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

Randal E. Bryant
David R. O'Hallaron

August 25, 2009

Notice

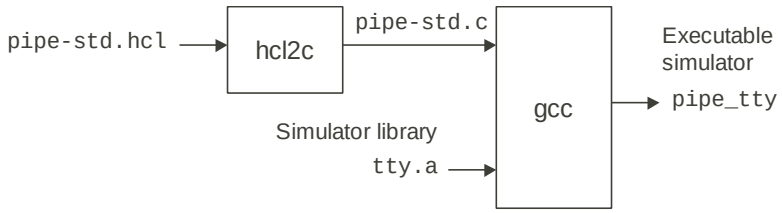
The material in this document is supplementary material to the book Computer Systems, A Programmer's Persepctive, Second Edition*, by Randal E. Bryant and David R. O'Hallaron, published by Prentice-Hall and copyrighted 2010. In this document, all references beginning with "CS:APP2e " are to this book. More information about the book is available at* `www.csapp.cs.cmu.edu`.

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-expr `in` {*int-expr*$_1$, *int-expr*$_2$, . . . , *int-expr*$_k$}	Set membership test
int-expr$_1$ `==` *int-expr*$_2$	Equality test
int-expr$_1$ `!=` *int-expr*$_2$	Not equal test
int-expr$_1$ `<` *int-expr*$_2$	Less than test
int-expr$_1$ `<=` *int-expr*$_2$	Less than or equal test
int-expr$_1$ `>` *int-expr*$_2$	Greater than test
int-expr$_1$ `>=` *int-expr*$_2$	Greater than or equal test
`!` *bool-expr*	NOT
bool-expr$_1$ `&&` *bool-expr*$_2$	AND
bool-expr$_1$ `||` *bool-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-expr* and *int-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-expr* to the values of each of the integer expressions comprising the set {*int-expr*$_1$, . . . *int-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:

```
[
    bool-expr_1   :   int-expr_1
    bool-expr_2   :   int-expr_2
                  ⋮
    bool-expr_k   :   int-expr_k
]
```

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.

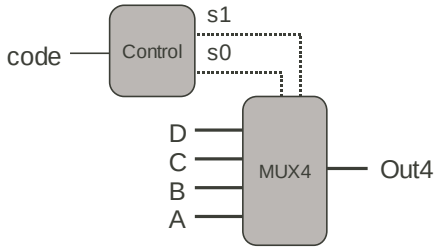
HCL expressions are used to define the behavior of a block of combinational logic. A block definition has one of the following forms:

$$\texttt{bool}\ name\ \texttt{=}\ bool\text{-}expr\texttt{;}$$
$$\texttt{int}\ name\ \texttt{=}\ int\text{-}expr\texttt{;}$$

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:




```
1 ## Simple example of an HCL file.
2 ## This file can be converted to C using hcl2c, and then compiled.
3
4 ## In this example, we will generate the MUX4 circuit shown in
5 ## Section 4.2.4.  It consists of a control block that generates
6 ## bit-level signals s1 and s0 from the input signal code,
7 ## and then uses these signals to control a 4-way multiplexor
8 ## with data inputs A, B, C, and D.
9
10 ## This code is embedded in a C program that reads
```

```
11 ## the values of code, A, B, C, and D from the command line
12 ## and then prints the circuit output
13
14 ## Information that is inserted verbatim into the C file
15 quote '#include <stdio.h>'
16 quote '#include <stdlib.h>'
17 quote 'int code_val, s0_val, s1_val;'
18 quote 'char **data_names;'
19
20 ## Declarations of signals used in the HCL description and
21 ## the corresponding C expressions.
22 boolsig s0 's0_val'
23 boolsig s1 's1_val'
24 intsig code 'code_val'
25 intsig  A 'atoi(data_names[0])'
26 intsig  B 'atoi(data_names[1])'
27 intsig  C 'atoi(data_names[2])'
28 intsig  D 'atoi(data_names[3])'
29
30 ## HCL descriptions of the logic blocks
31 bool s1 = code in { 2, 3 };
32
33 bool s0 = code in { 1, 3 };
34
35 int Out4 = [
36         !s1 && !s0 : A; # 00
37         !s1        : B; # 01
38         !s0        : C; # 10
39         1          : D; # 11
40 ];
41
42 ## More information inserted verbatim into the C code to
43 ## compute the values and print the output
44 quote 'int main(int argc, char *argv[]) {'
45 quote '  data_names = argv+2;'
46 quote '  code_val = atoi(argv[1]);'
47 quote '  s1_val = gen_s1();'
48 quote '  s0_val = gen_s0();'
49 quote '  printf("Out = %d\n", gen_Out4());'
50 quote '  return 0;'
51 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

