Project2: Optimizing the Performance of a Pipelined Processor
CS 359, COMPUTER ARCHITECTURE, Yanyan Shen, Spring 2017

1 Part C

1.1 Description

- The program **ncopy.ys** copies a len-element integer array *src* to a non-overlapping *dst*, returning the count number of positive integers contained in *src*.
- In this part, we need to minimize the running time of **ncopy.ys** as less as possible. As we have learned in the class, the *jump* instructions (including *jmp*, *jle*, *jl*, *je*, *jne*, *jge* and *jg* in Y86 instructions). Of course we can write one loop to solve this problem, but it costs to much time. So we use some other methods to solve this problem.

1.2 Solution

ncopy.ys

```
#/* $begin ncopy-ys */
##################################################################
# ncopy.ys - Copy a src block of len ints to dst.
# Return the number of positive ints (>0) contained in src.
#
# Include your name and ID here.
#
# Describe how and why you modified the baseline code.
#
##################################################################
# Do not modify this portion
# Function prologue.
ncopy:  pushl %ebp              # Save old frame pointer
        rrmovl %esp,%ebp        # Set up new frame pointer
        pushl %esi              # Save callee-save regs
        pushl %ebx
        pushl %edi
        mrmovl 8(%ebp),%ebx     # src
        mrmovl 16(%ebp),%edx    # len
        mrmovl 12(%ebp),%ecx    # dst
##################################################################
        xorl  %eax, %eax        # initialize the count to 0

##################################################################
Loop8:
        iaddl $-8,  %edx        # len = len - 8
        andl  %edx, %edx        # to see if the len is less than 0
        jl Loop4

        mrmovl (%ebx), %esi
        mrmovl 4(%ebx), %edi
```

```
        rmmovl %esi, (%ecx)
        andl %esi, %esi
        jle StageOneOf8
        iaddl $1, %eax

StageOneOf8:
        rmmovl %edi, 4(%ecx)
        andl %edi, %edi
        jle StageTwoOf8
        iaddl $1, %eax

StageTwoOf8:
        mrmovl 8(%ebx), %esi
        mrmovl 12(%ebx), %edi
        rmmovl %esi, 8(%ecx)
        andl %esi, %esi
        jle StageThreeOf8
        iaddl $1, %eax

StageThreeOf8:
        rmmovl %edi, 12(%ecx)
        andl %edi, %edi
        jle StageFourOf8
        iaddl $1, %eax

StageFourOf8:
        mrmovl 16(%ebx), %esi
        mrmovl 20(%ebx), %edi
        rmmovl %esi, 16(%ecx)
        andl %esi, %esi
        jle StageFiveOf8
        iaddl $1, %eax

StageFiveOf8:
        rmmovl %edi, 20(%ecx)
        andl %edi, %edi
        jle StageSixOf8
        iaddl $1, %eax

StageSixOf8:
        mrmovl 24(%ebx), %esi
        mrmovl 28(%ebx), %edi
        rmmovl %esi, 24(%ecx)
        andl %esi, %esi
        jle StageSevenOf8
        iaddl $1, %eax

StageSevenOf8:
        rmmovl %edi, 28(%ecx)
        andl %edi, %edi
```

