Project 1: Optimizing the Performance of a Pipelined Processor

Li Zikun, 521120910252, lizk1024@sjtu.edu.cn
Hu Chenzhi, 521021910107, ether-wind@sjtu.edu.cn
Wang Shunyu, 521120910244, lzywsy03@sjtu.edu.cn

April 29, 2023

1 Introduction

In this project, we have three parts. In part A, we wrote some simple Y86 programs, for example, sum, rsum and copy , to get familiar with the usage of Y86. In part B, we made a little modification to the SEQ processor for a new instruction: `iaddl`. In part C, which is the critical and the hardest part of the whole lab, we optimized the performance of the pipelined processor with the ncopy function.

[Li and Wang finished part A and B together, Hu finished part C, and the three of us finished the report together.]

2 Experiments

2.1 Part A

2.1.1 Analysis

In part A, we were tasked with using `Y86 ISA` to implement `C` language code functionality. Our objective was to use Y86 assembly language to achieve the same functionality as the code in `example.c`, and we accomplished two tasks: 1) traverse a linked list and return the sum of all long-type integers in the list (using both iterative and recursive algorithms), and 2) copy a sequence of long-type integers to another memory address.

To achieve this, we first studied Y86 register names, instruction format, and execution method. The core techniques for each of the three different functionalities are as follows:

1. For the `sum` function, we traversed the linked list using a loop to sum all `long-type` integers. We did this by loading and storing memory data, iterating through each element of the list, and incrementally summing the values of the

long integers in the list with the help of registers. We used the \%eax register to record current element value, \%edx register to record the address of next element, and \%ecx register to record the sum of all elements. If there are more elements after the current one, we jump back to the loop to continue processing. If the end of the linked list is reached, the program returns the sum stored in the \%ecx register. A stack is also used to ensure proper program execution.

2. For the \%rsum function, we used recursion to sum long-type integers in a linked list. We loaded and stored memory data in the same way as in the "sum" function, but instead of a loop, we relied on recursive calls to the rsum_list function to process each element in the list. Each time an element is processed, its value is added to the \%eax register, and the address of the next element and the current element's value are stored in the \%edx and\%edi registers respectively. If \%edx is equal to zero, the recursion ends, and the result stored in the \%eax register is returned. A stack is also used to ensure proper program execution.

3. For the copy_block function, the code copies the data from a source address src to a destination address dest by looping through the data in the source address and copying it to the corresponding location in the destination address. The main function in the code calls the copy_block function to perform the data copy. A stack is used to store the original information saved in registers before function invocation by using the pushl, and is restored using the popl instruction at the end of the function execution.

The challenges in implementing these functionalities included understanding how many registers we could use in our code, learning various Y86 ISA opcodes and instruction formats, as well as understanding and using registers to store and control data flow, controlling program flow by using various jump instructions, mastering stack operations, and writing complex algorithms to traverse a linked list and copy a block of memory. It was also important to have proficient debugging and error fixing skills.

2.1.2 Code

The core code for three tasks in Part A is respectively in Code Listing 1, 2, 3.

```
1 .align 4        # Sample linked list
2 ele1:       # Define memory with label ele1
3         .long 0x00a
4         .long ele2
5 ele2:       # Define memory with label ele2
6         .long 0x0b0
7         .long ele3
8 ele3:       # Define memory with label ele3
9         .long 0xc00
10        .long 0
11
12 main:
13   irmovl ele1,%edx    # Initialize %edx with the address of ele1
14         call sum_list     # Call the sum_list function
15   ret
```

```
16
17 sum_list:       # sums a list of long integers starting at the address
       in %edx
18   irmovl $0,%ecx    # Set the initial value of %ecx to 0
19
20 loop: mrmovl (%edx),%eax  # Load memory address at %edx into %eax
21         addl %eax,%ecx    # Add to %ecx
22         mrmovl 4(%edx ),%eax    # Load the next memory address to %eax
23         rrmovl %eax,%edx    # Move the next address to %edx
24
25 test:   andl %edx,%edx    # Check if %edx is equal to 0
26         jne loop      # If not 0 jump back to loop
27         ret       # Return the sum stored in %ecx
```

