Design Document

Team 1, Section 3
Ethan Bennett, Medhansh Khattar, Harrison Carpenter

1.Design Philosophy

Our design is aimed at reducing the total number of instruction types to implement 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. Our design is inspired by RISC-V and follows a load-store architecture. Once we decided on 16-bit core instruction formats, we moved on to developing RTLs for each and planning all components required for our Datapath in the future. Eventually, we decided to shift to multicycle RTLs rather than pipelining Single-Cycle RTLs. We then developed a multicycle Datapath and its control states, before integrating and testing it all on Quartus. All components are written in Verilog.

2.Performance

The performance will be measured using the 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.

3.1 Registers

In our 16-bit processor design, registers are at the heart of executing operations efficiently. These small, fast storage locations are directly accessible by the CPU, designed to temporarily hold data, instructions, or addresses that are being worked on. Registers significantly speed up the processor's ability to execute instructions by reducing the time needed to access data from memory.

Our processor is equipped with a carefully selected set of registers, each designated for specific roles

Special Purpose Registers:

- Zero Register (x0): A unique register always containing the value zero, useful for operations requiring a constant zero.
- Return Address (x1, ra): Holds the address to which the processor will return after a function call, essential for navigating through code execution.
- Stack Pointer (x2, sp): Points to the current top of the stack, managing the call stack for functions and local variables.

Input/Output Registers:

- User Input (x3, IN): Dedicated to receiving inputs from external sources, enabling interaction with the processor.
- Output (x4, OUT): Stores data intended for external output, facilitating communication from the processor to the outside world.

Temporary and Assembler Registers:

- Temporaries (x5-x7, t0-t2): These registers offer scratchpad storage for temporary data during quick calculations and intermediate steps.
- Function Argument (a0-a4): These are the registers used to pass function arguments.

3.2 List of 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	s2	
x11	a0	Function arguments and return values.

x12	a1	
x13	a2	
x14	a3	
x15	a4	

Fig 1: List of Registers

4. Instruction Types

We have 5 different types of Instructions

- Arithmetic Type (A-Type)
- Immediate Type (I-Type)
- Store Type (S-Type)
- Branch Type (B-Type)
- Jump and Link Type (JAL-Type)

4.1 A-Type (Arithmetic)

Allows for two arithmetic operations (addition and subtraction) between 2 registers.

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

Fig 1.2 Instruction Format

The A-type takes in 3 registers, the register destination and 2 source registers rs1 and rs2, each of 4 bits. The opcode helps our processor identify that this is an A-Type (000). Similarly the 1 but funct chooses add/sub.

The add function takes the values stored in register source 1 & 2, adds them and stores the result in rd, register destination. It has the opcode 000, and funct 0.

The sub function takes the values stored in register source 1 & 2, subtracts rs2's value from rs1's, and stores the result in rd, register destination. It has the opcode 000, and funct 1.

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]

Fig 1.3 Instructions of this type

4.2 I-Type (Immediate)

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

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

Fig 1.4 Instruction Format

The I-type takes in 2 registers, the register destination and 1 source register, of 4 bits each. It also takes in a 4 bit immediate, a 3 bit opcode, and a 1-bit funct. The opcode for the I-Type is 001. It supports 2 type, addi and lw.

The addi (add immediate) takes in one register operand, register source 1 and the immediate. It adds them and stores it in rd, the register destination. With the opcode 001, its funct is 0.

The lw (load word) loads a value from the memory and stores it into the register destination. To calculate the memory address , it takes the address from register source 1, adds the 4-bit immediate byte offset. With the opcode 001, its funct is 1.

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)

Fig 1.5 Instructions of this type

4.3 S-Type (Store)

Allows storing of a register value to memory.

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

Fig 1.6 Instruction Format

The s-type takes in 2 registers, rs1 and rs2, each of 4 bits. It also takes in 5 immediate bits allowing for a larger offset. Additionally, it has a 3 bit opcode (010). Notably, the s-type does not have a funct bit since it has only one instruction in this type, the store word.

The sw (store word) takes the address stored in rs1, adds the 4 bit byte offset to calculate the target memory address. THen it stores the value in register source 2 onto that memory address. It has no funct bit.

To make the hardware lineup for rs2 and rs1, we chose the immediates to be from 7:3 for S-Type. Later as we examine Control Signals,we will see that we don't need space for rd, as we have regwrite as 0.

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

Fig 1.7 Instructions of this type

4.4 B-Type (Branch)

Allows for two types of branches, branch equal (beq) and branch greater than (bgt).

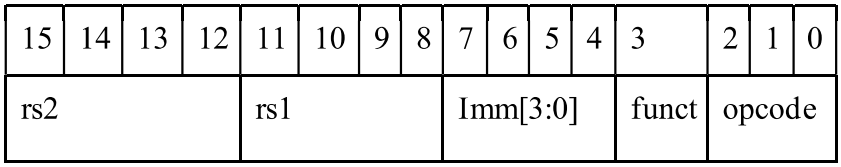


Fig 1.8 Instruction Format

The B-Type takes in 2 register sources for a boolean comparison, each of 4 bits. It also takes in 4 bits of immediate that form the branching offset. The B-type takes in 3 bits of opcode (011) and 1 bit funct to differentiate between beq/bgt.

The beq, branch if equal branches if the value in rs1 is equal to the value in rs2. It has 4 bits of immediate that calculate the offset to branch. Essentially it offsets Program Counter by the immediate value if condition is satisfied. It has opcode 011 and funct 0.

Similarly, the bgt function, branch if greater than, branches if the value in rs1 is greater than the value in rs2. It has 4 bits of immediate that calculate the offset to branch. Essentially it offsets Program Counter by the immediate value if condition is satisfied. It has opcode 011 and funct 1.

To make the hardware lineup for rs2 and rs1, we chose the immediates to be from 7:3 for S-Type. Later as we examine Control Signals,we will see that we don't need space for rd, as we have regwrite as 0.

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

Fig 1.9 Instructions of this type

4.5. JAL-Type (Jump-And-Link)

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**.

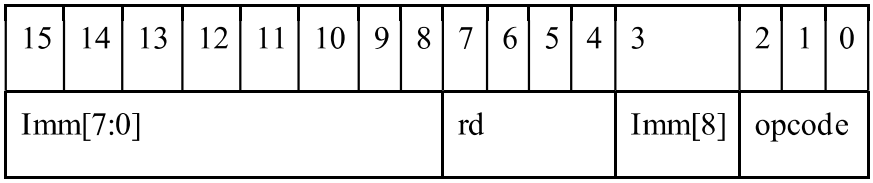


Fig 1.10 Instruction Format

The JAL-type takes in an 9-bit immediate which allows a large branching range. It also takes in a 4 bit register destination, and a 3 bit opcode (100). Notably, it has no requirement for a funct.

