Project 1: Optimizing the Performance of a Pipelined Processor

518030910211 Ziqi Zhao bugenzhao@sjtu.edu.cn
518030910188 Yimin Zhao doctormin@sjtu.edu.cn

May 3, 2020

1 Introduction

Part A
In part A, we write three simple assembly programs to implement 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 Y86's sequential design 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 dependency relations of `iaddl` to all boolean signals.
- Design the datapath for `iaddl`, i.e., 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 as much as possible, which leaves $CPE = 12.96$.
- Beyond the changes on instruction order, we apply 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 elements after several rounds of unrolling to achieve complete unrolling, which leaves $CPE = 8.95$.
- Modify the pipeline design in `HCL` file to achieve 100% accuracy in branch prediction for certain code pattern, which **brings CPE down to 7.79**.

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)

Special Notes for Testing the Part C

- In the Part C, we first achieved full marks by simply adding the `iaddl` instruction to `HCL` file, and in an almost software-only way. The hardware implementation of this approach is in `pipe-full.hcl`.
- After that, we made a further exploration and made a modification to the branch prediction part, and finally achieved a CPE of 7.79. **The hardware implementation of this approach is in `pipe-zzcc.hcl`.**

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-saved register: `EBX, EDI, ESI`.
- 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.ys

```
# 518030910211 ZiqiZhao
# 518030910188 YiminZhao

    .pos    0
main:
    irmovl  stack, %esp     # initialize stack
    rrmovl  %esp, %ebp      # initialize frame
    irmovl  ele1, %eax
    pushl   %eax            # argument
    call    sum_list
    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
    rrmovl  %esp, %ebp
    xorl    %eax, %eax      # clear %eax
    mrmovl  8(%ebp), %edx   # %edx = ls
    jmp     test
loop:
    mrmovl  (%edx), %ecx    # tmp = ls->val
    addl    %ecx, %eax      # val += tmp
    mrmovl  4(%edx), %edx   # ls = ls->next
test:
    andl    %edx, %edx      # ls == 0?
    jne     loop            # no -> loop
return:
    rrmovl  %ebp, %esp      # leave
    popl    %ebp
    ret


# Stack
    .pos    0x400
stack:
```

rsum.ys

```
# 518030910211 ZiqiZhao
# 518030910188 YiminZhao

# Set up stack
    .pos    0
    irmovl  stack, %esp
    rrmovl  %esp, %ebp
    irmovl  ele1, %eax
    pushl   %eax
    call    rsum_list
    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
    pushl   %ebx            # save %ebx
    xorl    %eax, %eax      # clear eax
    mrmovl  8(%ebp), %edx   # get ls
    andl    %edx, %edx      # ls == NULL?
    je      return          # yes -> return
do:
    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
return:
    popl    %ebx            # restore %ebx
    rrmovl  %ebp, %esp      # leave
    popl    %ebp
    ret


# Stack
    .pos    0x400
stack:
```

copy.ys

```
# 518030910211 ZiqiZhao
# 518030910188 YiminZhao

# Set up stack
    .pos    0
    irmovl  stack, %esp
    rrmovl  %esp, %ebp
    irmovl  $3, %eax        # len
    pushl   %eax
    irmovl  dest, %eax      # dest
    pushl   %eax
    irmovl  src, %eax       # src
    pushl   %eax
```

```
    call    copy_block
    halt

.align 4
# Source block
src:
    .long 0x00a
    .long 0x0b0
    .long 0xc00
    .long 0x888            # Should not copy this

# Destination block
dest:
    .long 0x111
    .long 0x222
    .long 0x333
    .long 0x999            # Should not write this

copy_block:
    pushl   %ebp           # enter
    rrmovl  %esp, %ebp
    pushl   %ebx           # save %ebx -> len
    pushl   %edi           # save %edi -> immediate
    pushl   %esi           # save %esi -> val
    xorl    %eax, %eax     # %eax = result = 0
    mrmovl  16(%ebp), %ebx # %ebx = len
    mrmovl  12(%ebp), %edx # %edx = dest
    mrmovl  8(%ebp), %ecx  # %ecx = src
    jmp     test
loop:
    irmovl  $4, %edi       # %edi = 4
    mrmovl  (%ecx), %esi   # val = *src
    addl    %edi, %ecx     # src += 1
    rmmovl  %esi, (%edx)   # *dest = val
    addl    %edi, %edx     # val += 1
    xorl    %esi, %eax     # result ^= val
    irmovl  $-1, %edi      # %edx = -1
    addl    %edi, %ebx     # len -= 1
test:
    andl    %ebx, %ebx     # len > 0?
    jg      loop           # yes -> loop
return:
    popl    %esi           # restore registers
    popl    %edi
    popl    %ebx
    rrmovl  %ebp, %esp     # leave
    popl    %ebp
    ret
```

```
# Stack
    .pos    0x400
stack:
```

