Comp Arch Crusaders' CSSE232 Accumulator

Last Edit: Jan 23, 2024

https://docs.google.com/document/d/1-wqjMJYvDuMb4TwNroaUSae_RZOBAOsYS3RniXAOliw/edit?usp=sharing

—

Thomas Bioren
Jada Hunter-Hays
Dan Seiler
Elenaor Brooks

General Specifications

Processor with:

- 16-bit memory address bus
- 16-bit memory data bus
- Mechanism for basic input
- Mechanism for basic output
- General computation support
- Parameterized and nested procedures support

Processor Overview

We selected the Accumulator architecture in order to prioritize simplicity. In alignment with the principle “Simplicity favors regularity”, we opted for a minimal number of instructions and instruction types (3) in our core set and 4 registers. There is a single input register and a single output register. Our general purpose register “Reggie” can be used to add additional parameters. We chose to include a status register in order to provide a range of values for a single compare. Branching is done by using the values in status register.

Measure Performance: Speed it takes to complete each operation

For the accumulator, we have one register available (“Reggie”), and three special purpose registers reserved for input (“in”), output (“out”), and status register (“sr”).

Core Instruction Set

Name	Type	Opcode	Description	
LW	I-Type	0x0	Reggie = Mem[SE(imm)]	Load mem[addr] into Reggie
SW	I-Type	0x1	Mem[SE(imm)] = Reggie	Store the value of Reggie into mem[imm]
LMEM	I-Type	0x2	Reggie = Mem[Mem[SE(imm)]]	Load MEM[MEM[addr]] into Reggie
SMEM	I-Type	0x3	MEM[MEM[SE(imm)]] = Reggie	Store the value of

				Reggie into MEM[MEM[addr]]
ADD	I-Type	0x4	Reggie = Reggie + Mem[SE(imm)]	Increment Reggie by mem[imm]
SUB	I-Type	0x5	Reggie = Reggie - Mem[SE(imm)]	Decrement Reggie by mem[imm]
ADDI	I-Type	0x6	Reggie = Reggie + SE(imm)	Increment Reggie by the immediate
CMPI	I-Type	0x7	If Reggie < SE(imm): sr[1:0] = **01** If Reggie = SE(imm): sr[1:0] = **10** If Reggie > SE(imm): sr[1:0] = **11**	Compare Reggie and the immediate and update sr 3:0 accordingly
CMP	I-Type	0x8	If Reggie < Mem[SE(imm)] : sr[1:0] = **01** If Reggie = Mem[SE(imm)] : sr[1:0] = **10** If Reggie > Mem[SE(imm)] : sr[1:0] = **11**	Compare Reggie and the mem[imm] and update sr 3:0 accordingly
B	B-Type	0x9	If comp_code == sr[1:0] PC = destination address Comp_code 00 == unconditional branch	Branch to dest_addr if comp_code equals SR[3:0]
BMEM	B-Type	0xA	If comp_code == sr[1:0] PC = destination address Comp_code 00 == unconditional branch	Branch to location stored at address of immediate.
LLI	I-Type	0xB	Reggie[11:0] = SE(imm)	Load the immediate into the most significant bits of reggie. The remaining upper bits are 0.
SLLI	I-Type	0xC	Reggie = Reggie << SE(imm)[2:0]	Logical Left shift Reggie by the immediate
LI	I-Type	0xE	input = Reggie	Put the value of input into Reggie. Any immediate provided will be ignored.
SO	I-Type	0xF	output = Reggie	Put the value of Reggie

				into output

Registers

- Input: The input register
- Ouptut: The output register
- Reggie: The main accumulator register
- SR: The status register used to store flags
    - bits 1:0 are used for comparison operations
        - If bit 0 is 1: then Reggie is less than compared value
        - If bit 1 is 1: then Reggie is equal to compared value
        - If bit 2 is 1: then Reggie is greater than compared value
        - If bit 3 is one then Reggie is zero

Memory Map

Type	Stack	Stack Pointer	Return Address	Text
Bits	0xFFFF:0x0400	0x03FFF:0x03FFE	0x03FFD:0x03FFC	0x3FFB:0x0000

The stack starts from the highest memory location (0xFFFF) and expands down. PC starts at 0x0000 and goes up to 0x3FFF.

Instruction Formats

Bits	15:14	13:12	11:8	7:4	3:0
I-Type	Immediate				Opcode
B-Type	Comparison Code	Memory/Destination Address			Opcode

Note: the higher immediate index, the more significant the bit. Refer to LUI and LLI to see how most and least significant bits are read

Procedure Calling Conventions

Since we only have one register, we need to get creative with how we pass arguments and return values. We will use special locations in memory for this.

Assembly Instructions: MEM[0x03FF:0x0000]

Return Address: MEM[0x0401:0x0400]

Stack Pointer: MEM[0x0403:0x0402]

Arguments/Return Values: MEM[0x040F:0x0404] (The first argument is at the lowest memory address in big endian format). No guarantee arguments will be preserved.

Saved Variables: MEM[0x041F:0x0410]

Temporary Variables: MEM[0x043F:0x0420]

Steps to a Proper Procedure Call:

1. Backup any temporary variables you do not want lost onto the stack.
2. Store any arguments into the arguments section of memory (MEM[11:2]).
3. Store the return address into the RA section of memory (MEM[1:0]).
4. Branch to the function.
5. The function should back up any saved variables it will modify.
6. The function should do whatever it's supposed to and when it's done, it stores its results into the return section in memory.
7. Branch to the value stored in the RA section of memory.

