Processor Project Team V (or something like this)

~~Project Team~~

Ethan Hutton, Ethan Townsend, Wesley Schuh, Garrett Doolittle

DATE?

Please add a table of contents after this page.

Description

Our 16-bit processor will use a memory to memory architecture capable of running a variety of general purpose programs, specifically Euclid's algorithm. We plan to implement our processor using a memory to memory architecture because of its efficiency in number of instructions and total bytes transferred when compared to other architectures. We do not require registers since we are using a memory to memory architecture, so our focus when designing instructions is on optimizing data transfers and prioritizing quick data storage and retrieval.

Performance Testing

A combination of unit tests on individual components and overall system tests in Verilog. Each operation's total time for completion can be measured in individual testing. Each operation can also be broken down into various hardware accesses that can also be tested and measured in regards to performance

you can omit this

Register Details

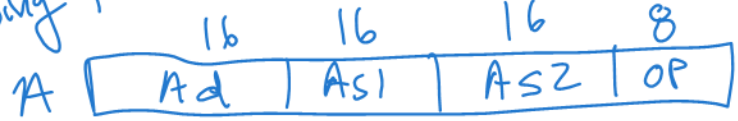
Since we are doing memory to memory we currently have no registers and have no plans to have any. This could change throughout the creation of our processor, but as of right now we have no registers to describe.

Instruction Types

A - arithmetic instructions. Will take a 8 bit opcode, 16 bit result address, and 2 16 bit addresses to pull the values that need to be computed. Things like shift left, shift right, shift right arithmetic, add, sub, xor, and, or will all use this type of instruction.

AI - Any instructions with an immediate and a register that need to be computed together. Similar to A type just an 8 bit opcode, 16 bit result address, 16 bit address to pull the value and a 16 bit immediate. Includes addi, andi, xori, ori.

Recommend drawing the formats: or point to the section below



label? address?

BR - branch instructions that will take an 8 bit opcode, a 16 bit tag, and 2 16 bit addresses. The BR instructions will compare an immediate with the value stored in memory and either branch to a given tag or just move on to the next line in the newPC. This instruction applies to blt, bge, bne, beq.

BRI - Similar to BR instructions, it will take an 8 bit opcode, a 16 bit immediate, a 16 bit tag, and a 16 bit address. Includes beqi, bnei, blti, bgei, bgti, blei.

why not have an SP reg?

J - jump instructions that will take an 8 bit opcode, a 16 bit tag, and a 16 bit address that is basically a stack pointer. It will jump to the current tag, while storing the current address on the stack. This will apply to to j.

SP - Instruction type used to push and pop addresses on the stack. Will take an 8 bit opcode. Used for the push and pop.

L - Instruction type used to to load an immediate into memory. This requires an 8 bit opcode, a 16 bit immediate, and a 16 bit destination address.

Addressing Modes

Our addressing will use direct addressing. There is no PC relative and every value will have to be loaded in from memory via a designated address.

Calling Conventions

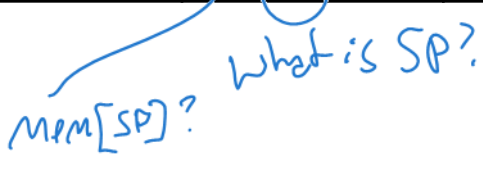
The only convention we currently have is that we push the address we need to return to on the stack and pop it off to use it as the place to return to once we're finished with a function

Arguments?
Return values?

ISA

Inst	Type	opcode	Usage	Logical Description	Description (verbose)
add	a	00000000	add dest s1 s2	dest = s1 + s2	Stores the sum of s1 and s2 into dest
sub	a	00100000	sub dest s1 s2	dest = s1 - s2	Stores the difference of s1 and s2 into dest
or	a	00000001	or dest s1 s2	dest = s1 | s2	Stores bitwise s1 OR s2 into dest
and	a	00000010	and dest s1 s2	dest = s1 & s2	Stores bitwise s1 AND s2 into dest
xor	a	00000100	xor dest s1 s2	dest = s1 ^ s2	Stores bitwise s1 XOR s2 into dest
sri	a	00111000	Sri dest s1 imm	dest = s1 >> imm	Stores s1 shifted right by imm into dest
srl	a	00111110	Srl dest s1 s2	dest = s1 >> s2	Stores s1 shifted right by s2 into dest
sll	a	00111100	sll dest s1 s2	dest = s1 << s2	Stores s1 shifted left by s2 into dest
sili	a	01111100	Sili dest s1 imm	dest = s1 << imm	Stores s1 shifted left by imm into dest
addi	ai	10000000	addi dest s1 imm	dest = s1 + imm	Stores the sum of s1 and imm into dest
andi	ai	10000010	andi dest s1 imm	dest = s1 & imm	Stores bitwise s1 AND imm into dest
ori	ai	10000001	ori dest s1 imm	dest = s1 | imm	Stores bitwise s1 OR imm into dest
xori	ai	10000100	xori dest s1 imm	dest = s1 ^ imm	Stores bitwise s1

