Comp Arch Crusaders Accumulator Processor

Developed by:

Thomas Bioren, Elenaor Brooks, Jada Hunter-Hays, Daniel Seiler

Rose-Hulman Institute of Technology

In association with:

Robert Williamson, *Rose-Hulman Institute of Technology*

February 21, 2024

Contents

1 Introduction to the Processor

Our design philosophies aligned with many of those of RISC-V. We prioritized simplicity, with a minimal number of instructions and instruction types. We aligned with the principle "smaller is faster" by using an accumulator architecture to minimize the number of registers used in the processor. We made the common case fast by designing our processor around the most commonly used instructions such as ADDI and LW/SW. We attempted to make smart compromises whenever possible, as well.

Choosing to create a small and inexpensive processor means that we sacrifice on raw performance. This is okay as we are not trying to create the fastest processor possible, we are trying to create an affordable and easy to program one.

2 Detailed Overview of Instruction Set

2.1 Core Instruction Set

We opted to use fewer instructions to make our processor smaller and simpler. This aligns with our principle "smaller is faster." Additionally, this makes the processor cheaper, which is better for us.

Name	Type	Opcode	Function	English Description
LW	I-Type	0x0	Reggie = Mem[SE(imm)]	Load mem[addr] into Reggie
SW	I-Type	0x1	Mem[SE(imm)] = Reggie	Store the value of Reggie into mem[imm]
LMEM	I-Type	0x2	Reggie = Mem[Mem[SE(imm)]]	Load MEM[MEM[addr]] into Reggie
SMEM	I-Type	0x3	MEM[MEM[SE(imm)]] = Reggie	Store the value of Reggie into MEM[MEM[addr]]
ADD	I-Type	0x4	Reggie = Reggie + Mem[SE(imm)]	Increment Reggie by mem[imm]
SUB	I-Type	0x5	Reggie = Reggie - Mem[SE(imm)]	Decrement Reggie by mem[imm]
ADDI	I-Type	0x6	Reggie = Reggie + SE(imm)	Increment Reggie by the immediate
CMPI	I-Type	0x7	If Reggie < SE(imm): sr[1:0] = **01** If Reggie = SE(imm): sr[1:0] = **10** If Reggie > SE(imm): sr[1:0] = **11**	Compare Reggie and the immediate and update sr 1:0 accordingly
CMP	I-Type	0x8	If Reggie < Mem[SE(imm)] : sr[1:0] = **01** If Reggie = Mem[SE(imm)] : sr[1:0] = **10** If Reggie > Mem[SE(imm)] : sr[1:0] = **11**	Compare Reggie and the mem[imm] and update sr 3:0 accordingly
B	B-Type	0x9	If comp_code == sr[1:0] PC = PC + destination address Comp_code 00 == unconditional branch	Branch to dest_addr if comp_code equals SR[1:0]
BMEM	B-Type	0xA	If comp_code == sr[1:0] PC = PC + destination address Comp_code 00 == unconditional branch	Branch to location stored at address of immediate.
LLI	I-Type	0xB	Reggie[11:0] = SE(imm)	Load the immediate into the most significant bits of reggie. The remaining upper bits are 0.
SLLI	I-Type	0xC	Reggie = Reggie << SE(imm)[2:0]	Logical Left shift Reggie by the immediate
LI	I-Type	0xE	input = Reggie	Put the value of input into Reggie. Any immediate provided will be ignored.
SO	I-Type	0xF	output = Reggie	Put the value of Reggie into output

Figure 1: Comp Arch Crusaders Instruction Set

2.2 Registers

As this is an accumulator architecture, there is one main register that can be modified through assembly. For ease of communication, we have named this register "Reggie" in our documentation.

2.3 Core Instruction Formats

To prioritize simplicity, our processor only has two instruction types: I-Type and B-Type.

I-Type: To be used with an immediate. All instructions that don't branch are I-Type. The immediate is either treated as an immediate or as a memory location depending on what the instruction does.

B-Type: The only two B-Type instructions are B and BMEM. The only difference between B-Type and I-Type instructions is that B-Type instructions use bits [15:14] as the comparison code. Comparison codes are used to determine if the processor should branch.

Bits	15:14	13:12	11:8	7:4	3:0
I-Type	Immediate				Opcode
B-Type	Comparison Code	Memory Immediate			Opcode

Figure 2: Comp Arch Crusaders Core Instruction Formats

Note: The higher the immediate index, the more significant the bit. Refer to LUI and LLI to see how the most and least significant bits are read.

2.4 Handling I/O

Handling Input: Input is handled with the *LI* instruction. This instruction will load whatever is on the input wires into the main accumulator register, Reggie.

