OCTOBER 18, 2023

CSSE232 PROCESSOR PROJECT

TEAM YELLOW 2324A (DINO NUGGIES)

SPENCER HALSEY
BLAISE SWARTWOOD
REILLY MOONEY
JACOB SCHEIBE

Table of Contents

Design Philosophy

Our design strives to strike a balance between efficiency and ease of use for the programmer. We want our processor to be able to run quickly, while still providing the tools necessary for the programmer to have an easy time implementing code. To this end, we decided to use a Load and Store type format to allow the programmer to easily understand our instruction set, while still allowing for fast instruction calls. Our philosophy revolves around the idea that if there is a way to make something easier for the programmer to understand without giving up too much performance, we'll do it.

Measuring Performance

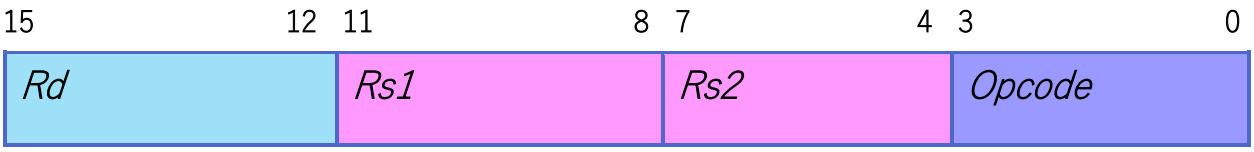
We will measure our performance based on versatility and ease of us (programmability) of our processor. The instructions are straight forward and simple to use while maintaining acceptable execution time. In terms of quantifiable numbers, we'll measure our execution time of a common function, Euclid's algorithm. If Euclid's can be easily implemented, and executed in a reasonable amount of time, we'll consider our processor a success.

Registers

Register	Name	Description	Saver
x0	zero	Zero constant	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3-x5	a0-a2	Fn args/return values	Caller
x6-x7	a3-a4	Fn args	Caller
x8-x10	s0-s2	Saved registers	Callee
x11-x13	t0-t2	Temporary registers	Caller
x14	br	Branch Compare register	Caller
x15	at	Assembler Temporary	Caller

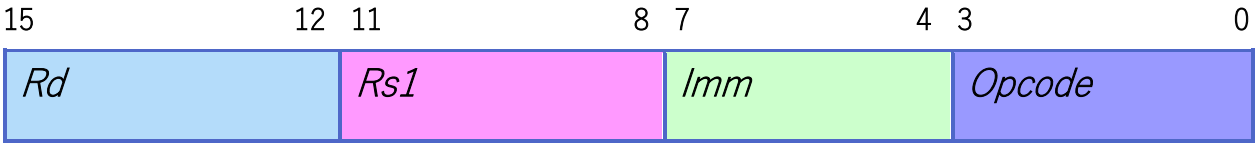
Instruction Types

Ralts-Type:



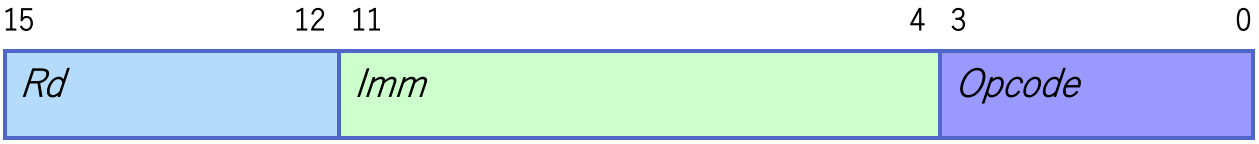
The Ralts -Type instruction format is mainly used for basic mathematical operations. The format contains a 4-bit operation code to distinguish what operation is being called, two 4 bit registers for operation arguments, and a 4 bit destination register for the result of the operation.

Ivysaur-Type:



The Ivysaur-Type instruction format is mainly used for immediate based operations, such as add immediate or store word. The format contains a 4-bit operation code, a 4-bit immediate value, a 4-bit argument register, and a 4-bit destination register.

Umbreon-Type:



The Umbreon-Type instruction format is mainly used for branching or immediate shifting operations. The format contains a 4-bit operation code, an 8-bit immediate value, and a 4 bit destination register.

*To convert instruction type into machine language, take the value, and convert into binary. EX: If you had add x1, x2, x3, you would convert it like this:

adds opcode = 0000
X1 = 0001
X2 = 0010
X3 = 0011

Then put it into the correct format based on its instruction type, in this case, R.

Rd Rs1 Rs2 Opcode

0001 0010 0011 0000

Instructions

Instruction	Instruction Name	Description	Meaning	Opcode	Type	Notes
Add	add	R[rd] = R[rs1] + R[rs2]	Rd = rs1 + rs2	0000	R	
Subtract	sub	R[rd] = R[rs1] - R[rs2]	Rd = rs1 – rs2	0001	R	
And	and	R[rd] = R[rs1] & R[rs2]	Rd = rs1 & rs2	0010	R	
Inclusive or	or	R[rd] = R[rs1] | R[rs2]	Rd = rs1 | rs2	0011	R	
Add Immediate	addi	R[rd] = R[rs1] + SE(imm)	Rd = rs1 + imm	0100	I	
Load Word	lw	R[rd] = M[R[rs1] + SE(imm)]	Rd = Memory[rs1 + imm]	0111	I	
Store Word	sw	M[R[rs1] + SE(imm)] = R[rd]	Memory[rs1 + imm] = rd	1000	I	Rd is not actually used as the rd, but as the value to set equal to
Branch if equal	beq	if(rs1 == BR) PC += SE(imm) << 1	If (rs1 == BR) go to PC + imm	1001	U	
Jump and link	jal	R[rd] = PC + 2 PC += SE(imm) << 1	Rd = PC + 2; Go to PC + imm	1100	U	PC relative
Jump and link register	jalr	R[rd] = PC + 2 PC = R[rs1]	Rd = PC + 2; Go to rs1 + imm	1101	I	Imm = 0, Register relative
Load upper immediate	lui	R[rd] = SE(imm) << 8	Rd[15:8] = imm	1110	U	

Load bottom immediate	lbi	R[rd] = R[rd] + imm	Rd[7:0] = imm	1111	U	

Shift immediate	si	If(imm[4] is 0)
		R[rd] = R[rd] << imm[3:0]
		
		Else (imm[4] is 1)
		R[rd] = R[rd] >> imm[3:0]
			If(imm[4] is 0)
			Rd = Rd << imm
			Else (imm[4] is 1)
			Rd = Rd >> imm
			
			If(imm[5] is 0)
			Shift logical
			Else(imm[5] is 1)
			Shift arithmetic
				0101	U	imm[4] determines if we shift right (1) or left (0) (based on sign)
						
						imm[5] determines if we shift logical (0) or shift arithmetic (1)

Load Input	Lin	R[rd] = INPUT	Load into destination Register	0110	U	
Load Output	Lout	OUTPUT = R[rd]	Get Output	1010	U	

Branching

How do you branch?

Our processor only provides a beq instruction, which means that other operations such as branch if less than (blt), branch if not equal to (bne), branch if greater than or equal to (bge) etc., have to be done using beq. Below is a template with the most efficient ways to perform each one. Additionally, you may notice that beq is a U-Type instruction, meaning it only uses one register. To account for

this, the br register holds the other value used for comparing. The branching template also demonstrates proper use of the dedicated register.

