CS:APP2e Web Aside ARCH:HCL: HCL Descriptions of Y86 Processors*

Randal E. Bryant
David R. O'Hallaron

June 5, 2012

Notice

The material in this document is supplementary material to the book Computer Systems, A Programmer's Perspective, Second Edition*, by Randal E. Bryant and David R. O'Hallaron, published by Prentice-Hall and copyrighted 2011. In this document, all references beginning with "CS:APP2e " are to this book. More information about the book is available at* `csapp.cs.cmu.edu`.

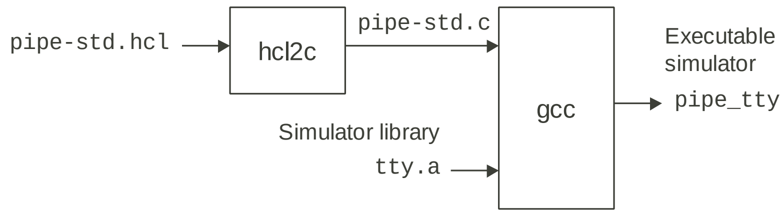
This document is being made available to the public, subject to copyright provisions. You are free to copy and distribute it, but you should not use any of this material without attribution.

This document describes the Hardware Control Language, HCL, devised to provide a simple, yet systematic way to describe the control logic for Y86 processors. It also includes copies of the HCL descriptions of Y86 processors SEQ and PIPE. Electronic versions of these HCL files are available at `www.csapp.cs.cmu.edu`.

1 HCL Reference Manual

HCL has some of the features of a hardware description language (HDL), allowing users to describe Boolean functions and word-level selection operations. On the other hand, it lacks many features found in true HDLs, such as ways to declare registers and other storage elements; looping and conditional constructs; module definition and instantiation capabilities; and bit extraction and insertion operations.

In its implementation, HCL is really just a language for generating a very stylized form of C code. All of the block definitions in an HCL file get converted to C functions by a program HCL2C These functions are then compiled and linked with library code implementing the other simulator functions to generate an executable simulation program, as diagrammed below:



This diagram shows the files used to generate the text version of the pipeline simulator.

It would be possible to describe the behavior of the control logic directly in C, rather than writing HCL and translating this to C. The advantage of the HCL route is that we more clearly separate the functionality of the hardware from the inner workings of the simulator.

HCL supports just two data types: `bool` (for "Boolean") signals are either 0 or 1, while `int` (for "integer") signals are equivalent to `int` values in C. Data type `int` is used for all types of multi-bit signals, such as words, register IDs, and instruction codes. When converted to C, both data types are represented as `int` data, but a value of type `bool` will only equal 0 or 1.

1.1 Signal Declarations

Expressions in HCL can reference named *signals* of type integer or Boolean. The signal names must start with a letter (`a`–`z` or `A`–`Z`), followed by any number of letters, digits, or underscores (_). Signal names are case sensitive. The Boolean and integer signal names used in HCL Boolean and integer expressions are really just aliases for C expressions. The declaration of a signal also defines the associated C expression. A signal declaration has one of the following forms:

```
boolsig  name  'C-expr'
intsig   name  'C-expr'
```

where *C-expr* can be an arbitrary C expression, except that it cannot contain a single quote (`'`) or a newline character (`\n`). When generating C code, HCL2C will replace any signal name with the corresponding C expression.

1.2 Quoted Text

Quoted text provides a mechanism to pass text directly through HCL2C into the generated C file. This can be used to insert variable declarations, `include` statements, and other things generally found in C files. The general form is:

```
quote  'string'
```

where *string* can be any string that does not contain single quotes (`'`) or newline characters (`\n`).

