~~Title Page~~

Stack Based Architecture Design Document

Team 2v(Jinhao Sheng, Luke McNeil, Austin Swatek, Yiju Hao)

Table of Contents

Executive Summary

The general purpose processor we are making is stack-based and will make use of two stacks of registers: one as a stack of the instructions used for the operations in a function, and the other for keeping track of the return values of functions. This processor has two instruction formats: O-type and A-type. O-type instructions have a 4-bit opcode, 12-bit function, and are for instructions that require no arguments. O-types that require jumps use direct addressing, while A-types that require jumps use pseudo-direct addressing. A-type instructions have a 4-bit opcode, 12-bit immediate value or address, and are necessary for functions that require arguments. Since it is a stack-based processor, functions are called on a last-in-first-out (LIFO) basis, with the caller function pushing the arguments that the callee expects to the top of the stack. The callee pops the arguments off of the stack, and before its return call, puts the required return values back on the stack. To return, the callee pops its return address off the return stack and jumps to that address.

Instruction Formats

Format Type	Size	Structure	Description
O	2 bytes	OP 4 | FUNCT 12	OP - basic operation of the instruction FUNCT - sets the variant of the operation This format type is used for instructions that take no arguments.
A	2 bytes	OP 4 | IMM/ADDR 12	OP - basic operation of the instruction IMM/ADDR - a 12 bit constant or address This format type is for any instruction that takes one argument which is either an immediate or address.

Instructions

Name	Type	Argument	Description	OP	Funct
add	O		Pop the top two values off the stack, add them, and put the result on the stack.	0x0	0x000
beq	A	label	Pop the top two values off the stack. If they are equal, then branch to label.	0x1	
bez	A	label	Pop the top value off the stack. If it is zero then branch to label.	0x2	
dup	O		Push onto the stack a duplicate of the value currently on top of the stack.	0x0	0x001
drop	O		Pop the top value off the stack, throwing it away.	0x0	0x002
halt	O		Jump back to this instruction. This is a good way to stop the program.	0x0	0x003
getin	O		Read a 16 bit number from input and push it to the stack.	0x0	0x004
j	A	target	Jump to target.	0x3	
jal	A	target	Jump to target, and push onto the return address stack the address of the next instruction.	0x4	
js	O		Pop the top value off the stack and jump to that address.	0x0	0x005
lui	A	immediate	Shift the immediate left by 12 bits and then push it into the stack	0x8	
over	O		Push onto the stack the value of the second element on the stack.	0x0	0x006
or	O		Pop the top two values off the stack, or them, and put the result on the stack.	0x0	0x007
pop	A	address	Pop the top value off the stack and put it in memory at address.	0x5	
push	A	address	Push the value at the specified address onto the stack.	0x6	
pushi	A	immediate	Push onto the stack sign extended immediate.	0x7	
return	O		Pop the top element off the return address stack, and jump there.	0x0	0x008
slt	O		Pop the top two elements off the stack, and push a 1 to the stack if the second from the top element is less than the top element. Otherwise, push a 0.	0x0	0x009
sub	O		Pop the top two values off the stack, subtract them, and put the result on the stack.	0x0	0x00A
swap	O		Swap the top two elements on the stack.	0x0	0x00B

Addressing Modes

Instructions	Format Type	Addressing Modes
js, return	O	Direct
j, jal, beq, bez	A	Pseudo-Direct

Pseudo-Direct ~~Example~~ explanation

- Going from 16 bit address to the 12 bits in the instruction
    a. Shift the 16 bit number right 1
    b. Chop off the 4 most significant bits
    c. Use this 12 bit number in the instruction ADDR field.
- Going from 12 bits in the instruction to a 16 bit address

    nice

    a. Shift the 12 bits to the left 1
    b. Put on the front of these 13 bits the 3 most significant bits from $PC.
    c. Use this 16 bit number as the address to go to.