Handling Output: Currently, output is tied directly to the value of Reggie at any given

clock cycle. Future plans are to implement a new instruction, *SO*, that will store the value of Reggie into a new output register.

2.5 Special Instruction Uses

ADD: ADD differs from RISC-V in that it takes an immediate. This immediate is a memory location. ADD takes the value stored at this memory location and increments Reggie by it.

SUB: SUB behaves the same as ADD but with subtraction.

CMP and CMPI: CMPI is similar to ARM's compare instruction. It compares Reggie to the immediate provided and sets the status register (SR) to a value corresponding to the result. CMP does the same, but the immediate provided is a memory address. The processor compares Reggie to the value stored at that address.

B and BMEM: These are conditional branches. They only branch if the comparison code provided matches what is in SR. B branches to the address of the immediate provided while BMEM branches to the address that is stored in memory at that location. Both are PC-relative.

LI: Loads input from the input wire into Reggie. It loads whatever is being inputted on the 3rd cycle of the instruction.

2.6 Example Code for Common Use Cases

Procedure calls are a common use case in assembly. The following is a proper procedure call using the Comp Arch Crusaders' ISA:

```
MAIN:
        # Create a saved and temp variable
        LLI 32
        SW S0
        LLI 10
        SW T0
        # Back up T0 by:
        # 1. Load SP into Reggie
        # 2. Decrement Reggie by 2
        # 3. Store Reggie back into SP
```

```
        # 4. Load T0 into Reggie
        # 5. Store Reggie into the memory location where SP points
        LW SP
        ADDI -2
        SW SP
        LW T0
        SMEM SP
        # Store the return address into RA
        LLI HERE_AFTER_CALL
        SW RA
        B 00 DO_SOMETHING
HERE_AFTER_CALL:
        # Restore T0 by:
        # 1. Load the item in memory that SP is pointing to into Reggie
        # 2. Storing Reggie into T0
        # 3. Loading SP into Reggie
        # 4. Incrementing Reggie by 2
        # 5. Storing Reggie back into SP
        LMEM SP
        SW T0
        LW SP
        ADDI 2
        SW SP
        LW S0
        # Output
        SO

DO_SOMETHING:
        # Back up the saved variable
        LW SP
        ADDI -2
        SW SP
        LW S0
        SMEM SP
        # Do whatever the function is supposed to do
        # (this overwrites S0 and T0)
        LLI 64
        SW S0
        LLI 128
        SW T0
        # Replace the backed up variable
        LMEM SP
        SW S0
        LW SP
        ADDI 2
        SW SP
        # Branch back to caller
        LW RA
        BMEM 00 RA
```

2.7 Multicycle RTL

We have two RTL Tables for ease of review. Please reference the Components List subsection for RTL symbols and their corrosponding components.

	LI/SO	CMPi	CMP	ADDI	ADD/SUB
fetch	PC <= PC + 2				
	IR <= Mem[PC]				
decode	Imm <= SE(IR[15:4])				
	CompCode <= SE(IR[15:14])				
	ALUout <= SE(IR[13:4])				
	LI: Reggie <= Input **SO**: Output <= Reggie	ALUout <= Reggie - imm	MDR <= Mem[imm]	ALUout <= Reggie + imm	MDR <= Mem[imm]
		if ALUout isZero, sr[1:0] = 00 if ALUout [0] == 0, sr[1:0] = 11 If ALUout[0] == 1, sr[1:0] = 01	ALUout <= Reggie - MDR	Reggie <= ALUout	ALUout <= Reggie op MDR
			if ALUout isZero, sr[1:0] = 00 if ALUout [0] == 0, sr[1:0] = 11 If ALUout[0] == 1, sr[1:0] = 01		Reggie <= ALUout

Figure 3: RTL Table 1

	LW/SW	B	BMEM	LMEM/SMEM	SLLI/LLI
fetch	PC <= PC + 2				
	IR <= Mem[PC]				
decode	Imm <= SE(IR[15:4])				
	CompCode <= SE(IR[15:14])				
	ALUout <= SE(IR[13:4])				
	SW: Mem[imm] <= Reggie **LW**: Reggie <= Mem[imm]	if(sr == comp_code OR comp_code == 00) ALUout <= PC +	if(sr == comp_code OR comp_code == 00) ALUout <= PC +	MDR <= Mem[imm]	**LLI:** Reggie[11:0] <= imm[11:0] **SLLI:** Reggie = Reggie << imm[3:0]

Figure 4: RTL Table 2

2.8 Component List

Our architecture uses the following generic components:

