Final Design Report
Team 1E

16-Bit MRAA

16-Bit Multi-Register Accumulator Architecture

By: Russell Johnson, Griffin Steffy,

Charley Beeman, Stella Park

Table of Contents

Executive Summary

The 16-bit MRAA is designed to create an optimal balance between usability and performance. The design philosophy which allowed us to achieve this was to minimize the processor area and complexity of software programming while maximizing the clock speed. Simplifying software design using our instruction set was achieved with a primary accumulator register and several supplementary registers to allow for ease of data manipulation. By following these principles we created a functional, efficient, and usable processor.

Introduction

The MRAA (Multi-Register Accumulator Architecture) is a very user-friendly processor. It can handle a variety of different tasks including running high-level programs with nested and parameterized functions as well as general arithmetic and logical calculations. The processor has the ability to be integrated onto an FPGA where it can take in a user-specified input as well output to an LCD. It was designed to optimize instruction count, while maintaining a relatively small footprint in hardware. Even with speed not being a priority, we were able to optimize our design to fit a 93.7 MHz clock, which is fast for the type of architecture we chose to implement.

Instruction Set Design

We decided to design a modified accumulator processor with 3 different instruction types: *I-type*, *R-type,* and *M-type*. This design minimizes the information needed for each instruction which maximizes the number of bits available for immediates in *I-type* instructions. All instructions have an op-code. *I-types* use a 12-bit immediate value and include arithmetic, logical, and jump instructions. The *jump* and *jump and link* instructions jump to the specified immediate; *jal* also stores the address of the next line into the return address register. The *jump0* and *jump1* instructions are used to branch in the program based on the result from the previous instruction by checking the most significant bit of the accumulator register. *R-types* have a function code in order to allow for more instructions. Results of all *R-type* instructions can be stored in either the accumulator register or a target register by setting the least significant bit of the instruction. This design choice reduces the number of instructions needed for arithmetic and logical operations, since the user won't need to use a move instruction after each computation. Also, we created a *subr R-type* instruction in addition to *sub*. This added instruction subtracts the value stored in the accumulator register from the specified register. This reduces the number of instructions needed to perform subtractions. The *M-type* instructions are memory instructions as well as *jump register* which jumps to the address stored in the specified register. This functions similarly to *I-type* instructions except it uses an 8-bit immediate used to offset an address stored in the available register.

Registers

Our design has 16 dedicated registers: zero, stack-pointer, return-address, iszero, (4) saves, (5) temporaries, in and out registers, along with acc (Accumulator). If the output of the ALU results in a 0, then $iszero is set to 0xFFFF, otherwise, it is set to 0x0000. The iszero register is primarily used for jump0 and jump1 which are based on the results of the previous instruction.

Implementation

Our focus on ease of use for the programmer as well as minimizing the time each instruction takes created a more complicated hardware design. We decided on a multi-cycle RTL implementation in order to decrease the length of each cycle and allow instructions to have variable execution times so that less complex instructions that are used frequently, such as *move*, can execute extremely fast. The multi-cycle implementation created complexity in our control unit due to timing issues with the memory and other components. Primarily, the instructions were appearing in the Instruction Register a cycle too late. To overcome this, we used a multiplexer to allow us to access the instruction as soon as they appeared from memory before they were written to the Instruction Register. Other timing issues were based in issues with delays in memory. We first attempted to overcome these issues by running the memory at double the clock speed compared to the rest of the processor. We found this dramatically reduced the processor's overall clock speed. Instead we implemented delay cycles where needed which, due to their limited frequency, reduced our overall execution time. We also overcame these memory limitations by reading the instructions from memory at the end of instructions instead of the beginning. This reduced the delay required by each instruction by one cycle. In order to implement this efficiently, we used Dual-port memory to allow us to read and write from memory in the same cycle, so that we could store data in the same cycle as we read the next instruction. While likely not the most hardware efficient design, we used case statements and a pointer variable dictating the state in the finite state machine for our control unit. This made the control unit far easier to implement and adjust without negatively affecting performance as the control unit was not involved in the critical path of our processor regardless. We also found using block memory instead of distributed memory increased the overall clock speed of our processor at higher memory sizes.

Testing methodology

Testing was very important as we moved through the design process of our architecture. Starting from when we were creating parts, all the way through to our final working design. We wrote exhaustive tests that also tested as many edge cases that we believed existed for each part and each step of the integration plan. This became very useful for the integration steps, because it allowed us to confidently ensure that at the end of each step our implementation was working properly. As we got to the later testing phases, we often changed or updated parts.

When this would occur we updated the test and reverified that specific part to provide insurance for our completed integrations.

Final results

The MRAA processor was very efficient and exceeded expectations in performance. It only required 134 bytes of space for both the Relative Prime program as well as the stack. Relative Prime took a total of 102,111 instructions that were performed throughout 408,458 cycles with a total execution time of 4.358 ms. The final design had a clock speed of 93.733 MHz as well as a cycle time of 10.669 ns.

Conclusion

Throughout the course of this project, we developed a robust processor design which followed our initial design principles as well as the skills required to complete an intensive long term project. We developed a processor which fulfilled our goals of minimizing processor area and programming complexity while maximizing our clock speed. Under the given constraints, our processor has excellent performance and flexible capabilities. By designing a processor from the bottom up we not only learned how computers function at a hardware level, but also the design process required to develop a complex component within given constraints and a strict timeframe. Using an iterative and procedural design process, we started with the abstract design and functionality, then learned to implement our ideas. A large part of the design process was adapting our design and ideas as new challenges arose. We also learned to develop several possible implementations and asses their merits within our design philosophy. We had a largely successful project and a strong cohesive group for our first foray into higher level design.

Design Document

Instructions

Throughout this section, $acc refers to the dedicated accumulator register, $reg refers to whatever register is being used for the instruction call, and imm represents the immediate value used for the instruction call.

M-type

These instructions are used for handling memory operations: load and store. It is also used for jr (jump register). The immediate is used as the offset, by adding it to the value in the register.

Opcode (4 bits)	Register (4 bits)	Immediate (8 bits)

Instruction	Description	Syntax	Opcode
load	$acc = Mem[$reg + SE(imm)]	load $reg, imm	0x0
store	Mem[$reg +SE(imm)] = $acc	store $reg, imm	0x1
jr	Jump to address in register ($reg + x)	jr $reg, imm	0x2

I-type

These instructions are used for handling immediates. This includes loading to and from memory, jumping to different addresses, along with doing some simple logical and arithmetic operations involving those immediates. The lui instruction loads the upper 4 bits of the immediate since an I-type can only use a 12-bit immediate; therefore, when combined you get a 16-bit immediate value.

.

Opcode (4 bits)	Immediate (12 bits)

Instruction	Description	Syntax	Opcode
jump	Jump to address	jump imm	0x3
jal	Jump and link	jal imm	0x4
jump1	Jumps if the MSB of acc is 1 (acc < 0)	jump1 imm	0x5

jump0	Jumps if the MSB of acc is 0 (acc >= 0)	jump0 imm	0x6
shiftl	acc = acc<<imm	shiftl imm	0x7
shiftr	Acc = acc>>imm	shiftr imm	0x8
lui	Loads upper 4 bits of imm into the upper 4 bits of $acc	lui imm	0x9
ori	Ors imm with the lower 3 bytes of $acc, stores in $acc	ori imm	0xA
andi	Ands imm with $acc, stores in $acc	andi imm	0xB
loadi	Loads immediate into $acc	loadi imm	0xC
addi	Add immediate to $acc register	addi imm	0xD
X	X	X	0xE
R TYPE	This opcode is reserved for R-Type instructions		0xF

Ex. Machine code

```
loadi 0xFFF % $acc <- 0xFFF
1100 1111 1111 1111
```

R-type

These instructions are used for handling registers. This includes moving the contents of a register to the accumulator as well as from the accumulator to a register. Also, there are instructions for adding, subtracting, anding, and oring registers with the accumulator. The user has the ability to choose whether to store the result in the accumulator or the register based on the “to acc” bit value (1 = acc, 0 = reg). All R-types have an opcode of 0xF, and the func code is used to determine which function is being used in the ALU.

