Project 1: Optimizing the Performance of a Pipelined Processor

[518021910753 严秉昊 1026148725@qq.com]

[518030910022 杨涵章 linqinluli@sjtu.edu.cn]

1. Introduction

1. Brief introduction: this experiment is divided into three parts. The assembly, assembly line and its optimization are involved. PartA lets us use Y86 assembly language to implement three functions, namely, the sum of the elements in the linked list, the recursive realization of the sum of the elements in the linked list, the copy and fetch or operation of the elements in the linked list. PartB is mainly in the seq folder, modify "*seq-full.hcl*", to add an assembly instruction *"iaddl"*;The PartC requires the optimization of the pipeline and the modification of the pipe - full. HCL and *"ncopy.ys"* to make the *"ncopy.ys"* run as fast as possible. After getting the experimental topic, our team made efforts for several days to achieve all the requirements of the experiment and summarized the possible problems and solutions in the experiment, which will be described in detail below.

2. Introduction of members and division of labor: Yan Binghao is in charge of PartA, and Yang Hanzhang is in charge of PartB/C. In terms of report writing, Yan Binghao was mainly responsible for it, and Yang Hanzhang supplemented it.

1. Experiments

A. Part A

i. Analysis

1. Overall analysis: the main content of PartA is that let's use Y86 assembly language to implement three functions, in *"example.c"*, the function is: the sum of the elements in the linked list *"sum_list()"*, the recursive implementation of the sum of the elements in the linked list *"rsum_list()"*, the linked list element copy and fetch or operation *"copy_block()"*.The content of the experiment is easy to understand. We know that elements in a linked list have not only data items, but also Pointers to subsequent

nodes, and the sample list tested in the experiment has been given, which we will show in the detailed code. Although the experiment is not difficult to understand, but it is difficult to implement.

2. Experiment difficulty: this experiment used a new Y86 language that had not been touched before. After learning, it was found to be a simplified version of X86, not a real language, but a virtual one created by the author of this book. The difficulty of this experiment mainly lies in the writing of assembly instructions. I think the most difficult part is to implement recursive call, because we have never done similar implementation in the study of embedded courses. The second is the assembly of the main function, not familiar with the previous, in the execution of the stack to the base pointer to the push operation to save. The details of the code and its comments are given in the next section, which is not detailed here.

3. Core technology: the core of this experiment is to achieve addition, recursive addition and replication of the function, loop in the loop process, which should pay attention to:

   A. In the Y86 language, several *move* instructions are somewhat different from X86 in that they move the previous register or immediate number in the instruction to the register that appears after it, so you should pay special attention when writing code, otherwise it will be easily reversed.
   B. In the *loop* cycle, we can see, we initialize *%eax* register the value is 0 before, used to store the current sum, using the *%edx* register to store the current pointer, every visit to a pointer points to area of memory, and in the "address + 4" operation, not directly on the number immediately to the register, and will put into another register, arithmetic operation on two registers, before it has to do with SHL, etc in X86 operating principle.

ii. Code

I have not found a way to effectively format the Y86 language, so the code layout is a bit ugly, please forgive me

1. *sum_list()* -The assembly code version

```
# Execution begins at address 0
        .pos        0
init:   irmovl      Stack, %esp         #set stack pointer
```

```
        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 , %eax        #set ele1 as the sum beginning
        pushl       %eax
        call        sumlist                #function call
        rrmovl      %ebp , %esp
        popl        %ebp
        ret

# int sum_list(list_ptr ls)
sumlist:
        pushl         %ebp
        rrmovl        %esp , %ebp
        xorl          %eax, %eax             #set eax (val) = 0
        mrmovl        8(%ebp) , %edx

Loop:
        mrmovl      (%edx) , %ecx          #ls->val    ==> %ecx
        addl        %ecx , %eax            #val += ls->val
        irmovl      $4 , %edi
        addl        %edi , %edx            #next ==> edx
        mrmovl      (%edx), %esi
        rrmovl      %esi , %edx            #ls->next ==>edx
        andl        %edx , %edx            #set condition codes
        jne         Loop                   #if ls != 0 goto Loop

End:
        rrmovl      %ebp , %esp
        popl        %ebp
        ret
        .pos        0x100
Stack:
```