Branching Template

Beq

Assembly	High-Level

add br, t0, x0
beq t1, Branch
 	
Branch:
add t0, t0, t1
	If (a == b) {
	a = a + b
	}

Bne

Assembly	High-Level

add br, t0, x0
beq t1, Branch
add t0, t0, t1
 	
Branch:
	If (a != b) {
	a = a + b
	}

Blt

Assembly	High-Level

sub t2, t0, t1
si t2, 31 //imm[4] is 1 to shift right
//imm[3:0] is 1111 to shift 15 bits
add br, x0, t2
addi t2, x0, 0
beq t2, Branch
	If (a < b) {
	a = a + b
	}

add br, x0, t0
beq t1, Branch
add t0, t0, t1
 	
Branch:	

Bgt

Assembly	High-Level

sub t2, t0, t1
si t2, 31 //imm[4] is 1 to shift right
//imm[3:0] is 1111 to shift 15 bits
add br, x0, t2
addi t2, x0, 1
beq t2, Branch
add br, x0, t0
beq t1, Branch
add t0, t0, t1
 	
Branch:
	If (a > b) {
	a = a + b
	}

Bge

Assembly	High-Level

sub t2, t0, t1
si t2, 31 //imm[4] is 1 to shift right
//imm[3:0] is 1111 to shift 15 bits
add br, x0, t2
addi t2, x0, 1
	If (a >= b) {
	a = a + b
	}

beq t2, Branch add t0, t0, t1 Branch:	

Ble

Assembly	High-Level

sub t2, t0, t1
si t2, 31 //imm[4] is 1 to shift right
//imm[3:0] is 1111 to shift 15 bits
add br, x0, t2
addi t2, x0, x0
beq t2, Branch
add t0, t0, t1
 	
Branch:
	If (a <= b) {
	a = a + b
	}

Addressing Mode

All of the instructions in the processor are PC relative. IE: When we branch to a new location, it will use the value of PC plus some immediate to designate the new address. The two instruction types that use an immediate, I and U, are PC relative and the immediate bits are signed. The only time when we don't use PC to determine the address is when we use the jump and link register operation, which directly adds the immediate to a register value.

Calling Conventions

The stack pointer (sp) must always be manually expanded and restored after use. It is imperative that the programmer always returns the stack to its original value after use. Argument and temporary registers are assumed to not hold the same value after jumping to a different function.

The programmer must manually save those if they want to keep the values. Saved registers must always be saved and restored at the beginning and end of every function call by the programmer if they are used. It is assumed that the callee will keep these values the same. Arguments are passed into a function using a0-a4, and values are returned from a function through a0-a2. Return values for addresses are generally stored in ra. Example assembly code for basically calling conventions (see RelPrime for a more detailed example):

Address	Assembly	Comments
0x0000:	lbi t0, 8	Moving stack pointer
0x0002	sub sp, sp, t0	
0x0004	sw ra, 0(sp)	Storing ra
0x0006	sw s0, 4 (sp)	Storing saved
…	…	…
0x0026	lw ra, 0(sp)	Loading ra
0x0028	lw s0, 4 (sp)	Loading saved
0x002A	lbi t0, 8	Moving stack pointer
0x002C	add sp, sp, t0	

Naming Conventions

The naming conventions should try to express all the information needed in a clear and concise manner. It should follow the general rules:

- Convey the information on what file does concisely
- Follows the file naming conventions

The common files should be named as follows:

- Modules: mod_(name)
- Test Benches: (name)_tb
- Inputs: in_(name)
- Outputs: out_(name)
- Wires: [x:0] w _(name) (x is the number of bits - 1)
- Reset: RST

- Control Signals: Name[0:0]
- Registers: [x:0] reg_(name) (x is the number of bits - 1)
- Clock: CLK_# (Use _# is any more than one are needed)

```
/*
Design: Mux
File Name: Mux.v
Function: 4:2 or 2:1 Mux
Author: Spencer Halsey
*/
```

Assembly Language Examples

Address	Assembly	Machine Code	Comments
0x0000	add x11, x0, x12	1011 0000 1100 0000	// add an instruction
0x0002	sub x11, x12, x0	1011 1100 0000 0001	// subtract an instruction
0x0004	and x11, x0, x12	1011 0000 1100 0010	// and two registers
0x0008	or x11, x0, x12	1011 0000 1100 0011	// or two registers
0x000C	addi x11, x0, 2	1011 0000 0010 0100	// add a register and immediate
0x0010	lw x11, 4(x12)	1011 1100 0100 0111	// loads a value from memory
0x0012	sw x11, x11, 4(x12)	1011 1100 0100 1000	// stores a value to memory
0x0014	L: beq x11, L	1011 0000 0000 1001	// branches if reg. is == compare reg.
0x0020	L: jal x11, L	1011 0000 0000 1100	// jump and link
0x0022	jalr x11, 0(x1)	1011 0001 0000 1101	// jump and link reg.
0x0024	lui x11, 2	1011 0000 0010 1110	// load upper 8 bits
0x0028	lbi x11, 2	1011 0000 0010 1111	// load bottom 8 bits
0x002C	sai x11, 2	1011 0000 0010 0101	// arithmetic immediate shift
0x0030	sli x11, 2	1011 0000 0010 0110	// logical immediate shift

Addition using 16-bit Immediate

Instruction	Binary

lui x11, 2	1011 0000 0010 1110
lbi x11, 2	1011 0000 0010 1111

Looping

Instruction	Binary
0x0012: relLoop (Some instruction)	-
0x001E jal x0, relLoop	0000 1111 0100 1100
Instruction	**Binary**
beq a0, END	0011 0000 0110 1001
0x0020: END (Some instruction)	-

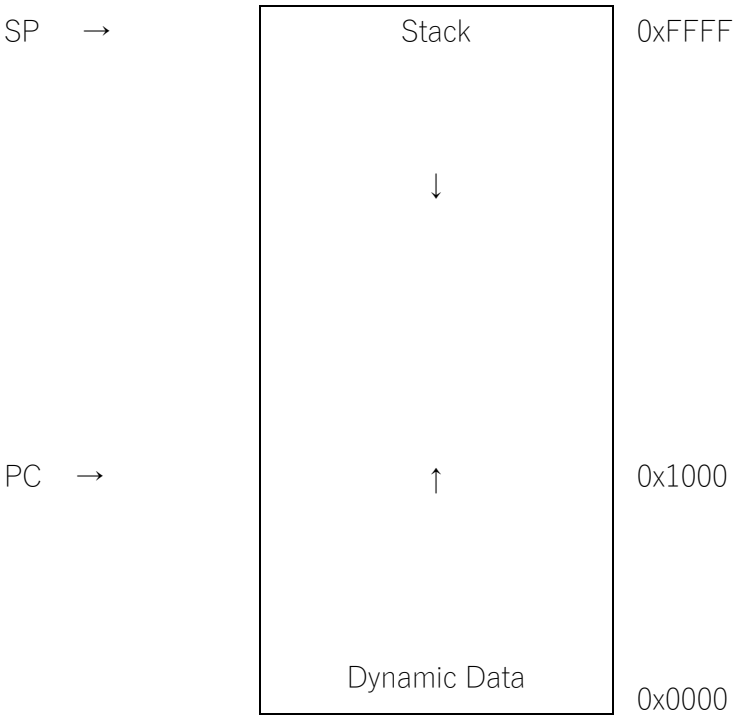
Branching

Instruction	Binary
add br, s2, x0	1110 1010 0000 0000
beq a0, END	0011 0000 0110 1001
0x0020: END (some instruction)	-