Example Basic Procedure Call:

```
MAIN:
      # Create a saved and temp variable
      LLI 32
      SW S0
      LLI 10
      SW T0
      # Back up T0 by:
      # 1. Load SP into Reggie
      # 2. Decrement Reggie by 2
      # 3. Store Reggie back into SP
      # 4. Load T0 into Reggie
      # 5. Store Reggie into the memory location where SP points
      LW SP
```

```
        ADDI -2
        SW SP
        LW T0
        SMEM SP
        # Store the return address into RA
        LLI HERE_AFTER_CALL
        SW RA
        B 0b00 DO_SOMETHING
HERE_AFTER_CALL:
        # Restore T0 by:
        # 1. Load the item in memory that SP is pointing to into Reggie
        # 2. Storing Reggie into T0
        # 3. Loading SP into Reggie
        # 4. Incrementing Reggie by 2
        # 5. Storing Reggie back into SP
        LMEM SP
        SW T0
        LW SP
        ADDI 2
        SW SP
        LW S0
        # Output
        SO

DO_SOMETHING:
        # Back up the saved variable
        LW SP
        ADDI -2
        SW SP
        LW S0
        SMEM SP
        # Do whatever the function is supposed to do
        # (this overwrites S0 and T0)
        LLI 64
        SW S0
        LLI 128
        SW T0
        # Replace the backed up variable
        LMEM SP
        SW S0
        LW SP
        ADDI 2
```

```
    SW SP
    # Branch back to caller
    LW RA
    BMEM RA
```

Multi Cycle RTLs

We have decided to make a multicycle processor and have updated our datapath and RTLs accordingly.

Overview of Cycles

1. **Fetch**: Increments PC and loads instruction register IR
2. **Decode**: Immediate, comp code register, and ALUOut register are loaded with respective sign extended values from IR. If it is not a branching instruction, ALUout's value is nonsense
3. **Cycle 3**: ALU performs operations for CMPi and ADDI. For ADD/SUB/LW, memory register is loaded with the value read from Memory with immediate being the direct address. Other operations perform loading and reading from Reggie, and BMEM checks SR against the value in CompCode.
4. **Cycle 4**: Now that a value has been retrieved from memory and loaded into MDR, ALU can operate with inputs Reggie and MDR and load the result into ALUout. B compares the value in SR with the value in CompCode. Now that MDR is updated, BMEM/B is able to update PC if needed. For CMPi/ADDI, MDR is loaded with memory using address ALUout.
5. **Cycle 5**: For ADD/SUB/LW/SW, MDR register is loaded with value read from Memory. It uses the direct address read from register ALUout. ADDI loads Reggie with value from MDR.
6. **Cycle 6**: 1 bit isZero and the 1 most significant bit from ALUOut is sent to Status Register, and based on those inputs, SR determines its value to be output on the rising clock edge.

https://docs.google.com/spreadsheets/d/17RdYQc5h7aGAGfkjJEd-nXyBQ9pQtTjd0pVD0kygmy0/edit?usp=sharing

Multicycle RTL

Below display split into 2 separate tables for ease of reading visual.

	LI/SO	CMPi	CMP	ADDI	ADD/SUB
fetch	PC <= PC + 2				
	IR <= Mem[PC]				
decode	Imm <= SE(IR[15:4])				
	CompCode <= SE(IR[15:14])				
	ALUout <= SE(IR[11:4])				
	LI: Reggie <= Input **SO**: Output <= Reggie	ALUout <= Reggie - imm	MDR <= Mem[imm]	ALUout <= Reggie + imm	MDR <= Mem[imm]
		if ALUout isZero, sr[1:0] = 00 if ALUout [0] == 0, sr[1:0] = 11 If ALUout[0] == 1, sr[1:0] = 01	ALUout <= Reggie - MDR	Reggie <= ALUout	ALUout <= Reggie op MDR
			if ALUout isZero, sr[1:0] = 00 if ALUout [0] == 0, sr[1:0] = 11 If ALUout[0] == 1, sr[1:0] = 01		Reggie <= ALUout

	LW/SW	B	BMEM	LMEM/SMEM	SLLI/LLI
fetch	PC <= PC + 2				
	IR <= Mem[PC]				
decode	Imm <= SE(IR[15:4])				
	CompCode <= SE(IR[15:12])				
	ALUout <= SE(IR[11:4])				

	SW: Mem[imm] <= Reggie **LW**: Reggie <= Mem[imm]	if(sr == comp_code OR comp_code == 00) PC <= imm	if(sr == comp_code OR comp_code == 00) MDR <= Mem[ALUout]	MDR <= Mem[imm]	**LLI:** Reggie[11:0] <= imm[11:0] **SLLI:** Reggie = Reggie << imm[3:0]
			if(sr == comp_code OR comp_code == 00) PC <= Mem[MDR]	**LMEM:** Reggie <= Mem[MDR] **SMEM:** Mem[MDR] <= Reggie	

Double Checking RTLs: First, check that it updates the PC by 2. Then check that it accesses memory/mem address by bytes. Confirm all immediates are sign extended, and we are using direct addressing.

