SI100B 2026 Spring Project: Design and Implementation of a miniCPU

SI100B 2026 Spring EE Part TA Team

June 7, 2026

Contents

Contents **I**

1 Introduction **1**

2 ISA of miniCPU **2**
2.1 List of Abbreviations . . . 2
2.2 R-type Instructions (Register) . . . 2
2.3 I-type Instructions (Immediate) . . . 3
2.4 S-type Instructions (Store) . . . 4
2.5 B-type Instructions (Branch) . . . 5
2.6 J-type Instructions (Jump) . . . 6

3 Architecture of miniCPU **7**
3.1 Main Module Descriptions . . . 8
3.1.1 Register File . . . 8
3.1.2 Program Counter (PC) . . . 8
3.1.3 Arithmetic Logic Unit (ALU) . . . 9
3.1.4 Data Memory . . . 10
3.1.5 Immediate Extender . . . 10
3.1.6 Controller . . . 11

4 Requirements **15**
4.1 Academic Policy . . . 15
4.2 Grading Overview . . . 16
4.3 Logisim Processor (30 pts) . . . 16
4.3.1 Processor Rubric . . . 16
4.3.2 Testing Details . . . 17
4.4 Python Assembler (20 pts) . . . 17
4.4.1 Syntax Requirements . . . 17
4.4.2 Assembler Rubric . . . 18
4.5 Reports (25 pts) . . . 18
4.6 Check (25 pts) . . . 19
4.7 Bonus (up to 25 pts) . . . 19
4.7.1 Bonus Requirements and Warnings . . . 20
4.8 Submission . . . 21

1 Introduction

This project aims to design and implement a simplified central processing unit (miniCPU), covering both its datapath and controller. Through this project, students will gain a deep understanding of core computer architecture concepts, including the register file, arithmetic logic unit (ALU), program counter, instruction memory, and data memory, as well as how instruction decoding and control signal generation enable instruction execution.

In computer architecture, the terms "architecture" and "microarchitecture" refer to the instruction set architecture (ISA) and the processor implementation, respectively. The ISA specifies which instructions the processor must support, while the microarchitecture describes the hardware circuits that implement those instructions. Popular ISAs include x86, ARM, MIPS, and RISC-V. The same ISA can have multiple hardware implementations (e.g., both Intel and AMD produce x86 processors that run the same programs but have different internal circuits). This project focuses on the RISC-V ISA and processor, a rapidly growing open-source alternative to proprietary ISAs like x86 and ARM.

This is a collaborative project! Pair programming is encouraged, as it may be an effective way to complete the project.

2 ISA of miniCPU

2.1. List of Abbreviations

Table 2.1: List of Abbreviations

Abbreviation	Description
`rs1`	Register `rs1`
`rs2`	Register `rs2`
`rd`	Register `rd`
`pc`	Register `pc`
`imm`	Immediate
`SE()`	Sign Extend
`ZE()`	Zero Extend
`[]`	Memory Access
`lab`	Label
`f3`	Funct3
`f7`	Funct7
`opcode`	Opcode

In this project, you will implement a subset of the RISC-V RV32-I instruction set, consisting of 13 instructions in three categories: arithmetic, memory access, and control flow. These instructions cover basic computation, data transfer, and program control, sufficient for building simple programs.

2.2. R-type Instructions (Register)

R-type instructions operate on registers (operands: `rs1, rs2, rd`). The result of the operation on `rs1` and `rs2` is written to `rd`. The required R-type instructions are shown in table 2.2.