The JAL_type doesn't require rs1 and rs2 because in the Execute Stage, both inputs are not selected in our multiplexer. Instead to accommodate a higher offset range, we allocate 7 bits from 15:8.

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

Fig 1.11 Instructions of this type

5. Memory Map

The memory map of our 16-bit processor defines how the memory space is organized and utilized. It is essential for both the hardware and software to understand where different types of data and instructions are stored and how they are accessed.

Text Segment **(0x0000 to 0x00FF)**: This region is reserved for the machine code of our program. It is where all the executable instructions reside. The text segment typically contains read-only data, therefore the instructions are not meant to be modified during execution.

Data Segment **(0x0100 to 0x7FFF)**: The data segment is used for variables and dynamically allocated data during program execution. This space is where the processor stores and modifies variables as the program runs. It's designed to be read-write, allowing both storage and retrieval of values.

Stack Segment **(0x8000 to 0xFFFF)**: The stack is a special region of memory that grows downwards, typically used for function call management. It stores return addresses, local variables, and is instrumental in handling function calls, interrupts, and subroutine calls. The stack pointer register (sp) manages the stack's top, providing access to the last stored element.

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

Fig 1.12 Memory Map

6. Addressing Modes

6.1. Register Addressing Mode:

Used in A-Type (Arithmetic) instructions, which do not utilize immediate bits. Operates by using register operands for computations, such as addition or subtraction between two registers, like add x1, x5, x6. The instruction directly references the physical registers on the processor where operands are located.

6.2 Immediate Addressing Mode:

In I-Type (Immediate) instructions, immediates are used to either add directly to a register's value or to provide an offset from a base address in memory. For example, addi x5, x6, 7 uses an immediate to add to the value in register x6 and stores the result in x5.
In lw x5, 4(x6), the immediate 4 is used to offset from the address contained in x6 to load a word into x5.

In S-Type (Store) instructions, immediates serve to offset memory addresses for storing values from registers.Example: sw x5, 4(x6) uses the immediate to calculate the memory address offset from the base address in x6 to store the contents of x5.

6.3 PC-Relative Addressing Mode:

Utilized by B-Type (Branch) and JAL-Type (Jump and Link) instructions.
These instructions employ immediate values to offset the Program Counter (PC). This is considered PC-relative addressing because the offset is applied to the current location of the program counter, enabling the instruction sequence to jump to a new code location based on the current point of execution. It maintains relative jumps, ensuring that the instruction's behavior remains consistent regardless of where it appears in the program. For instance, beq x5, x6, -8 will branch to an instruction 8 steps back if the values in x5 and x6 are equal, and jal x0, 16 will jump to a location 16 steps ahead, storing the return address in x0.

6.4 Symmetry and Rearrangement in Immediate Values:

In S-Type instructions, immediate values are rearranged to maintain symmetry with other instruction types. This rearrangement ensures that different instructions that use immediate values have a consistent way of encoding these values, which simplifies the decoding process during instruction execution.

7. Procedure Calling Conventions

In a 16-bit processor environment, the calling conventions establish a framework that governs the interactions between a function (the caller) and the function it invokes (the callee). This ensures that both parties manage the stack and the registers effectively, maintaining the integrity of the program flow and state. The caller is responsible for managing the return address and preparing argument registers for the callee. It must also handle the preservation of any temporary registers it needs post-call. Conversely, the callee is tasked with utilizing these provided arguments and managing save registers so that upon its completion, the caller's environment is restored to its original state. Below, the table outlines the specific responsibilities that fall to the caller and callee, enabling a harmonious and error-free function call process.

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

Fig 1.13 Caller vs Callee Responsibilities

7.1 Procedure calling convention example

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

The assembly code provided is an example of a procedure calling convention in a 16-bit processor architecture. This code snippet demonstrates how a calling function (callAddTwo) prepares for a procedure call, how it calls another function (addTwo), and then how it cleans up afterward. The addTwo procedure itself shows how it performs its task and returns control back to the caller. We also list the line numbers and explain which part of the calling convention each line demonstrates.

Lines 1 and 2 are where the caller prepares for the call by handling the stack and preserving the return address. Line 3 is the actual procedure call. Lines 4 and 5 show the callee restoring the caller's state before returning. Line 6 is where the callee hands control back to the caller. Finally, Lines 7 and 8 demonstrate the callee performing its operation and preparing to return control back to the caller. This sequence ensures both the caller and callee uphold their responsibilities, allowing for seamless execution of the program.

8. Machine Language Translation of relPrime

The following table contains our code for rel prime, and the converted machine code based on the instruction types we made earlier. For your reference there is a comment section on the end which gives us an English explanation for our assembly code. The address increments by 2, indicating PC+2

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

Fig 1.14 Assembly and Machine code for relPrime

9. Single-Cycle RTL

In the context of our 16-bit design, Single-Cycle RTL (Register Transfer Language) describes the micro-operations that occur within one clock cycle for each instruction. This approach ensures that every instruction executes in a single cycle of the clock, which simplifies the control logic and can potentially lead to higher clock frequencies. Note: We decided to use Single Cycle only as a reference before moving onwards to multicycle RTL, hence we don't have a supporting Datapath for this step.

9.1 ADD

This instruction begins by incrementing the PC to point to the next instruction. It then retrieves the current instruction from the instruction memory at the PC address. The instruction specifies two source registers whose contents are fetched from the register file. The values in these registers are added together, and the result is written back into the destination register specified by the instruction.

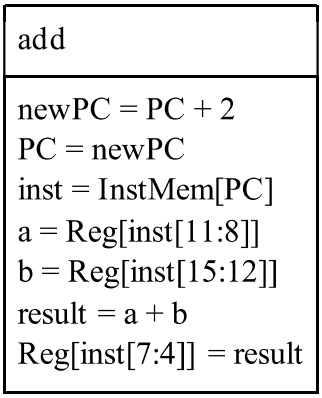


Fig 1.15 ADD SRTL

9.2 SUB

The PC is incremented to point to the next instruction, which is then fetched from instruction memory. The specified registers are read, and their contents are subtracted (the value in the second register is subtracted from the first). The result of this subtraction is stored back into the destination register.