2.1.3 Evaluation

- **sum.ys**

```
../yas sum.ys
../yis sum.yo
Stopped in 31 steps at PC = 0x15.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%ecx:   0x00000000      0x00000c00
%esp:   0x00000000      0x000003fc
%ebp:   0x00000000      0x00000400

Changes to memory:
0x03f4: 0x00000000      0x00000400
0x03f8: 0x00000000      0x00000015
0x03fc: 0x00000000      0x00000018
```

Figure 1: Part A: 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 68 steps at PC = 0x15.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%edx:   0x00000000      0x00000020
%esp:   0x00000000      0x000003fc
%ebp:   0x00000000      0x00000400

Changes to memory:
0x03c0: 0x00000000      0x00000c00
0x03c4: 0x00000000      0x000003d4
0x03c8: 0x00000000      0x00000058
0x03d0: 0x00000000      0x000000b0
0x03d4: 0x00000000      0x000003e4
0x03d8: 0x00000000      0x00000058
0x03dc: 0x00000000      0x00000028
0x03e0: 0x00000000      0x0000000a
0x03e4: 0x00000000      0x000003f4
0x03e8: 0x00000000      0x00000058
0x03ec: 0x00000000      0x00000020
0x03f4: 0x00000000      0x00000400
0x03f8: 0x00000000      0x00000015
0x03fc: 0x00000000      0x00000018
```

Figure 2: Part A: 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 58 steps at PC = 0x25.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%ecx:   0x00000000      0x00000034
%edx:   0x00000000      0x00000044
%esp:   0x00000000      0x000003f4
%ebp:   0x00000000      0x00000400

Changes to memory:
0x0038: 0x00000111      0x0000000a
0x003c: 0x00000222      0x000000b0
0x0040: 0x00000333      0x00000c00
0x03ec: 0x00000000      0x00000400
0x03f0: 0x00000000      0x00000025
0x03f4: 0x00000000      0x00000028
0x03f8: 0x00000000      0x00000038
0x03fc: 0x00000000      0x00000003
```

Figure 3: Part A: 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 `dest` 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`. The task is not so difficult once we understand the processing logic and the syntax of `HCL`, and all we need to do is change the followings in `seq-full.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 constant (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.

