Gold-2324b-01

Accumulator Processor

Fred Hamer

Owen Leonard

Ian Samsa

Lucas Watson

Table of Contents

Processor Introduction

Our processor is based on an accumulator-type architecture, taking advantage of the simplicity of only using one main register to store computation results and data extracted from memory. Complementing this main register are four specialty data registers, used to store data outputs from memory, computation results from the ALU, a stack pointer, and a program counter. We used a minimalist design philosophy focused on simple programming and avoidance of superfluous instructions, making it as easy as possible for the programmer to use the processor and its instruction set. This manifested into a simple instruction set with only common and necessary instructions for programs, which resulted in only 13 instructions and 5 instruction types. We also applied this philosophy to our performance metrics, not only measuring our performance in terms of execution time, but also with regards to the simplicity of the instruction set, allowing the programmer to save time while programming and structuring code segments and procedures.

I/O

Our processor handles input by storing the input value into the main register, and then storing that value in memory in one of the dedicated argument slots to use in the future if need be. Output is taken straight from the main operating register, so all relevant data must be in the main register if the programmer desires to get or use that data elsewhere, including as a return value of a procedure.

RISC-V Differences

Jal	Jal utilizes direct addressing in this architecture. It has a 13-bit range, which covers more destination addresses than necessary since the processor memory only has 10-bit addressing.
Push/Pop	Push and pop are unique commands to our architecture that are used to store return addresses on the stack. They are used in conjunction to automatically update PC and SP, which prevents programmers from having to manually calculate SP adjustments to implement in their code.

ZE vs SE	Our Immediate Genie solely uses zero extension, whereas RISC-V primarily uses sign extension.

Core Instruction Format

R-Type		
10 bits [15:6]	3 bits [5:3]	3 bits [2:0]
Memory Address (Addr)	Func3	OP Code

C-Type			
2 bits [15:14]	8 bits [13:6]	3 bits [5:3]	3 bits [2:0]
Address of Compared Value	PC-Relative Destination Address (Addr)	Func3	OP Code

J-Type		
10 bits [15:6]	3 bits [5:3]	3 bits [2:0]
Direct Address (Addr)	Func3	OP Code

I-Type		
10 bits [15:6]	3 bits [5:3]	3 bits [2:0]
Immediate	Func3	OP Code

	P-Type		
10 bits [15:6]		3 bits [5:3]	3 bits [2:0]
-----		Func3	OP Code

Example code for common use cases

Loads a value from address 8 and stores it at address 10

```
0x0050: lw 8
0x0052:  sw 10
```

Adds 1 forever onto the register

```
0x0040:  addi 1
0x0042:  jal 64
```

Stores 2 at address 2 and compares address 2 with register to branch

```
0x0040:  addi 2
0x0042:  sw 2
0x0044:  add 2
0x0046:  bne 2 2
0x0048:  subi 1 (does not run)
0x004A:  sub 2
```

Instruction RTL

Add/Sub	Load/Store	C-Type	J-Type
IR = Mem[PC] PC = PC + 2	IR = Mem[PC] PC = PC + 2	IR = Mem[PC] PC = PC + 2	IR = Mem[PC] PC = PC + 2
B = ZE[IR[15:6]] A = 0 OUT = A + B	B = ZE[IR[15:6]] A = 0 OUT = A + B	B = ZE[IR[15:6]] A = 0 OUT = A + B	B = ZE[IR[15:6]] A = 0 OUT = A + B
MDR = Mem[OUT]	Load: Reg = Mem[OUT] Store: Mem[OUT] = Reg	A = 0 B = 2 OUT = A + B	PC = OUT;
B = MDR A = Reg Reg = A op B		MDR = Mem[OUT] Out = PC + ZE[IR[13:6]]	
		B = MDR A = Reg *ALU Compares MDR and Reg to see if branch*	
		If compare is successful, alter PC A = PC B = ZE [IR[13:6]] PC = A + B	
I-Type	**Push**	**Pop**	
IR = Mem[PC] PC = PC + 2	IR = Mem[PC] PC = PC + 2	IR = Mem[PC] PC = PC + 2	
B = ZE[IR[15:6]] A = 0 OUT = A + B	B = ZE[IR[15:6]] A = 0 OUT = A + B	B = ZE[IR[15:6]] A = 0 OUT = A + B	
B = ZE[IR[15:6]] A = Reg Reg = A op B	A = PC B = 2 OUT = A + B	A = SP B = 2 SP= A + B	
	Mem[SP] = OUT	PC = Mem[SP]	
	A = SP B = 2 SP = A - B		

