Arch lab

Part A

`Sum.ys`

```
# JiangCan 2019011325
    .pos 0
init:
    irmovl Stack,%esp # Set up stack pointer
    irmovl Stack,%ebp # Set up base pointer
    call Main # Execute main program
    halt # Terminate program

    .align 4
ele1:
    .long 0x00a
    .long ele2
ele2:
    .long 0x0b0
    .long ele3
ele3:
    .long 0xc00
    .long 0

Main:
    pushl %ebp
    rrmovl %esp,%ebp
    irmovl ele1,%edx
    pushl %edx                      # prepare parameters
    call sum_list                   # call function
    rrmovl %ebp,%esp
    popl %ebp
    ret

sum_list:
    pushl %ebp
    rrmovl %esp,%ebp
    irmovl $0x10,%ecx
    subl %ecx,%esp
    irmovl $0x0,%ecx
    rmmovl %ecx,-0x4(%ebp)          # val = -4(%ebp) = 0
    jmp    end
loop:
    mrmovl 0x8(%ebp),%eax
    mrmovl (%eax),%eax              # %eax = ls->val

    mrmovl 0xfffffffc(%ebp),%ecx    # %ecx = -4(%ebp) = val
    addl %eax,%ecx
    rmmovl %ecx,0xfffffffc(%ebp)    # val += ls->val

    mrmovl 0x8(%ebp),%eax           # %eax = ls
    mrmovl 0x4(%eax),%eax           # %eax = ls->next
    rmmovl %eax,0x8(%ebp)           # ls = ls->next
end:
    mrmovl 0x8(%ebp),%ecx
    irmovl $0x0,%eax
    xorl %eax,%ecx
    jne loop                        # while (ls)
    mrmovl 0xfffffffc(%ebp),%eax
    rrmovl %ebp,%esp
    popl %ebp
    ret

    .pos 0x500
Stack:
```

`rsum.ys`

```
# JiangCan 2019011325
    .pos 0
init:
    irmovl Stack,%esp
    irmovl Stack,%ebp
    call Main
    halt

    .align 4
ele1:
    .long 0x00a
    .long ele2
ele2:
    .long 0x0b0
    .long ele3
ele3:
    .long 0xc00
    .long 0

Main:
    pushl %ebp
    rrmovl %esp,%ebp
    irmovl ele1,%edx
    pushl %edx                      # prepare parameters
    call Rsum                  # call function
    rrmovl %ebp,%esp
    popl %ebp
    ret

Rsum:
    pushl %ebp
    rrmovl %esp,%ebp
    irmovl $0x18,%edx
    subl %edx,%esp
    irmovl $0x0,%edx
    mrmovl 0x8(%ebp),%eax           # %eax = ls
    xorl %edx,%eax
    jne else                        # else
    irmovl $0x0,%eax
    jmp return                      # if (!ls) return 0
else:
    mrmovl 0x8(%ebp),%eax           # %eax = ls
    mrmovl (%eax),%eax
    rmmovl %eax,0xfffffff0(%ebp)    # -0xf(%ebp) = val = ls->val

    mrmovl 0x8(%ebp),%eax           # %eax = ls
    mrmovl 0x4(%eax),%eax           # %eax = ls->next
    irmovl $0xc,%edx
    subl %edx,%esp
    pushl %eax                      # prepare parameter = ls->next
    call Rsum                  # recursive, %eax = rest

    irmovl $0x10,%edx
    addl %edx,%esp                  # restore the stack
    mrmovl 0xfffffff0(%ebp),%edx    # %edx = -0xf(%ebp) = val
    addl %edx,%eax                  # %eax = val + rest
return:
    rrmovl %ebp,%esp
    popl %ebp
    ret

    .pos 0x500
Stack:
```

