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

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

# 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 viewïijŇthis 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 refering to C language functions
- Always draw a picture of the stack to ensure the correctness of fetching a variable.

### 2.1.2 Code

**sum.ys**

```
# 518030910211 ZiqiZhao
# 518030910188 YiminZhao

# Set up stack
        .pos    0
        irmovl  stack,  %esp
        rrmovl  %esp,   %ebp
        pushl   %edx            # save %edx
        irmovl  ele1,   %eax
        pushl   %eax
        call    sum_list
        popl    %edx            # flatten the stack for ele1
```

```
        popl    %edx            # restore %edx
        halt

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

# sum_list func
sum_list:
        pushl   %ebp            # enter
        pushl   %ecx            # save %ecx
        rrmovl  %esp,   %ebp
        xorl    %eax,   %eax    # clear %eax
        mrmovl  12(%ebp),%edx   # get ls
        jmp     test
loop:
        mrmovl  (%edx), %ecx
        addl    %ecx,   %eax
        mrmovl  4(%edx),%edx
test:
        andl    %edx,   %edx
        jne     loop            # %edx != 0
return:
        rrmovl  %ebp,   %esp    # leave
        popl    %ecx
        popl    %ebp
        ret

# Stack
        .pos    0x400
stack:
```

**rsum.ys**

```
# 518030910211 ZiqiZhao
# 518030910188 YiminZhao

# Set up stack
    .pos    0
    irmovl  stack, %esp
    rrmovl  %esp, %ebp
    pushl   %edx
    irmovl  ele1, %eax
    pushl   %eax
    call    rsum_list
    popl    %edx           # eat ele1
    popl    %edx           # restore %edx
    halt

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

# rsum_list func
rsum_list:
    pushl   %ebp            # enter
    rrmovl  %esp, %ebp
    xorl    %eax, %eax
    mrmovl  8(%ebp), %edx   # get ls
    andl    %edx, %edx
    je      return          # ls == NULL
do:
    pushl   %ebx            # save %ebx
    mrmovl  (%edx), %ebx    # mov ls->val to %ebx
    mrmovl  4(%edx), %eax
    pushl   %eax            # push ls->next
    call    rsum_list
    addl    %ebx, %eax      # ret = val + ret
    popl    %edx            # eat para
    popl    %ebx            # restore %ebx
```

```
return:
    rrmovl  %ebp, %esp      # leave
    popl    %ebp
    ret


# Stack
    .pos    0x400
stack:
```

**copy.ys**

```
# 518030910211 ZiqiZhao
# 518030910188 Yimin Zhao

# Set up stack
    .pos    0
    irmovl  stack, %esp
    rrmovl  %esp, %ebp
    irmovl  $3, %eax
    pushl   %eax
    irmovl  src, %eax
    pushl   %eax
    irmovl  dest, %eax
    pushl   %eax
    call    copy_block
    halt
.align 4
# Source block
src:
    .long 0x00a
    .long 0x0b0
    .long 0xc00

# Destination block
dest:
    .long 0x111
    .long 0x222
    .long 0x333

copy_block:
    pushl   %ebp
    rrmovl  %esp, %ebp
    pushl   %ecx
    pushl   %edx
```

```
    pushl   %edi
    irmovl  $0, %eax          # %eax = result = 0
    mrmovl  16(%ebp), %ecx    # %ecx = len
    mrmovl  12(%ebp), %edx    # %edx = src
    mrmovl  8(%ebp), %edi     # %edi = dest
    jmp     while_loop

while_loop:
    andl    %ecx, %ecx        # check if %ecx == 0?
    jle     return            # if so, jump to "return"
    mrmovl  (%edx), %esi      # %esi = val = *src
    irmovl  $4, %ebx          # %ebx = 4
    addl    %ebx, %edx        # src++
    rmmovl  %esi, (%edi)      # *dest = val
    addl    %ebx, %edi        # dest++
    xorl    %esi, %eax
    irmovl  $-1, %ebx
    addl    %ebx, %ecx        # len--
    jmp     while_loop

return:
    popl    %edi
    popl    %edx
    popl    %ecx
    rrmovl  %ebp, %esp
    popl    %ebp
    ret
# Stack
    .pos    0x400
stack:
```

#### 2.1.3 Evaluation

**sum.ys**

```
../yas sum.ys
../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.ys

- 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 addresss is set to be $0x400$.

**rsum.ys**

```
../yas rsum.ys
../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.ys

- 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 addresss is set to be $0x400$.

