CS:APP3e Web Aside ARCH:HCL: HCL Descriptions of Y86-64 Processors*

Randal E. Bryant
David R. O'Hallaron

December 29, 2014

Notice

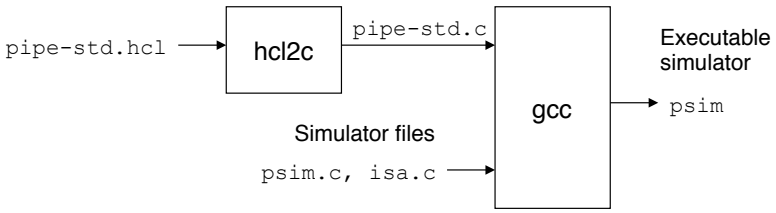
The material in this document is supplementary material to the book Computer Systems, A Programmer's Perspective, Third Edition*, by Randal E. Bryant and David R. O'Hallaron, published by Prentice-Hall and copyrighted 2016. In this document, all references beginning with "CS:APP3e " are to this book. More information about the book is available at* `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 Y86-64 processors. It also includes copies of the HCL descriptions of Y86-64 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 with source code implementing the other simulator functions to generate an executable simulation program. This is diagrammed below for generating the simulator for the pipeline processor



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

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

[1] C data type `long long int` was introduced in ISO C99 to provide an integer data type of at least 64 bits.

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.

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-expr*$_i$, indicating whether this case should be selected, and an integer expression *int-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.

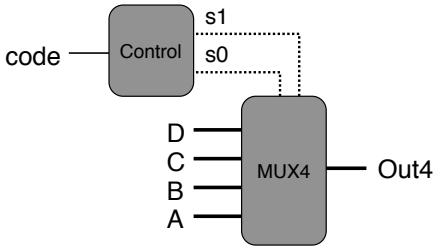
HCL expressions are used to define the behavior of a block of combinational logic. A block definition has one of the following forms:

```
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:APP3e 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 'long long 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 wordsig code 'code_val'
25 wordsig  A 'atoll(data_names[0])'
26 wordsig  B 'atoll(data_names[1])'
27 wordsig  C 'atoll(data_names[2])'
28 wordsig  D 'atoll(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 word 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 = atoll(argv[1]);'
47 quote '  s1_val = gen_s1();'
48 quote '  s0_val = gen_s0();'
49 quote '  printf("Out = %lld\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:

```
long long 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 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 = atoll(argv[1]);
  s1_val = gen_s1();
  s0_val = gen_s0();
  printf("Out = %lld\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

