Team V

Diet Vilk

Memory to Memory Architecture

Garrett Doolittle, Ethan Hutton,
Wesley Schuh, and Ethan Townsend

Table of Contents

Description

Our 16-bit processor uses a memory to memory architecture capable of running a variety of general purpose programs. A memory to memory architecture was chosen because of its efficiency in the number of instructions used to complete operations when compared to other architectures. The design uses 7 registers, none of which are accessible to programs. These registers allow for smooth multi-cycle operation of the processor.

Green Sheet

Inst	T	Op	Usage	Logical Description	Description (verbose)
add	a	00	add dest s1 s2	Mem[dest] = Mem[s1] + Mem[s2]	Stores the sum of s1 and s2 into dest
sub	a	60	sub dest s1 s2	Mem[dest] = Mem[s1] - Mem[s2]	Stores the difference of s1 and s2 into dest
or	a	01	or dest s1 s2	Mem[dest] = Mem[s1] | Mem[s2]	Stores bitwise s1 OR s2 into dest
and	a	02	and dest s1 s2	Mem[dest] = Mem[s1] & Mem[s2]	Stores bitwise s1 AND s2 into dest
xor	a	04	xor dest s1 s2	Mem[dest] = Mem[s1] ^ Mem[s2]	Stores bitwise s1 XOR s2 into dest
srl	a	3e	srl dest s1 s2	Mem[dest] = Mem[s1] >> Mem[s2]	Stores s1 shifted right by s2 into dest
sll	a	3c	sll dest s1 s2	Mem[dest] = Mem[s1] << Mem[s2]	Stores s1 shifted left by s2 into dest
sri	ai	38	sri dest s1 imm	Mem[dest] = Mem[s1] >> imm	Stores s1 shifted right by imm into dest
sili	ai	7c	sili dest s1 imm	Mem[dest] = Mem[s1] << imm	Stores s1 shifted left by imm into dest
addi	ai	80	addi dest s1 imm	Mem[dest] = Mem[s1] + imm	Stores the sum of s1 and imm into dest
andi	ai	82	andi dest s1 imm	Mem[dest] = Mem[s1] & imm	Stores bitwise s1 AND imm into dest
ori	ai	81	ori dest s1 imm	Mem[dest] = Mem[s1] | imm	Stores bitwise s1 OR imm into dest
xori	ai	84	xori dest s1 imm	Mem[dest] = Mem[s1] ^ imm	Stores bitwise s1 XOR imm into dest
j	j	a0	j label	PC = label	Sets the PC to the given value
ja	ja	20	ja s1	PC = Mem[s1]	Sets the PC to the value at the given address
push	ja	5f	push s1	SP -= 2; stack[SP] =Mem[s1]	Pushes the value at s1 onto the stack
pop	ja	5e	pop s1	Mem[s1] = stack[SP] SP += 2	Pops the value at the top of the stack into s1
beq	br	30	beq dest s1 s2	if(Mem[s1] == Mem[s2]): PC = dest	Sets the PC to dest if s1 is equal to s2
blt	br	3a	blt dest s1 s2	if(Mem[s1] < Mem[s2]): PC = dest	Sets the PC to dest if s1 is less than s2
bne	br	39	bne dest s1 s2	if(Mem[s1] != Mem[s2]): PC = dest	Sets the PC to dest if s1 is not equal to s2
bge	br	34	bge dest s1 s2	if(Mem[s1] >= Mem[s2]): PC = dest	Sets the PC to dest if s1 is greater than or equal to s2
beqi	bri	b0	beqi dest s1 imm	if(Mem[s1] == imm): PC = dest	Sets the PC to dest if s1 is equal to imm
blti	bri	ba	blti dest s1 imm	if(Mem[s1] < imm): PC = dest	Sets the PC to dest if s1 is greater than imm
bnei	bri	b9	bnei dest s1 imm	if(Mem[s1] != imm): PC = dest	Sets the PC to dest if s1 is not equal to imm
bgei	bri	b4	bgei dest s1 imm	if(Mem[s1] >= imm): PC = dest	Sets the PC to dest if s1 is greater than or equal to imm
bgti	bri	a4	bgti dest s1 imm	if(Mem[s1] > imm): PC = dest	Sets the PC to dest if s1 is greater than imm
blei	bri	ac	blei dest s1 imm	if(Mem[s1] <= imm): PC = dest	Sets the PC to dest if s1 is less than or equal to imm
pushra	sp	7f	pushra	SP -= 2; stack[SP] = PC + 16	Pushes the return address onto the stack
mov	l	bf	mov dest imm	mem[dest] = imm	Sets the value at the dest memory address to be imm

Performance Testing

This design was tested using a combination of unit tests on individual components and overall system tests written in Verilog and executed in modelsim. Final performance figures were concluded from the execution of a given relative prime algorithm.

Instruction Types & Blocking

All instructions are specified using an 8-bit opcode which is frequently represented in hex.

A - arithmetic instructions. 16 bit result address, and 2 16 bit addresses to pull the values that need to be computed. Things like shift left, shift right, shift right arithmetic, add, sub, xor, and, and or use this type of instruction.

7 bytes	16	16	16	8
A:	**dest**	**s2**	**s1**	**opcode**

AI - Any instructions with an immediate and a register that need to be computed together. 16 bit result address, 16 bit address to pull the value and a 16 bit immediate. Includes addi, andi, xori, ori.

7 bytes	16	16	16	8
AI:	**dest**	**imm**	**s1**	**opcode**

BR - Branch instructions utilizing a 16 bit Label, and 2 16 bit addresses. The BR instructions will compare the value stored in memory at address s1 with the value at address s2 and either branch to a given tag or just move on to the next line in the newPC. Instructions of this type are blt, bge, bne, and beq.

7 bytes	16	16	16	8
BR:	**dest**	**s2**	**s1**	**opcode**

BRI - Similar to BR instructions, take a 16 bit immediate, a 16 bit label, and a 16 bit address. The BR instructions compare an immediate with the value stored in memory and either branch to a given tag or just move on to the next instruction. Includes beqi, bnei, blti, bgei, bgti, blei.

7 bytes	16	16	16	8
BRI:	**dest**	**imm**	**s1**	**opcode**

J - jump instructions that will take a 16 bit label. It unconditionally jumps to the given label.

3 bytes	16	8
J:	**label**	**opcode**

JA - jump instructions that will take a 16 bit address. It jumps unconditionally to the location stored in the given address.

3 bytes	16	8
JA:	**s1**	**opcode**

SP - Instruction type used to push and pop addresses on the stack. Used for the push and pop.

1 byte	8
SP:	**opcode**

L - Instruction type used to to load an immediate into memory. Loads a 16 bit immediate into a 16 bit destination address.

5 bytes	16	16	8
L:	**dest**	**imm**	**opcode**

Addressing Modes

The processor uses direct addressing.

Calling Conventions

Before calling a function, push any values that are desired to be saved across the function call, then push the input arguments on the stack and then push the return address. Note that pushra is used right before jumping. If this convention is not followed, unintended function calls may occur. Within the called function, pop off the return address and input arguments. In preparation to return, push the return values onto the stack before jumping to the return address that was given with the function call. The caller then pops off any return values and the stack should be restored to how it was before the call.

