Team Pink Final Report

Abe Gizaw, Liz Fogarty,
Havalock Yin, Jay Smith

Table of Contents

High-Level Description

We are designing an accumulator architecture. We will be using four accumulators to reduce the amount of instructions that can access the stack/memory and keep track of more variables. Some of our instructions take a 2-bit ID value to specify which accumulator is used for an operation. We have 32 instructions and six types. All our procedures take advantage of the stack, by using it as a place to store the return address, accumulator registers, and arguments.

Register	Name	Symbol	Assignment
R[0]	reggie 0	r0	Normal Accumulator Register
R[1]	reggie 1	r1	Normal Accumulator Register
R[2]	reggie 2	r2	Allocated for lui Values
R[3]	reggie 3	r3	Allocated for Return Values

Performance Measures

We agreed that we would be measuring “performance” based on a combination of the speed and cost of the processor. Our design utilizes a 4-accumulator architecture, finding a balance between speed and cost. While not as fast as a load/store architecture, it surpasses the performance of a 1-accumulator setup. Conversely, it's more cost-effective compared to load/store systems, though slightly more expensive than a single accumulator design. Hence, we've determined that assessing "performance" based on cost-effectiveness is the most appropriate measure for our purposes.

Rough Cost Breakdown:

- FPGA Board: $30-100
- Peripheral Components: $20-100
- PCB: $50-500
- Microcontrollers or Microprocessors” $2-20

Low-End Estimate: ~$100

High-End Estimate: ~$1000

Instruction Set

Instruction	Name	Format	Opcode	Bitwise Description
add	Add	R	00000	R[ID] = R[ID] + M[ME(imm)]
sub	Subtract	R	00001	R[ID] = R[ID] - M[SE(imm)]
xor	XOR	R	00010	R[ID] = R[ID] ⊕ M[ME(imm)]
or	OR	R	00011	R[ID] = R[ID] | M[ME(imm)]
and	AND	R	00100	R[ID] = R[ID] & M[ME(imm)]
sll	Shift Left Logical	R	00101	R[ID] = R[ID] << M[ME(imm)]
set	Set	R	00110	R[ID] = R[ID] >> M[ME(imm)]
stop	Stop	J	00111	PC = 0x00
ori	Or Immediate	I	01000	R[ID] = R{ID] | SE(imm)
xori	Xor Immediate	I	01001	R[ID] = R{ID] ^ SE(imm)
andi	And Immediate	I	01010	R[ID] = R{ID] & SE(imm)
addi	Add Immediate	I	01011	R[ID] = R[ID] + SE(imm)
slli	Shift Left Logical Imm	I	01100	R[ID] = R[ID] << imm[4:0]
srli	Shift Right Logical Imm	I	01101	R[ID] = R[ID] >> imm[4:0]
srai	Shift Right Arith Imm	I	01110	R[ID] = R[ID] >> imm[4:0]
slt	Set Less Than	R	01111	R[ID] = (R[ID] < SE(imm))?1:0
beq	Branch ==	B	10000	if (R[ID1] == R[ID2]) PC += SE(imm2) << 1
bne	Branch !=	B	10001	if (R[ID1] != R[ID2]) PC += SE(imm2) << 1
blt	Branch <	B	10010	if (R[ID1] < R[ID2]) PC += SE(imm2) << 1
bge	Branch >=	B	10011	if (R[ID1] >= R[ID2]) PC += SE(imm2) << 1
jal	Jump And Link	J	10100	M[SP] = PC + 4 PC += SE(imm) << 1
load	Load	R	10101	R[ID] = M[SE(imm)]
store	Store	R	10110	M[SE(imm)] = R[ID]

storesp	StoreSp	I	10111	R[ID] = M[SP + SE(imm)]
loadsp	LoadSp	I	11000	M[SP + SE(imm) + 8] = R[ID]
movesp	MoveSp	J	11001	SP += SE(imm)
input	Input	J	11010	Mem[SP + SE(imm)] = input
lui	Lui	J	11011	R[2] = SE(imm) << 5
jb	Jump Back	J	11100	PC = M[SP]
swap	Swap	C	11101	R[ALUOut]= R[ID1] R[ID1] = R[ID2] R[ID2] = R[ALUOut]
alter	Alter Registers	A	11110	R[returnID] = R[argID1] + R[argID2]
output	Output	J	11111	Output = R[3]

I/O

Our processor takes in inputs through the Input Instruction. Input & Output are both J types. Input takes in an immediate and puts the input at the stack pointer + the immediate. Effectively treating it like an offset. From here, memory_data is read and the corresponding instructions are added to the stack & executed. Output and Input also act as registers and the values are stored there.

