VIOLET TEAM DESIGN SPECIFICATIONS

Rose-Hulman Institute of Technology CSSE232:
Computer Architecture I

Drew Egler, William Gardner, Athena Henderson, Chirag Sirigere

High Level Design Description

The processor being designed by the group will utilize a stack architecture. This means that all instructions will be done through the pushing and popping of values in and off the stack. No registers will be utilized by users since most of the process will be done with the stack and the memory. Our design will utilize three basic 2-byte instruction formats. First is the P-type. Here, 5-bits will be dedicated to the opcode, 8-bits will be for the immediate, 1-bit will be for determining if the value is the upper or lower half of a 16-bit number, and 2-bits will be the dead space, which means they are a "do not care." The P-type will be used for instructions including push and all branching. Second is the A-type. Here, 5-bits will be dedicated to the opcode and 1-bit is for swapping the order of stack parts used in operations. For instance, subtraction would become Stack(sp0) = Stack(sp1) - Stack(sp0). The other 10-bits will be dead-bits. The A-type will be used for instructions based around bit arithmetic and addresses, including alu operations, pop, peek, loads from memory, and jumps. Third is the S-Type. Here, 5-bits will be dedicated to the opcode, and 3-bits will be for the shift amount, or "SHAMT" for short. The other 8-bits will be dead-bits. The S-type will be used for all bit shifting.

Instruction Formats

P-Type			
5	8	1	2
Opcode	Immediate	isUpper	Don't Care
A-Type			
5	1	10	
Opcode	Swap (swaps order of stack used in operation)	Don't Cares	
S-Type			
5	3	8	
Opcode	Shift Amount	Don't Cares	

The instructions are made with idea of operations for relPrime, basic math, and combinational logic in mind. Each of the A-type formatted instructions will remove either one or two numbers off the stack. The branches always remove the numbers they used for comparisons, and push is used to put information on the stack. Jumps, pop, and peek require that you have their address on the stack before calling them. This is further explained in the operations section of the instruction set table. There is a key at the bottom of the table to help explain the jargon used in each operation.

Instruction Set

NAME	FORMAT	OPERATION	OPCODE
Add	A	sp0 = Remove(sp0) + Remove(sp1)	00000
	Add: The top two values are popped off the stack and added together. The result is then put back on top of the stack.		
And	A	sp0 = Remove(sp0) & Remove(sp1)	00001
	And: The top two values are popped off the stack and the AND operation is used on them. The result is then put back on top of the stack.		
Beq	P	if (Remove(sp0) == Remove(sp1)) PC = PC + 2 + SEAddr	00010

	Branch if EQual: The top two values are popped off the stack and compared. If the two are equal, then branch to the new address.		
Bez	P	if (Remove(sp0) == 0) PC = PC + 2 + SEAddr	00011
	Branch if Equal to Zero: The top value is popped off the stack and compared. If the value is equal to 0, then branch to the new address.		
Bge	P	if (Remove(sp0) >= Remove(sp1)) PC = PC + 2 + SEAddr	00100
	Branch if Greater than or Equal: The top two values are popped off the stack and compared. If the first value is greater than or equal to the second, then branch to the new address.		
Bgt	P	if (Remove(sp0) > Remove(sp1)) PC = PC + 2 + SEAddr	00101
	Branch if Greater Than: The top two values are popped off the stack and compared. If the first value is greater than the second, then branch to the new address.		
Ble	P	if (Remove(sp0) <= Remove(sp1)) PC = PC + 2 + SEAddr	00110
	Branch if Less than or Equal: The top two values are popped off the stack and compared. If the first value is less than or equal to the second, then branch to the new address.		
Blt	P	if (Remove(sp0) < Remove(sp1)) PC = PC + 2 + SEAddr	00111
	Branch if Less Than: The top two values are popped off the stack and compared. If the first value is less than the second, then branch to the new address.		
Bne	P	if (Remove(sp0) != Remove(sp1)) PC = PC + 2 + SEAddr	01000
	Branch if Not Equal: The top two values are popped off the stack and compared. If the two are not equal, then branch to the new address.		
Bnz	P	if (Remove(sp0) != 0) PC = PC + 2 + SEAddr	01001
	Branch if Not equal to Zero: The top value is popped off the stack and compared. If the value is not equal to 0, then branch to the new address.		
Inc	A	sp0 = Remove(sp0) + 1	01010
	Increment/Increase: The top value is popped off the stack and increased by 1. The result is then put back on top of the stack.		
J	A	PC = Remove(sp0)	01011
	Jump: The PC is set to the top value popped off the stack.		
Jal	A	PC = Remove(sp0); sp0 = (PC+2)	01100
	Jump And Link: While it adds 2 to the old PC, PC is also set to the top value popped off the stack, then the address of the next instruction is put on top of the stack.		
LShift	S	sp0 = Remove(sp0) << SHAMT	01101
	Left Shift: The top value is popped off the stack and is shifted left SHAMT amount with ‘b0s. The result is then put back on top of the stack.		
LShiftSE	S	sp0 = Remove(sp0) << SHAMT (shifts in ‘b1s instead of ‘b0s)	01110
	Left Shift Sign Extend: The top value is popped off the stack and is shifted left SHAMT amount with 1s. The result is then put back on top of the stack.		
Nor	A	sp0 = ~(Remove(sp0) | Remove(sp1))	01111
	Nor: The top two values are popped off the stack and the NOR operation is used on them. The result is then put back on top of the stack.		

