

# Project 1: Optimizing the Performance of a Pipelined Processor

52

52

Zhang

Cai

@sjtu.edu.cn

@sjtu.edu.cn

April 30, 2023

## 1 Introduction

In this project, we need to solve questions of three parts including Part A, Part B, Part C.

In Part A, to improve our understanding of y86 assembly language and better prepare ourselves for Part C, we should write Y86 programs using y86 assembly language to simulate example.c file. Specifically, we need to write three programs: sum.js, rsum.js, and copy.js. The sum.js program iteratively sums linked list elements in a non-recursive form, rsum.js recursively sums linked list elements, and copy.js copies a source block to a destination block. To solve these problems, we should first learn the y86 assembly language and then read asum.js in the *y86 – code* directory. After that, we can sequentially use appropriate instructions to write sum.js, rsum.js, and copy.js.

In Part B, we are required to extend an instruction named *iaddl* to the ISA by modifying the seq-full.hcl, in order to gain a better understanding of the hardware description language. To achieve this goal, we should first learn the fundamental knowledge about *HCL*. Then we need to carefully read the seq-full.hcl to understand how to add corresponding codes for *iaddl*. After that, we should write out the description of *iaddl* and add the necessary codes to the seq-full.hcl file.

In Part C, we are provided with a file called ncopy.js, which can copy *len* elements from the array *src* to the array *dst*. Our task is to reduce the average CPE (cycles per element) as much as possible, and if we can achieve an average CPE less than 10, we will get full grades for Part C. To reduce the CPE, we should try our best to avoid load/use cases in the code, add *iaddl* instruction, and use loop unrolling strategy, which should be considered carefully.

The arrangement of our group members: student finishes part A, part B. **Student** finishes part C and finally writes the report of this project. **Zhang** Cai

## 2 Experiments

### 2.1 Part A

#### 2.1.1 Analysis

We are required to write Y86 programs to simulate the `example.c` file. Specifically, we need to write three programs: `sum.js`, `rsum.js`, and `copy.js`. The `sum.js` program iteratively sums linked list elements in a non-recursive form, `rsum.js` recursively sums linked list elements, and `copy.js` copies a source block to a destination block. Moreover, in `sum.js` and `rsum.js`, we should return the result of `0xcba` in register `%eax`, while in `copy.js`, we need to return the checksum (xor value) of all the words copied in the register `%eax`.

To successfully complete this part, we must have a basic understanding of the Y86 assembly language. The difficulties in this part include understanding the basic logic of assembly programs, such as the use of *push* and *pop* operations when calling functions, knowing the method for passing parameters in assembly programs, and being familiar with related instructions and using them correctly. By using *jmp*, various kinds of *mov*, logical instructions, and so on, we can easily solve this part. The core technique here is to understand the meaning of instructions well and to organize our code carefully.