```
 1 ##################################################################
 2 #  HCL Description of Control for Single Cycle Y86 Processor SEQ   #
 3 #  Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010       #
 4 ##################################################################
 5
 6 ##################################################################
 7 #    C Include's.  Don't alter these                             #
 8 ##################################################################
 9
10 quote '#include <stdio.h>'
11 quote '#include "isa.h"'
12 quote '#include "sim.h"'
13 quote 'int sim_main(int argc, char *argv[]);'
14 quote 'int gen_pc(){return 0;}'
15 quote 'int main(int argc, char *argv[])'
16 quote '  {plusmode=0;return sim_main(argc,argv);}'
17
18 ##################################################################
19 #    Declarations.  Do not change/remove/delete any of these      #
```

```
20 ################################################################
21
22 ##### Symbolic representation of Y86 Instruction Codes #############
23 intsig INOP     'I_NOP'
24 intsig IHALT    'I_HALT'
25 intsig IRRMOVL  'I_RRMOVL'
26 intsig IIRMOVL  'I_IRMOVL'
27 intsig IRMMOVL  'I_RMMOVL'
28 intsig IMRMOVL  'I_MRMOVL'
29 intsig IOPL     'I_ALU'
30 intsig IJXX     'I_JMP'
31 intsig ICALL    'I_CALL'
32 intsig IRET     'I_RET'
33 intsig IPUSHL   'I_PUSHL'
34 intsig IPOPL    'I_POPL'
35
36 ##### Symbolic represenations of Y86 function codes                 #####
37 intsig FNONE    'F_NONE'        # Default function code
38
39 ##### Symbolic representation of Y86 Registers referenced explicitly #####
40 intsig RESP     'REG_ESP'       # Stack Pointer
41 intsig RNONE    'REG_NONE'      # Special value indicating "no register"
42
43 ##### ALU Functions referenced explicitly                           #####
44 intsig ALUADD   'A_ADD'         # ALU should add its arguments
45
46 ##### Possible instruction status values                           #####
47 intsig SAOK     'STAT_AOK'              # Normal execution
48 intsig SADR     'STAT_ADR'      # Invalid memory address
49 intsig SINS     'STAT_INS'      # Invalid instruction
50 intsig SHLT     'STAT_HLT'      # Halt instruction encountered
51
52 ##### Signals that can be referenced by control logic ####################
53
54 ##### Fetch stage inputs               #####
55 intsig pc 'pc'                         # Program counter
56 ##### Fetch stage computations         #####
57 intsig imem_icode 'imem_icode'         # icode field from instruction memory
58 intsig imem_ifun  'imem_ifun'          # ifun field from instruction memory
59 intsig icode      'icode'              # Instruction control code
60 intsig ifun       'ifun'               # Instruction function
61 intsig rA         'ra'                 # rA field from instruction
62 intsig rB         'rb'                 # rB field from instruction
63 intsig valC       'valc'               # Constant from instruction
64 intsig valP       'valp'               # Address of following instruction
65 boolsig imem_error 'imem_error'        # Error signal from instruction memory
66 boolsig instr_valid 'instr_valid'      # Is fetched instruction valid?
67
68 ##### Decode stage computations        #####
69 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
];
```

