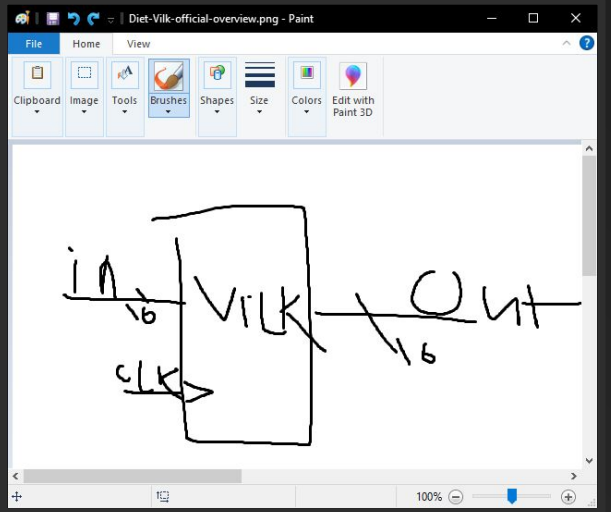
Diet Vilk - The CPU

Memory To Memory

- Loads Opcodes, memory addresses, and labels, one cycle at a time
- Loads memory addresses, not values, from the PC
- Uses registers to hold values after they are loaded. These registers are inaccessible to the programmer



Instruction Set

- Unique instructions
  - Comparing immediates
  - Pushra
  - Push, pop
  - Mov
  - All jumps and branches are absolute

Inst	Type	opcode	Usage	Logical Description	Description (verbose)
add	a	00000000 = 0x00	add dest s1 s2	Mem[dest] = Mem[s1] + Mem[s2]	Stores the sum of s1 and s2 into dest
sub	a	01100000 = 0x60	sub dest s1 s2	Mem[dest] = Mem[s1] - Mem[s2]	Stores the difference of s1 and s2 into dest
or	a	00000001 = 0x01	or dest s1 s2	Mem[dest] = Mem[s1] | Mem[s2]	Stores bitwise s1 OR s2 into dest
and	a	00000010 = 0x02	and dest s1 s2	Mem[dest] = Mem[s1] & Mem[s2]	Stores bitwise s1 AND s2 into dest
xor	a	00000100 = 0x04	xor dest s1 s2	Mem[dest] = Mem[s1] ^ Mem[s2]	Stores bitwise s1 XOR s2 into dest
srl	a	00111110 = 0x3e	srl dest s1 s2	Mem[dest] = Mem[s1] >> Mem[s2]	Stores s1 shifted right by s2 into dest
sll	a	00111100 = 0x3c	sll dest s1 s2	Mem[dest] = Mem[s1] << Mem[s2]	Stores s1 shifted left by s2 into dest
sri	ai	00111000 = 0x38	sri dest s1 imm	Mem[dest] = Mem[s1] >> imm	Stores s1 shifted right by imm into dest
sili	ai	01111100 =0x7c	sili dest s1 imm	Mem[dest] = Mem[s1] << imm	Stores s1 shifted left by imm into dest
addi	ai	10000000 = 0x80	addi dest s1 imm	Mem[dest] = Mem[s1] + imm	Stores the sum of s1 and imm into dest
andi	ai	10000010 = 0x82	andi dest s1 imm	Mem[dest] = Mem[s1] & imm	Stores bitwise s1 AND imm into dest
ori	ai	10000001 = 0x81	ori dest s1 imm	Mem[dest] = Mem[s1] | imm	Stores bitwise s1 OR imm into dest
xori	ai	10000100 = 0x84	xori dest s1 imm	Mem[dest] = Mem[s1] ^ imm	Stores bitwise s1 XOR imm into dest

j	j	10100000 = 0xa0	j label	PC = label	Sets the PC to the given value
ja	ja	00100000 = 0x20	ja s1	PC = Mem[s1]	Sets the PC to the value at the given address
push	ja	01011111 = 0x5f	push s1	SP -= 2; stack[SP] =Mem[s1]	Pushes the value at s1 onto the stack
pop	ja	01011110 = 0x5e	pop s1	Mem[s1] = stack[SP] SP += 2	Pops the value at the top of the stack into s1
beq	br	00110000 = 0x30	beq dest s1 s2	if(Mem[s1] == Mem[s2]) PC = dest	Sets the PC to the given value if s1 is equal to s2
blt	br	00111010 = 0x3a	blt dest s1 s2	if(Mem[s1] < Mem[s2]) PC = dest	Sets the PC to the given value if s1 is less than s2
bne	br	00111001 = 0x39	bne dest s1 s2	if(Mem[s1] != Mem[s2]) PC = dest	Sets the PC to the given value if s1 is not equal to s2
bge	br	00110100 = 0x34	bge dest s1 s2	if(Mem[s1] >= Mem[s2]) PC = dest	Sets the PC to the given value if s1 is greater than or equal to s2
beqi	bri	10110000 = 0xb0	beqi dest s1 imm	if(Mem[s1] == imm) PC = dest	Sets the PC to the given value if s1 is equal to imm
blti	bri	10111010 = 0xba	blti dest s1 imm	if(Mem[s1] < imm) PC = dest	Sets the PC to the given value if s1 is greater than imm
bnei	bri	10111001 = 0xb9	bnei dest s1 imm	if(Mem[s1] != imm) PC = dest	Sets the PC to the given value if s1 is not equal to imm
bgei	bri	10110100 = 0xb4	bgei dest s1 imm	if(Mem[s1] >= imm) PC = dest	Sets the PC to the given value if s1 is equal to or equal to imm
blei	bri	10101100 = 0xac	blei dest s1 imm	if(Mem[s1] <= imm) PC = dest	Sets the PC to the given value if s1 is less than or equal to imm
pushra	sp	01111111 = 0x7f	pushra	SP -= 2; stack[SP] = PC + 16	Pushes the return address of a subsequent method call onto the stack
mov	I	10111111 = 0xbf	mov dest imm	mem[dest] = imm	Sets the value at the dest memory address to be the given immediate imm

