Violet-2223b-01

Jackson Shen, Jenna Strassburger, and Ethan Norfleet

Milestone 1

Design Description

We will be working on a load-store architecture, similar to Risc-V. Our focus will be on ensuring the processor runs quickly, with a secondary focus on ease of programming for the user. Consistency will be maintained when possible, however the main goal of our processor will be to execute the desired program as fast as possible. In order to maximize efficiency, we are willing to work with a larger instruction set, as a trade-off for the versatility of a particular instruction.

Measurement of Performance

The key measure of performance will be the amount of time it takes for the smallest relatively prime program, implemented with Euclid's algorithm, to execute. Additionally, we will consider the raw values of the average cycles per instruction and the clock frequency.

Register Usage (16 bit registers)

Register	Name	Description	Saver
x0	zero	Zero constant	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3-x4	t0-t1	Temp registers	Caller
x5-x9	s0-s4	Saved registers	Callee
x10-x11	a0-a1	Return registers	Caller
x12-x14	a2-a4	Fn-args	Caller
x15	at	Assembler temp	Caller

Machine Language Instruction

Inst	Name	FMT	ID	Desc	Comments
add	Add	R	0000	R[rd] = R[rs1]+R[rs2]	3 registers operands; add
grt	Greater than?	R	0001	R[rd] = (rs1>rs2)?0:1	3 registers operands; compare the argument. Return 0 if rs1 <= rs2. Return 1 otherwise
sub	Subtract	R	0010	R[rd]=R[rs1]-R[rs2]	3 registers operands; subtract
eq	Equal?	R	0011	R[rd] = (rs1==rs2)?0:1	3 registers operands; compare the arguments. Return 0 if not equal. Return 1 otherwise
jalr	Jump and link register	R	0100	R[rd] = PC + 2 PC = R[rs1] + PC	2 register operands; jump the amount of the immediate relative to rs1
lui	Load upper immediate	I	0101	R[rd] = SE(imm)<<8	2 register operands; load constant and shift 8
jal	Jump and link	I	0110	R[rd] = PC + 2 PC = PC + SE(imm)<<1	2 register operands; jump the amount of the immediate
addi	Add immediate	M	1000	R[rs1] = R[rs2]+SE(imm)	2 register operands; add immediate to register
lw	Load word	M	1001	R[rs1] = M[R[rs2] + SE(imm)]	2 register operands; word from memory to register
sw	Store word	M	1010	M[R[rs1] + SE(imm)] = R[rs2]	2 register operands; word from register to memory
bne	Branch not equal	M	1011	if(rs1!=rs2 + SE(imm)) PC = PC + R[rs1]<<1	2 register operands; change raw address if not equal
wri	Write	M	1100	If (rs1 = <special mem place>):	Handles writing to output port

				<special mem stuff> = R[rs2]	
rea	Read	M	1101	If (rs1 = <special mem place>): R[rs2] = <special mem stuff>	Handles reading from input port

Instruction Diagrams

R-type instructions

rs2	rs1	rd	4 bit ID

I-type instructions

8 bit immediate	rd	4 bit ID

M-type instructions

rs2	rs1	4 bit immediate	4 bit ID

Addressing mode and dealing with immediates

We will be using relative addressing, where branching is relative to the current pc. We will sign-extend immediates.

Calling conventions

The caller will save registers that it will need to use in the future like return address and temporaries. All other registers will be saved by the callee. Each function keeps track of its usage of the stack. It will clear out information and return the stack pointer to its former location before it returns. Finally, a function will never modify any part of the stack above its original location.

