### Project 1: Optimizing the Performance of a Pipelined Processor

520021910863 Heda Chen

520030910287 Xinming Shu Xin\_ming Shu@sjtu.edu.cn

1. Introduction

Part A

What we need to do in this part is write three simple assembly programs to implement three functions in example.c in directory sim/misc. These programs are functionally equivalent to the C. We also annotate the code detailedly. The code is accurate, efficient and readable.

Part B

In Part B, we modify the *hcl* file of the Y86’s sequential design to add the new instruction *iaddl*. We add instruction code for iaddl instruction, then add dependency relations of *iaddl* to all boolean signals and design the datapath for *iaddl* such as generate control signals for *src* and *dst.*

Part C

In Part C, we implement the *iaddq* instruction in *pipe-full.hcl*, and then optimize the efficiency of the assembler file *ncopy.ys* and the file *pipe-full.hcl*. We try several methods such as applying loop unrolling and modifying the pipeline design in branch prediction for certain code pattern, which optimizes the performance and brings CPE down to 9.17.

Contribution

Heda Chen: finished Part A, B and C

Xinming Shu: retry Part A, B and C and finished the report

1. Experiments
   1. Part A
      1. Analysis

This part is a ‘warm-up’ session of this project. Write and simulate the functions of the three functions in example.c using the Y86 directive. Then use YAS to compile and use YIS to run.

* Key Points

1. Protect the callee-saved register EBX, EDI, ESI.
2. Pull the correct element from the stack.

* Difficult Points

1) Implement function recursion smartly and clearly.

2) To be a proficient in the use of Y86 instruction set.

* Core Technique

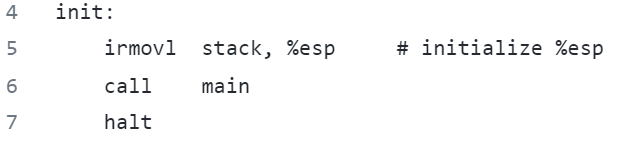
1) Code line by line referring to functions wrote by C language.

2) Mimicking C functions, division of functional areas with enough and clear label.

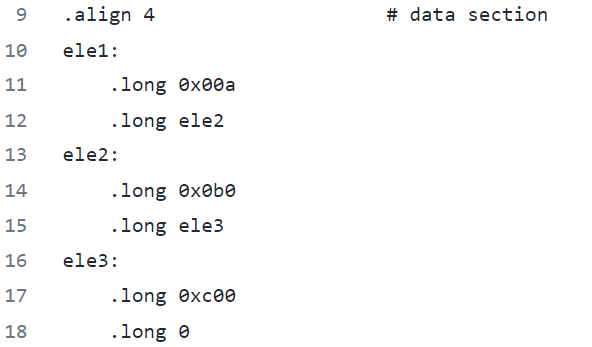
3) Draw a picture of stack to ensure the correctness.

* + 1. Code
* sum.ys

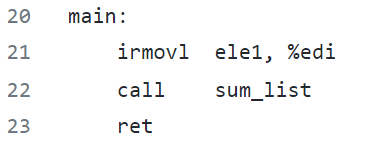
1) Initialize the stack frame



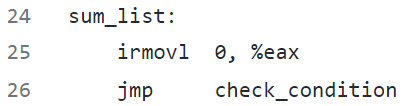
2) Data definition for sample linked list according to the file *project1*



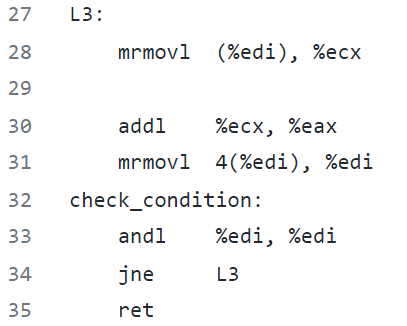
3) Define main and call *sum\_list*: following the calling conventions, use %rdi to store the first argument



