

**The Eggo Stack**  
**By: The Toaster Troop (Team Number: 2V)**



William Dalby, Christian Meinzen, Victoria Szalay

## Table of Contents

Patch Notes:	3
Synopsis:	5
The Eggo Stack:	6
Overview of Instruction Set	6
Implementation	7
Testing Methodology	8
Assembler	9
Memory Mapped I/O	9
Concluding Statements:	10
Appendix A: List of Commands	12
Appendix B: ISA designs	13
Appendix C: Reserved Registers	13
Appendix D: Addressing Modes	13
Appendix E: Procedure Calling Convention	14
Appendix F: Local Variable Convention	14
Appendix G: Chosen Size of Register Stack	14
Appendix H: Memory Allocation	15
Appendix I: RTL	16
Appendix J: State Transition Diagram for Multicycle	18
Appendix K: Datapath	20
Appendix L: Control Symbol Tables	21
Appendix M: RelPrime Code	24
Appendix N: Common Code Snippets	28
Appendix O: Instruction Description and Semantics	35
Appendix P: System Components	42
Appendix Q: RTL Verification	52
Appendix R: Testing Methodology	54
Appendix S: Integration Plan	55
Appendix T: System Testing Plan	58
Appendix U: Assembler Semantics and Details	59
Appendix V: Processor Testbench Data	62
Appendix W: Team Member Journals	62
William Dalby's Journal	62
Christian Meinzen's Journal	66
Victoria Szalay's Journal	73

#### Patch Notes:

- Methodology for pushing things to the memory stack has changed. Instead of pushing the upper and lower immediates of the memory address we wish to store in, it is relative to a Stack Pointer. The callee must move their stack pointer and then move it back upon return.
- Added a funct to the C type instruction to allow for pushing to memory and registers separately.
- Changed the A-type ISA to have a duplicate amount along with a shamt.
- The procedure calling was changed in that instead of putting in an address to return to, we push a label that is after the jump.
- Updated ISA, common operations, and relprime code to reflect these changes
- Some of the inputs for the hardware have been changed to be control bits. For example, flip is now a control bit on the execution stack rather than an input.
- The memory addresses can only be 10 bits long so the input to memory to read or write to has been clipped to 10 bits.
- Shift methodology has been changed. We now send a signed shamt to the shifter and it shifts the appropriate amount in the appropriate direction (left > 0, right < 0).
- The datapath was changed to make the input to the shift component 16 bits with the sign extender since our original implementation invalidly inputted an 8 bit value.
- The calculation for shifting was moved to a dedicated or gate rather than through the ALU.
- We removed two inputs to the ALU (the clipped instruction, and 0) as they were never used.

## Synopsis:

The Eggo Stack (ES) architecture is a predominantly stack architecture created by the group known as the Toaster Troop. Said group consists of Victoria “Tori” Szalay, Christian Meinzen, and William “Will” Dalby. It operates with 16 bit words and has a 32 register stack. It has 18 operations which can be seen in the *List of Commands* section (pg 4). The semantics of how the stack works with these operations can be found in Appendix C (pg 32). This architecture has three instruction set designs consisting of A, B, and C type. These instruction sets have 4 bit opcodes. A-type handles arithmetic operations involving the stack, B-type handles branching and jumping operations, and C-type handles pushing and popping with immediates. Due to the fact that we are working with 16-bit words and 16-bit instructions, loading memory addresses into the stack is done in two parts. The execution stack loads the upper byte of the immediate, then the lower (or vice versa) and or’s them together to put the full address at the top of the stack. Push and pop operations then look at this address at the top of the stack to find where the to push or pop from. A similar operation is performed for the jump stack (js) command which takes an address at the top of the stack and sets the PC equal to it. There are 4 reserved registers to hold a stack pointer, a global pointer, a zero value, and a return value. ES uses Pseudo Direct addressing for branching and jumping. There is a  $2^{16}$  bit memory stack where 0x0000 to 0x2000 is reserved for program text, 0x2001 to 0x7fcd is reserved for dynamic and static memory/stack allocation, and 0x7fce to 0x7ffe is reserved for input and output components. The Eggo Stack runs at a cycle frequency of 65.6 MHz and is able to execute the rel\_prime code in 183,774 instructions. These instructions have a CPI of 4.055. The Eggo Stack is able to run rel\_prime in 11.35 ms.

## The Eggo Stack:

### *Overview of Instruction Set*

The Eggo Stack is a stack type architecture. It consists of a 32 register deep execution stack with 16 bit wide registers. 32 registers was deemed all that was necessary for most general purposes. The system has a 10 bit memory but was designed with 16 bit memory in mind and as such has allocations for a 16 bit memory file. Items from 0x0000 to 0x2000 are dedicated to text memory. This includes any and all machine code for the processor to perform. The addresses that are reserved for dynamic data are addresses 0x2000 to 0x7ff0. These addresses would store any values that the programmer desires while writing programs. Items above 0x7ff0 are reserved for I/O. This mapping translates directly to a 10 bit memory file as well.

The Eggo Stack has 18 instructions of 3 types. All instructions have a 4 bit operation code. The first 6 instructions are for pushing and popping and are of type C. This type of instruction has an 8 bit immediate from bits 11 to 4. The immediate is used for pushing immediates as well as accessing different memory locations. C type also has a two bit function code (funct) which distinguishes between register pushing/popping and memory pushing/popping. A funct of 0b00 implies pushing/popping to memory and a funct of 0b01 implies pushing/popping to registers in the register file. The final notable aspect of C type instructions is that there is a two bit section for register addresses. The next 8 instructions are of A type and are used to perform arithmetic operations. There is a 5 bit shift amount (shamt) which holds a signed 5 bit value which dictates how many times a shift is performed. There is also a 2 bit duplication amount (damt) which determines how many items are duplicated from the execution stack. Arithmetic operations are performed in reference to items already in the stack. For example, the instruction 'add' takes the top two items from the execution stack and adds them together. The result is then placed back on the execution stack. Finally, the last 4 operations are of type B and handle jumping and branching. Instructions of this type have a 12 bit immediate value which is used to determine the address to jump to. For the 10 bit memory, no operation needs to take place with these as an entire address can fit into that immediate slot. As for 16 bit memory, pseudo-direct addressing would be used where the upper 4 bits would come from the current PC and the lower 12 would be from the immediate. There is a jump stack (js) command which jumps to the address at the top of the execution stack. This could be used to jump to any 16 bit address in the system.

There are four reserved registers consisting of a stack pointer, a global pointer, a return value register, and a zero register. These correspond to 0b01, 0b10, 0b00, and 0b11 respectively. The global pointer is currently not used. It can be used as a method to store local variables rather than in the stack at the moment, however. The stack pointer is initialized to 0x7ff0 which is the top of the dynamic data segment of memory. It points to the top of memory and grows down. This is where the immediate section of type C instructions is used to point to memory locations

in reference to the stack pointer. The return register is used to put return values in a register. The zero register is not writable and stores a 0. This can be used to put a 0 into the execution stack easily or to eliminate values from the execution stack by popping them to the zero register.

### *Implementation*

The entire processor can be broken down into four subsystems. The first subsystem is the PC subsystem. This system consists of the PC register and any components needed to change the PC when branching, jumping, or simply going to the next instruction. There is an adder which adds 1 to the PC. This is for the 10 bit memory which is not word aligned. For a 16 bit word aligned memory, this adder would add by 2. This adder is used to get the PC to be the next instruction's address. This would be done after any operation is performed. There is also an 'or' gate which combines the upper 4 bits of the PC with the lower 12 bits of the instruction. This is for jumping or branching address calculations. This or gate works for both the 10 and 16 bit memories as the upper 6 bits do not matter and as such will be chopped in the 10 bit implementation anyway. There is also an input from the 'A' register. This is for the jump stack command as that register would contain the value that it would jump to. Finally, there is a series of logic gates that control whether the PC writes or not. Firstly, there is PCwrite. This is for normal operations and allows the PC to increment. Next, there are gates for branching conditions. The first is for branching on equal (beq) where we must be doing a beq command and have the output from the Arithmetic Logic Unit (ALU) be zero. This is so that when the ALU performs a subtraction (to find out whether the two items are equal) they should subtract to 0 and thus be equal which means we must branch. The next is for branching on not equal (bne) where we must NOT have an output of 0 which means the two items were not equal and thus we must branch.

The next subsystem is the push subsystem. This consists of items which could be pushed into the execution stack. There is a Sign Extender and a Zero Extender which append the sign bit and zeros to the front of an 8 bit input respectively. The Zero Extender is used for push lower immediate (pushLi) to isolate the lower 8 bits. The Sign Extender is used to get 16 bit inputs for the shifters (logical and arithmetic). The shifters take in a 16 bit value and shift them by a certain amount based on their operation. The logical shifter shifts in zeros whereas the arithmetic shifter will shift in the sign bit (when shifting right) or zeros (when shifting left). The shamt comes either from the static 8 for pushing upper immediates (pushUi) so that the upper 8 bits are isolated or from the shamt in the instruction. There is also a register file which holds the reserved registers. Which value is read from depends on the register specified by the instruction and whether the registers are written to is determined by the regWrite control bit. The top and second from top values of the execution stack are also available here, in addition to the output of the ALU and the output from memory. All of these items are passed through a mux to the pushVal

register. This register holds the value to be pushed onto the execution stack. The mux is controlled by pushVal control bit.

The next subsystem is the memory subsystem. Items in this subsystem include the memory component, the PC register, the ALUout register, and outputs from memory (MemOut and IR). This system is dedicated to accessing different memory locations and making sure they output the correct values. There is a mux which differentiates where memory should read or write to. If the Instruction or Data (IorD) bit is 0, memory is read or written from the address in the PC. This would be for fetching the next instruction. If the IorD bit is 1, the memory address is from ALUout. This is the address that would be calculated from the pushM/popM commands with their 8 bit immediate. This would be for reading from memory and pushing that to the execution stack or for writing to memory a value from the execution stack. The write value for memory is always the second item from the execution stack (B).

The final system is the execution stack system. This consists of the execution stack and its inputs as well as the ALU. The A and B registers are constantly being fed the top two items from the stack. The ALU is also constantly performing the operation specified by ALUop. The ALU can perform 'or', 'add', 'sub', and 'set less than (slt)' with control values of 0b00, 0b01, 0b10, and 0b11 respectively. SLT outputs a 16 bit value with either 1 (if input 1 was less than input 2) or 0 (if input was not less than input 2). The ALU also has a zero flag which indicates if the current operation has a result of zero. This is used to calculate branching logic. The execution stack also has four operations consisting of push, pop, duplicate, and flip - 0b00, 0b01, 0b10, 0b11 respectively. The push value comes from the pushValue register. The amount of values to pop off are determined by the popAmt control bit. Duplicate takes the top items (determined by the damt) and duplicates them in the order that they appeared. Flip takes the top two items and flips their position. No items in the stack can be modified if the ESAct bit is 0. The A and B registers still read the top two items but the execution stack itself does not change unless specified.

### *Testing Methodology*

We used several different testing methods while testing the multiple implementation layers of our Eggo Stack. For component testing, we focused on testing all of the functions provided for a particular component and any edge cases presented by those functions. For example, our sign extender has the single function of extending the most significant bit of an 8-bit value into a 16-bit value. This component has two different edge cases of the significant bit either being a 1 or a 0. Using our testing methodology, this means we tested but edge cases separately in our testbench.

For our integration testing, we were combining our components together by subsystems of the overall datapath that will perform operations together. After combining them, we tested our subsystems by hardcoding our inputs and ensuring that we received the expected outputs. An

example of this is our PC subsystem. In the PC subsystem, the inputs are the control bits to the PC register's write enable and the mux that decides the source to the PC. We tested this subsystem by inputting different combinations of the control bits and checking our PC register value. The integration testing was completed by combining all four of our subsystems together into a complete datapath.

For our system testing, we tested our new datapath by manually setting control values for each of 18 instructions and checking that the correct components were changed/modified. We then combined our control and datapath. We first tested this system by inputting a single instruction and ensuring that it ran properly. After that, we ran the same series of instructions that we ran on the datapath alone and checked that we received identical outputs.

Throughout our testing procedures, we made sure to apply any fixes that were indicated by failed tests.

We used several different testing methodologies while testing the processor.

### *Assembler*

The Eggo Stack also comes with a Java Assembler which converts assembly code to machine code. This assembler is a three pass assembler. The first stage consists of translating pseudocode into regular assembly instructions. Pseudocode consists of a push command which pushes a full 16 bit value, a branch greater than, branch less than, branch greater than or equal, and a branch less than or equal. This first pass also converts hex (0x) and binary (0b) to decimal values. The second pass assigns addresses to the instructions and assigns labels to addresses. Finally, the last stage takes the instructions and converts them to machine code. This means writing the opcode and then the appropriate immediate values. Once the machine code is obtained, the final item is written to a .coe file of the users choice. More details on usage can be found in Appendix U.

### *Memory Mapped I/O*

Implementing memory mapped input/output into the Eggo Stack is an additional feature that allows for quick changes in program arguments. Instead of having the assembler reassemble different code every time the arguments to the program change, certain memory addresses are used for input that could later be implemented by hardware components. The traditional way to add arguments to a program would be to use the pushLi and pushUi instructions to add the 16-bit immediate onto the stack. Now, with memory mapped I/O, the first instructions are pushR \$sp and pushM n, where 'n' is some number greater than one. These instructions will take the address of the stack pointer and add 'n' to it so that it will access a reserved I/O address. The Eggo Stack will then take the inputted number as the argument to the program. This needs only the input value to be changed rather than the entire datapath to recompile. When the processor is



completed, it can place the answer in an output register that could be hardcoded to any hardware component. This is done by the last instructions consisting of a pushR \$sp and popM n, just like the input instructions. Both the input and output registers can be seen in the simulation module for easy testing. It is important to note that the memory addresses for the I/O starts at 0x7fcd and goes up to 0x7fff. This reserved memory provides a large amount of space for any hardware implementations as well as leaving a large space for the dynamic data storage in stack memory.

### *Concluding Statements:*

The Eggo Stack architecture implementation has exceeded expectations. It runs relPrime code put together through our assembler that can take ordinary stack assembly code, and convert it into binary that the processor can then compute. This is done quite efficiently. The Eggo Stack runs at a cycle frequency of 65.6 MHz and is able to execute the rel\_prime code in 183,774 instructions. These instructions have a CPI of 4.055. The Eggo Stack is able to run rel\_prime in 11.35 ms. With our I/O implementation, it is easy and quick to change the arguments to the code without additional compilation or assembly just by changing the input\_IO value.

Some challenges that showed up along the way include memory, I/O, timing issues, stack procedure, and combining subsystems. It was a long process to test memory because of the constant reassembling. After implementing the block memory, it was still a hassle to check any subsystem that included the manipulation of memory. Input/Output was also tedious work due to the amount of variables that could be factored in. After the whole datapath was implemented, any issue with I/O could come from any part of the datapath, which made it hard to track down. Timing issues were probably the main source of issues when implementing components. Figuring which components needed to be on the positive edge and negative edge of the clock cycle was not always clear, especially when dealing with memory, since part of memory was blocked compared to our traditional approach of unblocked components. The last struggle of implementing our architecture was combining subsystems to configure for stack procedures. Since the stack architecture allows access to the top two elements on the stack at one time, the datapath design needed to centralize around the stack in a way so that any operation performed on the stack would happen quickly. This challenge was present early on in the design stages, and was present throughout the entire process. Despite these challenges, however, our team was able to take each problem and find a solution so that the processor could perform as intended.

Looking forward, some further implementations our team would like to make to the Eggo Stack architecture include faster run-time, handling stack overflow, FPGA board I/O, and a more advanced assembler. While testing and debugging our current design, our team noticed that our relPrim code and RTL design could be improved slightly for faster run-time. Handling stack overflow and interrupts is another implementation that our team looked at, but did not get around to implementing. It would be very interesting to add a handler because it might bring additional

troubles like FPGA space overflow. Along with a handler, it would be nice if the FPGA board could provide an input, our processor run code, and spit out an answer displayed on the FPGA board again. This was attempted, but unexpected errors occurred when connecting the rotary debouncer, which put a halt to the FPGA implementation. Another improvement that could be made is with our assembler. Right now, it functions quite well to take stack assembly code and convert it into binary, but with more time, an advanced assembler could be made to include more pseudocode instruction, automatic file import, and automatic memory updating. While there are multiple directions this design could go, the current version of the Eggo Stack architecture proves its effectiveness in code processing.

