Project 1: Optimizing the Performance of a Pipelined Processor

521021910604, Xiaohan Qin, galaxy-1@sjtu.edu.cn
521021910299, Xiaozhi Zhu, zhuxiaozhi@sjtu.edu.cn
521021910308, Leqi Hu, huleqi0119@sjtu.edu.cn

April 26, 2023

1 Introduction

Part A

Part A involves writing three straightforward assembly programs to implement three functions in `example.c`. Our priority is functional equivalence with the `example.c` functions to ensure correctness. Additionally, we add comments in the assembly code to make it more readable.

Part B

For Part B, we are tasked with modifying the HCL file of Y86's sequential design to include a new instruction - `iaddl`. Thus, we analyze the flow of this instruction and then add the corresponding micro-instructions. As a bonus, we have also implemented the instruction `leave` and passed all the tests.

Part C

In Part C, we are asked to modify the file `ncopy.ys` and `pipe-full.hcl` to make `ncopy.ys` run as fast as possible. We rearrange the order of some instructions to solve the load/use hazard. We also use the new instruction `iaddl` and loop unrolling technology to achieve a better score. After that, we continue to make further improvements to the original code by using binary search trees and modifying the `pipe-full.hcl` to improve prediction. Finally, we achieve a **`CPE` of 7.62**. By deleting some unnecessary code, we can even achieve a **`CPE` of 7.18**. (Although just for test)

Contribution

- Xiaohan Qin: Part A (coding) & Part B (coding and designing) & Part C (coding and designing) & optimization & report writing
- Xiaozhi Zhu: Part A (coding) & Part B (coding and reviewing) & Part C (coding and reviewing) & report writing
- Leqi Hu: Part A (reviewing) & Part B (coding and testing) & Part C (coding and testing) & 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, namely `sum.ys`, `rsum.ys`, `copy.ys`.

To implement sum.ys, we use register %eax to store the answer and jne to implement a loop to iterate over the linked list.
For rsum.ys, we use the stack to store the returned value and implement a recursive call, making the logic of our assembly code align with the function in the example. As for copy.ys, two registers are used to store the address of the source and destination, and another to copy the data from source to destination.

2.1.2 Code

sum.ys

```
Main:
    irmovl ele1, %edx
    call Sum
    ret

# %edx is the head of the list
Sum:
    irmovl $0, %eax         # sum
    andl %edx, %edx         # check if null, set CC
    jmp Test
Loop:
    mrmovl (%edx), %ecx     # get head->data
    addl %ecx, %eax         # sum += head->data
    mrmovl 4(%edx), %edx    # head = head->next
    andl %edx, %edx         # set CC
Test:
    jne Loop
    ret
```

For sum.ys,we store the stack address in %esp and then call main. The main code is as above.

rsum.ys

```
main:
    irmovl ele1, %esi       # set head
    call rsum
    ret

rsum:
    xorl %eax, %eax         # int val = 0
    xorl %ecx, %ecx         # int rest = 0
    andl %esi, %esi         # check if ls==null
    je recursive_end
    mrmovl (%esi), %eax     # int val = ls->val
    pushl %eax              # push val
    mrmovl 4(%esi), %esi    # ls = ls->next
    call rsum
    popl %ecx               # pop val
recursive_end:
    addl %ecx, %eax         # return val + rest
    ret
```

For rsum.ys, we store the stack address in %esp, and then call main. The main code is as above.

copy.ys

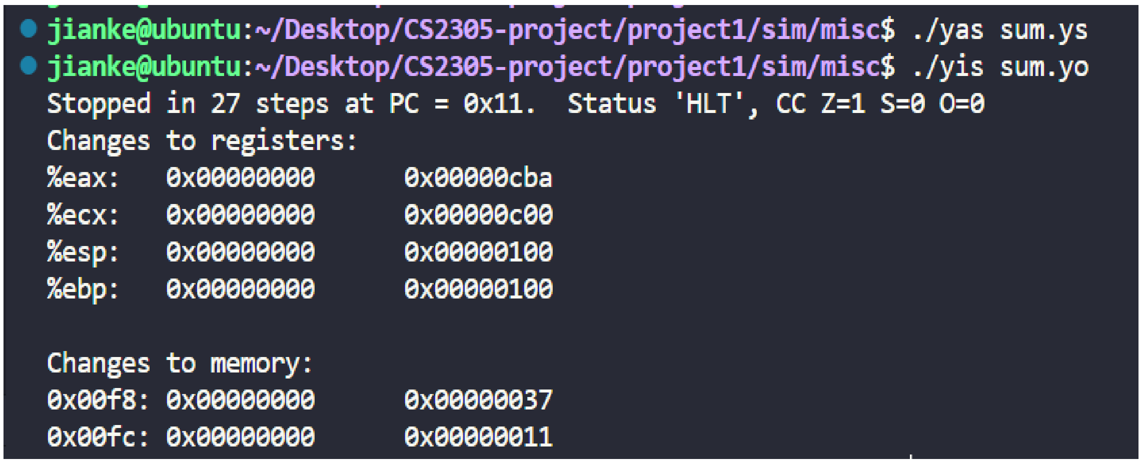
```
main:
    irmovl src , %edi       # src
    irmovl dest , %esi      # dest
    irmovl $3 , %edx        # len
    call copy
    ret

copy:
    xorl    %eax , %eax     # store the result
loop:
    andl    %edx , %edx     # if %edx == 0, jump to end
    jle     end
    mrmovl  (%edi) , %ecx   # value = *src
    irmovl  $4 , %ebx
    addl    %ebx , %edi     # src++
    rmmovl  %ecx , (%esi)   # *dest = value
    addl    %ebx , %esi     # dest++
    xorl    %ecx , %eax     # result ^= value
    irmovl  $1 , %ebx
    subl    %ebx , %edx     # len--
    jmp     loop
end:
    ret
```

