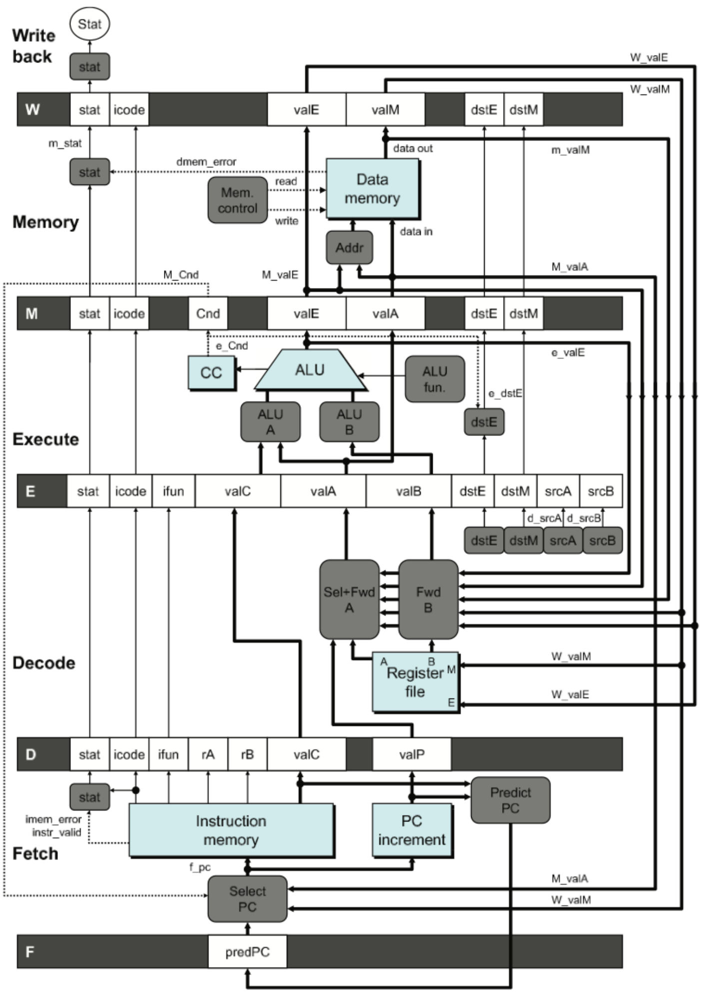
CS359-Project2-Report

Group Member:

1. Li Zhige 李止戈 5140219115
2. Lv Chengheng 吕呈恒 5140219046
3. Zhu Shunjia 朱舜佳 5140219105



Randal E. Bryant
David R. O'Hallaron

1. Objective:

- Become familiar with the Y86 tools and write simple Y86 programs.
- Extend the SEQ simulator with new instructions
- Learn about the design and implementation of a pipelined Y86 processor.
- Optimize the pipelined Y86 processor to maximize performance

2. Requirement:

1. In Part A, we are required to write and simulate three Y86 programs **- sum.us**, **rum.ys** and **copy.ys**
2. In Part B, we are required to add a new instruction named *iaddl* to the processor of which the function is to add a constant to register.
3. In Part C, we are required to modify the **ncopy.ys** and **pipe-full.hcl** to improve the performance of the pipelined Y86 processor.

3. Implementation

Part A

In order to become familiar with the Y86 tools, in this part we are going to write and simulate three Y86 programs: **sum.ys, rsum.ys** and **copy.ys**. The C version of these program have been provided in the **examples.c**, so our work is to tranlate them into Y86 instrction version. Actually these programs are somehow quite simple, however, it is our first time to deal with Y86 problem, we still need some time to be familiar with the Y86 instrutions.