Code Listing 1: Core Code of sum.ys

```
1 .align 4        # Sample linked list
2 ele1:       # Define memory with label ele1
3         .long 0x00a
4         .long ele2
5 ele2:       # Define memory with label ele2
6         .long 0x0b0
7         .long ele3
8 ele3:       # Define memory with label ele3
9         .long 0xc00
10        .long 0
11
12 main:
13   irmovl ele1,%edx    # Initialize %edx with the address of ele1
14         call rsum_list      # Calls the sum_list function
15   ret
16 rsum_list:
17   pushl %edi    # Push %edi into stack
18   irmovl $0,%eax    # Set the initial value of result register %eax to
         0
19   andl %edx,%edx    # Check if %edx is equal to 0
20   je re     # If 0 jump to re
21   mrmovl (%edx),%edi  # Load memory address at %edx into %edi
22   mrmovl 4(%edx),%edx # Load next memory address of %edx into %edx
23   call rsum_list    # Call the rsum_list function
24   addl %edi,%eax    # Add value of %edi to %eax
25 re:
26   popl %edi   # Pop %edi out of stack
27   ret     # Return the sum stored in %eax
```

Code Listing 2: Core Code of rsum.ys

```
1 .align 4
2 src:        # Source block
3   .long 0x00a
4   .long 0x0b0
5   .long 0xc00
6
7 dest:         # Destination block
8   .long 0x111
```

```
 9   .long 0x222
10   .long 0x333
11
12 main:
13   irmovl src, %esi  # Initialize source address
14   irmovl dest, %edi   # Initialize destination address
15   irmovl $3, %edx   # Initialize count
16   call copy_block   # Call copy_block to perform the copy
17   ret
18
19 copy_block:
20   pushl %ebp    # Save old frame pointer
21   pushl %ebx    # Save callee-saved register
22   pushl %ecx    # Save callee-saved register
23   xorl %eax, %eax   # Initialize accumulator
24   irmovl $4, %ecx   # Set increment value for addresses
25   irmovl $1, %ebx   # Set decrement value for count
26
27 loop:
28   andl %edx, %edx   # Check if count is greater than zero
29   jle finish # If not, jump to finish
30   mrmovl (%esi), %ebp # Move data from source address to temporary
       register
31   rmmovl %ebp, (%edi)   # Move data from temporary register to
       destination address
32   xorl %ebp, %eax   # Accumulate negated data value
33   addl %ecx, %esi   # Update source address
34   addl %ecx, %edi   # Update destination address
35   subl %ebx, %edx   # Decrement count
36   jmp loop    # Jump to loop
37
38 finish:
39   popl %ecx     # Restore callee-saved register
40   popl %ebx     # Restore callee-saved register
41   popl %ebp     # Restore old frame pointer
42   ret       # Return to the caller
```

Code Listing 3: Core Code of copy.ys

2.1.3 Evaluation

We have written three functions based on the original C code, and after consulting the material in the book, we have produced Y86 versions of them. After examination, as the result shown in Figure 1, 2, 3, all three functions are correct.

2.2 Part B

2.2.1 Analysis

In this part, we are going to extend the SEQ processor for a new instruction: `iaddl`, whose function is to add an immediate operand to a register. We will modify the file `seq-full.hcl` to enable the instruction `iaddl`. We will dig into the execution of the instruction, and figure out the details in each stage of the instruction. First, we append the new instruction to the Fetch stage. Then we

```
wsy@wsy-virtual-machine:~/cs/project1-handout-2023/project1-handout-2023/sim/mis
e$ ./yas sum.ys
wsy@wsy-virtual-machine:~/cs/project1-handout-2023/project1-handout-2023/sim/mis
e$ ./yis sum.yo
Stopped in 26 steps at PC = 0xb.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%ecx:   0x00000000      0x00000cba
%esp:   0x00000000      0x00000100

Changes to memory:
0x00f8: 0x00000000      0x0000002f
0x00fc: 0x00000000      0x0000000b
```

