```
//------------------------------------------------------
// mips-behavioral.v
// modified by P. M. Kogge 3/7/08 from original mips.v by
// Max Yi (byyi@hmc.edu) and David_Harris@hmc.edu 12/9/03
// Model of subset of MIPS processor described in Ch 1
//     except rewritten as a simple behavioral model
// NOTE ALSO ORIGINAL DID NOT IMPLEMENT ADDI; THAT'S BEEN FIXED
//------------------------------------------------------

// top level design for testing
// SAME AS IN THE STRUCTURAL FORM

module top #(parameter WIDTH = 8, REGBITS = 3)();
   reg                 clk;
   reg                 reset;
   wire                memread, memwrite;
   wire    [WIDTH-1:0] adr, writedata;
   wire    [WIDTH-1:0] memdata;

   // instantiate devices to be tested
   mips  #(WIDTH,REGBITS)  dut(clk,  reset,  memdata,  memread,
memwrite, adr, writedata);

   // external memory for code and data
   exmemory  #(WIDTH)  exmem(clk,  memwrite,  adr,  writedata,
memdata);

   // initialize test
   initial
      begin
         reset <= 1; # 22; reset <= 0;
      end

   // generate clock to sequence tests
   always
      begin
         clk <= 1; # 5; clk <= 0; # 5;
      end

   always@(negedge clk)
      begin
         if(memwrite)
            if(adr == 5 & writedata == 7)
               $display("Simulation completely successful");
            else $display("Simulation failed");
      end
endmodule
```

```
// external memory accessed by MIPS
// SAME AS IN THE STRUCTURAL FORM
module exmemory #(parameter WIDTH = 8)
                 (clk, memwrite, adr, writedata, memdata);

   input                  clk;
   input                  memwrite;
   input      [WIDTH-1:0] adr, writedata;
   output reg [WIDTH-1:0] memdata;

   reg  [31:0] RAM [(1<<WIDTH-2)-1:0];
   wire [31:0] word;

   initial
      begin
         $readmemh("memfile.dat",RAM);
      end

   // read and write bytes from 32-bit word
   always @(posedge clk)
      if(memwrite)
         case (adr[1:0])
            2'b00: RAM[adr>>2][7:0] <= writedata;
            2'b01: RAM[adr>>2][15:8] <= writedata;
            2'b10: RAM[adr>>2][23:16] <= writedata;
            2'b11: RAM[adr>>2][31:24] <= writedata;
         endcase

   assign word = RAM[adr>>2];
   always @(*)
      case (adr[1:0])
         2'b00: memdata <= word[31:24];
         2'b01: memdata <= word[23:16];
         2'b10: memdata <= word[15:8];
         2'b11: memdata <= word[7:0];
      endcase
endmodule
```

```
// simplified MIPS processor IN BEHAVIORAL FORM
//   I/O from processor doesn't change
//   even though there are no internal modules
module mips #(parameter WIDTH = 8, REGBITS = 3)
             (input             clk, reset,
              input  [WIDTH-1:0] memdata,
              output             memread, memwrite,
              output [WIDTH-1:0] adr, writedata);

   parameter   FETCH1  =  4'b0001;
   parameter   FETCH2  =  4'b0010;
   parameter   FETCH3  =  4'b0011;
   parameter   FETCH4  =  4'b0100;
   parameter   DECODE  =  4'b0101;
   parameter   MEMADR  =  4'b0110;
   parameter   LBRD    =  4'b0111;
   parameter   LBWR    =  4'b1000;
   parameter   SBWR    =  4'b1001;
   parameter   RTYPEEX =  4'b1010;
   parameter   RTYPEWR =  4'b1011;
   parameter   BEQEX   =  4'b1100;
   parameter   JEX     =  4'b1101;
   parameter   ADDIEX  =  4'b1110;  //NEW

   parameter   LB      =  6'b100000;
   parameter   SB      =  6'b101000;
   parameter   RTYPE   =  6'b000000;
   parameter   BEQ     =  6'b000100;
   parameter   J       =  6'b000010;
   parameter   ADDI    =  6'b001000; //NEW

// We add the following as the function values for RTYPES
   parameter   ADD     =  6'b100000;
   parameter   SUB     =  6'b100010;
   parameter   AND     =  6'b100100;
   parameter   OR      =  6'b100101;
   parameter   SLT     =  6'b101010;

   reg [3:0] state, nextstate;
   reg       pcwrite, pcwritecond;
```

```
// next definitions made for implementing data flow behaviorally
// that are all updated at rising edge of clock
// These are all registers that would show up in data flow
   reg [7:0] ir0,ir1,ir2,ir3,mdr,areg,wrd,pcreg,res;
//  nextxxx  are  internal  values  used  to  compute  new  register
values
   reg [7:0] nextir0,nextir1,nextir2,nextir3;
   reg [7:0] nextpcreg,nextres,nextareg,nextwrd;
   reg [7:0] regfile [7:0] //register file of 8 entries
// following are internal values to make behavioral run right
   reg [31:0] instr;

   wire slt;


   // register updates at each clock
   always @(posedge clk)
      ir0 <= nextir0;
      ir1 <= nextir1;
      ir2 <= nextir2;
      ir3 <= nextir3;
      mdr <= memdata; // note always importing data from memory
      res <= nextres;
