CSE 140 Final Project Report

PROJECT TITLE: Pipelined RISC-V Emulator

TEAM MEMBERS: Anthony Rubick

Code Structure and Implementation

The project is implemented in the Rust programming language, I won't go into detail about the language itself, but for a quick primer on the syntax I'd recommend this article, the official book, Rust by Example, or this cheatsheet.

The code is comprised of two separate crates, a library (`lib.rs`) and a binary (`main.rs`). The library contains all the logic and data structures for the emulator. Whereas the binary is a simple wrapper that takes in a file path to one of the sample programs, sets up the state (registers and data memory), and executes the program through the library.

Dependencies

The library uses the following dependencies:

- `anyhow`: for easy error handling
- `ux`: for custom-length integer types (by default, Rust only has 8, 16, 32, 64, and 128-bit integers)

as well as the following additional dependencies for testing:

- `pretty_assertions`: for better assertion error messages (e.g. pretty-printing the values of the variables in the assertion)

As you can see, the dependencies are minimal and mostly for error handling and testing purposes. That is to say, the emulator is implemented from scratch, without relying on any external libraries for the core logic.

High-Level Overview

The core functionality of the emulator is split among various modules, each of which is responsible for a different aspect of the emulator. The modules are as follows:

- `cpu` (`cpu.rs`): This module contains the implementation of the CPU. It has functions for fetching, decoding, executing, memory access, and write back stages of the instruction cycle. It also holds the state of the CPU (register file, data and instruction memory, pc, stage registers, etc.). This is arguably the most important module, as it is what connects all the other modules together. This module also contains implementations of Data Memory (the `DataMemory` struct), Instruction Memory (the `InstructionMemory` struct), and the Branch and Jump Unit (the `branching_jump_unit` function).
- `alu` (`alu.rs`): This module contains the implementations of the ALU (Arithmetic Logic Unit) and the ALU Control Unit, represented by the `alu` and `alu_control_unit`

functions, respectively. The ALU is responsible for performing arithmetic and logical operations on the data. The ALU Control Unit is responsible for determining which operation to perform based on the instruction.

- `registers` (`registers.rs`): This module contains the implementation of the register file, represented by the `Registers` struct. The register file is a wrapper around an array of 32 32-bit integers, representing the 32 registers of the RISC-V ISA. It provides methods for reading and writing to the registers by an index implemented as the `RegisterMapping` Enum.
- `instruction` (`instruction.rs`): This module contains the implementation of the `Instruction` struct, which represents a RISC-V instruction. It provides methods for decoding an instruction from a 32-bit integer, as well as extracting various fields from the instruction (e.g. `opcode`, `funct3`, `funct7`, etc.).
- `signals` (`signals.rs`): This module contains definitions for the various control signals used by the CPU. These signals are used to control which operations are performed by the CPU to properly execute a given instruction. This module also contains an implementation of the Control Unit, represented by the `control_unit` function, which is responsible for generating these control signals based on the `opcode` of the instruction.
- `stages` (`stages.rs`): This module contains definitions for the various stage registers used by the CPU, which are responsible for propagating information between pipeline stages. These stage registers are used to pipeline the execution of instructions, allowing multiple instructions to be in different stages of execution at the same time. The stage registers are implemented as Enums (the C/CPP equivalent would be a tagged union), and wrapped into the `StageRegisters` struct.
- `hazard_detection` (`hazard_detection.rs`): This module contains implementations of both the Forwarding Unit, and the Hazard Detection Unit, represented by the `forwarding_unit` function and `HazardDetectionUnit` struct, respectively. The Forwarding Unit is responsible for determining if a value can be forwarded from a previous stage to the current stage, while the Hazard Detection Unit is responsible for detecting stall conditions (e.g. a load-use hazard).
    - Note about Control Hazards: Control hazards are handled by predicting that branches are Not Taken, and then flushing the pipeline if the Branch and Jump Unit determines that the branch should be Taken.
- `utils` (`utils.rs`): This module contains various utility functions for converting bit strings (strings with only '0's and '1's) to bit vectors, and for converting bit vectors into integers. These functions are used to parse instructions from the sample programs.

Detailed Overview

The code base is highly documented, with detailed comments explaining the purpose of each function, struct, and module.

The following is simply a recap of the most important functions and structs in each module, along with a brief explanation of their purpose.

Implementations (or parts of them) will often be omitted for brevity, but documentation will not be, the full code can be found in the repository (which will be made public after submission).

Additionally, derived traits such as `Debug`, `Clone`, etc. are implemented for most structs, but ommitted for brevity.

CPU module

`InstructionMemory` struct

The `InstructionMemory` struct is a simple wrapper around a vector of 32-bit integers, representing the instruction memory of the CPU.

```
/// an array that holds the instructions of the program.
/// Each instruction is a 32-bit integer.
/// The program counter (PC) will be used to index this array to get the
current instruction.
/// The PC will be updated by the `Fetch()` function to get the next
instruction in the next cycle.
pub struct InstructionMemory {
    instructions: Vec<u32>,
}
```

It provides a constructor, and a method for fetching an instruction at a given address.

```rust
impl InstructionMemory {
    /// create a new `InstructionMemory` instance.
    ///
    /// # Arguments
    ///
    /// * `rom` - the program instructions
    ///
    /// # Returns
    ///
    /// a new `InstructionMemory` instance
    #[must_use]
    pub fn new(rom: Vec<u32>) -> Self {
        Self { rom }
    }

    /// get the instruction at the given program counter value.
    ///
    /// # Panics
    ///
    /// * if the program counter value is not aligned to 4 bytes
    ///
    /// # Arguments
    ///
    /// * `pc` - the program counter value
    ///
    /// # Returns
    ///
    /// * `Some(u32)` - the instruction at the given program counter value
    #[must_use]
    pub fn get_instruction(&self, pc: u32) -> Option<u32> {
        if pc as usize / 4 >= self.rom.len() {
            // we've reached the end of the program
            return None;
        }
        // check invariant
        assert!(pc % 4 == 0, "PC not aligned to 4 bytes");

        Some(self.rom[pc as usize / 4])
    }
}
```

`DataMemory` struct

The `DataMemory` struct is a simple wrapper around a vector of 32-bit integers, representing the data memory of the CPU.

```rust
/// an array that holds the data of the program.
/// Each data is a 32-bit integer.
pub struct DataMemory {
    d_mem: [u32; 32],
}
```

It provides a constructor, and methods for reading and writing to the data memory.

```rust
impl DataMemory {
    /// create a new `DataMemory` instance.
    ///
    /// # Returns
    ///
    /// a new `DataMemory` instance, initialized with all zeros
    #[must_use]
    pub const fn new() -> Self {
        Self { d_mem: [0; 32] }
    }

    /// read a 32-bit value from the data memory
    ///
    /// # Panics
    ///
    /// * if the address is out of bounds or not aligned to 4 bytes
    ///
    /// # Arguments
    ///
    /// * `address` - the address to read from
    ///
    /// # Returns
    ///
    /// the 32-bit value at the address
    #[must_use]
    pub fn read(&self, address: u32) -> u32 {
        // check invariants first
        assert!(
            address as usize / 4 < self.d_mem.len(),
            "Address out of bounds"
        );
        assert!(address % 4 == 0, "Address not aligned to 4 bytes");

        self.d_mem[address as usize / 4]
    }

    /// write a 32-bit value to the data memory
    ///
    /// # Panics
    ///
    /// * if the address is out of bounds or not aligned to 4 bytes
    ///
    /// # Arguments
    ///
    /// * `address` - the address to write to
    /// * `value` - the value to write
```

```
    pub fn write(&mut self, address: u32, value: u32) {
        // check invariants first
        assert!(
            address as usize / 4 < self.d_mem.len(),
            "Address out of bounds"
        );
        assert!(address % 4 == 0, "Address not aligned to 4 bytes");

        self.d_mem[address as usize / 4] = value;
    }
}
```

`branching_jump_unit` function

The `branching_jump_unit` function is responsible for determining the next value of the program counter based on the current instruction and the control signals.

```rust
/// The Branch and Jump Unit is responsible for determining whether a branch
or jump should be taken.
///
/// # Arguments
///
/// * `branch_jump` - a 2 bit control signal that tells the Branching and
Jump Unit what type of branching to consider.
/// * `alu_zero` - a signal that tells the Branching and Jump Unit whether
the ALU result is zero.
/// * `alu_control` - the operation that the ALU performed.
/// * `operands_equal` - a signal that tells the Branching and Jump Unit
whether the operands to the alu were are equal.
/// * `funct3` - the funct3 field of the instruction (only for branch
instructions)
///
/// # Returns
///
/// * `Ok(None)` - if no branch or jump should be taken
/// * `Ok(Some((u32, PCSrc)))` - the target address and the source of the
next PC value (which also determines where the returned target address
should be stored)
/// * `Err(&str)` - if the arguments are invalid or the operation is not
supported
fn branching_jump_unit(
    branch_jump: BranchJump,
    alu_control: ALUControl,
    alu_result: u32,
    alu_zero: bool,
    operands_equal: bool,
    funct3: Option<u3>,
    immediate: Immediate,
) -> Result<PCSrc> {
    ... // implementation omitted
}
```

The function takes in various arguments, such as the control signals, the ALU result, and the immediate value, and returns the next value of the program counter, along with the source of the next PC value (which also determines where the returned target address should be stored).

