此部分参考了网络文章https://blog.csdn.net/u012336567/article/details/51867766?ops_request_misc=&request_id=&biz_id=102&utm_term=archlab&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-1-51867766.first_rank_v2_pc_rank_v29&spm=1018.2226.3001.4187

实验之前

按照说明安装yas和yis错误

```
yum install flex
yum install flex-devel
yum install tk
yum install tcl
```

PartA

```
/*
 * Architecture Lab: Part A
 *
 * High level specs for the functions that the students will rewrite
 * in Y86-64 assembly language
 */

/* $begin examples */
/* linked list element */
typedef struct ELE {
    long val;
    struct ELE *next;
} *list_ptr;

/* sum_list - Sum the elements of a linked list */
long sum_list(list_ptr ls)
{
    long val = 0;
    while (ls) {
        val += ls->val;
        ls = ls->next;
    }
    return val;
}
```

```
25
26 /* rsum_list - Recursive version of sum_list */
27 long rsum_list(list_ptr ls)
28 {
29     if (!ls)
30         return 0;
31     else {
32         long val = ls->val;
33         long rest = rsum_list(ls->next);
34         return val + rest;
35     }
36 }
37
38 /* copy_block - Copy src to dest and return xor checksum of src */
39 long copy_block(long *src, long *dest, long len)
40 {
41     long result = 0;
42     while (len > 0) {
43         long val = *src++;
44         *dest++ = val;
45         result ^= val;
46         len--;
47     }
48     return result;
49 }
50 /* $end examples */
```

打开example.c，易得ELE结构是链表，sum_list()是链表求和，rsum_list(list_ptr ls)是递归链表求和，copy_block(long *src, long *dest, long len)是复制数组。

sum.ys

```
1 #Execution begins at address 0,written by peanwang仿照csapp p254编写
2         .pos 0
3         irmovq stack,%rsp
4         call main
5         halt
6 # Sample linked list从实验说明获取测试数据
7         .align 8
8 ele1:
```

```
        .quad 0x00a
        .quad ele2
ele2:
        .quad 0x0b0
        .quad ele3
ele3:
        .quad 0xc00
        .quad 0
#This is main function ele1为链表头作第一个参数
main:
        irmovq ele1,%rdi
        call sum_list
        ret
#long sum_list(list_ptr ls)

sum_list:
        irmovq $0,%rax #将返回值初始化为0
        jmp test

loop:
    mrmovq 0(%rdi),%rsi
    addq %rsi,%rax #val += ls->val;
    mrmovq 8(%rdi),%rsi
    rrmovq %rsi,%rdi #ls = ls->next;

#test为跳转条件判断
test:
    andq %rdi,%rdi #因为链表尾地址为0
    jne loop #jne是一个条件转移指令。不相等或非0则跳转，即等效于while (ls)的效果
    ret



#stack starts here and grows to lower addresses
        .pos 0x200
stack:

```

rax中即为返回结果cba，与正确结果一致

```
[xze@localhost misc]$ ./yas ./sum.ys
[xze@localhost misc]$ ./yis sum.yo
Stopped in 29 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:    0x0000000000000000      0x0000000000000cba
%rsp:    0x0000000000000000      0x0000000000000200

Changes to memory:
0x01f0:  0x0000000000000000      0x000000000000005b
0x01f8:  0x0000000000000000      0x0000000000000013
[xze@localhost misc]$
```

rsum.ys

```
#Execution begins at address 0,written by peanwang仿照csapp p254编写
        .pos 0
        irmovq stack,%rsp
        call main
        halt
# Sample linked list从实验说明获取测试数据
        .align 8
ele1:
        .quad 0x00a
        .quad ele2
ele2:
        .quad 0x0b0
        .quad ele3
ele3:
        .quad 0xc00
        .quad 0
#This is main function ele1为链表头作第一个参数
main:
        irmovq ele1,%rdi
        call rsum_list
        ret
#long sum_list(list_ptr ls)

rsum_list:
        pushq %r14
        irmovq $0, %rax//返回值初始化为0
        andq %rdi,%rdi//判断返回条件，即是否为最后一个元素
        je return
        mrmovq 0(%rdi), %r14 //r14存储结点值
```

```
        mrmovq 8(%rdi), %rdi //存储结点地址
        call rsum_list
        addq %r14, %rax

return:
        popq %r14
        ret


#stack starts here and grows to lower addresses
        .pos 0x200
stack:
```

