```
1  # Analysis order:
2  #
3  # utils.vhd
4  # logging.vhd
5  # AnsiEscape.vhd
6  # sh2_constants.vhd
7  # mau.vhd
8  # sh2_pmau.vhd
9  # memory_interface.vhd
10 # sh2_dmau.vhd
11 # sh2_alu.vhd
12 # sh2_control.vhd
13 # sh2_reg.vhd
14 # sh2_cpu.vhd
15 # memory.vhd
16 # sh2_cpu_tb.vhd
17 # reg.vhd
18 # alu.vhd
19
20 # Dependenices:
21
22 # sh2_pmau.o:  sh2_constants.o  mau.o
23 # memory_interface.o: utils.o logging.o
24 # sh2_dmau.o: mau.o sh2_constants.o
25 # sh2_control.o: memory_interface.o logging.o sh2_pmau.o sh2_dmau.o sh2_alu.o utils.o
26 # sh2_cpu.o: sh2_pmau.o memory_interface.o sh2_control.o logging.o sh2_constants.o sh2_dmau.o sh2_reg.o sh2_alu.o
27 # memory.o: logging.o utils.o
28 # sh2_cpu_tb.o: utils.o logging.o AnsiEscape.o sh2_cpu.o memory.o
29 # reg.o: logging.o
30
31 GHDL = ghdl
32
33 # Directory for object files.
34 WORKDIR = work/
35
36 WAVEFORM = sh2_cpu_tb.ghw
37
38 # BUILDFLAGS = --std=08 -Wuseless -Werror -Wruntime-error -Wnowrite -fsynopsys --workdir=$(WORKDIR)
39 BUILDFLAGS = --std=08 -Wuseless -Werror -Wruntime-error -Wnowrite -fsynopsys --workdir=$(WORKDIR)
40
41 RUNFLAGS = --ieee-asserts=disable --wave=$(WAVEFORM)
42
43 SOURCES = utils.vhd logging.vhd AnsiEscape.vhd sh2_constants.vhd mau.vhd sh2_pmau.vhd \
44       memory_interface.vhd sh2_dmau.vhd sh2_alu.vhd sh2_control.vhd sh2_reg.vhd \
45       sh2_cpu.vhd memory.vhd sh2_cpu_tb.vhd reg.vhd alu.vhd
46
47 OBJECTS = $(patsubst %.vhd,$(WORKDIR)%.o,$(SOURCES))
48
49 # The top level entity
50 TOPLEVEL = sh2_cpu_tb
51
52 # Note that $@ specifies the first target, which in this case is the top level entity
53 all: $(TOPLEVEL)
54
55 $(TOPLEVEL): $(OBJECTS)
56   $(GHDL) -e $(BUILDFLAGS) $@
57
58 # Dependencies:
59 $(WORKDIR)sh2_pmau.o: $(WORKDIR)sh2_constants.o  $(WORKDIR)mau.o
60
61 $(WORKDIR)memory_interface.o: $(WORKDIR)utils.o $(WORKDIR)logging.o
62
63 $(WORKDIR)sh2_dmau.o: $(WORKDIR)mau.o $(WORKDIR)sh2_constants.o
64
65 $(WORKDIR)sh2_control.o: $(WORKDIR)memory_interface.o $(WORKDIR)logging.o  $(WORKDIR)sh2_pmau.o \
66              $(WORKDIR)sh2_dmau.o $(WORKDIR)sh2_alu.o  $(WORKDIR)utils.o
```

```
67
68 $(WORKDIR)sh2_cpu.o: $(WORKDIR)sh2_pmau.o $(WORKDIR)memory_interface.o $(WORKDIR)sh2_control.o \
69            $(WORKDIR)logging.o $(WORKDIR)sh2_constants.o $(WORKDIR)sh2_dmau.o $(WORKDIR)sh2_reg.o $(WORKDIR)sh2_alu.o
70
71 $(WORKDIR)memory.o:  $(WORKDIR)logging.o $(WORKDIR)utils.o
72
73 $(WORKDIR)sh2_cpu_tb.o: $(WORKDIR)utils.o  $(WORKDIR)logging.o $(WORKDIR)AnsiEscape.o \
74             $(WORKDIR)sh2_cpu.o $(WORKDIR)memory.o
75
76 $(WORKDIR)reg.o: $(WORKDIR)logging.o
77
78 # Note that $> is the first prereq (ie, the source file).
79 $(OBJECTS): $(WORKDIR)%.o: %.vhd
80   $(GHDL) -a $(BUILDFLAGS) $<
81
82 asm:
83   cd asm && $(MAKE)
84
85 test: $(TOPLEVEL) asm
86   ghdl -r $(TOPLEVEL) $(RUNFLAGS)
87
88 clean:
89   ghdl --clean --workdir=$(WORKDIR) --std=08
90   rm -rf asm/*.bin
91 # rm -rf *.cf *.o $(WORKDIR)
92
93 view:
94   gtkwave $(WAVEFORM) &
95
96 .PHONY: test clean build view asm run
97
```

```
1   ----------------------------------------------------------------------------
2   --
3   --  Generic ALU and Status Register
4   --
5   --  This is a generic implementation of the ALU for simple microprocessors.
6   --  It does not include a multiplier, MAC, divider, or barrel shifter.
7   --
8   --  Packages included are:
9   --     ALUConstants - constants for all entities making up the ALU
10  --
11  --  Entities included are:
12  --     FBlockBit - one bit of an F-Block
13  --     AdderBit  - one bit of an adder (a full adder)
14  --     FBlock    - F-Block (logical operations)
15  --     Adder     - adder
16  --     Shifter   - shift and rotate and swap operations
17  --     ALU       - the actual ALU
18  --
19  --  Revision History:
20  --     25 Jan 21  Glen George       Initial revision.
21  --     27 Jan 21  Glen George       Changed left/right shift selection to a
22  --                                  constant.
23  --     27 Jan 21  Glen George       Changed F-Block to be on B input of adder.
24  --     27 Jan 21  Glen George       Updated comments.
25  --     29 Jan 21  Glen George       Fixed a number of wordsize bugs.
26  --     29 Jan 21  Glen George       Fixed overflow signal in adder.
27  --     11 Apr 25  Glen George       Removed Status Register.
28  --     11 May 25  Zack Huang        Removed Swap
29  --
30  ----------------------------------------------------------------------------
31
32
33  --
34  --  Package containing the constants used by the ALU and all of its
35  --  sub-modules.
36  --
37
38  library ieee;
39  use ieee.std_logic_1164.all;
40
41  package  ALUConstants  is
42
43  --  Adder carry in select constants
44  --     may be freely changed
45
46     constant CinCmd_ZERO   : std_logic_vector(1 downto 0) := "00";
47     constant CinCmd_ONE    : std_logic_vector(1 downto 0) := "01";
48     constant CinCmd_CIN    : std_logic_vector(1 downto 0) := "10";
49     constant CinCmd_CINBAR : std_logic_vector(1 downto 0) := "11";
50
51
52  --  Shifter command constants
53  --     may be freely changed except a single bit pattern (currently high bit)
54  --     must distinguish between left shifts and rotates and right shifts and
55  --     rotates
56
57     constant SCmd_LEFT  : std_logic_vector(2 downto 0) := "0--";
58     constant SCmd_LSL   : std_logic_vector(2 downto 0) := "000";
59     constant SCmd_ASL   : std_logic_vector(2 downto 0) := "001";
60     constant SCmd_ROL   : std_logic_vector(2 downto 0) := "010";
61     constant SCmd_RLC   : std_logic_vector(2 downto 0) := "011";
62     constant SCmd_RIGHT : std_logic_vector(2 downto 0) := "1--";
63     constant SCmd_LSR   : std_logic_vector(2 downto 0) := "100";
64     constant SCmd_ASR   : std_logic_vector(2 downto 0) := "101";
65     constant SCmd_ROR   : std_logic_vector(2 downto 0) := "110";
66     constant SCmd_RRC   : std_logic_vector(2 downto 0) := "111";
```

```
67
68
69  --  ALU command constants
70  --     may be freely changed
71
72     constant ALUCmd_FBLOCK  : std_logic_vector(1 downto 0) := "00";
73     constant ALUCmd_ADDER   : std_logic_vector(1 downto 0) := "01";
74     constant ALUCmd_SHIFT   : std_logic_vector(1 downto 0) := "10";
75
76
77  end package;
78
79
80
81  --
82  --  FBlockBit
83  --
84  --  This is a bit of the F-Block for doing logical operations in the ALU.  The
85  --  operations available are:
86  --     FCmd    Operation
87  --     0000    0
88  --     0001    A nor B
89  --     0010    not A and B
90  --     0011    not A
91  --     0100    A and not B
92  --     0101    not B
93  --     0110    A xor B
94  --     0111    A nand B
95  --     1000    A and B
96  --     1001    A xnor B
97  --     1010    B
98  --     1011    not A or B
99  --     1100    A
100 --     1101    A or not B
101 --     1110    A or B
102 --     1111    1
103 --
104 --  Inputs:
105 --    A    - first operand bit (bus A)
106 --    B    - second operand bit (bus B)
107 --    FCmd - operation to perform (4 bits)
108 --
109 --  Outputs:
110 --    F    - F-Block output (based on input busses and command)
111 --
112
113 library ieee;
114 use ieee.std_logic_1164.all;
115
116 entity  FBlockBit  is
117
118     port(
119         A    : in   std_logic;                      -- first operand
120         B    : in   std_logic;                      -- second operand
121         FCmd : in   std_logic_vector(3 downto 0);   -- operation to perform
122         F    : out  std_logic                       -- result
123     );
124
125 end  FBlockBit;
126
127
128 architecture  dataflow  of  FBlockBit  is
129 begin
130
131     F  <=  FCmd(3)  when  ((A = '1') and (B = '1'))  else
132            FCmd(2)  when  ((A = '1') and (B = '0'))  else
```

```
133             FCmd(1)  when  ((A = '0') and (B = '1'))  else
134             FCmd(0)  when  ((A = '0') and (B = '0'))  else
135             'X';
136
137 end  dataflow;
138
139
140
141 --
142 --  AdderBit
143 --
144 --  This is a bit of the adder for doing addition in the ALU.
145 --
146 --  Inputs:
147 --    A  - first operand bit (bus A)
148 --    B  - second operand bit (bus B)
149 --    Ci - carry in (from previous bit)
150 --
151 --  Outputs:
152 --    S  - sum for this bit
153 --    Co - carry out for this bit
154 --
155
156 library ieee;
157 use ieee.std_logic_1164.all;
158
159 entity  AdderBit  is
160
161     port(
162         A  : in   std_logic;        -- first operand
163         B  : in   std_logic;        -- second operand
164         Ci : in   std_logic;        -- carry in from previous bit
165         S  : out  std_logic;        -- sum (result)
166         Co : out  std_logic         -- carry out to next bit
167     );
168
169 end  AdderBit;
170
171
172 architecture  dataflow  of  AdderBit is
173 begin
174
175     S  <=  A  xor  B  xor  Ci;
176     Co <=  (A  and  B)  or  (A  and Ci)  or  (B  and  Ci);
177
178 end  dataflow;
179
180
181
182 --
183 --  FBlock
184 --
185 --  This is the F-Block for doing logical operations in the ALU.  The F-Block
186 --  operations are:
187 --     FCmd    Operation
188 --     0000    0
189 --     0001    FBOpA nor FBOpB
190 --     0010    not FBOpA and FBOpB
191 --     0011    not FBOpA
192 --     0100    FBOpA and not FBOpB
193 --     0101    not FBOpB
194 --     0110    FBOpA xor FBOpB
195 --     0111    FBOpA nand FBOpB
196 --     1000    FBOpA and FBOpB
197 --     1001    FBOpA xnor FBOpB
198 --     1010    FBOpB
```

```
199 --     1011    not FBOpA or FBOpB
200 --     1100    FBOpA
201 --     1101    FBOpA or not FBOpB
202 --     1110    FBOpA or FBOpB
203 --     1111    1
204 --
205 --  Generics:
206 --    wordsize - width of the F-Block in bits (default 8)
207 --
208 --  Inputs:
209 --    FBOpA   - first operand
210 --    FBOpB   - second operand
211 --    FCmd    - operation to perform (4 bits)
212 --
213 --  Outputs:
214 --    FResult - F-Block result (based on input busses and command)
215 --
216
217 library ieee;
218 use ieee.std_logic_1164.all;
219
220 entity  FBlock  is
221
222     generic (
223         wordsize : integer := 8      -- default width is 8-bits
224     );
225
226     port(
227         FBOpA   : in   std_logic_vector(wordsize - 1 downto 0); -- first operand
228         FBOpB   : in   std_logic_vector(wordsize - 1 downto 0); -- second operand
229         FCmd    : in   std_logic_vector(3 downto 0);            -- operation to perform
230         FResult : out  std_logic_vector(wordsize - 1 downto 0)  -- result
231     );
232
233 end  FBlock;
234
235
236
237 architecture  structural  of  FBlock  is
238
239     component  FBlockBit
240         port(
241             A    : in   std_logic;                     -- first operand
242             B    : in   std_logic;                     -- second operand
243             FCmd : in   std_logic_vector(3 downto 0);   -- operation to perform
244             F    : out  std_logic                      -- result
245         );
246     end component;
247
248 begin
249
250     F1:  for  i  in  FResult'Range  generate        -- make enough FBlockBits
251     begin
252         FBx: FBlockBit  port map  (FBOpA(i), FBOpB(i), FCmd, FResult(i));
253     end generate;
254
255 end  structural;
256
257
258
259 --
260 --  Adder
261 --
262 --  This is the adder for doing addition in the ALU.
263 --
264 --  Generics:
```

```
265 --    wordsize - width of the adder in bits (default 8)
266 --
267 --  Inputs:
268 --    AddOpA - first operand
269 --    AddOpB - second operand
270 --    Cin    - carry in (from status register)
271 --    CinCmd - operation for carry in (2 bits)
272 --
273 --  Outputs:
274 --    AddResult - sum
275 --    Cout      - carry out for the addition
276 --    HalfCOut  - half carry out for the addition
277 --    Overflow  - signed overflow
278 --
279
280 library ieee;
281 use ieee.std_logic_1164.all;
282 use work.ALUConstants.all;
283
284 entity  Adder  is
285
286     generic (
287         wordsize : integer := 8      -- default width is 8-bits
288     );
289
290     port(
291         AddOpA    : in   std_logic_vector(wordsize - 1 downto 0);   -- first operand
292         AddOpB    : in   std_logic_vector(wordsize - 1 downto 0);   -- second operand
293         Cin       : in   std_logic;                                 -- carry in
294         CinCmd    : in   std_logic_vector(1 downto 0);              -- carry in operation
295         AddResult : out  std_logic_vector(wordsize - 1 downto 0);   -- sum (result)
296         Cout      : out  std_logic;                                 -- carry out
297         HalfCout  : out  std_logic;                                 -- half carry out
298         Overflow  : out  std_logic                                  -- signed overflow
299     );
300
301 end  Adder;
302
303
304 architecture  structural  of  Adder  is
305
306     component  AdderBit
307         port(
308             A  : in   std_logic;        -- first operand
309             B  : in   std_logic;        -- second operand
310             Ci : in   std_logic;        -- carry in from previous bit
311             S  : out  std_logic;        -- sum (result)
312             Co : out  std_logic         -- carry out to next bit
313         );
314     end component;
315
316     signal  carry : std_logic_vector(wordsize downto 0);        -- intermediate carry results
317
318 begin
319
320     -- get the carry in based on CinCmd
321     carry(0)  <=  '0'      when  CinCmd = CinCmd_ZERO  else
322                   '1'      when  CinCmd = CinCmd_ONE  else
323                   Cin      when  CinCmd = CinCmd_CIN  else
324                   not Cin  when  CinCmd = CinCmd_CINBAR  else
325                   'X';
326
327     A1:  for  i  in  AddResult'Range  generate      -- make enough AdderBits
328     begin
329         ABx: AdderBit  port map  (AddOpA(i), AddOpB(i), carry(i),
330                                   AddResult(i), carry(i + 1));
```

```
331     end generate;
332
333     Cout     <= carry(wordsize);        -- compute carry out
334     HalfCout <= carry(4);               -- half carry (carry into high nibble)
335     -- overflow if carry into sign bit doesn't match the carry out
336     Overflow <= carry(wordsize - 1) xor carry(wordsize);
337
338 end  structural;
339
340
341
342 --
343 --  Shifter
344 --
345 --  This is the shifter for doing shift/rotate operations in the ALU.  The
346 --  shift operations are defined by the constants SCmd_LEFT, SCmd_RIGHT,
347 --  SCmd_LSL, SCmd_ROL, SCmd_RLC, SCmd_LSR, SCmd_ASR, SCmd_ROR, and SCmd_RRC.
348 --
349 --  Generics:
350 --    wordsize - width of the shifter in bits (default 8)
351 --               must be an even number of bits
352 --
353 --  Inputs:
354 --    SOp     - operand
355 --    Cin     - carry in (from status register)
356 --    SCmd    - operation to perform (3 bits)
357 --
358 --  Outputs:
359 --    SResult - shift result
360 --    Cout    - carry out from the shift (link)
361 --
362
363 library ieee;
364 use ieee.std_logic_1164.all;
365 use ieee.numeric_std.std_match;
366 use work.ALUConstants.all;
367
368 entity  Shifter  is
369
370     generic (
371         wordsize : integer := 8      -- default width is 8-bits
372     );
373
374     port(
375         SOp     : in   std_logic_vector(wordsize - 1 downto 0); -- operand
376         Cin     : in   std_logic;                               -- carry in
377         SCmd    : in   std_logic_vector(2 downto 0);            -- shift operation
378         SResult : out  std_logic_vector(wordsize - 1 downto 0); -- sum (result)
379         Cout    : out  std_logic                                -- carry out
380     );
381
382 end  Shifter;
383
384
385 architecture  dataflow  of  Shifter  is
386 begin
387
388     -- middle bits get either bit to the left or bit to the right
389     SResult(wordsize - 2 downto 1)  <=
390
391         -- right shift
392         SOp(wordsize - 1 downto 2)                                          when  std_match(SCmd, SCmd_RIGHT)  else
393
394         -- left shift
395         SOp(wordsize - 3 downto 0)                                          when  std_match(SCmd, SCmd_LEFT)  else
396
```

```
397         -- unknown command
398         (others => 'X');
399
400
401     -- high bit gets low bit, high bit, bit to the right, 0, or Cin depending
402     --   on shift mode
403     SResult(wordsize - 1)  <=
404         SOp(wordsize - 2)    when  std_match(SCmd, SCmd_LEFT)  else  -- shift/rotate left
405         SOp(0)               when  SCmd = SCmd_ROR   else  -- rotate right
406         SOp(wordsize - 1)    when  SCmd = SCmd_ASR   else  -- arithmetic shift right
407         '0'                  when  SCmd = SCmd_LSR   else  -- logical shift right
408         Cin                  when  SCmd = SCmd_RRC   else  -- rotate right w/carry
409         'X';                                               -- anything else is illegal
410
411
412     -- low bit gets high bit, bit to the left, 0, or Cin depending on mode
413     SResult(0)  <=
414         SOp(1)            when  std_match(SCmd, SCmd_RIGHT)  else  -- shift/rotate right
415         '0'               when  SCmd = SCmd_LSL or SCmd = SCmd_ASL   else  -- shift left
416         SOp(wordsize - 1) when  SCmd = SCmd_ROL   else  -- rotate left
417         Cin               when  SCmd = SCmd_RLC   else  -- rotate left w/carry
418         'X';                                             -- anything else is illegal
419
420
421     -- compute the carry out, it is low bit when shifting right and high bit
422     --    when shifting left
423     Cout  <=  SOp(0)             when  std_match(SCmd, SCmd_RIGHT)  else
424               SOp(wordsize - 1)  when  std_match(SCmd, SCmd_LEFT)   else
425               'X';
426
427 end  dataflow;
428
429
430
431 --
432 --  ALU
433 --
434 --  This is the actual ALU model for the CPU.  It includes the FBlock,
435 --  Adder, and Shifter modules.  It also outputs a number of status bits.
436 --
437 --  Generics:
438 --    wordsize - width of the ALU in bits (default 8)
439 --
440 --  Inputs:
441 --    ALUOpA   - first operand
442 --    ALUOpB   - second operand
443 --    Cin      - carry in (from status register)
444 --    FCmd     - F-Block operation to perform (4 bits)
445 --    CinCmd   - adder carry in operation for carry in (2 bits)
446 --    SCmd     - shift operation to perform (3 bits)
447 --    ALUCmd   - ALU operation to perform - selects result (2 bits)
448 --
449 --  Outputs:
450 --    Result   - ALU result
451 --    Cout     - carry out from the operation
452 --    HalfCOut - half carry out for addition
453 --    Overflow - signed overflow for addition
454 --    Zero     - zero result
455 --    Sign     - result sign (1 negative, 0 positive)
456 --
457
458 library ieee;
459 use ieee.std_logic_1164.all;
460 use work.ALUConstants.all;
461
462 entity  ALU  is
```

```
463
464     generic (
465         wordsize : integer := 8      -- default width is 8-bits
466     );
467
468     port(
469         ALUOpA   : in      std_logic_vector(wordsize - 1 downto 0);   -- first operand
470         ALUOpB   : in      std_logic_vector(wordsize - 1 downto 0);   -- second operand
471         Cin      : in      std_logic;                                 -- carry in
472         FCmd     : in      std_logic_vector(3 downto 0);              -- F-Block operation
473         CinCmd   : in      std_logic_vector(1 downto 0);              -- carry in operation
474         SCmd     : in      std_logic_vector(2 downto 0);              -- shift operation
475         ALUCmd   : in      std_logic_vector(1 downto 0);              -- ALU result select
476         Result   : buffer  std_logic_vector(wordsize - 1 downto 0);   -- ALU result
477         Cout     : out     std_logic;                                 -- carry out
478         HalfCout : out     std_logic;                                 -- half carry out
479         Overflow : out     std_logic;                                 -- signed overflow
480         Zero     : out     std_logic;                                 -- result is zero
481         Sign     : out     std_logic                                  -- sign of result
482     );
483
484 end  ALU;
485
486
487 architecture  structural  of  ALU  is
488
489     component  FBlock
490         generic(
491             wordsize : integer
492         );
493         port(
494             FBOpA   : in   std_logic_vector(wordsize - 1 downto 0);
495             FBOpB   : in   std_logic_vector(wordsize - 1 downto 0);
496             FCmd    : in   std_logic_vector(3 downto 0);
497             FResult : out  std_logic_vector(wordsize - 1 downto 0)
498         );
499     end component;
500
501     component  Adder
502         generic(
503             wordsize : integer
504         );
505         port(
506             AddOpA    : in   std_logic_vector(wordsize - 1 downto 0);
507             AddOpB    : in   std_logic_vector(wordsize - 1 downto 0);
508             Cin       : in   std_logic;
509             CinCmd    : in   std_logic_vector(1 downto 0);
510             AddResult : out  std_logic_vector(wordsize - 1 downto 0);
511             Cout      : out  std_logic;
512             HalfCout  : out  std_logic;
513             Overflow  : out  std_logic
514         );
515     end component;
516
517     component  Shifter
518         generic(
519             wordsize : integer
520         );
521         port(
522             SOp     : in   std_logic_vector(wordsize - 1 downto 0);
523             Cin     : in   std_logic;
524             SCmd    : in   std_logic_vector(2 downto 0);
525             SResult : out  std_logic_vector(wordsize - 1 downto 0);
526             Cout    : out  std_logic
527         );
528     end component;
```

```
529
530     signal  FBRes   : std_logic_vector(wordsize - 1 downto 0);  -- F-Block result
531     signal  AddRes  : std_logic_vector(wordsize - 1 downto 0);  -- adder result
532     signal  ShRes   : std_logic_vector(wordsize - 1 downto 0);  -- shifter result
533
534     signal  AddCout : std_logic;                                -- adder carry out
535     signal  ShCout  : std_logic;                                -- shifter carry out
536
537 begin
538
539     -- wire up the blocks
540     FB1:   FBlock   generic map (wordsize)
541                     port map  (ALUOpA, ALUOpB, FCmd, FBRes);
542     Add1:  Adder    generic map (wordsize)
543                     port map  (ALUOpA, FBRes, Cin, CinCmd,
544                                AddRes, AddCout, HalfCout, Overflow);
545     Sh1:   Shifter  generic map (wordsize)
546                     port map  (ALUOpA, Cin, SCmd, ShRes, ShCout);
547
548     -- figure out the result
549     Result  <=  FBRes   when  ALUCmd = ALUCmd_FBLOCK  else  -- want F-Block
550                 AddRes  when  ALUCmd = ALUCmd_ADDER   else  -- want adder
551                 ShRes   when  ALUCmd = ALUCmd_SHIFT   else  -- want shifter
552                 (others => 'X');                            -- unknown command
553
554     -- figure out the carry out
555     COut  <=  '0'      when  ALUCmd = ALUCmd_FBLOCK  else   -- want F-Block
556               AddCout  when  ALUCmd = ALUCmd_ADDER   else   -- want adder
557               ShCout   when  ALUCmd = ALUCmd_SHIFT   else   -- want shifter
558               'X';                                          -- unknown command
559
560     -- zero flag is set when the result is 0
561     Zero  <=  '1'  when  Result = (Result'range => '0')  else
562               '0';
563
564     -- compute the sign flag value
565     Sign  <=  Result(wordsize - 1);
566
567 end  structural;
568
```

```
1   --------------------------------------------------------------------------------
2   --
3   --  Generic Memory Access Unit
4   --
5   --  This is an implementation of a generic memory access unit for
6   --  microprocessors.  This unit generates the memory address for either load
7   --  and store operations or instruction access.  It is parameterized by the
8   --  number of sources and the data width.
9   --
10  --  Packages included are:
11  --     MemUnitConstants - constants for the memory access unit
12  --     array_type_pkg   - type definition for 2-D arrays of std_logic
13  --
14  --  Entities included are:
15  --     AdderBit - one bit of an adder (a full adder)
16  --     MemUnit  - generic memory access unit
17  --
18  --  Revision History:
19  --     27 Jan 21  Glen George       Initial revision.
20  --      4 Feb 21  Glen George       Added missing library declaration.
21  --      4 Feb 21  Glen George       Added initialization of low bit of carry
22  --                                  for the adder.
23  --
24  --------------------------------------------------------------------------------
25
26
27  --
28  --  Package containing the type definition for 2-D arrays of std_logic.  The
29  --  type is an unconstrained 2-D array which is supported in VHDL-2008.
30  --
31
32  library ieee;
33  use ieee.std_logic_1164.all;
34
35  package array_type_pkg is
36
37  --  a 2D array of std_logic (VHDL-2008)
38     type  std_logic_array  is  array (natural range<>) of std_logic_vector;
39
40
41  end package;
42
43
44
45  --
46  --  Package containing the constants for the Memory Unit
47  --
48
49  library ieee;
50  use ieee.std_logic_1164.all;
51
52  package MemUnitConstants is
53
54  --  memory access unit constants for pre- and post- increment and decrement
55  --     these constants may be freely changed
56
57     constant MemUnit_PRE  : std_logic := '0';            -- pre- inc/dec
58     constant MemUnit_POST : std_logic := '1';            -- post- inc/dec
59     constant MemUnit_INC  : std_logic := '0';            -- pre/post increment
60     constant MemUnit_DEC  : std_logic := '1';            -- pre/post decrement
61
62
63  end package;
64
65
66
```

```
67  --
68  --  AdderBit
69  --
70  --  This is a bit of the adder for doing addition in the ALU.
71  --
72  --  Inputs:
73  --    A  - first operand bit (bus A)
74  --    B  - second operand bit (bus B)
75  --    Ci - carry in (from previous bit)
76  --
77  --  Outputs:
78  --    S  - sum for this bit
79  --    Co - carry out for this bit
80  --
81  --
82  --  NOTE: entity already defined in alu.vhd
83
84  -- library ieee;
85  -- use ieee.std_logic_1164.all;
86  --
87  -- entity  AdderBit  is
88  --
89  --     port(
90  --         A  : in   std_logic;        -- first operand
91  --         B  : in   std_logic;        -- second operand
92  --         Ci : in   std_logic;        -- carry in from previous bit
93  --         S  : out  std_logic;        -- sum (result)
94  --         Co : out  std_logic         -- carry out to next bit
95  --     );
96  --
97  -- end  AdderBit;
98  --
99  --
100 -- architecture  dataflow  of  AdderBit  is
101 -- begin
102 --
103 --     S  <=  A  xor  B  xor  Ci;
104 --     Co <=  (A  and  B)  or  (A  and Ci)  or  (B  and  Ci);
105 --
106 -- end  dataflow;
107
108
109
110 --
111 --  MemUnit
112 --
113 --  This is a generic memory access unit.  It allows for pre- or post-
114 --  increment/decrement of an address as well as multiple sources for the
115 --  address and offset.
116 --
117 --  Generics:
118 --    srcCnt       - number of possible sources
119 --    offsetCnt    - number of possible address offsets
120 --    maxIncDecBit - maximum value for IncDecBit input (for optimization)
121 --    wordsize     - address width
122 --
123 --  Inputs:
124 --    AddrSrc    - array (srccnt x wordsize) of address sources
125 --    SrcSel     - source to use (log srccnt bits)
126 --    AddrOff    - array (offsetcnt x wordsize) of address offsets
127 --    OffsetSel  - offset to use (log offsetcnt bits)
128 --    IncDecSel  - whether to increment (0) or decrement (1) address source
129 --    IncDecBit  - bit of address source to increment/decrement
130 --    PrePostSel - whether to pre- (0) or post- (1) inc/dec address source
131 --
132 --  Outputs:
```

```
133 --    Address    - address bus (wordsize bits)
134 --    AddrSrcOut - incremented/decremented source (wordsize bits)
135 --
136
137 library ieee;
138 use ieee.std_logic_1164.all;
139 use work.array_type_pkg.all;
140 use work.MemUnitConstants.all;
141
142 entity  MemUnit  is
143
144     generic (
145         srcCnt       : integer;
146         offsetCnt    : integer;
147         maxIncDecBit : integer := 0; -- default is only inc/dec bit 0
148         wordsize     : integer := 16 -- default address width is 16 bits
149     );
150
151     port(
152         AddrSrc    : in      std_logic_array(srccnt - 1 downto 0)(wordsize - 1 downto 0);
153         SrcSel     : in      integer  range srccnt - 1 downto 0;
154         AddrOff    : in      std_logic_array(offsetcnt - 1 downto 0)(wordsize - 1 downto 0);
155         OffsetSel  : in      integer  range offsetcnt - 1 downto 0;
156         IncDecSel  : in      std_logic;
157         IncDecBit  : in      integer  range maxIncDecBit downto 0;
158         PrePostSel : in      std_logic;
159         Address    : out     std_logic_vector(wordsize - 1 downto 0);
160         AddrSrcOut : buffer  std_logic_vector(wordsize - 1 downto 0)
161     );
162
163 end  MemUnit;
164
165
166 architecture  dataflow  of  MemUnit  is
167
168     -- need adders for computing the address
169     component  AdderBit  port(
170         A  : in   std_logic;        -- first operand
171         B  : in   std_logic;        -- second operand
172         Ci : in   std_logic;        -- carry in from previous bit
173         S  : out  std_logic;        -- sum (result)
174         Co : out  std_logic         -- carry out to next bit
175     );
176     end component;
177
178     -- intermediate carry results
179     --   for adder
180     signal  acarry : std_logic_vector(wordsize downto 0);
181     --   for incrementer/decrementer
182     signal  idcarry : std_logic_vector(wordsize downto 0);
183
184     -- source address, depends on whether doing pre- or post- inc/dec
185     signal  SrcAddr : std_logic_vector(wordsize - 1 downto 0);
186
187     -- input to incrementer/decrementer adder, depends on whether doing an
188     --    increment or decrement and which bit it is being applied to
189     signal  IncDecIn : std_logic_vector(wordsize - 1 downto 0);
190
191     -- incremented/decremented source address
192     signal  OutSrcAddr : std_logic_vector(wordsize - 1 downto 0);
193
194
195     signal SrcSelMux : std_logic_vector(wordsize - 1 downto 0);
196
197     signal AddrOffMux : std_logic_vector(wordsize - 1 downto 0);
198
```

```
199 begin
200
201     -- compute the input for the incrementer/decrementer
202     --    when incrementing the input is all 0's except for the IncDecBit (bit
203     --       to start incrementing)
204     --    when decrementing the input is 0's until the IncDecBit (the bit to
205     --       start decrementing) and after that it is all 1's
206     --    thus the IncDecBit is always a 1
207     IDin:  for  i  in  IncDecIn'Range  generate     -- generate the bits independently
208     begin
209
210         -- if past the maximum allowable bit to increment/decrement use 0 when
211         --    incrementing and 1 when decrementing (optimization)
212         IDin1:  if  (i > maxIncDecBit)  generate
213             IncDecIn(i)  <=  '0'  when  IncDecSel = MemUnit_INC  else
214                              '1'  when  IncDecSel = MemUnit_DEC  else
215                              'X';
216         end generate;
217
218         -- if not past the maximum allowable bit to increment/decrement use a
219         --    0 if below the IncDecBit and a 1 if at the IncDecBit and a 0 if
220         --    above the IncDecBit and incrementing and a 1 if above the
221         --    IncDecBit and decrementing
222         IDin2:  if  (i <= maxIncDecBit)  generate
223             IncDecIn(i) <= '1'  when  (i = IncDecBit) or
224                                       ((IncDecSel = MemUnit_DEC) and
225                                        (i >= IncDecBit))  else
226                            '0';
227         end generate;
228
229     end generate;
230
231
232     -- adder for doing increment/decrement
233     --    it adds the increment/decrement input to the selected source address
234     --    to generate the source address output (AddrSrcOut)
235     idcarry(0) <= '0';                              -- there is no carry in
236
237
238     SrcSelMux <= AddrSrc(SrcSel);
239
240     IDA1:  for  i  in  AddrSrcOut'Range  generate   -- make enough AdderBits
241     begin
242         IDABx: AdderBit  port map  (
243           SrcSelMux(i),
244           IncDecIn(i),
245           idcarry(i),
246           OutSrcAddr(i),
247           idcarry(i + 1));
248     end generate;
249
250
251     -- input to the offset adder is either the original source or the
252     --    incremented/decremented source, depending on whether doing pre- or
253     --    post- increment/decrement
254     SrcAddr  <=  AddrSrc(SrcSel)  when  PrePostSel = MemUnit_POST else
255                  OutSrcAddr       when  PrePostSel = MemUnit_PRE  else
256                  (others => 'X');
257
258
259     -- adder for adding offset to the source address
260     acarry(0) <= '0';                               -- there is no carry in
261
262     AddrOffMux <= AddrOff(OffsetSel);
263
264     AA1:  for  i  in  Address'Range  generate       -- make enough AdderBits
```

```
265     begin
266         AABx: AdderBit  port map  (
267           SrcAddr(i),
268           AddrOffMux(i),
269           acarry(i),
270           Address(i),
271           acarry(i + 1));
272     end generate;
273 
274 
275     -- output the incremented/decremented source address
276     AddrSrcOut  <=  OutSrcAddr;
277 
278 end  dataflow;
279 
280 
281 
282 
```

```
1   ----------------------------------------------------------------------------
2   --
3   --  Generic Register Array
4   --
5   --  This is an implementation of a Register Array for the register-based
6   --  microprocessors.  It allows the registers to be accessed as single words
7   --  or double words.  Multiple interfaces to the registers are allowed so they
8   --  may be simultaneously used as ALU registers and address registers.  Double
9   --  word access may be used for addressing (typically used in 8-bit
10  --  processors) for example.
11  --
12  --  Entities included are:
13  --     RegArray  - the register array
14  --
15  --  Revision History:
16  --     25 Jan 21  Glen George       Initial revision.
17  --     11 Apr 25  Glen George       Added separate address register interface.
18  --
19  ----------------------------------------------------------------------------
20
21
22  --
23  --  RegArray
24  --
25  --  This is a generic register array.  It contains regcnt wordsize bit
26  --  registers along with the appropriate reading and writing controls.  The
27  --  registers can also be read and written as double width registers.  There
28  --  is also two separate access ports and a write port to allow the registers
29  --  to be used as address registers simultaneous to their use in other blocks
30  --  such as the ALU.
