
# 课程项目

燕言言
**QQ: 2214871526**
**邮箱: yanyanthunder@foxmail.com**

## 课程项目

**主要内容：**实验内容是为一个小型的类C语言（C--）实现一个编译器。如果你顺利完成了本实验任务，那么不仅你的编程能力将会得到大幅提高，而且你最终会得到一个比较完整的、能将C--源代码转换成MIPS汇编代码的编译器，所得到的汇编代码可以在SPIM Simulator上运行。

**课程项目总共分为五个阶段：**词法和语法分析、语义分析、中间代码生成、目标代码生成以及中间代码优化。每个阶段的输出是下一个阶段的输入，后一个阶段总是在前一个阶段的基础上完成。其中，目标代码生成以及中间代码优化均基于第三次中间代码生成。

**实验助教：**


燕言言　　QQ：2214871526

邮箱：yanyanthunder@foxmail.com

何天行　　QQ：976792132

邮箱：976792132@qq.com

# 提纲

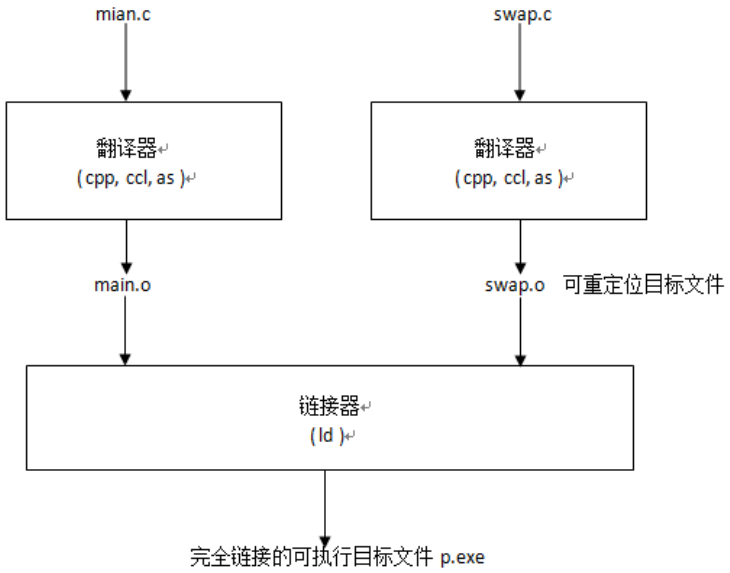
- 课程项目
  - 实验内容
  - 目标代码
  - 实验设计
- QtSPIM模拟器
  - QtSPIM部署及使用

# 实验内容

## 以C语言编译链接为例

- 我们以常见的C语言编译链接过程为例，概要性介绍目标代码生成的相关概念

| | main.c | swap.c |
|---|---|---|
| 1 | void swap(); | int *bufp0 = &buf[0] |
| 2 | | int *bufp1; |
| 3 | int buf[2]={1, 2}; | |
| 4 | | void swap() |
| 5 | int main() | { |
| 6 | { | int temp; |
| 7 | swap(); | bufp1 = &buf[1]; |
| 8 | return 0; | temp = *bufp0; |
| 9 | } | *bufp0 = *bufp1; |
| 10 | | *bufp1 = temp; |
| 11 | | } |




gcc version 7.5.0 (Ubuntu 20.04)

# 实验内容

## C语言编译链接

- 编译。一般而言,大多数编译系统都提供编译驱动程序(complier driver)，其根据用户需求调用语言预处理器,编译器,汇编器和链接器，最终输出一个可以在目标平台上运行的可执行文件
  a) 预处理用于将所有的#include头文件以及宏定义替换成其真正的内容（gcc -E）。驱动程序调用C预处理器(cpp)将源程序main.c翻译成一个ASCII码的中间文件main.i
  b) 编译是将经过预处理之后的程序转换成特定汇编代码的过程（gcc -S）。驱动程序调用C编译器(ccl)将main.i翻译成一个ASCII汇编语言文件main.s
  c) 汇编是将编译生成的汇编代码转换成机器代码，生成目标文件（gcc -c）。驱动程序调用汇编器(as),它将main.s翻译成一个可重定位的目标文件main.o

gcc version 7.5.0 (Ubuntu 20.04)

# 实验内容

## C语言编译链接

- 链接。链接过程将多个目标文件（main.o，swap.o）以及所需的库文件（Linux系统上的静态库.a或者共享库.so等）链接成最终的可执行文件（ld –o xxx xx.o …libraries…）。链接具体过程如下：
  a) 合并段。将多个目标的代码段、数据阶段、bss段等合并在一起，并调整段偏移
  b) 汇总所有符号。每个目标文件在编译时都会生成自己的符号表，链接过程会将所有的符号信息合并起来以便于后续符号解析
  c) 完成符号的重定位。经过合并段并调整段偏移后，输入文件的各个段的虚拟地址已经确定。链接器还需计算各个符号的虚拟地址（比如函数入口和全局变量的虚拟地址调整）。因为每个符号在段内的相对位置是固定的，所以段内各个符号的地址也确定了，，链接器需要为每个符号加上一个偏移量，使其调整到正确的虚拟地址（符号重定位）

gcc version 7.5.0 (Ubuntu 20.04)

# 实验内容

## C--目标代码生成

- 在实验三中间代码生成基础上，实验四的主要内容就是将C--源代码翻译为MIPS32汇编代码
- 准确地说，我们是将实验三实现的中间代码生成器的输出作为实验四目标代码生成器的输入，经过指令选择、寄存器分配，栈管理之后，实现MIPS32汇编代码输出
- 实验四输出的目标代码会在spim模拟器上运行，只要输出结果和预期一致，即可认为完成了目标代码生成器的实现
- 当完成实验四之后，即拥有一个独立编写、可实际运行的C--编译器

# 实验内容

## C--假设

- 输入文件中不包含任何词法、语法或语义错误（函数也必有return语句）
- 不会出现注释、八进制或十六进制整型常数、浮点型常数或者变量
- 整型常数都在16bits位的整数范围内，也就是说你不必考虑如果某个整型常数无法在addi等包含立即数的指令中表示时如何处理
- 不会出现类型为结构体或高维数组（高于1维的数组）的变量。
- 没有全局变量的使用，并且所有变量均不重名，变量的存储空间都放到该变量所在的函数的活动记录中

相较于中间代码，第四次实验输出的目标代码可读性较差，更偏底层

# 实验内容

- **C--假设**
  - 任何函数参数都只能是简单变量，也就是说数组和结构体不会作为参数传入某个函数中
  - 函数不会返回结构体或数组类型的值
  - 函数只会进行一次定义（没有函数声明）

- 在进行实验四之前，请阅读实验指导等相关资料，以确保已经了解MIPS32汇编语言以及SPIM Simulator的使用方法

相较于中间代码，第四次实验输出的目标代码可读性较差，更偏底层

# 实验内容

- 输入格式
  - 程序的输入是一个包含C--源代码的文本文件，程序需要能够接收一个输入文件 名和一个输出文件名作为参数。例如，在Linux命令行下运行
  - ./cc test1.cmm out1.s
  - 即可将输出结果写入当前目录下名为out1.s的文件中
- 输出格式
  - 实验四要求程序将运行结果输出到文件，比如output.s。对于每个输入文件，目标代码生成程序应输出相应的MIPS32汇编代码
  - 测试时将使用SPIM Simulator对输出的汇编代码的正确性进行测试，任何能被SPIM Simulator执行且结果正确的输出都将被接受
  - 尽量先保证目标汇编代码的正确生成

# 实验内容

- 实验四任务分配

注意：实验四无选做内容，大家必须实现实验内容上的所有要求，以将中间代码翻译为正确的目标代码

# 提纲

- 课程项目
  - 实验内容
  - 目标代码
  - 实验设计
- QtSPIM模拟器
  - QtSPIM部署及使用

# 目标代码

## 目标代码生成挑战

- **指令选择（Instruction Selection）问题**。中间代码与目标代码之间并不是严格一一对应的。有可能某条中间代码对应多条目标代码，也有可能多条中间代码对应一条目标代码
- **寄存器分配（Register Allocation）问题**。中间代码中可以使用数目不受限的变量和临时变量，但处理器所拥有的寄存器数量是有限的。RISC(精简指令集计算机)机器的一大特点就是运算指令的操作数总是从寄存器中获得
- **栈管理问题**。中间代码中没有处理有关函数调用的细节。函数调用在中间代码中被抽象为若干条ARG语句和一条CALL语句。但在目标机器上一般不会有专门的器件为函数调用进行参数传递，目标代码中必须借助于寄存器或栈来完成参数传递
- 实验四我们的主要任务就是编写程序来处理这三个问题

