In this assignment, you will build a processor emulator that defends execution from instruction-count leakage based on conditional branching. Suppose you have the following program:

```
int main(int argc, char **argv)
{
    assert(argc == 3);
    int result = 0;
    if (strcmp(argv[1], "supersecretdata") == 0) {
        for (int i = 0; i < 65536; i++) {
            result += stoi(argv[2]);
        }
    } else {
        result = stoi(argv[2]) * 65536;
    }
    return 0;
}
```

Much like in the ISA assignment, this program exhibits significant information leakage depending on the user input, which an adversary may be able to steal. Unlike in the ISA assignment, merely modifying the functional units of the processor is insufficient to hide the information leaked about the program's execution. The reason for leakage in this instance is because of the *number of instructions executed* on either side of the branch. That is, different inputs will result in different numbers of instructions executed, so leakage is possible even when the instruction execution latency is uniform (much like the processor in the Pipeline Emulator.

In this assignment, you will develop a series of processors much like first pipeline emulator; however, this processor is based on an instruction set exclusively using control. First, you will be asked to implement a processor that implements control instructions that are evaluated in a single cycle. Then, you will be asked to pipeline this processor and implement mechanisms for safety against control hazards. Finally, you will implement a processor that speculatively executes both sides of a conditional jump before committing one side of the branch. This approach has been propsed in recent literature referred to as Secure Multi-Path Execution (SecMPE).

After demonstrating that you can leak this information, you will modify the processor to hide any potential leakage. You will also extend the RISC-V ISA to optimize the processor for common cases for which you determine the application doesn't leak any meaningful information. Your submission should include the **code/programs** associated with each part of the assignment, as well as a **written component** that you will answer as you go.

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

Pull the assignment stencil from the course repository. 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.

Deadlines

Section	Deadline	Submission URL	Points
Introducing COVBISC	April 17, 11:59pm	Gradescope (Part 1)	5pts
Single Cycle and Simple Pipeline	April 24, 11:59pm	Gradescope (Part 2)	15pts
Hazard Checking	April 24, 11:59pm	Gradescope (Part 2)	15pts
SecMPE COVBISC	May 4, 11:59pm	Gradescope (Part 3)	10pts
Paranoid Processor	May 4, 11:59pm	Gradescope (Part 3)	5pts

1 Introducing COVBISC

Like in HW1, you will be constructing a processor emulator based on a custom instruction set. Rather than using VBISC, this assignemnt will use the Control Only Very Basic Instruction Set Computer (COVBISC) instruction set[1]. The full instruction set is enumerated in Table 1. Like in VBISC, COVBISC assembly does not expect any preamble and exits with a simple `end` instruction and COVBISC 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
`nop`	no operation	`nop`	A do nothing operation.
`jmp`	jump	`jmp` IMM	Set the program counter to its current state plus the offset specified by the immediate.
`be`	branch equal to	`be` RS1, RS2, IMM	Conditionally set the program counter to its current state plus the offset specified by the immediate if source 1 is equal to source 2.
`bne`	branch not equal	`bne` RS1, RS2, IMM	Conditionally set the program counter to its current state plus the offset specified by the immediate if source 1 is not equal to source 2.
`blt`	branch less than	`blt` RS1, RS2, IMM	Conditionally set the program counter to its current state plus the offset specified by the immediate if source 1 is strictly less than source 2.
`bge`	branch greater than or equal to	`bge` RS1, RS2, IMM	Conditionally set the program counter to its current state plus the offset specified by the immediate if source 1 is greater than or equal to source 2.
`end`	end	`end`	The program has finished. Exit.

Table 1: The complete COVBISC instruction set.

Your Task

Programming

In order to get a better understanding of the instruction set, your job is to write a series of test cases that exhaust each path of the provided test programs in the `test-progs` directory. That is, for every possible path through the program, you should have a set of inputs that exhaust that path (you may need multiple sets of inputs for the same test file). Then, in a directory labeled `test-inputs`, submit one file for each test n called `input<n>`, where the format of the file is exactly:

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

```
<program input>
r1: <n1>
r2: <n2>
r3: <n3>
r4: <n4>
r5: <n5>
r6: <n6>
r7: <n7>
```

As an example, to test `test-progs/test0` where each register is input with value 0, you would create a file called `test-inputs/input0` with the following contents[2]:

```
test-progs/test0
r1: 0
r2: 0
r3: 0
r4: 0
r5: 0
r6: 0
r7: 0
```

For your convenience, an oracle program called `ask-covbisc-oracle.py` is provided. To use the oracle, run `python3 ask-covbisc-oracle.py /path/to/input<n>` where the final argument is the path to your test.

Written Response

In a section clearly labeled "Introducing COVBISC", please respond to the following prompts.

1. Think about the possible outputs of a program in the COVBISC instruction set. What are some common features of programs in high-level language that cannot be represented in this assembly language? Which features of COVBISC make it such that these features are unable to be represented?

2. Are programs in the COVBISC instruction subject to the leakage vulnerability described in the SecMPE paper? If so, write an example program that may leak its input via the side channel described. If not, provide a justification as to why leakage is not possible.

