

Design Document

Team 1, Section 3

Ethan Bennett, Medhansh Khattar, Harrison Carpenter

[Google Docs link](#)

Design Philosophy

Our design is aimed at reducing the total number of instruction types to implement the relatively prime functionality. By doing so, we plan to limit the number of bits for funct to 1, and opcode to 3 bits. This enables us to allow the use of multiple registers for load-store architecture, and in continuation, branch to farther addresses in 16-bit instructions. Once we decided on 16 bit core instruction formats, we moved onto developing RTLs for each and planning all components required for our datapath in the future.

Performance

Performance will be measured using execution time of the relPrime function. As such, we have opted to implement a multicycle architecture, as that will decrease the execution time of the program.

Registers

Register	Name	Description
x0	zero	Zero constant.
x1	ra	Return address.
x2	sp	Stack pointer.
x3	IN	User input register.
x4	OUT	Output register.
x5	t0	Temporaries.
x6	t1	
x7	t2	
x8	s0, fp	Callee-saved registers.
x9	s1	
x10	a0	Function arguments and return values.
x11	a1	
x12	a2	
x13	a3	
x14	t3	Another temporary.
x15	at	Assembler temporary.

Instruction Types

A-Type (Arithmetic)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rs2				rs1				rd				funct	opcode		

Allows for two arithmetic operations (addition and subtraction) between registers.

Instruction	Opcode	Funct	Example	Meaning	Description
add	000	0	add x5, x6, x7	x5 = x6 + x7	R[rd] = R[rs1] + R[rs2]
sub	000	1	sub x5, x6, x7	x5 = x6 - x7	R[rd] = R[rs1] - R[rs2]

I-Type (Immediate)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Imm[3:0]				rs1				rd				funct	opcode		

Allows for two immediate operations (adding a register value with an immediate, and loading a value to a register from memory).

Instruction	Opcode	Funct	Example	Meaning	Description
addi	001	0	addi x5, x6, 7	x5 = x6 + 7	R[rd] = R[rs1] + SE(imm)
lw	001	1	lw x5, 0(x6)	Load a value from memory address x6 with an offset of 0 bytes into x5.	R[rd] = M[R[rs1]]+SE(imm)

S-Type (Store)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

Imm[8 6:4]	rs1	Imm[3:0 7]	opcode
------------	-----	------------	--------

Allows storing of a register value to memory.

Instruction	Opcode	Example	Meaning	Description
sw	010	sw x5, 4(x6)	Store the value from x5 to memory address x6, offset by 4 bytes.	$M[R[rs1+SE(imm)]=R[rs2]$

B-Type (Branch)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rs2				rs1				Imm[3:0]				funct	opcode		

Allows for two types of branches, being branch equal (beq) and branch greater than (bgt). Branch not equal and branch inclusive inequality is implemented by branching over unconditional jumps with their opposites. Note that this immediate is in terms of **bytes**.

Instruction	Opcode	Funct	Example	Meaning	Description
beq	011	0	beq x5, x6, -8	If x5 and x6 are equal, go back 8 instructions.	if(rs1 == rs2) PC += SE(imm)<<1
bgt	011	1	bgt x5, x6, 4	If x5 is greater than x6, go forward 4 instructions.	if(rs1 > rs2) PC += SE(imm)<<1

JAL-Type (Jump-And-Link)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Imm[7:0]								rd				Imm[8]	opcode		

Allows for unconditional jumps, with a larger range than a branch instruction. Also updates the return address. Note that this immediate is in terms of **bytes**.

Instruction	Opcode	Funct	Example	Meaning	Description
jal	100	0	jal ra, relPrime	Jump to relPrime and set return address.	$R[rd] = PC + 2$ $PC += SE(imm) \ll 1$

Memory Map

0x0000 0x00FF	-	Stack
0x0100 - 0x7FFF		Text
0x8000 0xFFFF	-	Data

Addressing Modes

This processor will use both Register and Immediate addressing modes. Immediate bits are not used in A-Type instructions, as those use register addressing. In I-Types, use the given immediate to either add to a register value, or offset a memory address. In S-Types, they are used to offset memory addresses. They are also rearranged to enable symmetry with other instruction types. In B-Types and JAL-Types, they are used to offset the PC (PC-Relative). This operation is PC relative, since moving the instruction around will still result in the same number of instructions jumped from that position.