sum.ys

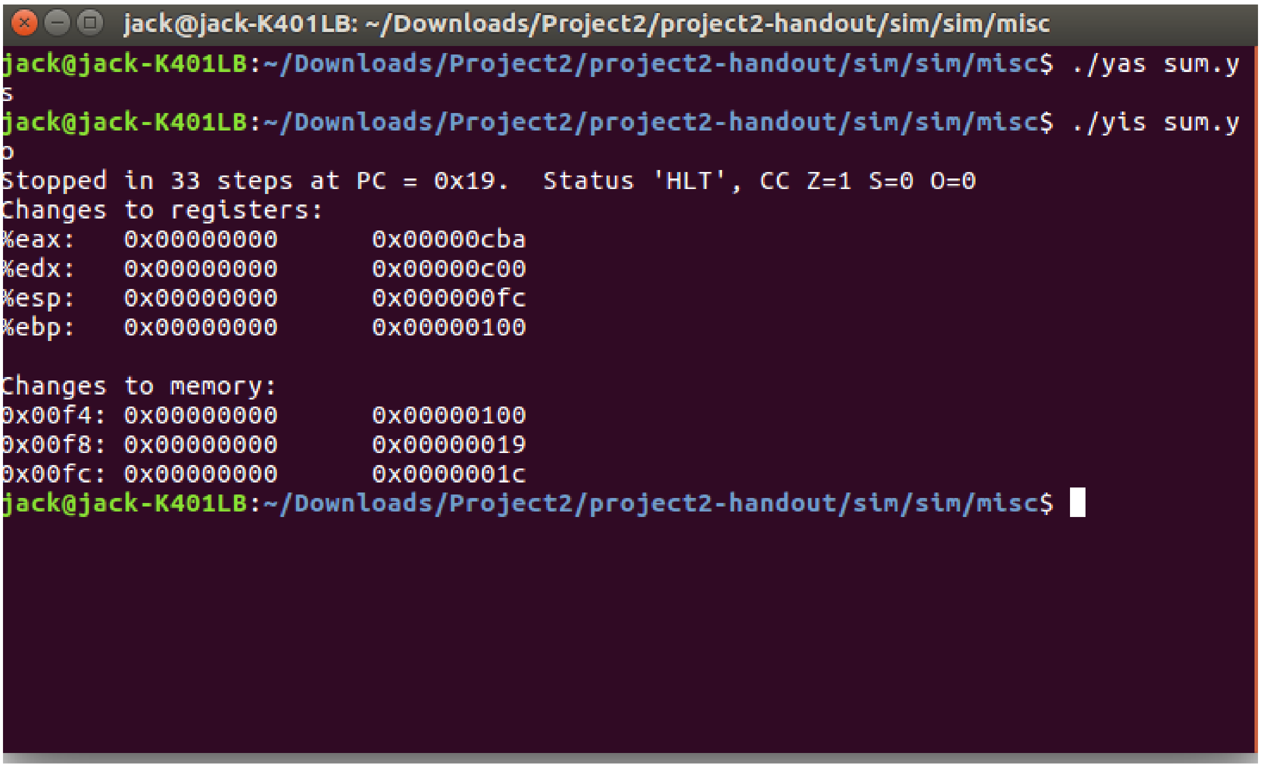
1. **code**

   The function of **sum.ys** is to iteratively sum the elements of a linked list. Here is the code of it.

```
# Student Name: Li Zhige
# Student ID: 5140219115

        .pos    0x0
init:   irmovl  Stack, %esp
        irmovl  Stack, %ebp
        irmovl  ele1, %eax
        pushl   %eax
        call    sum_list
        halt
.align 4
ele1:
        .long   0x00a
        .long   ele2
ele2:
        .long   0x0b0
        .long   ele3
ele3:
        .long   0xc00
        .long   0

sum_list:
        pushl   %ebp
        rrmovl  %esp, %ebp
        mrmovl  8(%ebp), %ecx
        xorl    %eax, %eax
Loop:
        andl     %ecx, %ecx    #if there is no next element, jump to end
        je      End
        mrmovl  (%ecx), %edx   #get the value of element
        addl    %edx, %eax     #add the value and get sum
        mrmovl  4(%ecx), %ecx  #get the next element
        jmp     Loop
End:
        rrmovl  %ebp, %esp
        popl    %ebp
        ret

        .pos    0x100
Stack:
```

After loading the pointer of first element in list to %ecx, it checks %ecx in every loop. If it was null, directly jump to End, else it will load the value of this element and add it to %eax which store the result of sum. And load the pointer of the next element to %ecx then run loop again. After adding all the elements' value, it will end and pop the initial %ebp.

1. **test result**



We can see from the picture that the result is *0x00000cba* which is exact the sum of three text elements. Also the value of %edx *0x00000c00* is the value of the last element in list.