ISA

Inst	Type	opcode	Usage	Logical Description	Description (verbose)
add	a	00000000 = 0x00	add dest s1 s2	Mem[dest] = Mem[s1] + Mem[s2]	Stores the sum of s1 and s2 into dest
sub	a	01100000 = 0x60	sub dest s1 s2	Mem[dest] = Mem[s1] - Mem[s2]	Stores the difference of s1 and s2 into dest
or	a	00000001 = 0x01	or dest s1 s2	Mem[dest] = Mem[s1] | Mem[s2]	Stores bitwise s1 OR s2 into dest
and	a	00000010 = 0x02	and dest s1 s2	Mem[dest] = Mem[s1] & Mem[s2]	Stores bitwise s1 AND s2 into dest
xor	a	00000100 = 0x04	xor dest s1 s2	Mem[dest] = Mem[s1] ^ Mem[s2]	Stores bitwise s1 XOR s2 into dest
srl	a	00111110 = 0x3e	srl dest s1 s2	Mem[dest] = Mem[s1] >> Mem[s2]	Stores s1 shifted right by s2 into dest
sll	a	00111100 = 0x3c	sll dest s1 s2	Mem[dest] = Mem[s1] << Mem[s2]	Stores s1 shifted left by s2 into dest
sri	ai	00111000 = 0x38	sri dest s1 imm	Mem[dest] = Mem[s1] >> imm	Stores s1 shifted right by imm into dest
sili	ai	01111100 =0x7c	sili dest s1 imm	Mem[dest] = Mem[s1] << imm	Stores s1 shifted left by imm into dest
addi	ai	10000000 = 0x80	addi dest s1 imm	Mem[dest] = Mem[s1] + imm	Stores the sum of s1 and imm into dest

andi	ai	10000010 = 0x82	andi dest s1 imm	Mem[dest] = Mem[s1] & imm	Stores bitwise s1 AND imm into dest
ori	ai	10000001 = 0x81	ori dest s1 imm	Mem[dest] = Mem[s1] | imm	Stores bitwise s1 OR imm into dest
xori	ai	10000100 = 0x84	xori dest s1 imm	Mem[dest] = Mem[s1] ^ imm	Stores bitwise s1 XOR imm into dest
j	j	10100000 = 0xa0	j label	PC = label	Sets the PC to the given value
ja	ja	00100000 = 0x20	ja s1	PC = Mem[s1]	Sets the PC to the value at the given address
push	ja	01011111 = 0x5f	push s1	SP -= 2; stack[SP] =Mem[s1]	Pushes the value at s1 onto the stack
pop	ja	01011110 = 0x5e	pop s1	Mem[s1] = stack[SP] SP += 2	Pops the value at the top of the stack into s1
beq	br	00110000 = 0x30	beq dest s1 s2	if(Mem[s1] == Mem[s2]) PC = dest	Sets the PC to the given value if s1 is equal to s2
blt	br	00111010 = 0x3a	blt dest s1 s2	if(Mem[s1] < Mem[s2]) PC = dest	Sets the PC to the given value if s1 is less than s2
bne	br	00111001 = 0x39	bne dest s1 s2	if(Mem[s1] != Mem[s2]) PC = dest	Sets the PC to the given value if s1 is not equal to s2
bge	br	00110100 = 0x34	bge dest s1 s2	if(Mem[s1] >= Mem[s2]) PC = dest	Sets the PC to the given value if s1 is greater than or equal to s2
beqi	bri	10110000 = 0xb0	beqi dest s1 imm	if(Mem[s1] == imm) PC = dest	Sets the PC to the given value if s1 is equal to imm
blti	bri	10111010 = 0xba	blti dest s1 imm	if(Mem[s1] < imm) PC = dest	Sets the PC to the given value if s1 is greater than imm
bnei	bri	10111001 = 0xb9	bnei dest s1 imm	if(Mem[s1] != imm) PC = dest	Sets the PC to the given value if s1 is not equal to imm
bgei	bri	10110100 = 0xb4	bgei dest s1 imm	if(Mem[s1] >= imm) PC = dest	Sets the PC to the given value if s1 is equal to or equal to imm
bgti	bri	10100100 = 0xa4	bgti dest s1 imm	if(Mem[s1] > imm) PC = dest	Sets the PC to the given value if s1 is

					greater than imm
blei	bri	10101100 = 0xac	blei dest s1 imm	if(Mem[s1] <= imm) PC = dest	Sets the PC to the given value if s1 is less than or equal to imm
pushra	sp	01111111 = 0x7f	pushra	SP -= 2; stack[SP] = PC + 16	Pushes the return address of a subsequent method call onto the stack
mov	I	10111111 = 0xbf	mov dest imm	mem[dest] = imm	Sets the value at the dest memory address to be the given immediate imm

Memory Map

Output Address	F010
Input Address	FFFE
Stack space (SP)	FFFD
Data (variable space)	F012
Program (text) (PC)	0101-F00E
Boot sector	0000-0100

Input/Output

The processor handles i/o operation though the use of reserved memory addresses. The memory module is contained within an io wrapper that diverts the signals when necessary.

Code Fragments

Basic iteration to 10 with a conditional used as a while loop:

Address	Assembly	Machine code	Comments
0x0000	mov m, 0	0xFF00 0000 BF	//a set instruction
0x0005	L: addi m, m, 1	0xFF00 0001 FF00 80	//an addi instruction with a label
0x000C	bnei L, m, 10	0x0005 000A FF00 BF	//a comparison instruction

If…else:

Address	Assembly	Machine code	Comments
0x0000	IF: bnei ELSE, m, 0	000E 0000 FF00 B9	//go to ELSE if m != 0
0x0007	addi c, c, 1	FF02 0001 FF02 80	//if no branch (m = 0), c += 1
0x000E	ELSE:addi c, c, -1	FF02 FFFF FF02 80	//if branch (m != 0), c -= 1

Basic swap:

Address	Assembly	Machine code	Comments
0x0000	addi c, a, 0	FF04 0000 FF00 80	//set temp c to value of a
0x0007	addi a, b, 0	FF00 0000 FF02 80	//set a to value of b
0x000E	addi b, c, 0	FF02 0000 FF04 80	//set b to temp c (value of a)

Basic function call with an input value and return value in a:

Address	Assembly	Machine code	Comments
0x0000	push a	FF00 5E	//saves the value at a on the stack
0x0003	pushra	7F	//pushes the PC of the instruction after a jump onto the stack
0x0004	j function	00DD A0	//jumps to the function
0x000B	pop a	FF00 5F	//pops the value on the top of the stack into a

Basic recursive addition:

Address	Assembly	Machine code	Comments
0x0000	FAD: pop ra	FF00 5E	// pops the value on the stack into mem[ra]
0x0003	pop m	FF02 5E	// pops the value on the stack into mem[m]
0x0006	pop tot	FF04 5E	// pops the value on the stack into mem[tot]
0x0009	bgti DB, m, 0	0016 FF02 0000 A4	//if (m > 0) Go to DB
0x0010	push tot	FF04 5F	//pushes the value at mem[tot] onto stack
0x0013	ja ra	FF00 20	//a jump that goes to the address stored in mem[ra]
0x0016	DB: add tot, tot, m	FF04 FF02 FF04 00	//Set mem[tot] to mem[tot] + mem[m]. With a DB label
0x001D	addi m, m, -1	FF02 FFFF FF02 80	//Set mem[m] to mem[m] - 1
0x0024	push tot	FF04 5F	//pushes the value at mem[tot] onto stack
0x0027	push m	FF02 5F	//pushes the value at mem[m] onto stack
0x002A	pushra	7F	//pushes the PC of the instruction after a jump onto the stack
0x002B	j FAD	00DD A0	A unconditional jump to FAD
0x003E	ja ra	FF00 20	//a jump that goes to the address stored in mem[ra]