RTL

- Only unique type is SP, L, JA, J, JA Push, JA Pop

JA Pop	SP	L
OP = Mem[PC] PC = PC + 1		
A = Mem[PC] PC = PC + 2		
B = Mem[SP]	SP = SP - 2	B = Mem[PC] PC = PC + 2
SP = SP + 2 Mem[A] = B	A = PC + 1	Mem[B] = A
	Mem[SP] = A PC = PC - 2	

A	AI	BR
OP = Mem[PC] PC = PC + 1		
A = Mem[PC] PC = PC + 2		
B = Mem[PC] PC = PC + 2	B = Mem[PC] PC = PC + 2	B = Mem[PC] PC = PC + 2
A = Mem[A]	A = Mem[A]	A = Mem[A]
B = Mem[B]	aluOut = (A op B) Dest = Mem[PC]	B = Mem[B]
aluOut = (A op B) Dest = Mem[PC]	Mem[Dest] = aluOut PC = PC + 2	Dest = Mem[PC] PC = PC + 2
Mem[Dest] = aluOut PC = PC + 2		if(A op B) PC = Dest

BRI	J	JA	JA Push
OP = Mem[PC] PC = PC + 1			
A = Mem[PC] PC = PC + 2			
B = Mem[PC] PC = PC + 2	PC = A	A = Mem[A]	SP = SP - 2 B = Mem[A]
A = Mem[A]		PC = A	Mem[SP] = B
Dest = Mem[PC] PC = PC + 2			
if(A op B) PC = Dest			

Calling Conventions

Caller:

- Push any data to be saved
- Push any arguments
- Call pushra
- Jump to the function

After:

- Pop any returned values
- Pop any saved data

Callee:

- Pop RA
- Pop any arguments
- Do the function
- Push return values
- JA to RA

Code Snippets

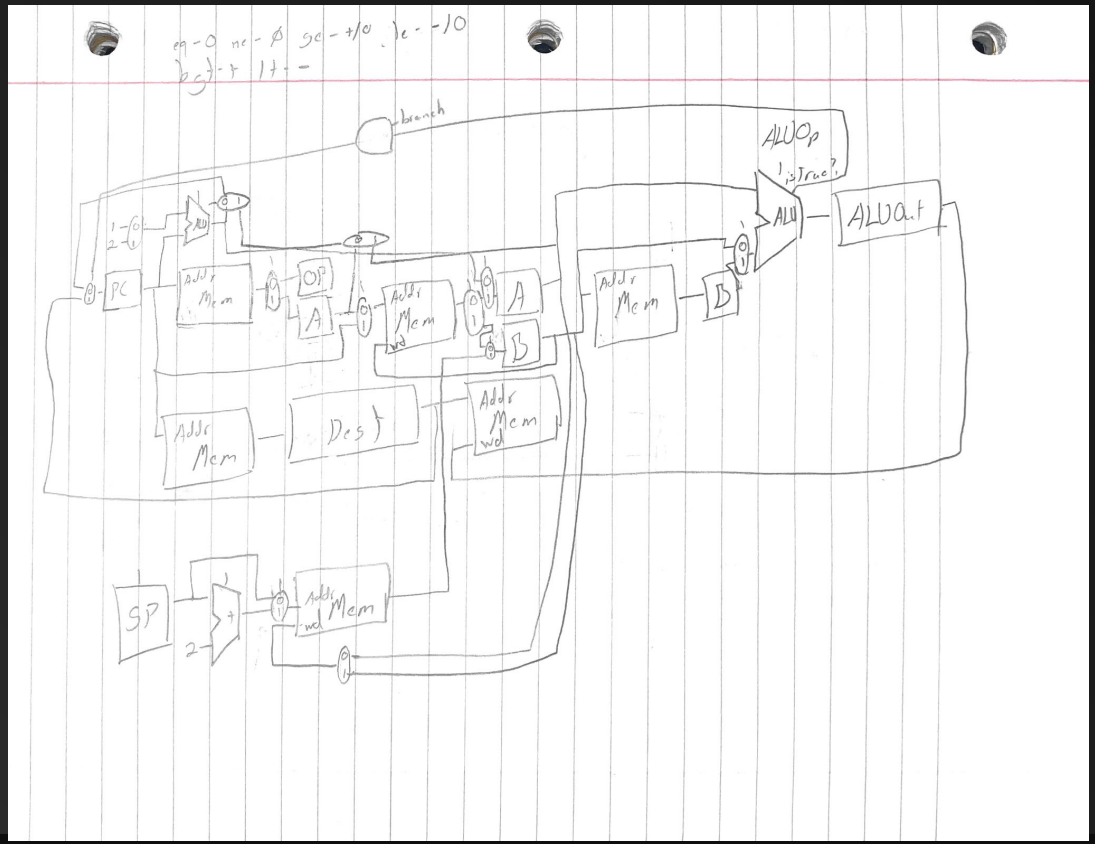
RelPrime

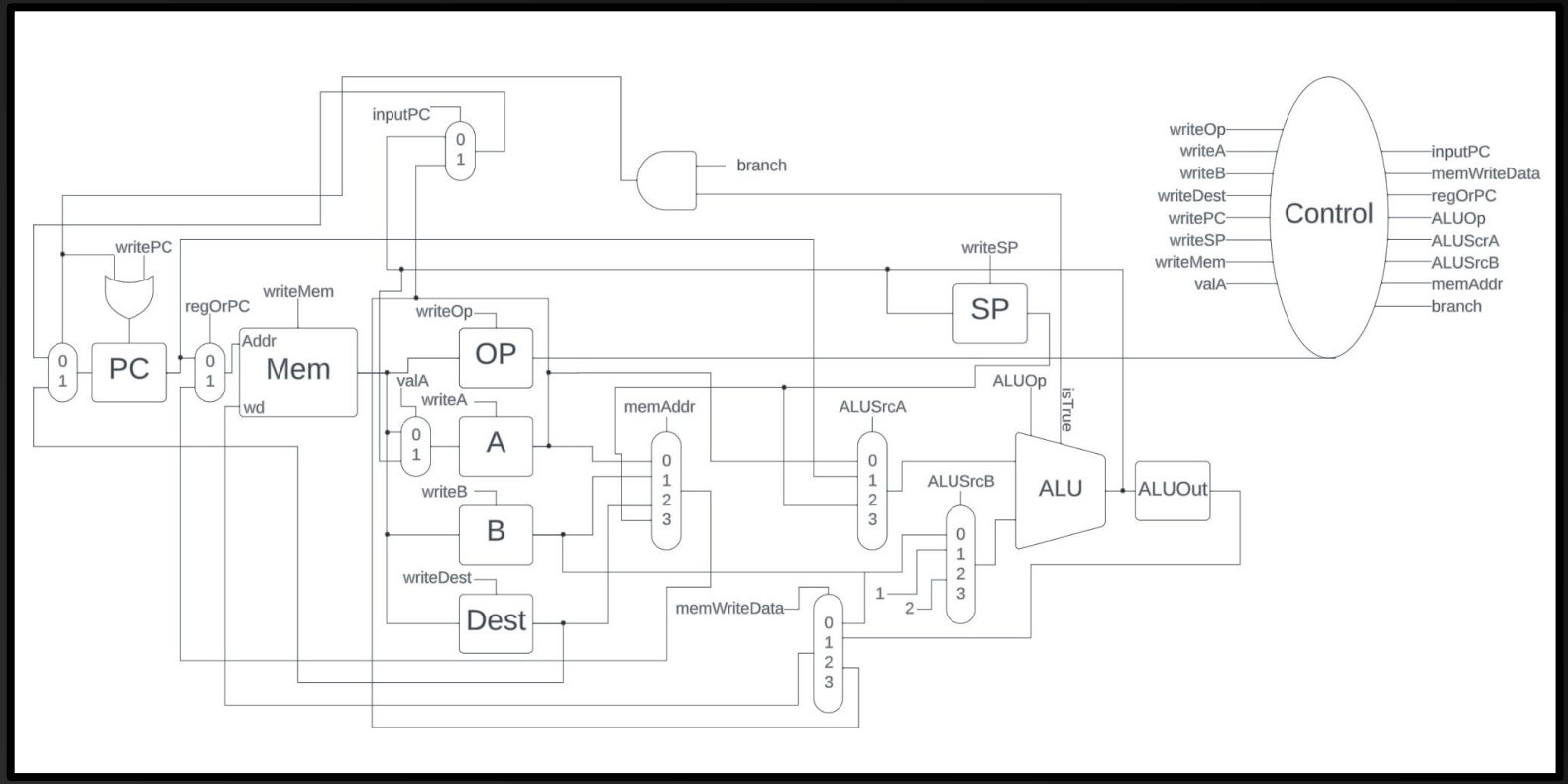
Address	Assembly	Machine code	Comments
0x0000	pop ra	FF00 5E	// pops the value on the stack into mem[ra]
0x0003	pop n	FF02 5E	// pops the value on the stack into mem[n]
0x0006	mov m, 2	FF04 0002 BF	//a set instruction
0x000b	Loop:push n	FF02 5F	//pushes the value at mem[n] onto stack
0x000e	push m	FF04 5F	//pushes the value at mem[m] onto stack
0x0011	pushra	7F	//pushes the PC of the instruction after a jump onto the stack
0x0012	j gcd	00CA A0	//a jump instruction to gcd
0x0015	pop c	FF06 5E	// pops the value on the stack into mem[c]
0x0018	beqi L, c, 1	0018 FFFF FF06 b0	//a comparison instruction
0x001f	addi m, m, 1	FF04 0001 FF04 80	//Set mem[m] to mem[m] + 1. with a label
0x0026	J Loop	000a a0	// jump instruction to the Loop tag
0x0029	L:push m	FF04 5F	//pushes the value at mem[m] onto stack
0x002c	ja ra	FF00 7E	//a jump that goes to the address stored in mem[ra]

