The Lime Instruction Set Manual

Document Version 20240221

Editors: Manohar Tulsi, Pearcy Luke, Raitova Naziia, Wang Yueqiao
Lime-2324b-02 | CSSE 232 | CSSE Department | Rose-Hulman Institute of Technology
2/22/24

Table of Contents


LIME

Introduction

Design Philosophy

The philosophy behind our design prioritizes short and uniformly sized instructions that are executed in a single clock cycle. We have tried to create an architecture that achieves this with minimal difficulty for the programmer. Our architecture sometimes requires the programmer to use multiple instructions for actions that would require only one in other architectures, but we believe this inconvenience is worth the compact and simple instructions.
Despite the small size of our 16-bit instructions, we can still handle immediate values that are up to 16 bits using our special UI register. This ensures that our architecture does not sacrifice performance for uniform instruction sizes and can access 16-bit addressed memory and use large immediate in operations. For convenience, many of our instructions maximize the small portion of the immediate that is part of the instruction, so the UI register does not need to be changed for most operations.

Multi-Cycle Design

The Lime instruction set architecture uses a multi-cycle design to increase flexibility and resource utilization. This approach enables the execution of different instructions in varying numbers of cycles, improving operational efficiency. Furthermore, this design reduces hardware requirements by eliminating the need for additional math operators for program counter (PC) operations.

Compare to RISC-V

Our processor is a load-store architecture like RISC-V, so its instruction set is easy to compare to that of RISC-V. For those that are familiar with RISK-V here is a list of key differences in the Lime instruction set:

- While most immediate values in RISK-V are sign extended, the immediate values in the Lime instruction set are handled differently for each instruction type.
    - For the 2RI type, only 3 bits in the binary instruction are reserved for the immediate values. These 3 bits are appended to the end of the 13 bits stored in the UI register by the most recent lui instruction forming a full 16 bit immediate.

    - For the RI type, the 6-bit immediate provided in the instruction is sign extended.
    - For the UJ type (jal instructions only), the 10-bit immediate provided in the instruction is sign extended.
    - For the L type (lui instructions only), the 13-bit immediate provided in the instruction is stored in the UI (upper immediate) register for future use with 2RI instructions.
- While jumps and branches in RISK-V are both absolute (not based on current PC value), branches in the Lime instruction set are PC relative meaning the provided immediate value is added to PC if the branch is taken.
    - With all branch instructions in the Lime instruction set being 2RI types, this means immediate values provided in branch instructions are appended to the value in the UI register.
    - The branch location is calculated after PC is incremented, so the actual address execution is moved to is one greater than the immediate provided.
- The lime architecture only supports 8 registers as opposed to the 32 registers of RISK-V.
    - For this reason, no register is reserved for the value 0, thus setting the value of registers using addi is inconvenient. The programmer can instead use the set instruction to set registers to a small immediate.
    - Registers x0 and x1 are used as a return address and stack pointer respectively.
    - Register x2 is the only saved register not reserved for a specific purpose (such as the stack pointer)
    - Registers x3, x4, and x5 are temporary registers
    - Registers x6 and x7 are used as procedure arguments with x6 also being the return register

Performance

Introduction

Performance measurement is crucial for evaluating the effectiveness of the Lime architecture. In this section, we discuss the metrics, methodology, and tools used to assess the performance of our processor.

Performance Metrics

The lime structure utilized multi cycle design and greatly utilized the two edges of clock signal, which leads to a better performance over RISC-V on some algorithm.

Cycle per Instruction

In general, lime instruction set has most instruction done by 4 cycle.

Benchmarking

CLK Frequency: 91.19Mhz

Cycle Time: 11.0ns

reprime Algorithm Performance

with Input 0x13b0:

194,113 cycles

2.13 ms

Registers

Programmable Registers

Register	Name	Description	Saver
$x0$	ra	Return address	Caller
$x1$	sp	Stack pointer	Callee
$x2$	$s0$	Saved register	Callee
$x3$	$t0$	Temporary register	Caller
$x4$	$t1$	Temporary register	Caller
$x5$	$t2$	Temporary register	Caller
$x6$	$a0$	Procedure argument (and return)	Caller
$x7$	$a1$	Procedure argument	Caller

