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:
```

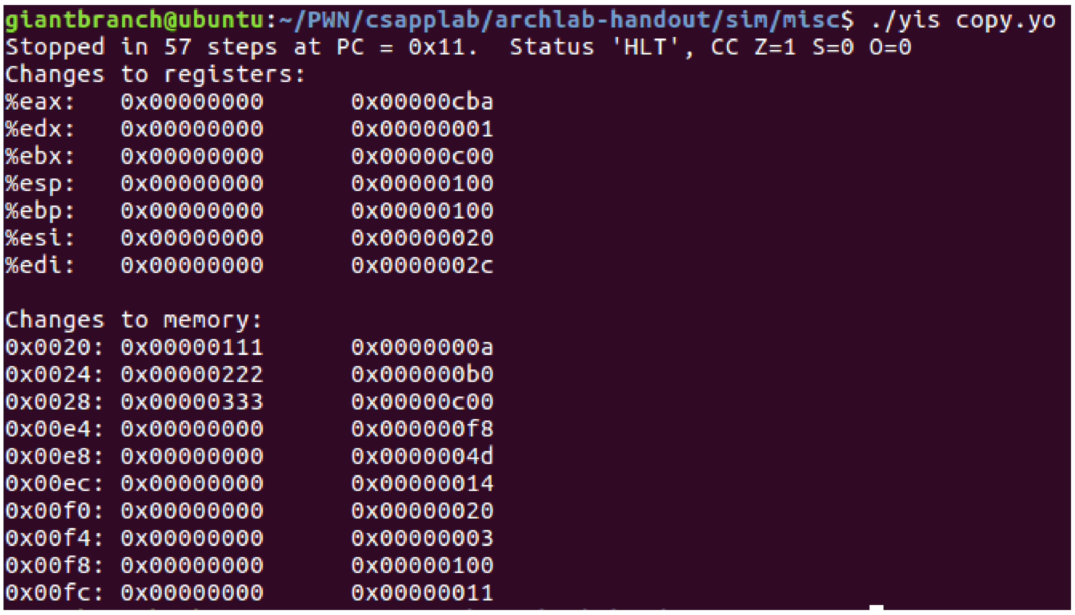
copy_block

```
# 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

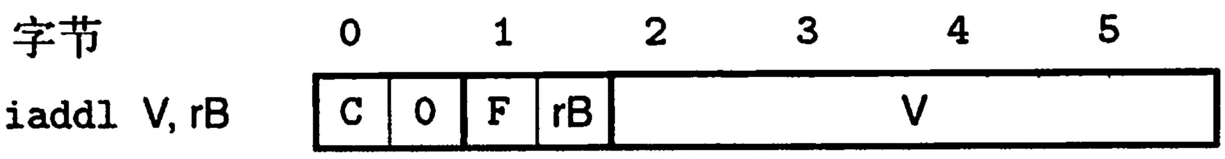


```
giantbranch@ubuntu:~/PWN/csapplab/archlab-handout/sim/misc$ ./yis copy.yo
Stopped in 57 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000cba
%edx:   0x00000000      0x00000001
%ebx:   0x00000000      0x00000c00
%esp:   0x00000000      0x00000100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

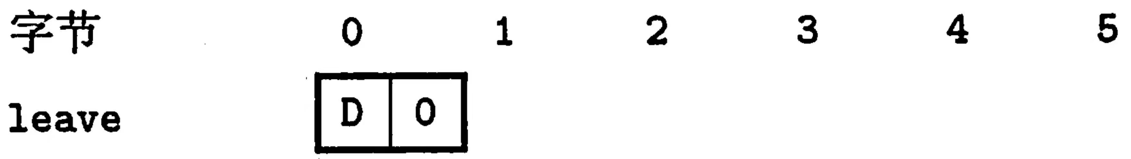


指令处理各阶段如下：

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

先将 Part B 中的 `iaddl, leave` 加入到 `pipe-full.hcl` 中，利用第五章的“循环展开方法”和第4章中的“加载使用冒险”进行优化。对 *ncopy* 函数进行 4 次的循环展开。在原始的函数中存在加载使用冒险，即 `mrmovl` 从存储器中读入 `src` 到 `esi`，`rmmovl` 从 `esi` 存储到 `dest` 中，期间因为加载使用冒险需要暂停一个周期，针对这个进行改进：主要是在这两条指令中插入另一条 `mrmovl` 指令，避免冒险，节省时间，也为后面的循环展开提前获取值。

