Information, Computation and Communication

Memory Hierarchies

Stream of Data

Advertisement

Life

Every day in 2022:

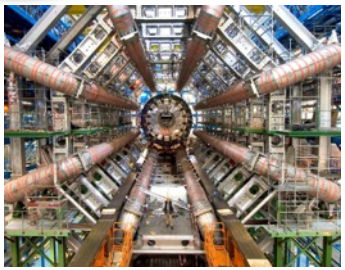
- **100 billion (10^9) WhatsApp messages**
- **333.2 billion (10^9) emails** were sent
- **1.145 trillion (10^{12}) MB** were generated

94'000'000'000'000'000'000'000 bytes in 2022 ($94 \cdot 10^{21}$ = 94 zettabyte)

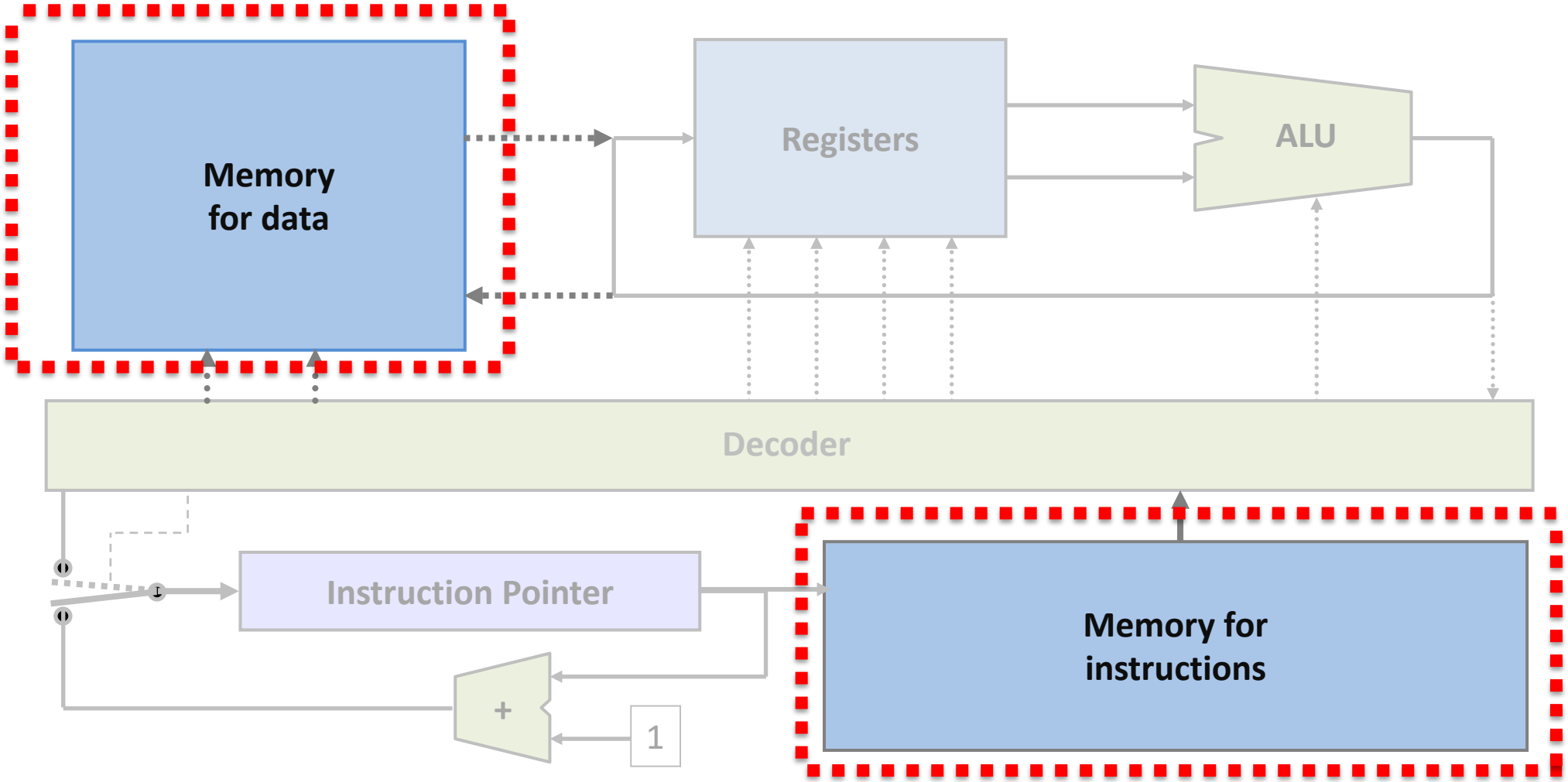
Source: https://techjury.net/

Commerce

Science



How and Where to Store Data?



Today's Objectives

- Overview of technologies used to store data (in storage or memory)
- Understand the concept of Memory Hierarchies:
  - Why are they used?
  - How are they implemented?
  - What are their implications?

Today's Agenda

1. Characteristics and Technology
2. Cache Principle
3. Functionality
4. Details
5. Example
6. Locality Principle

Part 1: Technology

Important Characteristics

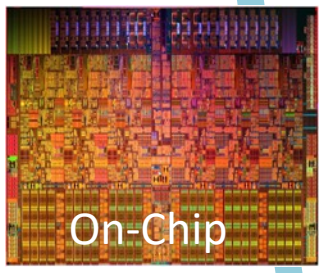
- Performance
  - **Latency** (s) = time needed to access one byte from the medium
  - **Bandwidth** (B/s) = number of consecutive bytes that can be accessed per second (also called **data rate**)
- Storage (Size)
  - **Capacity** (B) = number of bytes that can be stored on the medium
  - **Costs** (CHF/B) = costs per unit (e.g., GB) of the medium
- Retention
  - **Volatile** memory only maintains data if device is powered
  - **Non-volatile** memory maintains data if power is lost.

Technologies

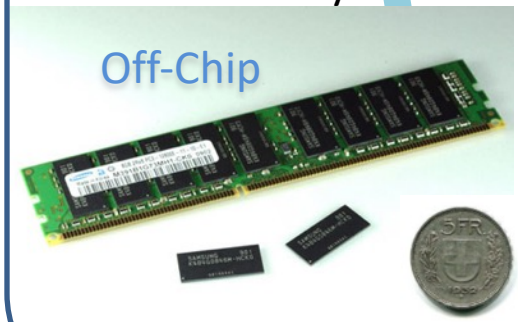
Volatile

Registers

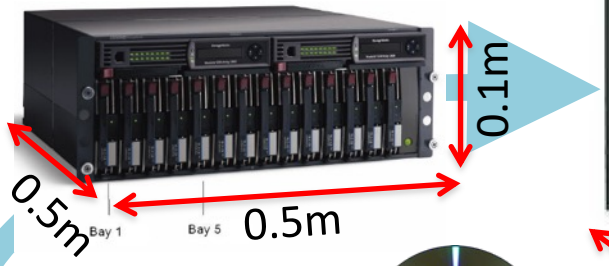
Cache Memory



RAM **Random Access Memory**
or Main Memory



Hard Disk Array



Magnetic tape robot

Hard Disk

SSD (Solid State Drive)

CD, DVD

USB FLASH

Non-volatile

Distance to the processor

Capacity

	Device	Text	Image	Audio	Video
10KB		2 pages			
100KB			1 photo		
1MB		1 book	1 photo HD	1m MP3	
10MB	Cache			1m HiFi	
100MB	CDs				
1GB	DVDs			1h HiFi	1h video
10GB	Blueray, RAM				1h video HD
100GB	Flash	Library			
1TB	Hard disk				
10TB	Magnetic tape	Library of Congress (US)			1000 movies

Examples/Exercises

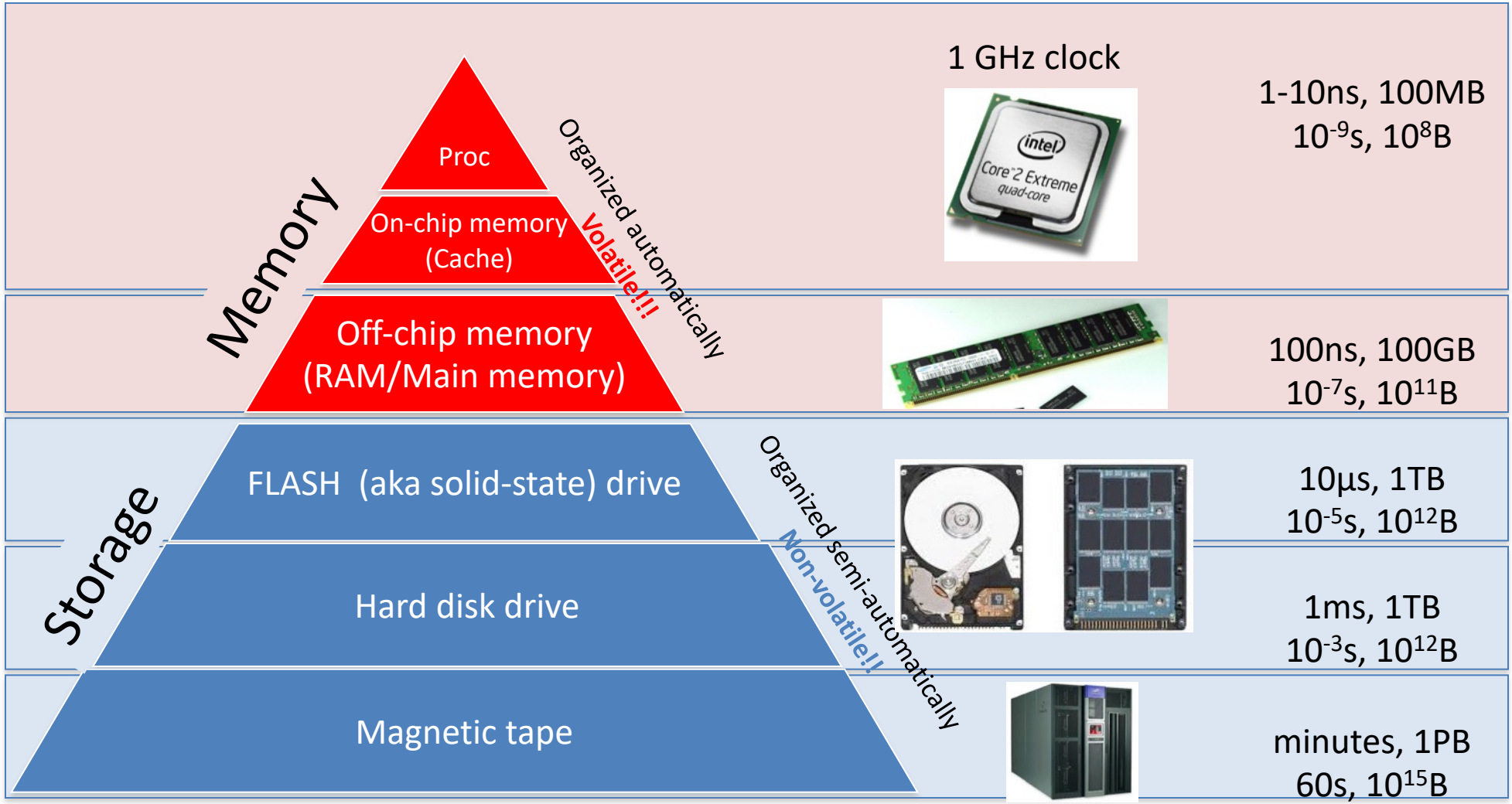
- A smartphone has a processor with 2 GHz (one clock tick every 0.5 ns) and a Flash memory with a latency of 5 µs. How many clock ticks does it take to access the memory (read some data)?
- 5 µs/0.5 ns = $5 * 10^{-6} / 5 * 10^{-10} = 10^4 = 10'000$ clock ticks

- A smartphone has also RAM (off-chip memory) that can be accessed within 100 ns. How many clock ticks does it take to read data from the RAM?
- 100 ns/0.5 ns = $100 * 10^{-9} / 5 * 10^{-10} = 200$ clock ticks

50x faster

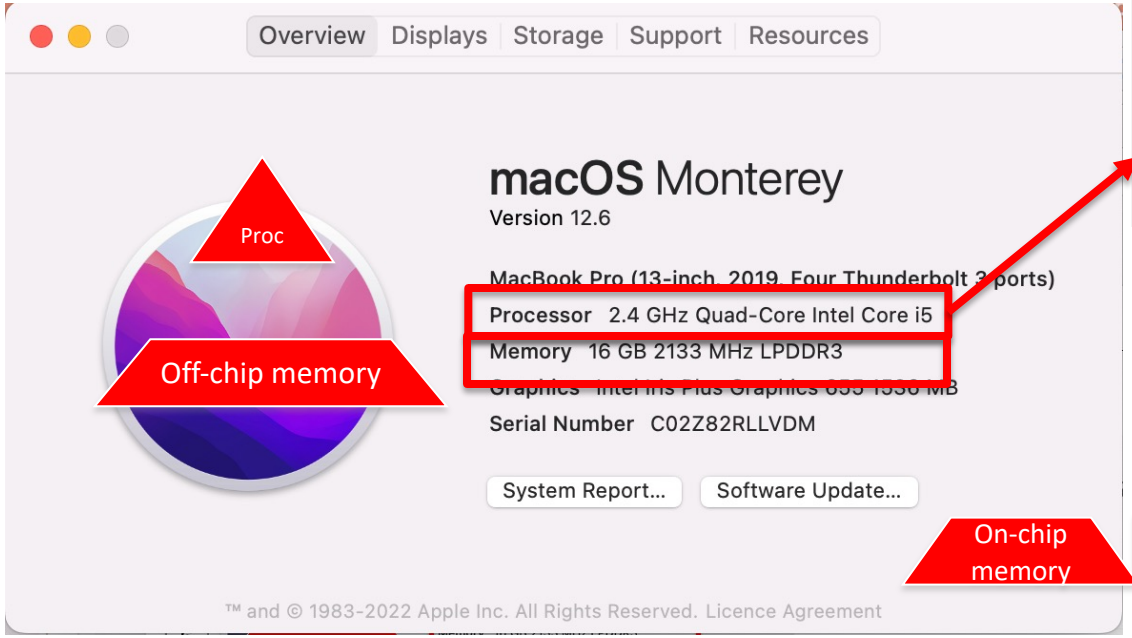
- Data in the registers can be used immediately, without delay.

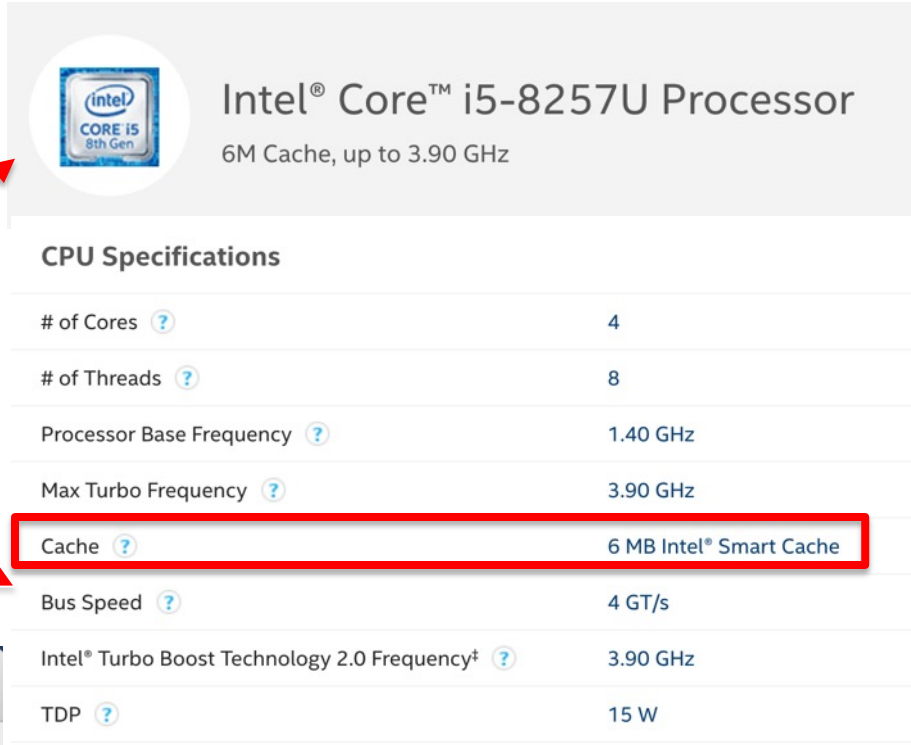
Memory/Storage Hierarchy



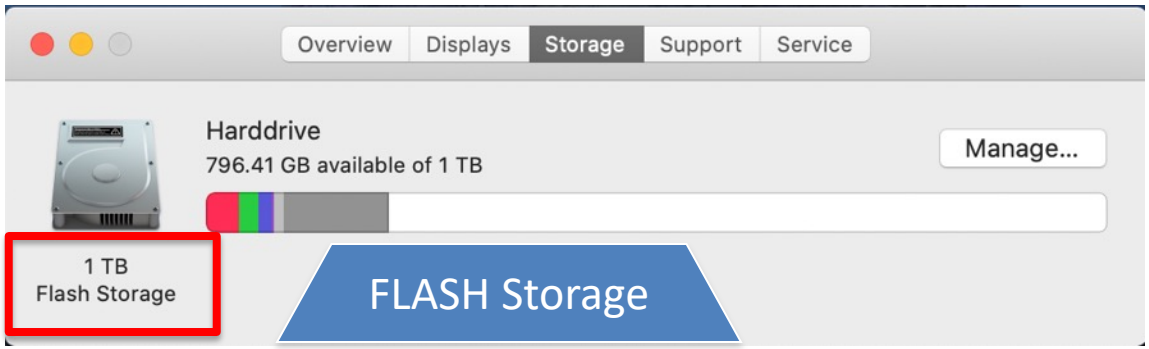
Memory forgets the data when it is turned off.
Storage remembers the data when it is turned off.

