

编译实习课程

MINIC

V1.1

2017 年 12 月

目录

目录.....	2
Lex&YACC.....	4
MiniC 相关说明.....	5
1 概述.....	5
2 语法描述	5
3 BNF.....	7
4 示例.....	8
5 MiniC 语法扩展.....	9
Eeyore 文档说明.....	10
1 概述.....	10
2 语法描述	10
3 BNF.....	13
4 示例.....	14
5 Eeyore 模拟器使用方式.....	15
寄存器分配.....	16
Tigger 文档说明.....	17
1 概述.....	17
2 语法描述	17
3 BNF.....	20
4 示例.....	21
5 Tigger 模拟器使用方式.....	22
RISCV 汇编手册.....	24
1.寄存器	24
2.伪指令	24
3.Bss.....	26
RISCV-工具链-模拟器	28
RISC-V GNU Compiler Toolchain	28

riscv-qemu	29
------------------	----

Lex&YACC

MiniC 相关说明

1 概述

MiniC是C语言的一个子集，对C语言语法进行了大量删减，以产生一种适用于编译实习课程的语言。

MiniC是为了取代原先编译实习课程使用的MiniJava语言而设计的，目的是更好地配合编译原理课程进度，在一定程度上减轻任务量。MiniC基础语法高度精炼，使同学们无论能力如何，都能完成编译器编写的过程。

同时，对于能力较强的同学，实习课程可以提供一些MiniC语法扩展内容，把从C语言中删除的一些语法加回MiniC，来提升MiniC语言的易用程度和表达能力，对完成扩展内容的同学提供一定程度的加分。

2 语法描述

- MiniC 取消了C 语言中的宏。
- MiniC 中的变量有两种类型，int 和一维int 数组。
- MiniC 中函数返回值只有int，参数可以是int 或int 数组，程序从main 函数开始执行。同时，MiniC 不会给函数默认返回值，如果执行完一个函数而没有return，会导致未知行为。
- 单目运算符有'!' 和 '-'
- 双目运算符有'+','-','*','/','%','&&','||','<','>','==','!='
- 合法的表达式参考BNF。
- 允许使用函数前置声明（参见样例中getint 函数）。
- MiniC 程序中允许C 风格的单行注释。
- MiniC 只保留if-else 条件分支语句和while 循环语句。
- 为了使MiniC 更容易实现，限定if,while 后面的括号里和逻辑运算符两边的运算分量只会出现如下两种形式：

$x > y || (a + b) != c$ 这样的逻辑表达式；

x 或 $f(x)$ 或 $a[x]$ 这样的单个变量或函数。

总之，不会出现类似if (a+b)或(b+c)||d这样的语句。

3 BNF

$\langle \textit{Goal} \rangle ::= (\textit{VarDefn} \mid \textit{FuncDefn} \mid \textit{FuncDecl})^* \textit{MainFunc}$

$\langle \textit{VarDefn} \rangle ::= \textit{Type Identifier} \text{' ;'}$
 $\mid \textit{Type Identifier} \text{' [} \langle \textit{INTEGER} \rangle \text{']' ;'}$

$\langle \textit{VarDecl} \rangle ::= \textit{Type Identifier}$
 $\mid \textit{Type Identifier} \text{' [} \langle \textit{INTEGER} \rangle \text{']'}$

$\langle \textit{FuncDefn} \rangle ::= \textit{Type Identifier} \text{' (} \langle \textit{VarDecl} \text{' , ' VarDecl } \rangle^* \text{')? ' } \text{' { } (\textit{FuncDecl} \mid \textit{Statement})^* \text{' } \text{' }$

$\langle \textit{FuncDecl} \rangle ::= \textit{Type Identifier} \text{' (} \langle \textit{VarDecl} \text{' , ' VarDecl } \rangle^* \text{')? ' ;'}$