Or	A	sp0 = Remove(sp0) | Remove(sp1)	10000
	Or: The top two values are popped off the stack and the OR operation is used on them. The result is then put back on top of the stack.		
Peek	A	MEM[sp0] = sp1; Remove(sp0)	10001
	Peek/Store word: The second from the top value is popped off the stack and is stored into memory at the address that is on top of the stack. The Address is removed from the stack, but the value stays.		
Pop	A	MEM[sp0] = Remove(sp1); Remove(sp0)	10010
	Pop: The second from the top value is popped off the stack and is stored into memory at the address that is on top of the stack. Both values are removed from the stack.		
Pusha	A	sp0 = MEM[sp0]	10011
	Push from Address/Load word: A value at the address on top of the stack is put on top of the stack.		

Push	P	if (isUpper == 0) then
		sp0 = ZEImm
		else
		sp0 = UpImm
			10100

	Push: The immediate passed in is put on top of the stack.		
RShift	S	sp0 = Remove(sp0) >> SHAMT	10101
	Right Shift: The top value is popped off the stack and is shifted right SHAMT amount with ‘b0s. The result is then put back on top of the stack.		
RShiftSE	S	sp0 = Remove(sp0) >> SHAMT (shifts in ‘b1s instead of ‘b0s)	10110
	Right Shift Sign Extend: The top value is popped off the stack and is shifted right SHAMT amount with ‘b1s. The result is then put back on top of the stack.		

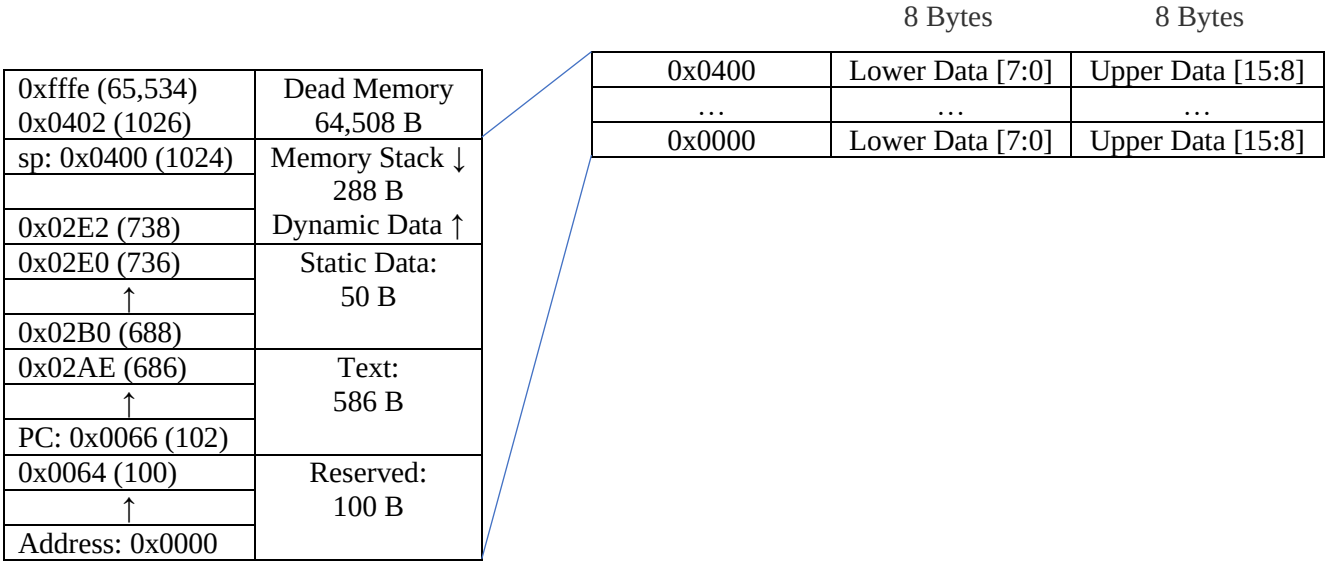
Sub	A	if (Swap == 0) then
		sp0 = Remove(sp0) - Remove(sp1)
		else
		sp0 = Remove(sp1) - Remove(sp0)
			10111

	Subtract: The top two values are popped off the stack and subtracted together. The result is then put back on top of the stack.		
Xor	A	sp0 = Remove(sp0) $\oplus$ Remove(sp1)	11000
	Xor: The top two values are popped off the stack and the XOR operation is used on them. The result is then put back on top of the stack.		