[2]This file is provided to you as part of the assignment stencil

2 Single Cycle and Simple Pipelined COVBISC Processors

In this section, you are asked to implement a processor in which all instructions are executed in a single cycle. The key challenge of implementing this processor is that the program counter needs to be appropriately updated with respect to the instruction set. Note that, by default, the program counter is always incremented. As such, the offset described in an instruction is with respect the program counter as it is *post increment* (e.g., an indirect branch).

Afterwards, you should take the logic that you have implemented in the single cycle processor, and break the work across multiple stages of execution. The work from your implementation in the main loop of the `single_cycle.hh` processor should be broken into stages where the work executed per stage should closely resemble the Simple Pipelined processor from HW1. In particular:

- `src/stages/fetch.hh` should handle all execution tasks related to fetching instructions (e.g., using instruction memory).
- `src/stages/decode.hh` should handle all execution tasks related to interpretting instructions and retrieving data from the register file.
- `src/stages/execute.hh` should handle all ALU related tasks. Unlike in VBISC, the ALU operations to execute in COVBISC are not directly related to the instruction type, but the result of any operation should be inferrable by the ALU operation performed.
- `src/stages/writeback.hh` should handle the updating of all register values (including the program counter). Note, you may know the true resolution of the progarm counter before Writeback, but the PC should only ever be updated in the Fetch and Writeback stages!

Your Task

Programming. Seeing as the instruction set has changed from HW1, you will need to implement a new decoder (`src/components/decoder.hh`) to interpret the instructions in this instruction set. From there, your job is to implement the logic to execute the COVBISC instruction set where each instruction is executed in a single cycle in the `src/processors/single_cycle.hh` file. You may want to refer back to your implementations of the decoder and the single cycle processor from HW1 to do so.

In order to translate your single cycle processor to the simple pipelined processor, you should implement each of the `src/stages/{fetch.hh, decode.hh, execute.hh, writeback.hh}` files. You will need to implement each of the pipeline registers in the `src/pipeline_registers` directory to communicate information between these stages, where stages refer to the associated register as `input_reg` and `output_reg`.

Written Response. In a section clearly labeled "Single Cycle and Simple Pipeline", please respond to the following prompts.

1. Do your single cycle processor and simple pipelined processors follow the same control flow through a program? Provide a sample trace of the instruction execution relative to clock cycles executed to justify your answer.
2. Describe how branches are predicted in your processors. Is this prediction mechanism static or dynamic? Local or global?

3. Think about the values that you needed to pass from one stage to another in the pipelined processor from HW1 in the VBISC instruction set. Which fields need to be communicated in this processor that didn't need to be communicated in that processor, and vice versa? What does your response tell you about the inter-stage communication complexity of implementing control instructions relative to arithmetic instructions?

Hints and suggestions. Note that, unlike HW1, mutliple stages of execution can all modify the program counter and as such, the PC is an attribute of the *processor* rather than of the *fetch stage*.

Seeing as your pipelined processor will need to exit, you should implement `drain`ing much like in HW1. Realize that control hazards are not handled in this processor. What does this mean for how the processor emulator exits?

3 Handling Control Hazards

In this part of the assignment, your task is to resolve the control hazards introduced by your `SimplePipeline` processor from Part 2. Unlike in HW1, the hazard checking logic for this processor should not be implemented in a unique component, but instead should exclusively be implemented in the main loop of the `src/processors/hazard_checking.hh` processor.

Control hazards arise when the view of the program counter in the `Fetch` stage is inconsistent with the true resolution of the branches executed by the program. Before jump instructions commit the true program counter, incorrect instructions may reside in the pipeline *speculatively*. It is your job to implement a "hazard checking" processor that clears the pipeline of instructions that should not be executed.

Your Task

Programming. Implement the logic to flush the pipeline of incorrect instructions in the `src/processors/hazard_checking.hh` file. Your processor should *not exit early* if it is the case that an `end` instruction is executed speculatively.

Written response.

1. In this processor, the PC is updated in the Writeback stage. As we have discussed in class, unconditional jumps *can* be resolved as early as the Decode stage, and conditional jumps can be resolved as early as the Execute stage. Contrast your hazard checking processor with this alternate construction. You may want to think about performance and complexity in your response.

2. Consider what would happen if the VBISC instructions from HW1 were also included in this instruction set (e.g., COVBISC also has arithmetic instructions). What types of additional hazards could occur? Describe what changes you might need to make in order to handle these hazards.

Hints and suggestions. You may want or need to modify the logic of some of your pipeline stages in order to appropriately check for hazards in the pipeline. If you make a modification to your pipeline stages, verify that any prior implementation still runs as expected!

4 SecMPE COVBISC

In order to implement the protocol proposed in SecMPE, you first need to think about how to reason about programs written for this type of processor. The goal of implementing secure multi-path execution is to ensure that the processor fully executes both sides of a conditional jump to completion before committing one side. Unfortunately, knowing when "a side of a conditional jump has completed" is non-trivial for at least one side of the branch.

