Part A

sum_list

```
# sum.ys

# Execution begins at address 0
    .pos 0
init:
# function prologue
    irmovl Stack, %esp
    irmovl Stack, %ebp

    call Main
    halt


# Sample linked list
.align 4
ele1:
    .long 0x00a
    .long ele2
ele2:
    .long 0x0b0
    .long ele3
ele3:
    .long 0xc00
    .long 0

# main() function
Main:
# function prologue
    pushl %ebp
    rrmovl %esp,%ebp

# arg1 = &ele1
    irmovl ele1,%ebx
    pushl %ebx
# Sum(&ele1)
    call Sum
```

```
# function ending
    rrmovl %ebp,%esp
    popl %ebp
    ret

# int sum_list(list_ptr ls)
Sum:
# function prologue
    pushl %ebp
    rrmovl %esp,%ebp

# 0:    Main's ebp          <- ebp
# 4:    ret_addr
# 8:    arg1
    mrmovl 8(%ebp),%ecx
#   eax = 0
    xorl %eax,%eax

#   ele1.next
    mrmovl 4(%ecx),%ebx

    andl %ebx,%ebx
#   if(!ebx)    jmp;
    je End

Loop:
#   ele?.val
    mrmovl (%ecx),%ebx
    addl %ebx,%eax

#   ecx = ecx.next
    mrmovl 4(%ecx),%ecx
    andl %ecx,%ecx
#   if(ecx)     jmp;
    jne Loop

End:
    rrmovl %ebp,%esp
    popl %ebp
#   返回值放在 eax
    ret

    .pos 0x100
Stack:
```

rsum_list

```
# rsum.ys

# Execution begins at address 0
    .pos 0
init:
# function prologue
    irmovl Stack, %esp
    irmovl Stack, %ebp

    call Main
    halt

# Sample linked list
.align 4
ele1:
    .long 0x00a
    .long ele2
ele2:
    .long 0x0b0
    .long ele3
ele3:
    .long 0xc00
    .long 0


# main() function
Main:
# function prologue
    pushl %ebp
    rrmovl %esp,%ebp

# arg1 = &ele1
    irmovl ele1,%ebx
    pushl %ebx

# rSum(&ele1)
    call rSum

# function ending
    rrmovl %ebp,%esp
    popl %ebp
    ret

# int rsum_list(list_ptr ls)
rSum:
# function prologue
    pushl %ebp
    rrmovl %esp,%ebp

# 0:    Main's ebp          <- ebp
# 4:    ret_addr
# 8:    arg1
    mrmovl 8(%ebp),%ecx

#   eax = 0
    xorl %eax,%eax
#   if(!ls)     return 0;
    andl %ecx,%ecx
    je End

#   将 %old_ecx 放到栈上
    pushl %ecx

#   ele.next
    mrmovl 4(%ecx),%ebx
```

```
#   rsum_list(ls->next)
    pushl %ebx
    call rSum

#   rSum 函数返回后, 栈帧回到主函数,
#   此时的栈顶结构为:
#   0:  4(%old_ecx) [arg1]
#   4:  %old_ecx
    popl %ecx
    popl %ecx

    mrmovl (%ecx),%ebx
    addl %ebx,%eax

End:
    rrmovl %ebp,%esp
    popl %ebp
    ret

    .pos 0x100
Stack:
```

