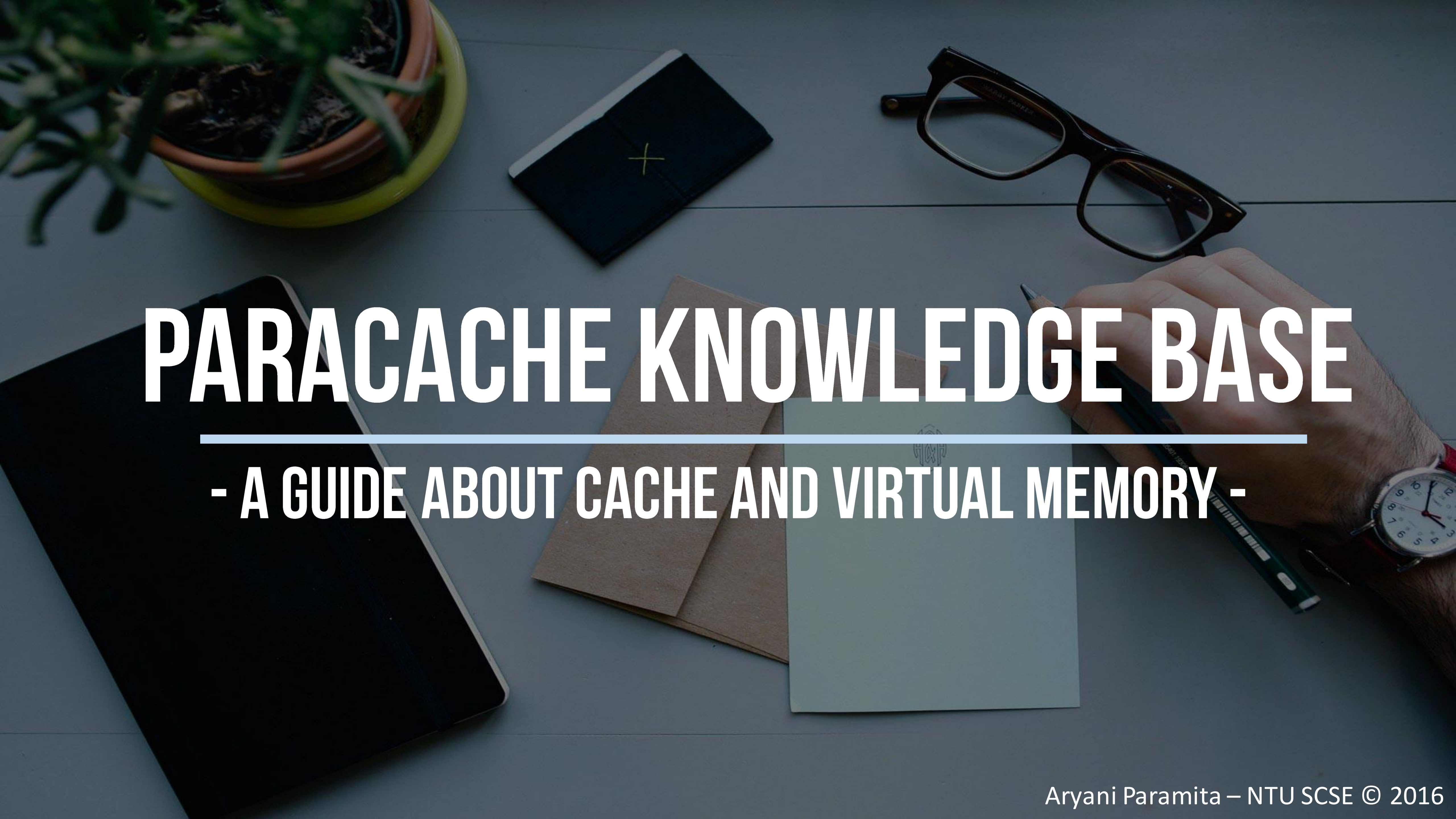
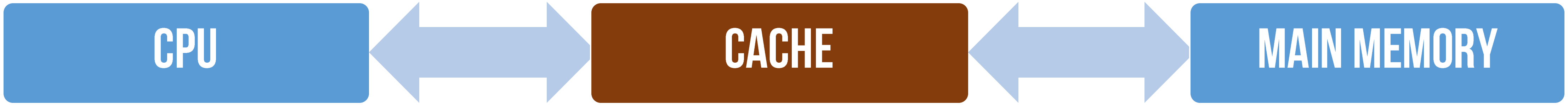


PARACACHE KNOWLEDGE BASE

- A GUIDE ABOUT CACHE AND VIRTUAL MEMORY -

Aryani Paramita – NTU SCSE © 2016



CPU

CACHE

MAIN MEMORY

CACHE

- To speed up accesses by storing recently used data closer to CPU instead of main memory
- Accessed by content → **Content Addressable Memory**