测试结果正确，思路就是模仿递归，递归就是在条件成立情况下不断入栈，然后返回局部的结果

```
[xze@localhost misc]$ ./yas ./rsum.ys
[xze@localhost misc]$ ./yis rsum.yo
Stopped in 42 steps at PC = 0x13.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%rax:    0x0000000000000000      0x0000000000000cba
%rsp:    0x0000000000000000      0x0000000000000200

Changes to memory:
0x01b8:  0x0000000000000000      0x0000000000000c00
0x01c0:  0x0000000000000000      0x0000000000000090
0x01c8:  0x0000000000000000      0x00000000000000b0
0x01d0:  0x0000000000000000      0x0000000000000090
0x01d8:  0x0000000000000000      0x000000000000000a
0x01e0:  0x0000000000000000      0x0000000000000090
0x01f0:  0x0000000000000000      0x000000000000005b
0x01f8:  0x0000000000000000      0x0000000000000013
[xze@localhost misc]$
```

copy.ys

```
#Execution begins at address 0,written by peanwang仿照csapp p254编写
        .pos 0
        irmovq stack, %rsp
        call main
        halt
    .align 8
# Source block
src:
        .quad 0x00a
```

```
        .quad 0x0b0
        .quad 0xc00
# Destination block
dest:
        .quad 0x111
        .quad 0x222
        .quad 0x333

main:
        irmovq src, %rdi//原链表
        irmovq dest, %rsi//目的链表
        irmovq $3, %rdx//rdx存储链表长度len
        call copy_block//调用
        ret


copy_block:
        pushq %r12
        pushq %r13
        pushq %r14              //因为y86某些操作不支持立即数，将要用的立即数保存
        irmovq $1, %r13
        irmovq $8, %r14
        irmovq $0, %rax #result初值设为0
        jmp loop_test
loop:
        mrmovq 0(%rdi), %r12  #r12作为中间变量
        addq %r14, %rdi       #src++
        rmmovq %r12, (%rsi)   #*dest = *(src - 1)
        addq %r14, %rsi       #dest++
        xorq %r12, %rax
        subq %r13, %rdx       //len--
loop_test:
        andq %rdx, %rdx   #测试len是否为0，不为0则继续循环
        jg loop
        popq %r14
        popq %r13
        popq %r12
        ret
    .pos    0x300
 Stack:
```

测试结果如下，观察到111，222，333储存内容变化

```
[xze@localhost misc]$ ./yas ./copy.ys
[xze@localhost misc]$ ./yis copy.yo
Stopped in 45 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:    0x0000000000000000      0x0000000000000cba
%rsp:    0x0000000000000000      0x0000000000000300
%rsi:    0x0000000000000000      0x0000000000000048
%rdi:    0x0000000000000000      0x0000000000000030

Changes to memory:
0x0030:  0x0000000000000111      0x000000000000000a
0x0038:  0x0000000000000222      0x00000000000000b0
0x0040:  0x0000000000000333      0x0000000000000c00
0x02f0:  0x0000000000000000      0x00000000000000ca
0x02f8:  0x0000000000000000      0x0000000000000013
[xze@localhost misc]$
```

