ENG 463-L1

Lab #4 ARM Cache Implementation

Brian Worts and Chris Jenson

05/4/21



**Table of Contents**

Introduction………………………………………………………………………………………. 2

Problem Statement…...…………………………………………………………………………... 2

Procedure/Results…...……...…...……………………………………………………………….. 2

Discussion……………………………………………………………………………………….. 7

Conclusion .................................................................................................................................... 7

Appendix A (Verilog Code)........................................................................................................... 8

**Intro:**

Caches are an essential part of modern computer systems and are best described as a hardware or software component that stores recently accessed data so that future requests for that data can be served faster, since accessing a cache is faster than accessing memory. One can judge a cache's performance on the miss/hit ratios where a hit means that the desired data already exists in the cache and a miss is when the data is not present in the cache. Caches can have various configurations and policies to best serve different specifications, and this lab looks at a hardware implementation of the best case scenario cache from Laboratory 1.

**Problem Statement:**

Using the provided trace files, study, via simulation, the performance of the best case architecture from Laboratory 1. The results should correlate with each other.

**Procedure/Results:**

Using the results obtained from Laboratory One, shown in Figures 1 and 2, the best case architecture was selected. From the data, the best case was determined to be a cache with LRU replacement policy, KN = 256 and K = 2 where there are 8 bytes per line of cache memory and K is the number of lines per set and N is the number of lines. Thus, an architecture diagram was created, shown in Figure 3.

