



NOVEMBER 13, 2023

PIKACHIP PROCESSOR

TEAM YELLOW 2324A (DINO NUGGIES)

SPENCER HALSEY

BLAISE SWARTWOOD

REILLY MOONEY

JACOB SCHEIBE

Table of Contents

Table of Contents.....	1
Design Philosophy.....	2
Measuring Performance	3
Conventions	3
Addressing Mode	3
Calling Conventions.....	3
Naming Conventions	4
Registers	4
Instruction Information.....	5
Instruction Types.....	5
Instructions List	6
Unique Instructions.....	8
Branching	8
Branching Template.....	8
Memory Allocation	11
Assembly Language Examples.....	12
RTL.....	13
R Type.....	14
I Type	14
U Type	14
RTL Reading and Interpreting.....	15
Component Designing and Testing.....	15
Component Table	15
Mux	16
Register	17
Register File	17
ALU	18
Immediate Generator	20
Memory.....	21
Datapath.....	22
Control	23

ALU Control	23
Control	24
State Diagram.....	26
Integration Plan.....	26
Unit Testing	26
Component Integration.....	26
ALU, ALUControl, ImmediateGenie Integration Test.....	27
Handling I/O	28
Green Sheet for ISA.....	Error! Bookmark not defined.
Unique Project Features.....	29
Key instructions.....	29
Extra Features	29
Assembler	29
Assembler Pseudoinstructions.....	31
Assembler Tags/Branching:.....	32
RelPrime Testing Data	32
Useful Test Results	Error! Bookmark not defined.
Conclusion.....	32
Appendix	33
Assembler Tag + Pseudoinstruction example.....	33
Reference Sheet	36
NOTES:.....	Error! Bookmark not defined.

Design Philosophy

Our design strives to strike a balance between efficiency and ease of use for the programmer. We want our processor to be able to run quickly, while still providing the tools necessary for the programmer to have an easy time implementing code. To this end, we decided to use a Load and Store type format to allow the programmer to easily understand our instruction set, while still allowing for fast instruction calls. Our philosophy revolves around the idea that if there is a way to

make something easier for the programmer to understand without giving up too much performance, we'll do it.

Measuring Performance

We will measure our performance based on versatility and ease of use (programmability) of our processor. The instructions are straight forward and simple to use while maintaining acceptable execution time. In terms of quantifiable numbers, we'll measure our execution time of a common function, Euclid's algorithm. If Euclid's can be easily implemented, and executed in a reasonable amount of time, we'll consider our processor a success.

Conventions

Addressing Mode

All of the instructions in the processor are PC relative. IE: When we branch to a new location, it will use the value of PC plus some immediate to designate the new address. The two instruction types that use an immediate, I and U, are PC relative and the immediate bits are signed. The only time when we don't use PC to determine the address is when we use the jump and link register operation, which directly adds the immediate to a register value.

Calling Conventions

The stack pointer (sp) must always be manually expanded and restored after use. It is imperative that the programmer always returns the stack to its original value after use. Argument and temporary registers are assumed to not hold the same value after jumping to a different function. The programmer must manually save those if they want to keep the values. Saved registers must always be saved and restored at the beginning and end of every function call by the programmer if they are used. It is assumed that the callee will keep these values the same. Arguments are passed into a function using a0-a4, and values are returned from a function through a0-a2. Return values for addresses are generally stored in ra. Example assembly code for basically calling conventions (see RelPrime for a more detailed example):

Address	Assembly	Comments
0x0000:	lbi t0, 8	Moving stack pointer
0x0002	sub sp, sp, t0	

0x0004	sw ra, 0(sp)	Storing ra
0x0006	sw s0, 4 (sp)	Storing saved
...
0x0026	lw ra, 0(sp)	Loading ra
0x0028	lw s0, 4 (sp)	Loading saved
0x002A	lbi t0, 8	Moving stack pointer
0x002C	add sp, sp, t0	

Naming Conventions

The naming conventions should try to express all the information needed in a clear and concise manner. It should follow the general rules:

- Convey the information on what file does concisely
- Follows the file naming conventions

The common files should be named as follows:

- Modules: mod_(name)
- Test Benches: (name)_tb
- Inputs: in_(name)
- Outputs: out_(name)
- Wires: [x:0] w _(name) (x is the number of bits - 1)
- Reset: RST
- Control Signals: Name[0:0]
- Registers: [x:0] reg_(name) (x is the number of bits - 1)
- Clock: CLK_# (Use _# is any more than one are needed)

Registers

Register	Name	Description	Saver
x0	zero	Zero constant	<hr/> Caller
x1	ra	Return address	
x2	sp	Stack pointer	

x3-x5	a0-a2	Fn args/return values	Callee
x6-x7	a3-a4	Fn args	Caller
x8-x10	s0-s2	Saved registers	Caller
x11-x13	t0-t2	Temporary registers	Callee
x14	br	Branch Compare register	Caller
x15	at	Assembler Temporary	Caller

Note: The br register, used for branching, stores the branching comparison. To use this register, add one of the comparison values to the register. The beq instruction will then handle the comparison between the argument given in beq and the value stored in the branch register. The argument given in the beq instruction is the value on the right of the comparison, while the value in the br register is the right side of the comparison.

Instruction Information

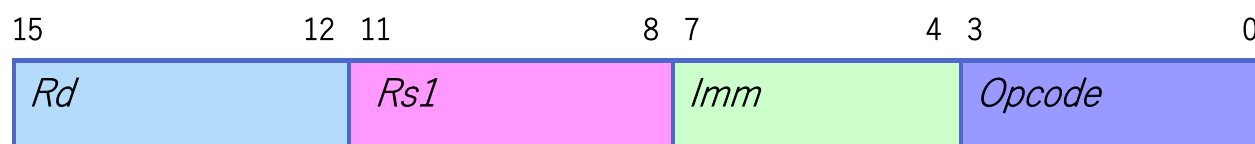
Instruction Types

Ralts-Type (R-Type):



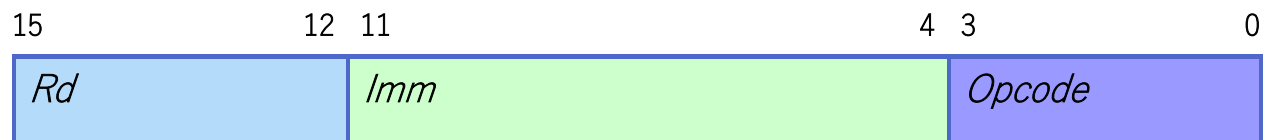
The Ralts -Type instruction format is mainly used for basic mathematical operations. The format contains a 4-bit operation code to distinguish what operation is being called, two 4 bit registers for operation arguments, and a 4 bit destination register for the result of the operation.

lvysaur-Type (I-Type):



The Irvysaur-Type instruction format is mainly used for immediate based operations, such as add immediate or store word. The format contains a 4-bit operation code, a 4-bit immediate value, a 4-bit argument register, and a 4-bit destination register.

Umbreon-Type (U-Type):



The Umbreon-Type instruction format is mainly used for branching or immediate shifting operations. The format contains a 4-bit operation code, an 8-bit immediate value, and a 4 bit destination register.

*To convert instruction type into machine language, take the value, and convert into binary. EX: If you had add x1, x2, x3, you would convert it like this:

adds opcode = 0000

X1 = 0001

X2 = 0010

X3 = 0011

Then put it into the correct format based on its instruction type, in this case, R.

Rd Rs1 Rs2 Opcode

0001 0010 0011 0000

Instructions List

Instruction	Instruction Name	Description	Meaning	Opcode	Type	Notes
-------------	------------------	-------------	---------	--------	------	-------

Add	add	$R[rd] = R[rs1] + R[rs2]$	$Rd = rs1 + rs2$	0000	R	
Subtract	sub	$R[rd] = R[rs1] - R[rs2]$	$Rd = rs1 - rs2$	0001	R	
And	and	$R[rd] = R[rs1] \& R[rs2]$	$Rd = rs1 \& rs2$	0010	R	
Inclusive or	or	$R[rd] = R[rs1] R[rs2]$	$Rd = rs1 rs2$	0011	R	
Add Immediate	addi	$R[rd] = R[rs1] + SE(imm)$	$Rd = rs1 + imm$	0100	I	
Load Word	lw	$R[rd] = M[R[rs1] + SE(imm)]$	$Rd = Memory[rs1 + imm]$	0111	I	
Store Word	sw	$M[R[rs1] + SE(imm)] = R[rd]$	$Memory[rs1 + imm] = rd$	1000	I	Rd is not actually used as the rd, but as the value to set equal to
Branch if equal	beq	if($rs1 == BR$) $PC += SE(imm) << 1$	If ($rs1 == BR$) go to $PC + imm$	1001	U	
Jump and link	jal	$R[rd] = PC + 2$ $PC += SE(imm) << 1$	$Rd = PC + 2$; Go to $PC + imm$	1100	U	PC relative
Jump and link register	jalr	$R[rd] = PC + 2$ $PC = R[rs1]$	$Rd = PC + 2$; Go to $rs1 + imm$	1101	I	Imm = 0, Register relative
Load upper immediate	lui	$R[rd] = SE(imm), 8'b0$	$Rd[15:8] = imm$	1110	U	
Load bottom immediate	lbi	$R[rd] = 8'b0, imm$	$Rd[7:0] = imm$	1111	U	works like lui, except loads the lower 8 bits of the immediate
Shift immediate	si	If($imm[4]$ is 0)	If($imm[4]$ is 0) $Rd = Rd << imm$	0101	U	$imm[4]$ determines if we shift right (1)

		$R[rd] = R[rd] \ll imm[3:0]$ Else (imm[4] is 1) $R[rd] = R[rd] \gg imm[3:0]$	Else (imm[4] is 1) $Rd = Rd \gg imm$ If(imm[5] is 0) Shift logical Else(imm[5] is 1) Shift arithmetic			or left (0) (based on sign) imm[5] determines if we shift logical (0) or shift arithmetic (1)
Load Input	Lin	$R[rd] = INPUT$	$Rd = INPUT$	0110	U	
Load Output	Lout	$OUTPUT = R[rd]$	$OUTPUT = Rs1$	1010	I	

Unique Instructions

Branching

Our processor only provides a beq instruction, which means that other operations such as branch if less than (blt), branch if not equal to (bne), branch if greater than or equal to (bge) etc., have to be done using beq. Below is a template with the most efficient ways to perform each one. Additionally, you may notice that beq is a U-Type instruction, meaning it only uses one register. To account for this, the br register holds the other value used for comparing. The branching template also demonstrates proper use of the dedicated register.