# 目标代码

## 指令选择——MIPS32汇编代码

- SPIM Simulator不仅是一个MIPS32的模拟器，也是一个MIPS32的汇编器。想要让SPIM Simulator正常模拟，首先需要为它准备符合格式的MIPS32汇编代码文本文件。非操作系统内核的汇编代码文件必须以.s或者.asm作为文件的后缀名
- 汇编代码由若干代码段和若干数据 段组成，其中代码段以.text开头，数据段以.data开头。汇编代码中的注释以#开头。汇编代码具体格式和内容请参考《实验指导》中的伪指令和《实验内容》中汇编代码示例

# 目标代码

## 指令选择——MIPS32汇编代码

- SPIM Simulator常用的伪指令及其对应的MIPS32指令如下表所示。我们只需要生成可读性更强的伪指令代码即可

| 伪指令 | 描述 | 对应的MIPS32指令 |
|---|---|---|
| li Rdest, imm | 把立即数imm（小于等于oxffff）加载到寄存器Rdest中。 | ori Rdest, $0, imm |
| | 把立即数imm（大于oxffff）加载到寄存器Rdest中。 | lui Rdest, upper(imm)[1]<br>ori Rdest, Rdest, lower(imm) |
| la Rdest, addr | 把地址（而非其中的内容）加载到寄存器Rdest中。 | lui Rdest, upper(addr)<br>ori Rdest, Rdest, lower(addr) |
| move Rdest, Rsrc | 把寄存器Rsrc中的内容移至寄存器Rdest中。 | addu Rdest, Rsrc, $0 |
| bgt Rsrc1, Rsrc2, label | 各种条件分支指令。 | slt $1, Rsrc1, Rsrc2<br>bne $1, $0, label |
| bge Rsrc1, Rsrc2, label | | sle $1, Rsrc1, Rsrc2<br>bne $1, $0, label |
| blt Rsrc1, Rsrc2, label | | sgt $1, Rsrc1, Rsrc2<br>bne $1, $0, label |
| ble Rsrc1, Rsrc2, label | | sge $1, Rsrc1, Rsrc2<br>bne $1, $0, label |

# 目标代码

## 指令选择——MIPS32汇编代码

- 数据段可以为汇编代码中所要用到的常量和全局变量申请空间，其格式为：

  name: storage_type value(s)

- 其中，name代表内存地址（标签）名，storage_type代表数据类型，value代表初始值。常见的storage_type如下表所示：

| storage_type | 描述 |
|---|---|
| .ascii str | 存储str于内存中，但不以null结尾。 |
| .asciiz str | 存储str于内存中，并以null结尾。 |
| .byte b1, b2, …, bn | 连续存储n个字节（8bits位）的值于内存中。 |
| .half h1, h2, …, hn | 连续存储n个半字（16bits位）的值于内存中。 |
| .word w1, w2, …, wn | 连续存储n个字（32bits位）的值于内存中。 |
| .space n | 在当前段分配n个字节的空间。 |

# 目标代码

- 指令选择——MIPS32汇编代码
  - 我们可以在数据段里面分配一些常见类型的连续内存空间并初始化，如下所示

```
1  var1: .word 3              # create a single integer variable with
2                             # an initial value of 3
3  array1: .byte 'a','b'      # create a 2-element character array with
4                             # its elements initialized to a and b
5  array2: .space 40          # allocate 40 consecutive bytes, with storage
6                             # uninitialized; could be used as a 40-element
7                             # character array, or a 10-element integer array
```

# 目标代码

- 指令选择——MIPS32汇编代码
  - SPIM Simulator也为我们提供了与控制台交互的机制，这些机制通过调用syscall的形式体现，如下表所示

| 服务 | Syscall代码 | 参数 | 结果 |
|---|---|---|---|
| print_int | 1 | $a0 = integer | |
| print_string | 4 | $a0 = string | |
| read_int | 5 | | integer (在$v0中) |
| read_string | 8 | $a0 = buffer, $a1 = length | |
| print_char | 11 | $a0 = char | |
| read_char | 12 | | char (在$a0中) |
| exit | 10 | | |
| exit2 | 17 | $a0 = result | |

# 目标代码

## 指令选择——MIPS32汇编代码

- 为实现系统调用，我们需要向寄存器$v0中存入一个代码（int常数），以指定系统调用服务类型。如果还需要额外的参数，还需要提前将其存入指定的寄存器，比如$a0和$a1
- 例如，我们需要向控制台输出一条提示信息，伪代码是print_string(_prompt)
- 对应的伪指令是：

```
1  li $v0, 4
2  la $a0, _prompt
3  syscall
```

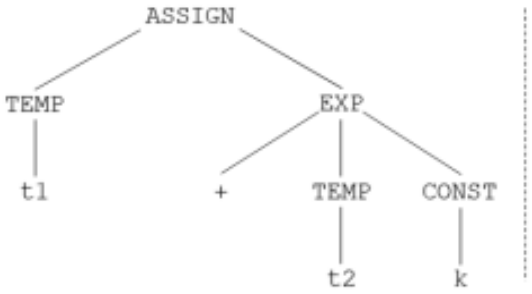
- 其中，_prompt在.data数据段中被定义：

```
1  .data
2  _prompt: .asciiz "Enter an integer:"
3  _ret: .asciiz "\n"
```

# 目标代码

## 指令选择算法

- 指令选择算法可以看成是一个模式匹配的问题。无论中间代码是线形还是树形的，我们都需要在其中找到特定的模式，然后将这些模式对应到目标代码上（可类比将语法树翻译为中间代码的过程）。以树形的IR翻译为例：




树形中间代码

```
addi reg(t1), reg(t2), k
```

```
1  if (current_node -> kind == ASSIGN)
2  {
3    left = current_node -> left;
4    right = current_node -> right;
5    if (left->kind == TEMP && right->kind == EXP)
6    {
7      op1 = right -> op1;
8      op2 = right -> op2;
9      if (right->op == '+' && op1->kind == TEMP && op2->kind == CONST)
10       emit_code("addi " + get_reg(left) + ", " + get_reg(op1) + ", "
11         + get_value(op2));
12   }
13 }
```

模式匹配算法

# 目标代码

## 指令选择算法

- 如果程序使用了线形IR，那么最简单的指令选择方式是逐条将中间代码对应到目标代码上
- 如图是将实验三的中间代码对应到MIPS32指令的一个例子，当然这个翻译方案并不唯一
- 注意，实验四输入中不包含结构体变量和高维数组

| 中间代码 | MIPS32指令 |
|---|---|
| LABEL x: | x: |
| x := #k | li reg(x)[1], k |
| x := y | move reg(x), reg(y) |
| x := y + #k | addi reg(x), reg(y), k |
| x := y + z | add reg(x), reg(y), reg(z) |
| x := y - #k | addi reg(x), reg(y), -k |
| x := y - z | sub reg(x), reg(y), reg(z) |
| x := y * z[2] | mul reg(x), reg(y), reg(z) |
| x := y / z | div reg(y), reg(z)<br>mflo reg(x) |
| x := *y | lw reg(x), 0(reg(y)) |
| *x = y | sw reg(y), 0(reg(x)) |
| GOTO x | j x |
| x := CALL f | jal f<br>move reg(x), $v0 |
| RETURN x | move $v0, reg(x)<br>jr $ra |
| IF x == y GOTO z | beq reg(x), reg(y), z |
| IF x != y GOTO z | bne reg(x), reg(y), z |
| IF x > y GOTO z | bgt reg(x), reg(y), z |
| IF x < y GOTO z | blt reg(x), reg(y), z |
| IF x >= y GOTO z | bge reg(x), reg(y), z |
| IF x <= y GOTO z | ble reg(x), reg(y), z |