PartB

根据说明，即添加iaddq指令，立即数与寄存器相加

参考书中图4.18，特别是OPq和irmovq两个指令，写出5个阶段

然后OPQ所在地方添加IIADDQ，然后是用到valC，并且只用一个寄存器，所以只有srcB

fetch	icode:ifun<-M1[PC] `instr_valid` rA,rB<-M1[PC+1] need_regids need_valC valC<-M1[PC+2]`need_valC` ValP<-PC+10
decode	valB<-R[rB] srcB:rb dstE:rb(opq)
execute	ValE<-ValB+ValC alu:valC,alu:valB;set_cc
memory	
writeback	R[rB]<-ValE PC<-valP

```
#/* $begin seq-all-hcl */
#this is xiaozhuoer's lab
####################################################################
#  HCL Description of Control for Single Cycle Y86-64 Processor SEQ   #
#  Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010       #
####################################################################

## Your task is to implement the iaddq instruction
```

```
## The file contains a declaration of the icodes
## for iaddq (IIADDQ)
## Your job is to add the rest of the logic to make it work

#########################################################################
#    C Include's.  Don't alter these                                    #
#########################################################################

quote '#include <stdio.h>'
quote '#include "isa.h"'
quote '#include "sim.h"'
quote 'int sim_main(int argc, char *argv[]);'
quote 'word_t gen_pc(){return 0;}'
quote 'int main(int argc, char *argv[])'
quote '  {plusmode=0;return sim_main(argc,argv);}'

#########################################################################
#    Declarations.  Do not change/remove/delete any of these            #
#########################################################################

##### Symbolic representation of Y86-64 Instruction Codes #############
wordsig INOP    'I_NOP'
wordsig IHALT   'I_HALT'
wordsig IRRMOVQ 'I_RRMOVQ'
wordsig IIRMOVQ 'I_IRMOVQ'
wordsig IRMMOVQ 'I_RMMOVQ'
wordsig IMRMOVQ 'I_MRMOVQ'
wordsig IOPQ    'I_ALU'
wordsig IJXX    'I_JMP'
wordsig ICALL   'I_CALL'
wordsig IRET    'I_RET'
wordsig IPUSHQ  'I_PUSHQ'
wordsig IPOPQ   'I_POPQ'
# Instruction code for iaddq instruction
wordsig IIADDQ  'I_IADDQ'

##### Symbolic represenations of Y86-64 function codes                  #####
wordsig FNONE    'F_NONE'        # Default function code

##### Symbolic representation of Y86-64 Registers referenced explicitly #####
wordsig RRSP     'REG_RSP'      # Stack Pointer
wordsig RNONE    'REG_NONE'     # Special value indicating "no register"
```

```
44
   ##### ALU Functions referenced explicitly                          #####
45 wordsig ALUADD  'A_ADD'         # ALU should add its arguments
46
   ##### Possible instruction status values                           #####
47 wordsig SAOK    'STAT_AOK'      # Normal execution
48 wordsig SADR    'STAT_ADR'      # Invalid memory address
49 wordsig SINS    'STAT_INS'      # Invalid instruction
50 wordsig SHLT    'STAT_HLT'      # Halt instruction encountered
51
   ##### Signals that can be referenced by control logic ####################
52
   ##### Fetch stage inputs                #####
53 wordsig pc 'pc'                         # Program counter
54 ##### Fetch stage computations          #####
55 wordsig imem_icode 'imem_icode'         # icode field from instruction memory
56 wordsig imem_ifun  'imem_ifun'          # ifun field from instruction memory
57 wordsig icode     'icode'              # Instruction control code
58 wordsig ifun      'ifun'               # Instruction function
59 wordsig rA        'ra'                 # rA field from instruction
60 wordsig rB        'rb'                 # rB field from instruction
61 wordsig valC      'valc'               # Constant from instruction
62 wordsig valP      'valp'               # Address of following instruction
63 boolsig imem_error 'imem_error'         # Error signal from instruction memory
64 boolsig instr_valid 'instr_valid'       # Is fetched instruction valid?
65
   ##### Decode stage computations         #####
66 wordsig valA    'vala'                 # Value from register A port
67 wordsig valB    'valb'                 # Value from register B port
68
   ##### Execute stage computations        #####
69 wordsig valE    'vale'                 # Value computed by ALU
70 boolsig Cnd     'cond'                 # Branch test
71
   ##### Memory stage computations         #####
72 wordsig valM    'valm'                 # Value read from memory
73 boolsig dmem_error 'dmem_error'         # Error signal from data memory
74
   ########################################################################
75 #    Control Signal Definitions.                                       #
76 ########################################################################
77
   ################ Fetch Stage     ###################################
```

```
78
    # Determine instruction code
79  word icode = [
80          imem_error: INOP;
81          1: imem_icode;          # Default: get from instruction memory
82  ];
83
    # Determine instruction function
84  word ifun = [
85          imem_error: FNONE;
86          1: imem_ifun;           # Default: get from instruction memory
87  ];
88
    bool instr_valid = icode in
89          { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
90                  IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ, IIADDQ };
91
    # Does fetched instruction require a regid byte?
92  bool need_regids =
93          icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
94                      IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ };
95
    # Does fetched instruction require a constant word?
96  bool need_valC =
97          icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL, IIADDQ };
98
    ################ Decode Stage    ###################################
99
    ## What register should be used as the A source?
100 word srcA = [
101         icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : rA;
102         icode in { IPOPQ, IRET } : RRSP;
103         1 : RNONE; # Don't need register
104 ];
105
    ## What register should be used as the B source?
106 word srcB = [
107         icode in { IOPQ, IRMMOVQ, IMRMOVQ, IIADDQ  } : rB;
108         icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
109         1 : RNONE;  # Don't need register
110 ];
111
    ## What register should be used as the E destination?
```

```
word dstE = [
        icode in { IRRMOVQ } && Cnd : rB;
        icode in { IIRMOVQ, IOPQ, IIADDQ} : rB;
        icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't write any register
];

## What register should be used as the M destination?
word dstM = [
        icode in { IMRMOVQ, IPOPQ } : rA;
        1 : RNONE;  # Don't write any register
];

################ Execute Stage   ###################################

## Select input A to ALU
word aluA = [
        icode in { IRRMOVQ, IOPQ } : valA;
        icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ } : valC;
        icode in { ICALL, IPUSHQ } : -8;
        icode in { IRET, IPOPQ } : 8;
        # Other instructions don't need ALU
];

## Select input B to ALU
word aluB = [
        icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
                      IPUSHQ, IRET, IPOPQ, IIADDQ } : valB;
        icode in { IRRMOVQ, IIRMOVQ } : 0;
        # Other instructions don't need ALU
];

## Set the ALU function
word alufun = [
        icode == IOPQ : ifun;
        1 : ALUADD;
];

## Should the condition codes be updated?
bool set_cc = icode in { IOPQ, IIADDQ };

