Internal Architecture Specification (IAS) of Cache Simulator

Nishad Saraf, Sachin Muradi and Unmeel Mokashi

Portland State University, Oregon.

ECE 586 – Computer Architecture

February 15, 2017

Contents

Main Features of L1 Data Cache Simulator

1. Data cache (DC) is a set-associative data cache.
2. Designed for 32-bit, byte addressable computer architecture.
3. DC has (N) 1024 sets, (k) 4 lines per set and (L) a line length of 32 bytes.
4. Uses LRU replacement policy for victim cache, write-back for store and allocate on write.

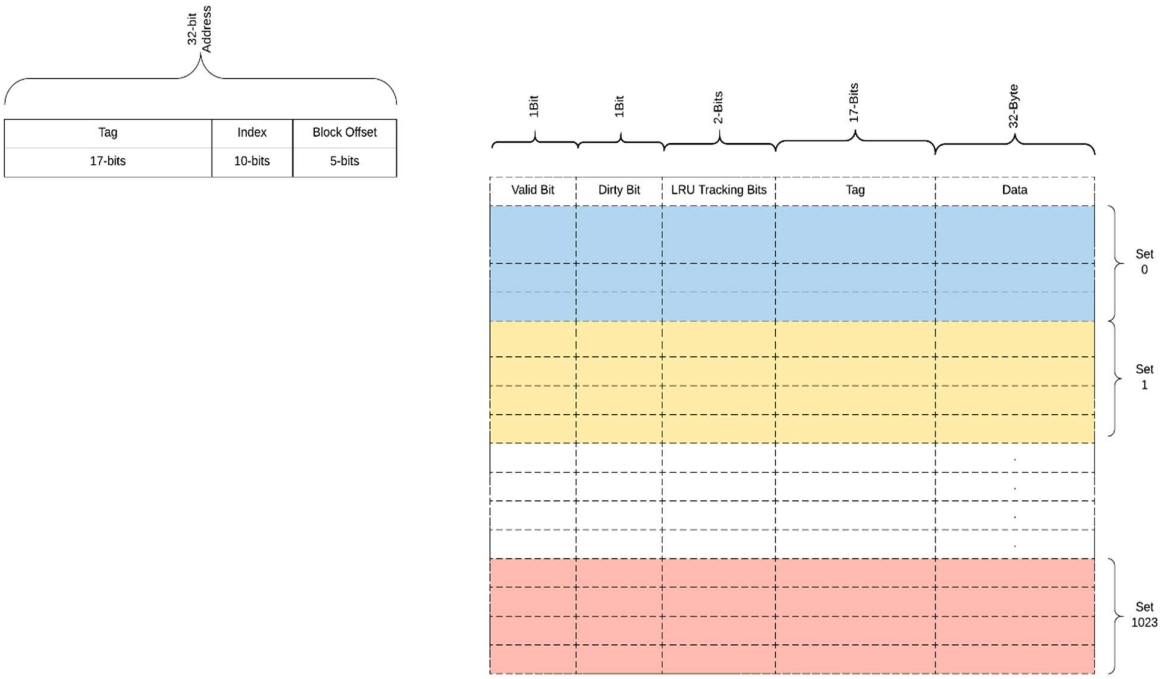


Figure 1: Data Cache. Image Tool: www.lucidcharts.com

Design Parameters

Total Number of Lines:

We know that the total number of lines in cache memory can be expressed by,

$$Number\ of\ Lines = \ k * N = 4 * 1024 = \mathbf{4,096\ \boldsymbol{lines}}$$

Block Offset:

$$Block\ Offset = \ \log_2 L = \ \log_2 32 = \mathbf{5\ \boldsymbol{bits}}$$

Index:

$$Index = \log_2 N = \log_2 1024 = \mathbf{10\ bits}$$

Overhead / Control Bits:

LRU Tracking Bits:

The number of bits required for LRU tracking depends upon the number of lines per set and obeys the following equation,

$$LRU\ tracking\ bits = \log_2 k = \log_2 4 = \mathbf{2\ bits}$$

Each set in our design contains four lines, to uniquely address each line inside a set 2 bits (value between 0-3) are sufficient. The LRU tracking bits of the most recently used line is assigned a value while all the LRU bits corresponding to remaining three are reduced by 1 each if their value is greater than the initial value of the most recently used line. The LRU bits remains the same for values lower than the initial value of the most recently updated line. Therefore, the line with the least value of LRU tracking will be replaced when data cache is full and new data needs to be addressed.

