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

519021910095 郭倩昀

519021910575 潘沐晨 [pmc296289396@sjtu.edu.cn](mailto:pmc296289396@sjtu.edu.cn)

May 9 2021

1. Introduction

In this project, our group took a glimpse into the Y86 assembly language. We learned the basic knowledge of the Y86 tools and in Part A we transferred three functions about linked list in example.c into Y86 code, which enabled us to get more familiar with the Y86 assembly language. Then in Part B, we added the iaddl instruction and the leave instruction to Y86’s sequential design by modifying the HCL file. Finally, in Part C, we improved and optimized the performance of the pipeline processor.

Our two group members all made contributions into the code of part A and the report. Student Guo Qianyun finished the code of part C while student Pan Muchen finished the code of part B.

1. Experiments
   1. Part A
      1. Analysis

In this part, we are asked to work in directory sim/misc, implementing and simulating three Y86 programs. This part of project is relatively simple and suitable for first leaners to get start. When we firstly get in touch with Y86 tools, we are not familiar with this assembly language. But after searching a lot of reference materials and continued attempt, we can finally learn how to write programs with Y86 and implement the three functions successfully.

Difficult point

* Get familiar with Y86 assembly language.
* Understand the meaning of every instructions and the function of every registers we used.
* Understand the process of memory and registers state changing, as well as how variables transmitted.
* Design how to choose the correct element from the stack.

Code technique

* Divide the program into different functional areas with enough and clear label.
* Coding the C language line by line into Y86 assembly language.
* Track on the change of stack, registers and memory to ensure the correctness of fetching a variable.
  + 1. Code

Before we start, we firstly learned the meaning of all Y86 instructions, which is shown in the bellow table1.

Table 1 the meaning of Y86 instructions

|  |  |
| --- | --- |
| instruction | meaning |
| halt | Termination of instruction execution |
| nop | A space occupying instruction, not do anything |
| irmovl | Move an immediate number into a register |
| rrmovl | Move the data in one register into another register |
| rmmovl | Move the data in register to memory |
| mrmovl | Move the data in memory to register |
| opl | Operating instruction, like addition and substract |
| jxx | Jump instruction, ‘xx’ is the jump condition |
| cmovxx | Conditional transfer instruction, only occurs between two registers |
| call & ret | Invoke function & return |
| push & pop | Enter the stack & leave the stack |

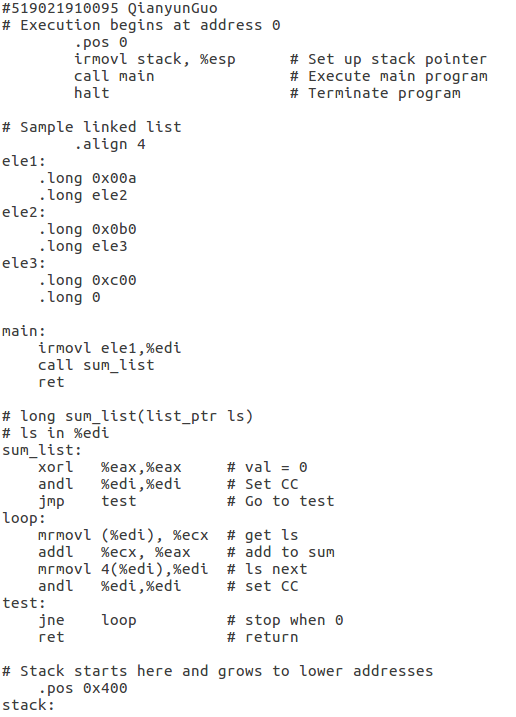
After comprehending the meaning of Y86 instructions, we can decode the given three functions into Y86 programs.

**sum.ys: Iteratively sum linked list elements**

In this part we should write a Y86 program that iteratively sums the elements of a linked list.

