16-bit Architecture HDL/FPGA Implimentation

Department of Electrical and Computer Engineering
University of Dayton
Adam Dulay

ECE 499 Special Topic in Computing

Table of Contents

Overview

Throughout this project, the design process, implementation, and testing of a custom 16-bit CPU/architecture is showcased. The design was implemented in VHDL, simulated in Questa modelsim, and synthesized in Intel's Quartus software. After the design was realized and tested, it was then compared, at the surface level, to a 16-bit architecture design that was implemented on the same board/FPGA in order to more clearly understand the benefits and drawbacks of the custom design.

Design

Instruction Set Design

The first step of any CPU design is determining the language it will speak (in this case the instruction set). Given that the CPU designed in this project is 16-bit, the instruction that drives it was also created to fit 16-bit words. The determination of the instructions the computer can execute was accomplished by first looking at different instruction sets of well-known CPUs throughout the years (notable entries include the 6502, Z80, x86, ARM, and 32-bit MIPS). Out of the many instructions all of these machines are capable of executing, the instructions that were most commonly shared among all of them were selected as possible instructions for the working set. Finally, a reduced instruction set computer (RISC) philosophy was determined to be favorable as simple is more often sufficient. Finally the instruction set was trimmed down to the final set of 16 instructions that can be seen below in Table 1. The most constraining factor for this instruction set design is the immediate values (up to 6 bits at the end of the immediate type instruction set). This means that the end programmer will only be able to load an immediate value from program memory up to 65 unsigned (31 signed).

Table 1: Instruction Set and Their Descriptions

Keyword	**Opcode**	**Description**
add	0x0	Adds contents of r1 and r2 and stores sum to rd
sub	0x1	Subtracts contents of r2 from r1 and stores difference to rd
mul	0x2	Multiplies contents of r1 and r2 and stores product to rd
div	0x3	Divides contents of r1 by r2 and stores quotient to rd
sr	0x4	Shifts contents of r1 right by a 6-bit immediate value of places and stores contents to rd
sl	0x5	Shifts contents of r1 left by a 6-bit immediate value of places and stores contents to rd

and	0x6	Bitwise AND of r1 and r2 and stores contents to rd
or	0x7	Bitwise OR of r1 and r2 and stores contents to rd
addi	0x8	Adds value of 6-bit immediate to r1 and stores contents to rd
xor	0x9	Bitwise XOR of r1 and r2 and stores contents to rd
not	0xA	Bitwise inversion of r1 and stores contents to rd (contents of r2 do not matter)
stor	0xB	Stores contents of register to stack pointer + 9-bit offset
load	0xC	Loads contents of stack pointer + 9-bit offset to register
bran	0xD	Checks if register operands are equal, if so jumps to address specified by 6-bit immediate
jmp	0xE	Jumps to contents of register + 9-bit offset
noop	0xF	Does nothing, moves onto next instruction

After the instructions the CPU will be able to interpret were determined, the instructions were then broken up into groups that have the same execution pattern. Below in Table 2, the instruction types, their syntax patterns, and the amount of clock cycles they take to execute can be seen. Some of the instruction types inherit from MIPS assembly, but most of the others are on their own as it was impossible to get them all to take the same amount of clock cycles to execute.

Table 2: Instruction Classifications, Syntax, and Execution Time

Classification	**Instructions Within Class**	**Syntax (Totals to 16-bits) ("+" indicates concatenation of bits)**	**Execution Time (cycles)**
Register type (r-type)	Add, sub, mult, div, and, or, xor, and not	Opcode (4-bits) + destination register (3-bits) + operand register 1 (3-bits) + operand register 2 (3-bits) + (3-bits unused)	8
Immediate type (i-type)	Addi, sr, sl	Opcode (4-bits) + destination register (3-bits) + operand register 1 (3-bits) + 6-bit immediate	7
store	N/A	Opcode (4-bits) + register to store (3-bits) + offset from $sp (9-bits)	6

load	N/A	Opcode (4-bits) + register to load to (3-bits) + offset from $sp (9-bits)	8
jump	N/A	Opcode (4-bits) + register for base address (3-bits) + offset from base address value (9-bits)	6
branch	N/A	Opcode (4-bits) + operand register 1 (3-bits) + operand register 2 (3-bits) + branch address (6-bits)	7
no-op	N/A	Opcode (4-bits) + do not care about other bits (12-bits)	3

Architecture

After the instruction set's syntax was decided, the finalized functional block diagram of the architecture was created. After 3 revisions, the final block diagram can be seen in Figure 1. It is centered around a data-bus like a motherboard (for simplicity). The design features a register file (containing 8 registers: A, B, C, D, E, F, SP, and Zero). Registers A-F are general-purpose registers, SP indicates the stack pointer register, and Zero indicates the "zero register" (a read-only register that always contains "0"). The registers can be accessed via 3-bit addresses. Directly below the register file, (in red), is the arithmetic logic unit (ALU). The ALU takes two opperand and outputs the result of the arithmetic operation (determined by the 4-bit instruction opcode) and an "equal flag" that is used to determine branching outcomes by the control logic. Between the ALU and the register file is a multiplexor (mux). The mux determines where the second operand for ALU calculations comes from (either from ROM in the case of an immediate instruction or from the register file in the case of a register instruction). To the left of the bus is the program counter, which is incremented after each instruction is executed, and has its value tied to the address of the program ROM (which determines which instruction is fed to the control logic next). The memory of the CPU, also seen to the left of the bus, is split into two subsections: random access memory (RAM) and program read-only memory (ROM). The stack pointer constantly points to an address in RAM for use in function stack call operations and could also be used for function recursion (while memory lasts). Finally, above everything else on the left, is the control logic. The control logic is the "brains" of the CPU, as it interprets the instructions and outputs a series of control signals (microcode) to the inner components of the CPU. The control logic was implemented sequentially in the form of a finite state machine (FSM), which means that the clock, and other control signals, are driven slower than the main input clock's rate by the FSM (everything is not linked on one clock input as that was the easiest way to implement the design). Finally, all sub-modules within the CPU had to be created to run sequentially (with a clock input), in order to be able to be driven by a FSM. This was the main sacrifice in speed/efficiency made for implementation simplicity in this design.

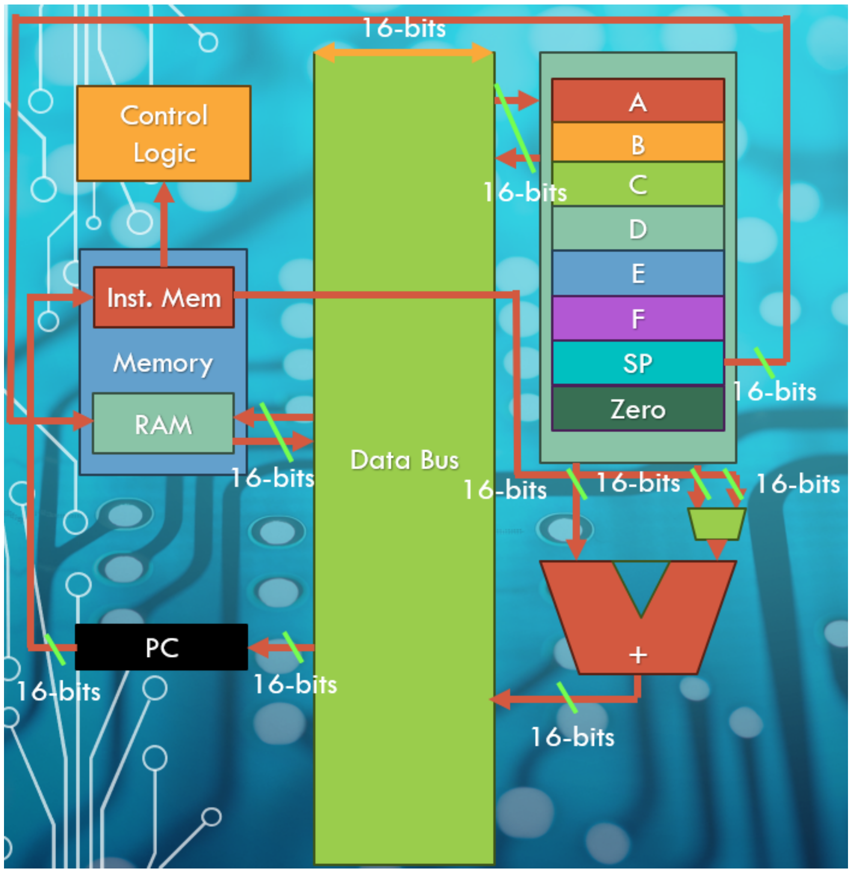


Figure 1 : CPU Architecture Functional Block Diagram

HDL Module Design

Once the Architecture of the CPU was conceptualized, the design work for each of the CPU's subsystems was stated throughout the rest of this section the design methodologies for each subsystem is discussed. This section is meant to be seen alongside the VHDL files.