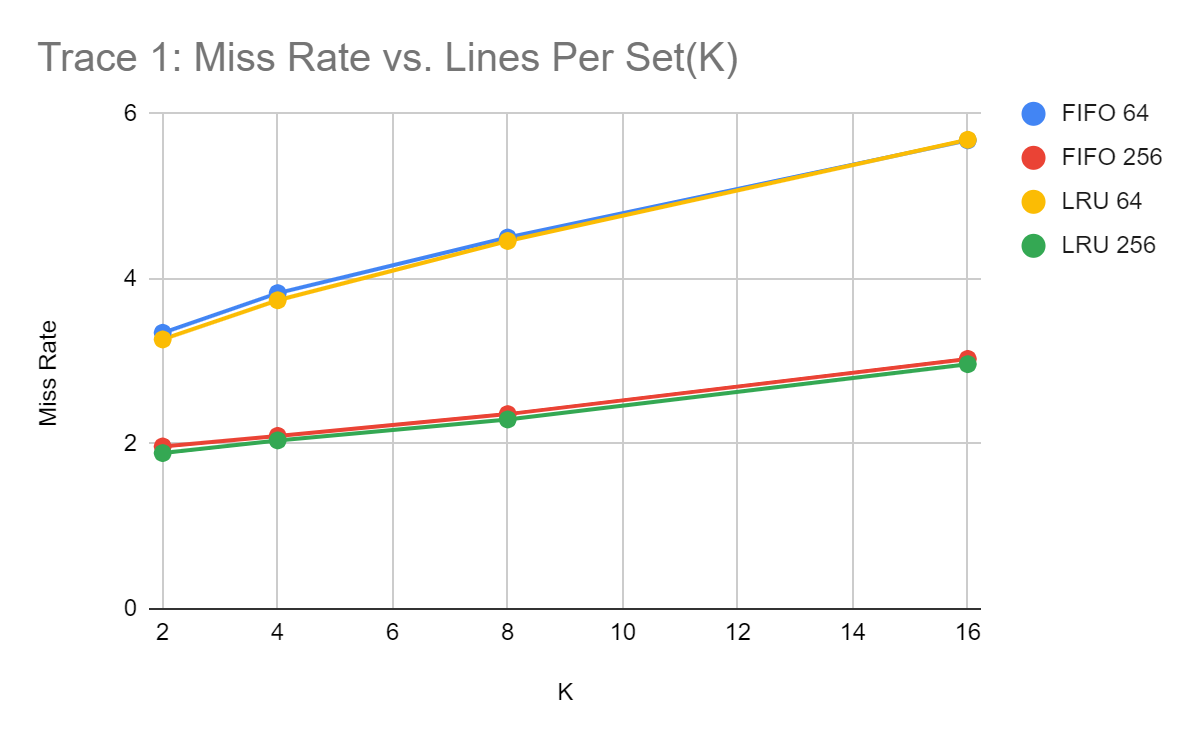
**

Figure 1: Results from Laboratory 1 for Trace 1, Miss Rate vs K

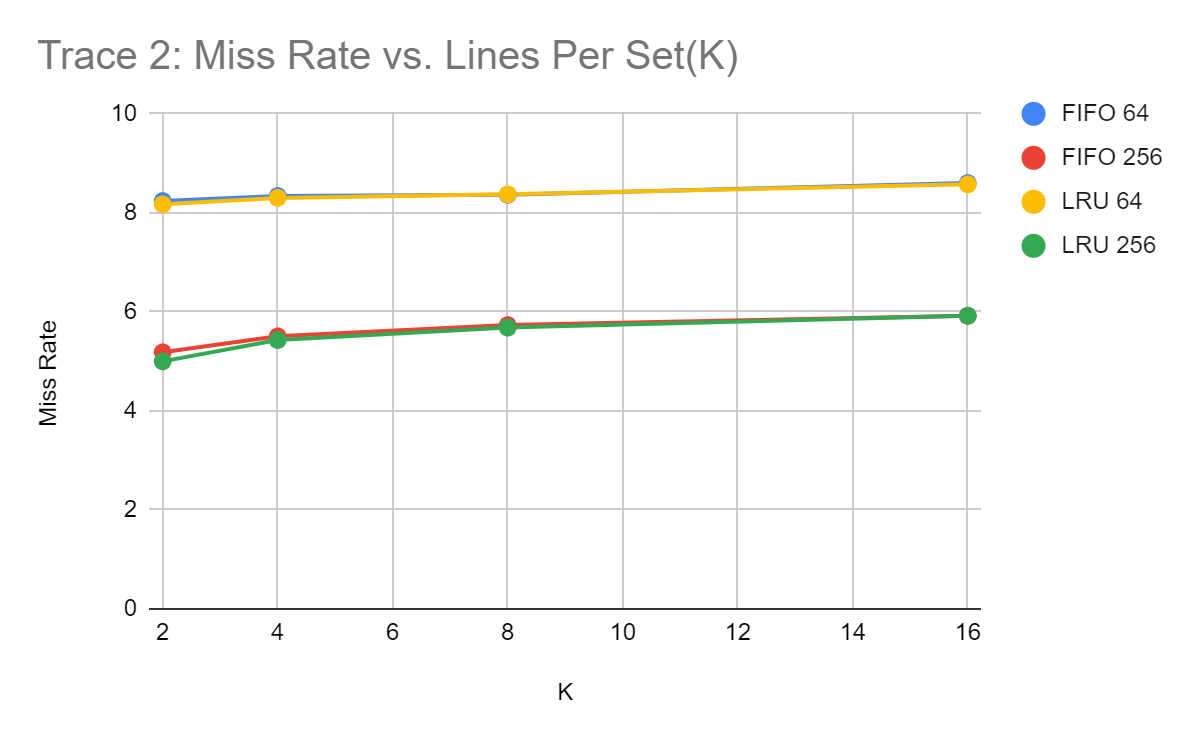
**

Figure 2: Results from Laboratory 1 for Trace 2, Miss Rate vs K

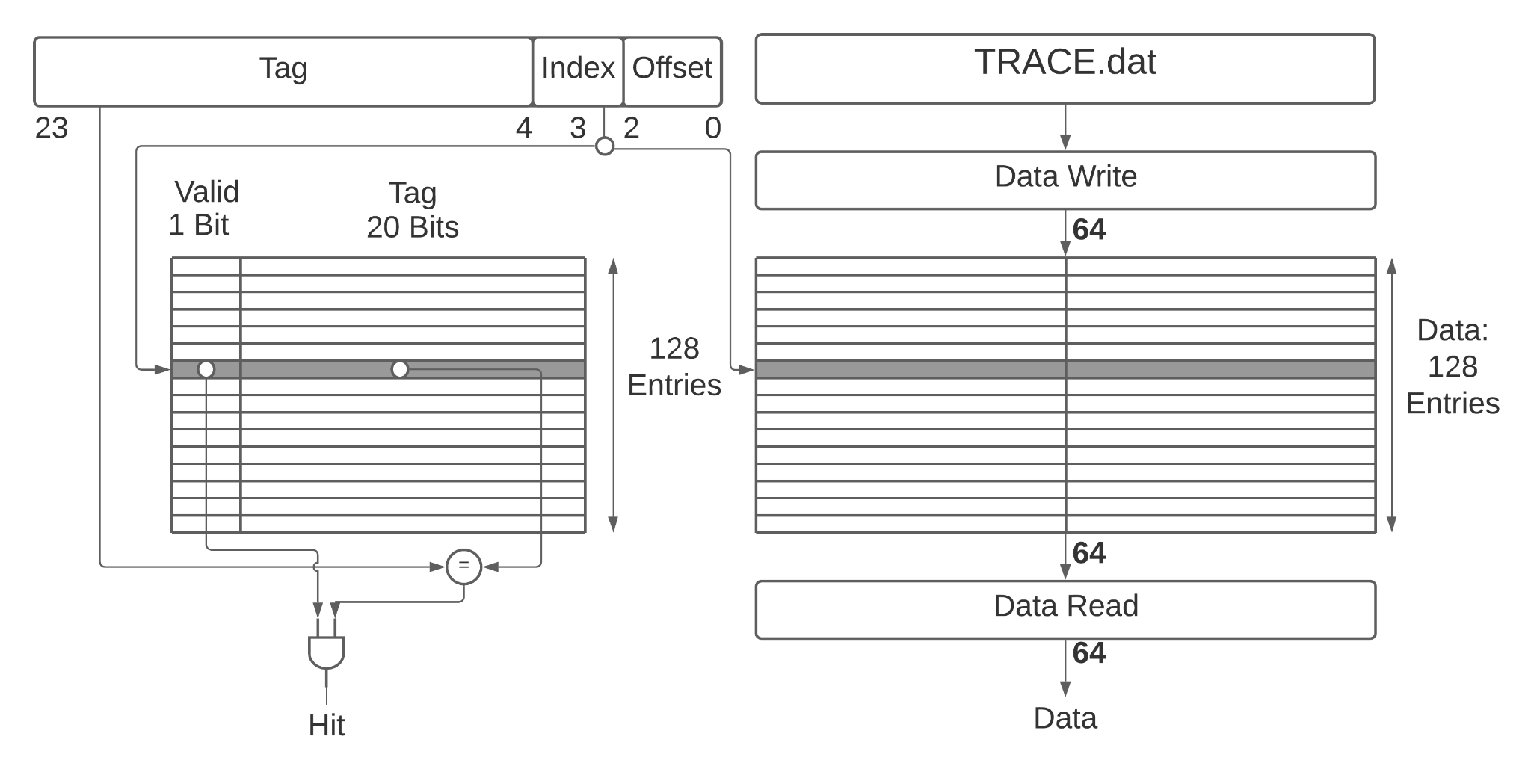


Figure 3: Cache Architecture Diagram, KN = 256, K = 2

With the architecture determined, the cache could be implemented using Verilog. First, the module for the Lower Level Memory, which held the memory references and was part of the testbench code, was created. The Cache and Cache Control were also implemented as part of the design code. These modules are given in Appendix 1.

With the code created, the design could be verified. In order to do so, the given trace files were assigned to the Lower Level Memory and read one at a time. It is important to note that, when a miss occurs in the cache, an extra clock cycle is required to write that data to the cache whereas when a hit occurs, the program can simply move on to the next reference. Both hits and misses require updates to the replacement priority index table that is used to keep track of the Least Recently Used References stored in the cache.

Figure 4 shows the initial “reset” state along with the first 3 References in Trace File 1, all of which were Misses as indicated by the “Hit” line staying low as well as each reference taking 3 clock cycles to perform the needed actions.

