The Lime Instruction Set Manual

Document Version 20240109-M1

Editors: Manohar Tulsi, Pearcy Luke, Raitova Naziia, Wang Yueqiao
CSSE 232, CSSE Department, Rose-Hulman Institute of Technology
January 9, 2024

Table of Contents

Introduction

The Lime architecture is a student-developed instruction set architecture (ISA) designed to implement higher-level algorithms, such as Euclid's GCD. It utilizes a 16-bit instruction format and 16-bit memory addresses, and includes eight programmable registers, each with 16-bit values and 3-bit identifiers, as well as additional registers like UI for handling large immediate. The instruction set includes various types, such as 3R, 2RI, UJ, and RI, which allow for arithmetic, logical, and control flow operations. The memory allocation scheme defines regions for stack, dynamic data, static data, text, and reserved memory. The architecture features instructions for performing arithmetic, logical, and control operations, as well as load and store operations. The design aims for simplicity, uniformity, and efficiency.

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

Execution Time

Measures the time taken for program completion. This metric is essential for reflecting the efficiency of the processor in executing a given workload. A shorter execution time generally indicates improved performance.

Number of Instruction

Evaluates the number of instructions executed. This metric provides insights into the efficiency of the Lime architecture by measuring the number of instructions required to complete a task. A smaller number of instructions executed suggests better overall instruction design.

Number of Clock Cycle

Quantifies the number of clock cycles required to complete the desired algorithm. This metric measures the processor's efficiency by indicating its instruction processing speed. With less Number of Clock Cycle the program use, a better design is presented.

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

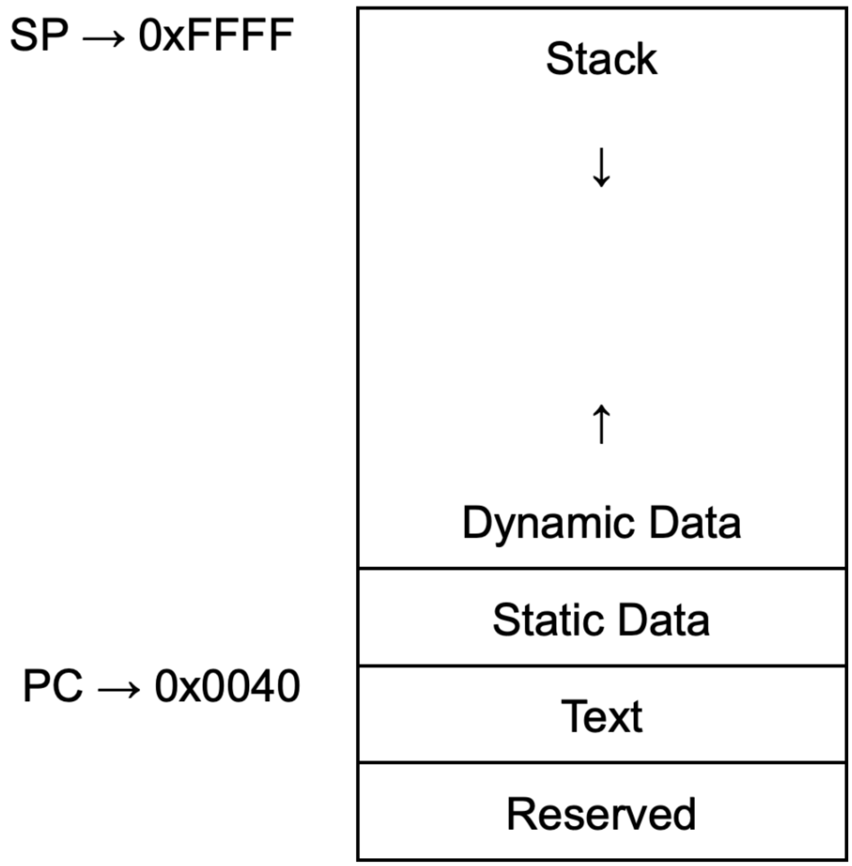
Special Registers

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 RRI instructions giving the least significant 3 bits as the immediate.

Memory

Memory Allocation



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.

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

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	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	$R[rd] = R[r1] \vert R[r2]$	
xor	xor	3R	0100	000	$R[rd] = R[r1] \,\hat{}\, R[r2]$	
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	$R[rd] = R[r1] << R[r2]$	sign extends
sra	shift right arithmetic	3R	1000	000	$R[rd] = R[r1] >> R[r2]$	sign extends