      areg <= nextreg;
      wrd <= nextwrd;
      instr <= ir3 & ir2 & ir1 & ir0;
      if(reset)
     begin  //reset state and pc
          state <= FETCH1;
          pcreg <= 0; // note reset of PC also
     end
      else
     begin //do the normal updates
          state <= nextstate;
          pcreg <= nextpcreg;
     end;
```

```
// next state logic - unchanged from original
   always @(*)
      begin
         case(state)
            FETCH1:  nextstate <= FETCH2;
            FETCH2:  nextstate <= FETCH3;
            FETCH3:  nextstate <= FETCH4;
            FETCH4:  nextstate <= DECODE;
            DECODE:  case(op)
                        LB:      nextstate <= MEMADR;
                        SB:      nextstate <= MEMADR;
                        RTYPE:   nextstate <= RTYPEEX;
                        BEQ:     nextstate <= BEQEX;
                        J:       nextstate <= JEX;
               ADDI:      nextstate <= ADDIEX; // NEW
                        default: nextstate <= FETCH1; // should
never happen
                     endcase
            MEMADR:  case(op)
                        LB:      nextstate <= LBRD;
                        SB:      nextstate <= SBWR;
                        default: nextstate <= FETCH1; // should
never happen
                     endcase
            LBRD:    nextstate <= LBWR;
            LBWR:    nextstate <= FETCH1;
            SBWR:    nextstate <= FETCH1;
            RTYPEEX: nextstate <= RTYPEWR;
            RTYPEWR: nextstate <= FETCH1;
         ADDIEX:  nextstate <= RTYPEWR;
            BEQEX:   nextstate <= FETCH1;
            JEX:     nextstate <= FETCH1;
            default: nextstate <= FETCH1; // should never happen
         endcase
      end
```

```
// Following execution logic replaces control signal assignment
by behavioral description
//  for  each  state,  the  appropriate  register  transfers  are
specified directly
   always @(*)
      regwrite <= 0;
      // Internal registers may or may not change; set to remain
constant
      // Note mdr always being loaded from memory in prior loop
      nextir0 <= ir0;
      nextir1 <= ir1;
      nextir2 <= ir2;
      nextir3 <= ir3;
      nextareg <= areg;
      nextwrd <= wrd;
      nextres <= res;
      //  Now  figure  out  which  cycle  we're  in  and  do  the
transfers.
      begin
            case(state)
               FETCH1:
                  begin //read into ir0
                     memread <= 1;
               adr <= pc;
               nextpc <= pc + 1;
               nextir0 <= memdata;
                  end
               FETCH2:
                  begin //read into ir0
                     memread <= 1;
                     adr <= pc;
               nextpc <= pc + 1;
               nextir1 <= memdata;
                  end
               FETCH3:
                  begin //read into ir0
                     memread <= 1;
                     adr <= pc;
               nextpc <= pc + 1;
               nextir2 <= memdata;
                  end
```

```
FETCH4:
   begin //read into ir0
      memread <= 1;
      adr <= pc;
nextpc <= pc + 1;
nextir3 <= memdata;
   end
DECODE:
   begin // read register file
// note in reality could be done continuously
// in the register update section
     nextareg <= regfile[23:21];
     nextwrd <= regfile[18:16];
   end
MEMADR:
   begin // compute address into res register
res <=  areg + (instr[5:0] & 2'b00);
   end
LBRD:
   begin  //do the read for LB
      memread <= 1;
      adr <= res;
   end
LBWR:
   begin  //save the just read data into RF
      regfile[18:16] <= mdr;
   end
SBWR:
   begin
      memwrite <= 1;
      writedata <= wrd;
   end
```

```
               RTYPEEX:
                  begin
                    case(instr[5:0])
               ADD: nextres <= areg + wrd;
               SUB: nextres <= areg + ~wrd + 8'b00000001;
               AND: nextres <= areg & wrd;
               OR:  nextres <= areg | wrd;
               SLT:
                    begin // do test in same way as original
                    assign    slt    =    (areg    +    ~wrd    +
8'b00000001)[WIDTH-1]
                    nextres <= 7'b0000000 & slt;
                    end;
             endcase
                  end
               RTYPEWR:
                  begin
                     regfile[13:11] <= res;
                  end
           ADDIEX: nextres <= areg + instr[7:0];  // NEW
               BEQEX:
                  if  (areg  ==  wrd)  nextpcreg  <=  pcreg  +
instr[7:0];
               JEX:
                  begin
                     nextpcreg <= instr[5:0] & 2'b00;
                  end
         endcase
      end
endmodule
```