Assembly

A rudimentary assembler for this architecture can be found in the repository. It is located at /implementation/asm2bin. Use the assembler by running `Python asm2bin.py` in the terminal, then input your desired assembly line. The assembler expects all numbers to be in decimal except the comparison codes, which it expects in binary. The numbers should have no formatting (no 0x..., 0b..., etc). In the future, the assembler will assemble entire assembly programs.

Example Assembly for RelPrime

Address	Assembly	Machine Code	Comments
0x0000	LW A0	0100000001000000	Load in n
0x0002	SW S0	0100000100000001	Store n in a saved register
0x0004	LLI 0	0000000000001001	Set Reggie to 0
0x0006	ADDI 2	0000000000100100	Set Reggie to 2
0x0008	SW S1	0100000100100001	Store value of Reggie (m) into S1
0x000A	LW S0	0100000100000000	Load S0 into Reggie

0x000C	SW A0	0100000001000001	Store Reggie into A0 (store n into 1st param for GCD)
0x000E	LW S1	0100000100100000	Load S1 into Reggie
0x0010	SW A1	0100000001100001	Store Reggie into A1 (store m into 2nd param for GCD)
0x0012	LLI 0x0018	0000000110001001	Store RA
0x0014	SW RA	0100000000000001	Store the RA into RA
0x0016	B 0b00 GCD	0000001010000111	Branch to GCD (0x28)
0x0018	LW S1	0100000100100000	Reggie = S1
0x001A	ADDI 1	0000000000010100	Increment S1 by 1
0x001C	SW S1	0100000100100001	S1 = Reggie
0x001E	LW A0	0100000001000000	Reggie = A0
0x0020	CMPI 1	0000000000010101	Compare Reggie to 1
0x0022	B 0b10 DONE	1000001001100111	Branch to DONE (0x26) if CMP == 0
0x0024	B 0b00 LOOP	0000000010100111	Branch to LOOP (0x0A)
0x0026	SO	0000000000001101	Store the result in the output
0x0028	LW A0	0100000001000000	Reggie = A0
0x002A	CMPI 0	0000000000000101	Compare Reggie to 0
0x002C	BMEM 0b10 RA	1010000000001110	Branch to RA if CMP == 0b10
0x002E	LW A0	0100000001000000	Reggie = A0
0x0030	CMP A1	0100000001100110	Compare Reggie to A1
0x0032	B 0b11 TRUE	1100001111000111	Branch to TRUE (0x3C) if CMP ==

			0b11
0x0034	LW A1	0100000001100000	Reggie = A1
0x0036	CMPI 0	0000000000000101	Compare Reggie to 0
0x0038	B 0b10 DONE_GCD	1000010010000111	Branch to DONE if CMP == 0b10
0x003A	B 0b00 LOOP_GCD	0000001011100111	Branch to LOOP_GCD (0x2e) unconditionally
0x003C	SUB A1	0100000001100011	Subtract A1 from Reggie
0x003E	SW A0	0100000001000001	A0 = Reggie
0x0040	B 0b00 CHECK	0000001101000111	Branch unconditionally to CHECK (0x34)
0x0042	LW A1	0100000001100000	Reggie = A1
0x0044	SUB A0	0100000001000011	Subtract A0 from Reggie
0x0046	B 0b00 CHECK	0000001101000111	Branch unconditionally to CHECK (0x34)
0x0048	BMEM 0b00 RA	1010000000001110	Branch back to caller

Naming Conventions

Wires:

- Naming: lowercase, with underscore for spaces e.g. data_bus, address_line, control_signal.
- Signal Direction: use prefixes like in_ or out_ to indicate input or output, e.g., in_data, out_result.
- Bit Width: specify the width, e.g., data_bus[7:0] for an 8-bit bus.

Registers:

- Naming: no underscores, by designated name from RTLs
- Bit Width: indicate the size, e.g., reg_data[15:0] for a 16-bit register.

Inputs and Outputs:
- Naming: begins or ends with in and out unless it's 1 bit or an opcode
- W indicates write
- R indicates read

Clocks:
- Naming: CLK.
- Edge Specification: if/whether the rising (posedge) or falling (negedge) edge is important.

Test Benches:
- Naming: begins with the name of module being tested and ends with_tb for test bench e.g., module_tb

Modules:
- Naming: choose clear 1 word names representing the functionality, e.g., adder, control.
- Ports: use input and output keywords for clarity, e.g., input [7:0] data_in.

Comments:
- Include comments to explain the purpose and functionality of each wire, register, input, output, and module.
- Document assumptions, constraints, and any non-trivial logic.

Formatting:
- Follow a consistent indentation style for readability.
- Align similar items, such as port declarations, for clarity.

Constants:
- Use all uppercase letters for constants, e.g., ADDR_WIDTH = 16.

