CSSE 232 COMP ARCH 1

MISC-V DESIGN DOCUMENT

TEAM ORANGE:

Drew Kilner, Yash Anand, Julian Ferrer Rodriguez, Eli Granade, Ziyu Xie

Contents

Introduction:

MISC-V, the Minimal Instruction Set Computer, is a Load\Store based assembly processor with a branch/jump delay slot that utilizes pipelining to streamline the execution of the Euclid's Algorithm Test bench. Our design philosophy centered on balancing speed of execution with ease of assembly programming. It utilizes a symbol-based assembly language for ease of programming, and a branch delay slot for speed. It's minimal instruction set is also tuned to prioritize Euclid's Algorithm, which uses a variety of important commands that the average programmer would use.

Register Names

Register	Name	Description	Saver
x0	zero	This register is always zero	-
x1	ra	This is the return address	caller
x2	sp	This is the stack pointer	-
x3	at	This is the assembler temporary	-
x4	a0	This is a temporary register that is used for function inputs and function return values	caller
x5	a1		
x6	s0	These are usable saved registers	callee
x7	s1		

Procedure Call Convention

Caller saves ra, t, and a registers as needed. Moves sp. Do `\/.`

Callee saves s registers, moves sp. Puts return values in a registers and restores all s registers. Move sp back. Do `/\.`

```
Caller restores ra, t, and a registers.
```

Base Instructions:

inst	fmt	func	opcode	description
+	R	0000	000	R[rd] = R[rs1] + R[rs2]
-	R	0001	000	R[rd] = R[rs1] – R[rs2]
|	R	0010	000	R[rd] = R[rs1] | R[rs2]
&	R	0011	000	R[rd] = R[rs1] & R[rs2]
+_	I	00	001	R[rd] = R[rs1] + SE(imm)
<<_	I	01	001	R[rd] = R[rs1] << imm
>>_	I	10	001	R[rd] = R[rs1] >> imm
X|_	I	11	001	R[rd] = R[rs1] ^ SE(imm)
<-	M		010	R[rd] = M[R[rs1] + SE(imm)]
->	M		011	M[R[rs1] + SE(imm)] = R[rd]
Y=	Y		100	If(rs1==rs2) PC += SE(imm) << 1
Y<	Y		101	If(rs1<rs2) PC += SE(imm) << 1
\/	J		110	R[rd] = PC+2 PC += SE(imm) << 1
/\	J		111	PC = R[rd]

Writing Instructions:

Type	Layout
R	rs1 (op) rs2, rd
I	rs1 (op) imm, rd
M	rd (op) rs1+imm
Y	rs1 (op) rs2, imm
J	rd (op) imm

15 14	13 12	11 9	8 6	5 3	2 0	Type
func[3:2]	func[1:0]	rs2	rs1	rd	op	R
func	imm[4:0]		rs1	rd	op	I
imm[6:3]		rs2	rs1	imm[2:0]	op	Y
imm[6:0]			rs1	rd	op	M
imm[9:0]				rd	op	J

Instruction Definitions

Instruction	Definitions
+	Takes the two registers with the operand between, adds the values, then puts it into the third register.
-	Subtracts the register after the operand from the first, then puts it into the last.
|	Takes the two registers with the operand between, bitwise or of the values, then puts it into the third register.
&	Takes the two registers with the operand between, bitwise and the values, then puts it into the third register.
+_	Takes the first register and the immediate after the operand, adds them and puts them into second register.
<<_	Takes the first register and shifts it left the immediate number of times, then puts it in the second register.
>>_	Takes the first register and shifts it right the immediate number of times, then puts it in the second register. This operation does not sign extend the shifted bits.
X|_	Takes the first register and xoris it with the sign extended immediate, then puts it into the second register.
<-	Loads a 16 bit word out of the memory address equal to the second register plus the immediate, and puts it in the first register
->	Takes a 16 bit word out of the first register and puts it in the memory address equal to the second register plus the immediate
Y=	If the first two registers are equal, then then bit shift the immediate left and add it to the program counter
Y<	If the first register is less than the second register, then then bit shift the immediate left and add it to the program counter
\/	Used to step into function. Put the current PC into the first register, add two to it, then bit shift the immediate left and add that to the PC
/\	Used to step out of function. Put the first register into the PC.

Example Commands in both MISC-V assembly and machine code:

assembly	comments
x6 Y= x7, skip x0 Y= x0, label skip: …continue code without branch	//branch not equal x6, x7
x6 Y< x7, skip x0 Y= x0, label skip: …continue code without branch	//branch if x6 >= x7

Simple function implementations:

Simple if-else statements:

```
If (a <= b) {
a++;
else {
a = a + b;
}
```

Assembly (a = x6, b = x7):	Machine code:
x7 Y< x6, 3	Y-Type 0000 |110|1 11|01 1|101
x6 +_ 1, x6	I-Type 00|00 001|1 10|11 0|001
x0 \/ 1	J-Type 0000 0000 1|000 0|110
x6 + x7, x6	R-Type 00|00 |111|1 10|11 0|000

Simple loop (adds all elements in array):

```
int[] a = {1, 3, 5, 7, 9};
int sum = 0;
for(int i = 0; i < a.length; i++){
sum = sum + a[i];
}
```

Assembly (address of array a in mem = x4, array length = x5):	Machine code:
zero + x4, x6	R-Type 00|00 |000|1 00|11 0|000
zero + x5, x7	R-Type 00|00 |000|1 00|11 1|000
x5 Y= zero, 5	Y-Type 0000 |110|1 01|10 1|100
x7 <- x6+0	M-Type 0000 000|1 10|10 0|010
x6 +_ 2, x6	I-Type 00|00 010|1 10|11 0|001
x5 +_ -1, x5	I-Type 00|11 111|1 10|10 1|001
x0 \/ -4	J-Type 1111 1111 00|00 0|110

Simple while loop:

```
While(a < b) {

a = a<<;

}
```

Assembly(a = x6, b = x7):	Machine code:
x7 Y< x6, 3	Y-Type 0000 |110|1 11|01 1|101
x6 <<_ 1	I-Type 01|00 011|1 10|11 1|001
x0 \/ -2	J-Type 1111 1111 10|00 0|110

I/O

For I/O operations, we dedicated the specific memory address 0x0000, which would allow for inputs and outputs using the -> and <- commands.