`copy.ys`

```
# JiangCan 2019011325
    .pos 0
init:
    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:
    pushl %ebp
    rrmovl %esp,%ebp
    irmovl 0x3,%edx
    pushl %edx
    irmovl dest,%edx
    pushl %edx
    irmovl src,%edx
    pushl %edx                      # preparing paramters
    call copy_block                 # call function
    rrmovl %ebp,%esp
    popl %ebp
    ret

copy_block:
    pushl %ebp
    rrmovl %esp,%ebp
    irmovl $0x10,%ecx
    subl %ecx,%esp
    irmovl $0x0,%ecx                # result = 0
    rmmovl %ecx,0xfffffff8(%ebp)    # -8(%ebp) = result
    jmp end
loop:
    mrmovl 0x8(%ebp),%eax           # %eax = src
    mrmovl (%eax),%eax              # %eax = val

    mrmovl 0xc(%ebp),%edx           # %edx = dest
    rmmovl %eax,(%edx)              # *dest = val

    mrmovl 0xfffffff8(%ebp),%ecx    # %ecx = result
    xorl %eax,%ecx
    rmmovl %ecx,0xfffffff8(%ebp)    # result ^= val

    mrmovl 0x10(%ebp),%ecx
    irmovl $0x1,%edx
    subl %edx,%ecx
    rmmovl %ecx,0x10(%ebp)          # len--

    mrmovl 0x8(%ebp),%eax           # %eax = src
    irmovl 0x4,%edx
    addl %eax,%edx
    rmmovl %edx,0x8(%ebp)           # src++

    mrmovl 0xc(%ebp),%eax           # %eax = dest
    irmovl 0x4,%edx
    addl %eax,%edx
    rmmovl %edx,0xc(%ebp)           # dest++
end:
    mrmovl 0x10(%ebp),%ecx
    irmovl $0x0,%edx
    subl %edx,%ecx
    jg loop                         # while(len>0)
    mrmovl 0xfffffff8(%ebp),%eax
    rrmovl %ebp,%esp
    popl %ebp
    ret

    .pos 0x500
Stack:
```

Part B

顺序实现

在这一部分中，我们需要扩展SEQ和PIPE处理器，以支持两个新指令：Iaddl和Leave。

Iaddl:rB 取出来，加上立即数，再存回去。

Leave : 取出 esp，ebp 的值，读取 ebp 位置的内存，使得 new_ebp=(ebp), new_esp = ebp+4

```
             leave
fetch:       icode:ifun <- M_1[PC]
             valP <- PC+2
decode:      valA <- R[%ebp]
             valB <- R[%ebp]
execute:     valE <- 4+valB
memory:      valM <- M_4[valA]
writeback:   R[%esp] <- valE
             R[%ebp] <- valM
PC update:   PC <- valP
```

```
             iaddl V,rb
fetch:       icode:ifun <- M_1[PC]
             rA:rB <- M_1[PC+1]
             valC <- M_4[PC+2]
             valP <- PC+6
decode:      valB <- R[rB]
execute:     valE <- valC+valB
memory:
writeback:   R[rB] <- valE
PC update:   PC <- valP
```

对于不同的阶段

1. Fetch
   a. add ILEAVE into conditions of `instr_valid`.
   b. add IIADDL into conditions of `instr_valid`, `need_regids`, `need_valC`.
2. Decode
   a. add ILEAVE into conditions of srcA=REBP, srcB=REBP, dstE=RESP, dstM=REBP.
   b. add IIADDL into conditions of srcB=rB, dstE=rB.
3. Execute
   a. add ILEAVE into conditions of aluA=4, aluB=valB.
   b. add IIADDL into conditions of aluA=valC, aluB=valB, set_cc.
4. Memory
   a. add ILEAVE into conditions of mem_read, mem_addr=valA.

流水线实现

流水线实现和顺序实现的步骤划分大体相当，不过有三处不同。首先，PC 阶段放到最前面了。其次，各种值有了传递关系，数据旁路等，但是这些关系已经封装好了。最后，把 leave 中的 srcA，srcB 修改成 srcA=ebp， srcB=esp，这样才能让 ebp 的值留到访存阶段。

1. data bypassing: Once src and dst are set properly, it is well done with existing code and need no modification.
2. ret instruction, mispredicted branch and exception handling: Well done with existing code.
3. use/load hazard: leave do a memory read, so it may cause use/load hazard and need to be inserted to conditions of use/load hazard. There are totally four places where use/load hazard are determined.

```
#/* Name: JiangCan, ID: 2019011325 */
#/* $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, ILEAVE, IIADDL } : 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, ILEAVE } &&
   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, ILEAVE } &&
   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, ILEAVE } && 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, ILEAVE } &&
   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 */
```