Components with Respective Testbenches

Path for all: rhit-csse232-2324b-project-gole-2324b-01/implementation/<filename_here>

Memory: Memory.v | Memory_TB.v

Memory Wrapper: Memory_Wrapper.v

Control Unit: ControlUnit.v

ALU: ALU.v | ALU_TB.v

Immediate Genie: Immediate_Genie.v | Immediate_Genie_TB.v

Main Register: Register.v | Register_TB.v

Instruction Register: IR.v | Register_TB.v

Out Register: RegisterOUT.v | Register_TB.v

PC Register: RegisterPC.v | Register_TB.v

SP Register: RegisterSP.v | Register_TB.v

MDR Register: RegisterMDE.v | Register_TB.v

Testing Approach

When testing each component, and the full data path itself, we opted for fully implementing one component at a time. We decided to implement the Register component first since they require the least effort to create. Most registers differ from one another due to various muxes feeding differing data into each, so we broke registers into separate components based on intended usage. Once the data input to the register, routed through each mux with varying sources, was implemented we tested to see if each possible input would correctly alter the register data. We also tested to make sure that data in the register was only altered if the respective write control bit was 1 to ensure that we could control data changes with a high degree of accuracy.

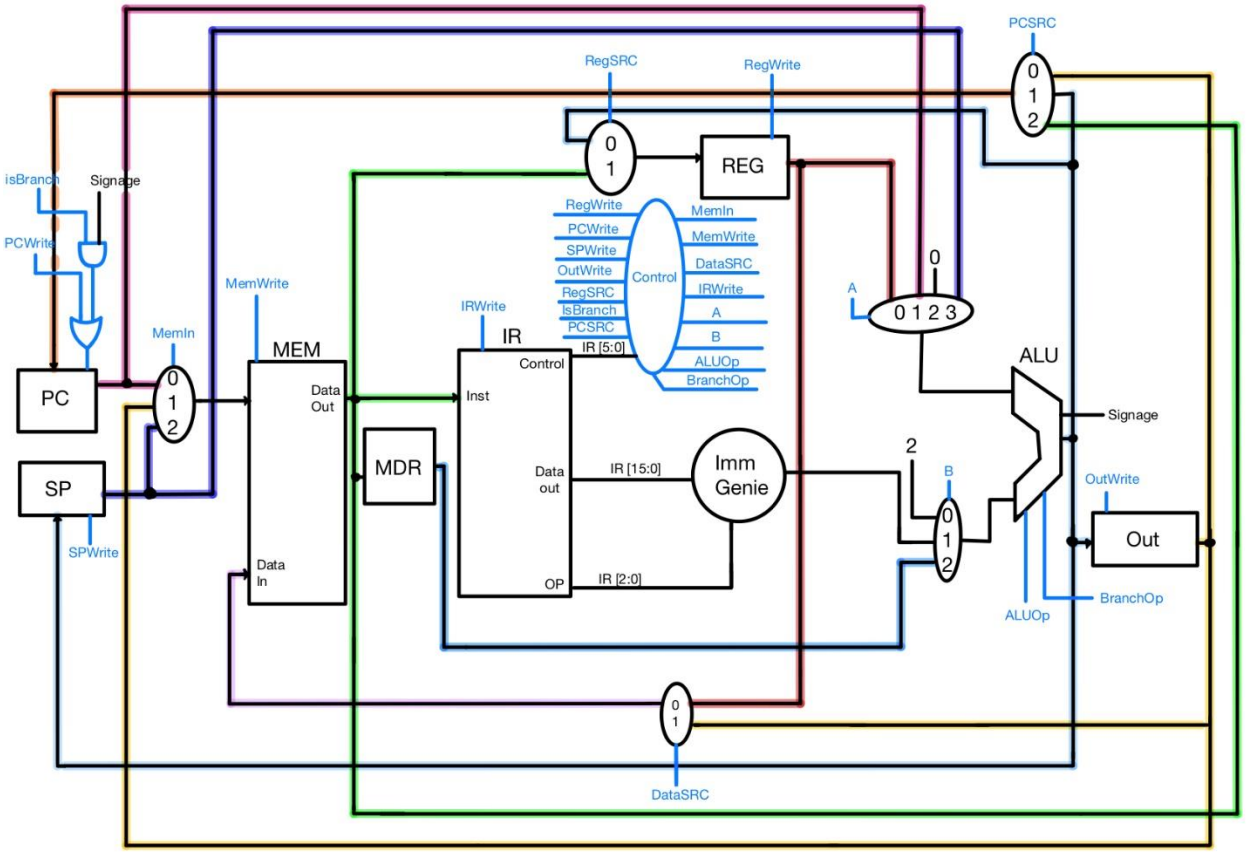
Next, we implemented the immediate genie which consisted of selecting various bits depending on the opcode in the instruction, or in other words, the first 6 bits of the instruction. The immediate genie was not very complicated to implement and therefore testing was a simple check to see if the correct bits were being selected by the component.