Figure 1: Result of sum.ys

```
wsy@wsy-virtual-machine:~/cs/project1-handout-2023/project1-handout-2023/sim/mis
e$ ./yas rsum.ys
wsy@wsy-virtual-machine:~/cs/project1-handout-2023/project1-handout-2023/sim/mis
e$ ./yis rsum.yo
Stopped in 42 steps at PC = 0xb.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%esp:   0x00000000      0x00000100

Changes to memory:
0x00dc: 0x00000000      0x00000c00
0x00e0: 0x00000000      0x00000050
0x00e4: 0x00000000      0x000000b0
0x00e8: 0x00000000      0x00000050
0x00ec: 0x00000000      0x0000000a
0x00f0: 0x00000000      0x00000050
0x00f8: 0x00000000      0x0000002f
0x00fc: 0x00000000      0x0000000b
```

Figure 2: Result of rsum.ys

```
wsy@wsy-virtual-machine:~/cs/project1-handout-2023/project1-handout-2023/sim/mis
e$ ./yas copy_block.ys
wsy@wsy-virtual-machine:~/cs/project1-handout-2023/project1-handout-2023/sim/mis
e$ ./yis copy_block.yo
Stopped in 47 steps at PC = 0xb.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%esp:   0x00000000      0x00000200
%esi:   0x00000000      0x00000018
%edi:   0x00000000      0x00000024

Changes to memory:
0x0018: 0x00000111      0x0000000a
0x001c: 0x00000222      0x000000b0
0x0020: 0x00000333      0x00000c00
0x01f8: 0x00000000      0x0000003b
0x01fc: 0x00000000      0x0000000b
```

Figure 3: Result of copy.ys

append it to the bool `need_regids` and `need_valC`, because it needs to read the register B and the immediate operand. In the Decode stage we set the `srcB` to `rB` and `dstE` to `rB` to store the result. In the memory stage, we set the `aluA` to `valC` and `aluB` to `valB` and append it to the `setCC` because CC needs to be updated.