Opcode (4 bits)	Register (4 bits)	Func (3 bits)	XXX (4 bits)	to acc (1 bit)

Instruction	Description	Syntax	Opcode	Func
move	Moves from $acc to temp reg or from temp reg to $acc depending on the “to acc” bit (1 = acc, 0 = reg)	move $reg, 1 move $reg, 0	0xF	0x0
add	Adds value of the accumulator register to the value of another specified register, result is stored in accumulator or register based on “to acc” bit	add $reg, 1 add $reg, 0	0xF	0x1
sub	Subtracts the value from a register from the value of the accumulator “acc-t0”, and the result is stored in the accumulator or register based on “to acc” bit	sub $reg, 1 sub $reg, 0	0xF	0x2
subr	Subtracts the value from the accumulator from the value of the register “t0-acc”, and the result is stored in the accumulator or register based on the “to acc” bit	subr $reg, 1 subr $reg, 0	0xF	0x3
or	acc||reg is stored in acc or reg based on “to acc” bit	or $reg, 1 or $reg, 0	0xF	0x4
and	acc&reg is stored in acc or reg based on “to acc” bit	and $reg, 1 and $reg, 0	0xF	0x5

Ex. Machine code

```
add $t0 1      # $acc <- $acc + $t0
1111 1000 0001 0001

move $t0 0    # $t0 <- $acc
1111 1000 0000 0000
```

To $acc: The “to acc” bit determines where the result is stored for the operation. If the bit is set to 1, the result from the operation will be sent to the acc register. If the result is 0, it will be stored in the specified register.

Registers

Register	Description	Hex
$0	Holds a constant value of 0	0x0
$sp	Points to the last active position on the stack	0x1
$ra	Holds return address	0x2
$iszero	Holds a 1 in the MSB if the last ALU operation resulted in 0 $iszero = SE[iszero ALU flag]	0x3
$s0-$s3	Saved Temporary Registers	0x4-0x7
$t0-$t4	Temporary registers	0x8-0xC
$in	Used for accessing hardware input	0xD
$out	Used for outputting to hardware	0xE
$acc	Accumulator register	0xF

Procedure Call Conventions

- Arguments and return values are stored in temporary registers as defined in the procedure being called
- $ra represents the return address, and will need to be backed up on the stack before procedure calls are made.
- The saved temporary registers ($s0-$s3) must be backed up before being used and restored after
- Depending on the use of the program the user might also need to backup the temporary register(s) to the stack as well.

Addressing Modes

Relative-to-PC Addressing:
This will be used by jump0 and jump1 of the addressing I-type instructions. PC-relative addressing will sign extend the 12-bit immediate.The result of that operation will be added to PC (PC+1) and stored back into PC.

PC = (PC+1) + [SE(12-bit imm)]

Register+Offset Addressing:
This addressing mode will be used for M-type instructions. The address will be the offset immediate added to value stored in the in the register specified.

Address = $reg + SE(imm)

Pseudo-Direct Addressing:
This is used for the jump and jal instructions. It will set the value of PC to the immediate shifted left 1 appended to the top 4 bits of PC.

$PC = PC_{15\text{-}12} + (imm)[11:0]$

Extended-Direct Addressing:
This mode will be used in logical and arithmetic instructions. Depending on the type of instruction the program will either zero extend or sign extend the 12-bit immediate of the I-Type instruction. If it is an arithmetic instruction it will be sign extended, and if it is a logical instruction it will be zero extended.

Relative Prime & Euclidean Algorithm - Assembly & Machine Code

Assembly	**Machine Code**
# Move Set Input as the argument	
move $in, 1 # $acc = $in	1111110100000001,
move $t0, 0 # $t0 = $acc	1111100000000000,
# During Execution	
# s0 → n	
# s1 → m	
relPrime:	
# Backup $ra, $s0, $s1	
move $sp, 1 # $acc = $sp	1111000100000001,
addi -3 # $acc = $acc - 3	1101111111111101,
move $sp, 0 # $sp = $acc	1111000100000000,
move $ra, 1 # $acc = $ra	1111001000000001,
store $sp, 0 # Mem[$sp + 0] = $acc	0001000100000000,
move $s0, 1 # $acc = $s0	1111010000000001,
store $sp, 1 # Mem[$sp+1] = $acc	0001000100000001,
move $s1, 1 # $acc = $s1	1111010100000001,
store $sp, 2 # Mem[$sp+2] = $acc	0001000100000010,
move $t0, 1 # $acc = $t0	1111100000000001,
move $s0, 0 # $s0 = $acc	1111010000000000,
loadi 2 # $acc = 2	1100000000000010,
move $t1, 0 # $t1 = $acc	1111100100000000,
move $s1, 0 # $s1 = $acc	1111010100000000,
loop:	
jal gcd # goto gcd	0100000000100111,
addi -1 # $acc = $acc - 1	1101111111111111,
move $iszero, 1 # $acc = $iszero	1111001100000001,
jump1 exitloop # goto exitloop if $acc[15] = 1	0101000000000111,
move $s1, 1 # $acc = $s1	1111010100000001,
addi 1 # $acc = $acc + 1	1101000000000001,
move $s1, 0 # $s1 = $acc	1111010100000000,
move $t1, 0 # $t1 = $acc	1111100100000000,
move $s0, 1 # $acc = $s0	1111010000000001,
move $t0, 0 # $t0 = $acc	1111100000000000,
jump loop # goto loop	0011000000010000,
exitloop:	
# Put m into output reg	
move $s1, 1 # $acc = $s1	1111010100000001,
move $out, 0 # $out = $acc	1111111000000000,
# Restore $ra, $s0, $s1	

```
        load $sp, 2          # $acc = Mem[$sp+2]                    0000000100000010,
        move $s1, 0          # $s1 = $acc                           1111010100000000,
        load $sp, 1          # $acc = Mem[$sp+1]                    0000000100000001,
        move $s0, 0          # $s0 = $acc                           1111010000000000,
        load $sp, 0          # $acc = Mem[$sp+0]                    0000000100000000,
        move $ra, 0          # $ra = $acc                           1111001000000000,
        move $sp, 1          # $acc = $sp                           1111000100000001,
        addi 3               # $acc = $acc + 3                      1101000000000011,
        move $sp, 0          # $sp = $acc                           1111000100000000,
        jump finish          # goto end of program                  0011000000111110,
gcd:
        move $t0, 1          # $acc = $t0                           1111100000000001,
        addi 0               # $acc = $acc + 0                      1101000000000000,
        move $iszero, 1      # $acc = $iszero                       1111001100000001,
        jump0 gcdloop        # goto gcdloop if $acc[15] = 0         0110000000000011,
        move $t1, 1          # $acc = $t1                           1111100100000001,
        move $t2, 0          # $t2 = $acc                           1111101000000000,
        jr $ra, 0            # goto address in $ra+0                0010001000000000,
gcdloop:
        move $t1, 1          # $acc = $t1                           1111100100000001,
        addi 0               # $acc = $acc + 0                      1101000000000000,
        move $iszero, 1      # $acc = $iszero                       1111001100000001,
        jump1 exitgcdloop    # goto exitgcdloop if $acc[15] = 1     0101000000001001,
        move $t1, 1          # $acc = $t1                           1111100100000001,
        sub $t0, 1           # $acc = $acc - $t0                    1111100001000001,
        jump0 else           # goto else if $acc[15] = 0            0110000000000011,
        move $t1, 1          # $acc = $t1                           1111100100000001,
        subr $t0, 0          # $t0 = $t0 - $acc                     1111100001100000,
        jump gcdloop         # goto gcdloop                         0011000000101110,
else:
        move $t0, 1          # $acc = $t0                           1111100000000001,
        subr $t1, 0          # $t1 = $t1 - $acc                     1111100101100000,
        jump gcdloop         # goto gcdloop                         0011000000101110,
exitgcdloop:                                                        1111100000000001,
        move $t0, 1          # $acc = $t0                           1111101000000000,
        move $t2, 0          # $t2 = $acc                           0010001000000000,
        jr $ra, 0            # goto address in $ra + 0
finish:
```

