archlab实验文档

PartA

sum.ys

```
################ Y86 code ######  C code ##########################

.pos 0
init: irmovl Stack, %esp
      irmovl Stack, %ebp
      irmovl ele1, %eax
      pushl %eax
      call sum_list              sum_list(ele1);
      halt

# prepare the data
.align 4
ele1:
        .long 0x00a              ele1->val = 0x00a;
        .long ele2               ele1->next = ele2;
ele2:
        .long 0x0b0              ele2->val = 0x0b0;
        .long ele3               ele2->next = ele3;
ele3:
        .long 0xc00              ele3->val = 0xc00;
        .long 0                  ele3->next = 0;

sum_list:                        int sum_list(struct ELE* ele)
    pushl    %ebp
    rrmovl    %esp, %ebp
    irmovl  $16, %eax
    subl    %eax, %esp

    irmovl  $0, %eax
    rmmovl    %eax, -4(%ebp)     int val = 0;
    jmp    L2                        goto L2;
L3:
    mrmovl    8(%ebp), %eax
    mrmovl    (%eax), %eax
    mrmovl  -4(%ebp), %edx
    addl    %eax, %edx           val = val + ele->val;
    rmmovl  %edx, -4(%ebp)
    mrmovl    8(%ebp), %eax
    mrmovl    4(%eax), %eax
    rmmovl    %eax, 8(%ebp)      ele = ele->next;
L2:
    irmovl  $0, %edx
```

```
    mrmovl  8(%ebp), %ecx
    subl    %edx, %ecx          if(ele != NULL)
    jne    L3                       goto L3;
    mrmovl    -4(%ebp), %eax    return val;
    rrmovl %ebp, %esp
    popl %ebp
    ret

.pos 0x400
Stack:
```

rsum.ys

```
################################Y86 code ##  C code ##################
.pos 0

init: irmovl Stack, %esp
      irmovl Stack, %ebp
      irmovl ele1, %eax
      pushl %eax
      call rsum_list                        rsum_list(ele1);
      halt

# prepare the data
.align 4
ele1:
        .long 0x00a                         ele1->val = 0x00a;
        .long ele2                          ele1->next = ele2;
ele2:
        .long 0x0b0                         ele2->val = 0x0b0;
        .long ele3                          ele2->next = ele3;
ele3:
        .long 0xc00                         ele3->val = 0xc00;
        .long 0                             ele3->next = 0;

rsum_list:                                  int rsum_list(struct ELE* ele);
    pushl    %ebp
    rrmovl    %esp, %ebp
    irmovl  $16, %eax
    subl    %eax, %esp

    irmovl  $0, %eax                        int val = 0;
    rmmovl    %eax, -4(%ebp)

    # null pointer test
    irmovl  $0, %edx
    mrmovl  8(%ebp), %ecx
    subl    %edx, %ecx                      if(ele == NULL)
    je    shouldReturn                          goto shouldReturn;

    # add one element
    mrmovl    8(%ebp), %eax
```

```
    mrmovl    (%eax), %eax
    mrmovl  -4(%ebp), %edx
    addl    %eax, %edx
    rmmovl  %edx, -4(%ebp)                  val = ele->val;

    # recursive call
    mrmovl    8(%ebp), %eax
    mrmovl    4(%eax), %eax
    pushl   %eax
    call rsum_list

    mrmovl  -4(%ebp), %edx
    addl    %eax, %edx
    rmmovl  %edx, -4(%ebp)                  val += rsum_list(ele->next);

shouldReturn:
    # return
    mrmovl    -4(%ebp), %eax
    rrmovl %ebp, %esp
    popl %ebp
    ret                                     return val;


.pos 0x400
Stack:
```

copy.ys

```
######### Y86 code ######################## C code ###################
.pos 0

init: irmovl Stack, %esp
      irmovl Stack, %ebp
      irmovl src, %eax
      pushl %eax
      irmovl dest, %eax
      pushl %eax
      irmovl $3, %eax
      pushl %eax
      call copy_block                       copy_block(src, dst, 3);
      halt

# prepare the data
.align 4
# Source block
src:                                        int src[3] = {0x00a, 0x0b0, 0xc00};
        .long 0x00a
        .long 0x0b0
        .long 0xc00

# Destination block
dest:                                       int dst[3] = {0x111, 0x222, 0x333};
```

```
        .long 0x111
        .long 0x222
        .long 0x333

copy_block:                                 int copy_block(int* src, int* dst, int
len);
    pushl    %ebp
    rrmovl    %esp, %ebp

    # I will use %ebx
    pushl   %ebx

    mrmovl  8(%ebp), %ecx
    # initial checksum is 0
    irmovl  $0, %eax                        int result = 0;
    # 12(%ebp) is ``dst``
    # 16(%ebp) is ``src``


loopTest:
    irmovl  $0, %edx
    subl    %edx, %ecx                      if(len <= 0)
    jle     shouldReturn                        goto shouldReturn;


    mrmovl    16(%ebp), %edx
    mrmovl  (%edx), %edx
    mrmovl    12(%ebp), %ebx
    rmmovl  %edx, (%ebx)                    *dst = *src;
    xorl    %edx, %eax                      result ^= *src;

    # adjust src , dst and len
    irmovl  $4, %edx
    mrmovl    16(%ebp), %ebx
    addl    %edx, %ebx
    rmmovl  %ebx, 16(%ebp)                  src--;
    irmovl  $4, %edx
    mrmovl    12(%ebp), %ebx
    addl    %edx, %ebx
    rmmovl  %ebx, 12(%ebp)                  dst--;
    irmovl  $1, %edx
    subl    %edx, %ecx                      len--;

    jmp     loopTest                        goto loopTest;

shouldReturn:
    popl   %ebx
    rrmovl %ebp, %esp
    popl %ebp
    ret                                     return result;

.pos 0x400
Stack:
```