```

```
################ Memory Stage    ###################################

## Set read control signal
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET, IIADDQ };

## Set write control signal
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };

## Select memory address
word mem_addr = [
        icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
        icode in { IPOPQ, IRET } : valA;
        # Other instructions don't need address
];

## Select memory input data
word mem_data = [
        # Value from register
        icode in { IRMMOVQ, IPUSHQ } : valA;
        # Return PC
        icode == ICALL : valP;
        # Default: Don't write anything
];

## Determine instruction status
word Stat = [
        imem_error || dmem_error : SADR;
        !instr_valid: SINS;
        icode == IHALT : SHLT;
        1 : SAOK;
];

################ Program Counter Update ############################

## What address should instruction be fetched at

word new_pc = [
        # Call.  Use instruction constant
        icode == ICALL : valC;
        # Taken branch.  Use instruction constant
```

```
        icode == IJXX && Cnd : valC;
        # Completion of RET instruction.  Use value from stack
        icode == IRET : valM;
        # Default: Use incremented PC
        1 : valP;
];
#/* $end seq-all-hcl */
```

测试结果如下

```
[root@localhost seq]# (cd ../ptest; make SIM=../seq/ssim TFLAGS=-i)
./optest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 58 ISA Checks Succeed
./jtest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 96 ISA Checks Succeed
./ctest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 22 ISA Checks Succeed
./htest.pl -s ../seq/ssim -i
Simulating with ../seq/ssim
  All 756 ISA Checks Succeed
[root@localhost seq]#
```

PartC

此部分参考了网络文章https://zhuanlan.zhihu.com/p/33561203?utm_source=zhihu&utm_medium=social&utm_oi=58502127026176

修改ncopy.ys和pipe-full.hcl.尽所能提高ncopy.ys性能

优化方法:

1、使用iaddq指令

2、消除load/use 冒险

3、改进jump位置，减少预测错误的影响

修改前

```
# You can modify this portion
        # Loop header
        xorq %rax,%rax          # count = 0;
        andq %rdx,%rdx          # len <= 0?
```

```
        jle Done                # if so, goto Done:

Loop:   mrmovq (%rdi), %r10     # read val from src...
        rmmovq %r10, (%rsi)     # ...and store it to dst
        andq %r10, %r10         # val <= 0?
        jle Npos                # if so, goto Npos:
        irmovq $1, %r10         #使用iaddq优化
        addq %r10, %rax         # count++
Npos:   irmovq $1, %r10
        subq %r10, %rdx         # len--
        irmovq $8, %r10         #使用iaddq优化
        addq %r10, %rdi         # src++
        addq %r10, %rsi         # dst++
        andq %rdx,%rdx          # len > 0?
        jg Loop                 # if so, goto Loop:
```