# 目标代码

## 寄存器分配——MIPS寄存器

- MIPS体系结构共有32个寄存器，在汇编代码中可以使用$0至$31来表示它们。为了便于表示和记忆，这32个寄存器也拥有各自的别名

| 寄存器编号 | 别名 | 描述 |
|---|---|---|
| $0 | $zero | 常数0。 |
| $1 | $at | (Assembler Temporary) 汇编器保留。 |
| $2 – $3 | $v0 – $v1 | (Values) 表达式求值或函数结果。 |
| $4 – $7 | $a0 – $a3 | (Arguments) 函数的首四个参数（跨函数不保留）。 |
| $8 – $15 | $t0 – $t7 | (Temporaries) 函数调用者负责保存（跨函数不保留）。 |
| $16 – $23 | $s0 – $s7 | (Saved Values) 函数负责保存和恢复（跨函数不保留）。 |
| $24 – $25 | $t8 – $t9 | (Temporaries) 函数调用者负责保存（跨函数不保留）。 |
| $26 – $27 | $k0 – $k1 | 中断处理保留。 |
| $28 | $gp | (Global Pointer) 指向静态数据段64K内存空间的中部。 |
| $29 | $sp | (Stack Pointer) 栈顶指针。 |
| $30 | $s8或$fp | MIPS32作为$s8，GCC作为帧指针。 |
| $31 | $ra | (Return Address) 返回地址。 |


# 目标代码

## 寄存器分配算法

- 寄存器访问性能高。使用寄存器存取数据，速度较快
- RISC机器的一个很显著的特点是，除了load/store之外，其余指令的所有操作数都必须来自寄存器而非内存。但数组和结构体必须放到内存
- 寄存器分配要解决的问题就是：在某个特定程序点上（比如某条汇编代码需要处理数据），选择哪个寄存器来保存哪个变量的值
- 寄存器需要分配使用（实验三中可以使用近乎无限的寄存器、临时变量等等）而不是随意使用的原因是
  - 寄存器数量少，RISC机器只有32个通用寄存器，还有一些被保留无法使用。因此，可供我们随意使用的寄存器只有$t0至$t9及$s0至$s8共19个寄存器

# 目标代码

## 寄存器分配算法——朴素寄存器分配算法

- 朴素寄存器分配算法思想最简单，但也最低效：将所有变量或者临时变量都放在内存里
  - 翻译每一条中间代码之前，我们先加载要用到的变量到寄存器中，得到计算结果后再将其写回内存
- 朴素寄存器分配算法可以保证将中间代码翻译成可以正常运行的目标代码，且实现和调试都特别容易
- 但这个方法最大的问题是对寄存器的利用率较低，它不仅闲置了MIPS提供的大部分通用寄存器（随意使用的$t0至$t9及$s0至$s8），也没有减少那些未被闲置的寄存器对目标代码访存次数
- 优点是实现简单一些，我们后续以朴素寄存器分配为例探讨一下目标代码生成的实现


# 目标代码

## 寄存器分配算法——局部寄存器分配算法

- 寄存器分配困难的原因是寄存器数量有限，迫使许多变量共用同一个寄存器。这就导致变量使用时需要频繁的换入换出
- 局部寄存器分配算法可以合理安排变量对寄存器共用关系以最大程度减少寄存器换入换出代价
  - 局部寄存器分配算法先将整段代码划分成一个个基本块（参考基本块划分算法及实现），在每个基本块内部采用各种启发式原则为块里出现的变量分配寄存器
    - ✓ 如果基本块内变量需要使用寄存器，就从空闲寄存器中分配一个
    - ✓ 如果没有空闲寄存器，选择本基本块内将来用不到或者最久以后才会用到的变量的寄存器溢出到内存
  - 基本块结束时，局部寄存器分配算法和朴素分配算法一样，都需要将基本块中所有修改过的变量都写回内存

# 目标代码

## 寄存器分配算法——局部寄存器分配算法

- 以中间代码 z := x op y为例，其中，op代表一个任意的二元运算符，x和y是op的操作数，z是运算结果。此语句的局部寄存器分配算法框架如下所示
- 其中，Free(r)表示将寄存器r标记为闲置，Ensure保证将操作数x加载到寄存器中，Allocate收集每个变量在程序点使用信息并分配寄存器

```
1  for each operation z = x op y
2    rx = Ensure(x)
3    ry = Ensure(y)
4    if (x is not needed after the current operation)
5      Free(rx)
6    if (y is not needed after the current operation)
7      Free(ry)
8    rz = Allocate(z)
9  emit MIPS32 code for rz = rx op ry
```

z := x + y中间代码翻译框架

```
1  Ensure(x):
2    if (x is already in register r)
3      result = r
4    else
5      result = Allocate(x)
6      emit MIPS32 code [lw result, x]
7    return result
8
9  Allocate(x):
10   if (there exists a register r that currently has not been assigned to
11       any variable)
12     result = r
13   else
14     result = the register that contains a value whose next use is farthest
15         in the future
16     spill result
17   return result
```

辅助函数算法

# 目标代码

## 寄存器分配算法——图染色法

- 局部寄存器分配算法只对一个基本块内的中间代码有效。当我们将其推广到多个基本块时，无法但看中间代码就确定程序的控制流走向（比如基本块最后一条语句是跳转语句，变量的使用信息会随着条件转移的不同而不同）
- 解决方案：将局部寄存器分配算法替换成全局寄存器分配算法。此算法必须能从中间代码的控制流中获取变量的活跃信息，而活跃变量分析就可以为我们提供这些信息

# 目标代码

## 寄存器分配算法——图染色法

- 活跃变量分析（Live-Variable Analysis）关注程序内部程序点p上对于某变量的定义，是否会在某条由p出发的路径上被使用
- 活跃变量：变量x在某一特定的中间代码i处是活跃的当且仅当
  - 在i中用到了变量x的值，则x在i运行之前是活跃的（活跃变量产生）
  - 变量x在i中被赋值且x未被i用到，x在i运行之前是不活跃的（活跃变量消亡）
  - 变量x在i运行之后是活跃的且未在i处给x赋值，x在i前也活跃（活跃变量传递）
  - 变量x在i运行之后活跃，x在i运行之后可能跳转到所有中间代码运行前都是活跃的（活跃变量传递）
- 活跃变量分析就是求解数据流方程，其中in[i]为i运行前活跃变量，out[i]为i运行后活跃变量，use[i]是i使用的变量，def[i]是i赋值变量，succ[i]是i的后续中间代码集合

$$in[i] = use[i] \cup (out[i] - def[i]) 和 out[i] = \cup_{j \in succ[i]} in[j]$$

# 目标代码

## 寄存器分配算法——图染色法

- 经过活跃变量分析之后，我们已经了解每个程序点上哪些变量在将来的控制流中被使用。寄存分配原则可以是同时活跃的变量尽量不要分配相同的寄存器，除非：
  - 赋值操作x := y中，x和y都活跃的情况下，也可以共用寄存器
  - 表达式x := y + z中，如果x不活跃，y活跃。尽管x和y不同时活跃，也避免共用寄存器以防止之后对x的赋值覆盖y的值
- 因此，我们定义变量x和y相互干扰条件为:
  - 存在中间代码i，满足$x \in out[i]$且$y \in out[i]$
  - 存在中间代码i，且i不是赋值操作x := y或y := x，且满足$x \in def[i]$且$y \in out[i]$
- 变量x和y相互干扰，尽量为其分配不同寄存器

# 目标代码

## 寄存器分配算法——图染色法

- 如果将变量或临时变量看做顶点，变量相互干扰则连接一条边，就可以得到一张干涉图。如果对每个顶点涂色（为变量分配寄存器），相邻顶点不能染同一种颜色，那么寄存器分配问题就变成了一个图染色问题
- 对于固定颜色数k（k个寄存器），判断一张干涉图能否被k着色是一个NP-Complete问题[1]。可采用启发式算法：
  1. 如果干涉图包含度小于等于k-1的顶点，将该顶点压入栈并从干涉图中删除
  2. 重复执行1，如果只剩下少于k个顶点，为每一个顶点分配一个颜色，然后依次弹出栈中顶点添加回干涉图，并选择邻居顶点没有使用过的颜色染色
  3. 当删除到某一步，所有顶点都至少包含k个邻居，我们仍可以选择一个顶点删除并压栈，标记其为待溢出的顶点，继续步骤1
  4. 被标记为待溢出顶点在最后被弹出栈，如果邻居少于k，可以为其染色并消除标记。否则，我们将其溢出到内存

