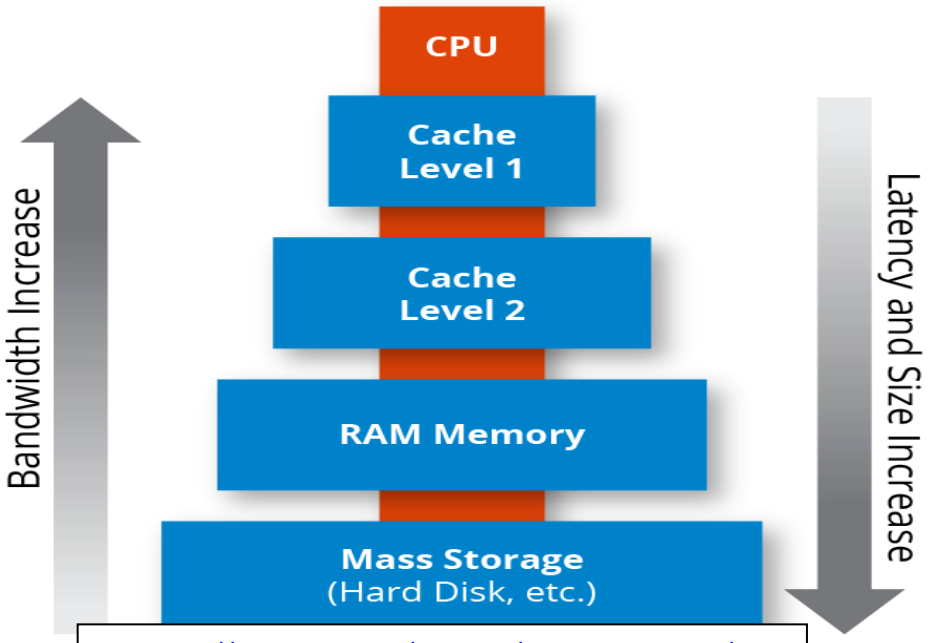
DOCUMENTATION OF CACHE



Src:https://hazelcast.com/glossary/memory-caching/

This program is designed to feel how a cache works in real hardware. It is just a demo to show how cache works in a standalone environment without the involvement of the main memory, also how the cache varies with different type of mapping.

For bonus assignment, We have designed 2 level cache which will help us in understanding how different mapping work in multi-level cache.

For bonus task, a separate python file is made to do the required job. Input format, assumptions, errors, constraint and all function are same. The two main differences are that the bonus has an extra function to write in direct mapping and how the table is printed. The procedure of each process is deeply explained.

Requirements:-

1) Python 3.7+
2) Libraries needed
   - Math
   - Queue
   - Sys
   - Tabulate

Assumptions & Points to remember:-

1) Initially, all words in the cache and main memory are Empty, i.e. “None” in python.
2) The cache will be running on a 32-bit system, which means the size of memory address and size of a word is 32 bit.
3) Since sys.sizeof(data) in python doesn’t return the exact size of data and depends on how information is called. Therefore, I have taken 35 bit(in python) as maximum for word.

   For more information on data allocation in python open the link below
   https://code.tutsplus.com/tutorials/understand-how-much-memory-your-python-objects-use--cms-25609
4) Any type of data can be stored given that it’s size is less than 35 bit as allotted by python.
5) S, i.e. size of the cache can be calculate using CL(Number of cache lines) and B (Block size, i.e. Number of words in a block). That’s why it is skipped.
6) Assuming each and input is in the power of two.
7) All type of mapping will go side by side, i.e. at any instance, you can get final standing of all three caches after one-another.
8) Constraints on input are defined below.
9) To write in 2 level cache, “Write through” method is used.
10) Since there is no main memory, if data is overwritten in the cache, it will be lost forever.
11) In case of the associative mapping Tag is divided in two part for better understanding and relating it with direct mapping

Input

Order of input from User is as follows:

*"S will be equal to no. of line * No. of block * Size of 1 block(32 bit in this case)'*

-This statement is printed for User to get an idea while using the application.