修改后

1、调用iaddq

2、循环展开，循环因子为5

```
# You can modify this portion
# Loop header
        iaddq $-4, %rdx         # calculate len - 4
        jg Loop                 # if len >= 4 goto Loop

RESTT:
        iaddq $4, %rdx          # restore the old value of len
REST:
        jg NOT_FINISHED         # if %rdx is greater than 0,then not finished yet
        ret
NOT_FINISHED:
        mrmovq (%rdi), %r10             # load *src
        iaddq $8, %rdi                    //src++
        andq %r10, %r10                 # 测试是否为最后一个
        jle ADD3                                # if not greater than 0, jump
        iaddq $1, %rax                  # add the count of postive numbers
ADD3:
        rmmovq %r10, (%rsi)             //dst=src
        iaddq $8, %rsi                  //dst++
        iaddq $-1, %rdx                //len--
```

```
        jmp REST
Loop:
        mrmovq (%rdi), %r10     # read val from src...
        iaddq $40, %rdi         # src+=5
        rmmovq %r10, (%rsi) # save *src to *rsi
        andq %r10, %r10         # test %r10
        jle ADD1                # 统计positive
        iaddq $1, %rax          # add the number of postive numbers
ADD1:
        mrmovq -32(%rdi), %r10  # read the second number
        iaddq $40, %rsi                 # dst += 5
        rmmovq %r10, -32(%rsi)  # save *(src + 1) to *(dst + 1)
        andq %r10, %r10                 # val <= 0?
        mrmovq -24(%rdi), %r10   # rearrange the instructions to avoid load/use hazzard
        jle ADD2                                # if not, add the count
        iaddq $1, %rax                  # count++
ADD2:
        rmmovq %r10, -24(%rsi)
        andq %r10, %r10
        mrmovq -16(%rdi), %r10
        jle ADD4
        iaddq $1, %rax
ADD4:
        rmmovq %r10, -16(%rsi)
        andq %r10, %r10
        mrmovq -8(%rdi), %r10
        jle ADD5
        iaddq $1, %rax
ADD5:
        rmmovq %r10, -8(%rsi)
        andq %r10, %r10
        jle ADD6
        iaddq $1, %rax
ADD6:
        iaddq $-5, %rdx                 # len -= 5 if(len - 4 > 0) that means len > 4
        jg Loop                                 # if so, goto Loop:

        jmp RESTT
##################################################################

```

```
[root@localhost pipe]# ./benchmark.pl
        ncopy
0       18
1       27      27.00
2       39      19.50
3       48      16.00
4       60      15.00
5       49      9.80
6       61      10.17
7       70      10.00
8       82      10.25
9       91      10.11
10      82      8.20
11      91      8.27
12      103     8.58
13      112     8.62
14      124     8.86
15      112     7.47
16      124     7.75
17      133     7.82
18      145     8.06
19      154     8.11
20      145     7.25
21      154     7.33
22      166     7.55
23      175     7.61
24      187     7.79
25      175     7.00
26      187     7.19
27      196     7.26
28      208     7.43
29      217     7.48
30      208     6.93
31      217     7.00
32      229     7.16
33      238     7.21
34      250     7.35
35      238     6.80
36      250     6.94
37      259     7.00
38      271     7.13
39      280     7.18
40      271     6.78
41      280     6.83
42      292     6.95
43      301     7.00
44      313     7.11
45      301     6.69
46      313     6.80
47      322     6.85
48      334     6.96
49      343     7.00
50      334     6.68
51      343     6.73
52      355     6.83
53      364     6.87
54      376     6.96
55      364     6.62
56      376     6.71
57      385     6.75
58      397     6.84
59      406     6.88
60      397     6.62
61      406     6.66
62      418     6.74
63      427     6.78
64      439     6.86
Average CPE     8.20
Score   46.0/60.0
[root@localhost pipe]#
```

```
[xze@localhost pipe]$ ./correctness.pl
Simulating with instruction set simulator yis
        ncopy
0       OK
1       OK
2       OK
3       OK
4       OK
5       OK
6       OK
7       OK
8       OK
9       OK
10      OK
11      OK
12      OK
13      OK
14      OK
15      OK
16      OK
17      OK
18      OK
19      OK
20      OK
21      OK
22      OK
23      OK
24      OK
25      OK
26      OK
27      OK
28      OK
29      OK
30      OK
31      OK
32      OK
33      OK
34      OK
35      OK
36      OK
37      OK
38      OK
39      OK
40      OK
41      OK
42      OK
43      OK
44      OK
45      OK
46      OK
47      OK
48      OK
49      OK
50      OK
51      OK
52      OK
53      OK
54      OK
55      OK
56      OK
57      OK
58      OK
59      OK
60      OK
61      OK
62      OK
63      OK
64      OK
128     OK
192     OK
256     OK
68/68 pass correctness test
[xze@localhost pipe]$
```