

Homework 8 & Homework 9

郭天魁
信息科学技术学院
1300012790

November 4, 2014

1 Homework 8

1.1 4.51

若在EMW三个阶段中将要当前的d_srcA,d_srcB进行写入,即发生了数据冒险,则将FD暂停,并在E处插入bubble。

需要注意:

1. 在D阶段中,如果是错误预测分支,那么应该插入bubble而非暂停。
2. 为正确处理conditional move,应用e_dstE。

具体代码见附录A或见附件。

1.2 4.55

错误预测仅可能在icode为IJXX且ifun不为UNCOND(即题中的J_YES)时发生,将要跳转到的位置应从valC与valP中选择。valP通过valA传递,我们将valC通过valE(在ALU中 $valE=valC+0$)传递。

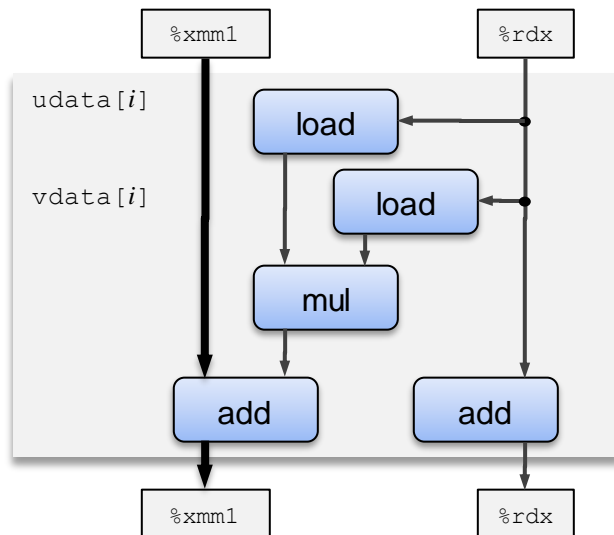
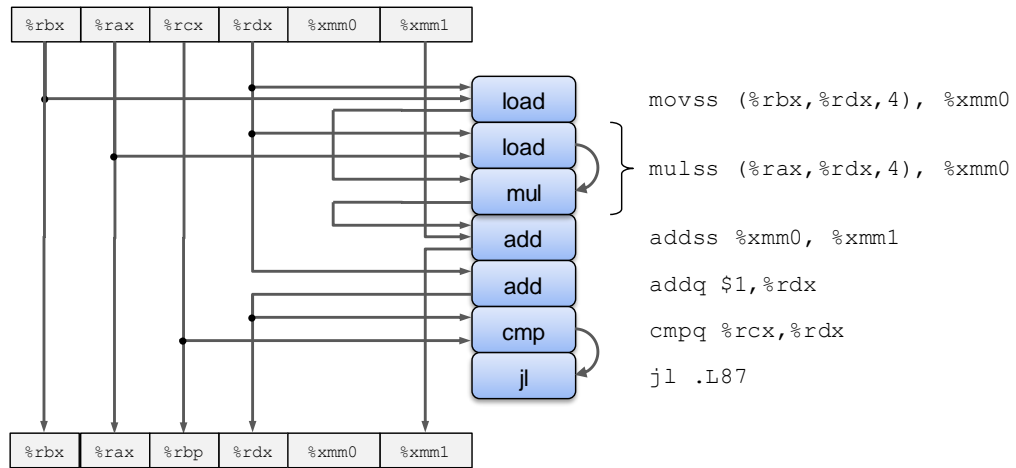
没什么需要注意的地方,把所有错误预测分支的情况加上对valP和valC(valA和valE)的大小判断,并判断ifun是否为UNCOND即可。在计算PC时,如果是错误预测分支,那么应该选择M_valE和M_valA中较大的那个,因为预测的是较小的那个。

具体代码见附录B或见附件。

2 Homework 9

2.1 5.15

A. 不得不说用ppt画图是我这辈子做过的最蠢的事情了……



加粗的即为关键路径。

B. 关键路径决定的CPE下界为float加法的延迟3.0。

C. 关键路径决定的CPE下界为int加法的延迟1.0。

D. 关键路径决定的CPE下界为浮点数加法的延迟3.0。对于需要更多延迟的浮点数乘法，由于其不在循环存储器的数据相关链中，不需要等待前一次迭代的累积值就可以执行，从而消除了瓶颈。

2.2 5.21

```
1 void psum2(float a[], float p[], long int n)
2 {
3     long int i;
4     float last_val, val;
5     last_val = p[0] = a[0];
6     for (i = 1; i < n - 1; i += 2) {
7         p[i] = last_val + a[i];
8         val = a[i] + a[i + 1];
9         val = last_val + val;
10        p[i + 1] = val;
11        last_val = val;
12    }
13    while(i < n) {
14        val = last_val + a[i];
15        p[i] = val;
16        last_val = val;
17    }
18 }
```