2. *rsum_list() -The assembly code version*

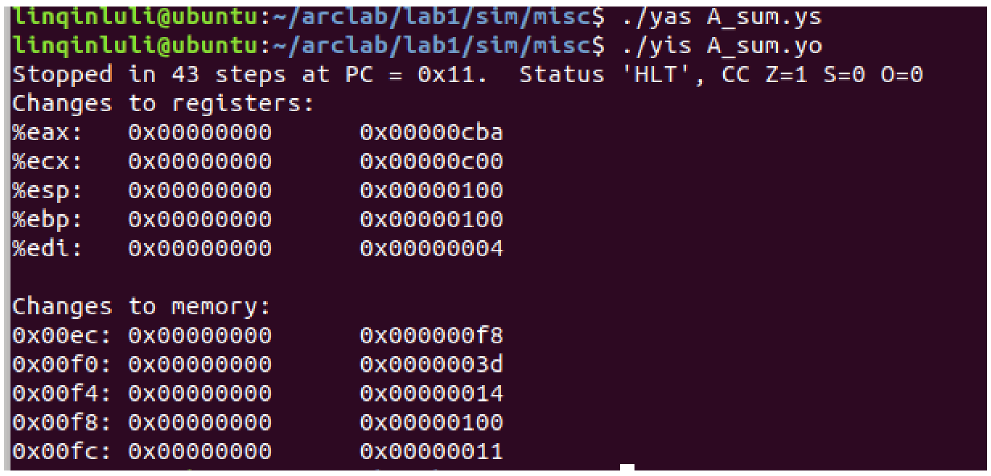
```
# Execution begins at address 0
        .pos        0
init:
        irmovl      Stack, %esp
        irmovl      Stack, %ebp
        call        Main
        halt

# Sample linked list
        .align      4
ele1:
        .long       0x00a
        .long       ele2
ele2:
        .long       0x0b0
        .long       ele3
ele3:
        .long       0xc00
        .long       0

Main:
        pushl       %ebp                #save
        rrmovl      %esp , %ebp
        irmovl      $4 , %esi           #set the immediate number
        irmovl      ele1 , %eax
        pushl       %eax
        call        rsum_list
        rrmovl      %ebp , %esp
        popl        %ebp                #restore
        ret

rsum_list:
        pushl       %ebp
        rrmovl      %esp , %ebp         #save
        pushl       %ebx
        subl        %esi , %esp
        xorl        %eax , %eax
        mrmovl      8(%ebp),%edx
        andl        %edx , %edx         #end of the recursion
        je          End
        mrmovl      (%edx) , %ebx
        addl        %esi , %edx
        mrmovl      (%edx) , %edi
        rmmovl      %edi , (%esp)
        call        rsum_list           #recursive call
        addl        %ebx , %eax         #eax(val) += ebx

End:
        addl        %esi , %esp
        popl        %ebx
        popl        %ebp                #restore
        ret

        .pos        0x100
Stack:
```

3. *copy_block() -The assembly code version*

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

# Source block
   .align 4
src:
         .long      0x00a
         .long      0x0b0
         .long      0xc00
# Destination block
dest:
        .long       0x111
        .long       0x222
        .long       0x333

Main:
        pushl       %ebp
        rrmovl      %esp ,    %ebp
        irmovl      $12 ,    %esi
        subl        %esi ,    %esp
        irmovl      src ,    %eax                #save source block
        rmmovl      %eax , (%esp)
        irmovl      dest , %eax                  #save destination block
        rmmovl      %eax , 4(%esp)
        irmovl      $3, %eax                     #set length
        rrmovl      %eax, %ecx
        call        copy_block
        popl        %ebp
        ret

copy_block:
        pushl       %ebp
        rrmovl      %esp, %ebp
        xorl        %eax , %eax

        mrmovl      12(%ebp) , %edx              #edx <==>dest
        mrmovl      8(%ebp) , %esi               #esi <==> src

Loop:
        mrmovl      (%esi) , %ebx
        rmmovl      %ebx , (%edx)                #copy element
        xorl        %ebx , %eax                  #checksum
        irmovl      $4 ,    %edi
        addl        %edi , %edx                  #dest++
        addl        %edi , %esi                  #src++
        irmovl      $1, %edi
        subl        %edi , %ecx                  #len--
        jne         Loop