RelPrime

Address	Assembly	Machine code	Comments
0x0000	pop ra	FF00 5E	// pops the value on the stack into mem[ra]
0x0003	pop n	FF02 5E	// pops the value on the stack into mem[n]
0x0006	mov m, 2	FF04 0002 BF	//a set instruction
0x000b	Loop:push n	FF02 5F	//pushes the value at mem[n] onto stack
0x000e	push m	FF04 5F	//pushes the value at mem[m] onto stack
0x0011	pushra	7F	//pushes the PC of the instruction after a jump onto the stack
0x0012	j gcd	00CA A0	//a jump instruction to gcd
0x0015	pop c	FF06 5E	// pops the value on the stack into mem[c]
0x0018	beqi L, c, 1	0018 FFFF FF06 b0	//a comparison instruction
0x001f	addi m, m, 1	FF04 0001 FF04 80	//Set mem[m] to mem[m] + 1. with a label
0x0026	J Loop	000a a0	// jump instruction to the Loop tag
0x0029	L:push m	FF04 5F	//pushes the value at mem[m] onto stack
0x002c	ja ra	FF00 7E	//a jump that goes to the address stored in mem[ra]

GCD

Address	Assembly	Machine code	Comments
0x0000	pop ra	FF08 5E	// pops the value on the stack into mem[ra]
0x0003	pop b	FF0a 5E	// pops the value on the stack into mem[b]
0x0006	pop a	FF0c 5E	// pops the value on the stack into mem[a]
0x0009	bnei LOOP, a, 0	0016 0000 FF0c B9	//if (a != 0) Go to LOOP
0x0010	push b	FF0a 5F	//pushes the value at mem[b] onto stack
0x0013	ja ra	FF08 20	//a jump that goes to the address stored in mem[ra]
0x0016	LOOP: bge ELSE, b, a	0051 FF0c FF0a 34	//if (b >= a) Go to ELSE. with a LOOP label
0x001D	sub a, a, b	FF0c FF0c FF0c 60	//Set mem[a] to mem[a] - mem[b]
0x0024	j NOEL	FF08 a0	A unconditional jump to NOELSE
0x0027	ELSE: sub b, b, a	FF0a FF0c FF0a 60	//Set mem[b] to mem[b] - mem[a]. With a ELSE label
0x002E	NOEL: bnei LOOP, b, 0	0016 0000 FF0a B9	//if (b != a0 Go to LOOP. with a NOELSE label
0x0035	push a	FF0c 5F	//pushes the value at mem[a] onto stack
0x0038	ja ra	FF08 20	//a jump that goes to the address stored in mem[ra]

Register Transfer Language Specifications

A	**AI**	**BR**
OP = Mem[PC] PC = PC + 1		
A = Mem[PC] PC = PC + 2		
B = Mem[PC] PC = PC + 2	B = Mem[PC] PC = PC + 2	B = Mem[PC] PC = PC + 2
A = Mem[A]	A = Mem[A]	A = Mem[A]
B = Mem[B]	aluOut = (A op B) Dest = Mem[PC]	B = Mem[B]
aluOut = (A op B) Dest = Mem[PC]	Mem[Dest] = aluOut PC = PC + 2	Dest = Mem[PC] PC = PC + 2
Mem[Dest] = aluOut PC = PC + 2		if(A op B) PC = Dest

BRI	**J**	**JA**	**JA Push**
OP = Mem[PC] PC = PC + 1			
A = Mem[PC] PC = PC + 2			
B = Mem[PC] PC = PC + 2	PC = A	A = Mem[A]	SP = SP - 2 B = Mem[A]
A = Mem[A]		PC = A	Mem[SP] = B
Dest = Mem[PC] PC = PC + 2			
if(A op B) PC = Dest			

JA Pop	SP	L
OP = Mem[PC] PC = PC + 1		
A = Mem[PC] PC = PC + 2		
B = Mem[SP]	SP = SP - 2	B = Mem[PC] PC = PC + 2
SP = SP + 2 Mem[A] = B	A = PC + 1	Mem[B] = A
	Mem[SP] = A PC = PC - 2	

Component Descriptions

Component	Inputs	Outputs	Behavior	RTL Symbols
Register (16bit)	inputValue [15:0] CLK [0:0] reset [0:0]	outputValue [15:0]	On the rising edge of the clock (CLK) the register reads the input_value port and outputs the new value. If the reset signal is high on the rising clock edge the register wipes its data and outputs 0x0000 instead.	PC, SP, A, B, IR,aluOut, Dest
Register (8bit)	inputValue [7:0] CLK [0:0] reset [0:0]	outputValue [7:0]	Similar to the 16 bit register, on the rising edge of the clock, the register will read the inputValue and output a new value. The reset signal will input 0x0000. This will currently only be used to read the opcode.	OP
Memory	CLK [0:0] memWrite [0:0] inputAddress [15:0] inputValue [15:0]	outputValue [15:0]	On the rising edge of the clock, the memory reads the memWrite and if it is 1, it will write the value given at inputValue into the address at inputAddress, if memWrite is not 1 (it is 0), the memory will output the value in memory at the address given by inputAddress to outputAddress.	Mem
ALU	inputA [15:0] inputB [15:0] ALUOp [2:0]	outputValue [15:0]	The ALU takes in two 16 bit immediate values and does the operation indicated by the opcode and outputs a 16 bit immediate.	(A op B)

For ALUOp values, see Control Signals (Page 21)

Naming Conventions

Functional modules: [component name]
Testing modules: [component name]_tb
Clocks: CLK[number (if applicable)]
Registers: [alphanumeric identifier]
Memory wires: [alphanumeric identifier]
Wires: [whatever you want] (fun required)
Inputs: input[alphanumeric identifier]
Outputs: OutputValue

Filenames and Testbenches

All files are found inside the /implementation folder of the repository.