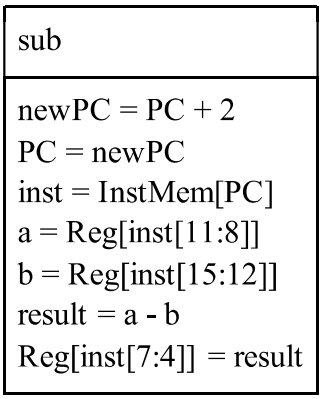


Fig 1.16 SUB SRTL

9.3 Add Immediate (ADDI)

Similar to the ADD instruction, this operation also increments the PC and fetches the current instruction. However, instead of reading two registers, it reads one register and an immediate value encoded within the instruction itself. The immediate value is sign-extended to match the register size and added to the value in the register. The result is then stored back into the destination register.

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

Fig 1.17 ADDI SRTL

9.4 Load Word (LW)

The LW instruction increments the PC and fetches the instruction. It reads a base address from one of the registers and adds it to a sign-extended immediate value to calculate the effective memory address. The word at this calculated memory address is then loaded and written into the destination register.

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

Fig 1.18 Load Word SRTL

9.5 Store Word (SW)

Like LW, SW increments the PC and fetches the instruction. It calculates the effective memory address in the same way, by adding a register's value to a sign-extended immediate value. The instruction specifies a source register whose content is then stored into the calculated memory address in the data memory.

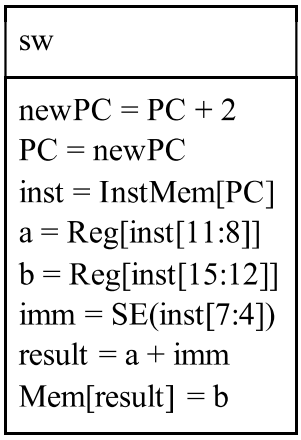


Fig 1.19 SW SRTL

9.6 Branch If Equal (BEQ)

After incrementing the PC, the instruction is fetched, and two registers are read. If the values in these registers are equal, the PC is updated to branch to a new address. This new address is the sum of the PC and a sign-extended, shifted immediate value from the instruction, effectively allowing the program to jump to a different part of the code.

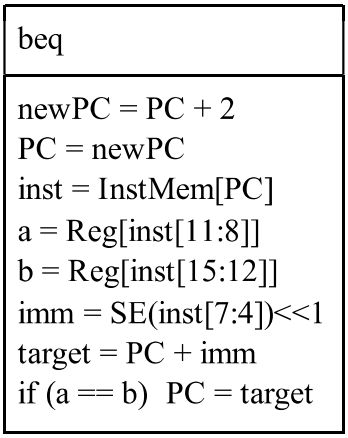


Fig 1.20 BEQ SRTL

9.7 Branch Greater Than (BGT)

This instruction also starts by incrementing the PC and fetching the next instruction. It compares the contents of two registers, and if the first is greater than the second, it branches to a new address calculated by adding the PC with a sign-extended, shifted immediate value.

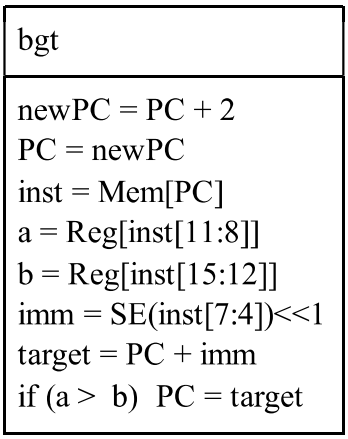


Fig 1.21 BGT SRTL

9.8 Jump and Link (JAL)

The JAL instruction increments the PC and fetches the instruction. It then sign-extends and shifts an immediate value, which is added to the PC to calculate the jump target address. The PC is updated to this new address, enabling the program to jump to a subroutine. Additionally, the address following the jump (the return address) is stored in a designated link register to allow for a return to the instruction following the JAL after the subroutine completes.

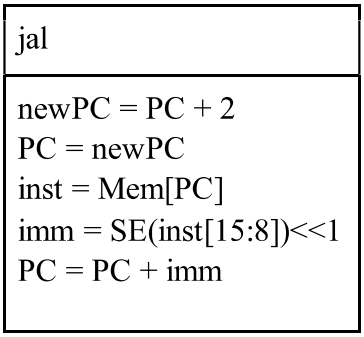


Fig 1.22 JAL SRTL

10. Component Specifications (Single Cycle)

Our 16-bit processor is designed with a set of core components, each serving a distinct purpose within the single-cycle processing structure. These components, encompassing registers, register files, memory units, and computational elements, work together to execute a wide range of instructions.

Registers capture and hold data, facilitating swift data retrieval. The register file acts as the central hub for accessing and writing register values, pivotal for the processor's operations. Instruction and data memories serve as repositories for instructions and data, respectively, enabling the sequential execution of instructions and manipulation of data. The Immediate Generator plays a crucial role in decoding and extending immediate values from instructions, providing the necessary operands for computation. At the center of arithmetic and logical operations is the ALU, which performs the actual computations required by the instructions. Lastly, the PCAdder is responsible for advancing the program counter, ensuring progression from one instruction to the next.

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 at that location in memory.	InstMem
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	ImmGen, SE

			corresponding 9 bit immediate then sign extends it to 16 bits.	
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

Fig 1.23 Single Cycle Component Specification

11. Component Specifications (Multi Cycle)

The multi-cycle architecture of our 16-bit processor employs a refined set of components, each tailored to perform specific tasks within the instruction execution process. Unlike single-cycle processors, where each instruction is executed in one clock cycle, the multi-cycle design breaks down the execution into several stages, allowing each component to operate over multiple clock cycles. This multi-cycle methodology enhances the processor's efficiency by allowing individual components to process parts of instructions concurrently. In the subsequent component specifications, we detail the inputs, outputs, and behaviors of the core hardware elements: Registers, Register File, Immediate Generator, ALU, and Memory.

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

Fig 1.24 Multi-Cycle Component Specification

12. Datapath

The datapath depicted here is a schematic of our multi-cycle processor's internal structure, outlining the flow and transformation of data during instruction execution. At the helm is the Program Counter (PC), which sequentially feeds the processor with instruction addresses. These instructions are fetched into the Instruction Register (IR) from the main memory. The Register File acts as the reservoir for operands, supplying the Arithmetic Logic Unit (ALU) with data via registers A and B. The ALU executes arithmetic and logical operations, outputting results to the ALUOut register.