```
120
121 ## What register should be used as the B source?
122 int srcB = [
123         icode in { IOPL, IRMMOVL, IMRMOVL  } : rB;
124         icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
125         1 : RNONE;  # Don't need register
126 ];
127
128 ## What register should be used as the E destination?
129 int dstE = [
130         icode in { IRRMOVL } && Cnd : rB;
131         icode in { IIRMOVL, IOPL} : rB;
132         icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
133         1 : RNONE;  # Don't write any register
134 ];
135
136 ## What register should be used as the M destination?
137 int dstM = [
138         icode in { IMRMOVL, IPOPL } : rA;
139         1 : RNONE;  # Don't write any register
140 ];
141
142 ################ Execute Stage   ######################################
143
144 ## Select input A to ALU
145 int aluA = [
146         icode in { IRRMOVL, IOPL } : valA;
147         icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
148         icode in { ICALL, IPUSHL } : -4;
149         icode in { IRET, IPOPL } : 4;
150         # Other instructions don't need ALU
151 ];
152
153 ## Select input B to ALU
154 int aluB = [
155         icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
156                        IPUSHL, IRET, IPOPL } : valB;
157         icode in { IRRMOVL, IIRMOVL } : 0;
158         # Other instructions don't need ALU
159 ];
160
161 ## Set the ALU function
162 int alufun = [
163         icode == IOPL : ifun;
164         1 : ALUADD;
165 ];
166
167 ## Should the condition codes be updated?
168 bool set_cc = icode in { IOPL };
169
```

```
################ 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;
];
```

3 PIPE

```
1  ##################################################################
2  #    HCL Description of Control for Pipelined Y86 Processor      #
3  #    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010   #
4  ##################################################################
5
6  ##################################################################
7  #    C Include's.  Don't alter these                             #
8  ##################################################################
9
10 quote '#include <stdio.h>'
11 quote '#include "isa.h"'
12 quote '#include "pipeline.h"'
13 quote '#include "stages.h"'
14 quote '#include "sim.h"'
15 quote 'int sim_main(int argc, char *argv[]);'
16 quote 'int main(int argc, char *argv[]){return sim_main(argc,argv);}'
17
18 ##################################################################
19 #    Declarations.  Do not change/remove/delete any of these     #
20 ##################################################################
21
22 ##### Symbolic representation of Y86 Instruction Codes #############
23 intsig INOP     'I_NOP'
24 intsig IHALT    'I_HALT'
25 intsig IRRMOVL  'I_RRMOVL'
26 intsig IIRMOVL  'I_IRMOVL'
27 intsig IRMMOVL  'I_RMMOVL'
28 intsig IMRMOVL  'I_MRMOVL'
29 intsig IOPL     'I_ALU'
30 intsig IJXX     'I_JMP'
31 intsig ICALL    'I_CALL'
32 intsig IRET     'I_RET'
33 intsig IPUSHL   'I_PUSHL'
34 intsig IPOPL    'I_POPL'
35
36 ##### Symbolic represenations of Y86 function codes           #####
37 intsig FNONE    'F_NONE'        # Default function code
38
39 ##### Symbolic representation of Y86 Registers referenced     #####
40 intsig RESP     'REG_ESP'           # Stack Pointer
41 intsig RNONE    'REG_NONE'          # Special value indicating "no register"
42
43 ##### ALU Functions referenced explicitly ##########################
44 intsig ALUADD   'A_ADD'             # ALU should add its arguments
45
46 ##### Possible instruction status values                      #####
47 intsig SBUB     'STAT_BUB'      # Bubble in stage
```