For `copy.ys`, we store the stack address in `%esp`, and then call `main`. The main code is as above.

2.1.3 Evaluation

- **sum.ys**



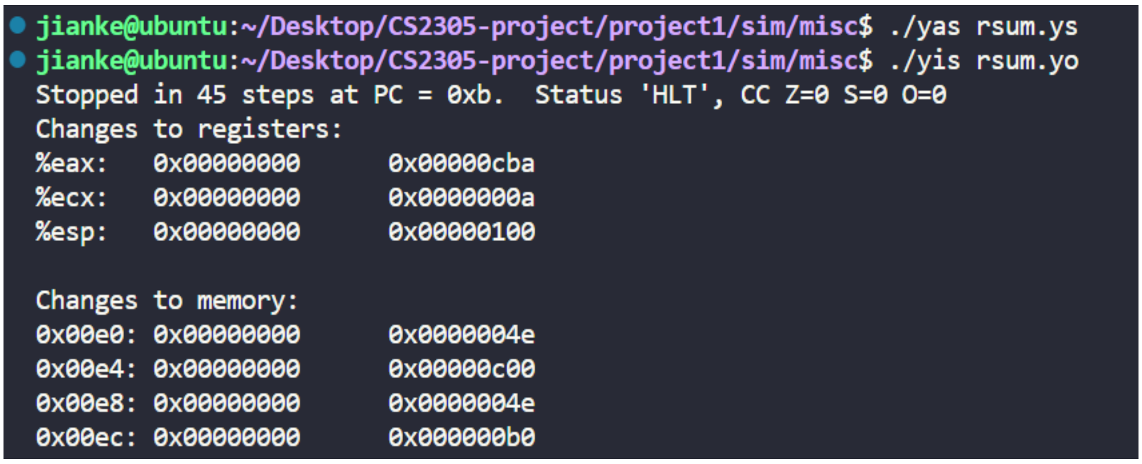
```
jianke@ubuntu:~/Desktop/CS2305-project/project1/sim/misc$ ./yas sum.ys
jianke@ubuntu:~/Desktop/CS2305-project/project1/sim/misc$ ./yis sum.yo
Stopped in 27 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%ecx:   0x00000000      0x00000c00
%esp:   0x00000000      0x00000100
%ebp:   0x00000000      0x00000100

Changes to memory:
0x00f8: 0x00000000      0x00000037
0x00fc: 0x00000000      0x00000011
```

Figure 1: sum.ys

The register `%ecx` has the correct answer `0xcba`.

- **rsum.ys**



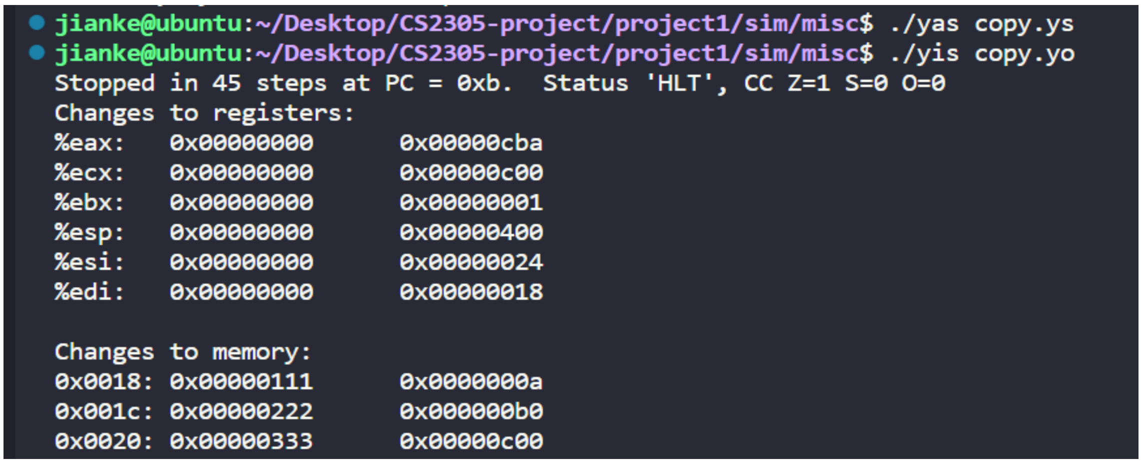
```
jianke@ubuntu:~/Desktop/CS2305-project/project1/sim/misc$ ./yas rsum.ys
jianke@ubuntu:~/Desktop/CS2305-project/project1/sim/misc$ ./yis rsum.yo
Stopped in 45 steps at PC = 0xb.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%ecx:   0x00000000      0x0000000a
%esp:   0x00000000      0x00000100

Changes to memory:
0x00e0: 0x00000000      0x0000004e
0x00e4: 0x00000000      0x00000c00
0x00e8: 0x00000000      0x0000004e
0x00ec: 0x00000000      0x000000b0
```

Figure 2: rsum.ys

The register `%eax` has the correct answer 0xcba.
The elements of the linked list are sumed recursively.