#### 2.1.2 Code

##### `sum.js`

```
# name:
# ID   :
# sum.js
# .pos 0
# init...
# Sample linked list...

Main:    pushl %ebp
         rrmovl %esp, %ebp

         irmovl ele1, %edx
         pushl %edx
         call sum                # sum(*head)
         rrmovl %ebp, %esp
         popl %ebp
         ret

sum:     pushl %ebp
         rrmovl %esp, %ebp
         mrmovl 8(%ebp), %ecx    # %ecx <- head
         xorl %eax, %eax
         andl %ecx, %ecx        # check if %ecx == NULL
         je End
Loop:    mrmovl (%ecx), %esi     # %esi <- *head
         addl %esi, %eax        # %eax = %eax + %esi
         mrmovl 4(%ecx), %ecx   # *head = head->next
         andl %ecx, %ecx
```

```

        jne Loop
End:     rrmovl %ebp, %esp
        popl %ebp
        ret

        .pos 0x100
Stack:

rsum.ys
# name: Zhang Xiangdong
# ID   : 521030910206
# rsum.ys
        .pos 0
# init...
# Sample linked list...

Main:   pushl %ebp
        rrmovl %esp, %ebp
        irmovl ele1, %edx
        pushl %edx
        call rsum                                # rsum(*head)
        rrmovl %ebp, %esp
        popl %ebp
        ret

rsum:   pushl %ebp
        rrmovl %esp, %ebp
        mrmovl 8(%ebp), %ecx                      # %ecx <- head
        xorl %edx, %edx                          # set zero
        andl %ecx, %ecx
        je ZERO

        mrmovl (%ecx), %edx                      # %edx <- *head
        pushl %edx                              # store the value

        pushl %ebp
        rrmovl %esp, %ebp
        mrmovl 4(%ecx), %ecx
        pushl %ecx
        call rsum                                # rsum(head->next)
        rrmovl %ebp, %esp
        popl %ebp

        popl %edx                                # get the previous value
ZERO:   rrmovl %ebp, %esp
        popl %ebp
        addl %edx, %eax                          # add the value to the return
        value, namely %eax
        ret
        .pos 0x200
Stack:

copy.ys
# name:
# ID   :
# copy.ys

```

```

        .pos 0
# init...
# Source block...
# Destination...

Main:    pushl %ebp
        rrmovl %esp, %ebp
        irmovl $3,%edx
        pushl %edx                # push parameters
        irmovl dest, %edx
        pushl %edx
        irmovl src, %edx
        pushl %edx
        call copy_block          # copy_block(*src, *dest, len)
        rrmovl %ebp,%esp
        popl %ebp
        ret

copy_block:    pushl %ebp
        rrmovl %esp, %ebp

        mrmovl 8(%ebp), %esi      # %esi = src
        mrmovl 12(%ebp), %edi     # %edi = dst
        mrmovl 16(%ebp), %ecx     # %ecx = len
        xorl %eax, %eax           # store the return value
        andl %ecx, %ecx
        je ZERO
Loop:    mrmovl (%esi), %ebx       # %ebx <- *(src)
        irmovl $4, %edx
        addl %edx, %esi           # src++

        rmmovl %ebx, (%edi)       # *(dst) <- %ebx
        addl %edx, %edi
        xorl %ebx, %eax           # %eax <- %ebx ^ %eax
        irmovl $1, %edx
        subl %edx, %ecx           # len--
        jne Loop

ZERO:    rrmovl %ebp, %esp
        popl %ebp
        ret

        .pos 0x400
Stack:

```

### 2.1.3 Evaluation

```

ahapbeangubuntu:~/Desktop/project1-handout-2023/sln/y86-cod
e$ ./misc/yls sum.yo
Stopped in 37 steps at PC = 0x11. Status 'HLT', CC Z=1 S=0
O=0
Changes to registers:
%eax: 0x00000000 0x00000c00
%edx: 0x00000000 0x0000001c
%esp: 0x00000000 0x00000100
%ebp: 0x00000000 0x00000100
%esi: 0x00000000 0x00000c00

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

```

Figure 1: the running result of sum.yo

We write a Y86 program sum.yo that iteratively sums the elements of a linked list. When testing our program, the graders do not spot any errors in our codes. And we can see that the summation 0xcba is correctly calculated and is stored in the register %eax.

```

ahapbeangubuntu:~/Desktop/project1-handout-2023/sln/y86-cod$ ./misc/yls rsum.yo
Stopped in 82 steps at PC = 0x11. Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax: 0x00000000 0x00000c00
%edx: 0x00000000 0x00000000
%esp: 0x00000000 0x00000100
%ebp: 0x00000000 0x00000100

Changes to memory:
0x00b0: 0x00000000 0x000000bc
0x00b4: 0x00000000 0x0000000e
0x00b8: 0x00000000 0x000000c4
0x00bc: 0x00000000 0x00000000
0x00c0: 0x00000000 0x000000d0
0x00c4: 0x00000000 0x00000000
0x00c8: 0x00000000 0x0000000c
0x00cc: 0x00000000 0x00000024
0x00d0: 0x00000000 0x000000d8
0x00d4: 0x00000000 0x00000000
0x00d8: 0x00000000 0x000000e4
0x00dc: 0x00000000 0x0000005e
0x00e0: 0x00000000 0x0000001c
0x00e4: 0x00000000 0x000000ec
0x00e8: 0x00000000 0x00000000
0x00ec: 0x00000000 0x000000f8
0x00f0: 0x00000000 0x0000003d
0x00f4: 0x00000000 0x00000014
0x00f8: 0x00000000 0x00000100
0x00fc: 0x00000000 0x00000011

