Semester Project: Basic Processor

[redacted]

CSCE 230
Dr. Riedesel
11/6/2016

Version	Change	Date
1.0	Created initial data path; implemented ALU operations	11/10/16
2.0	Added additional R, D, and B instructions in controller; extended data path; added memory interface	11/17/16
2.1	Added support for conditional operations; added NCZV register to store and appropriately write conditional flag values; added assemblertron9000	11/30/16
3.0	Implemented J instructions in the control unit; Added support for required input/output devices on the Altera boards	12/9/16

Overview

Over the course of the last semester, many additions, improvements and implementations have been combined to create a basic processor. Several components from previous labs were brought together to create a functioning Central Processing Unit. In part II of the project, the CPU could only decode binary instructions, perform operations such as addition and logical algebra and read and store data to and from registers in active memory. Next, an instruction address generator, memory and immediate block were added to the project. These enabled the CPU to adjust the program counter to given values and store values in a memory block. Then, in part IV, a register was created to hold the values of the NCZV flags in order to implement the conditional aspect of the machine code. Pins were also added so that the Altera board could interact with the processor. Finally, in part V, the processor was able to handle all R, D, B and J instructions. It can also fully interact with each input/output device on the Altera board. At each stage, new and old functionality was tested and the CPU's maximum speed was determined. The resulting basic processor can decode machine language, perform logical operations, read and write data to and from memory, adjust the program counter to given values, branch to specified values when conditions are satisfied and can interact with an Altera board.

Processor Design

The processor follows a five stage set up which includes: Fetch, Decode, Execute, Memory and Write Back. Each instruction must undergo each of these stages even if it does not require each of them. For example, addi does not need the memory stage, yet the processor does not skip it, it simply doesn't execute anything in that stage. The processor takes in 24 bit instructions of R, B, J, and D type. The processor has 16 registers although the first can only contain 0 and the 15th is reserved for a return address. These registers can store up to 16 bits of information. The following is a description of the most important components in the processor, only a few were selected to keep this report to a reasonable number of pages.

Control Unit

The only major component written in VHDL, the control unit is the brains of the processor. It has three main purposes: to keep track of which stage the instruction cycle is in, to modify the

data path appropriately based on the current stage and instruction being executed, and to handle the logic which determines if a conditional instruction should execute.
An internal counter which increments clock beat is used to keep track of the stage. The counter starts at 0, and will continue to count from 1 to 5, and then roll over to 1. Only by setting the reset bit to high will the stage go back to 0. Each time the clock comes high, the control unit checks the current stage, and then decodes the instruction's Opcode to determine what the various output flags should be set to.
It keeps track of which stage the processor is in for an instruction. During each stage, the control unit decodes the Opcode of the instruction, and sets up the appropriate data path for that stage and instruction via read/write flags and selecting mux inputs. For R-type instructions, the control unit also decodes from the prescribed opxcode to the native ALU opcode.
It's also given the opx which is used by the alu to manipulate the data. And then, a negative bit, a zero bit, an overflow bit and a carry bit are all received from the ALU. Then, the clock and reset for the processor are passed in as well. Inside the control unit, five stages take place: fetch, decode, execute, memory, write. Then, after the magic, the control unit sends out an alu_op code which is used by the ALU, and other flag bits which control data flow. For example, b_select is used to determine which input is used in a 16 bit 2 to 1 multiplexer.

16 Bit Register Bank

The 16-bit register bank/file is home to 16 16-bit registers. The write register is selected by a value passed into a decoder, as mentioned above. Then, as long as write_enabled is set, a value is passed into the decoder to be held until it is reset. The registers store data by using D flip-flops.

Arithmetic Logic Unit

The ALU performs various data manipulations and boolean logic such as ADD, OR, AND, and XOR. It takes in five inputs, two 16 bit integers, a two bit alu_op code and both an a and b invert value. The 16 bit integers are the two values that will manipulated. The instruction to take place is determined by the alu_op code, as shown below. Finally, with the current implementations b is inverted to subtract the two given integers. Finally, the ALU outputs five values: the result of the manipulation, a negative bit, a zero bit, an overflow bit and a carry bit. All these inputs, except for the result are fed back into the control unit and used for further processes.