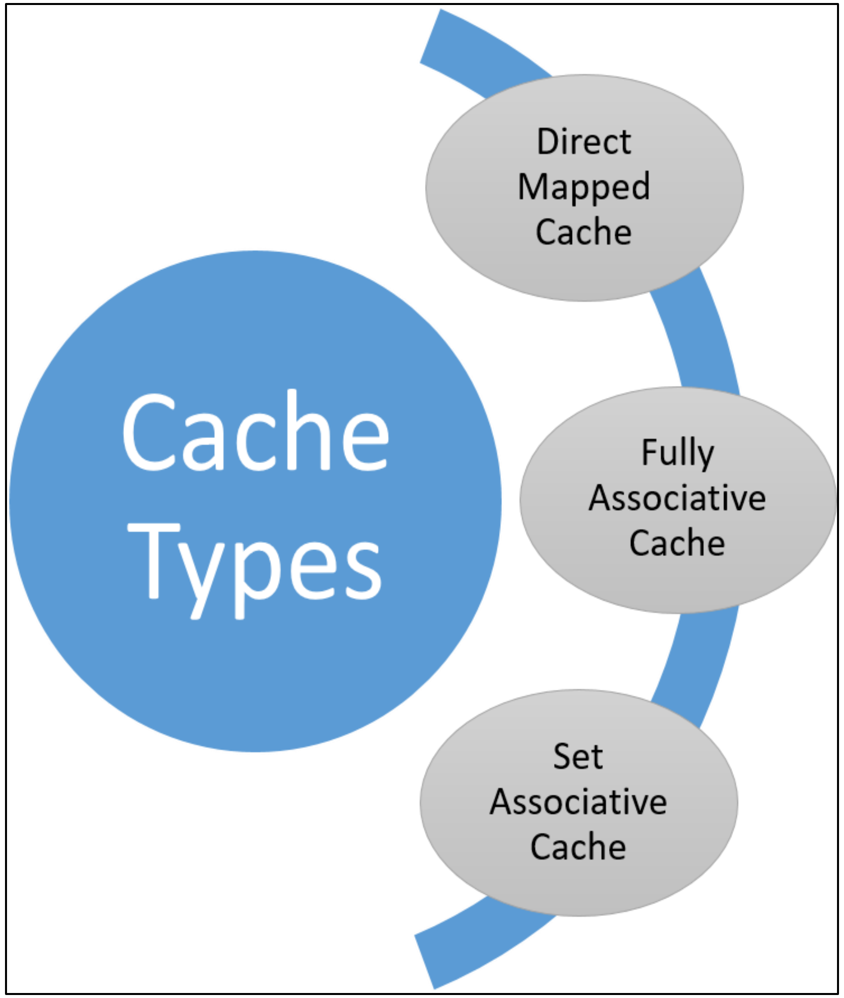
HIT / MISS

- **Cache Hit** - When requested data is found in cache
- **Cache Miss** – When requested data is not found in the cache

REPLACEMENT POLICY

How do we choose victim cache line?

- **FIFO** (First In First Out)
- **LRU** (Least Recently Used)
- **Random**



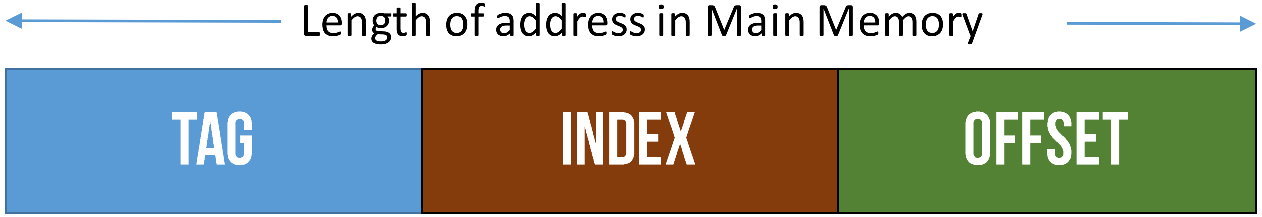
Cache Types

Direct Mapped Cache

Fully Associative Cache

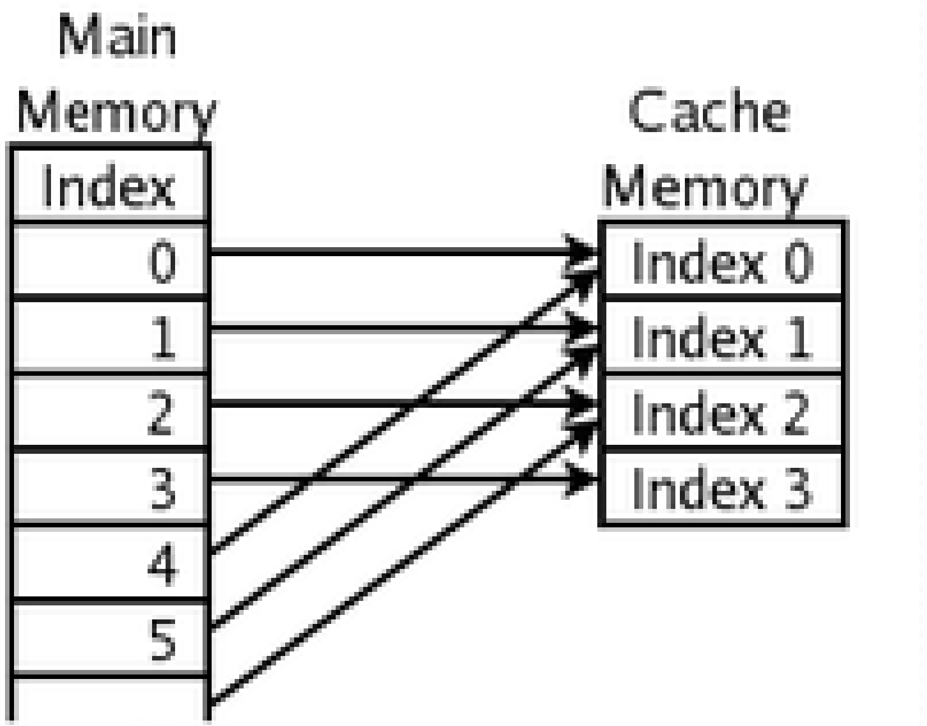
Set Associative Cache

DIRECT MAPPED CACHE



Length of address in Main Memory

TAG | INDEX | OFFSET



DEFINITIONS

- **TAG** – distinguished one cache memory block with another
- **INDEX** – identifies the cache block
- **OFFSET** – points to desired data in cache block