This datapath supports both immediate and register-based addressing: the Immediate Generator (IMM) extracts and prepares immediate values for execution, while the Memory Data Register (MDR) facilitates data transfers between the processor and memory. PC-relative addressing allows for conditional and unconditional branching, critical for control flow within programs, whereas register-offset addressing is used for accessing data in memory, relative to a base address in the register file.

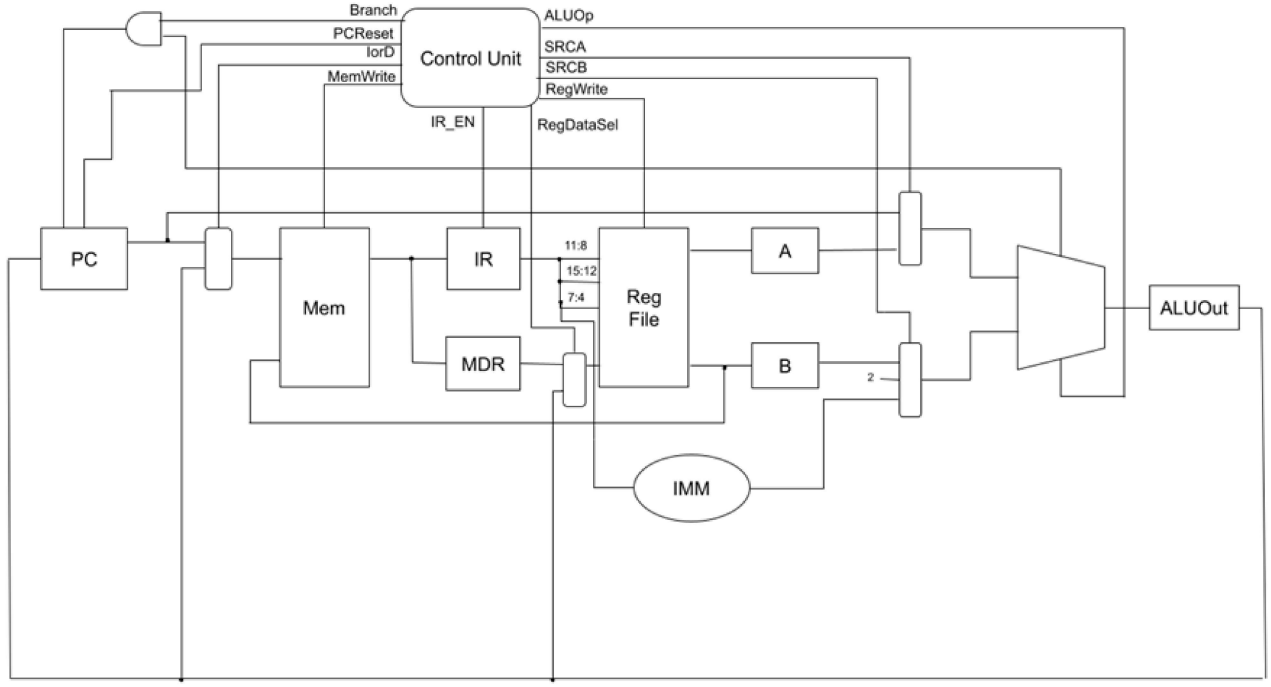


Fig 1.25 Datapath

13. Multi-Cycle RTL

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

Reg[IR[7:4]] <= ALUOut	MDR <= Mem[ALUOut]	Mem[ALUOut] <= B		PC <= ALUOut
	Reg[IR[7:4]] <= MDR			

Fig 1.26 Multicycle RTL

The multi-cycle RTL representation for our 16-bit processor encapsulates the sequence of operations that occur over several clock cycles to execute different types of instructions. The multi-cycle approach allows for a more resource-efficient design, hence we choose to pivot towards multi-cycle from this point onwards.

14. Control Signal Descriptions

This section has the functionality of various control signals within the processor. Each bit-wide signal is responsible for orchestrating a specific aspect of the operation. For instance, the 'Branch' signal determines the validity of a branching operation, while 'MemWrite' and 'RegWrite' control the write operations to memory and the register file, respectively. Other signals like 'IorD', 'RegDataSel', 'SRCASel', 'SRCBSel', 'ALUOp', 'IREN', and 'PC Reset' manage the flow of data through the processor—selecting sources for the ALU, enabling the instruction register, and resetting the program counter, amongst others.

Branch	1 bit	ANDed with Zero to determine if a branch is valid
IorD	1 bit	Controls source memory address (PC 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
SRCASel	1 bit	Chooses ALU Source A
SRCBSel	2 bits	Chooses ALU Source B
ALUOP	1 bit	Chooses ALU Operation
IREN	1 bit	Enables writing to IR register
PC Reset	1 bit	Resets PC to zero

Fig 1.27 Control Signal Table

15. Control Signal State Diagrams

The diagram below offer a visual representation of the states and transitions between various control signals during different phases of instruction processing. It tells how the control unit's signals change from the fetch to the decode and execution stages, depending on the type of instruction being processed. The state diagrams break down complex instruction cycles into simpler stages, showing the sequential activation of control signals that lead to the successful execution of instructions like 'add', 'sub', 'addi', 'lw', 'sw', 'JAL', and conditional branches. Each diagram node specifies the control signals' states, effectively mapping the journey an instruction follows as it moves through the processor's datapath.