Procedure Calling Conventions

Caller	Callee
Preserve return address across calls	Use given return address
Preserve temporary registers across calls	Freely use temporary registers
Freely use save registers across procedures	Preserve save registers, and restore before returning to caller
Preserve essential argument registers before calling	Use argument registers given

Procedure calling convention example

```
callAddTwo: addi sp, sp, -2 #offset stack by 2 bytes
            sw ra, 0(sp) #save ra to stack
            jal ra, addTwo #JAL to addTwo.
                                #Don't bother saving a0, we are done with
                                #it after this call anyway
            lw ra, 0(sp) #restore stack
            Addi sp, sp, 2 #restore stack
            jal zero, 0(ra) #JAL back to caller
addTwo:     addi a0, 2 #add a0 and 2
            jal zero, 0(ra) #JAL back to caller
```

Machine Language Translation of relPrime

Address	Assembly	Machine Code	Comment
0x0000	relPrime: addi sp, sp, -4	1100 0010 0010 0001	# Allocate stack space for local variables
0x0002	sw ra, 2(sp)	0010 0001 0000 1010	# Save return address
0x0004	sw a0, 0(sp)	0000 0001 0101 0010	# Save argument 'n'
0x0006	addi a1, a1, 2	0010 1011 1011 0001	# m = 2
0x0008	relPrime_loop: addi a2, a0, 0	0000 1100 1010 0001	# Copy 'n' to a2 for gcd function
0x000A	addi a3, a1, 0	0000 1101 1011 0001	# Copy 'm' to a3 for gcd function
0x000C	jal ra, gcd	0000 0101 0000 1100	# Call gcd(n, m)
0x000E	beq a0, zero, +4	1010 0000 0010 0011	# if gcd(n, m) == 1, don't update 'm'
0x0010	jal zero, update_m	0000 0001 1000 0100	# update 'm'
0x0012	addi a1, a1, 1	0001 1011 1011 0001	# m = m + 1
0x0014	jal zero, relPrime_loop	1111 1101 0000 0100	# Loop back
0x0016	update_m: addi a0, a1, 0	0000 1011 1010 0001	# Prepare return value
0x0018	lw ra, 2(sp)	0100 0010 0001 1001	# Restore return address

0x001A	lw a0, 0(sp)	0000 0010 1010 1001	# Restore 'n'
0x001C	addi sp, sp, 4	0100 0010 0010 0001	# Deallocate stack space
0x001E	jal zero, 0(ra)	XXXX XXXX X000 0100	# Return
0x0020	gcd: addi sp, sp, -6	1010 0010 0010 0001	# Allocate stack space for local
0x0022	sw ra, 4(sp)	0000 0001 0100 0010	# Save return address
0x0024	sw a0, 2(sp)	0000 1010 0010 0010	# Save 'a'
0x0026	sw a1, 0(sp)	0000 1011 0000 0010	# Save 'b'
0x0028	gcd_loop: beq a0, zero, 5	0000 1010 0101 0011	# if a == 0, jump to the jal 5 instructions down
0x002A	beq a1, zero, update_a	0000 1011 0101 1011	# if b == 0, update 'a'
0x002C	bgt a1, a0, subtract_b	1010 1011 0111 1011	# if a > b, a = a - b
0x002E	sub a0, a0, a1	1011 1010 1010 1000	# subtract a1 from a0, store in a0
0x0030	jal zero, gcd_loop	1111 1101 1000 0100	# loop
0x0032	jal zero, end_gcd	0000 0011 1000 0100	# jump to end_gcd
0x0034	update_a: addi a0, a1, 0	0000 1011 1010 0001	# a = b
0x0036	lw a1, 0(sp)	0000 0010 1011 1001	# Restore 'b'
0x0038	jal zero, gcd_loop	1111 1100 0000 0100	# loop

