The Design, Simulation, and Synthesis of a Custom 16-Bit CPU

Skylar Cane Overby

Follow this and additional works at: https://louis.uah.edu/honors-capstones

Recommended Citation
https://louis.uah.edu/honors-capstones/828

This Thesis is brought to you for free and open access by the Honors College at LOUIS. It has been accepted for inclusion in Honors Capstone Projects and Theses by an authorized administrator of LOUIS.
The Design, Simulation, and Synthesis of a Custom 16-Bit CPU

by

Skylar Cane Overby

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Diploma

to

The Honors College

of

The University of Alabama in Huntsville

April 27, 2023

Honors Capstone Director: Dr. Rhonda Gaede

Student (signature) 5/1/23

Digitally signed by Rhonda Kay Gaede
Date: 2023.05.02 09:48:28 -05'00'

Director (signature) Date

Digitally signed by William Wilkerson
Date: 2023.05.03 05:14:04 -05'00'

Honors College Dean (signature) Date
Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.

In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

____________________________
Student Name (printed)

____________________________
Student Signature

Date

5/2/23
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dedication</td>
<td>4</td>
</tr>
<tr>
<td>Abstract</td>
<td>5</td>
</tr>
<tr>
<td>Introduction</td>
<td>6</td>
</tr>
<tr>
<td>Relevant Computer Architecture Theory</td>
<td>7</td>
</tr>
<tr>
<td>Design Phase</td>
<td>10</td>
</tr>
<tr>
<td>Simulation Phase</td>
<td>13</td>
</tr>
<tr>
<td>Software Design Phase</td>
<td>18</td>
</tr>
<tr>
<td>Synthesis Phase</td>
<td>21</td>
</tr>
<tr>
<td>Conclusion</td>
<td>24</td>
</tr>
<tr>
<td>Reference List</td>
<td>25</td>
</tr>
<tr>
<td>Appendix A: Verilog Designs</td>
<td>26</td>
</tr>
<tr>
<td>Appendix B: Verilog Test Drivers</td>
<td>42</td>
</tr>
<tr>
<td>Appendix C: Submodule Simulations</td>
<td>52</td>
</tr>
<tr>
<td>Appendix D: Prototype CPU Deliverables</td>
<td>61</td>
</tr>
</tbody>
</table>
Dedication

This capstone project is dedicated to my project director, Dr. Rhonda Gaede. Without her knowledge of computer architecture this project would not have been possible. This project is also dedicated to Dr. Aleksandar Milenković and Dr. Earl Wells. Their classes in embedded systems and digital hardware design provided me with the knowledge needed to complete this capstone. Finally, this capstone is dedicated to the UAH Honors College and all my peers who supported me in this endeavor.
Abstract

The goal of this project is to design and implement a custom 16-bit single-cycle CPU. The project is split into 4 phases: design, simulation, software design, and synthesis. The design phase consists of designing the custom instruction set, data path, and low-level CPU architecture. The simulation phase begins with implementing the design in software using Verilog in Quartus Prime Lite. The simulation phase tests and debugs the Verilog design using Questa. This is followed by the Software Design phase, where a program is written to demonstrate the functionality of the CPU. Once proper functionality is achieved, the implementation phase rapidly prototypes the design on physical hardware using an FPGA. The CPU is tested extensively both physically and in software. The design process is iterative and passes through each phase multiple times. Project deliverables include: the instruction set, CPU schematics, CPU Verilog implementation, and program assembly code. This project integrates concepts from multiple engineering courses into a single design task. Completion of this project demonstrates proficiency in the fundamentals of digital hardware design and computer architecture.
Introduction

The goal of this project is to design and implement a custom 16-bit single-cycle CPU. This project covers the entire process of designing, simulating, synthesizing, and documenting a simple custom processor. This project integrates knowledge from multiple engineering courses and provides a concrete deliverable demonstrating proficiency in the fundamentals of digital hardware design and computer architecture. This project builds upon similar CPU design projects already offered in senior and graduate level computer architecture courses. The UAH computer architecture course asks graduate students to design and simulate a custom CPU data path. This project uses free software and inexpensive FPGA hardware to expand on this assignment and prototype a physical CPU circuit. The CPU is then incorporated into a very basic embedded design by programming the CPU to run a simple shift-and-add multiplication algorithm. This project serves to demonstrate how free tools and inexpensive hardware can be utilized to provide students with an engaging project that synthesizes multiple sub disciplines of computer engineering.

This project is split into four phases: design, simulation, software design, and synthesis. Each phase generates its own set of concrete deliverables, allowing the project to be split up over multiple weeks. The CPU design process is iterative and constantly goes back over previous phases for bug fixes and implementation tweaks. The design phase creates the CPU instruction set and data path. The simulation phase creates and simulates Verilog models of the CPU data path. The software design phase creates a program that runs on the CPU. The synthesis phase implements the CPU design on an FPGA.
Relevant Computer Architecture Theory

A CPU consists primarily of a data path. A data path consists of all the digital logic structures required to interpret and execute binary instructions. The purpose of the data path is instruction execution, so data path design must first begin with an instruction set. An instruction set architecture is a set of instructions and formats that defines how software runs on a CPU. An example of a commonly used instruction set is the MIPS instruction set architecture. A subset of the MIPS ISA is shown in Figure 1.

![Figure 1: MIPS ISA Subset (Patterson- Hennessy, 2021).](image)

The MIPS ISA reference sheet lists the name, format, operation, and operation code for each instruction. This information is essential for the creation of a data path because the data path architecture is derived from the instruction requirements. Each of these
instructions fall into one of three formats. These formats indicate how the assembly
instructions are translated into binary code for execution. The formats for this instruction
subset are listed in Figure 2.

![MIPS Instruction Formats](image)

**Figure 2:** MIPS Instruction Formats (Patterson-Hennessy, 2021).

The R format is for register operations. The I format is for operations that use
immediate numerical values. The J format is for jump instructions. Each format requires
specific digital logic structures for proper execution. When all these structures are
combined into one data path, they require control signals to ensure only the structures
necessary are used. A data path for this subset of the MIPS ISA is shown in Figure 3.

![A Simple MIPS Data Path](image)

**Figure 3:** A Simple MIPS Data Path (Patterson-Hennessy, 2021).
The primary substructures of the data path are the program counter, instruction memory, control unit, register file, ALU, and data memory. The program counter is a binary counter that indexes the program memory and advances program execution. The instruction memory stores the instructions the CPU will execute. For this project, the test software is translated into a series of binary instructions and stored in instruction memory. The control unit translates the operation code of the current instruction into a series of control signals that activate only the required logic structures. The register file contains the CPU registers. Registers are digital hardware primitives that store the operands during math operations. The ALU is the Arithmetic Logic Unit. It handles the basic math operations of the CPU. The data memory is used to load and store the results of CPU operations. Using the MIPS ISA as a guide, this project will design and prototype a custom CPU. This process begins with designing the instruction set architecture.
Design Phase