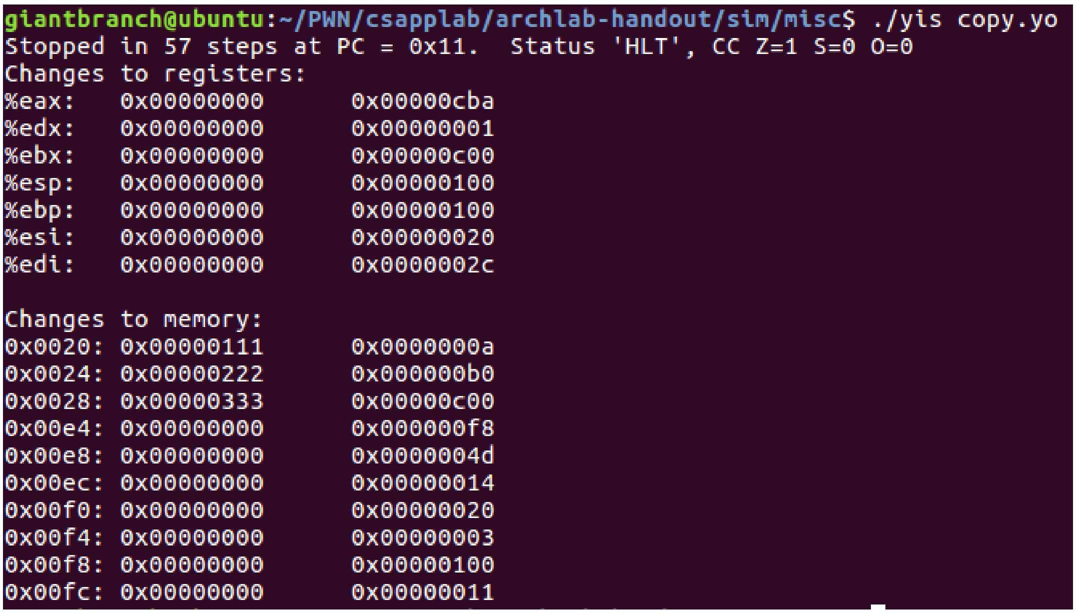
copy_block

```
# copy.ys

# Execution begins at address 0
    .pos 0
init:
# function prologue
    irmovl Stack, %esp
    irmovl Stack, %ebp

    call Main
    halt

.align 4
# Source block
src:
    .long 0x00a
    .long 0x0b0
    .long 0xc00
# Destination block
dest:
    .long 0x111
    .long 0x222
    .long 0x333

# main() function
Main:
# function prologue
    pushl %ebp
    rrmovl %esp,%ebp

# arg3: len
```

```
    irmovl $3,%ebx
    pushl %ebx
# arg2: dest
    irmovl dest,%ebx
    pushl %ebx
# arg1: src
    irmovl src,%ebx
    pushl %ebx
# copy_block(&src, &dest, len)
    call Copy

# function ending
    rrmovl %ebp,%esp
    popl %ebp
    ret

# int copy_block(int *src, int *dest, int len)
Copy:
# function prologue
    pushl %ebp
    rrmovl %esp,%ebp

# 0:    Main's ebp          <- ebp
# 4:    ret_addr
# 8:    arg1
# 12:   arg2
# 16:   arg3
    mrmovl 8(%ebp),%esi     # src
    mrmovl 12(%ebp),%edi    # dest
    mrmovl 16(%ebp),%ecx    # len
#   result: eax = 0
    xorl %eax,%eax

    andl %ecx,%ecx
#   if(!ecx)    jmp;
    je End

Loop:
    irmovl $4,%edx

#   int val = *src++;
    mrmovl (%esi),%ebx
    addl %edx,%esi

#   *dest++ = val;
    rmmovl %ebx,(%edi)
    addl %edx,%edi

#   result ^= val;
    xorl %ebx,%eax

    irmovl $1,%edx
    subl %edx,%ecx

    andl %ecx,%ecx
#   if(ecx)     jmp;
    jne Loop
```

```
End:
    rrmovl %ebp,%esp
    popl %ebp

    # 返回值放在 %eax
    ret

    .pos 0x100
Stack:
```

结果

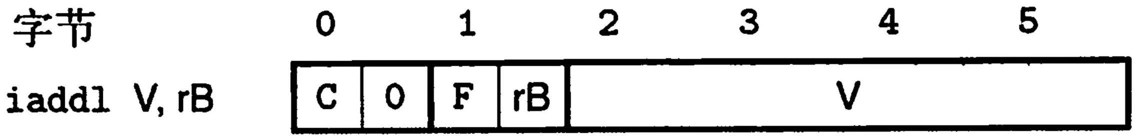


```
giantbranch@ubuntu:~/PWN/csapplab/archlab-handout/sim/misc$ ./yis copy.yo
Stopped in 57 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%edx:   0x00000000      0x00000001
%ebx:   0x00000000      0x00000c00
%esp:   0x00000000      0x00000100
%ebp:   0x00000000      0x00000100
%esi:   0x00000000      0x00000020
%edi:   0x00000000      0x0000002c

Changes to memory:
0x0020: 0x00000111      0x0000000a
0x0024: 0x00000222      0x000000b0
0x0028: 0x00000333      0x00000c00
0x00e4: 0x00000000      0x000000f8
0x00e8: 0x00000000      0x0000004d
0x00ec: 0x00000000      0x00000014
0x00f0: 0x00000000      0x00000020
0x00f4: 0x00000000      0x00000003
0x00f8: 0x00000000      0x00000100
0x00fc: 0x00000000      0x00000011
```

Part B

iaddl

指令格式：



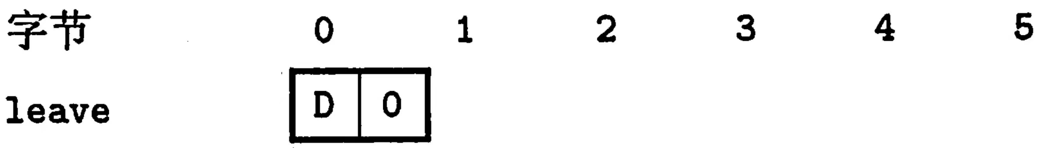
字节	0	1	2	3	4	5
iaddl V, rB	C 0	F rB	V			

指令处理各阶段如下：

阶段	***iaddl V, rB***
取指	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_4[PC+2]$ $valP \leftarrow PC+6$
译码	$valB \leftarrow R[rB]$
执行	$valE \leftarrow valB + valC$ $SetCC$
访存	
写回	$R[rB] \leftarrow valE$
更新PC	$PC \leftarrow valP$