**copy.ys**

```
../yas copy.ys
../yis copy.yo
Stopped in 61 steps at PC = 0x25.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%ebx:   0x00000000      0xffffffff
%esp:   0x00000000      0x000003f4
%ebp:   0x00000000      0x00000400
%esi:   0x00000000      0x00000c00

Changes to memory:
0x0034: 0x00000111      0x0000000a
0x0038: 0x00000222      0x000000b0
0x003c: 0x00000333      0x00000c00
0x03ec: 0x00000000      0x00000400
0x03f0: 0x00000000      0x00000025
0x03f4: 0x00000000      0x00000034
0x03f8: 0x00000000      0x00000028
0x03fc: 0x00000000      0x00000003
```

Figure 3: partA-copy.ys

- 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 addresss 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 circult 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 constand 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.]

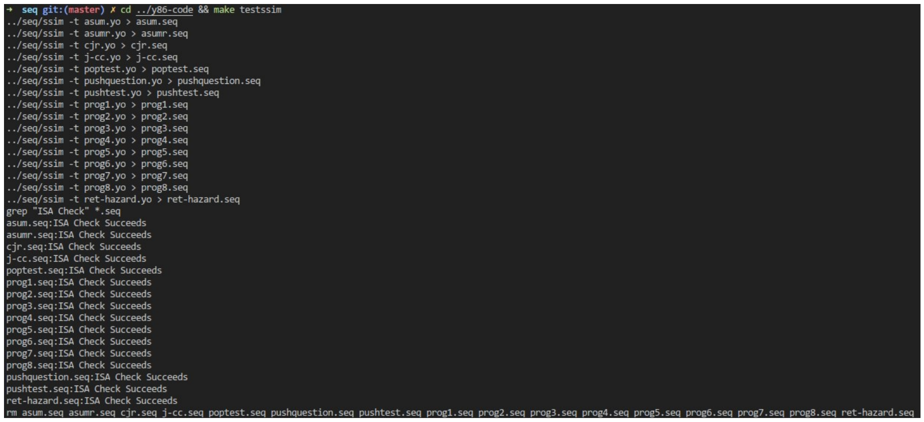
#### 2.2.2 Code

**Modifications in SEQ-full.hcl**

```
-------------------------------------------------------------------
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 };
-------------------------------------------------------------------
## What register should be used as the B source?
int srcB = [
        icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL  } : rB;
```

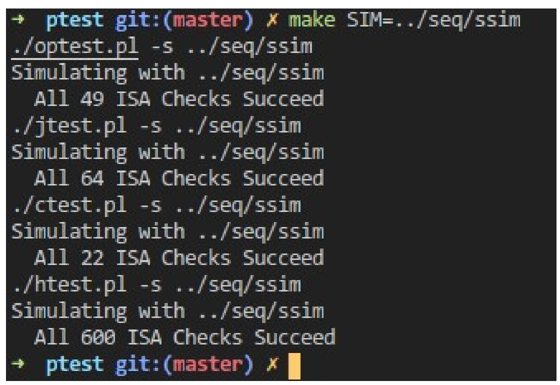
```
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register
];
-------------------------------------------------------------------
## What register should be used as the E destination?
int dstE = [
        icode in { IRRMOVL } && Cnd : rB;
        icode in { IIRMOVL, IOPL, IIADDL } : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't write any register
];
-------------------------------------------------------------------
## Select input A to ALU
int aluA = [
        icode in { IRRMOVL, IOPL } : valA;
        icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : valC;
        icode in { ICALL, IPUSHL } : -4;
        icode in { IRET, IPOPL } : 4;
        # Other instructions don't need ALU
];
-------------------------------------------------------------------
## Select input B to ALU
int aluB = [
        icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                IPUSHL, IRET, IPOPL, IIADDL } : valB;
        icode in { IRRMOVL, IIRMOVL } : 0;
        # Other instructions don't need ALU
];
-------------------------------------------------------------------
## Set the ALU function
int alufun = [
        icode == IOPL : ifun;
        icode == IIADDL : ALUADD;
        1 : ALUADD;
];
-------------------------------------------------------------------
## Should the condition codes be updated?
bool set_cc = icode in { IOPL, IIADDL };
-------------------------------------------------------------------
```

