# CS323 Project 4 Report

11812804 董正

11813225 王宇辰

# 1 Introduction

## 1.1 Project Requirements

With the three-address code in hand, our compiler will handle these low-level stuff and finally generate executable code (MIPS32 assembly code), which can run on a MIPS32 machine.

- Combine the first three projects.
- Implement register allocation.
- Implement procedure call convention.

We did not implement any advanced features. All strategies are naive algorithm.

## 1.2 Development and Test Environment

- Linux
  - `Ubuntu 18.04.6 LTS x86_64` with `Linux 5.4.0-87-generic`
  - `gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0`
  - `g++ (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0`
  - `flex 2.6.4`
  - `bison (GNU Bison) 3.0.4`
  - `GNU Make 4.1`
- macOS (Arm)
  - `macOS 12.1 21C52 arm64` with `Darwin Kernel Version 21.2.0`
  - `Apple clang version 13.0.0 (clang-1300.0.29.30)`
  - `flex 2.6.4`
  - `bison (GNU Bison) 3.8.2`
  - `GNU Make 3.81`

# 2 Design and Implementation

## 2.1 Register Allocation

We used only `$t0, $t1, $t2` for expressions, which is easy to implement. Therefore, the strategy is **write-through**, i.e. write to the corresponding memory immediately after an register is updated.

First, we need to allocate memory for each variable used. When scanning the whole IR program, store all variable names in a set:

```
1 set<string> vars_set;
```

After generating MIPS code, append variable allocation to the head of the code.

```
for (auto s : vars_set) {
  fprintf(output, "_%s: .word 0\n", s.c_str());
}
```

Therefore, it will be like

```
.data
_t0: .word 0
_t1: .word 0
_t2: .word 0
```

And `t0 := t1 * t2` will be generated to

```
lw $t1, _t1
lw $t2, _t2
mul $t0, $t1, $t2
sw $t0, _t0 # write through
```

# 2.2 Caller's and Callee's Sequence

## 2.2.1 Caller's Sequence

- IR: `ARG v0`

  For transmiting arguments, we did not use `$a0...` registers. Instead, we pushed arguments into stack. Additionally, we used an register `$s0` to count the number of arguments, which is initialized to zero.

  Function `emit_arg(tac *arg)`:

  1. Decrease `$sp`
  2. Increase `$s0`
  3. Push arg into `0($sp)`
- IR: `ret := CALL func`

  Function `emit_call(tac *call)`:

  1. Record current `$sp` to `$t4`, this is the start address where callee gets arguments
  2. Decrease `$sp` and store `$s0`
  3. Set `$s0` to zero for the callee to use
  4. Decrease `$sp` and store `$ra`
  5. Decrease `$sp` and store all variables

     We did this because recusive function shares the same variable names and the value in memory will be overwrite
  6. `jal`

7. Increase `$sp` and load `$ra, $s0` and all variables
8. Set `$s0` to zero in case that there is another function call after it

### 2.2.2 Callee's Sequence

- IR: `PARAM v0`

  Function `emit_param(tac *param)`:

  1. Load paramter from `0($t4)`
  2. Increase `$t4`
  3. Insert variable name to the set
- IR: `RETURN v0`

  Function `emit_return(tac *return_)`:

  1. Load return value to `$v0`
  2. `jr $ra`

# 3 Conclusion

In this project, we implemented an generator of IR codes to generate MIPS32 codes. The most difficulty we met is how to design the caller's and callee's sequences, because we did no bonus part in the previous projects. Therefore, it is very hard to control the variables and our generated asm code is very ugly.

This the end of the course. We learned a lot about compliers and implemented a simple compiler by ourselves, although it is unpolished. We believe it will benifit us in our future study.