Project 1: Optimizing the Performance of a Pipelined Processor

000, Dou Yi-Ming, douyiming@sjtu.edu.cn
001, Fang Tian-Cheng, fangtiancheng@sjtu.edu.cn

May 10, 2021

1 Introduction

In this project, we completed three tasks step by step.

Firstly, we learned and wrote three Y86 programs, which respectively simulates the behaviour of three examples written in C. The first two examples sum the linked list elements, with iterative and recursive methods, while the third example copy a source block to the destination block. We begin with learning the fundamental grammar of Y86, which is somewhat similar to MIPS, but the rule of Y86 is more complex. Then, we read the C program of the examples to understand their function and translate them into assembly language Y86.

Secondly, we are required to make the processor support new instructions: **iaddl** and **leave**. The compiler is already able to compile the two instructions, meaning that all we need to do is modifying the circuit described by the HCL file. Therefore, we added new connections into the HCL file, making the compiler able to recognize and compile the new instructions.

Eventually, having been prepared by completing Part A and Part B, we came to the heart of the lab: optimization of the Y86 benchmark program and processor design. The function of the benchmark is simply copying a block of integers to the destination block, while counting the sum of the positive numbers. We begin with a intuitive optimization: replacing irmovq and addq with iaddq, this also requires modifying the HCL file pipe-full.hcl, similar to what we have done in Part B. Then, we refer to the CS:APP2e textbook and implemented the **jump table**, which is an elegant and efficient optimization into our assembly program, which significantly reduced the CPE of the program.

Fang Tian-Cheng finished part A and B, Dou Yi-Ming finished part C and the report.p

2 Experiments

2.1 Part A

2.1.1 Analysis

This part is fundamental and essential to the whole project. The analysis of the task includes the detailed understanding of the difficulties:

1. The main difficulty of this part is to understand Y86 CPU structure and Y86 assembly syntax. After consulting books, asking students, our group is finally closely familiar with it. The Y86 machine contains 8 registers. Some registers have its own different purposes, such as the ESP reg stores the stack top pointer, etc. Unlike MIPS architecture, the code segment and data segments of Y86 CPU are stored in the same memory. So we can easily handle the relationship between data and memory. Y86 assembly language is also rich and interesting. In Y86, data has three storage addresses: register, memory and immediate part of a instruction. We use 'r', 'm' and 'i' to represent these three places respectively. For 32-bit and 64-bit data lengths, we use 'l' and 'q' to represent them respectively. For example, if we want to move a 32-bit immediate from instruction to register, we can use "irmovl imm, reg" to tell the CPU. Another difference from MIPS instruction is that the destination of Y86 assembly language is the next location rather than the first.

2. Another difficulty is the coordination of instructions. For example, if I want to implement an "if" statement to determine whether a register is zero. Then I can use 'andl' and 'je' to execute. Because there is a flag register in Y86 CPU, in which there is a ZF bit to indicate whether the result of the last instruction is zero or not. If it is zero, ZF is set to high level, and je jumps to realize the branch of instruction.

3. The last difficulty is the use of stack in Y86. First, we mark the bottom of the stack at the end of the program, and then assign the address to the ESP reg. Y86 stack is characterized by upward growth. I guess this may be to protect the growth of this program stack from damaging other programs before destroying its own programs. Every time I push a 32-bit data to the stack, the ESP register first subtracts 4, and then writes the data to the position pointed by ESP. Similarly, whenever I pop a 32-bit data from the stack, the CPU first reads 4 bytes from the position indicated by ESP, and then adds 4 to ESP. When implementing a recursive function, if there is data that exists before the recursive call and still needs to be used after the end of the recursion, then we should send it into the stack for protection.