Assembly Language Fragments

Description	Assembly	Machine Code
Loading an address	lui 0x0A00 # $acc = 0xA000 ori 0x0BCD # $acc = 0xABCD jr $acc, 0 # jump to address in $acc	1001101000000000 1010101111001101 0010111100000000
Iteration	addi 1	1101000000000001
Cond:$t0 > $acc	sub $t0, 1 jump1 label	1111100000100001 0101xxxxxxxxxxxx
Cond: $t0 < $acc	subr $t0, 1 jump1 label	1111100000110001 0101xxxxxxxxxxxx
Cond: $t0 <= $acc	sub $t0, 1 jump0 label	1111100000100001 0110xxxxxxxxxxxx
Cond: $t0 >= $acc	subr $t0, 1 jump0 label	1111100000110001 0110xxxxxxxxxxxx
Cond: $t0 == $acc	sub $t0, 0 move $iszero, 1 jump1 label	1111100000100000 1111001100000001 0101xxxxxxxxxxxx
Cond: $t0 != $acc	sub $t0, 0 move $iszero, 1 jump0 label	1111100000100000 1111001100000001 0110xxxxxxxxxxxx
Read Data From Input	move $in, 1	1111110100000001
Write Data to Output	lui 0x0A00 ori 0x0BCD move $out, 0	1001101000000000 1010101111001101 1111111000000000

RTL

Cycles	Multicycle RTL						
	M-Type			R-Type			
	JR	Load	Store	add, sub, subr, or, and		move	
				toacc = 1	toacc = 0	toacc = 1	toacc = 0
0 (reset)	IR = Mem[PC] (reset state)						
1	PC=PC+1						
2	A = SE[IR[7:0]] B= Reg[IR[11:8]]			A = Reg[0xF] (acc) B = Reg[IR[11:8]]			
3	ALUout = A +B A = Reg[0xF] (acc)			ALUout = A op B		Reg[0xF] (acc) = B	Reg[IR[11:8]] = A
4	PC=ALUout	M=Mem[ALUout]	Mem[ALUout] = A	Reg[0xF] (acc) = ALUout	Reg[IR[11:8]] = ALUout	IR = Mem[PC]	
5	*Jump Delay Cycle*	*Memory Delay Cycle*	IR = Mem[PC]				
6	IR = Mem[PC]	Reg[0xF] (acc)=M					
7		IR = Mem[PC]					

Cycles	Multicycle RTL					
	I-Type					
	Sign Extend		Zero Extend			
	Jump0, and Jump1	**Arithmetic Inst.**	**Logical Inst.**	**Jump Inst.**	**lui**	**jal**
0 (reset)	IR = Mem[PC] (reset state)					
1	PC=PC+1					
2	A = Reg[0xF] (acc) B = SE(IR[11:0]) if (IR[15:12] == 0x6 && acc[15] = 0) || (IR[15:12] == 0x5 && acc[15] = 1) then PC = PC + SE(IR[11:0])		B=ZE(IR(11:0)) A = Reg[0xF] (acc)	PC = PC[15:12]+ZE(IR[11:0])	Reg[0xF] (acc) = ZE(IR[11:8])<<12	Reg[0x2] = PC PC = PC[15:12]+ZE(IR[11:0])
3	*Jump Delay Cycle (If Branching)* *Otherwise Straight to Cycle 4*	ALUout = A op B		*Jump Delay Cycle*	IR = Mem[PC]	*Jump Delay Cycle*
4	IR = Mem[PC]	Reg[0xF] (acc) = ALUout				IR = Mem[PC]
5		IR = Mem[PC]				
6						
7						

General Components Specifications

ALU ✔

Description:

Takes in 2 16-bit inputs and performs an operation. The 16-bit output goes to ALUout. Operations: pass_b (000), add (001), sub (010), subr (011), or (100), and (101), lsift (110), rshift (111)

It also outputs a 1 bit value that is determined by if the output was 0 or not (stored in $iszero register).

Input:

2 16 bit values

Output:

1 16 bit value

2 1 bit signals

Control Signals:

ALUctrl (3 bits)

RTL Symbols:

op

Building:

The ALU will consist of 2 16 bit input data busses, 2 bit input control signal, 1 16 bit output data bus, and 1 bit output control signal (iszero, overflow). We will use the control signal to decide which output of the operations to use via muxes. The operations that can be performed will be implemented through blocks of each type of operation. We will aslo

Testing:

We will test each function of our ALU individually. Since we know that each of those components will work we can specify edge cases for each type of operation and test those. We will also test each input for the ALU ctrl and ensure that it corresponds to the correct type of operation. We will test a case that results in 0.

<u>Adder</u> ✔

Description:

Takes in 2 16-bit inputs, adds them together creating a 16 bit output

Input:

2 16 bit values

Output:

1 16 bit value

Control Signals:

N/A

RTL Symbols:

+

Building:

It will have two 16 bit input data busses as well as one 16 bit output data bus. First we will design a 1 bit adder, then extend that block to become a 4 bit adder. We'll extend that 4 bit adder to a 16 bit adder that takes in our 2 16 bit data busses and outputs the 16 bit data on the bus.

Testing:

We will specify certain 16 bit values that could cause our adder to fail, and test them with our adder. Our edge case that we will test is when there is a carry on the MSB of the output.

<u>Memory and Instruction Data</u> ✔

Description:

Takes in a 16-bit address of a location in memory as well as 16-bits of data. If MemWrite=0, then the input address is read. If MemWrite=1, then the input data is written to the input address.

Input:

3 16 bit values

Clock

Output:

1 16 bit value

Control Signals:

MemWrite (1 bit)

RTL Symbols:

Mem[address]

Building:

This will have 3 16 bit input data busses (Read Address, Write Address, and Write Data) as well as 1 16 bit output data bus. The memory unit itself will be built using Xilinx's core generator. We will use distributed memory to optimize performance.

Testing:

We will test this by writing data to specific addresses in memory and reading it to verify the correct data was written.

Reg File ✔
Description:

Takes in 1 4-bit register address, the 4-bit address of the register to write to, and 16-bits of data to write. If RegWrite=1, then the data is written to the register specified to write to. The data in the register that is input to the Reg File as well as the data in the acc register are output (16 bits each).