The design phase begins with creating a custom instruction set architecture. The instruction set architecture provides the scaffolding from which the rest of the CPU will be built. The instruction set architecture consists of the individual operations the CPU will be able to execute, the class types the operations fall under, and how to translate operations into binary code. The goal of this project is to create a custom 16-bit CPU, so each instruction will be 16-bits in length. CPU registers will also be 16-bits. Each instruction will be categorized into one of the four following instruction types: register type, immediate type, jump type, and halt type. Register type instructions deal with register-to-register operations. Immediate type instructions deal with operations involving registers and immediate number values. Jump type instructions deal with instructions that jump to specified addresses in instruction memory. Halt type instructions deal with instructions that perform no operation or stop program execution. Each instruction format indicates how the instruction will be translated. The formats are divided into fields specifying the operations, registers used, and target addresses. The instruction type fields for this custom CPU are shown in Table 1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Fields</th>
<th>Field Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>4-bits</td>
</tr>
<tr>
<td>R-Type</td>
<td>opcode Rs Rt Rd</td>
<td></td>
</tr>
<tr>
<td>I-Type</td>
<td>opcode Rs Rt</td>
<td>Immediate</td>
</tr>
<tr>
<td>J-Type</td>
<td>opcode</td>
<td>Address</td>
</tr>
<tr>
<td>H-Type</td>
<td>opcode</td>
<td>Don't Care</td>
</tr>
</tbody>
</table>

Table 1: CPU Instruction Types
Once the instruction types are defined, the CPU operations must be categorized into each type. This CPU design has a total of 16 registers and can directly address 32 bytes of data memory. Registers 0 and 15 will be used as zero and link registers respectively. The word size for data memory is 2-bytes or 16-bits. The largest allowed binary value for an immediate number is 15. These values are derived from the register and immediate fields in the previous diagram. However, clever use of preloaded data memory, indirect addressing, and certain instruction operations can allow for the use of more data memory and larger immediate number values. For the sake of simplicity, the design will proceed with 32-bytes of data memory and 4-bit immediate number values. The opcode size for this CPU is 4-bits, so the CPU can perform a maximum of 16 unique instruction operations.

The end goal of this project is to have a basic program running on the CPU, so the instruction operations must support common programming constructs such as math operations, looping, logical comparisons, and function calls. The custom instruction set in Figure 2 supports these constructs. Each instruction is classified according to its respective instruction type and given a binary opcode. Once the instruction set is defined, the CPU data path is designed according to the instruction requirements. This project’s goal is to create a CPU that completes every instruction in a single cycle, so the CPU has no pipeline registers or forwarding. Single-cycle CPU’s are considered inefficient, but they are simple to design. The data path requires certain control signals to be paired with each instruction to ensure proper functionality of all path components. For example, an ALU operation signal is required to specify what operation the ALU should carry out for each instruction. The following control signals are derived from this custom instruction set.
### Table 2: CPU Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Type</th>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>R</td>
<td>0001</td>
<td>Rd = Rs + Rt</td>
</tr>
<tr>
<td>SUB</td>
<td>R</td>
<td>0010</td>
<td>Rd = Rs - Rt</td>
</tr>
<tr>
<td>AND</td>
<td>R</td>
<td>0011</td>
<td>Rd = Rs &amp; Rt</td>
</tr>
<tr>
<td>OR</td>
<td>R</td>
<td>0100</td>
<td>Rd = Rs</td>
</tr>
<tr>
<td>SLT</td>
<td>R</td>
<td>0101</td>
<td>Rd = (Rs &lt; Rt) ? 1 : 0</td>
</tr>
<tr>
<td>LW</td>
<td>I</td>
<td>0110</td>
<td>Rt = M[Rs + SignExtImm]</td>
</tr>
<tr>
<td>SW</td>
<td>I</td>
<td>0111</td>
<td>M[Rs + SignExtImm] = Rt</td>
</tr>
<tr>
<td>ADDI</td>
<td>I</td>
<td>1000</td>
<td>Rt = Rs + Imm</td>
</tr>
<tr>
<td>BEQ</td>
<td>I</td>
<td>1001</td>
<td>if(Rs == Rt), PC = Imm &lt;&lt; 1</td>
</tr>
<tr>
<td>SLL</td>
<td>I</td>
<td>1010</td>
<td>Rt = Rs &lt;&lt; Imm</td>
</tr>
<tr>
<td>SRL</td>
<td>I</td>
<td>1011</td>
<td>Rt = Rs &gt;&gt; Imm</td>
</tr>
<tr>
<td>J</td>
<td>J</td>
<td>1100</td>
<td>PC = Address &lt;&lt; 1</td>
</tr>
<tr>
<td>JL</td>
<td>J</td>
<td>1101</td>
<td>R15 = PC + 2, PC + Address &lt;&lt; 1</td>
</tr>
<tr>
<td>RET</td>
<td>J</td>
<td>1110</td>
<td>PC = Rs</td>
</tr>
<tr>
<td>NOP</td>
<td>H</td>
<td>0000</td>
<td>No Operation</td>
</tr>
<tr>
<td>HALT</td>
<td>H</td>
<td>1111</td>
<td>PC = PC</td>
</tr>
</tbody>
</table>

### Table 3: Data Path Control Signals

<table>
<thead>
<tr>
<th>Control Signal Size</th>
<th>2-bit</th>
<th>1-bit</th>
<th>1-bit</th>
<th>1-bit</th>
<th>2-bit</th>
<th>3-bit</th>
<th>1-bit</th>
<th>1-bit</th>
<th>1-bit</th>
<th>1-bit</th>
<th>1-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>d_reg</td>
<td>w_reg</td>
<td>r_mem</td>
<td>w_mem</td>
<td>m_reg</td>
<td>alu_op</td>
<td>alu_src</td>
<td>ret</td>
<td>branch</td>
<td>jump</td>
<td>halt</td>
</tr>
<tr>
<td>ADD</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AND</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SLT</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LW</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SW</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ADDI</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BEQ</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SLL</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SRL</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>J</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JL</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RET</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NOP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>HALT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Simulation Phase

With the instruction set architecture complete and data path outlined, the CPU can now be modeled and simulated in software. This project uses standard free classroom modeling and simulation tools to test and verify the CPU schematic. The CPU digital hardware submodules are modeled in Verilog, an IEEE standard hardware description language. The Verilog design for this custom CPU was written using a MIPS design pulled from FPGA4Student as a reference (Anana, 2017). The CPU data path schematic is drawn up using Intel’s Quartus Prime Lite. The CPU and its various submodules are simulated using Intel’s Questa software and a Verilog design generated from the CPU schematic diagram. The CPU data path is shown in Figure 4, and the Verilog designs for each submodule are included in Appendix A.

Figure 4: CPU Data Path
To verify the proper functionality of the CPU data path, all data path submodules as well as each individual instruction must be simulated. Multiple instructions can be verified in a single simulation by programming them into the CPU’s instruction memory. Each simulation requires a test driver written in Verilog that covers all critical module functionality. To verify proper instruction execution, the CPU data path was connected to output pins at several critical points. The test drivers for the CPU and its primary submodules are included in Appendix B. The simulations for the primary CPU submodules are included in Appendix C. The simplest submodules were not rigorously simulated and did not need their own individual test drivers. The CPU was put through four primary simulations to verify the functionality of all instructions. The first simulation verifies all register type instructions, all halt type instructions, the add immediate instruction and the load and store instructions. The second simulation verifies all logical shift instructions and the branch equals instruction. The third simulation verifies the jump instruction. The fourth simulation verifies the jump and link and return instructions. The results of the four simulations are shown in Figures 5 through 8. The output values indicate all instructions are operating as expected.
Figure 5.1: Simulation 1

Figure 5.2: Simulation 1 Continued
Figure 6.1: Simulation 2

Figure 6.2: Simulation 2 Continued
Figure 7: Simulation 3

Figure 8: Simulation 4
Software Design Phase

Once the CPU is successfully simulated and verified, software can be written to run on the CPU to test the simulation under a potential use case. The test program written for this CPU is a shift-and-add algorithm designed to multiply two 3-bit numbers together. This algorithm works by taking two input binary numbers, A and B, and an empty register C to store the result. The program checks to see if the least significant bit of B is equal to 1. If it is, the CPU adds A to register C. Then, A is shifted left logically, and B is shifted right logically. The process loops for each bit in the 3-bit number inputs. Once done, the result is stored to data memory. This simple program demonstrates that the CPU supports common coding constructs and can be programmed to run simple algorithms. The assembly and binary code for this test program is shown in Table 3.