Tag:

$$Tag\ size = bits\ address - block\ offset - index = 32 - 10 - 5 = \mathbf{17\ bits}$$

Tag must be saved along with each cache line to distinguish different address that could be placed in the same set. To locate an address in the cache, use the index to determine which set the data would reside in. Then compare the tag associated with that block to the tag from memory address if it matches and valid bit is set to 1 then the data required is found. A cache hit means that the CPU tried to access an address, and a matching cache block (index, offset, and matching tag) was available in cache. In our case, since the value of block offset bits and index bits are 5 and 10 bits

respectively out of 32 bits address the remaining 17 bits can be used for tag size allocation.

Dirty Bit:

In a write-back cache, the data copied from main memory, is not updated until the cache block needs to be replaced. Write-back mechanism implements a dirty bit which keeps a track of whether this data in cache memory was updated or not. Dirty bit set to 1 indicated that the data in the main memory location is inconsistent and needs to be updated. If dirty bit is set to zero, no action is taken during block replacement as the data in cache and memory are consistent.

Valid Bit:

When the system boots up the cache is empty and there is no valid data. All the data present is a garbage data. During read operation to verify the validity of data we need a valid bit. This bit can help us to keep a track whether the data is a garbage or a valid data. When the system is initialized all the valid bits are set to 0, now whenever data is written into a specific block the corresponding bit is set to one. While reading the data, first the valid bit is checked if it is set to 1 then only it is sent to the CPU or else the required data is fetched from the main memory.

Number of Overhead Bits per Line:

$$Number\ of\ Overhead\ Bits\ per\ Line$$
$$= Tag\ size + LRU\ tracking\ bits + dirty\ bit + valid\ bit$$
$$= 17 + 2 + 1 + 1 = \mathbf{21}\ \boldsymbol{bits}$$

Total Number of Overhead Bits:

$$Total\ Number\ of\ Overhead\ Bits$$
$$= Number\ of\ Overhead\ Bits\ per\ Line * Total\ Number\ of\ Lines$$
$$= 21 * 4096 = \mathbf{86,016\ bits}$$

Data Bits per Cache Lines:

$$Data\ Bits\ per\ Cache\ Lines = 8 * line\ length = 8 * 32 = \mathbf{256\ bits}$$

Total Cache Size:

$$Total\ Cahe\ Size$$
$$= Total\ Number\ of\ Lines(Data\ Bits\ per\ Cache\ Lines$$
$$+ Number\ of\ Overhead\ Bits\ per\ Line) = 4096(256 + 21)$$
$$= \mathbf{1,134,592\ bits = 1,108Kb}$$

Function and Timing:

The designed cache simulator works like a normal L1 data cache. In this simulator, we are not really concerned with what data we are going to store in the cache / memory. The main focus of the simulator is to calculate the timings for data access (access from cache as well as the main memory). The memory is accessed via two operations here: load and store, that is, read and write operations.

