**ArchLab 报告**

软73 沈冠霖 2017013569

**1.PartA**

**1.1 Sum**

1. # Execution begins at address 0
2. .pos 0
3. init:
4. irmovl Stack, %esp
5. # Set up stack pointer
6. irmovl Stack, %ebp
7. # Set up base pointer
8. call Main
9. # Execute main program
10. halt
11. # Terminate program
12. # Sample linked list
13. .align 4
14. ele1:
15. .long 0x00a
16. .long ele2
17. ele2:
18. .long 0x0b0
19. .long ele3
20. ele3:
21. .long 0xc00
22. .long 0
23. Main:
24. //初始化,更新esp,ebp,存储寄存器
25. pushl %ebp
26. rrmovl %esp,%ebp
27. pushl %ebx

#设置sum的参数

1. irmovl ele1,%ebx
2. pushl %esi
3. pushl %edi
4. pushl %ebx

#调用sum

1. call Sum

#恢复寄存器，esp，ebp

1. popl %ebx
2. popl %edi
3. popl %esi
4. popl %ebx
5. rrmovl %ebp,%esp
6. popl %ebp
7. ret
8. Sum:
9. #初始化，更新esp，ebp，存储寄存器
10. pushl %ebp
11. rrmovl %esp,%ebp
12. pushl %edx
13. pushl %ecx

#读取参数，ls=ecx

1. mrmovl 8(%ebp),%ecx

#int val = 0(val=edx)

1. irmovl $0,%edx
2. andl %ecx,%ecx
3. je End
4. Loop:
5. //存储ls->val,ls->next
6. //ebx = ls.val,esi = ls.next
7. mrmovl (%ecx),%ebx
8. mrmovl 4(%ecx),%esi

//val += ls->val

1. addl %ebx,%edx

//ls = ls->next

1. rrmovl %esi,%ecx

//while跳转判断

#While(ls)

1. andl %ecx,%ecx
2. jne Loop

//退出程序

1. End:
2. //return val
3. rrmovl %edx,%eax

//恢复寄存器,esp,ebp,返回

popl %ecx

1. popl %edx
2. rrmovl %ebp,%esp
3. popl %ebp
4. ret

#设置栈地址

1. .pos 0x100
2. Stack:

**1.2 rsum**

# Execution begins at address 0

.pos 0

init:

irmovl Stack, %esp

# Set up stack pointer

irmovl Stack, %ebp

# Set up base pointer

call Main

# Execute main program

halt

# Terminate program

#设置数据

# Sample linked list

.align 4

ele1:

.long 0x00a

.long ele2

ele2:

.long 0x0b0

.long ele3

ele3:

.long 0xc00

.long 0

Main:

#初始化esp，ebp，存储寄存器

pushl %ebp

rrmovl %esp,%ebp

pushl %ebx

#设置参数ls

irmovl ele1,%ebx

pushl %esi

pushl %edi

pushl %ebx

call Rsum

#恢复寄存器，esp，ebp

popl %ebx

popl %edi

popl %esi

popl %ebx

#rrmovl %ebp,%esp

popl %ebp

ret

Rsum:

#初始化:更新esp,ebp,存储寄存器

pushl %ebp

rrmovl %esp,%ebp

pushl %edx

pushl %ecx

#读取参数,ls = ecx

mrmovl 8(%ebp),%ecx

#while的初始判断

#if(!ls)

andl %ecx,%ecx

jne Else

#return 0

irmovl $0,%eax

jmp End

Else:

#进入while循环

#存储ls->val,ls->next

#ebx = ls.val,esi = ls.next

mrmovl (%ecx),%ebx

mrmovl 4(%ecx),%esi

#val = ls->val

#edx = val

rrmovl %ebx,%edx

#rest = rsum\_list(ls->next)

#push 寄存器

pushl %ebx

pushl %edi

#放置参数

pushl %esi

call Rsum

#回收寄存器

popl %esi

popl %edi

popl %ebx

#return val+rest

#edi = val + rest

#return edi

irmovl $0,%edi

addl %eax,%edi

addl %edx,%edi

rrmovl %edi,%eax

jmp End

#退出程序

End:

#恢复寄存器,esp,ebp,返回

popl %ecx

popl %edx

#rrmovl %ebp,%esp

popl %ebp

ret

#设置stack

.pos 0x500

Stack:

**1.3 copy**

# Execution begins at address 0

.pos 0

init:

irmovl Stack, %esp

# Set up stack pointer

irmovl Stack, %ebp

# Set up base pointer

call Main

# Execute main program

halt

# Terminate program

#设置数据

.align 4

# Source block

src:

.long 0x00a

.long 0x0b0

.long 0xc00

# Destination block

dest:

.long 0x111

.long 0x222

.long 0x333

Main:

#初始化esp，ebp，备份寄存器

pushl %ebp

rrmovl %esp,%ebp

pushl %ebx

pushl %esi

pushl %edi

#初始化参数

irmovl src,%ebx

irmovl dest,%esi

irmovl $3,%edi

pushl %ebx

pushl %esi

pushl %edi

call Copy

#恢复esp，ebp，寄存器

popl %edi

popl %esi

popl %ebx

popl %edi

popl %esi

popl %ebx

popl %ebp

ret

Copy:

#初始阶段:存储ebp和寄存器

pushl %ebp

rrmovl %esp,%ebp

pushl %edx

pushl %ecx

#读取参数

#edx = src, ebx = dest, ecx = len

mrmovl 16(%ebp),%edx

mrmovl 12(%ebp),%ebx

mrmovl 8(%ebp),%ecx

#eax = result

#result = 0

irmovl $0,%eax

//循环初始判断

//if(len <= 0)

andl %ecx,%ecx

jle End

//进入while循环

Loop:

#edi = val,val = \*src

mrmovl (%edx),%edi

#src+=4（src++）,esi = 4

irmovl $0x04,%esi

addl %esi,%edx

#\*dest = val

rmmovl %edi,(%ebx)

#dest +=4（dest++）

addl %esi,%ebx

#result ^= val

xorl %edi,%eax

#len --,esi = 1

irmovl $0x01,%esi

subl %esi,%ecx

#循环退出判断

andl %ecx,%ecx

jg Loop

End:

#恢复寄存器，返回

popl %ecx

popl %edx

popl %ebp

ret

#设置栈

.pos 0x500

Stack:

1. **PartB**
   1. **顺序实现**

Iaddl指令很简单，就是把rB取出来，加上立即数，再存回去。

Leave指令则需要取出esp，ebp的值，读取ebp位置的内存，使得new\_ebp=(ebp), new\_esp = ebp+4

|  |  |  |
| --- | --- | --- |
| 阶段 | iaddl V,rB | leave |
| 取指 | icode:ifun=M1[PC]  rA:rB=M1[PC+1]  valC=M4[PC+2]  valP=PC+6 | icode:ifun=M1[PC]  没有rA，rB，valC  valP=PC+1 |
| 译码 | srcA不需要, srcB=rB  valA不需要, valB=R[rB] | srcA=%esp,srcB=%ebp  valA=R[%esp],valB=R[%ebp] |
| 执行 | valE = valC+valB  set CC | valE = valB+4 |
| 访存 | 空 | valM=M4[valB] |
| 写回 | R[rB]=valE | R[%esp]=valE,R[%ebp]=valM |
| 更新PC | PC=valP | PC=valP |

对于iaddl，在取指阶段需要设置其有效，有寄存器，有立即数；在译码写回阶段需要设置其srcB=rB，dstE=rB；在执行阶段需要设置其aluA=valC，aluB=valB，加法，设置条件码；PC阶段需要设置其PC是valP。

对于leave，在取指阶段需要设置其有效；在译码写回阶段需要设置其srcA=esp，srcB=ebp，dstE=esp，dstM=ebp；在执行阶段需要设置其aluA=valB，aluB=4；在访存阶段需要设置只读不写，mem\_addr=valB;PC阶段需要设置其PC是valP。

* 1. **流水线实现**

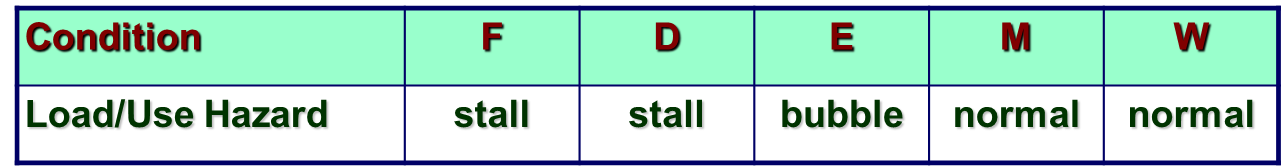
流水线实现和顺序实现的步骤划分大体相当，不过有三处不同。首先，PC阶段放到最前面了。其次，各种值有了传递关系，数据旁路等，但是这些关系已经封装好了。最后，应该把leave中的srcA，srcB修改成srcA=ebp，srcB=esp，这样才能让ebp的值留到访存阶段。