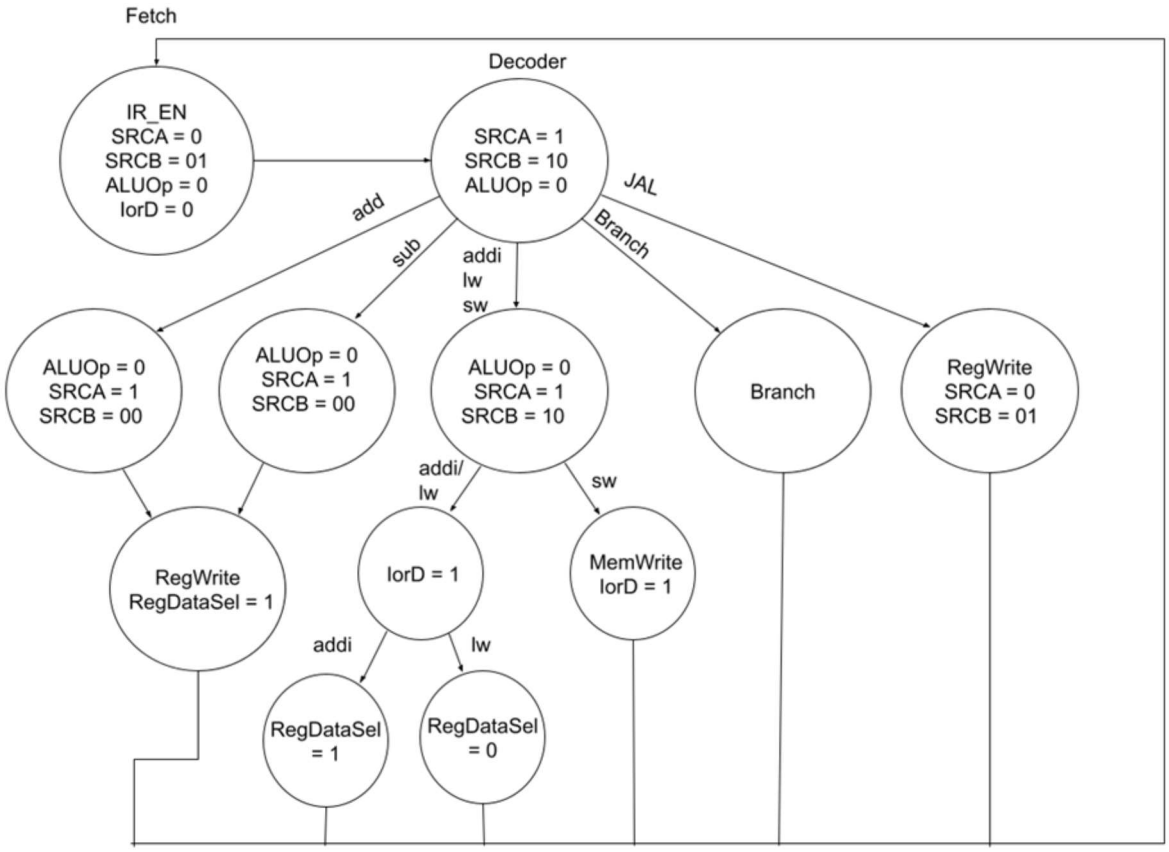


Fig 1.28 Control Signal Table

16. Test Checklist

The following is the list of tests that we plan to complete for each component. It outlines various tests conducted on the Arithmetic Logic Unit (ALU), detailing operations such as addition and subtraction with different operand signs and their expected outcomes. This includes simple positive and negative integer operations, ensuring that the ALU handles sign extension from the imam generator and two's complement arithmetic correctly. Additionally, the test results for the Program Counter (PC) and Memory are provided, which verify the processor's ability to write to the PC, execute memory write and read operations, and confirm the correct behavior when writing to specific memory addresses.

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	Memory Out = 1	Overwrite address 299

	MemWrite = 1		
Register File	Address 1 = 8 Address 2 = 0 Destination = 1 Regwrite = 1 Data = 50	1 <= 50	Write 50 to address 1, and read from 8 and 0.
	Address 1 = 2 Address 2 = 0 Destination = 1 Regwrite = 0 Data = 50	Nothing written	Read from 8 and 0. Do not write.
	Address 1 = 1 Address 2 = 0 Destination = 0 Regwrite = 1 Data = 75	0 <= 75	Write 75 to Address 0. Read from 1 and 0.
	Address 1 = 1 Address 2 = 0 Destination = 0 Regwrite = 0 Data = 75	Nothing written	Read from 1 and 0, do not write.

Fig 1.29 Test Checklist

17. Integration Test Plan

This table provides a structured approach to validating the processor's design through incremental integration and testing of various components. We follow the principle of building bottom up, adding the smallest of components to minimize errors. Starting with combining the PC and ALU to ensure the PC can be correctly incremented, the plan progresses through stages that include multiplexers (MUXes) and memory, with each step introducing new components into the integration. The plan culminates with full integration tests, one without the control unit or immediate generator, and the final one including all component.

Number	INTEGRATION	COMPLETE?	Description

1	PC + ALU	YES	PC can be incremented by ALU.
2	PC + SRCB MUX + ALU	YES	Integration 1, but with a mux in place of source B.
3	PC + SRCA MUX + SRCB MUX + ALU	YES	Integration 2, with a mux in source A.
4	PC + MEM + ALU + MUXES	YES	Fetch instruction, then increment pc + write ALU result to memory
5	Partial Full Integration (No control unit or Immediate Generator)	YES	Full integration, no control unit or immediate generator
6	Full Integration	Yes, but failure exists	Full integration and implementation of datapath

Fig 1.30 Test Checklist

18. Assembler

(For code refer to Appendix A.5)

The assembler translates assembly language instructions into 16-bit machine code suitable for a custom processor. It supports various types of instructions, such as arithmetic operations (add, sub), immediate operations (addi, lw), store (sw), branch (beq, bgt), and jump (jal). To encode an instruction, it parses the instruction mnemonic and operands, then generates the corresponding binary opcode, function code, and register or immediate values based on the instruction set architecture (ISA) specification. Register operands are converted to their 4-bit binary equivalents, while immediate values are sign-extended and converted to binary with the required bit length. The resulting bits are concatenated to form the complete 16-bit machine code. The function ensures that the machine code for each instruction type adheres to the predefined format, considering the specific bit lengths for opcodes, function codes, registers, and immediate as defined by the ISA.

19. Python Scripts for Single-Cycle RTLs

(For Code refer to Appendix A.6)

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

19.1 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.

19.2 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.

19.3 Core Instruction Function

- 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.