2.2.2 Code

The code of `seq-full.hcl` is shown in Code Listing 4.

```
#/* $begin seq-all-hcl */
####################################################################
#  HCL Description of Control for Single Cycle Y86 Processor SEQ   #
#  Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010       #
####################################################################

## Your task is to implement the iaddl instruction
## The file contains a declaration of the icodes
## for iaddl (IIADDL) .
## Your job is to add the rest of the logic to make it work

####################################################################
#    C Include's.  Don't alter these                               #
####################################################################

quote '#include <stdio.h>'
quote '#include "isa.h"'
quote '#include "sim.h"'
quote 'int sim_main(int argc, char *argv[]);'
quote 'int gen_pc(){return 0;}'
quote 'int main(int argc, char *argv[])'
quote '  {plusmode=0;return sim_main(argc,argv);}'

####################################################################
#    Declarations.  Do not change/remove/delete any of these       #
####################################################################

##### Symbolic representation of Y86 Instruction Codes #############
intsig INOP   'I_NOP'
intsig IHALT  'I_HALT'
intsig IRRMOVL  'I_RRMOVL'
intsig IIRMOVL  'I_IRMOVL'
intsig IRMMOVL  'I_RMMOVL'
intsig IMRMOVL  'I_MRMOVL'
intsig IOPL 'I_ALU'
intsig IJXX 'I_JMP'
intsig ICALL  'I_CALL'
intsig IRET 'I_RET'
intsig IPUSHL 'I_PUSHL'
intsig IPOPL  'I_POPL'
# Instruction code for iaddl instruction
intsig IIADDL 'I_IADDL'

##### Symbolic represenations of Y86 function codes
      #####
```

```
45 intsig FNONE    'F_NONE'        # Default function code
46
47 ##### Symbolic representation of Y86 Registers referenced explicitly
       #####
48 intsig RESP     'REG_ESP'     # Stack Pointer
49 intsig REBP     'REG_EBP'     # Frame Pointer
50 intsig RNONE    'REG_NONE'    # Special value indicating "no register"
51
52 ##### ALU Functions referenced explicitly
       #####
53 intsig ALUADD 'A_ADD'   # ALU should add its arguments
54
55 ##### Possible instruction status values
       #####
56 intsig SAOK 'STAT_AOK'    # Normal execution
57 intsig SADR 'STAT_ADR'  # Invalid memory address
58 intsig SINS 'STAT_INS'  # Invalid instruction
59 intsig SHLT 'STAT_HLT'  # Halt instruction encountered
60
61 ##### Signals that can be referenced by control logic
       ####################
62
63 ##### Fetch stage inputs    #####
64 intsig pc 'pc'        # Program counter
65 ##### Fetch stage computations    #####
66 intsig imem_icode 'imem_icode'    # icode field from instruction memory
67 intsig imem_ifun  'imem_ifun'     # ifun field from instruction memory
68 intsig icode    'icode'   # Instruction control code
69 intsig ifun   'ifun'    # Instruction function
70 intsig rA   'ra'      # rA field from instruction
71 intsig rB   'rb'      # rB field from instruction
72 intsig valC   'valc'    # Constant from instruction
73 intsig valP   'valp'    # Address of following instruction
74 boolsig imem_error 'imem_error'   # Error signal from instruction
       memory
75 boolsig instr_valid 'instr_valid' # Is fetched instruction valid?
76
77 ##### Decode stage computations   #####
78 intsig valA 'vala'      # Value from register A port
79 intsig valB 'valb'      # Value from register B port
80
81 ##### Execute stage computations  #####
82 intsig valE 'vale'      # Value computed by ALU
83 boolsig Cnd 'cond'      # Branch test
84
85 ##### Memory stage computations   #####
86 intsig valM 'valm'      # Value read from memory
87 boolsig dmem_error 'dmem_error'   # Error signal from data memory
88
89
90 ########################################################################
91 #    Control Signal Definitions.                                       #
92 ########################################################################
93
94 ################ Fetch Stage     ###################################
95
96 # Determine instruction code
```

```
 97 int icode = [
 98         imem_error: INOP;
 99         1: imem_icode;    # Default: get from instruction memory
100 ];
101
102 # Determine instruction function
103 int ifun = [
104         imem_error: FNONE;
105         1: imem_ifun;   # Default: get from instruction memory
106 ];
107
108 bool instr_valid = icode in
109         { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
110                 IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL };
111
112 # Does fetched instruction require a regid byte?
113 bool need_regids =
114         icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
115                         IIRMOVL, IRMMOVL, IMRMOVL, IIADDL };
116
117 # Does fetched instruction require a constant word?
118 bool need_valC =
119         icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL};
120
121 ################ Decode Stage    ###################################
122
123 ## What register should be used as the A source?
124 int srcA = [
125         icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : rA;
126         icode in { IPOPL, IRET } : RESP;
127         1 : RNONE; # Don't need register
128 ];
129
130 ## What register should be used as the B source?
131 int srcB = [
132         icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL } : rB;
133         icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
134         1 : RNONE;  # Don't need register
135 ];
136
137 ## What register should be used as the E destination?
138 int dstE = [
139         icode in { IRRMOVL } && Cnd : rB;
140         icode in { IIRMOVL, IOPL, IIADDL } : rB;
141         icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
142         1 : RNONE;  # Don't write any register
143 ];
144
145 ## What register should be used as the M destination?
146 int dstM = [
147         icode in { IMRMOVL, IPOPL } : rA;
148         1 : RNONE;  # Don't write any register
149 ];
150
151 ################ Execute Stage   ###################################
152
153 ## Select input A to ALU
```

```
154 int aluA = [
155         icode in { IRRMOVL, IOPL } : valA;
156         icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : valC;
157         icode in { ICALL, IPUSHL } : -4;
158         icode in { IRET, IPOPL } : 4;
159         # Other instructions don't need ALU
160 ];
161
162 ## Select input B to ALU
163 int aluB = [
164         icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
165                         IPUSHL, IRET, IPOPL, IIADDL} : valB;
166         icode in { IRRMOVL, IIRMOVL } : 0;
167         # Other instructions don't need ALU
168 ];
169
170 ## Set the ALU function
171 int alufun = [
172         icode == IOPL : ifun;
173         1 : ALUADD;
174 ];
175
176 ## Should the condition codes be updated?
177 bool set_cc = icode in { IOPL, IIADDL };
178
179 ################ Memory Stage    ###################################
180
181 ## Set read control signal
182 bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
183
184 ## Set write control signal
185 bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };
186
187 ## Select memory address
188 int mem_addr = [
189         icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
190         icode in { IPOPL, IRET } : valA;
191         # Other instructions don't need address
192 ];
193
194 ## Select memory input data
195 int mem_data = [
196         # Value from register
197         icode in { IRMMOVL, IPUSHL } : valA;
198         # Return PC
199         icode == ICALL : valP;
200         # Default: Don't write anything
201 ];
202
203 ## Determine instruction status
204 int Stat = [
205         imem_error || dmem_error : SADR;
206         !instr_valid: SINS;
207         icode == IHALT : SHLT;
208         1 : SAOK;
209 ];
210
```

```
################ 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 */
```