End:    popl        %ebp
        ret

        .pos 0x100
Stack:
```

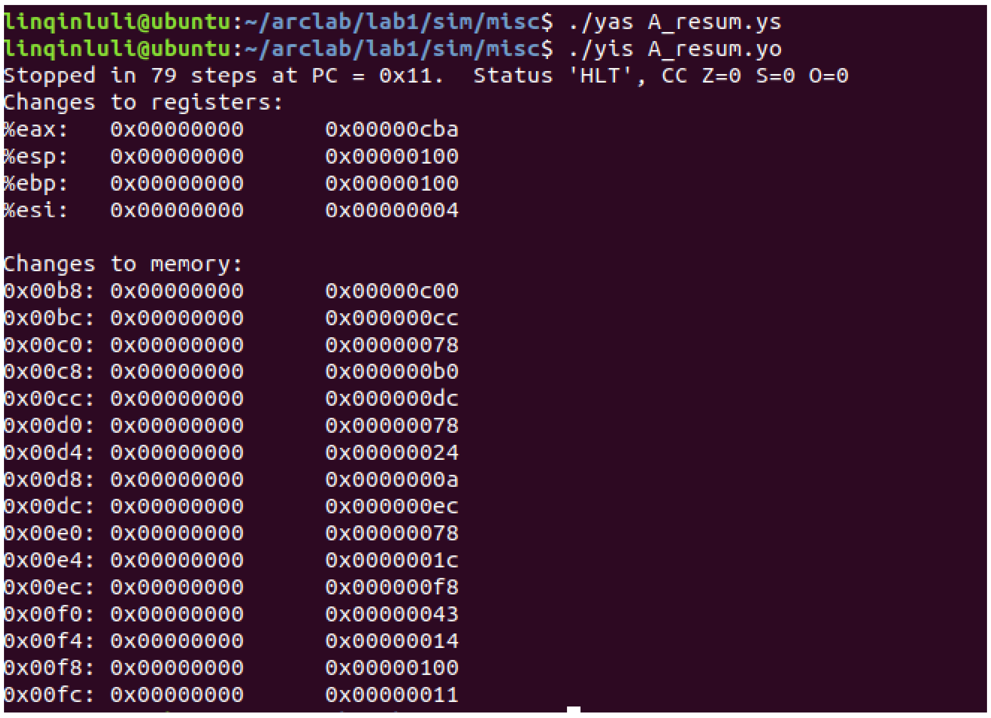
iii. Evaluation

After the assembly instruction is written and compiled, we execute the executable file, and the result is exactly the same as the Evaluation standard. The contents of register *%eax* are the final calculation result. The experimental results are as follows:



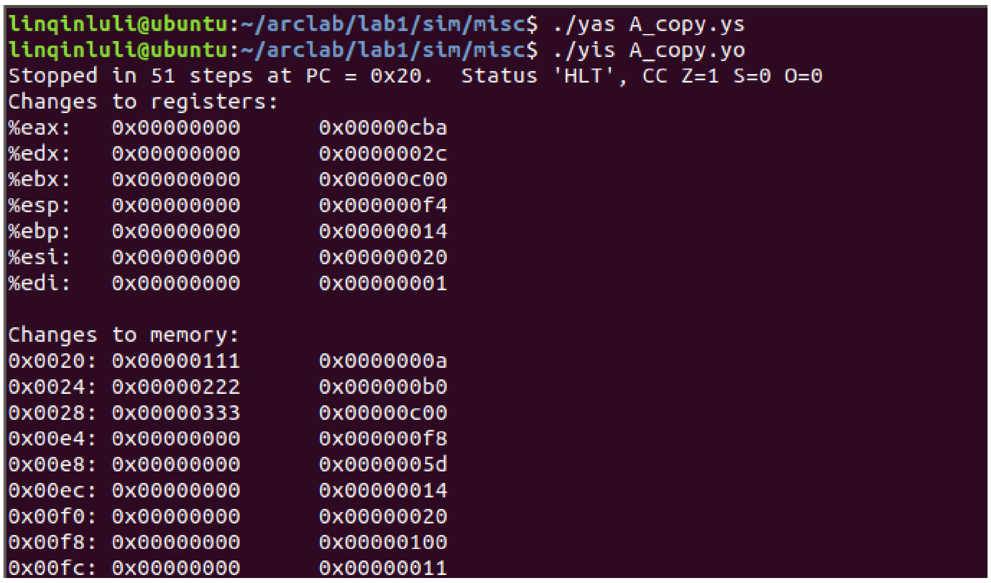
```
linqinluli@ubuntu:~/arclab/lab1/sim/misc$ ./yas A_sum.ys
linqinluli@ubuntu:~/arclab/lab1/sim/misc$ ./yis A_sum.yo
Stopped in 43 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%ecx:   0x00000000      0x00000c00
%esp:   0x00000000      0x00000100
%ebp:   0x00000000      0x00000100
%edi:   0x00000000      0x00000004

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

Figure 1: sum_list()



```
linqinluli@ubuntu:~/arclab/lab1/sim/misc$ ./yas A_resum.ys
linqinluli@ubuntu:~/arclab/lab1/sim/misc$ ./yis A_resum.yo
Stopped in 79 steps at PC = 0x11.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%esp:   0x00000000      0x00000100
%ebp:   0x00000000      0x00000100
%esi:   0x00000000      0x00000004

Changes to memory:
0x00b8: 0x00000000      0x00000c00
0x00bc: 0x00000000      0x000000cc
0x00c0: 0x00000000      0x00000078
0x00c8: 0x00000000      0x000000b0
0x00cc: 0x00000000      0x000000dc
0x00d0: 0x00000000      0x00000078
0x00d4: 0x00000000      0x00000024
0x00d8: 0x00000000      0x0000000a
0x00dc: 0x00000000      0x000000ec
0x00e0: 0x00000000      0x00000078
0x00e4: 0x00000000      0x0000001c
0x00ec: 0x00000000      0x000000f8
0x00f0: 0x00000000      0x00000043
0x00f4: 0x00000000      0x00000014
0x00f8: 0x00000000      0x00000100
0x00fc: 0x00000000      0x00000011
```

Figure 2: surm_list()



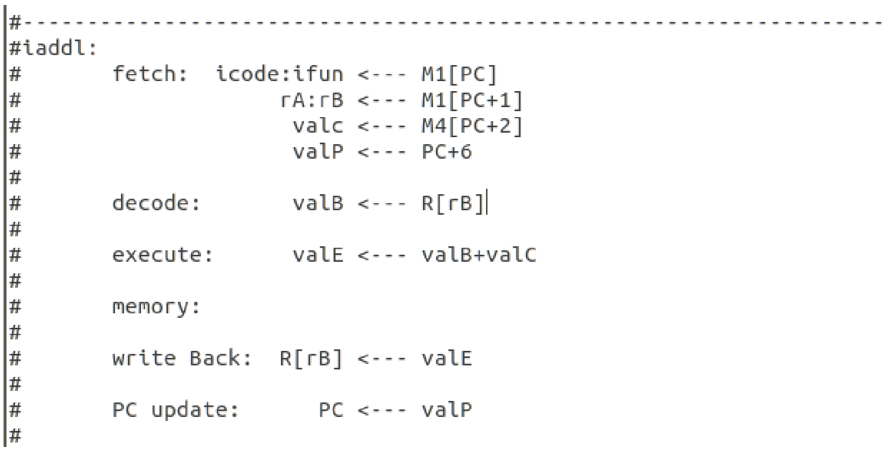
```
linqinluli@ubuntu:~/arclab/lab1/sim/misc$ ./yas A_copy.ys
linqinluli@ubuntu:~/arclab/lab1/sim/misc$ ./yis A_copy.yo
Stopped in 51 steps at PC = 0x20.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%edx:   0x00000000      0x0000002c
%ebx:   0x00000000      0x00000c00
%esp:   0x00000000      0x000000f4
%ebp:   0x00000000      0x00000014
%esi:   0x00000000      0x00000020
%edi:   0x00000000      0x00000001

Changes to memory:
0x0020: 0x00000111      0x0000000a
0x0024: 0x00000222      0x000000b0
0x0028: 0x00000333      0x00000c00
0x00e4: 0x00000000      0x000000f8
0x00e8: 0x00000000      0x0000005d
0x00ec: 0x00000000      0x00000014
0x00f0: 0x00000000      0x00000020
0x00f8: 0x00000000      0x00000100
0x00fc: 0x00000000      0x00000011
```