```

Figure 2: the running result of sum.yo

We write a Y86 program rsum.yo that recursively sums the elements of a linked list. We use a recursive way to solve this question and we get the right result 0xcba in the register %eax. And when testing our program, the graders do not spot any errors in our codes.

```

ahapbeangubuntu:~/Desktop/project1-handout-2023/sln/y86-cod$ ./misc/yls copy.yo
Stopped in 54 steps at PC = 0x11. Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000 0x00000c00
%edx: 0x00000000 0x00000001
%ebx: 0x00000000 0x00000c00
%esp: 0x00000000 0x00000400
%ebp: 0x00000000 0x00000400
%esi: 0x00000000 0x00000020
%edi: 0x00000000 0x0000002c

Changes to memory:
0x0020: 0x00000111 0x0000000a
0x0024: 0x00000222 0x00000000
0x0028: 0x00000333 0x00000000
0x002c: 0x00000444 0x00000000
0x0030: 0x00000555 0x00000000
0x0034: 0x00000666 0x00000000
0x0038: 0x00000777 0x00000000
0x003c: 0x00000888 0x00000000
0x0040: 0x00000999 0x00000000
0x0044: 0x00000aaa 0x00000000
0x0048: 0x00000bbb 0x00000000
0x004c: 0x00000ccc 0x00000000
0x0050: 0x00000ddd 0x00000000
0x0054: 0x00000eee 0x00000000
0x0058: 0x00000fff 0x00000000
0x005c: 0x00001000 0x00000000
0x0060: 0x00001111 0x00000000

```

Figure 3: the running result of sum.yo

We write a Y86 program copy.yo that copies a block of words from one part

of memory to another area of memory, computing the checksum (Xor) of all the words copied. We successfully get the checksum `0xcba` in `%eax` and we can see the previous `0x111 0x222 0x333` in `dst` array are changed to be `0x00a 0x0b0 0xc00` which showing that we achieving copying the `src` to the `dst`. And when testing our program,the graders do not spot any errors in our codes.

## 2.2 Part B

### 2.2.1 Analysis

We need to extend a instruction to the ISA named `iaddl` by modifying the `seq-full.hcl`.To accomplete this,first we should know what is HCL and how does it work.Then we need to figure out the detailed process of `iaddl`.Finally we need to modify the `seq-full.hcl`. The difficulties here include understanding hcl file, correctly writing the description of `iaddl`.

### 2.2.2 Code

The code that is changed in `seq-full.hcl` is displayed here.

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

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 };

##### Decode 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
];
```

```

##### 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 } : 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 in { IOPL, IIADDL } : ifun;
    1 : ALUADD;
];

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

```

## 2.2.3 Evaluation

Evaluation1: we've written a description of the computations required for the iaddl instruction.

