COMPUTER ORGANIZATION

CACHE PROJECT

ONE LEVEL

AND

TWO LEVEL CACHE

SUBMITTED BY

DOLLY SIDAR(2019304)

One Level Cache

ASSUMPTION

- ❏ Word Length = 32 bits(4 bytes)
- ❏ Data input is String (not more than 32 bit)
- ❏ At the time of cache miss, a block of memory will load in a cache or replace it with an existing block in that line. (According to cache theory, at the time of cache miss, a block of memory is loaded from Main memory, and If data is written to the cache, at some point, it must also be written to main memory. All these changes will happen at the time of read and write. In my program, I am showing only cache memory changes and assuming that there are also some changes in the main memory. So concerning these changes, the next searching and loading will perform.)
- ❏ I am printing the cache structure with the block address location stored.
- ❏ There is no limit in the address input(it can be 32, 16, 20,24...bits.)CPU generates addresses according to the size of the physical address space, but here I am only focusing on cache memory, not taking the main memory size as an input. So user input addresses according to valid main memory size.(for example in 4GB main memory , CPU generates addresses of 32 bit) but once the process (loading and searching) starts , address input must be of the same bit until the process stops.
- ❏ In one level cache, the number of the cache line can be one.But in set-associative mapping, if the cache line is one, then the number of sets should be one, as we cannot divide one cache line further.
- ❏ In the two-level cache, the number of cache lines should be greater than one(power of 2 must be >0) as if the number of a cache line is one, then the number of a cache line in level one should be zero, and that is not possible.

INPUT FORMAT

CL: Number of the cache line(Integer)
B: block size (in words)
Request: Read/Write/ stop(to stop the process)
S: Set in case of set associative
Read: Address
Write: Address, Data(String)

FUNCTIONS

In every mapping, there are two functions one is for read another is for write, and it takes an address as an argument. First, it will check the request READ/WRITE according to it will call their respective function.

Read Function

When the input address is found in the cache memory, a Read hit occurs, and the address is read from a cache. But when a read results in a Read miss, it'll get that block of memory and put it in the cache or replace it with an existing block in that line and then read the address. Addressing is mapping dependent.

Write Function

when a program writes data to cache, it writes that data to the main memory as well. So according to the write policy, I am using a write-through and write-allocated combination in my program. According to our project, I am showing changes in cache memory only. If the address to which it wants to write the data is present in the cache, then it prints write hit.but if the address is not present, i.e., write miss, then write location is loaded to cache, followed by a write-hit operation. Addressing is mapping dependent.

Replacement Method

Only needed for associative and set-associative mapping
Using FIFO(first in first out), it replaces the cache line that has been in the cache the longest.

Direct Mapping

Function:

Both functions take an address as the argument
direct_Map_read()- For read
direct_Map_write()- For write

First, the address space is split into three parts TAG | LINE NUM |BLOCK OFFSET(num of bits required to represent a block size). According to direct mapping, each particular block is to be stored in a specific cache line. So, with the help of line number, it will go to that line and compare tags if it matches, catch hit otherwise cache miss. If a cache line is already taken up by any memory block and a new block to be loaded in the cache, then the old block is replaced with the new block.

OUTPUT FOR DIRECT MAPPING

```
num of cache lines:4
block size:4
Request: Read
```

```
CPU generates address:001011000110
Read miss
Address NOT FOUND  in the cache line  0
Current Tag is replaced by an empty block in the cache line   0
Tag:  001011  Block Offset:  0110
['001011', 'null', 'null', 'null']
Request: Write
CPU generates address:001011000110
data :234a
Write hit
Address FOUND, writing data to cache  in the cache line  0
Tag:  001011  Block Offset:  0110
['001011', 'null', 'null', 'null']
Request: stop
```

Associative Mapping

Functions

associative_Map_read() - Read
associative_Map_write() - Write

In this type, any block can go into any line of the cache. This means that address is split into two parts BLOCK OFFSET that is used to identify which word in the block is needed, and the remaining bits become TAG. So if it finds the address in the cache memory, it's a cache hit otherwise cache miss. When the cache is filled, replacing is followed.

OUTPUT FOR ASSOCIATIVE MAPPING

```
num of cache lines:4
block size:4
Request: Read
CPU generates address:0100011010101
Read miss
Address NOT FOUND  in the cache line  1
Current Tag is replaced by an empty block in the cache line   1
Tag:  0100011  Block Offset:  0101
['null', '0100011', 'null', 'null']
Request: Write
CPU generates address:1110010101010
data :23d4
Write miss
Address  NOT FOUND  in the cache line  2
Current Tag is replaced by an empty block in the cache line 2
```

```
Tag:  1110010  Block Offset:  1010
['null', '0100011', '1110010', 'null']
Request: Write
CPU generates address:0100011010101
data:24
Write hit
Address FOUND, writing data to cache  in the cache line  1
Tag:  0100011  Block Offset:  0101
['null', '0100011', '1110010', 'null']
Request: stop
```

Set - Associative Mapping

Function

setAssociative_Map_read - Read
setAssociative_Map_write - Write

In this, a group of few lines together, creating a set. The number of lines in a set can be found by the number of lines divided by the Num of the set. So the address is split into three parts TAG | SET NUM | BLOCK OFFSET. By finding the Set num, it can go to that specific set in the cache memory and compare the Tags within the cache lines in that set, if it finds a catch hit, else cache miss. When any set in the cache gets filled then it replaces the cache line that has been in the cache the longest.
The replacement method is the same as the associative mapping.