4) Implement *sum\_list*: initialize and jump into loop



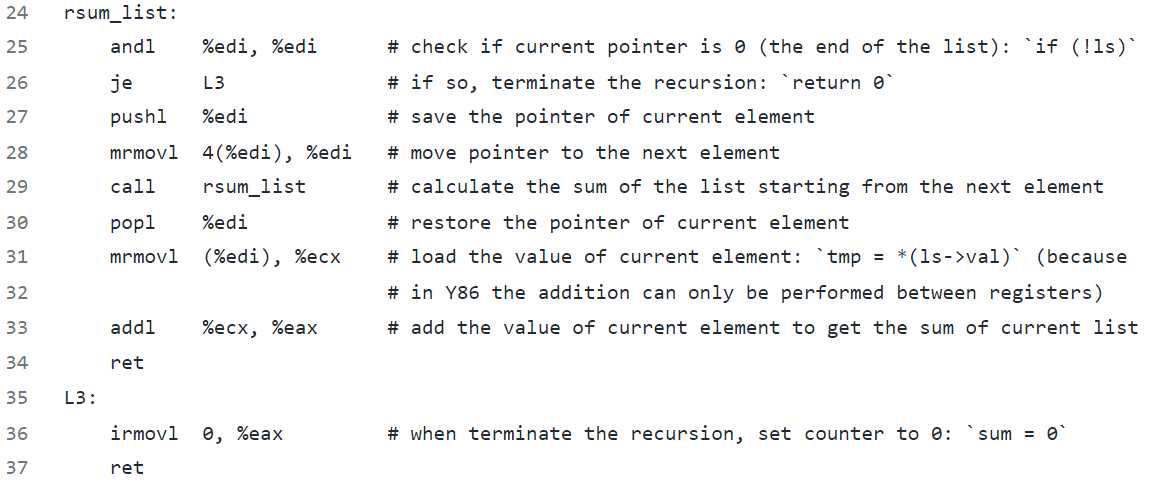
5) Implement loop and check



* rsum.ys

1) Initialize %esp, define data and main as *rum.ys*

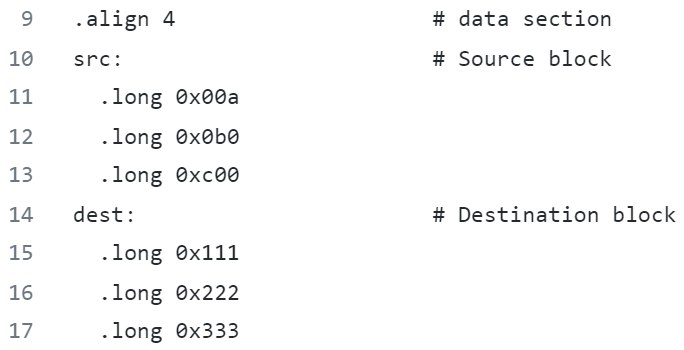
2) Sum the linked list elements recursively



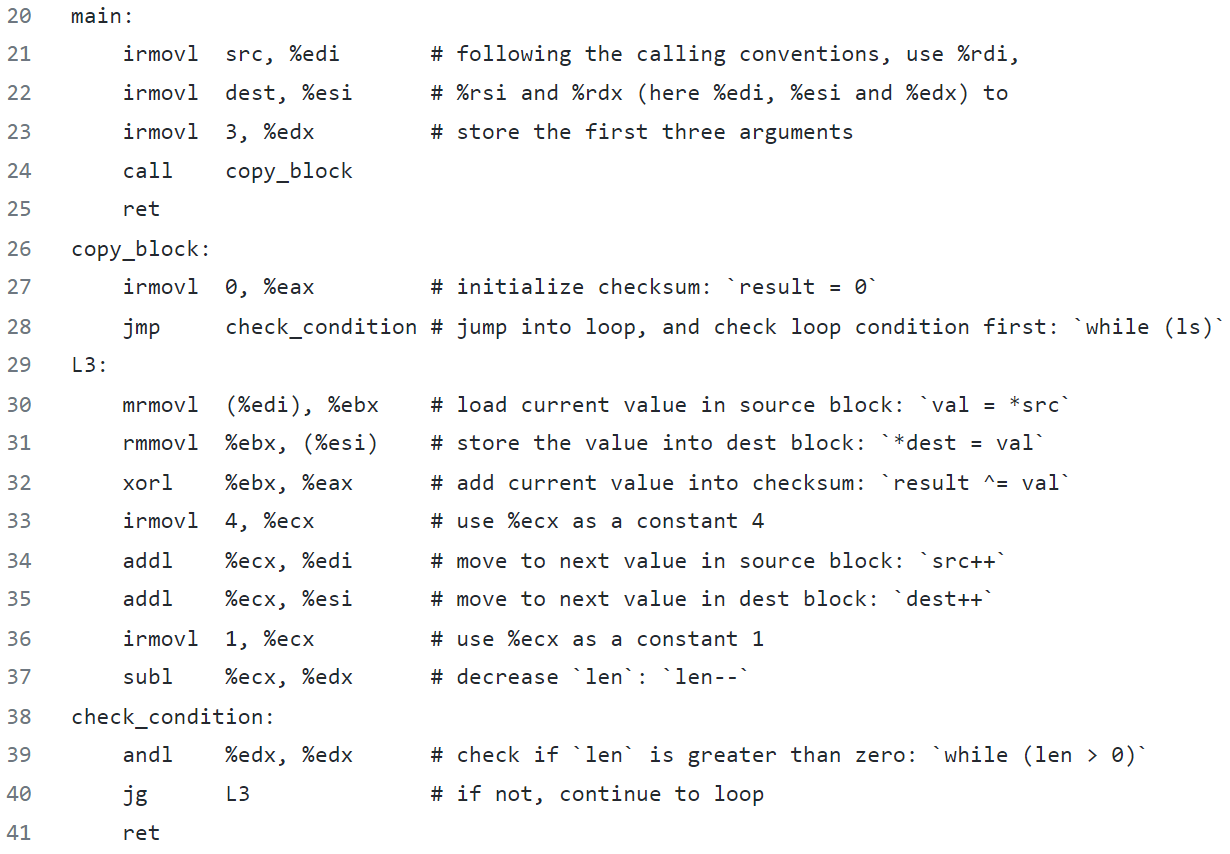
* copy.ys

Copy a block of words from one part of memory to another (non-overlapping region), computing the checksum Xor of all copied words

