Stack Based Architecture Final Report

Team 2v (Jinhao Sheng, Luke McNeil, Austin Swatek)

Table of Contents

Introduction

The general purpose processor we are making is stack-based and will make use of two stacks of registers: one as a stack of the instructions used for the operations in a function, and the other for keeping track of the return values of functions. This processor has two instruction formats: O-type and A-type. O-type instructions have a 4-bit opcode, 12-bit function, and are for instructions that require no arguments. O-types that require jumps use direct addressing, while A-types that require jumps use pseudo-direct addressing. A-type instructions have a 4-bit opcode, 12-bit immediate value or address, and are necessary for functions that require arguments. Since it is a stack-based processor, functions are called on a last-in-first-out (LIFO) basis, with the caller function pushing the arguments that the callee expects to the top of the stack. The callee pops the arguments off of the stack, and before its return call, puts the required return values back on the stack. To return, the callee pops its return address off the return stack and jumps to that address.

*****Important Note***:** Both the main stack and the return stack are 64 registers big. If one tries to push to the stack when it is full the data that is in the bottom position of the stack will be lost.

Instruction Set Design

Instruction Types

Format Type	Size	Structure	Description
O	2 bytes	OP 4 | FUNCT 12	OP - basic operation of the instruction FUNCT - sets the variant of the operation This format type is used for instructions that take no arguments.
A	2 bytes	OP 4 | IMM/ADDR 12	OP - basic operation of the instruction IMM/ADDR - a 12 bit constant or address This format type is for any instruction that takes one argument which is either an immediate or address.

Instructions

Name	Type	Argument	Description	OP	Funct
add	O		Pop the top two values off the stack, add them, and put the result on the stack.	0x0	0x000
beq	A	label	Pop the top two values off the stack. If they are equal, then branch to label.	0x1	
bez	A	label	Pop the top value off the stack. If it is zero then branch to label.	0x2	
dup	O		Push onto the stack a duplicate of the value currently on top of the stack.	0x0	0x001
drop	O		Pop the top value off the stack, throwing it away.	0x0	0x002
halt	O		Jump back to this instruction. This is a good way to stop the program.	0x0	0x003
getin	O		Read a 16 bit number from input and push it to the stack.	0x0	0x004
j	A	target	Jump to target.	0x3	
jal	A	target	Jump to target, and push onto the return address stack the address of the next instruction.	0x4	
js	O		Pop the top value off the stack and jump to that address.	0x0	0x005
lui	A	immediate	Shift the immediate left by 12 bits and then push it into	0x8	

			the stack		
over	O		Push onto the stack the value of the second element on the stack.	0x0	0x006
or	O		Pop the top two values off the stack, or them, and put the result on the stack.	0x0	0x007
pop	A	address	Pop the top value off the stack and put it in memory at address.	0x5	
push	A	address	Push the value at the specified address onto the stack.	0x6	
pushi	A	immediate	Push onto the stack sign extended immediate.	0x7	
return	O		Pop the top element off the return address stack, and jump there.	0x0	0x008
slt	O		Pop the top two elements off the stack, and push a 1 to the stack if the second from the top element is less than the top element. Otherwise, push a 0.	0x0	0x009
sub	O		Pop the top two values off the stack, subtract them, and put the result on the stack.	0x0	0x00A
swap	O		Swap the top two elements on the stack.	0x0	0x00B
getin2	O		Read a 16 bit number from input2 and push it to the stack.	0x0	0x00C

RTL

Notes:

- stack is 64 registers
- Rstack is the return address stack and is 64 registers

add, sub, or

```
inst = Mem[PC]
stack[0] = stack[1] OP stack[0]
stack[1] = stack[2]
stack[2] = stack[3]
...
stack[63] = 0
PC = PC + 2
```

beq

```
inst = Mem[PC]
A = stack[0]
B = stack[1]
stack[0] = stack[2]
stack[1] = stack[3]
...
stack[62] = 0
stack[63] = 0
if (A-B==0):
    PC = (PC[15:13])...(inst[11:0] << 1)
else: PC = PC + 2
```

bez

```
inst = Mem[PC]
A = stack[0]
stack[0] = stack[1]
stack[1] = stack[2]
...
stack[63] = 0
if (A==0):
    PC = (PC[15:13])...(inst[11:0] << 1)
else: PC = PC + 2
```

dup

```
inst = Mem[PC]
stack[63] = stack[62]
stack[62] = stack[61]
...
stack[1] = stack[0]
PC = PC + 2
```

drop

```
inst = Mem[PC]
stack[0] = stack[1]
stack[1] = stack[2]
...
stack[63] = 0
PC = PC + 2
```

halt

```
inst = Mem[PC]
```

getin or getin2	**j**	**jal**
inst = Mem[PC] stack[63] = stack[62] stack[62] = stack[61] ... stack[1] = stack[0] stack[0] = INPUT or INPUT2 PC = PC + 2	inst = Mem[PC] PC = (PC[15:13])...(inst[11:0] << 1)	inst = Mem[PC] Rstack[63] = Rstack[62] Rstack[62] = Rstack[61] ... Rstack[1] = Rstack[0] Rstack[0] = PC + 2 PC = (PC[15:13])...(inst[11:0] << 1)
js	**lui**	**over**
inst = Mem[PC] A = stack[0] stack[0] = stack[1] stack[1] = stack[2] ... stack[63] = 0 PC = A	inst = Mem[PC] stack[63] = stack[62] stack[62] = stack[61] ... stack[1] = stack[0] stack[0] = inst[3:0] << 12 PC = PC + 2	inst = Mem[PC] stack[63] = stack[62] ... stack[2] = stack[1] stack[1] = stack[0] stack[0] = stack[2] PC = PC + 2
pop	**push**	**pushi**
inst = Mem[PC] A = stack[0] stack[0] = stack[1] stack[1] = stack[2] ... stack[63] = 0 Mem[PC[15:13]...(inst[11:0] << 1)] = A PC = PC + 2	inst = Mem[PC] stack[63] = stack[62] ... stack[1] = stack[0] stack[0] = Mem[PC[15:13]...(inst[11:0] << 1)] PC = PC + 2	inst = Mem[PC] stack[63] = stack[62] ... stack[1] = stack[0] stack[0] = SE(inst[11:0]) PC = PC + 2
return	**slt**	**swap**
inst = Mem[PC] A = Rstack[0] Rstack[0] = Rstack[1] Rstack[1] = Rstack[2] ... Rstack[63] = 0 PC = A	inst = Mem[PC] A = stack[0] B = stack[1] if (B < A): stack[0] = 1 else: stack[0] = 0 stack[1] = stack[2] stack[2] = stack[3] ... stack[63] = 0 PC = PC + 2	inst = Mem[PC] A = stack[0] stack[0] = stack[1] stack[1] = A PC = PC + 2

Addressing Modes

Instructions	Format Type	Addressing Modes
js, return	O	Direct
j, jal, beq, bez	A	Pseudo-Direct

Pseudo-Direct Example

- Going from 16 bit address to the 12 bits in the instruction
    a. Shift the 16 bit number right 1
    b. Chop off the 4 most significant bits
    c. Use this 12 bit number in the instruction ADDR field.
- Going from 12 bits in the instruction to a 16 bit address
    a. Shift the 12 bits to the left 1
    b. Put on the front of these 13 bits the 3 most significant bits from $PC.
    c. Use this 16 bit number as the address to go to.

Direct Example

- Here the jump is looking at the value in a 16 bit register. The address to jump to is simply those 16 bits.

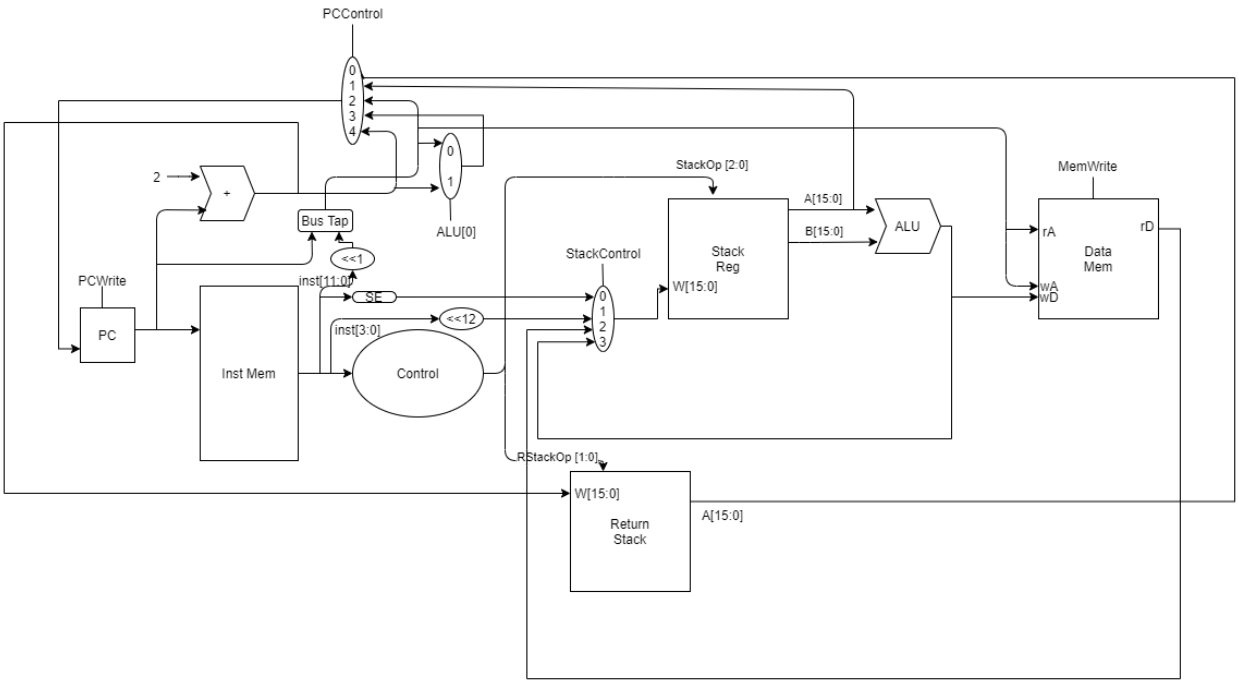
Procedure Calling Conventions

To prepare to call a function the caller must put all arguments that the callee expects to receive on top of the stack. Then the caller must call jal to go to the callee. This command will push the return address to the return address stack. The callee's responsibilities are to pop the arguments off the stack and leave on the stack any return values. The callee will then do the return instruction which will pop the top element off the return address stack before going back to the correct spot.

Included below in the code fragments section is an example of nested function calling.

Final Model

Datapath Design



Required Components

1. Multiplexer
    a. Inputs - at most 8 of i[15:0]
    b. Outputs - o[15:0] and goes to PC
    c. Control - 3 bits vary from 000 to 111
2. Sign Extender (12 bit -> 16 bit)
    a. Inputs - a[11:0] is the thing to be sign extended
    b. Outputs - r[15:0] is the sign extended 16 bit result
    c. Control Bits - none
    d. Description - This component sign extends a 12 bit bus into a 16 bit
    e. RTL Symbols - SE
3. Left Shifter
    a. 1 bit (12 bit -> 13 bit)
    b. 12 bit (4 bit -> 16 bit)
4. ALU with special operations
    a. select A
    b. select B
    c. if A==B: 1
       else: 0
    d. if A==0: 1
       else: 0
    e. if B<A: 1
       else: 0
5. Adder
    a. Inputs - a[15:0], b[15:0] are two things to add
    b. Outputs - r[15:0] the unsigned result of adding a[15:0] and b[15:0]
    c. Control Bits - none
    d. Description - adds the 2 inputs as unsigned values
    e. RTL Symbols - +
6. 16 Bit Register
    a. Inputs - D[15:0] is the data to write, CLK the clock
    b. Outputs - O[15:0] is the output of the value in the register
    c. Control Bits - write determines if the PC should be set to value of D[15:0]
    d. Description - This is a component that can remember values and is able to be changed
    e. RTL Symbols - PC
7. Instruction Memory Block
    a. Inputs - ra[15:0] is the read address and CLK is the clock
    b. Outputs - d[15:0] is the data in memory at ra[15:0]
    c. Control Bits - none
    d. Description - This component is storage for instructions that are being executed
    e. RTL Symbols - Mem
8. Data Memory Block

    a. Inputs - ra[15:0] is the read address, wa[15:0] is the write address, wd[15:0] is the write data, and CLK is the clock
    b. Outputs - d[15:0] is the data in memory at ra[15:0]
    c. Control Bits - write tells whether or not to write to memory
    d. Description - This component is storage for data that is pushed by the programmer
    e. RTL Symbols - Mem
9. Control Unit
    a. Inputs - inst[15:0]
    b. Outputs - see control bits above
    c. Control Bits -
    d. Description - Takes in the instruction and then sets all of the control bits throughout the datapath to correctly run that instruction
    e. RTL Symbols - none
10. Stack Register
    a. Inputs - w[15:0] is what should be written to the top of the stack if anything, CLK is the clock, and reset sets all of the registers to 0 if it is high on the negedge
    b. Outputs - a[15:0] is the top value of the stack, b[15:0] is the value below the top
    c. Control Bits - stackOP[2:0] is what stack operation should be done, these include
        i. 000, none - used for halt and j, does nothing to the stack
        ii. 001, push - used for many instructions such as pushi, pushes to the stack what is on w[15:0]
        iii. 010, pop and replace - used for add, sub, or, and slt, pops the top off and replaces the next with w, which should contain a result of doing something with a[15:0] and b[15:0]
        iv. 011, pop - used for many instructions such as drop, top element is popped
        v. 100, pop2 - used for beq, pops top 2 things and uses them
        vi. 101, swap - used for swap, doesn't change stack size and swaps top two
    d. Description - This components output will always be the top two things currently on the stack. The stack can be changed by different stackOP and providing different values to w[15:0] to decide what to push or replace, data is written to the stack at the negative edge of the clock
    e. RTL Symbols - stack, Rstack
11. Merger (3 bit and 13 bit -> 16 bit)
    a. Inputs - a[2:0] and b[12:0]
    b. Outputs - r[15:0] which is a 16 bit bus with a[2:0] corresponding to its top 3 bits and b[12:0] being the bottom 13 bits
    c. Control Bits - none
    d. Description - This component takes a 3 bit and 13 bit input and combines them into 1 bus where the 3 bit input is now the most significant 3 bits.
    e. RTL Symbols - … (we used a … in the RTL to show that two busses should be joined)

Control Signals

1. PCControl - 3 bits
    - 0 = top of return stack
    - 1 = top of stack
    - 2 = PC[15:13]...(inst[11:0] << 1)
    - 3 = PC[15:13]...(inst[11:0] << 1) or (PC + 2)
    - 4 = PC + 2
2. StackControl - 3 bits
    - 0 = SE(inst[11:0])
    - 1 = inst[3:0] << 12
    - 2 = memory read data
    - 3 = ALU result
    - 4 = input
    - 5 = input2
3. StackOP - 3 bits
    - 0 = none
    - 1 = push
    - 2 = pop and replace
    - 3 = pop
    - 4 = pop 2
    - 5 = swap
4. RStackOP - 2 bit
    - 0 = none
    - 1 = push
    - 3 = pop
5. ALUOP - 4 bit
    - 0 = add
    - 1 = sub
    - 2 = and
    - 3 = or
    - 4 = xor
    - 5 = select A
    - 6 = select B
    - 7 = return A==B ? 1 : 0;
    - 8 = return A==0 ? 1 : 0;
    - 9 = return B<A ? 1 : 0;
6. PCWrite - whether or not to write to PC
7. MemWrite - whether or not to write to Data Mem

Control

('X' means that the control bit doesn't affect the result)

Name	Type	OP	Funct	stackOP	rstackOP	ALUOP	stack Control	PC Control	MemWrite	PCWrite
add	O	0x0	0x000	2	0	0	3	4	0	1
dup	O	0x0	0x001	1	0	5	3	4	0	1
drop	O	0x0	0x002	3	0	X	X	4	0	1
halt	O	0x0	0x003	0	0	X	X	X	0	0
getin	O	0x0	0x004	1	0	X	4	4	0	1
js	O	0x0	0x005	3	0	X	X	1	0	1
over	O	0x0	0x006	1	0	6	3	4	0	1
or	O	0x0	0x007	2	0	3	3	4	0	1
return	O	0x0	0x008	0	3	X	X	0	0	1
slt	O	0x0	0x009	2	0	9	3	4	0	1
sub	O	0x0	0x00A	2	0	1	3	4	0	1
swap	O	0x0	0x00B	5	0	X	X	4	0	1
getin2	O	0x0	0x00C	1	0	X	5	4	0	1
beq	A	0x1		4	0	7	X	3	0	1
bez	A	0x2		3	0	8	X	3	0	1
j	A	0x3		0	0	X	X	2	0	1
jal	A	0x4		0	1	X	X	2	0	1
pop	A	0x5		3	0	X	X	4	1	1
push	A	0x6		1	0	X	2	4	0	1
pushi	A	0x7		1	0	X	0	4	0	1
lui	A	0x8		1	0	X	1	4	0	1
addi	A	0x9								

To test control we will simply feed each instruction through control. Then we will check to see if all of the control signals in the table which are not "X" are set appropriately. If they are correct, then our control component works.

Testing

Unit Tests

1. Multiplexer
    - The test will put different inputs on each of the input wires, it will then go through each selection bit combination and see if the output is correct.
2. Sign Extender (12 bit -> 16 bit)
    - The test will give several 12 bit wires as input and see if the correct value is returned in 16 bit form. The value of the number representing signed binary should never change.
3. Left Shifter
    - Give various inputs, then verify that the number was shifted left the correct number of places and that zeroes were filled on the right.
4. ALU with special operations
    - Test each operation with various inputs. The operations are detailed above in the Required Components section.
5. Adder
    - Test that the adder does unsigned addition on various inputs. Make sure to check adding 2 as that is what it will be used for.
6. 16 Bit Register
    - Verify that the register can be written to and that the value can be later read after some cycles.
7. Instruction Memory Block
    - Verify that instructions can be put in memory through an initialization file. Also test that when getting these instructions that the correct instruction is retrieved for a given address.
8. Data Memory Block
    - Verify that initialization works using an initialization file. Verify that read works given an address. Verify that it does not write when the write control bit is low. Verify that it does write when the write control bit is high and that when read the new value is actually there.
9. Control Unit
    - Verify that for every instruction the correct control bits are set throughout the datapath.
10. Stack Register
    - Verify that each of the 6 stackOPs function correctly. Verify that the reset signal actually resets the register stack. Verify that the write happens on the falling edge of the clock while a read can happen on the rising edge.
11. Merger (3 bit and 13 bit -> 16 bit)
    - Verify that when given various 3 bit values and a 13 bit values that the result is a 16 bit value with the 3 bits from the input in the most significant bits and the 13 in the least.

