

Archlab 报告

刘鸣霄 2021010584 自 12

Part A

这部分有一个小技巧。我们可以先把 `example.c` 编译然后反汇编,得到 `x86` 汇编程序,然后魔改成 `Y86` 汇编程序。最主要的改动是: 不像 `x86` 程序那样频繁地访问内存,用寄存器保存频繁使用的变量,这样对于 `y86` 语言也更方便。

([Part B 在第 6 页](#))

`sum.js`

```
#刘鸣霄 2021010584
    .pos 0

init:
    irmovl stack,%esp
    irmovl stack,%ebp #初始化栈指针和帧指针
    call   main       #调用主函数
    halt

# Sample linked list    #被测试数据
    .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,%edx
    pushl   %edx          #准备参数 ls
    call    sum_list      #调用 sum_list(ls)
    rrmovl  %ebp,%esp
    popl    %ebp
    ret

sum_list:
    pushl   %ebp
    rrmovl  %esp,%ebp
```

```

    mrmovl 0x8(%ebp),%ecx    #%ecx=ls
    xorl   %eax,%eax        #%eax=val=0
    jmp    test
loop:
    mrmovl (%ecx),%edx      #%edx=ls->val
    addl   %edx,%eax        #val+=ls->val
    mrmovl 0x4(%ecx),%ecx   #ls=ls->next
test:
    andl   %ecx,%ecx        #while(ls!=NULL)
    jne    loop
    popl   %ebp
    ret

    .pos 0x100
stack:                                #栈底地址

```

运行结果:

```

thu@ubuntu:~/Desktop/archlab-handout/sim/misc$ ./yas sum.y
thu@ubuntu:~/Desktop/archlab-handout/sim/misc$ ./yis sum.yo
Stopped in 36 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000    0x00000c0a
%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

```

rsum.y

```

#刘鸣霄 2021010584
    .pos 0

init:
    irmovl stack,%esp
    irmovl stack,%ebp
    call   main
    halt

# Sample linked list
    .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,%edx
    pushl   %edx                # 准备参数 ls
    call    rsum_list          # 调用 rsum_list
    rrmovl  %ebp,%esp
    popl    %ebp
    ret

rsum_list:
    pushl   %ebp
    rrmovl  %esp,%ebp
    mrmovl  0x8(%ebp),%ecx      # %ecx=ls
    xorl    %eax,%eax          # %eax=0
    andl    %ecx,%ecx
    jne     else               # if(ls!=NULL) goto else
    jmp     exit               # else goto exit
else:
    mrmovl  (%ecx),%eax         # val=ls->val
    pushl   %eax               # -4(%ebp)=val=ls_val
    mrmovl  0x4(%ecx),%eax
    pushl   %eax               # -8(%ebp)=ls->next,
                                # 准备递归调用 rsum_list 的参数
    call    rsum_list          # 递归调用 rsum_list,返回值%eax=rest=
                                # rsum_list(ls->next)
    mrmovl  0xffffffffc(%ebp),%edx  # %edx=-4(%ebp)=val
    addl    %edx,%eax          # %eax+=val,然后返回
exit:
    rrmovl  %ebp,%esp
    popl    %ebp
    ret

        .pos 0x100
stack:

```

运行结果:

```

thu@ubuntu:~/Desktop/archlab-handout/sim/misc$ ./yas rsum.y
thu@ubuntu:~/Desktop/archlab-handout/sim/misc$ ./yis rsum.yo
Stopped in 70 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      0x0000006f
0x00c8: 0x00000000      0x00000c00
0x00cc: 0x00000000      0x000000dc
0x00d0: 0x00000000      0x0000006f
0x00d4: 0x00000000      0x00000024
0x00d8: 0x00000000      0x000000b0
0x00dc: 0x00000000      0x000000ec
0x00e0: 0x00000000      0x0000006f
0x00e4: 0x00000000      0x0000001c
0x00e8: 0x00000000      0x0000000a
0x00ec: 0x00000000      0x000000f8
0x00f0: 0x00000000      0x0000003d
0x00f4: 0x00000000      0x00000014
0x00f8: 0x00000000      0x00000100
0x00fc: 0x00000000      0x00000011

