ECE485/585 Microprocessor System Design

Portland State University

L1 Cache Simulation

Andy Goetz, Bradon Kanyid,
Eric Krause, and Kevin Riedl

December 3, 2012

Version

Document Review		
Version	**Comments**	**Date**
1.0	Initial release.	12/3/2012

1 Specification

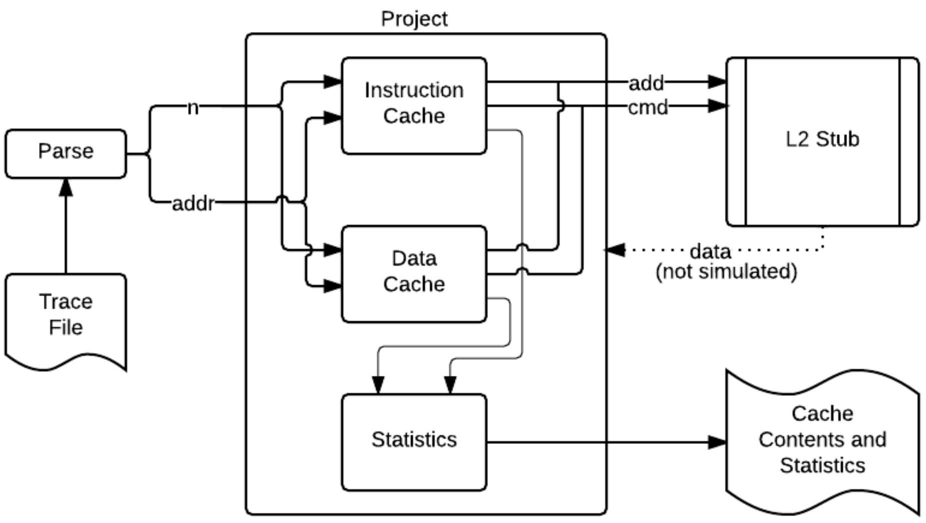

Figure 1: Level 1 Diagram

This project functionally simulates a split instruction/data L1 cache for a 32-bit processor in a system with multiple processors. The system employs MESI protocol to ensure cache coherence. The instruction cache is 2-way set associative, consists of 16K sets, and has 64-byte lines. The data cache is 4-way set associative, consists of 16K sets, and has 64-byte lines. Both caches employ LRU replacement policy and are backed by an L2 cache (which is modeled as a stub in this simulation). Statistics regarding number of reads, writes, hits, and misses are generated, as well as a hit percentage rate. This simulation has a single-cycle interface between a processor and L1, and between L1 and L2. All processor reads and writes are a single byte.

The 32-bit addresses from the processor are broken down into the following fields:

31:20	= 12-bit tag
19:6	= 14-bit index
5:0	= 6-bit offset

We specify the following method for interface with L2. A 26-bit address specifies a 64-byte cache line, and must be supplied with one of the following 2-bit cmd_out commands:

00	No Operation (ignore any input on address lines)
01	Read from L2
10	Write to L2
11	Read with intent to modify

In order to run, our simulation requires a trace file formatted using the protocol specified in the project description. A print command will output human-readable cache contents and statistics.

2 Assumptions

In the course of designing our L1 cache, we were forced to make several assumptions regarding the CPU it was designed for:

- *The cache hierarchy is inclusive.* By making the L2 cache support an inclusive policy, the synchronization logic between the L1 and L2 cache is greatly simplified.

- *The data cache is write-through.* The L1 and L2 caches together are required to support memory writebacks. However, because the cache design also needs to support MESI, we decided to implement the L1 data cache as a simple write-through cache. Because the cache hierarchy uses an inclusive policy, evictions forced by MESI in the L2 cache will force an eviction in the L1 cache. If the L1 cache had a dirty line, the MESI eviction would force a writeback to main memory in the L1, greatly complicating our code.
- *Cache contents (actual data) are irrelevant for this simulation.* Thus, all byte offsets are ignored. Because we only stub out the processor, the next level cache, and our cache eviction policy is based entirely on memory addresses, there is no need to examine data values.
- *All read and write operations reference single byte locations.* In order to simplify the cache design, we assume that all reads and writes reference a single byte. This means we do not need to worry about supporting unaligned memory references.

3 Pseudocode

Pseudocode for the data and instruction caches are listed below:

3.1 Data Cache

```
On clock edge,
  get command
  dispatch to handler

  reset handler:
    for all sets in cache
       set LRU bits to 0
       for all ways in set
          set Valid bit to 0
          set tag bits to 0
       end for
    end for

  invalidate handler:
    for all ways in the set indexed by the invalidate command
      if tag in tag array at offset matches invalidate tag
        set valid bit for matching tag to 0

  read handler:
    reads++ for stats
    for all ways in the set indexed by the read command
      see if the tag of the read address matches any tag in tag array and valid
      if yes, there's a hit.
      else miss

      for hit
        calculate new LRU bits based on hit way being MRU
        increase hit count for stats

      for miss
        increase miss count for stats
        for all ways in set indexed by read command
          check if any way is invalid
          if yes,
            fetch from L2 w/ read_out command
            calculate new LRU bits based on invalid way being MRU
            write read tag to tag array indexed by invalid way
            set the invalid way to be valid now
          if no, that means your ways are all full. Must evict the LRU way.
            fetch from L2 w/ read_out command
            calculate new LRU bits based on evicted way being MRU
            write read tag to tag array indexed by evicted way

  write handler:
    writes++ for stats
    for all ways in the set indexed by the write command
      see if the tag of the write address matches any tag in tag array and valid
      if yes, there's a hit.
      else miss

      for hit
        calculate new LRU bits based on hit way being MRU
        increase hit count for stats
        write out to L2

      for miss
        increase miss count for stats
        for all ways in set indexed by write command
          check if any way is invalid

          if yes,
            fetch from L2 cache w/ read w/ intent to modify command
            calculate new LRU bits based on invalid way being MRU
            write read tag to tag array indexed by invalid way
            set the invalid way to be valid now
            write out modified data to L2 cache
          if no, that means your ways are all full. Must evict the LRU way.
            fetch from L2 cache w/ read w/ intent to modify command
            calculate new LRU bits based on evicted way being MRU
            write written tag to tag array indexed by evicted way
            write out modified data to L2 cache

  print handler:
    for all the sets in the cache
    if any line is valid,
      print out that set, including index, lru, valid bits, and tag bits
```

