RHIPS- Extended™ Final Report

Team 1F

Computer Architecture

Zackery Painter

2/20/2021

Contents

Brief Introduction

RHIPS-Extended™ is an extension of the RHIPS™ processor created by John Neill, Zack Painter, Anthony Sparks, and Jack Thorp for Computer Architecture in the Fall of 2020 at Rose-Hulman Institute of Technology. The extended version is a 16 bit multicycle processor. The processor has 16 bit data addresses and 4-but register address bus. All data buses are 16 bit.

In-depth Explanation

Testing

THE PROCESSOR WAS IMPLEMENTED BY TESTING COMPONENTS INDIVIDUALLY, TESTING SUB-SECTIONS, AND TESTING THE FINAL DATAPATH.

- Component testing
  - Each component was tested according to the design document.
  - After testing, multiple components were put together to make a full subsection.
- Sub-section testing
  - After components were tested, the components were put together and tested in subsections defined in the design document
  - After the subsections were tested, a symbol was created from the subsection and added to the final processor.
- Final processor
  - The final subsection (section F) was the final processor.
  - The final processor was comprised of 3 subsections and connected on the high-level schematic diagram.

Instruction Set Design

The instruction set was originally designed with 4 bit opcodes to get a total of 16 instructions. However, after running out of instructions, it was decided to take 2 of the opcodes and make a 3-bit ext code to make a total of 16 additional instructions. This gave a total of 32 instructions possible. To accommodate the new instruction types, it was decided to use the branch instruction opcodes because they are pseudo-instructions. The new type is below.

Extension (Ext-Type)

15 – 12	11	11 – 10	8 – 7	4 – 3	0
Opcode	KD	Ext. Code	General Purpose 1	General Purpose 2	
0001 or 0010	1 bit	3 bits	4 bits	4 bits	

Description of Type: Extension types are used as an extension of the original instruction set. Ext-Types are used for various types but are mainly designed for kernel instructions. For example, the KD section is used to choose between a kernel destination register.

Implementation decisions

The implementation plan was to split the data path into 5 sections.

A. Section A was the Instruction Memory and the Data Memory, as well as the memory management and switching logic.
B. Section B was divided into Section B1 and B2
    a. B1 was the main register file and input muxes
    b. B2 was the kernel register file and the input muxes
C. Section C was the combination of B1 and B2, along with switching logic on the output of the output of both register files.
D. Section D was the input registers to the ALU, the ALU and the result registers.
E. Section E was the combination of section D and C
F. Section F was the entire processor, including PC logic and Sections A-E

The final processor was comprised of subsections A, BC and D. This made it easier to test and connect together. It also made it cleaner in the upper-level data path.

Final Comments on Design Choices

Overall, the design is good and was simple to implement. The current design allows for the implementation and expansion of the kernel and a future operating system. User modes and Kernel modes are supported through switching physically between 2 separate register files. Additionally, the future ability to read instructions directly out of the data memory can expand the number of programs that can run on this processor. However, this leads to some security issues that would need to be addressed through hardware changes. This leads to additional changes that should be made, including making branch more efficient, as at the moment it takes 3 instructions to branch.

Extras

Assembler

The assembler takes an assembly program, strips all comments and most spaces, and turns the instruction into machine code. It also has a configuration file defining types, instructions, and registers. This file gets loaded into a lookup table and directly assembled into the complete instruction.

The assembler has two major features. First, it gives a side-by-side view of the instruction and machine code. This was important because it made it easier to debug any errors in the waveform. Second, it gave the option to splice a pre-assembled kernel into a final memory file to be loaded into Xilinx.

Below is an example of both

```
0: slt $0, $0, $0 : 0000000000000000
1: slt $0, $0, $0 : 0000000000000000
2: slt $0, $0, $0 : 0000000000000000
3: slt $0, $0, $0 : 0000000000000000
4: slt $0, $0, $0 : 0000000000000000
5: slt $0, $0, $0 : 0000000000000000
6: slt $0, $0, $0 : 0000000000000000
7: slt $0, $0, $0 : 0000000000000000
8: slt $0, $0, $0 : 0000000000000000
9: slt $0, $0, $0 : 0000000000000000
10: slt $0, $0, $0 : 0000000000000000
11: slt $0, $0, $0 : 0000000000000000
12: slt $0, $0, $0 : 0000000000000000
13: slt $0, $0, $0 : 0000000000000000
14: slt $0, $0, $0 : 0000000000000000
15: slt $0, $0, $0 : 0000000000000000
16: slt $0, $0, $0 : 0000000000000000
17: slt $0, $0, $0 : 0000000000000000
18: slt $0, $0, $0 : 0000000000000000
19: slt $0, $0, $0 : 0000000000000000
20: slt $0, $0, $0 : 0000000000000000
21: slt $0, $0, $0 : 0000000000000000
22: slt $0, $0, $0 : 0000000000000000
23: slt $0, $0, $0 : 0000000000000000
24: slt $0, $0, $0 : 0000000000000000
25: slt $0, $0, $0 : 0000000000000000
26: slt $0, $0, $0 : 0000000000000000
27 : and $t0, $t0, $0 : 0100011101110000 : relPrime
28: ori $t0, 2 : 1000011100000010
29: or $t1, $0, $in : 0101100000001010
30: l2m $ra, 0 : 1011001000000000
31 : or $a0, $0, $t1 : 0101001100001000 : relLoop
32: or $a1, $0, $t0 : 0101010000000111
33: l2m $t0, 1 : 1011011100000001
34: l2m $t1, 2 : 1011100000000010
35: jal gcd : 1100000000110001
36: and $t0, $0, $0 : 0100011100000000
37: ori $t0, 1 : 1000011100000001
38: beq finish, $v0, $t0 : 0100111000000000,
1000111000101101,
0001000001010111
41: l2r $t0, 1 : 1001011100000001
```

Figure 1- Side-By-Side Non-spliced – Replrime

```
MEMORY_INITIALIZATION_RADIX=2;
MEMORY_INITIALIZATION_VECTOR=
0001011000000000,
0001011000000000,
0001011000000000,
0000000000000000,
0100000000110100,
0100000110011001,
0100111000000000,
1000111000001010,
0001000100000001,
0010000000000000,
0100000110011001,
1000000100000010,
0100111010010000,
1000111000011000,
0001000000000001,
0100000110011001,
1000000100000001,
0100111010010000,
1000111000010101,
0001000000000001,
0001011000000000,
0100001010011001,
1000001000011011,
0010000000000000,
0100100010011001,
1000100000000001,
0001011000000000,
0100011101110000,
1000011100000010,
0101100000001010,
1011001000000000,
0101001100001000,
0101010000000111,
1011011100000001,
1011100000000010,
1100000000110001,
0100011100000000,
1000011100000001,
0100111000000000,
1000111000101101,
```

Figure 2- Final file - With kernel spliced in – Relprime

Kernel

The kernel is very basic, but can be improved in the future. The kernel handles simple exceptions and basic software and hardware interrupts. The kernel has an “idle loop” at address zero. At this loop, the processor waits for an interrupt to start the program at the end of the kernel. The assembler automatically decides where the main loop starts based on how big the kernel is. The interrupt handler is always at address 0x4. The design document provides details of kernel specific registers. To return from the kernel, it can use the instruction retkern, but this is not included in the main RTL.

```
0 : term : 0001011000000000 : DeadLoop
1: term : 0001011000000000
2: term : 0001011000000000
3: slt $0, $0, $0 : 0000000000000000
4 : and $k0, $errmask, $errreg : 0100000000110100 : interruptHandler
5: and $k1, $00, $00 : 0100000110011001
6: bne valid, $k0, $k1 : 0100111000000000,
1000111000001010,
0001000100000001
9: retkern : 0010000000000000
10 : and $k1, $00, $00 : 0100000110011001 : valid
11: ori $k1, 2 : 1000000100000010
12: beq InvalidAddr, $k0, $k1 : 0100111010010000,
1000111000011000,
0001000000000001
15: and $k1, $00, $00 : 0100000110011001
16: ori $k1, 1 : 1000000100000001
17: beq StartExec, $k0, $k1 : 0100111010010000,
1000111000010101,
0001000000000001
20: term : 0001011000000000
21 : and $pc_temp, $00, $00 : 0100001010011001 : StartExec
22: ori $pc_temp, 27 : 1000001000011011
23: retkern  : 0010000000000000
24 : and $returncode, $00, $00 : 0100100010011001 : InvalidAddr
25: ori $returncode, 1 : 1000100000000001
26: term : 0001011000000000
```

Figure 3- Kernel side-by-side

Conclusion

RHIPS-Extended™ is a Load Store processor with the capabilities to have a full kernel and user mode. Additionally, it is flexible enough that changes can be made easily without breaking the underlying instruction set. In the future, various timing improvements could be made to increase performance and efficiency. For example, doing multiple setup steps at once before writing into the register files. Overall, this is a very versatile processor with many capabilities.