Figure 3: copy_block()

B. Part B

i. Analysis

This experiment requires that in *“sim/seq”* folder, the *“seq_full.hcl”* file is modified, and two new instructions, *“iaddl”* and *“leave”*, are added. Referring to the content of chapter 4 of *“Computer Systems: A Programmer's Perspective”*, and comparing the calculation process of *irmovl*,*opl* and *popl*, through analysis, we know that the corresponding changes should be made in each part of the pipeline, and the calculation process of *“iaddl”* is as follows:



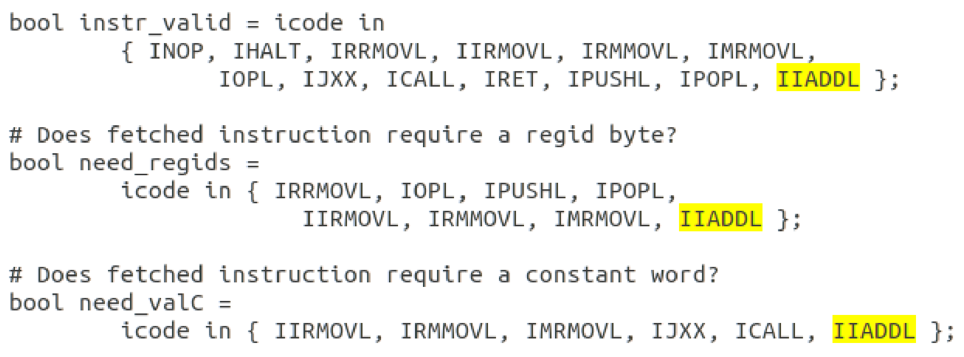
```
#------------------------------------------------------------------
#iaddl:
#       fetch:  icode:ifun <--- M1[PC]
#                    rA:rB <--- M1[PC+1]
#                     valc <--- M4[PC+2]
#                     valP <--- PC+6
#
#       decode:       valB <--- R[rB]
#
#       execute:      valE <--- valB+valC
#
#       memory:
#
#       write Back:  R[rB] <--- valE
#
#       PC update:      PC <--- valP
#
```

Figure 4: The implementation of the *"iaddl"*

ii. Code

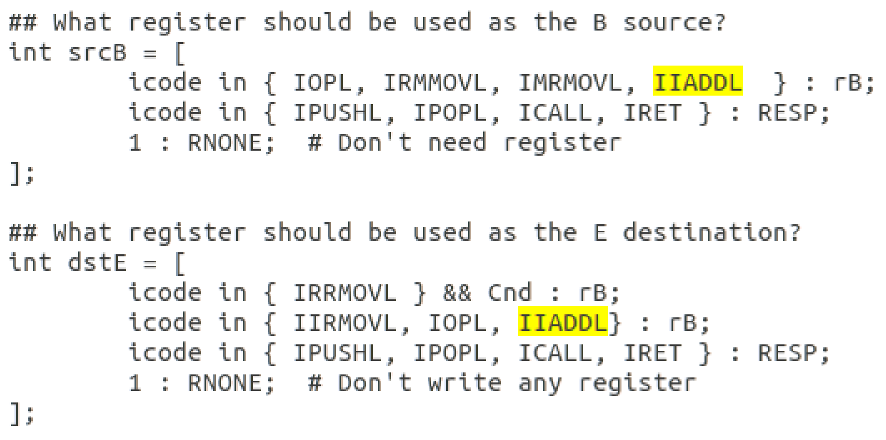
In order to show the part we changed, this part of the code is given in the form of a picture, and the part we changed is marked in yellow.