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

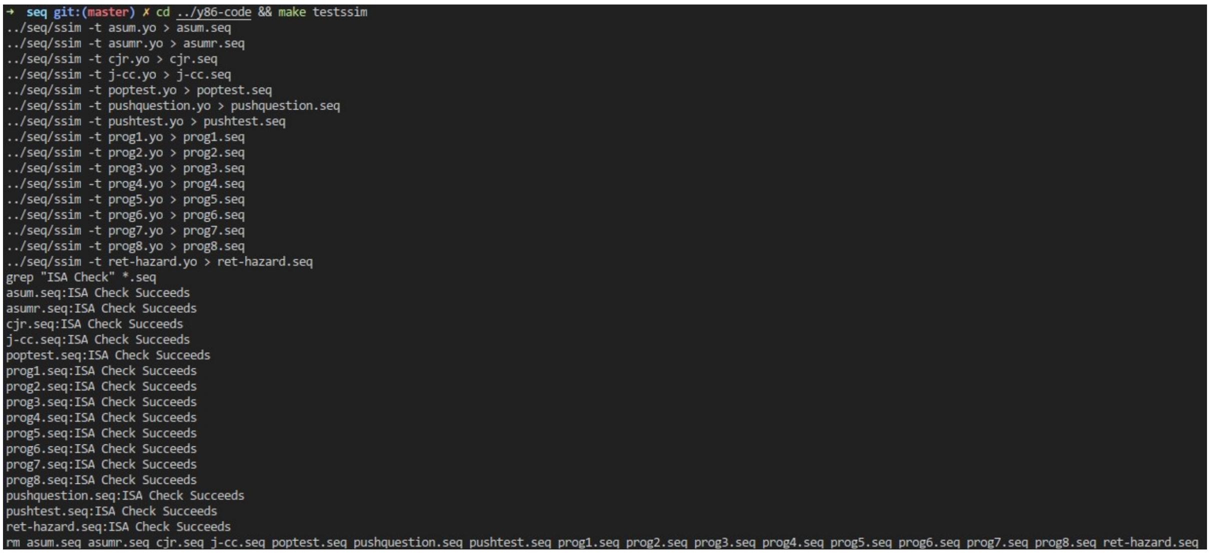

Figure 4: Part B: benchmark test

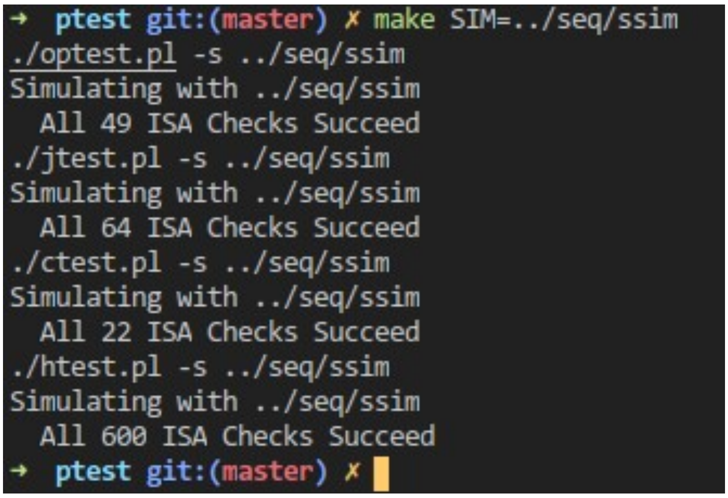

Figure 5: Part B: regression test

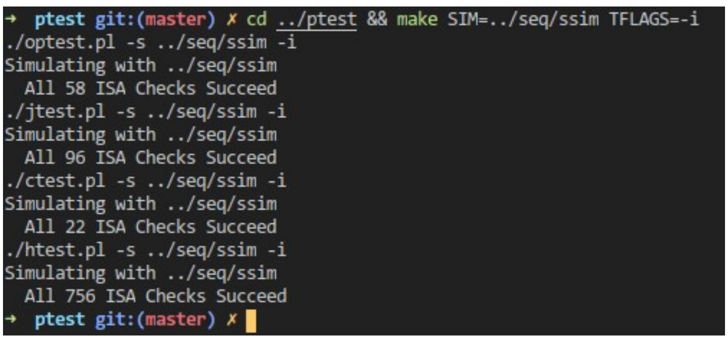

Figure 6: Part B: `iaddl` test

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:

Added `iaddl` instruction to `pipe-full.hcl`

Like what we have done in Part B, we added `iaddl` to the pipeline design. It avoided the overhead of using registers to store constants, and also increased the number of registers we could use, which paved the way for our further optimization.

Avoid Load and Use: CPE $\rightarrow$ 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: CPE $\rightarrow$ 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: CPE $\rightarrow$ 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 in `HCL` and other files of the Y86 pipeline design. And it finally got us here:

Optimized for Our Branch Prediction Design: CPE $\rightarrow$ 7.79

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 7.79. For more details of this part, **please refer to the code section or conclusion part**.

