ASSIGNMENT 12 REPORT

DIWAKAR PRAJAPATI
2018CS10330

*Give input in a file named "**config.txt**".*
To run the program type in terminal:

```
g++ cache.cpp
./a.out
```

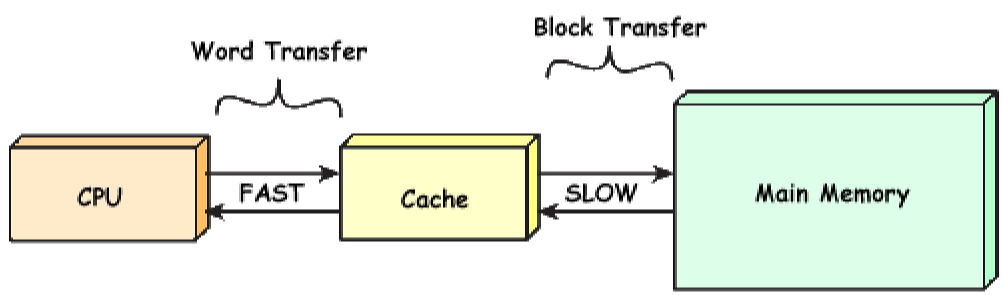
There are two output files:

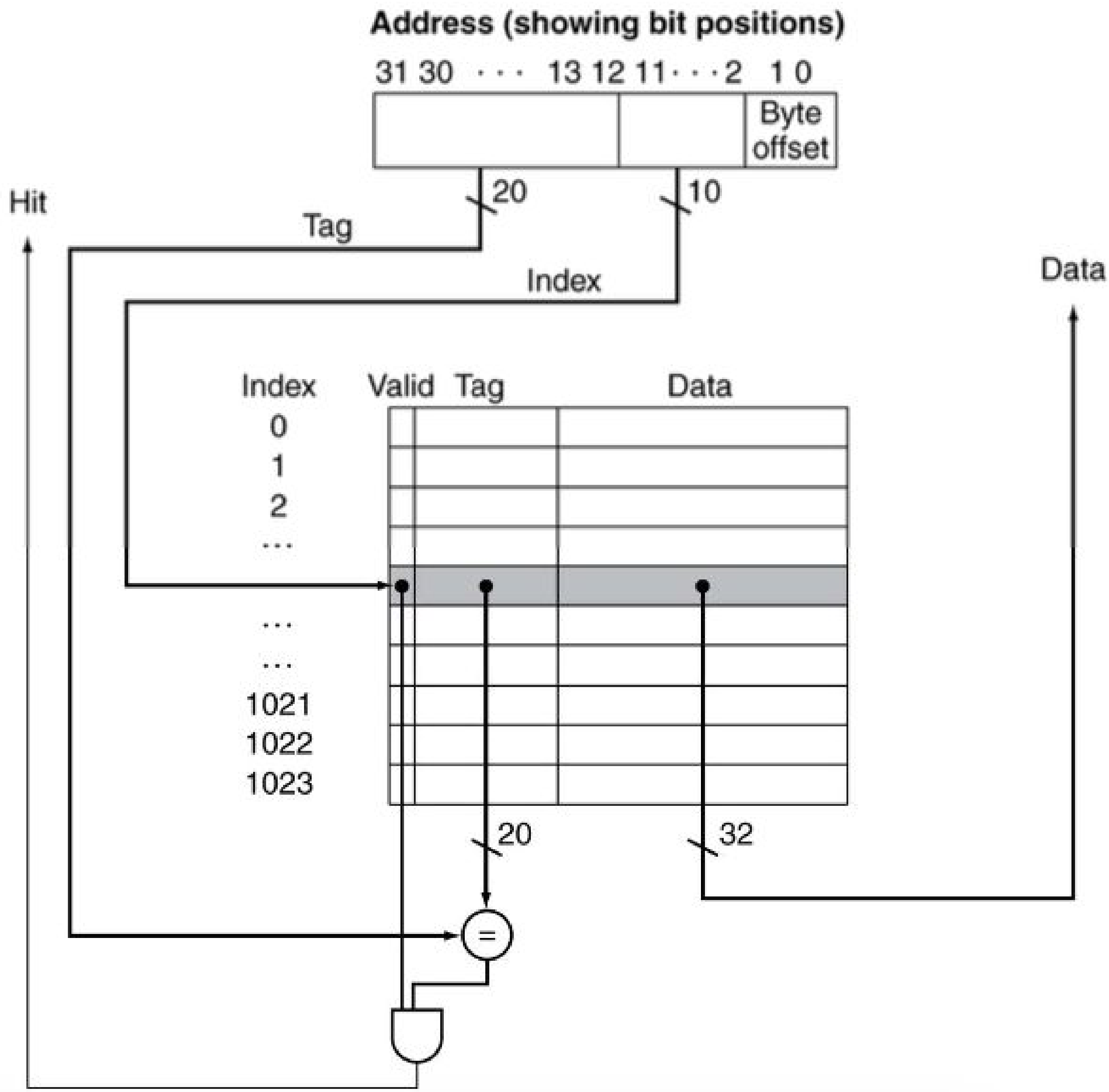
- *"**output.txt**" : Contains the final cache status along with other details of access, hit and miss.*
- *"**details.txt**" : Contains a detailed simulation of the cache, showing each and every action.*