1. Data section, source and destination block



1. Copy src to dest and return Xor validation



* + 1. Evaluation
* sum.ys

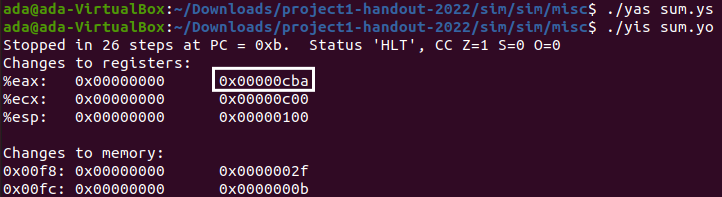


Figure 1 Part A\_sum

The %eax register has the correct value which is the return value of the function **0xcba**.

* rsum.ys

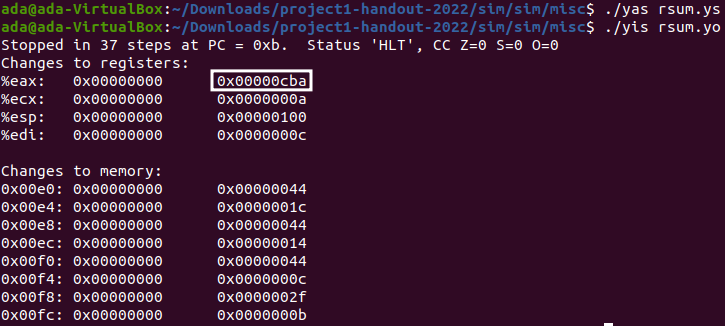


Figure 2 Part A\_rsum

The %eax register has the correct value **0xcba**.

* copy.ys

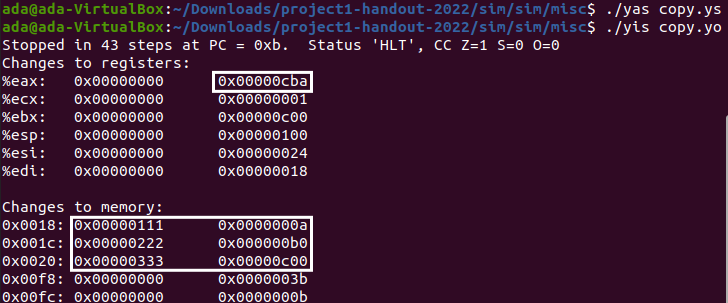


Figure 3 Part A\_copy

1. The %eax register has the correct value which is the return value of the function **0xcba**.
2. Values are written into the memory correctly as I labeled in the picture above.
   1. Part B
      1. Analysis

In this part, we should extend the SEQ processor to support instruction *iaddl* by modifying *seq-full.hcl*.

* Difficult Points

1) Understand the processing logic and take a good command of the syntax of *HCL*.

2) Judge whether iaddl should be added into the functions in stages.

* Steps

1) In Fetch Stage, add *IIADDL* in the choices regions of *(bool) instr\_valid, (bool) need\_regid* and *(bool) need\_valC*.

2) In Decode Stage, when icode is *IIADDL*, *srcB* is from *rB* since the second operand of *iaddl* is a register, *dstE* (where the result from ALU is passed towards) is *rB* since ‘*iaddl constant, rB*’ means *rB += constant* (*rB* is updated).

3) In Execute Stage, add *IIADDL* in the choices region of *(bool) set\_cc* since *iaddl* operation involves ALU operation which will set flags.

4) In Execute Stage, 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*).

5) In Execute Stage, when icode is *IIADDL*, *aluB* (the second op) is *valB* (the value of the second register that is read) for the same reason above.