Register File

The register file was implemented using an array of 16-bit standard logic vectors, which can be looked up using 3-bit addresses (also in the form of standard logic vectors). The module can output two 16-bit operands (which are found by inputting their respective register's addresses into the address inputs) that inevitably go to the ALU. In addition to the ALU outputs, the register file can also output a register's contents onto the bus, and it constantly outputs the contents of the stack pointer directly to the RAM module.

ALU

The ALU was originally designed using three asynchronous processes (two for shifting and one to asynchronously assign the result output), but during implementation, the ALU had to be changed, at the expense of performance, in order to be sequential (on a clock). The ALU still works the same way as it did asynchronously except it waits to output the result until the rising edge of the input clock line (which is driven by the control logic's FSM). The ALU was one of a few of the subsystems that had to be converted into a synchronous module, with a register, in order to function.

Program Counter

The program counter is, and has always been, a synchronous module. Incrimenting, jumping, or branching based on the 2-bit opcode it receives from the control logic's FSM, it changes its contents in order to point to the next instruction to be fetched by the CPU. This module can not only accept new addresses from the bus, but also directly from the control FSM in order to save some clock cycles during branch instructions.

Program ROM

The program ROM was fashioned from the "memory_2.vhd" example file from ECE 501. The module uses a Quartus ".mif" file in order to instantiate the memory cells (in this case with the program instructions). A file "sim_mem_init_16.vhd" is needed in order to interpret the ".mif" file in Questa, and was created by modifying the "sim_mem_init.vhd" file, also from ECE 501, to work with 16-bit words instead of 8-bit ones. The ROM module outputs the contents of the memory address input lines to the data out lines at the rising edge of the input clock signal. This was done not only so it was FSM-compatible for the control logic, but also in order to eliminate any memory hazards (which would have been possible in RAM if it was not implemented in the same manner).

RAM

The RAM unit was designed with the same process as the ROM module. The only differences between the two are that the RAM file allows the control logic's FSM to write in data values to its memory cells from the data bus, and that the RAM unit is initialized with a blank ".mif" file (filled with all 0's). As mentioned earlier, the RAM is also clocked as to prevent any memory hazards from occuring (such as reading while writing).

Control Logic

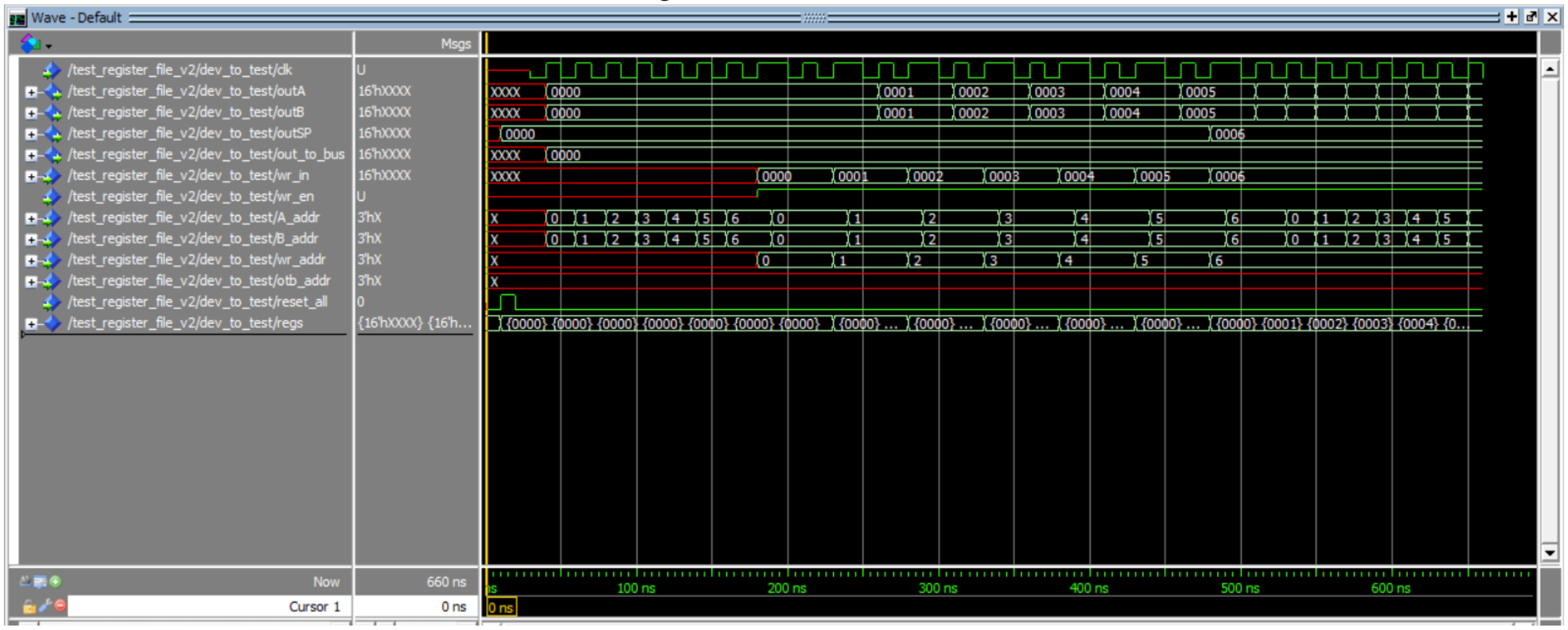
The control logic is easily the largest submodule as it consists of a FSM. The FSM is driven by the main input clock of the CPU, and raises and lowers all of the submodules' control signals depending on its current state. The FSM's design is a textbook example of a Mealy Machine as it is a FSM that determines its next state not only based on its current state but also from its input signals (the input instructions themselves).

Evaluation and Implementation: HDL Testbench

Before they could be wired together into the final CPU design, each module had to be tested individually in order to ensure that each component worked as intended. In this section, the test waveforms are shown and discussed, however the figures in this section do not do the testbenches complete justice, so the Questa simulation files/projects will be attached with this report. The memory modules (ROM and RAM) were not tested/implemented before the final build as they were based directly off of a proven design from ECE 501 ("memory_2.vhd").

Register File

The register file's testbench is performed by first resetting all of the registers asynchronously, to ensure that all of them are in a known state, and then reading them sequentially using a for loop. After the first read for loop, the values of each of the registers is overwritten with the for loop's counter variable (so they sequentially count up to match their addresses). Finally, after they are overwritten, they are read once more to prove they all retain their content. While all of this is happening, the stack pointer constantly outputs its contents to its respective output line. A crude view of the testbench can be seen in Figure 2.



Wave - Default
Msgs
/test_register_file_v2/dev_to_test/clk
/test_register_file_v2/dev_to_test/outA
/test_register_file_v2/dev_to_test/outB
/test_register_file_v2/dev_to_test/outSP
/test_register_file_v2/dev_to_test/out_to_bus
/test_register_file_v2/dev_to_test/wr_in
/test_register_file_v2/dev_to_test/wr_en
/test_register_file_v2/dev_to_test/A_addr
/test_register_file_v2/dev_to_test/B_addr
/test_register_file_v2/dev_to_test/wr_addr
/test_register_file_v2/dev_to_test/otb_addr
/test_register_file_v2/dev_to_test/reset_all
/test_register_file_v2/dev_to_test/regs
Now 660 ns
Cursor 1 0 ns
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns

Figure 2: Register File Testbench Waveform

ALU

The ALU's testbench, performed asynchronously for simplicity (it still tests the arithmetic functionality suitably). The operand inputs are loaded with test values, and a for loop increments through the opcodes and checks the outputs. As can be seen in the waveform in Figure 3, the equal flag signal goes high when the "0xD" opcode is present, which is used for a branch instruction. This means that the two operands that were inputted at that time were equal in value.

