ASSIGNMENT 5

CS 473: Compiler Design / Fall 2025

Description

In this assignment, your compiler will take the list of IRs from the previous assignment and us it to generate MIPS assembly code that can be executed by a simulator. Your submission will abstract the MIPS architecture as a simple stack-machine, similar to the IR used in class and in the previous assignments. You should use the lexical parser that you implemented in Assignment 1, the syntactic parser that you implemented for Assignment 2, the AST and semantic analysis that you implemented for Assignment 3, and the translation from AST to IR that you implemented for Assignment 4.

The Cardinal Language

The Cardinal language is defined in a separate document.

MIPS as a stack machine

We saw in class how to use a stack machine as a very simple model that can execute code. You should treat MIPS as a stack machine:

- Decide on two registers to use when compiling each expression (for instance, `$v0` and `$v1`). The contents of these registers are not saved, so you can only rely on their contents immediately after writing them. For instance, the following code will result in undefined contents of register `$v0`:

```
emitInstruction("li $v0,473");
mips_astExp(…); // This call may overwrite the contents of register $v0
// Contents of register $v0 are now undefined and cannot be used
```

  Instead, organize your code differently:

```
mips_astExp(…); // This call may overwrite the contents of register $v0
emitInstruction("li $v0,473");
// Contents of register $v0 are guaranteed to be 473
```

- Familiarize yourself with how to use the stack in MIPS, which was covered in class.
- Ensure that each expression takes its operands from the stack and leaves the result on the stack.
- Ensure that statements leave the stack unmodified, consuming all values that are pushed.

Intrinsic functions and system-calls

You should implement compiler intrinsics by calling the appropriate system call provided by the MIPS simulator: https://hwlabnitc.github.io/MIPS/mips_syscalls&tutorial#system-calls

Structure of the output

The autograder generates file `out.mips` as the result of compiling each Cardinal source code file. You should inspect this file as you develop your solution to ensure the correctness of the generated MIPS assembly.

Function Implementation strategy

This assignment requires your compiler to support function definition and invocation. Here is a way to add incremental support:

1. Add support for leaf functions (i.e., functions that do not call any other functions but may call intrinsics) that return an integer and take no arguments.
    a. Use the MIPS calling convention seen in class to jump to the start of the function.
    b. On return, make sure you push the return value before jumping back to the callee.
    c. When compiling a function call, you can expect the return value on top of the stack after the function returns
2. Add support for non-leaf recursive functions that take any number of arguments using **stack frames.** Start as simple as possible and add complexity as needed.
    a. Decide on the format of your stack frame and write it down, you'll be using it throughout this process.
    b. Start by supporting non-leaf functions that take zero arguments by saving the return address on the stack before each function call, and jumping back to it when returning
    c. Add support to a frame-pointer inside your stack-frame, so you can use it for the next steps
    d. Add support to function arguments to the stack-frame.
        i. Push all arguments when calling a function
        ii. Inside a function, access arguments via the argument index (computed for Assignment 3) and the frame-pointer. For instance, argument N may be found on ($fp+N)*4. Refer back to your design for the stack-frame to figure out your formula.
    e. When returning:
        i. Compute the return value of the function. Do this first so that the return value can still access any arguments.
        ii. Save the return value in a register
        iii. Save the return address in another register

iv. Reset the stack-pointer and frame-pointer so you "pop" all values from the stack, and restore the stack to how it looked before the function was called. Be careful not to overwrite the return value and return address.
v. Push the saved return value back to the stack
vi. Jump back to the return address.

3. Add support for local variables
    a. Reserve extra space in the stack-frame for all local variables at the start of each function
    b. Then, before executing any function code, initialize each local variable
    c. You can access local variables just as if they are extra arguments, using the indexes you computed during Assignment 3.

Extending Cardinal

Assignment 5 requires you to add an extension to Cardinal, and then upload a short video explaining your extension. Below you have a list of possible extensions.

Minor extensions

- Add a for loop to Cardinal that behaves like C
    - for(<init>, <step>, <cond>) { <body> }
        - Executes <init> before entering the for
        - Executes <body>
        - At the end of <body>, executes <step>
        - After executing <step>, executes <cond> and repeats Steps 2-4 once if <cond> is not zero
        - Exits the loop if <cond> is zero
- Ternary operator as an expression, like in C: <cond> ? <then> : <else>
    - Unlike the if statement, this is an expression
    - Evaluate the condition first
    - If true, the whole expression evaluates to the result of <then>
    - If false, the whole expression evaluates to the results of <else>
- Add a variable that contains how many iterations are left on repeat:
    - E.g., repeat(10) { print_int(nnn); }
        - Prints 10987654321
    - Using the variable outside of a repeat results in a symbol error
    - Defining a variable with the same name? Up to you. May be an error, or may shadow the repeat variable.
- Add more statements and expressions. You have to support ALL of these:
    - Binary statements: +=, -=, *=, /=, |=, &=, ^=
    - Unary expression negation (flips all the bits): ~
    - Binary shift expressions: <<, >>
        - You can modify the IR as needed to support binary shifts

- Minor extensions are graded as follows using videos of the given length:
    - Show and explain the changes in lexer and parser rules, show that your lexer and parser work as expected with positive and negative examples
    - Show and explain the changes in the semantic analysis, show that your semantic analysis works as expected with positive and negative examples
    - Show and explain the changes in the AST to IR translation and IR tree to list
    - Show that your extension works as expected with end-to-end positive examples (e.g., a correct for loop that iterates the expected number of times)

Optimizations

Instead of extending Cardinal, you can perform some type of program optimization. Files optimize.h and optimize.c contain entry-points for optimizations at different levels (AST, Tree IR, List IR, List IR with explicit PUSH/POP operations.