- "*Number of the cache line*:", i.e. CL
- "*Block size*:" *i.e. B*
- "*N for n-way set associative memory*:", *i.e. N*

```
S will be equal to no. of line * No. of block * Size of 1 block(32 bit in this case)
No. of cache lines:16
Block size :4
N for n-way set associative memory4
```

These three inputs will be processed, and User will be asked to enter an integer ranging from 1 to 4, with information printed as shown below

```
Type 1 to read
Type 2 to write
Type 3 to print Caches
Type 4 to exit
```

"Type 1 to read

Type 2 to write

Type 3 to print Caches

Type 4 to exit"

- Command ranging from 1-4, for further processing

If command ==1	If command == 2	If command == 3	If command == 4
○ “*Address*:”  To read data from memory	○ “*Address*:” ○ “*Data*:” To store given data into provided memory address	To print all caches	To close program

Constraint

- CL and B must be integers of the power of 2 with power >= 1.
- N must lie between Cl/2 and CL
- All address input must be 32 bit.

Errors

The order of errors is in the sequence of possible occurring.

- If S, CL or N is given as negative:

```
“ Please enter positive integer."
```

- If S, CL or N is given as in any other format than:

```
" Please enter integer "
```

- If Cl is not in the power of 2:

```
" Number of cache lines must be in power of 2 "
```

- If B is not in the power of 2:

```
" Block size must be in power of 2 "
```

- If N doesn’t lie in between 1 and CL:

```
" N must lie between 1 and Cl "
```

- If Cl or B is strictly less than 2

```
" CL and B must be greater than or equal to 2 "
```

- If the command doesn’t lie between 1 and 4

```
" Please enter integer between 1 and 4 "
```

- If the address given is in any other form:

```
" Please enter integer between 1 and 4 "
```

- If the length of the address is not 32 Bit

```
"Address must 32 bit "
```

- If the given data is not under 32 bit

```
"Data should be under word limit i.e. 32 bit."
```

Output

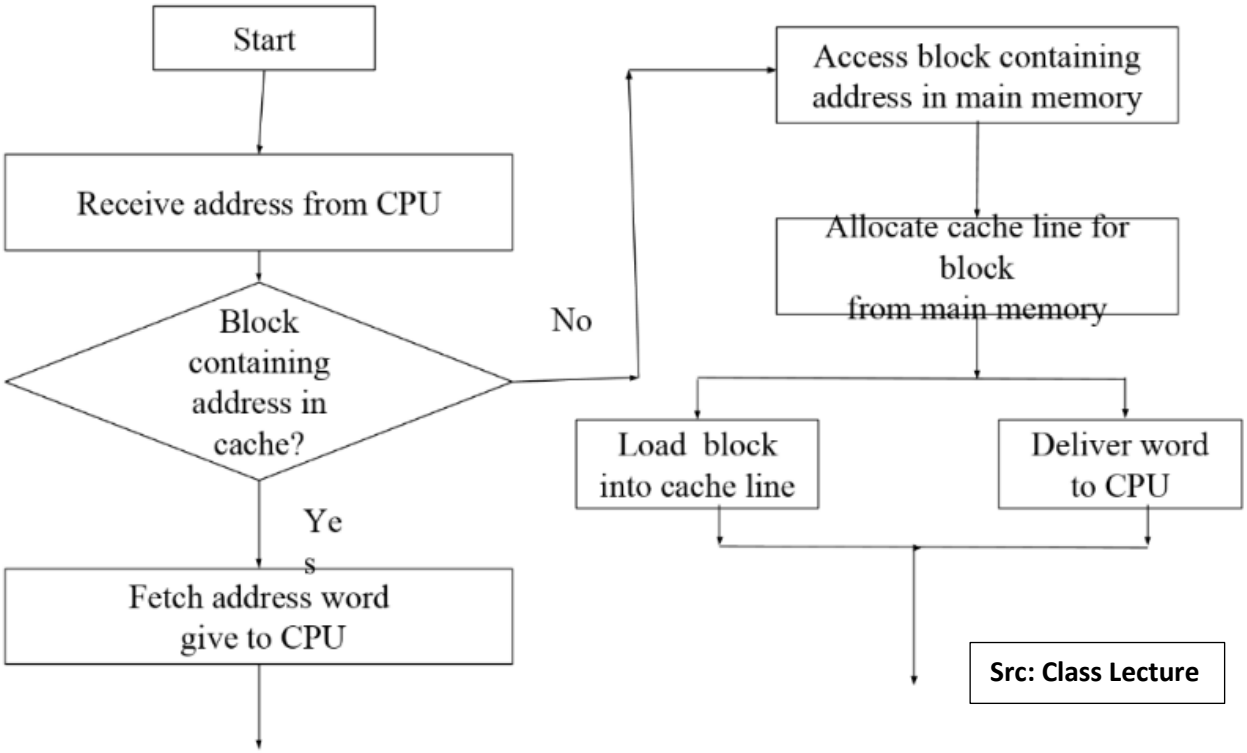
The program will show the following outputs depending on inputs.