2.1.2 Code

[In this part, you should place your code and make it readable in Microsoft Word, please. Writing necessary comments for codes is a good habit.] The code of sum.ys, rsum.ys and cpoy.ys is demonstrated with detailed comments:

sum.ys:

```
.pos    0x0
init:
    irmovl  stack, %esp
    call    main
    halt

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

main:
    irmovl  ele1,   %edi        # move the head addr of the list to %edi
    irmovl  $0,     %eax        # %rax store the sum of list, 0 init
    call    sumFunc
    ret

sumFunc:
    andl    %edi,   %edi        # while(%edi != 0){
    je      sumFuncEnd          #       else return
    mrmovl  0(%edi),    %ebp    #       move data from memory[0+%edi] to %ebp
    addl    %ebp,    %eax       #       %eax += data
    mrmovl  4(%edi),    %edi    #       %edi = %edi -> next
    jmp     sumFunc             # }

sumFuncEnd:
    ret                         # return

    .pos    0x200
stack:
```

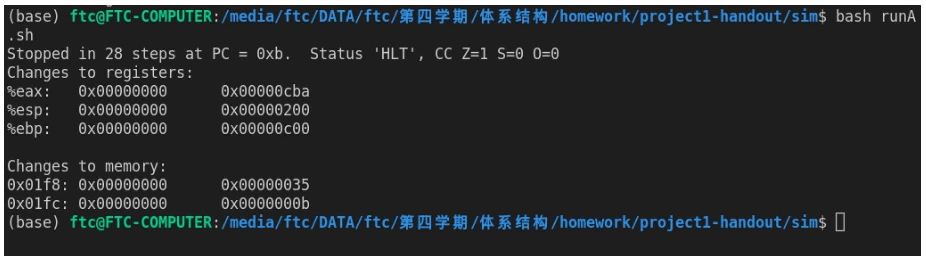
rsum.ys:

```
.pos    0x0
```

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

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

main:
    irmovl  ele1,   %edi        # move the head addr of the list to %edi
    irmovl  $4,     %esi        # set %esi = const int 4
    call    rsumFunc
    ret

rsumFunc:                       # return value save in %eax, ptr save in %edi
    andl    %edi,   %edi        #   if(%edi == 0)
    je      rsumFuncEnd         #       return 0
    pushl   %edi,               #   push %edi to stack
    mrmovl  4(%edi),    %edi    #   %edi = %edi -> next
    call    rsumFunc            #
    popl    %edi                #   fetch data = memory[%ecx]
    mrmovl  0(%edi), %ebp
    addl    %ebp,   %eax        #   %eax += data
    ret

rsumFuncEnd:                    # return 0
    irmovl  $0,     %eax        # set return value = 0
    ret                         # return

    .pos    0x200
stack:
```

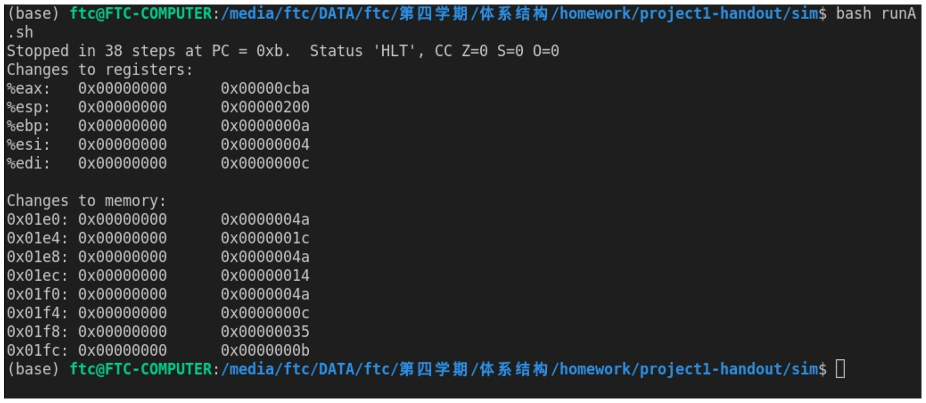
copy.ys:

```
.pos    0x0
init:
    irmovl  stack, %esp
    call    main
    halt

    .align  4
src:
    .long 0x00a
    .long 0x0b0
    .long 0xc00

dest:
    .long 0x111
    .long 0x222
    .long 0x333
# eax: return value
# ebx: src
# ecx: dst
# edx: len