```
48 intsig SAOK     'STAT_AOK'      # Normal execution
49 intsig SADR     'STAT_ADR'      # Invalid memory address
50 intsig SINS     'STAT_INS'      # Invalid instruction
51 intsig SHLT     'STAT_HLT'      # Halt instruction encountered
52
53 ##### Signals that can be referenced by control logic ##############
54
55 ##### Pipeline Register F ########################################
56
57 intsig F_predPC 'pc_curr->pc'        # Predicted value of PC
58
59 ##### Intermediate Values in Fetch Stage #############################
60
61 intsig imem_icode  'imem_icode'      # icode field from instruction memory
62 intsig imem_ifun   'imem_ifun'       # ifun  field from instruction memory
63 intsig f_icode  'if_id_next->icode'  # (Possibly modified) instruction code
64 intsig f_ifun   'if_id_next->ifun'   # Fetched instruction function
65 intsig f_valC   'if_id_next->valc'   # Constant data of fetched instruction
66 intsig f_valP   'if_id_next->valp'   # Address of following instruction
67 boolsig imem_error 'imem_error'      # Error signal from instruction memory
68 boolsig instr_valid 'instr_valid'    # Is fetched instruction valid?
69
70 ##### Pipeline Register D ########################################
71 intsig D_icode 'if_id_curr->icode'   # Instruction code
72 intsig D_rA 'if_id_curr->ra'         # rA field from instruction
73 intsig D_rB 'if_id_curr->rb'         # rB field from instruction
74 intsig D_valP 'if_id_curr->valp'     # Incremented PC
75
76 ##### Intermediate Values in Decode Stage  ##########################
77
78 intsig d_srcA    'id_ex_next->srca'  # srcA from decoded instruction
79 intsig d_srcB    'id_ex_next->srcb'  # srcB from decoded instruction
80 intsig d_rvalA 'd_regvala'           # valA read from register file
81 intsig d_rvalB 'd_regvalb'           # valB read from register file
82
83 ##### Pipeline Register E ########################################
84 intsig E_icode 'id_ex_curr->icode'   # Instruction code
85 intsig E_ifun  'id_ex_curr->ifun'    # Instruction function
86 intsig E_valC  'id_ex_curr->valc'    # Constant data
87 intsig E_srcA  'id_ex_curr->srca'    # Source A register ID
88 intsig E_valA  'id_ex_curr->vala'    # Source A value
89 intsig E_srcB  'id_ex_curr->srcb'    # Source B register ID
90 intsig E_valB  'id_ex_curr->valb'    # Source B value
91 intsig E_dstE 'id_ex_curr->deste'    # Destination E register ID
92 intsig E_dstM 'id_ex_curr->destm'    # Destination M register ID
93
94 ##### Intermediate Values in Execute Stage ##########################
95 intsig e_valE 'ex_mem_next->vale'       # valE generated by ALU
96 boolsig e_Cnd 'ex_mem_next->takebranch' # Does condition hold?
97 intsig e_dstE 'ex_mem_next->deste'      # dstE (possibly modified to be RNONE)
```

```
 98
 99 ##### Pipeline Register M                  ########################
100 intsig M_stat 'ex_mem_curr->status'     # Instruction status
101 intsig M_icode 'ex_mem_curr->icode'     # Instruction code
102 intsig M_ifun  'ex_mem_curr->ifun'      # Instruction function
103 intsig M_valA  'ex_mem_curr->vala'      # Source A value
104 intsig M_dstE 'ex_mem_curr->deste'      # Destination E register ID
105 intsig M_valE  'ex_mem_curr->vale'      # ALU E value
106 intsig M_dstM 'ex_mem_curr->destm'      # Destination M register ID
107 boolsig M_Cnd 'ex_mem_curr->takebranch' # Condition flag
108 boolsig dmem_error 'dmem_error'         # Error signal from instruction memory
109
110 ##### Intermediate Values in Memory Stage ##########################
111 intsig m_valM 'mem_wb_next->valm'       # valM generated by memory
112 intsig m_stat 'mem_wb_next->status'     # stat (possibly modified to be SADR)
113
114 ##### Pipeline Register W ##########################################
115 intsig W_stat 'mem_wb_curr->status'     # Instruction status
116 intsig W_icode 'mem_wb_curr->icode'     # Instruction code
117 intsig W_dstE 'mem_wb_curr->deste'      # Destination E register ID
118 intsig W_valE  'mem_wb_curr->vale'      # ALU E value
119 intsig W_dstM 'mem_wb_curr->destm'      # Destination M register ID
120 intsig W_valM  'mem_wb_curr->valm'      # Memory M value
121
122 ####################################################################
123 #    Control Signal Definitions.                                   #
124 ####################################################################
125
126 ################ Fetch Stage     ###################################
127
128 ## What address should instruction be fetched at
129 int f_pc = [
130         # Mispredicted branch.  Fetch at incremented PC
131         M_icode == IJXX && !M_Cnd : M_valA;
132         # Completion of RET instruction.
133         W_icode == IRET : W_valM;
134         # Default: Use predicted value of PC
135         1 : F_predPC;
136 ];
137
138 ## Determine icode of fetched instruction
139 int f_icode = [
140         imem_error : INOP;
141         1: imem_icode;
142 ];
143
144 # Determine ifun
145 int f_ifun = [
146         imem_error : FNONE;
147         1: imem_ifun;
```