Components	Inputs	Outputs	Behavior	RTL Symbols	Component Path	TB Path
Register	reg_in[15:0] CLK[0:0] reset[0:0]	reg_out[15:0]	On each rising CLK edge, reads input signal and outputs value. If reset is high, wipes data and outputs 0	PC, Reggie, MDR, IR, ALUout	implementation\Datapath\Register.v	implementation\Datapath\Register_tb.v
2-Bit Register	CLK[0:0] Reg2_in[1:0] Reset[0:0] SRw[0:0]	reg2_out[1:0]	On each rising CLK edge, reads input signal and outputs value. If reset is high, wipes data and outputs 0x00	CompCode	implementation\Datapath\CompCode.v	implementation\Datapath\CompCode_tb.v
Status Register	CLK[0:0] Reg2_in[0:0] isZero[0:0] Reset[0:0] SRw[0:0]	Reg2_out[1:0]	On each rising CLK edge, reads input signal and isZero and outputs a value based on the specific comp codes layed out . If reset is high, wipes data and outputs 0x00	SR	implementation\Datapath\StatusRegister.v	implementation\Datapath\StatusRegister_tb.v
ALU	aluop[1:0] aluA_in[15:0] aluB_in[15:0]	alu_out[15:0] isZero[0:0]	Reads the ALUOp[1:0] to determine the operation and then outputs result	+, -, op	implementation\Datapath\ALU.v	implementation\Datapath\ALU_tb.v
Immediate Generator	Imm_in[11:0]	imm_out[15:0]	SE a 12 bit number to 16	Imm	implementation\Datapath\ImmediateGenerator.v	implementation\Datapath\tb_imm.v
Memory	memW[0:0] memR[0:0] dataW_in[15:0] addr_in[15:0]	mem_out[15:0]	On each rising CLK edge, if memwrite is high, then it reads input from datawrite and outputs its value.	Mem	implementation\Datapath\Memory.v	implementation\Datapath\tb_memory.v
Control	opc[3:0] compc[15:14]	IorD[0:0] aluop[1:0] regw[0:0] srw [0:0] compcodew [0:0] memw[0:0] aluSrcA[2:0] aluSrcB[2:0]	Takes in the opc[3:0] and compc[3:0] to determine the output needed for the control signals.	Ctrl	implementation\Datapath\Control.v	implementation\Datapath\control_tb.v

Figure 5: Component List

2.9 Testing

Our philosophy for testing was to robustly test components, checking for edge cases and unpredicted behavior. Once component level testing was completed we moved onto stage level testing where we broke down our datapath into 5 distinct stages, where each stage was then tested rigorously, before we finally tested the complete datapath. We hoped that by integrating larger chunks with every testing step we could find errors more effectively.

Component	Test Type	Input Signals	Test Description	Expected Output
Register	Reset	Reg_in[15:0] = 0x0002 Reset[0:0] = 1	Set register to 0x002 for a clock cycle, then turn the reset signal high (1) and verify output at the next cycle is 0x0000.	Out[15:0] = 0x0000
	Write/Read	regW =1/ 0 1.Reg_in[15:0] = 0x0002 2.Reg_in[15:0] = 0x0020 3.Reg_in[15:0] = 0x0200 4. Reg_in[15:0] = 0x2000	Set Register to 0x0002 And verify write on next 2 clock cycles with write signal =0 Repeat process for remaining Reg_in signals	1.Reg_in[15:0] = 0x0002 2.Reg_in[15:0] = 0x0020 3.Reg_in[15:0] = 0x0200 4. Reg_in[15:0] = 0x2000
SR	Less-than	Reg2_in[0:0] =1 isZero[0:0] =0	Verify output	Out[1:0] =0b 01
	Greater than	Reg2_in[0:0] =0 isZero[0:0] =0	Verify output	Out[1:0] = 0b11
	equals	isZero[0:0] =1 3. Reg2_in[:0]= 0 4. Reg2_in[:0]= 1	Verify output for each signal.	Out[1:0] = 0b10
	reset	Reg2_in[0:0] = b0 isZero[0:0] = 0 Reset[0:0] = 1	Set register to 0b01 for a clock cycle, then turn the reset signal high (1) and verify output at the next cycle is 0b00.	Out[1:0] = 0b00
Immediate Generator	Sign extension	imm_in = 12'b111111111000;	Starts with the provided immediate and adds one to the input for 32 cycles and checks the corresponding output	SE(imm_in)

Figure 6: Component Test Plan

Since the behavior for ALU and Control was more complex, their tests have been summarized below.

ALU Test Description:

Addition Test Description: Verify that the ALU correctly performs addition operations.