# esp: stack
main:
    irmovl  src,    %ebx        # move src addr to %ebx
    irmovl  dest,   %ecx        # move dst addr to %ecx
    irmovl  $3,     %edx        # set len = 3
    irmovl  $4,     %esi        # set %esi = 4
    irmovl  $1,     %ebp        # set %ebp = 1
    irmovl  $0,     %eax        # set return value = 0
    call    copyFunc
    ret

copyFunc:                       # return value save in %eax
    andl    %edx,   %edx        #   if(%edx == 0)
    je      copyFuncEnd         #       return 0
    mrmovl  0(%ebx),    %edi    #   %edi = [%ebx]
    addl    %esi,   %ebx        #   %ebx += 4
    rmmovl  %edi,   0(%ecx)     #   [%ecx] = %edi
    addl    %esi,   %ecx        #   %ecx += 4
    xorl    %edi,   %eax        #   %eax ^= %edi
    subl    %ebp,   %edx        #   %edx -= 1
    jmp     copyFunc

copyFuncEnd:
    ret                         # return

    .pos    0x200
stack:
```



```
(base) ftc@FTC-COMPUTER:/media/ftc/DATA/ftc/第四学期/体系结构/homework/project1-handout/sim$ bash runA
.sh
Stopped in 28 steps at PC = 0xb.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%esp:   0x00000000      0x00000200
%ebp:   0x00000000      0x00000c00

Changes to memory:
0x01f8: 0x00000000      0x00000035
0x01fc: 0x00000000      0x0000000b
(base) ftc@FTC-COMPUTER:/media/ftc/DATA/ftc/第四学期/体系结构/homework/project1-handout/sim$ 
```

Figure 1: Result of sum.ys



```
(base) ftc@FTC-COMPUTER:/media/ftc/DATA/ftc/第四学期/体系结构/homework/project1-handout/sim$ bash runA
.sh
Stopped in 38 steps at PC = 0xb.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%esp:   0x00000000      0x00000200
%ebp:   0x00000000      0x0000000a
%esi:   0x00000000      0x00000004
%edi:   0x00000000      0x0000000c

Changes to memory:
0x01e0: 0x00000000      0x0000004a
0x01e4: 0x00000000      0x0000001c
0x01e8: 0x00000000      0x0000004a
0x01ec: 0x00000000      0x00000014
0x01f0: 0x00000000      0x0000004a
0x01f4: 0x00000000      0x0000000c
0x01f8: 0x00000000      0x00000035
0x01fc: 0x00000000      0x0000000b
(base) ftc@FTC-COMPUTER:/media/ftc/DATA/ftc/第四学期/体系结构/homework/project1-handout/sim$ 
```

Figure 2: Result of rsum.ys

2.1.3 Evaluation

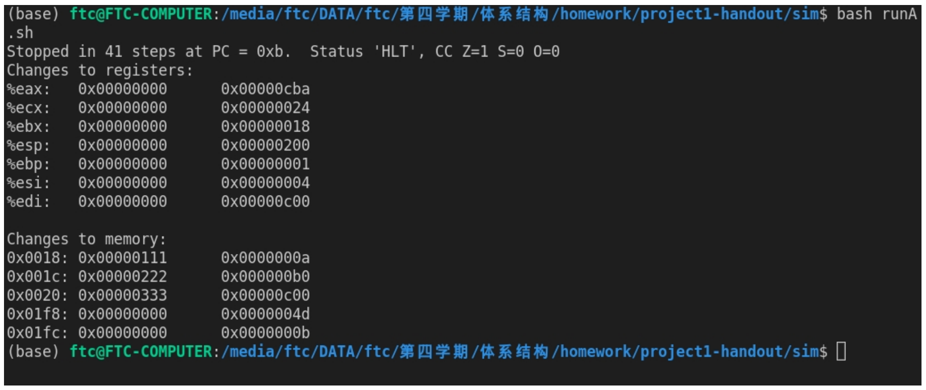
The results of the code above is shown respectively in Fig.1, Fig.5 and Fig.3.

2.2 Part B

2.2.1 Analysis

The main difficulties:

- The main problem we are facing is to understand the "seq-full.hcl" file. Fortunately, the comments in this file are detailed enough. We only need to read the comments to know what the code is doing. Such as "REBP" in line 49, it represents register EBP and stores frame pointer, which can be easily known from the comments below.
- Secondly, we need to get familiar with the function of "IIADDL" and "ILEAVE". For "IIADDL", it has the formula like "iaddl imm, reg", which adds an immediate to a register. For "ILEAVE", it has the same effect as "rrmov esp, ebp" and "popl ebp". After we have figured out the functions of the two instructions, we can start to implement them.