AND	OR	XOR	ADD	SUB
00	01	10	11	11

Note: The ADD and SUB codes are the same. This is because a subtract operation is processed identically to an ADD operation, with the difference being the second operand is inverted.

Immediate Block

The immediate block was a given component which allows the padding of smaller bit signals into larger ones. For example, it pads lw and sw immediate values from 7 bits to 16 bits to be used in the ALU. When storing data to memory it extends a 16-bit register value to a 24-bit memory value. By changing the extend flag the behavior can be altered to pad with 1's or 0's.

Instruction Address Generator

The instruction address generator was a given component which holds a program counter (PC) register. The PC contains the memory address of the next instruction to be executed. It is updated to point to the next instruction automatically, but can be fed values via the jump register and b-type instructions. Updating the PC is controlled by two flags, PC_select and PC_enable. In stage two of each cycle, the instruction register will be updated with the new instruction as specified by PC.

Memory

The memory interface was given, and represents a series of 24-bit data places accessed by a 16-bit address. The memory can be modified via the lw and sw commands, and also holds the instructions to be executed. The instruction address generator fetches the instruction to be executed in stage one of all commands. In order to write and read from memory, mem_write and mem_read flags need to be set in the control unit.

Control Unit and Muxes

In order to accommodate the new components and instructions, several more conditional checks were added to the control unit. Most notably was more if and elsif statements to check for D and B type instructions. Additionally, conditionals were added to the various flags associated with the instruction address generator and memory interface. Four new mixes were also added to handle the increase in available data sources. Mux C selects from four inputs to determine what register should be written to based on the type of instruction. Mux Y selects what value should be stored in reg Y, the "write back" buffer. Mux ma selects between the ALU output and the current address outputted by the instruction address generator. Mux B selects between the value in reg B and the immediate value provided by d-type instructions, and feeds the result into the ALU.

I/O Memory Interface

The I/O memory interface was given to us to store pertinent data in registers for the various inputs and outputs on the board. In this phase, functionality was added for 2 outputs (Green LEDs and Hex Display) and 2 inputs (Toggle Switches and Push Buttons). The output being written to and the input being read from can be selected by setting any of the first four bits of the address bus. Additionally, to write to an output, the memWrite bit must be set. All output and inputs are mapped to physical pins on the board, so any change to the registers in this interface will result in/be the result of actual changes in board state.

I/O vs Main Memory Read/Write Logical Gates

In the processor, it does not have separate commands to load/store in main memory and I/O memory. As a result, a data path to accommodate for this had to be designed. The load/store instructions allow for 16-bit addressing, however, the main memory block only has 1024 bytes of memory. Therefore, only 10 bits were needed to completely index this memory. The remaining bits are used to index I/O memory. To ensure that only one memory is being written to at any time, a logic gate to 'or' together the first 4 (I/O) bits together was implemented, and then the result was 'anded' with the write_enable bit. If any of the I/O bits are set and write_enable is set, the write_enable bit of the I/O memory goes high. The negated

version of this signal goes to the main memory. This ensures only one Memory block can be written to or read from at a given time.

Capabilities

Implemented Instructions

All instructions implemented by the cpu are 24 bits, and can be defined as one of four types: R, D, B, and J type.

R Type

Register to register instructions use data stored on registers to perform various arithmetic and memory navigation functions. Three registers are specified; RegS, RegT and RegD. RegS and RegT contain the data to be manipulated, and the result will be stored in RegD. Implemented R Type instructions are:

- Add - sums RegS and RegT and stores the result in RegD
- Sub - subtracts RegT from RegS and stores the difference in RegD
- AND - logically and's each bit RegT[n] with RegS[n] and stores the result in RegD[n]
- OR - logically or's each bit RegT[n] with RegS[n] and stores the result in RegD[n]
- XOR - logically xor's each bit RegT[n] with RegS[n] and stores the result in RegD[n]
- Jr - jumps to the instruction address stored in RegS
- Cmp -compares two register values by subtracting RegT from RegS

R type instructions are written in the following way:

23 22 21 20	19 18 17 16	15	14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
OpCode	Cond	S	opx	RegD	RegS	RegT

- **<u>OpCode[23..20]</u>** - Many R Type instructions have the same OpCode. When performing Add, Sub, AND, OR and XOR all have the same OpCode, while shift left logical, compare, and jump register have separate OpCodes. Add, Sub, AND, OR and XOR all have OpCode 0000. sll, cmp, and jr have OpCodes 0011, 0010, and 0001 respectively.
- **<u>Cond[19..16]</u>** - On the falling edge of the clock in the control unit, the conditional flags are checked in order to determine if the next instruction (j, br, etc.) should be completed. These values are stored in a register until they are needed by the control unit to determine the next instruction.
- **<u>S[15]</u>** - This bit is used to replace the NCZV flags in the register. The value will only be replaced when the S bit is high, otherwise it will hold the value from the last compare. Note: the assembler currently only uses the S-bit for cmp.
- **<u>opx[14..12]</u>** - The opx code specifies the operation to be performed by the ALU. sll, cmp, and jr have the same opx, 000. This is because the OpCode for each instruction is already unique. For other r type instructions:

Instruction	Add	Sub	AND	OR	XOR
opx	100	011	111	110	101

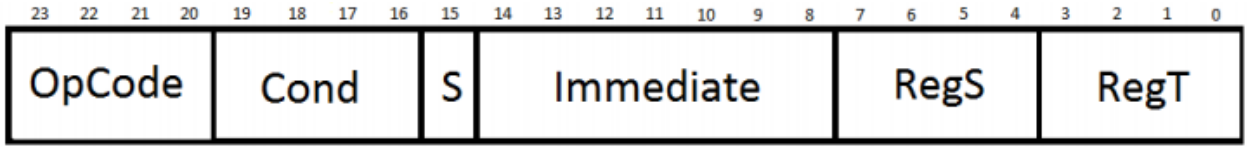
- **RegD[11..8]** - Specifies the write register, that is the register the result of the operation is to be stored in.
- **RegS[7..4]** - Specifies the register first term of the arithmetic operation.
- **RegT[3..0]** - Specifies the second term of the arithmetic operation.

D Type

These instructions represent any data transfer between memory and the CPU. There are only two registers, RegS and RegT. However, there is also an immediate value which can alter the memory address to be stored in or loaded from. Implemented D Type instructions are:

- Lw - Loads the contents of the memory address retrieved from adding RegS and the immediate value
- Sw- Stores the contents of RegT into the memory address retrieved from adding RegS and the immediate value
- Addi- Adds the value in RegS to the immediate value then stores the result in RegT

D type instructions are written in the following way:



- **OpCode[23..20] -** The opcode for lw, sw and addi is 0100, 0101 and 0110 respectively. A check was added for each of these codes was added in the control unit.
- **Cond[19..16]** - On the falling edge of the clock in the control unit, the conditional flags are checked in order to determine if the next instruction (j, br, etc.) should be completed. These values are stored in a register until they are needed by the control unit to determine the next instruction.
- **S[15]** - This bit is used to replace the NCZV flags in the register. The value will only be replaced when the S bit is high, otherwise it will hold the value from the last compare. Note: the assembler currently only uses the S-bit for cmp.
- **Immediate[14..8]** - In the case of a load or store, the immediate value is sign extended to 16 bits and then added to RegS in order to get the correct memory location. For an add immediate, the immediate value is sign extended and added to RegS and then stored in RegT.
- **RegS[7..4]** - The value stored in RegS is added to the immediate value in order to determine the final memory address or value needed for the sw, lw or addi
- **RegT[3..0]** - Specifies the write register, that is the register the result of the instruction is to be stored in.

