**ECE485/585 - Cache Simulator**

**Sean Koppenhafer, Luis Santiago, J.S. Peirce**

**Basic Specifications**

To start off, we would like to lay out some basic specifications that define the cache described in the project description:

Cache lines = 128k

Sets = 8k

Address bits = 32

Byte select bits = 6

Index select bits = 13

Tag select bits = 13

However, because our cache simulator asks the user for the number of sets, the cache line size in bytes, and the number of ways in each set, these values can change. The above values are only true when using 64 byte lines, 16 ways per set, and 8k total sets.

**Assumptions**

1. Our cache must keep our pseudo LRU bits and our MESIF state consistent with the higher level caches.
2. A read to the L1 instruction cache means the same thing as a read to the L1 data cache for our cache.
3. A read or a write to the L1 cache updates our cache state (tag bits, MESIF state, pseudo LRU bits).
4. Snooping a write should only occur in the invalid state or when another processors cache is evicting a dirty line.
5. Addresses are passed to L2 and other caches without clearing the byte offset bits. Assumes those devices clear the bits if needed.

**Design Decisions**

A major design decision revolved around how the input trace file would be handled. In the end, we decided to write a Perl script that would pre-process the trace file and output a formatted trace file. The Perl script looks for lines with these two formats at the beginning of the line:

n address

n

If a line with the format of *n address* is found, the Perl script prints out the line to the formatted trace file with only one space between the trace operation and the address. If only *n* is found, the script writes the trace operation to the output file, appending a dummy address of 0 to match the format of *n address.* This ensures that our cache simulator can always read a trace file – even if it has comments!

Another major design decision revolved around allowing the user to decide the size of the cache. At the beginning of the simulation, the simulator will prompt the user for the number of sets (must be a power of 2), the line size in bytes (must be a power of 2), and the number of ways in a set (must be a power of 2 less than 128). The line size and the number of ways must be a power of 2 so that the address can be cleanly divided up in the byte select, index and tag bits. The number of ways in set must be less than 128 because our pseudo LRU algorithm can only handle 64 ways in a set. This is because the largest data type available is a uint64\_t. To go into sets with more ways, we would have had to implement a complicated algorithm that involved using an array of uint64\_t variables. We instead decided to constrain the number of ways.

We also decided to implement the posted write buffer for extra credit. Our posted write buffer can hold 10 items total before it becomes full and must write a line back to DRAM. Every 5 trace operations, the posted write buffer writes the oldest item in the buffer back to DRAM. If a cache miss occurs while executing a trace operation, our simulator will first check the posted write buffer for the data before going out to DRAM. If a cache eviction occurs on a dirty cache line, our simulator will first check for a previous write to the same cache line in the write buffer. If the cache line is found in the write buffer, the old write will be discarded and the new write will be added into the write buffer. Also, on cache clear, the write buffer is also cleared by simulating writing all the items in the buffer back to DRAM.

Pseudo LRU (P-LRU) models a balanced binary tree where the root of the tree is bit 0. We approach this as two different parts updating the P-LRU and finding the least recently used. Finding last recently used consists of reading the P\_LRU bits beginning at bit 0. This provides the direction using the P-LRU scheme. A 1 indicates right and 0 indicates left. To find the next bit to read or write we used the value of the current bit index beginning at 0 and apply bit index times 2 plus 2 (bit# \* 2 + 2) when going right and bit index times 2 plus 1 (bit# \* 2 + 1) when going left. We began with a variable named victim set at 0 and at every level we left shift “victim” by one bit and make bit 0 of victim equal to the bit just read that determined the direction on the P-LRU.

Updating the P-LRU was implemented by computing ( initial value of root is 0 an n=1) and traversing the bits with the same approach as finding the victim. Once root is calculated we compare with the variable “tag\_matched\_line” . If tag\_matched\_line is greater than or equal to root, we clear our current bit on P-LRU or if it is less than we set the current bit indicating that LRU must be in opposite direction of my current traversal. The process is repeated using the next bit based on direction taken with a new root of the tree, increasing n by 1 at each level of the tree. Then root is updated by if we go right or if we go left. Process repeats splitting into smaller trees until the last bottom level is reached.

**Verification**

To verify that our simulator functioned correctly, we used a few different testing techniques. Our first testing technique involved unit testing. For each sub-system of the cache (MESIF, pseudo LRU), we created a dummy main function that would test only it's own sub-module. Within the sub-modules, we wrote a battery of tests that would be called from the main function. The tests were designed to be as exhaustive as possible. For example, in the MESIF module, we went into every MESIF state and made sure that the correct transition happened when a trace operation occurred. To avoid having to check the output by hand, we used some extra debug variables and the assert library to make sure our output was correct for the given situation. If a test failed, the simulator would fail out and tell us the exact line that the failure occurred at.

Once all of the unit tests were passing for each sub-module, we hooked the trace decode logic up to the MESIF and pseudo LRU sub-modules. We then re-wrote all of the unit tests for each sub-module into separate trace files. By running through the units tests again in trace file form, we were able to identify bugs that were occurring in the decode logic, since the sub-modules themselves were proven correct in our previous tests.

Once the unit tests in trace form were passing, we created some new trace tests. First, we wrote a test that would test to see if the cache properly handled reads/writes along with hits/misses. If the number of reads/write and hits/misses didn't match what we expected at the end of the trace, we knew that there was an error.

We wrote two tests for our posted write buffer. The first test evicts a non-dirty line, confirming that the write buffer does not receive the non-dirty line. Then a dirty line is evicted and the write buffer is printed, showing the dirty line in the buffer. We then do a read to the same address as the evicted line to make sure that the cache reads from the write buffer and not DRAM. Finally, 3 more reads to the same address occur and then a print to confirm that the write buffer writes the line back to DRAM after 5 trace operations. Our second test checks to see if the write buffer successfully removes a currently present line when a new dirty line is written to the same address. The combination of these two tests should check all of the corner cases of the write buffers behavior.

**Test Traces**

**Hit\_miss.txt**

//Assumes:

// 8k sets

// 16 ways

// 64 byte lines

//All operations to first index (index 0)

//End of trace stats should print out:

//18 Reads

//7 Writes

//20 Hits

//5 Misses

//0.8 hit ratio

//Line 0

0 0 //Read, miss

0 0 //Read, hit

2 0 //Read, hit

2 0 //Read, hit

1 0 //Write, hit

//Line 1

2 80000 //Read, miss

2 80000 //Read, hit

0 80000 //Read, hit

0 80000 //Read, hit

1 80000 //Write, hit

//Line 2

1 100000 //Write, miss

1 100000 //Write, hit

1 100000 //Write, hit

0 100000 //Read, hit

0 100000 //Read, hit

//Line 3

0 180000 //Read, miss

0 180000 //Read, hit

2 180000 //Read, hit

2 180000 //Read, hit

1 180000 //Write, hit

//Line 4

0 200000 //Read, miss

0 200000 //Read, hit

2 200000 //Read, hit

2 200000 //Read, hit

1 200000 //Write, hit

9 //Print the cache

**Hit\_miss2.txt**

//Assumes:

// 8k sets

// 2 ways

// 64 byte lines

//Checks hit/misses when an eviction happens

//End of trace stats:

//16 Reads

//6 Writes

//17 Hits

//5 Misses

//0.773 hit ratio

//Index 0, Line 0

0 0 //Read, miss

0 0 //Read, hit

2 0 //Read, hit

2 0 //Read, hit

1 0 //Write, hit

//Index 0, Line 1

2 80000 //Read, miss

2 80000 //Read, hit

0 80000 //Read, hit

0 80000 //Read, hit

1 80000 //Write, hit

//Evict a line

1 100000 //Write, miss

1 100000 //Write, hit

1 100000 //Write, hit

0 100000 //Read, hit

0 100000 //Read, hit

//Read in evicted line

0 0 //Read, miss

0 0 //Read, hit

2 0 //Read, hit

2 0 //Read, hit

1 0 //Write, hit

//Read to different index

0 800060 //Read miss

2 800060 //Read hit

**MESIF\_E.txt**

// Assumes:

// 8k sets

// 16 ways

// 64 byte lines

// test exhaustive exclusive MESIF State

0 FFFFFFF8 //Read, I->E

0 FFFFFFF8 //Read, NOHIT

0 FFFFFFF7 //Read, HIT

2 FFFFFFF9 //Read, HITM

5 FFFFFFF9 //Snoop write

1 FFFFFFF9 //Write, E->M

3 FFFFFFF9 //Snoop Inv, M->I

0 FFFFFFF8 //Read, I->E

4 FFFFFFF0 //Snoop read, E->S

3 FFFFFFF0 //Snoop inv, S->I

0 FFFFFFF8 //Read, I->E

3 FFFFFFF8 //Snoop inv, E->I

0 FFFFFFF8 //Read, I->E

6 FFFFFFF8 //Snoop RWIM, E->I

**MESIF\_F.txt**

// Assumes:

// 8k sets

// 16 ways

// 64 byte lines

// test exhaustive Forward MESIF State

0 524F6201 //Read, I->F

2 524F6200 //Read, NOHIT

0 524F6201 //Read, HIT

0 524F6202 //Read, HITM

1 524F6200 //Write, F->M

3 524F6200 //Snoop inv, M->I

2 524F6202 //Read, HITM, I->F

// Next line should never happened but testing

// it won’t change

5 524F6200 //Snoop write

4 524F6200 //Snoop read, F->S

3 524F6202 //Snoop inv, S->I

2 524F6202 //Read, I->F

3 524F6202 //Snoop inv, F->I

2 524F6202 //Read, I->F

6 524F6202 //Snoop RWIM, F->I

**MESIF\_I.txt**

// Assumes:

// 8k sets

// 16 ways

// 64 byte lines

// test exhaustive invalid MESIF State

3 02 //Snoop INV

4 02 //Snoop read

5 02 //Snoop write

6 02 //Snoop RWIM

**MESIF\_M.txt**

// Assumes:

// 8k sets

// 16 ways

// 64 byte lines

// test exhaustive Modified MESIF State

1 1640 //Write, I->M

1 1640 //Write

0 1640 //Read

2 1640 //Read

5 1640 //Snoop Write

4 1640 //Snoop Read, M->S

3 1640 //Snoop INV, S->I

1 1640 //Write, I->M

6 1640 //Snoop RWIM, M->I

1 1641 //Write, I->M

3 1641 //Snoop INV, M->I

**MESIF\_S.txt**

// Assumes:

// 8k sets

// 16 ways

// 64 byte lines

// test exhaustive Shared MESIF State

0 00205001 //Read, I->F

4 00205000 //Snoop READ, F->S

2 00205000 //Read

0 00205001 //Read

4 00205000 //Snoop Read

5 00205000 //Snoop write

1 00205000 //Write, S->M

3 00205000 //Snoop inv, M->I

0 00205001 //Read, I->F

4 00205000 //Snoop READ, F->S

3 00205000 //Snoop inv, S->I

0 00205001 //Read, I->F

4 00205000 //Snoop READ, F->S

6 00205000 //Snoop RWIM, S->I

**Simple\_LRU.txt**

//Assume:

// 8k sets

// 16 ways

// 64 byte lines

//This trace fills up all 16 indexes in one sets.