Example: My Computer





On-chip memory



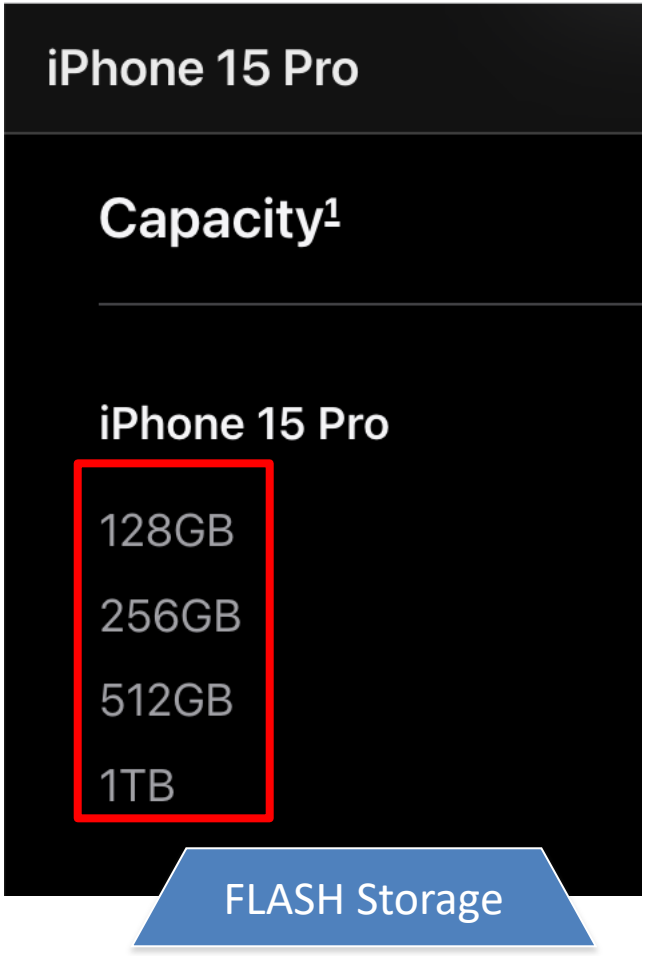
(1) Processor 2.4 GHz: Cache size: 6MB; Access time: $1/2.4 * 10^{-9} \approx 0.4$ns

(2) Memory (RAM): Size 16 GB; Access time: 10-100ns (Bandwidth: GB/s)

(3) Storage (flash, hard drive): Size 1TB; Latency: µs (Bandwidth: GB/s)

iPhone 15 Pro

https://www.apple.com/iphone-15-pro/specs/ and Wikipedia



Apple A17 Pro

General information	
Launched	September 12, 2023
Designed by	Apple Inc.
Common manufacturer(s)	TSMC[1]
Product code	APL1V02
Max. CPU clock rate	to 3.78 GHz[2]
Cache	
L2 cache	16 MB (performance cores) 4 MB (efficiency cores)
L4 cache	24 MB (system level cache)
Architecture and class	
Application	Mob
Technology node	3 nm (N3B)
Instruction set	ARMv8.6-A[3]
Physical specifications	
Transistors	19 billion
Cores	6 (4 efficiency, 2 performance)[4]
Memory (RAM)	8GB
GPU(s)	
History	
Predecessor(s)	Apple A16

Proc

Cache

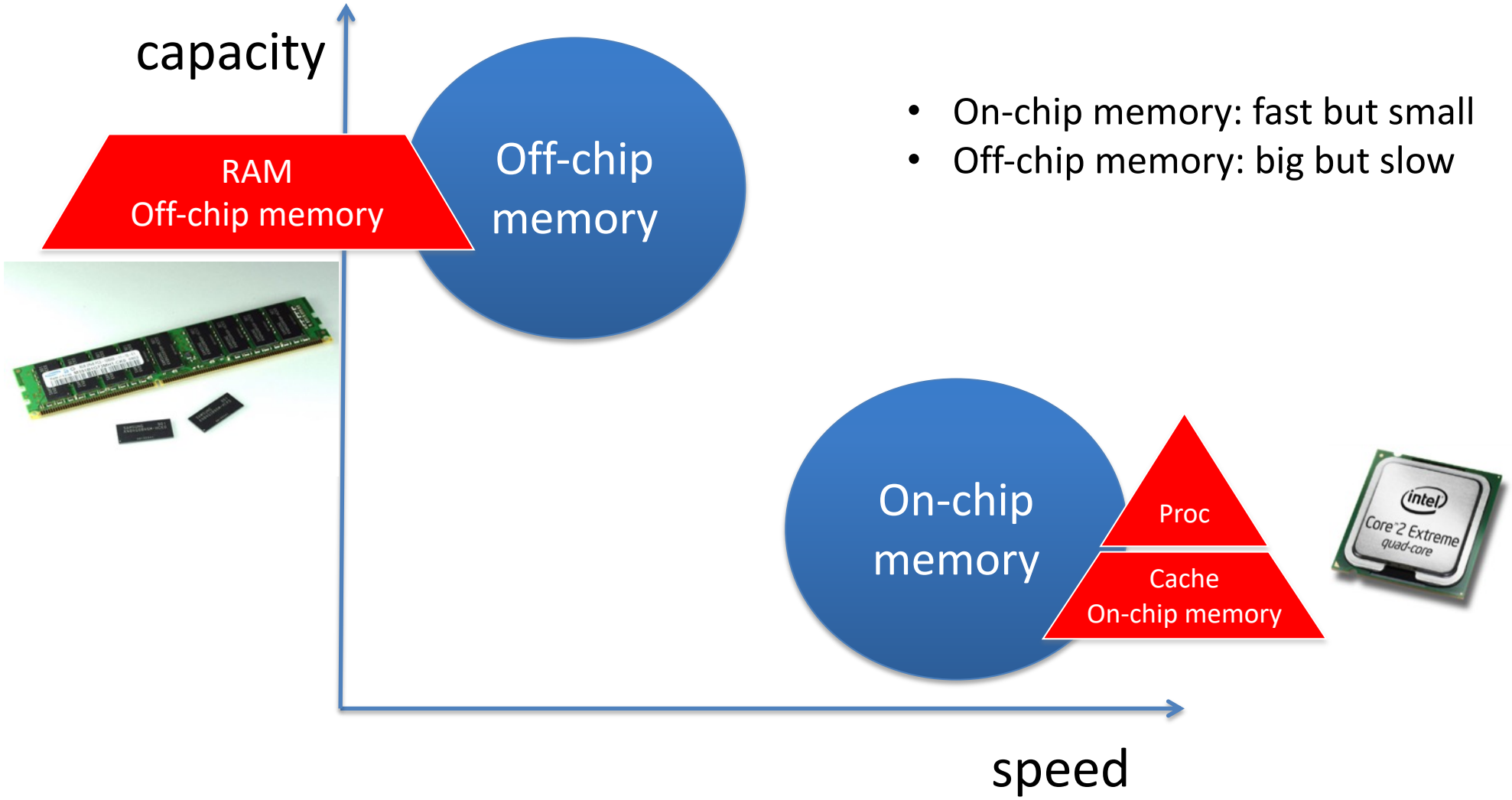
RAM

Part 2: Cache Principle

Ideal Memory

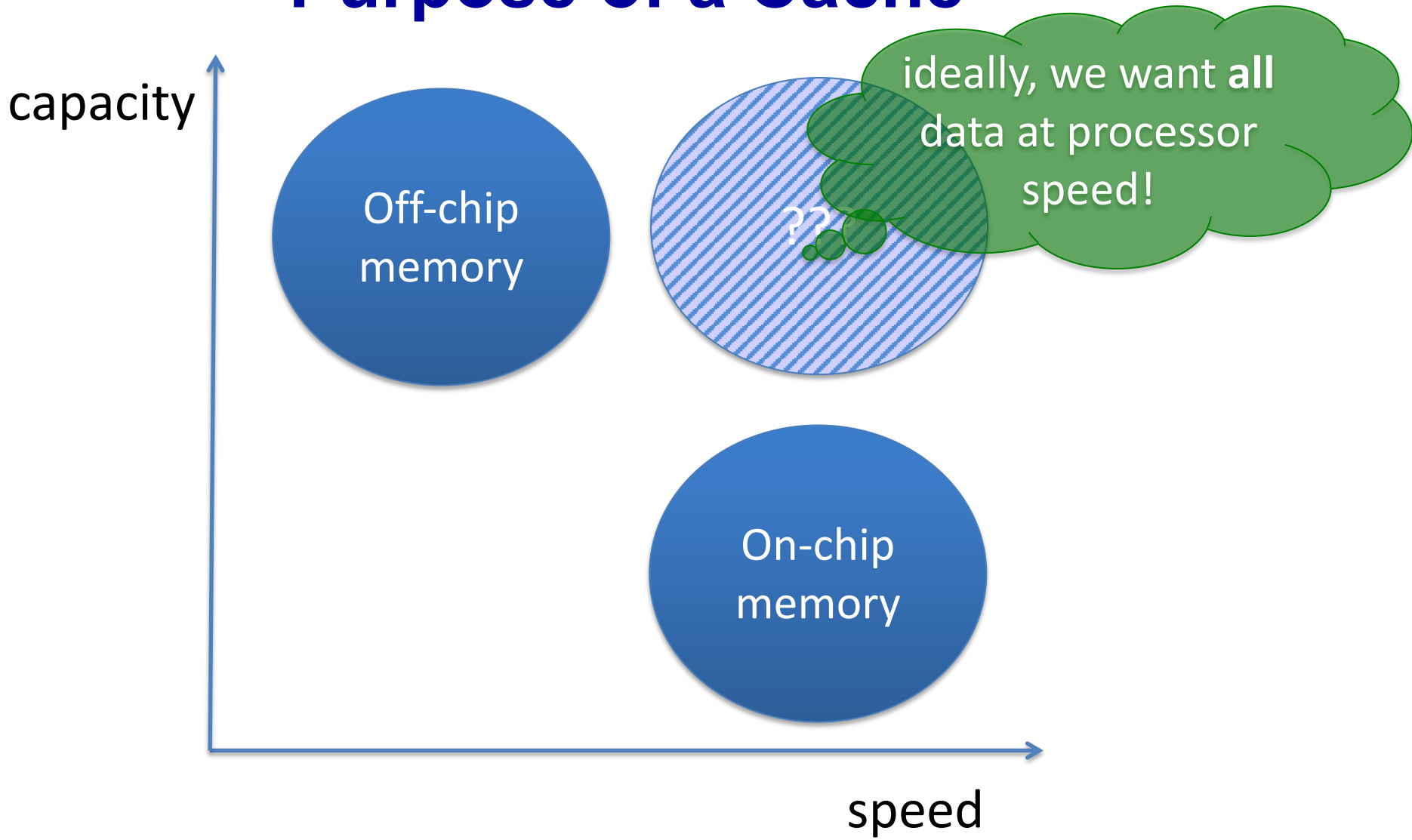
- Big (has large capacity)
- Fast (is accessible at high speed)
- Not possible…
  - Too expensive
  - Consumes too much energy
  - Occupies too much space

Situation: Technology Offers Us



- On-chip memory: fast but small
- Off-chip memory: big but slow

Purpose of a Cache



Analogy with Real Life

Clothes in your suitcase

- Limited space
- Temporary use
  - Time for a trip
  - According to the climate
  - Selected colors
- With you

... **but in your closet**

- All your clothes
- Permanent
- Only at home

What Happens in this Example?

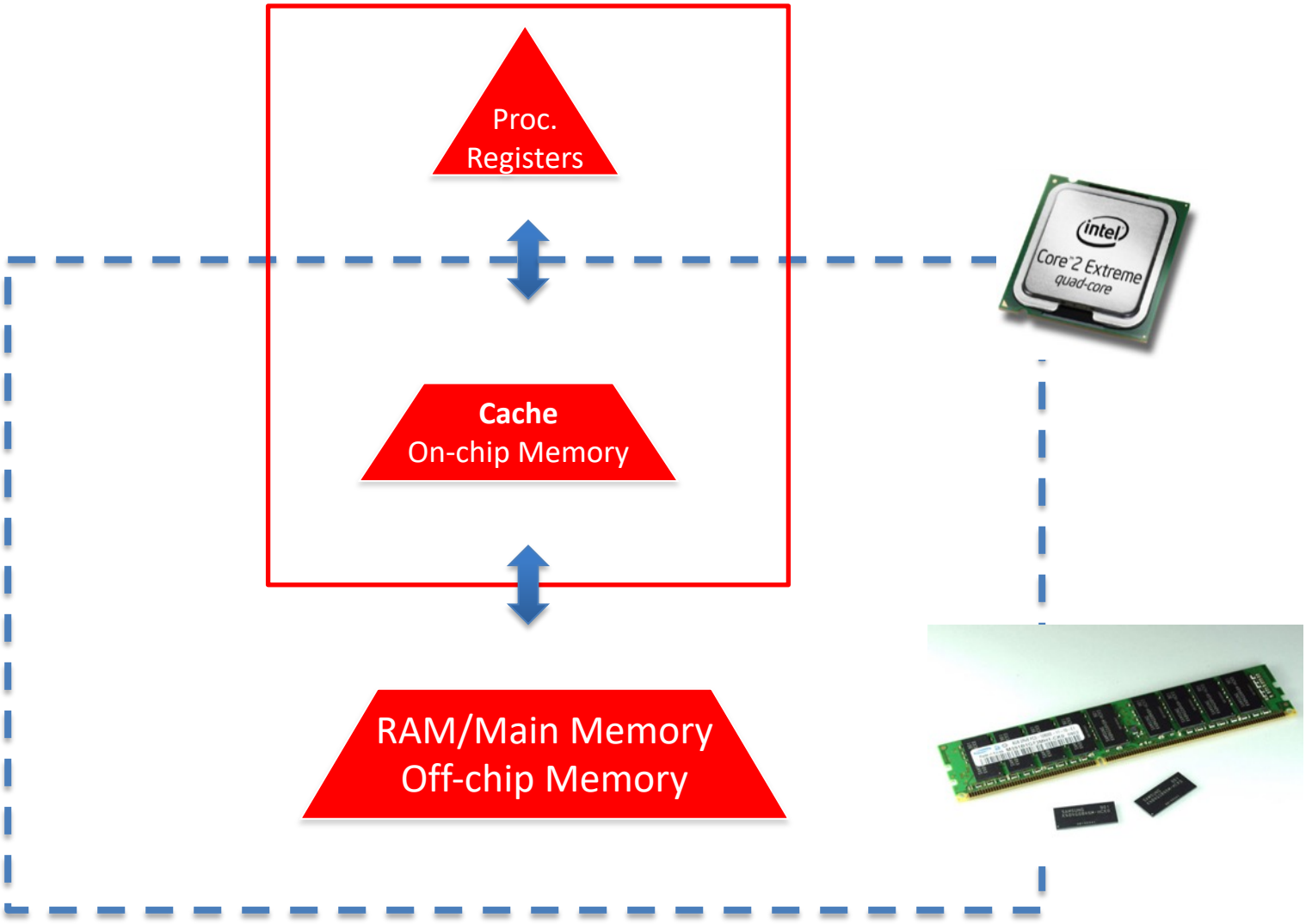
- We move things from one level of “memory” to the next
- We put in the “nearest memory” what we believe we will need soon

What Happens in a Computer's Memory

- Data passes from one level to the other
- We try to keep in the fast memory the data that is going to be used soon
- This is the principle of caching