rsum.ys

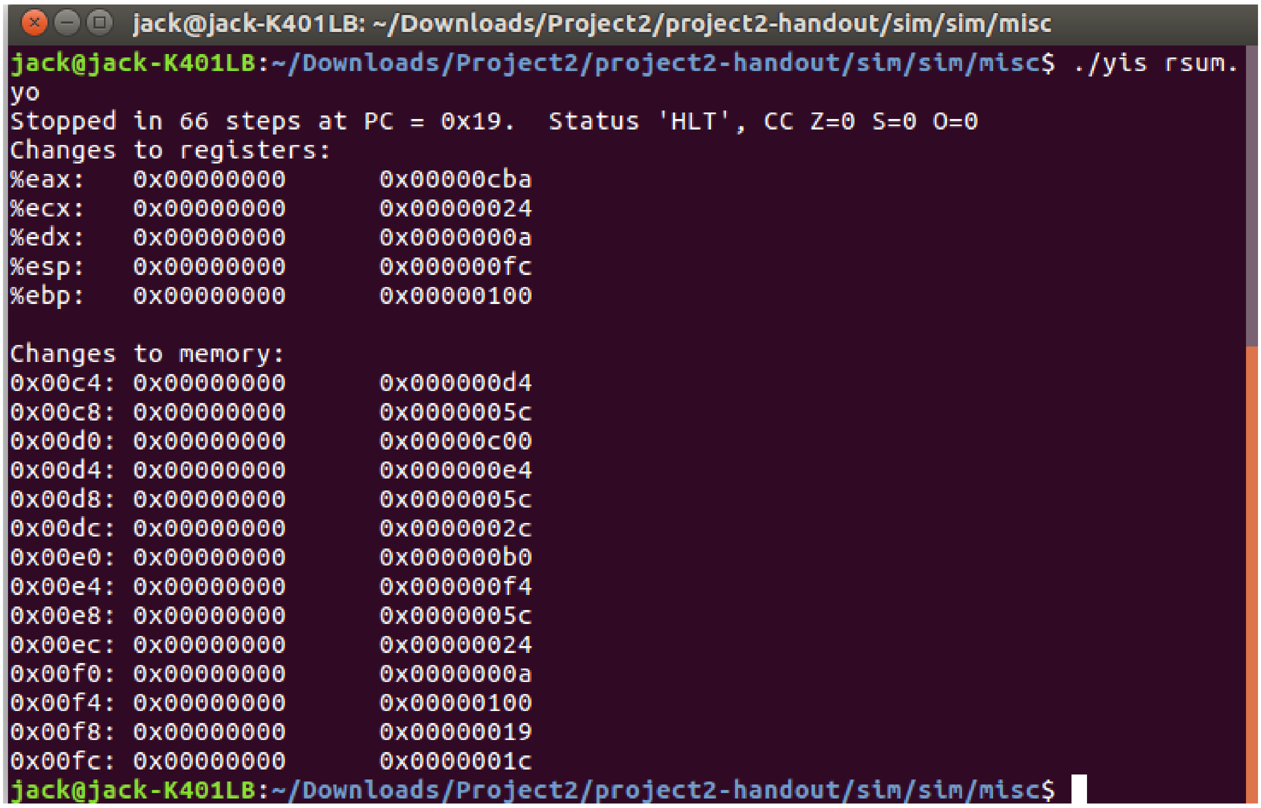
1. **code**

The function of **rsum.ys** is to recursively sum the elements of a linked list. Here is the code of it. This code is similar to the code in sum.ys, except that it should use a function rsum list that recursively sums a list of numbers.Here is the code.

```
# Student Name: Li Zhige
# Student ID: 5140219115
        .pos 0
init:   irmovl Stack, %esp
        irmovl Stack, %ebp
        irmovl ele1, %eax
        pushl  %eax
        call   rsum_list

        halt

.align 4
ele1:
     .long 0x00a
     .long ele2
ele2:
        .long 0x0b0
     .long ele3
ele3:
     .long 0xc00
     .long 0
rsum_list:
     pushl %ebp
     rrmovl %esp,%ebp
     mrmovl 8(%ebp),%ecx
     xorl %eax,%eax          #initial sum to 0
     andl %ecx,%ecx          #if there is no next element, jump to end
     je  End
     mrmovl (%ecx),%edx        #get the value
     pushl %edx               #push value into stack
     mrmovl 4(%ecx),%ecx       #get the next element
     pushl %ecx
     call rsum_list
     popl %ecx
     popl %edx
     addl %edx,%eax          #pop value and add them
End:
     rrmovl %ebp, %esp
     popl   %ebp
     ret
     .pos 0x100
Stack:
```

The begging and ending part of this program is just like what is in **sum.ys**. But in the main body, it store the value and push it to stack then call rsum_list again and again until the pointer point to an empty address then it will return and pop the value and add them all up.

2. **test result**



```
jack@jack-K401LB: ~/Downloads/Project2/project2-handout/sim/sim/misc
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/misc$ ./yis rsum.
yo
Stopped in 66 steps at PC = 0x19.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%ecx:   0x00000000      0x00000024
%edx:   0x00000000      0x0000000a
%esp:   0x00000000      0x000000fc
%ebp:   0x00000000      0x00000100

Changes to memory:
0x00c4: 0x00000000      0x000000d4
0x00c8: 0x00000000      0x0000005c
0x00d0: 0x00000000      0x00000c00
0x00d4: 0x00000000      0x000000e4
0x00d8: 0x00000000      0x0000005c
0x00dc: 0x00000000      0x0000002c
0x00e0: 0x00000000      0x000000b0
0x00e4: 0x00000000      0x000000f4
0x00e8: 0x00000000      0x0000005c
0x00ec: 0x00000000      0x00000024
0x00f0: 0x00000000      0x0000000a
0x00f4: 0x00000000      0x00000100
0x00f8: 0x00000000      0x00000019
0x00fc: 0x00000000      0x0000001c
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/misc$
```

The result is similar to that of **sum.ys** since actually they have the same function. But we can see that changes to memory are more than that of the above result, we can understand it since obviously recursive algorithm needs more memory space. And the value of %edx became *0x0000000a* now because in the recursive algorithm, the first element was added at the last.

copy.ys

1. **code**

   copy.ys is to copy a block of words from one part of memory to another (nonoverlapping area) area of memory, and computing the checksum (Xor) of all the words copied. Here is the code.

```
# Student Name: Li Zhige
# Student ID: 5140219115

        .pos 0
init:   irmovl  Stack, %esp
        irmovl  Stack, %ebp
        irmovl  $3, %eax
        pushl   %eax
        irmovl  dest, %eax
        pushl   %eax
        irmovl  src, %eax
        pushl   %eax
        call copy_block

        halt

.align  4
src:
        .long   0x00a
        .long   0x0b0
        .long   0xc00

dest:
        .long   0x111
        .long   0x222
        .long   0x333


copy_block:
        pushl   %ebp
        rrmovl  %esp, %ebp
        mrmovl  0x8(%ebp), %ebx     #get src
        mrmovl  0xc(%ebp), %ecx     #get dest
        mrmovl  0x10(%ebp), %edx    #get length
        xorl    %eax, %eax          #initial eax to 0

Loop:
        andl    %edx, %edx          #if length == 0, jump to end
        je      End
        mrmovl  (%ebx), %esi        #get the value from src
        rmmovl  %esi, (%ecx)        #store it to dest
        xorl    %esi, %eax          #operate xor and store the result in eax

        irmovl  $4, %edi
        addl    %edi, %ebx          #get the next src
        addl    %edi, %ecx          #get the next dest
        irmovl  $-1, %edi
        addl    %edi, %edx          #minus length

        jmp     Loop
```

```
    End:
            rrmovl  %ebp, %esp
            popl    %ebp
            ret


            .pos    0x100
    Stack:
```

In this program %ebx store the src, %ecx store the dest, %edx store the length and %eax store the result. In every loop, we check if the length is 0, if it is , directly jump to end, if not we get the value from src and store it to dest. Also, we operate the xor operation and store the result in %eax. After adding the address of src and dest by 4 and reduce the length by 1, we jump to another loop.