//Single access in order to test basic operation.

//Then it goes through and evicts all of the lines one at a time

//If correct, all cache references should be misses.

//Fill all 16 ways in the set

0 800000

2 1000000

0 1800000

2 2000000

0 2800000

2 3000000

0 3800000

2 4000000

0 4800000

2 5000000

0 5800000

2 6000000

0 6800000

2 7000000

0 7800000

2 8000000

//Evict 4 lines. Should evict 0, 8, 4, 12 in order

0 8800000

2 9000000

0 9800000

2 10000000

//Try to access the 4 evicted lines again. Should evict 2, 10, 6, 14

0 800000

0 2800000

0 4800000

0 6800000

//Try to access the 4 newly evicted lines. Should evict 1, 9, 5, 13

0 1800000

0 3800000

0 5800000

0 7800000

//Finally, try to access the last 4 evicted lines. Should evict 3, 11, 7, 15

2 1000000

2 3000000

2 5000000

2 7000000

**Simple\_LRU2.txt**

//Assume:

// 2 sets

// 4 ways

// 64 byte lines

//Byte select = 6 bits, index = 1 bits, tag = 26 bits

//Checks to see what happens when other caches invalidate a few lines in a set

//Before evicting something

//Fill 3 lines

0 0 //Read, NOHIT

1 81 //Write, HIT

2 101 //Read, HIT

//Invalidate 0 and 2

3 0 //Snoop invalidate

3 101 //Snoop invalidate

//Fill the set

0 3200

0 3300

0 3400

//Line 1 and then 2 should be evicted

1 3500

0 3602

9

**Simple\_LRU3.txt**

//Assume:

// 2 sets

// 16 ways

// 64 byte lines

// The trace fills up 8 indexes in a set in order.

// Then access line 0 and then later on line 4

// then it continues to fill the the cache then

// begins to evict lines

// In index 0

0 800000

2 1000000

0 1800000

2 2000000

0 2800000

2 3000000

0 3800000

2 4000000

// fills half way (from 0-7) and then access line 0

0 800000

// add 3 (from 8-10)more new lines

0 4800000

2 5000000

0 5800000

// reads line 4

0 2800000

// continue to fill the cache (from 11-15)

2 6000000

0 6800000

2 7000000

0 7800000

2 8000000

//Evict 4 lines. Should evict 2, 8, 6, 12 in order

0 8800000

2 9000000

0 9800000

2 10000000

// used the evicted lines to evict next

// should evict 1,10,5,14

0 1800000

0 4800000

0 3800000

0 6800000

// should evict 3, 9, 7, 13

0 18000000

2 3000000

0 5800000

0 7800000

// read 0 then should evict 11

0 800000

2 6000000

**Simple\_LRU4.txt**

//Assume:

// 2 sets

// 8 ways

// 64 byte lines

//

// Trace is designed to test a 8 way set

// The trace fills up 8 indexes in a set in order.

// Then access first 7 lines already in the set.

// Evict lines to ensure LRU works correctly

// fills cache

0 800000

2 1000000

0 1800000

2 2000000

0 2800000

2 3000000

0 3800000

2 4000000

// reads 0-6 leaving 7 as true Last recently used

// ensure line are updating correctly after initial store

0 800000

2 1000000

0 1800000

2 2000000

0 2800000

2 3000000

0 3800000

// evicts 0,4,2,7,1,5,3,6

0 4800000

2 5000000

0 5800000

2 6000000

0 6800000

2 7000000

0 7800000

2 8000000

**Simple\_LRU5.txt**

//Assume:

// less than 32k sets

// 16 ways

// 64 byte lines

//Simple test to ensure correct update of LRU bits.

//Fills the cache, then read all the even lines.

// That should make initial evictions of all odds.

// then evict evens

0 800000

2 1000000

0 1800000

2 2000000

0 2800000

2 3000000

0 3800000

2 4000000

0 4800000

2 5000000

0 5800000

2 6000000

0 6800000

2 7000000

0 7800000

2 8000000

// read even 0,2,4,6,8,10,12,14

0 800000

0 1800000

0 2800000

0 3800000

0 4800000

0 5800000

0 6800000

0 7800000

// should evict odd 1,9,5,13,3,11,7,15

0 8800000

0 9000000

0 A000000

0 B000000

2 c000000

1 d000000

0 e000000

2 f000000

// should evict even 0,8,4,12,2,10,6,14

2 1000000

2 2000000

2 3000000

2 4000000

2 5000000

2 6000000

2 7000000

2 8000000

**fill\_write\_buffer.txt**

// Assumes:

// 2 sets

// 2 ways

// 64 byte lines

// test write buffer that holds one item at the time.

// \*\*\*\*\*\*\* Don’t forget to modify code to ensure write buffer is size 1\*\*\*\*\*\*\*\*\*\*\*\*\*

//Write two dirty lines to evict later

1 1001 //Write

1 200A //Write

//Evict the first dirty line and print the write buffer

0 3004 //Read, NOHIT

9 //Print cache and write buffer

//Evict the second dirty line. The write buffer should write the previous

//line back to DRAM because it is full, and then put in the newly evicted line.

2 4008 //Read, NOHIT

9 //Print cache and write buffer

**overwrite\_write\_in\_WB.txt**

//Assume:

// 2 sets

// 2 ways

// 64 byte lines

//

//This test evicts a dirty cache line into the write buffer

//Then it evicts the same dirty line again

//Tests to see if the write buffer only holds the newest value

//Fills both ways in a Set

1 1001 //Write

0 200A //Read, HIT

//Evict the dirty line( line 0). Then print write buffer.

0 3004 //Read, NOHIT

9 //Print cache and write buffer

//Write to the same address that is in the write buffer. That evicts line 1

//Then next 2 reads evicts line 0 and 1.

//The second read evicts the same dirty line as before which overrides the old dirty

//line already in the posted write buffer.

1 1001 //Write

0 400A //Read, HIT

0 500A //Read, HIT

9 //Print cache and write buffer

9 //Print cache and write buffer

**write\_DRAM\_write\_buffer.txt**

//Assume:

// 2 sets

// 2 ways

// 64 byte lines

//

//This test evicts a dirty cache line into the write buffer

//Then it evicts the same dirty line again

//Tests to see if the write buffer only holds the newest value

//Fills both ways in a Set

1 1001 //Write

0 200A //Read, HIT

//Evict the dirty line( line 0). Then print write buffer.

0 3004 //Read, NOHIT

9 //Print cache and write buffer

//Write to the same address that is in the write buffer. That evicts line 1

//Then next 2 reads evicts line 0 and 1.

//The second read evicts the same dirty line as before which overrides the old dirty

//line already in the posted write buffer.

1 1001 //Write

0 400A //Read, HIT

0 500A //Read, HIT

9 //Print cache and write buffer

9 //Print cache and write buffer

**Source code**

**f\_proj.c**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* J.S. PEIRCE jpei2@pdx.edu

\* SEAN KOPPENHAFER koppen2@pdx.edu

\* LUIS SANTIAGO lsantiag@pdx.edu

\*

\* 5 NOVEMBER 2014

\* ECE485 FINAL PROJECT

\* CACHE DESIGN SIMULATION PROGRAM

\*

\* -MESIF COHERENCE PROTOCOL

\* -PSEUDO LRU

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "MESIF.h"

#include "cache.h"

#include "pseudolru.h"

/\* Defines for write buffer \*/

#define WB\_SIZE 10 //Defines the size of the posted write buffer

#define WB\_COUNT 5 //Number of traces before the write buffer writes to memory

#define SLOT\_VALID 1

#define SLOT\_INVALID 0

/\*TRACE RESULTS:\*/

#define READ\_DATA\_L1 0

#define WRITE\_DATA\_L1 1

#define READ\_INS\_L1 2

#define SNOOP\_INV 3

#define SNOOP\_READ 4

#define SNOOP\_WRITE 5

#define SNOOP\_RWIM 6

#define CLEAR\_RESET 8

#define PRINT\_STATES 9

/\*TRACES: (N ADDRESS)

\*0 READ REQUEST FROM L1 FROM D CACHE

\*1 WRITE REQUEST FROM L1 FROM D CACHE

\*2 READ REQUSET FROM L1 INSTRUCTION CACHE

\*3 SNOOPED INVCALIDATE COMMAND

\*4 SNOOPED READ REQUEST

\*5 SNOOPED WRITE REQUEST

\*6 SNOOPED RFO

\*8 CLEAR CACHE AND RESET STATE (FLUSH OR EOF)

\*9 PRINT CONTENTS AND STATE OF EACH VALID LINE

\*/

typedef struct {

uint8\_t operation;

uint32\_t address;

} trace\_line;

typedef struct {

uint32\_t buffer\_slot[WB\_SIZE];

uint8\_t slot\_valid[WB\_SIZE];

uint8\_t head, tail, full;

uint8\_t trace\_counter;

uint8\_t tag\_index; /\* The index of the most recenetly matched tag \*/

uint32\_t truncate\_mask;

} posted\_write\_buffer;

/\* Trace related functions \*/

void decode\_trace(uint8\_t trace\_op, uint32\_t address);

uint8\_t check\_tags(uint32\_t tag, cache\_set\* set);

void init\_cache(void);

void reset\_cache(void);

void print\_cache(void);

void print\_statistics(void);

void cleanup(FILE\*);

/\* CPU reads and writes \*/

void L1\_read\_or\_write(uint8\_t CPU\_op, uint32\_t tag, uint32\_t address, uint8\_t tag\_matched\_line, cache\_set\* indexed\_set);

void fill\_invalid\_line(uint8\_t CPU\_op, uint32\_t tag, uint32\_t address, cache\_set\* set);

void fill\_valid\_line(uint8\_t CPU\_op, uint32\_t tag, uint32\_t address, cache\_set\* set);