System Tests

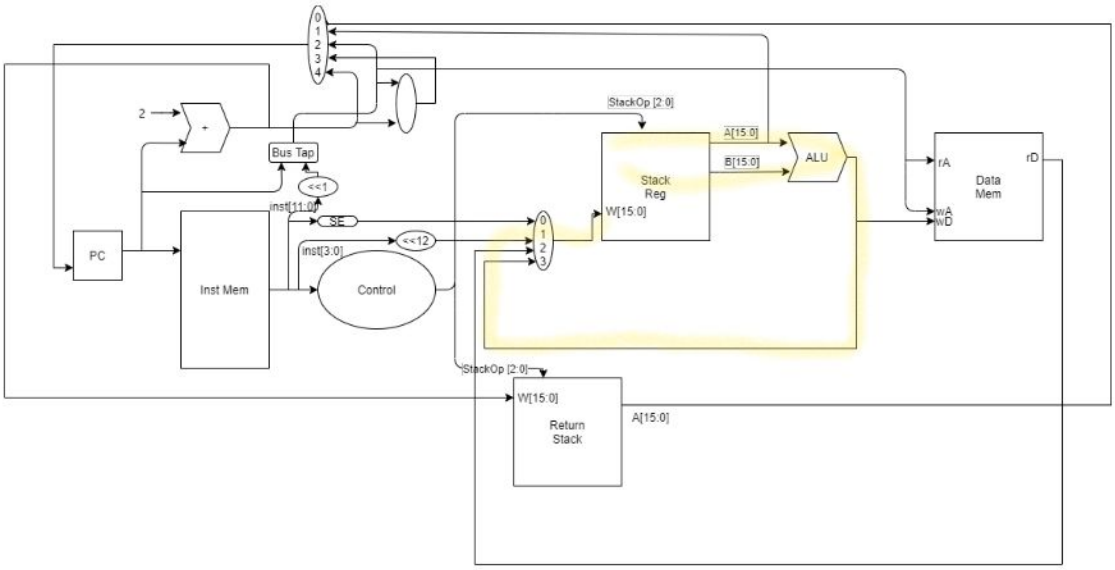
1. Use **pushi** and **lui** instructions and verify that the correct values are pushed to the stack.
2. Push 2 numbers to the stack with pushi. Then do an **add** instruction and see if the top of the stack is the correct value.
3. Push 2 numbers to the stack with pushi. Then do a **sub** instruction and see if the top of the stack is the correct value.
4. Push 2 numbers to the stack with pushi. Then do an **or** instruction and see if the top of the stack is the correct value.
5. Push 2 numbers to the stack with pushi. Then do a **slt** instruction and see if the top of the stack is the correct value.
6. Test the stack manipulation instructions **dup**, **drop**, **swap**, and **over** to see if they correctly change the stack.
7. Test **getin** and **getin2** to make sure the correct input value is put on top of the stack.
8. Do a **halt** instruction and make sure that the PC no longer is updated.
9. Use **j** and **js** and verify that PC is set to the correct value.
10. Call a simple function with **jal** and make sure that **return** sets the PC to the correct return value.
11. Verify that **beq** and **bez** conditionally branch under the correct circumstances.
12. Verify that **pop** and **push** correctly interact with data memory. Make sure that if you have a push instruction directly after a pop instruction that the correct value is pushed.
13. Verify that the simple for loop as specified in the above Code Fragments section produces the correct output.
14. Verify that RelPrime produces the correct output for various inputs.

Integration Plan

1. Push/Pop

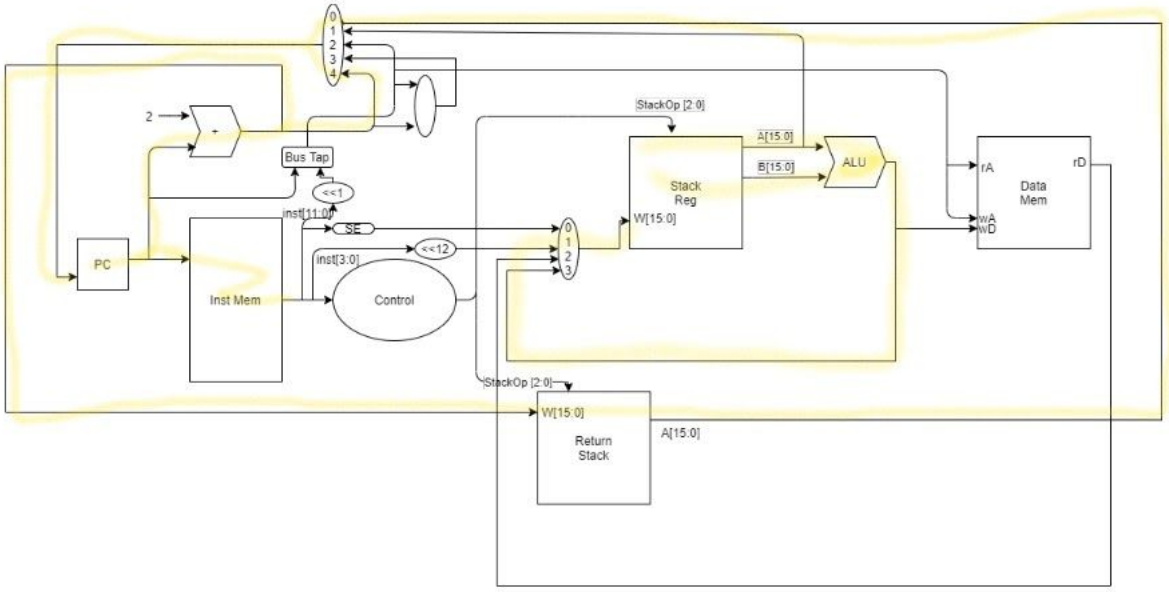
The first parts we plan to integrate are the Stack Register, ALU. Together, they should be able to push and pop on to the Stack depending on the instruction. Tests will start from initializing value in StackOps[2:0], ALUOp, and also put some values on the stack first, letting ALU perform different operations based on the control input, and then change the value of StackOps.The parts integrated in this step are highlighted in the

picture below.

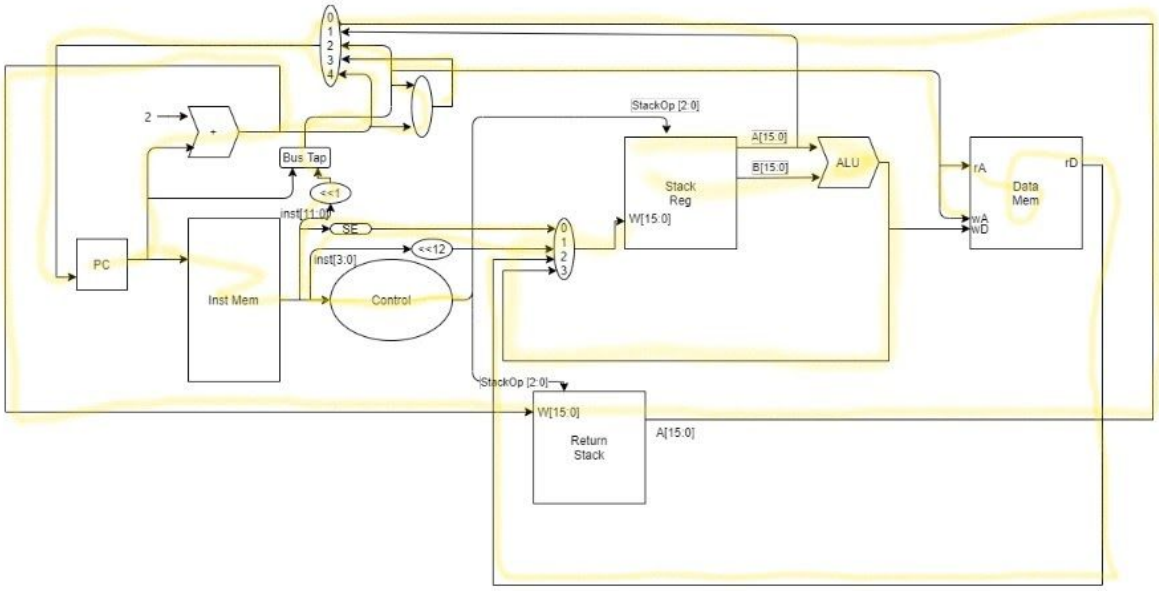


2. **Updating PC**

The next parts we plan to integrate are the PC, Adder, Instruction Memory, Return Stack, and multiplexers. Combining these parts, the value in PC should be updated the way we intend to. During the test, the value of RStackOp[1:0] and control in multiplexer will be initialized. Tests will also consist of repeating push and pop from Return Stack and monitoring the value in PC. The diagram below highlights the parts integrated so far.



3. **Adding Control, Data Memory, Bus Tap, Sign-Extender, Left-Shifter.**
   The last parts we will integrate are Control, Data Memory, Bus Tap, Sign-Extender, Left-Shifter. Since we already have the PC part and Stack Part working individually, the test will consist of the select bit in PCControl that will output the value of BusTap or LeftShifter. After testing these two parts, the test will move on to selecting bit in stackControl that will output the result of Sign-Extender and the other Left-Shifter. Last but not least, the Data Memory will be tested by checking whether the rD will write right on the stackReg and if we can write data correctly by inputting rA, wA and wD



Performance

Running relprime(0x13B0)

- number of instructions in relprime - 37
- total number of bytes to store relprime - 54
- number of instructions executed - 122357
- number of cycles - 122357
- cpi - 1
- cycle time - 17.44ns or 57.3Mhz
- execution time - 2.134ms

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	2028	4656	43%
Number of Slice Flip Flops	2120	9312	22%
Number of 4 input LUTs	3569	9312	38%
Number of bonded IOBs	98	232	42%
Number of BRAMs	8	20	40%
Number of GCLKs	2	24	8%

Assembler and Simulator

We created an assembler and simulator written in Chez Scheme to make development in our instruction set architecture easier and quicker. The assembler is given a file containing a program and it gives back the machine code of that program. This can be output either as binary or hexadecimal. Additionally, the simulator can be used to test before even converting into machine code. This simulator takes in a program just like the assembler, and it runs the code representing the stack as a list in Scheme. It is able to output statistics such as maximum stack size and number of instructions executed. Additionally, this simulator can show the stack during and after each instruction. This simulator is most useful in finding out if your code actually works. If it is found that it does not, it can then be walked through in Scheme which is a much faster debugging process than Xilinx.

Appendices

Stack Based Architecture Design Document

Team 2v(Jinhao Sheng, Luke McNeil, Austin Swatek)

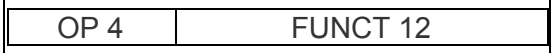
Table of Contents

Executive Summary

The general purpose processor we are making is stack-based and will make use of two stacks of registers: one as a stack of the instructions used for the operations in a function, and the other for keeping track of the return values of functions. This processor has two instruction formats: O-type and A-type. O-type instructions have a 4-bit opcode, 12-bit function, and are for instructions that require no arguments. O-types that require jumps use direct addressing, while A-types that require jumps use pseudo-direct addressing. A-type instructions have a 4-bit opcode, 12-bit immediate value or address, and are necessary for functions that require arguments. Since it is a stack-based processor, functions are called on a last-in-first-out (LIFO) basis, with the caller function pushing the arguments that the callee expects to the top of the stack. The callee pops the arguments off of the stack, and before its return call, puts the required return values back on the stack. To return, the callee pops its return address off the return stack and jumps to that address.

*****Important Note***:** Both the main stack and the return stack are 64 registers big. If one tries to push to the stack when it is full the data that is in the bottom position of the stack will be lost.

Instruction Formats

Format Type	Size	Structure	Description
O	2 bytes		OP - basic operation of the instruction FUNCT - sets the variant of the operation This format type is used for instructions that take no arguments.
A	2 bytes		OP - basic operation of the instruction IMM/ADDR - a 12 bit constant or address This format type is for any instruction that takes one argument which is either an immediate or address.

Instructions

Name	Type	Argument	Description	OP	Funct
add	O		Pop the top two values off the stack, add them, and put the result on the stack.	0x0	0x000
beq	A	label	Pop the top two values off the stack. If they are equal, then branch to label.	0x1	
bez	A	label	Pop the top value off the stack. If it is zero then branch to label.	0x2	
dup	O		Push onto the stack a duplicate of the value currently on top of the stack.	0x0	0x001
drop	O		Pop the top value off the stack, throwing it away.	0x0	0x002
halt	O		Jump back to this instruction. This is a good way to stop the program.	0x0	0x003
getin	O		Read a 16 bit number from input and push it to the stack.	0x0	0x004
j	A	target	Jump to target.	0x3	
jal	A	target	Jump to target, and push onto the return address stack the address of the next instruction.	0x4	
js	O		Pop the top value off the stack and jump to that address.	0x0	0x005
lui	A	immediate	Shift the immediate left by 12 bits and then push it into the stack	0x8	
over	O		Push onto the stack the value of the second element on the stack.	0x0	0x006
or	O		Pop the top two values off the stack, or them, and put the result on the stack.	0x0	0x007
pop	A	address	Pop the top value off the stack and put it in memory at address.	0x5	
push	A	address	Push the value at the specified address onto the stack.	0x6	
pushi	A	immediate	Push onto the stack sign extended immediate.	0x7	
return	O		Pop the top element off the return address stack, and jump there.	0x0	0x008
slt	O		Pop the top two elements off the stack, and push a 1 to the stack if the second from the top element is less than the top element. Otherwise, push a 0.	0x0	0x009
sub	O		Pop the top two values off the stack, subtract them, and put the result on the stack.	0x0	0x00A
swap	O		Swap the top two elements on the stack.	0x0	0x00B
getin2	O		Read a 16 bit number from input2 and push it to the stack.	0x0	0x00C

Addressing Modes

Instructions	Format Type	Addressing Modes
js, return	O	Direct
j, jal, beq, bez	A	Pseudo-Direct

Pseudo-Direct Example

- Going from 16 bit address to the 12 bits in the instruction
    a. Shift the 16 bit number right 1
    b. Chop off the 4 most significant bits
    c. Use this 12 bit number in the instruction ADDR field.
- Going from 12 bits in the instruction to a 16 bit address
    a. Shift the 12 bits to the left 1
    b. Put on the front of these 13 bits the 3 most significant bits from $PC.
    c. Use this 16 bit number as the address to go to.

Direct Example

- Here the jump is looking at the value in a 16 bit register. The address to jump to is simply those 16 bits.

Procedure Calling Conventions

To prepare to call a function the caller must put all arguments that the callee expects to receive on top of the stack. Then the caller must call jal to go to the callee. This command will push the return address to the return address stack. The callee's responsibilities are to pop the arguments off the stack and leave on the stack any return values. The callee will then do the return instruction which will pop the top element off the return address stack before going back to the correct spot.

Included below in the code fragments section is an example of nested function calling.

RelPrime

RelPrime and Sample Procedure Call					
ADDR	**MC**	**LABEL**	**ASM**	**STACK**	**RETURN STACK**
0x0	0x0004	MAIN:	getin	() -> (n)	()
0x2	0x4003		jal RELPRIME	(n) -> (relprime(n))	() -> (0x4)
0x4	0x0003		halt	(relprime(n))	()
0x6	0x7002	RELPRIME:	pushi 2	(n) -> (n, m)	(0x4)
0x8	0x0006	RPLOOP:	over	(n, m) -> (n, m, n)	(0x4)
0xA	0x0006		over	(n, m, n) -> (n, m, n, m)	(0x4)
0xC	0x400F		jal GCD	(n, m, n, m) -> (n, m, gcd)	(0x4) -> (0x4, 0x8)
0xE	0x7001		pushi 1	(n, m, gcd) -> (n, m, gcd, 1)	(0x4)
0x10	0x100C		beq RETURNM	(n, m, gcd, 1) -> (n, m)	(0x4)
0x12	0x7001		pushi 1	(n, m) -> (n, m, 1)	(0x4)
0x14	0x0000		add	(n m 1) -> (n, m+1)	(0x4)
0x16	0x3004		j RPLOOP	(n, m+1)	(0x4)
0x18	0x000B	RETURNM:	swap	(n, m) -> (m, n)	(0x4)
0x1A	0x0002		drop	(m, n) -> (m)	(0x4)
0x1C	0x0008		return	(m)	(0x4) -> ()
0x1E	0x0006	GCD:	over	(n, m, a, b) -> (n, m, a, b, a)	(0x4, 0x8)
0x20	0x2020		bez RETURNB	(n, m, a, b, a) -> (n, m, a, b)	(0x4 0x8)
0x22	0x0001	LOOP:	dup	(n, m, a, b) -> (n, m, a, b, b)	(0x4 0x8)
0x24	0x2023		bez RETURNA	(n, m, a, b, b) -> (n, m, a, b)	(0x4 0x8)
0x26	0x0006		over	(n, m, a, b) -> (n, m, a, b, a)	(0x4 0x8)
0x28	0x0006		over	(n, m, a, b, a) -> (n, m, a, b, a, b)	(0x4 0x8)
0x2A	0x000B		swap	(n, m, a, b, a, b) -> (n, m, a, b, b, a)	(0x4 0x8)
0x2C	0x0009		slt	(n, m, a, b, b, a) -> (n, m, a, b, b<a)	(0x4 0x8)
0x2E	0x201D		bez ELSE	(n, m, a, b, b<a) -> (n, m, a, b)	(0x4 0x8)
0x30	0x000B		swap	(n, m, a, b) -> (n, m, b, a)	(0x4 0x8)
0x32	0x0006		over	(n, m, b, a) -> (n, m, b, a, b)	(0x4 0x8)
0x34	0x000A		sub	(n, m, b, a, b) -> (n, m, b, a-b)	(0x4 0x8)
0x36	0x000B		swap	(n, m, b, a-b) -> (n, m, a-b, b)	(0x4 0x8)
0x38	0x3011		j LOOP	(n, m, a-b, b)	(0x4 0x8)
0x3A	0x0006	ELSE:	over	(n, m, a, b) -> (n, m, a, b a)	(0x4 0x8)
0x3C	0x000A		sub	(n, m, a, b, a) -> (n, m, a, b-a)	(0x4 0x8)
0x3E	0x3011		j LOOP	(n, m, a, b-a)	(0x4 0x8)
0x40	0x000B	RETURNB:	swap	(n, m, a, b) -> (n, m, b, a)	(0x4 0x8)
0x42	0x0002		drop	(n, m, b, a) -> (n, m, b)	(0x4 0x8)
0x44	0x0008		return	(n, m, b)	(0x4, 0x8) -> (0x4)
0x46	0x0002	RETURNA:	drop	(n, m, a, b) -> (n, m, a)	(0x4, 0x8)
0x48	0x0008		return	(n, m, a)	(0x4, 0x8) -> (0x4)