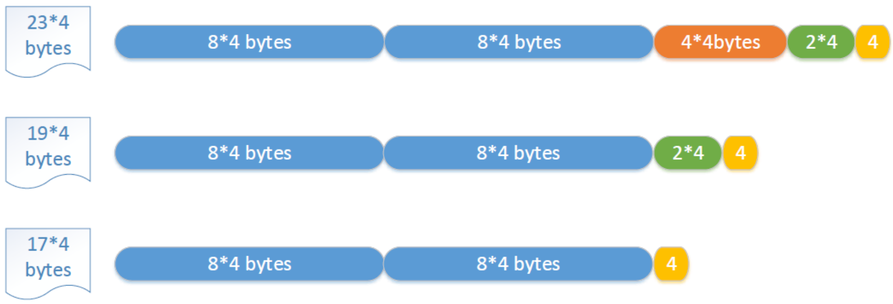
```
        jle Forward8
        iaddl $1, %eax

Forward8:
        iaddl $32, %ebx
        iaddl $32, %ecx
        jmp Loop8

##########################################
Loop4:
        iaddl $8, %edx
        iaddl $-4, %edx
        andl  %edx, %edx
        jl Loop2

        mrmovl (%ebx), %esi
        mrmovl 4(%ebx), %edi
        rmmovl %esi, (%ecx)
        andl %esi, %esi
        jle StageOneOf4
        iaddl $1, %eax

StageOneOf4:
        rmmovl %edi, 4(%ecx)
        andl %edi, %edi
        jle StageTwoOf4
        iaddl $1, %eax

StageTwoOf4:
        mrmovl 8(%ebx), %esi
        mrmovl 12(%ebx), %edi
        rmmovl %esi, 8(%ecx)
        andl %esi, %esi
        jle StageThreeOf4
        iaddl $1, %eax

StageThreeOf4:
        rmmovl %edi, 12(%ecx)
        andl %edi, %edi
        jle Forward4
        iaddl $1, %eax

Forward4:
        iaddl $16, %ebx
        iaddl $16, %ecx
        iaddl $-4, %edx
        jmp Loop2

##########################################
Loop2:
```

```
        iaddl $4, %edx
        iaddl $-2, %edx
        andl %edx, %edx
        jl Loop1

        mrmovl (%ebx), %esi
        mrmovl 4(%ebx), %edi
        rmmovl %esi, (%ecx)
        andl %esi, %esi
        jle StageOneOf2
        iaddl $1, %eax

StageOneOf2:
        rmmovl %edi, 4(%ecx)
        andl %edi, %edi
        jle Forward2
        iaddl $1, %eax

Forward2:
        iaddl $8, %ebx
        iaddl $8, %ecx
        iaddl $-2, %edx
        jmp Loop1

##########################################
Loop1:
        iaddl $2, %edx
        iaddl $-1, %edx
        andl %edx, %edx
        jl Done

        mrmovl (%ebx), %esi
        rmmovl %esi, (%ecx)
        andl %esi, %esi
        jle Done
        iaddl $1, %eax
        jmp Done
##########################################
# Do not modify the following section of code
# Function epilogue.
Done:
        popl %edi               # Restore callee-save registers
        popl %ebx
        popl %esi
        rrmovl %ebp, %esp
        popl %ebp
        ret
##########################################
# Keep the following label at the end of your function
End:
```

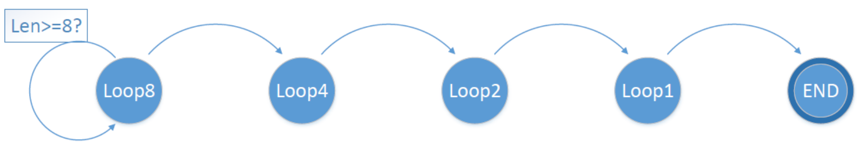
```
#/* $end ncopy-ys */
```

1.3 Analysis

1.3.1 sum.ys



- To explain our solution clearly, we provide the above picture to express the overview of our solution.
- Our main idea is: In each copy instruction, we copy 4 *bytes*. Firstly, we copy each $8 * 4$ *bytes* in a loop until the left data needed to copy is less than $8 * 4$ *bytes*. Secondly, we check if the left data is more or equal to $4 * 4$ *bytes*. If so, we copy $4 * 4$ *bytes* in this step. Thirdly, we check if the left data is more or equal to $2 * 4$ *bytes*. If so, we copy $2 * 4$ *bytes* in this step. Finally, we check if there are still $1 * 4$ *bytes* left. If so, we copy these 4 *bytes* and end the problem. Since we need to return the count number of positive integers contained in *src*, we will do a judge after each copy instruction to decided if we need to increase the count by 1.
- For the registers, we define *esi* and *edi* to store the value of contiguous 2 words(4 bytes each word). We define *ebx* to store the address of current *src* and *dst* to store the address of current *dst*. We define *edx* to store the *len* of left words needed to copy. We define *eax* to store the count of positive integers in our copyed data.
- For the Loops, we define 4 main blocks: *Loop*8, *Loop*4, *Loop*2 and *Loop*1. However, only the *Loop*8 is the "real" Loop which means only *Loop*8 can be excuted more than 1 time. In the above picture, blue means *Loop*8, oringe means *Loop*4, green means *Loop*2 and yellow means *Loop*1. We provide following picture to represent the relations among these Loop.



- We take *Loop*4 for example. First we plus 4 to *edx* and check if it is less than 0. If it is, the program will jump to *Loop*2. Else, we do 4 copy instructions in *Loop*4. The register *esi* and *edi* stores the current data we need to copy. After each copy instruction, we will check if the data is a positive integer. If it is, the register *eax* will be added by 1. Else, the problem will ignore the add instruction and jump to next copy instruction.
- Assume we need to copy n words and n is a very large number. Assume the count number of positive integers contained in *src* is m. In our progress, we need $O(log_8^n + m)$ jumps.

1.4 Outcome

1.5 Review