$\langle \textit{MainFunc} \rangle ::= \text{' int ' main ' (' ') ' } \text{' { } (\textit{FuncDecl} \mid \textit{Statement})^* \text{' } \text{' }$

$\langle \textit{Type} \rangle ::= \text{' int'}$

$\langle \textit{Statement} \rangle ::= \text{' { } (\textit{Statement})^* \text{' }$
 $\mid \text{' if ' (' Expression ') ' Statement (' else ' Statement)?}$
 $\mid \text{' while ' (' Expression ') ' Statement}$
 $\mid \textit{Identifier} \text{' = ' Expression ;'}$
 $\mid \textit{Identifier} \text{' [' Expression '] ' = ' Expression ;'}$
 $\mid \textit{VarDefn}$
 $\mid \text{' return ' Expression ;'}$

$\langle \textit{Expression} \rangle ::= \textit{Expression} \text{' ('+' | '-' | '*' | '/' | '%') Expression}$
 $\mid \textit{Expression} \text{' ('&&' | '||' | '<' | '==' | '>' | '!=') Expression}$
 $\mid \textit{Expression} \text{' [' Expression ']'}$
 $\mid \langle \textit{INTEGER} \rangle$
 $\mid \textit{Identifier}$
 $\mid \text{' (' ! ' | '-') Expression}$
 $\mid \textit{Identifier} \text{' ((Identifier \text{' , ' Identifier })^* \text{')? ' }$
 $\mid \text{' (' Expression ')'}$

$\langle \textit{Identifier} \rangle ::= \langle \textit{IDENTIFIER} \rangle$

4 示例

```
1  int getint(); // 前置函数声明, getint 函数是 MiniC 内置函数, 返回一个读入的整数
2  int putchar(int c); // 内置函数, 用于输出字符 (参数为 ascii 码), 返回值无意义
3  // 注意! base 语法集不包括形如 int putchar(int); 这种参数名没有具体给出的函数声明
4  int putint(int i); // 内置函数, 用于输出整数, 返回值无意义
5  int getchar(); // 内置函数, 返回一个读入的字符的 ascii 码 (此程序未使用到该函数)
6  int f(int x) { /* 该函数以递归方式计算 Fibonacci 数 */
7      if (x < 2) /* if-else 语句 */
8          return 1;
9      else
10         return f(x - 1) + f(x - 2); /* 递归函数调用 */
11 }
12 int g(int x) { /* 该函数以数组和循环语句计算 Fibonacci 数 */
13     int a[40]; /* int 数组声明
14     注意! base 中数组大小必须是常数, 不可写成 int a[x]; 或 int a[10+30]; 这样 */
15     a[0] = 1;
16     a[1] = 1;
17     int i;
18     i = 2; /* 注意! base 语法集不包括初始化赋值语句 int i = 2; */
19     while (i < x + 1) { /* while 循环是 base 语法集唯一的循环语句 */
20         a[i] = a[i - 1] + a[i - 2];
21         i = i + 1;
22     }
23     return a[x];
24 }
25 int n; // 声明了一个全局变量
26 int main() {
27     n = getint();
28     if (n < 0 || n > 30) /* 不带 else 的 if 语句 */
29         return 1;
30     putint(f(n));
31     putchar(10); // 输出换行符
32     putint(g(n));
33     putchar(10);
34     return 0;
35 }
```


5 MiniC 语法扩展

MiniC设计者们亲身实践了MiniC大部分语法扩展，深切体会到实现一些复杂语法扩展的不易。

对MiniC语法进行扩展时，应尽量遵循“细致”的原则，避免涉及范围过大的扩展。

比如，整数数据类型扩展：加入8 位，16 位，32 位，64 位的有符号和无符号整数。这个扩展涉及的范围就有些大，可以考虑分解成多个扩展：带符号整数扩展、不同长度的整数运算扩展、char 字符串扩展（8 位有符号数组组成的数组）、字符与字符串表示扩展(加入"abc","\n" 这样的表达式)。