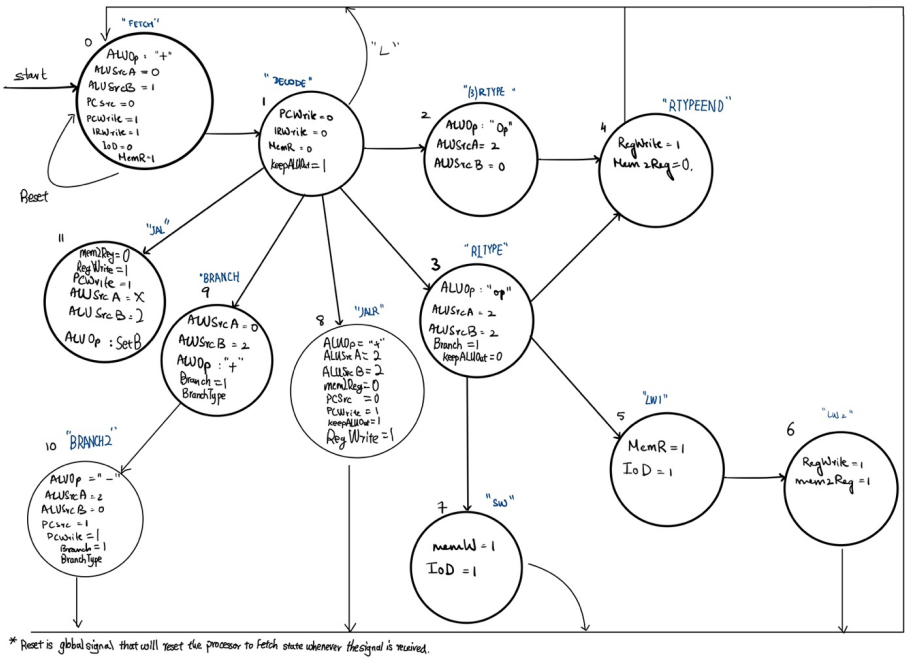
Special Registers

Non-Programmable Registers

Register	Name	Description
UI	Upper Immediate	For storing the most significant 13 bits of large immediate using lui instruction. They can then be used in 2RI instructions giving the least significant 3 bits as the immediate. This special register belongs to immediate generator.

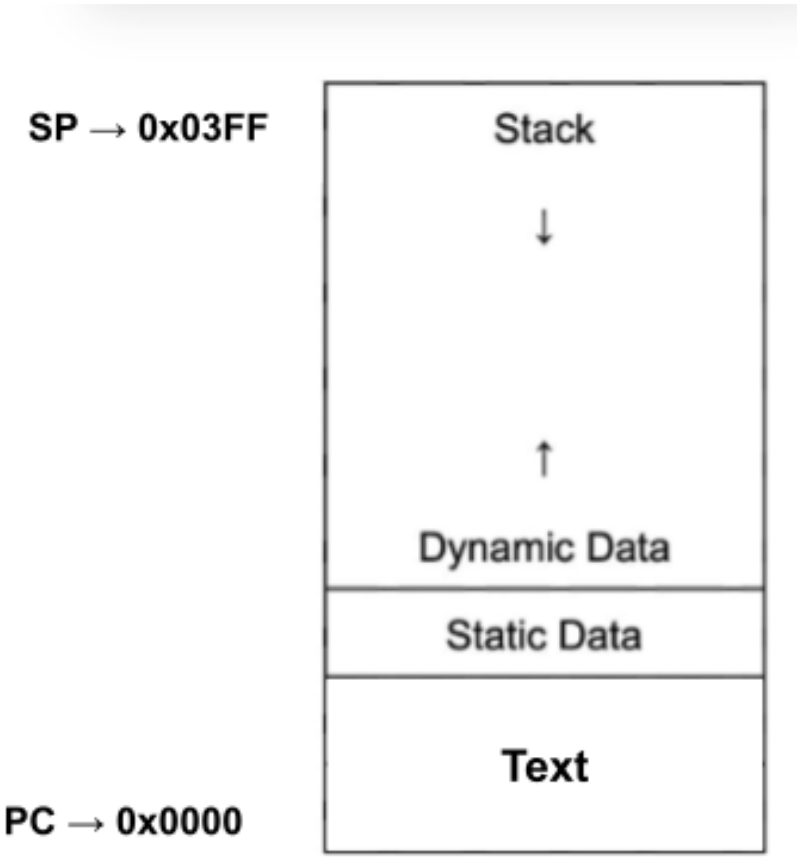
Control System

Control Status Diagram



Memory

Memory Allocation



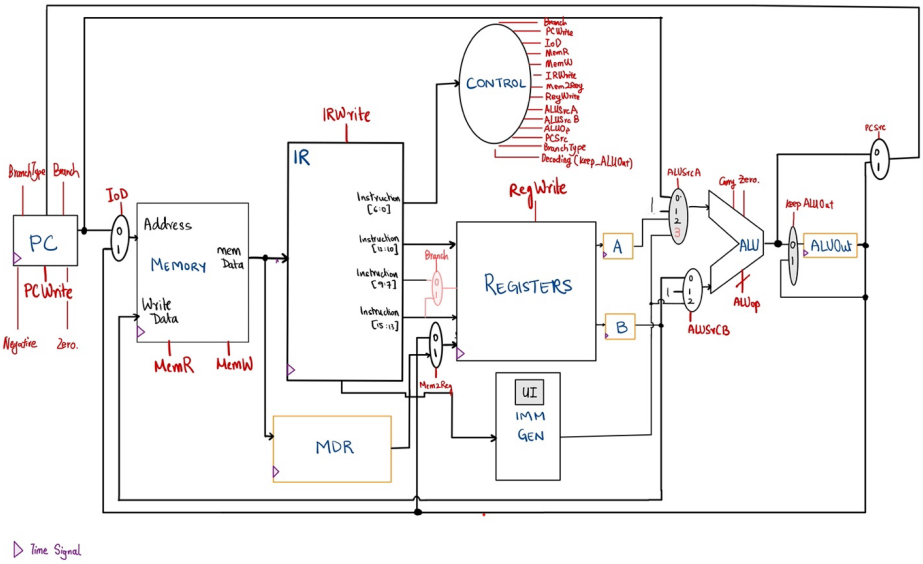
Memory Location for I/O

Commented [YW1]: @Pearcy, Luke

The memory address 0xFC00 is reserved for input and output. Writing to this address outputs data and reading from it inputs data.

Register-Transfer Level (RTL)

Datapath



Note that control signals in this diagram are notated in red. All red wires are assumed to conncect to control. Simple registers are outlined in yellow. The clock signal input for components that require it is indicated by a purple triangle

Naming Convention

The naming convention for RTL (Register-Transfer Level) components follows a systematic pattern to ensure clear and consistent identification of signals. This pattern involves specifying three key aspects for each signal: its type, direction, and name.