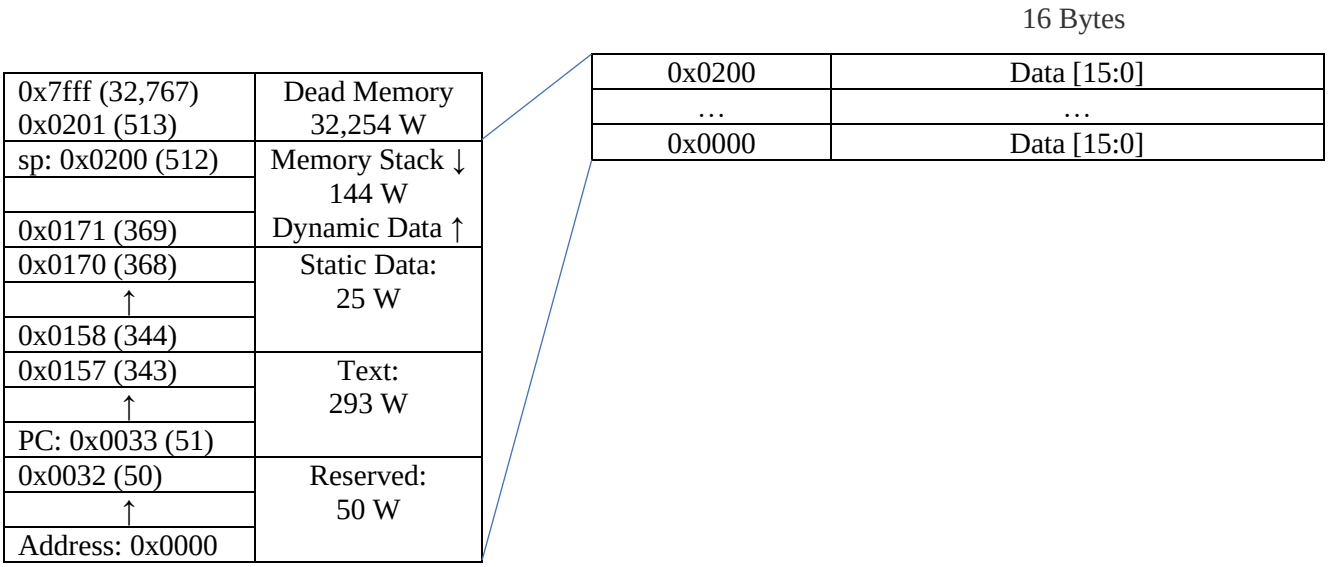
Key	
sp#	# of slots away from the top of the stack (i.e. sp0 = top of stack)
Remove	Removes that information from that stack slot
PC	Program Counter
MEM	Memory
SEAddr	{7{immediate [7]}, Immediate, 1’b0}
ZEImm	{8’b0, Immediate}
UpImm	{Immediate, 8’b0}
SHAMT	Shift Amount
INPUT	Input value externally from processor

Below are the two different addressing types for memory. Since the client wanted 10-bit addresses, we use the byte addresses to access the different values within memory. However, since words need to be 16-bits, we translate the byte addressing to word addressing. The processor uses byte addressing for memory, but when it goes to read, it translates the byte address into word address to pull out full 16-bit numbers.

Byte Addressed Memory Allocation for 10-bit Memory Block



Word Addressed Memory Allocation for 10-bit Memory Block



Static Variables

There are four pre-determined static variables for all functions to use. These four variables are described below. The rest of the space in static variables is for other variables.

0x02B0 = ans
This is where return values from functions will be stored.
0x02B2 = i
This is where loop indexes can be stored.
0x02B4 = n
This is where an input for functions will be stored.
0x02B6 = m
This is where another input for functions can be stored.

Procedure Calling Conventions

Below are translations for the following high-level code to this processor's assembly and machine code next to its PC addresses. These snippets are meant to give examples of how to use the assembly code for its main purpose, relPrime, and different common coding tasks, such as loops and recursion.