OUTPUT FOR SET-ASSOCIATIVE MAPPING

```
num of cache lines:4
block size:4
Set: 2
Request: Read
CPU generates address:01000110100100
Read miss
address NOT FOUND
Current Tag is replaced by an empty block
Tag:  010001101  Block Offset:  0100  Set:  0
[['010001101', 'null'], ['null', 'null']]
Request: Write
CPU generates address:010001101100010
data :23a
Write miss
address NOT FOUND
Current Tag is replaced by an empty block
Tag:  0100011011  Block Offset:  0010  Set:  0
```

```
[['010001101', '0100011011'], ['null', 'null']]
Request: stop
```

Two Level Cache

- ❑ **In the two-level cache, when it does not find the address in the cache memory(level 1 and level 2), it loads into BOTH level 1 and level2.**
  First, it searches in level 1 cache memory for the Read and Writes, if it finds cache hit, there is no need to search in level 2, but if it fails to find the address cache miss occurs, and its search in level 2 if a cache miss occurs again it loads in both the level 1 and level 2 cache.

- ❑ **If it finds an address in level two, then it also loads the same address in level 1.**
  So if the processor wants to search that address again , there is no need to search in level 2, and there are more chances of a cache hit in level one.

- ❑ **Anything that is removed from level 1 goes into level 2.**
  It means if level one cache is full, there will be a replacement, in that case, the removed block memory will go in level 2 cache. If the address comes again for search, it will be present in level 2 cache and increase the cache hit's probability.

INPUT FORMAT
same as level one cache. Num of cache lines for level 1 will be half of level 2, i.e., CL/2(as the size of level 1 is half of level 2).

FUNCTION
Same two functions, one for read and another for write.

Read Function
The above three points would be followed to read the address in cache memory.

Write Function
When an address found in level one, i.e., write hit (writes data to cache) at the same time, address load in level 2 followed by write hit operation (write is done synchronously both to the level 1 and level 2) vice-verse when address found in level 2. If address not found in any cache, then address load in both the cache followed by write hit operation.

Direct Mapping

same as level one, the only difference is first it searches in level 1, then in level 2(if it's not present in level1). Loading and searching, I explained above.

OUTPUT FORMAT

```
num of cache lines:4
block size:4
Request: Read
CPU generates address:010010011010101
Read miss
Address NOT FOUND in the cache level 1
Current Tag is replaced by an empty block in cache line   1
Tag:  010010011  Block Offset:  0101
Read miss
Address NOT FOUND in the cache level 2
Current Tag is replaced by an empty block in the cache line   1
Tag:  010010011  Block Offset:  0101
cache1 ['null', '010010011']
cache2 ['null', '010010011', 'null', 'null']
Request: Write
CPU generates address:010010011010101
data:23
Write hit
address FOUND, writing data to cache  in the cache line  1   and cache level one
Tag:  010010011  Block Offset:  0101
cache1 ['null', '010010011']
cache2 ['null', '010010011', 'null', 'null']
Request: stop
```

Associative Mapping

Same as level 1, loading searching explained, the removed block memory will go in level 2 cache.

OUTPUT FORMAT

```
num of cache lines:4
block size:4
Request: Write
CPU generates address:001010001011111
data :23d
Write miss
address NOT FIND in cache level 1
```

```
Current Tag is replaced by empty block in cache line   0
Tag:  00101000101  Block Offset:  1111
Write miss
address NOT FIND in cache level 2
Current Tag is replaced by empty block in cache line   0
Tag:  00101000101  Block Offset:  1111
cache1 ['00101000101', 'null']
cache2 ['00101000101', 'null', 'null', 'null']
Request: stop
```

Set-Associative Mapping

Num of a set will be the same, but Num of cache lines in that set will be divided according to the number of lines in the cache. Loading and searching are the same as in one level cache, as explained above. Removed block memory will go in the same set, where it is removed, i.e., if B1 block is removed from set 0 in level1, then it goes in the same set 0 in level 2.

OUTPUT FORMAT

```
num of cache lines :4
block size :4
Set: 2
Request :Read
CPU generates address :00101101011010
Read miss
address NOT FIND in cache level 1
Current Tag is replaced by empty block
Tag:  001011010  Block Offset:  1010  Set:  1
Read miss
address NOT FIND in cache level 2
Current Tag is replaced by empty block
Tag:  001011010  Block Offset:  1010  Set:  1
cache1 [['null'], ['001011010']]
cache2 [['null', 'null'], ['001011010', 'null']]
Request :Write
CPU generates address :00101101011010
data :23
Write hit
address FOUND in cache level 1
Tag:  001011010  Block Offset:  1010  Set:  1
cache1 [['null'], ['001011010']]
cache2 [['null', 'null'], ['001011010', 'null']]
Request :stop
```