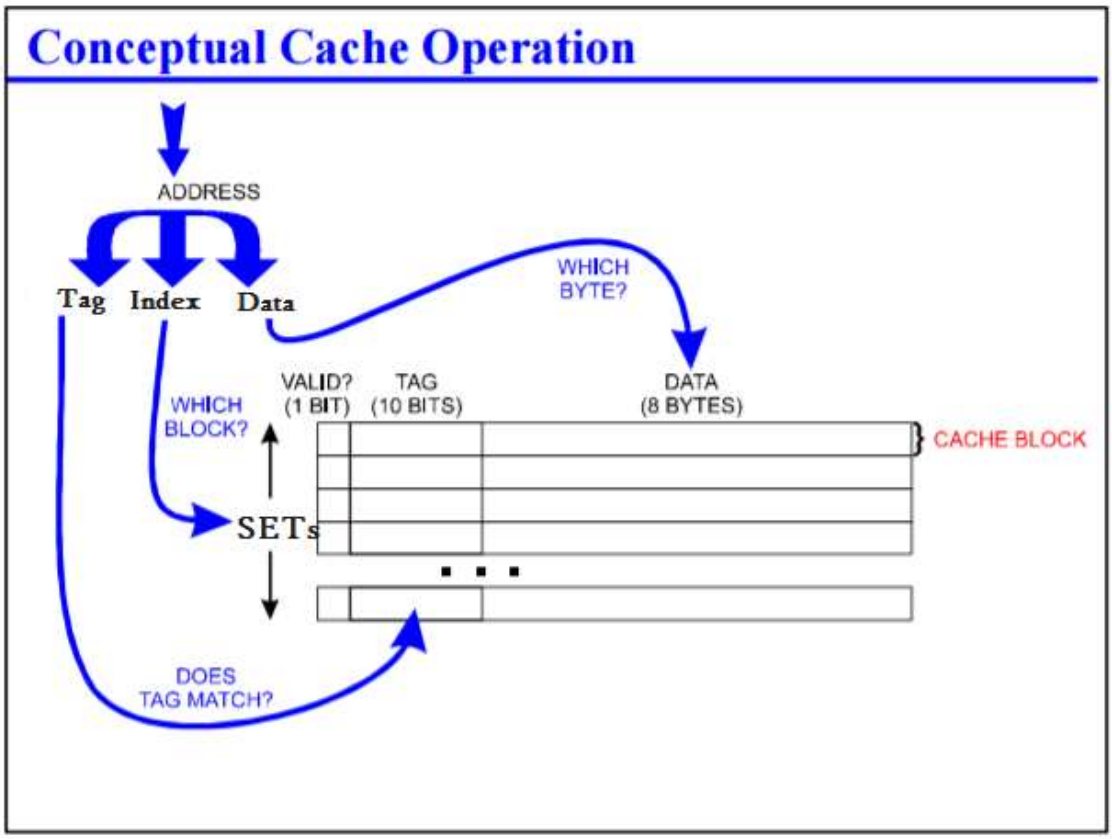


Figure 2: The operation of cache (Hit and Miss)

Ref: https://www.ece.cmu.edu/~ece548/handouts/04cachor.pdf

At the very beginning (reset), the cache lines are empty and valid bit is zero. When the data is present in the cache, the valid bit is set to one. In the cache operation, depending on the index bits, the set is selected. Then the tag bits are compared with each cache lines in that particular set, if valid bit is 1 and tag matches it is a cache hit. If the requested tag bits don't match it is cache miss. In case of miss, the cache requests main memory for the address. Main memory is usually very slow compared to cache. This causes, more clock cycles for data access at CPU side. Once the address along with data is streamed-in to the cache, data is available for CPU.

Write Back with Write Allocate:

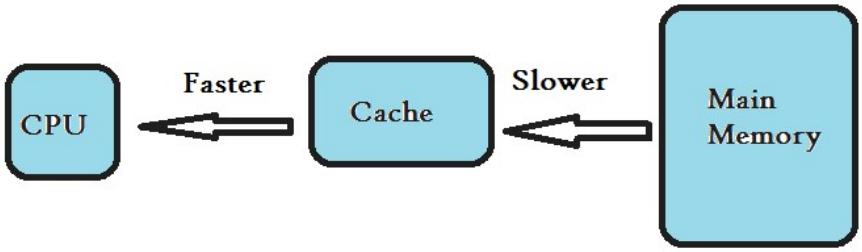
1. On cache hit, writes to cache setting "dirty" bit for the block, here main memory is not updated.
2. On miss it updates the block in main memory and brings the block to the cache.
3. Subsequently writes to the same block, if the block originally caused a miss, will hit in the cache next time, setting dirty bit for the block.
4. That will eliminate extra memory accesses and result in very efficient execution compared with Write Through with Write Allocate combination.

Eviction:

When the all the cache lines are filled with valid data or the valid bit of any cache line is zero, then that particular cache line is removed from the cache. This is called as eviction. For this cache simulator design, we use the LRU (Least Recently Used) policy of eviction. In this policy, the cache line which is least recently used is evicted. To implement the LRU policy, there are more bits added to the cache line. These bits will be used as counter and the one with the least count value will be the least recently used cache line and hence will be evicted.