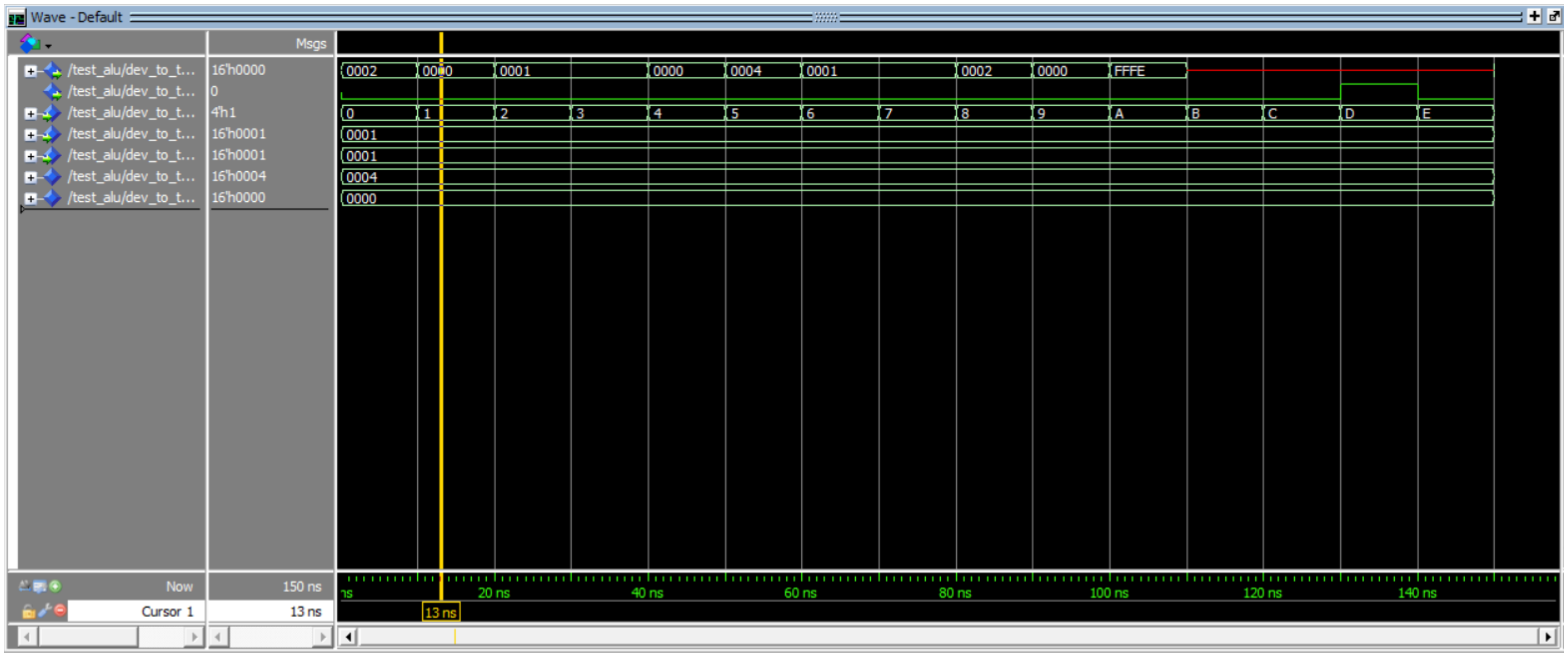

Wave - Default
Msgs
/test_alu/dev_to_t... 16'h0000
/test_alu/dev_to_t... 0
/test_alu/dev_to_t... 4'h1
/test_alu/dev_to_t... 16'h0001
/test_alu/dev_to_t... 16'h0001
/test_alu/dev_to_t... 16'h0004
/test_alu/dev_to_t... 16'h0000
Now 150 ns
Cursor 1 13 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
13 ns

Figure 3: ALU Testbench Waveform

Program Counter

The program counter's testbench was conducted synchronously in order to test its naturally-incrimenting function (it can increment its internal address value without the help of the ALU). The program counter is first reset, to make its contents known values, and then the clock line is stimulated in order to see its contents increment. Throughout the testbench, the program counter's opcde is changed in order to load new values into its internal register (which simulates a branch or jump instruction). All of this behavior can be seen in the waveform in Figure 4.

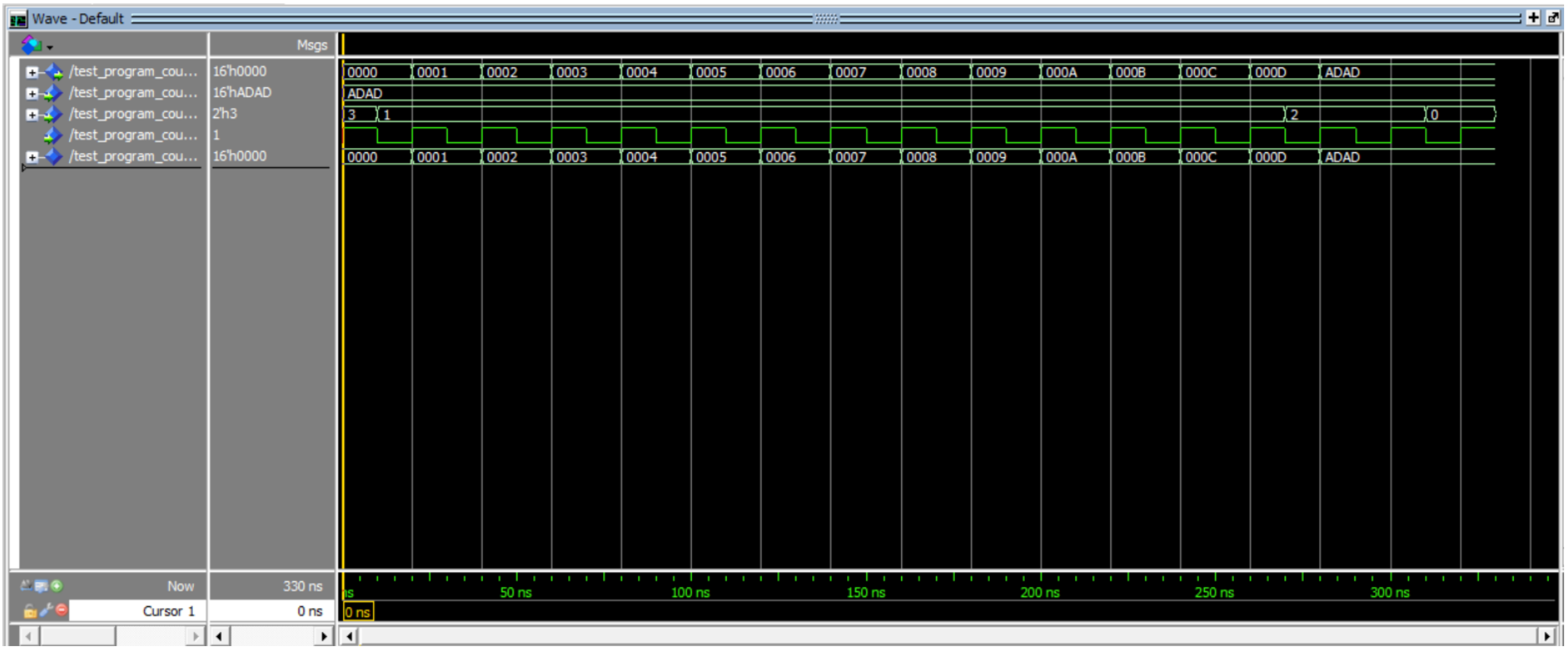
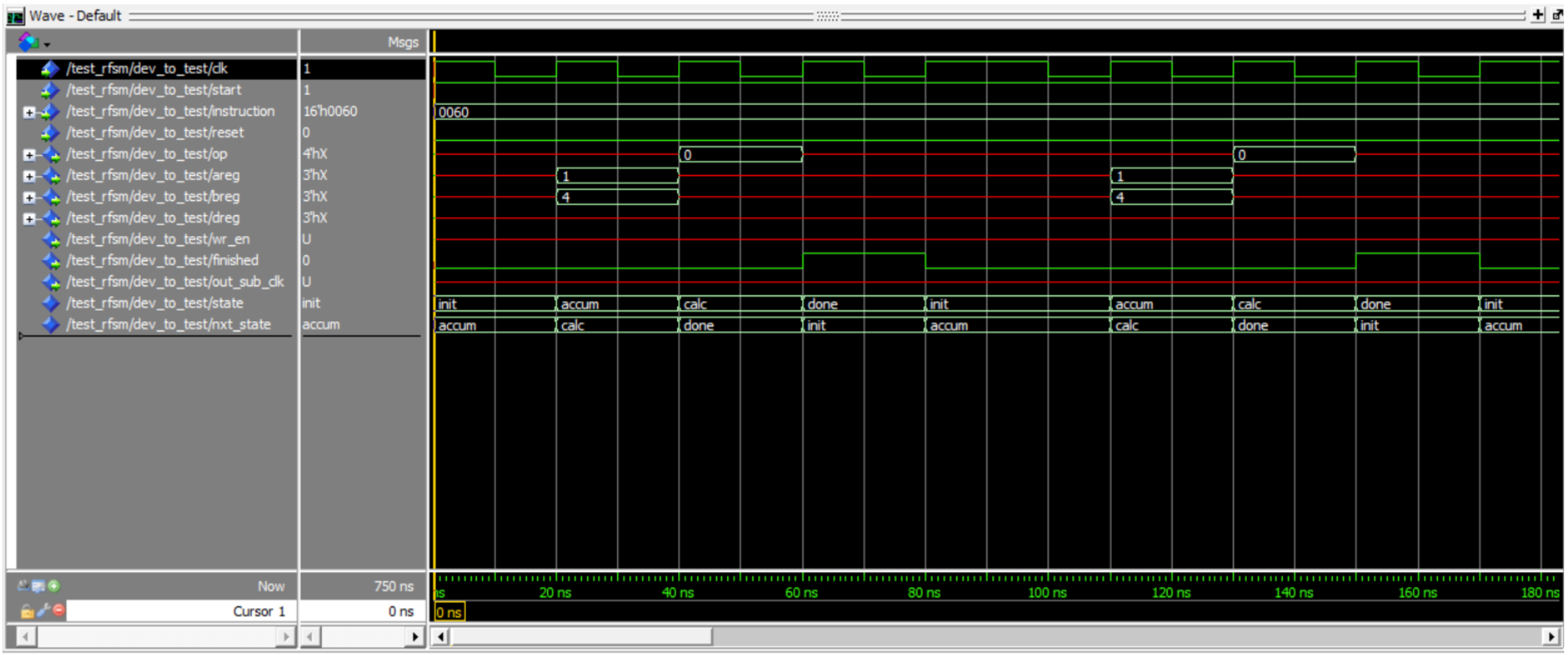