APPENDIX – DESIGN JOURNAL

RHIPS- Extended™ Processor Design

By: Zackery Painter

1/11/2021

Special Notes

RHIPS-Extended™ is an extension of the RHIPS™ processor created by John Neill, Zack Painter, Anthony Sparks, and Jack Thorp for Computer Architecture in the Fall of 2020 at Rose-Hulman Institute of Technology. The original documentation for RHIPS can be found online at http://zacksportfolio.ddns.net/wp-content/uploads/Final_Report_2Z.pdf. The purpose of this extension is to improve and fix many design errors in the original RHIPS design as well as continue exploring how processors function through more advanced features and improvements.

Executive Summary

RHIPS-Extended™ processor is a hybrid design, employing aspects of both assembler and memory-to-memory instructions. Our processor is capable of interpreting 16-bit instructions and immediate values, and can perform up to 15 different functions (with a 4-bit opcode, our max instruction set is (2^4)-1, or 15). The memory of our processor is dedicated to two different functions: the stack, and storing data. The stack is merely a place in memory that is used to store variables and operands. Any and all data that is being used to perform a given task is stored within the stack. The memory also stores return values, which are any values that the processor needs to 'pass on' to subsequent processes later on in a function. Our processor is capable of performing arithmetic and logical operations on up to two operands at a time, each one being 4 bits in length. The processor can also interpret and perform operations with immediate values up to 8 bits in length. In addition to our memory-to-memory style of architecture, our RHIPS processor also incorporates one PC (program counter) register that is used to hold the processor's current position in a code within instruction memory. To perform its necessary functions, our processor utilizes elements such as a sign extender, an ALU (Arithmetic Logic Unit), multiplexers (MUX's), a left shifter, and basic logic gates such as AND and OR. RHIPS-Extended™ also includes an extension in the form of a kernel and additional instructions.

Basic Overview

Instruction Types:

Arithmetic (A-Type)

15	12 | 11	8 | 7	4 | 3	0
Opcode	Destination	Operand1	Operand2	
4 bits	4 bits	4 bits	4 bits	

Description of Type: Arithmetic types are types that use the ALU the most, A-Types work with two registers.

Jump (J-Type)

15	12 | 11	0
Opcode	Operand1 / Destination	
4 bits	12 bits	

Description of Type: Jump types are used for jumping from one destination to another

Immediate (I-Type)

15	12 | 11	8 | 7	0
Opcode	Op1/ Destination	Immediate	
4 bits	4 bits	8 bits	

Description of Type: Immediate types are used for instructions that need to take a direct (signed) number as part of the instruction, for example adding an immediate to a register.

Extension (Ext-Type)

15	12 | 11	11 | 10	8 | 7	4 | 3	0
Opcode	KD	Ext. Code	General Purpose 1	General Purpose 2	
0001 or 0010	1 bit	3 bits	4 bits	4 bits	

Description of Type: Extension types are used as an extension of the original instruction set. Ext-Types are used for various types but are mainly designed for kernel instructions. For example, the KD section is used to choose between a kernel destination register.

* KD means Kernel Destination, this can also be used for general purpose.
** General Purpose 1 and 2 can be used as operands and destinations depending on the instruction, see instruction RTL for instruction-specific usage.
*** Ext. Code means Extension code

Full Instruction Set List:

Instruction	OP Code / Ext. Code	Type
Slt	0000	Arithmetic
Beq	0001 / 000	Extension
Bne	0001 / 001	Extension
Add	0011	Arithmetic
And	0100	Arithmetic
Or	0101	Arithmetic
Sub	0110	Arithmetic
Ls	0111	Immediate
Ori	1000	Immediate
L2r	1001	Immediate
Addi	1010	Immediate
L2m	1011	Immediate
Jal	110	Jump
J	1101	Jump
Jr	1110	Jump
Ccp	0001 / 010	Extension
Cmp	0001 / 011	Extension
Km2mm	0001 / 100	Extension
Mm2km	0001 / 101	Extension
Term	0001 / 110	Extension
Syscall	0001 / 111	Extension

Syntax and Semantics – Real Instructions:

ADD

```
add dest, op1, op2
```

15 – 12	11 – 8	7 – 4	3 – 0
Opcode	Destination	Operand1	Operand2
4 bits	4 bits	4 bits	4 bits

Put the sum of op1 and op2 into dest.

Step number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = A + B	Do the addition and place the result in the Result register
[4]	Reg[IR[11-8]] = Result	Place the value from the Result register into the register in the main register
[5]	PC = PC + 1	Place newPC into PC by setting PCWrite to 1 and taking

ADDI

```
addi dest/op1, imm
```

15	12 | 11	8 | 7	0
Opcode	Op1/ Destination	Immediate	
4 bits	4 bits	8 bits	

Put the sum of op1 and an 8 bit sign-extended immediate into op1.

Step Number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = C + IR[7-0]	Select IR[7-0] and C do the add between the 2 and place the result in the Result register
[4]	Reg[IR[11-8]] = Result	Place the result from the result register into the address in Reg at the address specified in IR[11-8]
[5]	PC= PC + 1	Place newPC into PC by setting PCWrite to 1 and taking

AND

```
and dest, op1, op2
```

15 – 12	11 – 8	7 – 4	3 – 0
Opcode	Destination	Operand1	Operand2
4 bits	4 bits	4 bits	4 bits

Step number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = A & B	Do the AND and place the result in the Result register
[4]	Reg[IR[11-8]] = Result	Place the value from the Result register into the register in the main register
[5]	PC = PC + 1	Place newPC into PC by setting PCWrite to 1 and taking

BNE

```
bne op1, op2
```

15 – 12	11	10 – 8	7 – 4	3 – 0
Opcode	KD	Ext. Code	General Purpose 1	General Purpose 2
0001 or 0010	1 bit	3 bits	4 bits	4 bits

If GP1 does not equal GP2, branch to the location of destination in memory.
**** The Assembler must load the address into the Branch Register**

Step Number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = B-A	Subtract A from B ALU sets Zero Flag if B-A !=0 (See notes below)
[4]	If (ZERO && BRANCH) PC = Reg[Branch]	This all happens outside of the ALU. The control sets the Branch control bit and if the ALU sees that B-A != 0, it sets the Zero flag. If so, the PC is set to the value in the branch register *The value in the branch register is always directly exposed out of the register file. ** We discard newPC *** The ALUOP sets the Zero flag to an inverted state 0 -> 1 1 -> 0
[5]	PC= PC + 1	If the branch is not taken, we'll just continue and increment PC

BEQ

```
bne op1, op2
```

15 – 12	11 – 11	10 – 8	7 – 4	3 – 0
Opcode	KD	Ext. Code	General Purpose 1	General Purpose 2
0001 or 0010	1 bit	3 bits	4 bits	4 bits

If GP1 equals GP2, branch to the location of destination in memory.
**** The Assembler must load the address into the Branch Register**

Step Number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = B-A	Subtract A from B ALU sets Zero Flag if B-A =0
[4]	If (ZERO && BRANCH) PC = Reg[Branch]	This all happens outside of the ALU. The control sets the Branch control bit and if the ALU sees that B-A=0, it sets the Zero flag. If so, the PC is set to the value in the branch register *The value in the branch register is always directly exposed out of the register file. ** We discard newPC
[5]	PC= PC + 1	If the branch is not taken, we'll just continue and increment PC

J

```
J destination
```

15	12 | 11	0
Opcode	Operand1 / Destination	
4 bits	12 bits	

Jump to the location of the destination in memory.

Step Number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = PC[15-11] | ZE(IR[11-0])	Or together the top 4 bits of PC and a zero extended version of IR[11-0]
[4]	PC = Result	Place the Result into PC

JAL

```
jal destination
```

15	12 | 11	0
Opcode	Operand1 / Destination	
4 bits	12 bits	

Step Number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = PC[15-11] | ZE(IR[11-0])	Or together the top 4 bits of PC and a zero extended version of IR[11-0]
[4]	Reg[IR[11-8] = PC	Place the PC in the register specified in IR[11-8]
[5]	PC = Result	Place the Result into PC

JR

```
jr register
```

15	12 | 11	0
Opcode	Operand1 / Destination	
4 bits	12 bits	

Jump to a specific register.

Step Number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = C	Pass through the Value from C into the result register
[4]	PC = Result	Place the Result into PC

OR

```
or dest, op1, op2
```

15 – 12	11 – 8	7 – 4	3 – 0
Opcode	Destination	Operand1	Operand2
4 bits	4 bits	4 bits	4 bits

Put the logical OR of op1 and op2 into dest.

Step number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = A or B	Do the or and place the result in the Result register
[4]	Reg[IR[11-8]] = Result	Place the value from the Result register into the register in the main register
[5]	PC = PC + 1	Place newPC into PC by setting PCWrite to 1 and taking

ORI

```
ori op1, imm
```

15	12 | 11	8 | 7	0
Opcode	Op1/ Destination	Immediate	
4 bits	4 bits	8 bits	

Put the logical OR of op1 and a zero-extended immediate into op1.