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	Instr Type	func 4	Opcode	Description	Note
addi	ADD Immediate	2RI	0000	001	$R[rd] = R[r1] + OR(UI, imm)$	
xori	XOR Immediate	2RI	0001	001	$R[rd] = R[r1] \,\hat{}\, OR(UI, imm)$	
ori	OR Immediate	2RI	0010	001	$R[rd] = R[r1] \mid OR(UI, imm)$	
andi	AND Immediate	2RI	0011	001	$R[rd] = R[r1] \,\&\, OR(UI, imm)$	
subi	SUB Immediate	2RI	0100	001	$R[rd] = R[r1] - OR(UI, imm)$	
slli	Shift Left Logical Imm	2RI	0101	001	$R[rd] = R[r1] << OR(UI, imm)$	
srli	Shift Right Logical Imm	2RI	0110	001	$R[rd] = R[r1] >> OR(UI, imm)$	
srai	Shift Right Arith Imm	2RI	0111	001	$R[rd] = R[r1] >> OR(UI, imm)$	
lw	Load Word	2RI	1000	001	$R[rd] = M[2 * (R[r1] + OR(UI, imm))]$	
sw	Store Word	2RI	1001	001	$M[2 * (R[r1] + OR(UI, imm))] = R[rd]$	multiplied by 2
jalr	Jump And Link Reg	2RI	1010	001	$R[rd] = PC + 2;$ $PC = R[rs1] + OR(UI, imm)$	multiplied by 2
beq	Branch if equal	2RI	1011	001	$if(R[rd] == R[r1])$ $PC += 2 * OR(UI, imm)$	PC relative and multiplied by 2
blt	Branch if less than	2RI	1100	001	$if(R[rd] < R[r1])$ $PC += 2 * OR(UI, imm)$	PC relative and multiplied by 2
bne	Branch if not equal	2RI	1101	001	$if(R[rd]! = R[r1])$ $PC += OR(UI, imm)$	PC relative and multiplied by 2
bge	Branch if greater or equal	2RI	1110	001	$if(R[rd] \geq R[r1])$ $PC += 2 * OR(UI, imm)$	

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	Instr Type	func 4	Opcode	Description	Note
inc	Increment Immediate	RI	0000	010	$R[rd] = R[rd] - SE(imm)$	
dec	Decrement Immediate	RI	0001	010	$R[rd] = R[rd] - SE(imm)$	
slipl	Shift Left in Place logical	RI	0010	010	$R[rd] = R[rd] << SE(imm)$	
sripl	Shift Right in Place logical	RI	0011	010	$R[rd] = R[rd] >> SE(imm)$	
slipa	Shift Left in Place arithmetic	RI	0100	010	$R[rd] = R[rd] << SE(imm)$	Sign Extends
sripa	Shift right In Place arithmetic	RI	0101	010	$R[rd] = R[rd] >> SE(imm)$	Sign Extends
set	Set	RI	0110	010	$R[rd] = SE(imm)$	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	func 4	Opcode	Description	Note
lui	load Upper Immediate	L	-	011	$\mathrm{UI} = \mathrm{immediate}[12:0]$	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	$PC = 2 * imm[9:0]$	moves execution to the instruction at address 2*(immediate 9:0)

* jal is used when programmer is calling a function.

Assembly 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

relPrime and Euclid's Algorithm

C Code

```
// 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	Comment
	relPrime:		
0x0000	lui 0	0000000000000 011	// Make Sure UI is Set to 0
0x0002	subi sp, sp, 4	001 001 100 0100 001	// Increase Stack by -4 Bytes for 2 16-Bit Values
0x0004	sw ra, 0(sp)	000 001 000 1001 001	// Save Return Address
0x0006	sw a0, 1(sp)	110 001 001 1001 001	// Store n in the Second Part of the Stack (Note: sw Multiplies Imm by 2)
0x0008	set a1, 2	111 000010 0110 010	// Set m = 2
0x000A	set s0, 1	010 000001 0110 010	// Set s0 = 1
	loop:		
0x000C	jal ra, gcd	000 [0 0000 1111] 100	// Jump to gcd Function, Result in a0
0x000E	beq a0, s0, exit_loop	110 010 [100] 1011 001	// If gcd = 1, Exit the Loop
0x0010	lw a0, 1(sp)	000 110 001 1000 001	// Else, prepare for next gcd calling, set a0 = n
0x0012	addi a1, a1, 1	111 111 001 0000 001	// Increment m: m = m + 1
0x0014	jal t0, loop	011 [000000110] 100	// Next Iteration
	exit_loop:		
0x0016	addi a0, a1, 0	110 111 000 0000 001	// Set Return Value to m
0x0018	lw ra, 0(sp)	001 001 000 1000 001	// Load Return Address from Stack
0x001A	addi sp, sp, 4	001 001 100 0000 001	// Restore Stack
0x001C	jalr t0, 0(ra)	011 000 000 1010 001	// Return to Caller
	gcd:		// gcd Function Label (a=a0, b=a1)
0x001E	set t0, 0	011 000000 0110 010	// t0 = 0
0x0020	bne a0, t0, gcd_loop	110 011 [011] 1101 001	// If a != 0, Skip to Loop
0x0022	add a0, a1, t0	110 111 011 0000 000	// set b as a return value
0x0024	jalr t0, 0(ra)	011 000 000 1010 001	// Return b if a = 0
	gcd_loop:		
0x0026	bne a1, t0, gcd_end	111 011 [110] 1101 001	// if b == 0, end loop
0x0028	bge a0, a1, greater	110 111 [011] 1110 001	// If a > b, jump to greater
0x002A	sub a1, a1, a0	111 111 110 0001 000	// b = b - a
0x002C	jal t0, gcd_loop	110 [000001101] 100	// Next Iteration
	greater:		
0x002E	sub a0, a0, a1	110 110 111 0001 000	// a = a - b

0x0030	jal t0, gcd_loop	011 [000001101] 100	// Next Iteration
	gcd_end		// gcd_end Label
0x0032	jalr t0, 0(ra)	011 000 000 1010 001	// back to caller