Figure 4: Program Counter Testbench Waveform

Control Logic

The control logic's FSM was tested using a bare-bones style approach in order to see if the method of describing it in VHDL was suitable for use as the control module. In the testbench, seen in Figure 5, a simple r-type simulation was conducted, and as the clock is pulsed, the FSM goes from state to state (all while raising and lowering pseudo control signals). Though technically a Moore machine, the basic FSM seen in the testbench was later scaled up and also made to respond to the current instruction word that is inputted by the program ROM. The final control logic (which was constructed alongside the top file and therefore cannot be easily tested by itself), will be seen driving the CPU later during the sample program section.



```
/test_rfsm/dev_to_test/clk           1
/test_rfsm/dev_to_test/start         1
/test_rfsm/dev_to_test/instruction   16'h0060
/test_rfsm/dev_to_test/reset         0
/test_rfsm/dev_to_test/op            4'hX
/test_rfsm/dev_to_test/areg          3'hX
/test_rfsm/dev_to_test/breg          3'hX
/test_rfsm/dev_to_test/dreg          3'hX
/test_rfsm/dev_to_test/wr_en         U
/test_rfsm/dev_to_test/finished      0
/test_rfsm/dev_to_test/out_sub_clk   U
/test_rfsm/dev_to_test/state         init
/test_rfsm/dev_to_test/nxt_state     accum
Now                                  750 ns
Cursor 1                             0 ns
```

Figure 5: Simplified Finite State Machine Testbench Waveform (Proof of Concept)

Evaluation and Implementation: Synthesis and RTL Implementation

In the section below, the previously tested modules are synthesized in Intel's Quartus software into a bitstream that can be uploaded onto an FPGA. Unfortunately, a hardware demonstration cannot be outlined here, but the Quartus project file will be included with this report so hardware implementation can be realized alongside this report. In addition to the difficulty with conveying hardware functionality, the RTL viewer in Quartus is not easily capturable for the written report, so it is encouraged that the RTL viewer be opened in the downloaded Quartus file in order to fully realize the final synthesized hardware designs shown from afar in the figures below.

Register File

Below, in Figure 6, is the RTL view of the register file. It consists of registers (made from 16 individual D-flip-flops) to synchronously store the data, and multiplexers to direct the desired contents from the inputted address.

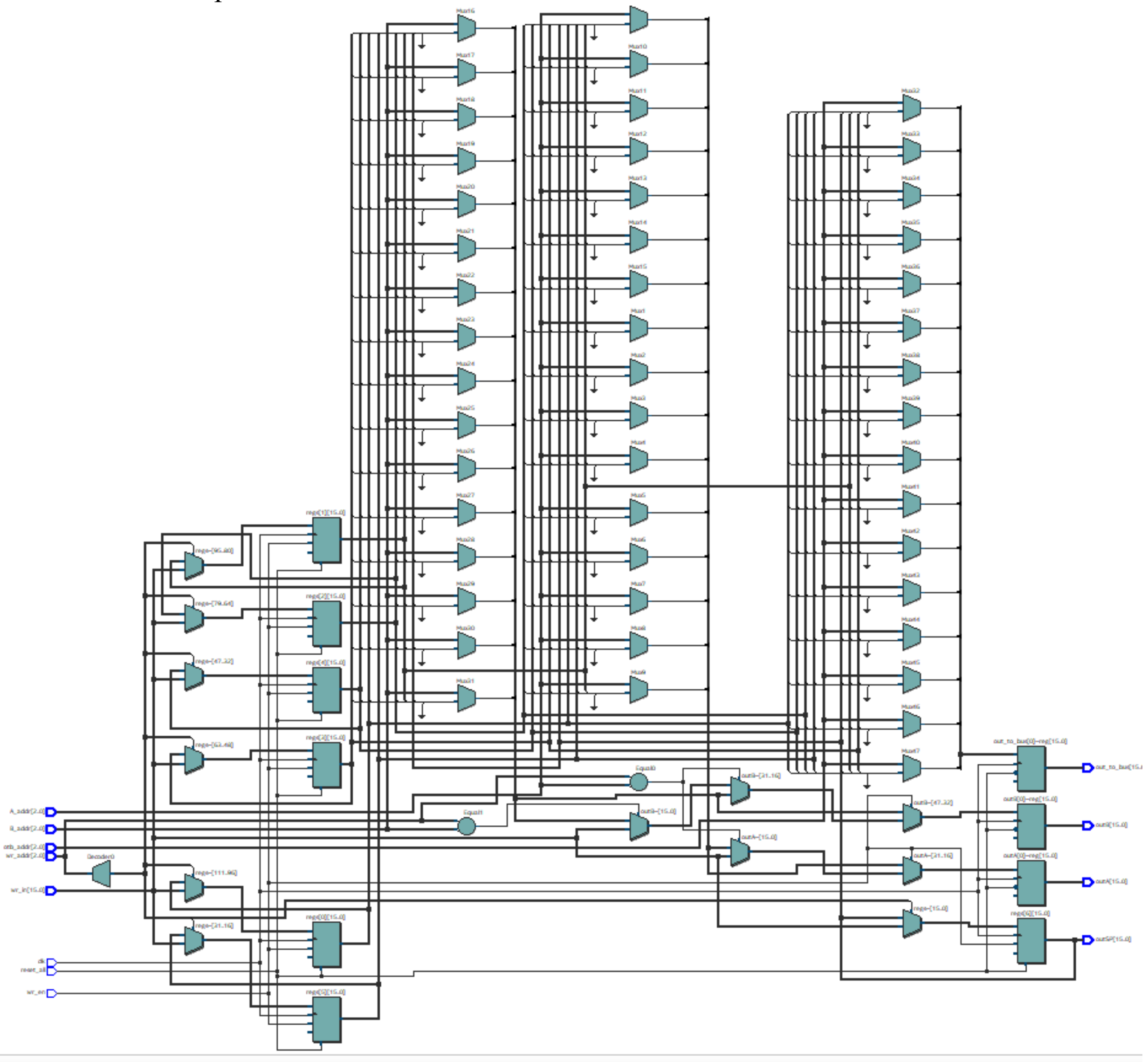


Figure 6: Register File RTL Bird's-Eye View

ALU

Below, in Figure 7, is the RTL view of the ALU module. The module itself is not that complicated (the size comes from the many shifter multiplexers in order to shift by any number of places (1-14) (left or right) in one clock cycle). The ALU also contains a multiplier, a divider, an and gate, an or gate, an xor gate, and a digital inverter, which work together with more multiplexers to direct the output based on the opcode given by the control logic's FSM.