Direction

The signal name includes information about the direction of each signal, differentiating between inputs and outputs. Inputs that receive external data are denoted by the prefix $input\_$, while outputs that represent the results or data produced by the component are denoted by the prefix $output\_$.

Component

The signal name includes information about component. For instance, signals associated with Arithmetic Logic Unit (ALU) operations are labeled with prefixes such as $input\_ALU\_$ or $output\_ALU\_$, indicating their connection to the ALU functionality.

Purpose

The signal name includes information about the purpose. This extra detail enhances understanding of each signal's role and functionality within the RTL component. For instance, the first ALU input is represented by $input\_ALU\_A$.

By following this systematic pattern, the naming convention not only improves readability but also provides crucial information about the role and characteristics of each signal in the RTL component. This approach promotes maintainability and ease of understanding for developers working with the hardware description of the component.

RTL Summary

Commented [YW2]: This also needs to be updated @Pearcy, Luke

Components	Inputs	Outputs	Multicycle RTL Symbols
ALU	$input\_ALU\_A[15:0]$ $input\_ALU\_B[15:0]$ $input\_ALU\_ALUOp[2:0]$	$output\_ALU[15:0]$ $output\_ALU\_Zero[0:0]$ $output\_ALU\_negative[0:0]$	$ALU(operation, input\_A, input\_B)$
Memory	$input\_mem\_write[0:0]$ $input\_mem\_addr[15:0]$ $input\_mem\_data[15:0]$ $CLK[0:0]$	$output\_mem\_data[15:0]$	$mem[address]$
Immediate Generator	$input\_imm[15:0]$ $CLK[0:0]$	$output\_imm[15:0]$	$imm[15:0]$ $UI[12:0]$
Control	$input\_control[6:0]$	$output\_control\_branch[0:0]$ $output\_control\_memRead[0:0]$ $output\_control\_ALUOp[3:0]$ $output\_control\_memWrite[0:0]$ $output\_control\_ALUSrc[0:0]$ $output\_control\_regWrite[0:0]$ $output\_control\_branchType[1:0]$ $output\_control\_keepALUOut[0:0]$	None (not used in RTL)
Program Counter	$input\_PC\_PCWrite$ $input\_PC\_newPC$ $input\_PC\_Zero[0:0]$ $input\_PC\_Neg[0:0]$ $input\_PC\_BranchType[1:0]$ $CLK[0:0]$	$output\_PC[15:0]$	$PC[15:0]$
Programmable Register File	$input\_reg\_readA\_address[2:0]$ $input\_reg\_readB\_address[2:0]$ $input\_reg\_write[0:0]$ $input\_reg\_write\_value[15:0]$ $input\_reg\_write\_address[2:0]$ $CLK[0:0]$	$output\_reg\_A[15:0]$ $output\_reg\_B[15:0]$	$Reg[index]$
Instruction Register	$input\_IR\_Instru[15:0]$ $input\_IR\_write[0:0]$ $CLK[0:0]$	$output\_IR\_Control[6:0]$ $output\_IR\_RegA[2:0]$ $output\_IR\_RegB[2:0]$ $output\_IR\_RegD[2:0]$ $output\_IR\_Imm[15:0]$	$IR[end:start]$ $IR = mem[PC]$
Simple Register	$input\_SR[15:0]$ $CLK[0:0]$	$output\_SR[15:0]$	A B MDR $ALUOut$

Arithmetic Logic Unit (ALU)

Implementation File:
rhit-csse232-2324b-project-lime-2324b-02/implementation/ALU.v

Inputs:

- **input_ALU_A[15:0]**: 16-bit input representing the first operand for the ALU.
- **input_ALU_B[15:0]**: 16-bit input representing the second operand for the ALU.
- **input_ALU_ALUOp[2:0]**: 3-bit input specifying the ALU operation code.

Outputs:

- **output_ALU[15:0]**: 16-bit output representing the result of the ALU operation.
- **output_ALU_Zero[0:0]**: Single-bit output indicating whether the ALU result is zero (0) or not.
- **output_ALU_negative[0:0]**: Single-bit output indicating whether the ALU result is negative or not.

Behavior:

- The ALU performs operations based on **input_ALUOp**, including addition, subtraction, bit shifts, logical AND/OR/XOR, multiplication, and special calculation.
- Output flags indicate if the result is zero or negative.

Multicycle RTL Symbols:

- **ALU(operation, input_A, input_B)**: Output of ALU with given operation and inputs.

Testing:

- Test all operations with arbitrary inputs to ensure correct behavior.
- Test all operations with edge cases (max, min, and zero values).
- Test flag functionality during subtraction operation.
- Ensure all operations complete within the cycle before the greatest possible delay.

Memory

Implementation File:
rhit-csse232-2324b-project-lime-2324b-02/implementation/memory_component.v

Inputs:

- **input_mem_write[0:0]**: Single-bit input indicating when a write instruction is enabled.
- **input_mem_addr[15:0]**: 16-bit input representing the memory address for read or write operations.
- **input_mem_data[15:0]**: 16-bit input representing the data to be written into the memory when a write instruction is enabled.
- **CLK[0:0]**: Clock signal.

Outputs:

- **output_mem_data[15:0]**: 16-bit data from the memory at the specified address.

Behavior:

- On the rising edge of the clock (CLK), reads 16-bit data at the memory address specified by **input_mem_addr** and outputs it to **output_mem_data**.
- If **input_mem_write** is 1, writes data from **input_mem_data** to the address specified by **input_mem_addr**.
- A special address 0xFC00 is reserved for input/output, reading from this address will input data and writing to it will output data

Multicycle RTL Symbols:

- **mem[address]**: the 16 bit value at this memory address

Testing:

- Test read and write operations for arbitrary memory addresses with arbitrary data.
- Ensure proper timing on read and write operations (should take one cycle).