The code of sum.ys is as bellow. The program includes setting up the stack structure, invoking functions and then halt. We use register %eax to save the sum of linked list elements, register %edi to point to the current element and register %ecx to temporarily store the read element value. When the main function is called, we initialize the %edi register and then call function sum\_list.

In every cycle in the loop of sum\_list, we load the value of elements from memory according to the %edi register, add the value into %eax register and then update %edi register. The test part is to determine when to stop loop. If loop stops, the function returns.



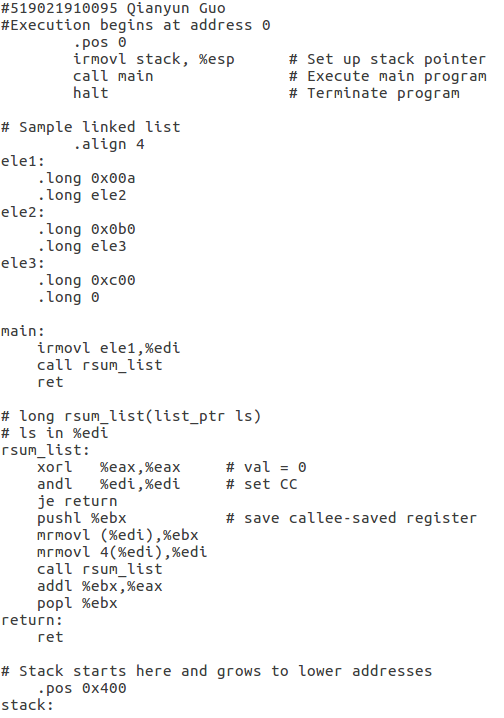
**rsum.ys: Recursively sum linked list elements**

In this part we are asked to write a Y86 program that recursively sums the elements of a linked list.

The program also includes setting up the stack structure, invoking functions and then halt. Similarly, We use register %eax to save the sum of linked list elements, register %edi to point to the current element and register %ebx to temporarily store the read element value, and how to fetch the correct element in stack is the same as the previous sum.ys program.

Specially, to implement recursion, we call rsum\_list in the rsum\_list function. The key is to use popl and pushl instruction to save callee-saved register %ebx. Every time in rsum\_list function, we need push the read value into stack and after the current function calling finished, pop the value and add it with the return value of the called function. The result is the return value of this function.

The code of rsum.ys is as bellow.



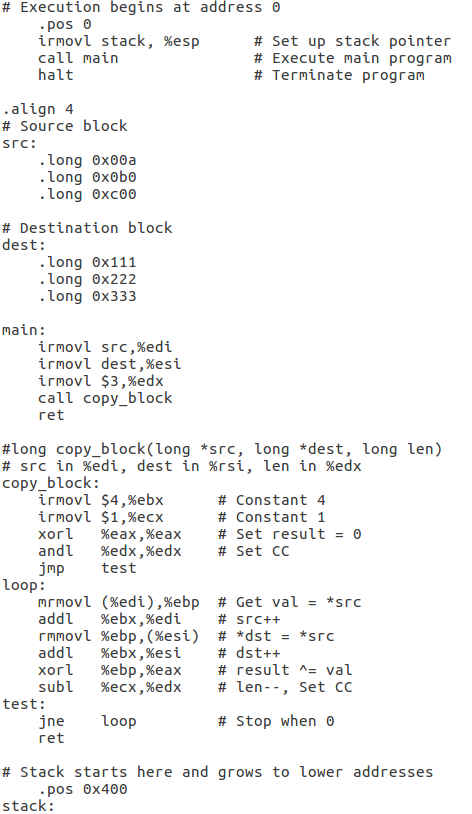
**copy.ys: Copy a source block to a destination block**

In this part we are supposed to write a Y86 program that copies a block of words from on part of memory to another which is not overlapped with the former part or memory. Meanwhile, compute the checksum (Xor) of all word copied.

The program also includes setting up the stack structure, invoking functions and then halt. We use register %eax to save the checksum of copied words, use register %edi and %esi to point to the source block and the destination block respectively, and register %ebp to temporarily store the read value from source block.

