ArchLab Documentation

Wu Jialong 2018013418

May 16, 2020

1 Part A

In this part, we are going to write Y86 programs for three functions. They are `sum.ys`, `rsum.ys` and `copy.ys`. To make life easy, we first use the following commands to generate x86 programs and then translate to Y86 programs.

```
gcc -m32 -c examples.c
objdump -d examples.o  > examples.txt
```

The resulting programs are showed in Appendix A. Comments along with the code indicate some essential assignments for registers and coresponding C statements.

2 Part B

In this part, we need to extend the SEQ and PIPE processor to support two new instructions: `iaddl` and `leave`. In the manner of Figure 4-18 in CSAPP2e, we describle the stages of them.

```
            leave
fetch:      icode:ifun <- M_1[PC]
            valP <- PC+2
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
```

```
            iaddl V,rb
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
memory:
writeback:  R[rB] <- valE
PC update:  PC <- valP
```

We first demonstrate our modification on `seq-full.hcl` from different stages:

1. Fetch Stage:
   - add `ILEAVE` into conditions of `instr_valid`.
   - add `IIADDL` into conditions of `instr_valid, need_regids, need_valC`.
2. Decode Stage:

- add `ILEAVE` into conditions of `srcA=REBP, srcB=REBP, dstE=RESP, dstM=REBP`.
- add `IIADDL` into conditions of `srcB=rB, dstE=rB`.

3. Execute Stage:
   - add `ILEAVE` into conditions of `aluA=4, aluB=valB`.
   - add `IIADDL` into conditions of `aluA=valC, aluB=valB, set_cc`.
4. Memory Stage:
   - add `ILEAVE` into conditions of `mem_read, mem_addr=valA`.

For PIPE architecture, it is much easier: modification on `pipe-full.hcl` are almost the same with what we have done on `seq-full.hcl`, except additional special cases processing. For example, in Decode Stage, we add `ILEAVE` into conditions of `d_srcA=REBP`.

There are many special cases and mechanisms to handle. We analysis their influence one by one:

1. data bypassing: Once `src` and `dst` are set properly, it is well done with existing code and need no modification.
2. `ret` instruction, mispredicted branch and exception handling: Well done with existing code.
3. use/load hazard: `leave` do a memory read, so it may **cause use/load hazard and need to be inserted to conditions of use/load hazard**. There are totally four places where use/load hazard are determined.

Modification details are present in Appendix B.

3 Part C

In this part, we succeed to additionaly support one more instructions in the PIPE simulator: `rmxchg` Exchange value in a register with a value in memory.

3.1 Modification

First, we modify assembler in `misc` directory to make `yas` and `yis` to support the new instruction. Details:

- `misc/yas-grammar.lex`: add `rmxchg` into `Instr`.
- `misc/isa.h`: add `I_RMXCHG` into `itype_t`.
- `misc/isa.c`: The following can be done by imitating the code of `rmmovl`.
  - add `rmxchg` into `instruction_set[]`.
  - In function `step_state`: add `I_RMXCHG` into conditions of `need_regids, need_imm`; and complete the C implement of `rmxchg` in the `switch` statement.

To make PIPE support it, we describe the instruction in stages:

```
            rmxchg rA,D(rB)
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
```

Note that in memory stage, there are both memory read and write. It **do not introduce a hazard** because memory read could be done in a short delay, but write would be done only on the next rising edge.

Then similiar with what we have done in Part B, we modify `pipe-full.hcl`:

1. Declarations: add `intsig IRMXCHG 'I_RMXCHG'`.
2. Fetch Stage:
   - add `IRMXCHG` into conditions of `instr_valid, need_regids, need_valC`.
3. Decode Stage:
   - add `IRMXCHG` into conditions of `d_srcA=D_rA, d_srcB=D_rB, d_dstM=D_rA`.
4. Execute Stage:
   - add `IRMXCHG` into conditions of `aluA=E_valC, aluB=E_valB`.
5. Memory Stage:
   - add `IRMXCHG` into conditions of `mem_read, mem_write, mem_addr=M_valE`.
6. Pipeline Register Control: add `IRMXCHG` into conditions of load/use hazards.

3.2 Test

To make sure our implement do not corrupt existing code, we first perform regression tests:

```
$ make clean; make VERSION=full
$ cd 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
```

Finally, we test our additional instruction. `test.ys` is modified from `copy.ys` and its function is to swap each corresponding element in two array with the same length.