B Type

These instructions represent any type of branch instruction. Branch instructions change the PC based on a condition. If the condition evaluates to true, the PC is changed, otherwise it remains the same. Implemented instructions are:

- B - Branches to the instruction at PC + 1 + label
- Bal - Stores PC + 1 in R15 and then branches relatively to PC + 1 + label

B type instructions are written in the following way:

23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OpCode	Cond	Label

- **OpCode[23..20] -** Specifies which branch instruction is to be carried out: 1000 for b and 1001 for bal
- **Cond[19..16]** - Specifies the condition that must be fulfilled in order to change the PC
- **Label[15..0] -** Specifies the new relative instruction address should the condition be fulfilled

J type

These instructions implement jump type instructions. A jump instruction is composed of an opCode and an immediate value. These instructions are always implemented; thus they have no conditional portion. When the processor receives a jump instruction, the program counter is set to the given immediate value. Implemented instructions are:

- J - Sets the PC to the immediate value
- Jal- Stores PC + 1 in R15 and then sets the PC to the immediate value
- Li - Loads a 16-bit constant into the specified register

J type instructions are written in the following way:

23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OpCode	Constant

- **OpCode[23..20] -** Specifies which branch instruction is to be carried out: 1100 for j 1101 for jal and 1110 for li
- **Constant[19..0] -** This is the value the PC will be reset to. However, for li, the bits [19..16] are used to determine where the given 16-bit immediate address will be stored.

Input/output Devices

The basic processor can be loaded onto an Altera board where is can execute operations in the given 'mif' file. There are five input/output devices that the processor can interact with.

<u>Push Buttons</u>

On the Altera board, there are a total of 4 push button inputs which are referred to as KEY in the processor. KEY0 is set up as a reset button, meaning that if pressed the entire system is reset and set back to the beginning. These buttons have been inverted so that if pressed, they have a value of one, which the developers felt made the more sense. The buttons send back the value of the buttons pressed, for example if KEY0 is 1, KEY1 is 2, KEY2 is 4 and KEY3 is 8. If one wanted KEY2 to be pressed, he/she would check if the value loaded from the buttons was equal to 4.

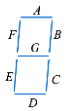
Switches

The switches are another input form on the board. There are a total of 10 switches with indexes 0 through 9. Again, these function with binary values, so SW0 is 1, SW5 is 32 and SW9 is 512 in decimal. So, if one wanted SW2, SW4 and SW7 to be high, the program would check if the value loaded from the switches was 148 in the program.

Green/Red Leds

These are a fairly simple output device. There are a total of ten red leds and 8 green leds. The leds are also represented by binary values. Thus, if the value of 1111 is sent to the red leds, then leds zero through 3 would light up while the rest would stay off.

Hex Display



This is the most complex of the output devices. There are four display units each with seven segments (hence the name Seven Segment Displays). In the following picture, A is the 0 bit, so if the value 1 was sent to the display, only that segment would light up. Thus, B represents 2, C is 4, D is 8 and so on. These displays make it possible to display up to four separate characters at a time.

Processor Speed

Clock speed is a very important component of processor speed. Clock speed is the inverse of the period of the clock (i.e. the amount of time it takes for 1 clock cycle). As more components and instructions have been added, the clock speed has slowed. This can be specifically attributed to the introduction of more complex instructions which required more expensive memory read/write instructions. Currently, the clock period is 1050 picoseconds, this means that the clock speed is $(1050 \times 10^{-12})^{-1}$ hertz or approximately 952 MHz. Because a single instruction takes 5 clock cycles, around 19 million instructions can be completed per second.

Testing

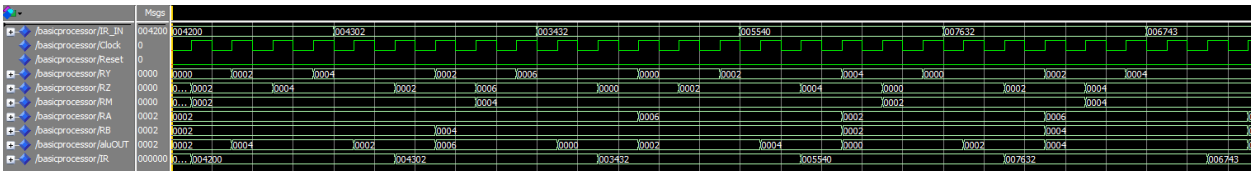
All testing for this project was carried out using ModelSim. 'Do' files were designed to simulate testing instructions. As the processor became more advanced testing was also done on the Altera board. This was especially important when input and output devices were introduced.