/\* Functions that break down the address \*/

uint32\_t extract\_byte\_select(uint32\_t address);

uint32\_t extract\_index(uint32\_t address);

uint32\_t extract\_tag(uint32\_t address);

/\* Posted Write Buffer \*/

posted\_write\_buffer write\_buffer;

void add\_to\_write\_buffer(uint32\_t address);

uint8\_t search\_write\_buffer(uint32\_t address);

void writeback\_line\_write\_buffer(void);

void clear\_write\_buffer(void);

/\* Global values \*/

cache\_set \*sets;

uint32\_t NUM\_SETS;

uint32\_t WAYS;

uint32\_t BYTES\_PER\_LINE;

uint32\_t TAG\_BITS;

uint32\_t INDEX\_BITS;

uint32\_t BYTE\_SELECT\_BITS;

/\* Cache statistics \*/

static uint32\_t cache\_reads;

static uint32\_t cache\_writes;

static uint32\_t cache\_hits;

static uint32\_t cache\_misses;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* MAIN LOOP:

\* -GET NEXT LINE OF TRACE FILE

\* -SPLIT LINE INTO PARTS

\* -COMPARE TAG

\* -UPDATE STATUS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (int argc, char \* argv[])

{

const uint8\_t buffer\_len = 20;

FILE \*trace\_file;

trace\_line trace;

time\_t t;

srand(time(&t));

char buffer[buffer\_len];

char \*operation\_ptr, \*address\_ptr;

char operation\_text[30];

uint32\_t inter\_ways, inter\_sets;

/\* Get cache specifications from user \*/

printf("In the following prompts, all non-powers of two will be rounded down to the nearest power of two.\n");

printf("For example, 31 will round down to 16.\n");

printf("1 is not an acceptable input. Will cause an error in the cache simulator.\n\n");

printf("Enter the number of sets in the cache (must be a power of 2): ");

scanf("%u", &NUM\_SETS);

printf("Enter the number of ways in the cache (must be power of 2 less than 128): ");

scanf("%u", &WAYS);

printf("Enter the number of bytes in a line (must be a power of 2): ");

scanf("%u", &BYTES\_PER\_LINE);

if( NUM\_SETS < 2 || WAYS < 2 || BYTES\_PER\_LINE < 2) {

printf("1 is not an acceptable input value. Exiting.\n");

exit(-3);

}

if(WAYS > 64) {

printf("Maximum number of allowed ways is 64. Exiting.\n");

exit(-4);

}

/\* Round down the number of ways and sets if not a power of 2 \*/

inter\_ways = log(WAYS) / log(2);

WAYS = pow(2, inter\_ways);

inter\_sets = log(NUM\_SETS) / log(2);

NUM\_SETS = pow(2, inter\_sets);

/\* Calculate cache variables from user input \*/

BYTE\_SELECT\_BITS = log(BYTES\_PER\_LINE) / log(2);

INDEX\_BITS = log(NUM\_SETS) / log(2);

TAG\_BITS = ADDRESS\_BITS - (BYTE\_SELECT\_BITS + INDEX\_BITS);

#ifdef DEBUG

printf("\nNumber of ways = %u\n", WAYS);

printf("Byte select bits = %u\n", BYTE\_SELECT\_BITS);

printf("Index bits = %u\n", INDEX\_BITS);

printf("Tag bits = %u\n\n", TAG\_BITS);

#endif

/\* Check for an illegal cache configuration \*/

if( (BYTE\_SELECT\_BITS + INDEX\_BITS) >= ADDRESS\_BITS ) {

printf("The entered cache configuration is too large to fit within a 32 bit address.\n");

exit(-2);

}

#ifdef TEST\_FILE

if(argc>1){

trace\_file = fopen(argv[1], "r");

}else{

printf("\nplease specify trace file: \"./a.out <file name> \"\n\n");

exit(-1);

}/\*end else\*/

#else

trace\_file = fopen("cc1.din", "r");

#endif

if(trace\_file == NULL) {

printf("Could not open the specified file\n");

exit(-1);

}/\*end if\*/

/\* If all checks pass above, allocate the sets \*/

sets = (cache\_set\*)malloc(sizeof(cache\_set) \* NUM\_SETS); //Malloc however many sets we need

init\_cache();

// #ifdef WARM\_CACHE

// sets[set\_index].line[line\_index].MESIF = (rand()%4);

// sets[set\_index].line[line\_index].tag = (rand()%exp(16,32));

// #endif

/\* Get and decode each operation in the trace file \*/

while( fgets(buffer, buffer\_len, trace\_file) != NULL ) {

if( feof(trace\_file) ) { /\* EOF has been reached \*/

break;

}

else {

operation\_ptr = strtok(buffer, " "); //Look for the space

address\_ptr = operation\_ptr + 2; //The address is two characters higher

trace.operation = (uint8\_t)atoi(operation\_ptr);

trace.address = (uint32\_t)strtol(address\_ptr, NULL, 16); //Convert hex string to int

#ifdef DEBUG

#ifdef PRETTY\_OUTPUT

if(trace.operation == READ\_DATA\_L1) strcpy(operation\_text, "READ\_DATA\_L1");

else if(trace.operation == WRITE\_DATA\_L1) strcpy(operation\_text, "WRITE\_DATA\_L1");

else if(trace.operation == READ\_INS\_L1) strcpy(operation\_text, "READ\_INS\_L1");

else if(trace.operation == SNOOP\_INV) strcpy(operation\_text, "SNOOP\_INV");

else if(trace.operation == SNOOP\_READ) strcpy(operation\_text, "SNOOP\_READ");

else if(trace.operation == SNOOP\_WRITE) strcpy(operation\_text, "SNOOP\_WRITE");

else if(trace.operation == SNOOP\_RWIM) strcpy(operation\_text, "SNOOP\_RWIM");

else if(trace.operation == CLEAR\_RESET) strcpy(operation\_text, "CLEAR\_RESET");

else if(trace.operation == PRINT\_STATES) strcpy(operation\_text, "PRINT\_STATES");

else strcpy(operation\_text, "ERROR");

printf("Trace op = %s, Trace address = 0x%X\n", operation\_text, trace.address);

#else

printf("Trace op = %d, Trace address = 0x%X\n", trace.operation, trace.address);

#endif

#endif

decode\_trace(trace.operation, trace.address);

#ifdef DEBUG

printf("\n");

#endif

}

}

/\* Cleanup \*/

cleanup(trace\_file);

return 0;

}/\*END MAIN\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* FUNCTION TO DECODE TRACE OPERATION

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Send in the trace op and the address \*/

void decode\_trace(uint8\_t trace\_op, uint32\_t address) {

const uint8\_t no\_match = 0xFF;

uint32\_t index, tag;

uint32\_t byte\_select, dirty\_line\_address;

cache\_set\* indexed\_set;

uint8\_t tag\_matched\_line;

/\* Update the trace counter if the buffer has something in it \*/

if(write\_buffer.head != write\_buffer.tail) {

write\_buffer.trace\_counter++;

}

/\* Decode the address \*/

byte\_select = extract\_byte\_select(address);

index = extract\_index(address);

tag = extract\_tag(address);

/\* Decode the trace operation \*/

indexed\_set = &sets[index];

/\* First, check for our CPU or snooping operations \*/

if( trace\_op < CLEAR\_RESET ) {

tag\_matched\_line = check\_tags(tag, indexed\_set); //Will return 0xFF if there is no match

switch( trace\_op ) {

case READ\_DATA\_L1:

L1\_read\_or\_write(CPU\_READ, tag, address, tag\_matched\_line, indexed\_set);

break;

case WRITE\_DATA\_L1:

L1\_read\_or\_write(CPU\_WRITE, tag, address, tag\_matched\_line, indexed\_set);

break;

case READ\_INS\_L1:

L1\_read\_or\_write(CPU\_READ, tag, address, tag\_matched\_line, indexed\_set);

break;

case SNOOP\_INV:

if(tag\_matched\_line != no\_match) {

/\* If line is modified, it must be put into the write buffer \*/

if(indexed\_set->line[tag\_matched\_line].MESIF == MODIFIED) {

dirty\_line\_address |= (indexed\_set->line[tag\_matched\_line].tag << (BYTE\_SELECT\_BITS + INDEX\_BITS)) | (index << BYTE\_SELECT\_BITS);

message\_to\_L2\_cache(READ, dirty\_line\_address);

add\_to\_write\_buffer(dirty\_line\_address);

}

other\_CPU\_operation(INVALIDATE, address, &indexed\_set->line[tag\_matched\_line]);

indexed\_set->valid\_ways--;

}

break;

case SNOOP\_READ:

if(tag\_matched\_line != no\_match) {

other\_CPU\_operation(READ, address, &indexed\_set->line[tag\_matched\_line]);

}

break;

case SNOOP\_WRITE:

if(tag\_matched\_line != no\_match) {

other\_CPU\_operation(WRITE, address, &indexed\_set->line[tag\_matched\_line]);

}

break;

case SNOOP\_RWIM:

if(tag\_matched\_line != no\_match) {

/\* If line is modified, it must be put into the write buffer \*/

if(indexed\_set->line[tag\_matched\_line].MESIF == MODIFIED) {

dirty\_line\_address |= (indexed\_set->line[tag\_matched\_line].tag << (BYTE\_SELECT\_BITS + INDEX\_BITS)) | (index << BYTE\_SELECT\_BITS);

message\_to\_L2\_cache(READ, dirty\_line\_address);

add\_to\_write\_buffer(dirty\_line\_address);

}

other\_CPU\_operation(RWIM, address, &indexed\_set->line[tag\_matched\_line]);

indexed\_set->valid\_ways--;

}

break;

default:

printf("ERROR: INVALID TRACE OPERATION REQUESTED: %d\n", trace\_op);

break;

}//End switch

} //End if

else if( trace\_op == CLEAR\_RESET ) { /\*Clear the cache and reset the state \*/

reset\_cache();

}/\*end else if\*/

else if( trace\_op == PRINT\_STATES) { /\* Print out all valid cache lines \*/

print\_cache();

}/\*end else if\*/

else {

printf("Not a trace operation\n");

}/\*end else\*/

if(write\_buffer.trace\_counter == WB\_COUNT) { /\* Need to write a line back to memory \*/

writeback\_line\_write\_buffer();

write\_buffer.trace\_counter = 0;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* FUNCTION THAT CHECKS FOR A MATCHING TAG WITHIN THE SET