```

copy.y

```

#刘鸣霄 2021010584
    .pos 0

init:
    irmovl stack,%esp
    irmovl stack,%ebp
    call    main
    halt

.align 4
# Source block
src:
    .long 0x00a
    .long 0x0b0
    .long 0xc00

# Destination block
dest:
    .long 0x111
    .long 0x222
    .long 0x333

main:
    pushl    %ebp
    rrmovl   %esp,%ebp
    irmovl   $0x3,%edx
    pushl    %edx          #准备参数 len=3
    irmovl   dest,%edx
    pushl    %edx          #准备参数 dest
    irmovl   src,%edx

```

```

    pushl    %edx          #准备参数 src
    call     copy_block    #调用 copy_block
    rrmovl   %ebp,%esp
    popl     %ebp
    ret

copy_block:
    pushl    %ebp
    rrmovl   %esp,%ebp
    pushl    %esi
    pushl    %edi
    pushl    %ebx          # 按照 handout 的指示
                          # 保存调用者寄存器
    mrmovl   0x8(%ebp),%esi # %esi=src
    mrmovl   0xc(%ebp),%edi # %edi=dest
    mrmovl   0x10(%ebp),%ecx # %ecx=len
    xorl     %ebx,%ebx     # %ebx=result=0
    andl     %ecx,%ecx     # 设置条件码
    jmp      test

loop:
    mrmovl   (%esi),%eax    # %eax=val=*src
    irmovl   $0x4,%edx     #
    addl     %edx,%esi      # src++
    rmmovl   %eax,(%edi)    # *dest=val
    addl     %edx,%edi      # dest++
    xorl     %eax,%ebx      # result^=val
    irmovl   $0x1,%edx     #
    subl     %edx,%ecx      # len--并且设置条件码
test:
    jg       loop          # while(len>0)
    rrmovl   %ebx,%eax      # %eax=result 然后返回
    popl     %ebx
    popl     %edi
    popl     %esi          # 恢复调用者寄存器
    popl     %ebp
    ret

    .pos 0x100
stack:

```

运行结果:

```

thu@ubuntu:~/Desktop/archlab-handout/sim/misc$ ./yas copy.y
thu@ubuntu:~/Desktop/archlab-handout/sim/misc$ ./yis copy.yo
Stopped in 61 steps at PC = 0x11. Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000 0x00000cba
%edx: 0x00000000 0x00000001
%esp: 0x00000000 0x00000100
%ebp: 0x00000000 0x00000100

Changes to memory:
0x0020: 0x00000111 0x0000000a
0x0024: 0x00000222 0x000000b0
0x0028: 0x00000333 0x00000c00
0x00e4: 0x00000000 0x000000f8
0x00e8: 0x00000000 0x0000004d
0x00ec: 0x00000000 0x00000014
0x00f0: 0x00000000 0x00000020
0x00f4: 0x00000000 0x00000003
0x00f8: 0x00000000 0x00000100
0x00fc: 0x00000000 0x00000011

```

Part B

SEQ

分析 `iaddl` 和 `leave` 两条指令在每个阶段的动作：其中 `iaddl` 是一个寄存器的数和一个立即数相加，结果存回那个寄存器；`leave` 指令等价于两条指令：`rrmovl %esp,%ebp` 和 `popl %ebp`

(去流水线)

```

#-----
# iaddl F:rB C
# fetch      icode:ifun <- M_1[PC]
#            rA:rB      <- M_1[PC+1]
#            valC       <- M_4[PC+2]
#            valP       <- PC+6
# decode     valB       <- R[rB]
# execute    valE       <- valC+valB
#            set CC
# memory
# writeback  R[rB]      <- valE
# PC_update  PC         <- valP
#-----
# leave
# fetch      icode:ifun <- M_1[PC]
#            valP       <- PC+1
# decode     valA       <- R[%ebp]
#            valB       <- R[%ebp]
# execute    valE       <- 4+valB
# memory     valM       <- M_4[valA]
# writeback  R[%esp]    <- valE
#            R[%ebp]    <- valM
# PC_update  PC         <- valP
#-----