Module	Testbench
memory.v	memorytest.v
register8.v	register8_tb.v
register16.v	register16_tb.v
alu.v	alu_tb.v
control.v	control_tb.v

Testing RTLs

A	**AI**	**BR**
add dest, s1, s2 Assume dest is stored in memory at address 0xFF00 and contains the value 0, s1 at 0xFF02 containing value 2, s2 at 0xFF04 containing value 3	addi dest, s1, 1 Assume dest is stored in memory at address 0xFF00 and contains the value 0, s1 at 0xFF02 containing value 2	beq LABEL, s1, s2 Initialized values: PC = 0 s1 = 0xFF00 s2 = 0xFF02 Label = 0x000A
IR = Mem[PC] PC = PC + 1 IR now contains the opcode 00 (plus unused bits 04). PC is incremented past the opcode. PC -> 0xFF00 FF04 FF02 | 00	IR = Mem[PC] PC = PC + 1 IR now contains the opcode 80 (plus unused bits 02). PC is incremented past the opcode PC -> 0xFF00 0001 FF02 | 80	Cycle 1 (IR = Mem[PC] PC = PC + 1): IR = 0x0030 PC = 1
A = Mem[PC] PC = PC + 2 A now contains the address	A = Mem[PC] PC = PC + 2 A now contains the address	Cycle 2 (A = Mem[PC] PC = PC + 2): IR = 0x0030

for s1, 0xFF02. PC is incremented past s1. PC -> 0xFF00 FF04 | FF02 00	for s1, 0xFF02. The PC is incremented past s1. PC -> 0xFF00 0001 | FF02 80	A = 0xFF00 PC = 3
B = Mem[PC] PC = PC + 2 B now contains the address for s2, 0xFF04. PC is incremented past s2. PC -> 0xFF00 | FF04 FF02 00	B = Mem[PC] PC = PC + 2 B now contains the immediate value (1). The PC is incremented past the imm. PC -> 0xFF00 | 0001 FF02 80	Cycle 3 (B = Mem[PC] PC = PC + 2): IR = 0x0030 A = 0xFF00 B = 0xFF02 PC = 5
B = Mem[B] The value of s2 (3) is loaded into B.	A = Mem[A] The value of s1 (2) is loaded into A.	Cycle 4 (A = Mem[A]): IR = 0x0030 A = 5 B = 0xFF02 PC = 5
A = Mem[C] The value of s1 (2) is loaded into A.	ALUOut = A + B destAddr = Mem[PC] The values of s1 and the immediate are added and stored into the variable ALUOut. destAdder now contains the address for the sum, 0xFF00.	Cycle 5 (B = Mem[B]): IR = 0x0030 A = 5 B = 5 PC = 5
ALUOut = A + B destAddr = Mem[PC] The values of s1 and s2 are added and stored into the variable ALUOut. destAdder now contains the address for the sum, 0xFF00.	Mem[destAddr] = ALUOut PC = PC + 2 The value of ALUOut (3) is stored at destAdder's address in memory (0xFF00). The PC is incremented past destAdder to the next instruction. PC -> | 0xFF00 0001 FF02 80	Cycle 6 (Dest = Mem[PC] PC = PC + 2): IR = 0x0030 A = 5 B = 5 Dest = 0x000A PC = 7
Mem[destAddr] = ALUOut The value of ALUOut (5) is stored at destAdder's address in memory (0xFF00). The PC is incremented past destAdder to the next instruction. PC -> | 0x00FF 00FE 00FD 00		Cycle 7 (if(A op B) PC = Dest): Since (A = B): IR = 0x0030 A = 5 B = 5 Dest = 0x000A PC = 0x000A
		Result A = B so PC is now at the Dest Label

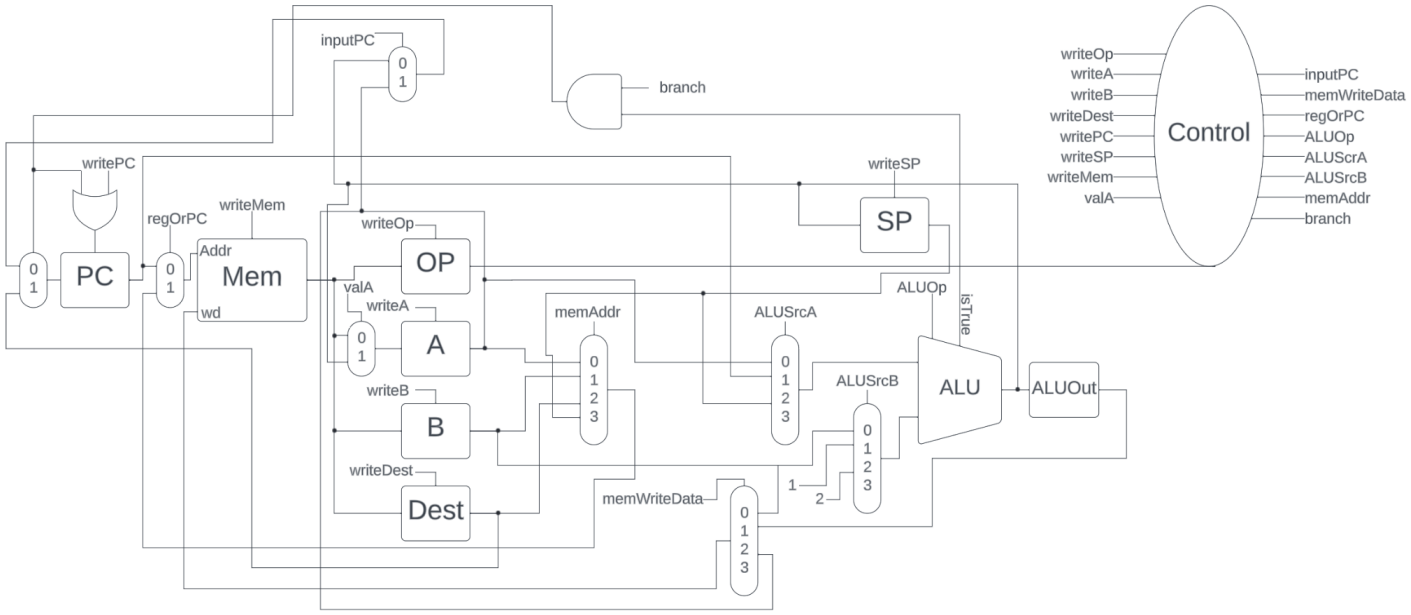
BRI	**J**	**JA**	**JA Push**	**JA Pop**
beqi LABEL, s1, imm Initialized values: PC = 0 s1 = 0xFF00 Imm = 5 Label = 0x000A	j label Initialize: PC = 0, Label = 0x0001	ja s1 Initialize: PC = 0, s1 = 0x0001	push s1 Initialize: PC = 0, s1 = 0x0001 sp = 0	pop s1 Initialize: PC = 0, s1 = 0x0001 sp = -2
Cycle 1 (IR = Mem[PC] PC = PC + 1): IR = 0x00B0 PC = 1	Start: IR = Mem[PC] IR is the opcode. PC = PC + 1 PC = 1 Increment the PC to the next address to grab	Start: IR = Mem[PC] IR is the opcode. PC = PC + 1 PC = 1 Increment the PC to the next address to grab	Start: IR = Mem[PC] IR is the opcode. PC = PC + 1 PC = 1 Increment the PC to the next address to grab	Start: IR = Mem[PC] IR is the opcode. PC = PC + 1 PC = 1 Increment the PC to the next address to grab
Cycle 2 (A = Mem[PC] PC = PC + 2): IR = 0x00B0 A = 0xFF00 PC = 3	A = Mem[PC] This is the label PC = PC + 2 PC = 3 Increment the PC to the next address	A = Mem[PC] This is s1 which is a given address A = s1 PC = PC + 2 PC = 3 Increment the PC to the next address	A = Mem[PC] This is s1 which is a given address A = s1 PC = PC + 2 PC = 3 Increment the PC to the next address	A = Mem[PC] This is s1 which is a given address A = s1 PC = PC + 2 PC = 3 Increment the PC to the next address
Cycle 3 (B = Mem[PC] PC = PC + 2): IR = 0x00B0 A = 0xFF00 B = 5 PC = 5	PC = label PC is now 0x0001 which is the given label	A = Mem[A] A is now the value stored in s1 which is the address we want to go to	sp = sp - 2 Now we decrement sp to make space on the stack sp = -2 B = Mem[A]	B = Mem[sp] We now load the value at sp onto B. let's say the value stored in B is 3. B = 3