3.2 Instruction Cache

```
On clock edge,
  get command
  dispatch to handler

  reset handler:
    for all sets in cache
       set LRU bit to 0
```

```
      for all ways in set
         set Valid bit to 0
         set tag bits to 0
      end for
   end for

 invalidate handler:
   for all ways in the set indexed by the invalidate command
     if tag in tag array at offset matches invalidate tag
       set valid bit for matching tag to 0

 instruction fetch handler:
   reads++ for stats
   for all ways in the set indexed by the read command
     see if the tag of the read address matches any tag in tag array and valid
     if yes, there's a hit.
     else miss

     for hit
       calculate new LRU bit based on hit way being MRU
       increase hit count for stats

     for miss
       increase miss count for stats
       for all ways in set indexed by read command
         check if either way is invalid

         if yes,
           fetch from L2 w/ read_out command
           calculate new LRU bits based on invalid way being MRU
           write read tag to tag array indexed by invalid way
           set the invalid way to be valid now
         if no, that means your ways are all full. Must evict the LRU way.
           fetch from L2 w/ read_out command
           calculate new LRU bits based on evicted way being MRU
           write read tag to tag array indexed by evicted way

 print handler:
   for all the sets in the cache
   if any line is valid,
     print out that set, including index, lru, valid bits, and tag bits
```

4 Testing

In order to test our implementation, we developed a set of test stimulus covering all of the corner cases. These tests are listed below.

In addition to the testplan developed by our team, we also performed blackbox testing with another ECE485 team. We traded testbenches, and confirmed that both of our implementations produced the same hit ratios.

4.1 Cached_Instr_Reads.trace

Summary	Multiple reads to the same instruction cache line.
Expected Results	Way 0 of Index 0 of Instruction Cache is valid. No other ways are valid.
Hit Ratio	75%

Test Vector	2 00000001
	2 00000002
	2 00000003
	2 00000004
	9

4.2 Cached_Data_Reads.trace

Summary	Multiple reads to the same data cache line.
Expected Results	Way 0 of Index 0 of Data Cache is valid. No other ways are valid.
Hit Ratio	75%

Test Vector	0 00000000
	0 00000001
	0 00000002
	0 00000003
	9

4.3 Interleaved_Read_Write.trace

Summary	Reads and Writes to the data cache.
Expected Results	Way 0 of Index 0 of Data Cache is valid. No other ways are valid.
Hit Ratio	75%

Test Vector	0 00000000
	1 00000001
	0 00000002
	1 00000003
	9

4.4 Same_Set_Instr.trace

Summary	Multiple reads from the same set in the instruction cache.
Expected Results	Both ways of Index 0 are valid.
Hit Ratio	0%

Test Vector	2 00000000
	2 00100000
	9

4.5 Same_Set_Data.trace

Summary	Multiple reads from the same set in the data cache.
Expected Results	All four ways of index 0 are valid.
Hit Ratio	0%

Test Vector	0 00000000
	1 00100000
	0 00200000
	1 00300000
	9

4.6 Instr_Conflict.trace

Summary	Enough reads to the same instruction cache set to cause an eviction.
Expected Results	Both ways of index 0 are valid. LRU bit is 1. Way 0 has Tag 200 in it, Way 1 has 100 in it.
Hit Ratio	0%

Test Vector	2 00000000
	2 10000000
	2 20000000
	9

4.7 Data_Conflict.trace

Summary	Enough reads and writes to the same instruction cache set to cause an eviction.
Expected Results	All ways of index 0 are valid. LRU way is 1. Way 0 has 500 in it, Wa 1 has 200 in it, way 2 has 300 in it, and way 3 has 400 in it.
Hit Ratio	0%

Test Vector	0 10000000
	1 20000000
	0 30000000
	1 40000000
	0 50000000
	9

4.8 Instr_Invalidate.trace

Summary	Multiple Reads Followed by Invalidate clears single line.
Expected Results	way 0 of index 0 is invalid, way 1 of index 0 is valid.
Hit Ratio	0%

Test Vector	2 00100000
	2 00200000
	9 00000000
	3 00100000
	9 00000000

4.9 Data_Invalidate.trace

Summary	Multiple Reads Followed by Invalidate clears single line.
Expected Results	way 1 of index 0 of data cache invalid, rest of ways of index 0 valid.
Hit Ratio	0%

Test Vector	0 00100000
	0 00200000
	0 00300000
	0 00400000
	9 00000000
	3 00200000
	9 00000000

4.10 Instr_Clear.trace

Summary	Read Followed by Clear empties data cache.
Expected Results	No ways in instruction cache are valid.
Hit Ratio	0%

Test Vector	2 00112300
	9 00000000
	8 00000000
	9 00000000

4.11 Data_Clear.trace

Summary	Read followed by Clear empties data cache.
Expected Results	All ways of Data Cache are invalid.
Hit Ratio	0%

Test Vector	0 00112300
	9 00000000
	8 00000000
	9 00000000

4.12 Instr_Invalidate_read.trace

Summary	Read, Invalidate, Read uses cleared way as LRU.
Expected Results	Way 0 of index 400 has tag of 001. Way 1 of index 400 has tag of 003.
Hit Ratio	0%

Test Vector	2 00110000
	2 00210000
	9 00000000
	3 00210000
	2 00310000
	9 00000000