Euclid's algorithm in MISC assembly:

Minimum instruction Euclid algorithm with BDS:

```
relPrime:   a0 <- x0+0
            sp +_ -8, sp
            ra -> sp+0
            s0 -> sp+2
            s1 -> sp+4
            a0 -> sp+6
            x0 +_ 2, s0
            x0 +_ 1, s1
loop1:      s0 +_ 0, a1
            a0 <- sp+6
            ra \/ gcd
            x0 + x0, x0
            a0 Y= s1, done
            x0 + x0, x0
            s0 +_ 1, s0
            x0 \/ loop1
            x0 + x0, x0
done:       s0 +_ 0, a0
            ra <- sp+0
            s0 <- sp+2
            s1 <- sp+4
            sp +_ 8, sp
            a0 -> x0+0
            ra \/ relPrime
            x0 + x0, x0
gcd:        sp +_ -2, sp
            ra -> sp+0
            a0 Y= x0, returnb
            x0 + x0, x0
loop2:      a1 Y= x0, returna
            x0 + x0, x0
            a1 Y< a0, agreater
            x0 + x0, x0
            a1 - a0, a1
            x0 \/ loop2
            x0 + x0, x0
agreater:   a0 - a1, a0
            x0 \/ loop2
            x0 + x0, x0
returnb:    a1 +_ 0, a0
returna:    ra <- sp+0
            sp +_ 2, sp
            ra /\ 1
            x0 + x0, x0
```

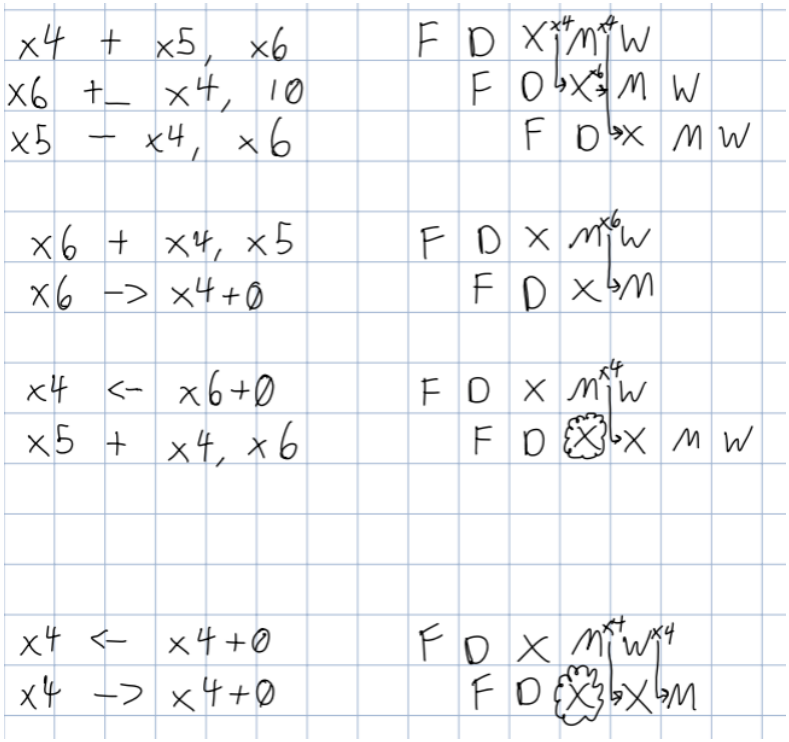
address	assembly	Instruction	comments
0x00	a0 <- x0+0	3C91	//begin relprime
0x01	sp +_ -8, sp	0022	
0x02	ra -> sp+0	3091	
0x03	s0 -> sp+2	008B	
0x04	s1 -> sp+4	04B3	
0x05	a0 -> sp+6	08BB	
0x06	x0 +_ 2, s0	0CA3	
0x07	x0 +_ 1, s1	0431	
0x08	s0 +_ 0, a1	0239	//begin loop1
0x09	a0 <- sp+6	01A9	
0x0A	ra \/ gcd	0CA2	
0x0B	x0 + x0, x0	038E	//BDS
0x0C	a0 Y= s1, done	0000	
0x0D	x0 + x0, x0	0F2C	
0x0E	s0 +_ 1, s0	0000	
0x0F	x0 \/ loop1	03B1	
0x10	x0 + x0, x0	FE46	//BDS
0x11	s0 +_ 0, a0	0000	//begin done
0x12	ra <- sp+0	01A1	
0x13	s0 <- sp+2	008A	
0x14	s1 <- sp+4	04B2	
0x15	sp +_ 8, sp	08BA	
0x16	a0 -> x0+0	0023	
0x17	ra /\ relPrime	FA8E	
0x18	x0 + x0, x0	0000	//BDS
0x19	sp +_ -2, sp	3C91	//begin gcd
0x1A	ra -> sp+0	008B	
0x1B	a0 Y= x0, returnb	1124	
0x1C	x0 + x0, x0	0000	//BDS
0x1D	a1 Y= x0, returna	115C	
0x1E	x0 + x0, x0	0000	//BDS
0x1F	a1 Y< a0, agreater	096D	
0x20	x0 + x0, x0	0000	//BDS
0x21	a1 - a0, a1	1968	
0x22	x0 \/ loop	FEC6	
0x23	x0 + x0, x0	0000	//BDS
0x24	a0 - a1, a0	1B20	//begin agreater
0x25	x0 \/ loop	FE06	
0x26	x0 + x0, x0	0000	//BDS
0x27	a1 +_ 0, a0	0161	//begin returnb
0x28	ra <- sp+0	008A	//begin returna
0x29	sp +_ 2, sp	0491	
0x2A	ra /\ 1	004F	
0x2B	x0 + x0, x0	0000	//BDS

Possible Hazards and Forwarding:

The following are all possible circumstances where forwarding or hazards cause problems. A branch and jump delay slot have been implemented to fix these. When a jump or branch is run the following instruction will also always run.

R-Type, I-Type, and M-Type instructions:

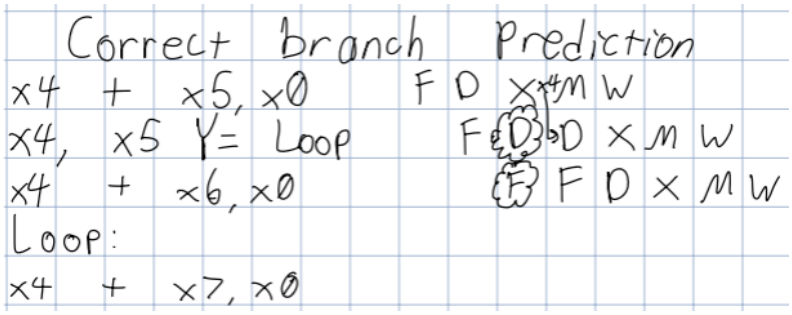
Forwarding after the execute stage, memory stage, and write-back stage.