Direct Example

- Here the jump is looking at the value in a 16 bit register. The address to jump to is simply those 16 bits.

Procedure Calling Conventions

To prepare to call a function the caller must put all arguments that the callee expects to receive on top of the stack. Then the caller must call jal to go to the callee. This command will push the return address to the return address stack. The callee's responsibilities are to pop the arguments off the stack and leave on the stack any return values. The callee will then do the return instruction which will pop the top element off the return address stack before going back to the correct spot.

Included below in the code fragments (section #7) is an example of nested function calling.

RelPrime

RelPrime and Sample Procedure Call					
ADDR	**MC**	**LABEL**	**ASM**	**STACK**	**RETURN STACK**
0x0	0x0004	MAIN:	getin	() -> (n)	()
0x2	0x4003		jal RELPRIME	(n) -> (relprime(n))	() -> (0x4)
0x4	0x0003		halt	(relprime(n))	()
0x6	0x7002	RELPRIME:	pushi 2	(n) -> (n, m)	(0x4)
0x8	0x0006	RPLOOP:	over	(n, m) -> (n, m, n)	(0x4)
0xA	0x0006		over	(n, m, n) -> (n, m, n, m)	(0x4)
0xC	0x400F		jal GCD	(n, m, n, m) -> (n, m, gcd)	(0x4) -> (0x4, 0x8)
0xE	0x7001		pushi 1	(n, m, gcd) -> (n, m, gcd, 1)	(0x4)
0x10	0x100C		beq RETURNM	(n, m, gcd, 1) -> (n, m)	(0x4)
0x12	0x7001		pushi 1	(n, m) -> (n, m, 1)	(0x4)
0x14	0x0000		add	(n m 1) -> (n, m+1)	(0x4)
0x16	0x3004		j RPLOOP	(n, m+1)	(0x4)
0x18	0x000B	RETURNM:	swap	(n, m) -> (m, n)	(0x4)
0x1A	0x0002		drop	(m, n) -> (m)	(0x4)
0x1C	0x0008		return	(m)	(0x4) -> ()
0x1E	0x0006	GCD:	over	(n, m, a, b) -> (n, m, a, b, a)	(0x4, 0x8)
0x20	0x2020		bez RETURNB	(n, m, a, b, a) -> (n, m, a, b)	(0x4 0x8)
0x22	0x0001	LOOP:	dup	(n, m, a, b) -> (n, m, a, b, b)	(0x4 0x8)
0x24	0x2023		bez RETURNA	(n, m, a, b, b) -> (n, m, a, b)	(0x4 0x8)
0x26	0x0006		over	(n, m, a, b) -> (n, m, a, b, a)	(0x4 0x8)
0x28	0x0006		over	(n, m, a, b, a) -> (n, m, a, b, a, b)	(0x4 0x8)
0x2A	0x000B		swap	(n, m, a, b, a, b) -> (n, m, a, b, b, a)	(0x4 0x8)
0x2C	0x0009		slt	(n, m, a, b, b, a) -> (n, m, a, b, b<a)	(0x4 0x8)
0x2E	0x201D		bez ELSE	(n, m, a, b, b<a) -> (n, m, a, b)	(0x4 0x8)
0x30	0x000B		swap	(n, m, a, b) -> (n, m, b, a)	(0x4 0x8)
0x32	0x0006		over	(n, m, b, a) -> (n, m, b, a, b)	(0x4 0x8)
0x34	0x000A		sub	(n, m, b, a, b) -> (n, m, b, a-b)	(0x4 0x8)
0x36	0x000B		swap	(n, m, b, a-b) -> (n, m, a-b, b)	(0x4 0x8)
0x38	0x3011		j LOOP	(n, m, a-b, b)	(0x4 0x8)
0x3A	0x0006	ELSE:	over	(n, m, a, b) -> (n, m, a, b a)	(0x4 0x8)
0x3C	0x000A		sub	(n, m, a, b, a) -> (n, m, a, b-a)	(0x4 0x8)
0x3E	0x3011		j LOOP	(n, m, a, b-a)	(0x4 0x8)
0x40	0x000B	RETURNB:	swap	(n, m, a, b) -> (n, m, b, a)	(0x4 0x8)
0x42	0x0002		drop	(n, m, b, a) -> (n, m, b)	(0x4 0x8)
0x44	0x0008		return	(n, m, b)	(0x4, 0x8) -> (0x4)
0x46	0x0002	RETURNA:	drop	(n, m, a, b) -> (n, m, a)	(0x4, 0x8)
0x48	0x0008		return	(n, m, a)	(0x4, 0x8) -> (0x4)