Input:

2 4 bit values, 2 16 bit values

Output:

3 16 bit values

Control Signals:

RegWrite (1 bit)

RTL Symbols:

Reg[address]

Building:

We will have all of the registers connected with a 32 16-bit bus to a 16-bit bus output mux o select which is being read from, and another similar mux to determine which is being written to.

1 16 bit input data bus will be for the input (external)

1 16 bit output data bus will be for output (external)

Testing:

By inputting a 16-bit data signal and the address of the register we will write to a specific register and the we will select the same mux to read it to see if it properly stored the value. We will also check to ensure none of the other registers were modified.

Sign Extenders ✔
Description:

Takes a 12 bit or 1 bit input and sign extends it to 16-bits

Input:

1 12 bit value or 1 1 bit value

Output:

1 16 bit value

Control Signals:

N/A

RTL Symbols:

SE(value)

Building:

We will have a 12 bit or 1 bit data bus, and through verilog, we will extend the top bit to the

upper 4 bits of the 16 bit data bus

Testing:

By inputting a 12 bit or 1 bit value we will test that the 16 bit value has sign-extended properly, Ensuring that the decimal value of our input has not changed whether the input is a positive or negative number.

Zero Extenders ✔

Description:

Takes a 12-bit or 4 bit input and zero extends it to 16-bits

Input:

1 12 bit or 4 bit value

Output:

1 16 bit value

Control Signals:

N/A

RTL Symbols:

ZE(value)

Building:

We will have 4 or 12 bit data bus, and through verilog, we will insert 0s as the upper 4 or 12 bits of the 16 bit data bus

Testing:

By inputting a 12 bit value we will test that the 16 bit value has zero-extended properly by using. We will test if the 12-bit data bus's MSB is a 1 or 0 and ensure that it still extends only 0s to the upper 4 or 12 bits of the output.

Left Shift 12 ✔

Description:

Takes a 16-bit input and left shift it by 12 creating a 16-bit output

Input:

1 16 bit value

Output:

1 16 bit value

Control Signals:

N/A

RTL Symbols:

<<4

Building:

We will have 16 bit data bus and using verilog we will move the lower 4 bits of the input to the upper 4 bits of the output, and insert 0s in the lower 12 bits

Testing:

We will input a 16 bit value and after processing it through, we will see that it has been left shifted by 12 properly. We will test to make sure the last 12 bits ([11:0]) of the output are set to 0.

<u>Mux 16 bit (Multiple Input) 1 Output</u> ✔

Description:

Take in up to 6 inputs (in our datapath) and using a control signal chooses one of those inputs to become the output

Input:

2, 3, 4, and 6 16 bit values

Output:

1 16 bit value

Control Signals:

Depending on the mux it is either a 1, 2, or 3 bit signal. Each signal is specific to the mux. *Signals:* IorM, ItypeSel, Asel, Bsel, jcontrol, destData,

RTL Symbols:

N/A

Building:

It will have 2, 3, 4, or 6 16 bit input data buses and 1 16 bit output data bus. The 1, 2, or 3 bit control signal determines which input to choose.

Testing:

We will go through all possibilities of the control signal and make sure the output is what is expected for that specific control signal.

<u>Mux 4 bit (3 Input) 1 Output</u> ✔

Description:

Take in 3 4 bit inputs and using a control signal chooses one of those inputs to become the output

Input:

3 4 bit values

Output:

1 4 bit value

Control Signals:

1 2 bit control signal, destAdd

RTL Symbols:

N/A

Building:

It will have 3 4 bit input data busses and 1 4 bit output data bus. The 2 bit control signal determines which input to choose.

Testing:

We will test the mux with each value for the control signal and make sure that the output is what we expect.

<u>Control Unit</u> ✔

Description:

Takes in 2 4-bit signals, 1 1 bit value, and possibly the MSB of the Accumulator Register and determines the control signals for the rest of the datapath

Input:

1 4 bit value (opcode)

1 4 bit value (func code)

5 1 bit values (clock, reset, toacc (IR[0]), MSB bit of $acc register, noOp (If IR = 0, noOp = 1))

Output:

13 1 bit values

2 2 bit value

Control Signals:

N/A

RTL Symbols:

Control signals are listed in the **Datapath** *section*

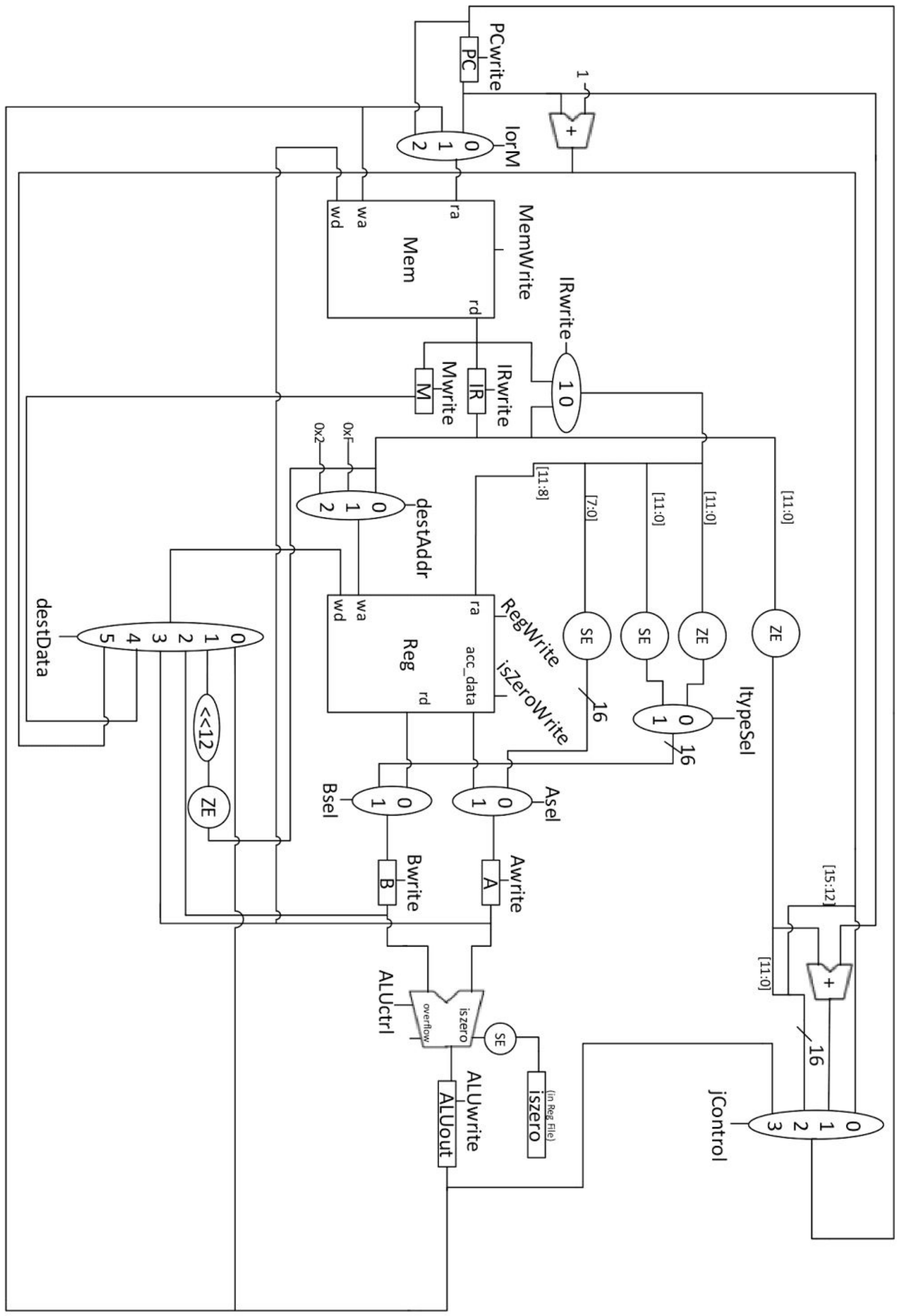
Testing RTL

- In order to test the RTL that we created, we picked instructions out to test every scenario. The instructions chosen were load, jr, store, [sub, subr, & move with toacc = 1], [sub, subr, & move with toacc = 0], jump, ori, addi, jump0, jump1, and lui. We made a test example for each instruction shown in the table below. We then verified each one worked. After testing each instruction, we determined our RTL works as expected.