1. 浅谈P、NP、NP-Complete和NP-Hard问题. https://zhuanlan.zhihu.com/p/433308577

# 目标代码

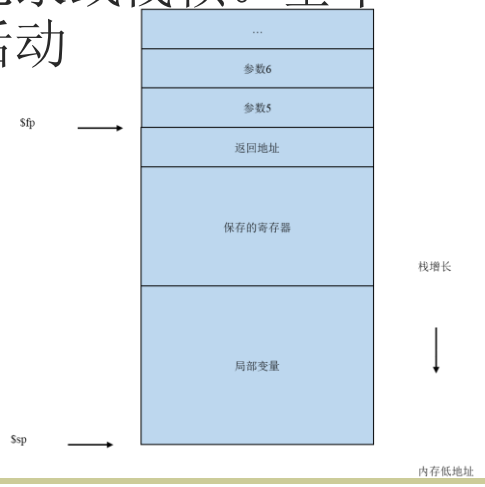
## 栈管理——控制流转移

- 在过程式程序设计语言中，函数调用包括控制流转移和数据流转移两个部分。控制流转移指将程序计数器PC当前值保存到$ra中然后跳转到目标函数第一行，可由jal指令实现
- 我们需要考虑的是函数调用者和被调用者之间的数据流转移
  - 调用者为函数传参，然后将控制流转移到被调用函数第一行
  - 被调用者需要将返回值保存到某个位置，然后将控制流转移回调用者处
- MIPS32中，函数调用使用jal指令，返回使用jr指令
  - 参数少于4个，使用$a0至$a3。多于4个，则前4个保存在寄存器，其他压栈
  - C--所有函数只返回一个整数，放到$v0即可

# 目标代码

## 栈管理——栈

- 另一个重要概念是栈。栈本质上就是按照后进先出原则维护的一块内存区域。除了参数传递，栈还有如下功能：
  - 在函数中使用jal调用另一个函数，$ra会被覆盖（需要调用者保存）
  - 寄存器分配过程溢出局部变量（C--没有全局变量）到栈中
  - 数组和结构体（虽然没有结构体）被分配到内存中（在栈上）
- 每个函数占用一块单独内存空间，被称为活动记录或栈帧。基本结构如图所示。栈指针$sp指向栈顶，$fp指向活动记录的底部



# 目标代码

## 栈管理——寄存器保存

- 如果函数f调用了另一个函数g，f就是调用者，g是被调用者。控制流从调用者转移到被调用者之后，可能使用一些原先保存着有用信息的寄存器，被调用者需要先将这些寄存器保存到栈中。调用者与被调用者哪个负责保存这些寄存器呢？
- MIPS32采用了一种调用者和被调用者共同保存的策略：
  - $t0至$t9由调用者负责保存
  - $s0至$s8由被调用者保存
- 调用者保存的寄存器在函数调用前后可能发生改变，被调用者保存的寄存器一定不会发生改变
  - $t0至$t9应分配给短期使用的变量或临时变量
  - $s0至$s8应分配给生存期长的，比如跨越了函数调用的变量或临时变量

# 提纲

- 课程项目
  - 实验内容
  - 目标代码
  - 实验设计
- QtSPIM模拟器
  - QtSPIM部署及使用

# 实验设计——目标代码生成

## 目标代码生成

- 左图所示的C--源码是实验内容样例1。右图是其对应的目标代码，包含全局符号、read及write函数定义

```
1   int main()
2   {
3       int a = 0, b = 1, i = 0, n;
4       n = read();
5       while (i < n)
6       {
7           int c = a + b;
8           write(b);
9           a = b;
10          b = c;
11          i = i + 1;
12      }
13      return 0;
1   }
```

实验手册样例1

```
1   .data
2   _prompt: .asciiz "Enter an integer:"
3   _ret: .asciiz "\n"
4   .globl main
5   .text
6   read:
7     li $v0, 4
8     la $a0, _prompt
9     syscall
10    li $v0, 5
11    syscall
12    jr $ra
13
14  write:
15    li $v0, 1
16    syscall
17    li $v0, 4
18    la $a0, _ret
19    syscall
20    move $v0, $0
21    jr $ra
```

read和write函数声明

# 实验设计——目标代码生成

## 目标代码生成

- 局部变量：a($t5)，b($t4)，i($t3)，n($t1)。其中n是通过调用read函数获取用户输入，然后存到$t1中

```
 1  int main()
 2  {
 3      int a = 0, b = 1, i = 0, n;
 4      n = read();
 5      while (i < n)
 6      {
 7          int c = a + b;
 8          write(b);
 9          a = b;
10          b = c;
11          i = i + 1;
12      }
13      return 0;
1   }
```

```
22  main:
23    li $t5, 0
24    li $t4, 1
25    li $t3, 0
26    addi $sp, $sp, -4
27    sw $ra, 0($sp)
28    jal read
29    lw $ra, 0($sp)
30    addi $sp, $sp, 4
31    move $t1, $v0
32    move $t2, $t1
33  label1:
34    blt $t3, $t2, label2
35    j label3
36  label2:
37    add $t1, $t5, $t4
38    move $a0, $t4
39    addi $sp, $sp, -4
40    sw $ra, 0($sp)
41    jal write
42    lw $ra, 0($sp)
43    addi $sp, $sp, 4
44    move $t5, $t4
45    move $t4, $t1
46    addi $t1, $t3, 1
47    move $t3, $t1
48    j label1
49  label3:
50    move $v0, $0
51    jr $ra
```

实验手册样例1

局部变量定义

# 实验设计——目标代码生成

## 目标代码生成

- 循环处理：while条件语句(label1)，i($t3)，n($t2)，循环体(label2)，循环外(label3)

```
 1  int main()
 2  {
 3      int a = 0, b = 1, i = 0, n;
 4      n = read();
 5      while (i < n)
 6      {
 7          int c = a + b;
 8          write(b);
 9          a = b;
10          b = c;
11          i = i + 1;
12      }
13      return 0;
1   }
```

```
22  main:
23    li $t5, 0
24    li $t4, 1
25    li $t3, 0
26    addi $sp, $sp, -4
27    sw $ra, 0($sp)
28    jal read
29    lw $ra, 0($sp)
30    addi $sp, $sp, 4
31    move $t1, $v0
32    move $t2, $t1
33  label1:
34    blt $t3, $t2, label2
35    j label3
36  label2:
37    add $t1, $t5, $t4
38    move $a0, $t4
39    addi $sp, $sp, -4
40    sw $ra, 0($sp)
41    jal write
42    lw $ra, 0($sp)
43    addi $sp, $sp, 4
44    move $t5, $t4
45    move $t4, $t1
46    addi $t1, $t3, 1
47    move $t3, $t1
48    j label1
49  label3:
50    move $v0, $0
51    jr $ra
```

实验手册样例1　　循环处理

# 实验设计——目标代码生成

## 目标代码生成

- 循环体分析：L7赋值(汇编L37)，L8输出(汇编L38-L43)，L9(汇编44)，L10(汇编L45)，L11(汇编L46-L47)

```
 1  int main()
 2  {
 3      int a = 0, b = 1, i = 0, n;
 4      n = read();
 5      while (i < n)
 6      {
 7          int c = a + b;
 8          write(b);
 9          a = b;
10          b = c;
11          i = i + 1;
12      }
13      return 0;
1   }
```

```
22  main:
23    li $t5, 0
24    li $t4, 1
25    li $t3, 0
26    addi $sp, $sp, -4
27    sw $ra, 0($sp)
28    jal read
29    lw $ra, 0($sp)
30    addi $sp, $sp, 4
31    move $t1, $v0
32    move $t2, $t1
33  label1:
34    blt $t3, $t2, label2
35    j label3
36  label2:
37    add $t1, $t5, $t4
38    move $a0, $t4
39    addi $sp, $sp, -4
40    sw $ra, 0($sp)
41    jal write
42    lw $ra, 0($sp)
43    addi $sp, $sp, 4
44    move $t5, $t4
45    move $t4, $t1
46    addi $t1, $t3, 1
47    move $t3, $t1
48    j label1
49  label3:
50    move $v0, $0
51    jr $ra
```