Sample RelPrime Assembly Code

Given Code

```
int relPrime(int n) {
  int m;
  m = 2;
  while (gcd(n, m) != 1) {  // n is the input from the outside world
    m = m + 1;
  }
  return m;
}
```

Translated to Assembly

```
RELPRIME:
addi sp, sp, -12
sw ra, sp, 0
sw a0, sp, 4
addi a1, x0, 2
sw a1, sp, 8

LOOP:
addi a1, a1 1
sw a1, sp, 8
jal ra, GCD
lw a1, sp, 8
bne a0, 1, LOOP
lw a0, sp, 8
lw ra, sp, 0
addi sp, sp, 12
jalr a0, 0(ra)
```

Address	Assembly	Machine Code	Comments
0x0004	RELPRIME: addi sp, sp, -12	0100001000101000	
0x0006	sw ra, sp, 0	0000001000011010	
0x0008	sw a0, sp, 4	0100001010101010	
0x000a	addi a1, x0, 2	0010000010111000	
0x000c	sw a1, sp, 8	1000001010111010	

0x000e	LOOP: addi a1, a1 1	0001101110111000	
0x0010	sw a1, sp, 8	1000001010111010	
0x0012	jal ra, GCD	0000111000010110	+0x0e
0x0014	lw a1, sp, 8	1000001010111001	
0x0016	bne a0, 1, LOOP	1000000110101011	-0x08
0x0018	lw a0, sp, 8	1000001010101001	
0x001a	lw ra, sp, 0	0000001000011001	
0x001c	addi sp, sp, 12	1100001000101000	
0x001e	jalr a0, ra, 0	0000000110100100	

Sample GCD Assembly Code

Given Code

```
int
gcd(int a, int b)
{
  if (a == 0) {
    return b;
  }

  while (b != 0) {
    if (a > b) {
      a = a - b;
    } else {
      b = b - a;
    }
  }

  return a;
}
```

Translated to Assembly

```
GCD:
bne a0, x0, ENDIF1
jalr a1, 0(ra)
ENDIF1:
addi t0, x0, 1
LOOP2:
    grt t1, a0, a1
    bne a0, x0, ENDIF2
    sub a0, a0, a1
    bne x0, t0, ENDELSE
    ENDIF2:
    sub a1, a1, a0
    ENDELSE:
    bne t0, x0, LOOP2
jalr a0, 0(ra)
```

Address	Assembly	Machine Code	Comments
0x0020	GCD: bne a0, x0, ENDIF1	0100000010101011	+0x04
0x0022	jalr a1, ra, 0	0000000110110100	
0x0024	ENDIF1: addi t0, x0, 1	0001000000111000	
0x0026	LOOP: grt t1, a0, a1	1011101001000001	
0x0028	bne a0, x0, ENDIF2	0110000010101011	+0x06
0x002a	sub a0, a0, a1	1011101010100010	
0x002c	bne x0, t0, ENDELSE	0100001100001011	+0x04
0x002e	ENDIF2: sub a1, a1, a0	1010101110110010	
0x0030	ENDELSE: bne t0, x0, LOOP	0110000000111011	-0x0a
0x0032	jalr a0, ra, 0	0000000110100100	