Immediate Generator (IG)

Implementation File:
rhit-csse232-2324b-project-lime-2324b-02/implementation/ImmediateGenerator.v

Inputs:

- **input_imm[15:0]**: 16-bit input of the instruction for the immediate to be parsed.
- **CLK[0:0]**: Clock signal.

Outputs:

- **output_imm[15:0]**: 16-bit output representing the immediate value generated by the Immediate Generator.

Behavior:

- Generates a 16-bit immediate value based on the opcode.
- For 2RI instructions, appends a 3-bit immediate in the instruction to the upper 13 bits of the upper immediate register.
- For lui instructions, stores the 13-bit immediate value in the instruction in its upper immediate register for later use.
- For RI instructions, sign extends immediate in instruction to 16-bit immediate
- For UJ instructions, sign extends immediate in instruction to 16-bit immediate

Multicycle RTL Symbols:

- IG[15:0]: the output of immediate generator (the 16 bit calculated immediate)
- UI[12:0]: the value stored in the UI register that is used for immediate in 2RI instruction types

Testing:

- Test for proper interpretation of instructions for all instruction formats.
- Validate correct immediate concatenation for RI types.
- Ensure proper sign extension when applicable.
- Verify proper timing, ready for ALU and PC applications in the same cycle the instruction is available.
- Ensure lui instruction takes only two cycles.

Control

Implementation File:
rhit-csse232-2324b-project-lime-2324b-02/implementation/Control.v

This module implements the control logic for lime processor, generating control signals based on the current state and input control signals. It defines various operational states such as FETCH, DECODE, RTYPE, RITYPE, RTYPEEND, LW1, LW2, SW, JALR, BRANCH, BRANCH2, and JAL. Using sequential logic, it transitions between these states and assigns output control signals to manage the processor's operation. Combinational logic calculates the ALU operation (ALUOp) and the sources for the ALU (ALUSrcA and ALUSrcB) based on the input control signals. The module interfaces with both the processor's datapath and the clock system, responding to the clock signal (CLK) and a reset signal (Reset) to ensure timely and correct execution of instructions.

Inputs:

- **input_control [6:0]:** 7-bit input control signals for instruction decoding and control flow management.
- **CLK:** Clock signal for synchronizing the control logic operations.

Outputs:

- $\boldsymbol{output\_control\_ALUOp}$ $[\mathbf{3}:\mathbf{0}]$: Control signal defining the operation to be performed by the ALU.
- $\boldsymbol{output\_control\_ALUSrcA}$ $[\mathbf{1}:\mathbf{0}]$: Control signal for selecting the source A for the ALU.
- $\boldsymbol{output\_control\_ALUSrcB}$ $[\mathbf{1}:\mathbf{0}]$: Control signal for selecting the source B for the ALU.
- $\boldsymbol{output\_control\_Branch}$ $[\mathbf{0}:\mathbf{0}]$: Control signal for branching decisions.
- $\boldsymbol{output\_control\_BranchType}$ $[\mathbf{1}:\mathbf{0}]$: Control signal indicating the type of branch operation.
- $\boldsymbol{output\_control\_IoD}$ $[\mathbf{0}:\mathbf{0}]$: Control signal for selecting between instruction and data for memory operations.
- $\boldsymbol{output\_control\_IRWrite}$ $[\mathbf{0}:\mathbf{0}]$: Control signal for enabling writing to the instruction register.
- $\boldsymbol{output\_control\_Mem2Reg}$ $[\mathbf{0}:\mathbf{0}]$: Control signal for selecting between memory and ALU results for writing back to the register.
- $\boldsymbol{output\_control\_MemR}$ $[\mathbf{0}:\mathbf{0}]$: Memory read control signal.
- $\boldsymbol{output\_control\_MemW}$ $[\mathbf{0}:\mathbf{0}]$: Memory write control signal.

- $\boldsymbol{output\_control\_PCSrc}$ $[\mathbf{0}:\mathbf{0}]$: Control signal for selecting the source for the Program Counter (PC) update.
- $\boldsymbol{output\_control\_PCWrite}$ $[\mathbf{0}:\mathbf{0}]$: Control signal for enabling writing to the PC.
- $\boldsymbol{output\_control\_RegWrite}$ $[\mathbf{0}:\mathbf{0}]$: Control signal for enabling writing to the register file.
- $\boldsymbol{output\_control\_keepALUOut}$ $[\mathbf{0}:\mathbf{0}]$: Control signal for making ALUOut read its output wire as input

Behavior:

- Interprets the 3 opcode bits of the instruction to know what bits are what.
- If necessary, reads func4 bits to determine the appropriate ALU operation and sends this to the ALU.

Multicycle RTL Symbols:

- None (not used in RTL).

Testing:

- Test for proper interpretation of opcode and func4.
- Validate proper control signal output for every instruction.
- Ensure proper timing (control signals should be available shortly after the instruction is available).

Testing Method:

- Test the general case in each instruction type.
- Test the special instruction.
- The tests will go through all states the instruction needed.

