Design Document

Zeen Wang, Jermaine Brown,
Helen Wang, Athena Henderson
Team: Yellow-2122a-01

Rose-Hulman Institute of Technology
CSSE 232 Computer Architecture I
Prof. Williamson
April 3, 2022

Link to the Google Doc *DesignDocument*

Description:

Our design uses a single register accumulator to store data and compare it to inputs. At all times we only use one register and use an allocated space in Memory for data. For our addresses we will use sign extensions to target specific places in memory to receive either data for destination. The input must have the correct first bit to target the proper place in memory.

We are going to use 2 registers, the accumulator($acc) and stack pointer($sp). The accumulator is the only register available by the programmer.

I:

Opcode	Immediate	Unused
5	8	3

AI:

Opcode	Address	Immediate
5	8	3

PC relative for bne and beq

A:

Opcode	Address	Unused
5	8	3

We left shift by 1 bit then we sign extent (the most significant bit will be 1 if it is a data and 0 if it is a instruction)

Instructions:

Name	Type	Operation	Opcode
load a	A	acc = Mem[getAddr(rt)]	00001
	Takes an 8 bit address a and loads the value at memory address a to the accumulator, using the address rule.		
save a	A	Mem[getAddr(rt)] = acc	00010
	Take an 8 bit address a and save the value in the accumulator into the memory with address a, using the address rule.		
loadui imm	I	acc = {imm, 8b'0}	00011
	Takes an 8 bit immediate and load it to the upper 8 bits of the accumulator		

bne	a, imm	AI	if(acc != Mem[getAddr(rt)]) PC = PC + 2 + getAddr(imm)	00100
		Takes an 8 bit address and a 3 bit immediate. If the value stored at address a is not equal to the value of the accumulator, then jump to the address calculated from the immediate using the branch address rule.		
beq	a, imm	AI	if(acc == Mem[getAddr(rt)]) PC = PC + 2 + getAddr(imm)	00101
		Takes an 8 bit address and a 3 bit immediate. If the value stored at address a is equal to the value of the accumulator, then jump to the address calculated from the immediate using the branch address rule.		
slt	a	A	acc = acc < Mem[getAddr(rt)] ? 1:0	00110
		Compare the value in the accumulator with the value stored at address a, if the accumulator is less than a then we set the accumulator to 1, else we set the accumulator to 0.		
slti	imm	I	acc = acc < SignExtent(imm) ? 1:0	00111
		Compare the value in the accumulator with the immediate, if the accumulator is less than the immediate then we set the accumulator to 1, else we set the accumulator to 0.		
j	a	A	PC = getAddr(rt)	01000
		Jump to the instruction with address a, calculated using the address rule.		
jal	a	I	Men[ra] = PC + 2 PC = getAddr(imm)	01001
		Jump to the instruction with address a, calculated using the address rule. Store the current PC + 2 to a fix memory location.		
sw	imm	I	Mem[sp + SignExtent(imm)] = acc	01010
		Stored the value in the accumulator onto the stack where it is offset imm to the stack pointer.		
lw	imm	I	acc = sp + SignExtent(imm)	01011
		Stored the value from the stack where it is off offset imm to the stack pointer to the accumulator.		
ms	imm	I	sp = sp + SignExtent(imm)	01100
		Move the stack pointer with the sign extended immediate.		

<table>
<tr><td rowspan="2">sub a</td><td>A</td><td>acc = acc - Mem[getAddr(rt)]</td><td>01101</td></tr>
<tr><td colspan="3">Subtract the value stored at address a from the accumulator and store the result in the accumulator</td></tr>
<tr><td rowspan="2">add a</td><td>A</td><td>acc = acc + Mem[getAddr(rt)]</td><td>01110</td></tr>
<tr><td colspan="3">Add the value stored at address a to the accumulator and store the result in the accumulator</td></tr>
<tr><td rowspan="2">addi imm</td><td>I</td><td>acc = acc + SignExtent(imm)</td><td>01111</td></tr>
<tr><td colspan="3">Add the sign extended immediate to the accumulator and store the result in the accumulator</td></tr>
<tr><td rowspan="2">and a</td><td>A</td><td>acc = acc & Mem[getAddr(rt)]</td><td>10000</td></tr>
<tr><td colspan="3">And the value stored at address a to the accumulator and store the result in the accumulator</td></tr>
<tr><td rowspan="2">or a</td><td>A</td><td>acc = acc | Mem[getAddr(rt)]</td><td>10001</td></tr>
<tr><td colspan="3">Or the value stored at address a to the accumulator and store the result in the accumulator</td></tr>
<tr><td rowspan="2">ori imm</td><td>I</td><td>acc = acc | ZeroExtent(imm)</td><td>10010</td></tr>
<tr><td colspan="3">Or the zero extended immediate to the accumulator and store the result in the accumulator</td></tr>
<tr><td rowspan="2">loadi imm</td><td>I</td><td>acc = SignExtent(imm)</td><td>10011</td></tr>
<tr><td colspan="3">Load the sign extended immediate to the accumulator.</td></tr>
<tr><td colspan="4">getAddr = {7{address[7]}, address, 1’b0}
ZeroExtent = {8b’0, imm}
SignExtent = {8{address[7]},imm}
ra = 0xFFFE
sp start at 0xFFFC</td></tr>
</table>

