

# Project 1: Optimizing the Performance of a Pipelined Processor

000, Ziqi Zhao, bugenzhao@sjtu.edu.cn  
001, Yimin Zhao, doctormin@sjtu.edu.cn

May 3, 2020

## 1 Introduction

### Part A

In part A, we write three simple assembly programs to mimic three functions in example.c. Based on ensuring correctness, we especially focus on the functional equivalence with the example C functions. By selecting and placing labels in the assembly code appropriately, the code is also very readable.

### Part B

In part B, we modify the HCL file of the SEQ to add a new instruction — iaddl. The following is the roadmap to finish this part:

- Clarify the computation process of iadd and write it down at the beginning in seq-full.hcl.
- Add any dependence relations of iaddl to all boosigs.
- Design the datapath for iaddl (generate control signals for src and dst)

### Part C

We achieve full scores in the benchmark testing **in just 2 hours**, but we **spent 2 more days** researching all the potential methods to optimize the performance even further. The following is our roadmap:

- Change the order of the instruction sequence to avoid data hazard and structure hazards, which leaves  $CPE = 12.96$ .
- Beyond the changes on instructions order, we use loop unrolling to reduce the number of conditional check and registers updating, which leaves  $CPE = 9.83$
- Use a binary search tree to find the precise remaining number of loops after several rounds of unrolling to achieve complete unrolling, which leaves  $CPE = 8.95$
- Modify the HCL file to achieve 100% accuracy in branch prediction for certain code pattern, which brings  $CPE$  down to 7.78.

### Contribution

**Ziqi Zhao** : Part A (coding) & Part B (coding) & Part C (coding & designing) & project report(reviewing)

**Yimin Zhao** : Part A (reviewing & coding) & Part B (reviewing) & Part C (designing) & project report(writing)

## 2 Experiments

### 2.1 Part A

#### 2.1.1 Analysis

In this part, we are asked to implement and simulate three y86 programs. From a macro point of view, this part is relatively easy. But there are plenty of optimizations worth exploring in terms of code readability and elegance.

#### Difficult Point

- Always pull the correct element from the stack.
- Be careful to protect the callee-save register.
- Implement function recursion smartly.

#### Core Technique

- Mimicking C functions, division of functional areas with enough and clear label.
- Get the fastest completion speed by coding line by line referring to C language functions
- Always draw a picture of the stack to ensure the correctness of fetching a variable.

#### 2.1.2 Code

**sum.js**

```
1 # 518030910211 ZiqiZhao
2 # 518030910188 YiminZhao
3
4 # Set up stack
5     .pos    0
6     irmovl  stack, %esp
7     rrmovl  %esp, %ebp
8     pushl   %edx           # save %edx
9     irmovl  ele1, %eax
10    pushl   %eax
11    call    sum_list
```

```

12         popl    %edx                # flatten the stack for ele1
13         popl    %edx                # restore %edx
14         halt
15
16 # Sample linked list
17 .align 4
18 ele1:
19         .long    0x00a
20         .long    ele2
21 ele2:
22         .long    0x0b0
23         .long    ele3
24 ele3:
25         .long    0xc00
26         .long    0
27
28 # sum_list func
29 sum_list:
30         pushl    %ebp                # enter
31         pushl    %ecx                # save %ecx
32         rrmovl   %esp, %ebp
33         xorl     %eax, %eax          # clear %eax
34         mrmovl   12(%ebp), %edx      # get ls
35         jmp      test
36 loop:
37         mrmovl   (%edx), %ecx
38         addl     %ecx, %eax
39         mrmovl   4(%edx), %edx
40 test:
41         andl     %edx, %edx
42         jne      loop                # %edx != 0
43 return:
44         rrmovl   %ebp, %esp          # leave
45         popl     %ecx
46         popl     %ebp
47         ret
48
49 # Stack
50         .pos     0x400
51 stack:

```

## rsum.ys

```
1  # 518030910211 ZiqiZhao
2  # 518030910188 YiminZhao
3
4  # Set up stack
5      .pos      0
6      irmovl    stack, %esp
7      rrmovl    %esp, %ebp
8      pushl     %edx
9      irmovl    ele1, %eax
10     pushl     %eax
11     call      rsum_list
12     popl      %edx          # eat ele1
13     popl      %edx          # restore %edx
14     halt
15
16 # Sample linked list
17 .align 4
18 ele1:
19     .long      0x00a
20     .long      ele2
21 ele2:
22     .long      0x0b0
23     .long      ele3
24 ele3:
25     .long      0xc00
26     .long      0
27
28 # rsum_list func
29 rsum_list:
30     pushl     %ebp          # enter
31     rrmovl    %esp, %ebp
32     xorl      %eax, %eax
33     mrmovl    8(%ebp), %edx # get ls
34     andl      %edx, %edx
35     je        return      # ls == NULL
36 do:
37     pushl     %ebx          # save %ebx
38     mrmovl    (%edx), %ebx  # mov ls->val to %ebx
39     mrmovl    4(%edx), %eax
40     pushl     %eax          # push ls->next
41     call      rsum_list
42     addl      %ebx, %eax    # ret = val + ret
43     popl      %edx          # eat para
44     popl      %ebx          # restore %ebx
```

```

45 return:
46     rrmovl %ebp, %esp    # leave
47     popl   %ebp
48     ret
49
50
51 # Stack
52     .pos    0x400
53 stack:

```

### copy.js

```

1  # 518030910211 ZiqiZhao
2  # 518030910188 Yimin Zhao
3
4  # Set up stack
5      .pos    0
6      irmovl  stack, %esp
7      rrmovl  %esp, %ebp
8      irmovl  $3, %eax
9      pushl   %eax
10     irmovl  src, %eax
11     pushl   %eax
12     irmovl  dest, %eax
13     pushl   %eax
14     call    copy_block
15     halt
16 .align 4
17 # Source block
18 src:
19     .long 0x00a
20     .long 0x0b0
21     .long 0xc00
22
23 # Destination block
24 dest:
25     .long 0x111
26     .long 0x222
27     .long 0x333
28
29 copy_block:
30     pushl   %ebp
31     rrmovl  %esp, %ebp
32     pushl   %ecx
33     pushl   %edx

```

```

34     pushl    %edi
35     irmovl   $0, %eax          # %eax = result = 0
36     mrmovl   16(%ebp), %ecx    # %ecx = len
37     mrmovl   12(%ebp), %edx    # %edx = src
38     mrmovl   8(%ebp), %edi     # %edi = dest
39     jmp      while_loop
40
41 while_loop:
42     andl     %ecx, %ecx        # check if %ecx == 0?
43     jle      return          # if so, jump to "return"
44     mrmovl   (%edx), %esi      # %esi = val = *src
45     irmovl   $4, %ebx         # %ebx = 4
46     addl     %ebx, %edx        # src++
47     rmmovl   %esi, (%edi)     # *dest = val
48     addl     %ebx, %edi        # dest++
49     xorl     %esi, %eax
50     irmovl   $-1, %ebx
51     addl     %ebx, %ecx        # len--
52     jmp      while_loop
53
54 return:
55     popl     %edi
56     popl     %edx
57     popl     %ecx
58     rrmovl   %ebp, %esp
59     popl     %ebp
60     ret
61 # Stack
62     .pos     0x400
63 stack:

```

### 2.1.3 Evaluation

#### sum.y

```
../yas sum.y
../yis sum.yo
Stopped in 36 steps at PC = 0x1b. Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000    0x00000cba
%esp: 0x00000000    0x00000400
%ebp: 0x00000000    0x00000400

Changes to memory:
0x03f0: 0x00000000    0x00000400
0x03f4: 0x00000000    0x00000017
0x03f8: 0x00000000    0x0000001c
```

Figure 1: partA-sum.y

- The `%eax` register has the correct value which is the return value of the function—`0xcba`.
- The memory is not corrupted since all the modifications locate at the stack whose starting address is set to be `0x400`.

#### rsum.y

```
../yas rsum.y
../yis rsum.yo
Stopped in 69 steps at PC = 0x1b. Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax: 0x00000000    0x00000cba
%esp: 0x00000000    0x00000400
%ebp: 0x00000000    0x00000400

Changes to memory:
0x03c0: 0x00000000    0x000003d0
0x03c4: 0x00000000    0x0000005c
0x03cc: 0x00000000    0x000000b0
0x03d0: 0x00000000    0x000003e0
0x03d4: 0x00000000    0x0000005c
0x03d8: 0x00000000    0x0000002c
0x03e0: 0x00000000    0x000003f0
0x03e4: 0x00000000    0x0000005c
0x03e8: 0x00000000    0x00000024
0x03f0: 0x00000000    0x00000400
0x03f4: 0x00000000    0x00000017
0x03f8: 0x00000000    0x0000001c
```

Figure 2: partA-rsum.y

- The `%eax` register has the correct value which is the return value of the function—`0xcba`.
- The memory is not corrupted since all the modifications locate at the stack whose starting address is set to be `0x400`.