### 2.2.3 Evaluation



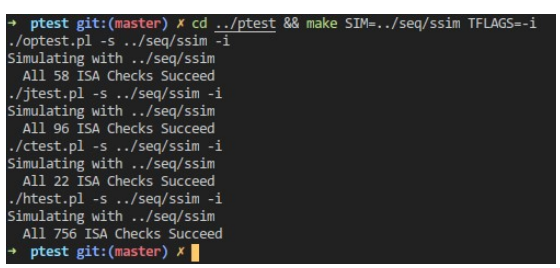
```
→  seq git:(master) ✗ cd ../y86-code && make testssim
../seq/ssim -t asum.yo > asum.seq
../seq/ssim -t asumr.yo > asumr.seq
../seq/ssim -t cjr.yo > cjr.seq
../seq/ssim -t j-cc.yo > j-cc.seq
../seq/ssim -t poptest.yo > poptest.seq
../seq/ssim -t pushquestion.yo > pushquestion.seq
../seq/ssim -t pushtest.yo > 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
asumr.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 asumr.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.ys as much as possible by modifying the ncopy.ys 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 "mrmovl" 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——**

```
# Entry
        iaddl $-9, %edx         # len -= 9, i.e., initial_len <= 9?
        irmovl $0, %eax         # count = 0
        jle Remaining           # if so, goto Remaining

# Loop unrolling part
Loop0:
        mrmovl (%ebx), %esi     # valA = src[0]
        mrmovl 4(%ebx), %edi    # valB = src[1]
        andl %esi, %esi         # valA <= 0?
        rmmovl %esi, (%ecx)     # dst[0] = valA
        jle Loop1               # if so, goto next loop
        iaddl $1, %eax          # count++
Loop1:
        mrmovl 8(%ebx), %esi    # valA = src[2]
        andl %edi, %edi         # valB <= 0?
        rmmovl %edi, 4(%ecx)    # dst[1] = valB
        jle Loop2               # if so, goto next loop
        iaddl $1, %eax          # count++
Loop2:
        mrmovl 12(%ebx), %edi   # valB = src[3]
        andl %esi, %esi         # valA <= 0?
        rmmovl %esi, 8(%ecx)    # dst[2] = valA
        jle Loop3               # if so, goto next loop
        iaddl $1, %eax          # count++
Loop3:
        mrmovl 16(%ebx), %esi   # valA = src[4]
        andl %edi, %edi         # valB <= 0?
        rmmovl %edi, 12(%ecx)   # dst[3] = valB
        jle Loop4               # if so, goto next loop
        iaddl $1, %eax          # count++
Loop4:
        mrmovl 20(%ebx), %edi   # valB = src[5]
        andl %esi, %esi         # valA <= 0?
        rmmovl %esi, 16(%ecx)   # dst[4] = valA
        jle Loop5               # if so, goto next loop
        iaddl $1, %eax          # count++
Loop5:
        mrmovl 24(%ebx), %esi   # valA = src[6]
        andl %edi, %edi         # valB <= 0?
        rmmovl %edi, 20(%ecx)   # dst[5] = valB
        jle Loop6               # if so, goto next loop
        iaddl $1, %eax          # count++
Loop6:
        mrmovl 28(%ebx), %edi   # valB = src[7]
```

