Datapath Design

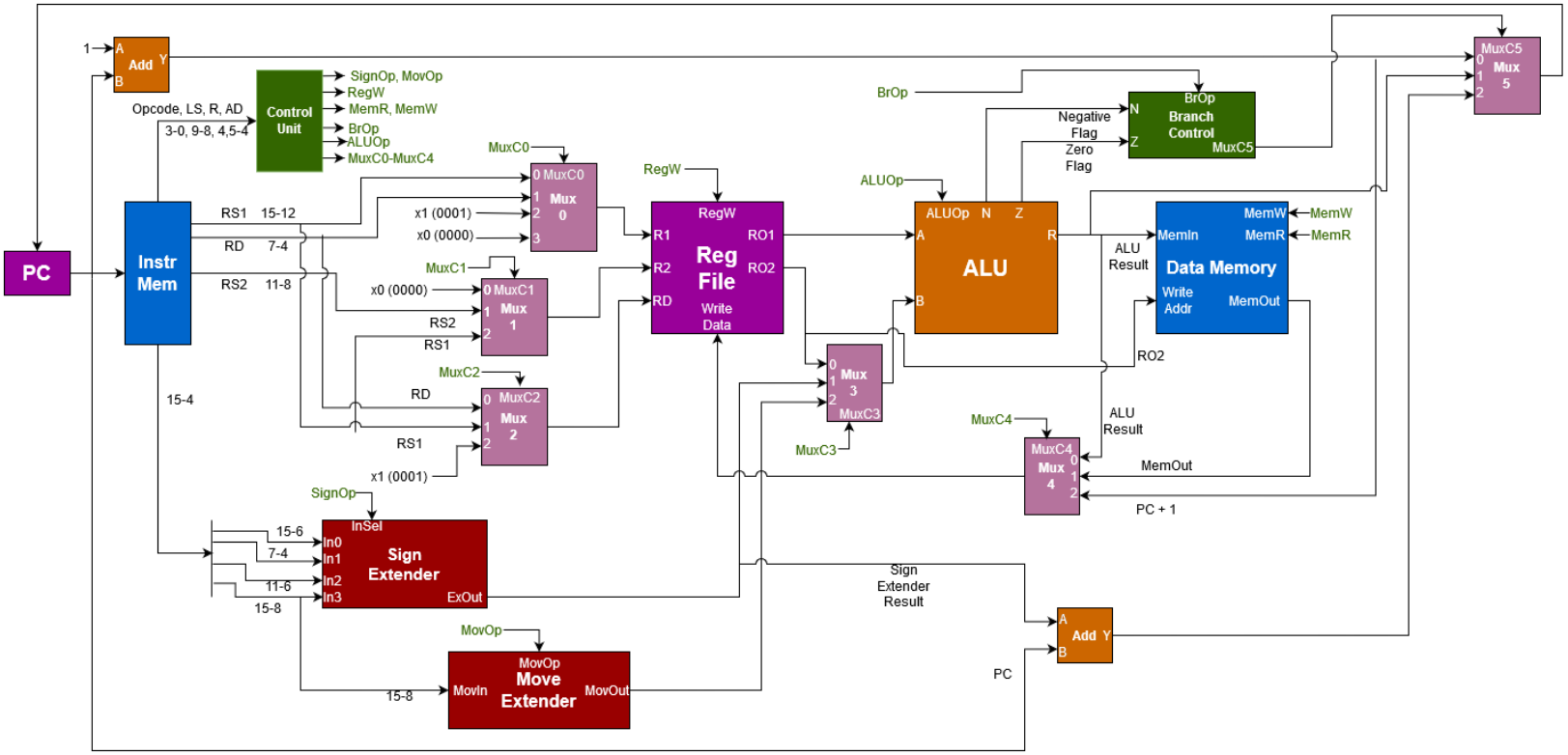
1 Design Requirements

A datapath must be designed to realize the RISC-Z ISA. The following components are included in this report:

- A register-level block diagram of the datapath, with all components and control signals labeled.
- A description of the inputs, outputs, and function of each component in the datapath.
- For each instruction of the RISC-Z ISA, the register transfers, or sequence of register transfers, required to fetch and execute the instruction are listed.
- A truth table listing all control signals and the values of each control signal required for executing each instruction.
- A discussion of the tradeoffs and other design decisions made in developing the datapath.

2 Block Diagram

Below is a block diagram detailing the components and flow of the datapath.



3 Datapath Components

In the sections below, each component of the datapath is described, including its inputs and outputs.

