Project 1: Optimizing the Performance of a Pipelined Processor

000, , bugenzhao@sjtu.edu.cn
001, , doctormin@sjtu.edu.cn

May 2, 2020

1 Introduction

Part A
In part A, we write three simple assembly programs to mimic three functions in example.c. Based on ensuring correctnesswe 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 viewthis 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 Techniques

- 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

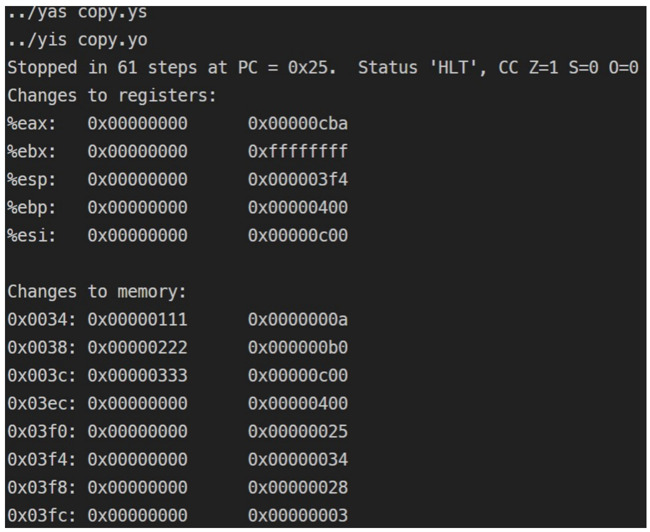
```
../yas rsum.ys
../yis rsum.yo
Stopped in 69 steps at PC = 0x1b.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%esp:   0x00000000      0x00000400
%ebp:   0x00000000      0x00000400

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

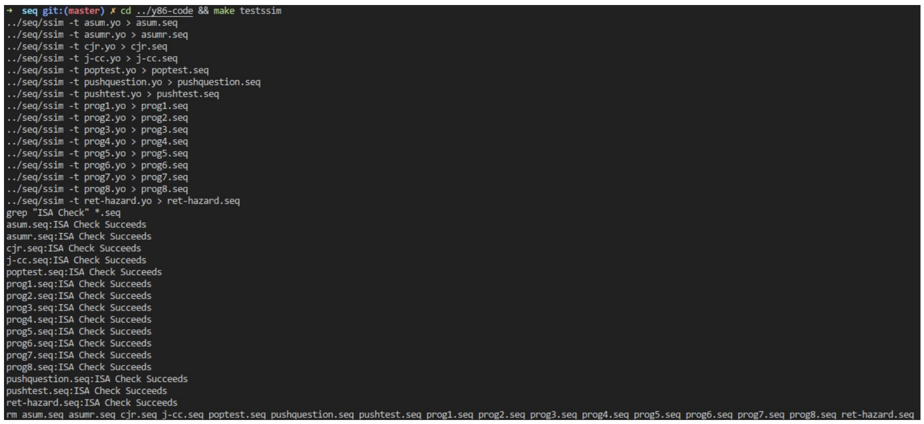


Figure 4: part B benchmark test

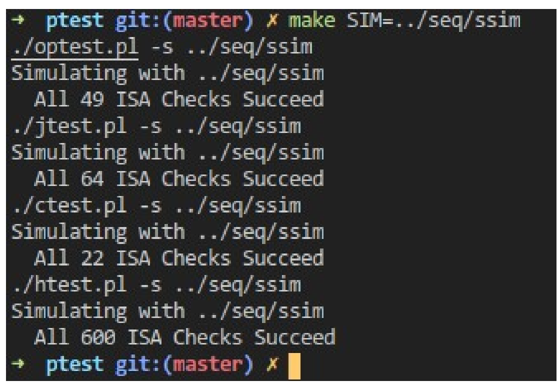


Figure 5: part B regression test

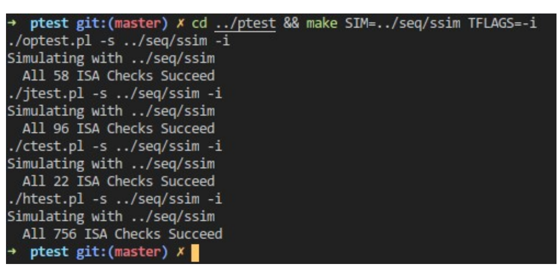


Figure 6: part B iaddl test

2.3 Part C

2.3.1 Analysis

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

2.3.2 Code

[In this part, you should place your code and make it readable in Microsoft Word, please. Writing necessary comments for codes is a good habit.]

2.3.3 Evaluation

[In this part, you should place the figures of experiments for your codes, prove the correctness and validate the performance with your own words for each figures explanation.]

3 Conclusion

3.1 Problems

[In this part you can list the obstacles you met during the project, and better add how you overcome them if you have made it.]

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.]