```
        andl %esi, %esi         # valA <= 0?
        rmmovl %esi, 24(%ecx)   # dst[6] = valA
        jle Loop7               # if so, goto next loop
        iaddl $1, %eax          # count++
Loop7:
        mrmovl 32(%ebx), %esi   # valA = src[8]
        andl %edi, %edi         # valB <= 0?
        rmmovl %edi, 28(%ecx)   # dst[7] = valB
        jle Loop8               # if so, goto next loop
        iaddl $1, %eax          # count++
Loop8:
        mrmovl 36(%ebx), %edi   # valB = src[9]
        andl %esi, %esi         # valA <= 0?
        rmmovl %esi, 32(%ecx)   # dst[8] = valA
        jle Loop9               # if so, goto next loop
        iaddl $1, %eax          # count++
Loop9:
        andl %edi, %edi         # valB <= 0?
        rmmovl %edi, 36(%ecx)   # dst[9] = valB
        jle LoopEnd             # if so, goto loop end
        iaddl $1, %eax
LoopEnd:
        iaddl $40, %ecx         # dst += 10 * 4
        iaddl $40, %ebx         # src += 10 * 4
        iaddl $-10, %edx        # len -= 10
        jg Loop0                # if so, goto Loop0
                                # else, goto process remaining elements
```

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

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

```
        iaddl $-3, %edx         # [1,6] -> [-2,3]       (+6)
        jg RemTestRR
        je Rem6
RemTestRL:
        iaddl $1, %edx          # [-2,-1] -> [-1,0]     (+5)
        jl Rem4
        je Rem5
RemTestRR:
        iaddl $-2, %edx         # [1,3] -> [-1,1]       (+8)
        jl Rem7
        je Rem8
```

**——Unrolling of Remaining Loops——**

```
Rem9:
        mrmovl 32(%ebx), %esi   # valA = src[8]
        rmmovl %esi, 32(%ecx)   # dst[8] = valA
Rem8:   # Note that %esi == 0, directly jumping here
        #  implies that RemXb will performs correctly.
        andl %esi, %esi         # valA <= 0?
        mrmovl 28(%ebx), %esi   # valA = src[7]
        jle Rem8b               # if so, goto Rem8b
        iaddl $1, %eax          # count++
Rem8b:  rmmovl %esi, 28(%ecx)   # dst[7] = valA
Rem7:
        andl %esi, %esi         # valA <= 0?
        mrmovl 24(%ebx), %esi   # valA = src[6]
        jle Rem7b               # if so, goto Rem7b
        iaddl $1, %eax          # count++
Rem7b:  rmmovl %esi, 24(%ecx)   # dst[6] = valA
Rem6:
        andl %esi, %esi         # valA <= 0?
        mrmovl 20(%ebx), %esi   # valA = src[5]
        jle Rem6b               # if so, goto Rem6b
        iaddl $1, %eax          # count++
Rem6b:  rmmovl %esi, 20(%ecx)   # dst[5] = valA
Rem5:
        andl %esi, %esi         # valA <= 0?
        mrmovl 16(%ebx), %esi   # valA = src[4]
        jle Rem5b               # if so, goto Rem5b
        iaddl $1, %eax          # count++
Rem5b:  rmmovl %esi, 16(%ecx)   # dst[4] = valA
Rem4:
        andl %esi, %esi         # valA <= 0?
        mrmovl 12(%ebx), %esi   # valA = src[3]
```

```
        jle Rem4b               # if so, goto Rem4b
        iaddl $1, %eax          # count++
Rem4b:  rmmovl %esi, 12(%ecx)   # dst[3] = valA
Rem3:
        andl %esi, %esi         # valA <= 0?
        mrmovl 8(%ebx), %esi    # valA = src[2]
        jle Rem3b               # if so, goto Rem3b
        iaddl $1, %eax          # count++
Rem3b:  rmmovl %esi, 8(%ecx)    # dst[2] = valA
Rem2:
        andl %esi, %esi         # valA <= 0?
        mrmovl 4(%ebx), %esi    # valA = src[1]
        jle Rem2b               # if so, goto Rem2b
        iaddl $1, %eax          # count++
Rem2b:  rmmovl %esi, 4(%ecx)    # dst[1] = valA
Rem1:
        andl %esi, %esi         # valA <= 0?
        mrmovl (%ebx), %esi     # valA = src[0]
        jle Rem1b               # if so, goto Rem1b
        iaddl $1, %eax          # count++
Rem1b:
        andl %esi, %esi         # valA <= 0?
        rmmovl %esi, (%ecx)     # dst[0] = valA
        jle Done                # if so, goto Done
        iaddl $1, %eax          # count++
```

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