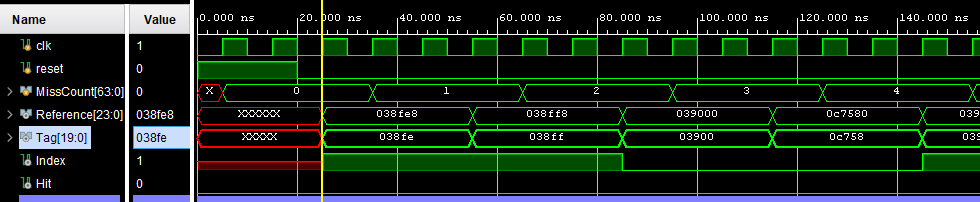


Figure 4: First references into the Cache (Misses) proving successful reading

Figure 5 shows the first hit of the trace file, at the 11th Reference along with hits at the 12th and 13th Reference as well. This can be observed from the “Hit” line going high, as well as all actions taking only 2 clock cycles for each hit as opposed to 3 as is needed for Misses.

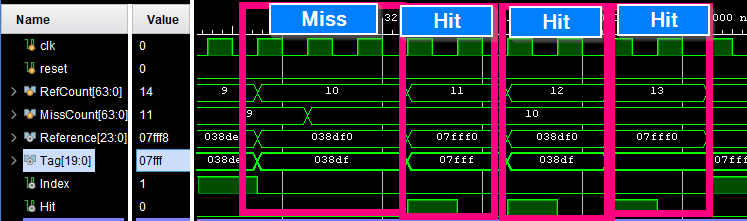


Figure 5: First Hits in the Cache, 2 clock cycles each

Figure 6 shows the first replacement that occurs in the Cache. This replacement occurs at Reference 785, Index 0, at Cache Address 0 where the Least Recently Used reference is. This replacement occurs once all cache locations have had their Valid Bits set to 1.

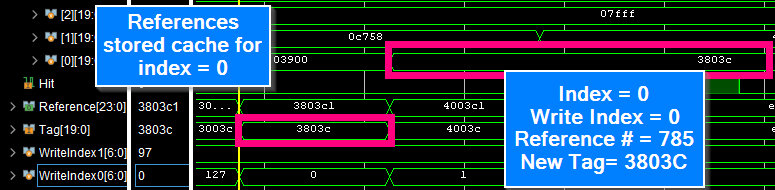


Figure 6: First replacement in Cache for Index = 0. Next occurs at WriteIndex = 1, then at 4

Now that the functionality of the Cache was verified, the full program was run for Trace 1 and Trace 2 in order to find the miss rates for each. The results, shown in Figures 7 and 8, are compared with the results from Laboratory 1 in Table 1.

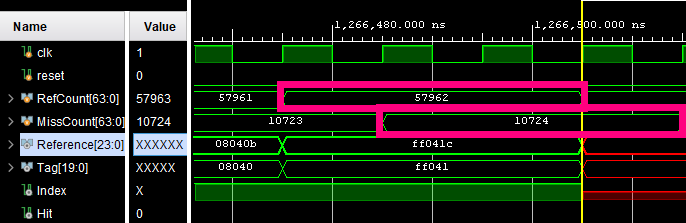


Figure 7: Final miss count (10724) and total reference count (57962) for Trace 1

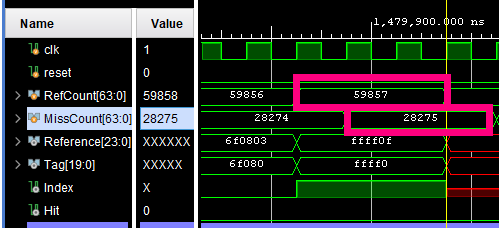


Figure 8: Final miss count (28275) and total reference count (59857) for Trace 2

Table 1: Miss rates (Miss Count Divided by Number of References Times 10)

|  |  |  |
| --- | --- | --- |
|  | **Laboratory 4** | **Laboratory 1** |
| Trace 1 Miss Rate | 1.7873% | 1.8887% |
| Trace 2 Miss Rate | 4.7125% | 5.0005% |

**Discussion:**

From the results shown in Table 1, it can be seen that the miss rates for the software implementation from Laboratory 1 and the hardware implementation from this Laboratory are within a reasonable margin of error of each other, thus verifying the implementation of the cache. The differences in the results of the labs most likely comes from existing logic errors in the Laboratory 1 implementation, though a detailed analysis of the first laboratory code would be needed in order to fix the errors. Confidence is much higher in the results from this lab as a result of the need to implement only 1 cache configuration, as opposed to 16 in lab 1, so more time was spent in this lab on manually verifying the results.