Inputs: Two positive numbers as inputs (aluA_in ,aluB_in) and the ALU operation (aluop) set to perform addition.
Expected Output: The alu_out should contain the sum of the two input numbers, and the isZero flag should be set accordingly.

Subtraction Test:

Description: Verify that the ALU correctly performs subtraction operations.
Inputs: Two numbers (aluA_in ,aluB_in) with aluA_in greater than or equal to aluB _in, and the ALU operation (aluop) set to perform subtraction.
Expected Output: The alu_out should contain the result of subtracting aluB_in from aluA_in, and the isZero flag should be set accordingly.

Left Shift Test:

Description: Verify that the ALU correctly performs left shift operations.
Inputs: A number (aluB_in) and a shift amount (aluA_in), and the ALU operation (aluop) set to perform left shift.
Expected Output: The alu_out should contain the result of left shifting aluB_in by the number of bits specified in aluA_in, and the isZero flag should be set accordingly if the result is zero.

Control Test Description:

Opcode: 0000, IorD: 1, memw: 0, regw: 1, pcw: 0, srw: 0, aluop: 00, regop: 00, aluSrcA: 00, aluSrcB: 00, compcodew: 0
Opcode: 0001, IorD: 1, memw: 1, regw: 0, pcw: 0, srw: 0, aluop: 00, regop: 00, aluSrcA: 00, aluSrcB: 00, compcodew: 0
Opcode: 0100, IorD: 0, memw: 0, regw: 1, pcw: 0, srw: 0, aluop: 10, regop: 00, aluSrcA:

00, aluSrcB: 00, compcodew: 0

Opcode: 0101, IorD: 0, memw: 0, regw: 1, pcw: 0, srw: 0, aluop: 11, regop: 00, aluSrcA: 00, aluSrcB: 00, compcodew: 0

The control unit interprets the opcode input to determine the control signals to be asserted or deasserted. It sets appropriate control signals based on the opcode to configure the ALU operation, memory access, register write, program counter update, etc. For undefined opcodes, it sets default values for all control signals to 0.

2.10 Datapath

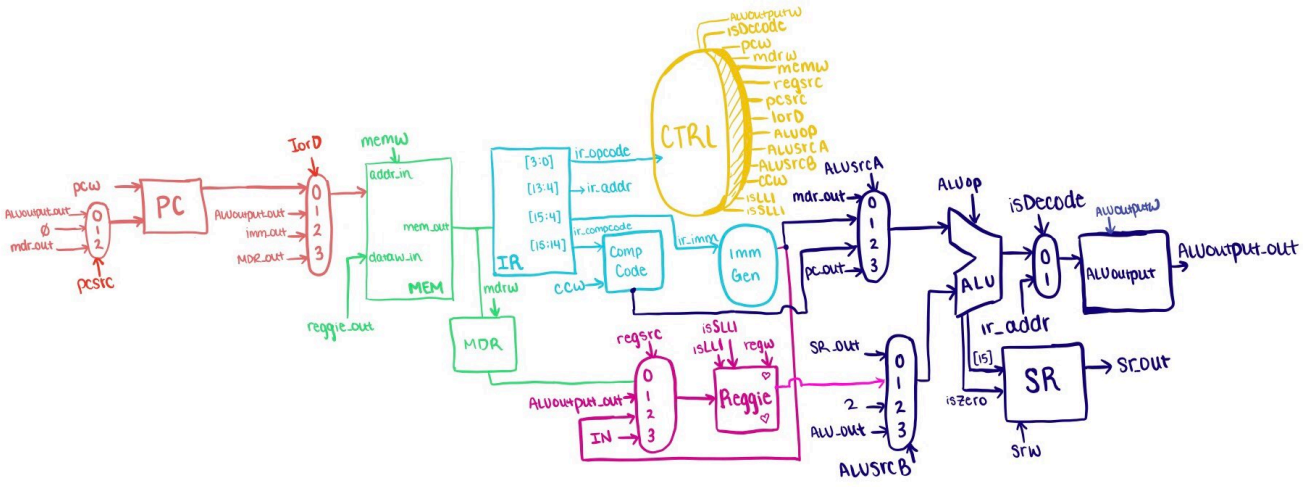

Figure 7: Complete Datapath

2.11 Control

Visualization of the Finite State Machine defined in Control.V.

