

# Project 1: Optimizing the Performance of a Pipelined Processor

Yanjie Ze , Zhenyu Tang  
519021910706 , 519021910891

{zeyanjie, tang\_zhenyu} @sjtu.edu.cn

May 8, 2021

## 1 Introduction

In this project, the task consists of 3 parts:

1. In Part A, we need to write Y86 assembly codes to implement 3 C functions. We wrote the code and checked the changed registers and memory.
2. In Part B, we are required to implement `iaddl` and `leave` in the given pipelined processor. After that, we tested them on the ISA.
3. In Part C, our goal is to optimize the performance of a pipelined processor. We combined Part B implementation here, and try 5 times improving the assembly code, using tricks like loop unrolling, preventing load use hazard, binary searching remaining loop number, rearranging jump instructions according to AT (Always Taken strategy), reaching final  $CPE = 8.93$ .

**Arrangment:** Yanjie Ze mainly took charge of finishing the coding part, and Zhenyu Tang mainly took charge of the report. To be more precise:

1. Ze first solved Part A and Part B. Then Tang also finished separately in Part A and Part B, and checked whether the solution of Ze was correct.
2. Ze finished Part C with  $CPE = 9.21$ , using three tricks and one night. Then Tang followed Ze's solution and began the writing of the final report.
3. Tang rewrote the binary search code in Part C based on Ze's solution and then added another trick getting  $CPE = 8.93$ , after which he finished the final report, together with Ze's participation.

The final version of project code consists of Tang's Part A, Ze's Part B, and Ze's Part C with Tang's modification.

## 2 Experiments

### 2.1 Part A

#### 2.1.1 Analysis

In this part, we are required to translate 3 functions written in C into Y86 assembly language, namely `sum_list()`, `rsum_list()` and `copy_block()`.

Implementing `sum_list()` within `sum.y8` is trivial referencing the codes on CS:APP book Fig. 4-7 with changing all the instructions to that of double words operands. As for `rsum_list()`, we implemented a recursive function using the stack. Therefore, we need to `pushl` the temporal data and then after calling the function `popl` them out and do the computation. For `copy_block()`, we copy data sized `len` from `src` to `dest`.

In all of the above implementations, we cannot change the callee-saved registers. We need to push them into the stack before usage and afterwards pop and resume them.

#### 2.1.2 Code

##### 1. Core part of `sum.y8`

```
1 # ...
2 # int sum_list(list_ptr ls)
3 # ls in %edi
4 sum_list:
5     pushl %ebp                # Save the callee-saved
6         register
7     xorl %eax, %eax           # int val = 0;
8     jmp test
9 loop:
10    mrmovl (%edi), %ebp        # Get ls->val
11    addl %ebp, %eax             # val += ls->val;
12    mrmovl 4(%edi), %edi        # ls = ls->next;
13 test:
14    andl %edi, %edi            # Judge on ls
15    jne loop                   # Stop when %edi is
16                                zero
17    popl %ebp                  # Resume the callee-
18                                saved register
19    ret
20 # ...
```

##### 2. Core part of `rsum.y8`

```
1 # ...
2 # int rsum_list(list_ptr ls)
```

```

3 # ls in %edi
4 rsum_list:
5     xorl %eax, %eax           # int val = 0;
6     xorl %ecx, %ecx           # Prevent problems
7     andl %edi, %edi           # Judge on input
8     je finish
9     mrmovl (%edi), %eax        # int val = ls->val;
10    pushl %eax                 # Save
11    mrmovl 4(%edi), %edi        # ls = ls->next;
12    call rsum_list
13    popl %ecx
14 finish:
15    addl %ecx, %eax
16    ret
17 # ...

```

### 3. Core part of copy.y

```

1 # ...
2 # int copy_block(int *src, int *dest, int len)
3 # src in %edi, dest in %esi, len in %edx
4 copy_block:
5     pushl %ebp                # Save callee-saved
6     registers
7     pushl %ebx
8     irmovl $4, %ecx           # Save constant 4 since
9     Y86 only supports addition and subtraction on
10    registers
11    irmovl $1, %ebx            # Save constant 1
12    xorl %eax, %eax            # int result = 0;
13    jmp test
14 loop:
15    mrmovl (%edi), %ebp        # int val = *src;
16    addl %ecx, %edi            # src++;
17    rmmovl %ebp, (%esi)        # *dest = val;
18    addl %ecx, %esi            # dest++;
19    xorl %ebp, %eax            # result ^= val;
20    subl %ebx, %edx            # len--;
21 test:
22    andl %edx, %edx            # Judge on len
23    jne loop                   # Stop when len = 0
24    popl %ebx                  # Resume callee-saved
25    registers
26    popl %ebp
27    ret
28 # ...