Code Listing 4: seq-full.hcl

2.2.3 Evaluation

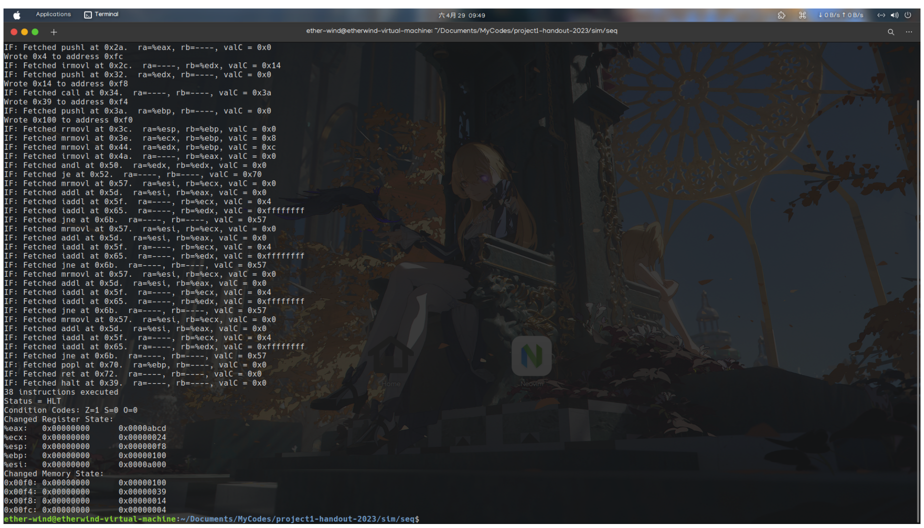

Figure 4: Result of Part B

We made extension to the file `seq-full.hcl` to enable the instruction `iaddl`, and then formed `ssim` to test it. As the result shown in Figure 4, we have the correct result

2.3 Part C

2.3.1 Analysis

Part C can be devided into two parts: the first part is addding `iaddl` operation to the pipeline, which is the same as Part B. The second part is optimizing the

performance of `ncopy.ys`.

To optimize the performance of `ncopy.ys`, first we need to replace the original addition operation with our own definied `iaddl` operation. In the original code, we need to store the immediate value into a register before performing an immediate operand addition operation. This would result in significant time wastage. With the definition of `iaddl`, it is possible to directly add the immediate operand to the register.