After the immediate genie, we moved onto working on the ALU. The ALU involved connecting two different muxes with a wide range of respective inputs and making sure the output of the ALU correctly performed specified arithmetic operations, i.e. addition or subtraction. Once created, we tested the ALU to ensure that each possible set of inputs to the source muxes could be selected. Then, we tested a variety of arithmetic operations by adding and subtracting various inputs and manually computing the correct output to cross-reference the ALU result.

The final component we implemented was memory. Memory was tested using the PC register since it was needed to select specific addresses within memory. Furthermore, memory has an input mux before it that selects the source to the memory input from either a data line or an address line in the case of MemRead vs MemWrite, so we ensured that this mux behavior was implemented first. Testing this component itself involved observing that the output of the mux matched the correct input line into memory. Memory testing was a little more complicated since we had to test storing a new value at a specific memory location as well. To test this, we used Model Sim and opened the memory analysis tab to manually verify that the correct addresses in memory were being altered.

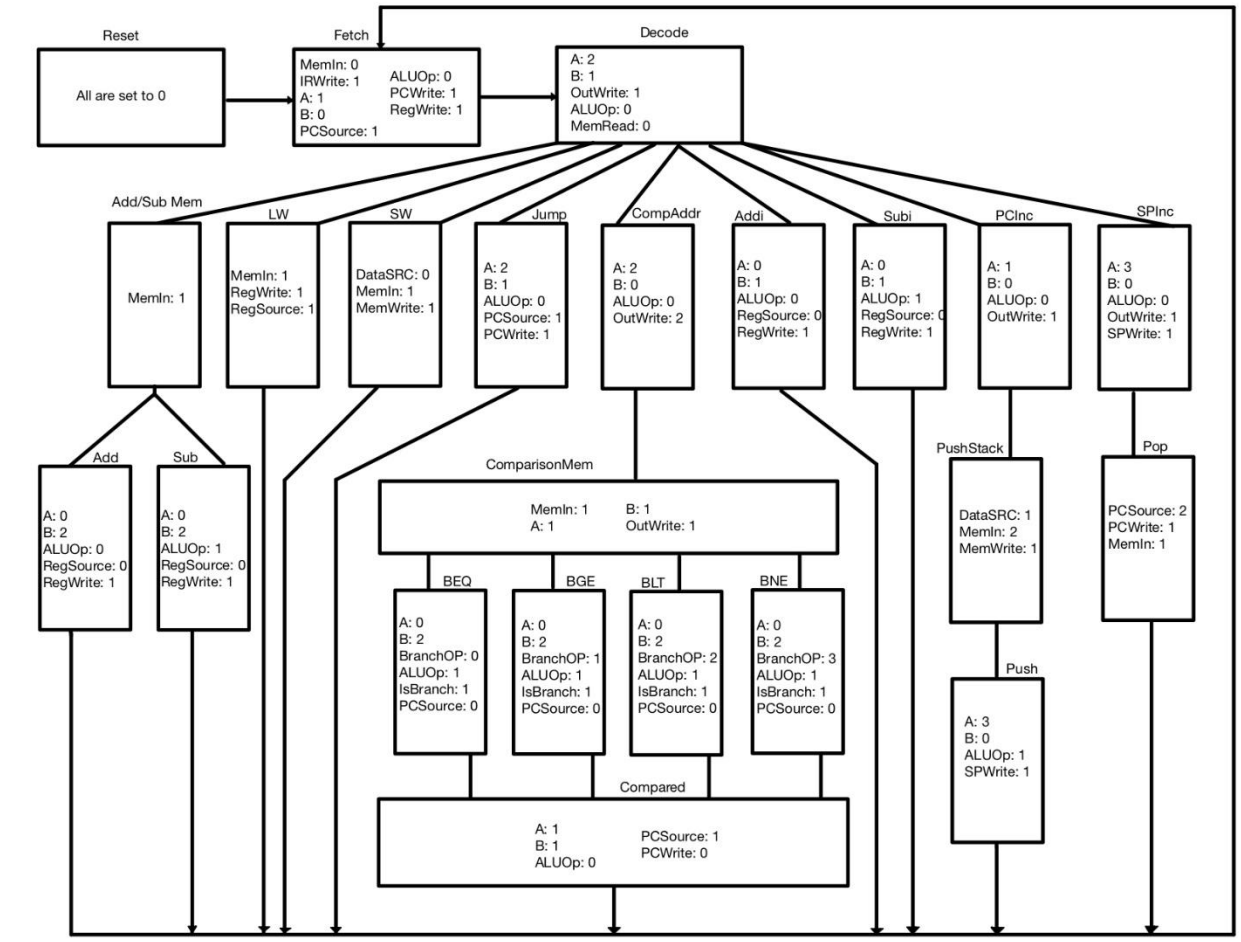
Finally, once all components were fully tested and completed, we linked them together and created our final data path. To be fully confident in our data path, we had to test every single instruction we created. We decided to implement and test instructions based on their type, making this a depth-first implementation. For example, we tested addi and subi first (I-types) before moving onto the next type, R-types. We repeated this testing process for each type, making necessary changes throughout until we were satisfied with all instructions. Once all instructions were tested and completed, we ran a full-scale test that utilized all instructions multiple times to see if it would output the correct values. Once this test ran successfully, the data path and all components were correctly implemented and ready for programmer use.

Datapath Diagram



Each instruction cycle begins with the fetch at PC, from there the processor is able to complete all operations. All control bits and paths are as labeled above, and support a full range of basic processor operations.

State Machine Diagram



SPWrite: Writes to SP when the signal is 1	MemIn: Selects between PC, OUT, and SP as the input address for Memory
MemWrite: Writes to Memory when the signal is 1	A: Selects between REG, PC, 0, and SP as the first ALU input value
IRWrite: Writes to IR when the signal is 1	B: Selects between 2, Immediate Genie, and MDR as the second ALU input value
OutWrite: Writes to OUT when the signal is 1	PCSource: Selects between OUT, the ALU output, and Memory output as the input for the PC value
PCWrite: Writes to PC when the signal is 1	IsBranch: Gets set to 1 when ready to potentially branch, as isBranch AND signage are required to write to PC
RegWrite: Writes to REG when the signal is 1	ALUOp: Determines between addition or subtraction as the operation the ALU completes
DataSRC: Selects between REG and OUT as the data input source for Memory	BranchOp: Selects between a BEQ, BGE, BLT, and BNE branch-type
RegSource: Selects between the ALU output and Memory as the input source for REG	