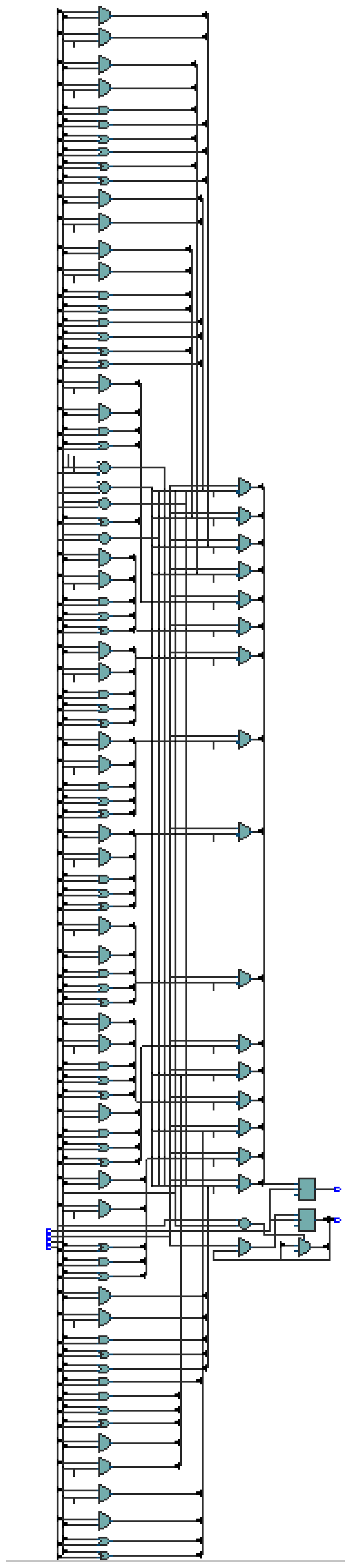


Figure 7: ALU RTL Bird's-Eye View

Program Counter

Below, in Figure 8, is the RTL view of the program counter module. This module is very simple compared to the ALU shown previously. The program counter again uses multiplexers to direct what is stored in the register, at the next rising edge of the input clock, based on the opcode it is given by the control logic's FSM.

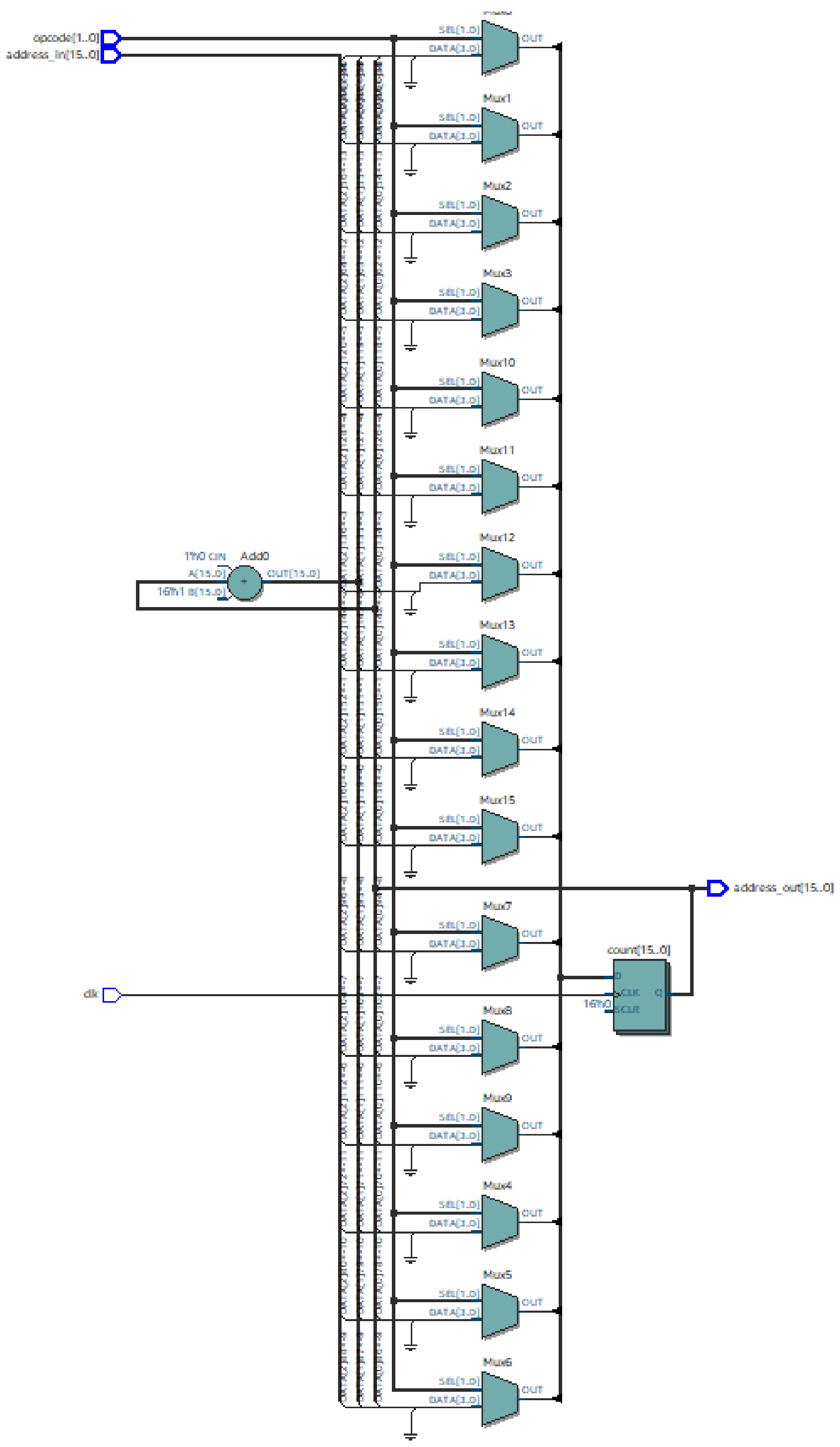


Figure 8: Program Counter RTL Bird's-Eye View

Control Logic

Below, in Figure 9, is the RTL view of the basic proof-of-concept FSM can be seen. The way Quartus synthesizes FSMs is not the way it would be implemented in real circuitry (using JK flip-flops), but it is helpful to know it is able to be synthesized for the FPGA.

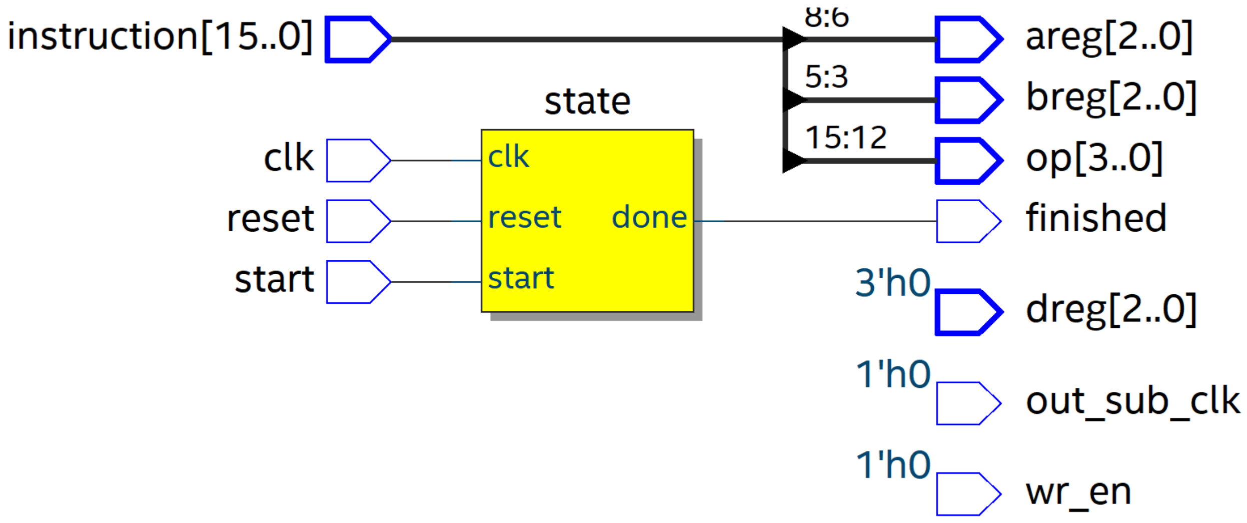
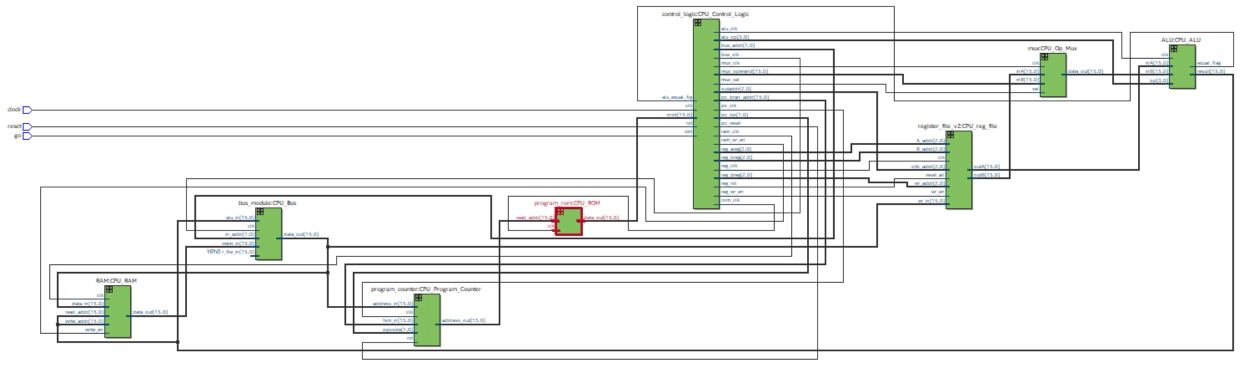


