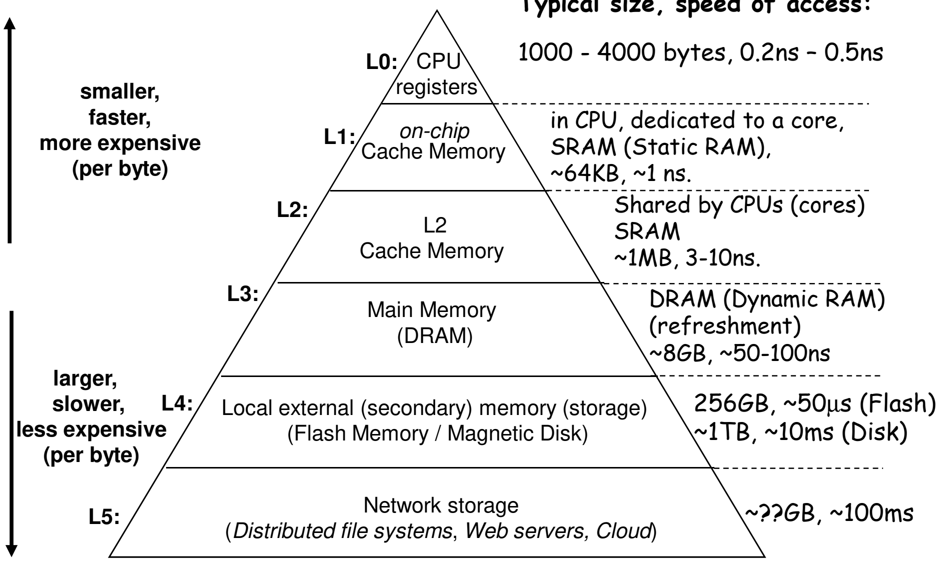
## 6 Memory Organization (Internal / External)

### 6.1 The memory hierarchy:



smaller,
faster,
more expensive
(per byte)

larger,
slower,
less expensive
(per byte)

Typical size, speed of access:

L0: CPU registers — 1000 - 4000 bytes, 0.2ns – 0.5ns

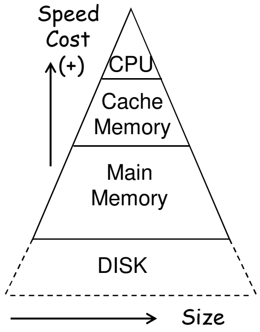
L1: *on-chip* Cache Memory — in CPU, dedicated to a core, SRAM (Static RAM), ~64KB, ~1 ns.

L2: L2 Cache Memory — Shared by CPUs (cores) SRAM ~1MB, 3-10ns.

L3: Main Memory (DRAM) — DRAM (Dynamic RAM) (refreshment) ~8GB, ~50-100ns

L4: Local external (secondary) memory (storage) (Flash Memory / Magnetic Disk) — 256GB, ~50µs (Flash) ~1TB, ~10ms (Disk)

L5: Network storage (*Distributed file systems*, *Web servers*, *Cloud*) — ~??GB, ~100ms



### 6.1 The memory hierarchy (cont'd):

There are memories of different speeds, costs and sizes.

Faster memory is more expensive.

To increase the average speed and (at the same time) to decrease the cost of the memory system, different types of memories are organized in the form of multilevel memory hierarchy.

A faster memory (SRAM), namely the cache memory, is inserted between the main memory and the CPU.

The CPU can directly access the main memory and cache memory (internal).

However, data on the disk must first be fetched into main (or cache) memory).

**The goal:** To provide a memory system with a high speed that is close to the fastest level of memory and cost per byte low as the cheapest level.

This solution takes the advantage of the **locality** of computer programs.

## 6.2 The principle of locality (Locality of reference):

Programs tend to reuse instructions and data they have used recently.

Observation: A typical program spends 90% of its execution time in only 10% of the code.

We can predict what instructions and data a program will use in the near future based on its access in the recent past.

There are two types of locality:

**Temporal Locality** (Locality in time): Recently accessed addresses are likely to be accessed in the near future.

**Spatial Locality** (Locality in space) : After an access to a memory address the next access will be likely to a nearby address.

Reasons for locality:

Structure of the program: The execution of instructions follows a sequential order. Program segments, such as routines and macros, tend to be stored in the same neighborhood of the memory space.

In addition, related data is stored in nearby locations.

Loops

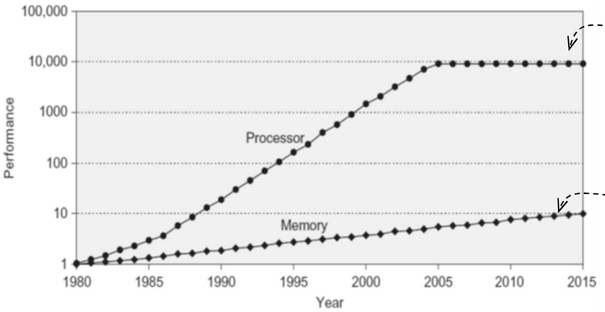
Arrays

## 6.3 The performance gap between the CPU and the main memory:

Main memories are implemented based on DRAM (dynamic RAM) technology.

The speed of processors are much higher than the speeds of main memories.

Therefore main memories cannot meet bandwidth requirements of the CPUs.



Memory requests per second, on average
**Single processor**

**Source:** Hennesy&Patterson, *Computer Architecture, 6/e*

Access per second, 1/latency

Multicore processors further increase the bandwidth requirements.

The Intel i7 6700 with 4 cores and 4.2 GHz clock rate can achieve a total peak bandwidth of about 537.6 GB/s (see slide 1.33).

The peak bandwidth for DRAM main memory, using two memory channels, is only 6.3% of this (2.133 GHz * 2 * 64 /8 = 34.1 GB/s).).

## 6.4 Memory Technologies

**RAM (Random-Access Memory):**

Actually, referring to only this type of memory as "random access" is a misuse of the term because all of the semiconductor (electronic) memories used in computer systems are random access (not sequential).

Distinguishing characteristic of RAM:

- CPU can access the memory to read data and to write new data directly (easily and rapidly without an additional device).

  These operations are accomplished through the use of electrical signals.

  It would be more precise to call this type of memory as "*Read-Write Memory*".
- RAM is *volatile*. If the power is interrupted, then the data are lost.

  Thus, RAM can be used only as temporary storage.

