One goal of this assignment is to expose you to architecture emulation in C++. Throughout the course, we will use toy and state-of-the-art simulation tools, all of which are C++ based. Using these tools is a standard step in hardware design in both academic and industry settings, and doing so will encourage you to think about the path that data takes through the processor. In addition, this assignment encourages you to reason about the processor design concerning the partitioning of the data path into pipeline stages, reason about how to detect pipeline hazards, and implement strategies to resolve these hazards.

Submission

You are asked to submit the programming component and the write up to Gradescope. The programming component has an autograder with test cases to help facilitate your development process, but not all graded test cases are given. You are also expected to test your code independently!

Each section of the assignment has a written component, which you should include in a single "Written Response" document. The purpose of these exercises is to encourage you to engage with the nuance of the programming task, so respond in as much detail as you feel is necessary. As a guiding principle, a strong response can be made in about one paragraph.

Collaboration Policy

Collaboration is highly encouraged on this assignment! Be sure that all submitted work is your own, but you are encouraged to ask for help from others during lab times and out of class.

Getting Started

This assignment is designed assuming that you can run it in the course devenv that you configured during lab 1. Please come by office hours if you have any issues getting the environment set up.

Download the source for this assignment to get started! The home of the repository has several subdirectories In the repository, you will find a `Makefile` to help you build the various sources for each part of this assignment.

1 Introducing the VBISC Instruction Set

In this assignment, you will be constructing a processor emulator based on the Very Basic Instruction Set Computer (VBISC) instruction set[1]. The full instruction set is enumerated in Table 1. Examine the starter code. Unlike commodity assembly files, VBISC assembly does not expect any preamble and exits with a simple `end` instruction. However, VBISC assembly is very particular about its formatting: whitespace between tokens is exactly one space, register declarations are all lowercase, commas must be placed exactly where they are defined, and instructions are separated by a newline (`\n`) character.

operation	Mnemonic	Format	Description
`ldi`	load immediate	`ldi` DEST, IMM	Place immediate value from instruction bytes in destination register.
`add`	add	`add` DEST, RS1, RS2	Add the value in the register specified at source 1 with the value at source 2 and store result in destination.
`sub`	subtract	`sub` DEST, RS1, RS2	Subtract the value in the register specified at source 2 from the value at source 1 and store result in destination.
`mul`	multiply	`mul` DEST, RS1, RS2	Multiply the value in the register at source 1 with the value at source 2 and store result in destination.
`div`	divide	`div` DEST, RS1, RS2	Multiply the value in the register at source 1 by the value at source 2 and store result in destination.
`mod`	modulo	`mod` DEST, RS1, RS2	Take the remainder of the value in the register at source 1 divided by the value at source 2 and store result in destination.
`end`	end	`end`	The program has finished. Exit.

Table 1: The complete VBISC instruction set.

This assignment asks you to build several processor emulators that implements the VBISC ISA. Therefore, the first part of this assignment encourages you to get comfortable with VBISC programming to build an intuition for the instruction set. In doing so, you are exploring the types of programs that a developer would be able to write (or more likely, that a compiler would produce) for this processor.

You'll find a repository called `test-progs`, in which there are three sample assembly programs that all use the VBISC instruction set.

Your Task

You are asked to complete several tasks surrounding the VBISC instruction set by way of programming in VBISC assembly and reasoning about their outputs.

Programming Suppose your processor has 8 registers (`r0`, `r1`, `r2`, ...). For each part, write a plain-text[2] VBISC assembly program that solves the tasks.

[1] This is a reduced version of the RISC-V ISA for this assignment!

[2] If you use a rich text editor IDE, sometimes it adds unexpected characters which will reject the file.

1. In a file called `sol0`, write a program that uses exactly 8 instructions to ensure that each register `r<n>` has the data n in it. The only immediates you are allowed to use must be even!

2. In a file called `sol1`, write a program that uses exactly 12 instructions to count up from 1 to 10 in `r1`.

3. In a file called `sol2`, write a program that loads if 181 is a multiple of 17. If it is, the program should exit with a zero in `r1`. If it is not, the program should exit with a non-zero number in `r1`. If the program is more than 25 instructions, it will be rejected.