```
(base) ftc@FTC-COMPUTER:/media/ftc/DATA/ftc/第四学期/体系结构/homework/project1-handout/sim$ bash runA
.sh
Stopped in 41 steps at PC = 0xb.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%ecx:   0x00000000      0x00000024
%ebx:   0x00000000      0x00000018
%esp:   0x00000000      0x00000200
%ebp:   0x00000000      0x00000001
%esi:   0x00000000      0x00000004
%edi:   0x00000000      0x00000c00

Changes to memory:
0x0018: 0x00000111      0x0000000a
0x001c: 0x00000222      0x000000b0
0x0020: 0x00000333      0x00000c00
0x01f8: 0x00000000      0x0000004d
0x01fc: 0x00000000      0x0000000b
(base) ftc@FTC-COMPUTER:/media/ftc/DATA/ftc/第四学期/体系结构/homework/project1-handout/sim$
```

Figure 3: Result of copy.ys

- Thirdly, we also need to figure out what the instructions do at each stage. Take “IIADDL” as an example. It is valid in IF stage, so it should be put in “instr-valid” set in line 108. In the same way, it contains an immediate number, so it needs to store a constant word when fetching the instruction. So we must put “IIADDL” in set “need-valC” in line 118. All in all, we need to read every line of code in the HCl file carefully to decide whether to add instructions to it.

2.2.2 Code

The code of seq-full.hcl is shown in the following part.

