```
1   ----------------------------------------------------------------------------------------------
2   --
3   --  SH-2 Instructions
4   --
5   --  This package defines the bits defining each type of instruction in the SH-2
6   --  instruction set. The encodings are placed here for the purpose of
7   --  organization.
8   --
9   --  Revision History
10  --      07 Jun 25   Zack Huang      Copied over from control unit
11  --
12  ----------------------------------------------------------------------------------------------
13
14  library ieee;
15  library std;
16
17  use std.textio.all;
18
19  use ieee.std_logic_1164.all;
20  use ieee.numeric_std.all;
21  use work.MemoryInterfaceConstants.all;
22  use work.Logging.all;
23
24  package SH2InstructionEncodings is
25
26    subtype Instruction is std_logic_vector(15 downto 0);
27
28    -- Instruction encodings.
29
30    -- Data Transfer Instructions:
31    constant  MOV_IMM_RN            :  Instruction := "1110------------";  -- MOV #imm, Rn
32
33    constant  MOV_AT_DISP_PC_RN     :  Instruction := "1-01------------";  -- MOV.X @(disp, PC), Rn (for bit decoding.)
34    constant  MOV_W_AT_DISP_PC_RN   :  Instruction := "1001------------";  -- MOV.W @(disp, PC), Rn
35    constant  MOV_L_AT_DISP_PC_RN   :  Instruction := "1101------------";  -- MOV.L @(disp, PC), Rn
36
37    constant  MOV_RM_RN             :  Instruction := "0110--------0011";  -- MOV Rm, Rn
38
39    constant  MOV_RM_AT_RN          :  Instruction := "0010--------00--";  -- MOV.X Rm, @Rn (for bit decoding).
40    constant  MOV_B_RM_AT_RN        :  Instruction := "0010--------0000";  -- MOV.B Rm, @Rn
41    constant  MOV_W_RM_AT_RN        :  Instruction := "0010--------0001";  -- MOV.W Rm, @Rn
42    constant  MOV_L_RM_AT_RN        :  Instruction := "0010--------0010";  -- MOV.L Rm, @Rn
43
44    constant  MOV_AT_RM_RN          :  Instruction := "0110--------00--";  -- MOV.X @Rm, Rn (for bit decoding).
45    constant  MOV_B_AT_RM_RN        :  Instruction := "0110--------0000";  -- MOV.B @Rm, Rn
46    constant  MOV_W_AT_RM_RN        :  Instruction := "0110--------0001";  -- MOV.W @Rm, Rn
47    constant  MOV_L_AT_RM_RN        :  Instruction := "0110--------0010";  -- MOV.L @Rm, Rn
48
49    constant  MOV_RM_AT_MINUS_RN    :  Instruction := "0010--------01--";  -- MOV.X Rm, @-Rn (for bit decoding).
50    constant  MOV_B_RM_AT_MINUS_RN  :  Instruction := "0010--------0100";  -- MOV.B Rm, @-Rn
51    constant  MOV_W_RM_AT_MINUS_RN  :  Instruction := "0010--------0101";  -- MOV.W Rm, @-Rn
52    constant  MOV_L_RM_AT_MINUS_RN  :  Instruction := "0010--------0110";  -- MOV.L Rm, @-Rn
53
54    constant  MOV_AT_RM_PLUS_RN     :  Instruction := "0110--------01--";  -- MOV.X @Rm+, Rn (for bit decoding)
55    constant  MOV_B_AT_RM_PLUS_RN   :  Instruction := "0110--------0100";  -- MOV.B @Rm+, Rn
56    constant  MOV_W_AT_RM_PLUS_RN   :  Instruction := "0110--------0101";  -- MOV.W @Rm+, Rn
57    constant  MOV_L_AT_RM_PLUS_RN   :  Instruction := "0110--------0110";  -- MOV.W @Rm+, Rn
58
59    constant  MOV_R0_AT_DISP_RN     :  Instruction := "1000000---------";  -- MOV.{B,W} RO, @(disp,Rn)
60    constant  MOV_B_R0_AT_DISP_RN   :  Instruction := "10000000--------";  -- MOV.B RO, @(disp,Rn)
61    constant  MOV_W_R0_AT_DISP_RN   :  Instruction := "10000001--------";  -- MOV.W RO, @(disp,Rn)
62
63    constant  MOV_L_RM_AT_DISP_RN   :  Instruction := "0001------------";  -- MOV.L Rm, @(disp, Rn)
64
65    constant  MOV_AT_DISP_RM_R0     :  Instruction := "1000010---------";  -- MOV.{B,W} @(disp, Rm), R0
66    constant  MOV_B_AT_DISP_RM_R0   :  Instruction := "10000100--------";  -- MOV.B @(disp, Rm), R0
```

```
67    constant  MOV_W_AT_DISP_RM_R0   :  Instruction := "10000101--------";  -- MOV.W @(disp, Rm), R0
68
69    constant  MOV_L_AT_DISP_RM_RN   :  Instruction := "0101------------";  -- MOV.L @(disp, Rm), Rn
70
71    constant  MOV_RM_AT_R0_RN       :  Instruction := "0000--------01--";  -- MOV.X Rm, @(R0, Rn)
72    constant  MOV_B_RM_AT_R0_RN     :  Instruction := "0000--------0100";  -- MOV.B Rm, @(R0, Rn)
73    constant  MOV_W_RM_AT_R0_RN     :  Instruction := "0000--------0101";  -- MOV.W Rm, @(R0, Rn)
74    constant  MOV_L_RM_AT_R0_RN     :  Instruction := "0000--------0110";  -- MOV.L Rm, @(R0, Rn)
75
76    constant  MOV_AT_R0_RM_RN       :  Instruction := "0000--------11--";  -- MOV.X @(R0, Rm), Rn
77    constant  MOV_B_AT_R0_RM_RN     :  Instruction := "0000--------1100";  -- MOV.B @(R0, Rm), Rn
78    constant  MOV_W_AT_R0_RM_RN     :  Instruction := "0000--------1101";  -- MOV.W @(R0, Rm), Rn
79    constant  MOV_L_AT_R0_RM_RN     :  Instruction := "0000--------1110";  -- MOV.L @(R0, Rm), Rn
80
81    constant  MOV_R0_AT_DISP_GBR    :  Instruction := "110000----------";  -- MOV.X R0, @(disp, GBR)
82    constant  MOV_B_R0_AT_DISP_GBR  :  Instruction := "11000000--------";  -- MOV.B R0, @(disp, GBR)
83    constant  MOV_W_R0_AT_DISP_GBR  :  Instruction := "11000001--------";  -- MOV.W R0, @(disp, GBR)
84    constant  MOV_L_R0_AT_DISP_GBR  :  Instruction := "11000010--------";  -- MOV.L R0, @(disp, GBR)
85
86    constant  MOV_AT_DISP_GBR_R0    :  Instruction := "110001----------";  -- MOV.X @(disp, GBR), R0
87    constant  MOV_B_AT_DISP_GBR_R0  :  Instruction := "11000100--------";  -- MOV.B @(disp, GBR), R0
88    constant  MOV_W_AT_DISP_GBR_R0  :  Instruction := "11000101--------";  -- MOV.W @(disp, GBR), R0
89    constant  MOV_L_AT_DISP_GBR_R0  :  Instruction := "11000110--------";  -- MOV.L @(disp, GBR), R0
90
91    constant  MOVA_AT_DISP_PC_R0    :  Instruction := "11000111--------";  -- MOVA @(disp, PC), R0
92
93    constant  MOVT_RN               :  Instruction := "0000----00101001";  -- MOVT Rn
94
95    constant  SWAP_RM_RN            :  Instruction := "0110--------100-";  -- SWAP.{B,W} Rm, Rn
96    constant  SWAP_B_RM_RN          :  Instruction := "0110--------1000";  -- SWAP.B Rm, Rn
97    constant  SWAP_W_RM_RN          :  Instruction := "0110--------1001";  -- SWAP.W Rm, Rn
98
99    constant  XTRCT_RM_RN           :  Instruction := "0010--------1101";  -- XTRCT Rm, Rn
100
101   -- Arithmetic Instructions:
102   constant ADD_RM_RN     : Instruction := "0011--------11--";
103   constant ADD_IMM_RN    : Instruction := "0111------------";
104   constant SUB_RM_RN     : Instruction := "0011--------10--";
105   constant NEG_RM_RN     : Instruction := "0110--------101-";
106   constant DT_RN         : Instruction := "0100----00010000";
107   constant EXT_RM_RN     : Instruction := "0110--------11--";
108
109   constant CMP_EQ_IMM    : Instruction := "10001000--------";   -- CMP/EQ #imm, R0
110   constant CMP_RM_RN     : Instruction := "0011--------0---";   -- CMP/{EQ,HS,GE,HI,GT} Rm, Rn
111   constant CMP_RN        : Instruction := "0100----00010-01";   -- CMP/{PL/PZ}
112   constant CMP_STR_RM_RN : Instruction := "0010--------1100";   -- CMP/STR
113
114   -- Logical Operations:
115   constant LOGIC_RM_RN   : Instruction := "0010--------10--";  -- AND, TST, OR, XOR
116   constant LOGIC_IMM_R0  : Instruction := "110010----------";  -- AND, TST, OR, XOR
117   constant NOT_RM_RN     : Instruction := "0110--------0111";  -- NOT
118
119   -- Shift Instruction:
120   constant SHIFT_RN      : Instruction := "0100----00-00-0-";  -- shift/rotate instructions
121   constant BSHIFT_RN     : Instruction := "0100----00--100-";  -- barrel shifter instructions
122
123   -- Branch Instructions:
124   constant BF     :   Instruction := "10001011--------"; -- BF      <label>
125   constant BF_S   :   Instruction := "10001111--------"; -- BF/S    <label>
126   constant BT     :   Instruction := "10001001--------"; -- BT      <label>
127   constant BT_S   :   Instruction := "10001101--------"; -- BT/S    <label>
128   constant BRA    :   Instruction := "1010------------"; -- BRA     <label>
129   constant BRAF   :   Instruction := "0000----00100011"; -- BRAF    Rm
130   constant BSR    :   Instruction := "1011------------"; -- BSR     <label>
131   constant BSRF   :   Instruction := "0000----00000011"; -- BSRF    Rm
132   constant JMP    :   Instruction := "0100----00101011"; -- JMP     @Rm
```

```
133   constant JSR    :   Instruction := "0100----00001011"; -- JSR     @Rm
134   constant RTS    :   Instruction := "0000000000001011"; -- RTS
135
136
137   -- System Control:
138   constant NOP      : Instruction := "0000000000001001";
139   constant CLRT     : Instruction := "0000000000001000";
140   constant CLRMAC   : Instruction := "0000000000101000";
141   constant SETT     : Instruction := "0000000000011000";
142
143   constant STC_SYS_RN             : Instruction := "0000----00--0010";  -- STC {SR, GBR, VBR}, Rn
144   constant STC_SR_RN              : Instruction := "0000----00000010";  -- STC SR,  Rn
145   constant STC_GBR_RN             : Instruction := "0000----00010010";  -- STC GBR, Rn
146   constant STC_VBR_RN             : Instruction := "0000----00100010";  -- STC VBR, Rn
147
148   constant STC_L_SYS_RN           : Instruction := "0100----00--0011";  -- STC.L {SR, GBR, VBR}, @-Rn
149   constant STC_L_SR_AT_MINUS_RN   : Instruction := "0100----00000011";  -- STC.L SR, @-Rn
150   constant STC_L_GBR_AT_MINUS_RN  : Instruction := "0100----00010011";  -- STC.L GBR, @-Rn
151   constant STC_L_VBR_AT_MINUS_RN  : Instruction := "0100----00100011";  -- STC.L VBR, @-Rn
152
153   constant LDC_RM_SYS             : Instruction := "0100----00--1110";  -- LDC Rm, {SR, GBR, VBR}
154   constant LDC_RM_SR              : Instruction := "0100----00001110";  -- LDC Rm, SR
155   constant LDC_RM_GBR             : Instruction := "0100----00011110";  -- LDC Rm, GBR
156   constant LDC_RM_VBR             : Instruction := "0100----00101110";  -- LDC Rm, VBR
157
158   constant LDC_L_RM_SYS           : Instruction := "0100----00--0111";  -- LDC.L @Rm+, {SR, GBR, VBR}
159   constant LDC_L_AT_RM_PLUS_SR    : Instruction := "0100----00000111";  -- LDC.L @Rm+, SR
160   constant LDC_L_AT_RM_PLUS_GBR   : Instruction := "0100----00010111";  -- LDC.L @Rm+, GBR
161   constant LDC_L_AT_RM_PLUS_VBR   : Instruction := "0100----00100111";  -- LDC.L @Rm+, VBR
162
163   constant LDS_RM_SYS             : Instruction := "0100----00--1010";  -- LDS Rm, {MACH, MACL, PR}
164   constant LDS_RM_MACH            : Instruction := "0100----00001010";  -- LDS Rm, MACH
165   constant LDS_RM_MACL            : Instruction := "0100----00011010";  -- LDS Rm, MACL
166   constant LDS_RM_PR              : Instruction := "0100----00101010";  -- LDS Rm, PR
167
168   constant LDS_L_RM_SYS           : Instruction := "0100----00--0110";  -- LDS.L @Rm+, {MACH, MACL, PR}
169   constant LDS_L_AT_RM_PLUS_MACH  : Instruction := "0100----00000110";  -- LDS.L @Rm+, MACH
170   constant LDS_L_AT_RM_PLUS_MACL  : Instruction := "0100----00010110";  -- LDS.L @Rm+, MACL
171   constant LDS_L_AT_RM_PLUS_PR    : Instruction := "0100----00100110";  -- LDS.L @Rm+, PR
172
173   constant STS_SYS_RN             : Instruction := "0000----00--1010";  -- STS Rm, {MACH, MACL, PR}
174   constant STS_MACH_RN            : Instruction := "0000----00001010";  -- STS Rm, MACH
175   constant STS_MACL_RN            : Instruction := "0000----00011010";  -- STS Rm, MACL
176   constant STS_PR_RN              : Instruction := "0000----00101010";  -- STS Rm, PR
177
178   constant STS_L_SYS_RN           : Instruction := "0100----00--0010";  -- STS.L @Rm+, {MACH, MACL, PR}
179   constant STS_L_AT_RM_PLUS_MACH  : Instruction := "0100----00000010";  -- STS.L @Rm+, MACH
180   constant STS_L_AT_RM_PLUS_MACL  : Instruction := "0100----00010010";  -- STS.L @Rm+, MACL
181   constant STS_L_AT_RM_PLUS_PR    : Instruction := "0100----00100010";  -- STS.L @Rm+, PR
182
183
184 end package SH2InstructionEncodings;
185
186
```

```
1   ----------------------------------------------------------------------------------------------
2   --
3   --  SH-2 Control signals
4   --
5   --  This package defines the control signals and constants that are used
6   --  internally within the SH-2 CPU to implement instructions.
7   --
8   --  Revision History
9   --      07 Jun 25   Zack Huang      Copied over from control unit
10  --      07 Jun 25   Zack Huang      Reorganized types, renamed signals for consistency
11  --
12  ----------------------------------------------------------------------------------------------
13
14  library ieee;
15  use ieee.std_logic_1164.all;
16  use ieee.numeric_std.all;
17
18  package SH2ControlSignals is
19
20      -- We define record types for the control signals going to each component of the CPU.