31  --
32  --  Generics:
33  --    regcnt   - number of registers in the array (must be a multiple of 2)
34  --    wordsize - width of each register
35  --
36  --  Inputs:
37  --    RegIn      - input bus to the registers
38  --    RegInSel   - which register to write (log regcnt bits)
39  --    RegStore   - actually write to a register
40  --    RegASel    - register to read onto bus A (log regcnt bits)
41  --    RegBSel    - register to read onto bus B (log regcnt bits)
42  --    RegAxIn    - input bus for address register updates
43  --    RegAxInSel - which address register to write (log regcnt bits - 1)
44  --    RegAxStore - actually write to an address register
45  --    RegA1Sel   - register to read onto address bus 1 (log regcnt bits)
46  --    RegA2Sel   - register to read onto address bus 2 (log regcnt bits)
47  --    RegDIn     - input bus to the double-width registers
48  --    RegDInSel  - which double register to write (log regcnt bits - 1)
49  --    RegDStore  - actually write to a double register
50  --    RegDSel    - register to read onto double width bus D (log regcnt bits)
51  --    clock      - the system clock
52  --    reset      - the system clock (async, active low)
53  --
54  --  Outputs:
55  --    RegA       - register value for bus A
56  --    RegB       - register value for bus B
57  --    RegA1      - register value for address bus 1
58  --    RegA2      - register value for address bus 2
59  --    RegD       - register value for bus D (double width bus)
60  --
61
62  library ieee;
63  use ieee.std_logic_1164.all;
64  use std.textio.all;
65  use work.Logging.all;
66
```

```
67  entity  RegArray  is
68
69      generic (
70          regcnt   : integer := 32;    -- default number of registers is 32
71          wordsize : integer := 8      -- default width is 8-bits
72      );
73
74      port(
75          RegIn      : in   std_logic_vector(wordsize - 1 downto 0);
76          RegInSel   : in   integer  range regcnt - 1 downto 0;
77          RegStore   : in   std_logic;
78          RegASel    : in   integer  range regcnt - 1 downto 0;
79          RegBSel    : in   integer  range regcnt - 1 downto 0;
80          RegAxIn    : in   std_logic_vector(wordsize - 1 downto 0);
81          RegAxInSel : in   integer  range regcnt - 1 downto 0;
82          RegAxStore : in   std_logic;
83          RegA1Sel   : in   integer  range regcnt - 1 downto 0;
84          RegA2Sel   : in   integer  range regcnt - 1 downto 0;
85          RegDIn     : in   std_logic_vector(2 * wordsize - 1 downto 0);
86          RegDInSel  : in   integer  range regcnt/2 - 1 downto 0;
87          RegDStore  : in   std_logic;
88          RegDSel    : in   integer  range regcnt/2 - 1 downto 0;
89          clock      : in   std_logic;
90          reset      : in   std_logic;
91          RegA       : out  std_logic_vector(wordsize - 1 downto 0);
92          RegB       : out  std_logic_vector(wordsize - 1 downto 0);
93          RegA1      : out  std_logic_vector(wordsize - 1 downto 0);
94          RegA2      : out  std_logic_vector(wordsize - 1 downto 0);
95          RegD       : out  std_logic_vector(2 * wordsize - 1 downto 0)
96      );
97
98  end  RegArray;
99
100
101 architecture  behavioral  of  RegArray  is
102
103     type  RegType  is array (regcnt - 1 downto 0) of
104                        std_logic_vector(wordsize - 1 downto 0);
105
106     signal  Registers : RegType;                -- the register array
107
108     -- aliases for the upper and lower input word
109     alias  RegDInHigh : std_logic_vector(wordsize - 1 downto 0) IS
110                           RegDIn(2 * wordsize - 1 downto wordsize);
111     alias  RegDInLow  : std_logic_vector(wordsize - 1 downto 0) IS
112                           RegDIn(wordsize - 1 downto 0);
113
114 begin
115
116     -- setup the outputs - choose based on select signals
117     RegA   <=  Registers(RegASel);
118     RegB   <=  Registers(RegBSel);
119     RegA1  <=  Registers(RegA1Sel);
120     RegA2  <=  Registers(RegA2Sel);
121     RegD   <=  Registers(2 * RegDSel + 1) & Registers(2 * RegDSel);
122
123
124     -- only write registers on the clock, plus async reset (active low)
125     process(clock, reset)
126         variable l : line;
127     begin
128         if (reset = '0') then
129             -- set all registers to 0 on async reset
130             Registers  <=  (others => (others => '0'));
131         elsif  rising_edge(clock)  then
132             -- update registers on clock rising edge
```

```
133
134             -- handle double word stores
135             if (RegDStore = '1')  then
136                 Registers(2 * RegDInSel + 1)  <=  RegDInHigh;
137                 Registers(2 * RegDInSel)      <=  RegDInLow;
138             end if;
139
140             -- handle address register stores second so they take precedence
141             --    over double word stores
142             if (RegAxStore = '1')  then
143                 Registers(RegAxInSel)  <=  RegAxIn;
144             end if;
145
146             -- handle normal stores last so they have highest precedence
147             if (RegStore = '1')  then
148                 LogWithTime(l, "Storing " & to_hstring(RegIn) & " to R" & to_string(RegInSel), LogFile);
149                 Registers(RegInSel)  <=  RegIn;
150             end if;
151         else
152             -- have registers retain their value
153             Registers  <=  Registers;
154         end if;
155
156     end process;
157
158 end  behavioral;
159
```

```
1   --------------------------------------------------------------------------------
2   --
3   --  Memory Subsystem
4   --
5   --  This component describes the memory for a 32-bit byte-addressable CPU
6   --  with a 32-bit address bus.  Only a portion of the full address space is
7   --  filled in.  Addresses outside the filled in range return 'X' when read
8   --  and generate error messages when written.
9   --
10  --  Revision History:
11  --     28 Apr 25  Glen George       Initial revision.
12  --     29 Apr 25  Glen George       Fixed some syntax errors.
13  --     29 Apr 25  Glen George       Fixed inconsistencies in byte vs word
14  --                                  addressing.
15  --     01 May 25  Zack Huang        Fixed compile errors
16  --     14 May 25  Chris M.          Track locations that have been written.
17  --     16 May 25  Zack Huang        Added more documentation
18  --
19  --------------------------------------------------------------------------------
20
21
22  --
23  --  MEMORY32x32
24  --
25  --  This is a memory component that supports a byte-addressable 32-bit wide
26  --  memory with 32-bits of address.  No timing restrictions are implemented,
27  --  but if the address bus changes while a WE signal is active an error is
28  --  generated.  Only a portion of the memory is actually usable.  Addresses
29  --  outside of the four usable ranges return 'X' on read and generate error
30  --  messages on write.  The size and address of each memory chunk are generic
31  --  parameters.
32  --
33  --  Generics:
34  --    MEMSIZE     - size of the four memory blocks in 32-bit words
35  --    START_ADDR0 - starting address of first memory block/chunk
36  --    START_ADDR1 - starting address of second memory block/chunk
37  --    START_ADDR2 - starting address of third memory block/chunk
38  --    START_ADDR3 - starting address of fourth memory block/chunk
39  --
40  --  Inputs:
41  --    RE0    - low byte read enable (active low)
42  --    RE1    - byte 1 read enable (active low)
43  --    RE2    - byte 2 read enable (active low)
44  --    RE3    - high byte read enable (active low)
45  --    WE0    - low byte write enable (active low)
46  --    WE1    - byte 1 write enable (active low)
47  --    WE2    - byte 2 write enable (active low)
48  --    WE3    - high byte write enable (active low)
49  --    MemAB  - memory address bus (32 bits)
50  --
51  --  Inputs/Outputs:
52  --    MemDB  - memory data bus (32 bits)
53  --
54
55  library ieee;
56
57  use ieee.std_logic_1164.all;
58  use ieee.numeric_std.all;
59
60  use work.Logging.all;
61  use work.Utils.all;
62
63  entity  MEMORY32x32  is
64
65      generic (
66          MEMSIZE     : integer := 256;   -- default size is 256 words
```

```
67          START_ADDR0 : integer;          -- starting address of first block
68          START_ADDR1 : integer;          -- starting address of second block
69          START_ADDR2 : integer;          -- starting address of third block
70          START_ADDR3 : integer           -- starting address of fourth block
71      );
72
73      port (
74          RE0    : in     std_logic;      -- low byte read enable (active low)
75          RE1    : in     std_logic;      -- byte 1 read enable (active low)
76          RE2    : in     std_logic;      -- byte 2 read enable (active low)
77          RE3    : in     std_logic;      -- high byte read enable (active low)
78          WE0    : in     std_logic;      -- low byte write enable (active low)
79          WE1    : in     std_logic;      -- byte 1 write enable (active low)
80          WE2    : in     std_logic;      -- byte 2 write enable (active low)
81          WE3    : in     std_logic;      -- high byte write enable (active low)
82          MemAB  : in     std_logic_vector(31 downto 0);  -- memory address bus
83          MemDB  : inout  std_logic_vector(31 downto 0)   -- memory data bus
84      );
85
86  end  MEMORY32x32;
87
88
89  architecture  behavioral  of  MEMORY32x32  is
90
91      -- define the type for the RAM chunks
92      type  RAMtype  is array (0 to MEMSIZE - 1) of std_logic_vector(31 downto 0);
93
94      -- now define the RAMs (initialized to X)
95      signal  RAMbits0  :  RAMtype  := (others => (others => 'X'));
96      signal  RAMbits1  :  RAMtype  := (others => (others => 'X'));
97      signal  RAMbits2  :  RAMtype  := (others => (others => 'X'));
98      signal  RAMbits3  :  RAMtype  := (others => (others => 'X'));
99
100     -- general read and write signals
101     signal  RE  :  std_logic;
102     signal  WE  :  std_logic;
103
104     -- data read from memory
105     signal  Curr_RAM  :  RAMtype;
106     signal  RamAddr   :  integer;
107     signal  MemData   :  std_logic_vector(31 downto 0);
108
109 begin
110
111     -- LogAddrRange : process
112     -- begin
113     --   LogWithTime("memory.vhd: Initializing memory from byte " & to_string(START_ADDR0), LogFile);
114     --   LogWithTime("memory.vhd: Initializing memory from byte " & to_string(START_ADDR1), LogFile);
115     --   LogWithTime("memory.vhd: Initializing memory from byte " & to_string(START_ADDR2), LogFile);
116     --   LogWithTime("memory.vhd: Initializing memory from byte " & to_string(START_ADDR3), LogFile);
117     --   wait; -- wait forever
118     -- end process LogAddrRange;
119
120     -- compute the general read and write signals (active low signals)
121     RE  <=  RE0  and  RE1  and  RE2  and  RE3;
122     WE  <=  WE0  and  WE1  and  WE2  and  WE3;
123
124
125     -- On input change, combinatorially compute the address and segment of RAM
126     -- that needs to be accessed.
127     ram_access: process (all) is
128     begin
129         -- Check that MemAB is a valid value and is within the range of the integer type
130         if not is_x(MemAB) and unsigned(MemAB) <= to_unsigned(integer'high, 32) then
131             if ((to_integer(unsigned(MemAB)) >= START_ADDR0) and
132                 (to_integer(unsigned(MemAB) - START_ADDR0) < (4 * MEMSIZE))) then
```

```
133                     Curr_RAM <= RAMBits0;
134                     RamAddr <= to_integer(unsigned(MemAB(31 downto 2))) - START_ADDR0 / 4;
135             elsif ((to_integer(unsigned(MemAB)) >= START_ADDR1) and
136                    (to_integer(unsigned(MemAB) - START_ADDR1) < (4 * MEMSIZE))) then
137                     Curr_RAM <= RAMBits1;
138                     RamAddr <= to_integer(unsigned(MemAB(31 downto 2))) - START_ADDR1 / 4;
139             elsif ((to_integer(unsigned(MemAB)) >= START_ADDR2) and
140                    (to_integer(unsigned(MemAB) - START_ADDR2) < (4 * MEMSIZE))) then
141                     Curr_RAM <= RAMBits2;
142                     RamAddr <= to_integer(unsigned(MemAB(31 downto 2))) - START_ADDR2 / 4;
143             elsif ((to_integer(unsigned(MemAB)) >= START_ADDR3) and
144                    (to_integer(unsigned(MemAB) - START_ADDR3) < (4 * MEMSIZE))) then
145                     Curr_RAM <= RAMBits3;
146                     RamAddr <= to_integer(unsigned(MemAB(31 downto 2))) - START_ADDR3 / 4;
147             else
148                 Curr_RAM <= (others => (others => 'X'));
149                 RamAddr <= -1;
150             end if;
151         end if;
152     end process;
153 
154 
155     -- Get the 32 bits from the address being read from/written to (as
156     -- a extension of the ram_access process)
157     MemData <= Curr_RAM(RamAddr) when RamAddr >= 0 and RamAddr < MEMSIZE else (others => 'X');
158 
159 
160     -- On read, simply output the (masked) bytes that were accessed
161     -- combinatorially (e.g. MemData)
162     read_proc: process (all) is
163     begin
164 
165         -- first check if reading
166         if  (RE = '0' and not IS_X(MemAB))  then
167             -- report "Reading " & to_hstring(MemAB) & ", got " & to_hstring(MemData);
168             MemDB <= MemData;
169 
170             -- only set the bytes that are being read
171             if  RE0 /= '0'  then
172                 MemDB(7 downto 0) <= (others => 'Z');
173             end if;
174             if  RE1 /= '0'  then
175                 MemDB(15 downto 8) <= (others => 'Z');
176             end if;
177             if  RE2 /= '0'  then
178                 MemDB(23 downto 16) <= (others => 'Z');
179             end if;
180             if  RE3 /= '0'  then
181                 MemDB(31 downto 24) <= (others => 'Z');
182             end if;
183 
184         else
185             -- not reading, send data bus to hi-Z
186             MemDB <= (others => 'Z');
187         end if;
188 
189     end process;
190 
191 
192     -- On write, set the desired bytes within the RAM segment being currently
193     -- accessed, at the correct address within the RAM (previously computed
194     -- combinatorially).
195     write_proc: process (all) is
196     begin
197 
198         -- check if writing
```

```
199         if  (WE'event and (WE = '0') and not (is_x(MemAB)) and
200              unsigned(MemAB) <= to_unsigned(integer'high, 32))  then
201             -- rising edge of write - write the data (check which address range)
202             -- report "Writing to " & to_hstring(MemAB) & " with: " & to_hstring(MemDB);
203 
204             -- write the updated value to memory (computed combinatorially
205             -- above), with byte masking
206             if  ((to_integer(unsigned(MemAB)) >= START_ADDR0) and
207                  (to_integer(unsigned(MemAB) - START_ADDR0) < (4 * MEMSIZE)))  then
208                 if (WE0 = '0') then RAMbits0(RamAddr)(7 downto 0) <= MemDB(7 downto 0); end if;
209                 if (WE1 = '0') then RAMbits0(RamAddr)(15 downto 8) <= MemDB(15 downto 8); end if;
210                 if (WE2 = '0') then RAMbits0(RamAddr)(23 downto 16) <= MemDB(23 downto 16); end if;
211                 if (WE3 = '0') then RAMbits0(RamAddr)(31 downto 24) <= MemDB(31 downto 24); end if;
212             elsif  ((to_integer(unsigned(MemAB)) >= START_ADDR1) and
213                     (to_integer(unsigned(MemAB) - START_ADDR1) < (4 * MEMSIZE)))  then
214                 if (WE0 = '1') then RAMbits0(RamAddr)(7 downto 0) <= MemDB(7 downto 0); end if;
215                 if (WE1 = '1') then RAMbits0(RamAddr)(15 downto 8) <= MemDB(15 downto 8); end if;
216                 if (WE2 = '1') then RAMbits0(RamAddr)(23 downto 16) <= MemDB(23 downto 16); end if;
217                 if (WE3 = '1') then RAMbits0(RamAddr)(31 downto 24) <= MemDB(31 downto 24); end if;
218             elsif  ((to_integer(unsigned(MemAB)) >= START_ADDR2) and
219                     (to_integer(unsigned(MemAB) - START_ADDR2) < (4 * MEMSIZE)))  then
220                 if (WE0 = '0') then RAMbits2(RamAddr)(7 downto 0) <= MemDB(7 downto 0); end if;
221                 if (WE1 = '0') then RAMbits2(RamAddr)(15 downto 8) <= MemDB(15 downto 8); end if;
222                 if (WE2 = '0') then RAMbits2(RamAddr)(23 downto 16) <= MemDB(23 downto 16); end if;
223                 if (WE3 = '0') then RAMbits2(RamAddr)(31 downto 24) <= MemDB(31 downto 24); end if;
224             elsif  ((to_integer(unsigned(MemAB)) >= START_ADDR3) and
225                     (to_integer(unsigned(MemAB) - START_ADDR3) < (4 * MEMSIZE)))  then
226                 if (WE0 = '0') then RAMbits3(RamAddr)(7 downto 0) <= MemDB(7 downto 0); end if;
227                 if (WE1 = '0') then RAMbits3(RamAddr)(15 downto 8) <= MemDB(15 downto 8); end if;
228                 if (WE2 = '0') then RAMbits3(RamAddr)(23 downto 16) <= MemDB(23 downto 16); end if;
229                 if (WE3 = '0') then RAMbits3(RamAddr)(31 downto 24) <= MemDB(31 downto 24); end if;
230             else
231                 -- outside of any allowable address range - generate an error
232                 assert (false)
233                     report  "Attempt to write to a non-existant address"
234                     severity  ERROR;
235             end if;
236 
237         end if;
238 
239         -- finally check if WE low with the address changing
240         if  (MemAB'event and (WE = '0'))  then
241             -- output error message
242             REPORT "Glitch on Memory Address bus"
243             SEVERITY  ERROR;
244         end if;
245 
246     end process;
247 
248 
249 end  behavioral;
250 
251 
```

```
1   --------------------------------------------------------------------------------
2   -- logging.vhd
3   --
4   -- Logging Utility package.
5   --
6   --  Revision History:
7   --    12 May 25  Chris M. Initial reivision.
8   --
9   --------------------------------------------------------------------------------
10  library std;
11  use std.textio.all;
12
13  package Logging is
14
15    -- Log file;
16    file LogFile : text open write_mode is "log.txt";
17
18    -- Log to stdout
19    procedure Log(message : in string;
20                  Enable  : boolean := true);
21
22    procedure Log(l       : inout line;
23                  message : in string;
24                  Enable : boolean := true);
25
26    -- Log to a file.
27    procedure Log(message          : in string;
28                  file file_handle : text;
29                  Enable           : boolean := true);
30
31    procedure Log(l                : inout line;
32                  message          : in string;
33                  file file_handle : text; Enable : boolean := true);
34
35    -- Log to stdout and prefix the message with the current time.
36    procedure LogWithTime(message : in string;
37                          Enable  : boolean := true);
38
39    procedure LogWithTime(l       : inout line;
40                          message : in string;
41                          Enable  : boolean := true);
42
43    -- Log to a file and prefix the message with the current time.
44    procedure LogWithTime(message          : in string;
45                          file file_handle : text;
46                          Enable           : boolean := true);
47
48    procedure LogWithTime(l                : inout line;
49                          message          : in string;
50                          file file_handle : text;
51                          Enable           : boolean := true);
52
53    -- Log to both stdout and a file.
54    procedure LogBoth(message          : in string;
55                      file file_handle : text;
56                      Enable           : boolean := true);
57
58    procedure LogBoth(l_1              : inout line;
59                      l_2              : inout line;
60                      message          : in string;
61                      file file_handle : text;
62                      Enable           : boolean := true);
63
64    -- Log to both stdout and a file and prefix the message with the current time.
65    procedure LogBothWithTime(message          : in string;
66                              file file_handle : text;
```

```
67                              Enable           : boolean := true);
68
69    procedure LogBothWithTime(l_1 : inout line;
70                              l_2 : inout line;
71                              message : in string;
72                              file file_handle : text;
73                              Enable : boolean := true);
74
75  end package Logging;
76
77
78  library std;
79  use std.textio.all;
80
81  package body Logging is
82
83    -- Log to stdout
84    procedure Log(message : in string;
85                  Enable  : boolean := true) is
86      variable l : line;
87    begin
88      if (not Enable) then
89        null;
90      else
91        write(l, message);
92        writeline(output, l);
93      end if;
94    end procedure Log;
95
96    procedure Log(l       : inout line;
97                  message : in string;
98                  Enable : boolean := true) is
99    begin
100     if (not Enable) then
101       null;
102     else
103       write(l, message);
104       writeline(output, l);
105     end if;
106   end procedure Log;
107
108   -- Log to a file.
109   procedure Log(message          : in string;
110                 file file_handle : text;
111                 Enable           : boolean := true) is
112     variable l : line;
113   begin
114     if (not Enable) then
115       null;
116     else
117       write(l, message);
118       writeline(file_handle, l);
119     end if;
120   end procedure Log;
121
122   procedure Log(l                : inout line;
123                 message          : in string;
124                 file file_handle : text; Enable : boolean := true) is
125   begin
126     if (not Enable) then
127       null;
128     else
129       write(l, message);
130       writeline(file_handle, l);
131     end if;
132   end procedure Log;
```

```
133
134   -- Log to stdout and prefix the message with the current time.
135   procedure LogWithTime(message : in string;
136                         Enable  : boolean := true) is
137     variable l : line;
138   begin
139     if (not Enable) then
140       null;
141     else
142       write(l, string'("[@"));
143       write(l, now);
144       write(l, string'("] "));
145       write(l, message);
146       writeline(output, l);
147     end if;
148   end procedure LogWithTime;
149
150   procedure LogWithTime(l       : inout line;
151                         message : in string;
152                         Enable  : boolean := true) is
153   begin
154     if (not Enable) then
155       null;
156     else
157       write(l, string'("[@"));
158       write(l, now);
159       write(l, string'("] "));
160       write(l, message);
161       writeline(output, l);
162     end if;
163   end procedure LogWithTime;
164
165   -- Log to a file and prefix the message with the current time.
166   procedure LogWithTime(message          : in string;
167                         file file_handle : text;
168                         Enable           : boolean := true) is
169     variable l : line;
170   begin
171     if (not Enable) then
172       null;
173     else
174       write(l, string'("[@"));
175       write(l, now);
176       write(l, string'("] "));
177       write(l, message);
178       writeline(file_handle, l);
179     end if;
180   end procedure LogWithTime;
181
182   procedure LogWithTime(l                : inout line;
183                         message          : in string;
184                         file file_handle : text;
185                         Enable           : boolean := true) is
186   begin
187     if (not Enable) then
188       null;
189     else
190       write(l, string'("[@"));
191       write(l, now);
192       write(l, string'("] "));
193       write(l, message);
194       writeline(file_handle, l);
195     end if;
196   end procedure LogWithTime;
197
198   -- Log to both stdout and a file.
```

```
199   procedure LogBoth(message          : in string;
200                     file file_handle : text;
201                     Enable           : boolean := true) is
202     variable l_1 : line;
203     variable l_2 : line;
204   begin
205     if (not Enable) then
206       null;
207     else
208       write(l_1, message);
209       writeline(file_handle, l_1);
210       writeline(output, l_2);
211     end if;
212   end procedure LogBoth;
213
214   procedure LogBoth(l_1              : inout line;
215                     l_2              : inout line;
216                     message          : in string;
217                     file file_handle : text;
218                     Enable           : boolean := true) is
219   begin
220     if (not Enable) then
221       null;
222     else
223       write(l_1, message);
224       writeline(file_handle, l_1);
225       writeline(output, l_2);
226     end if;
227   end procedure LogBoth;
228
229   -- Log to both stdout and a file and prefix the message with the current time.
230   procedure LogBothWithTime(message          : in string;
231                             file file_handle : text;
232                             Enable           : boolean := true) is
233     variable l_1 : line;
234     variable l_2 : line;
235   begin
236     if (not Enable) then
237       null;
238     else
239       write(l_1, string'("[@"));
240       write(l_1, now);
241       write(l_1, string'("] "));
242       write(l_1, message);
243
244       write(l_2, string'("[@"));
245       write(l_2, now);
246       write(l_2, string'("] "));
247       write(l_2, message);
248
249       writeline(output, l_1);
250       writeline(file_handle, l_2);
251     end if;
252   end procedure LogBothWithTime;
253
254   procedure LogBothWithTime(l_1 : inout line;
255                             l_2 : inout line;
256                             message : in string;
257                             file file_handle : text;
258                             Enable : boolean := true) is
259   begin
260     if (not Enable) then
261       null;
262     else
263       write(l_1, string'("[@"));
264       write(l_1, now);
```

```
265       write(l_1, string'("] "));
266       write(l_1, message);
267
268       write(l_2, string'("[@"));
269       write(l_2, now);
270       write(l_2, string'("] "));
271       write(l_2, message);
272
273       writeline(output, l_1);
274       writeline(file_handle, l_2);
275     end if;
276   end procedure LogBothWithTime;
277
278 end package body Logging;
279
```

```
1   --------------------------------------------------------------------------------
2   -- utils.vhd
3   --
4   -- Miscellaneous functions and procedures for SH-2 block testing. Currently
5   -- includes randomization for an SH-2 word, as well as utility functions
6   -- for converting between ints, and std_logic_vector.
7   --
8   -- Packages provided:
9   --    Utils    - generic utility functions.
10  --
11  --  Revision History:
12  --    28 April 25   Zach H.    Initial revision.
13  --    30 April 25   Chris M.   Add conversion functions.
14  --    01 June  25   Zach H.    Combined functions into a single package
15  --
16  --------------------------------------------------------------------------------
17
18
19  library ieee;
20  use ieee.std_logic_1164.all;
21  use ieee.numeric_std.all;
22  use ieee.math_real.all;
23
24  package Utils is
25
26      type rng is protected
27          impure function rand_slv(len : integer) return std_logic_vector;
28      end protected rng;
29
30    function int_to_slv  (i : integer; width : natural)  return std_logic_vector;
31    function uint_to_slv (i : natural; width : natural)  return std_logic_vector;
32    function slv_to_int  (slv : std_logic_vector) return integer;
33    function slv_to_uint (slv : std_logic_vector) return natural;
34
35  end package Utils;
36
37  package body Utils is
38
39      type rng is protected body
40          variable seed1, seed2 : integer := 1000;
41
42          impure function rand_slv(len : integer) return std_logic_vector is
43              variable r : real;
44              variable slv : std_logic_vector(len - 1 downto 0);
45          begin
46              for i in slv'range loop
47                  uniform(seed1, seed2, r);
48                  slv(i) := '1' when r > 0.5 else '0';
49              end loop;
50              return slv;
51          end function;
52      end protected body;
53
54    function int_to_slv (i : integer; width : natural) return std_logic_vector is
55      variable max_int : signed(width - 1 downto 0);
56      variable min_int : signed(width - 1 downto 0);
57    begin
58
59      max_int := (others => '1');
60      max_int(max_int'high) := '0';
61
62      min_int := (others => '0');
63      min_int(max_int'high) := '1';
64
65      assert ((width <= 32) and (to_signed(i, width) <= MAX_INT) and
66              (to_signed(i, width) >= MIN_INT))
```

```
67        report "signed integer " & to_string(i) & " cannot be converted to a " &
68               "std_logic_vector of width " & to_string(width)
69        severity ERROR;
70
71      return std_logic_vector(to_signed(i, width));
72
73    end function;
74
75    function uint_to_slv (i : natural; width : natural)  return std_logic_vector is
76    begin
77
78      assert ((width <= 32) and (i <= 2**(width - 1) - 1))
79        report "signed integer " & to_string(i) & " cannot be converted to a " &
80               "std_logic_vector of width " & to_string(width)
81        severity ERROR;
82
83      return std_logic_vector(to_unsigned(i, width));
84
85    end function;
86
87    function slv_to_int  (slv : std_logic_vector) return integer is
88      constant MIN_32_SIGNED : std_logic_vector(31 downto 0) := x"80000000";
89    begin
90      -- The VHDL integer range is guaranteed to be at least,
91      -- -2,147,483,647 to +2,147,483,647, but 2**31 in two's complement is
92      -- -2,147,483,648. Trying to convert this to an integer causes a runtime
93      -- error.
94      if (slv'length = 32) then
95        assert (slv /= MIN_32_SIGNED)
96          report "std_logic_vector " & to_string(slv) & " cannot be represented as " &
97                 "an integer."
98          severity ERROR;
99      end if;
100
101     assert ((slv'length <= 32))
102       report "std_logic_vector is too wide to be represented as an integer (length = " &
103              to_string(slv'length) & " )"
104       severity ERROR;
105
106     return to_integer(signed(slv));
107
108   end function;
109
110   function slv_to_uint (slv : std_logic_vector) return natural is
111     constant MAX_32_SIGNED : std_logic_vector(31 downto 0) := x"7fffffff";
112   begin
113
114     if (slv'length = 32) then
115       assert (slv /= MAX_32_SIGNED)
116         report "std_logic_vector " & to_string(slv) & " cannot be represented as " &
117                "an integer."
118         severity ERROR;
119     end if;
120
121     assert ((slv'length <= 32))
122       report "std_logic_vector is too wide to be represented as an integer (length = " &
123              to_string(slv'length) & " )"
124       severity ERROR;
125
126     return to_integer(unsigned(slv));
127
128   end function;
129
130 end package body Utils;
131
```

```
1  --------------------------------------------------------------------------------
2  --  sh2_constants.vhd;
3  --
4  -- Shared SH-2 constants.
5  --
6  -- This file describes constants that are shared across multiple blocks of
7  -- the SH-2.
8  --
9  -- Packages Provided:
10 --    SH2Constants
11 --
12 --  Revision History:
13 --    16 April 25   Chris M. Initial reivision.
14 --
15 --------------------------------------------------------------------------------
16
17 library ieee;
18
19 use ieee.std_logic_1164.all;
20 use ieee.numeric_std.all;
21
22 package SH2Constants is
23   constant SH2_WORDSIZE     : integer := 32;
24   constant SH2_REGCNT       : integer := 16;
25
26   -- Note that these must be hard-coded to avoid overflow in static computation.
27   -- This is just (-(2**(SH2_WORDSIZE - 1)))
28   constant SH2_SIGNED_MIN   : integer := -2147483648;
29   constant SH2_SIGNED_MAX   : integer := 2147483647;
30   constant SH2_UNSIGNED_MIN : integer := 0;
31   constant SH2_UNSIGNED_MAX : unsigned(SH2_WORDSIZE - 1 downto 0) := (others => '1');
32
33 end package SH2Constants;
34
35
36
```

```
1   --------------------------------------------------------------------------------
2   -- SH2ALU
3   --
4   -- This entity implements all the operations required for the SH-2 ALU. It
5   -- supports arithmetic, logical, and shift operations as required for the SH-2
6   -- instruction set. It does not implement DSP operations such as MAC or barrel
7   -- shifting. This entity does not do instruction decoding - the control unit
8   -- must provide the correct operand values (which could come from registers or
9   -- immediate values) and control signals to produce the correct result (which
10  -- may then be written back to a register by the control unit). Additionally,
11  -- this entity outputs carry, sign, overflow, and zero flags, which can be
12  -- used by the control unit to set the T bit in the status register.
13  --
14  --  Revision History:
15  --     26 Apr 25    Zack Huang      copied over from HW 1, prepare for testing
16  --     25 May 25    Zack Huang      added barrel shifter (extra credit instructions)
17  --
18  --------------------------------------------------------------------------------
19
20  -- Import libraries
21  library ieee;
22  use ieee.std_logic_1164.all;
23
24  package  SH2ALUConstants  is
25
26  --  Adder carry in select constants
27     constant CinCmd_ZERO   : std_logic_vector(1 downto 0) := "00";
28     constant CinCmd_ONE    : std_logic_vector(1 downto 0) := "01";
29     constant CinCmd_CIN    : std_logic_vector(1 downto 0) := "10";
30     constant CinCmd_CINBAR : std_logic_vector(1 downto 0) := "11";
31
32
33  --  Shifter command constants
34     constant SCmd_LEFT  : std_logic_vector(2 downto 0) := "0--"; -- BIT DECODED - DO NOT CHANGE
35     constant SCmd_LSL   : std_logic_vector(2 downto 0) := "000"; -- BIT DECODED - DO NOT CHANGE
36     constant SCmd_ASL   : std_logic_vector(2 downto 0) := "001"; -- BIT DECODED - DO NOT CHANGE
37     constant SCmd_ROL   : std_logic_vector(2 downto 0) := "010"; -- BIT DECODED - DO NOT CHANGE
38     constant SCmd_RLC   : std_logic_vector(2 downto 0) := "011"; -- BIT DECODED - DO NOT CHANGE
39     constant SCmd_RIGHT : std_logic_vector(2 downto 0) := "1--"; -- BIT DECODED - DO NOT CHANGE
40     constant SCmd_LSR   : std_logic_vector(2 downto 0) := "100"; -- BIT DECODED - DO NOT CHANGE
41     constant SCmd_ASR   : std_logic_vector(2 downto 0) := "101"; -- BIT DECODED - DO NOT CHANGE
42     constant SCmd_ROR   : std_logic_vector(2 downto 0) := "110"; -- BIT DECODED - DO NOT CHANGE
43     constant SCmd_RRC   : std_logic_vector(2 downto 0) := "111"; -- BIT DECODED - DO NOT CHANGE
44
45     -- Barrel shifter command constants
46     constant BSCmd_L2  : std_logic_vector(2 downto 0) := "000"; -- BIT DECODED - DO NOT CHANGE
47     constant BSCmd_R2  : std_logic_vector(2 downto 0) := "001"; -- BIT DECODED - DO NOT CHANGE
48     constant BSCmd_L8  : std_logic_vector(2 downto 0) := "010"; -- BIT DECODED - DO NOT CHANGE
49     constant BSCmd_R8  : std_logic_vector(2 downto 0) := "011"; -- BIT DECODED - DO NOT CHANGE
50     constant BSCmd_L16 : std_logic_vector(2 downto 0) := "100"; -- BIT DECODED - DO NOT CHANGE
51     constant BSCmd_R16 : std_logic_vector(2 downto 0) := "101"; -- BIT DECODED - DO NOT CHANGE
52
53  --  ALU command constants
54     constant ALUCmd_FBLOCK  : std_logic_vector(1 downto 0) := "00";
55     constant ALUCmd_ADDER   : std_logic_vector(1 downto 0) := "01";
56     constant ALUCmd_SHIFT   : std_logic_vector(1 downto 0) := "10";
57     constant ALUCmd_BSHIFT  : std_logic_vector(1 downto 0) := "11";
58
59
60      constant OpA_Zero     : std_logic_vector(1 downto 0) := "00"; -- clear OperandA before using it in a computation
61      constant OpA_One      : std_logic_vector(1 downto 0) := "01"; -- Set OperandA value to 1
62      constant OpA_B        : std_logic_vector(1 downto 0) := "10"; -- Set OperandA to have the value of OperandB
63      constant OpA_None     : std_logic_vector(1 downto 0) := "11"; -- Pass OperandA through
64
65      -- FBlock commands (for convenience)
66      constant FCmd_A         : std_logic_vector(3 downto 0) := "1100";
```

```
67      constant FCmd_B         : std_logic_vector(3 downto 0) := "1010";
68      constant FCmd_BNOT      : std_logic_vector(3 downto 0) := "0101";
69      constant FCmd_ONES      : std_logic_vector(3 downto 0) := "1111";
70      constant FCmd_AND       : std_logic_vector(3 downto 0) := "1000";
71      constant FCmd_OR        : std_logic_vector(3 downto 0) := "1110";
72      constant FCmd_XOR       : std_logic_vector(3 downto 0) := "0110";
73
74  end package;
75
76  -- import libraries
77
78  library ieee;
79  use ieee.numeric_std.all;
80  use ieee.std_logic_1164.all;
81  use work.SH2ALUConstants.all;
82
83  -- Set the SH2 ALU control signals as follows for each instruction:
84  -- It is assumed that single-operand instructions operate on OperandB,
85  -- while dual-operand instructions operate on OperandA and OperandB, in
86  -- that order. This is because SH-2 instructions are almost in the format
87  -- "OPCODE Rm, Rn", which usually does something like Rn <= Rn OPERATION Rm.
88  -- If the instruction is unary, then it usually only acts on Rm. As such, the
89  -- CPU can assign OperandA <= Rn and OperandB <= Rm, and then put the result
90  -- back in Rn. This is convenient because of the implementation of the generic
91  -- ALU being used inside this SH-2-specific ALU. We exclude DSP instructions
92  -- (MAC, SHLLn, SHLRN, etc) and multi-clock instructions (DIV, MUL, etc) for
93  -- now.
94  --
95  -- The possible ALU operations with the control signals that produce them
96  -- are listed below. Note that this entity also outputs carry, overflow,
97  -- zero, and sign flags so that they can be used by the CPU for setting
98  -- the T bit, doing sign-extension, checking compare results, etc.