```

ahopbeangbuntu@~/Desktop/project-handout-2023/stn/seq5: ./ssim -t ../y88-code/asum.yo
V86 Processor: ipw-fall-hs1
112 bytes of code read
If: fetched (movl) at 0x0: ra=0x0, rb=0x0, valC = 0x100
If: fetched (movl) at 0x4: ra=0x0, rb=0x0, valC = 0x100
If: fetched (jg) at 0x8: ra=0x0, rb=0x0, valC = 0x2
If: fetched (movl) at 0x12: ra=0x0, rb=0x0, valC = 0x4

```

(a) evaluation2.1

```

38 instructions executed
Status = HLT
Condition Codes: Z=1 S=0 O=0
Changed Register State:
%eax: 0x00000000 0x0000abcd
%ecx: 0x00000000 0x00000024
%esp: 0x00000000 0x000000f8
%ebp: 0x00000000 0x00000100
%esi: 0x00000000 0x0000a000
Changed Memory State:
0x00f0: 0x00000000 0x00000100
0x00f4: 0x00000000 0x00000039
0x00f8: 0x00000000 0x00000014
0x00fc: 0x00000000 0x00000004
ISA Check Succeeded

```

(b) evaluation2.2

```

ahopbeangbuntu@~/Desktop/project-handout-2023/stn/y88-code$ make testssim
../seq/ssim -t asum.yo > asum.seq
../seq/ssim -t asum.yo > asum.seq
../seq/ssim -t cjr.yo > cjr.seq
../seq/ssim -t j-cc.yo > j-cc.seq
../seq/ssim -t poptest.yo > poptest.seq
../seq/ssim -t pushquestion.yo > pushquestion.seq
../seq/ssim -t pushtest.yo > pushtest.seq
../seq/ssim -t prog1.yo > prog1.seq
../seq/ssim -t prog2.yo > prog2.seq
../seq/ssim -t prog3.yo > prog3.seq
../seq/ssim -t prog4.yo > prog4.seq
../seq/ssim -t prog5.yo > prog5.seq
../seq/ssim -t prog6.yo > prog6.seq
../seq/ssim -t prog7.yo > prog7.seq
../seq/ssim -t prog8.yo > prog8.seq
../seq/ssim -t ret-hazard.yo > ret-hazard.seq
grep -isn Check -s seq
asum.seq:ISA Check Succeeded
asum.seq:ISA Check Succeeded
cjr.seq:ISA Check Succeeded
j-cc.seq:ISA Check Succeeded
poptest.seq:ISA Check Succeeded
prog1.seq:ISA Check Succeeded
prog2.seq:ISA Check Succeeded
prog3.seq:ISA Check Succeeded
prog4.seq:ISA Check Succeeded
prog5.seq:ISA Check Succeeded
prog6.seq:ISA Check Succeeded
prog7.seq:ISA Check Succeeded
prog8.seq:ISA Check Succeeded
pushquestion.seq:ISA Check Succeeded
pushtest.seq:ISA Check Succeeded
ret-hazard.seq:ISA Check Succeeded

```

(c) evaluation2.3

Figure 4: evaluation2 results

Evaluation2: we pass the benchmark regression tests in y86-code.



Figure 5: evaluation3 results

Evaluation3: we pass the regression tests in ptest for iaddl.

## 2.3 Part C

### 2.3.1 Analysis

In this part, we need to modify the *ncopy.js* and the *pipe - full.hcl* with the goal of making *ncopy.js* run as fast as possible. For evaluation, we run the *./correctness.pl* for checking correctness and run *./benchmark.pl* for getting *CPE* and grades.

The difficulties lie in many aspects including properly modifying the *pipe - full.hcl* file to get better performance while maintaining normal running standard and we need to be familiar with pipeline to find out which cases could bring bubbles to the pipeline and then try our best to change them. **We get 9.09 CPE for the final test.** We complete our work with several steps:

- Try our best to do code scheduling to prevent load/use case that bring bubbles to pipeline.
- Add *iaddl* instruction to the pipeline by modifying the *pipe - full.hcl*.
- Do loop unrolling of 5 cases including 8 4 3 2 1 *times*.
- Take advantage of the two registers *%esi*, *%edi* to improve *CPE* to prevent load/use case To the fullest extent.

### 2.3.2 Code

#### 1. pipe-full.hcl Code (only show the modified part)