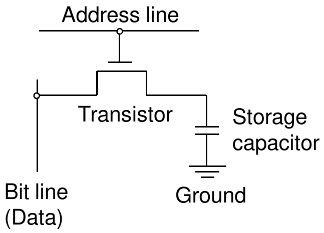
Memory latency measures:

- **Access time:** Time between write/read request and when desired word has been stored or made available for use
- **Cycle time:** Minimum time between two unrelated requests to memory

There are two types of RAM; DRAM (Dynamic RAM) and SRAM (Static RAM).

**DRAM (Dynamic RAM):**

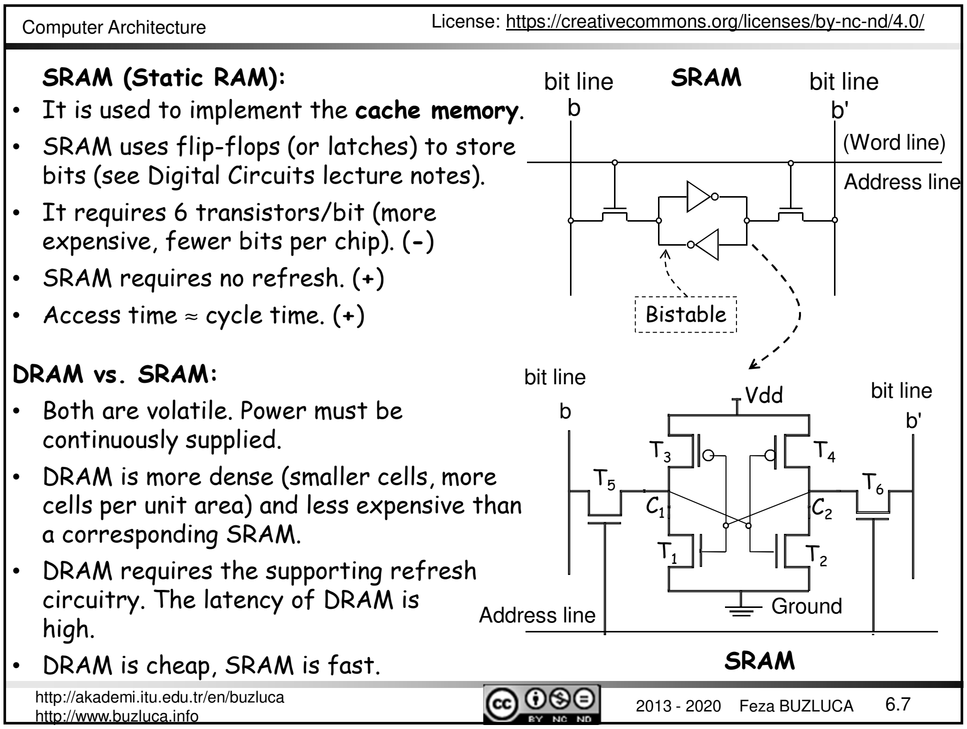
- It is used to implement the **main memory**.
- A dynamic RAM (DRAM) is made with cells that store data as charge on capacitors.
- Because capacitors have a natural tendency to discharge, dynamic RAMs require periodic charge **refreshing** to maintain data storage (~ every 8ms).
- The stored charge leaks away, even with power continuously applied (The term *dynamic* refers to this tendency ).



- During the refresh process, the memory is unavailable (latency). (-)
- Must be re-written after being read. Reading destroys the information (latency). (-)
- Difference between access time and cycle time. Cycle time > access time (-)
- Cheap and dense: one transistor/bit. More bits can be placed on one chip. (+)

Some improvements:

- SDRAM (Synchronous DRAM): Added clock to DRAM interface. Burst mode.
- DDRAM (Double data rate DRAM): Data is transferred on both the rising edge and falling edge.

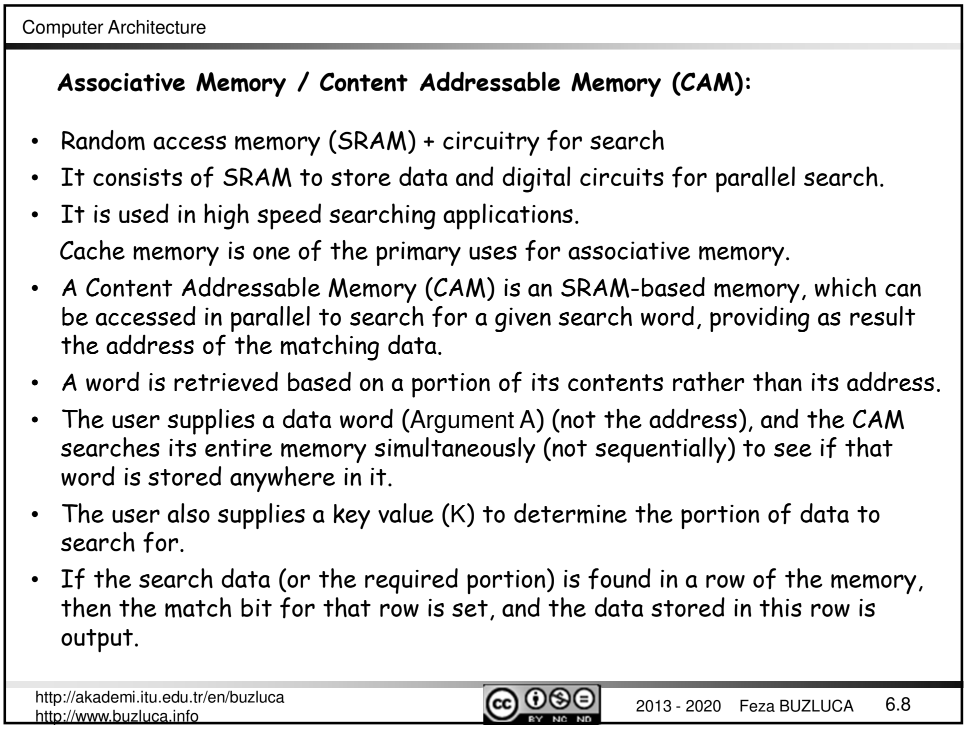


**SRAM (Static RAM):**

- It is used to implement the **cache memory**.
- SRAM uses flip-flops (or latches) to store bits (see Digital Circuits lecture notes).
- It requires 6 transistors/bit (more expensive, fewer bits per chip). (-)
- SRAM requires no refresh. (+)
- Access time ≈ cycle time. (+)

**DRAM vs. SRAM:**

- Both are volatile. Power must be continuously supplied.
- DRAM is more dense (smaller cells, more cells per unit area) and less expensive than a corresponding SRAM.
- DRAM requires the supporting refresh circuitry. The latency of DRAM is high.
- DRAM is cheap, SRAM is fast.