PartB

iaddl

阶段	iaddl V, rB
取指	icode:ifun <-- M1[PC]
	rA:rB <-- M1[PC + 1]
	valC <-- M4[PC + 2]
	valP <-- PC + 6
译码	valB <-- R[rB]
执行	valE <-- valB + valC
访存	
写回	R[rB] <-- valE
更新PC	PC <-- valP

leave

阶段	leave
取指	icode:ifun <-- M1[PC]
	valP <-- PC + 1
译码	valB <-- R[%ebp]
执行	valE <-- valB + 4
访存	valM <-- M4[valB]
写回	R[%esp] <-- valE
	R[%ebp] <-- valM
更新PC	PC <-- valP

SEQ处理器的模拟器修改过程

在hcl文件中的对应阶段，把相应的信号写清楚即可。

下面描述在各个阶段需要做的更改，采用 `git diff` 的格式，列出对源文件的增删情况。

在取指阶段：

```
 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 };
```

在译码阶段:

```
 ## 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
 ];
```

在执行阶段:

```
 ## 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
 ];

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

在访存阶段:

```
 ################ Memory Stage    ###################################

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

 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
 ];
```

在写回阶段：

```
 ## 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
 ];
```

PIPE处理器的模拟器修改过程

pipe处理器的修改基本类似于顺序执行处理器，只是需要在各个变量名前面加上阶段名的前缀。同样采用`git diff`的格式，列出对源文件的增删情况。

取指：

```
 # Is instruction valid?
 bool instr_valid = f_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 =
        f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
-                    IIRMOVL, IRMMOVL, IMRMOVL };
+                    IIRMOVL, IRMMOVL, IMRMOVL, IIADDL };
```

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

译码:

```
 int d_srcA = [
        D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
        D_icode in { IPOPL, IRET } : RESP;
+    D_icode in { ILEAVE } : REBP;
        1 : RNONE; # Don't need register
 ];

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

执行:

```
 ## Select input A to ALU
 int aluA = [
-       E_icode in { IRRMOVL, IOPL } : E_valA;
-       E_icode in { IIRMOVL, IRMMOVL, IMRMOVL } : E_valC;
+       E_icode in { IRRMOVL, IOPL, ILEAVE } : E_valA;
+       E_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : E_valC;
        E_icode in { ICALL, IPUSHL } : -4;
        E_icode in { IRET, IPOPL } : 4;
        # Other instructions don't need ALU
 ## Select input B to ALU
 int aluB = [
        E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
-                    IPUSHL, IRET, IPOPL } : E_valB;
+                    IPUSHL, IRET, IPOPL, IIADDL } : E_valB;
        E_icode in { IRRMOVL, IIRMOVL } : 0;
+    E_icode in { ILEAVE } : 4;
        # Other instructions don't need ALU
 ];

 ## Should the condition codes be updated?
-bool set_cc = E_icode == IOPL &&
+bool set_cc = E_icode in {IOPL, IIADDL} &&
        # State changes only during normal operation
        !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS, SHLT };
```

访存:

```
 ## Select memory address
 int mem_addr = [
        M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : M_valE;
-       M_icode in { IPOPL, IRET } : M_valA;
+       M_icode in { IPOPL, IRET, ILEAVE } : M_valA;
        # Other instructions don't need address
 ];

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

写回:

```
 ## What register should be used as the E destination?
 int d_dstE = [
-       D_icode in { IRRMOVL, IIRMOVL, IOPL} : D_rB;
-       D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
+       D_icode in { IRRMOVL, IIRMOVL, IOPL, IIADDL} : D_rB;
+       D_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 d_dstM = [
        D_icode in { IMRMOVL, IPOPL } : D_rA;
+    D_icode in { ILEAVE } : REBP;
        1 : RNONE;  # Don't write any register
 ];
```

PartC

新增的指令

- cmpl rA, rB; 计算rB - rA并更新CC，但是不改变rB
- testl rA, rB; 计算rA & rB并更新CC，但是不改变rB

动机

在实现PartA部分的代码时，我发现Y86指令集中没有直接提供比较功能，所以想要实现`cmpl rA, rB`，必须用`pushl rB; subl rA, rB; popl rB;`来实现。于是，在这部分，我想直接实现`cmpl`和`testl`指令。

修改模拟器的过程

新声明指令类（`cmpl`和`testl`都属于使用算术运算，但是不更新目标寄存器的指令）：