AIM: Simulation Software for cache memory:

Introduction:

Cache Memory acts as a bridge in between Main Memory and the CPU. Since the main memory is very large, searching some data at a particular address in main memory takes too much time. So, we introduce cache. Cache acts as a short term memory, which stores addresses which have been accessed recently and frequently.





Address (showing bit positions)
31 30 · · · 13 12 11 · · · 2 1 0
Byte offset
20
10
Hit
Tag
Index
Data
Index
Valid
Tag
Data
0
1
2
...
...
...
1021
1022
1023
20
32
=

Assumptions:
1: The given address is byte addressable.
I am assuming that the given address is the actual address of the main memory.
Address has three parts: **TAG | INDEX | BYTE OFFSET**

2: I have not made the main memory since that was not our primary motive. So in the case of MISS, I am filling random data in the cache. Until I evit that block, I will get consistent data for that corresponding block address, but once the block is evicted and because we don't have main memory for storage so we would get different data when we would fetch the same address.

3: I have not allotted each set equally for HIGH and LOW priority, instead I have used a single bit which differentiates its priority, i.e. 0 for low priority and 1 for high priority. So the high priority and the low priority group sizes are not fixed, it can have all the blocks at high priority and it can all have all the blocks at low priority, depending on the instructions. So the size of the LOW Priority group and the HIGH Priority group is variable during the course instruction execution.

I have used this method so that, like a particular set of addresses which belong to the same address are accessed again and again (assuming T to be large enough) then all the blocks can be at high priority at the same time.

This increases the efficiency, like if we divide the set equally there may be a chance to remove a high priority block although if it is accessed again and again.

Inside a particular priority I am using Least Recently Used Policy (LRU). For this I have used an integer associated with each block which keeps an account of the last access to that particular block, based on this I am replacing the least recently used block.

4: When I change the valid bit to invalid, the dirty bit remains unchanges.

5: While evicting I am just over writing the new data and no change in valid or dirty bit takes place.

6: I am using the write back concept, i.e. in case of writing I write only to cache and set dirty bit to one, and while evicting I am going to write into the main memory if the dirty bit is 1.

Design Detail:
A cache is divided into sets(which have counts of power of 2 like, 1, 2, 4, 8, 16...).
Each set has some blocks which is equal to the associativity of the cache, which means how many blocks are associated to each set of the cache.
Each block has an array of datas.
While reading I check if there is any valid block already present which has the tag same as the given address, then it is a READ_HIT and I return the data from the data array of the block using the byte offset. If it is not found then it is READ_MISS, in this case I am generating a random array for the data array into that cache block. This random generation of array is analogous to reading from main memory. After this valid bit of that becomes one.

While writing I check if there is any valid block already present which has the tag same as the given address, then it is a WRITE_HIT and then I write into the data array of that block and use the byte offset. If it is not found then it is WRITE_MISS, in this case I am generating a random array for the data array into that cache block, this is analogous to bringing the block from the main memory and then writing the data into it and then writing the block into the cache. Also the valid bit becomes 1 and dirty bit becomes 1.