#### copy.y8

```

../yas copy.y8
../yis copy.yo
Stopped in 61 steps at PC = 0x25. Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000 0x000000cba
%ebx: 0x00000000 0xffffffff
%esp: 0x00000000 0x0000003f4
%ebp: 0x00000000 0x000000400
%esi: 0x00000000 0x000000c00

Changes to memory:
0x0034: 0x00000111 0x0000000a
0x0038: 0x00000222 0x000000b0
0x003c: 0x00000333 0x000000c00
0x003e: 0x00000000 0x000000400
0x003f: 0x00000000 0x00000025
0x0040: 0x00000000 0x00000034
0x0041: 0x00000000 0x00000028
0x0042: 0x00000000 0x00000003

```

Figure 3: partA-copy.y8

- The `%eax` register has the correct value which is the return value of the function—`0xcba`.
- Values are written into the memory correctly as shown in the first three rows in the "Changes to memory" part in Figure 3
- The memory is not corrupted since all the modifications other than 3 source values locate at the stack whose starting address is set to be `0x400`.

## 2.2 Part B

### 2.2.1 Analysis

In part B, we are asked to extend the SEQ processor to support instruction "iaddl" by modifying SEQ-full.hcl. Once we understand the processing logic and HCL syntax of the y86 seq processor, the problem becomes very simple. We can do it in five minutes because all we need to do is change the followings in the HCL

- Add "IIADDL" in the choices region of (bool) `instr_valid` since `iaddl` is a valid instruction.



- Add "IIADDL" in the choices region of (bool) need\_regid since iaddl operation involves one register.
- Add "IIADDL" in the choices region of (bool) need\_valC since iaddl operation involves one constend (represented by valC in the circuit of y86 SEQ).
- Add "IIADDL" in the choices region of (bool) set\_cc since iaddl operation involves ALU operation which will set flags.
- When icode is IIADDL, alufun will be ALUADD since the operation is "adding" the constant to rB.
- When icode is IIADDL, srcB is from rB since the second operand of iaddl is a register.
- When icode is IIADDL, dstE (where the result from ALU is passed towards) is rB since "iaddl constant, rB" means rB += constant (rB is updated).
- When icode is IIADDL, aluA (the first op) is valC (the constant in the instruction) since "iaddl constant, rB" means the first op is the constant (valC).
- When icode is IIADDL, aluB (the second op) is valB (the value of the second register that is read) for the same reason above.

[In this part, you should give an overall analysis for the task, like difficult point, core technique and so on.]

### 2.2.2 Code

#### Modifications in SEQ-full.hcl

```

1  -----
2  bool instr_valid = icode in
3  { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
4    IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL };
5  -----
6  # Does fetched instruction require a regid byte?
7  bool need_regids =
8      icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
9        IIRMOVL, IRMMOVL, IMRMOVL, IIADDL };
10 -----
11 # Does fetched instruction require a constant word?
12 bool need_valC =
13     icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL };
14 -----
15 ## What register should be used as the B source?
16 int srcB = [
17     icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL } : rB;

```

```

18         icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
19         1 : RNONE; # Don't need register
20     ];
21     -----
22     ## What register should be used as the E destination?
23     int dstE = [
24         icode in { IRRMOVL } && Cnd : rB;
25         icode in { IIRMOVL, IOPL, IIADDL } : rB;
26         icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
27         1 : RNONE; # Don't write any register
28     ];
29     -----
30     ## Select input A to ALU
31     int aluA = [
32         icode in { IRRMOVL, IOPL } : valA;
33         icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : valC;
34         icode in { ICALL, IPUSHL } : -4;
35         icode in { IRET, IPOPL } : 4;
36         # Other instructions don't need ALU
37     ];
38     -----
39     ## Select input B to ALU
40     int aluB = [
41         icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
42             IPUSHL, IRET, IPOPL, IIADDL } : valB;
43         icode in { IRRMOVL, IIRMOVL } : 0;
44         # Other instructions don't need ALU
45     ];
46     -----
47     ## Set the ALU function
48     int alufun = [
49         icode == IOPL : ifun;
50         icode == IIADDL : ALUADD;
51         1 : ALUADD;
52     ];
53     -----
54     ## Should the condition codes be updated?
55     bool set_cc = icode in { IOPL, IIADDL };
56     -----

```

## 2.2.3 Evaluation

```
* seq git:(master) # cd ../y86-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 "ISA Check" *.seq
asum.seq:ISA Check Succeeds
asum.seq:ISA Check Succeeds
cjr.seq:ISA Check Succeeds
j-cc.seq:ISA Check Succeeds
poptest.seq:ISA Check Succeeds
prog1.seq:ISA Check Succeeds
prog2.seq:ISA Check Succeeds
prog3.seq:ISA Check Succeeds
prog4.seq:ISA Check Succeeds
prog5.seq:ISA Check Succeeds
prog6.seq:ISA Check Succeeds
prog7.seq:ISA Check Succeeds
prog8.seq:ISA Check Succeeds
pushquestion.seq:ISA Check Succeeds
pushtest.seq:ISA Check Succeeds
ret-hazard.seq:ISA Check Succeeds
rm asum.seq asum.seq cjr.seq j-cc.seq poptest.seq pushquestion.seq pushtest.seq prog1.seq prog2.seq prog3.seq prog4.seq prog5.seq prog6.seq prog7.seq prog8.seq ret-hazard.seq
```