Timing:

In timing, we are dealing with the number of cycles it takes to access the data for all the memory accesses (cache and main memory), in case of cache hit as well as miss.



The simulator tracks the number of cycles it requires to perform read and write access from the cache, in case of hit. When the cache miss happens, the cache line is evicted and the data is streamed-in from the main memory. The eviction from the cache and the streaming-in of data from memory incurs more cycles for the data access for both read and right operations. For this simulator, we assume that it takes one cycle for the cache hit.

$$\boldsymbol{Hit\ time = 1\ cycle}$$

In case of cache miss, the data is requested from main memory (dynamic RAM), which is slower than cache (static RAM). In this simulator, we assume that the total cost (miss penalty) for accessing the data from memory is 50 cycles. This also includes eviction time if required.

$$\boldsymbol{Miss\ time = 50\ cycles\ for\ main\ memory\ access + 1\ cycle\ for\ cache\ access}$$

For the given miss rate (r), Total access time:

$$\boldsymbol{Access\ time} = \boldsymbol{Hit\ time} + (\boldsymbol{Miss\ rate} \times \boldsymbol{Miss\ time}) = \mathbf{1} + (\boldsymbol{r} \times \mathbf{51})$$

Input/ Output Samples and formats:

Input:

The following lines specify the format the cache simulator will take in as input and includes sample test cases along with the debug commands that will be used. The input to the simulator is through a text file via the ***stdin*** console.

Consider a sample input through a text file. The text file contains all the address which the cache must read from or write to. The data at these addresses is not of significance for this project application. The text file contains the address and the operation that must performed for that address i.e. read or write, denoted by 'r' and 'w' respectively in the input text file.

Output:

The output from the cache simulator after the run intends to display a set of parameters. These parameters can be used to make a judgement about various aspects of the like the number of misses, hits etc. It can also be used to compare the simulation results and use it as a basis to optimize the cache.

The output to be displayed will be:

1. **Accesses**: Total number of times there was a load/store operation
2. **Reads**: Total number of times the data was read from the memory
3. **Writes**: Total number of times the data was written to the memory

4. **Cycles with cache**: The number of cycles required to during all the memory reads/writes
5. **Cycles without cache**: The number of cycles required if there was no cache and each time there would have been a need for memory success. For e.g. it will be equal to, (number of cycles for memory access) * (Number of total memory accesses)
6. **Stream-ins**: The total number of times there was a need to write a paragraph of data from memory to the cache
7. **Stream-outs**: The total number of times there was a need to write a paragraph of data from cache to the memory
8. **Misses**: The times the required data was not present in the cache; stream-in required
9. **Hits**: The times the required data was already present in the cache; no stream-in required
10. **Read Hits**: Number of times the data to be loaded was already present in the cache
11. **Write Hits**: Number of times the data to be written existed in the cache on request

Some of the other parameters which we intend to display are:

1. Cache set selected (0-1023)
2. Cache line selected (0-3)
3. Value of Tag (17 bits)
4. Status of Dirty bit (1/0)

In the due course of the project we plan to optimize the cache simulator specific to the application and some of the additional parameters to be displayed can be:

1. Number of Hits (with tiling)
2. Number of Hits (without tiling)
3. Number of Misses (with tiling)
4. Number of Misses (without tiling)

Sample:

A small sample cache input and output is as shown below:

r 0x1000 r 0x4f200 r 0x9d400 w 0x9d400 r 0x1008 r 0x4f840 r 0x9d400 w 0x9d400

Accesses	=	8
Reads	=	6
Writes	=	2
Cycles with cache	=	308
Cycles without cache	=	400
Stream-ins	=	5
Stream-outs	=	1
Misses	=	5
Hits	=	3
Read Hits	=	1
Write Hits	=	2

Line History:

Line History is a way to store and remember the last status of the cache lines. So, if there in any interruption, the cache will remember the status of LRU bits, the selected cache set, selected cache line, the data stored in the line, the operation performed; read/write, tag, dirty bit. All these functionalities can be achieved by creating a linked-list. Each node in linked-list will store the above-mentioned parameters; so, if the debug -d command is found in middle, the resumption takes

place latter from that point onwards. This can be achieved only if the history is kept tracked of, and this is precisely done by storing the parameters of the cache in the data structure.