Integration Plan and Component Description

Our integration plan was a depth first approach, choosing to get one full instruction working and then build off that to implement the remaining instructions and instruction-types. We began with ADDI, as it seemed to be easiest as we just had to read an immediate input and add that with the value already in the register. Initially, the plan was to create the data path based on our diagram in this document, but only focusing on components needed for addi (some registers like MDR and SP are not needed for addi). This file was called Addi, with its testbench being Addi_TB. Since we believed our components were complete at this point, we only had to test that they were connected correctly in Addi, and we did this by utilizing a memory.txt file that contained an instruction for different addi calls in hex format. For example, one line would add 1, but further lines would add 2,3,4, etc. Once this worked, we moved onto subi, since it was another I-Type and quickly realized that it was already implemented correctly using the same components that had already been connected in the data path.

Initially, we decided to make different files for each instruction and combine them at the very end. However, after completing addi, we decided to us Addi.v as our main data path file and implement all instructions within this file. We then implemented each instruction type sequentially before moving onto the next, until all planned instructions had been implemented. We were then ready to add an input and output wire so users could run a procedure with specific values (arguments) they desired. We decided to make another component to separate these two wires, letting them change when needed, called OutputPort. We connected the input wire to the register so that it could be stored at a memory address using the instruction “sw.” The output wire would be directly connected to the main register as well so that whatever value was contained in the register at the end would display as the output as well.

After implementing the OutputPort file and connecting it to the other components in Addi, we were able to test relPrime and GCD in a testbench file that we named complete_datapath_tb. To do this, we first implemented all instructions needed in the memory.txt file so that it could run properly. This was done by writing our program in our assembly code and translating into hex format using our assembler. Next, we tested various input arguments ranging from 3 to 5040 to see if they would output correct values. All values portrayed the correct output confirming that we’d fully implemented the data path correctly and the implementation plan was finished.

Addressing Modes

Our processor uses direct addressing along with zero extension to perform our sole J-type instruction, JAL. This offers a wide range of addresses to jump to but requires the programmer to know the exact address of their destination as it gives them full control over this field. We also use PC-relative addressing along with zero extension to perform all C-type instructions, which combined with our eight bit branch input in C-type instructions allows for a wide range of addresses that the programmer can select and branch to.

Procedure Calling Convention

When calling another procedure/function with our instruction set, there are a few guidelines to ensure that everything executes correctly and achieves the desired outcome of the programmer. First, the PUSH instruction is used to put the PC on the stack. Next, the JAL instruction is called, which will jump to the desired instruction by setting PC to the desired address. The called procedure is then required to have the POP instruction as its last instruction, as this will go onto the stack, set PC to the previously stored value, and then increment the stack pointer. This PC value, during the PUSH instruction, is automatically set to the address after the JAL instruction, so it always goes to the right place in the code.

Memory Map

Our memory layout includes a hardcoded 0 value at the address 0x0000, a comparison argument for what C-types compare to at address 0x0002, general arguments and data storage that range from addresses 0x0004 to 0x003E, our text and beginning of where the instructions start at 0x0040, and finally our stack which starts at the “top” of memory at address 0x0400.

0x0400 Stack | | | V Dynamic Data	
0x0040	Text
0x0004 – 0x003E	Arguments
0x0002	Immediate Value (Comparisons)
0x0000	0 Value

Green Sheet

Base Integer Instructions

Inst	Name	FMT	funct3	Opcode	Description
add	ADD	R	000	000	R = R + mem[ZE(mem_addr)]
sub	SUB	R	001	000	R = R – mem[ZE(mem_addr)]
lw	Load Word	R	010	000	R = mem[ZE(mem_addr)]
sw	Store Word	R	011	000	mem[ZE(mem_addr)]= R
beq	Branch ==	C	000	001	if(R == mem[ZE(compare_addr)]) PC = PC + New Address
bne	Branch !=	C	001	001	if(R!= mem[ZE(compare_addr)]) PC = PC + New Address
blt	Branch <	C	010	001	if(R < mem[ZE(compare_addr)]) PC = PC + New Address
bge	Branch >=	C	011	001	if(R >= mem[ZE(compare_addr)]) PC = PC + New Address
jal	Jump and Link	J	000	011	PC = newAddr
addi	ADD Immediate	I	000	010	R = R + imm
subi	SUB Immediate	I	001	010	R = R - imm
push	Pushes an address	P	000	100	PC = PC + 4 Mem[SP] = PC SP = SP – 2
pop	Pops an address	P	001	100	SP = SP + 2 PC = SP[Address]