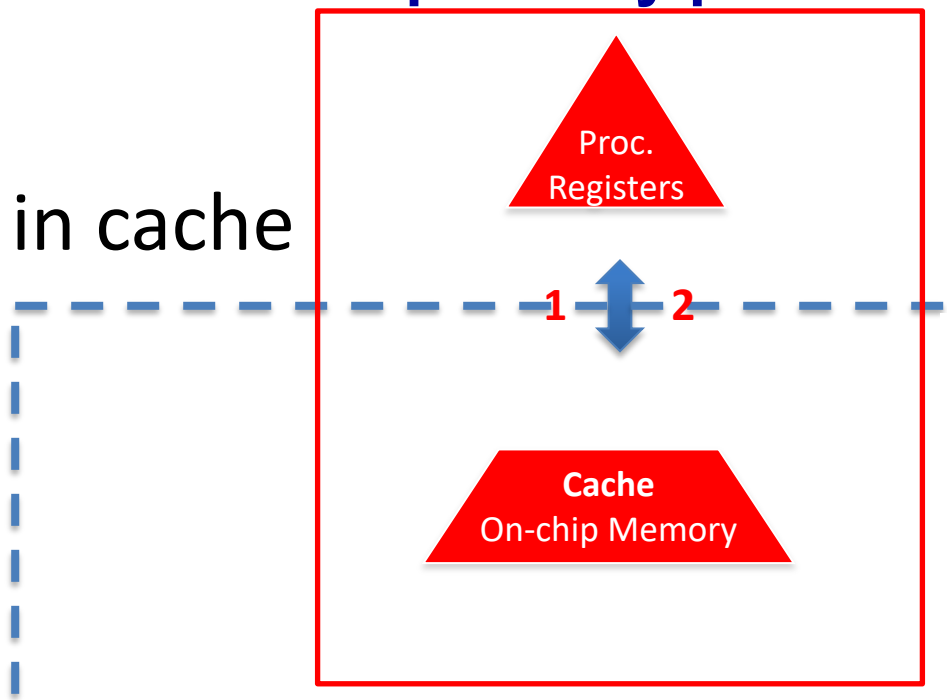
Part 3: Cache Functionality

Cache and Main Memory



Cache and Main Memory:

if information required by processor is in cache



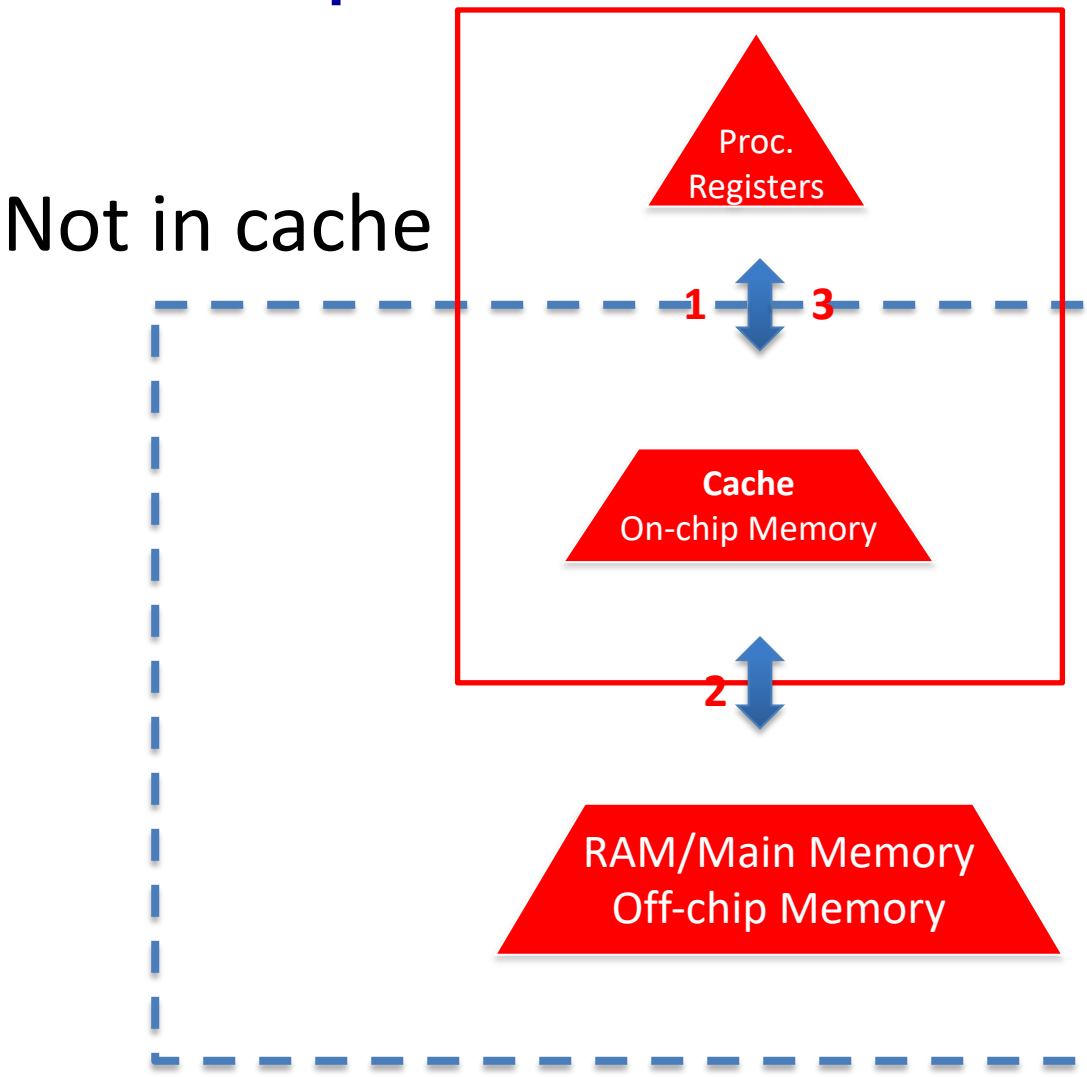
in cache

immediate access

(1) Processor is asking for data

(2) Cache responds immediately

Cache and Main Memory:
if required information is not in cache



(1) Processor is asking for data

(2) Access request forwarded to RAM

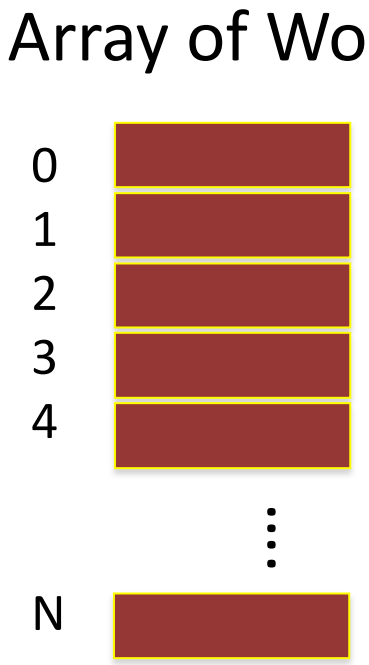
(3) Processor is waiting for response

Part 4: Memory Hierarchy in Detail

Memory from Processor's Point of View

- Memory consists of **words**
  - Typically, 4 or 8 bytes (32 or 64 bits)
- Every word has an address between 0 and N

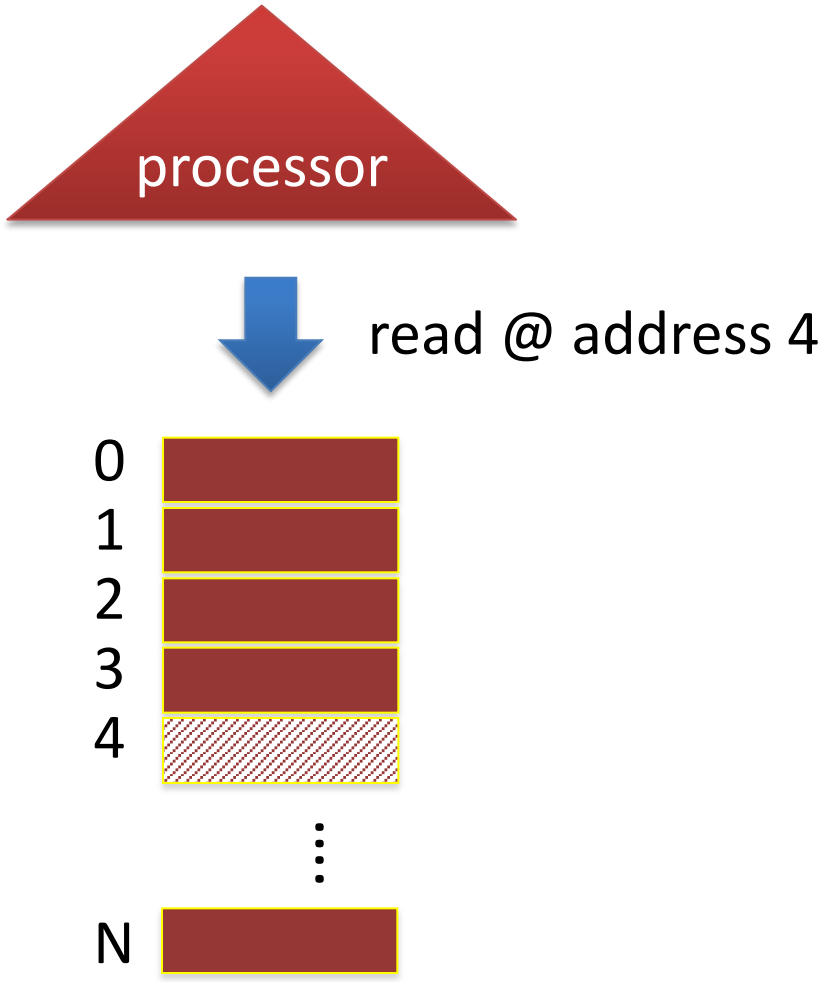
Memory from Processor's Point of View



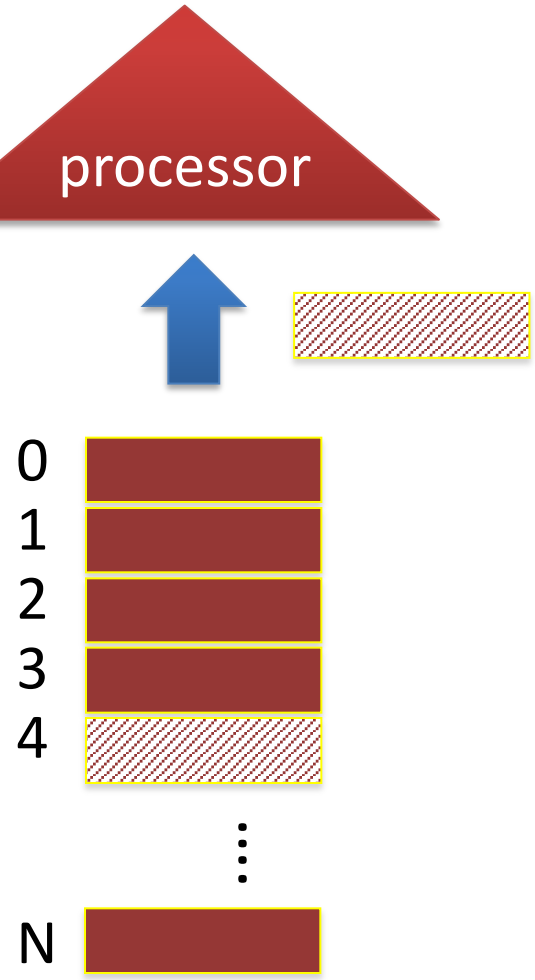
Memory from Processor's Point of View

- Processor asks the memory for the word in address A
- Memory returns the content of the word to the processor
- (Typically) the processor places it in a register

Memory from Processor's Point of View

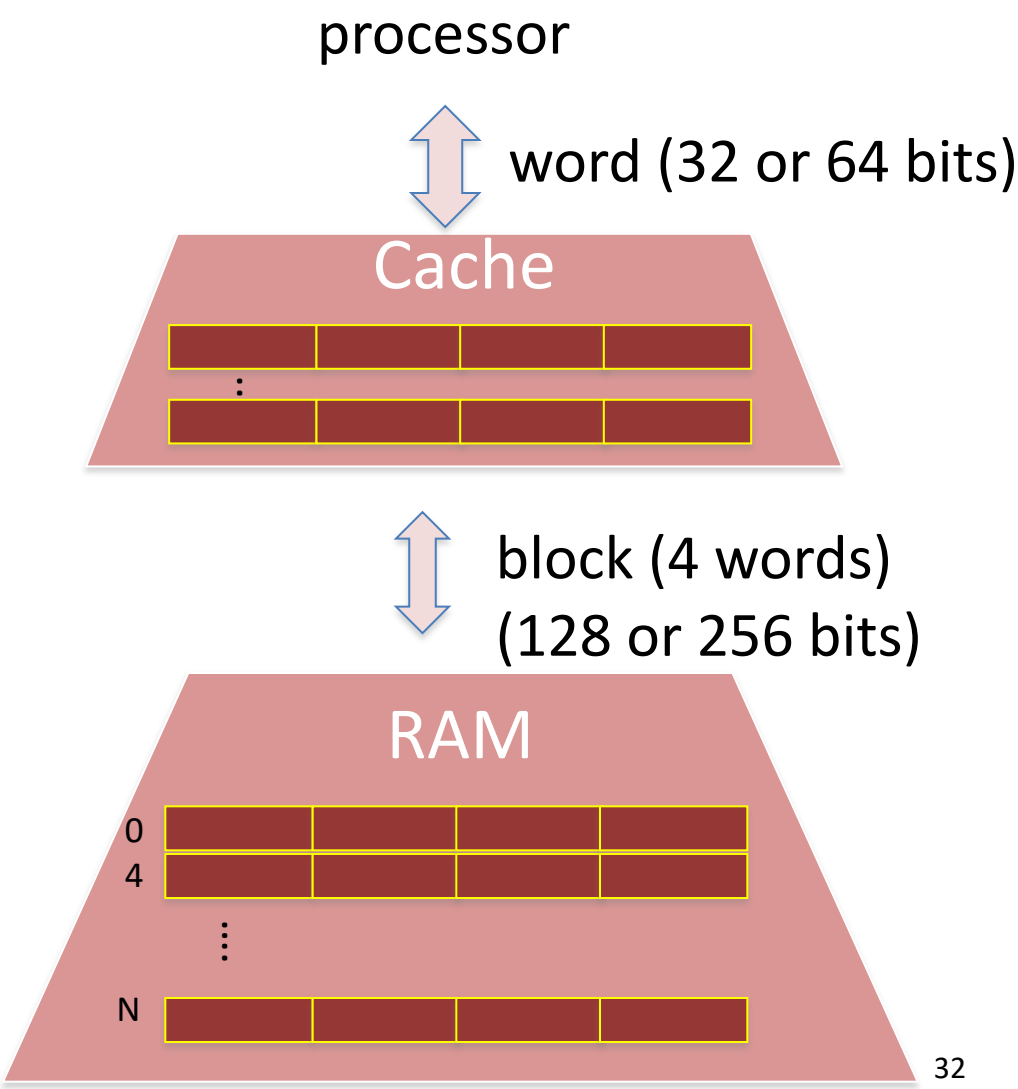


Memory from Processor's Point of View



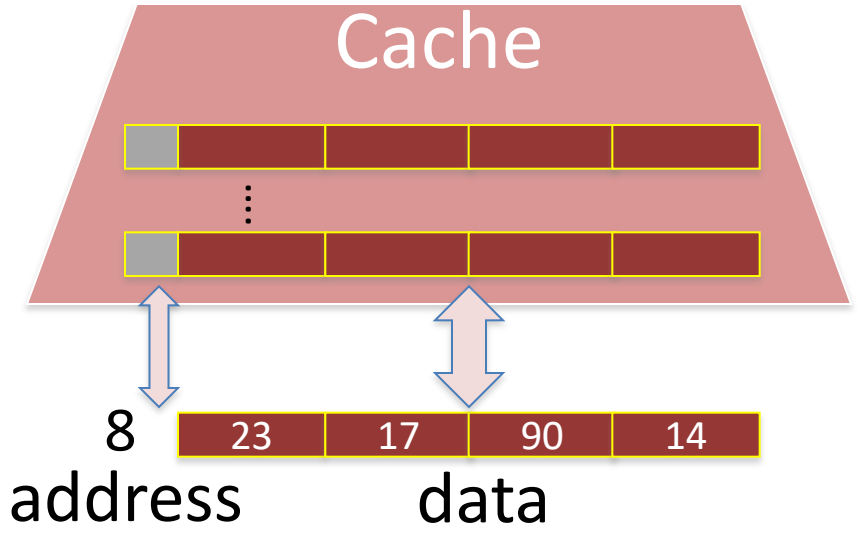
Physical Memory Implementation

- Unit of transfer: a **block**
- The cache is organized in blocks
- The RAM is organized in blocks
- The transfer between the two is done in blocks



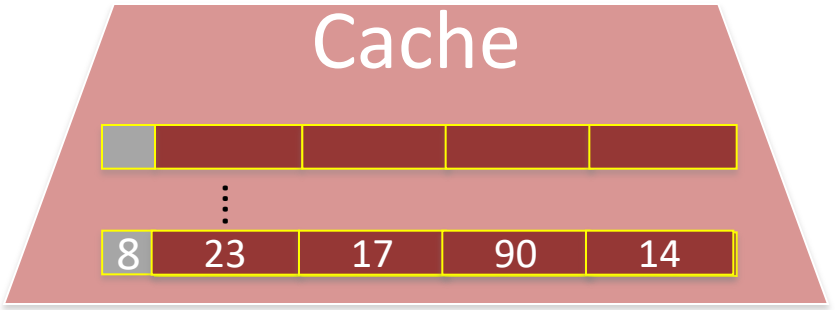
Physical Memory Implementation

- How to we know which blocks of the RAM are in the cache?
  - Each block stores its address
- Transfer the data
- Insert the address



Physical Memory Implementation

- How to we know which blocks of the RAM are in the cache?
  - Each block stores its address
- Transfer the data
- Insert the address



Six Questions

How does the processor **read** a word…

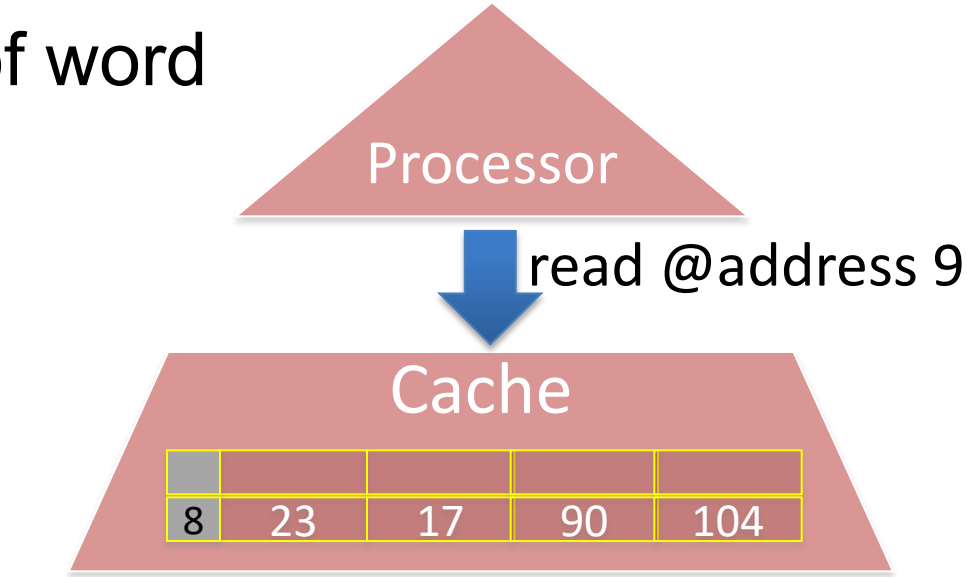
1. …if it is **in** the cache?
2. …if it is **not in** the cache?