```

### 2.1.3 Evaluation

1. For `sum_list()`, as can be seen in Fig. 1, `%eax` is changed to `0xcba`, and no callee-saved registers are changed, which is correct. Note that stack pointers are all set `0x200` in the programs.

```
zhenyu@zhenyu-Legion-Y7000-2019-PG0:~/桌面/storage/school_usage/computer_architecture/prj-1/ArchLab/misc$ ./yas sum.y
zhenyu@zhenyu-Legion-Y7000-2019-PG0:~/桌面/storage/school_usage/computer_architecture/prj-1/ArchLab/misc$ ./yis sum.yo
Stopped in 28 steps at PC = 0xb. Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000      0x00000cba
%esp: 0x00000000      0x00000200

Changes to memory:
0x01f8: 0x00000000      0x0000002f
0x01fc: 0x00000000      0x0000000b
```

Figure 1: Result of `sum.y`

2. For `rsum_list()`, as can be seen in Fig. 2, `%eax` is also changed to `0xcba`, and no callee-saved registers are changed, which is correct.

```
zhenyu@zhenyu-Legion-Y7000-2019-PG0:~/桌面/storage/school_usage/computer_architecture/prj-1/ArchLab/misc$ ./yas rsum.y
zhenyu@zhenyu-Legion-Y7000-2019-PG0:~/桌面/storage/school_usage/computer_architecture/prj-1/ArchLab/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      0x00000200

Changes to memory:
0x01e0: 0x00000000      0x0000004e
0x01e4: 0x00000000      0x00000c00
0x01e8: 0x00000000      0x0000004e
0x01ec: 0x00000000      0x000000b0
0x01f0: 0x00000000      0x0000004e
0x01f4: 0x00000000      0x0000000a
0x01f8: 0x00000000      0x0000002f
0x01fc: 0x00000000      0x0000000b
```

Figure 2: Result of `rsum.y`

3. For `copy_block()`, as can be seen in Fig. 3, the data are copy to the correct place and the return number is `0xcba`.

## 2.2 Part B

### 2.2.1 Analysis

In this part, we are required to implement `iaddl` instruction within the existing processor description in HCL. The instruction uses an immediate and a value of a register respectively as the ALU input, does addition and writes to a register. Therefore, we simply add `IIADDL` into the corresponding code parts.

```

zhenyu@zhenyu-Legion-Y7000-2019-PG0:~/桌面/storage/school_usage/computer_archite
cture/prj-1/ArchLab/misc$ ./yas copy.y
zhenyu@zhenyu-Legion-Y7000-2019-PG0:~/桌面/storage/school_usage/computer_archite
cture/prj-1/ArchLab/misc$ ./yis copy.yo
Stopped in 43 steps at PC = 0xb. Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000      0x00000c0a
%ecx: 0x00000000      0x00000004
%esp: 0x00000000      0x00000200
%esi: 0x00000000      0x00000024
%edi: 0x00000000      0x00000018

Changes to memory:
0x0018: 0x00000111      0x0000000a
0x001c: 0x00000222      0x000000b0
0x0020: 0x00000333      0x00000c00
0x01f8: 0x00000000      0x0000003b
0x01fc: 0x00000000      0x0000000b

```

Figure 3: Result of copy.y

## 2.2.2 Code

```

1 # ...
2 bool instr_valid = icode in
3   { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
4     IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL };
5
6 # Does fetched instruction require a regid byte?
7 bool need_regids =
8   icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
9     IIRMOVL, IRMMOVL, IMRMOVL, IIADDL };
10
11 # Does fetched instruction require a constant word?
12 bool need_valC =
13   icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL };
14 # ...
15 ## What register should be used as the B source?
16 int srcB = [
17   icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL } : rB;
18   # ...
19 ];
20
21 ## What register should be used as the E destination?
22 int dstE = [
23   icode in { IRRMOVL } && Cnd : rB;
24   icode in { IIRMOVL, IOPL, IIADDL } : rB;
25   # ...
26 ];
27 # ...
28 ## Select input A to ALU
29 int aluA = [
30   icode in { IRRMOVL, IOPL } : valA;
31   icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : valC;