Core Instruction Format

15 13 12 6 5 3 2 0

Memory Address		Func3	Opcode	**R-Type**
Immediate		Func3	Opcode	**I-Type**
------		Func3	Opcode	**P-Type**
Direct Address		Func3	Opcode	**J-Type**
Comp Addr	PC-Relative Destination Address	Func3	Opcode	**C-Type**

Opcodes, Base Conversion

Binary	Hex	Opcode
000 000	00	add
000 001	01	beq
000 010	02	addi
000 011	03	jal
000 100	04	push
000 101	05	
000 110	06	
000 111	07	
001 000	08	sub
001 001	09	bne
001 010	0A	subi
001 011	0B	
001 100	0C	pop
001 101	0D	
001 110	0E	
001 111	0F	

Binary	Hex	Opcode
010 000	10	lw
010 001	11	blt
010 010	12	
010 011	13	
010 100	14	
010 101	15	
010 110	16	
010 111	17	
011 000	18	sw
011 001	19	bge
011 010	1A	
011 011	1B	
011 100	1C	
011 101	1D	
011 110	1E	
011 111	1F	

Memory Allocation

0x0400	**Stack** | v ^ | **Dynamic Data**
0x0040	**Text**
0x0004 – 0x003E	**Arguments**
0x0002	**Immediate Value**
0x0000	**0 Value**

Unique Features

One of our unique features is our automatic PUSH and POP instructions. They are as simple to use as writing PUSH and POP, as the rest is automatically taken care of by the control unit. PUSH automatically pushes the PC onto the stack and decrements SP down to the next open memory slot. This makes it easy to store the return address of the program counter when using JAL, so that POP can be performed, which takes the value stored at SP, sets PC to it, and increments SP up to the next value stored on the stack.

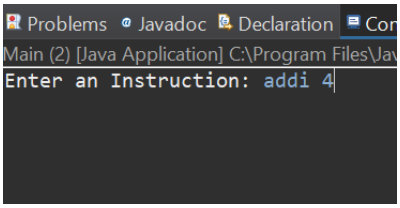
Another unique feature is the use of the memory address 0x0002 as the comparison address for the C-type instructions. It requires the user to put whatever they want to compare REG to in that slot, but due to the minimal number of bits required to represent the address in binary and in the subsequent machine code, it allows for a wide range of addresses to branch to at eight bits. This allows for larger programs to be written that utilize a larger range of branch addresses. It also is no extra cost in the Datapath, as the B mux already has the value 2 hardcoded to one of its inputs for PC incrementing, so adding 0 and 2 to get the address 0x0002 is low-cost and routing this result to the MemIn mux to access that address in memory is a similarly inexpensive individual operation.

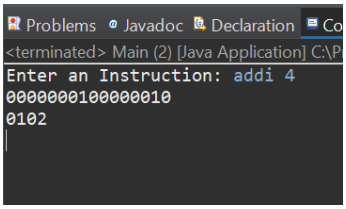
Extra Features

Assembler

For an extra feature we created an instruction assembler in java which can either take manual input from a user and output the respective binary and hex for that instruction or read from a txt file local to the users machine, parse the file, and then print all instructions from that file. Below are some screenshots from the assembler and example outputs to the console.

Manual Input





At the top of file one can specify if they want manual input or txt file input by switching a Boolean labeled “manual” to either true or false. If the Boolean is set to true, an example of the assembler process can be seen above.