- The first output will be statement below:

(To inform the User)

```
"S will be equal to no. of line * No. of block * Size of 1 block(32 bit in this case)")
```

While reading



Src: Class Lecture

If data is found in cache

```
" Hit "
```

If data is not found & also cache block is empty

```
"Address not found "
```

If data is not found But cache is not empty

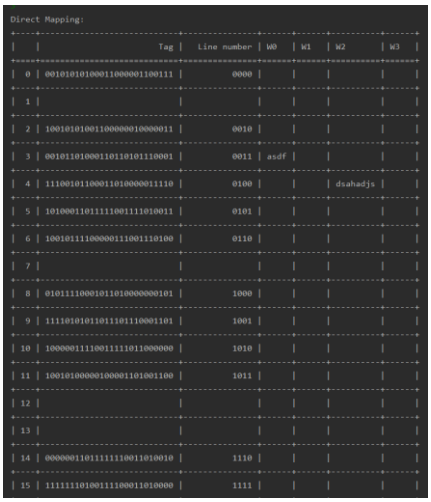
```
"Current data is replaced by data at " + address
```

In case of multi-level cache, Hit is replaced by “Hit at * level”, “Address not found ” is replaced by "Address not found at level 1 and 2 both" (in Direct mapping); ‘ "Address not found " (Cache number 1)’, ‘ “Address not found " (Cache number 2)’ (in Set associative), depending level and "Address not found" (in Associative Mapping).

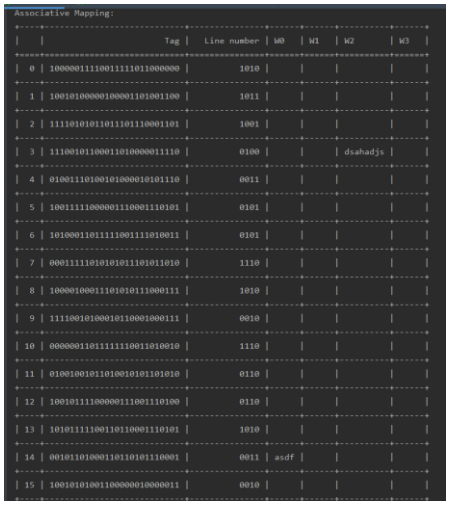
As far as replacement is considered, It remains the same.

Final cache structure

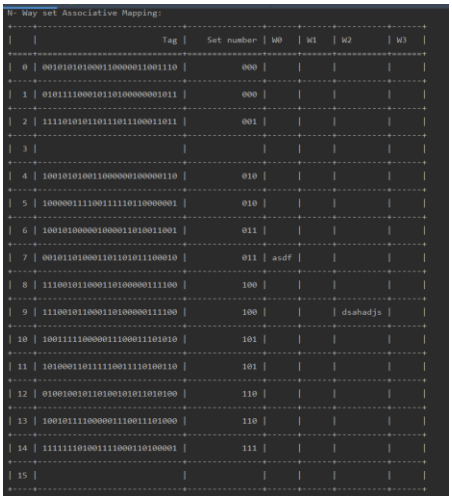
All three type of mapping will include a Tag column followed line number column(except Set number in case of N-way set associative), both in binary. After these two column we have B number of words column representing a block in row.