GCD

Address	Assembly	Machine code	Comments
0x0000	pop ra	FF08 5E	// pops the value on the stack into mem[ra]
0x0003	pop b	FF0a 5E	// pops the value on the stack into mem[b]
0x0006	pop a	FF0c 5E	// pops the value on the stack into mem[a]
0x0009	bnei LOOP, a, 0	0016 0000 FF0c B9	//if (a != 0) Go to LOOP
0x0010	push b	FF0a 5F	//pushes the value at mem[b] onto stack
0x0013	ja ra	FF08 20	//a jump that goes to the address stored in mem[ra]
0x0016	LOOP: bge ELSE, b, a	0051 FF0c FF0a 34	//if (b >= a) Go to ELSE. with a LOOP label
0x001D	sub a, a, b	FF0c FF0c FF0c 60	//Set mem[a] to mem[a] - mem[b]
0x0024	j NOEL	FF08 a0	A unconditional jump to NOELSE
0x0027	ELSE: sub b, b, a	FF0a FF0c FF0a 60	//Set mem[b] to mem[b] - mem[a]. With a ELSE label
0x002E	NOEL: bnei LOOP, b, 0	0016 0000 FF0a B9	//if (b != a0 Go to LOOP. with a NOELSE label
0x0035	push a	FF0c 5F	//pushes the value at mem[a] onto stack
0x0038	ja ra	FF08 20	//a jump that goes to the address stored in mem[ra]

The Original Datapath



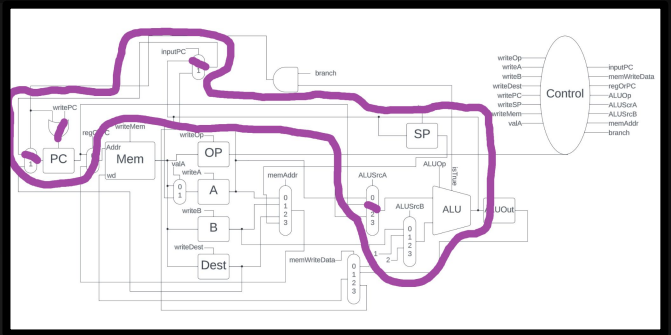


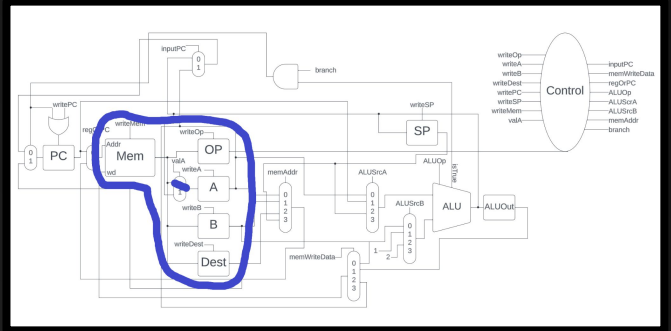
THE DATAPATH

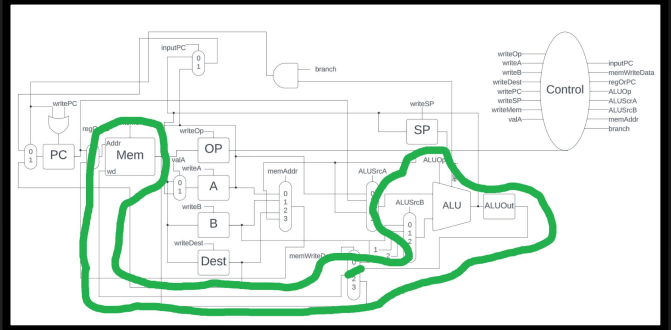
"It's not my fault you guys made such a baller datapath" - Garrett Doolittle

Testing

- Build and test components
- Connect components into 3 different stages
- Test the different stages
- Connect the stages







Unique Challenge

Memory

Super Fast RelPrime

Assembly	Machine code	Assembly	Machine code
addi n -2 0	80 fffe 0000 ff02	j BD	a0 00b0
mov v 2	bf 0002 ff04	pop m	5e ff06
sili m n 15	7c ff02 000f ff06	bnei END m 0	b9 ff06 0000 00ee
bnei END m 0	b9 ff06 0000 00ee	mov v 7	Bf 0007 ff04
mov v 3	Bf 0003 ff04	push n	5f ff02
push n	5f ff02	push v	5f ff04
push v	5f ff04	pushra	7f
pushra	7f	j BD	a0 00b0
j BD	a0 00b0	pop m	5e ff06
pop m	5e ff06	bnei END m 0	b9 ff06 0000 00ee
bnei END m 0	b9 ff06 0000 00ee	mov v 11	Bf 000b ff04
mov v 5	Bf 0005 ff04	push n	5f ff02
push n	5f ff02	push v	5f ff04
push v	5f ff04	pushra	7f
pushra	7f		