```
---------------------add following definition----------------------
quote 'int gen_aluA();'                 # Declaration of gen_aluA
quote 'int gen_aluB();'                 # Declaration of gen_aluB

# For JXX in ID and ALU in EX, check the cc generated by ALU
# Note that the simulator do not generate `cc_in` correctly
boolsig f_cnd_alu
        'cond_holds(compute_cc(id_ex_curr->ifun,
                        gen_aluA(), gen_aluB()),
                        if_id_next->ifun)'

# For JXX in ID and non-ALU in EX, check the cc register
boolsig f_cnd_other 'cond_holds(cc, if_id_next->ifun)'
------------------------modify f_predPC---------------------------
# Predict next value of PC
int f_predPC = [
        f_icode == ICALL : f_valC;
```

```
        f_icode == IJXX && f_ifun == UNCOND : f_valC;

        # Decode stage is ALU and will set CC -> always taken by default
        f_icode == IJXX && (D_icode in {IOPL, IIADDL}): f_valC;

        # Decode stage is not ALU
         # Execute stage is ALU -> compute CC
        f_icode == IJXX && (E_icode in {IOPL, IIADDL}) &&
        !f_cnd_alu : f_valP;

         # Execute stage is not ALU -> check cc -> ZF SF OF
        f_icode == IJXX && !(E_icode in {IOPL, IIADDL}) &&
        !f_cnd_other : f_valP;
        # Other JXX
        f_icode == IJXX : f_valC;

        # Otherwise
        1 : f_valP;
];
---------------add conditions to D_Bubble & E_BUbble----------------
bool D_bubble =
        # Mispredicted branch taken
        (E_icode == IJXX && E_ifun != UNCOND &&
        (M_icode in {IOPL, IIADDL}) && !e_Cnd) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVL, IPOPL }
        && E_dstM in { d_srcA, d_srcB })
        && IRET in { D_icode, E_icode, M_icode };

# Should I stall or inject a bubble into Pipeline Register E?
# At most one of these can be true.
bool E_stall = 0;
bool E_bubble =
        # Mispredicted branch taken
        (E_icode == IJXX && E_ifun != UNCOND &&
        (M_icode in {IOPL, IIADDL}) && !e_Cnd) ||
        # Conditions for a load/use hazard
        E_icode in { IMRMOVL, IPOPL } &&
         E_dstM in { d_srcA, d_srcB};
```

### 2.3.3 Evaluation

```
Ξ (bz-parallels) 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 com-

pletion 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 intruction: 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 t "andl rA, rA $\rightarrow$ JXX"(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:

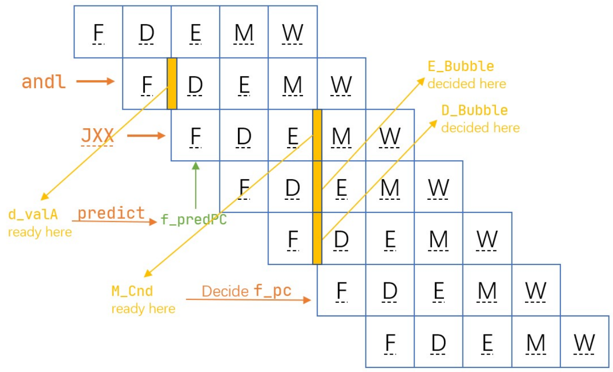
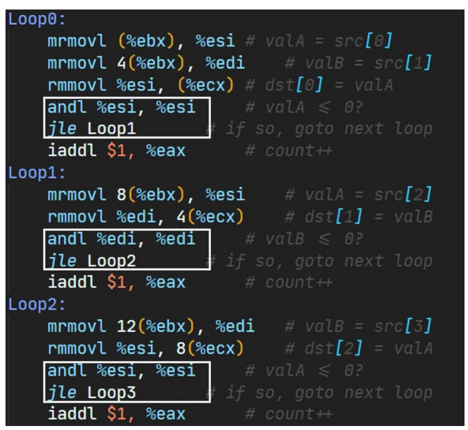



Figure 11: partC attempt1

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 "JXX op1, op2, destination".
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.

## 3.2 Achievements

[In this part you can list the strength of your project solution, like the performance improvement, coding readability, partner cooperation and so on. You can also write what you have learned if you like.]