**Associative Memory / Content Addressable Memory (CAM):**

- Random access memory (SRAM) + circuitry for search
- It consists of SRAM to store data and digital circuits for parallel search.
- It is used in high speed searching applications.
  Cache memory is one of the primary uses for associative memory.
- A Content Addressable Memory (CAM) is an SRAM-based memory, which can be accessed in parallel to search for a given search word, providing as result the address of the matching data.
- A word is retrieved based on a portion of its contents rather than its address.
- The user supplies a data word (Argument A) (not the address), and the CAM searches its entire memory simultaneously (not sequentially) to see if that word is stored anywhere in it.
- The user also supplies a key value (K) to determine the portion of data to search for.
- If the search data (or the required portion) is found in a row of the memory, then the match bit for that row is set, and the data stored in this row is output.

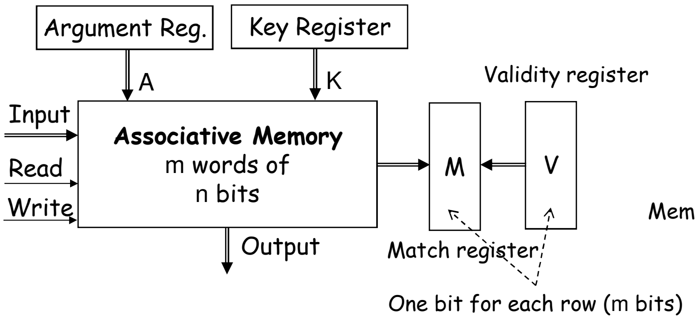
**Associative Memory / Content Addressable Memory (CAM)** (cont'd):

On power-up, the memory contains random values.

An additional flag (valid bit V) is necessary to indicate whether or not a memory line has been loaded with valid data.

On power-up, the hardware sets valid bits of all lines to "invalid" (V=0).

When data is written to line i, the corresponding valid bit is set ($V_i$=1).



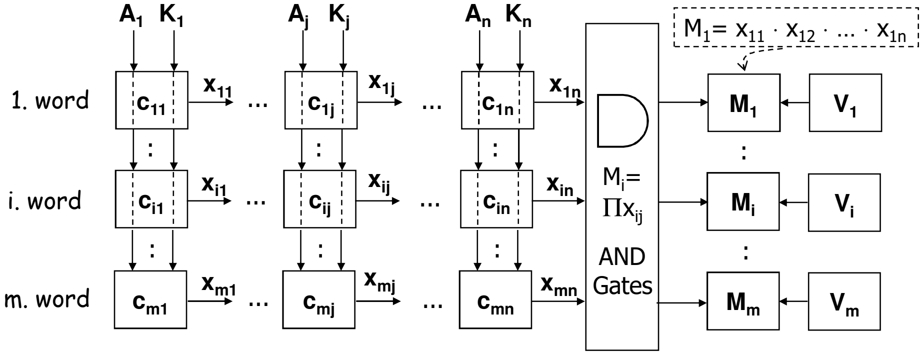
**Example:**

A: 1011 1111
K: 1111 0000

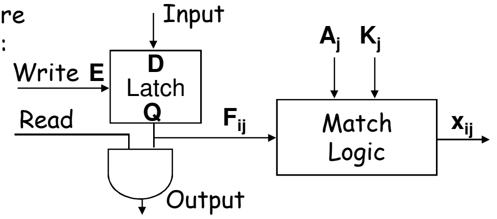
Memory: 1100 1111 M=0
1011 1010 M=1

Out: 1011 1010

**Internal Structure:** $A_i$ and $K_i$ values are broadcasted to all cells in parallel.



The structure of a cell ($c_{ij}$):



$$X_{ij} = \overline{K_j} + \overline{A_j \oplus F_{ij}}$$

$$X_{ij} = \overline{K_j} + A_j \odot F_{ij}$$

$$X_{ij} = \overline{K_j} + A_j \cdot F_{ij} + \overline{A_j} \cdot \overline{F_{ij}}$$

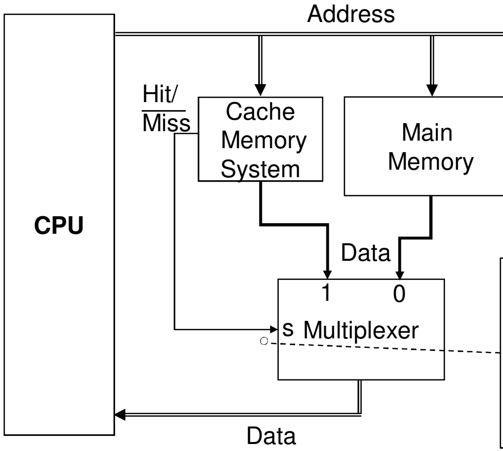
$$M_i = V_i \cdot \prod_{j=1}^{n} x_{ij}$$

## 6.5 Cache Memory , The internal memory system

### 6.5.1 Cache Memory Principles

The goal is to store copies of frequently accessed data in the fast cache memory to reduce the average access time.

**Hit**: The requested word is found in cache memory.
**Miss**: A **miss** occurs if the requested data is not found in cache memory.



**Example:**
**For read operations only:**
Cache memory access time: 20 ns.
Main memory access time: 100 ns.
Hit ratio: H=0.9 (90% hit)
Average memory access time:
$t_a$= 0.9*20 + 0.1*100 = 28 ns.

In systems with asynchronous bus operations like the MC68000, the DTACK signal is sent to the CPU at different times based on there being a hit or not. Thus, when data is retrieved from cache memory, the bus cycle is completed faster.

### 6.5.1 Cache Memory Principles (cont'd)

**Block transfer:**

Upon a cache miss (the requested element is not found in cache), ***a block*** of elements that contains the requested element is brought (copied) from main memory to cache memory.

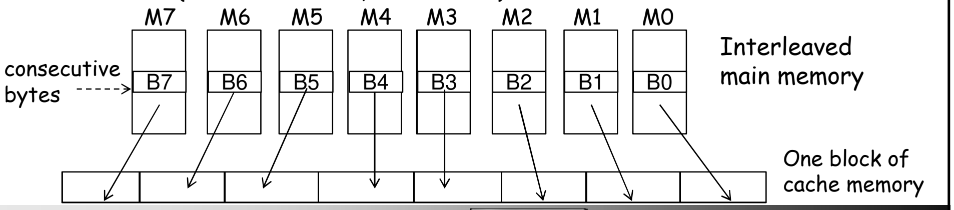
**Reason for block** (instead of a single word) **transfer: spatial locality**.

The next element to be requested will most likely be located near the currently requested element.

The disadvantage of transferring a block instead of a single element is that it takes longer.

To reduce the block transfer time between main and cache memories, the *memory interleaving technique* is used.