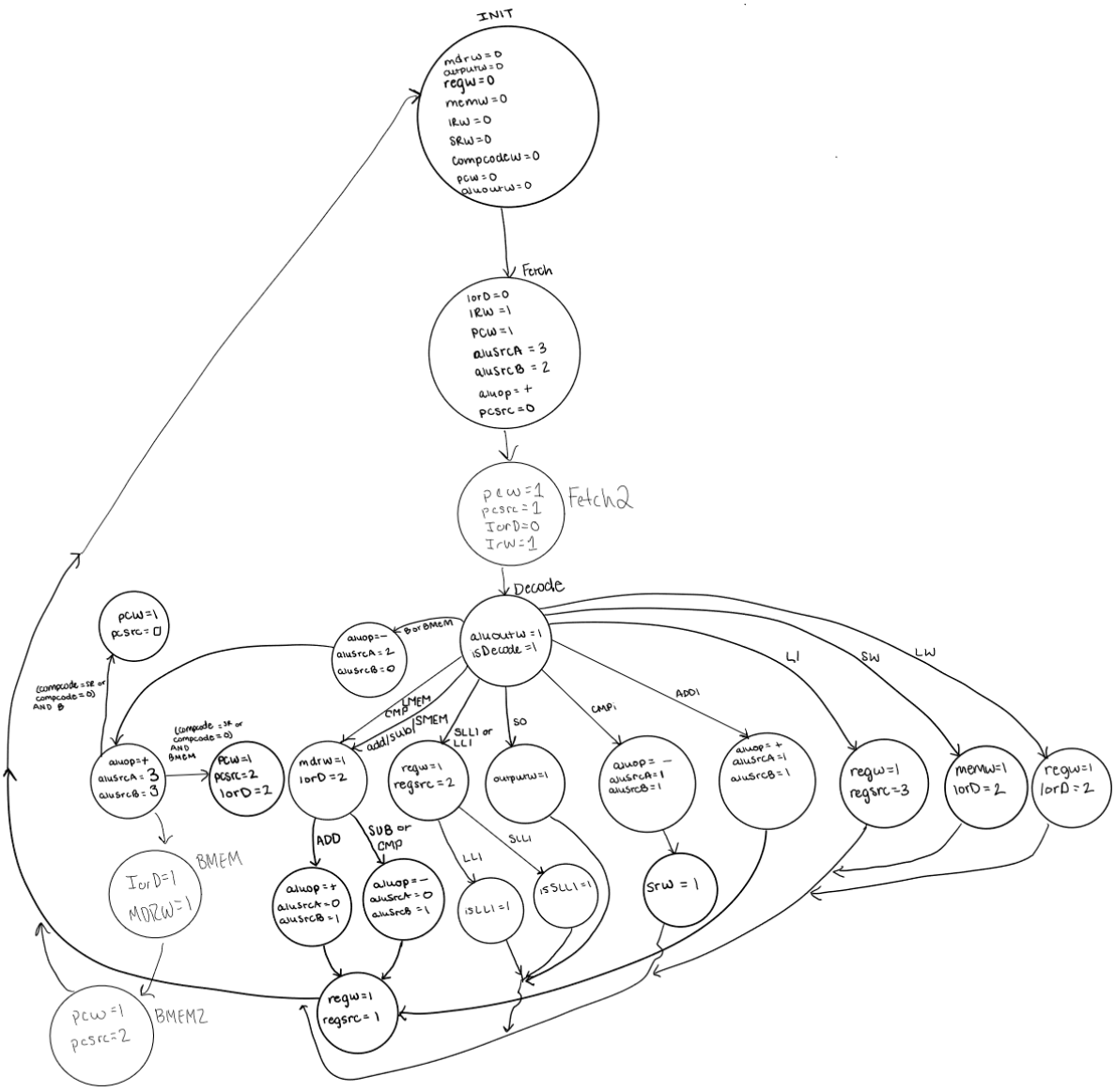

Figure 8: Control Diagram

The control signals are defined in the tables below.

regsrc[1:0]	Mux selector bit for whether Reggie's input comes from MDR ,,ALUout,imm,input
regw[0:0]	High: overwrites value in Reggie; Low: maintains the same value in register
srw[0:0]	High: overwrites value in SR; Low: maintains the same value in register
memw[0:0]	High: overwrites value in Memory; Low: maintains the same value in register
compcodew[0:0]	High: overwrites value in CompCode, Low: maintains the same value in register
IorD[0:0]	Mux selector bit for whether the value from PC is data or an instruction
ALUsrcA[1:0]	Mux selector for ALU input A: 00 = MDR_out 01= imm_out 10 = compcode_out 11 = PC
ALUsrcB[1:0]	Mux selector for ALU input B 00 = SR_out 01= Reg_out 10 = 2 11 = ALU_out
ALUop[1:0]	Signals what kind of operation the ALU will perform Table with values is above
ALUoutputw[0:0]	Write bit
isDecode[0:0]	Mux selector bit for aluOUT register
pcw[0:0]	Write bit for PC
pcsrc[1:0]	Mux selector bit for Pc input: output, 0,mdr_out
mdrw[0:0]	Write bit for mdr register

