```
#/* $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

##########################################################################
##########################################################################
#                          Leader's name and ID.                         #
##########################################################################
#Descriptions:                                                           #
# Both instructions leave and iaddl are implemented here, which are#
# similar to those of 'seq'.                                            #
# For iaddl/IIADDL, the work is almost the same as that of 'seq'      #
# version. The difference is that information, such as source           #
# register or destine register is acquired from pipeline registers.#
# (It's very lucky to see all forwarding logic has already been       #
# implmeneted)                                                          #
# For leave/ILEAVE, the work is more complicated. Since this is a     #
# load instruction, attention must be paid to avoidance of data        #
# hazards. By careful analysis, we decide that ILEAVE can be           #
# grouped with IMRMOVL, IPOPL when coping with data hazards, which #
# largely reduced complexity of the job.                               #
##########################################################################
#iaddl Details:                                                         #
# 1.instr_valid                                                         #
# 2.need_regids                                                         #
# 3.need_valC                                                           #
# 4.d_srcB - D_rB                                                       #
# 5.d_dstE - D_rB                                                       #
# 6.aluA - valC                                                         #
# 7.aluB - valB                                                         #
# 8.set_cc                                                              #
#leave Details:                                                         #
# 1.instr_vali                                                          #
# 2.need_regids                                                         #
# 3.d_srcA - REBP                                                       #
# 4.d_dstE - RESP                                                       #
# 5.d_dstM - REBP                                                       #
```

```
# 6.aluA - E_valA                                #
# 7.aluB - 4                                     #
# 8.mem_addr - M_valA                                #
# 9.mem_read                                     #
# 10.F_stall                                     #
# 11.D_stall                                     #
# 12.D_bubble                                    #
# 13.E_bubble                                    #
##################################################################
##################################################################

##################################################################
#       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, IIADDL, ILEAVE,
          IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };

# 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, IIADDL, ILEAVE,
```

```
                    IIRMOVL, IRMMOVL, IMRMOVL };

# 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;
      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, ILEAVE } : E_valA;
	E_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : E_valC;
	E_icode in { ICALL, IPUSHL } : -4;
	E_icode in { IRET, IPOPL } : 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 } : E_valB;
	E_icode in { IRRMOVL, IIRMOVL } : 0;
	E_icode in { ILEAVE } : 4;
	# 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 } : M_valA;
        M_icode in { 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 } &&	#Dst value generated after M stage
	  E_dstM in { d_srcA, d_srcB } ||	#but D needs the registers
	# 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 } &&	#Dst value generated after M stage
	  E_dstM in { d_srcA, d_srcB };		#but E needs the registers

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 */
```