99  --
100 -- ADD(C,V) - Result <= OperandA + OperandB
101 --   - FCmd <= FCmd_B
102 --   - CinCmd <= CinCmd_ZERO
103 --   - LoadA <= '1'
104 --   - SCmd <= "XX"
105 --   - ALUCmd <= ALUCmd_ADDER
106 -- SUB(C,V), CMP/XX - Result <= OperandA - OperandB
107 --   - FCmd <= FCmd_BNOT
108 --   - CinCmd <= CinCmd_ONE
109 --   - LoadA <= '1'
110 --   - SCmd <= "XX"
111 --   - ALUCmd <= ALUCmd_ADDER
112 -- NEG(C) - Result <= 0 - OperandB
113 --   - FCmd <= FCmd_BNOT
114 --   - CinCmd <= CinCmd_ONE
115 --   - LoadA <= '0'
116 --   - SCmd <= "XX"
117 --   - ALUCmd <= ALUCmd_ADDER
118 -- DT - Result <= OperandA - 1
119 --   - FCmd <= FCmd_ONES
120 --   - CinCmd <= CinCmd_ZERO
121 --   - LoadA <= '1'
122 --   - SCmd <= "XX"
123 --   - ALUCmd <= ALUCmd_ADDER
124 -- MOV - Result <= OperandB
125 --   - FCmd <= FCmd_B
126 --   - CinCmd <= CinCmd_ZERO
127 --   - LoadA <= '1'
128 --   - SCmd <= "XX"
129 --   - ALUCmd <= ALUCmd_FBLOCK
130 -- AND/TST - Result <= OperandA & OperandB
131 --   - FCmd <= FCmd_AND
132 --   - CinCmd <= CinCmd_ZERO
```

```
133 --  - LoadA <= '1'
134 --  - SCmd <= "XX"
135 --  - ALUCmd <= ALUCmd_FBLOCK
136 -- OR - Result <= OperandA | OperandB
137 --  - FCmd <= FCmd_OR
138 --  - CinCmd <= CinCmd_ZERO
139 --  - LoadA <= '1'
140 --  - SCmd <= "XX"
141 --  - ALUCmd <= ALUCmd_FBLOCK
142 -- XOR - Result <= OperandA ^ OperandB
143 --  - FCmd <= FCmd_XOR
144 --  - CinCmd <= CinCmd_ZERO
145 --  - LoadA <= '1'
146 --  - SCmd <= "XX"
147 --  - ALUCmd <= ALUCmd_FBLOCK
148 -- NOT - Result <= ~OperandB
149 --  - FCmd   <= FCmd_BNOT
150 --  - CinCmd <= CinCmd_ZERO
151 --  - LoadA <= '1'
152 --  - SCmd   <= "XX"
153 --  - ALUCmd <= ALUCmd_FBLOCK
154 -- SHAL/SHLL - Result <= OperandA << 1
155 --  - FCmd   <= "XX"
156 --  - CinCmd <= CinCmd_ZERO
157 --  - LoadA <= '1'
158 --  - SCmd   <= SCmd_LSR
159 --  - ALUCmd <= ALUCmd_SHIFT
160 -- SHAR - Result <= OperandA >> 1 (sign-extended)
161 --  - FCmd   <= "XX"
162 --  - CinCmd <= CinCmd_ZERO
163 --  - LoadA <= '1'
164 --  - SCmd   <= SCmd_ASR
165 --  - ALUCmd <= ALUCmd_SHIFT
166 -- SHLR - Result <= OperandA >> 1
167 --  - FCmd   <= "XX"
168 --  - CinCmd <= CinCmd_ZERO
169 --  - LoadA <= '1'
170 --  - SCmd   <= SCmd_LSR
171 --  - ALUCmd <= ALUCmd_SHIFT
172 -- ROTL - Result <= rotate_left(OperandA)
173 --  - FCmd   <= "XX"
174 --  - CinCmd <= CinCmd_ZERO
175 --  - LoadA <= '1'
176 --  - SCmd   <= SCmd_ROL
177 --  - ALUCmd <= ALUCmd_SHIFT
178 -- ROTR - Result <= rotate_right(OperandA)
179 --  - FCmd   <= "XX"
180 --  - CinCmd <= CinCmd_ZERO
181 --  - LoadA <= '1'
182 --  - SCmd   <= SCmd_ROR
183 --  - ALUCmd <= ALUCmd_SHIFT
184 -- ROTCL - Result <= rotate_left(OperandA, T)
185 --  - FCmd   <= "XX"
186 --  - CinCmd <= CinCmd_CIN
187 --  - LoadA <= '1'
188 --  - SCmd   <= SCmd_RLC
189 --  - ALUCmd <= ALUCmd_SHIFT
190 -- ROTCR - Result <= rotate_right(OperandA, T)
191 --  - FCmd   <= "XX"
192 --  - CinCmd <= CinCmd_CIN
193 --  - LoadA <= '1'
194 --  - SCmd   <= SCmd_RRC
195 --  - ALUCmd <= ALUCmd_SHIFT
196
197 entity sh2alu is
198     port (
```

```
199         OperandA : in    std_logic_vector(31 downto 0); -- first operand
200         OperandB : in    std_logic_vector(31 downto 0); -- second operand
201         TIn      : in    std_logic;                     -- T bit from status register
202         LoadA    : in    std_logic;                     -- determine if OperandA is loaded ('1') or zeroed ('0')
203         FCmd     : in    std_logic_vector(3 downto 0);  -- F-Block operation
204         CinCmd   : in    std_logic_vector(1 downto 0);  -- carry in operation
205         SCmd     : in    std_logic_vector(2 downto 0);  -- shift operation
206         ALUCmd   : in    std_logic_vector(1 downto 0);  -- ALU result select
207
208         Result   : buffer  std_logic_vector(31 downto 0); -- ALU result
209         Cout     : out   std_logic;                       -- carry out
210         Overflow : out   std_logic;                       -- signed overflow
211         Zero     : out   std_logic;                       -- result is zero
212         Sign     : out   std_logic                        -- sign of result
213     );
214 end entity sh2alu;
215
216 architecture structural of sh2alu is
217
218     component ALU is
219
220         generic (
221             wordsize : integer := 8
222         );
223         port (
224             AluOpA : in    std_logic_vector(wordsize - 1 downto 0);
225             AluOpB : in    std_logic_vector(wordsize - 1 downto 0);
226             Cin    : in    std_logic;
227             FCmd   : in    std_logic_vector(3 downto 0);
228             CinCmd : in    std_logic_vector(1 downto 0);
229             SCmd   : in    std_logic_vector(2 downto 0);
230             AluCmd : in    std_logic_vector(1 downto 0);
231
232             Result   : buffer  std_logic_vector(wordsize - 1 downto 0);
233             Cout     : out   std_logic;
234             HalfCout : out   std_logic;
235             Overflow : out   std_logic;
236             Zero     : out   std_logic;
237             Sign     : out   std_logic
238         );
239     end component ALU;
240
241     signal BarrelShifter : std_logic_vector(31 downto 0);
242
243     signal ALUResult : std_logic_vector(31 downto 0);
244
245 begin
246     -- We use a generic ALU to implement all of the SH-2 ALU operations. We
247     -- pass in the T bit in place of a dedicated carry input, and the CPU can
248     -- route the correct output flag (carry, sign, zero, overflow) back into
249     -- the status register.
250     ALUinternal : component ALU
251         generic map (
252             wordsize => 32
253         )
254         port map (
255             AluOpA   => OperandA and LoadA,
256             AluOpB   => OperandB,
257             Cin      => TIn,
258             FCmd     => FCmd,
259             SCmd     => SCmd,
260             AluCmd   => AluCmd,
261             CinCmd   => CinCmd,
262             Result   => ALUResult,
263             Cout     => Cout,
264             Overflow => Overflow,
```

```
265             Zero     => Zero,
266             Sign     => Sign
267         );
268
269     -- We also add in a barrel shifter to implement extra-credit instructions
270     -- For convenience, we will re-use the SCmd bits to control this barrel shifter.
271     with SCmd select
272         BarrelShifter <= OperandA(29 downto 0) & "00"                   when BSCmd_L2,
273                          "00" & OperandA(31 downto 2)                   when BSCmd_R2,
274                          OperandA(23 downto 0) & "00000000"             when BSCmd_L8,
275                          "00000000" & OperandA(31 downto 8)             when BSCmd_R8,
276                          OperandA(15 downto 0) & "0000000000000000"     when BSCmd_L16,
277                          "0000000000000000" & OperandA(31 downto 16)    when BSCmd_R16,
278                          (others => 'X') when others;
279
280     -- Mux between the generic ALU and the barrel shifter to get the result
281     Result <= BarrelShifter when ALUCmd = ALUCmd_BSHIFT else ALUResult;
282
283 end architecture structural;
284
```

```
1   --------------------------------------------------------------------------------
2   --  sh2_dmau.vhd
3   --
4   -- SH-2 DMAU (Data Memory Access Unit).
5   --
6   --  This is an implementation of the SH-2's DMAU using Glen A. George's
7   --  generic memory access unit. The SH-2 is a Princeton architecture CPU
8   --  with only a shared memory and data bus. This DMAU entity calculates
9   --  memory addresses based on the input control signals, and contains
10  --  the GBR (General Base Register). The SH-2 has the following addressing
11  --  modes:
12  --
13  --    1.  Direct Register Addressing
14  --    2.  Indirect Register Addressing
15  --    3.  Post-increment indirect register addressing
16  --    4.  Pre-decrement indirect register addressing
17  --    5.  Indirect register addressing with displacement
18  --    6.  Indirect indexed register addressing
19  --    7.  Indirect GBR addressing with displacement
20  --    8.  Indirect indexed GBR addressing
21  --    9.  PC relative addressing with displacement
22  --    10. PC relative addressing
23  --    11. Immediate addressing
24  --
25  -- The modes are described in further detail in Table 4.7 of the SH-2
26  -- programming manual.
27  --
28  -- Note that the DMAU only takes care of addressing modes related to memory
29  -- locations. As such, the DMAU will not do direct register addressing, since
30  -- that is done by the register array, nor will it do immediate addressing,
31  -- which is done by the control unit during instruction decoding. Finally,
32  -- there are not safeguards against addressing modes which do not actually
33  -- exist in the SH-2.
34  --
35  --  Revision History:
36  --    16 April 25   Chris M. Initial reivision.
37  --    23 April 25   Chris M. Add seperate calculated offsets to AddrOff matrix
38  --                           instead of muxing between a single offset.
39  --
40  --    23 April 25   Chris M. Removed 12-bit offset input and OffExtendSel
41  --                           because 12-bit offsets and sign-extension is not
42  --                           used for memory accesses, only for relative jumps.
43  --
44  --    1 May 25    Chris M. Changed PrePostSel in MAU to be POST when
45  --                         IncDecSel is none (only worked before because
46  --                         Pre/Post logic in MAU was inverted).
47  --
48  --    7 May 25    Chris M. Add
49  --
50  -- [TODO]:
51  --    - Don't allow inputs that don't correspond to addressing modes.
52  --    - Make mapping from IndexSel to DMAUOffsetSel one-to-one ?
53  --    - Only have one offset input ?
54  --
55  --------------------------------------------------------------------------------
56
57  library ieee;
58  library std;
59
60  use ieee.std_logic_1164.all;
61
62  use work.MemUnitConstants.all; -- memory access unit constants for pre/post inc/dec.
63  use work.SH2Constants.all;     -- global SH-2 constants.
64  use work.array_type_pkg.all;   -- 2D Array of std_logic (VHDL-2008 only).
65
66  package SH2DmauConstants is
```

```
67
68    -- BaseSel constants.
69    --
70    constant BaseSel_REG : std_logic_vector(1 downto 0)  := "00"; -- 0
71    constant BaseSel_GBR : std_logic_vector(1 downto 0)  := "01"; -- 1
72    constant BaseSel_PC  : std_logic_vector(1 downto 0)  := "10"; -- 2
73
74    -- IndexSel constants.
75    constant IndexSel_NONE   : std_logic_vector(1 downto 0) := "00";   -- 0
76    constant IndexSel_OFF4   : std_logic_vector(1 downto 0) := "01";   -- 1
77    constant IndexSel_OFF8   : std_logic_vector(1 downto 0) := "10";   -- 2
78    constant IndexSel_R0     : std_logic_vector(1 downto 0) := "11";   -- 3
79
80    -- OffsetScalarSel constant. What to scale the offset (or increment value) by.
81    -- BIT-DECODED, DO NOT CHANGE VALUES
82    constant OffScalarSel_ONE  :  std_logic_vector(1 downto 0) := "00"; -- 0
83    constant OffScalarSel_TWO  :  std_logic_vector(1 downto 0) := "01"; -- 1
84    constant OffScalarSel_FOUR :  std_logic_vector(1 downto 0) := "10"; -- 2
85
86    -- IncDecSel constants.
87    --
88    constant IncDecSel_NONE     : std_logic_vector(1 downto 0) := "00";
89    constant IncDecSel_PRE_DEC  : std_logic_vector(1 downto 0) := "01";
90    constant IncDecSel_POST_INC : std_logic_vector(1 downto 0) := "10";
91
92  end package SH2DmauConstants;
93
94
95  library ieee;
96
97  use ieee.std_logic_1164.all;
98  use ieee.numeric_std.all;
99
100 use work.MemUnitConstants.all; -- memory access unit constants for pre/post inc/dec.
101 use work.SH2Constants.all;     -- global SH-2 constants.
102 use work.array_type_pkg.all;   -- 2D Array of std_logic (VHDL-2008 only).
103 use work.SH2DmauConstants.all;
104
105
106 -- SH2Dmau
107 --
108 -- This is the SH-2s DMAU. It implements the following addressing modes.
109 --
110 --
111 -- +-------------------------------------------------------+-------------------------------------------------------+
112 -- |                   Addressing Mode                     |               Formula(s)                              |
113 -- +-------------------------------------------------------+-------------------------------------------------------+
114 -- | 1.  Indirect Register Addressing                      | Addr = @(Rn)                                          |
115 -- +-------------------------------------------------------+-------------------------------------------------------+
116 -- | 2.  Post-increment indirect register addressing       | (After the instruction is executed)                   |
117 -- |                                                       | Addr = @(Rn + 1)                                      |
118 -- |                                                       | Addr = @(Rn + 2)                                      |
119 -- |                                                       | Addr = @(Rn + 4)                                      |
120 -- +-------------------------------------------------------+-------------------------------------------------------+
121 -- | 3.  Pre-decrement indirect register addressing        | (Before the instruction is executed)                  |
122 -- |                                                       | Addr = @(Rn - 1)                                      |
123 -- |                                                       | Addr = @(Rn - 2)                                      |
124 -- |                                                       | Addr = @(Rn - 4)                                      |
125 -- +-------------------------------------------------------+-------------------------------------------------------+
126 -- | 4.  Indirect register addressing with displacement    | Addr = @(Rn + zero_extend(Off4))                      |
127 -- | -                                                     | Addr = @(Rn + zero_extend(Off4) * 2)                  |
128 -- | -                                                     | Addr = @(Rn + zero_extend(Off4) * 4)                  |
129 -- +-------------------------------------------------------+-------------------------------------------------------+
130 -- | 5.  Indirect indexed register addressing              | Addr = @(Rn + R0), Rn =/= R0                          |
131 -- +-------------------------------------------------------+-------------------------------------------------------+
132 -- | 6.  Indirect GBR addressing with displacement         | Addr = @(GBR + zero_extend(Off8))                     |
```

```
-- | -                                                    | Addr = @(GBR + zero_extend(Off8) * 2)               |
-- | -                                                    | Addr = @(GBR + zero_extend(Off8) * 4)               |
-- +------------------------------------------------------+-----------------------------------------------------+
-- | 7.  Indirect indexed GBR addressing                  | Addr = @(GBR + R0)                                  |
-- +------------------------------------------------------+-----------------------------------------------------+
-- | 8.  PC relative addressing with displacement         | Addr = @(PC + zero_extend(Off8) * 2)                |
-- | -                                                    | Addr = @((PC & 0xFFFFFFFC) + zero_extend(Off8) * 4) |
-- +------------------------------------------------------+-----------------------------------------------------+
--
-- "Base" sources are the sources to the left hand side of an address
-- calculation, and consist of Rn (a register), GBR (Global Base Register), or
-- PC (Program Counter).
--
-- "Index" sources are the sources to the right hand side of an address
-- calculation, and consist of an immediate offset which is either sign or
-- zero extended and then scaled by 1, 2, or 4, R0, or Rn.
--
-- Inputs:
--   RegSrc       -   Input register. One of three possible base sources.
--   R0Src        -   R0 Register input.
--   PCSrc        -   Program Counter Source.
--   GBRIn        -   GBR input.
--   GBRWriteEn   -   GBR write enable. Active high.
--   Off4         -   4-Bit Offset.
--   Off8         -   8-Bit Offset.
--   BaseSel      -   Which base register source to select.
--   IndexSel     -   Which index source to select (None, Off4, Off8, Rn,
--                    or R0). Note that Off4, Off8 are zero extended before
--                    being added to the base.
--
--   OffScalar    -   What to scale the offset by (1, 2, 4)
--   IncDecSel    -   Post Increment or PreDecrement the base. Note that the
--                    amount added or subtracted is scaled ny OffScalar. Note
--                    that even when this is set to NONE, an
--                    incremented/decremented value is output to AddrSrcOut as a
--                    result of the generic memory access unit design.
--   Clk          -   Clk input.
--
-- Outputs:
--
--   Addr       - output address
--   AddrSrcOut - incremented/decremented address (for storing back into register).
--   GBROut     - GBR output.
--
entity SH2Dmau is
  port (
    RegSrc       : in     std_logic_vector(SH2_WORDSIZE - 1 downto 0);
    R0Src        : in     std_logic_vector(SH2_WORDSIZE - 1 downto 0);
    PCSrc        : in     std_logic_vector(SH2_WORDSIZE - 1 downto 0);
    GBRIn        : in     std_logic_vector(SH2_WORDSIZE - 1 downto 0);
    GBRWriteEn   : in     std_logic;
    Off4         : in     std_logic_vector(3 downto 0);
    Off8         : in     std_logic_vector(7 downto 0);
    BaseSel      : in     std_logic_vector(1 downto 0);
    IndexSel     : in     std_logic_vector(1 downto 0);
    OffScalarSel : in     std_logic_vector(1 downto 0);
    IncDecSel    : in     std_logic_vector(1 downto 0);
    Clk          : in     std_logic;

    Address      : out    std_logic_vector(SH2_WORDSIZE - 1 downto 0);
    AddrSrcOut   : buffer std_logic_vector(SH2_WORDSIZE - 1 downto 0);
    GBROut       : out    std_logic_vector(SH2_WORDSIZE - 1 downto 0)
  );
end SH2Dmau;

architecture structural of SH2Dmau is
```

```
199
200     -- ZeroExtend a std_logic_vector into an SH2_WORDSIZE std_logic_vector.
201     --
202     pure function ZeroExtend(slv : std_logic_vector) return std_logic_vector is
203       variable result : std_logic_vector(SH2_WORDSIZE - 1 downto 0);
204     begin
205       result := (others => '0');
206       result(slv'range) := slv;
207       return result;
208     end function;
209
210
211     -- shift_left is defined for unsigned/signed types only; wrap for slv.
212     --
213     pure function shift_left_slv(slv : std_logic_vector;
214                                  k   : natural) return std_logic_vector is
215     begin
216       return std_logic_vector(shift_left(unsigned(slv), k));
217     end function;
218
219     -- Global Base Register.
220     signal GBR : std_logic_vector(SH2_WORDSIZE - 1 downto 0);
221
222
223     -- MemUnit generic constants.
224     --
225     constant SRCCNT       : integer := 3;  -- Number of vectors in AddrSrc matrix.
226     constant OFFSETCNT    : integer := 4;  -- Number of vectors in AddrOff matrix.
227     constant MAXINCDECBIT : integer := 2;  -- The maximum index of the bit of
228                                            -- the calculted address that we want
229                                            -- to increment or decrement.
230
231     -- DMAUAddrSrc.
232     -- Consists of RegSrc, GBR, and PC. Note that there is a one-to-one mapping
233     -- between the BaseSel constants.
234     --
235     signal DMAUAddrSrc : std_logic_array(SRCCNT - 1 downto 0)(SH2_WORDSIZE - 1 downto 0);
236
237
238     -- DMAUSrcSel constants.
239     constant DMAUAddrSrc_REG : integer := 0;
240     constant DMAUAddrSrc_GBR : integer := 1;
241     constant DMAUAddrSrc_PC  : integer := 2;
242     signal   DMAUSrcSel      : integer  range SRCCNT - 1 downto 0;
243
244
245     -- DMAUOffsetSel constants.
246     constant DMAUOffsetSel_ZERO   : integer := 0;
247     constant DMAUOffsetSel_OFF4   : integer := 1;
248     constant DMAUOffsetSel_OFF8   : integer := 2;
249     constant DMAUOffsetSel_R0     : integer := 3;
250     signal   DMAUOffsetSel        : integer range OFFSETCNT - 1 downto 0;
251
252     signal DMAUAddrOff : std_logic_array(OFFSETCNT - 1 downto 0)(SH2_WORDSIZE - 1 downto 0);
253
254
255     constant  DMAU_INC       : std_logic := '0';
256     constant  DMAU_DEC       : std_logic := '1';
257     signal    DMAUIncDecSel  : std_logic;
258
259     -- The bit of the calculted address we want to increment or decrement.
260     --    + 1 = increment bit 0
261     --    * 2 = increment bit 1
262     --    * 4 = increment bit 2
263     signal DMAUIncDecBit  : integer range MAXINCDECBIT downto 0;
264
```

```
265     constant DMAU_PRE     : std_logic := '0';
266     constant DMAU_POST    : std_logic := '1';
267     signal DMAUPrePostSel : std_logic;
268
269     -- The low two bits of the PC are masked if PC is selected as the base address,
270     -- Off8 is selected as the index, and the scaled factor is 4.
271     constant  PC_MASK : std_logic_vector(SH2_WORDSIZE - 1 downto 0) := x"FFFFFFFC";
272     signal    PCMux   : std_logic_vector(SH2_WORDSIZE - 1 downto 0);
273
274
275     -- Zero/sign extedned offsets.
276     signal Off4ZeroExtended   : std_logic_vector(SH2_WORDSIZE - 1 downto 0);
277     signal Off8ZeroExtended   : std_logic_vector(SH2_WORDSIZE - 1 downto 0);
278
279 begin
280
281   WriteGBR : process(Clk)
282   begin
283     if rising_edge(Clk) then
284       if (GBRWriteEn = '1') then
285         GBR <= GBRIn;
286       end if;
287     end if;
288   end process WriteGBR;
289
290   GBROut <= GBR;
291
292   -- DMAUAddrSrc  ------------------------------------------------------------
293
294   -- The low two bits of the PC are masked if PC is selected as the base address,
295   -- Off8 is selected as the index, and the scaled factor is 4.
296   PCMux <= (PCSrc and PC_MASK) when (BaseSel = BaseSel_PC)      and
297                                     (IndexSel = IndexSel_OFF8)  and
298                                     (OffScalarSel = OffScalarSel_FOUR) else
299            PCSrc;
300
301   DMAUAddrSrc(DMAUAddrSrc_REG) <= RegSrc;
302   DMAUAddrSrc(DMAUAddrSrc_GBR) <= GBR;
303   DMAUAddrSrc(DMAUAddrSrc_PC)  <= std_logic_vector(unsigned(PCMux) + 4);
304
305
306   -- DMAUSrcSel --------------------------------------------------------------
307   DMAUSrcSel <= to_integer(unsigned(BaseSel));
308
309   -- DMAUAddrOff  ------------------------------------------------------------
310   Off4ZeroExtended  <= ZeroExtend(Off4);
311   Off8ZeroExtended  <= ZeroExtend(Off8);
312
313
314   -- Populate the DMAUAddrOff matrix.
315   DMAUAddrOff(DMAUOffsetSel_ZERO) <= (others => '0');
316
317   DMAUAddrOff(DMAUOffsetSel_OFF4) <=
318     shift_left_slv(Off4ZeroExtended, to_integer(unsigned(OffScalarSel)));
319
320   DMAUAddrOff(DMAUOffsetSel_OFF8)   <=
321       shift_left_slv(Off8ZeroExtended, to_integer(unsigned(OffScalarSel)));
322
323   DMAUAddrOff(DMAUOffsetSel_R0) <= R0Src;
324
325
326   -- DMAUOffsetSel  ----------------------------------------------------------
327   DMAUOffsetSel <=
328       DMAUOffsetSel_ZERO when (IndexSel = IndexSel_NONE) else
329
330       DMAUOffsetSel_OFF4 when (IndexSel = IndexSel_OFF4) else
```

```
331
332       DMAUOffsetSel_OFF8 when (IndexSel = IndexSel_OFF8) else
333
334       DMAUOffsetSel_R0 when (IndexSel = IndexSel_R0) else
335
336       DMAUOffsetSel;
337
338
339   -- DMAUIncDecSel ------------------------------------------------------------
340
341   with IncDecSel select DMAUIncDecSel <=
342     DMAU_INC      when  IncDecSel_POST_INC,
343     DMAU_DEC      when  IncDecSel_PRE_DEC,
344     '0'           when  others;
345
346   -- DMAUIncDecBit ------------------------------------------------------------
347
348   DMAUIncDecBit <= 0 when (OffScalarSel = OffScalarSel_ONE) and
349                           ((IncDecSel = IncDecSel_PRE_DEC) or
350                           (IncDecSel = IncDecSel_POST_INC)) else
351
352                    1 when (OffScalarSel = OffScalarSel_TWO) and
353                           ((IncDecSel = IncDecSel_PRE_DEC) or
354                           (IncDecSel = IncDecSel_POST_INC)) else
355
356                    2 when (OffScalarSel = OffScalarSel_FOUR) and
357                           ((IncDecSel = IncDecSel_PRE_DEC) or
358                           (IncDecSel = IncDecSel_POST_INC)) else
359                    0;
360
361   -- DMAUPrePostSel -----------------------------------------------------------
362
363   -- Note that we must pick between either PRE or POST inc/dec. If we
364   -- select PRE when we don't care, the output address will always be pre
365   -- incremented or decremented.
366   with IncDecSel select DMAUPrePostSel <=
367     DMAU_PRE       when IncDecSel_PRE_DEC,
368     DMAU_POST      when IncDecSel_POST_INC,
369     DMAU_POST      when others;
370
371
372   SH2Dmau_Instance : entity work.MemUnit
373     generic map (
374       srcCnt        => SRCCNT,
375       offsetCnt     => OFFSETCNT,
376       maxIncDecBit  => MAXINCDECBIT,
377       wordsize      => SH2_WORDSIZE
378     )
379     port map (
380       -- Inputs:
381       AddrSrc    => DMAUAddrSrc,
382       SrcSel     => DMAUSrcSel,
383       AddrOff    => DMAUAddrOff,
384       OffsetSel  => DMAUOffsetSel,
385       IncDecSel  => DMAUIncDecSel,
386       IncDecBit  => DMAUIncDecBit,
387       PrePostSel => DMAUPrePostSel,
388       -- Ouputs:
389       Address    => Address,
390       AddrSrcOut => AddrSrcOut
391     );
392
393 end structural;
394
395
```

```
1   --------------------------------------------------------------------------------
2   --  sh2_pmau.vhd
3   --
4   -- SH-2 PMAU (Program Memory Access Unit).
5   --
6   --  This is an implementation of the SH-2's PMAU using Glen A. George's generic
7   --  MAU (memory access unit). The purpose of the PMAU is to calculate program
8   --  memory addresses for branch instructions. The program counter is modified
9   --  in the following ways depending on the branch instruction used:
10  --
11  --    - PC <- PC + 2*disp:8  (relative)
12  --    - PC <- PC + 2*disp:12 (relative)
13  --    - PC <- PC + Rm        (register relative)
14  --    - PC <- PR             (PR direct)
15  --    - PC <- PC + 2         (increment)
16  --    - PC <- Rm             (register direct)
17  --
18  --
19  --  Revision History:
20  --    16 April 25   Chris M. Initial reivision.
21  --    01 May   25   Chris M. Added PRWriteEn and seperate offset signals. Made
22  --                           PrePostSel in MAU be POST when we don't care.
23  --    02 May   25   Chris M. Changed SignExtend function to wrap numeric_std
24  --                           conversion.
25  --    07 May   25   Chris M. Add reset signal and logic.
26  --    26 May   25   Chris M. Add 2 to 8-bit offset.
27  --    29 May   25   Chris M. Add PCWriteEn and PCIn. Add PCRegOut and PCCalcOut.
28  --
29  --------------------------------------------------------------------------------
30
31  library ieee;
32  library std;
33  library work;
34
35  use work.SH2Constants.all;
36  use ieee.std_logic_1164.all;
37
38  -- SH2Pmau
39  --
40  -- This is the SH-2s PMAU. It handles altering the PC according to the input
41  -- control signals. The ways in which the PC can change are
42  --
43  --    PC <- PC + 2*disp:8  (relative)
44  --    PC <- PC + 2*disp:12 (relative)
45  --    PC <- PC + Rm        (register relative)
46  --    PC <- PR             (PR direct)
47  --    PC <- PC + 2         (increment)
48  --    PC <- Rm             (register direct)
49  --
50  -- Note that the possible adresses sources to the general memory access unit
51  -- are only the zero vector or the PC. Thus, direct replacements of the PC
52  -- will be accomplished by adding the offset to the zero vector. Finally
53  -- note neither the PC nor the PR are stored within the program memory access
54  -- unit.
55  --
56  -- Inputs:
57  --   RegIn         : Register source input.
58  --   PRIn          : PR Register input (for writing to PR).
59  --   PRWriteEn     : Enable writing to PR (active high).
60  --   PCIn          : Input for parallel loading of PC.
61  --   PCWriteCtrl   : Write control signal for PC. Can either hold the current
62  --                   value, write PCIn to the PC register on the rising edge,
63  --                   or write the calculated PC value on the rising edge.
64  --   Off8          : 8-bit signed offset input.
65  --   Off12         : 12-bit signed offset input.
66  --   PCAddrMode    : Program address mode select signal.
```

```
67  --   Clk           : Clock.
68  --   Reset         : system reset (active low).
69  --
70  -- Outputs:
71  --   PCCalcOut     : Calculated PC address output. Note that this is not the
72  --                   value of the PR register.
73  --   PCRegOut      : Current value of the PC register.
74  --
75  --   PROut         : PR (Procedure Register) output.
76  --
77  entity SH2Pmau is
78    port (
79      RegIn         : in std_logic_vector(SH2_WORDSIZE - 1 downto 0);
80      PRIn          : in std_logic_vector(SH2_WORDSIZE - 1 downto 0);
81      PRWriteEn     : in std_logic;
82      PCIn          : in std_logic_vector(SH2_WORDSIZE - 1 downto 0);
83      PCWriteCtrl   : in std_logic_vector(1 downto 0);
84      Off8          : in std_logic_vector(7 downto 0);
85      Off12         : in std_logic_vector(11 downto 0);
86      PCAddrMode    : in std_logic_vector(2 downto 0);
87      Clk           : in std_logic;
88      Reset         : in std_logic;
89      PCCalcOut     : out std_logic_vector(31 downto 0);   -- TODO
90      PCRegOut      : out std_logic_vector(31 downto 0);   -- TODO
91      PROut         : out std_logic_vector(SH2_WORDSIZE - 1 downto 0)
92    );
93  end entity SH2Pmau;
94
95
96  library ieee;
97  use ieee.std_logic_1164.all;
98
99  package SH2PmauConstants is
100
101   constant PCAddrMode_INC                    : std_logic_vector(2 downto 0) := "000";  -- PC <- PC + 2
102   constant PCAddrMode_RELATIVE_8             : std_logic_vector(2 downto 0) := "001";  -- PC <- PC + disp:8
103   constant PCAddrMode_RELATIVE_12            : std_logic_vector(2 downto 0) := "010";  -- PC <- PC + disp:12
104   constant PCAddrMode_REG_DIRECT_RELATIVE    : std_logic_vector(2 downto 0) := "011";  -- PC <- PC + Rm
105   constant PCAddrMode_REG_DIRECT             : std_logic_vector(2 downto 0) := "100";  -- PC <- Rm
106   constant PCAddrMode_PR_DIRECT              : std_logic_vector(2 downto 0) := "101";  -- PC <- PR
107   constant PCAddrMode_HOLD                   : std_logic_vector(2 downto 0) := "110";  -- PC <- PC
108
109
110   constant PCWriteCtrl_HOLD       : std_logic_vector(1 downto 0) := "00";   -- Hold the current PC.
111   constant PCWriteCtrl_WRITE_IN   : std_logic_vector(1 downto 0) := "01";   -- Write PCIn to the PC reg.
112   constant PCWriteCtrl_WRITE_CALC : std_logic_vector(1 downto 0) := "10";   -- Write the calculated PC.
113
114
115 end package SH2PmauConstants;
116
117 library ieee;
118 library std;
119
120 use ieee.std_logic_1164.all;
121 use ieee.numeric_std.all;
122
123 use std.textio.all;
124
125 use work.SH2PmauConstants.all;
126 use work.SH2Constants.all;
127 use work.MemUnitConstants.all;
128 use work.array_type_pkg.all;
129
130 architecture structural of SH2Pmau is
131
132   -- SignExtend a std_logic_vector into an SH2_WORDSIZE std_logic_vector.
```

```
133   --
134   pure function SignExtend(slv : std_logic_vector) return std_logic_vector is
135     variable result : std_logic_vector(SH2_WORDSIZE - 1 downto 0);
136   begin
137     -- slv -> signed, resize to sign-extend, then convert to slv.
138     result := std_logic_vector(resize(signed(slv), SH2_WORDSIZE));
139     return result;
140   end function;
141
142   -- shift_left is defined for unsigned/signed types only; wrap for slv.
143   --
144   pure function shift_left_slv(slv : std_logic_vector;
145                                k   : natural) return std_logic_vector is
146   begin
147     return std_logic_vector(shift_left(unsigned(slv), k));
148   end function;
149
150   -- Possible sources are PC, PR, and Rm.
151   constant SRCCNT : integer := 3;
152
153   -- Possible offfsets are None, Off8, Off12, or Rm.
154   constant OFFSETCNT : integer := 4;
155
156   -- Adding two is the same as incrementing bit 1 of the PC.
157   constant MAXINCDECBIT : integer := 1;
158
159   constant PMAUAddrSrc_PC : integer := 0;
160   constant PMAUAddrSrc_PR : integer := 1;
161   constant PMAUAddrSrc_Rm : integer := 2;
162   signal PMAUAddrSrc : std_logic_array(SRCCNT - 1 downto 0)(SH2_WORDSIZE - 1 downto 0);
163   signal PMAUSrcSel  : integer range SRCCNT - 1 downto 0;
164
165   constant PMAUAddrOff_NONE   : integer := 0;
166   constant PMAUAddrOff_OFF8   : integer := 1;
167   constant PMAUAddrOff_OFF12  : integer := 2;
168   constant PMAUAddrOff_REG    : integer := 3;
169
170   signal PMAUAddrOff : std_logic_array(OFFSETCNT - 1 downto 0)(SH2_WORDSIZE - 1 downto 0);
171   signal PMAUOffsetSel : integer range OFFSETCNT - 1 downto 0;
172
173
174   -- constant MemUnit_INC  : std_logic := '0';            -- pre/post increment
175   signal PMAUIncDecSel  : std_logic;
176
177   -- MAXINCDECBIT := 1
178   signal PMAUIncDecBit  : integer range 0 to 1;
179
180   -- constant MemUnit_POST : std_logic := '1';            -- post- inc/dec
181   signal PMAUPrePostSel : std_logic;
182
183   signal CalculatedPC  : std_logic_vector(SH2_WORDSIZE - 1 downto 0);
184   signal IncrementedPC : std_logic_vector(SH2_WORDSIZE - 1 downto 0);
185
186   -- signal PC : std_logic_vector(SH2_WORDSIZE - 1 downto 0);
187   -- signal PR : std_logic_vector(SH2_WORDSIZE - 1 downto 0);
188
189   -- signal PCMux : std_logic_vector(SH2_WORDSIZE - 1 downto 0);
190
191   signal PRReg : std_logic_vector(SH2_WORDSIZE - 1 downto 0);
192
193   signal PCReg : std_logic_vector(SH2_WORDSIZE - 1 downto 0);
194   signal PCMux : std_logic_vector(SH2_WORDSIZE - 1 downto 0);
195
196   constant ZERO_32 : std_logic_vector(SH2_WORDSIZE - 1 downto 0) := (others => '0');
197
198 begin
```

```
199
200   PCCalcOut <= (ZERO_32) when (Reset = '0') else
201             PCMux;
202
203
204   PCRegOut <= PCReg;
205
206   -- PCOut <= (ZERO_32) when (Reset = '0') else
207   --          PCMux;
208
209   PROut <= (ZERO_32) when (Reset = '0') else
210            PRReg;
211
212   with PCAddrMode select
213       PCMux <= IncrementedPC    when  PCAddrMode_INC,
214                CalculatedPC     when  PCAddrMode_RELATIVE_8 | PCAddrMode_RELATIVE_12,
215                CalculatedPC     when  PCAddrMode_REG_DIRECT_RELATIVE,
216                CalculatedPC     when  PCAddrMode_REG_DIRECT,
217                PRReg            when  PCAddrMode_PR_DIRECT,
218                PCReg            when  PCAddrMode_HOLD,
219                (others => '0')  when  others;
220
221   UpdateRegisters : process(Clk, reset)
222   begin
223
224     if reset = '0' then
225       -- Reset PC and PR to all zeros.