Code Fragments

Return

Int main() {return (2 + 3) - 1;}

ADDR	MC	LABEL	ASM	STACK	RETURN STACK
0x0	0x7002	MAIN:	pushi 2	() -> (2)	()
0x2	0x7003		pushi 3	(2) -> (2, 3)	()
0x4	0x0000		add	(2, 3) -> (5)	()
0x6	0x7001		pushi 1	(5) -> (5, 1)	()
0x8	0x000A		sub	(5, 1) -> (4)	()
0xA	0x0003		halt	(4)	()

Loading an 16 bit address onto the stack. (load 1000 0000 0000 0001)

ADDR	MC	LABEL	ASM	STACK	RETURN STACK
0x0	0x8008	MAIN:	lui 0x8	() ->（0x8000）	()
0x2	0x7001		pushi 1	(0x8000) -> (0x8000, 0x1)	()
0x4	0x0007		or	(0x8000, 0x1) -> (0x8001)	()
0x6	0x0003		halt		()

For Loop

int main(){
int x = 1
for(int i = 0; i < 5; i++){ x++;}

ADDR	MC	LABEL	ASM	STACK	RETURN STACK
0x0	0x7001	MAIN:	pushi 1	() -> (x)	()
0x2	0x7000		pushi 0	(x)-> (x i)	()
0x4	0x0001	LOOP:	dup	(x i) -> (x i i)	()
0x6	0x7005		pushi 5	(x i i) -> (x i i 5)	()
0x8	0x0009		slt	(x i i 5) -> (x i i<5)	()
0xA	0x7001		pushi 1	(x i i<5) -> (x i i<5 1)	()
0xC	0x1009		beq OP	(x i i<5 1) -> (x i)	()
0xE	0x0002		drop	(x)	()
0x10	0x0003		halt	(x)	()
0x12	0x7001	OP:	pushi 1	(x i) -> (x i 1)	()
0x14	0x0000		add	(x i 1) -> (x i++)	()
0x16	0x000A		swap	(x i++) -> (i++ x)	()
0x18	0x7001		pushi 1	(i++ x) -> (i++ x 1)	()
0x1A	0x0000		add	(i++ x 1) -> (i++ x++)	()
0x1C	0x000B		swap	(i++ x++) -> (x++ i++)	()
0x1E	0x3002		j LOOP	(x++ i++)	()

Return Chain

int main() {return f1(2);}

int f1(int a) {return f2(a);}
int f2(int b) {return f3(b);}
int f3(int c) {return c+1;}

ADDR	MC	LABEL	ASM	STACK	RETURN STACK
0x0	0x7002	MAIN:	pushi 2	() -> (2)	()
0x2	0x4003		jal F1	(2)	() -> (0x4)
0x4	0x0003		halt	(3)	()
0x6	0x4005	F1:	jal F2	(2)	(0x4) -> (0x4, 0x8)
0x8	0x0008		return	(3)	(0x4) -> ()
0xA	0x4007	F2:	jal F3	(2)	(0x4, 0x8) -> (0x4, 0x8, 0xC)
0xC	0x0008		return	(3)	(0x4, 0x8) -> (0x4)
0xE	0x7001	F3:	pushi 1	(2) -> (2, 1)	(0x4, 0x8, 0xC)
0x10	0x0000		add	(2, 1) -> (3)	(0x4, 0x8, 0xC)
0x12	0x0008		return	(3)	(0x4, 0x8, 0xC) -> (0x4, 0x8)

Reading Data From the Input Port

ADDR	MC	LABEL	ASM	STACK	RETURN STACK
0x0	0x0004	MAIN:	getin	() -> （input）	()
0x2	0x0003		halt	(input)	()

RTL

Notes:

- stack is 64 registers
- Rstack is the return address stack and is 64 registers

add, sub, or	**beq**	**bez**

inst = Mem[PC]
stack[0] = stack[1] OP stack[0]
stack[1] = stack[2]
stack[2] = stack[3]
...
stack[63] = 0
PC = PC + 2
	inst = Mem[PC]
	A = stack[0]
	B = stack[1]
	stack[0] = stack[2]
	stack[1] = stack[3]
	...
	stack[62] = 0
	stack[63] = 0
	if (A-B==0):
	  PC = (PC[15:13])...(inst[11:0] << 1)
	else: PC = PC + 2
		inst = Mem[PC]
		A = stack[0]
		stack[0] = stack[1]
		stack[1] = stack[2]
		...
		stack[63] = 0
		if (A==0):
		  PC = (PC[15:13])...(inst[11:0] << 1)
		else: PC = PC + 2

dup	**drop**	**halt**
inst = Mem[PC] stack[63] = stack[62] stack[62] = stack[61] ... stack[1] = stack[0] PC = PC + 2	inst = Mem[PC] stack[0] = stack[1] stack[1] = stack[2] ... stack[63] = 0 PC = PC + 2	inst = Mem[PC]
getin or getin2	**j**	**jal**
inst = Mem[PC] stack[63] = stack[62] stack[62] = stack[61] ... stack[1] = stack[0] stack[0] = INPUT or INPUT2 PC = PC + 2	inst = Mem[PC] PC = (PC[15:13])...(inst[11:0] << 1)	inst = Mem[PC] Rstack[63] = Rstack[62] Rstack[62] = Rstack[61] ... Rstack[1] = Rstack[0] Rstack[0] = PC + 2 PC = (PC[15:13])...(inst[11:0] << 1)
js	**lui**	**over**
inst = Mem[PC] A = stack[0] stack[0] = stack[1] stack[1] = stack[2] ... stack[63] = 0 PC = A	inst = Mem[PC] stack[63] = stack[62] stack[62] = stack[61] ... stack[1] = stack[0] stack[0] = inst[3:0] << 12 PC = PC + 2	inst = Mem[PC] stack[63] = stack[62] ... stack[2] = stack[1] stack[1] = stack[0] stack[0] = stack[2] PC = PC + 2

pop	**push**	**pushi**
inst = Mem[PC] A = stack[0] stack[0] = stack[1] stack[1] = stack[2] ... stack[63] = 0 Mem[PC[15:13]...(inst[11:0] << 1)] = A PC = PC + 2	inst = Mem[PC] stack[63] = stack[62] ... stack[1] = stack[0] stack[0] = Mem[PC[15:13]...(inst[11:0] << 1)] PC = PC + 2	inst = Mem[PC] stack[63] = stack[62] ... stack[1] = stack[0] stack[0] = SE(inst[11:0]) PC = PC + 2
return	**slt**	**swap**
inst = Mem[PC] A = Rstack[0] Rstack[0] = Rstack[1] Rstack[1] = Rstack[2] ... Rstack[63] = 0 PC = A	inst = Mem[PC] A = stack[0] B = stack[1] if (B < A): stack[0] = 1 else: stack[0] = 0 stack[1] = stack[2] stack[2] = stack[3] ... stack[63] = 0 PC = PC + 2	inst = Mem[PC] A = stack[0] stack[0] = stack[1] stack[1] = A PC = PC + 2

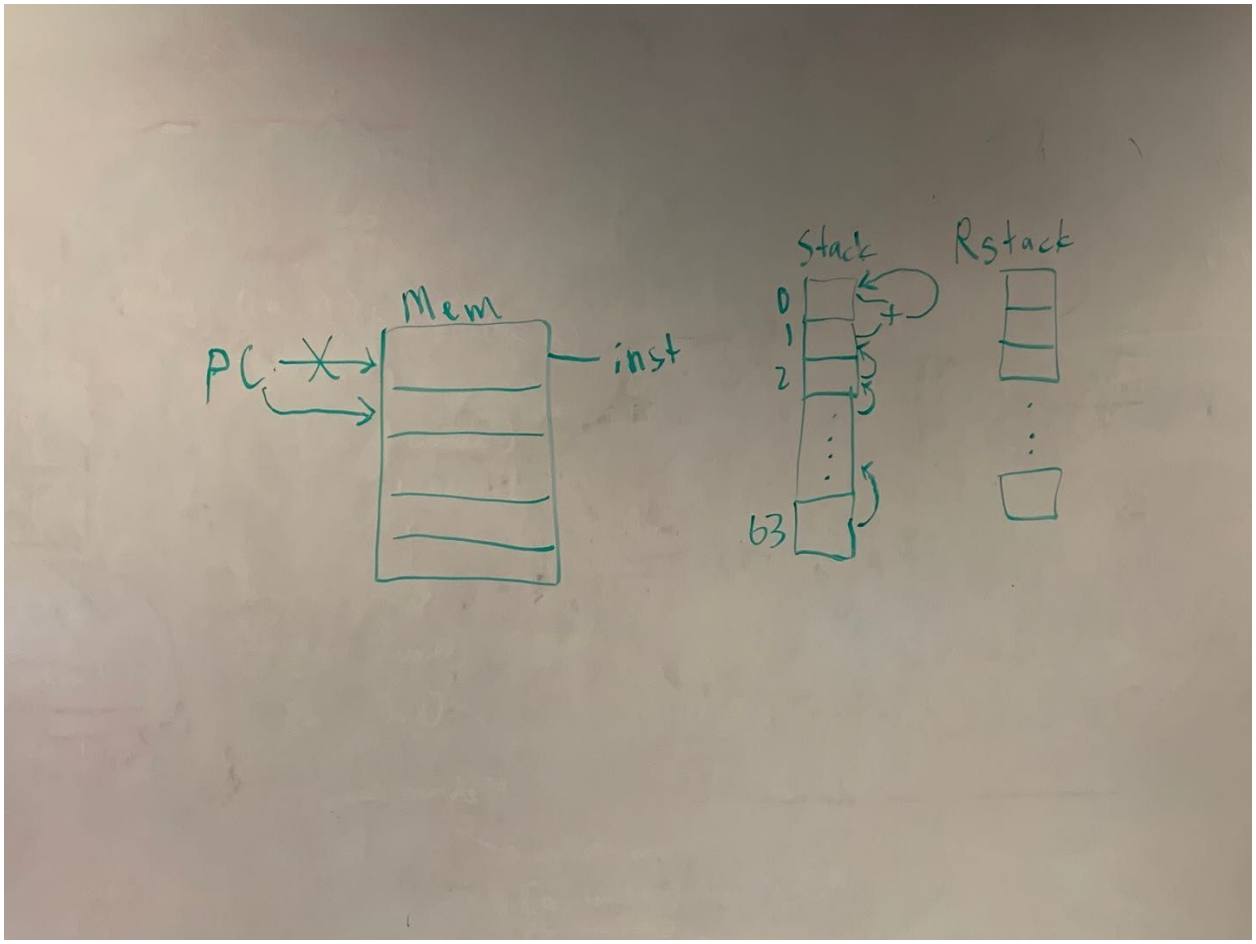
RTL Verification

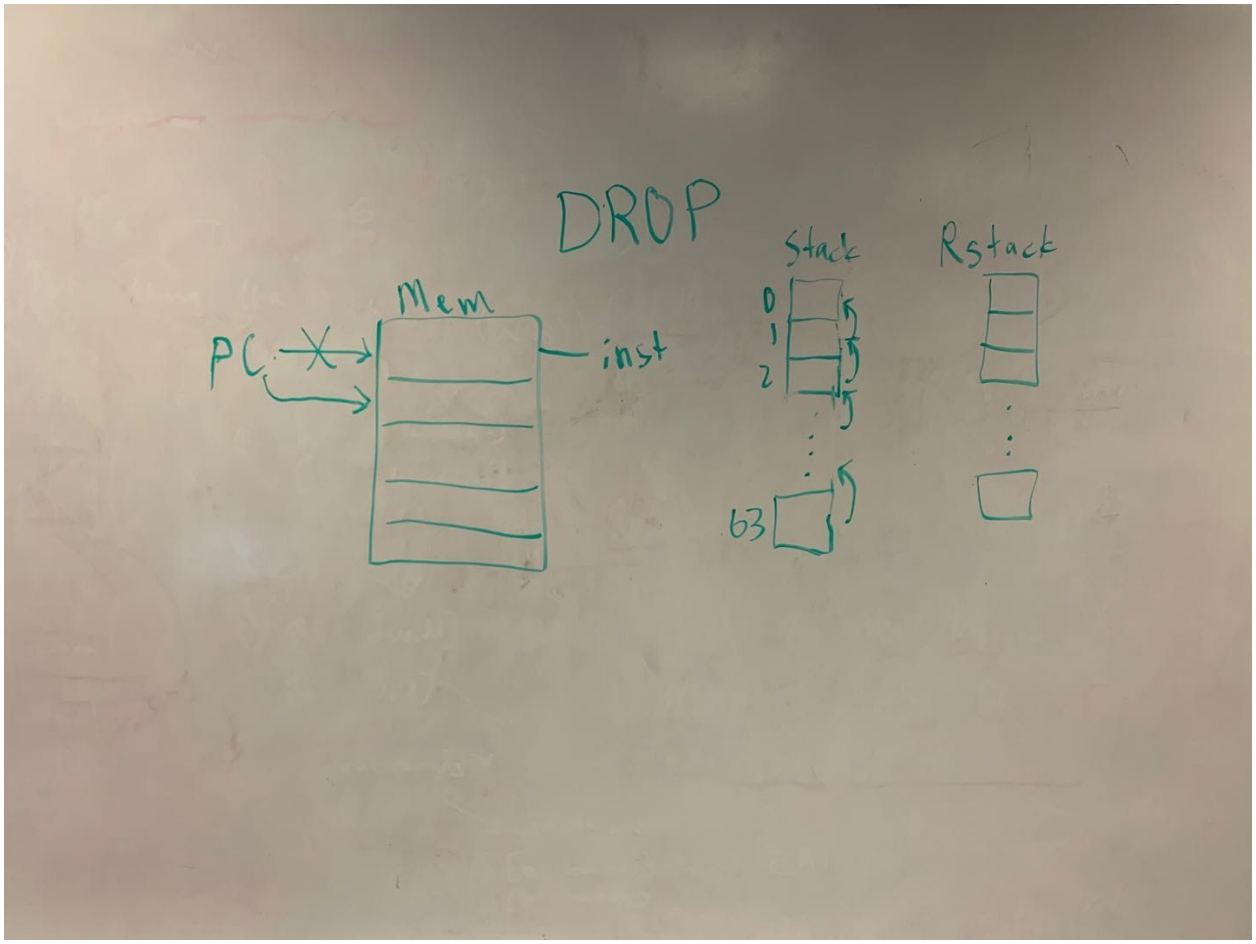
We verified our RTl systematically by going through each instruction. We drew two stacks, PC and Memory and kept track of the number and order of steps that should be taken.

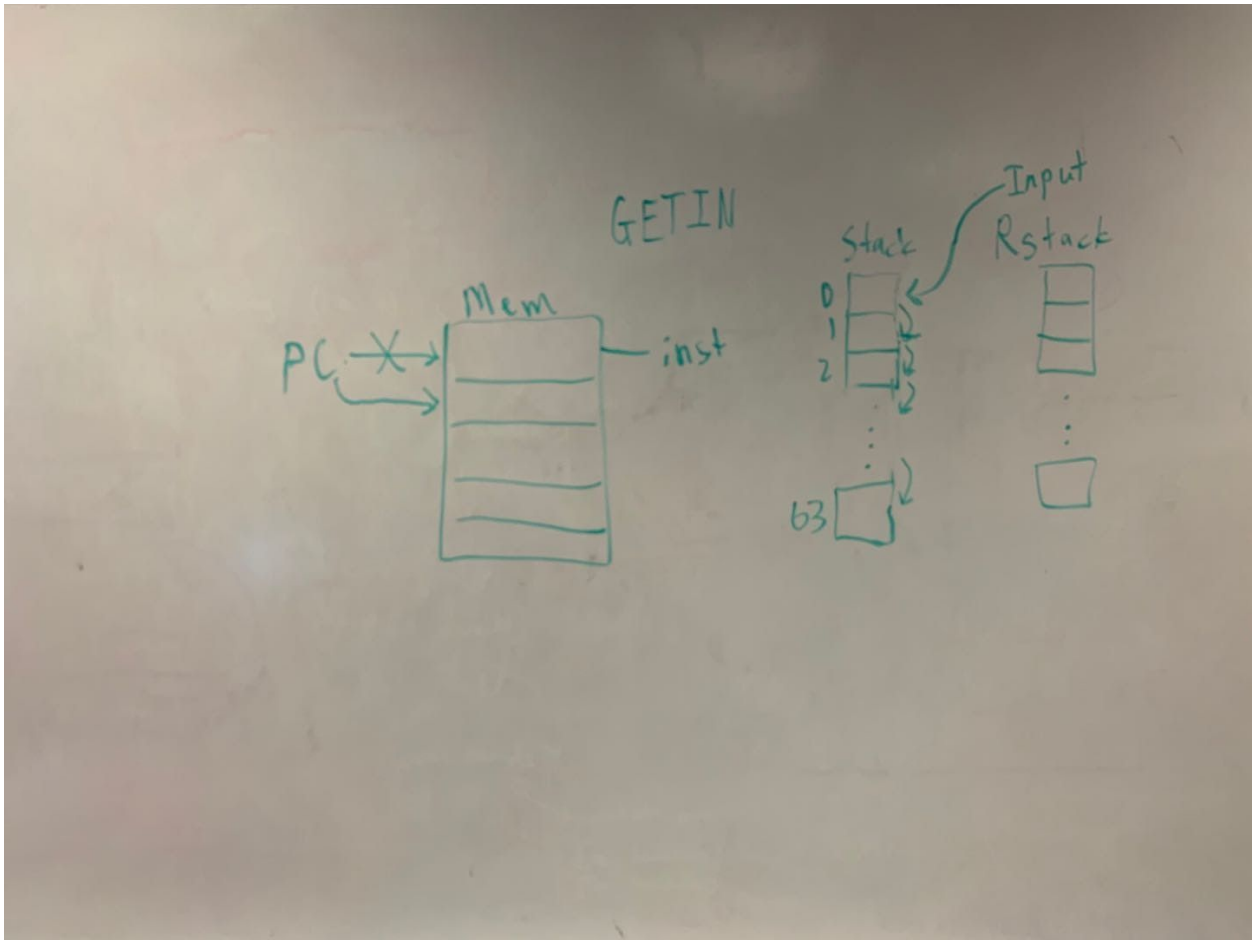
From doing this testing we caught several errors in our RTL.