\* RETURNS 0xFF IF NO TAG MATCHES AND THE LINE INDEX IF THERE IS A MATCH

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

uint8\_t check\_tags(uint32\_t tag, cache\_set\* set) {

uint32\_t index;

uint8\_t match\_index = 0xFF;

/\* Check the lines in the set for matching tag \*/

for(index = 0; index < WAYS; index++) {

if( set->line[index].MESIF != INVALID) {

if( tag == set->line[index].tag ) {

match\_index = index;

#ifdef DEBUG

printf("Tag 0x%X found at line %d\n", tag, index);

#endif

break;

}/\*End if\*/

}/\*End if\*/

}/\*End for\*/

return match\_index;

}/\*END CHECK TAG BIT\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* INIT THE CACHE

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void init\_cache(void) {

int set\_index, line\_index;

clear\_write\_buffer();

/\* Generate the truncate mask for the posted write buffer \*/

for(set\_index = 0; set\_index < BYTE\_SELECT\_BITS; set\_index++) {

write\_buffer.truncate\_mask |= 1 << set\_index;

}

/\*INITIALIZE CACHE ARRAY\*/

for(set\_index = 0; set\_index < NUM\_SETS; set\_index++) {

sets[set\_index].pseudo\_LRU = 0;

sets[set\_index].valid\_ways = 0;

sets[set\_index].line = (cache\_line\*)malloc(sizeof(cache\_line) \* WAYS);

for(line\_index = 0; line\_index < WAYS; line\_index++){

/\* Init values in each line \*/

sets[set\_index].line[line\_index].tag = 0;

sets[set\_index].line[line\_index].MESIF = INVALID;

}/\*end inner for\*/

}/\*end outer for\*/

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* RESETS THE CACHE

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void reset\_cache(void) {

int set\_index, line\_index;

clear\_write\_buffer();

/\* Reset cache statistics \*/

cache\_hits = 0;

cache\_misses = 0;

cache\_reads = 0;

cache\_writes = 0;

/\* Reset cache sets \*/

for(set\_index = 0; set\_index < NUM\_SETS; set\_index++) {

sets[set\_index].pseudo\_LRU = 0;

sets[set\_index].valid\_ways = 0;

for(line\_index = 0; line\_index < WAYS; line\_index++){

sets[set\_index].line[line\_index].tag = 0;

sets[set\_index].line[line\_index].MESIF = INVALID;

}/\*end inner for\*/

}/\*end outer for\*/

}/\*end reset\_cache\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* PRINTS OUT ALL VALID CACHE LINES

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void print\_cache(void) {

int set\_index, line\_index;

char MESIF\_text[10];

uint32\_t tag\_bits;

uint8\_t MESIF\_state;

printf("\n/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\*\* PRINTING VALID LINES IN CACHE\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\n");

for(set\_index = 0; set\_index < NUM\_SETS; set\_index++) {

for(line\_index = 0; line\_index < WAYS; line\_index++){

if(sets[set\_index].line[line\_index].MESIF != INVALID ) {

tag\_bits = sets[set\_index].line[line\_index].tag;

MESIF\_state = sets[set\_index].line[line\_index].MESIF;

#ifdef PRETTY\_OUTPUT

if(MESIF\_state == EXCLUSIVE) strcpy(MESIF\_text, "EXCLUSIVE");

else if(MESIF\_state == SHARED) strcpy(MESIF\_text, "SHARED");

else if(MESIF\_state == FORWARD) strcpy(MESIF\_text, "FORWARD");

else if(MESIF\_state == MODIFIED) strcpy(MESIF\_text, "MODIFIED");

else strcpy(MESIF\_text, "ERROR");

printf("Set number: 0x%X, Line number: %d, MESIF state: %s, Tag bits: 0x%X\n", set\_index, line\_index, MESIF\_text, tag\_bits);

#else

printf("Set number: 0x%X, Line number: %d, MESIF state: %d, Tag bits: 0x%X\n", set\_index, line\_index, MESIF\_state, tag\_bits);

#endif

}/\*end inner if\*/

} //End inner for

} //End outer for

printf("\n/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\*\* PRINTING VALID LINES IN WRITE BUFFER\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\n");

for(set\_index = 0; set\_index < WB\_SIZE; set\_index++) {

if(write\_buffer.slot\_valid[set\_index]) {

printf("Write buffer slot %d holds cache line: 0x%X\n", set\_index, write\_buffer.buffer\_slot[set\_index]);

}

}

printf("\n");

}/\*end print\_cache\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* PRINTS CACHE STATISTICS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void print\_statistics(void) {

double hit\_ratio = (double)cache\_hits / ((double)cache\_hits + (double)cache\_misses);

printf("\n/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\*\* PRINTING CACHE STATISTICS\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\n");

printf("Cache reads: %u\n", cache\_reads);

printf("Cache writes: %u\n", cache\_writes);

printf("Cache hits: %u\n", cache\_hits);

printf("Cache misses: %u\n", cache\_misses);

printf("Cache hit ratio: %g\n", hit\_ratio);

}/\*end print\_statistics\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLEANUP FUNCTION

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void cleanup(FILE\* trace\_file) {

uint32\_t set\_index;

fclose(trace\_file);

/\* Free all the lines \*/

for(set\_index = 0; set\_index < NUM\_SETS; set\_index++) {

free(sets[set\_index].line);

}

free(sets);

print\_statistics();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* DETERMINES WHAT TO DO ON A CPU READ OR WRITE DEPENDING ON IF THE LINE IS

\* ALREADY IN THE CACHE OR NOT

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void L1\_read\_or\_write(uint8\_t CPU\_op, uint32\_t tag, uint32\_t address,

uint8\_t tag\_matched\_line, cache\_set\* indexed\_set) {

const uint8\_t no\_match = 0xFF;

/\* Update cache statistics \*/

if(CPU\_op == READ\_DATA\_L1 || CPU\_op == READ\_INS\_L1) cache\_reads++;

else cache\_writes++;

if( tag\_matched\_line == no\_match ) { /\* Not in the cache already \*/

cache\_misses++;

if( indexed\_set->valid\_ways < WAYS) { /\* There is an invalid line \*/

fill\_invalid\_line(CPU\_op, tag, address, indexed\_set);

}

else { /\* All lines are currently valid \*/

fill\_valid\_line(CPU\_op, tag, address, indexed\_set);

}

}/\*end if\*/

else { /\* In the cache already \*/

indexed\_set->pseudo\_LRU = update\_LRU(tag\_matched\_line, indexed\_set->pseudo\_LRU);

CPU\_operation(CPU\_op, address, &indexed\_set->line[tag\_matched\_line]);

cache\_hits++;

}/\*end else\*/

#ifdef DEBUG

printf("After read/write, pseudo LRU bits = 0x%llX\n", (long long unsigned int)indexed\_set->pseudo\_LRU);

#endif

}/\*end L1\_read\_or\_write\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* FUNCTION THAT FILLS AN INVALID CACHE LINE

\* HANDLE UPDATING MESIF and PLRU in here as well

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void fill\_invalid\_line(uint8\_t CPU\_op, uint32\_t tag, uint32\_t address, cache\_set\* set) {

uint8\_t line\_filled;

uint8\_t line\_index, in\_write\_buffer;

/\* Search write buffer first \*/

in\_write\_buffer = search\_write\_buffer(address); /\* Will return 1 if found \*/

/\* Replace TAG in an invalid line \*/

for(line\_index = 0; line\_index < WAYS; line\_index++) {

if( set->line[line\_index].MESIF == INVALID ) {

set->line[line\_index].tag = tag;

set->valid\_ways++;

line\_filled = line\_index;

break;

}/\*end if\*/

}/\*end for\*/

#ifdef DEBUG

printf("Valid ways = %d\n", set->valid\_ways);

#endif

/\* Update the LRU bits and MESIF state \*/

set->pseudo\_LRU = update\_LRU(line\_filled, set->pseudo\_LRU);

CPU\_operation(CPU\_op, address, &set->line[line\_index]);

#ifndef SILENT

if(in\_write\_buffer) printf("Read cache line from write buffer: 0x%X\n", address);

else printf("Read cache line from DRAM: 0x%X\n", address);

#endif

}/\*end fill\_invalid\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* FUNCTION THAT FILLS A LINE WHEN ALL LINES ARE VALID

\* HANDLE UPDATING MESIF and PLRU in here as well

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void fill\_valid\_line(uint8\_t CPU\_op, uint32\_t tag, uint32\_t address, cache\_set\* set) {

uint8\_t line\_to\_evict, in\_write\_buffer;

uint32\_t index\_bits = extract\_index(address); //Needed to rebuild the address of an evicted dirty line

uint32\_t dirty\_line\_address = 0;

/\* Search write buffer first \*/

in\_write\_buffer = search\_write\_buffer(address); /\* Will return 1 if found \*/

/\* Find the line to evict and change the MESIF state to INVALID

\*\* IF evicted line was in the modified state, write it to write buffer \*/

line\_to\_evict = FindVictim(set->pseudo\_LRU);

if(set->line[line\_to\_evict].MESIF == MODIFIED) {

dirty\_line\_address |= (set->line[line\_to\_evict].tag << (BYTE\_SELECT\_BITS + INDEX\_BITS)) | (index\_bits << BYTE\_SELECT\_BITS);

add\_to\_write\_buffer(dirty\_line\_address);

}

set->line[line\_to\_evict].MESIF = INVALID;

#ifdef DEBUG

printf("Index of line to evict = %d\n", line\_to\_evict);

#endif

/\* Update the tag \*/

set->line[line\_to\_evict].tag = tag;

/\* Update the LRU bits and MESIF state \*/

set->pseudo\_LRU = update\_LRU(line\_to\_evict, set->pseudo\_LRU);

CPU\_operation(CPU\_op, address, &set->line[line\_to\_evict]);

#ifndef SILENT

if(in\_write\_buffer) printf("Read cache line from write buffer: 0x%X\n", address);

else printf("Read cache line from DRAM: 0x%X\n", address);

#endif

}/\*end fill\_valid\_line\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* EXTRACTS THE BYTE SELECT BITS FROM THE ADDRESS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

uint32\_t extract\_byte\_select(uint32\_t address) {

uint32\_t byte\_select\_mask;

uint32\_t byte\_select\_bits;

uint8\_t index;

byte\_select\_mask = 0;