3.1 Program Counter

The program counter is a 16-bit register that holds the address of the next instruction. The program counter will be updated based on the previous instruction. The PC will feed directly into the instruction memory to select the location of the next instruction.

3.2 Instruction Memory

The instruction memory holds all the instructions for the program. The 16-bit address of the program counter will be inputted into the instruction memory, which will output the 16-bit machine code value of the current instruction.

Inputs: PC Address (16-bits)
Outputs: Instruction (16-bits)

3.3 Register File

The register file is a multi-port memory array that contains the values of all the registers. The register file accepts three 4-bit inputs, which denote the two registers to be read (R1 and R2) and the register to be written to (RD). It also has a 1-bit write enable (RegW) that controls when a register is written to. The register file also has a 16-bit write data input (WriteData), which contains the value to be written to a register if needed. The register then outputs two 16-bit values (RO1 and RO2), which are the contents of the two registers selected to read.

Inputs: R1 (4-bits), R2 (4-bits), RD (4-bits), RegW (1-bit), WriteData (16-bits)
Outputs: RO1 (16-bits), RO2 (16-bits)

3.4 ALU

The arithmetic logic unit (ALU) can perform a variety of arithmetic operations. The ALU accepts two 16-bit inputs (A and B), which are the values used in the arithmetic operation. A 4-bit input control signal (ALUOp) indicates what operation is to be performed. The ALU then outputs the 16-bit result (R). It also has three 1-bit output flags: Negative (N), Overflow (O), and Zero (Z).

Inputs: A (16-bits), B (16-bits), ALUOp (4-bits)
Outputs: R (16-bits), O (1-bit), N (1-bit), Z (1-bit)

The ALU operations can be seen in the table below.

ALUOp	**Operation**
0000	Nothing
0001	Add
0010	AND
0011	NAND
0100	OR
0101	Subtract
0110	Logical Left Shift
0111	Arithmetic Left Shift
1000	Logical Right Shift
1001	Arithmetic Right Shift

3.5 Data Memory

The data memory holds any variables or values stored for program usage. The data memory accepts a 16-bit input (MemIn), a memory write enable 1-bit input (MemW), and a 16-bit input write address (WriteAddr), which contains the address to write to. It also has a 16-bit output (MemOut), which is the contents read from the MemIn address. If MemW is high, then MemIn will be written to the address MemAddr.

Inputs: MemIn (16-bits), WriteAddr (16-bits), MemW (1-bit)
Outputs: MemOut (16-bits)

3.6 Sign Extender

The sign extender unit extends the sign bit of the input value. The sign extender can accept inputs of varying lengths: In0 (10-bits), In1 (4-bits), In2 (6-bits), and In3 (8-bits). It also has a 2-bit input select signal (InSel), which determine which input should be extended. The extended value is then outputted as a 16-bit value (ExOut).

Inputs: In0 (10-bits), In1 (4-bits), In2 (6-bits), In3 (8-bits), InSel (2-bits)
Outputs: ExOut (16-bits)

3.7 Move Extender

The move extender is used for the move lower (movl) and mov upper (movu) instructions. It accepts an 8-bit input (MovIn) and a 1-bit control signal (MovOp). The move extender will place 8 0's to either the bottom or top of the 8-bit input depending on MovOp. The 16-bit value will then be outputted (MovOut).

Inputs: MovIn (8-bits), MovOp (1-bit)
Outputs: MovOut (16-bits)

3.8 Adders

There are two adders used in addition to the ALU. Add1 adds a hard coded 1 to the PC. Add2 adds the PC to a branch or jump offset value. Both adders accept two 16-bits inputs (A and B) and output a 16-bit value (Y).

Inputs: A (16-bits), B (16-bits)
Outputs: Y (16-bits)

3.9 Control Unit

The control unit receives the opcode, LS, R, and AD bits from the instruction memory and sets control signals for the other datapath components in order to execute the proper instruction. The opcode identifies the instruction, LS is used to differentiate load and store, R differentiates jump and return, and AD indicates the direction and

arithmetic of shifts. The control unit then outputs: register write enable (RegW), memory read enable (MemR), memory write enable (MemW), branch logic control (BrOp), ALU logic control (ALUOp), move extender control (MovOp), sign extender control (SignOp), and multiplexer control signals (MuxC0 - MuxC4). The full truth table can be seen in Section 5.