0x003A	subtract_b: sub a1, a1, a0	1010 1011 1011 1000	# else, b = b - a
0x003C	jal zero, gcd_loop	1111 1011 0000 0100	# loop
0x003E	end_gcd: lw ra, 4(sp)	0100 0010 0001 1001	# restore stack
0x0040	lw a0, 2(sp)	0010 0010 0001 1001	# restore stack
0x0042	lw a1, 0(sp)	0000 0010 0001 1001	# restore stack
0x0044	addi sp, sp, 6	0110 0010 0010 0001	# restore stack
0x0046	jal zero, 0(ra)	XXXX XXXX X000 0100	# return to caller

Single-Cycle RTL

add

```
newPC = PC + 2
PC = newPC
inst = InstMem[PC]
a = Reg[inst[11:8]]
b = Reg[inst[15:12]]
result = a + b
Reg[inst[7:4]] = result
```

addi

```
newPC = PC + 2
PC = newPC
inst = InstMem[PC]
a = Reg[inst[11:8]]
imm = SE(inst[15:12])
result = a + imm
Reg[inst[7:4]] = result
```

lw

```
newPC = PC + 2
PC = newPC
inst = InstMem[PC]
a = Reg[inst[11:8]]
imm = SE(inst[15:12])
result = a + imm
memOut = Mem[result]
Reg[inst[7:4]] = memOut
```

sw

```
newPC = PC + 2
PC = newPC
inst = InstMem[PC]
a = Reg[inst[11:8]]
b = Reg[inst[15:12]]
imm = SE(inst[7:4])
result = a + imm
Mem[result] = b
```

beq

```
newPC = PC + 2
PC = newPC
inst = InstMem[PC]
a = Reg[inst[11:8]]
b = Reg[inst[15:12]]
imm = SE(inst[7:4]) << 1
target = PC + imm
if (a == b) PC = target
```

bgt

```
newPC = PC + 2
PC = newPC
inst = Mem[PC]
a = Reg[inst[11:8]]
b = Reg[inst[15:12]]
imm = SE(inst[7:4]) << 1
target = PC + imm
if (a > b) PC = target
```

sub
<pre> newPC = PC + 2 PC = newPC inst = InstMem[PC] a = Reg[inst[11:8]] b = Reg[inst[15:12]] result = a - b Reg[inst[7:4]] = result </pre>

jal
<pre> newPC = PC + 2 PC = newPC inst = Mem[PC] imm = SE(inst[15:8]) << 1 PC = PC + imm </pre>

Generic Component Specifications (Single Cycle)

Component	Inputs	Outputs	Behavior	RTL Symbols
Register	newPC [15:0]	inst_address [15:0]	Takes a 16 bit value on the rising clock edge, and outputs that same value.	Single+ Multicycle: PC Multicycle exclusive: IR, MDR, A, B, ALUOut
Register File	rs1[3:0] rs2[3:0] rd[3:0] reg_data[15:0] reg_write[0:0]	reg_a[3:0] reg_b[3:0]	Takes 3 register addresses and can read rs1 and rs2 into reg_a and reg_b, as well as write a 16 bit value to rd. Also has a 1 bit input that enables writing to registers.	Reg
Instruction Memory	inst_address [15:0]	inst[15:0]	Takes a 16 bit instruction address and outputs the instruction	InstMem

			at that location in memory.	
Data Memory	data_address [15:0] data_data* [15:0] data_write [0:0]	data_out[15:0]	Takes a 16 bit memory address and outputs the value at it, and can write 16 bits of data to that address if data_write is enabled.	DataMem
Immediate Generator	inst[15:0]	imm[15:0]	Reads a 16 bit instruction and constructs a corresponding 9 bit immediate then sign extends it to 16 bits.	ImmGen, SE
ALU	operand_a [15:0] operand_b [15:0] aluop[0:0]	aluout[15:0] aluzero[0:0]	Performs an arithmetic operation between its two 16 bit operands, and outputs the result as a 16 bit signed integer.	+, -
PCAdder	PC[15:0] offset[15:0]	newPC[15:0]	Adds 2 OR an immediate offset to the program counter to increment to the next instruction.	PCInc

*change this later; it sounds stupid

Generic Component Specifications (Multi Cycle)