Generic Components

Components	Inputs	Outputs	Behavior	RTL Symbols
Register	reg_in[15:0] CLK[0:0] reset[0:0]	reg_out[15:0]	On each rising CLK edge, reads input signal and outputs value. If reset is high, wipes data and outputs 0	PC, Reggie, MDR, IR, ALUout
2-Bit Register	CLK[0:0] Reg2_in[1:0] Reset[0:0] SRw[0:0]	reg2_out[1:0]	On each rising CLK edge, reads input signal and outputs value. If reset is high, wipes data and outputs 0x00	CompCode
Status Register	CLK[0:0]	Reg2_out[1:0]	On each rising CLK	SR

	Reg2_in[0:0] isZero[0:0] Reset[0:0] SRw[0:0]		edge, reads input signal and isZero and outputs a value based on the specific comp codes layed out . If reset is high, wipes data and outputs 0x00	
ALU	aluop[1:0] aluA_in[15:0] aluB_in[15:0]	alu_out[15:0] isZero[0:0]	Reads the ALUOp[1:0] to determine what operation to perform and then outputs result	+, -, op
Immediate Generator	Imm_in[11:0]	imm_out[15:0]	SE a 12 bit number to 16	Imm
Memory	memW[0:0] memR[0:0] dataW_in[15:0] addr_in[15:0]	mem_out[15:0]	On each rising CLK edge, if memwrite is high, then it reads input from datawrite and outputs its value.	Mem
Control	opc[3:0] compc[15:14]	IorD[0:0] aluop[1:0] regw[0:0] srw [0:0] compcodew [0:0] memw[0:0] aluSrcA[2:0] aluSrcB[2:0]	Takes in the opc[3:0] and compc[3:0] to determine the output needed for the control signals.	Ctrl

MAIN ALU OPERATION TABLE

ALUop	Operation	Inputs	Outputs
11	Sub	Imm_out[15:0] Mem_out[15:0] reggie_data[15:0] mdr_data[15:0]	alu_out[15:0]
10	Add	Imm_out[15:0]	alu_out[15:0]

		Mem_out[15:0] reggie_data[15:0] Mdr_data[15:0]	
01	Shift Lower Logical Immediate	Imm_out[15:0] Mem_out[15:0] reggie_data[15:0] Mdr_data[15:0]	alu_out[15:0]

Control Signals

regop[0:0]	Mux selector bit for whether Reggie's input comes from MDR or mem_out
regw[0:0]	High: overwrites value in Reggie; Low: maintains the same value in register
srw[0:0]	High: overwrites value in SR; Low: maintains the same value in register
memw[0:0]	High: overwrites value in Memory; Low: maintains the same value in register
compcodew[0:0]	High: overwrites value in CompCode, Low: maintains the same value in register
IorD[0:0]	Mux selector bit for whether the value from PC is data or an instruction
ALUsrcA[1:0]	Mux selector for ALU input A: 00 = MDR_out 01= imm_out 11 = compcode_out
ALUsrcB[1:0]	Mux selector for ALU input B 00 = SR_out 01= Reg_out 11 = 4
ALUop[1:0]	Signals what kind of operation the ALU will perform Table with values is above

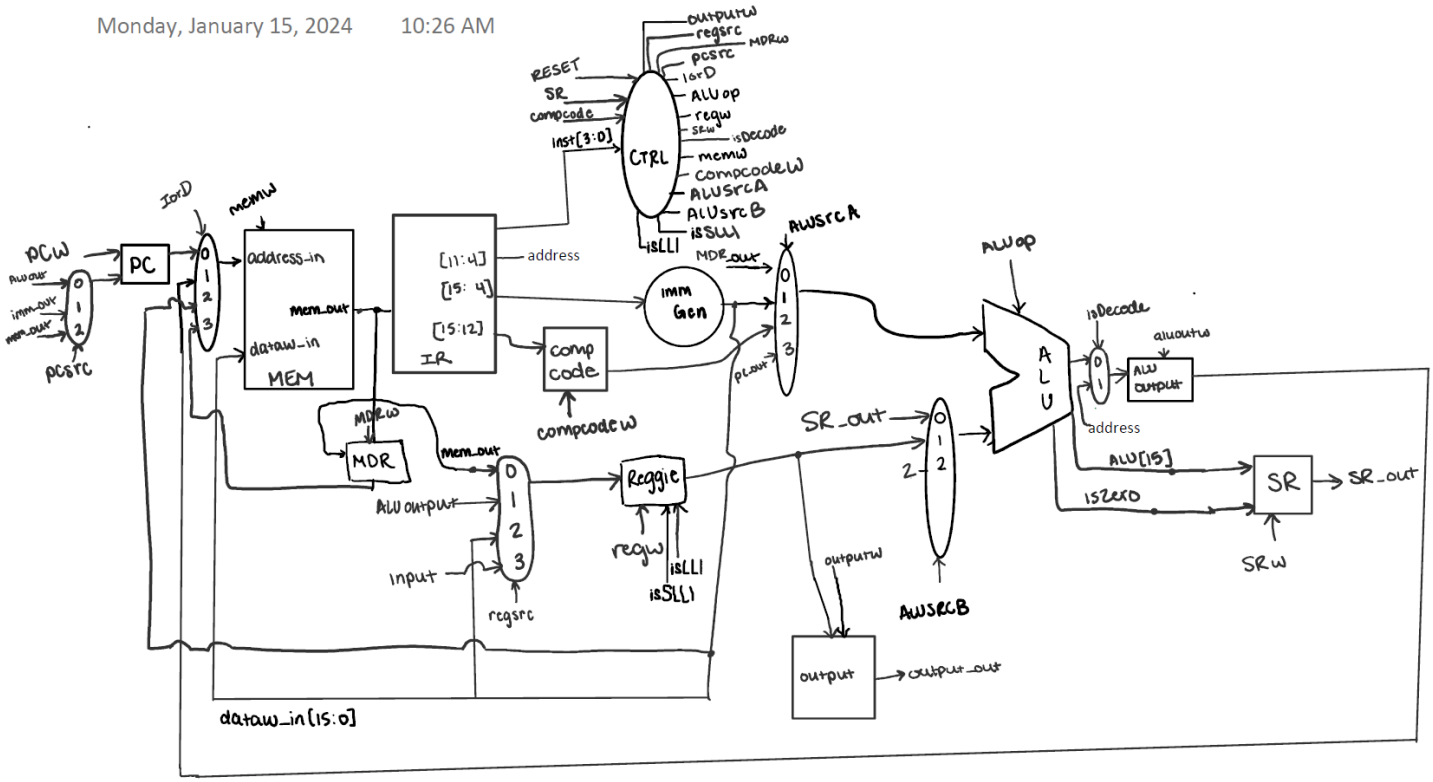
MUX ALUsrcA	Meaning
0	ALU input is from memory
1	ALU input is from immediate generator
2	ALU input is from CompCode