A pipe-nobypass.hcl

```
1  /* $begin pipe-all-hcl */
2  #####
3  #    HCL Description of Control for Pipelined Y86 Processor      #
4  #    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010  #
5  #####
6
7  ## Your task is to make the pipeline work without using any
   forwarding
8  ## The normal bypassing logic in the file is disabled.
9  ## You can only change the pipeline control logic at the end of this
   file.
10 ## The trick is to make the pipeline stall whenever there is a data
    hazard.
11
12 #####
13 #    C Include's.  Don't alter these                             #
14 #####
15
16 quote '#include <stdio.h>'
17 quote '#include "isa.h"'
18 quote '#include "pipeline.h"'
19 quote '#include "stages.h"'
20 quote '#include "sim.h"'
21 quote 'int sim_main(int argc, char *argv[]);'
22 quote 'int main(int argc, char *argv[]){return sim_main(argc,argv)
    ;}'
23
24 #####
25 #    Declarations.  Do not change/remove/delete any of these      #
26 #####
27
28 ##### Symbolic representation of Y86 Instruction Codes #####
29 intsig INOP      'I_NOP'
30 intsig IHALT     'I_HALT'
31 intsig IRRMOVL   'I_RRMOVL'
32 intsig IIRMOVL   'I_IRMOVL'
33 intsig IRMMOVL   'I_RMMOVL'
34 intsig IMRMOVL   'I_MRMOVL'
35 intsig IOPL      'I_ALU'
36 intsig IJXX      'I_JMP'
37 intsig ICALL     'I_CALL'
38 intsig IRET      'I_RET'
39 intsig IPUSHL    'I_PUSHL'
40 intsig IPOPL     'I_POPL'
```

```

41
42 ##### Symbolic representations of Y86 function codes #####
43 intsig FNONE 'F_NONE' # Default function code
44
45 ##### Symbolic representation of Y86 Registers referenced #####
46 intsig RESP 'REG_ESP' # Stack Pointer
47 intsig RNONE 'REG_NONE' # Special value indicating "no
    register"
48
49 ##### ALU Functions referenced explicitly #####
50 intsig ALUADD 'A_ADD' # ALU should add its arguments
51
52 ##### Possible instruction status values #####
53 intsig SBUB 'STAT_BUB' # Bubble in stage
54 intsig SAOK 'STAT_AOK' # Normal execution
55 intsig SADR 'STAT_ADR' # Invalid memory address
56 intsig SINS 'STAT_INS' # Invalid instruction
57 intsig SHLT 'STAT_HLT' # Halt instruction encountered
58
59 ##### Signals that can be referenced by control logic #####
60
61 ##### Pipeline Register F #####
62
63 intsig F_predPC 'pc_curr->pc' # Predicted value of PC
64
65 ##### Intermediate Values in Fetch Stage #####
66
67 intsig imem_icode 'imem_icode' # icode field from instruction
    memory
68 intsig imem_ifun 'imem_ifun' # ifun field from instruction
    memory
69 intsig f_icode 'if_id_next->icode' # (Possibly modified)
    instruction code
70 intsig f_ifun 'if_id_next->ifun' # Fetched instruction function
71 intsig f_valC 'if_id_next->valc' # Constant data of fetched
    instruction
72 intsig f_valP 'if_id_next->valp' # Address of following
    instruction
73 boolsig imem_error 'imem_error' # Error signal from instruction
    memory
74 boolsig instr_valid 'instr_valid' # Is fetched instruction valid?
75
76 ##### Pipeline Register D #####
77 intsig D_icode 'if_id_curr->icode' # Instruction code
78 intsig D_rA 'if_id_curr->ra' # rA field from instruction
79 intsig D_rB 'if_id_curr->rb' # rB field from instruction

