ASSIGNMENT 4

CS 473: Compiler Design / Fall 2022

Description

In this assignment, your compiler will traverse the AST from previous assignments and generate MIPS assembly code that can be executed by a simulator. Your submission will abstract the MIPS architecture as a simple stack-machine, as we saw in class. You should use the lexical parser that you implemented in Assignment 1, the syntactic parser that you implemented for Assignment 2, and the AST and semantic analysis that you implemented for Assignment 3.

The Albatross Language

The Albatross 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
```

- Implement functions `push0` and `push1` to push each register to the top of the stack.
- Implement functions `pop0` and `pop1` to pop the top of the stack to each register.
- 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 any values that are pushed.

Intrinsic functions and system-calls

The AST from Assignment 3 now has two new nodes for intrinsic function calls, both as statements and expressions. You should implement these by calling the appropriate system call provided by the MIPS simulator: http://courses.missouristate.edu/kenvollmar/mars/Help/SyscallHelp.html

Structure of the output

The autograder generates file `out.mips` as the result of compiling each Albatross source code file. You should inspect this file as you develop your solution to ensure the correctness of the generated MIPS assembly. You are encouraged to follow the structure below:

```
# Define all variables here in a .data section
# You may have to introduce variables not in the original program
# (e.g., repeat loops, string literals)

 .data
i:    .word   0       # defines i as an integer variable
s:    .asciiz "CS473" # defines variable s as a zero-terminated string

.text

# Global variable initialization code goes here
# (e.g., var i int := 400 + 73;)
li $v0, 400
li $v1, 73
add $v0,$v0,$v1
sw $v0, i

# jump to the top-level code with a label
j _main

# functions go here
f1:
…

f2:
…

# top-level code starts here
_main:
…
```

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 different return types on leaf functions: string and void.
3. Add support for passing up to 4 arguments on leaf functions. You can use the MIPS calling convention seen in class for that (i.e., using registers a0-a4). You don't need to save any register.
4. Add support for non-leaf recursive functions that take any number of arguments using **stack frames**
    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.
5. (bonus) Add support for local variables
    a. Reserve extra space in the stack-frame for each local variable
    b. Before starting the function, execute the initialization of each local variable
    c. You can access local variables just as if they are extra arguments, using the indexes you computed during Assignment 3.

Entry Point

You will reuse files `lexer.lex`, `parser.y` , `semantic_analysis_(symbols|types).[ch]` and `frame.[ch]` from Assignment 3.

You will modify file transform.c with any code transformations your compiler requires, and file mips_ast.c by implementing an AST traversal that emits MIPS code, thus compiling programs in Albatross.

You may need to modify your semantic analysis to add the Albatross intrinsics as functions.

Due Date and Resubmission Policy

This assignment is due on **November 12 2022** (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.

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 your assignment until **November 19 2022** (following Saturday) 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 **10% bonus points**, which you can earn by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment4` label, and non-anonymous to the instructors to count towards the bonus. You can claim bonus points through **a Gradescope quiz**.

There is **one extra test** that allows you to implement support for local variables for an extra bonus of 10%.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of **150%**. You can check your current grade at any point by submitting your code and checking the autograder. The automatic grade is determined by 15 tests, that will check if your submission outputs the expected result. Each test is worth 10%.

Graduate students

There is no extra requirement for graduate students. You will add support for array the array operations added in previous assignments in Assignment 5.

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.