Recursion

Instruction	Binary
0x0012, CountDown:	-
add br, t0, x0	1110 1011 0000 0000
beq a0, END	0011 0001 1110 1001
sw a0, 0(sp)	0011 0000 0000 1000
addi a0, a0, -1	0011 0011 1111 0100
jal x0, CountDown	0000 1111 0100 1100
0x001E, END: (some instruction)	-

Memory Allocation



relPrime

Address	Assembly	Machine Code	Comments	RISCV Equ.
0x0000: relPrime	lbi t0, 16	1011 0001 0000 1111	Moving stack pointer	Addi sp, sp, -16
0x0002	sub sp, sp, t0	0010 0010 1011 0001		
0x0004	sw ra, 0(sp)	0001 0010 0000 1000	Storing ra	sw ra, 0(sp)
0x0006	sw s0, 4 (sp)	1000 0010 0100 1000	Storing saved	sw s0, 4 (sp)
0x0008	sw s1, 8 (sp)	1001 0010 1000 1000	Storing saved	sw s1, 4 (sp)
0x000A	sw s2, 12 (sp)	1010 0010 1100 1000	Storing saved	sw s2, 4 (sp)
0x000C	add s0, a0, x0	1000 0011 0000 0000	S0 = n	add s0, a0, x0

0x000E	addi s1, x0, 2	1001 0000 0010 0100	S1 = m	addi s1, x0, 2
0x0010	addi s2, x0, 1	1010 0000 0001 0100	S2 = 1	addi s2, x0, 1
0x0012: relLoop	add a0, s0, x0	0011 1000 0000 0000	Put n in a0	add a0, s0, x0
0x0014	add a1, s1, x0	0100 1001 0000 0000	Put m in a1	add a1, s1, x0
0x0016	jal ra, gcd	0001 0001 1010 1100 (imm = 26, offset = 52	Jump to gcd	jal ra, gcd
0x0018	add br, s2, x0	1110 1010 0000 0000	Put s2 into br	beq a0, s2, END
0x001A	beq a0, END	0011 0000 0110 1001 (imm = 6, offset = 12)	Branch if a0 == 1	
0x001C	addi s1, s1, 1	1001 1001 0001 0100	Increment m by 1	addi s1, s1, 1
0x001E	jal x0, relLoop	0000 1111 0100 1100 (imm = -12, offset = -24)	Loop again	jal x0, relLoop
0x0020: END	add a0, s1, x0	0011 1001 0000 0000	Put m to return	add a0, s1, x0
0x0022	lw ra, 0 (sp)	0001 0010 0000 0111	Loading ra	lw ra, 0 (sp)
0x0024	lw s0, 4 (sp)	1000 0010 0100 0111	Loading saved	lw s0, 4 (sp)
0x0026	lw s1, 8 (sp)	1001 0010 1000 0111	Loading saved	lw s1, 8 (sp)
0x0028	lw s2, 12 (sp)	1010 0010 1100 0111	Loading saved	lw s2, 12 (sp)

0x002A	lbi t0, 16	1011 0001 0000 1111	Moving stack pointer	Addi sp, sp, 16
0x002C	add sp, sp, t0	0010 0010 1011 0000		
0x002E	jalr x0, (0)ra	0000 0001 0000 1101	Returning to what called	jalr x0, ra

GCD

Address	Assembly	Machine Code	Comments	RISCV Equ.
0x0030: **gcd**	addi sp, sp, -4	0010 0010 1110 0100	Moving stack pointer	Addi sp, sp, -4
0x0032	sw ra, 0(sp)	0001 0010 0000 1000		sw ra, 0(sp)
0x0034	add br, x0, x0	1110 0000 0000 0000	Br = 0	beq a0, x0, END1
0x0036	beq a0, **END1**	0011 00001011 1001 (imm 9, offset 18)	Branch if a0 = 0 to end1	
0x0038: **gcdLoop**	add br, x0, x0	1110 0000 0000 0000	Br = 0	beq a1, x0, END2
0x003A	beq a1, **END2**	0100 00001010 1001 (imm 8, offset 16)	Move to end2	
0x003c	sub t2, a1, a0	1000 0100 0011 0001		
0x003e	si t2, 31	1000 0001 1111 0101	//imm[3:0] is 1111 to shift 15 bits //imm[4] is 1 to shift right	
0x0040	add br, x0, t2	1110 0000 1000 0000		

0x0042	addi t2, x0, 0	1000 0000 0000 0100		
0x0044	beq t2, SubA	1000 0000 0111 1001		
0x0046	add br, x0, a0	1110 0000 0011 0000		
0x0048	beq a1, SubA	0100 0000 0011 1001		
0x004a	sub a1, a1, a0	0100 0100 0011 0001	//b = b - a	sub a1, a1, a0
0x004c	jal x0, gcdLoop	0000 11111011 1100 (imm -5, offset -10)	//loop	jal x0, LOOP
0x004e: SubA	sub, a0, a0, a1	0011 0011 0100 0001	//a = a – b	sub, a0, a0, a1
0x0050	jal x0, gcdLoop	0000 11111001 1100 (imm -7, offset -14)	// loop	jal x0, LOOP
0x0052: END1	add a0, a1, x0	0011 0100 0000 0000	//put b in a0 for return	add a0, a1, x0
0x0054: END2	lw ra, 0(sp)	0001 0010 0000 0111	//a is in a0 already	lw ra, 0(sp)
0x0056	addi sp, sp, 4	0010 0010 0100 0100	Moving stack pointer	addi sp, sp, 4
0x005C	Jalr x0, ra	0000 0010 0000 1101	Jump back	Jalr x0, ra

Components

Component	Inputs	Outputs	Behavior	RTL Symbols
Register	CLK RST In_Data[15:0]	Out_Data[15:0]	Accepts input on clock uptick if the operation is supposed to write to it,	PC, IR, A, B, ALUout, MDR, SP

			stores the input, and uses it as its output Takes in the input on the rising CLK edge, outputs the stored value on the falling edge	
Register File	CLK RST In_RegWrite[0:0] In_ReadReg1[3:0] In_ReadReg2[3:0] In_WriteAddr[3:0] In_Data[15:0]	Out_ReadData1[15:0] Out_ReadData2[15:0]	Holds all of the registers used by the processor. Allows reading and writing of registers simultaneously. Stores/Reads a register on the rising CLK edge, reads the register on the falling CLK edge	
Instruction Register	In_Instruction[15:0]	Out_Opcode[3:0] Out_Reg1[3:0] Out_Reg2[3:0] Out_RegRd[3:0]	Decodes the instruction, and outputs the given information to other components	
Memory	CLK RST In_StackWrite[0:0] In_StackRead[0:0] In_StackOp[0:0]	Out_MemData[15:0]	Keeps track of return addresses and used for general storage by the programmer Stores the input on the rising CLK edge, reads the value on the falling edge	Mem
ALU	RST In_A[15:0] In_B[15:0] In_ALUCtrl[3:0]	Out_ALUResult[15:0] Out_Zero[0:0]	Will perform operations such as, addition, subtraction, or, and	+, -, /, |, & , >>, <<,
Control	CLK RST In_Instructions[4:0]	Out_IorD[0:0] Out_MemRead[0:0] Out_MemWrite[0:0] Out_IRWrite[0:0] Out_RegWrite[0:0]	Receives a 5-bit opcode, handles the input, and outputs the correct values as signals to complete the correct instruction	