```

```

80 intsig D_valP 'if_id_curr->valp'      # Incremented PC
81
82 ##### Intermediate Values in Decode Stage #####
83
84 intsig d_srcA   'id_ex_next->srca'     # srcA from decoded instruction
85 intsig d_srcB   'id_ex_next->srcb'     # srcB from decoded instruction
86 intsig d_rvalA  'd_regvala'           # valA read from register file
87 intsig d_rvalB  'd_regvalb'           # valB read from register file
88
89 ##### Pipeline Register E #####
90 intsig E_icode  'id_ex_curr->icode'    # Instruction code
91 intsig E_ifun   'id_ex_curr->ifun'     # Instruction function
92 intsig E_valC   'id_ex_curr->valc'     # Constant data
93 intsig E_srcA   'id_ex_curr->srca'     # Source A register ID
94 intsig E_valA   'id_ex_curr->vala'     # Source A value
95 intsig E_srcB   'id_ex_curr->srcb'     # Source B register ID
96 intsig E_valB   'id_ex_curr->valb'     # Source B value
97 intsig E_dstE   'id_ex_curr->deste'    # Destination E register ID
98 intsig E_dstM   'id_ex_curr->destm'    # Destination M register ID
99
100 ##### Intermediate Values in Execute Stage #####
101 intsig e_valE   'ex_mem_next->vale'    # valE generated by ALU
102 boolsig e_Cnd   'ex_mem_next->takebranch' # Does condition hold?
103 intsig e_dstE   'ex_mem_next->deste'    # dstE (possibly modified to
    be RNONE)
104
105 ##### Pipeline Register M #####
106 intsig M_stat   'ex_mem_curr->status'   # Instruction status
107 intsig M_icode  'ex_mem_curr->icode'    # Instruction code
108 intsig M_ifun   'ex_mem_curr->ifun'     # Instruction function
109 intsig M_valA   'ex_mem_curr->vala'     # Source A value
110 intsig M_dstE   'ex_mem_curr->deste'    # Destination E register ID
111 intsig M_valE   'ex_mem_curr->vale'     # ALU E value
112 intsig M_dstM   'ex_mem_curr->destm'    # Destination M register ID
113 boolsig M_Cnd   'ex_mem_curr->takebranch' # Condition flag
114 boolsig dmem_error 'dmem_error'        # Error signal from
    instruction memory
115
116 ##### Intermediate Values in Memory Stage #####
117 intsig m_valM   'mem_wb_next->valm'    # valM generated by memory
118 intsig m_stat   'mem_wb_next->status'   # stat (possibly modified to
    be SADR)
119
120 ##### Pipeline Register W #####
121 intsig W_stat   'mem_wb_curr->status'   # Instruction status
122 intsig W_icode  'mem_wb_curr->icode'    # Instruction code

```

```

123 intsig W_dstE 'mem_wb_curr->deste'      # Destination E register ID
124 intsig W_vale 'mem_wb_curr->vale'      # ALU E value
125 intsig W_dstM 'mem_wb_curr->destm'      # Destination M register ID
126 intsig W_valM 'mem_wb_curr->valm'      # Memory M value
127
128 #####
129 #      Control Signal Definitions.      #
130 #####
131
132 ##### Fetch Stage #####
133
134 ## What address should instruction be fetched at
135 int f_pc = [
136     # Mispredicted branch. Fetch at incremented PC
137     M_icode == IJXX && !M_Cnd : M_valA;
138     # Completion of RET instruction.
139     W_icode == IRET : W_valM;
140     # Default: Use predicted value of PC
141     1 : F_predPC;
142 ];
143
144 ## Determine icode of fetched instruction
145 int f_icode = [
146     imem_error : INOP;
147     1: imem_icode;
148 ];
149
150 # Determine ifun
151 int f_ifun = [
152     imem_error : FNONE;
153     1: imem_ifun;
154 ];
155
156 # Is instruction valid?
157 bool instr_valid = f_icode in
158     { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
159       IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
160
161 # Determine status code for fetched instruction
162 int f_stat = [
163     imem_error: SADR;
164     !instr_valid : SINS;
165     f_icode == IHALT : SHLT;
166     1 : SAOK;
167 ];
168

```

```

169 # Does fetched instruction require a regid byte?
170 bool need_regids =
171     f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
172                IIRMOVL, IRMMOVL, IMRMOVL };
173
174 # Does fetched instruction require a constant word?
175 bool need_valC =
176     f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
177
178 # Predict next value of PC
179 int f_predPC = [
180     f_icode in { IJXX, ICALL } : f_valC;
181     1 : f_valP;
182 ];
183
184 ##### Decode Stage #####
185
186
187 ## What register should be used as the A source?
188 int d_srcA = [
189     D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : D_rA;
190     D_icode in { IPOPL, IRET } : RESP;
191     1 : RNONE; # Don't need register
192 ];
193
194 ## What register should be used as the B source?
195 int d_srcB = [
196     D_icode in { IOPL, IRMMOVL, IMRMOVL } : D_rB;
197     D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
198     1 : RNONE; # Don't need register
199 ];
200
201 ## What register should be used as the E destination?
202 int d_dstE = [
203     D_icode in { IRRMOVL, IIRMOVL, IOPL } : D_rB;
204     D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
205     1 : RNONE; # Don't write any register
206 ];
207
208 ## What register should be used as the M destination?
209 int d_dstM = [
210     D_icode in { IMRMOVL, IPOPL } : D_rA;
211     1 : RNONE; # Don't write any register
212 ];
213
214 ## What should be the A value?

