# CSCE 230 Project

Final Report

# Group #9

Johnathan Carlson, Tyler Zinsmaster, Jake Ediger

This project details the creation and implementation of a Reduced Instruction Set Computer processor built in VHDL, implemented on an Altera DE1 Cyclone II FPGA development board.

|  |  |  |
| --- | --- | --- |
| **Version** | **Changes/Additions** | **Date** |
| Phase II | Added documentation for B-Type, J-type, and D-type instructions | 11/2/17-11/8/17 |
| Phase III | Extending documentation for instruction types, as well as the initial assembler | 11/13/17-11/22/17 |
| Phase IV | Finishing the basic processor | 11/19/17 |
| Phase V and Extra Credit | Finished assembler, Optional Phase V, Bonus IO, Demonstration | 11/27/17-11/3 |

# Overview

The project to design and implement a processor began with designing rudimentary parts in VHDL, a programming language that is used to implement circuits on boards like the Altera FPGA used in this project. Designing a processor on in VHDL involves making many individual parts, and putting them all together to perform tasks. 4 types of instructions were implemented separately- J-type, B-type, R-type, and D-type. After each part was created, it needed to be tested, integrated, and optimized for the processor to work. Each phase had a particular addition and/or modification. In a broad breakdown, each phase consisted of the following:

Parts added in each phase:

**Phase I-** Phase I was not a true phase, as all the parts were created prior to the beginning of the project. The parts included the 16x16 register file, the 16-bit ALU(Arithmetic Logic Unit) that can produce NZVC flags, and the barebones of the control unit. The control unit in this phase had minimal capabilities, only being able to execute R-type Instructions.

**Phase II-** Integrated the parts created in phase I to create a basic data-path compatible with R-type instructions. Additional parts needed included the IR(instruction register), immediate extender, as well other minor components. The control unit was updated with relevant flags. The adders within the ALU were optimized with carry look-ahead logic.

**Phase III-** The remaining three types of instructions- J-type, D-type, and B-type, were implemented. A memory interface, as well as the instruction address generator (IAG), were added to the data-path. The control unit was updated to accommodate these new capabilities.

**Phase IV-**The final required phase, phase IV, was based in implementing I/O. With I/O, a memory interface was needed to be fully implemented, again updating the control unit. In addition, conditional logic for all required flags was fully implemented within the Control Unit.

**Phase V and Extra Features-** Many bonus features were added to increase the functionality and ease of use of the processor. The Phase V material included Implementing J-Type instructions and expanding the I/O communication capabilities with the board to include the other Hex display panels and red LEDS. This was originally planned to be a required phase but due to time constraints was made bonus material. In addition to the Phase V I/O expansion, the I/O Memory Interface was expanded to allow for GPIO expansion header data, and a component was made to translate binary inputs to the hex display into hex display notation.

**Assembler-**The assembler was initially created to be a very basic two pass version to deal with instructions. As it was developed further, it became increasingly valuable for testing purposes. It allowed for efficient testing without re-compilation of the main program, as well catching small errors that would have gone undetected otherwise. The assembler was created in python.

# Detailed Processor Development and Implementation

## Control Unit

The control unit is arguably the most important part of the processor. It dictates all instruction specific logic and controls program flow. The control unit can be thought of as an interpreter for the processor, or, if one would rather, the nervous system that sends signals from the brain to the various organs within one’s body. It translates the individual bits within instructions to usable logic, and sets flags that determine the behavior of the individual components of the processor based on the required 5 steps that make up instruction flow and execution. The control unit was by no means complete at the beginning of phase II, when it was first implemented and used. Now, it has logic for all of the described instructions below, in addition to conditional flag checking which allows for program-level conditionals.