具体可以扩展内容，按教学通知为准。

Eeyore 文档说明

1 概述

Eeyore /ˈiːjʊə(r)/是一种三地址码，用为MiniC语法分析后的输出格式。Eeyore的设计同样遵循简洁的原则，使代码易读易调试。

2 语法描述

Eeyore 要求**每条语句单独占一行**。

2.1 变量

- Eeyore中的变量有三种：原生变量，临时变量，函数参数。这三种变量分别以'T','t','p'开头，后面跟随一个整数编号，编号从0开始，每个函数内部单独编号。如p0,p1,t0,T0。
- Eeyore的变量声明形如 `var t0` 和 `var 8 T0`，前者声明了一个int 型临时变量，后者声明了有2个元素的原生int 数组。
- **注意！函数参数不需要声明。**
- 除函数参数变量外，其余变量不允许重名。
- 函数内声明的变量作用域为变量声明语句到函数结束语句，函数外声明的变量作用域为变量声明语句到程序最后。
- 所谓原生变量，是指MiniC中使用的变量转到Eeyore中对应的变量。相应地，临时变量是指MiniC中没有显式对应变量的变量。

其实这两种变量在语义上**没必要做如此区分**，Eeyore 区分二者是为了方便用户调试。用个例子来说明，把左边的MiniC 语句翻译到Eeyore：

MiniC	Eeyore
<code>int a;</code>	<code>var T0</code>
<code>int b;</code>	<code>var T1</code>
<code>int c;</code>	<code>var T2</code>
<code>a = b + 2 * c;</code>	<code>var t0</code>

	<pre> t0 = 2 * T2 var t1 t1 = T1 + t0 T0 = t1 </pre>
--	---

上面T0,T1,T2 是原生变量，分别对应MiniC中的a,b,c。t0,t1是临时变量，分别对应中间运算结果2*c, b+2*c。

2.2 表达式

Eeyore 表达式有以下特点:

- 允许直接把整数用作运算符（如t0 = 2 * T2）。
- Eeyore 表达式支持的单目运算符有'!', '-'
- 支持的双目运算符有'!=', '==', '>', '<', '&&', '||', '+', '-', '*', '/', '%', 前6 个是逻辑运算符。
- 数组操作语句形如T0 [t0] = t1 和t0 = T0 [t1]。
- 注意！因为MiniC的base语法集只有int和int数组类型，数组操作语句中括号内的数应当是4的倍数。

2.3 函数

- Eeyore 中的函数以'f_' 开头，后接函数名，如f_main,f_getint。
- 函数定义语句形如f_putint [1]，中括号内的整数表示该函数的参数个数，函数结束处应有函数结束语句，形如end f_xxx。
- 函数外的变量声明语句被视为全局变量声明，函数内的视为局部变量声明。
- 函数调用语句形如：t0 = call f_xxx。
- 传参数指令形如：param t1，所有传参都是**传值**，多个参数需依次传入。
- 作为参数的变量，在param Variable 语句之后，到函数调用前，不可修改。（这样限制的目的是使寄存器分配时避免繁琐的分类讨论）
- 函数返回语句形如return t0。

2.4 标号与跳转

- Eeyore中的标号以小写字母'l'开头，后接整数编号，编号从0开始，如l0,l1。标号用来指明跳转语句的跳转地点，标号声明语句形如l0:。
- 跳转语句分两种：无条件跳转、条件跳转。如goto l1和if t0 < 1 goto l0。

2.5 缩进

Eeyore没有缩进要求，但是允许缩进，为了之后代码调试的便利，我们建议正确使用缩进。

2.6 注释

Eeyore 允许单行注释，与 C 语言注释类似使用//，处理时自动忽略改行从//之后所有内容。

2.7 系统库支持

Eeyore模拟器提供对输入输出的系统调用支持，对应的函数原型如下：

- int getint() //从标准输入读取一个整数
- int putint(int x) //输出x 到标准输出
- int getchar() //从标准输入中读取一个字符
- int putchar(int x) //输出ASCII 为x 的字符

