Part A

前置

这里需要了解一下链表的结构体及对应的 Y86_64 汇编

```
typedef struct ELE {
    long val;
    struct ELE *next;
} *list_ptr;
```

```
# Sample linked list
.align 8
ele1:
        .quad 0x00a
        .quad ele2
ele2:
        .quad 0x0b0
        .quad ele3
ele3:
        .quad 0xc00
        .quad 0
```

sum.ys

要求

Write a Y86-64 program `sum.ys` that iteratively sums the elements of a linked list. Your program should consist of some code that sets up the stack structure, invokes a function, and then halts.

In this case, the function should be Y86-64 code for a function (`sum_list`) that is functionally equivalent to the C `sum_list` function in Figure 1. Test your program using the following three-element list:

源码

```
/* 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;
}
```

题解

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

```
        .pos 0
        irmovq stack, %rsp
        call main
        halt

# Sample linked list
.align 8
ele1:
        .quad 0x00a
        .quad ele2
ele2:
        .quad 0x0b0
        .quad ele3
ele3:
        .quad 0xc00
        .quad 0

main:
        irmovq ele1, %rdi
        call sum_list
        ret

# long sum_list(list_ptr ls)
# ls in %rdi
sum_list:
        irmovq $0, %rax
        jmp test

loop:
        mrmovq (%rdi), %r8
        addq %r8, %rax
        mrmovq 8(%rdi), %rdi

test:
        andq %rdi, %rdi
        jne loop
        ret

        .pos 0x200
stack:
```

rsum.ys

要求

Write a Y86-64 programr `sum.ys` that recursively sums the elements of a linked list. This code shouldbe similar to the code in `sum.ys`, except that it should use a functionrsumlistthat recursively sums alist of numbers, as shown with the C function `rsum_list` in Figure 1. Test your program using the samethree-element list you used for testing `list.ys`.

源码

```c
/* rsum_list - Recursive version of sum_list */
long rsum_list(list_ptr ls)
{
    if (!ls)
    return 0;
    else {
    long val = ls->val;
    long rest = rsum_list(ls->next);
    return val + rest;
    }
}
```

题解

```
# Execution begins at address 0
        .pos 0
        irmovq stack, %rsp
        call main
        halt

# Sample linked list
.align 8
ele1:
        .quad 0x00a
        .quad ele2
ele2:
        .quad 0x0b0
        .quad ele3
ele3:
        .quad 0xc00
        .quad 0

main:
        irmovq ele1, %rdi
        call rsum_list
        ret

# long rsum_list(list_ptr ls)
# ls in %rdi
rsum_list:
        andq %rdi, %rdi
        je zero
        mrmovq (%rdi), %rbx
        mrmovq 8(%rdi), %rdi
        pushq %rbx
        call rsum_list
        popq %rbx
        addq %rbx, %rax
        ret

zero:
        xorq %rax, %rax
        ret

        .pos 0x200
stack:
```

copy.ys

要求

Write a program (`copy.ys`) that 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.Your program should consist of code that sets up a stack frame, invokes a function `copy_block`, andthen halts. The function should be functionally equivalent to the C function `copy_block` shown in FigureFigure 1. Test your program using the following three-element source and destination blocks:

源码

```
/* copy_block - Copy src to dest and return xor checksum of src */
long copy_block(long *src, long *dest, long len)
{
    long result = 0;
    while (len > 0) {
    long val = *src++;
    *dest++ = val;
    result ^= val;
    len--;
    }
    return result;
}
```