Branching Template

Note: The following branching templates are pseudo-instructions

Beq

Assembly	High-Level
add br, t0, x0 beq t1, Branch Branch: add t0, t0, t1	If (a == b) { a = a + b }

Bne

Assembly	High-Level
add br, t0, x0 beq t1, Branch add t0, t0, t1 Branch:	If (a != b) { a = a + b }

Blr

Assembly	High-Level
sub t2, t0, t1 slr t2, 31 //imm[4] is 1 to shift right //imm[3:0] is 1111 to shift 15 bits add br, x0, t2 addi t2, x0, 0 beq t2, Branch add br, x0, t0 beq t1, Branch add t0, t0, t1 Branch:	If (a < b) { a = a + b }

Bgt

Assembly	High-Level
<pre> sub t2, t0, t1 si t2, 31 //imm[4] is 1 to shift right //imm[3:0] is 1111 to shift 15 bits add br, x0, t2 addi t2, x0, 1 beq t2, Branch add br, x0, t0 beq t1, Branch add t0, t0, t1 Branch: </pre>	<pre> If (a > b) { a = a + b } </pre>

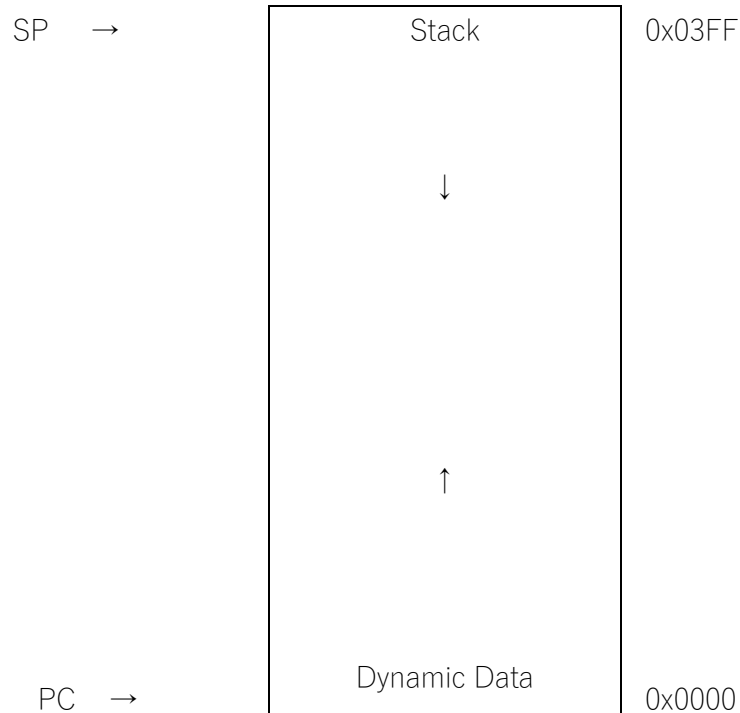
Bge

Assembly	High-Level
<pre> sub t2, t0, t1 si t2, 31 //imm[4] is 1 to shift right //imm[3:0] is 1111 to shift 15 bits add br, x0, t2 addi t2, x0, 1 beq t2, Branch add t0, t0, t1 Branch: </pre>	<pre> If (a >= b) { a = a + b } </pre>

Ble

Assembly	High-Level
<pre>sub t2, t0, t1 si t2, 31 //imm[4] is 1 to shift right //imm[3:0] is 1111 to shift 15 bits add br, x0, t2 addi t2, x0, x0 beq t2, Branch add t0, t0, t1 Branch:</pre>	<pre>If (a <= b) { a = a + b }</pre>

Memory Allocation



```

/*
Design: Mux
File Name: Mux.v
Function: 4:2 or 2:1 Mux
Author: Spencer Halsey
*/

```

Assembly Language Examples

Address	Assembly	Machine Code	Comments
0x0000	add x11, x0, x12	1011 0000 1100 0000	// add an instruction
0x0002	sub x11, x12, x0	1011 1100 0000 0001	// subtract an instruction
0x0004	and x11, x0, x12	1011 0000 1100 0010	// and two registers
0x0008	or x11, x0, x12	1011 0000 1100 0011	// or two registers
0x000C	addi x11, x0, 2	1011 0000 0010 0100	// add a register and immediate
0x0010	lw x11, 4(x12)	1011 1100 0100 0111	// loads a value from memory
0x0012	sw x11, 4(x12)	1011 1100 0100 1000	// stores a value to memory
0x0014	L: beq x11, L	1011 0000 0000 1001	// branches if reg. is == compare reg.
0x0020	L: jal x11, L	1011 0000 0000 1100	// jump and link
0x0022	jalr x11, 0(x1)	1011 0001 0000 1101	// jump and link reg.
0x0024	lui x11, 2	1011 0000 0010 1110	// load upper 8 bits
0x0028	lbi x11, 2	1011 0000 0010 1111	// load bottom 8 bits
0x002C	sai x11, 2	1011 0000 0010 0101	// arithmetic immediate shift
0x0030	sli x11, 2	1011 0000 0010 0110	// logical immediate shift

Addition using 16-bit Immediate

Instruction	Binary
lui x11, 2	1011 0000 0010 1110
lbi x11, 2	1011 0000 0010 1111

Looping

Instruction	Binary
0x0012: relLoop (Some instruction)	-
0x001E: jal x0, relLoop	0000 1111 0100 1100
Instruction	Binary
beq a0, END	0011 0000 0110 1001
0x0020: END (Some instruction)	-

Branching

Instruction	Binary
add br, s2, x0	1110 1010 0000 0000
beq a0, END	0011 0000 0110 1001
0x0020: END (some instruction)	-

Recursion

Instruction	Binary
0x0012, CountDown:	-
add br, t0, x0	1110 1011 0000 0000
beq a0, END	0011 0001 1110 1001
sw a0, 0(sp)	0011 0000 0000 1000
addi a0, a0, -1	0011 0011 1111 0100
jal x0, CountDown	0000 1111 0100 1100
0x001E, END: (some instruction)	-

RTL

R Type

R Type	add	sub	and	or
	$IR \leq mem[PC]$ $PC \leq PC + 2$			
	$A \leq Reg[IR[11:8]]$ $B \leq Reg[IR[7:4]]$			
	$ALUOut \leq A + B$	$ALUOut \leq A - B$	$ALUOut \leq A \& B$	$ALUOut \leq A B$
	$Reg[IR[15:12]] = ALUOut$			

I Type

I Type		addi	Lw	sw	jalr
		$PC \leq PC + 2$ $IR \leq mem[PC]$			
		$A \leq Reg[IR[11:8]]$ $B \leq Reg[IR[15:12]]$			
		$ALUOut \leq A + SE(IR[7:4]) \text{ (imm)}$			$PC \leq A$ $Reg[IR[15:12]] \leq PC$
		$Reg[IR[15:12]] \leq ALUOut$	$MDR \leq Memory[ALUOut]$	$Memory[ALUOut] \leq B$	
			$Reg[IR[15:12]] \leq MDR$		

U Type

U Type	Beq	Lui	Lbi	Jal	Si
	$PC \leq PC + 2$ $IR \leq mem[PC]$				
	$A \leq Reg[IR[11:8]]$ $B \leq Reg[IR[15:12]]$				
	$A \leq Reg[IR[15:12]]$ $B \leq Br$ $Imm = SE(IR[11:4])$ $\ll 1$	$Imm = SE(IR[11:4])$ *Depending on type of shift, we might not need to SE	$Imm = IR[11:4]$	$ALUOut \leq PC + SE(2 * immediate)$	$A \leq Reg[IR[15:12]]$ $ALUOut \leq A \gg \ll imm[0:3]$
	$Target = PC + Imm$	$Result = Imm \ll 8$	$Result = IR[15:12] + Imm$	$PC \leq ALUOut$	$Reg[IR[15:12]] \leq ALUOut$
	If $(A == B)$ $PC = Target$	$Reg[IR[15:12]] = Result$	$Reg[IR[15:12]] = Result$	$Reg[IR[15:12]] \leq PC$	

RTL Reading and Interpreting

To read our multicycle RTL, the first two steps of any instruction is the same. Then, we break up significantly different instructions into their own categories and give them their unique RTL. Since we were originally planning on pipelining, we had to adjust and then readjust our RTL back to multicycle. Otherwise, read down each box for an instruction and boxes that span multiple instructions are applicable to all those instructions at that stage.