1. **test result**

```
jack@jack-K401LB: ~/Downloads/Project2/project2-handout/sim/sim/misc
ys
Exiting
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/misc$ ./yas copy.
ys
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/misc$ ./yis copy.
yo
Stopped in 54 steps at PC = 0x29.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%ecx:   0x00000000      0x00000044
%ebx:   0x00000000      0x00000038
%esp:   0x00000000      0x000000f4
%ebp:   0x00000000      0x00000100
%esi:   0x00000000      0x00000c00
%edi:   0x00000000      0xffffffff

Changes to memory:
0x0038: 0x00000111      0x0000000a
0x003c: 0x00000222      0x000000b0
0x0040: 0x00000333      0x00000c00
0x00ec: 0x00000000      0x00000100
0x00f0: 0x00000000      0x00000029
0x00f4: 0x00000000      0x0000002c
0x00f8: 0x00000000      0x00000038
0x00fc: 0x00000000      0x00000003
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/misc$
```

From the changes of memory we can see the dest had been changed. Since the specialness of the value of elements in src, xor operation actually get the sum, so the result of %eax is *0x00000cba*. %esi store the last element of src and edi is -1 in the end.

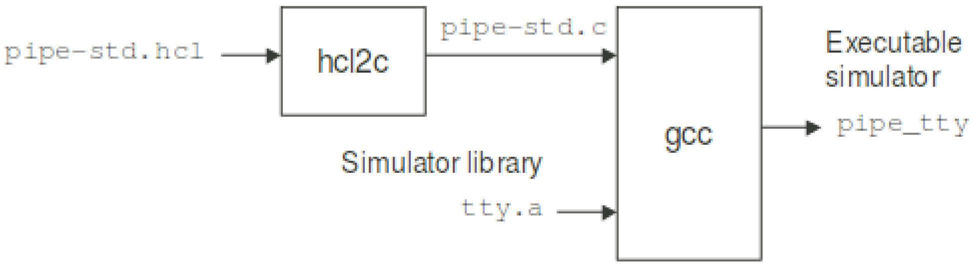
Part B

Introduction

The main funcion of this part is to extend the SEQ processor to support a new instruction *iaddl*. The main method is to first use the irmovl instruction to set the register to constant and then use *addl* instruction to add the value.

Instruction Set

CL has some of the features of a hardware description language (HDL), allowing users to describe Boolean functions and word-level selection operations. Its struction looks like this:



The main grammar looks like this:

- signal declaration:

$$boolsig \quad name \quad 'C-expr'$$

$$intsig \quad name \quad 'C-expr'$$

These instructions declare the signal names.

- Quoted Text

Quoted text provides a mechanism to pass text directly through HCL to C file. It goes like this:

Syntax	Meaning
`0`	Logic value 0
`1`	Logic value 1
$name$	Named Boolean signal
$int\text{-}expr$ `in` $\{int\text{-}expr_1, int\text{-}expr_2, \ldots, int\text{-}expr_k\}$	Set membership test
$int\text{-}expr_1$ `==` $int\text{-}expr_2$	Equality test
$int\text{-}expr_1$ `!=` $int\text{-}expr_2$	Not equal test
$int\text{-}expr_1$ `<` $int\text{-}expr_2$	Less than test
$int\text{-}expr_1$ `<=` $int\text{-}expr_2$	Less than or equal test
$int\text{-}expr_1$ `>` $int\text{-}expr_2$	Greater than test
$int\text{-}expr_1$ `>=` $int\text{-}expr_2$	Greater than or equal test
`!` $bool\text{-}expr$	NOT
$bool\text{-}expr_1$ `&&` $bool\text{-}expr_2$	AND
$bool\text{-}expr_1$ `||` $bool\text{-}expr_2$	OR

- Expressions and Blocks

Most of the definitions go like this:

$$
\begin{array}{lcl}
[& & \\
\quad bool\text{-}expr_1 & : & int\text{-}expr_1 \\
\quad bool\text{-}expr_2 & : & int\text{-}expr_2 \\
 & \vdots & \\
\quad bool\text{-}expr_k & : & int\text{-}expr_k \\
] & &
\end{array}
$$

The expression contains a series of cases. Each case i consists of a boolean expression $bool - expr_i$, indicating whether this case should be selected. If selected, it has the value of $int - expr_i$,.

Detail

In order to add the instruction, we need to modify the file **seq-full.hcl**. First, we should simply add the instruction we want to the instruction set. The instructions we need to add is *ileave* and *iiaddl*. I'll show it next. The red background means the old code version that we need to delete. The green background means the new code version that we need to add.

- First both o f the instructions are valid instructions:

```
bool instr_valid = icode in
bool instr_valid = icode in
    { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
     IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
     IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, ILEAVE, IIADDL };
```

- Next, because the *IIADDL* operates on registers and has ALU output so it needs a require a regid byte:

```
bool need_regids =
    icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
          IIRMOVL, IRMMOVL, IMRMOVL };
    icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
          IIRMOVL, IRMMOVL, IMRMOVL , IIADDL};
```

- Also, *IIADDL* operates on constant word:

```
bool need_valC =
    icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
    icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL };
```

- The *IIADDL* only operates on source B. But *ILEAVE* can operate both source A and source B. In this function, it needs to give the value of REBP(base register).

```
    icode in { ILEAVE } : REBP;
    1 : RNONE; # Don't need register
];

## What register should be used as the B source?
int srcB = [
    icode in { IOPL, IRMMOVL, IMRMOVL } : rB;
    icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL } : rB;
    icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
    icode in { ILEAVE } : REBP;
    1 : RNONE; # Don't need register
```

- The E destination means the execution destination. For *IIADDL*, it should return the B register. For *ILEAVE* instruction, it should return the stack pointer.

```
int dstE = [
    icode in { IRRMOVL } && Cnd : rB;
    icode in { IIRMOVL, IOPL} : rB;
    icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
    icode in { IIRMOVL, IOPL, IIADDL} : rB;
    icode in { IPUSHL, IPOPL, ICALL, IRET, ILEAVE } : RESP;
    1 : RNONE; # Don't write any register
];
```