21
22      -- Memory interface control signals
23      type mem_ctrl_t is record
24          Enable      : std_logic;                        -- if memory needs to be accessed (read or write)
25          AddrSel     : std_logic;
26          ReadWrite   : std_logic;                        -- if should do memory read (0) or write (1)
27          Mode        : std_logic_vector(1 downto 0);     -- if memory access should be by byte, word, or longword
28          Sel         : std_logic;                        -- select memory address source, from DMAU output (0) or PMAU output (1)
29          OutSel      : std_logic_vector(2 downto 0);     -- what should be output to memory
30      end record;
31
32      -- ALU control signals
33      type alu_ctrl_t is record
34          OpBSel      : std_logic;                        -- input mux to Operand B, either RegB (0) or Immediate (1)
35          LoadA       : std_logic;                        -- determine if OperandA is loaded ('1') or zeroed ('0')
36          FCmd        : std_logic_vector(3 downto 0);     -- F-Block operation
37          CinCmd      : std_logic_vector(1 downto 0);     -- carry in operation
38          SCmd        : std_logic_vector(2 downto 0);     -- shift operation
39          ALUCmd      : std_logic_vector(1 downto 0);     -- ALU result select
40          TCmpSel     : std_logic_vector(2 downto 0);     -- how to compute T from ALU status flags
41          Immediate   : std_logic_vector(7 downto 0);     -- 8-bit immediate
42          ImmediateMode : std_logic;                        -- Immediate extension mode
43          ExtMode     : std_logic_vector(1 downto 0);     -- mode for extending register value (zero or signed)
44          TFlagSel    : std_logic_vector(2 downto 0);     -- source for next value of T flag
45      end record;
46
47      -- Register array control signals
48      type reg_ctrl_t is record
49          DataInSel   : std_logic_vector(3 downto 0);     -- source for register input data
50          EnableIn    : std_logic;                        -- if data should be written to an input register
51          InSel       : integer range 15 downto 0;        -- which register to write data to
52          ASel        : integer range 15 downto 0;        -- which register to read to bus A
53          BSel        : integer range 15 downto 0;        -- which register to read to bus B
54      end record;
55
56      -- DMAU control signals
57      type dmau_ctrl_t is record
58          GBRWriteEn  : std_logic;
59          Off4        : std_logic_vector(3 downto 0);
60          Off8        : std_logic_vector(7 downto 0);
61          BaseSel     : std_logic_vector(1 downto 0);
62          IndexSel    : std_logic_vector(1 downto 0);
63          OffScalarSel: std_logic_vector(1 downto 0);
64          IncDecSel   : std_logic_vector(1 downto 0);
65          AxInSel     : integer range 15 downto 0;        -- which address register to write to
66          AxStore     : std_logic;                        -- if data should be written to the address register
```

```
67          A1Sel       : integer range 15 downto 0;        -- which register to read to address bus 1
68          A2Sel       : integer range 15 downto 0;        -- which register to read to address bus 2
69      end record;
70
71      -- PMAU control signals
72      type pmau_ctrl_t is record
73          PCAddrMode  : std_logic_vector(2 downto 0);     -- What PC addressing mode is desired (if not branch, just increment)
74          Off8        : std_logic_vector(7 downto 0);     -- 8-bit offset for relative addressing
75          Off12       : std_logic_vector(11 downto 0);    -- 12-bit offset for relative addressing
76          PCWriteCtrl : std_logic_vector(1 downto 0);     -- What to write to the PC register
77          ConditionalBranch  : std_logic;                 -- if performing a conditional branch (checked during ID stage)
78          Condition   : std_logic;                        -- condition to check for condional branches
79          DelayBranch : std_logic;                        -- if the branch being performed should be delayed
80      end record;
81
82      -- System control signals
83      type sys_ctrl_t is record
84          PRWriteEn   : std_logic;                        -- Enable writing to PR
85          RegCtrl     : std_logic_vector(1 downto 0);
86          RegSel      : std_logic_vector(2 downto 0);
87          RegSrc      : std_logic_vector(1 downto 0);
88      end record;
89
90      type ctrl_t is record
91          MemCtrl      : mem_ctrl_t;
92          ALUCtrl      : alu_ctrl_t;
93          REGCtrl      : reg_ctrl_t;
94          PMAUCtrl     : pmau_ctrl_t;
95          DMAUCtrl     : dmau_ctrl_t;
96          SysCtrl      : sys_ctrl_t;
97      end record;
98
99
100     -- Internal control signals for controlling muxes within the CPU
101
102     -- Constants for RegDataInSel in sh2_cpu architecture. Used to determine what to input to the
103     -- register array's RegDataIn.
104     constant RegDataIn_ALUResult        : std_logic_vector(3 downto 0) := "0000"; -- ALU result
105     constant RegDataIn_Immediate        : std_logic_vector(3 downto 0) := "0001"; -- Sign-extended 8-bit immediate
106     constant RegDataIn_RegA             : std_logic_vector(3 downto 0) := "0010"; -- RegA output
107     constant RegDataIn_RegB             : std_logic_vector(3 downto 0) := "0011"; -- RegB output
108     constant RegDataIn_SysReg           : std_logic_vector(3 downto 0) := "0100"; -- System register
109     constant RegDataIn_RegA_SwapB       : std_logic_vector(3 downto 0) := "0111"; -- RegA output with low two bytes swapped
110     constant RegDataIn_RegA_SwapW       : std_logic_vector(3 downto 0) := "1000"; -- RegA output with low/high words swapped
111     constant RegDataIn_RegB_RegA_Center : std_logic_vector(3 downto 0) := "1001"; -- Low word of RegB and high word of RegA
112     constant RegDataIn_SR_TBit          : std_logic_vector(3 downto 0) := "1010"; -- T-bit (in the LSB)
113     constant RegDataIn_PR               : std_logic_vector(3 downto 0) := "1011"; -- Procedure Register (PR)
114     constant RegDataIn_DB               : std_logic_vector(3 downto 0) := "1100"; -- Data Bus (DB) value
115     constant RegDataIn_Ext              : std_logic_vector(3 downto 0) := "1101"; -- Sign/zero extended register values.
116
117     -- Constants for ExtMode in sh2_cpu architecture. Used to determine how RegB will be extended
118     -- to a long-word.
119     --
120     -- WARNING: Changing these will break bit decoding of instructions.
121     --
122     constant Ext_SignB_RegA : std_logic_vector(1 downto 0) := "10";  -- Sign extend low byte of Reg A
123     constant Ext_SignW_RegA : std_logic_vector(1 downto 0) := "11";  -- Sign extend low word of Reg A
124     constant Ext_ZeroB_RegA : std_logic_vector(1 downto 0) := "00";  -- Zero extend low byte of Reg A
125     constant Ext_ZeroW_RegA : std_logic_vector(1 downto 0) := "01";  -- Zero extend low word of Reg A
126
127
128     -- Constants for ReadWrite used in memory_tx architecture. Determines whether to read or
129     -- write to memory.
130     --
131     constant ReadWrite_Read     : std_logic := '0';
132     constant ReadWrite_Write    : std_logic := '1';
```

```
133
134     -- Constants for selecting what to output to MemDataOut in memory_tx entity in the sh2_cpu
135     -- entity.
136     --
137     constant MemOut_RegA    : std_logic_vector(2 downto 0) := "000"; -- Output RegA to data bus
138     constant MemOut_RegB    : std_logic_vector(2 downto 0) := "001"; -- Output RegB to data bus
139     constant MemOut_SysReg  : std_logic_vector(2 downto 0) := "010"; -- Output a system register to data bus
140
141
142     constant ALUOpB_RegB    : std_logic := '0'; -- Use RegB as B input to ALU.
143     constant ALUOpB_Imm     : std_logic := '1'; -- Use immediate as B input to ALU
144
145     constant TFlagSel_T         : std_logic_vector(2 downto 0) := "000";    -- Have T retain its value
146     constant TFlagSel_Zero      : std_logic_vector(2 downto 0) := "001";    -- Set T to the ALU zero flag
147     constant TFlagSel_Carry     : std_logic_vector(2 downto 0) := "010";    -- Set T to the ALU carry flag
148     constant TFlagSel_Overflow  : std_logic_vector(2 downto 0) := "011";    -- Set T to the ALU overflow flag
149     constant TFlagSel_Clear     : std_logic_vector(2 downto 0) := "100";    -- clear T (to 0)
150     constant TFlagSel_Set       : std_logic_vector(2 downto 0) := "101";    -- set T (to 1)
151     constant TFlagSel_Cmp       : std_logic_vector(2 downto 0) := "110";    -- set T to a value computed from
152                                                                             -- the ALU flags
153
154     -- WARNING: Changing these will break bit decoding of instuctions.
155
156     -- How to calculate the T-bit in the
157     constant TCmp_EQ            : std_logic_vector(2 downto 0) := "000"; --
158     constant TCmp_HS            : std_logic_vector(2 downto 0) := "010";
159     constant TCmp_GE            : std_logic_vector(2 downto 0) := "011";
160     constant TCmp_HI            : std_logic_vector(2 downto 0) := "110";
161     constant TCmp_GT            : std_logic_vector(2 downto 0) := "111";
162     constant TCmp_STR           : std_logic_vector(2 downto 0) := "100";
163
164     constant MemSel_ROM         : std_logic := '1';
165     constant MemSel_RAM         : std_logic := '0';
166
167     constant MemAddrSel_PMAU    : std_logic := '0';
168     constant MemAddrSel_DMAU    : std_logic := '1';
169
170     constant SysRegCtrl_None    : std_logic_vector(1 downto 0) := "00";     -- do nothing with system register
171     constant SysRegCtrl_Load    : std_logic_vector(1 downto 0) := "01";     -- load system register with new value
172     constant SysRegCtrl_Clear   : std_logic_vector(1 downto 0) := "10";     -- clear system register
173
174     constant SysRegSrc_RegB     : std_logic_vector(1 downto 0) := "00";     -- load system register from register bus B
175     constant SysRegSrc_DB       : std_logic_vector(1 downto 0) := "01";     -- load system register from data bus
176     constant SysRegSrc_PC       : std_logic_vector(1 downto 0) := "10";     -- load system register from PC
177
178     -- WARNING: Changing these will break bit decoding of instuctions.
179     constant SysRegSel_System   : std_logic_vector(2 downto 0) := "0--";
180     constant SysRegSel_SR       : std_logic_vector(2 downto 0) := "000";
181     constant SysRegSel_GBR      : std_logic_vector(2 downto 0) := "001";
182     constant SysRegSel_VBR      : std_logic_vector(2 downto 0) := "010";
183     constant SysRegSel_Control  : std_logic_vector(2 downto 0) := "1--";
184     constant SysRegSel_MACH     : std_logic_vector(2 downto 0) := "100";
185     constant SysRegSel_MACL     : std_logic_vector(2 downto 0) := "101";
186     constant SysRegSel_PR       : std_logic_vector(2 downto 0) := "110";
187
188     -- Whether to sign or zero extend the immediate into a 32-bit word.
189     constant ImmediateMode_Sign     : std_logic := '0';
190     constant ImmediateMode_Zero     : std_logic := '1';
191
192
193 end package SH2ControlSignals;
194
```

```
1    ----------------------------------------------------------------------------------------------
2    --                                                                                          -
3    --  Control Unit                                                                            -
4    --                                                                                          -
5    --  This entity implements the control unit for the SH2 CPU. This CPU is                   -
6    --  pipelined, with 5 stages: instruction fetch (IF), instruction decode (ID),             -
7    --  execute (X), memory access (MA), and writeback (WR). Most instructions only            -
8    --  involve the IF, ID, and X stages, while instructions that read from memory             -
9    --  also require MA and WR. On each clock, there are five instructions in                  -
10   --  flight, with one instructions being fetched from memory for maximum                    -
11   --  throughput. Pipeline bubbles are inserted to deal with memory bus                      -
12   --  contention between IF and MA. Additionally, branch instructions are                    -
13   --  designed to discard the previous two instructions so that there is no                  -
14   --  branch delay.                                                                           -
15   --                                                                                          -
16   --  Revision History:                                                                       -
17   --     06 May 25  Zack Huang        Initial revision                                        -
18   --     07 May 25  Chris Miranda     Initial implentation of MOV and branch instruction decoding.   -
19   --     10 May 25  Zack Huang        Implementing ALU instruction                            -
20   --     14 May 25  Chris M.          Formatting.                                             -
21   --     16 May 25  Zack Huang        Documentation, renaming signals                         -
22   --     25 May 25  Zack Huang        Finishing ALU and system instructions                   -
23   --     26 May 25  Chris M.          Add T flag as input to control unit. Add delay slot simulation -
24   --                                  signals.                                                -
25   --                                                                                          -
26   --     29 May 25  Chris M.          Add PCWriteCtrl and DelayedBranchTaken signals to control unit -
27   --                                  output.                                                 -
28   --     07 Jun 25  Zack Huang        Re-organized control signals and constants              -
29   --     08 Jun 25  Zack Huang        Implementing pipelining                                 -
30   --                Chris M.                                                                  -
31   --     09 Jun 25  Zack Huang        Non-delay branches working                              -
32   --                                                                                          -
33   -- Notes:                                                                                   -
34   --  - When reading/writing to registers, RegB is always Rm and RegA is always Rn           -
35   --  - When reading/writing to addresses (in registers), RegA2 is always @(Rm) and          -
36   --    RegA2 is always @(Rn).                                                                -
37   --                                                                                          -
38   ----------------------------------------------------------------------------------------------
39
40   library ieee;
41   library std;
42
43   use ieee.std_logic_1164.all;
44   use ieee.numeric_std.all;
45
46   use std.textio.all;
47
48   use work.SH2PmauConstants.all;
49   use work.SH2DmauConstants.all;
50   use work.MemoryInterfaceConstants.all;
51   use work.SH2InstructionEncodings.all;
52   use work.SH2ControlSignals.all;
53   use work.SH2ALUConstants.all;
54   use work.Logging.all;
55   use work.Utils.all;
56
57   entity  SH2Control  is
58
59       port (
60           -- Input signals
61           MemDataIn           : in  std_logic_vector(31 downto 0);    -- data read from memory
62           TFlagIn             : in  std_logic;                        -- T Flag input from top level CPU
63           clock               : in  std_logic;                        -- system clock
64           reset               : in  std_logic;                        -- system reset (active low, async)
65
66           -- Control signal groups
```

```
67            MemCtrl             : out mem_ctrl_t;                       -- memory interface control signals
68            ALUCtrl             : out alu_ctrl_t;                       -- ALU control signals
69            RegCtrl             : out reg_ctrl_t;                       -- register array control signals
70            DMAUCtrl            : out dmau_ctrl_t;                      -- DMAU control signals
71            PMAUCtrl            : out pmau_ctrl_t;                      -- PMAU control signals
72            SysCtrl             : out sys_ctrl_t                        -- system control signals
73    );
74
75    end  SH2Control;
76
77    architecture dataflow of sh2control is
78        --
79        -- Default control signals - shouldn't touch registers, but does fetch an
80        -- instruction and increment PC
81
82        -- Default control signals for Memory Interface
83        constant DEFAULT_MEM_CTRL : mem_ctrl_t := (
84            Enable      => '0',                         -- do nothing
85            AddrSel     => MemAddrSel_PMAU,             -- unused
86            ReadWrite   => ReadWrite_Read,              -- unused
87            Mode        => WordMode,                    -- unused
88            Sel         => MemAddrSel_PMAU,             -- unused
89            OutSel      => (others => 'X')              -- unused
90        );
91
92        -- Default control signals for ALU
93        constant DEFAULT_ALU_CTRL : alu_ctrl_t := (
94            TFlagSel    => TFlagSel_T,                  -- T flag retains its value
95            OpBSel      => ALUOpB_RegB,                 -- unused (assuming result of ALU not stored to a register)
96            LoadA       => '0',                         -- unused
97            FCmd        => (others => 'X'),             -- unused
98            CinCmd      => (others => 'X'),             -- unused
99            SCmd        => (others => 'X'),             -- unused
100           ALUCmd      => (others => 'X'),             -- unused
101           TCmpSel     => TCmp_EQ,                     -- unused
102           Immediate   => (others => '0'),             -- unused
103           ImmediateMode => ImmediateMode_Sign,        -- unused
104           ExtMode     => Ext_ZeroB_RegA               -- unused
105       );
106
107       -- Default control signals for Register Array
108       constant DEFAULT_REG_CTRL : reg_ctrl_t := (
109           EnableIn    => '0',                             -- Do not write to any general-purpose register
110           DataInSel   => RegDataIn_ALUResult,             -- unused
111           InSel       => 0, ASel        => 0,             -- unused
112           BSel        => 0                                -- unused
113       );
114
115       -- Default control signals for DMAU (Data Memory Access Unit)
116       constant DEFAULT_DMAU_CTRL : dmau_ctrl_t := (
117           AxStore     => '0',                             -- Do not write to address register
118           GBRWriteEn  => '0',                             -- Do not write to GBR
119           Off4        => (others => '0'),                 -- unused
120           Off8        => (others => '0'),                 -- unused
121           BaseSel     => (others => '0'),                 -- unused
122           IndexSel    => (others => '0'),                 -- unused
123           OffScalarSel=> (others => '0'),                 -- unused
124           IncDecSel   => (others => '0'),                 -- unused
125           A1Sel       => 0, A2Sel       => 0,             -- unused
126           AxInSel     => 0                                -- unused
127       );
128
129       -- Default control signals for PMAU (Program Memory Access Unit)
130       -- These defaults ensure the PC is incremented for the next instruction fetch.
131       constant DEFAULT_PMAU_CTRL : pmau_ctrl_t := (
132           PCAddrMode  => PCAddrMode_INC,                  -- Increment PC
```

```
133           PCWriteCtrl => PCWriteCtrl_WRITE_CALC,          -- Write to PC
134           Off8        => (others => '0'),                 -- unused
135           Off12       => (others => '0'),                 -- unused
136           ConditionalBranch  => '0',                      -- Not performing a conditional branch
137           Condition   => '0',                             -- Not performing a conditional branch
138           DelayBranch => '0'                              -- Assume no branch by default
139       );
140
141       constant HOLD_PMAU_CTRL : pmau_ctrl_t := (
142           PCAddrMode  => PCAddrMode_HOLD,                 -- Hold PC constant (for bubbles)
143           PCWriteCtrl => PCWriteCtrl_WRITE_CALC,          -- Write to PC
144           Off8        => (others => '0'),                 -- unused
145           Off12       => (others => '0'),                 -- unused
146           ConditionalBranch  => '0',                      -- Not performing a conditional branch
147           Condition   => '0',                             -- Not performing a conditional branch
148           DelayBranch => '0'                              -- Assume no branch by default
149       );
150
151       -- Default control signals for System Control
152       constant DEFAULT_SYS_CTRL : sys_ctrl_t := (
153           PRWriteEn   => '0',                             -- Do not write to PR (Procedure Register)
154           RegCtrl     => SysRegCtrl_None,                 -- Do nothing with system registers
155           RegSel      => SysRegSel_SR,                    -- unused
156           RegSrc      => SysRegSrc_RegB                   -- unused
157       );
158
159       constant DEFAULT_CTRL : ctrl_t := (
160           ALUCtrl      => DEFAULT_ALU_CTRL,
161           REGCtrl      => DEFAULT_REG_CTRL,
162           PMAUCtrl     => DEFAULT_PMAU_CTRL,
163           DMAUCtrl     => DEFAULT_DMAU_CTRL,
164           SysCtrl      => DEFAULT_SYS_CTRL,
165           MemCtrl      => DEFAULT_MEM_CTRL
166       );
167
168
169       -- The instruction register.