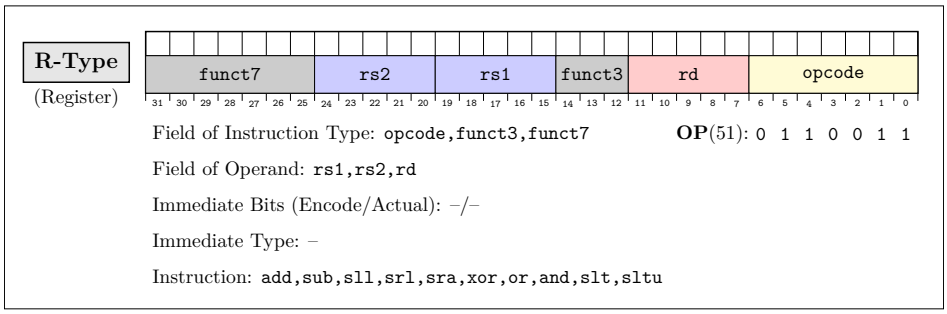


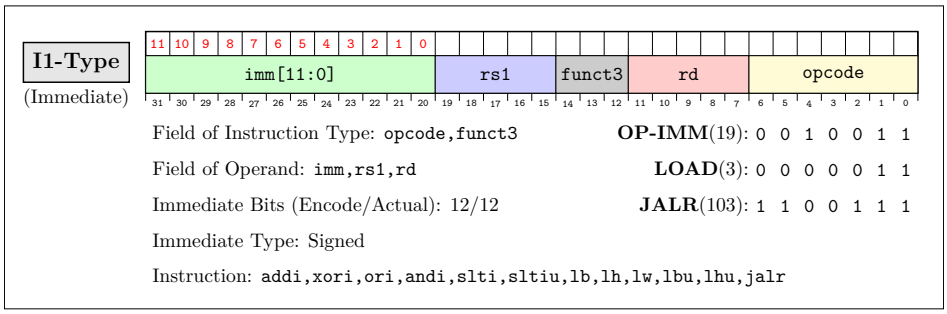
Figure 2.1: R-type Instruction Encoding

Table 2.2: R-type Instruction List

Instruction	Description	Type	opcode	f3	f7
add rd,rs1,rs2	rd=rs1+rs2	R	0110011	000	0000000
Add				Register addition	
sub rd,rs1,rs2	rd=rs1-rs2	R	0110011	000	0100000
Sub				Register subtraction	
and rd,rs1,rs2	rd=rs1&rs2	R	0110011	111	0000000
And				Register AND	
or rd,rs1,rs2	rd=rs1|rs2	R	0110011	110	0000000
Or				Register OR	
slt rd,rs1,rs2	rd=(rs1<rs2)	R	0110011	010	0000000
Set Less Than				Register set if less (signed)	

2.3. I-type Instructions (Immediate)

I-type instructions use an immediate value (operands: `imm, rs1, rd`). These include immediate arithmetic and load instructions. The required I-type instructions are shown in table 2.3.



(a) I1-type Instruction Encoding

Figure 2.2: I-type Instruction Encoding

Table 2.3: I-type Instruction List

Instruction	Description	Type	opcode	f3	f7
`addi rd,rs1,imm`	`rd=rs1+SE(imm)`	I	0010011	000	-------
Add Immediate					Immediate addition
`andi rd,rs1,imm`	`rd=rs1&SE(imm)`	I	0010011	111	-------
And Immediate					Immediate AND
`ori rd,rs1,imm`	`rd=rs1|SE(imm)`	I	0010011	110	-------
Or Immediate					Immediate OR
`slti rd,rs1,imm`	`rd=(rs1<SE(imm))`	I	0010011	010	-------
Set Less Than Immediate					Immediate set if less (signed)
`lw rd,imm(rs1)`	`rd=[rs1+SE(imm)]`	I	0000011	010	-------
Load Word					Load word

2.4. S-type Instructions (Store)

S-type instructions store data (operands: `imm, rs1, rs2`). These write the value of `rs2` to the address `imm(rs1)`. The required S-type instruction is shown in table 2.4.

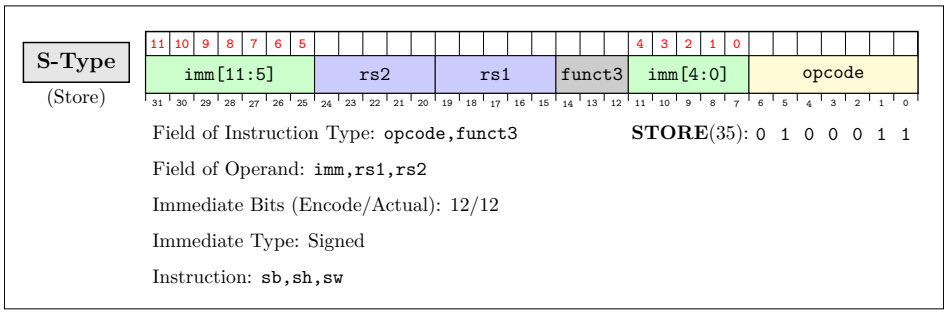


Figure 2.3: S-type Instruction Encoding

Table 2.4: S-type Instruction List

Instruction	Description	Type	opcode	f3	f7
sw rs2,imm(rs1)	[rs1+SE(imm)]=rs2	S	0100011	010	-------
Store Word					Store word