Component Designing and Testing

Component Table

Component	Inputs	Outputs	Behavior	RTL Symbols
Register	CLK RST In_Data[15:0]	Out_Data[15:0]	Accepts input on clock uptick if the operation is supposed to write to it, stores the input, and uses it as its output Takes in the input on the rising CLK edge, outputs the stored value on the falling edge	PC, IR, A, B, ALUout, MDR, SP
Register File	CLK RST In_RegWrite[0:0] In_ReadReg1[3:0] In_ReadReg2[3:0] In_WriteAddr[3:0] In_Data[15:0]	Out_ReadData1[15:0] Out_ReadData2[15:0]	Holds all of the registers used by the processor. Allows reading and writing of registers simultaneously. Stores/Reads a register on the rising CLK edge, reads the register on the falling CLK edge	
Instruction Register	In_Instruction[15:0]	Out_Opcode[3:0] Out_Reg1[3:0] Out_Reg2[3:0] Out_RegRd[3:0]	Decodes the instruction, and outputs the given information to other components	
Memory	CLK RST	Out_MemData[15:0]	Keeps track of return addresses and used for	Mem

	In_StackWrite[0:0] In_StackRead[0:0] In_StackOp[0:0]		general storage by the programmer Stores the input on the rising CLK edge, reads the value on the falling edge	
ALU	RST In_A[15:0] In_B[15:0] In_ALUCtrl[3:0]	Out_ALUResult[15:0] Out_Zero[0:0]	Will perform operations such as, addition, subtraction, or, and	+, -, /, , &, >>, <<,
Control	CLK RST In_Instructions[4:0]	Out_IorD[0:0] Out_MemRead[0:0] Out_MemWrite[0:0] Out_IRWrite[0:0] Out_RegWrite[0:0] Out_ALUSrcA[0:0] Out_ALUSrcB[0:0] Out_ALUCtrl[3:0]	Receives a 5-bit opcode, handles the input, and outputs the correct values as signals to complete the correct instruction Takes in the opcode input on the rising CLK edge, outputs the control value on the falling edge	
ALUControl	In_Si[1:0] In_Inst[3:0]	Out_ALUCtrl[3:0]	Gets 2-bits from the Immediate Genie and a 4-bit opcode. This is only considered when doing a shift operation.	
Mux	In_Select[0:0] In_A[15:0] In_B[15:0]	Out_Result[15:0]	Takes 2 inputs and outputs the value corresponding to the control signal	
ImmGen	In_Inst[15:0]	Out_Imm[15:0] Out_Si[1:0]	Takes a 4 or 8-bit input and sign extends it, and outputs it	Imm, SE

Mux
Build

We are planning to build a 2:1 Multiplexer that selects between two different 16 bit input values and outputs the selected 16 bit value. It will be built using a switch case for the select. It then will output into a register when the selection has finished.

Testing Mux

We will test the two possible cases for the Multiplexer. Firstly, when the select bit is 0, it should select the first input. We will check to see if it matches the correct value. The program will then wait some arbitrary amount, then perform the second test. The second test is when the select bit is 1, it should select the second input. We will check to see if it matches the correct value. Then we will print all failures.

Register

Build

We are planning to build a 16 bit register with a clock and a reset. It will write the 16 bit input value on the positive clock edge. Unless it is provided with a reset signal, in which it will set the result to zero.

Testing

Firstly, we will test inputting some value into the register, with reset being inactive. On the positive clock edge, it will set the value of the register. We will then check to make sure that the correct value has been set. It will then wait some arbitrary amount of time. Secondly, we will test the reset signal on the register. We will input some non-zero value into the register, and input a active reset signal (1). On the positive clock edge, it will set the value of the register to zero. We will then check to make sure that the correct value has been set.

Register File

Build

We are planning to build a container file which contains all of our 16 dedicated registers (defined on page 3). The Register File will be able to read values from two registers on a negative clock edge, and write data/a register to a register on a positive clock edge. We will specify which registers need to be read/written to with signals. It will also have a signal to determine if it should write to a register.

Test

To test the ALU, we'll start with testing the reset signal. Firstly, we'll input an active (1) reset signal to the register file. We'll then check all of the registers to make sure that they have been zeroed on the positive clock edge. Then we'll read one register on the negative clock edge, and make sure that the value is correct. Then we will write to a register with the given data (with RegWrite being 1) and check to make sure the value is correct. Then we will try to write to another register (with RegWrite being 0), and ensure that no value is written. We will then write to a register (with RegWrite being 1) with the given register, ensure the value is correct, then reset to ensure that the value is being reset.

ALU

Build ALU:

To build the ALU, we are going to take in two 16 bit values and the ALUCtrl signal. The ALUCtrl will determine what operation we are going to perform as noted in the table below. Then, a switch case statement will be made for each of them and then the resulting operation will be performed and stored in Out_ALUResult.

In_ALUCtrl	Operation
0000	add
0001	sub
0010	and
0011	or
0100	xor
0101	left shift logical
0110	right shift logical
0111	not
1000	multiply
1001	divide
1010	Increment
1011	decrement
1100	Left shift arithmetic
1101	Right shift arithmetic
1110	Get B value (immediate)

Testing ALU:

To ensure the ALU operations are performed correctly, we will test every single operation at least once with rather random values. We will make sure to use a mix of positive and negative values to ensure all operations are performed successfully. For shifting, we will have 4 different tests, one for each type and make sure our operations are performed well.

Test Instruction	Input	Expected Output
Add	In_A = 6 In_B = -8 In_ALUCtrl = 4'b0000	Out_ALUResult = -2 Out_Zero = 0
Sub	In_A = 12288 In_B = 8 In_ALUCtrl = 4'b0001	Out_ALUResult = 12280 Out_Zero = 0
And	In_A = -1 In_B = -8 In_ALUCtrl = 4'b0010	Out_ALUResult = -8 Out_Zero = 0
Or	In_A = 6 In_B = -8 In_ALUCtrl = 4'b0011	Out_ALUResult = -2 Out_Zero = 0
Xor	In_A = 8074 In_B = 12200 In_ALUCtrl = 4'b0100	Out_ALUResult = 6 Out_Zero = 0
Left Shift logical	In_A = 3 In_B = 2 In_ALUCtrl = 4'b0101	Out_ALUResult = 12 Out_Zero = 0
Right shift logical	In_A = 48 In_B = 2 In_ALUCtrl = 4'b0110	Out_ALUResult = 12 Out_Zero = 0
Not operation	In_A = -8 In_B = 0 In_ALUCtrl = 4'b0111	Out_ALUResult = 7 Out_Zero = 0
Multiply	In_A = 10 In_B = -8 In_ALUCtrl = 4'b1000	Out_ALUResult = -80 Out_Zero = 0
Divide	In_A = 10 In_B = 2	Out_ALUResult = 5 Out_Zero = 0

	In_ALUCtrl = 4'b1001	
Increment	In_A = -8 In_B = 0 In_ALUCtrl = 4'b1010	Out_ALUResult = -7 Out_Zero = 0
Decrement	In_A = 2 In_B = 1 In_ALUCtrl = 4'b1011	Out_ALUResult = 1 Out_Zero = 0
Left Shift Arithmetic	In_A = 2 In_B = 2 In_ALUCtrl = 4'b1100	Out_ALUResult = 8 Out_Zero = 0
Right Shift Arithmetic	In_A = -32 In_B = 3 In_ALUCtrl = 4'b1101	Out_ALUResult = -4 Out_Zero = 0
Pass through	In_A = -32 In_B = 7 In_ALUCtrl = 4'b1110	Out_ALUResult = 7 Out_Zero = 0

Immediate Generator

Build immediate generator:

We are planning to build an unlocked immediate generator that takes in a reg instruction and outputs immediate. It will be built using a switch case for the opcode.

Testing immediate generator:

To ensure thorough testing of the immediate generator, at least one R type will be tested to ensure an immediate value of 0 is given. Then, for each instruction that uses an immediate for the type, we will test both a positive and negative immediate value and make sure the desired immediate value is returned. The input was a 16-bit instruction, and the output was a 16 bit immediate. Since no clock is used, there will simply be a short wait time (#10) between setting variables.

Instruction	Input (instr)	Output
Addi positive	16'b00000000100110100	3
Addi negative	16'b0010001111110100	-1
Lw positive	16'b0010000101000111	4

Lw negative	16'b0011001010100111	-6
Sw positive	16'b0001100100101000	2
Sw negative	16'b1001011011101000	-2
Beq positive	16'b1000000010001001	8
Beq negative	16'b1001111111001001	-4
Jal positive	16'b0010000001101100	6
Jal negative	16'b1011111101101100	-10
jalr	16'b0001001000000111	0
Lui positive	16'b1101000011011110	16'b00001101_00000000
Lui negative	16'b1110111010111110	16'b11101011_0000_0000
Lbi positive	16'b1100010010101111	16'b00000000_01001010
Lbi negative	16'b1101101011101111	16'b00000000_10101110
Si	16'b0011001101010101	5
Si	16'b0010000111110101	15