For two numbers to be relatively prime, their greatest common divisor (gcd) must be one (i.e. they must have no common divisors other than one). Euclid's algorithm is used to determine the gcd of two numbers.

High Level Language	Assembly Code	Machine Code	PC Address

int relPrime(int n)
{
  int m;
 	
  m = 2;
  while (gcd(n, m) != 1) {
  m = m + 1;
  }
  return m;
}
 	
int gcd(int a, int b) {
  if (a == 0) {
  return b;
  }
 	
  while (b != 0) {
  if (a > b) {
  a = a - b;
  } else {
  b = b - a;
  }
  }
 	
  return a;
}
	RELPRIME:
	P Push 2
	  LOOP:
	P Push LOWER(m)
	P Push UPPER(m)
	A Or
	A Peek
	P Push LOWER(b)
	P Push UPPER(b)
	A Or
	A Peek
	P Push LOWER(n)
	P Push UPPER(n)
	A Or
	A Pusha
	P Push LOWER(a)
	P Push UPPER(a)
	A Or
	A Pop
	P Push LOWER(GCD)
	P Push UPPER(GCD)
	A Or
	A Jal
	P Push LOWER(ans)
	P Push UPPER(ans)
	A Or
	A Pusha
	P Push 1
		RELPRIME:
		A010
		  LOOP:
		A5B0
		A014
		8000
		8800
		A5D0
		A014
		8000
		8800
		A5A0
		A014
		8000
		9800
		A5C0
		A014
		8000
		9000
		A5B0
		A004
		8000
		6000
		A580
		A014
		8000
		9800
		A008
			
			0x66
			
			0x68
			0x6A
			0x6C
			0x6E
			0x70
			0x72
			0x74
			0x76
			0x78
			0x7A
			0x7C
			0x7E
			0x80
			0x82
			0x84
			0x86
			0x88
			0x8A
			0x8C
			0x8E
			0x90
			0x92
			0x94
			0x96
			0x98

	P Beq END	1028	0x9A
	A Inc	5000	0x9C
	P Push LOWER(LOOP)	A340	0x9E
	P Push UPPER(LOOP)	A004	0xA0
	A Or	8000	0xA2
	A J	5800	0xA4
	END:	END:	
	P Push LOWER(ans)	A580	0xA6
	P Push UPPER(ans)	A014	0xA8
	A Or	8000	0xAA
	P Pop	9000	0xAC
	P Push LOWER(RTN)	A1A0	0xAE
	P Push UPPER(RTN)	A00C	0xB0
	A Or	8000	0xB2
	A J	5800	0xB4
	GCD:	GCD:	
	P Push LOWER(a)	A5C0	0xB6
	P Push UPPER(a)	A014	0xB8
	A Or	8000	0xBA
	A Pusha	9800	0xBC
	P Bnz LOOP	4860	0xBE
	P Push LOWER(b)	A5D0	0xC0
	P Push UPPER(b)	A014	0xC2
	A Or	8000	0xC4
	A Pusha	9800	0xC6
	P Push LOWER(ans)	A580	0xC8
	P Push UPPER(ans)	A014	0xCA
	A Or	8000	0xCC
	P Pop	9000	0xCE
	P Push LOWER(RTN)	A1A0	0xD0
	P Push UPPER(RTN)	A00C	0xD2
	A Or	8000	0xD4
	A J	5800	0xD6
	LOOP2:	LOOP2:	
	P Push LOWER(b)	A5D0	0xD8
	P Push UPPER(b)	A014	0xDA
	A Or	8000	0xDC
	A Pusha	9800	0xDE
	P Bez END	1910	0xE0
	P Push LOWER(a)	A5C0	0xE2
	P Push UPPER(a)	A014	0xE4
	A Or	8000	0xE6
	A Pusha	9800	0xE8
	P Push LOWER(b)	A5D0	0xEA
	P Push UPPER(b)	A014	0xEC
	A Or	8000	0xEE
	A Pusha	9800	0xF0
	P Push LOWER(a)	A5C0	0xF2
	P Push UPPER(a)	A014	0xF4
	A Or	8000	0xF6
	A Pusha	9800	0xF8
	P Push LOWER(b)	A5D0	0xFA
	P Push UPPER(b)	A014	0xFC
	A Or	8000	0xFE
	A Pusha	9800	0x100
	P Bge ELSE2	2038	0x102
	A Sub (swapped)	BC00	0x104