19.4 Testing

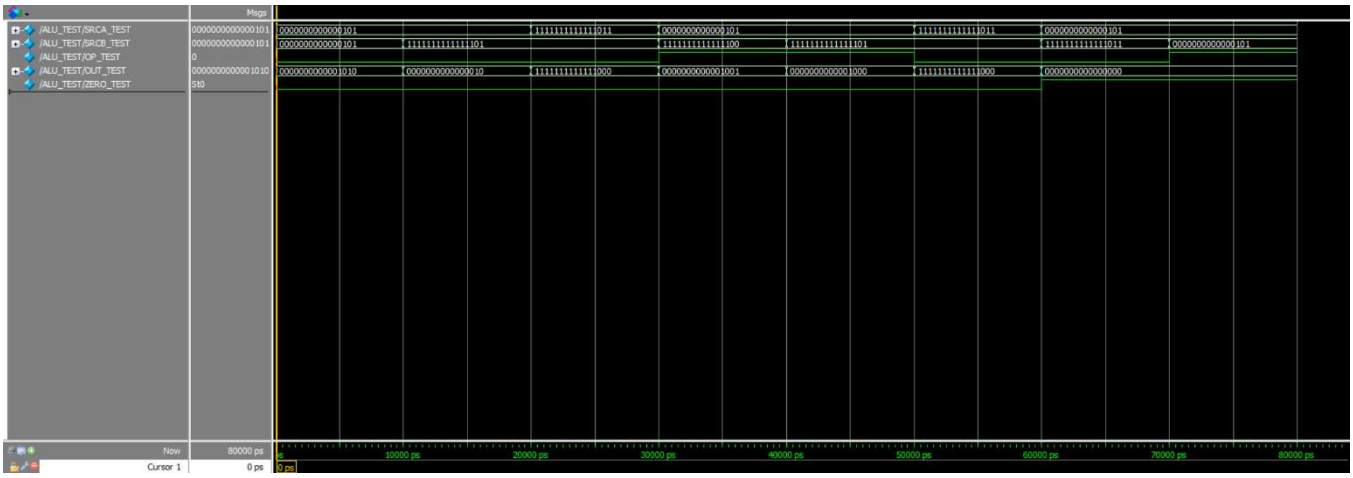
The scripts include a run_tests() function that systematically executes each instruction with predefined scenarios, using assertions to validate the outcomes. For instance, in testing addi, the script simulates the addition of an immediate value to a register's content and checks if the result is as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

19.5 Mini-Program Simulation and Instruction Encoding

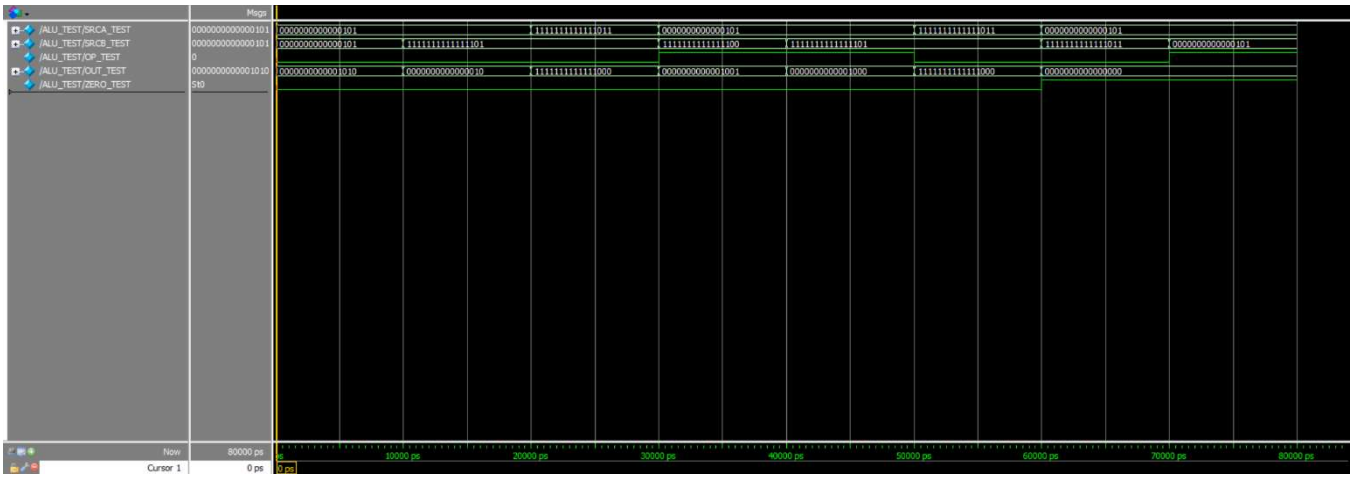
The mini-program serves as a practical example of how the instruction set can be utilized to perform a real-world task, such as summing an array. Instruction Encoding (encode_instruction(opcode, r1, r2, imm)) This function is fundamental to the mini-program. It encodes the instructions by compactly packing 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.

