Archlab 实验报告

黄泽文 2022013014 未央-软件21

I. Part A

1. sum_list

完整汇编代码如下。

```
/* sum_list func */

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


      .align 4            # example variable
ele1:
      .long 0x00a
      .long ele2
ele2:
      .long 0x0b0
      .long ele3
ele3:
      .long 0xc00
      .long 0


Main:
    pushl %ebp
    rrmovl %esp,%ebp
      irmovl ele1, %edx    # %edx: head
      pushl %edx          # set argument of sum_list
    call sum_list           # sum_list();
    rrmovl %ebp,%esp
    popl %ebp
    ret


sum_list:
      pushl %ebp                    # set stack
      rrmovl %esp, %ebp
      irmovl $0, %eax               # int val = 0; // %eax: val
      mrmovl 8(%ebp), %ecx           # %ecx: ls
      jmp loop_start

      loop_start:                   # while
            mrmovl (%ecx), %edx       # %edx: ls->val
            addl %edx, %eax           # val += ls->val;

```

```
45              mrmovl 4(%ecx), %ecx        # ls = ls->next; %edx: ls->next
46              andl %ecx, %ecx             # if ls == 0
47              je loop_end
48              jmp loop_start
49
50          loop_end:                       # restore stack
51              rrmovl %ebp, %esp
52              popl %ebp
53              ret                     # return val; // %eax
54
55
56      .pos 0x100
57  Stack:
58
```

程序运行结果如下， 函数返回了 `0xcba`，即链表中所有元素的和。

```
thu@ubuntu:~/Documents/CSAPP/archlab-handout/sim/misc$ ./yis ./part_a/sum.yo
Stopped in 37 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%edx:   0x00000000      0x00000c00
%esp:   0x00000000      0x00000100
%ebp:   0x00000000      0x00000100

Changes to memory:
0x00ec: 0x00000000      0x000000f8
0x00f0: 0x00000000      0x0000003d
0x00f4: 0x00000000      0x00000014
0x00f8: 0x00000000      0x00000100
0x00fc: 0x00000000      0x00000011
```

2. rsum_list

其余代码与 `sum_list` 相同， `rsum_list` 部分代码如下。

```
1   /* rsum_list func */
2
3   rsum_list:
4         pushl %ebp
5         rrmovl %esp, %ebp
6         mrmovl 8(%ebp), %ecx    # %ecx: ls
7
8         andl %ecx, %ecx         # if (!ls)
9         je final
10
11        mrmovl (%ecx), %edx     # %edx: ls->val (*)
12        pushl %edx              # save now value
13
14        mrmovl 4(%ecx), %edx    # %edx: ls->next
15        pushl %edx              # set ls->next to be argument of next call
16        call rsum_list          # after call, %eax: rest = rsum_list(ls->next);
17        popl %edx
18        popl %edx               # %edx: val = ls->val saved in (*)
19        addl %edx, %eax          # %eax: val + rest, rtn value
20        jmp end
21
```

```
22          final:
23                irmovl $0, %eax     #  %eax: 0, the rtn value
24                jmp end
25
26          end:
27                rrmovl %ebp, %esp
28                popl %ebp
29                ret             # return val; // or return 0, %eax
```

程序运行结果如下，函数也成功返回了 `0xcba`。

```
thu@ubuntu:~/Documents/CSAPP/archlab-handout/sim/misc$ ./yis ./part_a/rsum.yo
Stopped in 73 steps at PC = 0x11.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%edx:   0x00000000      0x0000000a
%esp:   0x00000000      0x00000100
%ebp:   0x00000000      0x00000100

Changes to memory:
0x00bc: 0x00000000      0x000000cc
0x00c0: 0x00000000      0x00000068
0x00c8: 0x00000000      0x00000c00
0x00cc: 0x00000000      0x000000dc
0x00d0: 0x00000000      0x00000068
0x00d4: 0x00000000      0x00000024
0x00d8: 0x00000000      0x000000b0
0x00dc: 0x00000000      0x000000ec
0x00e0: 0x00000000      0x00000068
0x00e4: 0x00000000      0x0000001c
0x00e8: 0x00000000      0x0000000a
0x00ec: 0x00000000      0x000000f8
0x00f0: 0x00000000      0x0000003d
0x00f4: 0x00000000      0x00000014
0x00f8: 0x00000000      0x00000100
0x00fc: 0x00000000      0x00000011
```