Instructions Tested

Test Code for Instruction	**Outcome**
load $t0, 4	Success!
jr $t0, 0	Success!
store $t0, 4	Success!
sub $t0, 1 sub $t0, 0	Success!
move $t0, 1 move $t0, 0	Success!
jump label	Success!
ori 0xABC	Success!
addi 0xABC	Success!
jump0 label	Success!
Jump1 label	Success!
lui 0xA00	Success!

Datapath


PCwrite
PC
1
+
IorM
0
1
2
MemWrite
Mem
ra
wa
wd
rd
IRwrite
1 0
IRwrite
IR
Mwrite
M
0xF
0x2
destAddr
0
1
2
[11:8]
[7:0]
[11:0]
[11:0]
[11:0]
RegWrite
Reg
ra
wa
wd
acc_data
rd
isZeroWrite
SE
SE
ZE
ZE
16
ItypeSel
0
1
16
destData
0
1
2
3
4
5
<<12
ZE
Asel
0
1
Bsel
0
1
Awrite
A
Bwrite
B
ALUctrl
overflow
iszero
SE
(in Reg File)
iszero
ALUwrite
ALUout
[15:12]
[11:0]
+
16
jControl
0
1
2
3

List of Control Signals

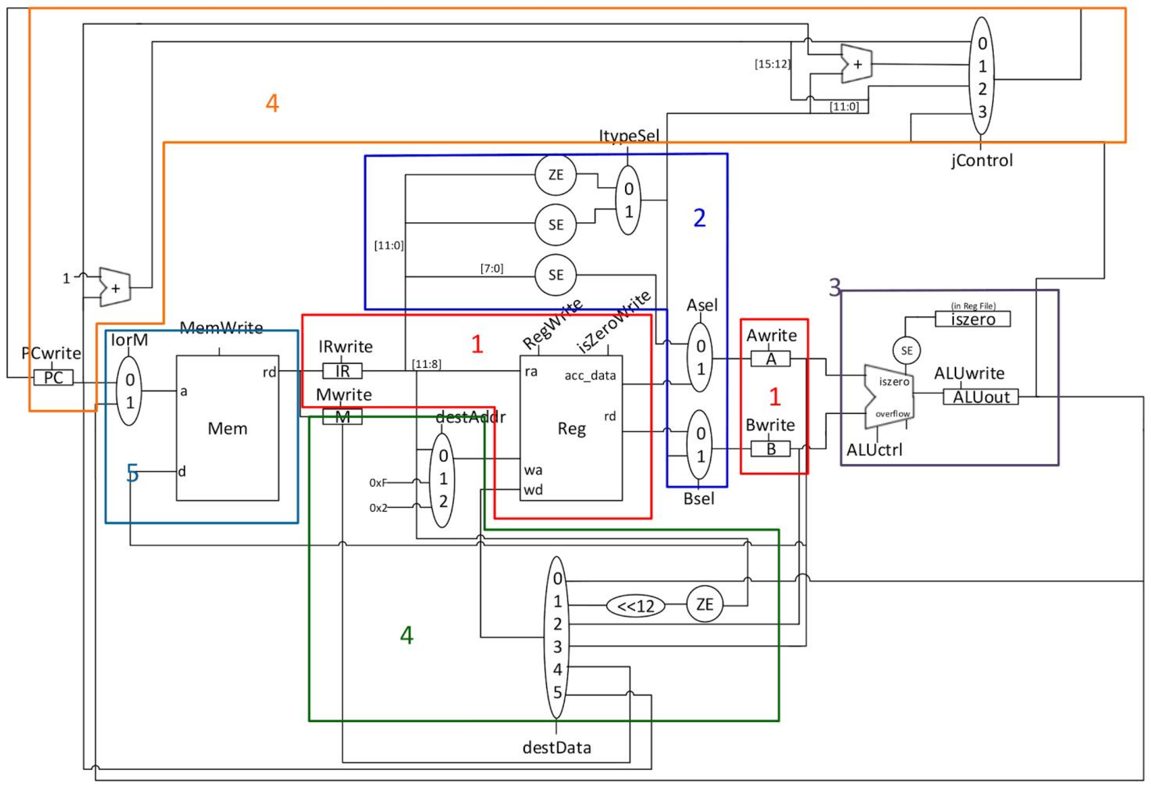
- PCwrite - Determines whether or not you write a new value to the PC
- IorM - Determines what you are fetching from memory (Instruction or other memory data)
- MemWrite - allows writes from memory
- IRwrite - Determines whether or not you write the instruction to IR
- Mwrite - determines whether or not to write to M
- destAddr - Comes from the LSB in instruction and if the instruction is a jal or not goes to $acc register or another register
- destData - Determines what data can be written the to Register File
- ItypeSel - Chooses between a ZE and SE immediate
- Asel - Chooses which value is stored in the A register
- Bsel - Chooses which value is stored in the B register
- Awrite - Enables the A register to be written to
- Bwrite - Enables the B register to be written to
- RegWrite - Allows the Register File to be written to
- isZeroWrite - Allows the ALU to write to the isZero register
- ALUctrl - Controls the operation for the ALU
- ALUwrite - determines whether or not to write to ALUOut
- jControl - Determines what the new PC Value will be