```
--- a/misc/isa.h
+++ b/misc/isa.h
@@ -25,7 +25,7 @@ typedef enum { R_ARG, M_ARG, I_ARG, NO_ARG } arg_t;
 /* Different instruction types */
 typedef enum { I_HALT, I_NOP, I_RRMOVL, I_IRMOVL, I_RMMOVL, I_MRMOVL,
                I_ALU, I_JMP, I_CALL, I_RET, I_PUSHL, I_POPL,
-              I_IADDL, I_LEAVE, I_POP2 } itype_t;
+              I_IADDL, I_LEAVE, I_POP2, I_TEST } itype_t;
```

在`misc/isa.c`中定义指令的格式：

```
instr_t instruction_set[] =
     {"mrmovl", HPACK(I_MRMOVL, F_NONE), 6, M_ARG, 1, 0, R_ARG, 1, 1 },
     {"addl",   HPACK(I_ALU, A_ADD), 2, R_ARG, 1, 1, R_ARG, 1, 0 },
     {"subl",   HPACK(I_ALU, A_SUB), 2, R_ARG, 1, 1, R_ARG, 1, 0 },
+    {"cmpl",   HPACK(I_TEST, A_SUB), 2, R_ARG, 1, 1, R_ARG, 1, 0 },
+    {"testl",   HPACK(I_TEST, A_AND), 2, R_ARG, 1, 1, R_ARG, 1, 0 },
```

声明对寄存器的使用：

```
     need_regids =
-       (hi0 == I_RRMOVL || hi0 == I_ALU || hi0 == I_PUSHL ||
+       (hi0 == I_RRMOVL || hi0 == I_ALU || hi0 == I_TEST || hi0 == I_PUSHL ||
         hi0 == I_POPL || hi0 == I_IRMOVL || hi0 == I_RMMOVL ||
         hi0 == I_MRMOVL || hi0 == I_IADDL);
```

执行过程与`I_ALU`（即计算类指令`opl rA, rB`）基本相同，只是不需要更新寄存器：

```
     case I_ALU:
+    case I_TEST:
        if (!ok1) {
            if (error_file)
                fprintf(error_file,
@@ -794,7 +797,10 @@ stat_t step_state(state_ptr s, FILE *error_file)
        argA = get_reg_val(s->r, hi1);
        argB = get_reg_val(s->r, lo1);
        val = compute_alu(lo0, argA, argB);
-       set_reg_val(s->r, lo1, val);
+    if(hi0 == I_ALU)
+    {
+        set_reg_val(s->r, lo1, val);
+    }
        s->cc = compute_cc(lo0, argA, argB);
        s->pc = ftpc;
        break;
```

在`lex`文件中加入新加的指令名：

```
--- a/misc/yas-grammar.lex
+++ b/misc/yas-grammar.lex
@@ -2,7 +2,7 @@
  #include "yas.h"
  unsigned int atoh(const char *);

-Instr
rrmovl|cmovle|cmovl|cmove|cmovne|cmovge|cmovg|rmmovl|mrmovl|irmovl|addl|subl|andl|xorl|
jmp|jle|jl|je|jne|jge|jg|call|ret|pushl|popl|"."byte|"."word|"."long|"."pos|"."align|ha
lt|nop|iaddl|leave
+Instr
rrmovl|cmovle|cmovl|cmove|cmovne|cmovge|cmovg|rmmovl|mrmovl|irmovl|addl|subl|andl|xorl|
jmp|jle|jl|je|jne|jge|jg|call|ret|pushl|popl|"."byte|"."word|"."long|"."pos|"."align|ha
lt|nop|iaddl|leave|cmpl|testl
```

在`pipe/pipe-full.hcl`文件中具体定义计算步骤

基本与`I_OPL`类似，只是`dstE`为空，不写入寄存器文件。

指令验证过程

```
.pos 0

irmovl  3, %eax
irmovl  2, %edx
cmpl %eax, %edx

jle lessthan
jmp greater

lessthan:
irmovl 1, %eax
testl %eax, %edx
jne end
irmovl 9, %edx
jmp end

greater:
irmovl  15, %edx
jmp end

end:
halt
```

这段指令中，如果实现正确的话，第一个`cmpl %eax, %edx`的结果为`2 - 3 = -1`，会跳转到`lessthan`分支。然后测试`1 & 2 == 0`，测试失败，继续执行`irmovl 9, %edx`，再停止。

验证结果

```
youkaichao@ubuntu:~/sim/misc$ ./yis test.yo
Stopped in 10 steps at PC = 0x3b.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000001
%edx:   0x00000000      0x00000009

Changes to memory:
youkaichao@ubuntu:~/sim/misc$
```

经过实际运行，最后`%edx`的确为9，说明实现的指令基本正确。