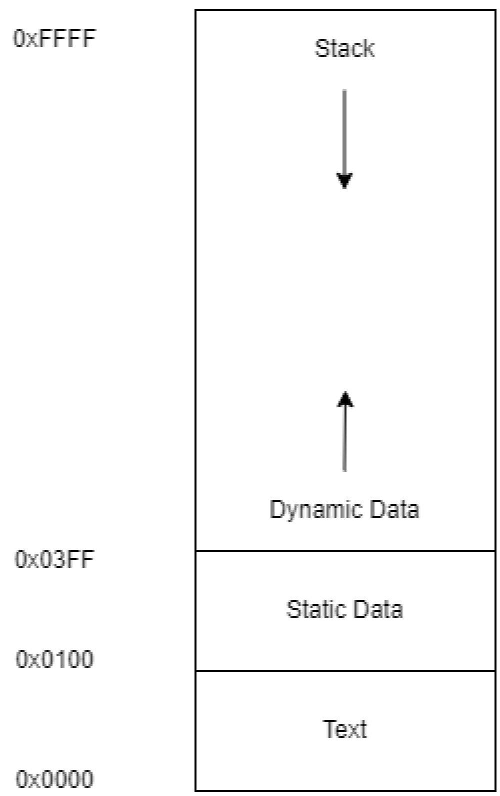
Assembly Code for Common Operations

Operation	Code
Load address	lui rd, SYMBOL[15:8] addi rd, SYMBOL[7:0]
Looping	LOOP: #do something in the loop bne t0, t1, LOOP
Conditional	bne t0, t1, ENDIF #instructions to execute if t0 == t1 ENDIF:
Recursion	REC: bne a0, x0, ENDIF addi pc, pc, -8

	sw pc, ra, 0
	sw pc, a0, -4
	addi a0, a0, -1
	jal ra, REC
	lw pc, ra, 0
	addi pc, pc, 8
	
	ENDIF: jal ra

Reading from input port	rea, SPECIAL, t0
Writing to output port	wri t0, SPECIAL Takes whatever value is at t0 and stores it in the SPECIAL memory location

Memory Map


0xFFFF
Stack
Dynamic Data
0x03FF
Static Data
0x0100
Text
0x0000

Milestone 2

RTL Description

R Type

Step	add/sub	grt/eq	jal/jalr
Instruction Fetch	IR <= Mem[PC] PC <= PC +2		
Instruction Decode	A <= Reg[IR[8:11]] B <= Reg[IR[12:15]]		
Execution	add: ALUOut <= A + B Sub: ALUOut <= A - B	ALUOut <= A - B	ALUOut <= PC+2
Mem Access Set Reg/Add	Reg[7:4] <= ALUOut	grt: Reg[7:4] = reg eq: Reg[7:4] = zero	Reg[IR[7:4]] <= ALUOut jal: ALUOut <= PC + SE(imm) << 1 jalr: ALUOut <= A + SE(imm) << 1
MDR/ Set Registers			PC <= ALUOut

M Type

Step	addi	lw	sw
Instruction Fetch	IR <= Mem[PC] PC <= PC +2		
Instruction Decode	A <= Reg[IR[8:11]] B <= Reg[IR[12:15]]		
Execution	ALUOut <= B + SE(imm)		
Mem Access Set Reg/Add	Reg[IR[11:8]] = ALUOut	MDR <= Memory[ALUOut]	Mem[ALUOut] = A

MDR/ Set Registers		Reg[IR[11:8]] <= MDR	

I Type

Step	bne	wri	rea
Instruction Fetch	IR <= Mem[PC] PC <= PC +2		
Instruction Decode	A <= Reg[IR[8:11]] B <= Reg[IR[12:15]]		
Execution	ALUout <= A - B		
Mem Access Set Reg/Add	If (ALUout != 0): ALUout <= PC + SR(Imm)<<1	if(A == <special>): <output> = R[IR[15:12]]	if(A == <special>): R[IR[15:12]] = <input>
MDR/ Set Registers	PC <= ALUOut		

Naming Conventions

Module - <name of component>_component
Cycle - <name of subsystem>_cycle
Test bench - <component-to-test>_tb
Input - <abbrev of control>_in
Output - <description of result>_out
Wires - <name of where it starts>-<name of where it ends>-wire
Reg - <name of where it starts>-<name of where it ends>-reg
Clock - <name of component>-clk