* + 1. Code

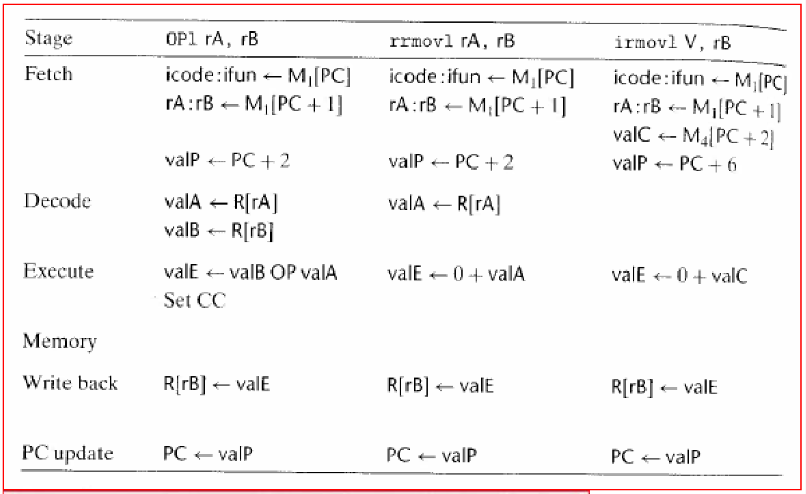
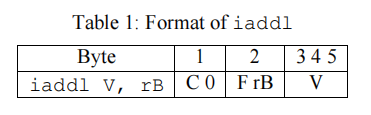


Figure 4 Stages of executing the instruction



Combining *irmovl, opl* calculation process, through analysis, we get *iaddl* calculation process, and then we modify the code as below:

1. Fetch Stage

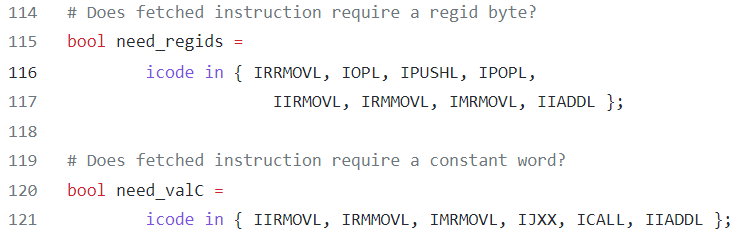


Figure 5 seq-full\_fetch

1. Decode Stage

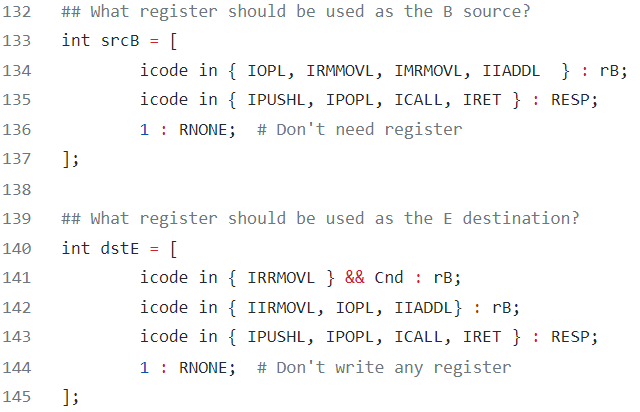


Figure 6 seq-full\_decode

1. Execute Stage

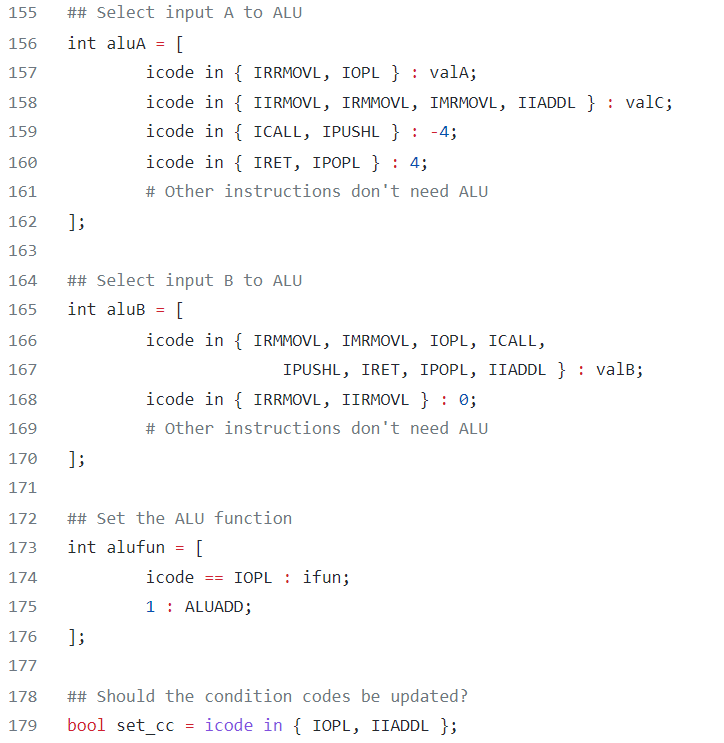
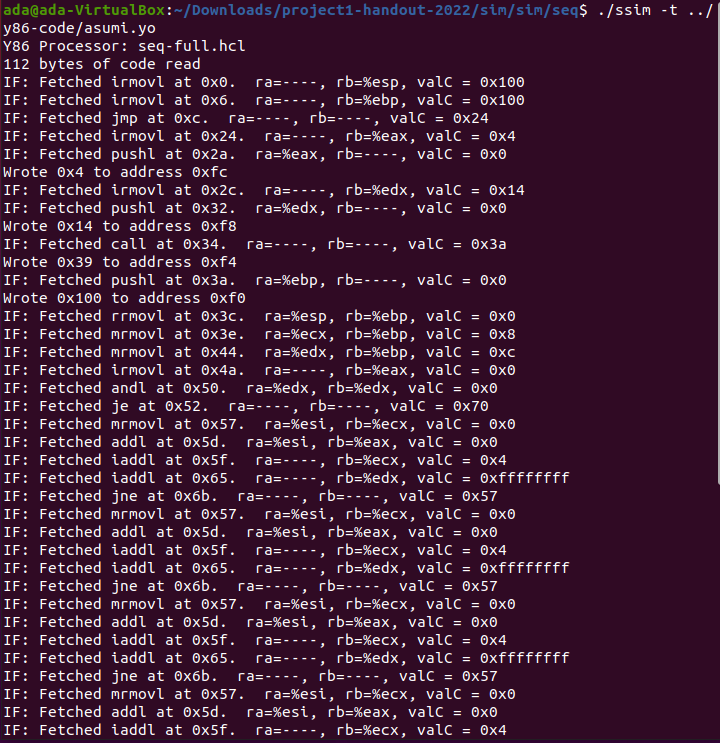