<table>
<thead>
<tr>
<th></th>
<th>Assembly Instruction</th>
<th>Binary Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADDI R1, R1, 5</td>
<td>1000_0001_0001_0101</td>
</tr>
<tr>
<td>1</td>
<td>ADDI R2, R2, 7</td>
<td>1000_0010_0010_0111</td>
</tr>
<tr>
<td>2</td>
<td>ADDI R3, R3, 4</td>
<td>1000_0011_0011_0100</td>
</tr>
<tr>
<td>3</td>
<td>ADDI R4, R4, 1</td>
<td>1000_0100_0100_0001</td>
</tr>
<tr>
<td>4</td>
<td>Loop: AND R2, R4, R5</td>
<td>0011_0010_0100_0001</td>
</tr>
<tr>
<td>5</td>
<td>BEQ R5, R0, SKIP</td>
<td>1001_0101_0000_0001</td>
</tr>
<tr>
<td>6</td>
<td>ADD R1, R14, R14</td>
<td>0001_0001_1110_1110</td>
</tr>
<tr>
<td>7</td>
<td>Skip: SLL R1, R1, 1</td>
<td>1010_0001_0001_0001</td>
</tr>
<tr>
<td>8</td>
<td>SRL R2, R2, 1</td>
<td>1011_0010_0010_0001</td>
</tr>
<tr>
<td>9</td>
<td>SUB R3, R4, R3</td>
<td>0010_0011_0100_0011</td>
</tr>
<tr>
<td>10</td>
<td>SLT R3, R4, R7</td>
<td>0101_0011_0100_0111</td>
</tr>
<tr>
<td>11</td>
<td>BEQ R7, R0, LOOP</td>
<td>1001_0111_0000_1001</td>
</tr>
<tr>
<td>12</td>
<td>ADDI R6, R6, 15</td>
<td>1000_0110_0110_1111</td>
</tr>
<tr>
<td>13</td>
<td>SW R6, R14, 0</td>
<td>0111_0110_1110_0000</td>
</tr>
<tr>
<td>14</td>
<td>LW R6, R14, 0</td>
<td>0110_0110_1110_0000</td>
</tr>
<tr>
<td>15</td>
<td>HALT</td>
<td>1111_0000_0000_0000</td>
</tr>
</tbody>
</table>

Table 4: Test Program
The expected results of the test program are verified in the following simulation. The program starts with the inputs 5 and 7. The register used to store the result is register 14. The program multiplies 3-bit numbers so the expected result will be stored in register 14 after three loop iterations. The expected result is 35 in decimal and 0x23 in hexadecimal. The simulation in Figure 9 shows the correct result is written to register 14 after three iterations, thus verifying the CPU is functional and running the software program as expected. It is important to note that a successful simulation does not indicate that the Verilog design is synthesizable. There are several Verilog constructs that do not synthesize into a physical circuit. FPGAs are unique in that they can synthesize some Verilog constructs that would not normally be synthesizable. An example would be the “initial” construct. FPGAs are able to synthesize this construct, but this construct is normally non-synthesizable.

Figure 9.1: Software Simulation
Figure 9.2: Software Simulation Continued

Figure 9.3: Software Simulation Continued
**Synthesis Phase**

Once the CPU is successfully simulated, verified, and programmed, the CPU can be synthesized on an FPGA for rapid physical prototyping. This project uses a Basys 3 Artix-7 FPGA board for synthesis and testing. This board is inexpensive compared to other FPGA's and can even be acquired with a student discount through some distributors, making it ideal for this student project. The CPU design is synthesized to the FPGA using Xilinx's free Vivado design suite. The CPU Verilog implementation was adjusted to make the design synthesizable. The Basys 3 has a limited number of I/O pins so most of the simulation verification signals were removed from the final implementation. An xdc file was used to pair the I/O pins of the Basys 3 to the I/O signals of the CPU implementation. This xdc file was written using an example file provided by Diligent as a reference (Diligent, 2015). Vivado was used to generate a bitstream from the design and program the FPGA. The adjusted CPU schematic, Verilog implementation, and xdc file are included in Appendix D. The Basys 3 Artix-7 FPGA board is shown in Figure 10.

![Figure 10: Basys 3 Artix-7 FPGA Board](image-url)
For this project the synthesized CPU will be considered fully functional if it can successfully run the test program created in the software design phase. The output for register 14 is assigned to LED[15...0] on the Basys 3 board. The 16-bit binary result displayed by these LEDs would indicate if the test program ran successfully. The clock and reset input signals are linked to SW0 and SW1 respectively. The CPU runs when SW0 is flipped back and forth to provide a clock pulse. The CPU is reset by flipping SW1. The results of the manual test of the synthesized CPU are shown in Figure 11. The hexadecimal result is 0x0023, which is 35 in decimal. This is the expected result of the test program. This confirms the CPU was successfully synthesized and is functioning properly.

Figure 11.1: Physical Hardware Test
Figure 11.2: Physical Hardware Test Continued

Figure 11.3: Physical Hardware Test Complete
Conclusions

This project culminated in the successful synthesis of a custom 16-bit CPU prototype. This project documents the entire design process beginning with the instruction set architecture and ending with a successful manual test. The CPU supports basic programming constructs, and its instruction set is sufficient for the creation of basic embedded software. Although the CPU prototype functions properly on the Basys 3 FPGA board, this does not necessarily mean the CPU can be fabricated as its own standalone circuit. This project provides an example of how a single design can incorporate theory from multiple engineering courses. This project also provides concrete deliverables that can be used in engineering courses to guide students through the design process.

This project could be expanded upon to incorporate more concepts from embedded system design, signal processing, and operating systems. An embedded systems course could build upon this project by upgrading the CPU design to communicate with sensors or other hardware. A signal processing course could build upon this project by having the CPU run a basic signal processing algorithm such as a Fourier Transform. An operating systems class could work to create a very basic operating system to run on the CPU. The purpose of all these additions would be to further integrate concepts students have studied in their computer engineering classes. The end goal of such additions would be to generate student interest and improve student education through a single unified design project.
Reference List