- The M destination means the memeory destination. For *IIADDL*, it doesn't operate the memory. But *ILEAVE* can opearte on base pointer.

```
int dstM = [
    icode in { IMRMOVL, IPOPL } : rA;
    icode in { ILEAVE } : REBP;
    1 : RNONE; # Don't write any register
];
```

- For alu calculation, the *IIADDL* operates on a register in a constant value. But for *ILEAVE* it only needs to return the 4 to add PC +4. For b egister, value doesn't change , we just need to add the instruction.

```
int aluA = [
    icode in { IRRMOVL, IOPL } : valA;
    icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
    icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : valC;
    icode in { ICALL, IPUSHL } : -4;
    icode in { IRET, IPOPL } : 4;
    icode in { IRET, IPOPL, ILEAVE } : 4;
    # Other instructions don't need ALU
];
```

```
int aluB = [
	icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
		IPUSHL, IRET, IPOPL } : valB;
	icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
		IPUSHL, IRET, IPOPL, IIADDL, ILEAVE } : valB;
	icode in { IRRMOVL, IIRMOVL } : 0;
	# Other instructions don't need ALU
];
```

- The *IIADDL* operation should update the condition codes to keep the condition dynamic.

```
## Should the condition codes be updated?
bool set_cc = icode in { IOPL };
bool set_cc = icode in { IOPL, IIADDL };
```

- The *ILEAVE* operates the memory adress by jump to adress.

```
int mem_addr = [
	icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
	icode in { IPOPL, IRET } : valA;
	icode in { IPOPL, IRET, ILEAVE } : valA;
	# Other instructions don't need address
];
```

Result

We follow the order of the pdf to test our solutions. It goes like this:

- Building a new simulator:

```
make VERSION=full
```

- Running simple program **asumi.yo**

```
./ssim -t ./y86-code/asumi.yo
```

Result:

jack@jack-K401LB: ~/Downloads/Project2/project2-handout/sim/sim/seq

```
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/seq$ ./ssim -t ..
/y86-code/asumi.yo
Y86 Processor: seq-full.hcl
112 bytes of code read
IF: Fetched irmovl at 0x0.  ra=----, rb=%esp, valC = 0x100
IF: Fetched irmovl at 0x6.  ra=----, rb=%ebp, valC = 0x100
IF: Fetched jmp at 0xc.  ra=----, rb=----, valC = 0x24
IF: Fetched irmovl at 0x24.  ra=----, rb=%eax, valC = 0x4
IF: Fetched pushl at 0x2a.  ra=%eax, rb=----, valC = 0x0
Wrote 0x4 to address 0xfc
IF: Fetched irmovl at 0x2c.  ra=----, rb=%edx, valC = 0x14
IF: Fetched pushl at 0x32.  ra=%edx, rb=----, valC = 0x0
Wrote 0x14 to address 0xf8
IF: Fetched call at 0x34.  ra=----, rb=----, valC = 0x3a
Wrote 0x39 to address 0xf4
IF: Fetched pushl at 0x3a.  ra=%ebp, rb=----, valC = 0x0
Wrote 0x100 to address 0xf0
IF: Fetched rrmovl at 0x3c.  ra=%esp, rb=%ebp, valC = 0x0
IF: Fetched mrmovl at 0x3e.  ra=%ecx, rb=%ebp, valC = 0x8
IF: Fetched mrmovl at 0x44.  ra=%edx, rb=%ebp, valC = 0xc
IF: Fetched irmovl at 0x4a.  ra=----, rb=%eax, valC = 0x0
IF: Fetched andl at 0x50.  ra=%edx, rb=%edx, valC = 0x0
```

jack@jack-K401LB: ~/Downloads/Project2/project2-handout/sim/sim/seq

```
IF: Fetched mrmovl at 0x57.  ra=%esi, rb=%ecx, valC = 0x0
IF: Fetched addl at 0x5d.  ra=%esi, rb=%eax, valC = 0x0
IF: Fetched iaddl at 0x5f.  ra=----, rb=%ecx, valC = 0x4
IF: Fetched iaddl at 0x65.  ra=----, rb=%edx, valC = 0xffffffff
IF: Fetched jne at 0x6b.  ra=----, rb=----, valC = 0x57
IF: Fetched popl at 0x70.  ra=%ebp, rb=----, valC = 0x0
IF: Fetched ret at 0x72.  ra=----, rb=----, valC = 0x0
IF: Fetched halt at 0x39.  ra=----, rb=----, valC = 0x0
38 instructions executed
Status = HLT
Condition Codes: Z=1 S=0 O=0
Changed Register State:
%eax:   0x00000000      0x0000abcd
%ecx:   0x00000000      0x00000024
%esp:   0x00000000      0x000000f8
%ebp:   0x00000000      0x00000100
%esi:   0x00000000      0x0000a000
Changed Memory State:
0x00f0: 0x00000000      0x00000100
0x00f4: 0x00000000      0x00000039
0x00f8: 0x00000000      0x00000014
0x00fc: 0x00000000      0x00000004
ISA Check Succeeds
```

- Retesting solutions using the benchemark programs

```
jack@jack-K401LB: ~/Downloads/Project2/project2-handout/sim/sim/y86-code
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/seq$ cd ../y86-co
de/
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/y86-code$ make te
stssim
../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
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/y86-code$
```

- Performing regression tests.

```
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/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
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/ptest$
```

```
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/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
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/ptest$
```

Conclusion

As we had achieved above, tests run successfully.

Part C

In this part, we need to optimize the pipeline and some specific codes. In **ncopy.ys**, we use instruction *iaddl*, use the loop unrolling technique and use a jump table to speed up the Branch Operation.In **pipe-full.hcl**, we implemented load forwarding, a better jump prediction and the instructions *iaddl* and *leave*.