170       signal IR : std_logic_vector(15 downto 0);
171
172       -- Aliases for instruction arguments.
173       -- There are 13 instruction formats, shown below:
174       --
175       -- Key:
176       --  xxxx: instruction code
177       --  mmmm: Source register
178       --  nnnn: Destination register
179       --  iiii: immediate data
180       --  dddd: displacment
181
182       -- 0 format:   xxxx xxxx xxxx xxxx
183       -- n format:   xxxx nnnn xxxx xxxx
184       -- m format:   xxxx mmmm xxxx xxxx
185       -- nm format:  xxxx nnnn mmmm xxxx
186       -- md format:  xxxx xxxx mmmm dddd
187       -- nd4 format: xxxx xxxx nnnn dddd
188       -- nmd format: xxxx nnnn mmmm dddd
189       -- d format:   xxxx xxxx dddd dddd
190       -- d12 format: xxxx dddd dddd dddd
191       -- nd8 format: xxxx nnnn dddd dddd
192       -- i format:   xxxx xxxx iiii iiii
193       -- ni format:  xxxx nnnn iiii iiii
194
195       -- n format
196       alias n_format_n    : std_logic_vector(3 downto 0) is IR(11 downto 8);
197
198       -- m format
```

```
199       alias m_format_m    : std_logic_vector(3 downto 0) is IR(11 downto 8);
200
201       -- nm format
202       alias nm_format_n   : std_logic_vector(3 downto 0) is IR(11 downto 8);
203       alias nm_format_m   : std_logic_vector(3 downto 0) is IR(7 downto 4);
204
205       -- md format
206       alias md_format_m   : std_logic_vector(3 downto 0) is IR(7 downto 4);
207       alias md_format_d   : std_logic_vector(3 downto 0) is IR(3 downto 0);
208
209       -- nd4 format
210       alias nd4_format_n  : std_logic_vector(3 downto 0) is IR(7 downto 4);
211       alias nd4_format_d  : std_logic_vector(3 downto 0) is IR(3 downto 0);
212
213       -- nmd format
214       alias nmd_format_n  : std_logic_vector(3 downto 0) is IR(11 downto 8);
215       alias nmd_format_m  : std_logic_vector(3 downto 0) is IR(7 downto 4);
216       alias nmd_format_d  : std_logic_vector(3 downto 0) is IR(3 downto 0);
217
218       -- d format
219       alias d_format_d    : std_logic_vector(7 downto 0) is IR(7 downto 0);
220
221       -- d12 format
222       alias d12_format_d  : std_logic_vector(11 downto 0) is IR(11 downto 0);
223
224       -- nd8 format
225       alias nd8_format_n  : std_logic_vector(3 downto 0) is IR(11 downto 8);
226       alias nd8_format_d  : std_logic_vector(7 downto 0) is IR(7 downto 0);
227
228       -- i format
229       alias i_format_i    : std_logic_vector(7 downto 0) is IR(7 downto 0);
230
231       -- ni format
232       alias ni_format_n   : std_logic_vector(3 downto 0) is IR(11 downto 8);
233       alias ni_format_i   : std_logic_vector(7 downto 0) is IR(7 downto 0);
234
235
236       -- control signals to control memory interface
237       signal MemEnable        : std_logic;                        -- if memory needs to be accessed (read or write)
238       signal MemAddrSel       : std_logic;
239       signal ReadWrite        : std_logic;                        -- if should do memory read (0) or write (1)
240       signal MemMode          : std_logic_vector(1 downto 0);     -- if memory access should be by byte, word, or longword
241       signal MemSel           : std_logic;                        -- select memory address source, from DMAU output (0) or PMAU output
242
243       signal Immediate        : std_logic_vector(7 downto 0);     -- 8-bit immediate
244       signal ImmediateMode    : std_logic;                        -- Immediate extension mode
245       signal MemOutSel        : std_logic_vector(2 downto 0);     -- what should be output to memory
246       signal TFlagSel         : std_logic_vector(2 downto 0);     -- source for next value of T flag
247       signal ExtMode          : std_logic_vector(1 downto 0);     -- mode for extending register value (zero or signed)
248
249           -- ALU control signals
250       signal ALUOpBSel        : std_logic;                        -- input mux to Operand B, either RegB (0) or Immediate (1)
251       signal LoadA            : std_logic;                        -- determine if OperandA is loaded ('1') or zeroed ('0')
252       signal FCmd             : std_logic_vector(3 downto 0);     -- F-Block operation
253       signal CinCmd           : std_logic_vector(1 downto 0);     -- carry in operation
254       signal SCmd             : std_logic_vector(2 downto 0);     -- shift operation
255       signal ALUCmd           : std_logic_vector(1 downto 0);     -- ALU result select
256       signal TCmpSel          : std_logic_vector(2 downto 0);     -- how to compute T from ALU status flags
257
258           -- register array control signals
259       signal RegDataInSel     : std_logic_vector(3 downto 0);     -- source for register input data
260       signal RegEnableIn      : std_logic;                        -- if data should be written to an input register
261       signal RegInSel         : integer  range 15 downto 0;       -- which register to write data to
262       signal RegASel          : integer  range 15 downto 0;       -- which register to read to bus A
263       signal RegBSel          : integer  range 15 downto 0;       -- which register to read to bus B
264       signal RegAxInSel       : integer  range 15 downto 0;       -- which address register to write to
```

```
265      signal RegAxStore       : std_logic;                        -- if data should be written to the address register
266      signal RegA1Sel         : integer  range 15 downto 0;       -- which register to read to address bus 1
267      signal RegA2Sel         : integer  range 15 downto 0;       -- which register to read to address bus 2
268
269          -- DMAU signals
270      signal GBRWriteEn       : std_logic;                        -- GBR write enable, active high
271      signal DMAUOff4         : std_logic_vector(3 downto 0);     -- 4-bit offset
272      signal DMAUOff8         : std_logic_vector(7 downto 0);     -- 8-bit offset
273      signal BaseSel          : std_logic_vector(1 downto 0);     -- which base register source to select
274      signal IndexSel         : std_logic_vector(1 downto 0);     -- which index source to select
275      signal OffScalarSel     : std_logic_vector(1 downto 0);     -- what to scale the offset by (1, 2, 4)
276      signal IncDecSel        : std_logic_vector(1 downto 0);     -- post-increment or pre-decrement the base
277
278          -- PMAU signals
279      signal PCAddrMode       : std_logic_vector(2 downto 0);     -- What PC addressing mode is desired.
280      signal PRWriteEn        : std_logic;                        -- Enable writing to PR.
281      signal PMAUOff8         : std_logic_vector(7 downto 0);     -- 8-bit offset for relative addressing.
282      signal PMAUOff12        : std_logic_vector(11 downto 0);    -- 12-bit offset for relative addressing.
283      signal PCWriteCtrl      : std_logic_vector(1 downto 0);     -- What to write to the PC register inside
284                                                                  -- the PMAU. Can either hold current value,
285                                                                  -- or write calculated PC.
286      signal ConditionalBranch: std_logic;                        -- If should perform a conditional branch (checked during ID stage)
287      signal Condition        : std_logic;                        -- Condition to check for a conditional branch
288      signal DelayBranch      : std_logic;                        -- If branch should be delayed
289
290          -- System control signals
291      signal SysRegCtrl       : std_logic_vector(1 downto 0);     -- how to update system registers
292      signal SysRegSel        : std_logic_vector(2 downto 0);     -- system register select
293      signal SysRegSrc        : std_logic_vector(1 downto 0);     -- source for data to input into a system register
294
295      -- Pipelining signals
296      type signal_array is array (0 to 3) of ctrl_t;
297
298      -- Define which set of signals is associated with each stage in the 5-stage pipeline
299      -- We don't include instruction fetch (IF) because it doesn't involve any
300      -- control signals - latching in IR data takes a clock on its own, so it's
301      -- implicitly a stage in the pipeline.
302      constant STAGE_ID   : integer := 0;     -- instruction decode
303      constant STAGE_X    : integer := 1;     -- execute instruction
304      constant STAGE_MA   : integer := 2;     -- memory access
305      constant STAGE_WR   : integer := 3;     -- writeback
306
307      signal decoded_signals  : ctrl_t;       -- signals combinatorially decoded from the IR
308
309      signal pipeline         : signal_array;                 -- pipeline the decoded control signals (DFFs)
310      signal pipeline_in_en   : std_logic;                    -- if the first stage in the pipeline (IF) should be enabled
311      signal pipeline_en      : std_logic_vector(0 to 3);     -- if each stage in the pipeline should be enabled
312
313      -- We need to insert a bubble if instruction fetch (IF) contention with
314      -- memory access (MA). In this case, MA takes precedence, and IF is delayed
315      -- by a clock. This is implemented by 1) not shifting forward stages IF, ID,
316      -- and X, 2) holding the PC constant, and 3) ignoring the current value on
317      -- the data bus
318      signal BubbleIF         : std_logic;
319
320      signal ShouldBranch     : std_logic;    -- if a branch should be performed
321
322  begin
323
324      decoded_signals <= (
325          ALUCtrl => (
326              OpBSel => ALUOpBSel,
327              LoadA => LoadA,
328              FCmd => FCmd,
329              CinCmd => CinCmd,
330              SCmd => SCmd,
```

```
331             ALUCmd => ALUCmd,
332             TCmpSel => TCmpSel,
333             Immediate => Immediate,
334             ImmediateMode => ImmediateMode,
335             ExtMode => ExtMode,
336             TFlagSel => TFlagSel
337         ),
338
339         RegCtrl => (
340             DataInSel => RegDataInSel,
341             EnableIn => RegEnableIn,
342             InSel => RegInSel,
343             ASel => RegASel,
344             BSel => RegBSel
345         ),
346
347         DMAUCtrl => (
348             GBRWriteEn => GBRWriteEn,
349             Off4 => DMAUOff4,
350             Off8 => DMAUOff8,
351             BaseSel => BaseSel,
352             IndexSel => IndexSel,
353             OffScalarSel => OffScalarSel,
354             IncDecSel => IncDecSel,
355             AxInSel => RegAxInSel,
356             AxStore => RegAxStore,
357             A1Sel => RegA1Sel,
358             A2Sel => RegA2Sel
359         ),
360
361         PMAUCtrl => (
362             PCAddrMode => PCAddrMode,
363             Off8 => PMAUOff8,
364             Off12 => PMAUOff12,
365             PCWriteCtrl => PCWriteCtrl,
366             ConditionalBranch => ConditionalBranch,
367             Condition => Condition,
368             DelayBranch => DelayBranch
369         ),
370         SysCtrl => (
371             PRWriteEn => PRWriteEn,
372             RegCtrl => SysRegCtrl,
373             RegSel => SysRegSel,
374             RegSrc => SysRegSrc
375         ),
376         MemCtrl =>  (
377             Enable => MemEnable,
378             AddrSel => MemAddrSel,
379             ReadWrite => ReadWrite,
380             Mode => MemMode,
381             OutSel => MemOutSel,
382             Sel => MemSel
383         )
384     );
385
386     -- Currently, IF occurs every clock, so if the instruction currently in the
387     -- MA stage requires memory access, a bubble needs to be inserted.
388     BubbleIF <= pipeline(STAGE_MA).MemCtrl.Enable and pipeline_en(STAGE_MA);
389
390     mem_access: process(all)
391     begin
392         if BubbleIF = '1' and pipeline_en(STAGE_MA) = '1' then
393             -- Perform memory access, ignoring IF this clock
394             MemCtrl <= pipeline(STAGE_MA).MemCtrl;
395         else
396             -- fetch instruction like usual
```

```
397             MemCtrl <= (
398                 Enable => '1',
399                 AddrSel => MemAddrSel_PMAU,
400                 ReadWrite => ReadWrite_READ,
401                 Mode => WordMode,
402                 OutSel => MemOutSel,
403                 Sel => MemSel_ROM
404             );
405         end if ;
406     end process mem_access;
407 
408     ShouldBranch <= '1' when pipeline(STAGE_ID).PMAUCtrl.ConditionalBranch = '1' and pipeline(STAGE_ID).PMAUCtrl.Condition = TFlagIn else '0';
409 
410     program_access: process(all)
411     begin
412         if BubbleIF = '1' then
413             -- Hold PC still for bubble
414             PMAUCtrl <= HOLD_PMAU_CTRL;
415         elsif pipeline_en(STAGE_X) then
416             -- Check if should branch
417             if pipeline(STAGE_ID).PMAUCtrl.ConditionalBranch = '1' then
418                 if ShouldBranch = '1' then
419                     PMAUCtrl <= pipeline(STAGE_ID).PMAUCtrl;
420                 else
421                     -- Increment PC (don't branch)
422                     PMAUCtrl <= DEFAULT_PMAU_CTRL;
423                 end if;
424             else
425                 -- Perform instruction like normal
426                 PMAUCtrl <= pipeline(STAGE_X).PMAUCtrl;
427             end if;
428         else
429             -- Update PC like normal
430             PMAUCtrl <= DEFAULT_PMAU_CTRL;
431         end if;
432     end process program_access;
433 
434     ALUCtrl <= pipeline(STAGE_X).ALUCtrl;
435 
436     register_access: process(all)
437     begin
438         if BubbleIF = '1' then
439             RegCtrl <= pipeline(STAGE_MA).RegCtrl;
440         elsif pipeline_en(STAGE_X) = '1' then
441             RegCtrl <= pipeline(STAGE_X).RegCtrl;
442         else
443             RegCtrl <= DEFAULT_REG_CTRL;
444         end if;
445     end process register_access;
446 
447     data_access: process(all)
448     begin
449         if pipeline_en(STAGE_MA) = '1' then
450             DMAUCtrl <= pipeline(STAGE_MA).DMAUCtrl;
451         else
452             DMAUCtrl <= DEFAULT_DMAU_CTRL;
453         end if;
454     end process data_access;
455 
456     sys_control: process(all)
457     begin
458         if BubbleIF = '1' then
459             SysCtrl <= pipeline(STAGE_MA).SysCtrl;
460         elsif pipeline_en(STAGE_X) = '1' then
461             SysCtrl <= pipeline(STAGE_X).SysCtrl;
```

```
462          else
463              SysCtrl <= DEFAULT_SYS_CTRL;
464          end if;
465      end process sys_control;
466
467      -- Decode the current instruction combinatorially
468      decode_proc: process (IR)
469          variable l : line;
470      begin
471
472
473          -- Default flag values are set here (these shouldn't change CPU state).
474          -- This is so that not every control signal has to be set in every single
475          -- instruction case. If an instruction enables writing to memory/registers,
476          -- then ensure that the default value is set here as "disable" to prevent
477          -- writes on the clocks following an instruction.