## Appendix A: List of Commands

Command	Opcode	Funct	Type
pushM	0x0 / 0b0000	0b00	C
popM	0x1 / 0b0001	0b00	C
pushR	0x0 / 0b0000	0b01	C
popR	0x1 / 0b0001	0b01	C
pushli	0x2 / 0b0010	n/a	C
pushui	0x3 / 0b0011	n/a	C
dup	0x4 / 0b0100	n/a	A
flip	0x5 / 0b0101	n/a	A
or	0x6 / 0b0110	n/a	A
add	0x7 / 0b0111	n/a	A
sub	0x8 / 0b1000	n/a	A
ls	0x9 / 0b1001	n/a	A
as	0xA / 0b1010	n/a	A
slt	0xB / 0b1011	n/a	A
beq	0xC / 0b1100	n/a	B
bne	0xD / 0b1101	n/a	B
j	0xE / 0b1110	n/a	B
js	0xF / 0b1111	n/a	B

## Appendix B: ISA designs

### Arithmetic Items: A-Type

4 bits	5 bits	5 bits	2 bit
OPCODE	UNUSED	SHAMT	DAMT (Duplicate amount)

### Jumping / Branching: B-Type

4 bits	12 bits
OPCODE	ADDRESS/IMMEDIATE

### Items involving immediates: C-type

4 bits	8 bits	2 bit	2 bits
OPCODE	IMMEDIATE	FUNCT	RESERVED REGISTER

## Appendix C: Reserved Registers

1. Return Value - V0 = 0b00
2. Stack Pointer - SP = 0b01
3. Global Pointer - GP = 0b10
4. Zero Register - zero = 0b11

## Appendix D: Addressing Modes

Pseudo Direct: For jumping and branching - 12 bits of address and 4 bits from PC

## Appendix E: Procedure Calling Convention

There are two distinct procedure call conventions that Eggo Stack is capable of.

1. The return address is placed at the top of the stack. Arguments are stored at the top of the stack. The Procedure is then called. The return value is stored in Mem[V0] by the callee. Caller then retrieves this and puts it on the stack. This means that when the caller regains control of the stack, it will contain only old values. In order to use this type of procedure call, the programmer can pushUi and pushLi the label that stores the return address and then “or” the two values together. This is not the type of procedure call that we use in our relprime code.
2. The arguments to a function are stored at the top of the stack before the programmer calls that function by jumping to a label. The callee is responsible for not deleting any values on the stack below its arguments. A function returns its values by keeping its return values on the top of the stack. Local variables are saved to memory as decided by the programmer if stack operations become too complicated. This is the

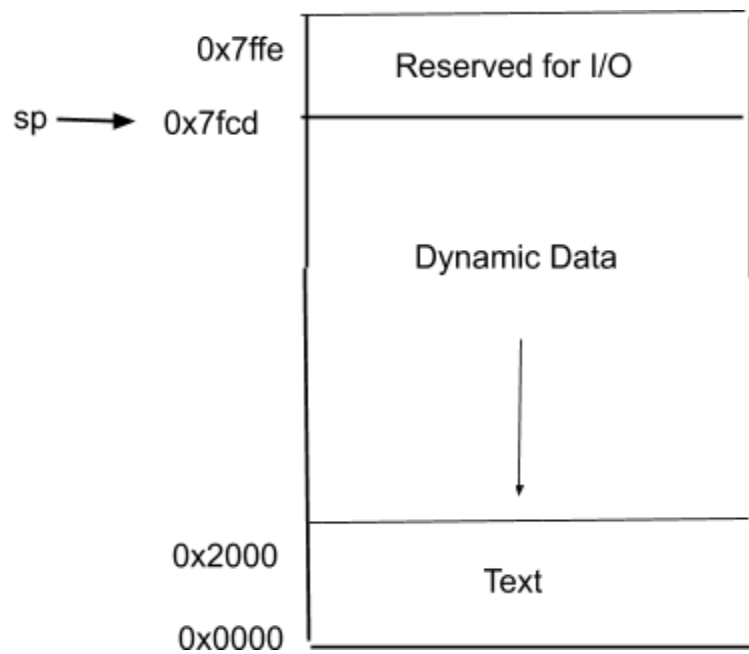
## Appendix F: Local Variable Convention

The programmer is responsible for storing local variables in the memory stack. This is done using push/pop M in reference to the stack pointer.

## Appendix G: Chosen Size of Register Stack

We have chosen the size of our register stack to be constructed out of 32 16-bit registers. This was considered to be more than enough registers for general use.

## Appendix H: Memory Allocation



## Appendix I: RTL

Step	Push/Pop M	Push/Pop R	Arithmetic/ Logic	beq/bne	j	js
	newPC = PC + 2 PC = newPC Inst = Mem[PC]					
	A = ES[Top] B = ES[Top + 2]					
	ALUout = A + SE(inst[11-4])	popR: Reg[inst[1-0]] = A  ES(pop 0) pushR: skip	ALUout = A op B ES(pop 1)	ALUOut = A == B	PC = PC[15-11]    inst[11-0]	Address = pop PC = address
	popM: Mem[ALUout] = ES(Top + 2) pushM: ES(pop 0)	pushR: ES(push Reg[inst[1-0]])	Skip	ES(pop 1) if ALUOut == 1 then PC = PC[15-11]    inst[11-0]	skip	ES(pop 0)
	popM: DONE pushM: MemOut = Mem[ALUout]		ES(push ALUout)			
	pushM: skip					
	pushM: ES(push MemOut) DONE					

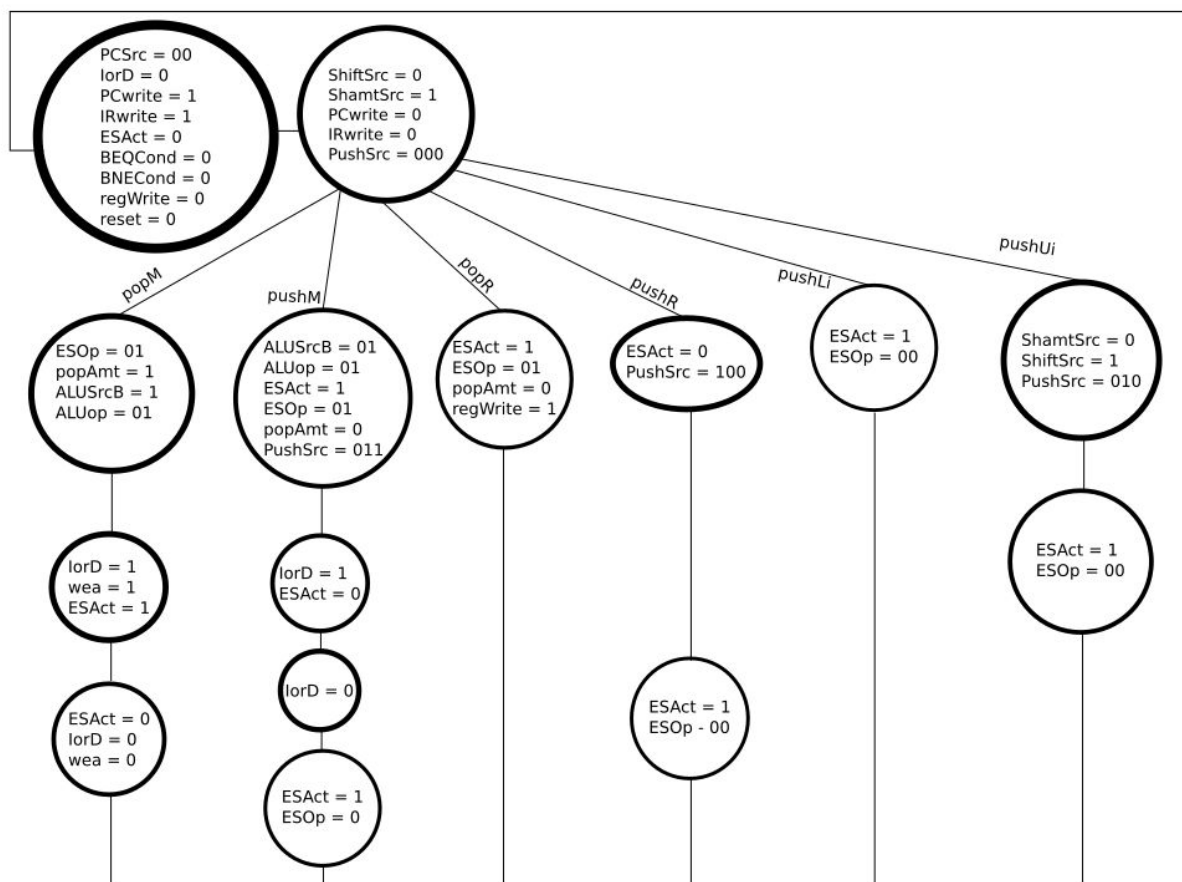
Step	slt	lsl/ lsr	flip	dup	pushUI/LI
------	-----	----------	------	-----	-----------

	newPC = PC + 2 PC = newPC Inst = Mem[PC]				
	A = ES[Top] B = ES[Top + 2]				
	ALUOut = A < B ES(pop 1)	ShiftOut = A shifted by Shamt in appropriate direction	Push A Push B	dupAMT = inst[1-0] If dupAMT == 0: ES[Top+2] = ES[Top] Top = Top+2 Done  If dupAMT == 1: ES[Top+2] = ES[Top-2] ES[Top+4] = ES[Top] Top = Top+4  If dupAMT==2: ES[Top+2] = ES[Top-4] ES[Top+4] = ES[Top-2] ES[Top+6] = ES[Top] Top = Top + 6  Else: ES[Top+2] = ES[Top - 6] ES[Top+4] = ES[Top-4] ES[Top+6] = ES[Top-2] ES[Top+8] = ES[Top] Top = Top + 8	UI: ShiftOut = SE(inst[11-4]) << 8  LI: push ZE(inst[11-4])
	skip	ES(push ShiftOut)			ES(push ShiftOut)
	ES(push ALUout)				