实验手册样例1

赋值及输出语句处理

# 实验设计——函数入口

## 输入输出处理

- 目标代码生成器接受一个C--源代码及一个输出目标代码文件名。词法、语法、语义及中间代码生成后，最终生成目标代码

```
 1  int main(int argc, char** argv) {
 2      if (argc <= 1) return 1;
 3      FILE* f = fopen(argv[1], "r");
 4      if (!f) {   perror(argv[1]);   return 1; }
 5      yyrestart(f);
 6      yyparse();  // 语法分析
 7      if (!errorflag) semantic(Root); // 语义分析
 8      // 中间代码生成
 9      CodeList codelisthead = Intercode(Root);
10      FILE* asmff;
11      if(argv[2] == NULL)asmff = fopen("output.s", "w");
12      else asmff =fopen(argv[2], "w");
13      // 目标代码生成
14      generate_asm(codelisthead, ff);
15      fclose(ff);
16      return 0;
17  }
```

main函数定义

# 实验设计——描述符

## 数据结构

- 我们定义变量和寄存器的描述符结构，用于生成目标代码时记录寄存器中存储的变量信息和变量存放在寄存器中的信息

```
 1  struct _VarStructure{
 2            变量名;
 3            变量存放的寄存器;
 4            变量在内存中的存储位置;
 5            变量链表;
 6  };
 7  typedef struct _VarStructure* VarStructure;
 8
 9  struct _Register{
10            寄存器名;
11            关联的变量VarStructure var;
12  };
13  typedef struct _Register* Register;
```

变量及结构体描述符信息

# 实验设计——指令集选择算法

- 目标代码生成入口
  - 我们在generate_asm执行之前，先定义寄存器、变量链表、堆栈及函数调用等信息。然后，输出目标代码文件数据段及read定义

打印.data及.text中的固定内容

```
1   Register r[32]; // 定义寄存器
2   Var varlist = NULL;
3   记录堆栈偏移位置、寄存器及函数实参或形参数目;
4   void generate_asm(CodeList curcode,FILE* file){
5
6       // 打印目标代码数据段
7       fprintf(file,".data\n_prompt: .asciiz \"Enter an
8   integer:\"\n_ret: .asciiz \"\\n\"\n.globl main\n.text\n");
9       // 打印read函数定义
10      fprintf(file,"\nread:\n");
11      fprintf(file,"\tli $v0, 4\n");
12      fprintf(file,"\tla $a0, _prompt\n");
13      fprintf(file,"\tsyscall\n");
14      fprintf(file,"\tli $v0, 5\n");
15      fprintf(file,"\tsyscall\n");
16      fprintf(file,"\tjr $ra\n");
        …
36  }
```

# 实验设计——指令集选择算法

- 目标代码生成入口
  - 目标代码文件的头部信息可以固定，直接写入.data .text中的read和write函数即可

```
 1  Register r[32]; // 定义寄存器
 2  Var varlist = NULL;
 3  记录堆栈偏移位置、寄存器及函数实参或形参数目;
 4  void generate_asm(CodeList curcode,FILE* file){
        …
17      // 打印write函数定义
18      fprintf(file,"\nwrite:\n");
19      fprintf(file,"\tli $v0, 1\n");
20      fprintf(file,"\tsyscall\n");
21      fprintf(file,"\tli $v0, 4\n");
22      fprintf(file,"\tla $a0, _ret\n");
23      fprintf(file,"\tsyscall\n");
24      fprintf(file,"\tmove $v0, $0\n");
25      fprintf(file,"\tjr $ra\n\n");
        …
36  }
```

打印write函数定义

# 实验设计——指令集选择算法

- 目标代码生成入口
  - 写完.data段及.text段的read和write后，我们可以初始化MIPS32寄存器。最后，依次遍历中间代码生成目标代码

```
 1   Register r[32]; // 定义寄存器
 2   Var varlist = NULL;
 3   记录堆栈偏移位置、寄存器及函数实参或形参数目;
 4   void generate_asm(CodeList curcode,FILE* file){
         …
26       for(int i = 0; i<32; i++){
27           Register x = (Register)malloc(sizeof(struct _Register));
28           设置寄存器名称;
29           设置寄存器关联的变量，初始可以置为空;
30           r[i] = x;
31       }
32       while(curcode != NULL){
33           generate_IR_asm(curcode->code); // 遍历每一条IR，生成目标代码
34           获取下一条IR;
35       }
36   }
```

设置寄存器数组信息并翻译每一条中间代码

# 实验设计——指令集选择算法

## 中间代码翻译

- 函数generate_IR_asm会根据当前中间代码的种类生成对应的目标代码。首先，我们处理label和函数定义（用户自定义函数）

```
1   void generate_IR_asm(InterCode intercode){
2       switch(intercode->kind){
3       case IR_LABEL:
4           向目标代码文件中写入label名;
5           break;
6       case IR_FUNC:
7           向目标代码文件中写入文件名;
8           fprintf(file, "\tsubu $sp, $sp, 4\n"); // 开辟栈空间
9           fprintf(file, "\tsw $fp, 0($sp)\n");   // 保存$fp
10          fprintf(file, "\tmove $fp, $sp\n");    // 重新设置$fp
11          申请一段较大的栈空间（可通过$fp加上一段偏移寻址），保存局部变量等;
12          初始化堆栈偏移量，函数形参个数等;
13          break;
14      …
    }
```

处理label和函数定义

# 实验设计——指令集选择算法

## 中间代码翻译

- 其次，我们处理赋值语句(x := #k, x := y, x := *y, *x := y)，对应四种指令(li, move, lw, sw)。使用朴素寄存器分配算法

处理赋值语句

```
1   void generate_IR_asm(InterCode intercode){
2       switch(intercode->kind){
3       case IR_ASSIGN:
4       {
5           获取赋值语句左右操作数left和right;
6           if(right->kind == OP_CONSTANT){              // x := #k
7               int x = getReg(left);                    // 获取寄存器并保存常量
8               fprintf(file, "\tli %s, %d\n", 寄存器x的名称, 常量值);
9               将寄存器x的内容溢出到栈中并退出;
10          };
11          if(left->kind==OP_ADDRESS && right->kind!=OP_ADDRESS)  { // x := y
12              为左右操作数分配寄存器;
13              fprintf(file, "\tmove %s, %s\n",寄存器x y的名称);
14              将寄存器x关联的变量值溢出到栈中记录偏移量，并退出;
15          }
16        …}
17      …
18  }
```

# 实验设计——指令集选择算法

## 中间代码翻译

- 接下来，我们处理算术运算符(x := y + z, x := y - z, x := y * z, x := y / z)，对应四种指令(add/addi, sub, mul, div和mflo)

算术运算符处理

```
1  void generate_IR_asm(InterCode intercode){
2      switch(intercode->kind){
3      case (IR_PLUS|IR_MINUS|IR_MUL|IR_DIV): // 篇幅有限，将其放一起
4      {
5          获取三个操作数result, lhs, rhs;
6          为其分配寄存器x, y, z，如果左右操作数是常数，将其加载到寄存器中;
7          switch (intercode->kind){
8            case IR_PLUS:
9              fprintf(file,"\tadd %s, %s, %s\n", x y z寄存器名);退出;
10           case IR_MINUS:
11             fprintf(file,"\tsub %s, %s, %s\n", x y z寄存器名);退出;
12           case IR_MUL:
13                fprintf(filex,"\tmul %s, %s, %s\n", x y z寄存器名); 退出;
14           case IR_DIV:
15             fprintf(file,"\tdiv %s, %s\n", y z寄存器名);
16             fprintf(file,"\tmflo %s\n", 寄存器名);}
17           将寄存器x关联的变量值溢出到栈中记录偏移量，并退出;
18     }
```

# 实验设计——指令集选择算法

## 中间代码翻译

- 然后，我们处理条件跳转(IF x (==|!=|>|<|>=|<=) y GOTO z)，对应六种指令(beq,bne,bgt,blt,bge,ble)

条件跳转处理

```
1  void generate_IR_asm(InterCode intercode){
2      switch(intercode->kind){
3      case IR_IFGOTO:
4      {
5          获取条件表达式左右操作数;获取比较运算符名称;获取跳转到的标签名;
6          为其分配寄存器x, y, 如果左右操作数是常数，将其加载到寄存器中;
7           if(strcmp(op, "==")==0)
8              fprintf(file, "\tbeq %s, %s, %s\n",x y寄存器名和标签名);
9           else if(strcmp(op, "!=")==0)
10             fprintf(file, "\tbne %s, %s, %s\n", x y寄存器名和标签名);
11          else if(strcmp(op, ">")==0)
12             fprintf(file, "\tbgt %s, %s, %s\n", x y寄存器名和标签名);
13          else if(strcmp(op, "<")==0)
14             fprintf(file, "\tblt %s, %s, %s\n", x y寄存器名和标签名);
15          else if(strcmp(op, ">=")==0) // "<="与其类似
16             fprintf(file, "\tbge %s, %s, %s\n", x y寄存器名和标签名);
17          退出;
18       }
```

