Design Document

Team 1, Section 3
Ethan Bennett, Jacob Steimle, Harrison Carpenter, Medhansh Khattar
Google Docs link

Design Philosophy

Our design is aimed at reducing the total number of instruction types to implement the relatively prime functionality. By doing so, we plan to limit the number of bits for funct to 1, and opcode to 3 bits. This enables us to allow the use of multiple registers for load-store architecture, and in continuation, branch to farther addresses in 16-bit instructions. Once we decided on 16 bit core instruction formats, we moved onto developing RTLs for each and planning all components required for our datapath in the future.

Performance

Performance will be measured using execution time of the relPrime function.

Registers

Register	Name	Description
x0	zero	Zero constant.
x1	ra	Return address.
x2	sp	Stack pointer.
x3	gp	Global pointer.
x4	tp	Thread pointer.
x5	t0	Temporaries.
x6	t1	
x7	t2	
x8	s0, fp	Callee-saved registers.
x9	s1	
x10	a0	Function arguments and return values.
x11	a1	
x12	a2	
x13	a3	
x14	t3	Another temporary.
x15	at	Assembler temporary.

Instruction Types

A-Type (Arithmetic)

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

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

Instruction	Opcode	Funct	Example	Meaning	Description
add	000	0	add x5, x6, x7	x5 = x6 + x7	R[rd] = R[rs1] + R[rs2]
sub	000	1	sub x5, x6, x7	x5 = x6 - x7	R[rd] = R[rs1] - R[rs2]

I-Type (Immediate)

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

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

Instruction	Opcode	Funct	Example	Meaning	Description
addi	001	0	addi x5, x6, 7	x5 = x6 + 7	R[rd] = R[rs1] + SE(imm)
lw	001	1	lw x5, 0(x6)	Load a value from memory address x6 with an offset of 0 bytes into x5.	R[rd] = M[R[rs1]]+SE(imm)

S-Type (Store)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Imm[8|6:4]	rs1	Imm[3:0|7]	opcode

Allows storing of a register value to memory.

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

B-Type (Branch)

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

Allows for two types of branches, being branch equal (beq) and branch greater than (bgt). Branch not equal and branch inclusive inequality is implemented by branching over unconditional jumps with their opposites.

Instruction	Opcode	Funct	Example	Meaning	Description
beq	011	0	beq x5, x6, -8	If x5 and x6 are equal, go back 8 instructions.	if(rs1 == rs2) PC += SE(imm)<<1
bgt	011	1	bgt x5, x6, 4	If x5 is greater than x6, go forward 4 instructions.	if(rs1 > rs2) PC += SE(imm)<<1

JAL-Type (Jump-And-Link)

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

Allows for unconditional jumps, with a larger range than a branch instruction. Also updates the return address.

Instruction	Opcode	Funct	Example	Meaning	Description
jal	100	0	jal ra, relPrime	Jump to relPrime and set return address.	R[rd] = PC + 2 PC += SE(imm)<<1

Memory Map

0x0000 - 0x00FF	Stack
0x01FF - 0x7FFF	Text
0x8000 - 0xFFFF	Data

Addressing Modes

This processor will use both Register and Immediate addressing modes. Immediate bits are not used in A-Type instructions, as those use register addressing. In I-Types, use the given immediate to either add to a register value, or offset a memory address. In S-Types, they are used to offset memory addresses. They are also rearranged to enable symmetry with other instruction types. In B-Types and JAL-Types, they are used to offset the PC (PC-Relative). This operation is PC relative, since moving the instruction around will still result in the same number of instructions jumped from that position.

Procedure Calling Conventions

Caller	Callee
Preserve return address across calls	Use given return address
Preserve temporary registers across calls	Freely use temporary registers
Freely use save registers across procedures	Preserve save registers, and restore before returning to caller
Preserve essential argument registers before calling	Use argument registers given

Procedure calling convention example

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

Machine Language Translation of relPrime

Address	Assembly	Machine Code	Comment
0x0000	relPrime: addi sp, sp, -4	1100 0010 0010 0001	# Allocate stack space for local variables
0x0002	sw ra, 2(sp)	0010 0001 0000 1010	# Save return address
0x0004	sw a0, 0(sp)	0000 0001 0101 0010	# Save argument 'n'
0x0006	addi a1, a1, 2	0010 1011 1011 0001	# m = 2
0x0008	relPrime_loop: addi a2, a0, 0	0000 1100 1010 0001	# Copy 'n' to a2 for gcd function
0x000A	addi a3, a1, 0	0000 1101 1011 0001	# Copy 'm' to a3 for gcd function
0x000C	jal ra, gcd	0000 0101 0000 1100	# Call gcd(n, m)
0x000E	beq a0, zero, +4	1010 0000 0010 0011	# if gcd(n, m) == 1, don't update 'm'
0x0010	jal zero, update_m	0000 0001 1000 0100	# update 'm'
0x0012	addi a1, a1, 1	0001 1011 1011 0001	# m = m + 1
0x0014	jal zero, relPrime_loop	1111 1101 0000 0100	# Loop back
0x0016	update_m: addi a0, a1, 0	0000 1011 1010 0001	# Prepare return value
0x0018	lw ra, 2(sp)	0100 0010 0001 1001	# Restore return address