```

因此对 seq-full 做下述修改，修改的简要描述如下：

取指阶段：

将 IIADDL 和 ILEAVE 加入到 instr_valid (的条件)

iaddl 需要寄存器和 value C，将 IIADDL 加入到 need_regids 和 need_valC 的条件。

leave 不需要寄存器和 value C

译码阶段：

iaddl 的 srcB=rB, dstE=rB

ileave 的 srcA=%ebp, srcB=%ebp, dstE=%esp, dstM=%ebp

依此在这些信号的条件处做相应修改

执行阶段：

iaddl: valE=valC+valB。因此 aluA=valC, aluB=valB。并且 iaddl 要 set CC。

leave: valE=4+valB。因此 aluA=4, aluB=valB。leave 不 set CC。

alu 在此都做相加运算。

依此做相应修改

访存阶段：

iaddl 不访存。

leave 要将 %ebp 指向的 old %ebp 读出来。因此将 ILEAVE 添加到 mem_read=1 和 mem_addr=valA(=%ebp) 的条件之中

PC 更新阶段：

无需修改

```
##### Fetch Stage #####

# Determine instruction code
int icode = [
    imem_error: INOP;
    1: imem_icode;    # Default: get from instruction memory
];

# Determine instruction function
int ifun = [
    imem_error: FNONE;
    1: imem_ifun;     # Default: get from instruction memory
];

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 };

##### Decode Stage #####

## 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
];

##### Execute Stage #####

## 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;
    1 : ALUADD;
];

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

##### Memory Stage #####

## 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
];

## Select memory input data
int mem_data = [
    # Value from register
    icode in { IRMMOVL, IPUSHL } : valA;
    # Return PC
    icode == ICALL : valP;

```

```

    # Default: Don't write anything
];

## Determine instruction status
int Stat = [
    imem_error || dmem_error : SADR;
    !instr_valid: SINS;
    icode == IHALT : SHLT;
    1 : SAOK;
];

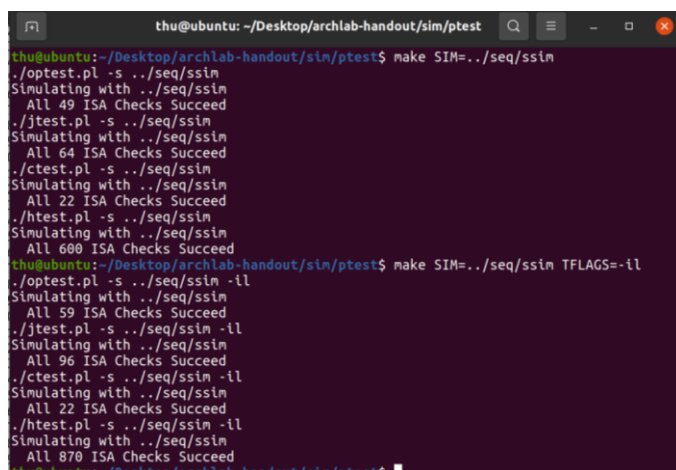
##### Program Counter Update #####

## What address should instruction be fetched at

int new_pc = [
    # Call. Use instruction constant
    icode == ICALL : valC;
    # Taken branch. Use instruction constant
    icode == IJXX && Cnd : valC;
    # Completion of RET instruction. Use value from stack
    icode == IRET : valM;
    # Default: Use incremented PC
    1 : valP;
];