Direct

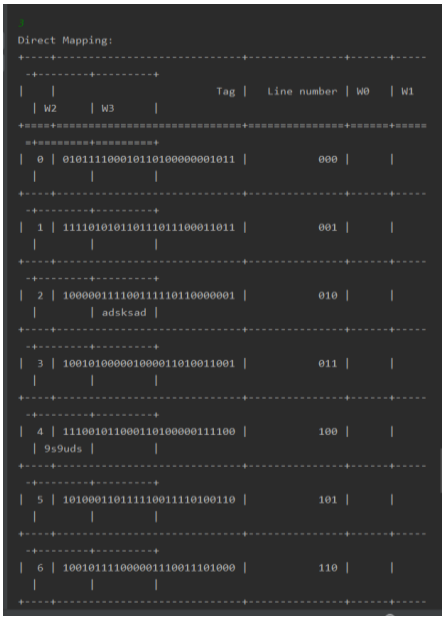


Associative



Set associative

If you see anything like figure on the right , it means that size of command prompt is not big enough. Try to maximize it or zooming out



There is no specific output while writing both in single level or multi-level cache.
Also the output format for multi-level cache is same except that instead of one 2 tables will be printed for each of mapping

Direct Mapping (multi-level)

Cache level 1

	Tag	Line number	W0	W1	W2	W3
0	010111100010110100000001011	000				
1	111101010110111011100011011	001				
2	100000111100111110110000001	010				
3	100101000001000011010011001	011				
4	111001011000110100000111100	100			dsahadjs	
5	101000110111110011110100110	101				
6	100101111000001110011101000	110				
7	111111101001111000110100001	111				

Cache level 2

	Tag	Line number	W0	W1	W2	W3
0	001010101000110000011100111	0000				
1						
2	100101010011000000100000011	0010				
3	001011010001101101011110001	0011	asdf			
4	111001011000110101000011110	0100			dsahadjs	
5	101000110111110011110101011	0101				
6	100101111000001110011110100	0110				
7						
8	010111100010110101000000101	1000				
9	111101010111011101110001101	1001				
10	100000111100111111011000000	1010				
11	100101000001000011010011100	1011				
12						
13						
14	000000110111111110011010010	1110				
15	111111101001111000110100000	1111				

Associative Mapping(multi-level)

N- Way set Associative Mapping:

Cache level 1

	Tag	Set number	W0	W1	W2	W3
0	111001011000110100000111001	00			dsahadjs	
1	010111100010110100000010110	00				
2	111101010110111011000110110	01				
3	101000110111110011101001101	01				
4	100101010011000000100000011100	10				
5	100000111100111110100000010	10				
6	001011010001101101011100100	11				
7	100101000001000011010110010	11				

Cache level 2

	Tag	Line number	W0	W1	W2	W3
0	100000111100111110110000000	1010				
1	100101000001000011010011100	1011				
2	111101010110111011100011101	1001				
3	111001011000110100000111110	0100			dsahadjs	
4	010011101001010000010101110	0011				
5	100111110000011100011110101	0101				
6	101000110111110011110010011	0101				
7	000111110101010111010111010	1110				
8	100001000111010101111000111	1010				
9	111100101000010110001000111	0010				
10	000000110111111110011010010	1110				
11	010010010110100101011010010	0110				
12	100101111000001110011110100	0110				
13	101011111001101110001110101	1010				
14	001011010001101101011100001	0011	asdf			
15	100101010011000000100000011	0010				

Set Associative Mapping

(multi-level)

Associative Mapping:

Cache level 1

	Tag	Line number	W0	W1	W2	W3
0	100000111100111110110000001	010				
1	100101000001000011010011001	011				
2	111101010110111011100011011	001				
3	111001011000110100000111100	100			dsahadjs	
4	100101111000001110011101000	110				
5	101011111001101100011101011	010				
6	001011010001101101011100010	011	asdf			
7	100101010011000000100000110	010				

Cache level 2

	Tag	Set number	W0	W1	W2	W3
0	001010101000110000011001110	000				
1	010111100010110100000001011	000				
2	111101010110111011100011011	001				
3						
4	100101010011000000100000110	010				
5	100000111100111110110000001	010				
6	100101000001000011010011001	011				
7	001011010001101101011100010	011				
8	111001011000110100000111100	100			dsahadjs	
9						
10	100111110000011100011101010	101				
11	101000110111110011110100110	101				
12	010010010110100101011010100	110				
13	100101111000001110011101000	110				
14	111111101001111000110100001	111				
15						

Functions:-

- **check_bin(value):-** This function takes address(value) as input in the form of a string and check whether it is binary or not, Providing a Bool output as follows:-

  Return type - Bool

  True – For binary

  False -Input is not binary

- **checkinput(input):-** This function again takes address(input) as input and calls check_bin . Along with this, it also checks whether the input is 32-bit binary or not.