2.5. B-type Instructions (Branch)

B-type instructions are for conditional branches (operands: `imm, rs1 , rs2`). These compare `rs1` and `rs2` and branch if the condition is met. The required B-type instruction is shown in table 2.5.

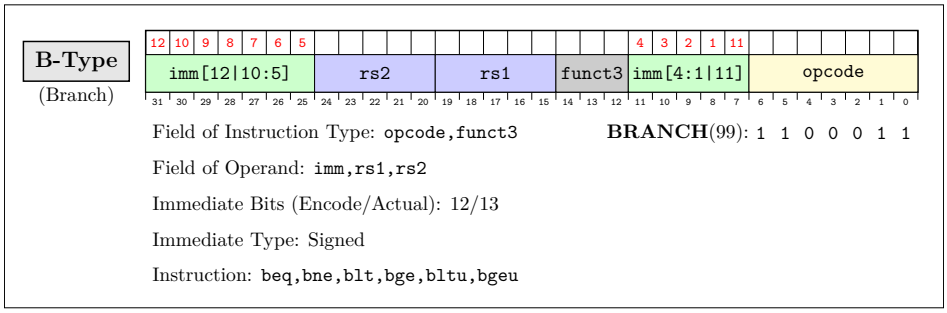


Figure 2.4: B-type Instruction Encoding

Table 2.5: B-type Instruction List

Instruction	Description	Type	opcode	f3	f7
bne rs1,rs2,lab	if(rs1!=rs2) pc+=SE(imm,1'b0)	B	1100011	001	-------
Branch If Not Equal					Branch if not equal

2.6. J-type Instructions (Jump)

J-type instructions are for unconditional jumps and link (operands: `imm, rd`). The required J-type instruction is shown in table 2.6.

Table 2.6: J-type Instruction List

Instruction	Description	Type	opcode	f3	f7
`jal rd,imm`	`rd=pc+4, pc=pc+SE(imm)`	J	1101111	000	-------
Jump and Link				Jump and link	

