Archlab Report

> 徐浩博 2020010108

Part A

- sum.ys
  - 代码及注释

```
# sum.ys
    .pos 0
init:   irmovl Stack, %esp
    irmovl Stack, %ebp
    call Main               # Main()
    halt

# Array of 4 elements
    .align 4
ele1:
    .long 0x00a
    .long ele2
ele2:
    .long 0x0b0
    .long ele3
ele3:
    .long 0xc00
    .long 0

Main:   pushl %ebp
    rrmovl %esp, %ebp
    irmovl ele1, %ebx       # %ebx = ele1
    pushl %ecx
    pushl %edx              # store caller registers
    pushl %ebx              # push ele1 as param
    call sum_list           # sum_list(ele1), %eax as ret value
    pushl %edx
    pushl %ecx              # restore caller registers
    rrmovl %ebp, %esp
    popl %ebp               # leave
    ret
sum_list: pushl %ebp
    rrmovl %esp, %ebp
    pushl %ebx
    pushl %esi
    pushl %edi              # store callee registers
    mrmovl 8(%ebp), %esi    # get param value: ls
    irmovl 0, %eax          # val = 0
loop:
    andl %esi, %esi
    je end                  # if (!ls) jmp to end
    mrmovl (%esi), %edx
    addl %edx, %eax         # val += ls -> val
```

```
    mrmovl 4(%esi), %edi
    rrmovl %edi, %esi       # ls = ls -> next
    jmp loop                # continue loop
end:
    popl %edi
    popl %esi
    popl %ebx               # restore callee registers
    rrmovl %ebp, %esp
    popl %ebp               # leave
    ret

.pos 0x400
Stack:
```

- 运行结果

  %eax 运算结果为0xcba，符合预期，运行结果正确

```
Stopped in 52 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%edx:   0x00000000      0x00000c00
%ebx:   0x00000000      0x00000014
%esp:   0x00000000      0x00000400
%ebp:   0x00000000      0x00000400
```

- rsum.ys
  - 代码及注释

```
# rsum.ys
    .pos 0
init:   irmovl Stack, %esp
    irmovl Stack, %ebp
    call Main               # Main()
    halt

# Array of 4 elements
    .align 4
ele1:
    .long 0x00a
    .long ele2
ele2:
    .long 0x0b0
    .long ele3
ele3:
    .long 0xc00
    .long 0

Main:   pushl %ebp
    rrmovl %esp, %ebp
    irmovl ele1, %ebx       # %ebx = ele1
    pushl %ecx
    pushl %edx              # store caller registers
    pushl %ebx              # push ele1 as param
    call rsum_list          # sum_list(ele1), %eax as ret value
```

```
    pushl %edx
    pushl %ecx              # restore caller registers
    rrmovl %ebp, %esp
    popl %ebp               # leave
    ret

rsum_list: pushl %ebp
    rrmovl %esp, %ebp
    pushl %ebx
    pushl %esi
    pushl %edi              # store callee registers
    mrmovl 8(%ebp), %esi    # get param value: ls
    andl %esi, %esi
    je init_condition       # if (!ls) jmp (for returning)

    mrmovl (%esi), %ebx     # %ebx = ls->val
    mrmovl 4(%esi), %edi    # %edi = ls->next
    pushl %ebx              # push ls->val for storing
    pushl %edi              # push ls->next as param
    call rsum_list          # rsum_list(ls->next)
    popl %edi               # restore ls->next
    popl %ebx               # restore ls->val
    addl %ebx, %eax         # return val: rest = rest + ls->val
    jmp end
init_condition:
    irmovl $0, %eax         # return 0
end:
    popl %edi
    popl %esi
    popl %ebx               # restore callee registers
    rrmovl %ebp, %esp
    popl %ebp               # leave
    ret

.pos 0x400
Stack:
```

- 运行结果

  %eax 运算结果为0xcba，符合预期，运行结果正确

```
Stopped in 100 steps at PC = 0x11.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%ebx:   0x00000000      0x00000014
%esp:   0x00000000      0x00000400
%ebp:   0x00000000      0x00000400
```