			B is now the value stored at s1. let's say the value stored at s1 is 3. B = 3	
Cycle 4 (A = Mem[A]): IR = 0x00B0 A = 5 B = 5 PC = 5		PC = A PC is now A, the address we want to go to	Mem[sp] = B Now we store the the value on the stack. sp should still be -2, but the value stored in it is now 3. This means the value stored in s1 is now pushed onto the stack	sp = sp + 2 We move sp back to where it was before we allocated space. sp = 0 Mem[A] = B We now store the value of B into A which is s1. s1 is still the same address, however, now the value of 3 is stored inside of it. This means the value was popped off the stack and into s1.
Cycle 5 (Dest = Mem[PC] PC = PC + 2): IR = 0x00B0 A = 5 B = 5 Dest = 0x000A PC = 7				
Cycle 6 (if(A op imm) PC = Dest): Since (A = imm): IR = 0x00B0 A = 5 B = 5 Dest = 0x000A PC = 0x000A				
Result A = imm so				

PC is now at the Dest Label				

SP	**L**
Assume all relevant registers are initialized to 0	Assume all relevant registers are initialized to 0
IR = Mem[PC] PC = PC + 1 This instruction fills the IR with the 8 bit op code and 8 bits of useless information (we ignore those bits) The PC is incremented one address, (8 bits) rather than the usual two addresses (16 bits)	IR = Mem[PC] PC = PC + 1 This instruction fills the IR with the 8 bit op code and 8 bits of useless information (we ignore those bits) The PC is incremented one address, (8 bits) rather than the usual two addresses (16 bits)
A = Mem[PC] PC = PC + 2 The register A gets the value in memory at the PC, in this instruction this value is overwritten later on.	A = Mem[PC] PC = PC + 2 The register A gets the value in memory at the PC, this value is an immediate in this instruction type, it is 16 bits, so there is no need for sign extension.
SP = SP - 2 Allocates space on the stack by moving the stack pointer down 2 addresses (16 bits)	B = Mem[PC] PC = PC + 2 The register B gets the value in memory at the PC, in this type, B is an address in memory.
A = PC + 1 This sets A to the value of the instruction two after the current instruction. The next instruction is always a J type, which is 1 address long (3 bits). The PC is already set 2 addresses into the jump instruction at this stage, so incrementing it one more would set it to be the address after the jump instruction.	Mem[B] = A Sets the address stored in B to be the value stored in A.
Mem[SP] = A PC = PC - 2 This saves the value of A onto the stack and sets the PC back to the start of the jump instruction.	

Datapath



Control Signals

writeOp [0:0] - flag that determines whether or not to write to the OP register
writeA [0:0] - flag that determines whether or not to write to the A register
writeB [0:0] - flag that determines whether or not to write to the B register
writeDest [0:0] - flag that determines whether or not to write to the Dest register
writePC [0:0] - flag that determines whether or not to write to the PC register
writeSP [0:0] - flag that determines whether or not to write to the SP register
writeMem [0:0] - flag that determines whether or not to store a value into the given memory address
valA [0:0] - selector that determines if register A gets a value from memory or from the ALU
memWriteData [1:0] - selector that determines the data to be written to the given register

memWriteData	Selection
00	B
01	ALUOut
10	A

inputPC [0:0] - selector that determines what value to write to the PC
regOrPC [0:0] - selector that determines whether to load the PC or another address in memory

ALUOp [3:0] - selector that determines the operation the ALU will perform

ALUOp	Instruction (op)
0000	add +
0001	sub -
0010	or |
0011	and &
0100	xor ^
0101	sll <<
0110	srl >>
0111	beq ==
1000	bne !=
1001	blt <
1010	bge >=
1011	ble <=
1100	bgt >

ALUSrcA [1:0] - selector that determines the first input to the ALU

ALUSrcA	Selection
00	A
01	PC
10	SP

ALUSrcB [1:0] - selector that determines the second input to the ALU

ALUSrcB	Selection
00	B
01	1
10	2

memAddr [1:0] - selector that determines which address goes into the memory

memAddr	Selection
00	A
01	B
10	Dest
11	SP

branch [0:0] - flag that determines whether or not we are branching even if the isTrue from the ALU turns out to be true

Component Testing Plan

16 bit register: The functionality of the 16 bit register was tested using the following inputs: cycle the CLK every 50ns starting at a value of 0. reset = 1 from 0ns to 50 ns to clear the register. While the clock is alternating starting at 100 ns the input value is incremented by 1 every cycle. On every following cycle the register output is checked to ensure it is outputting the same value that was input in the previous cycle. After 128 cycles reset is set high again for one cycle and the register is checked to be outputting 0.

8 bit register: The functionality of the 8 bit register was tested using a CLK cycling every 50ns starting at a value of 0. The register is fed a high reset signal for the first 50ns of the test to ensure a fresh initial condition. Starting at 100 ns the input value was incremented by 1 every clock cycle. Each cycle, the output of the register is checked to ensure that the value is the same one it was fed in the previous cycle. After 128 cycles the reset signal is set high for one cycle and the register output is checked to be zero.

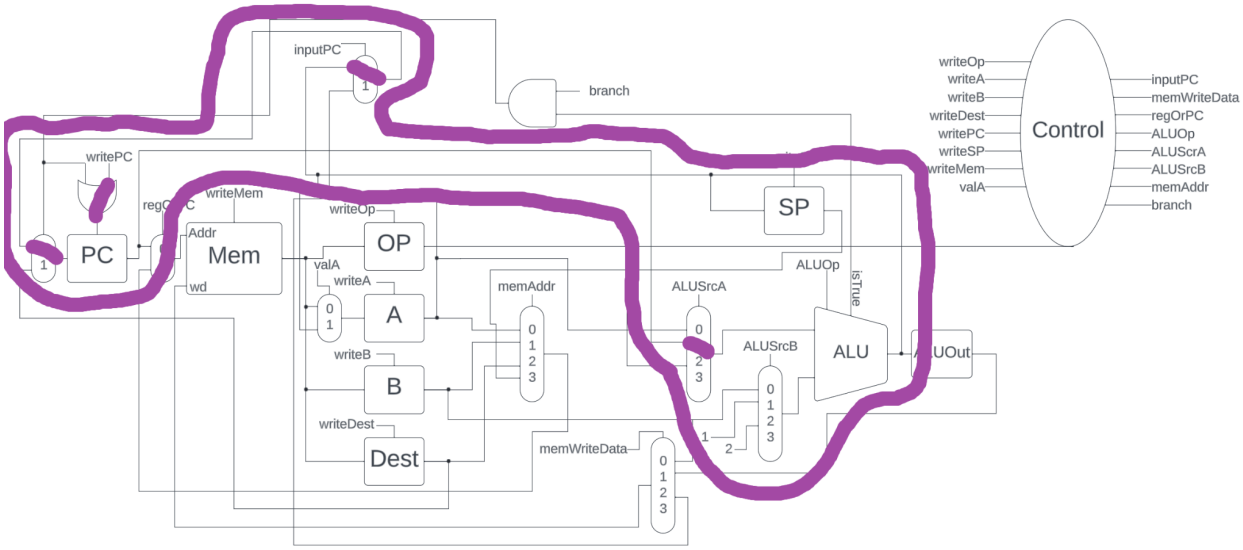
Memory: The functionality of the memory was tested using a CLK cycling every 50ns starting at a value of 0, then a value was chosen and set as the input for the memory. Then the input address is set to 0, and the memwrite signal is set to high. The input address is then incremented every clock cycle until the entire address space has been addressed once. Then memWrite is set to 0, the input value is changed to another value, and there is an arbitrary delay, before resetting the inputaddress and checking that all the memory addresses are storing the correct value.