Inputs: opcode (4-bits), LS (2-bits), R (1-bit), AD (2-bits)
Outputs: RegW (1-bit), MemR (1-bit), MemW (1-bit), BrOp (3-bits), ALUOp (4-bits), MovOp (1-bit), SignOp (2-bits), MuxC0 (2-bits), MuxC0 (2-bits), MuxC1 (2-bits), MuxC2 (2-bits), MuxC3 (2-bits), MuxC4 (2-bits)

3.10 Branch Control Unit

The branch control unit controls whether a branch or jump is taken. The branch control unit receives a branch logic control (BrOp) signal from the main control unit, which identifies what type of branch or jump is being evaluated. It also accepts the negative (N) and Zero (Z) flags from the ALU to determine whether or not a branch should be taken. The branch control unit then outputs the mux 5 control signal (MuxC5), which determines what value is loaded into the PC. The full truth table can be seen in Section 5.

Inputs: BrOp (3-bits), N (1-bit), Z (1-bit)
Outputs: MuxC5 (2-bits)

3.11 Various Multiplexers

Various multiplexers are used throughout the datapath to select one output from multiple inputs. The multiplexers used throughout the datapath are listed below. Each multiplexer has a control signal (MuxC#) from the control unit. The functions of each multiplexer are described below.

Mux0 (4-to-1): Controls input for register file R1

Inputs: RS1 (4-bits), RD (4-bits), x0 (4-bits), x1 (4-bits), MuxC0 (2-bits)
Outputs: M0Out (4-bits)

Mux1 (4-to-1): Controls input for register file R2

Inputs: x0 (4-bits), RS2 (4-bits), RS1 (4-bits), MuxC1 (2-bits)
Outputs: M1Out (4-bits)

Mux2 (4-to-1): Controls input for register file RD

Inputs: RD (4-bits), RS1 (4-bits), x1 (4-bits), MuxC2 (2-bits)
Outputs: M2Out (4-bits)

Mux3 (4-to-1): Controls input for ALU B

Inputs: R (16-bits), ExOut (16-bits), MovOut (16-bits), MuxC3 (2-bits)
Outputs: M3Out (16-bits)

Mux4 (4-to-1): Controls input for register file WriteData

Inputs: ALU Result (16-bits), PC+1 (16-bits), MemOut (16-bits), MuxC4 (2-bits)
Outputs: M4Out (16-bits)

Mux5 (4-to-1): Controls input for Program Counter

Inputs: PC+1 (16-bits), ALU Result (16-bits), PC+Label (16-bits), MuxC5 (2-bits)
Outputs: M5Out (16-bits)

4 Instruction Datapath Usage

The sections below describe the components and registers in the datapath used by each instruction.

4.1 The Add Register Instruction: `addr`

- Fetch: Read current instruction value from Instruction Memory

- Register Read: Read RS1 and RS2 from the Register File
- ALU Operation: RS1 + RS2
- Register Write: Write the ALUResult to RD
- Increment PC: $PC \leftarrow PC + 1$

4.2 The And Register Instruction: `andr`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RS1 and RS2 from the Register File
- ALU Operation: RS1 & RS2
- Register Write: Write the ALUResult to RD
- Increment PC: $PC \leftarrow PC + 1$

4.3 The Nand Register Instruction: `nandr`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RS1 and RS2 from the Register File
- ALU Operation: RS1 $\sim$ & RS2
- Register Write: Write the ALUResult to RD
- Increment PC: $PC \leftarrow PC + 1$

4.4 The Or Register Instruction: `orr`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RS1 and RS2 from the Register File
- ALU Operation: RS1 | RS2
- Register Write: Write the ALUResult to RD
- Increment PC: $PC \leftarrow PC + 1$

4.5 The Subtract Register Instruction: `subr`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RS1 and RS2 from the Register File
- ALU Operation: RS1 - RS2
- Register Write: Write the ALUResult to RD
- Increment PC: $PC \leftarrow PC + 1$

4.6 The Add Immediate Instruction: `addi`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RD from the Register File
- Sign Extension: Extend sign of immediate value
- ALU Operation: RD + sign extended immediate value
- Register Write: Write the ALUResult to RD
- Increment PC: $PC \leftarrow PC + 1$

4.7 The Move Upper Instruction: `movu`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RD from the Register File
- Move Extension: Add 8 0's to lower half of immediate value
- ALU Operation: RD | move extended immediate value
- Register Write: Write the ALUResult to RD
- Increment PC: $PC \leftarrow PC + 1$

4.8 The Move Lower Instruction: `movl`

- Fetch: Read current instruction value from Instruction Memory
- Move Extension: Add 8 0's to upper half of immediate value
- ALU Operation: RD | move extended immediate value
- Register Write: Write the ALUResult to RD
- Increment PC: $PC \leftarrow PC + 1$