/\* Generate the byte select mask \*/

for(index = 0; index < BYTE\_SELECT\_BITS; index++) {

byte\_select\_mask |= 1 << index;

}

/\* Extract byte select bits \*/

byte\_select\_bits = (address & byte\_select\_mask);

#ifdef DEBUG

printf("Byte select bits = 0x%X\n", byte\_select\_bits);

#endif

return byte\_select\_bits;

}/\*end extract\_byte\_select\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* EXTRACTS THE INDEX BITS FROM THE ADDRESS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

uint32\_t extract\_index(uint32\_t address) {

uint32\_t index\_mask;

uint32\_t index\_bits;

uint8\_t index;

index\_mask = 0;

/\* Generate the index mask \*/

for(index = BYTE\_SELECT\_BITS; index < (BYTE\_SELECT\_BITS + INDEX\_BITS); index++) {

index\_mask |= 1 << index;

}/\*end for\*/

/\* Extract index bits \*/

index\_bits = (address & index\_mask) >> BYTE\_SELECT\_BITS; //Shift into the LSB

#ifdef DEBUG

printf("Index bits = 0x%X\n", index\_bits);

#endif

return index\_bits;

}/\*end extract\_index\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* EXTRACTS THE TAG BITS FROM THE ADDRESS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

uint32\_t extract\_tag(uint32\_t address) {

uint32\_t tag\_mask;

uint32\_t tag\_bits;

uint8\_t index;

tag\_mask = 0;

/\* Generate the tag mask \*/

for(index = (BYTE\_SELECT\_BITS + INDEX\_BITS); index < ADDRESS\_BITS; index++) {

tag\_mask |= 1 << index;

}

/\* Extract tag bits \*/

tag\_bits = (address & tag\_mask) >> (BYTE\_SELECT\_BITS + INDEX\_BITS); //Shift into the LSB

#ifdef DEBUG

printf("Tag bits = 0x%X\n", tag\_bits);

#endif

return tag\_bits;

}/\*end extract\_tag\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* ADDS DIRTY EVICTED CACHE LINE TO THE WRITE BUFFER

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void add\_to\_write\_buffer(uint32\_t address) {

uint32\_t line\_address = address & ~write\_buffer.truncate\_mask; //Set byte offset bits to 0

uint8\_t in\_buffer = search\_write\_buffer(address); /\* Will return 1 if it is in the buffer \*/

/\* Buffer could be full, so stall and write to DRAM \*/

if(write\_buffer.full) {

#ifndef SILENT

printf("Write buffer full - stalling processor to write to DRAM\n");

#endif

writeback\_line\_write\_buffer();

}

if(in\_buffer) { /\* We already have the tag as a write in the buffer \*/

/\* Remove the old write \*/

write\_buffer.buffer\_slot[write\_buffer.tag\_index] = 0;

write\_buffer.slot\_valid[write\_buffer.tag\_index] = 0;

}

/\* Add the new write \*/

write\_buffer.buffer\_slot[write\_buffer.tail] = line\_address;

write\_buffer.slot\_valid[write\_buffer.tail] = 1;

write\_buffer.tail = (write\_buffer.tail + 1) % WB\_SIZE; /\* Wrap around the queue \*/

if(write\_buffer.tail == write\_buffer.head)

write\_buffer.full = 1;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* SEARCHES THE WRITE BUFFER FOR AN EXISTING CACHE LINE

\* RETURNS 1 IF THE LINE IS IN THE BUFFER, 0 IF THE LINE IS NOT IN THE BUFFER

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

uint8\_t search\_write\_buffer(uint32\_t address) {

uint8\_t index;

uint8\_t retval = 0;

uint32\_t line\_address = address & ~write\_buffer.truncate\_mask; //Set byte offset bits to 0

#ifdef DEBUG

printf("Search Write Buffer: Address: 0x%X, Line Address: 0x%X\n", address, line\_address);

#endif

/\* If the two positions are not equal, then the buffer is not empty \*/

if(write\_buffer.tail != write\_buffer.head) {

for(index = 0; index < WB\_SIZE; index++) {

if(write\_buffer.buffer\_slot[index] == line\_address) {

if(write\_buffer.slot\_valid[index]) { /\* The line is in the write buffer \*/

write\_buffer.tag\_index = index;

retval = 1;

break;

} /\* End slot valid \*/

}

} /\* End for \*/

}

return retval;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* WRITES A LINE BACK TO DRAM FROM THE WRITE BUFFER EVERY WB\_COUNT TRACES

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void writeback\_line\_write\_buffer(void) {

/\* Head may not be valid from a newer write to the same address.

\*\* Keep moving forward until a valid line is found

\*/

while(write\_buffer.slot\_valid[write\_buffer.head] != SLOT\_VALID) {

write\_buffer.head = (write\_buffer.head + 1) % WB\_COUNT; /\* Wrap around \*/

}

#ifndef SILENT

printf("Write buffer Writeback to DRAM, Address: 0x%X\n", write\_buffer.buffer\_slot[write\_buffer.head]);

#endif

write\_buffer.buffer\_slot[write\_buffer.head] = 0;

write\_buffer.slot\_valid[write\_buffer.head] = 0;

write\_buffer.head = (write\_buffer.head + 1) % WB\_COUNT; /\* Wrap around \*/

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLEARS THE WRITE BUFFER

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void clear\_write\_buffer(void) {

int index;

/\* Clear write buffer \*/

write\_buffer.head = 0;

write\_buffer.tail = 0;

write\_buffer.full = 0;

write\_buffer.trace\_counter = 0;

write\_buffer.tag\_index = 0;

for(index = 0; index < WB\_SIZE; index++) {

write\_buffer.buffer\_slot[index] = 0;

write\_buffer.slot\_valid[index] = 0;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* EOF

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**Cache.h**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* J.S. PEIRCE jpei2@pdx.edu

\* SEAN KOPPENHAFER koppen2@pdx.edu

\* LUIS SANTIAGO lsantiag@pdx.edu

\*

\* 5 NOVEMBER 2014

\* ECE485 FINAL PROJECT

\* CACHE DESIGN SIMULATION PROGRAM

\*

\*

\* cache.h General header

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <assert.h>

#include <string.h>

#include <time.h>

#include <math.h>

#ifndef \_CACHE\_H

#define \_CACHE\_H

/\* General defines for all functions \*/

#define SILENT

#define TEST\_FILE

//#define DEBUG

#define PRETTY\_OUTPUT

/\* Cache size definitions \*/

#define ADDRESS\_BITS 32

/\*BUS OPS:\*/

#define READ 1

#define WRITE 2

#define INVALIDATE 3

#define RWIM 4

#define SNOOPING 0xFF

/\*SNOOP RESULTS:\*/

#define NOHIT 0

#define HIT 1

#define HITM 2

/\* Holds all variables stored in a cache line \*/

typedef struct {

uint32\_t tag;

uint8\_t MESIF;

uint8\_t last\_bus\_op; /\* MESIF debug - our CPU accesses memory \*/

uint8\_t last\_snoop\_result; /\* MESIF debug - our CPU snooping - NOHIT, HIT or HITM \*/

} cache\_line;

/\* Holds all variables stored in a set \*/

typedef struct {

uint64\_t pseudo\_LRU;

uint32\_t valid\_ways;

cache\_line\* line;

} cache\_set;

#endif

**MESIF.h**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* J.S. PEIRCE jpei2@pdx.edu

\* SEAN KOPPENHAFER koppen2@pdx.edu

\* LUIS SANTIAGO lsantiag@pdx.edu

\*

\* 9 NOVEMBER 2014

\* ECE485 FINAL PROJECT

\*

\* MESIF.h

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "cache.h"

#ifndef \_MESIF\_H

#define \_MESIF\_H

/\* Define the MESIF states \*/

#define INVALID 0

#define EXCLUSIVE 1

#define SHARED 2

#define FORWARD 3

#define MODIFIED 4

/\* Define CPU operations \*/

#define CPU\_READ 0

#define CPU\_WRITE 1

/\* Used to simulate a bus operation and to capture the snoop results of the last

\*\* level caches of other processors

\*/

void bus\_operation(uint8\_t bus\_op, uint32\_t address, uint8\_t snoop\_result);

/\* Simulate snoop results from other processors \*/

uint8\_t get\_snoop\_result(uint32\_t address);

/\* Return our snoop result from another processors operation \*/

void put\_snoop\_result(uint32\_t address, uint8\_t snoop\_result);

/\* Used to simulate our processor accessing the cache.

\*\* Changes the MESIF state accordingly

\*\* CPU\_op can be 0 or 1. 0 = CPU read. 1 = CPU write

\*/

void CPU\_operation(uint8\_t CPU\_op, uint32\_t address, cache\_line\* line);

/\* Used to simulate our processor snooping another processors cache

\*\* access.

\*/

void other\_CPU\_operation(uint8\_t bus\_op, uint32\_t address, cache\_line\* line);

/\* Used to pass a message to the L2 cache \*/

void message\_to\_L2\_cache(uint8\_t bus\_op, uint32\_t address);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* DEBUG FUNCTIONS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Our CPU accessing memory \*/

void F\_CPU(cache\_line);

void E\_CPU(cache\_line);

void S\_CPU(cache\_line);

void I\_CPU(cache\_line);

void M\_CPU(cache\_line);

/\* Our CPU snooping \*/

void F\_SNOOP(cache\_line);

void E\_SNOOP(cache\_line);

void S\_SNOOP(cache\_line);

void I\_SNOOP(cache\_line);

void M\_SNOOP(cache\_line);

#endif

**MESIF.c**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* J.S. PEIRCE jpei2@pdx.edu

\* SEAN KOPPENHAFER koppen2@pdx.edu

\* LUIS SANTIAGO lsantiag@pdx.edu

\*

\* 9 NOVEMBER 2014

\* ECE485 FINAL PROJECT

\*

\* MESIF.c

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "MESIF.h"

/\* Used to simulate a bus operation and to capture the snoop results of the last

\*\* level caches of other processors

\*\* snoop\_result = 0xFF when our processor is snooping and it performs a bus operation

\*/

void bus\_operation(uint8\_t bus\_op, uint32\_t address, uint8\_t snoop\_result) {

char bus\_op\_text[10];

char snoop\_text[5];

#ifndef SILENT

#ifdef PRETTY\_OUTPUT

if( bus\_op == READ ) strcpy(bus\_op\_text, "READ");

else if( bus\_op == WRITE ) strcpy(bus\_op\_text, "WRITE");

else if( bus\_op == INVALIDATE ) strcpy(bus\_op\_text, "INVALIDATE");

else if( bus\_op == RWIM ) strcpy(bus\_op\_text, "RWIM");

else strcpy(bus\_op\_text, "ERROR");

/\* When our processor performs a bus\_op while snooping, it passes 0xFF to snoop\_result \*/

if(snoop\_result == 0xFF) {

printf("Bus OP: %s, Address: %X\n", bus\_op\_text, address);

}

else {

if( snoop\_result == NOHIT ) strcpy(snoop\_text, "NOHIT");

else if( snoop\_result == HIT ) strcpy(snoop\_text, "HIT");

else if( snoop\_result == HITM ) strcpy(snoop\_text, "HITM");

else strcpy(snoop\_text, "ERROR");

printf("Bus OP: %s, Address: %X, Snoop Result: %s\n", bus\_op\_text, address, snoop\_text);

}

#else

/\* When our processor performs a bus\_op while snooping, it passes 0xFF to snoop\_result \*/

if(snoop\_result == 0xFF) {

printf("Bus OP: %d, Address: %X\n", bus\_op, address);

}

else {

printf("Bus OP: %d, Address: %X, Snoop Result: %d\n", bus\_op, address, snoop\_result);

}

#endif

#endif

}

/\* Simulate snoop results from other processors.