Address rule: We left shift by 1 bit then we sign extent (the most significant bit will be 1 if it is a data and 0 if it is a instruction)
Branch address: Left shift the immediate by 1, sign extend it to 16 bits then add it to the value of the current PC plus 2.

Types:

A:

Opcode	Direct Address (rt)	Unused
15 — 11	10 — 3	2 — 0

I:

Opcode	Immediate	Unused
15 — 11	10 — 3	2 — 0

AI:

Opcode	Direct Address (rt)	Immediate
15 — 11	10 — 3	2 — 0

Call procedure:

For the callers, they are responsible to store the $acc register value, and put the return address on the $acc. For callees, they are responsible to restore the value in the Data memory, and callees will move the stack to store the original value of the data in the stack memory, and restore them back before return. Also, it's callee's responsibility to store the return address in the stack memory and use them for return.

Example program(s):

High Level Code	Assembly	Machine Code	Addresses
int **relPrime**(int n) {	loadi 2	10011 00000010 000	0x 0030
int m;	save m	00010 10000011 000	0x 0032
m = 2;	ms -6	01100 11111010 000	0x 0034
while (gcd(n, m) != 1) {	loop: load m	00001 1000001 000	0x 0036
m = m + 1;	sw 0	01010 00000000 000	0x 0038
}	save a	00010 10000000 000	0x 003A
return m;	load n	00001 10000100 000	0x 003C
}	sw 2	01010 00000010 000	0x 003E
	save b	00010 10000001 000	0x 0040
	load ra	00001 11111111 000	0x 0042
	sw 4	01010 00000100 000	0x 0044
	jal gcd	01001 11100111 000	0x 0046
	save o	00010 10000101 000	0x 0048
	lw 0	01011 00000000 000	0x 004A
	save m	00010 10000011 000	0x 004C
	lw 2	01011 00000010 000	0x 004E
	save n	00010 10000100 000	0x 0050
	lw 4	01011 00000100 000	0x 0052
	save ra	00010 11111111 000	0x 0054
	loadi 1	10011 00000001 000	0x 0056
	bne o, end	00100 10000101 100	0x 0058
	load m	00001 10000011 000	0x 005A

	add 1 save m j loop end: ms 6 j ra	01110 00000001 000 00010 10000011 000 01000 00011011 000 01100 00000110 000 01000 11111111 000	0x 005C 0x 005E 0x 0060 0x 0062 0x 0064

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;
}
	gcd:
	loadi 0
	bne a, loop
	load b
	j ra
	loop:
	loadi 0
	bne b, go
	j end
	go: load b
	slt a
	save i
	loadi 1
	bne i, else
	load a
	sub b
	save a
	j loop
	else: load b
	sub a
	save b
	j loop
	end: load a
	j ra
		
		10011 00000000 000
		00100 10000000 010
		00001 10000001 000
		01000 11111111 000
		
		10011 00000000 000
		00100 10000001 001
		01000 00010110 000
		00001 10000001 000
		00110 10000000 000
		00010 10000010 000
		10011 00000001 000
		00100 10000010 011
		00001 10000000 000
		01101 10000001 000
		00010 10000000 000
		01000 00000101 000
		00001 10000001 000
		01101 10000000 000
		00010 10000001 000
		00010 00000101 000
		00001 10000000 000
		00010 11111111 000
			
