

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.

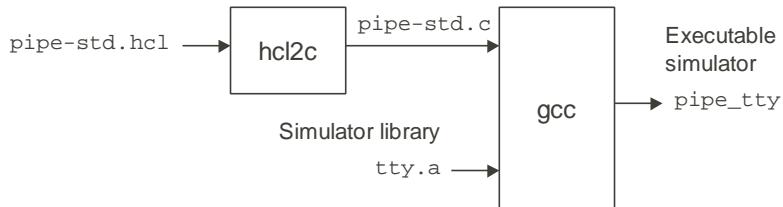
This document describes the Hardware Control Language, HCL, devised to provide a simple, yet systematic way to describe the control logic for Y86 processors. It also includes copies of the HCL descriptions of Y86 processors SEQ and PIPE. Electronic versions of these HCL files are available at www.csapp.cs.cmu.edu.

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:

*Copyright © 2010, R. E. Bryant, D. R. O'Hallaron. All rights reserved.



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
<i>name</i>	Named Boolean signal
<i>int-expr</i> in $\{int\text{-}expr_1, int\text{-}expr_2, \dots, int\text{-}expr_k\}$	Set membership test
<i>int-expr</i> ₁ == <i>int-expr</i> ₂	Equality test
<i>int-expr</i> ₁ != <i>int-expr</i> ₂	Not equal test
<i>int-expr</i> ₁ < <i>int-expr</i> ₂	Less than test
<i>int-expr</i> ₁ <= <i>int-expr</i> ₂	Less than or equal test
<i>int-expr</i> ₁ > <i>int-expr</i> ₂	Greater than test
<i>int-expr</i> ₁ >= <i>int-expr</i> ₂	Greater than or equal test
! <i>bool-expr</i>	NOT
<i>bool-expr</i> ₁ && <i>bool-expr</i> ₂	AND
<i>bool-expr</i> ₁ <i>bool-expr</i> 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 $\{int\text{-}expr_1, \dots, int\text{-}expr_k\}$, 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:

```
[  
  bool-expr1 : int-expr1  
  bool-expr2 : int-expr2  
  ...  
  bool-exprk : int-exprk  
]
```

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.

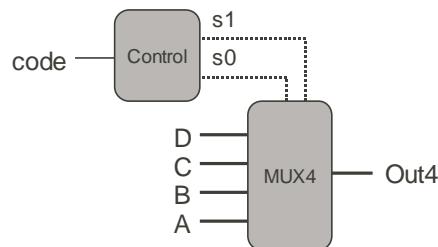
HCL expressions are used to define the behavior of a block of combinational logic. A block definition has one of the following forms:

```
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:



```

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

```

20 # Declarations. Do not change/remove/delete any of these      #
21 ##### Symbolic representation of Y86 Instruction Codes #####
22
23 ##### Symbolic representation of Y86 Instruction Codes #####
24 intsig INOP      'I_NOP'
25 intsig IHALT    'I_HALT'
26 intsig IRRMOVL  'I_RRMOVL'
27 intsig IIRMOVL  'I_IIRMOVL'
28 intsig IRMMOVL  'I_RMMOVL'
29 intsig IMRMOVL  'I_MRMOVL'
30 intsig IOPL     'I_ALU'
31 intsig IJXX     'I JMP'
32 intsig ICALL    'I_CALL'
33 intsig IRET     'I_RET'
34 intsig IPUSHL   'I_PUSHL'
35 intsig IPOPL    'I_POPL'
36
37 ##### Symbolic representations of Y86 function codes      #####
38 intsig FNONE    'F_NONE'          # Default function code
39
40 ##### Symbolic representation of Y86 Registers referenced explicitly #####
41 intsig RESP     'REG_ESP'        # Stack Pointer
42 intsig RNONE    'REG_NONE'       # Special value indicating "no register"
43
44 ##### ALU Functions referenced explicitly      #####
45 intsig ALUADD   'A_ADD'         # ALU should add its arguments
46
47 ##### Possible instruction status values      #####
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 ##### Fetch stage inputs      #####
56 intsig pc 'pc'            # Program counter
57 ##### Fetch stage computations #####
58 intsig imem_icode 'imem_icode' # icode field from instruction memory
59 intsig imem_ifun  'imem_ifun'  # ifun field from instruction memory
60 intsig icode     'icode'        # Instruction control code
61 intsig ifun      'ifun'         # Instruction function
62 intsig rA        'ra'           # rA field from instruction
63 intsig rB        'rb'           # rB field from instruction
64 intsig valC     'valc'         # Constant from instruction
65 intsig valP     'valp'         # Address of following instruction
66 boolsig imem_error 'imem_error' # Error signal from instruction memory
67 boolsig instr_valid 'instr_valid' # Is fetched instruction valid?
68
69 ##### Decode stage computations #####