MUX ALUsrcB	Meaning
0	ALU input is from status register
1	ALU input is from Reggie
2	ALU input is 4

SR Value	Meaning
0	ALUout[15] = 0 && isZero == 0
1	ALUout[15] = 1 && isZero == 0
2	ALUout[15] = 0 && isZero == 1

Multicycle Datapath

Monday, January 15, 2024 10:26 AM



Unit Testing Plan

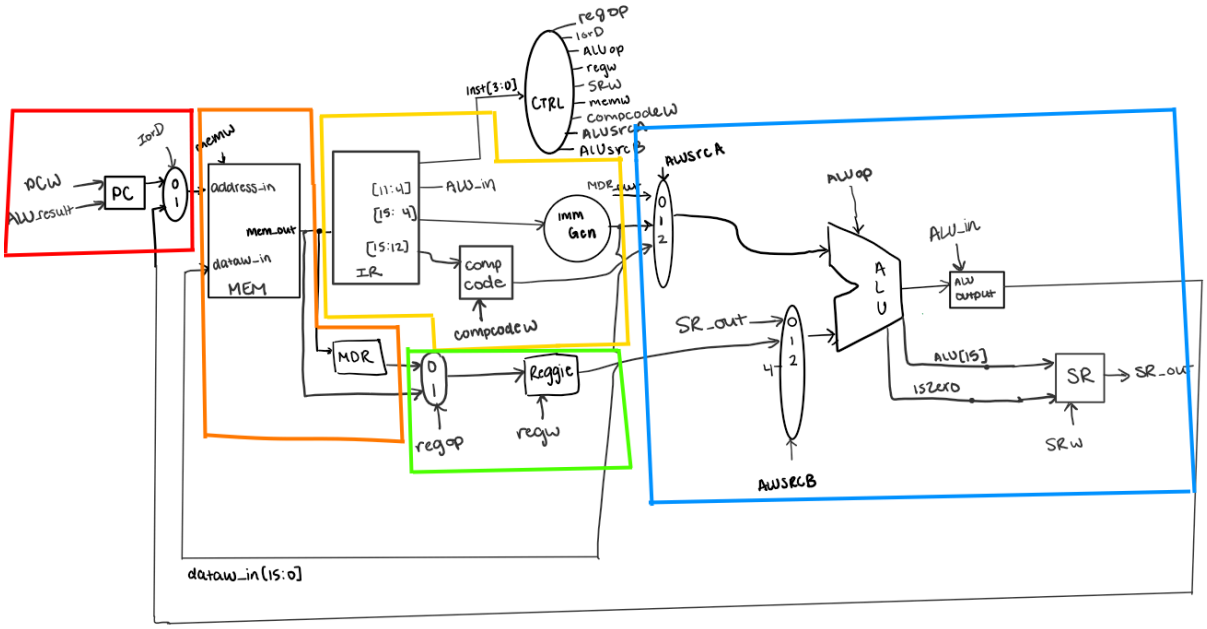
Component	Test Type	Input Signals	Test Description	Expected Output
Register	Reset	Reg_in[15:0] = 0x0002 Reset[0:0] = 1	Set register to 0x002 for a clock cycle, then turn the reset signal high (1) and verify output at the next cycle is 0x0000.	Out[15:0] = 0x0000
	Write/Read	regW =1/ 0 1.Reg_in[15:0] = 0x0002 2.Reg_in[15:0] = 0x0020 3.Reg_in[15:0] = 0x0200 4. Reg_in[15:0] = 0x2000	Set Register to 0x0002 And verify write on next 2 clock cycles with write signal =0 Repeat process for remaining Reg_in signals	1.Reg_in[15:0] = 0x0002 2.Reg_in[15:0] = 0x0020 3.Reg_in[15:0] = 0x0200 4. Reg_in[15:0] = 0x2000
2-bit register	Less-than	Reg2_in[0:0] =1 isZero[0:0] =0	Verify output	Out[1:0] =0b 01
	Greater than	Reg2_in[0:0] =0 isZero[0:0] =0	Verify output	Out[1:0] = 0b11
	equals	isZero[0:0] =1 1. Reg2_in[:0]= 0 2. Reg2_in[:0]= 1	Verify output for each signal.	Out[1:0] = 0b10
	reset	Reg2_in[0:0] = b0 isZero[0:0] = 0 Reset[0:0] = 1	Set register to 0b01 for a clock cycle, then turn the reset signal high (1) and verify output at the next cycle is 0b00.	Out[1:0] = 0b00