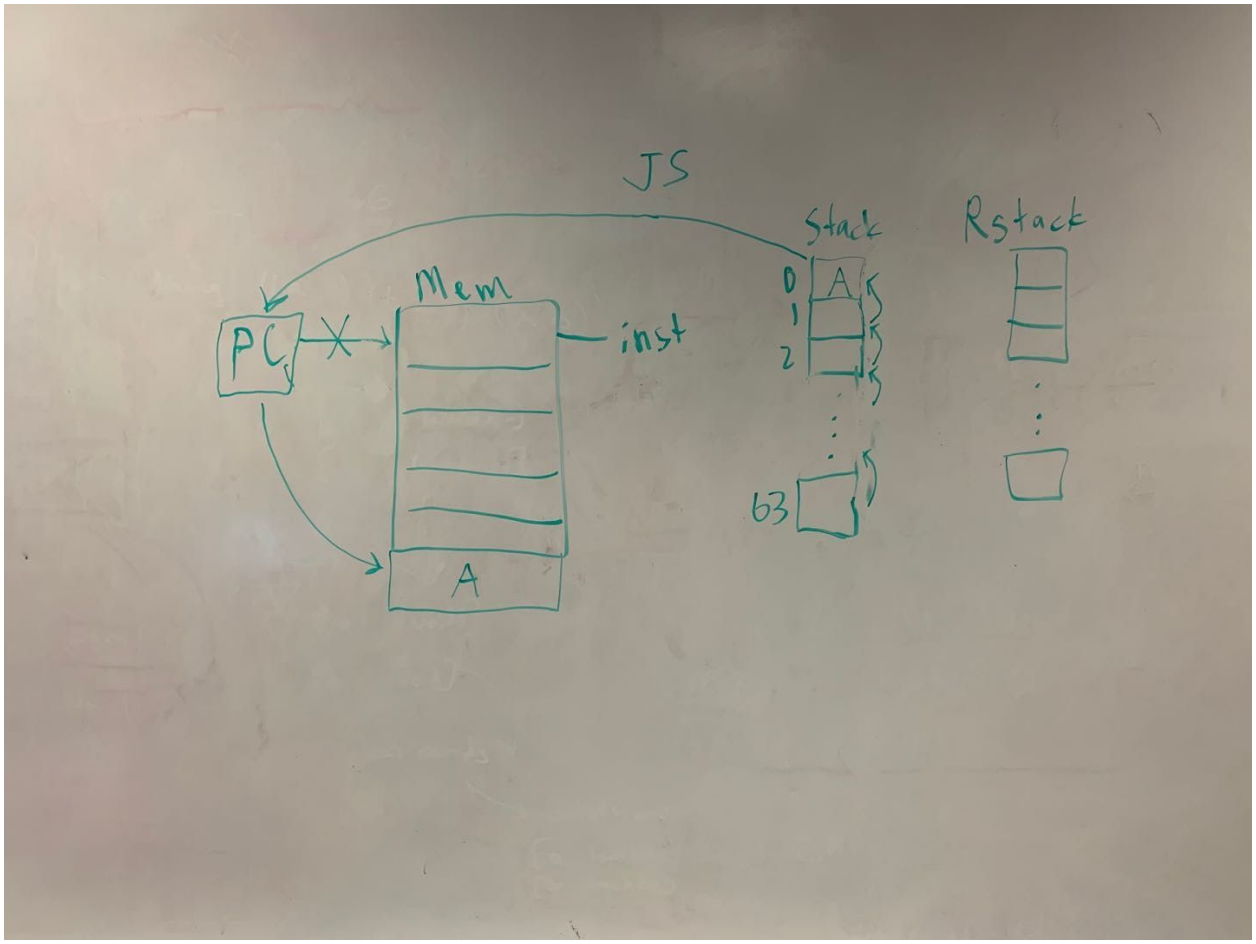
- All of the instructions were incrementing the PC by adding 4. This should be 2 since our instructions are 16 bits long.
- jal - it previously set stack[0] to PC+2 then stack[1] = stack[0] … This meant that the value of PC+2 was being put in the entire stack. This was fixed by starting at stack[63] and going down and setting stack[0] lastl.
- pushi - it previously shifted inst[11:0] left one before sign extending, we do not want this to happen since we only do the sign shifting when creating an address.