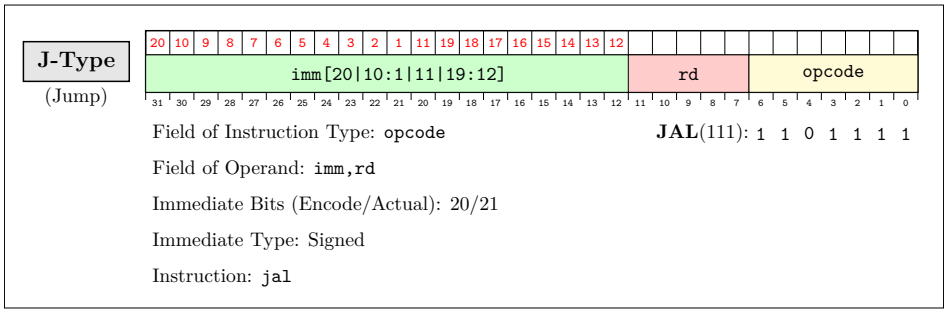


Figure 2.5: J-type Instruction Encoding

3 Architecture of miniCPU

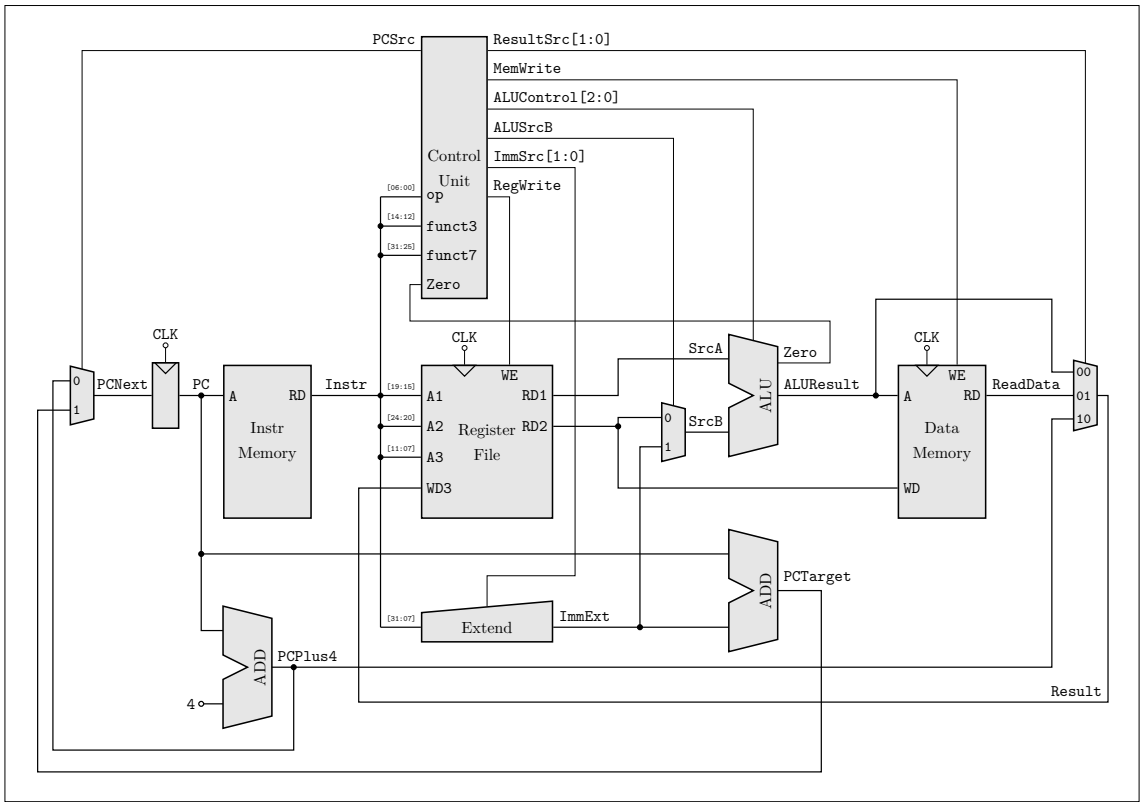


Figure 3.1: Overall Architecture of miniCPU

The miniCPU uses a classic five-stage structure: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). In this project, you will implement a single-cycle CPU, where each instruction completes all five stages in one clock cycle. Figure 3.1 shows the main components and their connections.

3.1. Main Module Descriptions

3.1.1 Register File

The register file is the core storage component of the CPU. According to the datapath diagram, it contains 32 general-purpose 32-bit registers.

Key Features:

- Two read ports (`A1/RD1`, `A2/RD2`) and one write port (`A3/WD3`).
- Read is combinational logic; write occurs on the rising clock edge when write enable (`WE`) is asserted.
- Register `x0` is always zero.

Interface definition is shown in tables 3.1 and 3.2.

Table 3.1: Register File Input Signals

Input	Width	Description
`CLK`	1	Clock signal
`RST`	1	Reset signal (asynchronous)
`WE`	1	Write enable (from Control Unit's `RegWrite`)
`A1`	5	Address of source register `rs1` (`Instr[19:15]`)
`A2`	5	Address of source register `rs2` (`Instr[24:20]`)
`A3`	5	Address of destination register `rd` (`Instr[11:07]`)
`WD3`	32	Data of destination register `rd` data (from `ResultMux`)

Table 3.2: Register File Output Signals

Output	Width	Description
`RD1`	32	Data of source register `rs1` (to `SrcA`)
`RD2`	32	Data of source register `rs2` (to `SrcBMux` and `DataMemory`)

3.1.2 Program Counter (PC)

The PC is a 32-bit register that stores the current instruction address.

Key Features:

- Updates on the rising clock edge, latching the value of `PCNext` to `PC`.

- `PCNext` is selected externally: either sequential `PCPlus4` or a jump target `PCTarget`.