Debug Commands:

The debug commands used for the simulator are -v, -d and -t. The following lines explains what each of these commands denote:

-v: Displays the version of the simulator design

-d: Displays the output parameters after running the simulation

-t: Displays the trace information

Internal Architectural Specification:

Driver module:

The driver module takes input from a ASCII text file and reads line by line for feeding input traces to the simulator.

It provides output to another ASCII text file by at first printing it in STDOUT and then associating the STDOUT with the text file which the driver module will open for providing all required final outputs.

The basic functionality of driver can be explained as:

Driver (input file)

- Scan the input file
- Look for characters like '-d', '-t','-v' and sets the program mode
- Reads the accesses and checks if its read (r) or write (w).
- Passes the address and command to cache

User interface:

The SW communicates input through reading a text file. User will enter the file name as 'trace' and output will be printed to stdout. And for output it prints it into STDOUT and associates it with a text file. The user gives input as text(.txt) file to the cache simulator, along with the debug commands specified in the same text file. The user has the option to debug, check the current version and view the trace information.

Input Format:

The following lines specify the format the cache simulator will take in as input.

1. The input to the simulator is through a text file which also includes the commands viz. -d, -t and -v. The input to the simulator is the type of access and the address of memory. The simulator differentiates the strings in input file with the space between type of access and the address.
2. Consider a sample input through a text file. The text file contains all the address which the cache must read from or write to. The data at these addresses is not of significance for this project application. The text file contains the address and the operation that must performed for that address i.e. read or write, denoted by 'r' and 'w' respectively in the input text file. As for address, it reads from 3rd character of hex address.
3. The inputs are hexadecimal addresses format e.g. 0x10dead3a. There are two types of input traces one for read operation and one for write operation.
    - The input for read operation is denoted as: r 0x10dead3a
    - The input for write operation is denoted as: w 0x10dead3a.

Output Format:

The output from the cache simulator after the run intends to display a set of parameters. These parameters can be used to make a judgement about various aspects of the like the number of misses, hits etc. It can also be used to compare the simulation results and use it as a basis to optimize the cache. The parameters displayed will be,

1. Cache access count (loads and stores)
2. Total no. of read access
3. Total no. of write access
4. Number of cycles consumed for all memory accesses
5. Total no. of machine cycles for memory accesses if architecture had no cache
6. No. of streams in operation
7. No. of streams out operation
8. Total cache hits
9. Total cache misses
10. Total read hits
11. Total write hits
12. Number of evictions

Some of the other parameters which we intend to display are,

1. Cache set selected (0-1023)
2. Cache line selected (0-3)
3. Value of Tag (17 bits)
4. Current status of Dirty bit (1/0).

Real Simulator Module:

We used object oriented approach to implement a L1 data cache simulator. This gives us better control over the data structure and makes design more modular.

We use the driver at the top of our hierarchy, which takes input file and feeds the data to simulator about the mode of operation. Once the mode of operation is fixed, the addresses and access are fed to cache. The cache is implemented as class. Cache class is fed with the address. There are two main methods implemented in side this class: read and write. These functions take the addresses and it separates its index bits to point to the appropriate set. Set is also a different class which is responsible to stream-in and stream-out the data from cache. It also evacuates the cache line, according the LRU bits set on the cache line. The set class monitors the counters for streaming the data and the LRU bits for the cache lines.

The cache class, implements two more functions which checks if it's a cache miss or cache hit for the given address. These functions monitor the counter for the number of cache cycles and total hits and misses. Inside the miss function, it checks if the set has any empty line or if the set is full. In case the set is already occupied it will use LRU policy to evacuate cache line.

The simulation model uses the write back policy by means of a dirty bit in cache line. It is set to 1 in every modification in a cache line associated with it if it is a write operation. If a line is dirty, then a write to main memory occurs during the time of evictions. The dirty bit is made dirty every time a write operation occurs (both hit and miss), and is set clean if a read-miss occurs, while nothing happens if a read-hit occurs.

Memory Allocation:

In the programming part, static memory allocation policy is used to allocate memory space. As the configuration of cache simulator is predefined we thought it's better to implement static allocation of reduce the complexity of code.