If the BranchJump control signal indicates a branch instruction, it determines if the branch should be taken based on the ALU result, the ALU operation, and the operands. If the BranchJump control signal indicates a jump instruction, it calculates the target address based on the immediate value. Otherwise (for non-branch and non-jump instructions). It returns a variant of the `PCSrc` enum that indicates the next value of the program counter.

`CPU` struct

The `CPU` struct is the main struct that holds the state of the CPU.

```rust
/// a struct that represents the CPU.
pub struct CPU {
    /// the program counter value of the current instruction.
    pc: u32,
    /// the next program counter value.
    pc_src: PCSrc,
    /// signal that, when flipped, flushes the IF stage (prevents it from
running for a cycle)
    /// this is used to handle stalls in the pipeline
    if_flush: bool,

    /// the total number of clock cycles that the CPU has executed.
    total_clock_cycles: u64,
    /// the stage registers of the CPU.
    /// These registers will be updated by the corresponding stage
functions.
    stage_registers: StageRegisters,

    /// an integer array that has 32 entries.
    /// This register file array will be initialized to have all zeros
unless otherwise specified.
    /// This register file will be updated by `write_back()` function.
    /// This register file can be indexed by with `RegisterMapping` enum
variants for ergonomics.
    rf: RegisterFile,
    /// an integer array that has 32 entries.
    /// Each entry of this array will be considered as one 4-byte memory
space.
    /// We assume that the data memory address begins from `0x0`.
    /// Therefore, each entry of the `d_mem` array will be accessed with the
following addresses.
    ///
    /// | Memory address calculated at Execute() | Entry to access in
`d_mem` array |
    /// |----------------------------------------|-------------------------
--------|
    /// |              `0x00000000`              |            `d_mem[0]`
|
    /// |              `0x00000004`              |            `d_mem[1]`
|
    /// |              `0x00000008`              |            `d_mem[2]`
|
    /// |                   …                    |                 …
|
    /// |              `0x0000007C`              |            `d_mem[31]`
|
```

```
    d_mem: DataMemory,
    /// an array that holds the instructions of the program.
    /// Each instruction is a 32-bit integer.
    /// The program counter (PC) will be used to index this array to get the
current instruction.
    /// The PC will be updated by the Fetch() function to get the next
instruction in the next cycle.
    i_mem: InstructionMemory,
}
```

It provides a constructor, methods for initializing registers and data memory, and methods for executing the CPU, some of these functions will be explained in more detail in the following sub-sections.

```rust
impl CPU {
    /// Initialize the CPU state
    ///
    /// # Arguments
    ///
    /// * `rom` - the program instructions
    ///
    /// # Returns
    ///
    /// a new `CPU` instance
    #[must_use]
    pub fn new(rom: Vec<u32>) -> Self {
        Self {
            pc: 0,
            pc_src: PCSrc::Init,
            if_flush: false,
            total_clock_cycles: 0,
            stage_registers: StageRegisters::default(),
            rf: RegisterFile::new(),
            d_mem: DataMemory::new(),
            i_mem: InstructionMemory::new(rom),
        }
    }

    /// Initialize the register file with the given mappings
    ///
    /// exposes the `initialize` method of the `RegisterFile` struct
    pub fn initialize_rf(&mut self, mappings: &[(RegisterMapping, u32)]) {
        self.rf.initialize(mappings);
    }

    /// Initialize the data memory with the given data
    ///
    /// # Arguments
    ///
    /// * `data` - a list of tuples where the first element is the address
to write to and the second element is the value to write
    pub fn initialize_dmem(&mut self, data: &[(u32, u32)]) {
        for (address, value) in data {
            self.d_mem.write(*address, *value);
        }
    }

    /// # Returns
    ///
    /// the total number of clock cycles that the CPU has executed
```

```rust
    #[must_use]
    pub const fn get_total_clock_cycles(&self) -> u64 {
        self.total_clock_cycles
    }

    /// is the program over
    ///
    /// # Returns
    ///
    /// `true` if the program is over, `false` otherwise
    #[must_use]
    pub fn is_done(&self) -> bool {
        ... // implementation omitted
        // true if the program counter is at the end of the instruction
memory
        // and all stages are flushed
    }

    /// Main loop of the CPU simulator
    ///
    /// This function will run the CPU until the program is over
    ///
    /// It will print the report of each clock cycle
    pub fn run(&mut self) {
        loop {
            println!();
            match self.run_step() {
                Ok(report) => {
                    println!("{report}");
                }
                Err(e) => {
                    eprintln!("Error: {e}");
                    break;
                }
            }

            if self.is_done() {
                break;
            }
        }

        println!("program terminated:");
        println!("total execution time is {} cycles",
self.total_clock_cycles);
    }
```

```rust
    /// Body of the main loop of the CPU simulator, separated for testing
purposes
    ///
    /// This function will run the CPU for one clock cycle
    ///
    /// Pipeline stages are executed in reverse order to simplify the
implementation
    ///
    /// # Returns
    ///
    /// * `Ok(Report)` - a report of what happened in the CPU during a clock
cycle
    ///
    /// # Errors
    ///
    /// * if there is an error in the CPU pipeline
    pub fn run_step(&mut self) -> Result<Report> {
        let mut report = String::new();

        self.total_clock_cycles += 1;

        report.push_str(format!("total_clock_cycles {} :\n",
self.total_clock_cycles).as_str());

        let wb_report = self.write_back();
        let mem_report = self.mem();
        self.execute()?;
        self.decode()?;
        let if_report = self.fetch();

        // mem will tell us if data memory was updated, so we add that to
the report
        report.push_str(&mem_report);
        // wb will tell us if registers were updated, so we add those to the
report
        report.push_str(&wb_report);
        // if will tell us if the pc was updated, so we add that to the
report
        report.push_str(&if_report);

        Ok(report)
    }

    /// the Fetch stage of the CPU.
    ///
    /// # Returns
```

```
    ///
    /// a report of what happened in the CPU during the fetch stage
    fn fetch(&mut self) -> String {
        ... // implementation omitted
    }

    /// the Decode stage of the CPU.
    ///
    /// This function will decode the instruction in the IF/ID stage and set
the ID/EX stage registers.
    ///
    /// # Errors
    ///
    /// * if the instruction in the IF/ID stage is invalid
    fn decode(&mut self) -> Result<()> {
        ... // implementation omitted
    }

    /// the Execute stage of the CPU.
    ///
    /// This function will execute the instruction in the ID/EX stage and
set the EX/MEM stage registers.
    ///
    /// # Errors
    ///
    /// * if the ALU control unit fails
    /// * if the branch and jump unit fails
    /// * if an invalid immediate value is found
    fn execute(&mut self) -> Result<()> {
        ... // implementation omitted
    }

    /// the Memory stage of the CPU.
    ///
    /// This function will read or write to the data memory based on the
control signals.
    ///
    /// # Returns
    ///
    /// a report of what happened in the CPU during the memory stage
    fn mem(&mut self) -> String {
        ... // implementation omitted
    }

    /// the Write Back stage of the CPU.
    ///
```

```
    /// This function will write the result of the ALU operation or the
memory read data to the register file.
    ///
    /// # Returns
    ///
    /// a report of what happened in the CPU during the write back stage
    fn write_back(&mut self) -> String {
        ... // implementation omitted
    }
}
```

`fetch` method

The `fetch` method is responsible for fetching the next instruction from the instruction memory and updating the program counter.