Interface definition is shown in tables 3.3 and 3.4.

Table 3.3: PC Input Signals

Input	Width	Description
`CLK`	1	Clock signal
`RST`	1	Reset signal (asynchronous)
`PCNext`	32	Next instruction address (from `PCMux`)

Table 3.4: PC Output Signals

Output	Width	Description
`PC`	32	Current instruction address

3.1.3 Arithmetic Logic Unit (ALU)

The ALU performs arithmetic and logic operations.

Key Features:

- **Arithmetic Operations:** Performs addition (`add`/`addi`) and subtraction (`sub`), computing `SrcA + SrcB` or `SrcA - SrcB`. Addition is also used for address calculation in load/store instructions (`lw`/`sw`).
- **Logical Operations:** Executes bitwise AND (`and`/`andi`) and OR (`or`/`ori`) operations on the two operands.
- **Comparison:** Implements set-less-than (`slt`/`slti`) by performing signed comparison. If `SrcA < SrcB` (treating both as signed integers), the result is 1; otherwise, 0.
- **Zero Flag Generation:** Produces a `Zero` output signal that is asserted (`1`) when `ALUResult` equals zero. For the required `bne` instruction, the controller should take the branch when the comparison result is nonzero; `beq` is not part of the required 13-instruction contract.
- **Operation Selection:** The 3-bit `ALUControl` signal selects the operation: `000` (Add), `001` (Sub), `010` (And), `011` (Or), `101` (Slt).

Interface definition is shown in tables 3.5 and 3.6.

Table 3.5: ALU Input Signals

Input	Width	Description
`SrcA`	32	Operand A (from `RD1`)
`SrcB`	32	Operand B (from `RD2` or `ImmExt`, based on `SrcBMux`)
`ALUControl`	3	ALU operation control

Table 3.6: ALU Output Signals

Output	Width	Description
`ALUResult`	32	Alu operation result
`Zero`	1	Zero flag (set to `1` if `ALUResult` is zero)

3.1.4 Data Memory

Data memory is used for data segment access.

Key Features:

- Read is combinational (output changes with address).
- Write is controlled by `WE` and occurs on the rising clock edge.

Interface definition is shown in tables 3.7 and 3.8.

Table 3.7: Data Memory Input Signals

Input	Width	Description
`CLK`	1	Clock signal
`RST`	1	Reset signal (asynchronous)
`WE`	1	Write enable (from `MemWrite`)
`A`	32	Memory address (from `ALUResult`)
`WD`	32	Write data to memory (from `RD2`)

Table 3.8: Data Memory Output Signals

Output	Width	Description
`RD`	32	Read data from memory

3.1.5 Immediate Extender

The Immediate Extender extracts the immediate value from the instruction and sign-extends it to 32 bits based on the instruction type.

Key Features:

- Extracts immediate fields from `Instr` based on `ImmSrc`.
- Performs sign or zero extension to generate a 32-bit `ImmExt`.

Interface definition is shown in tables 3.9 and 3.10.

Table 3.9: Immediate Extender Input Signals

Input	Width	Description
`Instr`	32	Current instruction (from `InstructionMemory`)
`ImmSrc`	2	Immediate type select (from Controller)

Table 3.10: Immediate Extender Output Signals

Output	Width	Description
`ImmExt`	32	Extended immediate value (to `SrcBMux` and `PCTarget`)

3.1.6 Controller

The Controller serves as the central logic unit of the processor, responsible for generating all control signals required to coordinate the datapath based on the input instruction. It decodes key instruction fields (Opcode, Funct3, Funct7) and incorporates feedback signals (Zero) from the datapath to drive multiplexers, the register file, and the ALU.

To accommodate the RISC-V ISA, the decoding logic operates through three primary functional dimensions:

- **Datapath Configuration and State Control:** The controller identifies the instruction format and category based on the `opcode` field. Based on this classification, it generates global control signals:
  - **Resource Access:** Generates `RegWrite` and `MemWrite` signals to control write permissions for the Register File and Data Memory, respectively.
  - **Data Routing:** Generates multiplexer selector signals (e.g., `ALUSrcB`, `ResultSrc`) to configure the data flow path within the processor (e.g., selecting between register or immediate operands, determining the source for register write-back).