4.13 Data_Invalidate_read.trace

Summary	Read, Invalidate, Read uses cleared way as LRU.
Expected Results	Index 400 of data cache has tag 001 in way 0, tag 002 in way 1, tag 005 in way 2, and tag 004 in way 3.
Hit Ratio	0%

Test Vector	0 00110000
	0 00210000
	0 00310000
	0 00410000
	9 00000000
	3 00310000
	0 00510000
	9 00000000

5 Source Code

The Verilog source code for our project is reproduced below:

5.1 memtest.v

```
////////////////////////////////////////////////////////////////////////////////
// ECE 485/585: Microprocessor System Design
// Portland State University - Fall 2012
// Final Project:
//
// File:      memtest.v (Test Bench)
// Authors: Andy Goetz, Bradon Kanyid, Eric Krause, and Kevin Riedl
// Description: This module reads in a stimulus file provided by the
// command line and passes commands to the cache.
//
////////////////////////////////////////////////////////////////////////////////

module test();

  parameter CLOCK_CYCLE  = 20;
  parameter CLOCK_WIDTH  = CLOCK_CYCLE/2;
  parameter TRUE   = 1'b1;
  parameter FALSE  = 1'b0;

  reg Clock;
  integer file;      // the file handle
  reg  done;
  reg  [3:0]         command;
  reg  [31:0]        value;
  reg  [9000:0]      filename;
  wire [25:0]        add_out;
  wire [1:0]         cmd_out;
  integer count;

  PROJECT project(
            .clk(Clock),
            .n(command),
            .add_in(value),
            .done(done),
            .add_out(add_out),
            .cmd_out(cmd_out)
            );

  L_NEXT l_next(
                  .add_in(add_out),
                  .cmd_in(cmd_out)
  );

  initial
  begin
          Clock = FALSE;
          done = FALSE;
```

```
         // Check to make sure that a stimulus file was provided
         if ($value$plusargs("stimulus=%s", filename) == FALSE)
           begin
           $display("ERROR: No Stimulus specified. Please specify\
           +stimulus=<filename> to start.");
           $finish;
         end

         // If it was, open the file
         file = $fopen(filename, "r");
         count = 2;

         // simulate initial reset
         #CLOCK_WIDTH Clock = FALSE;
         command = 4'd8;
         #CLOCK_WIDTH Clock = TRUE;

         // While there are lines left to be read:
         while (count > 1)
         begin
                 // Parse the line
                 #CLOCK_WIDTH Clock = FALSE;
                 count = $fscanf(file, "%d %x", command, value);
                 #CLOCK_WIDTH Clock = TRUE;
         end

         // Close the file, and finish up
         $fclose(file);
         done = TRUE;
  end
endmodule
```

5.2 PROJECT.v

```
////////////////////////////////////////////////////////////////////////////////
// ECE 485/585: Microprocessor System Design
// Portland State University - Fall 2012
// Final Project:
//
// File:     PROJECT.v
// Authors: Andy Goetz, Bradon Kanyid, Eric Krause, and Kevin Riedl
// Description: Top-level wrapper module for project
//
////////////////////////////////////////////////////////////////////////////////
module PROJECT(
  input clk,
  input clear,
  input [3:0] n,
  input [31:0] add_in,
  input done,
  output reg [25:0] add_out,
  output reg [1:0]  cmd_out
  );

  // valid commands from tracefile
  parameter RESET       = 4'd8;
  parameter INVALIDATE  = 4'd3;
  parameter READ        = 4'd0;
  parameter WRITE       = 4'd1;
  parameter INST_FETCH  = 4'd2;
  parameter PRINT       = 4'd9;

  // signals from file to caches
  wire [31:0] i_add, d_add;
  assign i_add = add_in;
  assign d_add = add_in;

  // signals between caches and next-level cache
  wire [1:0]    l2_i_cmd, l2_d_cmd;
  wire [25:0]   l2_i_add,  l2_d_add;

  // signals to/from stats
  wire [31:0] i_hit;
  wire [31:0] d_hit;
  wire [31:0] i_miss;
  wire [31:0] d_miss;
  wire [31:0] i_reads;
  wire [31:0] d_reads;
  wire [31:0] d_writes;

  //mux the L2 outputs
  always @(n)
  begin
              if(n == INST_FETCH)
    begin
      add_out = l2_i_add;
      cmd_out = l2_i_cmd;
    end
                  else
    begin
      add_out = l2_d_add;
      cmd_out = l2_d_cmd;
    end
  end

        INS_CACHE i_cache (
                .clk(clk),
                .n(n),
                .add_in(add_in),
                .add_out(l2_i_add),
                .cmd_out(l2_i_cmd),
                .hit(i_hit),
                .miss(i_miss),
                .reads(i_reads)
                );

        DATA_CACHE d_cache (
                .n(n),
                .add_in(add_in),
                .clk(clk),
                .add_out(l2_d_add),
                .cmd_out(l2_d_cmd),
                .hit(d_hit),
                .miss(d_miss),
                .reads(d_reads),
                .writes(d_writes)
                );

        STATS stats(
                .print(done),
                .ins_reads(i_reads),
                .ins_hit(i_hit),
                .ins_miss(i_miss),
                .data_reads(d_reads),
                .data_writes(d_writes),
                .data_hit(d_hit),
                .data_miss(d_miss)
                );

endmodule
```