```



```

215 ## DO NOT MODIFY THE FOLLOWING CODE.
216 ## No forwarding.  valA is either valP or value from register file
217 int d_valA = [
218     D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
219     1 : d_rvalA; # Use value read from register file
220 ];
221
222 ## No forwarding.  valB is value from register file
223 int d_valB = d_rvalB;
224
225 ##### Execute Stage #####
226
227 ## Select input A to ALU
228 int aluA = [
229     E_icode in { IRRMOVL, IOPL } : E_valA;
230     E_icode in { IIRMOVL, IRMMOVL, IMRMOVL } : E_valC;
231     E_icode in { ICALL, IPUSHL } : -4;
232     E_icode in { IRET, IPOPL } : 4;
233     # Other instructions don't need ALU
234 ];
235
236 ## Select input B to ALU
237 int aluB = [
238     E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
239                 IPUSHL, IRET, IPOPL } : E_valB;
240     E_icode in { IRRMOVL, IIRMOVL } : 0;
241     # Other instructions don't need ALU
242 ];
243
244 ## Set the ALU function
245 int alufun = [
246     E_icode == IOPL : E_ifun;
247     1 : ALUADD;
248 ];
249
250 ## Should the condition codes be updated?
251 bool set_cc = E_icode == IOPL &&
252     # State changes only during normal operation
253     !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS,
254         SHLT };
255
256 ## Generate valA in execute stage
257 int e_valA = E_valA; # Pass valA through stage
258
259 ## Set dstE to RNONE in event of not-taken conditional move
260 int e_dstE = [

```

```

260         E_icode == IRRMOVL && !e_Cnd : RNONE;
261         1 : E_dstE;
262 ];
263
264 ##### Memory Stage #####
265
266 ## Select memory address
267 int mem_addr = [
268     M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : M_vale;
269     M_icode in { IPOPL, IRET } : M_valA;
270     # Other instructions don't need address
271 ];
272
273 ## Set read control signal
274 bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET };
275
276 ## Set write control signal
277 bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };
278
279 /* $begin pipe-m_stat-hcl */
280 ## Update the status
281 int m_stat = [
282     dmem_error : SADR;
283     1 : M_stat;
284 ];
285 /* $end pipe-m_stat-hcl */
286
287 ## Set E port register ID
288 int w_dstE = W_dstE;
289
290 ## Set E port value
291 int w_vale = W_vale;
292
293 ## Set M port register ID
294 int w_dstM = W_dstM;
295
296 ## Set M port value
297 int w_valM = W_valM;
298
299 ## Update processor status
300 int Stat = [
301     W_stat == SBUB : SAOK;
302     1 : W_stat;
303 ];
304
305 ##### Pipeline Register Control #####

```

```

306
307 # Should I stall or inject a bubble into Pipeline Register F?
308 # At most one of these can be true.
309 bool F_bubble = 0;
310 bool F_stall =
311     # Modify the following to stall the update of pipeline
        register F
312     e_dstE != RNONE && e_dstE in { d_srcA, d_srcB } ||
313     E_dstM != RNONE && E_dstM in { d_srcA, d_srcB } ||
314     M_dstE != RNONE && M_dstE in { d_srcA, d_srcB } ||
315     M_dstM != RNONE && M_dstM in { d_srcA, d_srcB } ||
316     W_dstE != RNONE && W_dstE in { d_srcA, d_srcB } ||
317     W_dstM != RNONE && W_dstM in { d_srcA, d_srcB } ||
318     # Stalling at fetch while ret passes through pipeline
319     IRET in { D_icode, E_icode, M_icode };
320
321 # Should I stall or inject a bubble into Pipeline Register D?
322 # At most one of these can be true.
323 bool D_stall =
324     # Modify the following to stall the instruction in decode
325     (e_dstE != RNONE && e_dstE in { d_srcA, d_srcB } ||
326     E_dstM != RNONE && E_dstM in { d_srcA, d_srcB } ||
327     M_dstE != RNONE && M_dstE in { d_srcA, d_srcB } ||
328     M_dstM != RNONE && M_dstM in { d_srcA, d_srcB } ||
329     W_dstE != RNONE && W_dstE in { d_srcA, d_srcB } ||
330     W_dstM != RNONE && W_dstM in { d_srcA, d_srcB }) &&
331     (!(E_icode == IJXX && !e_Cnd)) ||
332     0;
333
334 bool D_bubble =
335     # Mispredicted branch
336     (E_icode == IJXX && !e_Cnd) ||
337     # Stalling at fetch while ret passes through pipeline
338     # but not condition for a generate/use hazard
339     !(e_dstE != RNONE && e_dstE in { d_srcA, d_srcB } ||
340     E_dstM != RNONE && E_dstM in { d_srcA, d_srcB } ||
341     M_dstE != RNONE && M_dstE in { d_srcA, d_srcB } ||
342     M_dstM != RNONE && M_dstM in { d_srcA, d_srcB } ||
343     W_dstE != RNONE && W_dstE in { d_srcA, d_srcB } ||
344     W_dstM != RNONE && W_dstM in { d_srcA, d_srcB }) &&
345     IRET in { D_icode, E_icode, M_icode };
346
347 # Should I stall or inject a bubble into Pipeline Register E?
348 # At most one of these can be true.
349 bool E_stall = 0;
350 bool E_bubble =