- copy.ys
  - 代码及注释

```
# copy.ys
    .pos 0
init:   irmovl Stack, %esp
    irmovl Stack, %ebp
```

```
    call Main                # Main()
    halt

# Array of 4 elements
    .align 4
src:
    .long 0x00a
    .long 0x0b0
    .long 0xc00
dest:
    .long 0x111
    .long 0x222
    .long 0x333

Main:   pushl %ebp
    rrmovl %esp, %ebp
    pushl %ecx
    pushl %edx               # store callee registers

    irmovl src, %ebx
    pushl %ebx               # push param: *src
    irmovl dest, %ebx
    pushl %ebx               # push param: *dest
    irmovl $3, %ebx
    pushl %ebx               # push param: len = 3
    call copy_block
    pushl %edx
    pushl %ecx               # restore callee registers
    rrmovl %ebp, %esp
    popl %ebp                # leave
    ret

copy_block: pushl %ebp
    rrmovl %esp, %ebp
    pushl %ebx
    pushl %esi
    pushl %edi               # store callee registers
    mrmovl 8(%ebp), %ecx     # %ecx = len
    mrmovl 12(%ebp), %edi    # %edi = dest
    mrmovl 16(%ebp), %esi    # %esi = src
    irmovl 0, %eax           # result = 0
loop:
    andl %ecx, %ecx
    je end                   # if (!len) jmp to return

    irmovl $4, %edx
    mrmovl (%esi), %ebx      # val = *src
    addl %edx, %esi          # src ++
    rmmovl %ebx, (%edi)      # *edi = val
    addl %edx, %edi          # dst ++
    xorl %ebx, %eax          # result ^= val
    irmovl $1, %edx
    subl %edx, %ecx          # len --
    jmp loop                 # continue loop
end:
```

```
    popl %edi
    popl %esi
    popl %ebx              # restore callee registers
    rrmovl %ebp, %esp
    popl %ebp              # leave
    ret

.pos 0x400
Stack:
```

- 运行结果

  %eax 运行结果为0xcba，且0x0020-0x0028地址对应的值成功被修改为0x00a、0x0b0、0xc00，这说明我们的运行结果是正确的。

```
Stopped in 70 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%edx:   0x00000000      0x00000001
%ebx:   0x00000000      0x00000003
%esp:   0x00000000      0x00000400
%ebp:   0x00000000      0x00000400

Changes to memory:
0x0020: 0x00000111      0x0000000a
0x0024: 0x00000222      0x000000b0
0x0028: 0x00000333      0x00000c00
```

Part B

- iaddl和leave的分阶段描述
  - iaddl
    - fetch $icode : ifun \leftarrow M_1[PC]$

      $r_A : r_B \leftarrow M_1[PC+1]$

      $valC \leftarrow M_4[PC+2]$

      $valP \leftarrow PC+6$
    - decode $valB \leftarrow R[r_B]$
    - execute $valE \leftarrow valB + valC$
    - memory
    - writeback $R[r_B] \leftarrow valE$
    - update PC $PC \leftarrow valP$
  - leave
    - fetch $icode : ifun \leftarrow M_1[PC]$

      $valP \leftarrow PC+1$
    - decode $valB \leftarrow R[\%ebp]$
    - execute $valE \leftarrow valB + 4$

- memory $valM \leftarrow M_4[valB]$
- writeback $R[\%ebp] \leftarrow valM$

  $R[\%esp] \leftarrow valE$
- update PC $PC \leftarrow valP$

- SEQ修改过程
  - fetch
    - 对于iaddl，在instr_valid、need_regids、need_valC内添加iaddl
    - 对于leave，在instr_valid

```
bool instr_valid = icode in
       { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
-             IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
+             IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL, ILEAVE

# Does fetched instruction require a regid byte?
bool need_regids =
       icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
-                   IIRMOVL, IRMMOVL, IMRMOVL };
+                   IIRMOVL, IRMMOVL, IMRMOVL, IIADDL };

# Does fetched instruction require a constant word?
bool need_valC =
-      icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
+      icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL };
```