Figure 4: part B benchmark test

```
→ ptest git:(master) # make SIM=../seq/ssim
./optest.pl -s ../seq/ssim
Simulating with ../seq/ssim
All 49 ISA Checks Succeed
./jtest.pl -s ../seq/ssim
Simulating with ../seq/ssim
All 64 ISA Checks Succeed
./ctest.pl -s ../seq/ssim
Simulating with ../seq/ssim
All 22 ISA Checks Succeed
./htest.pl -s ../seq/ssim
Simulating with ../seq/ssim
All 600 ISA Checks Succeed
→ ptest git:(master) #
```

Figure 5: part B regression test

```
→ ptest git:(master) # cd ../ptest && make SIM=../seq/ssim TFLAGS=-i
./optest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
All 58 ISA Checks Succeed
./jtest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
All 96 ISA Checks Succeed
./ctest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
All 22 ISA Checks Succeed
./htest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
All 756 ISA Checks Succeed
→ ptest git:(master) #
```

Figure 6: part B iaddl test

## 2.3 Part C

### 2.3.1 Analysis

In this part, we were asked to speed up the program `ncopy.y8` as much as possible by modifying the `ncopy.y8` and `HCL`. The following is our roadmap:

#### Avoid Load and Use: CPI → 12.96

For the pipeline design in CS:APP 2e, "load and use" or "rmmovl" then "rmmovl" will cause penalty, which must be avoided to improve the performance. On the one hand, we rearranged the order of instructions to avoid stalling as much as possible. On the other hand, we use two registers to store the variable "val", loading them separately and ahead of time.

#### 10-way Loop Unrolling: CPI → 9.83

There's much overhead in testing and updating procedure of loops, and one way to minimize it is to perform a technique named "loop unrolling". That is, we do multiple loops and update the relevant data at once, to reduce the number of times we execute the 'add' and 'jxx' instructions.

#### Search Tree for Remaining Elements: CPI → 8.95

For large inputs, the more ways we unroll the loops, the better the program performs. However, for small inputs, it is important to choose a good method to process the remaining elements. The simplest way is to write another loop for them, but a much better way is to totally unroll the code, that is, jump to different position for different number of remaining ones. Since Y86 does not support relative jump instruction, we designed a search tree to get the correct jump destination for each possibility.

---

The above optimization took us two hours, so far we have reached full marks But **can it be even faster** ? We spent another two days poring over the implementation logic and `HCL` and other files of the y86 pipeline. And it finally gets us here:

---

#### Optimized for Our Branch Prediction Design: CPI → 7.78

For this program, a significant performance factor is the branch prediction failure for `count++`. In "`pipe-zzcc.hcl`", we made a special optimization for the situation like this:

Instruction:	any instruction	non-alu instruction	jxx
Stages:	EX	ID	IF

Note that in this case, we can forward the conditions from EX stage to IF and predict the branch with 100% accuracy. Thus, we optimized the program to ensure that there were as many of these patterns as possible and took much advantage of it, which leads to an average CPE of 7.78.

### 2.3.2 Code

#### —10-Way Loop Unrolling—

```

1  # Entry
2      iaddl $-9, %edx          # len -= 9, i.e., initial_len <= 9?
3      irmovl $0, %eax         # count = 0
4      jle Remaining          # if so, goto Remaining
5
6  # Loop unrolling part
7  Loop0:
8      mrmovl (%ebx), %esi     # valA = src[0]
9      mrmovl 4(%ebx), %edi    # valB = src[1]
10     andl %esi, %esi         # valA <= 0?
11     rmmovl %esi, (%ecx)     # dst[0] = valA
12     jle Loop1              # if so, goto next loop
13     iaddl $1, %eax         # count++
14  Loop1:
15     mrmovl 8(%ebx), %esi     # valA = src[2]
16     andl %edi, %edi         # valB <= 0?
17     rmmovl %edi, 4(%ecx)    # dst[1] = valB
18     jle Loop2              # if so, goto next loop
19     iaddl $1, %eax         # count++
20  Loop2:
21     mrmovl 12(%ebx), %edi    # valB = src[3]
22     andl %esi, %esi         # valA <= 0?
23     rmmovl %esi, 8(%ecx)    # dst[2] = valA
24     jle Loop3              # if so, goto next loop
25     iaddl $1, %eax         # count++
26  Loop3:
27     mrmovl 16(%ebx), %esi    # valA = src[4]
28     andl %edi, %edi         # valB <= 0?
29     rmmovl %edi, 12(%ecx)   # dst[3] = valB
30     jle Loop4              # if so, goto next loop
31     iaddl $1, %eax         # count++
32  Loop4:
33     mrmovl 20(%ebx), %edi    # valB = src[5]
34     andl %esi, %esi         # valA <= 0?
35     rmmovl %esi, 16(%ecx)   # dst[4] = valA
36     jle Loop5              # if so, goto next loop
37     iaddl $1, %eax         # count++
38  Loop5:
39     mrmovl 24(%ebx), %esi    # valA = src[6]
40     andl %edi, %edi         # valB <= 0?
41     rmmovl %edi, 20(%ecx)   # dst[5] = valB
42     jle Loop6              # if so, goto next loop
43     iaddl $1, %eax         # count++
44  Loop6:
45     mrmovl 28(%ebx), %edi    # valB = src[7]