```

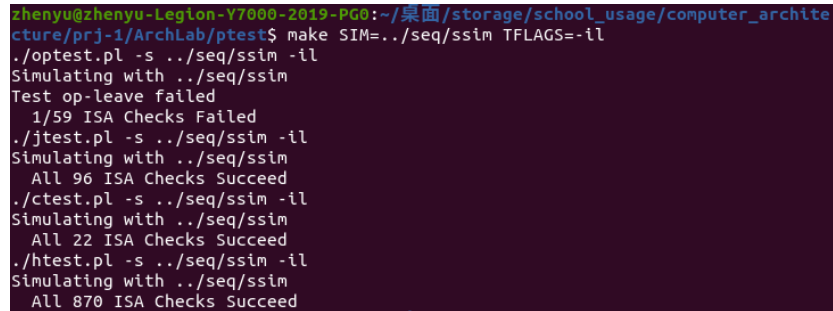
```

32 | # ...
33 | ];
34 |
35 | ## Select input B to ALU
36 | int aluB = [
37 |     icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
38 |             IPUSHL, IRET, IPOPL, IIADDL } : valB;
39 | # ...
40 | ];
41 | # ...
42 | ## Should the condition codes be updated?
43 | bool set_cc = icode in { IOPL, IIADDL };
44 | # ...

```

### 2.2.3 Evaluation

As can be seen in Fig. 4, all the 870 ISA Checks, including `iaddl` and `leave`, succeed.



```

zhenyu@zhenyu-Legion-Y7000-2019-PC0:~/桌面/storage/school_usage/computer_archite
cture/prj-1/ArchLab/ptest$ make SIM=../seq/ssim TFLAGS=-il
./optest.pl -s ../seq/ssim -il
Simulating with ../seq/ssim
Test op-leave failed
  1/59 ISA Checks Failed
./jtest.pl -s ../seq/ssim -il
Simulating with ../seq/ssim
  All 96 ISA Checks Succeed
./ctest.pl -s ../seq/ssim -il
Simulating with ../seq/ssim
  All 22 ISA Checks Succeed
./htest.pl -s ../seq/ssim -il
Simulating with ../seq/ssim
  All 870 ISA Checks Succeed

```

Figure 4: Test on Part B

## 2.3 Part C

### 2.3.1 Analysis

In this part, we are required to optimize the performance of a pipelined processor on a function. We can optimize it on both the processor and the assembly code. The improvement history can be seen in Fig. 5.

On the processor part, we added the newly **implemented** `iaddl` to the pipeline, with which we were able to directly add immediates to registers rather than loading them into the registers first. Since in the assembly part, we would