INSTRUCTION BREAKDOWN

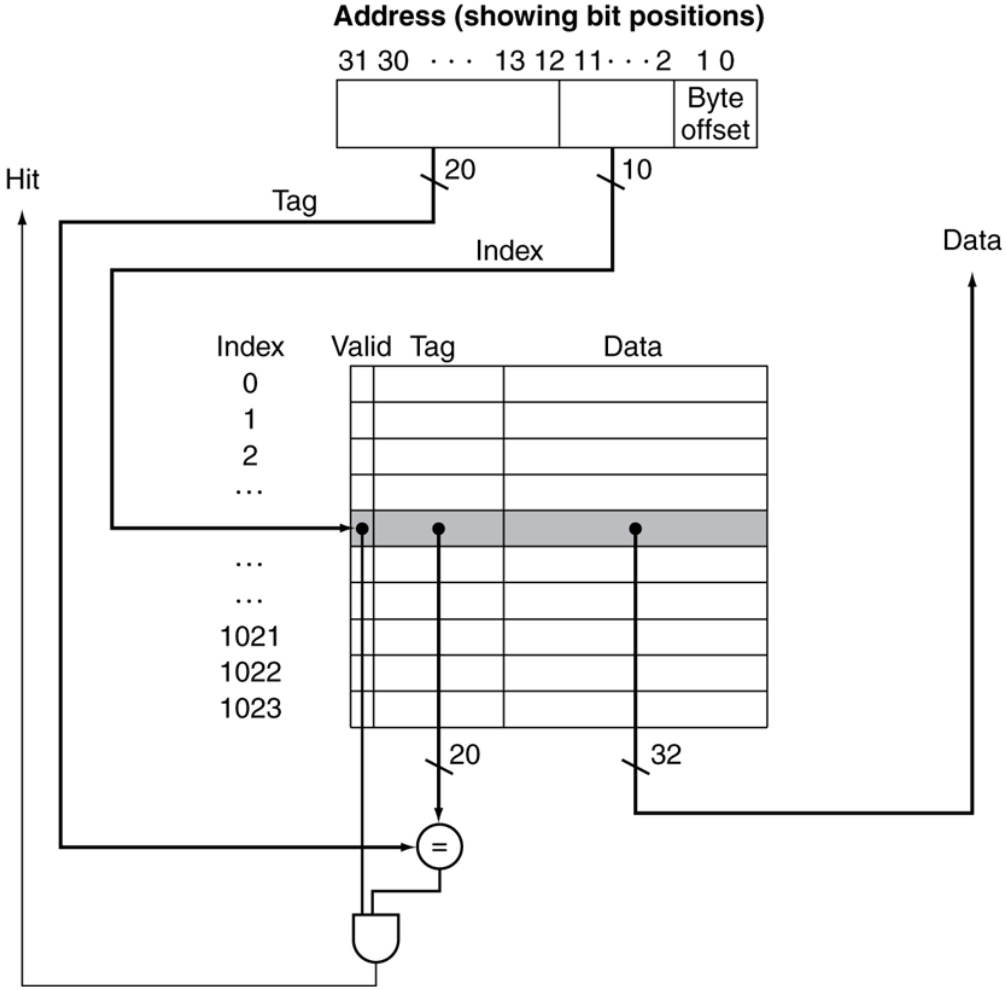
Each of the address of load instruction is broken into three parts: tag, index and offset. Main memory size = M , cache size = C and offset bits = Z results in the following (right figure).

Length of load instruction	: $\log_2(M)$	**bits**
OFFSET	: z	**bits**
INDEX	: $\log_2(C) – z$	**bits**
TAG	: $\log_2(M) – \log_2(C)$	**bits**
CACHE BLOCKS	: $\log_2(C)$	**blocks**

DIRECT MAPPED CACHE

HOW IT WORKS

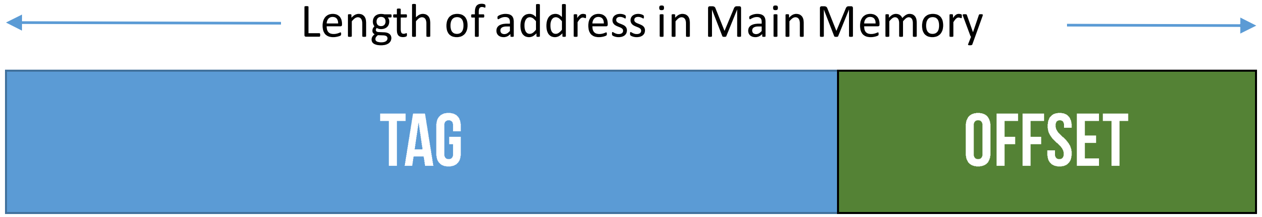
- The requested address is broken down into **tag, index,** and **offset.**
- Cache table with corresponding **index** will be examined.
  - If valid bit of the index is equals to 0, **cache miss** is obtained
  - Else tag bit of the requested address will be compared with tag bit in cache table
    - If tag is matched, **cache hit** is obtained
    - Else **cache miss** is obtained
- When **cache hit** is obtained, data from cache table will be returned. Else, data will be retrieved from main memory.



PRO / CONS

Direct mapping is simple and inexpensive to implement, but if a program accesses 2 blocks that map to the same line repeatedly, the cache begins to thrash back and forth reloading the line over and over again leads to high miss rate.