Program Counter

Implementation File:
rhit-csse232-2324b-project-lime-2324b-02/implementation/PC.v

Inputs:

- **input_PC_PCWrite**: A flag indicating whether the Program Counter (PC) will be updated.
- **input_PC_newPC**: The new value for the Program Counter (PC).
- **input_ zero**: A signal to the ALU to tell PC if the result of the subtraction comparison for a branch was zero
- **input_negative**: A signal to the ALU to tell PC if the result of the subtraction comparison for a branch was negative
- **input_branchType[1:0]**: A control signal to tell PC what kind of branch is being executed
- **input_PC_isbranch**: A control signal to tell PC if a branch instruction is being executed
- **CLK[0:0]**: The clock signal.

Outputs:

- **output_PC[15:0]**: The value of the Program Counter, representing the address of the instruction.

Behavior:

- On the falling edge of the clock (CLK), if the **input_PC_PCWrite** flag is asserted, the PC undergoes an update.
- The PC's value is set to the new value specified by **input_PC_newPC**.
- If the **input_PC_PCWrite** flag is not asserted, the Program Counter (PC) remains unchanged, maintaining its current value.

Multicycle RTL Symbols:

- **PC[15:0]**: The instruction address value held in PC.

Testing:

- Test for proper incrementing of the Program Counter.
- Validate correct jumping/branching behavior.
- Ensure proper PC output.

- Test for edge cases (max/min values) to verify robustness.
- Validate that the PC is ready by the end of the last cycle for all instructions.

Programmable Register File

Implementation File:
rhit-csse232-2324b-project-lime-2324b-02/ProgrammableRegisterFile.v

Inputs:

- **input_reg_readA_address[2:0]**: 3-bit address specifying the register location for reading Operand A.
- **input_reg_readB_address[2:0]**: 3-bit address specifying the register location for reading Operand B.
- **input_reg_write[0:0]**: Single-bit signal to enable the write operation to the register.
- **input_reg_write_value[15:0]**: 16-bit value to be written into the register when a write operation is enabled.
- **input_reg_write_address[2:0]**: 3-bit address specifying the register location for writing data when **reg_write** is enabled.
- **CLK[0:0]**: Single-bit clock signal used to synchronize read and write operations in the programmable register.

Outputs:

- **output_reg_A[15:0]**: 16-bit output representing the data read from Register A.
- **output_reg_B[15:0]**: 16-bit output representing the data read from Register B.

Behavior:

- On the rising edge of the clock (CLK), if the **reg_set[0:0]** input is 1, the register corresponding to the value of **register_id[0:2]** is set to the **input_value[0:15]**.
- The **output_value[15:0]** is set to the value of the register corresponding to **register_id[0:2]**.

Multicycle RTL Symbols:

- **Reg[index]**: The value of the register at this index.

Testing:

- Test for writing arbitrary values to every register.
- Validate reading values from every register.
- Ensure proper timing on all registers for read and write (outputs should be ready in time for ALU to finish in that cycle).

Instruction Register

Implementation File:
rhit-csse232-2324b-project-lime-2324b-02/implementation/InstructionRegister.v

Inputs:

- **input_IR_Instru[15:0]**: 16-bit input bus for storing the instruction data.
- **input_IR_write[0:0]**: Single-bit control signal indicating whether to write data into the instruction register.
- **CLK[0:0]**: Clock signal.

Outputs:

- **output_IR_control[6:0]**: 7-bit output signifying the opcode and func4 of the current instruction fetched from the Instruction Register (IR) [6:0]. Sent to the control module.
- **output_IR_regA[3:0]**: 3-bit output sourced from the Instruction Register (IR) [12:10], utilized in 3R and 2RI types to determine the address of register r1. Sent to the Programmable Register File.
- **output_IR_regB[3:0]**: 3-bit output obtained from the Instruction Register (IR) [9:7], employed in 3R types to specify the address of register r2. Sent to the Programmable Register File.
- **output_IR_regD[3:0]**: 3-bit output, extracted from the Instruction Register (IR) [12:10], used in 3R, 2RI, UJ, and RI types to identify the address of register rd. Sent to the Programmable Register File.
- **output_IR_imm[15:0]**: 16-bit output derived from the complete instruction in the Instruction Register (IR) [15:0]. Sent to the Immediate Generator.

Behavior:

- On the rising edge of the clock (CLK), receives a 16-bit instruction through the **input[15:0]** wire if **input_ir_write[0:0]** equals 1.
- Identifies addresses for A and B and D, putting them to **output_IR_reg_A** , **output_IR_reg_B**, and **output_IR_regD** respectively.
- Sets the destination register address to **output_reg_dest**.

Multicycle RTL Symbols:

- **IR[end:start]**: The respective bits of the instruction.
- **IR=mem[PC]**: Fetching the instruction at the address indicated by PC.

Testing:

- Test for proper fetching of the instruction.
- Validate proper splitting of the instruction and sending it out.
- Ensure proper timing (instruction should be ready after the first cycle for all instructions).

Simple Register

Implementation File:
rhit-csse232-2324b-project-lime-2324b-02/implementation/SimpleRegister.v

Inputs:

- **input_SR[15:0]**: Input that updates every cycle.
- **CLK[0:0]**: Single-bit clock signal used to synchronize read and write operations in the register.

Outputs:

- **output_SR[15:0]**: Outputs the value in the register.

Behavior:

- On the falling edge of the clock (CLK), the register is set to the **input[15:0]**.
- The output is always the value currently in the register.

Multicycle RTL Symbols:

- **A**: Register storing the value for the next clock cycle, serving as input A for the ALU.
- **B**: Register storing the value for the next clock cycle, serving as input B for the ALU.
- **MDR**: Memory Data Register, storing data fetched from memory.
- **ALUOut**: Output of the Arithmetic Logic Unit (ALU).

Testing:

- Test for reading of input.
- Validate outputting value.
- Ensure proper timing.