Memory

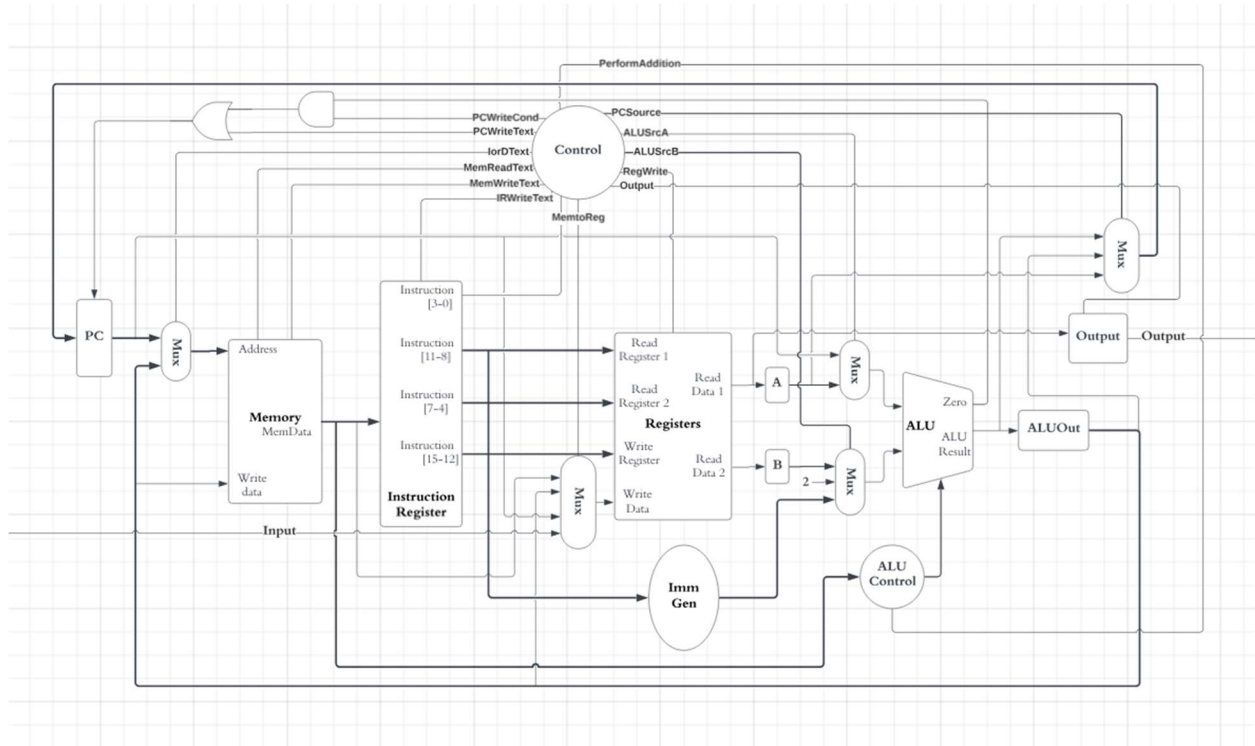
Build memory unit:

We are planning to build a clocked memory unit that takes an address as an input, and outputs an instruction. It will be built using a single port RAM template.

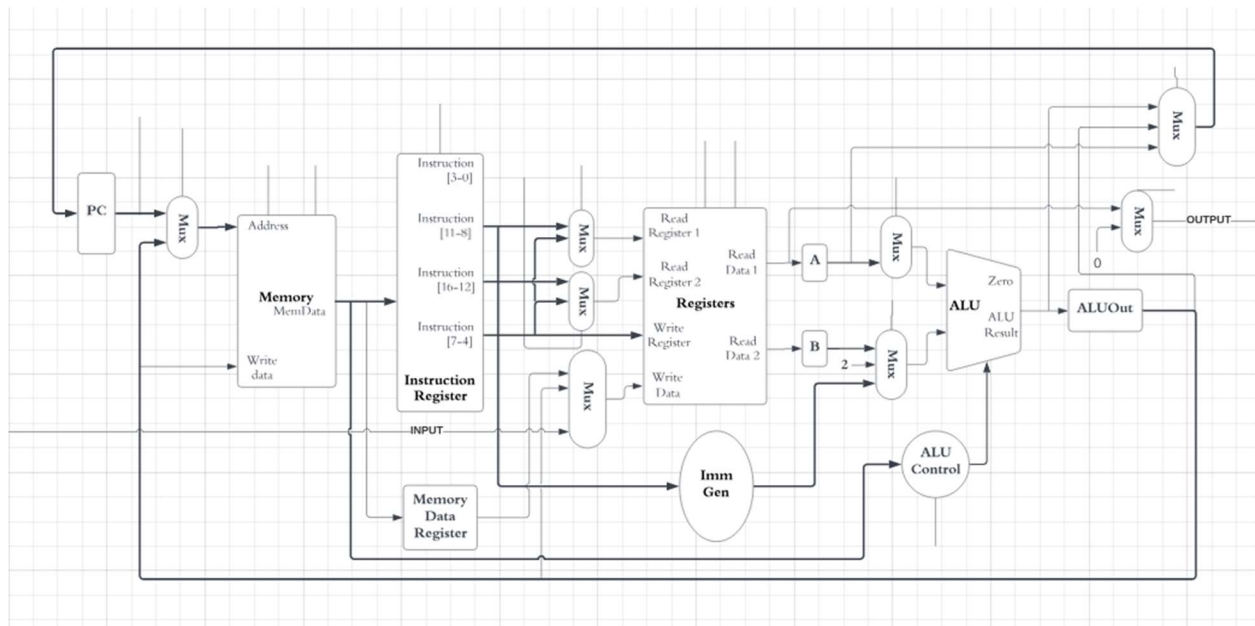
Testing memory unit:

Since the memory unit will only take the 4-bit opcode as the address input, we need to test each unique opcode. When the simulation runs, the input address should change, and the output value should change on the next rising edge. In this case, the input will be the 4-bit opcode and the output will be the desired instruction.

Datapath



Datapath without Control



To read and interpret this datapath, assume that all the connections are wires that can take in the appropriate number of bits. The datapath is set up and connected entirely with wires. Each element

is connected with the appropriate inputs and outputs to link together to create one connected datapath with the control.

Control

ALU Control

The ALU Control will make sure that each instruction opcode is then taken in and processed to do the correct ALU operation depending on the instruction. The immediate genie will also pass in the 2 unique shift bits to the ALU Controller which will be used only for shifts to determine the specific type of shift mentioned earlier.

ALU Control Testing

The ALU Control testing will test each instructions and make sure the correct output is given. It will really care about the Si input bits for the shifting and make sure the processing happens correctly.

Instruction	Input	Output
Add	In_Inst = 4'b0000 In_Si = 2'b00	Out_ALUCtrl = 4'b0000
Sub	In_Inst = 4'b0001 In_Si = 2'b00	Out_ALUCtrl = 4'b0001
And	In_Inst = 4'b0010 In_Si = 2'b00	Out_ALUCtrl = 4'b0010
Or	In_Inst = 4'b0011 In_Si = 2'b00	Out_ALUCtrl = 4'b0011
Lw	In_Inst = 4'b0111 In_Si = 2'b00	Out_ALUCtrl = 4'b0000
Sw	In_Inst = 4'b1000 In_Si = 2'b00	Out_ALUCtrl = 4'b0000
Beq	In_Inst = 4'b1001 In_Si = 2'b00	Out_ALUCtrl = 4'b0001
Jal	In_Inst = 4'b1100 In_Si = 2'b00	Out_ALUCtrl = 4'b0000
Lui	In_Inst = 4'b1110 In_Si = 2'b00	Out_ALUCtrl = 4'b1110
Lbi	In_Inst = 4'b1111	Out_ALUCtrl = 4'b0011

	In_Si = 2'b00	
SLL	In_Inst = 4'b0101 In_Si = 2'b00	Out_ALUCtrl = 4'b0101
SRL	In_Inst = 4'b0101 In_Si = 2'b01	Out_ALUCtrl = 4'b0110
SLA	In_Inst = 4'b0101 In_Si = 2'b10	Out_ALUCtrl = 4'b1100
SRA	In_Inst = 4'b0101 In_Si = 2'b11	Out_ALUCtrl = 4'b1101

Control

Description of Control Signals

lorD (Instruction or Data) – determines whether the fetch operation in memory is fetching an instruction or accessing data. When set to 1, it is signaled that the memory operation is related to accessing data. When set to 0, the memory operation will fetch an instruction.

MemRead – indicates whether a memory read operation should be executed during the current clock cycle. When set to 1, it signals that the processor needs to read data from memory. When set to 0, this indicates that the current operation does not need to access data from memory.

MemWrite – indicates whether a memory write operation should be executed during the current clock cycle. When set to 1, it signals that the processor needs to write data to memory. When set to 0, it indicated that the current operation does not need to write data to memory.

IRWrite (Instruction Register Write) – indicates whether the current instruction should be written to a register in the register file. When set to 1, the signal indicates that the processor should write the fetched instruction to the IR register. This makes the instruction available for subsequent processing stages. If the signal is set to 0, that means writing the fetched instruction to the IR register does not occur in this clock cycle.

RegWrite – indicates whether the result of an operation writes back to a register within the register file. When set to 1, this signals that the result of the current operation should be written to a specified register in the register file. When 0, RegWrite indicates that the current operation does not write data to a register.

PCSource – indicates which value will go into pc. When set to 0, the value will come from ALUResult. This allows the pc to increment by 2 each fetch state. When set to 1, the value will come from ALUOut. This allows the adding to pc using an immediate, which happens for jal and beq, and happens in the decode stage. When set to 2, the value will come from Rs1. This allows inputting from a register, and happens for the jalr instruction.

PCWriteCond – indicates whether or not this is a branch instruction. Works alongside the zero signal outputted in the ALU to input into an AND gate. If both are 1, then a 1 will be outputted and PC will write during that stage. If PCWriteCond is 1, then the instruction is a beq, recalling we only have one branch instruction.

PCWrite – indicates whether or not to jump. This works alongside the output of the AND gate described in PCWriteCond to feed into an OR gate. If either values are 1, pc will change. If PCWrite is 1, then the instruction is either a jal or a jalr, and is 0 otherwise.

MemtoReg – indicates which data will write into the write register. This is a 2-bit signal that determines whether the input data will come from the PC, the input wire, the memory, or ALUOut.