Simulator Program Structure:

The section describes the data structures used in the simulator for cache line, set class etc. Functions used in the simulator and their logical connection is explained in the simulator algorithm section.

1. Macros for different cache parameters. (tag bits, bytes per line, overhead bits etc.)
2. A structure to store the status of cache (different cycle counts, access counts, hit rate)
3. Cache class
   i) Read function
   ii) Write function
   iii) Check hit function
   iv) Check miss function
4. Set class
   i) Stream in function
   ii) Stream out function
   iii) Replace cache line function
   iv) Eviction function
5. Cache line class
6. Helping functions:
   i) Print cache status
   ii) Hex to binary conversion
   iii) Decimal to binary function
   iv) Get the index from address
   v) Driver function

Simulator Algorithm:

The main algorithm is the following:

1. Initialize cache
2. Open the trace file for reading

3. Create a new cache object
4. Read a line from the file
5. Parse the line and read or write accordingly
6. If the line is "#eof" continue, otherwise go back to step 4
7. Print the results
8. Close the file

Error Cases and their Handling:

The errors can occur at different stages. If the entered input file is an invalid/missing .txt file, then the software will throw an error to the user.

Another piece of error case that can occur is when the data in file is invalid. For example, the data in the file is not in the correct/expected format, say missing a -w or -r or an address. In such a case, again there should be an error displayed to the user to correct the arguments entered.

These cases can be handled easily using an if…else set of statements for each of the error cases.

Test Cases:

The cache simulator will be tested using the sample set of test cases provided in the specification documentation and the Homework1 for ECE-586(Winter'17) trace file.

Macro Definitions:

Macros are used to define the parameters which will remain constant throughout the simulator.

`#define BYTES_LINE 32`	Defines number of data bytes per line
`#define OVERHEAD 21`	Defines the number of overhead bits
`#define TOTAL_LINES 4096`	Defines total number of lines
`#define MISS_COST 50`	Defines number of cycles for stream-in
`#define ACCESS_COST 1`	Defines number of cycles for each access
`#define SETS 1024`	Defines the number of sets
`#define LINES 4`	Defines number of lines per set
`#define TAG 17`	Defines the number of tag bits use

Read Algorithm:

The read algorithm is defined to read the data from the cache. A brief outline of can be summarized as follows:

1. On encountering a ‘r’-read in the input, the simulator transfers control to the read algorithm function.
2. The first step is to convert the number from hexadecimal format to binary and retrieve the details related to that particular set. Once, these details are available, we need to check if there is hit or miss for that particular address.
3. Now, if there’s a hit then just increment the number of read hit counts and continue to the next step, as the read for that address is complete.
4. If there is a ‘miss’ then there will be a check done to calculate if any line is free and available. If any line is available, then just stream-in the new required data from that address. The other case, is that there is a ‘miss’ but not a single line is available.
5. In such a case, another algorithm is to implemented to first clear a cache line. This is called, line eviction policy. So, once a line is evicted i.e. streamed out, the new data is written at that place. In this case, there is another counter denoting the number of memory access which is incremented.
6. In either case of a miss the miss counter is incremented to keep a track of number of misses.
7. Once success, step out of the read operation.

Write Algorithm:

The write algorithm is defined to write data to cache. It can be summarized as follows:

1. On encountering a ‘w’-write in the input; the simulator transfers control to the write algorithm function.
2. Like the read operation, the first step is to convert the number from hexadecimal format to binary and retrieve the details related to that particular set. Once, these details are available, we need to check if there is hit or miss for that particular address.
3. Now, if there’s a hit then just increment the number of write hit counts and continue to the next step, as the write for that address is complete.
4. If there is a ‘miss’ then there will be a check done to calculate if any line is free and available. If any line is available, then just stream-in the new required data from that address. The other case, is that there is a ‘miss’ but not a single line is available.
5. In such a case, another algorithm is to implemented to first clear a cache line. This is called, line eviction policy. So, once a line is evicted i.e. streamed out, the new data is written at that place. In this case, there is another counter denoting the number of memory access which is incremented.
6. In either case of a miss the miss counter is incremented to keep a track of number of misses. Once success, step out of the write operation.