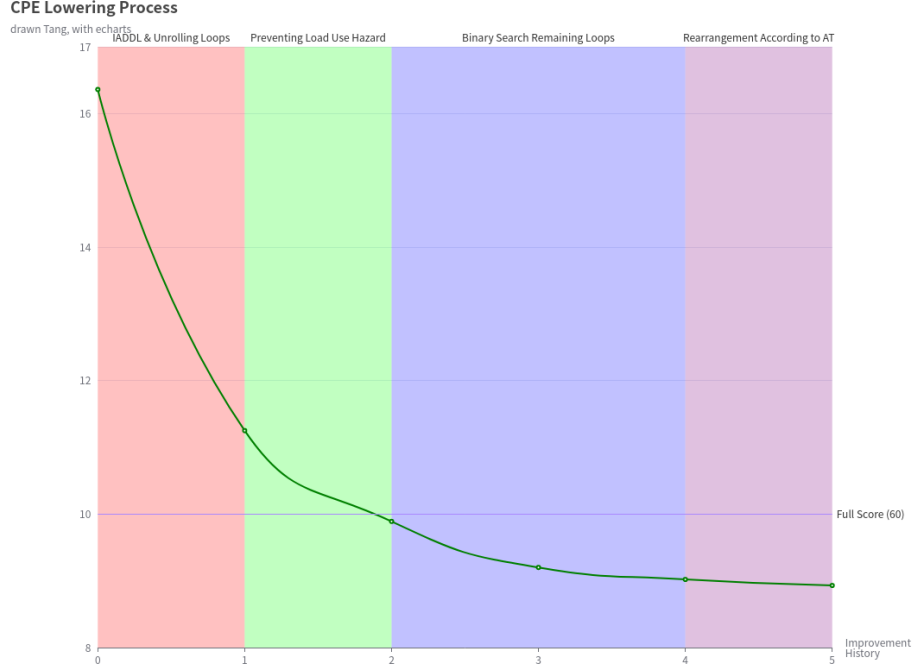


Figure 5: The history of  $CPE$  down

try our best to avoid load-use hazard, optimization on load-use hazard in this part sees no much improvement in test, which we discarded.

On the assembly code, we first did **loop unrolling** and checked the best version of unrolling. As can be seen in Tab. 1, unrolling to **6 times** outperforms the other, and therefore we chose it.

Unroll times	1	4	5	6	7	8	10	12
Score (/60)	0	27.2	29.5	30.0	29.4	28.2	25.5	21.9
$CPE$	16.36	11.37	11.27	11.25	11.27	>11.37	>11.37	>11.37

Table 1: Performance of unrolling

And then, we **prevented the load use hazard** by using two registers to store the data and loading beforehand. In the loop we load data in one register and switch to another to do operation on the previously loaded data, thus preventing bubbles and lowering  $CPE$ . In this case, the best unrolling time is supposed to be 4 ~ 6 in Tab. 2.

For the rest part of the loop besides unrolling, we did a **binary search** and get the remaining number of loops and then unroll them again. Here we get  $CPE = 9.21$  for unrolling 6 times, and  $CPE = 9.20$  for unrolling 8 times.

Unroll times	3	4	5	6	7
Improved score (/60)	59.8	60	60	60	59.4
Improved <i>CPE</i>	10.01	9.89	9.90	9.95	10.03

Table 2: Performance after preventing the load use hazard

Further rewriting the binary searching process we made it down to  $CPE = 9.02$ .

Another trick is to **arrange the conditional jump instructions**. As can be seen from the assignment of **bubbles** in the pipeline, the strategy used to predict a branch is **AT (Always Taken)** rather than the predict not taken strategy we used in implementing the MIPS pipelined processor. Hence in the binary search tree we arrange **je** after **jl** etc. since the latter contains more cases and is more probable to be taken. This trick helped lower *CPE* from 9.02 to less than 8.93.

The critical part here is **trade-offs**. Unrolling more cycles will increase the performance, but also enlarge the instructions size. Moreover, unrolling more cycles will also increase the overhead of binary searching since the remain cases go up in numbers. Also, the arrangement according to AT is also a trade-off where we supposed the data to be uniformly distributed and hence chose the seemingly better

### 2.3.2 Code

The following is the main part of implementation codes.

```

1 # ...
2 # Loop header
3 xorl %eax,%eax # count = 0;
4 iaddl $-6, %edx # len = len - 6
5 jl restjudge # if len<0, goto rest:
6
7 loop1: mrmovl (%ebx), %esi # val1
8 mrmovl 4(%ebx), %edi # val2
9 rmmovl %esi, (%ecx) # store val1
10 andl %esi, %esi # val1 <=0?
11 jle loop2 # if so, go to next loop
12 iaddl $1, %eax # count ++
13
14 loop2: mrmovl 8(%ebx), %esi # val1
15 rmmovl %edi, 4(%ecx) # store val2
16 andl %edi, %edi # val2 <=0?
17 jle loop3 # if so, go to next loop
18 iaddl $1, %eax # count ++
19
20 # The rest loops are similar to the above two, here omitted
21 # ...
22
23 loop6:

```



```

24 rmmovl %edi, 20(%ecx) # store val2
25 andl %edi, %edi # val2 <=0?
26 jle update # if so, go to update, since this is the final loop
27 iaddl $1, %eax # count ++
28
29 update: iaddl $24, %ebx # update
30 iaddl $24, %ecx # update
31 iaddl $-6, %edx # len -= 6
32 jge loop1 # if len >= 0, go to new loop1
33
34 ### rest loop ###
35 # binary search
36 # rearranged jl, jg and je according to AT
37 restjudge:
38     iaddl $3, %edx
39     jl searchleft
40     jg searchright
41     jmp restloop3
42 searchright:
43     iaddl $-1, %edx
44     je restloop4
45     jmp restloop5
46 searchleft:
47     iaddl $2, %edx
48     je restloop1
49     jg restloop2
50     jmp Done
51 ### search finish ###
52
53 ### begin unrolling ###
54 restloop1: mrmovl (%ebx), %esi
55 rmmovl %esi, (%ecx)
56 andl %esi, %esi
57 jle Done # finish
58 iaddl $1, %eax
59 jmp Done
60
61 restloop2: mrmovl (%ebx), %esi # val1
62 mrmovl 4(%ebx), %edi # val2
63 rmmovl %esi, (%ecx) # store val1
64 andl %esi, %esi # val1 <=0?
65 jle restloop21 # if so, go to next loop
66 iaddl $1, %eax # count ++
67
68 restloop21: rmmovl %edi, 4(%ecx)
69 andl %edi, %edi
70 jle Done
71 iaddl $1, %eax
72 jmp Done
73

```

```

74 # the other rest loops are alike, here omitted
75
76 # ...

```

### 2.3.3 Evaluation

1. We check the correctness like below:

```

1 zhenyu@zhenyu-Legion-Y7000-2019-PG0:/storage/school_usage/
  computer_architecture/prj-1/ArchLab/pipe$ ./correctness
  .pl -p
2 Simulating with pipeline simulator psim
3 ncopy
4 0 OK
5 # omit the other output OKs
6 64 OK
7 128 OK
8 192 OK
9 256 OK
10 68/68 pass correctness test

```

2. And then check the copied length:

```

1 zhenyu@zhenyu-Legion-Y7000-2019-PG0:/storage/school_usage/
  computer_architecture/prj-1/ArchLab/pipe$ ./check-len.
  pl < ncopy.yo
2 ncopy length = 678 bytes

```

3. And then the CPE, the final CPE is around 8.93:

```

1 zhenyu@zhenyu-Legion-Y7000-2019-PG0:/storage/school_usage/
  computer_architecture/prj-1/ArchLab/pipe$ ./benchmark.
  pl
2 ncopy
3 0 40
4 1 39 39.00
5 2 48 24.00
6 3 53 17.67
7 # omit the outputs
8 60 414 6.90
9 61 413 6.77
10 62 422 6.81
11 63 427 6.78
12 64 434 6.78

```

## 3 Conclusion

### 3.1 Problems

During the project, we met several obstacles and overcame them:

1. When referencing the codes on CS:APP writing Part A, Tang found that the YAS assembler triggered an error **Invalid line**. Double checking the codes on book and the provided test data, he figured out that the data size mismatch that of the operators in the book. Hence, he changed all the operators suffixed “q” to “l” and the problem was solved.
2. When trying Part C at first, Tang encountered a problem that using `iaddl` the *CPE* directly goes down to about 2, which was seemingly impossible with just a few modifications. Checking the copied length he found that the `iaddl` had not been implemented in the processor in Part C. Combining Part B solution in the pipelined processor in Part C he finally got the answer back to normal.
3. When trying modifying on Ze’s Part C solution, Tang tried optimizing the stalling and bubbles conditions. However, the trial saw barely any improvement, and afterwards he recognized that the load use hazard has already been prevented by Ze. Therefore, such trial is not expected to optimize a lot.

### 3.2 Achievements

For Part A, our program works as expected on the tested data.

For Part B, our processor passed the test of all given 870 ISA checks.

For Part C, our project passed the correctness check, and yields a *CPE* of 8.93 or lower, almost half of that of the raw one. In the recent 10 tests, 8.93 occurs twice, 8.92 occurs 7 times, and 8.91 occurs once.

In the project, we have our codes commented as fully as possible, making it more convenient to cooperate. Moreover, we worked with each other over GitHub for coding and Overleaf (SJTU) for report writing, getting more familiar using those tools.

We managed to bridge between what we have learned in the class and practice. With this project, we learned to write assembly codes, and writing a new hardware description language - HCL. More importantly, reading the pre-written HCL codes we learned about the implementation of pipelining in a CISC

ISA, which is similar to our implementation on that of MIPS ISA, but with some noticeable differences like **valP** to predict PC for the next instruction. We also gain a deeper view inside data hazard, strategies for solving control hazards and the compiler technique of loop unrolling etc modifying on the assembly codes.