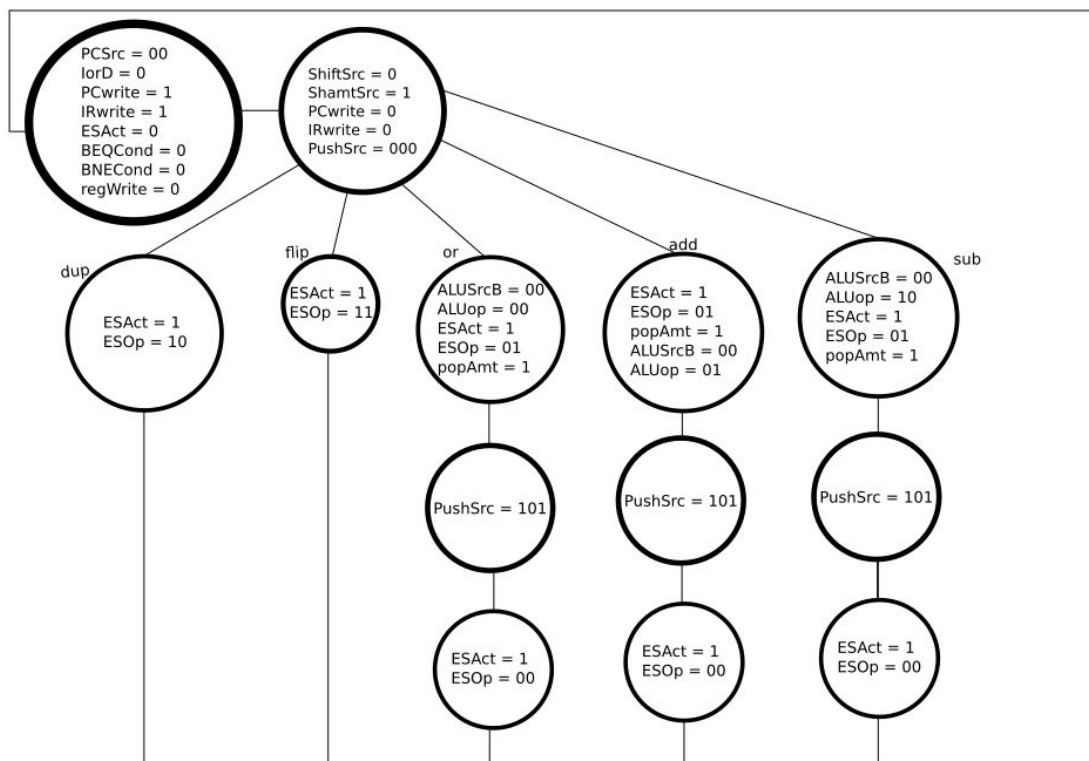


## Appendix J: State Transition Diagram for Multicycle

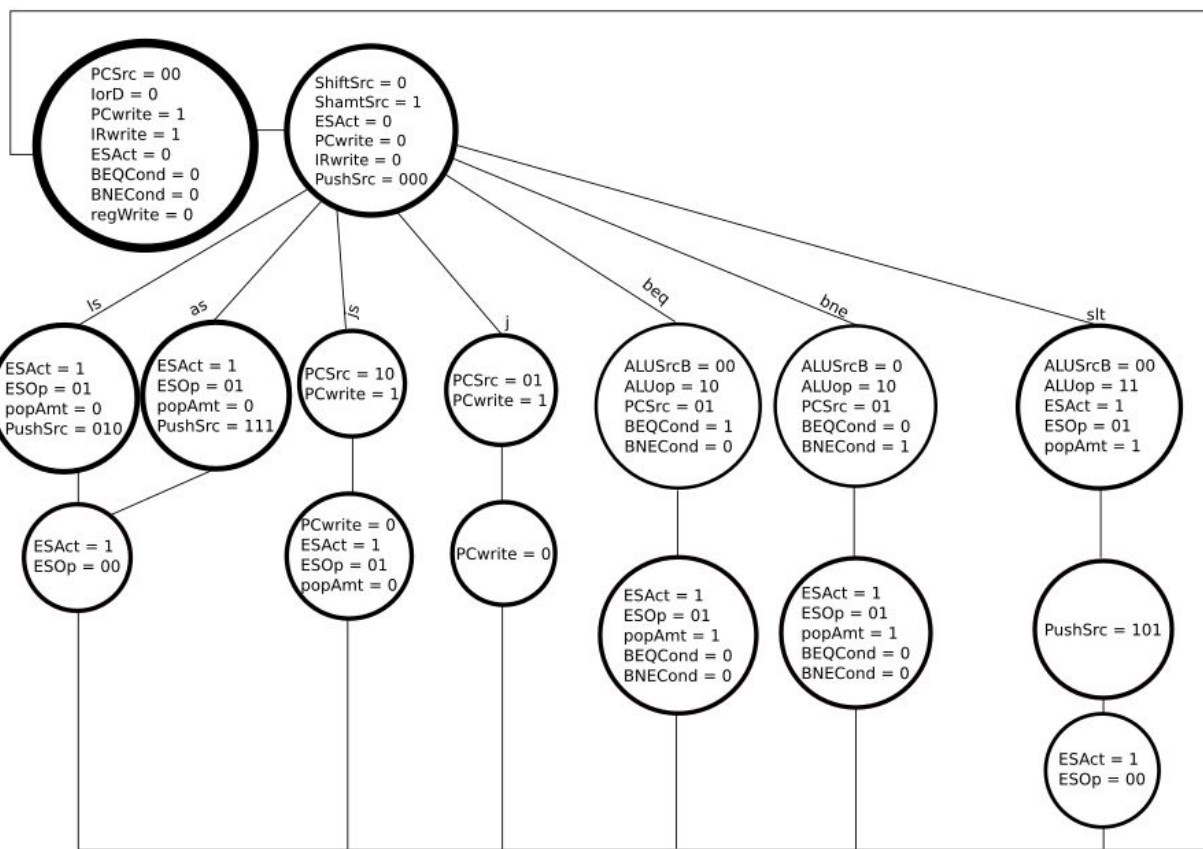
## Push and Pop



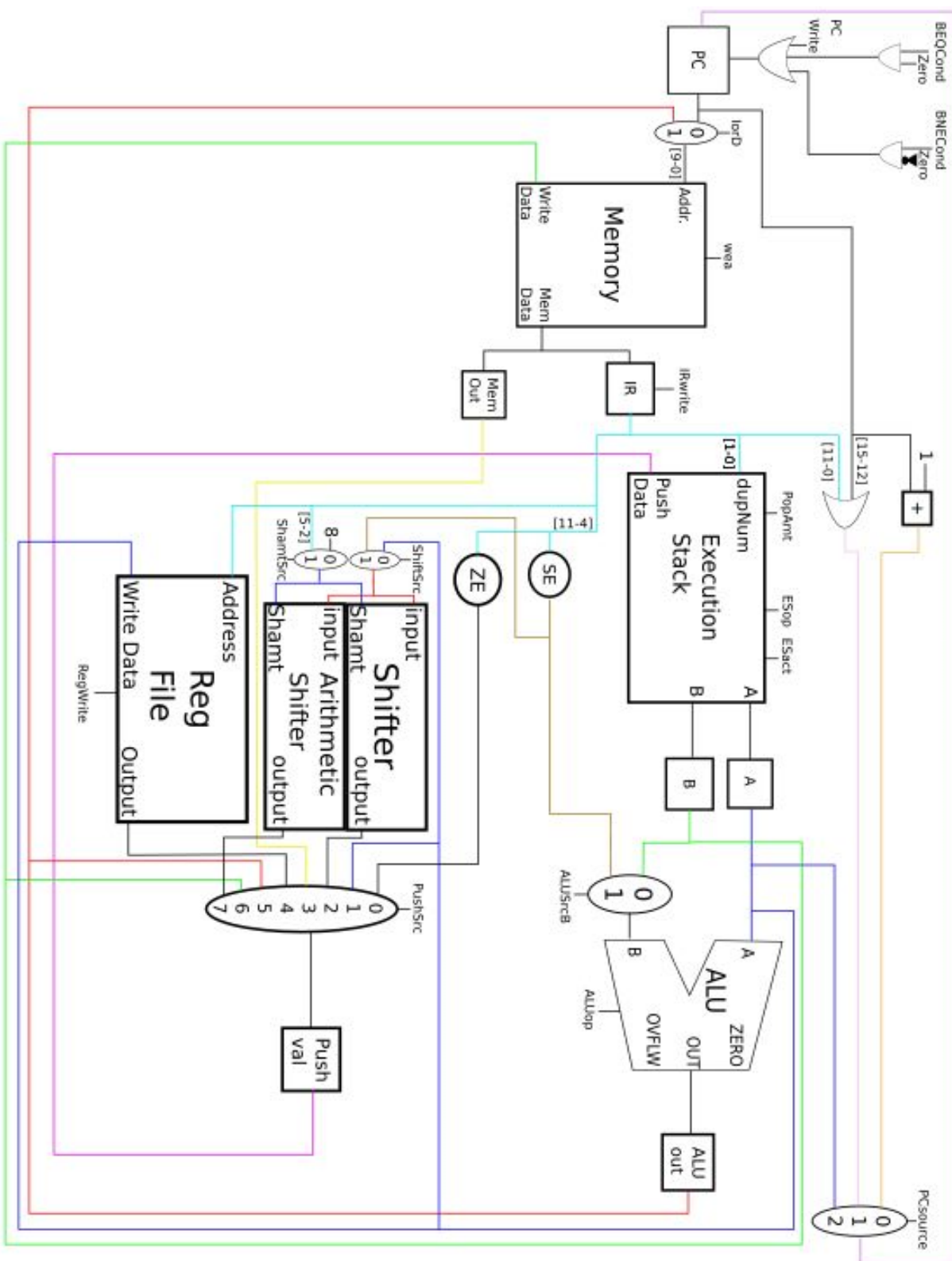
## Arithmetic Instructions



## SLT and Shifting



## Appendix K: Datapath



## Appendix L: Control Symbol Tables

PCSrc	
Value	Source
00	PC + 1
01	PC[15-12]   IR[11-0]
10	A

ALUSrcB	
Value	Source
0	B
1	SE(inst[11-0])

ALUOp	
Value	Operation
00	Or
01	Add
10	Sub
11	SLT

<b>ESOp</b>	
<b>Value</b>	<b>Operation</b>
00	push
01	pop (popNum)
10	dup (dupNum)
11	flip

<b>PushSRC</b>	
<b>Value</b>	<b>Source</b>
000	ZE(inst[11-4])
001	A
010	ShiftOut
011	MemOut
100	RegFile
101	ALUOut
110	B
111	ArithmeticShiftOut

<b>ShamtSrc</b>	
<b>Value</b>	<b>Source</b>
0	8
1	inst[6-2]

ShiftSrc	
Value	Source
0	A
1	SE(inst[11-4])

IorD	
Value	Source
0	PC[9-0]
1	ALUout[9-0]

## Appendix M: RelPrime Code

<u>Address/Label</u>	<u>Assembly</u>	<u>Machine</u>	<u>Comments</u>
0x2000	pushLi 0xb0	0010101100000000	
0x2002	pushUi 0x13	0011000100110000	
0x2004	or	0110000000000000	Initialize stack to 5040
0x2006	j REL_PRIME	1110110000000110	#call rel_prime
0x2008 / RETURN_TO_MAIN	popR \$v0	0001000000000100	
0x200A	j DONE	1110110001001111	
0x200C / REL_PRIME	pushLi 2	0010000000100000	
0x200E	dup 1	0100000000000001	
0x2010	pushR \$sp	0000000000000101	
0x2012	popM -2	0001111111100000	
0x2014	pushR \$sp	0000000000000101	
0x2016	popM 0	0001000000000000	
0x2018 / WHILE_REL_PRIME	pushLi 4	0010000001000000	
0x201A	pushR \$sp	0000000000000101	
0x201C	sub	1000000000000000	
0x201E	popR \$sp	0001000000000101	
0x2020	j GCD	1110110000100101	
0x2022 / BACK_FROM_GCD	pushLi 4	0010000001000000	
0x2024	pushR \$sp	0000000000000101	
0x2026	add	0111000000000000	
0x2028	popR \$sp	0001000000000101	

0x202A	pushLi 1	0010000000010000	
0x202C	beq RETURN_M	1100110000100010	
0x202E	pushR \$sp	0000000000000101	
0x2030	pushM -2	0000111111100000	
0x2032	pushLi 1	0010000000010000	
0x2034	add	0111000000000000	
0x2036	dup 0	0100000000000000	
0x2038	pushR \$sp	0000000000000101	
0x203A	popM -2	0001111111100000	
0x203C	pushR \$sp	0000000000000101	
0x203E	pushM 0	0000000000000000	
0x2040	flip	0101000000000000	
0x2042	j WHILE_REL_PRIME	1110110000001100	
0x2044 / RETURN_M	pushR \$sp	0000000000000101	
0x2046	pushM -2	0000111111100000	
0x2048	j RETURN_TO_MAIN	1110110000000100	
0x204A <sup>1</sup> / GCD	Dup 1	0100000000000001	Assumes b then
0x204C	pushR \$sp	0000000000000101	Then a in stack
0x204E	popM -2	0001111111100000	From top to bottom
0x2050	pushR \$sp	0000000000000101	Save a and b to 0
0x2052	popM 0	0001000000000000	And -2 from sp
0x2054	flip	0101000000000000	respectively
0x2056	dup 0	0100000000000000	

---

<sup>1</sup> This is the last instruction in the relPrime program



0x2058	pushR \$0	0000000000000111	
0x205A	beq RETURN_B	1100110001001001	If b==0 return b
0x205C / WHILE	dup 1	0100000000000001	Assumes a then b
0x205E	flip	0101000000000000	
0x2060	pushR \$0	0000000000000111	
0x2062	beq RETURN_A	1100110001001011	If b == 0, while loop
0x2064	popR \$0	0001000000000111	done
0x2066	dup 1	0100000000000001	
0x2068	flip	0101000000000000	
0x206A	slt	1011000000000000	
0x206C	pushLi 1	0010000000010000	
0x206E	bne B_MINUS_A	1101110001000000	If (b<a) we do a = a-b
0x2070	sub	1000000000000000	Else, B_MINUS_A
0x2072	pushR \$sp	0000000000000101	
0x2074	popM 0	0001000000000000	
0x2076	pushR \$sp	0000000000000101	
0x2078	pushM -2	0000111111100000	Replace old a value with
0x207A	pushR \$sp	0000000000000101	The new one
0x207C	pushM 0	0000000000000000	
0x207E	j WHILE	1110110000101110	Go back to while loop
0x2080 / B_MINUS_A	flip	0101000000000000	
0x2082	sub	1000000000000000	
0x2084	pushR \$sp	0000000000000101	Replaces the old b value
0x2086	popM -2	0001111111100000	With the new one
0x2088	pushR \$sp	0000000000000101	

0x208A	pushM -2	0000111111100000	
0x208C	pushR \$sp	0000000000000101	
0x208E	pushM 0	0000000000000000	
0x2090	j WHILE	1110110000101110	Go back to while loop
0x2092 / RETURN_B	popR \$0	0001000000000111	Assumes a, b so we pop
0x2094	j BACK_FROM_GCD	1110110000010001	A off
0x2096 / RETURN_A	popR \$0	0001000000000111	Assumes a, a, b so we
0x2098	flip	0101000000000000	Pop, then flip then pop
0x209A	popR \$0	0001000000000111	So we just have a
0x209C / DONE	j BACK_FROM_GCD	1110110000010001	
0x209E	j DONE	1110110001001111	End of program

## Appendix N: Common Code Snippets

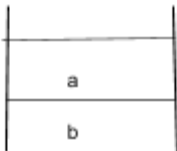
**1. Conditional Statements:***a. Less than*

```

C code:
if(a < b){
    a = a + b
}
else{
    b = a + b
}

```

ASSUMPTION:  
Initial stack before if statement:

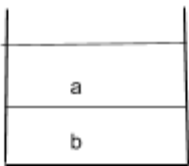


Address	Assembly	Machine Code
0x2000	PushR \$sp	0b 0000 0000 0000 0100
0x2002	pushM 0	0b 0000 0000 0000 0000
0x2004	PushR \$sp	0b 0000 0000 0000 0100
0x2006	pushM -2	0b 0000 1111 1110 0000
0x2008	add	0b 0111 0000 0000 0000
0x200A	PushR \$sp	0b 0000 0000 0000 0100
0x200C	pushM 0	0b 0000 0000 0000 0000
0x200E	PushR \$sp	0b 0000 0000 0000 0100
0x2010	pushM -2	0b 0000 1111 1110 0000
0x2012	slt	0b 1011 0000 0000 0000
0x2014	pushR \$zero	0b 0000 0000 0000 0111
0x2016	Bne DOIT	0b 1101 0000 0011 0000
0x2018	PushR \$sp	0b 0000 0000 0000 0100
0x201A	popM -2	0b 0001 1111 1110 0000
0x2030 / DOIT	PushR \$sp	0b 0000 0000 0000 0100
0x2032	popM 0	0b 0001 0000 0000 0000

b. *Greater than or equal*

```
C code:
if(a >= b){
    a = a + b
}
else{
    b = a + b
}
```

ASSUMPTION;  
Initial stack before if statement:

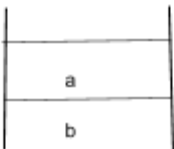


Address	Assembly	Machine
<b>Items up to the slt are the same as in the less than code</b>		
0x2012	slt	0b 1011 0000 0000 0000
0x2014	pushR \$zero	0b 0000 0000 0000 0111
0x2016	Beq DOIT	0b 1100 0000 0011 0000
<b>Items in the DOIT section are the same as in the less than code</b>		

c. *Less than or equal*

```
C code:
if(a <= b){
    a = a + b
}
else{
    b = a + b
}
```

ASSUMPTION;  
Initial stack before if statement:



Address	Assembly	Machine
<b>Items up to the slt are the same as in the less than code</b>		
0x2010	flip	0b 0101 0000 0000 0000
0x2012	slt	0b 1011 0000 0000 0000
0x2014	pushR \$zero	0b 0000 0000 0000 0111
0x2016	Beq DOIT	0b 1100 0000 0011 0000
<b>Items in the DOIT section are the same as in the less than code</b>		

*d. Greater than*

```
C code:
if(a > b){
    a = a + b
}
else{
    b = a + b
}
```

ASSUMPTION;  
Initial stack before if statement:

a
b

Address	Assembly	Machine
<b>Items up to the slt are the same as in the less than code</b>		
0x2010	flip	0b 0101 0000 0000 0000
0x2012	slt	0b 1011 0000 0000 0000
0x2014	pushR \$zero	0b 0000 0000 0000 0111
0x2016	Bne DOIT	0b 1100 0000 0011 0000
<b>Items in the DOIT section are the same as in the less than code</b>		

## 2. Looping

### a. While loop

C code:  
a = 0;  
b = 0;  
while (a < 10){  
    b = b + a;  
    a++;  
}

ASSUMPTION;  
Initial stack looks like:



Address	Assembly	Machine Code
0x2000	pushR \$zero	0b 0000 0000 0000 0111
0x2002	pushR \$sp	0b 0000 0000 0000 0100
0x2004	dup 1	0b 0100 0000 0000 0001
0x2006	popM 0	0b 0001 0000 0000 0000
0x2008	popM -2	0b 0001 1111 1110 0000
0x200A / LOOP	pushR \$zero	0b 0000 0000 0000 0111
0x200C	pushR \$sp	0b 0000 0000 0000 0100
0x200E	pushM 0	0b 0000 0000 0000 0000
0x2010	pushLi 10	0b 0010 0000 1010 0000
0x2012	slt	0b 1011 0000 0000 0000
0x2014	beq DONE	0b 1100 0000 0011 0000
0x2016	pushR \$sp	0b 0000 0000 0000 0100
0x2018	pushM 0	0b 0000 0000 0000 0000
0x201A	dup 0	0b 0100 0000 0000 0000
0x201C	pushR \$sp	0b 0000 0000 0000 0100
0x201E	pushM -2	0b 0000 1111 1110 0000
0x2020	add	0b 0111 0000 0000 0000
0x2022	pushR \$sp	0b 0000 0000 0000 0100
0x2024	popM -2	0b 0001 1111 1110 0000

0x2026	pushLi 1	0b 0010 0000 0001 0000
0x2028	add	0b 0111 0000 0000 0000
0x202A	pushR \$sp	0b 0000 0000 0000 0100
0x202C	popM 0	0b 0001 0000 0000 0000
0x202E	j LOOP	0b 1110 0000 0000 1010
0x2030 / DONE	OTHER CODE	

*b. For loop*

```
C code:
int b = 0;
for(int a = 0; a<10, a++){
    b = b + a;
}
```

ASSUMPTION;  
Initial stack looks like:



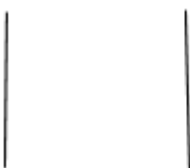
The machine and assembly code is the same as the code for the while loop.