Implementation Detail

1: The block structure : This is a block or line,

```
struct block{
	int tag, last_recent_access; // tag -> tag of a block
	bool valid, dirty;    /Valid and dirty bit
	int priority;         //Priority bit
	int block_size;       // Stores the block size = size of data vetor
	bool first_access;    // True initially, becomes false after the first access.
	vector<int> data;     // For storing the data corresponding to this block.
};
```

2: The cache-set structure : It is a set of a block

```
struct cache_set{
	vector<block> _set;
};
```

3: Decimal to Binary and Binary to Decimal functions:

```
int todecimal(string s){ . . . }
string tobinary(int n){ . . .}
```

4:Initialize Cache: It initializes tag with -1, valid with 0, dirty with 0, data with a data array of ZEROs.

```
void initialize_cache(vector<cache_set> &cache, int associativity, int block_size, int no_of_set){. . .}
```

5: Print Cache Statistics: Print stats in a beautiful format.

```
void print_cache(vector<cache_set> &cache, int associativity, int no_of_set, int flag){ . . }
```

6: Read Data : Return the block’s data to the corresponding address, which may be then used by the comparator to extract the exact data using the byte offset.

```
vector<int> read_data(vector<cache_set> &cache, int address, int no_of_bits_in_byte_offset, int no_of_set,
int associativity, int block_size){ . . . }
```

7: Write Data: Given data and an address, writes data to that address in the cache.

```
void write_data(vector<cache_set> &cache, int address, int data, int no_of_bits_in_byte_offset, int
no_of_set, int associativity, int block_size){ . . . }
```

8: Convert High to Low Priority: Checks if a valid HIGH Priority block has not been accesses for T accesses the it makes the block into LOW Priority.

```
void update_outdated(vector<cache_set> &cache, int no_of_bits_in_byte_offset, int no_of_set, int
associativity, int T){ . . . }
```

9:Main function: CPU of my Cache Simulation. Handel’s all function calls.

```
int main(){ . . . }
```

Test Cases:

All the test cases are present in folder name "**testcases**".

There are a total of 30 test cases, including a variety of cache combinations, various types of address accesses.

Test Case1: Sample input in the assignment description.

Cache Size: 16
Block Size: 2
Associativity: 2
T = 4

Test Case2: Cache_Size = 1024
Block Size = 32
Associativity = 4
T = 10
50 accesses.
Main Memory size = 1000
Since we have a large cache, also large block size so there is a high hit ratio of 0.71

Test Case3: Cache_Size = 64
Block Size = 32
Associativity = 1 T = 5
50 accesses.
Main Memory size = 500
Large block size so there is a very high hit ratio of 0.96.

Same instructions for Test Case 4 - 9

Test Case4: Cache_Size = 16 T = 1
Block Size = 2 Associativity = 2
500 accesses.
Main Memory size = 100
We have small cache, small block size, and low associativity
Hit ratio is very small = 0.144

Test Case5: Cache_Size = 64 T = 4
Block Size = 2 Associativity = 2
500 accesses.
Main Memory size = 100
We have average sized cache, small address, hit ratio is descent = 0.592

Test Case6: Cache_Size = 64 T = 4
Block Size = 8 Associativity = 4
500 accesses.
Main Memory size = 100
Same specifications as of previous, except block size increased, hit ratio = 0.602
Not much effect on hit ratio.

Test Case7: Cache_Size = 64 T = 4
Block Size = 16 Associativity = 4 (Full Associativity)
500 accesses.
Main Memory size = 100
Same specifications as of previous, except block size increased, hit ratio = 0.6
No much effect seen.

Test Case8: Cache_Size = 64 T = 4
Block Size = 8 Associativity = 8
500 accesses.
Main Memory size = 100
Associativity increased, hit ratio = 0.608
Not much effect

Test Case9: Cache_Size = 64 T = 4
Block Size = 8 Associativity = 1 (Single Associativity)
500 accesses.
Main Memory size = 100
Associativity decreased, hit ratio = 0.634
Little increase in hit ratio.

Same instructions for Test Case 10 - 11

Test Case10: Cache_Size = 64 T = 16
Block Size = 8 Associativity = 4
500 accesses. Inst_access_prob = 80%
Main Memory size = 1000
Associativity increased, hit ratio = 0.838
Because, inst_access_prob = 80%, so hit probability increased

Test Case11: Cache_Size = 64 T = 16
Block Size = 2 Associativity = 4
500 accesses. Inst_access_prob = 80%
Main Memory size = 1000
Associativity increased, hit ratio = 0.576
Reducing block size reduced the hit ratio.