- **copy.ys**



```
jianke@ubuntu:~/Desktop/CS2305-project/project1/sim/misc$ ./yas copy.ys
jianke@ubuntu:~/Desktop/CS2305-project/project1/sim/misc$ ./yis copy.yo
Stopped in 45 steps at PC = 0xb.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%ecx:   0x00000000      0x00000c00
%ebx:   0x00000000      0x00000001
%esp:   0x00000000      0x00000400
%esi:   0x00000000      0x00000024
%edi:   0x00000000      0x00000018

Changes to memory:
0x0018: 0x00000111      0x0000000a
0x001c: 0x00000222      0x000000b0
0x0020: 0x00000333      0x00000c00
```

Figure 3: copy.ys

The register `%eax` has the correct answer 0xcba.
Values are written into the memory `0x18,0x1c,0x20` correctly.

2.2 Part B

2.2.1 Analysis

In part B, we extended the Y86-64 ISA to include a new instruction `iaddl` and implemented it in `pipe-full.hcl`. We also implemented `leave` as a bonus. The instruction is implemented in the following steps:

- Add `IIADDL` and `IIADDL` to the validation check in the `(bool) instr_valid` statement, since they are both valid instructions.
- Include `IIADDL` in the options of the `(bool) need_regid` section since `iaddl` operation requires one register.

- Include `IIADDL` in the `(bool) need_valC` section since `iaddl` operation requires one immediate value.
- Add `ILEAVE` to the selection code of `srcA`, when the `icode` is `ILEAVE`, `%ebp` must be chosen as `srcA`.
- Add `IIADDL` and `ILEAVE` to the selection code of `srcB` so that `srcB` takes the value from `rB` when the `icode` is `IIADDL` and takes the value from `%esp` when the `icode` is `ILEAVE`.
- Add `ILEAVE` to the selection code of `dstM`, when the `icode` is `ILEAVE`, `%ebp` must be chosen as `dstM`.
- Add `IIADDL` to the selection code of `dstE` so that `dstE` takes the value from `rB` when the `icode` is `IIADDL`.
- Add `IIADDL` and `ILEAVE` to the selection code of `aluA` so that `aluA` takes the value from `valC` when the `icode` is `IIADDL`, since `iaddl` operation requires one immediate value, and takes the value 4 when the `icode` is `ILEAVE`, since we need to store `%ebp+4` in `%esp`.
- Add `IIADDL` and `ILEAVE` to the selection code of `aluB` so that `aluB` takes the value from `valB` when the `icode` is `IIADDL` and takes the value from `valA` when the `icode` is `ILEAVE`.
- Set `mem_read` signal when the `icode` is `ILEAVE`.
- Select `valA` as memory address when the `icode` is `ILEAVE`.
- Include `IIADDL` in the `(bool) set_cc` section since `iaddl` operation may affect the condition code.

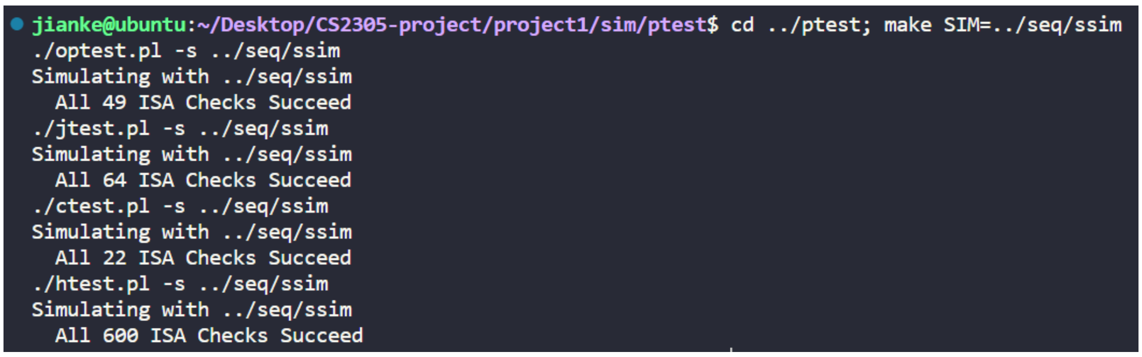
2.2.2 Code

In this part, we only paste the code modified when implementing `iaddl` and `leave`.

```
bool instr_valid = icode in
    { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
           IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL, ILEAVE };
--------------------------------------------------------------
bool need_regids =
    icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                 IIRMOVL, IRMMOVL, IMRMOVL, IIADDL };
--------------------------------------------------------------
bool need_valC =
    icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL };
--------------------------------------------------------------
int srcA = [
    icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : rA;
    icode in { IPOPL, IRET } : RESP;
    icode in { ILEAVE } : REBP;
    1 : RNONE; # Don't need register
];
--------------------------------------------------------------
int srcB = [
    icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL  } : rB;
    icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
    1 : RNONE;  # Don't need register
];
--------------------------------------------------------------
int dstE = [
    icode in { IRRMOVL } && Cnd : rB;
    icode in { IIRMOVL, IOPL, IIADDL } : rB;
```

