The Lime Instruction Set Manual

Document Version 20240220

Editors: Manohar Tulsi, Pearcy Luke, Raitova Naziia, Wang Yueqiao

CSSE 232, CSSE Department, Rose-Hulman Institute of Technology

2/20/24

# Table of Contents

[Table of Contents 2](#_Toc158967635)

[Updates from Last Version 3](#_Toc158967636)

[Potential Changes in Next Version 4](#_Toc158967637)

[Introduction 5](#_Toc158967638)

[Design Philosophy 5](#_Toc158967639)

[Why Multi-Cycle? 5](#_Toc158967640)

[Performance 6](#_Toc158967641)

[Introduction 6](#_Toc158967642)

[Performance Metrics 6](#_Toc158967643)

[Registers 7](#_Toc158967644)

[Programmable Registers 7](#_Toc158967645)

[Special Registers 7](#_Toc158967646)

[Non-Programmable Registers 7](#_Toc158967647)

[Control System 8](#_Toc158967648)

[Control Status Diagram 8](#_Toc158967649)

[Memory Allocation 9](#_Toc158967650)

[Register-Transfer Level (RTL) 10](#_Toc158967651)

[Datapath 10](#_Toc158967652)

[Naming Convention 11](#_Toc158967653)

[Direction 11](#_Toc158967654)

[Component 11](#_Toc158967655)

[Purpose 11](#_Toc158967656)

[RTL Summary 12](#_Toc158967657)

[Arithmetic Logic Unit (ALU) 13](#_Toc158967658)

[Inputs: 13](#_Toc158967659)

[Outputs: 13](#_Toc158967660)

[Behavior: 13](#_Toc158967661)

[Multicycle RTL Symbols: 13](#_Toc158967662)

[Testing: 13](#_Toc158967663)

[Memory 14](#_Toc158967664)

[Inputs: 14](#_Toc158967665)

[Outputs: 14](#_Toc158967666)

[Behavior: 14](#_Toc158967667)

[Multicycle RTL Symbols: 14](#_Toc158967668)

[Testing: 14](#_Toc158967669)

[Immediate Generator (IG) 15](#_Toc158967670)

[Inputs: 15](#_Toc158967671)

[Outputs: 15](#_Toc158967672)

[Behavior: 15](#_Toc158967673)

[Multicycle RTL Symbols: 15](#_Toc158967674)

[Testing: 15](#_Toc158967675)

[Control 16](#_Toc158967676)

[Inputs: 16](#_Toc158967677)

[Outputs: 16](#_Toc158967678)

[Behavior: 16](#_Toc158967679)

[Multicycle RTL Symbols: 16](#_Toc158967680)

[Testing: 16](#_Toc158967681)

[Program Counter 17](#_Toc158967682)

[Inputs: 17](#_Toc158967683)

[Outputs: 17](#_Toc158967684)

[Behavior: 17](#_Toc158967685)

[Multicycle RTL Symbols: 17](#_Toc158967686)

[Testing: 17](#_Toc158967687)

[Programmable Register File 18](#_Toc158967688)

[Inputs: 18](#_Toc158967689)

[Outputs: 18](#_Toc158967690)

[Behavior: 18](#_Toc158967691)

[Multicycle RTL Symbols: 18](#_Toc158967692)

[Testing: 18](#_Toc158967693)

[Instruction Register 19](#_Toc158967694)

[Inputs: 19](#_Toc158967695)

[Outputs: 19](#_Toc158967696)

[Behavior: 19](#_Toc158967697)

[Multicycle RTL Symbols: 19](#_Toc158967698)

[Testing: 20](#_Toc158967699)

[Simple Register 21](#_Toc158967700)

[Inputs: 21](#_Toc158967701)

[Outputs: 21](#_Toc158967702)

[Behavior: 21](#_Toc158967703)

[Multicycle RTL Symbols: 21](#_Toc158967704)

[Testing: 21](#_Toc158967705)

[Testing Strategy 22](#_Toc158967706)

[Component Testing 22](#_Toc158967707)

[Integration Plan and Testing 23](#_Toc158967708)

[System Testing 24](#_Toc158967709)

[Instructions 25](#_Toc158967710)