```
################ 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, ILEAVE } : 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, 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 } : valB;
icode in { IRRMOVL, IIRMOVL } : 0;
icode in { ILEAVE } : valA;
  # 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;
];
#/* $end seq-all-hcl */
```

2.2.3 Evaluation

The result of asumi is shown in Fig

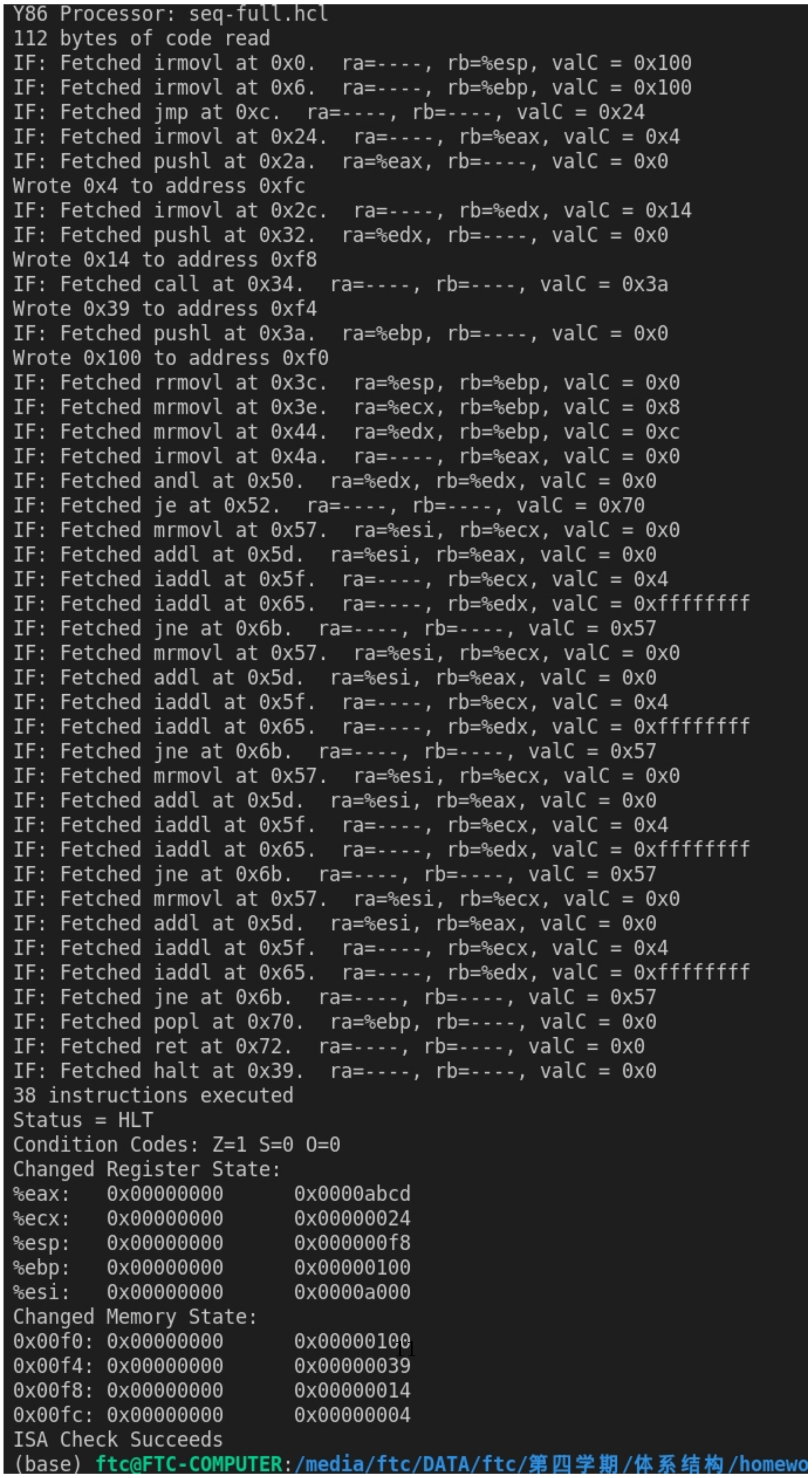
2.3 Part C

2.3.1 Analysis

In this part, we need to optimize the performance of the benchmark program by modifying the code and the processor design. The analysis of the task includes the main difficulties and core technique.

1. Main difficulties:

    - Although the function of the benchmark program is simply copying a group of integers into another, merely designing the Y86 program intuitively may make the program execute with low efficiency. In this example, there is a loop in the C program that will be executed for 8 times, and for each iteration, the CPU need to add 4 to both source and destination address, which costs a lot, thus we consider whether we can reduce or even eliminate the adding procedure. However,