### 3. Procedure Calling

#### a. Caller to Callee

C code:  
(\*some code\*)  
a = getGCD(a,b);

ASSUMPTION;  
Initial stack looks like:



Address	Assembly	Machine
0x200C	pushLI RETURN	0b 0010 0010 0110 0000
0x200E	pushUI RETURN	0b 0011 0010 0000 0000
0x2010	pushR \$sp	0b 0000 0000 0000 0100
0x2012	pushM 0	0b 0000 0000 0000 0000
0x2014	pushR \$sp	0b 0000 0000 0000 0100
0x2016	pushM -2	0b 0000 1111 1110 0000
0x2018	pushLi 6	0b 0010 0000 0110 0000
0x201A	pushR \$sp	0b 0000 0000 0000 0100
0x201C	sub	0b 1000 0000 0000 0000
0x201E	popR \$sp	0b 0001 0000 0000 0100
0x2020	j GCD	0b 1110 0000 0100 1100
0x2022	pushLI 0x06	0b 0010 0000 0110 0000
0x2024	pushR \$sp	0b 0000 0000 0000 0100
0x2026 / RETURN	add	0b 0111 0000 0000 0000
0x2028	popR \$sp	0b 0001 0000 0000 0100
0x202A	pushR \$v0	0b 0000 0000 0000 0101
0x202C	pushR \$sp	0b 0000 0000 0000 0100
0x202E	pushM -4	0b 0000 1111 1100 0000
0x2030	bne DONE1	0b 1101 0000 0100 0010



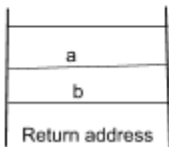
## b. Callee to Caller

```

C code:
int getGCD(int a, int b){
    ...
    ...
    return a;
}

```

ASSUMPTION;  
Initial stack looks like:



Address	Assembly	Machine
0x2040	pushR \$sp	0b 0000 0000 0000 0100
0x2042	popM 0	0b 0001 0000 0000 0000
0x2044	pushR \$sp	0b 0000 0000 0000 0100
0x2046	popM -2	0b 0001 1111 1110 0000
	OTHER CODE	
0x2060	pushR \$sp	0b 0000 0000 0000 0100
0x2062	pushM 0	0b 0000 0000 0000 0000
0x2064	pushR \$V0	0b 0000 0000 0000 0111
0x2066	popM 0	0b 0001 0000 0000 0000
0x2068	js	0b 1111 0000 0000 0000

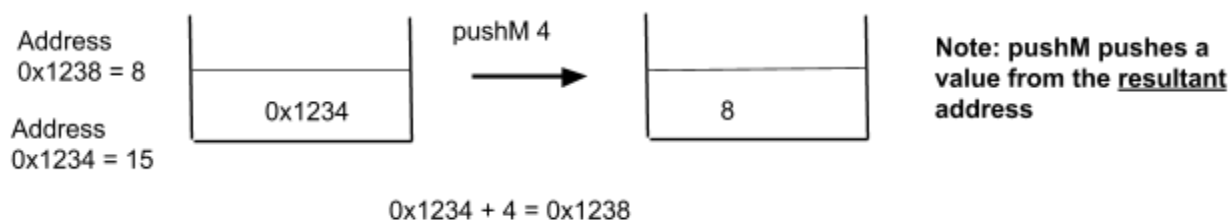
## Appendix O: Instruction Description and Semantics

1. **pushM** takes the 16-bit address from the top of the stack, adds an immediate to that address, and returns the value that is stored in the altered address

ISA: C-type

Example: **pushM 2**

Visualization of the stack

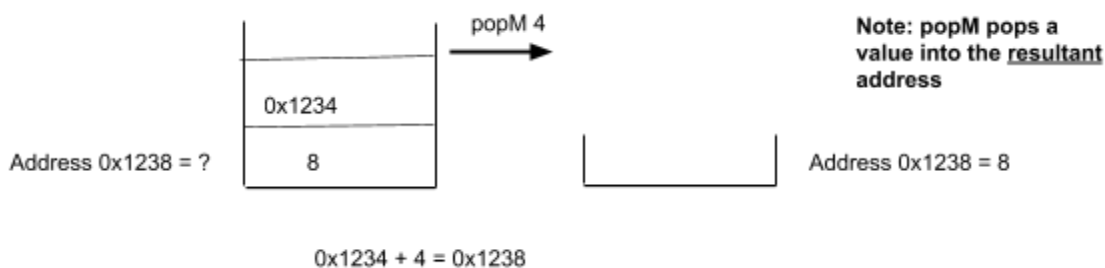


2. **popM** takes the 16-bit address that is on the top of the stack, adds an immediate to the address, and stores the value below that into the resultant address

ISA: C-type

Example: **popM 4**

Visualization of the stack

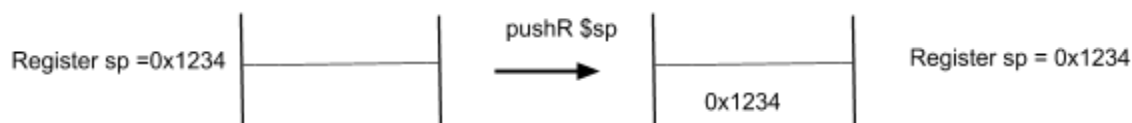


3. **pushR** pushes a 16 bit value from a specified register onto the stack.

ISA: C-type

Example: **pushR \$sp**

Visualization of the stack

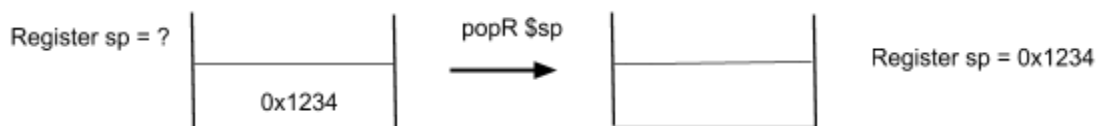


4. **popR** stores a 16 bit value from the top of the stack into the specified register. The value is then popped off the stack.

ISA: C-type

Example: **popR \$sp**

Visualization of the stack

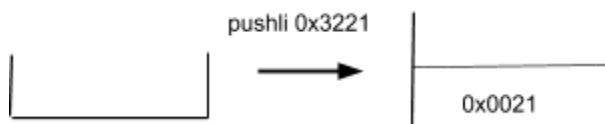


5. **pushli** takes the lower 8 bits of a 16 bit immediate, zero extends it, and stores it on the top of the stack

ISA: C-type

Example: **pushli 0x3221**

Visualization of the stack

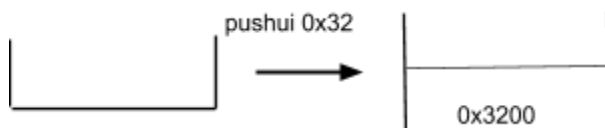


6. **pushui** takes the upper 8 bits of a 16 bit immediate, zero extends it, and stores it on the top of the stack.

ISA: C-type

Example: **pushui 0x32**

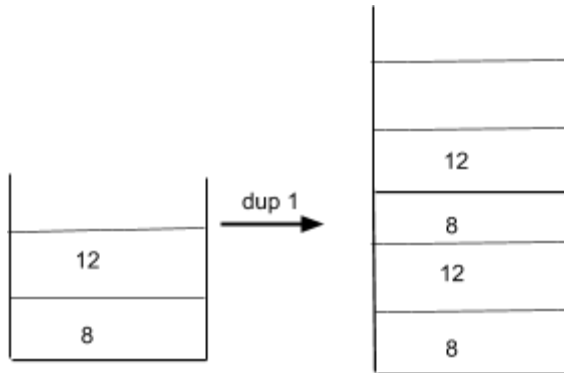
Visualization of the stack



7. **dup** looks at the specified amount of data from the top of the stack, copies the data, and pushes it on to the top of the stack

ISA: A-type

Example: **dup 1**

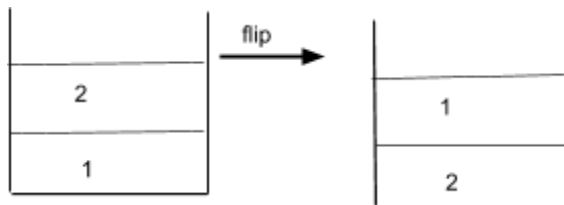


8. **flip** takes the two topmost values in the stacks and reverses their order on the stack.

ISA: A-type

Example: **flip**

Visualization of stack

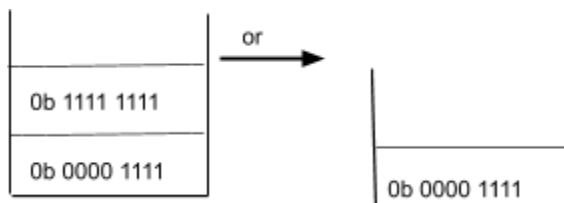


9. **or** looks at the two topmost values of the stack and performs the bitwise 'or' operation. The result is stored at the top of the stack.

ISA: A-type

Example: **or**

Visualization of stack

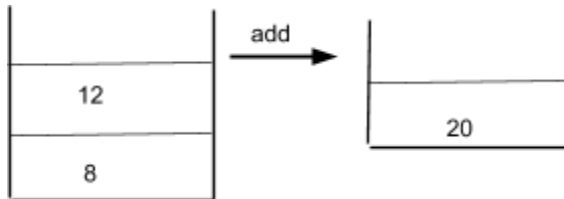


- 10. add** takes the two values stored at the top of the stack, adds the values, and then stores the result of add in place of the two parameters

ISA: A-type

Example: **add**

Visualization of the stack

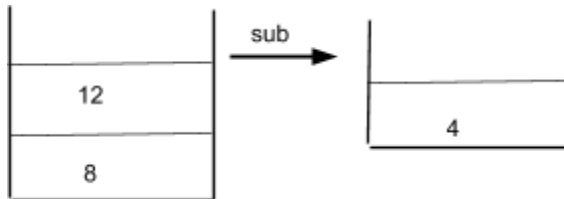


- 11. sub** takes the two values stored at the top of the stack, subtracts the values, and then stores the result of sub at the top of the stack

ISA: A-type

Example: **sub**

Visualization of the stack

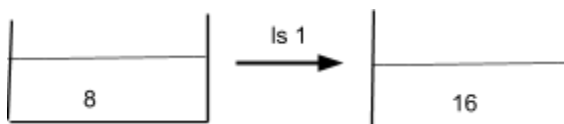


- 12. ls** shifts the value at the top of the stack, logically shifts by the shamt (signed), and replaces the value it operated it on with its result

ISA: A-type

Example: **ls**

Visualization of the stack

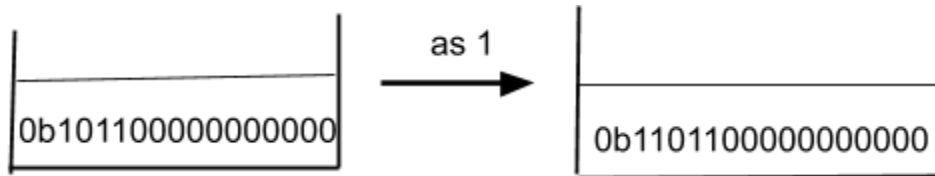


- 13. as** shifts the value at the top of the stack, arithmetically shifts it by the shamt (signed), and replaces the value it operated on with its result. Maintains the sign of the original value.

ISA: A-type

Example: **as**

Visualization of the stack



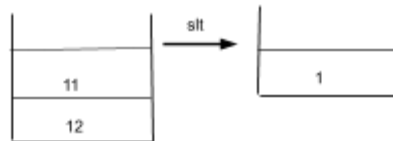
- 14. slt** compares the two top-most values of the stack. If the top value of the stack is less than the second value, then return 1. Otherwise, return 0.

ISA: A-type

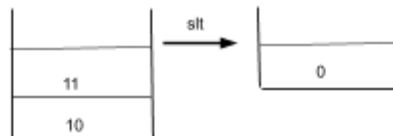
Example: **slt**

Visualization of the stack

Case 1:  
11 is less than  
12



Case 2:  
11 is **not** less  
than 10

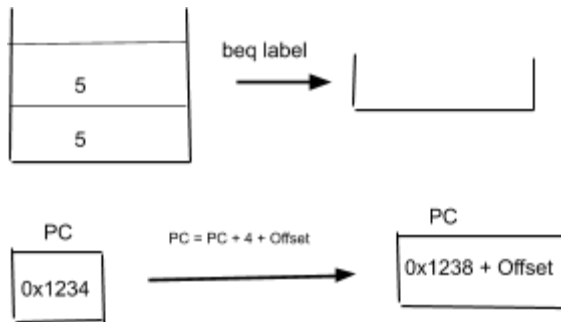


- 15. beq** compares the two top-most values of the stack. If the two values are equal, then the program execution goes to the address referenced by its label.

ISA: B-type

Example: **beq LABEL**

Visualization of Stack

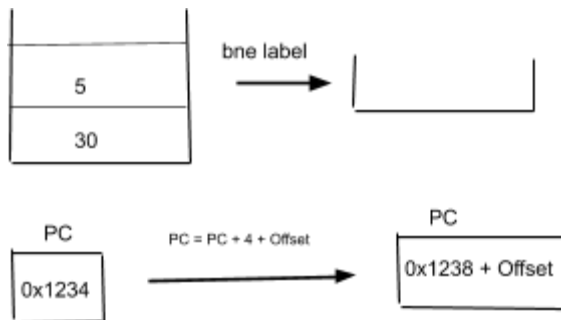


- 16. bne** compares the values compares the top-most values of the stack. If the two values are **not** equal, then the program execution goes to the address referenced by its label.

ISA: B-type

Example: **bne LABEL**

Visualization of stack

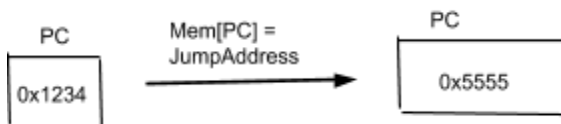


- 17. j** causes the memory execution to go to the specified location in memory.

ISA: B-type

Example: **j 0x5555**

Visualization of stack

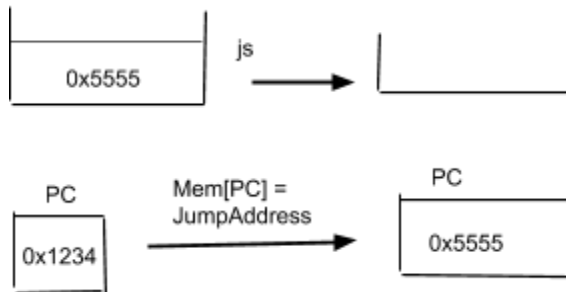


- 18. js** causes the memory execution to jump to the address that is on the top of the stack  
(assume that there is an address at the top of the stack)

ISA: B-type

Example: **js**

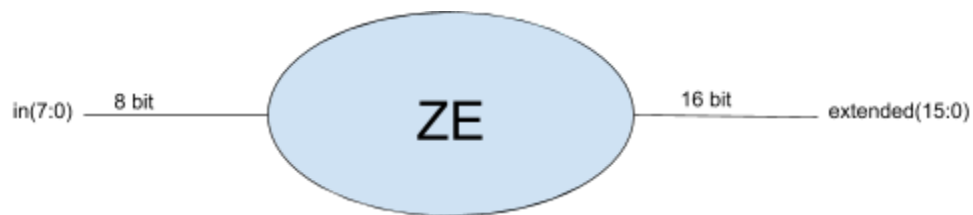
Visualization of stack



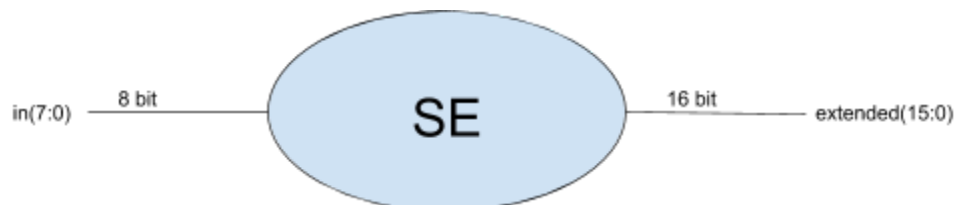


## Appendix P: System Components

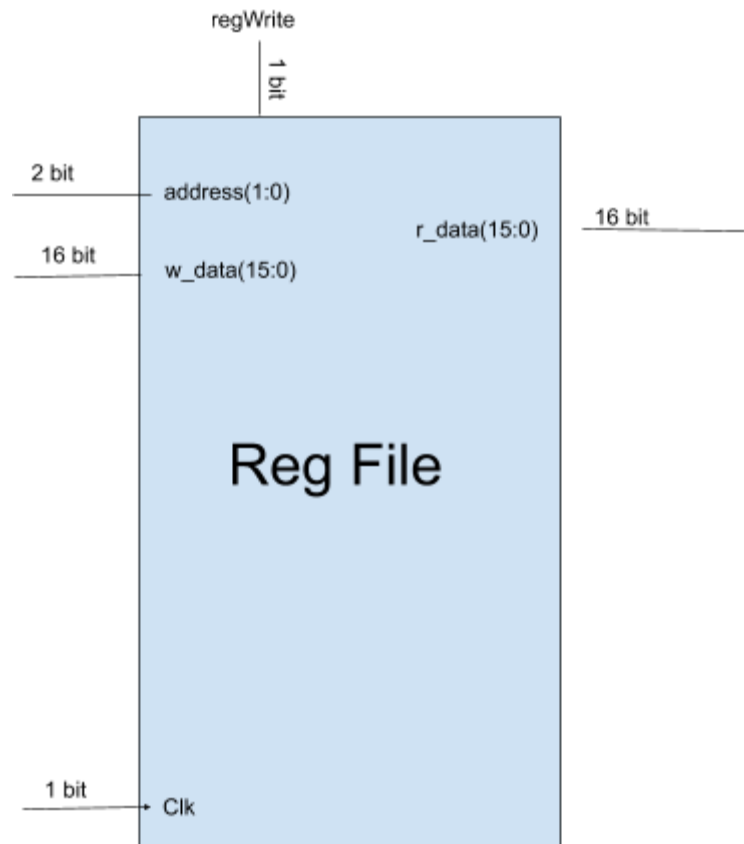
- **Zero Extender:** Zero extender takes an 8 bit immediate and extends it to a 16 bit immediate by filling in its 8 most significant bits with zeros
  - **Implements RTL Code:** ZE()
  - **Control signal:** None
  - **Input/Output signals:** in(7:0),extended(15:0)
  - **Hardware Implementation:** Assign the eight least significant bits of extended(15:0) to in(7:0). Assign the eight most significant bits to 0.
  - **Unit Testing:** This unit was tested by inputting various in(7:0) and checking that each extended(15:0) matched its input.