```

```

351         # Mispredicted branch
352         (E_icode == IJXX && !e_Cnd) ||
353         # Modify the following to inject bubble into the execute
           stage
354         e_dstE != RNONE && e_dstE in { d_srcA, d_srcB } ||
355         E_dstM != RNONE && E_dstM in { d_srcA, d_srcB } ||
356         M_dstE != RNONE && M_dstE in { d_srcA, d_srcB } ||
357         M_dstM != RNONE && M_dstM in { d_srcA, d_srcB } ||
358         W_dstE != RNONE && W_dstE in { d_srcA, d_srcB } ||
359         W_dstM != RNONE && W_dstM in { d_srcA, d_srcB } ||
360         0;
361
362     # Should I stall or inject a bubble into Pipeline Register M?
363     # At most one of these can be true.
364     bool M_stall = 0;
365     # Start injecting bubbles as soon as exception passes through memory
           stage
366     bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR,
           SINS, SHLT };
367
368     # Should I stall or inject a bubble into Pipeline Register W?
369     bool W_stall = W_stat in { SADR, SINS, SHLT };
370     bool W_bubble = 0;
371     /* $end pipe-all-hcl */

```

B pipe-btfnt.hcl

```
1  /* $begin pipe-all-hcl */
2  #####
3  #    HCL Description of Control for Pipelined Y86 Processor      #
4  #    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010  #
5  #####
6
7  ## Your task is to modify the design so that conditional branches
   are
8  ## predicted as being taken when backward and not-taken when forward
9  ## The code here is nearly identical to that for the normal pipeline
   .
10 ## Comments starting with keyword "BBTFNT" have been added at places
11 ## relevant to the exercise.
12
13 #####
14 #    C Include's.  Don't alter these                             #
15 #####
16
17 quote '#include <stdio.h>'
18 quote '#include "isa.h"'
19 quote '#include "pipeline.h"'
20 quote '#include "stages.h"'
21 quote '#include "sim.h"'
22 quote 'int sim_main(int argc, char *argv[]);'
23 quote 'int main(int argc, char *argv[]){return sim_main(argc,argv)
   ;}'
24
25 #####
26 #    Declarations.  Do not change/remove/delete any of these      #
27 #####
28
29 ##### Symbolic representation of Y86 Instruction Codes #####
30 intsig INOP      'I_NOP'
31 intsig IHALT     'I_HALT'
32 intsig IRRMOVL   'I_RRMOVL'
33 intsig IIRMOVL   'I_IRMOVL'
34 intsig IRMMOVL   'I_RMMOVL'
35 intsig IMRMOVL   'I_MRMOVL'
36 intsig IOPL      'I_ALU'
37 intsig IJXX      'I_JMP'
38 intsig ICALL     'I_CALL'
39 intsig IRET      'I_RET'
40 intsig IPUSHL    'I_PUSHL'
41 intsig IPOPL     'I_POPL'
```

```

42
43 ##### Symbolic representations of Y86 function codes #####
44 intsig FNONE 'F_NONE' # Default function code
45
46 ##### Symbolic representation of Y86 Registers referenced #####
47 intsig RESP 'REG_ESP' # Stack Pointer
48 intsig RNONE 'REG_NONE' # Special value indicating "no
    register"
49
50 ##### ALU Functions referenced explicitly #####
51 intsig ALUADD 'A_ADD' # ALU should add its arguments
52 ## BBTFT: For modified branch prediction, need to distinguish
53 ## conditional vs. unconditional branches
54 ##### Jump conditions referenced explicitly
55 intsig UNCOND 'C_YES' # Unconditional transfer
56
57 ##### Possible instruction status values #####
58 intsig SBUB 'STAT_BUB' # Bubble in stage
59 intsig SAOK 'STAT_AOK' # Normal execution
60 intsig SADR 'STAT_ADR' # Invalid memory address
61 intsig SINS 'STAT_INS' # Invalid instruction
62 intsig SHLT 'STAT_HLT' # Halt instruction encountered
63
64 ##### Signals that can be referenced by control logic #####
65
66 ##### Pipeline Register F #####
67
68 intsig F_predPC 'pc_curr->pc' # Predicted value of PC
69
70 ##### Intermediate Values in Fetch Stage #####
71
72 intsig imem_icode 'imem_icode' # icode field from instruction
    memory
73 intsig imem_ifun 'imem_ifun' # ifun field from instruction
    memory
74 intsig f_icode 'if_id_next->icode' # (Possibly modified)
    instruction code
75 intsig f_ifun 'if_id_next->ifun' # Fetched instruction function
76 intsig f_valC 'if_id_next->valc' # Constant data of fetched
    instruction
77 intsig f_valP 'if_id_next->valp' # Address of following
    instruction
78 boolsig imem_error 'imem_error' # Error signal from instruction
    memory
79 boolsig instr_valid 'instr_valid' # Is fetched instruction valid?
80