226       PCReg <= (others => '0');
227       PRReg <= (others => '0');
228
229     elsif (rising_edge(Clk)) then
230
231       -- PCReg <= PCMux;
232
233       -- Only write to register if their enable signal is set.
234       if (PRWriteEn = '1') then
235         PRReg <= PRIn;
236       else
237         PRReg <= PRReg;
238       end if;
239
240       -- Choose what to do to the PC based on the PC write control signal.
241       PCReg <= PCMux;
242
243       case PCWriteCtrl is
244
245           when PCWriteCtrl_HOLD =>
246               PCReg <= PCReg;
247
248           when PCWriteCtrl_WRITE_IN =>
249               PCReg <= PCIn;
250
251          when PCWriteCtrl_WRITE_CALC =>
252              PCReg <= PCMux;
253
254           -- This has to be set intially otherwise instructions are not loaded.
255           -- TODO: Hold when an uncrecognized signal is sent.
256           when others =>
257               PCReg <= PCReg;
258
259       end case;
260
261     end if;
262
263   end process;
264
```

```
265
266   -- PMAUAddrSrc ------------------------------------------------------------
267
268   PMAUAddrSrc(PMAUAddrSrc_PC) <= (ZERO_32) when (Reset = '0') else
269                                  PCReg;
270
271   PMAUAddrSrc(PMAUAddrSrc_PR) <= (ZERO_32) when (Reset = '0') else
272                                  PRReg;
273
274   PMAUAddrSrc(PMAUAddrSrc_Rm) <= (ZERO_32) when (Reset = '0') else
275                                  RegIn;
276
277   -- PMAUSrcSel -------------------------------------------------------------
278
279   with PCAddrMode select PMAUSrcSel <=
280
281     PMAUAddrSrc_PC  when   PCAddrMode_INC | PCAddrMode_RELATIVE_8 | PCAddrMode_RELATIVE_12 |
282                            PCAddrMode_REG_DIRECT_RELATIVE,
283     PMAUAddrSrc_PR  when   PCAddrMode_PR_DIRECT,
284     PMAUAddrSrc_Rm  when   PCAddrMode_REG_DIRECT,
285     0               when   others;
286
287
288   -- PMAUAddrOff ------------------------------------------------------------
289
290   PMAUAddrOff(PMAUAddrOff_NONE)   <=   (others => '0');
291
292   -- 2 * SignExtend(Off8) (*2 is shift left by 1)
293
294   -- TODO: How the fuck does this fix it ????
295   --
296   PMAUAddrOff(PMAUAddrOff_OFF8)   <=   (ZERO_32) when (Reset = '0') else
297
298                                        -- TODO: ????
299                                        std_logic_vector(unsigned(shift_left_slv(SignExtend(Off8), 1)) + to_unsigned(4, 32));
300
301   -- 2 * SignExtend(Off12)
302
303   -- TODO: I guess this needs an offset too ???
304   PMAUAddrOff(PMAUAddrOff_OFF12)  <=   (ZERO_32) when (Reset = '0') else
305
306                                        std_logic_vector(unsigned(shift_left_slv(SignExtend(Off12), 1)) + to_unsigned(4, 32));
307
308   PMAUAddrOff(PMAUAddrOff_REG)    <=   (ZERO_32) when (Reset = '0') else
309                                        RegIn;
310
311   -- PMAUOffsetSel ----------------------------------------------------------
312
313   with PCAddrMode select PMAUOffsetSel <=
314
315     PMAUAddrOff_NONE   when   PCAddrMode_REG_DIRECT | PCAddrMode_PR_DIRECT | PCAddrMode_INC,
316     PMAUAddrOff_OFF8   when   PCAddrMode_RELATIVE_8,
317     PMAUAddrOff_OFF12  when   PCAddrMode_RELATIVE_12,
318     PMAUAddrOff_REG    when   PCAddrMode_REG_DIRECT_RELATIVE,
319     0                  when    others;
320
321   -- PMAUIncDecSel ----------------------------------------------------------
322   PMAUIncDecSel <= MemUnit_INC;
323
324   -- PMAUIncDecBit ----------------------------------------------------------
325   PMAUIncDecBit <= 1;
326
327   -- PMAUPrePostSel ---------------------------------------------------------
328   PMAUPrePostSel <= MemUnit_POST;
329
330   SH2Pmau_Instance : entity work.MemUnit
```

```
331     generic map (
332       srcCnt        => SRCCNT,
333       offsetCnt     => OFFSETCNT,
334       maxIncDecBit  => MAXINCDECBIT,
335       wordsize      => SH2_WORDSIZE
336     )
337     port map (
338       -- Inputs:
339       AddrSrc    => PMAUAddrSrc,
340       SrcSel     => PMAUSrcSel,
341       AddrOff    => PMAUAddrOff,
342       OffsetSel  => PMAUOffsetSel,
343       IncDecSel  => PMAUIncDecSel,
344       IncDecBit  => PMAUIncDecBit,
345       PrePostSel => PMAUPrePostSel,
346       -- Ouputs:
347       Address    => CalculatedPC,
348       AddrSrcOut => IncrementedPC
349     );
350 end structural;
351
352
353
354
```

```
1   --------------------------------------------------------------------------------
2   -- SH2Regs
3   --
4   -- This entity implements the general-purpose registers for the SH-2 CPU. The
5   -- SH-2 contains 16 registers, numbered R0 through R15, each 32 bits wide.
6   -- These registers can be used for both ALU instructions and memory addressing.
7   -- To allow the CPU to perform ALU operations and memory accesses in parallel,
8   -- two dual-port memory interfaces are provided. Each interface supports
9   -- writing to one register and reading from two registers in a single clock.
10  -- Writing to the same register through both of these interfaces should be
11  -- avoided, but if it occurs, then the "regular" interface will take
12  -- precendence over the memory addressing interface.
13  --
14  --------------------------------------------------------------------------------
15
16  -- import libraries
17  library ieee;
18  use ieee.std_logic_1164.all;
19  use ieee.numeric_std.all;
20
21  entity SH2Regs is
22      port (
23          RegDataIn  : in   std_logic_vector(31 downto 0);    -- data to write to a register
24          EnableIn   : in   std_logic;                        -- if data should be written to an input register
25          RegInSel   : in   integer  range 15 downto 0;       -- which register to write data to
26          RegASel    : in   integer  range 15 downto 0;       -- which register to read to bus A
27          RegBSel    : in   integer  range 15 downto 0;       -- which register to read to bus B
28          RegAxIn    : in   std_logic_vector(31 downto 0);    -- data to write to an address register
29          RegAxInSel : in   integer  range 15 downto 0;       -- which address register to write to
30          RegAxStore : in   std_logic;                        -- if data should be written to the address register
31          RegA1Sel   : in   integer  range 15 downto 0;       -- which register to read to address bus 1
32          RegA2Sel   : in   integer  range 15 downto 0;       -- which register to read to address bus 2
33          clock      : in   std_logic;                        -- system clock
34          reset      : in   std_logic;                        -- system reset (async, active low)
35          RegA       : out  std_logic_vector(31 downto 0);    -- register bus A
36          RegB       : out  std_logic_vector(31 downto 0);    -- register bus B
37          RegA1      : out  std_logic_vector(31 downto 0);    -- address register bus 1
38          RegA2      : out  std_logic_vector(31 downto 0)     -- address register bus 2
39      );
40  end SH2Regs;
41
42  architecture structural of SH2Regs is
43
44      component  RegArray  is
45
46          generic (
47              regcnt   : integer := 32;    -- default number of registers is 32
48              wordsize : integer := 8      -- default width is 8-bits
49          );
50
51          port(
52              RegIn      : in   std_logic_vector(wordsize - 1 downto 0);      -- input bus to the registers
53              RegInSel   : in   integer  range regcnt - 1 downto 0;           -- which register to write (log regcnt bits)
54              RegStore   : in   std_logic;                                    -- actually write to a register
55              RegASel    : in   integer  range regcnt - 1 downto 0;           -- register to read onto bus A (log regcnt bits)
56              RegBSel    : in   integer  range regcnt - 1 downto 0;           -- register to read onto bus B (log regcnt bits)
57              RegAxIn    : in   std_logic_vector(wordsize - 1 downto 0);      -- input bus for address register updates
58              RegAxInSel : in   integer  range regcnt - 1 downto 0;           -- which address register to write (log regcnt bits - 1)
59              RegAxStore : in   std_logic;                                    -- actually write to an address register
60              RegA1Sel   : in   integer  range regcnt - 1 downto 0;           -- register to read onto address bus 1 (log regcnt bits)
61              RegA2Sel   : in   integer  range regcnt - 1 downto 0;           -- register to read onto address bus 2 (log regcnt bits)
62              RegDIn     : in   std_logic_vector(2 * wordsize - 1 downto 0);  -- input bus to the double-width registers
63              RegDInSel  : in   integer  range regcnt/2 - 1 downto 0;         -- which double register to write (log regcnt bits - 1)
64              RegDStore  : in   std_logic;                                    -- actually write to a double register
65              RegDSel    : in   integer  range regcnt/2 - 1 downto 0;         -- register to read onto double width bus D (log regcnt bi
66              clock      : in   std_logic;                                    -- the system clock
```

```
67              reset      : in   std_logic;                                    -- system reset (async, active low)
68              RegA       : out  std_logic_vector(wordsize - 1 downto 0);      -- register value for bus A
69              RegB       : out  std_logic_vector(wordsize - 1 downto 0);      -- register value for bus B
70              RegA1      : out  std_logic_vector(wordsize - 1 downto 0);      -- register value for address bus 1
71              RegA2      : out  std_logic_vector(wordsize - 1 downto 0);      -- register value for address bus 2
72              RegD       : out  std_logic_vector(2 * wordsize - 1 downto 0)   -- register value for bus D (double width bus)
73          );
74
75      end  component;
76  begin
77
78      -- Specialize the provided generic register array entity for the SH-2 CPU.
79      -- We will use the memory interfaces exactly as implemented, passing the
80      -- bits through directly. We will use RegIn, RegInSel, RegA, and RegB as
81      -- the "normal" memory interface used for reading ALU operands and writing
82      -- ALU results. We will use RegAXIn, RegAxSel, RegA1, and RegA2 for
83      -- register accesses dedicated to memory addressing. We must be able to
84      -- read up to two registers to get indirect indexed register addressing (R0
85      -- + Rn), and we should also be able to update address register values to
86      -- implement pre/post increment/decrement.
87      Registers: RegArray
88      generic map (
89          wordsize => 32,
90          regcnt => 16
91      )
92      port map(
93          clock => clock,
94          reset => reset,
95          -- dual register access for ALU operations
96          RegIn => RegDataIn,
97          RegInSel => RegInSel,
98          RegStore => EnableIn,
99          RegASel => RegASel,
100         RegBSel => RegBSel,
101         RegA => RegA,
102         RegB => RegB,
103         -- dual register access for indirect/relative memory access
104         RegAxIn => RegAxIn,
105         RegAxInSel => RegAxInSel,
106         RegAxStore => RegAxStore,
107         RegA1Sel => RegA1Sel,
108         RegA2Sel => RegA2Sel,
109         RegA1 => RegA1,
110         RegA2 => RegA2,
111         -- unused
112         RegDIn => (others => '0'),
113         RegDInSel => 0,
114         RegDStore => '0',
115         RegDSel => 0
116     );
117 end structural;
118
```

```
1   ----------------------------------------------------------------------------
2   --
3   --  Memory Interface
4   --
5   --  These entities act as interfaces between the SH-2 CPU and the memory unit
6   --  being used to simulate SRAM (memory.vhd). These entities take control
7   --  signals generated by the control unit and use them to read/write to the
8   --  memory, performing byte shifting and conversions between little-endian and
9   --  big-endian as necessary. Note that the SH-2 CPU treats memory as big-endian,
10  --  though our internal implementation uses little-endian representation.
11  --  However, this interface should handle all conversions, and so the rest of
12  --  the CPU should not core about endianness at all. To avoid complications
13  --  with the data bus being inout, this memory interface is split into
14  --  MemoryInterfaceTx, which outputs signals to initiate a read/write, and
15  --  MemoryInterfaceRx, which reads back data returned from the data bus. If
16  --  using this interface, the data bus should not be used elsewhere, as these
17  --  entities provide the complete interface for reading/writing memory.
18  --
19  --  Revision History:
20  --     03 May 25  Zack Huang        Implement memory interface for byte,
21  --                                  word, and longword read/writes.
22  --     16 May 25  Zack Huang        Add documentation
23  --
24  ----------------------------------------------------------------------------
25
26  library ieee;
27
28  use ieee.std_logic_1164.all;
29  use ieee.numeric_std.all;
30
31  package MemoryInterfaceConstants is
32
33      -- MemEnable constants
34      constant MemEnable_OFF  : std_logic := '0';     -- disble memory reading/writing
35      constant MemEnable_ON   : std_logic := '1';     -- enable memory reading/writing
36
37      -- MemMode constants (BIT-DECODED, DO NOT CHANGE VALUES)
38      constant ByteMode       : std_logic_vector(1 downto 0) := "00"; -- read/write a single byte
39      constant WordMode       : std_logic_vector(1 downto 0) := "01"; -- read/write two bytes as a word
40      constant LongwordMode   : std_logic_vector(1 downto 0) := "10"; -- read/write four bytes as a longword
41
42      -- ReadWrite constants
43      constant Mem_READ       : std_logic := '0';     -- read from memory
44      constant Mem_WRITE      : std_logic := '1';     -- write to memory
45
46  end package MemoryInterfaceConstants;
47
48
49  library ieee;
50  library std;
51
52  use std.textio.all;
53
54  use ieee.std_logic_1164.all;
55  use ieee.numeric_std.all;
56
57  use work.MemoryInterfaceConstants.all;
58  use work.Logging.all;
59
60  -- MemoryInterfaceTx
61  --
62  -- This entity implements the "output" portion of the memory interface, in that
63  -- it takes in control signals from the CPU and outputs the necessary
64  -- read-enables/write-enables and values on the data bus in order to perform
65  -- a read/write. Control signals allow for this unit to be enabled/disabled,
66  -- and to use different memory access modes (byte, word, longword). Though this
```

```
67  -- entity does not output to the CPU address bus, it must know the address in
68  -- order to mask the right enable bits for reading/writing. Note that when
69  -- performing a write, the MemDataIn input is truncated to the low 8 bits in
70  -- byte mode and the low 16 bits for word mode.
71  --
72  entity MemoryInterfaceTx is
73      port (
74          clock     :  in     std_logic;                          -- system clock
75          MemEnable :  in     std_logic;                          -- if memory interface should be active or not
76          ReadWrite :  in     std_logic;                          -- memory read (0) or write (1)
77          MemMode   :  in     std_logic_vector(1 downto 0);       -- memory access mode (byte, word, or longword)
78          Address   :  in     unsigned(31 downto 0);              -- memory address bus (MUST BE ALIGNED!)
79          MemDataOut   :  in     std_logic_vector(31 downto 0);      -- the input to write to memory
80          RE        :  out    std_logic_vector(3 downto 0);       -- read enable mask (active low)
81          WE        :  out    std_logic_vector(3 downto 0);       -- write enable mask (active low)
82          DB        :  out    std_logic_vector(31 downto 0)       -- memory data bus
83      );
84  end entity;
85
86
87  architecture structural of MemoryInterfaceTx is
88  begin
89
90      output_proc: process(MemEnable, ReadWrite, MemMode, Address, MemDataOut, clock)
91        variable l : line;
92      begin
93          -- When clock goes low, if this interface is enabled and should perform
94          -- either a read or a write, then output the read-enable and
95          -- write-enable signals in order depending on the memory mode to
96          -- read/write a byte, word, or longword.
97          if MemEnable = MemEnable_ON and clock = '0' and not is_x(address) then
98              if ReadWrite = Mem_READ then    -- If performing a read
99                  WE(3 downto 0) <= (others => '1');  -- Disable writing
100                 DB <= (others => 'Z');              -- set data bus to high impedance so it can be read from
101
102                 -- Enable specific bytes based on type of read
103                 case MemMode is
104                     when ByteMode =>
105                         -- Enable only the specific byte being read
106                         RE(0) <= '0' when address mod 4 = 0 else '1';
107                         RE(1) <= '0' when address mod 4 = 1 else '1';
108                         RE(2) <= '0' when address mod 4 = 2 else '1';
109                         RE(3) <= '0' when address mod 4 = 3 else '1';
110
111                         LogWithTime(l, "memory_interface.vhd: Reading byte at address 0x" & to_hstring(address), LogFile);
112
113                     when WordMode =>
114                         assert (address mod 2 = 0)
115                         report "Memory interface Tx: Cannot read word from non-aligned address: " & to_hstring(address)
116                         severity error;
117
118                         -- Enable only the specific pair of bytes being read (address must be word-aligned)
119                         RE(0) <= '0' when address mod 4 = 0 else '1';
120                         RE(1) <= '0' when address mod 4 = 0 else '1';
121                         RE(2) <= '0' when address mod 4 = 2 else '1';
122                         RE(3) <= '0' when address mod 4 = 2 else '1';
123
124                         LogWithTime(l, "memory_interface.vhd: Reading word at address 0x" & to_hstring(address), LogFile);
125
126                     when LongwordMode =>
127                         assert (address mod 4 = 0)
128                         report "Memory interface Tx: Cannot read longword from non-aligned address: " & to_hstring(address)
129                         severity error;
130
131                         -- Enable all bytes to read a longword. Address must be longword-aligned.
132                         RE(3 downto 0) <= (others => '0');
```

```
133
134                     LogWithTime(l, "memory_interface.vhd: Reading longword at address 0x" & to_hstring(address), LogFile);
135
136                 when others =>
137                     assert (false)
138                     report "Memory interface Tx: unrecognized read mode" & to_hstring(address)
139                     severity error;
140
141                     -- When unrecognized mode, don't read/write anything
142                     RE <= (others => '1');
143
144             end case;
145
146         elsif ReadWrite = Mem_WRITE then        -- If performing a write:
147             RE(3 downto 0) <= (others => '1');  -- Disable reading
148             DB <= (others => 'Z');              -- Set data bus bytes to high impedance by default
149
150             -- Enable specific bytes based on memory mode
151             case MemMode is
152                 when ByteMode =>
153                     -- Enable only the specific byte being written
154                     WE(0) <= '0' when address mod 4 = 0 else '1';
155                     WE(1) <= '0' when address mod 4 = 1 else '1';
156                     WE(2) <= '0' when address mod 4 = 2 else '1';
157                     WE(3) <= '0' when address mod 4 = 3 else '1';
158
159                     -- Set the correct data bus byte to the byte being written
160                     if address mod 4 = 0 then
161                         DB(7 downto 0)   <= MemDataOut(7 downto 0);
162                     elsif address mod 4 = 1 then
163                         DB(15 downto 8)  <= MemDataOut(7 downto 0);
164                     elsif address mod 4 = 2 then
165                         DB(23 downto 16) <= MemDataOut(7 downto 0);
166                     elsif address mod 4 = 3 then
167                         DB(31 downto 24) <= MemDataOut(7 downto 0);
168                     end if;
169
170                     LogWithTime(l, "memory_interface.vhd: Writing byte (0x" & to_hstring(MemDataOut(7 downto 0)) &
171                                    ") at address 0x" & to_hstring(address), LogFile);
172
173                 when WordMode =>
174                     assert (address mod 2 = 0)
175                     report "Memory interface Tx: Cannot write word to non-aligned address: " & to_hstring(address)
176                     severity error;
177
178                     -- Enable only the specific pair of bytes being read (address must be word-aligned)
179                     WE(0) <= '0' when address mod 4 = 0 else '1';
180                     WE(1) <= '0' when address mod 4 = 0 else '1';
181                     WE(2) <= '0' when address mod 4 = 2 else '1';
182                     WE(3) <= '0' when address mod 4 = 2 else '1';
183
184                     if address mod 4 = 0 then
185                         -- Convert input data from little-endian to big-endian by reversing the bytes,
186                         -- the output to the low word of the data bus
187                         DB(15 downto 0) <= MemDataOut(7 downto 0) & MemDataOut(15 downto 8);
188
189                         LogWithTime(l, "memory_interface.vhd: Writing word (0x" &
190                                        to_hstring(MemDataOut(7 downto 0) & MemDataOut(15 downto 8)), LogFile);
191                     elsif address mod 2 = 0 then
192                         -- Convert input data from little-endian to big-endian by reversing the bytes,
193                         -- the output to the high word of the data bus
194                         DB(31 downto 16) <= MemDataOut(7 downto 0) & MemDataOut(15 downto 8);
195
196                         LogWithTime(l, "memory_interface.vhd: Writing word (0x" &
197                                        to_hstring(MemDataOut(7 downto 0) & MemDataOut(15 downto 8)), LogFile);
198                     end if;
```

```
199
200                         LogWithTime(") at address 0x" & to_hstring(address), LogFile);
201
202                     when LongwordMode =>
203                         assert (address mod 4 = 0)
204                         report "Memory interface Tx: Cannot write longword to non-aligned address: " & to_hstring(address)
205                         severity error;
206
207                         -- Enable all bytes to read a longword. Address must be longword-aligned.
208                         WE(3 downto 0) <= (others => '0');
209
210                         -- Reverse bytes to convert little-endian to big-endian
211                         DB(7 downto 0) <= MemDataOut(31 downto 24);
212                         DB(15 downto 8) <= MemDataOut(23 downto 16);
213                         DB(23 downto 16) <= MemDataOut(15 downto 8);
214                         DB(31 downto 24) <= MemDataOut(7 downto 0);
215
216                         LogWithTime(l, "memory_interface.vhd: Writing longword (0x" & to_hstring(MemDataOut) &
217                                        ") at address 0x" & to_hstring(address), LogFile);
218
219                     when others =>
220                         assert (false)
221                         report "Memory interface Tx: Invalid memory mode for write"
222                         severity error;
223
224                         -- When unrecognized mode, don't read/write anything
225                         WE(3 downto 0) <= (others => '1');
226
227                 end case;
228             end if;
229         else
230             -- should not enable memory interface
231             RE <= (others => '1');
232             WE <= (others => '1');
233         end if;
234
235     end process output_proc;
236 end architecture;
237
238
239 library ieee;
240 use ieee.std_logic_1164.all;
241 use ieee.numeric_std.all;
242
243 use work.MemoryInterfaceConstants.all;
244
245 -- MemoryInterfaceRx
246 --
247 -- This entity implements the "input" portion of the memory interface, in that
248 -- it reads bits returned from the data bus after a read is performed and
249 -- converts it from big-endian back into little-endian, which is used
250 -- internally. The memory mode and address of the read that was performed must
251 -- be provided so this entity knows how to shift the bytes into the correct
252 -- positions. All inputs are sign-extended to 32 bits in accordance with the
253 -- SH-2 spec.
254 --
255 entity MemoryInterfaceRx is
256     port (
257         MemEnable :  in     std_logic;                      -- if memory interface should be active or not
258         MemMode   :  in     std_logic_vector(1 downto 0);   -- memory access mode (byte, word, or longword)
259         Address   :  in     unsigned(31 downto 0);          -- memory address bus
260         DB        :  in     std_logic_vector(31 downto 0);  -- memory data bus
261         MemDataIn :  out    std_logic_vector(31 downto 0)   -- data read from memory
262     );
263 end entity;
264
```

```
265 architecture structural of MemoryInterfaceRx is
266 begin
267     output_proc: process(MemEnable, MemMode, Address, DB)
268     begin
269         if MemEnable = MemEnable_ON then
270             -- Shift and sign-extend based on the mode
271             case MemMode is
272                 when ByteMode =>
273                     -- Mask out the correct byte from the data bus and sign-extend
274                     if (Address mod 4 = 0) then
275                         MemDataIn(7 downto 0) <= DB(7 downto 0);
276                         MemDataIn(31 downto 8) <= (others => DB(7));
277                     elsif (Address mod 4 = 1) then
278                         MemDataIn(7 downto 0) <= DB(15 downto 8);
279                         MemDataIn(31 downto 8) <= (others => DB(15));
280                     elsif (Address mod 4 = 2) then
281                         MemDataIn(7 downto 0) <= DB(23 downto 16);
282                         MemDataIn(31 downto 8) <= (others => DB(23));
283                     elsif (Address mod 4 = 3) then
284                         MemDataIn(7 downto 0) <= DB(31 downto 24);
285                         MemDataIn(31 downto 8) <= (others => DB(31));
286                     end if;
287 
288                 when WordMode =>
289                     -- Mask out the correct pair of byte from the data bus, reverse
290                     -- their order to convert them from big-endian to little-endian,
291                     -- and then sign-extend them
292                     if (Address mod 4 = 0) then
293                         MemDataIn(7 downto 0)  <= DB(15 downto 8);
294                         MemDataIn(15 downto 8) <= DB(7 downto 0);
295                         MemDataIn(31 downto 16) <= (others => DB(15));
296                     elsif (Address mod 4 = 2) then
297                         MemDataIn(7 downto 0) <= DB(31 downto 24);
298                         MemDataIn(15 downto 8) <= DB(23 downto 16);
299                         MemDataIn(31 downto 16) <= (others => DB(31));
300                     end if;
301 
302                 when LongwordMode =>
303                     -- Reverse the read bytes from the data bus to convert them
304                     -- from big-endian to little-endian. No sign-extension is
305                     -- necessary.
306                     MemDataIn(7 downto 0) <= DB(31 downto 24);
307                     MemDataIn(15 downto 8) <= DB(23 downto 16);
308                     MemDataIn(23 downto 16) <= DB(15 downto 8);
309                     MemDataIn(31 downto 24) <= DB(7 downto 0);
310 
311                 when others =>
312                     assert (false)
313                     report "Memory interface Rx: Invalid memory mode"
314                     severity error;
315 
316                     -- When invalid memory mode, don't read anything
317                     MemDataIn <= (others => 'X');
318             end case;
319         else
320             -- Interface not enabled, don't read anything
321             MemDataIn <= (others => 'X');
322         end if;
323 
324     end process output_proc;
325 
326 end architecture;
327 
```

```
1    ----------------------------------------------------------------------------------------------
2    --                                                                                          -
3    --  Control Unit                                                                            -
4    --                                                                                          -
5    --                                                                                          -
6    --  Revision History:                                                                       -
7    --     06 May 25  Zack Huang        Initial revision                                        -
8    --     07 May 25  Chris Miranda     Initial implentation of MOV and branch instruction decoding.   -
9    --     10 May 25  Zack Huang        Implementing ALU instruction                            -
10   --     14 May 25  Chris M.          Formatting.                                             -
11   --     16 May 25  Zack Huang        Documentation, renaming signals                         -
12   --     25 May 25  Zack Huang        Finishing ALU and system instructions                   -
13   --     26 May 25  Chris M.          Add T flag as input to control unit. Add delay slot simulation -
14   --                                  signals.                                                -
15   --                                                                                          -
16   --     29 May 25  Chris May         Add PCWriteCtrl and DelayedBranchTaken signals to control unit -
17   --                                  output.                                                 -
18   --                                                                                          -
19   -- Notes:                                                                                   -
20   --  - When reading/writing to registers, RegB is always Rm and RegA is always Rn           -
21   --  - When reading/writing to addresses (in registers), RegA2 is always @(Rm) and           -
22   --    RegA2 is always @(Rn).                                                                -
23   --                                                                                          -
24   -- TODO:                                                                                    -
25   --  - Remove redundant assignment of default signals.                                       -
26   --  - Better names for:                                                                     -
27   --      Instruction_RegEnableIn, RegEnableIn                                                -
28   --                                                                                          -
29   --  - Generate DMAU signals with vectors.                                                   -
30   --  - Document register output conventions.                                                 -
31   --  - Document bit decoding.                                                                -
32   --  - Add short instruction operation to std_match case.                                    -
33   --  - Use slv_to_uint more.                                                                 -
34   --  - DEAL WITH DOUBLE DELAYED BRANCHES - NOT POSSIBLE                                      -
35   ----------------------------------------------------------------------------------------------
36
37
38   library ieee;
39   library std;
40
41   use std.textio.all;
42
43   use ieee.std_logic_1164.all;
44   use ieee.numeric_std.all;
45   use work.MemoryInterfaceConstants.all;
46   use work.Logging.all;
47
48   package SH2InstructionEncodings is
49
50     subtype Instruction is std_logic_vector(15 downto 0);
51
52
53     -- Instruction encodings.
54
55     -- Data Transfer Instructions:
56     constant  MOV_IMM_RN            :  Instruction := "1110------------";  -- MOV #imm, Rn
57
58     constant  MOV_AT_DISP_PC_RN     :  Instruction := "1-01------------";  -- MOV.X @(disp, PC), Rn (for bit decoding.)
59     constant  MOV_W_AT_DISP_PC_RN   :  Instruction := "1001------------";  -- MOV.W @(disp, PC), Rn
60     constant  MOV_L_AT_DISP_PC_RN   :  Instruction := "1101------------";  -- MOV.L @(disp, PC), Rn
61
62     constant  MOV_RM_RN             :  Instruction := "0110--------0011";  -- MOV Rm, Rn
63
64     constant  MOV_RM_AT_RN          :  Instruction := "0010--------00--";  -- MOV.X Rm, @Rn (for bit decoding).
65     constant  MOV_B_RM_AT_RN        :  Instruction := "0010--------0000";  -- MOV.B Rm, @Rn
66     constant  MOV_W_RM_AT_RN        :  Instruction := "0010--------0001";  -- MOV.W Rm, @Rn
```

```
67      constant  MOV_L_RM_AT_RN        :  Instruction := "0010--------0010";  -- MOV.L Rm, @Rn
68
69      constant  MOV_AT_RM_RN          :  Instruction := "0110--------00--";  -- MOV.X @Rm, Rn (for bit decoding).
70      constant  MOV_B_AT_RM_RN        :  Instruction := "0110--------0000";  -- MOV.B @Rm, Rn
71      constant  MOV_W_AT_RM_RN        :  Instruction := "0110--------0001";  -- MOV.W @Rm, Rn
72      constant  MOV_L_AT_RM_RN        :  Instruction := "0110--------0010";  -- MOV.L @Rm, Rn
73
74      constant  MOV_RM_AT_MINUS_RN    :  Instruction := "0010--------01--";  -- MOV.X Rm, @-Rn (for bit decoding).
75      constant  MOV_B_RM_AT_MINUS_RN  :  Instruction := "0010--------0100";  -- MOV.B Rm, @-Rn
76      constant  MOV_W_RM_AT_MINUS_RN  :  Instruction := "0010--------0101";  -- MOV.W Rm, @-Rn
77      constant  MOV_L_RM_AT_MINUS_RN  :  Instruction := "0010--------0110";  -- MOV.L Rm, @-Rn
78
79      constant  MOV_AT_RM_PLUS_RN     :  Instruction := "0110--------01--";  -- MOV.X @Rm+, Rn (for bit decoding)
80      constant  MOV_B_AT_RM_PLUS_RN   :  Instruction := "0110--------0100";  -- MOV.B @Rm+, Rn
81      constant  MOV_W_AT_RM_PLUS_RN   :  Instruction := "0110--------0101";  -- MOV.W @Rm+, Rn
82      constant  MOV_L_AT_RM_PLUS_RN   :  Instruction := "0110--------0110";  -- MOV.W @Rm+, Rn
83
84      constant  MOV_R0_AT_DISP_RN     :  Instruction := "1000000---------";  -- MOV.{B,W} RO, @(disp,Rn)
85      constant  MOV_B_R0_AT_DISP_RN   :  Instruction := "10000000--------";  -- MOV.B RO, @(disp,Rn)
86      constant  MOV_W_R0_AT_DISP_RN   :  Instruction := "10000001--------";  -- MOV.W RO, @(disp,Rn)
87
88      constant  MOV_L_RM_AT_DISP_RN   :  Instruction := "0001------------";  -- MOV.L Rm, @(disp, Rn)
89
90      constant  MOV_AT_DISP_RM_R0     :  Instruction := "1000010---------";  -- MOV.{B,W} @(disp, Rm), R0
91      constant  MOV_B_AT_DISP_RM_R0   :  Instruction := "10000100--------";  -- MOV.B @(disp, Rm), R0
92      constant  MOV_W_AT_DISP_RM_R0   :  Instruction := "10000101--------";  -- MOV.W @(disp, Rm), R0
93
94      constant  MOV_L_AT_DISP_RM_RN   :  Instruction := "0101------------";  -- MOV.L @(disp, Rm), Rn
95
96      constant  MOV_RM_AT_R0_RN       :  Instruction := "0000--------01--";  -- MOV.X Rm, @(R0, Rn)
97      constant  MOV_B_RM_AT_R0_RN     :  Instruction := "0000--------0100";  -- MOV.B Rm, @(R0, Rn)
98      constant  MOV_W_RM_AT_R0_RN     :  Instruction := "0000--------0101";  -- MOV.W Rm, @(R0, Rn)
99      constant  MOV_L_RM_AT_R0_RN     :  Instruction := "0000--------0110";  -- MOV.L Rm, @(R0, Rn)
100
101     constant  MOV_AT_R0_RM_RN       :  Instruction := "0000--------11--";  -- MOV.X @(R0, Rm), Rn
102     constant  MOV_B_AT_R0_RM_RN     :  Instruction := "0000--------1100";  -- MOV.B @(R0, Rm), Rn
103     constant  MOV_W_AT_R0_RM_RN     :  Instruction := "0000--------1101";  -- MOV.W @(R0, Rm), Rn
104     constant  MOV_L_AT_R0_RM_RN     :  Instruction := "0000--------1110";  -- MOV.L @(R0, Rm), Rn
105
106     constant  MOV_R0_AT_DISP_GBR    :  Instruction := "110000----------";  -- MOV.X R0, @(disp, GBR)
107     constant  MOV_B_R0_AT_DISP_GBR  :  Instruction := "11000000--------";  -- MOV.B R0, @(disp, GBR)
108     constant  MOV_W_R0_AT_DISP_GBR  :  Instruction := "11000001--------";  -- MOV.W R0, @(disp, GBR)
109     constant  MOV_L_R0_AT_DISP_GBR  :  Instruction := "11000010--------";  -- MOV.L R0, @(disp, GBR)
110
111     constant  MOV_AT_DISP_GBR_R0    :  Instruction := "110001----------";  -- MOV.X @(disp, GBR), R0
112     constant  MOV_B_AT_DISP_GBR_R0  :  Instruction := "11000100--------";  -- MOV.B @(disp, GBR), R0
113     constant  MOV_W_AT_DISP_GBR_R0  :  Instruction := "11000101--------";  -- MOV.W @(disp, GBR), R0
114     constant  MOV_L_AT_DISP_GBR_R0  :  Instruction := "11000110--------";  -- MOV.L @(disp, GBR), R0
115
116     constant  MOVA_AT_DISP_PC_R0    :  Instruction := "11000111--------";  -- MOVA @(disp, PC), R0
117
118     constant  MOVT_RN               :  Instruction := "0000----00101001";  -- MOVT Rn
119
120     constant  SWAP_RM_RN            :  Instruction := "0110--------100-";  -- SWAP.{B,W} Rm, Rn
121     constant  SWAP_B_RM_RN          :  Instruction := "0110--------1000";  -- SWAP.B Rm, Rn
122     constant  SWAP_W_RM_RN          :  Instruction := "0110--------1001";  -- SWAP.W Rm, Rn
123
124     constant  XTRCT_RM_RN           :  Instruction := "0010--------1101";  -- XTRCT Rm, Rn
125
126
127
128     -- Arithmetic Instructions:
129     constant ADD_RM_RN     : Instruction := "0011--------11--";
130     constant ADD_IMM_RN    : Instruction := "0111------------";
131     constant SUB_RM_RN     : Instruction := "0011--------10--";
132     constant NEG_RM_RN     : Instruction := "0110--------101-";
```

```
133     constant DT_RN         : Instruction := "0100----00010000";
134     constant EXT_RM_RN     : Instruction := "0110--------11--";
135
136     constant CMP_EQ_IMM    : Instruction := "10001000--------";   -- CMP/EQ #imm, R0
137     constant CMP_RM_RN     : Instruction := "0011--------0---";   -- CMP/{EQ,HS,GE,HI,GT} Rm, Rn
138     constant CMP_RN        : Instruction := "0100----00010-01";   -- CMP/{PL/PZ}
139     constant CMP_STR_RM_RN : Instruction := "0010--------1100";   -- CMP/STR
140
141     -- Logical Operations:
142     constant LOGIC_RM_RN   : Instruction := "0010--------10--";  -- AND, TST, OR, XOR
143     constant LOGIC_IMM_R0  : Instruction := "110010----------";  -- AND, TST, OR, XOR
144     constant NOT_RM_RN     : Instruction := "0110--------0111";  -- NOT
145
146     -- Shift Instruction:
147     constant SHIFT_RN      : Instruction := "0100----00-00-0-";  -- shift/rotate instructions
148     constant BSHIFT_RN     : Instruction := "0100----00--100-";  -- barrel shifter instructions
149
150     -- Branch Instructions:
151     constant BF     :   Instruction := "10001011--------"; -- BF      <label>
152     constant BF_S   :   Instruction := "10001111--------"; -- BF/S    <label>
153     constant BT     :   Instruction := "10001001--------"; -- BT      <label>
154     constant BT_S   :   Instruction := "10001101--------"; -- BT/S    <label>
155     constant BRA    :   Instruction := "1010------------"; -- BRA     <label>
156     constant BRAF   :   Instruction := "0000----00100011"; -- BRAF    Rm
157     constant BSR    :   Instruction := "1011------------"; -- BSR     <label>
158     constant BSRF   :   Instruction := "0000----00000011"; -- BSRF    Rm
159     constant JMP    :   Instruction := "0100----00101011"; -- JMP     @Rm
160     constant JSR    :   Instruction := "0100----00001011"; -- JSR     @Rm
161     constant RTS    :   Instruction := "0000000000001011"; -- RTS
162
163
164     -- System Control:
165     constant NOP      : Instruction := "0000000000001001";
166     constant CLRT     : Instruction := "0000000000001000";
167     constant CLRMAC   : Instruction := "0000000000101000";
168     constant SETT     : Instruction := "0000000000011000";
169
170     constant STC_SYS_RN             : Instruction := "0000----00--0010";  -- STC {SR, GBR, VBR}, Rn
171     constant STC_SR_RN              : Instruction := "0000----00000010";  -- STC SR,  Rn
172     constant STC_GBR_RN             : Instruction := "0000----00010010";  -- STC GBR, Rn
173     constant STC_VBR_RN             : Instruction := "0000----00100010";  -- STC VBR, Rn
174
175     constant STC_L_SYS_RN           : Instruction := "0100----00--0011";  -- STC.L {SR, GBR, VBR}, @-Rn
176     constant STC_L_SR_AT_MINUS_RN   : Instruction := "0100----00000011";  -- STC.L SR, @-Rn
177     constant STC_L_GBR_AT_MINUS_RN  : Instruction := "0100----00010011";  -- STC.L GBR, @-Rn
178     constant STC_L_VBR_AT_MINUS_RN  : Instruction := "0100----00100011";  -- STC.L VBR, @-Rn
179
180     constant LDC_RM_SYS             : Instruction := "0100----00--1110";  -- LDC Rm, {SR, GBR, VBR}
181     constant LDC_RM_SR              : Instruction := "0100----00001110";  -- LDC Rm, SR
182     constant LDC_RM_GBR             : Instruction := "0100----00011110";  -- LDC Rm, GBR
183     constant LDC_RM_VBR             : Instruction := "0100----00101110";  -- LDC Rm, VBR
184
185     constant LDC_L_RM_SYS           : Instruction := "0100----00--0111";  -- LDC.L @Rm+, {SR, GBR, VBR}
186     constant LDC_L_AT_RM_PLUS_SR    : Instruction := "0100----00000111";  -- LDC.L @Rm+, SR
187     constant LDC_L_AT_RM_PLUS_GBR   : Instruction := "0100----00010111";  -- LDC.L @Rm+, GBR
188     constant LDC_L_AT_RM_PLUS_VBR   : Instruction := "0100----00100111";  -- LDC.L @Rm+, VBR
189
190     constant LDS_RM_SYS             : Instruction := "0100----00--1010";  -- LDS Rm, {MACH, MACL, PR}
191     constant LDS_RM_MACH            : Instruction := "0100----00001010";  -- LDS Rm, MACH
192     constant LDS_RM_MACL            : Instruction := "0100----00011010";  -- LDS Rm, MACL
193     constant LDS_RM_PR              : Instruction := "0100----00101010";  -- LDS Rm, PR
194
195     constant LDS_L_RM_SYS           : Instruction := "0100----00--0110";  -- LDS.L @Rm+, {MACH, MACL, PR}
196     constant LDS_L_AT_RM_PLUS_MACH  : Instruction := "0100----00000110";  -- LDS.L @Rm+, MACH
197     constant LDS_L_AT_RM_PLUS_MACL  : Instruction := "0100----00010110";  -- LDS.L @Rm+, MACL
198     constant LDS_L_AT_RM_PLUS_PR    : Instruction := "0100----00100110";  -- LDS.L @Rm+, PR
```

```
199
200    constant STS_SYS_RN             : Instruction := "0000----00--1010";  -- STS Rm, {MACH, MACL, PR}
201    constant STS_MACH_RN            : Instruction := "0000----00001010";  -- STS Rm, MACH
202    constant STS_MACL_RN            : Instruction := "0000----00011010";  -- STS Rm, MACL
203    constant STS_PR_RN              : Instruction := "0000----00101010";  -- STS Rm, PR
204
205    constant STS_L_SYS_RN           : Instruction := "0100----00--0010";  -- STS.L @Rm+, {MACH, MACL, PR}
206    constant STS_L_AT_RM_PLUS_MACH  : Instruction := "0100----00000010";  -- STS.L @Rm+, MACH
207    constant STS_L_AT_RM_PLUS_MACL  : Instruction := "0100----00010010";  -- STS.L @Rm+, MACL
208    constant STS_L_AT_RM_PLUS_PR    : Instruction := "0100----00100010";  -- STS.L @Rm+, PR
209
210
211  end package SH2InstructionEncodings;
212
213
214  library ieee;
215  use ieee.std_logic_1164.all;
216  use ieee.numeric_std.all;
217
218  package SH2ControlConstants is
219
220      -- Internal control signals for controlling muxes within the CPU
221
222      -- Constants for RegDataInSel in sh2_cpu architecture. Used to determine what to input to the
223      -- register array's RegDataIn.