Example Syntax of unique Instructions

storesp r1 0x4	This instruction takes an offset and a register and stores the value at the register onto the stack + offset
loadsp r1 0x4	This instruction takes an offset and a register, and loads the value at stack + offset+ 8 onto the specified register
movesp 8	This instruction will take an immediate and move the stack pointer that much bits
set r1 2	Set the ID register to immediate
lui 0x7AA	loads an 11-bit immediate value into the upper 11 bits of R[2] and fills in the other 5 bits with 0's.
swap r1 2	Swaps the values in 2 registers
alter r2 2 r1 +	Takes in 3 IDs, return ID, arg1 ID, and arg2 ID. Also takes in an operation represented by some immediate. This instruction will calculate R[ID1] operation R[ID2] and store it in R[ReturnID]
output 3	Outputs what's in the ID. In this case R[3]

df	15 11	10 9	8 7	6 5	4 0
R	Addy			ID	Opcode
I	Immediate			ID	Opcode
B	Immediate		ID2	ID	Opcode

J	Immediate				Opcode
C	n/a		ID2	ID	Opcode
A	Operation	ReturnID	ID2	ID	Opcode

Example code

While Loop:

```
public static int countEvens() {
int num = 1;
int count = 0;
    while (num <= 20) {
        if (num % 2 == 0) {
            count++;
        }
        num++;
    }
return count;
}
```

Address	**Assembly**	**Machine Code**	**Comments**
0x0000	set r0 1	0000000010000110	
0x0002	set r1 0	0000000000100110	
0x0004	set r2 20	0000101001000110	
0x0006	while blt r2 r0 break	0000110001010010	
0x0008	andi r0 1	0000000010001010	
0x000A	set r3 0	0000000001100110	
0x000C	bne r0 r3 not_even	0000001110010001	
0x000E	add r1 1	0000000010100000	
0x0010	not_even add r0 1	0000000010000000	
0x0012	bge r2 r0 while	1111001001010011	
0x0014	break swap r3 r1	0000000011111101	
0x0016	jb 0	0000000000011100	

Conditionals:

```
public static int max(int a, int b) {
   if(a >= b)
      return a;
   Else
      return b;
}
```

Address	Assembly	Machine Code	Comments
0x0000	loadsp r0 0x0	0000000000011000	
0x0002	loadsp r1 0x2	0000000100111000	
0x0004	blt r0 r1 end	0000011010010010	
0x0006	swap r3 r0	0000000001111101	
0x0008	jb 0	0000001010010011	
0x000A	end swap r3 r1	0000000011111101	
0x000C	jb 0	0000000000011100	

Multi-Cycle Components

Component	Inputs	Input Bit	Outputs	Output Bit	Behavior	RTL Symbols
Register File	ID	2	Reg[ID]	16	*Stores information in the register for easy access*	Reg Reg[0] Reg[1] Reg[2] Reg[3]
Registers	Data Starting address	16 16	Content in the Register	16	*These are the registers in the datapath.*	A B ALUOut SP Output IR PC
ALU	ALUSrcA ALUSrcB, ALUOp	16 16 3	ALUSrcA op ALUSrcB shouldBranch	16 1	*Operation determined from ALU Control. Once ALUOp received, you will perform the operation on SrcA and SrcB*	+, -, <<<, >>>, >>, |, &, ⊕ , ==, !=, <, >=, L0, L1, add0, add6+

ALU Control	ALUOp Opcode Operation	2 5 5	Operation for ALU	4	*Either 00 for add, 01 for subtract, or a 10 for looking at instruction opcode, and 11 for alter opcode*	
Memory	Address Data	16	Instruction	16	*Pulls the instruction located at PC & returns it*	Mem[addr]
Immediate Generator	Instruction[15-0]	16	Instruction	16	*Takes in a 16-bit instruction and decodes it into a 16 bit immediate value. For R types it Address extends(hard codes 7 MSB to 0001 111.) and Sign-extends everything else*	SE() AE()
Mux				1 to 16	Uses a selector bit to go to another operation	
Control Unit	Instruction[4-0]	5	Branch MemRead MemtoReg ALUop MemWrite RegWrite	Depends on ALU	Takes in the opcode and interprets it. After that, the control unit will generate control signals.	
InputIO	n/a	n/a	Data input from user	16		inputio
OutputIO	Data output to user	16	n/a	n/a		outputio

Multi-Cycle Components Files & Testbenches

Component	**Filename**	**Filename Path**	**Testbench**	**Testbench Path**
Register File	register_file.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/register_file.v	tb_register_file.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/tb_register_file.v
Registers	register.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/register.v	tb_register.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/tb_register.v
ALU	alu.v	https://github.com/rhit-csse2	tb_alu.v	https://github.com/rhit-csse232/rhit-csse

		32/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/alu.v		232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/tb_alu.v
ALU Control	alu_control.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/alu_control.v	tb_alu_control.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/tb_alu_control.v
Memory	memory.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/memory.v	tb_memory.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/tb_memory.v
Immediate Generator	immediate_generator.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/immediate_generator.v	tb_immediate_generator.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/tb_immediate_generator.v
Mux	mux.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/mux.v	tb_mux.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/tb_mux.v
Control Unit	control_unit.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/control_unit.v	tb_control_unit.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/tb_control_unit.v

Multi-Cycle RTL Instructions

Name / Type	Instruction	Multi-Cycle RTL	Comment
CYCLE 1		**Cycle 1:** PC = PC + 2 Inst ← Mem[PC]	Fetch:
CYCLE 2		**Cycle 2:** A ← Reg[inst[6:5]] B ← Reg[inst[8:7]]	Decode:

Name / Type	Instruction	Multi-Cycle RTL	Comment
		Mem[addr] ← Mem[AE(inst[15:7])] ALUOut ← PC + SE(inst[15:9] << 1)	
Add / R	add	**Cycle 3:** ALUOut ← A + Mem[addr] **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Perform Operation Return
Subtract / R	sub	**Cycle 3:** ALUOut ← A - Mem[addr] **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous R type instruction.
XOR / R	xor	**Cycle 3:** A ← A ⊕ Mem[addr] **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous R type instruction.
OR / R	or	**Cycle 3:** ALUOut ← A | Mem[addr] **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous R-type instruction.
AND / R	and	**Cycle 3:** ALUOut ← A & Mem[addr] **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous R type instruction.
Shift Left Logical / R	sll	**Cycle 3:** ALUOut ← A <<< Mem[addr] **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous R type instruction.
Set Less Than / R	slt	**Cycle 3:** ALUOut ← if(A < Mem[addr])? 1 : 0 **Cycle 4:** Reg[inst[6:5]] ← ALUOut	In general similar to how the B-types are written out The ALU will subtract A and the value from memory and store it in an output “isNegative” The output will be the first bit in the operation A - Mem[addr]
Load / R	load	**Cycle 3:** Reg[inst[6:5]] ← Mem[addr]	Loads what's in Mem[addr] and puts into r1

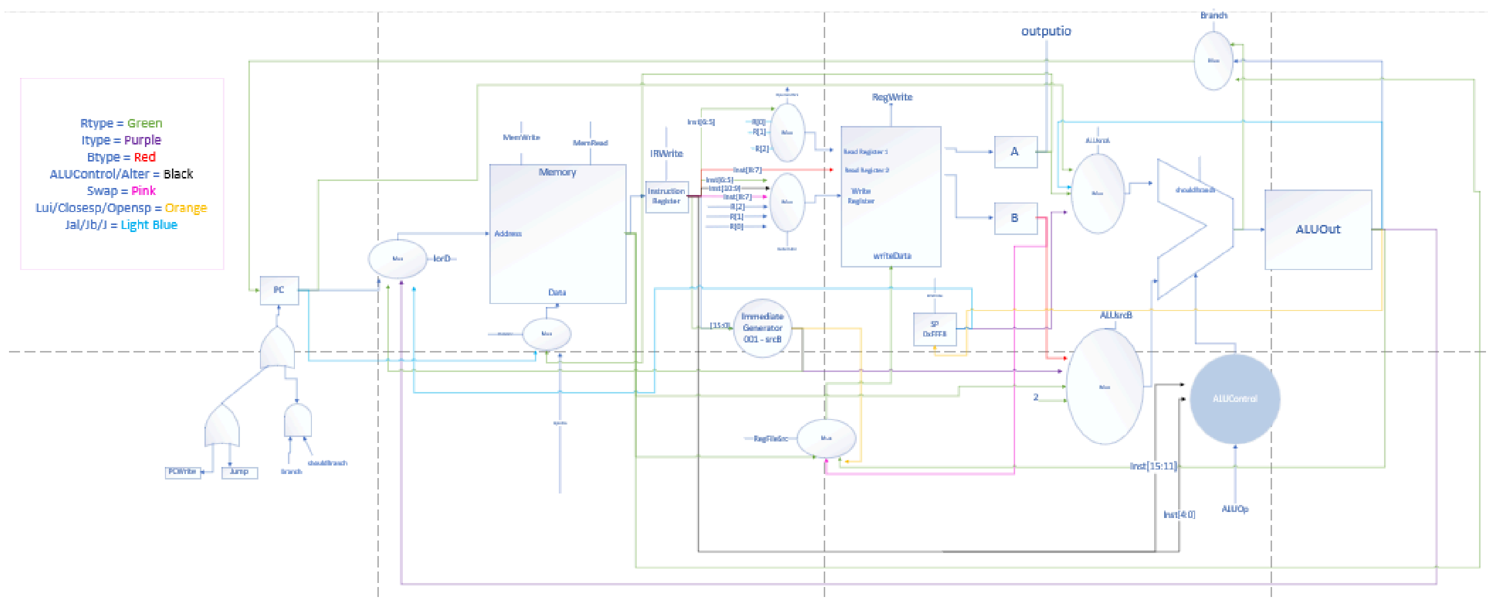
Name / Type	Instruction	Multi-Cycle RTL	Comment
Store / R	store	**Cycle 3:** Mem[AE(inst[15:7])] ← A	Stores what's in A and puts into Mem[addr]
Set / I	srl	**Cycle 3:** Reg[inst[6:5]] ← SE(inst[15:5])	Sets a register to a immediate
LoadSp / I	loadsp	**Cycle 3:** ALUOut ← SE(inst[15:5]) add8+ SP **Cycle 4:** Mem[addr] ← Mem[ALUOut] **Cycle 5:** Reg[inst[6:5]] ← Mem[addr]	Imm gen will have a signal to know to SE [15:7] Go into Regfile and write data into r1
StoreSp / I	storesp	**Cycle 3:** ALUOut ← SE(inst[15:5]) + SP **Cycle 4:** Mem[ALUOut] ← A	Calculate where in SP we are referencing Write data into memory at the address SP
Or Immediate / I	ori	**Cycle 3:** ALUOut ← A | SE(inst[15:5]) **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Perform Operation Return
Xor Immediate / I	xori	**Cycle 3:** ALUOut ← A ⊕ SE(inst[15:5]) **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous I type instruction.
And Immediate / I	andi	**Cycle 3:** ALUOut ← A & SE(inst[15:5]) **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous I type instruction.
Add Immediate / I	addi	**Cycle 3:** ALUOut ← A + SE(inst[15:5]) **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous I type instruction.
Shift Left Logical Imm / I	slli	**Cycle 3:** ALUOut ← A <<< SE(inst[15:5]) **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous I type instruction.