					XOR imm into dest
j	j	10100000	j dest	PC = dest	Sets the PC to the given value
beq	br	00110000	beq dest s1 s2	if(s1==s2) PC = dest	Sets the PC to the given value if s1 is equal to s2
blt	br	00111010	blt dest s1 s2	if(s1<s2) PC = dest	Sets the PC to the given value if s1 is less than s2
bne	br	00111001	bne dest s1 s2	if(s1!=s2) PC = dest	Sets the PC to the given value if s1 is not equal to s2
bge	br	00110100	bge dest s1 s2	if(s1>=s2) PC = dest	Sets the PC to the given value if s1 is greater than or equal to s2
beqi	bri	10110000	beqi dest s1 imm	if(s1==imm) PC = dest	Sets the PC to the given value if s1 is equal to imm
blti	bri	10111010	blti dest s1 imm	if(s1<imm) PC = dest	Sets the PC to the given value if s1 is greater than imm
bnei	bri	10111001	bnei dest s1 imm	if(s1!=imm) PC = dest	Sets the PC to the given value if s1 is not equal to imm
bgei	bri	10110100	bgei dest s1 imm	if(s1>=imm) PC = dest	Sets the PC to the given value if s1 is equal to or equal to imm
bgti	bri	10100100	bgti dest s1 imm	if(s1>imm) PC = dest	Sets the PC to the given value if s1 is greater than imm
blei	bri	10101100	blei dest s1 imm	if(s1<=imm) PC = dest	Sets the PC to the given value if s1 is less than or equal to imm
pushpc	sp	01111111	push s1	SP -= 2; stack[SP] = PC	Pushes the PC onto the stack,



					updates the stack pointer accordingly
poppc	sp	01111110	~~pop s1~~ poppc	PC = stack[SP] SP += 2	Pop PC off the stack, updates the stack pointer accordingly
push	j	01011111	push s1	SP -= 2; stack[SP] = s1	Pushes the value at s1 onto the stack
pop	j	01011110	pop s1	s1 = stack[SP] SP += 2	Pops the value at the top of the stack into s1
mov	L	10111111	mov dest imm	dest = imm	Sets the value at the dest memory address to be the given immediate imm

Memory Map

Stack Pointer Location	FFFF
Stack space (SP)	FFFE
data	F001
Program (text) (PC)	0101-F000
Boot sector	0000-0100

Instruction Blocking

a type: (7 byte)	dest	s1	s2	opcode
ai type: (7 byte)	dest	s1	imm	opcode
br type: (7 byte)	dest	s1	s2	opcode
bri type: (7 byte)	dest	s1	imm	opcode
sp type: (1 byte)	-	-	-	opcode

j type: (3 bytes)	dest	-	-	opcode
I type: (5 bytes)	dest	-	imm	opcode

Code Fragments

Basic iteration to 10 with a conditional used as a while loop:

Address	Assembly	Machine code	Comments
0x0000	mov m, 0	0x00EF XXXX 0000 BF	//a set instruction
0x0007	L: addi m, m, 1	0x00EF 00EF 0001 80	//an addi instruction with a label
0x000E	bne m, 10, L	0x0007 00EF 000A BF	//a comparison instruction

Basic swap:

Address	Assembly	Machine code	Comments
0x0000	addi c, a, 0	0x00DF 00FF 0000 80	//set temp c to value of a
0x0007	addi a, b, 0	0x00FF 00EF 0000 80	//set a to value of b
0x000E	addi b, c, 0	0x00EF 00DF 0000 80	//set b to temp c (value of a)

Basic function call:

what is "a"? immediate?

Address	Assembly	Machine code	Comments
0x0000	push a	0x00FF XXXX XXXX 5E	//saves the value of a on the stack
0x0007	pushpc	0xXXXX XXXX XXXX 7F	//saves the PC on the stack
0x000E	j function	0x00EF 00DF 0000 80	//jumps to the function
0x0015	pop a	0x00FF XXXX XXXX 5F	//pops the value on the top of the stack into a

why?

is m in memory?

RelPrime

Address	Assembly	Machine code	Comments
0x0000	mov m, 2	0x00EF XXXX 0002 BF	//a set instruction
0x0007	addi sp, sp, -2	0x00FF 00FF FFFE 80	//an addi instruction
0x000E	L: addi m, m, 1	0x00EF 00EF 0001 80	//an addi instruction with a label
0x0015	pushpc	0xXXXX XXXX XXXX 7F	//a pushpc instruction that pushes the PC onto the stack
0x001C	j gcd	0x00CA XXXX XXXX A0	//a jump instruction to gcd
0x0023	bne m, -1, L	0x000E 00EF FFFF BF	//a comparison instruction
0x002A	addi sp, sp, 2	0x00FF 00FF 0004 80	// an addi instruction
0x0031	poppc	0xXXXX XXXX XXXX 7E	//a poppc instruction that sets the pc to the top value of the stack