ALUSrcA – used to select a source for the first ALU input

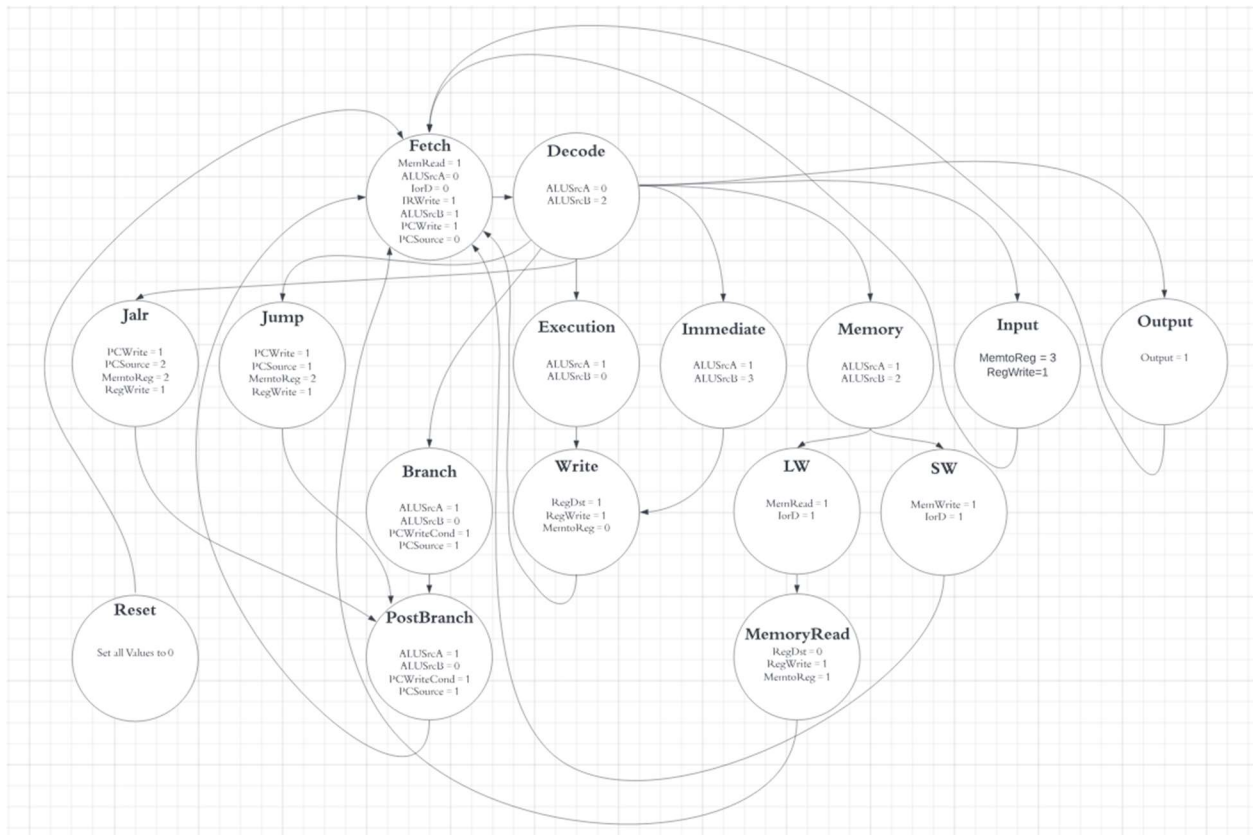
ALUSrcB – used to select a source for the second ALU input

ALUCtrl – used to select which operation the ALU should perform

Output – indicates whether or not to change the value of the output register. Value goes to one when during an Lout instruction.

Perform Addition – ensures that the ALUOut does addition in the decode and fetch stage.

State Diagram



Integration Plan

Unit Testing

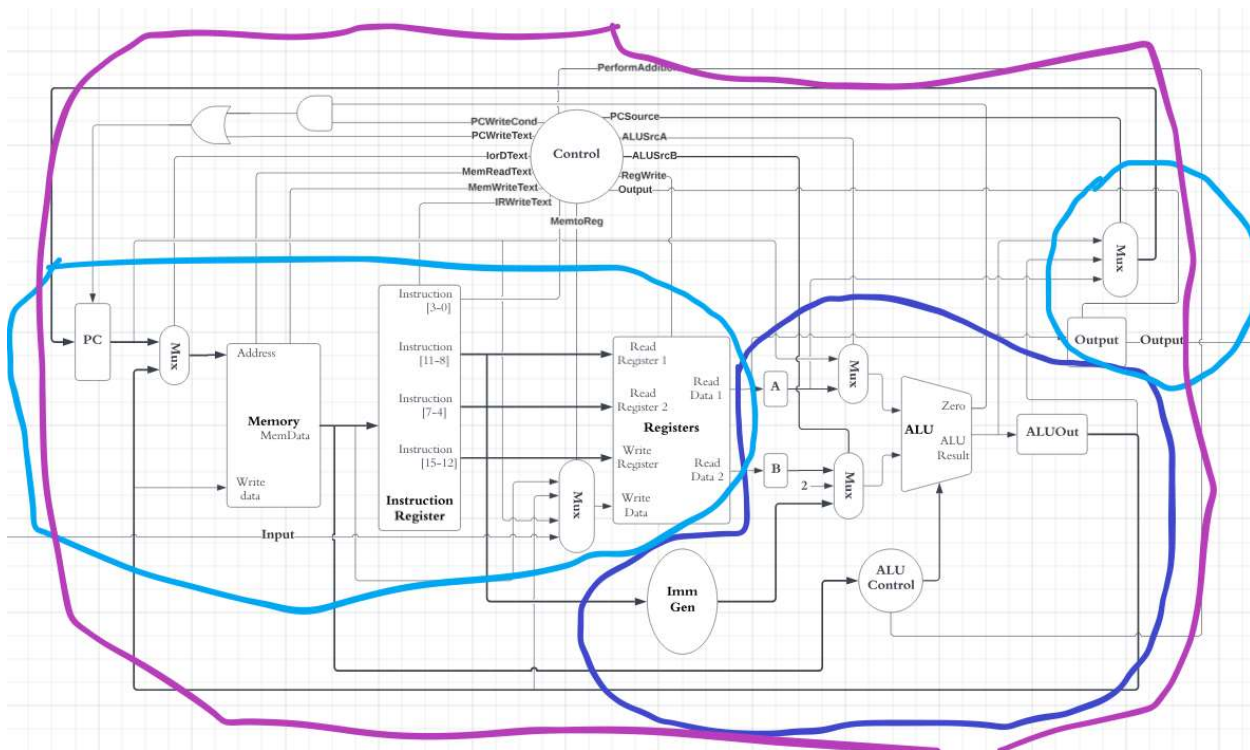
Begin by writing unit tests for each individual component including, registers, register file, memory, immediate generator, ALU, and Control. Make sure each component is working as expected in isolation.

Component Integration

Gradually integrate components in the following order, ensuring proper communication between each component and proper functionality over multiple components. Once finished, we will create a top-level file which will essentially serve as the main file for our processor, that when executed will run rel-prime.

Integration	File Name
PC, Memory, IR, Register File	Integration_Testing/PC_Memory_IR_RegisterFile_tb

ALU, ALUControl, ImmGenie, ALUOut	Integration_Testing/ALU_ALUControl_ImmGenie, ALUOut_tb	
Processor	Integration_Testing/Processor_tb	



ALU, ALUControl, ImmediateGenie Integration Test

We are going to connect these components together, with the two bit op from the immediate genie going into the ALU control along with a given 4 bit opcode. This will determine the ALU Ctrl signal which will be passed into the ALU. Values will be given for the value of A to the ALU, and sometimes register values will be given to the value of B for the ALU, or values from the immediate genie will be passed in. The operation will be performed and the final result from the ALUOut will be compared to our expected result.

Tests:

The tests for these will focus on the I and U types because immediates from the immediate genie are actually used and need to make sure they are being passed around correctly. At least one test for each different instruction will be used to make sure the correct operation is being performed for each one.

Instruction	Input	Output
add r3, r5, r7	Instr 16'b0011010101110000 In_A = -4 In_B = 8 ALUSrcB = 2'b00	Out_ALUResult = 4 Out_Zero = 0
addi sp, sp, -6	Instr 16'b0010001010100100 In_A = 14 In_B = 30 ALUSrcB = 2'b10	Out_ALUResult = 8 Out_Zero = 0
lw s1, 4 (sp)	Instr 16'b1001001001000111 In_A = 30 In_B = 14 ALUSrcB = 2'b10	Out_ALUResult = 34 Out_Zero = 0
lui r2, 112	Instr 16'b0011111100101110 In_A = 4 In_B = 2 ALUSrcB = 2'b10	Out_ALUResult = 16'b11110010_00000000 Out_Zero = 0
si t2, 00111111 (shifting a neg A 15 bits)	Instr 16'b1101001111110101 In_A = 16'b1100_0010_0001_0000 In_B = 16'b0010_0000_0101_0100 ALUSrcB = 2'b10	Out_ALUResult = -1 Out_Zero = 0
si t2, 00111111 (shifting a pos A 15 bits)	Instr 16'b1101001111110101 In_A = 16'b0100_0010_0001_0000 In_B = 16'b0010_0000_0101_0100 ALUSrcB = 2'b10	Out_ALUResult = 0 Out_Zero = 0

Handling I/O

There are two instructions dedicated to handling I/O.

- Lin will handle getting data into a destination register, rd. The data will be taken and placed in the given register when this instruction is called to load inputs.
- Lout will handle getting data out of the given register, rs1. The data will simply be taken from that register and returned to the user through a wire.

Unique Project Features

Key instructions

The main instructions that caused us to change our datapath design focused on si, shifting, and branching beq

- For shifting (si), we had all the different shifting ways – left, right, arithmetic, logical – in a single instruction. We used extra bits of the immediate to indicate which shifting method is being used. Therefore, we had to add an ALU control to take in these extra bits from the immediate and determine the opcode for the right shift. This was an extra design choice we did that is unique to our design.
- For branching (beq), we only had a single branch instruction br. We did this because we wanted to free up more opcodes, and instead we dedicated a register to simply always be the comparing register value for jumps. This made it so to perform one branch we had to do multiple instructions, but it simplified other elements of our datapath.