具体对应的 MiniC 代码和 Eeyore 代码用法参见章节“示例”

3 BNF

$\langle Declaration \rangle ::= \text{'var' } \langle INTEGER \rangle? \text{ Variable}$

$\langle FunctionDecl \rangle ::= \text{Function '[' } \langle INTEGER \rangle \text{ ']' '\n' ((Expression | Declaration) '\n')^* \text{'end'}}$

Function

$\langle RightValue \rangle ::= \text{Variable} \mid \langle INTEGER \rangle$

$\langle Expression \rangle ::= \text{Variable '=' RightValue OP2 RightValue}$

$\mid \text{Variable '=' OP1 RightValue}$

$\mid \text{Variable '=' RightValue}$

$\mid \text{Variable '[' RightValue ']' = RightValue}$

$\mid \text{Variable = Variable '[' RightValue ']'}$

$\mid \text{'if' RightValue LogicalOP RightValue 'goto' Label}$

$\mid \text{'goto' Label}$

$\mid \text{Label ':'}$

$\mid \text{'param' RightValue}$

$\mid \text{Variable '=' 'call' Function}$

$\mid \text{'return' RightValue}$

$\langle Identifier \rangle ::= \langle IDENTIFIER \rangle$

$\langle Variable \rangle ::= \langle VARIABLE \rangle$

$\langle Label \rangle ::= \langle LABEL \rangle$

$\langle Function \rangle ::= \langle FUNCTION \rangle$

4 示例

MiniC	Eeyore
<pre> int getint(); int putint(int x); int n; int a[10]; int main(){ n = getint(); if (n > 10) return 1; int s; int i; i = 0; s = i; while (i < n) { a[i] = getint(); s = s + a[i]; //++i; i = i + 1 } putint(s); return 0; } </pre>	<pre> var T0 var 40 T1 f_main [0] T0 = call f_getint var t0 t0 = T0 - 10 var t1 t1 = t0 > 0 if t1 == 0 goto 10 return 1 10: var T2 var T3 T3 = 0 T2 = T3 11: var t2 t2 = T3 < T0 if t2 == 0 goto 12 var t3 t3 = 4 * T3 var t4 t4 = call f_getint T1 [t3] = t4 var t5 t5 = T1 [t3] T2 = T2 + t5 T3 = T3 + 1 goto 11 12: param T2 call f_putint return 0 end f_main </pre>

5 Eeyore 模拟器使用方式

Usage ./ Eeyore [-d] <filename >

-d : enable debug mode

- e.g. ./ Eeyore -d test.in

出现"> " 提示符表示进入debug 模式, 支持如下指令:

+ p <pc/symbol/label/function name >

- e.g. p pc , p c1 , p t1

- Print the value of the symbol

+ s <number >

- e.g. s 10

- Run n Step

+ n

- e.g. n

- Run 1 Step

+ u <number/function/label >

- e.g. u 10, u f_g , u l1

- Run until pc equal to number or until the function or label.

+ r

- e.g. r

- Disable the debug mode and run until the program exit.

寄存器分配

线性扫描

图染色

Tigger 文档说明

1 概述

Tigger /'tɪgə(r)/ 是面向 RISC-V 的一种中间表示，用作寄存器分配的输出格式。为了让同学们快速熟悉 Tigger 语法，Tigger 遵循一贯的简洁易读风格，被设计得与 Eeyore 很像。

2 语法描述

2.1 寄存器

Tigger 共有 28 个可用的寄存器，这些寄存器的名称与 RISC-V 保持一致（相比 RISC-V，删去了一些不需要编译器管理寄存器）。

- x0：该寄存器恒等于 0，不可更改
- s0-s11：没什么特殊之处，被调用者保存。
- t0-t6：没什么特殊之处，调用者保存。
- a0-a7：用来传递函数参数，调用者保存。其中 a0 和 a1 也被用作传递函数返回值，但因为 MiniC 中所有函数返回值都是 int，所以实际上只有 a0 被用作传递返回值。可以看出，最多只能通过寄存器传递 8 个参数。简单起见，限定所有函数参数个数不超过 8 个。