The modification of the relevant *"seq_full"* file is as follows:



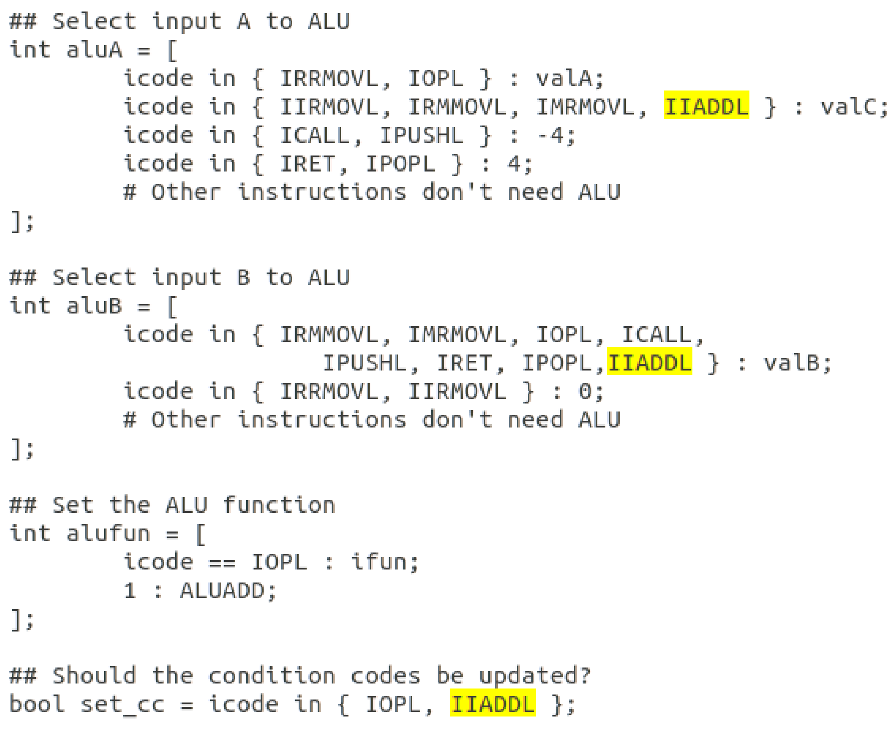
```
bool instr_valid = icode in
        { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
               IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL };

# 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 };
```

Figure 5: Fetch Stage



```
## 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;
        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 } : RESP;
        1 : RNONE;  # Don't write any register
];
```

Figure 6: Decode Stage

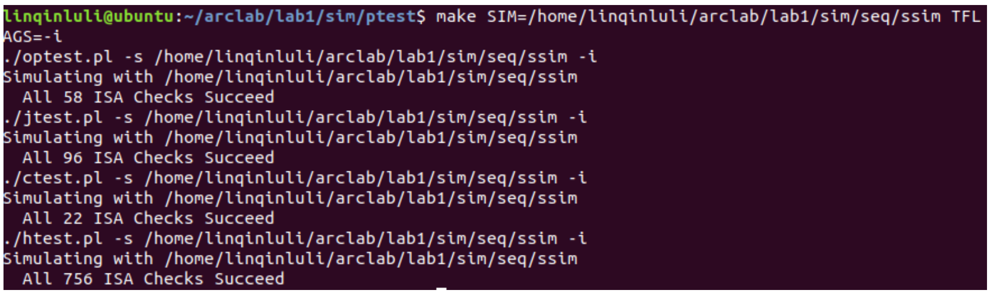


```
## Select input A to ALU
int aluA = [
        icode in { IRRMOVL, IOPL } : valA;
        icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : valC;
        icode in { ICALL, IPUSHL } : -4;
        icode in { IRET, IPOPL } : 4;
        # Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
        icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                      IPUSHL, IRET, IPOPL,IIADDL } : 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 };
```

Figure 7: Execute Stage

iii. Evaluation

Experimental results:
Executor: make SIM=../seq/ssim TFLAGS = - i.
The results are as follows



```
linqinluli@ubuntu:~/arclab/lab1/sim/ptest$ make SIM=/home/linqinluli/arclab/lab1/sim/seq/ssim TFL
AGS=-i
./optest.pl -s /home/linqinluli/arclab/lab1/sim/seq/ssim -i
Simulating with /home/linqinluli/arclab/lab1/sim/seq/ssim
  All 58 ISA Checks Succeed
./jtest.pl -s /home/linqinluli/arclab/lab1/sim/seq/ssim -i
Simulating with /home/linqinluli/arclab/lab1/sim/seq/ssim
  All 96 ISA Checks Succeed
./ctest.pl -s /home/linqinluli/arclab/lab1/sim/seq/ssim -i
Simulating with /home/linqinluli/arclab/lab1/sim/seq/ssim
  All 22 ISA Checks Succeed
./htest.pl -s /home/linqinluli/arclab/lab1/sim/seq/ssim -i
Simulating with /home/linqinluli/arclab/lab1/sim/seq/ssim
  All 756 ISA Checks Succeed
```

Figure 8: Experimental results

C. Part C

i. Analysis

1. First we need to analyze the *"guide"* in the *"ncopy.c"* source code, the implementation is a *"copyblock"* function *"ncopy"* and finally return a positive integer, which represents the number or length of "value" in *"src"* or *"dst"*. And what we're going to do in this part is implement this through the Y86-64 programming language and make the program execute as fast as possible.
2. Part C is mainly in the *"sim/pipe"* folder. The task is to modify *"ncopy.ys"* and *"pipe full. hcl"* files to make *"ncopy.ys"* run as fast as possible.
3. Among them, we modified the file *"ncopy.ys"* to use the *"iaddl"* instruction implemented by the PartB, and then the speed was improved. The implementation method of the instruction was similar to that of the PartB, which is not detailed here.
4. In the experiment, the *"ncopy"* function in *"ncopy.ys"* has been cyclically expanded for a total of 4 times. And in the original function there is a load use risk, because the load use risk so you need to pause one or more cycles, we have to improve on this.