 Appendix A: Verilog Designs

module my_cpu(
    clk,
    reset,
    alu_out,
    curr_instr,
    imm,
    instr_op_code,
    mem_out,
    next_pc_val,
    pc_val,
    regfile_data_1,
    regfile_data_2,
    write_back
);
    input wire  clk;
    input wire  reset;
    output wire  [15:0] alu_out;
    output wire  [15:0] curr_instr;
    output wire  [15:0] imm;
    output wire  [3:0] instr_op_code;
    output wire  [15:0] mem_out;
    output wire  [15:0] next_pc_val;
    output wire  [15:0] pc_val;
    output wire  [15:0] regfile_data_1;
    output wire  [15:0] regfile_data_2;
    output wire  [15:0] write_back;
    wire  [15:0] alu_output;
    wire  alu_src;
    wire  [3:0] curr_op;
    wire  [15:0] current_instr;
    wire  halt;
    wire  [15:0] immediate;
    wire  jump;
    wire  [15:0] memory_output;
    wire  [15:0] next_pc_value;
    wire  [15:0] pc_value;
    wire  r_mem;
    wire  [15:0] reg_one_data;
    wire  [15:0] reg_two_data;
    wire  ret;
    wire  w_mem;
    wire  w_reg;
wire [15:0] write_back_out;
wire [2:0] SYNTHESIZED_WIRE_0;
wire [15:0] SYNTHESIZED_WIRE_1;
wire [15:0] SYNTHESIZED_WIRE_2;
wire [15:0] SYNTHESIZED_WIRE_3;
wire [15:0] SYNTHESIZED_WIRE_23;
wire [1:0] SYNTHESIZED_WIRE_5;
wire SYNTHESIZED_WIRE_6;
wire SYNTHESIZED_WIRE_7;
wire [3:0] SYNTHESIZED_WIRE_24;
wire [3:0] SYNTHESIZED_WIRE_25;
wire [1:0] SYNTHESIZED_WIRE_10;
wire [15:0] SYNTHESIZED_WIRE_12;
wire [15:0] SYNTHESIZED_WIRE_14;
wire [3:0] SYNTHESIZED_WIRE_15;
wire [3:0] SYNTHESIZED_WIRE_17;
wire SYNTHESIZED_WIRE_18;
wire [15:0] SYNTHESIZED_WIRE_20;
wire [15:0] SYNTHESIZED_WIRE_21;
wire [15:0] SYNTHESIZED_WIRE_22;

my_alu.b2v_inst(
    .alu_ctrl(SYNTHESIZED_WIRE_0),
    .src1(reg_one_data),
    .src2(SYNTHESIZED_WIRE_1),
    .zero_bit(SYNTHESIZED_WIRE_7),
    .res(alu_output));

mux_2x16 b2v_inst10(
    .selection(ret),
    .in1(SYNTHESIZED_WIRE_2),
    .in2(reg_one_data),
    .out_data(SYNTHESIZED_WIRE_3));

mux_2x16 b2v_inst11(
    .selection(halt),
    .in1(SYNTHESIZED_WIRE_3),
    .in2(pc_value),
    .out_data(next_pc_value));

mux_2x16 b2v_inst12(
    .selection(alu_src),
    .in1(reg_two_data),
    .in2(immediate),
.out_data(SYNTHESIZED_WIRE_1));

mux_3x16 b2v_inst14(
  .in1(memory_output),
  .in2(alu_output),
  .in3(SYNTHESIZED_WIRE_23),
  .selection(SYNTHESIZED_WIRE_5),
  .out_data(write_back_out));

assign SYNTHESIZED_WIRE_18 = SYNTHESIZED_WIRE_6 & SYNTHESIZED_WIRE_7;

mux_plus15 b2v_inst16(
  .in1(SYNTHESIZED_WIRE_24),
  .in2(SYNTHESIZED_WIRE_25),
  .selection(SYNTHESIZED_WIRE_10),
  .out_data(SYNTHESIZED_WIRE_17));

adder_2x16 b2v_inst18(
  .in1(SYNTHESIZED_WIRE_23),
  .in2(SYNTHESIZED_WIRE_12),
  .result(SYNTHESIZED_WIRE_20));

program_counter b2v_inst19(
  .clk(clk),
  .reset(reset),
  .next_val(next_pc_value),
  .value(pc_value));

my_control b2v_inst2(
  .reset(reset),
  .opcode(curr_op),
  .w_reg(w_reg),
  .r_mem(r_mem),
  .w_mem(w_mem),
  .alu_src(alu_src),
  .ret(ret),
  .branch(SYNTHESIZED_WIRE_6),
  .jump(jump),
  .halt(halt),
  .alu_op(SYNTHESIZED_WIRE_0),
  .d_reg(SYNTHESIZED_WIRE_10),
  .m_reg(SYNTHESIZED_WIRE_5));
sign_extender b2v_inst20(
  .in(SYNTHESIZED_WIRE_25),
  .out(immediate));

shifter b2v_inst21(
  .in(immediate),
  .out(SYNTHESIZED_WIRE_12));

shifter b2v_inst22(
  .in(SYNTHESIZED_WIRE_14),
  .out(SYNTHESIZED_WIRE_22));

splitter b2v_inst3(
  .in(current_instr),
  .jump(SYNTHESIZED_WIRE_14),
  .op(curr_op),
  .rd(SYNTHESIZED_WIRE_25),
  .rs(SYNTHESIZED_WIRE_15),
  .rt(SYNTHESIZED_WIRE_24));

adder_plus2 b2v_inst4(
  .in1(pc_value),
  .result(SYNTHESIZED_WIRE_23));

my_instr_mem b2v_inst5(
  .pc(pc_value),
  .instr(current_instr));

my_reg_file b2v_inst6(
  .clk(clk),
  .reset(reset),
  .w_reg_ctrl(w_reg),
  .read_reg1(SYNTHESIZED_WIRE_15),
  .read_reg2(SYNTHESIZED_WIRE_24),
  .write_data(write_back_out),
  .write_reg(SYNTHESIZED_WIRE_17),
  .reg1_data(reg_one_data),
  .reg2_data(reg_two_data));

my_data_mem b2v_inst7(
  .clk(clk),
  .read_ctrl(r_mem),
  .write_ctrl(w_mem),
  .address_in(alu_output),
Simulation CPU, my_cpu.v

module my_instr_mem(
    input [15:0] pc,
    output wire [15:0] instr
);
wire [3:0] mem_address = pc[4:1];
reg [15:0] mem[15:0];
initial begin
    mem[0] = 16'b1000000100010101;
    mem[1] = 16'b1000001000100111;
    mem[2] = 16'b1000001100110011;
    mem[3] = 16'b1000010001000001;
    mem[4] = 16'b0011001001000101;
    mem[5] = 16'b1001010100000001;
    mem[6] = 16'b0001000111101110;
    mem[7] = 16'b1010000100010001;
endmodule
mem[8] = 16'b1011001000100001;
mem[9] = 16'b0010001101000011;
mem[10] = 16'b0101001101000111;
mem[12] = 16'b1000011001101111;
mem[13] = 16'b011101101100000;
mem[14] = 16'b0110011011100000;
mem[15] = 16'b1111000000000000;

end
assign instr = (pc[15:0] < 32) ? mem[mem_address[3:0]] : 16'd0;
endmodule

module my_reg_file(
    input clk,reset,w_reg_ctrl,
    input [3:0] read_reg1,read_reg2,write_reg,
    input [15:0] write_data,
    output [15:0] reg1_data,reg2_data,reg14_contents
);
integer count;
reg [15:0] registers [15:0];
initial begin
    for(count=0;count<16;count=count+1)
        registers[count] <= 16'b0;
end
always @(posedge clk or posedge reset) begin
    if(reset) begin
        for(count=0;count<16;count=count+1)
            registers[count] <= 16'b0;
    end
    else begin
        if(w_reg_ctrl) begin
            registers[write_reg] <= write_data;
        end
    end
end
assign reg1_data = (read_reg1 == 0) ? 16'b0 : registers[read_reg1];
assign reg2_data = (read_reg2 == 0) ? 16'b0 : registers[read_reg2];
assign reg14_contents = registers[14];
endmodule
module my_control(
    input [3:0] opcode,
    input reset,
    output reg [2:0] alu_op,
    output reg [1:0] d_reg,m_reg,
    output reg w_reg,r_mem,w_mem,alu_src,ret,branch,jump,halt
); 
always @(*)
begin
if(reset == 1'b1) begin
    alu_op = 3'b000;
    d_reg = 2'b00;
    m_reg = 2'b00;
    w_reg = 1'b0;
    r_mem = 1'b0;
    w_mem = 1'b0;
    alu_src = 1'b0;
    ret = 1'b0;
    branch = 1'b0;
    jump = 1'b0;
    halt = 1'b0;
end
else begin
    case (opcode)
    4'b0000: begin
        alu_op = 3'b000;
        d_reg = 2'b00;
        m_reg = 2'b00;
        w_reg = 1'b0;
        r_mem = 1'b0;
        w_mem = 1'b0;
        alu_src = 1'b0;
        ret = 1'b0;
        branch = 1'b0;
        jump = 1'b0;
        halt = 1'b0;
    end
    4'b0001: begin
        alu_op = 3'b010;
        d_reg = 2'b01;
        m_reg = 2'b01;
        w_reg = 1'b1;
        r_mem = 1'b1;
        w_mem = 1'b0;
    end
end
end
alu_src = 1'b0;
ret = 1'b0;
branch = 1'b0;
jump = 1'b0;
halt = 1'b0;
end

