CS323 Project Phrase 4 Report

12112421 王德涵 12112620 徐霄阳 12112822 孔繁初

Assumptions

As described in the requirement document, our implementation is based on the following assumptions:

Assumption 1 The intermediate code are logically correct, meaning that you can run them on the IR simulator and obtain the correct output.
Assumption 2 There are no structure or array variables, so you don't need to translate the DEC instruction.
Assumption 3 All integer constants are in the range $[-2^{16}, 2^{16}-1)$ so that they can be safely represented by MIPS32 immediates.

Translation Schemes

Register Allocation

We employ a method similar to the way cache operates for register allocation. Specifically, the following steps are taken whenever a variable needs to be used:

1. If the variable is in a register, get its value directly from the register.
2. If the variable is not in a register, and there is an available space in the registers, allocate a register to the variable.
3. If the variable is not in a register, and there are no available spaces in the registers, find the least recently used variable in the registers, store it back to memory, and allocate the register to the requested variable.

We only use register t0-t9,s0-s7(total 18) registers to store the variables, for others have special uses.
However, due to the difference between the MIPS code generation order and the actual execution order, extra handling is required for statements like `if` statements that introduce branching. When encountering an if statement, we store back into memory all variables that may be used in the registers shortly after, to mitigate potential issues.

Recursion and backtracking

When we meet a code whose format is **var := call fun1**, we will store all variables possibly used in the active function to memories, and pick them out from memories after the called function (fun1) is finished.

TAC Translation

We didn't use the provided starter code. Instead, we implemented an end-to-end compiler by directly adapting the compiler front-end to the target code generator, which means, our compiler can accept both **.spl** and **.ir** file as input file. We translate TAC code instructions by

instruction, that is, simply convert the TAC by one-to-one mapping. Our mapping scheme can be seen in the code file **mips.c**.
Specifically, for calling a function, our compiler will save all the local variables in the stack frame, and restore them after the callee returns.

Extra Test Cases

In out extra test cases, **test case a** includes calling a function that requires the caller to pass up to 32 parameters, showing that the generated code can correctly pass more than 4 parameters. **Test case b** includes a relatively complicated calling stack, showing that the generated code can handle local variables in different running environments properly. **Test case c** includes calculating 3^6 using exponentiation by squaring algorithm, and the result can be calculated correctly.
To show that our code is end-to-end, we add all test cases in **test 3** in folder ./test too. You can see how our code generate **ir** code and **Mips** code buy running **run_test.py** here.

Run code

For end-to-end, you just need to put your test cases in ./test folder, then run **run_test.py. This python program will delet all .ir and .s files in folder.** To avoid this, you need to backup them if necessary. This program will generate ir codes based on our own algorithm, and generate Mips code based on **our** ir codes, but not the provide one.