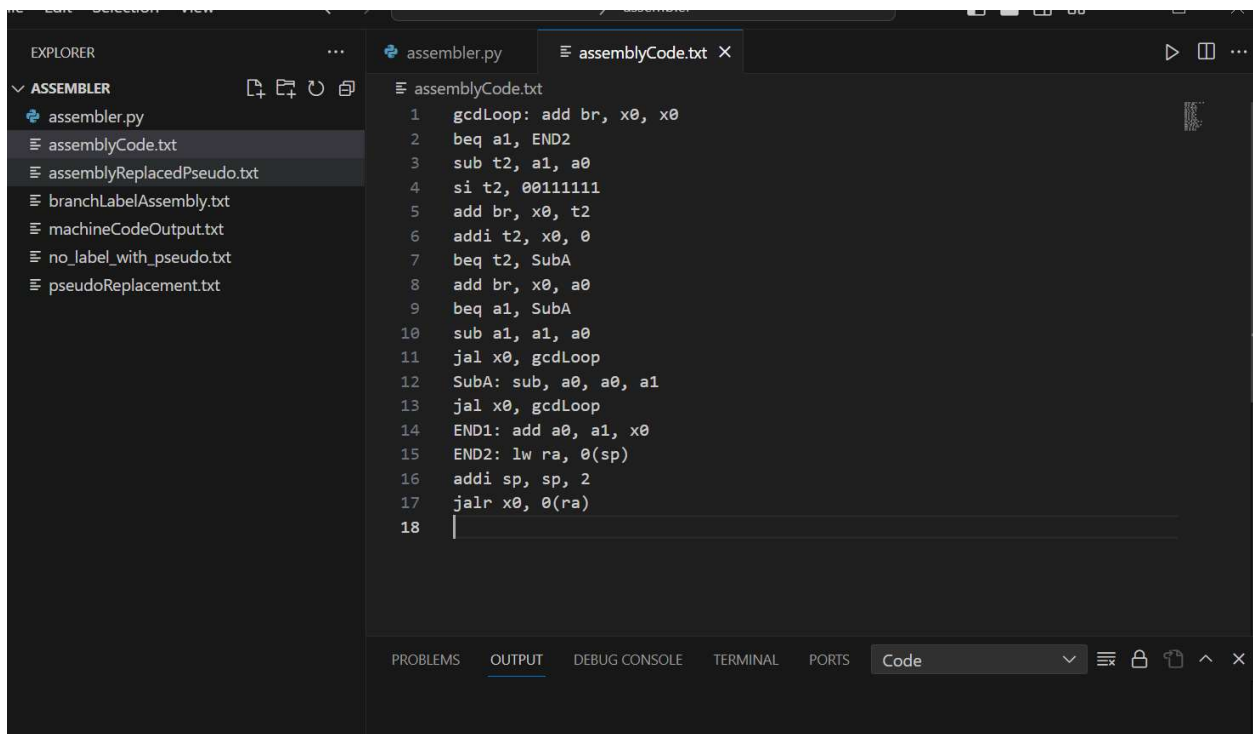
We had to modify our datapath to work around these instructions and focused on making sure our design would work properly.

Extra Features

Assembler

The assembly code must follow these certain rules for the machine code to be calculated correctly:

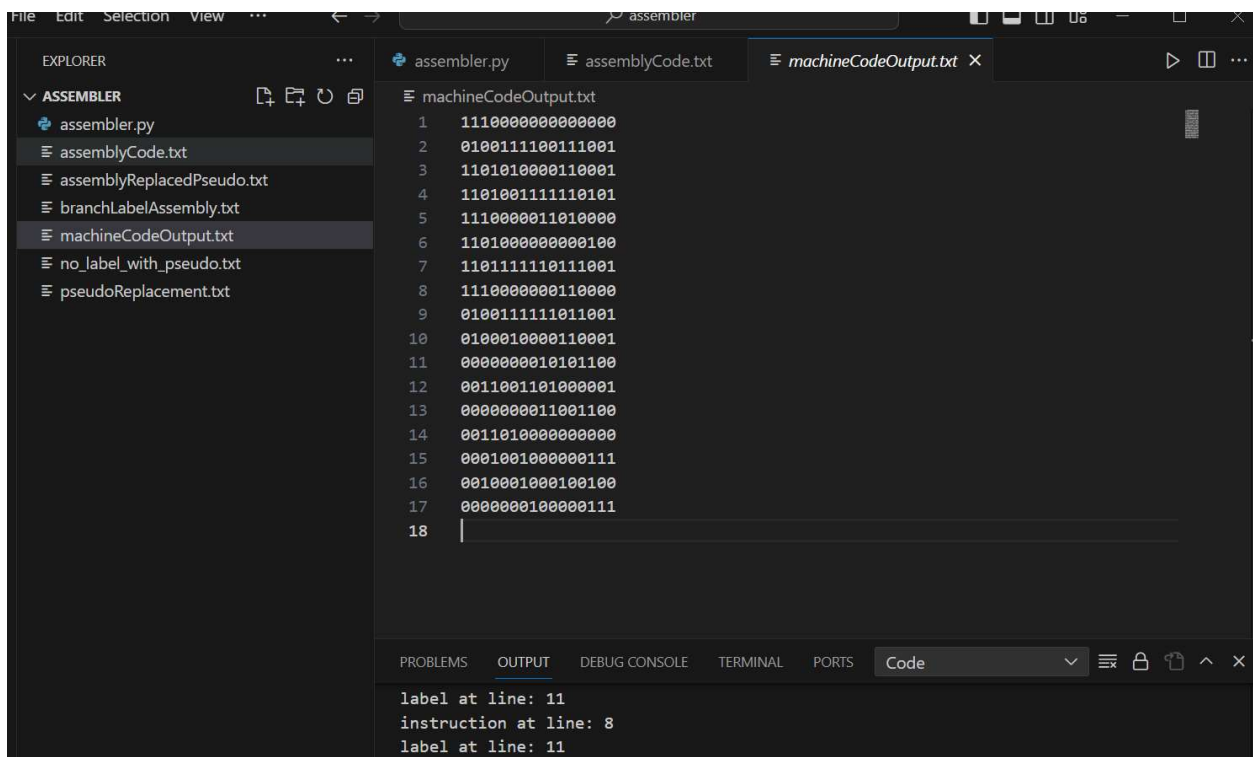
- Put your input code into **assemblyCode.txt**



The screenshot shows the Visual Studio Code editor with the 'ASSEMBLER' project open. The 'EXPLORER' sidebar on the left lists several files, with 'assemblyCode.txt' selected. The main editor window displays the contents of 'assemblyCode.txt', which contains assembly code for a gcd function. The code is as follows:

```
1 gcdLoop: add br, x0, x0
2 beq a1, END2
3 sub t2, a1, a0
4 si t2, 00111111
5 add br, x0, t2
6 addi t2, x0, 0
7 beq t2, SubA
8 add br, x0, a0
9 beq a1, SubA
10 sub a1, a1, a0
11 jal x0, gcdLoop
12 SubA: sub, a0, a0, a1
13 jal x0, gcdLoop
14 END1: add a0, a1, x0
15 END2: lw ra, 0(sp)
16 addi sp, sp, 2
17 jalr x0, 0(ra)
18
```

- The output machine code will end up in machineCodeOutput.txt



The screenshot shows the Visual Studio Code editor with the 'ASSEMBLER' project open. The 'EXPLORER' sidebar on the left lists several files, with 'machineCodeOutput.txt' selected. The main editor window displays the contents of 'machineCodeOutput.txt', which contains binary machine code. The code is as follows:

```
1 1110000000000000
2 0100111100111001
3 1101010000110001
4 1101001111110101
5 1110000011010000
6 1101000000001000
7 1101111110111001
8 1110000001100000
9 0100111111011001
10 0100010000110001
11 0000000010101100
12 0011001101000001
13 0000000011001100
14 0011010000000000
15 0001001000000111
16 0010001000100100
17 0000000100000111
18
```

Below the main editor window, the 'OUTPUT' panel shows the following text:

```
label at line: 11
instruction at line: 8
label at line: 11
```

- All immediates are given in decimal representation except for the immediate given to si, which is given in an 8 bit binary format

- There are no extra lines between code instructions, as each line is taken as a 2 bit address when calculating tags
- There are no extra lines in the file at the end of the instruction

Assembler Pseudoinstructions

The assembler can account for pseudoinstructions. To create a pseudoinstruction, in the `pseudoinstruction.txt`, add your pseudo instructions and parameters using `val1`, `val2`, ... Add a colon when the template for the instruction is complete. Then in the next lines, one instruction per line, put the instructions that make up the pseudoinstruction. Add an empty line after the final instruction. There are examples of pseudoinstructions already in the `pseudoinstructions` file.

The screenshot shows a Visual Studio Code editor with the following components:

- EXPLORER (Left Panel):** Shows the project structure under the 'ASSEMBLER' folder:
 - `assembler.py`
 - `assemblyCode.txt`
 - `assemblyReplacedPseudo.txt`
 - `branchLabelAssembly.txt`
 - `machineCodeOutput.txt`
 - `no_label_with_pseudo.txt`
 - `pseudoReplacement.txt` (Selected)
- EDITOR (Main Panel):** Displays the content of `pseudoReplacement.txt`:


```

1  rob val1, val2, val3:
2  and val3, val3, val3
3  add val1, val2, val3
4  sub val2, val3, val1
5  add val3, val1, val1
6
7  bob val1, val2, val3:
8  lui val1, val3
9  addi val2, val1, val3
10
11 ala val1, val2:
12 lbi val1, val2
13 lui val1, val2
14
15 vin val1, val2, val3, val4:
16 and val1, val2, val2
17 or val3, val3, val4
18
19 brian val1, val2:
20 beq val1, val2
21 jal val1, val2
      
```
- OUTPUT (Bottom Panel):** Shows the output of the assembler:


```

label at line: 11
instruction at line: 8
label at line: 11
instruction at line: 10
label at line: 0
instruction at line: 12
label at line: 0

[Done] exited with code=0 in 0.00s
      
```
- NOTIFICATION (Bottom Right):** A message box stating: "You have Docker installed on your system. Do you want to install the recommended extensions from Microsoft for it?" with buttons for "Install" and "Show Recommendations".