Figure 7 seq-full\_execute

* + 1. Evaluation

Build a new instance of the SEQ simulator (ssim) based on this HCL file, and then test it on a simple Y86 program.

1. First, we run a simple program *asumi.yo* (testing iaddl), part of the content was omitted below



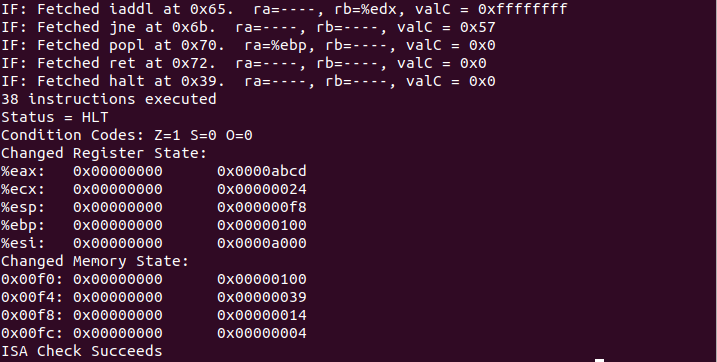


Figure 8 Run asumi in TTY mode

1. Then we retesting the solution using the benchmark programs

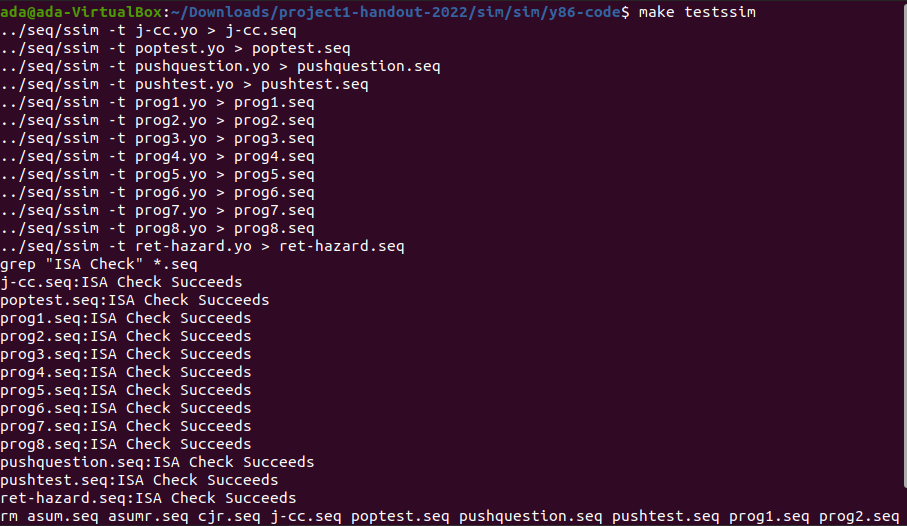


Figure 9 Part B\_Benchmark test

1. Performing regression test

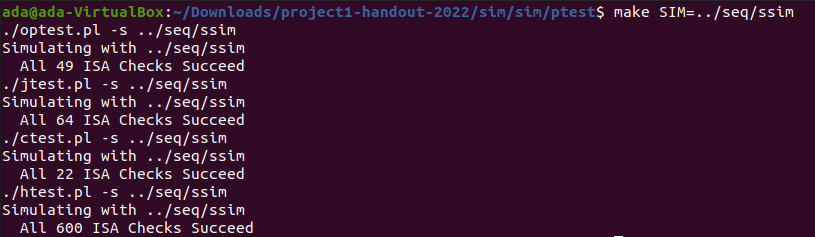


Figure 10 Part B\_Regression test

1. Test the implementation of *iaddl*

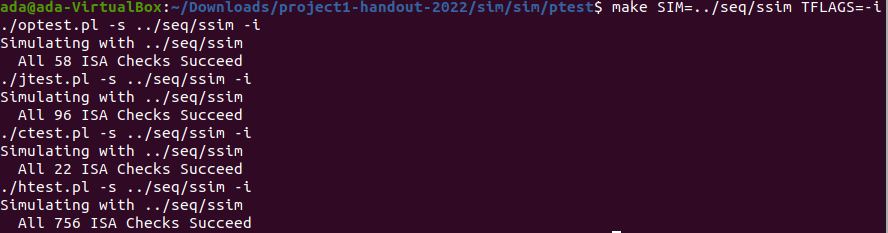


Figure 11 Part B\_test *iaddl*

* 1. Part C
     1. Analysis

In this part, we were asked to speed up the program *ncopy.ys* as much as possible by modifying the *nocpy.ys* and *HCL*.

1. Add *iaddl* instruction to *pipe-full.hcl*

Add the use of instruction *iaddl*, modify the file *pipe-full* as what we did in the file *seq-full* in Part B.

1. Use the instruction *iaddl* to optimize the code:



Figure 12 Count++ before substitution



Figure 13 Count++ after substitution

1. Avoid load and use conflict:

When a load and use hazard occurs, it will waste a clock cycle, which can be avoided and utilized by reordering instructions. What’s more, we use two registers to store the variable *val*, loading them separately and ahead of time.



Figure 14 Load-use hazard

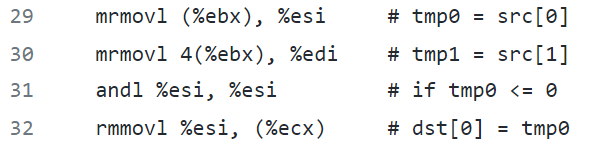


Figure 15 Deal with load-use hazard

1. 7-way Loop Unrolling

We perform a technique named ‘loop unrolling’ to deal with much overhead in testing and updating procedure of loops. We do multiple loops for the part that > 6, and update the relevant data at once, to reduce the number of times we execute the *add* and *jxx* instructions.

1. Search Tree for Remaining Elements

For large inputs, it’s better to unroll the loops for more ways. However, for small inputs, it is important to choose a good method to process the remainning elements. For the part that <= 6, the simplest way is to write another loop for them. But there is another way performs better, that is, jump to different position for different number of remaining ones. Since Y86 does not support relative jump instruction, we designed a search tree to get the correct jump destination for each possibility.