5.3 INS_CACHE.v

```
////////////////////////////////////////////////////////////////////////////////
// ECE 485/585: Microprocessor System Design
// Portland State University - Fall 2012
// Final Project:
//
// File:     INS_CACHE.v
// Authors: Andy Goetz, Bradon Kanyid, Eric Krause, and Kevin Riedl
// Description: Simulates an instruction cache.
//
////////////////////////////////////////////////////////////////////////////////
`define SETS 1024*16
`define WAYS 2
`define SETBITS 14
`define  TAGBITS 12

module INS_CACHE(
  // INPUTS
  input [3:0]   n,        // from trace file
  input [31:0] add_in,  // from trace file
  input clk,

  // OUTPUTS
  output reg [25:0] add_out = 26'bZ,  // to next-level cache
  output reg [1:0]  cmd_out = NOP,    // to next-level cache
  output reg [31:0] hit      = 32'b0,  // to statistics module
  output reg [31:0] miss     = 32'b0,  // to statistics module
  output reg [31:0] reads     = 32'b0    // to statistics module
  );

  parameter TRUE        = 1'b1;
  parameter FALSE       = 1'b0;

  // instruction cache only reponds to following values of n:
  parameter RESET          = 4'd8;
  parameter INVALIDATE    = 4'd3;
  parameter INST_FETCH    = 4'd2;
  parameter PRINT          = 4'd9;

  // instruction cache sends following commands to next-level cache
  parameter READ_OUT     = 2'b01;
  parameter NOP          = 2'b00;

  // CACHE ELEMENTS
  // LRU: 1 bit per set. Encoding:  1 = Way 1 is LRU.  0 = Way 0 is LRU
  reg LRU [`SETS-1:0];
  // Valid: 1 bit per way.  Encoding:  1 = way is valid, 0 = not valid
  reg Valid [`SETS-1:0][`WAYS-1:0];
  // Tag: Tag is of size TAGBITS.  One tag per way.
  reg [`TAGBITS-1:0] Tag [`SETS-1:0][`WAYS-1:0];

  // loop counters
  integer set_cnt, way_cnt;
```

```
// internal
reg done = 1'b0;

// assignments
wire [11:0] curr_tag = add_in[31:20];
wire [13:0] curr_index = add_in[19:6];

always @(posedge clk)
begin
  add_out = 26'bZ;   // always initialize address out to high-z
  cmd_out = NOP;     // default to NOP, if a read happens, it will be updated
  done    = FALSE;   // and set internal done signal to false

  case(n)
    // RESET: iterates through all elements in the cache and sets
    //    everything to 0. Also initializes hit/miss/read counters.
    RESET:
    begin
      hit   = 32'b0;
      miss  = 32'b0;
      reads = 32'b0;

      // for every set...
      for (set_cnt = 0; set_cnt < `SETS; set_cnt = set_cnt + 1'b1)
      begin
        LRU[set_cnt] = 1'b0;  // set the LRU to 0
        // for each way of set...
        for (way_cnt = 0; way_cnt < `WAYS; way_cnt = way_cnt + 1'b1)
        begin
          // clear valid and tag bits.
          Valid  [set_cnt][way_cnt]  = FALSE;
          Tag    [set_cnt][way_cnt]  = `TAGBITS'b0;
        end
      end
    end


    // INVALIDATE: use address passed in with invalidate command as an
    //    index to a given line. Then, invalidate the line for which the
    //    stored tag equals the tag passed in add_in.
    INVALIDATE:
    begin
      for (way_cnt = 0; way_cnt < `WAYS; way_cnt = way_cnt + 1'b1)
      begin
        if (!done)
        begin
          if (Tag[curr_index][way_cnt] == curr_tag)
          begin
            done                         = TRUE;
            Valid[curr_index][way_cnt]   = FALSE;
          end
        end
      end
```

```
end


INST_FETCH:
begin
  reads = reads + 1'b1;  // always increment read count

  // First, look at both lines.  if for either, the tags match
  //     and the line is valid, then the read was a hit.  done
  //     is set to true, and execution will drop through the rest
  //     of the INST_FETCH routine.
  for (way_cnt = 0; way_cnt < `WAYS; way_cnt = way_cnt + 1'b1)
  begin
    if (done == FALSE)
      if (Tag[curr_index][way_cnt] == curr_tag &&
          Valid[curr_index][way_cnt] == TRUE)
      begin
        LRU[curr_index] = ~way_cnt[0];
        hit             = hit + 1'b1;
        done             = TRUE;
      end
    else ;
  end

  //  at this point, if done is still false, then the fetch was not a hit.
  if (done == FALSE)
    miss = miss + 1'b1;

  // Next, look at both lines.  If either is empty then
  //     do a read and and put result in the empty line, then set
  //     done to true, and execution will drop through the rest of
  //     the INST_FETCH routine.
  for (way_cnt = 0; way_cnt < `WAYS; way_cnt = way_cnt + 1'b1)
  begin
    if (done == FALSE)
      if (Valid[curr_index][way_cnt] == FALSE)
      begin
        // set L_NEXT command/address
        add_out                     = add_in[31:6]; // perform read
        cmd_out                     = READ_OUT;     // perform read

        Tag[curr_index][way_cnt]    = curr_tag;
        Valid[curr_index][way_cnt]  = TRUE;
        LRU[curr_index]             = ~way_cnt[0];
        done                        = TRUE;
      end
  end

  // Reaching this point means an eviction is needed because the
  //     instruction fetch was a miss, and there was no empty line
  //     in which to put the incoming read.  So evict the LRU
  if (done == FALSE)
    begin
      // set L_NEXT command/address
```

```
            add_out                                     = add_in[31:6]; // perform read
            cmd_out                                     = READ_OUT;     // perform read

            Tag[curr_index][LRU[curr_index]]          = curr_tag;
            LRU[curr_index]                           = ~LRU[curr_index];
          end
      end

      PRINT:
      begin
        // print header
        $display("\n-------- INSTRUCTION CACHE CONTENTS --------");
        $display(" Index | LRU | V[0]|Tag[0]| V[1]|Tag[1]");
        // cycle through all of the ways within a set
        for (way_cnt = 0;  way_cnt < `SETS; way_cnt = way_cnt+1)
        begin
          // print out the whole set if there are any valid lines
          if (Valid[way_cnt][0] | Valid[way_cnt][1])
          begin
            $display(" %4h  |  %d  |  %d  | %3h  |  %d  | %3h",
              way_cnt[`SETBITS-1:0],
              LRU[way_cnt],
              Valid[way_cnt][0],
              // print X's if invalid
              Valid[way_cnt][0] ? Tag[way_cnt][0] : `TAGBITS'hX,
              Valid[way_cnt][1],
              // print X's if invalid
              Valid[way_cnt][1] ? Tag[way_cnt][1] : `TAGBITS'hX
            );
          end
        end
        $display("--- END OF INSTRUCTION CACHE CONTENTS ----\n");
      end

      default: ;  // commands this module doesn't respond to
    endcase
  end

endmodule
```