MUX ALUsrcA	**Meaning**
0	ALU input is from memory
1	ALU input is from immediate generator
2	ALU input is from CompCode
3	ALU input is from PC

MUX ALUsrcB	**Meaning**
0	ALU input is from status register
1	ALU input is from Reggie
2	ALU input is 2
3	ALU input is from ALU_out

SR Value	**Meaning**	**Branch Meaning**
0	ALUout[15] = X && isZero == X	Unconditional branch
1	ALUout[15] = 1 && isZero == 0	Reggie < value
2	ALUout[15] = 0 && isZero == 1	Reggie == value
3	ALUout[15] = 0 && isZero == 0	Unconditional branch

ALUop	**Operation**	**Inputs**	**Outputs**
11	Sub	Imm_out[15:0] Mem_out[15:0] reggie_data[15:0] mdr_data[15:0]	alu_out[15:0]
10	Add	Imm_out[15:0] Mem_out[15:0] reggie_data[15:0] Mdr_data[15:0]	alu_out[15:0]
01	Shift Lower Logical Immediate	Imm_out[15:0] Mem_out[15:0] reggie_data[15:0] Mdr_data[15:0]	alu_out[15:0]

Figure 9: Control Signal Definitions

2.12 Integration Plan

We implemented the 5 stages in the table below. Then when those subsections were fully implemented and tested, we linked them into the full datapath, testing that individual instructions ran, to finally checking branching and procedure calls, verifying output and fixing errors as they arose.

Stage	Components	Test inputs	Description	Outputs	Paths
Stage 1	PC Register and IorD MUX	Test PC Write: PCw = 1 ALUresult = 0x1234 IorD = x Test IorD ALUout: PC = 0x0000 ALUoutput = 0x1234 IorD = 1 Test IorD PC: PC = 0x1234 ALUoutput = 0x0000 IorD = 0	Test PC Write: Tests writing to the program counter. Test IorD ALUout: Set the PC and ALUout to different values and select with the MUX. Test IorD PC: Set the PC and ALUout to different values and select with the MUX	Test PC Write: PC = 0x1234 Test IorD ALUout: MEM address_in = 0x1234 Test IorD PC: Mem address_in = 0x1234	implementation\Datapath\Stage1.v implementation\Datapath\tb_stage1.v
Stage 2	Memory and MDR Register	Test MDR: MDRw = 1 Mem_out = 0x1234 Test Read: memw = 0 Addr_in = 0x1234 Mem[0x1234] = 0x1234 Test Write memw = 1 Addr_in = 0x1234 Dataw_in = 0x1234	Test MDR: Tests that MDR responds to mem_out. Test Read: Tests if memory reads proper value at address. Test Write: Ensures memory value is properly set.	Test MDR: MDR = 0x1234 Test Read: Mem_out = 0x1234 Test Write: Mem[0x1234] = 0x1234	implementation\Datapath\Stage2.v implementation\Datapath\tb_stage2.v
Stage 3	IR and CompCode Registers and Immediate Generator	Mem_out = 0x1234 Compcodew = 1	Ensures correct bits are being taken in IR and comp code and imm gen are connected appropriately.	Imm Gen = 0x0234	implementation\Datapath\Stage3.v implementation\Datapath\tb_stage3.v
Stage 4	RegOp MUX and Reggie	Regop = 1 Regw = 1 Mem_out = 0x1234	Make sure the MUX selects the correct bit and writes to Reggie.	Reggie = 0x1234	implementation\Datapath\Stage4.v implementation\Data path\tb_stage4.v
Stage 5	aluSrcA and B MUXes, ALU, and ALU_output and SR Registers	ALU source A = 0x1200 ALU source B = 0x0034 ALU_in = 1 SR_w = 1	Make sure the ALU outputs correct signals to all registers and their values update.	ALU Output = SR = 0x1234	implementation\Datapath\Stage5.v implementation\Datapath\tb_stage5.v

Datapath	All 5 stages	reset = 1;
		IN = 16'h0000;
		
		reset = 0;
		IN=16'hABCD;
		
		IN = 16'hBEEF;
		