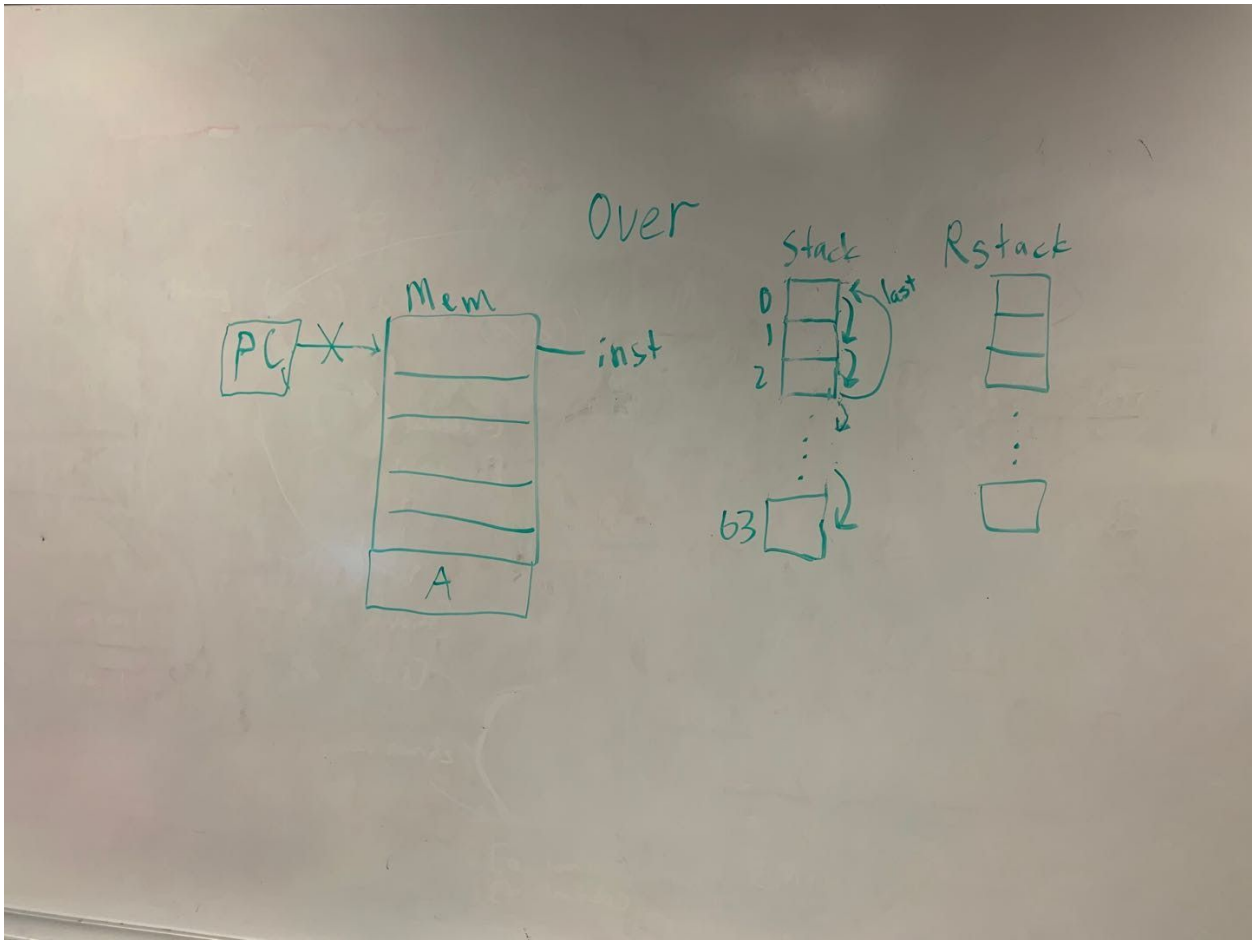
ADD

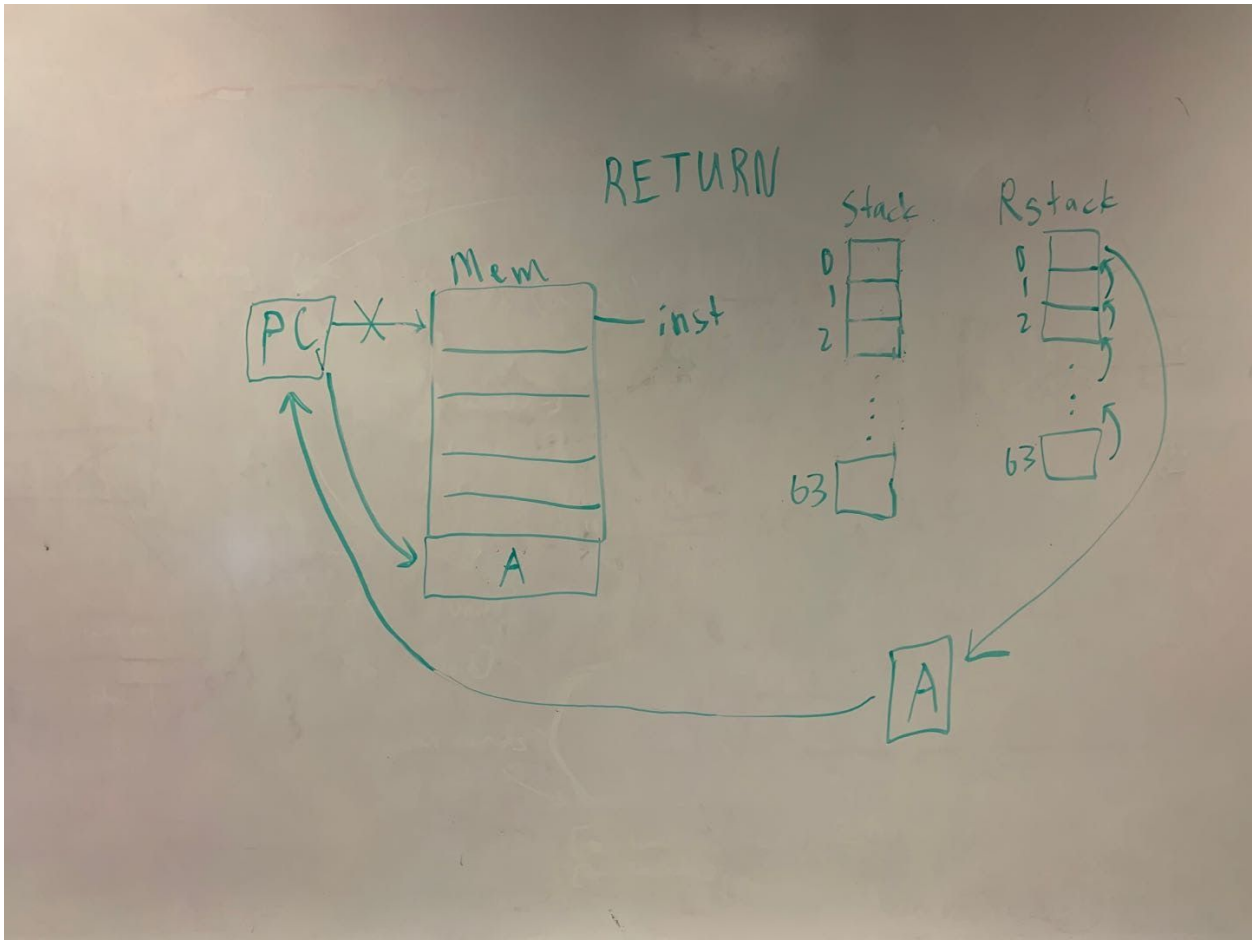


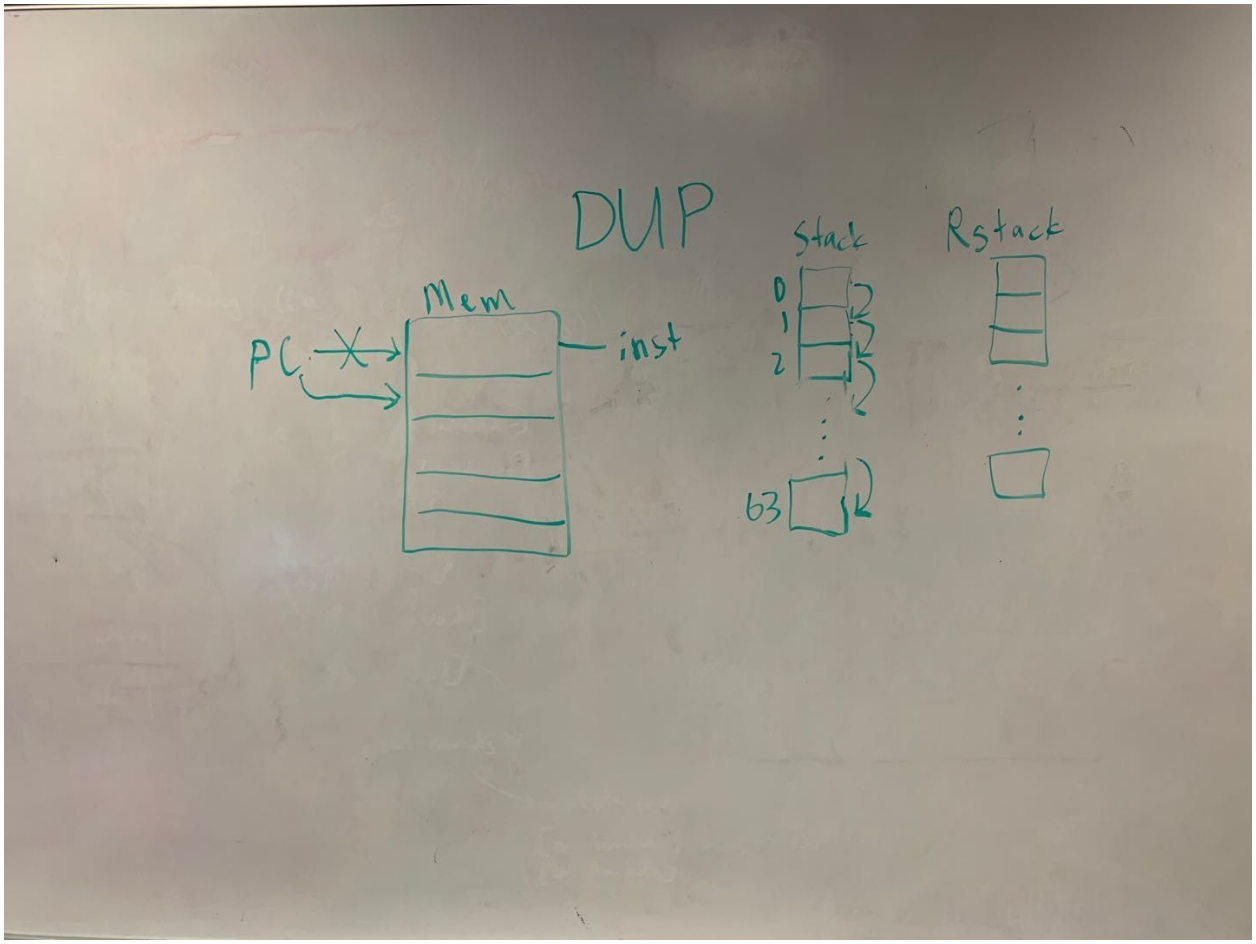

DROP
Mem
PC
inst
Stack
Rstack
0
1
2
63

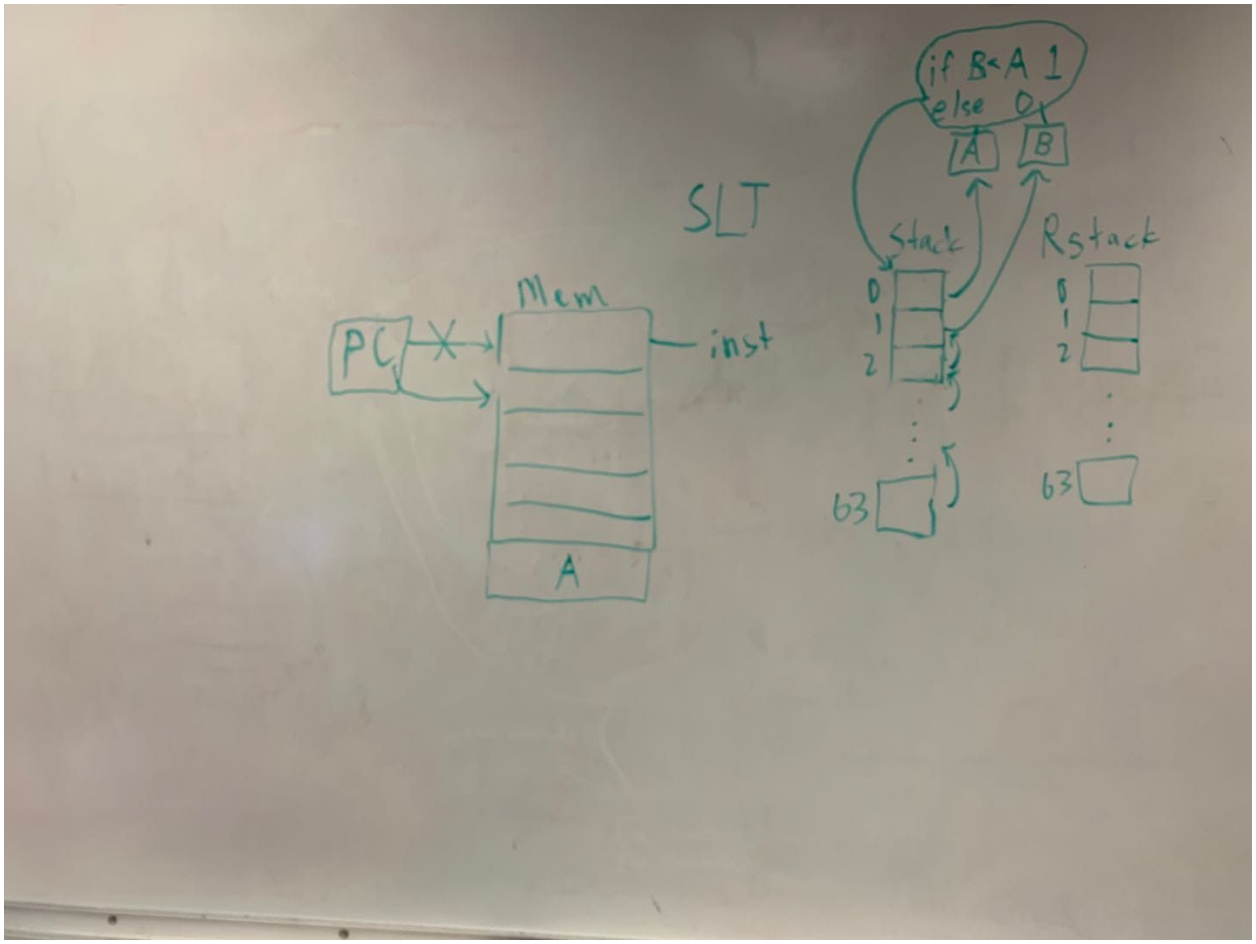

GETIN
Mem
PC
inst
Stack
Input
Rstack
0
1
2
63

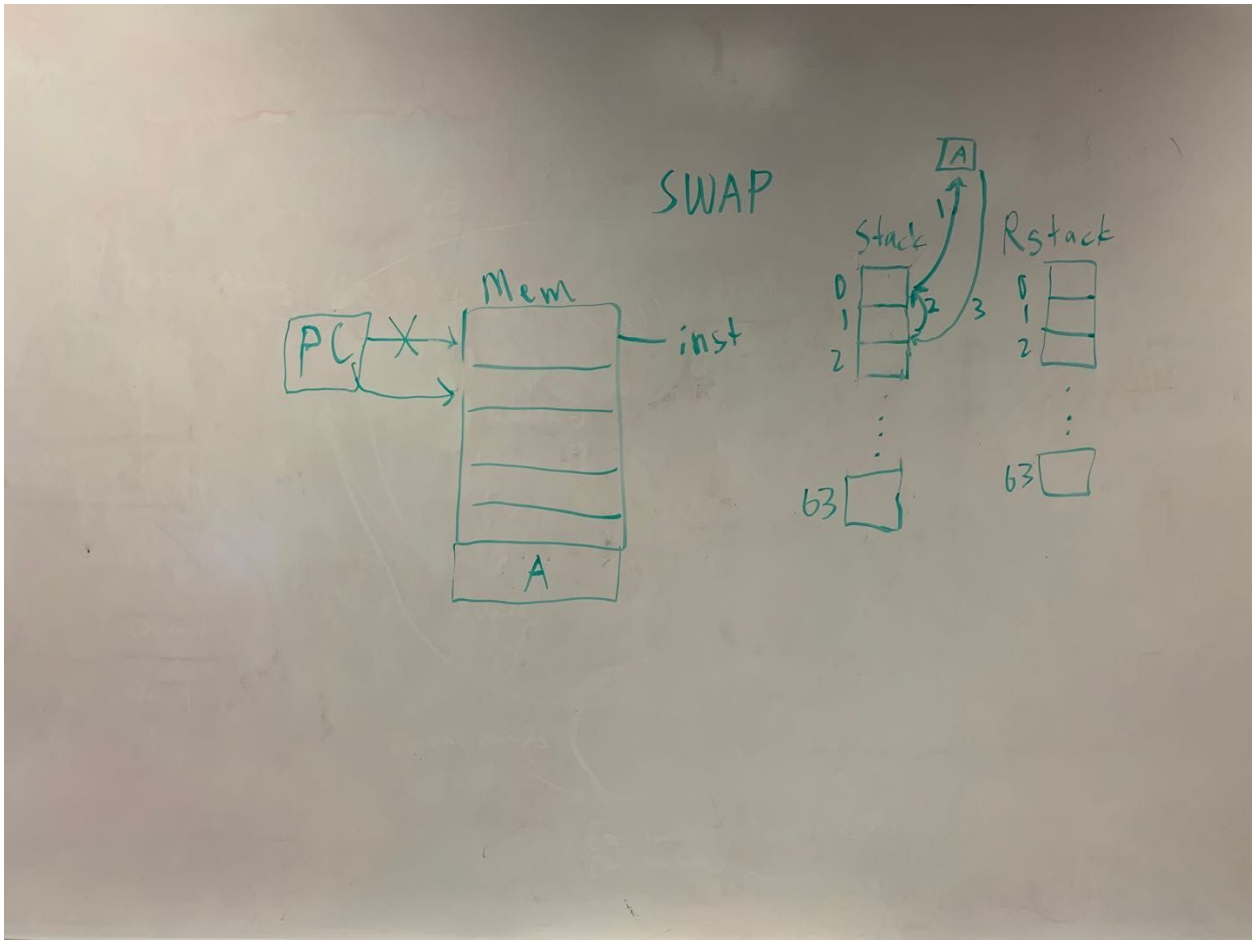

JS
Stack
Rstack
Mem
PC
inst
0
1
2
63
A
A

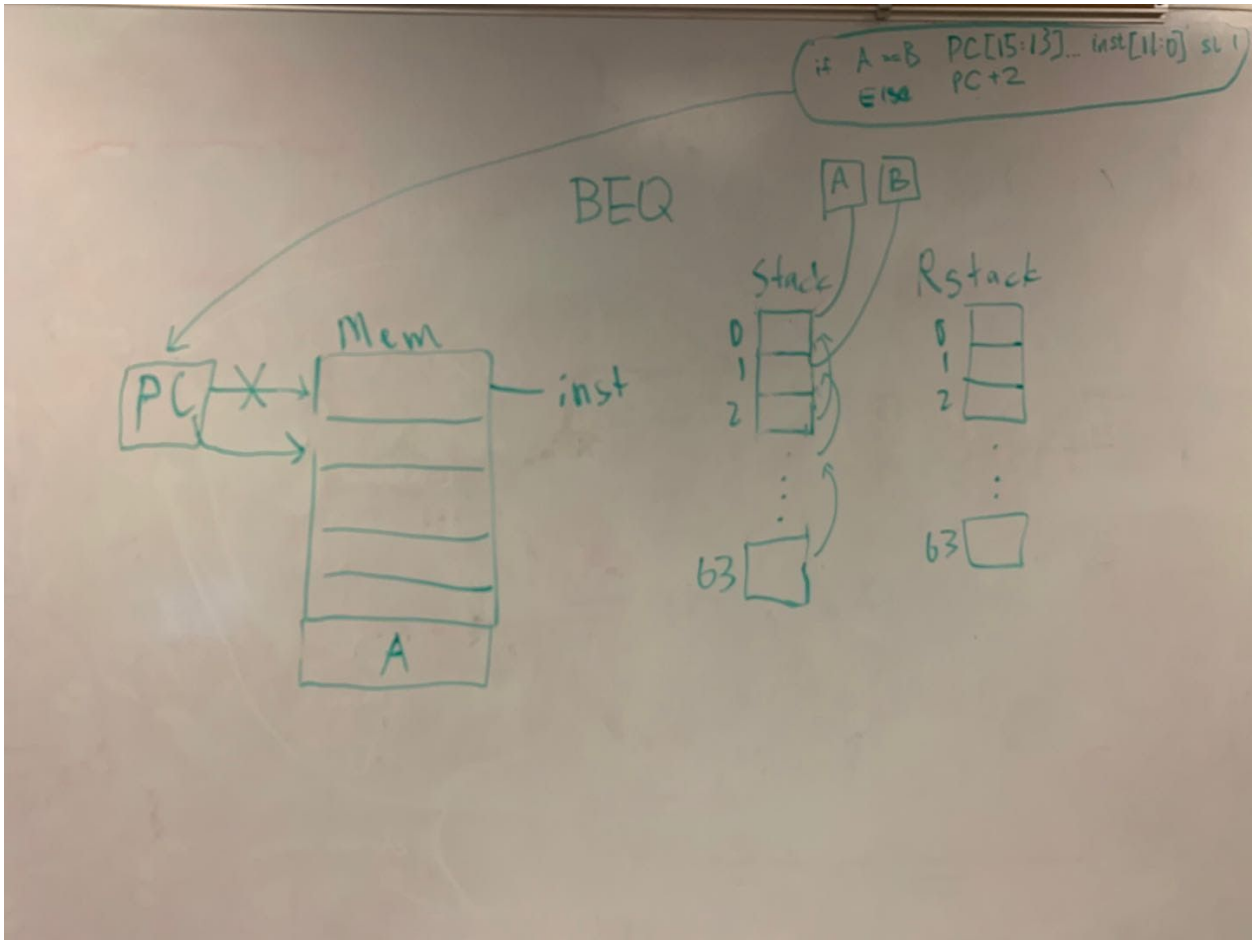

Over
Mem
PC
inst
A
Stack
Rstack
0
1
2
last
63

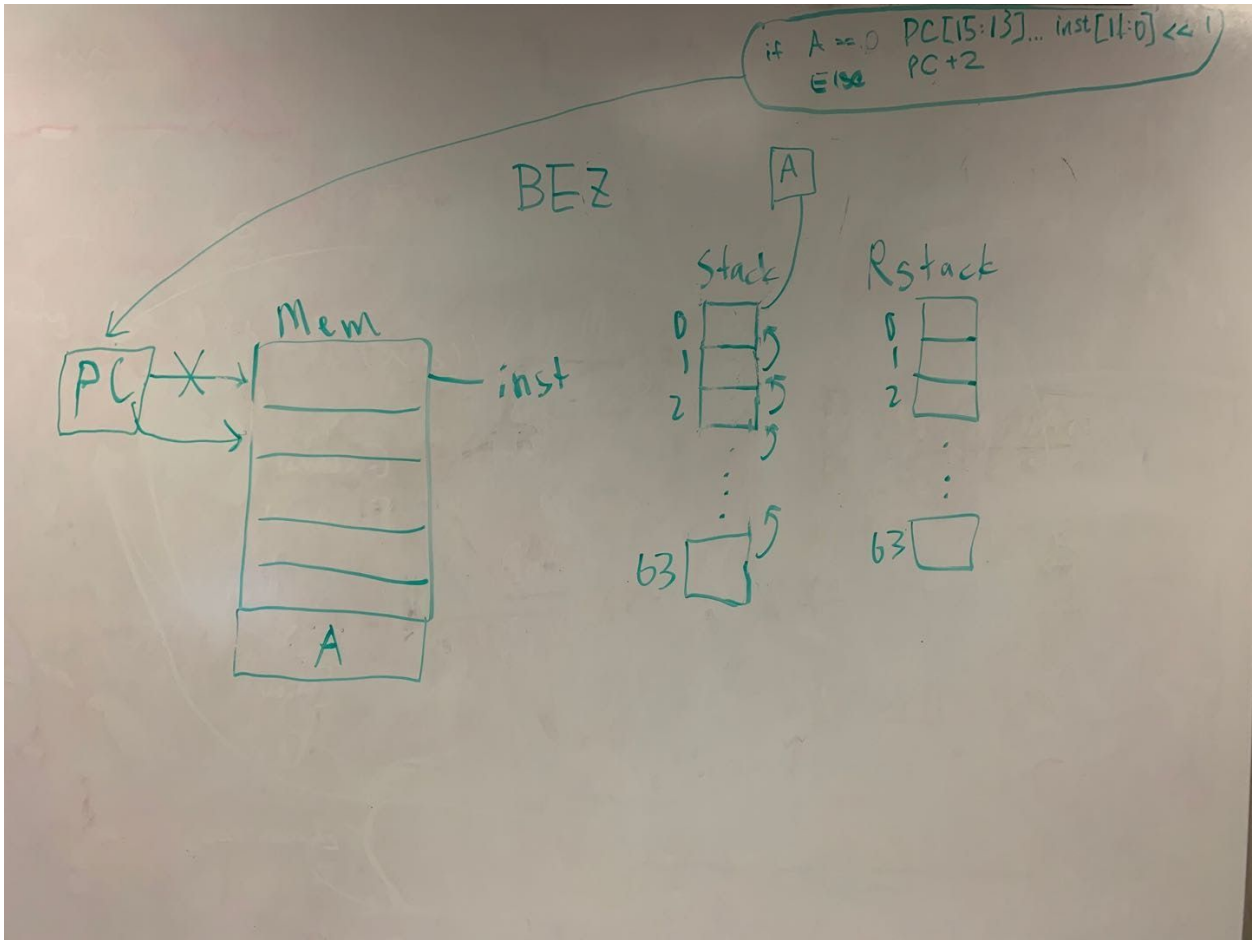

RETURN
Stack
Rstack
Mem
PC
inst
0
1
2
63
A
A

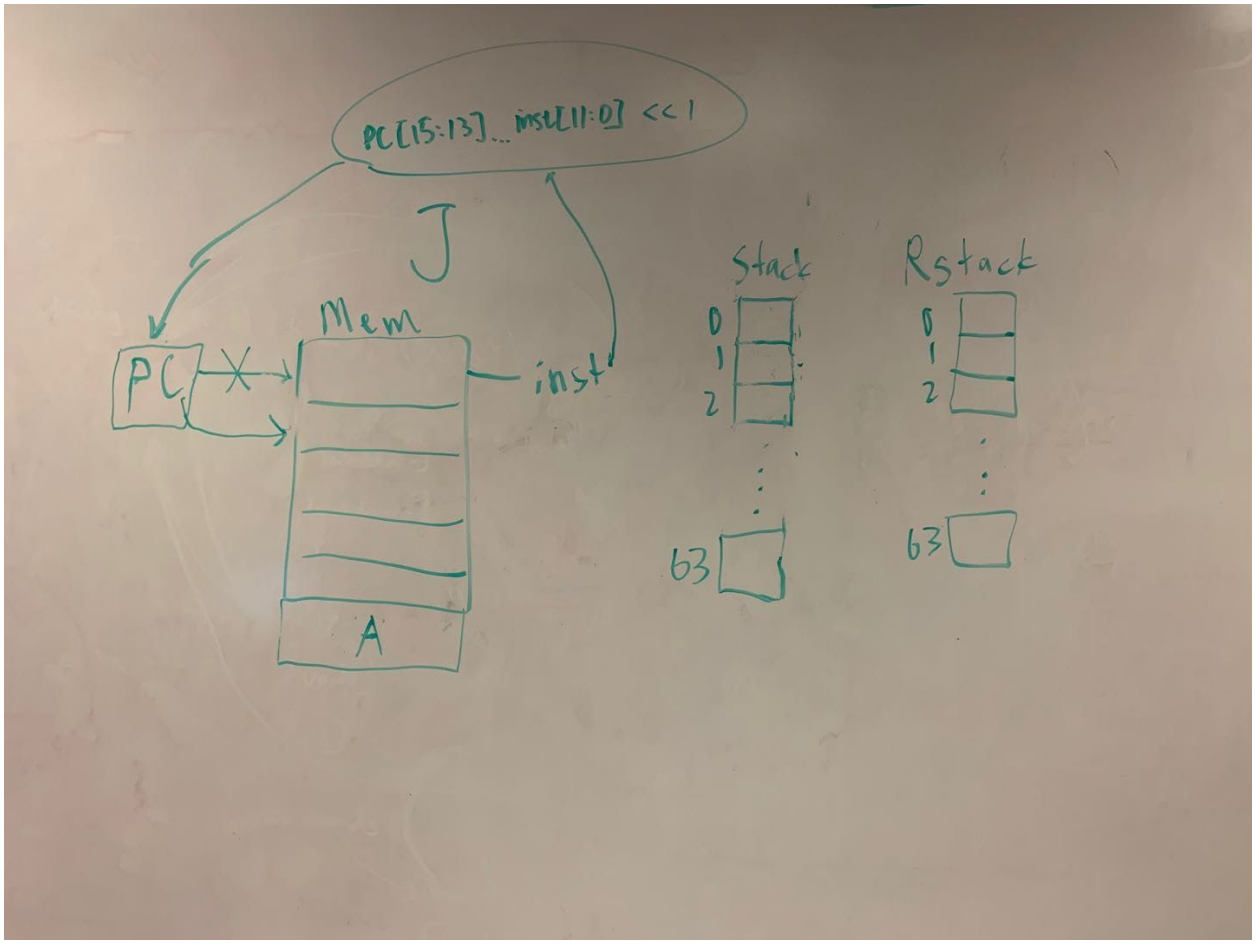

DUP
Mem
PC
inst
Stack
0
1
2
63
Rstack

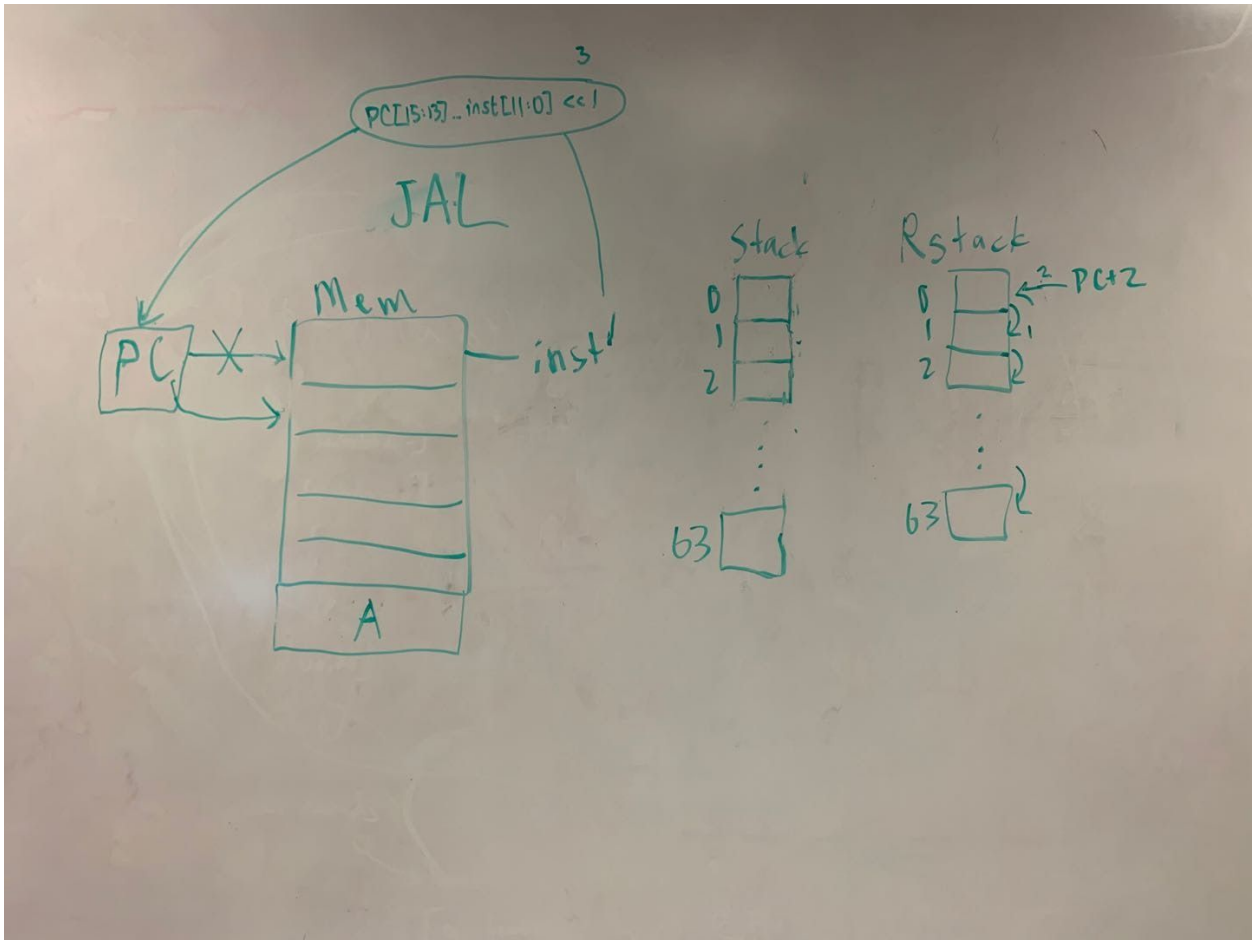

SLT
Mem
PC
inst
A
if B<A 1
else 0
A
B
Stack
Rstack
0
1
2
63
0
1
2
63

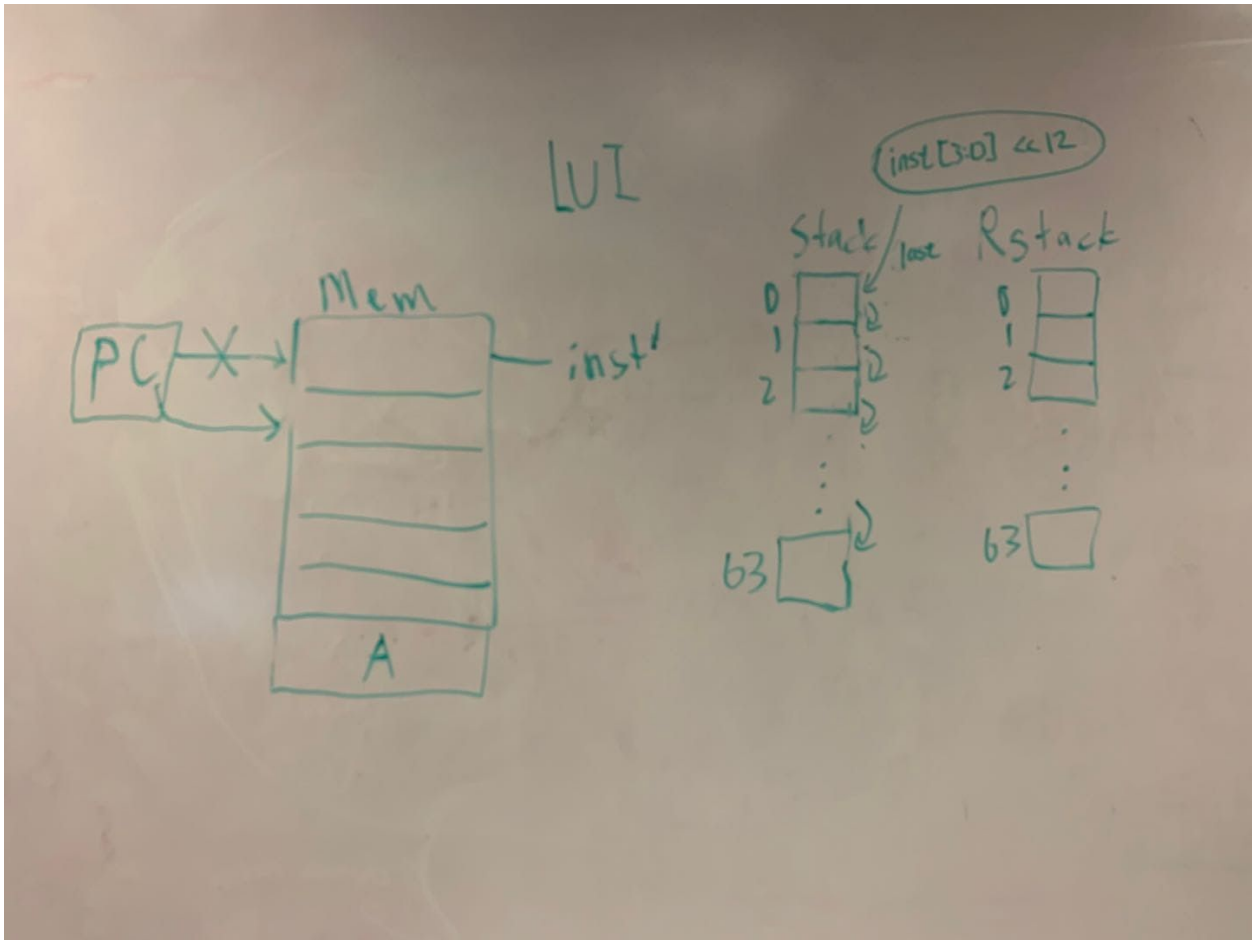

SWAP
Mem
PC
inst
A
Stack
Rstack
A
1
2
3
0
1
2
63
0
1
2
63

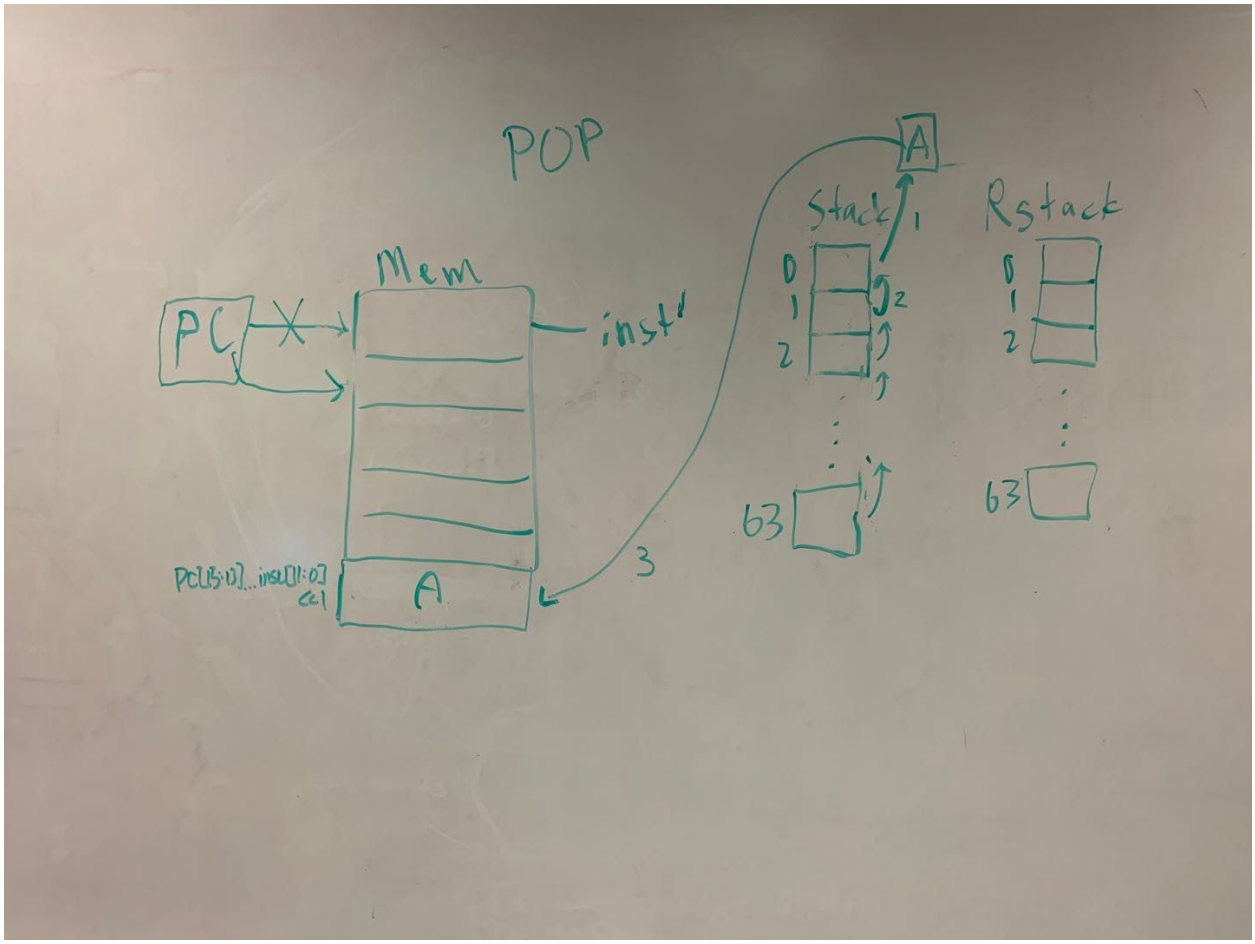

if A=B PC[15:13]... inst[11:0] sl 1
else PC+2
BEQ
A
B
Stack
Rstack
Mem
PC
inst
0
1
2
63
A

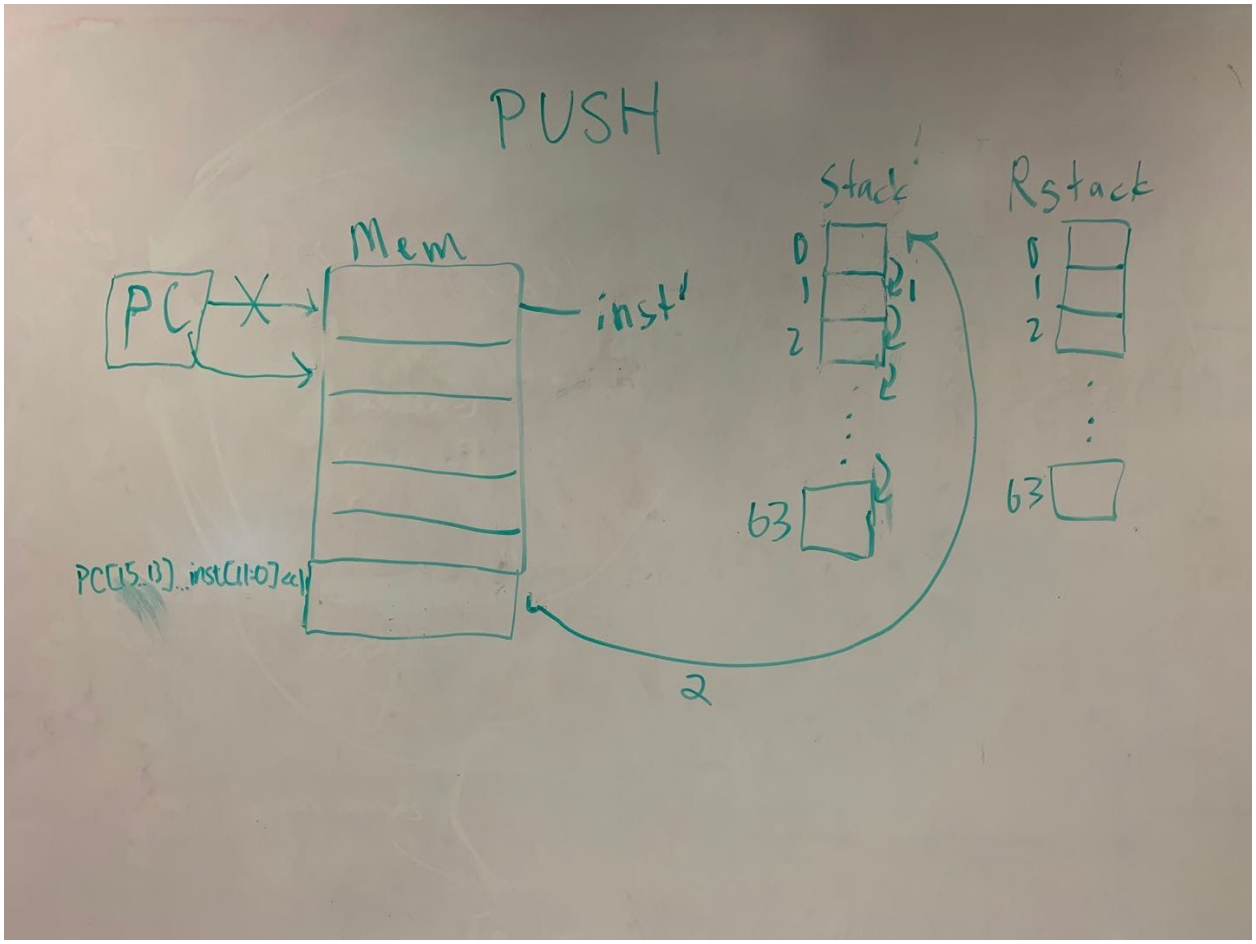

if A == 0 PC[15:13]... inst[11:0] << 1
Else PC+2
BEZ
A
Stack
Rstack
Mem
PC
inst
0
1
2
63
A


PC[15:13]... inst[11:0] << 1
J
Mem
PC
inst
A
Stack
0
1
2
63
Rstack
0
1
2
63

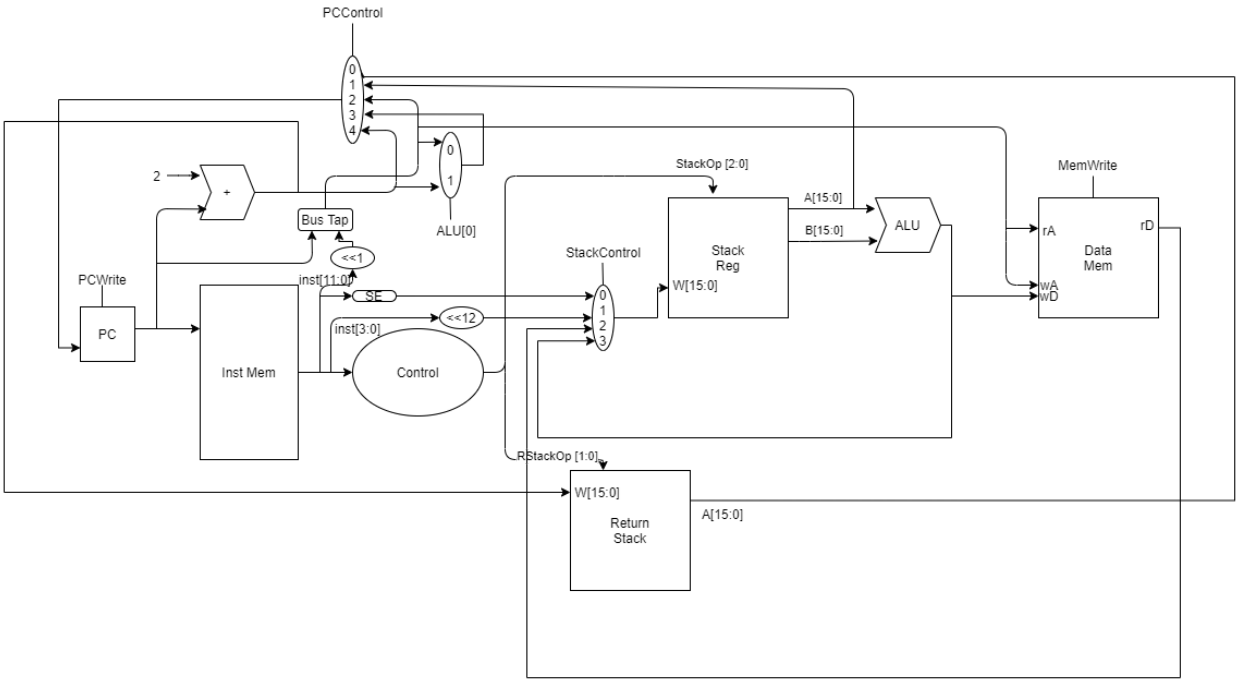

3
PC[15:15]..inst[11:0] << 1
JAL
Mem
PC
inst
A
Stack
0
1
2
63
Rstack
2
PC+2
0
1
2
63


LUI
inst[3:0] << 12
Stack
Rstack
Mem
PC
inst
A
0
1
2
63
0
1
2
63


POP
Mem
PC
inst
A
3
Stack
Rstack
A
0
1
2
63
1
2


PUSH
Mem
PC
inst
Stack
Rstack
0
1
2
63
PC[15..0] inst[11:0]
2

Datapath Design



Required Components

1. Multiplexer