4.9 The Jump Instruction: `jmp`

- Fetch: Read current instruction value from Instruction Memory
- Register Write: Write PC+1) to x1
- Sign Extension: Extend sign of immediate value
- Adder: Add PC to sign extended immediate value
- Adjust PC: $PC \leftarrow PC + immediate\ value$

4.10 The Return Instruction: `ret`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read x1 and x0 from the Register File
- ALU Operation: x1 + x0
- Adjust PC: $PC \leftarrow ALUResult\ (x1)$

4.11 The Branch Equal Instruction: `beq`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RS1 and RS2 from the Register File
- ALU Operation: RS1 - RS2
- Sign Extension: Extend sign of immediate value

- Adder: Add PC to sign extended immediate value
- Branch Logic: Determine whether or not to take branch
- Adjust PC: $PC \leftarrow PC + 1$ or $PC \leftarrow PC + immediate\ value$

4.12 The Branch Not Equal Instruction: `bne`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RS1 and RS2 from the Register File
- ALU Operation: RS1 - RS2
- Sign Extension: Extend sign of immediate value
- Adder: Add PC to sign extended immediate value
- Branch Logic: Determine whether or not to take branch
- Adjust PC: $PC \leftarrow PC + 1$ or $PC \leftarrow PC + immediate\ value$

4.13 The Branch Greater Than Instruction: `bgt`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RS1 and RS2 from the Register File
- ALU Operation: RS1 - RS2
- Sign Extension: Extend sign of immediate value
- Adder: Add PC to sign extended immediate value
- Branch Logic: Determine whether or not to take branch
- Adjust PC: $PC \leftarrow PC + 1$ or $PC \leftarrow PC + immediate\ value$

4.14 The Branch Less Than Instruction: `blt`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RS1 and RS2 from the Register File
- ALU Operation: RS1 - RS2
- Sign Extension: Extend sign of immediate value
- Adder: Add PC to sign extended immediate value
- Branch Logic: Determine whether or not to take branch
- Adjust PC: $PC \leftarrow PC + 1$ or $PC \leftarrow PC + immediate\ value$

4.15 The Load Register Instruction: `ld`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RS1 and x0 from the Register File
- ALU Operation: RS1 + x0
- Data Memory Read: Read data at the address of ALUResult (RS1)
- Register Write: Write the memory read value to RD
- Adjust PC: $PC \leftarrow PC + 1$

4.16 The Store Register Instruction: `str`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RS1 and x0 from the Register File
- ALU Operation: RS1 + x0
- Data Memory Write: Write data from ALUResult (RS1) to address in RD
- Adjust PC: $PC \leftarrow PC + 1$

4.17 The Shift Logical Left Instruction: `shl`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RS1 from the Register File
- Sign Extension: Extend sign of immediate value
- ALU Operation: Logically left shift RS1 by the sign extended immediate value
- Register Write: Write the ALUResult to RS1
- Increment PC: $PC \leftarrow PC + 1$

4.18 The Shift Arithmetic Left Instruction: `sal`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RS1 from the Register File
- Sign Extension: Extend sign of immediate value
- ALU Operation: Arithmetically left shift RS1 by the sign extended immediate value
- Register Write: Write the ALUResult to RS1
- Increment PC: $PC \leftarrow PC + 1$

4.19 The Shift Logical Right Instruction: `shr`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RS1 from the Register File
- Sign Extension: Extend sign of immediate value
- ALU Operation: Logically right shift RS1 by the sign extended immediate value
- Register Write: Write the ALUResult to RS1
- Increment PC: $PC \leftarrow PC + 1$

4.20 The Shift Arithmetic Left Instruction: `sar`

- Fetch: Read current instruction value from Instruction Memory
- Register Read: Read RS1 from the Register File
- Sign Extension: Extend sign of immediate value
- ALU Operation: Arithmetically right shift RS1 by the sign extended immediate value
- Register Write: Write the ALUResult to RS1
- Increment PC: $PC \leftarrow PC + 1$

4.21 The Halt Instruction: `halt`

- Fetch: Read current instruction value from Instruction Memory
- Halt: Stop execution of the program

5 Control Logic

The truth tables for the logic components of the datapath are shown below.
Note: "X" represents a don't care.