Again, a lot of the implementation is omitted for brevity, but the comments give a good idea of what the function does.

```rust
/// the Fetch stage of the CPU.
///
/// # Returns
///
/// a report of what happened in the CPU during the fetch stage
fn fetch(&mut self) -> String {
    // check if the decode stage indicates a stall
    if self.stage_registers.idex == IdEx::Stall {
        // in this case, we don't need to do anything in the fetch stage
        return String::from("pipeline stalled in the decode stage\n");
    }

    // if the execute stage told us to flush the IF stage, we don't need to
do anything
    if self.if_flush {
        self.if_flush = false;
        return String::from("pipeline flushed\n");
    }

    // increment the program counter
    self.pc = self.pc_src.next(self.pc);
    // get the current instruction from the ROM,
    let Some(instruction_code) = self.i_mem.get_instruction(self.pc) else {
        // flush IfId and set pc to PCSrc::END if the program is over
        self.pc_src = PCSrc::End;
        self.stage_registers.ifid = IfId::Flush;
        return String::new();
    };

    // set the IF/ID stage registers
    self.stage_registers.ifid = IFID::If {
        instruction_code,
        pc: self.pc,
    };

    /// report if the pc was modified
    let report: String = /* ommitted */ ;

    // if the pc_src was init, branch, or jump, we need to reset it to next
    if /* ommitted */ {
        self.pc_src = PCSrc::Next;
    }

    report
}
```

The `next` method of the `PCSrc` enum is used to determine the next value of the program counter based on the current value, and whether the pc should be incremented by 4, or set to a branch or jump target.
We will go into details in a later section.

`decode` method

The `decode` method is responsible for decoding the instruction in the IF/ID stage and setting the ID/EX stage registers.

The `decode` method also handles load-use hazards (see the `HazardDetectionUnit` struct for more details), and sets the control signals for the instruction (using the `control_unit` function).

```rust
/// the Decode stage of the CPU.
///
/// This function will decode the instruction in the IF/ID stage and set the
ID/EX stage registers.
///
/// # Errors
///
/// * if the instruction in the IF/ID stage is invalid
fn decode(&mut self) -> Result<()> {
    // if the fetch stage failed, flush and exit early
    let (instruction_code, pc) = match self.stage_registers.ifid {
        /* ommitted */
    };

    // decode the instruction
    let instruction = Instruction::from_machine_code(instruction_code)?;

    // check for a load-use hazard
    if HazardDetectionUnit::prime(instruction, self.stage_registers.idex)
        .detect_stall_conditions()
    {
        self.stage_registers.idex = IdEx::Stall;
        return Ok(());
    }

    // read the register file
    let (rs1, read_data_1) = match instruction {
        /* ommitted */
    };
    let (rs2, read_data_2) = match instruction {
        /* ommitted */
    };

    // sign extend the immediate value
    let immediate: Immediate = match instruction {
        /* ommitted */
    };

    // set the control signals
    let control_signals = control_unit(instruction.opcode())?;

    // set the ID/EX stage registers
    self.stage_registers.idex = IdEx::Id {
        instruction,
        rs1,
        read_data_1,
```

```
            rs2,
            read_data_2,
            immediate,
            pc,
            control_signals,
        };

        Ok(())
    }
```

`execute` method

The `execute` method is responsible for executing the instruction in the ID/EX stage and setting the EX/MEM stage registers.

The `execute` method also handles control hazards (see the `branching_jump_unit` function for more details), and sets the alu control signal (using the `alu_control_unit` function).

```rust
/// the Execute stage of the CPU.
///
/// This function will execute the instruction in the ID/EX stage and set
the EX/MEM stage registers.
///
/// # Errors
///
/// * if the ALU control unit fails
/// * if the branch and jump unit fails
/// * if an invalid immediate value is found
fn execute(&mut self) -> Result<()> {
    // if the decode stage failed, flush and exit early
    let (instruction, read_data_1, read_data_2, immediate, pc,
control_signals) =
        match self.stage_registers.idex {
            /* ommitted */
        };

    // ALU control unit
    let alu_control = alu_control_unit(
        control_signals.alu_op,
        instruction.funct3(),
        instruction.funct7(),
    )?;

    // forwarding unit
    let (forward_a, forward_b) = forwarding_unit(
        self.stage_registers.exmem,
        self.stage_registers.wb_stage,
        self.stage_registers.idex,
    );

    // data forwarding
    let read_data_1 = match (
        forward_a,
        self.stage_registers.exmem,
        self.stage_registers.wb_stage,
    ) {
        (ForwardA::ExMem, ExMem::Ex { alu_result, .. }, _) =>
Some(alu_result),
        (ForwardA::MemWb, _, Wb::Mem { wb_data, .. }) => wb_data,
        _ => read_data_1,
    };
    let read_data_2 = /* similar to read_data_1 */;

    // ALU operation
```

```
    let alu_operand_a: u32 = match control_signals.alu_src_a {
        ALUSrcA::Register => read_data_1.unwrap(),
        ALUSrcA::PC => pc,
        ALUSrcA::Constant0 => 0,
    };
    let alu_operand_b: u32 = match control_signals.alu_src_b {
        ALUSrcB::Register => read_data_2.unwrap(),
        ALUSrcB::Immediate => match immediate {
            Immediate::SignedImmediate(imm) | Immediate::JumpOffset(imm) =>
imm as u32,
            Immediate::UpperImmediate(imm) => imm,
            /* other cases omitted, but they throw errors */
        },
        ALUSrcB::Constant4 => 4,
    };

    let (alu_zero, alu_result) = alu(alu_control, alu_operand_a,
alu_operand_b);

    // signal used by the branch and jump unit to help it resolve the branch
or jump instruction
    let operands_equal = alu_operand_a == alu_operand_b;

    // branch and jump address calculation
    let pc_src = branching_jump_unit(
        control_signals.branch_jump,
        alu_control,
        alu_result,
        alu_zero,
        operands_equal,
        instruction.funct3(),
        immediate,
    )?;

    // set the EX/MEM stage registers
    self.stage_registers.exmem = ExMem::Ex {
        instruction,
        alu_result,
        alu_zero,
        read_data_2,
        pc_src,
        pc,
        control_signals,
    };
    Ok(())
}
```

See also, the `forwarding_unit` function, the `alu` function, and the `alu_control_unit` function for more details.

`mem` method

The `mem` method is responsible for reading or writing to the data memory based on the control signals.

```rust
/// the Memory stage of the CPU.
///
/// This function will read or write to the data memory based on the control
signals.
///
/// # Returns
///
/// a report of what happened in the CPU during the memory stage
fn mem(&mut self) -> String {
    // if the execute stage failed, flush
    let (instruction, alu_result, read_data_2, pc, control_signals) =
        match self.stage_registers.exmem {
            ExMem::Flush => {
                self.stage_registers.memwb = MemWb::Flush;
                return String::new();
            }
            ExMem::Ex { /* ommitted */ } => {
                // if the branch and jump unit told us to take a branch or
jump, we need to flush the pipeline
                /* implementation ommitted */
            }
        };

    let (memwb, report) = match (control_signals.mem_read,
control_signals.mem_write) {
        (true, false) => {
            // load
            /* ommitted */
        }
        (false, true) => {
            // store
            /* ommitted */
        }
        (false, false) => {
            // no memory operation
            /* ommitted */
        }
        (true, true) => panic!("invalid control signals for memory stage"),
    };

    self.stage_registers.memwb = memwb;

    report
}
```

If the Branch and Jump Unit determined that there would be a branch/jump taken in the Execute stage, we don't actually flush the pipeline until this stage.
The reason we do this is due to the fact we run the pipeline in reverse order.