Step Number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = C | IR[7-0]	Select IR[7-0] and C do the or between the 2 and place the result in the Result register
[4]	Reg[IR[11-8]] = Result	Place the result from the result register into the address in Reg at the address specified in IR[11-8]
[5]	PC= PC + 1	Place newPC into PC by setting PCWrite to 1 and taking

L2R

```
L2r op1, imm
```

15 — 12	11 — 8	7 — 0
Opcode	Op1/ Destination	Immediate
4 bits	4 bits	8 bits

Load the value from the address specified in the immediate from memory into the register (op1)

Step number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = DataMem[IR[7-0]]	Place the value stored in Data Memory at the address IR[7-0] into Result
[4]	Reg[IR[11-8]]=Result	Place the value from the Result register into the Register in the main register
[5]	PC= PC + 1	Place newPC into PC by setting PCWrite to 1 and taking

SLT

```
slt dest, op1, op2
```

15 — 12	11 — 8	7 — 4	3 — 0
Opcode	Destination	Operand1	Operand2
4 bits	4 bits	4 bits	4 bits

If op1 is less than op2, a 1 is stored in dest. If op1 is not less than op2, a 0 is stored in dest.

Step number	**RTL**	**Description**
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = 1 if A<B	Do the slt place the result in the Result register
[4]	Reg[IR[11-8]] = Result	Place the value from the Result register into the register in the main register
[5]	PC = PC + 1	Place newPC into PC by setting PCWrite to 1 and taking

SUB

```
sub dest, op1, op2
```

15	12 | 11	8 | 7	4 | 3	0
Opcode	Destination	Operand1	Operand2
4 bits	4 bits	4 bits	4 bits

Subtract op2 from op1, and store the difference into dest.

Step number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = A op B	Do the specified Operation and place the result in the Result register
[4]	Reg[IR[11-8]] = Result	Place the value from the Result register into the register in the main register
[5]	PC = PC + 1	Place newPC into PC by setting PCWrite to 1 and taking

LS

```
ls op1, imm
```

15 – 12	11 – 8	7 – 0
Opcode	Op1/ Destination	Immediate
4 bits	4 bits	8 bits

Left shift an immediate by 8.

Step Number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = C op IR[7-0]	Select IR[7-0] and C do the operation between the 2 and place the result in the Result register
[4]	Reg[IR[11-8]] = Result	Place the result from the result register into the address in Reg at the address specified in IR[11-8]
[5]	PC= PC + 1	Place newPC into PC by setting PCWrite to 1 and taking

L2M

```
L2m op1, imm
```

15	12 | 11	8 | 7	0
Opcode	Op1/ Destination	Immediate	
4 bits	4 bits	8 bits	

Loads the value from the register into the memory at the address in the immediate

Step Number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = C	Pass the value, C through the ALU into the result register
[4]	DataMem[IR7-0]] = Result	Place the result register into the address IR[11-8] into Data Memory
[5]	PC= PC + 1	Place newPC into PC by setting PCWrite to 1 and taking

CCP

```
ccp dest, op1
```

15 — 12	11 — 11	10 — 8	7 — 4	3 — 0
Opcode	KD	Ext. Code	General Purpose 1	General Purpose 2
0001	1 bit	3 bits	4 bits	4 bits

Copy from a register to a kernel register. Destination can be k0 or k1.

Step Number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = B	Pass through B
[4]	Reg[0xD]=1	Switch to Kernel Mode
[5]	Kreg[IR[11]]=Result	Copy the result to the kernel register
[6]	PC= PC + 1	Increment PC

CMP

```
cmp dest, op1
```

15 – 12	11 – 11	10 – 8	7 – 4	3 – 0
Opcode	KD	Ext. Code	General Purpose 1	General Purpose 2
0001	1 bit	3 bits	4 bits	4 bits

Copy from a kernel register to a register in the main processor. Op1 can be k0 or k1.

Step Number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Reg[0xD]=1	Switch to Kernel Mode
[4]	B = Kreg[IR[11]]	Load the kernel register into B
[5]	Result = B	Pass through B
[6]	Reg[0xD]=0	Switch to User Mode
[7]	Reg[IR[7-4]] = Result	Set the specified register to the result
[8]	PC= PC + 1	Increment PC

TERM

Term

15 – 12	11	11 – 8	7 – 4	3 – 0
Opcode	KD	Ext. Code	General Purpose 1	General Purpose 2
0001	1 bit	3 bits	4 bits	4 bits

Terminate and force the processor to end the current program and go to a locked state by forcing the clock to 0. (Set the HOLD register to 0)

Step Number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Reg[0xD] = 1	Go into Kernel Mode
[4]	PC = 0x0	Set PC to 0x0
[5]	Kreg[0x7] = 1	Hold the processor in a locked state

SYSCALL

```
syscall code
```

15 — 12	11	11 — 8	7 — 4	3 — 0
Opcode	KD	Ext. Code	General Purpose 1	General Purpose 2
0001	1 bit	3 bits	4 bits	4 bits

I'm not completely sure how I want this to work yet. Essentially, set the processor into a different state or do things such as change between kernel operations and a program in user space.

Step Number	RTL	Description
[1]	a.) instMem[PC] b.) IR = instMem[PC]	Take instruction out of Instruction Memory and place it into the IR register. newPC is a wire. It will stay on the wire (Pre-calculated by the PC adder).
[2]	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]	Load 3 common values into A,B, and C registers simultaneously.
[3]	Result = ZE[A]	Zero Extend A
[4]	Reg[0xD] = 1	Switch to Kernel Mode
[5]	KReg[0x4] = Result	Record the Type of Interrupt
[6]	KReg[0x6] = IR	Record the Instruction
[7]	KReg[0x2] = PC	Record the location of interrupt
[8]	PC = 0x4	Go to Interrupt Handler

Syntax and Semantics – Pseudo-Instructions:

Pseudo-Instructions are instructions that are to be handled by the assembler, the number of instructions in the program may increase above what is written by the programmer.

BEQ – Pseudo-Instruction syntax

```
Beq destination, op1, op2
```

Beq is assembled into three instructions, and, ori, and real beq. and to clear the br register, ori to load the immediate beq to do the real branch

BNE – Pseudo-Instruction syntax

```
bne destination, op1, op2
```

Bne is assembled into three instructions, and, ori, and real bne. and to clear the br register, ori to load the immediate bnr to do the real branch

Register and Memory Layout:

Memory is separated into separate memory blocks. The sections are: main register file, kernel memory, kernel registers, main memory / stack, and instruction memory.

Main Registers:

This is a memory to memory processor, so the register file has been reserved as a portion of the top 15 memory addresses.

Res. Mem.	Address	Description
$0	0x0	“Zero Register” This register is always zero
$pc	0x1	Program Counter
$ra	0x2	Return address, holds the to return to when jumping
$a0	0x3	Argument Register, holds the first argument for functions
$a1	0x4	Argument Register, holds the second argument for function calls
$v0	0x5	Result register 1, holds the resulting value from a given operation
$v1	0x6	Result register 2, holds the resulting value from a given operation if the first is in use
$t0-$t2	0x7 - 0x9	Temporary memory space for assembler or programmer, not for long-term data storage

$IN	0xA	A simple external input register
$OUT	0xB	A simple external output register
$sp	0xC	Stack Pointer
$memPage	0xD	Switch Between Using Kernel Memory and Main Memory.
$br	0xE	Branch Temporary
$mp	0xF	Memory Pointer

Kernel Registers:

Located inside the kernel register file.

Register Name	Address	**Description**
$k0	0x0	Kernel Register 1, used for the kernel operations
$k1	0x1	Kernel Register 2, used for kernel operations
$PC_Temp	0x2	This is equivalent to the epc in MIPS. Saves the PC in event of an error
$ErrMask	0x3	This register is a mask to determine if we are currently listening for specific errors or events. We don't want to get stuck in an error loop.
$ErrReg	0x4	A series of Flags to record the cause of an error, the codes will be defined later.
$kra	0x5	Kernel return address (To be used inside of the kernel)
$FlaggedInst	0x6	Holds the instruction that caused an interrupt if the interrupt happened because of an instruction
$HOLD	0x7	If set to 1, the processor will no longer increment PC until reset is triggered
$ReturnCode	0x8	Change the offset to be added to an address going into Main Memory.
$00	0x9	Constant 0
$Kbr	0xE	Branch Register for Kernel

Data Memory Layout:

Memory sections are changed by changing the offset register in the Kreg

Description	**Address**
Data Memory	0-256
Kernel Memory	257-512
Additional Instruction Memory	512-1024

Instruction Memory

The instruction memory contains the kernel and the active running program. It may also contain other program that are not running.

Kernel Instruction Memory	0x000 – 0x00f
Instruction Memory	0x010 – 0xfff

RHIPS Assembly Coding Conventions

General

The top 15 addresses of memory are reserved to be used in place of registers.

Procedure Calling Conventions:

When making a procedure call, all registers should be backed up to the stack. When returning from a procedure call, backed up registers should be restored.