Component Specifications

Rising edge

Component	Input	Output	Behavior	RTL Symbols
2-way Mux	OP IN0[15:0] IN1[15:0] RESET	OUT[15:0]	If OP[0] = 0, output IN0[15:0]. If OP[0] = 1, output IN1[15:0].	
4-way Mux	OP[1:0] IN0[15:0] IN1[15:0] IN2[15:0] IN3[15:0] RESET	OUT[15:0]	If OP[1:0] = 00, output IN0[15:0]. If OP[1:0] = 01, output IN1[15:0]. If OP[1:0] = 10, output IN2[15:0]. If OP[1:0] = 11, output IN3[15:0].	
ALU	INSTID[3:0] IN0[15:0] IN1[15:0] RESET	ALUOUT[15:0] ZERO[0] POS[0]	Check the value of INST[3:0]. If INST[3:0] = 0000, 0100, 0110, 1000, 1001, 1010, or 1011, ALUOUT[15:0] = IN0[15:0] + IN1[15:0]. Otherwise, ALUOUT[15:0] = IN0[15:0] - IN1[15:0].	+ -
Instruction Register	CLK INST[15:0] RESET	RS1[3:0] RS2[3:0] RD[3:0]	On the rising edge, load INST[15:12] into RS2, INST[11:8] into	IR

			RS1, and INST[7:4] into RD	
Inst Mem	ADDR[15:0] CLK	INST[15:0]	Load the instruction from the specified memory location	IMEM
Reg File	CLK RS1[3:0] RS2[3:0] RD[3:0] WRITEDATA[15:0] RESET WRITE	REG1[15:0] REG2[15:0]	On the rising edge, reads the values of RS1[3:0] and RS2[3:0] and outputs REG1 and REG2	Reg
Data Mem	CLK ADDRESS[15:0] WRITEDATA[15:0] WRITE	MEM[15:0]	On the rising edge, write WRITEDATA[15:0] FROM ADDRESS[15:0] into MEM[15:0]	Mem
Imm. Gen	CLK RESET INST[15:0]	IMM[15:0]	On the rising edge, generate an immediate (Sign extend INST[15:0] or shift by one depending on ID)	<< SE
Reg	CLK RESET WRITE INPUT[15:0]	REGOUT[15:0]	On the rising edge, store the value of INPUT[15:0]	A, B, PC, MDR
SmallReg	CLK RESET WRITE INPUT[3:0]	REGOUT[3:0]	On the rising edge, store the value of INPUT[3:0]	Forwarding Unit

Process to Verify RTL

- Peer review RTL
- Ran each RTL with an instruction and compared result with our expected