\*\* Use low order two bits to determine which snoop

\*\* result to return

\*\* 00 = NOHIT, 01 = HIT, 10 = HITM, 11 = UNUSED

\*/

uint8\_t get\_snoop\_result(uint32\_t address) {

uint8\_t low\_order\_bits = address & 0x3;

uint8\_t retval;

if( low\_order\_bits == 0x2 ) {

retval = HITM;

}

else if( low\_order\_bits == 0x1 ) {

retval = HIT;

}

else {

retval = NOHIT;

}

return retval;

}

/\* Used to report the result of our snooping bus operations by other caches \*/

void put\_snoop\_result(uint32\_t address, uint8\_t snoop\_result) {

char snoop\_text[5];

#ifndef SILENT

#ifdef PRETTY\_OUTPUT

if( snoop\_result == NOHIT ) strcpy(snoop\_text, "NOHIT");

else if( snoop\_result == HIT ) strcpy(snoop\_text, "HIT");

else if( snoop\_result == HITM ) strcpy(snoop\_text, "HITM");

else strcpy(snoop\_text, "ERROR");

printf("Address: %X, Snoop Result: %s\n", address, snoop\_text);

#else

printf("Address: %X, Snoop Result: %d\n", address, snoop\_result);

#endif

#endif

}

/\* Used to simulate our processor accessing the cache.

\*\* Changes the MESIF state accordingly.

\*\* CPU\_op can be 0 or 1. 0 = CPU read. 1 = CPU write

\*/

void CPU\_operation(uint8\_t CPU\_op, uint32\_t address, cache\_line\* line) {

uint8\_t snoop\_result = get\_snoop\_result(address);

uint8\_t bus\_op = 0xFF; /\* When no bus\_op is set for the transistion \*/

uint8\_t MESIF\_state = line->MESIF;

#ifdef DEBUG

if(MESIF\_state == INVALID) printf("Initial MESIF state: INVALID\n");

else if(MESIF\_state == EXCLUSIVE) printf("Initial MESIF state: EXCLUSIVE\n");

else if(MESIF\_state == SHARED) printf("Initial MESIF state: SHARED\n");

else if(MESIF\_state == FORWARD) printf("Initial MESIF state: FORWARD\n");

else if(MESIF\_state == MODIFIED) printf("Initial MESIF state: MODIFIED\n");

else printf("Initial MESIF state: ERROR\n");

#endif

/\* Modify the MESIF state for the line \*/

switch( MESIF\_state ) {

case INVALID:

if( CPU\_op == CPU\_READ ) {

bus\_op = READ;

bus\_operation(bus\_op, address, snoop\_result);

/\* Hit goes high when HITM goes high \*/

if( (snoop\_result == HIT) || (snoop\_result == HITM) ) MESIF\_state = FORWARD;

else MESIF\_state = EXCLUSIVE;

}

else {

bus\_op = RWIM;

bus\_operation(bus\_op, address, snoop\_result);

MESIF\_state = MODIFIED;

}

break;

case FORWARD:

if( CPU\_op == CPU\_WRITE ) {

bus\_op = INVALIDATE;

bus\_operation(bus\_op, address, snoop\_result);

MESIF\_state = MODIFIED;

}

break;

case SHARED:

if( CPU\_op == CPU\_WRITE ) {

bus\_op = INVALIDATE;

bus\_operation(bus\_op, address, snoop\_result);

MESIF\_state = MODIFIED;

}

break;

case EXCLUSIVE:

if( CPU\_op == CPU\_WRITE ) {

MESIF\_state = MODIFIED;

}

break;

case MODIFIED:

MESIF\_state = MODIFIED;

break;

}

#ifdef DEBUG

if(MESIF\_state == INVALID) printf("Modified MESIF state: INVALID\n");

else if(MESIF\_state == EXCLUSIVE) printf("Modified MESIF state: EXCLUSIVE\n");

else if(MESIF\_state == SHARED) printf("Modified MESIF state: SHARED\n");

else if(MESIF\_state == FORWARD) printf("Modified MESIF state: FORWARD\n");

else if(MESIF\_state == MODIFIED) printf("Modified MESIF state: MODIFIED\n");

else printf("Modified MESIF state: ERROR\n");

#endif

/\* Update data in the cache line \*/

line->MESIF = MESIF\_state;

line->last\_bus\_op = bus\_op;

}

/\* Used to simulate our processor snooping another processors cache

\*\* access.

\*/

void other\_CPU\_operation(uint8\_t bus\_op, uint32\_t address, cache\_line\* line) {

uint8\_t snoop\_result;

uint8\_t MESIF\_state = line->MESIF;

#ifdef DEBUG

if(MESIF\_state == INVALID) printf("Initial MESIF state: INVALID\n");

else if(MESIF\_state == EXCLUSIVE) printf("Initial MESIF state: EXCLUSIVE\n");

else if(MESIF\_state == SHARED) printf("Initial MESIF state: SHARED\n");

else if(MESIF\_state == FORWARD) printf("Initial MESIF state: FORWARD\n");

else if(MESIF\_state == MODIFIED) printf("Initial MESIF state: MODIFIED\n");

else printf("Initial MESIF state: ERROR\n");

#endif

switch( MESIF\_state ) {

case INVALID:

snoop\_result = NOHIT;

break;

case FORWARD:

if( bus\_op == RWIM ) {

MESIF\_state = INVALID;

snoop\_result = HIT;

bus\_operation(WRITE, address, SNOOPING);

message\_to\_L2\_cache(INVALIDATE, address);

}

else if( bus\_op == INVALIDATE ) {

MESIF\_state = INVALID;

snoop\_result = NOHIT;

message\_to\_L2\_cache(INVALIDATE, address);

}

else if( bus\_op == READ ) {

MESIF\_state = SHARED;

snoop\_result = HIT;

bus\_operation(WRITE, address, SNOOPING);

}

break;

case SHARED:

snoop\_result = HIT;

if( (bus\_op == RWIM) || (bus\_op == INVALIDATE) ) {

MESIF\_state = INVALID;

message\_to\_L2\_cache(INVALIDATE, address);

}

break;

case EXCLUSIVE:

if( bus\_op == RWIM ) {

MESIF\_state = INVALID;

snoop\_result = HIT;

bus\_operation(WRITE, address, SNOOPING);

message\_to\_L2\_cache(INVALIDATE, address);

}

else if( bus\_op == INVALIDATE ) {

MESIF\_state = INVALID;

snoop\_result = NOHIT;

message\_to\_L2\_cache(INVALIDATE, address);

}

else if( bus\_op == READ ) {

MESIF\_state = SHARED;

snoop\_result = HIT;

bus\_operation(WRITE, address, SNOOPING);

}

break;

case MODIFIED:

if( (bus\_op == RWIM) || (bus\_op == INVALIDATE) ) {

MESIF\_state = INVALID;

snoop\_result = HITM;

/\* Decode logic does a read to L2 already before writing to buffer \*/

bus\_operation(WRITE, address, SNOOPING);

message\_to\_L2\_cache(INVALIDATE, address);

}

else if( bus\_op == READ ) {

MESIF\_state = SHARED;

snoop\_result = HITM;

message\_to\_L2\_cache(READ, address);

bus\_operation(WRITE, address, SNOOPING);

}

break;

}

put\_snoop\_result(address, snoop\_result);

line->MESIF = MESIF\_state;

line->last\_snoop\_result = snoop\_result;

#ifdef DEBUG

if(MESIF\_state == INVALID) printf("Modified MESIF state: INVALID\n");

else if(MESIF\_state == EXCLUSIVE) printf("Modified MESIF state: EXCLUSIVE\n");

else if(MESIF\_state == SHARED) printf("Modified MESIF state: SHARED\n");

else if(MESIF\_state == FORWARD) printf("Modified MESIF state: FORWARD\n");

else if(MESIF\_state == MODIFIED) printf("Modified MESIF state: MODIFIED\n");

else printf("Modified MESIF state: ERROR\n");

#endif

}

/\* Used to pass a message to the L2 cache \*/

void message\_to\_L2\_cache( uint8\_t bus\_op, uint32\_t address) {

char bus\_op\_text[10];

#ifndef SILENT

#ifdef PRETTY\_OUTPUT

if( bus\_op == READ ) strcpy(bus\_op\_text, "READ");

else if( bus\_op == WRITE ) strcpy(bus\_op\_text, "WRITE");

else if( bus\_op == INVALIDATE ) strcpy(bus\_op\_text, "INVALIDATE");

else if( bus\_op == RWIM ) strcpy(bus\_op\_text, "RWIM");

else strcpy(bus\_op\_text, "ERROR");

printf("L2 bus\_op: %s, Address: %X\n", bus\_op\_text, address);

#else

printf("L2 bus\_op: %d, Address: %X\n", bus\_op, address);

#endif

#endif

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* DEBUG FUNCTIONS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Test F states in our CPU \*/

void F\_CPU(cache\_line line) {

line.MESIF = FORWARD;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

/\* Test all read conditions in F \*/

printf("\nRead in F, NOHIT\n");

CPU\_operation(CPU\_READ, 0x0, &line);

assert( line.MESIF == FORWARD );

printf("\nRead in F, HIT\n");