Testing Strategy

Component Testing

For component testing we plan to test a series of arbitrary values (including negative and positive numbers when appropriate) as well as edge cases (max, min, and zero values) for all value inputs with every combination of control signals that would be expected to occur in our processor to ensure the outputs and behavior of the components match our expectations. We also measure the time this takes and ensure it can finish fast enough for our RTL to work.

Components	Testing
ALU	Test all operations with some arbitrary inputs to ensure t behaves as expected Test all operations with edge cases (max, min, and zero values) Test for proper flag functionality when using the subtract operation Test to ensure all operations can be completed before cycled ends after greatest possible delay for receiving inputs
Memory	Test read and write for arbitrary memory addresses with arbitrary data Test for proper timing on read and write (should take one cycle)
Immediate Generator	Test for proper instruction interpretation for all instruction formats Test for proper immediate concatenation for ri types Test for proper sign extension when applicable Test for proper timing (should be ready for ALU and PC applications in same cycle instruction is availible) lui instruction should take only one cycle
Control	Test for proper interpretation of opcode and func4 Test for proper control signal output for every instruction Test for proper timing (control signals should be available shortly after instruction is available)
Program Counter	Test for proper incrementing Test for proper jumping/branching Test for proper PC output Test for edge cases(max/min values)

	Test that pc is ready by end of last cycle for all instructions
Programmable Register File	Test for writing arbitrary values to every register Test for reading values from every register Test for proper timing on all registers for read and write (outputs should be ready in time for alu to finish in that cycle)
Instruction Register	Test for proper fetching of instruction Test for proper splitting of instruction and sending it out Test for proper timing (instruction should be ready after the first cycle for all instructions)
Simple Register	Test for reading of input Test for outputting value Test for proper timing

Integration Plan and Testing

We will also make tests for small groups of components that are wired together to make sure the smaller subsystems behave as expected.

Component Subsystems	Components Included	Tests
ALU and related registers	ALU, simple registers A,B, and ALUout	Test all operations with some arbitrary inputs to ensure it behaves as expected Test all operations with edge cases (max, min, and zero values) Test for proper flag functionality when using the subtract operation Test for proper timing on all operations Test that registers have the values we expect when we expect them
PC, Instruction Register, and Memory	PC, Instruction Register, and Memory	Test that the Instruction Register can fetch the instruction at

		address indicated by PC in the Memory and split it up properly Test that the instruction is ready within the proper time frame
ALU and related registers with Immediate Generator and Programable Registers	ALU, Immediate Generator, Programable Registers, and simple registers A,B, and ALUout	Test that immediate are handled as expected and used properly in operations Test that values in registers are fetched and used properly in the ALU Test that values from the ALU are properly put into registers Test that all of this happens with the timing we expect

System Testing

Finally, we will test the entire processor when it is put together by writing tests for every instruction. Each of these instruction tests will test a variety of arbitrary values as well as edge cases for immediate when applicable. They will also use different registers and ensure the registers used have the expected values in them after instruction execution. We will also be sure that instructions still behave consistent with their descriptions when many of the registers selected are the same register. We will also make sure the instructions do not take more time than the allotted cycles and that they can operate properly regardless of the state the processor is in when the instruction is run (values in simple registers or still on wires etc.).

Instructions

Core Instruction Formats

All our instructions are 16 bits.
All memory addresses are 16 bits.
All 8 programmable registers have 16-bit values and 3-bit identifiers.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Type Name
rd			r1			r2			func 4				opcode			3R type
immediate [12 : 0]													opcode			L type with a special register
rd			r1			immediate [2 : 0]			func 4				opcode			2RI type
rd			immediate [9 : 0]										opcode			UJ type
rd			immediate [5 : 0]						func 4				opcode			RI type

Multiple Cycle Instruction RTL Cycle Summary

Cycles	$3R$	RI	$2RI$	lw/sw
1 Fetch	$PC = ALU(+, PC, 1)$ $IR = Mem([PC])$ $ALUOut = ALU(+, PC, 1)$			
2 Decode	$A = Reg[IR[12:10]]$ $B = Reg[IR[9:7]]$ $ALUOut = ALUOut$			
3	$ALUOut = ALU(ALUOp, A, B)$	$ALUOut = ALU(ALUOp, A, IG([9:7]))$		$ALUOut = ALU(+, A, IG([9:7]))$
4	4) $Reg[IR[15:13]] = ALUOut$			$lw: MDR = Mem[ALUOut]$ $sw: Mem[ALUOut] = Reg(IR[15:13])$
5				$lw: Reg(IR[15:13]) = MDR$

Cycles	$jalr$	$Branch$	jal	lui
1 Fetch	$PC = ALU(+, PC, 1)$ $IR = Mem([PC])$ $ALUOut = ALU(+, PC, 1)$			
2 Decode	$A = Reg[IR[12:10]]$ $B = Reg[IR[9:7]]$ $ALUOut = ALUOut$			
3	$Reg(IR[15:13])$ $= ALUOut$ $PC = ALU(IG([9:7]))$	$A = Reg[IR[12:10]]$ $B = Reg[IR[15:13]]$ $ALUOut = ALU(+, PC, IG([9:7]))$	$A = Reg[IR[12:10]]$ $B = Reg[IR[15:13]]$ $ALUOut$ $= ALU(+, PC, IG([9:7]))$	$*$
4		$If\ (A\ branchType\ B)$ $PC = ALUOut$		

∗lui will be finished after decode cycle

Table of Instructions

3R Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Type Name
rd			r1			r2			func 4				opcode			3R type