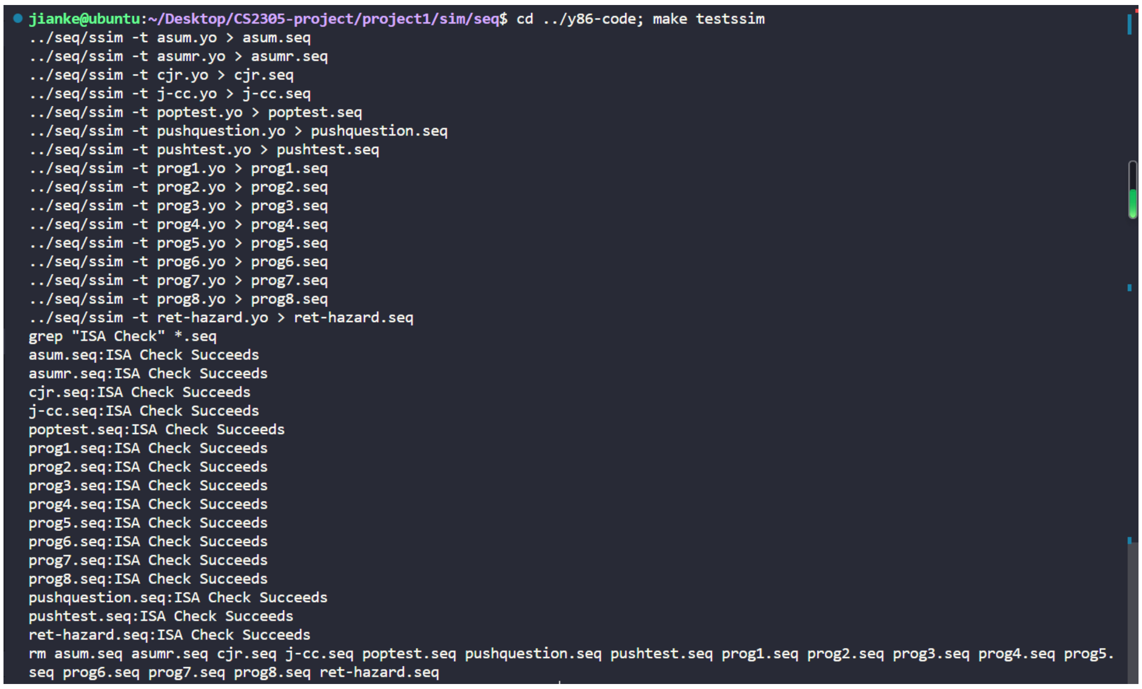
```
        icode in { IPUSHL, IPOPL, ICALL, IRET, ILEAVE } : RESP;
        1 : RNONE;  # Don't write any register
    ];
    ---------------------------------------------------------------
    int dstM = [
        icode in { IMRMOVL, IPOPL } : rA;
        icode in { ILEAVE } : REBP;
        1 : RNONE;  # Don't write any register
    ];
    ---------------------------------------------------------------
    int aluA = [
        icode in { IRRMOVL, IOPL } : valA;
        icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : valC;
        icode in { ICALL, IPUSHL } : -4;
        icode in { IRET, IPOPL, ILEAVE } : 4;
        # Other instructions don't need ALU
    ];
    ---------------------------------------------------------------
    int aluB = [
        icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                      IPUSHL, IRET, IPOPL, IIADDL } : valB;
        icode in { IRRMOVL, IIRMOVL } : 0;
        icode in { ILEAVE } : valA;
        # Other instructions don't need ALU
    ];
    ---------------------------------------------------------------
    bool mem_read = icode in { IMRMOVL, IPOPL, IRET, ILEAVE };
    ---------------------------------------------------------------
    int mem_addr = [
        icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
        icode in { IPOPL, IRET, ILEAVE } : valA;
        # Other instructions don't need address
    ];
    ---------------------------------------------------------------
    bool set_cc = icode in { IOPL, IIADDL };
```