The following is the complete HCL code for the SEQ processor.

```
1  #/* $begin seq-all-hcl */
2  ####################################################################
3  #  HCL Description of Control for Single Cycle Y86-64 Processor SEQ   #
4  #  Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010       #
5  ####################################################################
6
7  ####################################################################
8  #    C Include's.  Don't alter these                               #
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 'word_t 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 ####################################################################
22
```

```
23 ##### Symbolic representation of Y86-64 Instruction Codes #############
24 wordsig INOP    'I_NOP'
25 wordsig IHALT   'I_HALT'
26 wordsig IRRMOVQ 'I_RRMOVQ'
27 wordsig IIRMOVQ 'I_IRMOVQ'
28 wordsig IRMMOVQ 'I_RMMOVQ'
29 wordsig IMRMOVQ 'I_MRMOVQ'
30 wordsig IOPQ    'I_ALU'
31 wordsig IJXX    'I_JMP'
32 wordsig ICALL   'I_CALL'
33 wordsig IRET    'I_RET'
34 wordsig IPUSHQ  'I_PUSHQ'
35 wordsig IPOPQ   'I_POPQ'
36
37 ##### Symbolic represenations of Y86-64 function codes                  #####
38 wordsig FNONE    'F_NONE'        # Default function code
39
40 ##### Symbolic representation of Y86-64 Registers referenced explicitly #####
41 wordsig RRSP     'REG_RSP'      # Stack Pointer
42 wordsig RNONE    'REG_NONE'     # Special value indicating "no register"
43
44 ##### ALU Functions referenced explicitly                              #####
45 wordsig ALUADD  'A_ADD'         # ALU should add its arguments
46
47 ##### Possible instruction status values                              #####
48 wordsig SAOK    'STAT_AOK'      # Normal execution
49 wordsig SADR    'STAT_ADR'      # Invalid memory address
50 wordsig SINS    'STAT_INS'      # Invalid instruction
51 wordsig SHLT    'STAT_HLT'      # Halt instruction encountered
52
53 ##### Signals that can be referenced by control logic ####################
54
55 ##### Fetch stage inputs                #####
56 wordsig pc 'pc'                         # Program counter
57 ##### Fetch stage computations          #####
58 wordsig imem_icode 'imem_icode'         # icode field from instruction memory
59 wordsig imem_ifun  'imem_ifun'          # ifun field from instruction memory
60 wordsig icode     'icode'              # Instruction control code
61 wordsig ifun      'ifun'               # Instruction function
62 wordsig rA        'ra'                 # rA field from instruction
63 wordsig rB        'rb'                 # rB field from instruction
64 wordsig valC      'valc'               # Constant from instruction
65 wordsig 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 wordsig valA    'vala'                 # Value from register A port
71 wordsig valB    'valb'                 # Value from register B port
72
```

```
73 ##### Execute stage computations        #####
74 wordsig valE    'vale'                  # Value computed by ALU
75 boolsig Cnd     'cond'                  # Branch test
76
77 ##### Memory stage computations         #####
78 wordsig valM    'valm'                  # Value read from memory
79 boolsig dmem_error 'dmem_error'         # Error signal from data memory
80
81
82 ################################################################
83 #    Control Signal Definitions.                               #
84 ################################################################
85
86 ################ Fetch Stage     ###################################
87
88 # Determine instruction code
89 word icode = [
90         imem_error: INOP;
91         1: imem_icode;          # Default: get from instruction memory
92 ];
93
94 # Determine instruction function
95 word 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, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
102                IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ };
103
104 # Does fetched instruction require a regid byte?
105 bool need_regids =
106         icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
107                       IIRMOVQ, IRMMOVQ, IMRMOVQ };
108
109 # Does fetched instruction require a constant word?
110 bool need_valC =
111         icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL };
112
113 ################ Decode Stage    ###################################
114
115 ## What register should be used as the A source?
116 word srcA = [
117         icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : rA;
118         icode in { IPOPQ, IRET } : RRSP;
119         1 : RNONE; # Don't need register
120 ];
121
122 ## What register should be used as the B source?
```

```
123 word srcB = [
124         icode in { IOPQ, IRMMOVQ, IMRMOVQ  } : rB;
125         icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
126         1 : RNONE;  # Don't need register
127 ];
128
129 ## What register should be used as the E destination?
130 word dstE = [
131         icode in { IRRMOVQ } && Cnd : rB;
132         icode in { IIRMOVQ, IOPQ} : rB;
133         icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
134         1 : RNONE;  # Don't write any register
135 ];
136
137 ## What register should be used as the M destination?
138 word dstM = [
139         icode in { IMRMOVQ, IPOPQ } : rA;
140         1 : RNONE;  # Don't write any register
141 ];
142
143 ################ Execute Stage   ###################################
144
145 ## Select input A to ALU
146 word aluA = [
147         icode in { IRRMOVQ, IOPQ } : valA;
148         icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
149         icode in { ICALL, IPUSHQ } : -8;
150         icode in { IRET, IPOPQ } : 8;
151         # Other instructions don't need ALU
152 ];
153
154 ## Select input B to ALU
155 word aluB = [
156         icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
157                         IPUSHQ, IRET, IPOPQ } : valB;
158         icode in { IRRMOVQ, IIRMOVQ } : 0;
159         # Other instructions don't need ALU
160 ];
161
162 ## Set the ALU function
163 word alufun = [
164         icode == IOPQ : ifun;
165         1 : ALUADD;
166 ];
167
168 ## Should the condition codes be updated?
169 bool set_cc = icode in { IOPQ };
170
171 ################ Memory Stage    ###################################
172
```

```
173 ## Set read control signal
174 bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
175
176 ## Set write control signal
177 bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
178
179 ## Select memory address
180 word mem_addr = [
181         icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
182         icode in { IPOPQ, IRET } : valA;
183         # Other instructions don't need address
184 ];
185
186 ## Select memory input data
187 word mem_data = [
188         # Value from register
189         icode in { IRMMOVQ, IPUSHQ } : valA;
190         # Return PC
191         icode == ICALL : valP;
192         # Default: Don't write anything
193 ];
194
195 ## Determine instruction status
196 word 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 word 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

The following is the complete HCL code for the PIPE processor.