478
479          -- Not accessing memory
480          MemEnable  <= '0';
481          ReadWrite  <= 'X';
482          MemMode    <= "XX";
483          MemOutSel              <= "XXX";
484          MemSel     <= MemSel_RAM;       -- access data memory by default.
485          MemAddrSel <= MemAddrSel_DMAU;  -- access data memory by default.
486
487          -- Register enables
488          RegEnableIn <= '0';             -- Disable register write
489          RegAxStore  <= '0';             -- Disable writing to address register.
490          TFlagSel    <= TFlagSel_T;      -- Keep T flag the same
491          GBRWriteEn  <= '0';             -- Don't write to GBR.
492          PRWriteEn   <= '0';             -- Don't write to PR.
493
494          SysRegCtrl    <= SysRegCtrl_NONE;      -- system register not selected
495          ImmediateMode <= ImmediateMode_SIGN;   -- sign-extend immediates by defualt
496          ExtMode       <= Ext_SignB_RegA;
497
498          PCAddrMode <= PCAddrMode_INC;
499
500          PCWriteCtrl <= PCWriteCtrl_WRITE_CALC;  -- Write the calculated PC by default.
501
502          ConditionalBranch  <= '0';  -- not performing a conditional branch
503          Condition          <= '0';  -- not performing a conditional branch
504          DelayBranch        <= '0';  -- not performing a delayed branch
505
506          if std_match(IR, ADD_RM_RN) then
507              -- ADD{C,V} Rm, Rn
508
509              LogWithTime(l, "sh2_control.vhd: Decoded Add R" & to_string(to_integer(unsigned(nm_format_m))) &
510                             " , R" & to_string(to_integer(unsigned(nm_format_n))), LogFile);
511
512              -- Register array signals
513              RegASel <= to_integer(unsigned(nm_format_n));
514              RegBSel <= to_integer(unsigned(nm_format_m));
515
516              RegInSel             <= to_integer(unsigned(nm_format_n));
517              RegDataInSel         <= RegDataIn_ALUResult;
518              RegEnableIn <= '1';
519
520              -- Bit-decoding T flag select (None, Carry, Overflow)
521              TFlagSel <= '0' & IR(1 downto 0);
522
523              -- ALU signals for addition
524              ALUOpBSel <= ALUOpB_RegB;
525              LoadA     <= '1';
526              FCmd      <= FCmd_B;
527
```

```
528             -- Bit-decode carry in value
529             CinCmd <= CinCmd_CIN when IR(1 downto 0) = "10" else    -- ADDC
530                       CinCmd_ZERO;                                  -- ADD, ADDV
531
532             SCmd   <= "XXX";
533             ALUCmd <= ALUCmd_ADDER;
534
535
536         elsif std_match(IR, SUB_RM_RN) then
537             -- SUB{C,V} Rm, Rn
538
539             -- Register array signals
540             RegASel <= to_integer(unsigned(nm_format_n));
541             RegBSel <= to_integer(unsigned(nm_format_m));
542
543             RegInSel             <= to_integer(unsigned(nm_format_n));
544             RegDataInSel         <= RegDataIn_ALUResult;
545             RegEnableIn <= '1';
546
547             -- Bit-decoding T flag select (None, Carry, Overflow)
548             TFlagSel <= '0' & IR(1 downto 0);
549
550             -- ALU signals for subtraction
551             ALUOpBSel <= ALUOpB_RegB;
552             LoadA     <= '1';
553             FCmd      <= FCmd_BNOT;
554
555             -- Bit-decode carry in value
556             CinCmd <= CinCmd_CINBAR when IR(1 downto 0) = "10" else     -- SUBC
557                       CinCmd_ONE;                                       -- SUB, SUBV
558
559             SCmd   <= "XXX";
560             ALUCmd <= ALUCmd_ADDER;
561
562         elsif std_match(IR, DT_RN) then
563             -- DT Rn
564
565             -- Register array signals
566             RegASel <= to_integer(unsigned(nm_format_n));
567             Immediate <= (others => '0');
568
569             RegInSel             <= to_integer(unsigned(nm_format_n));
570             RegDataInSel         <= RegDataIn_ALUResult;
571             RegEnableIn <= '1';
572
573             -- Bit-decoding T flag select (None, Carry, Overflow)
574             TFlagSel <= TFlagSel_Zero;
575
576             -- ALU signals to subtract 1 from Rn
577             ALUOpBSel <= ALUOpB_Imm;
578             LoadA     <= '1';
579             FCmd      <= FCmd_BNOT;
580             CinCmd    <= CinCmd_ZERO;
581             SCmd      <= "XXX";
582             ALUCmd    <= ALUCmd_ADDER;
583
584         elsif std_match(IR, NEG_RM_RN) then
585             -- NEG{C} Rm, Rn
586
587             -- Register array signals
588             RegASel <= to_integer(unsigned(nm_format_n));
589             RegBSel <= to_integer(unsigned(nm_format_m));
590
591             RegInSel             <= to_integer(unsigned(nm_format_n));
592             RegDataInSel         <= RegDataIn_ALUResult;
593             RegEnableIn <= '1';
```

```
594
595             -- Bit-decoding T flag select
596             TFlagSel <= TFlagSel_Carry when IR(0) = '0' else    -- NEGC
597                                 TFlagSel_T;                             -- NEG
598
599             -- ALU signals for negation
600             ALUOpBSel <= ALUOpB_RegB;
601             LoadA     <= '0';
602             FCmd      <= FCmd_BNOT;
603
604             -- Bit-decode carry in value
605             CinCmd <= CinCmd_CINBAR when IR(0) = '0' else   -- NEGC
606                       CinCmd_ONE;                           -- NEG
607
608             SCmd   <= "XXX";
609             ALUCmd <= ALUCmd_ADDER;
610
611         elsif std_match(IR, EXT_RM_RN) then
612             -- EXT{U,S}.{B,W Rm, Rn}
613
614             -- Register array signals
615             RegASel <= to_integer(unsigned(nm_format_n));
616             RegBSel <= to_integer(unsigned(nm_format_m));
617
618             RegInSel             <= to_integer(unsigned(nm_format_n));
619             RegDataInSel         <= RegDataIn_Ext;
620             ExtMode              <= IR(1 downto 0);     -- bit-decode extension mode
621             RegEnableIn <= '1';
622
623         elsif std_match(IR, ADD_IMM_RN) then
624             -- ADD #imm, Rn
625
626             -- Register array signals
627             RegASel <= to_integer(unsigned(nm_format_n));
628
629             RegInSel             <= to_integer(unsigned(nm_format_n));
630             RegDataInSel         <= RegDataIn_ALUResult;
631             RegEnableIn <= '1';
632             Immediate            <= ni_format_i;
633
634             -- ALU signals for addition
635             ALUOpBSel <= ALUOpB_Imm;
636             LoadA     <= '1';
637             FCmd      <= FCmd_B;
638             CinCmd    <= CinCmd_ZERO;
639             SCmd      <= "XXX";
640             ALUCmd    <= ALUCmd_ADDER;
641
642         elsif std_match(IR, LOGIC_RM_RN) then
643             -- {AND, TST, OR, XOR} Rm, Rn
644
645             -- Register array signals
646             RegASel <= to_integer(unsigned(nm_format_n));
647             RegBSel <= to_integer(unsigned(nm_format_m));
648
649             RegInSel             <= to_integer(unsigned(nm_format_n));
650             RegDataInSel         <= RegDataIn_ALUResult;
651             RegEnableIn <= IR(1) or IR(0);   -- exclude TST
652
653             -- Enable TFlagSel for TST
654             TFlagSel <= TFlagSel_Zero when IR(1 downto 0) = "00"    -- TST
655                                 else TFlagSel_T;                            -- AND, OR, XOR
656
657             -- ALU signals for logic instructions using the FBlock
658             ALUOpBSel <= ALUOpB_RegB;
659             LoadA     <= '1';
```

```
660
661             -- Bit-decode f-block operation
662             FCmd <= FCmd_AND when IR(1) = '0'           else    -- AND, TST
663                     FCmd_XOR when IR(1 downto 0) = "10" else    -- XOR
664                     FCmd_OR;                                    -- OR
665
666             CinCmd <= CinCmd_ZERO;
667             SCmd   <= "XXX";
668             ALUCmd <= ALUCmd_FBLOCK;
669
670         elsif std_match(IR, LOGIC_IMM_R0) then
671             -- {AND, TST, OR, XOR} immediate, R0
672
673             -- Register array signals
674             RegASel <= 0;
675
676             RegInSel            <= 0;
677             RegDataInSel        <= RegDataIn_ALUResult;
678             RegEnableIn <= IR(9) or IR(8);   -- exclude TST
679             Immediate           <= i_format_i;
680             ImmediateMode       <= ImmediateMode_ZERO;
681
682             -- Enable TFlagSel for TST
683             TFlagSel <= TFlagSel_Zero when IR(9 downto 8) = "00"    -- TST
684                                 else TFlagSel_T;                            -- AND, OR, XOR
685
686             -- ALU signals for logic instructions using the FBlock
687             ALUOpBSel <= ALUOpB_Imm;
688             LoadA     <= '1';
689
690             -- Bit-decode f-block operation
691             FCmd <= FCmd_AND when IR(9) = '0' else              -- AND, TST
692                     FCmd_XOR when IR(9 downto 8) = "10" else    -- XOR
693                     FCmd_OR;                                    -- OR
694
695             CinCmd <= CinCmd_ZERO;
696             SCmd   <= "XXX";
697             ALUCmd <= ALUCmd_FBLOCK;
698
699         elsif std_match(IR, NOT_RM_RN) then
700             -- NOT Rm, Rn
701
702             -- Register array signals
703             RegASel <= to_integer(unsigned(nm_format_n));
704             RegBSel <= to_integer(unsigned(nm_format_m));
705
706             RegInSel            <= to_integer(unsigned(nm_format_n));
707             RegDataInSel        <= RegDataIn_ALUResult;
708             RegEnableIn <= '1';
709
710             -- ALU signals for logical negation
711             ALUOpBSel <= ALUOpB_RegB;
712             LoadA     <= '1';
713             FCmd      <= FCmd_BNOT;
714             CinCmd    <= CinCmd_ZERO;
715             SCmd      <= "XXX";
716             ALUCmd    <= ALUCmd_FBLOCK;
717
718         elsif std_match(IR, CMP_EQ_IMM) then
719             -- CMP/EQ #Imm, R0
720
721             -- Register array signals
722             RegASel <= 0;
723
724             Immediate           <= i_format_i;
725             ImmediateMode       <= ImmediateMode_ZERO;
```

```
726
727             -- Compute T flag based on ALU flags
728             TFlagSel <= TFlagSel_CMP;
729             TCMPSel <= TCmp_EQ;
730
731             -- ALU Instructions that perform a subtraction (Rn - immediate) so that
732             -- the ALU output flags can be used to compute the T flag
733             ALUOpBSel <= ALUOpB_Imm;
734             LoadA     <= '1';
735             FCmd      <= FCmd_BNOT;
736             CinCmd <= CinCmd_ONE;
737             SCmd   <= "XXX";
738             ALUCmd <= ALUCmd_ADDER;
739
740         elsif std_match(IR, CMP_RM_RN) then
741             -- CMP/XX 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             -- Compute T flag based on ALU flags
748             TFlagSel <= TFlagSel_CMP;
749             TCMPSel <= IR(2 downto 0);              -- bit decode T flag CMP condition
750
751             -- ALU Instructions that perform a subtraction (Rn - Rm) so that
752             -- the ALU output flags can be used to compute the T flag
753             ALUOpBSel <= ALUOpB_RegB;
754             LoadA     <= '1';
755             FCmd      <= FCmd_BNOT;
756             CinCmd <= CinCmd_ONE;
757             SCmd   <= "XXX";
758             ALUCmd <= ALUCmd_ADDER;
759
760         elsif std_match(IR, CMP_STR_RM_RN) then
761             -- CMP/STR Rm, Rn
762
763             -- Register array signals
764             RegASel <= to_integer(unsigned(nm_format_n));
765             RegBSel <= to_integer(unsigned(nm_format_m));
766
767             -- Compute T flag based on ALU flags
768             TFlagSel <= TFlagSel_CMP;
769             TCMPSel <= TCMP_STR;
770
771         elsif std_match(IR, CMP_RN) then
772             -- CMP/{PL/PZ} Rn
773
774             -- Register array signals
775             RegASel <= to_integer(unsigned(nm_format_n));
776
777             -- Compare to 0
778             Immediate <= (others => '0');
779
780             -- Compute T flag based on ALU flags
781             TFlagSel <= TFlagSel_CMP;
782             TCMPSel <= IR(2) & "11";                -- bit decode CMP mode (either GT or GE)
783
784             -- ALU Instructions that perform a subtraction (Rn - 0) so that
785             -- the ALU output flags can be used to compute the T flag
786             ALUOpBSel <= ALUOpB_Imm;
787             LoadA     <= '1';
788             FCmd      <= FCmd_BNOT;
789             CinCmd <= CinCmd_ONE;
790             SCmd   <= "XXX";
791             ALUCmd <= ALUCmd_ADDER;
```

```
792
793         elsif std_match(IR, SHIFT_RN) then
794             -- Shift operations
795             -- {ROTL, ROTR, ROTCL, ROTCR, SHAL, SHAR, SHLL, SHLR} Rn
796             -- Uses bit decoding to compute control signals (to reduce code size)
797
798             -- Register array signals
799             RegASel              <= to_integer(unsigned(n_format_n));
800             RegInSel             <= to_integer(unsigned(n_format_n));
801             RegDataInSel         <= RegDataIn_ALUResult;
802             RegEnableIn <= '1';
803
804             TFlagSel <= TFlagSel_Carry;
805
806             -- ALU signals
807             ALUOpBSel <= ALUOpB_RegB;
808             LoadA     <= '1';
809             FCmd      <= "XXXX";
810
811             -- Bit-decode carry command
812             CinCmd    <= CinCmd_CIN when (IR(5) and IR(2)) = '1' else   -- ROTCL, ROTCR
813                          CinCmd_ZERO;                                   -- all others
814
815             SCmd   <= IR(0) & IR(2) & IR(5);  -- bit-decode shift operation
816             ALUCmd <= ALUCmd_SHIFT;
817
818         elsif std_match(IR, BSHIFT_RN) then
819             -- Barrel shift operations
820             -- {SHLL,SHLR}{2,8,16} Rn
821             -- Uses bit decoding to compute control signals (to reduce code size)
822
823             -- Register array signals
824             RegASel              <= to_integer(unsigned(n_format_n));
825             RegInSel             <= to_integer(unsigned(n_format_n));
826             RegDataInSel         <= RegDataIn_ALUResult;
827             RegEnableIn <= '1';
828
829             TFlagSel <= TFlagSel_T;
830
831             -- ALU signals
832             LoadA     <= '1';
833             SCmd   <= IR(5) & IR(4) & IR(0);  -- bit-decode barrel shift operation
834             ALUCmd <= ALUCmd_BSHIFT;
835
836         -- Data Transfer Instruction ----------------------------------------
837
838         -- MOV #imm, Rn
839         -- ni format
840         elsif std_match(IR, MOV_IMM_RN) then
841
842             LogWithTime(l, "sh2_control.vhd: Decoded MOV H'" & to_hstring(ni_format_i) &
843                            ", R" & to_string(slv_to_uint(ni_format_n)), LogFile);
844
845             RegInSel             <= to_integer(unsigned(ni_format_n));
846             RegDataInSel         <= RegDataIn_Immediate;
847             RegEnableIn <= '1';
848             Immediate            <= ni_format_i;
849
850         -- MOV.W @(disp, PC), Rn
851         -- nd8 format
852         -- NOTE: Testing this assumes MOV into memory works.
853         --
854         elsif std_match(IR, MOV_W_AT_DISP_PC_RN) then
855             LogWithTime(l,
856             "sh2_control.vhd: Decoded MOV.W @(0x" & to_hstring(nd8_format_d) &
857             ", PC), R" & to_string(slv_to_uint(nd8_format_n)), LogFile);
```

```
858
859
860             RegInSel             <= to_integer(unsigned(nd8_format_n));   -- Writing to register n
861             RegDataInSel         <= RegDataIn_DB;                         -- Writing output of data bus to register.
862             RegEnableIn <= '1';                                  -- Writes to register.
863
864             RegASel <= to_integer(unsigned(nd8_format_n));
865
866             -- Instruction reads word from program memory (ROM).
867             MemEnable <= '1';
868             ReadWrite <= ReadWrite_READ;
869             MemMode   <= WordMode;
870             MemSel    <= MemSel_ROM;
871
872             -- DMAU signals for PC Relative addressing with displacement (word mode)
873             BaseSel      <= BaseSel_PC;
874             IndexSel     <= IndexSel_OFF8;
875             OffScalarSel <= OffScalarSel_TWO;
876             IncDecSel    <= IncDecSel_NONE;
877             DMAUOff8     <= nd8_format_d;
878
879
880         -- MOV.L @(disp, PC), Rn
881         -- nd8 format
882         elsif std_match(IR, MOV_L_AT_DISP_PC_RN) then
883             LogWithTime(l,
884             "sh2_control.vhd: Decoded MOV.L @(0x" & to_hstring(nd8_format_d) &
885             ", PC), R" & to_string(slv_to_uint(nd8_format_n)), LogFile);
886
887             RegInSel             <= to_integer(unsigned(nd8_format_n));  -- Writing to register n
888             RegDataInSel         <= RegDataIn_DB;                        -- Writing output of data bus to register.
889             RegEnableIn <= '1';                                 -- Writes to register.
890
891             -- Instruction reads from longword memory.
892             MemEnable <= '1';
893             ReadWrite <= ReadWrite_READ;
894             MemMode   <= LongwordMode;
895             MemSel    <= MemSel_ROM;
896
897             -- DMAU signals for PC Relative addressing with displacement (longword mode)
898             BaseSel      <= BaseSel_PC;
899             IndexSel     <= IndexSel_OFF8;
900             OffScalarSel <= OffScalarSel_FOUR;
901             IncDecSel    <= IncDecSel_NONE;
902             DMAUOff8     <= nd8_format_d;
903
904
905         -- MOV Rm, Rn
906         -- nm format
907         -- Note: for bit decoding, this must be done before MOV_AT_RM_RN
908         elsif std_match(IR, MOV_RM_RN) then
909             LogWithTime(l,
910               "sh2_control.vhd: Decoded MOV R" & to_string(slv_to_uint(nm_format_m)) &
911               "R" & to_string(slv_to_uint(nm_format_n)) , LogFile);
912
913             -- report "Instruction: MOV Rm, Rn";
914             RegBSel              <= to_integer(unsigned(nm_format_m));
915             RegInSel             <= to_integer(unsigned(nm_format_n));
916             RegDataInSel         <= RegDataIn_RegB;
917             RegEnableIn <= '1';
918
919         -- MOV.X Rm, @Rn
920         -- nm format
921         elsif std_match(IR, MOV_RM_AT_RN) then
922             LogWithTime(l,
923               "sh2_control.vhd: Decoded MOV.X R" & to_string(slv_to_uint(nm_format_m)) &
```

```
924                ", @R" & to_string(slv_to_uint(nm_format_n)) , LogFile);
925
926             -- Writes a byte to memory to memory
927             MemEnable <= '1';             -- Uses memory.
928             ReadWrite <= ReadWrite_WRITE; -- Writes.
929             MemMode   <= IR(1 downto 0);  -- bit decode memory mode
930
931             MemOutSel <= MemOut_RegB; -- Output RegB (Rm) to memory data bus.
932
933             RegBSel  <= to_integer(unsigned(nm_format_m)); -- RegB is Rm.
934             RegA1Sel <= to_integer(unsigned(nm_format_n)); -- RegA is @(Rn)
935
936             -- DMAU signals (for Indirect Register Addressing)
937             BaseSel      <= BaseSel_REG;
938             IndexSel     <= IndexSel_NONE;
939             OffScalarSel <= OffScalarSel_ONE;
940             IncDecSel    <= IncDecSel_NONE;
941
942         -- MOV.X @Rm, Rn
943         -- nm format
944         -- Note: for bit decoding, this must be done after MOV_RM_RN
945         elsif std_match(IR, MOV_AT_RM_RN) then
946             LogWithTime(l,
947               "sh2_control.vhd: Decoded MOV.X @R" & to_string(slv_to_uint(nm_format_m)) &
948               ", R" & to_string(slv_to_uint(nm_format_n)) , LogFile);
949
950             -- Instruction reads byte from memory.