```

```

81 ##### Pipeline Register D #####
82 intsig D_icode 'if_id_curr->icode'      # Instruction code
83 intsig D_rA   'if_id_curr->ra'          # rA field from instruction
84 intsig D_rB   'if_id_curr->rb'          # rB field from instruction
85 intsig D_valP 'if_id_curr->valp'        # Incremented PC
86
87 ##### Intermediate Values in Decode Stage #####
88
89 intsig d_srcA   'id_ex_next->srca'      # srcA from decoded instruction
90 intsig d_srcB   'id_ex_next->srcb'      # srcB from decoded instruction
91 intsig d_rvalA  'd_regvala'            # valA read from register file
92 intsig d_rvalB  'd_regvalb'            # valB read from register file
93
94 ##### Pipeline Register E #####
95 intsig E_icode  'id_ex_curr->icode'     # Instruction code
96 intsig E_ifun   'id_ex_curr->ifun'      # Instruction function
97 intsig E_valC   'id_ex_curr->valc'      # Constant data
98 intsig E_srcA   'id_ex_curr->srca'      # Source A register ID
99 intsig E_valA   'id_ex_curr->vala'      # Source A value
100 intsig E_srcB   'id_ex_curr->srcb'      # Source B register ID
101 intsig E_valB   'id_ex_curr->valb'      # Source B value
102 intsig E_dstE   'id_ex_curr->deste'     # Destination E register ID
103 intsig E_dstM   'id_ex_curr->destm'     # Destination M register ID
104
105 ##### Intermediate Values in Execute Stage #####
106 intsig e_vale   'ex_mem_next->vale'     # valE generated by ALU
107 boolsig e_Cnd   'ex_mem_next->takebranch' # Does condition hold?
108 intsig e_dstE   'ex_mem_next->deste'     # dstE (possibly modified to
    be RNONE)
109
110 ##### Pipeline Register M #####
111 intsig M_stat   'ex_mem_curr->status'    # Instruction status
112 intsig M_icode  'ex_mem_curr->icode'     # Instruction code
113 intsig M_ifun   'ex_mem_curr->ifun'      # Instruction function
114 intsig M_valA   'ex_mem_curr->vala'      # Source A value
115 intsig M_dstE   'ex_mem_curr->deste'     # Destination E register ID
116 intsig M_vale   'ex_mem_curr->vale'      # ALU E value
117 intsig M_dstM   'ex_mem_curr->destm'     # Destination M register ID
118 boolsig M_Cnd   'ex_mem_curr->takebranch' # Condition flag
119 boolsig dmem_error 'dmem_error'        # Error signal from
    instruction memory
120
121 ##### Intermediate Values in Memory Stage #####
122 intsig m_valM   'mem_wb_next->valm'      # valM generated by memory
123 intsig m_stat   'mem_wb_next->status'    # stat (possibly modified to
    be SADR)