PC conventions:

The PC should not be changed directly by the programmer

RA conventions:

Callee should be sure to back-up RA if multiple functions are being called, RA shouldn't be changed directly

A and V conventions:

A and V should only be set if returning values from or passing values to functions These registers can be loaded into other sections of memory to preserve them in sub-functions.

K register convensions:

These registers are to only be used by the kernel.

Temporary conventions:

These can be set temporarily by the compiler of the programmer, but it should be noted that long-term storage must be stored outside of the 15th address of memory.

IN/OUT conventions:

The input can't be written to but can be read from, and the output can't be read from but can be written to.

Branch Conventions and Notes:

The assembler should place the full address into the branch temporary register before the expected branch occurs. This should happen even if the branch does not happen.

Memory Allocation Notes:

Kernel memory is to only be used for the kernel and for temporarily storing instructions while working with the instruction memory. The instruction memory is what where the active program and kernel are stored. Finally, the main memory contains a stack and the user should not store beyond the stack pointer. The user must also store below the additional program memory line.

Multi-Cycle RTL

Below is a multi-cycle representation of the entire instruction set. There may be some differences than listed above, but the RTL below should be considered more complete.

All Instructions	Most General Types (A-Types / I-Types / etc)	Jump Types	CCP	CMP / TERM	SYSCALL

1	instMem[PC]								
1b	IR = instMem[PC]								
2	A = Reg[IR[7-4]] B = Reg[IR[3-0]] C = Reg[IR[11-8]]								
3	**A-Type** Result = A op B	**I-Type** Result = C op IR[7-0]	**L2R** Result = MainMem[IR[7-0]]	**J / JAL** Result = PC[15-11] | ZE(IR[11-0])	**L2M / JR** Result = C	**BEQ / BNE** Result = B-A	**CCP** Result = B	**CMP / Term** Reg[0xD]=1	**Syscall** Result = ZE(A)
4	**A-Type / I Type / L2R / JAL** Reg[IR[11-8]]=Result			**J** PC = Result **J DONE**	**JR** PC = Result **DONE**	**L2M** MainMem[IR[11-8]] = Result	**BEQ /BNE** If (ZERO && BRANCH) PC = Reg[Branch]	**CCP** Reg[0xD]=1	**CMP** B=Kreg[IR[11]]	**TERM** PC = 0x0	**Syscall** Reg[0xD] = 1
5	**A-Type / I-Type / L2R / L2M / BEQ /BNE <u>(If not branched)</u>** PC = PC + 1 **A-Type / I-Type/ L2R / L2M /BEQ /BNE DONE**						**CCP** Kreg[IR[11]]= Result	**CMP** Result = B	**TERM** Kreg[0x7] = 1 **TERM DONE**	**SYSCALL** KReg[0x4] = Result
6	**CCP** PC = PC + 1 **CCP DONE**			**CMP** Reg[0xD]=0			**SYSCALL** KReg[0x6] = IR			
7	**CMP** Reg[IR[7-4]] = Result					**SYSCALL** KReg[0x2] = PC				
8	**CMP** PC = PC + 1 **BNE DONE**					**SYSCALL** PC = 0x4 **SYSCALL DONE**				

Code snippets and RelPrime

RelPrime – Assembly Code

```
0x10 RELPRIME:
0x10 L2r m, 2
0x11 Or $a1,$a1, n
0x12 Or $a2, $a2, m
0x13 LOOP:
0x14 Jal GCD
0x15 Addi $a2, 1
0x16 Ori $t0, 1
0x17 Beq $v0, $t0, JUMPDONE
0x18 J LOOP
0x19 JUMPDONE:
0x1A J DONE
0x1B GCD:
0x1B And $v0, $v0, $0
0x1C Beq $a1, $v0, Zero
0x1D GCDLOOP:
0x1D Beq $a2, $v0, RETURN
0x1E Or $t1, $a1, $0
0x1F Or $t2, $a2, $0
0x20 Slt $t1,$t1, $t2
0x21 Beq $t1, $t0, DECB
0x22 J DECA
0x23 DECA:
0x24 Sub $a1, $a2
0x25 J GCDLOOP
0x26 DECB:
```

```
0x27 Sub $a2, $a2, $a1
0x28 J GCDLOOP
0x29 ZERO:
0x29 Or $v0, $v0, $a2
0x2A Jr $ra
0x2B RETURN:
0x2B Or $v0, $v0, $a1
0x2C Jr $ra
0x2D DONE:
0x2D And $v0, $v0, $0
0x2E Or $v0, $v0, $a2
0x2F Addi $v0, -1
0x30 Term
End of function
```

RelPrime – Assembled

```
0x10  RELPRIME:
0x10            1001 0001 0000 0010
0x11            0101 0011 0011 0010
0x12            0101 0100 0100 0001
0x13  LOOP:
0x13            1101 0000 0000 1100
0x14            0100 0100 0000 0001
0x15            1001 1000 0000 0001
0x16            0100 1110 0000 0000
0x17            1000 1110 0001 1010
0x18            0001 0000 1000 0010
0x19            0101 1111 1111 1000
```

```
0x1A  JUMPDONE:


0x1B            0101 0000 0010 1110
0x1C  GCD:
0x1C            0011 0101 0101 0000
0x16            0100 1110 0000 0000
0x17            1000 1110 0011 1110
0x1D            0001 0000 0011 0101
0x1E  GCDLOOP:
0x1F            0100 1110 0000 0000
0x20            1000 1110 0100 0011
0x21            0001 0000 0100 0101
0x2b            1000 1000 0011 0000
0x2d            1000 1001 0100 0000
0x2f            0000 1000 1000 1001
0x16            0100 1110 0000 0000
0x17            1000 1110 0011 1110
0x32            0001 0000 1000 0111
0x34            0101 0000 0000 0100


0x36  DECA:


0x36            1011 0011 0011 0100
0x38            0101 1111 1111 0000
```

```
0x3a  DECB:


0x3a            1011 0100 0100 0011

0x3c            1011 1111 1110 1000


0x3e  ZERO:


0x3e            1000 0101 0101 0100

0x41             1100 0000 0000 0010


0x43  RETURN:


0x43            1000 0101 0101 0011

0x45            1100 0000 0000 0010


0x47  DONE:


0x47            0111 0101 0101 0000

0x49            1000 0101 0101 0100
0x4A            0100 0101 1111 1111

0x4B            0001 0000 0000 0000
```

Additional Code Examples:

Loop, Branch, and subtraction example:

```
X = 0xF;
while(true){
      X=X-1;
      if(X==0){
            break
      }
}
```

```
0x10 L2r X, 0xF
0x11 LOOP:
0x12  Sub X, 1
0x13  Beq END ,$0, X
0x14     J LOOP


0x15  END:
```

Load a 16 bit immediate into a register

```
0x10 ori $t1, $0, 0xff

0x11 ls $t1, 8

0x12 ori $t1, $t1, 0xff
```

Assembler:

The processor has three types of instructions we must be able to assemble. Each type will be assembled in similar methods but are slightly different. Most instructions should be represented by directly relating the order of appearance to the bit order. For example, if operand1 appears before operand2, operand1 is represented first in machine code.

Note: ". . ." Indicates an arbitrary operation of the type

Assembling an Arithmetic Type:

Pseudo Code:
Destination = Operand1 . . . Operand2

Assembly Instruction Format:

Instruction Destination, Operand1, Operand2

Machine Code format:

[opcode, destination, operand1, operand2]

Example:

`add $t0 $t1 $t2` becomes

opcode	dest.	operand1	operand 2
0011	1011	1000	1001

Assembling a Jump Type:

Pseudo Code:
PC= PC[15-12] | Destination

Assembly Instruction Format:
Instruction Dest

Machine Code format:
[opcode, destination]

Example:

`j 0x2f`

opcode	dest./operand		
0101	0000	0010	1111

Assembling a Immediate Type:

Pseudo Code:
Dest = Data in Dest . . . SE[Immediate]

Assembly Instruction Format:

Instruction Address of Destination Immediate.