5.4 DATA_CACHE.v

```
////////////////////////////////////////////////////////////////////////////////
// ECE 485/585: Microprocessor System Design
// Portland State University - Fall 2012
// Final Project:
//
// File:     DATA_CACHE.v (Data Cache)
// Authors: Andy Goetz, Bradon Kanyid, Eric Krause, and Kevin Riedl
// Description: Simulates a read/write data cache.
//
//
//
////////////////////////////////////////////////////////////////////////////////
`define SETS 1024*16
`define WAYS 4
`define SETBITS 14
`define TAGBITS 12

module DATA_CACHE(
  // INPUTS
  input [3:0]  n,        // from trace file
  input [31:0] add_in,   // from trace file
  input clk,

  // OUTPUTS
  output reg [25:0] add_out = 26'bZ, // to next-level cache
  output reg [1:0]  cmd_out = NOP,    // to next-level cache
  output reg [31:0] hit     = 32'b0, // to statistics module
  output reg [31:0] miss    = 32'b0, // to statistics module
  output reg [31:0] reads   = 32'b0, // to statistics module
  output reg [31:0] writes  = 32'b0  // to statistics module
  );

  parameter TRUE        = 1'b1;
  parameter FALSE        = 1'b0;

  // data cache only reponds to following values of n
  parameter RESET        = 4'd8;
  parameter INVALIDATE  = 4'd3;
  parameter READ         = 4'd0;
  parameter WRITE        = 4'd1;
  parameter PRINT        = 4'd9;

  // data cache sends following commands to next-level cache
  parameter READ_OUT     = 2'b01;
  parameter WRITE_OUT    = 2'b10;
  parameter RW_OUT       = 2'b10;  // Read with intent to write
  parameter NOP          = 2'b00;

  //  CACHE ELEMENTS
  // LRU:      6 bits per set.
  reg [5:0]    LRU [`SETS-1:0];
  // Valid:   1 bit per way.  Encoding:  1 = way is valid, 0 = not valid
```

```
reg          Valid [`SETS-1:0] [`WAYS-1:0];
// Tag:      Tag is of size TAGBITS.  One tag per way.
reg [11:0]   Tag   [`SETS-1:0] [`WAYS-1:0];

// loop counters
integer set_cnt,way_cnt;

// internal
reg done = FALSE;
reg [1:0] lru_way;      // temp variable, holds output from decode_lru
reg [5:0] lru_calc_in; // temp variable, holds output from next_lru

// assignments
wire [11:0] curr_tag   = add_in[31:20];
wire [13:0] curr_index = add_in[19:6];

always @(posedge clk)
begin
  add_out = 26'bZ;
  cmd_out = NOP;
  done    = FALSE;

  case(n)
    RESET:
    // clear all Valid bits in the Data Cache and
    // reset the statistics counters
    begin
      hit    = 32'b0;
      miss   = 32'b0;
      reads  = 32'b0;
      writes = 32'b0;

      // for every set
      for (set_cnt = 0; set_cnt < `SETS; set_cnt = set_cnt + 1'b1)
      begin
        LRU[set_cnt] = 6'b0;
     // for all ways
        for (way_cnt = 0; way_cnt < `WAYS; way_cnt = way_cnt + 1'b1)
        begin
          Valid [set_cnt][way_cnt] = FALSE;
          Tag   [set_cnt][way_cnt] = 12'b0;
        end
      end
    end

    INVALIDATE:
    begin
      // when an invalidate command is passed in, check to see if
      // any line in the cache matches the address passed in, if
      // it does, clear the Valid bit for that line.
      for (way_cnt = 0; way_cnt < `WAYS; way_cnt = way_cnt + 1'b1)
      begin
        if (done == FALSE)
      begin
```

```
            if (Tag[curr_index][way_cnt] == curr_tag)
            begin
              Valid[curr_index][way_cnt] = FALSE;
              done = TRUE;
            end
        end
      end
      end

      READ:
      begin
        // increment the number of total reads since reset occurred
        reads = reads + 1'b1;
        // search the ways within the set, if there is a hit, update the LRU
        // and increment the hit counter
        for (way_cnt = 0; way_cnt < `WAYS; way_cnt = way_cnt + 1'b1)
        begin
          if (done == FALSE)
       begin
            if (Tag[curr_index][way_cnt] == curr_tag &&
                Valid[curr_index][way_cnt] == TRUE)
            begin
              lru_calc_in        = next_lru(LRU[curr_index], way_cnt[1:0]);
              LRU[curr_index]    = lru_calc_in;
              hit                = hit + 1'b1;
              done               = TRUE;
            end
       end
        end

        // if there was no hit, increment the miss counter
        if (done == FALSE)
          miss = miss + 1'b1;

        // if there was no hit, check to see if there is an empty
        // line in the set, if not, evict the LRU of the line
        // and replace it with the newly read in value.
        for (way_cnt = 0; way_cnt < `WAYS; way_cnt = way_cnt + 1'b1)
        begin
          if (done == FALSE)
          begin
            if (Valid[curr_index][way_cnt] == FALSE)
            begin
              add_out                     = add_in[31:6];    // generate read
              cmd_out                     = READ_OUT;        // generate read

              lru_calc_in              = next_lru(LRU[curr_index], way_cnt[1:0]);
              LRU[curr_index]             = lru_calc_in;
              Tag[curr_index][way_cnt]    = curr_tag;
              Valid[curr_index][way_cnt]  = TRUE;
              done                        = TRUE;
            end
          end
        end
      if (done == FALSE)
      begin
        add_out                      = add_in[31:6];   // generate read
        cmd_out                      = READ_OUT;       // generate read