Test Case12: Cache_Size = 32 T = 100
Block Size = 4 Associativity = 4
500 accesses. Inst_access_prob = 20%
Main Memory size = 100
Hit ratio = 0.48

Same instructions for Test Case 13 - 1

Test Case13: Cache_Size = 64 T = 4
Block Size = 8 Associativity = 4
500 accesses. Inst_access_prob = 80%
Main Memory size = 1000
Hit ratio = 0.806

Test Case14: Cache_Size = 64 T = 4
Block Size = 8 Associativity = 4
500 accesses. Inst_access_prob = 80%
Main Memory size = 1000
Hit ratio = 0.806

Test Case15: Cache_Size = 16 T = 4
Block Size = 4 Associativity = 2
500 accesses. Inst_access_prob = 20%
Main Memory size = 1000
Hit ratio = 0.227

Test Case16: Cache_Size = 32 T = 50
Block Size = 4 Associativity = 4
1000 accesses. Inst_access_prob = 60%
Main Memory size = 100
Hit ratio = 0.608

Test Case17:	Cache_Size = 16	T = 20
	Block Size = 2	Associativity = 4
	2000 accesses. Inst_access_prob = 50%	
	Main Memory size = 1000	
	Hit ratio = 0.5035	
Test Case18:	Cache_Size = 16	T = 20
	Block Size = 2	Associativity = 4
	2000 accesses. Inst_access_prob = 50%	
	Main Memory size = 1000	
	Hit ratio = 0.5035	
Test Case 19:	Cache_Size = 256	T = 50
	Block Size = 16	Associativity = 4
	3000 accesses.	
	Main Memory size = 50000	
	Hit ratio = 0.006	
	VERY LOW HIT RATIO	

Same instructions for Test Case 20 - 21

Test Case 20: Cache_Size = 16 T = 4
Block Size = 1 Associativity = 1
500 accesses.
Main Memory size = 100
Hit ratio = 0.146

Test Case 21: Cache_Size = 16 T = 4
Block Size = 1 Associativity = 8
500 accesses.
Main Memory size = 100
Hit ratio = 0.158

Test Case 22: Cache_Size = 16 T = 4
Block Size = 1 Associativity = 8
500 accesses. Inst_access_prob = 90%
Main Memory size = 1000
Hit ratio = 0.696

Test Case 23: Cache_Size = 32 T = 10000
Block Size = 4 Associativity = 8
500 accesses.
Main Memory size = 1000
Hit ratio = 0.638

Test Case 24: Cache_Size = 32 T = 4
Block Size = 4 Associativity = 8
500 accesses. Inst_access_prob = 90%
Main Memory size = 1000
Hit ratio = 0.912

Test Case 25: Cache_Size = 1024 T = 10
Block Size = 64 Associativity = 8
5000 accesses.
Main Memory size = 10000
Hit ratio = 0.1066

Test Case 26: Cache_Size = 1024 T = 10
Block Size = 64 Associativity = 8
5000 accesses. Inst_access_prob = 90%
Main Memory size = 1000
Hit ratio = 0.9968

Test Case 27: Cache_Size = 1024 T = 10
Block Size = 64 Associativity = 8
5000 accesses. Inst_access_prob = 90%
Main Memory size = 1000
Hit ratio = 0.9054

Test Case 28: Cache_Size = 128 T = 10
Block Size = 16 Associativity = 8
5000 accesses. Inst_access_prob = 20%
Main Memory size = 1000
Hit ratio = 0.9984

Test Case 29: Cache_Size = 1024 T = 10000
Block Size = 64 Associativity = 8
5000 accesses. Inst_access_prob = 20%
Main Memory size = 10000
Hit ratio = 0.2842

Test Case 30: Cache_Size = 1024 T = 10
Block Size = 64 Associativity = 8
5000 accesses. Inst_access_prob = 50%
Main Memory size = 10000
Hit ratio = 0.5472

Conclusion:

We can conclude that neither the single associative nor the full associative is the most efficient. Something in between gives higher performance.

Increasing the cache size increases the hit ratio, we cannot increase it too much, since then our motive of introducing the cache would be demolished.

There are lots of tradeoffs in the cache configurations. Obviously, one performing better in one type of instruction set may not have reasonable performance in a different instruction set.