`write_back` method

The `write_back` method is responsible for writing pc+4, the result of the ALU operation, or the memory read data to the register file. It also sets the WB stage registers (which is used for data forwarding, among other things).

```rust
/// the Write Back stage of the CPU.
///
/// This function will write the result of the ALU operation or the memory
read data to the register file.
///
/// # Returns
///
/// a report of what happened in the CPU during the write back stage
fn write_back(&mut self) -> String {
    // if the memory stage failed, flush
    let (instruction, alu_result, mem_read_data, pc, control_signals) =
        match self.stage_registers.memwb {
            MemWb::Flush => {
                self.stage_registers.wb_stage = Wb::Flush;
                return String::new();
            }
            /* other case ommitted, but essentially just unpacking the MemWb
enum */
        };

    match (control_signals.reg_write, instruction.rd()) {
        (true, Some(rd)) => {
            // write to register file
            match control_signals.wb_src {
                crate::signals::WriteBackSrc::NA => {
                    /* ommitted, but sets `wb_data` to None */
                }
                crate::signals::WriteBackSrc::ALU => {
                    /*
                    ommitted, but sets `wb_data` to Some(alu_result),
                    writes the alu result to the register file,
                    returns a report of the register write
                    */
                }
                crate::signals::WriteBackSrc::Mem => {
                    /*
                    ommitted, but sets `wb_data` to `mem_read_data`,
                    writes the memory read data to the register file,
                    returns a report of the dmem write
                    */
                }
                crate::signals::WriteBackSrc::PC => {
                    /*
                    ommitted, but sets `wb_data` to `Some(pc + 4)`,
                    writes the pc + 4 to the rd register,
                    returns a report of the register write
```

```
                    */
                }
            }
        }
        (true, None) | (false, _) => {
            // no write to register file
            String::new() /* Note: no `;` here, so this is the return value
*/
        }
    }
}
```

ALU module

The ALU module contains the implementation of the ALU and the ALU Control Unit.

`alu` function

The `alu` function is the implementation of the ALU, it's pretty self-explanatory.

```rust
/// This function mimics the ALU in a risc-v processor, it takes in the ALU
control signal, two 32-bit unsigned integers and returns a tuple of a
boolean and a 32-bit unsigned integer.
///
/// # Arguments
///
/// * `alu_control` - the ALU control signal, determines the operation to
perform.
/// * `a` - the first operand.
/// * `b` - the second operand.
///
/// # Returns
///
/// A tuple of a boolean and a 32-bit unsigned integer.
///
/// The boolean indicates whether the result of the operation is zero.
///
/// The 32-bit unsigned integer is the result of the operation.
pub fn alu(alu_control: ALUControl, a: u32, b: u32) -> (bool, u32) {
    let result = match alu_control {
        ALUControl::ADD => a.wrapping_add(b),
        ALUControl::SUB => a.wrapping_sub(b),
        ALUControl::AND => a & b,
        ALUControl::OR => a | b,
        ALUControl::SLL => a << b,
        #[allow  ]
        ALUControl::SLT => u32::from((a as i32) < (b as i32)),
        ALUControl::SLTU => u32::from(a < b),
        ALUControl::XOR => a ^ b,
        ALUControl::SRL => a >> b,
        #[allow  ]
        ALUControl::SRA => (a as i32 >> b) as u32,
    };

    (result == 0, result)
}
```

`alu_control_unit` function

The `alu_control_unit` function is the implementation of the ALU Control Unit, it determines which operation to perform based on the instruction.

```rust
/// This function mimics the ALU Control Unit in a risc-v processor.
///
/// The ALU operation signal is a 2 bit signal that tells the ALU Control
Unit what type of instruction is being executed.
///
/// The `funct3` and `funct7` fields are used in combination with `alu_op`
to determine the exact operation to be performed by the ALU.
///
/// This function is an implementation of the following Verilog module:
///
/* ommitted for brevity */
///
/// but it is extended to handle various branch instructions.
///
/// # Arguments
///
/// * `alu_op` - a 2 bit signal that tells the ALU Control Unit what type of
instruction is being executed.
/// * `funct3` - a 3 bit signal that is used in combination with `alu_op` to
determine the exact operation to be performed by the ALU.
/// * `funct7` - a 7 bit signal that is used in combination with `alu_op` to
determine the exact operation to be performed by the ALU.
///
/// # Returns
///
/// The ALU control signal.
///
/// # Errors
///
/// This function will return an error if the combination of `alu_op`,
`funct3` and `funct7` doesn't match any of the valid combinations.
#[allow  ]
pub fn alu_control_unit(
    alu_op: ALUOp,
    funct3: Option<u3>,
    funct7: Option<u7>,
) -> Result<ALUControl> {
    Ok(match (alu_op, funct3.map(u8::from), funct7.map(u8::from)) {
        (ALUOp::ADD, _, _) => ALUControl::ADD,
        (ALUOp::BRANCH, Some(funct3), _) => match funct3 {
            0b000 | 0b001 => ALUControl::SUB,  // beq or bne
            0b100 | 0b101 => ALUControl::SLT,  // blt or bge
            0b110 | 0b111 => ALUControl::SLTU, // bltu or bgeu
            _ => bail!("Invalid funct3 for branch instruction"),
        },
        (ALUOp::FUNCT, Some(funct3), Some(funct7)) => match (funct7, funct3)
```

```
{
            (0b000_0000, 0b000) => ALUControl::ADD,  // add
            (0b010_0000, 0b000) => ALUControl::SUB,  // sub
            (0b000_0000, 0b111) => ALUControl::AND,  // and
            (0b000_0000, 0b110) => ALUControl::OR,   // or
            (0b000_0000, 0b010) => ALUControl::SLT,  // slt
            (0b000_0000, 0b011) => ALUControl::SLTU, // sltu
            (0b000_0000, 0b100) => ALUControl::XOR,  // xor
            (0b000_0000, 0b001) => ALUControl::SLL,  // sll, slli
            (0b000_0000, 0b101) => ALUControl::SRL,  // srl, srli
            (0b010_0000, 0b101) => ALUControl::SRA,  // sra, srai
            _ => bail!("Invalid funct3 and funct7 combination"),
        },
        (ALUOp::FUNCT, Some(funct3), None) => match funct3 {
            0b000 => ALUControl::ADD,  // addi
            0b010 => ALUControl::SLT,  // slti
            0b011 => ALUControl::SLTU, // sltui
            0b100 => ALUControl::XOR,  // xori
            0b110 => ALUControl::OR,   // ori
            0b111 => ALUControl::AND,  // andi
            _ => bail!("Invalid funct3 and funct7 combination"),
        },
        _ => bail!("Invalid ALU operation"),
    })
}
```

Again, the code is fairly self-explanatory, all it's doing it matching combinations of the `ALUOp`, `funct3`, and `funct7` fields to corresponding `ALUControl` signals.

Registers module

The Registers module contains the implementation of the Register File, the defition of the `RegisterMapping` Enum, and implementations of the `Index` and `IndexMut` traits to allow the register file to be indexed by the `RegisterMapping` enum.

`RegisterFile` struct

The `RegisterFile` struct is a simple wrapper around an array of 32-bit integers, representing the register file of the CPU.

```
/// a struct that represents the register file of the CPU.
pub struct RegisterFile {
    registers: [u32; 32],
}
```

It provides a constructor, a method for initializing the register file with mappings, and methods for reading and writing to the register file.

```rust
impl RegisterFile {
    /// Create a new `RegisterFile` with all registers initialized to 0
    #[must_use]
    pub const fn new() -> Self {
        Self {
            registers: [0; REGISTERS_COUNT as usize],
        }
    }

    /// Initialize the register file with the provided defaults, makes
everything else 0
    ///
    /// # Arguments
    ///
    /// * `mappings` - a list of tuples where the first element is the
register to write to and the second element is the value to write
    ///
    /// # Panics
    ///
    /// Panics if the register to write to is `RegisterMapping::Zero`
    pub fn initialize(&mut self, mappings: &[(RegisterMapping, u32)]) {
        self.registers = [0; 32];
        for (mapping, value) in mappings {
            self[*mapping] = *value;
        }
    }

    /// Read the value of a register
    ///
    /// # Arguments
    ///
    /// * `reg` - the register to read from
    ///
    /// # Returns
    ///
    /// The value of the register
    #[must_use]
    pub const fn read(&self, reg: RegisterMapping) -> u32 {
        self.registers[reg as usize]
    }

    /// Write a value to a register
    ///
    /// # Arguments
    ///
    /// * `reg` - the register to write to
```

```
    /// * `value` - the value to write
    pub fn write(&mut self, reg: RegisterMapping, value: u32) {
        self.registers[reg as usize] = value;
    }
}
```

`RegisterMapping` enum

The `RegisterMapping` enum is a simple enum, internally backed by a `u8`, that pairs register names to their underlying index in the register file.

```
/// This enum represents the mapping of the registers to their indices in
the register file.
#[repr(u8)]
pub enum RegisterMapping {
    Zero = 0,
    Ra = 1,
    Sp = 2,
    /*
    ...
    you get the idea
    ...
    */
    T5 = 30,
    T6 = 31,
}
```

`Index` and `IndexMut` trait implementations

This is very simple, just cast the `RegisterMapping` variant to a `usize` and use that as the index to the register file.