	P Push LOWER(a)	A5C0	0x106
	P Push UPPER(a)	A014	0x108
	A Or	8000	0x10A
	P Pop	9000	0x10C
	P Push 0	A000	0x10E
	P Bez ELSE3	1828	0x110
	ELSE2:	ELSE2:	
	A Sub	B800	0x112
	P Push LOWER(b)	A5D0	0x114
	P Push UPPER(b)	A014	0x116
	A Or	8000	0x118
	P Pop	9000	0x11A
	ELSE3:	ELSE3:	
	P Push LOWER(LOP2)	A6C0	0x11C
	P Push UPPER(LOP2)	A004	0x11E
	A Or	8000	0x120
	A J	5800	0x122
	END2:	END2:	
	P Push LOWER(a)	A5C0	0x124
	P Push UPPER(a)	A014	0x126
	A Or	8000	0x128
	A Pusha	9800	0x12A
	P Push LOWER(ans)	A580	0x12C
	P Push UPPER(ans)	A014	0x12E
	A Or	8000	0x130
	P Pop	9000	0x132
	RTN:	RTN:	
	A J	5800	0x134

if (n == 0) {
n++;
} else {
n = 2;
}
	P Push LOWER(n)	A5A0	0x66

	P Push UPPER(n)	A014	0x68
	A Or	8000	0x6A
	A Pusha	9800	0x6C
	P Bnz ELSE	4868	0x6E
	P Push LOWER(n)	A5A0	0x70
	P Push UPPER(n)	A014	0x72
	A Or	8000	0x74
	A Pusha	9800	0x76
	A Inc	5000	0x78
	P Push LOWER(n)	A5A0	0x7A
	P Push UPPER(n)	A014	0x7C
	A Or	8000	0x7E
	A Pop	9000	0x80
	P Push LOWER(END)	A464	0x82
	P Push UPPER(END)	A004	0x84
	A Or	8000	0x86
	A J	5800	0x88
	ELSE:	ELSE:	
	P Push 2	A010	0x8A
	P Push LOWER(n)	A5A0	0x8C
	P Push UPPER(n)	A014	0x8E
	A Or	8000	0x90
	A Pop	9000	0x92
	END:	END:	

while (n != 0) {
n = n - m
}
	LOOP:	LOOP:	

	P Push LOWER(n)	A5A0	0x66
	P Push UPPER(n)	A014	0x68
	A Or	8000	0x6A

	A Pusha P Bez END P Push LOWER(m) P Push UPPER(m) A Or A Pusha P Push LOWER(n) P Push UPPER(n) A Or A Pusha A Sub P Push LOWER(n) P Push UPPER(n) A Or A Pop P Push LOWER(LOOP) P Push UPPER(LOOP) A Or A J	9800 1888 A5B0 A014 8000 9800 A5A0 A014 8000 9800 B800 A5A0 A014 8000 9000 A330 A004 8000 5800	0x6C 0x6E 0x70 0x72 0x74 0x76 0x78 0x7A 0x7C 0x7E 0x80 0x82 0x84 0x86 0x88 0x8A 0x8C 0x8E 0x90

int count = 0;
for (int i = 0; i < n; i++) {
count++;
}
	P Push 0
	P Push 0
	P Push LOWER(i)
	P Push UPPER(i)
	A Or
	A Pop
	LOOP:
	P Push LOWER(n)
	P Push UPPER(n)
	A Or
	A Pusha
	P Push LOWER(i)
	P Push UPPER(i)
	A Or
	A Pusha
	P Bge END
	A Inc
	P Push LOWER(i)
	P Push UPPER(i)
	A Or
	A Pusha
	A Inc
	P Push LOWER(i)
	P Push UPPER(i)
	A Or
	A Pop
	P Push LOWER(LOOP)
	P Push UPPER(LOOP)
	A Or
	A J
		A000
		A000
		A590
		A014
		8000
		9000
		LOOP:
		A5A0
		A014
		8000
		9800
		A590
		A014
		8000
		9800
		2070
		5000
		A590
		A014
		8000
		9800
		5000
		A590
		A014
		8000
		9000
		A390
		A004
		8000
		5800
			0x66
			0x68
			0x6A
			0x6C
			0x6E
			0x70
			