Syntax	Meaning
`0`	Logic value 0
`1`	Logic value 1
$name$	Named Boolean signal
$int\text{-}expr$ `in` $\{int\text{-}expr_1$`,` $int\text{-}expr_2$`,` ... `,` $int\text{-}expr_k\}$	Set membership test
$int\text{-}expr_1$ `==` $int\text{-}expr_2$	Equality test
$int\text{-}expr_1$ `!=` $int\text{-}expr_2$	Not equal test
$int\text{-}expr_1$ `<` $int\text{-}expr_2$	Less than test
$int\text{-}expr_1$ `<=` $int\text{-}expr_2$	Less than or equal test
$int\text{-}expr_1$ `>` $int\text{-}expr_2$	Greater than test
$int\text{-}expr_1$ `>=` $int\text{-}expr_2$	Greater than or equal test
`!` $bool\text{-}expr$	NOT
$bool\text{-}expr_1$ `&&` $bool\text{-}expr_2$	AND
$bool\text{-}expr_1$ `||` $bool\text{-}expr_2$	OR

Figure 1: **HCL Boolean expressions.** These expressions evaluate to 0 or 1. The operations are listed in descending order of precedence, where those within each group have equal precedence.

1.3 Expressions and Blocks

There are two types of expressions: Boolean and integer, which we refer to in our syntax descriptions as $bool\text{-}expr$ and $int\text{-}expr$, respectively. Figure 1 lists the different types of Boolean expressions. They are listed in descending order of precedence, with the operations within each group (groups are separated by horizontal lines) having equal precedence. Parentheses can be used to override the normal operator precedence.

At the top level are the constant values `0` and `1` and named Boolean signals. Next in precedence are expressions that have integer arguments but yield Boolean results. The set membership test compares the value of the first integer expression $int\text{-}expr$ to the values of each of the integer expressions comprising the set $\{int\text{-}expr_1, \ldots int\text{-}expr_k\}$, yielding 1 if any matching value is found. The relational operators compare two integer expressions, generating 1 when the relation holds and 0 when it does not.

The remaining expressions in Figure 1 consist of formulas using Boolean connectives (`!` for NOT, `&&` for AND. and `||` for OR).

There are just three types of integer expressions: numbers, named integer signals, and case expressions. Numbers are written in decimal notation and can be negative. Named integer signals use the naming rules described earlier. Case expressions have the following general form:

$$
\begin{array}{l}
\texttt{[} \\
\quad bool\text{-}expr_1 \quad \texttt{:} \quad int\text{-}expr_1 \\
\quad bool\text{-}expr_2 \quad \texttt{:} \quad int\text{-}expr_2 \\
\quad \qquad\qquad\quad \vdots \\
\quad bool\text{-}expr_k \quad \texttt{:} \quad int\text{-}expr_k \\
\texttt{]}
\end{array}
$$

The expression contains a series of cases, where each case i consists of a Boolean expression $bool\text{-}expr_i$, indicating whether this case should be selected, and an integer expression $int\text{-}expr_i$, indicating the value resulting for this case. In evaluating a case expression, the Boolean expressions are conceptually evaluated in sequence. When one of them yields 1, the value of the corresponding integer expression is returned as the case expression value. If no Boolean expression evaluates to 1, then the value of the case expression is 0. One good programming practice is to have the last Boolean expression be `1`, guaranteeing at least one matching case.

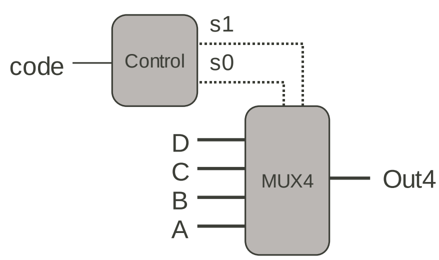
HCL expressions are used to define the behavior of a block of combinational logic. A block definition has one of the following forms:

```
bool name = bool-expr;
 int name = int-expr;
```

where the first form defines a Boolean block, while the second defines a word-level block. For a block declared with *name* as its name, HCL2C generates a function `gen_`*name*. This function has no arguments, and it returns a result of type `int`.