`yas` and `yis` test results:

```
$ cd misc; ./yas test.ys; ./yis test.yo
Stopped in 125 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%esp:   0x00000000      0x00000500
%ebp:   0x00000000      0x00000500

Changes to memory:
0x0014: 0x0000000a      0x00000111
0x0018: 0x000000b0      0x00000222
0x001c: 0x00000c00      0x00000333
0x0020: 0x00000111      0x0000000a
0x0024: 0x00000222      0x000000b0
0x0028: 0x00000333      0x00000c00
0x04dc: 0x00000000      0x00000cba
0x04e4: 0x00000000      0x000004f8
0x04e8: 0x00000000      0x0000004d
0x04ec: 0x00000000      0x00000020
0x04f0: 0x00000000      0x0000002c
0x04f8: 0x00000000      0x00000500
0x04fc: 0x00000000      0x00000011
```

`psim` test results:

```
$ cd pipe; ./psim -t ../misc/test.yo
[too long, omitted]
146 instructions executed
Status = HLT
Condition Codes: Z=1 S=0 O=0
Changed Register State:
%eax:   0x00000000      0x00000cba
%esp:   0x00000000      0x00000500
%ebp:   0x00000000      0x00000500
Changed Memory State:
0x0014: 0x0000000a      0x00000111
0x0018: 0x000000b0      0x00000222
0x001c: 0x00000c00      0x00000333
```

```
0x0020: 0x00000111      0x0000000a
0x0024: 0x00000222      0x000000b0
0x0028: 0x00000333      0x00000c00
0x04dc: 0x00000000      0x00000cba
0x04e4: 0x00000000      0x000004f8
0x04e8: 0x00000000      0x0000004d
0x04ec: 0x00000000      0x00000020
0x04f0: 0x00000000      0x0000002c
0x04f8: 0x00000000      0x00000500
0x04fc: 0x00000000      0x00000011
ISA Check Succeeds
CPI: 142 cycles/125 instructions = 1.14
```

These indicate that a Y86 program with instruction `rmxchg` works well, which successfully swap two arrays, `0x0014-0x001c` and `0x0020-0x0028`.

Appendix A Y86 Programs and Comments for Part A

`sum.ys`: Iteratively sum linked list elements

```
1  # wujialong 2018013418
2      .pos 0
3  init:
4      irmovl Stack,%esp
5      irmovl Stack,%ebp
6      call Main
7      halt
8
9      .align 4
10 ele1:
11     .long 0x00a
12     .long ele2
13 ele2:
14     .long 0x0b0
15     .long ele3
16 ele3:
17     .long 0xc00
18     .long 0
19
20 Main:
21     pushl %ebp
22     rrmovl %esp,%ebp
23     irmovl ele1,%edx
24     pushl %edx                      # prepare parameters
25     call sum_list                   # call function
26     rrmovl %ebp,%esp
27     popl %ebp
28     ret
29
30 sum_list:
31     pushl %ebp
32     rrmovl %esp,%ebp
33     irmovl $0x10,%ecx
34     subl %ecx,%esp
35     irmovl $0x0,%ecx
36     rmmovl %ecx,-0x4(%ebp)          # val = -4(%ebp) = 0
37     jmp    end
38 loop:
39     mrmovl 0x8(%ebp),%eax
40     mrmovl (%eax),%eax              # %eax = ls->val
41
42     mrmovl 0xfffffffc(%ebp),%ecx    # %ecx = -4(%ebp) = val
```

```
    addl %eax,%ecx
    rmmovl %ecx,0xfffffffc(%ebp)    # val += ls->val

    mrmovl 0x8(%ebp),%eax           # %eax = ls
    mrmovl 0x4(%eax),%eax           # %eax = ls->next
    rmmovl %eax,0x8(%ebp)           # ls = ls->next
end:
    mrmovl 0x8(%ebp),%ecx
    irmovl $0x0,%eax
    xorl %eax,%ecx
    jne loop                        # while (ls)
    mrmovl 0xfffffffc(%ebp),%eax
    rrmovl %ebp,%esp
    popl %ebp
    ret

    .pos 0x500
Stack:
```

`rsum.ys`: Recursively sum linked list elements

```
# wujialong 2018013418
    .pos 0
init:
    irmovl Stack,%esp
    irmovl Stack,%ebp
    call Main
    halt

    .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                      # prepare parameters
    call rsum_list                  # call function
```

```
    rrmovl %ebp,%esp
    popl %ebp
    ret

rsum_list:
    pushl %ebp
    rrmovl %esp,%ebp
    irmovl $0x18,%edx
    subl %edx,%esp
    irmovl $0x0,%edx
    mrmovl 0x8(%ebp),%eax           # %eax = ls
    xorl %edx,%eax
    jne else                        # else
    irmovl $0x0,%eax
    jmp return                      # if (!ls) return 0
else:
    mrmovl 0x8(%ebp),%eax           # %eax = ls
    mrmovl (%eax),%eax
    rmmovl %eax,0xfffffff0(%ebp)    # -0xf(%ebp) = val = ls->val

    mrmovl 0x8(%ebp),%eax           # %eax = ls
    mrmovl 0x4(%eax),%eax           # %eax = ls->next
    irmovl $0xc,%edx
    subl %edx,%esp
    pushl %eax                      # prepare parameter = ls->next
    call rsum_list                  # recursive, %eax = rest

    irmovl $0x10,%edx
    addl %edx,%esp                  # restore the stack
    mrmovl 0xfffffff0(%ebp),%edx    # %edx = -0xf(%ebp) = val
    addl %edx,%eax                  # %eax = val + rest
return:
    rrmovl %ebp,%esp
    popl %ebp
    ret

    .pos 0x500
Stack:
```