ALU: The functionality of the ALU was tested by setting up at least 3 test cases for each operation. The operations were designed to test the edge cases of the operations they are testing as well as a middle/average case.

Iterative Implementation Plan

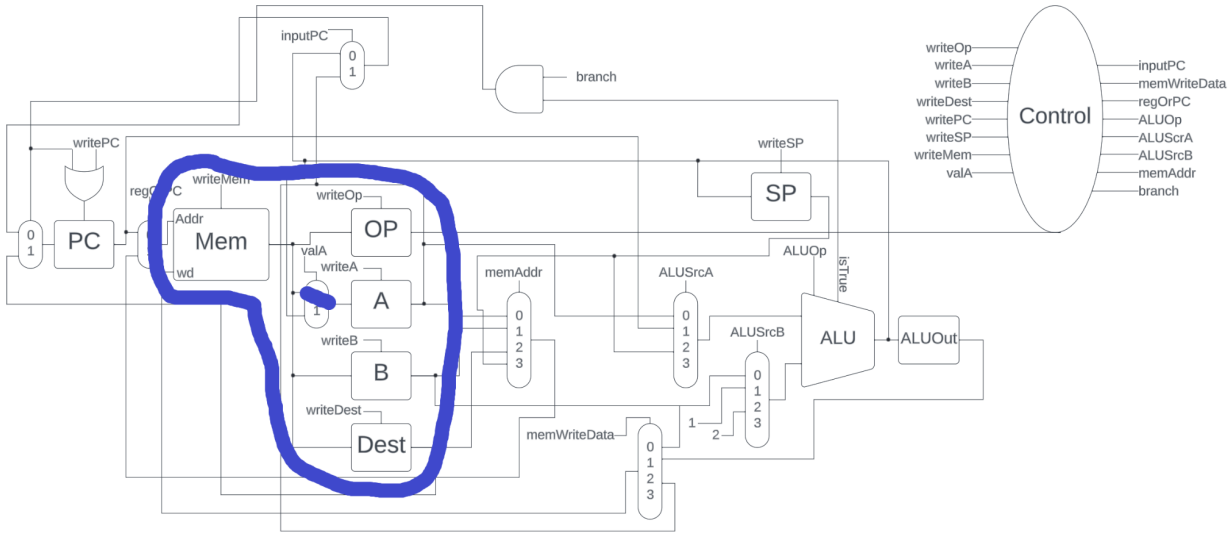
All stages are included in the github under /implementation/stage_X.v
Each stage has a corresponding testbench under /implementation/stage_X_tb.v

Stage 1



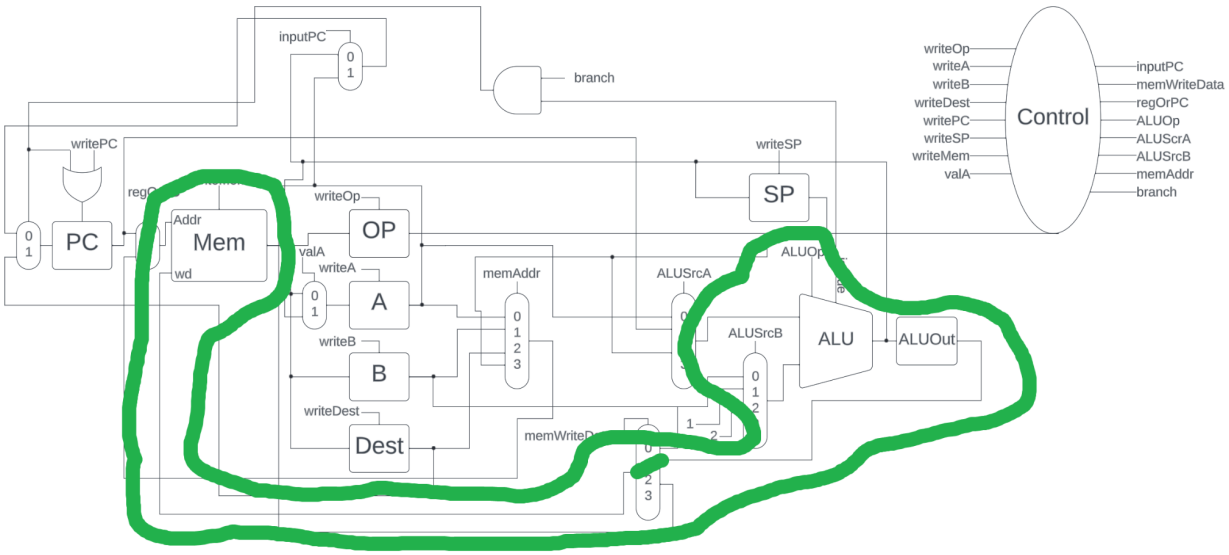
Integrate the PC register and the ALU.
Make sure the PC increments by 1 with the correct ALUOp and either decrements or increments by 2 with the correct ALUOp.

Stage 2



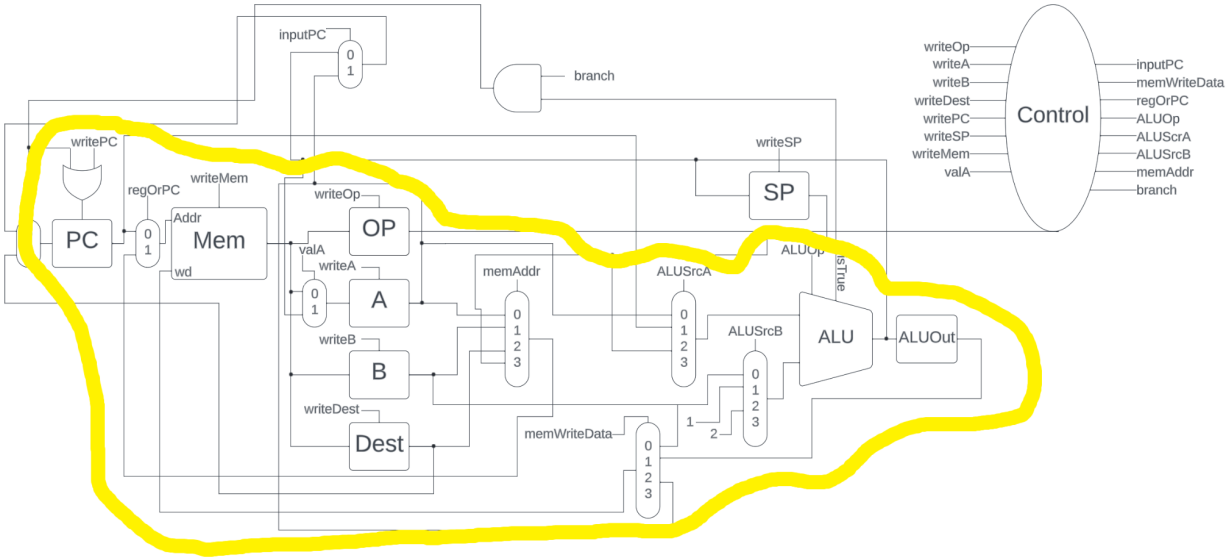
Integrate the memory into the system.
Load instruction/addresses from memory into a register. Ensure the OP register and A register have the correct opcode and address after the PC is incremented. Also make sure that the registers are not overwritten when other registers are written into.