Some of the hurdles the group had to overcome were the diagramming of the cache and determining how the code would perform the LRU replacement priority in Verilog. Utilizing the available resources, such as the Computer Organization And Design Textbook, the team was able to work out how to best represent the Cache with the given specifications. Working out the cache states on paper also showed the group how to control the replacement priorities for the cache in order to best implement the LRU policy.

**Conclusion:**

Simulating the cache systems yielded results that matched our expectations, that the hardware implementation would yield results nearly identical to the software implementation from Lab 1. The LRU replacement policy was used since it had a lower miss rate than the FIFO implementations, and the values K = 2 and KN = 256 were chosen as that implementation had the lowest miss rate overall. This lab furthered the groups understanding of how cache systems work and their respective qualities, as well as the capabilities of Verilog for programming hardware that is both functional and efficient.

**Appendix A - Source Code**

`timescale 1ns / 1ps

//`define REPO "LRU";

`define L (8) //Num Bytes per line

`define K (2) //2, 4, 8, 16 //Number lines per set

`define KN (256) //64, 256 //#Num Sets

`define Sets (128) //64, 256 //#Num Sets in cache

`define DataLength (57961) //Num datapoints Trace1

//parameter DataLength = 59856; //Num datapoints Trace2\

module testbench();

reg clk;

//Generate Clock

initial

begin

clk = 0;

while(1)

begin

#5 clk = ~clk;

end

end

/////////////////////////////////////////////

//Reset

reg reset; //Active Low

initial

//Initize reset

begin

reset = 1;

#20 reset = 0;

end

//

wire [23:0] Reference;

wire [19:0] Tag;

wire Index;

wire NewInstructionFetch;

//

/////////////////////////////

//Memory

wire [63:0] ReadAddress;

Memory u\_Memory(

.clk(clk),

.reset(reset),

.Reference(Reference),

.ReadAddress(ReadAddress),

.NewInstructionFetch(NewInstructionFetch)

);

////////////////////////////////

/////////////////////////////

//Processor

/\*

Processor u\_Processor(

);

////////////////////////////////

\*/

/////////////////////////////

//Cache Control

//Produces Hit/Miss

wire WrEn;

wire RdEn;

wire Hit;

CacheControl u\_CacheControl(

.clk(clk),

.reset(reset),

.Reference(Reference),

.Tag(Tag),

.Index(Index),

.WrEn(WrEn),

.RdEn(RdEn),

.ReadAddress(ReadAddress),

.NewInstructionFetch(NewInstructionFetch),

.Hit(Hit)

);

////////////////////////////////

/////////////////////////////

//CacheMem

// wire [31:0] CacheReadOut;

// wire [31:0] CacheWriteIn;

// wire [31:0] CacheIndex;

Cache u\_Cache(

.clk(clk),

.Index(Index),

.Tag(Tag),

.Hit(Hit),

.WrEn(WrEn),

.RdEn(RdEn),

.NewInstructionFetch(NewInstructionFetch)

);

////////////////////////////////

endmodule

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ps

module Cache(

input clk,

input Index,

input [19:0] Tag,

input WrEn,

input RdEn,

input NewInstructionFetch,

output reg Hit //1 = miss, 2 = hit

);

reg [6:0] WriteIndex1;

reg [6:0] WriteIndex0;

reg doReplacement1;

reg doReplacement0;

reg [6:0] LRUReplacementPriority1 [127:0];

reg [6:0] LRUReplacementPriority0 [127:0];

reg ValidBitsIndex1 [127:0]; //128 1 bits

reg ValidBitsIndex0 [127:0]; //128 1 bits

reg [19:0] TagBits1 [127:0]; //128 20 bits

reg [19:0] TagBits0 [127:0]; //128 20 bits

integer i;

initial

begin

WriteIndex1 = 0;