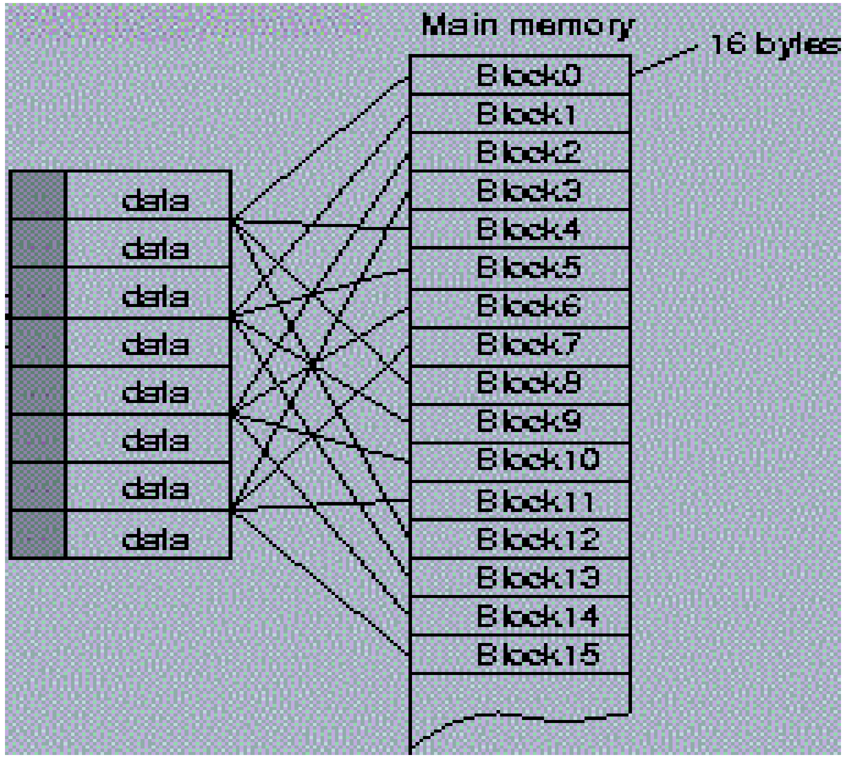
FULLY ASSOCIATIVE CACHE



Length of address in Main Memory

TAG

OFFSET



DEFINITIONS

- **TAG** – distinguished one cache memory block with another
- **OFFSET** – points to desired data in cache block

INSTRUCTION BREAKDOWN

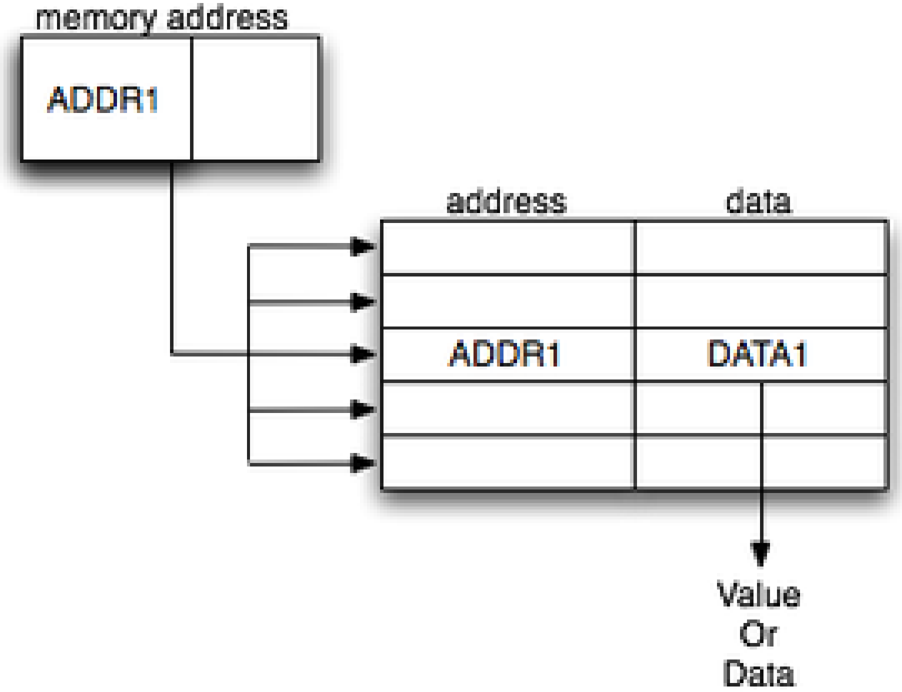
Each of the address of load instruction is broken into three parts: tag and offset.
Main memory size = M , cache size = C and offset bits = Z results in the following (right figure).

Length of load instruction	: $\log_2(M)$	**bits**
OFFSET	: z	**bits**
TAG	: $\log_2(M) - \log_2(C)$	**bits**
CACHE BLOCKS	: $\log_2(C)$	**blocks**

FULLY ASSOCIATIVE CACHE

HOW IT WORKS

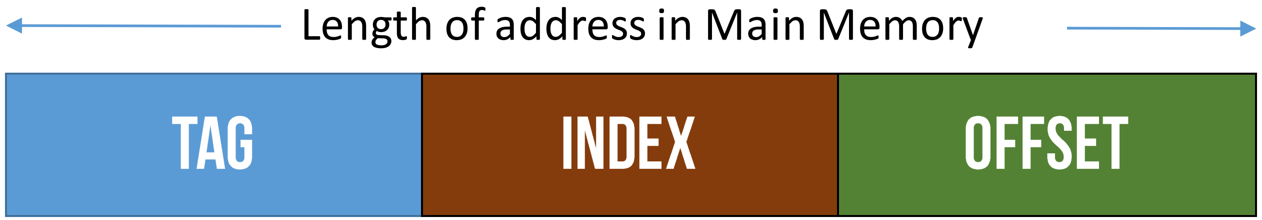
- The requested address is broken down into **tag** and **offset.**
- Requested **tag** will be searched through cache with valid bit
  - If there is **tag** matched with one of the index in the cache table, **cache hit** is obtained
  - Else, **cache miss** is obtained
- When **cache hit** is obtained, data from cache table will be returned. Else, data will be retrieved from main memory.