    a. Inputs - at most 8 of i[15:0]
    b. Outputs - o[15:0] and goes to PC
    c. Control - 3 bits vary from 000 to 111
2. Sign Extender (12 bit -> 16 bit)
    a. Inputs - a[11:0] is the thing to be sign extended
    b. Outputs - r[15:0] is the sign extended 16 bit result
    c. Control Bits - none
    d. Description - This component sign extends a 12 bit bus into a 16 bit
    e. RTL Symbols - SE
3. Left Shifter
    a. 1 bit (12 bit -> 13 bit)
    b. 12 bit (4 bit -> 16 bit)
4. ALU with special operations
    a. select A
    b. select B
    c. if A==B: 1
       else: 0
    d. if A==0: 1
       else: 0
    e. if B<A: 1
       else: 0
5. Adder
    a. Inputs - a[15:0], b[15:0] are two things to add
    b. Outputs - r[15:0] the unsigned result of adding a[15:0] and b[15:0]
    c. Control Bits - none
    d. Description - adds the 2 inputs as unsigned values
    e. RTL Symbols - +
6. 16 Bit Register
    a. Inputs - D[15:0] is the data to write, CLK the clock
    b. Outputs - O[15:0] is the output of the value in the register
    c. Control Bits - write determines if the PC should be set to value of D[15:0]
    d. Description - This is a component that can remember values and is able to be changed
    e. RTL Symbols - PC
7. Instruction Memory Block
    a. Inputs - ra[15:0] is the read address and CLK is the clock
    b. Outputs - d[15:0] is the data in memory at ra[15:0]
    c. Control Bits - none
    d. Description - This component is storage for instructions that are being executed
    e. RTL Symbols - Mem
8. Data Memory Block