In Section IV.C of the paper, you will notice that the proposed protocol injects `eosJmp` instructions to allow the execution of a sequence of instructions to jump back to an alternative target. `eosJmp` instructions work like unconditional jumps in the COVBSIC instruction set in that they specify an instruction target to execute next. The first `eosJmp` instruction to execute for a given branch indicates the alternative target of the original conditional jump, and the second `eosJmp` instruction to execute indicates the address of the instruction following the first `eosJmp` instruction in the event that this was the correct side of the branch to execute, as determined by the condition.

In this assignment, you will implement a different instruction, called `eos`. Unlike an `eosJmp` instruction, the `eos` instruction trigger a hardware routine in the next part of this assignment, where secure branching is implemented entirely in the processor. In particular, the usage of the `eos` instruction is as follows:

operation	Mnemonic	Format	Description
`eos`	end of secure	`eos`	When encountered during a speculative period, this instruction flips the execution to the other side of the branch. Otherwise, this instruction is treated as a `nop` by the processor.

As per the paper, the hardware only makes sense assuming that programs have been compiled specifically for this processor type with conditional branches protected in software.

Your Task

Programming. To build a better understanding of how the execution should work in the SecMPE, we first need to have assembly files that indicate the branches that our processor should protect. For each of the provided assembly programs, your job is to extend each of the COVBISC programs in the `test-progs` directory to the SecMPE COVBISC assembly language. For example, to extend `test-progs/test0`, you should write a new file called `test-progs/test0-sec` that has the same functionality, but instead protects the instructions on either side of the conditional branch.

Written response.

1. For each of `test0-sec`, `test1-sec`, and `test2-sec`, compare the register inputs that would trigger all control paths through the programs. To what extent does the execution remain similar or different?

2. By modifying both the program and the instruction set, the SecMPE protocol is engaging in *hardware-software co-design* to implement secure multi-path execution. The next questions ask you to think about the implications of such a design decision:

(a) Suppose the implementation of the SecMPE protocol was a *software only* approach. What difficulties would be faced in implementing such an instruction set?

(b) Imagine if CompanyX adopts the SecMPE COVBISC instruction set in order to secure their processors. Would end-users be safe? Why or why not? Be sure to use the implications of hardware-software co-design in your response.

Hints and suggestions. For the purposes of this assignment, two programs A and B are functionally equivalent if A executes at most one additional `nop` instruction with respect to B. However, some transformations from COVBISC to SecMPE COVBISC are impossible without at least one additional `eos` instruction outside of a branch.

5 Implementing a Paranoid Processor

Given that you have now written programs that can run on a "Paranoid Prediction Processor", all that's left to do is write a processor that implements the SecMPE protocol! In the originally proposed design, the processor uses an additional component, the *jumpback table*, to indicate where `eosJmp` instructions should refer to. When executing a conditional branch, the SecMPE processor *always executes the not-taken side of the branch first*, and your paranoid processor should do the same! In your implementation, this will all be done in the Branch Target Buffer (BTB) defined in the `src/components/btb.hh` header file.

At a high level, the role of the table is to, for each program counter associated with a conditional branch, store all of the relevant metadata to perform ① the jump back, ② track the true outcome of the branch, and ③ determine whether the program would have exited if that side of the branch should have exited.

To account for nested conditional jumps, the table has *multiple entries*. The bottom-most (e.g., highest index) entry refers to the current conditional branch being executed. Entries are created whenever a conditional branch is decoded, and the entry is filled when the fields are decipherable as the instruction flows through the pipeline.

Note that, as with earlier processor designs, the program counter should continue to be speculatively incremented in the `ParanoidFetch` stage, and any later updates to the program counter are to be done in the `ParanoidWriteback` stage. This applies to unconditional jumps and resolved conditional branches after both sides have executed.

Your Task

Programming. In order to add the paranoid prediction logic to the processor emulator, you should implement the `tick` function in each of the `src/stages/paranoid_{fetch, decode, execute, writeback}.hh` classes. By default, these classes call the non-paranoid implementation that you have developed in Part 3. You may find it helpful to continue to call these functions with some extended auxiliary logic or re-implement their functionality in these classes.

In addition, you should complete the main loop of the `executeProgram` function in the `src/processors/paranoid.hh` processor class, and add the necessary fields to the `src/components/btb.hh` class to facilitate secure branching. Note, the `BTB` is an attribute to each of the paranoid stages, so any stage.

Written response.

1. In your paranoid BTB, what speculative state needs to be maintained? If the instruction set was more verbose (e.g., a combination of VBISC and COVBISC), what additional speculative state would need to be maintained?

2. Which components are the ones that make the most sense to maintain non-committed speculative state?

Hints and suggestions. You may find it useful to consider Figure 5 from the SecMPE paper as a starting template for your BTB implementation.

There may be additional fields that are helpful to communicate between stages using the `pipeline_registers`. If you add fields to these registers, your code must remain backwards compatible (e.g., the `HazardCheckingProcessor` must still be able to execute).

6 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 other assignments in this course have been 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?