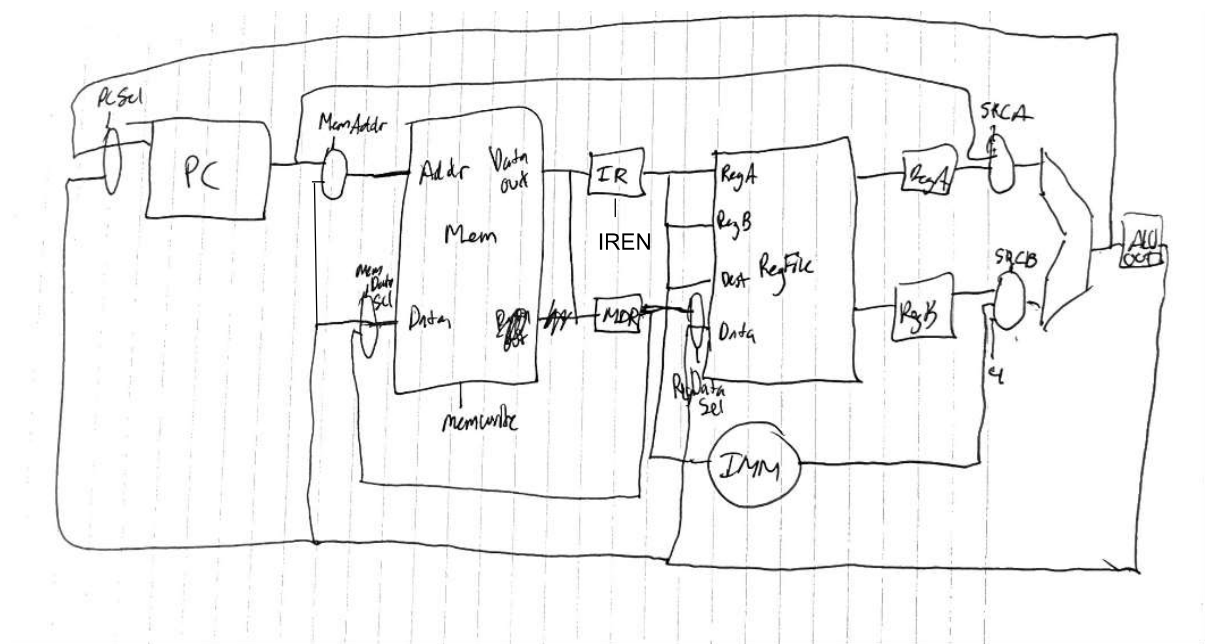
Component	Inputs	Outputs	Behavior	RTL Symbols
Register	newPC [15:0]	inst_address [15:0]	Takes a 16 bit value on the rising clock edge, and outputs that same value.	PC, IR, MDR, A, B, ALUOut
Register File	rs1[3:0] rs2[3:0] rd[3:0] reg_data[15:0] reg_write[0:0]	reg_a[3:0] reg_b[3:0]	Takes 3 register addresses and can read rs1 and rs2 into reg_a and reg_b, as well as write a 16 bit value to rd. Also has a 1 bit input that enables writing to registers.	Reg
Immediate Generator	inst[15:0]	imm[15:0]	Reads a 16 bit instruction and constructs a corresponding 9 bit immediate then sign extends it to 16 bits.	ImmGen, SE
ALU	operand_a [15:0] operand_b [15:0] aluop[0:0]	aluout[15:0]	Performs an arithmetic operation between its two 16 bit operands, and outputs the result as a 16 bit signed integer.	+, -
Memory	mem_address[15:0] mem_data[15:0] mem_write[0:0]	mem_out[15:0]	Takes a 16 bit memory address and outputs the data at that location in memory. Can	Mem

			also write 16 bits of data to that address if mem_write is enabled.	
--	--	--	--	--

Multi-Cycle RTL

A-Type	I-Type	S-Type	B-Type	JAL-Type
IR <= Mem[PC] PC <= PC + 2				
A <= Reg[IR[11:8]] B <= Reg[IR[15:12]] ALUOut <= PC + 2 + SE(Immediate*2)<<1				
ALUOut <= A op B	ALUOut <= A + SE(Immediate*2)	PC <= ALUZero?A:B		
Reg[IR[7:4]] <= ALUOut	MDF <= Mem[ALUOut] Mem[ALUOut]			
	Reg[IR[7:4]] <= MDF			