951             MemEnable <= '1';            -- Instr does memory access.
952             ReadWrite <= ReadWrite_READ; -- Instr reads from memory.
953             MemMode   <= IR(1 downto 0); -- bit decode memory mode
954
955             -- DMAU signals for Indirect Register addressing.
956             BaseSel      <= BaseSel_REG;
957             IndexSel     <= IndexSel_NONE;
958             OffScalarSel <= OffScalarSel_ONE;
959             IncDecSel    <= IncDecSel_NONE;
960
961             -- Output @(Rm) to RegA2.
962             RegA2Sel <= to_integer(unsigned(nm_format_m));
963
964             RegInSel             <= to_integer(unsigned(nm_format_n));
965             RegDataInSel         <= RegDataIn_DB;
966             RegEnableIn <= '1';
967
968
969         -- MOV.B Rm, @-Rn
970         -- nm format
971         elsif std_match(IR, MOV_RM_AT_MINUS_RN) then
972             LogWithTime(l,
973               "sh2_control.vhd: Decoded MOV.X R" & to_string(slv_to_uint(nm_format_m)) &
974               ", @-R" & to_string(slv_to_uint(nm_format_n)) , LogFile);
975
976             -- Writes a byte to memory
977             MemEnable <= '1';             -- Uses memory.
978             ReadWrite <= ReadWrite_WRITE; -- Writes.
979             MemMode   <= IR(1 downto 0);  -- bit decode memory mode
980
981             MemOutSel <= MemOut_RegB; -- Output RegB (Rm) to memory data bus.
982
983             RegBSel                <= to_integer(unsigned(nm_format_m));  -- Output Rm from RegB output.
984             RegA1Sel               <= to_integer(unsigned(nm_format_n));  -- Output @(Rn) from RegA1 output.
985             RegAxInSel             <= to_integer(unsigned(nm_format_n));  -- Store calculated address into Rn
986             RegAxStore <= '1';                                 -- Enable writes to address registers.
987
988             -- DMAU signals (for Pre-decrement indirect register addressing)
989             BaseSel      <= BaseSel_REG;
```

```
990             IndexSel     <= IndexSel_NONE;
991             OffScalarSel <= IR(1 downto 0);         -- bit decode offset scalar factor
992             IncDecSel    <= IncDecSel_PRE_DEC;
993
994         -- MOV.B @Rm+, Rn
995         -- nm format
996         elsif std_match(IR, MOV_AT_RM_PLUS_RN) then
997             LogWithTime(l,
998               "sh2_control.vhd: Decoded MOV.{B,W,L} @R" & to_string(slv_to_uint(nm_format_m)) &
999               "+, R" & to_string(slv_to_uint(nm_format_n)) , LogFile);
1000
1001            -- MOV with post-increment. This Instruction reads a byte, word,
1002            -- or longword from an address in Rm, into Rn. The address is
1003            -- incremented and stored in Rm after the value is retrieved.
1004
1005            -- Reads a byte from memory.
1006            MemEnable <= '1';             -- Uses memory.
1007            ReadWrite <= ReadWrite_READ;  -- Reads.
1008            MemMode   <= IR(1 downto 0);  -- bit-decode memory word mode
1009
1010            -- Output @Rm from RegA2
1011            RegA2Sel <= to_integer(unsigned(nm_format_m));
1012
1013            -- Write output of Data Bus to Rn
1014            RegInSel             <= to_integer(unsigned(nm_format_n));
1015            RegDataInSel         <= RegDataIn_DB;
1016            RegEnableIn <= '1';  -- Enable writing to registers.
1017
1018            -- Write the incremented address to Rm
1019            RegAxInSel             <= to_integer(unsigned(nm_format_m));
1020            RegAxStore <= '1'; -- Enable writing to address register in the writeback state.
1021
1022            -- DMAU signals for post-increment indirect register addressing
1023            BaseSel      <= BaseSel_REG;
1024            IndexSel     <= IndexSel_NONE;
1025            OffScalarSel <= IR(1 downto 0);       -- bit-decode offset scalar select
1026            IncDecSel    <= IncDecSel_POST_INC;
1027
1028
1029        -- MOV.{B,W} RO, @(disp,Rn)
1030        -- nd4 format
1031        -- Note that the displacement depends on the mode of the address, so in
1032        -- byte mode, the displacement represents bytes, in word mode it represents
1033        -- words, etc. This is done to maximize it's range.
1034        elsif std_match(IR, MOV_R0_AT_DISP_RN) then
1035
1036            LogWithTime(l,
1037              "sh2_control.vhd: Decoded MOV.{B,W} R0, @(0x" & to_hstring(nd4_format_d) &
1038              ", " & to_string(slv_to_uint(nd4_format_n)) & ")", LogFile);
1039
1040            -- Instruction writes a byte to data memory.
1041            MemEnable   <= '1';
1042            ReadWrite   <= ReadWrite_WRITE;
1043
1044            MemMode     <= ByteMode when IR(8) = '0' else     -- bit-decode byte/word mode
1045                                   WordMode;
1046
1047            -- Output RegB (R0) to memory data bus
1048            MemOutSel <= MemOut_RegB;
1049
1050            -- Output R0 to RegB
1051            RegBSel <= 0;
1052
1053            -- Output Rn to RegA1. The DMAU will use this to calculate the address
1054            -- to write to.
1055            RegA1Sel <= to_integer(unsigned(nd4_format_n));
```

```
1056
1057            -- DMAU signals for Indirect register addressing with displacement
1058            BaseSel       <=  BaseSel_REG;
1059            IndexSel      <=  IndexSel_OFF4;
1060            OffScalarSel  <=  OffScalarSel_ONE when IR(8) = '0' else      -- bit-decode byte/word
1061                              OffScalarSel_TWO;
1062            IncDecSel     <=  IncDecSel_NONE;
1063            DMAUOff4      <=  nd4_format_d;
1064
1065
1066         -- MOV.L Rm, @(disp, Rn)
1067         -- nmd format
1068         elsif std_match(IR, MOV_L_RM_AT_DISP_RN) then
1069
1070            LogWithTime(l,
1071              "sh2_control.vhd: Decoded MOV.L R" & to_string(slv_to_uint(nmd_format_m)) &
1072              ", @(0x" & to_hstring(nmd_format_d) & ", R" & to_string(slv_to_uint(nmd_format_n)) & ")", LogFile);
1073
1074            MemEnable <= '1';
1075            ReadWrite <= ReadWrite_WRITE;
1076            MemMode   <= LongwordMode;
1077
1078            -- Output Rm to RegB.
1079            RegBSel <= to_integer(unsigned(nmd_format_m));
1080
1081            -- Output Rn to RegA1. The DMAU will use this to calculate the address
1082            -- to write to.
1083            RegA1Sel <= to_integer(unsigned(nmd_format_n));
1084
1085            -- Output RegB (Rm) to memory data bus. This will be written to memory.
1086            MemOutSel <= MemOut_RegB;
1087
1088            -- DMAU signals for Indirect register addressing with displacement (longword mode)
1089            BaseSel      <= BaseSel_REG;
1090            IndexSel     <= IndexSel_OFF4;
1091            OffScalarSel <= OffScalarSel_FOUR;
1092            IncDecSel    <= IncDecSel_NONE;
1093            DMAUOff4     <= nmd_format_d;
1094
1095
1096        -- MOV.{B,W} @(disp, Rm), R0
1097        -- md format
1098        -- Note that these instructions are very similar to MOV @(disp, PC), Rn
1099        elsif std_match(IR, MOV_AT_DISP_RM_R0) then
1100
1101            LogWithTime(l,
1102              "sh2_control.vhd: Decoded MOV.{B,W} @(0x" & to_hstring(md_format_d) &
1103              ", R" & to_string(slv_to_uint(md_format_m)) & "), R0", LogFile);
1104
1105            -- Writing sign-extended byte from data bus to R0.
1106            RegInSel             <= 0;             -- Select R0 to write to.
1107            RegDataInSel         <= RegDataIn_DB;  -- Write DataBus to reg.
1108            RegEnableIn <= '1';           -- Enable Reg writing for this instruction.
1109
1110            -- Output @Rm from RegA2
1111            RegA2Sel <= to_integer(unsigned(md_format_m));
1112
1113            MemEnable  <=  '1';            -- Instr uses memory.
1114            ReadWrite  <=  ReadWrite_READ; -- Reads.
1115            MemMode    <=  ByteMode when IR(8) = '0' else       -- bit-decode word mode
1116                                   WordMode;
1117            MemSel     <=  MemSel_RAM;     -- Reads from RAM
1118
1119            -- DMAU signals for Indirect register addressing with displacement (byte mode)
1120            BaseSel      <= BaseSel_REG;
1121            IndexSel     <= IndexSel_OFF4;
```

```
1122             OffScalarSel <= OffScalarSel_ONE when IR(8) = '0' else          -- bit-decode offset scale
1123                             OffScalarSel_TWO;
1124             IncDecSel    <= IncDecSel_NONE;
1125             DMAUOff4     <= md_format_d;
1126 
1127 
1128         -- MOV.L @(disp, Rm), Rn
1129         -- nmd
1130         elsif std_match(IR, MOV_L_AT_DISP_RM_RN) then
1131 
1132             LogWithTime(l,
1133               "sh2_control.vhd: Decoded MOV.L @(0x" & to_hstring(nmd_format_d) &
1134               ", R" & to_string(slv_to_uint(nmd_format_m)) & "), R" & to_string(slv_to_uint(nmd_format_n))
1135               , LogFile);
1136 
1137             -- Writing longword from data bus to Rn.
1138             RegInSel             <= to_integer(unsigned(nmd_format_n));   -- Select Rn to write to.
1139             RegDataInSel         <= RegDataIn_DB;                         -- Write DataBus to reg.
1140             RegEnableIn <= '1';                                  -- Enable Reg writing for this instruction.
1141 
1142              -- Output @Rm from RegA2
1143              RegA2Sel <= to_integer(unsigned(nmd_format_m));
1144 
1145             MemEnable  <=  '1';               -- Instr uses memory.
1146             ReadWrite  <=  ReadWrite_READ;    -- Reads.
1147             MemMode    <=  LongwordMode;      -- Reads longword.
1148             MemSel     <=  MemSel_RAM;        -- Reads from RAM
1149 
1150 
1151             -- DMAU signals for Indirect register addressing with displacement (longword mode)
1152             BaseSel      <= BaseSel_REG;
1153             IndexSel     <= IndexSel_OFF4;
1154             OffScalarSel <= OffScalarSel_FOUR;
1155             IncDecSel    <= IncDecSel_NONE;
1156             DMAUOff4     <= nmd_format_d;
1157 
1158 
1159         -- MOV.{B,W,L} Rm, @(R0, Rn)
1160         -- nm format
1161         elsif std_match(IR, MOV_RM_AT_R0_RN) then
1162 
1163             LogWithTime(l,
1164               "sh2_control.vhd: Decoded MOV.X R" & to_string(slv_to_uint(nm_format_m)) &
1165               ", @(R0, R" & to_string(slv_to_uint(nm_format_n)) & ")", LogFile);
1166 
1167             -- Instr writes a byte to memory.
1168             MemEnable <= '1';
1169             ReadWrite <= ReadWrite_WRITE;
1170             MemMode   <= IR(1 downto 0);      -- bit-decode memory mode
1171 
1172             -- Output Rm to RegB.
1173             RegBSel <= to_integer(unsigned(nm_format_m));
1174 
1175             -- Output Rn to RegA1. The DMAU will use this to calculate the address
1176             -- to write to.
1177             RegA1Sel <= to_integer(unsigned(nm_format_n));
1178 
1179             -- Output R0 to RegA2.
1180             RegA2Sel <= 0;
1181 
1182             -- Output RegB (Rm) to memory data bus. This will be written to memory.
1183             MemOutSel <= MemOut_RegB;
1184 
1185             -- DMAU Signals for Indirect Register Addressing
1186             BaseSel       <= BaseSel_REG;
1187             IndexSel      <= IndexSel_R0;
```

```
1188             OffScalarSel  <= OffScalarSel_ONE;
1189             IncDecSel     <= IncDecSel_NONE;
1190
1191
1192         -- MOV.{B,W,L} @(R0, Rm), Rn
1193         -- nm format
1194         elsif std_match(IR, MOV_AT_R0_RM_RN) then
1195
1196             LogWithTime(l,
1197               "sh2_control.vhd: Decoded MOV.X @(R0, R" & to_string(slv_to_uint(nm_format_m)) &
1198               "), R" & to_string(slv_to_uint(nm_format_n)), LogFile);
1199
1200             -- Writing sign-extended byte from data bus to Rn.
1201             RegInSel             <= slv_to_uint(nm_format_n);     -- Select Rn to write to.
1202             RegDataInSel         <= RegDataIn_DB;                 -- Write DataBus to reg.
1203             RegEnableIn <= '1';                          -- Enable Reg writing for this instruction.
1204
1205             -- Output @Rm from RegA2
1206             RegA2Sel <= slv_to_uint(nm_format_m);
1207
1208             -- Output @R0 from RegA1
1209             RegA1Sel <= 0;
1210
1211             MemEnable  <=  '1';             -- Instr uses memory.
1212             ReadWrite  <=  ReadWrite_READ;  -- Reads.
1213             MemMode    <=  IR(1 downto 0);  -- bit decode memory mode
1214             MemSel     <=  MemSel_RAM;      -- Reads from RAM
1215
1216             -- DMAU Signals for Indirect indexed Register Addressing
1217             BaseSel       <= BaseSel_REG;
1218             IndexSel      <= IndexSel_R0;
1219             OffScalarSel  <= OffScalarSel_ONE;
1220             IncDecSel     <= IncDecSel_NONE;
1221
1222
1223         -- MOV.{B,W,L} R0, @(disp, GBR)
1224         -- d format
1225         elsif std_match(IR, MOV_R0_AT_DISP_GBR) then
1226
1227             LogWithTime(l,
1228               "sh2_control.vhd: Decoded MOV.X R0, @(0x" & to_hstring(d_format_d) &
1229               ", GBR)", LogFile);
1230
1231             -- Writing to memory
1232             MemEnable <= '1';
1233             ReadWrite <= ReadWrite_WRITE;
1234             MemMode   <= IR(9 downto 8);      -- bit-decode memory mode
1235
1236             -- Output R0 to RegB.
1237             RegBSel <= 0;
1238
1239             -- Output RegB (Rm) to memory data bus. This will be written to memory.
1240             MemOutSel <= MemOut_RegB;
1241
1242             -- DMAU signals for Indirect GBR addressing with displacement
1243             BaseSel       <=  BaseSel_GBR;
1244             IndexSel      <=  IndexSel_OFF8;
1245             OffScalarSel  <=  IR(9 downto 8);     -- bit decode offset scalar select
1246             IncDecSel     <=  IncDecSel_NONE;
1247             DMAUOff8      <=  d_format_d;
1248
1249         -- MOVA @(disp, PC), R0
1250         -- d format
1251         -- disp*4 + PC -> R0
1252         -- Note: due to bit decoding, this must come before MOV_AT_DISP_GBR_R0
1253         elsif std_match(IR, MOVA_AT_DISP_PC_R0) then
```

```
1254
1255            LogWithTime(l,
1256                "sh2_control.vhd: Decoded MOVA @(" & to_hstring(d_format_d) &
1257                ", PC), R0", LogFile);
1258
1259            -- Note that this instruction moves the address, disp*4 + PC
1260            -- (calculated by the DMAU) into R0. It does NOT move the data at
1261            -- this address.
1262
1263            RegAxInSel            <=  0;   -- Write address to R0
1264            RegAxStore <= '1';  -- Enable writing to address register in writeback state.
1265
1266            -- DMAU signals for PC Relative addressing with displacement (longword mode)
1267            BaseSel      <= BaseSel_PC;
1268            IndexSel     <= IndexSel_OFF8;
1269            OffScalarSel <= OffScalarSel_FOUR;
1270            IncDecSel    <= IncDecSel_NONE;
1271            DMAUOff8     <= nd8_format_d;
1272
1273        -- MOV.{B,W,L} @(disp, GBR), R0
1274        -- d format
1275        -- Note: due to bit decoding, this must come after MOVA_AT_DISP_PC_R0
1276        elsif std_match(IR, MOV_AT_DISP_GBR_R0) then
1277
1278            LogWithTime(l,
1279                "sh2_control.vhd: Decoded MOV.X @(0x" & to_hstring(d_format_d) &
1280                ", GBR), R0", LogFile);
1281
1282           RegInSel             <= 0;               -- Write to R0
1283           RegDataInSel         <= RegDataIn_DB;    -- Write Data bus to R0
1284           RegEnableIn <= '1';             -- Enable register writing for this instruction.
1285
1286           MemEnable <= '1';
1287           ReadWrite <= ReadWrite_READ;
1288           MemMode   <= IR(9 downto 8); -- bit decode memory mode
1289
1290            -- DMAU signals for Indirect GBR addressing with displacement (byte mode)
1291            BaseSel      <=  BaseSel_GBR;
1292            IndexSel     <=  IndexSel_OFF8;
1293            OffScalarSel <=  IR(9 downto 8);          -- bit decode offset scalar select
1294            IncDecSel    <=  IncDecSel_NONE;
1295            DMAUOff8     <=  d_format_d;
1296
1297        -- MOVT Rn
1298        -- n format.