To improve instruction pipeline efficiency and allow processors to better predict and optimize instructions, we adopted loop unrolling to optimize the code. We tried four-way unrolling and eight-way unrolling, and finally chose the eight-way unrolling method to optimize the code.

The next optimization direction is to reduce data hazards. Performing the instruction `mrmovl (%rbx), %rsi` followed immediately by the instruction `rmmovl %rsi, (%rcx)` will result in a data hazard. This sequence of instructions creates a hazard known as a data hazard or a RAW (Read After Write) hazard. The first instruction reads data from memory and stores it in the register `%rsi`. The second instruction writes data from the register `%rsi` to memory. However, the second instruction is dependent on the result of the first instruction, which means that it cannot be executed until the first instruction completes. To avoid the hazard, pipeline stalling will be used. However, this will lead to significant time wastage.

To reduce the number of pipeline stalls, we can move the next `mrmovl` instruction between the two instructions. In the first `mrmovl` instruction, data will be loaded into register `%rsi`, and in the second `mrmovl` instruction, the next data will be loaded into register `%rdi`. After the modification, both two `mrmovl` instructions will avoid data hazards.

Due to the utilization of loop unrolling, there will be a few residual loops remained to be executed. Simply, we can directly unroll the remaining loops, decrementing the `len` variable by one every time a loop is executed, and exiting the loop when `len` equals 0. However, this method may not be very effective when dealing with small amounts of data. To address this problem, a ternary search tree can be used to efficiently search for the remaining number of loop iterations. The search tree is shown in Figure 5. Meanwhile, the order of the remaining loops is reversed so that after entering the loop, the code can be executed to the end.