```

```

46      andl %esi, %esi      # valA <= 0?
47      rmmovl %esi, 24(%ecx) # dst[6] = valA
48      jle Loop7           # if so, goto next loop
49      iaddl $1, %eax       # count++
50 Loop7:
51      mrmovl 32(%ebx), %esi # valA = src[8]
52      andl %edi, %edi      # valB <= 0?
53      rmmovl %edi, 28(%ecx) # dst[7] = valB
54      jle Loop8           # if so, goto next loop
55      iaddl $1, %eax       # count++
56 Loop8:
57      mrmovl 36(%ebx), %edi # valB = src[9]
58      andl %esi, %esi      # valA <= 0?
59      rmmovl %esi, 32(%ecx) # dst[8] = valA
60      jle Loop9           # if so, goto next loop
61      iaddl $1, %eax       # count++
62 Loop9:
63      andl %edi, %edi      # valB <= 0?
64      rmmovl %edi, 36(%ecx) # dst[9] = valB
65      jle LoopEnd         # if so, goto loop end
66      iaddl $1, %eax
67 LoopEnd:
68      iaddl $40, %ecx       # dst += 10 * 4
69      iaddl $40, %ebx       # src += 10 * 4
70      iaddl $-10, %edx      # len -= 10
71      jg Loop0             # if so, goto Loop0
72                          # else, goto process remaining elements

```

### —Binary Search Tree for Finding the Number of Remaining Loops—

```

1  # The following block is a binary search tree to
2  # find the number of remaining loops
3  # (which must be less than 10) at minimal cost
4  Remaining:
5      iaddl $6, %edx        # [-9,0] -> [-3,6]      (+3)
6  RemTest:
7      irmovl $0, %esi
8      jg RemTestR
9      je Rem3
10 RemTestL:
11      iaddl $2, %edx        # [-3,-1] -> [-1,1]      (+1)
12      je Rem1
13      jg Rem2
14      jmp Done              # -1 + 1 = 0
15 RemTestR:

```

```

16      iaddl $-3, %edx      # [1,6] -> [-2,3]      (+6)
17      jg RemTestRR
18      je Rem6
19 RemTestRL:
20      iaddl $1, %edx      # [-2,-1] -> [-1,0]      (+5)
21      jl Rem4
22      je Rem5
23 RemTestRR:
24      iaddl $-2, %edx      # [1,3] -> [-1,1]      (+8)
25      jl Rem7
26      je Rem8

```

### —Unrolling of Remaining Loops—

```

1 Rem9:
2      mrmovl 32(%ebx), %esi # valA = src[8]
3      rmmovl %esi, 32(%ecx) # dst[8] = valA
4 Rem8:  # Note that %esi == 0, directly jumping here
5        # implies that RemXb will performs correctly.
6      andl %esi, %esi      # valA <= 0?
7      mrmovl 28(%ebx), %esi # valA = src[7]
8      jle Rem8b           # if so, goto Rem8b
9      iaddl $1, %eax      # count++
10 Rem8b: rmmovl %esi, 28(%ecx) # dst[7] = valA
11 Rem7:
12      andl %esi, %esi      # valA <= 0?
13      mrmovl 24(%ebx), %esi # valA = src[6]
14      jle Rem7b           # if so, goto Rem7b
15      iaddl $1, %eax      # count++
16 Rem7b: rmmovl %esi, 24(%ecx) # dst[6] = valA
17 Rem6:
18      andl %esi, %esi      # valA <= 0?
19      mrmovl 20(%ebx), %esi # valA = src[5]
20      jle Rem6b           # if so, goto Rem6b
21      iaddl $1, %eax      # count++
22 Rem6b: rmmovl %esi, 20(%ecx) # dst[5] = valA
23 Rem5:
24      andl %esi, %esi      # valA <= 0?
25      mrmovl 16(%ebx), %esi # valA = src[4]
26      jle Rem5b           # if so, goto Rem5b
27      iaddl $1, %eax      # count++
28 Rem5b: rmmovl %esi, 16(%ecx) # dst[4] = valA
29 Rem4:
30      andl %esi, %esi      # valA <= 0?
31      mrmovl 12(%ebx), %esi # valA = src[3]