			0x 0002
			0x 0004
			0x 0006
			0x 0008
			0x 000A
			0x 000C
			0x 000E
			0x 0010
			0x 0012
			0x 0014
			0x 0016
			0x 0018
			0x 001A
			0x 001C
			0x 001E
			0x 0020
			0x 0022
			0x 0024
			0x 0026
			0x 0028
			0x 002A
			0x 002C
			0x 002E

if (n == 0) {
n++;
} else {
n = 2;
}
	loadi 0
	bne n, else
	load n
	add 1
	save n
	j done
	else: loadi 2
	save n
	done:
		10011 00000000 000
		00100 10000100 100
		00001 10000100 000
		01110 00000001 000
		00010 10000100 000
		01000 00001001 000
		10011 00000010 000
		00010 10000100 000
			0x 0002
			0x 0004
			0x 0006
			0x 0008
			0x 000A
			0x 000C
			0x 000E
			0x 0010
			0x 0012

while (n != 0) {
n = n - m
}
	loop: loadi 0
	beq n, done
	load n
	sub m
	save n
		10011 00000000 000
		00101 10000100 100
		00001 10000100 000
		01101 10000011 000
		00010 10000100 000
			0x 0002
			0x 0004
			0x 0006
			0x 0008
			0x 000A

	j loop done:	01000	00000001	000	0x 000C 0x 000E

int count = 0;
for (int i = 0; i < n; i++) {
count++;
}
	loadi 0
	save count
	save i
	loop: beq n, done
	add 1
	save count
	save i
	j loop
	done:
		10011
		00010
		00010
		00101
		01110
		00010
		00010
		01000
			00000000
			10000110
			10000010
			10000100
			00000001
			10000110
			10000010
			00000100
				000
				000
				000
				100
				000
				000
				000
				000
					0x 0002
					0x 0004
					0x 0006
					0x 0008
					0x 000A
					0x 000C
					0x 000E
					0x 0010
					0x 0012

Data: 0xFF00 a(value = m) 0xFF02 b(value = n) 0xFF04 i 0xFF06 m 0xFF08 n 0xFF0A o 0xFF0C count		Stack: 0x1FFF			

Milestone 2:
-RTL
-Input, Output, Control signals + descriptions + bit size
-Components needed for RTL + how control & input signals affect output
-RTL symbols implemented by each component
-Set up + calling of relprime
-Minimum procedure call code
-Assembler

RTL:

Name	Fetch	Decode	
load a	IR = Mem[PC]; PC = PC + 2	MDR = Mem[SE(IR[10-3]<<1)] ALUOut = sp +SE(IR[10-3])	Acc = MDR
save a			Mem[SE(IR[10-3]<<1)] = Acc

loadui imm			Mem[SE(IR[10-3])] = Acc
bne a, imm			if(Acc != MDR) PC = PC + 2 + SE(IR[10-3]<<1)
beq a, imm			if(Acc == MDR) PC = PC + 2 + SE(IR[10-3]<<1)
slt a			if(Acc < MDR) Acc = 1 else Acc = 0
slti imm			if(Acc < SE(IR[10-3])) Acc = 1 else Acc = 0
j a			PC = SE(IR[10-3]<<1)
jal a			Mem[ra] = PC + 2 PC = SE(IR[10-3]<<1)
sw imm			Mem[ALUOut] = Acc
lw imm			Acc = Mem[ALUOut]
ms imm			Sp = sp + (SE(IR[10-3]))
sub a			ALUOut= Acc - MDR
add a			ALUOut= Acc + MDR
addi imm			ALUOut= Acc + (SE(IR[10-3]))
and a			ALUOut= Acc & MDR
or a			ALUOut= Acc | MDR

ori imm			ALUOut= Acc | (ZE(IR[10-3]))
loadi imm			ALUOut= SE(IR[10-3])

RTL Symbols

1. 16-bit Registers
   Input wires:
      16-bit bus for Input data,
      1-bit Enable wire,
      1-bit clock wire
   Output wire:
      16-bit bus for output data
2. Memory
   Input wires:
      16-bit bus for Memory Address,
      16-bit bus for Input data,
      1-bit Memory write control wire,
      1-bit Memory read control wire,
      1-bit clock wire
   Output wire:
      16-bit bus for output data
3. ALU
   Input wires:
      16-bit bus for Input A,
      16-bit bus for Input B,
      2-bit bus for ALUopcode,
      1-bit clock wire
   Output wire:
      16-bit bus for output data
      1-bit zero wire (high for 0)
   Opcode:
      00 for And
      01 for Or
      10 for Add
      11 for Sub
4. 1 bit-Mux

Input wires:
- 16-bit bus for Input A,
- 16-bit bus for Input B,
- 1-bit bus for ALUopcode

Output wire:
- 16-bit bus for output data

Opcode:
- 0 for A
- 1 for B

Input/Output/Control:

Name:	**Bit Size:**	**I/O/C?**	**Description:**	**Affect:**

- No changes are made to assembly language and machine language. Keeped the sign extend feather instead of making use of the 3 unused bits in jump so that ALU is not needed to calculate the address