			0x72
			0x74
			0x76
			0x78
			0x7A
			0x7C
			0x7E
			0x80
			0x82
			0x84
			0x86
			0x88
			0x8A
			0x8C
			0x8E
			0x90
			0x92
			0x94
			0x96
			0x98
			0x9A
			0x9C
			0x9E

int count_down (int n) {	CD:	CD:	
if (n == 0) {	P Push LOWER(n)	A5A0	0x66
return 0;	P Push UPPER(n)	A014	0x68
}	A Or	8000	0x6A
return (n +	A Pusha	9800	0x6C
count_down(n-1));	P Bnz ELSE	4848	0x6E
}	P Push 0	A000	0x70
	P Push LOWER(ans)	A580	0x72
	P Push UPPER(ans)	A014	0x74
	A Or	8000	0x76
	A Pop	9000	0x78
	P Push LOWER(END)	A5C0	0x7A
	P Push UPPER(END)	A004	0x7C
	A Or	8000	0x7E
	A J	5800	0x80
	ELSE:	ELSE:	
	P Push LOWER(n)	A5A0	0x82
	P Push UPPER(n)	A014	0x84
	A Or	8000	0x86
	A Pusha	9800	0x88
	P Push 1	A008	0x8A
	P Push LOWER(n)	A5A0	0x8C
	P Push UPPER(n)	A014	0x8E
	A Or	8000	0x90
	A Pusha	9800	0x92
	A Sub	B800	0x94
	P Push LOWER(n)	A5A0	0x96
	P Push UPPER(n)	A014	0x98
	A Or	8000	0x9A
	A Pop	9000	0x9C
	P Push LOWER(CD)	A330	0x9E
	P Push UPPER(CD)	A004	0xA0
	A Or	8000	0xA2
	P Jal	5800	0xA4
	P Push LOWER(ans)	A580	0xA6
	P Push UPPER(ans)	A014	0xA8
	A Or	8000	0xAA
	A Pusha	9800	0xAC
	A Add	0000	0xAE
	P Push LOWER(ans)	A580	0xB0
	P Push UPPER(ans)	A014	0xB2
	A Or	8000	0xB4
	A Pop	9000	0xB6
	END:	END:	
	A J	5800	0xB8

The register transfer language, or RTL, is broken into 6 stages. Most instructions use 3 or 4, but Push from Address, or Pusha, uses 6 stages in order to grab an address from the stack, grab a value from memory, store that back on the stack, then allowing the control bits to settle. The stages are called Fetch, Decode, Execute, Store, Pusha (for pusha to write memory onto the stack), and None for Pusha.

RTL Instructions

Operations	Inst Fetch	Inst Decode	Execute	Store	Pusha	None

A-Type (excluding J, Jal, Pop, Peek, and Pusha)	IR = Mem[PC]
	PC = PC + 2
		isUp = IR[2]
		Swap = IR[10]
		SHAMT = IR[10:8]
		Imm = IR[10:3]
		UpImm = Imm<<16
		ALUOut = PC +(SE(Imm)<<1)
			if(Swap == 0)then
			A = R(sp0); B = R(sp1)
			else
			A = R(sp1); B = R(sp0)
			ALUOut = A op B
				sp0 = ALUOut		

Inc			A = R(sp0) ALUOut = A + 1	sp0 = ALUOut		

Branch between 2 numbers			A = R(sp0)
			B = R(sp1)
			if(A op B)then
			PC=ALUOut
			else			

Branch compare to 0			A = R(sp0)
			if(A op 0)then
			PC = ALUOut
			else			

Pop			A = R(sp0) B = R(sp1)	Mem[A]=B		
Peek			A = R(sp0) B = E(sp1)			

Push			if(isUp == 1)then
			sp0 = UpImm
			else
			sp0 = Imm			

Pusha			A = R(sp0)	MDR = Mem[A]	sp0 = MDR	reset ctrl
J			PC = R(sp0)			
Jal			ALUOut = PC + 2 PC = R(sp0)	sp0 = ALUOut		
LShift			A = R(sp0) ALUOut = A << SHAMT	sp0 = ALUOut		
LShiftSE			A = R(sp0) ALUOut = A ~<< SHAMT			
RShift			A = R(sp0) ALUOut = A >> SHAMT			
RShiftSE			A = R(sp0) ALUOut = A ~>> SHAMT			
Input			Mem[n] = INPUT			

Key	
sp#	# of slots away from the top of the stack (i.e. sp0 = top of stack)
R(_)	Removes that information from that stack slot
E(_)	Read the information from that stack slot, but keep it on the stack

This table shows a simplified explanation of the inputs, outputs, and control signals and their size from our main components. The Register encompasses all registers like PC, the instruction register (IR), the memory data register (MDR), and the ALU output register (ALUOut). This table is also followed by a short description of the commponents.