```

```

32         jle Rem4b          # if so, goto Rem4b
33         iaddl $1, %eax      # count++
34 Rem4b:  rmmovl %esi, 12(%ecx) # dst[3] = valA
35 Rem3:
36         andl %esi, %esi     # valA <= 0?
37         mrmovl 8(%ebx), %esi # valA = src[2]
38         jle Rem3b          # if so, goto Rem3b
39         iaddl $1, %eax      # count++
40 Rem3b:  rmmovl %esi, 8(%ecx) # dst[2] = valA
41 Rem2:
42         andl %esi, %esi     # valA <= 0?
43         mrmovl 4(%ebx), %esi # valA = src[1]
44         jle Rem2b          # if so, goto Rem2b
45         iaddl $1, %eax      # count++
46 Rem2b:  rmmovl %esi, 4(%ecx) # dst[1] = valA
47 Rem1:
48         andl %esi, %esi     # valA <= 0?
49         mrmovl (%ebx), %esi  # valA = src[0]
50         jle Rem1b          # if so, goto Rem1b
51         iaddl $1, %eax      # count++
52 Rem1b:
53         andl %esi, %esi     # valA <= 0?
54         rmmovl %esi, (%ecx)  # dst[0] = valA
55         jle Done           # if so, goto Done
56         iaddl $1, %eax      # count++

```

—Modification to hcl (all in pipe-zzcc.hcl)—

```

1  -----add following definition-----
2  quote 'int gen_aluA();'          # Declaration of gen_aluA
3  quote 'int gen_aluB();'          # Declaration of gen_aluB
4
5  # For JXX in ID and ALU in EX, check the cc generated by ALU
6  # Note that the simulator do not generate 'cc_in' correctly
7  boolsig f_cnd_alu
8      'cond_holds(compute_cc(id_ex_curr->ifun,
9                      gen_aluA(), gen_aluB()),
10                  if_id_next->ifun)'
11
12  # For JXX in ID and non-ALU in EX, check the cc register
13  boolsig f_cnd_other 'cond_holds(cc, if_id_next->ifun)'
14  -----modify f_predPC-----
15  # Predict next value of PC
16  int f_predPC = [
17      f_icode == ICALL : f_valC;

```



```

18     f_icode == IJXX && f_ifun == UNCOND : f_valC;
19
20     # Decode stage is ALU and will set CC -> always taken by
21     ↪ default
22     f_icode == IJXX && (D_icode in {IOPL, IIADDL}): f_valC;
23
24     # Decode stage is not ALU
25     # Execute stage is ALU -> compute CC
26     f_icode == IJXX && (E_icode in {IOPL, IIADDL}) &&
27     !f_cnd_alu : f_valP;
28
29     # Execute stage is not ALU -> check cc -> ZF SF OF
30     f_icode == IJXX && !(E_icode in {IOPL, IIADDL}) &&
31     !f_cnd_other : f_valP;
32     # Other JXX
33     f_icode == IJXX : f_valC;
34
35     # Otherwise
36     1 : f_valP;
37 ];
38 -----add conditions to D_Bubble & E_Bubble-----
39 bool D_bubble =
40     # Mispredicted branch taken
41     (E_icode == IJXX && E_ifun != UNCOND &&
42     (M_icode in {IOPL, IIADDL}) && !e_Cnd) ||
43     # Stalling at fetch while ret passes through pipeline
44     # but not condition for a load/use hazard
45     !(E_icode in { IMRMOVL, IPOPL }
46     && E_dstM in { d_srcA, d_srcB })
47     && IRET in { D_icode, E_icode, M_icode };
48
49 # Should I stall or inject a bubble into Pipeline Register E?
50 # At most one of these can be true.
51 bool E_stall = 0;
52 bool E_bubble =
53     # Mispredicted branch taken
54     (E_icode == IJXX && E_ifun != UNCOND &&
55     (M_icode in {IOPL, IIADDL}) && !e_Cnd) ||
56     # Conditions for a load/use hazard
57     E_icode in { IMRMOVL, IPOPL } &&
58     E_dstM in { d_srcA, d_srcB };