```

## instruction fetch
# Is instruction valid?
bool instr_valid = f_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 =
  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 };

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

## execute stage
## 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 } : 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 } : E_valB;
    E_icode in { IRRMOVL, IIRMOVL } : 0;
    # Other instructions don't need ALU
];

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

```

## 2.ncopy.vs Code(only show the core part)

```

    xorl %eax,%eax                # count = 0;
    iaddl $-8, %edx
    jl pre_len4
len8:
    mrmovl (%ebx), %esi           # loop unrolling with len=8
    mrmovl 4(%ebx), %edi          # use two registers to reduce data
    hazard
    rmmovl %esi, (%ecx)
    andl %esi, %esi
    rmmovl %edi, 4(%ecx)

```

```

        jle len8_1
        iaddl $1, %eax                # count++
len8_1:
        andl %edi, %edi
        jle len8_2
        iaddl $1, %eax

len4:
        mrmovl (%ebx), %esi          # loop unrolling with len = 4
        mrmovl 4(%ebx), %edi
        rmmovl %esi, (%ecx)
        andl %esi, %esi
        rmmovl %edi, 4(%ecx)
        jle len4_1
        iaddl $1, %eax                # count++
len4_1:
        andl %edi, %edi
        jle len4_2
        iaddl $1, %eax

        # loop unrolling with len = 3, len = 2, len = 1 ...

```

### 2.3.3 Evaluation



Figure 6: evaluation1 results

Evaluation1: We successfully pass the sdriver and the ldriver test.

```

ahapbeangubuntu:~/Desktop/project1-handout-2023/sin/pipe$ ./misc/yis sdr1
ver-yo
Stopped in 51 steps at PC = 0x29. Status 'HLT', CC Z=0 S=1 O=0
Changes to registers:

```

Figure 7: evaluation2 results

Evaluation2: Our function ncopy works properly with YIS.

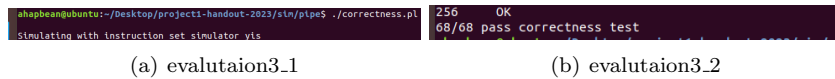


Figure 8: evaluation3 results

Evaluation3: Our code run correctly with correctness.pl.

```

ahapbeangubuntu:~/Desktop/project1-handout-2023/sin/y86-code$ make testps1
rm
./pipe/ps1m -t asun.yo > asun.pipe
./pipe/ps1m -t asunr.yo > asunr.pipe
./pipe/ps1m -t cjr.yo > cjr.pipe
./pipe/ps1m -t j-cc.yo > j-cc.pipe
./pipe/ps1m -t popentest.yo > popentest.pipe
./pipe/ps1m -t pushquestion.yo > pushquestion.pipe
./pipe/ps1m -t prog1.yo > prog1.pipe
./pipe/ps1m -t prog2.yo > prog2.pipe
./pipe/ps1m -t prog3.yo > prog3.pipe
./pipe/ps1m -t prog4.yo > prog4.pipe
./pipe/ps1m -t prog5.yo > prog5.pipe
./pipe/ps1m -t prog6.yo > prog6.pipe
./pipe/ps1m -t prog7.yo > prog7.pipe
./pipe/ps1m -t prog8.yo > prog8.pipe
./pipe/ps1m -t ret-hazard.yo > ret-hazard.pipe
diff -ISA Check Succeeded
asun.pipe:ISA Check Succeeded
asunr.pipe:ISA Check Succeeded
cjr.pipe:ISA Check Succeeded
j-cc.pipe:ISA Check Succeeded
popentest.pipe:ISA Check Succeeded
prog1.pipe:ISA Check Succeeded
prog2.pipe:ISA Check Succeeded
prog3.pipe:ISA Check Succeeded
prog4.pipe:ISA Check Succeeded
prog5.pipe:ISA Check Succeeded
prog6.pipe:ISA Check Succeeded
prog7.pipe:ISA Check Succeeded
prog8.pipe:ISA Check Succeeded
pushquestion.pipe:ISA Check Succeeded
ret-hazard.pipe:ISA Check Succeeded
rm asun.pipe asunr.pipe cjr.pipe j-cc.pipe popentest.pipe pushquestion.pipe
pushquestion.pipe prog1.pipe prog2.pipe prog3.pipe prog4.pipe prog5.pipe prog6
.pipe prog7.pipe prog8.pipe ret-hazard.pipe

```

Figure 9: evaluation4 results

Evaluation4: We test our simulator against the Y86 benchmark programs and pass all test.