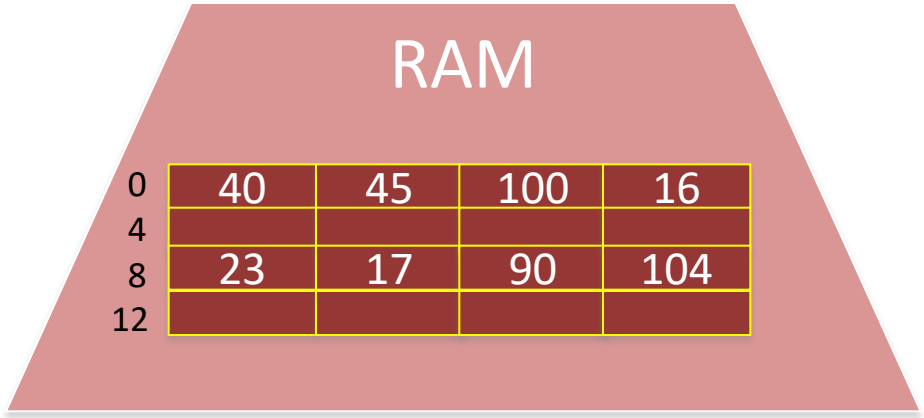
How does the processor **write** a word...

3. …if it is **in** the cache?
4. …if it is **not in** the cache?
5. What happens if the cache is full? (Replace a block)
6. What happens if replaced block has been modified?

1. Read a Word in Cache

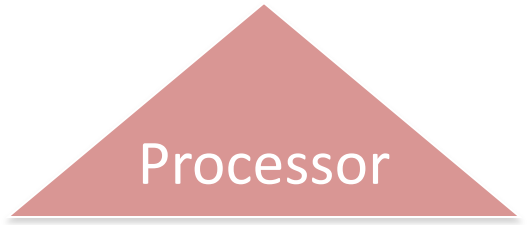
1. Processor sends address of word

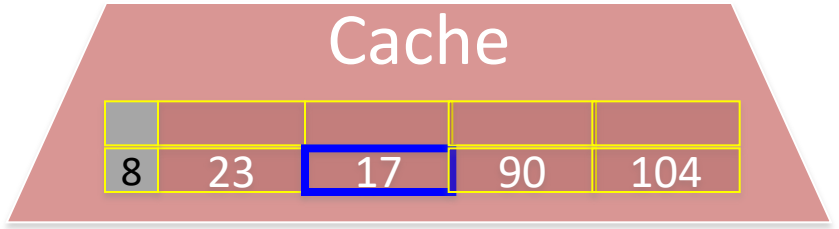


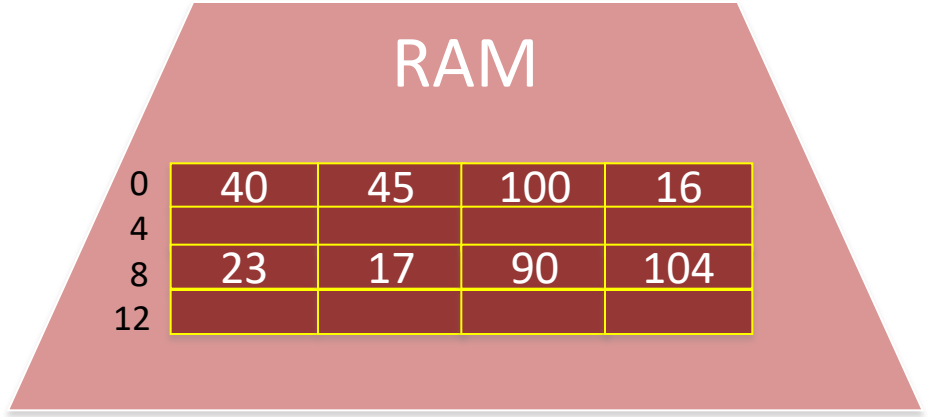


1. Read a Word in Cache

1. Processor sends address of word
2. Cache checks if word is present

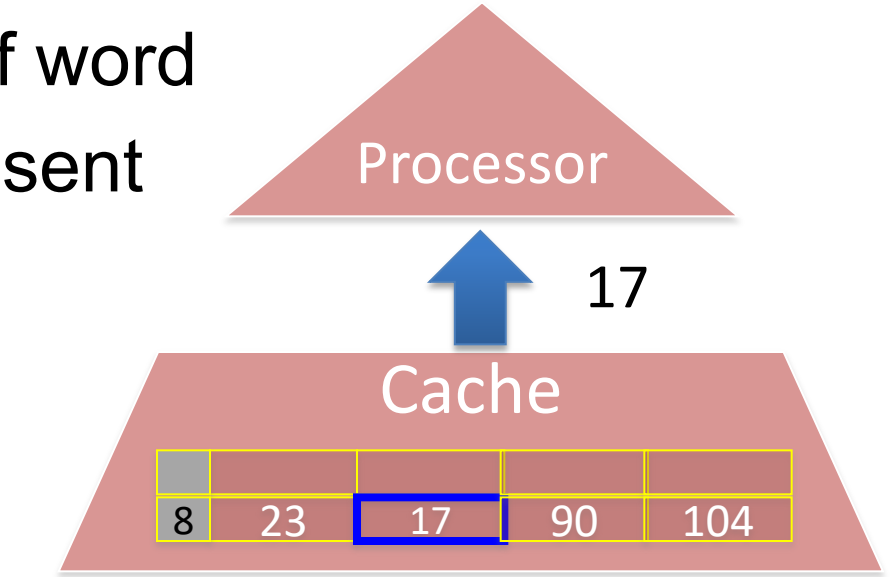


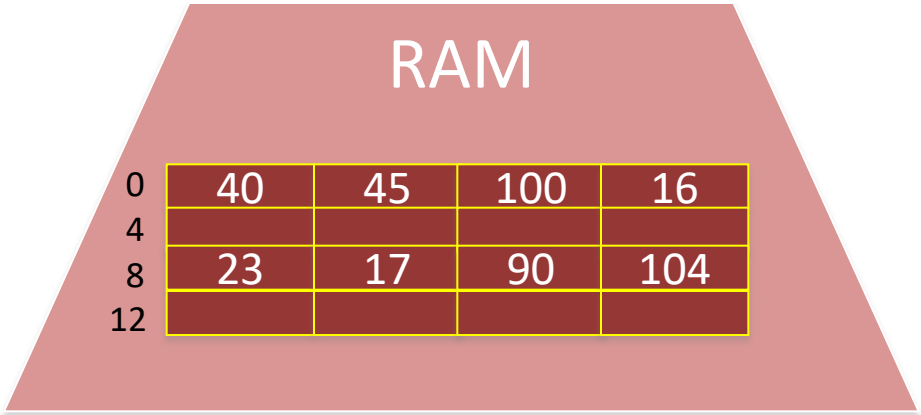




1. Read a Word in Cache

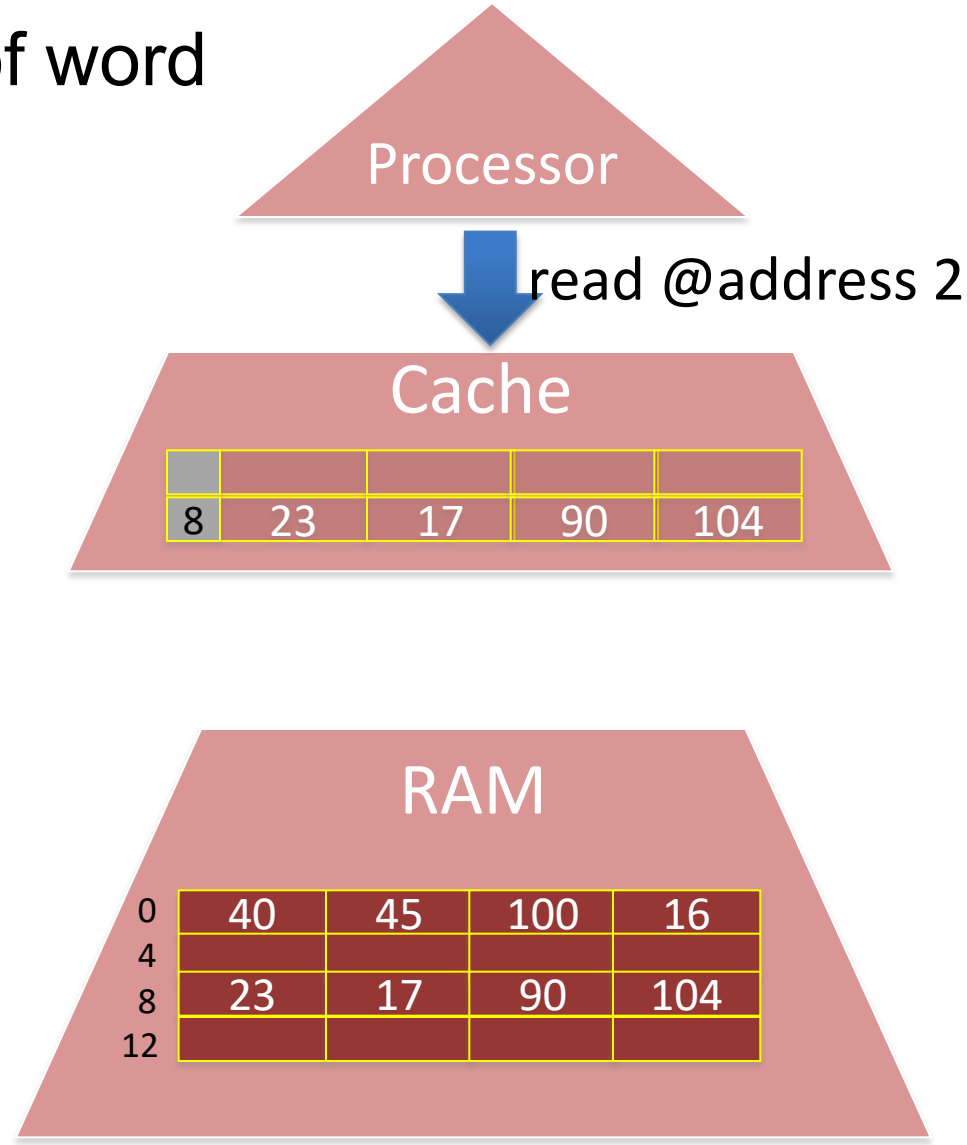
1. Processor sends address of word
2. Cache checks if word is present
3. If yes, cache sends word