Name / Type	Instruction	Multi-Cycle RTL	Comment
Shift Right Logical Imm / I	srli	**Cycle 3:** ALUOut ← A >>> SE(inst[15:5]) **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous I type instruction.
Shift Right Arith Imm / I	srai	**Cycle 3:** ALUOut ← A >> SE(inst[15:5]) **Cycle 4:** [inst[6:5]] ←ALUOut	
Branch == / B	beq	**Cycle 3:** if (A == B) PC ← ALUOut	SE(inst[15:9]) represents the # of lines to jump. Multiply by 2 since addresses are 2 bits
Branch != / B	bne	**Cycle 3:** if (A != B) PC ← ALUOut	The ALU will subtract A and B and store either a 1 or 0 in the output "isZero".
Branch < / B	blt	**Cycle 3:** if (A < B) PC ← ALUOut	The ALU will subtract A and B and store the first bit of the operation in the output "isNegative".
Branch >= / B	bge	**Cycle 3:** if (A >= B) PC ← ALUOut	The ALU will subtract A and B and store the first bit of the operation in the output "isNegative".
Input / J	input	**Cycle 3:** ALUOut ← SP + SE(inst[15:5]) **Cycle 4:** Mem[ALUOut] ← inputio	
Output / J	output	**Cycle 3:** Outputio ← A	Nothing needed.
Jump And Link / J	jal	**Cycle 3:** ALUOut ← SP - 2 **Cycle 4:** Mem[ALUOut] ← PC ALUOut ← ALUOut - 2 A ← Reg[2]	

Name / Type	Instruction	Multi-Cycle RTL	Comment
		Cycle 5: Mem[ALUOut] ← A ALUOut ← ALUOut - 2 A ← Reg[1] **Cycle 6:** Mem[ALUOut] ← A ALUOut ← ALUOut - 2 A ← Reg[0] **Cycle 7:** Mem[ALUOut] ← A SP ← ALUOut PC = PC + SE(inst[15:5] << 1)	
Jump Back / J	jb	**Cycle 3:** Mem[addr] ← Mem[SP] ALUOut= SP + 2 **Cycle 4:** Reg[0] ← Mem[addr] Mem[addr] ←Mem[ALUOut] ALUOut ← ALUOut + 2 **Cycle 5:** Reg[1] ← Mem[addr] Mem[addr] ←Mem[ALUOut] ALUOut ← ALUOut + 2 **Cycle 6:** Reg[2] ← Mem[addr] Mem[addr] ←Mem[ALUOut] ALUOut ← ALUOut + 2 **Cycle 7:** PC ← Mem[addr] SP ← ALUOut	
MoveSp/ J	movesp	**Cycle 3:** ALUOut ← SP + SE(inst[15:5]) **Cycle 4:** SP ← ALUOut	Move sp Store the value back in memory
Lui / J	lui	**Cycle 3:** Reg[2] ← SE(inst[15:5] <<< 5)	
Swap / C	swap	**Cycle 3:** ALUOut ← A Reg[inst[6:5]] ← B **Cycle 4:**	RN is hard-coded to somewhere in top of memory

Name / Type	**Instruction**	**Multi-Cycle RTL**	**Comment**
		Reg[inst[8:7]] ← ALUOut	
Alter Register / A	alter	**Cycle 3:** ALUOut ← A op B **Cycle 4:** Reg[inst[10:9]] ← ALUOut	op ← inst[15:11] will go into the ALU as the ALUOp