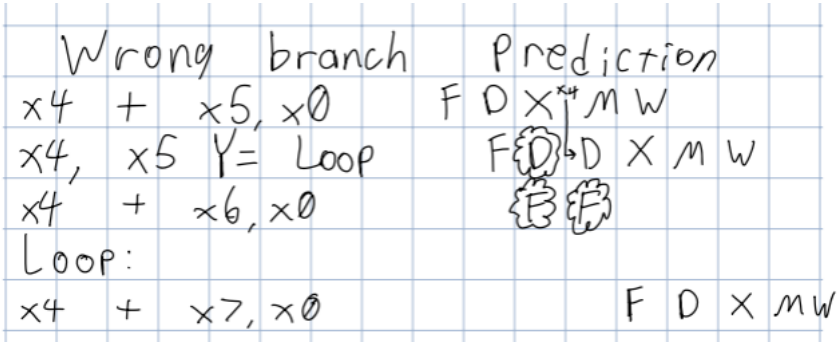
Loading data and immediately using it will need to stall, storing data that was just changed will need to be forwarded from write-back.

Y-Type instructions:

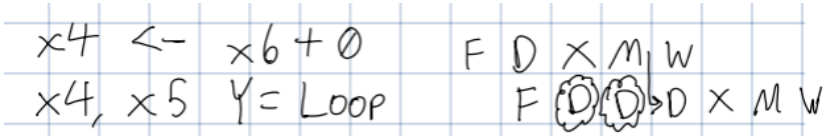
On a correct branch prediction, if data for branch is updated in cycle before it must stall



On an incorrect branch prediction, will have to flush the instruction that was predicted



Loading from memory then using in a branch will cause 2 stalls.



J-Type instructions:

Jump instructions will always delay 1 instruction.

Instruction	R-Type	I-Type	M-Type	Y-Type	J-Type
Rd of previous R or I type instruction used	Forward from MEM	Forward from MEM	Forward to from MEM to EX for address, and WB to	Stall in decode and then forward from	N/A

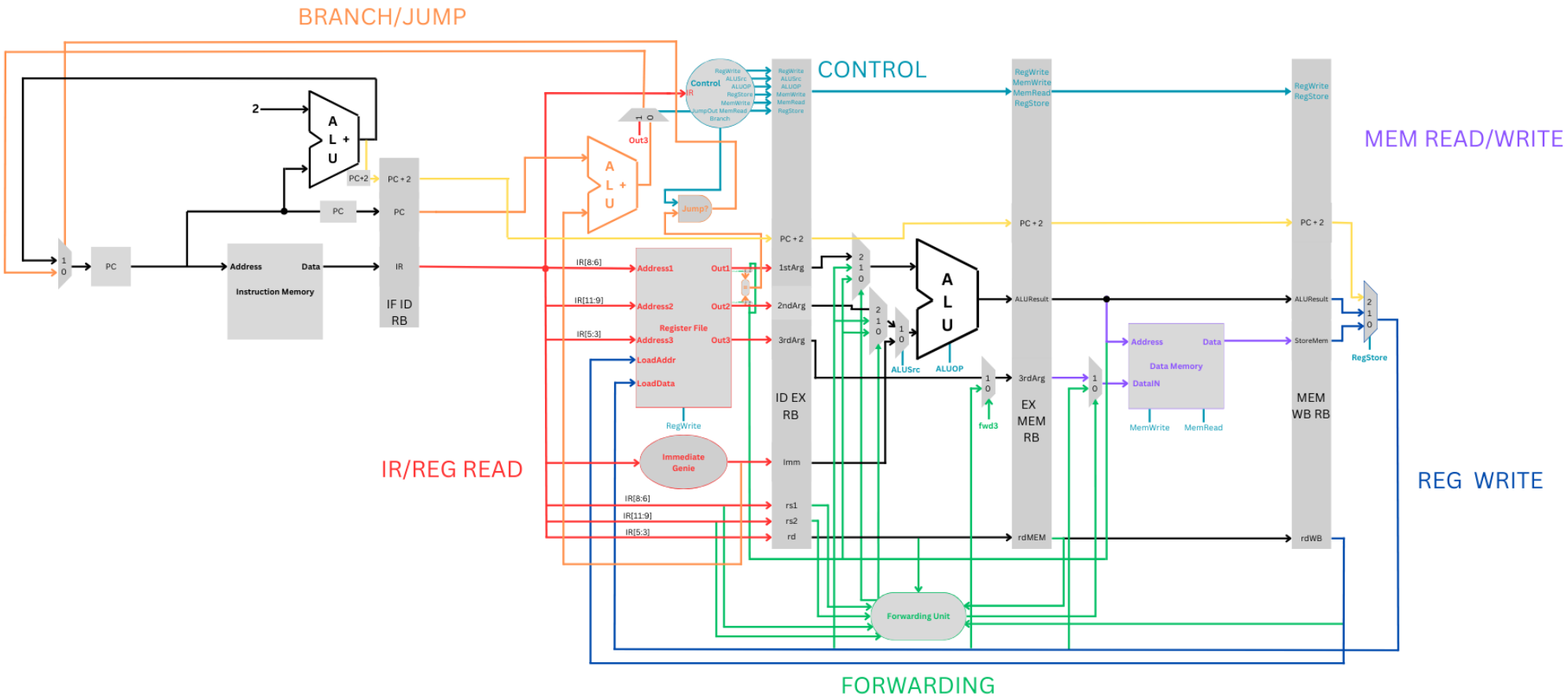
			MEM for load data	MEM to decode	
Rd of R or I type instruction 2 before used	Forward from WB	Forward from WB	Forward from WB to EX for address or data	Forward from MEM to decode	N/A
Rd of previous <- instruction used	Stall in EX, then forward from WB	Stall in EX, then forward from WB	Stall in EX then forward from WB	Stall twice in decode	N/A
Rd of <- instruction 2 before used	Forward from WB	Forward from WB	Forward from WB to EX for address or data	Stall once in decode	N/A
					Stall once for every jump