Assembler Tags/Branching:

The assembler can handle having tags for branching and jumping added in the same line as pseudoinstructions and regular instructions. To add a tag into the data, use the format:

NAME: instruction

Having the name of the tag match in all the places it is being used is important. For placing the tag on an instruction, make sure to use a colon and space to separate the tag from the instruction. So for example:

LOOP: add r2, r3, r2

beq r2, LOOP

RelPrime Testing Data

Benchmark Data

Running the default value: 5040

Number of bytes: **110 bytes**

Number of instructions: **112,000 instructions**

Number of cycles: **460,000 cycles**

Average cycles per instruction: **4.09 Average CPI**

Cycle Time: **50.1 ns**

Total Execution Time: **23 Milliseconds**

Logical gates and registers used: **446 Total Registers, 16,384 memory bits**

Our results are standard for a load-store multicycle architecture. On average, most of our instructions took 4 cycles, with the exception being lw. Our final relPrime code ended up being 55 instructions, and with how many loops and branches we used, it took a total of 112,000 instructions. Overall, the results match our expectations and the computation of relprime successfully occurred.

Conclusion

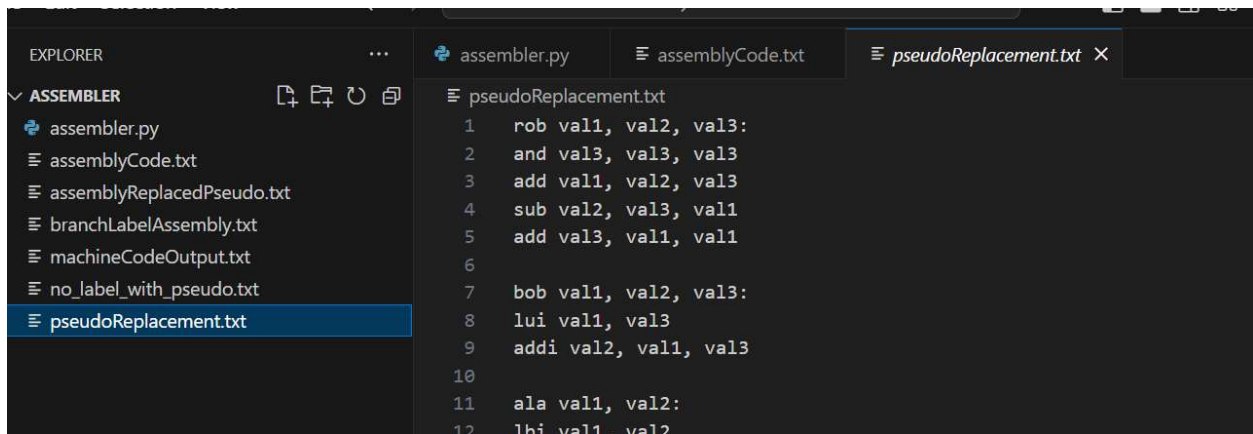
Overall, we created a 16 bit, load-store, multicycle architecture for our processor. We had 16 opcodes to perform a variety of basic instructions for a processor. Our unique design had a single

shift instruction that used extra immediate space to specify the exact type of shift (arithmetic, logical, left, right). We also had a single branch instruction `beq` and a unique register dedicated to being used to comparing for branches, which was `br`. We implemented the rest of the branch variations through pseudoinstructions. Our processor successfully ran the `relPrime` code and generated the expected value from such tests. We also created an assembler with our project that could handle the generation of pseudoinstructions given the template in a file for our instruction set.

Appendix

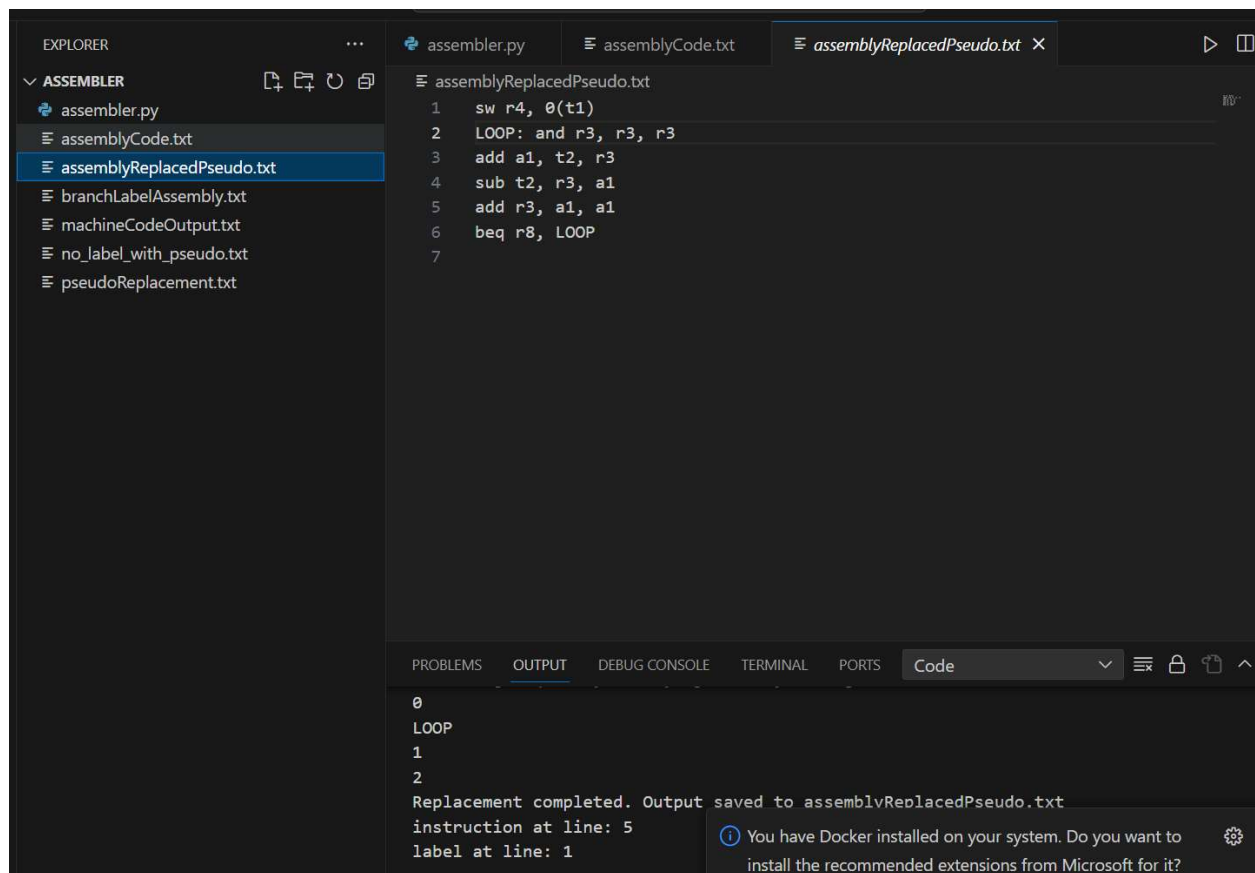
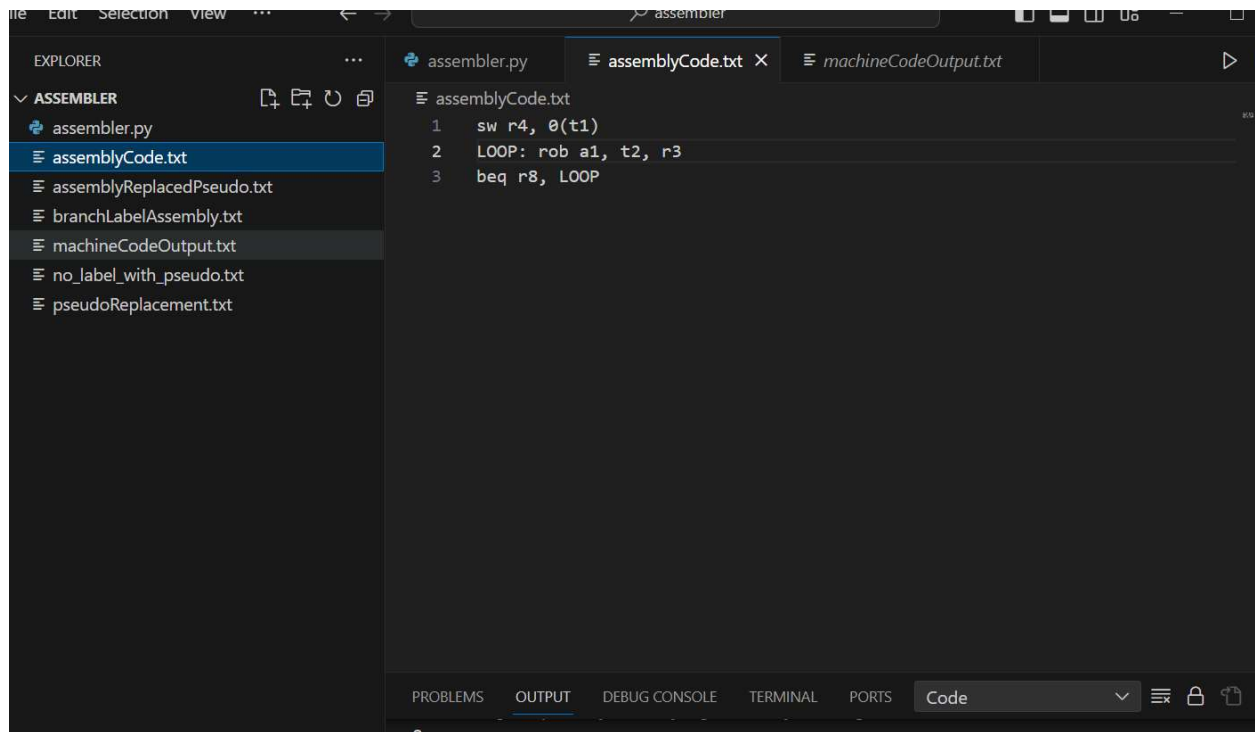
Assembler Tag + Pseudoinstruction example