This is so simple because the `RegisterMapping` enum is defined to be backed by `u8`, meaning it's treated like any other `u8` value with the benifit of having named variants.

Additionally, for `IndexMut`, we panic if the index is for the zero register, as it should not be written to so a mutable reference to it should not be given.

This is worth discussing, but I will not be including the implementation here as it's sufficiently explained above.

Instruction module

`Instruction` Enum

The `Instruction` enum is a simple enum that represents the different types of instructions that can be executed by the CPU.
Each variant of the enum corresponds to a different type of instruction, and contains the necessary fields to represent that instruction as per the RISC-V ISA.

```rust
/// An enum that represents the different types of instructions that can be executed by the CPU.
pub enum Instruction {
    RType {
        funct7: u7,
        rs2: RegisterMapping,
        rs1: RegisterMapping,
        funct3: u3,
        rd: RegisterMapping,
        opcode: u7,
    },
    IType {
        /// only used for the shift instructions
        funct7: Option<u7>,
        /// only used for the shift instructions
        shamt: Option<u5>,
        imm: i12,
        rs1: RegisterMapping,
        funct3: u3,
        rd: RegisterMapping,
        opcode: u7,
    },
    SType {
        imm: i12,
        rs2: RegisterMapping,
        rs1: RegisterMapping,
        funct3: u3,
        opcode: u7,
    },
    SBType {
        imm: i13, // 12 bits stored in machine code + last bit is always 0
        rs2: RegisterMapping,
        rs1: RegisterMapping,
        funct3: u3,
        opcode: u7,
    },
    UType {
        imm: u20,
        rd: RegisterMapping,
        opcode: u7,
    },
    UJType {
        imm: u21, // 20 bits stored in machine code + last bit is always 0
        rd: RegisterMapping,
        opcode: u7,
```

```
    },
}
```

The `Instruction` enum provides a method to convert a 32-bit machine code instruction into an `Instruction` enum variant.
And it provides methods to get fields of the instruction, such as the opcode, funct3, funct7, etc.

```rust
impl Instruction {
    /// Convert a 32-bit machine code instruction into an `Instruction` enum variant.
    ///
    /// # Arguments
    ///
    /// * `machine_code` - the 32-bit machine code instruction
    ///
    /// # Returns
    ///
    /// * `Result<Instruction>` - The decoded `Instruction`, if the machine code is valid. Otherwise, an error is returned.
    ///
    /// # Errors
    ///
    /// This function will return an error if the machine code is invalid.
    pub fn from_machine_code(machine_code: u32) -> Result<Self> {
        /* basically the same as HW3, so omitted */
    }

    /// Get the opcode of the instruction.
    ///
    /// # Returns
    ///
    /// * `u7` - the opcode of the instruction.
    #[must_use]
    pub const fn opcode(&self) -> u7 {
        /* omitted */
    }

    /// Get the funct3 field of the instruction.
    ///
    /// # Returns
    ///
    /// * `Option<u3>` - the funct3 field of the instruction, if it has one.
    #[must_use]
    pub const fn funct3(&self) -> Option<u3> {
        /* omitted */
    }

    /// Get the funct7 field of the instruction.
    ///
    /// # Returns
    ///
    /// * `Option<u7>` - the funct7 field of the instruction, if it has one.
    #[must_use]
```

```rust
    pub const fn funct7(&self) -> Option<u7> {
        /* omitted */
    }

    /// Get the shamt field of the instruction.
    ///
    /// # Returns
    ///
    /// * `Option<u5>` - the shamt field of the instruction, if it has one.
    #[must_use]
    pub const fn shamt(&self) -> Option<u5> {
        /* omitted */
    }

    /// Get the rd field of the instruction.
    ///
    /// # Returns
    ///
    /// * `Option<RegisterMapping>` - the rd field of the instruction, if it
has one.
    #[must_use]
    pub const fn rd(&self) -> Option<RegisterMapping> {
        /* omitted */
    }

    /// Get the rs1 field of the instruction.
    ///
    /// # Returns
    ///
    /// * `Option<RegisterMapping>` - the rs1 field of the instruction, if
it has one.
    #[must_use]
    pub const fn rs1(&self) -> Option<RegisterMapping> {
        /* omitted */
    }

    /// Get the rs2 field of the instruction.
    ///
    /// # Returns
    ///
    /// * `Option<RegisterMapping>` - the rs2 field of the instruction, if
it has one.
    #[must_use]
    pub const fn rs2(&self) -> Option<RegisterMapping> {
        /* omitted */
```

```
    }
}
```

Signals module

The Signals module contains the definition of the control signals used by the CPU, as well as the implementation of the Control Unit.

Control Unit implementation

The `control_unit` function is the implementation of the Control Unit, it determines the control signals for the instruction based on the opcode.

```rust
/// Control Unit implementation
///
/// # Arguments
///
/// * `opcode` - the opcode of the instruction
///
/// # Returns
///
/// * `ControlSignals` - the control signals that the Control Unit generates
///
/// # Errors
///
/// * if the opcode is not recognized / not supported
///
/// # Description
///
/// the control unit considers 9 types of instructions:
///
/// 1. `lui` instruction
/// 2. `auipc` instruction
/// 3. `jal` instruction
/// 4. `jalr` instruction
/// 5. branch instructions
/// 6. load instructions
/// 7. store instructions
/// 8. R-type instructions
/// 9. I-type instructions
pub fn control_unit(opcode: u7) -> Result<ControlSignals> {
    match u8::from(opcode) {
        // lui
        0b011_0111 => Err(anyhow::anyhow!("lui instruction not supported
yet")),

        // auipc
        0b001_0111 => Err(anyhow::anyhow!("auipc instruction not supported
yet")),

        // jal
        0b110_1111 => Ok(ControlSignals {
            reg_write: true,
            branch_jump: BranchJump::Jal,
            alu_src_a: ALUSrcA::PC,
            alu_src_b: ALUSrcB::Immediate,
            alu_op: ALUOp::ADD,
            mem_write: false,
```

```
            wb_src: WriteBackSrc::PC,
            mem_read: false,
        }),

        // jalr
        0b110_0111 => Ok(ControlSignals {
            reg_write: true,
            branch_jump: BranchJump::Jal,
            alu_src_a: ALUSrcA::Register,
            alu_src_b: ALUSrcB::Immediate,
            alu_op: ALUOp::ADD,
            mem_write: false,
            wb_src: WriteBackSrc::PC,
            mem_read: false,
        }),

        // branch
        0b110_0011 => Ok(ControlSignals {
            reg_write: false,
            branch_jump: BranchJump::Branch,
            alu_src_a: ALUSrcA::Register,
            alu_src_b: ALUSrcB::Register,
            alu_op: ALUOp::BRANCH,
            mem_write: false,
            wb_src: WriteBackSrc::NA,
            mem_read: false,
        }),

        // load
        0b000_0011 => Ok(ControlSignals {
            reg_write: true,
            branch_jump: BranchJump::No,
            alu_src_a: ALUSrcA::Register,
            alu_src_b: ALUSrcB::Immediate,
            alu_op: ALUOp::ADD,
            mem_write: false,
            wb_src: WriteBackSrc::Mem,
            mem_read: true,
        }),

        // store
        0b010_0011 => Ok(ControlSignals {
            reg_write: false,
            branch_jump: BranchJump::No,
            alu_src_a: ALUSrcA::Register,
            alu_src_b: ALUSrcB::Immediate,
```

```
            alu_op: ALUOp::ADD,
            mem_write: true,
            wb_src: WriteBackSrc::NA,
            mem_read: false,
        }),

        // R-type
        0b011_0011 => Ok(ControlSignals {
            reg_write: true,
            branch_jump: BranchJump::No,
            alu_src_a: ALUSrcA::Register,
            alu_src_b: ALUSrcB::Register,
            alu_op: ALUOp::FUNCT,
            mem_write: false,
            wb_src: WriteBackSrc::ALU,
            mem_read: false,
        }),

        // I-type
        0b001_0011 => Ok(ControlSignals {
            reg_write: true,
            branch_jump: BranchJump::No,
            alu_src_a: ALUSrcA::Register,
            alu_src_b: ALUSrcB::Immediate,
            alu_op: ALUOp::FUNCT,
            mem_write: false,
            wb_src: WriteBackSrc::ALU,
            mem_read: false,
        }),

        _ => Err(anyhow::anyhow!("opcode not recognized")),
    }
}
```