    a. Inputs - ra[15:0] is the read address, wa[15:0] is the write address, wd[15:0] is the write data, and CLK is the clock
    b. Outputs - d[15:0] is the data in memory at ra[15:0]
    c. Control Bits - write tells whether or not to write to memory
    d. Description - This component is storage for data that is pushed by the programmer
    e. RTL Symbols - Mem
9. Control Unit
    a. Inputs - inst[15:0]
    b. Outputs - see control bits above
    c. Control Bits -
    d. Description - Takes in the instruction and then sets all of the control bits throughout the datapath to correctly run that instruction
    e. RTL Symbols - none
10. Stack Register
    a. Inputs - w[15:0] is what should be written to the top of the stack if anything, CLK is the clock, and reset sets all of the registers to 0 if it is high on the negedge
    b. Outputs - a[15:0] is the top value of the stack, b[15:0] is the value below the top
    c. Control Bits - stackOP[2:0] is what stack operation should be done, these include
        i. 000, none - used for halt and j, does nothing to the stack
        ii. 001, push - used for many instructions such as pushi, pushes to the stack what is on w[15:0]
        iii. 010, pop and replace - used for add, sub, or, and slt, pops the top off and replaces the next with w, which should contain a result of doing something with a[15:0] and b[15:0]
        iv. 011, pop - used for many instructions such as drop, top element is popped
        v. 100, pop2 - used for beq, pops top 2 things and uses them
        vi. 101, swap - used for swap, doesn't change stack size and swaps top two
    d. Description - This components output will always be the top two things currently on the stack. The stack can be changed by different stackOP and providing different values to w[15:0] to decide what to push or replace, data is written to the stack at the negative edge of the clock
    e. RTL Symbols - stack, Rstack
11. Merger (3 bit and 13 bit -> 16 bit)
    a. Inputs - a[2:0] and b[12:0]
    b. Outputs - r[15:0] which is a 16 bit bus with a[2:0] corresponding to its top 3 bits and b[12:0] being the bottom 13 bits
    c. Control Bits - none
    d. Description - This component takes a 3 bit and 13 bit input and combines them into 1 bus where the 3 bit input is now the most significant 3 bits.
    e. RTL Symbols - … (we used a … in the RTL to show that two busses should be joined)

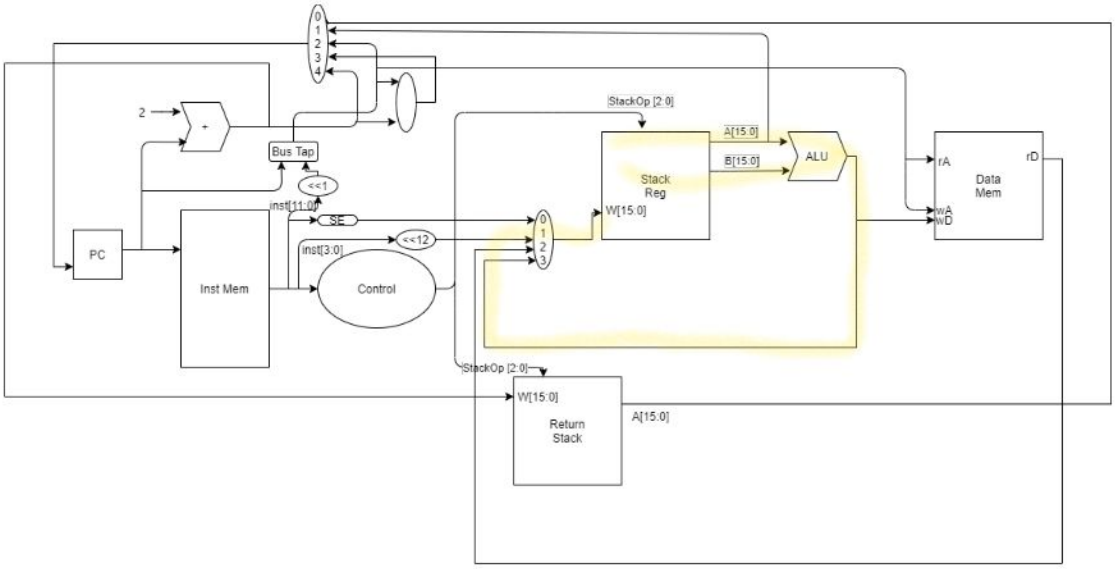
Unit Testing

1. Multiplexer
    - The test will put different inputs on each of the input wires, it will then go through each selection bit combination and see if the output is correct.
2. Sign Extender (12 bit -> 16 bit)
    - The test will give several 12 bit wires as input and see if the correct value is returned in 16 bit form. The value of the number representing signed binary should never change.
3. Left Shifter
    - Give various inputs, then verify that the number was shifted left the correct number of places and that zeroes were filled on the right.
4. ALU with special operations
    - Test each operation with various inputs. The operations are detailed above in the Required Components section.
5. Adder
    - Test that the adder does unsigned addition on various inputs. Make sure to check adding 2 as that is what it will be used for.
6. 16 Bit Register
    - Verify that the register can be written to and that the value can be later read after some cycles.
7. Instruction Memory Block
    - Verify that instructions can be put in memory through an initialization file. Also test that when getting these instructions that the correct instruction is retrieved for a given address.
8. Data Memory Block
    - Verify that initialization works using an initialization file. Verify that read works given an address. Verify that it does not write when the write control bit is low. Verify that it does write when the write control bit is high and that when read the new value is actually there.
9. Control Unit
    - Verify that for every instruction the correct control bits are set throughout the datapath.
10. Stack Register
    - Verify that each of the 6 stackOPs function correctly. Verify that the reset signal actually resets the register stack. Verify that the write happens on the falling edge of the clock while a read can happen on the rising edge.
11. Merger (3 bit and 13 bit -> 16 bit)
    - Verify that when given various 3 bit values and a 13 bit values that the result is a 16 bit value with the 3 bits from the input in the most significant bits and the 13 in the least.

Integration Plan

1. Push/Pop

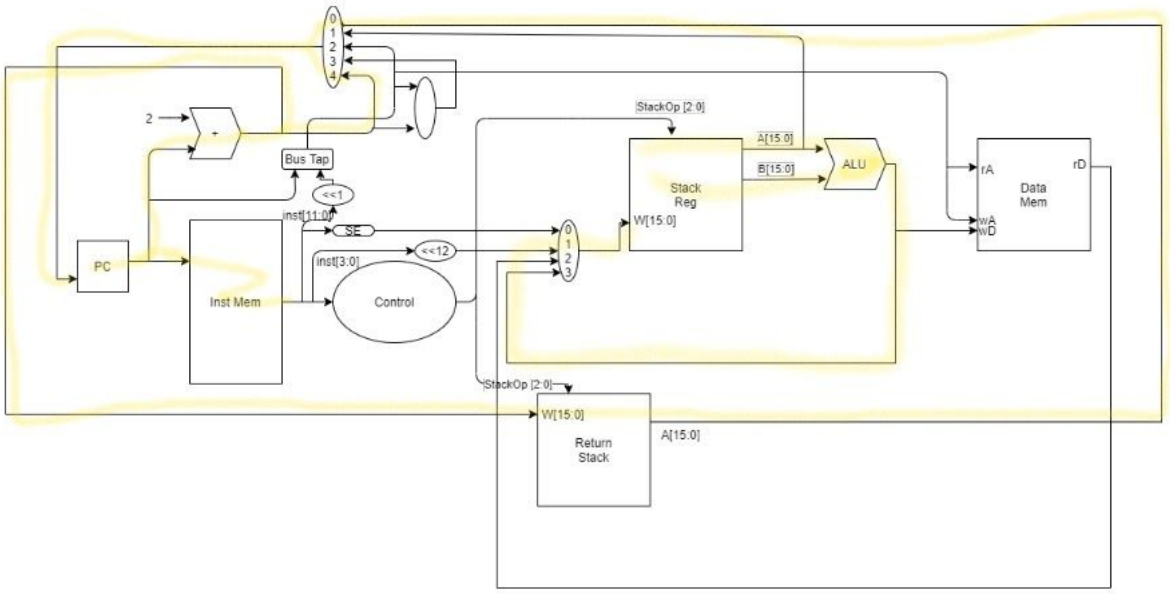
The first parts we plan to integrate are the Stack Register, ALU. Together, they should be able to push and pop on to the Stack depending on the instruction. Tests will start from initializing value in StackOps[2:0], ALUOp, and also put some values on the stack first, letting ALU perform different operations based on the control input, and then change the value of StackOps.The parts integrated in this step are highlighted in the picture below.



2. Updating PC

The next parts we plan to integrate are the PC, Adder, Instruction Memory, Return Stack, and multiplexers. Combining these parts, the value in PC should be updated the way we intend to. During the test, the value of RStackOp[1:0] and control in multiplexer will be initialized. Tests will also consist of repeating push and pop from Return Stack

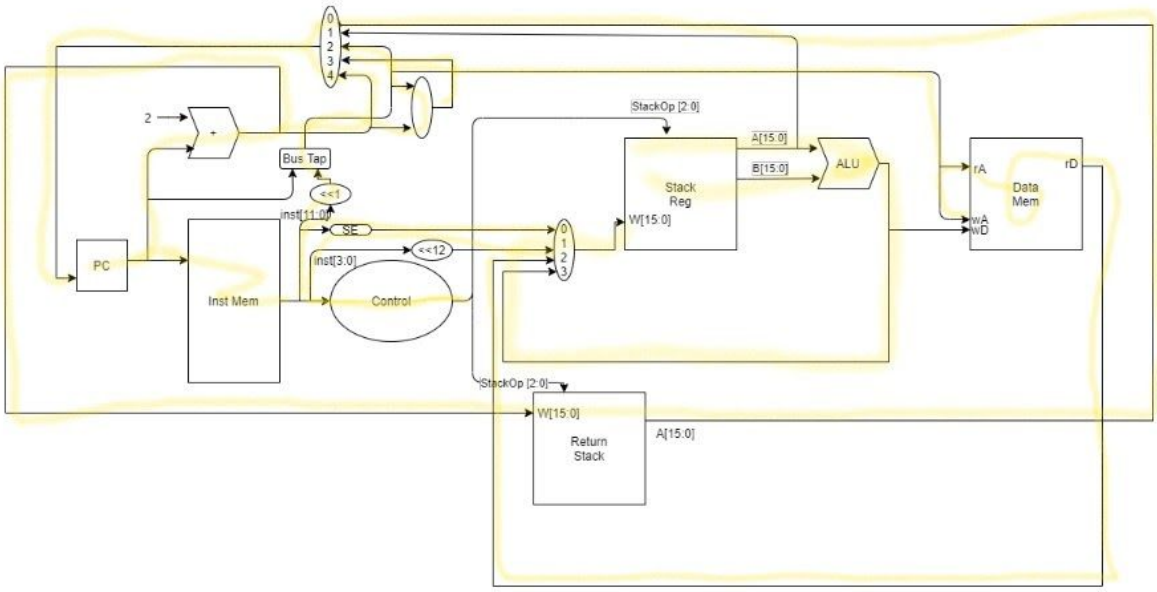
and monitoring the value in PC. The diagram below highlights the parts integrated so far.



3. Adding Control, Data Memory, Bus Tap, Sign-Extender, Left-Shifter.

The last parts we will integrate are Control, Data Memory, Bus Tap, Sign-Extender, Left-Shifter. Since we already have the PC part and Stack Part working individually, the test will consist of the select bit in PCControl that will output the value of BusTap or LeftShifter. After testing these two parts, the test will move on to selecting bit in stackControl that will output the result of Sign-Extender and the other Left-Shifter. Last but not least, the Data Memory will be tested by checking whether the rD will write right

on the stackReg and if we can write data correctly by inputting rA, wA and wD



Control Signals

1. PCControl - 3 bits
    - 0 = top of return stack
    - 1 = top of stack
    - 2 = PC[15:13]...(inst[11:0] << 1)
    - 3 = PC[15:13]...(inst[11:0] << 1) or (PC + 2)
    - 4 = PC + 2
2. StackControl - 3 bits
    - 0 = SE(inst[11:0])
    - 1 = inst[3:0] << 12
    - 2 = memory read data
    - 3 = ALU result
    - 4 = input
    - 5 = input2
3. StackOP - 3 bits
    - 0 = none
    - 1 = push
    - 2 = pop and replace
    - 3 = pop
    - 4 = pop 2
    - 5 = swap
4. RStackOP - 2 bit
    - 0 = none
    - 1 = push
    - 3 = pop
5. ALUOP - 4 bit
    - 0 = add
    - 1 = sub
    - 2 = and
    - 3 = or
    - 4 = xor
    - 5 = select A
    - 6 = select B
    - 7 = return A==B ? 1 : 0;
    - 8 = return A==0 ? 1 : 0;
    - 9 = return B<A ? 1 : 0;
6. PCWrite - whether or not to write to PC
7. MemWrite - whether or not to write to Data Mem

Control

('X' means that the control bit doesn't affect the result)

Name	Type	OP	Funct	stackOP	rstackOP	ALUOP	stack Control	PC Control	MemWrite	PCWrite
add	O	0x0	0x000	2	0	0	3	4	0	1
dup	O	0x0	0x001	1	0	5	3	4	0	1
drop	O	0x0	0x002	3	0	X	X	4	0	1
halt	O	0x0	0x003	0	0	X	X	X	0	0
getin	O	0x0	0x004	1	0	X	4	4	0	1
js	O	0x0	0x005	3	0	X	X	1	0	1
over	O	0x0	0x006	1	0	6	3	4	0	1
or	O	0x0	0x007	2	0	3	3	4	0	1
return	O	0x0	0x008	0	3	X	X	0	0	1
slt	O	0x0	0x009	2	0	9	3	4	0	1
sub	O	0x0	0x00A	2	0	1	3	4	0	1
swap	O	0x0	0x00B	5	0	X	X	4	0	1
getin2	O	0x0	0x00C	1	0	X	5	4	0	1
beq	A	0x1		4	0	7	X	3	0	1
bez	A	0x2		3	0	8	X	3	0	1
j	A	0x3		0	0	X	X	2	0	1
jal	A	0x4		0	1	X	X	2	0	1
pop	A	0x5		3	0	X	X	4	1	1
push	A	0x6		1	0	X	2	4	0	1
pushi	A	0x7		1	0	X	0	4	0	1
lui	A	0x8		1	0	X	1	4	0	1
addi	A	0x9								

To test control we will simply feed each instruction through control. Then we will check to see if all of the control signals in the table which are not "X" are set appropriately. If they are correct, then our control component works.

System Tests

1. Use **pushi** and **lui** instructions and verify that the correct values are pushed to the stack.
2. Push 2 numbers to the stack with pushi. Then do an **add** instruction and see if the top of the stack is the correct value.
3. Push 2 numbers to the stack with pushi. Then do a **sub** instruction and see if the top of the stack is the correct value.
4. Push 2 numbers to the stack with pushi. Then do an **or** instruction and see if the top of the stack is the correct value.
5. Push 2 numbers to the stack with pushi. Then do a **slt** instruction and see if the top of the stack is the correct value.
6. Test the stack manipulation instructions **dup**, **drop**, **swap**, and **over** to see if they correctly change the stack.
7. Test **getin** and **getin2** to make sure the correct input value is put on top of the stack.
8. Do a **halt** instruction and make sure that the PC no longer is updated.
9. Use **j** and **js** and verify that PC is set to the correct value.
10. Call a simple function with **jal** and make sure that **return** sets the PC to the correct return value.
11. Verify that **beq** and **bez** conditionally branch under the correct circumstances.
12. Verify that **pop** and **push** correctly interact with data memory. Make sure that if you have a push instruction directly after a pop instruction that the correct value is pushed.
13. Verify that the simple for loop as specified in the above Code Fragments section produces the correct output.
14. Verify that RelPrime produces the correct output for various inputs.

Performance

Running relprime(0x13B0)

- number of instructions in relprime - 37
- total number of bytes to store relprime - 54
- number of instructions executed - 122357
- number of cycles - 122357
- cpi - 1
- cycle time - 17.44ns or 57.3Mhz
- execution time - 2.134ms

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	2028	4656	43%
Number of Slice Flip Flops	2120	9312	22%
Number of 4 input LUTs	3569	9312	38%
Number of bonded IOBs	98	232	42%
Number of BRAMs	8	20	40%
Number of GCLKs	2	24	8%

Jinhao Sheng WORK LOG

MILESTONE 1 WORK Monday,

October 5, 2020 Met with team[120

min]

We decided to build a stack based processor. We created a Google doc to begin our design document. We then decide to have two stacks and no registers and implement two types of instructions.