```

```

70 intsig valA      'vala'                      # Value from register A port
71 intsig valB      'valb'                      # Value from register B port
72
73 ##### Execute stage computations          #####
74 intsig vale     'vale'                      # Value computed by ALU
75 boolsig Cnd     'cond'                      # Branch test
76
77 ##### Memory stage computations          #####
78 intsig valM     'valm'                      # Value read from memory
79 boolsig dmem_error 'dmem_error'            # Error signal from data memory
80
81
82 ##### Control Signal Definitions.        #####
83 #           #
84 ##### Fetch Stage                      #####
85
86 ##### Fetch Stage                      #####
87
88 # Determine instruction code
89 int icode = [
90     imem_error: INOP;
91     1: imem_icode;                      # Default: get from instruction memory
92 ];
93
94 # Determine instruction function
95 int ifun = [
96     imem_error: FNONE;
97     1: imem_ifun;                      # Default: get from instruction memory
98 ];
99
100 bool instr_valid = icode in
101     { INOP, IHALT, IIRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
102       IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
103
104 # Does fetched instruction require a regid byte?
105 bool need_regids =
106     icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
107                 IIRMOVL, IRMMOVL, IMRMOVL };
108
109 # Does fetched instruction require a constant word?
110 bool need_valC =
111     icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
112
113 ##### Decode Stage                      #####
114
115 ## What register should be used as the A source?
116 int srcA = [
117     icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
118     icode in { IPOPL, IRET } : RESP;
119     1 : RNONE; # Don't need register

```

```

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

```

```

170
171 ##### Memory Stage #####
172
173 ## Set read control signal
174 bool mem_read = icode in { IMRMOVL, IPOPL, IRET } ;
175
176 ## Set write control signal
177 bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL } ;
178
179 ## Select memory address
180 int mem_addr = [
181     icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
182     icode in { IPOPL, IRET } : valA;
183     # Other instructions don't need address
184 ];
185
186 ## Select memory input data
187 int mem_data = [
188     # Value from register
189     icode in { IRMMOVL, IPUSHL } : valA;
190     # Return PC
191     icode == ICALL : valP;
192     # Default: Don't write anything
193 ];
194
195 ## Determine instruction status
196 int Stat = [
197     imem_error || dmem_error : SADR;
198     !instr_valid: SINS;
199     icode == IHALT : SHLT;
200     1 : SAOK;
201 ];
202
203 ##### Program Counter Update #####
204
205 ## What address should instruction be fetched at
206
207 int new_pc = [
208     # Call. Use instruction constant
209     icode == ICALL : valC;
210     # Taken branch. Use instruction constant
211     icode == IJXX && Cnd : valC;
212     # Completion of RET instruction. Use value from stack
213     icode == IRET : valM;
214     # Default: Use incremented PC
215     1 : valP;
216 ];
217 /* $end seq-all-hcl */

```

3 PIPE

```

1 /* $begin pipe-all-hcl */
2 ##### HCL Description of Control for Pipelined Y86 Processor #
3 # Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010 #
4 ##### #
5 ##### C Include's. Don't alter these #
6 ##### #
7 # Declarations. Do not change/remove/delete any of these #
8 ##### #
9 ##### #
10 ##### Symbolic representation of Y86 Instruction Codes #####
11 quote '#include <stdio.h>'
12 quote '#include "isa.h"'
13 quote '#include "pipeline.h"'
14 quote '#include "stages.h"'
15 quote '#include "sim.h"'
16 quote 'int sim_main(int argc, char *argv[]);'
17 quote 'int main(int argc, char *argv[]){return sim_main(argc,argv);}'
18 #
19 ##### Declaration of Y86 Instructions #####
20 # Declarations. Do not change/remove/delete any of these #
21 ##### #
22 ##### Symbolic representations of Y86 function codes #####
23 intsig INOP      'I_NOP'
24 intsig IHALT     'I_HALT'
25 intsig IRRMOVL   'I_RRMOVL'
26 intsig IIRMOVL   'I_IIRMOVL'
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 representations of Y86 Registers referenced #####
37 intsig FNONE     'F_NONE'      # Default function code
38 #
39 ##### ALU Functions referenced explicitly #####
40 intsig RESP       'REG_ESP'     # Stack Pointer
41 intsig RNONE     'REG_NONE'    # Special value indicating "no register"
42 #
43 ##### Possible instruction status values #####
44 intsig ALUADD    'A_ADD'      # ALU should add its arguments
45 #
46 #
47 ##### 
```

```

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

```

```

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

```

```

148         1: imem_ifun;
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;  
356 /* $end pipe-all-hcl */
```