1. **函数调用栈**

函数调用时，首先反向push形参，然后再push返回地址，即ra寄存器的值。接着将sp的值赋给fp即帧寄存器，并将其push入栈中。再push在外部仍在使用中的需要保护的寄存器。最后是被调用的函数使用当前栈。



图1 函数调用栈示意图

函数返回时，利用fp找到ra寄存器的位置，用一条jr指令完成return。

1. **符号表**

符号表设计采用C99标准，即只有全局符号表和当前的局部符号表，在分支语句中不允许定义变量。访问时首先访问局部符号表，若局部符号表中找不到，则转去全局符号表，若仍找不到，则报错。符号表采用python的字典结构，通过变量名来索引：

全局符号表：

WHOLE\_VALTABLE[idname]={'type':T\_type,'width':4,'value':initVal,'reg':None,'const':ISCONST,'array':True}。

记录了类型、宽度、当前值、当前使用的寄存器、是否为const变量、是否为数组变量这些信息。

局部符号表：

LOCAL\_VALTABLE[idname]={'type':T\_type,'width':4,'offset':str(stack\_offset),'value':initVal,'reg':None,'const':False,'array':True}

记录了类型、宽度、栈中的偏移、当前值、当前使用的寄存器、是否为const变量这些信息。

1. **全局变量和局部变量**

全局变量使用mips的.data段定义，位于数据区，可以加const限制，可以定义数组。

代码如下：

writeANDprint('.data')

#全局变量声明

for val in WHOLE\_VALTABLE.keys():

if(WHOLE\_VALTABLE[val]['array']):

writeANDprint(val+':\t.space\t'+str(WHOLE\_VALTABLE[val]['width']))

else:

Type = '.word' if WHOLE\_VALTABLE[val]['width']==4 else '.byte'

writeANDprint(val+':\t'+Type+'\t'+WHOLE\_VALTABLE[val]['value'])

局部变量使用mips的栈结构，通过sp指针来控制栈的存取，中间代码采用push,pop等来实现，详细内容参考中间代码设计以及实现代码。

1. **字符串**

需要打印的字符串使用mips的.data段定义，位于数据区，如”\n”将如下所示翻译：

.data

string0: .asciiz "\n"

之后需要使用只需调用string0这个名字即可。

1. **寄存器分配**

基本策略为持续申请下一个寄存器，且不能覆盖已经在使用的寄存器。在每条C语言语句或语句块结束后，将所有寄存器释放。代码如下所示：

#申请新的寄存器

def getRegt(n):

global point\_t

if(n==-1):

if(point\_t==0):

reg='$t0'

else:

reg = '$t'+str(point\_t-1)

return reg

for \_ in range(10):

reg = '$t'+str(point\_t)

point\_t+=1

if reg not in REG\_USED:

break

return reg

1. **数组**

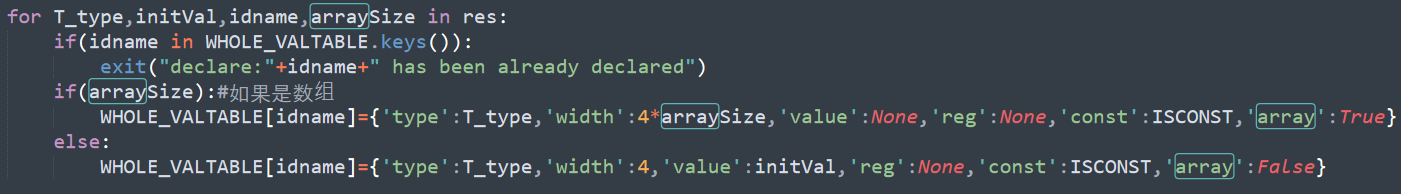
全局数组在定义时采用mips的.data段定义，如int b[20]将被翻译为：

.data

b: .space 80

同时将基地址存入全局符号表。在使用时，加载基地址读取偏移，使用arraywhole中间代码来访问数组的某一项。

定义时代码如下：



局部数组在定义时通过newstack中间代码来申请栈中的空间，同时将基地址存入局部符号表之中。在使用时，加载基地址读取偏移，使用arraylocal中间代码来访问数组的某一项。

定义时代码如下：