```

程序能够测试成功。由于测例太多，在此只展示 `pctest` 中的回归测试。



```

thu@ubuntu: ~/Desktop/archlab-handout/sim/pctest
thu@ubuntu:~/Desktop/archlab-handout/sim/pctest$ make SIM=../seq/ssin
./optest.pl -s ../seq/ssin
Simulating with ../seq/ssin
All 49 ISA Checks Succeed
./jtest.pl -s ../seq/ssin
Simulating with ../seq/ssin
All 64 ISA Checks Succeed
./ctest.pl -s ../seq/ssin
Simulating with ../seq/ssin
All 22 ISA Checks Succeed
./htest.pl -s ../seq/ssin
Simulating with ../seq/ssin
All 600 ISA Checks Succeed
thu@ubuntu:~/Desktop/archlab-handout/sim/pctest$ make SIM=../seq/ssin TFLAGS=-il
./optest.pl -s ../seq/ssin -il
Simulating with ../seq/ssin
All 59 ISA Checks Succeed
./jtest.pl -s ../seq/ssin -il
Simulating with ../seq/ssin
All 96 ISA Checks Succeed
./ctest.pl -s ../seq/ssin -il
Simulating with ../seq/ssin
All 22 ISA Checks Succeed
./htest.pl -s ../seq/ssin -il
Simulating with ../seq/ssin
All 870 ISA Checks Succeed
thu@ubuntu:~/Desktop/archlab-handout/sim/pctest$

```

PIPE

`pipe-full.hcl` 在六个阶段的修改和 `seq` 一模一样，在此省略。

不同的是，在流水线处理器中，要分析一下这些指令产生的数据冲突和控制冲突。这两个指令不会产生数据冲突。在译码阶段，`valA`，`valB` 仍然能够正确地被取到这两个指令也不会遇到跳转分支错误或者处理 `ret` 的情况。

但是, `leave` 会从内存中读取数据(`load`), 而这个数据可能马上会被下一条指令使用(`use`), 因此 `leave` 会引起数据加载/使用冒险, 应该把它加入到加载/使用冒险的条件之中。

(去 [part C](#))

```
##### Pipeline Register Control #####

# 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 };

# Should I stall or inject a bubble into Pipeline Register M?
# At most one of these can be true.
```

```

bool M_stall = 0;
# Start injecting bubbles as soon as exception passes through memory
stage
bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR,
SINS, SHLT };

# Should I stall or inject a bubble into Pipeline Register W?
bool W_stall = W_stat in { SADR, SINS, SHLT };
bool W_bubble = 0;

```

测试能够通过。

```

thu@ubuntu: ~/Desktop/archlab-handout/sim/ptest
thu@ubuntu:~/Desktop/archlab-handout/sim/ptest$ make SIM=../pipe/psim
./optest.pl -s ../pipe/psim
Simulating with ../pipe/psim
All 49 ISA Checks Succeed
./jtest.pl -s ../pipe/psim
Simulating with ../pipe/psim
All 64 ISA Checks Succeed
./ctest.pl -s ../pipe/psim
Simulating with ../pipe/psim
All 22 ISA Checks Succeed
./htest.pl -s ../pipe/psim
Simulating with ../pipe/psim
All 600 ISA Checks Succeed
thu@ubuntu:~/Desktop/archlab-handout/sim/ptest$ make SIM=../pipe/psim TFLAGS=-il
./optest.pl -s ../pipe/psim -il
Simulating with ../pipe/psim
All 59 ISA Checks Succeed
./jtest.pl -s ../pipe/psim -il
Simulating with ../pipe/psim
All 96 ISA Checks Succeed
./ctest.pl -s ../pipe/psim -il
Simulating with ../pipe/psim
All 22 ISA Checks Succeed
./htest.pl -s ../pipe/psim -il
Simulating with ../pipe/psim
All 870 ISA Checks Succeed
thu@ubuntu:~/Desktop/archlab-handout/sim/ptest$

```

Part C

rmxchg

rmxchg 的功能是交换一个内存中的值和一个寄存器中的值

([去功能描述](#))

完成 part C，首先修改模拟器程序使之支持新的指令。

对 yas-grammer.lex 的修改：在 instr 后面添加 rmxchg

对 isa.h 的修改：在枚举类型 itype_t 中添加 I_RMXCHG

对 isa.c 的修改：在 instruction_set 数组中添加：

```
{ "rmxchg", HPACK(I_RMXCHG, F_NONE), 6, R_ARG, 1, 1, M_ARG, 1, 0 }
```

在下面的 switch 语句中添加：

```

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);
    temp_val = get_reg_val(s->r, hi1);
    if (!get_word_val(s->m, cval, &val))
        return STAT_ADR;
    set_reg_val(s->r, hi1, val);
    if (!set_word_val(s->m, cval, temp_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;