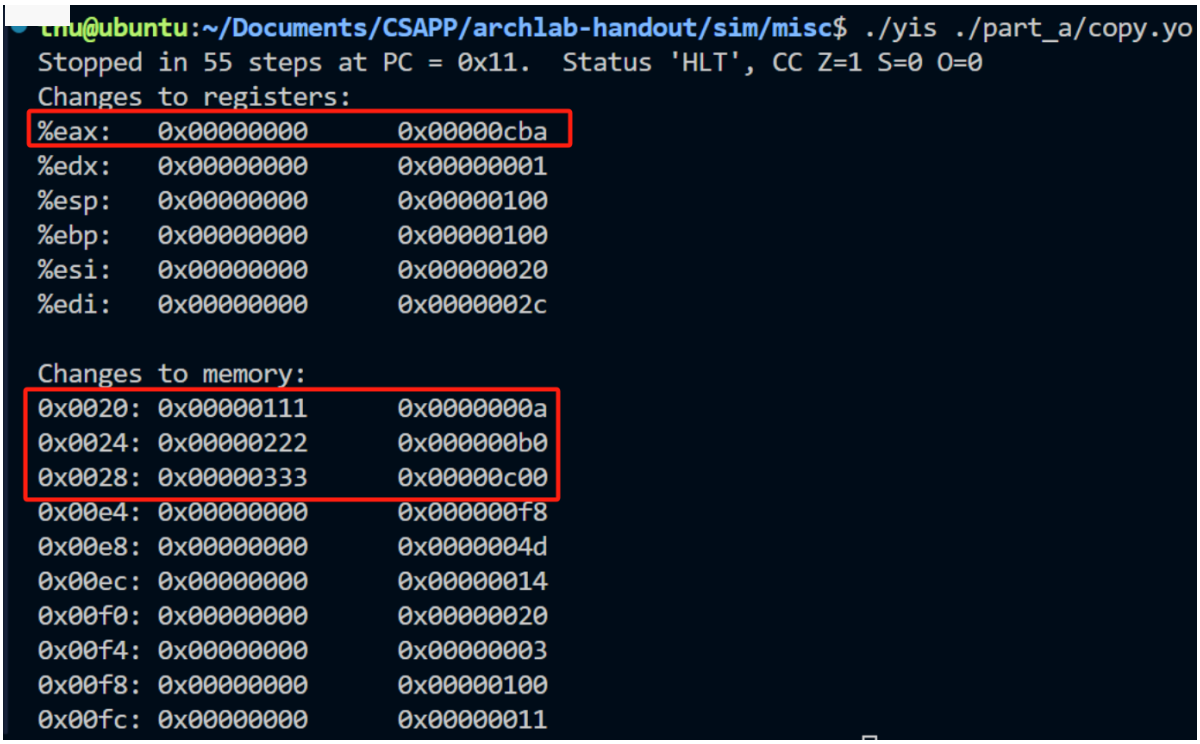
3. copy_block

完整汇编代码如下。

```
1   /* copy_block func */
2
3       .pos 0
4   init:
5       irmovl Stack, %esp
6       irmovl Stack, %ebp
7       call Main
8       halt
9
10
11          .align 4        # example variable
12  # Source block
13  src:
14          .long 0x00a
15          .long 0x0b0
16          .long 0xc00
17  # Destination block
18  dest:
```

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


Main:
    pushl %ebp
    rrmovl %esp,%ebp

        irmovl $3, %eax         # third argument: len = 3
        pushl %eax
        irmovl dest, %eax       # second argument: dest
        pushl %eax
        irmovl src, %eax        # first argument: src
        pushl %eax

    call copy_block
    rrmovl %ebp,%esp
    popl %ebp
    ret


copy_block:
       pushl %ebp
       rrmovl %esp, %ebp
       mrmovl 8(%ebp), %esi     # %esi: src
       mrmovl 12(%ebp), %edi    # %edi: dest
       mrmovl 16(%ebp), %ecx    # ecx: cnt
       irmovl $0, %eax          # int result = 0; // %eax: result
       loop:
             mrmovl (%esi), %edx # %edx: value = *src;
             rmmovl %edx, (%edi) # *dest = value;
             xorl %edx, %eax     # result = result xor val;

             irmovl $4, %edx
             addl %edx, %esi     # src++;
             addl %edx, %edi     # dest++;

             irmovl $1, %edx
             subl %edx, %ecx     # len--;
             andl %ecx, %ecx     # if (len != 0)
             jne loop
       end:
             rrmovl %ebp, %esp
             popl %ebp
             ret             # return result; // %eax


    .pos 0x100
Stack:

```

程序运行结果如下，程序返回了 `0xcba` 的结果，且 `dest` 所在的内存 `0x20 - 0x28` 中的值被 `dest` 覆盖。



II. Part B

`IADDL` 指令将一个立即数的值加到一个寄存器上，并根据结果设置条件码。

`ILEAVE` 指令将栈恢复到上一个状态，相当于 `RRMOVL %ebp, %esp` 和 `IPOPL %ebp` 两条指令合并的效果。考察实际发生的变化，`%ebp` 指向的地址更新为其当前所指向的值，`%esp` 指向的地址为更新 `%esp` 当前所指向的地址+4。为了保持设计的一致性，译码阶段将 `%ebp` 同时存进 `valA` 与 `valB` 中。

1. SEQ

`IADDL` 指令与 `ILEAVE` 指令在 `SEQ` 实现中不同阶段的计算如表所示。

Stage	IADDL V, rB	ILEAVE
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6	icode:ifun ← M_1[PC] valP ← PC + 1
Decode	valB ← R[rB]	valA ← R[%ebp] valB ← R[%ebp]
Execute	valE ← valC + valB Set CC	valE ← 4 + valB
Memory		valM ← M_4[valA]
Write Back	R[rB] ← valE	R[$esp] ← valE R[$ebp] ← valM
Update PC	PC ← valP	PC ← valP

在 HCL 代码中，直接根据表中不同阶段内容补全语句即可。

(1) Fetch

添加两个指令有效并指定 `IADDL` 需要寄存器和立即数。