PRO / CONS

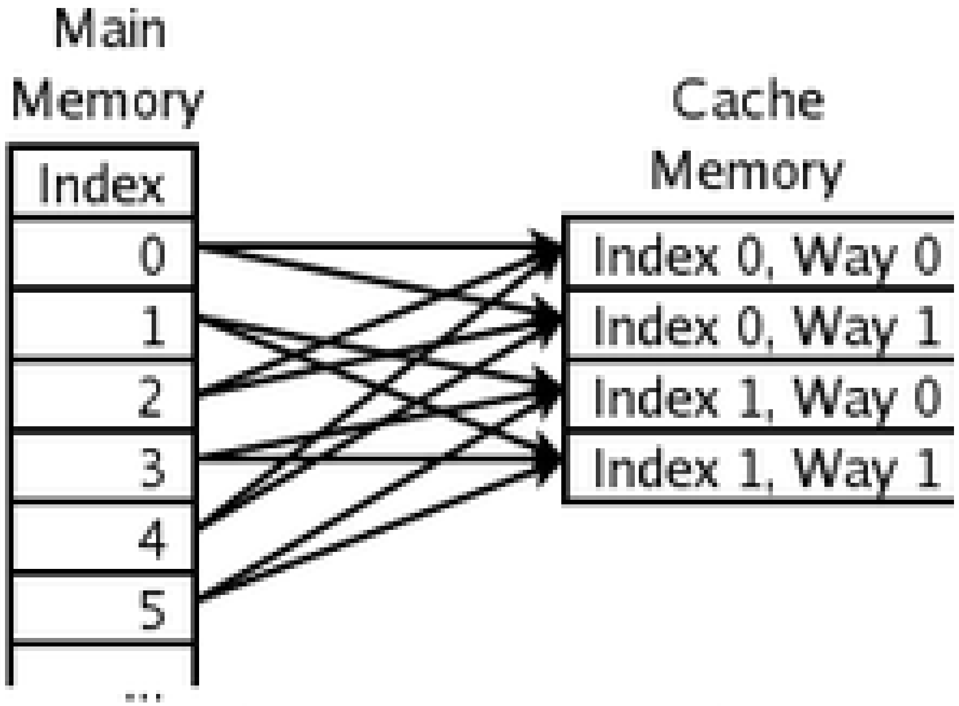
Fully Associative is the most efficient utilisation of cache blocks, yet it is expensive to transverse through the cache to find each requested tag. Since there is no specified slot for each instruction, an algorithm of replacement policy must be designed along with implementation of fully associative cache.

[N-WAY] SET ASSOCIATIVE CACHE



N-way set associative cache combines the idea of direct mapped cache and fully associative cache. It has direct mapped principle to an index, yet fully associative concept within the index. **An index** contains **N** blocks of way.

2-Way Associative Cache Fill



INSTRUCTION BREAKDOWN

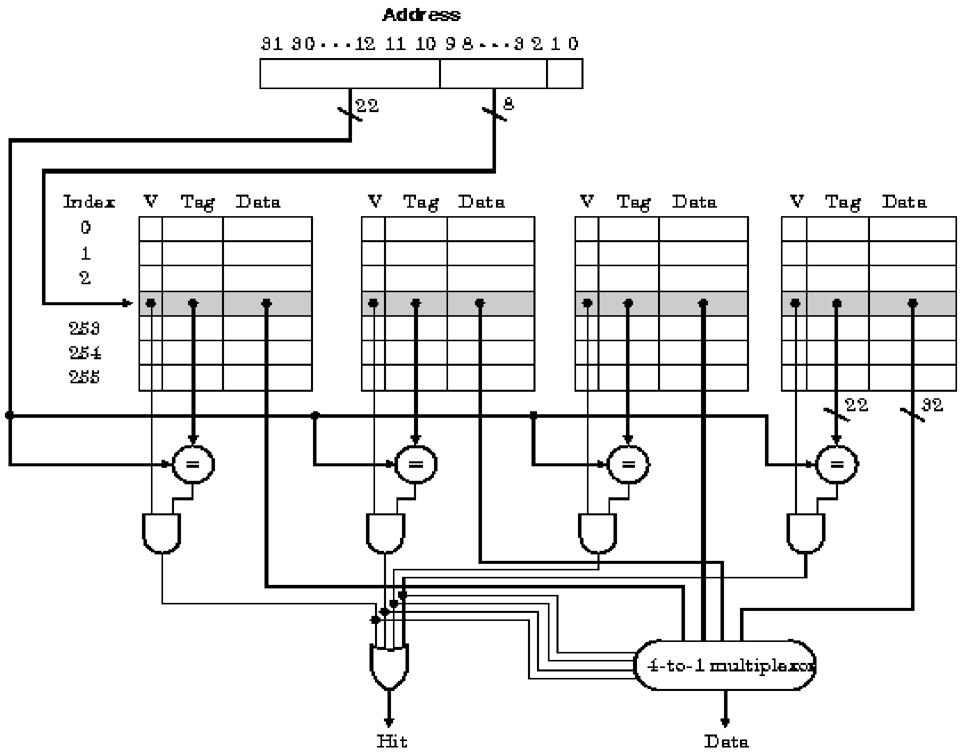
Each of the address of load instruction is broken into three parts: tag, index and offset. **N-way** set associative, Main memory size = M , cache size = C and offset bits = Z results in the following (right figure).

Length of load instruction	: $\log_2(M)$	**bits**
OFFSET	: z	**bits**
INDEX	: $\log_2(C) - z - \log_2(N)$	**bits**
TAG	: $\log_2(M) - \log_2(C) + \log_2(N)$	**bits**
CACHE BLOCKS	: $\log_2(C)$	**blocks**

[N-WAY] SET ASSOCIATIVE CACHE

HOW IT WORKS

- The requested address is broken down into **tag, index** and **offset.**
- Cache table with corresponding **index** will be examined. **N-ways** of cache blocks will be transversed.
  - If one of the **way** has the requested index, a **cache hit** will be obtained
  - Else **cache miss** is obtained
- When **cache hit** is obtained, data from cache table will be returned. Else, data will be retrieved from main memory.



PRO / CONS

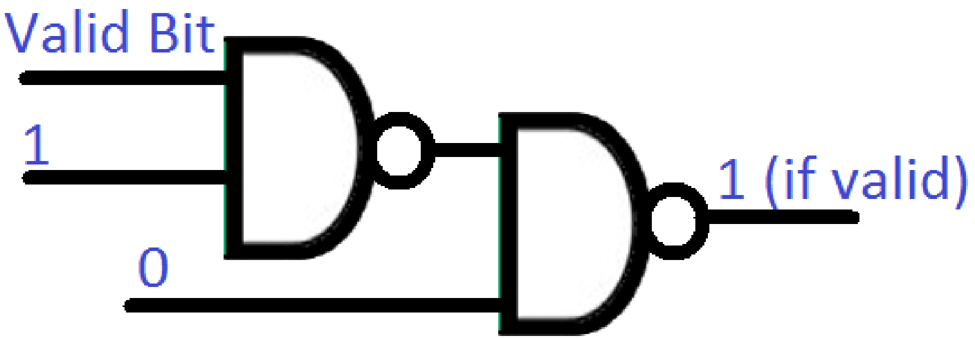
Combining direct mapped and fully associative principle is seen as the most balanced way to obtain high hit rate meanwhile maintaining the resources cost. However, N-Way associative could be hard for initial implementation as various concepts is involved.