Part II

Since only ALU functionality was implemented in this iteration of the processor, only the ADD, SUB, AND, OR, and XOR instructions needed to be tested. This was accomplished by creating a sequence of instructions that utilized all of the above functionality. Instructions built upon one another to insure correctness in data read/write functionality in addition to data processing. The following instructions and do file were used to test this component of the project. The output is also included in the form of a wave diagram. Note: for the sake of testing, the constant assigned to 'r0' was 2.

```
force Reset 0 0
force Clock 0 0, 1 1 -repeat 2
force IR_IN 00000000010000101000000000 0
force IR_IN 00000000010000110000001 0 10
force IR_IN 00000000011101000110010 20
force IR_IN 00000000010101010100000 30
force IR_IN 00000000011101100011001 0 40
force IR_IN 00000000011001110100001 1 50
run 70
```

Add	r2 ,	r0 ,	r0
Add	r3 ,	r0 ,	r2
Sub	r4 ,	r3 ,	r2
Xor	r5 ,	r4 ,	r0
And	r6 ,	r3 ,	r2
Or	r7 ,	r4 ,	r3



As a result of this testing, an error in the implementation of the control unit was found. The control unit output 'b_select' was never being set to 0 in stage 1. As a result, the ALU was never receiving the input from register B (RB) because the multiplexor was receiving an undefined set of select bits. After making this change, all tests were completed successfully.

Part III

Amazingly, there was only one major bug throughout this stage. When implementing the functionality for the branch instruction, the register file's write_enable bit was set too early. As a result, the PC register was being set prior to the writeback stage, which caused its final value to be incorrect.

To test this phase an arrangement of instructions that built upon one another (similar to last phase) were used. Tested instructions include addi, stw, ldw, bal, and add. Please see the web handin for the result of these tests.

Part IV

For phase IV both modelsim and the Altera board were used to test and verify that the added components were working appropriately. During testing two main bugs were found. The first came when trying to use the cmp command to set the NCZV flags for the next conditional instruction. Before implementing the S bit, the NCZV flags were not updating as planned. After implementing the ps_enable and a NCZV flag register to store the values after a cmp or similar operation, based on the S bit, conditionals would execute correctly.

After this the second major bug was found. After a conditional command did not execute due to the conditional failing, as expected, the following command would do nothing and increment PC by 5. This bug was much less straightforward, and was ultimately found to be in how conditionals were prevented from executing. The bug was caused by not executing stage two when the conditional failed. Thus, certain flags such as pc_enable and inc_select were not set to the right value, causing unexpected PC behavior. This was fixed by always allowing stages 1 and 2 to execute, and halting the execution at stage 3 in the event of a failed conditional. Included below are the code and signal used to determine that conditional branches were executing as expected, in both the positive and negative cases.

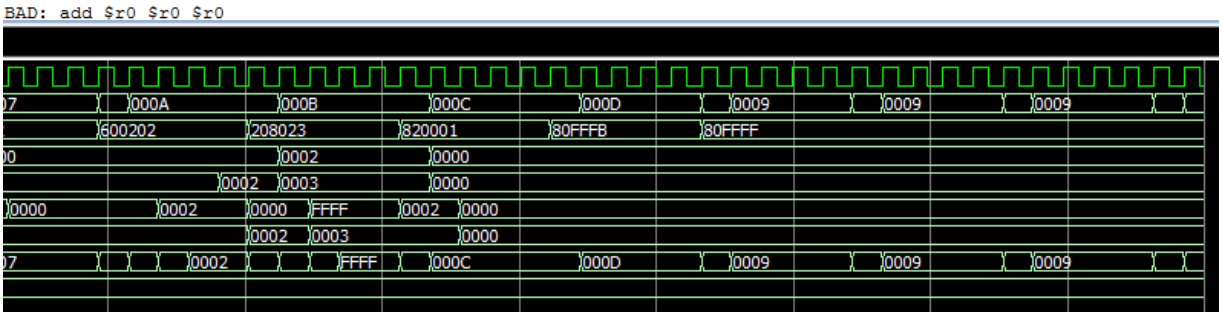
```
b EQUALITY_TEST
b END

EQUALITY_TEST: addi $r2 $r0 3
addi $r3 $r0 3
beq $r2 $r3 TEST_INEQUALITY
    b BAD

END: b END

TEST_INEQUALITY: addi $r2 $r0 2
    beq $r2 $r3 BAD
    b END

BAD: add $r0 $r0 $r0
```