## Instruction Types

* **R-type –** R-type instructions were the first instructions to be implemented on the processor. R-type instructions include Add, Sub, AND, OR, XOR, JR, and CMP, and are all standard instructions. Add, Sub, and CMP make use of adders within the ALU, and these adders are improved from the original design provided in instructions, utilizing carry look-up logical behavior. See “Processor Speed” section for further explanation.
* **D-type –** D-type instructions deal with data transfer between the memory and CPU. These instructions include load word, store word, add immediate, and sub immediate. Add immediate and sub immediate utilize the alu by having their immediate values selected from a mux into the second alu input, rather than from the registry. Sub immediate is technically implemented within the hardware, but is unused in programs created through the assembler. This is because for sub immediate instructions, the assembler automatically inverts the value of the immediate and outputs an equivalent add immediate instruction. The sub immediate hardware instruction works in cases where it is tested.
* **B-type-** B-type instructions deal with branching, the main way loops and functions can be implemented in assembly. Instructions include Branch and Branch and link. These utilize the loading of addresses into the IAG, based on the address of a label the instruction branches to.
* **BONUS: J-type-** J-type instructions involve jumping commands. Jump, jump and link, and load immediate are all J-type instructions. Within the processor, J-type instructions utilize flags within the Control Unit at the different stages. This did not require major restructuring of the Control Unit or the rest of the design, as it was made with J-types in mind. However, Load Immediate necessitated an additional selection for the Multiplexor that chooses which register to save the MUX Y output into (assuming the corresponding flag for writing to the registry is true in this stage, which it is), due to the location of the output-register-specifying-bits within the instruction. J types also differ from B types in that while they both set flags that determine behavior of the Instruction Address Generator, J types do so a stage earlier than B types. This is because B types tend to require the address immediately after the specified branch address to be loaded next, so the instructions allow for the Instruction Address Generator to use its internal adder to increment the label address by 1 before outputting. J types need to jump directly to the address, as there is no address set by a label to which you would need to increment past.

## 

## I/O

Green LED’s, push buttons, and slider switches were all implemented in phase IV. The VHDL code determines which action to be performed based on the 4 bit code that corresponds to each part of I/O. A lot of difficulty occurred when moving the VHDL from phase III to the FPGA, which was necessary for phase IV I/O. Prior to implementing the program on the FPGA, a modeling software was used to test correctness. In order to transfer the functionality from the modeling software to the FPGA, the data-path needed to be drawn out so each stage could be understood. After creation of the data-path diagram, full implementation could be completed. A large step was made once we had functional I/O, but it was far from complete.

There was anomalous behavior upon the first attempts at manipulating IO beyond lighting a single LED, and after a few hours of testing, it became apparent that output pins used for debugging purposes within the top-level processor VHDL file were being mapped erroneously upon compilation and flashing to the board. Upon removal of these pins, further unwanted behavior was observed. However, this was rectified quickly and with due diligence, thanks to frustrated button-mashing. It was found that the inputs from the push buttons, while low by default on testing simulation software, was high by default when mapped as input pins to the board. This was true for all input pins on the physical hardware, and thus, simple inversion allowed for correct output.

BONUS: During Phase V, the optional red LEDs and other three Hex display panels were added to the I/O Memory Interface and mapped to proper pins. In addition, a component was made and then placed within the interface to convert any binary value of up to four bits into a 7 bit equivalent Hex display value. Any bits past the fourth bit are ignored for the translation as the Hex display implementation has been set up for up for hex values from 0 to F for each panel. This decision to translate binary inputs within the VHDL saved what would have potentially been several hours of work and assembly code resources, by removing the need for in-program translation and mapping every time one would want to use the Hex display. The benefit of this beyond the obvious time factor is that it saves vital assembly language and device memory resources, which increases available space for complexity of a program significantly.

There was interest in expanding I/O capabilities as much as possible without running out of potential 4-bit I/O memory identifier combinations. Using the DE1 reference manual as a guide, 4 GPIO Expansion header pins were mapped for additional functionality. One pin output the clock status for debugging, to ensure that the clock continued to work throughout tests. Another was an input pin, the use of which is as various in function as it is limited in scope, taking in either a 1 or a 0. Then, an output pin analogous to the input pin. Lastly, the reset pin was moved from the initial mapping of the first pushbutton, to a GPIO header pin. This allowed for greater variety of input from the pushbuttons while maintaining general reset capabilities. Though the GPIO in/out pins are largely unused, a red LED was connected to the output pin as a proof of concept to show that it works. Original plans for the header pins were audacious and involved transmission of signals between two or more development boards over radio frequencies. Due to time constraints and high levels of radio interference over the bands the transmitter/receiver worked within, this idea was scrapped. It is still possible to have a wired single-bit communication channel between two boards, but this is not used in the demonstration due to fear of damaging one or both boards, and also due to having little use within the scope of the final demonstration.

## Bonus: Assembler

The assembler had already been partially completed and had limiting functionality prior to phase IV. During phase IV, the rest of the assembler was completed. The assembler was designed in python, and proved to be a valuable tool in testing as well as implementation.