  - decode & wirte back
    - 对于iaddl，设定srcB和dstE为rB
    - 对于leave，设定srcA，srcB和dstM为ebp，dstE为esp

```
@@ -126,27 +127,30 @@ bool need_valC =
 int srcA = [
        icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : rA;
        icode in { IPOPL, IRET } : RESP;
+       icode in { ILEAVE } : REBP;
        1 : RNONE; # Don't need register
 ];

 ## What register should be used as the B source?
 int srcB = [
-       icode in { IOPL, IRMMOVL, IMRMOVL  } : rB;
+       icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL  } : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
+       icode in { ILEAVE } : REBP;
        1 : RNONE;  # Don't need register
 ];

 ## What register should be used as the E destination?
 int dstE = [
        icode in { IRRMOVL } && Cnd : rB;
-       icode in { IIRMOVL, IOPL} : rB;
-       icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
+       icode in { IIRMOVL, IOPL, IIADDL} : rB;
+       icode in { IPUSHL, IPOPL, ICALL, IRET, ILEAVE } : RESP;
        1 : RNONE;  # Don't write any register
 ];

 ## What register should be used as the M destination?
 int dstM = [
        icode in { IMRMOVL, IPOPL } : rA;
+       icode in { ILEAVE } : REBP;
        1 : RNONE;  # Don't write any register
 ];
```

- execute
  - 对于iaddl，设置valC和valB相加，最后还要设定set_cc
  - 对于leave，设置4和valB相加

```
@@ -155,16 +159,16 @@ int dstM = [
 ## Select input A to ALU
 int aluA = [
        icode in { IRRMOVL, IOPL } : valA;
-       icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
+       icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : valC;
        icode in { ICALL, IPUSHL } : -4;
-       icode in { IRET, IPOPL } : 4;
+       icode in { IRET, IPOPL, ILEAVE } : 4;
        # Other instructions don't need ALU
 ];

 ## Select input B to ALU
 int aluB = [
        icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
-                     IPUSHL, IRET, IPOPL } : valB;
+                     IPUSHL, IRET, IPOPL, IIADDL, ILEAVE } : valB;
        icode in { IRRMOVL, IIRMOVL } : 0;
        # Other instructions don't need ALU
 ];
@@ -176,12 +180,12 @@ int alufun = [
 ];

 ## Should the condition codes be updated?
-bool set_cc = icode in { IOPL };
+bool set_cc = icode in { IOPL, IIADDL };
```

  - memory
    - 对于iaddl，无需进行memory操作
    - 对于leave，设置mem_read并设置mem_addr为valB

```
 ## Set read control signal
-bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
+bool mem_read = icode in { IMRMOVL, IPOPL, IRET, ILEAVE };

 ## Set write control signal
 bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };
@@ -190,6 +194,7 @@ bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };
 int mem_addr = [
        icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
        icode in { IPOPL, IRET } : valA;
+       icode in { ILEAVE } : valB;
        # Other instructions don't need address
 ];
```

- PIPE修改过程
  - PIPE在fetch、decode、execute、memory、write back、update PC上与SEQ几乎相同，唯一一点是给leave的d_srcA设为esp
  - 在处理冒险时，我们从数据冒险和控制冒险两方面考量两个指令
    - 对于iaddl指令，存在数据冒险，即iaddl对寄存器的修改可能影响下一步的指令，然而PIPE内的数据旁路可以解决这个问题（iaddl可以类比于rrmovl指令）；iaddl不存在控制冒险。结合这两点，我们并不需要额外新增冒险处理。
    - 对于leave指令，我们可以将其拆为rrmovl和popl两个指令。rrmovl会产生数据冒险，但数据旁路已经解决了这个问题；popl也会产生数据冒险，且由于涉及memory，并不能通过数据旁路解决，因此应当仿效popl进行如下流水线控制动作：