RESOURCES CALCULATION (CACHE ANALYSIS)

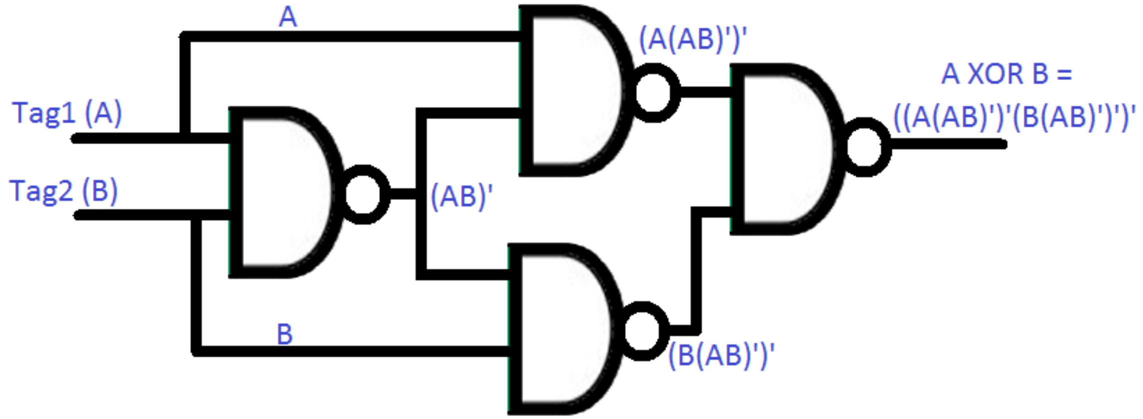
- Different cache types has varied resource utilisation. This section will explain NAND gates used in each type of cache.
- Decision made by each cache type when an instruction is loaded:
  - Direct Mapped : 1 * valid_bit + valid_bit_1* tag_comparison.
  - Fully Associative : cache_size * valid_bit + valid_bit_1 * tag_comparison
  - N-way set associative : N* valid_bit + valid_bit_1 *tag_comparison
- Valid_bit_1 is a variable to calculate the number of (validbit==1) in respective situation, meanwhile valid_bit and tag_comparison are the NAND needed for each action.

RESOURCES CALCULATION (CACHE ANALYSIS)

Valid_Bit
(2 NAND GATES)



Tag_Comparison_per_bit
(4 NAND GATES)



RESOURCES CALCULATION (CACHE ANALYSIS)

- Tag_comparison = cache_bit * tag_comparison_per_bit.
- Hence, the costs of each cache can be summarised as:

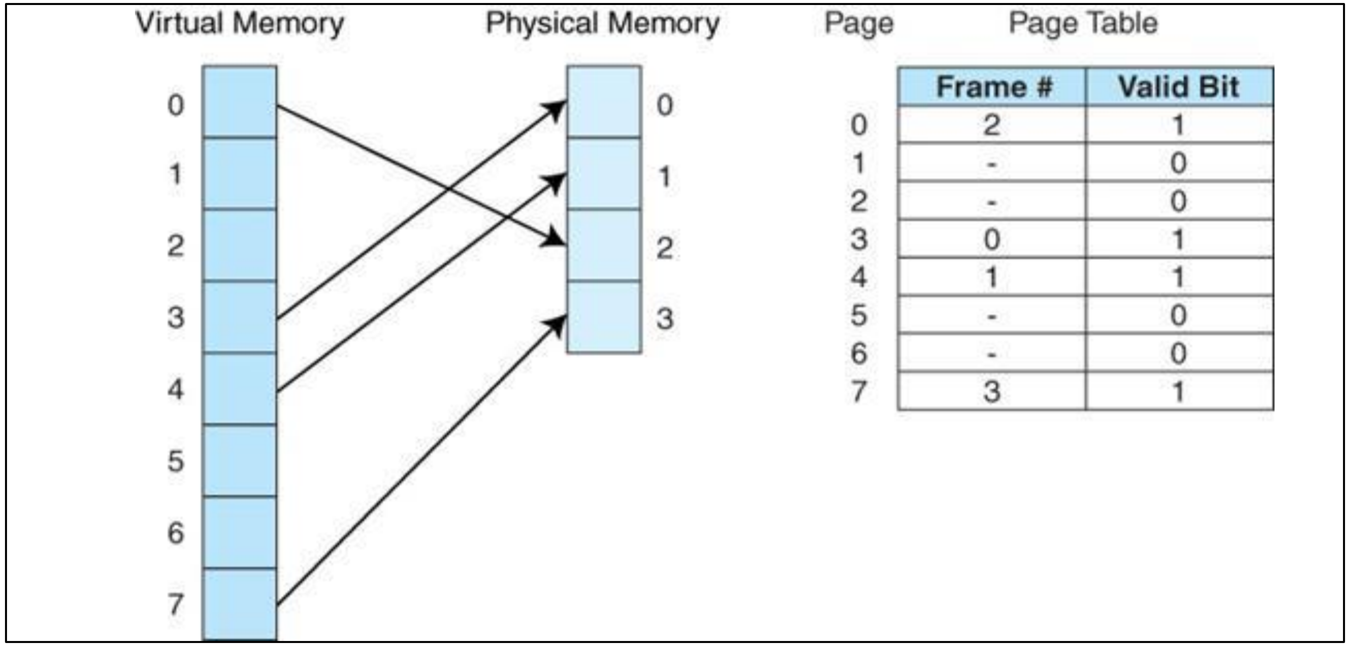
Direct Mapped	: 1 * 2 + valid_bit_1* 4 * cache_bit .
Fully Associative	: cache_size * 2 + valid_bit_1 * 4 * cache_bit
N-way set associative	: N* 2 + valid_bit_1 *4 * cache_bit

VIRTUAL MEMORY

- Extends the memory capacity of the main memory by using a portion of the disk drive.
- Allows the system to run programs that are bigger than the size of main memory
- Paging is implemented in this system.