1299        elsif std_match(IR, MOVT_RN) then
1300
1301            LogWithTime(l,
1302                "sh2_control.vhd: Decoded MOVT R" & to_string(slv_to_uint(n_format_n)),
1303                LogFile);
1304
1305            RegInSel             <= to_integer(unsigned(n_format_n));
1306            RegDataInSel         <= RegDataIn_SR_TBit;
1307            RegEnableIn <= '1';
1308
1309
1310        -- SWAP.B Rm, Rn
1311        -- nm format
1312        -- Rm -> Swap upper and lower 2 bytes -> Rn
1313        elsif std_match(IR, SWAP_RM_RN) then
1314
1315            LogWithTime(l,
1316                "sh2_control.vhd: Decoded SWAP.X R" & to_string(slv_to_uint(nm_format_m))
1317                & ", R" & to_string(nm_format_n), LogFile);
1318
1319            RegASel      <= slv_to_uint(nm_format_m);
```

```
1320             RegInSel     <= slv_to_uint(nm_format_n);
1321
1322             -- Bit decode if whether byte or word mode
1323             RegDataInSel <= RegDataIn_RegA_SwapB when IR(0) = '0' else
1324                             RegDataIn_RegA_SwapW;
1325
1326             RegEnableIn <= '1';
1327
1328
1329         -- XTRCT Rm, Rn
1330         -- nm format
1331         -- Center 32 bits of Rm and Rn -> Rn
1332         elsif std_match(IR, XTRCT_RM_RN) then
1333
1334             LogWithTime(l,
1335                 "sh2_control.vhd: Decoded XTRCT R" & to_string(slv_to_uint(nm_format_m))
1336                 & ", R" & to_string(nm_format_n), LogFile);
1337
1338             RegASel <= slv_to_uint(nm_format_n);
1339             RegBSel <= slv_to_uint(nm_format_m);
1340
1341             RegInSel <= slv_to_uint(nm_format_n); -- Write to Rn
1342
1343             RegDataInSel <= RegDataIn_REGB_REGA_CENTER;
1344
1345             RegEnableIn <= '1';
1346
1347
1348
1349         -- Branch Instructions ------------------------------------------------
1350
1351         -- BF <label> (where label is disp*2 + PC)
1352         -- d format
1353          elsif std_match(IR, BF) then
1354
1355              LogWithTime(l,
1356                  "sh2_control.vhd: Decoded BF (label=" & to_hstring(d_format_d) &
1357                  "*2 + PC)", LogFile);
1358
1359              -- If T=0, disp*2 + PC -> PC; if T=1, nop (where label is disp*2 + PC)
1360
1361              -- Should not check T flag here because it may not have been updated
1362              -- yet - need to delay until the ID stage.
1363              PCAddrMode  <= PCAddrMode_RELATIVE_8;
1364              ConditionalBranch <= '1';
1365              Condition <= '0';
1366              PMAUOff8    <= d_format_d;
1367
1368
1369          -- BF/S <label> (where label is disp*2 + PC)
1370          -- d format
1371          elsif std_match(IR, BF_S) then
1372
1373              LogWithTime(l,
1374                  "sh2_control.vhd: Decoded BF/S (label=" & to_hstring(d_format_d) &
1375                  "*2 + PC)", LogFile);
1376
1377              -- Should not check T flag here because it may not have been updated
1378              -- yet - need to delay until the ID stage.
1379              PCAddrMode   <= PCAddrMode_RELATIVE_8;
1380              ConditionalBranch <= '1';
1381              Condition    <= '0';
1382              DelayBranch  <= '1';
1383              PMAUOff8     <= d_format_d;
1384
1385
```

```
1386          -- BT <label> (where label is disp*2 + PC)
1387          -- d format
1388          elsif std_match(IR, BT) then
1389
1390              -- Branch true without delay slot.
1391
1392              LogWithTime(l,
1393                  "sh2_control.vhd: Decoded BT (label=" & to_hstring(d_format_d) &
1394                  "*2 + PC)", LogFile);
1395
1396              -- If T=1, disp*2 + PC -> PC; if T=0, nop (where label is disp*2 + PC)
1397
1398              -- Should not check T flag here because it may not have been updated
1399              -- yet - need to delay until the ID stage.
1400              PCAddrMode   <= PCAddrMode_RELATIVE_8;
1401              ConditionalBranch <= '1';
1402              Condition    <= '1';
1403              PMAUOff8     <= d_format_d;
1404
1405
1406          -- BT/S <label> (where label is disp*2 + PC)
1407          -- d format
1408          elsif std_match(IR, BT_S) then
1409
1410              LogWithTime(l,
1411                  "sh2_control.vhd: Decoded BT/S (label=" & to_hstring(d_format_d) &
1412                  "*2 + PC)", LogFile);
1413
1414
1415              -- Should not check T flag here because it may not have been updated
1416              -- yet - need to delay until the ID stage.
1417              PCAddrMode   <= PCAddrMode_RELATIVE_8;
1418              ConditionalBranch <= '1';
1419              Condition    <= '1';
1420              DelayBranch  <= '1';
1421              PMAUOff8     <= d_format_d;
1422
1423
1424          -- BRA <label> (where label is disp*2 + PC)
1425          -- d12 format
1426          elsif std_match(IR, BRA) then
1427
1428              LogWithTime(l,
1429                  "sh2_control.vhd: Decoded BRA (label=" & to_hstring(d12_format_d) &
1430                  "*2 + PC)", LogFile);
1431
1432             DelayBranch <= '1';
1433             PCWriteCtrl                    <= PCWriteCtrl_WRITE_CALC;
1434
1435             PCAddrMode <= PCAddrMode_RELATIVE_12;
1436             PMAUOff12              <= d12_format_d;
1437
1438
1439          -- BRAF Rm
1440          -- m format
1441          elsif std_match(IR, BRAF) then
1442
1443              -- Delayed branch, Rm + PC -> PC
1444              -- Note that the PMAU's register input is always RegB.
1445
1446               RegBSel <= slv_to_uint(m_format_m);
1447
1448               LogWithTime(l,
1449                   "sh2_control.vhd: Decoded BRAF R" & to_string(slv_to_uint(m_format_m)), LogFile);
1450
1451               DelayBranch <= '1';
```

```
1452                PCWriteCtrl                    <= PCWriteCtrl_WRITE_CALC;
1453
1454                PCAddrMode <= PCAddrMode_REG_DIRECT_RELATIVE;
1455
1456
1457            -- BSR <label> (where label is disp*2)
1458            -- d12 format
1459            elsif std_match(IR, BSR) then
1460
1461                LogWithTime(l,
1462                    "sh2_control.vhd: Decoded BSR (label=" & to_hstring(d12_format_d) &
1463                    "*2 + PC)", LogFile);
1464
1465                DelayBranch <= '1';
1466                PCWriteCtrl                    <= PCWriteCtrl_WRITE_CALC;
1467
1468                PCAddrMode <= PCAddrMode_RELATIVE_12;
1469                PMAUOff12  <= d12_format_d;
1470
1471                PRWriteEn <= '1';
1472
1473                -- Control signals to write PC to PR.
1474                SysRegSrc              <= SysRegSrc_PC;
1475                SysRegCtrl <= SysRegCtrl_LOAD;
1476
1477
1478            -- BSRF Rm
1479            -- m format
1480            --
1481            -- Branch to sub-routine far.
1482            -- PC -> PR, Rm + PC -> PC
1483            elsif std_match(IR, BSRF) then
1484
1485                LogWithTime(l,
1486                    "sh2_control.vhd: Decoded BSRF R" & to_string(slv_to_uint(m_format_m)), LogFile);
1487
1488               -- Basically BSR, but with a different target.
1489                DelayBranch <= '1';
1490                PCWriteCtrl                    <= PCWriteCtrl_WRITE_CALC;
1491
1492                PCAddrMode <= PCAddrMode_REG_DIRECT_RELATIVE;
1493
1494                PRWriteEn <= '1';
1495
1496                -- Control signals to write PC to PR.
1497                SysRegSrc              <= SysRegSrc_PC;
1498                SysRegCtrl <= SysRegCtrl_LOAD;
1499
1500
1501            -- JMP @Rm
1502            -- m format
1503            -- Delayed branch, Rm -> PC
1504            elsif std_match(IR, JMP) then
1505
1506                LogWithTime(l,
1507                    "sh2_control.vhd: Decoded JMP @R" & to_string(slv_to_uint(m_format_m)), LogFile);
1508
1509                -- PMAU Register input is RegB.
1510                RegBSel <= slv_to_uint(m_format_m);
1511
1512                DelayBranch <= '1';
1513                PCWriteCtrl                    <= PCWriteCtrl_WRITE_CALC;
1514                PCAddrMode         <= PCAddrMode_REG_DIRECT;
1515
1516
1517            -- JSR @Rm
```

```
1518          -- m format
1519          -- Delayed branch, PC -> PR, Rm -> PC
1520          elsif std_match(IR, JSR) then
1521
1522              LogWithTime(l,
1523                  "sh2_control.vhd: Decoded JSR @R" & to_string(slv_to_uint(m_format_m)), LogFile);
1524
1525              RegBSel <= slv_to_uint(m_format_m);
1526
1527              DelayBranch <= '1';
1528              PCWriteCtrl                    <= PCWriteCtrl_WRITE_CALC;
1529              PCAddrMode         <= PCAddrMode_REG_DIRECT;
1530
1531              PRWriteEn <= '1';
1532
1533              SysRegSrc <= SysRegSrc_PC;
1534              SysRegCtrl <= SysRegCtrl_LOAD;
1535
1536          elsif std_match(IR, RTS) then
1537
1538              LogWithTime(l,
1539                  "sh2_control.vhd: Decoded RTS", LogFile);
1540
1541              PCAddrMode         <= PCAddrMode_PR_DIRECT;
1542              DelayBranch <= '1';
1543              PCWriteCtrl                    <= PCWriteCtrl_WRITE_CALC;
1544
1545
1546         -- System Control Instructions ---------------------------------------
1547
1548         elsif std_match(IR, CLRT) then
1549
1550             LogWithTime(l, "sh2_control.vhd: Decoded CLRT", LogFile);
1551
1552             TFlagSel <= TFlagSel_CLEAR;     -- clear the T flag
1553
1554         elsif std_match(IR, CLRMAC) then
1555
1556             LogWithTime(l, "sh2_control.vhd: Decoded CLRMAC", LogFile);
1557
1558             SysRegCtrl <= SysRegCtrl_CLEAR;
1559             SysRegSel <= SysRegSel_MACL;
1560
1561         elsif std_match(IR, SETT) then
1562
1563             LogWithTime(l, "sh2_control.vhd: Decoded SETT", LogFile);
1564
1565             TFlagSel <= TFlagSel_SET;       -- set the T flag
1566
1567         elsif std_match(IR, STC_SYS_RN) then
1568
1569             -- STC {SR, GBR, VBR}, Rn
1570             -- Uses bit decoding to choose the system register to store
1571
1572             LogWithTime(l, "sh2_control.vhd: Decoded STC XXX, Rn", LogFile);
1573
1574             RegInSel <= to_integer(unsigned(n_format_n));
1575
1576             -- selects data source to store to a register through bit decoding
1577             SysRegSel <= "0" & IR(5 downto 4);
1578             RegDataInSel <= RegDataIn_SysReg;
1579             RegEnableIn <= '1';
1580
1581         elsif std_match(IR, STS_SYS_RN) then
1582
1583             -- STS {MACH, MACL, PR}, Rn
```

```
1584             -- Uses bit decoding to choose the system register to store
1585
1586             LogWithTime(l, "sh2_control.vhd: Decoded STS XXX, Rn", LogFile);
1587
1588             RegInSel <= to_integer(unsigned(n_format_n));
1589
1590             -- selects data source to store to a register through bit decoding
1591             SysRegSel <= "1" & IR(5 downto 4);
1592             RegDataInSel <= RegDataIn_SysReg;
1593             RegEnableIn <= '1';
1594
1595         elsif std_match(IR, STC_L_SYS_RN) then
1596
1597             -- STC.L {SR, GBR, VBR}, @-Rn
1598             -- Uses bit decoding to choose the system register to store
1599             LogWithTime(l, "sh2_control.vhd: Decoded STC.L XXX, @-Rn", LogFile);
1600
1601             -- Writes a byte to memory
1602             MemEnable <= '1';               -- Uses memory.
1603             ReadWrite <= ReadWrite_WRITE;   -- Writes.
1604             MemMode   <= LongwordMode;      -- bit decode memory mode
1605
1606             -- selects data source to store to a register through bit decoding
1607             SysRegSel <= "0" & IR(5 downto 4);
1608             MemOutSel <= MemOut_SysReg;
1609
1610             RegA1Sel               <= to_integer(unsigned(nm_format_n));  -- Output @(Rn) from RegA1 output.
1611             RegAxInSel             <= to_integer(unsigned(nm_format_n));  -- Store calculated address into Rn
1612             RegAxStore <= '1';                              -- Enable writes to address registers.
1613
1614             -- DMAU signals (for Pre-decrement indirect register addressing)
1615             BaseSel      <= BaseSel_REG;
1616             IndexSel     <= IndexSel_NONE;
1617             OffScalarSel <= OffScalarSel_FOUR;
1618             IncDecSel    <= IncDecSel_PRE_DEC;
1619
1620         elsif std_match(IR, STS_L_SYS_RN) then
1621
1622             -- STC.L {MACH, MACL, PR}, @-Rn
1623             -- Uses bit decoding to choose the system register to store
1624             LogWithTime(l, "sh2_control.vhd: Decoded STC.L XXX, @-Rn", LogFile);
1625
1626             -- Writes a byte to memory
1627             MemEnable <= '1';               -- Uses memory.
1628             ReadWrite <= ReadWrite_WRITE;   -- Writes.
1629             MemMode   <= LongwordMode;      -- bit decode memory mode
1630
1631             -- selects data source to store to a register through bit decoding
1632             SysRegSel <= "1" & IR(5 downto 4);
1633             MemOutSel <= MemOut_SysReg;
1634
1635             RegA1Sel               <= to_integer(unsigned(nm_format_n));  -- Output @(Rn) from RegA1 output.
1636             RegAxInSel             <= to_integer(unsigned(nm_format_n));  -- Store calculated address into Rn
1637             RegAxStore <= '1';                              -- Enable writes to address registers.
1638
1639             -- DMAU signals (for Pre-decrement indirect register addressing)
1640             BaseSel      <= BaseSel_REG;
1641             IndexSel     <= IndexSel_NONE;
1642             OffScalarSel <= OffScalarSel_FOUR;
1643             IncDecSel    <= IncDecSel_PRE_DEC;
1644
1645
1646         elsif std_match(IR, LDC_RM_SYS) then
1647
1648             -- LDC Rm, GBR must actually load into the GBR in the DMAU for later instructions
1649             -- to work. Must modify other system control register loads to load to their actual
```

```
1650             -- locations as well.