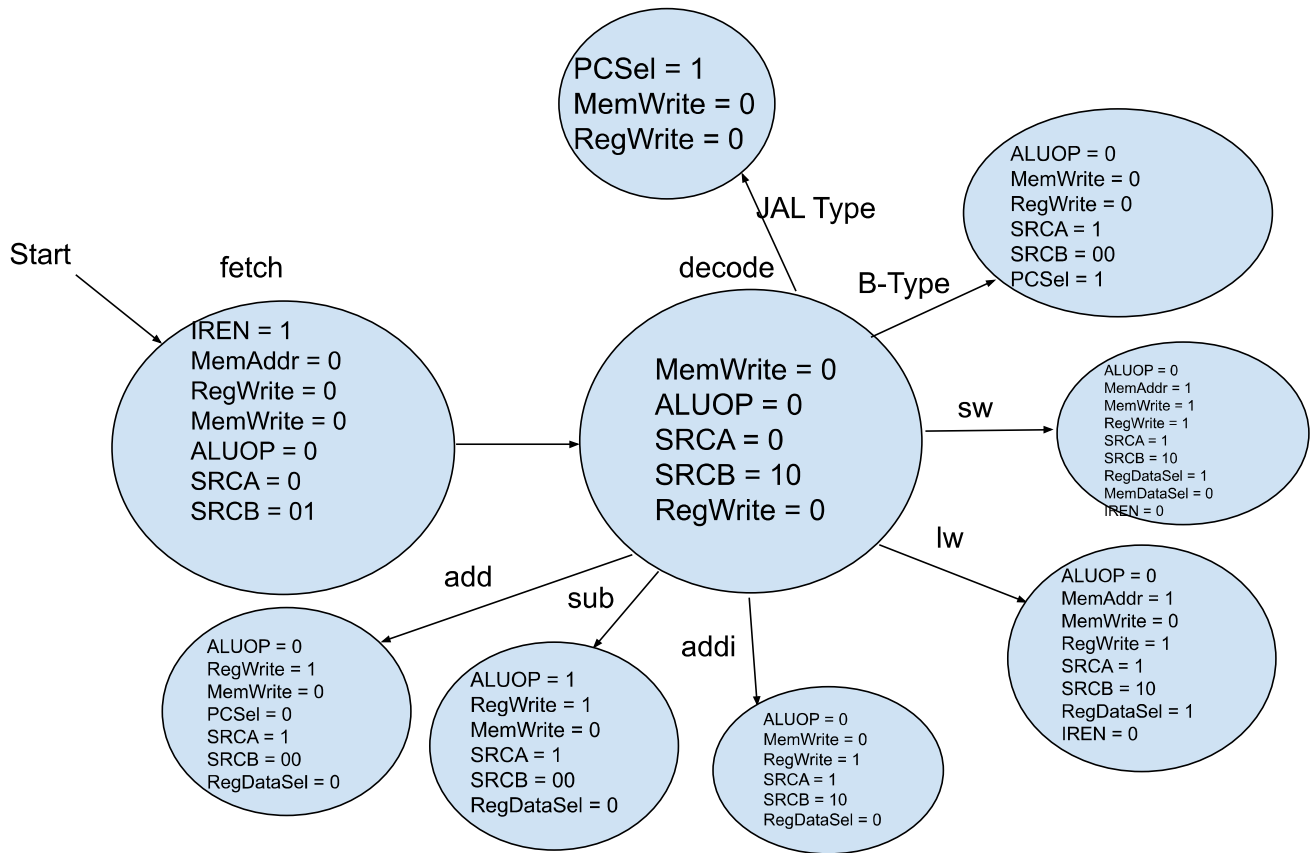
Prospective Datapath



Control Signal Descriptions

PCSel	1 bit	Controls if the PC is incremented by 4, or by an immediate offset.
MemAddr	1 bit	Controls source memory address (PC or ALUOUT)
MemDataSel	1 bit	Controls memory data input (MDR or ALUOUT)
RegDataSel	1 bit	Controls register data input (MDR or ALUOUT)
MemWrite	1 bit	Enables write to memory
RegWrite	1 bit	Enables write to register file
SRCA	1 bit	Chooses ALU Source A
SRCB	2 bits	Chooses ALU Source B
ALUOP	1 bit	Chooses ALU Operation
IREN	1 bit	Enables writing to IR register