The details of code are blow:

1. In **ncopy.ys**, our jump table looks like:

```
#jump table
.align 4
    .long J1
    .long J2
    .long J3
    .long J4
    .long J5
    .long J6
    .long J7
    .long J8
    .long J9
    .long J10
    .long J11
    .long J12
    .long J13
    .long J14
    .long J15
    .long J16
Table:
```

We can jump to the n-th jump label according to this tabel. There are 16 jump labels from *J1* to *J16*:

```
......
J2:
    andl %edi, %edi
    mrmovl 0(%ebx), %edi
    rmmovl %edi, 0(%ecx)
    jle J1
    iaddl $1, %eax
J1:
    andl %edi, %edi
    jle Final
    iaddl $1, %eax
Final:
    popl %edi               # Restore callee-save registers
    popl %ebx
    leave
    ret
```

When the number of elements is big, we unroll the loop in the size of 16. Then fewer comparisions are made at the running time.

```
# You can modify this portion
    # Loop header
    xorl %eax,%eax      # count = 0;
    xorl %edi, %edi
    iaddl $-16, %edx
    jl Rest

Loop:
    andl %edi, %edi
    mrmovl 60(%ebx), %edi
    rmmovl %edi, 60(%ecx)
    jle L16
    iaddl $1, %eax
L16:
    andl %edi, %edi
    mrmovl 56(%ebx), %edi
    rmmovl %edi, 56(%ecx)
    jle L15
    iaddl $1, %eax
L15:
    andl %edi, %edi
    mrmovl 52(%ebx), %edi
    rmmovl %edi, 52(%ecx)
    jle L14
    iaddl $1, %eax
    ......
```

2. In **pipe-full.hcl**, we implement a lot of optimization like

   First, we add the instruction *IIADDL* and the instruction *ILEAVE* to the project. The modify is showed in part B.

Then in order to achieve better performance and pipline running, we need to change the file to adapt to the pipline instruction running. (The red background means the old code version that we need to delete. The green background means the new code version that we need to add.)

- f_PC: This set is used to decide what address should instruction be fetched at.

```
int f_pc = [
    # Mispredicted branch. Fetch at incremented PC
    M_icode == IJXX && !M_Cnd : M_valA;
    M_icode == IMRMOVL && !M_Cnd && E_icode == IRMMOVL && D_icode == IJXX &&
D_ifun == CLE : D_valP;
    M_icode == IJXX && !M_Cnd && !(W_icode == IRMMOVL && M_ifun == CLE):
M_valA;
    # Completion of RET instruction.
    W_icode == IRET : W_valM;
    # Default: Use predicted value of PC
    1 : F_predPC;
];
```

- f_predPC: This set is used to predict next value of PC.

```
int f_predPC = [
    f_icode in { IJXX, ICALL } : f_valC;
    f_icode == ICALL
    || f_icode == IJXX : f_valC;
    1 : f_valP;
];
```

- e_valAL: This set is used to achieve the forwarding function. Generate valA in execute stage, if necessary, from memory.

```
## Generate valA in execute stage
## LB: With load forwarding, want to insert valM
## from memory stage when appropriate
##
int e_valA = [
E_icode in {IRMMOVL, IPUSHL} && (E_srcA == M_dstM) : m_valM; # forwarding valM
    1 : E_valA; # Use valA from stage pipe register
];
```

```
## Generate valA in execute stage
int e_valA = E_valA; # Pass valA through stage
```

- F_stall: This set is used to judge whether the pipleine should stall or inject a bubble into pipeline register F. Most of the time it can be true.

```
bool F_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } &&
     E_dstM in { d_srcA, d_srcB } ||
    E_icode in { IMRMOVL, IPOPL} &&
     (E_dstM == d_srcB
     || !(D_icode in { IRMMOVL, IPUSHL}) && E_dstM == d_srcA) ||
```

- D_stall

```
bool D_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } &&
     E_dstM in { d_srcA, d_srcB };
     (E_dstM == d_srcB
     || !(D_icode in { IRMMOVL, IPUSHL}) && E_dstM == d_srcA);
```

- D_bubble

```
bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    (E_icode == IJXX && !e_Cnd && !(W_icode == IMRMOVL && M_icode == IRMMOVL &&
E_ifun == CLE)) ||
    # 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 }) &&
    !(E_icode in { IMRMOVL, IPOPL } &&
     (E_dstM == d_srcB
     || !(D_icode in { IRMMOVL, IPUSHL}) && E_dstM == d_srcA)) &&
     IRET in { D_icode, E_icode, M_icode };
```

- E_bubble

```
bool E_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    (E_icode == IJXX && !e_Cnd && !(W_icode == IMRMOVL && M_icode == IRMMOVL &&
E_ifun == CLE)) ||
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } &&
     E_dstM in { d_srcA, d_srcB};
     (E_dstM == d_srcB
     || !(D_icode in { IRMMOVL, IPUSHL}) && E_dstM == d_srcA);
```

Result

- First, check length

```
./check-len.pl < ncopy.yo
```

- Regression tests in y86-code and ptest document.----------Tests succeed.
- Construct the driver programs.

```
make drivers
make psim VERSION=full
```