```

### 2.3.3 Evaluation

```
Ξ (bz-parallel) sim/pipe git:(master) ▶ make testpsim
make -C ../ptest SIM=../pipe/psim TFLAGS=-i
make[1]: Entering directory '/home/bugenzhao/ComputerArch-Prj1/sim/ptest'
./optest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
All 58 ISA Checks Succeed
./jtest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
All 96 ISA Checks Succeed
./ctest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
All 22 ISA Checks Succeed
./htest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
All 756 ISA Checks Succeed
make[1]: Leaving directory '/home/bugenzhao/ComputerArch-Prj1/sim/ptest'
```

Figure 7: partC regression test

```
make -C ../y86-code testpsim
make[1]: Entering directory '/home/bugenzhao/ComputerArch-Prj1/sim/y86-code'
../pipe/psim -t asum.yo > asum.pipe
../pipe/psim -t asumr.yo > asumr.pipe
../pipe/psim -t cjr.yo > cjr.pipe
../pipe/psim -t j-cc.yo > j-cc.pipe
../pipe/psim -t poptest.yo > poptest.pipe
../pipe/psim -t pushquestion.yo > pushquestion.pipe
../pipe/psim -t pushtest.yo > pushtest.pipe
../pipe/psim -t prog1.yo > prog1.pipe
../pipe/psim -t prog2.yo > prog2.pipe
../pipe/psim -t prog3.yo > prog3.pipe
../pipe/psim -t prog4.yo > prog4.pipe
../pipe/psim -t prog5.yo > prog5.pipe
../pipe/psim -t prog6.yo > prog6.pipe
../pipe/psim -t prog7.yo > prog7.pipe
../pipe/psim -t prog8.yo > prog8.pipe
../pipe/psim -t ret-hazard.yo > ret-hazard.pipe
grep "ISA Check" *.pipe
asum.pipe:ISA Check Succeeds
asumr.pipe:ISA Check Succeeds
cjr.pipe:ISA Check Succeeds
j-cc.pipe:ISA Check Succeeds
poptest.pipe:ISA Check Succeeds
prog1.pipe:ISA Check Succeeds
prog2.pipe:ISA Check Succeeds
prog3.pipe:ISA Check Succeeds
prog4.pipe:ISA Check Succeeds
prog5.pipe:ISA Check Succeeds
prog6.pipe:ISA Check Succeeds
prog7.pipe:ISA Check Succeeds
prog8.pipe:ISA Check Succeeds
pushquestion.pipe:ISA Check Succeeds
pushtest.pipe:ISA Check Succeeds
ret-hazard.pipe:ISA Check Succeeds
rm asum.pipe asumr.pipe cjr.pipe j-cc.pipe poptest.pipe pushquestion.pipe push
pe prog8.pipe ret-hazard.pipe
make[1]: Leaving directory '/home/bugenzhao/ComputerArch-Prj1/sim/y86-code'
```

Figure 8: partC benchmark test

```

60      OK
61      OK
62      OK
63      OK
64      OK
128     OK
192     OK
256     OK
68/68 pass correctness test

```

Figure 9: partC correctness test

```

58      325      5.60
59      328      5.56
60      336      5.60
61      338      5.54
62      346      5.58
63      345      5.48
64      356      5.56
Average CPE      7.79
Score    60.0/60.0

```

Figure 10: partC CPE test

### 3 Conclusion

In this project, we completed the tasks of three parts, which were gradually developed. The first part made us familiar with y86 assembly syntax, the second part made us familiar with y86 SEQ circuit logic, and the third part encouraged us to transform assembly code and y86 pipeline circuit logic. The following is a summary of the completion of the three parts:

#### Part A

- We write assembly code for three simple functions.
- We take care to protect the stack and registers.
- We focus on the readability and functional equivalence of the code.

#### Part B

- We modify SEQ-full.hcl to add an instruction: iaddl.

#### Part C

- We reorder the instructions to avoid hazards.

- We do 10-way loop unrolling to speed up the while loop.
- We create a binary search tree to find the number of remaining loops at the minimal cost and then completely unroll the loops.
- We modify the HCL file of the pipeline to optimize the branch prediction, achieving 100% accuracy for a certain code mode (non-ALU followed by jxx).

### 3.1 Problems

We only meet problems in Part C. They are two unsuccessful attempts to modify the pipeline logic to lift accuracy of branch prediction to 100%. Based on attempt 2, we have made some small changes to achieve the goal, which is explained in detail in **3.2 achievement**.