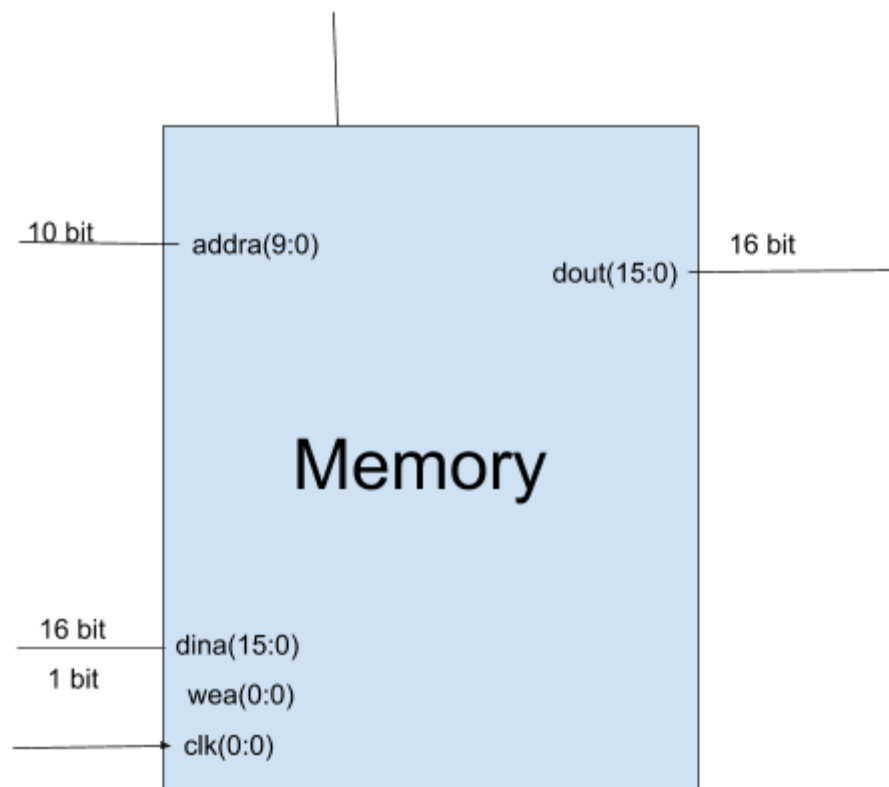
- **Sign Extender:** Sign extender takes an 8 bit immediate and extends it to a 16 bit immediate by filling in its 8 most significant bits with its 8th bit value.
  - **Implements RTL Code:** SE()
  - **Control signal:** None
  - **Input/Output signals:** in(7:0),extended(15:0)
  - **Hardware Implementation:** Assign the eight least significant bits of extended(15:0) to in(7:0). Assign the eight most significant bits to in(7).
  - **Unit Testing:** The test cases are as follows:
    - in(7:0) has a MSB of 0. We check that in(7:0) is equivalent to extended(15:0).
    - in(7:0) has a MSB of 1. We calculate the expected extended(15:0) and compare to the actual result.



- **Register File (4-bit):** The register file contains 4 registers that refer to commonly used registers such as \$V0 (return value), \$sp (stack pointer for memory), \$gp (global pointer), and \$zero (zero register) that allows for reading and writing.
  - **Implements RTL Code:** Reg[]
  - **Control signal:** MemToReg, regWrite
  - **Input/Output signals:** clk, address(1:0), w\_data(15:0), r\_data(15:0)
  - **Hardware Implementation:**
  - **UnitTesting:** The test cases are as follows:
    - Write to special registers v,sp, and gb and check that r\_data matches accordingly
    - Read z register and ensure that value is 0
    - Write to the z register and ensure that it always outputs 0



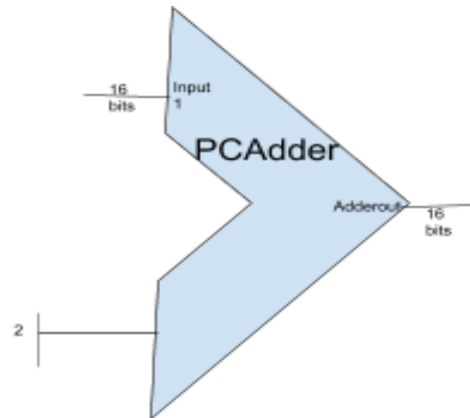
- **Memory:** allows information to be written and read within a range of addresses. See page 6 for how those addresses are allocated.
  - **Implements RTL Code:** Mem[]
  - **Control signal:** MemRead, MemWrite, inst, wea
  - **Input/Output signals:** addra(9:0), dina(15:0), dout(15:0), clk(0:0)
  - **Hardware Implementation:** Use IP(Core Generator and Architecture Wizard) to generate a v6.3 block memory component with a write width of 16, write depth of 1024, and read depth of 1024. The operating mode is write first and is always enabled. A small\_memory.coe file is available with arbitrary values stored.
  - **Unit Testing:** The test cases are as follows:
    - Read and checks various values from the memory file that were initialized in the .coe file (Reading without writing)
    - Write and checks various values to addresses that were not written to in the .coe file
    - Rewrites and checks the value of an address that was already written to in the .coe file



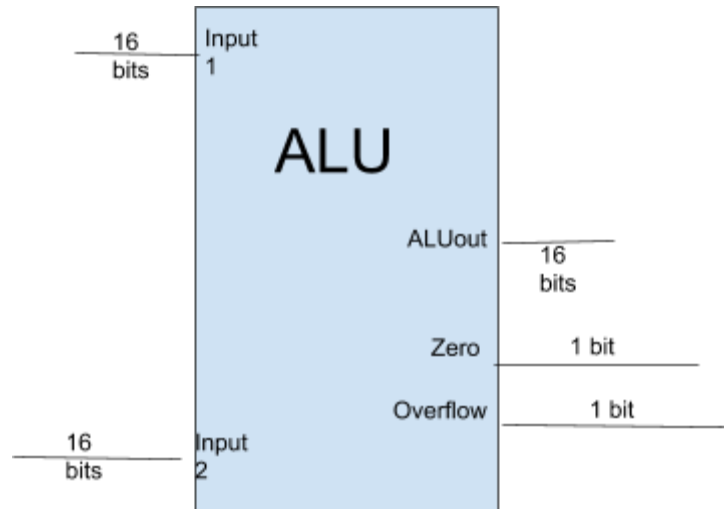
- **Register:** A component that stores the 16-bit value while another value is being processed on datapath
  - **Implements RTL Code:** PC, ALUout, A, B, C, D, E, IR, MDR
  - **Control signal:** PCWrite, PCWriteCond, PCsource, IRWrite
  - **Input/Output signals:** clk, regWrite, w\_data(15:0), r\_data(15:0)
  - **Hardware Implementation:** If regWrite is enabled, save value of w\_data to reg variable. If regWrite is disabled, store value of reg variable to r\_data. A register commits these actions on the negative clock edge.
  - **Unit Testing:** The test cases are as follows:
    - Write an initial value to a register and check r\_data
    - Rewrite a different value to the register and check r\_data to ensure that there was a change
    - Check r\_data again to ensure that the register holds data over time
    - Check that a value larger than 16-bits will result in a fail



- **PCAdder:** The ALU component takes in two 16-bit inputs (from PC) and adds 2 for proceeding to the next instruction.
  - **Implements RTL Code:** + 2
  - **Control signal:**
    - N/A
  - **Input/Output signals:** Input1 / AdderOut
  - **Hardware Implementation:** This adder is simply used to add '2' to the PC so that we can increment to the next instruction. This is done by taking in the current PC value and adding 2 to it. The value 2 is hardwired in while PC is the input value. The output is our new PC value.
  - **Unit Testing:** This unit was tested by putting in a 16 bit value and adding 2 to it. The output was compared to the expected value.

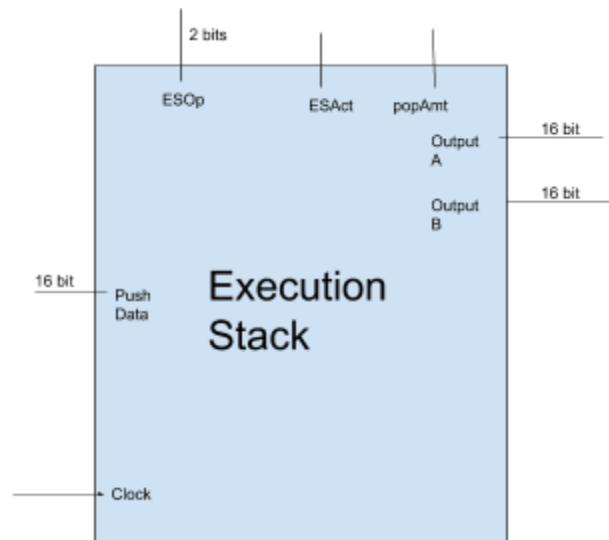


- **ALU:** The ALU component takes in two 16-bit inputs, and arithmetically computes a single 16-bit result from using functions: “add”, “subtract”, “or”, and “zero detect”.
  - **Implements RTL Code:** +, -, ||, isZero()
  - **Control signal:**
    - ALUsrcA - 2 bits
    - ALUsrcB - 2 bits
    - ALUOp - 2 bits
  - **Input/Output signals:** Input1, Input2, Op/ Zero, Overflow, ALUout
  - **Hardware Implementation:** The ALU has 3 inputs: a, b, and op. The inputs a and b are the values that we would like to operate on while op is the value that determines what operation we shall perform. Once the operation is performed, the value is placed into a 17 bit register. The most significant bit of which is the overflow detector. The remaining bits are the result. The zero detector takes this result and or’s each bit to see if there are any 1’s in the result. If there is, the zero value is 0, and if not it is 1.
  - **Unit Testing:** This unit was tested by selecting two random 16 bit inputs and performing each of the operations that the ALU could perform on these inputs. The output was compared against the expected value. As for the zero and overflow values, they had special inputs to test which would guarantee that they output 1. Subtraction of equal numbers provided the test for the zero output (as well as having b be smaller than a and performing a set less than operation which outputs 0 in that case). Overflow was tested via inputting two 16 bit values which when added would result in a 17 bit output.

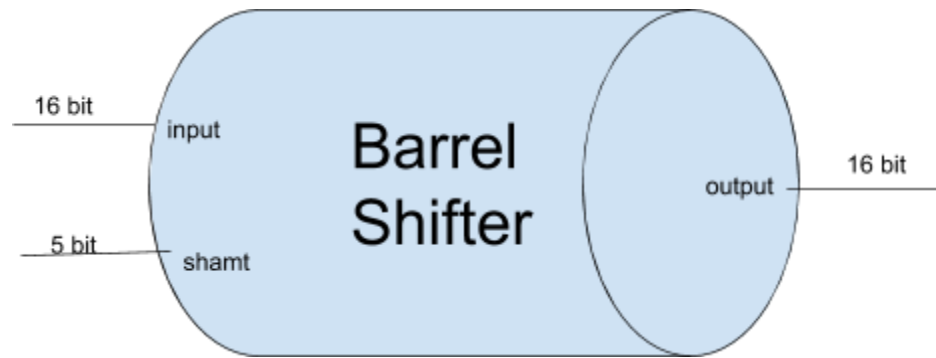


- **Execution Stack** is a component that serves as workspace with temporary data manipulation. It is capable of popping off four 16-bit values at once, duplicating 4 16-bit values, or pushing a single 16-bit value onto the stack.
  - **Implements RTL Code:** Pop, Push, Top, Flip
  - **Control signal:**
    - ESOp - 2 bits, ESAct - 1 bit
  - **Input/Output signals:** push, popNum, dupNum, flip / OutputA, OutputB, OutputC, Output
  - **Hardware Implementation:** The execution stack has the following inputs: 16-bit pushVal, 2-bit popNum and dupNum, 2-bit ESOp and a clock. There are 4 16-bit output registers, and there is an internal 32 register file that can contain 16-bit entries as well as an integer that references the Top of Stack (tos). The pushVal will be placed in reference to the tos, and then the tos will be updated. The popNum will determine how many 16-bit values are placed into the different outputs, and the tos will be updated according to said number. The dupNum will determine how many registers according to the tos will be duplicated, resulting in another update of the tos. There is also a flip function that will reverse the top two registers according to the tos (this does not update the tos). The function that will be executed is determined by the 2-bit ESOp, which is the controller for the execution stack.

- **Unit Testing:** The execution stack was tested through a series of conditions based upon the four operation. The test was based on the OutA value after a popNum=0 condition. First the stack was filled by a series of 33 pushes (This also tests Out of Bounds errors). Then the stack tried to duplicate for each of the four values of dupNum (Note that this was 8 different conditions to include Out of Bounds errors). The stack was then brought down to a total of 4 registers filled where the popNum conditions are tested with their respective Out of Bounds errors. Last, three registers are filled, and the flip operation is tested (The bottom of the stack should not have changed, but the top two should be flipped).



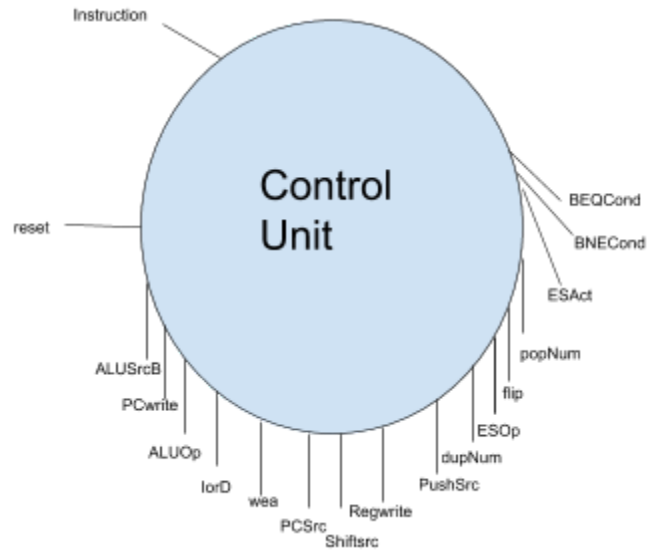
- **Barrel Shifter:** The Barrel Shifter takes in a 16-bit input, shifts the input by a given 4-bit shamt amount, that then produces a 16-bit output
  - **Implements RTL Code:** SL(), SR()
  - **Control signal:**
    - ShiftDirection - 1 bit
  - **Input/Output signals:** input, shamt / output
  - **Hardware Implementation:** The shifter takes in two items, the input value and the shift amount (shamt). The shamt is a signed 4 bit integer and so can be from -8 to 7. The positive values correspond to a left shift as that is considered to occur more often than a right shift. Negative values are a right shift. The output is a 16 bit integer which is the input shifted by the specified amount.
  - **Unit Testing:** This unit was tested by inputting a random 16 bit value and shifting by assorted positive and negative values. The output was checked to see if the 1 bits were in the correct place and that 0's had been shifted in.





- **Control Unit:** Provides the correct control signals to the appropriate components.
  - **Implements RTL Code:** N/A
  - **Control Signal Descriptions:**
    - popAmt
      - provides the correct number of pops to Execution Stack
    - ESOp
      - Tells the execution stack which operation to perform
    - ESAct
      - Tells the execution when to execute an operation like an enable bit
    - ALUSrcB
      - Specifies whether or not the second input to the ALU is extended or from a register
    - ALUOp
      - Specifies which operation the ALU performs
    - PCSrc
      - Specifies whether the PC is set to the next instruction address or a branched address
    - PushSrc
      - Specifies whether the Execution stack pushes a value from a register, memory, the ALU, the barrel shifter(shifters), or the zero extender
    - ShiftSrc
      - Specifies whether the Shifter shifts an immediate by 2 or by a different amount.
    - RegWrite
      - Tells the register file whether or not to write to one of its registers
    - IorD
      - Determines whether the address of the memory component is from the PC or the ALU
    - wea
      - Tells the memory whether or not to write to the address
    - Reset
      - Sets control unit to default values
    - PCwrite
      - Enables writing to PC when we DO NOT branch
    - BNECond
      - Enables writing to PC when we perform a bne instruction.
    - BEQCond
      - Enables writing to PC when we perform a beq instruction.

- **Output signals: all controls above**
- **Input signals: reset, opcode, funct, damp**
- **Unit Testing:** This unit was tested by manually setting the appropriate opcode, funct, and damp for all 18 instructions. The appropriate control signal values were then checked for correctness. Reset was tested by setting it to 1 and seeing if changing the other inputs produced a different result from expected.