[Core Instruction Formats 25](#_Toc158967711)

[Instruction RTL 26](#_Toc158967712)

[Table of Instructions 27](#_Toc158967713)

[3R Type 27](#_Toc158967714)

[2RI Type 28](#_Toc158967715)

[RI Type 30](#_Toc158967716)

[L Type 31](#_Toc158967717)

[UJ Type 32](#_Toc158967718)

[Coding Examples 33](#_Toc158967719)

[loop 33](#_Toc158967720)

[Array Access 34](#_Toc158967721)

[Recursion 35](#_Toc158967722)

[and Euclid's Algorithm 37](#_Toc158967723)

# Updates from Last Version

1. Instruction RTL Revised for jalr, Branch, jal, and lui
2. RTL Datapath

# Potential Changes in Next Version

1. Instruction func4 and opcode order
2. Memory allocation

# 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 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.

## Why Multi-Cycle?

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.

# 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.

# 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 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. |

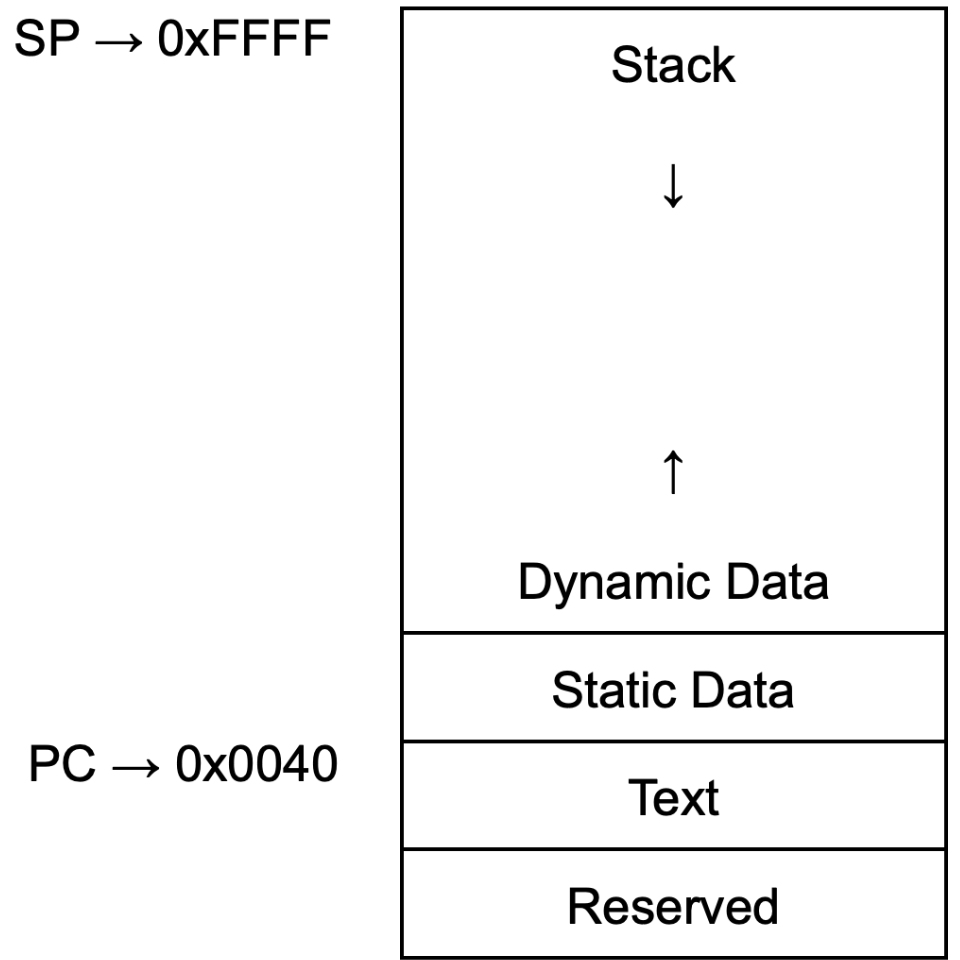
# Control System

## Control Status Diagram

A diagram of a complex structure

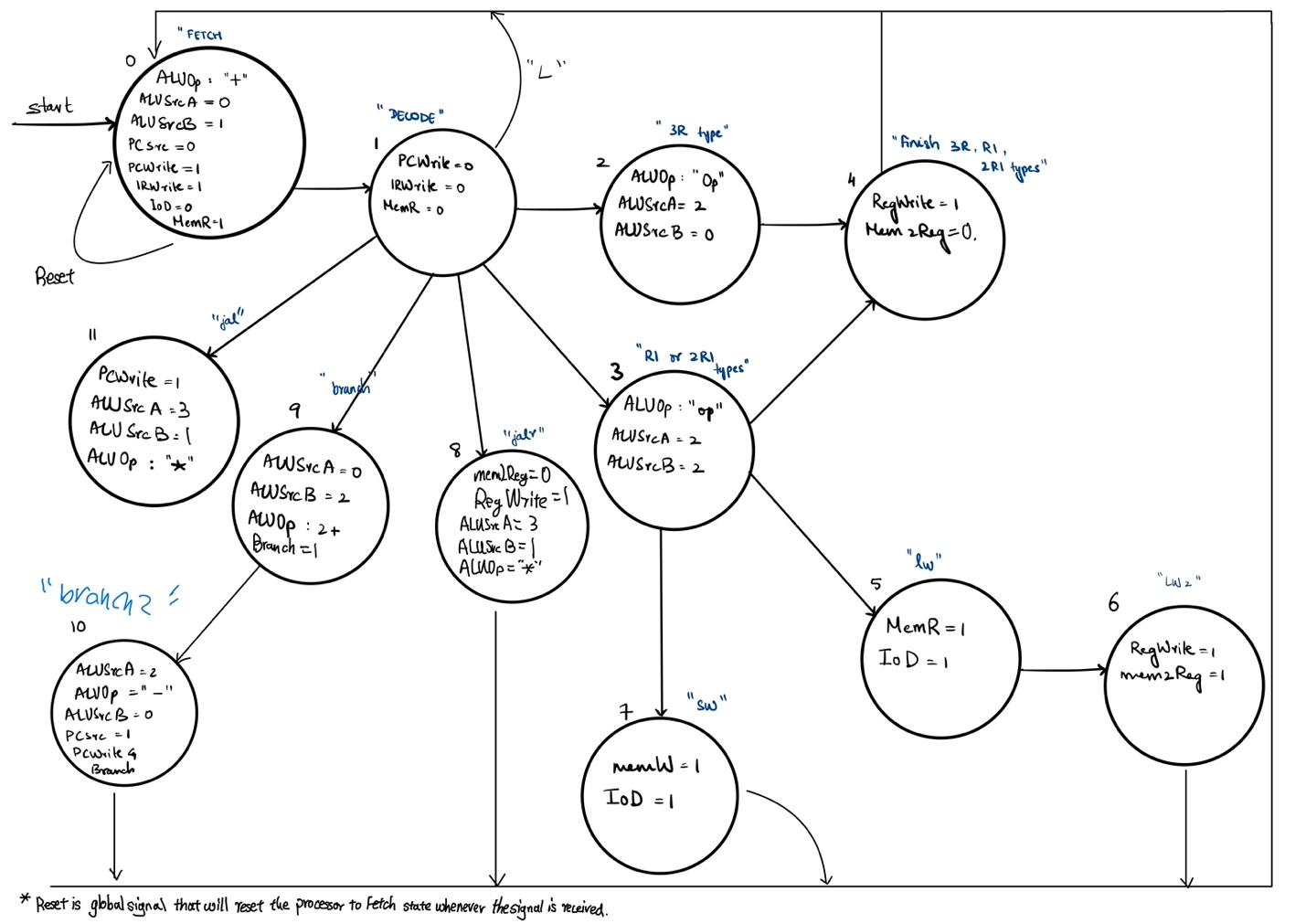
Description automatically generated with medium confidenceMemory

## Memory Allocation



# Register-Transfer Level (RTL)

## Datapath



## 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 , while outputs that represent the results or data produced by the component are denoted by the prefix .

### Component

The signal name includes information about component. For instance, signals associated with Arithmetic Logic Unit (ALU) operations are labeled with prefixes such as or , 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 .