```
Y86 Processor: seq-full.hcl
112 bytes of code read
IF: Fetched irmovl at 0x0.  ra=----, rb=%esp, valC = 0x100
IF: Fetched irmovl at 0x6.  ra=----, rb=%ebp, valC = 0x100
IF: Fetched jmp at 0xc.  ra=----, rb=----, valC = 0x24
IF: Fetched irmovl at 0x24.  ra=----, rb=%eax, valC = 0x4
IF: Fetched pushl at 0x2a.  ra=%eax, rb=----, valC = 0x0
Wrote 0x4 to address 0xfc
IF: Fetched irmovl at 0x2c.  ra=----, rb=%edx, valC = 0x14
IF: Fetched pushl at 0x32.  ra=%edx, rb=----, valC = 0x0
Wrote 0x14 to address 0xf8
IF: Fetched call at 0x34.  ra=----, rb=----, valC = 0x3a
Wrote 0x39 to address 0xf4
IF: Fetched pushl at 0x3a.  ra=%ebp, rb=----, valC = 0x0
Wrote 0x100 to address 0xf0
IF: Fetched rrmovl at 0x3c.  ra=%esp, rb=%ebp, valC = 0x0
IF: Fetched mrmovl at 0x3e.  ra=%ecx, rb=%ebp, valC = 0x8
IF: Fetched mrmovl at 0x44.  ra=%edx, rb=%ebp, valC = 0xc
IF: Fetched irmovl at 0x4a.  ra=----, rb=%eax, valC = 0x0
IF: Fetched andl at 0x50.  ra=%edx, rb=%edx, valC = 0x0
IF: Fetched je at 0x52.  ra=----, rb=----, valC = 0x70
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched popl at 0x70.  ra=%ebp, rb=----, valC = 0x0
IF: Fetched ret at 0x72.  ra=----, rb=----, valC = 0x0
IF: Fetched halt at 0x39.  ra=----, rb=----, valC = 0x0
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 Succeeds
(base) ftc@FTC-COMPUTER:/media/ftc/DATA/ftc/第四学期/体系结构/homewo
```

Figure 4: Result of asumi.ys

optimizing this may need us to think beyond our intuition, which may be difficult.

- Furthermore, one relatively simple way to reduce the execution time is that we can write an efficient program by minimizing the repetitive operations such as adding 4 to the address in each iteration, with the sacrifice of space. Nevertheless, the size of our ncopy file is limited to 1000 bytes, meaning that we should balance the cost of time and space, which cause some trouble.

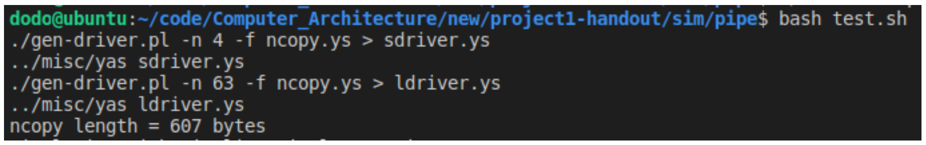
2. Core technique:

   - Firstly, we found that all of the adding operations in the origin code execute two instructions: irmovl and addl, which is definitely unnecessary since we have already defined iaddl. Therefore, we simply replace each pair of irmovl and addl with the instruction iaddl. Moreover, to make the processor recognize the instruction iaddl, we also modified the file pipe-full.hcl, the procedure is similar to Part B.
   - Secondly, as is mentioned above, during the loop, the address of source and destination is repeatedly being added with 4, so we consider whether we can reduce this repetitive operation. After referring CS:APP2e textbook, we found that there is a method called jump table, the principle of which is directly add the number to the address in the code instead of updating the address each time, and when k integers have been copied, the address is updated at one time ($address = address + 4 \times k$), thus the loop is unrolled. This method significantly reduce the amount of the repetitive updating operation by about 90% when $k = 10$.
   - Eventually, though we have reduced the CPE to less than 10 by applying the two strategies described above, we are actually able to reduce the CPE even more, by applying a larger k, such as 16. However, the cost of space is larger if we apply a larger k. We take the balance of time and space into consideration, thus implementing the jump table with $k = 10$ and the length of the ncopy file is only 607 bytes.

2.3.2 Code

The code of ncopy.ys is demonstrated with necessary comments:

```
#/* $begin ncopy-ys */
##################################################################
# ncopy.ys - Copy a src block of len ints to dst.
# Return the number of positive ints (>0) contained in src.
#
# Include your name and ID here.
# Dou Yiming  519021910366
# Fang Tiancheng  519021910173
#
# Describe how and why you modified the baseline code.
# 1. replace irmovq & addq with iaddq
# 2. implement 10-way loop unrolling
# 3. use jump table to handle elements left
#
##################################################################
# Do not modify this portion
# Function prologue.
ncopy:
    pushl %ebp # Save old frame pointer
    rrmovl %esp,%ebp # Set up new frame pointer
    pushl %esi # Save callee-save regs
    pushl %ebx
    pushl %edi
    mrmovl 8(%ebp),%ebx # src
    mrmovl 16(%ebp),%edx # len
    mrmovl 12(%ebp),%ecx # dst

##################################################################
# You can modify this portion
    # Loop header
    xorl %eax,%eax # count = 0;
    iaddl $-10, %edx
    jl handle_left
    # andl %edx,%edx # len <= 0?
    # jle Done # if so, goto Done:

Loop:
    mrmovl (%ebx), %esi # read val from src...
    rmmovl %esi, (%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle Npos1
    iaddl $1, %eax # count++
    #jle Npos # if so, goto Npos:

Npos1:
    mrmovl 4(%ebx), %esi # read val from src...
    rmmovl %esi, 4(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle Npos2
    iaddl $1, %eax # count++

Npos2:
    mrmovl 8(%ebx), %esi # read val from src...
    rmmovl %esi, 8(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle Npos3
    iaddl $1, %eax # count++

Npos3:
    mrmovl 12(%ebx), %esi # read val from src...
    rmmovl %esi, 12(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle Npos4
    iaddl $1, %eax # count++

Npos4:
    mrmovl 16(%ebx), %esi # read val from src...
    rmmovl %esi, 16(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle Npos5
    iaddl $1, %eax # count++

Npos5:
    mrmovl 20(%ebx), %esi # read val from src...
    rmmovl %esi, 20(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle Npos6
    iaddl $1, %eax # count++

Npos6:
    mrmovl 24(%ebx), %esi # read val from src...
    rmmovl %esi, 24(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle Npos7
    iaddl $1, %eax # count++

Npos7:
    mrmovl 28(%ebx), %esi # read val from src...
    rmmovl %esi, 28(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle Npos8
    iaddl $1, %eax # count++

Npos8:
    mrmovl 32(%ebx), %esi # read val from src...
    rmmovl %esi, 32(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle Npos9
    iaddl $1, %eax # count++
Npos9:
    mrmovl 36(%ebx), %esi # read val from src...
    rmmovl %esi, 36(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle Npos
    iaddl $1, %eax # count++

Npos:
    iaddl $40, %ebx # src+=10
    iaddl $40, %ecx # dst+=10
    iaddl $-10, %edx # len-=10
    jge Loop

handle_left:
    addl %edx, %edx
    addl %edx, %edx
    mrmovl Table(%edx), %esi
    pushl %esi
    ret

left9:
    mrmovl 32(%ebx), %esi # read val from src...
    rmmovl %esi, 32(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle left8
    iaddl $1, %eax # count++

left8:
    mrmovl 28(%ebx), %esi # read val from src...
    rmmovl %esi, 28(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle left7
    iaddl $1, %eax # count++

left7:
    mrmovl 24(%ebx), %esi # read val from src...
    rmmovl %esi, 24(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle left6
    iaddl $1, %eax # count++

left6:
    mrmovl 20(%ebx), %esi # read val from src...
    rmmovl %esi, 20(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle left5
    iaddl $1, %eax # count++

left5:
    mrmovl 16(%ebx), %esi # read val from src...
    rmmovl %esi, 16(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle left4
    iaddl $1, %eax # count++

left4:
    mrmovl 12(%ebx), %esi # read val from src...
    rmmovl %esi, 12(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle left3
    iaddl $1, %eax # count++

left3:
    mrmovl 8(%ebx), %esi # read val from src...
    rmmovl %esi, 8(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle left2
    iaddl $1, %eax # count++

left2:
    mrmovl 4(%ebx), %esi # read val from src...
    rmmovl %esi, 4(%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle left1
    iaddl $1, %eax        # count++

left1:
    mrmovl (%ebx), %esi # read val from src...
    rmmovl %esi, (%ecx) # ...and store it to dst
    andl %esi, %esi # val <= 0?
    jle Dones
    iaddl $1, %eax        # count++
    #ret