We've gone over all these instructions in class, except for: `lui` and `auipc`, which aren't implemented so can be ignored; as well as `jal` and `jalr`, which we'll go into more detail about soon.

Jump Instructions

The `jal` and `jalr` instructions are jump instructions, they set the program counter to a new value, and store the old value in a register.

Let's look at what we set the `ControlSignals` to for these instructions line by line. First, `jal`:

```
...
// jal
0b110_1111 => Ok(ControlSignals {
    reg_write: true,
    branch_jump: BranchJump::Jal,
    alu_src_a: ALUSrcA::PC,
    alu_src_b: ALUSrcB::Immediate,
    alu_op: ALUOp::ADD,
    mem_write: false,
    wb_src: WriteBackSrc::PC,
    mem_read: false,
}),
...
```

- `reg_write: true` - we write to a register (the `RA` register specifically)
- `branch_jump: BranchJump::Jal` - we're jumping
- `alu_src_a: ALUSrcA::PC` - the first operand of the ALU is the program counter (we calculate the jump target by adding the jump offset to the program counter)
- `alu_src_b: ALUSrcB::Immediate` - the second operand of the ALU is the immediate value in the instruction (the jump offset)
- `alu_op: ALUOp::ADD` - we're adding the program counter and the jump offset
- `mem_write: false` - we're not writing to memory
- `wb_src: WriteBackSrc::PC` - we're writing the program counter + 4 to a register
- `mem_read: false` - we're not reading from memory

Now, `jalr`:

```
...
// jalr
0b110_0111 => Ok(ControlSignals {
    reg_write: true,
    branch_jump: BranchJump::Jal,
    alu_src_a: ALUSrcA::Register,
    alu_src_b: ALUSrcB::Immediate,
    alu_op: ALUOp::ADD,
    mem_write: false,
    wb_src: WriteBackSrc::PC,
    mem_read: false,
}),
...
```

- `reg_write: true` - we write to a register (the `RA` register specifically)

- `branch_jump: BranchJump::Jal` - we're jumping
- `alu_src_a: ALUSrcA::Register` - the first operand of the ALU is a register (the base register)
- `alu_src_b: ALUSrcB::Immediate` - the second operand of the ALU is the immediate value in the instruction (the jump offset)
- `alu_op: ALUOp::ADD` - we're adding the base register and the jump offset
- `mem_write: false` - we're not writing to memory
- `wb_src: WriteBackSrc::PC` - we're writing the program counter + 4 to a register
- `mem_read: false` - we're not reading from memory

Control Signal Definitions

The `ControlSignals` struct is a simple struct that contains all the control signals that the Control Unit generates for an instruction.

```rust
/// a struct that holds the control signals that the Control Unit generates.
///
/// A decent chunk of these are actually entirely unnessary for this
implementation, but are included nonetheless for completeness.
pub struct ControlSignals {
    /// tells the register file to write to the register specified by the
instruction.
    pub reg_write: bool,
    /// The BranchJump signal is a 2 bit signal that tells the Branching and
Jump Unit what type of branching to consider.
    pub branch_jump: BranchJump,
    /// The ALUSrcA signal is a 1 bit signal that tells the ALU whether to
use the register value (0), the PC (1), or the constant 0 as the second
operand.
    pub alu_src_a: ALUSrcA,
    /// The ALUSrcB signal is a 1 bit signal that tells the ALU whether to
use the register value (0), the immediate value (1), or the constant 4 as
the second operand.
    pub alu_src_b: ALUSrcB,
    /// The ALU operation signal is a 2 bit signal that tells the ALU
Control Unit what type of instruction is being executed.
    pub alu_op: ALUOp,
    /// The mem_write signal is a 1 bit signal that tells the data memory
unit whether to write to memory.
    pub mem_write: bool,
    /// controls what source the write back stages uses.
    pub wb_src: WriteBackSrc,
    /// The mem_read signal is a 1 bit signal that tells the data memory
unit whether to read from memory.
    pub mem_read: bool,
}
```

The other control signals are defined as `u8` backed enums, implemented as follows:

```rust
#[repr(u8)]
/// a 2 bit signal that tells the ALU Control Unit what type of instruction is being executed
pub enum ALUOp {
    /// The ALU should perform an ADD operation, this is the case for memory load and store instructions.
    #[default]
    ADD = 0b00,
    /// The ALU should perform an operation specified by the funct3 field of the instruction (which specifies the type of branching to perform).
    /// This is the case for SB-type instructions.
    BRANCH = 0b01,
    /// The ALU should perform an operation specified by the funct7 and funct3 fields of the instruction.
    /// This is the case for R-type and I-type instructions.
    FUNCT = 0b10,
}

#[repr(u8)]
/// a 1 bit signal that tells the ALU whether to use the register value (0), the PC (1), or the constant 0 as the second operand.
pub enum ALUSrcA {
    #[default]
    Register = 0,
    PC = 1,
    Constant0 = 2,
}

#[repr(u8)]
/// a 1 bit signal that tells the ALU whether to use the register value (0), the immediate value (1), or the constant 4 as the second operand.
pub enum ALUSrcB {
    #[default]
    Register = 0,
    Immediate = 1,
    Constant4 = 2,
}

#[repr(u8)]
/// a 4 bit signal that tells the ALU what operation to perform.
pub enum ALUControl {
    AND = 0b0000,
    OR = 0b0001,
    #[default]
    ADD = 0b0010,
    SLL = 0b0011,
```

```
    SLT = 0b0100,
    SLTU = 0b0101,
    SUB = 0b0110,
    XOR = 0b0111,
    SRL = 0b1000,
    SRA = 0b1010,
}

#[repr(u8)]
/// a 2 bit control signal that tells the Branching and Jump Unit what type
of branching to consider.
pub enum BranchJump {
    #[default]
    No = 0b00,
    Branch = 0b01,
    Jal = 0b10,
}

#[repr(u8)]
/// a 2 bit control signal that tells the WB stage what source to write back
to the register file.
pub enum WriteBackSrc {
    #[default]
    NA = 0b00,
    ALU = 0b01,
    Mem = 0b10,
    PC = 0b11,
}
```

`PCSrc` Enum

The `PCSrc` enum is a simple enum that represents the different sources the program counter can be set to.

```rust
/// a signal that specifies where the next PC should come from.
pub enum PCSrc {
    /// Go to the first instruction in the program (initial PC value)
    Init,
    #[default]
    /// The next PC value comes from PC + 4
    Next,
    /// The next PC value comes from the branch target address,
    BranchTarget { offset: i32 },
    /// The next PC value comes from the jump target address
    JumpTarget { target: u32 },
    /// program has ended
    End,
}
```

it provides a method to get the next program counter value based on the current program counter value.

```rust
impl PCSrc {
    /// Calculate the next PC value based on the current PC value.
    ///
    /// # Arguments
    ///
    /// * `pc` - the current program counter value
    ///
    /// # Returns
    ///
    /// * `u32` - the next program counter value
    #[must_use]
    pub const fn next(&self, pc: u32) -> u32 {
        match self {
            Self::Init => 0,
            Self::Next => pc + 4,
            Self::BranchTarget { offset } =>
pc.wrapping_add_signed(*offset),
            Self::JumpTarget { target } => *target,
            Self::End => pc,
        }
    }
}
```

Stages module

The Stages module contains the definition of the stage registers used by the CPU, as well as the implementation of the `StageRegisters` struct.
It also contains the `Immediate` enum, used to represent the different types of immediate values in RISC-V instructions.