4'b0010: begin
alu_op = 3'b011;
d_reg = 2'b01;
m_reg = 2'b01;
w_reg = 1'b1;
r_mem = 1'b0;
w_mem = 1'b0;
alu_src = 1'b0;
ret = 1'b0;
branch = 1'b0;
jump = 1'b0;
halt = 1'b0;
end

4'b0011: begin
alu_op = 3'b000;
d_reg = 2'b01;
m_reg = 2'b01;
w_reg = 1'b1;
r_mem = 1'b0;
w_mem = 1'b0;
alu_src = 1'b0;
ret = 1'b0;
branch = 1'b0;
jump = 1'b0;
halt = 1'b0;
end

4'b0100: begin
alu_op = 3'b001;
d_reg = 2'b01;
m_reg = 2'b01;
w_reg = 1'b1;
r_mem = 1'b0;
w_mem = 1'b0;
alu_src = 1'b0;
ret = 1'b0;
branch = 1'b0;
jump = 1'b0;
halt = 1'b0;
end
```plaintext
| Opcode | alu_op | d_reg | m_reg | w_reg | r_mem | w_mem | alu_src | ret | branch | jump | halt |
|--------|--------|-------|-------|-------|-------|-------|---------|-----|--------|------|------|-------|
| 4'b0101 |       |       |       |       |       |       |         |     |        |      |      |       |
|        | 3'b110 | 2'b01 | 2'b01 | 1'b1  | 1'b0  | 1'b0  | 1'b0    | 1'b0| 1'b0   | 1'b0 | 1'b0 |       |
| 4'b0110 |       |       |       |       |       |       |         |     |        |      |      |       |
|        | 3'b010 | 2'b00 | 2'b00 | 1'b1  | 1'b0  | 1'b0  | 1'b1    | 1'b0| 1'b0   | 1'b0 | 1'b0 |       |
| 4'b0111 |       |       |       |       |       |       |         |     |        |      |      |       |
|        | 3'b010 | 2'b00 | 2'b00 | 1'b0  | 1'b0  | 1'b1  | 1'b1    | 1'b0| 1'b0   | 1'b0 | 1'b0 |       |
| 4'b1000 |       |       |       |       |       |       |         |     |        |      |      |       |
|        | 3'b010 | 2'b00 | 2'b00 | 1'b0  | 1'b0  | 1'b0  |         |     |        |      |      |       |
```
w_reg = 1'b1;
r_mem = 1'b0;
w_mem = 1'b0;
alu_src = 1'b1;
ret = 1'b0;
branch = 1'b0;
jump = 1'b0;
halt = 1'b0;
end

4'b1001: begin
alu_op = 3'b011;
d_reg = 2'b00;
m_reg = 2'b00;
w_reg = 1'b0;
r_mem = 1'b0;
w_mem = 1'b0;
alu_src = 1'b0;
ret = 1'b0;
branch = 1'b1;
jump = 1'b0;
halt = 1'b0;
end

4'b1010: begin
alu_op = 3'b100;
d_reg = 2'b00;
m_reg = 2'b00;
w_reg = 1'b1;
r_mem = 1'b0;
w_mem = 1'b0;
alu_src = 1'b1;
ret = 1'b0;
branch = 1'b0;
jump = 1'b0;
halt = 1'b0;
end

4'b1011: begin
alu_op = 3'b101;
d_reg = 2'b00;
m_reg = 2'b00;
w_reg = 1'b1;
r_mem = 1'b0;
w_mem = 1'b0;
alu_src = 1'b1;
ret = 1'b0;
jump = 1'b0;
halt = 1'b0;
end
36

branch = 1'b0;
jump = 1'b0;
halt = 1'b0;
end

4'b1100: begin
alu_op = 3'b000;
d_reg = 2'b00;
m_reg = 2'b00;
w_reg = 1'b0;
r_mem = 1'b0;
w_mem = 1'b0;
alu_src = 1'b0;
ret = 1'b0;
branch = 1'b0;
jump = 1'b1;
halt = 1'b0;
end

4'b1101: begin
alu_op = 3'b000;
d_reg = 2'b10;
m_reg = 2'b10;
w_reg = 1'b1;
r_mem = 1'b0;
w_mem = 1'b0;
alu_src = 1'b0;
ret = 1'b0;
branch = 1'b0;
jump = 1'b1;
halt = 1'b0;
end

4'b1110: begin
alu_op = 3'b000;
d_reg = 2'b00;
m_reg = 2'b00;
w_reg = 1'b0;
r_mem = 1'b0;
w_mem = 1'b0;
alu_src = 1'b0;
ret = 1'b1;
branch = 1'b0;
jump = 1'b0;
halt = 1'b0;
end

4'b1111: begin
module my_control;

alu_op = 3'b000;
d_reg = 2'b00;
m_reg = 2'b00;
w_reg = 1'b0;
r_mem = 1'b0;
w_mem = 1'b0;
alu_src = 1'b0;
ret = 1'b0;
branch = 1'b0;
jump = 1'b0;
halt = 1'b1;
end

default: begin
alu_op = 3'b000;
d_reg = 2'b00;
m_reg = 2'b00;
w_reg = 1'b0;
r_mem = 1'b0;
w_mem = 1'b0;
alu_src = 1'b0;
ret = 1'b0;
branch = 1'b0;
jump = 1'b0;
halt = 1'b0;
end
endcase
end
endmodule

Control Unit, my_control.v

module my_data_mem(
    input clk,read_ctrl,write_ctrl,
    input [15:0] address_in,write_data,
    output [15:0] out_data
);

integer count;
reg [15:0] mem [31:0];
wire [3:0] address = address_in[4:1];
initial begin
    for(count=0;count<31;count=count+1)
        mem[count] <= 16'b0;
end
always @(posedge clk) begin

if(write_ctrl)  
    mem[address] <= write_data;
end
assign out_data = (read_ctrl == 1'b1) ? mem[address] : 16'b0;
endmodule

module my_alu(  
    input [15:0] src1,  
    input [15:0] src2,  
    input [2:0] alu_ctrl,  
    output reg [15:0] res,  
    output zero_bit  
);  
always @(*) begin  
    case(alu_ctrl)  
        3'b000: res = src1 & src2;  
        3'b001: res = src1 | src2;  
        3'b010: res = src1 + src2;  
        3'b011: res = src1 - src2;  
        3'b100: res = src1 << src2;  
        3'b101: res = src1 >> src2;  
        3'b110: begin  
            if(src1 < src2) res = 16'd1;  
            else res = 16'd0;  
            end  
        default:res = src1 + src2;  
    endcase  
end  
assign zero_bit = (res == 16'd0) ? 1'b1 : 1'b0;
endmodule