```
#/* $begin pipe-all-hcl */
################################################################
#    HCL Description of Control for Pipelined Y86-64 Processor     #
#    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2014     #
################################################################

################################################################
#    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 Y86-64 Instruction Codes #############
wordsig INOP    'I_NOP'
wordsig IHALT   'I_HALT'
wordsig IRRMOVQ 'I_RRMOVQ'
wordsig IIRMOVQ 'I_IRMOVQ'
wordsig IRMMOVQ 'I_RMMOVQ'
wordsig IMRMOVQ 'I_MRMOVQ'
wordsig IOPQ    'I_ALU'
wordsig IJXX    'I_JMP'
wordsig ICALL   'I_CALL'
wordsig IRET    'I_RET'
wordsig IPUSHQ  'I_PUSHQ'
wordsig IPOPQ   'I_POPQ'

##### Symbolic represenations of Y86-64 function codes            #####
wordsig FNONE    'F_NONE'        # Default function code

##### Symbolic representation of Y86-64 Registers referenced      #####
wordsig RRSP     'REG_RSP'           # Stack Pointer
wordsig RNONE    'REG_NONE'          # Special value indicating "no register"

##### ALU Functions referenced explicitly ##########################
wordsig ALUADD  'A_ADD'              # ALU should add its arguments
```

```
46
47 ##### Possible instruction status values                        #####
48 wordsig SBUB    'STAT_BUB'      # Bubble in stage
49 wordsig SAOK    'STAT_AOK'      # Normal execution
50 wordsig SADR    'STAT_ADR'      # Invalid memory address
51 wordsig SINS    'STAT_INS'      # Invalid instruction
52 wordsig SHLT    'STAT_HLT'      # Halt instruction encountered
53
54 ##### Signals that can be referenced by control logic ##############
55
56 ##### Pipeline Register F ########################################
57
58 wordsig F_predPC 'pc_curr->pc'       # Predicted value of PC
59
60 ##### Intermediate Values in Fetch Stage #########################
61
62 wordsig imem_icode  'imem_icode'      # icode field from instruction memory
63 wordsig imem_ifun   'imem_ifun'       # ifun  field from instruction memory
64 wordsig f_icode 'if_id_next->icode'  # (Possibly modified) instruction code
65 wordsig f_ifun  'if_id_next->ifun'   # Fetched instruction function
66 wordsig f_valC  'if_id_next->valc'   # Constant data of fetched instruction
67 wordsig 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 wordsig D_icode 'if_id_curr->icode'   # Instruction code
73 wordsig D_rA 'if_id_curr->ra'        # rA field from instruction
74 wordsig D_rB 'if_id_curr->rb'        # rB field from instruction
75 wordsig D_valP 'if_id_curr->valp'     # Incremented PC
76
77 ##### Intermediate Values in Decode Stage  #######################
78
79 wordsig d_srcA   'id_ex_next->srca'  # srcA from decoded instruction
80 wordsig d_srcB   'id_ex_next->srcb'  # srcB from decoded instruction
81 wordsig d_rvalA 'd_regvala'          # valA read from register file
82 wordsig d_rvalB 'd_regvalb'          # valB read from register file
83
84 ##### Pipeline Register E ########################################
85 wordsig E_icode 'id_ex_curr->icode'   # Instruction code
86 wordsig E_ifun  'id_ex_curr->ifun'    # Instruction function
87 wordsig E_valC  'id_ex_curr->valc'    # Constant data
88 wordsig E_srcA  'id_ex_curr->srca'    # Source A register ID
89 wordsig E_valA  'id_ex_curr->vala'    # Source A value
90 wordsig E_srcB  'id_ex_curr->srcb'    # Source B register ID
91 wordsig E_valB  'id_ex_curr->valb'    # Source B value
92 wordsig E_dstE 'id_ex_curr->deste'    # Destination E register ID
93 wordsig E_dstM 'id_ex_curr->destm'    # Destination M register ID
94
95 ##### Intermediate Values in Execute Stage #######################
```