Assembly	Machine code	Assembly	Machine code
j BD	a0 00b0	pop m	5e ff06
pop m	5e ff06	bnei END m 0	b9 ff06 0000 00ee
bnei END m 0	b9 ff06 0000 00ee	BD pop ra	5e ff08
mov v 13	Bf 000d ff04	pop b	5e ff0a
push n	5f ff02	pop a	5e ff0c
push v	5f ff04	mov i 11	bf 000b ff0e
pushra	7f	F sll x b i	3c ff0a ff0e ff10
j BD	a0 00b0	bge E x a	34 ff10 ff0c 00da
pop m	5e ff06	A sub a a x	60 ff0c ff10 ff0c
bnei END m 0	b9 ff06 0000 00ee	bge A a x	34 ff0c ff10 00cc
mov v 17	Bf 0011 ff04	E addi i i -1	80 ff0e ffff ff0e
push n	5f ff02	bnei F i -1	b9 ff0e ffff 00be
push v	5f ff04	push a	5f ff0c
pushra	7f	ja ra	20 ff08
j BD	a0 00b0	END addi t v 0	80 ff04 0000 f010

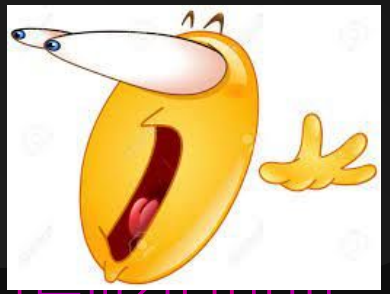
Performance

Regular RelPrime

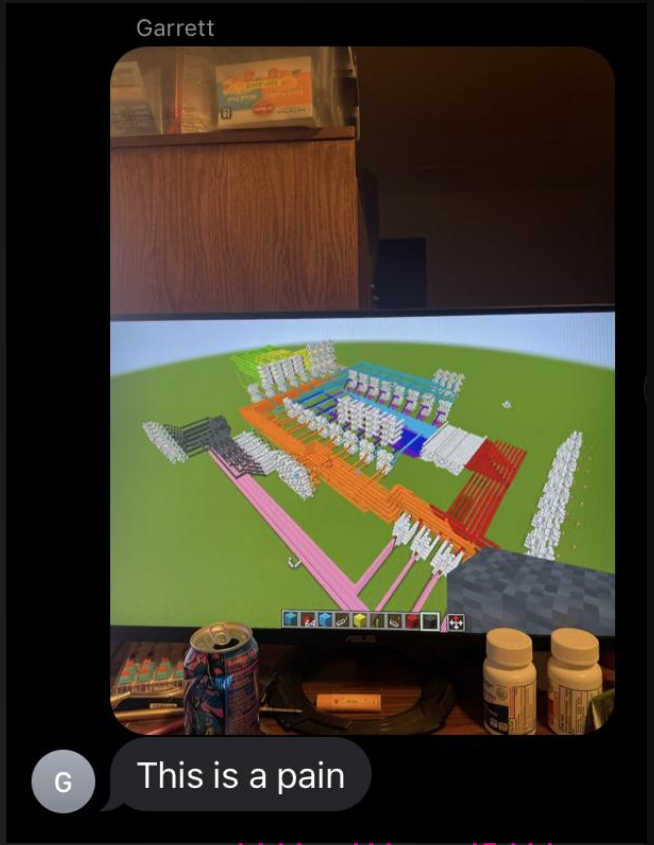
Input: 0x13b0
Total instructions: 40876
Total cycles: 234,872
CPI: 5.746
CT: 10.3 ns
Execution Time: 2.41 ms

Fast RelPrime

Input: 0x13b0
Total executed instructions: 294
Total Cycles: 1801
CPI: 6.13
CT: 10.3 ns
Execution time: 18.6 µs



Any Questions?