Integration of Datapath Plan

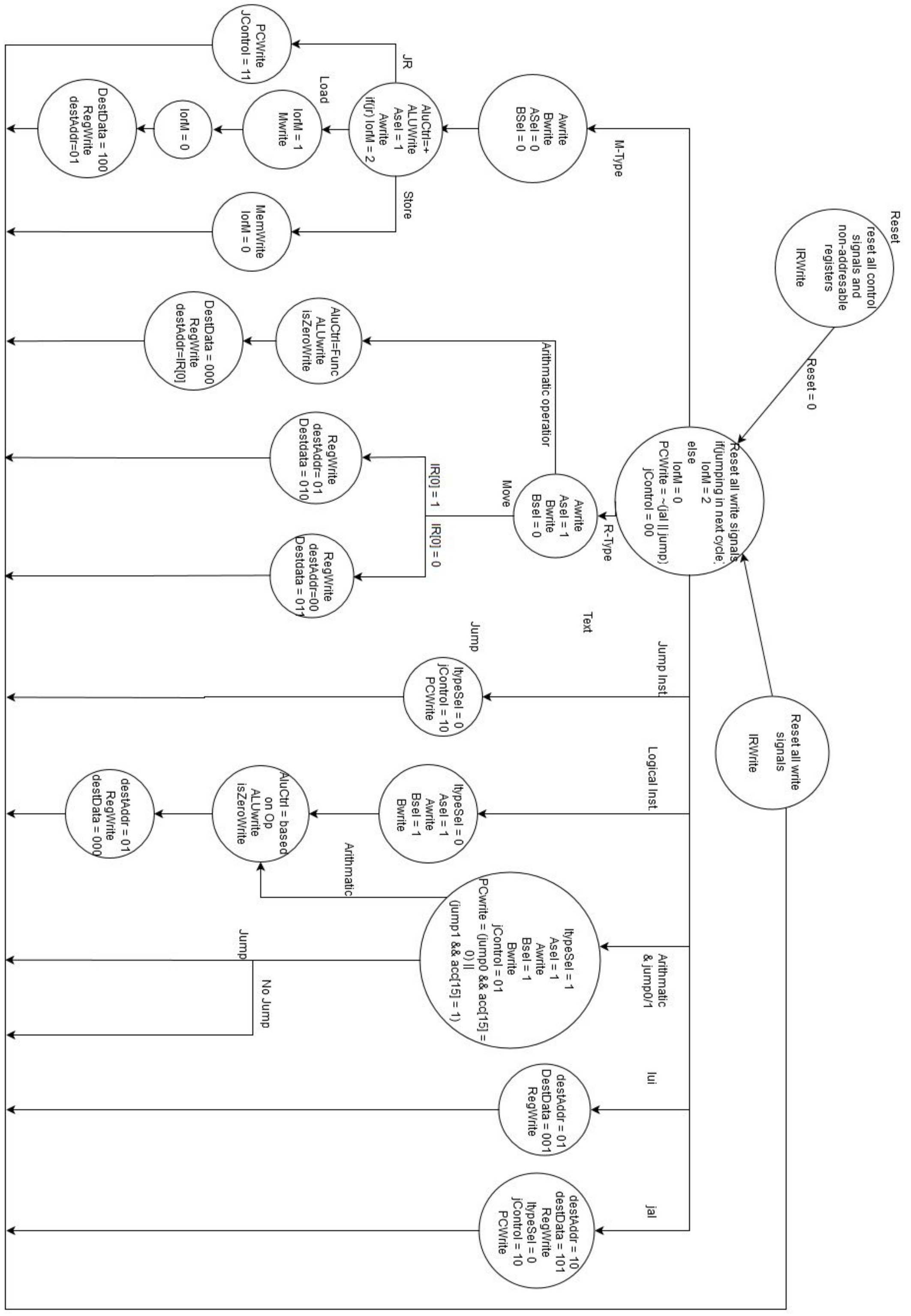
1. Add a RegFile, IR, A, & B
    - Test that the RegFile properly reads the register specified from IR and stores the ACC register data in A as well as the specified register's data in B
    - Test that the RegFile properly writes data to a register by setting the "wa" input to RegFile to acc's address and the "wd" input to the RegFile to a constant 16-bit value. Then read acc and verify that A holds the new value.
2. Add 8-bit SE, Asel mux, Bsel mux, 16-bit SE and ZE, Imm mux
    - Test that we can properly choose the value that gets stored in A and B correctly based off the immediates that come from the Instruction or the data being read from the register file
3. Add ALU, iszero SE, ALUOut register
    - Test that we can operate on the ALU with the values in the A and B registers
    - Test that we can detect the if the operation ended in 0 and store that in the iszero register.
    - Test that we can properly write the output of the ALU to ALUOut
4. Add muxes used to write to register and M register
    - Implement the M register, but it will not be connected to memory yet. It will just be set to a value.
    - Make sure we can properly select the correct write address and write data for the reg file from the two corresponding muxes
5. Add Jump Control & PC Control
    - Implement PC register, and jControl mux
    - Test PC = PC+1 with adder, jcontrol = 0, and PCWrite
    - Test PC = PC + ZE(12bit imm) with adder, jcontrol = 1, and PCWrite
    - Test PC = PC[15:12]+ZE(12bit imm)[11:0], jcontrol = 2, and PCWrite
    - Test PC = ALUOut with jcontrol = 3, and PCWrite
    - Test PC = PC with no PCWrite signal (for jal instructions)
6. Add Memory Unit
    - Test the ability to load instructions based off the value in the PC register and load the result in the IR register (Instruction should also be stored in the M register)
    - Test the ability to load values from memory at location of ALUOut, and store the result in the MDC Register
    - This step adds an option to write to the register file with data from memory. We will test this case in this step as well.
7. Add Control Unit
    - Add the control unit to implemented datapath
    - Run through as many different instructions as it takes in order to make sure that every state in from the finitie state machine has been tested and works properly
    - Make any modifications to the control unit that is needed
8. Go over the final schematic. Make sure it is as efficient as possible and that all units work properly.

9. Add input and output integration with the FPGA
    - Test different inputs and make sure that the correct output is displayed on the FPGA

Integration Plan Diagram



Control Unit Finite State Machine


Reset
reset all control signals and non-addresable registers IRWrite
Reset = 0
Reset all write signals if(jumping in next cycle) IorM = 2 else IorM = 0 PCWrite = ~(jal || jump) jControl = 00
Reset all write signals IRWrite
M-Type
Awrite Bwrite ASel = 0 BSel = 0
AluCtrl=+ ALUWrite Asel = 1 Awrite if(jr) IorM = 2
JR
PCWrite JControl = 11
Load
IorM = 1 Mwrite
IorM = 0
DestData = 100 RegWrite destAddr=01
Store
MemWrite IorM = 0
R-Type
Awrite Asel = 1 Bwrite Bsel = 0
Arithmatic operator
AluCtrl=Func ALUwrite isZeroWrite
DestData = 000 RegWrite destAddr=IR[0]
Move
IR[0] = 1
RegWrite destAddr= 01 Destdata = 010
IR[0] = 0
RegWrite destAddr=00 Destdata = 011
Text
Jump Inst.
Jump
ItypeSel = 0 jControl = 10 PCWrite
Logical Inst.
ItypeSel = 0 Asel = 1 Awrite Bsel = 1 Bwrite
AluCtrl = based on Op ALUwrite isZeroWrite
destAddr = 01 RegWrite destData = 000
Arithmatic & jump0/1
ItypeSel = 1 Asel = 1 Awrite Bsel = 1 Bwrite jControl = 01 PCwrite = (jump0 && acc[15] = 0) || (jump1 && acc[15] = 1)
Arithmatic
Jump
No Jump
lui
destAddr = 01 DestData = 001 RegWrite
jal
destAddr = 10 destData = 101 RegWrite ItypeSel = 0 jControl = 10 PCWrite

Control Unit Testing

For testing the control units, we created a loop to iterate through each opcode. After printing out the waveform we verified that the correct control signals were triggering/not-triggering during the correct cycle for each opcode. Additional testing of each was done during the final stage of integration in Xilinx. We made sure that every state in the control unit worked as anticipated. Changes that were made are outlined in the Design Journal Document.

System Testing

System Testing Plan

We will begin testing our system by creating small fragments of code that test one type of instruction at a time. With this we will be able to verify that our entire datapath is working for each instruction. After testing the instructions we will move to a more complicated fragment of code, that could cause problems with timing, like jumps and memory. Once we have verified that the mini-program works on our processor correctly, we will test the euclidean formula program that we wrote at the beginning of the project for milestone 1 with a smaller value like 0xC. If we do not get the correct answer, we will use the waveforms to detect issues and resolve them accordingly. After we verify our system is working correctly with a small input value, we will test the desired value of 0x13B0 to ensure that we are working at full capacity.