ALU, my_alu.v
module splitter(
    input [15:0] in,
    output wire [3:0] op,rs,rt,rd,
    output wire [15:0] jump
);
assign op = in[15:12];
assign rs = in[11:8];
assign rt = in[7:4];
assign rd = in[3:0];
assign jump = {4'b0000,in[11:0]};
endmodule

module program_counter(
    input clk,reset,
    input [15:0] next_val,
    output [15:0] value
);
reg [15:0] pc;
initial begin
    pc <= 16'd0;
end
always @(posedge clk or posedge reset) begin
    if(reset) begin
        pc <= 16'd0;
    end
    else begin
        pc <= next_val;
    end
end
assign value = pc;
endmodule

module sign_extender(
    input [3:0] in,
    output reg [15:0] out
);
always @(*) begin
    out <= $signed(in);
end
endmodule
module shifter(
    input [15:0] in,
    output reg [15:0] out
);
always @(*) begin
    out <= in << 1;
end
endmodule

module adder_2x16(
    input [15:0] in1,in2,
    output reg [15:0] result
);
always @(*) begin
    result = in1 + in2;
end
endmodule

module adder_plus2(
    input [15:0] in1,
    output reg [15:0] result
);
always @(*) begin
    result = in1 + 16'd2;
end
endmodule
module mux_3x16(
    input [15:0] in1,in2,in3,
    input [1:0] selection,
    output reg [15:0] out_data
);
always @(*) begin
    case(selection)
        2'b00: out_data = in1;
        2'b01: out_data = in2;
        2'b10: out_data = in3;
        default: out_data = in1;
    endcase
end
endmodule

module mux_2x16(
    input [15:0] in1,in2,
    input selection,
    output reg [15:0] out_data
);
always @(*) begin
    case(selection)
        1'b0: out_data = in1;
        1'b1: out_data = in2;
        default: out_data = in1;
    endcase
end
endmodule
module mux_plus15(
    input [3:0] in1,in2,
    input [1:0] selection,
    output reg [3:0] out_data
);

always @(*) begin
    case(selection)
    2'b00: out_data = in1;
    2'b01: out_data = in2;
    2'b10: out_data = 4'd15;
    default: out_data = in1;
    endcase
end
endmodule

4-bit 2-Input Multiplexor, mux_plus15.v
Appendix B: Verilog Test Drivers

module tb_my_cpu();
   // Inputs
   reg clk, reset;
   // Outputs
   wire [15:0] alu_out, curr_instr, imm, mem_out, next_pc_val, pc_val, regfile_data_1, regfile_data_2, write_back;
   wire [3:0] instr_op_code;
   // Unit Test
   my_cpu uut (  
      .clk(clk),
      .reset(reset),
      .alu_out(alu_out),
      .curr_instr(curr_instr),
      .imm(imm),
      .mem_out(mem_out),
      .next_pc_val(next_pc_val),
      .pc_val(pc_val),
      .regfile_data_1(regfile_data_1),
      .regfile_data_2(regfile_data_2),
      .write_back(write_back),
      .instr_op_code(instr_op_code)
   );
   initial begin
      reset = 0;
      clk = 0;
      forever #5 clk = ~clk;
   end
endmodule

CPU Test Driver, tb_my_cpu.v

module tb_my_instr_mem();
   // Inputs
   reg [15:0] pc;
   // Outputs
   wire [15:0] instr;
   // Unit Test
   my_instr_mem uut (  
      .pc(pc),
      .instr(instr)
   );
   initial begin
module tb_my_reg_file();
// Inputs
reg clk, reset, w_reg_ctrl;
reg [3:0] read_reg1, read_reg2, write_reg;
reg [15:0] write_data;
// Outputs
wire [15:0] reg1_data, reg2_data;
// Unit Test
my_reg_file uut (
    .clk(clk),
    .reset(reset),
    .w_reg_ctrl(w_reg_ctrl),
    .read_reg1(read_reg1),
    .read_reg2(read_reg2),
    .write_reg(write_reg),
    .write_data(write_data),
    .reg1_data(reg1_data),
    .reg2_data(reg2_data)
);
initial begin
    clk = 0;
    reset = 0;
    w_reg_ctrl = 0;
    read_reg1 = 4'd0;
    read_reg2 = 4'd0;
    write_reg = 4'd0;
    write_data = 16'd0;
    forever #5 clk = ~clk;
end
always begin
    #10;  w_reg_ctrl = 1;
          write_reg = 4'd0;
          write_data = 16'd0;
    #10;  write_reg = 4'd1;
          write_data = 16'd1;
    #10;  write_reg = 4'd2;
          write_data = 16'd2;
          read_reg1 = 4'd0;
end

read_reg2 = 4'd1;
#10; write_reg = 4'd3;
write_data = 16'd3;
read_reg1 = 4'd1;
read_reg2 = 4'd2;
#10; write_reg = 4'd4;
write_data = 16'd4;
read_reg1 = 4'd2;
read_reg2 = 4'd3;
#10; write_reg = 4'd5;
write_data = 16'd5;
read_reg1 = 4'd3;
read_reg2 = 4'd4;
#10; write_reg = 4'd6;
write_data = 16'd6;
read_reg1 = 4'd4;
read_reg2 = 4'd5;
#10; write_reg = 4'd7;
write_data = 16'd7;
read_reg1 = 4'd5;
read_reg2 = 4'd6;
#10; write_reg = 4'd8;
write_data = 16'd8;
read_reg1 = 4'd6;
read_reg2 = 4'd7;
#10; write_reg = 4'd9;
write_data = 16'd9;
read_reg1 = 4'd7;
read_reg2 = 4'd8;
#10; write_reg = 4'd10;
write_data = 16'd10;
read_reg1 = 4'd8;
read_reg2 = 4'd9;
#10; write_reg = 4'd11;
write_data = 16'd11;
read_reg1 = 4'd9;
read_reg2 = 4'd10;
#10; write_reg = 4'd12;
write_data = 16'd12;
read_reg1 = 4'd10;
read_reg2 = 4'd11;
#10; write_reg = 4'd13;
write_data = 16'd13;
read_reg1 = 4'd11;
read_reg2 = 4'd12;
#10; write_reg = 4'd14;
   write_data = 16'd14;
   read_reg1 = 4'd12;
   read_reg2 = 4'd13;
#10; write_reg = 4'd15;
   write_data = 16'd15;
   read_reg1 = 4'd13;
   read_reg2 = 4'd14;
#10; read_reg1 = 4'd14;
   read_reg2 = 4'd15;
#10; reset = 1;

end
endmodule

module tb_my_controls();
// Inputs
reg [3:0] opcode;
reg reset;
// Outputs
wire [2:0] alu_op;
wire [1:0] d_reg,m_reg;
wire w_reg,r_mem,w_mem,alu_src,ret,branch,jump,halt;
// Test Unit
my_control uut (  
   .opcode(opcode),  
   .reset(reset),  
   .alu_op(alu_op),  
   .d_reg(d_reg),  
   .m_reg(m_reg),  
   .w_reg(w_reg),  
   .r_mem(r_mem),  
   .w_mem(w_mem),  
   .alu_src(alu_src),  
   .ret(ret),  
   .branch(branch),  
   .jump(jump),  
   .halt(halt)
);
initial begin  
   opcode = 4'b0000;
   reset = 1'b0;
end
always begin
  #10;  opcode = 4'b0001;
  #10;  opcode = 4'b0010;
  #10;  opcode = 4'b0011;
  #10;  opcode = 4'b0100;
  #10;  opcode = 4'b0101;
  #10;  opcode = 4'b0110;
  #10;  opcode = 4'b0111;
  #10;  opcode = 4'b1000;
  #10;  opcode = 4'b1001;
  #10;  opcode = 4'b1010;
  #10;  opcode = 4'b1011;
  #10;  opcode = 4'b1100;
  #10;  opcode = 4'b1101;
  #10;  opcode = 4'b1110;
  #10;  opcode = 4'b1111;
  #10;  opcode = 4'b0010;
      reset = 1'b1;
end
endmodule