Code Fragments

this doesn't "return" a value

Sample Return					
Int main() {return (2 + 3) - 1;}					
ADDR	**MC**	**LABEL**	**ASM**	**STACK**	**RETURN STACK**
0x0	0x7002	MAIN:	pushi 2	() -> (2)	()
0x2	0x7003		pushi 3	(2) -> (2, 3)	()
0x4	0x0000		add	(2, 3) -> (5)	()
0x6	0x7001		pushi 1	(5) -> (5, 1)	()
0x8	0x000A		sub	(5, 1) -> (4)	()
0xA	0x0003		halt	(4)	()

Loading an 16 bit address onto the stack. (load 1000 0000 0000 0001)					
ADDR	**MC**	**LABEL**	**ASM**	**STACK**	**RETURN STACK**
0x0	0x8008	MAIN:	lui 0x8	() -> (0x8000)	()
0x2	0x7001		pushi 1	(0x8000) -> (0x8000, 0x1)	()
0x4	0x0007		or	(0x8000, 0x1) -> (0x8001)	()
0x6	0x0003		halt		()

does this load only 4 bits?

Sample For Loop					
int main(){					
int x = 1					
for(int i = 0; i < 5; i++){ x++;}					
ADDR	**MC**	**LABEL**	**ASM**	**STACK**	**RETURN STACK**
0x0	0x7001	MAIN:	pushi 1	() -> (x)	()
0x2	0x7000		pushi 0	(x)-> (x i)	()
0x4	0x0001	LOOP:	dup	(x i) -> (x i i)	()
0x6	0x7005		pushi 5	(x i i) -> (x i i 5)	()
0x8	0x0009		slt	(x i i 5) -> (x i i<5)	()
0xA	0x7001		pushi 1	(x i i<5) -> (x i i<5 1)	()
0xC	0x1009		beq OP	(x i i<5 1) -> (x i)	()
0xE	0x0002		drop	(x)	()
0x10	0x0003		halt	(x)	()
0x12	0x7001	OP:	pushi 1	(x i) -> (x i 1)	()
0x14	0x0000		add	(x i 1) -> (x i++)	()
0x16	0x000A		swap	(x i++) -> (i++ x)	()
0x18	0x7001		pushi 1	(i++ x) -> (i++ x 1)	()
0x1A	0x0000		add	(i++ x 1) -> (i++ x++)	()
0x1C	0x000B		swap	(i++ x++) -> (x++ i++)	()
0x1E	0x3002		j LOOP	(x++ i++)	()

Return Chain
int main() {return f1(2);} **int f1(int a) {return f2(a);}**

int f2(int b) {return f3(b);} int f3(int c) {return c+1;}					
ADDR	**MC**	**LABEL**	**ASM**	**STACK**	**RETURN STACK**
0x0	0x7002	MAIN:	pushi 2	() -> (2)	()
0x2	0x4003		jal F1	(2)	() -> (0x4)
0x4	0x0003		halt	(3)	()
0x6	0x4005	F1:	jal F2	(2)	(0x4) -> (0x4, 0x8)
0x8	0x0008		return	(3)	(0x4) -> ()
0xA	0x4007	F2:	jal F3	(2)	(0x4, 0x8) -> (0x4, 0x8, 0xC)
0xC	0x0008		return	(3)	(0x4, 0x8) -> (0x4)
0xE	0x7001	F3:	pushi 1	(2) -> (2, 1)	(0x4, 0x8, 0xC)
0x10	0x0000		add	(2, 1) -> (3)	(0x4, 0x8, 0xC)
0x12	0x0008		return	(3)	(0x4, 0x8, 0xC) -> (0x4, 0x8)