224      --
225      constant RegDataIn_ALUResult        : std_logic_vector(3 downto 0) := "0000"; -- ALU result
226      constant RegDataIn_Immediate        : std_logic_vector(3 downto 0) := "0001"; -- Sign-extended 8-bit immediate
227      constant RegDataIn_RegA             : std_logic_vector(3 downto 0) := "0010"; -- RegA output
228      constant RegDataIn_RegB             : std_logic_vector(3 downto 0) := "0011"; -- RegB output
229      constant RegDataIn_SysReg           : std_logic_vector(3 downto 0) := "0100"; -- System register
230      constant RegDataIn_RegA_SWAP_B      : std_logic_vector(3 downto 0) := "0111"; -- RegA output with low two bytes swapped
231      constant RegDataIn_RegA_SWAP_W      : std_logic_vector(3 downto 0) := "1000"; -- RegA output with low/high words swapped
232      constant RegDataIn_REGB_REGA_CENTER : std_logic_vector(3 downto 0) := "1001"; -- Low word of RegB and high word of RegA
233      constant RegDataIn_SR_TBit          : std_logic_vector(3 downto 0) := "1010"; -- T-bit (in the LSB)
234      constant RegDataIn_PR               : std_logic_vector(3 downto 0) := "1011"; -- Procedure Register (PR)
235      constant RegDataIn_DB               : std_logic_vector(3 downto 0) := "1100"; -- Data Bus (DB) value
236      constant RegDataIn_Ext              : std_logic_vector(3 downto 0) := "1101"; -- Sign/zero extended register values.
237
238      -- Constants for ExtMode in sh2_cpu architecture. Used to determine how RegB will be extended
239      -- to a long-word.
240      --
241      -- WARNING: Changing these will break bit decoding of instructions.
242      --
243      constant Ext_Sign_B_RegA : std_logic_vector(1 downto 0) := "10";  -- Sign extend low byte of Reg A
244      constant Ext_Sign_W_RegA : std_logic_vector(1 downto 0) := "11";  -- Sign extend low word of Reg A
245      constant Ext_Zero_B_RegA : std_logic_vector(1 downto 0) := "00";  -- Zero extend low byte of Reg A
246      constant Ext_Zero_W_RegA : std_logic_vector(1 downto 0) := "01";  -- Zero extend low word of Reg A
247
248
249      -- Constants for ReadWrite used in memory_tx architecture. Determines whether to read or
250      -- write to memory.
251      --
252      constant ReadWrite_READ     : std_logic := '0';
253      constant ReadWrite_WRITE    : std_logic := '1';
254
255      -- Constants for selecting what to output to MemDataOut in memory_tx entity in the sh2_cpu
256      -- entity.
257      --
258      constant MemOut_RegA    : std_logic_vector(2 downto 0) := "000"; -- Output RegA to data bus
259      constant MemOut_RegB    : std_logic_vector(2 downto 0) := "001"; -- Output RegB to data bus
260      constant MemOut_SysReg  : std_logic_vector(2 downto 0) := "010"; -- Output a system register to data bus
261
262
263      constant ALUOpB_RegB    : std_logic := '0'; -- Use RegB as B input to ALU.
264      constant ALUOpB_Imm     : std_logic := '1'; -- Use immediate as B input to ALU
```

```
265
266       constant TFlagSel_T         : std_logic_vector(2 downto 0) := "000";    -- Have T retain its value
267       constant TFlagSel_Zero      : std_logic_vector(2 downto 0) := "001";    -- Set T to the ALU zero flag
268       constant TFlagSel_Carry     : std_logic_vector(2 downto 0) := "010";    -- Set T to the ALU carry flag
269       constant TFlagSel_Overflow  : std_logic_vector(2 downto 0) := "011";    -- Set T to the ALU overflow flag
270       constant TFlagSel_SET       : std_logic_vector(2 downto 0) := "100";    -- clear T (to 0)
271       constant TFlagSel_CLEAR     : std_logic_vector(2 downto 0) := "101";    -- set T (to 1)
272       constant TFlagSel_CMP       : std_logic_vector(2 downto 0) := "110";    -- set T to a value computed from
273                                                                               -- the ALU flags
274
275       -- WARNING: Changing these will break bit decoding of instuctions.
276
277       -- How to calculate the T-bit in the
278       constant TCMP_EQ            : std_logic_vector(2 downto 0) := "000"; --
279       constant TCMP_HS            : std_logic_vector(2 downto 0) := "010";
280       constant TCMP_GE            : std_logic_vector(2 downto 0) := "011";
281       constant TCMP_HI            : std_logic_vector(2 downto 0) := "110";
282       constant TCMP_GT            : std_logic_vector(2 downto 0) := "111";
283       constant TCMP_STR           : std_logic_vector(2 downto 0) := "100";
284
285       constant MemSel_ROM         : std_logic := '1';
286       constant MemSel_RAM         : std_logic := '0';
287
288       constant MemAddrSel_PMAU    : std_logic := '0';
289       constant MemAddrSel_DMAU    : std_logic := '1';
290
291       constant SysRegCtrl_NONE    : std_logic_vector(1 downto 0) := "00";     -- do nothing with system register
292       constant SysRegCtrl_LOAD    : std_logic_vector(1 downto 0) := "01";     -- load system register with new value
293       constant SysRegCtrl_CLEAR   : std_logic_vector(1 downto 0) := "10";     -- clear system register
294
295       constant SysRegSrc_RegB     : std_logic_vector(1 downto 0) := "00";     -- load system register from register bus B
296       constant SysRegSrc_DB       : std_logic_vector(1 downto 0) := "01";     -- load system register from data bus
297       constant SysRegSrc_PC       : std_logic_vector(1 downto 0) := "10";     -- load system register from PC
298
299       -- WARNING: Changing these will break bit decoding of instuctions.
300       constant SysRegSel_System   : std_logic_vector(2 downto 0) := "0--";
301       constant SysRegSel_SR       : std_logic_vector(2 downto 0) := "000";
302       constant SysRegSel_GBR      : std_logic_vector(2 downto 0) := "001";
303       constant SysRegSel_VBR      : std_logic_vector(2 downto 0) := "010";
304       constant SysRegSel_Control  : std_logic_vector(2 downto 0) := "1--";
305       constant SysRegSel_MACH     : std_logic_vector(2 downto 0) := "100";
306       constant SysRegSel_MACL     : std_logic_vector(2 downto 0) := "101";
307       constant SysRegSel_PR       : std_logic_vector(2 downto 0) := "110";
308
309       -- Whether to sign or zero extend the immediate into a 32-bit word.
310       constant ImmediateMode_SIGN     : std_logic := '0';
311       constant ImmediateMode_ZERO     : std_logic := '1';
312
313       -- None -> Slot -> Target -> None
314       -- constant DelayedBR_NONE   : std_logic_vector(1 downto 0);
315       -- constant DelayedBR_SLOT   : std_logic_vector(1 downto 0);
316       -- constant DelayedBR_TARGET : std_logic_vector(1 downto 0);
317
318   end package SH2ControlConstants;
319
320
321   library ieee;
322   library std;
323
324   use ieee.std_logic_1164.all;
325   use ieee.numeric_std.all;
326
327   use std.textio.all;
328
329   use work.SH2PmauConstants.all;
330   use work.SH2DmauConstants.all;
```

```
331  use work.MemoryInterfaceConstants.all;
332  use work.SH2InstructionEncodings.all;
333  use work.SH2ControlConstants.all;
334  use work.SH2ALUConstants.all;
335  use work.Logging.all;
336  use work.Utils.all;
337
338  entity  SH2Control  is
339
340      port (
341          MemDataIn   : in  std_logic_vector(31 downto 0);    -- data read from memory
342          TFlagIn     : in  std_logic;                        -- T Flag input from top level CPU.
343          clock       : in  std_logic;                        -- system clock
344          reset       : in  std_logic;                        -- system reset (active low, async)
345
346          -- control signals to control memory interface
347          MemEnable   : out std_logic;                        -- if memory needs to be accessed (read or write)
348          MemAddrSel  : out std_logic;
349          ReadWrite   : out std_logic;                        -- if should do memory read (0) or write (1)
350          MemMode     : out std_logic_vector(1 downto 0);     -- if memory access should be by byte, word, or longword
351          MemSel      : out std_logic;                        -- select memory address source, from DMAU output (0) or PMAU output (1)
352
353          Immediate   : out std_logic_vector(7 downto 0);     -- 8-bit immediate
354          ImmediateMode   : out std_logic;                    -- Immediate extension mode
355          MemOutSel   : out std_logic_vector(2 downto 0);     -- what should be output to memory
356          TFlagSel    : out std_logic_vector(2 downto 0);     -- source for next value of T flag
357          ExtMode     : out std_logic_vector(1 downto 0);     -- mode for extending register value (zero or signed)
358
359          -- ALU control signals
360          ALUOpBSel   : out std_logic;                        -- input mux to Operand B, either RegB (0) or Immediate (1)
361          LoadA       : out std_logic;                        -- determine if OperandA is loaded ('1') or zeroed ('0')
362          FCmd        : out std_logic_vector(3 downto 0);     -- F-Block operation
363          CinCmd      : out std_logic_vector(1 downto 0);     -- carry in operation
364          SCmd        : out std_logic_vector(2 downto 0);     -- shift operation
365          ALUCmd      : out std_logic_vector(1 downto 0);     -- ALU result select
366          TCmpSel     : out std_logic_vector(2 downto 0);     -- how to compute T from ALU status flags
367
368          -- register array control signals
369          RegDataInSel: out std_logic_vector(3 downto 0);     -- source for register input data
370          RegEnableIn : out std_logic;                        -- if data should be written to an input register
371          RegInSel    : out integer  range 15 downto 0;       -- which register to write data to
372          RegASel     : out integer  range 15 downto 0;       -- which register to read to bus A
373          RegBSel     : out integer  range 15 downto 0;       -- which register to read to bus B
374          RegAxIn     : out std_logic_vector(31 downto 0);    -- data to write to an address register
375          RegAxInSel  : out integer  range 15 downto 0;       -- which address register to write to
376          RegAxStore  : out std_logic;                        -- if data should be written to the address register
377          RegA1Sel    : out integer  range 15 downto 0;       -- which register to read to address bus 1
378          RegA2Sel    : out integer  range 15 downto 0;       -- which register to read to address bus 2
379
380          -- DMAU signals
381          GBRWriteEn      : out std_logic;
382          DMAUOff4        : out std_logic_vector(3 downto 0);
383          DMAUOff8        : out std_logic_vector(7 downto 0);
384          BaseSel         : out std_logic_vector(1 downto 0);
385          IndexSel        : out std_logic_vector(1 downto 0);
386          OffScalarSel    : out std_logic_vector(1 downto 0);
387          IncDecSel       : out std_logic_vector(1 downto 0);
388
389          -- PMAU signals
390          PCAddrMode      : out std_logic_vector(2 downto 0);   -- What PC addressing mode is desired.
391          PRWriteEn       : out std_logic;                      -- Enable writing to PR.
392          PMAUOff8        : out std_logic_vector(7 downto 0);   -- 8-bit offset for relative addressing.
393          PMAUOff12       : out std_logic_vector(11 downto 0);  -- 12-bit offset for relative addressing.
394          PCIn            : out std_logic_vector(31 downto 0);  -- PC input for parallel loading.
395          PCWriteCtrl     : out std_logic_vector(1 downto 0);   -- What to write to the PC register inside
396                                                                -- the PMAU. Can either hold current value,
```

```
397                                                             -- write PCIn, or write calculated PC.
398
399          -- System control signals
400          SysRegCtrl      : out std_logic_vector(1 downto 0);
401          SysRegSel       : out std_logic_vector(2 downto 0);
402          SysRegSrc       : out std_logic_vector(1 downto 0);
403
404          -- Branch control signals:
405          DelayedBranchTaken : out std_logic  -- Whether the delayed branch is taken or not.
406  );
407
408  end  SH2Control;
409
410  architecture dataflow of sh2control is
411
412      -- TODO: Comment FSM and timing.
413      type state_t is (
414          fetch,
415          execute,
416          writeback
417      );
418
419      signal state : state_t;
420
421
422    -- The instruction register.
423    signal IR : std_logic_vector(15 downto 0);
424
425    -- Aliases for instruction arguments.
426    -- There are 13 instruction formats, shown below:
427    --
428    -- Key:
429    --  xxxx: instruction code
430    --  mmmm: Source register
431    --  nnnn: Destination register
432    --  iiii: immediate data
433    --  dddd: displacment
434
435    -- 0 format:   xxxx xxxx xxxx xxxx
436    -- n format:   xxxx nnnn xxxx xxxx
437    -- m format:   xxxx mmmm xxxx xxxx
438    -- nm format:  xxxx nnnn mmmm xxxx
439    -- md format:  xxxx xxxx mmmm dddd
440    -- nd4 format: xxxx xxxx nnnn dddd
441    -- nmd format: xxxx nnnn mmmm dddd
442    -- d format:   xxxx xxxx dddd dddd
443    -- d12 format: xxxx dddd dddd dddd
444    -- nd8 format: xxxx nnnn dddd dddd
445    -- i format:   xxxx xxxx iiii iiii
446    -- ni format:  xxxx nnnn iiii iiii
447
448    -- n format
449    alias n_format_n : std_logic_vector(3 downto 0) is IR(11 downto 8);
450
451    -- m format
452    alias m_format_m : std_logic_vector(3 downto 0) is IR(11 downto 8);
453
454    -- nm format
455    alias nm_format_n : std_logic_vector(3 downto 0) is IR(11 downto 8);
456    alias nm_format_m : std_logic_vector(3 downto 0) is IR(7 downto 4);
457
458    -- md format
459    alias md_format_m : std_logic_vector(3 downto 0) is IR(7 downto 4);
460    alias md_format_d : std_logic_vector(3 downto 0) is IR(3 downto 0);
461
462    -- nd4 format
```

```
463     alias nd4_format_n : std_logic_vector(3 downto 0) is IR(7 downto 4);
464     alias nd4_format_d : std_logic_vector(3 downto 0) is IR(3 downto 0);
465
466     -- nmd format
467     alias nmd_format_n : std_logic_vector(3 downto 0) is IR(11 downto 8);
468     alias nmd_format_m : std_logic_vector(3 downto 0) is IR(7 downto 4);
469     alias nmd_format_d : std_logic_vector(3 downto 0) is IR(3 downto 0);
470
471     -- d format
472     alias d_format_d : std_logic_vector(7 downto 0) is IR(7 downto 0);
473
474     -- d12 format
475     alias d12_format_d : std_logic_vector(11 downto 0) is IR(11 downto 0);
476
477     -- nd8 format
478     alias nd8_format_n : std_logic_vector(3 downto 0) is IR(11 downto 8);
479     alias nd8_format_d : std_logic_vector(7 downto 0) is IR(7 downto 0);
480
481     -- i format
482     alias i_format_i : std_logic_vector(7 downto 0) is IR(7 downto 0);
483
484     -- ni format
485     alias ni_format_n : std_logic_vector(3 downto 0) is IR(11 downto 8);
486     alias ni_format_i : std_logic_vector(7 downto 0) is IR(7 downto 0);
487
488     -- Internal signals computed combinatorially to memory signals can
489     -- be output on the correct clock.
490     signal Instruction_MemEnable   : std_logic;
491     signal Instruction_ReadWrite   : std_logic;
492     signal Instruction_MemSel      : std_logic;
493     signal Instruction_MemAddrSel  : std_logic;
494
495     -- The memory mode for a given instruction. The same as the constants in the
496     -- MemoryInterfaceConstants package.
497     signal Instruction_WordMode : std_logic_vector(1 downto 0);
498
499     -- Register write enable for the current instruction. Output to RegisterArray
500     -- during the execute state so that it is high when the rising clock of the writeback state occurs.
501     signal Instruction_RegEnableIn  : std_logic;
502
503     -- Address register write enable for the current instruction. Output to RegisterArray
504     -- during the execute state so that it is high when the rising clock of the writeback state occurs.
505     signal Instruction_RegAxStore : std_logic;
506
507     -- Program addressing mode for the current instruction. Output during the
508     -- writeback state so that it is ready by the following fetch state.
509     signal Instruction_PCAddrMode : std_logic_vector(2 downto 0);
510
511     -- What to write to the TFlag for the current instruction. Output during
512     -- the execute state so that it is high when the rising clock of the writeback state occurs.
513     signal Instruction_TFlagSel    : std_logic_vector(2 downto 0);
514
515     -- If the system register should be loaded or not
516     signal Instruction_SysRegCtrl  : std_logic_vector(1 downto 0);
517
518     -- If the GBR register should be updated or not. Output during execute state so it is
519     -- high for the rising clock edge of writeback.
520     signal Instruction_GBRWriteEn  : std_logic;
521
522     -- If the PR register should be updated or not. Output during execute state so it is
523     -- high for the rising clock edge of writeback.
524     signal Instruction_PRWriteEn  : std_logic;
525
526     -- Signal to simulate delay slot.
527     signal Instruction_DelaySlotEn : std_logic;
528
```

```
529    -- If a delayed branch will be taken or not. If this is true ('1'), then
530    -- we are currently executing branch slot instruction of a delayed branch,
531    -- and the next PC will be calculated using the saved signals below.
532    signal Instruction_DelayedBranchTaken : std_logic;
533
534  begin
535
536      -- Is this valid ???
537      DelayedBranchTaken <= Instruction_DelayedBranchTaken;
538
539      -- Outputs that change based on the CPU state
540      with state select
541          PCAddrMode <= Instruction_PCAddrMode when writeback,  -- increment PC during writeback state
542                        PCAddrMode_HOLD        when others;     -- otherwise, hold PC
543
544      with state select
545          MemEnable <= Instruction_MemEnable when execute,        -- if instruction requires memory access
546                       MemEnable_ON           when fetch,        -- enable to fetch instruction
547                       MemEnable_OFF          when writeback;    -- no memory access during writeback
548
549      with state select
550          ReadWrite <= Instruction_ReadWrite when execute,        -- if instruction does read/write
551                       Mem_READ              when fetch,         -- read instruction during fetch
552                       'X'                   when writeback;     -- no memory access during writeback
553
554      with state select
555          MemMode <= Instruction_WordMode when execute,       -- instruction memory mode
556                     WordMode             when fetch,         -- fetch instruction word
557                     (others => 'X')      when others;        -- no memory access during writeback
558
559      with state select
560          MemSel <= Instruction_MemSel when execute,      -- if instruction access RAM or ROM
561                    MemSel_ROM         when fetch,        -- access ROM to fetch instruction
562                    'X'                when others;       -- no memory access during writeback
563
564      with state select
565          MemAddrSel <= Instruction_MemAddrSel when execute,  -- if instruction accesses PMAU or DMAU address
566                        MemAddrSel_PMAU        when fetch,    -- access program memory during fetch
567                        'X'                    when others;   -- no memory access during writeback
568
569      with state select
570          GBRWriteEn <= Instruction_GBRWriteEn when execute,      -- if instruction updates GBR
571                        '0'                    when others;       -- don't change GBR
572
573      with state select
574          PRWriteEn  <= Instruction_PRWriteEn  when execute,      -- if instruction updates GBR
575                        '0'                    when others;       -- don't change GBR
576
577      -- Only modify registers after execute clock
578      RegEnableIn <= Instruction_RegEnableIn when state = execute else '0';
579
580      -- Only modify address registers after execute clock
581      RegAxStore <= Instruction_RegAxStore when state = execute else '0';
582
583      -- Only modify T flag bit after execute clock
584      TFlagSel <= Instruction_TFlagSel when state = execute else TFlagSel_T;
585
586      -- Only update system register after execute clock (on writeback)
587      SysRegCtrl <= Instruction_SysRegCtrl when state = execute else SysRegCtrl_NONE;
588
589      decode_proc: process (state)
590        variable l : line;
591      begin
592
593          if (state = execute) then
594
```

```
595         -- Default flag values are set here (these shouldn't change CPU state).
596         -- This is so that not every control signal has to be set in every single
597         -- instruction case. If an instruction enables writing to memory/registers,
598         -- then ensure that the default value is set here as "disable" to prevent
599         -- writes on the clocks following an instruction.
600
601         -- Not accessing memory
602         Instruction_MemEnable  <= '0';
603         Instruction_ReadWrite  <= 'X';
604         Instruction_WordMode   <= "XX";
605         MemOutSel              <= "XXX";
606         Instruction_MemSel     <= MemSel_RAM;       -- access data memory by default. TODO: change name
607         Instruction_MemAddrSel <= MemAddrSel_DMAU;  -- access data memory by default. TODO: change name
608
609         -- Register enables
610         Instruction_RegEnableIn <= '0';             -- Disable register write
611         Instruction_RegAxStore  <= '0';             -- Disable writing to address register.
612         Instruction_TFlagSel    <= TFlagSel_T;      -- Keep T flag the same
613         Instruction_GBRWriteEn  <= '0';             -- Don't write to GBR.
614         Instruction_PRWriteEn   <= '0';             -- Don't write to PR.
615
616         -- If a delayed branch was taken previously, then don't change the PC since the target
617         -- address was calculated when the delayed branch decoded.
618         if (DelayedBranchTaken = '1') then
619             Instruction_PCAddrMode <= PCAddrMode_HOLD;
620         else
621             -- Increment the PC.
622             Instruction_PCAddrMode <= PCAddrMode_INC;
623             -- LogWithTime("Changing PC to PCAddrMode_INC");
624         end if;
625
626         Instruction_SysRegCtrl <= SysRegCtrl_NONE;      -- system register not selected
627         ImmediateMode          <= ImmediateMode_SIGN;   -- sign-extend immediates by defualt
628         ExtMode                <= Ext_Sign_B_RegA;
629
630         PCWriteCtrl <= PCWriteCtrl_WRITE_CALC;  -- Write the calculated PC by default.
631
632         Instruction_DelayedBranchTaken <= '0'; -- The delayed branch taken flag is set to not
633                                                -- taken by default.
634
635         if std_match(IR, ADD_RM_RN) then
636             -- ADD{C,V} Rm, Rn
637
638             LogWithTime(l, "sh2_control.vhd: Decoded Add R" & to_string(to_integer(unsigned(nm_format_m))) &
639                            " , R" & to_string(to_integer(unsigned(nm_format_n))), LogFile);
640
641             -- Register array signals
642             RegASel <= to_integer(unsigned(nm_format_n));
643             RegBSel <= to_integer(unsigned(nm_format_m));
644
645             RegInSel             <= to_integer(unsigned(nm_format_n));
646             RegDataInSel         <= RegDataIn_ALUResult;
647             Instruction_RegEnableIn <= '1';
648
649             -- Bit-decoding T flag select (None, Carry, Overflow)
650             Instruction_TFlagSel <= '0' & IR(1 downto 0);
651
652             -- ALU signals for addition
653             ALUOpBSel <= ALUOpB_RegB;
654             LoadA     <= '1';
655             FCmd      <= FCmd_B;
656
657             -- Bit-decode carry in value
658             CinCmd <= CinCmd_CIN when IR(1 downto 0) = "10" else    -- ADDC
659                       CinCmd_ZERO;                                  -- ADD, ADDV
660
```

```
661             SCmd   <= "XXX";
662             ALUCmd <= ALUCmd_ADDER;
663
664
665         elsif std_match(IR, SUB_RM_RN) then
666             -- SUB{C,V} Rm, Rn
667
668             -- Register array signals
669             RegASel <= to_integer(unsigned(nm_format_n));
670             RegBSel <= to_integer(unsigned(nm_format_m));
671
672             RegInSel             <= to_integer(unsigned(nm_format_n));
673             RegDataInSel         <= RegDataIn_ALUResult;
674             Instruction_RegEnableIn <= '1';
675
676             -- Bit-decoding T flag select (None, Carry, Overflow)
677             Instruction_TFlagSel <= '0' & IR(1 downto 0);
678
679             -- ALU signals for subtraction
680             ALUOpBSel <= ALUOpB_RegB;
681             LoadA     <= '1';
682             FCmd      <= FCmd_BNOT;
683
684             -- Bit-decode carry in value
685             CinCmd <= CinCmd_CINBAR when IR(1 downto 0) = "10" else     -- SUBC
686                       CinCmd_ONE;                                       -- SUB, SUBV
687
688             SCmd   <= "XXX";
689             ALUCmd <= ALUCmd_ADDER;
690
691         elsif std_match(IR, DT_RN) then
692             -- DT Rn
693
694             -- Register array signals
695             RegASel <= to_integer(unsigned(nm_format_n));
696             Immediate <= (others => '0');
697
698             RegInSel             <= to_integer(unsigned(nm_format_n));
699             RegDataInSel         <= RegDataIn_ALUResult;
700             Instruction_RegEnableIn <= '1';
701
702             -- Bit-decoding T flag select (None, Carry, Overflow)
703             Instruction_TFlagSel <= TFlagSel_Zero;
704
705             -- ALU signals to subtract 1 from Rn
706             ALUOpBSel <= ALUOpB_Imm;
707             LoadA     <= '1';
708             FCmd      <= FCmd_BNOT;
709             CinCmd    <= CinCmd_ZERO;
710             SCmd      <= "XXX";
711             ALUCmd    <= ALUCmd_ADDER;
712
713         elsif std_match(IR, NEG_RM_RN) then
714             -- NEG{C} Rm, Rn
715
716             -- Register array signals
717             RegASel <= to_integer(unsigned(nm_format_n));
718             RegBSel <= to_integer(unsigned(nm_format_m));
719
720             RegInSel             <= to_integer(unsigned(nm_format_n));
721             RegDataInSel         <= RegDataIn_ALUResult;
722             Instruction_RegEnableIn <= '1';
723
724             -- Bit-decoding T flag select
725             Instruction_TFlagSel <= TFlagSel_Carry when IR(0) = '0' else    -- NEGC
726                                     TFlagSel_T;                             -- NEG
```

```
727
728             -- ALU signals for negation
729             ALUOpBSel <= ALUOpB_RegB;
730             LoadA     <= '0';
731             FCmd      <= FCmd_BNOT;
732
733             -- Bit-decode carry in value
734             CinCmd <= CinCmd_CINBAR when IR(0) = '0' else   -- NEGC
735                       CinCmd_ONE;                           -- NEG
736
737             SCmd   <= "XXX";
738             ALUCmd <= ALUCmd_ADDER;
739
740         elsif std_match(IR, EXT_RM_RN) then
741             -- EXT{U,S}.{B,W Rm, Rn}
742
743             -- Register array signals
744             RegASel <= to_integer(unsigned(nm_format_n));
745             RegBSel <= to_integer(unsigned(nm_format_m));
746
747             RegInSel             <= to_integer(unsigned(nm_format_n));
748             RegDataInSel         <= RegDataIn_Ext;
749             ExtMode              <= IR(1 downto 0);     -- bit-decode extension mode
750             Instruction_RegEnableIn <= '1';
751
752         elsif std_match(IR, ADD_IMM_RN) then
753             -- ADD #imm, Rn
754
755             -- Register array signals
756             RegASel <= to_integer(unsigned(nm_format_n));
757
758             RegInSel             <= to_integer(unsigned(nm_format_n));
759             RegDataInSel         <= RegDataIn_ALUResult;
760             Instruction_RegEnableIn <= '1';
761             Immediate            <= ni_format_i;
762
763             -- ALU signals for addition
764             ALUOpBSel <= ALUOpB_Imm;
765             LoadA     <= '1';
766             FCmd      <= FCmd_B;
767             CinCmd    <= CinCmd_ZERO;
768             SCmd      <= "XXX";
769             ALUCmd    <= ALUCmd_ADDER;
770
771         elsif std_match(IR, LOGIC_RM_RN) then
772             -- {AND, TST, OR, XOR} Rm, Rn
773
774             -- Register array signals
775             RegASel <= to_integer(unsigned(nm_format_n));
776             RegBSel <= to_integer(unsigned(nm_format_m));
777
778             RegInSel             <= to_integer(unsigned(nm_format_n));
779             RegDataInSel         <= RegDataIn_ALUResult;
780             Instruction_RegEnableIn <= IR(1) or IR(0);   -- exclude TST
781
782             -- Enable TFlagSel for TST
783             Instruction_TFlagSel <= TFlagSel_Zero when IR(1 downto 0) = "00"    -- TST
784                                     else TFlagSel_T;                            -- AND, OR, XOR
785
786             -- ALU signals for logic instructions using the FBlock
787             ALUOpBSel <= ALUOpB_RegB;
788             LoadA     <= '1';
789
790             -- Bit-decode f-block operation
791             FCmd <= FCmd_AND when IR(1) = '0'           else    -- AND, TST
792                     FCmd_XOR when IR(1 downto 0) = "10" else    -- XOR
```

```
793                     FCmd_OR;                                    -- OR
794
795             CinCmd <= CinCmd_ZERO;
796             SCmd   <= "XXX";
797             ALUCmd <= ALUCmd_FBLOCK;
798
799         elsif std_match(IR, LOGIC_IMM_R0) then
800             -- {AND, TST, OR, XOR} immediate, R0
801
802             -- Register array signals
803             RegASel <= 0;
804
805             RegInSel             <= 0;
806             RegDataInSel         <= RegDataIn_ALUResult;
807             Instruction_RegEnableIn <= IR(9) or IR(8);   -- exclude TST
808             Immediate            <= i_format_i;
809             ImmediateMode        <= ImmediateMode_ZERO;
810
811             -- Enable TFlagSel for TST
812             Instruction_TFlagSel <= TFlagSel_Zero when IR(9 downto 8) = "00"    -- TST
813                                     else TFlagSel_T;                            -- AND, OR, XOR
814
815             -- ALU signals for logic instructions using the FBlock
816             ALUOpBSel <= ALUOpB_Imm;
817             LoadA     <= '1';
818
819             -- Bit-decode f-block operation
820             FCmd <= FCmd_AND when IR(9) = '0' else              -- AND, TST
821                     FCmd_XOR when IR(9 downto 8) = "10" else    -- XOR
822                     FCmd_OR;                                    -- OR
823
824             CinCmd <= CinCmd_ZERO;
825             SCmd   <= "XXX";
826             ALUCmd <= ALUCmd_FBLOCK;
827
828         elsif std_match(IR, NOT_RM_RN) then
829             -- NOT Rm, Rn
830
831             -- Register array signals
832             RegASel <= to_integer(unsigned(nm_format_n));
833             RegBSel <= to_integer(unsigned(nm_format_m));
834
835             RegInSel             <= to_integer(unsigned(nm_format_n));
836             RegDataInSel         <= RegDataIn_ALUResult;
837             Instruction_RegEnableIn <= '1';
838
839             -- ALU signals for logical negation
840             ALUOpBSel <= ALUOpB_RegB;
841             LoadA     <= '1';
842             FCmd      <= FCmd_BNOT;
843             CinCmd    <= CinCmd_ZERO;
844             SCmd      <= "XXX";
845             ALUCmd    <= ALUCmd_FBLOCK;
846
847         elsif std_match(IR, CMP_EQ_IMM) then
848             -- CMP/EQ #Imm, R0
849
850             -- Register array signals
851             RegASel <= 0;
852
853             Immediate            <= i_format_i;
854             ImmediateMode        <= ImmediateMode_ZERO;
855
856             -- Compute T flag based on ALU flags
857             Instruction_TFlagSel <= TFlagSel_CMP;
858             TCMPSel <= TCmp_EQ;
```

```
859
860             -- ALU Instructions that perform a subtraction (Rn - immediate) so that
861             -- the ALU output flags can be used to compute the T flag
862             ALUOpBSel <= ALUOpB_Imm;
863             LoadA     <= '1';
864             FCmd      <= FCmd_BNOT;
865             CinCmd <= CinCmd_ONE;
866             SCmd   <= "XXX";
867             ALUCmd <= ALUCmd_ADDER;
868
869         elsif std_match(IR, CMP_RM_RN) then
870             -- CMP/XX Rm, Rn
871
872             -- Register array signals
873             RegASel <= to_integer(unsigned(nm_format_n));
874             RegBSel <= to_integer(unsigned(nm_format_m));
875
876             -- Compute T flag based on ALU flags
877             Instruction_TFlagSel <= TFlagSel_CMP;
878             TCMPSel <= IR(2 downto 0);              -- bit decode T flag CMP condition
879
880             -- ALU Instructions that perform a subtraction (Rn - Rm) so that
881             -- the ALU output flags can be used to compute the T flag
882             ALUOpBSel <= ALUOpB_RegB;
883             LoadA     <= '1';
884             FCmd      <= FCmd_BNOT;
885             CinCmd <= CinCmd_ONE;
886             SCmd   <= "XXX";
887             ALUCmd <= ALUCmd_ADDER;
888
889         elsif std_match(IR, CMP_STR_RM_RN) then
890             -- CMP/STR Rm, Rn
891
892             -- Register array signals
893             RegASel <= to_integer(unsigned(nm_format_n));
894             RegBSel <= to_integer(unsigned(nm_format_m));
895
896             -- Compute T flag based on ALU flags
897             Instruction_TFlagSel <= TFlagSel_CMP;
898             TCMPSel <= TCMP_STR;
899
900         elsif std_match(IR, CMP_RN) then
901             -- CMP/{PL/PZ} Rn
902
903             -- Register array signals
904             RegASel <= to_integer(unsigned(nm_format_n));
905
906             -- Compare to 0
907             Immediate <= (others => '0');
908
909             -- Compute T flag based on ALU flags
910             Instruction_TFlagSel <= TFlagSel_CMP;
911             TCMPSel <= IR(2) & "11";                -- bit decode CMP mode (either GT or GE)
912
913             -- ALU Instructions that perform a subtraction (Rn - 0) so that
914             -- the ALU output flags can be used to compute the T flag
915             ALUOpBSel <= ALUOpB_Imm;
916             LoadA     <= '1';
917             FCmd      <= FCmd_BNOT;
918             CinCmd <= CinCmd_ONE;
919             SCmd   <= "XXX";
920             ALUCmd <= ALUCmd_ADDER;
921
922         elsif std_match(IR, SHIFT_RN) then
923             -- Shift operations
924             -- {ROTL, ROTR, ROTCL, ROTCR, SHAL, SHAR, SHLL, SHLR} Rn
```

```
925             -- Uses bit decoding to compute control signals (to reduce code size)
926
927             -- Register array signals
928             RegASel              <= to_integer(unsigned(n_format_n));
929             RegInSel             <= to_integer(unsigned(n_format_n));
930             RegDataInSel         <= RegDataIn_ALUResult;
931             Instruction_RegEnableIn <= '1';
932
933             Instruction_TFlagSel <= TFlagSel_Carry;
934
935             -- ALU signals
936             ALUOpBSel <= ALUOpB_RegB;
937             LoadA     <= '1';
938             FCmd      <= "XXXX";
939
940             -- Bit-decode carry command
941             CinCmd    <= CinCmd_CIN when (IR(5) and IR(2)) = '1' else   -- ROTCL, ROTCR
942                          CinCmd_ZERO;                                   -- all others
943
944             SCmd   <= IR(0) & IR(2) & IR(5);  -- bit-decode shift operation
945             ALUCmd <= ALUCmd_SHIFT;
946
947         elsif std_match(IR, BSHIFT_RN) then
948             -- Barrel shift operations
949             -- {SHLL,SHLR}{2,8,16} Rn
950             -- Uses bit decoding to compute control signals (to reduce code size)
951
952             -- Register array signals
953             RegASel              <= to_integer(unsigned(n_format_n));
954             RegInSel             <= to_integer(unsigned(n_format_n));
955             RegDataInSel         <= RegDataIn_ALUResult;
956             Instruction_RegEnableIn <= '1';
957
958             Instruction_TFlagSel <= TFlagSel_T;
959
960             -- ALU signals
961             LoadA     <= '1';
962             SCmd   <= IR(5) & IR(4) & IR(0);  -- bit-decode barrel shift operation
963             ALUCmd <= ALUCmd_BSHIFT;
964
965         -- Data Transfer Instruction ------------------------------------------
966
967         -- MOV #imm, Rn
968         -- ni format
969         elsif std_match(IR, MOV_IMM_RN) then
970
971             LogWithTime(l, "sh2_control.vhd: Decoded MOV H'" & to_hstring(ni_format_i) &
972                           ", R" & to_string(slv_to_uint(ni_format_n)), LogFile);
973
974             RegInSel             <= to_integer(unsigned(ni_format_n));
975             RegDataInSel         <= RegDataIn_Immediate;
976             Instruction_RegEnableIn <= '1';
977             Immediate            <= ni_format_i;
978
979         -- MOV.W @(disp, PC), Rn
980         -- nd8 format
981         -- NOTE: Testing this assumes MOV into memory works.