```
96 wordsig e_valE 'ex_mem_next->vale'      # valE generated by ALU
97 boolsig e_Cnd 'ex_mem_next->takebranch' # Does condition hold?
98 wordsig e_dstE 'ex_mem_next->deste'      # dstE (possibly modified to be RNONE)
99
100 ##### Pipeline Register M                  ########################
101 wordsig M_stat 'ex_mem_curr->status'     # Instruction status
102 wordsig M_icode 'ex_mem_curr->icode'     # Instruction code
103 wordsig M_ifun  'ex_mem_curr->ifun'      # Instruction function
104 wordsig M_valA  'ex_mem_curr->vala'      # Source A value
105 wordsig M_dstE 'ex_mem_curr->deste'      # Destination E register ID
106 wordsig M_valE  'ex_mem_curr->vale'      # ALU E value
107 wordsig 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 wordsig m_valM 'mem_wb_next->valm'       # valM generated by memory
113 wordsig m_stat 'mem_wb_next->status'     # stat (possibly modified to be SADR)
114
115 ##### Pipeline Register W ###########################################
116 wordsig W_stat 'mem_wb_curr->status'     # Instruction status
117 wordsig W_icode 'mem_wb_curr->icode'     # Instruction code
118 wordsig W_dstE 'mem_wb_curr->deste'      # Destination E register ID
119 wordsig W_valE  'mem_wb_curr->vale'      # ALU E value
120 wordsig W_dstM 'mem_wb_curr->destm'      # Destination M register ID
121 wordsig W_valM  'mem_wb_curr->valm'      # Memory M value
122
123 ####################################################################
124 #    Control Signal Definitions.                                   #
125 ####################################################################
126
127 ################ Fetch Stage     ###################################
128
129 ## What address should instruction be fetched at
130 word 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 word f_icode = [
141         imem_error : INOP;
142         1: imem_icode;
143 ];
144
145 # Determine ifun
```

```
146 word 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, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
154           IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ };
155
156 # Determine status code for fetched instruction
157 word 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 { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
167                      IIRMOVQ, IRMMOVQ, IMRMOVQ };
168
169 # Does fetched instruction require a constant word?
170 bool need_valC =
171         f_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL };
172
173 # Predict next value of PC
174 word 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 word d_srcA = [
184         D_icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : D_rA;
185         D_icode in { IPOPQ, IRET } : RRSP;
186         1 : RNONE; # Don't need register
187 ];
188
189 ## What register should be used as the B source?
190 word d_srcB = [
191         D_icode in { IOPQ, IRMMOVQ, IMRMOVQ  } : D_rB;
192         D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
193         1 : RNONE;  # Don't need register
194 ];
195
```

```
196 ## What register should be used as the E destination?
197 word d_dstE = [
198         D_icode in { IRRMOVQ, IIRMOVQ, IOPQ} : D_rB;
199         D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
200         1 : RNONE;  # Don't write any register
201 ];
202
203 ## What register should be used as the M destination?
204 word d_dstM = [
205         D_icode in { IMRMOVQ, IPOPQ } : 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 word 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 word 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 word aluA = [
234         E_icode in { IRRMOVQ, IOPQ } : E_valA;
235         E_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : E_valC;
236         E_icode in { ICALL, IPUSHQ } : -8;
237         E_icode in { IRET, IPOPQ } : 8;
238         # Other instructions don't need ALU
239 ];
240
241 ## Select input B to ALU
242 word aluB = [
243         E_icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
244                      IPUSHQ, IRET, IPOPQ } : E_valB;
245         E_icode in { IRRMOVQ, IIRMOVQ } : 0;
```

```
246         # Other instructions don't need ALU
247 ];
248
249 ## Set the ALU function
250 word alufun = [
251         E_icode == IOPQ : E_ifun;
252         1 : ALUADD;
253 ];
254
255 ## Should the condition codes be updated?
256 bool set_cc = E_icode == IOPQ &&
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 word e_valA = E_valA;    # Pass valA through stage
262
263 ## Set dstE to RNONE in event of not-taken conditional move
264 word e_dstE = [
265         E_icode == IRRMOVQ && !e_Cnd : RNONE;
266         1 : E_dstE;
267 ];
268
269 ################ Memory Stage ######################################
270
271 ## Select memory address
272 word mem_addr = [
273         M_icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : M_valE;
274         M_icode in { IPOPQ, IRET } : M_valA;
275         # Other instructions don't need address
276 ];
277
278 ## Set read control signal
279 bool mem_read = M_icode in { IMRMOVQ, IPOPQ, IRET };
280
281 ## Set write control signal
282 bool mem_write = M_icode in { IRMMOVQ, IPUSHQ, ICALL };
283
284 #/* $begin pipe-m_stat-hcl */
285 ## Update the status
286 word 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 word w_dstE = W_dstE;
294
295 ## Set E port value
```

```
296 word w_valE = W_valE;
297
298 ## Set M port register ID
299 word w_dstM = W_dstM;
300
301 ## Set M port value
302 word w_valM = W_valM;
303
304 ## Update processor status
305 word 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 { IMRMOVQ, IPOPQ } &&
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 { IMRMOVQ, IPOPQ } &&
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 { IMRMOVQ, IPOPQ } && 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 { IMRMOVQ, IPOPQ } &&
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 */
```