Memory	Read mem *(using the data in memory from lab7)*	memw[0:0]= 0 Memr[0:0] =1 Dataw_in[15:0] = 0x1111 1. address_in[15:0]= 0x0000 2. address_in[15:0]= 0x0002 3. address_in[15:0]= 0x0003	Read data at each address and verify output data is correct. Repeat for all addresses given	1. Mem_out[15:0] = 0x1234 2. Mem_out[15:0] = 0xDEAD 3. Mem_out[15:0] = 0xBEEF
Write is dependent on reading working	Write	Memw[0:0] = 1/0 Memr[0:0] 0/1 1. Dataw_in[15:0]: 0x3456 Address_in[15:0]: 0x0000 2.Dataw_in[15:0]: 0x1111 Address_in[15:0]: 0x0002	1st clock signal memw =1 (memr =0) Write data to the given address 2nd cycle Memr =1 (memw =0) data and verify data Repeat this test for next word and address	Mem_out[15:0]: 0x3456
ALU	addition	aluop[1:0]= b10 1. aluA_in[15:0]= 0x56ce aluB_in[15:0]= 0x2710 2. aluA_in[15:0]= 0x8888 aluB_in[15:0]= 0x7714	Add the two values of A and B together and verify result. Also verify isZero for each test. Ensure sign extension of output.	1. alu_out[1:0]=0x7dde isZero[0:0] =0 2. alu_out[15:0]=0x0000 isZero[0:0] =1
	subtraction	aluop[1:0]= b11 1. aluA_in[15:0]= 0x08ae aluB_in[15:0]= 0x0457 2. aluA_in[15:0]= 0x5555 aluB_in[15:0]= 0x5555	Subtract B from A and verify result. Also verify isZero for each test. Ensure sign extension of output	1. alu_out[1:0]=0x0457 isZero[0:0] =0 2. alu_out[15:0]=0x0000 isZero[0:0] =1 3. alu_out[15:0]=0x08ae

		3. aluA_in[15:0]= 0x0457 aluB_in[15:0]= 0xfba9		isZero[0:0] =0
	or	aluop[1:0]= b01 1. aluA_in[15:0]= 0x56ce aluB_in[15:0]= 0x2710 2.aluA_in[15:0]= 0x8888 aluB_in[15:0]= 0x7714	Bitwise OR A and B and compare results with the data provided. Ensure sign extension of output.	1. alu_out[15:0]=0xF600 isZero[0:0] =0 2. alu_out[15:0]=0xff9c isZero[0:0] =0
Complete Datapath	PC Register and IorD MUX	Test PC Write: PCw = 1 ALUresult = 0x1234 IorD = x Test IorD ALUout: PC = 0x0000 ALUoutput = 0x1234 IorD = 1 Test IorD PC: PC = 0x1234 ALUoutput = 0x0000 IorD = 0	Test PC Write: Tests writing to the program counter. Test IorD ALUout: Set the PC and ALUout to different values and select with the MUX. Test IorD PC: Set the PC and ALUout to different values and select with the MUX	Test PC Write: PC = 0x1234 Test IorD ALUout: MEM address_in = 0x1234 Test IorD PC: Mem address_in = 0x1234
	Memory and MDR Register	Test MDR: MDRw = 1 Mem_out = 0x1234 Test Read: memw = 0 Addr_in = 0x1234 Mem[0x1234] = 0x1234 Test Write memw = 1 Addr_in = 0x1234 Dataw_in = 0x1234	Test MDR: Tests that MDR responds to mem_out. Test Read: Tests if memory reads proper value at address. Test Write: Ensures memory value is properly set.	Test MDR: MDR = 0x1234 Test Read: Mem_out = 0x1234 Test Write: Mem[0x1234] = 0x1234

	IR and CompCode Registers and Immediate Generator	Mem_out = 0x1234 Compcodew = 1	Ensures correct bits are being taken in IR and comp code and imm gen are connected appropriately.	Imm Gen = 0x0234
	RegOp MUX and Reggie	Regop = 1 Regw = 1 Mem_out = 0x1234	Make sure the MUX selects the correct bit and writes to Reggie.	Reggie = 0x1234
	aluSrcA and B MUXes, ALU, and ALU_output and SR Registers	ALU source A = 0x1200 ALU source B = 0x0034 ALU_in = 1 SR_w = 1	Make sure the ALU outputs correct signals to all registers and their values update.	ALU Output = SR = 0x1234

Integration Plan



We decided to break our datapath into five distinct integration groups for testing.

- ➔ **PC register and IorD mux**
- ➔ **Memory and MDR register**
- ➔ **IR register, compcode register and Immediate generator**
- ➔ **Regop mux and Reggie**
- ➔ **alusrcA mux, alusrcB mux, ALU and ALU_output register, SR register**

We plan to implement each subsection on the diagram above. Then when those subsections are implemented and tested, we will go in order linking each subjection together. So the PC subsection will then be integrated with memory and then the Immediate Generator subsection will be linked with the other two sections and tested. We will incrementally implement our processor, adding a new subsystem each cycle.

PC register and IorD mux

Verify that the PC updates correctly each cycle, also ensure that if pcW is 0, no update occurs, and verify IorD mux output for the given control signal.