* + 1. Code
* **7-Way Loop Unrolling**

1. xorl %eax,%eax    # count = 0;
2. iaddl -6, %edx    # len <= 6?
3. jle S06           # **if** so, **goto** S06
5. L0:
6. mrmovl (%ebx), %esi     # tmp0 = src[0]
7. mrmovl 4(%ebx), %edi    # tmp1 = src[1]
8. andl %esi, %esi         # **if** tmp0 <= 0
9. rmmovl %esi, (%ecx)     # dst[0] = tmp0
10. jle L1                  # skip increasing counter
11. iaddl 1, %eax           # **else** counter++
12. L1:
13. mrmovl 8(%ebx), %esi    # tmp0 = src[2]
14. andl %edi, %edi         # **if** tmp1 <= 0
15. rmmovl %edi, 4(%ecx)    # dst[1] = tmp1
16. jle L2                  # skip increasing counter
17. iaddl 1, %eax           # **else** counter++
18. L2:
19. mrmovl 12(%ebx), %edi   # tmp1 = src[3]
20. andl %esi, %esi         # **if** tmp0 <= 0
21. rmmovl %esi, 8(%ecx)    # dst[2] = tmp0
22. jle L3                  # skip increasing counter
23. iaddl 1, %eax           # **else** counter++
24. L3:
25. mrmovl 16(%ebx), %esi   # tmp0 = src[4]
26. andl %edi, %edi         # **if** tmp1 <= 0
27. rmmovl %edi, 12(%ecx)   # dst[3] = tmp1
28. jle L4                  # skip increasing counter
29. iaddl 1, %eax           # **else** counter++
30. L4:
31. mrmovl 20(%ebx), %edi   # tmp1 = src[5]
32. andl %esi, %esi         # **if** tmp0 <= 0
33. rmmovl %esi, 16(%ecx)   # dst[4] = tmp0
34. jle L5                  # skip increasing counter
35. iaddl 1, %eax           # **else** counter++
36. L5:
37. mrmovl 24(%ebx), %esi   # tmp0 = src[6]
38. andl %edi, %edi         # **if** tmp1 <= 0
39. rmmovl %edi, 20(%ecx)   # dst[5] = tmp1
40. jle L6                  # skip increasing counter
41. iaddl 1, %eax           # **else** counter++
42. L6:
43. andl %esi, %esi         # **if** tmp0 <= 0
44. rmmovl %esi, 24(%ecx)   # dst[6] = tmp0
45. jle ChkCond             # skip increasing counter
46. iaddl 1, %eax           # **else** counter++
47. ChkCond:
48. iaddl 28, %ebx          # src += 7 \* **sizeof**(Byte)
49. iaddl 28, %ecx          # dst += 7 \* **sizeof**(Byte)
50. iaddl -7, %edx          # len -= 7
51. jg L0                   # **if** len > 0, **goto** L0