1.4 HCL Example

The following example shows a complete HCL file. The C code generated by processing it with HCL2C is completely self-contained. It can be compiled and run using command line arguments for the input signals. More typically, HCL files define just the control part of a simulation model. The generated C code is then compiled and linked with other code to form the executable simulator. We show this example just to give a concrete example of HCL. The circuit is based on the `MUX4` circuit shown in CS:APP2e Figure 4.14 and reproduced here:




```
## Simple example of an HCL file.
## This file can be converted to C using hcl2c, and then compiled.

## In this example, we will generate the MUX4 circuit shown in
## Section 4.2.4.  It consists of a control block that generates
## bit-level signals s1 and s0 from the input signal code,
## and then uses these signals to control a 4-way multiplexor
## with data inputs A, B, C, and D.

## This code is embedded in a C program that reads
```

```
## the values of code, A, B, C, and D from the command line
## and then prints the circuit output

## Information that is inserted verbatim into the C file
quote ’#include <stdio.h>’
quote ’#include <stdlib.h>’
quote ’int code_val, s0_val, s1_val;’
quote ’char **data_names;’

## Declarations of signals used in the HCL description and
## the corresponding C expressions.
boolsig s0 ’s0_val’
boolsig s1 ’s1_val’
intsig code ’code_val’
intsig  A ’atoi(data_names[0])’
intsig  B ’atoi(data_names[1])’
intsig  C ’atoi(data_names[2])’
intsig  D ’atoi(data_names[3])’

## HCL descriptions of the logic blocks
bool s1 = code in { 2, 3 };

bool s0 = code in { 1, 3 };

int Out4 = [
        !s1 && !s0 : A; # 00
        !s1        : B; # 01
        !s0        : C; # 10
        1          : D; # 11
];

## More information inserted verbatim into the C code to
## compute the values and print the output
quote ’int main(int argc, char *argv[]) {’
quote ’  data_names = argv+2;’
quote ’  code_val = atoi(argv[1]);’
quote ’  s1_val = gen_s1();’
quote ’  s0_val = gen_s0();’
quote ’  printf("Out = %d\n", gen_Out4());’
quote ’  return 0;’
quote ’}’
```

This file defines Boolean signals `s0` and `s1` and integer signal `code` to be aliases for references to global variables `s0_val`, `s1_val`, and `code_val`. It declares integer signals `A`, `B`, `C`, and `D`, where the corresponding C expressions apply the standard library function `atoi` to strings passed as command line arguments.

The definition of the block named `s1` generates the following C code:

```
int gen_s1()
```

```
{
    return ((code_val) == 2 || (code_val) == 3);
}
```

As can be seen here, set membership testing is implemented as a series of comparisons, and that every reference to signal `code` is replaced by the C expression `code_val`.

Note that there is no direct relation between the signal `s1` declared on line 23 of the HCL file, and the block named `s1` declared on line 31. One is an alias for a C expression, while the other generates a function named `gen_s1`.

The quoted text at the end generates the following main function:

```
int main(int argc, char *argv[]) {
  data_names = argv+2;
  code_val = atoi(argv[1]);
  s1_val = gen_s1();
  s0_val = gen_s0();
  printf("Out = %d\n", gen_Out4());
  return 0;
}
```

The main function calls the functions `gen_s1`, `gen_s0`, and `gen_Out4` that were generated from the block definitions. We can also see how the C code must define the sequencing of block evaluations and the setting of the values used in the C expressions representing the different signal values.

2 SEQ

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

####################################################################
#    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’

##### 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 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 };

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

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

################ 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;
        1 : RNONE; # Don't need register
```

```
];

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

## What register should be used as the M destination?
int dstM = [
        icode in { IMRMOVL, IPOPL } : rA;
        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 } : valC;
        icode in { ICALL, IPUSHL } : -4;
        icode in { IRET, IPOPL } : 4;
        # Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
        icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                      IPUSHL, IRET, IPOPL } : 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 };
```

```

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

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

## 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 } : 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 */
```

3 PIPE

```
#/* $begin pipe-all-hcl */
####################################################################
#    HCL Description of Control for Pipelined Y86 Processor        #
#    Copyright (C) Randal E. Bryant, David R. O’Hallaron, 2010     #
####################################################################

####################################################################
#    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’

##### 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 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 };

# 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 };

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

# 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;
        1 : RNONE; # Don’t need register
];

## What register should be used as the B source?
int d_srcB = [
        D_icode in { IOPL, IRMMOVL, IMRMOVL  } : 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} : D_rB;
        D_icode in { IPUSHL, IPOPL, ICALL, IRET } : 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;
        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 } : 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 } : 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 == IOPL &&
        # 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;
        # Other instructions don't need address
];

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

## 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 */
```