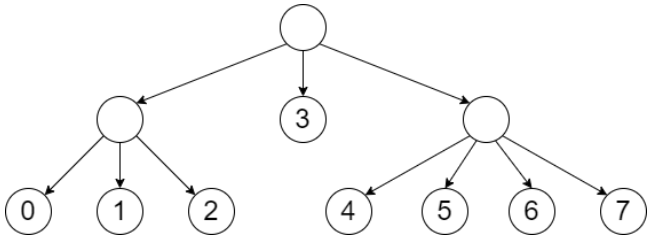

Figure 5: Ternary Search Tree

2.3.2 Code

The code of `pipe-full.hcl` is the same as part B and is omitted here. The core code of `ncopy.ys` is shown in Code Listing 5.

```
# You can modify this portion
  # Loop header
  xorl %eax,%eax    # count = 0;
  iaddl $-8, %edx   # len = len - 8;
  andl %edx,%edx    # len < 0?
  jl Remain   # if so, goto Remain:

Loop: mrmovl (%ebx), %esi # read val from src
  mrmovl 4(%ebx), %edi  # read next val from src
  rmmovl %esi, (%ecx) # store val to dst
  andl %esi, %esi   # val <= 0?
  jle LNpos1    # if so, goto LNpos1:
  iaddl $1, %eax    # count++

LNpos1: rmmovl %edi, 4(%ecx)  # store val to dst
  andl %edi, %edi   # val <= 0?
  jle LNpos2    # if so, goto LNpos2:
  iaddl $1, %eax    # count++

LNpos2: mrmovl 8(%ebx), %esi  # read val from src
  mrmovl 12(%ebx), %edi # read next val from src
  rmmovl %esi, 8(%ecx)  # store val to dst
  andl %esi, %esi   # val <= 0?
  jle LNpos3    # if so, goto LNpos3:
  iaddl $1, %eax    # count++

LNpos3: rmmovl %edi, 12(%ecx) # store val to dst
  andl %edi, %edi   # val <= 0?
  jle LNpos4    # if so, goto LNpos4:
  iaddl $1, %eax    # count++

LNpos4: mrmovl 16(%ebx), %esi # read val from src
  mrmovl 20(%ebx), %edi # read next val from src
  rmmovl %esi, 16(%ecx) # store val to dst
  andl %esi, %esi   # val <= 0?
  jle LNpos5    # if so, goto LNpos5:
  iaddl $1, %eax    # count++

LNpos5: rmmovl %edi,20(%ecx)  # store val to dst
  andl %edi, %edi   # val <= 0?
  jle LNpos6    # if so, goto LNpos6:
  iaddl $1, %eax    # count++

LNpos6: mrmovl 24(%ebx), %esi # read val from src
  mrmovl 28(%ebx), %edi # read next val from src
  rmmovl %esi, 24(%ecx) # store val to dst
  andl %esi, %esi   # val <= 0?
  jle LNpos7    # if so, goto LNpos7:
  iaddl $1, %eax    # count++

LNpos7: rmmovl %edi, 28(%ecx) # store val to dst
```

```
52   andl %edi, %edi   # val <= 0?
53   jle Next    # if so, goto Next:
54   iaddl $1, %eax    # count++
55
56 Next: iaddl $-8, %edx   # len-=8
57   iaddl $32, %ebx   # src+=8
58   iaddl $32, %ecx   # dst+=8
59   andl %edx,%edx    # len >= 0?
60   jge Loop    # if so, goto Loop:
61
62 # Maybe just remain less than 8
63 Remain: iaddl $5, %edx    # len += 8
64   jl Left     # if len < 0, goto Left:
65   jg Right    # if len > 0, goto Right:
66   je R3     # if len = 0, goto R3:
67
68 Left: iaddl $1, %edx    # len++
69   je R2     # if len = 0, goto R2:
70   iaddl $1, %edx    # len++
71   je R1     # if len = 0, goto R1:
72   jmp Done    # goto Done:
73
74 Right:  iaddl $-2, %edx   # len -= 2
75   je R5     # if len = 0, goto R5:
76   jl R4     # if len < 0, goto R4:
77   iaddl $-2, %edx   # len -= 2
78   je R7     # if len = 0, goto R7:
79   jmp R6      # goto R6:
80
81 R7: mrmovl 24(%ebx), %esi # read val7 from src
82   andl %esi, %esi   # val7 <= 0?
83   rmmovl %esi, 24(%ecx) # store val7 to dst
84
85 R6: mrmovl 20(%ebx), %esi # read val6 from src
86   jle R61     # if val7 <= 0, goto R61:
87   iaddl $1, %eax    # count++
88
89 R61:  rmmovl %esi, 20(%ecx) # store val6 to dst
90   andl %esi, %esi   # val6 <= 0?
91
92 R5: mrmovl 16(%ebx), %esi # read val5 from src
93   jle R51     # if val6 <= 0, goto R51:
94   iaddl $1, %eax    # count++
95
96 R51:  rmmovl %esi, 16(%ecx) # store val5 to dst
97   andl %esi, %esi   # val5 <= 0?
98
99 R4: mrmovl 12(%ebx), %esi # read val4 from src
100   jle R41     # if val5 <= 0, goto R41:
101   iaddl $1, %eax    # count++
102
103 R41:  rmmovl %esi, 12(%ecx) # store val4 to dst
104   andl %esi, %esi   # val4 <= 0?
105
106 R3: mrmovl 8(%ebx), %esi  # read val3 from src
107   jle R31     # if val4 <= 0, goto R31:
108   iaddl $1, %eax    # count++
```

```
109
110 R31:  rmmovl %esi, 8(%ecx)  # store val3 to dst
111   andl %esi, %esi   # val3 <= 0?
112
113 R2: mrmovl 4(%ebx), %esi  # read val2 from src
114   jle R21     # if val3 <= 0, goto R21:
115   iaddl $1, %eax    # count++
116
117 R21:  rmmovl %esi, 4(%ecx)  # store val2 to dst
118   andl %esi, %esi   # val2 <= 0?
119
120 R1: mrmovl (%ebx), %esi # read val1 from src
121   jle R11     # if val2 <= 0, goto R11
122   iaddl $1, %eax    # count++
123
124 R11:  rmmovl %esi, (%ecx) # store val1 to dst
125   andl %esi, %esi   # val <= 0?
126   jle Done    # if so, goto Done:
127   iaddl $1, %eax    # count++
```