5.1 Main Control Truth Table

Inputs				Outputs						
opcode	**R**	**LS**	**AD**	**RegW**	**MemR**	**MemW**	**BrOp**	**ALUOp**	**MovOp**	**SignOp**
0001	X	X	X	1	0	0	000	0001	X	X
0010	X	X	X	1	0	0	000	0010	X	X
0011	X	X	X	1	0	0	000	0011	X	X
0100	X	X	X	1	0	0	000	0100	X	X
0101	X	X	X	1	0	0	000	0101	X	X
0110	X	X	X	1	0	0	000	0001	X	11
0111	X	X	X	1	0	0	000	0100	0	X
1000	X	X	X	1	0	0	000	0100	1	X
1001	0	X	X	1	0	0	001	0000	X	00
1001	1	X	X	0	0	0	110	0001	X	X
1010	X	X	X	0	0	0	010	0101	X	01
1011	X	X	X	0	0	0	011	0101	X	01
1100	X	X	X	0	0	0	100	0101	X	01
1101	X	X	X	0	0	0	101	0101	X	01
1110	X	10	X	1	1	0	000	0001	X	X
1110	X	01	X	0	0	1	000	0001	X	X
1111	X	X	00	1	0	0	000	0110	X	10
1111	X	X	10	1	0	0	000	0111	X	10
1111	X	X	01	1	0	0	000	1000	X	10
1111	X	X	11	1	0	0	000	1001	X	10
0000	X	X	X	X	X	X	X	X	X	X

Inputs				Outputs				
opcode	**R**	**LS**	**AD**	**MuxC0**	**MuxC1**	**MuxC2**	**MuxC3**	**MuxC4**
0001	X	X	X	00	01	00	00	00
0010	X	X	X	00	01	00	00	00
0011	X	X	X	00	01	00	00	00
0100	X	X	X	00	01	00	00	00
0101	X	X	X	00	01	00	00	00
0110	X	X	X	01	X	00	01	00
0111	X	X	X	01	X	00	10	00
1000	X	X	X	01	X	00	10	00
1001	0	X	X	X	X	10	X	10
1001	1	X	X	10	00	X	00	X
1010	X	X	X	00	01	X	00	X
1011	X	X	X	00	01	X	00	X
1100	X	X	X	00	01	X	00	X
1101	X	X	X	00	01	X	00	X
1110	X	10	X	00	00	00	00	01
1110	X	01	X	01	00	X	00	X
1111	X	X	00	00	X	01	01	00
1111	X	X	10	00	X	01	01	00
1111	X	X	01	00	X	01	01	00
1111	X	X	11	00	X	01	01	00
0000	X	X	X	X	X	X	X	X

5.2 Branch Control Truth Table

Inputs			Outputs
BranchOp	**N**	**Z**	**MuxC5**
000	X	X	00
001	X	X	10
110	X	X	01
010	X	1	10
010	X	0	00
011	X	0	10
011	X	1	00
100	0	0	10
100	1	X	00
101	1	0	10
101	0	X	00

6 Design Considerations

6.1 Cost vs Speed Trade-offs

One speed vs cost consideration was the inclusion of two dedicated adders. These adders add a cost to the hardware components, but allow instructions to be completed in one cycle. Similarly, the inclusion of many multiplexers add to the hardware costs but provide a variety of option for possible inputs, which in turn allow an instruction to be completed in one clock cycle. The design decision of choosing a single cycle datapath over a multi-cycle data path also involved considering additional costs and execution speeds. These trade-offs are further discussed in the following section.

6.2 Single Cycle vs Multi Cycle vs Pipelined

A single cycle design was chosen for this datapath. The simplicity of the single cycle design merged well the simplistic design of the ISA. A single cycle design will also result in less power usage. The trade off of using a single cycle design would the the idle time of the datapath. The clock cycle must cater to the longest instruction time (critical path). However, in this simple design, the difference between the critical path and other instructions is not very large. A mutli-cycle design would have resulted in a lower clock period as compared to single cycle, but would also add more power consumption and greater complexity to the design. An ISA of this simplicity would most likely be implemented on a lower power board, which would benefit from the reduced power consumption of the single cycle design. A pipelined design would lower the average amount of clock cycles needed for each instruction, but would also add greater complexity to the design. A pipelined or multi-cycle design was not used due to the time constraints for the design and implementation of this datapath.

6.3 Shared and Dedicated Components

The ALU is one of the major shared components, as it is used in some form in almost all instructions. In addition to the ALU there are two dedicated adders: one for incrementing the PC by 1 and one to add branch/jump offsets. These dedicated adders where chosen as the ALU needs to be used in parallel with these add operations. For example, in branch instructions, the ALU is used to compare the two source registers, so it can not be used to calculate the branch address.

6.4 Edge-triggered vs. Latching Registers

Latching registers will be used in this design, as it is a single cycle datapath. The control signals will dictate when key values need to be loaded, read, written, etc. Edge-triggered registers seem to be better suited for multi-cycle designs, as you would use multiple clock signals.