	Input Signals	Output Signals	Control Signals
Memory	• Address (15:0) • Write Data (15:0)	• Memory Data (15:0)	• MemWrite • Clock
Register	• Data (15:0)	• Result (15:0)	• Write Enable • Clock
Stack Memory	• Plop (15:0)	• sp0 (15:0) • sp1 (15:0)	• StackWrite • StackRemove0 • StackRemove1
ALU	• A (15:0) • B (15:0)	• Result (15:0) • zero? • negative?	• ALUOp(3:0)

- Memory – PC feeds in new addresses to sets of instructions. Instructions are fed to the stack and other components to carry out the operation, and to Control to make control signals for the other components. Data is an integer to be put into a memory location. Address is the location for where to put/get the Data. Memory Data is the number that was stored in the specified memory location. MemWrite decides whether to write into a location in memory or not. Memory is always reading from its address unless MemWrite is on.
- Register – Stores data between cycles to use later. Some registers only save for one cycle, and others save for until they are told to override the value.
- Stack Memory – Has one input because that is what the stack uses to write onto it. Sp0 is what the processor retrieves from the top, and sp1 is the second from the top. StackWrite allows numbers to be written on top of the stack, where StackRemove 0 and 1 remove either the top or second from top respectively.
- ALU – A and B are the two numbers to go through the ALU and be applied to one of its operations. Result is the outcome of how A and B are combined within the ALU. ALU Control is what decides which operation the ALU with preform. “zero?” and “negative?” are flags outputs that are used to tell whether the processor should branch or not.

List of the RTL symbols that will be implemented

- PC
- Memory
- Stack Memory
- ALU
- Sign Extend
- Shift Left 16 (SL)

Clock Cycle Specification List

- PC saves on the rising edge
- Memory saves on the falling edge
- IR saves on the rising edge
- ALU saves on the rising edge
- ALUOut saves on the rising edge
- Stack saves on the falling edge

Control Signal Description

- Write – The control bit that does two things. One, it allows a value to written onto a stack. Two, it allows a value to be written onto memory.
- Read – The control bit that does two things, One, it allows a value to be just read from stack. Two, it allows a value to be read from memory.
- Swap – While a flag in the instruction, it is still important to bring up. If the value of swap is 0, then value A is sp0 and value B is sp1. If the value of swap is 1, then A is sp1 and B is sp0.
- Op – The control bit that determines which operation to execute within the ALU.
- Remove 0 – The control bit the determines whether sp0 is removed from the stack. If value is 0, then sp0 is kept on the stack. If value is 1, then sp0 is removed.
- Remove 1 – The control bit the determines whether sp1 is removed from the stack. If value is 0, then sp1 is kept on the stack. If value is 1, then sp1 is removed.

Integration Plan

- PC: For every time an instruction is ran, the value of the program counter will be taken and added with 2. Once done, it will travel the wire straight back to PC register.
- Memory Block: The component that takes the PC value and outputs the corresponding instruction.
- Instruction Register: The component that divides the instruction value from the memory block and divides it into the corresponding values. In this case, bit 2 is flag isUp, bit 10 is flag swap, bit 10:8 is SHAMT, and bit 10:3 is the immediate value.
- Stack: The component that acts as the storage for all data values. Since every data value exists on it, there is no need for register number inputs. There will be a control value that will tell what instruction to execute and the stack will remove values accordingly. One input that does exist is when a value is placed back onto the stack.
- ALU: The component will take in three values: value A, value B, and the op value. Inside the ALU will be a number of logic gates that will allow for the correct op to execute.