# 实验设计——指令集选择算法

- 中间代码翻译
  - 我们处理函数调用(x := CALL f)及实参传递(ARG x)。函数调用对应两种指令(jal,move)，参数传递需要借助寄存器或者栈

```
 1  void generate_IR_asm(InterCode intercode){
 2      switch(intercode->kind){
 3      case IR_ARG:
 4      {
 5          找到参数对应的变量arg;
 6          fprintf(file, "\tlw $s0, %d($fp)\n", arg在栈中相对于$fp的偏移量);
 7          fprintf(file, "\tsubu $sp, $sp, 4\n");
 8          fprintf(file, "\tsw $s0, 0($sp)\n");
 9          argNum++;
10          break;
11      }
        …
    }
```

实参传递

# 实验设计——指令集选择算法

## 中间代码翻译

- 我们处理函数调用(x := CALL f)及实参传递(ARG x)。函数调用对应两种指令(jal,move)，参数传递需要借助寄存器或者栈

函数调用

```
1  void generate_IR_asm(InterCode intercode){
2      switch(intercode->kind){
3      case IR_CALL:
4      {
5          fprintf(file, "\tli $v1,%d\n",argNum*4); // 实参占用的栈空间
6          fprintf(file, "\tsubu $sp, $sp, 4\n");
7          fprintf(file, "\tsw $v1, 0($sp)\n");      // 将实参占用栈空间保存在栈上
8          argNum = 0;                               // 调用前处理ARG空间
9          fprintf(file,"\tsubu $sp, $sp, 4\n");
10         fprintf(file,"\tsw $ra, 0($sp)\n");       // 保存$ra寄存器
11         为返回值分配一个寄存器x;
12         fprintf(file, "\tjal %s\n", 函数名);
13         fprintf(file, "\tmove %s, $v0\n", 寄存器x的名字); 将x溢出到栈上;
14         fprintf(file,"\tlw $ra, 0($sp)\n");       // 恢复$ra的值
15         fprintf(file,"\taddi $sp, $sp, 4\n");
16         回收实参占用的栈空间并退出;
17     }
18 }
```

# 实验设计——指令集选择算法

- 中间代码翻译
  - 我们接着处理read函数调用(READ x)

```
 1    void generate_IR_asm(InterCode intercode){
 2        switch(intercode->kind){
 3        case IR_READ:
 4        {
 5            fprintf(file, "\taddi $sp, $sp, -4\n");
 6            fprintf(file, "\tsw $ra, 0($sp)\n");    // 保存$ra寄存器内容
 7            fprintf(file, "\tjal read\n");
 8            为read参数x分配寄存器;
 9            fprintf(file, "\tmove %s, $v0\n",寄存器x的名称);
10            将x关联到的变量值溢出到栈上并标记偏移量;
11            fprintf(file, "\tlw $ra, 0($sp)\n");
12            fprintf(file, "\taddi $sp, $sp, 4\n");  // 恢复$ra寄存器
13            退出;
14        }
```

read 函数调用

# 实验设计——指令集选择算法

## 中间代码翻译

- 我们接着处理write函数调用(WRITE x)。需要考虑参数种类

```
1  void generate_IR_asm(InterCode intercode){
2      switch(intercode->kind){
3      case IR_WRITE:
4      {
5          fprintf(file,"\taddi $sp, $sp, -4\n");
6          fprintf(file,"\tsw $ra, 0($sp)\n");    // 保存$ra寄存器内容
7          为x分配寄存器;
8          if(intercode->u.op->kind==OP_VARIABLE || intercode->u.op->kind ==OP_TEMP){
9              fprintf(file,"\tmove $a0, %s\n", 寄存器x的名称);
10         }else if(intercode->u.op->kind ==OP_ADDRESS || intercode->u.op->kind == OP_ARR){
11             fprintf(file, "\tlw $a0, 0(%s)\n", 变量或者数组名);
12         }
13         fprintf(file,"\tjal write\n");   // 跳转到write函数
14         将寄存器x关联的变量值溢出到栈中记录偏移量;
15         fprintf(file,"\tlw $ra, 0($sp)\n");
16         fprintf(file,"\taddi $sp, $sp, 4\n");    // 恢复$ra内容
17         退出;
18     }
```

read 函数调用

# 提纲

- 课程项目
  - 实验内容
  - 目标代码
  - 实验设计
- QtSPIM模拟器
  - QtSPIM部署及使用

# QtSPIM部署及使用

## 命令行版spim安装

- 命令行版本的spim安装及使用较为简单，我们可以直接通过以下命令安装:

  sudo apt install spim

- 验证密码后，确认安装即可。运行命令为

  spim –file xx.s

```
lulu@ubuntu:~/lab4$ sudo apt install spim
[sudo] password for lulu:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  gir1.2-goa-1.0 libfwupdplugin1 libxmlb1
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  xfonts-100dpi xfonts-75dpi
The following NEW packages will be installed:
  spim xfonts-100dpi xfonts-75dpi
0 upgraded, 3 newly installed, 0 to remove and 13 not upgraded.
Need to get 7,424 kB of archives.
```

```
Need to get 7,424 kB of archives.
After this operation, 8,364 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 https://mirrors.nju.edu.cn/ubuntu focal/universe amd64 xfonts-100dpi all 1
:1.0.4+nmu1 [3,823 kB]
Get:2 https://mirrors.nju.edu.cn/ubuntu focal/universe amd64 xfonts-75dpi all 1:
1.0.4+nmu1 [3,368 kB]
Get:3 https://mirrors.nju.edu.cn/ubuntu focal/universe amd64 spim amd64 8.0+dfsg
-6.1 [233 kB]
Fetched 7,424 kB in 1s (13.4 MB/s)
Selecting previously unselected package xfonts-100dpi.
(Reading database ... 190997 files and directories currently installed.)
Preparing to unpack .../xfonts-100dpi_1%3a1.0.4+nmu1_all.deb ...
Unpacking xfonts-100dpi (1:1.0.4+nmu1) ...
Selecting previously unselected package xfonts-75dpi.
Preparing to unpack .../xfonts-75dpi_1%3a1.0.4+nmu1_all.deb ...
Unpacking xfonts-75dpi (1:1.0.4+nmu1) ...
Selecting previously unselected package spim.
Preparing to unpack .../spim_8.0+dfsg-6.1_amd64.deb ...
Unpacking spim (8.0+dfsg-6.1) ...
Setting up xfonts-100dpi (1:1.0.4+nmu1) ...
Setting up spim (8.0+dfsg-6.1) ...
```

# QtSPIM部署及使用

## GUI版spim安装

- 首先，在浏览器中输入SPIM Simulator的官方地址

  http://pages.cs.wisc.edu/~larus/spim.html

### SPIM A MIPS32 Simulator

James Larus
spim@larusstone.org

Microsoft Research
*Formerly:* Professor, Computer Sciences Department, University of Wisconsin-Madison

***spim* has moved to SourceForge.org. New versions of *spim* are at that site. This site only contains old versions of the simulator.**

Why move? SourceForge offers a number of services, such as a source code repository and collaborative tools, that make it easier to share code with other developers. Now that *spim* has an open source license, the time has come to move off a 20-year old web site and join the 21st century.

**Table of Contents**



*spim* is a self-contained simulator that runs MIPS32 programs. It reads and executes assembly language programs written for this processor. *spim* also provides a simple debugger and minimal set of operating system services. *spim* does not execute binary (compiled) programs.

*spim* implements almost the entire MIPS32 assembler-extended instruction set. (It omits most floating point comparisons and rounding modes and the memory system page tables.) The MIPS architecture has several variants that differ in various ways (e.g., the MIPS64 architecture supports 64-bit integers and addresses), which means that *spim* will not run programs compiled for all MIPS processors. MIPS compilers also generate a number of assembler directives that *spim* cannot process. These directives usually can be safely ignored.

Earlier versions of *spim* (before 7.0) implemented the MIPS-I instruction set used on the MIPS R2000/R3000 computers. This architecture is obsolete (though, never surpassed for its simplicity and elegance). *spim* now supports the more modern MIPS32 architecture, which is the MIPS-I instruction set augmented with a large number of occasionally useful instructions. MIPS code from earlier versions of SPIM should run without changes, except code that handles exceptions and interrupts. This part of the architecture changed over time (and was poorly implemented in earlier versions of *spim*). This type of code will need to be updated. Examples of new exception handling are in the files: exceptions.s and Tests/tt.io.s.