Figure 9: Proof of Concept Control Logic RTL Bird's-Eye View

Overal CPU

Below, in Figures 10, 11, and 12, are captures of the RTL view of the final CPU design. It should be mentioned that the shape of it somewhat resembles the MIPS architecture diagram, which makes sense as the instruction set demonstrated in this report was heavy based on MIPS instruction types. At the top, the control logic can be seen, with its many signals going to each and every other sub module. The datapath follows the same ROM, register, memory, steps like a MIPS processor, but differs in its placement of the ALU at the end (as it would occur before the memory module in MIPS).



control_logic:CPU_Control_Logic
mux:CPU_Op_Mux
ALU:CPU_ALU
register_file_v2:CPU_reg_file
bus_module:CPU_Bus
program_rom:CPU_ROM
RAM:CPU_RAM
program_counter:CPU_Program_Counter
clock
reset
go

Figure 10: CPU RTL Bird's-Eye View

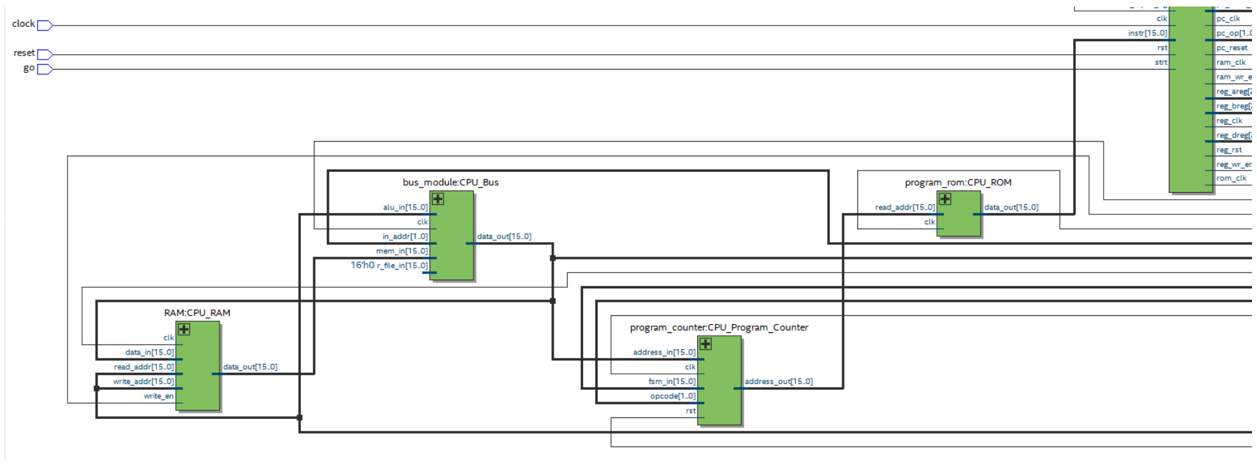


Figure 11: CPU RTL Left Side View (for better clairity)

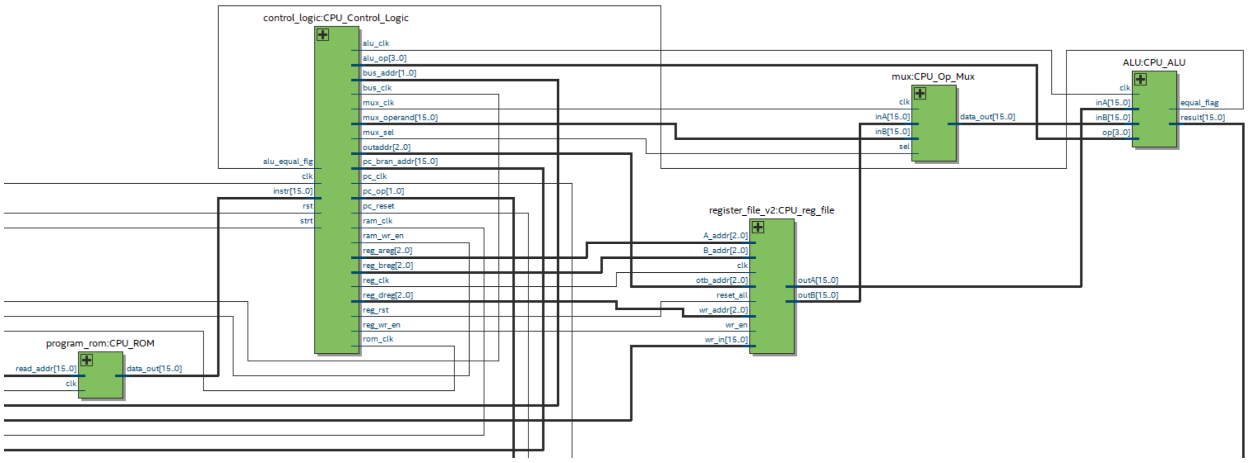
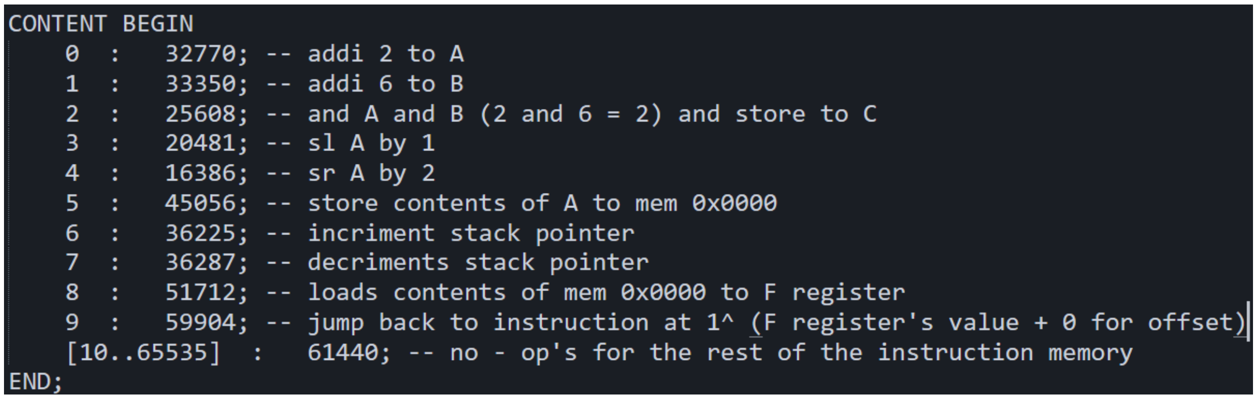


Figure 12: CPU RTL Right Side View (for better clairity)