Stage 3



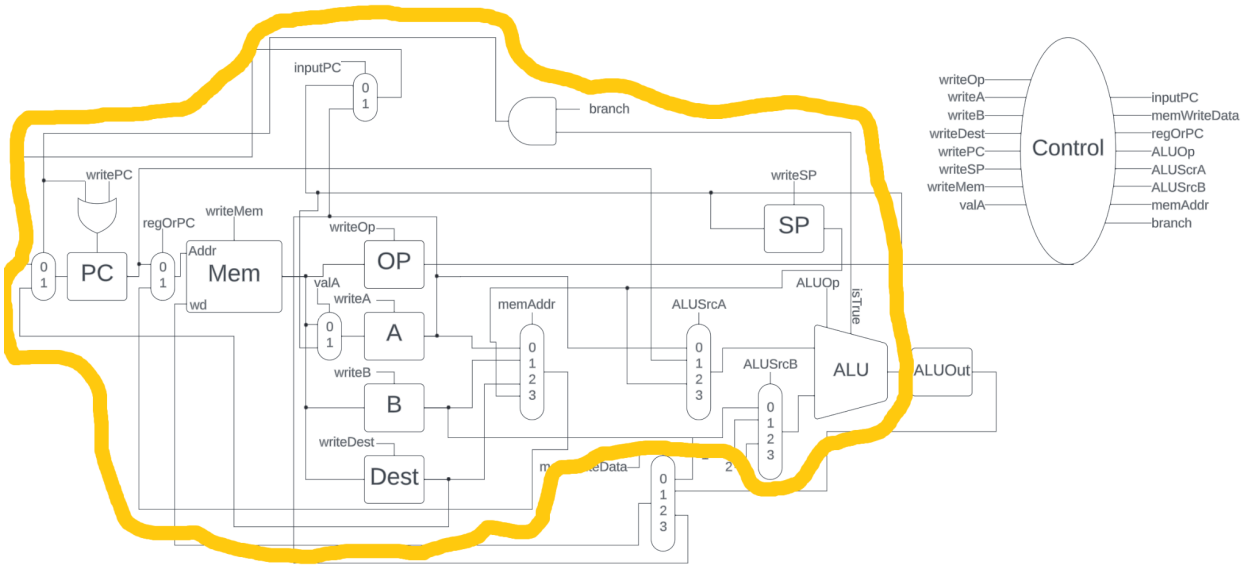
Verify ALU performs as expected.
Load two values into the ALU. Make sure the correct value is loaded into the ALUOut register according to ALUOp. Verify that value can be stored in memory.

Stage 4



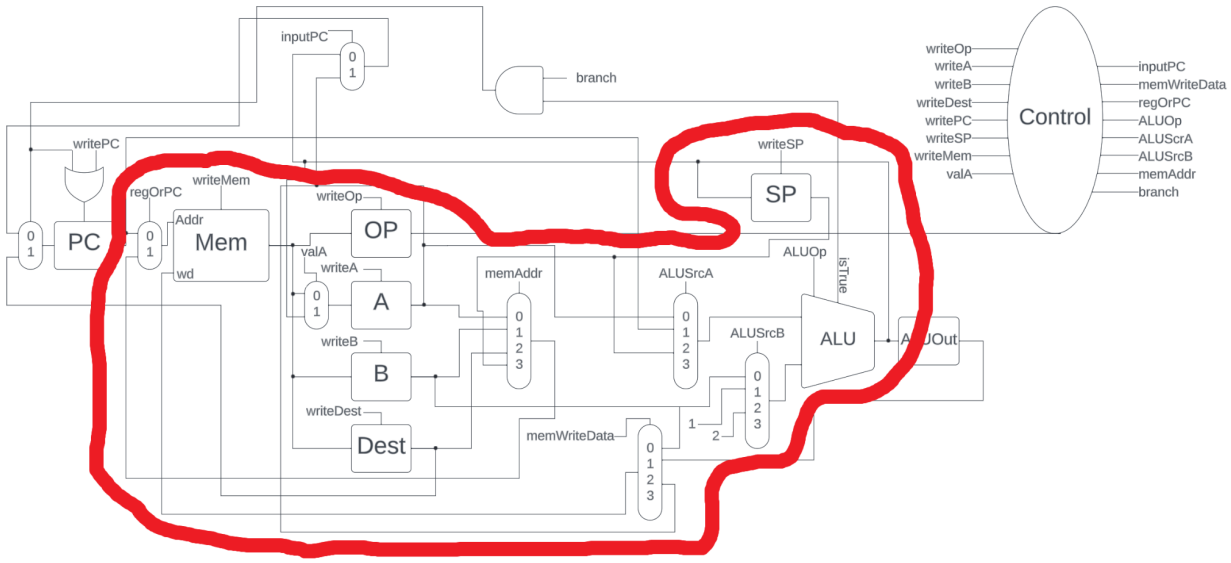
Connecting stages 1, 2, and 3 to make the A and AI type instruction.
Connect the PC to the address input of the memory component. Verify the OP register contains the correct opcode, and the registers A and B contain the desired addresses. Load the values of A and B into their respective registers if necessary. Load the Dest address into the Dest register and load that address into the address input for memory. Perform the required operation with the ALU, and write the value of ALUOut to Dest. Verify that Dest contains this new value.

Stage 5



Creating the BR and BRI instruction types.
The first thing we need to test for the ALU is the isTrue? part of it to ensure that branching will occur. This will have us compare two values and make sure that the comparison returns 1 on the designated branch instruction. To test this we just need to make up two values that are greater than, greater than or equal, less than, less than or equal, equal, or not equal. After this we need to make sure that the PC is only overwritten if we're branching. To do this we just have a branch and an and gate to make sure we are branching. This should ensure that only on a branch, we load the label in dest.