Green Sheet

Inst	T	Op	Usage	Logical Description	Description (verbose)
add	a	00	add dest s1 s2	Mem[dest] = Mem[s1] + Mem[s2]	Stores the sum of s1 and s2 into dest
sub	a	60	sub dest s1 s2	Mem[dest] = Mem[s1] - Mem[s2]	Stores the difference of s1 and s2 into dest
or	a	01	or dest s1 s2	Mem[dest] = Mem[s1] | Mem[s2]	Stores bitwise s1 OR s2 into dest
and	a	02	and dest s1 s2	Mem[dest] = Mem[s1] & Mem[s2]	Stores bitwise s1 AND s2 into dest
xor	a	04	xor dest s1 s2	Mem[dest] = Mem[s1] ^ Mem[s2]	Stores bitwise s1 XOR s2 into dest
srl	a	3e	srl dest s1 s2	Mem[dest] = Mem[s1] >> Mem[s2]	Stores s1 shifted right by s2 into dest
sll	a	3c	sll dest s1 s2	Mem[dest] = Mem[s1] << Mem[s2]	Stores s1 shifted left by s2 into dest
sri	ai	38	sri dest s1 imm	Mem[dest] = Mem[s1] >> imm	Stores s1 shifted right by imm into dest
sili	ai	7c	sili dest s1 imm	Mem[dest] = Mem[s1] << imm	Stores s1 shifted left by imm into dest
addi	ai	80	addi dest s1 imm	Mem[dest] = Mem[s1] + imm	Stores the sum of s1 and imm into dest
andi	ai	82	andi dest s1 imm	Mem[dest] = Mem[s1] & imm	Stores bitwise s1 AND imm into dest
ori	ai	81	ori dest s1 imm	Mem[dest] = Mem[s1] | imm	Stores bitwise s1 OR imm into dest
xori	ai	84	xori dest s1 imm	Mem[dest] = Mem[s1] ^ imm	Stores bitwise s1 XOR imm into dest
j	j	a0	j label	PC = label	Sets the PC to the given value
ja	ja	20	ja s1	PC = Mem[s1]	Sets the PC to the value at the given address
push	ja	5f	push s1	SP -= 2; stack[SP] =Mem[s1]	Pushes the value at s1 onto the stack
pop	ja	5e	pop s1	Mem[s1] = stack[SP] SP += 2	Pops the value at the top of the stack into s1
beq	br	30	beq dest s1 s2	if(Mem[s1] == Mem[s2]): PC = dest	Sets the PC to dest if s1 is equal to s2
blt	br	3a	blt dest s1 s2	if(Mem[s1] < Mem[s2]): PC = dest	Sets the PC to dest if s1 is less than s2
bne	br	39	bne dest s1 s2	if(Mem[s1] != Mem[s2]): PC = dest	Sets the PC to dest if s1 is not equal to s2
bge	br	34	bge dest s1 s2	if(Mem[s1] >= Mem[s2]): PC = dest	Sets the PC to dest if s1 is greater than or equal to s2
beqi	bri	b0	beqi dest s1 imm	if(Mem[s1] == imm): PC = dest	Sets the PC to dest if s1 is equal to imm
blti	bri	ba	blti dest s1 imm	if(Mem[s1] < imm): PC = dest	Sets the PC to dest if s1 is greater than imm
bnei	bri	b9	bnei dest s1 imm	if(Mem[s1] != imm): PC = dest	Sets the PC to dest if s1 is not equal to imm
bgei	bri	b4	bgei dest s1 imm	if(Mem[s1] >= imm): PC = dest	Sets the PC to dest if s1 is greater than or equal to imm
bgti	bri	a4	bgti dest s1 imm	if(Mem[s1] > imm): PC = dest	Sets the PC to dest if s1 is greater than imm
blei	bri	ac	blei dest s1 imm	if(Mem[s1] <= imm): PC = dest	Sets the PC to dest if s1 is less than or equal to imm
pushra	sp	7f	pushra	SP -= 2; stack[SP] = PC + 16	Pushes the return address onto the stack
mov	I	bf	mov dest imm	mem[dest] = imm	Sets the value at the dest memory address to be imm

Performance Testing

This design was tested using a combination of unit tests on individual components and overall system tests written in Verilog and executed in modelsim. Final performance figures were concluded from the execution of a given relative prime algorithm.

Instruction Types & Blocking

All instructions are specified using an 8-bit opcode which is frequently represented in hex.

A - arithmetic instructions. 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, and or use this type of instruction.

7 bytes	16	16	16	8
A:	**dest**	**s2**	**s1**	**opcode**

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

7 bytes	16	16	16	8
AI:	**dest**	**imm**	**s1**	**opcode**

BR - Branch instructions utilizing a 16 bit Label, and 2 16 bit addresses. The BR instructions will compare the value stored in memory at address s1 with the value at address s2 and either branch to a given tag or just move on to the next line in the newPC. Instructions of this type are blt, bge, bne, and beq.

7 bytes	16	16	16	8
BR:	**dest**	**s2**	**s1**	**opcode**

BRI - Similar to BR instructions, take a 16 bit immediate, a 16 bit label, and a 16 bit address. The BR instructions compare an immediate with the value stored in memory and either branch to a given tag or just move on to the next instruction. Includes beqi, bnei, blti, bgei, bgti, blei.

7 bytes	16	16	16	8
BRI:	**dest**	**imm**	**s1**	**opcode**

J - jump instructions that will take a 16 bit label. It unconditionally jumps to the given label.

3 bytes	16	8
J:	**label**	**opcode**

JA - jump instructions that will take a 16 bit address. It jumps unconditionally to the location stored in the given address.

3 bytes	16	8
JA:	**s1**	**opcode**

SP - Instruction type used to push and pop addresses on the stack. Used for the push and pop.

1 byte	8
SP:	**opcode**

L - Instruction type used to to load an immediate into memory. Loads a 16 bit immediate into a 16 bit destination address.