  - **Immediate Extraction:** Generates the `ImmSrc` signal, instructing the Extend Unit to correctly extract and sign-extend the immediate value according to the current instruction format.

- **ALU Operation Control:** The `ALUControl` signal dictates the specific arithmetic or logical operation performed by the ALU. Its generation follows a two-level logic:
  - For certain instruction categories (such as memory access or control flow), the ALU operation is implicitly defined by the instruction type (e.g., address calculation or comparison). In these cases, the controller determines the operation solely from the `opcode`.
  - For general arithmetic-logic instructions, the specific operation is not uniquely defined by the `opcode`. The controller must further decode the `funct3` and `funct7` fields to resolve the precise sub-operation (e.g., addition, subtraction, logical ops).
- **Branch and Jump Logic:** The controller determines the address of the next instruction. By combining the instruction type (Branch or Jump) with the `Zero` flag from the ALU (indicating condition status), it generates the `PCSrc` signal. This signal dictates whether the Program Counter (PC) proceeds sequentially (PC+4) or updates to the computed target address (PCTarget).

Control Path Supporting Modules

Several small combinational modules work alongside the controller to facilitate datapath operations:

1. `PCPlus4Adder`
   - **Function:** Computes the address of the next sequential instruction.
   - **Inputs:** `PC` (32-bit, current program counter value)
   - **Outputs:** `PCPlus4` (32-bit, computed as `PC + 4`)
   - **Usage:** Provides the default next instruction address for non-branching instructions and serves as the return address for `jal`.

2. `PCTargetAdder`
    - **Function:** Calculates the target address for branch and jump instructions.
    - **Inputs:** `PC` (32-bit), `ImmExt` (32-bit, sign-extended immediate)
    - **Outputs:** `PCTarget` (32-bit, computed as `PC + ImmExt`)
    - **Usage:** Used for relative addressing in branch (`bne`) and jump (`jal`) instructions.
3. `PCMux` **(PC Source Multiplexer)**
    - **Function:** Selects the next program counter value.
    - **Inputs:** `PCPlus4` (32-bit), `PCTarget` (32-bit), `PCSrc` (1-bit control signal)
    - **Outputs:** `PCNext` (32-bit)
    - **Operation:** When `PCSrc = 0`, outputs `PCPlus4`; when `PCSrc = 1`, outputs `PCTarget`.
    - **Usage:** Implements control flow by selecting between sequential execution and branch/jump targets.
4. `SrcBMux` **(ALU Source B Multiplexer)**
    - **Function:** Selects the second operand for the ALU.
    - **Inputs:** `RD2` (32-bit, register data), `ImmExt` (32-bit, extended immediate), `ALUSrcB` (1-bit control signal)
    - **Outputs:** `SrcB` (32-bit)
    - **Operation:** When `ALUSrcB = 0`, outputs `RD2` (for R-type instructions); when `ALUSrcB = 1`, outputs `ImmExt` (for I-type instructions and address calculations).
5. `ResultMux` **(Result Source Multiplexer)**
    - **Function:** Selects the data to be written back to the register file.
    - **Inputs:** `ALUResult` (32-bit), `ReadData` (32-bit, from data memory), `PCPlus4` (32-bit), `ResultSrc` (2-bit control signal)
    - **Outputs:** `Result` (32-bit, connected to `WD3` of register file)
    - **Operation:**

- Select `ALUResult` for arithmetic/logic instructions
- Select `ReadData` for load instructions
- Select `PCPlus4` for `jal` instruction to save return address

Interface definition is shown in tables 3.11 and 3.12.