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

如果有组合处理ret，只采取load/use hazard的措施（如上），因此对于冒险部分的修改如下：

```
@@ -324,7 +328,7 @@ int Stat = [
 bool F_bubble = 0;
 bool F_stall =
        # Conditions for a load/use hazard
-       E_icode in { IMRMOVL, IPOPL } &&
+       E_icode in { IMRMOVL, IPOPL, ILEAVE } &&
         E_dstM in { d_srcA, d_srcB } ||
        # Stalling at fetch while ret passes through pipeline
        IRET in { D_icode, E_icode, M_icode };
@@ -333,7 +337,7 @@ bool F_stall =
 # At most one of these can be true.
 bool D_stall =
        # Conditions for a load/use hazard
-       E_icode in { IMRMOVL, IPOPL } &&
+       E_icode in { IMRMOVL, IPOPL, ILEAVE } &&
         E_dstM in { d_srcA, d_srcB };

 bool D_bubble =
@@ -341,7 +345,7 @@ bool D_bubble =
        (E_icode == IJXX && !e_Cnd) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
-       !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }) &&
+       !(E_icode in { IMRMOVL, IPOPL, ILEAVE } && E_dstM in { d_srcA, d_srcB }) &&
          IRET in { D_icode, E_icode, M_icode };

 # Should I stall or inject a bubble into Pipeline Register E?
@@ -351,7 +355,7 @@ bool E_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Cnd) ||
        # Conditions for a load/use hazard
-       E_icode in { IMRMOVL, IPOPL } &&
+       E_icode in { IMRMOVL, IPOPL, ILEAVE } &&
         E_dstM in { d_srcA, d_srcB};
```

其余部分修改与SEQ基本相同，在此就不赘述了。

Part C

我们实现了rmxchg指令

- rmxchg的分阶段描述
  - fetch $icode : ifun \leftarrow M_1[PC]$

    $r_A : r_B \leftarrow M_1[PC+1]$

    $valC \leftarrow M_4[PC+2]$

    $valP \leftarrow PC+6$
    - decode $valA \leftarrow R[r_A]$

      $valB \leftarrow R[r_B]$
  - execute $valE \leftarrow valB + valC$
  - memory $valM \leftarrow M_4[valE]$

    $M_4[valE] \leftarrow R[r_A]$
  - writeback $R[r_A] \leftarrow valM$
  - update PC $PC \leftarrow valP$
- 对于rmxchg产生的冒险
  - 这部分实际上与PartB的leave，以及mrmovl相似

Condition	F	D	E	M	W

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

- 对于PIPE的修改过程
  - fetch：添加声明`intsig IRMXCHG 'I_RMXCHG'`，并在instr_valid、need_regids、need_valC内添加iaddl
  - decode & write back：设置d_srcA、d_srcB、d_dstM分别为D_rA，D_rB，D_rA
  - excute：设置aluA、aluB分别为E_valB、E_valC
  - memory：设置mem_addr为M_valE，并且将IRMXCHG加入mem_write和mem_read
  - hazard：同PartB的leave
- 其他修改内容：
  - misc/yas-grammar.lex：向Instr添加rmxchg
  - misc/isa.h：向itype_t添加I_RMXCHG
  - misc/isa.c：
    - 向instruction_set、need_regids、need_imm添加rmxchg
    - 在模拟运行的step_state里加入如下代码：

```
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);
    int t = 0;
    get_word_val(s->m, cval, &t);
    val = get_reg_val(s->r, hi1);
    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;
    }
    set_reg_val(s->r, hi1, t);
    s->pc = ftpc;
    break;
```

- 测试
    - 测试程序
        - 为了充分测试rmxchg指令的运行以及冒险能否顺利解决，我们设计了以下y86代码

```
# 2020010108 徐浩博
    .pos 0
init:   irmovl Stack, %esp
    irmovl Stack, %ebp
    irmovl $3, %eax
    irmovl $5, %ebx
    irmovl $1, %ecx
    pushl %eax
    rmxchg %ebx, $0(%esp)
    addl %ecx, %ebx
    popl %eax
    halt

.pos 0x400
Stack:
```

主要含义是将eax压入栈，并将栈顶地址和ebx交换，紧接着给ebx加上ecx测试冒险现象，预期的结果是ebx_new=eax+ecx=4，eax_new=ebx=5

        - 测试方法：使用编译通过的psim测试
    - 运行结果
        - 运行结果如下，可以发现寄存器的值满足预期，ISA Check也成功通过，说明我们的指令测试成功

```
15 instructions executed
Status = HLT
Condition Codes: Z=0 S=0 O=0
Changed Register State:
%eax:   0x00000000      0x00000005
%ecx:   0x00000000      0x00000001
%ebx:   0x00000000      0x00000004
%esp:   0x00000000      0x00000400
%ebp:   0x00000000      0x00000400
Changed Memory State:
0x03fc: 0x00000000      0x00000005
ISA Check Succeeds
CPI: 11 cycles/10 instructions = 1.10
```