Read From txt File

```
if(!manual) {
    //Read a txt file with the assembly code
    String filename = "C:\\Users\\watsonlm\\OneDrive - Rose-Hulman Institute of Technology\\Documents\\csse232\\projectCode.txt";
    //Holds labels like "GCD:" or "RELPRIME:" and maps them to their address in the file
    Map<Integer, String> labels = new HashMap<>();
    labels = parseForLables(filename, labels);
    //Parses instruction for binary and hex representations
    parseForInstructions(filename, labels, instsBinary);
    instsHex = binaryToHex(instsBinary);
    System.out.println("----------------------");

    //This for loop is for the 64 dead lines at the beginning of our program
    //To make copy and pasting easier, we pad the first 64 lines with "0000"
    for(int i = 0; i < 64; i++) {
        System.out.println("0000");
    }

    //This for loop adds lines of "0000" in between every hex instruction as our PC increments by 2
    //Makes copy and pasting easier
    for(int i = 0; i < instsHex.size(); i++) {
        System.out.println(instsHex.get(i));
        System.out.println("0000");
    }
    System.out.println("----------------------");
    System.out.println(labels);
}
```

If the Boolean is set to false, the user can specify the file path to the txt file on their machine and the block of code above will run. Below are some examples of input from a txt file and the output to the console.

```
START: 64
sw 4
lw 0
addi 2
sw 6
jal 84

RELPRIMELOOP: 74
beq 2 74
lw 6
addi 1
sw 6
jal 84

GCD: 84
lw 0
sw 2
lw 4
bne 2 10
lw 0
addi 1
sw 2
lw 6
jal 74
```

```
0000000100011000
0000000000010000
0000000010000010
0000000110011000
0001010100000011
1001001010000001
0000000110010000
0000000001000010
0000000110011000
0001010100000011
0000000000010000
0000000010011000
0000000100010000
1000001010001001
0000000000010000
0000000001000010
0000000010011000
0000000110010000
0001001010000011
```

```
0000
0000
0000
0000
0000
0000
0118
0000
0010
0000
0082
0000
0198
0000
1503
0000
9281
0000
0190
0000
```

```
{64=START, 84=GCD, 74=RELPRIMELOOP}
```

The assembler will first print all instructions in binary, then print all instructions in hex with some clever padding of “0000” lines to make copy and pasting into testing software much easier, then print all labels mapped to their address in the file for easy debugging and code writing.

A more detailed and in-depth description of the code base and instructions on how to use the assembler can be found in the appendix of this document and the Gold-2324b-01 implementation repository linked as a PDF document.

Benchmark Data

Total Bytes: 90

Total # Instructions: 116,585

Total # Cycles: 418,704

Average CPI: 3.58

Cycle Time: 11.6ns

Execution Time: .004856 seconds

Total Logic Elements: 19,992 (90%)

Total # Registers: 16,503

Total Memory Bits: 0

Our results are within the expected range that we were hoping to be in. They are competitive with what we were hoping to achieve and support our design philosophy and ideology. The CPI is slightly higher than we'd initially designed for, but the Cycle Time and Execution Time of the RelPrime program are well within our design targets.

Conclusion

Ultimately our accumulator-based architecture process was successful in terms of all established metrics, including the ability to execute all desired instructions, run in a timely manner with a competitive execution time, and achieving a primary design philosophy of keeping a simplified instruction set to make the programming simple using this processor. The design process saw some slight change and variation from the original planned instruction set, adding P-Type instructions, and adjusting various control bits and details to work with our RTL, Datapath, and Control Unit. Our processor uses a Multi-Cycle Datapath (represented by our Multi-Cycle RTL) to reduce hardware cost and volume and maintain efficient functionality within each cycle. This, combined with our Control Unit makes it easy to determine what is happening in each step, which helped us simplify our Control Unit and RTL. Our implementation avoided any major issues that would've required significant reconstruction and reconsideration of our processor, but we did have to revise our instruction set, RTL, Datapath, and Control Unit up until the final implementation to adapt to small roadblocks and errors that we encountered along the way. We are ecstatic with the final processor and what it can accomplish, as well as the dedication and teamwork from each member to produce a refined final product that meets and even exceeds our target expectations.

Appendix

A more robust set of instructions for the assembler as well as an in-depth code base description can be found at the following link or in the Gold-2324b-01 implementation repository linked as a PDF document.

AssemblerInstructions.pdf