For instance, you can write an optimization that removes sequences of PUSH-POP to the same register.

Major extensions

You can suggest a major extension to the Cardinal compiler **instead** of performing this assignment. Please reach out to Prof. Pina about this opportunity. Here are some ideas:

- Write a debugger for the AVES IR that allows to visualize the stack before and after each individual IR. You can write such a debugger as a JavaScript webpage.
- Reimplement a portion of the Cardinal compiler in a different language (e.g., Rust, Java, Python, etc.)
    - This includes lexer, parser, semantic analysis, IR translation, and MIPS code emission
    - You don't have to implement the whole specification for Cardinal
- Reimplement the front end to target LLVM IR instead of AVES IR

Entry Point

You will reuse files `lexer.lex`, `parser.y` , `semantic_analysis_(symbols|types).[ch]` and `frame.[ch]`, `transform.c`, `ast_to_ir.c`, and `ir_tree_to_list.c` from Assignment 4.

You will modify file `ir_list_to_mips.c` by implementing an IR traversal that emits MIPS code, thus compiling programs in Cardinal. You can modify files `optimize.c` and `ir_pushpop.c` as needed.

Due Date and Resubmission Policy

This assignment has 2 due dates, one for each half of the tests. The extension is due on the second date.

First Due Date

The first half of the assignment is due on **November 22nd 2025** (Saturday) at **5pm CST**. There is no late policy.

The code and date used for your submission is defined by the last commit to your Git repository.

Second Due Date

The second half of the assignment is due on **December 6th 2025** (Saturday) at **5pm CST**. There is no late policy.

The code and date used for your submission is defined by the last commit to your Git repository.

Resubmissions

Each half of the assignment is resubmitted in separate. ~~To resubmit this assignment, your **original grade** (as defined by the autograder) should be **equal to or higher than 30%** for undergraduate students.~~ You can resubmit the first half of your assignment until **November 29th 2025** at **5pm CST** and the second half until **December 13th 2025** at **5pm CST**. Together with your resubmission, you will have to submit a written description of what you changed from the original submission (on Gradescope).

Bonus Points

This assignment has a total of **30% bonus points**.

You can earn 10% extra points by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment5` label, and non-anonymous to the instructors to count towards the bonus. You can claim bonus points through **a Gradescope quiz**.

You can earn 10% extra points by fixing all memory leaks in your program. Each passing test without memory leaks is worth 1% bonus points.

You can earn 10% extra points by passing all the tests by the first due date. Each passing tests 6 through 10 by the first due date is worth a 2% bonus.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of **80%**. You can check your current grade at any point by submitting your code and checking the autograder. The automatic grade is determined by 10 tests, that will check if your submission outputs the expected result. **Each test in the first half is worth 10%, each test in the second half is worth 6%.**

This assignment also has a 20% component for your extension of Cardinal. You will have to submit a 5 minute video screencast as explained above to claim points for your extension.

> You can record such a video using Zoom, which you may already have installed to attend lectures remotely. Simply start a meeting (without any other participants), share your screen, and start recording. Note that Zoom requires some time to process your video after you record it, so plan accordingly. Extension requests to upload videos after the due time and date because Zoom is still processing them will be denied.

Instructors will stop watching videos at the 5 minute timestamp (nothing past that point in the video will be graded). This video should be a screencast of your IDE open on the code submitted, and you should highlight the code. Note that longer videos are not better videos, and you should record a video as short as needed to show all the expressions and answer the questions above. Attempts to speed up the video playback result in a 0% grade for the extension.

Graduate students

There is no extra requirement for graduate students. Graduate students are encouraged to add support for arrays as their extension, but they can add any other extension they choose.

Errors and Omissions

If you find an error or an omission, please post it on Piazza as soon as you find it.

Hardcoding and Academic Integrity

Any hardcoding will result in a 0% grade. Hardcoding is when you submit code that detects which test is being run, and simply outputs the expected result. For instance, detecting that test 22 is running, and replacing the usual execution of your submission with `printf("expected result").`

The academic integrity policy described in the syllabus applies to this assignment. You are responsible for writing all the code that you submit. We will use an automatic tool that detects plagiarism on all submitted code, and we will investigate all instances where plagiarism is more than likely.

Please refer to the syllabus for the full academic integrity policy.