Report on Lab 4

编译方法

在 `Code` 文件夹下输入命令 `make` 即可自动编译生成 `parser` 文件。

功能简介

文件结构如下所示：

```
Code
├── Makefile
├── intercode.h./c
├── lexical.l
├── main.c
├── mipscode.h/.c
├── semantic.h/.c
├── symboltable.h/.c
├── syntax.y
├── syntaxtree.h/.c
└── translate.h/.c
```

Lab 1：`lexical.h` 存放与词法分析相关的 `flex` 代码；`syntax.y` 存放与语法分析相关的 `bison` 代码；`syntaxtree.h` 和 `syntaxtree.c` 用于存放与语法树相关的结构体定义和函数。
Lab 2：`symboltable.c` 和 `symboltable.h` 存放与类型相关的结构体、函数和已定义的 Symbol/Structure/Function 表；`semantic.c` 和 `semantic.h` 存放进行语义分析的函数。
Lab 3：`main.c` 存放主函数；`intercode.h` 和 `intercode.c` 存放与中间代码的构造与操作相关的定义和函数；`translate.h` 和 `translate.c` 存放与翻译的函数。
Lab 4：`mipscode.h` 和 `mipscode.c` 用于存放把 IR 翻译为 MIPS 汇编的函数。

MIPS语句容器

由于本系列实验后续不再基于 MIPS 汇编进一步操作，因此本实验中直接将 MIPS 汇编以字符串形式存储；与 IR 类似，依然使用链表形式进行存储：

```
struct MipsCodes {
    MipsCode* head;
    MipsCode* tail;
};
struct MipsCode {
    char* code;
    MipsCode* next;
    MipsCode* prev;
};
```

我为 MIPS 汇编设计了如下的封装函数和宏定义：

```
MipsCode* get_MipsCode(char* fmt, ...);
MipsCodes* get_MipsCodes();
MipsCodes* get_Init_MipsCodes();
void Append_MipsCode(MipsCodes* codes, MipsCode *code);
MipsCodes* translate_ic(InterCodes* ic);
void fprint_MipsCodes(FILE* f, MipsCodes* mc);

#define AM(fmt, ...) Append_MipsCode(mc, get_MipsCode(fmt, ##__VA_ARGS__))
```

因此在添加语句时，只需要保证已经定义过 `MipsCodes* mc` 即**可直接使用类似** `AM("  sw $t0, %d($sp)", x_offset);` **的类 stdio 格式便捷地进行翻译。**