        lru_way                      = decode_lru(LRU[curr_index]);
        Tag[curr_index][lru_way]     = curr_tag;
        lru_calc_in                  = next_lru(LRU[curr_index], lru_way);
        LRU[curr_index]              = lru_calc_in;
      end
    end

    WRITE:
    begin
      // increment the number of total writes since reset occurred
      writes = writes + 1;

      // search the ways within the set, if there is a hit, update the LRU
      // and increment the hit counter
      for (way_cnt = 0; way_cnt < `WAYS; way_cnt = way_cnt + 1'b1)
      begin
        if (done == FALSE)
        begin
          if (Tag[curr_index][way_cnt] == curr_tag &&
              Valid[curr_index][way_cnt] == TRUE)
          begin
            // :: already have data  ::

            // :: modify the data ::

            lru_calc_in       = next_lru(LRU[curr_index], way_cnt[1:0]);
            LRU[curr_index]   = lru_calc_in;
            hit               = hit + 1'b1;

            add_out           = add_in[31:6];   // write out to L2
            cmd_out           = WRITE_OUT;      // write out to L2
            done              = TRUE;
          end
        end
      end

      // if there was no hit, increment the miss counter
      if (done == FALSE)
        miss = miss + 1'b1;

      // if there was no hit, check to see if there is an empty
      // line in the set, if not, evict the LRU of the line
      // and replace it with the newly read in value.
      for (way_cnt = 0; way_cnt < `WAYS; way_cnt = way_cnt + 1'b1)
      begin
        if (done == FALSE)
        begin
          if (Valid[curr_index][way_cnt] == FALSE)
```

```
            begin
              add_out = add_in[31:6]; // read data w/ intent to mod
              cmd_out = RW_OUT;       // read data w/ intent to mod

              // :: modify the data ::

              lru_calc_in                = next_lru(LRU[curr_index], way_cnt[1:0]);
              LRU[curr_index]                 = lru_calc_in;
              Tag[curr_index][way_cnt]        = curr_tag;
              Valid[curr_index][way_cnt]      = TRUE;
              done                            = TRUE;

              add_out                         = add_in[31:6]; // write out to L2
              cmd_out                         = WRITE_OUT;    // write out to L2
            end
          end
        end

        if (done == FALSE)
        begin
          add_out                        = add_in[31:6]; // read in w/ intent to mod
          cmd_out                        = RW_OUT;       // read in w/ intent to mod

          // :: modify the data ::

          lru_way                        = decode_lru(LRU[curr_index]);
          Tag[curr_index][lru_way]       = curr_tag;
          Valid[curr_index][lru_way]     = TRUE;
          lru_calc_in                    = next_lru(LRU[curr_index], lru_way);
          LRU[curr_index]                = lru_calc_in;

          add_out                        = add_in[31:6]; // write out to L2
          cmd_out                        = WRITE_OUT;    // write out to L2
        end
      end

      // Print all of the contents of the Data Cache
      PRINT:
      begin
        // print header
        $display("----------- DATA CACHE CONTENTS ----------");
  $display(" INDEX | LRU | V[0]|Tag[0]| V[1]|Tag[1]| V[2]|Tag[2]| V[3]|Tag[3]");
        // cycle through all of the ways within a set
        for (set_cnt = 0; set_cnt < `SETS; set_cnt = set_cnt+1)
        begin
          // print out the whole set if there are any valid lines
          if (Valid[set_cnt][3] | Valid[set_cnt][2] |
              Valid[set_cnt][1] | Valid[set_cnt][0] )
          begin
$display(" %4h  |  %d  |  %d  | %3h  |  %d  | %3h  |  %d  | %3h  |  %d
| %3h",
              set_cnt[`SETBITS-1:0],
              decode_lru(LRU[set_cnt]),
              Valid[set_cnt][0],
              Valid[set_cnt][0] ? Tag[set_cnt][0] : `TAGBITS'hX,
              Valid[set_cnt][1],
              Valid[set_cnt][1] ? Tag[set_cnt][1] : `TAGBITS'hX,
              Valid[set_cnt][2],
              Valid[set_cnt][2] ? Tag[set_cnt][2] : `TAGBITS'hX,
              Valid[set_cnt][3],
              Valid[set_cnt][3] ? Tag[set_cnt][3] : `TAGBITS'hX
            );
          end
        end
        $display("-------- END OF DATA CACHE CONTENTS --------");
      end

      default: ;  // commands this module doesn't respond to
    endcase
  end

  function [1:0] decode_lru;
  input [5:0] lru_bits;
    begin
        if      (!(|lru_bits[5:3]))  decode_lru = 2'd0;
        else if (!(|lru_bits[2:1]))  decode_lru = 2'd1;
        else if (!  lru_bits[0])     decode_lru = 2'd2;
        else                         decode_lru = 2'd3;
    end
  endfunction

  function [5:0] next_lru;
    input  [5:0] lru_bits;
    input  [1:0] way_accessed;
    begin
      case (way_accessed)
      // Set the first 3 bits (this defines MRU 0)
      2'd0: next_lru = (lru_bits | 6'b111000);

      // Clear bit 0, Set bits 3 & 4 (MRU 1)
      2'd1: next_lru = ((lru_bits & 6'b011111) | 6'b000110);

      // Clear bits 1 & 3, Set bit 5 (MRU 2)
      2'd2: next_lru = ((lru_bits & 6'b101011) | 6'b000001);

      // Clear bits 2,4,5 (MRU 3)
      2'd3: next_lru = (lru_bits & 6'b110100);
      endcase
    end
  endfunction

endmodule


//this is not a browser cache
```