而流水线实现和顺序实现主要的区别是需要考虑数据冒险和控制冒险。

根据流水线的机制，每个会造成后面指令产生数据冒险的指令都已经靠数据旁路和bubble等规避了问题，而每个因为先前指令产生控制冒险的指令也已经靠指令预测和预测错误处理，数据旁路，bubble等机制解决了问题。因此，添加一条新指令基本上只需要考虑两点。1.新的指令会造成后面的哪些指令产生数据冒险。2.新的指令会不会因为前面的指令产生控制冒险。而鉴于iaddl，leave和我添加的新指令都不涉及指令跳转，因此没有问题2产生，为了简单，以下只考虑问题1。

首先，iaddl会修改寄存器rB，会造成后面用rB的指令有数据冒险。而因为先前的iaddl产生数据冒险的指令最早在执行阶段前用到rB的值，而iaddl得到rB的新值在执行阶段后，这只差一个阶段，可以完全靠数据旁路解决。这和opl几乎完全一样，可以类比opl，用现有的数据旁路解决，不需要额外操作。

其次，对于leave，其会修改寄存器esp，ebp，造成后面指令的数据冒险。修改得到esp的值在执行阶段后，而用到esp值的指令都最早在执行阶段前，只差一个阶段，可以用数据旁路完全解决。而修改寄存器ebp则在访存阶段后，用到寄存器ebp的指令则最早在执行阶段前，差了2个阶段。假如leave的下一条指令就在执行阶段用到ebp，则就有问题。这和mrmovl的情形类似，需要添加一个bubble，使得指令状态如下。而如果下一个指令是ret，也一样（数据冒险优先级更高）。



1. **PartC**

我实现了两个指令，第一个是rmxchg rA,D(rB)，交换寄存器rA和rB+D对应的内存值。另一个是cmp rA，rB，相当于计算rB-rA来更新条件码，但是并不修改rB，rA的值。

* 1. **阶段划分**

|  |  |  |
| --- | --- | --- |
| 阶段 | rmxchg rA,D(rB) | cmp rA,rB |
| 取指 | icode:ifun=M1[PC]  rA:rB=M1[PC+1]  valC=M4[PC+2]  valP=PC+6 | icode:ifun=M1[PC]  rA:rB=M1[PC+1]  valP=PC+2 |
| 译码 | valA=R[rA], valB=R[rB] | srcA= rA,srcB= rB  valA=R[rA],valB=R[rB] |
| 执行 | valE = valC+valB | valE = valB-valA  set CC |
| 访存 | Mem\_addr = valE  既读还写 | 不操作 |
| 写回 | R[rA]=valM | 不操作 |
| 更新PC | PC=valP | PC=valP |

* 1. **避免数据和控制冒险**

根据PartB的结论，只需要考虑这两个指令造成的后面指令的数据冒险即可。先说内存访问，因为rmxchg在访存阶段修改了内存，但是后续指令只能在访存阶段前才会用到这个内存，只差一个阶段，不会有数据冒险。而cmp指令不访问内存。

再说寄存器和条件码。Rmxchg会修改寄存器rA的值，而且是在访存阶段后才会修改，而用到rA的值则是最早在执行阶段前。这和mrmovl和之前的leave类似，需要在这个指令执行完后，让执行阶段变成bubble，前两个阶段stall。

* 1. **具体实现**

除了按照上面的方法修改pipe-full.hcl外，还需要修改其他代码。首先，要修改yas-grammar.lex，让你的新指令通过编译。其次，要修改isa.h和isa.cpp。首先，要在isa.h和isa.cpp添加你指令名的定义。其次，要在isa.cpp的对应位置添加你指令的数据描述（是否用寄存器或者立即数）和行为描述（报错的情况，执行哪些操作）来进行验证。具体修改如下面三个图。