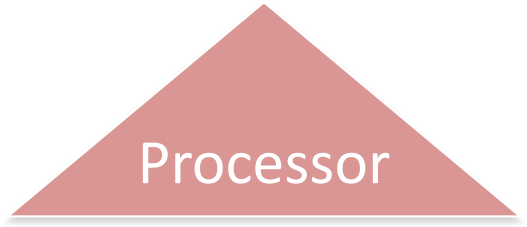
2. Read a Word not in Cache

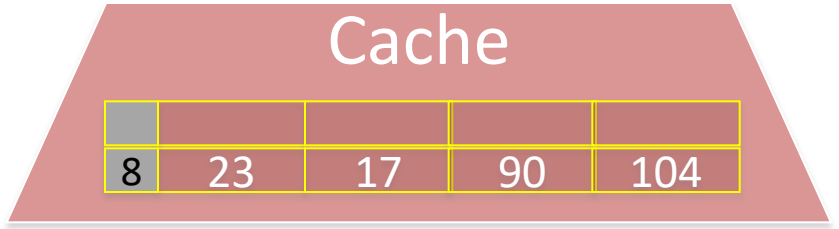
1. Processor sends address of word

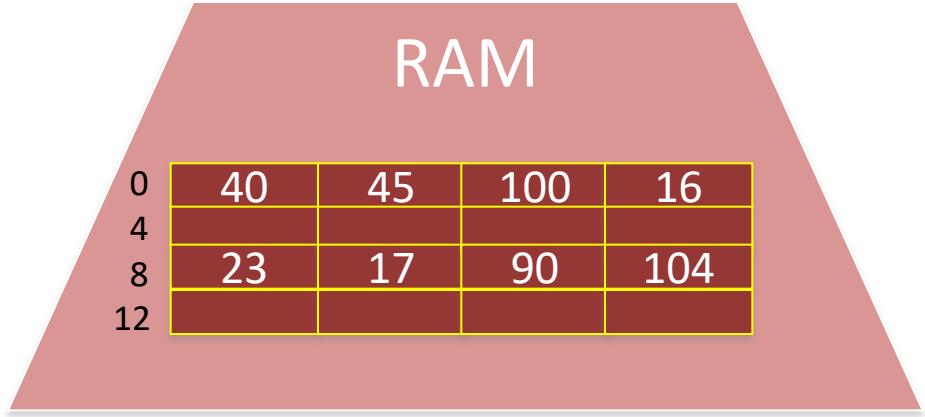


2. Read a Word not in Cache

1. Processor sends address of word
2. Cache checks if word is present

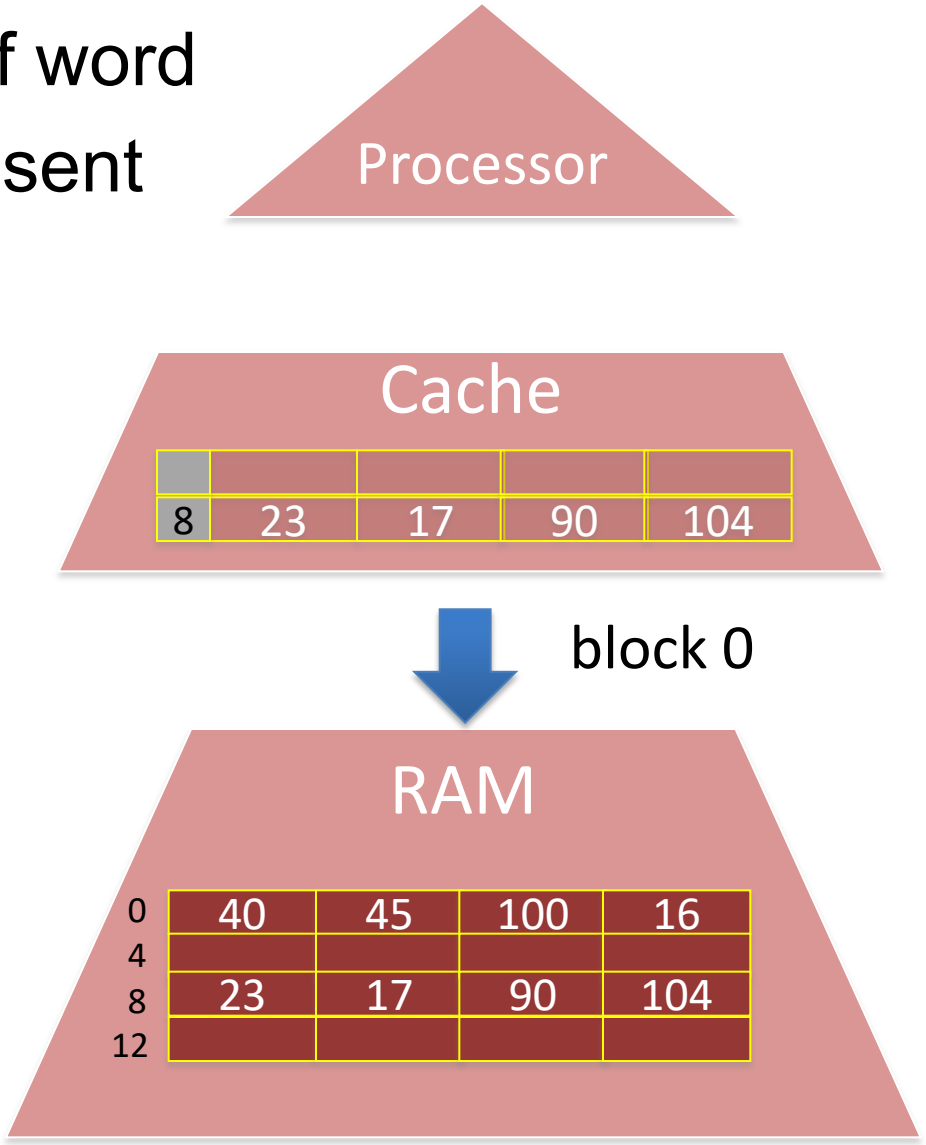






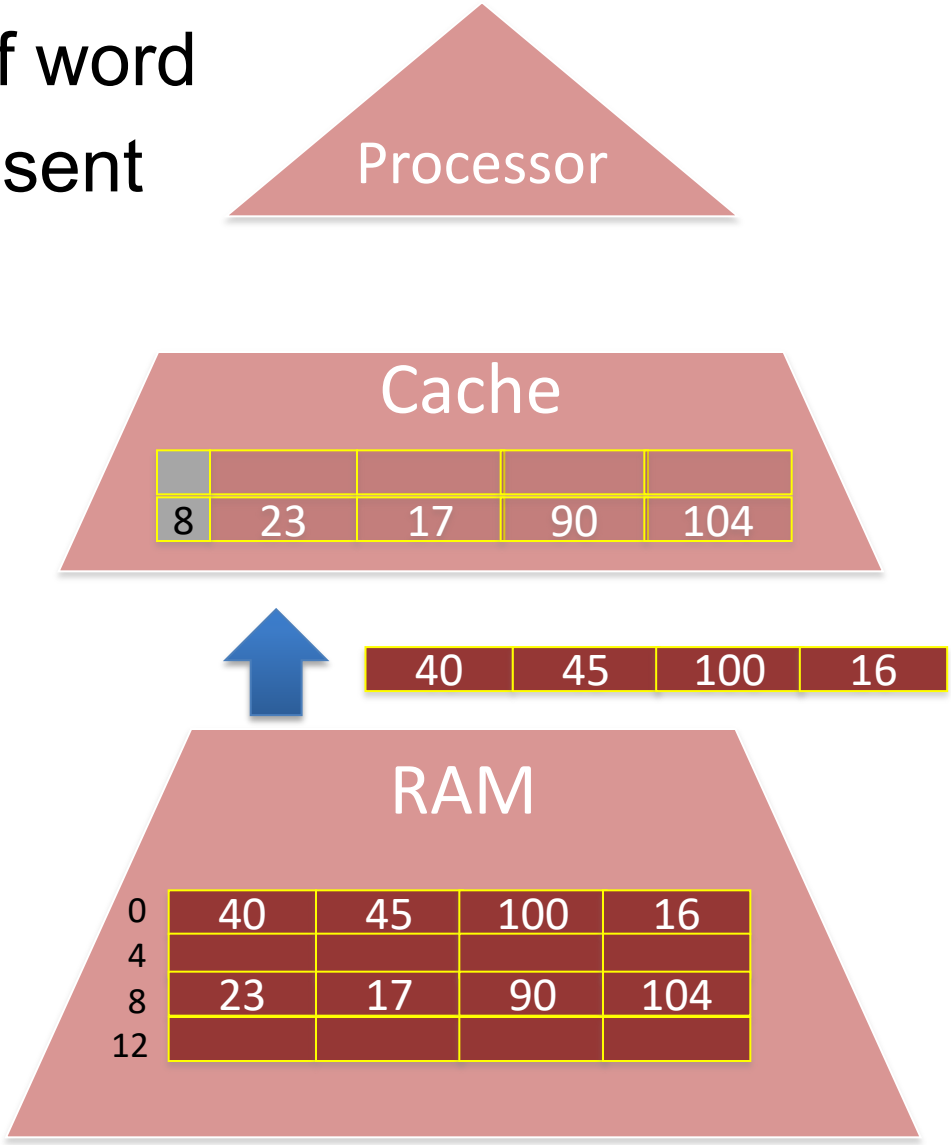
2. Read a Word not in Cache

1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   cache loads memory block
   from RAM



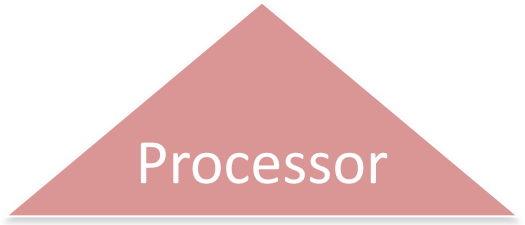
2. Read a Word not in Cache

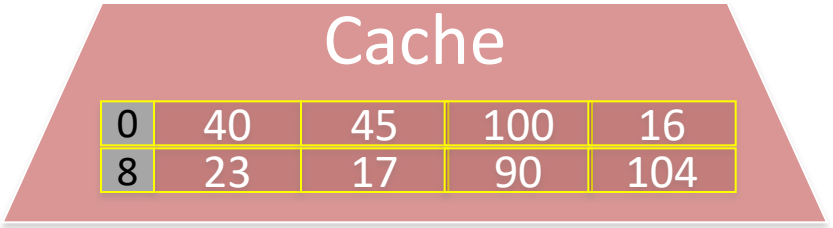
1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   cache loads memory block from RAM
4. RAM returns block

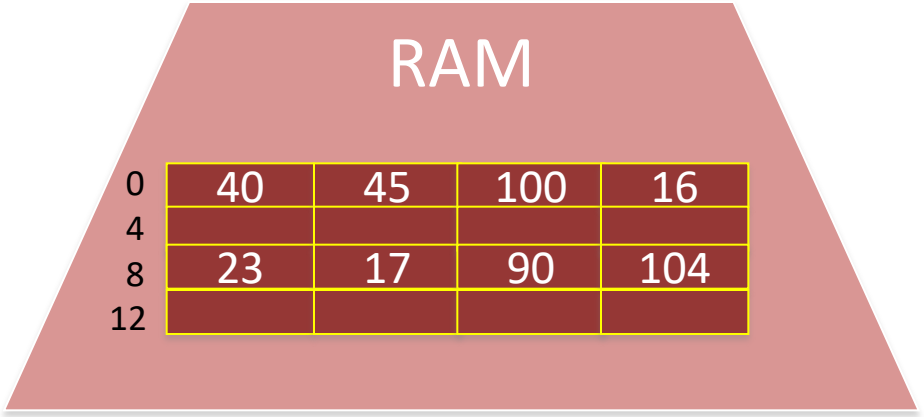


2. Read a Word not in Cache

1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   cache loads memory block from RAM
4. RAM returns block
5. Cache stores block+address

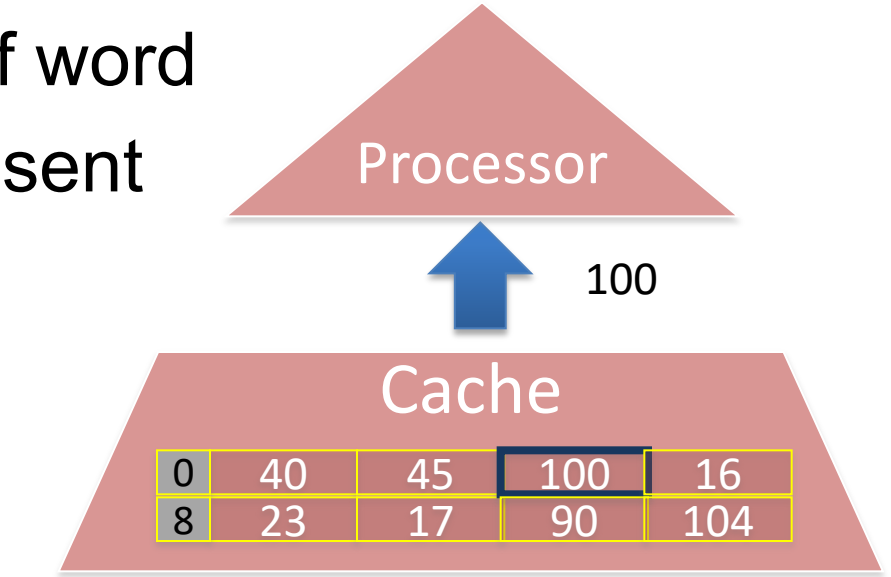


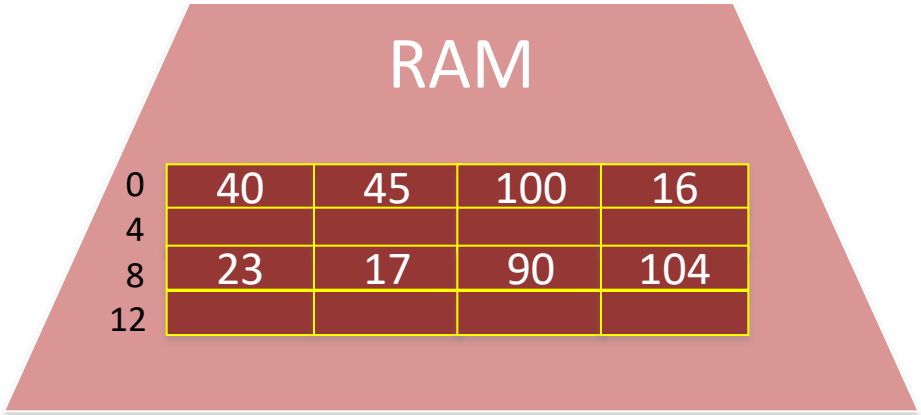




2. Read a Word not in Cache

1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   cache loads memory block from RAM
4. RAM returns block
5. Cache stores block+address
6. Cache sends word



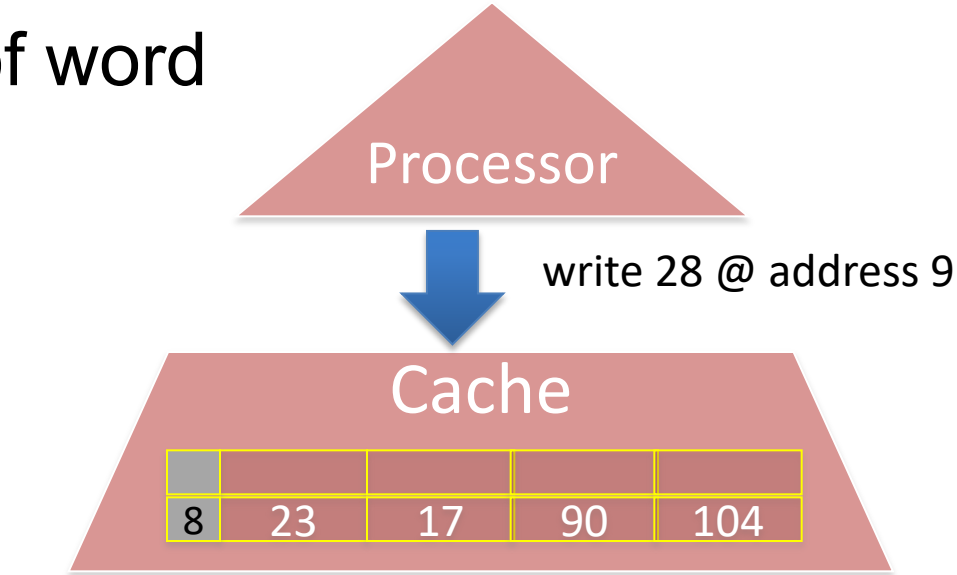


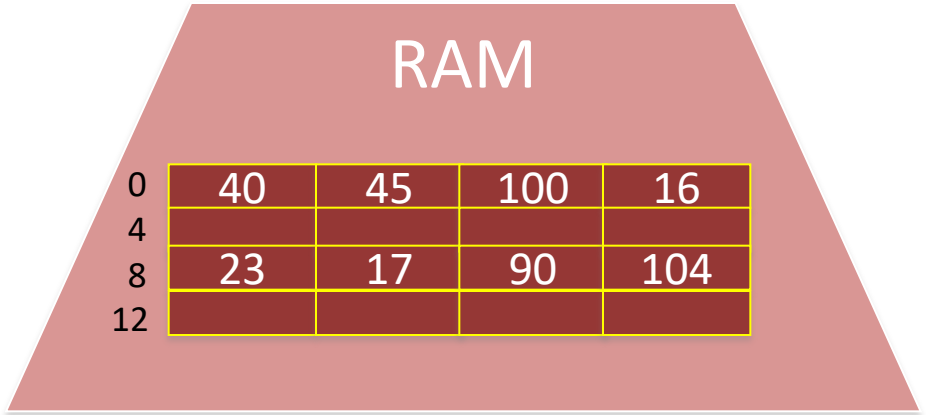
How does Processor Write a Word?

- It is almost identical to a read!