Memory and MDR register

Memory is a key part of our datapath, and therefore should be rigorously tested early on in implementation. In the Memory unit tests we have verified reading, writing correct and the correct address. Verify that MDR updates synchronously, based on the write enable input.

IR register, compcode register and Immediate generator

IR and CompCode tests include checking that the data updates synchronously, and data is only written when the register's write enable input is high.

For the immediate generator, verify sign extension for positive and negative numbers, while maintaining data at lower bit positions. Based on feedback from a past team, the immediate generator may need extensive care to ensure proper functionality.

Regop mux and Reggie

Verify regOp mux outputs correct data for the given control signal and that Reggie updates synchronously, and data is only written when the REggie's write enable input is high.

alusrcA mux, alusrcB mux, ALU and ALU_output register

Verify alusrcA mux outputs correct data for the given control signal. Verify alusrcB mux outputs correct data for the given control signal. ALU-output updates synchronously, and data is only written when the ALU_output's write enable input is high.

For the ALU, there are 4 different possible inputs that need testing,

Compcode and SR:

This operation checks that comp and SR are equal for branching so the main operation is subtraction and verification that the output is zero. Verify for all possible combinations of comp codes and SR codes to ensure proper functionality.

Reggie and imm:

Verify that for the 3 ALU operations(+,-, OR) that the ALU outputs the correct data, keeping in mind both Reggie and MDR are 2's complement numbers. Make sure the ALU's isZero output signal is correct in all cases as it is relied upon for branching and comparisons.

Reggie and MDR:

Verify that for the 3 ALU operations(+,-, OR) that the ALU outputs the correct data, keeping in mind both Reggie and MDR are 2's complement numbers. Make sure the ALU's isZero output signal is correct in all cases as it is relied upon for branching and comparisons.

MDR and 4:

Verify that ALU adds 4 to MDR for various MDR inputs. Make sure the ALU's isZero output signal is correct in all cases as it is relied upon for branching and comparisons.

SR register should have been rigorously tested in the component testing phase, As verification of other operations the output in SR should be checked when other operations run to ensure the correct data is written as other operations occur.

Control

Inputs

- **Opcode**: 4-bit input representing the opcode for the instruction being executed from IR

Outputs

IorD	Indicates whether the instruction is a memory access instruction (lw or sw).
regop	Specifies the input into reggie
regw	Indicates whether the instruction involves writing to registers.
memw	Indicates whether the instruction involves writing to memory
aluSrcA, aluSrcB	Specify the ALU sources for operands
srw	Indicates whether the instruction

	modifies the status register
pcw	Indicates whether the instruction involves updating the program counter
compcodew	Indicates whether the instruction involves writing to comp code register
aluop	Specifies the ALU operation to be performed

Expected Behavior:

Given Testbench:

Opcode: 0000, IorD: 1, memw: 0, regw: 1, pcw: 0, srw: 0, aluop: 00, regop: 00, aluSrcA: 00, aluSrcB: 00, compcodew: 0

Opcode: 0001, IorD: 1, memw: 1, regw: 0, pcw: 0, srw: 0, aluop: 00, regop: 00, aluSrcA: 00, aluSrcB: 00, compcodew: 0

Opcode: 0100, IorD: 0, memw: 0, regw: 1, pcw: 0, srw: 0, aluop: 10, regop: 00, aluSrcA: 00, aluSrcB: 00, compcodew: 0

Opcode: 0101, IorD: 0, memw: 0, regw: 1, pcw: 0, srw: 0, aluop: 11, regop: 00, aluSrcA: 00, aluSrcB: 00, compcodew: 0

The control unit interprets the opcode input to determine the control signals to be asserted or deasserted. It sets appropriate control signals based on the opcode to configure the ALU operation, memory access, register write, program counter update, etc. For undefined opcodes, it sets default values for all control signals to 0.

ALU Test Descriptions

Addition Test:

- Description: Verify that the ALU correctly performs addition operations.

- Inputs: Two positive numbers as inputs (aluA_in, aluB_in) and the ALU operation (aluop) set to perform addition.
- Expected Output: The alu_out should contain the sum of the two input numbers, and the isZero flag should be set accordingly.

Subtraction Test:

- Description: Verify that the ALU correctly performs subtraction operations.
- Inputs: Two numbers (aluA_in, aluB_in) with aluA_in greater than or equal to aluB_in, and the ALU operation (aluop) set to perform subtraction.
- Expected Output: The alu_out should contain the result of subtracting aluB_in from aluA_in, and the isZero flag should be set accordingly.