Testing. Test your assembly program by using it as input to the `ask-vbisc-oracle.py solutions/sol<n>`. The program will dump the register state after each instruction.

Submission Guidelines. Submit your solution files to Gradescope as part of this repository. The autograder will look for the `solutions` subdirectory and copy files `sol0`, `sol1` and `sol2` into the autograding infrastructure. Be sure your submission matches this expected format!

The rest of the assignment is oriented towards constructing the hardware for your emulated processor. The processor only needs to be functional for the instructions in the instruction set, so think about which instructions from RISC-V are omitted from this ISA and how that will influence the construction of the processor. You may find it helpful to draw a modified version of our RISC-V processor diagram for reference. Which components remain? What needs to change?

2 Implementing a Decoder

The ISA serves as the interface through which the software and hardware can interact. Instructions in the ISA define the expected hardware procedures for to produce the desired output from the provided sources. To deduce what these sources are and how they should interact, the processor hardware depends on a special component called a *decoder*. The decoder takes an instruction as input and parses the bits to produce these fields.

The decoder is very closely related to the ISA. For each instruction in the ISA definition, the decoder must have logic to set the requisite control fields for the processor to implement the logic expected by the software. As such, in many ways the decoder serves as the closest hardware component to the hardware-software interface.

In real hardware, instructions are encoded into executable byte codes by the compiler. At a high level, executing the program implies moving program counter in the processor to the address of the beginning of that program's `main` function, and fetching and decoding the byte code from there. This additional level of encoding is beyond the scope of this assignment. Instead, instructions will be encoded in instruction memory as strings of assembly instructions, and the decoder emulator will parse those strings as strings.

Your Task

Now that you've worked with interface exposed to the software, in this section you will develop the first emulated component of the hardware to interface with that software by way of the decoder. To do this, you will model the bit parsing procedure as string parsing. Before you begin, you may find it useful to see the implementation of the emulated `InstructionMemory` component, which is located at `src/components/instruction_memory.hh`. Notice that components in this assignment are going to model hardware components as C++ classes, defined in header files.

Programming Complete the `decode` function in the `src/components/decoder.hh` file. To be successful, this function should implement the behavior specified in its preamble comment. You may assume that the calling function is going to provide an instruction string that is produced by a call to `getInstruction(int)` from instruction memory, and that it will provide placeholder arguments for the rest of the arguments. If the instruction is unrecognized by your emulated `Decoder`, the `decode` function should use the `panic` function from the `src/util.hh` to exit the simulation with the message "`Unimplemented instruction!`"

Instead of returning a long tuple, the output of your `decode` function should set the `operation` field with one of the operations from Table 1; `destination` should be set to the destination register address of the instruction; `source_register1` and `source_register2` should be set to the instructions sources, if they are registers; and `imm` should be set to the integer immediate function. If the interpreted instruction does not use all of the fields provided in the input, then your `decode` function should provide a `NULL` equivalent to the that reference[3].

Testing. In the `src` directory, you will find a file called `decoder_tester.cc`. It imports your `Decoder` class and tests its functionality for various instructions. Be sure to add more test cases to verify its functionality! You can do so by calling the `decode` function, and authenticating that the returned values match what you would expect given the inputs.

[3]What is a "reference"? In C++, a variable may be declared as a pointer (`*`) like in C, or as a reference (`&`). A reference acts as an alias to the declared variable, which means you do not need to "de-reference" it to set it (i.e., instead of `void fn(int *x) { *x = 10; }`, use `void fn(int &x) { x = 10; }`). See this answer for further reading!

Build the tester executable by calling `make decoder_tester`, which writes a binary to `bin/decoder_tester`. You can run this binary to verify the behavior!

Submission Guidelines. When you have finished this section, you can upload your repository to Gradescope. The autograding infrastructure uses a modified/extended version of the `decode_tester` to evaluate your implementation for this part.

Hints and suggestions. You may want to look at the `std::string` library for some help functions to decode instruction fields. In particular, you may find the `find` and `substr` functions useful. With that said, there are several legitimate ways to implement your Decoder component, so use the method that is most intuitive to you!