3. Write a Word in Cache

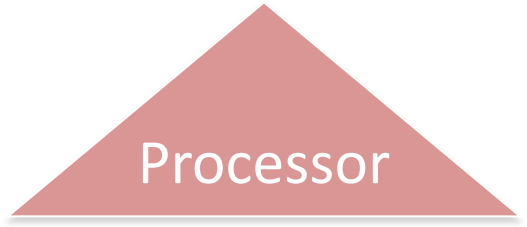
1. Processor sends address of word

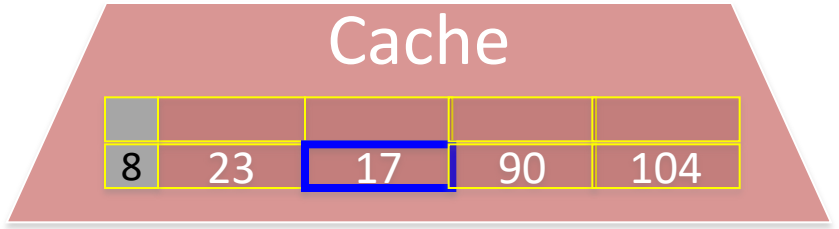


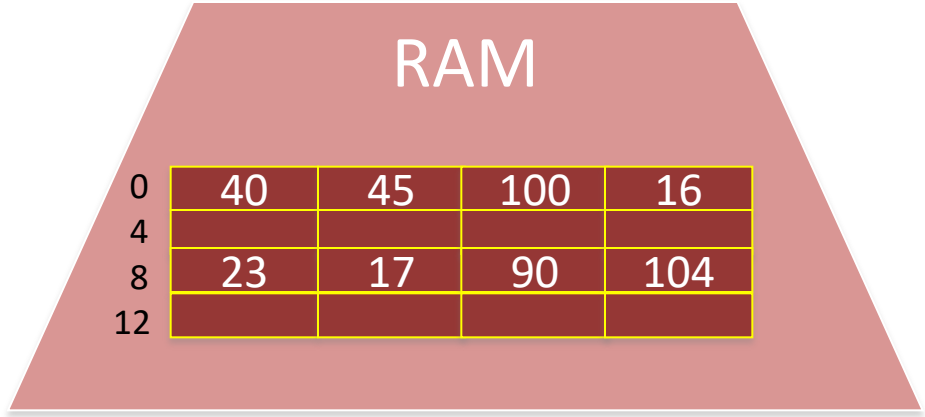


3. Write a Word in Cache

1. Processor sends address of word
2. Cache checks if word is present

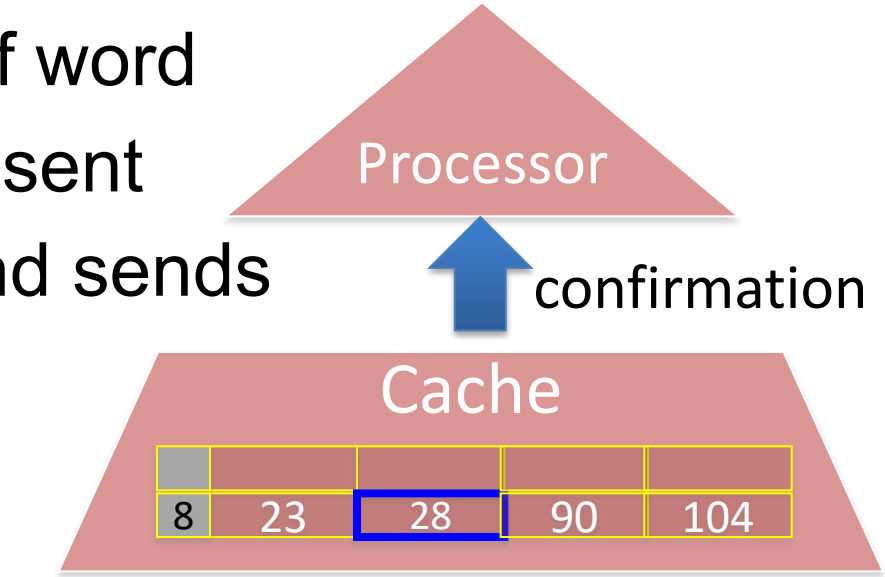


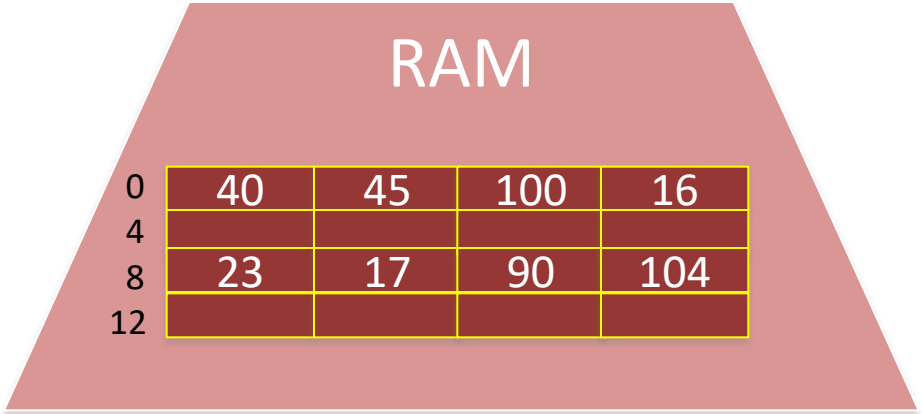




3. Write a Word in Cache

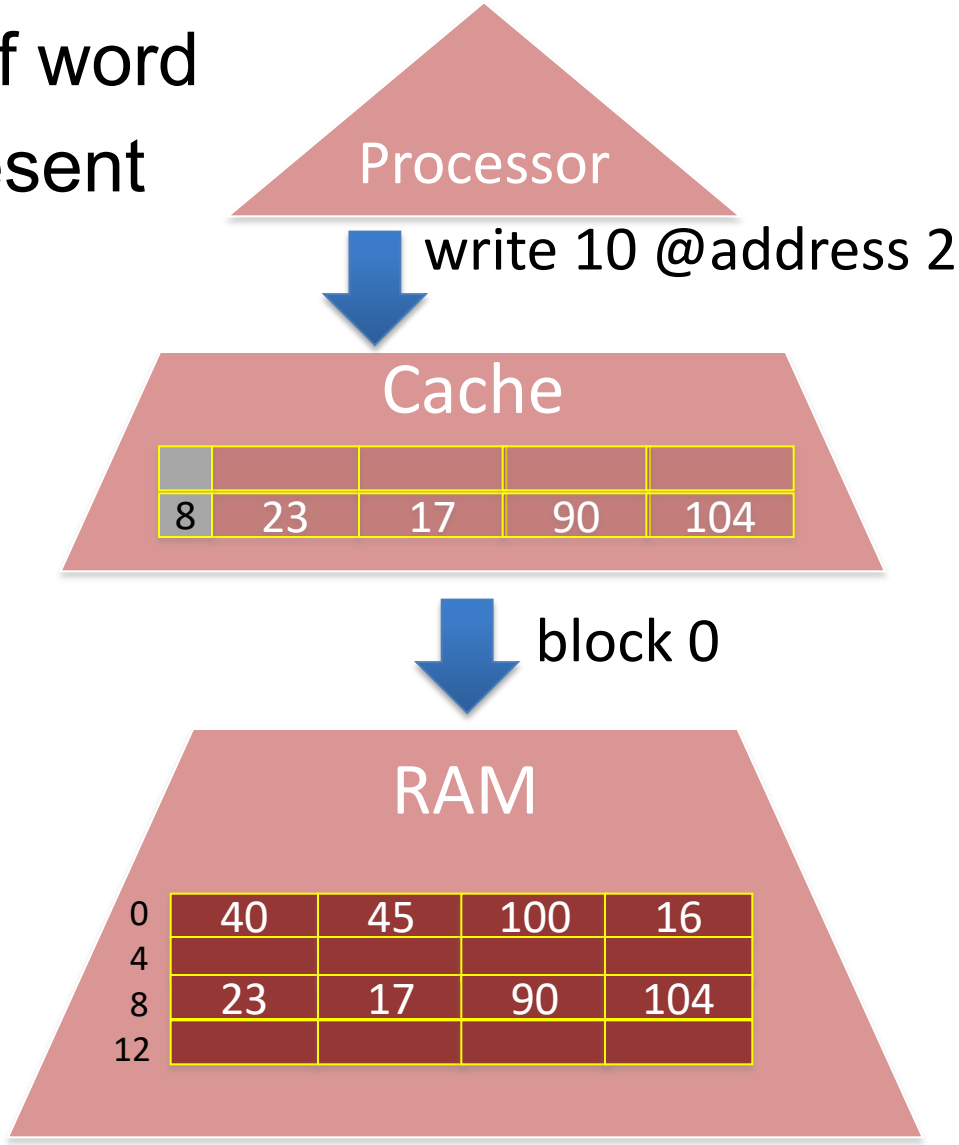
1. Processor sends address of word
2. Cache checks if word is present
3. If yes, cache writes word and sends a confirmation to processor





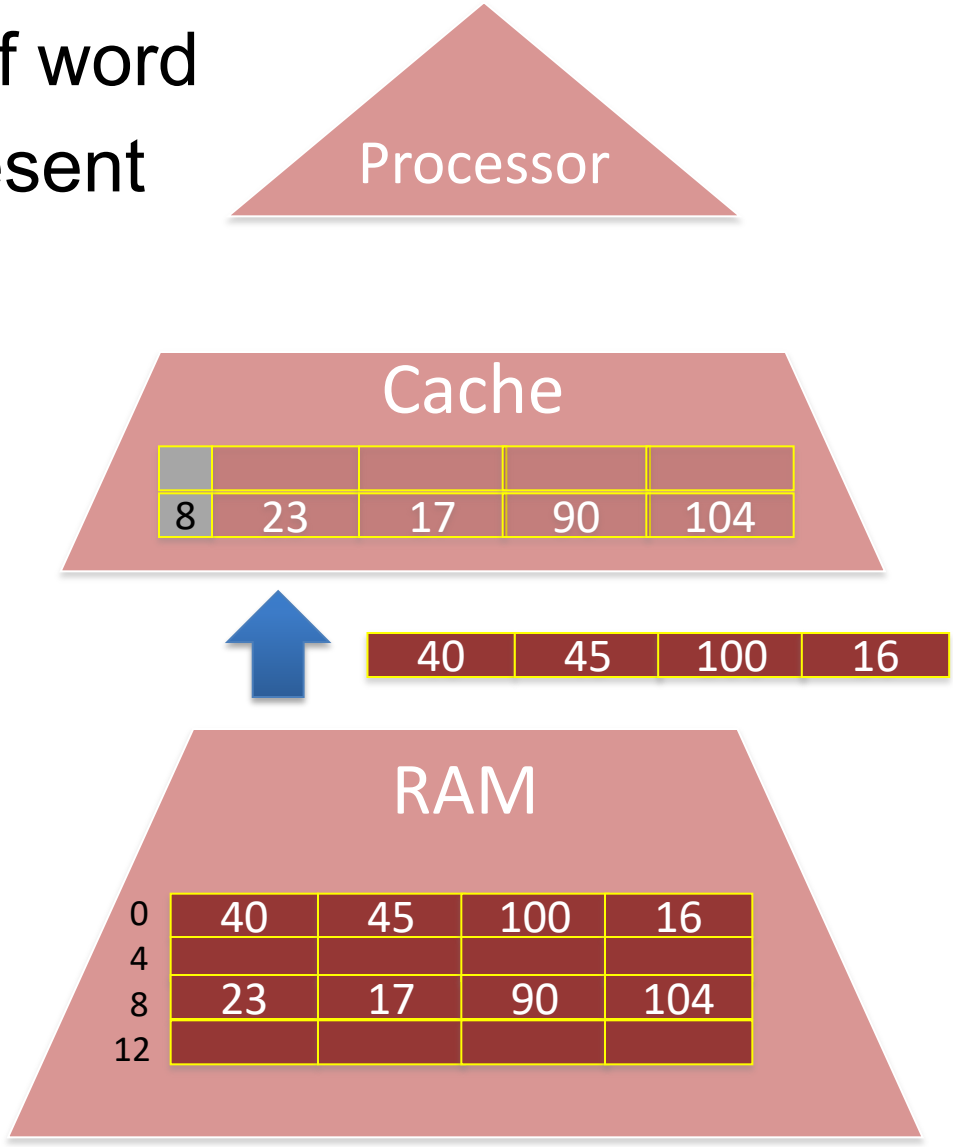
4. Write a Word not in Cache

1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   cache loads memory block
   from RAM



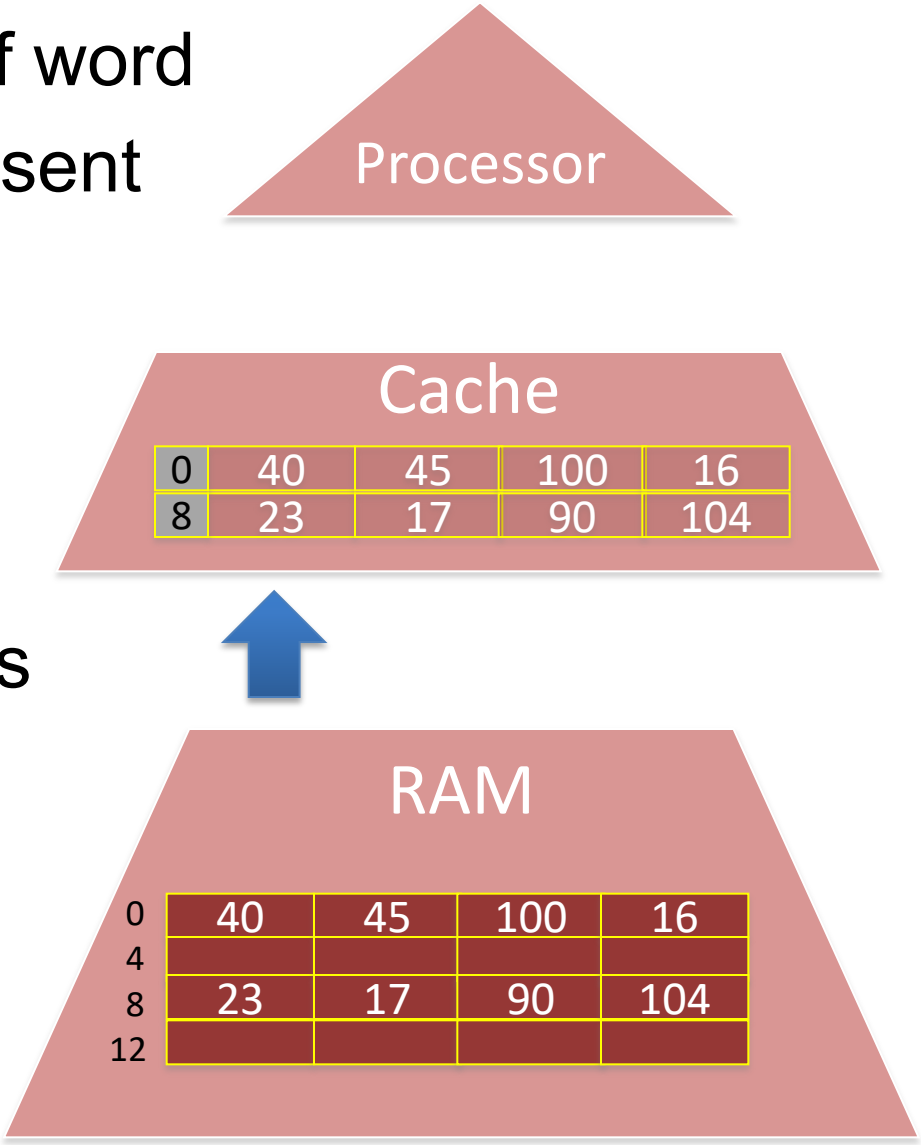
4. Write a Word not in Cache

1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   cache loads memory block from RAM
4. RAM returns block



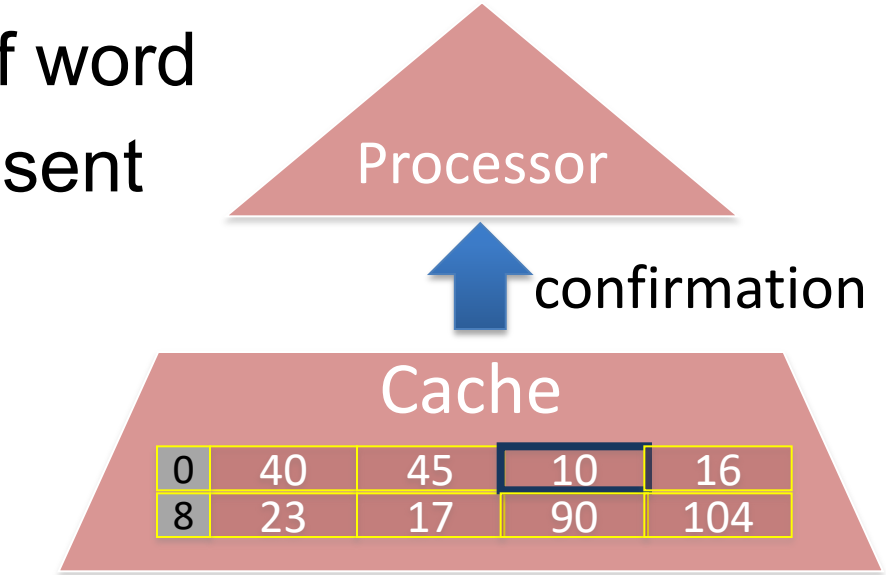
4. Write a Word not in Cache

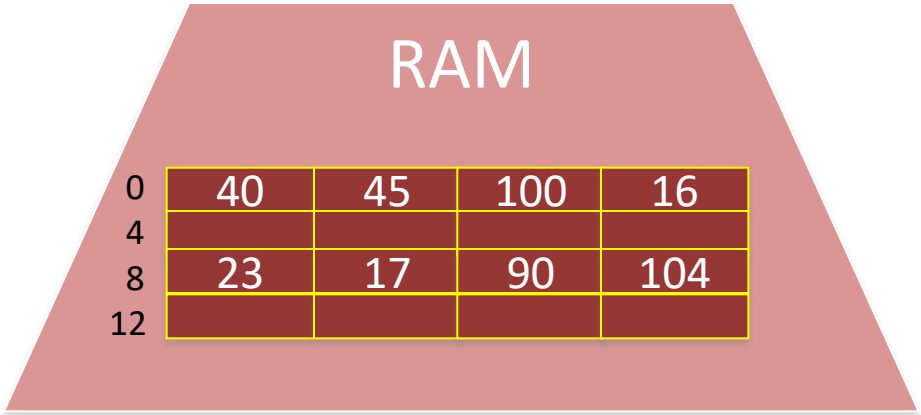
1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   cache loads memory block from RAM
4. RAM returns block
5. Cache stores block+address



4. Write a Word not in Cache

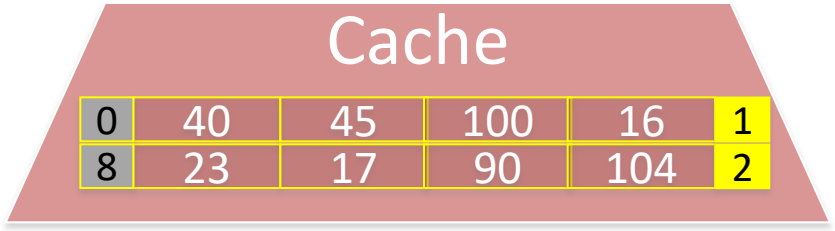
1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   cache loads memory block from RAM
4. RAM returns block
5. Cache stores block+address
6. Cache writes word and sends confirmation to processor





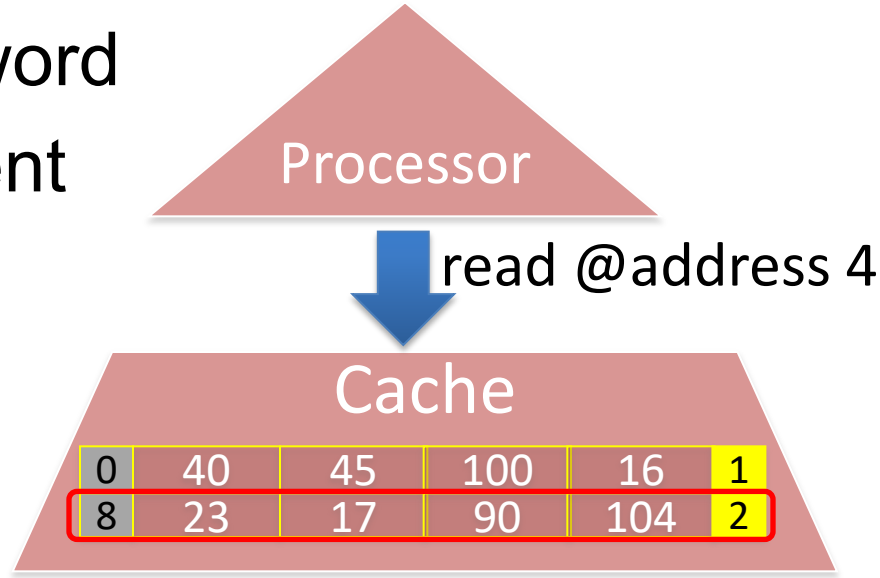
5. What Happens if Cache is Full?

- We need to replace a block
- Which block to choose?
- There are different strategies one could choose from.
- We discuss LRU (Least Recently Used) strategy:
  - Replaces the block that used the least recently
  - Adds a counter to every block indicating how long this block was unused
  - Counter is updated at every access:
    +1 if block is unused and set to 1 is block is used

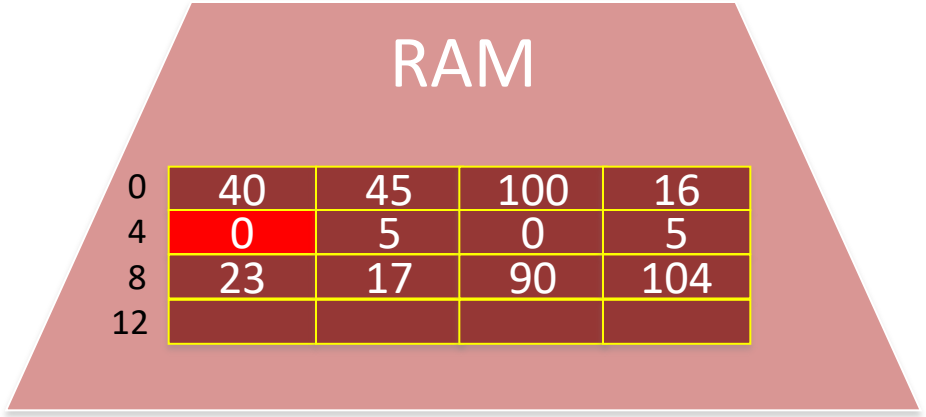


5. What Happens if Cache is Full?

1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   identify block to replace.
   LRU strategy chooses
   least recently used block.