leave

指令格式：



指令处理各阶段如下：

阶段	***leave***
取指	$icode : ifun \leftarrow M_1[PC]$ $valP \leftarrow PC+1$
译码	$valA \leftarrow R[\%ebp]$ $valB \leftarrow R[\%ebp]$
执行	$valE \leftarrow valB + 4$
访存	$valM \leftarrow M_4[valA]$
写回	$R[\%ebp] \leftarrow valM$ $R[\%esp] \leftarrow valE$
更新PC	$PC \leftarrow valP$

seq-full.hcl

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

```
####################################################################

## Your task is to implement the iaddl and leave instructions
## The file contains a declaration of the icodes
## for iaddl (IIADDL) and leave (ILEAVE).
## 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 'int 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 Instruction Codes #############
intsig INOP     'I_NOP'
intsig IHALT    'I_HALT'
intsig IRRMOVL  'I_RRMOVL'
intsig IIRMOVL  'I_IRMOVL'
intsig IRMMOVL  'I_RMMOVL'
intsig IMRMOVL  'I_MRMOVL'
intsig IOPL 'I_ALU'
intsig IJXX 'I_JMP'
intsig ICALL    'I_CALL'
intsig IRET 'I_RET'
intsig IPUSHL   'I_PUSHL'
intsig IPOPL    'I_POPL'
# Instruction code for iaddl instruction
intsig IIADDL   'I_IADDL'
# Instruction code for leave instruction
intsig ILEAVE   'I_LEAVE'

##### Symbolic represenations of Y86 function codes                  #####
intsig FNONE    'F_NONE'        # Default function code

##### Symbolic representation of Y86 Registers referenced explicitly #####
intsig RESP     'REG_ESP'       # Stack Pointer
intsig REBP     'REG_EBP'       # Frame Pointer
intsig RNONE    'REG_NONE'      # Special value indicating "no register"

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

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

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

##### Fetch stage inputs        #####
intsig pc 'pc'              # Program counter
##### Fetch stage computations      #####
intsig imem_icode 'imem_icode'      # icode field from instruction memory
intsig imem_ifun  'imem_ifun'       # ifun field from instruction memory
intsig icode      'icode'       # Instruction control code
intsig ifun   'ifun'        # Instruction function
intsig rA     'ra'          # rA field from instruction
intsig rB     'rb'          # rB field from instruction
intsig valC   'valc'        # Constant from instruction
intsig 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     #####
intsig valA 'vala'          # Value from register A port
intsig valB 'valb'          # Value from register B port

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

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


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

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

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

# Determine instruction function
int ifun = [
    imem_error: FNONE;
    1: imem_ifun;       # Default: get from instruction memory
];

bool instr_valid = icode in
    { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
           IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL, ILEAVE };

# Does fetched instruction require a regid byte?
bool need_regids =
    icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
             IIRMOVL, IRMMOVL, IMRMOVL, IIADDL };

# Does fetched instruction require a constant word?
bool need_valC =
    icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL };

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

## What register should be used as the A source?
int srcA = [
    icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : rA;
    icode in { IPOPL, IRET } : RESP;
    icode in { ILEAVE } : REBP;
    1 : RNONE; # Don't need register
];

## What register should be used as the B source?
int srcB = [
    icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL } : rB;
    icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
    icode in { ILEAVE } : REBP;
    1 : RNONE;  # Don't need register
];

## What register should be used as the E destination?
int dstE = [
    icode in { IRRMOVL } && Cnd : rB;
    icode in { IIRMOVL, IOPL, IIADDL} : rB;
    icode in { IPUSHL, IPOPL, ICALL, IRET, ILEAVE } : RESP;
    1 : RNONE;  # Don't write any register
];

## What register should be used as the M destination?
int dstM = [
    icode in { IMRMOVL, IPOPL } : rA;
    icode in { ILEAVE } : REBP;
    1 : RNONE;  # Don't write any register
];

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

## Select input A to ALU
int aluA = [
    icode in { IRRMOVL, IOPL } : valA;
    icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : valC;
    icode in { ICALL, IPUSHL } : -4;
    icode in { IRET, IPOPL, ILEAVE } : 4;
    # Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
    icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
              IPUSHL, IRET, IPOPL, IIADDL, ILEAVE } : valB;
    icode in { IRRMOVL, IIRMOVL } : 0;
    # Other instructions don't need ALU
];

## Set the ALU function
int alufun = [
    icode == IOPL : ifun;
    1 : ALUADD;
];

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

################ Memory Stage    ###################################

## Set read control signal
bool mem_read = icode in { IMRMOVL, IPOPL, IRET, ILEAVE };

## Set write control signal
bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };

## Select memory address
int mem_addr = [
    icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
    icode in { IPOPL, IRET, ILEAVE } : valA;
    # Other instructions don't need address
];

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

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

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

## What address should instruction be fetched at

int 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 */
```

注意

- 要在 `ssim.c` 中将 `#include <tk.h>` 注释掉，因为好像没有 GUI。
- 在 `Makefile` 中将所有有关 `Tcl/Tk` 的代码注释掉。

Part C