**What's New?**

*QtSpim* is a new user interface for *spim* built on the Qt UI framework. Qt is cross-platform, so the same user iterface and same code will run on Windows, Linux, and Mac OS X (yeah!). Moreover, the interface is clean and up-to-date (unlike the archaic X windows interface).

*Spim* has moved to SourceForge! The source code for all version of *spim* are in an SVN repository and compiled version are available for download. There is also a bug tracker and discussion forum.

**QtSpim**

The newest version of *spim* is called *QtSpim*, and unlike all of the other version, it runs on Microsoft Windows, Mac OS X, and Linux—the same source code and the same user interface on all three platforms! *QtSpim* is the version of *spim* that currently being actively maintaned. The other versions are still available, but please stop using them and move to *QtSpim*. It has a modern user interface, extensive help, and is consistent across all three platforms. *QtSpim* makes my life far easier, and will likely improve yours and your students' experience as well.

A compiled, immediately installable version of *QtSpim* is available for Microsoft Windows, Mac OS X, and Linux can be downloaded from: https://sourceforge.net/projects/spimsimulator/files/. Full source code is also available (to compile *QtSpim*, you need Nokia's Qt framework, a very nice cross-platform UI framework that can be downloaded from here).

# QtSPIM部署及使用

- **GUI版spim安装**
  - QtSPIM可执行文件下载链接为：

    https://sourceforge.net/projects/spimsimulator/files/
  - Ubuntu 20.04可以选择下载较新的版本，比如9.1.23或者9.1.24

| | | |
|---|---|---|
| qtspim_9.1.9_linux64.deb | 2013-01-23 | 1.1 MB |
| qtspim_9.1.9_linux32.deb | 2013-01-23 | 1.1 MB |
| PCSpim_9.1.9.zip | 2013-01-20 | 5.7 MB |
| QtSpim_9.1.9_Windows.zip | 2013-01-20 | 14.1 MB |

| | | |
|---|---|---|
| QtSpim_9.1.24_mac.mpkg.zip | 2023-08-04 | 22.7 MB |
| qtspim_9.1.24_linux64.deb | 2023-08-04 | 18.5 MB |
| QtSpim_9.1.24_Windows.msi | 2023-08-04 | 21.5 MB |
| QtSpim_9.1.23_Windows.msi | 2021-12-06 | 21.5 MB |
| qtspim_9.1.23_linux64.deb | 2021-12-06 | 18.5 MB |
| QtSpim_9.1.23_mac.mpkg.zip | 2021-12-06 | 22.2 MB |
| qtspim_9.1.22_linux64.deb | 2020-05-09 | 18.5 MB |

# QtSPIM部署及使用

## GUI版spim安装

- QtSPIM可执行文件下载链接为：

  https://sourceforge.net/projects/spimsimulator/files/

- Ubuntu 20.04可以选择下载较新的版本，比如9.1.23或者9.1.24。例如，我下载了qtspim_9.1.23_linux64.deb，其安装命令为：

  sudo dpkg –i qtspim_9.1.23_linux64.deb

- 安装之后，运行qtspim即可启动SPIM Simulator GUI版本

```
lulu@ubuntu:~/lab4$ sudo dpkg -i qtspim_9.1.23_linux64.deb
[sudo] password for lulu:
Selecting previously unselected package qtspim.
(Reading database ... 191758 files and directories currently installed.)
Preparing to unpack qtspim_9.1.23_linux64.deb ...
Unpacking qtspim (9.1.23) ...
Setting up qtspim (9.1.23) ...
Processing triggers for gnome-menus (3.36.0-1ubuntu1) ...
Processing triggers for desktop-file-utils (0.24-1ubuntu3) ...
Processing triggers for mime-support (3.64ubuntu1) ...
Processing triggers for man-db (2.9.1-1) ...
lulu@ubuntu:~/lab4$ whereis qtspim
qtspim: /usr/bin/qtspim /usr/lib/qtspim /usr/share/qtspim /usr/share/man/man1/qt
spim.1.gz
lulu@ubuntu:~/lab4$ qtspim
```

# QtSPIM部署及使用

- **spim安装信息查看**
  - 查看Ubuntu已经安装的spim软件信息：

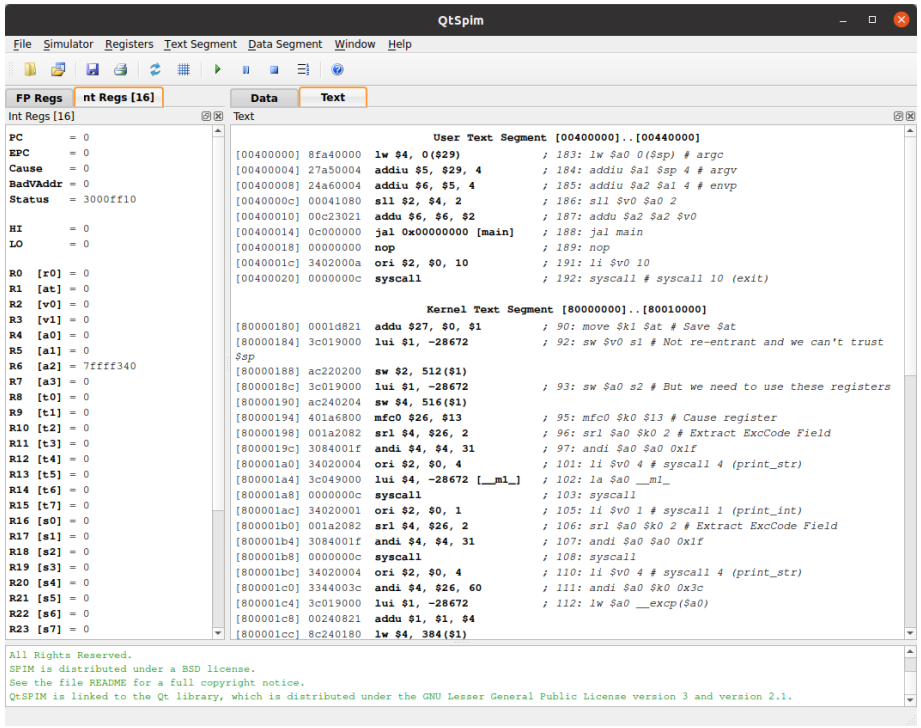
    apt list --installed | grep spim
  - 输出以下信息：

```
lulu@ubuntu:~/lab4/test$ apt list --installed | grep spim

WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

qtspim/now 9.1.23 amd64 [installed,local]
spim/focal,now 8.0+dfsg-6.1 amd64 [installed]
```

