Project 1: Optimizing the Performance of a Pipelined Processor

000, x, x-email
001, y, y-email
002, z, z-email

April 29, 2018

1 Introduction

[In this section you should briefly introduce the task in your own words, and what you've done in this project. A simple copy from project1.pdf is not permitted.]

[You should also list the arrangement of each member here. For example, you can write, "student-x finished part A and B, student-y finished part C and student-z finished the report" (of course we suggest each student to make contributions to coding tasks.)]

2 Experiments

The experiment includes 3 parts.

2.1 Part A

2.1.1 Analysis

- **sum.ys** is a program that iteratively sums the elements of a linked list.

  The basic idea is that we use a conditional jump in a loop which iteratively check whether the next element is equal to zero and if not add up the value to the sum.

  - In init part, the stack structure is set up, then the program jumps to Main function, and finally halts.
  - In Main, we first store the first element to the stack before a call to function sum_list.

- In sum_list function, we first do the conventions which saves a copy of initial %ebp and set %ebp to the beginning of the stack frame. Then we initialize the sum=0, and the go to a loop which iteratively add up elements' value into our sum.
- In loop: firstly, the element pointed to is added and then, we increment the pointer address which make it points to the next element. If the next element is equal to zero, jump to done, otherwise loop agian.
- In done: we resume the %esp and %ebp to the initial value set in init part. Then we can safely let Main function return.

- **rsum.ys** is a program that recursively sums the elements of a linked list.

  This most of the 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.

  - In rsum_list, the key idea is that we use %eax to store the iterative temporary sum meanwhile store the value of the current element in %edx. Also, a very important point is that we should store the address of the next element(if it is not zero) always in 8(%ebp), such that in every recursive step, we always update the desired element, which in this case we update $element[i+1]$ with the sum of all elements from $i+1$ to the end.

- **copy.ys** copies a block of words from one part of memory to another (non-overlapping area) area of memory, computing the checksum (Xor) of all the words copied.

  - The initialization step is similar to the above implementations.
  - In Main: firstly, the store the src, dest and len into main function stack frame for future use. After these preliminaries, copy block function is called. After returning from copy block, we need to resume the esp and ebp to the initial value set in init part and this is done by "done" function part as similar to above implementations. Finally Main function is returned.
  - copy_block: In copy block, firstly we do the conventions like saving a copy of caller's ebp and set ebp to the beginning of copy block 's stack frame. Then we use 3 registers %ebx %ecx, %esi to store temporary needed values for iteration. Also, %eax, the stored length, is subtracted by 1 and a conditional jump instruction was added to terminated the loop when the length is equal to zero. In each iteration, we copy the value stored in the current source block to the current destination block. The addresses of both is calculated by a increment factor %esi added to the current address(%edx for src and %ebc for dest).

    Finally, we resume the esp and ebp and return.

2.1.2 Code

- **sum.ys**

```
#Execution begins at address 0
     .pos     0           #start address for all Y86 programs
Init:
     irmovl   Stack, %esp      #Initialize stack pointer
     irmovl   Stack, %ebp      #Initialize base pointer
     jmp Main
     halt

.align  4
ele1:                     #Elements initialization
         .long 0x00a
         .long ele2
ele2:
         .long 0x0b0
         .long ele3
ele3:
         .long 0xc00
         .long 0

Main:
         irmovl   ele1,%esi    #starting pointer
         pushl    %esi
         call sum_list
         halt

sum_list:
         pushl    %ebp
         rrmovl   %esp, %ebp   #read the stack pointer
         mrmovl   8(%ebp),%ebx     #ebx = start pointer ele1
         irmovl   $0,%eax      #sum=0
loop:        mrmovl  (%ebx),%edx #The element
         addl     %edx,%eax
         mrmovl   4(%ebx), %esi    #4(%ebx) is address of next node
         andl     %esi, %esi   #if %esi=zero,jump to done
         je       done     #If the pointer points to zero, return
         rrmovl   %esi,%ebx
         jmp      loop
done:        popl     %esi          #restore the registers
         popl     %edx
         popl     %ebx
         rrmovl   %ebp, %esp
         popl     %ebp
         ret
#stack starts here and grows to lower addresses
         .pos     0x400
Stack:


```

- **rsum.ys**

```
 #Execution begins at address 0
     .pos     0
```

```
Init:
    irmovl  Stack, %esp      #Initialize stack pointer
    irmovl  Stack, %ebp
    jmp     Main
    halt

.align  4
ele1:
        .long 0x00a
        .long ele2
ele2:
        .long 0x0b0
        .long ele3
ele3:
        .long 0xc00
        .long 0


Main:
        irmovl  ele1,%esi   #p_ele1
        pushl   %esi
        xorl    %eax, %eax  #set eax=0
        call rsum_list
        halt

rsum_list:
        pushl   %ebp
        rrmovl  %esp, %ebp  #read the stack pointer
        pushl   %ebx        #save ebx
        pushl   %ecx        #save ecx
        pushl   %edx        #save edx
        pushl   %esi        #save esi
        mrmovl  8(%ebp),%edx    #edx=p_ele[i]
        mrmovl  0(%edx),%eax    #eax=ele[i]
        mrmovl  4(%edx),%ebx    #ebx=p_ele[i+1]
        andl    %ebx, %ebx  #if p_ele[i+1] == 0
        je      done        #return ele[i]
        pushl   %ebx        #else: 8(%ebp)=p_ele[i+1]
        rrmovl  %eax, %ecx  #ecx = ele[i]
        call    rsum_list
        popl    %edx        #restore the stack pointer
        addl    %ecx,%eax   #eax += rsum(p_ele[i+1])
done:                     #return
        popl    %esi            #restore the registers
        popl    %edx
        popl    %ecx
        popl    %ebx
        rrmovl  %ebp, %esp
        popl    %ebp
        ret

        .pos    0x120
Stack:
```

- **copy.ys**

```
#Execution begins at address 0
```

```
    .pos    0
Init:
    irmovl  Stack, %esp         #Initialize stack pointer
    irmovl  Stack, %ebp
    jmp Main
    halt

.align  4
src:
    .long   0x00a
    .long   0x0b0
    .long   0xc00
dest:
    .long   0x111
    .long   0x222
    .long   0x333


Main:
        irmovl  src,%esi        #src
        pushl   %esi
        irmovl  dest,%esi       #dest
        pushl   %esi
        irmovl  $3,%esi         #len
        pushl   %esi
        call copy_block
        halt

copy_block:
        pushl   %ebp
        rrmovl  %esp, %ebp      #read the stack pointer
        pushl   %ebx            #save ebx
        pushl   %ecx            #save ecx
        pushl   %edx            #save edx
        pushl   %esi            #save esi
        mrmovl  8(%ebp),%eax        #eax=len,len-1,...,0
        irmovl  $0,%ebx         #tmp=0
        irmovl  $0,%ecx         #ecx=0
        irmovl  $0,%esi         #esi=0,4,8...
loop:
        mrmovl  16(%ebp),%edx       #edx = p_src
        addl    %esi,%edx       #edx = p_src_cur
        mrmovl  0(%edx),%edx        #edx = src_cur
        xorl    %edx,%ecx       #result ^= src_cur

        mrmovl  12(%ebp),%ebx       #ebx = p_dest
        addl    %esi,%ebx       #ebx = p_dest_cur
        rmmovl  %edx,0(%ebx)        #*p_dest_cur = src_cur

        irmovl  $1,%ebx         #eax-=1
        subl    %ebx,%eax       #subl   %ebx,%eax -> eax = eax - ebx
        je      done
        irmovl  $4,%ebx         #tmp = 4
        addl    %ebx,%esi       #esi+=tmp
        jmp     loop
done:       rrmovl  %ecx,%eax
```

```
        popl    %esi            #restore the registers
        popl    %edx
        popl    %ecx
        popl    %ebx
        rrmovl  %ebp, %esp
        popl    %ebp
        ret

        .pos    0x120
Stack:
```

2.1.3 Evaluation

- **sum.ys**

  figureeeeeeeeee

- **rsum.ys**

  figureeeeeeeeee

- **copy.ys**

  figureeeeeeeeee

[In this part, you should place the figures of experiments for your codes, prove the correctness and validate the performance with your own words for each figure's explanation.]

2.2 Part B

An operation *iaddl* added to the control file *seq−full.hcl* to extend the SEQ processor is required in this part.

2.2.1 Analysis

To add *iaddl* to the SEQ processor, the steps is as follows:

1. $M_1[PC]$ is used to get the icode and ifun which conbine a byte.
2. we need to get which register we begi to use, and we can use M 1[P C + 1] to get the second byte which contains two registers tags. Thirdly, we get the rest of the instruction to get the instant value.
3. Decode the instruction by which we could get the value in the register and store it in the valB.
4. Execute the add operation.
5. Write the result back to the register and Finally update the PC to prepare for the next instruction.

2.2.2 Code

2.2.3 Evaluation

[In this part, you should place the figures of experiments for your codes, prove the correctness and validate the performance with your own words for each figure's explanation.]

2.3 Part C

2.3.1 Analysis

[In this part, you should give an overall analysis for the task, like difficult point, core technique and so on.]

2.3.2 Code

[In this part, you should place your code and make it readable in Microsoft Word, please. Writing necessary comments for codes is a good habit.]

2.3.3 Evaluation

[In this part, you should place the figures of experiments for your codes, prove the correctness and validate the performance with your own words for each figure's explanation.]

3 Conclusion

3.1 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.]

3.2 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.]