You may also find it helpful to use `gdb` to inspect the intermediate state of your Decoder.

Written Response In a section of your written response document clearly marked "Decoder Response", respond to the following prompts.

1. How are instructions encoded in your processor emulator? What are the strengths and weaknesses of this approach in terms of modeling real processors?

2. Think about how your Decoder component will be executed as a member of the processor emulator. Compare and contrast a decoder's execution in hardware from the emulated decoder.

3 Building a Single Cycle Processor

In this section, you are going to increase the hardware robustness of the hardware-software interface such that instructions may be meaningfully executed. You can think of a processor as a set of well-defined instructions, the collection of components to execute these instructions, and the logic that coordinates the flow of information through these components. Seeing as the ISA and interface between the ISA and hardware have been defined, all that remains are various strategies to coordinate this flow of information and exploring their trade-offs.

One scheme to coordinate the flow of information through components would be to handle one instruction per cycle. That is, each clock cycle must be configured such that the whole of an instructions execution happens within that cycle. To do so, the instruction must be fetched from `InstructionMemory` so that it may be decoded by the `Decoder`. This provides sufficient information to fetch source data from the `RegisterFile` and feed it as input to the `ALU` to compute the output and write the result back to the `RegisterFile`.

Your Task

This part of the assignment asks you to construct a single cycle processor that integrates all of the provided components. The starter scaffolding for this portion can be found at `src/processors/single_cycle.hh`. Notice that the `SingleCycleProcessor` class has attributes for `InstructionMemory`, `Decoder`, `RegisterFile`, and `ALU` components. The `Decoder` component refers to the class declaration from the previous section whereas the `InstructionMemory`, `RegisterFile`, and `ALU` classes are provided to you in the `src/components` directory. Take a look at the definitions of these other components to get a sense of how the key functions may be used, as they will be relevant to your implementation. You will not need to modify the definitions of these components at any point during this assignment!

To run a program in the `SingleCycleProcessor`, the path to the assembly is passed to the constructor (which is in turn passed to the `InstructionMemory` class for storage). The constructing function can then run that program by calling `SingleCycleProcessor::executeProgram()`. In this function, you will find the main execution loop. You can think of each cycle through the loop as the execution of a clock cycle. *Do not modify the code after the central loop* of this function – doing so may result in failing the autograder!

Programming Complete the `executeProgram()` function in the `SingleCycleProcessor` class declared in the `src/processors/single_cycle.hh`. On each clock cycle, your processor emulator should ① retrieve an instruction from instruction memory, ② decode it into interpretable signals, ③ send those signals to the appropriate execution components, ④ update the processor state as appropriate, and ⑤ cleanly exit when the processor reaches the end of the program. You may add attributes to the `SingleCycleProcessor` class, but you should not add parameters to the constructor.

Testing. You can test your processor by compiling the `SingleCycleProcessor` with the command `make single_cycle` and executing the sample VBISC programs in `test-progs` by running `./bin/single-cycle`. For each of the sample programs, consider the number of instructions and the instructions executed to reason about the final register state. This will help you determine whether your processor executed correctly.

Submission Guidelines. Much like in the previous section, an extended version of the `single_cycle.cc` is used to test your implementation.

Hints and suggestions. Think about how the components in the five-stage RISC-V processor interact with one another, and map the changes from RISC-V onto VBISC. The interaction

between components will be similar. Some (but not all) elements of the RISC-V processor that help manage the flow of information during execution that may similarly be useful in your processor emulator.

It may help to think of your implementation in this step as assembling and connecting various components. For example, consider how the output of `InstructionMemory` acts as the input to the `Decoder`. Your implementation of the interaction between these components will emulate the wired connection between these components; the "interaction" is small but important!

Written Response In a section of your written response document clearly marked "Single Cycle Response", respond to the following prompts.

1. Describe any additional components that you added to the `SingleCycleProcessor` class and how they are utilized/modified during execution. If you did not add any elements, provide a pseudocode description of your emulated processor.