```

分析该指令：这个指令既有 `rmmovl` 的特点也有 `mrmmovl` 的特点，其格式为
`rmxchg rA,D(rB)`

```

#-----
#   rmxchg    rA:rB    D
#   fetch     icode:ifun <- M_1[PC]
#             rA:rB     <- M_1[PC+1]
#             valC      <- M_4[PC+2]
#             valP      <- PC+6
#   decode    valA      <- R[rA]
#             valB      <- R[rB]
#   execute   valE      <- valC+valB
#   memory    valM      <- M_4[valE]
#             M_4[valE] <- valA
#   writeback R[rA]     <- valM
#   PC_update PC        <- valP
#-----

```

对 `pipe-full.hcl` 的修改如下：

取指阶段: rmxchg 添加到 valid_instr, need_rigids, need_valC 的条件中

译码阶段: valA=R[rA], valB=R[rB]

执行阶段: valE=valC+valB; aluA=valC, aluB=valB;

访存阶段: mem_read=1, mem_write=1, mem_addr=valE;

这个指令会从内存中读取数据, 也会引发加载/使用冒险, 因此在寄存器的控制中加载/使用冒险的条件中加入此指令。

([去检查程序](#))

```
...
# Instruction code for rmxchg instruction
intsig IRMXCHG 'I_RMXCHG'
...
##### Fetch Stage #####

## What address should instruction be fetched at
int f_pc = [
    # Mispredicted branch. Fetch at incremented PC
    M_icode == IJXX && !M_Cnd : M_valA;
    # Completion of RET instruction.
    W_icode == IRET : W_valM;
    # Default: Use predicted value of PC
    1 : F_predPC;
];

## Determine icode of fetched instruction
int f_icode = [
    imem_error : INOP;
    1: imem_icode;
];

# Determine ifun
int f_ifun = [
    imem_error : FNONE;
    1: imem_ifun;
];

# Is instruction valid?
bool instr_valid = f_icode in
    { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
      IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL,
      ILEAVE, IRMXCHG };

# Determine status code for fetched instruction
int f_stat = [
    imem_error: SADR;
```

```

    !instr_valid : SINS;
    f_icode == IHALT : SHLT;
    1 : SAOK;
];

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

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

# Predict next value of PC
int f_predPC = [
    f_icode in { IJXX, ICALL } : f_valC;
    1 : f_valP;
];

##### Decode Stage #####

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

## What should be the A value?
## Forward into decode stage for valA
int d_valA = [
    D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
    d_srcA == e_dstE : e_valE;      # Forward valE from execute
    d_srcA == M_dstM : m_valM;      # Forward valM from memory
    d_srcA == M_dstE : M_valE;      # Forward valE from memory
    d_srcA == W_dstM : W_valM;      # Forward valM from write back
    d_srcA == W_dstE : W_valE;      # Forward valE from write back
    1 : d_rvalA; # Use value read from register file
];

int d_valB = [
    d_srcB == e_dstE : e_valE;      # Forward valE from execute
    d_srcB == M_dstM : m_valM;      # Forward valM from memory
    d_srcB == M_dstE : M_valE;      # Forward valE from memory
    d_srcB == W_dstM : W_valM;      # Forward valM from write back
    d_srcB == W_dstE : W_valE;      # Forward valE from write back
    1 : d_rvalB; # Use value read from register file
];

##### Execute Stage #####

## Select input A to ALU
int aluA = [
    E_icode in { IRRMOVL, IOPL } : E_valA;
    E_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL, IRMXCHG } : 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, IRMXCHG } : 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;
    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 };

## Generate valA in execute stage
int e_valA = E_valA;    # Pass valA through stage

## Set dstE to RNONE in event of not-taken conditional move
int e_dstE = [
    E_icode == IRRMOVL && !e_Cnd : RNONE;
    1 : E_dstE;
];

##### Memory Stage #####

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

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

/* $begin pipe-m_stat-hcl */
## Update the status
int m_stat = [