Extension to 64-bit machine:

To extend the current machine to 64-bits following thing needs to be considered,

1. Assuming all the other parameters to remain the same, on a 64-bit address machine the number of tag bits will increase increasing the total number of overhead bits. Thus, the overall memory size of cache will increase.
2. On a 64-bit machine the hardware complexity is more compared to a 32- bits machine. Increase in the number of transistor will increase the overall size of system.

Limitation of Cache Simulator

The main limitation of our cache simulator is that it is a static model. Parameters like the number of sets, line size, tag size, etc. are hardcoded into the program. The user is not allowed to design cache of his own configuration. Also, the design is as per the requirement mentioned in the handout which means the policies that govern the read and write operations are predefined and simulation is performed in accordance to these rules.

Size in Bits:

Please refer to section Design Parameters, Page No. 4

Function and Timing:

Please refer to section Function and Timing, Page No. 7

Sample Input and Output:

Sample#1:

Input:

```
-d
w 0x080      w 0x088      w 0x0a0      r 0xf0000    r 0x080
r 0x088      r 0x0f0      r 0x0a0
```

Output:

```
Debug command recorded.


Statistics:
Access count:7 (read count:4, write count:3)
Cycles with cache: 147
Cycles without cache: 350
Stream in count: 7
```

```
Stream out count: 0
Replacement count: 0
Miss count: 7
Hit count: 0 (read hit: 0, write hit: 0)
Write back count: 0

Debug Information:
Set:0   Line:0 Dirty:1       Valid:1        LRU:0   Tag:11110000000000000
```

Sample#2:

Input:

```
w 0x00044220 w 0x00044440 w 0x00044240 w 0x000ffff0
r 0x00044220 r 0xff000000 r 0x00044440 r 0x00044240
```

Output:

```
Statistics:
Access count:8 (read count:4, write count:4)
Cycles with cache: 168
Cycles without cache: 400
Stream in count: 8
Stream out count: 0
Replacement count: 0
Miss count: 8
Hit count: 0 (read hit: 0, write hit: 0)
Write back count: 0
```

Sample#3:

Input:

```
w 0xff00ff00 w 0xff00ff20 w 0xff00fff0 w 0xff00ffff w 0xff000000
w 0xff00000f w 0xff000080 w 0xff0000f0 w 0xff000084 w 0xff0000ff
-d -t
r 0xff00ff00 r 0xff00ff0f r 0xff00ff00 r 0xff00ff0f r 0xff000084
r 0xff000086 r 0xff000088 r 0xff00ff00 r 0xff00ff20 r 0xff00ff00
r 0xff00ff20 r 0xff00fff0 r 0xff00ffff r 0x12345678
```

Output:

```
reading from address: 0xff00ff00
reading set 32...
R missed, space available, stream in
reading from address: 0xff00ff0f
reading set 57...
R missed, space available, stream in
Debug command recorded.
Trace command recorded.

Statistics:
Access count:24 (read count:14, write count:10)
Cycles with cache: 504
Cycles without cache: 1200
Stream in count: 24
Stream out count: 0
Replacement count: 0
Miss count: 24
Hit count: 0 (read hit: 0, write hit: 0)
Write back count: 0

Debug Information:
Set:32 Line:0 Dirty:1       Valid:1       LRU:0   Tag:11111111000000010
Set:57 Line:1 Dirty:1       Valid:1       LRU:0   Tag:00010010001101000
```

INDEX

Bibliography

- Lecture Notes of Dr. Herbert Mayer
- http://www.ccs.neu.edu/course/com3200/parent/NOTES/cache-basics.html
- https://www.udacity.com/course/viewer#!/c-ud007/l-1025869122/m-1007830043
- http://web.cecs.pdx.edu/~herb/ece585f16/
- https://courses.cs.washington.edu/courses/cse378/07au/.../L18-Cache-Wrap-up.ppt
- https://courses.cs.washington.edu/courses/cse461/13au/lecture_slides/Dec_4.pptx
- https://www.d.umn.edu/~gshute/arch/cache-addressing.xhtml
- https://web.cs.iastate.edu/~prabhu/Tutorial/CACHE/interac.html