Performance

- Total Number of Bytes Required:
  - Number of Bytes for Program: 128
  - Number of Bytes for Stack: 6
  - **Total for Program & Stack: 134**

- Total Number of Instructions executed when n = 0x13B0
  - **102111**

- Total Number of cycles required when n = 0x13B0
  - **408,458**

- Cycle time for design
  - **10.669ns**
  - **93.733 MHz**

- Total execution time when n = 0x13B0
  - $CPI = \frac{\#\,Cycles}{\#\,Inst} = \frac{405454}{102111} = 4.1$
  - $Exec.\,Time = CPI \times Cycle\ time \times \#\,instructions = 4.1 \times 10.669n \times 102111$

    = **4.358ms**

- Device utilization summary

  Selected Device : 3s500efg320-4

Number of Slices:	*475 out of*	*4656*	*10%*
Number of Slice Flip Flops:	*403 out of*	*9312*	*4%*
Number of 4 input LUTs:	*751 out of*	*9312*	*8%*
Number of IOs:	*35*		
Number of bonded IOBs:	*35 out of*	*232*	*15%*
Number of BRAMs:	*1 out of*	*20*	*5%*
Number of GCLKs:	*1 out of*	*24*	*4%*

Input/Output

We decided to put our processor on the Spartan-3e FPGA board for one of our extra features. We set it up so that the input can be changed by using the rotary button, and both the input and output values are displayed on the lcd. The button directly below the rotary button is used as the reset button for the processor. The user must first move the rotary button until the desired input is displayed on the LCD and then hit the reset button in order to run the new input through the processor and have the correct corresponding output displayed on the LCD.

Design Process Journal

Milestone 1: Due October 3, 2017

Week Assignments:
We decided to all work collectively on this milestone in order to efficiently develop a design that we all liked and agreed upon. We met Sunday October 1st from 4:30 PM to 7:30 PM and Monday October 2nd from 5:30 PM to 8:30 PM

We decided to design a modified accumulator processor. This design would allow us to minimize the information needed for each instruction which maximizes the number of bits available for immediates in I-type instructions. Our instructions have an op-code and some have a function code. The opcode is the main label for each instruction and the function code primarily labels arithmetic operations. Because we only use additional registers beyond the accumulator in R-type instructions, we have an excess of bits available to address these registers beyond what we may need. In R-type instructions we also have the least significant bit set to be a flag to determine whether to store the result in the accumulator or the other register used. We also created an M-type instruction type used exclusively for memory instructions. This functions similarly to I-types except it only has an 8-bit immediate used to offset an address stored in the available register.

Our design has 16 dedicated registers with some of our choices resembling MIPS. Similar MIPS we have a zero, stack-pointer, return-address, (2) argument, (1) return, (2) saves, (3) temporaries, (2) in, and (2) out registers. The 16th register in our design is called $iszero. It constantly updates the register to whether the last ALU operation ended with zero as the result. This is used for branch control as we can use this register to determine if two values are equal. Primarily branches are controlled by checking the most significant bit of acc to determine if &acc is less than zero. We chose to control branches this way to be able to simplify the hardware implementation and addressing as this allows us to use the full 12-bits as an immediate to extend our branch range.

Milestone 2: Due October 10, 2017

- We met on Thursday Oct. 5 for 45 minutes to distribute the initial tasking for the weekend before our final meeting as a group on Sunday. Assignments are shown below.

Week Assignments:
Russell Johnson: Continue work on Assembler, update machine code in Design Doc, move RTL
Griffin Steffy: I-type RTL
Charley Beeman: M-Type RTL, Reformat Design Doc
Stella Park: R-Type RTL (not move)

- We also met on Sunday Oct. 8 for 3 hours as a group to combine and simplify the RTL, generate a list of component specifications, test our RTL, and edit this journal.

Based on feedback from Dr. Taylor, we made a handful of minor modifications to our design document. In the instructions section, we specified what $acc, $reg, and imm meant as well as whether or not the imm was sign extended or zero extended. We decided to get rid of the jacc instruction and add a new instruction called jr to the M-types because it gives the user the ability to jump to values in other registers and acc is now addressable. Making acc addressable also simplifies the RTL. In the registers section, we got rid of the a and v registers and used those spots for additional temporary registers in order to make the programming simpler. Also, we got rid of the second $in and $out registers so that we only have one of each that we have to use. In the addressing modes section, we specified that the immediate for Register+Offset addressing is sign extended. Also, we stated which instructions use Pseudo-Direct addressing. Reorganized the entire document by adding a hyperlinked table of contents and using tables for the Euclidean assembly and machine code section as well as the machine code fragments section.

For our RTL we decided to use a multi-cycle style of processor because of the speed benefits it provides over single cycle, and the hardware simplification that pipeline doesn't have. Our goal was to simplify the hardware as much as possible. After we created our multi-cycle RTL we were able to start a list of components for our data path and began thinking about what control signals will be needed for our hardware implementation.

Milestone 3: Due October 18, 2017

Week Assignments:

- We met on tuesday Oct. 17 in class time for 30 minutes to discuss about meeting time and got started on the milestone 3.
- We also met on wednesday Oct. 18 from 9:00am-2:00pm to design a datapath, list control signals, modify and specify component lists, and test the datapath design.
    - This milestone was all don collectively as a group

Looking at the RTL description from Milestone 2, we started by drawing block diagram of datapath. The datapath design was modified while we were going through the testing. For the testing of the RTL, load, jr, stor, jump, ori, addi, jump0, jump1, and lui were chosen. Through the testing of each instruction, we observed that RTL was working as expected.

Furthermore, we changed the format of the general components specification from chart to listing. This reformating allowed us to add on more information about building and testing each components and clarification of each components. We also added specifications for building and testing each component. Additionally, we made a detailed plan for implementing our datapath.

We ended up slightly modifying the RTL after building the datapath. We changed the relative-to-pc addressing mode so that it is PC+1 and there is no left shift because this simplifies the memory design so that everything is in blocks of 16 bits. We also modified the pseudo-direct addressing mode so that it takes the upper 4 bits of PC and 12 bits of the immediate for simplification. So, the RTL for jump0, jump1, jump, and jal all changed in order to incorporate these two modified addressing modes. Lastly, we modified the store RTL by loading acc into A at the end of cycle 3 in order to make the store faster in cycle 4.

Our accumulator-based architecture influenced our design of our datapath by simplifying writing to memory since the address always come from the ALU and the data always comes from acc. Also, everything is centered around a register file which we are familiar with and is easier to use. Based on the way that our RTL was laid out, multicycle was the most logical fit.

Our process for testing was to test the basic functionality of the component with multiple cases as well as making sure that edge cases were tested if needed. As described previously, we did end up modifying our RTL and addressing modes slightly based on errors discovered while making the datapath.

Milestone 4: Due October 25, 2017

Week Assignments:

- Griffin: Xilinx implementation and testing of Muxes, Extenders, and shifter, Reg File
- Russell & Griffin: Xilinx implementation and testing of ALU and Register File, revamp integration plan, and start integration
- Charlie: Labs and FSM
- Stella: Updating Design Document and Control Unit Design

For this milestone we began by fixing the RTL for the move instruction. We recognized that at this point we can't read and write from the register file properly simultaneously, so we split the operation into two steps. Second we saw that some of the wires and muxes were unneeded/mistakenly pointing to the wrong location in the datapath so we fixed those issues as well.

We started by implementing the easy components like Muxes, Extenders, and shifters. We verified that those components were working to our standards through our test benches. We also implemented our ALU and tested to make sure that all operations worked correctly as well as the "iszero" functionality, however, we do not have overflow implemented at this time. Next we began implementing the Register File and created a test for that as well. We tested the reading and writing functionality to make sure that is was working properly as well. After the Reg File we started Implementing other components except the memory component.