		Out_ALUSrcA[0:0] Out_ALUSrcB[0:0] Out_ALUCtrl[3:0]	Takes in the opcode input on the rising CLK edge, outputs the control value on the falling edge	
ALUControl	In_Si[1:0] In_Inst[3:0]	Out_ALUCtrl[3:0]	Gets 2-bits from the Immediate Genie and a 4-bit opcode. This is only considered when doing a shift operation.	
Mux	In_Select[0:0] In_A[15:0] In_B[15:0]	Out_Result[15:0]	Takes 2 inputs and outputs the value corresponding to the control signal	
ImmGen	In_Inst[15:0]	Out_Imm[15:0] Out_Si[1:0]	Takes a 4 or 8-bit input and sign extends it, and outputs it	Imm, SE

Description of Control Signals

IorD (Instruction or Data) – determines whether the fetch operation in memory is fetching an instruction or accessing data. When set to 1, it is signaled that the memory operation is related to accessing data. When set to 0, the memory operation will fetch an instruction.

MemRead – indicates whether a memory read operation should be executed during the current clock cycle. When set to 1, it signals that the processor needs to read data from memory. When set to 0, this indicates that the current operation does not need to access data from memory.

MemWrite – indicates whether a memory write operation should be executed during the current clock cycle. When set to 1, it signals that the processor needs to write data to memory. When set to 0, it indicated that the current operation does not need to write data to memory.

IRWrite (Instruction Register Write) – indicates whether the current instruction should be written to a register in the register file. When set to 1, the signal indicates that the processor should write the fetched instruction to the IR register. This makes the instruction available for subsequent processing stages. If the signal is set to 0, that means writing the fetched instruction to the IR register does not occur in this clock cycle.

RegWrite – indicates whether the result of an operation writes back to a register within the register file. When set to 1, this signals that the result of the current operation should be written to a specified register in the register file. When 0, RegWrite indicates that the current operation does not write data to a register.

ALUSrcA – used to select a source for the first ALU input

ALUSrcB – used to select a source for the second ALU input

ALUCtrl – used to select which operation the ALU should perform

RTL Instructions

R Type

R Type	add	sub	and	or
Fetch	IR <= mem[PC] PC <= PC+2			
Decode	A <= Reg[IR[11:8]] B<= Reg[IR[7:4]]			
	ALUout <= A + B	ALUout <= A - B	ALUout <= A & B	ALUout <= A | B
	Reg[IR[15:12]] = ALUout			

I Type

I Type		addi	Lw	sw	jalr
Fetch		PC <= PC + 2 IR <= mem[PC]			
Decode		A <= Reg[IR[11:8]] B <=Reg[IR[15:12]]			
		ALUOut <= A + SE(IR[7:4]) (imm)			PC <= A
			MDR <= Memory[ALUOut]	Memory[ALUOut] <= B	
		Reg[IR[15:12] <= ALUOut	Reg[IR[15:12]] <= MDR		Reg[IR[15:12]] <= PC

U Type

U Type	Beq	Lui	Lbi	Jal	Si
Fetch	PC <= PC + 2 IR <= mem[PC]				
Decode	A <= Reg[IR[11:8]] B <= Reg[IR[15:12]]				
	A <= Reg[IR[15:12] B <= Br Imm = SE(IR[11:4]) << 1	Imm = SE(IR[11:4]) *Depending on type of shift, we might not need to SE	Imm = IR[11:4]	ALUOut <= PC + SE(2*immediate)	A <= Reg[IR[15:12] ALUOut <= A >>|<< imm[0:3]
	Target = PC + Imm	Result = Imm << 8	Result = IR[15:12] + Imm	PC <= ALUOut	Reg[IR[15:12] <= ALUOut
	If (A == B) PC = Target	Reg[IR[15:12]] = Result	Reg[IR[15:12]] = Result	Reg[IR[15:12]] <= PC	

RTL Double Checking For Errors

Each group member did one type of RTL. We all reviewed the RTL for the other instruction types each member did and confirmed that it made sense. If anything was incorrect, we discussed it and whether we needed to change/modify anything. Changes were made as necessary. Finally, in our group meeting, we verbally confirmed that everyone agreed each type's RTL was correct and we were satisfied to continue working on the next milestone.

Changes to Assembly & Machine Language

To free up an opcode for more instructions in the future, as well as some for inputs and outputs, we changed shifting to take up one instruction rather than two instructions. Shifting now consists of an opcode, register, and the immediate. One bit of the immediate determines whether to shift left or right, depending on the bit value. Another bit of the immediate determines whether to shift arithmetically or logically. To prepare for pipelining with this project, we also changed to having only one branch instruction, beq. The other branching operations – blt, bge, etc – are formed by doing

several operations instead of being a set dedicated instruction. This makes our project more unique compared RISCV and will allow for easier Datapath implementation and pipelining.

Building and testing Mux

Build

We are planning to build a 2:1 Multiplexer that selects between two different 16 bit input values and outputs the selected 16 bit value. It will be built using a switch case for the select. It then will output into a register when the selection has finished.

Testing Mux

We will test the two possible cases for the Multiplexer. Firstly, when the select bit is 0, it should select the first input. We will check to see if it matches the correct value. The program will then wait some arbitrary amount, then perform the second test. The second test is when the select bit is 1, it should select the second input. We will check to see if it matches the correct value. Then we will print all failures.

Building and testing Register

Build

We are planning to build a 16 bit register with a clock and a reset. It will write the 16 bit input value on the positive clock edge. Unless it is provided with a reset signal, in which it will set the result to zero.

Testing

Firstly, we will test inputting some value into the register, with reset being inactive. On the positive clock edge, it will set the value of the register. We will then check to make sure that the correct value has been set. It will then wait some arbitrary amount of time. Secondly, we will test the reset signal on the register. We will input some non-zero value into the register, and input a active reset signal (1). On the positive clock edge, it will set the value of the register to zero. We will then check to make sure that the correct value has been set.

Building and testing Register File

Build

We are planning to build a container file which contains all of our 16 dedicated registers (defined on page 3). The Register File will be able to read values from two registers on a negative clock edge, and write data/a register to a register on a positive clock edge. We will specify which registers need to be read/written to with signals. It will also have a signal to determine if it should write to a register.

Test

To test the ALU, we'll start with testing the reset signal. Firstly, we'll input an active (1) reset signal to the register file. We'll then check all of the registers to make sure that they have been zeroed on the positive clock edge. Then we'll read one register on the negative clock edge, and make sure that the value is correct. Then we will write to a register with the given data (with RegWrite being 1) and check to make sure the value is correct. Then we will try to write to another register (with RegWrite being 0), and ensure that no value is written. We will then write to a register (with RegWrite being 1) with the given register, ensure the value is correct, then reset to ensure that the value is being reset.