`copy.ys`: Copy a source block to a destination block

```
# wujialong 2018013418
    .pos 0
init:
    irmovl Stack,%esp
    irmovl Stack,%ebp
```

```
6       call Main
7       halt
8
9       .align 4
10  # Source block
11  src:
12      .long 0x00a
13      .long 0x0b0
14      .long 0xc00
15  # Destination block
16  dest:
17      .long 0x111
18      .long 0x222
19      .long 0x333
20
21  Main:
22      pushl %ebp
23      rrmovl %esp,%ebp
24      irmovl 0x3,%edx
25      pushl %edx
26      irmovl dest,%edx
27      pushl %edx
28      irmovl src,%edx
29      pushl %edx                      # preparing paramters
30      call copy_block                 # call function
31      rrmovl %ebp,%esp
32      popl %ebp
33      ret
34
35  copy_block:
36      pushl %ebp
37      rrmovl %esp,%ebp
38      irmovl $0x10,%ecx
39      subl %ecx,%esp
40      irmovl $0x0,%ecx                # result = 0
41      rmmovl %ecx,0xfffffff8(%ebp)    # -8(%ebp) = result
42      jmp end
43  loop:
44      mrmovl 0x8(%ebp),%eax           # %eax = src
45      mrmovl (%eax),%eax              # %eax = val
46
47      mrmovl 0xc(%ebp),%edx           # %edx = dest
48      rmmovl %eax,(%edx)              # *dest = val
49
50      mrmovl 0xfffffff8(%ebp),%ecx    # %ecx = result
```

```
51      xorl %eax,%ecx
52      rmmovl %ecx,0xffffffff8(%ebp)    # result ^= val
53
54      mrmovl 0x10(%ebp),%ecx
55      irmovl $0x1,%edx
56      subl %edx,%ecx
57      rmmovl %ecx,0x10(%ebp)           # len--
58
59      mrmovl 0x8(%ebp),%eax            # %eax = src
60      irmovl 0x4,%edx
61      addl %eax,%edx
62      rmmovl %edx,0x8(%ebp)            # src++
63
64      mrmovl 0xc(%ebp),%eax            # %eax = dest
65      irmovl 0x4,%edx
66      addl %eax,%edx
67      rmmovl %edx,0xc(%ebp)            # dest++
68  end:
69      mrmovl 0x10(%ebp),%ecx
70      irmovl $0x0,%edx
71      subl %edx,%ecx
72      jg loop                          # while(len>0)
73      mrmovl 0xffffffff8(%ebp),%eax
74      rrmovl %ebp,%esp
75      popl %ebp
76      ret
77
78      .pos 0x500
79  Stack:
```

Appendix B Modification in HCL for Part B

Modifications on `seq-full.hcl`:

```
diff --git a/seq/seq-full.hcl b/seq/seq-full.hcl
index a998fef..0762e34 100644
--- a/seq/seq-full.hcl
+++ b/seq/seq-full.hcl
@@ -1,3 +1,4 @@
+#/* Name: wujialong, ID: 2018013418 */
 #/* $begin seq-all-hcl */
 ####################################################################
 #  HCL Description of Control for Single Cycle Y86 Processor SEQ   #
@@ -109,16 +110,16 @@ int ifun = [

 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 Stage    ###################################

@@ -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
 ];

@@ -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, ILEAVE, IIADDL } : 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 Stage    ###################################

 ## 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 };
@@ -189,7 +193,7 @@ 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 } : valA;
+       icode in { IPOPL, IRET, ILEAVE } : valA;
        # Other instructions don't need address
 ];
```

Modification on `pipe-full.hcl`: Almost the same with what have been done with SEQ, only present unique modifications about load/use hazards.

```
@@ -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};

 # Should I stall or inject a bubble into Pipeline Register M?
```

Appendix C List of Manually Changed Files in Part C

Here is a total list of manually changed files in `sim` directory for supporting extra instruction. TA could use `diff` command to see detailed modifications.

```
sim/misc/isa.c
sim/misc/isa.h
sim/misc/yas-grammar.lex
sim/pipe/pipe-full.hcl
```

Note we **do not change HCL file of SEQ** because it is not required in the Part C of the assignment.