Stage 6

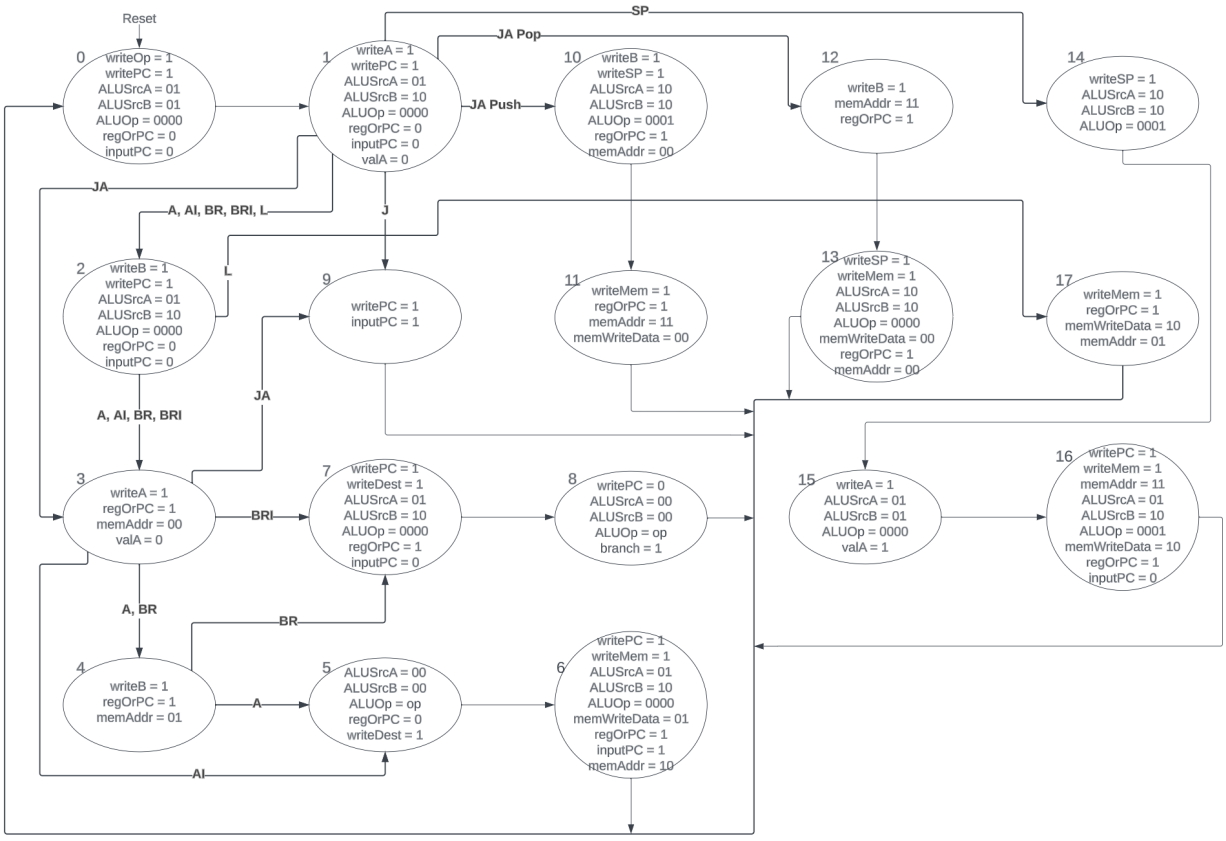


Integrate SP for the JA, SP, and L instruction types.
Use the ALUSrc selectors to increment/decrement SP by 2, and load the result back into SP.
Verify SP now contains the desired value.

Stage 7

At this point all the components should be implemented,this stage is to go back over and make sure all the instructions work as intended.

Finite State Machine



Control Unit Design

The control unit should be implemented in a fashion very similar to a standard multi-cycle control. The control module needs inputs for the opcode, the clock, and a reset, and outputs for each control signal. There will also be an integer variable for the current state. It then uses a switch case statement with input state on the rising edge of the clock to adjust certain signals based on the state, and opcode if necessary. Before the switch case, all the signals get set to 0 so that they do not stay active if they were active in a previous cycle. On the falling edge of the clock, we will transition to the next state using another switch case and by checking opcode if necessary. To test this module, manually input the opcode of each instruction type and check if the control signals are what is expected for each cycle.

System Testing

How to Test Properly:
First, test each type of instruction. All other instructions of the same type should work because of the testing done earlier for the individual component and unit testing. After that combine instructions of the same type. Probably 2 or 3 in one line to make sure that the datapath is resetting properly after each instruction. After that combine instructions of different types. It is probably best to combine small snippets of relPrime/gcd. Eventually build up to gcd. Then finally all of relPrime.

How we tested the system:
First test a single add instruction. Then test all of relPrime and GCD and fix every place it breaks.

Assembler

There are a few conventions for the assembler:

- No commas, just spaces in between instructions, memory addresses, integers, etc.
- All labels must start with a capital letter.
- All memory addresses you're loading must be lowercase. Note that the memory addresses are global. If you have a memory address labeled 'a' in one function and 'a' in another, the later 'a' will overwrite the first 'a' even though they're in separate functions.
- The label "END" will take you to a specific location in memory. This address was to just see the output when it returns without I/O.
- Use the words "Input" and "Output" as memory addresses to access their respective functions. (Capitalization matters.)

Minecraft Implementation

After completing our processor in Quartus and testing it to make sure relPrime ran properly we decided to also implement the processor in Minecraft. To implement it in minecraft we used some parts like a super compact ALU that we found online but most of the stuff including: the registers, state machine, memory, muxes, and I/O were all created by us. We had to do thorough tests of all of these components before we implemented them in the final processor. Then putting it all together was no small task. Luckily it was sped up exponentially with the use of world edit. It is definitely not the most optimized processor in minecraft but it is very close to our actual processor design with a few exceptions. Those being the SP register is only 7 bits because the memory only has 7 bits of address. And the opcodes were reordered so that it would not be as complicated to implement the different instructions. After about 30 hours of work we got our first successful run of relPrime with the input 6 and output 5. After that we successfully ran an input of 60 to get an output of 7. This took about 2.5 hours to run. If the game was running at normal settings the test bench input of 0x13b0 would take just under 36 days to run. To run it open the Minecraft world named Computer in the Minecraft Implementation folder then input the reversed value of your input and press the input button then press the lever to turn on the clock. If you want to make it run faster run this command **/tick rate 1200** then just wait for output.

Design Document Patch Notes

- [M3] Switched s1 and s2/immediate in instruction types. Just makes more sense to us to load first address first before we load anything else
- [M3] All of our instructions now have only as many bits as they need, meaning there are now no wasted bits like there was previously
- [M4] Modified RTL and RTL testing of JA type to add a cycle where Mem[A] is loaded into A, as the datapath did not support the original method
- [M5] Updated naming conventions
- [M5] Expanded Control Unit Design section to describe the implementation plan in better detail
- [M5] Added valA to list of control signals as it mysteriously vanished (was never added) and removed some redacted signals
- [M5] The FSM needed some tweaks as we found out making the project

Performance Data

Total bytes required to store and run Euclid's algorithm and relPrime:

- Storing code 216 Bytes
- Holding variables 12 Bytes
- Maximum on the stack 6 Bytes

Total executed instruction for running relPrime with input 13b0 is 40876
Total cycles is 234,872 cycles
Giving us a CPI of 5.746
The cycle time of the processor is 10.3 ns
Which gives us an execution time of 2.41 ms
Logic elements: 715 / 6,272 (11 %)
Total registers: 131
Total memory bits: 16,384 / 276,480 (6 %)

We also rewrote a better version of the relPrime function that takes advantage of bit shifting:
With the Program and the input of 13b0:
Total executed instructions: 294
Total Cycles: 1801
CPI of: 6.13
Cycle time of 10.3 ns
Execution time of 18.6 µs