#### 3.1.1 Attempt 1

In this attempt, we look at a particular code distribution, which is "andl op1, op2 → JXX des"(because in our code, there are a lot of this kind of code distributions). We hope that the branch prediction of JXX under this distribution can reach 100% accuracy. The logic is shown in Figure 11:

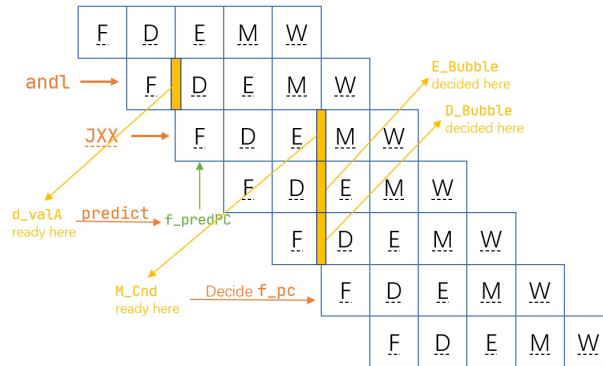


Figure 11: partC attempt1

Basically, it means that when we detect a JXX after an andl, we will do branch prediction according to the value of the first op of "andl". Because in our ncopy.ys, there are a lot of this kind of codes:

```

Loop0:
    mrmovl (%ebx), %esi # valA = src[0]
    mrmovl 4(%ebx), %edi # valB = src[1]
    rmmovl %esi, (%ecx) # dst[0] = valA
    andl %esi, %esi # valA ≤ 0?
    jle Loop1 # if so, goto next loop
    iaddl $1, %eax # count++
Loop1:
    mrmovl 8(%ebx), %esi # valA = src[2]
    rmmovl %edi, 4(%ecx) # dst[1] = valB
    andl %edi, %edi # valB ≤ 0?
    jle Loop2 # if so, goto next loop
    iaddl $1, %eax # count++
Loop2:
    mrmovl 12(%ebx), %edi # valB = src[3]
    rmmovl %esi, 8(%ecx) # dst[2] = valA
    andl %esi, %esi # valA ≤ 0?
    jle Loop3 # if so, goto next loop
    iaddl $1, %eax # count++

```

Figure 12: partC loop

For "andl op1, op2 → JXX des".

When  $op1 == op2$ , everything is fine, the prediction accuracy will reach 100%. But we don't check whether  $op1 == op2$  in this attempt which will lead to errors for this case. When errors happen, we need to stall the pipeline (this is solved by setting D\_Bubble and E\_Bubble correctly) and retrieve the old branch destination address which is unsolvable because this information has been lost in the pipeline when JXX enters its WB stage.

### 3.1.2 Attempt 2

In this attempt, we still look at a particular code distribution, which is

## 3.2 Achievements

### 3.2.1 Attemp 3 Eventually Succeed

### 3.2.2 Performance Improvement

By modifying the pipeline logic, we managed to accelerate the program to a very surprising degree:

$$CPE = 12.98 \rightarrow CPE = 9.83 \rightarrow CPE = 8.95 \rightarrow CPE = 7.78$$

We even try to unlimit the number of the array size. By modifying benchmark.pl, we push the upper bound to be 400 to test the best performance of our implementation (see in Figure 13).

390	1953	5.01
391	1955	5.00
392	1963	5.01
393	1962	4.99
394	1973	5.01
395	1980	5.01
396	1980	5.00
397	1983	4.99
398	1991	5.00
399	1994	5.00
400	2002	5.00
Average CPE		5.55
Score	60.0/60.0	

Figure 13: partC larger scale test

From analysis, we could safely estimate that the theoretical minimum *CPE* should be around 5.0.

**So it's almost certain that the optimizations we've done under the existing ISA framework are close to extreme.**

### 3.2.3 Code readability

We put a lot of effort into the readability of the code for parts A.

- For functions that contain loops, we can always break it down into three logical regions: "loop", "test", and "return".
- For various registers, we always annotate its purpose with comments
- For code that's not so obvious, we always leave the comment showing its counterpart in the C function next to it.

We have also left understandable and sufficient comments in the header of the modified files in partB and partC, please open `ncopy.js`, `pipe-zzcc.hcl`, `SEQ-full.hcl` to read

## 3.3 Feelings

This is a very interesting and valuable project. Ziqi and I work together perfectly. When Ziqi achieved full score for partC in two hours, we started thinking about whether we could get CPE down to the limit of 5. We ended up spending two days talking, drawing, and writing code to test. But what was so exciting was that after two big failures, we finally got there in a very "weird" way. It should be noted, however, that there is an inherent problem with the simulator for the pipeline processor, otherwise our attempt 2

should have been successful and would not have required special means to implement it. This project allowed us to reap the valuable sense of accomplishment brought by not giving up. We are also very grateful to have been able to take this course and to be introduced to such a valuable project.