Changes to Assembly and Machine Language

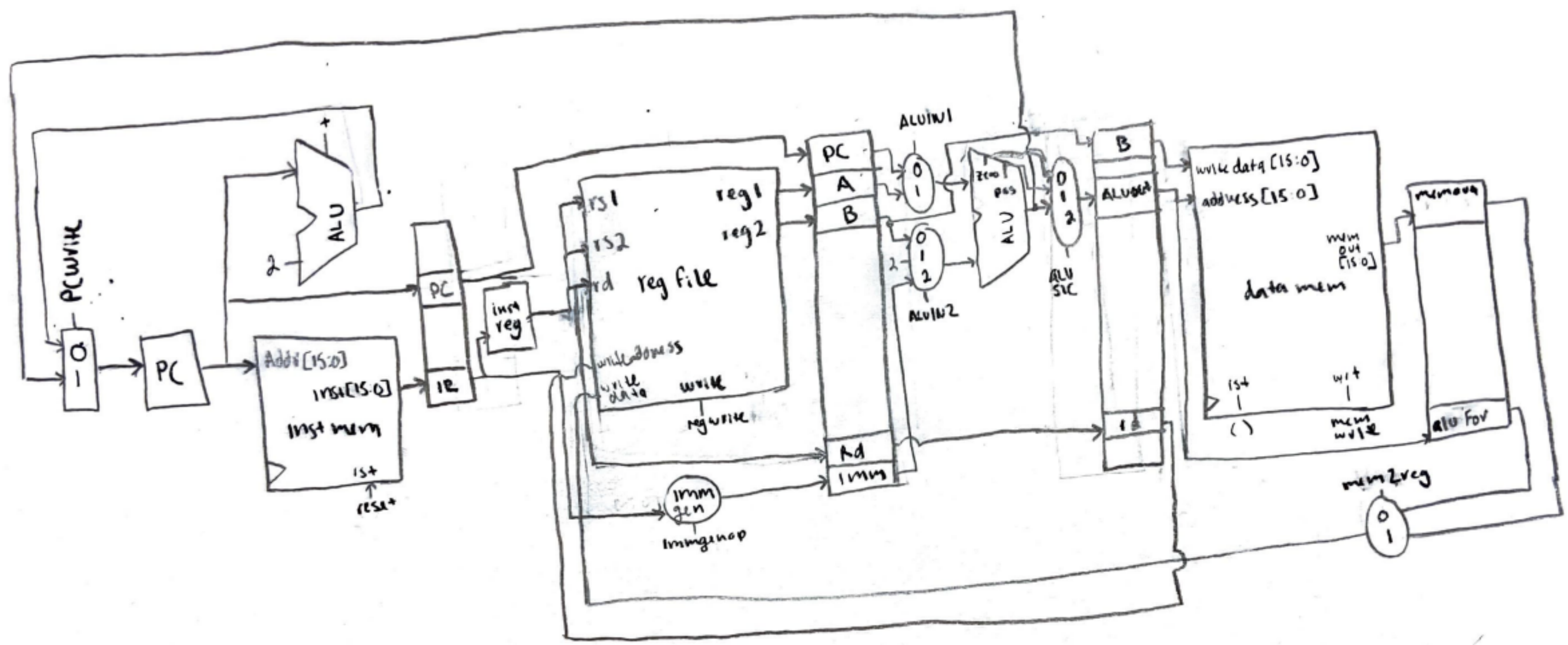
Assembly Changes

- We corrected our assembly to take a 4 bit ID number, rather than a 2 bit op and a 2 bit func
- Made some minor naming changes for consistency. EX: calling the first register rs1 and the second register rs2
- Corrected logic behind reading and writing to input ports
- To adjust for size, corrected all references to PC + 4 to PC + 2

Machine Changes

- Corrected translation of immediates for labels

Datapath Diagram



Control

Instruction	Execution					Memory Access		Writeback		
Control	IMMGENOP	ALUOP	ALUIN1	ALUIN2	ALUSRC	MEMREAD	MEMWRITE	PCWRITE	mem2reg	reset
add	0	0	0	0	0	0	0	0	0	0
sub	0	1	0	0	0	0	0	0	0	0
grt	0	1	0	0	1	0	0	0	0	0
eq	0	1	0	0	2	0	0	0	0	0
jal	2	0	1	2	0	0	0	1	0	1
jalr	0	0	1	1	0	0	0	1	0	1
addi	0	0	0	2	0	0	0	0	0	0
lui	3	1	0	2	0	0	0	0	0	0
lw	0	0	0	2	0	1	0	0	1	0
sw	0	0	0	2	0	0	1	0	0	0
bne	2	1	1	0	0	0	0	1	0	1
wri	0	1	0	0	0	0	1	0	0	0
rea	0	1	0	0	0	1	0	0	0	0

Unit Test Description Per Component

R-Type ALU

A > B	A = B	ALUOP	ALUOUT	grt	zero
Y	N	+	A + B	1	0
N	Y	+	A + B	0	1
N	N	+	A + B	0	0
Y	N	-	A - B	1	0
N	Y	-	A - B	0	1
N	N	-	A - B	0	0

Imgen

ImmCon	Output
0	Old Immgen value
1	Inst

jal and jalr ALU

ALUOp	Output
+	I1 + I2
-	I1 - I2

jal and jalr PC Change

PCWrite	Output
0	oldPC
1	ALUOut

2-way Mux (Milestone 4 Update)

OP[0]	IN0[15:0]	IN1[15:0]	output
0	0000000000000000	0000000000000001	0000000000000000
1	0000000000000000	0000000000000001	0000000000000001