5.5 STATS.v

```
////////////////////////////////////////////////////////////////////////////////
// ECE 485/585: Microprocessor System Design
// Portland State University - Fall 2012
// Final Project:
//
// File:     STATS.v
// Authors: Andy Goetz, Bradon Kanyid, Eric Krause, and Kevin Riedl
// Description: Generates statistics such as hit ratio about cache.
//
////////////////////////////////////////////////////////////////////////////////
module STATS(
  // INPUTS
  input print, // mux to determine reads/writes
  input [31:0] ins_reads,
  input [31:0] ins_hit,
  input [31:0] ins_miss,

  input [31:0] data_reads,
  input [31:0] data_writes,
  input [31:0] data_hit,
  input [31:0] data_miss
    );

  always @(posedge print)
  begin
    $display(" STATISTICS: ");
    $display(" Hits   = %d", data_hit + ins_hit);
    $display(" Miss   = %d", data_miss + ins_miss);
    $display(" Reads  = %d", data_reads + ins_reads);
    $display(" Writes = %d", data_writes);
    $display(" Hit Ratio = %.1f%%", (data_reads + ins_reads + data_writes) == 0 ?
      0 : 100.0*(data_hit + ins_hit)/(data_reads + ins_reads + data_writes));
  end
endmodule
```