2.2.3 Evaluation



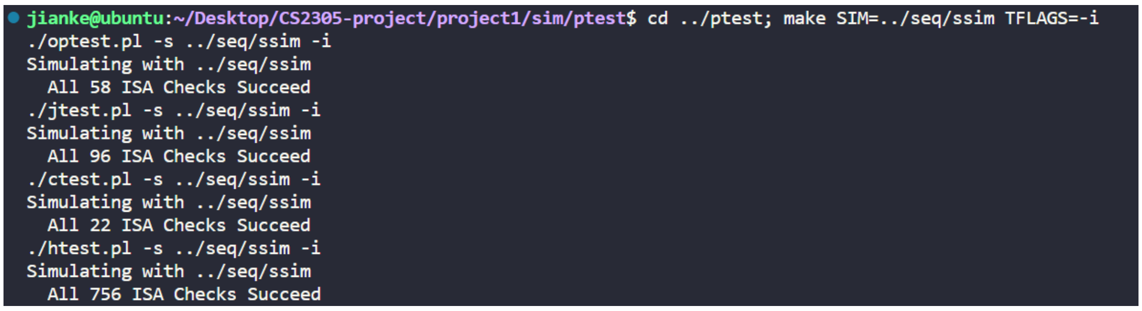
```
jianke@ubuntu:~/Desktop/CS2305-project/project1/sim/ptest$ cd ../ptest; make SIM=../seq/ssim
./optest.pl -s ../seq/ssim
Simulating with ../seq/ssim
  All 49 ISA Checks Succeed
./jtest.pl -s ../seq/ssim
Simulating with ../seq/ssim
  All 64 ISA Checks Succeed
./ctest.pl -s ../seq/ssim
Simulating with ../seq/ssim
  All 22 ISA Checks Succeed
./htest.pl -s ../seq/ssim
Simulating with ../seq/ssim
  All 600 ISA Checks Succeed
```

Figure 4: regression test for part B



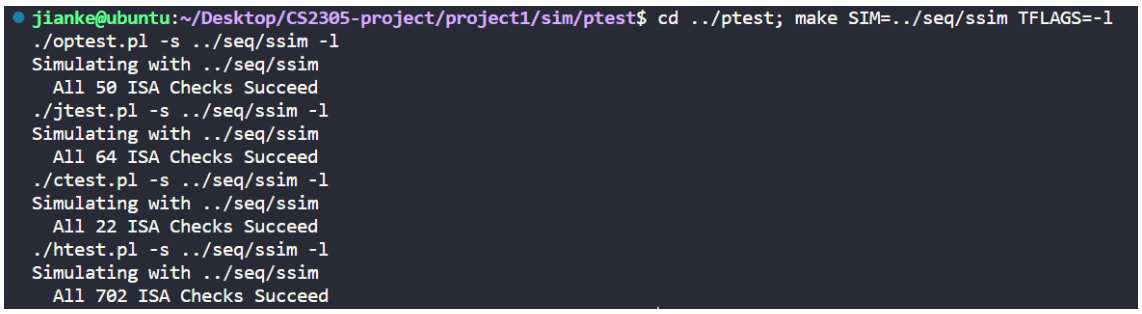
```
jianke@ubuntu:~/Desktop/CS2305-project/project1/sim/seq$ cd ../y86-code; make testssim
../seq/ssim -t asum.yo > asum.seq
../seq/ssim -t asumr.yo > asumr.seq
../seq/ssim -t cjr.yo > cjr.seq
../seq/ssim -t j-cc.yo > j-cc.seq
../seq/ssim -t poptest.yo > poptest.seq
../seq/ssim -t pushquestion.yo > pushquestion.seq
../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 5: benchmark result for part B



```
jianke@ubuntu:~/Desktop/CS2305-project/project1/sim/ptest$ 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
```

Figure 6: iaddl test for part B



```
jianke@ubuntu:~/Desktop/CS2305-project/project1/sim/ptest$ cd ../ptest; make SIM=../seq/ssim TFLAGS=-l
./optest.pl -s ../seq/ssim -l
Simulating with ../seq/ssim
  All 50 ISA Checks Succeed
./jtest.pl -s ../seq/ssim -l
Simulating with ../seq/ssim
  All 64 ISA Checks Succeed
./ctest.pl -s ../seq/ssim -l
Simulating with ../seq/ssim
  All 22 ISA Checks Succeed
./htest.pl -s ../seq/ssim -l
Simulating with ../seq/ssim
  All 702 ISA Checks Succeed
```

Figure 7: leave test for part B

2.3 Part C

2.3.1 Analysis

In this part, we are asked to modify the file `ncopy.ys` and `pipe-full.hcl` to make `ncopy.ys` run as fast as possible. We accomplished our final goal through the following stages:

Use `iaddl` and rearrange some instructions: CPE →12.96
Same as what we have done in `Part B`, we modified the `pipe-full.hcl` and added `iaddl` to the pipeline design. With `iaddl`, we can add a immediate number to a register without using extra registers to save temp variable and thus use less instructions.

8-way loop unrolling: CPE→10.62
In this program, most of the time overhead comes from judgment and jump instructions. Taking this into consideration, we decided to do multiple loops and update the relevant data at once, which reduced jump and assignment instructions. After some experiment, we chose to unroll 8 ways.

Solve load/use hazard: CPE→9.83
In the original file `ncopy.ys`, "load and use" or `mrmovl` then `rmmovl` will cause penalty, which must be avoided to improve the performance. We rearranged the instructions to avoid stalling as much as possible.

Use binary search tree: CPE→8.96
For small inputs, loop unrolling may work not so well. We can simply write another loop for these inputs, but there is a better method: use binary search tree to quickly find what the input is and just execute once instead of using loop. In this way, we get 8.96.

Modify the `HCL` to get a better prediction: CPE→7.62
For this program, the most important factor affecting CPE is the judgment and jump of `count++`, if we can improve the jump prediction accuracy, then we can highly improve our performance. After learning from the textbook and internet, we modified `pipe-full.hcl` and the order of some instructions, then we use **forwarding** to inform `JXX` whether a jump is required, which avoid the pipeline to stall.

(Experiment) delete unnecessary instructions: CPE→7.18
There are some instructions at the beginning and end that we cannot modify, but actually they are unnecessary. By deleting some `pushl` and `popl`, we can still get a correct result while **CPE is only 7.18**. But due to regulations, this is just our experiment.

2.3.2 Code

Due to page limit, we only show part of the core code, and the complete code can be viewed via the submitted file and my github

a sample block for loop unrolling

