

archlab Part A文档

软件52班 张迁瑞 2015013226

程序说明与注释

sum.js

第一个程序，非递归地计算链表的和。

这是我和Y86的第一次接触，它让我对Y86有了总体上的认识。

Y86与X86的编程思想基本相同，只是在某些具体语句上会有差别，按照书上的提示，可以先用编译器编译成X86的代码，之后再手动改成符合Y86的形式。

不过，为了防止考试的时候出现这样的题目自己不会做，我还是选择手动将C代码“翻译”成了Y86代码。稍微有点麻烦，不过也很有收获。

后面内容为part A中的三个程序。

```
init:      .pos 0
           irmovl Stack, %esp
           irmovl Stack, %ebp
           call   Main
           halt

ele1:      .align 4           # test data
           .long  0x00a
           .long  ele2
ele2:      .long  0x0b0
           .long  ele3
ele3:      .long  0xc00
           .long  0

Main:      pushl   %ebp
           rrmovl  %esp, %ebp
           irmovl  ele1, %edx      # edx=ele1
           pushl   %edx           # set edx as parameter of sum_list
           call    sum_list
           rrmovl  %ebp, %esp
           popl    %ebp
           ret
```

```

sum_list:  pushl    %ebp
           rrmovl   %esp, %ebp
           xorl     %eax, %eax          # init result value  对应C代码 int val=0
           mrmovl   8(%ebp), %edx       # init edx as the start of list
           andl     %edx, %edx          # test if array length is 0
           je      End                 # if array length is 0, stop calculating

Loop:      mrmovl   (%edx), %esi        # esi = current value  对应C代码中循环
           addl     %esi, %eax          # add current value to total value 对应C代码val+=ls->val
           irmovl   $4, %edi           # edi=4
           addl     %edi, %edx          # set edx as next
           mrmovl   (%edx), %esi       # get next value to esi 对应C代码 ls = ls -> next

           #andl    %esi, %esi
           rrmovl   %esi, %edx
           andl     %edx, %edx          # test if next is 0      对应C代码while(ls)
           je      End
           jmp      Loop

End:       rrmovl   %ebp, %esp
           popl     %ebp
           ret

# the stack starts here and grows to lower address
           .pos 0x100

Stack:

```

程序截图：

```

owen@ubuntu:~/archLab/sim/misc$ ./yis sum.yo
Stopped in 47 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000      0x000000c0a
%esp: 0x00000000      0x000000100
%ebp: 0x00000000      0x000000100
%edi: 0x00000000      0x000000004

Changes to memory:
0x00ec: 0x00000000      0x000000f8
0x00f0: 0x00000000      0x0000003d
0x00f4: 0x00000000      0x00000014
0x00f8: 0x00000000      0x000000100
0x00fc: 0x00000000      0x00000011

```

rsum.js

递归地计算列表的和。在sum.js的基础上，这个函数的完成很容易。将函数的循环部分改为递归即可。

唯一值得强调的是，使用递归的时候要注意寄存器状态的保存，要选择合适的时机将它们压入栈中，否则它们的值就会在递归中被破坏。

```

        .pos 0
init:    irmovl  Stack, %esp
        irmovl  Stack, %ebp
        call    Main
        halt

        .align 4          # test data
ele1:    .long   0x00a
        .long   ele2
ele2:    .long   0x0b0
        .long   ele3
ele3:    .long   0xc00
        .long   0

Main:    pushl   %ebp
        rrmovl  %esp, %ebp
        xorl    %eax, %eax
        irmovl  ele1, %edx    # edx=ele1
        pushl   %edx          # set edx as parameter of sum_list
        call    rsum_list
        popl    %edx
        rrmovl  %ebp, %esp
        popl    %ebp
        ret

rsum_list: pushl   %ebp
        rrmovl  %esp, %ebp
        mrmovl  8(%ebp), %edx    # init edx as the start of list
        andl    %edx, %edx      # test if array length is 0
        je      End            # if array length is 0, stop calculating 对应C
代码 if(!ls) return 0;

recursive:
        mrmovl  (%edx), %esi    # esi = current value 对应C代码 int val = ls->va
1
        pushl   %esi           # save esi
        irmovl  $4, %edi       # edi=4
        addl    %edi, %edx     # set edx as next
        mrmovl  (%edx), %esi    # get next value to esi
        rrmovl  %esi, %edx
        pushl   %edx
        call    rsum_list      # recursively call rsum_list 对应C代码 int rest
= rsum_list(ls->next)
        popl    %edx
        popl    %esi           # recover esi
        addl    %esi, %eax      # add current value to total value 对应C代码 re
turn val + rest