In the loop of copy\_block, we fetch word from source block according to register %edi and store the word into %ebp. Then update the value of %edi to next word in source block. Afterwards, we move the value in %ebp to the destination block in memory according to %esi and then update the %esi to the next. What’s more, we calculate the checksum in the loop as well.

It’s worth mentioning that Y86 instruction set don’t support the direct calculation between immediate and register. Thus, in program we use irmovl instruction to save constant into register %ebx and %ecx, then calculate between two reigsters.

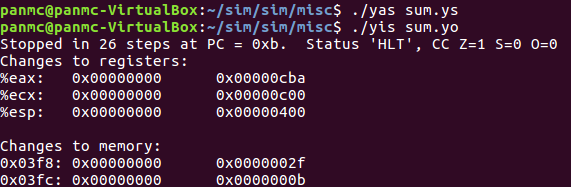


* + 1. Evaluation

**sum.ys**

Input “./yas sum.ys”, we can get the executable file sum.yo.

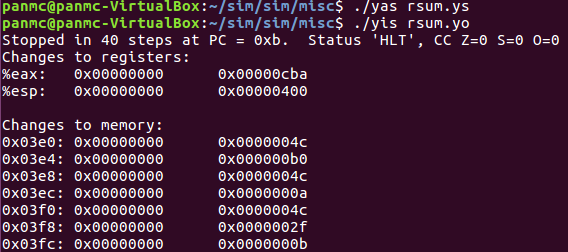
Then input “./yis sum.ys” to execute the file and get the results.



The %eax register has the correct value 0xcba which is the sum of the sample elements.

**rsum.ys**

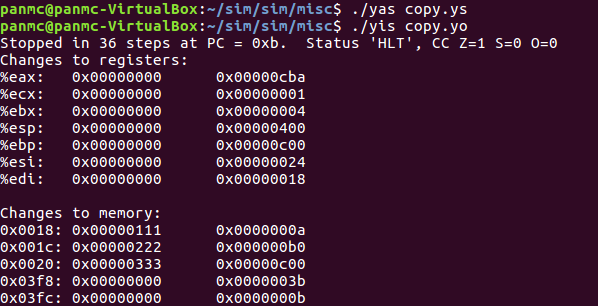
Through the same evaluation process, we can get the execution results.



By recursively sums the elements, the %eax register also has the correct value 0xcba.

**copy.ys**

Bellow is the execution results of copy.ys.



We successfully moved the word 0x00a, 0x0b0 and 0xc00 in the source block to the destination block which previously contains 0x111, 0x222 and 0x333. And the checksum in %eax is the correct value 0xcba.

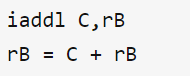
* 1. Part B
     1. Analysis

In part B, we are asked to extend the SEQ processor to support instruction iaddl and leave by modifying file seq-full.hcl. It’s important for this task to understand the implementation of HCL file and the data path of the two new instructions.

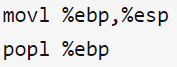
Difficult point

* Understand the processing logic and the syntax of HCL.
* Design the data path of the iaddl instruction and the leave instruction.
  + 1. Code

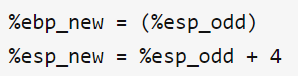
The function of iaddl is to add a constant value to a register. To achieve this, we should firstly use irmovl to move the constant value to another register then use addl to add the value to the destination register. The format of iaddl is as below.



The instruction leave writes the value in top of stack register %ebp to register %esp. The format of it is as bellow.



We can translate this instruction into:



The stage division of these two instructions is shown in the table 2.

Table 2 the stage division of iaddl and leave

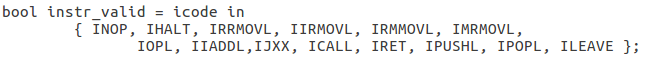
|  |  |  |
| --- | --- | --- |
| stage | iaddl | leave |
| fetch |  |  |
|  |  |
|  |  |
|  |  |
| Decode |  |  |
|  |  |
| Execute |  |  |
| Set CC |  |
| Memory |  |  |
| Write back |  |  |
|  |  |
| PC update |  |  |

According to the table, we can modify the seq-full.hcl file.