```
loop4:
    mrmovl 20(%ebx), %edi           # src[5]
    andl %esi, %esi                 # if src[4] <= 0, jump to loop5
    rmmovl %esi, 16(%ecx)           # dst[4] = src[4]
    jle loop5
    iaddl $1, %eax                  # else count++
########################################################################
# 8-way loop unrolling, so update by add 8 at once
update:
    iaddl $32, %ebx                 # src += 8
    iaddl $32, %ecx                 # dst += 8
    iaddl $-8, %edx                 # len -= 8
    mrmovl (%ebx), %esi             # src[0]
    jge loop0                       # if so, goto Loop0, else goto rest
```

In this block, we use `%esi` and `%edi` to interleave the data to be copied, so that we can avoid load/use hazard. We modify the order of `rmmovl %esi, 16(%ecx)` and `andl %esi, %esi`, so that the result of `andl` can be passed to `jle loop5` by using forwarding. After 8 loops, we update the source, destination and length.

binary search tree

```
rest:
    iaddl $5, %edx              # len += 5 {-3, -2, -1, 0, 1, 2, 3, 4}
    mrmovl (%ebx), %esi         # read val from src in advance
    jl searchleft               # {-3, -2, -1}
    je resloop3_1               # len = 3
    jmp searchright             # {1, 2, 3, 4}

searchleft:
    iaddl $2, %edx              # len += 2 { -1, 0, 1}
    mrmovl (%ebx), %esi         # read val from src in advance
    jl Done                     # len = 0
    je resloop1_1               # len = 1
    jmp resloop2_1              # len = 2

    iaddl $-2, %edx             # len -= 2 {-1, 0, 1, 2}
    mrmovl 4(%ebx), %edi        # read val from src in advance
    jl resloop4_1               # len = 4
    je resloop5_1               # len = 5
    jmp resloop6_1              # len = 6 or 7
```

In this block, we use binary search tree to quickly detect the rest number and use different `resloop` to respond to different situations.

modification in pipe-full.hcl

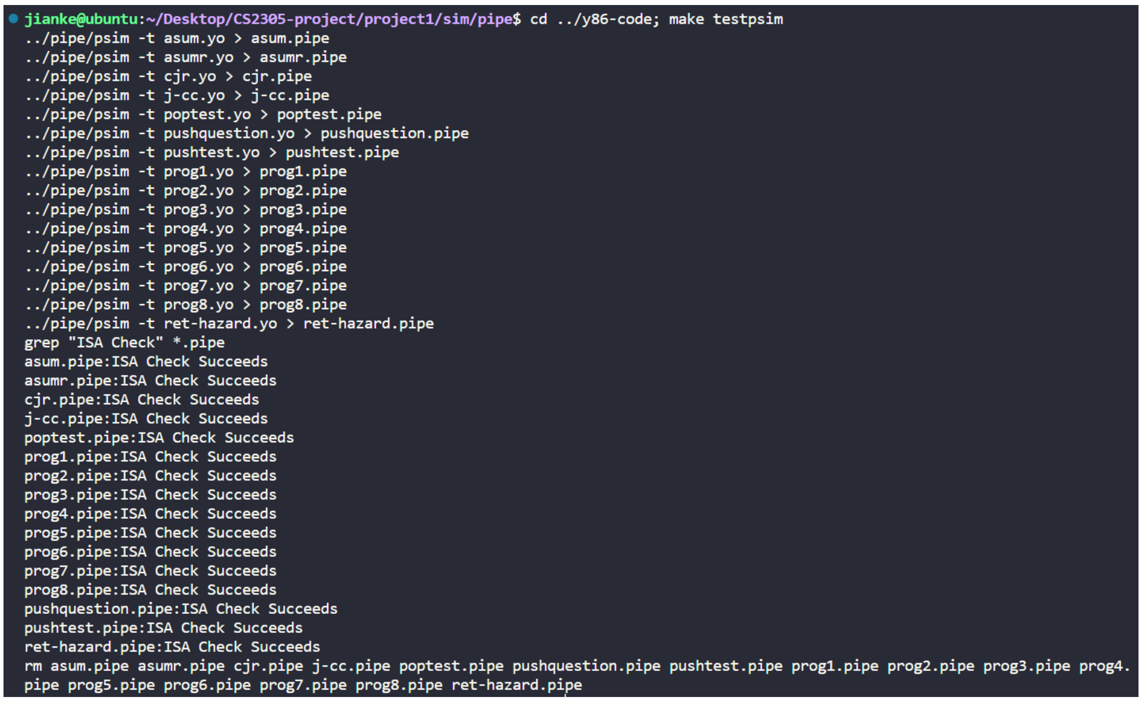
```
intsig JMP 'C_YES'
# ALU in EX, use forwardding to get cc
boolsig f_take 'cond_holds(compute_cc(id_ex_curr->ifun, gen_aluA(),
                gen_aluB()), gen_f_ifun())'
boolsig f_alucc 'cond_holds(cc, gen_f_ifun())'
########################################################################
int f_pc = [
        # Unconditional jump. Fetch at target address
        M_icode == IJXX && M_ifun == JMP : F_predPC;

