ArchLab 报告

软 73 沈冠霖 2017013569

1.PartA

1.1 Sum

```
# 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


# Sample linked list
.align 4
ele1:
.long 0x00a
.long ele2
ele2:
.long 0x0b0
.long ele3
ele3:
.long 0xc00
.long 0

Main:
//初始化,更新esp,ebp,存储寄存器
pushl %ebp
rrmovl %esp,%ebp
pushl %ebx
```

```
 #设置sum的参数
irmovl ele1,%ebx
pushl %esi
pushl %edi
pushl %ebx

 #调用sum
call Sum

 #恢复寄存器，esp，ebp
popl %ebx
popl %edi
popl %esi
popl %ebx
rrmovl %ebp,%esp
popl %ebp
ret


Sum:
#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

 #int val = 0(val=edx)
irmovl $0,%edx
andl %ecx,%ecx
je End

Loop:
//存储ls->val,ls->next
//ebx = ls.val,esi = ls.next
mrmovl (%ecx),%ebx
```

```
mrmovl 4(%ecx),%esi

 //val += ls->val
addl %ebx,%edx

 //ls = ls->next
rrmovl %esi,%ecx

 //while跳转判断
 #While(ls)
andl %ecx,%ecx
jne Loop

 //退出程序
End:
//return val
rrmovl %edx,%eax

 //恢复寄存器,esp,ebp,返回
popl %ecx
popl %edx
rrmovl %ebp,%esp
popl %ebp
ret

 #设置栈地址
.pos 0x100
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:
```

2. PartB

2.1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。

2.2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）。

Condition	F	D	E	M	W
Load/Use Hazard	stall	stall	bubble	normal	normal

3. 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。

3.1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]	icode:ifun=M1[PC] rA:rB=M1[PC+1] valP=PC+2

	valP=PC+6	
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

3.2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。

3.3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。

```
(hi0 == I_RRMOVL || hi0 == I_ALU || hi0 == I_PUSHL ||
 hi0 == I_POPL || hi0 == I_IRMOVL || hi0 == I_RMMOVL ||
 hi0 == I_MRMOVL || hi0 == I_IADDL||hi0 == I_RMXCHG||hi0 == I_CMP);

if (need_regids) {
ok1 = get_byte_val(s->m, ftpc, &byte1);
ftpc++;
hi1 = HI4(byte1);
lo1 = LO4(byte1);
}

need_imm =
(hi0 == I_IRMOVL || hi0 == I_RMMOVL || hi0 == I_MRMOVL ||
 hi0 == I_JMP || hi0 == I_CALL || hi0 == I_IADDL||hi0 == I_RMXCHG);
```

图 1. 修改数据描述

```
case I_CMP:
if (!ok1) {
    if (error_file)
    fprintf(error_file,
        "PC = 0x%x, Invalid instruction address\n", s->pc);
    return STAT_ADR;
}
argA = get_reg_val(s->r, hi1);
argB = get_reg_val(s->r, lo1);
s->cc = compute_cc(A_SUB, argA, argB);
s->pc = ftpc;
break;
```

图 2. 修改 cmp 的行为描述

```
case I_RMXCHG:
if (!ok1) {
    if (error_file)
    fprintf(error_file,
        "PC = 0x%x, Invalid instruction address\n", s->pc);
    return STAT_ADR;
}
if (!okc) {
    if (error_file)
    fprintf(error_file,
        "PC = 0x%x, Invalid instruction address\n", s->pc);
    return STAT_INS;
}
if (!reg_valid(hi1)) {
    if (error_file)
    fprintf(error_file,
        "PC = 0x%x, Invalid register ID 0x%.1x\n",
        s->pc, hi1);
    return STAT_INS;
}
if (reg_valid(lo1))
    cval += get_reg_val(s->r, lo1);
    get_word_val(s->m,cval,&dval);
val = get_reg_val(s->r, hi1);
set_reg_val(s->r,hi1,dval);
if (!set_word_val(s->m, cval, val)) {
    if (error_file)
    fprintf(error_file,
        "PC = 0x%x, Invalid data address 0x%x\n",
        s->pc, cval);
    return STAT_ADR;
}
s->pc = ftpc;
break;
```