2.2 表达式、标号、跳转语句

- 所有的表达式计算都在寄存器上进行。
- 所有在 Eeyore 中支持的运算符，在 Tigger 中都支持。
- 注意！因为 MiniC 里只有 int 和 int 数组类型，所以形似数组赋值语句的赋值语句中括号内的数是 4 的倍数。
- 注意！由于 RISC-V 某些规则的原因，Tigger 中只有 '+' 和 '<' 运算符允许作为 `Reg = Reg OP2 <INTEGER>` 语句中的 OP2。
- 标号与跳转语句和 Eeyore 中的语法相同，标号是全局的。

2.3 函数

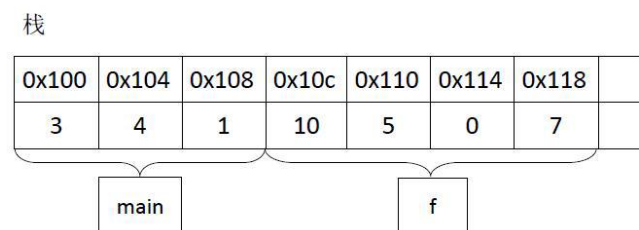
- 函数定义语句形如 `f_xxx [2] [3]`，第一个中括号内是参数个数，第二个是该函数需要用到的栈空间的大小（除以 4 之后）。
- 函数结束语句和 Eeyore 中的一样，形如 `end f_xxx`。
- 函数必须以返回语句返回。返回值通过寄存器传递。
- 函数调用语句形如 `call f_xxx`。

2.4 栈内存操作

程序运行时，每个被调用的函数都会维护一个连续的栈空间，大小为函数定义语句中的第二个参数。

局部变量都可以在栈中找到，因此 Tigger 中不再有局部变量了。

- `store Reg <INTEGER>` 语句中，`<INTEGER>` 是一个小于函数定义语句第二个系数的非负整数。该语句会把寄存器`<Reg>` 的值存入当前函数栈空间第`<INTEGER>` 个位置。
 - `load <INTEGER> Reg` 语句中，`<INTEGER>` 是一个小于函数定义语句第二个系数的非负整数。该语句会把当前函数栈空间第`<INTEGER>` 个整数存入寄存器`<Reg>`。
 - `loadaddr <INTEGER> Reg` 语句中，`<INTEGER>` 是一个小于函数定义语句第二个系数的非负整数。该语句会把当前函数栈空间第`<INTEGER>` 个位置的内存地址存到寄存器`<Reg>`。
- 举个例子，假设某个时刻函数调用关系是 `main[0][3] -> f[0][4]`，正在执行函数 `f`，假设此时的栈如下图所示：



栈计数从 0 开始，此时语句 `load 2 s0`，会使 `s0 = 0`；语句 `loadaddr 2 s0`，会使 `s0 = 0x114`；语句 `store s0 2` 会把图中的 0 改成 `s0` 的值。

2.5 全局变量

- 全局变量名称以 `v` 开头，后接一个整数编号，编号从 0 开始，比如 `v0,v1`。
- `<VARIABLE> = <INTEGER>` 用来声明一个初始值为 `<INTEGER>` 的全局变量 `<VARIABLE>`，即 `<VARIABLE>` 这个名称表示的内存地址上 4 字节的内容为 `<INTEGER>`。
- `<VARIABLE> = malloc <INTEGER>` 用来声明数组，`<VARIABLE>` 这个名称表示的内存地址之后的 `<INTEGER>` 字节的内容为一个数组。注意！`<INTEGER>` 是 4 的倍数。
- `load <VARIABLE> Reg` 表示把 `<VARIABLE>` 这个全局变量对应内存地址上 4 字节的内容加载到寄存器 `Reg`。
- `loadaddr <VARIABLE> Reg` 表示把 `<VARIABLE>` 这个全局变量对应内存地址加载到寄存器 `Reg`。
- 注意！由于 RISC-V 汇编的原因，没有 `store Reg <VARIABLE>` 语句。该语句可以通过 `loadaddr` 语句与数组访问语句结合起来完成。