5 bytes	16	16	8
L:	**dest**	**imm**	**opcode**

Addressing Modes

The processor uses direct addressing.

Memory Map

Output Address	F010
Input Address	FFFE
Stack space (SP)	FFFD
Data (variable space)	F012
Program (text) (PC)	0101-F00E
Boot sector	0000-0100

Input/Output

The processor handles i/o operation though the use of reserved memory addresses. The memory module is contained within an io wrapper that diverts the signals when necessary.

Code Fragments

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

Address	Assembly	Machine code	Comments
0x0000	mov m, 0	0xFF00 0000 BF	//a set instruction
0x0005	L: addi m, m, 1	0xFF00 0001 FF00 80	//an addi instruction with a label
0x000C	bnei L, m, 10	0x0005 000A FF00 BF	//a comparison instruction

Code Fragments

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

Address	Assembly	Machine code	Comments
0x0000	mov m, 0	0xFF00 0000 BF	//a set instruction
0x0005	L: addi m, m, 1	0xFF00 0001 FF00 80	//an addi instruction with a label
0x000C	bnei L, m, 10	0x0005 000A FF00 BF	//a comparison instruction

If…else:

Address	Assembly	Machine code	Comments
0x0000	IF: bnei ELSE, m, 0	000E 0000 FF00 B9	//go to ELSE if m != 0
0x0007	addi c, c, 1	FF02 0001 FF02 80	//if no branch (m = 0), c += 1
0x000E	ELSE:addi c, c, -1	FF02 FFFF FF02 80	//if branch (m != 0), c -= 1

Basic swap:

Address	Assembly	Machine code	Comments
0x0000	addi c, a, 0	FF04 0000 FF00 80	//set temp c to value of a
0x0007	addi a, b, 0	FF00 0000 FF02 80	//set a to value of b
0x000E	addi b, c, 0	FF02 0000 FF04 80	//set b to temp c (value of a)

Basic function call with an input value and return value in a:

Address	Assembly	Machine code	Comments
0x0000	push a	FF00 5E	//saves the value at a on the stack
0x0003	pushra	7F	//pushes the PC of the instruction after a jump onto the stack
0x0004	j function	00DD A0	//jumps to the function
0x000B	pop a	FF00 5F	//pops the value on the top of the stack into a

Basic recursive addition:

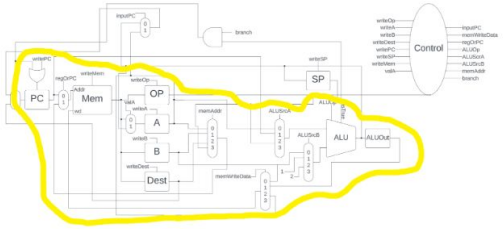
Address	Assembly	Machine code	Comments
0x0000	FAD: pop ra	FF00 5E	// pops the value on the stack into mem[ra]
0x0003	pop m	FF02 5E	// pops the value on the stack into mem[m]
0x0006	pop tot	FF04 5E	// pops the value on the stack into mem[tot]
0x0009	bgti DB, m, 0	0016 FF02 0000 A4	//if (m > 0) Go to DB
0x0010	push tot	FF04 5F	//pushes the value at mem[tot] onto stack
0x0013	ja ra	FF00 20	//a jump that goes to the address stored in mem[ra]
0x0016	DB: add tot, tot, m	FF04 FF02 FF04 00	//Set mem[tot] to mem[tot] + mem[m]. With a DB label
0x001D	addi m, m, -1	FF02 FFFF FF02 80	//Set mem[m] to mem[m] - 1
0x0024	push tot	FF04 5F	//pushes the value at mem[tot] onto stack
0x0027	push m	FF02 5F	//pushes the value at mem[m] onto stack
0x002A	pushra	7F	//pushes the PC of the instruction after a jump onto the stack
0x002B	j FAD	00DD A0	A unconditional jump to FAD
0x003E	ja ra	FF00 20	//a jump that goes to the address stored in mem[ra]

Component Descriptions

Component	Inputs	Outputs	Behavior	RTL Symbols
Register (16bit)	inputValue [15:0] CLK [0:0] reset [0:0]	outputValue [15:0]	On the rising edge of the clock (CLK) the register reads the input_value port and outputs the new value. If the reset signal is high on the rising clock edge the register wipes its data and outputs 0x0000 instead.	PC, SP, A, B, IR,aluOut, Dest
Register (8bit)	inputValue [7:0] CLK [0:0] reset [0:0]	outputValue [7:0]	Similar to the 16 bit register, on the rising edge of the clock, the register will read the inputValue and output a new value. The reset signal will input 0x0000. This will currently only be used to read the opcode.	OP
Memory	CLK [0:0] memWrite [0:0] inputAddress [15:0] inputValue [15:0]	outputValue [15:0]	On the rising edge of the clock, the memory reads the memWrite and if it is 1, it will write the value given at inputValue into the address at inputAddress, if memWrite is not 1 (it is 0), the memory will output the value in memory at the address given by inputAddress to outputAddress.	Mem
ALU	inputA [15:0] inputB [15:0] ALUOp [2:0]	outputValue [15:0]	The ALU takes in two 16 bit immediate values and does the operation indicated by the opcode and outputs a 16 bit immediate.	(A op B)