Instruction	Name	Instruction Type	Func. 4	Opcode	Description
add	add	3R	0000	000	R[rd] = R[r1] + R[r2]
sub	subtract	3R	0001	000	R[rd] = R[r1] - R[r2]
and	and	3R	0010	000	R[rd] = R[r1] & R[r2]
or	or	3R	0011	000	R[rd]= R[r1] | R[r2]
xor	xor	3R	0100	000	R[rd] = R[r1] ^ R[r2]
sll	shift left logical	3R	0101	000	R[rd]= R[r1] << R[r2]
srl	shift right logical	3R	0110	000	R[rd]= R[r1] >> R[r2]
sla	shift left arithmetic	3R	0111	000	R[rd]= R[r1] << R[r2]
sra	shift right arithmetic	3R	1000	000	R[rd]= R[r1] >> R[r2]

2RI Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Type Name
rd			$r1$			immediate [2 : 0]			$func\ 4$				opcode			2RI type

Instruction	Name	Instruction Type	func 4	Opcode	Description
$addi$	ADD Immediate	2RI	0000	001	$R[rd] = R[r1] + IG(UI, imm)$
$subi$	SUB Immediate	2RI	0001	001	$R[rd] = R[r1] - IG(UI, imm)$
$andi$	AND Immediate	2RI	0010	001	$R[rd] = R[r1] \,\&\, IG(UI, imm)$
ori	OR Immediate	2RI	0011	001	$R[rd] = R[r1] \mid IG(UI, imm)$
$xori$	XOR Immediate	2RI	0100	001	$R[rd] = R[r1] \hat{} IG(UI, imm)$
$slli$	Shift Left Logical Imm	2RI	0101	001	$R[rd] = R[r1] << IG(UI, imm)$
$srli$	Shift Right Logical Imm	2RI	0110	001	$R[rd] = R[r1] >> IG(UI, imm)$
$slai$	Shift Left Arith Imm	2RI	0111	001	$R[rd] = R[r1] << IG(UI, imm)$
$srai$	Shift Right Arith Imm	2RI	1000	001	$R[rd] = R[r1] >> IG(UI, imm)$
lw	Load Word	2RI	1001	001	$R[rd] = M[2 * (R[r1] + IG(UI, imm))]$
sw	Store Word	2RI	1010	001	$M[2 * (R[r1] + IG(UI, imm))] = R[rd]$
$jalr$	Jump And Link Register	2RI	1011	001	$R[rd] = PC + 1$ $PC = IG(UI, imm)$
beq	Branch if equal	2RI	1100	001	$if(R[rd] == R[r1])\ PC += IG(UI, imm)$
blt	Branch if less than	2RI	1101	001	$if(R[rd] < R[r1])\ PC += IG(UI, imm)$
bne	Branch if not equal	2RI	1110	001	$if(R[rd] != R[r1])\ PC += IG(UI, imm)$
bge	Branch if greather or equal	2RI	1111	001	$if(R[rd] >= R[r1])\ PC += IG(UI, imm)$

RI Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Type Name
rd			immediate [5 : 0]						func 4				opcode			RI type

Instruction	Name	Instruction Type	func 4	Opcode	Description	Note
inc	Increment Immediate	RI	0000	010	$R[rd] = R[rd] + IG(imm)$	sign extends
dec	Decrement Immediate	RI	0001	010	$R[rd] = R[rd] - IG(imm)$	sign extends
$andd$	and in place	RI	0010	010	$R[rd] = R[rd] \,\&\, IG(imm)$	sign extends
orr	or in place	RI	0011	010	$R[rd] = R[rd] \mid IG(imm)$	sign extends
$xorr$	xor in place	RI	0100	010	$R[rd] = R[rd] \,\hat{}\, IG(imm)$	sign extends
$slipl$	Shift Left In Place logical	RI	0101	010	$R[rd] = R[rd] << IG(imm)$	sign extends
$sripl$	Shift Right In Place logical	RI	0110	010	$R[rd] = R[rd] >> IG(imm)$	sign extends
$slipa$	Shift Left In Place arithmetic	RI	0111	010	$R[rd] = R[rd] << IG(imm)$	sign extends
$sripa$	Shift right In Place arithmetic	RI	1000	010	$R[rd] = R[rd] >> IG(imm)$	sign extends
set	Set	RI	1100	010	$R[rd] = IG(imm)$	sign extends

L Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Type Name
immediate [12 : 0]													opcode			L type with a special register

Instruction	Name	Instr Type	func4	Opcode	Description	Note
lui	load Upper Immediate	L	-	011	UI = immediate[12: 0]	sets non-programable UI (upper immediate) register to be used with 2RI instructions

UJ Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Type Name
rd			immediate [9 : 0]										opcode			UJ type

Instruction	Name	Instr Type	func 4	Opcode	Description	Note
jal	Jump and Link	UJ	-	100	$R[rd] = PC + 1$ $PC = imm[9:0]$	moves execution to the instruction at address 2*(immediate 9:0)

Coding Examples

while loop

C Code

```c
int main () {
   int a = 5;

   while( a < 20 ) {
      a++;
   }

   return a;
}
```

Assembly Code

Address	Assembly	Machine Code	Comment
	main:		
0x0000	addi a0, a0, 5	110 110 101 0000 001	// a = 5
0x0002	addi t0, t0, 7	011 011 111 0000 001	// t0 = 7
	loop:		
0x0004	bge a0, t0, loop_end	110 011 [011] 1110 001	// if (a < 7)
0x0006	addi a0, a0, 1	110 110 001 0000 001	// a + +
0x0008	jal t1, loop	100 [0 0000 0010]100	
	loop_end:		
0x000A	jal t1, 0(ra)	100 000 000 1010 001	// return a