ii. Code

```
ncopy:
        xorl        %eax , %eax
        iaddl       $-4 , %edx              #len = len -4
        andl        %edx , %edx
        jl          remian
Loop:
        mrmovl      (%ebx) , %esi           #loop unrolling
        mrmovl    4(%ebx),%edi            #load-use hazard
        rmmovl    %esi , (%ecx)
        andl      %esi ,%esi
        jle       LNpos1
        iaddl     $1 , %eax
LNpos1:
        rmmovl    %edi , 4(%ecx)
        andl      %edi , %edi
        jle       LNpos2
        iaddl     $1, %eax
LNpos2:
        mrmovl    8(%ebx) , %esi
        mrmovl    12(%ebx), %edi          #load-use hazard
        rmmovl    %esi , 8(%ecx)
        andl      %esi , %esi
        jle       LNpos3
        iaddl     $1 , %eax
LNpos3:
        rmmovl    %edi , 12(%ecx)
        andl      %edi , %edi
        jle       nextLoop
        iaddl     $1, %eax
nextLoop:
        iaddl     $16, %ebx
        iaddl     $16, %ecx
        iaddl     $-4, %edx
        jge       Loop

remian:
        iaddl     $4 , %edx               # Restore the true len
        iaddl     $-1, %edx
        jl        Done
        mrmovl    (%ebx) , %esi
        mrmovl    4(%ebx), %edi
        rmmovl    %esi , (%ecx)
        andl      %esi , %esi
        jle       rNpos
        iaddl     $1 , %eax
rNpos:
        iaddl     $-1, %edx
        jl        Done
        rmmovl    %edi , 4(%ecx)
        andl      %edi , %edi
        jle       rNpos1
        iaddl       $1, %eax
rNpos1:
        iaddl       $-1 , %edx
        jl          Done
        mrmovl      8(%ebx) , %esi
        rmmovl      %esi , 8(%ecx)
        andl        %esi ,%esi
        jle         Done
        iaddl       $1 , %eax
```

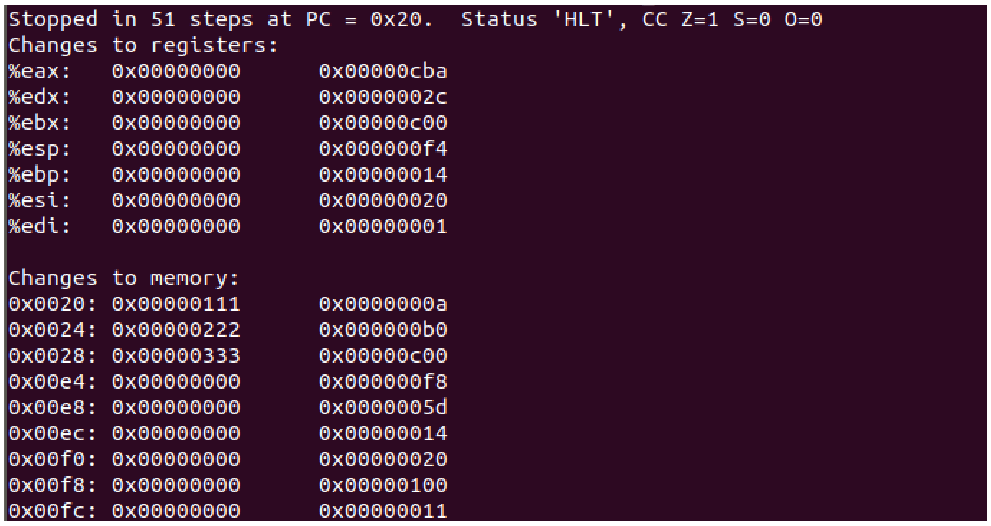
iii. Evaluation

1. Building a new simulator

   unix> make drivers

   results:
   With this instruction we can create the following two useful "driver" files:
   1.sdriver.yo
   2.ldriver.yo
2. Testing on ISA simulator