Control Unit Test Driver, tb_my_controls.v

module tb_my_data_mem();
  // Inputs
  reg clk, read_ctrl, write_ctrl;
  reg [15:0] address_in, write_data;
  // Outputs
  wire [15:0] out_data;
  // Unit Test
  my_data_mem uut (    
      .clk(clk),
      .read_ctrl(read_ctrl),
      .write_ctrl(write_ctrl),
      .address_in(address_in),
      .write_data(write_data),
      .out_data(out_data)
  );
  initial begin
    clk = 1'b0;
    read_ctrl = 1'b0;
    write_ctrl = 1'b0;
    address_in = 16'b0;
    write_data = 16'b0;
    forever #5 clk = ~clk;
always begin
  #10;  write_ctrl = 1'b1;
       write_data = 16'd1;
  #10;  address_in = 16'd2;
       write_data = 16'd3;
  #10;  address_in = 16'd4;
       write_data = 16'd5;
  #10;  address_in = 16'd6;
       write_data = 16'd7;
  #10;  address_in = 16'd8;
       write_data = 16'd9;
  #10;  read_ctrl = 1'b1;
       write_ctrl = 1'b0;
       address_in = 16'd0;
  #10;  address_in = 16'd2;
  #10;  address_in = 16'd4;
  #10;  address_in = 16'd6;
  #10;  address_in = 16'd8;
end
endmodule
always begin
    #10;  src1 = 16’d3;
        src2 = 16’d2;
        alu_ctrl = 3’b010;
    #10;  alu_ctrl = 3’b011;
    #10;  alu_ctrl = 3’b000;
    #10;  alu_ctrl = 3’b001;
    #10;  alu_ctrl = 3’b100;
    #10;  alu_ctrl = 3’b101;
    #10;  alu_ctrl = 3’b110;
end
endmodule

ALU Test Driver, tb_my_alu.v

module tb_splitter();
// Inputs
reg [15:0] in;
// Outputs
wire [15:0] jump;
wire [3:0] op,rs,rt,rd;
// Unit Test
splitter uut (  
    .in(in),  
    .op(op),  
    .rs(rs),  
    .rt(rt),  
    .rd(rd),  
    .jump(jump)  
);
initial begin  
    in = 15’b0001001000110100;
    #10;  in = 15’b0101011001111000;
end
endmodule

Bus Splitter Test Driver, tb_splitter.v

module tb_program_counter();
// Inputs
reg clk,reset;
reg [15:0] next_val;
// Outputs
wire [15:0] value;
// Unit Test
program_counter uut (  

module tb_program_counter();
// Inputs
reg [3:0] in;
// Outputs
wire [15:0] out;
// Unit Test
sign_extender uut (  
  .in(in),
  .out(out)
);
initial begin
  in = 4'd7;
  #10;  in = 4'd8;
end
endmodule

module tb_sign_extender();
// Inputs
reg [3:0] in;
// Outputs
wire [15:0] out;
// Unit Test
sign_extender uut (  
  .in(in),
  .out(out)
);
initial begin
  in = 4'd7;
  #10;  in = 4'd8;
end
endmodule

module tb_shifter();
// Inputs
reg [15:0] in;
// Outputs
wire [15:0] out;
// Unit Test
shifter uut (  
   .in(in),  
   .out(out) 
);
initial begin  
   in = 16'd2;  
   #10; in = 16'd4;  
   #10; in = 16'd8; 
end
endmodule

Shifter Test Driver, tb_shifter.v

module tb_adder_2x16();
// Inputs
reg [15:0] in1,in2;
// Outputs
wire [15:0] result;
// Unit Test
adder_2x16 uut (  
   .in1(in1),  
   .in2(in2),  
   .result(result) 
);
initial begin  
   in1 = 16'd4;  
   in2 = 16'd5;  
   #10; in1 = in1 + 1;  
   #10; in1 = in1 + 1;  
   #10; in1 = in1 + 1; 
end
endmodule

Adder Test Driver, tb_adder_2x16.v

module tb_mux_3x16();
// Inputs
reg [15:0] in1,in2,in3;
reg [1:0] selection;
// Outputs
wire [15:0] out_data;
// Unit Test
mux_3x16 uut (  
   .in1(in1),  
   .in2(in2),  
   .selection(selection),  
   .out(out_data) 
);
initial begin  
   in1 = 16'd4;  
   in2 = 16'd5;  
   #10; in1 = in1 + 1;  
   #10; in1 = in1 + 1;  
   #10; in1 = in1 + 1;  
   #10; in1 = in1 + 1; 
end
endmodule
module tb_mux_2x16();
    // Inputs
    reg [15:0] in1,in2;
    reg selection;
    // Outputs
    wire [15:0] out_data;
    // Unit Test
    mux_2x16 uut ( 
        .in1(in1),
        .in2(in2),
        .selection(selection),
        .out_data(out_data)
    );
    initial begin
        in1 = 16'd4;
        in2 = 16'd5;
        selection = 0;
        forever #10 selection = ~selection;
    end
endmodule

module tb_mux_3x16();
    // Inputs
    .in3(in3),
    .selection(selection),
    .out_data(out_data)
);
    initial begin
        in1 = 16'd4;
        in2 = 16'd5;
        in3 = 16'd6;
        selection = 0;
    end
endmodule

3-Input Multiplexer Test Driver, tb_mux_3x16.v

2-Input Multiplexer Test Driver, tb_mux_2x16.v
Appendix C: Submodule Simulations

Instruction Memory:

Control Unit:
Register File:
ALU:
Data Memory:

Program Counter:
Splitter:

Sign Extender:
Shifter:

Adder 2x16:
Mux 3x16:

Mux 2x16:
Appendix D: Prototype CPU Deliverables

module my_cpu(
    clk,
    reset,
    register_14_contents
);

input wire   clk;
input wire   reset;
output wire  [15:0] register_14_contents;

wire  [15:0] alu_output;
wire   alu_src;
wire  [3:0] curr_op;
wire  [15:0] current_instr;
wire   halt;
wire  [15:0] immediate;
wire   jump;
wire  [15:0] memory_output;
wire  [15:0] next_pc_value;
wire  [15:0] pc_value;
wire r_mem;
wire [15:0] reg_14_con;
wire [15:0] reg_one_data;
wire [15:0] reg_two_data;
wire ret;
wire w_mem;
wire w_reg;
wire [15:0] write_back_out;
wire [2:0] SYNTHESIZED_WIRE_0;
wire [15:0] SYNTHESIZED_WIRE_1;
wire [15:0] SYNTHESIZED_WIRE_2;
wire [15:0] SYNTHESIZED_WIRE_3;
wire [15:0] SYNTHESIZED_WIRE_23;
wire [1:0] SYNTHESIZED_WIRE_5;
wire SYNTHESIZED_WIRE_6;
wire SYNTHESIZED_WIRE_7;
wire [3:0] SYNTHESIZED_WIRE_24;
wire [3:0] SYNTHESIZED_WIRE_25;
wire [1:0] SYNTHESIZED_WIRE_10;
wire [15:0] SYNTHESIZED_WIRE_12;
wire [15:0] SYNTHESIZED_WIRE_14;
wire [3:0] SYNTHESIZED_WIRE_15;
wire [3:0] SYNTHESIZED_WIRE_17;
wire SYNTHESIZED_WIRE_18;
wire [15:0] SYNTHESIZED_WIRE_20;
wire [15:0] SYNTHESIZED_WIRE_21;
wire [15:0] SYNTHESIZED_WIRE_22;