The assembler used is known as a two-pass assembler, which was necessary to be sure everything was correct. It outputs a .mif file, as well as an organized debugging interface. An example of a mif file is further down the report. The assembler was versatile in that any file could be run and tested, without the need to be redesigned for each new test. It stores the first 8 memory addresses in an array for use with I/O. A clean output provided easy debugging and testing. A variety of test files were used in the assembler, each testing a certain phase. Some phases required more than 1 test file to ensure correctness.

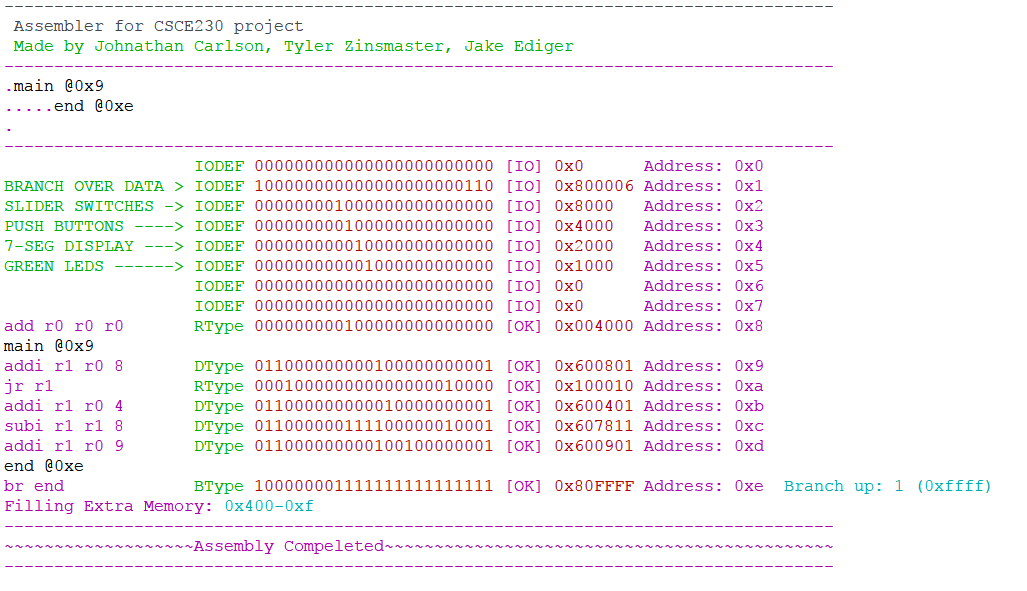


Figure 1: This image is a sample output of the assembler. The data is from one of the test files used in phase IV to test I/O with the FPGA, condensed to save room.

# Speed Overview

When flashed to the Altera DE1 FPGA board, the processor has a frequency, or, clock speed, of 50MHz. This is the fastest clock speed available for use on the board without a signal generator. Within ModelSim, however, the processor functions properly with a minimum clock high/low cycle period 334ps. This is equivalent to a frequency of 2.994\*10^9 Hz, which is 2.994GHz. So, theoretically the upper stable limit of our processor is just ever so slightly shy of the 3 GHz mark. This is a factor of 59.88 times faster than the fastest provided clock pin from the DE1 board. For most tasks that this processor would be applied to, however, such an increase in clock speed would be of minimal benefit. In fact, due to the lack of a real-time clock, such a high clock speed would be of detriment to desired behavior in various scenarios. One such scenario would be the nightrider game used in the visual demonstration of the board, which depends on delay set through looped branching in order to be able to “capture” a flashing light within possible human reaction time. With a higher clockspeed, the program would necessarily become more complicated to accommodate, and for a program that would rely upon similar timings, one which uses up most of the board’s available resources already, this could create a real constraint on the available 1024 spaces in memory.

BONUS: Instead of using the form of adders detailed in early phase instructions, the processor implements faster addition via carry look-ahead.

With carry look-ahead addition logic, it is more efficient than regular logical addition for up to 4 bits at a time. Knowing this, our adder has four separate carry look-ahead mappings of 4 bits each for maximum efficiency. For a 4 bit carry look-ahead addition function, the logic is as such:

NOTE: s\_(i) is sum bit, x\_(i) is first bit, y\_(i) is added bit, and c\_(i) is initial carry bit

G is gen, P is prop, and the reason they are useful is that the final sum is the initial halfsum (xi XOR yi) XOR CarryIn. The Gen and Prop components allow for the simplified finding of the CarryIn logically.