```

```

124
125 ##### Pipeline Register W #####
126 intsig W_stat 'mem_wb_curr->status'      # Instruction status
127 intsig W_icode 'mem_wb_curr->icode'      # Instruction code
128 intsig W_dstE 'mem_wb_curr->deste'      # Destination E register ID
129 intsig W_vale 'mem_wb_curr->vale'      # ALU E value
130 intsig W_dstM 'mem_wb_curr->destm'      # Destination M register ID
131 intsig W_valM 'mem_wb_curr->valm'      # Memory M value
132
133 #####
134 # Control Signal Definitions. #
135 #####
136
137 ##### Fetch Stage #####
138
139 ## What address should instruction be fetched at
140 int f_pc = [
141     # Mispredicted branch. Fetch at incremented PC
142     M_icode == IJXX && M_ifun != UNCOND &&
143     M_vale < M_valA && !M_Cnd : M_valA;
144     M_icode == IJXX && M_ifun != UNCOND &&
145     M_vale >= M_valA && M_Cnd : M_vale;
146     # Completion of RET instruction.
147     W_icode == IRET : W_valM;
148     # Default: Use predicted value of PC
149     1 : F_predPC;
150 ];
151
152 ## Determine icode of fetched instruction
153 int f_icode = [
154     imem_error : INOP;
155     1: imem_icode;
156 ];
157
158 # Determine ifun
159 int f_ifun = [
160     imem_error : FNONE;
161     1: imem_ifun;
162 ];
163
164 # Is instruction valid?
165 bool instr_valid = f_icode in
166     { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
167       IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
168
169 # Determine status code for fetched instruction

```



```

170 int f_stat = [
171     imem_error: SADR;
172     !instr_valid : SINS;
173     f_icode == IHALT : SHLT;
174     1 : SAOK;
175 ];
176
177 # Does fetched instruction require a regid byte?
178 bool need_regids =
179     f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
180                 IIRMOVL, IRMMOVL, IMRMOVL };
181
182 # Does fetched instruction require a constant word?
183 bool need_valC =
184     f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
185
186 # Predict next value of PC
187 int f_predPC = [
188     # BBTFNT: This is where you'll change the branch prediction
189     # rule
189     f_icode == ICALL ||
190     f_icode == IJXX &&
191     (f_ifun == UNCOND || f_valC < f_valP) : f_valC;
192     1 : f_valP;
193 ];
194
195 ##### Decode Stage #####
196
197
198 ## What register should be used as the A source?
199 int d_srcA = [
200     D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : D_rA;
201     D_icode in { IPOPL, IRET } : RESP;
202     1 : RNONE; # Don't need register
203 ];
204
205 ## What register should be used as the B source?
206 int d_srcB = [
207     D_icode in { IOPL, IRMMOVL, IMRMOVL } : D_rB;
208     D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
209     1 : RNONE; # Don't need register
210 ];
211
212 ## What register should be used as the E destination?
213 int d_dstE = [
214     D_icode in { IRRMOVL, IIRMOVL, IOPL } : D_rB;

```

```

215         D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
216         1 : RNONE; # Don't write any register
217 ];
218
219 ## What register should be used as the M destination?
220 int d_dstM = [
221     D_icode in { IMRMOVL, IPOPL } : D_rA;
222     1 : RNONE; # Don't write any register
223 ];
224
225 ## What should be the A value?
226 ## Forward into decode stage for valA
227 int d_valA = [
228     D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
229     d_srcA == e_dstE : e_valE; # Forward valE from execute
230     d_srcA == M_dstM : m_valM; # Forward valM from memory
231     d_srcA == M_dstE : M_valE; # Forward valE from memory
232     d_srcA == W_dstM : W_valM; # Forward valM from write back
233     d_srcA == W_dstE : W_valE; # Forward valE from write back
234     1 : d_rvalA; # Use value read from register file
235 ];
236
237 int d_valB = [
238     d_srcB == e_dstE : e_valE; # Forward valE from execute
239     d_srcB == M_dstM : m_valM; # Forward valM from memory
240     d_srcB == M_dstE : M_valE; # Forward valE from memory
241     d_srcB == W_dstM : W_valM; # Forward valM from write back
242     d_srcB == W_dstE : W_valE; # Forward valE from write back
243     1 : d_rvalB; # Use value read from register file
244 ];
245
246 ##### Execute Stage #####
247
248 # BBTFNT: When some branches are predicted as not-taken, you need
      some
249 # way to get valC into pipeline register M, so that
250 # you can correct for a mispredicted branch.
251
252 ## Select input A to ALU
253 int aluA = [
254     E_icode in { IRRMOVL, IOPL } : E_valA;
255     E_icode in { IIRMOVL, IRMMOVL, IMRMOVL } : E_valC;
256     E_icode == IJXX && E_ifun != UNCOND : E_valC;
257     E_icode in { ICALL, IPUSHL } : -4;
258     E_icode in { IRET, IPOPL } : 4;
259     # Other instructions don't need ALU