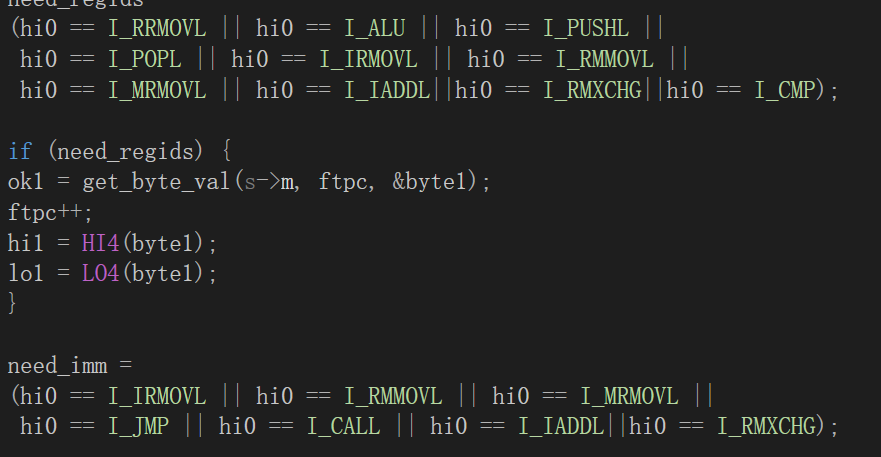


图1. 修改数据描述

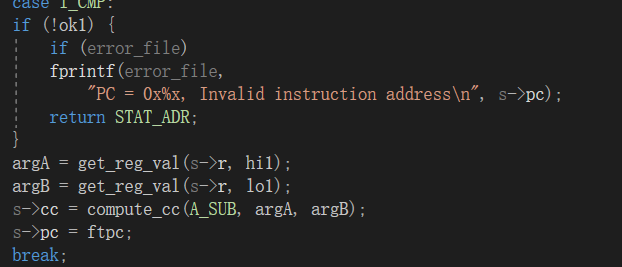


图2. 修改cmp的行为描述

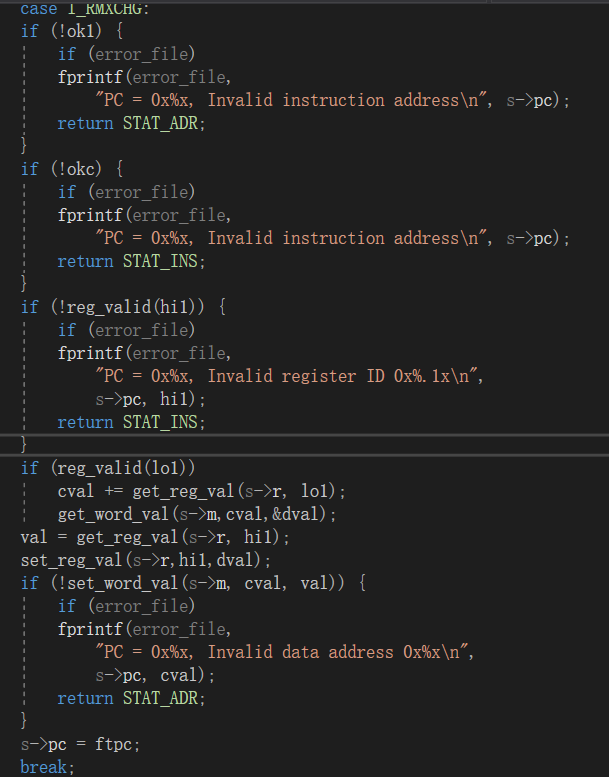


图3. 修改rmxchg的行为描述

* 1. **验证**

我修改了原先用于验证iaddl和leave的两个.ys，变为testrm.ys和testcmp.ys。

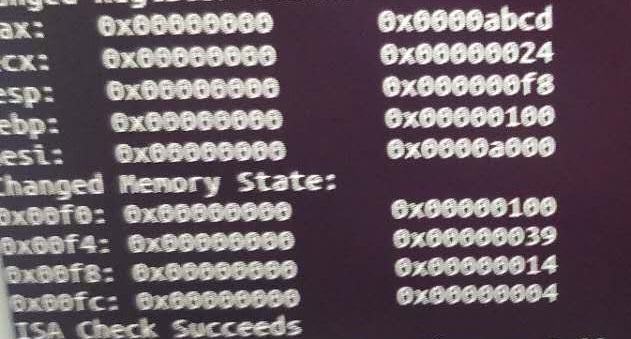
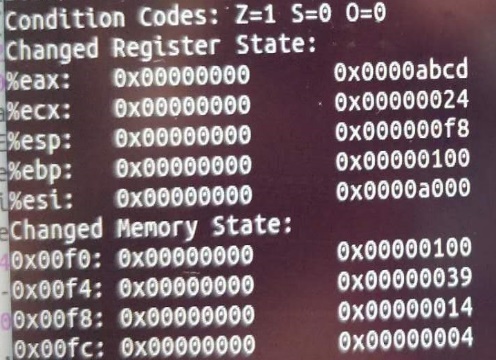
我把原先代码的部分代码等价替换为了有rmxchg或者cmp的部分。对于rmxchg，我考虑到这个会造成后面指令的数据冒险，便设置了这个代码片段：

rmxchg %esi,(%ecx) # get \*Start

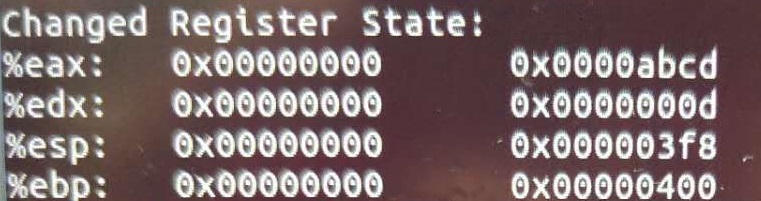
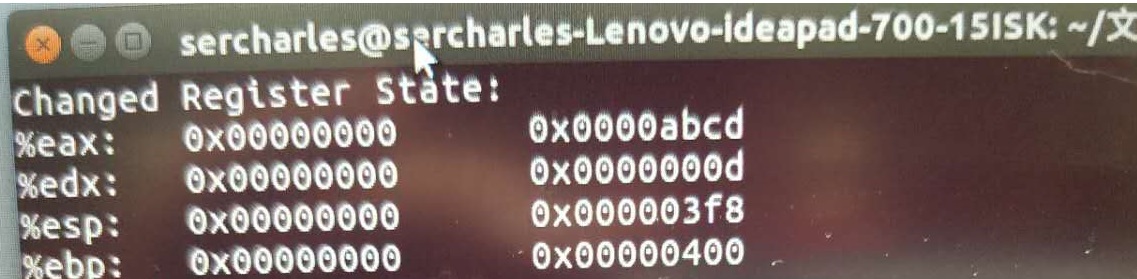
addl %esi,%eax # add to sum

而对于cmp，我把所有条件跳转前的指令都改成了cmp。

我将修改后的代码和未修改的代码的运行结果进行了比较，结果如下：