4-way Mux (Milestone 4 Update)

OP[1:0]	IN0[15:0]	IN1[15:0]	IN2[15:0]	IN3[15:0]	output
00	00000000 00000000	00000000 00000001	00000000 00000010	00000000 00000011	00000000 00000000
01	00000000 00000000	00000000 00000001	00000000 00000010	00000000 00000011	00000000 00000001
10	00000000 00000000	00000000 00000001	00000000 00000010	00000000 00000011	00000000 00000010
11	00000000 00000000	00000000 00000001	00000000 00000010	00000000 00000011	00000000 00000011

Reg File (Milestone 4 Update)

rs1[3:0]	rs2[3:0]	rd[3:0]	writedata[15:0]	Reg1[15:0]	Reg2[15:0]
0000	0001	0000	00000000000 00101	00000000 00000101	00000000 00000000
0000	0001	0001	00000000000 00101	00000000 00000000	00000000 00000101
0000	0001	0000	00000000000 00111	00000000 00000111	00000000 00000000
0000	0001	0001	00000000000 00111	00000000 00000000	00000000 00000111

Put 5 into reg1
Put 5 into reg2
Put 7 into reg1
Put 7 into reg2

ALU (Milestone 4 Update)

For +, loop through 0000, 0100, 0110, 1000, 1001, 1010, and 1011. For - loop through 0001, 0010, 0011, 0101, 0111, 1100, 1101, 1110, and 1111

INST[3:0]	IN0[15:0]	IN1[15:0]	ALUOUT[15:0]	ZERO?[0]	POS?[0]
+	0000000000000000	0000000000000000	0000000000000000	1	0
+	0000000000000001	0000000000000001	0000000000000010	0	1
+	1111111111111111	1111111111111111	1111111111111110	0	0

-	0000000000000000	0000000000000000	0000000000000000	1	0
-	0000000000000010	0000000000000001	0000000000000001	0	1
-	1111111111111110	1111111111111111	1111111111111111	0	0

Integration Testing Plan
Inst fetch:
Hard-code instructions into memory and check whether they are read into the instruction register properly

Inst decode:
Hard-code instruction inputs from IR and check whether rs1, rs2, imm, and rd.

Execution:
Loop over all op codes, test all alu input sources with both positive and negative numbers.

Mem access:
Hard code values and locations, and check whether they were loaded into memory correctly

MDR load:
Check whether PC updates properly based on controls.

Specific Instructions

add:

rs1[3:0]	rs2[3:0]	MEM[0000]	MEM[0001]	rd[3:0]	MEM[0010]
0000	0001	0000000000000000	0000000000000000	0010	0000000000000000
0000	0001	0000000000000001	0000000000000001	0010	0000000000000010
0000	0001	1111111111111111	1111111111111111	0010	1111111111111110

grt:

rs1[3:0]	rs2[3:0]	MEM[0000]	MEM[0001]	rd[3:0]	MEM[0010]
0000	0001	0000000000000000	0000000000000000	0010	0000000000000000

0000	0001	000000000 0000000	000000000 0000001	0010	000000000 0000000
0000	0001	000000000 0000001	000000000 0000000	0010	000000000 0000001

sub:

rs1[3:0]	rs2[3:0]	MEM[0000]	MEM[0001]	rd[3:0]	MEM[0010]
0000	0001	000000000 0000000	000000000 0000000	0010	000000000 0000000
0000	0001	000000000 0000010	000000000 0000001	0010	000000000 0000001
0000	0001	1111111111 111110	000000000 0000001	0010	1111111111 111111

eq:

rs1[3:0]	rs2[3:0]	MEM[0000]	MEM[0001]	rd[3:0]	MEM[0010]
0000	0001	000000000 0000000	000000000 0000000	0010	000000000 0000001
0000	0001	000000000 0000000	000000000 0000001	0010	000000000 0000000
0000	0001	000000000 0000001	000000000 0000000	0010	000000000 0000000