题解

```
# Execution begins at address 0
        .pos 0
        irmovq stack, %rsp
        call main
        halt

.align 8
# Source block
src:
    .quad 0x00a
    .quad 0x0b0
    .quad 0xc00
# Destination
dest:
    .quad 0x111
    .quad 0x222
    .quad 0x333

main:
        irmovq src, %rdi
        irmovq dest, %rsi
        irmovq $0x3, %rdx
        call copy_block
        ret

# long copy_block(long *src, long *dest, long len)
# src in %rdi, %dest in %rsi, len in %rdx
copy_block:
```

```
        xorq %rax, %rax
        irmovq $0x8, %r8
        irmovq $0x1, %r9
        andq %rdx, %rdx
        jg loop

loop:
        mrmovq (%rdi), %rsi
        addq %r8, %rdi
        xorq %rsi, %rax
        addq %r8, %rsi
        subq %r9, %rdx
        andq %rdx, %rdx
        jg loop
        ret

        .pos 0x200
stack:
```

Part B

seq-full.hcl

要求

在 `sim/seq` 文件夹里，修改 `seq-full.hcl` 文件，添加 `iaddq` 指令

前置

iaddq 指令描述如下：

state	do
fetch	icode:ifun<-M1[PC]
	rA,rB<-M1[PC+1]
	valC<-M1[PC+2]
	ValP<-PC+10
decode	valB<-R[rB]
execute	ValE<-ValB+ValC
memory	
writeback	R[rB]<-ValE
	PC<-valP

还需要更改一下 Makefile，因为我的虚拟机里没有 tk 库

这里需要将我用 ### 标记的地方注释掉（我这个是已经注释掉的）

```
# Modify this line to indicate the default version

```

```
VERSION=std

# Comment this out if you don't have Tcl/Tk on your system

### GUIMODE=-DHAS_GUI

# Modify the following line so that gcc can find the libtcl.so and
# libtk.so libraries on your system. You may need to use the -L option
# to tell gcc which directory to look in. Comment this out if you
# don't have Tcl/Tk.

### TKLIBS=-L/usr/lib -ltk -ltcl

# Modify the following line so that gcc can find the tcl.h and tk.h
# header files on your system. Comment this out if you don't have
# Tcl/Tk.

### TKINC=-isystem /usr/include/tcl8.5

# Modify these two lines to choose your compiler and compile time
# flags.
```

或者是安装必备的依赖库

```
sudo apt-get install tcl tcl-dev tk tk-dev
```

不过因为 Makefile 里的版本太老，我们需要修改一下，需要修改的地方我用 ## 进行了标注

```
# Modify this line to indicate the default version

VERSION=full ##

# Comment this out if you don't have Tcl/Tk on your system

GUIMODE=-DHAS_GUI

# Modify the following line so that gcc can find the libtcl.so and
# libtk.so libraries on your system. You may need to use the -L option
# to tell gcc which directory to look in. Comment this out if you
# don't have Tcl/Tk.

TKLIBS=-L /usr/lib -ltk -ltcl

# Modify the following line so that gcc can find the tcl.h and tk.h
# header files on your system. Comment this out if you don't have
# Tcl/Tk.

TKINC=-isystem /usr/include/tcl8.6 ##

# Modify these two lines to choose your compiler and compile time
# flags.

CC=gcc
CFLAGS=-Wall -O2 -DUSE_INTERP_RESULT ##
```