Table 3.11: Controller Input Signals

Input	Width	Description
`op`	7	Opcode (`Instr[6:0]`)
`funct3`	3	3-bit function code (`Instr[14:12]`)
`funct7`	7	7-bit function code (`Instr[31:25]`)
`Zero`	1	Zero flag from ALU

Table 3.12: Controller Output Signals

Output	Width	Description
`PCSrc`	1	PC source select (`0`: `PCPlus4`, `1`: `PCTarget`)
`ResultSrc`	2	Write-back data select (`00`: `ALUResult`, `01`: `ReadData`, `10`: `PCPlus4`)
`MemWrite`	1	Data memory write enable
`ALUControl`	3	ALU operation (`000`: `Add`, `001`: `Sub`, `010`: `And`, `011`: `Or`, `101`: `Slt`)
`ALUSrcB`	1	ALU operand B select (`0`: `RD2`, `1`: `ImmExt`)
`ImmSrc`	2	Immediate extension type (`00`: `I`, `01`: `S`, `10`: `B`, `11`: `J`)
`RegWrite`	1	Register file write enable

4 Requirements

This chapter defines the 2026 Spring project requirements and grading rules. The required implementation remains the 13-instruction miniCPU described in the ISA chapter: `add`, `addi`, `sub`, `and`, `andi`, `or`, `ori`, `slt`, `slti`, `lw`, `sw`, `bne`, and `jal`.

4.1. Academic Policy

- You may use AI tools to assist with processor design, assembler design, and report writing. You are responsible for all AI-generated results that you submit.
- Every group member should understand the submitted processor, assembler, tests, and report well enough to explain them independently during the face-to-face check.
- If you reference public resources, including open-source code or AI-generated suggestions, cite the source in your written report and explain what you changed.
- Plagiarism, ghostwriting, inter-group copying, or any other academic misconduct will receive the most severe penalty allowed by course policy. The teaching team may use both technical and non-technical methods to investigate misconduct.
- Poorly organized or superficial reports may lose points even if writing tools were used. Tool assistance does not replace technical understanding.

4.2. Grading Overview

The base project is worth 100 points. Optional bonus work can add points, but the total bonus contribution is capped at 25 points.

Table 4.1: 2026 Spring Grading Overview

Component	Points	Main Grading Method
Logisim Processor	**30**	Gradescope functional tests
Python Assembler	**20**	Gradescope assembler tests
Reports	**25**	Submitted PDF report
Check	**25**	10-minute face-to-face check
Bonus	**up to 25**	Optional extra work with evidence

4.3. Logisim Processor (30 pts)

Implement the 13 required instructions in Logisim. You may choose your internal datapath, control logic, module boundaries, and signal names. Gradescope tests are based on observable behavior through data memory (DMEM), and the tests do not impose constraints on internal signals. Correct behavior matters more than matching a particular internal wiring style.

The provided Logisim template contains only instruction memory (IMEM), data memory (DMEM), and the necessary module partitioning. You must complete the processor logic around that template.

4.3.1 Processor Rubric

Table 4.2: Logisim Processor Rubric

Item	Points	Notes
`addi`	**4**	Required for many tests to construct values
`sw`	**4**	Required for DMEM-based answer checking
`add, sub, and, andi, or, ori, slt, slti, lw, bne, jal`	**22**	2 points for each of the other 11 required instructions

4.3.2 Testing Details

- Gradescope will run assembly programs and check the expected final results in DMEM.
- Some tests are intentionally nontrivial. For example, an `addi` immediate may contain bit patterns that look like `funct7 = 0100000`; your decoder should still classify the instruction by opcode and function fields correctly.
- Students will receive the test assembly code, the precompiled instruction binaries, and the expected results for the provided tests.
- You should also design your own tests that inspect register updates, memory updates, PC changes, branches, jumps, negative immediates, and edge cases involving `x0`.

4.4. Python Assembler (20 pts)

Write a Python assembler that converts the required miniCPU assembly language into machine-code output suitable for loading into IMEM. The assembler must support the 13 required instructions and their formats exactly as specified in the ISA chapter.

4.4.1 Syntax Requirements

Your assembler must support:

- comments;
- labels for branch and jump targets;
- comma-separated operands, such as `add x1,x2,x3`;
- memory operands, such as `lw x1,8(x2)` and `sw x3,-4(x4)`.

4.4.2 Assembler Rubric

Table 4.3: Python Assembler Rubric

Test Group	Points	Focus
Arithmetic instructions	**5**	R-type and immediate arithmetic/logic encodings
Memory instructions	**5**	`lw/sw` address offsets and immediate encoding
Jump and branch instructions	**5**	Label resolution and PC-relative offsets
Standard test assembly	**5**	Comment handling and complete assembly parsing

4.5. Reports (25 pts)

The report is the PDF document you submit to Gradescope. In older wording this part may have been called "Document"; in this handout, "Reports" refers to this 25-point PDF submission, while "Check" or "Face-to-Face Check" refers to the separate 25-point live check.