Array Access

C code:

```c
int main(){
    int[] array = int[20];
    for(int i=0; i < array.length; i++){
        array[i]=array[i] * 2
    }
}
```

Assembly Code:

```
main:
        lui [array address]
loop:
        bgt t0
set t0, 0                   //t0 is i
        lw t1, 0(t0)
        inc t1                      //inciment t1
        jal a0, loop
```

Recursion

C code:

```c
int simpleRecursion(int n) {
        if (n == 0) {
                return 1;
        }

        else {
                return simpleRecursion(n - 1);
        }
}
```

Assembly code

Address	Assembly	Machine Code	Comment
	simpleRecursion:		
0x0000	subi sp, sp, 2	001 001 010 0100 001	
0x0002	sw ra, 0(sp)	000 001 000 1001 001	
	// Base case: if n == 0, return 1		
0x0004	addi t0, t0, 0	110 110 000 0000 001	
0x0006	beq a0, t0, base_case	110 011 [101] 1011 001	
	// Recursive case: return simpleRecursion(n - 1)		
0x0008	subi a0, a0, 1	101 101 001 0100 001	
0x000A	jal ra, simpleRecursion	000 [0 0000 0101] 100	
0x000C	lw a0, 0(sp)	000 110 000 1000 001	
0x000E	jal ra, end_recursion	000 [0 0000 0010] 100	
	base_case:		
0x0010	addi a0, a0, 1 // Return 1 for the base case	110 110 001 0000 001	

	end_recursion:		
0x0012	lw ra, 0(sp)	001 001 000 1000 001	
0x0014	add sp, sp, 2	001 001 010 0000 001	
0x0016	jalr t1, 0(ra)	100 000 000 1010 001	

relPrime and Euclid's Algorithm

C Code

```c
// Find m that is relatively prime to n.
int relPrime(int n)
{
   int m;

   m = 2;

   while (gcd(n, m) != 1) {  // n is the input from the outside world
     m = m + 1;
   }

   return m;
}

// The following method determines the Greatest Common Divisor of a and b
// using Euclid's algorithm.
int gcd(int a, int b)
{
  if (a == 0) {
    return b;
  }

  while (b != 0) {
    if (a > b) {
      a = a - b;
    } else {
      b = b - a;
    }
  }

  return a;
}
```

Assembly Code

Address	**Assembly**	**Machine Code** (spaces and brackets for readability)
0x0000	set t0, 0	011 000000 1100 010
0x0001	lui (1111110000000) -128	1111110000000 011
0x0002	lw a0, 0(t0)	110 011 000 1001 001
0x0003	**relPrime:** lui 0	0000000000000 011
0x0004	dec sp 4	0010001000001010
0x0005	sw ra, 0(sp)	000 001 000 1010 001
0x0006	sw a0, 1(sp)	110 001 001 1010 001
0x0007	set a1, 2	111 000010 1100 010
0x0008	sw a1, 2(sp)	111 001 010 1010 001
0x0009	**loop:** jal ra, gcd	000 [00 0001 1001] 100
0x000A	set t2, 1	101 000001 1100 010
0x000B	beq a0, t2, exit_loop	110 101 [101] 1100 001
0x000C	lw a0, 1(sp)	110 001 001 1001 001
0x000D	lw a1, 2(sp)	111 001 010 1001 001
0x000E	addi a1, a1, 1	111 111 001 0000 001
0x000F	sw a1, 2(sp)	111 001 010 1010 001
0x0010	jal t0, loop	011 [00 0000 1001] 100
0x0011	**exit_loop:** lw a1, 2(sp)	111 001 010 1001 001
0x0012	addi a0, a1, 0	110 111 000 0000 001
0X0013	lw ra, 0(sp)	000 001 000 1001 001
0x0014	lui (1111110000000) -128	1111110000000 011
0x0015	inc sp, sp, 4	0010001000000010
0x0016	set ra, 0	000 000000 1100 010
0x0017	sw a0, 0(ra)	110 000 000 1010 001
0x0018	jal t1, 0x0000	100 0000000000 100

Address	Assembly	Machine Code (spaces and brackets for readability)
0x0019	**gcd:** set s0, 0	010 000000 1100 010
0x001A	sw ra, 3(sp)	000 001 011 1010 001
0x001B	bne a0, s0, gcd_loop	110 011 [010] 1110 001
0x001C	addi a0, a1, 0	110 111 000 0000 001
0x001D	jalr t0, 0(ra)	011 000 000 1011 001
0x001E	**gcd_loop:** beq a1, s0, gcd_end	111 010 [110] 1100 001
0x001F	beq a0, a1, goto 0x001F	110 111 [001] 1100 001
0x0020	bge a0, a1, greater	110 111 [010] 1111 001
0x0021	sub a1, a1, a0	111 111 110 0001 000
0x0022	jal t0, gcd_loop	011 [00 0001 1110] 100
0x0023	**greater:** sub a0, a0, a1	110 110 111 0001 000
0x0024	jal t0, gcd_loop	011 [000001 1110] 100
0x0025	gcd_end: lw ra, 3(sp)	000 001 011 1001 001

Tools for Programmers

Online Assembler

The Lime architecture website can be found at: https://lime.wic.monster
This website includes an online assembler that can translate Lime processor assembly into machine code. It also has links to the github repository for the design documentation and verilog implementation of the lime processor and instruction set.

Local Python Assembler