Dones:
    leave
    ret

.align 4
        .long Done
.long left1
```



```
dodo@ubuntu:~/code/Computer_Architecture/new/project1-handout/sim/pipe$ bash test.sh
./gen-driver.pl -n 4 -f ncopy.ys > sdriver.ys
../misc/yas sdriver.ys
./gen-driver.pl -n 63 -f ncopy.ys > ldriver.ys
../misc/yas ldriver.ys
ncopy length = 607 bytes
```

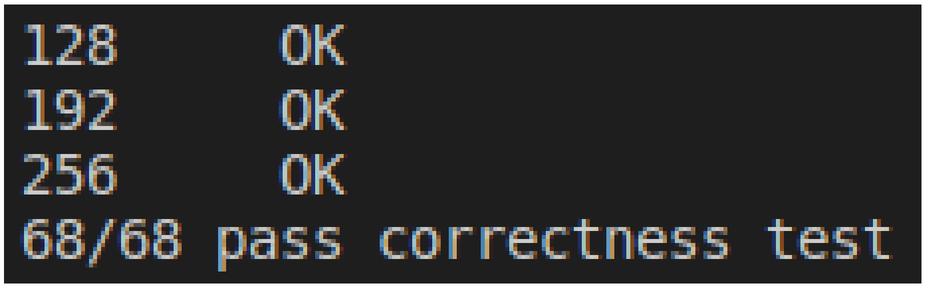
Figure 5: Length of ncopy

```
        .long left2
        .long left3
        .long left4
        .long left5
        .long left6
        .long left7
        .long left8
        .long left9
Table:

##################################################################
# Do not modify the following section of code
# Function epilogue.
Done:
popl %edi               # Restore callee-save registers
popl %ebx
popl %esi
rrmovl %ebp, %esp
popl %ebp
ret
##################################################################
# Keep the following label at the end of your function
End:
#/* $end ncopy-ys */
```

2.3.3 Evaluation

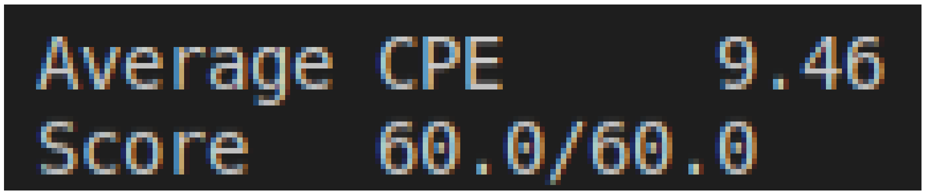
The result of the correctness test and the benchmark test in shown in the following figures.

The length of ncopy is 607 bytes. All of the correctness tests are passed. The average CPE is 9.46.



```
128     OK
192     OK
256     OK
68/68 pass correctness test
```

Figure 6: Length of ncopy



```
Average CPE     9.46
Score   60.0/60.0
```

Figure 7: Length of ncopy

3 Conclusion

3.1 Problems

1. When we first meet the Y86 assembly language, we was a little bit scared because of the strangeness, but after referring to a large number of textbooks and materials on the Internet, we learned that in fact, this assembly language is not very different from the MIPS language we learned in class. So we quickly grasped the grammar of Y86 and began to try to write simple programs.

2. In MIPS, the data segment in memory and the code segment of assembly language are separated, while in Y86, the two are integrated. At the same time, we need to specify the starting position of the stack in Y86. These differences make us feel uncomfortable at first, but after getting familiar with this new idea, we find that Y86 provides a more flexible operation mode than MIPS to a certain extent.

3. In the last task, we need to modify the assembly language code to improve efficiency on the basis of ensuring that the functions are consistent with the original. Because the optimization method of assembly language is often counter-intuitive, this is a big challenge for us. However, after referring to the CSAPP textbook, we got a lot of inspiration and optimized the assembly code perfectly.

3.2 Achievements

1. In terms of code readability, we have written comments as detailed as possible in each file, which greatly improves the legibility of our code. At the same time, we try our best to write our code in a standardized and professional mode. On the one hand, it makes the teacher more comfortable when checking, and on the other hand, it greatly reduces the time and effort we need to debug the program.

2. When it comes to performance, we optimized the CPE to 9.46 through the instruction replacement and loop unrolling methods mentioned above, which is a perfect result that can get full marks. We believe that the process of optimizing the assembler is the most rewarding stage when completing this project.

3. In addition, it should be particularly emphasized that in the process of cooperating to complete such a project, we have learned a lot of life-long skills, including fast data retrieval ability, code debugging ability, and the courage to solve an unknown problem. , These gains go far beyond the content of the project itself.