Reading Data From the Input Port					
ADDR	**MC**	**LABEL**	**ASM**	**STACK**	**RETURN STACK**
0x0	0x0004	MAIN:	getin	() -> （input）	()
0x2	0x0003		halt	(input)	()

RTL

Notes:

- stack is 64 registers
- Rstack is the return address stack and is 64 registers

add, sub, or inst = Mem[PC] stack[0] = stack[1] OP stack[0] stack[1] = stack[2] stack[2] = stack[3] ... stack[63] = 0 PC = PC + 4 2?	**beq** inst = Mem[PC] A = stack[0] B = stack[1] stack[0] = stack[2] stack[1] = stack[3] ... stack[62] = 0 stack[63] = 0 if (A-B==0): PC = (PC[15:13])...(inst[11:0] << 1) else: PC = PC + 4	**bez** inst = Mem[PC] A = stack[0] stack[0] = stack[1] stack[1] = stack[2] ... stack[63] = 0 if (A==0): PC = (PC[15:13])...(inst[11:0] << 1) else: PC = PC + 4
dup inst = Mem[PC] stack[63] = stack[62] stack[62] = stack[61] ... stack[1] = stack[0] PC = PC + 4	**drop** inst = Mem[PC] stack[0] = stack[1] stack[1] = stack[2] ... stack[63] = 0 PC = PC + 4	**halt** ~~inst = Mem[PC]~~
getin inst = Mem[PC] stack[63] = stack[62] stack[62] = stack[61] ... stack[1] = stack[0] stack[0] = INPUT PC = PC + 4	**j** inst = Mem[PC] PC = (PC[15:13])...(inst[11:0] << 1)	**jal** inst = Mem[PC] Rstack[0] = PC + 4 Rstack[1] = Rstack[0] ... Rstack[63] = Rstack[62] PC = (PC[15:13])...(inst[11:0] << 1)
js inst = Mem[PC] A = stack[0] stack[0] = stack[1] stack[1] = stack[2] ... stack[63] = 0 PC = A	**lui** inst = Mem[PC] stack[63] = stack[62] stack[62] = stack[61] ... stack[1] = stack[0] stack[0] = inst[11:0] << 12 PC = PC + 4	**over** inst = Mem[PC] stack[63] = stack[62] ... stack[2] = stack[1] stack[1] = stack[0] stack[0] = stack[2] PC = PC + 4

pop inst = Mem[PC] A = stack[0] stack[0] = stack[1] stack[1] = stack[2] ... stack[63] = 0 Mem[PC[15:13]...(inst[11:0] << 1)] = A PC = PC + 4	**push** inst = Mem[PC] stack[63] = stack[62] ... stack[1] = stack[0] stack[0] = PC[15:13]...(inst[11:0] << 1)] PC = PC + 4	**pushi** inst = Mem[PC] stack[63] = stack[62] ... stack[1] = stack[0] stack[0] = SE(inst[11:0] << 1) PC = PC + 4
return inst = Mem[PC] A = Rstack[0] Rstack[0] = Rstack[1] Rstack[1] = Rstack[2] ... Rstack[63] = 0 PC = A	**slt** inst = Mem[PC] A = stack[0] B = stack[1] if (B < A): stack[0] = 1 else: stack[0] = 0 stack[1] = stack[2] stack[2] = stack[3] ... stack[63] = 0 PC = PC + 4	**swap** inst = Mem[PC] A = stack[0] stack[0] = stack[1] stack[1] = A PC = PC + 4