		IN = 6'hDEAD;
			Take selected chunks of machine code and run using these inputs verifying outputs	Depends on provided code	implementation\Datapath\StagedDatapath.v
					
					implementation\Datapath\tb_StagedDatapath.v

Figure 10: Integration Plan And Testing Information

3 Green Sheet

Name	Type	Opcode	Function	English Description
LW	I-Type	0x0	Reggie = Mem[SE(imm)]	Load mem[addr] into Reggie
SW	I-Type	0x1	Mem[SE(imm)] = Reggie	Store the value of Reggie into mem[imm]
LMEM	I-Type	0x2	Reggie = Mem[Mem[SE(imm)]]	Load MEM[MEM[addr]] into Reggie
SMEM	I-Type	0x3	MEM[MEM[SE(imm)]] = Reggie	Store the value of Reggie into MEM[MEM[addr]]
ADD	I-Type	0x4	Reggie = Reggie + Mem[SE(imm)]	Increment Reggie by mem[imm]
SUB	I-Type	0x5	Reggie = Reggie - Mem[SE(imm)]	Decrement Reggie by mem[imm]
ADDI	I-Type	0x6	Reggie = Reggie + SE(imm)	Increment Reggie by the immediate
CMPI	I-Type	0x7	If Reggie < SE(imm): sr[1:0] = **01** If Reggie = SE(imm): sr[1:0] = **10** If Reggie > SE(imm): sr[1:0] = **11**	Compare Reggie and the immediate and update sr 1:0 accordingly
CMP	I-Type	0x8	If Reggie < Mem[SE(imm)] : sr[1:0] = **01** If Reggie = Mem[SE(imm)] : sr[1:0] = **10** If Reggie > Mem[SE(imm)] : sr[1:0] = **11**	Compare Reggie and the mem[imm] and update sr 3:0 accordingly
B	B-Type	0x9	If comp_code == sr[1:0] PC = PC + destination address Comp_code 00 == unconditional branch	Branch to dest_addr if comp_code equals SR[1:0]
BMEM	B-Type	0xA	If comp_code == sr[1:0] PC = PC + destination address Comp_code 00 == unconditional branch	Branch to location stored at address of immediate.
LLI	I-Type	0xB	Reggie[11:0] = SE(imm)	Load the immediate into the most significant bits of reggie. The remaining upper bits are 0.
SLLI	I-Type	0xC	Reggie = Reggie << SE(imm)[2:0]	Logical Left shift Reggie by the immediate
LI	I-Type	0xE	input = Reggie	Put the value of input into Reggie. Any immediate provided will be ignored.
SO	I-Type	0xF	output = Reggie	Put the value of Reggie into output

Bits	15:14	13:12	11:8	7:4	3:0
I-Type	Immediate				Opcode
B-Type	Comparison Code	Memory Immediate			Opcode

Type	Stack	Stack Pointer	Return Address	Text
Bits	0xFFFF:0x0400	0x03FFF:0x03FFE	0x03FFD:0x03FFC	0x3FFB:0x0000

Address	Name
RA	0x03FD
SP	0x03FF
A0 – A6	0x0404 – 0x040E
S0 – S7	0x0410 – 0x041E
T0 – T15	0x0420 – 0x043E

Figure 11: Green Sheet

4 Processor's Unique Features

4.1 Status Register

We felt that, even with our limited instruction set, it is important to still have an equivalent to RISC-V's BEQ, BLE, and BGE along with our unconditional branches. To contend with this, we chose to create a status register. Our Status Register (SR) is a 2-bit register that holds comparison codes. The codes map to the comparision values below:

Code	Meaning
00	Unconditional Branch
01	Reggie < value
10	Reggie == value
11	Reggie > value

The instruction CMP and CMPI set the SR. B and BMEM read the SR and compare it to the comparison code provided in the instruction. If the comparison codes match, the program executes the branch. If they do not, the processor does not branch.

4.2 No MDR

Our processor does not take advantage of an MDR. We did this to align with our design principle to make the processor as simple as possible. Not using an MDR removes one more register from our design, making it cheaper. For our design, we did not find many extreme disadvantages with this decision as our main goal is not to create the fastest processor possible, but a cheap and simple processor.

4.3 Special Memory Addresses

To keep the number of registers down, we used special memory addresses instead of special registers like SP, RA, etc. The green sheet shows what the current addresses are for the special addresses but this is by no means set in stone. Feel free to modify the assembler

or use different addresses altogether. Implementing special addresses means that we needed special instructions to handle using them. This is what BMEM, SMEM, and LMEM are. BMEM reads the information stored at the memory address provided in the immediate and branches there. SMEM reads the information stored at the memory address provided in the immediate and stores the value of Reggie there. LMEM is like SMEM but sets Reggie to MEM[MEM[IMM]].