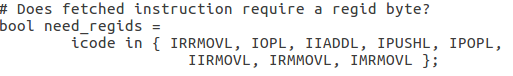
Firstly, define ILEAVE as symbolic representation of leave instruction codes.



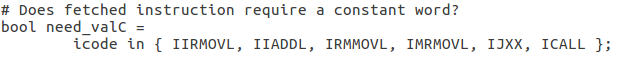
Add IIADDL and ILEAVE in the choice region instr\_valid to make them valid.



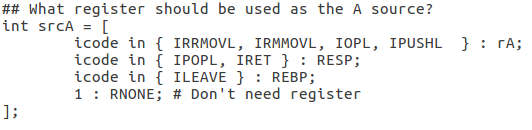
Add IIADDL in the choice region of need\_regids since iaddl operation involves one register. The leave instruction doesn’t need it.



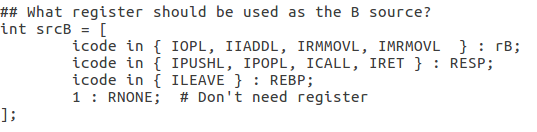
Add IIADDL in the choice region of need\_valC since iaddl operation need constant parameter. The leave instruction doesn’t need it.



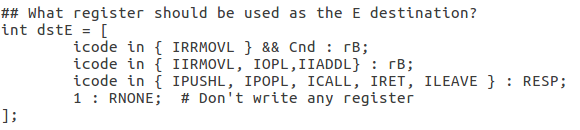
When icode is ILEAVE, set the srcA as REBP. For iaddl, there is no need to set because the first operand of iaddl is not a register.



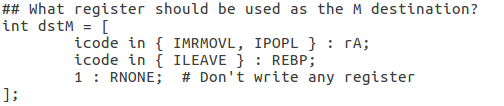
When icode is IIADDL, set the srcB as rB. When icode is ILEAVE, set the srcB as REBP.



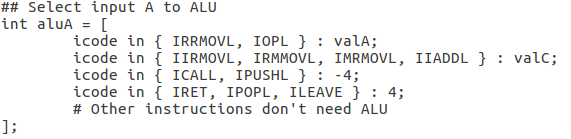
When icode is IIADDL, set the dstE as rB. When icode is ILEAVE set the dstE as RESP. Because iaddl adds the immediate number into the destination register and leave writes value into register %esp.



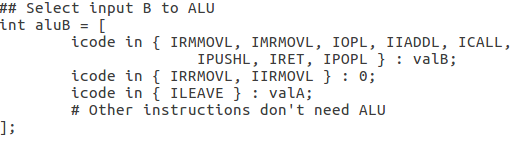
When icode is ILEAVE, set the dstM as REBP because we also need to update the value in register %ebp. For IIADDL, there is no need to set.



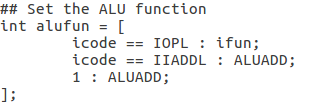
When icode is IIADDL, set aluA as valC. When icode is ILEAVE, set aluA as 4. It’s the first operand of ALU.



When icode is IIADDL, set aluB as valB. When icode is ILEAVE, set aluB as valA. It’s the second operand of ALU.



When icode is IIADDL, alufun will be ALUADD since the operation is “adding”.



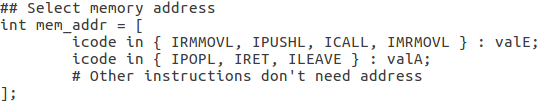
Add IIADDL in the choice region of set\_cc since iaddl operation involves ALU operation which will set flags. For leave, we don’t need it.



When icode is ILEAVE, set the control signal mem\_read. Instruction iaddl doesn’t need access to memory.