Data are stored in different memory modules; that is, consecutive memory addresses are located in successive memory modules, so they can be accessed at the same time (similar to RAID, section 7.2).



### 6.5.1 Cache Memory Principles (cont'd)

**Replacement Techniques:**

The capacity of cache memory is lower than that of main memory.
At any moment, only a part of the data in main memory can be kept in cache memory.

When cache memory is full, a *replacement algorithm* is needed to determine which block in cache will be replaced by the new block from main memory.

There are different replacement techniques; the most common ones are:

- **FIFO** (*First In, First Out*): The block that has been in the cache the longest is replaced.
- **LRU** (*Least Recently Used*): The block that has been used the least while residing in the cache is replaced.

  The access history of each block is taken into consideration.

  An *aging counter* is assigned to each block to keep track of references to that block in cache.

A **hardware unit,** called the *Cache Memory Controller* or *Cache Memory Management Unit,* performs cache operations.

### 6.5.2 Cache Memory Mapping Techniques

- Is a content of main memory currently present in cache memory?
- If present, where in cache memory is the data located?

**1. Full Associative Mapping**

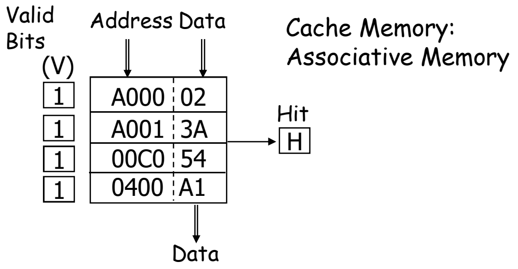
a) Without blocks:

In practice, all mapping techniques operate on data blocks.
For the sake of simplicity, we will first look at a technique that does not use blocks.

**Method:** The most frequently referenced addresses and their data are kept in an associative memory.

The address generated by the CPU is searched for in the cache (content addressable memory).

- If there is a hit, data is read from cache.
- If a miss occurs, data is read from main memory and also copied into cache.

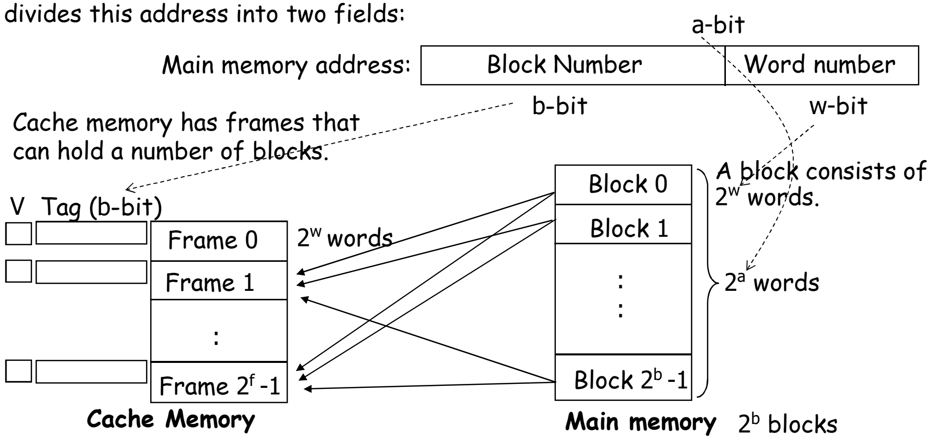


| (V) | Address | Data |
|---|---|---|
| 1 | A000 | 02 |
| 1 | A001 | 3A |
| 1 | 00C0 | 54 |
| 1 | 0400 | A1 |

Without blocks, the technique benefits from only temporal locality but not spatial locality.
Therefore, in practice, data blocks are moved between main and cache memories.

**b) Full Associative Mapping** (with blocks)

The CPU generates a main memory address, and the cache controller unit (CCU) divides this address into two fields:



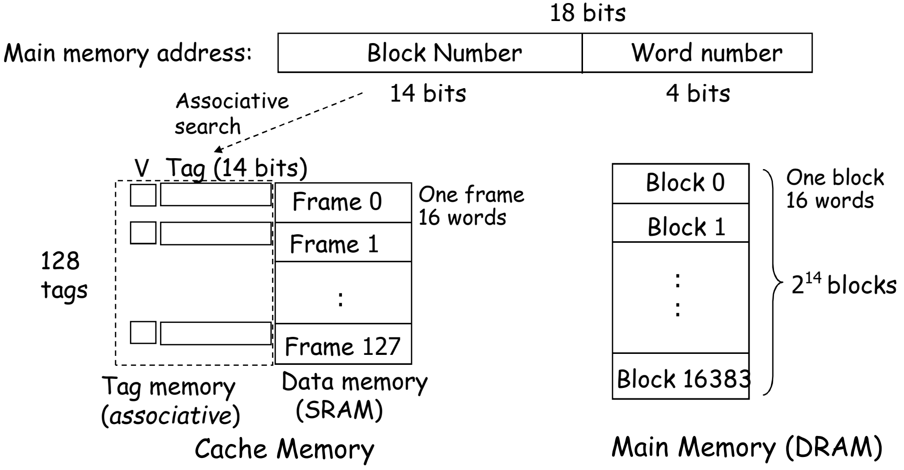
**A block of main memory can be placed into any frame of cache memory.**

To determine which block is currently residing in a frame, the tag is used.

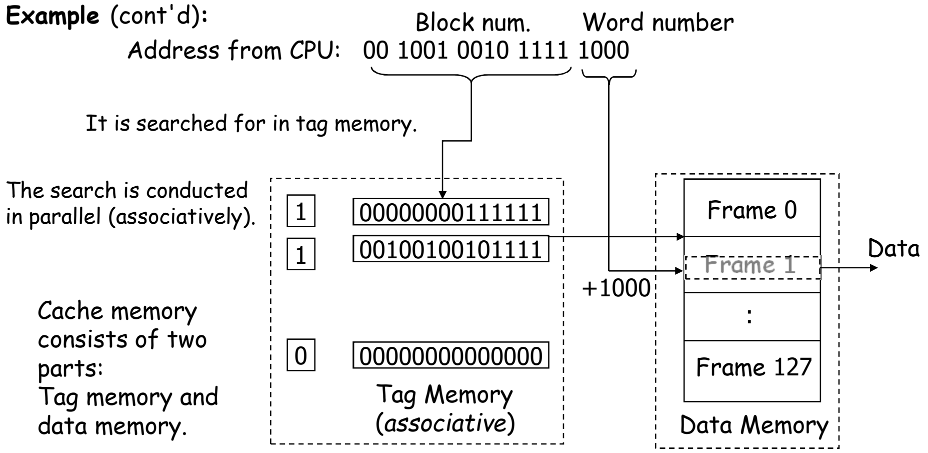
In this technique, the tag is the block number of the main memory address.