Building and testing ALU

Build ALU:

To build the ALU, we are going to take in two 16 bit values and the ALUCtrl signal. The ALUCtrl will determine what operation we are going to perform as noted in the table below. Then, a switch case statement will be made for each of them and then the resulting operation will be performed and stored in Out_ALUResult.

In_ALUCtrl	Operation
0000	add
0001	sub
0010	and
0011	or
0100	xor
0101	left shift logical

0110	right shift logical
0111	not
1000	multiply
1001	divide
1010	Increment
1011	decrement
1100	Left shift arithmetic
1101	Right shift arithmetic
1110	Get B value (immediate)

Testing ALU:

To ensure the ALU operations are performed correctly, we will test every single operation at least once with rather random values. We will make sure to use a mix of positive and negative values to ensure all operations are performed successfully. For shifting, we will have 4 different tests, one for each type and make sure our operations are performed well.

Test Instruction	Input	Expected Output
Add	In_A = 6 In_B = -8 In_ALUCtrl = 4'b0000	Out_ALUResult = -2 Out_Zero = 0
Sub	In_A = 12288 In_B = 8 In_ALUCtrl = 4'b0001	Out_ALUResult = 12280 Out_Zero = 0
And	In_A = -1 In_B = -8 In_ALUCtrl = 4'b0010	Out_ALUResult = -8 Out_Zero = 0
Or	In_A = 6 In_B = -8 In_ALUCtrl = 4'b0011	Out_ALUResult = -2 Out_Zero = 0
Xor	In_A = 8074 In_B = 12200 In_ALUCtrl = 4'b0100	Out_ALUResult = 6 Out_Zero = 0
Left Shift logical	In_A = 3	Out_ALUResult = 12

	In_B = 2 In_ALUCtrl = 4'b0101	Out_Zero = 0
Right shift logical	In_A = 48 In_B = 2 In_ALUCtrl = 4'b0110	Out_ALUResult = 12 Out_Zero = 0
Not operation	In_A = -8 In_B = 0 In_ALUCtrl = 4'b0111	Out_ALUResult = 7 Out_Zero = 0
Multiply	In_A = 10 In_B = -8 In_ALUCtrl = 4'b1000	Out_ALUResult = -80 Out_Zero = 0
Divide	In_A = 10 In_B = 2 In_ALUCtrl = 4'b1001	Out_ALUResult = 5 Out_Zero = 0
Increment	In_A = -8 In_B = 0 In_ALUCtrl = 4'b1010	Out_ALUResult = -7 Out_Zero = 0
Decrement	In_A = 2 In_B = 1 In_ALUCtrl = 4'b1011	Out_ALUResult = 1 Out_Zero = 0
Left Shift Arithmetic	In_A = 2 In_B = 2 In_ALUCtrl = 4'b1100	Out_ALUResult = 8 Out_Zero = 0
Right Shift Arithmetic	In_A = -32 In_B = 3 In_ALUCtrl = 4'b1101	Out_ALUResult = -4 Out_Zero = 0
Pass through	In_A = -32 In_B = 7 In_ALUCtrl = 4'b1110	Out_ALUResult = 7 Out_Zero = 0

Building and testing Immediate Generator

Build immediate generator:

We are planning to build an unclocked immediate generator that takes in a reg instruction and outputs immediate. It will be built using a switch case for the opcode.

Testing immediate generator:

To ensure thorough testing of the immediate generator, at least one R type will be tested to ensure an immediate genie of 0 is given. Then, for each instruction that uses an immediate for the type, we will test both a positive and negative immediate value and make sure the desired immediate value is returned. The input was a 16-bit instruction, and the output was a 16 bit immediate. Since no clock is used, there will simply be a short wait time (#10) between setting variables.

Instruction	Input (instr)	Output
Addi positive	16'b0000000100110100	3
Addi negative	16'b0010001111110100	-1
Lw positive	16'b0010000101000111	4
Lw negative	16'b0011001010100111	-6
Sw positive	16'b0001100100101000	2
Sw negative	16'b1001011011101000	-2
Beq positive	16'b1000000010001001	8
Beq negative	16'b1001111111001001	-4
Jal positive	16'b0010000001101100	6
Jal negative	16'b1011111101101100	-10
jalr	16'b0001001000000111	0
Lui positive	16'b1101000011011110	16'b00001101_00000000
Lui negative	16'b1110111010111110	16'b11101011_0000_0000
Lbi positive	16'b1100010010101111	16'b00000000_01001010
Lbi negative	16'b1101101011101111	16'b00000000_10101110
Si	16'b0011001101010101	5
Si	16'b0010000111110101	15

Building and testing Memory

Build memory unit:

We are planning to build a clocked memory unit that takes an address as an input, and outputs an instruction. It will be built using a single port RAM template.

Testing memory unit:

Since the memory unit will only take the 4-bit opcode as the address input, we need to test each unique opcode. When the simulation runs, the input address should change, and the output value should change on the next rising edge. In this case, the input will be the 4-bit opcode and the output will be the desired instruction.

RTL Changes Description

- We moved our RTL to follow a pipelining format, but ultimately decided to go with a multicycle implementation and changed our RTL parts back to multicycle basics
- We fixed several issues in our RTL where branching and jumping was not worked correctly.

Partial Implementation and Tests

We have partial implementations and tests for Immediate Genie, ALU, Mux, and Register done.

Assembler Information:

The assembly code must follow these certain rules for the machine code to be calculated correctly:

- All immediates are given in decimal representation except for the immediate given to si, which is given in an 8 bit binary format
- There are no extra lines between code instructions, as each line is taken as a 2 bit address when calculating tags

The assembler can now account for pseudoinstructions. To create a pseudoinstruction, in the pseudoinstruction.txt, add your pseudo instructions and parameters using val1, val2, ... Add a colon when the template for the instruction is complete. Then in the next lines, one instruction per line, put the instructions that make up the pseudoinsturctions. Add an empty line after the final instruction.