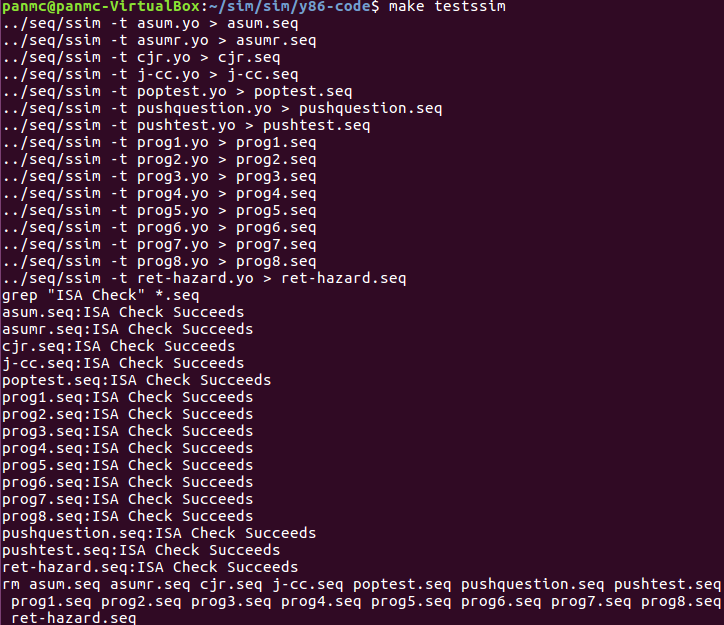
When icode is ILEAVE, set mem\_addr as valA. For iaddl, we don’t need it.



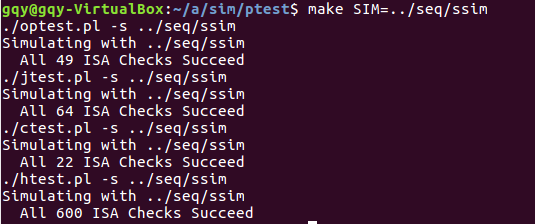
* + 1. Evaluation

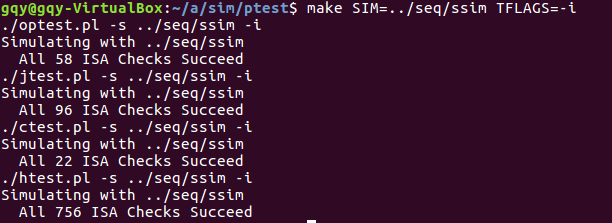
Test the implementation on the Y86 benchmark programs in directory y86-code. The result is as bellow.

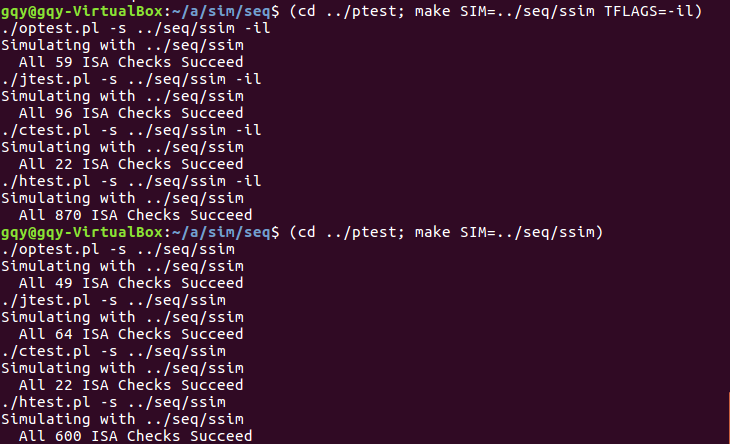
The result shows the test on benchmark programs is success.



Then we can run the extensive set of regression tests in directory ptest. The following results is test except iaddl and leave, test iaddl and test leave respectively. The results show the test on regression is success.







* 1. Part C
     1. Analysis
     2. Code
     3. Evaluation

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.]