Machine Code format:

[opcode, Destination Addr., Sign Extended immediate]

Example:

```
L2r $t0, 4
```

opcode	Dest.	Immediate
1010	0101	0000 0100

```
L2r $t0, -255
```

opcode	Dest.	Immediate
1010	0101	1111 1111

RTL “Parts List”:

Component	Abbreviation	Description	General Implementation	Control Bits
PC	PC	Holds the address of the current instruction	Generic Register (16 bits)	PCWrite
Memory Manager	MemMan	Determines if we load an instruction from Kernel memory or Instruction Memory.	Special Verilog and combinational logic	InstOrKernel
Adder	Addr	Adds 1 to increment PC	Simple Adder verilog (16 bits in/out)	None
Instruction Memory	InstMem	Holds the current program to be executed	General Verilog memory	InstWrite
Kernel Memory	KernMem	Holds data for the kernel and additional kernel programs	General Verilog Memory	None
IR	IR	Holds the current instruction	Generic Register (16 bits)	IRWrite
Register File	RegFile	Main register file	Generic Register File with internal logic	RegWrite(A/B)
Sign Extender	SE	Extends to either 16 or 8 bits	Verilog Sign Extender (8-16 bits)	None
Multiplexers	MUX (13)	Choose between various things (See Datapath for specific instances)	Verilog MUX	Various control
A,B, C, Result Register	A,B, C, ResReg	Holds results between cycles	Generic Register (16 bits)	WriteEnable
Data Memory	DatMem	Long-term storage for the program data	Simple Verilog Memory	MemWrite/Read
Arithmetic Logic Unit	ALU	Does the operation defined by the instruction	Verilog Defined ALU	Zero, ALUOP

RTL Error Checking Methods (Taken from original RHIPS doc):

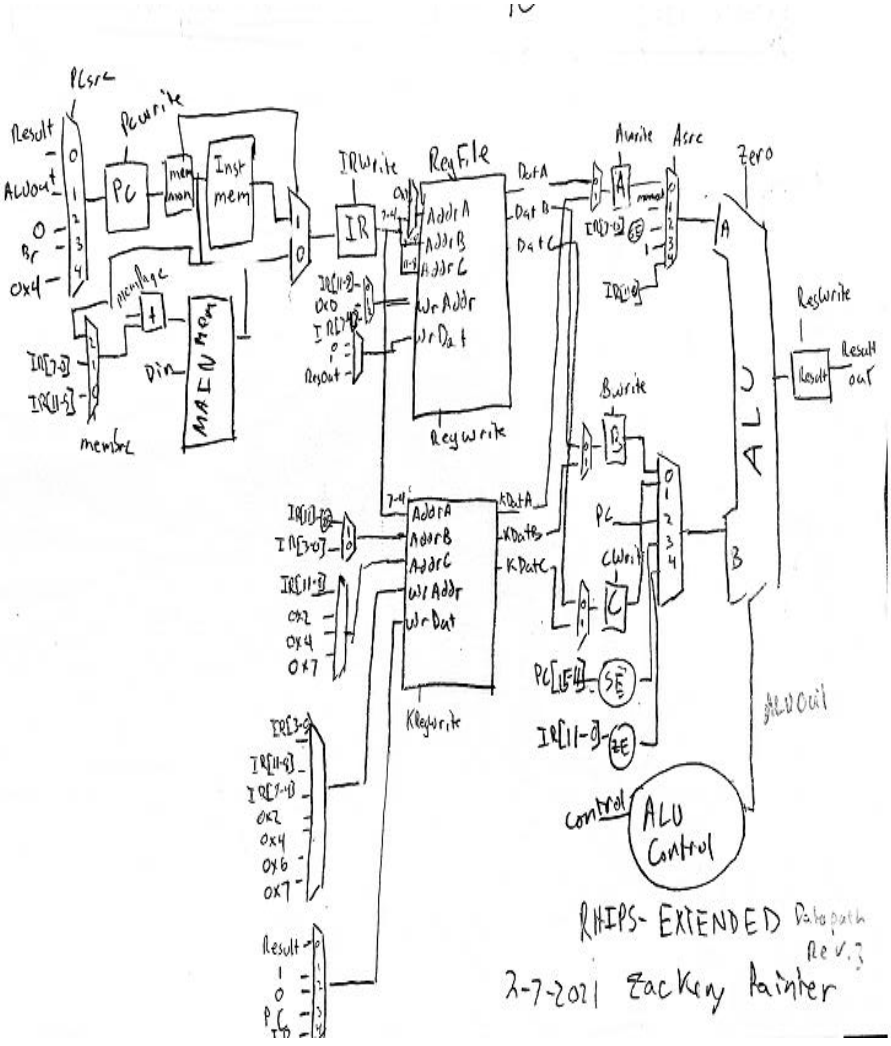
We double checked all the RTL for dependencies that were out of order and made sure that everything that needed to happen within the instruction was taken care of.

We also developed an automated testing system in JAVA to simulate our RTL and compare it to an expected result. This showed an error in our RTL relating to branching. We have fixed the error and amended the RTL and conventions to reflect the changes.

The test bench will be submitted after it is fully completed. We are working on expanding the test bench to run some tests on more complicated things. This isn't to replace Xilinx, but it will make some component logic testing easier than doing it by hand and give a second validation to the validity of our components.

Control Bit descriptions

KRegWrite	Enable Writing to the Kernel register file
AwriteEn	Enable Writing to the A register
BwriteEn	Enable Writing to the B register
CwriteEn	Enable Writing to the C register
ALUOP	Control the ALU
ResWriteEn	Enable Writing the the Res register
RAddrSrcA	Select the source of the address for port A for the reg file
KAddrBSrc	Select the source of the address for port B for the kernel reg file
Asrc	Select the source of the A port on the ALU
BSrc	Select the source of the B port on the ALU
KDatInSrc	Select the source of the data for the Kernel Data input port
RwriteAddrSr	Select the source of the write address for the register file
KAddrC	Select the source of the address for port C for the Kernel reg file
KWrAddrSrc	Select the source of the write address for the kernel reg file
memSrc	Select the source of the address going into the main memory
DatWrite	Enable writing on the data memory


PCsrc
PCwrite
Result
ALUout
0
Br
0x4
PC
mem addr
Instr mem
IRWrite
IR
RegFile
AddrA
AddrB
AddrC
WrAddr
WrDat
RegWrite
DatA
Dat B
Dat C
Awrite
Asrc
A
B
ALU
Zero
RegWrite
Result
Result out
B.write
C
PC
CWrite
ALUOut
MemRdC
MEM
Din
MemWr
KAddrA
KAddrB
KAddrC
KWrAddr
KWrDat
KRegWrite
control
ALU Control
RHIPS- EXTENDED Datapath Rev. 3
2-7-2021 Zachary Painter

Datapath

Testing and Verification

Implementation in Xilinx:

Part	Description
Generic Registers (PC, Result, A, B, C, IR)	A generic synchronous register as defined by the default register in Xilinx
Data Memory / Instruction Memory / Kernel Memory	All of the bigger memory blocks will be done using Xilinx's block memory engine.
Register File / Kernel Register File	This will be a Verilog file containing a set of output ports and input ports. The output ports will be assigned as registers. See the example in the implementation file
All Mux (Due to the large amount, assume all are similar, but with different bit amounts)	From experience, Xilinx MUX schematic objects are not great, so it will be a simple Verilog script to patch the input port to the output port based on a signal
ALU	This will be a Verilog combinational logic file
Control / ALU Control	This is a Verilog control module.
Zero Extender and Sign extender	A Verilog file to take an input and extend it to 16 bits by zero extending it or sign extending it.

Unit Testing:

Each individual unit will be exhaustively tested. However, parts with identical Verilog or schematic units will only be tested once. For example, not all muxes will be tested, but only one will be tested. Also, one register will be tested although 6 registers are needed.

Unit	Test Description
Generic Register	Test a normal write by reading the data back. Test the write enable by trying to clear the register by writing zero. Read the value again and compare. If the value is constant, it has passed.
Block Memory	Set a short address, write and then read. Try clearing by writing 0, but disable write enable. If passed, it should NOT be 0 Set a long address write and then read. Try clearing and writing 0, but disable write enable. If passed, it should NOT be 0
Register File	Test read and writing by writing all bits and reading them. Next, check if Write enable works by using the test described in generic register.
MUX	Attach a mux to 3 different values. Switch between them and check if the value at the output of the mux matches the expected value
ALU	Do one of each operation and check that it has the correct result. Check overflow detection by adding really big numbers.
Control	Will not be tested yet
Zero Extender / Sign Extender	Put a value on the input and compare the result of the output with the expected result

Hardware Integration Plan

The hardware integration will work in an "onion-like" design. Testing will start with individual components and gradually work to larger subsystems. The subsystems will be tested individually and then combined to larger sections of the completed datapath. The result will be the entire data path. The subsystems and correct tests are listed below.

A separate schematic was created for each test with a certain number of debug ports and ports to enter data directly into the path. Later, more complicated tests, incorporated a control module so the control did not have to be set manually. The Control module had been tested previously and was found to work correctly. Each subsystem will have it's own schematic and subsequent tests will incorporate those schematics as independent symbols. Each section will not need to know exactly how that subsection functions, but it should expect a correct value will be placed at the output(s) of the schematic's symbol.

Below is a diagram determining the testing regions and the tests that will be performed.

