```
module btbcon (CLK, op1, PC1, btin1, hit, tak, btout, op2, PC2, btin2);

input CLK;
input [1:0] op1;
input [31:0] PC1;
input [31:0] btin1;

input [1:0] op2;
input [31:0] PC2;
input [31:0] btin2;

output hit;
output tak;
output [31:0] btout;

// the two sets of our BTB cache
// here’s how the bits break down:
// bits 60 - 34 are the BrIt bits (27 in all)
// bits 33-4 are the BrTa bits (30 in all)
// bits 3-2 are the Prediction bits - these bits correspond to the state
//          bits from the 2-bit saturating counter in hw2
// bit 1 is the LRU bit - high indicates this was LRU
// bit 0 is the inUse bit - high indicates this is a valid entry
reg [60:0] leftCache [0:7];
reg [60:0] rightCache [0:7];

reg [60:0] leftPredict;   // "temp" variables that we need to munge our cache
reg [60:0] rightPredict;  // this is necessary because 2-d arrays aren’t bit
                          // accessible
reg [60:0] leftUpdate;
reg [60:0] rightUpdate;

reg predictHit;      // did we get a BTB hit?
reg predictLeftHit;  // did we get a hit in the left register?

reg updateHit;
reg updateLeftHit;

reg [33:0] outWire;
reg [2:0] counter; // used for initializing cache only

assign hit = outWire[33];
assign tak = outWire[32];
assign btout = outWire[31:0];

// used to define state of the two-bit saturating counter
‘define ZERO  2’b00
‘define ONE   2’b01
‘define TWO   2’b10
‘define THREE 2’b11

// must properly initialize cache by setting all use bits to 0
initial
begin
  counter = 3’b000;
  repeat(8)
    begin
      leftPredict = leftCache[counter];
      leftPredict[0] = 1’b0;
      leftCache[counter] = leftPredict;
      rightPredict = rightCache[counter];
      rightPredict[0] = 1'b0;
      rightCache[counter] = rightPredict;

      counter = counter + 1;
    end
end

// give outWire the value of the output_logic function and then
// process op2
always @ (posedge CLK)
begin

  // grab the appropriate cache line, addressed by PC bits 4 through 2
  leftPredict = leftCache[PC1[4:2]];
  rightPredict = rightCache[PC1[4:2]];

  leftUpdate = leftCache[PC2[4:2]];
  rightUpdate = rightCache[PC2[4:2]];

  // check the BrIT field for a BTB hit with respect to op1
  if((leftPredict[60:34] == PC1[31:5]) && leftPredict[0])
    begin
      predictHit = 1'b1;
      predictLeftHit = 1'b1;
    end

  else if((rightPredict[60:34] == PC1[31:5]) && rightPredict[0])
    begin
      predictHit = 1'b1;
      predictLeftHit = 1'b0;
    end

  else
    predictHit = 1'b0;  // must remember to set this bit low if miss!

  // set the results of the prediction
  outWire = predict_output(predictHit,predictLeftHit,leftPredict,rightPredict);

  // check the BrIT field for a BTB hit with respect to op2
  if((leftUpdate[60:34] == PC2[31:5]) && leftUpdate[0])
    begin
      updateHit = 1'b1;
      updateLeftHit = 1'b1;
    end

  else if((rightUpdate[60:34] == PC2[31:5]) && rightUpdate[0])
    begin
      updateHit = 1'b1;
      updateLeftHit = 1'b0;
    end

  else
    updateHit = 1'b0;  // must remember to set this bit low if miss!