**Example:** *Full Associative Mapping*

| | |
|---|---|
| Main Memory: 256K x words | Address: a = 18 bits |
| Block size: 16 words | w = 4 bits Main memory contains $2^{14}$ blocks. b = 14 |
| Cache Memory: 2K x words data can be stored. | Cache memory contains $2^7$ =128 frames. f = 7 |



**Example** (cont'd):



The search is conducted in parallel (associatively).

Cache memory consists of two parts:
Tag memory and data memory.

If the replacement technique is LRU (*Least Recently Used*), then tag memory also contains *aging counters*. Each frame has its own counter.

When a frame is not referenced, its counter is incremented.

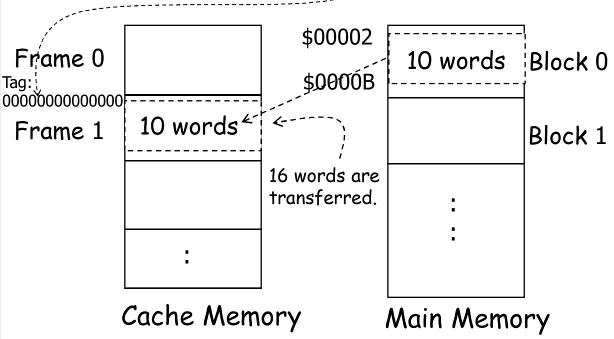
When a replacement is necessary, the frame that has the highest counter value ("oldest") is replaced.

**Example**: Array in a cache memory system

Main Memory: 256K x words  Cache Memory: 2K x words  Block size: 16 words

***Case A)*** A program in this system accesses an array with the starting address $00002 and a size of 10 words. The array is not in cache memory.
Assume: When the CPU starts accessing the array, the least recently used frame is F #1.

Starting address of the array: 00 0000 0000 0000 0010 ($00002)
End address of the array: 00 0000 0000 0000 1011 ($0000B)



When the CPU accesses the first word of the array ($00002), a miss occurs.

Although the array has 10 words, the cache management system transfers Block 0 in its entirety (16 words) to Frame 1.

When the CPU accesses the next 9 elements of the array, hits will occur.

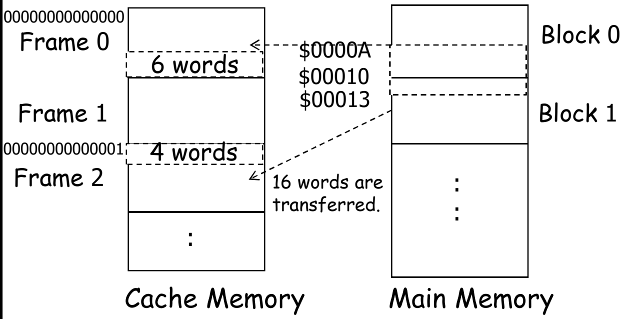
In total: 1 miss, 9 hits.

**Example:** Array in a cache memory system (cont'd)

Main Memory: 256K x words  Cache Memory: 2K x words  Block size: 16 words

**Case B)** A program in this system accesses an array with the starting address $0000A and a size of 10 words. The array is not in cache memory.
Assume that, the least recently used frames are Frames #0 and #2.

Starting address of the array: 00 0000 0000 0000 1010 ($0000A)
End address of the array: 00 0000 0000 0001 0011 ($00013)



Each block is processed separately.

Different blocks have different tag values.

In this example, two block replacements are necessary.

Although the size of the array is smaller than the block (frame) size, it occupies two frames in cache memory.

**Example:** Array in a cache memory system (cont'd)

- When the CPU accesses the first word of the array ($0000A), a miss occurs.
- The cache management system transfers Block 0 in its entirety (16 words) to Frame 0.
- Next 5 accesses to the array generate hits.
- When the CPU accesses the 7th word of the array ($00010), a miss occurs because this word is in another block and its address has a new tag value.
- The cache management system transfers Block 1 with its entirety (16 words) to Frame 2 (because of LRU).
- Next 3 accesses to the array generate hits.
- In total: 2 misses, 8 hits

If the starting address of the array was selected properly (for example $00000), the array would occupy only one block and one frame in the cache memory as in case A.

Therefore, placement policies for arrays in main memory affect the performance of cache systems.

## 2. Direct Mapping

An incoming main memory block is always placed into a specific, fixed cache frame location.

It is not necessary to search for the location of a block in the cache because it is predetermined and fixed.

Therefore, associative memory is not necessary.

As the size of main memory is greater than that of the cache, several blocks of main memory map to the same cache frame.

It is necessary to determine which main memory block is currently residing in a frame.

The cache memory control unit divides the address from the CPU into three fields:

a bits

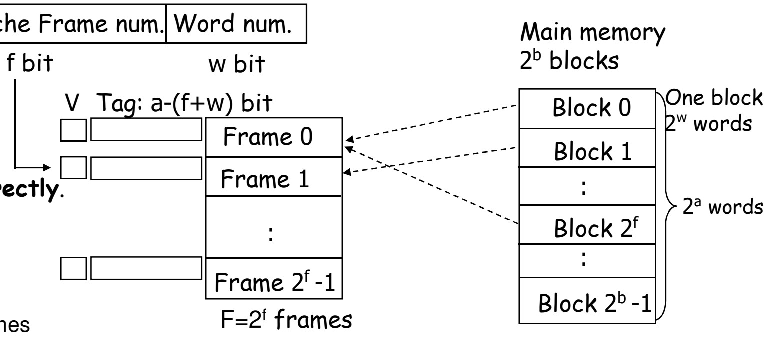
| Tag | Cache Frame number | Word number |
|---|---|---|
| a-(f+w) bits | f bits | w bits |

It indicates which of the blocks that can be placed in this frame is currently in cache.

This field determines the number of the cache frame that will hold this data.
The blocks that have this filed in common (same) try to reside in the same frame.
Only one of them can reside in the cache any given moment.

a bit

| Tag | Cache Frame num. | Word num. |
|---|---|---|
| a-(f+w) bit | f bit | w bit |



The frame is determined **directly**.

i = B mod F

i : Frame number
B: Block number
F: Number of frames
F=$2^f$

A main memory block can only reside in the cache frame with the number that is determined by the "Cache Frame number" filed of the address.

Even if there are unused (empty) frames in the cache, two blocks with the same "Cache Frame Number" field cannot be in the cache at the same time.