Euclid's algorithm in MISC assembly NO HAZARDS:

```
relPrime:   sp +_ -8, sp
            x0 + x0, x0
            x0 + x0, x0
            ra -> sp+0
            s0 -> sp+2
            s1 -> sp+4
            x0 +_ 2, s0
            x0 +_ 1, s1
            a0 -> sp+6
loop1:      s0 +_ 0, a1
            a0 <- sp+6
            ra \/ gcd
            x0 + x0, x0
            a0 Y= s1, done
            s0 +_ 1, s0
            x0 \/ loop1
            x0 + x0, x0
done:       s0 +_ 0, a0
            ra <- sp+0
            s0 <- sp+2
            s1 <- sp+4
            ra /\ 4
            x0 + x0, x0
gcd:        sp +_ -2, sp
            x0 + x0, x0
            x0 + x0, x0
            ra -> sp+0
            a0 Y= x0, returnb
loop2:      a1 Y= x0, returna
            a1 Y< a0, agreater
            a1 - a0, a1
            x0 + x0, x0
            x0 \/ loop2
            x0 + x0, x0
agreater:   a0 - a1, a0
            x0 \/ loop2
            x0 + x0, x0
returnb:    a1 +_ 0, a0
returna:    ra <- sp+0
            x0 + x0, x0
            x0 + x0, x0
            ra /\ 1
            x0 + x0, x0
```

address	assembly	Machine code	comments
0x00	sp +_ -8, sp	0011 1000 1001 0001	//begin relprime
0x01	x0 + x0, x0	0000 0000 0000 0000	//Delay
0x02	x0 + x0, x0	0000 0000 0000 0000	//Delay
0x03	ra -> sp+0	0000 0000 1000 1011	
0x04	s0 -> sp+2	0000 0100 1011 0011	
0x05	s1 -> sp+4	0000 1000 1011 1011	
0x06	x0 +_ 2, s0	0000 0100 0011 0001	
0x07	x0 +_ 1, s1	0000 0010 0011 1001	
0x08	a0 -> sp+6	0000 1100 1010 0011	
0x09	s0 +_ 0, a1	0000 0001 0111 0001	//begin loop1
0x0A	a0 <- sp+6	0000 1100 1001 0010	
0x0B	ra \/ gcd	0000 0010 0100 1110	//imm = 9
0x0C	x0 + x0, x0	0000 0000 0000 0000	//Delay
0x0D	a0 Y= s1, done	0000 1111 0001 1100	//imm = 3
0x0E	s0 +_ 1, s0	0000 0011 1011 0001	
0x0F	x0 \/ loop	1111 1110 1100 0110	//imm = -5
0x10	x0 + x0, x0	0000 0000 0000 0000	//Delay
0x11	s0 +_ 0, a0	0000 0001 1010 0001	//begin done
0x12	ra <- sp+0	0000 0000 1000 1010	
0x13	s0 <- sp+2	0000 0100 1011 0010	
0x14	s1 <- sp+4	0000 1000 1011 1010	
0x15	ra /\ 4	0000 0001 0000 1111	
0x16	x0 + x0, x0	0000 0000 0000 0000	//Delay
0x17	sp +_ -2, sp	0011 1100 1001 0001	//begin gcd
0x18	x0 + x0, x0	0000 0000 0000 0000	//Delay
0x19	x0 + x0, x0	0000 0000 0000 0000	//Delay
0x1A	ra -> sp+0	0000 0000 1000 1011	
0x1B	a0 Y= x0, returnb	0000 0001 0011 1100	//imm = 7
0x1C	x0 + x0, x0	0000 0000 0000 0000	//Delay
0x1D	a1 Y= x0, returna	0000 0001 0111 1100	//begin loop2;imm=7
0x1E	x0 + x0, x0	0000 0000 0000 0000	//Delay
0x1F	a1 Y< a0, agreater	0000 1001 0101 1101	//imm = 3
0x20	x0 + x0, x0	0000 0000 0000 0000	//Delay
0x21	a1 - a0, a1	0001 1001 0110 1000	
0x22	x0 + x0, x0	0000 0000 0000 0000	//Delay
0x23	x0 \/ loop	1111 1111 0100 0110	//imm = -3
0x24	x0 + x0, x0	0000 0000 0000 0000	//Delay
0x25	a0 - a1, a0	0001 1011 0010 1000	//begin agreater
0x26	x0 \/ loop	1111 1110 1100 0110	//imm = -5
0x27	x0 + x0, x0	0000 0000 0000 0000	//Delay
0x28	a1 +_ 0, a0	0000 0001 0110 0001	//begin returnb
0x29	ra <- sp+0	0000 0000 1000 1010	//begin returna
0x2A	x0 + x0, x0	0000 0000 0000 0000	//Delay
0x2B	x0 + x0, x0	0000 0000 0000 0000	//Delay
0x2C	ra /\ 1	0000 0000 0100 1111	
0x2D	x0 + x0, x0	0000 0000 0000 0000	//Delay

Datapath Layout



Component List and Descriptions		All test benches can be found at (original filename)_tb.v		
Component	**Inputs**	**Outputs**	**Behavior**	**RTL Symbols**
Register (Register.v)	RegWrite[0:0], CLK [0:0], Reset [0:0], RegInput[15:0]	RegOutput 1[15:0]	On the rising edge of the clock (*CLK*) the register reads the *RegWrite* 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
Register Block IF/ID (IF_ID.v)	IPC[15:0], IIR[15:0], RegWrite[0:0], CLK [0:0], Reset [0:0]	OPC[15:0], OIR[15:0]	On the rising clock edge (CLK) if RegWrite is on, then it takes in all I inputs and stores them, outputting them for the next cycle on the matching O outputs. When Reset is on, then all registers reset to zero, and blank out.	IF ID RB

Register Block ID/EX (ID_EX.v)	IRegWrite[0:0], IALUSrc[0:0], IALUOP[2:0], IBranch[0:0], IMemWrite[0:0], IMemRead[0:0], IRegStore[0:0], I1stArg[15:0], I2ndArg[15:0], I3rdArg[15:0], IImm[15:0], IRs1[15:0], IRs2[15:0], IRd[15:0], RegWrite[0:0], CLK [0:0], Reset [0:0]	ORegWrite[0:0], OALUSrc[0:0], OALUOP[2:0], OBranch[0:0], OMemWrite[0:0], OMemRead[0:0], ORegStore[0:0], O1stArg[15:0], O2ndArg[15:0], O3rdArg[15:0], OImm[15:0], ORs1[15:0], ORs2[15:0], ORd[15:0]	On the rising clock edge (CLK) if RegWrite is on, then it takes in all I inputs and stores them, outputting them for the next cycle on the matching O outputs. When Reset is on, then all registers reset to zero, and blank out.	ID EX RB
Register Block EX/MEM (EX_MEM.v)	IRegWrite[0:0], IALUSrc[0:0], IALUOP[2:0], IMemWrite[0:0], IMemRead[0:0], IRegStore[0:0], IALUResult[15:0], I3rdArg[15:0], IRD[15:0], RegWrite[0:0], CLK [0:0], Reset [0:0]	ORegWrite[0:0], OALUSrc[0:0], OALUOP[2:0], OMemWrite[0:0], OMemRead[0:0], ORegStore[0:0], OALUResult[15:0], O3rdArg[15:0], ORD[15:0]	On the rising clock edge (CLK) if RegWrite is on, then it takes in all I inputs and stores them, outputting them for the next cycle on the matching O outputs. When Reset is on, then all registers reset to zero, and blank out.	EX MEM RB
Register Block MEM/WB	IRegWrite[0:0], IRegStore[0:0],	ORegWrite[0:0], ORegStore[0:0],	On the rising clock edge (CLK) if RegWrite is on, then it takes in	MEM WB RB