* **Binary Search Tree for Finding the Number of Remaing Loops**

1. S06:              # len in (x-6: [0, 6] -> [-6, 0])
2. iaddl 3, %edx # (x-6: [0, 6] -> [-6, 0]) => (x-3: [0, 6] -> [-3, 3])
3. jg S46          # len-3 > 0, len in [4, 6]
4. je R3           # len-3 == 0, len = 3
5. S02:              # len in (x-3: [0, 2] -> [-3, -1])
6. iaddl 2, %edx   # (x-3: [0, 2] -> [-3, -1]) => (x-1: [0, 2] -> [-1, 1])
7. jl R0           # len-1 < 0, len = 0
8. je R1           # len-1 == 0, len = 1
9. jmp R2          # len-1 > 0, len = 2
10. S46:             # len in (x-3: [4, 6] -> [1, 3])
11. iaddl -2, %edx # (x-3: [4, 6] -> [1, 3]) => (x-5: [4, 6] -> [-1, 1])
12. jl R4          # len-5 < 0, len = 4
13. je R5          # len-5 == 0, len = 5

* Unrolling of Remaining Loops

1. R6:
2. mrmovl 20(%ebx), %esi   # tmp = src[6]
3. andl %esi, %esi         # **if** tmp <= 0
4. rmmovl %esi, 20(%ecx)   # dst[6] = tmp
5. jle R5                  # skip increasing counter
6. iaddl 1, %eax           # **else** counter++
7. R5:
8. mrmovl 16(%ebx), %esi   # tmp = src[5]
9. andl %esi, %esi         # **if** tmp <= 0
10. rmmovl %esi, 16(%ecx)   # dst[5] = tmp
11. jle R4                  # skip increasing counter
12. iaddl 1, %eax           # **else** counter++
13. R4:
14. mrmovl 12(%ebx), %esi   # tmp = src[4]
15. andl %esi, %esi         # **if** tmp <= 0
16. rmmovl %esi, 12(%ecx)   # dst[4] = tmp
17. jle R3                  # skip increasing counter
18. iaddl 1, %eax           # **else** counter++
19. R3:
20. mrmovl 8(%ebx), %esi    # tmp = src[3]
21. andl %esi, %esi         # **if** tmp <= 0
22. rmmovl %esi, 8(%ecx)    # dst[3] = tmp
23. jle R2                  # skip increasing counter
24. iaddl 1, %eax           # **else** counter++
25. R2:
26. mrmovl 4(%ebx), %esi    # tmp = src[2]
27. andl %esi, %esi         # **if** tmp <= 0
28. rmmovl %esi, 4(%ecx)    # dst[2] = tmp
29. jle R1                  # skip increasing counter
30. iaddl 1, %eax           # **else** counter++
31. R1:
32. mrmovl (%ebx), %esi     # tmp = src[1]
33. andl %esi, %esi         # **if** tmp <= 0
34. rmmovl %esi, (%ecx)     # dst[1] = tmp
35. jle R0                  # skip increasing counter
36. iaddl 1, %eax           # **else** counter++
37. R0:
    * 1. Evaluation
38. Rebuild the simulator and the driver programs, test the solution in GUI mode on a small 4-element array and test your solution on a larger 63-element array.