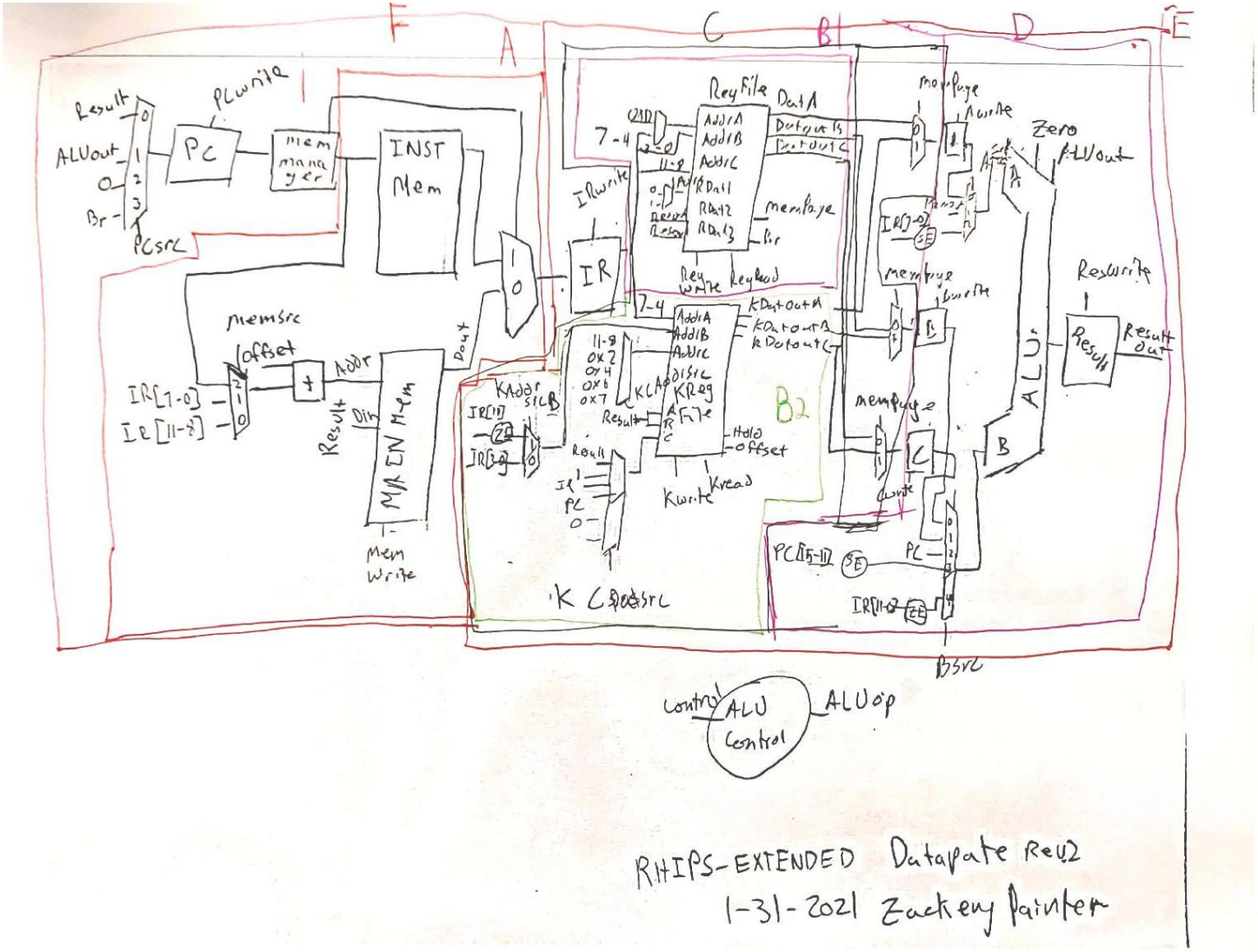


Figure 4- Above shows the different planned stages of testing

Sub-Section Descriptions

Section Name	Components Tested	General Description
A	Inst Mem, MAINMem, Mux1,2, Addr	Test InstMem and Main Mem combined and switching between them
B1	RegFile	Test selecting from different Address inputs and data inputs and read and write out of the register
B2	KernFile	Test selecting from different Address inputs and data inputs and read and write out of the register
C	RegFile, KernFile	Test both B1 and B2 and select between the 2 Kernel Files and expect the result
D	A,B,C,Result,ALU	Test reading out of the A,B,C registers and doing an operation in the ALU. Then store in Result
E	RegFile,KRegFile,A,B,C,ALU	Reading in and out of the register files and into the ALU. Store into Result and back into a register
F	InstMem, MainMem, RegFile, KRegFile, A, B, C, ALU, Adder	Test entire datapath

Sub-Section Test Detailed Specification

A. Test A
    1. This test assumes that a schematic has already been created
        i. Place a few dummy instructions into coe files for the Main Memory and Instruction Memory block memory symbols.
        ii. Place an address of a known instruction on the line and choose the output of the Instruction Memory. Check that the output value matches the expected value
        iii. Repeat step ii. for the Kernel Memory as well.
B. Test B
    1. Test B1
        i. Run the following tests for RegFile independently
        ii. Write into the place specified by the selected MUX value. Repeat for all mux entries on ALL ports.
        iii. Next, test reading by cycling through all possible MUX values on ALL ports, simultaneously.

iv. This test will not test read/write enable because that should be tested in the unit test module.

2. Test B2 – This test extends B1
    i. Repeat B1 but for KRegFile

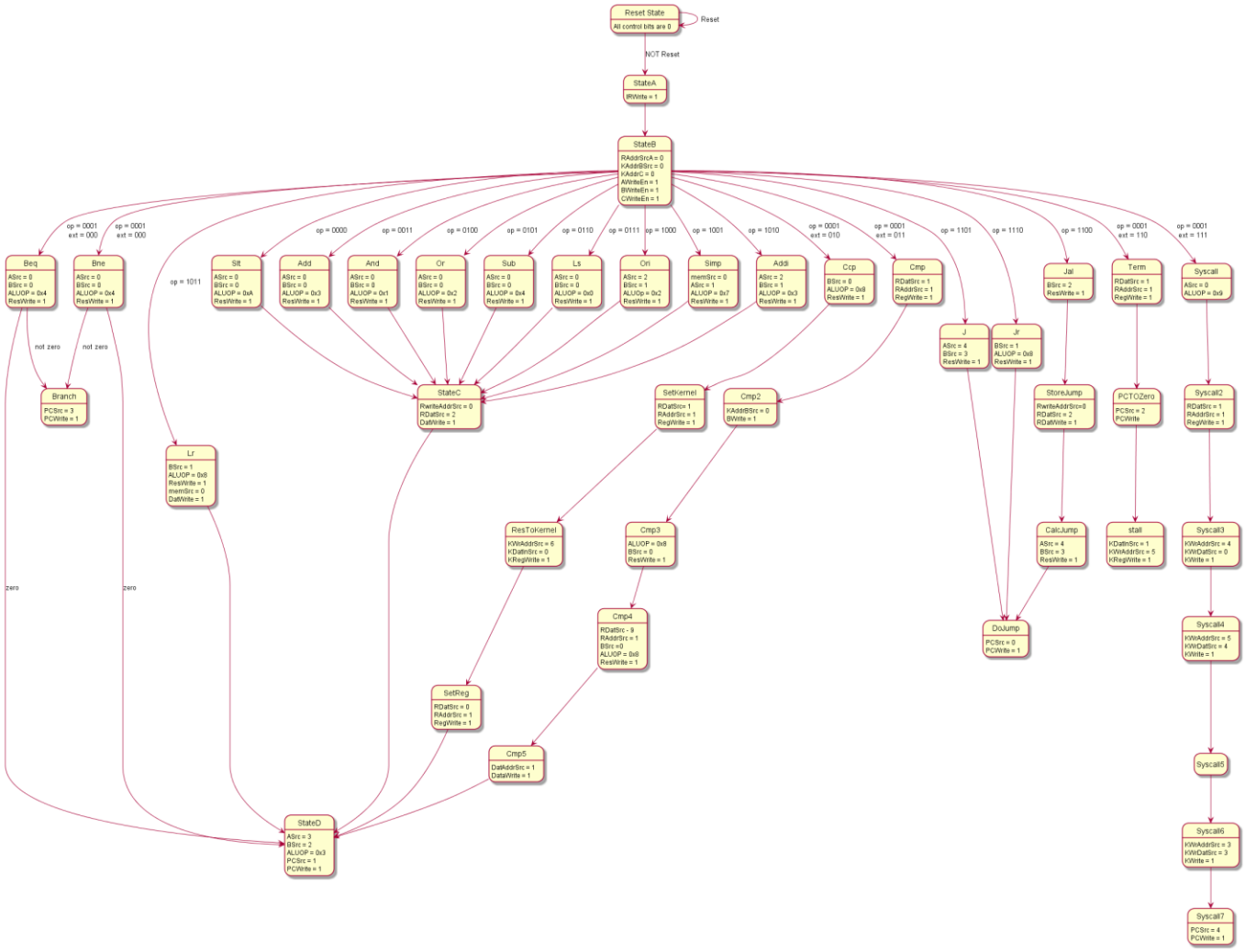
C. Test – C
    i. B1 and B2, but select the correct output from the output mux.

D. Test –D
    i. Test the ALU by loading into A B, C and do an operation and put into result

E. Test – E
    i. Combine C and D.

F. Test -F
    i. Add E and the rest of the data path
    ii. Place test instructions and run the instruction and read the output.
    iii. There should be entire datapath now

Finite State Diagram – General Control



ALU Control Diagram

ALUOpCode	Operation
0x0	Left shift
0x1	And
0x2	Or
0x3	Add
0x4	Sub
0x7	Pass Through A
0x8	Pass Through B
0x9	No Operation
0xA	Set less than (A < B)

Branch control:

Branching is determined by the main control unit. The control system reads the output of the zero bit and determines if it branches.