        # Mispredicted branch.  Fetch at incremented PC
        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;
];
int f_predPC = [
        f_icode == ICALL : f_valC;
        # Unconditional jump.  Predict target address
        f_icode == IJXX && f_ifun == JMP : f_valC;
        # Important! if EX an DE are both ALU, we should use D's cc, missing this
        # will cause a bug
        f_icode == IJXX && (D_icode in {IOPL, IIADDL}): f_valC;
        # Conditional jump and ALU in EX, use forwarding to get cc
        f_icode == IJXX && (E_icode in {IOPL, IIADDL}) && !f_take : f_valP;
        # ALU is not in EX, get cc directly from ALU
        f_icode == IJXX && !(E_icode in {IOPL, IIADDL}) && !f_alucc: f_valP;
        # Other JXX, default take
        f_icode == IJXX : f_valC;
        1 : f_valP;
];
bool D_bubble =
        # Mispredicted branch
        (E_icode == IJXX && E_ifun != JMP && (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
        (E_icode == IJXX && E_ifun != JMP && (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};
```

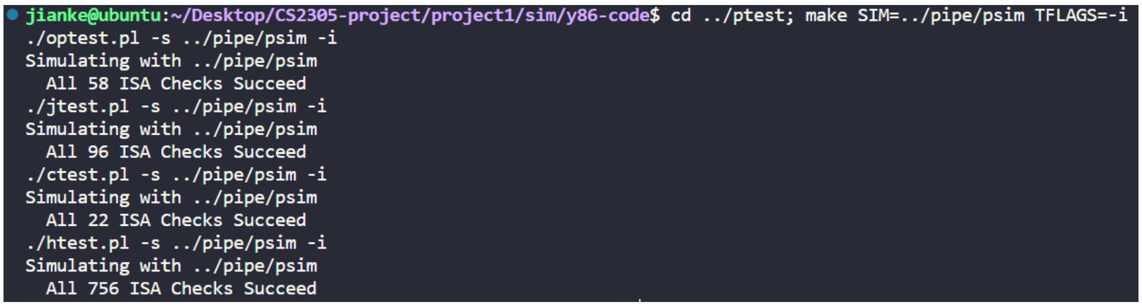
In this block, we modify the logic of prediction. If an ALU operation and JXX are not adjacent, we can compute the signal 'cc' and forward it to JXX, then JXX can directly konw whether to jump or not and needn't stall. (That's why we choose to modify the order of instructions in unrolling loops)

2.3.3 Evaluation



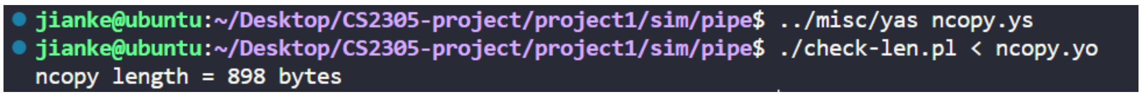
```
jianke@ubuntu:~/Desktop/CS2305-project/project1/sim/pipe$ cd ../y86-code; make testpsim
../pipe/psim -t asum.yo > asum.pipe
../pipe/psim -t asumr.yo > asumr.pipe
../pipe/psim -t cjr.yo > cjr.pipe
../pipe/psim -t j-cc.yo > j-cc.pipe
../pipe/psim -t poptest.yo > poptest.pipe
../pipe/psim -t pushquestion.yo > 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 pushtest.pipe prog1.pipe prog2.pipe prog3.pipe prog4.
pipe prog5.pipe prog6.pipe prog7.pipe prog8.pipe ret-hazard.pipe
```

Figure 8: benchmark result for psim



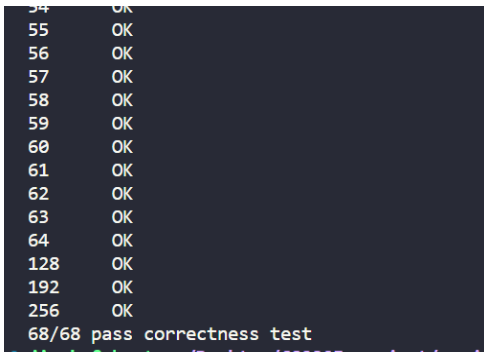
```
jianke@ubuntu:~/Desktop/CS2305-project/project1/sim/y86-code$ cd ../ptest; make SIM=../pipe/psim TFLAGS=-i
./optest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
  All 58 ISA Checks Succeed
./jtest.pl -s ../pipe/psim -i
Simulating with ../pipe/psim
  All 96 ISA Checks Succeed
./ctest.pl -s ../pipe/psim -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
```

Figure 9: regression result for psim



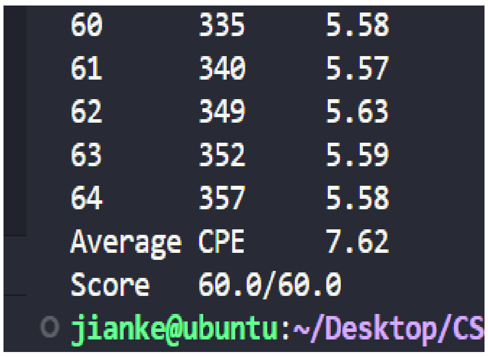
```
jianke@ubuntu:~/Desktop/CS2305-project/project1/sim/pipe$ ../misc/yas ncopy.ys
jianke@ubuntu:~/Desktop/CS2305-project/project1/sim/pipe$ ./check-len.pl < ncopy.yo
ncopy length = 898 bytes
```

Figure 10: length check for ncopy.ys



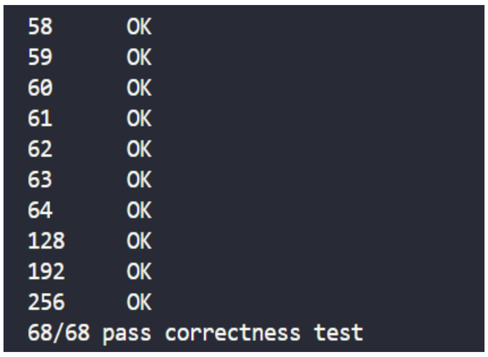
```
55      OK
56      OK
57      OK
58      OK
59      OK
60      OK
61      OK
62      OK
63      OK
64      OK
128     OK
192     OK
256     OK
68/68 pass correctness test
```

Figure 11: correctness check



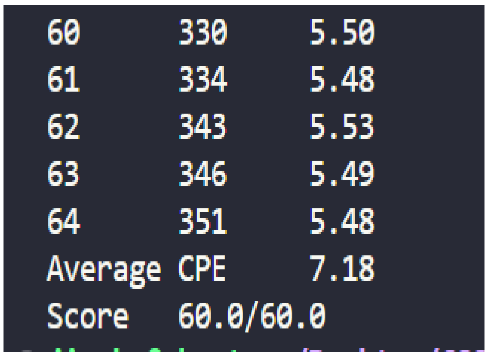
```
60      335     5.58
61      340     5.57
62      349     5.63
63      352     5.59
64      357     5.58
Average CPE     7.62
Score   60.0/60.0
jianke@ubuntu:~/Desktop/CS
```

Figure 12: final CPE



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

Figure 13: further test



