top circuit. The stack for instance needed to successfully push and pop given address values and always show the current address at its output. The program counter needed to provide the correct count depending on the control signals provided to it. The datapath needed to receive the correct control signals from the instruction decoder that also needed to send them at the correct times, depending on the instruction received from the instruction memory.

After behavioral simulations were deemed successful, UART input and output was tested using picocom, a minimal dumb-terminal emulation program[4]. The test program included the following:

```
TFRI r0 0b0000000000000001000
TFRI r1 0b0000000000000000100
TFRI r2 0b0000000000000000010
TFRI r3 0b0000000000000000001
STW  r0 0b0000000000000111111
STW  r1 0b0000000000000111110
STW  r2 0b0000000000000111101
STW  r3 0b0000000000000111100
```

Figure 3.1: Test program

In the above program, four different numbers are first loaded onto four separate registers. Each of these registers was loaded into the last four locations of the instruction memory. Using two switches as a selector, the value located at the selected address was sent from memory to the connected computer upon pressing the button BTNC[1].

III. Results

Due to time constraints, the implementation of the UART loader was unsuccessful. In its place, instructions were loaded directly into the instruction memory, through the use of a coefficient file. In doing so, the correct output was produced on the terminal according to its corresponding address. It also turned out that the words produced on the output were reversed, as the bits were sent with the MSB being first. Lastly, the system clock had to be divided down to 25MHz due to combinational delays.

Conclusions

Overall, the project was successful. A 32-bit microprocessor was produced, with an extensive amount of instructions. Both memories are expandable, providing the possibility of incorporation into a larger design. Having larger instructions could mean the possibility for more instructions, each with their own functions. Improvements could be made to the UART loader and output, making the two function as they should. Other improvements could include having two system buses instead of one, pipelining, and readjusting the instruction formats. Lastly, it would be interesting to decrease combinational delays to boost the system clock up to 100MHz as opposed to 25MHz.

References

[1] "Nexys A7 Reference Manual." Digilent Inc., Dec. 12, 2018

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

[3] Llamocca, Daniel. "Unit 3: External Peripherals: Interfacing" VHDL Coding for FPGAs. https://www.secs.oakland.edu/~llamocca/Courses/ECE4710/Notes%20-%20Unit%203.pdf

[4] N. Patavalis, "NPAT-efault/PICOCOM: Minimal dumb-terminal emulation program," GitHub, https://github.com/npat-efault/picocom (accessed Mar. 20, 2024).