Before we began integrating our design we first redesigned our plan to be more simple and easier to follow than originally. After implementing our components we knew which components would be the easiest start with so we decided to start with the register File as well as the A, B, and IR registers. This was the first step we implemented and tested by making sure that you the register file would read and write properly. We then implemented the second step which involved adding the mux for controlling the input to A, the mux for controlling the input to B, and the mux controlling whether you SE or ZE the 12 bit immediate going into B. We were able to get a successful test bench from that step as well.

The last thing we did for this operation was design the Control Unit for our datapath. We decided to use a Finite State Machine because it complied with multi-cycle the best. We also began thinking about how we were going to test our control unit. We decided that we would test it similar to our components, by looping through the possible situations and ensuring that the correct control signals were being triggered properly. We will do this testing in steps for each type of instruction. So the single test is not an information overload.

Milestone 5: Due November 1, 2017

Week Assignments:

- We met on Sunday Oct. 29 from 4:00 pm to 7:30 pm to modify RTL, datapath, and control unit and continue on implementing parts, unit tests, and processor.
- We met again on Oct. 30 in class time and from 5:00 pm to 7:00 pm to additionally work on processor and integration..
- We met once again on Oct. 31 from 10:00 am to 11:30 am to fix some previous RTL and integration steps, as well as continuing to work on the control unit. We also met from 5:15 pm to 10:00 pm to finish up the control unit and integrate the last step of integration.
- We also met on Nov. 1 from 10:00am to 11:30am as well as 1:00pm to 2:30pm in order to go over the control unit and start implementing tests for it
- Charley: Control Unit design and testing
- Griffen: help Russell with xilinx testing, Visio datapath, test Control Unit
- Russell :finish implementing and testing all parts, integrate xilinx, test Control Unit
- Stella: update design doc, assist with testing

Our first task of the week was adding a instruction that we forgot originally, and that instruction was jal. Since this is instruction is very similar to jal, we had the workings of it figured out already, it was just missing from the RTL. Second we optimized a few cycles on our original RTL, by combining cycle two of the I-Type jump0, jump1, and arithmetic instructions. We were able to combine them into a single cycle instead of splitting the jumps into two seperate cycles.

By changing our RTL and adding in an instruction we had to make changes in our datapath and FSM control unit design. We changed the toAcc control to destAddr and it became a 2 bit signal, Also we changed the 5 bit input mux for destData to become a 6 input mux. We added some branches on the FSM to accommodate for the jal instruction, as well as match our optimized RTL. The next step we took was when we were redoing our datapath we recreated it in the microsoft program visio, to make the final design much cleaner. We added the new files to the repository.

Next we finished up integration by integrating the final steps of our design. We developed a graphical representation of our steps in the plan to aid us when doing the integration. We created exhaustive tests for each integration step, which also verified the step before it was workings as well. The last stage we implemented was memory because we were waiting to see if our Lab results were optimal. We are starting with block memory because it is the most simple type, but later we are leaving the option open to use distributed memory to increase the speed and performance of our processor.

Next based off of our FSM we began implementing our control unit into xilinx. We used a case switch architecture to switch the unit between states based off of the previous state. We will test this unit by inputting different opcodes and ensuring that it follows the correct state path. We will begin by testing one instruction at a time to verify visually with the waveform, but once we begin adding more and more instructions to test we will begin using pass fail statements to verify our results, after we understand how to test the control unit properly.

We modified our control unit testing in the design doc to be based off of our FSM in order to make testing more practical. We also wrote a general system test plan in the design doc. Lastly, we touched up our FSM and Datapath to make sure we had consistent naming structure for all the control signals.

Milestone 6: Due November 8, 2017

Week Assignments:

- We met as a group on Sunday, Nov. 5 from 2:00pm to 11:00pm in order to work on integration of the control unit into our complete processor schematic. We also made modifications to the datapath and control unit as specified below.
- We met on Monday, Nov. 6 from 5:30pm to 7:30pm in order to look at our potions for using different types of memory: single port block, dual port block, and distributed.
- Also, we met from Tuesday, Nov. 7 at 7:30pm to Wednesday, Nov. 8 at 3:30am in order to finalize the full implementation of our processor, maximize speed and efficiency, and calculate performance of our processor
- **Griffin**: Integration testing of overall processor as well as the control unit, update datapath & FSM, update design doc, assist Russell with using LCD on FPGA
- **Charlie** :Integration testing of overall processor as well as the control unit, work on setting up input for processor on FPGA
- **Russell**: Integration testing of overall processor as well as the control unit, finalize the processor schematic and testing in Xilinx, finalize the assembler, update design doc, set up output to LCD on FPGA
- **Stella**: Integration testing of overall processor as well as the control unit, update design doc

We integrated the control unit into the final processor schematic in Xilinx. We ran a rigorous test procedure including the testing of a multitude of instructions that tested every possible state in the control unit and made sure that the correct outputs were achieved. While testing the addition of the control unit, we ended up having to make a handful of hardware modifications to our processor. We found out that we needed to use a buffer in order to append the upper 4 bits of PC to the 12 bit immediate for jumps. Xilinx had a 1 bit buffer symbol, so we made our own append symbol that took in the two signals, used 16 of the 1 bit buffers, and output the appended 16 bit value. We also decided to switch from Single Port Block Memory to Simple Dual Port Block Memory so that we could have a separate read and write address which saved us having to add in additional delays and simplified our design. We also needed to have our instructions available a state earlier so we added a mux that switches between the output from the IR register and the output from the memory unit; so we can pass the instruction to the desired places in the datapath a full cycle earlier whenever IRwrite is enabled (Bypass the waiting for the instruction to be written to the IR register and then read), once IRwrite is set back to 0, then the output from IR is passed through. This solution solved many of our initial issues with programs not running properly. We decided to make the processor faster by eliminating the double clock for the memory unit and instead put in a couple delay cycles into our control unit: 1 that all the jumps go to in order to give enough time for the PC to be updated before reading the next instruction and 1 that all instructions go to before returning to the first state in order to give the memory unit time to read. This increased the processor from around 50MHz to around 90MHz.

After making the changes to our design, we ran through the control unit to finalize any changes made, and note them. After that we updated the FSM to then show the changes we made. This included adding write signals, adding in wait states, as well as reset states.

Once we finalized our changes in our design, we update our datapath as well. This included adding muxes, changing a few connections. We updated our design using visio.

We then ran the relPrime program with an initial “n” value of 0x13B0 that is placed in the $in register, and we got the correct answer of 0x000B in the $out register.

After our processor was complete, we began doing performance analytics on our design. We started of by finding the number of bytes used by our program. Using the memory file after running a simulation we were able to use that to find the total number of bytes between program and data memory to be 134 bytes. We then found the number of instructions and cycles when running our relPrime program with the input 5,040. We used our control unit to do this by displaying a counter in two different locations. After we obtained those values, we used the Synthesis report to find our cycle time of 10.669ns (93.733 MHz). We used these two sets of numbers to calculate our programs execution time. We calculated our execution time to be 4.358 ms. After getting the execution time, we found the Utilization summary from the synthesis report.

We then started to work on putting our processor on an FPGA board as well as setting up inputs and outputs. Eventually we ended up setting it up so that the user must first move the rotary button until the desired input is displayed on the LCD and then hit the reset button in order to run the new input through the processor and have the correct corresponding output displayed on the LCD.