左，右图分别是修改过的代码（testrm）和未修改过的代码的运行结果，完全一致。



上，下图分别是修改过（testcmp）和没修改过的代码运行后的寄存器结果

可以看出，我增加的两条新指令仍然能够让程序得到正确的结果。可以初步说明我增加的新指令正确。

而且，经过测试，添加了我这两条指令的psim仍然能够成功执行原有的大概几百个程序，说明没有对其他指令造成影响。

1. **总结**

这次实验让我了解了Y86指令集，编程，以及顺序和流水线指令的实现，我还通过partC初步了解了Y86模拟器的实现原理和修改方法。

在指令集的实现中，我发现了一个很好的思路：先设计出指令在顺序实现中的情况，再按照流水线的数据传输模型进行微调，最后考虑流水线中，这个新指令可能受到的，或者造成其他指令的数据或控制冒险。在这一系列操作中，把新指令类比成已有的指令则是非常高效和准确的手段。毕竟，这些指令都能被拆分成六个阶段，它们都会因为类似的问题产生数据或控制冒险，其相似度远大于差别。这就是课上复用思想的一种体现。

感谢助教和从业臻同学在增加新指令方面的帮助。但是要吐槽说明文档写的不够详细：首先，没有说明白那些命令行操作在当前目录究竟是什么的情况下才能执行。其次，对于添加一条全新指令（partC）的叙述非常少，而且有问题，几乎不需要改psim.c，而大篇幅在修改yas-grammar.lex（没有从神帮助我都不知道要改这个）和isa中。希望以后的文档能够改进。