```

```

260 ];
261
262 ## Select input B to ALU
263 int aluB = [
264     E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
265                 IPUSHL, IRET, IPOPL } : E_valB;
266     E_icode in { IRRMOVL, IIRMOVL } : 0;
267     E_icode == IJXX && E_ifun != UNCOND : 0;
268     # Other instructions don't need ALU
269 ];
270
271 ## Set the ALU function
272 int alufun = [
273     E_icode == IOPL : E_ifun;
274     1 : ALUADD;
275 ];
276
277 ## Should the condition codes be updated?
278 bool set_cc = E_icode == IOPL &&
279     # State changes only during normal operation
280     !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS,
281         SHLT };
282
281
282 ## Generate valA in execute stage
283 int e_valA = E_valA;    # Pass valA through stage
284
285 ## Set dstE to RNONE in event of not-taken conditional move
286 int e_dstE = [
287     E_icode == IRRMOVL && !e_Cnd : RNONE;
288     1 : E_dstE;
289 ];
290
291 ##### Memory Stage #####
292
293 ## Select memory address
294 int mem_addr = [
295     M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : M_valE;
296     M_icode in { IPOPL, IRET } : M_valA;
297     # Other instructions don't need address
298 ];
299
300 ## Set read control signal
301 bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET };
302
303 ## Set write control signal
304 bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };

```

```

305
306 /* $begin pipe-m_stat-hcl */
307 ## Update the status
308 int m_stat = [
309     dmem_error : SADR;
310     1 : M_stat;
311 ];
312 /* $end pipe-m_stat-hcl */
313
314 ## Set E port register ID
315 int w_dstE = W_dstE;
316
317 ## Set E port value
318 int w_valE = W_valE;
319
320 ## Set M port register ID
321 int w_dstM = W_dstM;
322
323 ## Set M port value
324 int w_valM = W_valM;
325
326 ## Update processor status
327 int Stat = [
328     W_stat == SBUB : SAOK;
329     1 : W_stat;
330 ];
331
332 ##### Pipeline Register Control #####
333
334 # Should I stall or inject a bubble into Pipeline Register F?
335 # At most one of these can be true.
336 bool F_bubble = 0;
337 bool F_stall =
338     # Conditions for a load/use hazard
339     E_icode in { IMRMOVL, IPOPL } &&
340     E_dstM in { d_srcA, d_srcB } ||
341     # Stalling at fetch while ret passes through pipeline
342     IRET in { D_icode, E_icode, M_icode };
343
344 # Should I stall or inject a bubble into Pipeline Register D?
345 # At most one of these can be true.
346 bool D_stall =
347     # Conditions for a load/use hazard
348     E_icode in { IMRMOVL, IPOPL } &&
349     E_dstM in { d_srcA, d_srcB };
350

```

```

351 bool D_bubble =
352     # Mispredicted branch
353     (E_icode == IJXX && E_ifun != UNCOND &&
354     (E_valC < E_valA && !e_Cnd ||
355     E_valC >= E_valA && e_Cnd)) ||
356     # BBTFNT: This condition will change
357     # Stalling at fetch while ret passes through pipeline
358     # but not condition for a load/use hazard
359     !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA,
360         d_srcB }) &&
361     IRET in { D_icode, E_icode, M_icode };
362 # Should I stall or inject a bubble into Pipeline Register E?
363 # At most one of these can be true.
364 bool E_stall = 0;
365 bool E_bubble =
366     # Mispredicted branch
367     (E_icode == IJXX && E_ifun != UNCOND &&
368     (E_valC < E_valA && !e_Cnd ||
369     E_valC >= E_valA && e_Cnd)) ||
370     # BBTFNT: This condition will change
371     # Conditions for a load/use hazard
372     E_icode in { IMRMOVL, IPOPL } &&
373     E_dstM in { d_srcA, d_srcB };
374
375 # Should I stall or inject a bubble into Pipeline Register M?
376 # At most one of these can be true.
377 bool M_stall = 0;
378 # Start injecting bubbles as soon as exception passes through memory
379 # stage
380 bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR,
381     SINS, SHLT };
382
381 # Should I stall or inject a bubble into Pipeline Register W?
382 bool W_stall = W_stat in { SADR, SINS, SHLT };
383 bool W_bubble = 0;
384 /* $end pipe-all-hcl */

```