During replacement, a decision algorithm is not necessary.
Since the frame of the incoming block is fixed, this frame will be replaced.

**Associative memory is not necessary** because there is no need to search in the cache.

**Example** (*Direct Mapping*):

Main Memory: 256K x word
Block size: 16 words
Cache Memory: 2K x word data capacity

Address: a = 18 bits
w = 4 bits Main memory contains $2^{14}$ blocks. b = 14
Cache memory contains $2^7$ =128 frames. f = 7

Main memory address: (18 bits)

| Tag | Frame number | Word number |
|---|---|---|
| 7 bits | 7 bits | 4 bits |

In this system, the data in the following two addresses try to reside in the same cache frame.

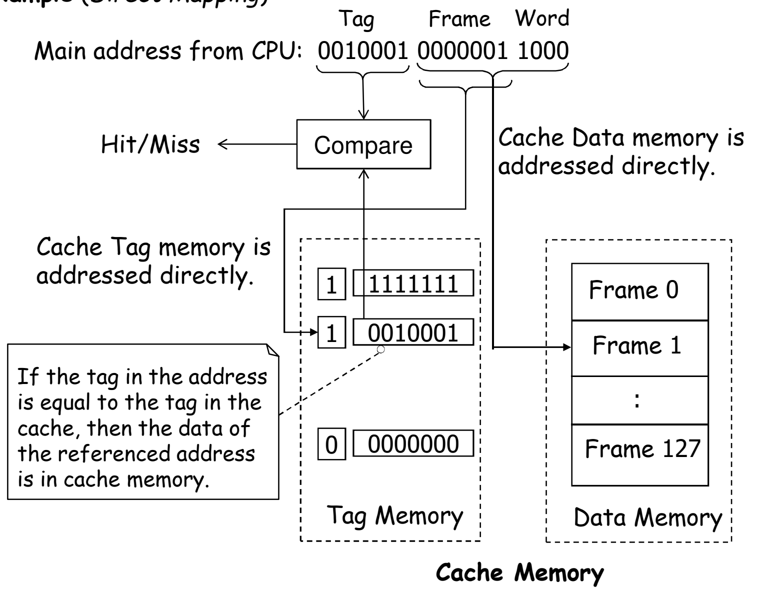
| Tag | Frame num. | Word num. |
|---|---|---|
| 0000000 | 0000000 | XXXX |
| 0000001 | 0000000 | XXXX |

The "Frame number" fields of both addresses are the same: 0000000. They will be placed in Frame 0.

At a specific point in time, only one of these data can be in cache memory.

To determine which data is currently in Frame 0 of cache memory, the tag value of the address is compared to the tag value stored in cache memory.

**Example** (*Direct Mapping*):

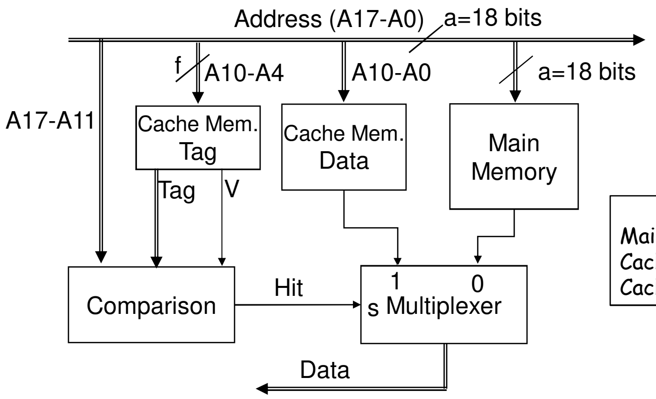

Tag
Frame
Word
Main address from CPU: 0010001 0000001 1000
Hit/Miss
Compare
Cache Data memory is addressed directly.
Cache Tag memory is addressed directly.
1
1111111
1
0010001
0
0000000
Tag Memory
Frame 0
Frame 1
:
Frame 127
Data Memory
Cache Memory

If the tag in the address is equal to the tag in the cache, then the data of the referenced address is in cache memory.

**Block diagram of the exemplary system:**

18 bits

| A17 A11 | A10 A4 | A3 A0 |
|---|---|---|
| Tag | Frame number | Word num. |
| 7 bits | 7 bits | 4 bits |



Main Memory: DRAM
Cache Data: SRAM
Cache Tag: SRAM

**3. Set Associative Mapping**

This method is a compromise between direct mapping and full associative mapping.

The cache is divided into a number of **sets**, and each set consists of a number of frames.

A given main memory block maps to a specific, fixed cache set (Snum), based on the equation Snum = B mod S, where S is the number of sets in the cache, B is the main memory block number, and Snum is the specific cache set to which block B maps.
However, an incoming block maps to any frame in the assigned cache set.

- Direct mapping is used to determine the set (fixed).
- Associative mapping is used to determine the frame in the set (flexible).

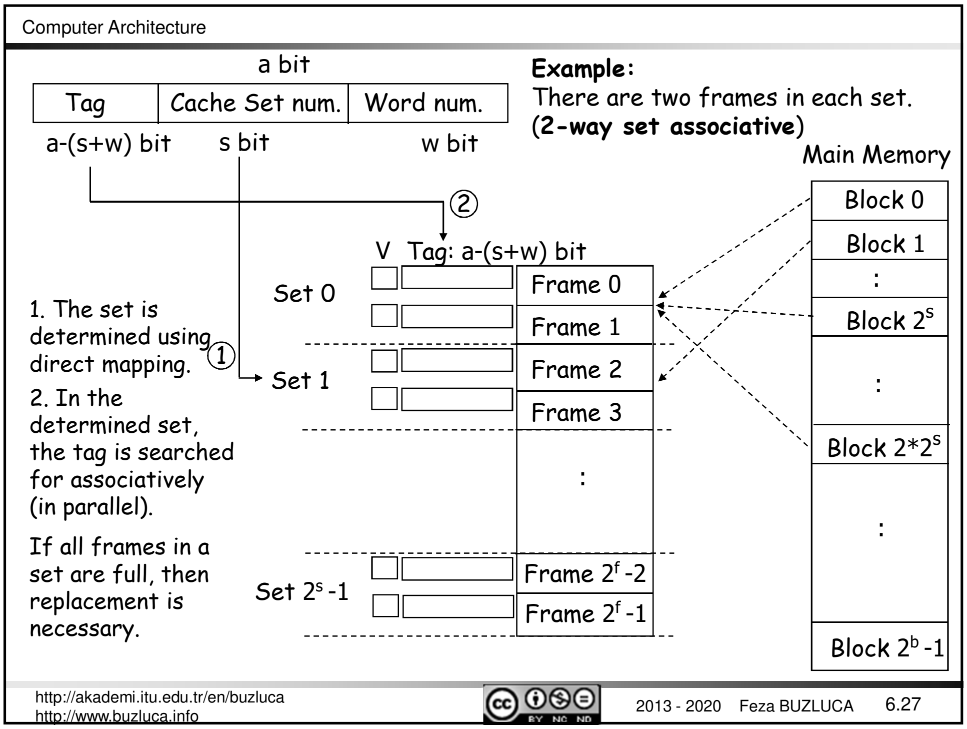
The cache controller divides the address issued by the CPU into three fields:

a bits

| Tag | Cache Set number | Word number |
|---|---|---|
| a-(s+w) bits | s bits | w bits |

Tag is used to search the block in within the determined set (associative).