20. APPENDIX

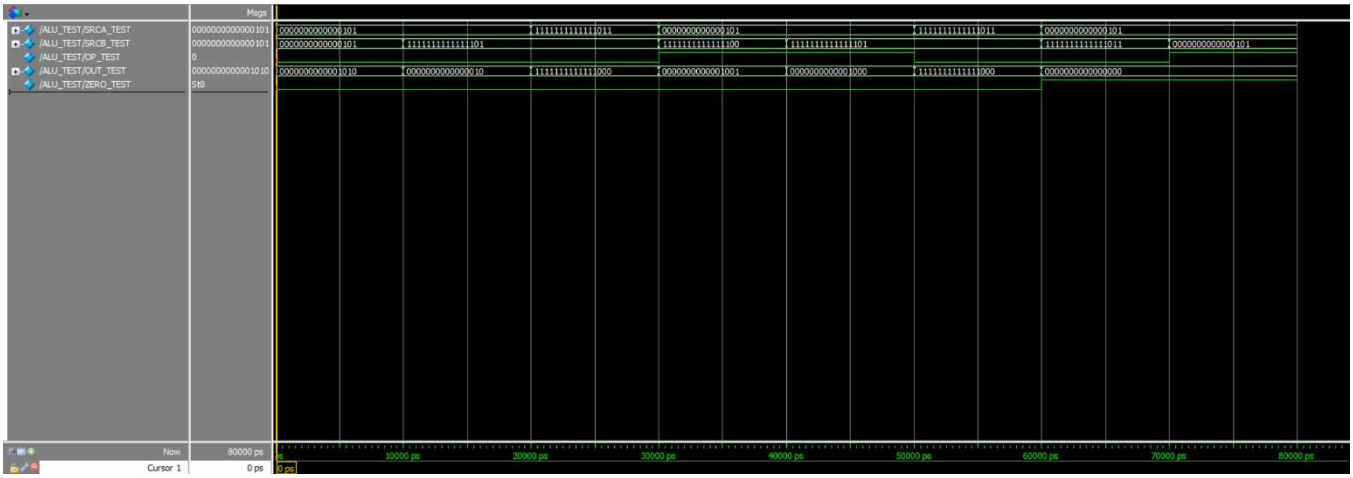
A.1 ALU Test



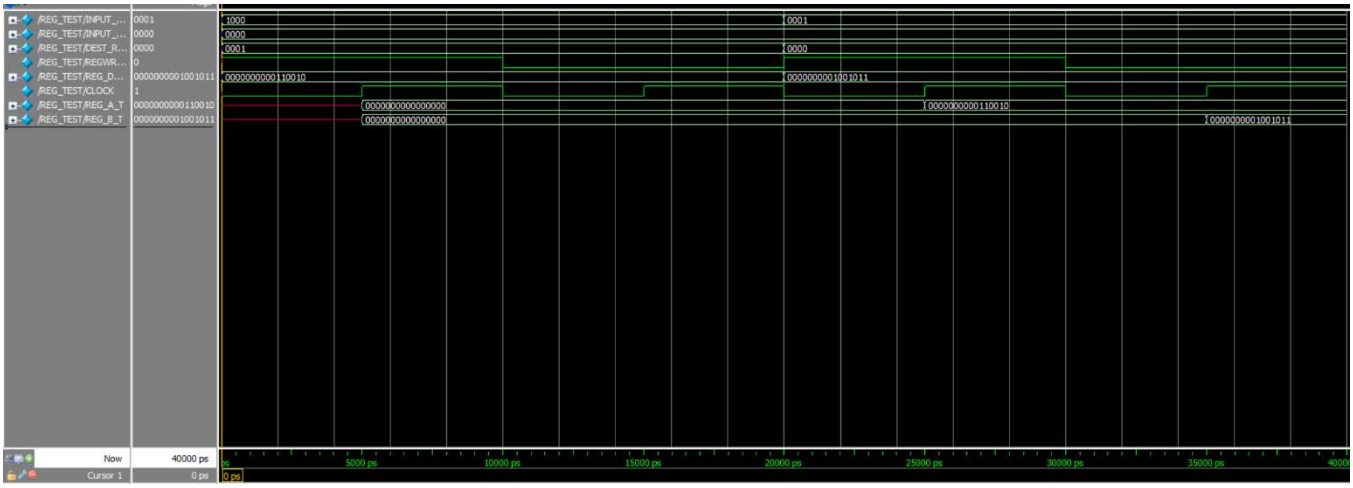
A.2 Memory Test



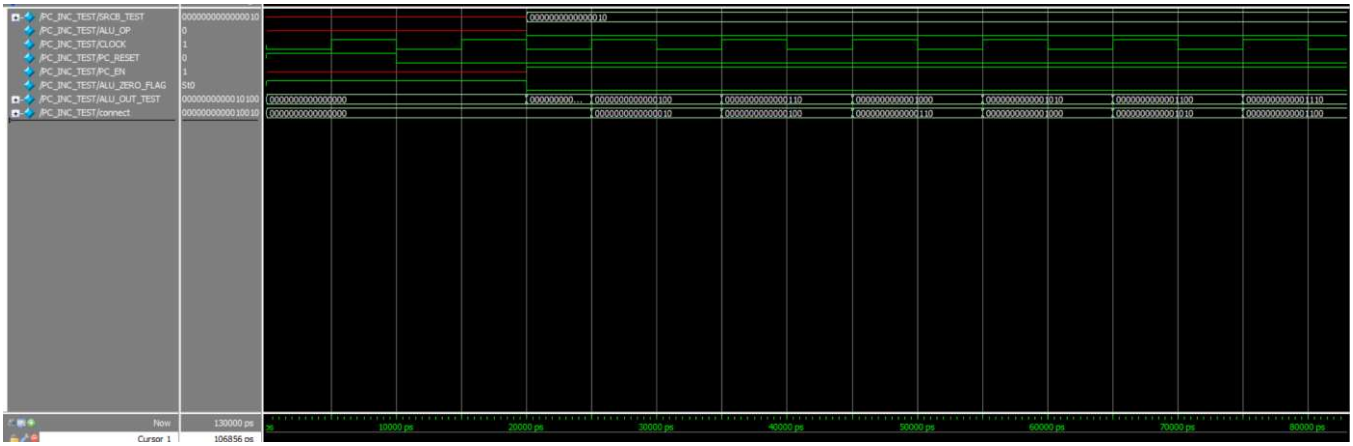
A.3 PC with 2 Muxes



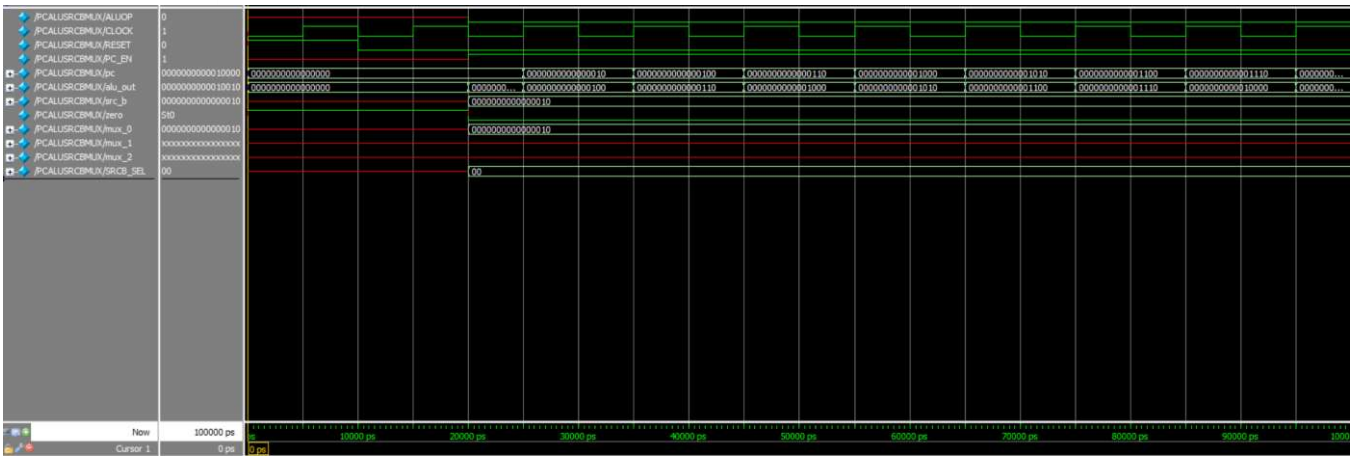
A.4 Register File Test



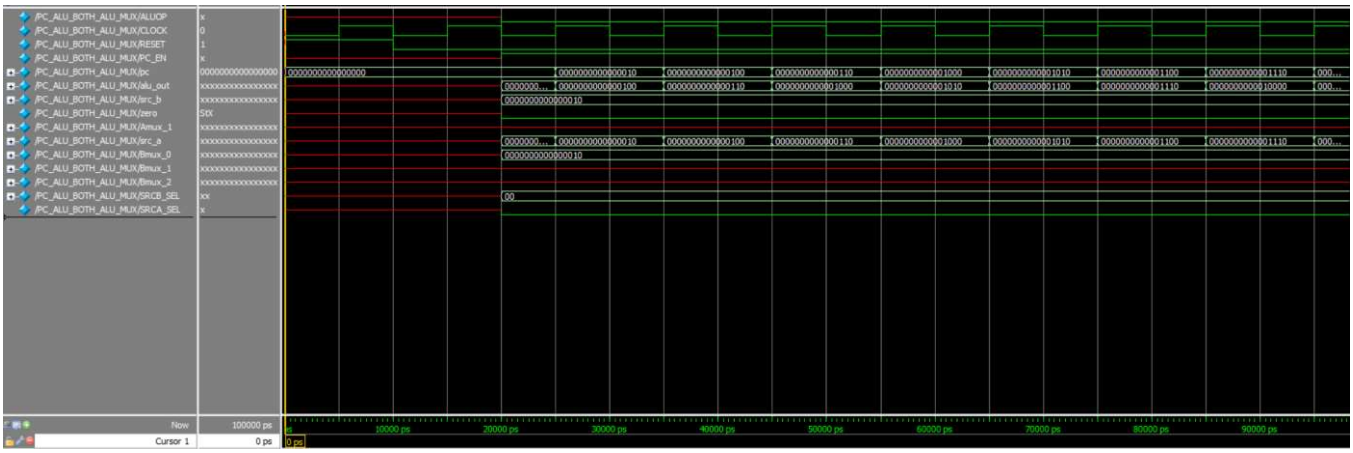
A.5 PC + ALU Integration



A.6 : PC + ALU + MUX

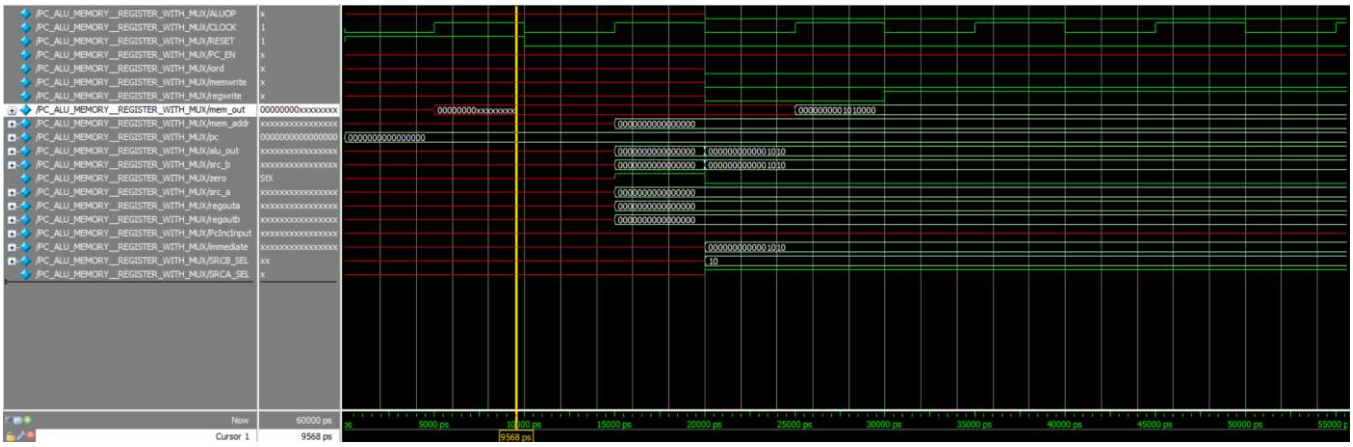


A.7 : PC + ALU + BOTH MUX



A.8 PC + ALU + MEMORY + MUX

A.9 PC + ALU + MEMORY + REGISTER FILE + MU



Note that this one broke somewhere.

A.10 : Assembly Scripts

```
import re

def reg_to_bin(reg_name):
    return format(int(reg_name[1:]), '04b')

def imm_to_bin(imm_value, bits, signed=False):
    value = int(imm_value)
    if signed and value < 0:
        value = (1 << bits) + value
    return format(value, '0{}b'.format(bits))

def assemble_instruction(instruction):
    parts = re.split(r'[ ,]+', instruction)
    opcode = ''
    funct = ''
    rd = ''
    rs1 = ''
    rs2 = ''
    imm = ''

    if parts[0] in ['add', 'sub']:
        opcode = '000'
        funct = '0' if parts[0] == 'add' else '1'
        rd = reg_to_bin(parts[1])
        rs1 = reg_to_bin(parts[2])
        rs2 = reg_to_bin(parts[3])
        machine_code = f'{opcode}{funct}0000{rd}{rs1}{rs2}'
```

```
    elif parts[0] in ['addi', 'lw']:
        opcode = '001'
        funct = '0' if parts[0] == 'addi' else '1'
        rd = reg_to_bin(parts[1])
        rs1 = reg_to_bin(parts[2])
        imm = imm_to_bin(parts[3], 4, signed=True)
        machine_code = f'{opcode}{funct}{rd}{rs1}{imm}'

    elif parts[0] == 'sw':
        opcode = '010'
        rs2 = reg_to_bin(parts[1])
        rs1 = reg_to_bin(parts[2])
        imm = imm_to_bin(parts[3], 5)
        machine_code = f'{opcode}{rs1}{rs2}{imm}'

    elif parts[0] in ['beq', 'bgt']:
        opcode = '011'
        funct = '0' if parts[0] == 'beq' else '1'
        rs1 = reg_to_bin(parts[1])
        rs2 = reg_to_bin(parts[2])
        imm = imm_to_bin(parts[3], 4, signed=True)
        machine_code = f'{opcode}{funct}{rs1}{rs2}{imm}'

    elif parts[0] == 'jal':
        opcode = '100'
        rd = reg_to_bin(parts[1])
        imm = imm_to_bin(parts[2], 9, signed=True)
        machine_code = f'{imm}{rd}{opcode}'[:16]

    else:
        raise ValueError(f"Unsupported instruction: {parts[0]}")

    return format(int(machine_code, 2), '016b')
```

A.11 Python SRTLs Testing

```
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


def SE(value):
    if value & 0x8:
        return value | 0xFFF0
    return value

# Instruction Functions

    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, my tests dont account for this
so far since they are disgned for bgt to apss


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
```

```
# Tests begin here single tests btw
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

    #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")

    #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")

    #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")

    #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")

    # 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")

    # 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")

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

    # su
    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()


    array = [1, 2, 3, 4, 5]
    array_address = 10
    Mem[array_address:array_address + len(array)] = array


    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, maybe offset is halfword so 5?)

    ]

    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
    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()
```