982         --
983         elsif std_match(IR, MOV_W_AT_DISP_PC_RN) then
984           LogWithTime(l,
985             "sh2_control.vhd: Decoded MOV.W @(0x" & to_hstring(nd8_format_d) &
986             ", PC), R" & to_string(slv_to_uint(nd8_format_n)), LogFile);
987
988
989             RegInSel             <= to_integer(unsigned(nd8_format_n));   -- Writing to register n
990             RegDataInSel         <= RegDataIn_DB;                         -- Writing output of data bus to register.
```

```
991             Instruction_RegEnableIn <= '1';                                  -- Writes to register.
992
993             RegASel <= to_integer(unsigned(nd8_format_n));
994
995             -- Instruction reads word from program memory (ROM).
996             Instruction_MemEnable <= '1';
997             Instruction_ReadWrite <= ReadWrite_READ;
998             Instruction_WordMode  <= WordMode;
999             Instruction_MemSel    <= MemSel_ROM;
1000
1001            -- DMAU signals for PC Relative addressing with displacement (word mode)
1002            BaseSel      <= BaseSel_PC;
1003            IndexSel     <= IndexSel_OFF8;
1004            OffScalarSel <= OffScalarSel_TWO;
1005            IncDecSel    <= IncDecSel_NONE;
1006            DMAUOff8     <= nd8_format_d;
1007
1008
1009        -- MOV.L @(disp, PC), Rn
1010        -- nd8 format
1011        elsif std_match(IR, MOV_L_AT_DISP_PC_RN) then
1012            LogWithTime(l,
1013              "sh2_control.vhd: Decoded MOV.L @(0x" & to_hstring(nd8_format_d) &
1014              ", PC), R" & to_string(slv_to_uint(nd8_format_n)), LogFile);
1015
1016            RegInSel             <= to_integer(unsigned(nd8_format_n));  -- Writing to register n
1017            RegDataInSel         <= RegDataIn_DB;                        -- Writing output of data bus to register.
1018            Instruction_RegEnableIn <= '1';                                 -- Writes to register.
1019
1020            -- Instruction reads from longword memory.
1021            Instruction_MemEnable <= '1';
1022            Instruction_ReadWrite <= ReadWrite_READ;
1023            Instruction_WordMode  <= LongwordMode;
1024            Instruction_MemSel    <= MemSel_ROM;
1025
1026            -- DMAU signals for PC Relative addressing with displacement (longword mode)
1027            BaseSel      <= BaseSel_PC;
1028            IndexSel     <= IndexSel_OFF8;
1029            OffScalarSel <= OffScalarSel_FOUR;
1030            IncDecSel    <= IncDecSel_NONE;
1031            DMAUOff8     <= nd8_format_d;
1032
1033
1034        -- MOV Rm, Rn
1035        -- nm format
1036        -- Note: for bit decoding, this must be done before MOV_AT_RM_RN
1037        elsif std_match(IR, MOV_RM_RN) then
1038            LogWithTime(l,
1039              "sh2_control.vhd: Decoded MOV R" & to_string(slv_to_uint(nm_format_m)) &
1040              "R" & to_string(slv_to_uint(nm_format_n)) , LogFile);
1041
1042            -- report "Instruction: MOV Rm, Rn";
1043            RegBSel              <= to_integer(unsigned(nm_format_m));
1044            RegInSel             <= to_integer(unsigned(nm_format_n));
1045            RegDataInSel         <= RegDataIn_RegB;
1046            Instruction_RegEnableIn <= '1';
1047
1048        -- MOV.X Rm, @Rn
1049        -- nm format
1050        elsif std_match(IR, MOV_RM_AT_RN) then
1051            LogWithTime(l,
1052              "sh2_control.vhd: Decoded MOV.X R" & to_string(slv_to_uint(nm_format_m)) &
1053              ", @R" & to_string(slv_to_uint(nm_format_n)) , LogFile);
1054
1055            -- Writes a byte to memory to memory
1056            Instruction_MemEnable <= '1';             -- Uses memory.
```

```
1057              Instruction_ReadWrite <= ReadWrite_WRITE; -- Writes.
1058              Instruction_WordMode  <= IR(1 downto 0);  -- bit decode memory mode
1059
1060              MemOutSel <= MemOut_RegB; -- Output RegB (Rm) to memory data bus.
1061
1062              RegBSel  <= to_integer(unsigned(nm_format_m)); -- RegB is Rm.
1063              RegA1Sel <= to_integer(unsigned(nm_format_n)); -- RegA is @(Rn)
1064
1065              -- DMAU signals (for Indirect Register Addressing)
1066              BaseSel      <= BaseSel_REG;
1067              IndexSel     <= IndexSel_NONE;
1068              OffScalarSel <= OffScalarSel_ONE;
1069              IncDecSel    <= IncDecSel_NONE;
1070
1071          -- MOV.X @Rm, Rn
1072          -- nm format
1073          -- Note: for bit decoding, this must be done after MOV_RM_RN
1074          elsif std_match(IR, MOV_AT_RM_RN) then
1075            LogWithTime(l,
1076              "sh2_control.vhd: Decoded MOV.X @R" & to_string(slv_to_uint(nm_format_m)) &
1077              ", R" & to_string(slv_to_uint(nm_format_n)) , LogFile);
1078
1079            -- Instruction reads byte from memory.
1080            Instruction_MemEnable <= '1';            -- Instr does memory access.
1081            Instruction_ReadWrite <= ReadWrite_READ; -- Instr reads from memory.
1082            Instruction_WordMode  <= IR(1 downto 0); -- bit decode memory mode
1083
1084            -- DMAU signals for Indirect Register addressing.
1085            BaseSel      <= BaseSel_REG;
1086            IndexSel     <= IndexSel_NONE;
1087            OffScalarSel <= OffScalarSel_ONE;
1088            IncDecSel    <= IncDecSel_NONE;
1089
1090            -- Output @(Rm) to RegA2.
1091            RegA2Sel <= to_integer(unsigned(nm_format_m));
1092
1093            RegInSel             <= to_integer(unsigned(nm_format_n));
1094            RegDataInSel         <= RegDataIn_DB;
1095            Instruction_RegEnableIn <= '1';
1096
1097
1098          -- MOV.B Rm, @-Rn
1099          -- nm format
1100          elsif std_match(IR, MOV_RM_AT_MINUS_RN) then
1101              LogWithTime(l,
1102                "sh2_control.vhd: Decoded MOV.X R" & to_string(slv_to_uint(nm_format_m)) &
1103                ", @-R" & to_string(slv_to_uint(nm_format_n)) , LogFile);
1104
1105              -- Writes a byte to memory
1106              Instruction_MemEnable <= '1';             -- Uses memory.
1107              Instruction_ReadWrite <= ReadWrite_WRITE; -- Writes.
1108              Instruction_WordMode  <= IR(1 downto 0);  -- bit decode memory mode
1109
1110              MemOutSel <= MemOut_RegB; -- Output RegB (Rm) to memory data bus.
1111
1112              RegBSel               <= to_integer(unsigned(nm_format_m));  -- Output Rm from RegB output.
1113              RegA1Sel              <= to_integer(unsigned(nm_format_n));  -- Output @(Rn) from RegA1 output.
1114              RegAxInSel            <= to_integer(unsigned(nm_format_n));  -- Store calculated address into Rn
1115              Instruction_RegAxStore <= '1';                               -- Enable writes to address registers.
1116
1117              -- DMAU signals (for Pre-decrement indirect register addressing)
1118              BaseSel      <= BaseSel_REG;
1119              IndexSel     <= IndexSel_NONE;
1120              OffScalarSel <= IR(1 downto 0);         -- bit decode offset scalar factor
1121              IncDecSel    <= IncDecSel_PRE_DEC;
1122
```

```
1123          -- MOV.B @Rm+, Rn
1124          -- nm format
1125          elsif std_match(IR, MOV_AT_RM_PLUS_RN) then
1126            LogWithTime(l,
1127              "sh2_control.vhd: Decoded MOV.{B,W,L} @R" & to_string(slv_to_uint(nm_format_m)) &
1128              "+, R" & to_string(slv_to_uint(nm_format_n)) , LogFile);
1129
1130            -- MOV with post-increment. This Instruction reads a byte, word,
1131            -- or longword from an address in Rm, into Rn. The address is
1132            -- incremented and stored in Rm after the value is retrieved.
1133
1134            -- Reads a byte from memory.
1135            Instruction_MemEnable <= '1';             -- Uses memory.
1136            Instruction_ReadWrite <= ReadWrite_READ;  -- Reads.
1137            Instruction_WordMode  <= IR(1 downto 0);  -- bit-decode memory word mode
1138
1139            -- Output @Rm from RegA2
1140            RegA2Sel <= to_integer(unsigned(nm_format_m));
1141
1142            -- Write output of Data Bus to Rn
1143            RegInSel             <= to_integer(unsigned(nm_format_n));
1144            RegDataInSel         <= RegDataIn_DB;
1145            Instruction_RegEnableIn <= '1';  -- Enable writing to registers.
1146
1147            -- Write the incremented address to Rm
1148            RegAxInSel             <= to_integer(unsigned(nm_format_m));
1149            Instruction_RegAxStore <= '1'; -- Enable writing to address register in the writeback state.
1150
1151            -- DMAU signals for post-increment indirect register addressing
1152            BaseSel      <= BaseSel_REG;
1153            IndexSel     <= IndexSel_NONE;
1154            OffScalarSel <= IR(1 downto 0);       -- bit-decode offset scalar select
1155            IncDecSel    <= IncDecSel_POST_INC;
1156
1157
1158          -- MOV.{B,W} RO, @(disp,Rn)
1159          -- nd4 format
1160          -- Note that the displacement depends on the mode of the address, so in
1161          -- byte mode, the displacement represents bytes, in word mode it represents
1162          -- words, etc. This is done to maximize it's range.
1163          elsif std_match(IR, MOV_R0_AT_DISP_RN) then
1164
1165            LogWithTime(l,
1166              "sh2_control.vhd: Decoded MOV.{B,W} R0, @(0x" & to_hstring(nd4_format_d) &
1167              ", " & to_string(slv_to_uint(nd4_format_n)) & ")", LogFile);
1168
1169            -- Instruction writes a byte to data memory.
1170            Instruction_MemEnable   <= '1';
1171            Instruction_ReadWrite   <= ReadWrite_WRITE;
1172
1173            Instruction_WordMode    <= ByteMode when IR(8) = '0' else     -- bit-decode byte/word mode
1174                                       WordMode;
1175
1176            -- Output RegB (R0) to memory data bus
1177            MemOutSel <= MemOut_RegB;
1178
1179            -- Output R0 to RegB
1180            RegBSel <= 0;
1181
1182            -- Output Rn to RegA1. The DMAU will use this to calculate the address
1183            -- to write to.
1184            RegA1Sel <= to_integer(unsigned(nd4_format_n));
1185
1186            -- DMAU signals for Indirect register addressing with displacement
1187            BaseSel       <=  BaseSel_REG;
1188            IndexSel      <=  IndexSel_OFF4;
```

```
1189            OffScalarSel  <=  OffScalarSel_ONE when IR(8) = '0' else      -- bit-decode byte/word
1190                              OffScalarSel_TWO;
1191            IncDecSel     <=  IncDecSel_NONE;
1192            DMAUOff4      <=  nd4_format_d;
1193
1194
1195          -- MOV.L Rm, @(disp, Rn)
1196          -- nmd format
1197          elsif std_match(IR, MOV_L_RM_AT_DISP_RN) then
1198
1199            LogWithTime(l,
1200              "sh2_control.vhd: Decoded MOV.L R" & to_string(slv_to_uint(nmd_format_m)) &
1201              ", @(0x" & to_hstring(nmd_format_d) & ", R" & to_string(slv_to_uint(nmd_format_n)) & ")", LogFile);
1202
1203            Instruction_MemEnable <= '1';
1204            Instruction_ReadWrite <= ReadWrite_WRITE;
1205            Instruction_WordMode  <= LongwordMode;
1206
1207            -- Output Rm to RegB.
1208            RegBSel <= to_integer(unsigned(nmd_format_m));
1209
1210            -- Output Rn to RegA1. The DMAU will use this to calculate the address
1211            -- to write to.
1212            RegA1Sel <= to_integer(unsigned(nmd_format_n));
1213
1214            -- Output RegB (Rm) to memory data bus. This will be written to memory.
1215            MemOutSel <= MemOut_RegB;
1216
1217            -- DMAU signals for Indirect register addressing with displacement (longword mode)
1218            BaseSel      <= BaseSel_REG;
1219            IndexSel     <= IndexSel_OFF4;
1220            OffScalarSel <= OffScalarSel_FOUR;
1221            IncDecSel    <= IncDecSel_NONE;
1222            DMAUOff4     <= nmd_format_d;
1223
1224
1225          -- MOV.{B,W} @(disp, Rm), R0
1226          -- md format
1227          -- Note that these instructions are very similar to MOV @(disp, PC), Rn
1228        elsif std_match(IR, MOV_AT_DISP_RM_R0) then
1229
1230            LogWithTime(l,
1231              "sh2_control.vhd: Decoded MOV.{B,W} @(0x" & to_hstring(md_format_d) &
1232              ", R" & to_string(slv_to_uint(md_format_m)) & "), R0", LogFile);
1233
1234              -- Writing sign-extended byte from data bus to R0.
1235              RegInSel             <= 0;             -- Select R0 to write to.
1236              RegDataInSel         <= RegDataIn_DB;  -- Write DataBus to reg.
1237              Instruction_RegEnableIn <= '1';           -- Enable Reg writing for this instruction.
1238
1239              -- Output @Rm from RegA2
1240              RegA2Sel <= to_integer(unsigned(md_format_m));
1241
1242              Instruction_MemEnable  <=  '1';            -- Instr uses memory.
1243              Instruction_ReadWrite  <=  ReadWrite_READ; -- Reads.
1244              Instruction_WordMode   <=  ByteMode when IR(8) = '0' else       -- bit-decode word mode
1245                                         WordMode;
1246              Instruction_MemSel     <=  MemSel_RAM;     -- Reads from RAM
1247
1248              -- DMAU signals for Indirect register addressing with displacement (byte mode)
1249              BaseSel      <= BaseSel_REG;
1250              IndexSel     <= IndexSel_OFF4;
1251              OffScalarSel <= OffScalarSel_ONE when IR(8) = '0' else          -- bit-decode offset scale
1252                              OffScalarSel_TWO;
1253              IncDecSel    <= IncDecSel_NONE;
1254              DMAUOff4     <= md_format_d;
```

```
1255
1256
1257          -- MOV.L @(disp, Rm), Rn
1258          -- nmd
1259          elsif std_match(IR, MOV_L_AT_DISP_RM_RN) then
1260
1261            LogWithTime(l,
1262              "sh2_control.vhd: Decoded MOV.L @(0x" & to_hstring(nmd_format_d) &
1263              ", R" & to_string(slv_to_uint(nmd_format_m)) & "), R" & to_string(slv_to_uint(nmd_format_n))
1264              , LogFile);
1265
1266            -- Writing longword from data bus to Rn.
1267            RegInSel             <= to_integer(unsigned(nmd_format_n));   -- Select Rn to write to.
1268            RegDataInSel         <= RegDataIn_DB;                         -- Write DataBus to reg.
1269            Instruction_RegEnableIn <= '1';                                  -- Enable Reg writing for this instruction.
1270
1271             -- Output @Rm from RegA2
1272             RegA2Sel <= to_integer(unsigned(nmd_format_m));
1273
1274            Instruction_MemEnable  <=  '1';               -- Instr uses memory.
1275            Instruction_ReadWrite  <=  ReadWrite_READ;    -- Reads.
1276            Instruction_WordMode   <=  LongwordMode;      -- Reads longword.
1277            Instruction_MemSel     <=  MemSel_RAM;        -- Reads from RAM
1278
1279
1280            -- DMAU signals for Indirect register addressing with displacement (longword mode)
1281            BaseSel      <= BaseSel_REG;
1282            IndexSel     <= IndexSel_OFF4;
1283            OffScalarSel <= OffScalarSel_FOUR;
1284            IncDecSel    <= IncDecSel_NONE;
1285            DMAUOff4     <= nmd_format_d;
1286
1287
1288          -- MOV.{B,W,L} Rm, @(R0, Rn)
1289          -- nm format
1290          elsif std_match(IR, MOV_RM_AT_R0_RN) then
1291
1292            LogWithTime(l,
1293              "sh2_control.vhd: Decoded MOV.X R" & to_string(slv_to_uint(nm_format_m)) &
1294              ", @(R0, R" & to_string(slv_to_uint(nm_format_n)) & ")", LogFile);
1295
1296            -- Instr writes a byte to memory.
1297            Instruction_MemEnable <= '1';
1298            Instruction_ReadWrite <= ReadWrite_WRITE;
1299            Instruction_WordMode  <= IR(1 downto 0);      -- bit-decode memory mode
1300
1301            -- Output Rm to RegB.
1302            RegBSel <= to_integer(unsigned(nm_format_m));
1303
1304            -- Output Rn to RegA1. The DMAU will use this to calculate the address
1305            -- to write to.
1306            RegA1Sel <= to_integer(unsigned(nm_format_n));
1307
1308            -- Output R0 to RegA2.
1309            RegA2Sel <= 0;
1310
1311            -- Output RegB (Rm) to memory data bus. This will be written to memory.
1312            MemOutSel <= MemOut_RegB;
1313
1314            -- DMAU Signals for Indirect Register Addressing
1315            BaseSel       <= BaseSel_REG;
1316            IndexSel      <= IndexSel_R0;
1317            OffScalarSel  <= OffScalarSel_ONE;
1318            IncDecSel     <= IncDecSel_NONE;
1319
1320
```

```
1321        -- MOV.{B,W,L} @(R0, Rm), Rn
1322        -- nm format
1323        elsif std_match(IR, MOV_AT_R0_RM_RN) then
1324
1325            LogWithTime(l,
1326              "sh2_control.vhd: Decoded MOV.X @(R0, R" & to_string(slv_to_uint(nm_format_m)) &
1327              "), R" & to_string(slv_to_uint(nm_format_n)), LogFile);
1328
1329            -- Writing sign-extended byte from data bus to Rn.
1330            RegInSel             <= slv_to_uint(nm_format_n);     -- Select Rn to write to.
1331            RegDataInSel         <= RegDataIn_DB;                 -- Write DataBus to reg.
1332            Instruction_RegEnableIn <= '1';                          -- Enable Reg writing for this instruction.
1333
1334            -- Output @Rm from RegA2
1335            RegA2Sel <= slv_to_uint(nm_format_m);
1336
1337            -- Output @R0 from RegA1
1338            RegA1Sel <= 0;
1339
1340            Instruction_MemEnable  <=  '1';             -- Instr uses memory.
1341            Instruction_ReadWrite  <=  ReadWrite_READ;  -- Reads.
1342            Instruction_WordMode   <=  IR(1 downto 0);  -- bit decode memory mode
1343            Instruction_MemSel     <=  MemSel_RAM;      -- Reads from RAM
1344
1345            -- DMAU Signals for Indirect indexed Register Addressing
1346            BaseSel       <= BaseSel_REG;
1347            IndexSel      <= IndexSel_R0;
1348            OffScalarSel  <= OffScalarSel_ONE;
1349            IncDecSel     <= IncDecSel_NONE;
1350
1351
1352        -- MOV.{B,W,L} R0, @(disp, GBR)
1353        -- d format
1354        elsif std_match(IR, MOV_R0_AT_DISP_GBR) then
1355
1356          LogWithTime(l,
1357            "sh2_control.vhd: Decoded MOV.X R0, @(0x" & to_hstring(d_format_d) &
1358            ", GBR)", LogFile);
1359
1360          -- Writing to memory
1361          Instruction_MemEnable <= '1';
1362          Instruction_ReadWrite <= ReadWrite_WRITE;
1363          Instruction_WordMode  <= IR(9 downto 8);      -- bit-decode memory mode
1364
1365          -- Output R0 to RegB.
1366          RegBSel <= 0;
1367
1368          -- Output RegB (Rm) to memory data bus. This will be written to memory.
1369          MemOutSel <= MemOut_RegB;
1370
1371          -- DMAU signals for Indirect GBR addressing with displacement
1372          BaseSel       <=  BaseSel_GBR;
1373          IndexSel      <=  IndexSel_OFF8;
1374          OffScalarSel  <=  IR(9 downto 8);     -- bit decode offset scalar select
1375          IncDecSel     <=  IncDecSel_NONE;
1376          DMAUOff8      <=  d_format_d;
1377
1378        -- MOVA @(disp, PC), R0
1379        -- d format
1380        -- disp*4 + PC -> R0
1381        -- Note: due to bit decoding, this must come before MOV_AT_DISP_GBR_R0
1382        elsif std_match(IR, MOVA_AT_DISP_PC_R0) then
1383
1384            LogWithTime(l,
1385                "sh2_control.vhd: Decoded MOVA @(" & to_hstring(d_format_d) &
1386                ", PC), R0", LogFile);
```

```
1387
1388            -- Note that this instruction moves the address, disp*4 + PC
1389            -- (calculated by the DMAU) into R0. It does NOT move the data at
1390            -- this address.
1391
1392            RegAxInSel             <=  0;   -- Write address to R0
1393            Instruction_RegAxStore <= '1';  -- Enable writing to address register in writeback state.
1394
1395            -- DMAU signals for PC Relative addressing with displacement (longword mode)
1396            BaseSel      <= BaseSel_PC;
1397            IndexSel     <= IndexSel_OFF8;
1398            OffScalarSel <= OffScalarSel_FOUR;
1399            IncDecSel    <= IncDecSel_NONE;
1400            DMAUOff8     <= nd8_format_d;
1401
1402        -- MOV.{B,W,L} @(disp, GBR), R0
1403        -- d format
1404        -- Note: due to bit decoding, this must come after MOVA_AT_DISP_PC_R0
1405        elsif std_match(IR, MOV_AT_DISP_GBR_R0) then
1406
1407            LogWithTime(l,
1408                "sh2_control.vhd: Decoded MOV.X @(0x" & to_hstring(d_format_d) &
1409                ", GBR), R0", LogFile);
1410
1411           RegInSel             <= 0;               -- Write to R0
1412           RegDataInSel         <= RegDataIn_DB;    -- Write Data bus to R0
1413           Instruction_RegEnableIn <= '1';             -- Enable register writing for this instruction.
1414
1415           Instruction_MemEnable <= '1';
1416           Instruction_ReadWrite <= ReadWrite_READ;
1417           Instruction_WordMode  <= IR(9 downto 8); -- bit decode memory mode
1418
1419          -- DMAU signals for Indirect GBR addressing with displacement (byte mode)
1420          BaseSel      <=  BaseSel_GBR;
1421          IndexSel     <=  IndexSel_OFF8;
1422          OffScalarSel <=  IR(9 downto 8);          -- bit decode offset scalar select
1423          IncDecSel    <=  IncDecSel_NONE;
1424          DMAUOff8     <=  d_format_d;
1425
1426        -- MOVT Rn
1427        -- n format.
1428        elsif std_match(IR, MOVT_RN) then
1429
1430            LogWithTime(l,
1431                "sh2_control.vhd: Decoded MOVT R" & to_string(slv_to_uint(n_format_n)),
1432                LogFile);
1433
1434            -- TODO: This is not consistent with the convention that RegB is
1435            -- always Rm !!!
1436            --
1437            RegInSel             <= to_integer(unsigned(n_format_n));
1438            RegDataInSel         <= RegDataIn_SR_TBit;
1439            Instruction_RegEnableIn <= '1';
1440
1441
1442        -- SWAP.B Rm, Rn
1443        -- nm format
1444        -- Rm -> Swap upper and lower 2 bytes -> Rn
1445        elsif std_match(IR, SWAP_RM_RN) then
1446
1447            LogWithTime(l,
1448                "sh2_control.vhd: Decoded SWAP.X R" & to_string(slv_to_uint(nm_format_m))
1449                & ", R" & to_string(nm_format_n), LogFile);
1450
1451            RegASel      <= slv_to_uint(nm_format_m);
1452            RegInSel     <= slv_to_uint(nm_format_n);
```

```
1453
1454            -- Bit decode if whether byte or word mode
1455            RegDataInSel <= RegDataIn_RegA_SWAP_B when IR(0) = '0' else
1456                            RegDataIn_RegA_SWAP_W;
1457
1458            Instruction_RegEnableIn <= '1';
1459
1460
1461        -- XTRCT Rm, Rn
1462        -- nm format
1463        -- Center 32 bits of Rm and Rn -> Rn
1464        elsif std_match(IR, XTRCT_RM_RN) then
1465
1466            LogWithTime(l,
1467                "sh2_control.vhd: Decoded XTRCT R" & to_string(slv_to_uint(nm_format_m))
1468                & ", R" & to_string(nm_format_n), LogFile);
1469
1470            RegASel <= slv_to_uint(nm_format_n);
1471            RegBSel <= slv_to_uint(nm_format_m);
1472
1473            RegInSel <= slv_to_uint(nm_format_n); -- Write to Rn
1474
1475            RegDataInSel <= RegDataIn_REGB_REGA_CENTER;
1476
1477            Instruction_RegEnableIn <= '1';
1478
1479
1480
1481        -- Branch Instructions ------------------------------------------------
1482
1483        -- BF <label> (where label is disp*2 + PC)
1484        -- d format
1485         elsif std_match(IR, BF) then
1486
1487             LogWithTime(l,
1488                 "sh2_control.vhd: Decoded BF (label=" & to_hstring(d_format_d) &
1489                 "*2 + PC)", LogFile);
1490
1491             -- If T=0, disp*2 + PC -> PC; if T=1, nop (where label is disp*2 + PC)
1492
1493             if (TFlagIn = '0') then
1494
1495                 Instruction_PCAddrMode  <= PCAddrMode_RELATIVE_8;
1496                 PMAUOff8                <= d_format_d;
1497
1498             else
1499
1500                 -- Go to the next instruction.
1501                 Instruction_PCAddrMode  <= PCAddrMode_INC;  -- Increment PC
1502
1503             end if;
1504
1505
1506
1507          -- BF/S <label> (where label is disp*2 + PC)
1508          -- d format
1509          elsif std_match(IR, BF_S) then
1510
1511              LogWithTime(l,
1512                  "sh2_control.vhd: Decoded BF/S (label=" & to_hstring(d_format_d) &
1513                  "*2 + PC)", LogFile);
1514
1515              if (TFlagIn = '0') then
1516                  -- Take the branch
1517
1518                  --  The delay will be taken.
```

```
1519                    Instruction_DelayedBranchTaken  <= '1';
1520                    PCWriteCtrl                     <= PCWriteCtrl_WRITE_CALC;
1521
1522                    Instruction_PCAddrMode <= PCAddrMode_RELATIVE_8;
1523                    PMAUOff8               <= d_format_d;
1524
1525                else
1526                    -- Go to the next instruction.
1527                    Instruction_PCAddrMode  <= PCAddrMode_INC;  -- Increment PC
1528                end if;
1529
1530
1531            -- BT <label> (where label is disp*2 + PC)
1532            -- d format
1533            elsif std_match(IR, BT) then
1534
1535                -- Branch true without delay slot.
1536
1537                LogWithTime(l,
1538                    "sh2_control.vhd: Decoded BT (label=" & to_hstring(d_format_d) &
1539                    "*2 + PC)", LogFile);
1540
1541                -- If T=1, disp*2 + PC -> PC; if T=0, nop (where label is disp*2 + PC)
1542
1543                if (TFlagIn = '1') then
1544
1545                   Instruction_PCAddrMode <= PCAddrMode_RELATIVE_8;
1546                   PMAUOff8               <= d_format_d;
1547
1548                else
1549                    -- Go to the next instruction.
1550                    Instruction_PCAddrMode  <= PCAddrMode_INC;  -- Increment PC
1551                end if;
1552
1553
1554            -- BT/S <label> (where label is disp*2 + PC)
1555            -- d format
1556            elsif std_match(IR, BT_S) then
1557
1558                LogWithTime(l,
1559                    "sh2_control.vhd: Decoded BT/S (label=" & to_hstring(d_format_d) &
1560                    "*2 + PC)", LogFile);
1561
1562
1563                -- If T=1, disp*2 + PC -> PC; if T=0, nop (where label is disp*2 + PC)
1564                if (TFlagIn = '1') then
1565
1566                    --  The delay will be taken.
1567                    Instruction_DelayedBranchTaken  <= '1';
1568                    PCWriteCtrl                     <= PCWriteCtrl_WRITE_CALC;
1569
1570                    Instruction_PCAddrMode <= PCAddrMode_RELATIVE_8;
1571                    PMAUOff8               <= d_format_d;
1572
1573                else
1574                    -- Go to the next instruction.
1575                    Instruction_PCAddrMode  <= PCAddrMode_INC;  -- Increment PC
1576                end if;
1577
1578
1579            -- BRA <label> (where label is disp*2 + PC)
1580            -- d12 format
1581            elsif std_match(IR, BRA) then
1582
1583                LogWithTime(l,
1584                    "sh2_control.vhd: Decoded BRA (label=" & to_hstring(d12_format_d) &
```

```
1585                 "*2 + PC)", LogFile);
1586
1587             Instruction_DelayedBranchTaken <= '1';
1588             PCWriteCtrl                    <= PCWriteCtrl_WRITE_CALC;
1589
1590             Instruction_PCAddrMode <= PCAddrMode_RELATIVE_12;
1591             PMAUOff12              <= d12_format_d;
1592
1593
1594          -- BRAF Rm
1595          -- m format
1596          elsif std_match(IR, BRAF) then
1597
1598              -- Delayed branch, Rm + PC -> PC
1599
1600              -- Note that the PMAU's register input is always RegB.
1601
1602             RegBSel <= slv_to_uint(m_format_m);
1603
1604             LogWithTime(l,
1605                 "sh2_control.vhd: Decoded BRAF R" & to_string(slv_to_uint(m_format_m)), LogFile);
1606
1607             Instruction_DelayedBranchTaken <= '1';
1608             PCWriteCtrl                    <= PCWriteCtrl_WRITE_CALC;
1609
1610             Instruction_PCAddrMode <= PCAddrMode_REG_DIRECT_RELATIVE;
1611
1612
1613          -- BSR <label> (where label is disp*2)
1614          -- d12 format
1615          elsif std_match(IR, BSR) then
1616
1617              LogWithTime(l,
1618                  "sh2_control.vhd: Decoded BSR (label=" & to_hstring(d12_format_d) &
1619                  "*2 + PC)", LogFile);
1620
1621              Instruction_DelayedBranchTaken <= '1';
1622              PCWriteCtrl                    <= PCWriteCtrl_WRITE_CALC;
1623
1624              Instruction_PCAddrMode <= PCAddrMode_RELATIVE_12;
1625              PMAUOff12  <= d12_format_d;
1626
1627              Instruction_PRWriteEn <= '1';
1628
1629              -- Control signals to write PC to PR.
1630              SysRegSrc              <= SysRegSrc_PC;
1631              Instruction_SysRegCtrl <= SysRegCtrl_LOAD;
1632
1633
1634          -- BSRF Rm
1635          -- m format
1636          --
1637          -- Branch to sub-routine far.
1638          -- PC -> PR, Rm + PC -> PC
1639          elsif std_match(IR, BSRF) then
1640
1641              LogWithTime(l,
1642                  "sh2_control.vhd: Decoded BSRF R" & to_string(slv_to_uint(m_format_m)), LogFile);
1643
1644             -- Basically BSR, but with a different target.
1645              Instruction_DelayedBranchTaken <= '1';
1646              PCWriteCtrl                    <= PCWriteCtrl_WRITE_CALC;
1647
1648              Instruction_PCAddrMode <= PCAddrMode_REG_DIRECT_RELATIVE;
1649
1650              Instruction_PRWriteEn <= '1';
```

```
1651
1652              -- Control signals to write PC to PR.