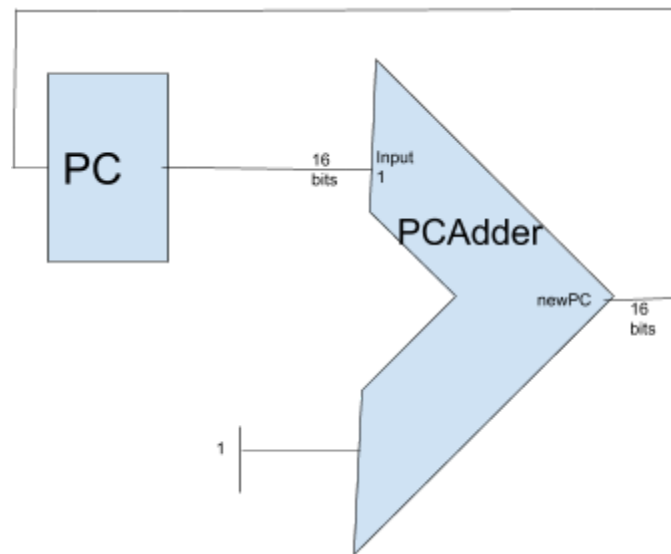


## Appendix Q: RTL Verification

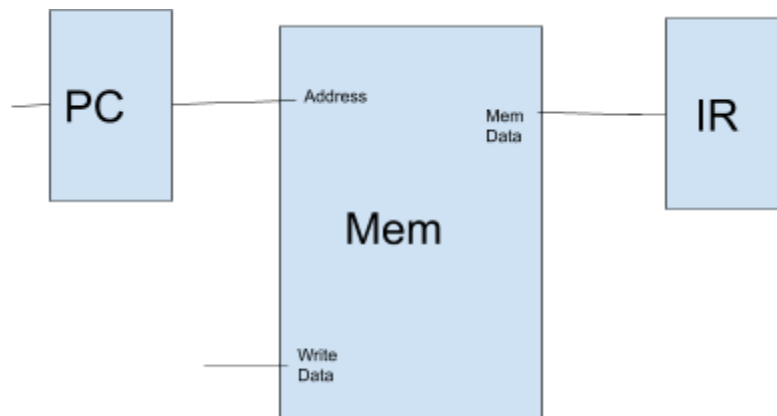
We verified our register transfer language by simulating the implementation of each instruction through the described above components. Below, we will show how we simulated add:

### Add

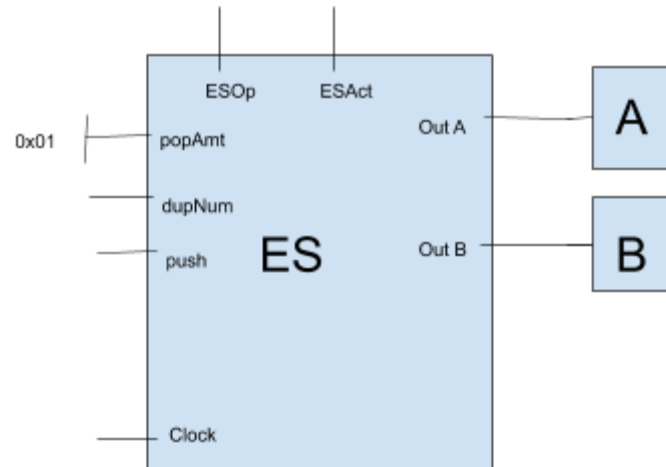
1.  $newPC = PC + 1$
2.  $PC = newPC$



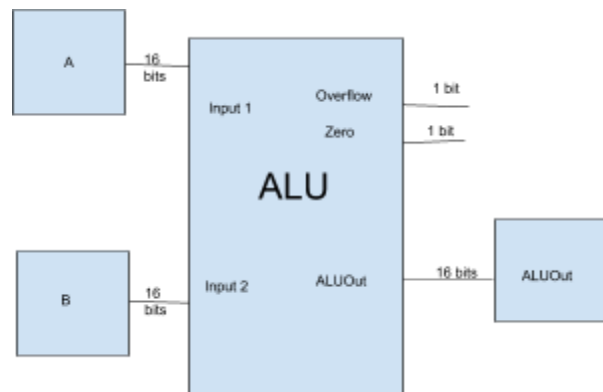
3.  $Inst = Mem[PC]$



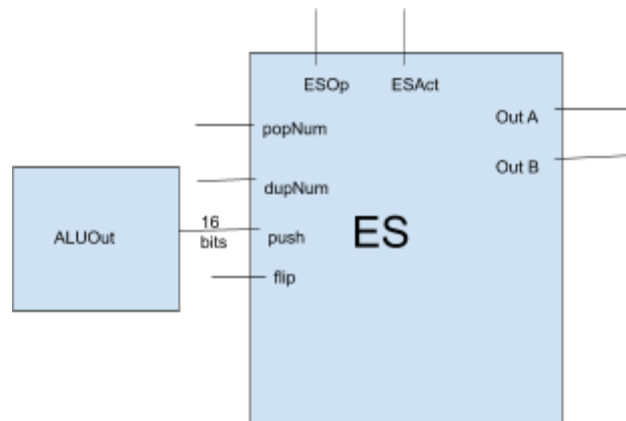
4.  $A = pop$
5.  $B = pop$



**6.  $ALUout = A+B$**



**7. *Push ALUout***



## Appendix R: Testing Methodology

We used to several different testing methodologies while testing the processor.

1. Component Testing:
  - a. For each component test, we decided to test all functions provided by the component and any edge cases presented by those functions.
2. Integration Testing:
  - a. We integrated our components together by sections of the overall datapath that will perform operations together. After combining them, we hard-coded inputs at the beginning of the section and checked the outputs to see that they were what we expected.
3. System Testing:
  - a. We plan to first completely test the datapath without the control unit attached and once the datapath successfully completes all types of instructions with inputted control signals, we will attach the control unit. We will then test the control unit with a single instruction, and then followed with a series of instructions.

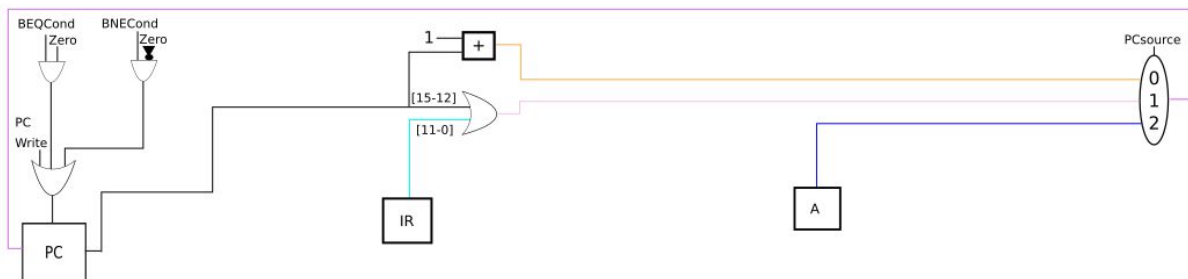
## Appendix S: Integration Plan

### 1. Subsystem level 1

#### a. PC\_subsys

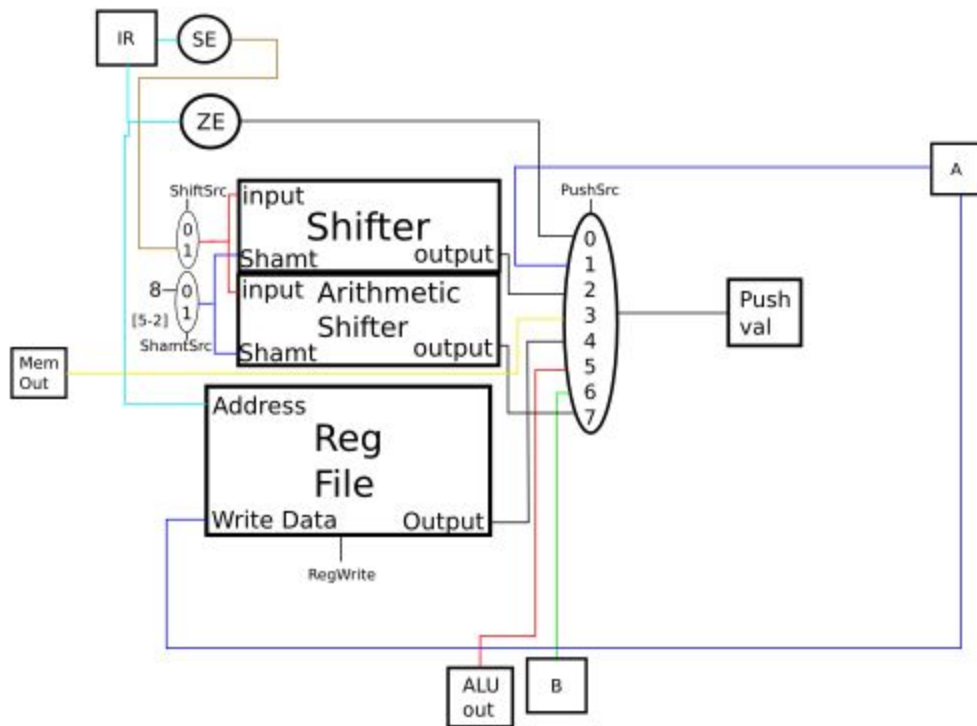
i. **Included components:** PC register, PCAdder

ii. **Testing plan:** Be able to increment PC's value by 1 using PCAdder. Check to make sure that the control signals for branching and normal operation work properly



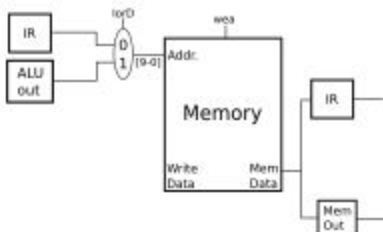
## b. Push\_subsys

- i. **Included components:** ShiftSrc mux, ShamtSrc mux, Shifter, Zero Extender, PushSrc mux, Reg File, PushVal register
- ii. **Testing plan:** Be able to feed unique values into shifter, zero extender, and reg file and have Push Val store the correct value based on value of control bit PushSrc.



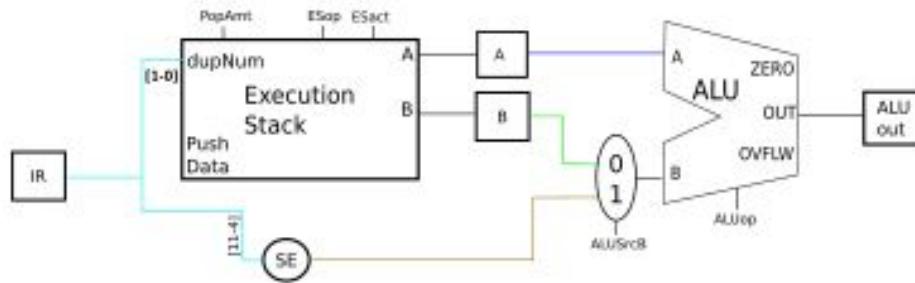
## c. Mem\_subsys

- i. **Included components:** Memory, IR register, Memout register, IorD mux
- ii. **Testing plan:** Be able to read data from memory based off of value of IorD control bit. Ensure that IR can be disabled and enabled using IRWrite control bit.



## d. ES\_subsys

- i. **Included components:** Execution stack, A register, B register, Sign Extender, ALUSrcB mux, ALU, ALUOut
- ii. **Testing plan:** Be able to feed contents of execution stack and sign extender into ALU and ensure that necessary results are outputted into ALUOut. Ensure that ALUSrcB mux switches inputs to ALU appropriately based off of ALUSrcB control bit.



2. Create datapath by connecting wires between the four subsystems.



## Appendix T: System Testing Plan

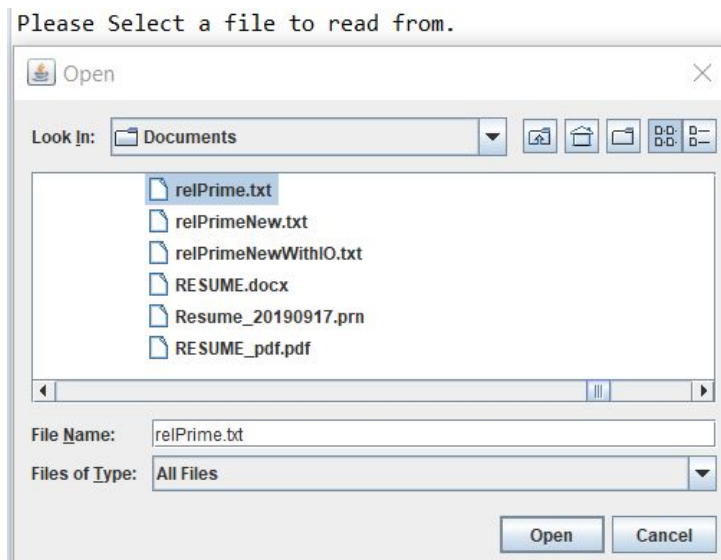
1. Complete a single instruction through datapath with manually set controls.
  - a. A type instruction testing
    - i. We will set all control values to perform one instance of the instruction  
Add. We will preload values into each of the necessary modules to do so.
      1. PC will be set to an address which contains an add instruction
      2. Memory will have an address with an add instruction
      3. The Execution Stack will contain the 2 values that we wish to add
      4. All control bits will be set manually
        - a. See multicycle diagrams
      5. We will then check the output of the ALU and the Execution Stack to see if the sum was pushed back onto the Stack
  - b. B type instruction Testing
    - i. We will set all control values to perform one instance of the instruction  
beq
      1. PC will be set to an address which contains a j instruction
      2. Memory will have an address with a j instruction
      3. All control values will be set manually
        - a. See multicycle diagrams
      4. We will then see if the PC has the correct value that we wished to  
Jump to
  - c. C type instruction testing
    - i. We will set all control values to perform one instance of the instruction  
pushLi
      1. PC will be set to an address which contains a pushR instruction
      2. Memory will have an address with a pushR instruction
      3. The register file will have a register with a known value (5) to read
      4. All control values will be set manually
        - a. See multicycle diagrams
      5. We will then check the execution stack to see if the value from the  
register file is in the stack
2. Complete a single instruction through datapath with connected control unit.
  - a. All processes, instructions, and result checking will be the same as in the manual portion but now the control unit will set the control bits.
3. Complete a series of instructions with connected control unit
  - a. LOOP: pushLi 6
  - b. pushUi 7
  - c. Or

- d. pushLi 8
- e. pushLi 10
- f. Add
- g. Beq LOOP

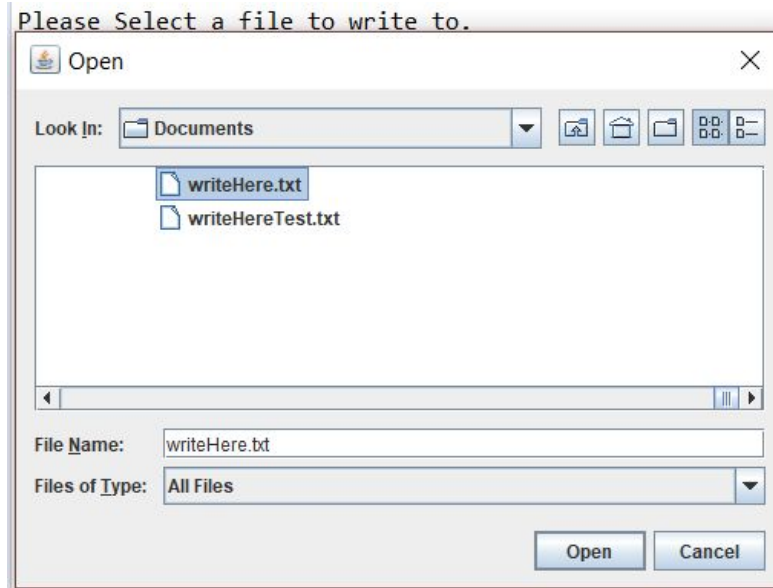
## Appendix U: Assembler Semantics and Details

- Instructions
  - In a text document, write in the assembly code you wish to run.
    - These are case sensitive
    - Immediates can be in hex, binary, or decimal
      - Hex is denoted 0x
      - Binary is denoted 0b
      - Items not denoted this way will be considered decimal
    - Instructions must be separated by lines
    - Labels must have instruction on the same line
      - i.e. DONE: j DONE
      - DONE: with no instruction will not work
    - There are a few pseudoinstructions
      - Push
        - Will push a 16 bit value to the stack
        - Push 0x13b0 will do:
          - pushLi 0xb0
          - pushUi 0x13
          - or
      - bge
        - Will branch if the top item on the stack is greater than or equal to the next thing in the stack.
        - bge LABEL will do:
          - slt
          - pushLi 1
          - bne LABEL
      - Ble
        - Will branch when the top item on the stack is less than or equal to the next thing on the stack
        - Ble LABEL will do:
          - Flip
          - Slt
          - pushLi 1

- bne
- bgt
  - Will branch when the top item on the stack is STRICTLY greater than the next item on the stack
  - bgt LABEL will do:
    - flip
    - slt
    - pushLi 1
    - beq LABEL
- blt
  - Will branch when the top item on the stack is STRICTLY less than the next item on the stack
  - blt LABEL will do:
    - slt
    - pushLi 1
    - beq LABEL
- Select the file you have written the code to



- Select the .coe file you wish to write the machine code to
  - THE FILE YOU WRITE TO WILL BE OVERWRITTEN



- Reload the .coe file in memory and run
- Notes
  - The java program will display:
    - The translated code (from pseudo instructions)
    - An array of address, instruction, immediate (if applicable)
    - The total output of the .coe file
      - This includes the radix setup and the machine code
    - This is so any outputs can be more easily debugged

## Appendix V: Processor Testbench Data

*Instructions:* 183774

*Cycle frequency:* 65.6 MHz

*Cycle time:* 15.23 ns


*Cycles:* 745206

*CPI:* 4.055

*Execution Time:* 11.35ms

*Size of Program:* 160 bytes

*Device Utilization Summary:*

Device Utilization Summary (estimated values)				
Logic Utilization	Used	Available	Utilization	
Number of Slices	4407	4656		94%
Number of Slice Flip Flops	774	9312		8%
Number of 4 input LUTs	8351	9312		89%
Number of bonded IOBs	77	232		33%
Number of BRAMs	1	20		5%
Number of GCLKs	1	24		4%

## Appendix W: Team Member Journals

### *William Dalby's Journal*

Friday 10/4/19

- Met for 1 hour about what type of ISA we wanted to use. We decided on a stack mixed with load store. We worked on the instruction types that we wanted and the instructions that we needed. Talked over the procedure calling convention and addressing modes.