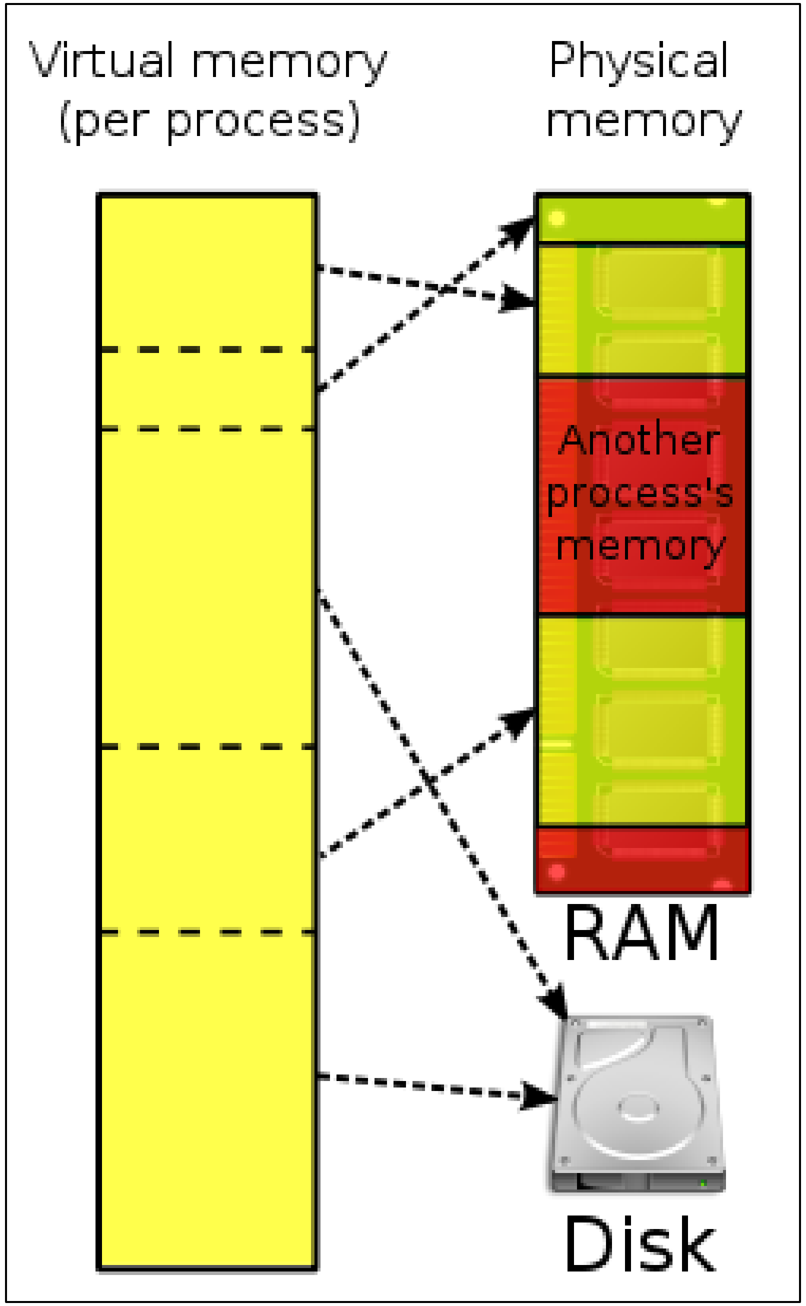
PAGE TABLE

- Data structure that maintain information concerning the location of each page, whether on disk or in memory.