`StageRegisters` struct

The `StageRegisters` struct is a simple struct that contains the stage registers used by the CPU.

```
/// a struct that holds the values of the pipeline stage registers.
pub struct StageRegisters {
    pub ifid: IfId,
    pub idex: IdEx,
    pub exmem: ExMem,
    pub memwb: MemWb,
    pub wb_stage: Wb,
}
```

`Immediate` enum

The `Immediate` enum is a simple enum that represents the different types of immediate values in RISC-V instructions.

```
/// An enum that represents the different types of immediate values in RISC-
V instructions.
pub enum Immediate {
    /// for I-type and S-type instructions
    SignedImmediate(i32),
    /// for U-type instructions
    UpperImmediate(u32),
    /// for SB-type instructions
    BranchOffset(i32),
    /// for UJ-type instructions
    JumpOffset(i32),
    /// for all other instructions
    None,
}
```

IfId, IdEx, ExMem, MemWb, and Wb Enums

These enums represent the different stages of the pipeline, and contain the intermediate values that are passed between stages.

```rust
/// The IF/ID pipeline stage register.
pub enum IfId {
    /// the values that are passed from the IF stage to the ID stage.
    If {
        /// the machine code of the instruction that was fetched.
        instruction_code: u32,
        /// the program counter value of the instruction.
        pc: u32,
    },
    #[default]
    /// used to flush the pipeline.
    Flush,
}

/// The ID/EX pipeline stage register.
pub enum IdEx {
    /// the values that are passed from the ID stage to the EX stage.
    Id {
        instruction: Instruction,
        rs1: Option<RegisterMapping>,
        read_data_1: Option<u32>,
        rs2: Option<RegisterMapping>,
        read_data_2: Option<u32>,
        immediate: Immediate,
        /// the program counter value of the instruction.
        pc: u32,
        control_signals: ControlSignals,
    },
    #[default]
    /// used to flush the pipeline.
    Flush,
    /// used to indicate a stall in the pipeline.
    Stall,
}

/// The EX/MEM pipeline stage register.
pub enum ExMem {
    /// the values that are passed from the EX stage to the MEM stage.
    Ex {
        instruction: Instruction,
        alu_result: u32,
        /// This variable will be updated by Execute() function and used
when deciding to use branch target address in the next cycle.
        /// The zero variable will be set to 1 by ALU when the computation
result is zero and unset to 0 if otherwise.
        alu_zero: bool,
```

```
        read_data_2: Option<u32>,
        /// the program counter value of the instruction.
        pc: u32,
        /// the next program counter value.
        pc_src: PCSrc,
        control_signals: ControlSignals,
    },
    #[default]
    /// used to flush the pipeline.
    Flush,
}

/// The MEM/WB pipeline stage register.
pub enum MemWb {
    /// the values that are passed from the MEM stage to the WB stage.
    Mem {
        instruction: Instruction,
        mem_read_data: Option<u32>,
        alu_result: u32,
        /// the program counter value of the instruction.
        pc: u32,
        control_signals: ControlSignals,
    },
    #[default]
    /// used to flush the pipeline.
    Flush,
}

/// used to store the value written to the register file in the WB stage, if
any, for data forwarding
///
/// since we execute stages backwards, if we want to forward data from the
MEM/WB stage to the ID/EX stage,
/// we need to store the value written to the register file in the WB stage.
/// Because otherwise, the value will be overwritten before we can forward
it.
pub enum Wb {
    /// information needed by the forwarding unit
    Mem {
        instruction: Instruction,
        wb_data: Option<u32>,
        control_signals: ControlSignals,
    },
    #[default]
    /// used to flush the pipeline.
```

```
    Flush,
}
```

Hazard Detection module

The Hazard Detection module contains the implementation of the data forwarding and hazard detection units used by the CPU to prevent data hazards.

The `ForwardA` and `ForwardB` Enums

These are 2 bit signals that the forwarding unit uses to tell the caller where to forward data from.

```
/// a 2 bit signal that tells the forwarding unit what to forward to the
ID/EX stage.
pub enum ForwardA {
    #[default]
    None = 0b00,
    ExMem = 0b10,
    MemWb = 0b01,
}

/// a 2 bit signal that tells the forwarding unit what to forward to the
ID/EX stage.
pub enum ForwardB {
    #[default]
    None = 0b00,
    ExMem = 0b10,
    MemWb = 0b01,
}
```

`forwarding_unit` function

The `forwarding_unit` function is the implementation of the data forwarding unit, it determines if data forwarding is needed for either of the operands of the ALU and specifies where the forwarded data should come from.

```rust
/// The forwarding unit determines whether to forward data from the EX/MEM
and/or MEM/WB stages to the ID/EX stage.
///
/// # Arguments
///
/// * `exmem` - the values in the EX/MEM pipeline stage register.
/// * `wb` - the values in the MEM/WB pipeline stage register.
/// * `idex` - the values in the ID/EX pipeline stage register.
///
/// # Returns
///
/// * `ForwardA` - the forwarding decision for source register 1.
/// * `ForwardB` - the forwarding decision for source register 2.
pub fn forwarding_unit(exmem: ExMem, wb: Wb, idex: IdEx) -> (ForwardA,
ForwardB) {
    // Initialize forwarding variables
    let mut forward_a = ForwardA::None;
    let mut forward_b = ForwardB::None;

    // Extract source registers from IDEX stage
    let (idex_source_reg1, idex_source_reg2) = match idex {
        /* omitted */
    };

    // Extract register write and destination register from EXMEM stage
    let (exmem_regwrite, exmem_dest_reg) = match exmem {
        /* omitted */
    };

    // Extract register write and destination register from MEMWB stage
    let (memwb_regwrite, memwb_dest_reg) = match wb {
        /* omitted */
    };

    // Determine forwarding for source register 1
    match idex_source_reg1 {
        None | Some(RegisterMapping::Zero) => (),
        Some(rs1) if exmem_regwrite && exmem_dest_reg == rs1 => forward_a =
ForwardA::ExMem,
        Some(rs1) if memwb_regwrite && memwb_dest_reg == rs1 => forward_a =
ForwardA::MemWb,
        _ => (),
    }

    // Determine forwarding for source register 2
    match idex_source_reg2 {
```

```
        None | Some(RegisterMapping::Zero) => (),
        Some(rs2) if exmem_regwrite && exmem_dest_reg == rs2 => forward_b =
ForwardB::ExMem,
        Some(rs2) if memwb_regwrite && memwb_dest_reg == rs2 => forward_b =
ForwardB::MemWb,
        _ => (),
    }

    // Return forwarding decisions
    (forward_a, forward_b)
}
```

This handles rtype data hazards, and in combination with the hazard detection unit, handles load-use data hazards as well assuming the output of the hazard detection unit is used correctly.

the `HazardDetectionUnit` struct

The `HazardDetectionUnit` struct is a simple struct that contains the information that the hazard detection unit needs to determine if the conditions for a stall are met (load-use hazards).

The only reason that the forwarding unit isn't implemented this way is because this was done later in the project and I didn't feel like doing the refactor.

```
/// The hazard detection unit determines whether there is a data hazard
between the ID and EX stages that requires stalling (e.g. load-use hazards)
/// (the forwarding unit handles rtype data hazards, and can handle load
hazards if a stall was performed)
pub struct HazardDetectionUnit {
    /// the source register 1 from the IF/ID stage
    ifid_rs1: Option<RegisterMapping>,
    /// the source register 2 from the IF/ID stage
    ifid_rs2: Option<RegisterMapping>,
    /// the destination register from the ID/EX stage
    idex_rd: Option<RegisterMapping>,
    /// a boolean indicating whether the instruction in the ID/EX stage
writes to memory
    idex_memread: bool,
}
```

This struct provides two methods, one to "prime" it with the information it needs, and another that determines if a stall condition is needed.