Sunday 10/6/19

- Met for 2.5 hours on the code for relPrime. Much discussion about the procedure calling conventions and how deep the stack in use is. The layout and usage of the instructions were designed by the team. There was talk about addressing modes and how to handle essentially everything in the code.

Monday 10/7/19

- Met for 3 hours redesigning our language. As of this moment, I'm still not very sure I understand what's going on with our architecture. I try to ask questions but they end up not being very helpful or are obvious to my peers. It is slightly discouraging. I will have to spend time understanding what our architecture is trying to do on my own time or I feel I will be a bit behind. The deliverables are done at this point though. We didn't designate exactly what to do for the next milestone.

Tuesday 10/8/19

- Met for an hour cleaning up the document and our journals. Prepared for the meeting on Wednesday.

Wednesday 10/9/19

- Met for 2 hours correcting the issues with our document. We added an executive summary and title page. We started to designate where our memory was going to be so that we wouldn't have any variable addresses. We created a list of things that we needed to fix before starting the second milestone.

Sunday 10/13/19

- We met for 3 hours fixing all of the issues from milestone 1. This included making the memory stack and designating where things are allocated. We changed how we saved things to memory by utilizing a stack pointer. This changed our code substantially (knocking off about a page and a half of code) and it all had to be rewritten. All of the addresses for the machine code had to be changed and a number of opcodes were changed. In addition, we added a funct to our C-type ISA so that we could push and pop from both registers and memory separately. We started to add RTL for each of our operations. This was scratchwork and will need to be changed into the summary table later. We began to discuss what hardware we'd need but we definitely to look into that more.

Monday 10/14/19

- Met for 2 hours. The main things that were worked on were solidifying our RTL for all of our operations. We put all of our scratchwork into a summary chart that, frankly, looks terrible. This is not because the information is bad but because the formatting just doesn't look good. My bad. We made diagrams and input output descriptions for all of the hardware we think that we'll need. These look pretty decent (my good) and I feel that we're starting to actually have a project coming together. We solidified a couple of our ideas and how the programmer will need to use them. At this point I feel quite a bit better about the project. While there are certain aspects where I feel a little behind my partners at this point I am understanding our operations better and better. I feel that the level of abstraction is decreasing which is helping me quite a bit. We are still not designating what needs to be done for the next milestone ahead of time. I think that the reason for this is that we do all of our work together as a team. This may not work out in the long run but at the moment we're rolling with it.

Monday 10/21/19

- Met for 2 hours. Mainly worked on the three labs that are due on Friday. I realized that I have messed up the ALU lab quite a bit by trying to jump straight to the 4bit alu rather than doing it incrementally. Will need to fix that. I also implemented the shifter but then lost it after a GIT accident.

Tuesday 10/22/19

- Tori and I met for 3 hours and worked on fixing the issues from the last milestone and testing our components. I worked mainly on adding the machine code for our common operations but I also remade the shifter. I also made the ALU but I am worried that it's implementation may not be the way that is desired by the project. I simply used a 17 bit register to store the output and then its most significant bit was the overflow detector. Also, there is no implementation of an adder. I just used the built in verilog commands. I feel as though I'm going to have to change this at some point to more closely resemble the ALU from the lab. I have not yet finished the lab yet so I'm going to roll with the ALU as it stands.

Wednesday 10/23/19

- Met for 2 hours during lab. I updated more of our machine code for our common operations. I worked on drawing the datapath. This was more out of lack of anything else to do as I cannot claim to be the best artist so I personally apologize for the appearance of the spaghetti that is our data path. I am a bit confused as to how it's all going to work, however. How will operations that do multiple pushes and pops work? The main one that I'm concerned by is the flip as it pushes a and then pushes b. SO that would mean that we need to push and then push again and I'm not sure how that's going to work on our datapath. The same goes for dup as it pushes multiple values but that command in itself is intimidating so there's a number of things to be concerned about there as well. I worry that the items I implemented might be lacking in terms of test cases. I'm not sure how extensively to test them but I'm sure that I need to implement more test cases.
- One other thing that came up during our meeting was that our memory addresses can only be 10 bits. This is quite confusing to me. I will probably ask about it during our meeting but I was wondering a number of things; why is this, how will our memory stack have to change to accommodate this, what does this mean for our memory pushes and pops, etc.

Friday 10/25/19

- Turned in the ALU lab (lab 06). There was quite a bit of unfun issues with this lab. A few things first about skipping around - don't. When it comes to components it is truly much much better to build from the smaller parts than to start trying to build a larger component straight away. I had to start over because of this and that wasn't fun. The other really big thing is to make sure that I'm using combinatorial logic rather than sequential. Everything else in the project can be working perfectly but if the logic isn't the right kind then the timing will be all off and nothing will work. This wasn't so much of an issue with add, sub, and, and or but with SLT as it had to forward some data. USE COMBINATORIAL LOGIC, FOREHEAD. Once the ALU was finished (thanks to some generous donations from my local Sidison's Bank), I added a modified 16bit ALU to our own implementation.

Wednesday 10/30/19

- Tried to catch up on Milestone 4. The datapath was redrawn. Had to integrate BEQ and BNE logic. Thanks to the second exam that wasn't so bad. As we look to do control more and more small issues or discrepancies seem to arise. One thing at the moment is that our shift has 2 different versions. The actual hardware at the moment can shift both directions and takes a signed shamt. The RTL version is two commands that do left shifting and right shifting. We believe we're going to switch to the bidirectional shifter in all aspects. There is also a debate whether or not to make the shamt larger. We never actually shift by more than 2 bits in any of our code that needs to run but having a larger shamt from those unused bits would be fine. Having to convert all images (the datapath) to a jpg so that google can insert it is frustrating. I also started to try to integrate some parts together. I am not currently very confident in my verilog skills but hopefully if I keep connecting the individual components it will work out.
- Git is messed up. Oh boy.

Days from Wednesday 10/30/19 up to but not including Wednesday 11/6/19

- These days are being combined into one entry as they were mostly the same. I fell behind in this project. I feel terrible and I've heard that it's upset Tori. I got caught up in other projects that I was trying to finish so that I could work more on this project but they took too long and this milestone is lacking in terms of my effort. I do have plans to make up for this in the coming days by taking over work on an assembler. I would very much like to make one and I feel that it's certainly achievable. I hope that this lapse in work doesn't reflect too poorly upon me and I am quite sorry if I upset any of my team members.

Wednesday 11/6/19

- Team met for 4 hours to complete milestone 5. I worked on making the multicycle diagram all pretty and wrote a testing method for our combined processor. I also worked a little bit on the assembler but parsing is hard. Alternatively, I've forgotten how to do it. I still think that getting the assembler done is a good project for me considering my partners seem quite involved in the datapath. Something that will probably be mentioned in our professor meeting on thursday is what to do with our shifting instruction. I'll mention it here so that hopefully I don't forget. Currently our hardware implementation of our shifter takes in a signed shamt. Our RTL and instructions don't describe this, they describe two different shift methods, one for left and one for right. Our main question is which method we should switch to and commit to. My thinking is that leaving it with the signed shamt is the easier option of the two.

Saturday 11/9/19

- Tori and I met for 4 hours about the datapath and correcting the manual control values. We were able to get almost all of the control values done in this time. I was also updating our multicycle diagrams while this was going on. We changed when pushSrc was set so that it was set in time for pushes.

Sunday 11/10/19



- Met with the group for 2 hours. Changed the clock edges for some of the components and began to work on timing.

Monday 11/11/19

- Began testing the individual instructions with control. I worked with Tori to iron out the different pushes and pops.

Tuesday 11/12/19

- Tested pushR, pushUi and popM. Figured out the multicycle for pushR and pushUi but I was unable to figure out popM. It seems that the write value is a cycle behind the address so then the write value is wrong when the address gets there. Changed the shifter to take in a larger shamt so that pushUi actually puts the right value into the stack.

Wednesday 11/13/19

- Worked with Tori to figure out the issue with popM. We realized that if we don't pop until a later cycle then the value for B will still be the write value and the address will still be calculated correctly. We then started to test various instructions with our connected control to verify that it still worked. Once we were satisfied, we put relPrime into our memory. It didn't work. But, we did discover that or pops too many values so we fixed that. Then we worked on testing recursion and procedure calling so that we have a better chance of making relPrime work. Making recursive functions hurts my brain.

From Wednesday 11/13/19 to Sunday 11/17/19

- Worked on updating the design document quite a bit. I also updated the assembler to handler a few pseudo instructions and made an arithmetic shifter, tested it, and added it into the datapath with control. Pretty sure it works. I talked with my roommate (Andrew Johnson) about our project and he pointed out a few things that could be improved. The main thing is that we could reduce like 6 cycles and also eliminate using a number of pushR instructions. I wish I could've updated our processor to have these changes but I couldn't find the time. That makes me quite sad. What it made me realize though is that I wish I had been more confident during this project. I feel that if I had been more assertive about certain things I could've been quite a bit more helpful for the overall project. This is not to say that I didn't contribute. I did the extra features and while I wish I could have done more with the assembler I'm happy with it. I just wish I'd been more assertive or less intimidated. I just feel that this could've lead to a better processor. I'll have to keep that in mind for future projects.

*Christian Meizen's Journal*

### **Milestone 1:**

Friday 10/4/19

- Discussed for one hour at 4:00 – 5:00 PM • Decided to make processor with Stack architecture • Discussed possible calling conventions and addressing modes • Created three ISA for pushing/popping, jumps/branches, and arithmetic/stack manipulation.

Sunday 10/6/19

- Met in O259 from 5:30 – 8:45 PM • Most of the time was used to discuss the Stack architecture and make sure all team members are on the same page • Will and I worked on framing some calling conventions and designed the assembly code for relPrime and GCD • Used the idea that memory would store return addresses and argument values when making a procedure call • Tori designed instruction set that we could use to get around our ISA and the 16-bit limitations.

Monday 10/7/19

- Met with Sid from 1:20 PM – 2:10 PM to discuss advice and clarification on addressing modes. Realize that using the stack for return addresses will be better than using memory. • Met with group from 3:30 – 6:45 PM to redesign assembly code and instruction set. • Removed ‘jal’ and added ‘dup’ and ‘flip’ instructions for better stack manipulation • Rewrote code for relPrime and GCD in our new instruction set using polished calling conventions where the return address and arguments are placed within the stack, and the return value is placed in memory. • Side notes: o It feels as though without pseudocode, our code will be complicated to efficiently program o While everyone has contributed in some way to the project, I feel as though I have created most of the assembly code. So, I am worried that my teammates might not understand how we are implementing our design through code.

Tuesday 10/8/19

- Met in F225 from 3:50 – (ins) to make final changes and commit Document for Milestone 1. • Made documentation a bit neater by formatting and inserting a header with some footnotes • Push personal journal and Document • Discuss about how we are going to complete Milestone 2 so we can talk to Sid tomorrow and not sound like idiots • I think that communication is key for this milestone because we are doing it during the break time.

## **Milestone 2:**

Friday 10/11/19

- Self-conducted meeting at home.
- Polished up Milestone 1 using critiques that Sid gave at group meeting
- Worked on adding more common operations.
- Thought of a new method of procedure calling. Use a jump and link instruction that takes the PC and puts it on the stack.

Sunday 10/13/19

- Group met in F225 from 7 PM – 10:15 PM to fix remaining Milestone 1 issues and begin Milestone 2.
- Created RTL code for all instructions except duplicate and flip, which we are asking Sid about how to implement in hardware
- Make a better table of contents and organization of document
- Settle on pushM and pushR for more efficient coding. Also, memory allocation will be used by offsetting from the \$sp register. Still need to have the \$sp address on the execution stack
- This meeting went far better than the previous. It seems that everyone understood each change and new implementation.

Monday 10/14/19

- Group met in F225 from 4 PM- 6 PM to finish the RTL code and change some hardware components after talking with Sid.
- Created diagrams that show the inputs and outputs of each required piece of hardware
- Decided on what control signals might be needed for each hardware component
- Implemented a barrel shifter that will both shift right and shift left.
- Put all of the RTL code into a Table so that it is neater and, for the most part, split into a multicycle process. (Might have to change later when designing the datapath)
- Update the code for relPrime and GCD with our new instructions and calling procedures that changed throughout Milestone 2
- This meeting went well, but it seemed that there is still a bit of confusion within the group on duplicated and the execution stack hardware components. Resolved before submitting document.

Tuesday 10/15/19

- Tori and I met in F225 from 4-5 PM to polish document one more time • Finished the RTL verification section by adding pictures of add command • Change pushNum -> dupNum for better clarity • Committed and pushed Milestone 2

### **Milestone 3:**

Saturday 10/19/19 • Started to work on the components for milestone 3. • Realized that I have no clue how to do Verilog, so I worked on tutorials instead • This was solo time at Starbucks when I was at home

Monday 10/21/19

- Worked on my own time throughout the day. • Completed components for Sign and Zero extenders as well as their tests

- Finished the register and register file, but their tests need some work • I am a little worried that Tori and Will have not worked on M3 and focused on the labs instead.

Tuesday 10/22/19 • Met with Sid to discuss Lab 08 and some of the execution stack component • Completed the tests for the register and register file. Tori said that she will work on the memory unit. • I think I finished the component for the execution stack, but the unit test did not work as I wanted.

Wednesday 10/23/19

- Will said that he completed the datapath. Tori and I checked it and agreed that it looked good, but there might need to be a few changes depending on testing. • Tori finished the Memory and started working on describing implementation • I finished the execution stack and started figuring out the control unit and how everything fit together within the datapath.

Milestone 4: Friday 10/25/19 • Rewriting some of the unit test explanations in the document for better understanding. • Work by myself because it was hard to find a time today to get together with the group

Sunday 10/27/19 • Met with group in F225 for about two hours. Started to create the control unit with Will, while Tori was working on combining components together. • Created a table to write down all of the control values for each instruction. • Realized that we need to edit the RTL code a little bit to allow multiple execution stack operations

Monday 10/28/19 • In class, talked with Sid for better understanding how to tell the Execution Stack to do multiple operations in different cycles. • Realize that we needed to update the ES component for an enable bit. Also got rid of the extra output registers because we never really used them. • Found out that the group could not get together until Wednesday, in which I will not be on campus. Asked Sid for extension.

Tuesday 10/29/19 • Today, I wanted to update all of the RTL code so that it correlates to how we want the operations in our ES to execute and be called. • After updating all of the RTL code, I wrote all of the Bubble-Control diagram for each instruction excluding branching and shifting (due to Will saying that we might be changing the slr and sll into one instruction). • Updated the Verilog for the execution stack so that it will go along with our updated datapath/control system. • Tori and Will said that they are able to finish M4 tomorrow.

### **Milestone 5:**

Friday 11/1/2019

• Met alone and worked on completing the execution stack due to some bugs I found while thinking of component implementation. • Wrote the test for execution stack so that it would incorporate my changes.