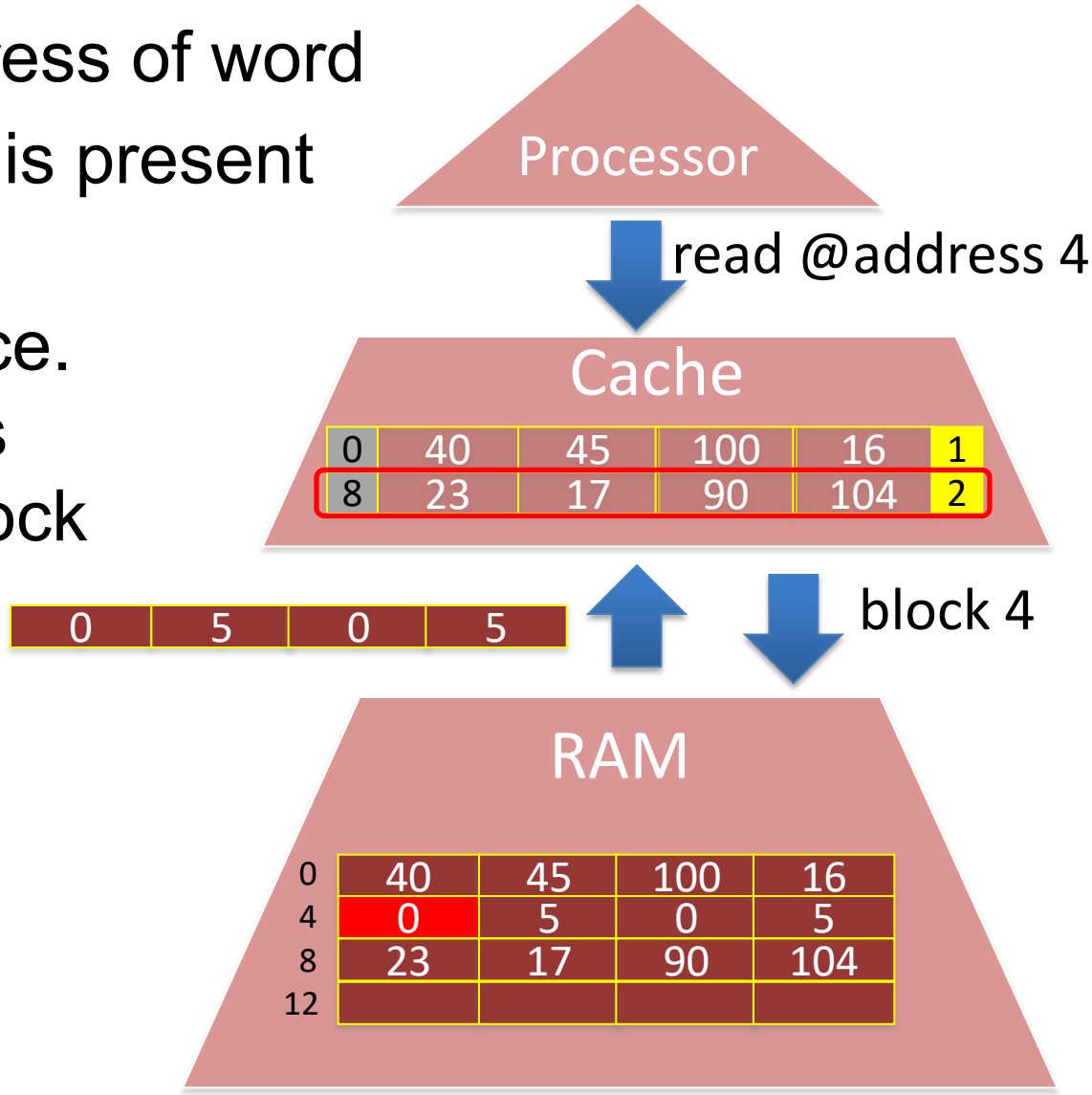


Continue as usual…



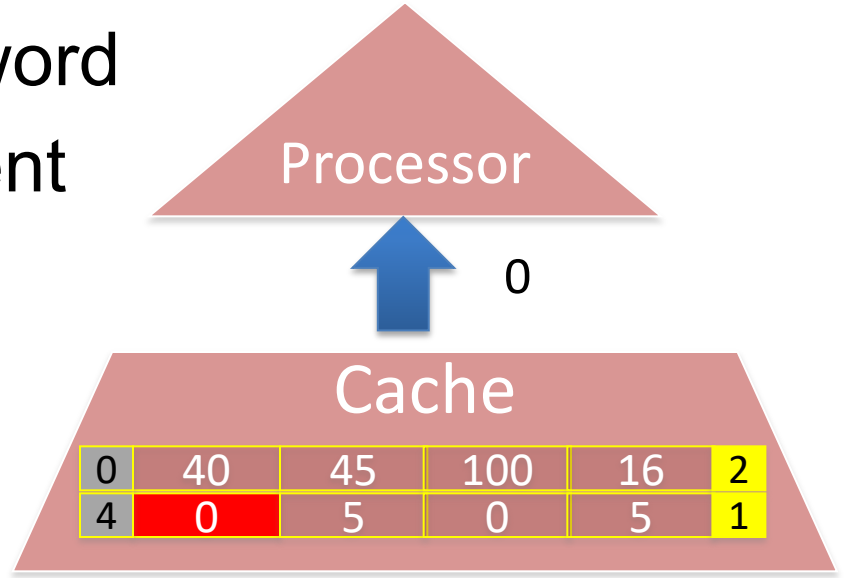
5. What Happens if Cache is Full?

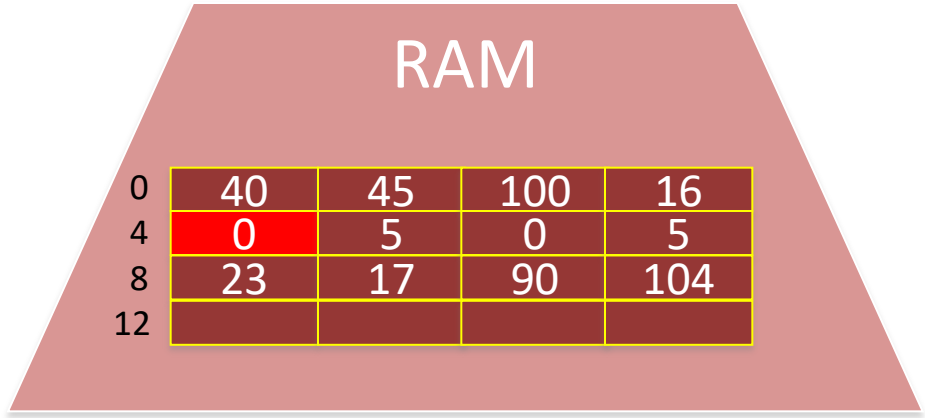
1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   identify block to replace.
   LRU strategy chooses
   least recently used block
4. Cache requests block
5. RAM returns block



5. What Happens if Cache is Full?

1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   identify block to replace.
   LRU strategy chooses
   least recently used block
4. Cache loads block
5. RAM returns block
6. Cache updates block and
   counters and sends word



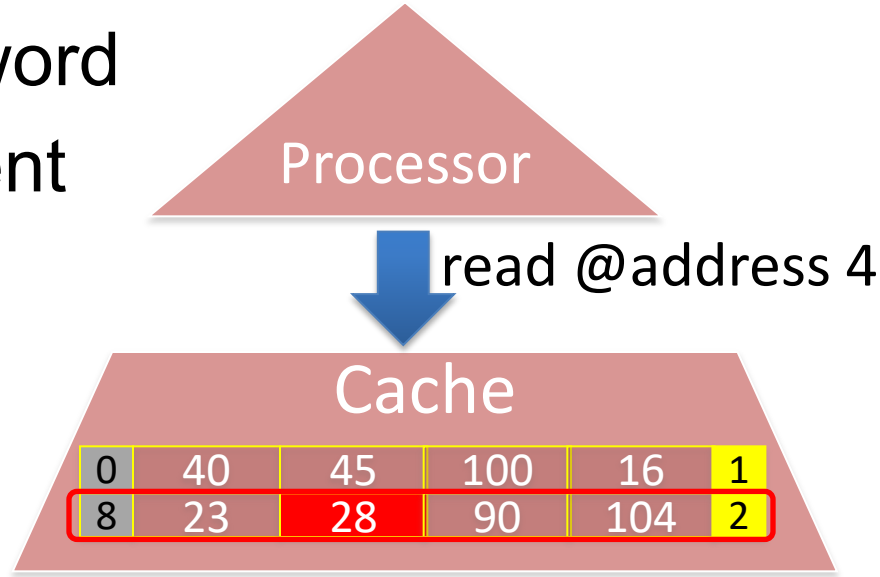


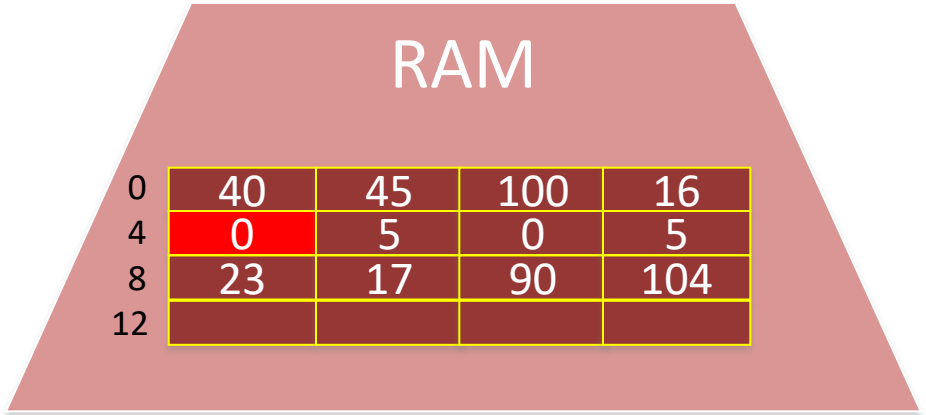
6. What Happens if Block was Modified?

- We have to send the content to the RAM before replacing the block

6. What Happens if Block was Modified?

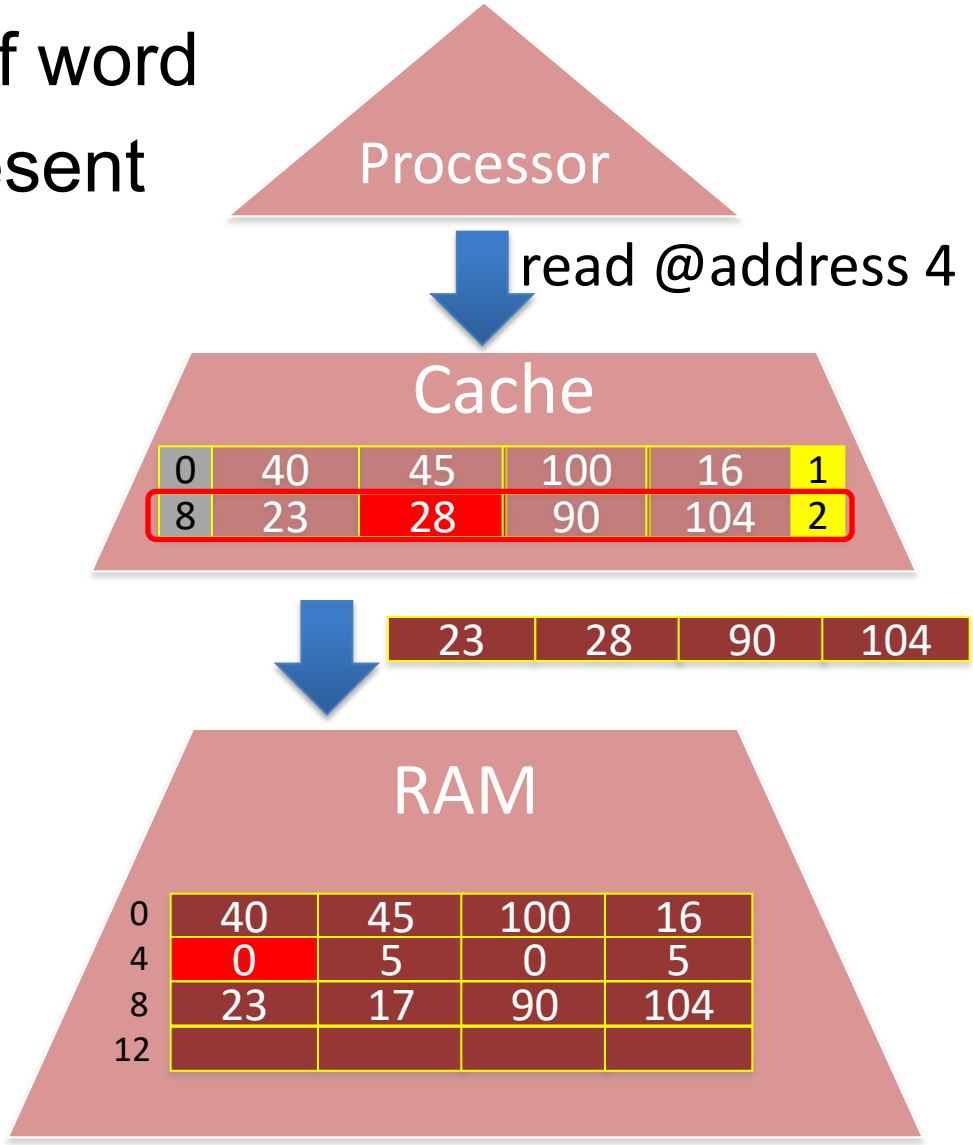
1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   identify block to replace.
   LRU strategy chooses
   least recently used block.





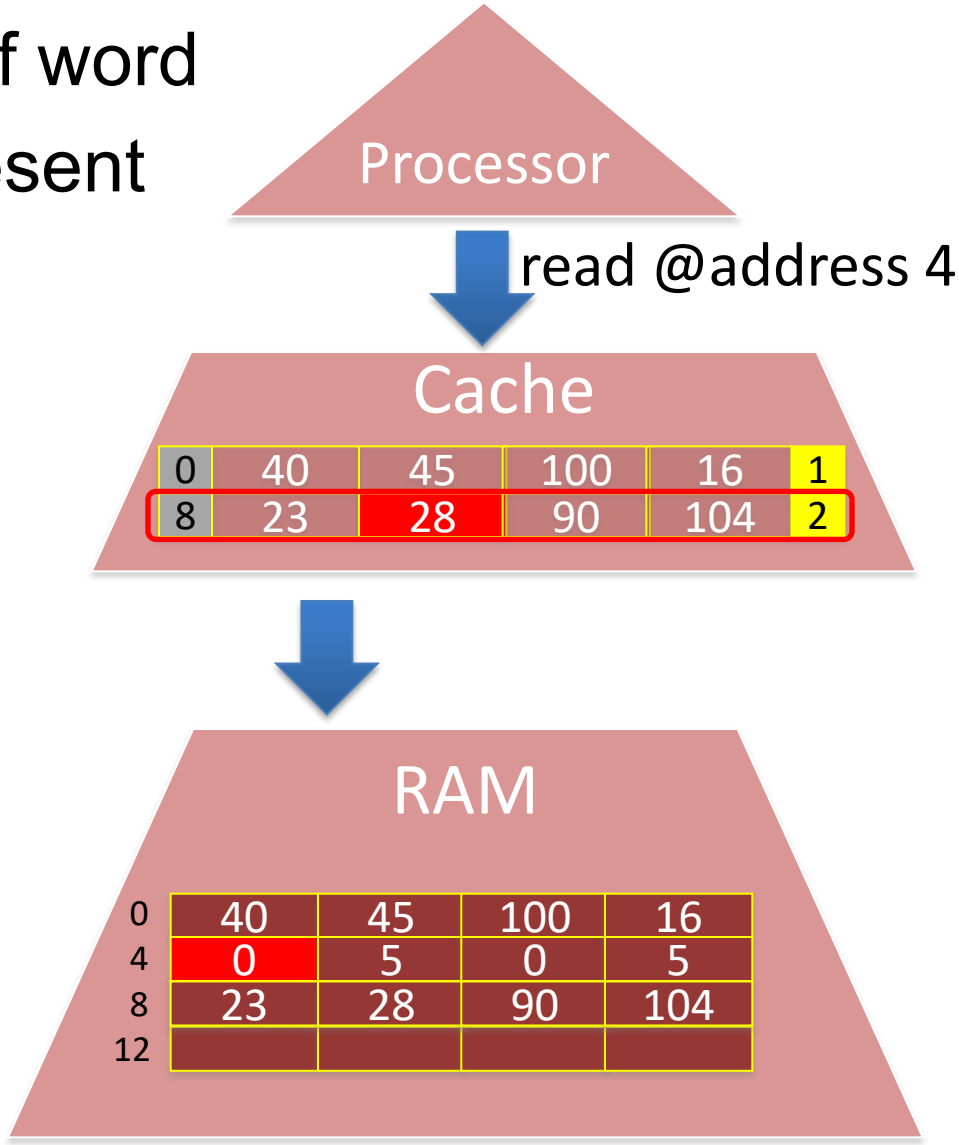
6. What Happens if Block was Modified?