0x001A	lw a0, 0(sp)	0000 0010 1010 1001	# Restore 'n'
0x001C	addi sp, sp, 4	0100 0010 0010 0001	# Deallocate stack space
0x001E	jal zero, 0(ra)	XXXX XXXX X000 0100	# Return
0x0020	gcd: addi sp, sp, -6	1010 0010 0010 0001	# Allocate stack space for local
0x0022	sw ra, 4(sp)	0000 0001 0100 0010	# Save return address
0x0024	sw a0, 2(sp)	0000 1010 0010 0010	# Save 'a'
0x0026	sw a1, 0(sp)	0000 1011 0000 0010	# Save 'b'
0x0028	gcd_loop: beq a0, zero, 5	0000 1010 0101 0011	# if a == 0, jump to the jal 5 instructions down
0x002A	beq a1, zero, update_a	0000 1011 0101 1011	# if b == 0, update 'a'
0x002C	bgt a1, a0, subtract_b	1010 1011 0111 1011	# if a > b, a = a - b
0x002E	sub a0, a0, a1	1011 1010 1010 1000	# subtract a1 from a0, store in a0
0x0030	jal zero, gcd_loop	1111 1101 1000 0100	# loop
0x0032	jal zero, end_gcd	0000 0011 1000 0100	# jump to end_gcd
0x0034	update_a: addi a0, a1, 0	0000 1011 1010 0001	# a = b
0x0036	lw a1, 0(sp)	0000 0010 1011 1001	# Restore 'b'
0x0038	jal zero, gcd_loop	1111 1100 0000 0100	# loop

0x003A	subtract_b: sub a1, a1, a0	1010 1011 1011 1000	# else, b = b - a
0x003C	jal zero, gcd_loop	1111 1011 0000 0100	# loop
0x003E	end_gcd: lw ra, 4(sp)	0100 0010 0001 1001	# restore stack
0x0040	lw a0, 2(sp)	0010 0010 0001 1001	# restore stack
0x0042	lw a1, 0(sp)	0000 0010 0001 1001	# restore stack
0x0044	addi sp, sp, 6	0110 0010 0010 0001	# restore stack
0x0046	jal zero, 0(ra)	XXXX XXXX X000 0100	# return to caller

RTL

add
newPC = PC + 2 PC = newPC inst = InstMem[PC] a = Reg[inst[11:8]] b = Reg[inst[15:12]] result = a + b Reg[inst[7:4]] = result

addi
newPC = PC + 2 PC = newPC inst = InstMem[PC] a = Reg[inst[11:8]] imm = SE(inst[15:12]) result = a + imm Reg[inst[7:4]] = result

lw
newPC = PC + 2 PC = newPC inst = InstMem[PC] a = Reg[inst[11:8]] imm = SE(inst[15:12]) result = a + imm memOut = Mem[result] Reg[inst[7:4]] = memOut

sw
newPC = PC + 2 PC = newPC inst = InstMem[PC] a = Reg[inst[11:8]] b = Reg[inst[15:12]] imm = SE(inst[7:4]) result = a + imm Mem[result] = b

beq
newPC = PC + 2 PC = newPC inst = InstMem[PC] a = Reg[inst[11:8]] b = Reg[inst[15:12]] imm = SE(inst[7:4])<<1 target = PC + imm if (a == b) PC = target

bgt
newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[inst[11:8]] b = Reg[inst[15:12]] imm = SE(inst[7:4])<<1 target = PC + imm if (a > b) PC = target

sub
newPC = PC + 2 PC = newPC inst = InstMem[PC] a = Reg[inst[11:8]] b = Reg[inst[15:12]] result = a + b Reg[inst[7:4]] = result

jal
newPC = PC + 2 PC = newPC inst = Mem[PC] imm = SE(inst[15:8])<<1 PC = PC + imm

Generic Component Specifications

Program Counter (PC) - Holds next instruction, which it also outputs each cycle.

Instruction Memory - Stores all instructions, outputs the 16 bit instruction referenced by the Program Counter.

Register File - File containing 16 registers, with 5 inputs (4 bit Register Source 1, 4 bit Register Source 2, 4 bit Register Destination, 16 bit data, and a 1 bit Register Write Enable). Also has two 16 bit data outputs corresponding to the data at Register Source 1 and 2.

ALU - ALU has two 16 bit inputs and one 16 bit output, as well as a 1 bit input to determine the operation. It will essentially support addition and subtraction either between two registers, or between a register and an immediate.

Data memory - Data memory has 3 inputs, a 16 bit data address, and 16 bit data, as well as a 1 bit memory write enable. It has one output, a 16 bit data line.

PC Increment Adder - Sole purpose is to increment the Program Counter by either A) 2 bytes, or B) Some offset specified in a branch/jump instruction. It will take a 16 bit address from the PC, and either some 9 bit immediate offset, or a hard wired value of 2 (a multiplexer will be used to decide).

Immediate Generator - Will take a 16 bit instruction and output a 16 bit immediate.

Component	Inputs	Outputs	Behavior	RTL Symbols
Register	newPC [15:0]	inst_address [15:0]	Takes a 16 bit value on the rising clock edge, and outputs that same value.	PC
Register File	rs1[3:0] rs2[3:0] rd[3:0] reg_data[15:0] reg_write[0:0]	reg_a[3:0] reg_b[3:0]	Takes 3 register addresses and can read rs1 and rs2 into reg_a and reg_b, as well as write a 16 bit value to rd. Also has a 1 bit input that enables writing to registers.	Reg
Instruction Memory	inst_address [15:0]	inst[15:0]	Takes a 16 bit instruction address and outputs the instruction at that location in	InstMem

			memory.	
Data Memory	data_address [15:0] data_data* [15:0] data_write [0:0]	data_out[15:0]		

*change this later; it sounds stupid