2.6 注释

Tigger 允许单行注释，与 C 语言注释类似使用 `//`，处理时自动忽略改行从 `//` 之后所有内容。

2.7 系统库支持

与 MiniC 和 Eeyore 中的输入输出函数原型相同。

四种输入输出函数都通过 `a0` 寄存器传递参数和返回值。

3 BNF

$\langle \text{Goal} \rangle \quad ::= (\text{FunctionDecl} \mid \text{GlobalVarDecl})^*$

$\langle \text{GlobalVarDecl} \rangle \quad ::= \langle \text{VARIABLE} \rangle '=' \langle \text{INTEGER} \rangle$
 $\quad \mid \langle \text{VARIABLE} \rangle '=' \text{'malloc'} \langle \text{INTEGER} \rangle$

$\langle \text{FunctionDecl} \rangle \quad ::= \text{Function} '[' \langle \text{INTEGER} \rangle ']' '[' \langle \text{INTEGER} \rangle ']' (\text{Expression})^* \text{'end'}$
 $\quad \text{Function}$

$\langle \text{Expression} \rangle \quad ::= \text{Reg} '=' \text{Reg} \text{OP2} \text{Reg}$
 $\quad \mid \text{Reg} '=' \text{Reg} \text{OP2} \langle \text{INTEGER} \rangle$
 $\quad \mid \text{Reg} '=' \text{OP1} \text{Reg}$
 $\quad \mid \text{Reg} '=' \text{Reg}$
 $\quad \mid \text{Reg} '=' \langle \text{INTEGER} \rangle$
 $\quad \mid \text{Reg} '[' \langle \text{INTEGER} \rangle ']' = \text{Reg}$
 $\quad \mid \text{Reg} = \text{Reg} '[' \langle \text{INTEGER} \rangle ']'$
 $\quad \mid \text{'if'} \text{Reg} \text{LogicalOP} \text{Reg} \text{'goto'} \text{Label}$
 $\quad \mid \text{'goto'} \text{Label}$
 $\quad \mid \text{Label} ':'$
 $\quad \mid \text{'call'} \text{Function}$
 $\quad \mid \text{'store'} \text{Reg} \langle \text{INTEGER} \rangle$
 $\quad \mid \text{'load'} \langle \text{INTEGER} \rangle \text{Reg}$
 $\quad \mid \text{'load'} \langle \text{VARIABLE} \rangle \text{Reg}$
 $\quad \mid \text{'loadaddr'} \langle \text{INTEGER} \rangle \text{Reg}$
 $\quad \mid \text{'loadaddr'} \langle \text{VARIABLE} \rangle \text{Reg}$
 $\quad \mid \text{'return'}$

$\langle \text{Reg} \rangle \quad ::= \text{'x0'} \mid \text{'s0'} \mid \text{'s1'} \mid \text{'s2'} \mid \text{'s3'} \mid \text{'s4'} \mid \text{'s5'} \mid \text{'s6'} \mid \text{'s7'} \mid \text{'s8'}$
 $\quad \mid \text{'s9'} \mid \text{'s10'} \mid \text{'s11'} \mid \text{'a0'} \mid \text{'a1'} \mid \text{'a2'} \mid \text{'a3'} \mid \text{'a4'} \mid \text{'a5'}$
 $\quad \mid \text{'a6'} \mid \text{'a7'} \mid \text{'t0'} \mid \text{'t1'} \mid \text{'t2'} \mid \text{'t3'} \mid \text{'t4'} \mid \text{'t5'} \mid \text{'t6'}$