2. Assume the following latencies for each component: Instruction Memory, 250ps; Register File, 150ps; Mux, 25ps; ALU, 150ps; Adder 100ps. What is the execution latency for `test-progs/test2` in your processor? Describe your methodology in your answer.

🥳 **Congratulations, you've built an emulated processor!** 🥳

4 Constructing a Simple Pipeline

This part of the assignment asks you to build instruction-level parallelism into your emulated processor with pipelining. In doing so, you will explore the positive and negative consequences of constructing such a pipeline.

The processor design in this section will be driven by a four-stage processor, where the stages are Fetch, Decode, Execute, and Writeback. The task performed by the Fetch stage should implement a fetch from instruction memory to pass to the Decode stage. The Decode stage interprets the fetched instruction to produce an operation and fetches the data required to process during the Execute stage from the appropriate sources. The Execute stage performs the operation specified by the Decode stage, and notifies the Writeback stage of the data and its destination. The Writeback stage sets the data to its destination.

In the Single Cycle processor, you developed a processor as a sort of "collection of components", where the processor class served as the wrapper for and logic connecting them together. When thinking of a pipelined processor, it may help to think of the processor as a "collection of components and stages", where the processor still remains as the wrapper and logic for connecting them together. Remember, unlike in a single cycle processor, pipelined processors use *pipeline registers* to coordinate the logic and flow of information between stages. These are accessed by the pipeline stages as inputs and outputs from their execution, and the processor emulator should access them accordingly!

4.1 Implementation Details

To emulate this pipeline, each stage is implemented as its own C++ class. The main execution of the class is the `tick` function. The `tick` function models the behavior that the stage should execute within the clock cycle.

Notice that this function returns a boolean value. This value is interpreted by the main loop of the `executeProgram` function in the `src/processors/simple_pipeline.hh` to indicate if that pipeline stage is `drained`. "Draining" in the context of your processor emulator is a term that is often used in simulators to help facilitate the exit routine. Think about what happens when you explicitly tell your device to shut down. From the software perspective, an interrupt is raised by your processor that sends a signal to all processes, which allows running applications to reach their exit sequence. From a hardware perspective, this means continuing to execute until all instructions to execute have completely cleared the processor state. To model this effect in the processor emulator, a stage returns `true` from its `tick` after the stage has fully drained.

Your Task

Programming You will be implementing the four-stage pipeline described above. In doing so, you will complete the `tick` functions in each of the pipeline stages declared in `src/stages` and each of the pipeline registers in `src/pipeline_registers`. Notice that the appropriate pipeline registers are members of the pipeline class as members `input_reg` and `output_reg`.

In addition to adding the pipeline logic to your pipelined processor emulator, it is your job to ensure that the emulator exits cleanly and appropriately. This entails setting the "drained" flag in each of the `Decode`, `Execute`, and `Writeback` stages as appropriate. If done incorrectly, not all instructions from the source program will execute and your register file will be in the incorrect state at the conclusion of the program.

Note, draining is set in the `Fetch` stage by the main loop of the `SimplePipelineProcessor` after the `Decode` stage drains. The `SimplePipelineProcessor` is declared in

`src/processors/simple_pipeline.hh` and is provided to you as part of the starter scaffolding. Your solution should work *without* editing this file!

Testing. Like with the single cycle processor, test your emulated pipelined processor by executing the sample VBISC programs in the `test-progs` repository by using the compile command `make simple_pipeline` running `./bin/simple_pipeline`. What do you notice about the output from this processor compared to the single cycle processor output? Try to reason about what the correct behavior should be for this processor, and use the test cases on Gradescope to confirm what you expect to happen.

Submission Guidelines. When you submit your repository to Gradescope, an extended version of the `simple_pipeline.cc` is used to test your implementation.

Hints and suggestions. You may find it helpful to start with your single cycle processor and thinking about what logic is associated with which pipeline stage. To determine which fields are necessary execute each stage, think about the arguments to the main function of the key components and how those may be retrieved from previous stages. A pipeline diagram may be helpful in determining these fields.