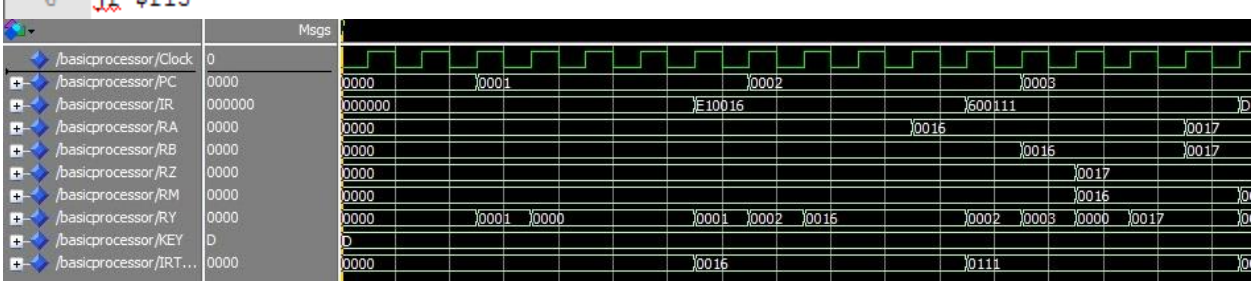


To test the newly added i/o components, .do files were used which forced the values of i/o to a given value. These values could then be used to determine if code was executing as expected. After confirming the processor worked in Altera, the check in was run on the Altera board and amazingly worked.

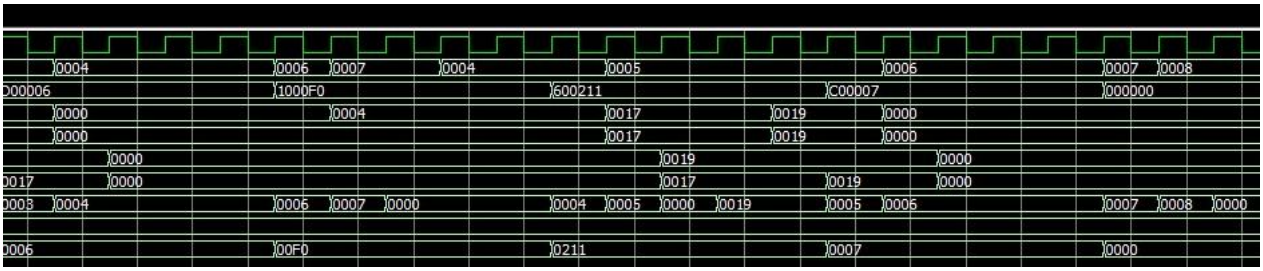
Part V

For part V testing was broken up into two portions. The first section of testing was reserved for ensuring j types worked correctly. Fortunately, testing was smooth and the desired outcome was reached on the first test. The program run and the wave from modelsim are included below

```
1  li $r1 22
2  addi $r1 $r1 1
3  jal 6
4  addi $r1 $r1 2
5  j 7
6  jr $r15
```



.



The section part of testing was to ensure the hex displays, buttons and switches were all working. When the hex displays were first tested, all the values being pushed were inverted once they reached the Altera board. Thus, the input into the displays were inverted to keep the displays working as expected.

Bonus Completed

Assembler

A custom built assembler was written in python to aid in the writing and testing of code. The assembler can write binary and hex values for all non-conditional instructions. So far, only conditional branches have been implemented for the assembler. The assembler can also handle the use of labels for jumps, branches, and load immediates. All immediate values are given in decimal. The assembler supports limited pseudo-code, allowing for conditional branches such as “beq $r1 $r2 NEXT”, which would assemble into two instructions, a cmp between r1 and r2 and a conditional branch to NEXT.