Control Signal State Diagrams



Test Checklist:

Component	Test	Result	Description
ALU	5 + 5	10	Positive and Positive
	5 + -4	1	Positive and negative, nonzero
	-5 + -5	-10	Negative and Negative
	5 + -5	0	Positive and Negative, zero
	5 - 4	1	Positive and positive, nonzero
	5 - 5	0	Positive and Positive, zero
	5 - -5	10	Positive and negative
	-5 - -4	-1	Negative and Negative, nonzero
	-5 - -5	0	Negative and Negative, zero
PC	PC_IN = 1	PC_OUT = 1	Test writing to PC.
Memory	Memory Address = 299 Data = 299 MemWrite = 1	Memory Out = 299	Write to memory, and read from that address
	Memory Address 299 Data = 1 MemWrite = 0	Memory Out = 299	Read from address 299
	Memory Address 299 Data = 1 MemWrite = 1	Memory Out = 1	Overwrite address 299

Integration Test Plan:

We will test in the following combinations: PC & ALU, PC & MEM & REG, REG & ALU, ALU & MEM. Each set of tests will check both common and edge cases. Specifically, for PC & ALU, we will test incrementing the current value of the PC. For PC & MEM & REG, we will test fetching an instruction from memory and decoding that instruction. For ALU and MEMORY, we will test writing to and reading from a memory address (not from the PC).

Python Scripts for testing Single RTLs (For Code refer to Appendix.1)

The Python scripts developed for RTL module simulation encompass all operations a processor needs to perform. Here's a closer look at how these scripts function:

Environment Initialization:

- Instruction Memory (InstMem): A list representing the instruction memory, sized to 1024 elements, stores the encoded instructions.
- Registers (Reg): A list of 16 elements represents the general-purpose registers in the system.
- Data Memory (Mem): Similar to InstMem, this 1024-element list serves as the data memory for the system.
- Program Counter (PC): Initialized to 0, PC keeps track of the current instruction being executed.

Development of Helper Functions:

- Reset Function (reset()): This function is pivotal for reinitializing the simulation environment. It resets the content of registers and memory arrays to zero and sets the Program Counter (PC) back to its initial state. This is crucial for ensuring that each test starts with a clean slate, thereby eliminating any residual effects from previous tests.
- Sign-Extension Function (SE(value)): A critical component in handling immediate values, SE takes an 8-bit value and extends it to 16 bits. This is especially important for branching and immediate arithmetic operations, where the immediate values need to align correctly with the register sizes.

Core Instruction Functions:

- `addi()`: Implements an 'add immediate' operation, fetching the instruction from `InstMem`, extracting operands, and performing the addition.
- `add()`, `sub()`: Similar to `addi`, but for register-to-register addition and subtraction.
- `sw()`, `lw()`: Store and load operations, handling data transfer between registers and memory.
- `bgt()`, `beq()`: Branch instructions, modifying the PC based on comparison results.
- `jal()`: A jump instruction, altering the flow of execution by modifying the PC.

Testing and Validation:

- The scripts include a `run_tests()` function that systematically executes each instruction with predefined scenarios, using assertions to validate the outcomes.
- For instance, in testing `addi`, the script simulates the addition of an immediate value to a register's content and checks if the result is as expected.