```
148 ];
149
150
151 # Is instruction valid?
152 bool instr_valid = f_icode in
153         { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
154           IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
155
156 # Determine status code for fetched instruction
157 int f_stat = [
158         imem_error: SADR;
159         !instr_valid : SINS;
160         f_icode == IHALT : SHLT;
161         1 : SAOK;
162 ];
163
164 # Does fetched instruction require a regid byte?
165 bool need_regids =
166         f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
167                      IIRMOVL, IRMMOVL, IMRMOVL };
168
169 # Does fetched instruction require a constant word?
170 bool need_valC =
171         f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
172
173 # Predict next value of PC
174 int f_predPC = [
175         f_icode in { IJXX, ICALL } : f_valC;
176         1 : f_valP;
177 ];
178
179 ################ Decode Stage ######################################
180
181
182 ## What register should be used as the A source?
183 int d_srcA = [
184         D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
185         D_icode in { IPOPL, IRET } : RESP;
186         1 : RNONE; # Don't need register
187 ];
188
189 ## What register should be used as the B source?
190 int d_srcB = [
191         D_icode in { IOPL, IRMMOVL, IMRMOVL  } : D_rB;
192         D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
193         1 : RNONE;  # Don't need register
194 ];
195
196 ## What register should be used as the E destination?
197 int d_dstE = [
```

```
198         D_icode in { IRRMOVL, IIRMOVL, IOPL} : D_rB;
199         D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
200         1 : RNONE;  # Don't write any register
201 ];
202
203 ## What register should be used as the M destination?
204 int d_dstM = [
205         D_icode in { IMRMOVL, IPOPL } : D_rA;
206         1 : RNONE;  # Don't write any register
207 ];
208
209 ## What should be the A value?
210 ## Forward into decode stage for valA
211 int d_valA = [
212         D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
213         d_srcA == e_dstE : e_valE;    # Forward valE from execute
214         d_srcA == M_dstM : m_valM;    # Forward valM from memory
215         d_srcA == M_dstE : M_valE;    # Forward valE from memory
216         d_srcA == W_dstM : W_valM;    # Forward valM from write back
217         d_srcA == W_dstE : W_valE;    # Forward valE from write back
218         1 : d_rvalA;  # Use value read from register file
219 ];
220
221 int d_valB = [
222         d_srcB == e_dstE : e_valE;    # Forward valE from execute
223         d_srcB == M_dstM : m_valM;    # Forward valM from memory
224         d_srcB == M_dstE : M_valE;    # Forward valE from memory
225         d_srcB == W_dstM : W_valM;    # Forward valM from write back
226         d_srcB == W_dstE : W_valE;    # Forward valE from write back
227         1 : d_rvalB;  # Use value read from register file
228 ];
229
230 ################ Execute Stage ######################################
231
232 ## Select input A to ALU
233 int aluA = [
234         E_icode in { IRRMOVL, IOPL } : E_valA;
235         E_icode in { IIRMOVL, IRMMOVL, IMRMOVL } : E_valC;
236         E_icode in { ICALL, IPUSHL } : -4;
237         E_icode in { IRET, IPOPL } : 4;
238         # Other instructions don't need ALU
239 ];
240
241 ## Select input B to ALU
242 int aluB = [
243         E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
244                      IPUSHL, IRET, IPOPL } : E_valB;
245         E_icode in { IRRMOVL, IIRMOVL } : 0;
246         # Other instructions don't need ALU
247 ];
```