Notice that the pipeline register classes essentially serve as accessor structs. The relevant data for each stage should be communicated via these registers and accessed from the `input_reg` before the result is set to the `output_reg`.

Written Response In a section of your written response document clearly marked "Simple Pipeline Response", respond to the following prompts.

1. The four-stage simple pipeline describes one mechanism to pipeline the single cycle processor. Describe an alternative pipeline that could have been derived. What would an advantage and disadvantage of this pipeline be?

2. Look at the main loop of the `SimplePipelineProccessor::executeProgram()` function. Would the output change if you changed the order in which the order of the stages `tick` functions were called? Why or why not?

3. Consider `test-progs/test1`. Identify the hazards that occur in its execution and the hazard type. Are any of these hazards are consistent in `test-progs/test0`? Justify your answer in +/- one sentence.

5 Deploying a Hazard Checking Unit

When you constructed your pipelined in the previous section, you will have noticed that the final `dumpRegister` call will have produced an incorrect output. This a function of the lack of *hazard* mitigation. Without care, a standard pipelined processor will suffer from inter-instruction dependencies leading to unexpected outcomes.

In this section, you will implement logic to correct for these hazards by introducing a *stalling* mechanism to the pipeline stages. When a processor stalls, it introduces a *bubble* in the pipeline. The bubble allows the instruction causing the hazard to be flushed from the pipeline. After which, the instruction being handled by the stalled pipeline stage can continue executing with the correct fields.

The processor will be extended to include "hazard checking hardware" to ensure that instructions dependent on later instructions do not execute. To do so, the processor uses a `HazardCheckingUnit`. This component serves as a *monitoring component*, which is tasked with overseeing the execution rather than performing meaningful execution for any particular instruction. Because of this, the `HazardCheckingUnit` is managed by the processor logic explicitly rather than by any individual stage. Its main function is executed at the end of a cycle.

Your Task

In this section, you will implement the logic for a `HazardCheckingUnit` that ensures the running programs have a correct output in the four-stage pipeline from the previous section spite of any hazards that may exist (i.e., the `HazardCheckingProcessor`). Your job will be to then integrate `HazardCheckingUnit` into the processor's main loop to ensure that safe execution of the program.

Programming You will implement a hazard checking unit and the logic to integrate it into a processor. The starter scaffolding for the `HazardCheckingUnit` component is in `src/components/hazard_checking_unit.hh`. Your job is to complete this class. That entails adding the requisite attributes to the `HazardCheckingUnit` class, constructing these attributes as needed, and implementing the `bool operandDependence()` function.

After you implement the `HazardCheckingUnit` component, you will implement the `HazardCheckingProcessor` declared in `src/processors/hazard_checking.hh`. This processor will integrate the `HazardCheckingUnit` as a member and call the unit as needed. To do so, you will look to extend the main loop in the `executeProgram()` function to ensure that the appropriate behaviors are utilized to implement hazard checking.

Testing. Test your hazard checking processor by executing the sample VBISC programs in the `test-progs` repository by using the compile command `make hazard_checking` running `./bin/hazard_checking`. What do you notice about the output from this processor compared to the single cycle processor output? Try to reason about what the correct behavior should be for this processor, and use the test cases on Gradescope to confirm what you expect to happen.

Submission Guidelines. When you submit your repository to Gradescope, an extended version of the `simple_pipeline.cc` is used to test your implementation.

Hints and suggestions. You will notice that the `HazardCheckingUnit` starter scaffolding is relatively minimal. This means that you get creative license to implement the logic for this component however makes the most sense to you. The `HazardCheckingUnit` will only ever be called by the `HazardCheckingProcessor`, which it is also your job to implement. This means your `HazardCheckingUnit` constructor can include as few or as many fields as necessary to implement its function without needing to conform to particular style for the autograder.

Written Response In a section of your written response document clearly marked "Hazard Checking Pipeline Response", respond to the following prompts.