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

|  |  |  |  |
| --- | --- | --- | --- |
| Components | Inputs | Outputs | Multicycle RTL Symbols |
| ALU |  |  |  |
| Memory |  |  |  |
| Immediate Generator |  |  |  |
| Control |  |  | None (not used in RTL) |
| Program Counter |  |  |  |
| Programmable Register File |  |  |  |
| Instruction Register |  |  |  |
| Simple Register |  |  |  |

## Arithmetic Logic Unit (ALU)

### 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

### 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**.

### 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)

### 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.

### 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 instruction takes only one cycle.

## Control

### Inputs:

* **input\_control[6:0]**: The opcode + funct4.

### Outputs:

* **output\_control\_branch[0:0]**: 1 if it's a branch instruction, 0 otherwise.
* **output\_control\_memRead[0:0]**: 1 if reading from memory, 0 otherwise.
* **output\_control\_ALUOp[2:0]**: ALU operation (+, -, and, or, ...).
* **output\_control\_memWrite[0:0]**: 1 if writing to memory, 0 otherwise.
* **output\_control\_ALUSrc[0:0]**: 1 if using immediate, 0 otherwise.
* **output\_control\_regWrite[0:0]**: 1 if the instruction involves writing to a register, 0 otherwise.
* **output\_control\_branchType[1:0]**: Specifies the type of branch instruction based on two bits (00 for beq, 01 for blt, 10 for bne, 11 for bge).

### 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).

## Program Counter

### 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).
* **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

### 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

### 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.
* Decodes the opcode and outputs it through **output\_opcode**.
* Identifies addresses for A and B, putting them to **output\_reg\_A** and **output\_reg\_B**.
* 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

### 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 |

## Instruction RTL

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Cycles |  |  |  |  |
| 1 |  | | | |
| 2 |  | | | |
| 3 |  |  | |  |
| 4 |  | | |  |
| 5 |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Cycles |  |  |  |  |
| 1 |  | | | |
| 2 |  | | | |
| 3 |  |  |  |  |
| 4 |  |  |  |  |

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 | Note |
|  | add |  |  |  |  |  |
|  | subtract |  |  |  |  |  |
|  | and |  |  |  |  |  |
|  | or |  |  |  |  |  |
|  | xor |  |  |  |  |  |
|  | shift left logical |  |  |  |  |  |
|  | shift right logical |  |  |  |  |  |
|  | shift left arithmetic |  |  |  |  | sign extends |
|  | shift right arithmetic |  |  |  |  | sign extends |

### 2RI Type

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Type Name |
|  | | |  | | | immediate | | |  | | | | opcode | | | 2RI type |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Instruction | Name | Instr Type | func 4 | Opcode | Description | Note |
|  | ADD Immediate | 2RI | 0000 | 001 |  |  |
|  | XOR Immediate | 2RI | 0001 | 001 |  |  |
|  | OR Immediate | 2RI | 0010 | 001 |  |  |
|  | AND Immediate | 2RI | 0011 | 001 |  |  |
|  | SUB Immediate | 2RI | 0100 | 001 |  |  |
|  | Shift Left Logical Imm | 2RI | 0101 | 001 |  |  |
|  | Shift Right Logical Imm | 2RI | 0110 | 001 |  |  |
|  | Shift Right Arith Imm | 2RI | 0111 | 001 |  |  |
|  | Load Word | 2RI | 1000 | 001 |  |  |
|  | Store Word | 2RI | 1001 | 001 |  | multiplied by 2 |
|  | Jump And Link Reg | 2RI | 1010 | 001 |  | multiplied by 2 |
|  | Branch if equal | 2RI | 1011 | 001 |  | PC relative and multiplied by 2 |
|  | Branch if less than | 2RI | 1100 | 001 |  | PC relative and multiplied by 2 |
|  | Branch if not equal | 2RI | 1101 | 001 |  | PC relative and multiplied by 2 |
|  | Branch if greater or equal | 2RI | 1110 | 001 |  |  |