Multi-Cycle Datapath

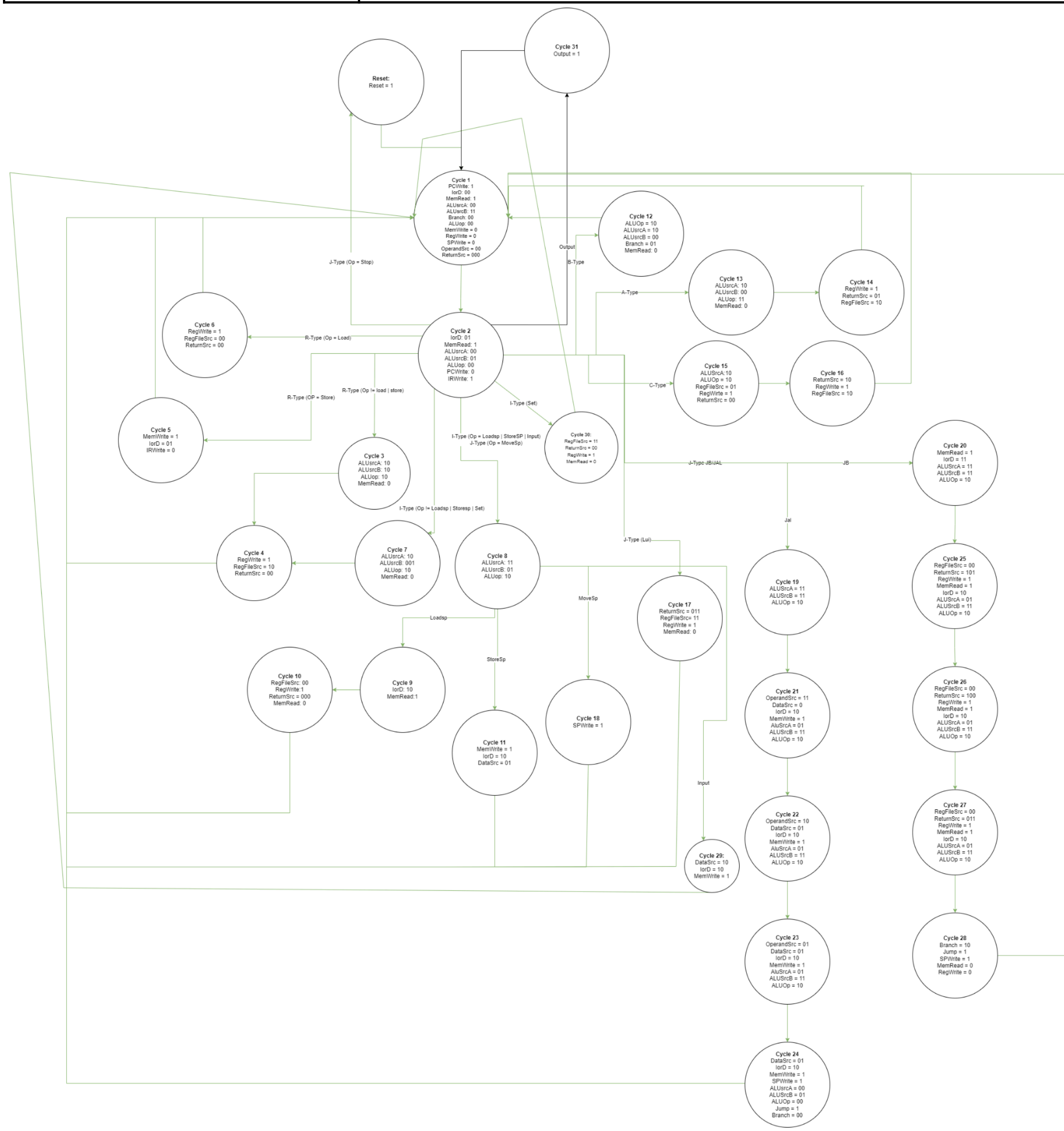


Control Signals

Signal	**Desc**
PCWrite	Controls when we write to PC 0 ⟶ Does not write onto PC 1 ⟶ Writes onto PC
Jump	Tells us if we are in a jump instruction 0 ⟶ Not a jump instruction 1 ⟶ Jump instruction
Branch	Tells us if we are in a branch instruction 00 ⟶ PC += 2 (we are not branching) 01 ⟶ PC = Jump/Branch to address 10 ⟶ PC = Jump back address
shouldBranch	Tells us if we should branch 0 ⟶ Does not Branch

	1 → Branches
IorD	Tells us what spot in memory we will be reading from 00 → PC Addr 01 → Immediate Generator 10 → ALUOut 11 → SP
MemWrite	Controls when we can write to memory 0 → Does not write to Mem 1 → Writes to Mem
MemRead	Controls when we can read from memory 0 → Does not read to Mem 1 → Reads to Mem
RegFileSrc	Controls what data gets sent to the regfile 00 → Reg file data = Mem 01 → Reg file data = B 10 → Reg file data = ALUOut 11 → Reg file data = immGen
IRWrite	Controls when we can write to the instruction register 0 → Does not write to IR 1 → Writes to IR
RegWrite	Controls when we can write onto the reg file 0 → Does not write to Reg 1 → Writes to Reg
ALUOp	Controls the ALU Control, allowing it to know what operation to send to the ALU 00 → Add 01 → Subtract 10 → Follow Instruction opcode 11 → Alter Opcode
ALUSrcA	Acts as input 1 to the ALU 00 → PC 01 → ALUOut 10 → A 11 → SP
ALUSrcB	Acts as input 2 to the ALU 00 → B 01 → ImmGen 10 → Mem 11 → 2
Return SRC	Controls what register will get written onto for certain instructions 00 → Inst[6:5] 01 → Inst[8:7] 10 → Inst[10:9]
DataSrc	Controls what data gets read into memory

	00 → A 01 → PC 10 → Inputio
OperandSrc	Controls what data gets read into A 00 - inst[6:5] 01 - r1 10 - r2 11 - r3



Testing Plan (Integration & Unit)