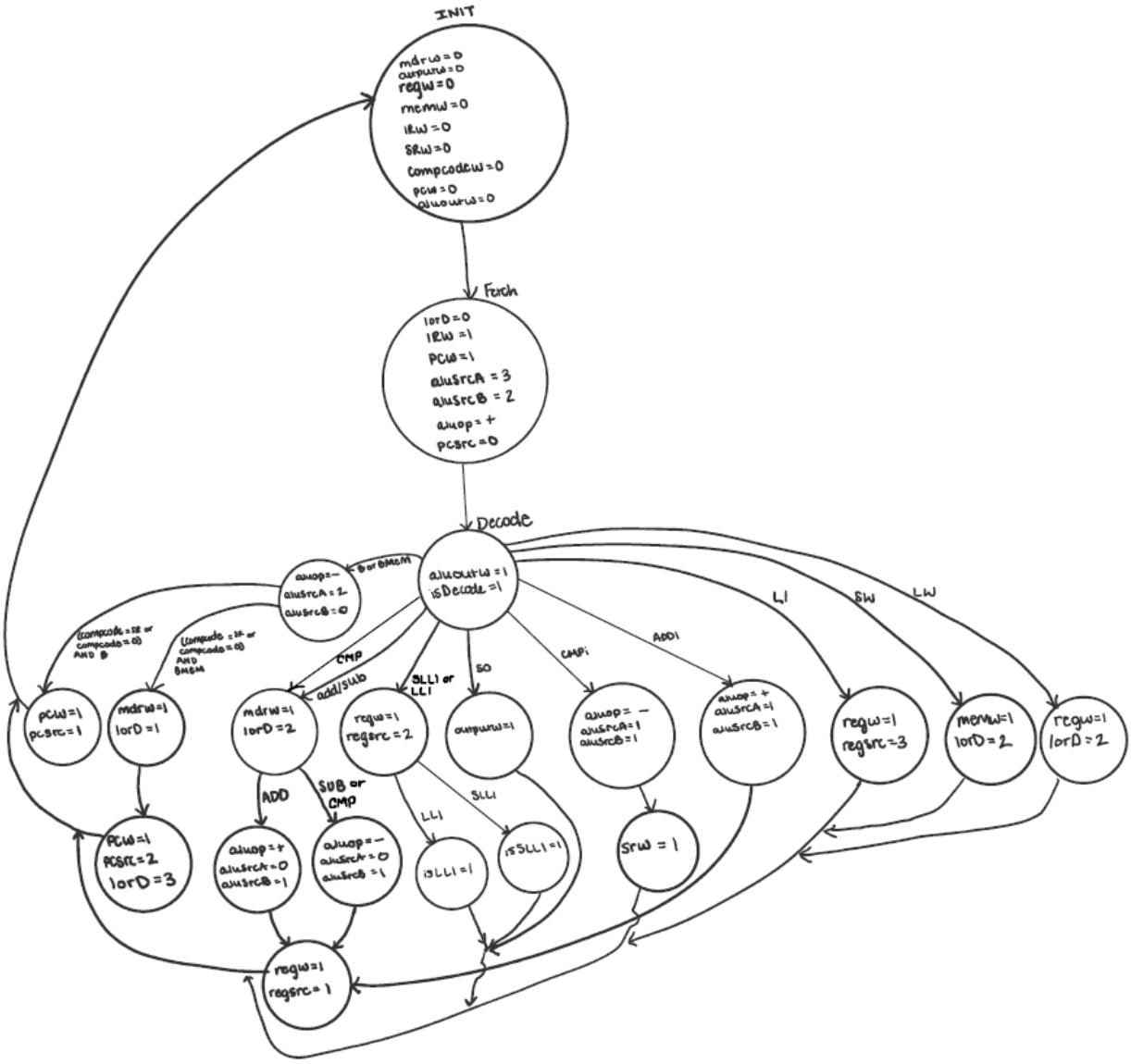
Left Shift Test:

- Description: Verify that the ALU correctly performs left shift operations.
- Inputs: A number (aluB_in) and a shift amount (aluA_in), and the ALU operation (aluop) set to perform left shift.
- Expected Output: The alu_out should contain the result of left shifting aluB_in by the number of bits specified in aluA_in, and the isZero flag should be set accordingly if the result is zero.

System Testing

After running testbenches for each individual component, the datapath needs to be tested. The initial basic test for the datapath is to test each of the 15 instructions. If those all function as expected (pass), then the next step is to run the program

Finite State Machine



Changes Made to Assembly and Machine Language

Jan 14 We deleted the addressing mode for M-type, and we added registers for sp and ra. We realized we were limited to only 64 bit jumps so we changed the status register to only be signaling 2 bits in order to reduce the size of comp_code for branching. This gives us more bits for immediate/direct addressing.

Jan 15 We decided to make our datapath multicycle. This means we need another register MDR to save steps.

Jan 16 We got rid of LUI to make more space for SP, and then we removed SRI to replace it with an “or” instruction.

Jan 25 Multicycle corrected according to original intent. No change in purpose or design choices

Jan 26 Added a MUX option to allow for the immediate to be input into Reggie. This increased regsrc bits to 2

Feb 1 PC needed a mux and 2 bit src since its input could come from either ALU, Memory, or Immediate.

IorD needed 2 more inputs for MDR and Immediate. Ideally, this means “IorD” should be renamed to just “MemSrc” but renaming this would cause other issues. This also requires IorD to be a 2 bit input instead of 1. This makes no powerful difference since we are reinstantiating the same mux component for each of the muxes.

Since we’re using the same ALU to increment PC, AluSrcA needs PC as a potential input.

Feb 3 Added 2 1-bit control bits for SLLI and LLI to Reggie so it doesn’t need to wait on ALU for an operation that can be performed inside the register. In order to retrieve the address from IR and input it into the ALUOutput (to speed up our cycles for branching), we need a mux to ALUoutput for a second possible input.

Feb 13 LMEM/BMEM does not have enough bits for comparison code and a return address. We realized this while converting BMEM 0b10 RA to machine code. To address this (pun intended), we decided to limit the branch range to be PC relative using only 2 bytes for address immediate. This minimizes the changes to the rest of our design (no changes made to datapath, code, or instruction format).

Milestone 5 Meeting Notes