The Lime Instruction Set Manual

Document Version 20240115-M2

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 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.

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

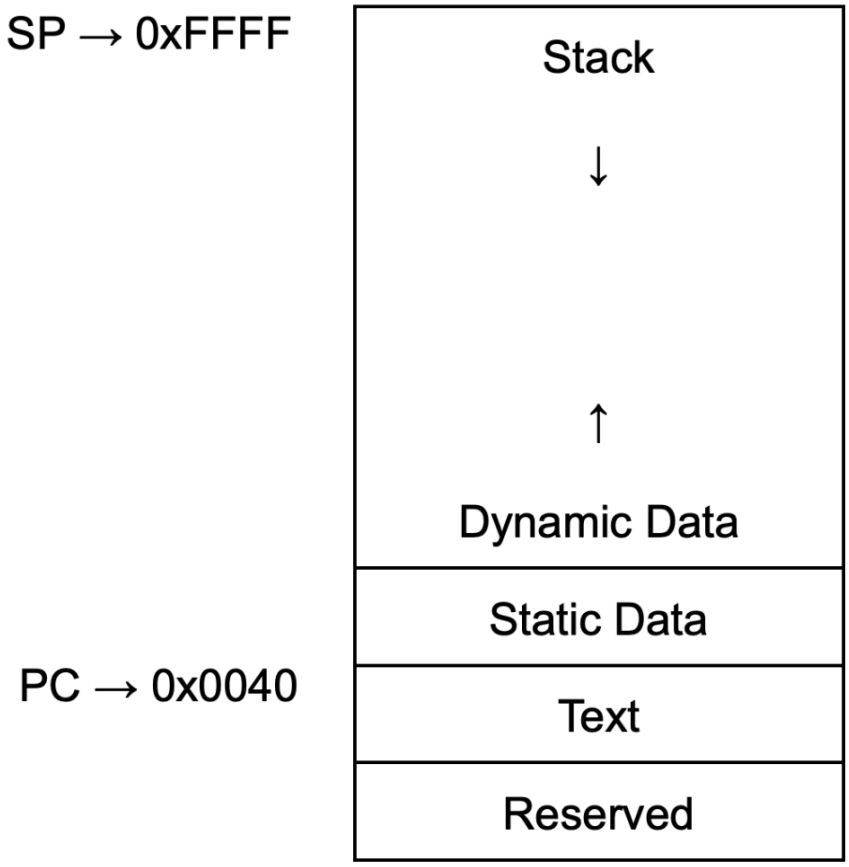
Special Registers

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



RTL Component

ALU

Inputs

// First Operand Input
input_A[15:0]: 16-bit input representing the first operand for the ALU.

// Second Operand Input
input_B[15:0]: 16-bit input representing the second operand for the ALU.

// ALU Operation Code Input
input_ALUOp[2:0]: 3-bit input specifying the ALU operation code.

// Clock Signal
CLK[0:0]: Single-bit clock signal used to synchronize ALU operations.

Outputs

// Result of ALU Operation
output_ALU[15:0]: 16-bit output representing the result of the ALU operation.

// Zero Flag Output
output_Zero[0:0]: Single-bit output indicating whether the ALU result is zero (0) or not.

// Negative Flag Output
output_negative[0:0]: Single-bit output indicating whether the ALU result is negative or not.

Behavior

On the rising edge of the clock(CLK) the ALU takes input_A and input_B and performs some operation as determined by input_ALUOp, the result is then the value of output_ALU. The ALU also has the flag output_Zero, which is 1 if output_ALU is zero, and the flag output_negative which is 1 if output_ALU is less than 0.

RTL Sysmbols

ALUoutput(operation,input_A,input_B): output of ALU with given operation and inputs
Operations: (determined by input_ALUOp
add: input_A + input_B
sub: input_A - input_B
bitshift: input_A bit-shifted by input_B
arithmeticBitshift: input_A arithmetic bit-shifted by input_B
and: input_A & inpu_B
or: input_A | input_B
xor: input_A ^ input_B

Control Unit

Inputs

input_control[6:0]: the opcode + funct4
CLK[0:0]: Single-bit clock signal used to synchronize operations.

Outputs

// Flags:
output_branch[0:0]: 1 if it's a branch instruction, 0 otherwise.
output_memRead[0:0]: 1 if reading from memory, 0 otherwise.
output_ALUOp[2:0]: ALU operation (+, -, and, or, ...)
output_memWrite[0:0]: 1 if writing to memory, 0 otherwise.
output_ALUSrc[0:0]: 1 if using immediate, 0 otherwise.
output_regWrite[0:0]: 1 if the instruction involves writing to a register, 0 otherwise.
output_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

On the rising edge of the clock(CLK) interprets the 3 opcode bits of the instruction to know what bits are what, and if necesary reads func 4 bits to determine apropraite ALU operation and sends this to the ALU.

RTL Sysmbols

Data Memory

Inputs

// Memory Write Enable Signal
input_mem_write[0:0]: Single-bit input indicating when a write instruction is enabled.