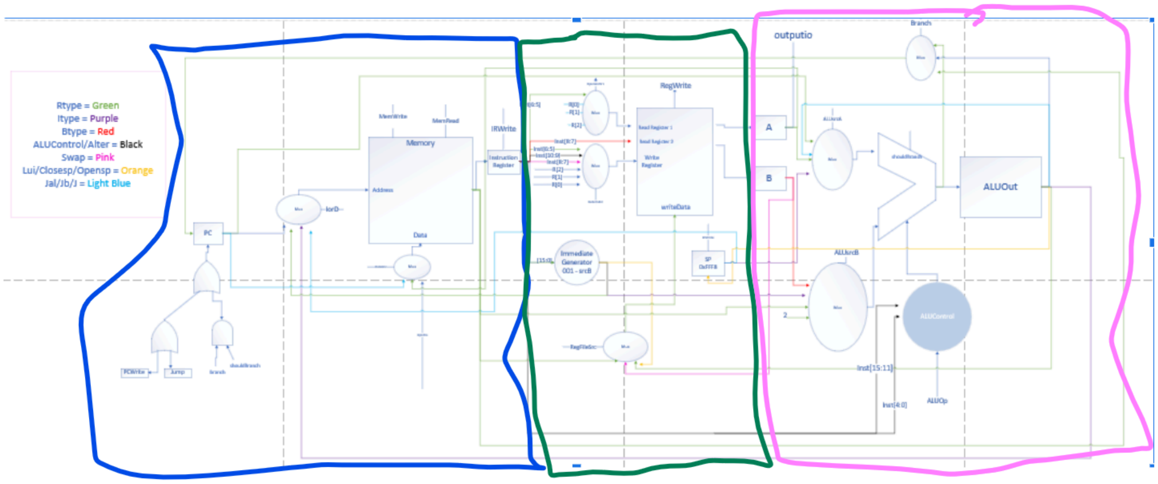
Phase 1

Tested each different component separately in their testbenches. Tested individual functionality, for example testing the opcodes make the ALU recognize the right operation. Parts of the implementation plan are shown below. ALU control, memory, immediate generator, and control unit were also tested separately.

Register File	Check after reset, info stored in R0-3 is as expected	1	tb_register
	Toggle RegWrite from 0-1 to test if regFile is writing & reading values	1	
	Check if writing to one spec register, affects output of other outputs (should not affect other registers)	1.5	
Registers	Check if input into Reg is also output on normal clock cycle	1	tb_register_file
	Check if input into Reg is also output on off-sequence clock cycle	1.5	
	Load input into Reg while write is 0 (should not change register)	1	
ALU	Check + adds values in registers	1	tb_alu
	Check - subtracts values in registers	1	
	Check <<< shifts values in registers	1	
	Check >>> shifts values in registers	1	
	Check >> shifts values in registers	1	
	Check | ors values in registers	1	
	Check & ands values in registers	1	
	Check ⊕ values in registers	1	
	Check == equals values in registers	1	
	Check != not equals values in registers	1	
	Check < less than values in registers	1	
	Check >= greater or equals values in registers	1	
	Check load 0, loads 0 values in registers	1	
	Check load 1, loads values in registers	1	

Phase 2

Split the datapath into thirds and combine those components and then test to make sure those smaller subunits of the whole datapath would work. The divided parts are shown below along with partial test snippets.



Example test plan for the Green unit:

Set 2: RegFile, SP, RegFileSrc Mux, Operand Mux, IMM Generator, ReturnSrc Mux Signals: ReturnSrc, OperandSrc, RegFileSrc, SPWrite, RegWrite	**Batch 1:** OperandSrc = 00 RegWrite = 0 Expect outputs to be content in the register defined in IRInput[6:5] and IR[8:7] (if valid) SPOut should always be 0x03FF Test ImmGen Output depended on IRInput Type Test With 1 of every type
	Batch 2: RegWrite = 1 RegFileSrc = 00 ReturnSrc = 000 Expect whatever is in MemInput to be written on the reg in IRInput[6:5]
	Batch 3: RegWrite = 1 RegFileSrc = 01 ReturnSrc = 001 Expect whatever is in ALUSrcBInput to be written on the reg in IRInput[10:9]
	Batch 4: RegWrite = 1 RegFileSrc = 10 ReturnSrc = 010 Expect whatever is in ALUOutInput to be written on the reg in IRInput[8:7]

	Batch 5: RegWrite = 1 RegFileSrc = 11 ReturnSrc = 011 Expect whatever is in ImmGenValOut to be written onto r2

Phase 3

Combined all the components in Phase 2 and tested them together without the control unit.

Phase 3 - Abe	Test sets of 3 components From Phase 2 together with RelPrime			
	Code	**Cycles**	**Hex**	**Expected Values**
	`set r0 8`	3	`0406`	r0 = 8
	`set r1 2`	3	`0126`	r1 = 2
	`set r2 1`	3	`00c6`	r2 = 1
	`movesp -4`	4	`ff99`	sp = 0x07fd
	`storesp r0 0`	4	`0017`	Mem[0x07fd] = 8
	`storesp r1 2`	4	`0137`	Mem[0x07fdf = 2
	`jal GCD`	7	`00d4`	SP = 0x0705 Mem[0x07fb] = 0x000e Mem[0x07f9] = 0x0001 Mem[0x07f7] = 0x0002 Mem[0x07f5] = 0x0008
	`loadsp r0 0`	5	`0018`	r0 = 8
	`loadsp r1 2`	5	`0138`	r1 = 2
	`set r2 0`	3	`0046`	r2 = 0
	`bne r0 r2 continue`	3	`0511`	PC = 0x0026
	`beq r1 r2 done`	3	`0b30`	PC = 0x0028
	`blt r1 r0 ChangeA`	3	`0432`	PC = 0x002e
	`alter r0 r0 r1 -`	4	`089e`	r0 = 6
	`beq r0 r0 continue`	3	`f410`	PC = 0x0026
	`beq r1 r2 done`	3	`0b30`	PC = 0x0028
	`blt r1 r0 ChangeA`	3	`0432`	PC = 0x002e
	`alter r0 r0 r1 -`	4	`089e`	r0 = 4
	`beq r0 r0 continue`	3	`f410`	PC = 0x0026
	`beq r1 r2 done`	3	`0b30`	PC = 0x0028