WriteIndex0 = 0;

doReplacement1 = 0;

doReplacement0 = 0;

Hit = 0;

for (i=0;i<128;i=i+1) begin

ValidBitsIndex1[i] = 0;

ValidBitsIndex0[i] = 0;

TagBits1[i] = 0;

TagBits0[i] = 0;

LRUReplacementPriority1[i] = 0;

LRUReplacementPriority0[i] = 0;

end

end

wire WrEnPosedgePulse;

wire WrEnNegedgePulse;

reg WrEn\_p1;

always @(posedge clk)

begin

WrEn\_p1 <= WrEn;

end

assign WrEnPosedgePulse = WrEn && ~WrEn\_p1;

assign WrEnNegedgePulse = ~WrEn && WrEn\_p1;

reg [6:0] HitIndex;

always @(posedge clk)

begin

//Replace Tag and Set Valid

/\*

if (WrEn)

begin

if (Index == 1)

begin

TagBits1[WriteIndex1] = Tag;

ValidBitsIndex1[WriteIndex1] = 1;

LRUReplacementPriority1[WriteIndex1] = 127;

end

else if (Index == 0)

begin

TagBits0[WriteIndex0] = Tag;

ValidBitsIndex0[WriteIndex0] = 1;

LRUReplacementPriority0[WriteIndex0] = 127;

end

end

\*/

//if (Hit == 2)

//begin

// LRUReplacementPriority1[HitIndex] = 127;

//end

end

integer j;

integer k;

integer m;

always @(\*)

begin

///////////////////////////////////////////////////////////////////////////////

//Sets the write index as the lowest value in the replacement priority array

WriteIndex1 = 0;

WriteIndex0 = 0;

for (j = 1; j < 128; j=j+1)

begin

if (LRUReplacementPriority1[j]< LRUReplacementPriority1[WriteIndex1])

begin

WriteIndex1 = j;

end

if (LRUReplacementPriority0[j]< LRUReplacementPriority0[WriteIndex0])

begin

WriteIndex0 = j;

end

end

end

always @(posedge clk)

begin

///////////////////////////////////////////////////////////////////////////////

//Decrement all values that are greater than the new index

if (WrEnNegedgePulse)

begin

//if (j == 128)

//begin

for (m = 0; m < 128; m = m + 1)

begin

if (Index == 1)

begin

if (LRUReplacementPriority1[m] > LRUReplacementPriority1[WriteIndex1])

begin

LRUReplacementPriority1[m] <= LRUReplacementPriority1[m] - 1;

end

end

else if (Index == 0)

begin

if (LRUReplacementPriority0[m] > LRUReplacementPriority0[WriteIndex0])

begin

LRUReplacementPriority0[m] <= LRUReplacementPriority0[m] - 1;

end

end

end

if (m == 128)

begin

if (Index == 1)

begin

TagBits1[WriteIndex1] <= Tag;

ValidBitsIndex1[WriteIndex1] <= 1;

LRUReplacementPriority1[WriteIndex1] <= 127;

end

else if (Index == 0)

begin

TagBits0[WriteIndex0] <= Tag;

ValidBitsIndex0[WriteIndex0] <= 1;

LRUReplacementPriority0[WriteIndex0] <= 127;

end

end

//end

end

if (Hit == 1)

begin

for (k = 0; k < 128; k=k+1)

begin

if (Index == 1)

begin

if (LRUReplacementPriority1[k] > LRUReplacementPriority1[HitIndex])

begin

LRUReplacementPriority1[k] <= LRUReplacementPriority1[k] - 1;

end

end

else if (Index == 0)

begin

if (LRUReplacementPriority0[k] > LRUReplacementPriority0[HitIndex])

begin

LRUReplacementPriority0[k] <= LRUReplacementPriority0[k] - 1;

end

end

end

if (k == 128)

begin

if (Index == 1)

begin

LRUReplacementPriority1[HitIndex] <= 127;

end

else if (Index == 0)

begin

LRUReplacementPriority0[HitIndex] <= 127;

end

end

end

end

reg [19:0] DebugTagBitsAtI;

reg [19:0] DebugTagBitsAt0;