```
mrmovl  (%ebx), %esi
rmmovl %esi, (%ecx)
```

pipe-full.hcl

```
#/* $begin pipe-all-hcl */
####################################################################
#
#    HCL Description of Control for Pipelined Y86 Processor
#
#    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 "pipeline.h"'
quote '#include "stages.h"'
quote '#include "sim.h"'
quote 'int sim_main(int argc, char *argv[]);'
quote 'int main(int argc, char *argv[]){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
#####
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 SBUB 'STAT_BUB'  # Bubble in stage
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
##############

##### Pipeline Register F
##########################################

intsig F_predPC 'pc_curr->pc'        # Predicted value of PC

##### Intermediate Values in Fetch Stage
###########################

intsig imem_icode  'imem_icode'      # icode field from instruction
memory
intsig imem_ifun   'imem_ifun'       # ifun  field from instruction
memory
intsig f_icode  'if_id_next->icode'  # (Possibly modified)
instruction code
intsig f_ifun   'if_id_next->ifun'   # Fetched instruction function
intsig f_valC   'if_id_next->valc'   # Constant data of fetched
instruction
intsig f_valP   'if_id_next->valp'   # Address of following
instruction
boolsig imem_error 'imem_error'      # Error signal from
instruction memory
boolsig instr_valid 'instr_valid'    # Is fetched instruction
valid?

##### Pipeline Register D
##########################################
intsig D_icode 'if_id_curr->icode'   # Instruction code
intsig D_rA 'if_id_curr->ra'         # rA field from instruction
intsig D_rB 'if_id_curr->rb'         # rB field from instruction
intsig D_valP 'if_id_curr->valp'     # Incremented PC
```

```
##### Intermediate Values in Decode Stage
#########################

intsig d_srcA    'id_ex_next->srca'  # srcA from decoded
instruction
intsig d_srcB    'id_ex_next->srcb'  # srcB from decoded
instruction
intsig d_rvalA 'd_regvala'       # valA read from register file
intsig d_rvalB 'd_regvalb'       # valB read from register file

##### Pipeline Register E
##########################################
intsig E_icode 'id_ex_curr->icode'   # Instruction code
intsig E_ifun  'id_ex_curr->ifun'    # Instruction function
intsig E_valC  'id_ex_curr->valc'    # Constant data
intsig E_srcA  'id_ex_curr->srca'    # Source A register ID
intsig E_valA  'id_ex_curr->vala'    # Source A value
intsig E_srcB  'id_ex_curr->srcb'    # Source B register ID
intsig E_valB  'id_ex_curr->valb'    # Source B value
intsig E_dstE 'id_ex_curr->deste'    # Destination E register ID
intsig E_dstM 'id_ex_curr->destm'    # Destination M register ID

##### Intermediate Values in Execute Stage
#########################
intsig e_valE 'ex_mem_next->vale'   # valE generated by ALU
boolsig e_Cnd 'ex_mem_next->takebranch' # Does condition hold?
intsig e_dstE 'ex_mem_next->deste'      # dstE (possibly modified
to be RNONE)

##### Pipeline Register M
#########################
intsig M_stat 'ex_mem_curr->status'     # Instruction status
intsig M_icode 'ex_mem_curr->icode' # Instruction code
intsig M_ifun  'ex_mem_curr->ifun'  # Instruction function
intsig M_valA  'ex_mem_curr->vala'      # Source A value
intsig M_dstE 'ex_mem_curr->deste'  # Destination E register ID
intsig M_valE  'ex_mem_curr->vale'      # ALU E value
intsig M_dstM 'ex_mem_curr->destm'  # Destination M register ID
boolsig M_Cnd 'ex_mem_curr->takebranch' # Condition flag
boolsig dmem_error 'dmem_error'         # Error signal from
instruction memory
```

```
##### Intermediate Values in Memory Stage
##########################
intsig m_valM 'mem_wb_next->valm'   # valM generated by memory
intsig m_stat 'mem_wb_next->status' # stat (possibly modified to be
SADR)