  Return Type – Bool

  True if everything is okay

  Else False

- **check():-** This function is responsible for running the whole code. All the required constraints on S, CL, B & N are checked using this function.

  Return Type – Bool

  True if everything is okay

  Else False

- **linenum(num):-** This function takes given address to extract the block number in which they are required to be stored in the cache in case of direct mapping. And in case of associative mapping, it is used to find line number in which it should be stored in direct mapping to compare easily

Return Type – int

An integer which represents block number in the cache

- **blocknum(num):-** This function takes given address to extract the word number in the selected block. This is used in direct as well as associative mapping.

  Return Type – int

  An integer which represents word number in required block

- **directMappingR(address):-** This function requires an address to do the process using the process of direct Mapping. It doesn't require any special parameter, as it uses a global variable and above described function. This function checks whether the given address lies in cache keeping direct mapping in mind and completing the task accordingly.

  Return Type – void

- **search(arr, input):-** This function takes an array(arr) and address(input) as inputs, to check whether it exists in cache or not. This function is used in associative mapping to prevent the repetition of blocks.

  Return Type – int

  Positive Integer/zero which represents block number in the cache

  -1 in case that address block doesn't lie in cache

- **associaativeMappingR(address):-** This function requires the address to do the process using the process of

Associative Mapping. It doesn't require any special parameter, as it uses the global variable like Direct Mapping and above described function. This function checks whether the given address lies in cache keeping Associative Mapping in mind and completing the task accordingly.

Return Type – void

- **associaativeMappingW(address):-** This function takes an address and data as inputs. It works in a similar manner as associaativeMappingR, but instead of just reading, it also writes data in the required address in cache.

  Return Type – void

- **searchNway(address,setnum):-** This function is used in Set Associative Mapping to search given address in a specific set.

  Return Type – int

  Positive Integer/zero which represents required block number in the cache -1 in case that address block doesn't lie in that Set.

- **getSetNumber(string):-** This function takes address(string) as an input to find the Set number in which the given address must lie for Set Associative Mapping.

  Return Type – int

  Positive Integer/zero which represents required Set number in the cache

- **NWaySetAssociativeMappingR(address):-** This function requires the address to do the required task using the process of Set Associative Mapping. It doesn't require any special parameter, as it uses the global variable like other two mappings and above described new function. This function checks whether the given address lies in the required Set in cache keeping Set Associative Mapping in mind and completing the task accordingly.

  Return Type – void

- **blocknumA(num):-** This function takes given address to extract the word number in the required block. This is used in Set Associative mapping.

  Return Type – int

  Integer - which represents word number in the required block.

- **NwaySetassociativeMappingW(address, data**):- This function takes an address and data as inputs. It works in a similar manner as NWaySetAssociaativeMappingR, but instead of just reading, it also writes data on required address in cache.

  Return Type – void

These are the function in a single Level cache program. For multi-level cache, all functions are the same in functionality. The only difference is that independent functions are named as function1 and function2 (where function represent above-explained functions except check_bin, check, checkinput, search). The other main functions have just been edited to make them compatible with multi-level caching. The only new

introduced function is **directMappingW(address, data)** which is used to write given data on caches following the required conditions, rest everything is same as associativeMappingR.

The global variables are declared in the beginning, to use from anywhere.

The above Screenshot which are used have the following input

```
16
4
2
1
00101010100011000001100111000010
1
10011010011110011011101010101010
1
11111110100111100011010000111101
1
01011110001011010000000101100010
1
01001110100101000010101110001101
1
10011111000001110001110101010111
1
10100011011111001111010011010111
1
00011111010101011101011010111011
1
10000100011101010111000111101001
1
11110010100010110001000111001011
1
00000011011111110011010010111001
1
01001001011010010101101010011000
1
10010111100000111001110100011010
1
10101111100110110001110101101011
1
00101101000110110101110001001100
1
10010101001100000010000011001000
1
10000011110011111011000000101011
1
10010100000100001101001100101101
1
11110101011011101110001101100110
1
11100101100011010000011110010010
2
11100101100011010000011110010010
dsahadjs
2
00101101000110110101110001001100
```

```
asdf
1
10010101001100000010000011001000
1
10000011110011111011000000101011
```