1651             if (std_match(IR, LDC_RM_GBR)) then
1652                 GBRWriteEn <= '1';
1653             end if;
1654
1655             -- LDC Rm, {SR, GBR, VBR}
1656             -- Uses bit decoding to choose the system register to load
1657
1658             LogWithTime(l, "sh2_control.vhd: Decoded LDC Rm, X", LogFile);
1659
1660             RegBSel <= to_integer(unsigned(m_format_m));
1661             SysRegCtrl <= SysRegCtrl_LOAD;
1662             SysRegSel <= "0" & IR(5 downto 4);      -- bit decode register to select
1663             SysRegSrc <= SysRegSrc_RegB;
1664
1665         elsif std_match(IR, LDC_L_RM_SYS) then
1666             -- LDC.L @Rm+, {SR, GBR, VBR}
1667             -- Uses bit decoding to choose the system register to load
1668
1669             if (std_match(IR, LDC_L_AT_RM_PLUS_GBR)) then
1670                 GBRWriteEn <= '1';
1671             end if;
1672
1673             LogWithTime(l, "sh2_control.vhd: Decoded LDC.L @Rm+, X", LogFile);
1674
1675             -- Reads a longword from memory
1676             MemEnable <= '1';             -- Uses memory.
1677             ReadWrite <= ReadWrite_READ;  -- Reads.
1678             MemMode   <= LongwordMode;    -- bit decode memory mode
1679
1680             -- Load into a system register
1681             SysRegCtrl <= SysRegCtrl_LOAD;
1682             SysRegSel <= "0" & IR(5 downto 4);    -- bit decode which system register to write to
1683             SysRegSrc <= SysRegSrc_DB;      -- load new register value from memory
1684
1685             -- Read from @Rm, and save with post-incremented value
1686             RegA2Sel   <= to_integer(unsigned(m_format_m));
1687             RegAxInSel <= to_integer(unsigned(m_format_m));
1688             RegAxStore <= '1';
1689
1690             -- DMAU signals (for post-increment indirect register addressing)
1691             BaseSel      <= BaseSel_REG;
1692             IndexSel     <= IndexSel_NONE;
1693             OffScalarSel <= OffScalarSel_FOUR;
1694             IncDecSel    <= IncDecSel_POST_INC;
1695
1696         elsif std_match(IR, LDS_RM_SYS) then
1697             -- LDS Rm, {MACH, MACL, PR}
1698             -- Uses bit decoding to choose the system register to load
1699
1700             -- Ensure that PR does actually get written to
1701             if (std_match(IR, LDS_RM_PR)) then
1702                 PRWriteEn <= '1';
1703             end if;
1704
1705             LogWithTime(l, "sh2_control.vhd: Decoded LDS Rm, X", LogFile);
1706
1707             RegBSel <= to_integer(unsigned(m_format_m));
1708             SysRegCtrl <= SysRegCtrl_LOAD;
1709             SysRegSel <= "1" & IR(5 downto 4);  -- bit decode register to select
1710             SysRegSrc <= SysRegSrc_RegB;
1711
1712         elsif std_match(IR, LDS_L_RM_SYS) then
1713             -- LDS.L @Rm+, {MACH, MACL, PR}
1714             -- Uses bit decoding to choose the system register to load
1715
```

```
1716             if (std_match(IR, LDS_L_AT_RM_PLUS_PR)) then
1717                 PRWriteEn <= '1';
1718             end if;
1719
1720             LogWithTime(l, "sh2_control.vhd: Decoded LDS.L @Rm+, X", LogFile);
1721
1722             -- Reads a longword from memory
1723             MemEnable <= '1';             -- Uses memory.
1724             ReadWrite <= ReadWrite_READ;  -- Reads.
1725             MemMode   <= LongwordMode;    -- bit decode memory mode
1726
1727             -- Load into a system register
1728             SysRegCtrl <= SysRegCtrl_LOAD;
1729             SysRegSel <= "1" & IR(5 downto 4);    -- bit decode which system register to write to
1730             SysRegSrc <= SysRegSrc_DB;      -- load new register value from memory
1731
1732             -- Read from @Rm, and save with post-incremented value
1733             RegA2Sel   <= to_integer(unsigned(m_format_m));
1734             RegAxInSel <= to_integer(unsigned(m_format_m));
1735             RegAxStore <= '1';
1736
1737             -- DMAU signals (for post-increment indirect register addressing)
1738             BaseSel      <= BaseSel_REG;
1739             IndexSel     <= IndexSel_NONE;
1740             OffScalarSel <= OffScalarSel_FOUR;
1741             IncDecSel    <= IncDecSel_POST_INC;
1742
1743         elsif std_match(IR, NOP) then
1744
1745             LogWithTime(l, "sh2_control.vhd: Decoded NOP", LogFile);
1746
1747         elsif not is_x(IR) then
1748             report "Unrecognized instruction: " & to_hstring(IR);
1749         end if;
1750
1751     end process decode_proc;
1752
1753     -- If previous instruction was a branch, don't enable next decoded instruction
1754     process(clock, reset)
1755     begin
1756         if reset = '0' then
1757             pipeline_in_en <= '1';
1758         elsif rising_edge(clock) then
1759             pipeline_in_en <= not ShouldBranch;
1760         end if;
1761     end process ;
1762
1763     pipeline_stages: for i in 0 to 3 generate
1764         process (clock, reset)
1765         begin
1766             if reset = '0' then
1767                 pipeline(i) <= DEFAULT_CTRL;
1768                 pipeline_en(i) <= '0';
1769             elsif rising_edge(clock) then
1770                 -- Advance stages of the pipeline that are enabled and not bubbled.
1771                 if BubbleIF = '1' then
1772                     if i < STAGE_MA then
1773                         -- hold previous stages constant
1774                         pipeline(i) <= pipeline(i);
1775                         pipeline_en(i) <= pipeline_en(i);
1776                     elsif i = STAGE_MA then
1777                         -- Insert a bubble at MA
1778                         pipeline(i) <= DEFAULT_CTRL;
1779                         pipeline_en(i) <= '0';
1780                     else
1781                         -- The rest of the stages can move on
```

```
1782                        pipeline(i) <= pipeline(i - 1);
1783                        pipeline_en(i) <= pipeline_en(i - 1);
1784                    end if;
1785                else
1786                    if i = 0 then
1787                        -- First item in pipeline comes from decoded signals
1788                        pipeline(i) <= decoded_signals;
1789                        pipeline_en(i) <= '0' when pipeline_in_en = '0' or ShouldBranch = '1' else '1';
1790                    elsif i = 1 then
1791                        pipeline(i) <= pipeline(i - 1);
1792                        if ShouldBranch = '1' then
1793                            -- TODO: delayed branches not working
1794                            pipeline_en(i) <= pipeline(STAGE_X).PMAUCtrl.DelayBranch;
1795                        else
1796                            pipeline_en(i) <= pipeline_en(i - 1);
1797                        end if;
1798                    else
1799                        -- Next, simply pipe each set of signals into a new DFF
1800                        pipeline(i) <= pipeline(i - 1);
1801                        pipeline_en(i) <= pipeline_en(i - 1);
1802                    end if;
1803                end if;
1804            end if;
1805        end process;
1806    end generate;
1807
1808    -- Register updates done on clock edges
1809    state_proc: process (clock, reset)
1810    begin
1811        if reset = '0' then
1812            IR <= NOP;
1813        elsif rising_edge(clock) then
1814            if BubbleIF = '0' then
1815                -- Latch in instruction from memory if no bubble. Should not
1816                -- occur when the IF and MA stages have contention on the
1817                -- memory bus (since MA takes precendence).
1818                IR <= MemDataIn(15 downto 0);
1819            end if;
1820        end if;
1821    end process state_proc;
1822
1823 end dataflow;
1824
```

```
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  --     07 Jun 25  Zack Huang        Move all control signals into record types for organization
35  --
36  ----------------------------------------------------------------------------
37
38
39  --
40  --  SH2_CPU
41  --
42  --  This is the complete entity declaration for the SH-2 CPU.  It is used to
43  --  test the complete design.
44  --
45  --  Inputs:
46  --    Reset  - active low reset signal
47  --    NMI    - active falling edge non-maskable interrupt
48  --    INT    - active low maskable interrupt
49  --    clock  - the system clock
50  --
51  --  Outputs:
52  --    AB     - memory address bus (32 bits)
53  --    RE0    - first byte read signal, active low
54  --    RE1    - second byte read signal, active low
55  --    RE2    - third byte read signal, active low
56  --    RE3    - fourth byte read signal, active low
57  --    WE0    - first byte write signal, active low
58  --    WE1    - second byte write signal, active low
59  --    WE2    - third byte write signal, active low
60  --    WE3    - fourth byte write signal, active low
61  --
62  --  Inputs/Outputs:
63  --    DB     - memory data bus (32 bits)
64  --
65
66  library ieee;
```

```
67  use ieee.std_logic_1164.all;
68  use ieee.numeric_std.all;
69  use std.textio.all;
70
71  use work.SH2Constants.all;
72  use work.SH2ControlSignals.all;
73  use work.SH2PmauConstants.all;
74  use work.SH2DmauConstants.all;
75  use work.MemoryInterfaceConstants.all;
76
77
78  entity  SH2CPU  is
79
80      port (
81          Reset   :  in     std_logic;                      -- reset signal (active low)
82          NMI     :  in     std_logic;                      -- non-maskable interrupt signal (falling edge)
83          INT     :  in     std_logic;                      -- maskable interrupt signal (active low)
84          clock   :  in     std_logic;                      -- system clock
85          AB      :  out    std_logic_vector(31 downto 0);  -- memory address bus
86          MemSel  :  out    std_logic;                      -- whether to access data memory (0) or program memory (1)
87          RE0     :  out    std_logic;                      -- first byte active low read enable
88          RE1     :  out    std_logic;                      -- second byte active low read enable
89          RE2     :  out    std_logic;                      -- third byte active low read enable
90          RE3     :  out    std_logic;                      -- fourth byte active low read enable
91          WE0     :  out    std_logic;                      -- first byte active low write enable
92          WE1     :  out    std_logic;                      -- second byte active low write enable
93          WE2     :  out    std_logic;                      -- third byte active low write enable
94          WE3     :  out    std_logic;                      -- fourth byte active low write enable
95          DB      :  inout  std_logic_vector(31 downto 0)   -- memory data bus
96      );
97
98  end  SH2CPU;
99
100 architecture structural of sh2cpu is
101
102     -- Sign-extends a standard logic vector to the SH2 wordsize
103     pure function SignExtend(slv : std_logic_vector) return std_logic_vector is
104     begin
105       return std_logic_vector(resize(signed(slv), SH2_WORDSIZE));
106     end function;
107
108     -- Zero-extends a standard logic vector to the SH2 wordsize
109     pure function ZeroExtend(slv : std_logic_vector) return std_logic_vector is
110     begin
111       return std_logic_vector(resize(unsigned(slv), SH2_WORDSIZE));
112     end function;
113
114     -- Returns the low byte of a standard logic vector
115     pure function LowByte(slv : std_logic_vector) return std_logic_vector is
116     begin
117       assert slv'length >= 8
118       report "slv must be >= 8 bits."
119       severity ERROR;
120
121       return slv(7 downto 0);
122     end function;
123
124     -- Returns the low word of a standard logic vector
125     pure function LowWord(slv : std_logic_vector) return std_logic_vector is
126     begin
127       assert slv'length >= 16
128       report "slv must be >= 16 bits"
129       severity ERROR;
130
131       return slv(15 downto 0);
132     end function;
```

```
133
134
135     -- Register array inputs
136     signal RegDataIn  : std_logic_vector(31 downto 0);      -- data to write to a register
137     signal RegAxIn    : std_logic_vector(31 downto 0);      -- data to write to an address register
138
139     -- register array outputs
140     signal RegA       : std_logic_vector(31 downto 0);      -- register bus A
141     signal RegB       : std_logic_vector(31 downto 0);      -- register bus B
142     signal RegA1      : std_logic_vector(31 downto 0);      -- address register bus 1
143     signal RegA2      : std_logic_vector(31 downto 0);      -- address register bus 2
144
145     -- ALU inputs
146     signal OperandA : std_logic_vector(31 downto 0);        -- first operand
147     signal OperandB : std_logic_vector(31 downto 0);        -- second operand
148
149     -- ALU outputs
150     signal Result   : std_logic_vector(31 downto 0);        -- ALU result
151     signal Cout     : std_logic;                            -- carry out
152     signal Overflow : std_logic;                            -- signed overflow
153     signal Zero     : std_logic;                            -- result is zero
154     signal Sign     : std_logic;                            -- sign of result
155
156     -- DMAU inputs
157     signal RegSrc       : std_logic_vector(31 downto 0);    -- input register
158     signal R0Src        : std_logic_vector(31 downto 0);    -- R0 register input
159     signal PCSrc        : std_logic_vector(31 downto 0);    -- program counter source
160     signal GBRIn        : std_logic_vector(31 downto 0);    -- GBR input
161
162     -- DMAU outputs
163     signal DataAddress  : std_logic_vector(31 downto 0);    -- output address
164     signal AddrSrcOut   : std_logic_vector(31 downto 0);    -- incremented/decremented address (store back to register)
165     signal GBROut       : std_logic_vector(31 downto 0);    -- GBR output
166                                                             --
167     -- PMAU inputs
168     signal RegIn       : std_logic_vector(31 downto 0);     -- register source input
169     signal PRIn        : std_logic_vector(31 downto 0);     -- PR register input (for writing to PR)
170
171     -- PMAU outputs
172     signal PCCalcOut : std_logic_vector(31 downto 0);       -- calculated PC address output
173     signal PCRegOut  : std_logic_vector(31 downto 0);       -- currenet value of the PC register
174     signal PROut     : std_logic_vector(31 downto 0);       -- PR (procedure register) output
175
176     -- Memory interface inputs
177     signal MemAddress  : std_logic_vector(31 downto 0);     -- memory address bus (MUST BE ALIGNED)
178     signal MemDataOut  : std_logic_vector(31 downto 0);     -- the data to output to memory
179
180     -- Memory interface outputs
181     signal ReadMask    : std_logic_vector(3 downto 0);      -- read enable mask (active low)
182     signal WriteMask   : std_logic_vector(3 downto 0);      -- write enable mask (active low)
183     signal MemDataIn   : std_logic_vector(31 downto 0);     -- the data read in from memory
184
185     -- CPU internal control signals
186     -- signal DelayedBranchTaken   : std_logic;                    -- Whether the delayed branch is taken or not.
187
188     -- CPU system/control registers
189     signal SR               : std_logic_vector(31 downto 0);    -- Status register.
190     signal VBR              : std_logic_vector(31 downto 0);    -- Vector Base Register.
191     signal MACL             : std_logic_vector(31 downto 0);    -- Multiply and Accumulate Low.
192     signal MACH             : std_logic_vector(31 downto 0);    -- Multiply and Accumulate High.
193
194     -- Aliases for status register bits.
195     alias MBit  : std_logic is SR(9);   -- The M and Q bits are used by DIVOU/S and DIV1 instructions.
196     alias QBit  : std_logic is SR(8);   -- ...
197     alias I3Bit : std_logic is SR(7);   -- Interrupt mask bits.
198     alias I2Bit : std_logic is SR(6);   -- ...
```

```
199     alias I1Bit : std_logic is SR(5);   -- ...
200     alias I0BIt : std_logic is SR(4);   -- ...
201     alias SBit  : std_logic is SR(1);   -- Used by MAC instructions.
202     alias TBit  : std_logic is SR(0);   -- True flag.
203
204     -- Intermediate terms
205     signal ExtendedReg      : std_logic_vector(31 downto 0);    -- extended register value (for EXT* instructions)
206     signal NextSysReg       : std_logic_vector(31 downto 0);    -- data to input into a system register
207     signal ImmediateExt     : std_logic_vector(31 downto 0);    -- sign-extended immediate
208     signal TCmp             : std_logic;                        -- The value of T generated from a compare
209     signal StrCmp           : std_logic;                        -- used to compare the bytes of RegA and RegB
210     signal SysReg           : std_logic_vector(31 downto 0);    -- Value of selected system/control register
211     signal PCNext           : std_logic_vector(31 downto 0);    -- The current PC incremented by two.
212     signal PrevPCReg        : std_logic_vector(31 downto 0);    -- the prevous value fo PC
213     signal PCUsed           : std_logic_vector(31 downto 0);    -- The PC that will be fetched from program memory.
214     signal TNext            : std_logic;                        -- next value for T bit in SR
215
216     -- RegA with the upper and lower halves of the low two bytes swapped (for the SWAP.B instruction).
217     signal RegASwapB : std_logic_vector(31 downto 0);
218
219     -- RegA with the high and low words swapped (for the SWAP.W instruction).
220     signal RegASwapW : std_logic_vector(31 downto 0);
221
222     -- The center 32-bits of RegB and RegA (ie, low word of RegB and high word of RegA).
223     signal RegB_RegA_Center : std_logic_vector(31 downto 0);
224
225     signal MemCtrl             : mem_ctrl_t;                    -- memory interface control signals
226     signal ALUCtrl             : alu_ctrl_t;                    -- ALU control signals
227     signal RegCtrl             : reg_ctrl_t;                    -- register array control signals
228     signal DMAUCtrl            : dmau_ctrl_t;                   -- DMAU control signals
229     signal PMAUCtrl            : pmau_ctrl_t;                   -- PMAU control signals
230     signal SysCtrl             : sys_ctrl_t;                    -- system control signals
231
232 begin
233
234     -- Output read enable signals when clock is low
235     RE0 <= ReadMask(0) when (not clock) else '1';
236     RE1 <= ReadMask(1) when (not clock) else '1';
237     RE2 <= ReadMask(2) when (not clock) else '1';
238     RE3 <= ReadMask(3) when (not clock) else '1';
239
240     -- Output write enable signals when clock is low
241     WE0 <= WriteMask(0) when (not clock) else '1';
242     WE1 <= WriteMask(1) when (not clock) else '1';
243     WE2 <= WriteMask(2) when (not clock) else '1';
244     WE3 <= WriteMask(3) when (not clock) else '1';
245
246     StorePCReg : process(clock)
247     begin
248         if rising_edge(clock) then
249             -- Store previous PC on rising clock
250             PrevPCReg <= PCRegOut;
251         end if;
252     end process;
253
254     -- The "next" PC is the current value of the PC register (NOT PCCalcOut) + 2.
255     PCNext <= std_logic_vector(unsigned(PrevPCReg) + to_unsigned(2, 32));
256
257     -- Decide which value of PC should be used
258     -- PCUsed <= PCNext when (DelayedBranchTaken = '1') else PCCalcOut;
259     PCUsed <= PCRegOut;
260
261     -- Decide which memory address to output
262     with MemCtrl.AddrSel select
263         MemAddress <=  PCUsed           when MemAddrSel_PMAU,
264                        DataAddress      when MemAddrSel_DMAU,
```

```
265                     (others => 'Z')  when others;
266
267     AB <= MemAddress;   -- Output memory address to the address bus
268
269     MemSel <= MemCtrl.Sel;
270
271     -- What to output to the data bus (to be written to an address).