### 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 | Instr Type | func 4 | Opcode | Description | Note |
|  | Increment Immediate | RI | 0000 | 010 |  |  |
|  | Decrement Immediate | RI | 0001 | 010 |  |  |
|  | Shift Left in Place logical | RI | 0010 | 010 |  |  |
|  | Shift Right in Place logical | RI | 0011 | 010 |  |  |
|  | Shift Left in Place arithmetic | RI | 0100 | 010 |  | Sign Extends |
|  | Shift right In Place arithmetic | RI | 0101 | 010 |  | Sign Extends |
|  | Set | RI | 0110 | 010 |  | 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 | func 4 | Opcode | Description | Note |
|  | load Upper Immediate | L | - | 011 |  | 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 |
|  | Jump and Link | UJ | - | 100 |  | moves execution to the instruction at address 2\*(immediate 9:0) |

## Coding Examples

### loop

#### C Code

1. int main () {
2. int a = 5;
4. while( a < 20 ) {
5. a++;
6. }
8. return a;
9. }

#### Assembly Code

|  |  |  |  |
| --- | --- | --- | --- |
| Address | Assembly | Machine Code | Comment |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

### Array Access

#### C code:

1. **int** main(){
2. **int**[] array = **int**[20];
3. **for**(**int** i=0; i < array.length; i++){
4. array[i]=array[i] \* 2
5. }
6. }

#### 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:

1. **int** simpleRecursion(**int** n) {
2. **if** (n == 0) {
3. **return** 1;
4. }
6. **else** {
7. **return** simpleRecursion(n - 1);
8. }
9. }

#### 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 |  |

### and Euclid's Algorithm

#### C Code

1. // Find m that is relatively prime to n.
2. int relPrime(int n)
3. {
4. int m;
6. m = 2;
8. while (gcd(n, m) != 1) {  // n is the input from the outside world
9. m = m + 1;
10. }
12. return m;
13. }
15. // The following method determines the Greatest Common Divisor of a and b
16. // using Euclid's algorithm.
17. int gcd(int a, int b)
18. {
19. if (a == 0) {
20. return b;
21. }
23. while (b != 0) {
24. if (a > b) {
25. a = a - b;
26. } else {
27. b = b - a;
28. }
29. }
31. return a;
32. }

#### Assembly Code

|  |  |  |  |
| --- | --- | --- | --- |
| Address | Assembly | Machine Code | Comment |
|  |  |  |  |
| 0x0000 |  |  | // Make Sure UI is Set to 0 |
| 0x0002 |  |  | // Increase Stack by -4 Bytes for 2 16-Bit Values |
| 0x0004 |  |  | // Save Return Address |
| 0x0006 |  |  | // Store n in the Second Part of the Stack (Note: sw Multiplies Imm by 2) |
| 0x0008 |  |  | // Set m = 2 |
| 0x000A |  |  | // Set s0 = 1 |
|  |  |  |  |
|  |  |  |  |
| 0x000C |  |  | // Jump to gcd Function, Result in a0 |
| 0x000E |  |  | // If gcd = 1, Exit the Loop |
| 0x0010 |  |  | // Else, prepare for next gcd calling, set a0 = n |
| 0x0012 |  |  | // Increment m: m = m + 1 |
| 0x0014 |  |  | // Next Iteration |
|  |  |  |  |
|  |  |  |  |
| 0x0016 |  |  | // Set Return Value to m |
| 0x0018 |  |  | // Load Return Address from Stack |
| 0x001A |  |  | // Restore Stack |
| 0x001C |  |  | // Return to Caller |
|  |  |  |  |
|  |  |  | // gcd Function Label (a=a0, b=a1) |
| 0x001E |  |  | // t0 = 0 |
| 0x0020 |  |  | // If a != 0, Skip to Loop |
| 0x0022 |  |  | // set b as a return value |
| 0x0024 |  |  | // Return b if a = 0 |
|  |  |  |  |
|  |  |  |  |
| 0x0026 |  |  | // if b == 0, end loop |
| 0x0028 |  |  | // If a > b, jump to greater |
| 0x002A |  |  | // b = b - a |
| 0x002C |  |  | // Next Iteration |
|  |  |  |  |
|  |  |  |  |
| 0x002E |  |  | // a = a - b |
| 0x0030 |  |  | // Next Iteration |
|  |  |  |  |
|  |  |  | // gcd\_end Label |
| 0x0032 |  |  | // back to caller |