Note how (due to the private fields), the `prime` function is the only way to create a new `HazardDetectionUnit` instance. Note also that the `detect_stall_conditions` function takes `self` by value and that `HazardDetectionUnit` doesn't implement `Copy` or `Clone`. Together, all this means that you can't reuse a `HazardDetectionUnit` instance after calling `detect_stall_conditions` on it, and must create a new instance each time.

```rust
impl HazardDetectionUnit {
    /// prime the hazard detection unit with the relevant current pipeline
state
    pub const fn prime(decoded_instruction: Instruction, idex_reg: IdEx) ->
Self {
        let ifid_rs1 = decoded_instruction.rs1();
        let ifid_rs2 = decoded_instruction.rs2();

        let idex_rd = match idex_reg {
            IdEx::Id { instruction, .. } => instruction.rd(),
            _ => None,
        };

        let idex_memread = match idex_reg {
            IdEx::Id {
                control_signals, ..
            } => control_signals.mem_read,
            _ => false,
        };

        Self {
            ifid_rs1,
            ifid_rs2,
            idex_rd,
            idex_memread,
        }
    }

    /// Detect whether a stall is required to resolve a data hazard
    pub fn detect_stall_conditions(self) -> bool {
        // check for a hazard with rs1
        let rs1_hazard = match (self.ifid_rs1, self.idex_rd,
self.idex_memread) {
            // hazard in the ID/EX stage
            (Some(rs1), Some(rd), true) if rs1 == rd => true,
            _ => false,
        };

        // check for a hazard with rs2
        let rs2_hazard = match (self.ifid_rs2, self.idex_rd,
self.idex_memread) {
            // hazard in the ID/EX stage
            (Some(rs2), Some(rd), true) if rs2 == rd => true,
            _ => false,
        };
```

```
        // return whether a stall is required
        rs1_hazard || rs2_hazard
    }
}
```

Utils module

Nothing here is really worth discussing in detail as it's not directly related to the emulator logic. If you're interested in what's in this module you can check the source code. Just like everything else, it's well documented.

Execution of Sample Programs, and other testing

Testing

Unit Testing

The project contains unit tests for the various modules, ensuring that core functionality is implemented correctly.

ALU Module

The `ALU` module (`alu.rs`) contains unit tests ensuring that the ALU correctly performs the various arithmetic operations required by the RISC-V ISA.

```
$ cargo test "alu::tests"

running 10 tests
test alu::tests::test_alu_add ... ok
test alu::tests::test_alu_or ... ok
test alu::tests::test_alu_and ... ok
test alu::tests::test_alu_sll ... ok
test alu::tests::test_alu_slt ... ok
test alu::tests::test_alu_sltu ... ok
test alu::tests::test_alu_sra ... ok
test alu::tests::test_alu_srl ... ok
test alu::tests::test_alu_sub ... ok
test alu::tests::test_alu_xor ... ok

test result: ok. 10 passed; 0 failed; 0 ignored; 0 measured; 22 filtered
out; finished in 0.00s
```

CPU Module

The `CPU` module (`cpu.rs`) contains unit tests ensuring that each pipeline stage executes correctly, and that the CPU properly handles data hazards.

```
$ cargo test "cpu::tests"

running 9 tests
test cpu::tests::test_cpu_decode ... ok
test cpu::tests::test_cpu_execute ... ok
test cpu::tests::test_cpu_fetch ... ok
test cpu::tests::test_cpu_new ... ok
test cpu::tests::test_cpu_mem ... ok
test cpu::tests::test_cpu_initialize_rf ... ok
test cpu::tests::test_cpu_write_back ... ok
test cpu::tests::test_data_load_hazard ... ok
test cpu::tests::test_data_rtype_hazard ... ok

test result: ok. 9 passed; 0 failed; 0 ignored; 0 measured; 23 filtered out;
finished in 0.00s
```

Instruction Module

The `Instruction` module (`instruction.rs`) contains unit tests ensuring that the `Instruction::from_machine_code` function can correctly decode various instructions.

```
$ cargo test "instruction::tests"

running 9 tests
test instruction::tests::test_add ... ok
test instruction::tests::test_andi ... ok
test instruction::tests::test_auipc ... ok
test instruction::tests::test_bne ... ok
test instruction::tests::test_jal ... ok
test instruction::tests::test_jal_2 ... ok
test instruction::tests::test_lui ... ok
test instruction::tests::test_lw ... ok
test instruction::tests::test_sb ... ok

test result: ok. 9 passed; 0 failed; 0 ignored; 0 measured; 23 filtered out;
finished in 0.00s
```

Integration Testing

Integration tests in `lib.rs` ensure that the CPU correctly executes the provided sample programs.

```
$ cargo test "tests::test_sample"

running 2 tests
test tests::test_sample_1 ... ok
test tests::test_sample_2 ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 30 filtered out;
finished in 0.00s
```

Sample Program 1

Sample Program 1: Execution

```
$ cargo run

Enter the name of the file name to run:

sample_part1.txt

total_clock_cycles 1 :


total_clock_cycles 2 :
pc is modified to 0x4


total_clock_cycles 3 :
pc is modified to 0x8


total_clock_cycles 4 :
pc is modified to 0xc


total_clock_cycles 5 :
x3 is modified to 0x10
pc is modified to 0x10


total_clock_cycles 6 :
x5 is modified to 0x1b
pc is modified to 0x14


total_clock_cycles 7 :


total_clock_cycles 8 :
x5 is modified to 0x2b


total_clock_cycles 9 :
memory 0x70 is modified to 0x2f
x5 is modified to 0x2f
```

```
total_clock_cycles 10 :

program terminated:
total execution time is 10 cycles
```

Sample Program 1: Pipeline Table

For reference, here is the pipeline table for sample program 1:

cycle	IF	ID	EX	MEM	WB		PC
1	I1						0
2	I2	I1					4
3	I3	I2	I1				8
4	I4	I3	I2	I1			12
5	I5	I4	I3	I2	I1		16
6	I6	I5	I4	I3	I2		20
7		I6	I5	I4	I3		
8			I6	I5	I4		
9				I6	I5		
10					I6		

Sample Program 2

Sample Program 2: Execution

```
$ cargo run

Enter the name of the file name to run:

sample_part2.txt

total_clock_cycles 1 :


total_clock_cycles 2 :
pc is modified to 0x4


total_clock_cycles 3 :
pc is modified to 0x8


total_clock_cycles 4 :
pipeline flushed


total_clock_cycles 5 :
x1 is modified to 0x4
pc is modified to 0x8


total_clock_cycles 6 :
pc is modified to 0xc


total_clock_cycles 7 :
pc is modified to 0x10


total_clock_cycles 8 :
pc is modified to 0x14


total_clock_cycles 9 :
x10 is modified to 0xc
```

```
total_clock_cycles 10 :
x30 is modified to 0x3
pipeline flushed


total_clock_cycles 11 :
x1 is modified to 0x14
pc is modified to 0x4


total_clock_cycles 12 :
pc is modified to 0x8


total_clock_cycles 13 :
pc is modified to 0xc


total_clock_cycles 14 :
pipeline flushed


total_clock_cycles 15 :
x1 is modified to 0x8
pc is modified to 0x14


total_clock_cycles 16 :


total_clock_cycles 17 :


total_clock_cycles 18 :
memory 0x20 is modified to 0x3


total_clock_cycles 19 :

program terminated:
total execution time is 19 cycles
```

Sample Program 2: Pipeline Table

For reference, here is the pipeline table for sample program 2:

cycle	IF	ID	EX	MEM	WB		PC	
1	I1						0	
2	I2	I1					4	
3	I3	I2	I1				8	
4	..	..	..	I1				I1 jumps pc to 8
5	I3	..	..	..	I1		8	
6	I4	I3	..	..	..		12	
7	I5	I4	I3	..	..		16	
8	I6	I5	I4	I3	..		20	
9		I6	I5	I4	I3			
10	..	..	..	I5	I4			I5 jumps pc to 4
11	I2	..	..	..	I5		4	
12	I3	I2	..	..	..		8	
13	I4	I3	I2	..	..		12	
14	..	..	..	I2	..			I2 jumps pc to 20
15	I6	..	..	..	I2		20	
16		I6	..	..	..			
17			I6	..	..			
18				I6	..			
19					I6			