Timing

Total Cycles to run relprime: 949377 cycles

Total Instructions: 46748

Minimum period: 25.162ns

Maximum frequency: 39.742MHz

```
Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 25.162ns (Maximum Frequency: 39.742MHz)
   Minimum input arrival time before clock: 11.840ns
   Maximum output required time after clock: 5.148ns
   Maximum combinational path delay: No path found

=========================================================================
```

APPENDIX – DESIGN JOURNAL – ZACKERY PAINTER

Zackery Painter Work Log

Milestone 1:

1-8-2020 – [2 hrs] Started to work on M1 and review last quarter's design document. I started to find changes I wanted to make and start writing it down.

1-11-2020 [4 hrs] Finish up writing the design journal for M1. Re-formatting last quarter's journal and re-assembling the code based on the new instruction set I designed. I also went through and wrote the RTL for the new Ext-Type instructions

- 11-17-2020 7:00- 10:00PM
    - Added commenting ability for the assembler.
    - Started working on the Kernel and switching between Kernel and User memory.
    - Added $kra (Kernel return address) to prevent tampering with ra
    - Added a register to capture the instruction that caused an interrupt
    - Added a hold register
    - Added a very simple memory manager layout
        - If address is < 0x1000 then choose inst mem.
        - If address is > 0xfff then "spoof" a system call to enable kernel memory mode and OR 0x0fff with the real address. For example INST[0x3fff] will actually be stored in KMEM[0x3fff]
    - Started to work on system call documentation (Not in design doc yet)
    - Define interrupt codes for the kernel. (Not in design doc yet)
    - Started making sure M2 is completed, review RTL start working on Multi-cycle FSM.
        - Start comparing created parts with what I had from last quarter to make sure I still have everything
    - Added an IDLE loop to kernel (Do nothing until we see an interrupt (Located at PC = 0x0)
    - Need to do:
        - Finish M2.
        - Get assembler to a better state (Multi-file / Kernel-level assembling)
            - Work on getting addresses handled better
            - Generating object files
            - Re-implement Human-readable and verify functions
            - Write assembling instructions for EXT type
        - Drawing up data path
        - Finish writing kernel
- 11-18-2021 8:00-10:30PM

    - Removed following instructions (Memory manager will handle what they do)
        - KKM2IMM
        - IMM2KMM
    - Re-wrote RTL to closer match the layout of a multi-cycle datapath.
        - I didn't test any of the new code because it is still the same as the last quarter, but I just re-wrote it.
- 11-18-21 11:23-12:07
    - Added "Parts List" to the design document
        - Copied known good parts from the single cycle datapath from RHIPS
        - Added A, B, and Result registers to store between cycles.
    - Added the previous notes on RTL testing. (Will upload our testbench later)
    - Started working on a plan to verify Ext-Types
    - Started to work on control for this datapath
    - Added some control bit descriptions. (Need to finish still)
- 11-19-2021 7:30-11:00
    - Started drawing a new multicycle datapath
    - Started working out control and logic.
    - (Need to implement branch logic and jump logic)
    - Start to go back and add components to the component list
    - I ended up re-making the entire parts list and control bit list
    - ****RE-IMPLEMENT BRANCH LOGIC!!!***
    - Besides getting Branch/Jump logic implemented, M2 is done and I have a datapath
    -
- 11-20-2021 3:15-4:54
    - I forgot my glasses so I didn't get much done
    - Started working on implementing branch and jump logic
    - I submitted M2, then continued to work on the FSM diagram
- 11-25-2021 10:30PM – 1:24 AM (1-25-2021)
    - Worked on finish up RTL (I forgot to log another day I worked a few hours on straightening it out)
    - Color coded cycles so it's easy for me to distinguish.
    - Most instructions have almost identical RTL for the top half of the diagram.
    - Screenshot for this stage is below
    -
    -

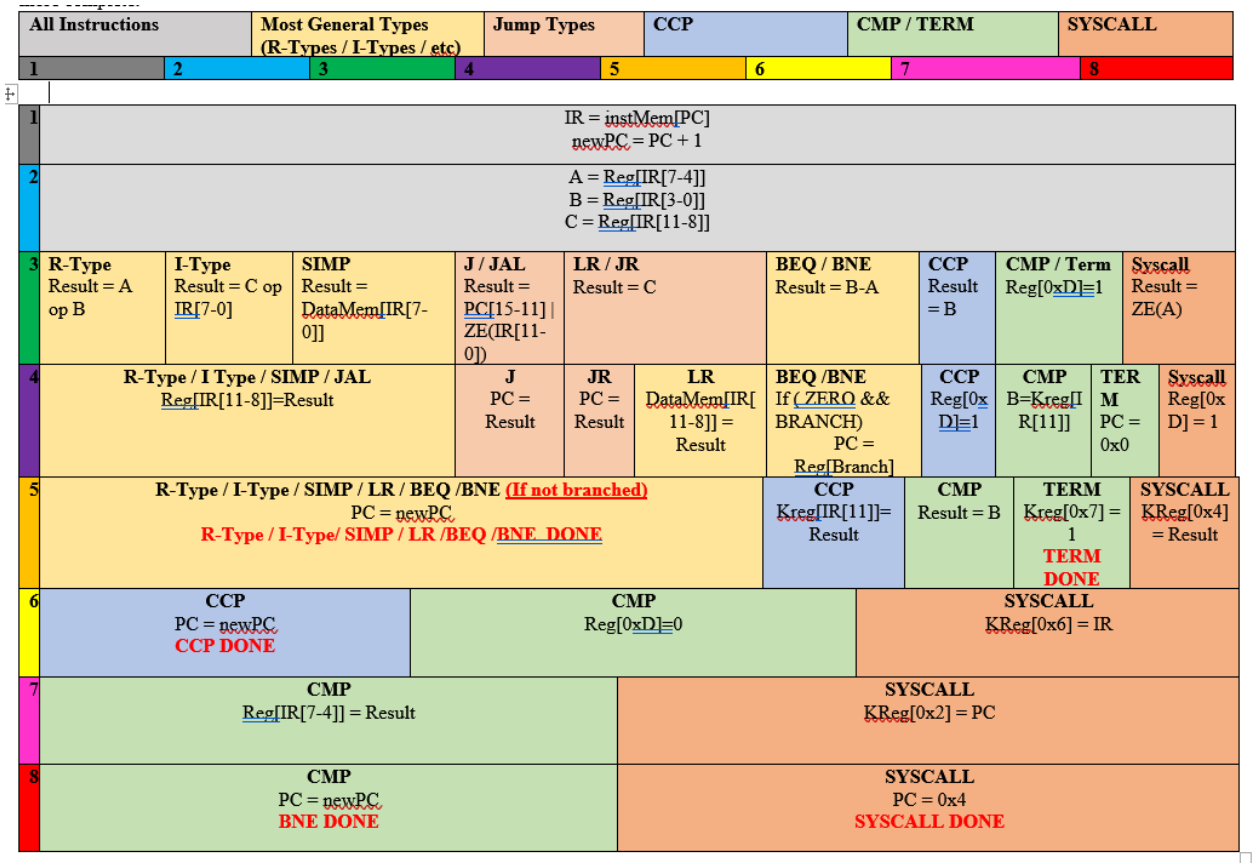


Figure 5- Screenshot for RTL on 1-25-2021

- 1-26-2021 7:36PM – 2: AM (~6.5 hrs.)
  - Started to move along M3. I'm a bit behind. Hoping to make it up tonight
  - Finished the Datapath, had to go back and add a good amount of control bits
  - I realized I don't have logic for hardware level interrupts.
    - I can add these later. I have support for software interrupts.
  - I'm going to either take out or copy over the RTL I have been working on for individual instructions. I have a full RTL, but not the individual one fixed in the documentation
    - This is complete now
  - Finished writing specs on Unit testing
  - Finished working on subsections
    - A-G
  - My implementation decisions:
    - I decided to break up the datapath based on the cycle that it would be completing as the full instruction. This allowed me to test fetching, decoding, executing, memory access, and various switching signals independent of each other.
    - The architecture design and the decision to include separate memory segments has created challenges that I might not otherwise have. For example,

there are a large amount of muxes that will be required to simply switch between Kernel and User space. Additionally, the partial support for interrupts has greatly complicated the design as some addresses must be hard-coded into the processor (0x4, 0x7, etc.). This was needed to record PC, causes, etc.
        - Additionally, I decided to include a separate port for the immediate on the ALU because it allowed me to not have to compromise an A, B, or C input. I can switch to using it based on the ALUOpcode. I think I can make the ALUControl give up to 16 bits worth if instructions, as it does not have to directly correlate to the actual OpCode of the instruction. (It can't because of the ext-code)
    - I still need to re-visit my components and testing for that. I think I have some old tests that should work fairly well. They proved to work well last quarter and most are basic enough for me to adapt here as well. (For unit tests)
    - I copied some old tests that I wrote last quarter. I did all the component and implementation testing last quarter so I know these work. I need to go back and re-name them to match my design doc, but they're all there. It's rather late and I have registration in a few hours so I am going to stop working for the night.
- 1-31-2021 : 2:00-5:00
    - I re-wrote my Datapath and Memory Layout once again
- 2-1-2021 : 1:00-4:00 5:00-6:00
    - Started working on FSM
    - I started implementing my plan, but ran into a weird error with the block memory.
    - I ended up fixing this by re-generating it until it worked
- 2-1-2021-2-2-2021 : 11:00PM-1:05AM
    - I started implementing my first stage. Most tests are implemented. All passed!
    - I need to go back and fix some RTL and datapath from some errors I found while writing control, I have it fixed on paper, but I need to scan and upload still.
    - I will most likely implement Stage B(1,2) tomorrow if I have time. (The milestone won't reflect that)
    - Currently Completed:
        - A – Yes
        - B1 – No
        - B2- No
        - C – No
        - D – No
        - E – No
        - F – No
- 2-7-2021 3:55-6:30 10:15-3:00AM
    - Started re-doing datapath (Again) to facilitate errors I found while working on the implementation
    - I finished B1 and started B2
    - Finished B2.
- 2-8-2021 9:00-11:00