Integration Plan Tests

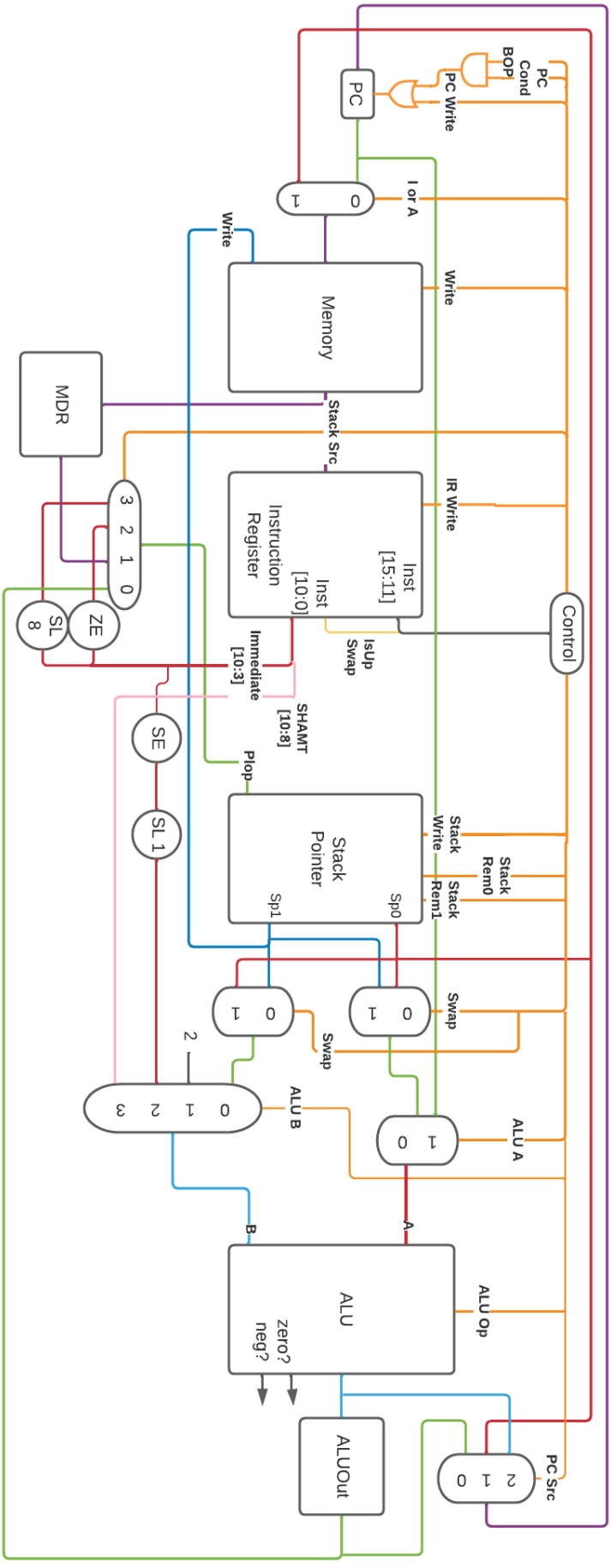
- For this there is no official plan for the tests. However, these steps are a maybe.
    - To truly test PC, two things can be done. One, just running a normal A-type like add. This will verify if PC+2 works. Two, somewhere within the test, a jump can test if PC will change accordingly.
    - To test the memory, two tests can be done. Both push and pop will require the memory in some way. For instance, if a value on memory wants to be pushed onto the stack, the read control will be enabled, and the value will travel to the stack to be “plopped” onto it. For pop, the test will be very similar. However, the write control will be enabled, and the value will be stored onto memory.
    - To test the instruction block, any instruction type will be perfect for the test. It would probably be best to test each instruction type, that being the P-type, A-type, and S-type.
    - To test the stack, since all other tests must use the stack to store data, it will be tested over the course.
    - To test the ALU, any A-type will satisfy the needs. It would probably be best if all instructions that used the ALU were tested.
    - THINGS TO NOTE: The crucial thing to remember when writing these tests is the exceptions that have been forgotten. Test benches will be created when these arise.

Changes made to the Assembly language and Machine language specifications

- Assembly
    - Made J, Jal, Peek, Pop, Pusha into A-types to account for the way we have to load 16 addresses inside of the stack by combining/ORing 2 8-bit numbers
    - Pusha now loads the integer value from the address on top of the stack, instead of loading an address from memory
    - Peek and Pop now look at the top two spots on the stack as addresses to where to save to and the number to save to, from top to bottom respectively. Peek keeps only the integer on the stack, and Pop removes both, the integer and address
    - J and Jal now look at the top of the stack for the address to go to and now take a full address location from the stack rather than being relative to PC
    - Pushi is now Push and now instead is used to push 8 bit numbers onto the stack and make top halves of address using the flag mentioned in the Machine Code section.
- Machine
    - All types are now 2 bytes which 5 bits are op code
    - P-type has an additional 8 bit immediate, a 1 bit flag to tell if to use it as an upper or not, and 2 dead bits
    - A-type has removed the 2 bits to indicate if the number is an immediate or an address, so now there are 10 dead bits
    - S-type has swapped the position of its SHAMT and immediate, so that it has more focus on the SHAMT rather than the immediate, since the SHAMT is the main use of S-type
    - Updated the Operation descriptions to show how flags affect some operations

Timings and Runtimes

- Cycle Time: 42.362 ns
- Number of Cycles with 0x13B0: 1,254,698
- Execution Time: 0.053151516676 seconds


PC
Cond
BOP
PC Write
I or A
Write
Memory
Write
MDR
Stack Src
IR Write
Instruction Register
Inst [15:11]
Inst [10:0]
IsUp
Swap
Control
Immediate [10:3]
SHAMT [10:8]
SE
SL 1
Plop
Stack Write
Stack Rem0
Stack Rem1
Stack Pointer
Sp1
Sp0
Swap
Swap
ALU B
ALU A
A
B
ALU
zero?
neg?
ALU Op
ALUOut
PC Src
3 2 1 0
SL 8
ZE