6 Testbench Output

The output of our tests is reproduced below.

```
Cached_Instr_Reads.trace: (PASS)
        Multiple reads to the same instruction cache line

-------- INSTRUCTION CACHE CONTENTS --------
 Index  | LRU  | V[0]|Tag[0]| V[1]|Tag[1]
 0000   |  1   |  1  | 000  |  0  | xxx
--- END OF INSTRUCTION CACHE CONTENTS ----

------------ DATA CACHE CONTENTS -----------
 INDEX  | LRU  | V[0]|Tag[0]| V[1]|Tag[1]| V[2]|Tag[2]| V[3]|Tag[3]
-------- END OF DATA CACHE CONTENTS --------
 STATISTICS:
 Hits   =          3
 Miss   =          1
 Reads  =          4
 Writes =          0
 Hit Ratio = 75.0%

Cached_Data_Reads.trace: (PASS)
        Multiple reads to the same data cache line

-------- INSTRUCTION CACHE CONTENTS --------
 Index  | LRU  | V[0]|Tag[0]| V[1]|Tag[1]
--- END OF INSTRUCTION CACHE CONTENTS ----

------------ DATA CACHE CONTENTS -----------
 INDEX  | LRU  | V[0]|Tag[0]| V[1]|Tag[1]| V[2]|Tag[2]| V[3]|Tag[3]
 0000   |  1   |  1  | 000  |  0  | xxx  |  0  | xxx  |  0  | xxx
-------- END OF DATA CACHE CONTENTS --------
 STATISTICS:
 Hits   =          3
 Miss   =          1
 Reads  =          4
 Writes =          0
 Hit Ratio = 75.0%

Interleaved_Read_Write.trace: (PASS)
        Reads and Writes to the data cache

-------- INSTRUCTION CACHE CONTENTS --------
 Index  | LRU  | V[0]|Tag[0]| V[1]|Tag[1]
--- END OF INSTRUCTION CACHE CONTENTS ----

------------ DATA CACHE CONTENTS -----------
 INDEX  | LRU  | V[0]|Tag[0]| V[1]|Tag[1]| V[2]|Tag[2]| V[3]|Tag[3]
 0000   |  1   |  1  | 000  |  0  | xxx  |  0  | xxx  |  0  | xxx
-------- END OF DATA CACHE CONTENTS --------
 STATISTICS:
 Hits   =          3
 Miss   =          1
 Reads  =          2
 Writes =            2
 Hit Ratio = 75.0%

Same_Set_Instr.trace: (PASS)
        Multiple reads from the same set in the instruction cache.

-------- INSTRUCTION CACHE CONTENTS --------
 Index | LRU | V[0]|Tag[0]| V[1]|Tag[1]
 0000  |  0  |  1  | 000  |  1  | 001
--- END OF INSTRUCTION CACHE CONTENTS ----

------------ DATA CACHE CONTENTS -----------
 INDEX | LRU | V[0]|Tag[0]| V[1]|Tag[1]| V[2]|Tag[2]| V[3]|Tag[3]
-------- END OF DATA CACHE CONTENTS --------
 STATISTICS:
 Hits   =            0
 Miss   =            2
 Reads  =            2
 Writes =            0
 Hit Ratio = 0.0%

Same_Set_Data.trace: (PASS)
        Multiple reads from the same set in the data cache.

-------- INSTRUCTION CACHE CONTENTS --------
 Index | LRU | V[0]|Tag[0]| V[1]|Tag[1]
--- END OF INSTRUCTION CACHE CONTENTS ----

------------ DATA CACHE CONTENTS -----------
 INDEX | LRU | V[0]|Tag[0]| V[1]|Tag[1]| V[2]|Tag[2]| V[3]|Tag[3]
 0000  |  0  |  1  | 000  |  1  | 001  |  1  | 002  |  1  | 003
-------- END OF DATA CACHE CONTENTS --------
 STATISTICS:
 Hits   =            0
 Miss   =            4
 Reads  =            2
 Writes =            2
 Hit Ratio = 0.0%

Instr_Conflict.trace: (PASS)
        Enough reads to the same instruction cache set to cause an eviction.

-------- INSTRUCTION CACHE CONTENTS --------
 Index | LRU | V[0]|Tag[0]| V[1]|Tag[1]
 0000  |  1  |  1  | 200  |  1  | 100
--- END OF INSTRUCTION CACHE CONTENTS ----

------------ DATA CACHE CONTENTS -----------
 INDEX | LRU | V[0]|Tag[0]| V[1]|Tag[1]| V[2]|Tag[2]| V[3]|Tag[3]
-------- END OF DATA CACHE CONTENTS --------
 STATISTICS:
 Hits   =            0
 Miss   =            3
 Reads  =            3
 Writes =           0
 Hit Ratio = 0.0%

Data_Conflict.trace (PASS)
        Enough reads and writes to the same instruction cache set to
        cause an eviction

  ———————— DATA CACHE CONTENTS ———————
   INDEX | LRU | V[0]|Tag[0]| V[1]|Tag[1]| V[2]|Tag[2]| V[3]|Tag[3]
   0000  |  1  |  1  | 500  |  1  | 200  |  1  | 300  |  1  | 400
  ———— END OF DATA CACHE CONTENTS ————
   STATISTICS:
   Hits   =           0
   Miss   =           5
   Reads  =           3
   Writes =           2
   Hit Ratio = 0.0%

Instr_Invalidate.trace (PASS)
        Multiple Reads Followed by Invalidate clears single line.

  ———— INSTRUCTION CACHE CONTENTS ————
   Index | LRU | V[0]|Tag[0]| V[1]|Tag[1]
   0000  |  0  |  1  | 001  |  1  | 002
  —— END OF INSTRUCTION CACHE CONTENTS ——

  ———— INSTRUCTION CACHE CONTENTS ————
   Index | LRU | V[0]|Tag[0]| V[1]|Tag[1]
   0000  |  0  |  0  | xxx  |  1  | 002
  —— END OF INSTRUCTION CACHE CONTENTS ——

   STATISTICS:
   Hits   =           0
   Miss   =           2
   Reads  =           2
   Writes =           0
   Hit Ratio = 0.0%

Data_Invalidate.trace (PASS)
        Multiple Reads Followed by Invalidate clears single line.

  ———————— DATA CACHE CONTENTS ———————
   INDEX | LRU | V[0]|Tag[0]| V[1]|Tag[1]| V[2]|Tag[2]| V[3]|Tag[3]
   0000  |  0  |  1  | 001  |  1  | 002  |  1  | 003  |  1  | 004
  ———— END OF DATA CACHE CONTENTS ————

  ———————— DATA CACHE CONTENTS ———————
   INDEX | LRU | V[0]|Tag[0]| V[1]|Tag[1]| V[2]|Tag[2]| V[3]|Tag[3]
   0000  |  0  |  1  | 001  |  0  | xxx  |  1  | 003  |  1  | 004
  ———— END OF DATA CACHE CONTENTS ————
   STATISTICS:
   Hits   =           0
   Miss   =           4
   Reads  =           4
   Writes =            0
   Hit Ratio = 0.0%

Instr_Clear.trace (PASS)
        Read Followed by Clear empties data cache

  ——————— INSTRUCTION CACHE CONTENTS ———————
   Index | LRU | V[0]|Tag[0]| V[1]|Tag[1]
   048c  |  1  |  1  | 001  |  0  | xxx
  ——— END OF INSTRUCTION CACHE CONTENTS ————

  ——————— INSTRUCTION CACHE CONTENTS ———————
   Index | LRU | V[0]|Tag[0]| V[1]|Tag[1]
  ——— END OF INSTRUCTION CACHE CONTENTS ————

   STATISTICS:
   Hits   =            0
   Miss   =            0
   Reads  =            0
   Writes =            0
   Hit Ratio = 0.0%

Data_Clear.trace (PASS)
        Read followed by Clear empties data cache

  ——————————— DATA CACHE CONTENTS ——————————
   INDEX | LRU | V[0]|Tag[0]| V[1]|Tag[1]| V[2]|Tag[2]| V[3]|Tag[3]
   048c  |  1  |  1  | 001  |  0  | xxx  |  0  | xxx  |  0  | xxx
  ——————— END OF DATA CACHE CONTENTS ———————

  ——————————— DATA CACHE CONTENTS ——————————
   INDEX | LRU | V[0]|Tag[0]| V[1]|Tag[1]| V[2]|Tag[2]| V[3]|Tag[3]
  ——————— END OF DATA CACHE CONTENTS ———————
   STATISTICS:
   Hits   =            0
   Miss   =            0
   Reads  =            0
   Writes =            0
   Hit Ratio = 0.0%

Instr_Invalidate_Read.trace: (PASS)
        Read, Invalidate, Read uses cleared way as LRU
  ——————— INSTRUCTION CACHE CONTENTS ———————
   Index | LRU | V[0]|Tag[0]| V[1]|Tag[1]
   0400  |  0  |  1  | 001  |  1  | 002
  ——— END OF INSTRUCTION CACHE CONTENTS ————

  ——————— INSTRUCTION CACHE CONTENTS ———————
   Index | LRU | V[0]|Tag[0]| V[1]|Tag[1]
   0400  |  0  |  1  | 001  |  1  | 003
  ——— END OF INSTRUCTION CACHE CONTENTS ————

   STATISTICS:
   Hits   =            0
```

```
    Miss   =          3
    Reads  =          3
    Writes =          0
    Hit Ratio = 0.0%

Data_Invalidate_Read.trace: (PASS)
          Read, Invalidate, Read uses cleared way as LRU.

  ------------ DATA CACHE CONTENTS -----------
    INDEX | LRU | V[0]|Tag[0]| V[1]|Tag[1]| V[2]|Tag[2]| V[3]|Tag[3]
    0400  |  0  |  1  | 001  |  1  | 002  |  1  | 003  |  1  | 004
  -------- END OF DATA CACHE CONTENTS --------

  ------------ DATA CACHE CONTENTS -----------
    INDEX | LRU | V[0]|Tag[0]| V[1]|Tag[1]| V[2]|Tag[2]| V[3]|Tag[3]
    0400  |  0  |  1  | 001  |  1  | 002  |  1  | 005  |  1  | 004
  -------- END OF DATA CACHE CONTENTS --------
    STATISTICS:
    Hits   =          0
    Miss   =          5
    Reads  =          5
    Writes =          0
    Hit Ratio = 0.0%
```