/\*

always @(posedge clk)

begin

//Replace Tag and Set Valid

if (RdEn)

begin

Hit <= 1;

if (Index == 1)

begin

for (i=0;i<128;i=i+1)

begin

if (TagBits1[i] == Tag)

begin

HitIndex <= i;

DebugTagBitsAtI <= TagBits1[i];

Hit <= 2;

end

end

end

else if (Index == 0)

begin

for (i=0;i<128;i=i+1)

begin

if (TagBits0[i] == Tag)

begin

HitIndex <= i;

DebugTagBitsAt0 <= TagBits0[i];

Hit <= 2;

end

end

end

end

else if (NewInstructionFetch)

begin

Hit <= 0;

HitIndex <= 0;

end

end \*/

always @(\*)

begin

//Replace Tag and Set Valid

if (RdEn)

begin

Hit = 0;

if (Index == 1)

begin

for (i=0;i<128;i=i+1)

begin

if (TagBits1[i] == Tag)

begin

HitIndex = i;

DebugTagBitsAtI = TagBits1[i];

Hit = 1;

end

end

end

else if (Index == 0)

begin

for (i=0;i<128;i=i+1)

begin

if (TagBits0[i] == Tag)

begin

HitIndex = i;

DebugTagBitsAt0 = TagBits0[i];

Hit = 1;

end

end

end

end

else if (NewInstructionFetch)

begin

Hit = 0;

HitIndex = 0;

end

end

endmodule

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ps

module CacheControl

(

input clk,

input reset,

input [23:0] Reference,

input Hit,

output [19:0] Tag,

output reg [63:0] ReadAddress,

output Index,

output reg WrEn,

output reg RdEn,

output reg NewInstructionFetch

);

reg [63:0] HitCount;

reg [63:0] MissCount;

reg [63:0] RefCount;

wire [2:0] Offset;

assign Tag = Reference[23:4];

assign Index = Reference[3];

assign Offset = Reference[2:0];

/\*

always @(\*)

//Output:

//RdEn

//WrEn

begin

if (reset)

begin

RdEn <= 1;

WrEn <= 0;

end

else

begin

if (NewInstructionFetch)

begin

RdEn <= 1;

end

end

end\*/

always @(posedge clk)

begin

//Drive ReadAddress

//Output:

//ReadAddress

if (reset)

begin

ReadAddress <= -1;

end

else

begin

if (NewInstructionFetch)

begin

ReadAddress <= ReadAddress + 1;

end

end

end

always @(posedge clk)

begin

//Output:

//NewInstructionFetch

if (reset)

begin

NewInstructionFetch <= 1;

RdEn <= 0;

WrEn <= 0;

HitCount <= 0;

MissCount <= 0;

RefCount <= 0;

end

else

begin

if (NewInstructionFetch)

begin

RefCount <= RefCount + 1;

NewInstructionFetch <= 0;

RdEn <= 1;

end

else if (RdEn)

begin

if (Hit == 0)

begin //MISS

WrEn <= 1;

MissCount <= MissCount + 1;

RdEn <= 0;

end

else if (Hit == 1)

begin //HIT

HitCount <= HitCount + 1;

NewInstructionFetch <= 1;

RdEn <= 0;

end

end

else if (WrEn)

begin

WrEn <= 0;

NewInstructionFetch <= 1;

end

end

end

endmodule

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ps

module Memory( //BITS

input clk,

input reset,

input [63:0] ReadAddress,

input NewInstructionFetch,

output reg [23:0] Reference

);

reg [23:0] Memory [59999:0];//57962

reg [23:0] Input;

integer i;

integer File;

initial

begin

i = 0;

//Initialize Memory File

File = $fopen("D:/CELab2/Lab4/TRACE1.DAT", "rb");

while (!$feof(File))

begin

Input = ($fgetc(File)) | ($fgetc(File) << 8) | ($fgetc(File) << 16);

Memory[i] = Input;

i=i+1;

end

$fclose(File);

end

always @(\*)

begin

if (reset)

begin

end

else

begin

Reference <= Memory[ReadAddress];

end

end

endmodule