```

```

    dmem_error : SADR;
    1 : M_stat;
];
/* $end pipe-m_stat-hcl */

## Set E port register ID
int w_dstE = W_dstE;

## Set E port value
int w_valE = W_valE;

## Set M port register ID
int w_dstM = W_dstM;

## Set M port value
int w_valM = W_valM;

## Update processor status
int Stat = [
    W_stat == SBUB : SAOK;
    1 : W_stat;
];

##### Pipeline Register Control #####

# 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, IRMXCHG } &&
    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, IRMXCHG } &&
    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, IRMXCHG } && 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, IRMXCHG } &&
    E_dstM in { d_srcA, d_srcB};

# Should I stall or inject a bubble into Pipeline Register M?
# At most one of these can be true.
bool M_stall = 0;
# Start injecting bubbles as soon as exception passes through memory
stage
bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR,
SINS, SHLT };

# Should I stall or inject a bubble into Pipeline Register W?
bool W_stall = W_stat in { SADR, SINS, SHLT };
bool W_bubble = 0;

```

测试程序: `test.js`, 在 `misc` 目录下有一个叫 `axchg.js` 跟他内容是一样的。改自 `copy.js`, 实现两个数组的交换, 返回值不变。

```

#Mingxiao Liu 2021010584
#该文件几乎类似于 copy.js
    .pos 0

init:
    irmovl stack,%esp
    irmovl stack,%ebp
    call    main
    halt

.align 4
# Source block
src:
    .long 0x00a

```

```

.long 0x0b0
.long 0xc00

# Destination block
dest:
    .long 0x111
    .long 0x222
    .long 0x333

main:
    pushl    %ebp
    rrmovl   %esp,%ebp
    irmovl   $0x3,%edx
    pushl    %edx
    irmovl   dest,%edx
    pushl    %edx
    irmovl   src,%edx
    pushl    %edx        #准备参数
    call     xchg_block   #调用
    rrmovl   %ebp,%esp
    popl     %ebp
    ret

xchg_block:
    pushl    %ebp
    rrmovl   %esp,%ebp
    pushl    %esi
    pushl    %edi
    pushl    %ebx        #保存调用者寄存器
    mrmovl   0x8(%ebp),%esi    #%esi=src
    mrmovl   0xc(%ebp),%edi    #%edi=dst
    mrmovl   0x10(%ebp),%ecx   #%ecx=len
    xorl     %ebx,%ebx        #%ebx=result
    andl     %ecx,%ecx        #设置条件码
    jmp      test

loop:
    mrmovl   (%esi),%eax       #%eax=*src
    xorl     %eax,%ebx        #result^=*src
    rmxchg   %eax,(%edi)
    rmmovl   %eax,(%esi)      #swap(*src,*dst)

    iaddl    $0x4,%esi        #src++
    iaddl    $0x4,%edi        #dst++
    irmovl   $0x1,%edx

```

```

    subl    %edx,%ecx    #len--
test:
    jg      loop         #while(len>0)
    rrmovl  %ebx,%eax
    popl    %ebx
    popl    %edi
    popl    %esi         #恢复调用者寄存器
    popl    %ebp
    ret

    .pos 0x100
stack:

```

测试结果:

```

thu@ubuntu: ~/Desktop/archlab-handout/sim/pipe
Status = HLT
Condition Codes: Z=1 S=0 O=0
Changed Register State:
%eax: 0x00000000 0x00000cba
%edx: 0x00000000 0x00000001
%esp: 0x00000000 0x00000100
%ebp: 0x00000000 0x00000100
Changed Memory State:
0x0014: 0x0000000a 0x00000111
0x0018: 0x000000b0 0x00000222
0x001c: 0x00000c00 0x00000333
0x0020: 0x00000111 0x0000000a
0x0024: 0x00000222 0x000000b0
0x0028: 0x00000333 0x00000c00
0x00e4: 0x00000000 0x000000f8
0x00e8: 0x00000000 0x0000004d
0x00ec: 0x00000000 0x00000014
0x00f0: 0x00000000 0x00000020
0x00f4: 0x00000000 0x00000003
0x00f8: 0x00000000 0x00000100
0x00fc: 0x00000000 0x00000011
ISA Check Succeeds
CPI: 75 cycles/61 instructions = 1.23

```