You may write the report in Markdown or LaTeX. A project submitted as a Word document is not accepted and is treated as an incomplete project. A handwritten report receives zero points for the Reports component, but the project implementation can still receive the implementation-related points it earns.

Your report should include:

1. **Processor.** Describe the datapath, controller, immediate extension, branch/jump logic, register file behavior, memory interface, and any design choices not directly fixed by the handout.
2. **Assembler.** Describe the supported syntax, parsing strategy, label handling, encoding strategy, and examples.
3. **Testing.** Provide representative assembly programs, expected register or DMEM states, actual results, screenshots or waveforms, and reproduction steps.
4. **Debugging.** Explain known issues, how you found them, and how you would fix them.
5. **Bonus.** If applicable, describe the motivation, implementation, tests, and impact on the required processor or assembler.

Use clear section structure, readable figures, and code formatting. A disorganized collection of text and screenshots may lose points even when the implementation works.

4.6. Check (25 pts)

Each group has 10 minutes for the face-to-face check:

- 5 minutes for a presentation or live demonstration of the processor and assembler;
- 5 minutes for questions from the teaching team.

The presentation format is flexible. You may use slides, open Logisim directly, run your assembler, or combine these methods. If you implemented bonus work or distinctive design choices, focus part of the presentation on those features.

Questions are designed to evaluate whether you understand what you submitted. Be prepared to explain control signal generation, immediate encoding, branch and jump behavior, memory access, assembler label resolution, test design, and each group member's contribution. There is no single standard script for the check; the discussion will follow your project.

4.7. Bonus (up to 25 pts)

Bonus tasks are optional. Bonus points can accumulate across categories, but the total bonus contribution is capped at 25 points.

Table 4.4: Bonus Rubric

Bonus Item	Points	Notes
Complete all R-type instructions	**2**	Beyond the required subset where applicable
Complete all I-type instructions	**8**	Includes correct immediate and load behavior
Complete all S-type instructions	**4**	Includes all store variants
Complete all B-type instructions	**3**	Includes all branch conditions
Complete all U-type instructions	**3**	Includes upper-immediate encodings
Complete five-stage pipeline	**25**	Must preserve correctness and handle hazards
Hardware-language implementation	**25**	For example, Verilog, Chisel, SpinalHDL, or another HDL
LaTeX report	**3**	Awarded for a complete, well-formatted LaTeX report
Marp or Beamer slides	**2**	Awarded for usable defense slides

4.7.1 Bonus Requirements and Warnings

- No standard tests are provided for extra instructions. If you claim instruction-set bonus points, you must design tests that demonstrate correctness.
- Both the processor and the assembler must support every bonus instruction involved in any bonus you claim.
- Half-word and byte memory instructions must be implemented completely, including addressing, data extraction or insertion, sign or zero extension, and memory write masking where needed. Incomplete support will not receive the corresponding bonus.
- HDL implementations must include self-built tests showing that the design can earn the base 30 Logisim-processor points. Contact the TAs early if you plan to use an HDL flow.
- Pipeline implementations must demonstrate correct handling of the three hazard categories: data hazards, control hazards, and structural hazards. Contact the TAs early if you plan to claim pipeline bonus.
- Innovation is encouraged. Possible directions include non-single-cycle microarchitectures such as multi-cycle or pipelined designs, RV32M/RV32F/RV32C subsets, pseudo-instructions such as `nop`

or `li`, Logisim peripherals, and application programs that exercise your processor.

4.8. Submission

Submit all materials by the course deadline. Late submissions may affect your grade.

1. **Project Code**: Submit a ZIP package to Gradescope containing your Logisim processor, Python assembler, source code, test assembly programs, generated machine-code files, expected results, and run or simulation instructions.
2. **Reports**: Submit the report as a PDF to Gradescope.
3. **Check Materials**: Prepare any slides, demos, or notes needed for the face-to-face check. If you claim bonus points, make sure the check materials and report both contain supporting evidence.