- Test solution.

```
./psim -t sdriver.yo
```

```
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/pipe$ ./psim -t .
./y86-code/asumi.yo
Y86 Processor: pipe-full.hcl
112 bytes of code read

Cycle 0. CC=Z=1 S=0 O=0, Stat=AOK
F: predPC = 0x0
D: instr = nop, rA = ----, rB = ----, valC = 0x0, valP = 0x0, Stat = BUB
E: instr = nop, valC = 0x0, valA = 0x0, valB = 0x0
   srcA = ----, srcB = ----, dstE = ----, dstM = ----, Stat = BUB
M: instr = nop, Cnd = 0, valE = 0x0, valA = 0x0
   dstE = ----, dstM = ----, Stat = BUB
W: instr = nop, valE = 0x0, valM = 0x0, dstE = ----, dstM = ----, Stat = BUB
        Fetch: f_pc = 0x0, imem_instr = irmovl, f_instr = irmovl
        Execute: ALU: + 0x0 0x0 --> 0x0

Cycle 1. CC=Z=1 S=0 O=0, Stat=AOK
F: predPC = 0x6
D: instr = irmovl, rA = ----, rB = %esp, valC = 0x100, valP = 0x6, Stat = AOK
E: instr = nop, valC = 0x0, valA = 0x0, valB = 0x0
   srcA = ----, srcB = ----, dstE = ----, dstM = ----, Stat = BUB
M: instr = nop, Cnd = 1, valE = 0x0, valA = 0x0
   dstE = ----, dstM = ----, Stat = BUB
W: instr = nop, valE = 0x0, valM = 0x0, dstE = ----, dstM = ----, Stat = BUB
        Fetch: f_pc = 0x6, imem_instr = irmovl, f_instr = irmovl
        Execute: ALU: + 0x0 0x0 --> 0x0
```

```
Cycle 52. CC=Z=1 S=0 O=0, Stat=AOK
F: predPC = 0x44
D: instr = <bad>, rA = %ecx, rB = %ebp, valC = 0x8, valP = 0x44, Stat = AOK
E: instr = rrmovl, valC = 0x0, valA = 0xf4, valB = 0x0
   srcA = %esp, srcB = ----, dstE = %ebp, dstM = ----, Stat = AOK
M: instr = nop, Cnd = 0, valE = 0x0, valA = 0x0
   dstE = ----, dstM = ----, Stat = BUB
W: instr = halt, valE = 0x0, valM = 0x0, dstE = ----, dstM = ----, Stat = HLT
        Fetch: f_pc = 0x44, imem_instr = mrmovl, f_instr = <bad>
        Execute: ALU: + 0xf4 0x0 --> 0xf4
53 instructions executed
Status = HLT
Condition Codes: Z=1 S=0 O=0
Changed Register State:
%eax:   0x00000000      0x0000abcd
%ecx:   0x00000000      0x00000024
%esp:   0x00000000      0x000000f8
%ebp:   0x00000000      0x00000100
%esi:   0x00000000      0x0000a000
Changed Memory State:
0x00f0: 0x00000000      0x00000100
0x00f4: 0x00000000      0x00000039
0x00f8: 0x00000000      0x00000014
0x00fc: 0x00000000      0x00000004
ISA Check Succeeds
CPI: 49 cycles/38 instructions = 1.29
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/pipe$
```

```
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/pipe$ ./psim -t
ldriver.yo
Y86 Processor: pipe-full.hcl
1571 bytes of code read

Cycle 0. CC=Z=1 S=0 O=0, Stat=AOK
F: predPC = 0x0
D: instr = nop, rA = ----, rB = ----, valC = 0x0, valP = 0x0, Stat = BUB
E: instr = nop, valC = 0x0, valA = 0x0, valB = 0x0
   srcA = ----, srcB = ----, dstE = ----, dstM = ----, Stat = BUB
M: instr = nop, Cnd = 0, valE = 0x0, valA = 0x0
   dstE = ----, dstM = ----, Stat = BUB
W: instr = nop, valE = 0x0, valM = 0x0, dstE = ----, dstM = ----, Stat = BUB
        Fetch: f_pc = 0x0, imem_instr = irmovl, f_instr = irmovl
        Execute: ALU: + 0x0 0x0 --> 0x0

Cycle 1. CC=Z=1 S=0 O=0, Stat=AOK
F: predPC = 0x6
D: instr = irmovl, rA = ----, rB = %esp, valC = 0x624, valP = 0x6, Stat = AOK
E: instr = nop, valC = 0x0, valA = 0x0, valB = 0x0
```

```
0x05a0: 0x00cdefab      0x00000030
0x05a4: 0x00cdefab      0xffffffcf
0x05a8: 0x00cdefab      0x00000032
0x05ac: 0x00cdefab      0x00000033
0x05b0: 0x00cdefab      0x00000034
0x05b4: 0x00cdefab      0x00000035
0x05b8: 0x00cdefab      0x00000036
0x05bc: 0x00cdefab      0x00000037
0x05c0: 0x00cdefab      0xffffffc8
0x05c4: 0x00cdefab      0xffffffc7
0x05c8: 0x00cdefab      0xffffffc6
0x05cc: 0x00cdefab      0xffffffc5
0x05d0: 0x00cdefab      0xffffffc4
0x05d4: 0x00cdefab      0xffffffc3
0x05d8: 0x00cdefab      0xffffffc2
0x05dc: 0x00cdefab      0xffffffc1
0x0600: 0x00000000      0x00000209
0x0610: 0x00000000      0x00000624
0x0614: 0x00000000      0x00000029
0x0618: 0x00000000      0x000003e0
0x061c: 0x00000000      0x000004e4
0x0620: 0x00000000      0x0000003f
ISA Check Succeeds
CPI: 338 cycles/328 instructions = 1.03
```