For example:

rob val1, val2, val3:

add val1, val2, val3

sub val2, val3, val2

ala val1, val2:

beq val1, val2

jal val1, val2

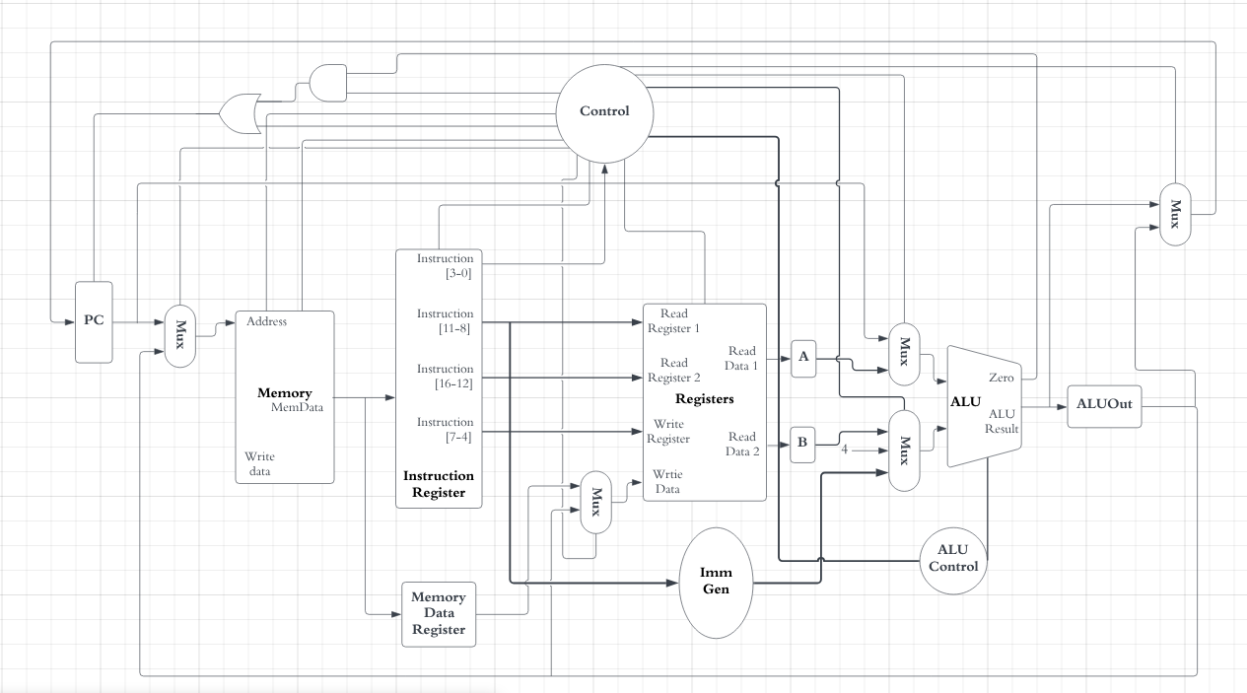
The assembler can handle having tags added in the same line as pseudoinstructions and regular instructions. To add a tag into the data, use the format:

NAME: instruction

So for example:

LOOP: add r2, r3, r2

Datapath



Control

ALU Control

The ALU Control will make sure that each instruction opcode is then taken in and processed to do the correct ALU operation depending on the instruction. The immediate genie will also pass in the 2 unique shift bits to the ALU Controller which will be used only for shifts to determine the specific type of shift mentioned earlier.

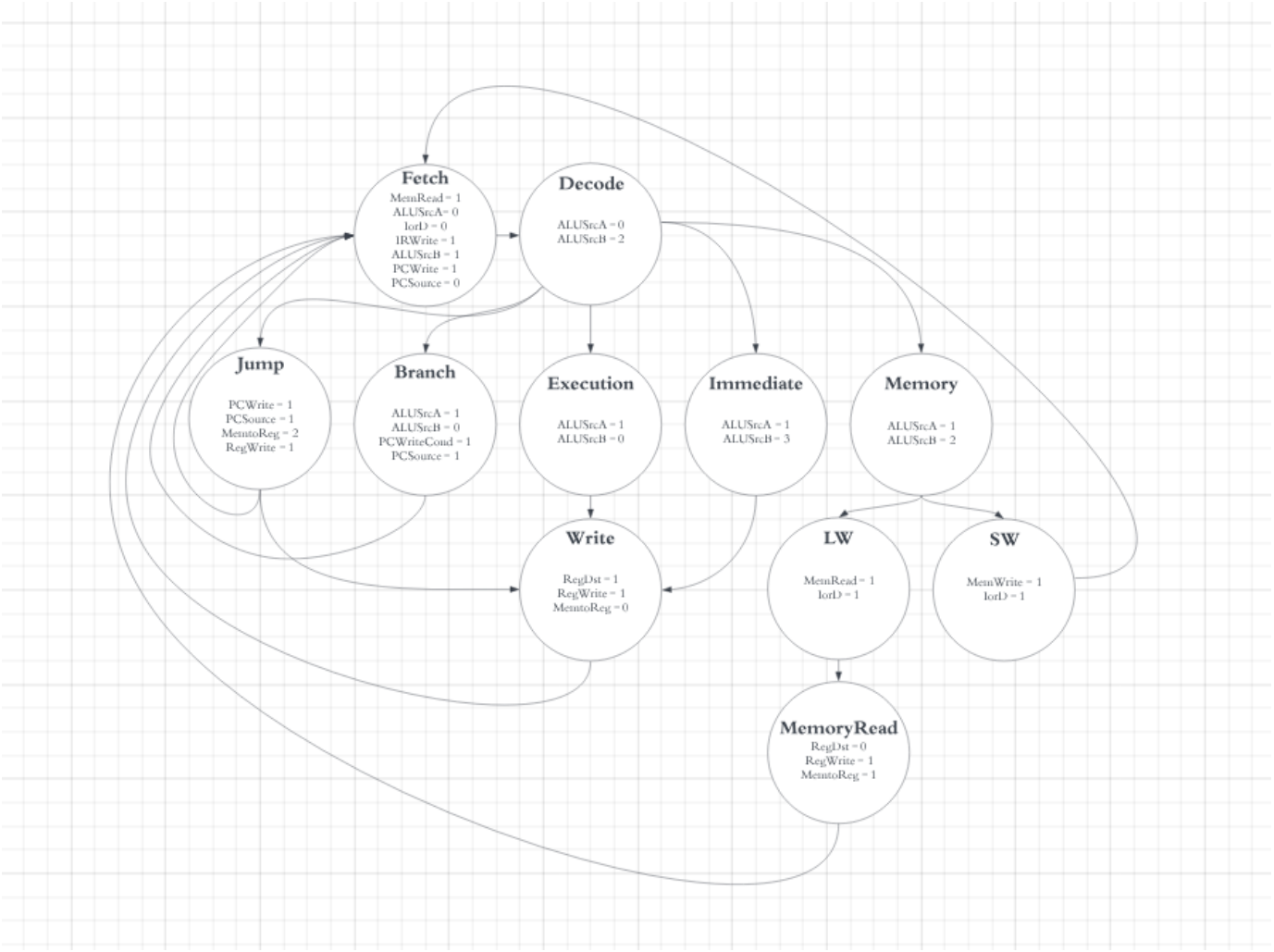
ALU Control Testing

The ALU Control testing will test each instructions and make sure the correct output is given. It will really care about the Si input bits for the shifting and make sure the processing happens correctly.