Code Listing 5: Core Code of ncopy.ys

2.3.3 Evaluation

Figure 6 shows the correctness of the code. Figure 7 shows that the average CPE is 9.09 and the score is 60.0, which indicates that the code performance is good.

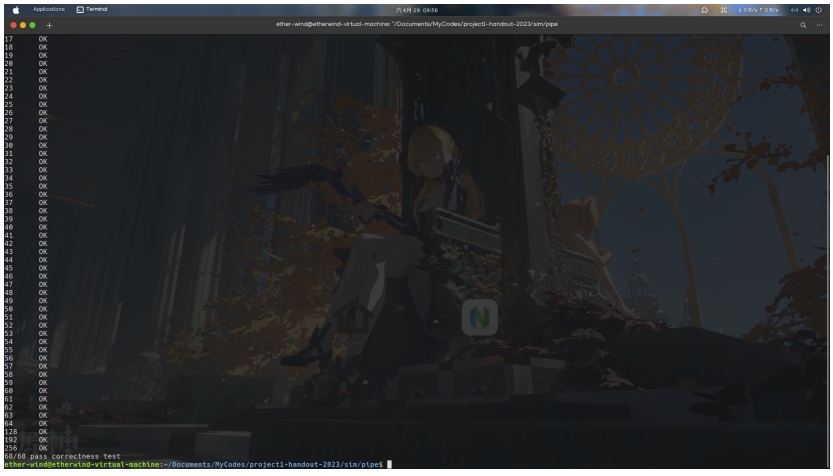

Figure 6: Correctness

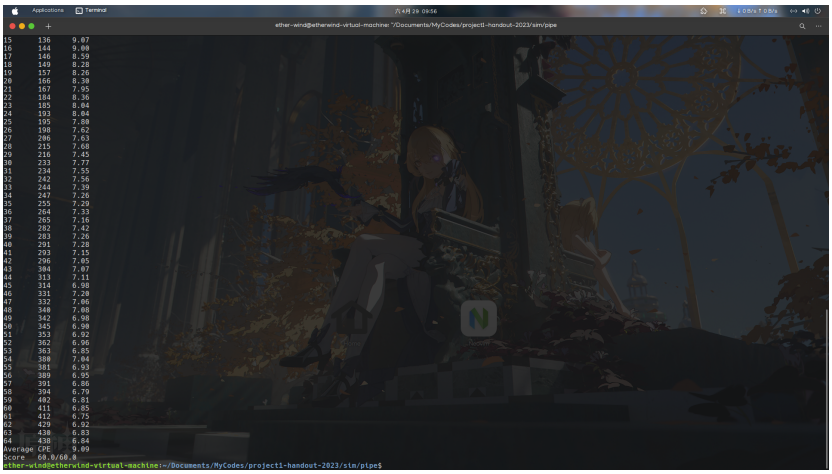

Figure 7: Benchmark

3 Conclusion

3.1 Problems

We encountered the obstacles of understanding and using the Y86 tools, but we compared it to the X86 instructions and finally got a comprehension of it. We faced the trouble of using Linux and running the codes on virtual machine but learned to do it. We found it difficult to optimize the pipeline, but by reviewing the lectures of Computer Architecture we got it done perfectly.

3.2 Achievements

In Part C, we adjust code order to avoid data hazards and use several techniques, such as loop unrolling and ternary search tree, to improve the performance of the code. The final average CPE is 9.09, which is a nice performance.

We made adequate annotation in the code part, which greatly enhanced the coding readability. Through the team cooperation, we collaborated smoothly and things did work out.