- Test on ISA simulator

```
make drivers
../misc/yis sdriver.yo
```

```
jack@jack-K401LB:~/Downloads/Project2/project2-handout/sim/sim/pipe$ ../misc/yis sdriver.yo
Stopped in 51 steps at PC = 0x29.  Status 'HLT', CC Z=0 S=1 O=0
Changes to registers:
%eax:   0x00000000      0x00000002
%ecx:   0x00000000      0x00000404
%edx:   0x00000000      0x0000031c
%esp:   0x00000000      0x0000044c
%ebp:   0x00000000      0x00000458

Changes to memory:
0x0404: 0x00cdefab      0xffffffff
0x0408: 0x00cdefab      0x00000002
0x040c: 0x00cdefab      0xfffffffd
0x0410: 0x00cdefab      0x00000004
0x0434: 0x00000000      0x0000031c
0x0444: 0x00000000      0x00000458
0x0448: 0x00000000      0x00000029
0x044c: 0x00000000      0x000003e0
0x0450: 0x00000000      0x00000404
0x0454: 0x00000000      0x00000004
```

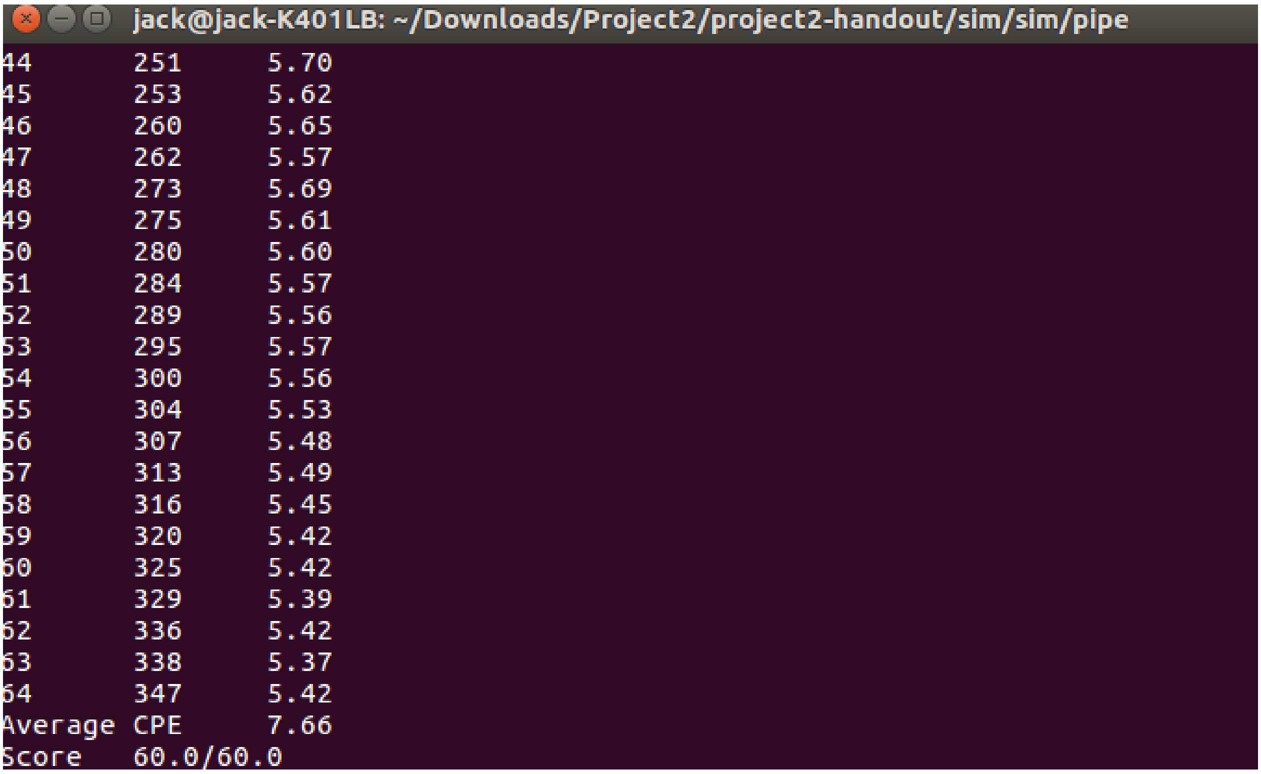
- Check correctness.

```
./correctness.pl
```

```
51      OK
52      OK
53      OK
54      OK
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
```

- CPE values test （7.66<10）

```
./benchmark.pl
```



```
jack@jack-K401LB: ~/Downloads/Project2/project2-handout/sim/sim/pipe
44      251     5.70
45      253     5.62
46      260     5.65
47      262     5.57
48      273     5.69
49      275     5.61
50      280     5.60
51      284     5.57
52      289     5.56
53      295     5.57
54      300     5.56
55      304     5.53
56      307     5.48
57      313     5.49
58      316     5.45
59      320     5.42
60      325     5.42
61      329     5.39
62      336     5.42
63      338     5.37
64      347     5.42
Average CPE     7.66
Score   60.0/60.0
```

4. Sense and Gains

Although we have learned all the theoretical knowledge in class, we didn't really know how to implement a pipelined CPU. Through this project, we have getten a deeper understanding of the Y86 instruction set, the assembly language, the pipeline and code optimization.

To complete this hard task, we refered to many resources on the Internet. We have learnt a lot from those internet forums like Stackoverflow and CSDN. Of course, We also discussed a lot about these problem. For example, to make our pipeline run faster in Part C, we mofidied the codes in **ncopy.ys** and **pipe-ful.hcl** again and again. The running CPE was just about 13 at first, we did a lot of work to reduce it to about 9.9. Finally, with careful analysis, we decided to unroll the loop and use a powerful tool called **Jump Table**. After this modification, the average CPE became about 7.6 which is much lower than before..

We began to work on this project just after we finished the projects of two courses - the Compiler Principle and the Programming Language. In fact, this project is not as hard as those two. The procedure we have experienced on these two projects also helps us handle with the Project2 more easily.

Finally, we would appreciate Teacher YanYan Shen and TA. With their help, we now better understand the relationship between hardware and software. We believe that these knowledge is really useful and we do benefit a lot from CS 359.