	blt r1 r0 ChangeA	3	0432	PC = 0x002e
	alter r0 r0 r1 -	4	089e	r0 = 2
	beq r0 r0 continue	3	f410	PC = 0x0026
	beq r1 r2 done	3	0b30	PC = 0x0028
	blt r1 r0 ChangeA	3	0432	PC = 0x002a
	alter r1 r1 r0 -	4	0a3e	r1 = 0
	beq r1 r1 continue	3	f8b0	PC = 0x0026
	beq r1 r2 done	3	0b30	PC = 0x0032
	swap r0 r3	4	019d	r0 = 0 r3 = 2
	jb 0	7	001c	r0 = 8 r1 = 2 r2 = 1 r3 = 2 SP = 0x07fd PC = 0x000e
	movesp 4	4	0099	SP = 0x0801
	beq r2 r3 finish	3	05d0	PC = 0x0012
	addi r1 1	4	00ab	r1 = 3
	beq r1 r1 SU	3	f0b0	PC = 0x0006
	movesp -4	4	ff99	sp = 0x07fd
	storesp r0 0	4	0017	Mem[0x07fd] = 8
	storesp r1 2	4	0137	Mem[0x07fdf = 3
	jal GCD	7	00d4	SP = 0x0705 Mem[0x07fb] = 0x000e Mem[0x07f9] = 0x0001 Mem[0x07f7] = 0x0003 Mem[0x07f5] = 0x0008
	loadsp r0 0	5	0018	r0 = 8
	loadsp r1 2	5	0138	r1 = 3
	set r2 0	3	0046	r2 = 0
	bne r0 r2 continue	3	0511	PC = 0x0026
	beq r1 r2 done	3	0b30	PC = 0x0028
	blt r1 r0 ChangeA	3	0432	PC = 0x002e
	alter r0 r0 r1 -	4	089e	r0 = 5
	beq r0 r0 continue	3	f410	PC = 0x0026
	beq r1 r2 done	3	0b30	PC = 0x0028

	blt r1 r0 ChangeA	3	0432	PC = 0x002e
	alter r0 r0 r1 -	4	089e	r0 = 2
	beq r0 r0 continue	3	f410	PC = 0x0026
	beq r1 r2 done	3	0b30	PC = 0x0028
	blt r1 r0 ChangeA	3	0432	PC = 0x002a
	alter r1 r1 r0 -	4	0a3e	r1 = 1
	beq r1 r1 continue	3	f8b0	PC = 0x0026
	beq r1 r2 done	3	0b30	PC = 0x0028
	blt r1 r0 ChangeA	3	0432	PC = 0x002e
	alter r0 r0 r1 -	4	089e	r0 = 1
	beq r0 r0 continue	3	f410	PC = 0x0026
	beq r1 r2 done	3	0b30	PC = 0x0028
	blt r1 r0 ChangeA	3	0432	PC = 0x002a
	alter r1 r1 r0 -	4	0a3e	r1 = 0
	beq r1 r1 continue	3	f8b0	PC = 0x0026
	beq r1 r2 done	3	0b30	PC = 0x0032
	swap r0 r3	4	019d	r0 = 2 r3 = 1
	jb 0	7	001c	r0 = 8 r1 = 3 r2 = 1 r3 = 2 SP = 0x07fd PC = 0x000e
	movesp 4	4	0099	SP = 0x0801
	beq r2 r3 finish	3	05d0	PC = 0x0016
	swap r1 r3	4	01bd	r1 = 1 r3 = 3
	xori r2 0	4	0049	r2 = 1
	srli r2 1	4	00cd	r2 = 0
	lui 0x3FF	3	7ffb	r2 = 0x6FE0
	srai r2 1	4	00ce	r2 = 0x3FF0
	store r1 0x01FF	3	ffb6	Mem[0x01FF] = 1
	add r2 0x01FF	4	ffc0	r2 = 0x3FF1
	sub r2 0x01FF	4	ffc1	r2 = 0x3FF0
	or r2 0x01FF	4	ffc3	r2 = 0x3FF1
	load r0 0x01FF	3	ff95	r0 = 1
	store r2 0x01FF	3	ffd6	Mem[0x01FF] = 0x3FF1