> 寄存器分配与管理采用朴素分配法，即所有变量均存储在内存中，因此不做阐述

栈管理

我的栈管理方式可以用下图表示：（图见下页）

- 黑色：原始状态
- 蓝色：调用者 CALL 之后的状态
- 红色：被调用者初始化后的状态

遵循以下原则：

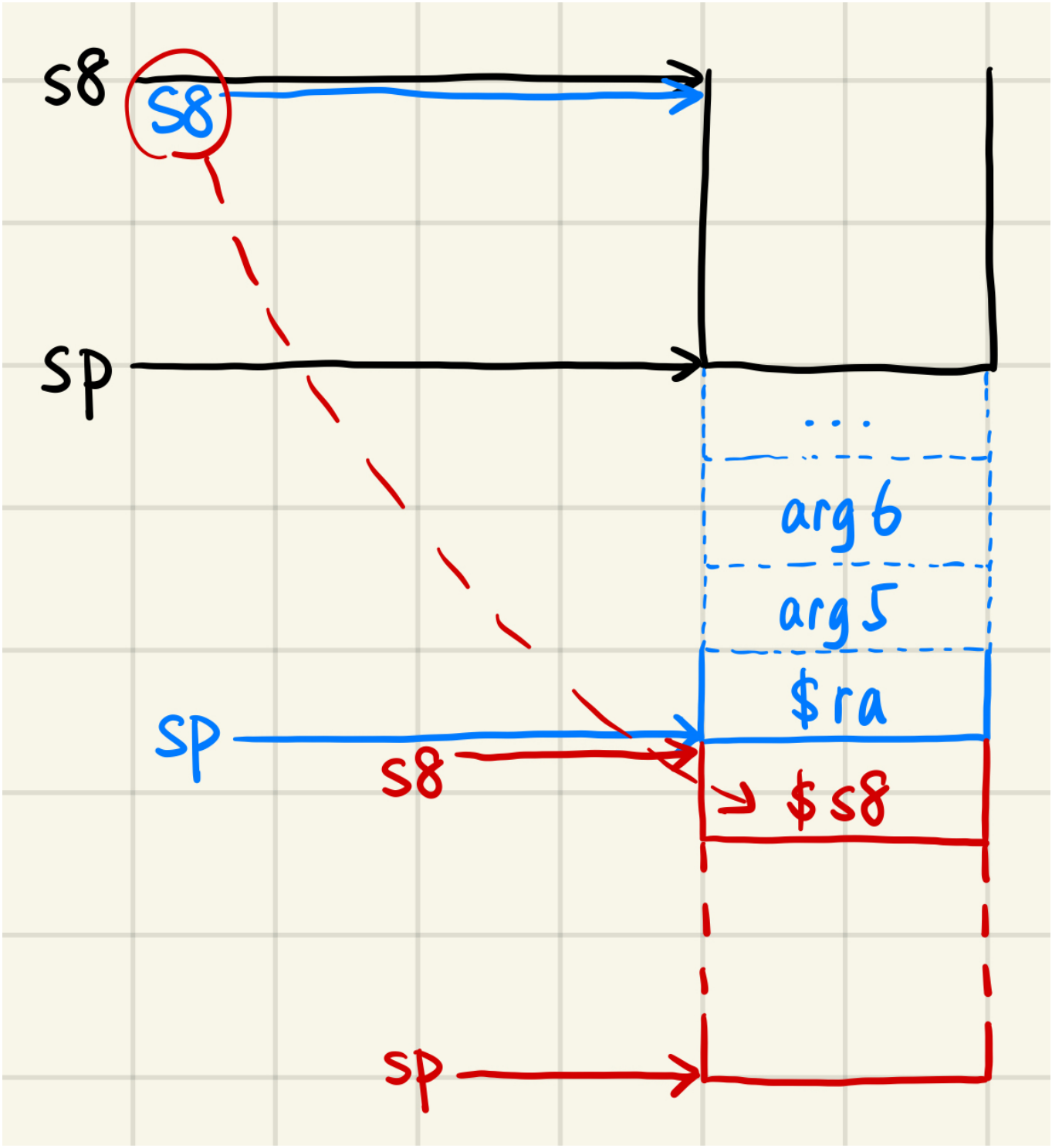
- 所有临时变量的偏移量以 `$sp` 为基准
- 所有栈传递参数的偏移量以 `$s8` 为基准
- 函数调用和返回时，需要保存和恢复 `$sp`、`$s8` 的值

因此过程处理如下：

- **函数开始时，统计本函数内部所需的临时空间大小**，将 `sp` 下移对应距离
- 函数调用时：（如有栈传参）**先固定** `sp`，在 `sp` 下方保存参数，然后将 `sp` 下移对应距离；然后将 `sp` 下移 4 字节，保存 `s8`；然后将 `s8` 移至原 `sp` 处
- 函数返回时：先保存返回值；然后将 `sp` 移至 `s8` 处；然后从 `-4($sp)` 处取出原 `s8` 并恢复

需要注意的是：

- 必须在函数开始时分配好所有本函数的临时变量空间，**不可以用一个开一个，即：不可以遇到新变量时临时** `addi $sp, $sp, -4`。原因在于：编译时无法预测分支是否会执行，因此无法确定分支后的 `sp` 相对于分支前的变量的实际偏移量。
- 整个过程，除函数调用外，**不得对** `sp` **产生任何更改**，否则会导致偏移错误。函数调用也需要在返回后立即将 `sp` 修正为原值。



令人头大的手册大坑

手册中自带的哈希函数，会将 `_t19` 和 `_t25` 映射到同一个值。经测试，这是算法本身导致的碰撞，无法通过增大映射空间的方式解决。