1. Your `HazardCheckingProcessor` implements a pipelined processor that can correctly produce output. Because of the pipeline, assume the cycle latency is half as long in the `HazardCheckingProcessor` as the `SingleCycleProcessor`. Are there instances in which the `HazardCheckingProcessor` performs faster than the `SingleCycleProcessor` and vice versa? You may find it helpful to justify your responses with VBISC assembly programs.
2. Speculate on the implications of including a monitoring component on performance. You may want to think about cycle latency and number of cycles executed in your response.

6 Forwarding to Optimize Hazards

While checking for hazards and appropriately bubbling will ensure that a program can execute correctly in a pipelined processor, it can lead to performance inefficiencies. For this reason, many processors implement *forwarding* between pipeline stages. If a processor's "forwarding hardware" determines that the output produced by a certain stage corresponds to the input from another stage, then the hardware can bypass the central logic through which data flows to the source as it is needed to avoid stalls.

This "forwarding hardware" is implemented in the processor as a monitoring unit in a similar manner to the `HazardCheckingUnit` that you implemented in the previous section. In each cycle, this hardware (a `ForwardingUnit`) checks the state produced by each pipeline stage. If any stage depends on this output as input for its operation in the next cycle, then the unit updates the appropriate source(s). In doing so, the `ForwardingUnit` can help the pipeline avoid stalls by preemptively alleviating the hazard. If the hazard cannot be alleviated by output forwarding, then the impacted pipeline stage must stall as before.

Your Task

In this section, you will implement both a `ForwardingUnit` and a `ForwardingProcessor`. Note, this component *extends and/or replaces* the functionality of a the `HazardCheckingUnit`, so the `ForwardingProcessor` will only have one monitoring unit. Much like the in the previous section, your `ForwardingUnit` will do this by checking for any operand dependence between cycles in this processor. Unlike the previous section, your `ForwardingUnit` will only return that a hazard exists if there is an input source that depends on an output that cannot be forwarded from elsewhere in the pipeline. The `ForwardingUnit` performs any possible forwarding while checking for these dependences in the `bool operandDependence()` function.

Programming You will implement the `ForwardingUnit` class that is declared in `src/components/forwarding_unit.hh` and the `ForwardingProcessor` that is declared in the `src/processors/forwarding.hh`.

You will notice that the starter scaffolding for the `ForwardingUnit` is minimal. As such, in your implementation of the `ForwardingUnit`, you are tasked to do three things in your implementation: ① adding the class attributes necessary to implement the forwarding logic, ② ensure that the `ForwardingUnit` constructor appropriately defines each of these attributes, and ③ implement the `operandDependence` function.

Your implementation of the `ForwardingProcessor` should extend the starter scaffolding to add a `ForwardingUnit` to the class declaration and define it in the constructor. Additionally, you will modify the main loop of the `executeProgram` function to incorporate the `ForwardingUnit` logic as appropriate to avoid stalls.

Testing. You can test your forwarding processor by executing the sample VBISC programs in the `test-progs` repository by using the compile command `make forwarding` running `./bin/forwarding`. What do you notice about the output from this processor compared to the single cycle processor output? Try to reason about what the correct behavior should be for this processor, and use the test cases on Gradescope to confirm what you expect to happen.

Submission Guidelines. When you submit your repository to Gradescope, an extended version of the `forwarding.cc` is used to test your implementation.

Hints and suggestions. Itemized list to be updated as needed. Be sure to refresh this document for updates!

Written Response In a section of your written response document clearly marked "Forwarding Response", respond to the following prompts.

1. In the `ForwardingProcessor`, is there ever a case in which a hazard cannot be resolved via forwarding? Why or why not? What would need to change about the processor architecture to change your answer?

7 Reflection

In a section of your written response document clearly marked "Reflection", respond to the following prompts. Your responses in this section will only be graded on a completion basis, and will be used to help refine future iterations of this assignment. Try not to spend more than 5-10 minutes on this section.

1. On a scale from 1-5, how difficult was this assignment? A score of 1 would imply 105 assignments are much easier, 3 implies similar difficulty, and 5 implies much harder. Please give your reasoning.
2. What was confusing about this assignment write-up and/or starter code?
3. About how long did you spend on this assignment? If you went to office hours, did you find them effective?
4. Do you have any other thoughts or comments?