```

ahapbeangubuntu:~/Desktop/project1-handout-2023/sin/ptest$ make Sinw.../ptp
e/ps1m
./ptest.pl -s ../pipe/ps1m
Simulating with ../pipe/ps1m
All 49 ISA Checks Succeed
./itest.pl -s ../pipe/ps1m
Simulating with ../pipe/ps1m
All 64 ISA Checks Succeed
./ctest.pl -s ../pipe/ps1m
Simulating with ../pipe/ps1m
All 22 ISA Checks Succeed
./itest.pl -s ../pipe/ps1m
Simulating with ../pipe/ps1m
All 600 ISA Checks Succeed

```

```

ahapbeangubuntu:~/Desktop/project1-handout-2023/sin/ptest$ make Sinw.../ptp
e/ps1m TFLAGS=-l
./ptest.pl -s ../pipe/ps1m -l
Simulating with ../pipe/ps1m
All 58 ISA Checks Succeed
./itest.pl -s ../pipe/ps1m -l
Simulating with ../pipe/ps1m
All 90 ISA Checks Succeed
./ctest.pl -s ../pipe/ps1m -l
Simulating with ../pipe/ps1m
All 22 ISA Checks Succeed
./itest.pl -s ../pipe/ps1m -l
Simulating with ../pipe/ps1m
All 756 ISA Checks Succeed

```

(a) evalutaion5.1

(b) evalutaion5.2

Figure 10: evaluation5 results

Evalutaion5: Our pipeline simulator pass ../ptest (-i).

```

ahapbeangubuntu:~/Desktop/project1-handout-2023/sin/pipe$ ./correctness.pl
-p
Simulating with pipeline simulator ps1m

```

```

256 OK
68/68 pass correctness test

```

(a) evalutaion6.1

(b) evalutaion6.2

Figure 11: evaluation6 results

Evaluation6: We pass ./correctness.pl -p.

```

ahapbeangubuntu:~/Desktop/project1-handout-2023/sin/pipe$ ./benchmark.pl
ncopy

```

```

Average CPE    9.09
Score  60.0/60.0

```

(a) evalutaion7.1

(b) evalutaion7.2

Figure 12: evaluation7 results

Evaluation7: Our code get 9.09 CPE by running the benchmark.pl.  
Evaluation8: Our codes are 602 bytes long namely less than 1000 bytes.

## 3 Conclusion

### 3.1 Problems

- Question: Do not know how the *.hcl* file work when doing part B. We solve this by reading corresponding chapters in CSAPP and search some internet information. Finally, we learn this well and solve part B successfully.
- Question: After loop unrolling, our CPE only reduce to 11.49 but all strategies seem to have been applied. After thinking for a relatively long time, we find that we do loop unrolling in a bad manner because we add the pointer with 4 each time. Actually, most of them can be deleted by knowing the concrete offset value. Through this improvement, we improve our codes and reduce the CPE to less than 10.

### 3.2 Achievements

- We code our part A code in a relatively clear manner because we add comments in detail in our codes.
- We make the *iaddl* run correctly in part B.
- After many times of improvement to the *ncopy.js* and the *pipe-full.hcl*, we make our CPE to 9.92 in part C.
- We think actively and try our best to improve our code. Finally we reduce the CPE to 9.09 which turns out to be a great breakthrough for us.