// Memory Address Input
input_mem_addr[15:0]: 16-bit input representing the memory address for read or write operations.

// Memory Data Input
input_mem_data[15:0]: 16-bit input representing the data to be written into the memory when a write instruction is enabled.

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), the data memory unit will read 16-bit data at the memory address specified by input_mem_addr and output it to output_mem_data. If input_mem_write is 1 it will also write the data it receives from input_mem_data to the adderess specified by input_mem_adres.

RTL Sysmbols

mem[adress]: the 16 bit value at this memory adress

Immediate Generator

Inputs

input_imm[15:0]: 16-bit input includes the entire instruction.

Outputs

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

Behavior

On the rising edge of the clock(CLK), the immediate generator will get the instrution and generate the imm by understanding the opcode.

RTL Sysmbols

imm[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 immediates in 2RI instruction types

Instruction Memory

Inputs

input_instrAddress[15:0]: 16-bit input representing the memory address
input_instr_data[15:0]: 16-bit input representing data to write
input_instr_write[0:0]: flag indicating if we are writing to the input_instr_data to the adress input_instrAddress
// Clock Signal
CLK[0:0]: Single bit clock signal used to synchronize Memory operations.

Outputs

output_instruction[15:0]: the instruction pointed to by PC

Behavior

On the rising edge of the clock(CLK), the instruction memory unit will read 16-bit instruction at the memory address specified by input_instrAdress and output it to output_instruction. If input_instr_write is 1 it will also write the data it recieves from input_instr_data to the adress specified byinput_instrAdress.

RTL Sysmbols

inst[15:0]: the bytes of the instruction pc currently points to

Program Counter

Inputs

input_PC[15:0]: the immediate that determines what to set(if PC_set) or add(if PC_branch) to PC
input_ALU_zero[0:0]: the signal from the ALU if output was zero
input_ALU_negative[0:0]: the signal from the ALU if output was negative
CLK[0:0]: clock
input_PC_set[0:0]: signal from control if pc is being set to a value
input_PC_branchType[1:0]: selector from control determining what kind of branch comparison is being performed
input_PC_isbranch[0:0]: signal from control if a branch comparison is being performed

Outputs

output_PC[15:0]: the updated value of the Program Counter.

Behavior

On the rising edge of the clock(CLK) the program counter checks PC_isbranch and PC_set:

*when PC_isbranch is 1 it checks PC_branchType and input_ALU_zero and input_ALU_negative to determine if a branch is necessary. If a branch is necessary it sets PC to PC+(input_PC *2)

*when PC_set is 1, it sets PC to input_PC*2

*the only other possibility is that that neither is 1 (they should never both be 1) in which case PC is incremented by 2 after the instruction is performed

Note that all addition and doubling is done within the PC component

RTL Sysmbols

PC[15:0]: the instruction address value held in PC

Programable Registers

Inputs

// Address for Reading Operand A
input_reg_readA_address[2:0]: 3-bit address specifying the register location for reading Operand A.

// Address for Reading Operand B
input_reg_readB_address[2:0]: 3-bit address specifying the register location for reading Operand B.

// Register Write Enable Signal
input_reg_write[0:0]: Single-bit signal to enable the write operation to the register.

// Value to be Written to the Register
input_reg_write_value[15:0]: 16-bit value to be written into the register when a write operation is enabled.

// Address for Writing to the Register
input_reg_write_address[2:0]: 3-bit address specifying the register location for writing data when reg_write is enabled.

// Clock Signal
CLK[0:0]: Single-bit clock signal used to synchronize read and write operations in the programmable register.

Outputs

// Output of Register A
output_reg_A[15:0]: 16-bit output representing the data read from Register A.

// Output of Register B
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].

RTL Sysmbols

Reg[index]: the value of the register at this index

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

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	$R[rd] = R[r1]\ \|\ R[r2]$	
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

RTL

$inst = Mem[PC]$
$input\_A = Reg[inst[12:10]]$
$input\_B = Reg[inst[9:7]]$
$Reg[inst[15:13]] = ALUoutput(operation, input\_A, input\_B)$
$PC = PC + 2$

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	$if(R[rd] \geq R[r1])$ $PC+= 2 * OR(UI, imm)$	