```
60      330     5.50
61      334     5.48
62      343     5.53
63      346     5.49
64      351     5.48
Average CPE     7.18
Score   60.0/60.0
```

Figure 14: CPE for further test

3 Conclusion

3.1 Problems

Part A and part B are relatively easy, they made us familiar with Y86 assembly syntax and Y86 SEQ circuit logic. We didn't spend too much time on them.

For part C, it went well at first and we get the full score quickly. But when we tried to modify the logic of jump prediction, we met many problems and bugs.

- We were not familiar with the signal variables. To make good use of them, we read `sim.h`, `isa.h`, `pipeline.h` and `stages.h` carefully. Finally we got familiar with the various variables and API.
- The logic of `f_pred PC` is very complex, when we tried to modify it, there were a lot of wrong situations. We solved them one by one.
- At first we just modified the `f PC` and `f_pred PC`. After finished, the psim can pass the `correctness.pl` perfectly, but it will be failed when tested with benchmark in `y86-code`. We spent a lot time to solve this problem, and finally we realized that we need to modify the bubble logic as well.

3.2 Achievements

Our achievements can be summarized as follows:

- **Good code readability.** After we finished our code, we rewrited the comments carefully to have a better readability. In our project, the code logic can be clearly understood. For example, in part A each cycle is clearly demarcated, in part C the loop unrolling and binary search tree have a good structure for easy understanding.
- **Additional instruction.** In part B, after finished `iaddl`, we learnt more about the instruction `leave` and completed it by modifying the `seq-full.hcl` as a bonus function.
- **Performance improvement.** In part C, we got the full score 60/60 quickly with a CPE of 9.83. But we were not satisfied with this performance, so we spent a lot of time to optimize it. Finally we got a CPE of **7.62**. After removing unnecessary instructions, it can even down to **7.18**, which we thought to be our best performance.

3.3 Feelings

In this project, we learnt about the design and implementation of a pipelined Y86 processor, then optimized both it and the benchmark program to maximize performance. The first few tasks are relatively easy and we quickly finished the requirements. The difficulty lies in the process of our subsequent continuous thinking to improve performance. By solving these problems one by one, we not only obtained a perfect CPE, but also gained a deeper understanding of the various processes of the pipeline and the transfer of parameters between them.

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