1653              SysRegSrc              <= SysRegSrc_PC;
1654              Instruction_SysRegCtrl <= SysRegCtrl_LOAD;
1655
1656
1657           -- JMP @Rm
1658           -- m format
1659           -- Delayed branch, Rm -> PC
1660           elsif std_match(IR, JMP) then
1661
1662              LogWithTime(l,
1663                  "sh2_control.vhd: Decoded JMP @R" & to_string(slv_to_uint(m_format_m)), LogFile);
1664
1665              -- PMAU Register input is RegB.
1666              RegBSel <= slv_to_uint(m_format_m);
1667
1668              Instruction_DelayedBranchTaken <= '1';
1669              PCWriteCtrl                    <= PCWriteCtrl_WRITE_CALC;
1670              Instruction_PCAddrMode         <= PCAddrMode_REG_DIRECT;
1671
1672
1673           -- JSR @Rm
1674           -- m format
1675           -- Delayed branch, PC -> PR, Rm -> PC
1676           elsif std_match(IR, JSR) then
1677
1678              LogWithTime(l,
1679                  "sh2_control.vhd: Decoded JSR @R" & to_string(slv_to_uint(m_format_m)), LogFile);
1680
1681              RegBSel <= slv_to_uint(m_format_m);
1682
1683              Instruction_DelayedBranchTaken <= '1';
1684              PCWriteCtrl                    <= PCWriteCtrl_WRITE_CALC;
1685              Instruction_PCAddrMode         <= PCAddrMode_REG_DIRECT;
1686
1687              Instruction_PRWriteEn <= '1';
1688
1689              SysRegSrc <= SysRegSrc_PC;
1690              Instruction_SysRegCtrl <= SysRegCtrl_LOAD;
1691
1692           elsif std_match(IR, RTS) then
1693
1694              LogWithTime(l,
1695                  "sh2_control.vhd: Decoded RTS", LogFile);
1696
1697              Instruction_PCAddrMode         <= PCAddrMode_PR_DIRECT;
1698              Instruction_DelayedBranchTaken <= '1';
1699              PCWriteCtrl                    <= PCWriteCtrl_WRITE_CALC;
1700
1701
1702          -- System Control Instructions ---------------------------------------
1703
1704          elsif std_match(IR, CLRT) then
1705
1706              LogWithTime(l, "sh2_control.vhd: Decoded CLRT", LogFile);
1707
1708              Instruction_TFlagSel <= TFlagSel_CLEAR;     -- clear the T flag
1709
1710          elsif std_match(IR, CLRMAC) then
1711
1712              LogWithTime(l, "sh2_control.vhd: Decoded CLRMAC", LogFile);
1713
1714              Instruction_SysRegCtrl <= SysRegCtrl_CLEAR;
1715              SysRegSel <= SysRegSel_MACL;
1716
```

```
1717        elsif std_match(IR, SETT) then
1718
1719            LogWithTime(l, "sh2_control.vhd: Decoded SETT", LogFile);
1720
1721            Instruction_TFlagSel <= TFlagSel_SET;       -- set the T flag
1722
1723        elsif std_match(IR, STC_SYS_RN) then
1724
1725            -- STC {SR, GBR, VBR}, Rn
1726            -- Uses bit decoding to choose the system register to store
1727
1728            LogWithTime(l, "sh2_control.vhd: Decoded STC XXX, Rn", LogFile);
1729
1730            RegInSel <= to_integer(unsigned(n_format_n));
1731
1732            -- selects data source to store to a register through bit decoding
1733            SysRegSel <= "0" & IR(5 downto 4);
1734            RegDataInSel <= RegDataIn_SysReg;
1735            Instruction_RegEnableIn <= '1';
1736
1737        elsif std_match(IR, STS_SYS_RN) then
1738
1739            -- STS {MACH, MACL, PR}, Rn
1740            -- Uses bit decoding to choose the system register to store
1741
1742            LogWithTime(l, "sh2_control.vhd: Decoded STS XXX, Rn", LogFile);
1743
1744            RegInSel <= to_integer(unsigned(n_format_n));
1745
1746            -- selects data source to store to a register through bit decoding
1747            SysRegSel <= "1" & IR(5 downto 4);
1748            RegDataInSel <= RegDataIn_SysReg;
1749            Instruction_RegEnableIn <= '1';
1750
1751        elsif std_match(IR, STC_L_SYS_RN) then
1752
1753            -- STC.L {SR, GBR, VBR}, @-Rn
1754            -- Uses bit decoding to choose the system register to store
1755            LogWithTime(l, "sh2_control.vhd: Decoded STC.L XXX, @-Rn", LogFile);
1756
1757            -- Writes a byte to memory
1758            Instruction_MemEnable <= '1';               -- Uses memory.
1759            Instruction_ReadWrite <= ReadWrite_WRITE;   -- Writes.
1760            Instruction_WordMode  <= LongwordMode;      -- bit decode memory mode
1761
1762            -- selects data source to store to a register through bit decoding
1763            SysRegSel <= "0" & IR(5 downto 4);
1764            MemOutSel <= MemOut_SysReg;
1765
1766            RegA1Sel               <= to_integer(unsigned(nm_format_n));  -- Output @(Rn) from RegA1 output.
1767            RegAxInSel             <= to_integer(unsigned(nm_format_n));  -- Store calculated address into Rn
1768            Instruction_RegAxStore <= '1';                                -- Enable writes to address registers.
1769
1770            -- DMAU signals (for Pre-decrement indirect register addressing)
1771            BaseSel      <= BaseSel_REG;
1772            IndexSel     <= IndexSel_NONE;
1773            OffScalarSel <= OffScalarSel_FOUR;
1774            IncDecSel    <= IncDecSel_PRE_DEC;
1775
1776        elsif std_match(IR, STS_L_SYS_RN) then
1777
1778            -- STC.L {MACH, MACL, PR}, @-Rn
1779            -- Uses bit decoding to choose the system register to store
1780            LogWithTime(l, "sh2_control.vhd: Decoded STC.L XXX, @-Rn", LogFile);
1781
1782            -- Writes a byte to memory
```

```
1783             Instruction_MemEnable <= '1';               -- Uses memory.
1784             Instruction_ReadWrite <= ReadWrite_WRITE;   -- Writes.
1785             Instruction_WordMode  <= LongwordMode;      -- bit decode memory mode
1786
1787             -- selects data source to store to a register through bit decoding
1788             SysRegSel <= "1" & IR(5 downto 4);
1789             MemOutSel <= MemOut_SysReg;
1790
1791             RegA1Sel               <= to_integer(unsigned(nm_format_n));  -- Output @(Rn) from RegA1 output.
1792             RegAxInSel             <= to_integer(unsigned(nm_format_n));  -- Store calculated address into Rn
1793             Instruction_RegAxStore <= '1';                                -- Enable writes to address registers.
1794
1795             -- DMAU signals (for Pre-decrement indirect register addressing)
1796             BaseSel      <= BaseSel_REG;
1797             IndexSel     <= IndexSel_NONE;
1798             OffScalarSel <= OffScalarSel_FOUR;
1799             IncDecSel    <= IncDecSel_PRE_DEC;
1800
1801
1802         elsif std_match(IR, LDC_RM_SYS) then
1803
1804             -- LDC Rm, GBR must actually load into the GBR in the DMAU for later instructions
1805             -- to work. Must modify other system control register loads to load to their actual
1806             -- locations as well.
1807             if (std_match(IR, LDC_RM_GBR)) then
1808                 Instruction_GBRWriteEn <= '1';
1809             end if;
1810
1811             -- LDC Rm, {SR, GBR, VBR}
1812             -- Uses bit decoding to choose the system register to load
1813
1814             LogWithTime(l, "sh2_control.vhd: Decoded LDC Rm, X", LogFile);
1815
1816             RegBSel <= to_integer(unsigned(m_format_m));
1817             Instruction_SysRegCtrl <= SysRegCtrl_LOAD;
1818             SysRegSel <= "0" & IR(5 downto 4);      -- bit decode register to select
1819             SysRegSrc <= SysRegSrc_RegB;
1820
1821         elsif std_match(IR, LDC_L_RM_SYS) then
1822             -- LDC.L @Rm+, {SR, GBR, VBR}
1823             -- Uses bit decoding to choose the system register to load
1824
1825             if (std_match(IR, LDC_L_AT_RM_PLUS_GBR)) then
1826                 Instruction_GBRWriteEn <= '1';
1827             end if;
1828
1829             LogWithTime(l, "sh2_control.vhd: Decoded LDC.L @Rm+, X", LogFile);
1830
1831             -- Reads a longword from memory
1832             Instruction_MemEnable <= '1';             -- Uses memory.
1833             Instruction_ReadWrite <= ReadWrite_READ;  -- Reads.
1834             Instruction_WordMode  <= LongwordMode;    -- bit decode memory mode
1835
1836             -- Load into a system register
1837             Instruction_SysRegCtrl <= SysRegCtrl_LOAD;
1838             SysRegSel <= "0" & IR(5 downto 4);    -- bit decode which system register to write to
1839             SysRegSrc <= SysRegSrc_DB;      -- load new register value from memory
1840
1841             -- Read from @Rm, and save with post-incremented value
1842             RegA2Sel   <= to_integer(unsigned(m_format_m));
1843             RegAxInSel <= to_integer(unsigned(m_format_m));
1844             Instruction_RegAxStore <= '1';
1845
1846             -- DMAU signals (for post-increment indirect register addressing)
1847             BaseSel      <= BaseSel_REG;
1848             IndexSel     <= IndexSel_NONE;
```

```
1849             OffScalarSel <= OffScalarSel_FOUR;
1850             IncDecSel    <= IncDecSel_POST_INC;
1851
1852         elsif std_match(IR, LDS_RM_SYS) then
1853             -- LDS Rm, {MACH, MACL, PR}
1854             -- Uses bit decoding to choose the system register to load
1855
1856             -- Ensure that PR does actually get written to
1857             if (std_match(IR, LDS_RM_PR)) then
1858                 Instruction_PRWriteEn <= '1';
1859             end if;
1860
1861             LogWithTime(l, "sh2_control.vhd: Decoded LDS Rm, X", LogFile);
1862
1863             RegBSel <= to_integer(unsigned(m_format_m));
1864             Instruction_SysRegCtrl <= SysRegCtrl_LOAD;
1865             SysRegSel <= "1" & IR(5 downto 4);  -- bit decode register to select
1866             SysRegSrc <= SysRegSrc_RegB;
1867
1868         elsif std_match(IR, LDS_L_RM_SYS) then
1869             -- LDS.L @Rm+, {MACH, MACL, PR}
1870             -- Uses bit decoding to choose the system register to load
1871
1872             if (std_match(IR, LDS_L_AT_RM_PLUS_PR)) then
1873                 Instruction_PRWriteEn <= '1';
1874             end if;
1875
1876             LogWithTime(l, "sh2_control.vhd: Decoded LDS.L @Rm+, X", LogFile);
1877
1878             -- Reads a longword from memory
1879             Instruction_MemEnable <= '1';             -- Uses memory.
1880             Instruction_ReadWrite <= ReadWrite_READ;  -- Reads.
1881             Instruction_WordMode  <= LongwordMode;    -- bit decode memory mode
1882
1883             -- Load into a system register
1884             Instruction_SysRegCtrl <= SysRegCtrl_LOAD;
1885             SysRegSel <= "1" & IR(5 downto 4);    -- bit decode which system register to write to
1886             SysRegSrc <= SysRegSrc_DB;      -- load new register value from memory
1887
1888             -- Read from @Rm, and save with post-incremented value
1889             RegA2Sel   <= to_integer(unsigned(m_format_m));
1890             RegAxInSel <= to_integer(unsigned(m_format_m));
1891             Instruction_RegAxStore <= '1';
1892
1893             -- DMAU signals (for post-increment indirect register addressing)
1894             BaseSel      <= BaseSel_REG;
1895             IndexSel     <= IndexSel_NONE;
1896             OffScalarSel <= OffScalarSel_FOUR;
1897             IncDecSel    <= IncDecSel_POST_INC;
1898
1899         elsif std_match(IR, NOP) then
1900
1901             LogWithTime(l, "sh2_control.vhd: Decoded NOP", LogFile);
1902
1903         elsif not is_x(IR) then
1904             report "Unrecognized instruction: " & to_hstring(IR);
1905         end if;
1906
1907         end if;
1908
1909     end process;
1910
1911     -- Register updates done on clock edges. The state machine logic is also encoded here.
1912     -- Currently it is a repeating cycle of:
1913     --  - fetch: read the current instruction from ROM at address PC
1914     --  - execute: latch the instruction into IR and decode it, performing the necessary
```

```
1915     --             computations on this clock and also outputting read/write signals for memory access
1916     --  - writeback: update registers with computed values (or values read from memory).
1917     --               Also increment the PC to advance to the next instruction.
1918     state_proc: process (clock, reset)
1919     begin
1920         if reset = '0' then
1921             state <= fetch;
1922             IR <= NOP;
1923         elsif rising_edge(clock) then
1924             if state = fetch then
1925                 state <= execute;
1926                 IR <= MemDataIn(15 downto 0); -- latch in instruction from memory
1927             elsif state = execute then
1928                 state <= writeback;
1929             elsif state = writeback then
1930                 state <= fetch;
1931             end if;
1932         end if;
1933     end process state_proc;
1934
1935 end dataflow;
1936
```

```
1   ----------------------------------------------------------------------------
2   --
3   --  Hitachi SH-2 CPU
4   --
5   --  This file implements the complete SH-2 CPU, implemented for EE 188, Spring
6   --  term 2024-2025. The CPU supports the entirety of the SH-2 instruction set
7   --  except for MUL/DIV, MAC instructions, and other multi-clock instructions.
8   --  The CPU consists of a register array, arithmetic logic unit (ALU), program
9   --  memory access unit (PMAU), data memory access unit (DMAU), and memory
10  --  interfaces for input/output. This CPU contains sixteen 32-bit general
11  --  purpose registers, along with special registers such as PC, PR, GBR, VBR,
12  --  and SR. Each instruction is encoded in 16 bits, and memory can be accessed
13  --  either byte-wise, word-wise, or longword-wise. Currently, the CPU is not
14  --  pipelined and executes every instruction in three clocks.
15  --
16  --  Revision History:
17  --     28 Apr 25  Glen George       Initial revision.
18  --     01 May 25  Zack Huang        Declare all sub-unit entities
19  --     03 May 25  Zack Huang        Add state machine, test basic I/O
20  --     04 May 25  Zack Huang        Integrate memory interface
21  --     07 May 25  Chris Miranda     Change code formatting.
22  --     11 May 25  Zack Huang        Start system control instructions
23  --     12 May 25  Chris M.          Add extra RegDataIn sources and connect
24  --                                  PCSrc of DMAU.
25  --     14 May 25  Chris M.          Tri-state address in writeback state.
26  --     16 May 25  Zack Huang        Documentation, renaming signals
27  --     19 May 25  Chris M.          Connect R0Src to DMAU.
28  --     24 May 25  Chris M.          Add GBRIn mux.
29  --     25 May 25  Zack Huang        Finishing ALU and system instructions
30  --     26 May 25  Chris M.          Add internal signal for XTRCT instruction
31  --                                  register manipulation. Also added this
32  --                                  as possible input to RegDataIn.
33  --     01 Jun 25  Zack Huang        Finishing documentation
34  --
35  ----------------------------------------------------------------------------
36
37
38  --
39  --  SH2_CPU
40  --
41  --  This is the complete entity declaration for the SH-2 CPU.  It is used to
42  --  test the complete design.
43  --
44  --  Inputs:
45  --    Reset  - active low reset signal
46  --    NMI    - active falling edge non-maskable interrupt
47  --    INT    - active low maskable interrupt
48  --    clock  - the system clock
49  --
50  --  Outputs:
51  --    AB     - memory address bus (32 bits)
52  --    RE0    - first byte read signal, active low
53  --    RE1    - second byte read signal, active low
54  --    RE2    - third byte read signal, active low
55  --    RE3    - fourth byte read signal, active low
56  --    WE0    - first byte write signal, active low
57  --    WE1    - second byte write signal, active low
58  --    WE2    - third byte write signal, active low
59  --    WE3    - fourth byte write signal, active low
60  --
61  --  Inputs/Outputs:
62  --    DB     - memory data bus (32 bits)
63  --
64
65  library ieee;
66  use ieee.std_logic_1164.all;
```

```
67  use ieee.numeric_std.all;
68  use std.textio.all;
69
70  use work.SH2Constants.all;
71  use work.SH2ControlConstants.all;
72  use work.SH2PmauConstants.all;
73  use work.SH2DmauConstants.all;
74  use work.MemoryInterfaceConstants.all;
75
76
77  entity  SH2CPU  is
78
79      port (
80          Reset   :  in     std_logic;                      -- reset signal (active low)
81          NMI     :  in     std_logic;                      -- non-maskable interrupt signal (falling edge)
82          INT     :  in     std_logic;                      -- maskable interrupt signal (active low)
83          clock   :  in     std_logic;                      -- system clock
84          AB      :  out    std_logic_vector(31 downto 0);  -- memory address bus
85          MemSel  :  out    std_logic;                      -- whether to access data memory (0) or program memory (1)
86          RE0     :  out    std_logic;                      -- first byte active low read enable
87          RE1     :  out    std_logic;                      -- second byte active low read enable
88          RE2     :  out    std_logic;                      -- third byte active low read enable
89          RE3     :  out    std_logic;                      -- fourth byte active low read enable
90          WE0     :  out    std_logic;                      -- first byte active low write enable
91          WE1     :  out    std_logic;                      -- second byte active low write enable
92          WE2     :  out    std_logic;                      -- third byte active low write enable
93          WE3     :  out    std_logic;                      -- fourth byte active low write enable
94          DB      :  inout  std_logic_vector(31 downto 0)   -- memory data bus
95      );
96
97  end  SH2CPU;
98
99  architecture structural of sh2cpu is
100
101     -- Sign-extends a standard logic vector to the SH2 wordsize
102     pure function SignExtend(slv : std_logic_vector) return std_logic_vector is
103     begin
104       return std_logic_vector(resize(signed(slv), SH2_WORDSIZE));
105     end function;
106
107     -- Zero-extends a standard logic vector to the SH2 wordsize
108     pure function ZeroExtend(slv : std_logic_vector) return std_logic_vector is
109     begin
110       return std_logic_vector(resize(unsigned(slv), SH2_WORDSIZE));
111     end function;
112
113     -- Returns the low byte of a standard logic vector
114     pure function LowByte(slv : std_logic_vector) return std_logic_vector is
115     begin
116       assert slv'length >= 8
117       report "slv must be >= 8 bits."
118       severity ERROR;
119
120       return slv(7 downto 0);
121     end function;
122
123     -- Returns the low word of a standard logic vector
124     pure function LowWord(slv : std_logic_vector) return std_logic_vector is
125     begin
126       assert slv'length >= 16
127       report "slv must be >= 16 bits"
128       severity ERROR;
129
130       return slv(15 downto 0);
131     end function;
132
```

```
133
134     -- Register array inputs
135     signal RegDataIn  : std_logic_vector(31 downto 0);      -- data to write to a register
136     signal RegEnableIn: std_logic;                          -- if data should be written to an input register
137     signal RegInSel   : integer  range 15 downto 0;         -- which register to write data to
138     signal RegASel    : integer  range 15 downto 0;         -- which register to read to bus A
139     signal RegBSel    : integer  range 15 downto 0;         -- which register to read to bus B
140     signal RegAxIn    : std_logic_vector(31 downto 0);      -- data to write to an address register
141     signal RegAxInSel : integer  range 15 downto 0;         -- which address register to write to
142     signal RegAxStore : std_logic;                          -- if data should be written to the address register
143     signal RegA1Sel   : integer  range 15 downto 0;         -- which register to read to address bus 1
144     signal RegA2Sel   : integer  range 15 downto 0;         -- which register to read to address bus 2
145
146     -- register array outputs
147     signal RegA       : std_logic_vector(31 downto 0);      -- register bus A
148     signal RegB       : std_logic_vector(31 downto 0);      -- register bus B
149     signal RegA1      : std_logic_vector(31 downto 0);      -- address register bus 1
150     signal RegA2      : std_logic_vector(31 downto 0);      -- address register bus 2
151
152     -- ALU inputs
153     signal OperandA : std_logic_vector(31 downto 0);        -- first operand
154     signal OperandB : std_logic_vector(31 downto 0);        -- second operand
155     signal LoadA    : std_logic;                            -- determine if OperandA is loaded ('1') or zeroed ('0')
156     signal FCmd     : std_logic_vector(3 downto 0);         -- F-Block operation
157     signal CinCmd   : std_logic_vector(1 downto 0);         -- carry in operation
158     signal SCmd     : std_logic_vector(2 downto 0);         -- shift operation
159     signal ALUCmd   : std_logic_vector(1 downto 0);         -- ALU result select
160
161     -- ALU outputs
162     signal Result   : std_logic_vector(31 downto 0);        -- ALU result
163     signal Cout     : std_logic;                            -- carry out
164     signal Overflow : std_logic;                            -- signed overflow
165     signal Zero     : std_logic;                            -- result is zero
166     signal Sign     : std_logic;                            -- sign of result
167
168     -- DMAU inputs
169     signal RegSrc       : std_logic_vector(31 downto 0);    -- input register
170     signal R0Src        : std_logic_vector(31 downto 0);    -- R0 register input
171     signal PCSrc        : std_logic_vector(31 downto 0);    -- program counter source
172     signal GBRIn        : std_logic_vector(31 downto 0);    -- GBR input
173     signal GBRWriteEn   : std_logic;                        -- GBR write enable, active high
174     signal DMAUOff4     : std_logic_vector(3 downto 0);     -- 4-bit offset
175     signal DMAUOff8     : std_logic_vector(7 downto 0);     -- 8-bit offset
176     signal BaseSel      : std_logic_vector(1 downto 0);     -- which base register source to select
177     signal IndexSel     : std_logic_vector(1 downto 0);     -- which index source to select
178     signal OffScalarSel : std_logic_vector(1 downto 0);     -- what to scale the offset by (1, 2, 4)
179     signal IncDecSel    : std_logic_vector(1 downto 0);     -- post-increment or pre-decrement the base
180
181     -- DMAU outputs
182     signal DataAddress  : std_logic_vector(31 downto 0);    -- output address
183     signal AddrSrcOut   : std_logic_vector(31 downto 0);    -- incremented/decremented address (store back to register)
184     signal GBROut       : std_logic_vector(31 downto 0);    -- GBR output
185                                                             --
186     -- PMAU inputs
187     signal RegIn       : std_logic_vector(31 downto 0);     -- register source input
188     signal PRIn        : std_logic_vector(31 downto 0);     -- PR register input (for writing to PR)
189     signal PRWriteEn   : std_logic;                         -- enable writing to PR (active high)
190     signal PCAddrMode  : std_logic_vector(2 downto 0);      -- program address mode select signal
191     signal PMAUOff8    : std_logic_vector(7 downto 0);      -- 8-bit signed offset input
192     signal PMAUOff12   : std_logic_vector(11 downto 0);     -- 12-bit signed offset input
193     signal PCIn        : std_logic_vector(31 downto 0);     -- input for parallel loading of PC
194     signal PCWriteCtrl : std_logic_vector(1 downto 0);      -- write control signal for PC
195
196     -- PMAU outputs
197     signal PCCalcOut : std_logic_vector(31 downto 0);       -- calculated PC address output
198     signal PCRegOut  : std_logic_vector(31 downto 0);       -- currenet value of the PC register
```

```
199     signal PROut     : std_logic_vector(31 downto 0);       -- PR (procedure register) output
200
201     -- Memory interface inputs
202     signal MemEnable   : std_logic;                         -- memory interface enable (active high)
203     signal ReadWrite   : std_logic;                         -- if reading or writing from memory
204     signal MemMode     : std_logic_vector(1 downto 0);      -- memory access mode (byte, word, or longword)
205     signal MemAddress  : std_logic_vector(31 downto 0);     -- memory address bus (MUST BE ALIGNED)
206     signal MemDataOut  : std_logic_vector(31 downto 0);     -- the data to output to memory
207
208     -- Memory interface outputs
209     signal ReadMask    : std_logic_vector(3 downto 0);      -- read enable mask (active low)
210     signal WriteMask   : std_logic_vector(3 downto 0);      -- write enable mask (active low)
211     signal MemDataIn   : std_logic_vector(31 downto 0);     -- the data read in from memory
212
213     -- CPU internal control signals
214     signal MemOutSel        : std_logic_vector(2 downto 0);     -- source for data that should be output to memory
215     signal RegDataInSel     : std_logic_vector(3 downto 0);     -- source for register input data
216     signal TFlagSel         : std_logic_vector(2 downto 0);     -- source for next value of T flag
217     signal Immediate        : std_logic_vector(7 downto 0);     -- immediate value from instruction
218     signal ImmediateMode    : std_logic;                        -- immediate extension mode (zero or signed)
219     signal ALUOpBSel        : std_logic;                        -- source for ALU Operand B
220     signal SysRegCtrl       : std_logic_vector(1 downto 0);     -- how to update system registers
221     signal SysRegSel        : std_logic_vector(2 downto 0);     -- system register select
222     signal SysRegSrc        : std_logic_vector(1 downto 0);     -- source for data to input into a system register
223     signal TNext            : std_logic;                        -- Next value for T bit
224     signal ExtMode          : std_logic_vector(1 downto 0);     -- mode for extending register value (zero or signed)
225     signal MemAddrSel       : std_logic;                        -- whether to output PMAU or DMAU address
226     signal TCmpSel          : std_logic_vector(2 downto 0);     -- how to compute T from ALU status flags
227     signal DelayedBranchTaken   : std_logic;                    -- Whether the delayed branch is taken or not.
228
229     -- CPU system/control registers
230     signal SR               : std_logic_vector(31 downto 0);    -- Status register.
231     signal VBR              : std_logic_vector(31 downto 0);    -- Vector Base Register.
232     signal MACL             : std_logic_vector(31 downto 0);    -- Multiply and Accumulate Low.
233     signal MACH             : std_logic_vector(31 downto 0);    -- Multiply and Accumulate High.
234
235     -- Aliases for status register bits.
236     alias MBit  : std_logic is SR(9);   -- The M and Q bits are used by DIVOU/S and DIV1 instructions.
237     alias QBit  : std_logic is SR(8);   -- ...
238     alias I3Bit : std_logic is SR(7);   -- Interrupt mask bits.
239     alias I2Bit : std_logic is SR(6);   -- ...
240     alias I1Bit : std_logic is SR(5);   -- ...
241     alias I0BIt : std_logic is SR(4);   -- ...
242     alias SBit  : std_logic is SR(1);   -- Used by MAC instructions.
243     alias TBit  : std_logic is SR(0);   -- True flag.
244
245     -- Intermediate terms
246     signal ExtendedReg      : std_logic_vector(31 downto 0);    -- extended register value (for EXT* instructions)
247     signal NextSysReg       : std_logic_vector(31 downto 0);    -- data to input into a system register
248     signal ImmediateExt     : std_logic_vector(31 downto 0);    -- sign-extended immediate
249     signal TCmp             : std_logic;                        -- The value of T generated from a compare
250     signal StrCmp           : std_logic;                        -- used to compare the bytes of RegA and RegB
251     signal SysReg           : std_logic_vector(31 downto 0);    -- Value of selected system/control register
252     signal PCNext           : std_logic_vector(31 downto 0);    -- The current PC incremented by two.
253     signal PrevPCReg        : std_logic_vector(31 downto 0);    -- the prevous value fo PC
254     signal PCUsed           : std_logic_vector(31 downto 0);    -- The PC that will be fetched from program memory.
255
256     -- RegA with the upper and lower halves of the low two bytes swapped (for the SWAP.B instruction).
257     signal RegASwapB : std_logic_vector(31 downto 0);
258
259     -- RegA with the high and low words swapped (for the SWAP.W instruction).
260     signal RegASwapW : std_logic_vector(31 downto 0);
261
262     -- The center 32-bits of RegB and RegA (ie, low word of RegB and high word of RegA).
263     signal RegB_RegA_Center : std_logic_vector(31 downto 0);
264
```

```
265 begin
266
267     -- Output read enable signals when clock is low
268     RE0 <= ReadMask(0) when (not clock) else '1';
269     RE1 <= ReadMask(1) when (not clock) else '1';
270     RE2 <= ReadMask(2) when (not clock) else '1';
271     RE3 <= ReadMask(3) when (not clock) else '1';
272
273     -- Output write enable signals when clock is low
274     WE0 <= WriteMask(0) when (not clock) else '1';
275     WE1 <= WriteMask(1) when (not clock) else '1';
276     WE2 <= WriteMask(2) when (not clock) else '1';
277     WE3 <= WriteMask(3) when (not clock) else '1';
278
279     StorePCReg : process(clock)
280     begin
281         if rising_edge(clock) then
282             -- Store previous PC on rising clock
283             PrevPCReg <= PCRegOut;
284         end if;
285     end process;
286
287     -- The "next" PC is the current value of the PC register (NOT PCCalcOut) + 2.
288     PCNext <= std_logic_vector(unsigned(PrevPCReg) + to_unsigned(2, 32));
289
290     -- Decide which value of PC should be used
291     PCUsed <= PCNext when (DelayedBranchTaken = '1') else PCCalcOut;
292
293     -- Decide which memory address to output
294     with MemAddrSel select
295         MemAddress <=  PCUsed           when MemAddrSel_PMAU,
296                        DataAddress      when MemAddrSel_DMAU,
297                        (others => 'Z')  when others;
298
299     AB <= MemAddress;   -- Output memory address to the address bus
300
301     -- What to output to the data bus (to be written to an address).
302     with MemOutSel select
303         MemDataOut <=   RegA            when MemOut_RegA,
304                         RegB            when MemOut_RegB,
305                         SysReg          when MemOut_SysReg,
306                         (others => 'Z') when others;
307
308     -- Compute the zero/sign-extended immediate from an instruction
309     ImmediateExt(7 downto 0) <= Immediate;
310     with ImmediateMode select
311         ImmediateExt(31 downto 8) <= (others => Immediate(7)) when ImmediateMode_SIGN,
312                                      (others => '0')          when ImmediateMode_ZERO,
313                                      (others => 'X')          when others;
314
315     -- RegA with the high and low bytes swapped.
316     RegASwapB <= RegA(31 downto 16) & RegA(7 downto 0) & RegA(15 downto 8);
317
318     -- RegA with the high and low words swapped.
319     RegASwapW <= RegA(15 downto 0) & RegA(31 downto 16);
320
321     -- The center 32-bits of RegB and RegA (ie, low word of RegB and high word of RegA).
322     RegB_RegA_Center <= RegB(15 downto 0) & RegA(31 downto 16);
323
324     -- the zero/sign-extended version of register B
325     with ExtMode select
326         ExtendedReg <= SignExtend(LowByte(RegB))  when  Ext_Sign_B_RegA,
327                        SignExtend(LowWord(RegB))  when  Ext_Sign_W_RegA,
328                        ZeroExtend(LowByte(RegB))  when  Ext_Zero_B_RegA,
329                        ZeroExtend(LowWord(RegB))  when  Ext_Zero_W_RegA,
330                        (others => 'X') when others;
```

```
331
332     -- Choose which system register to select
333     with SysRegSel select
334         SysReg <= SR      when SysRegSel_SR,
335                   GBROut  when SysRegSel_GBR,
336                   VBR     when SysRegSel_VBR,
337                   MACH    when SysRegSel_MACH,
338                   MACL    when SysRegSel_MACL,
339                   PROut   when SysRegSel_PR,
340                   (others => 'X') when others;
341
342     -- Select the data to write the the register based on the decoded instruction.
343     with RegDataInSel select
344         RegDataIn <= Result                     when  RegDataIn_ALUResult,
345                      ImmediateExt               when  RegDataIn_Immediate,
346                      RegA                       when  RegDataIn_RegA,
347                      RegB                       when  RegDataIn_RegB,
348                      SysReg                     when  RegDataIn_SysReg,
349                      RegASwapB                  when  RegDataIn_RegA_SWAP_B,
350                      RegASwapW                  when  RegDataIn_RegA_SWAP_W,
351                      RegB_RegA_Center           when  RegDataIn_REGB_REGA_CENTER,
352                      ExtendedReg                when  RegDataIn_Ext,
353                      MemDataIn                  when  RegDataIn_DB,
354
355                      -- Extract the T bit from the status register.
356                      SR and X"00000001"         when  RegDataIn_SR_TBit,
357                      PROut                      when  RegDataIn_PR,
358                      (others => 'X')            when  others;
359
360
361     -- The address being stored to a register is the pre-decremented or
362     -- post-incremented address when we are in that mode. If we are not in
363     -- that mode, it is just the normal address.
364     with IncDecSel select
365         RegAxIn <= AddrSrcOut  when IncDecSel_PRE_DEC | IncDecSel_POST_INC,
366                    DataAddress when others;
367
368     -- Route control signals and data into register array
369     registers : entity work.SH2Regs
370     port map (
371         -- Inputs:
372         clock       => clock,
373         reset       => reset,
374         RegDataIn   => RegDataIn,
375         EnableIn    => RegEnableIn,
376         RegInSel    => RegInSel,
377         RegASel     => RegASel,
378         RegBSel     => RegBSel,
379         RegAxIn     => RegAxIn,
380         RegAxInSel  => RegAxInSel,
381         RegAxStore  => RegAxStore,
382         RegA1Sel    => RegA1Sel,
383         RegA2Sel    => RegA2Sel,
384         -- Outputs:
385         RegA    => RegA,
386         RegB    => RegB,
387         RegA1   => RegA1,
388         RegA2   => RegA2
389     );
390
391
392     -- Always use RegA as Operand A for ALU
393     OperandA <= RegA;
394
395     -- Input mux for ALU Operand B
396     with ALUOpBSel select
```

```
397         OperandB <=  RegB            when ALUOpB_RegB,
398                      ImmediateExt    when ALUOpB_Imm,
399                      (others => 'X') when others;
400
401     -- If two registers share a byte value
402     StrCmp <= '1' when (RegA(31 downto 24) = RegB(31 downto 24)) or
403                         (RegA(23 downto 16) = RegB(23 downto 16)) or
404                         (RegA(15 downto 8)  = RegB(15 downto 8))  or
405                         (RegA(7 downto 0)   = RegB(7 downto 0))
406                   else '0';
407
408     -- Compute T flag value based on ALU output flags. Used for operations
409     -- of the form CMP/XX.
410     with TCmpSel select
411         TCmp <= Zero                                when TCMP_EQ,   -- 1 if Rn = Rm
412                 Cout                                when TCMP_HS,   -- 1 if Rn >= Rm, unsigned
413                 not (Sign xor Overflow)             when TCMP_GE,   -- 1 if Rn >= Rm, signed
414                 Cout and (not Zero)                 when TCMP_HI,   -- 1 if Rn >  Rm, unsigned
415                 not ((Sign xor Overflow) or Zero)   when TCMP_GT,   -- 1 if Rn >  Rm, signed
416                 StrCmp                              when TCMP_STR,  -- 1 if Rn byte matches Rm byte
417                 'X' when others;
418
419     -- Select what value T should be set to
420     with TFlagSel select
421         TNext <=  TBit       when TFlagSel_T,           -- retain T flag
422                   Cout       when TFlagSel_Carry,       -- Set T flag to ALU carry flag
423                   Overflow   when TFlagSel_Overflow,    -- Set T flag to ALU overflow flag
424                   Zero       when TFlagSel_Zero,        -- set T flag to ALU Zero flag
425                   '0'        when TFlagSel_CLEAR,       -- clear T flag
426                   '1'        when TFlagSel_SET,         -- set T flag
427                   TCmp       when TFlagSel_CMP,         -- compute T flag based on compare result
428                   'X'        when others;
429
430     -- Route ALU control signals
431     alu : entity work.sh2alu
432     port map (
433         -- Inputs:
434         OperandA => OperandA,
435         OperandB => OperandB,
436         TIn      => TBit,
437         LoadA    => LoadA,
438         FCmd     => FCmd,
439         CinCmd   => CinCmd,
440         SCmd     => SCmd,
441         ALUCmd   => ALUCmd,
442         -- Outputs:
443         Result   => Result,
444         Cout     => Cout,
445         Overflow => Overflow,
446         Zero     => Zero,
447         Sign     => Sign
448     );
449
450     -- Use RegA1 (@Rn) if we are writing and RegA2 (@Rm) if we are reading.
451     with ReadWrite select
452         RegSrc <= RegA2          when Mem_READ,
453                   RegA1          when Mem_WRITE,
454                   (others => 'X') when others;
455
456     -- Connect PCSrc to PCUsed
457     PCSrc <= PCUsed;
458
459     -- R0 comes from RegA2 when we are reading and RegA1 when we are writing.