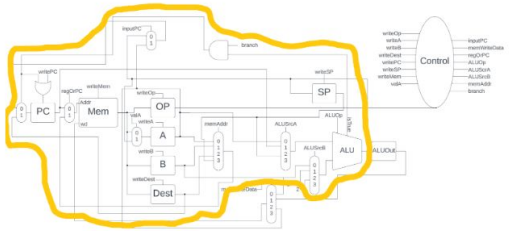
For ALUOp values, see Control Signals (Page 21)

Stage 4



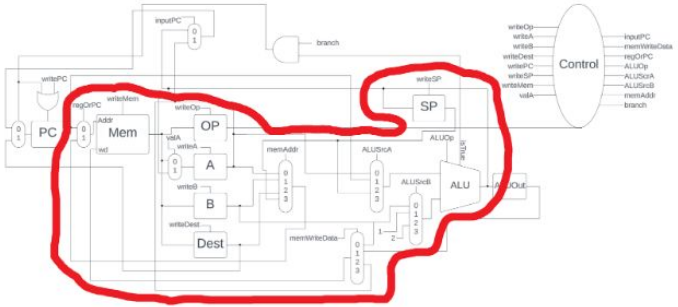
Connecting stages 1, 2, and 3 to make the A and AI type instruction.
Connect the PC to the address input of the memory component. Verify the OP register contains the correct opcode, and the registers A and B contain the desired addresses. Load the values of A and B into their respective registers if necessary. Load the Dest address into the Dest register and load that address into the address input for memory. Perform the required operation with the ALU, and write the value of ALUOut to Dest. Verify that Dest contains this new value.

Stage 5



Creating the BR and BRI instruction types.
The first thing we need to test for the ALU is the isTrue? part of it to ensure that branching will occur. This will have us compare two values and make sure that the comparison returns 1 on the designated branch instruction. To test this we just need to make up two values that are greater than, greater than or equal, less than, less than or equal, equal, or not equal. After this we need to make sure that the PC is only overwritten if we're branching. To do this we just have a branch and an and gate to make sure we are branching. This should ensure that only on a branch, we load the label in dest.

Stage 6

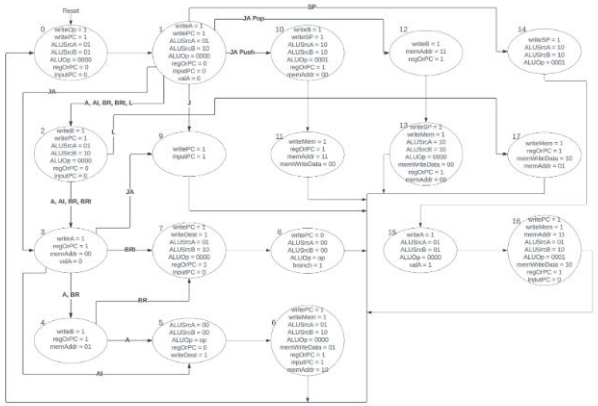


Integrate SP for the JA, SP, and L instruction types.
Use the ALUSrc selectors to increment/decrement SP by 2, and load the result back into SP.
Verify SP now contains the desired value.

Stage 7

At this point all the components should be implemented,this stage is to go back over and make sure all the instructions work as intended.

Finite State Machine



Control Unit Design

The control unit should be implemented in a fashion very similar to a standard multi-cycle control. The control module needs inputs for the opcode, the clock, and a reset, and outputs for each control signal. There will also be an integer variable for the current state. It then uses a switch case statement with input state on the rising edge of the clock to adjust certain signals based on the state, and opcode if necessary. Before the switch case, all the signals get set to 0 so that they do not stay active if they were active in a previous cycle. On the falling edge of the clock, we will transition to the next state using another switch case and by checking opcode if necessary. To test this module, manually input the opcode of each instruction type and check if the control signals are what is expected for each cycle.

Assembler

There are a few conventions for the assembler:

- No commas, just spaces in between instructions, memory addresses, integers, etc.
- All labels must start with a capital letter.
- All memory addresses you're loading must be lowercase. Note that the memory addresses are global. If you have a memory address labeled 'a' in one function and 'a' in another, the later 'a' will overwrite the first 'a' even though they're in separate functions.
- The label "END" will take you to a specific location in memory. This address was to just see the output when it returns without I/O.
- Use the words "Input" and "Output" as memory addresses to access their respective functions. (Capitalization matters.)

Performance Data

Total bytes required to store and run Euclid's algorithm and relPrime:

- Storing code 216 Bytes
- Holding variables 12 Bytes
- Maximum on the stack 6 Bytes

Top-level Entity Name	DietVilk
Family	Cyclone IV E
Total logic elements	715 / 6,272 (11 %)
Total registers	131
Total pins	34 / 92 (37 %)
Total virtual pins	0
Total memory bits	16,384 / 276,480 (6 %)

	Fmax	Restricted Fmax	Clock Name	Note
1	97.5 MHz	97.5 MHz	CLK	