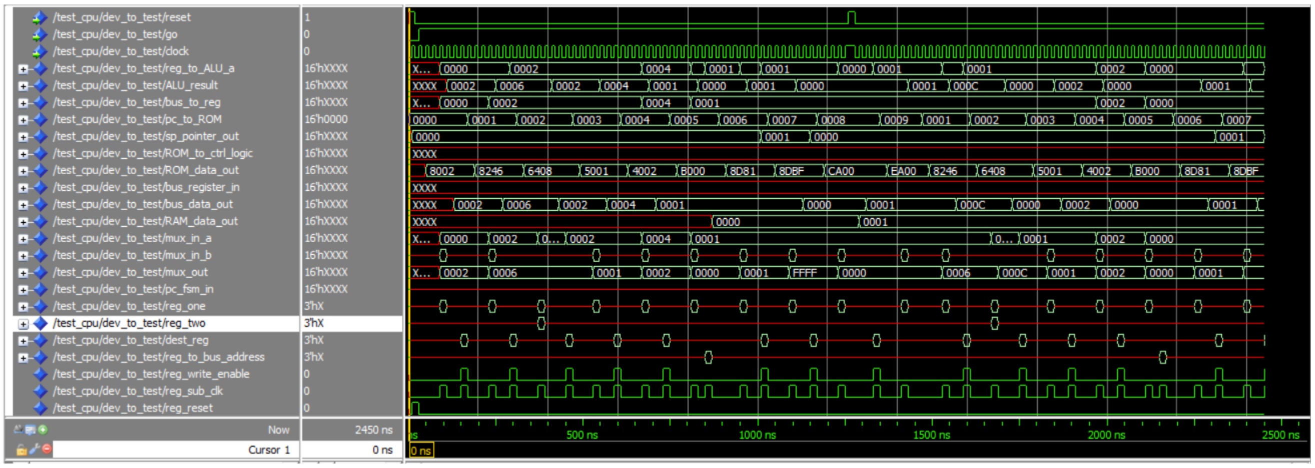
Sample Program

In this section, a sample program is simulated in Questa, so that all of the CPU's internal signals can be seen on the waveform diagram. Below, in Figure 13, is a sample program that demonstrates some of the CPU's capabilities. The program is written in the form of a ".mif" file, which is needed for Quartus to initialize the bytes into memory. The drawback to this approach is that all instructions must be converted to decimal numbers before they are written into the file, which adds a layer of difficulty when trying to write code for the CPU. Below, in Figure 14, is a snippet of the final waveform that is produced by running the sampe program (only shown to convey the scale of the output waveform). Some of the important features of the program/waveform, shown in Figures 15 and 16, are when the jump occurs and the ROM's output address changes (which can be seen in Figure 15) or when the contents of memory are loaded into the F register (which can be seen in Figure 16).

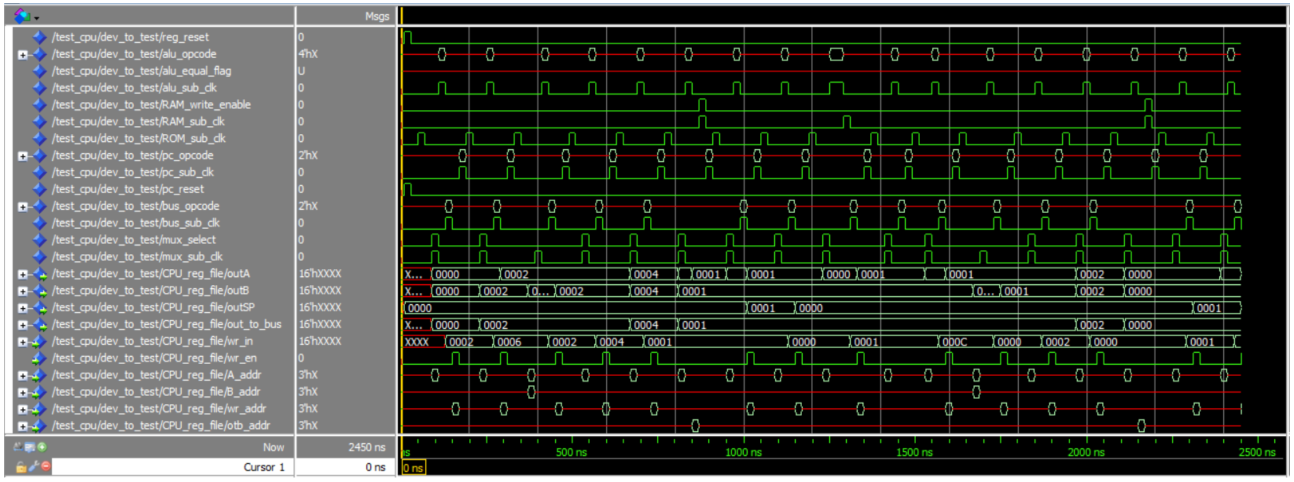


```
CONTENT BEGIN
    0  :   32770; -- addi 2 to A
    1  :   33350; -- addi 6 to B
    2  :   25608; -- and A and B (2 and 6 = 2) and store to C
    3  :   20481; -- sl A by 1
    4  :   16386; -- sr A by 2
    5  :   45056; -- store contents of A to mem 0x0000
    6  :   36225; -- incriment stack pointer
    7  :   36287; -- decriments stack pointer
    8  :   51712; -- loads contents of mem 0x0000 to F register
    9  :   59904; -- jump back to instruction at 1^ (F register's value + 0 for offset)
    [10..65535]  :   61440; -- no - op's for the rest of the instruction memory
END;
```

Figure 13: Contents of Program ROM in .mif File (inputs are in decimal by default)



/test_cpu/dev_to_test/reset	1
/test_cpu/dev_to_test/go	0
/test_cpu/dev_to_test/clock	0
/test_cpu/dev_to_test/reg_to_ALU_a	16'hXXXX
/test_cpu/dev_to_test/ALU_result	16'hXXXX
/test_cpu/dev_to_test/bus_to_reg	16'hXXXX
/test_cpu/dev_to_test/pc_to_ROM	16'h0000
/test_cpu/dev_to_test/sp_pointer_out	16'hXXXX
/test_cpu/dev_to_test/ROM_to_ctrl_logic	16'hXXXX
/test_cpu/dev_to_test/ROM_data_out	16'hXXXX
/test_cpu/dev_to_test/bus_register_in	16'hXXXX
/test_cpu/dev_to_test/bus_data_out	16'hXXXX
/test_cpu/dev_to_test/RAM_data_out	16'hXXXX
/test_cpu/dev_to_test/mux_in_a	16'hXXXX
/test_cpu/dev_to_test/mux_in_b	16'hXXXX
/test_cpu/dev_to_test/mux_out	16'hXXXX
/test_cpu/dev_to_test/pc_fsm_in	16'hXXXX
/test_cpu/dev_to_test/reg_one	3'hX
/test_cpu/dev_to_test/reg_two	3'hX
/test_cpu/dev_to_test/dest_reg	3'hX
/test_cpu/dev_to_test/reg_to_bus_address	3'hX
/test_cpu/dev_to_test/reg_write_enable	0
/test_cpu/dev_to_test/reg_sub_clk	0
/test_cpu/dev_to_test/reg_reset	0
Now	2450 ns
Cursor 1	0 ns



	Msgs
/test_cpu/dev_to_test/reg_reset	0
/test_cpu/dev_to_test/alu_opcode	4'hX
/test_cpu/dev_to_test/alu_equal_flag	U
/test_cpu/dev_to_test/alu_sub_clk	0
/test_cpu/dev_to_test/RAM_write_enable	0
/test_cpu/dev_to_test/RAM_sub_clk	0
/test_cpu/dev_to_test/ROM_sub_clk	0
/test_cpu/dev_to_test/pc_opcode	2'hX
/test_cpu/dev_to_test/pc_sub_clk	0
/test_cpu/dev_to_test/pc_reset	0
/test_cpu/dev_to_test/bus_opcode	2'hX
/test_cpu/dev_to_test/bus_sub_clk	0
/test_cpu/dev_to_test/mux_select	0
/test_cpu/dev_to_test/mux_sub_clk	0
/test_cpu/dev_to_test/CPU_reg_file/outA	16'hXXXX
/test_cpu/dev_to_test/CPU_reg_file/outB	16'hXXXX
/test_cpu/dev_to_test/CPU_reg_file/outSP	16'hXXXX
/test_cpu/dev_to_test/CPU_reg_file/out_to_bus	16'hXXXX
/test_cpu/dev_to_test/CPU_reg_file/wr_in	16'hXXXX
/test_cpu/dev_to_test/CPU_reg_file/wr_en	0
/test_cpu/dev_to_test/CPU_reg_file/A_addr	3'hX
/test_cpu/dev_to_test/CPU_reg_file/B_addr	3'hX
/test_cpu/dev_to_test/CPU_reg_file/wr_addr	3'hX
/test_cpu/dev_to_test/CPU_reg_file/otb_addr	3'hX
Now	2450 ns
Cursor 1	0 ns