jalr:

rs1[3:0]	MEM[0000]	oldPC	rd[3:0]	MEM[0010]	newPC
0000	000000000 0000000	00000000 00000000	0010	000000000 0000010	00000000 00000000
0000	000000000 0000000	00000000 00000001	0010	000000000 0000011	00000000 00000001
0000	000000000 0000001	00000000 00000000	0010	000000000 0000010	00000000 00000001

lui:

imm	rd[3:0]	MEM[0010]
00000000	0010	000000000 0000000

00000001	0010	0000000100000000
111111111	0010	11111111100000000

jal:

imm	oldPC	rd[3:0]	MEM[0010]	newPC
00000000	0000000000000000	0010	0000000000000010	0000000000000000
00000000	0000000000000001	0010	0000000000000011	0000000000000001
00000001	0000000000000000	0010	0000000000000010	0000000000000010

addi:

rs1[3:0]	rs2[3:0]	MEM[0000]	MEM[0001]	imm
0000	0001	0000000000000000	0000000000000000	00000000
0000	0001	0000000000000001	0000000000000001	00000000
0000	0001	1111111111111110	1111111111111111	00000001

lw:

rs1[3:0]	rs2[3:0]	MEM[0000]	MEM[0001]	imm
0000	0001	0000000000000001	0000000000000000	00000001
0000	0001	0000000000000001	0000000000000001	00000000
0000	0001	1111111111111111	1111111111111110	00000001

sw:

rs1[3:0]	rs2[3:0]	MEM[0000]	MEM[0001]	imm
0000	0001	0000000000000000	0000000000000001	00000001

0000	0001	0000000000000001	0000000000000001	00000000
0000	0001	1111111111111110	1111111111111111	00000001

bne:

rs1[3:0]	rs2[3:0]	MEM[0000]	MEM[0001]	imm	oldPC	newPC
0000	0001	0000000000000000	0000000000000000	00000000	00000000	00000000
0000	0001	0000000000000000	0000000000000000	00000001	00000000	00000000
0000	0001	0000000000000001	0000000000000000	00000000	00000000	00000010
0000	0001	0000000000000001	0000000000000000	00000001	00000000	00000000

wri:

rs1[3:0]	rs2[3:0]	MEM[0001]	<special mem place>	Old <special mem stuff>	New <special mem stuff>
0000	0001	0000000000000001	0000	0000000000000000	0000000000000001
0000	0001	0000000000000001	0011	0000000000000000	0000000000000000
0011	0001	0000000000000001	0000	0000000000000000	0000000000000000
0011	0001	0000000000000001	0011	0000000000000000	0000000000000001

rea:

rs1[3:0]	rs2[3:0]	Old MEM[0001]	<special mem place>	<special mem stuff>	New MEM[0001]
0000	0001	0000000000000001	0000	0000000000000000	0000000000000000
0000	0001	0000000000000001	0011	0000000000000000	0000000000000001

0011	0001	0000000000000001	0000	0000000000000000	0000000000000001
0011	0001	0000000000000001	0011	0000000000000000	0000000000000000

Control List

- pcwrite - allows to write to PC and update
- regWrite - allows to write to registers
- immGenOp- tells the immediate generator how to handle the immediate
- ALUIN1
- ALUIN2
- ALUSRC
- mem2Reg
- reset
- memWrite- allows to write in mem

Changes to RTL

- Small changes to existing RTL
  - All references to 4 were changed to 2 to account for the size of the instruction set
  - Names of components were changed to match the names in the component table
  - Rearranged cycles for pipeline
- Updated components
  - Corrected inputs and outputs
  - Applied the correct naming conventions
  - Updated behavior description
  - Added an overall register component rather than specific registers (A,B, PC)
  - Added reset and data write enable inputs

ADD FINAL THOUGHTS