272     with MemCtrl.OutSel select
273         MemDataOut <=   RegA            when MemOut_RegA,
274                         RegB            when MemOut_RegB,
275                         SysReg          when MemOut_SysReg,
276                         (others => 'Z') when others;
277
278     -- Compute the zero/sign-extended immediate from an instruction
279     ImmediateExt(7 downto 0) <= AluCtrl.Immediate;
280     with AluCtrl.ImmediateMode select
281         ImmediateExt(31 downto 8) <= (others => AluCtrl.Immediate(7)) when ImmediateMode_SIGN,
282                                      (others => '0')          when ImmediateMode_ZERO,
283                                      (others => 'X')          when others;
284
285     -- RegA with the high and low bytes swapped.
286     RegASwapB <= RegA(31 downto 16) & RegA(7 downto 0) & RegA(15 downto 8);
287
288     -- RegA with the high and low words swapped.
289     RegASwapW <= RegA(15 downto 0) & RegA(31 downto 16);
290
291     -- The center 32-bits of RegB and RegA (ie, low word of RegB and high word of RegA).
292     RegB_RegA_Center <= RegB(15 downto 0) & RegA(31 downto 16);
293
294     -- the zero/sign-extended version of register B
295     with AluCtrl.ExtMode select
296         ExtendedReg <= SignExtend(LowByte(RegB))  when  Ext_SignB_RegA,
297                        SignExtend(LowWord(RegB))  when  Ext_SignW_RegA,
298                        ZeroExtend(LowByte(RegB))  when  Ext_ZeroB_RegA,
299                        ZeroExtend(LowWord(RegB))  when  Ext_ZeroW_RegA,
300                        (others => 'X') when others;
301
302     -- Choose which system register to select
303     with SysCtrl.RegSel select
304         SysReg <= SR      when SysRegSel_SR,
305                   GBROut  when SysRegSel_GBR,
306                   VBR     when SysRegSel_VBR,
307                   MACH    when SysRegSel_MACH,
308                   MACL    when SysRegSel_MACL,
309                   PROut   when SysRegSel_PR,
310                   (others => 'X') when others;
311
312     -- Select the data to write the the register based on the decoded instruction.
313     with RegCtrl.DataInSel select
314         RegDataIn <= Result                     when  RegDataIn_ALUResult,
315                      ImmediateExt               when  RegDataIn_Immediate,
316                      RegA                       when  RegDataIn_RegA,
317                      RegB                       when  RegDataIn_RegB,
318                      SysReg                     when  RegDataIn_SysReg,
319                      RegASwapB                  when  RegDataIn_RegA_SwapB,
320                      RegASwapW                  when  RegDataIn_RegA_SwapW,
321                      RegB_RegA_Center           when  RegDataIn_REGB_REGA_CENTER,
322                      ExtendedReg                when  RegDataIn_Ext,
323                      MemDataIn                  when  RegDataIn_DB,
324
325                      -- Extract the T bit from the status register.
326                      SR and X"00000001"         when  RegDataIn_SR_TBit,
327                      PROut                      when  RegDataIn_PR,
328                      (others => 'X')            when  others;
329
330
```

```
331     -- The address being stored to a register is the pre-decremented or
332     -- post-incremented address when we are in that mode. If we are not in
333     -- that mode, it is just the normal address.
334     with DMAUCtrl.IncDecSel select
335         RegAxIn <= AddrSrcOut  when IncDecSel_PRE_DEC | IncDecSel_POST_INC,
336                    DataAddress when others;
337
338     -- Route control signals and data into register array
339     registers : entity work.SH2Regs
340     port map (
341         -- Inputs:
342         clock       => clock,
343         reset       => reset,
344         RegDataIn   => RegDataIn,
345         EnableIn    => RegCtrl.EnableIn,
346         RegInSel    => RegCtrl.InSel,
347         RegASel     => RegCtrl.ASel,
348         RegBSel     => RegCtrl.BSel,
349         RegAxIn     => RegAxIn,
350         RegAxInSel  => DMAUCtrl.AxInSel,
351         RegAxStore  => DMAUCtrl.AxStore,
352         RegA1Sel    => DMAUCtrl.A1Sel,
353         RegA2Sel    => DMAUCtrl.A2Sel,
354         -- Outputs:
355         RegA    => RegA,
356         RegB    => RegB,
357         RegA1   => RegA1,
358         RegA2   => RegA2
359     );
360
361
362     -- Always use RegA as Operand A for ALU
363     OperandA <= RegA;
364
365     -- Input mux for ALU Operand B
366     with ALUCtrl.OpBSel select
367         OperandB <=  RegB            when ALUOpB_RegB,
368                      ImmediateExt    when ALUOpB_Imm,
369                      (others => 'X') when others;
370
371     -- If two registers share a byte value
372     StrCmp <= '1' when (RegA(31 downto 24) = RegB(31 downto 24)) or
373                         (RegA(23 downto 16) = RegB(23 downto 16)) or
374                         (RegA(15 downto 8)  = RegB(15 downto 8))  or
375                         (RegA(7 downto 0)   = RegB(7 downto 0))
376                   else '0';
377
378     -- Compute T flag value based on ALU output flags. Used for operations
379     -- of the form CMP/XX.
380     with AluCtrl.TCmpSel select
381         TCmp <= Zero                              when TCMP_EQ,   -- 1 if Rn = Rm
382                 Cout                              when TCMP_HS,   -- 1 if Rn >= Rm, unsigned
383                 not (Sign xor Overflow)           when TCMP_GE,   -- 1 if Rn >= Rm, signed
384                 Cout and (not Zero)               when TCMP_HI,   -- 1 if Rn >  Rm, unsigned
385                 not ((Sign xor Overflow) or Zero) when TCMP_GT,   -- 1 if Rn >  Rm, signed
386                 StrCmp                            when TCMP_STR,  -- 1 if Rn byte matches Rm byte
387                 'X' when others;
388
389     -- Select what value T should be set to
390     with AluCtrl.TFlagSel select
391         TNext <=  TBit       when TFlagSel_T,           -- retain T flag
392                   Cout       when TFlagSel_Carry,       -- Set T flag to ALU carry flag
393                   Overflow   when TFlagSel_Overflow,    -- Set T flag to ALU overflow flag
394                   Zero       when TFlagSel_Zero,        -- set T flag to ALU Zero flag
395                   '0'        when TFlagSel_CLEAR,       -- clear T flag
396                   '1'        when TFlagSel_SET,         -- set T flag
```

```
397                 TCmp       when TFlagSel_CMP,         -- compute T flag based on compare result
398                 'X'        when others;
399
400     -- Route ALU control signals
401     alu : entity work.sh2alu
402     port map (
403         -- Inputs:
404         OperandA => OperandA,
405         OperandB => OperandB,
406         TIn      => TBit,
407         LoadA    => ALUCtrl.LoadA,
408         FCmd     => ALUCtrl.FCmd,
409         CinCmd   => ALUCtrl.CinCmd,
410         SCmd     => ALUCtrl.SCmd,
411         ALUCmd   => ALUCtrl.ALUCmd,
412         -- Outputs:
413         Result   => Result,
414         Cout     => Cout,
415         Overflow => Overflow,
416         Zero     => Zero,
417         Sign     => Sign
418     );
419
420     -- Use RegA1 (@Rn) if we are writing and RegA2 (@Rm) if we are reading.
421     with MemCtrl.ReadWrite select
422         RegSrc <= RegA2           when Mem_READ,
423                   RegA1           when Mem_WRITE,
424                   (others => 'X') when others;
425
426     -- Connect PCSrc to PCUsed
427     PCSrc <= PCUsed;
428
429     -- R0 comes from RegA2 when we are reading and RegA1 when we are writing.
430     with MemCtrl.ReadWrite select
431         R0Src <= RegA1            when Mem_READ,
432                  RegA2            when Mem_WRITE,
433                  (others => 'X')  when others;
434
435     -- Route DMAU control signals
436     dmau : entity work.sh2dmau
437     port map (
438         -- Inputs:
439         RegSrc       => RegSrc,
440         R0Src        => R0Src,
441         PCSrc        => PCSrc,
442         GBRIn        => GBRIn,
443         GBRWriteEn   => DMAUCtrl.GBRWriteEn,
444         Off4         => DMAUCtrl.Off4,
445         Off8         => DMAUCtrl.Off8,
446         BaseSel      => DMAUCtrl.BaseSel,
447         IndexSel     => DMAUCtrl.IndexSel,
448         OffScalarSel => DMAUCtrl.OffScalarSel,
449         IncDecSel    => DMAUCtrl.IncDecSel,
450         Clk          => clock,
451         -- Outputs:
452         Address    => DataAddress,
453         AddrSrcOut => AddrSrcOut,
454         GBROut     => GBROut
455     );
456
457
458     -- PMAU Register input is always RegB.
459     RegIn <= RegB;
460
461     -- Route PMAU control signals
462     pmau : entity work.sh2pmau
```

```
463     port map (
464         -- Inputs:
465         RegIn       => RegIn,
466         PRIn        => PRIn,
467         PCIn        => (others => '0'),         -- default PC is all 0s
468         PRWriteEn   => SysCtrl.PRWriteEn,
469         PCWriteCtrl => PMAUCtrl.PCWriteCtrl,
470         Off8        => PMAUCtrl.Off8,
471         Off12       => PMAUCtrl.Off12,
472         PCAddrMode  => PMAUCtrl.PCAddrMode,
473         Clk         => clock,
474         Reset       => Reset,
475         -- Outputs:
476         PCRegOut    => PCRegOut,
477         PCCalcOut   => PCCalcOut,
478         PROut       => PROut
479     );
480
481     -- Route memory interface control signals
482     memory_tx : entity work.MemoryInterfaceTx
483     port map (
484         -- Inputs:
485         clock      => clock,
486         MemEnable  => MemCtrl.Enable,
487         ReadWrite  => MemCtrl.ReadWrite,
488         MemMode    => MemCtrl.Mode,
489         Address    => unsigned(MemAddress),
490         MemDataOut => MemDataOut,
491         -- Outputs:
492         RE => ReadMask,
493         WE => WriteMask,
494         DB => DB
495     );
496
497     -- Route memory interface control signals
498     memory_rx : entity work.MemoryInterfaceRx
499     port map (
500         -- Inputs:
501         MemEnable => MemCtrl.Enable,
502         MemMode => MemCtrl.Mode,
503         Address => unsigned(MemAddress),
504         DB      => DB,
505         -- Outputs:
506         MemDataIn => MemDataIn
507     );
508
509     -- Route control unit control signals
510     control_unit : entity work.SH2Control
511     port map (
512         -- Inputs:
513         MemDataIn   => MemDataIn,
514         TFlagIn     => TNext,
515         clock       => clock,
516         reset       => reset,
517
518         -- Outputs:
519         MemCtrl => MemCtrl,
520         ALUCtrl => ALUCtrl,
521         RegCtrl => RegCtrl,
522         DMAUCtrl => DMAUCtrl,
523         PMAUCtrl => PMAUCtrl,
524         SysCtrl => SysCtrl
525     );
526
527     -- Mux system register input values based on SysRegSrc. Note that individual
528     -- write-enables (like GBRWriteEn and PRWriteEn) must be enabled seperately.
```

```
529     NextSysReg <= RegB      when SysCtrl.RegSrc = SysRegSrc_RegB  else
530                   MemDataIn when SysCtrl.RegSrc = SysRegSrc_DB    else
531
532                   -- The return address of a BSR is the PC at the point of decoding the BSR plus 4.
533                   std_logic_vector(unsigned(PCRegOut) + to_unsigned(4, 32))  when SysCtrl.RegSrc = SysRegSrc_PC    else
534
535                   (others => 'X');
536
537     GBRIn <= NextSysReg;    -- set GBR to selected sysreg value (when GBRWriteEn active)
538     PRIn  <= NextSysReg;    -- set PR to selected sysreg value (when PRWriteEn active)
539
540     register_proc: process(clock, reset)
541     begin
542         if reset = '0' then
543             -- Reset system registers (async)
544             SR   <=  (others => '0');
545             VBR  <=  (others => '0');
546             MACH <=  (others => '0');
547             MACL <=  (others => '0');
548
549         elsif rising_edge(clock) then
550             SR(0) <= TNext;     -- set new value of T
551
552             if SysCtrl.RegCtrl = SysRegCtrl_LOAD then
553                 -- Load new value into a system register
554                 -- (note that PR and GBR are handled separately)
555                 if SysCtrl.RegSel = SysRegSel_SR then
556                     SR <= NextSysReg;
557                 elsif SysCtrl.RegSel = SysRegSel_VBR then
558                     VBR <= NextSysReg;
559                 elsif SysCtrl.RegSel = SysRegSel_MACH then
560                     MACH <= NextSysReg;
561                 elsif SysCtrl.RegSel = SysRegSel_MACL then
562                     MACL <= NextSysReg;
563                 end if;
564             elsif SysCtrl.RegCtrl = SysRegCtrl_CLEAR then
565                 -- Clear a system register value
566                 -- (note that PR and GBR are handled separately)
567                 if SysCtrl.RegSel = SysRegSel_SR then
568                     SR <= (others => '0');
569                 elsif SysCtrl.RegSel = SysRegSel_VBR then
570                     VBR <= (others => '0');
571                 elsif (SysCtrl.RegSel = SysRegSel_MACH) or (SysCtrl.RegSel = SysRegSel_MACL) then
572                     -- Reset both MACH and MACL for CLRMAC instruction
573                     MACH <= (others => '0');
574                     MACL <= (others => '0');
575                 end if;
576             end if;
577         end if;
578     end process register_proc;
579
580 end architecture structural;
581
```

```
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.SH2ControlSignals.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             Tick;       -- tick CPU one more time to push instructions through pipeline
534         end procedure;
535 
536         -- Runs a test on the CPU. First loads an assembled program into ROM,
537         -- clocks the CPU until it stops running, then checks if the contents
538         -- of memory match up with the expected values. This procedure requires
539         -- the path of the test, which is then postfixed with ".bin" and
540         -- ".expect" to compute the path of the binary file and expect file,
541         -- respectively. The resulting memory is also dumped to a ".dump" file
542         -- for debugging.
543         procedure RunTest(path : string) is
544         begin
545             LogBothWithTime("Running test: " & path, LogFile);
546             LoadProgram(path);          -- write program in to ROM
547             RunCPU;                     -- execute program
548             DumpMemory(path, 0, 64);    -- dump memory to file
549             CheckOutput(path);          -- check RAM has expected values
550         end procedure;
551 
552     begin
553 
554         -- Run all CPU tests
555 
556         RunTest("asm/mov_reg");                         -- Tests Mov between registers                    (working)
557         RunTest("asm/reg_indirect");                    -- Tests register indirect addressing             (working)
558         RunTest("asm/arith");                           -- Tests arithmetic instructions (add, sub, etc)  (working)
559         RunTest("asm/logic");                           -- Tests logic instructions (and, or, xor, etc)   (working)
560         RunTest("asm/shift");                           -- Tests shift instructions (shll, shlr, etc)     (working)
561         RunTest("asm/cmp");                             -- Test CMP operations                            (working)
562         RunTest("asm/ext");                             -- Test zero/sign extension instructions          (working)
563         RunTest("asm/bshift");                          -- Test barrel shift instructions                 (working)
564         RunTest("asm/sr");                              -- Tests status register (SETT, CLRT)             (working)
565         RunTest("asm/system");                          -- Tests system register operations (LDC, STC)    (working)
566         RunTest("asm/control");                         -- Tests control register operations (LDS, STS)   (working)
567 
568         RunTest("asm/mov_wl_at_disp_pc_rn");            -- Tests Mov (disp, PC), Rn                           (working)
569         RunTest("asm/mov_bwl_at_rm_rn");                -- Tests Mov @Rm, Rn                                  (working)
570         RunTest("asm/mov_bwl_rm_at_minus_rn");          -- Tests Mov Rm, @-Rn                                 (working)
571         RunTest("asm/mov_bwl_at_rm_plus_rn");           -- Test Mov @Rm+, Rn                                  (working)
572         RunTest("asm/mov_bwl_r0_or_rm_at_disp_rn");     -- Test Mov R0, @(disp, Rn) and Mov Rm, @(disp,Rn)    (working)
573         RunTest("asm/mov_bwl_at_disp_rm_r0_or_rn");     -- Test Mov @(disp, Rm), R0 and Mov @(disp, Rm), Rn   (working)
574         RunTest("asm/mov_rm_at_r0_rn");                 -- Test Mov Rm, @(R0, Rn)                             (working)
575         RunTest("asm/mov_b_at_r0_rm_rn");               -- Test Mov @(R0, Rm), Rn                             (working)
576         RunTest("asm/mov_bwl_r0_at_disp_gbr");          -- Test Mov R0, @(disp, GBR)                          (working)
577         RunTest("asm/mov_at_disp_gbr_r0");              -- Test Mov @(disp, GBR), R0                          (working)
578         RunTest("asm/mova_at_disp_pc_r0");              -- Test Mova @(disp, PC), R0                          (working)
579         RunTest("asm/movt_rn");                         -- Test Movt Rn                                       (working)
580         RunTest("asm/swap");                            -- Test SWAP.B Rm, Rn and SWAP.W Rm, Rn               (working)
581         RunTest("asm/xtrct");                           -- Test XTRCT Rm, Rn                                  (working)
582         RunTest("asm/branch");                          -- Test branch instructions.
583         RunTest("asm/branch_conditional");              -- Test conditional branch instructions (BT, BF)      (working)
584         RunTest("asm/branch_conditional_delay");        -- Test conditional branch instructions (BT/S, BF/S)
585 
586         wait;
587     end process;
588 
589 end behavioral;
590 
591 
```