460     with ReadWrite select
461         R0Src <= RegA1            when Mem_READ,
462                  RegA2            when Mem_WRITE,
```

```
463                 (others => 'X')  when others;
464
465     -- Route DMAU control signals
466     dmau : entity work.sh2dmau
467     port map (
468         -- Inputs:
469         RegSrc       => RegSrc,
470         R0Src        => R0Src,
471         PCSrc        => PCSrc,
472         GBRIn        => GBRIn,
473         GBRWriteEn   => GBRWriteEn,
474         Off4         => DMAUOff4,
475         Off8         => DMAUOff8,
476         BaseSel      => BaseSel,
477         IndexSel     => IndexSel,
478         OffScalarSel => OffScalarSel,
479         IncDecSel    => IncDecSel,
480         Clk          => clock,
481         -- Outputs:
482         Address    => DataAddress,
483         AddrSrcOut => AddrSrcOut,
484         GBROut     => GBROut
485     );
486
487
488     -- Default PC in is all zeroes.
489     PCIn <= (others => '0');
490
491     -- PMAU Register input is always RegB.
492     RegIn <= RegB;
493
494     -- Route PMAU control signals
495     pmau : entity work.sh2pmau
496     port map (
497         -- Inputs:
498         RegIn       => RegIn,
499         PRIn        => PRIn,
500         PRWriteEn   => PRWriteEn,
501         PCIn        => PCIn,
502         PCWriteCtrl => PCWriteCtrl,
503         Off8        => PMAUOff8,
504         Off12       => PMAUOff12,
505         PCAddrMode  => PCAddrMode,
506         Clk         => clock,
507         Reset       => Reset,
508         -- Outputs:
509         PCRegOut    => PCRegOut,
510         PCCalcOut   => PCCalcOut,
511         PROut       => PROut
512     );
513
514     -- Route memory interface control signals
515     memory_tx : entity work.MemoryInterfaceTx
516     port map (
517         -- Inputs:
518         clock      => clock,
519         MemEnable  => MemEnable,
520         ReadWrite  => ReadWrite,
521         MemMode    => MemMode,
522         Address    => unsigned(MemAddress),
523         MemDataOut => MemDataOut,
524         -- Outputs:
525         RE => ReadMask,
526         WE => WriteMask,
527         DB => DB
528     );
```

```
529
530     -- Route memory interface control signals
531     memory_rx : entity work.MemoryInterfaceRx
532     port map (
533         -- Inputs:
534         MemEnable => MemEnable,
535         MemMode => MemMode,
536         Address => unsigned(MemAddress),
537         DB      => DB,
538         -- Outputs:
539         MemDataIn => MemDataIn
540     );
541
542     -- Route control unit control signals
543     control_unit : entity work.SH2Control
544     port map (
545         -- Inputs:
546         MemDataIn   => MemDataIn,
547         TFlagIn     => TBit,
548         clock       => clock,
549         reset       => reset,
550
551         -- Outputs:
552         Immediate       => Immediate,
553         ImmediateMode   => ImmediateMode,
554         TFlagSel        => TFlagSel,
555         ExtMode         => ExtMode,
556
557         -- Memory interface control signals:
558         MemEnable    => MemEnable,
559         ReadWrite    => ReadWrite,
560         MemMode      => MemMode,
561         MemSel       => MemSel,
562         MemOutSel    => MemOutSel,
563         MemAddrSel   => MemAddrSel,
564
565         -- ALU control signals:
566         ALUOpBSel    => ALUOpBSel,
567         LoadA        => LoadA,
568         FCmd         => FCmd,
569         CinCmd       => CinCmd,
570         SCmd         => SCmd,
571         ALUCmd       => ALUCmd,
572         TCmpSel      => TCmpSel,
573
574         -- Register Array control signals:
575         RegDataInSel    => RegDataInSel,
576         RegEnableIn     => RegEnableIn,
577         RegInSel        => RegInSel,
578         RegASel         => RegASel,
579         RegBSel         => RegBSel,
580         RegAxIn         => RegAxIn,
581         RegAxInSel      => RegAxInSel,
582         RegAxStore      => RegAxStore,
583         RegA1Sel        => RegA1Sel,
584         RegA2Sel        => RegA2Sel,
585
586         -- DMAU control signals:
587         GBRWriteEn      => GBRWriteEn,
588         DMAUOff4        => DMAUOff4,
589         DMAUOff8        => DMAUOff8,
590         BaseSel         => BaseSel,
591         IndexSel        => IndexSel,
592         OffScalarSel    => OffScalarSel,
593         IncDecSel       => IncDecSel,
594
```

```
595         -- PMAU control signals:
596         PCAddrMode   => PCAddrMode,
597         PRWriteEn    => PRWriteEn,
598         PCWriteCtrl  => PCWriteCtrl,
599         PCIn         => PCIn,
600         PMAUOff8     => PMAUOff8,
601         PMAUOff12    => PMAUOff12,
602 
603         -- system control signals
604         SysRegCtrl  => SysRegCtrl,
605         SysRegSel   => SysRegSel,
606         SysRegSrc   => SysRegSrc,
607 
608         -- Branch control signals.
609         DelayedBranchTaken => DelayedBranchTaken
610     );
611 
612     -- Mux system register input values based on SysRegSrc. Note that individual
613     -- write-enables (like GBRWriteEn and PRWriteEn) must be enabled seperately.
614     NextSysReg <= RegB      when SysRegSrc = SysRegSrc_RegB  else
615                   MemDataIn when SysRegSrc = SysRegSrc_DB    else
616 
617                   -- The return address of a BSR is the PC at the point of decoding the BSR plus 4.
618                   std_logic_vector(unsigned(PCRegOut) + to_unsigned(4, 32))  when SysRegSrc = SysRegSrc_PC    else
619 
620                   (others => 'X');
621 
622     GBRIn <= NextSysReg;    -- set GBR to selected sysreg value (when GBRWriteEn active)
623     PRIn  <= NextSysReg;    -- set PR to selected sysreg value (when PRWriteEn active)
624 
625     register_proc: process(clock, reset)
626     begin
627         if reset = '0' then
628             -- Reset system registers (async)
629             SR   <=  (others => '0');
630             VBR  <=  (others => '0');
631             MACH <=  (others => '0');
632             MACL <=  (others => '0');
633 
634         elsif rising_edge(clock) then
635             SR(0) <= TNext;     -- set new value of T
636 
637             if SysRegCtrl = SysRegCtrl_LOAD then
638                 -- Load new value into a system register
639                 -- (note that PR and GBR are handled separately)
640                 if SysRegSel = SysRegSel_SR then
641                     SR <= NextSysReg;
642                 elsif SysRegSel = SysRegSel_VBR then
643                     VBR <= NextSysReg;
644                 elsif SysRegSel = SysRegSel_MACH then
645                     MACH <= NextSysReg;
646                 elsif SysRegSel = SysRegSel_MACL then
647                     MACL <= NextSysReg;
648                 end if;
649             elsif SysRegCtrl = SysRegCtrl_CLEAR then
650                 -- Clear a system register value
651                 -- (note that PR and GBR are handled separately)
652                 if SysRegSel = SysRegSel_SR then
653                     SR <= (others => '0');
654                 elsif SysRegSel = SysRegSel_VBR then
655                     VBR <= (others => '0');
656                 elsif (SysRegSel = SysRegSel_MACH) or (SysRegSel = SysRegSel_MACL) then
657                     -- Reset both MACH and MACL for CLRMAC instruction
658                     MACH <= (others => '0');
659                     MACL <= (others => '0');
660                 end if;
```

```
661             end if;
662         end if;
663     end process register_proc;
664
665 end architecture structural;
666
```

```
1   --------------------------------------------------------------------------------
2   --
3   --  SH2 CPU Testbench
4   --
5   --  This file contains the full testbench for the SH2 CPU, implemented for EE
6   --  188, Spring term 2024-2025. We instantiate the CPU itself along with two
7   --  memory units for program memory (ROM) and data memory (RAM). The control
8   --  signals for these memory units are muxed between the CPU and the testbench
9   --  itself, allowing us to modify/read memory as part of the tests and then
10  --  make this memory available to the CPU for simulation. For testing, we are
11  --  using real SH2 programs assembled using the AS Macroassembler, which we
12  --  load into ROM for the CPU to access. Then, we run the program and then
13  --  check the contents of RAM after to see if they match up with the expected
14  --  values (written in ".expect" files). Test results are printed to the
15  --  console, while logging is output to "log.txt".
16  --
17  --  Revision History:
18  --     01 May 25    Zack Huang      initial revision
19  --     03 May 25    Zack Huang      working with data/program memory units
20  --     01 Jun 25    Zack Huang      cleaning up code, finish documentation
21  --
22  --------------------------------------------------------------------------------
23
24
25  library ieee;
26  use ieee.std_logic_1164.all;
27  use ieee.numeric_std.all;
28  use ieee.math_real.all;
29  use std.textio.all;
30
31  use work.Logging.all;
32  use work.ANSIEscape.all;
33  use work.SH2ControlConstants.all;
34
35  entity sh2_cpu_tb is
36  end sh2_cpu_tb;
37
38  architecture behavioral of sh2_cpu_tb is
39
40      -- Stimulus signals for unit under test
41      signal Reset   :  std_logic;                      -- reset signal (active low)
42      signal NMI     :  std_logic;                      -- non-maskable interrupt signal (falling edge)
43      signal INT     :  std_logic;                      -- maskable interrupt signal (active low)
44      signal clock   :  std_logic;                      -- system clock
45
46      -- Outputs from unit under test
47      signal CPU_AB  :  std_logic_vector(31 downto 0);   -- program memory address bus
48      signal CPU_RE0     :  std_logic;                      -- first byte active low read enable
49      signal CPU_RE1     :  std_logic;                      -- second byte active low read enable
50      signal CPU_RE2     :  std_logic;                      -- third byte active low read enable
51      signal CPU_RE3     :  std_logic;                      -- fourth byte active low read enable
52      signal CPU_WE0     :  std_logic;                      -- first byte active low write enable
53      signal CPU_WE1     :  std_logic;                      -- second byte active low write enable
54      signal CPU_WE2     :  std_logic;                      -- third byte active low write enable
55      signal CPU_WE3     :  std_logic;                      -- fourth byte active low write enable
56      signal CPU_DB      :  std_logic_vector(31 downto 0);   -- memory data bus
57      signal CPU_MEMSEL  :  std_logic;                      -- if should access data memory (0) or program memory (1)
58
59      -- test signals used to read/write the RAM independently of the CPU
60      signal TEST_AB      :  std_logic_vector(31 downto 0);   -- memory address bus
61      signal TEST_RE0     :  std_logic;                       -- first byte active low read enable
62      signal TEST_RE1     :  std_logic;                       -- second byte active low read enable
63      signal TEST_RE2     :  std_logic;                       -- third byte active low read enable
64      signal TEST_RE3     :  std_logic;                       -- fourth byte active low read enable
65      signal TEST_WE0     :  std_logic;                       -- first byte active low write enable
66      signal TEST_WE1     :  std_logic;                       -- second byte active low write enable
```

```
67      signal TEST_WE2     :  std_logic;                      -- third byte active low write enable
68      signal TEST_WE3     :  std_logic;                      -- fourth byte active low write enable
69      signal TEST_DB      :  std_logic_vector(31 downto 0);  -- memory data bus
70      signal TEST_MEMSEL  :  std_logic;                      -- if should access data memory (0) or program memory (1)
71
72      -- Memory control signals
73      signal RAM_RE0     :  std_logic;                     -- first byte active low read enable
74      signal RAM_RE1     :  std_logic;                     -- second byte active low read enable
75      signal RAM_RE2     :  std_logic;                     -- third byte active low read enable
76      signal RAM_RE3     :  std_logic;                     -- fourth byte active low read enable
77      signal RAM_WE0     :  std_logic;                     -- first byte active low write enable
78      signal RAM_WE1     :  std_logic;                     -- second byte active low write enable
79      signal RAM_WE2     :  std_logic;                     -- third byte active low write enable
80      signal RAM_WE3     :  std_logic;                     -- fourth byte active low write enable
81      signal RAM_DB      :  std_logic_vector(31 downto 0);  -- data memory data bus
82      signal RAM_AB      :  std_logic_vector(31 downto 0);  -- data memory address bus
83
84      signal ROM_RE0     :  std_logic;                     -- first byte active low read enable
85      signal ROM_RE1     :  std_logic;                     -- second byte active low read enable
86      signal ROM_RE2     :  std_logic;                     -- third byte active low read enable
87      signal ROM_RE3     :  std_logic;                     -- fourth byte active low read enable
88      signal ROM_WE0     :  std_logic;                     -- first byte active low write enable
89      signal ROM_WE1     :  std_logic;                     -- second byte active low write enable
90      signal ROM_WE2     :  std_logic;                     -- third byte active low write enable
91      signal ROM_WE3     :  std_logic;                     -- fourth byte active low write enable
92      signal ROM_DB      :  std_logic_vector(31 downto 0);  -- program memory data bus
93      signal ROM_AB      :  std_logic_vector(31 downto 0);  -- program memory address bus
94
95      signal CPU_ACTIVE  :  boolean    := false;   -- if the cpu outputs or test signals should be routed into the memory units
96
97      signal CPU_RD      :  std_logic;
98      signal CPU_WR      :  std_logic;
99      signal TEST_RD     :  std_logic;
100     signal TEST_WR     :  std_logic;
101
102 begin
103
104     -- We initialize two memory units: one called RAM for data memory, and one
105     -- called ROM for program memory. We enforce that the CPU is not able to
106     -- write to ROM. To allow both the CPU and test bench to communicate with
107     -- the memory units, we fully mux every control signal to/from the two
108     -- memory units. When CPU_ACTIVE is true, all of memory control signals are
109     -- routed between the CPU and memory. When CPU_ACTIVE is false, all of the
110     -- memory control signals are routed to/from the testbench signals.
111
112     CPU_RD  <= CPU_RE0 and CPU_RE1 and CPU_RE2 and CPU_RE3;         -- CPU read signal, active low
113     TEST_RD <= TEST_RE0 and TEST_RE1 and TEST_RE2 and TEST_RE3;     -- testbench read signal, active low
114
115     CPU_WR  <= CPU_WE0 and CPU_WE1 and CPU_WE2 and CPU_WE3;         -- CPU write signal, active low
116     TEST_WR <= TEST_WE0 and TEST_WE1 and TEST_WE2 and TEST_WE3;     -- testbench write signal, active low
117
118     -- Muxing between the CPU address bus and testbench address bus based on CPU_ACTIVE
119     ROM_AB <= CPU_AB when CPU_ACTIVE else TEST_AB;
120
121     -- Muxing between the CPU address bus and testbench address bus based on CPU_ACTIVE
122     RAM_AB <= CPU_AB when CPU_ACTIVE else TEST_AB;
123
124     -- Muxing between the CPU data bus and testbench data bus based on
125     -- CPU_ACTIVE and if the memory unit is being selected. Set the data bus to
126     -- high impedance if not being used.
127     ROM_DB <= CPU_DB  when     CPU_ACTIVE and CPU_WR  = '0' and CPU_MEMSEL  = MEMSEL_ROM  else
128               TEST_DB when not CPU_ACTIVE and TEST_WR = '0' and TEST_MEMSEL = MEMSEL_ROM else
129               (others => 'Z');
130
131     -- Muxing between the CPU data bus and testbench data bus based on
132     -- CPU_ACTIVE and if the memory unit is being selected. Set the data bus to
```

```
133     -- high impedance if not being used.
134     RAM_DB <= CPU_DB  when     CPU_ACTIVE and CPU_WR  = '0' and CPU_MEMSEL  = MEMSEL_RAM  else
135               TEST_DB when not CPU_ACTIVE and TEST_WR = '0' and TEST_MEMSEL = MEMSEL_RAM else
136               (others => 'Z');
137
138     -- Muxing between the RAM and ROM data bus based on the selected memory.
139     -- Set the data bus to high impedance if not being used.
140     CPU_DB <= RAM_DB when CPU_MEMSEL = MEMSEL_RAM and CPU_RD = '0' else
141               ROM_DB when CPU_MEMSEL = MEMSEL_ROM and CPU_RD = '0' else
142               (others => 'Z');
143
144     -- Muxing between the RAM and ROM data bus based on the selected memory.
145     -- Set the data bus to high impedance if not being used.
146     TEST_DB <= RAM_DB when TEST_MEMSEL = '0' and TEST_RD = '0' else
147                ROM_DB when TEST_MEMSEL = '1' and TEST_RD = '0' else
148               (others => 'Z');
149
150     -- Mux the write-enable signals between the CPU and testbench signals based on CPU_ACTIVE
151     -- and if the memory unit is currently selected. Note that we ignore the CPU control
152     -- signals in this case since we don't want ROM to be writeable by the CPU.
153     ROM_WE0 <= '1' when CPU_ACTIVE else TEST_WE0 when TEST_MEMSEL = '1' else '1';
154     ROM_WE1 <= '1' when CPU_ACTIVE else TEST_WE1 when TEST_MEMSEL = '1' else '1';
155     ROM_WE2 <= '1' when CPU_ACTIVE else TEST_WE2 when TEST_MEMSEL = '1' else '1';
156     ROM_WE3 <= '1' when CPU_ACTIVE else TEST_WE3 when TEST_MEMSEL = '1' else '1';
157
158     -- Mux the read-enable signals between the CPU and testbench signals based on CPU_ACTIVE
159     -- and if the memory unit is currently selected
160     ROM_RE0 <= CPU_RE0  when CPU_ACTIVE and CPU_MEMSEL = '1' else
161                TEST_RE0 when not CPU_ACTIVE and TEST_MEMSEL = '1' else
162                '1';
163
164     ROM_RE1 <= CPU_RE1  when CPU_ACTIVE and CPU_MEMSEL = '1' else
165                TEST_RE1 when not CPU_ACTIVE and TEST_MEMSEL = '1' else
166                '1';
167
168     ROM_RE2 <= CPU_RE2  when CPU_ACTIVE and CPU_MEMSEL = '1' else
169                TEST_RE2 when not CPU_ACTIVE and TEST_MEMSEL = '1' else
170                '1';
171
172     ROM_RE3 <= CPU_RE3  when CPU_ACTIVE and CPU_MEMSEL = '1' else
173                TEST_RE3 when not CPU_ACTIVE and TEST_MEMSEL = '1' else
174                '1';
175
176     -- Mux the write-enable signals between the CPU and testbench signals based on CPU_ACTIVE
177     -- and if the memory unit is currently selected
178     RAM_WE0 <= CPU_WE0 when CPU_ACTIVE and CPU_MEMSEL = '0' else
179                TEST_WE0 when not CPU_ACTIVE and TEST_MEMSEL = '0' else
180                '1';
181
182     RAM_WE1 <= CPU_WE1 when CPU_ACTIVE and CPU_MEMSEL = '0' else
183                TEST_WE1 when not CPU_ACTIVE and TEST_MEMSEL = '0' else
184                '1';
185
186     RAM_WE2 <= CPU_WE2 when CPU_ACTIVE and CPU_MEMSEL = '0' else
187                TEST_WE2 when not CPU_ACTIVE and TEST_MEMSEL = '0' else
188                '1';
189
190     RAM_WE3 <= CPU_WE3 when CPU_ACTIVE and CPU_MEMSEL = '0' else
191                TEST_WE3 when not CPU_ACTIVE and TEST_MEMSEL = '0' else
192                '1';
193
194     -- Mux the read-enable signals between the CPU and testbench signals based on CPU_ACTIVE
195     -- and if the memory unit is currently selected
196     RAM_RE0 <= CPU_RE0 when CPU_ACTIVE and CPU_MEMSEL = '0' else
197                TEST_RE0 when not CPU_ACTIVE and TEST_MEMSEL = '0' else
198                '1';
```

```
199
200     RAM_RE1 <= CPU_RE1 when CPU_ACTIVE and CPU_MEMSEL = '0' else
201                TEST_RE1 when not CPU_ACTIVE and TEST_MEMSEL = '0' else
202                '1';
203
204     RAM_RE2 <= CPU_RE2 when CPU_ACTIVE and CPU_MEMSEL = '0' else
205                TEST_RE2 when not CPU_ACTIVE and TEST_MEMSEL = '0' else
206                '1';
207
208     RAM_RE3 <= CPU_RE3 when CPU_ACTIVE and CPU_MEMSEL = '0' else
209                TEST_RE3 when not CPU_ACTIVE and TEST_MEMSEL = '0' else
210                '1';
211
212     -- Instantiate UUT
213     UUT: entity work.sh2cpu
214     port map (
215         Reset => Reset,
216         NMI => NMI,
217         INT => INT,
218         clock => clock,
219         AB => CPU_AB,
220         RE0 => CPU_RE0,
221         RE1 => CPU_RE1,
222         RE2 => CPU_RE2,
223         RE3 => CPU_RE3,
224         WE0 => CPU_WE0,
225         WE1 => CPU_WE1,
226         WE2 => CPU_WE2,
227         WE3 => CPU_WE3,
228         DB => CPU_DB,
229         memsel => CPU_MEMSEL
230     );
231
232     LogWithTime("sh2_cpu_tb.vhd: [RAM] Initializing memory from byte " &
233                 to_string(16#0000#) & " to " & to_string(16#0000# + 1024), LogFile);
234     LogWithTime("sh2_cpu_tb.vhd: [RAM] Initializing memory from byte " &
235                 to_string(16#1000#) & " to " & to_string(16#1000# + 1024), LogFile);
236     LogWithTime("sh2_cpu_tb.vhd: [RAM] Initializing memory from byte " &
237                 to_string(16#2000#) & " to " & to_string(16#2000# + 1024), LogFile);
238     LogWithTime("sh2_cpu_tb.vhd: [RAM] Initializing memory from byte " &
239                 to_string(16#3000#) & " to " & to_string(16#3000# + 1024), LogFile);
240     LogWithTime("sh2_cpu_tb.vhd: [RAM] Valid Data Memory Range is 0x0000 to 0x40000", LogFile);
241
242     -- Instantiate RAM memory unit
243     ram : entity work.MEMORY32x32
244     generic map (
245         MEMSIZE => 1024,
246         -- four contiguous blocks of memory (1024 bytes each)
247         START_ADDR0 => 16#0000#,
248         START_ADDR1 => 16#1000#,
249         START_ADDR2 => 16#2000#,
250         START_ADDR3 => 16#3000#
251     )
252     port map (
253         RE0 => RAM_RE0,
254         RE1 => RAM_RE1,
255         RE2 => RAM_RE2,
256         RE3 => RAM_RE3,
257         WE0 => RAM_WE0,
258         WE1 => RAM_WE1,
259         WE2 => RAM_WE2,
260         WE3 => RAM_WE3,
261         MemAB => RAM_AB,
262         MemDB => RAM_DB
263     );
264
```

```
265     LogWithTime("sh2_cpu_tb.vhd: [ROM] Initializing memory from byte " &
266                 to_string(16#0000#) & " to " & to_string(16#0000# + 1024), LogFile);
267     LogWithTime("sh2_cpu_tb.vhd: [ROM] Initializing memory from byte " &
268                 to_string(16#1000#) & " to " & to_string(16#1000# + 1024), LogFile);
269     LogWithTime("sh2_cpu_tb.vhd: [ROM] Initializing memory from byte " &
270                 to_string(16#2000#) & " to " & to_string(16#2000# + 1024), LogFile);
271     LogWithTime("sh2_cpu_tb.vhd: [ROM] Initializing memory from byte " &
272                 to_string(16#3000#) & " to " & to_string(16#3000# + 1024), LogFile);
273     LogWithTime("sh2_cpu_tb.vhd: [ROM] Valid Program Memory Range is 0x0000 to 0x40000", LogFile);
274
275     -- Instantiate ROM memory unit
276     rom : entity work.MEMORY32x32
277     generic map (
278         MEMSIZE => 1024,
279         -- four contiguous blocks of memory (1024 bytes each)
280         START_ADDR0 => 16#0000#,
281         START_ADDR1 => 16#1000#,
282         START_ADDR2 => 16#2000#,
283         START_ADDR3 => 16#3000#
284     )
285     port map (
286         RE0 => ROM_RE0,
287         RE1 => ROM_RE1,
288         RE2 => ROM_RE2,
289         RE3 => ROM_RE3,
290         WE0 => ROM_WE0,
291         WE1 => ROM_WE1,
292         WE2 => ROM_WE2,
293         WE3 => ROM_WE3,
294         MemAB => ROM_AB,
295         MemDB => ROM_DB
296     );
297
298     process
299
300         -- Writes a word of data to a given memory address using the testbench
301         -- control signals. Requires that the address is word-aligned.
302         procedure WriteWord(address : unsigned; data : std_logic_vector) is
303         begin
304             assert address mod 2 = 0
305             report "WriteWord: Cannot write word to unaligned address"
306             severity error;
307
308             TEST_AB <= std_logic_vector(address);   -- Output address to address bus
309
310             -- Shift word of data over to correct location
311             TEST_DB(15 downto 0)  <= data when address mod 4 = 0 else (others => 'X');
312             TEST_DB(31 downto 16) <= data when address mod 4 = 2 else (others => 'X');
313
314             -- Write only the word being addressed
315             TEST_WE0 <= '0' when address mod 4 = 0 else '1';
316             TEST_WE1 <= '0' when address mod 4 = 0 else '1';
317             TEST_WE2 <= '0' when address mod 4 = 2 else '1';
318             TEST_WE3 <= '0' when address mod 4 = 2 else '1';
319
320             wait for 5 ns;  -- wait for signal to propagate
321
322             -- Disable writing
323             TEST_WE0 <= '1';
324             TEST_WE1 <= '1';
325             TEST_WE2 <= '1';
326             TEST_WE3 <= '1';
327
328             wait for 5 ns;  -- wait for signal to propagate
329         end procedure;
330
```

```
331         -- Reads a longword of data from a given memory address using the
332         -- testbench control signals. Assumes that the address is
333         -- longword-aligned.
334         procedure ReadLongword(address : unsigned ; data : out std_logic_vector) is
335         begin
336             TEST_AB <= std_logic_vector(address);   -- Output address to address bus
337             TEST_DB <= (others => 'Z');             -- Data bus unused, don't set
338 
339             -- Read all 4 bytes of longword
340             TEST_RE0 <= '0';
341             TEST_RE1 <= '0';
342             TEST_RE2 <= '0';
343             TEST_RE3 <= '0';
344 
345             wait for 5 ns;  -- wait for signal to propagate
346 
347             -- Reverse bytes to convert from big-endian (in memory) to little-endian (in CPU)
348             data := TEST_DB(7 downto 0) & TEST_DB(15 downto 8) & TEST_DB(23 downto 16) & TEST_DB(31 downto 24);
349 
350             -- Disable writing
351             TEST_RE0 <= '1';
352             TEST_RE1 <= '1';
353             TEST_RE2 <= '1';
354             TEST_RE3 <= '1';
355 
356             wait for 5 ns;  -- wait for signal to propagate
357         end procedure;
358 
359         -- Reads in a binary file and writes each byte into program memory
360         -- using the testbench control signals. Is used so that we can test
361         -- the SH-2 CPU on real assembled machine code.
362         -- Reference: https://stackoverflow.com/a/42581872
363         procedure LoadProgram(path : string) is
364             -- Used to read a file byte-by-byte
365             type char_file_t is file of character;
366             file char_file : char_file_t;
367 
368             -- A single character/byte read from a file
369             variable char_v : character;
370             subtype byte_t is natural range 0 to 255;
371             variable byte_v : byte_t;
372 
373             variable curr_opcode : std_logic_vector(15 downto 0);   -- the current instruction bits
374             variable curr_pc     : unsigned(31 downto 0);           -- the current program address
375         begin
376             -- Write to ROM
377             CPU_ACTIVE <= false;
378             TEST_MEMSEL <= '1';
379 
380             curr_pc := to_unsigned(0, 32);  -- the current address in program memory
381 
382             file_open(char_file, path & ".bin"); -- read file as "characters" to get individual bytes
383             while not endfile(char_file) loop
384                 -- Read a byte from the file
385                 read(char_file, char_v);
386                 byte_v := character'pos(char_v);
387 
388                 -- set low byte of instruction
389                 curr_opcode(7 downto 0) := std_logic_vector(to_unsigned(byte_v, 8));
390 
391                 -- Read a byte from the file
392                 read(char_file, char_v);
393                 byte_v := character'pos(char_v);
394 
395                 -- set high byte of instruction
396                 curr_opcode(15 downto 8) := std_logic_vector(to_unsigned(byte_v, 8));
```

```
397
398                 LogWithTime(
399                   "Read " & to_hstring(curr_opcode(15 downto 8)) &  " " &
400                             to_hstring(curr_opcode(7 downto 0)) &
401                   " @ PC 0x" & to_hstring(curr_pc), LogFile);
402
403                 -- Write instruction word into memory
404                 WriteWord(curr_pc, curr_opcode);
405
406                 -- Increment program address
407                 curr_pc := curr_pc + 2;
408             end loop;
409
410             file_close(char_file);      -- close file
411         end procedure;
412
413         -- Dumps the bytes in data memory to a file in a human-readable format
414         -- for debugging. The start position and total length of memory to be
415         -- output are given as arguments.
416         procedure DumpMemory(path : string; start : integer; length : integer) is
417             file out_file       : text;         -- output file
418             variable curr_line  : line;         -- current line to output
419
420             variable curr_addr  : unsigned(31 downto 0);            -- current address to read
421             variable data_out   : std_logic_vector(31 downto 0);    -- data at current address
422             variable curr_byte  : std_logic_vector(7 downto 0);     -- printing data byte-by-byte
423         begin
424             -- Access RAM
425             CPU_ACTIVE <= false;
426             TEST_MEMSEL <= '0';
427
428             -- Write to output file
429             file_open(out_file, path & ".dump", write_mode);
430
431             -- File header
432             write(curr_line, YELLOW & "Memory dump for " & path & ANSI_RESET);
433             writeline(out_file, curr_line);
434
435             curr_addr := to_unsigned(start, 32);
436             for i in 1 to length loop
437                 ReadLongword(curr_addr, data_out);              -- read longword from memory
438
439                 write(curr_line, to_hstring(curr_addr) & " ");  -- display address
440
441                 -- Output longword bytes in reverse order to convert from
442                 -- big-endian (memory) to little-endian (to be output).
443                 for j in 3 downto 0 loop
444                     curr_byte := data_out(7 + 8 * j downto 8 * j);  -- get current byte
445
446                     -- Color unitialized memory grey.
447                     if (curr_byte = "XXXXXXXX" or   curr_byte = "UUUUUUUU") then
448                         write(curr_line, GREY);
449                     end if;
450
451                     write(curr_line, to_hstring(curr_byte) & " ");  -- write byte
452                     write(curr_line, ANSI_RESET);                   -- reset color
453                 end loop;
454
455                 writeline(out_file, curr_line);         -- output line to file
456                 curr_addr := curr_addr + 4;             -- increment data address
457             end loop;
458         end procedure;
459
460         -- Reads an "expect" file from memory and checks if this file matches with
461         -- the current contents of memory. This is done so that we can test the
462         -- SH-2 CPU for correctness.
```

```
463         --
464         -- Note that the expect files contain only lines of the form:
465         --   AAAAAAAA BBBBBBBB ; optional comment
466         -- where AAAAAAAA is a hexadecimal address and BBBBBBBB is 32 bits of data.
467         -- This function checks that the data at every address matches the data
468         -- provided in the expect files.
469         procedure CheckOutput(path : string) is
470             file test_file : text;  -- test file
471             variable row   : line;  -- current line in test file
472 
473             variable address : unsigned(31 downto 0);                   -- memory address to check
474             variable expected_value : std_logic_vector(31 downto 0);    -- expected value given in test file
475             variable actual_value : std_logic_vector(31 downto 0);      -- actual contents of memory
476         begin
477             -- Access RAM
478             CPU_ACTIVE <= false;
479             TEST_MEMSEL <= '0';
480 
481             file_open(test_file, path & ".expect", read_mode);  -- read expect file
482 
483             while not endfile(test_file) loop
484                 -- Read expected address/value pairs from test file
485                 readline(test_file, row);
486                 hread(row, address);
487                 hread(row, expected_value);
488 
489                 -- Read value at address from RAM
490                 ReadLongword(address, actual_value);
491 
492                 -- Check that the values match up
493                 assert expected_value = actual_value
494                     report path & ": expected " & to_hstring(expected_value) & " at address " &
495                            to_hstring(address) & ", got " & to_hstring(actual_value) & " instead."
496                     severity error;
497             end loop;
498         end procedure;
499 
500         -- Simulate one cycle of the clock
501         procedure Tick is
502         begin
503             clock <= '0';
504             wait for 10 ns;
505             clock <= '1';
506             wait for 10 ns;
507         end procedure;
508 
509         -- We define the CPU exit signal to be when it tries to access
510         -- address 0xFFFFFFFC (signed integer representation of -4) for
511         -- the sake of testing.
512         impure function CheckDone return boolean is
513         begin
514             return CPU_AB = X"FFFFFFFC";
515         end function;
516 
517         -- Resets the CPU and executes the program currently stored in ROM.
518         -- Simply keeps clocking the CPU until the exit condition is met.
519         procedure RunCPU is
520         begin
521             -- Give memory control to CPU
522             CPU_ACTIVE <= true;
523 
524             -- Reset CPU
525             reset <= '0';
526             Tick;
527 
528             -- Run program until finished
```

```
529             reset <= '1';
530             while not CheckDone loop
531                 Tick;
532             end loop;
533         end procedure;
534 
535         -- Runs a test on the CPU. First loads an assembled program into ROM,
536         -- clocks the CPU until it stops running, then checks if the contents
537         -- of memory match up with the expected values. This procedure requires
538         -- the path of the test, which is then postfixed with ".bin" and
539         -- ".expect" to compute the path of the binary file and expect file,
540         -- respectively. The resulting memory is also dumped to a ".dump" file
541         -- for debugging.
542         procedure RunTest(path : string) is
543         begin
544             LogBothWithTime("Running test: " & path, LogFile);
545             LoadProgram(path);          -- write program in to ROM
546             RunCPU;                     -- execute program
547             DumpMemory(path, 0, 64);    -- dump memory to file
548             CheckOutput(path);          -- check RAM has expected values
549         end procedure;
550 
551     begin
552 
553         -- Run all CPU tests
554 
555         RunTest("asm/mov_reg");                         -- Tests Mov between registers
556         RunTest("asm/reg_indirect");                    -- Tests register indirect addressing
557         RunTest("asm/arith");                           -- Tests arithmetic instructions (add, sub, etc)
558         RunTest("asm/logic");                           -- Tests logic instructions (and, or, xor, etc)
559         RunTest("asm/shift");                           -- Tests shift instructions (shll, shlr, etc)
560         RunTest("asm/cmp");                             -- Test CMP operations
561         RunTest("asm/ext");                             -- Test zero/sign extension instructions
562         RunTest("asm/bshift");                          -- Test barrel shift instructions
563         RunTest("asm/sr");                              -- Tests status register (SETT, CLRT)
564         RunTest("asm/system");                          -- Tests system register operations (LDC, STC)
565         RunTest("asm/control");                         -- Tests control register operations (LDS, STS)
566         RunTest("asm/mov_wl_at_disp_pc_rn");            -- Tests Mov (disp, PC), Rn
567         RunTest("asm/mov_bwl_at_rm_rn");                -- Tests Mov @Rm, Rn
568         RunTest("asm/mov_bwl_rm_at_minus_rn");          -- Tests Mov Rm, @-Rn
569         RunTest("asm/mov_bwl_at_rm_plus_rn");           -- Test Mov @Rm+, Rn
570         RunTest("asm/mov_bwl_r0_or_rm_at_disp_rn");     -- Test Mov R0, @(disp, Rn) and Mov Rm, @(disp,Rn)
571         RunTest("asm/mov_bwl_at_disp_rm_r0_or_rn");     -- Test Mov @(disp, Rm), R0 and Mov @(disp, Rm), Rn
572         RunTest("asm/mov_rm_at_r0_rn");                 -- Test Mov Rm, @(R0, Rn)
573         RunTest("asm/mov_b_at_r0_rm_rn");               -- Test Mov @(R0, Rm), Rn
574         RunTest("asm/mov_bwl_r0_at_disp_gbr");          -- Test Mov R0, @(disp, GBR)
575         RunTest("asm/mov_at_disp_gbr_r0");              -- Test Mov @(disp, GBR), R0
576         RunTest("asm/mova_at_disp_pc_r0");              -- Test Mova @(disp, PC), R0
577         RunTest("asm/movt_rn");                         -- Test Movt Rn
578         RunTest("asm/swap");                            -- Test SWAP.B Rm, Rn and SWAP.W Rm, Rn
579         RunTest("asm/xtrct");                           -- Test XTRCT Rm, Rn
580         RunTest("asm/branch");                          -- Test branch instructions.
581 
582         wait;
583     end process;
584 
585 end behavioral;
586 
587 
```