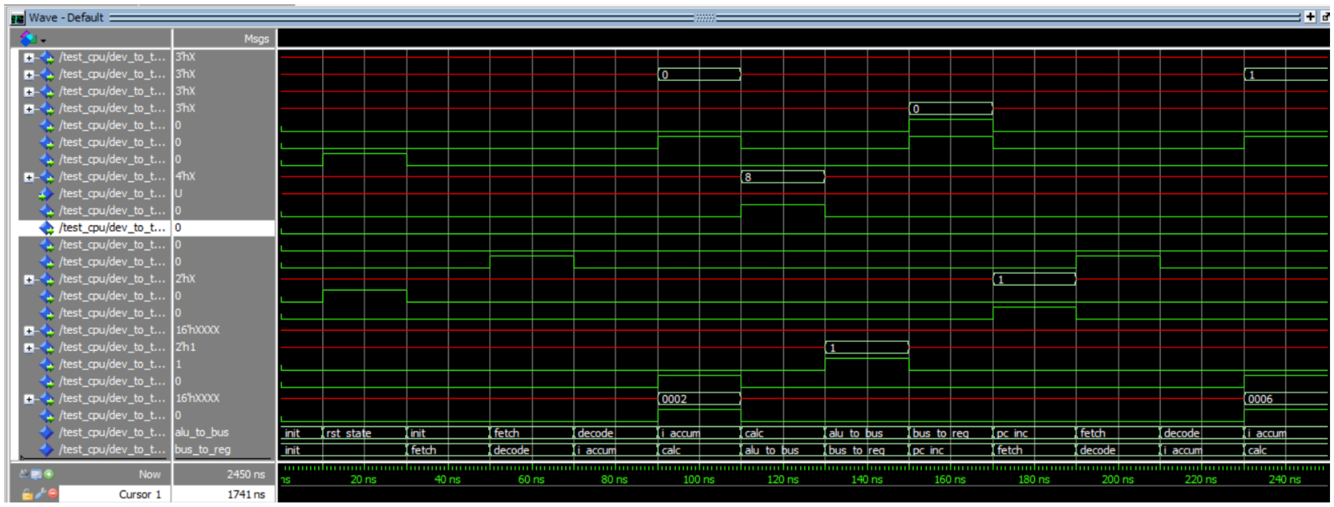


Figure 14: Bird's-Eye View of Sample Waveforms of Program Running

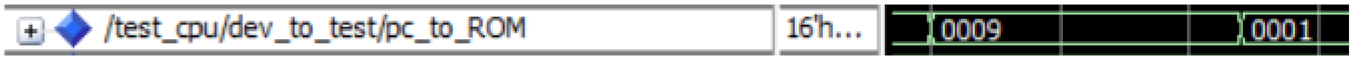


Figure 15: Jump Instruction Address Change (the one at address 9 in code)

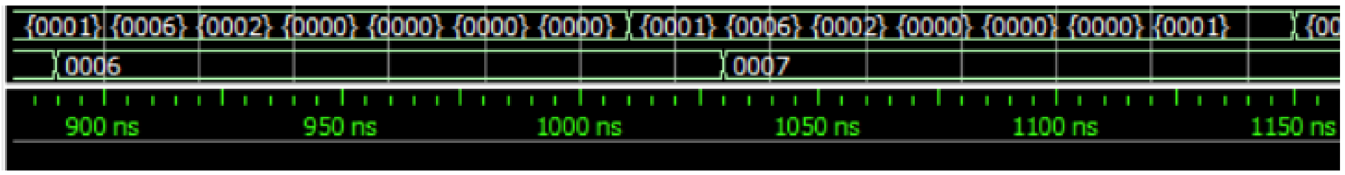


Figure 16: Load Word Instruction Register Write (the one ate address 8)

Comparison

After the custom CPU design was realized for the DE2 board (Cyclone IV), and some sample programs were written for it, compared to another 16-bit design that was implemented on the same FPGA/development board. The TinyVLW8 (TVLW8) or (the professionally-designed CPU), whose architecture can be seen in Figure 17, is a small CPU, intended for embedded controls applications, that was created in an IEEE journal entry. The design is highly optimized for size and instruction execution time, so this would serve as a great benchmark for what is possible on the Cyclone IV/DE2. The professionally-designed processor uses only 8 opcodes/instructions (despite still having 16-bit instructions), and groups them based off of the functional element within the architecture. This means that the datapath for each instruction is heavily reduced compared to if it had to travel throughout the entire architecture, like in the custom CPU's architecture. Below, in Figure 18, is the instruction set of the TLW8.

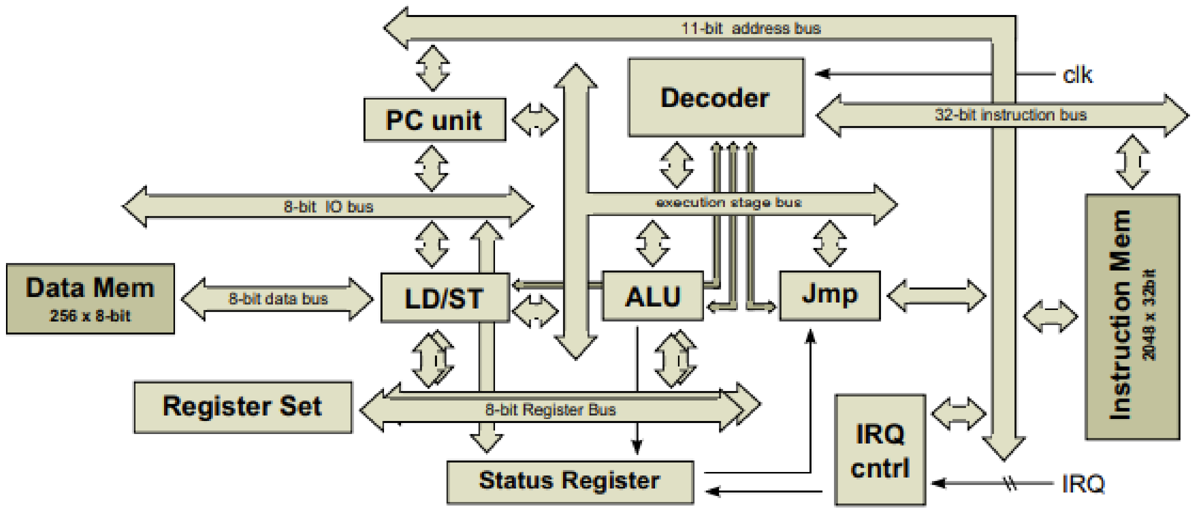
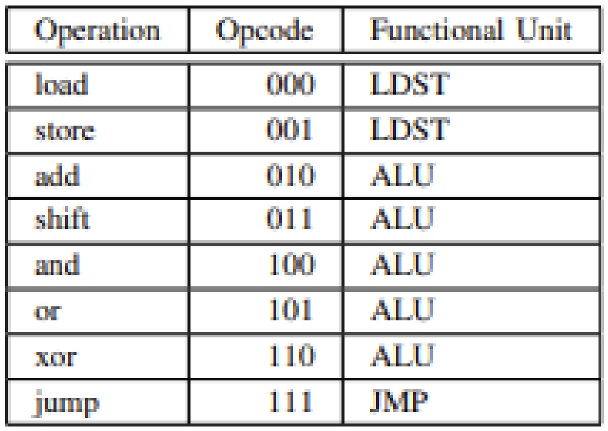


Figure 17: Architecture Block Diagram of the TinyVLW8 CPU



Operation	Opcode	Functional Unit
load	000	LDST
store	001	LDST
add	010	ALU
shift	011	ALU
and	100	ALU
or	101	ALU
xor	110	ALU
jump	111	JMP

Figure 18: Architecture Block Diagram of the TinyVLW8 CPU

Though quite a bit more complex at the architecture block-diagram level, as the TVLW8 allows for a lot of bidirectionallity in its dataflow, it can save on the amount of clock cycles per instruction (CPI) because it does not need to wait for the data to propagate through a central bus (like in the custom CPU). The instructions in the professional CPU's vocabulary are also homogonized to all fit within the same finite state space (the professional CPU uses “fetch”, “decode”, “execute”, and “write back” as its control states). Having all instructions utilize the same CPI, in a RISC setting, allows for the ability for pipelining, which allows for a CPU to utilize all its parts at once to execute multiple instructions at the same time (something the custom design will never be able to do). There is absolutely nothing that the custom CPU can perform faster than the professionally-designed CPU can, which is to be expected for an ametur design.

Conclusion

Overall, this was an extremely invaluable experience for me to be able to take on a project like this. Though my first HDL CPU design was very crude, and could not compete with any aspect of the professionally-designed CPU, the experience I have gained through this project will help me continue to hone my HDL design skills and further improve my future designs. I plan to continue to optimize my design, especially as I will be taking computer architecture at the graduate level this spring. This project has further boosted my passion for all things digital design-related, and I hope that the knowledge gained from this experience will also aid me in my future career in the world of digital VLSI/ASIC design.

Sources

O. Stecklina and M. Methfessel, "A Tiny Scale VLIW Processor for Real-Time Constrained Embedded Control Tasks," 2014 17th Euromicro Conference on Digital System Design, Verona, Italy, 2014, pp. 559-566, doi: 10.1109/DSD.2014.31.
https://ieeexplore.ieee.org/document/6927291