$\langle \text{Label} \rangle \quad ::= \langle \text{LABEL} \rangle$

$\langle \text{Function} \rangle \quad ::= \langle \text{FUNCTION} \rangle$

4 示例

```
f_fac [1] [3]
    a0 = a0 + -1
    if a0 <= x0 goto l1
    store a0 0
    call f_fac
    store a0 1
    store s0 2
    loadaddr 0 s0
    a0 = s0[0]
    a0 = a0 + -1
    call f_fac
    a1 = s0[4]
    load 2 s0
    a0 = a0 + a1
    goto l2
l1:
    a0 = 1
l2:
    return
end f_fac
f_main [0] [0]
    call f_getint
    call f_fac
    call f_putint
    a0 = 10
    call f_putchar
    a0 = 0
    return
end f_main
```

5 Tigger 模拟器使用方式

Usage ./ Tigger [-d] <filename >

-d : enable debug mode

e.g. ./ Tigger -d test.in

出现"> " 提示符表示进入debug 模式, 支持如下指令:

+ l

- Print current line number

+ n

- Run one step

+ pr <Reg >

- e.g. pr a0 , pr s0

- Print register value

+ prx <Reg >

- e.g. prx a0 , prx s0

- Print register value as hexadecimal

+ ps <stacknum >

- e.g. ps 0, ps 1

- Print the value of stack memory

+ psx <stacknum >

- e.g. psx 0, psx 1

- Print the value of stack memeory as hexadecimal

+ pg <variable >

- e.g. pg v0 , pg v1

- Print the value of global variable

+ b <number >

- e.g. b 10

- Set a breakpoint at a certain line

+ d <number >

- e.g. d 10

- Delete the breakpoint at a certain line

- + c

- Run until meet a breakpoint

- + q

- Quit Tigger simulator

RISCV 汇编手册

1.寄存器

Register	ABI Name	Desctiption	Saver
X0	zero	Hard-wired zero	---
X1	ra	Return address	Caller
X2	sp	Stack pointer	Callee
X3	gp	Global pointer	---
X4	tp	Thread pointer	---
X5	t0	Temporary/alternate link register	Caller
X6-7	t1-2	Temporaries	Caller
X8	s0/fp	Saved register/frame pointer	Callee
X9	s1	Saved register	Callee
X10-11	a0-1	Function arguments/return values	Caller
X12-17	a2-7	Function arguments	Caller
X18-27	s2-11	Saved registers	Callee
X28-31	t3-6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP arguments/return values	Caller
f12-17	fa2-11	FP arguments	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

2.伪指令

ops

reg = integer	li reg,integer
reg1 = reg2 op reg3	-----
reg1 = reg2 + reg3	add reg1,reg2,reg3
reg1 = reg2 - reg3	sub reg1,reg2,reg3
reg1 = reg2 * reg3	mul reg1,reg2,reg3
reg1 = reg2 / reg3	div reg1,reg2,reg3
%	rem reg1,reg2,reg3

<	slt reg1,reg2,reg3
>	sgt reg1,reg2,reg3
&&	seqz reg1,reg2 add reg1,reg1,-1 and reg1,reg1,reg3 snez reg1,reg1
	or reg1,reg2,reg3 snez reg1,reg1
!=	xor reg1,reg2,reg3 snez reg1,reg1
==	xor reg1,reg2,reg3 seqz reg1,reg1
reg1 = reg2 op integer	-----
+	add reg1,reg2,integer
<	slti reg1,reg2,integer
reg1 = reg2	mv
reg1 [int] = reg2	sw
reg1 = reg2 [int]	lw
if reg1 op reg2 goto Label	-----
<	blt reg1,reg2,label
>	bgt reg1,reg2,label
!=	bne reg1,reg2,label
==	beq reg1,reg2,label
<=	ble reg1,reg2,label
>=	ble reg2,reg1,label
goto label	j label
label :	.label:
call function	call finction
store reg int	sw reg,int*4(sp)

load int reg	lw reg,int*4(sp)
load global_var reg	lui reg,%hi(global_var) lw reg,%lo(global_var)(reg)
loadaddr int reg	add reg,sp,int*4
loadaddr global_var reg	lui reg,%hi(global_var) add reg,reg,%lo(global_var)
return	lw ra,stk-4(sp) add sp,sp,stk jr ra