This field is used to identify the specific cache set that should hold the targeted block.



Computer Architecture

a bit

| Tag | Cache Set num. | Word num. |
|---|---|---|
| a-(s+w) bit | s bit | w bit |

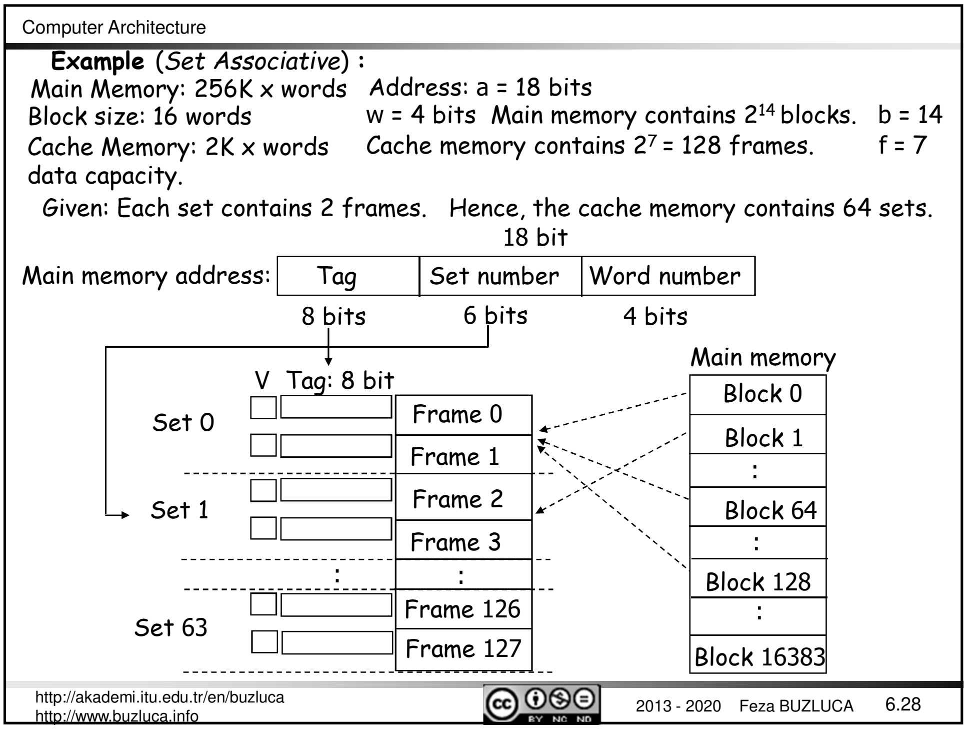
**Example:**
There are two frames in each set.
(**2-way set associative**)

1. The set is determined using direct mapping. ①

2. In the determined set, the tag is searched for associatively (in parallel).

If all frames in a set are full, then replacement is necessary.

http://akademi.itu.edu.tr/en/buzluca
http://www.buzluca.info
2013 - 2020 Feza BUZLUCA 6.27



Computer Architecture

**Example** (*Set Associative*) :

Main Memory: 256K x words Address: a = 18 bits

Block size: 16 words w = 4 bits Main memory contains $2^{14}$ blocks. b = 14

Cache Memory: 2K x words data capacity. Cache memory contains $2^7$ = 128 frames. f = 7

Given: Each set contains 2 frames. Hence, the cache memory contains 64 sets.

18 bit

Main memory address:

| Tag | Set number | Word number |
|---|---|---|
| 8 bits | 6 bits | 4 bits |

http://akademi.itu.edu.tr/en/buzluca
http://www.buzluca.info
2013 - 2020 Feza BUZLUCA 6.28

**Summary of mapping techniques:**

- **Associative mapping** is the most *flexible* and *efficient* technique because an incoming main memory block can reside in any frame of the cache.

  The disadvantage of this technique is the high *hardware cost* due to the associative memory.

- In **direct mapping,** the cache frame for each memory block is fixed.

  The main disadvantage is the *inefficient* use of the cache; a number of main memory blocks may compete for a cache frame even if there exist other empty frames.

  This disadvantage decreases the hit ratio.

  The main advantage of this technique is its simplicity; no search is needed.

  No replacement mechanism is needed because block locations are fixed.

- The cache utilization efficiency of the **set associative mapping** technique is expected to be *moderate* (i.e., not as high as the fully associative mapping technique and not as low as the direct mapping technique).

  The technique inherits the simplicity of the direct mapping technique in terms of determining the target set.

  By changing the number of frames in a set, we can bring it closer to one of the other two techniques.

### 6.5.3 Cache Memory – Main Memory Interactions

→ **Read (Hit):** Data is read from cache memory.

→ **Read (Miss):**

**a) Read-Through (RT):**

While the data (block) is being brought from main memory to cache, it is also read by the CPU simultaneously.

Cache memory and main memory are accessed in parallel.

**b) No Read-Through (NRT):**

Data are first brought from main memory to cache memory, and then the CPU reads data from the cache.

→ **Write (Hit):**

**a) Write-Through (WT):**

In each write operation, data is written to cache and also to main memory.

Disadvantage: It increases the access time.

Advantage: It provides coherence between the cache frames and their counterparts in main memory.

→ **Write (Hit)** (cont'd):

**b) Write-Back (WB):**

Writes are done only to the cache.

A block is written back to main memory only when a replacement is needed.

There are two types of write-back policies: *Simple write-back* and *flagged write-back*.

- **Simple Write-Back (SWB):**

  The replaced frame is always written back to main memory.

  It is not checked whether the frame was changed or not.

- **Flagged Write-Back (FWB):**

  Every cache frame is assigned a bit, called the *dirty bit*, to indicate that at least one write operation has been made to the block while residing in cache.

  At replacement time, the dirty bit is checked: if it is set, then the block is written back to main memory; otherwise, it is simply overwritten by the incoming block.

  The dirty bit is stored in the tag memory of the cache.