Instruction	RTL
$addi$ $xori$ ori $andi$ $subi$ $slli$ $srli$ $srai$	$inst = Mem[PC]$ $input\_A = Reg[inst[12:10]]$ $input\_B = imm$ $Reg[inst[15:13]] = ALUoutput(operation, input\_A, input\_B)$ $PC = PC + 2$
lw	$//\ Instruction\ Decode$ $inst = Mem[PC];$ $//FetchInstructionfromMemory$ $//\ ALU\ Inputs$ $input_A = Reg[inst[12:10]];$ $//FirstALUinputisthevalueinregisterr1$ $input_B = imm;$ $//\ Second\ ALU\ input\ is\ the\ immediate\ value\ generated\ by\ the\ Immediate\ Generator$ $//\ ALU\ Operation$ $//theALUoperation: 2 * (input_A + input_B)$ $ALUoutput = ALU(operation, input_A, input_B);$ $//\ Perform\ ALU\ operation$ $//\ Data\ Memory\ Access$ $dataMemAdd = ALUoutput;$ $//\ Address\ of\ Data\ Memory\ is\ the\ ALU\ operation\ result$ $dataMemData = dataMem[dataMemAdd];$ $//\ Read\ data\ from\ Data\ Memory\ at\ the\ specified\ address$ $//\ Register\ Write$ $Reg[inst[15:13]] = dataMemData;$ $//\ Write\ the\ data\ read\ from\ Data\ Memory\ to\ the\ specified\ register$ $//\ Instruction\ Fetch$ $PC = PC + 2;\ //\ Increment\ Program\ Counter$

sw	$// \ Instruction \ Decode$ $inst = Mem[PC];$ $//Fetch \ Instruction \ from \ Memory$ $// \ ALU \ Inputs$ $input_A = Reg[inst[12:10]];$ $//First \ ALU \ input \ is \ the \ value \ in \ register \ 1$ $input_B = imm;$ $//Second \ ALU \ input \ is \ the \ immediate \ value \ generated \ by \ the \ Immediate \ Generator$ $// \ ALU \ Operation$ $//the \ ALU \ operation \ is \ 2 * (input_A + input_B)$ $ALUoutput = ALU(operation, input_A, input_B);$ $//Perform \ ALU \ operation$ $// \ Data \ Memory \ Access \ - \ Store \ Word$ $dataMemAdd \ = \ ALUoutput;$ $// \ Address \ of \ Data \ Memory \ is \ the \ ALU \ operation \ result$ $dataMemData = Reg[inst[15:13]];$ $//Data \ to \ be \ stored \ is \ the \ value \ in \ register \ specified \ by \ inst[15:13]$ $// \ Store \ Word \ Operation$ $dataMem[dataMemAdd] = dataMemData;$ $//Store \ the \ data \ from \ the \ register \ to \ Data \ Memory \ at \ the \ specified \ address$ $// \ Instruction \ Fetch$ $PC \ = \ PC \ + \ 2;$ $// \ Increment \ Program \ Counter$
$jalr$	$// \ Instruction \ Fetch$ $PC \ = \ PC \ + \ 2;$ $// \ Increment \ Program \ Counter$ $// \ Instruction \ Decode$ $inst = Mem[PC];$ $// \ Fetch \ Instruction \ from \ Memory$ $// \ PC \ Control \ Inputs$ $input_PC = imm;$ $// \ Second \ input \ is \ the \ immediate \ value \ generated \ by \ the \ Instruction \ Generator$ $//ALU \ do \ nothing$ $// \ Register \ Write \ - \ Jump \ And \ Link \ Reg$ $Reg[inst[15:13]] = PC;$ $// \ Write \ the \ current \ PC \ value \ to \ the \ destination \ register \ rd$ $// \ PC \ Operation$ $PC \ = imm * 2;$ $// \ Perform \ ALU \ operation$
beq blt	$// \ Instruction \ Decode$ $inst = Mem[PC];$

bne	$// Fetch Instruction from Memory$
	
	$// ALU Inputs$
	$input_A = Reg[inst[12:10]];//FirstALUinputisthevalueinregisterr1$
	$input_B = Reg[inst[15:13]];//SecondALUinputisthevalueinregisterrd$
	
	$//ALU do minus and output flag$
	$ALU_output = ALU(sub, input_A, input_B);$
	$// Perform ALU operation$
	
	$// PC Operation$
	$if(op(output\_zero, output\_negaive)){$
	$PC += imm * 2;$
	$// Perform PC operation$
	$}$
	$else\{$
	$// Instruction Fetch$
	$PC = PC + 2;$
	$// Increment Program Counter\}$

bge	

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

Instruction	RTL
inc	$inst = Mem[PC]$ $input\_A = Reg[inst[15:13]]$ $input\_B = imm$ $Reg[inst[15:13]] = ALUoutput(operation, input\_A, input\_B)$ $PC = PC + 2$
dec	
$slipl$	
$sripl$	
$slipa$	
$sripa$	
set	$inst = Mem[PC]$ $Reg[inst[15:13]] = imm$ $PC = PC + 2$

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	$UI = immediate[12:0]$	sets non-programable UI (upper immediate) register to be used with 2RI instructions

RTL
$inst = Mem[PC]$ $UI = imm$ $PC = PC + 2$

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	$\mathrm{PC} = 2 * \mathrm{imm}[9:0]$	moves execution to the instruction at address 2*(immediate 9:0)

* jal is used when programmer is calling a function.

RTL
$inst = Mem[PC]$ $PC = ALUoutput(bitshift, imm, 2)$ $PC = PC + 2$

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

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:

```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	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