##### Pipeline Register W
##########################################
intsig W_stat 'mem_wb_curr->status'     # Instruction status
intsig W_icode 'mem_wb_curr->icode' # Instruction code
intsig W_dstE 'mem_wb_curr->deste'  # Destination E register ID
intsig W_valE  'mem_wb_curr->vale'      # ALU E value
intsig W_dstM 'mem_wb_curr->destm'  # Destination M register ID
intsig W_valM  'mem_wb_curr->valm'  # Memory M value

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

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

## What address should instruction be fetched at
int f_pc = [
    # Mispredicted branch.  Fetch at incremented PC
    M_icode == IJXX && !M_Cnd : M_valA;
    # Completion of RET instruction.
    W_icode == IRET : W_valM;
    # Default: Use predicted value of PC
    1 : F_predPC;
];

## Determine icode of fetched instruction
int f_icode = [
    imem_error : INOP;
    1: imem_icode;
];

# Determine ifun
int f_ifun = [
    imem_error : FNONE;
    1: imem_ifun;
];

# Is instruction valid?
bool instr_valid = f_icode in
    { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
      IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL, ILEAVE };

# Determine status code for fetched instruction
int f_stat = [
    imem_error: SADR;
    !instr_valid : SINS;
    f_icode == IHALT : SHLT;
    1 : SAOK;
];

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

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

# Predict next value of PC
int f_predPC = [
    f_icode in { IJXX, ICALL } : f_valC;
    1 : f_valP;
];

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



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

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

## What register should be used as the E destination?
int d_dstE = [
    D_icode in { IRRMOVL, IIRMOVL, IOPL, IIADDL} : D_rB;
    D_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 d_dstM = [
    D_icode in { IMRMOVL, IPOPL } : D_rA;
    D_icode in { ILEAVE } : REBP;
    1 : RNONE;  # Don't write any register
];

## What should be the A value?
## Forward into decode stage for valA
int d_valA = [
    D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
    d_srcA == e_dstE : e_valE;    # Forward valE from execute
    d_srcA == M_dstM : m_valM;    # Forward valM from memory
    d_srcA == M_dstE : M_valE;    # Forward valE from memory
    d_srcA == W_dstM : W_valM;    # Forward valM from write back
    d_srcA == W_dstE : W_valE;    # Forward valE from write back
    1 : d_rvalA;  # Use value read from register file
];

int d_valB = [
    d_srcB == e_dstE : e_valE;    # Forward valE from execute
    d_srcB == M_dstM : m_valM;    # Forward valM from memory
    d_srcB == M_dstE : M_valE;    # Forward valE from memory
    d_srcB == W_dstM : W_valM;    # Forward valM from write back
    d_srcB == W_dstE : W_valE;    # Forward valE from write back
    1 : d_rvalB;  # Use value read from register file
];
```

```
################ Execute Stage
#####################################

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

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

## Set the ALU function
int alufun = [
    E_icode == IOPL : E_ifun;
    1 : ALUADD;
];

## Should the condition codes be updated?
bool set_cc = E_icode in { IOPL, IIADDL } &&
    # State changes only during normal operation
    !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS,
SHLT };