The assembler is a two pass model. The first pass will eliminate pseudo code by substituting key words and generating new lines of assembly, and will perform a first pass label substitution on all original and new lines generated by the pseudo code. The second pass will complete label substitution, and convert the line into hex. The assembler can also write to a .mif.

Group Experiences

Part II

As previously mentioned, we’ve had a fairly smooth ride. Overall, we’ve only had one large bug which was due to incorrectly copying over the VDHL for the control unit. Although frustrating, by talking through the problem together we were able to determine its source and ultimately correct our problem. Any spats that we’ve run into have been ironed out through thoughtful conversation and understanding. Each member of our team has a role, yet is always willing to help other members and gain a better understanding of building a processor as a whole.

Part III

This week went smoothly as far as group relations are concerned. We only had one large bug which was due to prematurely setting a write bit in the control unit. Although frustrating, by talking through the problem together we were able to determine its source and ultimately correct our problem.

Part IV

These last two weeks were by far the most difficult part of the project so far. Between all the extra tests and Thanksgiving break finding time to meet as a group was nearly impossible. As a group, either met during lab time or two of us outside of lab time or just Abbie as she sat alone in the lab room on the Tuesday of Thanksgiving. The biggest problem we faced, was initially not having a register to store the NCZV flags. This mistake was caught by Abbie and quickly corrected. However, the processor still faced a problem which we realized only after lots of guessing and checking.

Part V

This week was extremely similar to part III. Our group worked well together as usual, although it was still difficult to find a chance to meet all together. The only difficulty we faced was when implementing the j types. Initially, we had forgotten to ignore the conditional flags. However, we caught this mistake after one run through in modelsim, which saved us a lot of time and frustration. It was a simple week that we were able to finish within approximately three hours total.

Individual Responses

[redacted]

As the only computer engineering major in this group, I felt relatively comfortable with scope of this project. That being said, there were phases early on in the project where I was a little confused about making changes to the data path. After a little time spent studying outside of group time however, I had a much better grasp of the concept and even began enjoying the project. Although I would not say I took a leading role, I believe I made significant contributions to the end product.

[redacted]

As the only dual computer/electrical engineering major in this group, this class and project were right up my ally. This project called on all the various topics we learned in class, and was a good review and application of them. Smart testing was required, as there is little to no debugging for the hardware. The project was time intensive, but I feel that we were lucky enough to not run into any major problems, and spent at most 10 hours a week on the project. Once everything was finally done, it was quite fun to implement new instructions and make all the various components which made up our final program.

[redacted]

As the only computer science major in the group, I felt a little out of place. I didn't have as much interest in the subject matter as either [redacted] o[redacted] That being said, it was also a bit more difficult for me to follow. In an attempt to increase my comprehension, I primarily did all the driving for the project. That way I could see what was actually happening and ask questions along the way. This honestly helped me immensely, also my partners were very patient and wanted me to succeed with them. They always took my concerns into account and either explained why it wasn't a problem or saw that there was a flaw in the logic. Overall, I feel I did my best to give back to the group and work through our project.

Conclusion

This semester projected proved both difficult and fun. While building a processor was a huge task, the thought and considerations that went into decision-making enhanced the overall understanding of computer components and the data path. The processor was created using different components made in each lab and then continually added to. The CPU went from decoding binary instructions to outputting values on an Altera hex display. Obviously great strides were made in both the processor and the group's comprehension. The resulting processor can perform a multitude of functions including: decoding machine language, performing logical operations, reading and writing data to and from memory, adjusting the program counter to given values, branching to specified values when conditions are satisfied and interacting with an Altera board.

MIF File

The following 'mif' file is from a jump testing file. It was included as it accounts for 5% of the report grade. The remaining lines were all zeros, if you wish to see a more complete example please look at one of our archived files, they surely won't disappoint.

```
WIDTH=24;
DEPTH=1024;
ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;
CONTENT BEGIN
0000 : 000000;
0001 : e10016;
0002 : 600111;
0003 : d00006;
0004 : 600211;
0005 : c00007;
0006 : 1000f0;
```