```
bool instr_valid = icode in
    { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
               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, IIADDL };

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

(2) Decode

为 `IADDL` 指定寄存器 `rB` 为 `srcB` 与 `dstE`，为 `ILEAVE` 指定寄存器 `%esp` 为 `srcA`、`srcB` 与 `dstE`，`%ebp` 为 `dstM`。

```
## What register should be used as the A source?
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, 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, 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
];
```

(3) Execute

`IADDL` 计算 `valC+valB` 并设置条件码，`ILEAVE` 计算 `4+valB`。

```
## Select input A to ALU
int aluA = [
    icode in { IRRMOVL, IOPL } : valA;
    icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : valC;
    icode in { ICALL, IPUSHL } : -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, IIADDL, ILEAVE } : valB;
    icode in { IRRMOVL, IIRMOVL } : 0;
    # Other instructions don't need ALU
];

## Set the ALU function
int alufun = [
    icode == IOPL : ifun;
    icode in { IIADDL, ILEAVE } : ALUADD;
    1 : ALUADD;
];

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

(4) Memory

设置 `ILEAVE` 读取内存，地址为 `valA`。

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

## Set write control signal
bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };

## Select memory address
int mem_addr = [
    icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
    icode in { IPOPL, IRET, ILEAVE } : valA;
    # Other instructions don't need address
];
```

2. PIPE

`IADDL` 指令与 `ILEAVE` 指令在 `PIPE` 中实现与 `SEQ` 大致类似，区别如下。最大的区别是需要对 `ILEAVE` 指令处理冲突问题，见最后一部分。

(1) Fetch

添加两个指令有效并指定 `IADDL` 需要寄存器和立即数。只需将 `icode` 更为 `f_icode`。

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

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

(2) Decode

为 `IADDL` 指定寄存器 `rB` 为 `srcB` 与 `dstE`，为 `ILEAVE` 指定寄存器 `%esp` 为 `srcA`、`srcB` 与 `dstE`，`%ebp` 为 `dstM`。只需将 `srcA`、`srcB`、`dstE`、`dstM` 都加上前缀 `d_`，`icode` 加上前缀 `D_`。

```
## What register should be used as the A source?
int d_srcA = [
    D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
    D_icode in { IPOPL, IRET } : RESP;
    D_icode == 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, IIADDL  } : D_rB;
    D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
    D_icode == ILEAVE : REBP;
    1 : RNONE;   # Don't need register
];

## What register should be used as the E destination?
int d_dstE = [
    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 == ILEAVE : REBP;
    1 : RNONE;   # Don't write any register
];
```

(3) Execute

`IADDL` 计算 `valC+valB` 并设置条件码，`ILEAVE` 计算 `4+valB`。除了加上前缀外，设置条件码的条件有所变化，但只需将 `E_icode == IOPL` 改为 `E_icode in { IOPL, IIADDL }` 即可。

```
## Select input A to ALU
int aluA = [
```

```
        E_icode in { IRRMOVL, IOPL } : E_valA;
        E_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : E_valC;
        E_icode in { ICALL, IPUSHL } : -4;
        E_icode in { IRET, IPOPL, ILEAVE } : 4;
        # Other instructions don't need ALU
    ];

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

    ## Set the ALU function
    int alufun = [
        E_icode == IOPL : E_ifun;
        E_icode in { IIADDL, ILEAVE } : ALUADD;
        1 : ALUADD;
    ];

    ## Should the condition codes be updated?
    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 };
```

(4) Memory

设置`ILEAVE`读取内存，地址为`valA`。只需更改前缀。

```
    ## Select memory address
    int mem_addr = [
        M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : M_valE;
        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, ILEAVE };

    ## Set write control signal
    bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };
```

(5) Pipeline Register Control

处理冒险情况。`ILEAVE`本质上执行了一个`IPOPL`指令，会出现加载使用冒险，导致后续代码若试图调用`%ebp`的值会得到错误结果。在代码中包含`IPOPL`处类似地补充`ILEAVE`，使程序在必要时等待访存阶段结束即可。

```
    # Should I stall or inject a bubble into Pipeline Register F?
    # At most one of these can be true.
    bool F_bubble = 0;
    bool F_stall =
        # Conditions for a load/use hazard
        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 };

# Should I stall or inject a bubble into Pipeline Register D?
# At most one of these can be true.
bool D_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL, ILEAVE } &&
     E_dstM in { d_srcA, d_srcB };

bool D_bubble =
    # Mispredicted branch
    (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, 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?
# At most one of these can be true.
bool E_stall = 0;
bool E_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL, ILEAVE } &&
     E_dstM in { d_srcA, d_srcB};
```