Instruction	Input	Output
Add	In_Inst = 4'b0000 In_Si = 2'b00	Out_ALUCtrl = 4'b0000
Sub	In_Inst = 4'b0001 In_Si = 2'b00	Out_ALUCtrl = 4'b0001
And	In_Inst = 4'b0010 In_Si = 2'b00	Out_ALUCtrl = 4'b0010
Or	In_Inst = 4'b0011 In_Si = 2'b00	Out_ALUCtrl = 4'b0011
Lw	In_Inst = 4'b0111 In_Si = 2'b00	Out_ALUCtrl = 4'b0000
Sw	In_Inst = 4'b1000 In_Si = 2'b00	Out_ALUCtrl = 4'b0000
Beq	In_Inst = 4'b1001 In_Si = 2'b00	Out_ALUCtrl = 4'b0001
Jal	In_Inst = 4'b1100 In_Si = 2'b00	Out_ALUCtrl = 4'b0000
Lui	In_Inst = 4'b1110 In_Si = 2'b00	Out_ALUCtrl = 4'b1110
Lbi	In_Inst = 4'b1111 In_Si = 2'b00	Out_ALUCtrl = 4'b0011
SLL	In_Inst = 4'b0101 In_Si = 2'b00	Out_ALUCtrl = 4'b0101
SRL	In_Inst = 4'b0101 In_Si = 2'b01	Out_ALUCtrl = 4'b0110

SLA	In_Inst = 4'b0101 In_Si = 2'b10	Out_ALUCtrl = 4'b1100
SRA	In_Inst = 4'b0101 In_Si = 2'b11	Out_ALUCtrl = 4'b1101

State Diagram



Integration Plan

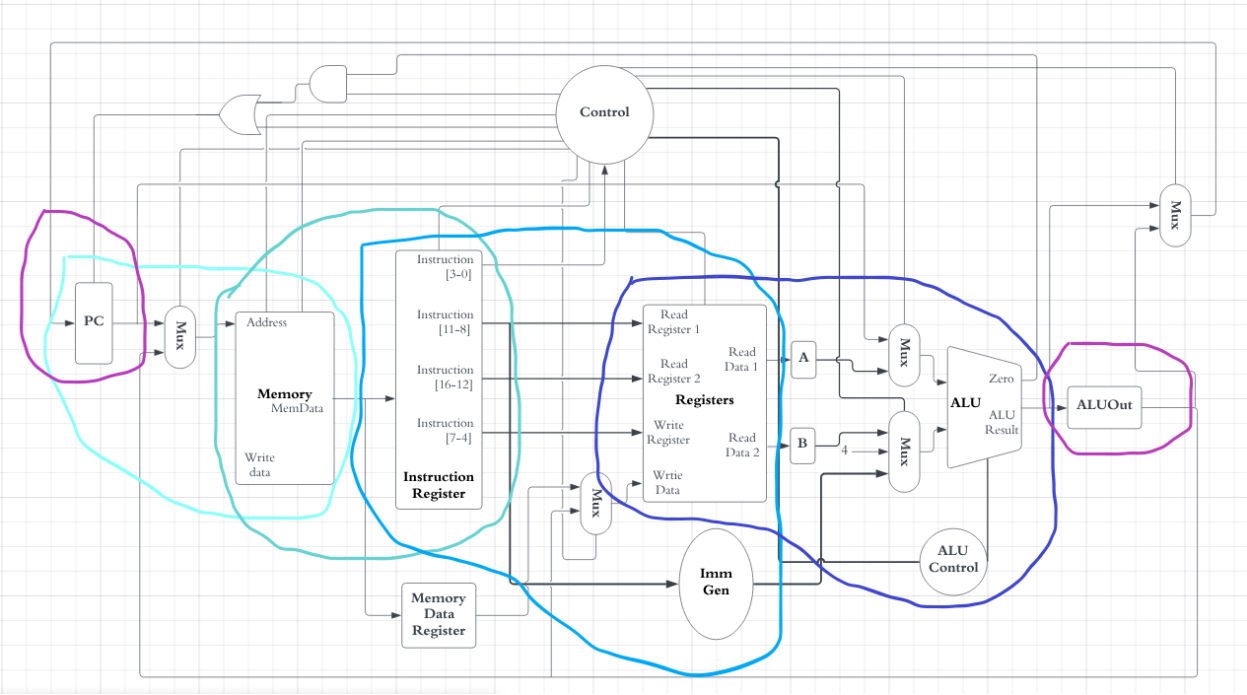
Unit Testing

Begin by writing unit tests for each individual component including, registers, register file, memory, immediate generator, ALU, and Control. Make sure each component is working as expected in isolation.

Component Integration

Gradually integrate components in the following order, ensuring proper communication between each component and proper functionality over multiple components. Once finished, we will create a top-level file which will essentially serve as the main file for our processor, that when executed will run rel-prime.

Integration	File Name
PC and Memory	Integration_Testing/PC_Memory_tb
Memory and Instruction Register	Integration_Testing/Memory_Instruction_tb
Instruction Register, Immediate Generator, and RegisterFile	Integration_Testing/IR_IG_RegisterFile
Register File and ALU	Integration_Testing/Register_ALU_tb
ALUOut and PC	Integration_Testing/ALUOut_PC_tb
ALU, ALUControl, and ImmediateGenie	Integration_Testing/ALU_ALUControl_ALUImmediate Genie_tb



ALU, ALUControl, ImmediateGenie Plan

We are going to connect these components together, with the two bit op from the immediate genie going into the ALU control along with a given 4 bit opcode. This will determine the ALUCtrl signal which will be passed into the ALU. Values will be given for the value of A to the ALU, and sometimes register values will be given to the value of B for the ALU, or values from the immediate genie will be passed in. The operation will be performed and the final result from the ALUOut will be compared to our expected result.

Tests:

The tests for these will focus on the I and U types because immediates from the immediate genie are actually used and need to make sure they are being passed around correctly. At least one test for each different instruction will be used to make sure the correct operation is being performed for each one.

Instruction	Input	Output
add r3, r5, r7	Instr 16'b0011010101110000 In_A = -4 In_B = 8 ALUSrcB = 2'b00	Out_ALUResult = 4 Out_Zero = 0

addi sp, sp, -6	Instr 16'b0010001010100100 In_A = 14 In_B = 30 ALUSrcB = 2'b10	Out_ALUResult = 8 Out_Zero = 0
lw s1, 4 (sp)	Instr 16'b1001001001000111 In_A = 30 In_B = 14 ALUSrcB = 2'b10	Out_ALUResult = 34 Out_Zero = 0
lui r2, 112	Instr 16'b0011111100101110 In_A = 4 In_B = 2 ALUSrcB = 2'b10	Out_ALUResult = 16'b11110010_00000000 Out_Zero = 0
si t2, 00111111 (shifting a neg A 15 bits)	Instr 16'b1101001111110101 In_A = 16'b1100_0010_0001_0000 In_B = 16'b0010_0000_0101_0100 ALUSrcB = 2'b10	Out_ALUResult = -1 Out_Zero = 0
si t2, 00111111 (shifting a pos A 15 bits)	Instr 16'b1101001111110101 In_A = 16'b0100_0010_0001_0000 In_B = 16'b0010_0000_0101_0100 ALUSrcB = 2'b10	Out_ALUResult = 0 Out_Zero = 0

NOTES:

Williamson Notes:

(9/4)

Starting with a mini scrum meeting (recap of what you did)

Want individual journals to have more detail (what decisions, why did we decide on it (2-3 sentences))

- Like the decision for how we did

Fix document name

Fix instructions (+4 -> +2)

Decide on how we need to do input/output (Need spare opcode, or some other special stuff (will talk about in class)

Other architecture that has a jump instruction that's just an op code, no rd, gives us more range to jump
Want to decide if text is dynamic/fixed in memory, won't really have static data

Fix GCD instruction (check branching negatives)

Shifting; make two if statements very clear

Add to code examples (add recursion, example of how to use br)