Tuesday, October 6, 2020

Design procedure call convention[30 min]

Met with team[120 min]

We drafted out example assembly programs like relprime and other fractions. We also wrote the machine code for each example programs.

M2 task assignment:

- 1st meeting: break instructions into small steps and move data from one register to another, determine single-cycle or multi-cycle
- Luke & Austin: RTL Description of each instruction\
  Jinhao & Yiju : A list of generic components specifications needed for RTL
- 2nd meeting: debug and test the processor through Xilinx ISE and fix existed problems

MILESTONE 2 WORK

Wednesday, October 14, 2020

Read through Design Document

MILESTONE 3 WORK

Saturday, October 17, 2020

Design Datapath for O-type Instruction[30 min]

Sunday, October 18, 2020

Met with team[120 min]

We verified our RTL by iterate each instruction on a whiteboard. We then drafted out our initial design of our datapath.

Monday, October 18, 2020

Met with team[60 min]

We designed our stack register, and also finished design our datapath.

Drawing datapath on draw.io[30min]

Tuesday, October 20, 2020

Write out unit testing for multiplexer[60min]

Wednesday, October 21, 2020

Met with team[90min]

Refine datapath, Unit Specification, and write out integration plan

MILESTONE 4 WORK

Friday, October 23, 2020

Write out leftshifter and signextender. [40min]

Sunday, October 25, 2020

Met with team[120min]

We integrated the Push/Pop part of our integration plan.

Monday, October 26, 2020

Write out test for leftshifter and signextender, make changes to integration plan[60min]

Tuesday, October 27, 2020

Met with team[90min]

We integrated the second part of our integration plan.

MILESTONE 5 WORK

Monday, November 2nd, 2020

Figure out integration test testing and plan system testing [60min]

MILESTONE 6 WORK

Tuesday, November 10th, 2020

Prepare Presentation[30min]

Luke McNeil

##
Milestone 1

1) Wednesday, September 30, 2020

   [10 min]
   Talked with team through chat and decided to do a stack
   architecture.

2) Saturday, October 3, 2020
   [2 hours]
   Wrote a first draft of the GCD and RELPRIME functions This included
   making up instructions as I went along. I usedinstructions
   presented in Sid's lecture on the stack architecture, as well as
   borrowing ideas for stack manipulation from Forth.

3) Sunday, October 4, 2020

   [1 hour]
   Spent some time trying to figure out what actually uses a
   stackarchitecture in the real world, and get a better idea of how
   it isimplemented in hardware. Did some googling and
   wikipedia reading. Watched youtube videos on Forth and Java
   Bytecode.

4) Monday, October 5, 2020

   [2 hours]
   Met with team to discuss various requirements for M1. These include
   coming up with what additional registers we would need (none). We
   designed the format types O and A. After the meeting andhaving
   talked to Sid for some advice I decided that we could have
   twostacks of registers. One is the main stack which is for
   computations,and the other is a return address stack. This way
   nested functions canalways remember where they should return
   to. This also allowed me todescribe the function calling
   conventions. I wrote a code fragmentwhere main calls f1 which calls
   f2 which calls f3 showing theseconventions in use.

5) Tuesday, October 6, 2020

   [1 hour]
   I converted my RELPRIME and GCD to the table format that is in
   thedesign doc. I then refined the descriptions of our format
   types. Ithen listed out all of our instructions so far in a table
   providingtheir format type, argument if any, and a description. I
   also thencreated a table explaining how to convert O types to
   machine codewhich is pretty easy.

   [1.5 hours]
   Met as a group discussed how to convert all instructions to
   machinecode. Figured out the addressing modes we are going to
   use. I wrotesome additional code fragments explaining simple
   addition and gettinginput.

This is what we decided on was our plan for Milestone 2
M2 task assignment:
   a) 1st meeting: break instructions into small steps and move data
      from   one register to another, determine single-cycle or
      multi-cycle

```
   b) Luke & Austin: RTL Description of each instruction
   c) Jinhao & Yiju : A list of generic components specifications
      needed for RTL
   d) 2nd meeting: debug and test the processor through Xilinx ISE and
     fix existing problems

########################################################################
Milestone 2

1) Thursday, Friday October 8-9, 2020

   [4 hours]
   Created a simulator for our stack language written in Chez
   Scheme. The simulator can take a program such as our relprime
   example and then run it, simulating the stack as a list.

2) Tuesday, October 13, 2020

   [3.5 hours]
   Wrote the RTL descriptions for add, sub, or, dup, swap, drop, and
   over. While doing this I had to figure out how we would refer to
   the stack of registers in RTL. The way I decided to do it was with
   Reg.push(value) which pushes onto the stack a value and Reg.pop()
   which pops off the top thing on the stack.

   I also edited our design document in response to Sid's feedback of
   M1. This includes making places for a title page, table of
   contents, executive summary, and additional sections. I then
   combined the table showing instruction description with the table
   showing opcode and funct into 1 table. I also replaced exit with
   halt, an instruction which always jumps to itself.

   I then wrote an assembler for our stack language in Chez Scheme. I
   used several procedures from the simulator to read in the
   program. I then used a hashtable to match opcodes and functs, and
   then had to do some annoying stuff with converting a number to
   binary. This seems like it should have been easier since the number
   is actually being stored in binary, I just don't know how to get to
   that.

3) Wednesday, October 14, 2020

   [3.5 hours]
   I recreated the RTL for the ones I had previously done and added it
   for all of the instructions. I changed from using Reg.pop() and
   Reg.push() to treating the stack as an array of size 64 where
   stack[0] is the top of the stack.

   I had to think about what would happen when our stack machine runs
   out of registers on the stack. I decided there would be 64
   registers in the stack and 64 registers in the return address
   stack. For this round of RTL I just assumed that data would be lost
   when the user tries to add something to an already full stack. This
   is not ideal, but it will make the implementation easier. This
   might change in the future.

   I then made a factorial program and put it in
   implementation/example-programs/fact.asm. I used this to see if
   limit of 64 registers would be enough. In this example it is. This
   is because of the fact that factorial(8) is already bigger than a
   16 bit number. Using updates to the simulator I found that
   factorial(8) only had a max-stack-size of 11 and a
```

```
   max-return-stack-size of 10. This is plenty less than 64. This
   might not be true though in other examples where the result does
   not grow to be more than 16 bits so quickly. A good one to try
   would be fibonacci.

   I also refactored the Design document to contain many different
   sections to make it more readable.

For M3 I will
    a) Get up to speed in Xilinx
    b) Start on datapath
    c) Write tests for small components

########################################################################
Milestone 3

1) Saturday, October 17, 2020

   [1.5 hours]
   I created a xilinx project and pushed it to git. It has a mux and a
   test file from the course website.
   I added a note in the design doc that specifies what will happen if
   a programmer tries to use more than 64 registers. (some information
   will be lost)

2) Sunday, October 18, 2020

   [2.5 hours]
   We met as a teams and spent a while verifying RTL. This consisted
   of drawing out what would happen in various parts and seeing if it
   was what we wanted.

   We then started drawing out a datapath on the whiteboard. We ended
   up just keeping to add things until we basically had a datapath
   that could work for all of our instructions.

   I then started playing around with what our stack of register
   component would need as input and output and what it should
   actually do. I started trying to draw out how it would work with a
   numPush and numPop input as well as 3 write wires. The output was
   the top of the stack and second from the top. I quickly realized
   that this was too much. I realized that when doing an add rather
   than doing two pops and then a push, it is simply a pop and then a
   replace.

3) Monday, October 19, 2020

   [4 hours]
   Jinhao and I met with Sid during office hours. This helped a lot in
   our design of the datapath and register stack. He gave us the
   inspiration to simply the register stack into having two inputs: a
   stackOP, and a w which is write data. The outputs remain the same
   just being the top and next value. This makes the design of the
   register stack much simpler.

   I met with Jinhao to further discuss the datapath. We drew a neater
   version of what we had on to the whiteboard. We ended up having to
   make some design decisions. For one we added another memory block
   separate from instruction memory. This is similar to what was done
   in single cycle in the book. Next we decided to push back some
   functionality into the ALU. We needed for example only to get out
   the second input given to the ALU. Rather than add a multiplexor to
```

```
   the first input giving an option for it just to be zero and telling
   the ALU to add, we decided that we could just make a special OP to
   give to the ALU to just give back the second input.

   I then implemented the register stack in Verilog. Most of the time
   spent here was figuring out how to get everything set up, not
   necessarily working on the logic. Once I got to the logic part it
   was pretty easy to just use for loops to shift the stack up or
   down. One thing I need to think about is when the write is
   done. Right now I have it set to do it on the posedge. I think we
   might want to switch that so that the ALU can read from the
   register stack, put a result on w, then on the negative edge of the
   clock w will be put on top of the stack. From my first
   implementation I had the for loop iteration for pushing and popping
   backwards which meant that I was overriding some data. Testing
   helped me find this issue.

3) Tuesday, October 20, 2020

   [2 hours]
   I spent a lot of time messing with the register stack to make it
   faster and smaller. It seems to be making a bunch of flip-flops for
   some reason that has to do with me doing something wrong. I'm not
   really sure what that is, but I will probably return to this
   component later to make it more efficient.

   I then implemented the merger component which was a verilog
   one-liner.

4) Wednesday, October 21, 2020

   [2 hours]
   We met as a group and ironed out the component list. I then added a
   unit testing plan description for each component. Jinhao and I then
   created several integration plans for testing to be done going
   forward.

Things for Milestone 4
       - Continue implementing necessary components and adding unit tests
                Jinhao - sign-extender
                Austin - adder
                Jinhao - left-shifter
                Luke - register
                Someone - memory blocks
       - Start on integration testing plans
            Group Meetings (might split out individual work later)
       - Implement Control
                 Luke

########################################################################
Milestone 4

1) Sunday, October 25, 2020

   [2 hours]
   Met as a team and started implementing the push/pop integration
   test. I decided that we should add more outputs to integration test
   then we had previously thought. I added a topOfStack,
   SecondOfStack, and ALUResult so we could thouroughly test the
   pieces. Additionally, we created a table that completely describes
   our control components logic. Since we are doing single-cycle this
   is simply a truth table.
```

2) Monday, October 26, 2020

   [2.5 hours]
   I finished up the testing for push/pop integration step. I then
   created a verilog component for control. I had to document in the
   design document what each control signal value meant. I also had to
   add an input option to the stackControl control signal so that we
   could correctly implement getin.

3) Tuesday, October 27, 2020

   [4 hours]
   We met as a group and implemented the updating PC integration
   test. This went pretty good, but there was some confusion with
   timing. We also do not have a memory block component yet, so we
   could not include that component in this test.

   I then added a blockmemory component. I had to think about how big
   we wanted both the data memory and instruction memory blocks to
   be. I decided that the data memory should have 2^12 addresses
   because that is how big our immediates are in
   instructions. Instruction memory can then be as big as we want or
   as little as it needs to be.

Things for Milestone 5
       - finish integration testing
       - get processor working, I think we should have it running
       relprime for this milestone

##
Milestone 5

1) Wednesday, October 28, 2020

   [2 hours]
   I got the Spartan board and tried to run the aluIO project on
   it. This worked, but only on Windows which was a pain. I then
   altered the project to contain our register stack instead of the
   ALU. I fed the same op output to the register stack. So for example
   you would go to the or operator "|" do to a push. To do these
   operations at discrete times I set the clock of the register stack
   to be a button on the Spartan board.

2) Friday, October 30, 2020

   [2 hours]
   I worked with Austin to get the last of the integration tests up
   and running. Here I had to determine which instructions we could
   actually test with this subset of hardware. This included pop and
   push which had not yet been tested. During this we made the change
   to the smaller memory blocks as discussed in the previous
   milestones.

3) Saturday, October 31, 2020

   [5 hours]
   Met with Austin to try and put the whole datapath together in
   verilog. Reading through the code a second time to make sure it
   matched our datapath caught several small bugs. This turned out
   pretty good and we ended up getting every instruction working

except for push and pop. To do this we had to think carefully about the clock. We made instruction memory read on the rising edge, and on the falling edge we had PC be set, and the register stacks written to. This allowed us to get RelPrime running in the simulator. We then spent maybe an hour or two trying to get it on the Spartan board with little success.

4) Sunday, November 1, 2020

[2 hours]
Met with Jinhao and Austin. Tried to get Jinhao in a place where he could change the coe file then regenerate and run those instructions. After this meeting I spent time running through the push and pop instructions to see exactly what we would need to do with the clock to make them work. I decided that we needed a clock edge between the rising edge and the falling edge of the one we currently had. I was not sure how to do this though.

5) Monday, November 2, 2020

[4 hours]
Met with Sid in the morning to discuss the clock issues we were facing. He gave the insight that we could put our fast and slow clock into and/or gates to produce the kind of clocks we wanted. I then added this logic to our verilog and made sure that push/pop worked by running a simple set of instructions.

I then spent a couple hours getting RelPrime running on the spartan board. This was made much easier by getting the ability to push to the spartan board from Linux (thanks Sid). I then took out the alu from the aluIO project and replaced it with our final_processor component. I made a couple additional modifications. The reset pin to the processor is the wire that is high whenever the rotary is changed. This way, whenever the input value is changed, PC will be reset to 0 and the processor will rerun the current program. This required me to replace our current PC which used synchronous clear with another 16 bit register with asynchronous clear. With these changes I could now see the results of RelPrime with any input.

6) Tuesday, November 3, 2020

[2 hours]
Met with Austin and added system test specifications to the design doc. This includes plans to test small sets of similar instructions then move up to trying bigger programs like RelPrime.

I also added another instruction "getin2" to get input from a second input wire. I just added this because I wanted to able to do computations on two numbers on the Spartan board.

##
Milestone 6

1) Monday, November 9, 2020

[5 hours]
I tried to add an addi instruction. This includes adding a multiplexer to the datapath right before the ALU. For some reason this change nearly double the processor's overall clock speed. I can not really figure out why this is since control is already setting the ALUOP, so another control on a mux which is the input to that ALU should not matter. Anyhow, I decided not to add addi

because of this.

I also added the timing stuff as requested in Milestone 6. The only tricky thing I had to do here was add an instruction count register to the control component to count up whenever the instruction was anything but halt.

2) Wednesday, November 11, 2020

[3 hours]
I implemented all of the system tests. In final_processor_tb I added code snippets and machine code translations to test every instruction and some bigger programs. To perform these tests you should copy the machine code for a specific system test into blockmemorykx1.mif and then uncomment that test in final_processor_tb. I had to make a couple fixes here. First, a push directly after a pop would not work correctly. I fixed this by connecting the top of the stack directly to data memory write data instead of waiting for the ALU. The second fix was to shift the top of stack left 1 bit before sending it to PC for the instruction js.

Austin Swatek: 232-2v Project Work Log

Milestone 1

Monday, Oct 5, 2020: 2 hours
Writing design document and figuring out processor specifics: stack-based, addressing, registers

Tuesday, Oct 6, 2020: 2 hours
Writing sample programs in assembly
Converting said assembly to machine code

M2 task assignment:
1st meeting: break instructions into small steps and move data from one register to another, determine single-cycle or multi-cycle
Luke & Austin: RTL Description of each instruction
Jinhao & Yiju: A list of generic components specifications needed for RTL

Milestone 2

2nd meeting: debug and test the processor through Xilinx ISE and fix existed problems

Wednesday, Oct 14, 2020: 1.5 hours
Reviewing Luke's RTL descriptions
Creating Executive Summary

M3 task assignment: Create design diagram
Begin Xilinx work
Begin Xilinx testing

Milestone 3

Sunday, Oct 18, 2020: 2.5 hours
Verifying RTL
Created initial datapath design

Tuesday, Oct 20, 2020: 1.5 hours
Created ALU with unique operations

Created ALU testing program

Wednesday, Oct 21, 2020: 2 hours
Came up with integration plan
Refine datapath, unit specification

M4 task assignment:
Things for Milestone 4

- Continue implementing necessary components and adding unit tests
    - Jinhao - sign-extender
    - Austin - adder
    - Jinhao - left-shifter
    - Luke - register
    - Someone - memory blocks
- Start on integration testing plans
    - Group Meetings (might split out individual work later)
- Implement Control
    - Luke

Milestone 4

Sunday, Oct 25, 2020: 2 hours with team
Integration of the stack register and ALU. This makes it possible to push/pop to/from the stack. Testing of integration and refining the integration based on the results of tests was done to make sure it worked properly, and will continue to work properly as other components are integrated.

Monday, Oct 26, 2020: 1.5 hours
Began working on integrating the PC and updating the PC. Determining inputs and outputs for each component to integrate. Did not finish Verilog integration. Did not create tests.

Tuesday, Oct 27, 2020: 1.5 hours with team
Started anew with integration of PC and PC updating. Decided to use Spartan 3E's integrated FD16RE: 16-bit data register during integration. Simple testing was done to check PC changes correctly.

Milestone 5

Friday, Oct 30, 2020: 0.75 hours
finishing integration tests. fixing push and pop, checking mem

Saturday, Oct 31, 2020: 3 hours with Luke
Checking and fixing final implementation of processor: updating control bits and signals to proper values.
Verified that rel-prime works, also tested a simple for-loop and other basic tests.
Attempted to put on Spartan board, but we were not successful.

Sunday, Nov 1, 2020: 1 hour
Luke was trying to help Jinhao and I fix issues with running the completed processor on our computers, since there was an issue with a component using a hard-coded path from when Luke integrated it.

Tuesday, Nov 3, 2020: 0.75 hours with Luke
Writing all of the system tests in the design document.
Deciding what other things need to be tested, if at all.
Figuring out if there are other instructions that should be added or could be added.

Milestone 6

Tuesday, Nov 10, 2020: 1.5 hours
Preparing presentation document. Started basic details, spent time figuring out what to include in it when looking at the requirements for it. Format is almost finalized.

Wednesday, Nov 11, 2020: 2 hours
Ironing out presentation layout and content. Still deciding on small details to include/remove, thinking about challenges during the project and things we would like to improve/change if we were to continue working on it. W

Upcoming week:
Finishing the presentation, then running through it with Luke and Jinhao. After mock presentation, can touch-up/finalize the details.

Begin cleaning up the final design document so it is ready for submission on Monday.