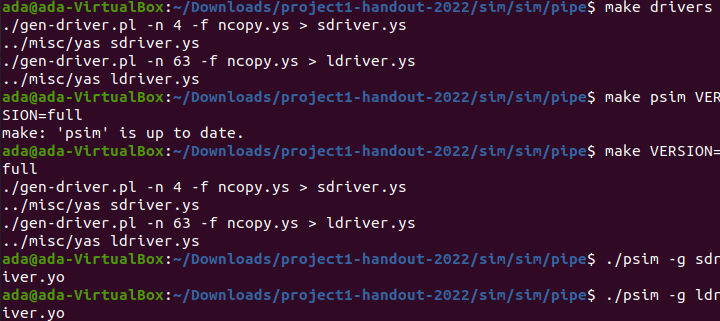


Figure 16 Part C\_Test the solution



Figure 17 Part C\_test on a 4-element array



Figure 18 Part C\_test on a 63-element array

1. Testing the pipeline simulator on the benchmark programs

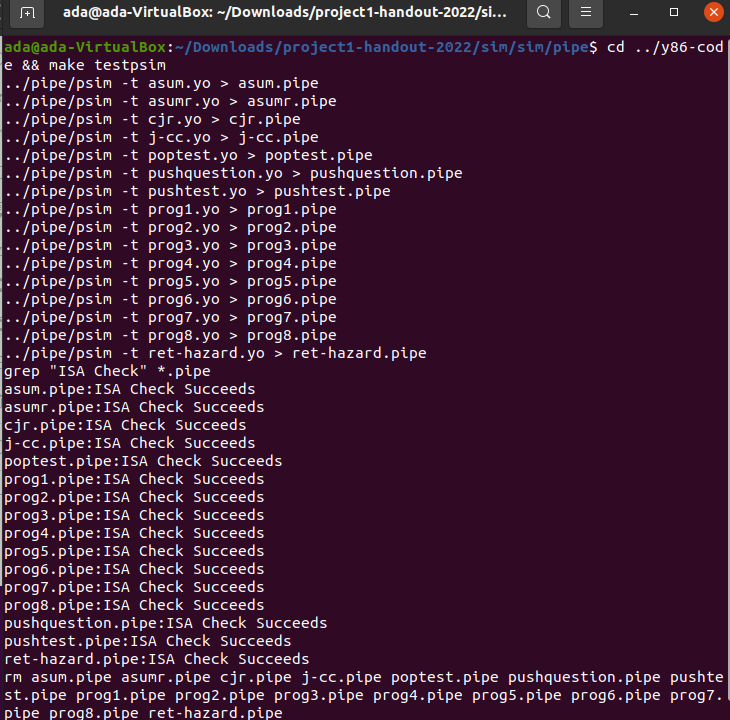


Figure 19 Part C\_benchmark test

1. Testing your pipeline simulator with extensive regression tests

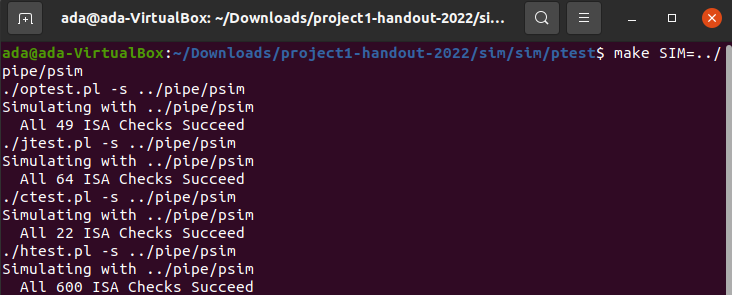


Figure 20 Part C\_regression test

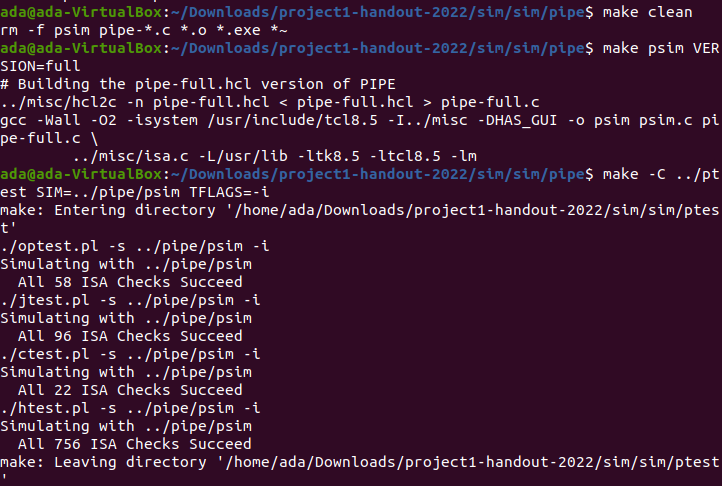


Figure 21 Part C\_regression test (with iaddl)

1. Testing your code on a range of block lengths with the pipeline simulator

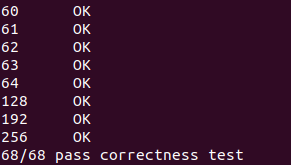


Figure 22 Part C\_correctness test

1. CPE test

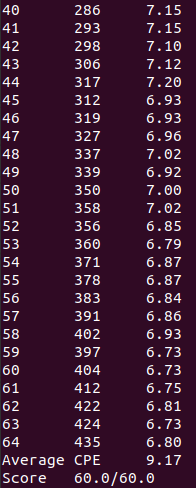


Figure 23 CPE test

1. Conclusion
   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.]

* 1. 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.]