图 3. 修改 rmxchg 的行为描述

3.4验证

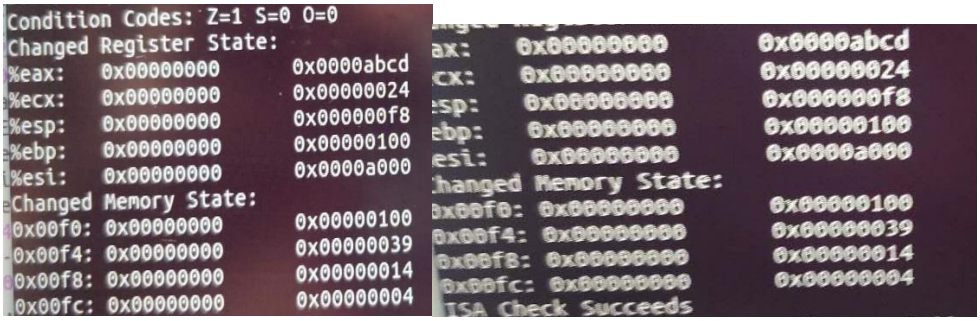
我修改了原先用于验证 iaddl 和 leave 的两个.ys，变为 testrm.ys 和 testcmp.ys。

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：

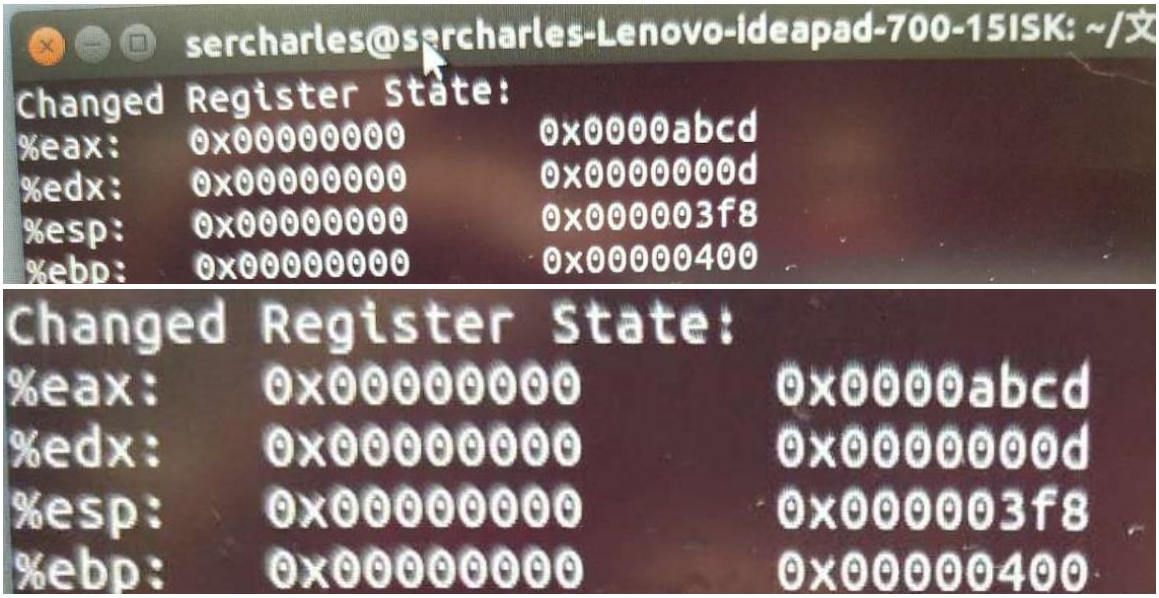
```
rmxchg %esi,(%ecx)   # get *Start
addl %esi,%eax       # add to sum
```

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。

4. 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。