2.3.2 Code

——10-Way Loop Unrolling——

```
##################################################################
# You can modify this portion
# 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`)——

Note: here we have omitted the changes for adding `iaddl`,
which is the same as those in `seq-full.hcl` and `pipe-full.hcl`

```
# 1. Added instruction 'iaddl'
#   Similar to the changes in '../seq/full.hcl'.
#
# 2. Optimization on branch prediction
#   a. For unconditional JMP, there's no need to insert any bubble.
#
#   b. A significant performance factor is the branch prediction failure for count++. In our design, we made a special optimization for the situation like this:
#         Instruction:    any     non-alu     jxx
#         Stage:          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 7.79.
#
#      Specifically, we have modified the following logic:
#     - Added some additional definitions around line 150;
#     - Modified SelectPC logic around line 180;
#     - Modified PredictPC logic around line 230;
#     - Modified pipeline bubble logic around line 400 and 410.

```

```
---------------------Added following definition----------------------
intsig cc     'cc'                      # Condition register

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
# So we can only get `cc_in` through this method
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)'

--------------------------Modified f_PC-----------------------------
## What address should instruction be fetched at
int f_pc = [
        # Unconditional jump: Use predicted value of PC
        M_icode == IJXX && M_ifun == UNCOND : F_predPC;

        # Mispredicted taken. Fetch at incremented PC (previously valP)
        M_icode == IJXX && (W_icode in {IOPL, IIADDL}) && !M_Cnd : M_valA;

        # Completion of RET instruction.
        W_icode == IRET : W_valM;

        # Default: Use predicted value of PC
        1 : F_predPC;
];

------------------------Modified 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;
];

---------------Added 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 };


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

Note: both designs in `pipe-full.hcl` and `pipe-zzcc.hcl` have passed the tests and got full marks. Considering that the latter is a superset of the former, we have omitted the test screenshots of `pipe-full.hcl` here. The following results are the latter one's.

```
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: Part C: regression test (with `iaddl` included)

```
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: Part C: 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: Part C: 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: Part C: 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, while the second part made us familiar with Y86 SEQ circuit logic, and the third part encouraged us to transform assembly code and Y86 pipeline design. 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 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 search tree to find the number of remaining loops at the minimal cost and then completely unroll the operations for remaining elements.
  - We modify the `HCL` file of the pipeline to optimize the branch prediction, achieving 100% accuracy for a certain pattern (non-ALU followed by `jxx`).

3.1 Problems

We've only had some tricky problems in Part C. They are two unsuccessful attempts to modify the pipeline logic to lift the accuracy of branch prediction. Based on the second attempt, however, we have made some small changes to achieve the goal, which is explained in detail in **3.2 Achievements**.

3.1.1 Attempt 1

In this attempt, we look at a particular code distribution, which is "`andl op1, op2` $\rightarrow$ `jxx dest`". We hope that the branch prediction of `jxx` under this distribution can reach 100% accuracy.

The logic is shown in Figure 11:

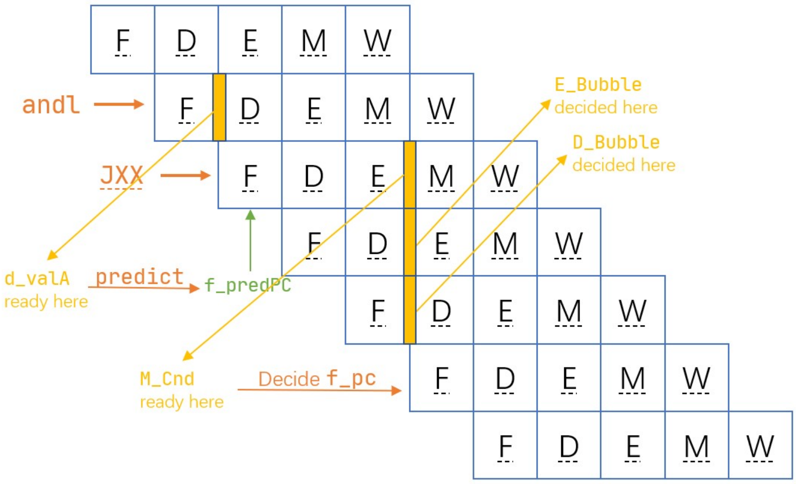

Figure 11: Part C: Attempt 1

Our initial plan is that, when we detect a `jxx` after an `andl`, we will do branch prediction according to the value of the first operand of `andl`. Since in our `ncopy.ys`, there are a lot of this code patterns:

```
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: Part C: Loop of Attempt 1

Let's take a closer look at "`andl op1, op2` → `jxx dest`".

When `op1 == op2`, i.e., `rA == rB`, the sign of `op1` is exactly the sign of `op1 & op2`, then we can compare the current conditional jump instruction type with the sign of `op1` to achieve a 100% accuracy.

But if `op1 != op2`, our prediction will not always succeed, which requires a restore from mispredicted branch, that is, we must find out an approach to determine whether we have mispredicted when the `jxx` is in MEM stage. However, this is unsolvable since the this information of two register sources has been lost in the pipeline when `andl` has entered its WB stage.

3.1.2 Attempt 2

In this attempt, we look at another code pattern, which is "`any instruction` → `non-alu instruction` → `jxx dest`", just like the following:

```
        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]
```

Figure 13: Part C: Attempt 2

Since the last instruction of `jxx` is a non-alu instruction, we are able to get the correct conditional flag when `jxx` is stll in the Fetch stage.

Specifically, if the instruction in Execute stage now is a non-alu one, then the current value of `CC` register will be the correct one for `jxx`. If the instruction in Execute stage now is an alu one, the the condition flags generated by ALU, i.e. in `cc_in`, will be the correct one.

We have ensured that if a non-alu instruction is followed by a `jxx`, the prediction must be correct. Thus, when we are processing with the logic of restoring form misprediction, we only need to focus on the case where the previous instruction of `jxx` is an alu instruction. According to this, we avoid the problems we encountered in our first attempt, since it is really easy to get the `ifun` field and decide the type of an instruction in Write-back stage.

However, when we finished the implementation in `HCL` file, **a strange problem arose**. The simulator using this branch prediction approach can pass all the ISA tests provided, but yielded a large number of "bad count" in `ncopy.ys`.

It took us about six hours to troubleshoot the problem and finally pin down the cause of the problem into two signals in the pipeline simulator: **`cc_in` and `e_valE`**. Please refer to the next section on how we finally solved this problem.

3.2 Achievements

3.2.1 A Successful Attempt 3

For the "bad count" problems, we explored for a long time and finally realized that when we predicted the branch, **the simulator may not generate the correct ALU output values**, say, `e_valE` and `cc_in`. After a talk with Chi Zhang, we adopted a compromise method, that is, manually call the C function in the simulator program to generate a correct ALU output, and take this as the basis of prediction. It finally succeeded and leaded to a CPE of 7.79.

```
# 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)'
```

In this case, we seem to predict the branch by doing complex operations in the Fetch stage, which also seems unreasonable in real pipelined processor design. However, according to the characteristics of the logic circuit, all signals in the circuit must be able to reach steady state in one clock cycle, so the output of ALU must be correct. **We don't need to add any additional complex hardware in the Fetch stage to perform such a prediction policy. We just need a simple unit instead to compare the `ifun` of the current `jxx` instruction with the three flags of `zf`, `sf` and `of` in `cc` or `cc_in`**, which is similar to the technique named "delayed slot" and is a fairly practical design in fact.

3.2.2 Performance Improvement

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

$$\text{CPE} = 12.98 \to \text{CPE} = 9.83 \to \text{CPE} = 8.95 \to \mathbf{CPE = 7.79}$$

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 Figure 14).

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 14: Part C: larger scale test

From analysis, we could safely estimate that the theoretical minimum CPE should be around 4.5 to 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 Part B and Part C. Please refer to `ncopy.ys`, `pipe-full.hcl`, `pipe-zzcc.hcl`, and `seq-full.hcl`.

3.3 Feelings

This is a very interesting and valuable project, and we have worked together perfectly. When Ziqi achieved full score for Part C in two hours, we started thinking about whether we could get CPE down to the limitation. We ended up spending two days talking, drawing, and writing code to test. But the most exciting thing 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 second attempt should have been successful and would not have required special means to implement it in the third attempt.

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.

Finally, we would like to specially thank Miss Shen and the TAs for their instruction and support, so that we can successfully complete this project.