--s\_(i) = x\_(i) ⊕ y\_(i) ⊕ c\_(i)

--c\_(i+1) = x\_(i)y\_(i) + x\_(i)c\_(i) + y\_(i)c\_(i)

--c\_(i+1) = x\_(i)y\_(i\_ + (x\_(i) + y\_(i))c\_(i)

--c\_(i+1) = G\_(i) + P\_(i)c\_(i)

--G\_(i) = x\_(i)y\_(i) and P\_(i) = x\_(i) + y\_(i)

--c\_(i+1) = G\_(i) + P\_(i)G\_(i−1) + P\_(i)P\_(i−1)c\_(i−1)

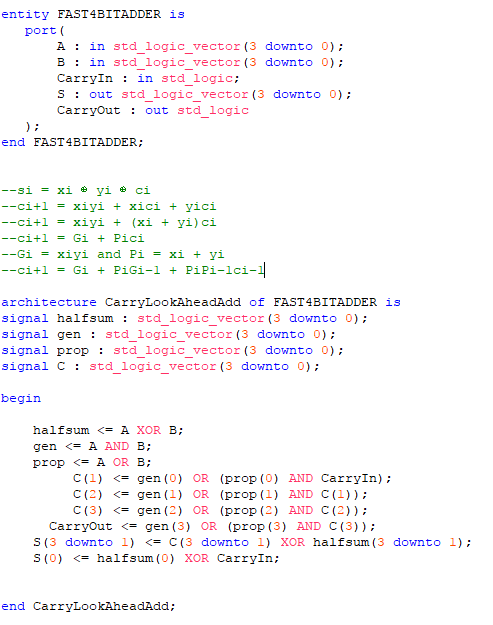


Figure 2: VHDL for the 4-bit Carry Look-Ahead adder implementation.

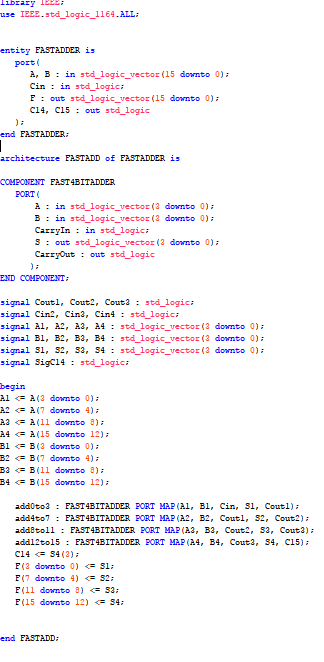


Figure 3: 16 bit full adder implementation, mapping 4 sets of 4 bits to the carry look-ahead component.

# **MIF File**

An example .mif file, discussed above in the *Assembler* section.

WIDTH=24;

DEPTH=1024;

ADDRESS\_RADIX=UNS;

DATA\_RADIX=BIN;

CONTENT BEGIN

0: 000000000000000000000000;

1: 100000000000000000001100;

2: 000000001000000000000000;

3: 000000000100000000000000;

4: 000000000010000000000000;

5: 000000000101000000000000;

6: 000000000110000000000000;

7: 000000000111000000000000;

8: 000000000001000000000000;

9: 000000000011000000000000;

10: 000000001110000000000000;

11: 000000001111000000000000;

12: 000000000000000000000000;

13: 000000000000000000000000;

14: 010000000000101000000010;

15: 010000000000100000000011;

16: 011000000000000100000101;

17: 011000000000000000000110;

18: 010000000000000000100101;

19: 010100000000000000110101;

20: 100000001111111111111101;

[20..1023] : 000000000000000000000000;

# Group Experiences throughout project

Group #9 experienced a lot of trouble along the path to building a processor, but managed to get all checkoffs completed on time, and all parts functional. A full assembler was also created, which aided in progress throughout the project. The demonstration was well put together and made use of all the parts created.

# Conclusion

Building a processor is both easier and more difficult than it sounds when first presented. It is easier once you understand that it is just a set of parts working in conjunction to perform tasks, and each of these parts are relatively simple by themselves. It becomes much more difficult when a time constraint is introduced, making the workload quite immense at times. Making everything compatible proved to be an extremely complex task, fixing one thing always seemed to break another thing.