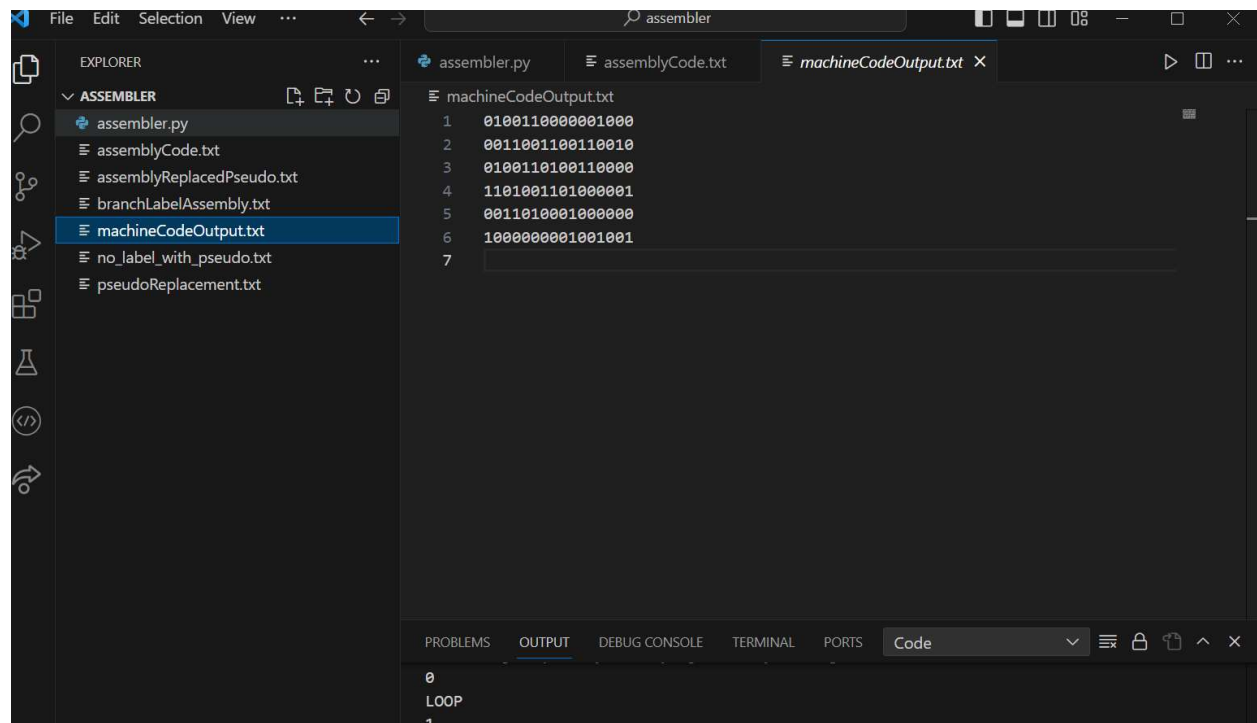
Using the assembler with a pseudo instruction and branch tag, showing an intermediate step of the pseudo replaced instruction



The screenshot shows a code editor with a dark theme. On the left, the 'EXPLORER' sidebar lists files under the 'ASSEMBLER' folder: `assembler.py`, `assemblyCode.txt`, `assemblyReplacedPseudo.txt`, `branchLabelAssembly.txt`, `machineCodeOutput.txt`, `no_label_with_pseudo.txt`, and `pseudoReplacement.txt` (which is selected). The main editor area shows the contents of `pseudoReplacement.txt` with the following assembly code:

```
1  rob val1, val2, val3:
2  and val3, val3, val3
3  add val1, val2, val3
4  sub val2, val3, val1
5  add val3, val1, val1
6
7  bob val1, val2, val3:
8  lui val1, val3
9  addi val2, val1, val3
10
11 ala val1, val2:
12 lhi val1, val2
```





Reference Sheet

Base Integer Instructions

Inst	Name	FMT	Opcode	Description	Note
add	Add	R	0000	$R[rd] = R[rs1] + R[rs2]$	
sub	Subtract	R	0001	$R[rd] = R[rs1] - R[rs2]$	
and	And	R	0010	$R[rd] = R[rs1] \& R[rs2]$	
or	Inclusive Or	R	0011	$R[rd] = R[rs1] R[rs2]$	
addi	Add Immediate	I	0100	$R[rd] = R[rs1] + SE(imm)$	
lw	Load Word	I	0111	$R[rd] = M[R[rs1] + SE(imm)]$	
sw	Store Word	I	1000	$M[R[rs1] + SE(imm)] = R[rd]$	
beq	Branch ==	U	1001	if $(rs1 == BR)$ $PC += SE(imm) \ll 1$	
jal	Jump And Link	U	1100	$R[rd] = PC + 2$ $PC += SE(imm) \ll 1$	PC relative
jalr	Jump And Link Register	I	1101	$R[rd] = PC + 2$ $PC = R[rs1]$	register relative
lui	Load Upper Immediate	U	1110	$R[rd] = SE(imm) \ll 8$	
lbi	Load Bottom Immediate	U	1111	$R[rd] = R[rd] + imm$	
si	Shift Immediate	U	0101	if $(imm[4] == 0)$ $R[rd] = R[rd] \ll imm[3:0]$ else if $(imm[4] == 1)$ $R[rd] = R[rd] \gg imm[3:0]$	
lin	Load Input	U	0110	$R[rd] = INPUT$	
lout	Load Output	I	1010	$OUTPUT = R[rd]$	

R = Register file access, SE = Sign extend

Core Instruction Formats

15	12	11	8	7	4	3	0	
rd		rs1		rs2		opcode		Ralts-type (R-type)
rd		rs1		imm		opcode		Ivysaur-type (I-type)
rd			imm			opcode		Umbreon-type (U-type)

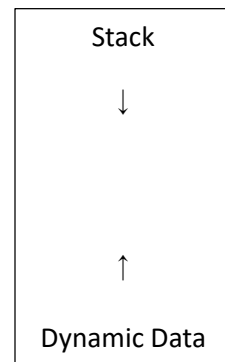
Registers

Register	Name	Description	Saver
x0	zero	Zero constant	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3-x5	a0-a2	Fn args/return values	Caller
x6-x7	a3-a4	Fn args	Caller
x8-x10	s0-s2	Saved registers	Callee
x11-x13	t0-t2	Temporaries	Caller
x14	br	Branch compare register	Caller
x15	at	Assembler Temporary	Caller

Memory Allocation

SP → 0x03FF

PC → 0x0000



Final RelPrime Code and Machine Code

Assembly Code:

```
START: Lin a0
add s0, a0, x0
jal ra, relPrime
Lout a0
InfLoop: Lin a1
add br, s0, x0
beq a1, InfLoop
jal x0, START
relPrime: lui t0, -1
lbi t0, -8
add sp, sp, t0
sw ra, 0(sp)
sw s0, 2(sp)
sw s1, 4(sp)
sw s2, 6(sp)
add s0, a0, x0
addi s1, x0, 2
addi s2, x0, 1
relLoop: add a0, s0, x0
add a1, s1, x0
jal ra, gcd
add br, s2, x0
beq a0, END
addi s1, s1, 1
jal x0, relLoop
END: add a0, s1, x0
lw ra, 0(sp)
lw s0, 2(sp)
lw s1, 4(sp)
lw s2, 6(sp)
lui t0, 0
lbi t0, 8
add sp, sp, t0
jalr x0, 0(ra)
gcd: addi sp, sp, -2
sw ra, 0(sp)
add br, x0, x0
beq a0, END1
gcdLoop: add br, x0, x0
beq a1, END2
sub t2, a1, a0
si t2, 00111111
add br, x0, t2
```

```
addi t2, x0, 0
beq t2, SubA
add br, x0, a0
beq a1, SubA
sub, a0, a0, a1
jal x0, gcdLoop
SubA: sub a1, a1, a0
jal x0, gcdLoop
END1: add a0, a1, x0
END2: lw ra, 0(sp)
addi sp, sp, 2
jalr x0, 0(ra)
```

Machine Code:

```
0011000000000110
1000001100000000
0001000010101100
0000001100001010
0100000000000110
1110100000000000
0100111110101001
0000111100001100
1011111111111110
1011111110001111
0010001010110000
0001001000001000
1000001000101000
1001001001001000
1010001001101000
1000001100000000
1001000000100100
1010000000010100
0011100000000000
0100100100000000
0001000110101100
1110101000000000
0011000001001001
1001100100010100
0000111100101100
0011100100000000
0001001000000111
1000001000100111
1001001001000111
1010001001100111
1011000000001110
1011000010001111
0010001010110000
```

0000000100001101
0010001011100100
0001001000001000
1110000000000000
0011000110101001
1110000000000000
0100000110001001
1101010000110001
1101001111110101
1110000011010000
1101000000000100
1101000010001001
1110000000110000
0100000001001001
0011001101000001
0000111010101100
0100010000110001
0000111001101100
0011010000000000
0001001000000111
0010001000100100
0000000100001101