3.Bss

在 BSS 段中，我们需要处理函数声明和全局变量声明，格式如下：

Function_declare:

function [int1] [int2]	.text .align 2 .global function .type function,@function function: add sp,sp,-stk sw ra stk-4(sp)
	stk = (int2 / 4 +1) * 16
end function	.size function,.-function
	stk = 0
f_main [0] [17] end f_main	.size f,.-f .text .align 2 .global main .type main,@function

	<pre> main: add sp,sp,-80 sw ra,76(sp) size main,.-main </pre>
	<pre> stk = (17 / 4 + 1) * 16 = 80 </pre>

Global_var_declare:

global_var = malloc int	.comm global_var,int*4,4
v2 = malloc 800012	.comm v2,3200048,4
v2 = malloc 800012	.comm v2,800012,8
	<pre> .global global_var .section .sdata .align 2 .type global_var, @object .size global_var, 4 global_var: .word int </pre>
v0 = 0	<pre> .global v0 .section .sdata .align 2 .type v0, @object .size v0, 4 v0: .word 0 </pre>

RISCV-工具链-模拟器

RISC-V GNU Compiler Toolchain

安装 RISC-V GNU Compiler Toolchain

官方网站 <https://github.com/riscv/riscv-gnu-toolchain>

git clone --recursive <https://github.com/riscv/riscv-gnu-toolchain>

Ubuntu 依赖软件安装命令

```
sudo apt-get install autoconf automake autotools-dev curl libmpc-dev
libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf
libtool patchutils bc zlib1g-dev
```

Fedora/CentOS/RHEL OS 依赖软件安装命令

```
sudo yum install autoconf automake libmpc-devel mpfr-devel gmp-devel gawk
bison flex texinfo patchutils gcc gcc-c++ zlib-devel
```

OS X 依赖软件安装命令

```
brew install gawk gnu-sed gmp mpfr libmpc isl zlib
```

安装 (使用 Newlib 库)

```
./configure --prefix=/opt/riscv
make
```

安装 (使用 Glibc 库)

64 位:

```
./configure --prefix=/opt/riscv
make linux
```

32 位:

```
./configure --prefix=/opt/riscv --with-arch=rv32gc --with-abi=ilp32d
make linux
```

Linux Multilib:

```
./configure --prefix=/opt/riscv --enable-multilib
make linux
```

使用

```
riscv64-unknown-linux-gnu-gcc hello.c -o hello
```

riscv-qemu

安装 riscv-qemu

官方网站 <https://github.com/riscv/riscv-qemu>

Qemu 有系统模式和用户模式两种，单独执行文件需要用户模式。

下载 QEMU：

```
git clone https://github.com/riscv/riscv-qemu
cd riscv-qemu
git submodule update --init dtc
git submodule update --init pixman
```

安装：

依赖软件

```
sudo apt-get install gcc libc6-dev pkg-config bridge-utils uml-utilities
zlib1g-dev libglib2.0-dev autoconf automake libtool libsdl1.2-dev
```

安装用户模式

```
./configure --target-list=riscv64-linux-user,riscv32-linux-user [--
prefix=INSTALL_LOCATION]
make
[make install] # if you supplied prefix above
```

使用

```
riscv64-unknown-linux-gnu-gcc hello.c -o hello
./riscv64-linux-user/qemu-riscv64 -L $RISCV/sysroot hello
```