Component Testing

Phase	Filename	Filename Path	Testbench Filename	Testbench Filepath	Test Description
Phase 2	magenta_stage_ALUControl_ALU_ALUOut_RegA_RegB.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/magenta_stage_ALUControl_ALU_ALUOut_RegA_RegB.v	tb_magenta_stage_ALUControl_ALU_ALUOut_RegA_RegB.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/tb_magenta_stage_ALUControl_ALU_ALUOut_RegA_RegB.v	Tests ALU, ALUControl, ALUOut, RegA, and RegB to ensure all arithmetic instructions can work
	inte_stage1_part1.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/inte_stage1_part1.v	tb_inte_stage1_part1.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/tb_inte_stage1_part1.v	Tests placing values into memory & taking values out. Also tests PCs updates
	regFile_immgen_sp.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/regFile_immgen_sp.v	tb_regFile_immgen_sp.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/tb_regFile_immgen_sp.v	Tests register file & ensures registers update accordingly as well as immediate generator produces correct values.
Phase 3	headless_machine.v	https://github.com/rhit-csse232/rhit-csse232-2324b-proj	Tb_headless_machine.v tb_headless_machine_IO.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/tb_h	Tests the three separate components together, without control unit & manual input

		ect-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/headless_machine.v		eadless_machine.v https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/tb_headless_machine_IO.v	Tests the three separate components together, without control unit & regular input
	final_machine.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/final_machine.v	tb_final_machine.v	https://github.com/rhit-csse232/rhit-csse232-2324b-project-pink-2324b-01/blob/final-tb/implementation/CSSE232ProectPinkQuatrus/tb_final_machine.v	Builds off headless TB but control is connected to tests different cycles

Performance:

Total number of Bytes:

- Euclid's Algorithm: 12
- relPrime: 70

Total number of Inst. for relPrime: 40,905
Total number of cycles required to execute relPrime: 13,094
Average Cycles per inst: 3.25
Cycle time for Design: 13.1 ns
Total Execution time for relPrime: 1.7 ms
Count of logical gates and registers:

- Registers: 192
- Logical Elements: 812 / 6,272 (13 %)
- Memory Bits: 16,384 / 276,480 (6 %)

Our processor performs as well as it does due to our limited memory access as well as our instructions like swap & alter. Swap and alter drastically reduce the number of lines & memory access needed to perform basic & common operations. Our four accumulator registers also allow us to store data in easy to rab locations cutting down on memory access times.

Unique Features:

4 Registers

Our architecture utilizes an accumulator but has 4 accumulator registers. Registers[0-1] are normal accumulator registers while Register[2] is dedicated to holding lui values and Register[3] is the return value location. Having 4 accumulator registers instead of 2 allows us to combine the convenience of a load/store architecture with the simplicity of an accumulator.

Address extending instructions (R-types)

Due to limited space in memory, the upper 7 bits of static memory are hardcoded and the lower 9 bits are specified by the instruction allowing access to a range within a 512-byte window. We coined the term “Address Extending” to describe this.

Using the stack for procedures

When a procedure is called, the user must open up space on the stack using movesp. Movesp should only open up 2^9 bits, although its immediate can hold more. The return address will always be put on index 0 of the stack by jal, and the user does not have to do it themselves. If a user decides to open up more, it is up to them to make sure things are put in the correct order. Arguments must be placed in order of how a procedure expects it. So a procedure that takes (a, b, c) expects a at 0x00, b at 0x02, c at 0x4, etc. Another thing to keep in mind is that movesp takes in an 11-bit immediate, but loadsp and storesp only take a 7-bit immediate. This means you can open up more space than you can access.

No MDR

Memory holds the data during each cycle so an MDR component in the processor is obsolete.

Instructions such as alter and swap

Alter and Swap eliminate repetitive instructions. Swap allows two registers to be swapped without creating a temporary variable. Alter cuts down three lines of code to one. Alter allows the user to select two registers and then perform a basic ALU operation. Instead of having separate instructions to load values, perform the operation, and then store the value back in the desired spot, alter speeds this process up. Eliminating 4-5 lines into one line.

Extra Features:

Assembler

To simplify converting our instructions into binary, a Python assembler was created. Using the Python terminal, the user can enter a file path name, choose the file type of the instructions, either a .csv or a .txt file, and whether or not they want notes added to the output, and if they want the output in binary or hex. Then the assembler converts the file and the output is added to the same file.

```
Excel or txt?(E,T)
T
Enter the absolute path to the assembly file: G:\My Drive\classes\232CompArch\project\rhit-csse232-2324b-project-pink-2324b-01\implementation\assembler2\tests\random.tx
Do you want a detailed output (addresses, in-line comments, spacing)?(Y/N) Y
Do you want values in hex or Binary?(H/B)B
```

Input Example:

```
Base:
   movesp -2  // random comment
   input 0
   jal TestInput
   output 3  //pls work
TestInput:
   loadsp r0 0
   addi r0 1 //add
   swap r0 r3
   jb 0
```

Output Example:

```
Binary:
Base:                      ~  5           0x0
11111111110 11001          ~  16           0x0  // random comment
00000000000 11010          ~  16           0x2
00000000001 10100          ~  16           0x4
00000000011 11111          ~  16           0x6  //pls work
TestInput:                 ~  10           0x8
000000000 00 11000         ~  16           0x8
000000001 00 01011         ~  16           0xa  //add
0000000 11 00 11101        ~  16           0xc
00000000000 11100          ~  16           0xe
```

Conclusion:

Our accumulator-based architecture offers a balanced and cost-effective system. Our design utilizes the high performance of a load/store system while also using the simplicity of an accumulator. Our instruction set allows users to create a variety of programs while reducing instructions where possible. Overall, our processor is fitted for a wide variety of processes while staying cost-effective and fast.