my_alu b2v_inst(  
    .alu_ctrl(SYNTHESIZED_WIRE_0),  
    .src1(reg_one_data),  
    .src2(SYNTHESIZED_WIRE_1),  
    .zero_bit(SYNTHESIZED_WIRE_7),  
    .res(alu_output));

mux_2x16 b2v_inst10(  
    .selection(ret),  
    .in1(SYNTHESIZED_WIRE_2),  
    .in2(reg_one_data),  
    .out_data(SYNTHESIZED_WIRE_3));

mux_2x16 b2v_inst11(  
    .selection(halt),  
    .in1(SYNTHESIZED_WIRE_3),  
    .in2(reg_one_data),  
    .out_data(SYNTHESIZED_WIRE_3));
.in2(pc_value),
.out_data(next_pc_value));

mux_2x16 b2v_inst12(  .selection(alu_src),
    .in1(reg_two_data),
    .in2(immediate),
    .out_data(SYNTHESIZED_WIRE_1));

mux_3x16 b2v_inst14(  .in1(memory_output),
    .in2(alu_output),
    .in3(SYNTHESIZED_WIRE_23),
    .selection(SYNTHESIZED_WIRE_5),
    .out_data(write_back_out));

assign SYNTHESIZED_WIRE_18 = SYNTHESIZED_WIRE_6 & SYNTHESIZED_WIRE_7;

mux_plus15 b2v_inst16(  .in1(SYNTHESIZED_WIRE_24),
    .in2(SYNTHESIZED_WIRE_25),
    .selection(SYNTHESIZED_WIRE_10),
    .out_data(SYNTHESIZED_WIRE_17));

adder_2x16 b2v_inst18(  .in1(SYNTHESIZED_WIRE_23),
    .in2(SYNTHESIZED_WIRE_12),
    .result(SYNTHESIZED_WIRE_20));

program_counter b2v_inst19(  .clk(clk),
    .reset(reset),
    .next_val(next_pc_value),
    .value(pc_value));

my_control b2v_inst2(  .reset(reset),
    .opcode(curr_op),
    .w_reg(w_reg),
    .r_mem(r_mem),
    .w_mem(w_mem),
    .alu_src(alu_src),
    .ret(ret),
    .branch(SYNTHESIZED_WIRE_6),
.jump(jump),
.halt(halt),
.alu_op(SYNTHESIZED_WIRE_0),
.d_reg(SYNTHESIZED_WIRE_10),
.m_reg(SYNTHESIZED_WIRE_5));

sign_extender b2v_inst20(
   .in(SYNTHESIZED_WIRE_25),
   .out(immediate));

shifter b2v_inst21(
   .in(immediate),
   .out(SYNTHESIZED_WIRE_12));

shifter b2v_inst22(
   .in(SYNTHESIZED_WIRE_14),
   .out(SYNTHESIZED_WIRE_22));

my_reg_file b2v_inst24(
   .clk(clk),
   .reset(reset),
   .w_reg_ctrl(w_reg),
   .read_reg1(SYNTHESIZED_WIRE_15),
   .read_reg2(SYNTHESIZED_WIRE_24),
   .write_data(write_back_out),
   .write_reg(SYNTHESIZED_WIRE_17),
   .reg14_contents(reg_14_con),
   .reg1_data(reg_one_data),
   .reg2_data(reg_two_data));

splitter b2v_inst3(
   .in(current_instr),
   .jump(SYNTHESIZED_WIRE_14),
   .op(curr_op),
   .rd(SYNTHESIZED_WIRE_25),
   .rs(SYNTHESIZED_WIRE_15),
   .rt(SYNTHESIZED_WIRE_24));

adder_plus2 b2v_inst4(
   .in1(pc_value),
   .result(SYNTHESIZED_WIRE_23));

my_instr_mem b2v_inst5(
   .pc(pc_value),
   .result(SYNTHESIZED_WIRE_23));
my_data_mem  b2v_inst7(
   .clk(clk),
   .read_ctrl(r_mem),
   .write_ctrl(w_mem),
   .address_in(alu_output),
   .write_data(reg_two_data),
   .out_data(memory_output));

mux_2x16  b2v_inst8(
   .selection(SYNTHESIZED_WIRE_18),
   .in1(SYNTHESIZED_WIRE_23),
   .in2(SYNTHESIZED_WIRE_20),
   .out_data(SYNTHESIZED_WIRE_21));

mux_2x16  b2v_inst9(
   .selection(jump),
   .in1(SYNTHESIZED_WIRE_21),
   .in2(SYNTHESIZED_WIRE_22),
   .out_data(SYNTHESIZED_WIRE_2));

assign register_14_contents = reg_14_con;
endmodule

set_property CLOCKS_DEDICATED_ROUTE FALSE [get_nets clk_IBUF]

# Switches
set_property PACKAGE_PIN V17 [get_ports {clk}]
   set_property IOSTANDARD LVCMOS33 [get_ports {clk}]
set_property PACKAGE_PIN V16 [get_ports {reset}]
   set_property IOSTANDARD LVCMOS33 [get_ports {reset}]

# LEDs
set_property PACKAGE_PIN U16 [get_ports {register_14_contents[0]}]
   set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[0]}]
set_property PACKAGE_PIN E19 [get_ports {register_14_contents[1]}]
   set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[1]}]
set_property PACKAGE_PIN U19 [get_ports {register_14_contents[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[2]}]
set_property PACKAGE_PIN V19 [get_ports {register_14_contents[3]}]

set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[3]}]
set_property PACKAGE_PIN W18 [get_ports {register_14_contents[4]}]

set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[4]}]
set_property PACKAGE_PIN U15 [get_ports {register_14_contents[5]}]

set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[5]}]
set_property PACKAGE_PIN U14 [get_ports {register_14_contents[6]}]

set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[6]}]
set_property PACKAGE_PIN V14 [get_ports {register_14_contents[7]}]

set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[7]}]
set_property PACKAGE_PIN V13 [get_ports {register_14_contents[8]}]

set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[8]}]
set_property PACKAGE_PIN V3 [get_ports {register_14_contents[9]}]

set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[9]}]
set_property PACKAGE_PIN W3 [get_ports {register_14_contents[10]}]

set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[10]}]
set_property PACKAGE_PIN U3 [get_ports {register_14_contents[11]}]

set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[11]}]
set_property PACKAGE_PIN P3 [get_ports {register_14_contents[12]}]

set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[12]}]
set_property PACKAGE_PIN N3 [get_ports {register_14_contents[13]}]

set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[13]}]
set_property PACKAGE_PIN P1 [get_ports {register_14_contents[14]}]

set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[14]}]
set_property PACKAGE_PIN L1 [get_ports {register_14_contents[15]}]

set_property IOSTANDARD LVCMOS33 [get_ports {register_14_contents[15]}]

Pin Assignments, constraints.xbc