(MEM_WB.v)	IALUResult[15:0], IStoreMem[15:0], IRD[15:0], RegWrite[0:0], CLK [0:0], Reset [0:0]	OALUResult[15:0], OStoreMem[15:0], ORD[15:0]	all I inputs and stores them, outputting them for the next cycle on the matching O outputs. When Reset is on, then all registers reset to zero, and blank out.	
Register File (Register_File.v)	Address1[2:0]; Address2[2:0]; Address3[2:0]; LoadData[15:0]; RegWrite[0:0], CLK [0:0], Reset [0:0]	Out1[15:0], Out2[15:0], Out3[15:0],	On the rising edge of the clock (CLK), if RegWrite is on the Register file will take the LoadData input and load it into the register numbered by LoadAddr. On the falling edge of the clock (CLK), the register file reads the registers numbered by the Address 1, 2, and 3 inputs and then outputs them on the Out 1, 2, and 3 outputs. If reset is on then all registers reset to zero.	ID EX RB
ALU (ALU.V)	FirstInput[15:0], SecondInput[15:0], ALUOp[2:0], CLK[0:0]	ALUres[15:0]	Performs some kind of operation on the 2 input data arguments based on the [2:0] ALUOp code, and outputs the result.	Main
ALU_Adder (no file)	FirstInput[15:0], SecondInput[15:0], CLK[0:0]	ALUres[15:0]	Performs addition on the 2 input data arguments, and outputs the result.	ALU +
Data Memory (Memory_Data.v, which is just a wrapper for raw_memory.v)	InputAddress [15:0], Data [15:0] MemWrite [0:0], MemRead [0:0], CLK [0:0]	OutputData [15:0]	Holds data in addressable segments as bytes. Has a read and a write control to determine operation. Only outputs data when it is reading	Data, Instruction

Instruction Memory (Memory_Text, which is just a wrapper for raw_memory.v)	InputAddress [15:0]	OutputData[15:0]	Gets the instruction from memory using PC as the input. Outputs the instruction to IR.	Data, Instruction
Mux for k, n-bit inputs (mux16b2.v, and mux16b4.v)	Input0[n-1:0],…, InputK[n-1:0]; Select[ceil(klog2):0]	Output[n-1:0]	Uses a selector input to choose between k different inputs to put on the output wire.	(not labled, but all trapazoids)
Forwarding Unit (Forward.v)	Rs1[2:0] Rs2[2:0] Rs1EX[2:0] Rs2EX[2:0] RdEX[2:0] RdMEM[2:0] RdWB[2:0] RegWriteMEM[0:0] RegWriteWB[0:0]	fwd1EX[1:0] fwd2EX[1:0] fwd3EX[0:0] Bfwd1[1:0] Bfwd2[1:0] fwdMEM[0:0]	Takes input from Rs1 and Rs2 from the ID/EX registers and the Rd from the EX/MEM and MEM/WB register. If any Rd outputs are the same is the Rs1 or Rs2 registers, the value is forwarded from the ALUResult to the ALU input by changing the mux select input.	Forwarding Unit
Control Unit (Control.v)	IR[15:0]	ALUop[2:0] ALUsrc[0:0] RegStore[0:0] RegWrite[0:0] MemRead[0:0] MemWrite[0:0] Branch[0:0]	Takes in the IR and decodes the instruction by setting all the control inputs accordingly to what instruction is being executed.	Control
Immediate Genie (Imm_Gen.v)	IR[15:0]	Immediate[15:0]	Reads the instruction and outputs an immediate based on the type of instruction. Will take the bits dependent on the type of instruction for the immediate and sign extend them for immediate type instructions. Will take the bits dependent on the type and shift them left 1 and sign extend for branch and jump instructions.	Immediate Genie

Comparator (Comparator.v)	IR[2:0], Out1[15:0], Out2[15:0],	BranchOutput[0:0]	Does logic depending on the opcode for branch equal, branch less than, or just a jump and sends a signal to the Jump? And gate to determine if we should jump or branch to update PC.	=

Integration Plan

Our plan for integrating the CPU is to assemble each stage individually, then hooking all five stages together to complete the CPU. Each stage will start with it's register block, if applicable.

Integration Item List

Component	Inputs	Outputs	Behavior
Fetch Stage (Fetch_Stage.v)	pc_in [15:0], reset, clk	new_pc[15:0], old_pcp2[15:0], old_pc[15:0], ir[15:0]	The fetch stage will be the stage associated with retrieving data from memory, and holding onto and increasing the current PC.
Decode Stage (Decode_Stage.v)	IPCP2[15:0], pc_in[15:0], ir_in[15:0], loadAddr[2:0], loadData[15:0], comparatorMux1Control[1:0], comparatorMux2Control[1:0], comparatorMuxForwardMEM[15:0], comparatorMuxForwardWB[15:0], rf_write, reset, clk,	RegWrite, ALUSrc, ALUOp[2:0], MemWrite[0:0], MemRead[0:0], RegStore[1:0], OPCP2[15:0], Arg1[15:0], Arg2[15:0], Arg3[15:0], Imm[15:0], Rs1[2:0],	The decode stage is the stage where the command is processed. If the command is a branch or a jump, then all logic that deals with branching and jumping is figured out here. Additionally, any data needed from registers is retrieved for later. This stage

	RB_write	Rs2[2:0], Rd[2:0], new_pc[15:0], jump[0:0]	has some overlap with the writeback stage since the writeback stage needs the register file to write its data.
Execute Stage (Execute_Stage.v)	IRegWrite, IALUSrc, IALUOp[2:0], IMemWrite, IMemRead, IRegStore[1:0], IPCP2[15:0], I1stArg[15:0], I2ndArg[15:0], I3rdArg[15:0], Imm[15:0], IRs1[2:0], IRs2[2:0], IRd[2:0], ALUResultMEM[15:0], loadDataWB[15:0], muxFwd1select[1:0], muxFwd2select[1:0], muxFwd3select[0:0], reset, clk, RB_write	ORegWrite, ORegStore[1:0], OMemWrite, OMemRead, OPCP2[15:0], OALUResult[15:0], O3rdArg[15:0], ORs1[2:0], ORs2[2:0], ORd[2:0]	The execute stage is where the main ALU lives, and is where the bulk of the processing happens as a result.
Memory Stage (Memory_Stage.v)	reset, clk, IRegWrite[1:0], IRegStore, MemWrite, MemRead, IPCP2[15:0], IALUResult[15:0],	ORegWrite, ORegStore[1:0], OPCP2[15:0], OALUResult[15:0], OStoreMem[15:0], rdWB[2:0], out[15:0]	The memory stage is where the -> and <- commands do their work. This stage contains the data memory segment is and can be accessed.

	thirdArg[15:0], rdMem[2:0], loadData[15:0], DataInSelect, in[15:0]		
Write Stage (Write_Stage.v)	reset, clk, RegWrite, RegStore[1:0], IPCP2[15:0], ALUResult[15:0], StoreMem[15:0], rdWB[2:0]	loadData[15:0], loadAddr[2:0], regWriteOut[0:0]	The writeback stage finaly places any results back into the register block if applicable.
CPU (CPU.v)	clk[0:0], reset[0:0], in[15:0],	out[15:0]	This is the complete processor, with all stages connected. This should be able to run any MISC-V program the user wishes it to.