→ **Write (Miss):**

**a) Write Allocate (WA):**

The main memory block is updated and brought to cache.

**a) No Write Allocate (NWA):**

The missed main memory block is updated in main memory and not brought to cache.

If an attempt is made to read this block later, a miss will occur, and data will be brought to cache.

The write-through (WT) policy can be used together with write-allocate (WA) or no-write-allocate (NWA) methods. WTWA, WTNWA

In write-back (WB) policy, to maintain coherence between cache and main memory at the beginning, the write-allocate (WA) method is used (WBWA).

**Information held in Tag memory:**

In addition to Valid (V) and tag bits, depending on the method used, the following data must be also kept in tag memory:

- If LRU is used *aging counters*,
- If flagged Write-Back (FWB ) method is used "dirty" bit (D).

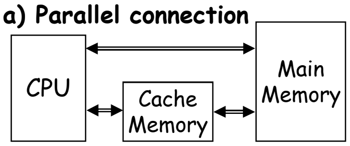
**A single line of a tag memory:**

| | | | |
|---|---|---|---|
| V | D | Counter | Tag |

## 6.5.4 CPU - Cache Memory – Main Memory Interconnection

**CPU - cache memory – main memory interconnection can be implemented in two ways:**

**a) Parallel connection**

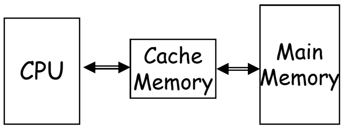


- **Read-Through (RT):** While a block is brought from main memory to cache it is also read by the CPU simultaneously (in parallel).
- **Load Through (LT):** A data is written to cache and to main memory simultaneously.

The parallel connection is suitable for the *Write-Through (WT )* method.

It can also be used with the *Write-Back (WB )* method.

**b) Serial connection**



- **No Read-Through (NRT):** The block is first brought from main memory to cache, and then the CPU reads the data from cache.
- **No Load-Through (NLT):** First, the data in cache is updated, then the frame is written to main memory.

This serial connection is suitable for the *Write-Back (WB)* method.

## 6.5.5 Access Time:

$t_a$ : *Average Memory Access Time*

w : Write ratio (number of write accesses / total number of all accesses)

h : Hit ratio

$t_{cache}$ : Cache memory access time

$t_{main}$ : Main memory access time

$t_{trans}$: Time to transfer a block between main memory and cache

$w_d$: The probability that a block in a cache is updated

| | **WT, RT/LT**<br>(Write-through , Parallel read/write) | | **WB,WA, NRT/NLT**<br>(Write-back, Serial read/write) | |
|---|---|---|---|---|
| Probability | **NWA** | **WA** | **SWB** | **FWB** |
| | *Access Time* | | *Access Time* | |
| **Read Hit**<br>(1-w)h | $t_{cache}$ | $t_{cache}$ | $t_{cache}$ | $t_{cache}$ |
| **Read Miss**<br>(1-w)(1-h) | $t_{trans}$ | $t_{trans}$ | $2t_{trans}+t_{cache}$ | $w_d\,(2t_{trans}+t_{cache})$ +<br>$(1-w_d)(t_{trans}+t_{cache})$ |
| **Write Hit**<br>wh | $t_{main}$ | $t_{main}$ | $t_{cache}$ | $t_{cache}$ |
| **Write Miss**<br>w(1-h) | $t_{main}$ | $t_{main}+t_{trans}$ | $2t_{trans}+t_{cache}$ | $w_d\,(2t_{trans}+t_{cache})$ +<br>$(1-w_d\,)(t_{trans}+t_{cache})$ |

## Access Time Calculation:

- **Write Through with Write Allocate, Read/Load Through (WTWA, RT/LT):**

Read Hit + Read Miss + Write Hit + Write Miss

$$t_a = (1\text{-}w)h\, t_{cache} + (1 - w)(1\text{-}h)t_{trans} + w \cdot h \cdot t_{main} + w(1\text{-}h)(t_{main} + t_{trans})$$

$$t_a = (1 - w)h\, t_{cache} + (1 - h)t_{trans} + w \cdot t_{main}$$

- **Write Through with No Write Allocate, Read/Load Through (WTNWA,RT/LT)**

$$t_a = (1 - w)h\, t_{cache} + (1\text{-}w)(1\text{-}h)t_{trans} + w \cdot h \cdot t_{main} + w(1\text{-}h)t_{main}$$

$$t_a = (1 - w)h\, t_{cache} + (1 - w)(1 - h)t_{trans} + w \cdot t_{main}$$

- **Simple Write Back with Write Allocate,No Read Through (SWBWA,NRT/NLT)**

Read Hit + Read Miss +Write Hit + Write Miss

$$t_a = (1 - w)h\, t_{cache} + (1 - w)(1\text{-}h)(2t_{trans} + t_{cache}) + w \cdot h \cdot t_{cache} + w(1 - h)(2t_{trans} + t_{cache})$$

$$t_a = t_{cache} + (1 - h) \cdot 2 \cdot t_{trans}$$

One $t_{trans}$ is needed to transfer a frame from cache to main memory and the second one is needed to bring the new block from main memory to cache.

- **Flagged Write Back,Write Allocate, No Read Through (FWBWA,NRT/NLT):**

$$t_a = t_{cache} + (1 - h)t_{trans} + w_d \cdot (1 - h)t_{trans}$$

$$t_a = t_{cache} + (1 - h)(1 + w_d)t_{trans}$$

## Exemplary processors with cache memories:

- **Intel386™:** Cache memory is outside of the CPU chip. SRAM memory.
- **Intel486™ (1989)**
  8-KByte on-chip (L1)
- **Intel® Pentium® (1993)**
  L1 on-chip: 8 KB instruction, 8 KB data cache (Harvard architecture)
- **Intel P6 Family: (1995-1999)**
  - **Intel Pentium Pro:**
    L1 on-chip: 8 KB instruction, 8 KB data cache (Harvard architecture)
    First L2 cache memory in the CPU chip.
    L2 on-chip: 256 KB. Different interconnections between L1, L2 and the CPU.
  - **Intel Pentium II:**
    L1 on-chip: 16 KB instruction, 16 KB data cache (Harvard architecture)
    L2 on-chip: 256 KB, 512 KB, 1 MB
- **Intel® Pentium® M (2003)**
  L1 on-chip: 32 KB instruction, 32 KB data cache
  L2 on-chip: up to 2 MByte
- **Intel® Core™ i9-9900 (2019)**
  Multicore: 8 cores. Private caches (L1: 512KiB) and shared caches (L2: 2 MiB)
  L3: 16 MiB smartcache: All cores share this cache.