Monday 11/4/2019

- Worked in class with Tori and discussed what components to work on.
- Decided that she should work on more of the memory and I should focus on the execution stack.
- Will did not show up, but he said that he was working on a lot of the ALU stuff and shifters.
- Realize that I am bad at creating tests – went to Sid for help during office hours.
- o Gave me better insight on how to generate good tests.
- Finished with the ES\_and\_ALU subsystem. Generated a test that looks at all of the different ALU functions. Could possibly be better to add more tests, but I feel it was sufficient to show correctness.

Tuesday 11/5/2019

- Started to work on the ES\_and\_pushVal subsystem. Realized that there was a problem with ex\_stack thanks to Sid.
- Focused on fixing the ex\_stack to adjust the register updating component.
- o I think there is still a problem...need to talk to Sid about the always conditionals for the stack.
- Finished the ES\_and\_pushVal subsystem later in Lab F217. This has not incorporated the problem mentioned above – need to check that it will not be affected later.
- Tori shown that she has worked on multiple parts of control and memory. Will says that he is working, but has not indicated via the design document.
- Need to talk to him about it tomorrow in class

Wednesday 11/6/2019

- In class:
  - o Talked with Will, who said that he had a lot going on in other classes. He said that he was available today to go hard on the datapath.
  - o Talked with Sid about the problem for conditionals. Inserted the clock as a conditional parameter, which seemed to do the trick.
  - This solved most of my problems with ex\_stack and ES\_and\_pushVal.
  - o Started working on Mem\_RegFile\_ES subsystem. Talked with Tori about how the memory works with other components since she worked on memory the most.
- In F217:
  - o Finished the Mem\_RegFile\_ES subsystem.
  - Understood how to generate own memory block using the code that Tori wrote.
  - I think that there might be an unnecessary delay when taking data from the execution stack to registers A and B.
  - o Helped Will with the datapath implementation because he was getting connection errors, just as I once did with the execution stack subsystems.
  - I do not think that we will have the datapath tested in time for the Milestone submission time.
  - Need to create a test for Friday so that we can

talk to Sid about the full implementation. o Tori and I started to talk about I/O. Came up with questions to ask Sid:

- Make sure that I/O is in reserved memory. Make sure that relPrime code is also in memory
- Ask how I/O is to be interpreted and connected with hardware.
- Ask if it is alright to create a component that has logic mixed with the memory block (this might change the datapath a little bit)

### **Milestone 6:**

Saturday 11/9/2019

- Tori and Will worked together on trying to get a series of instructions to work on the processor, I was unable to meet so I stayed in communication via Group Me and provided any debugging/tech support
- We realized that the clocks for registers and components need to be on different edges for faster processing
- After they pushed their changes to the repo, I looked at the new code. I realized that the ES needed a reset, certain multicycle signals needed to be changed in the diagram.
- Communicating in this way was very inefficient, but it was much better than postponing the work until the next day

Sunday 11/10/2019

- Met in F225 from 4:00 – 6:00 PM
- Since Will finished the assembler, he focused most of his time on updating the design document to see if there are any extra components that we don't need.
- o Found out that we could reduce the mux to ALUSrcB to one bit.
- Tori continued to work on writing code for the series of instructions. Once this is finished we can then implement our control.
- I spent this time helping Tori figure out some of the clocking issues and testing. Half way through, I switched over to rewriting code to align with the updates that Will found in our datapath.
- The testing and datapath seems to be complete. We are thinking that we can now implement control. After that, it is running our program followed by I/O.
- o I will be looking over our code for relPrime again tonight to make sure that it is efficient and updated to our datapath since it has been a while since it has been touched.

Monday and Tuesday – (I was dead in bed)

- Tori and Will said that they are putting together control and testing different programs using the assembler.
- I have rewritten some of the relPrime code to be faster/updated. I have also designed the I/O implementation via paper

Wednesday 11/13/2019

- Met in class and in F217 from 4:00 PM – 6:00PM
- Write the I/O implementation in Verilog. \*Need to implement in datapath\*
- Write a recursive program to test stack pointer manipulation as well as procedure calling
- Update the memory layout to include I/O and get rid of the static data that is never used.

### **Post Milestone 6:**

- Finished updating I/O so that it can take in an input from the test bench, so now we do not have to recompile the code every time
- Took Friday off
- Went into F217 from 4 PM - 10 PM working on trying to get I/O to work on the FPGA board. Ran into errors that I could not solve
- Room F217 from 11AM - 2PM working on FPGA again, no success. Also updated some of the design document.
- Room F217 from 4 PM - 1 AM finishing the design document and presentation
- Push everything to git repo
- Finally finished!

### *Victoria Szalay's Journal*

**10-2:** Met with group (15 min). We decided to either use a memory-to-memory architecture or a stack architecture

**10-4:** Met with group(1 hour). We decided on a stack architecture and began creating our ISA. We decided on three different types of instructions: A, B, and C. We then decided on which special registers to keep, along with a list of instructions that we need to implement.

**10-6:** Met with group(2.5 hours). Christian and Will worked on RelPrime code and convention call procedures while I worked on the instruction syntax and semantics so that all three of us would be on the same page. Some decisions I made about the instruction syntax is that

1. All instructions are lower case
2. Most instructions will be named similarly to MIPS instructions
3. The op-codes are organized by type

I also have begun identifying some pseudo-instructions/common operations that will make writing assembly shorter since it is extremely verbose right now



The biggest challenge was deciding on what variations of push to include so that we can load immediates on the stack and get/put values into memory.

**10-7:** Met with group(3 hours) . Christian and I discussed the ISA that we currently have and push/pop further. We then added ISA instructions and deleted some based on the discussions that we had with you. Will and Christian then discussed/finalized the calling procedures. I created/updated the op-codes for those instructions while Christian updated the RelPrime code to match our syntax. I went through and drew stack for Christian's code to check and see that it made sense. Will typed all completed code and I began translating the finalized code into opcode. We decided on the size of the register stack based on our discussion with you.

**10-8:** Met with group(30 minutes). We finalized the milestone 1 document and are going to push our stuff. We plan on looking through milestone 2 to make a plan on how to complete it over break. **End of milestone 1.**

**10-9:** Met with group (1 hour). Discussed feedback and decided what fixes to have done by the time we meet again. We got a rough outline of the memory allocation stack done and changed our reserved memory locations to reserved registers. We decided to add two new instructions, pushR and popR, to our ISA.

**10-12:** I worked on the project (45 minutes). I added pushR and pop R to our ISA, along with a pushM and popM. I then added some conditionals and loops to our common operations section, but ran into a problem with how to preserve our local variables, which was also brought up in our feedback.

**10-13:** Met with group(3 hours). We implemented our special sp register into our realprime code, which shortened the code by about a page and a half. We decided to keep the sp register at the top of memory stack and to store our local variables with a modified pushM and popM that adds onto sp with an immediate. Now the programmer can save local variables onto the memory stack and then deallocate the same memory once the function is complete. Christian completed what was left of common operations while Will fixed the realprime code. I began work on Milestone 2 by creating the execution stack component and listing all of the necessary components. I then started writing RTL instructions for all of our instructions. Eventually all three of us worked together to finish single-cycle RTL for all of our instructions except for flip and dupli. We then compiled questions for Sid that we still have for milestone 1 and 2. An important side note is that we are still considering ways to implement our procedure conventions, and ran into a problem on how to save the return address in the exact order that we specified on the execution stack (return address on the bottom of the stack before the call) . We came up with either an instruction that directly changes the pc counter or using a label that we can push before the arguments. We will finalize that decision after discussing it with you tomorrow.

**10-14:** Went to office hours to ask questions the group decided on (15 min). Met with group(2 hours). I shared the answers to the questions I'd asked during office hours with Will and Christian. Will and Christian then finished the RTL code for flip, dup, sll, and slr while I started to work on what control signals we would need for our components. Will drew our basic component pictures while Christian and I filled out the rest of the information in component list. Christian and I finalized designing the execution stack component while Will compiled our RTL code into a simplified multi-cycle chart. We decided to finish our summary of how we checked our components tonight and to review/refine our document together tomorrow.

**10-15:** Met with Christian (1 hour). We fixed several small errors in our required hardware section and then created our process of checking the RTL code. We then turned in Milestone 2.

**End of Milestone 2.** We began to look at M3's documentation. We decided to begin the work by implementing our components. From our experience, it would probably be best to begin by designing the ALU, register unit, adder, and a mux. This makes sense because our memory, barrel shifter, execution stack, and register file are more complex. A good goal to have would be to have these units tested and implemented in Xilinx by Thursday.

**10-16:** Met with group (10 min). We divvied up the labs. I get lab 7 memory, Will has lab 6, and Christian has lab 8. We are meeting tomorrow 5-7.

**10-18:** Worked on memory lab 7 (1 hour).

**10-21:** Worked on memory lab 7 (3 hours). Completed the memory portion. Met with group (2 hours). Christian worked on making corrections on feedback while Will worked on ALU lab. Will and Christian divided their time creating components while I continued working on the memory lab and completed the control portion. We as a group plan on completing the register file and execution stack together. Worked on memory lab (2 hours). The memory and control components work separately but do not work together. It has to be a problem with how I am connecting them.

**10-22:** Worked on the project for a total of 5 hours for the day. Worked and finished memory lab 7 (2 hours). Met with Will (3 hours). I fetched Christian's Xilinx components and went through/checked them in order to see how they worked and to see what in the design document needed changed. I found that the register file currently doesn't have our zero register set to where it cannot be written to. It currently is not always zero. Other than that, everything looked great. Will worked on fixing things from milestone 2. I created our memory component, but am unsure whether or not an address length of 16 will be too much for the FPGA board like lab 7 suggests. If so, we are going to have to adjust our pushM, popM, and program counter accordingly.

**10-23:** Met with group (2 hours). I demonstrated lab 7 and then completed the memory testbench testing for milestone 3. Christian worked on execution stack while Will began the datapath. We ran into an issue with whether or not popNum was a control or not, but we have decided that it is. Met with Christian (4 hours). I updated most of the components on the list, created the control unit, described all of the control symbols and created the implementation plan. Will finished ALU testing and updated the component list. Christian finished testing on the execution stack. **End of Milestone 3.** We have decided to perfect our unit testing for the next milestone and to begin implementing our implementation plan after our meeting with you.

**10-25:** Worked on control list graphs (30 min).

**10-27:** Met with group (1.5 hours). We worked together on a control signals chart. We are confused about the timing of the execution stack since it is performing more than two operations at a time with our execution stack.

**10-28:** Met with group (1 hour). We discussed it with you and a major design decision has been made. Instead of popping off two values every time we do an operation, we are performing a peek. Once the operations on the peeked values are performed through the datapath, the value is popped back onto the stack and the other operations that occur inside the execution stack depending on the instruction. The execution stack deals with putting its final value on like so:

1. The stack always pushes one value onto the stack, what it pushes onto the stack is determined by pushSrc
2. The control signal popAmt determines how many pops to perform on the stack

Christian agreed to change our RTL and execution stack to reflect these changes. I have agreed to finish the multicycle control diagram. Due to tests and special events this week, we have asked for an extension on Milestone 4. I won't be able to work on the project until Wednesday due to seven hours of exams on Tuesday night and Wednesday morning.

**10-30:** Worked on state transition diagram (3 hours). We had to add additional logic and controls to our datapath in order for beq and bne to work properly. I also changed our RTL to always perform push/popM calculation so that our transition state diagram would not have an empty bubble. I then created as much as I could of our state transition diagram. I plan on finishing that and working on the control unit implementation tonight and having that done before our meeting with you as well as the first sublevel of the implementation plan. Worked on and finished transition state diagram (1 hour).

**10-31:** Worked on control unit component (3 hours). I am attempting to put the multicycle diagram into verilog, but am slightly confused on how to represent the four stages (fetch, decode, execute, and done). Talked to you and then looked at code on website to better understand.

**11-1:** Worked on control unit component (2.5 hours). I have begun testing just the control unit, but plan on adapting the .coe file so that I can test the control unit and memory together. **End of Milestone 4.**

**11-3:** Worked on control unit testing (1.5 hours). I am currently only using a testbench that sets one instruction and checking that the appropriate controls are set and states are reached. I am currently having issues reaching the correct state after the decode stage. For some reason, it is always the state that is 6 states above the desired one.

**11-4:** Worked on control unit (5 hours). The error from the day before was resolved after I asked you. I had forgotten to extend the number of bits on the current state and next state variables so it was cutting off my values as I was setting them. Once this issue was fixed, all of the instructions worked individually and set the correct control bits. It is worth noting that to complete the max number of states per instruction ( 5 states), the clock needs flipped nine times. I then connected the memory and control modules, and then inputted machine code into the .coe file. I then tested a sequence of seven instructions.

**11-5:** Worked on the project for a total of 6.5 hours for the day. Worked on and completed sublevel test 1, e (4.5 hours). This subtest integrates the ALU, ALUout, Memory, and Memout. This subtest is essentially checking whether the popM and pushM instructions work properly without the execution stack or controls involved. I had trouble with making the timing on my tests for awhile, but I used an always block to make it easier. Worked on sublevel test 1, c (2 hours). This subtest tests that the shifter, sign extender, and zero extender can properly extend and shift the proper bits of the opcode from memory. I noticed a flaw and showed it to Will, who found a flaw in our original datapath that is fairly easy to fix. Essentially, we found that we need to add an additional sign extender component to the datapath so that we can input into the shifter, which takes sixteen bits in. I applied the fix in the partial implementation test.

**11-6:** Worked on project for a total of 4 hours today. Worked on and finished sublevel test 1, c (2 hours) from yesterday. After the change was made, the tests I conducted went smoothly. Began implementing the entire datapath (1.5 hours) with the control signals as inputs. Passed it off to Will and began going over the design document to update it. I discussed how we should implement IO and the sp register with Christian for about thirty minutes. We are compiling a list of questions for office hours today and tomorrow. **End of Milestone 5.**

**11-7:** Worked on project for a total of 9 hours today. Redid subtests in order to make implementation of datapath easier. Completely connected datapath and began datapath testing. A slight correction was made to our datapath was made in that one of the sources to mux ALUSrcB has to be sign extended.

**11-8:** Worked on project for a total of three hours. I changed our pcadder to increment by 1 because I did not realize that we are not using block addressing until I began hand-setting control signals for the datapath.

**11-9:** Worked on project for a total of eight hours. Will and I debugged some name and size errors on the datapath and then created a sequence of instructions to put through the datapath. Pushli worked perfectly the first time, but we had to correct our es\_subsystem in order to get add to work. Specifically, I accidentally used the wrong alu instance which Will fixed. We then slightly changed our control sequence because the pushSrc needs set a cycle before the execution stack pushes it. I am now trying to completely get our branching instructions to work.

**11-10:** Worked on project for a total of five hours today. We are having random pushes and pops with our instructions, and I figured out it was timing issues between our registers and components. I met with the group for two hours and we changed it to where components are on the positive clock edge and registers are on the negative clock edge. This made our arithmetic instructions work. We were having problems with beq but we figured out that it needed split into more stages.

**11-11:** Worked on project for a total of 12 hours. We almost have all instructions through the datapath, and I have corrected our state diagram to reflect any changes made so that they can be applied to the control.v file for total integration. Will helped test some different sequences of instructions to ensure branching really works.

**11-12:** Worked on project for a total of 10 hours. I completed branch/jump instruction testing for the manual control datapath. I then updated the control unit to reflect the many changes that we made to the multicycle transition diagram. I combined the control and datapath and struggled with the sizes of the controls for awhile. Once I began testing the testbench, I ran into several errors. All of these can be traced back to unnecessary resets I'd been making at the beginning of each current stage. Once I found this, everything worked. The combined datapath and control are officially confirmed for 10 instructions, but I am not counting arithmetic instructions. Will has been working on testing the others by manually setting the controls in the datapath, so hopefully we can integrate these tomorrow and get input/output completed.

**11-13:** Worked on project for a total of 7 hours. Finished adding and testing every single instruction through integrated control and datapath. Worked with Will to initialize the sp register properly and to update the assembler so that we could begin writing test programs faster.

**11-14:** Worked on project for a total of 9 hours. Began writing test code for our processor and also to familiarize with our assembler, including a multiplication function in order to see the best

way to begin rewriting our relprime code. I then handed off the multiplication function to Christian so that he could try to get IO to work with a function. Will and I decided to write relPrime separately so that we wouldn't get stuck on the same things. Will completed relPrime before me and then began testing our processor specs while I tried to help Christian with IO as much as I could. Will jumped in as well once he was done with timing. We had multiple problems with version control of the project tonight, which slowed us down a bit.

**11-15:** Worked on project for a total of 1.5 hours. Helped Christian as much I could so that IO would be complete after getting help from you. **End of Milestone 6.**

**11-17:** Worked on project for a total of 8 hours. Helped create comments and waveforms for repo. Worked a lil on design document. **End of project.**