之后还要配置一下环境，先在当前目录输入 `make clean;make VERSION=full` 生成用于测试的 ssim 文件

在 `/sim/y86-code` 目录中打开 Makefile 文件

然后在里面的 SEQFILES 变量中加上 `asumi.seq` （因为这个文件里面用到了 iaddq 指令）

再输入指令 `make clean;make testssim` 生成 `asumi.yo` 文件

最后在之前的目录中输入 `./ssim -t ../y86-code/asumi.yo` 来测试是否正确

题解

```
#/* $begin seq-all-hcl */
####################################################################
#  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"

##### ALU Functions referenced explicitly                          #####
wordsig ALUADD  'A_ADD'     # ALU should add its arguments

##### Possible instruction status values                           #####
wordsig SAOK    'STAT_AOK'  # Normal execution
wordsig SADR    'STAT_ADR'  # Invalid memory address
wordsig SINS    'STAT_INS'  # Invalid instruction
wordsig SHLT    'STAT_HLT'  # Halt instruction encountered

##### Signals that can be referenced by control logic ####################

##### Fetch stage inputs        #####
wordsig pc 'pc'             # Program counter
##### Fetch stage computations      #####
wordsig imem_icode 'imem_icode'     # icode field from instruction memory
wordsig imem_ifun  'imem_ifun'      # ifun field from instruction memory
wordsig icode     'icode'       # Instruction control code
wordsig ifun      'ifun'        # Instruction function
wordsig rA    'ra'          # rA field from instruction
wordsig rB    'rb'          # rB field from instruction
wordsig valC      'valc'        # Constant from instruction
wordsig valP      'valp'        # Address of following instruction
boolsig imem_error 'imem_error'     # Error signal from instruction memory
boolsig instr_valid 'instr_valid'   # Is fetched instruction valid?

##### Decode stage computations     #####
wordsig valA    'vala'          # Value from register A port
wordsig valB    'valb'          # Value from register B port

##### Execute stage computations    #####
wordsig valE    'vale'          # Value computed by ALU
boolsig Cnd 'cond'          # Branch test

##### Memory stage computations     #####
wordsig valM    'valm'          # Value read from memory
boolsig dmem_error 'dmem_error'     # Error signal from data memory


################################################################
#    Control Signal Definitions.                               #
################################################################

################ Fetch Stage     ###################################

# Determine instruction code
word icode = [
    imem_error: INOP;
    1: imem_icode;      # Default: get from instruction memory
];

# Determine instruction function
```

```
word ifun = [
    imem_error: FNONE;
    1: imem_ifun;       # Default: get from instruction memory
];

bool instr_valid = icode in
    { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
           IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ, IIADDQ };

# Does fetched instruction require a regid byte?
bool need_regids =
    icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
             IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ };

# Does fetched instruction require a constant word?
bool need_valC =
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL, IIADDQ };

################ Decode Stage    ###################################

## What register should be used as the A source?
word srcA = [
    icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : rA;
    icode in { IPOPQ, IRET } : RRSP;
    1 : RNONE; # Don't need register
];

## What register should be used as the B source?
word srcB = [
    icode in { IOPQ, IRMMOVQ, IMRMOVQ, IIADDQ } : rB;
    icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE;  # Don't need register
];

## 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
];
```

```
160
161 ## Select input B to ALU
162 word aluB = [
163     icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
164                IPUSHQ, IRET, IPOPQ, IIADDQ } : valB;
165     icode in { IRRMOVQ, IIRMOVQ } : 0;
166     # Other instructions don't need ALU
167 ];
168
169 ## Set the ALU function
170 word alufun = [
171     icode == IOPQ : ifun;
172     1 : ALUADD;
173 ];
174
175 ## Should the condition codes be updated?
176 bool set_cc = icode in { IOPQ, IIADDQ };
177
178 ################ Memory Stage    ###################################
179
180 ## Set read control signal
181 bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
182
183 ## Set write control signal
184 bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
185
186 ## Select memory address
187 word mem_addr = [
188     icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
189     icode in { IPOPQ, IRET } : valA;
190     # Other instructions don't need address
191 ];
192
193 ## Select memory input data
194 word mem_data = [
195     # Value from register
196     icode in { IRMMOVQ, IPUSHQ } : valA;
197     # Return PC
198     icode == ICALL : valP;
199     # Default: Don't write anything
200 ];
201
202 ## Determine instruction status
203 word Stat = [
204     imem_error || dmem_error : SADR;
205     !instr_valid: SINS;
206     icode == IHALT : SHLT;
207     1 : SAOK;
208 ];
209
210 ################ Program Counter Update ############################
211
212 ## What address should instruction be fetched at
213
214 word new_pc = [
215     # Call.  Use instruction constant
216     icode == ICALL : valC;
217     # 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 */
```