All tests for the integration items can be found at (original filename)_tb.v

Naming Conventions

Our naming conventions for components are either a descriptive acronym done in all caps, or a descriptive word/group of words in all caps that represents the component.

Our naming conventions for test benches is the name of the file it tests, then _tb appended at the end.

Our naming conventions for inputs/outputs is Upper camel case, unless there is a input and output with the same name, in which case the input starts with an extra I, and the output starts with a capital O.

RTL	R-type	I-type	M-type	Y-Type	J-type
Fetch	IR <= InstructionMem[PC]				
Decode	PC <= PC + 2 1stArg <= Reg[IR[8:6]] 2ndArg <= Reg[IR[11:9]] 3rdArg <= Reg[IR[5:3]] Imm <= ImmGen[Reg[IR]] Y=: If(1stArg == 2ndArg): Y<: If(1stArg < 2ndArg): \/: PC <= PC + SE(2*immediate) /\: PC <= 3rdArg				
Execute	ALUResult <= 1stArg op 2ndArg	ALUResult <= 1stArg op imm	ALUResult <= 1stArg + SE[15:9]		
Mem			->: DataMem[ALUResult] <= 3rdArg <-: StoreMem <= DataMem[ALUResult]		
Write	Reg[IR[5:3]] <= ALUResult	Reg[IR[5:3]] <= ALUResult	<-: Reg[IR[5:3]] <= StoreMem		\/: Reg[IR[5:3]] <= PC + 2

RTL Symbol descriptions:

PC – Program counter

IR – Instruction Register, holds the current instruction bits

1stArg – First register argument (rs1)

2ndArg - Second register argument (rs2)

3rdArg - Third register argument, load register and store data (rd)

ALUResult - Register holding result of our ALU

Zero – Register holding a 1 or 0 if ALUResult is 0

StoreMem - Data to store in rd from the memory

Op – opcode

Zero – is zero register to hold zero output of alu

Reg – Our main addressable register file

MainALU – alu that performs all functions

SimpleAdderALU – alu that is just an adder (for pc relative operations)

DataMem – section of memory that is for addressable and changeable memory

InstructionMem – section of memory that is non addressable and holds instructions

To double check this RTL, we have had each team member go over and ask questions to ensure that no major errors will impede our work.

Component Testing Plan

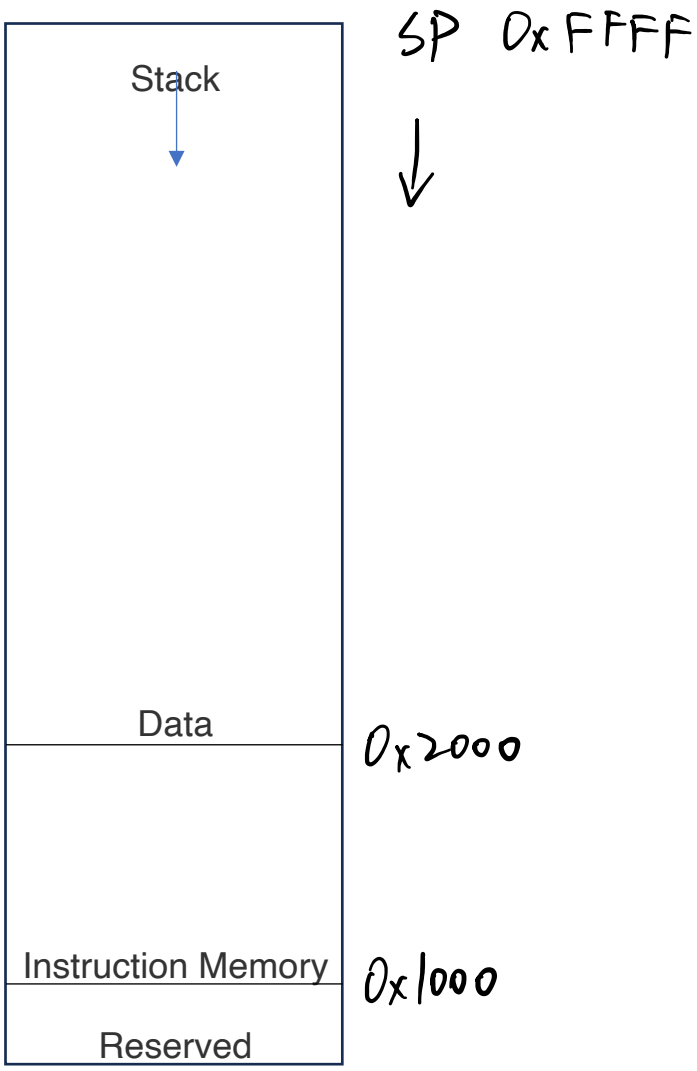
1. **Register Block (PC, IF/ID, ID/EX, EX/MEM, MEM/WB)**: For each register block, a test should be written to verify that it correctly holds and transfers data. Verify that data is stored and outputted correctly when RegWrite is on and the clock is on the rising edge. Also, check that the register resets when the Reset signal is high during a rising clock edge. Check that the register block holds and releases data correctly across multiple clock cycles.
2. **Register File**: Create tests to verify that it stores and retrieves data correctly. Test that on a rising clock edge, data is loaded into the appropriate register when RegWrite is on. On a falling clock edge, verify that data is correctly read from the addresses provided. Also, test the reset functionality.
3. **Register**: Create tests to verify that it stores and retrieves data correctly. Test that on a falling clock edge, and if RegWrite is on, that the register updates with a variety of values. Also check that it turns to zero when the reset bit is on.
4. **Arithmetic Logic Unit (ALU)**: Write tests to confirm the ALU performs the expected operations based on the ALUOp code. Test all possible ALUOp codes with different input data.
5. **Data Memory and Instruction Memory**: Verify that data is stored and retrieved correctly based on the MemWrite and MemRead control signals. Test that the memory components only output data when reading.
6. **Multiplexer (Mux for k, n-bit inputs)**: Create tests that confirm the correct input is outputted based on the Select signal. Test with different numbers of inputs and different Select codes.
7. **Forwarding Unit**: Write tests to ensure that the correct value is forwarded when Rd from EX/MEM or MEM/WB matches Rs1 or Rs2 from ID/EX. Test with different values and different matches.
8. **Immediate Genie**: Verify that the correct immediate value is output based on the type of instruction from IR. Test with different instruction types.
9. **Control:** Create tests for each instruction opcode and func, and make sure that all control signals are correctly output for the instruction.