   unix> ../misc/yis sdriver.yo



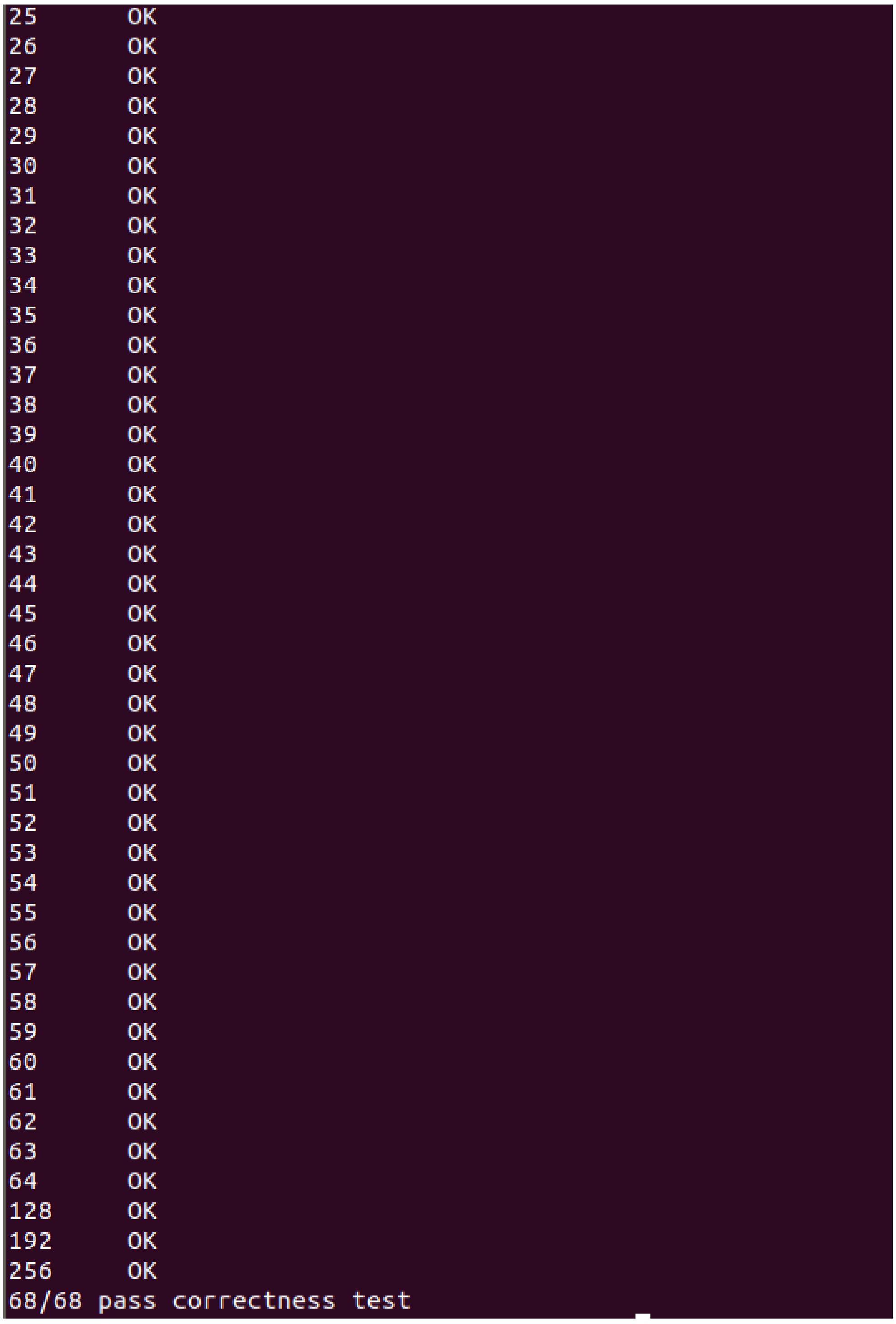
```
Stopped in 51 steps at PC = 0x20.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%edx:   0x00000000      0x0000002c
%ebx:   0x00000000      0x00000c00
%esp:   0x00000000      0x000000f4
%ebp:   0x00000000      0x00000014
%esi:   0x00000000      0x00000020
%edi:   0x00000000      0x00000001

Changes to memory:
0x0020: 0x00000111      0x0000000a
0x0024: 0x00000222      0x000000b0
0x0028: 0x00000333      0x00000c00
0x00e4: 0x00000000      0x000000f8
0x00e8: 0x00000000      0x0000005d
0x00ec: 0x00000000      0x00000014
0x00f0: 0x00000000      0x00000020
0x00f8: 0x00000000      0x00000100
0x00fc: 0x00000000      0x00000011
```

Figure : Result1

3. Test correctness:

   unix>./correctness.pl

   results:



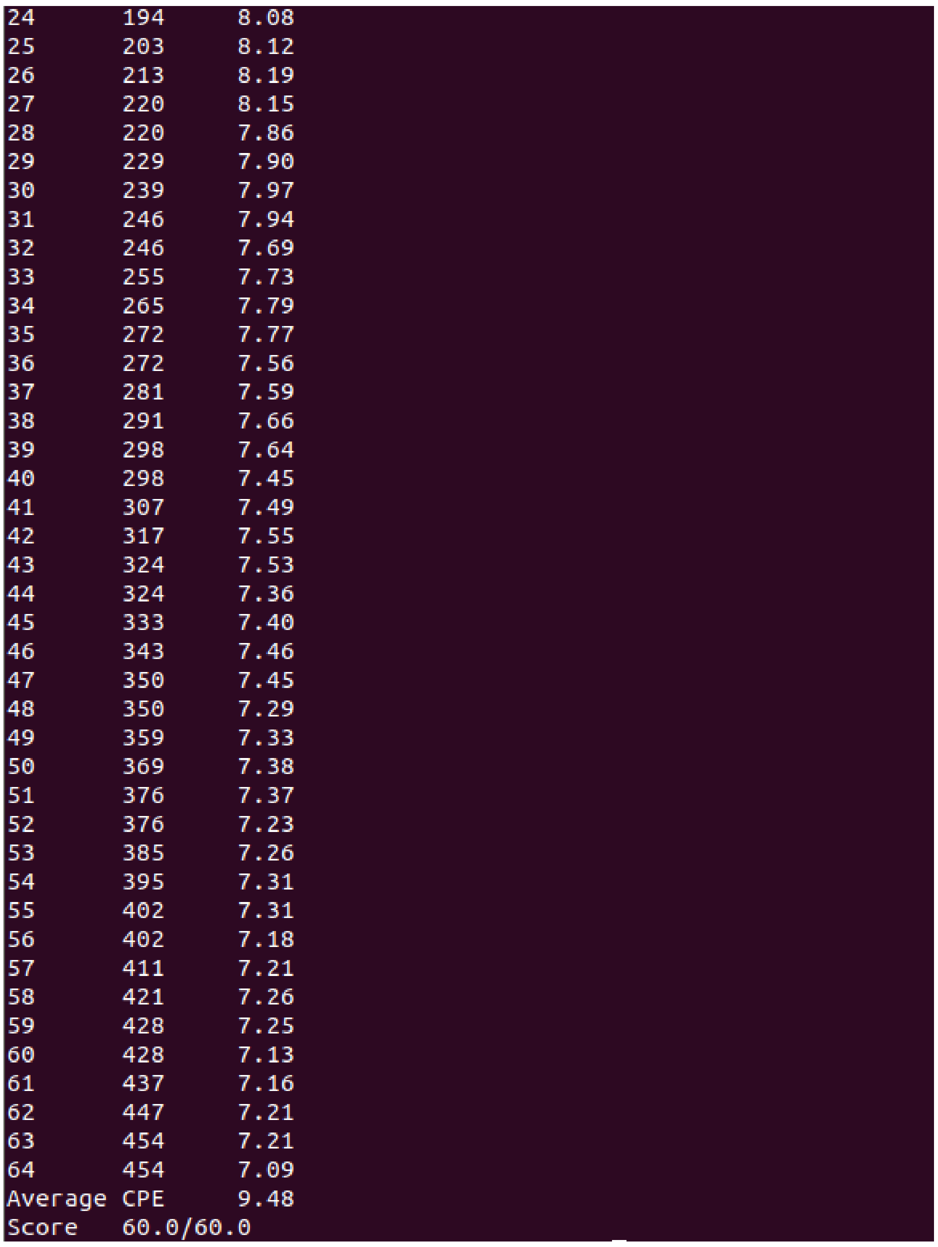
```
25      OK
26      OK
27      OK
28      OK
29      OK
30      OK
31      OK
32      OK
33      OK
34      OK
35      OK
36      OK
37      OK
38      OK
39      OK
40      OK
41      OK
42      OK
43      OK
44      OK
45      OK
46      OK
47      OK
48      OK
49      OK
50      OK
51      OK
52      OK
53      OK
54      OK
55      OK
56      OK
57      OK
58      OK
59      OK
60      OK
61      OK
62      OK
63      OK
64      OK
128     OK
192     OK
256     OK
68/68 pass correctness test
```

Figure 10: Correctness detection

4. Benchmark program:

unix>./benchmark.pl

The CPE time is detected by the benchmark in the y86-code file.



```
24      194     8.08
25      203     8.12
26      213     8.19
27      220     8.15
28      220     7.86
29      229     7.90
30      239     7.97
31      246     7.94
32      246     7.69
33      255     7.73
34      265     7.79
35      272     7.77
36      272     7.56
37      281     7.59
38      291     7.66
39      298     7.64
40      298     7.45
41      307     7.49
42      317     7.55
43      324     7.53
44      324     7.36
45      333     7.40
46      343     7.46
47      350     7.45
48      350     7.29
49      359     7.33
50      369     7.38
51      376     7.37
52      376     7.23
53      385     7.26
54      395     7.31
55      402     7.31
56      402     7.18
57      411     7.21
58      421     7.26
59      428     7.25
60      428     7.13
61      437     7.16
62      447     7.21
63      454     7.21
64      454     7.09
Average CPE     9.48
Score   60.0/60.0
```

2. Conclusion

A. Problems

1. Problems such as unable to locate software packages often occur during compilation. Solution: search the Internet for solutions and install software packages.TK and TCL, in particular, want version 8.5, version 8.6 is not ok, because some features have been deleted.
2. 2. Unable to obtain locks. This and the operating system to talk about the process synchronization is closely related to the process of removing the lock can be, there is a tutorial online.
3. 3. The experimental course is difficult to understand and requires careful reading to understand what we are trying to achieve.
4. 4. Coding is difficult and involves new languages, so you should

read the book CSAPP to learn.

B. Achievements

It took our group about two whole days to finish the homework. In general, the work is very comprehensive and difficult.
Our team spent nearly one morning on the work to understand the experiment requirements and process on the PDF, and another afternoon to learn the relevant knowledge of Y86 language, and at the same time to review the fourth chapter of the in-depth understanding of the computer system. We spent a whole day on the preparatory work before the experiment. However, the sharpening of the knife does not miss the wood work, we in the second day of the experiment or relatively smooth. At the same time, the three of us had a clear division of labor and conducted targeted learning according to different requirements of different parts, which greatly improved the work efficiency. Considering that Y86 language is similar to assembly language, Yan Binghao studied Y86 language and assembly language to ensure the smooth completion of PartA, at the same time, PartB part is similar with that of the experiment on the computer system structure, Yang Hanzhang again review the five stage pipeline design and experiment on computer systems have written the code, PartC most difficult part, involves the CSAPP optimization method of the fifth chapter of "loop unrolling method" and "load using risk" in the fourth chapter, both of us have carefully read the two chapters, the content of Part C was completed in cooperation, so the experiment was completed successfully.
This experiment allowed us to have a systematic and comprehensive review of the knowledge we had learned before, and at the same time, we learned a twin language of X86, Y86, and at the same time, we strengthened our team cooperation ability and gained a lot.