## Generate valA in execute stage
int e_valA = E_valA;    # Pass valA through stage

## Set dstE to RNONE in event of not-taken conditional move
int e_dstE = [
    E_icode == IRRMOVL && !e_Cnd : RNONE;
    1 : E_dstE;
];
```

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

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

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

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

#/* $begin pipe-m_stat-hcl */
## Update the status
int m_stat = [
    dmem_error : SADR;
    1 : M_stat;
];
#/* $end pipe-m_stat-hcl */

## Set E port register ID
int w_dstE = W_dstE;

## Set E port value
int w_valE = W_valE;

## Set M port register ID
int w_dstM = W_dstM;

## Set M port value
int w_valM = W_valM;

## Update processor status
int Stat = [
    W_stat == SBUB : SAOK;
    1 : W_stat;
];
```

```
################ Pipeline Register Control
#########################

# Should I stall or inject a bubble into Pipeline Register F?
# At most one of these can be true.
bool F_bubble = 0;
bool F_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } &&
     E_dstM in { d_srcA, d_srcB } ||
    # Stalling at fetch while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode };

# Should I stall or inject a bubble into Pipeline Register D?
# At most one of these can be true.
bool D_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } &&
     E_dstM in { d_srcA, d_srcB };

bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Stalling at fetch while ret passes through pipeline
    # but not condition for a load/use hazard
    !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB
}) &&
      IRET in { D_icode, E_icode, M_icode };

# Should I stall or inject a bubble into Pipeline Register E?
# At most one of these can be true.
bool E_stall = 0;
bool E_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } &&
     E_dstM in { d_srcA, d_srcB};

# Should I stall or inject a bubble into Pipeline Register M?
# At most one of these can be true.
bool M_stall = 0;
```

```
# Start injecting bubbles as soon as exception passes through
memory stage
bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR,
SINS, SHLT };

# Should I stall or inject a bubble into Pipeline Register W?
bool W_stall = W_stat in { SADR, SINS, SHLT };
bool W_bubble = 0;
#/* $end pipe-all-hcl */
```

ncopy.ys

基本的优化思路：

1. `mrmovl  (%ebx), %esi; rmmovl %esi, (%ecx);`

```
#/* $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

##################################################################
# You can modify this portion
    # Loop header
    xorl %eax,%eax      # count = 0;
    andl %edx,%edx      # len <= 0?
    jle Done        # if so, goto Done:
    iaddl $-4,%edx
    jl Ending

fLoop:  mrmovl (%ebx), %esi # read val from src...
    mrmovl 4(%ebx), %edi

    rmmovl %esi, (%ecx) # ...and store it to dst
    andl %esi, %esi     # val <= 0?
    jle Cal1        # if so, goto Npos:
    iaddl $1, %eax      # count++

Cal1:   rmmovl %edi, 4(%ecx)    # ...and store it to dst
    andl %edi, %edi     # val <= 0?
    jle Cal2        # if so, goto Npos:
    iaddl $1, %eax      # count++

Cal2:   mrmovl 8(%ebx), %esi    # read val from src...
    mrmovl 12(%ebx), %edi

    rmmovl %esi, 8(%ecx)    # ...and store it to dst
    andl %esi, %esi     # val <= 0?
    jle Cal3        # if so, goto Npos:
    iaddl $1, %eax      # count++

Cal3:   rmmovl %edi, 12(%ecx)   # ...and store it to dst
    andl %edi, %edi     # val <= 0?
    jle Judge       # if so, goto Npos:
    iaddl $1, %eax      # count++

Judge:  iaddl $16, %ebx
    iaddl $16, %ecx
    iaddl $-4, %edx
    jl Ending
    jmp fLoop

Ending: iaddl $4, %edx

    andl %edx,%edx      # len <= 0?
    jle Done        # if so, goto Done:
    iaddl $-2, %edx
    jl End1
    mrmovl (%ebx), %esi # read val from src...
    mrmovl 4(%ebx), %edi

    rmmovl %esi, (%ecx) # ...and store it to dst
    andl %esi, %esi     # val <= 0?
    jle eCal1       # if so, goto Npos:
    iaddl $1, %eax      # count++

eCal1:  rmmovl %edi, 4(%ecx)    # ...and store it to dst
    andl %edi, %edi     # val <= 0?
    jle eCal2       # if so, goto Npos:
    iaddl $1, %eax      # count++

eCal2:  iaddl $8, %ebx
    iaddl $8, %ecx

    andl %edx,%edx      # len <= 0?
    jle Done        # if so, goto Done:
    jmp End2

End1:   iaddl $2, %edx
End2:   mrmovl (%ebx), %esi # read val from src...
    rmmovl %esi, (%ecx) # ...and store it to dst
    andl %esi, %esi     # val <= 0?
    jle Done        # if so, goto Npos:
    iaddl $1, %eax      # count++

##################################################################
# Do not modify the following section of code
# Function epilogue.
Done:
    popl %edi               # Restore callee-save registers
    popl %ebx
    popl %esi
    leave
    ret
##################################################################
# Keep the following label at the end of your function
End:
#/* $end ncopy-ys */
```