10. **Comparator:** Write tests to test each type of branch and jump and confirm that the branches Y= and Y< work properly. Test to see that during jumps it always outputs a 1.

Integration Testing Plan

1. **Fetch Stage**: Ensure that the stage initializes correctly, and double check it works as we build the CPU up.
2. **Decode Stage**: Ensure that all 5 types are decoded and dealt with appropriately, as well as checking any possible edge cases that may arise when testing and programming overall.
3. **Execute Stage**: Ensure that a variety of inputs utilize the ALU appropriately, depending on what types utilize the stage.
4. **Memory Stage**: Ensure that both -> and <- commands move the data as appropriate.
5. **Writeback Stage**: Ensure that the mux selects correctly, since this stage cannot access the registers alone.
6. **Final Stage (CPU)**: Run tests using a variety of instruction sequences to verify the complete system works as expected. Test edge cases and error conditions to ensure system stability.

Memory Map



(Can break data into dynamic, static, stack later if needed, but for now it's just a stack that grows from top to bottom)

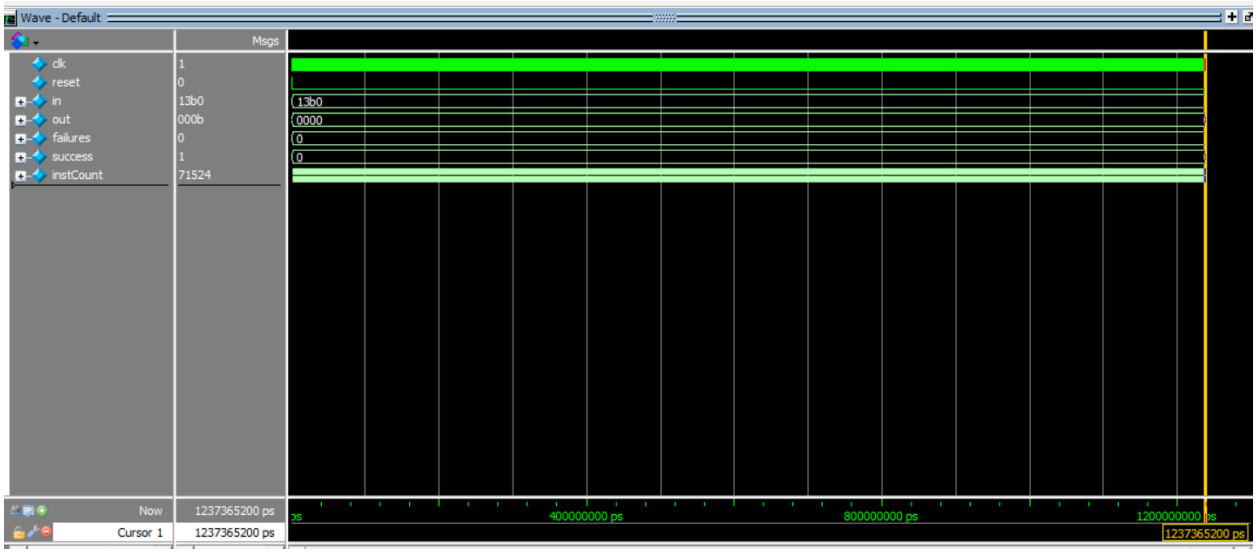
Control Signals

Control Signal	Description
ALUop	3-bit select input that goes into the main ALU to select the operation.
ALUsrc	Select input for the mux choosing the B input for the ALU. Selects between the immediate and the 2ndArg register.
RegStore	Select input for the mux choosing the load data for the register file. Chooses between ALUResult and StoreMem.
RegWrite	The input for the Register File enabling writing to the Register File.
MemRead	Input for data memory enabling reading from memory.
MemWrite	Input for data memory enabling writing to memory.
Branch	Input for AND gate to decide whether we will branch.
JumpOut	Signal for whether we are jumping out of a function.

Control State Diagram

	RegWrite	ALUSrc	ALUOp	RegStore	MemWrite	MemRead	Branch	JumpOut
R	**1**	**1**	**Op**	**01**	**0**	**0**	**0**	**0**
I	**1**	**0**	**Op**	**01**	**0**	**0**	**0**	**0**
M: <-	**1**	**0**	**+**	**00**	**0**	**1**	**0**	**0**
M: ->	**0**	**0**	**+**	**00**	**1**	**0**	**0**	**0**
Y	**0**	**0**	**-**	**00**	**0**	**0**	**1**	**0**
J: V	**0**	**0**	**0**	**10**	**0**	**0**	**1**	**0**
J: Λ	**0**	**0**	**0**	**00**	**0**	**0**	**1**	**1**

Benchmark Data



total number of bytes required to store both Euclid's algorithm and relPrime as well as any memory variables or constants.:

Instruction Mem size: 88 bytes using branch delay slot, 68 bytes using stalling instead.

Data Mem size: 10 bytes for stack used by functions.

total number instructions executed when relPrime is called with 0x13B0 (the result should be 0x000B using the algorithm specified in the project specifications).: 71520

total number of cycles required to execute `relPrime under the same conditions as Step 2.: 71524

average cycles per instruction based on the data collected in Steps 2 and 3.: ~1 (using branch delay slot)

The total execution time for relPrime under the same conditions as Step 2.: 1.2373652 ms

Flow Summary you should record the number and percentage of logic elements (i.e. logic gates used on the FPGA), the "Total registers" count, and the "Total memory bits" percentage. An example report is shown here:

Total logic elements 6,297 / 10,320 (61 %)