Mini-Program Simulation and Instruction Encoding:

- The mini-program serves as a practical example of how the instruction set can be utilized to perform a real-world task, such as summing an array.
- Instruction Encoding (`encode_instruction(opcode, r1, r2, imm)`):
 - This function is fundamental to the mini-program. It encodes the instructions by compactly packing the opcode, register identifiers, and immediate values into a single instruction format.
 - The encoded instructions are then stored in `InstMem`, ready to be executed sequentially.
- Execution Flow:
 - The mini-program is designed to initialize an array in memory and calculate its sum using the `lw` (load), `add`, and `bgt` (branch) instructions.
 - It demonstrates the integration of multiple instructions and showcases how the system handles memory operations, arithmetic operations, and conditional branches.
- Debugging and Validation:

- Enhanced print statements are used within the mini-program to provide real-time feedback on the execution flow, which aids in debugging and understanding how each instruction alters the state of the simulation.
- The final assertion checks if the calculated sum matches the expected result, validating the correct execution of the mini-program.

APPENDIX (please put all required stuff above this, atm anything beyond this is the appendix)

A.1

```
# Initialize memory and registers
InstMem = [0] * 1024 # Instruction memory
Reg = [0] * 16       # 16 registers
Mem = [0] * 1024     # Data memory

# Program Counter
PC = 0

def reset():
    global Reg, Mem, PC
    Reg = [0] * 16
    Mem = [0] * 1024
    PC = 0

# Helper function for sign-extension
def SE(value):
    if value & 0x8: # If the highest bit (bit 3) is set
        return value | 0xFFFF0 # Extend with 1s
    return value

# Instruction Functions
def addi():
    global PC
    inst = InstMem[PC]
    a = Reg[(inst >> 8) & 0xF]
    imm = SE((inst >> 4) & 0xF)
    Reg[inst & 0xF] = a + imm
    PC += 2

def add():
    global PC
    inst = InstMem[PC]
    a = Reg[(inst >> 8) & 0xF]
    b = Reg[(inst >> 4) & 0xF]
    Reg[inst & 0xF] = a + b
    PC += 2
```

```

def sw():
    global PC
    inst = InstMem[PC]
    a = Reg[(inst >> 8) & 0xF]
    b = Reg[(inst >> 4) & 0xF]
    imm = SE(inst & 0xF)
    Mem[a + imm] = b
    PC += 2

def lw():
    global PC
    inst = InstMem[PC]
    a = Reg[(inst >> 8) & 0xF]
    imm = SE((inst >> 4) & 0xF)
    Reg[inst & 0xF] = Mem[a + imm]
    PC += 2

def bgt():
    global PC
    inst = InstMem[PC]
    a = Reg[(inst >> 8) & 0xF]
    b = Reg[(inst >> 4) & 0xF]
    imm = SE(inst & 0xF) << 1
    target = PC + imm
    if a > b:
        PC = target
    else:
        PC += 2 # Increment PC if condition is not met

def beq():
    global PC
    inst = InstMem[PC]
    a = Reg[(inst >> 8) & 0xF]
    b = Reg[(inst >> 4) & 0xF]
    imm = SE(inst & 0xF) << 1
    target = PC + imm
    if a == b:
        PC = target
    else:

```

```

        PC += 2

def jal():
    global PC
    inst = InstMem[PC]
    imm = SE((inst >> 8) & 0xFF) << 1
    PC += imm

def sub():
    global PC
    inst = InstMem[PC]
    a = Reg[(inst >> 8) & 0xF]
    b = Reg[(inst >> 4) & 0xF]
    Reg[inst & 0xF] = a - b
    PC += 2

# Test routines
def run_tests():
    # Reset memory and registers between tests
    global Reg, Mem, PC

    def reset():
        global Reg, Mem, PC
        Reg = [0] * 16
        Mem = [0] * 1024
        PC = 0

    # Test for 'addi'
    print("Testing 'addi'")
    reset()
    InstMem[0] = (1 << 8) | (3 << 4) | 2 # addi: R2 = R1 + 3; assuming
R1=1
    Reg[1] = 1
    addi()
    assert Reg[2] == 4, "addi failed"
    print("addi passed")

    # Test for 'add'
    print("Testing 'add'")
    reset()