CPU\_operation(CPU\_READ, 0x1, &line);

assert( line.MESIF == FORWARD );

printf("\nRead in F, HITM\n");

CPU\_operation(CPU\_READ, 0x2, &line);

assert( line.MESIF == FORWARD );

/\* Test F -> M with all three hit possibilities \*/

printf("\nF -> M, NOHIT\n");

CPU\_operation(CPU\_WRITE, 0x0, &line);

assert( line.MESIF == MODIFIED );

assert( line.last\_bus\_op == INVALIDATE );

line.MESIF = FORWARD;

printf("\nF -> M, HIT\n");

CPU\_operation(CPU\_WRITE, 0x1, &line);

assert( line.MESIF == MODIFIED );

assert( line.last\_bus\_op == INVALIDATE );

line.MESIF = FORWARD;

printf("\nF -> M, HITM\n");

CPU\_operation(CPU\_WRITE, 0x2, &line);

assert( line.MESIF == MODIFIED );

assert( line.last\_bus\_op == INVALIDATE );

}

/\* Test E states in our CPU \*/

void E\_CPU(cache\_line line) {

line.MESIF = EXCLUSIVE;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

/\* Test all read conditions in E \*/

printf("\nRead in E, NOHIT\n");

CPU\_operation(CPU\_READ, 0x0, &line);

assert( line.MESIF == EXCLUSIVE );

printf("\nRead in E, HIT\n");

CPU\_operation(CPU\_READ, 0x1, &line);

assert( line.MESIF == EXCLUSIVE );

printf("\nRead in E, HITM\n");

CPU\_operation(CPU\_READ, 0x2, &line);

assert( line.MESIF == EXCLUSIVE );

/\* Test E.M with all three hit possibilities \*/

printf("\nE -> M, NOHIT\n");

CPU\_operation(CPU\_WRITE, 0x0, &line);

assert( line.MESIF == MODIFIED );

line.MESIF = EXCLUSIVE;

printf("\nE -> M, HIT\n");

CPU\_operation(CPU\_WRITE, 0x1, &line);

assert( line.MESIF == MODIFIED );

line.MESIF = EXCLUSIVE;

printf("\nE -> M, HITM\n");

CPU\_operation(CPU\_WRITE, 0x2, &line);

assert( line.MESIF == MODIFIED );

}

/\* Test S states in our CPU \*/

void S\_CPU(cache\_line line) {

line.MESIF = SHARED;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

/\* Test all read conditions in E \*/

printf("\nRead in S, NOHIT\n");

CPU\_operation(CPU\_READ, 0x0, &line);

assert( line.MESIF == SHARED );

printf("\nRead in S, HIT\n");

CPU\_operation(CPU\_READ, 0x1, &line);

assert( line.MESIF == SHARED );

printf("\nRead in S, HITM\n");

CPU\_operation(CPU\_READ, 0x2, &line);

assert( line.MESIF == SHARED );

/\* Test E.M with all three hit possibilities \*/

printf("\nS -> M, NOHIT\n");

CPU\_operation(CPU\_WRITE, 0x0, &line);

assert( line.MESIF == MODIFIED );

assert( line.last\_bus\_op == INVALIDATE );

line.MESIF = SHARED;

printf("\nS -> M, HIT\n");

CPU\_operation(CPU\_WRITE, 0x1, &line);

assert( line.MESIF == MODIFIED );

assert( line.last\_bus\_op == INVALIDATE );

line.MESIF = SHARED;

printf("\nS -> M, HITM\n");

CPU\_operation(CPU\_WRITE, 0x2, &line);

assert( line.MESIF == MODIFIED );

assert( line.last\_bus\_op == INVALIDATE );

}

/\* Test I states in our CPU \*/

void I\_CPU(cache\_line line) {

line.MESIF = INVALID;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

/\* Check all reads \*/

printf("\nI -> E, NOHIT\n");

CPU\_operation(CPU\_READ, 0x0, &line);

assert( line.MESIF == EXCLUSIVE );

assert( line.last\_bus\_op == READ );

line.MESIF = INVALID;

printf("\nI -> F, HIT\n");

CPU\_operation(CPU\_READ, 0x1, &line);

assert( line.MESIF == FORWARD );

assert( line.last\_bus\_op == READ );

line.MESIF = INVALID;

printf("\nI -> F, HITM\n");

CPU\_operation(CPU\_READ, 0x2, &line);

assert( line.MESIF == FORWARD );

assert( line.last\_bus\_op == READ );

/\* Check all writes \*/

line.MESIF = INVALID;

printf("\nI -> M, NOHIT\n");

CPU\_operation(CPU\_WRITE, 0x0, &line);

assert( line.MESIF == MODIFIED );

assert( line.last\_bus\_op == RWIM );

line.MESIF = INVALID;

printf("\nI -> M, HIT\n");

CPU\_operation(CPU\_WRITE, 0x1, &line);

assert( line.MESIF == MODIFIED );

assert( line.last\_bus\_op == RWIM );

line.MESIF = INVALID;

printf("\nI -> M, HITM\n");

CPU\_operation(CPU\_WRITE, 0x2, &line);

assert( line.MESIF == MODIFIED );

assert( line.last\_bus\_op == RWIM );

}

/\* Test M states in our CPU \*/

void M\_CPU(cache\_line line) {

line.MESIF = MODIFIED;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

/\* Check all reads \*/

printf("\nREAD in M, NOHIT\n");

CPU\_operation(CPU\_READ, 0x0, &line);

assert( line.MESIF == MODIFIED );

printf("\nREAD in M, HIT\n");

CPU\_operation(CPU\_READ, 0x1, &line);

assert( line.MESIF == MODIFIED );

printf("\nREAD in M, HITM\n");

CPU\_operation(CPU\_READ, 0x2, &line);

assert( line.MESIF == MODIFIED );

/\* Check all writes \*/

printf("\nWRITE in M, NOHIT\n");

CPU\_operation(CPU\_WRITE, 0x0, &line);

assert( line.MESIF == MODIFIED );

printf("\nWRITE in M, HIT\n");

CPU\_operation(CPU\_WRITE, 0x1, &line);

assert( line.MESIF == MODIFIED );

printf("\nWRITE in M, HITM\n");

CPU\_operation(CPU\_WRITE, 0x2, &line);

assert( line.MESIF == MODIFIED );

}

/\* Test our CPU snooping in forward \*/

void F\_SNOOP(cache\_line line) {

line.MESIF = FORWARD;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

/\* Test all 4 bus operation that could be snooped \*/

printf("\nSnoop READ in F\n");

other\_CPU\_operation(READ, 0x0, &line);

assert( line.MESIF == SHARED );

assert( line.last\_snoop\_result == HIT );

line.MESIF = FORWARD;

printf("\nSnoop WRITE in F\n");

other\_CPU\_operation(WRITE, 0x0, &line);

assert( line.MESIF == FORWARD );

line.MESIF = FORWARD;

printf("\nSnoop RWIM in F\n");

other\_CPU\_operation(RWIM, 0x0, &line);

assert( line.MESIF == INVALID );

assert( line.last\_snoop\_result == HIT );

line.MESIF = FORWARD;

printf("\nSnoop INVALIDATE in F\n");

other\_CPU\_operation(INVALIDATE, 0x0, &line);

assert( line.MESIF == INVALID );

assert( line.last\_snoop\_result == NOHIT );

}

/\* Test our CPU snooping in exclusive \*/

void E\_SNOOP(cache\_line line) {

line.MESIF = EXCLUSIVE;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

/\* Test all 4 bus operation that could be snooped \*/

printf("\nSnoop READ in E\n");

other\_CPU\_operation(READ, 0x0, &line);

assert( line.MESIF == SHARED );

assert( line.last\_snoop\_result == HIT );

line.MESIF = EXCLUSIVE;

printf("\nSnoop WRITE in E\n");

other\_CPU\_operation(WRITE, 0x0, &line);

assert( line.MESIF == EXCLUSIVE );

line.MESIF = EXCLUSIVE;

printf("\nSnoop RWIM in E\n");

other\_CPU\_operation(RWIM, 0x0, &line);

assert( line.MESIF == INVALID );

assert( line.last\_snoop\_result == HIT );

line.MESIF = EXCLUSIVE;

printf("\nSnoop INVALIDATE in E\n");

other\_CPU\_operation(INVALIDATE, 0x0, &line);

assert( line.MESIF == INVALID );

assert( line.last\_snoop\_result == NOHIT );

}

/\* Test our CPU snooping in shared \*/

void S\_SNOOP(cache\_line line) {

line.MESIF = SHARED;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

/\* Test all 4 bus operation that could be snooped \*/

printf("\nSnoop READ in S\n");

other\_CPU\_operation(READ, 0x0, &line);

assert( line.MESIF == SHARED );

line.MESIF = SHARED;

printf("\nSnoop WRITE in S\n");

other\_CPU\_operation(WRITE, 0x0, &line);

assert( line.MESIF == SHARED );

line.MESIF = SHARED;

printf("\nSnoop RWIM in S\n");

other\_CPU\_operation(RWIM, 0x0, &line);

assert( line.MESIF == INVALID );

line.MESIF = SHARED;

printf("\nSnoop INVALIDATE in S\n");

other\_CPU\_operation(INVALIDATE, 0x0, &line);

assert( line.MESIF == INVALID );

}

/\* Test our CPU snooping in shared \*/

void I\_SNOOP(cache\_line line) {

line.MESIF = INVALID;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

/\* Test all 4 bus operation that could be snooped \*/

printf("\nSnoop READ in I\n");

other\_CPU\_operation(READ, 0x0, &line);

assert( line.MESIF == INVALID );

line.MESIF = INVALID;

printf("\nSnoop WRITE in I\n");

other\_CPU\_operation(WRITE, 0x0, &line);

assert( line.MESIF == INVALID );

line.MESIF = INVALID;

printf("\nSnoop RWIM in I\n");

other\_CPU\_operation(RWIM, 0x0, &line);

assert( line.MESIF == INVALID );

line.MESIF = INVALID;

printf("\nSnoop INVALIDATE in I\n");

other\_CPU\_operation(INVALIDATE, 0x0, &line);

assert( line.MESIF == INVALID );

}

/\* Test our CPU snooping in forward \*/

void M\_SNOOP(cache\_line line) {

line.MESIF = MODIFIED;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

/\* Test all 4 bus operation that could be snooped \*/

printf("\nSnoop READ in M\n");

other\_CPU\_operation(READ, 0x0, &line);

assert( line.MESIF == SHARED );

assert( line.last\_snoop\_result == HITM );

line.MESIF = MODIFIED;

printf("\nSnoop WRITE in M\n");

other\_CPU\_operation(WRITE, 0x0, &line);

assert( line.MESIF == MODIFIED );

line.MESIF = MODIFIED;

printf("\nSnoop RWIM in M\n");

other\_CPU\_operation(RWIM, 0x0, &line);

assert( line.MESIF == INVALID );

assert( line.last\_snoop\_result == HITM );

line.MESIF = MODIFIED;

printf("\nSnoop INVALIDATE in M\n");

other\_CPU\_operation(INVALIDATE, 0x0, &line);

assert( line.MESIF == INVALID );

}

/\* MESIF unit test - can uncomment for debug \*/

//int main() {

// cache\_line line;

//

// line.MESIF = INVALID;

// line.tag = 0;

//

// /\* Check our CPU doing operations \*/

// printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

// printf("Our CPU accessing memory\n");

// printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

// F\_CPU(line);

// E\_CPU(line);

// S\_CPU(line);

// I\_CPU(line);

// M\_CPU(line);

//

// /\* Check us snooping other CPUs \*/

// printf("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

// printf("Our CPU snooping\n");

// printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

// F\_SNOOP(line);

// E\_SNOOP(line);

// S\_SNOOP(line);

// I\_SNOOP(line);

// M\_SNOOP(line);

//

// return 0;

//}

**Pseudo\_LRU.h**

//

// pseudolru.h

//

//

// Created by LUIS SANTIAGO on 11/21/14.