    - I finished part C and started working on Part D
        - Including Kernel or Register selection bits (Address 0xD)
- 2-8-2021 7:00-12:00AM
    - I had to change the design rules of the Zero Extender because Xilinx refused to simulate it anymore for some reason, even though it works in other places
    - I finished Test D. I'll start on Test E soon.
- 2-9-2021 10:11-2:00AM
    - Fixed some control bits and started re-doing the FSM
    - Finished the FSM diagram.
- 2-10-2021 1:30 – 5:00
    - Finished Control
    - Tried to finish M5 before it was due.
- 2-11-2021 7:00 – 3AM
    - I put the entire datapath together, but have not tested it yet, or updated my documentation.
    - I spent a good 15-20 minutes just getting the schematic to compile
- 2-12-2021 6:30-??
    - I found the issue with the garbage data (It was an issue with the sign extender)
    - Fixed many other issues, especially with timing,
    - Still working on validating everything works
- 2-14-20201
    - I forgot to record it today, but I did a lot of work on the datapath
- 2-15-2021
    - Also forgot to do it today
    - But I finished the datapath and testing. Everything is done.
    - I started writing relprime
- 2-16-2021 6:30-11:33
    - Keep debugging relprime
    - I also finished the assembler
    - I'm still figuring out what's going on
- 2-17-2021 12:00-3:45AM
    - I kept working on relprime.
    - Added a "dead" loop at address 0 of the coe file so that the processor stalled when started.
    - The control waits for a start signal then jumps to 0x4 (future will be kernel) then 0x15 (user space)
    - Worked on various improvements to the assembler to automatically inject the "kernel" into the coe file
    - I have run it for a while and didn't ever get a good result, It's 4am though so I'm going to bed now.
    - I'll debug it later. Here's a screen shot of what my debug file from my assembler looks like
    - (slt is all zeros so it makes sense to just make a ton of those, it does nothing)

```
0: term : 0001011000000000
1: slt $0, $0, $0 : 0000000000000000
2: slt $0, $0, $0 : 0000000000000000
3: slt $0, $0, $0 : 0000000000000000
4: slt $0, $0, $0 : 0000000000000000
5: j 15 : 1101000000001111
6: slt $0, $0, $0 : 0000000000000000
7: slt $0, $0, $0 : 0000000000000000
8: slt $0, $0, $0 : 0000000000000000
9: slt $0, $0, $0 : 0000000000000000
10: slt $0, $0, $0 : 0000000000000000
11: slt $0, $0, $0 : 0000000000000000
12: slt $0, $0, $0 : 0000000000000000
13: slt $0, $0, $0 : 0000000000000000
14: slt $0, $0, $0 : 0000000000000000
15: and $t0, $t0, $0 : 0100011101110000
16: ori $t0, 2 : 1000011100000010
17: or $t1, $0, $in : 0101100000001010
18: l2m $ra, 0 : 1011001000000000
19: or $a0, $0, $t1 : 0101001100001000
20: or $a1, $0, $t0 : 0101010000000111
21: l2m $t0, 1 : 1011011100000001
22: l2m $t1, 2 : 1011100000000010
23: jal gcd : 1100000000010011
24: and $t0, $0, $0 : 0100011100000000
25: ori $t0, 1 : 1000011100000001
26: bne finish, $v0, $t0 : 0100111000000000,
1000111000010001,
0001000101010111
29: l2r $t0, 1 : 1001011100000001
30: l2r $t1, 2 : 1001100000000010
31: addi $t0, 1 : 1010011100000001
32: or $out, $0, $t0 : 0101101100000111
33: term : 0001011000000000
34: beq returnb, $a0, $0 : 0100111000000000,
1000111000100011,
0001000000110000
37: beq done, $a1, $0 : 0100111000000000,
1000111000100101,
0001000001000000
40: slt $t0, $a1, $a0 : 0000011101000011
41: and $t1, $0, $0 : 0100100000000000
```

- 2-17-2021 –11-12, 2-5PM
    - Finally got relprime to run fully!!
    - Next, I will try to get timing done.
    - Probably won’t get it done before 5

APPENDIX – TESTS

Below is an example of the tests performed.

```
[PASS] TEST 1
[PASS] TEST 2
[PASS] TEST 3
[PASS] TEST 4
[PASS] TEST 5
[PASS] TEST 6
[PASS] TEST 7
[PASS] TEST 8
[PASS] TEST 9
[PASS] TEST 10
[PASS] TEST 11
[PASS] TEST 12
All test Passed!
```

APPENDIX – PROOF OF RESULTS

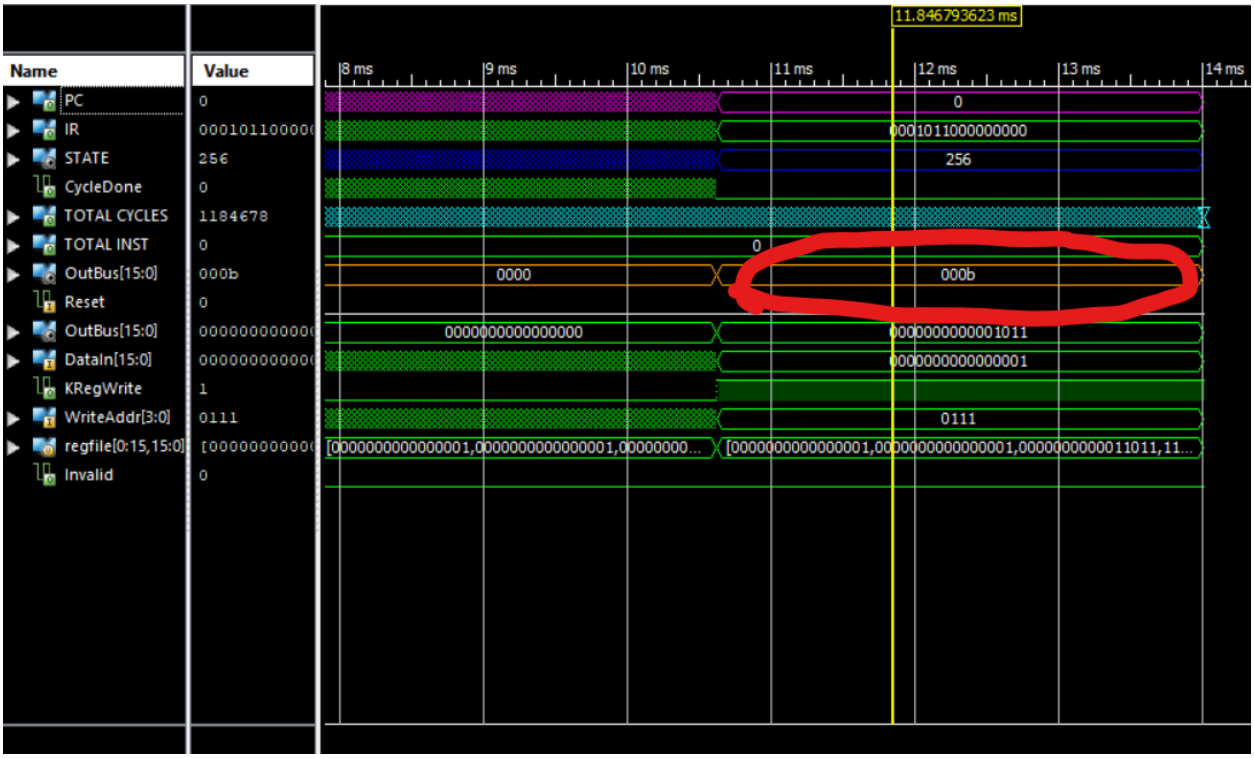


Figure 6 - Waveform of final RelPrime and Kernel