```



```

    InstMem[0] = (1 << 8) | (2 << 4) | 3 # add: R3 = R1 + R2; assuming
R1=1, R2=2
    Reg[1] = 1
    Reg[2] = 2
    add()
    assert Reg[3] == 3, "add failed"
    print("add passed")

    # Test for 'sw'
    print("Testing 'sw'")
    reset()
    InstMem[0] = (1 << 8) | (2 << 4) | 3 # sw: Mem[R1+3] = R2; assuming
R1=1, R2=5
    Reg[1] = 1
    Reg[2] = 5
    sw()
    assert Mem[4] == 5, "sw failed"
    print("sw passed")

    # Test for 'lw'
    print("Testing 'lw'")
    reset()
    Mem[4] = 8
    InstMem[0] = (1 << 8) | (3 << 4) | 3 # lw: R3 = Mem[R1+3]; assuming
R1=1, Mem[4]=8
    Reg[1] = 1
    lw()
    assert Reg[3] == 8, "lw failed"
    print("lw passed")

    # Test for 'bgt'
    print("Testing 'bgt'")
    reset()
    InstMem[0] = (2 << 8) | (1 << 4) | 2 # bgt: if R2 > R1, PC += 4;
assuming R1=1, R2=3
    Reg[1] = 1
    Reg[2] = 3
    bgt()
    assert PC == 4, "bgt failed"
    print("bgt passed")

```

```

    # Test for 'beq'
    print("Testing 'beq'")
    reset()
    InstMem[0] = (1 << 8) | (1 << 4) | 2 # beq: if R1 == R1, PC += 4;
assuming R1=1
    Reg[1] = 1
    beq()
    assert PC == 4, "beq failed"
    print("beq passed")

    # Test for 'jal'
    print("Testing 'jal'")
    reset()
    InstMem[0] = (3 << 8) # jal: PC += 6; immediate = 3
    jal()
    assert PC == 6, "jal failed"
    print("jal passed")

    # Test for 'sub'
    print("Testing 'sub'")
    reset()
    InstMem[0] = (2 << 8) | (1 << 4) | 3 # sub: R3 = R2 - R1; assuming
R1=1, R2=3
    Reg[1] = 1
    Reg[2] = 3
    sub()
    assert Reg[3] == 2, "sub failed"
    print("sub passed")

    print("All tests passed!")

def encode_instruction(opcode, r1, r2, imm):
    return (opcode << 12) | (r1 << 8) | (r2 << 4) | imm

def mini_program_test():
    # Reset memory and registers
    reset()

```

```

# Initializing an array in memory
array = [1, 2, 3, 4, 5] # Example array
array_address = 10      # Starting address of the array in memory
Mem[array_address:array_address + len(array)] = array

# Writing a mini-program using our instruction set to calculate the
sum
# Assuming we use R1 for the sum, R2 for array elements, R3 as index,
R4 for array size
Reg[3] = array_address # Starting address of the array
Reg[4] = len(array)    # Size of the array

# Program instructions
program = [
    encode_instruction(1, 0, 0, 1), # addi R1, R1, 0 (initialize R1
to 0)
    encode_instruction(3, 2, 3, 2), # lw R2, R3 (R2 = Mem[R3])
    encode_instruction(2, 1, 1, 2), # add R1, R1, R2 (R1 = R1 + R2)
    encode_instruction(1, 3, 3, 3), # addi R3, R3, 1 (R3 = R3 + 1)
    encode_instruction(4, 4, 3, 0xFB), # bgt R4, R3, -10 (if R4 > R3,
jump back 10 bytes to continue loop)

]

for i, inst in enumerate(program):
    InstMem[i * 2] = inst

# Run the mini-program
while PC < len(program) * 2:
    inst = InstMem[PC]
    opcode = inst >> 12
    print(f"Executing instruction at PC={PC}, inst={inst:04x},
opcode={opcode}") # Enhanced debug output
    if opcode == 1:
        addi()
    elif opcode == 2:
        add()
    elif opcode == 3:
        lw()
    elif opcode == 4:

```

```
        bgt()
    else:
        print(f"Unrecognized opcode: {opcode}")
        break

# Verify the result
expected_sum = sum(array)
assert Reg[1] == expected_sum, f"Mini-program failed, expected sum {expected_sum}, got {Reg[1]}"
print(f"Mini-program passed, correctly calculated the sum of {array} as {Reg[1]}")

if __name__ == "__main__":
    run_tests()
    print("Running Mini-Program Test")
    mini_program_test()
```