```
248
249 ## Set the ALU function
250 int alufun = [
251         E_icode == IOPL : E_ifun;
252         1 : ALUADD;
253 ];
254
255 ## Should the condition codes be updated?
256 bool set_cc = E_icode == IOPL &&
257         # State changes only during normal operation
258         !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS, SHLT };
259
260 ## Generate valA in execute stage
261 int e_valA = E_valA;    # Pass valA through stage
262
263 ## Set dstE to RNONE in event of not-taken conditional move
264 int e_dstE = [
265         E_icode == IRRMOVL && !e_Cnd : RNONE;
266         1 : E_dstE;
267 ];
268
269 ################ Memory Stage ######################################
270
271 ## Select memory address
272 int mem_addr = [
273         M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : M_valE;
274         M_icode in { IPOPL, IRET } : M_valA;
275         # Other instructions don't need address
276 ];
277
278 ## Set read control signal
279 bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET };
280
281 ## Set write control signal
282 bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };
283
284 #/* $begin pipe-m_stat-hcl */
285 ## Update the status
286 int m_stat = [
287         dmem_error : SADR;
288         1 : M_stat;
289 ];
290 #/* $end pipe-m_stat-hcl */
291
292 ## Set E port register ID
293 int w_dstE = W_dstE;
294
295 ## Set E port value
296 int w_valE = W_valE;
297
```

```
298 ## Set M port register ID
299 int w_dstM = W_dstM;
300
301 ## Set M port value
302 int w_valM = W_valM;
303
304 ## Update processor status
305 int Stat = [
306         W_stat == SBUB : SAOK;
307         1 : W_stat;
308 ];
309
310 ################ Pipeline Register Control ##########################
311
312 # Should I stall or inject a bubble into Pipeline Register F?
313 # At most one of these can be true.
314 bool F_bubble = 0;
315 bool F_stall =
316         # Conditions for a load/use hazard
317         E_icode in { IMRMOVL, IPOPL } &&
318          E_dstM in { d_srcA, d_srcB } ||
319         # Stalling at fetch while ret passes through pipeline
320         IRET in { D_icode, E_icode, M_icode };
321
322 # Should I stall or inject a bubble into Pipeline Register D?
323 # At most one of these can be true.
324 bool D_stall =
325         # Conditions for a load/use hazard
326         E_icode in { IMRMOVL, IPOPL } &&
327          E_dstM in { d_srcA, d_srcB };
328
329 bool D_bubble =
330         # Mispredicted branch
331         (E_icode == IJXX && !e_Cnd) ||
332         # Stalling at fetch while ret passes through pipeline
333         # but not condition for a load/use hazard
334         !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }) &&
335           IRET in { D_icode, E_icode, M_icode };
336
337 # Should I stall or inject a bubble into Pipeline Register E?
338 # At most one of these can be true.
339 bool E_stall = 0;
340 bool E_bubble =
341         # Mispredicted branch
342         (E_icode == IJXX && !e_Cnd) ||
343         # Conditions for a load/use hazard
344         E_icode in { IMRMOVL, IPOPL } &&
345          E_dstM in { d_srcA, d_srcB};
346
347 # Should I stall or inject a bubble into Pipeline Register M?
```

```
348 # At most one of these can be true.
349 bool M_stall = 0;
350 # Start injecting bubbles as soon as exception passes through memory stage
351 bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR, SINS, SHLT };
352
353 # Should I stall or inject a bubble into Pipeline Register W?
354 bool W_stall = W_stat in { SADR, SINS, SHLT };
355 bool W_bubble = 0;
```