//

//

#ifndef \_\_\_\_pseudolru\_\_

#define \_\_\_\_pseudolru\_\_

#include "cache.h"

/\* Internal datatype \*/

typedef struct {

uint32\_t victim; // initiate to 0 and it is used to generate

// binary value of Least recently used

uint32\_t levels; // saves Log2(Ways) used to count leves

// on tree and determine when done

uint32\_t MIDPOINT; // Value to determine walk direction

uint8\_t Bits; // index of bits in LRU

uint64\_t LRU; // Stores passed LRU (only for simplicity)

uint8\_t n; // used to calculate ways

}LRU\_Data;

/\* Global functions \*/

// Used to update the LRU bits

// Input set Pseudo\_LRU and the index of the line

// used referenced in the set

// Returns an updated LRU value

uint64\_t update\_LRU(uint8\_t index, uint64\_t LRU);

// returns the least recently used line in the set

// using P\_LRU

uint8\_t FindVictim(uint64\_t LRU);

/\* Internal functions \*/

void decode\_LRU(LRU\_Data\* packLRU);

void GoRight(LRU\_Data\* packLRU);

void GoLeft(LRU\_Data\* packLRU);

void Check\_Tree(LRU\_Data\* packLRU, uint32\_t line);

#endif /\* defined(\_\_\_\_pseudolru\_\_) \*/

**Pseudo\_LRU.c**

//

// pseudolru.c

// J.S. PEIRCE jpei2@pdx.edu

// SEAN KOPPENHAFER koppen2@pdx.edu

// LUIS SANTIAGO lsantiag@pdx.edu

//

//

// ECE485 FINAL PROJECT

//

// Created by LUIS SANTIAGO on 11/21/14.

//

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

WAYS is a global variable deffined in \_\_\_\_\_\_\_\_\_

\*/

#include "pseudolru.h"

#define CHECK\_BIT(var,pos) ((var) & (1<<(pos)))

#define SET\_BIT(var,n) ((var) |= (1 << (n)))

#define CLR\_BIT(var,n) ((var) &= ~(1 << (n)))

extern uint32\_t WAYS; /\* Defined in f\_proj.h \*/

void decode\_LRU(LRU\_Data\* packLRU)

{

/\* check LRU bits and use them as flag to get the next bit

to read in order to find the direction at the next lower level

When travering each level it creates the binary value of victim

-Bit is defined in find victim as Bit=0 which is the

root of tree.

-Compute next bit by:

- when going right (Bit\*2)+2

- going left (Bit\*2)+1

\*/

if (CHECK\_BIT(packLRU->LRU,packLRU->Bits)) {

//Update to indicate indicate next lower level

--packLRU->levels;

//Calculate next bit to read

packLRU->Bits=(packLRU->Bits\*2)+2;

// check that we have not reached the end

if (packLRU->levels>0)

{

packLRU->victim = packLRU->victim<<1;

SET\_BIT(packLRU->victim,0);

decode\_LRU( packLRU);

}else{

packLRU->victim = packLRU->victim<<1;

packLRU->victim |= 1 << 0;

return;

}

}

else {

--packLRU->levels; //indicate next lower level

//Calculate next bit to read

packLRU->Bits=(packLRU->Bits\*2)+1;

// check that we have not reached the end

if (packLRU->levels>0)

{

packLRU->victim=packLRU->victim<<1;

decode\_LRU(packLRU);

}else{

packLRU->victim= packLRU->victim<<1;

return;

}

}

}

//Uses the LRU value from a set to find the victim line

uint8\_t FindVictim (uint64\_t LRU)

{

LRU\_Data myStruct;

LRU\_Data \*packLRU;

packLRU= & myStruct;

packLRU->Bits=0;

packLRU->victim=0;

packLRU->LRU=LRU;

// levels determine how many iteration are needed based on WAYS

packLRU->levels=log2(WAYS);

assert((pow(2,packLRU->levels))==WAYS); // ensures it is a power of 2

decode\_LRU(packLRU );

return packLRU->victim;

}

/\* Searches down a right branch \*/

void GoRight(LRU\_Data\* packLRU)

{

uint32\_t midpoint = packLRU->MIDPOINT;

// first clear bit

CLR\_BIT(packLRU->LRU , packLRU->Bits);

// reduce levels

--packLRU->levels;

// calculate and Store next bit to check

packLRU->Bits=(packLRU->Bits\*2)+2;

// update N in order to correctly update midpoint

++packLRU->n;

// update midpoint and store back into struct

packLRU->MIDPOINT= midpoint + WAYS/(pow(2,packLRU->n));

}

/\* Searches down the left branch \*/

void GoLeft(LRU\_Data\* packLRU)

{

uint32\_t midpoint = packLRU->MIDPOINT;

// first clear bit

SET\_BIT(packLRU->LRU , packLRU->Bits);

--(packLRU->levels);

// calculate and Store next bit to check

packLRU->Bits=(packLRU->Bits\*2)+1;

// update N in order to correctly update midpoint

++packLRU->n;

// update midpoint and store back into struct

packLRU->MIDPOINT= midpoint - WAYS/(pow(2,packLRU->n));

}

/\* Check line value given against the root of the tree

next it will create a subtree of either left or right side and

and repeat, in the process it will update the LRU bits

accordinly to update for the MOST recently used

\*/

void Check\_Tree(LRU\_Data\* packLRU, uint32\_t line)

{

/\* midpoint originally was set by update LRU

Each desition acter this must update midpoint

\*/

if (line < packLRU->MIDPOINT) {

if (packLRU->levels > 0)

{

GoLeft(packLRU);

Check\_Tree (packLRU, line);

}

}

else {

if (packLRU->levels > 0)

{

GoRight(packLRU);

Check\_Tree (packLRU, line);

}

}

}

uint64\_t update\_LRU(uint8\_t line, uint64\_t LRU)

{

// creates a struct and saves the pointer

// Struct contains some if the elements needed to

// compute and update LRU

LRU\_Data myStruct;

LRU\_Data \*packLRU;

packLRU= & myStruct;

packLRU->Bits=0;

packLRU->victim=0;

packLRU->LRU=LRU;

packLRU->n=1;

packLRU->levels=log2(WAYS) ;

packLRU->MIDPOINT= WAYS/(pow(2,packLRU->n));

Check\_Tree(packLRU, line);

return packLRU->LRU;

}

//int main()

//{

// uint8\_t i, victim;

// char cont;

//

// cache\_set test\_set;

// test\_set.pseudo\_LRU=0;

// LRU\_Data \*packLRU;

// LRU\_Data myStruct;

// packLRU= & myStruct;

//

// printf( "Enter Cache Set ways:\n");

// scanf("%u", &WAYS);

//

// //Fill up all entered ways

//

// for(i = 0; i < WAYS; i++) {

// test\_set.pseudo\_LRU = update\_LRU(i, test\_set.pseudo\_LRU);

// test\_set.valid\_ways++;

// printf("On init %d, LRU = 0X%llx\n\n", i, test\_set.pseudo\_LRU);

// }

//

// printf( "\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

// printf( "\nPress y (lower case) with Victim test:\n");

// scanf(" %c", &cont);

//

// if(cont=='y'){

// // Evict all lines in order

// for(i = 0; i < WAYS; i++) {

// victim = FindVictim(test\_set.pseudo\_LRU);

// test\_set.pseudo\_LRU = update\_LRU(victim, test\_set.pseudo\_LRU);

// printf("Victim is index %d\n\n", victim);

// }

// }

//

// return 0;

//}

**format\_input.pl**

#!/usr/bin/perl

# Formats the input text file for our Cache Simulator

# Removes empty lines from the end of the file and removes excess spaces

#

# Output format is: n address

# Wrriten by Sean Koppenhafer 11/15/2014

# ECE485 Cache Simulator - Luis Santiago, Sean Koppenhafer, Steve Pierce

use warnings;

use strict;

use Getopt::Long;

my $filename = undef;

my $output\_filename = "formatted\_trace.txt";

my $read\_file;

my $write\_file;

GetOptions(

"f=s" => \$filename,

);

die("Enter the filename of the input you wanted formatted with -f \"filename\"\n") unless defined ($filename);

#Open the files

open($read\_file, "<", $filename) or die "Cannot open file $filename\n";

open($write\_file, ">", $output\_filename) or die "Cannot create file $output\_filename\n";

my $formatted\_string;

#Read in the unformatted text and format it

while( my $row = <$read\_file> ) {

chomp $row;

#Looks for rows that have our format but with too many spaces

#Then makes it so there is only 1 space between operation and address

if($row =~ m/^(\d)\s+([\da-f]+)/i) {

$formatted\_string = "$1 $2\n";

print $write\_file $formatted\_string;

}

elsif($row =~ m/^(\d)/) { #Is a print or reset operation

$formatted\_string = "$1 0\n"; #Provide a dummy address

print $write\_file $formatted\_string;

} }

close($read\_file);

close($write\_file);