5 Extra Features

5.1 Assembler

The assembler is our key extra feature. It allows for easy debugging of assembly code without manually assembling the program every time you make an edit. The source code for the assembler is located in implementation/asm2bin. The assembler allows the user to program labels into their code. For example, if a user wants to branch to line 0x044, they do not need to manually branch to that address, they can place a label there and branch to the label with an instruction like B 00 LABEL. Labels alone made programming the processor significantly easier, but along with the ease of use of the assembler, iterating code is extremely simple.

6 Benchmark Data

The tables below show the benchmark data from our processor.

Fmax	**Restricted Fmax**
23.67 MHz	23.67 MHz
281.37 MHz	281.37 MHz
337.84 MHz	337.84 MHz

Table 1: Cycle Time = 42.2 ns

Total Logic Elements	412 / 6272 (7 %)
Total Registers	127
Total Pins	66 / 180 (37%)
Total Virtual Pins	0
Total Memory Bits	16384 / 276480 (6%)
Embedded Multiplier 9-Bit Elements	0 / 30 (0%)
Total PLLs	0 / 2 (0%)

Table 2: Hardware Information

Trial Computing Given	**Number of Cycles**	**Number of Instructions**
11	500420	36604
13	248986	-
17	3592968	-

Table 3: Cycles Required to Execute RelPrime

Instructions	98
Stack	20
Input	2
Output	2

Table 4: Bytes Stored

The average CPI based on the table above is 13.65. Our execution time is 215 ms.

7 Conclusion

Our team set out to create a low-level assembly language with a cheap easy-to-program design. Our design successfully utilizes minimal hardware with only one main register, a single ALU, and one central control unit while still completing all operations necessary for a computer. The other registers we use are special purpose only, making our design easy to understand, and 2 of those 5 special purpose registers are only 2-bit registers. There are only 4 different outputs that those 2-bit registers can provide. Additionally, we ensured to properly document our special registers, ensurring ease of use by a programmer.

The MUXes, while we have six, are virtually instantaneous in terms of time, and allow a diverse range of inputs for each of our registers. Since each of our MUXes take in 2-bit control signals, they can take in 4 different inputs, and with the simplicity of our design, many of our registers can utilize more for future instructions making it flexible for the future needs of clients.

Things that our team accomplished when implementing this design were organization of testing and configuring our components with rigorous testing. Testing components in this architecture is vital due to known limited scalability. Testing components in each respective stage, and testing them all together in the datapath allows for proper testing and proper regularization. Using an assembler to test to see if our instructions are being read correctly in real-time made for efficient debugging for small programs such as Euclid's Algorithm or simply Multiplication.

Our design decisions were driven by the simplicity of our design. For the accumulator architecture, it is vital that clock cycles are lined up and that parameters are tuned. This is because if not, this leads to the accumulator possibly not performing optimally. Some additional difficulties our team faced were implementation errors such as assigning the wrong size, incorrect data preprocessing, and missing hardware components in integration testing while configuring stages of integration. Overall, while we experienced difficulties with keeping

the project simple as the size of the project became larger and more complex, we still were able to successfully utilize minimal hardware. So if you want to utilize a cheap, efficient, and effective system this is the design to have.

8 Useful Tests Collected

We selected to test multiply as a smaller segment of the final RelPrime code. Since it's using repeated addition to multiply, it successfully tested our more complex instructions recursively like branching and memory access. Multiply was also easier to test because we knew expected answers more easily than we did for relative prime numbers. This also gave us a glimpse of how efficiently our program would run.

```
LLI 0
SW S0
LLI 1
SW S1
FOR: CMPI 7
B 11 DONE
LW S0
ADDI 8
SW S0
LW S1
ADDI 1
SW S1
B 00 FOR
DONE: LW S0
SO
```

Appendix

Assembler

Formatting Requirements:

- Comparison codes must be in binary without any prefixes. Ex: 00, 11, 01.
- All other numbers must be in decimal.
- If a line has a label, there must be an instruction on that line as well.
- Labels must end in a colon and have a space after. Ex: LABEL: ADDI 24.

Running the Assembler: The assembler is written in Python3, and thus requires Python3 to run. To assemble code, place your assembly into asm.txt, located in the implementation/asm2bin directory. Next, run asm2bin.py with the command python asm2bin.py. This should place the assembled code into the bin.txt file in the same directory. Copy this machine code into memory.txt and run the program.

Using Labels: Labels are a key feature of the assembler. To implement a label, write it on the same line as the line that you want to branch to. All labels must end with a colon and have a space after. When branching to a line with a label, you should use the label's name without the colon instead of the destination address.