Change instruction type names

Write template from what we can do with that

Write normal cycle rtl, can talk about pipelining later ^ will make pipelining easier

Milestone 2 Work:

1. Refactor RTL
2. Component Writing:
    a. ~~Register~~
    b. ~~Mux~~
    c. Memory -> 2 types (data and uhhh something else)
    d. ~~ALU~~
    e. ~~Register File~~
    f. ~~Immediate Genie~~
3. Component Testing
4. Datapath Design
5. Integration plan for testing
    a. Sunday consider
6. Description of Each Control Signal
7. RTL Changes Paragraph - getting them in the 5 stages of pipelining

Notes 10/12:

Blocks of code for examples

Some instructions where second cycle is different; keep them the same (adjust RTL)

Branches and jumps, deal with PC = PC+2 at the top

- Increment in the last step after we have done all the work
- Subtract it
- can forward the not +2 one to the future PCs
- change addressing mode, branches go to PC + imm*2 + 2

Getting stuff into alignment

Assembler option,

Hardcode each main instruction, get pseudo instruction and outputs all the different binary values

ALU- > +- >> << not ALU out (double check all the components and in the list in the correct place)

Register file has instances of register inside of it

~~Add Register file to the component list~~

~~Memory and control also needs to be clocked~~

~~Register File has clocks~~

~~Make sure naming conventions match in component correctly~~

Make sure every single input is actually used

Address has to be a input to memory

~~Specify what happens on each edge for clocked components~~

Branches do not work…?-> when go to run the code at the end, probably going to break when it hits a branch; the fix will take 15 seconds

~~Add reset signals~~

Check U-type RTL (LBI)

1. An updated design document that includes the following:
   - ~~A complete but uncluttered block diagram of the datapath. Neat, hand-drawn and scanned diagrams are acceptable.~~
   - ~~An unambiguous English description of each control signal.~~
   - ~~An updated list of components, based on any parts needed to implement your datapath. Since you now have the datapath design,~~

~~your updated component list should include the basic specs for your control unit.~~

- For each component (aside from control), a brief description of the unit tests to verify the implementation is correct.
- Integration plan for iteratively combining parts into a complete datapath.
- Tests for each step in your integration plan. Be specific enough so that each stage will be tested thoroughly.
- Changes made to the RTL descriptions.
- Be sure you are keeping things you added earlier up to date if you make changes.

2. A partial implementation in your `Implementation` directory:
   - Complete implementations of some components, built according to your spec in Milestone 2.
   - Tests for some components, designed according to your spec in this Milestone.

Assembler-> storing pseudos how???

How detailed does our integration plan need to be???

Yes do it

➔ ALUControl/IntegrationPlanTest for these
➔ ~~RegFile~~
➔ Memory
➔ Integration Plan
➔ Tests for integration plan
➔ File designs for integration plan
➔ Control & Tests

➔ An updated list of the control signals.
➔ Changes made to the RTL descriptions.
➔ Changes to the integration plan.

Tests & Verilog for all components
Tests written for all the integration plans & implementation

REVISED CODE:

```
relPrime: addi sp, sp,-6
sw ra, 0(sp)
sw s0, 2 (sp)
sw s1, 4 (sp)
sw s2, 6 (sp)
add s0, a0, x0
addi s1, x0, 2
addi s2, x0, 1
relLoop: add a0, s0, x0
add a1, s1, x0
jal ra, gcd
add br, s2, x0
beq a0, END
addi s1, s1, 1
jal x0, relLoop
END: add a0, s1, x0
lw ra, 0 (sp)
lw s0, 2 (sp)
lw s1, 4 (sp)
lw s2, 6 (sp)
lbi t0, 16
add sp, sp, t0
jalr x0, 0(ra)
gcd: addi sp, sp,-2
sw ra, 0(sp)
add br, x0, x0
```

beq a0, END1

gcdLoop: add br, x0, x0

beq a1, END2

sub t2, a1, a0

si t2, 00111111

add br, x0, t2

addi t2, x0, 0

beq t2, SubA

add br, x0, a0

beq a1, SubA

sub a1, a1, a0

jal x0, gcdLoop

SubA: sub, a0, a0, a1

jal x0, gcdLoop

END1: add a0, a1, x0

END2: lw ra, 0(sp)

addi sp, sp, 2

jalr x0, 0(ra)

RELPRIME CODE ERRORS:

Adding +4 not +2 for loading and storing, format for jalr, format for si

Theoretical Machine Code: (just need to manually set/add jump to skip doing gcd an extra unwanted time because the code will continue doing it one time after unwantedly)

0010001010100100

0001001000001000

1000001000101000

1001001001001000

1010001001101000

1000001100000000

1001000000100100

1010000000010100

0011100000000000

0100100100000000

0001111100111100

1110101000000000

0011111111011001

1001100100010100

0000000001101100

0011100100000000

0001001000000111

1000001000100111

1001001001000111

1010001001100111

1011000100001111

0010001010110000

0000000100000111

0010001011100100

0001001000001000

1110000000000000

0011111100101001

1110000000000000

0100111100111001

1101010000110001

1101001111110101

1110000011010000

1101000000000100

1101111110111001

1110000000110000

0100111111011001

0100010000110001

0000000010101100

0011001101000001

0000000011001100

0011010000000000

0001001000000111

0010001000100100

0000000100000111

Williamson Notes Milestone 3

- Integration plan needs to be independent, meaning to make subcomponents separately
    - 4-7 steps, think about when we need control
- Add numbers to muxs, control signals
- Assign numbers to states and draw them in the diagram
- Build a table with the lists of all the inputs. Verbal input with which cycle
- Pick one instruction of each type and test them for each module
- Clock stuff seems to lead to problems

Williamson Notes Milestone 4

- Make a pdf without comments -> PUSH TODAY
- Check # of states based on RTL (add a reset state)
- Need to update integration stage blobs
- Think about reset state for PC and SP (weirdness of sp in the register file)
- Put instructions for the Assembler into the main design doc

- Don’t name instantiation with “uut (unit under test)” in the integration name them according to what the component is and what the unit goes into
- Second version of the datapath without control wires
- Instantiate a register called PC for integration test with memory
- Decide on IO (will need to rewrite relPrime code with that)
- What’s gonna happen when the code finishes? (could rerun relPrime and wait for input; will need to add some logic so relPrime doesn’t rerun if the input changes; or could have a control signal that selects another program to run)

Mem Lab:

Keep Mem file 10 bits

Convert 16 bit address into 10 bits

Takes the form of another Verilog file

Write tests that check if it goes over the 10 bit max

(Check for memory writes over 03ff)

Shift 16 bit address 1 to the right to get word addresses

IO:
Want to use opcode to load data in

Individual Work:

Reilly, Spencer: Rework memory

Spencer: Add I/O into register data selection Mux

Document Notes:

Add explanation of BR register in explanation

Mention that branching templates are sudos

Change around memory

Mention types are called I type, R type etc

Make LBI clearer (on instruction table)

Add more component descriptions

Add control signal descriptions

Test other branches as sudo instructions (MAKE SURE IT WORKS)

Make reference sheet

Address ^

Pokemon W

Is PC being set correctly?

Is Output being set correctly?
How to test basic instructions?