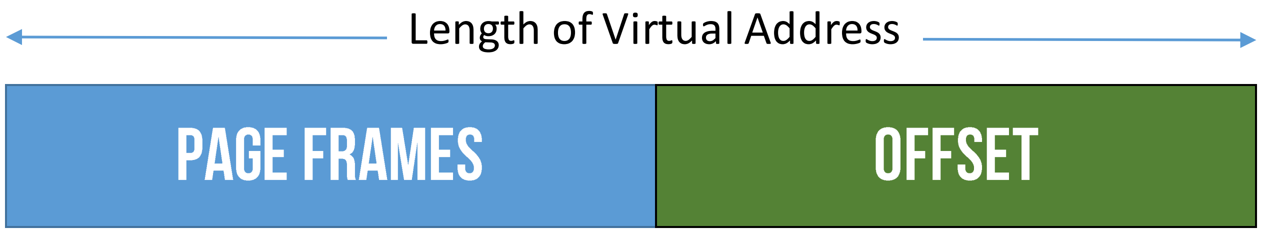


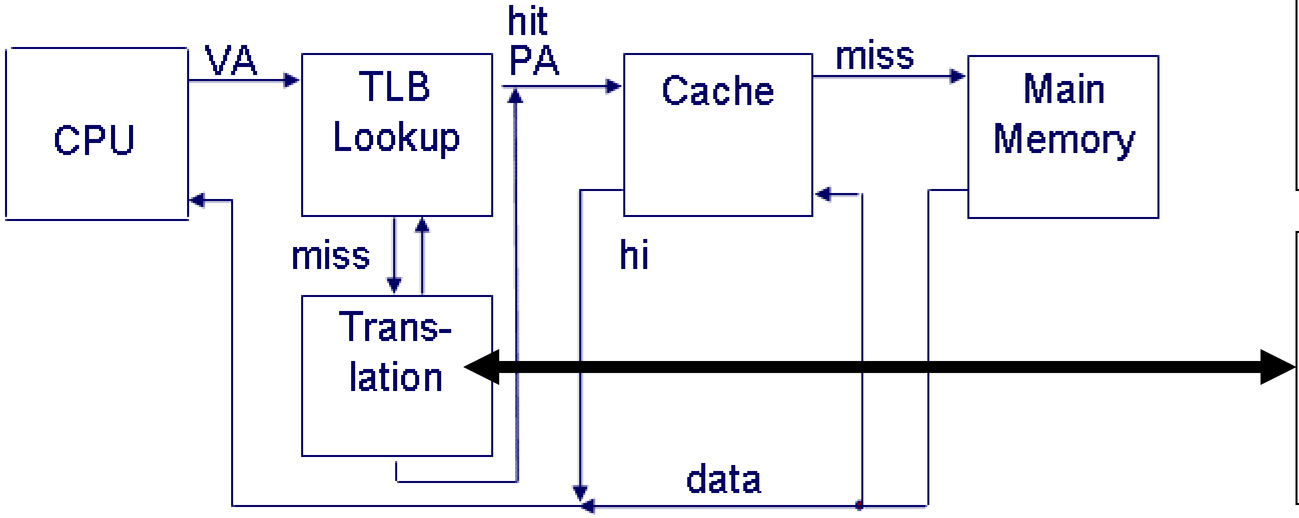
Page Table

Page	Frame #	Valid Bit
0	2	1
1	-	0
2	-	0
3	0	1
4	1	1
5	-	0
6	-	0
7	3	1



VIRTUAL MEMORY





DEFINITIONS

- **Page Frames** – virtual memory partitions of main memory.
- **OFFSET** – points to desired data in page frame
- **Physical address** – actual memory address of physical memory
- **Virtual address** – addresses where physical address mapped into

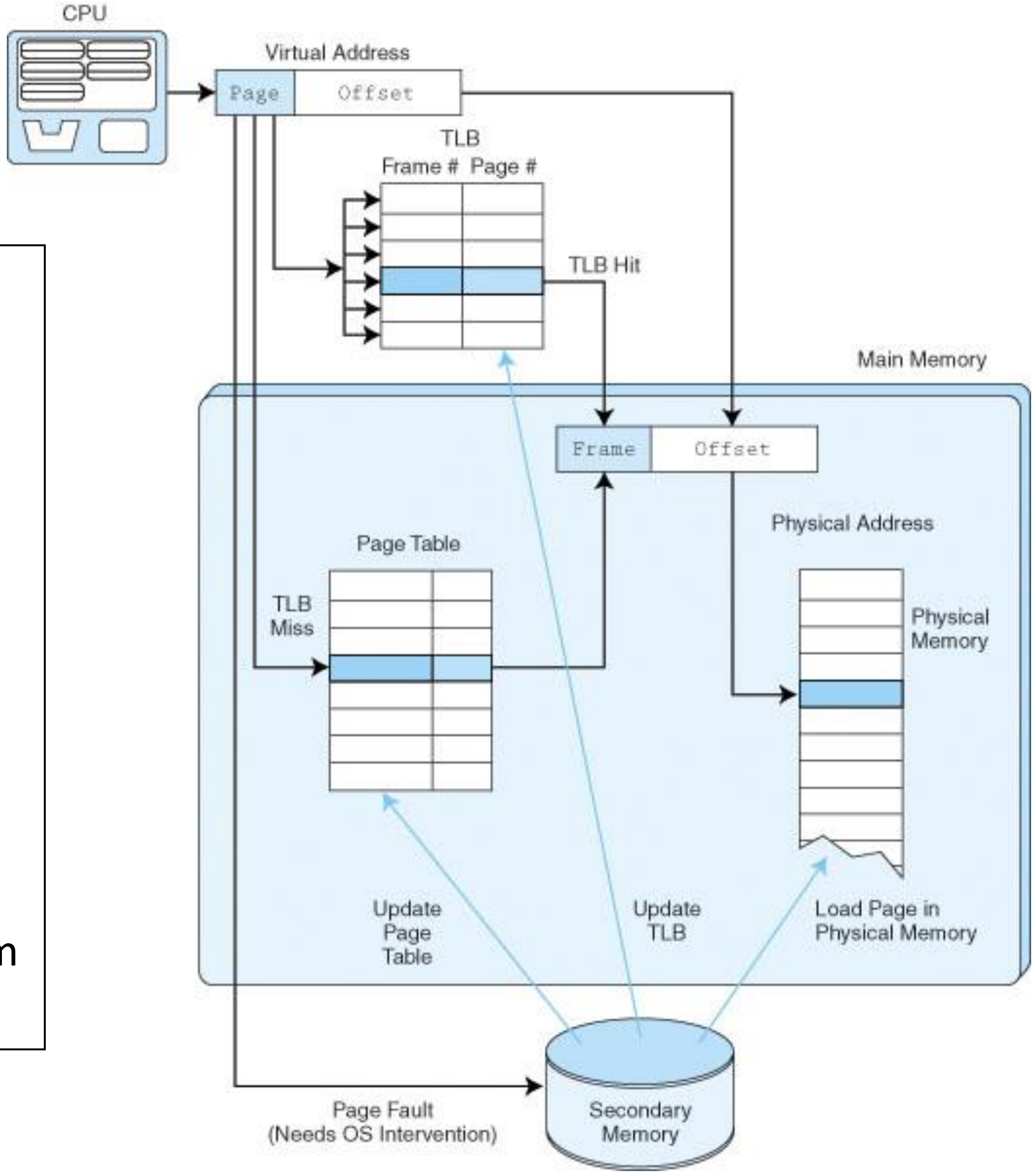
TLB (Translation Lookaside Buffer)

A memory cache that stores recent translations of virtual memory to physical addresses for faster retrieval

VIRTUAL MEMORY

HOW IT WORKS

- The requested address is broken down into **page and offset.**
- Requested page is searched in TLB with fully associative principle.
  - If there is a frame number that matched the page, **TLB HIT** is obtained, and data will be loaded from TLB.
  - Else, page will be searched through page table with direct mapping principle.
    - If matched page id is found in page table, **Page Table Hit** is obtained and data will be loaded from Page Table. TLB will be updated accordingly.
    - Else, **Miss** is obtained, and data will be loaded from secondary memory.



THANK YOU