Total memory bits 0 / 423,936 (0 %)

```
relPrime:   sp +_ -8, sp
        x0 + x0, x0
        x0 + x0, x0
        ra -> sp+0
        s0 -> sp+2
        s1 -> sp+4
        x0 +_ 2, s0
        x0 +_ 1, s1
        a0 -> sp+6
loop1:  s0 +_ 0, a1
a0 <- sp+6
        ra \/ gcd
        x0 + x0, x0
        a0 Y= s1, done
        s0 +_ 1, s0
        x0 \/ loop1
        x0 + x0, x0
done:       s0 +_ 0, a0
        ra <- sp+0
        s0 <- sp+2
        s1 <- sp+4
        ra /\ 4
        x0 + x0, x0
gcd:        sp +_ -2, sp
        x0 + x0, x0
x0 + x0, x0
        ra -> sp+0
        a0 Y= x0, returnb
loop2:  a1 Y= x0, returna
        a1 Y< a0, agreater
        a1 - a0, a1
        x0 + x0, x0
        x0 \/ loop2
        x0 + x0, x0
agreater:   a0 - a1, a0
        x0 \/ loop2
        x0 + x0, x0
returnb:    a1 +_ 0, a0
returna:    ra <- sp+0
        x0 + x0, x0
        x0 + x0, x0
        ra /\ 1
x0 + x0, x0
```

Extra features

One of the main extra features we included with the MISC-V processor is a MISC-V assembler, which is in the main directory of the MISC-V project. Its filename is “assembler.py”. It can be run with a python environment of the user’s choice. It automatically reads the contents of the file program.txt and puts the compiled program into the memory.txt file in the implementation folder in the same directory. This program does not have any error catching, so if the assembly code does not have correct syntax, the program will output the wrong items. The example relprime code input is to the left of this text, and the memory file output can be found to the right.

```
3091
0000
0000
008B
04B3
08BB
0431
0239
0CA3
01A9
0CA2
030E
0000
0F24
03B1
FE86
0000
01A1
008A
04B2
08BA
010F
0000
3C91
0000
0000
008B
1114
1154
096D
1968
0000
FF06
0000
1B20
FE46
0000
0161
008A
0000
0000
004F
0000
```

Conclusion

Our design focused on optimizing the execution of Euclid's Algorithm, balancing speed and ease of programming. Our processor uses a symbol-based assembly language, enhancing accessibility. Our testing strategy encompassed both unit and integration testing, ensuring the reliability and efficiency of our design. The processor executed the algorithm with approximately 71520 instructions and 71524 cycles, indicating an average cycle per instruction close to 1.

Overall, we are very proud of the results. Beside all the technical knowledge, we learned valuable lessons on project management and communication. Being swift and decisive with our choices really helped us minimize the unnecessary work. We might expand on the processor by adding our incomplete branch predictor, as originally planned. We scraped it because it will have minimal impact on the relPrime algorithm.

Reference Sheet

inst	fmt	func	opcode	description
+	R	0000	000	R[rd] = R[rs1] + R[rs2]
-	R	0001	000	R[rd] = R[rs1] – R[rs2]
|	R	0010	000	R[rd] = R[rs1] | R[rs2]
&	R	0011	000	R[rd] = R[rs1] & R[rs2]
+_	I	00	001	R[rd] = R[rs1] + SE(imm)
<<_	I	01	001	R[rd] = R[rs1] << imm
>>_	I	10	001	R[rd] = R[rs1] >> imm
X|_	I	11	001	R[rd] = R[rs1] ^ SE(imm)
<-	M		010	R[rd] = M[R[rs1] + SE(imm)]
->	M		011	M[R[rs1] + SE(imm)] = R[rd]
Y=	Y		100	If(rs1==rs2) PC += SE(imm) << 1
Y<	Y		101	If(rs1<rs2) PC += SE(imm) << 1
\/	J		110	R[rd] = PC+2 PC += SE(imm) << 1
/\	J		111	PC = R[rd]

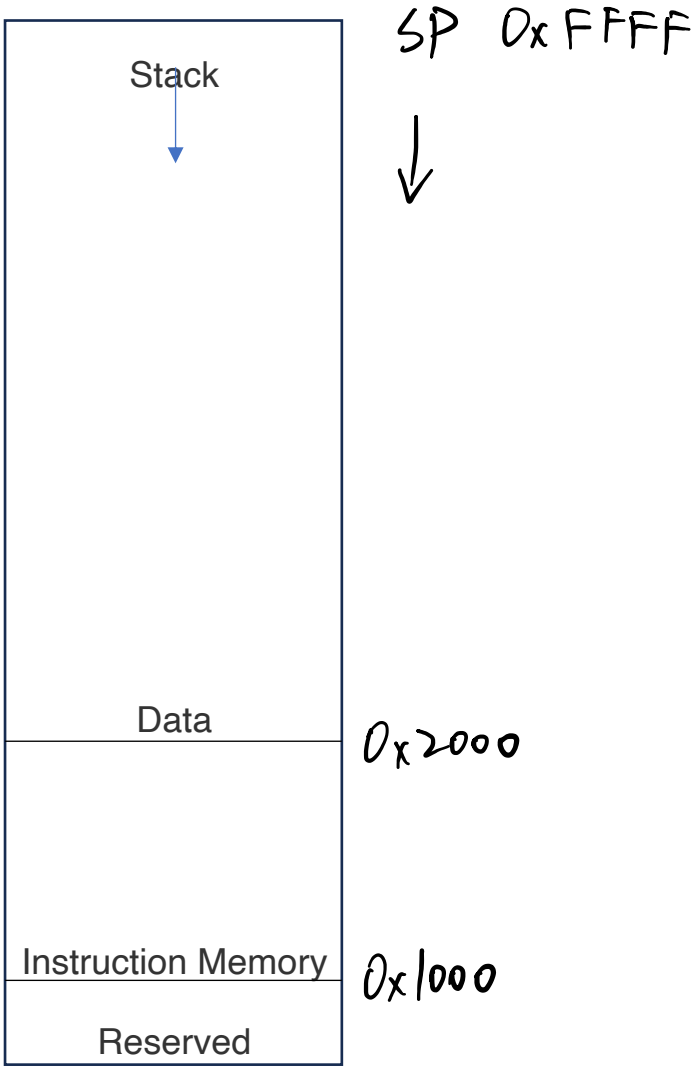
Register Names

Register	Name	Description	Saver
x0	zero	This register is always zero	-
x1	ra	This is the return address	caller
x2	sp	This is the stack pointer	-
x3	at	This is the assembler temporary	-
x4	a0	This is a temporary register that is used for function inputs and function return values	caller
x5	a1		
x6	s0	These are usable saved registers	callee
x7	s1		

Writing Instructions:

Type	Layout
R	rs1 (op) rs2, rd
I	rs1 (op) imm, rd
M	rd (op) rs1+imm
Y	rs1 (op) rs2, imm
J	rd (op) imm

Memory Map



(Can break data into dynamic, static, stack later if needed, but for now it's just a stack that grows from top to bottom)