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 (rt)	Immediate
5	8	3

PC relative for bne and beq

A:

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

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
n = 6; c = relPrime(n);	loadi 6 save n jal relPrime load m save c	10011 00000110 000 00010 10000100 000 01001 00011000 000 00001 10000011 000 00010 10000111 000	0x 0066 0x 0068 0x 006A 0x 006C 0x 006E

int **relPrime**(int n) {
    int m;
    m = 2;
    while (gcd(n, m) != 1) {
        m = m + 1;
    }
    return m;
}
	relPrime:
	loadi 2
	save m
	ms -6
	loop: load m
	sw 0
	save a
	load n
	sw 2
	save b
	load ra
	sw 4
	jal gcd
	save o
	lw 0
	save m
		
		10011 00000010 000
		00010 10000011 000
		01100 11111010 000
		00001 1000001 000
		01010 00000000 000
		00010 10000000 000
		00001 10000100 000
		01010 00000010 000
		00010 10000001 000
		00001 11111111 000
		01010 00000100 000
		01001 11100111 000
		00010 10000101 000
		01011 00000000 000
		00010 10000011 000
			
			0x 0030
			0x 0032
			0x 0034
			0x 0036
			0x 0038
			0x 003A
			0x 003C
			0x 003E
			0x 0040
			0x 0042
			0x 0044
			0x 0046
			0x 0048
			0x 004A
			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	01110 00000001 000	0x 005C
	save m	00010 10000011 000	0x 005E
	j loop	01000 00011011 000	0x 0060
	end: ms 6	01100 00000110 000	0x 0062
	j ra	01000 11111111 000	0x 0064
int **gcd**(int a, int b) {	gcd:		
if (a == 0) {	loadi 0	10011 00000000 000	0x 0002
return b;	bne a, loop	00100 10000000 010	0x 0004
}	load b	00001 10000001 000	0x 0006
while (b != 0) {	j ra	01000 11111111 000	0x 0008
if (a > b) {	loop:		0x 000A
a = a - b;	loadi 0	10011 00000000 000	0x 000C
} else {	bne b, go	00100 10000001 001	0x 000E
b = b - a;	j end	01000 00010110 000	0x 0010
}	go: load b	00001 10000001 000	0x 0012
}	slt a	00110 10000000 000	0x 0014
return a;	save i	00010 10000010 000	0x 0016
}	loadi 1	10011 00000001 000	0x 0018
	bne i, else	00100 10000010 011	0x 001A
	load a	00001 10000000 000	0x 001C
	sub b	01101 10000001 000	0x 001E
	save a	00010 10000000 000	0x 0020
	j loop	01000 00000101 000	0x 0022
	else: load b	00001 10000001 000	0x 0024
	sub a	01101 10000000 000	0x 0026
	save b	00010 10000001 000	0x 0028
	j loop	00010 00000101 000	0x 002A
	end: load a	00001 10000000 000	0x 002C
	j ra	00010 11111111 000	0x 002E
if (n == 0) {	loadi 0	10011 00000000 000	0x 0002
n++;	bne n, else	00100 10000100 100	0x 0004
} else {	load n	00001 10000100 000	0x 0006
n = 2;	add 1	01110 00000001 000	0x 0008
}	save n	00010 10000100 000	0x 000A
	j done	01000 00001001 000	0x 000C
	else: loadi 2	10011 00000010 000	0x 000E
	save n	00010 10000100 000	0x 0010

	done:		0x 0012

while (n != 0) {
        n = n - m
}
	loop: loadi 0
	        beq n, done
	        load n
	        sub m
	        save n
	        j loop
	done:
		10011 00000000 000
		00101 10000100 100
		00001 10000100 000
		01101 10000011 000
		00010 10000100 000
		01000 00000001 000
			0x 0002
			0x 0004
			0x 0006
			0x 0008
			0x 000A
			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 00000000 000
		00010 10000110 000
		00010 10000010 000
		00101 10000100 100
		01110 00000001 000
		00010 10000110 000
		00010 10000010 000
		01000 00000100 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 0xFF0E c		Stack: 0x1FFF	

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)

Link to the Google Doc *DesignDocument*

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:
MDRWrite	1	C	Tells system whether data should be written to (1) or read from (0) MDR
IRWrite	1	C	Tells system whether data should be written to (1) or read from (0) IR
PCWrite	1	C	Tells system whether data should be written to (1) or read from (0) PC
AccWrite	1	C	Tells system whether data should be written to (1) or read from (0) Acc
SpWrite	1	C	Tells system whether data should be written to (1) or read from (0) Sp
ALUSrc	2	C	Determines which operation is performed by the ALU (00 - And, 01 - Or, 10 - Add, 11 - Sub)
Zero?	1	C	Signal goes high to show that ALU is zero
MemWrite	1	C	Tells system whether data should be written to memory (1) or not
MemRead	1	C	Tells system whether data should be read from memory (1) or not
Reset	1	C	Reset signal
ReadIn	1	I	Input data from memory
WriteOut	1	O	Output data to memory

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