```

```

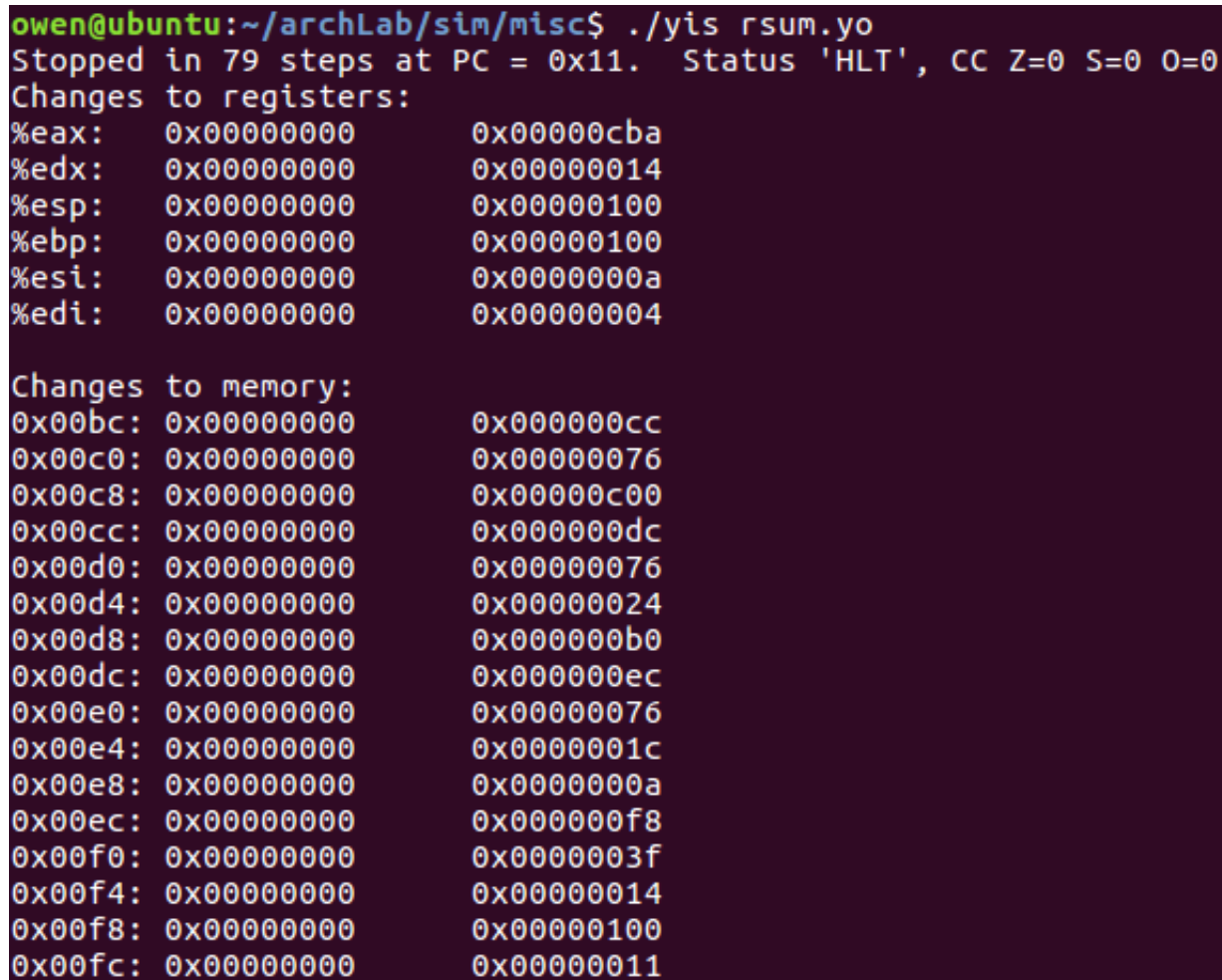
End:      rrmovl   %ebp, %esp
          popl     %ebp
          ret

# the stack starts here and grows to lower address
          .pos 0x100

Stack:

```

程序截图：



```

owen@ubuntu:~/archLab/sim/misc$ ./yis rsum.yo
Stopped in 79 steps at PC = 0x11.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax:  0x00000000      0x00000cba
%edx:  0x00000000      0x00000014
%esp:  0x00000000      0x00000100
%ebp:  0x00000000      0x00000100
%esi:  0x00000000      0x0000000a
%edi:  0x00000000      0x00000004

Changes to memory:
0x00bc: 0x00000000      0x000000cc
0x00c0: 0x00000000      0x00000076
0x00c8: 0x00000000      0x00000c00
0x00cc: 0x00000000      0x000000dc
0x00d0: 0x00000000      0x00000076
0x00d4: 0x00000000      0x00000024
0x00d8: 0x00000000      0x000000b0
0x00dc: 0x00000000      0x000000ec
0x00e0: 0x00000000      0x00000076
0x00e4: 0x00000000      0x0000001c
0x00e8: 0x00000000      0x0000000a
0x00ec: 0x00000000      0x000000f8
0x00f0: 0x00000000      0x0000003f
0x00f4: 0x00000000      0x00000014
0x00f8: 0x00000000      0x00000100
0x00fc: 0x00000000      0x00000011

```

copy.yo

进行给定长度区块的复制，并计算checksum。

有了前两个程序的积累，这个程序的完成也很轻松，本质上就是一个循环，和第一个程序很类似。

```

          .pos 0
init:     irmovl   Stack, %esp
          irmovl   Stack, %ebp
          call     Main
          halt

```

```

        .align 4          # src and dest

src:
        .long 0x00a
        .long 0x0b0
        .long 0xc00

dest:
        .long 0x111
        .long 0x222
        .long 0x333

Main:    pushl    %ebp
        rrmovl   %esp, %ebp
        irmovl   src, %edx          # push parameters into stack
        pushl    %edx
        irmovl   dest, %edx
        pushl    %edx
        irmovl   $3, %edx          # push length of area
        pushl    %edx
        call     copy_block
        rrmovl   %ebp, %esp
        popl     %ebp
        ret

copy_block: pushl    %ebp
        rrmovl   %esp, %ebp
        mrmovl   8(%ebp), %ecx      # ecx=length
        xorl     %eax, %eax         # init result, eax=result 对应C代码 int result=0
        mrmovl   12(%ebp), %edx     # edx=dest
        mrmovl   16(%ebp), %ebx     # ebx=src
        andl     %ecx, %ecx         # if length=0, return
        je      End

Loop:    mrmovl   (%ebx), %esi       # 对应C代码 int val = src
        rmmovl   %esi, (%edx)       # copy src to dest 对应C代码 *dest=val
        xorl     %esi, %eax         # result^=val 对应C代码 result^=val
        irmovl   $4, %edi          # edi = 4
        addl     %edi, %ebx         # src++ 对应C代码src++
        addl     %edi, %edx         # dest++ 对应C代码dest++
        irmovl   $1, %esi          # esi=1
        subl     %esi, %ecx         # len-- 对应C代码len--
        andl     %ecx, %ecx         # check if len is zero 对应C代码(len>0)
        jne     Loop

End:     rrmovl   %ebp, %esp
        popl     %ebp
        ret

        .pos 0x100

```

Stack:

程序截图：

```
owen@ubuntu:~/archLab/sim/misc$ ./yis copy.yo
Stopped in 57 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000      0x00000cba
%edx: 0x00000000      0x0000002c
%ebx: 0x00000000      0x00000020
%esp: 0x00000000      0x00000100
%ebp: 0x00000000      0x00000100
%esi: 0x00000000      0x00000001
%edi: 0x00000000      0x00000004

Changes to memory:
0x0020: 0x00000111      0x0000000a
0x0024: 0x00000222      0x000000b0
0x0028: 0x00000333      0x00000c00
0x00e4: 0x00000000      0x000000f8
0x00e8: 0x00000000      0x0000004d
0x00ec: 0x00000000      0x00000003
0x00f0: 0x00000000      0x00000020
0x00f4: 0x00000000      0x00000014
0x00f8: 0x00000000      0x00000100
0x00fc: 0x00000000      0x00000011
```

困难与收获

这一部分基本没有什么困难，我甚至连调试都没有用上，基本就是写完程序，用yas和yis运行之后就能得到正确的结果。仅有的一次bug经历是忘记保护寄存器的值，返回看了一下代码就解决了。

收获的话，主要还是对于archLab的整个结构有了初步的了解，特别是认识到了Y86的不足。比如常数无法直接加（减）到寄存器上，想要给寄存器改变一个常数的值，还得先把这个常数赋值到另一个寄存器上，再把两个寄存器上的数字相加，非常麻烦。很自然地想要为Y86增添寄存器加减常数的命令，于是就过渡到了Part B。Part A的作用，大概就是这样一個铺垫吧。