  case(op2)

  'ZERO:   ; // no-op
  default:   // else we have an update
    begin
```

```
if(updateHit)  // handle the case where we have a BTB hit on an update op
  begin

    // update the prediction bits
    if(updateLeftHit)
      begin
        leftUpdate[3:2] = next_predict_logic(op2[0],leftUpdate[3:2]);
        leftUpdate[1] = 0;  // set LRU bits
        rightUpdate[1] = 1;
      end

    else
      begin
        rightUpdate[3:2] = next_predict_logic(op2[0],rightUpdate[3:2]);
        rightUpdate[1] = 0;
        leftUpdate[1] = 1;
      end


    if(op2==3) // op == 3 in this case
      begin
        if(updateLeftHit)
          leftUpdate[33:4] = btin2[31:2];
        else
          rightUpdate[33:4] = btin2[31:2];
      end
  end   // end if(updateHit)

else        // handle the case where we have a BTB miss on an update op
  begin

    if(op2[0]) // we only need to do something on a miss if we update taken
      begin
        // implement replacement scheme here
        // we'll do a case on the prediction bits concat with the use bits
        // of the left and right cache entries (in that order)

        casex({leftUpdate[3:2],leftUpdate[0],rightUpdate[3:2],rightUpdate[0]})
         //PrUPrU
        6'bxx0xxx: // update the left cache block.  this case corresponds
                   // to a compulsory cache miss; we set the LRU bit even
                   // though we don't really need to
          leftUpdate = {PC2[31:5],btin2[31:2],'ONE,1'b0,1'b1};

        6'bxx1xx0: // update the right cache block.  this case corresponds
                   // to the case where the left cache block is filled
                   // but not the right one.  we'll need to update LRU here
          begin
            rightUpdate = {PC2[31:5],btin2[31:2],'ONE,1'b0,1'b1};
            leftUpdate[1] = 1'b1;
          end

        6'b0x11x1: // both blocks in use; left says Take, right says NT
          begin
            rightUpdate = {PC2[31:5],btin2[31:2],'ONE,1'b0,1'b1};
            leftUpdate[1] = 1'b1;
          end

        6'b1x10x1: // both blocks in use; left says NT, right say Take
          begin
            leftUpdate = {PC2[31:5],btin2[31:2],'ONE,1'b0,1'b1};
```

```
                rightUpdate[1] = 1’b1;
              end

            6’b0x10x1: // both blocks in use; both predict Take
              begin
                if(leftUpdate[1])  // the left block is LRU - throw it out
                  begin
                    leftUpdate = {PC2[31:5],btin2[31:2],‘ONE,1’b0,1’b1};
                    rightUpdate[1] = 1;
                  end
                else  // the right block is LRU - throw it out
                  begin
                    rightUpdate = {PC2[31:5],btin2[31:2],‘ONE,1’b0,1’b1};
                    leftUpdate[1] = 1;
                  end
              end

            6’b1x11x1: // both blocks in use; both predict NT
              begin
                if(leftUpdate[1])  // the left block is LRU - throw it out
                  begin
                    leftUpdate = {PC2[31:5],btin2[31:2],‘ONE,1’b0,1’b1};
                    rightUpdate[1] = 1;
                  end
                else  // the right block is LRU - throw it out
                  begin
                    rightUpdate = {PC2[31:5],btin2[31:2],‘ONE,1’b0,1’b1};
                    leftUpdate[1] = 1;
                  end
              end

            default: $display("something went wrong in the replacement");

            endcase

          end

      end   // end else

    end     // end default: begin
  endcase   // end case(op[2])

  // now write the cache lines back to the cache
  leftCache[PC2[4:2]] = leftUpdate;
  rightCache[PC2[4:2]] = rightUpdate;

end  // end always @ (posedge CLK)


function [33:0] predict_output;
  input BTBhit;
  input leftHit;
  input [60:0] leftTemp;
  input [60:0] rightTemp;

  if(BTBhit & leftHit)        // a hit in the "left" cache
    begin
      if(~leftTemp[3])  // we have a BTB hit and we predict taken
        predict_output = {1’b1,1’b1,leftTemp[33:4],‘ZERO};
      else             // we have a BTB hit and we predict not taken
        predict_output = {1’b1,1’b0,{32{1’bx}}};
    end

  else if(BTBhit & ~leftHit)  // a hit in the "right" cache
    begin
      if(~rightTemp[3])  // we have a BTB hit and we predict taken
        predict_output = {1’b1,1’b1,rightTemp[33:4],‘ZERO};
      else             // we have a BTB hit and we predict not taken
        predict_output = {1’b1,1’b0,{32{1’bx}}};
    end

  else // we have a BTB miss
    predict_output = {1’b0,{33{1’bx}}};

endfunction


function [1:0] next_predict_logic;
  input in;
  input [1:0] state;

  if(in) // in == 1
    begin
      if(state == ‘THREE)
        next_predict_logic = ‘TWO;
      else // all other states go to ZERO on a 1
        next_predict_logic = ‘ZERO;
    end

  else // in == 0
    begin
      if(state == ‘ZERO)
        next_predict_logic = ‘ONE;
      else
        next_predict_logic = ‘THREE;
    end
endfunction

endmodule
```