# QtSPIM部署及使用

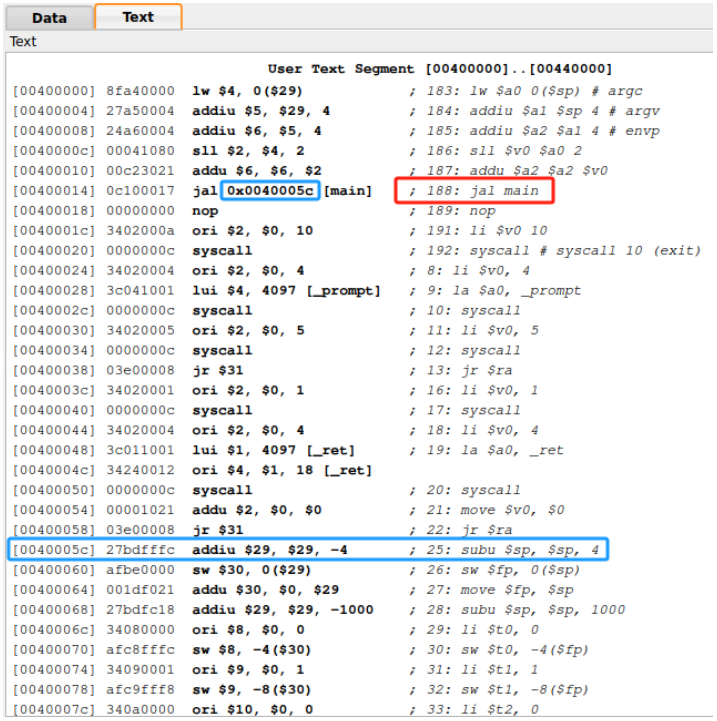
## GUI版spim使用

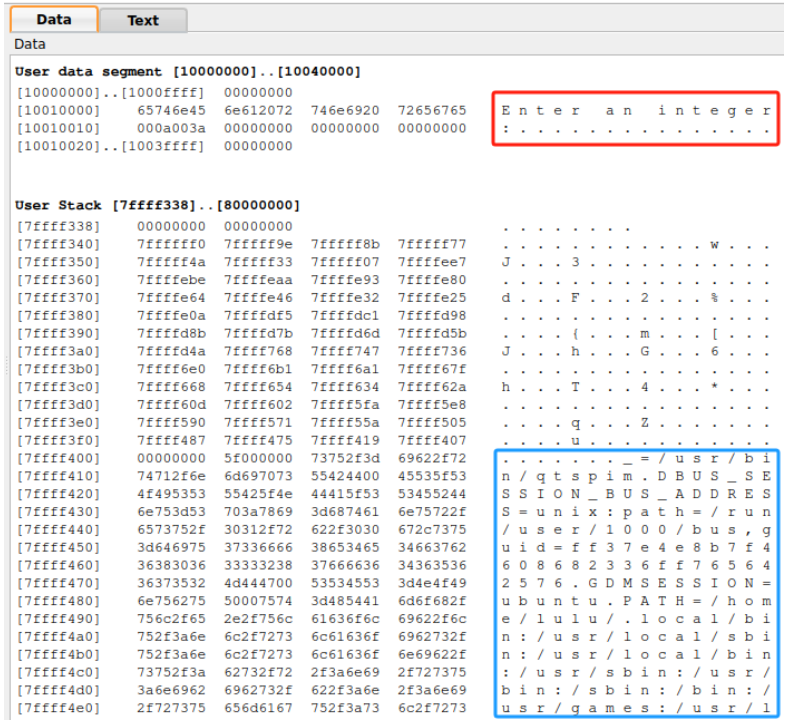
- SPIM Simulator GUI版本运行界面如左图（代码段Text）所示，控制台如右图所示。其中，控制台用于汇编代码输入和输出：




# QtSPIM部署及使用

- **GUI版spim使用**
  - 加载生成的汇编代码文件，Text标签中是代码段内容，如左图所示，Data标签中是数据段内容，如右图所示



# QtSPIM部署及使用

- GUI版spim使用
  - 实验内容样例一执行结果

Data | Text

Text

```
                    User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)            ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4         ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4          ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2            ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2          ; 187: addu $a2 $a2 $v0
[00400014] 0c100017  jal 0x0040005c [main]    ; 188: jal main
[00400018] 00000000  nop                      ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10           ; 191: li $v0 10
[00400020] 0000000c  syscall                  ; 192: syscall # syscall 10 (exit)
[00400024] 34020004  ori $2, $0, 4            ; 8: li $v0, 4
[00400028] 3c041001  lui $4, 4097 [_prompt]   ; 9: la $a0, _prompt
[0040002c] 0000000c  syscall                  ; 10: syscall
[00400030] 34020005  ori $2, $0, 5            ; 11: li $v0, 5
[00400034] 0000000c  syscall                  ; 12: syscall
[00400038] 03e00008  jr $31                   ; 13: jr $ra
[0040003c] 34020001  ori $2, $0, 1            ; 16: li $v0, 1
[00400040] 0000000c  syscall                  ; 17: syscall
[00400044] 34020004  ori $2, $0, 4            ; 18: li $v0, 4
[00400048] 3c011001  lui $1, 4097 [_ret]      ; 19: la $a0, _ret
[0040004c] 34240012  ori $4, $1, 18 [_ret]
[00400050] 0000000c  syscall                  ; 20: syscall
[00400054] 00001021  addu $2, $0, $0          ; 21: move $v0, $0
[00400058] 03e00008  jr $31                   ; 22: jr $ra
[0040005c] 27bdfffc  addiu $29, $29, -4       ; 25: subu $sp, $sp, 4
```

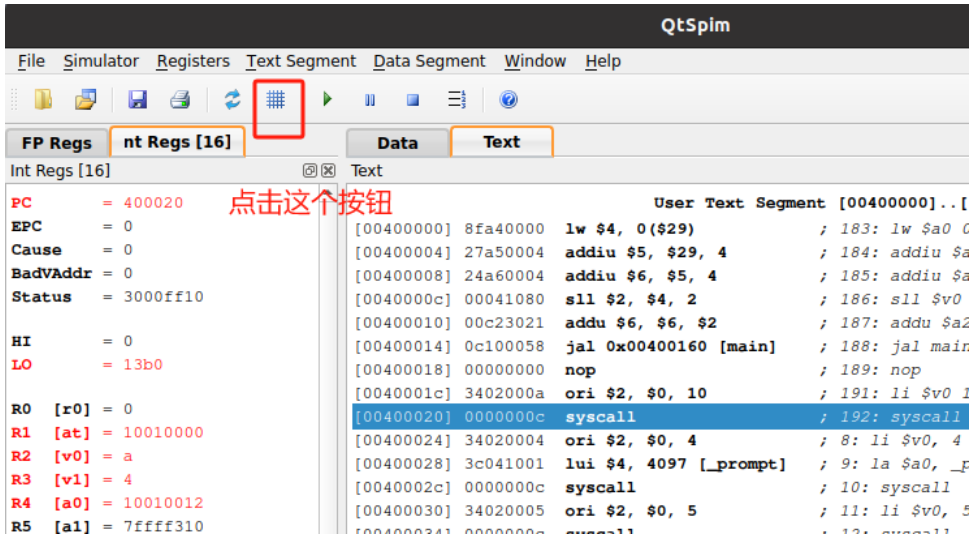
Console

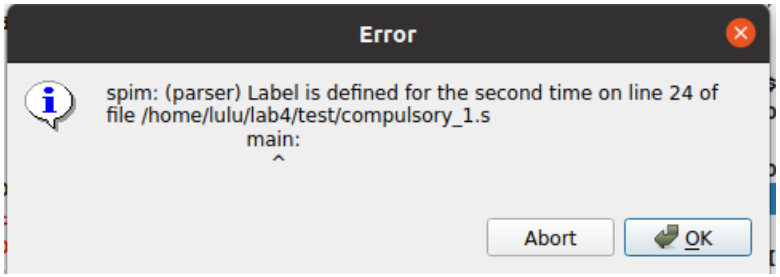
```
Enter an integer:7
1
1
2
3
5
8
13
```

# QtSPIM部署及使用

## GUI版spim使用

- QtSPIM Simulator执行完一个汇编代码文件后，需要点击Reinitialize Simulator按钮后，才能加载新的汇编代码文件，否则会报错



# QtSPIM部署及使用

- GUI版spim使用
  - 实验内容样例二执行结果

Data | Text

Text

```
                    User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)            ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4         ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4          ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2            ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2          ; 187: addu $a2 $a2 $v0
[00400014] 0c100058  jal 0x00400160 [main]    ; 188: jal main
[00400018] 00000000  nop                      ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10           ; 191: li $v0 10
[00400020] 0000000c  syscall                  ; 192: syscall # syscall 10 (exit)
[00400024] 34020004  ori $2, $0, 4            ; 8: li $v0, 4
[00400028] 3c041001  lui $4, 4097 [_prompt]   ; 9: la $a0, _prompt
[0040002c] 0000000c  syscall                  ; 10: syscall
[00400030] 34020005  ori $2, $0, 5            ; 11: li $v0, 5
[00400034] 0000000c  syscall                  ; 12: syscall
[00400038] 03e00008  jr $31                   ; 13: jr $ra
[0040003c] 34020001  ori $2, $0, 1            ; 16: li $v0, 1
[00400040] 0000000c  syscall                  ; 17: syscall
[00400044] 34020004  ori $2, $0, 4            ; 18: li $v0, 4
[00400048] 3c011001  lui $1, 4097 [_ret]      ; 19: la $a0, _ret
[0040004c] 34240012  ori $4, $1, 18 [_ret]
[00400050] 0000000c  syscall                  ; 20: syscall
[00400054] 00001021  addu $2, $0, $0          ; 21: move $v0, $0
[00400058] 03e00008  jr $31                   ; 22: jr $ra
[0040005c] 27bdfffc  addiu $29, $29, -4       ; 25: subu $sp, $sp, 4
```

Console

```
Enter an integer:7
5040
```


# 谢谢大家