1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   identify block to replace.
   LRU strategy chooses
   least recently used block.
4. Write block to RAM



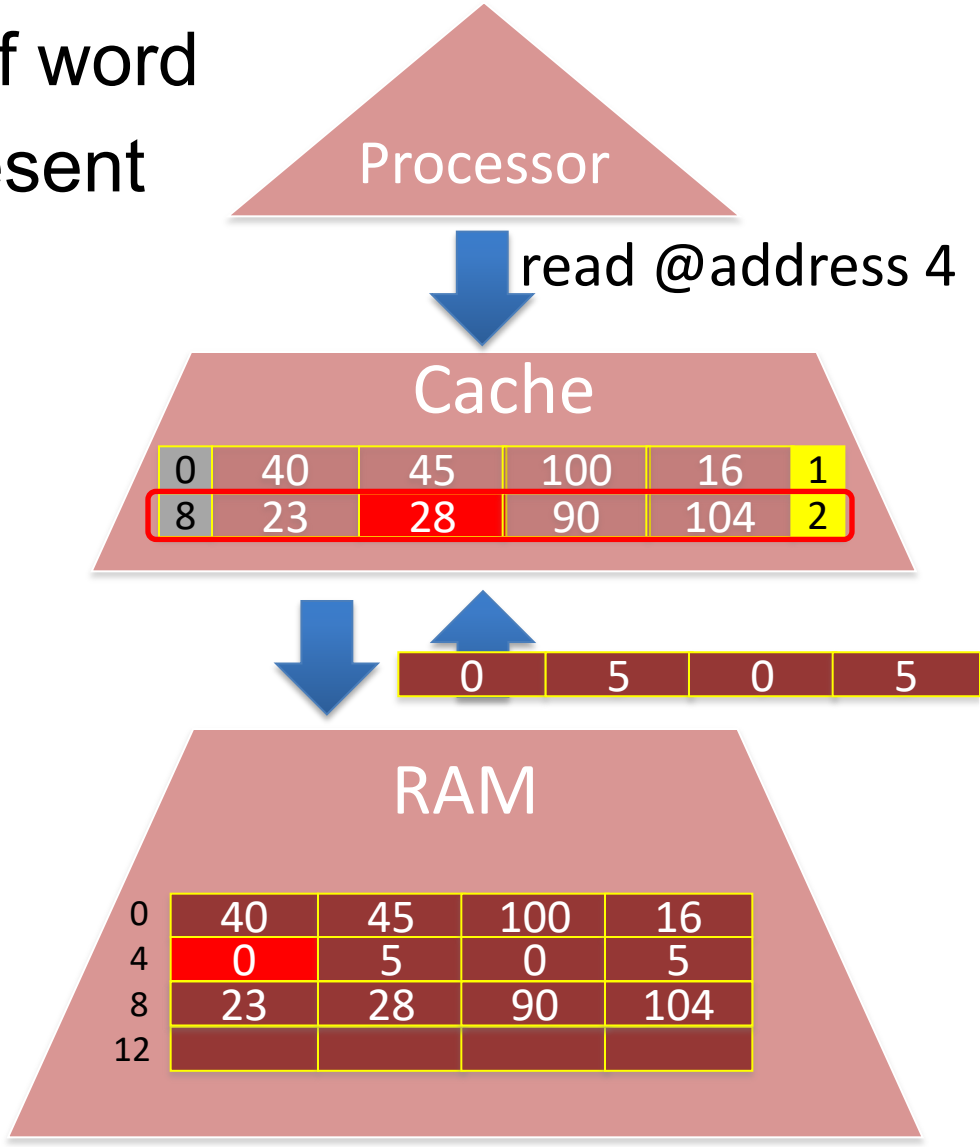
6. What Happens if Block was Modified?

1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   identify block to replace.
   LRU strategy chooses
   least recently used block.
4. Write block to RAM



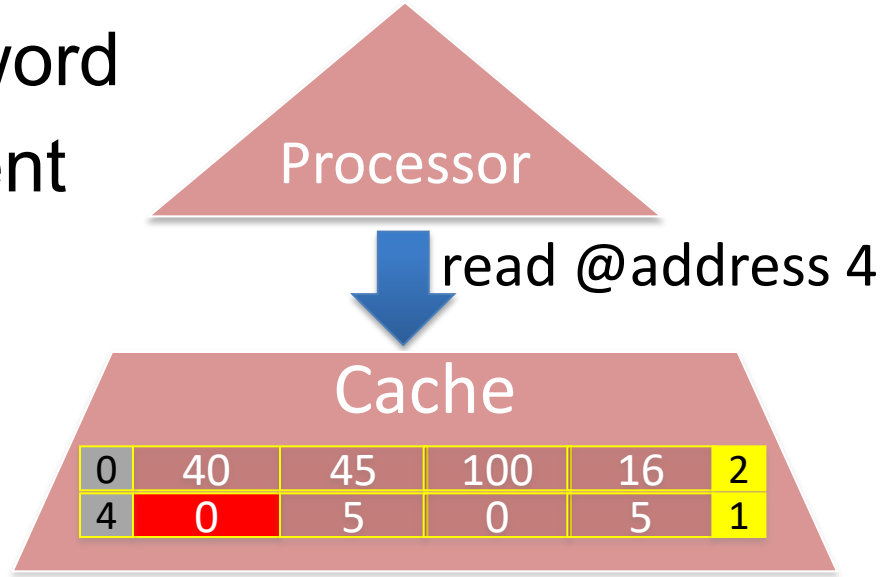
6. What Happens if Block was Modified?

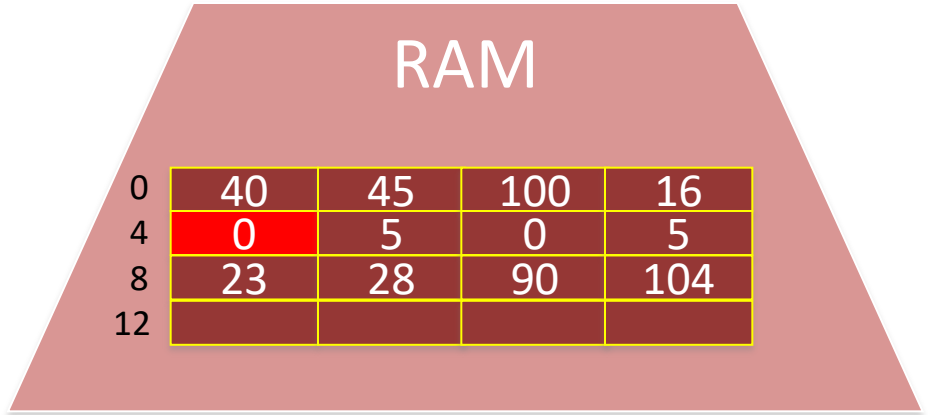
1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   identify block to replace.
   LRU strategy chooses
   least recently used block.
4. Write block to RAM
5. Request block 4



6. What Happens if Block was Modified?

1. Processor sends address of word
2. Cache checks if word is present
3. If not, cache miss:
   identify block to replace.
   LRU strategy chooses
   least recently used block.
4. Write block to RAM
5. Request block 4





Part 5: Examples

Example of Last Lesson

Sum of *n* integers
input: n output: m

s ← 0
 	
while n > 0
  s ← s + n
  n ← n – 1
 	
m ← s

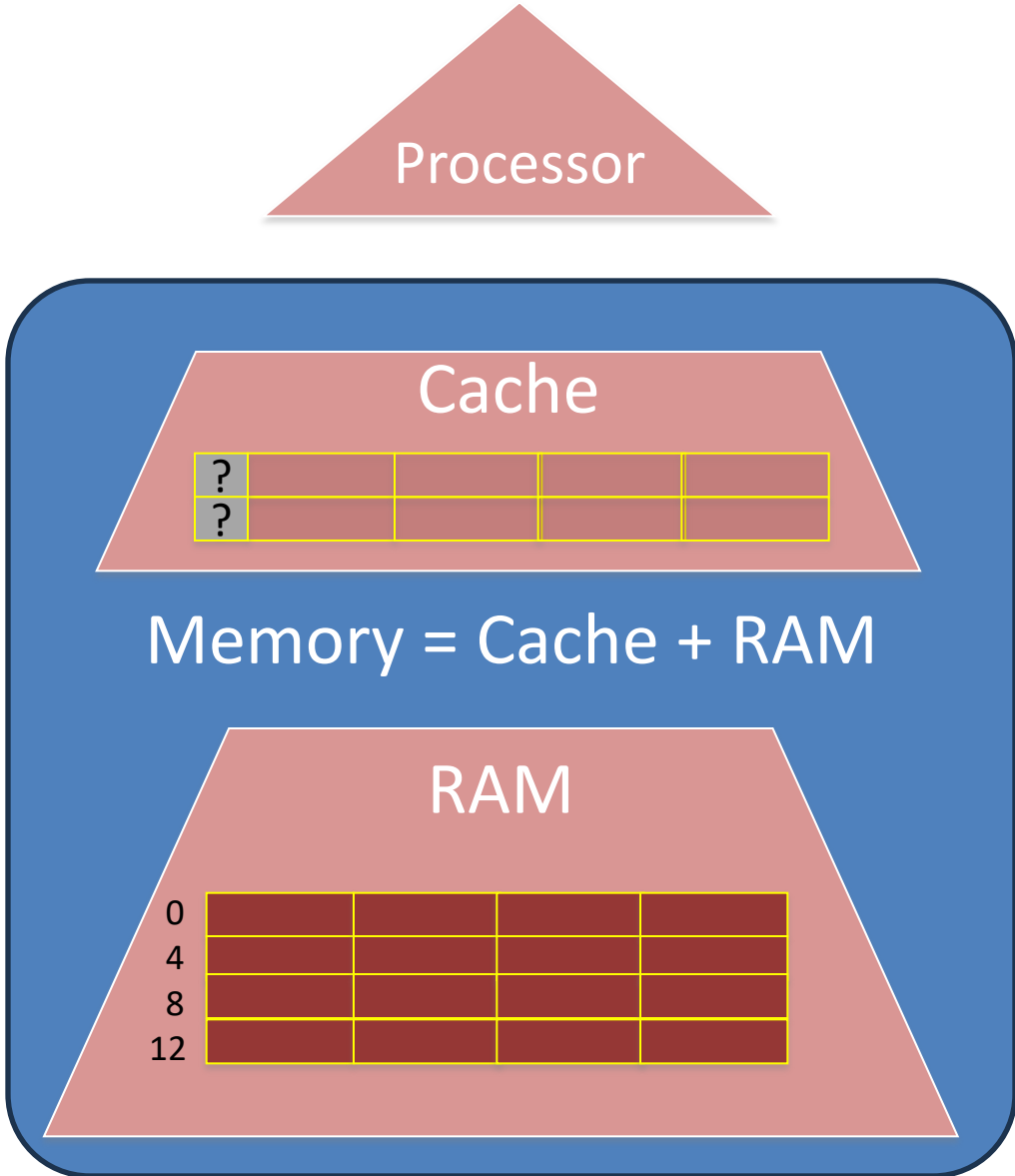
We will focus on the loop

Sum of *n* integers
input: n output: m

s ← 0
 	
while n > 0
  s ← s + n
  n ← n – 1
 	
m ← s

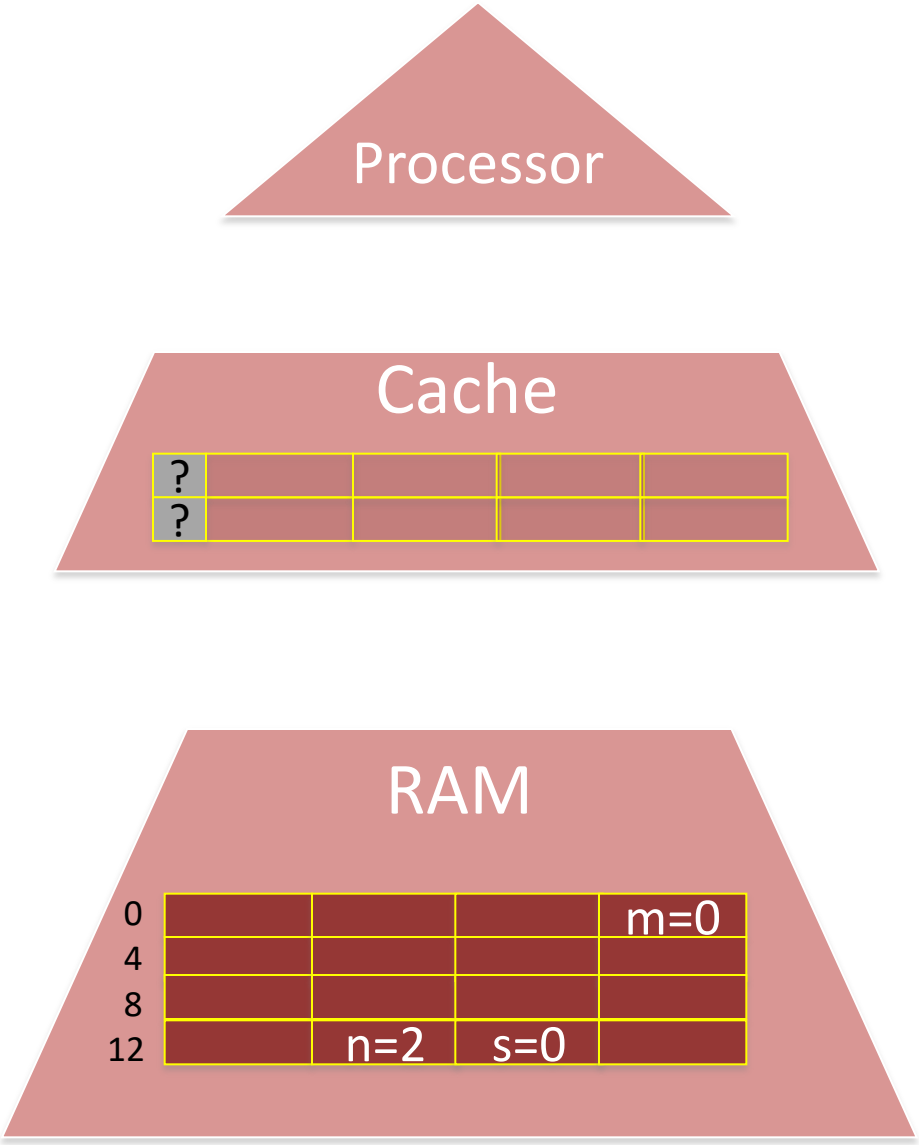
Memory Layout

- Cache with 2 blocks
- RAM with 4 blocks
- Blocks of 4 words
- Processor with 2 registers (no optimization, does not reuse content in registers)



Example Data Placement in RAM

- m : address 3
- n : address 13
- s : address 14
- s and m start with 0
- n starts with 2



Execution

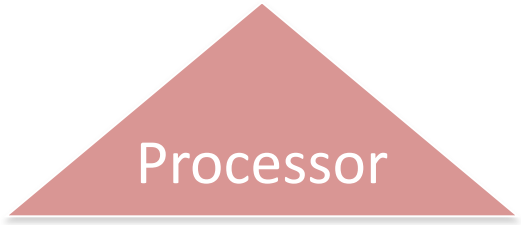
Sum of *n* integers

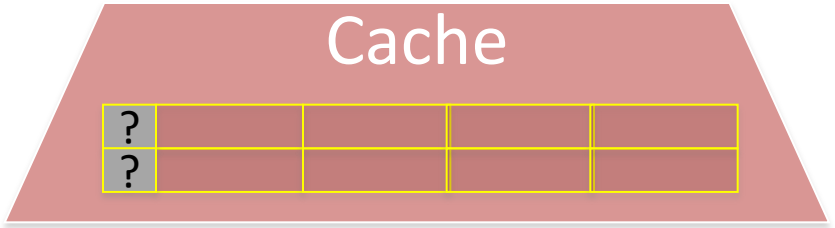
input: n
output: m

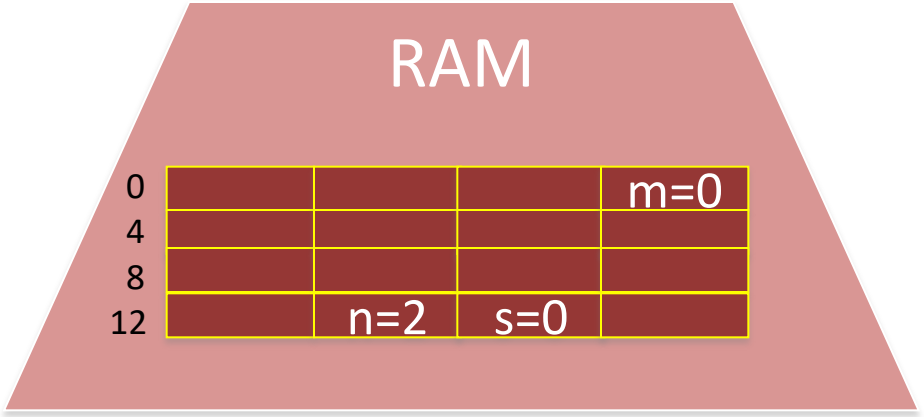
```
s ← 0

while n > 0
    s ← s + n
    n ← n – 1

m ← s
```







Execution

Sum of *n* integers

input: n

output: m

s ← 0

while n > 0

    s ← s + n

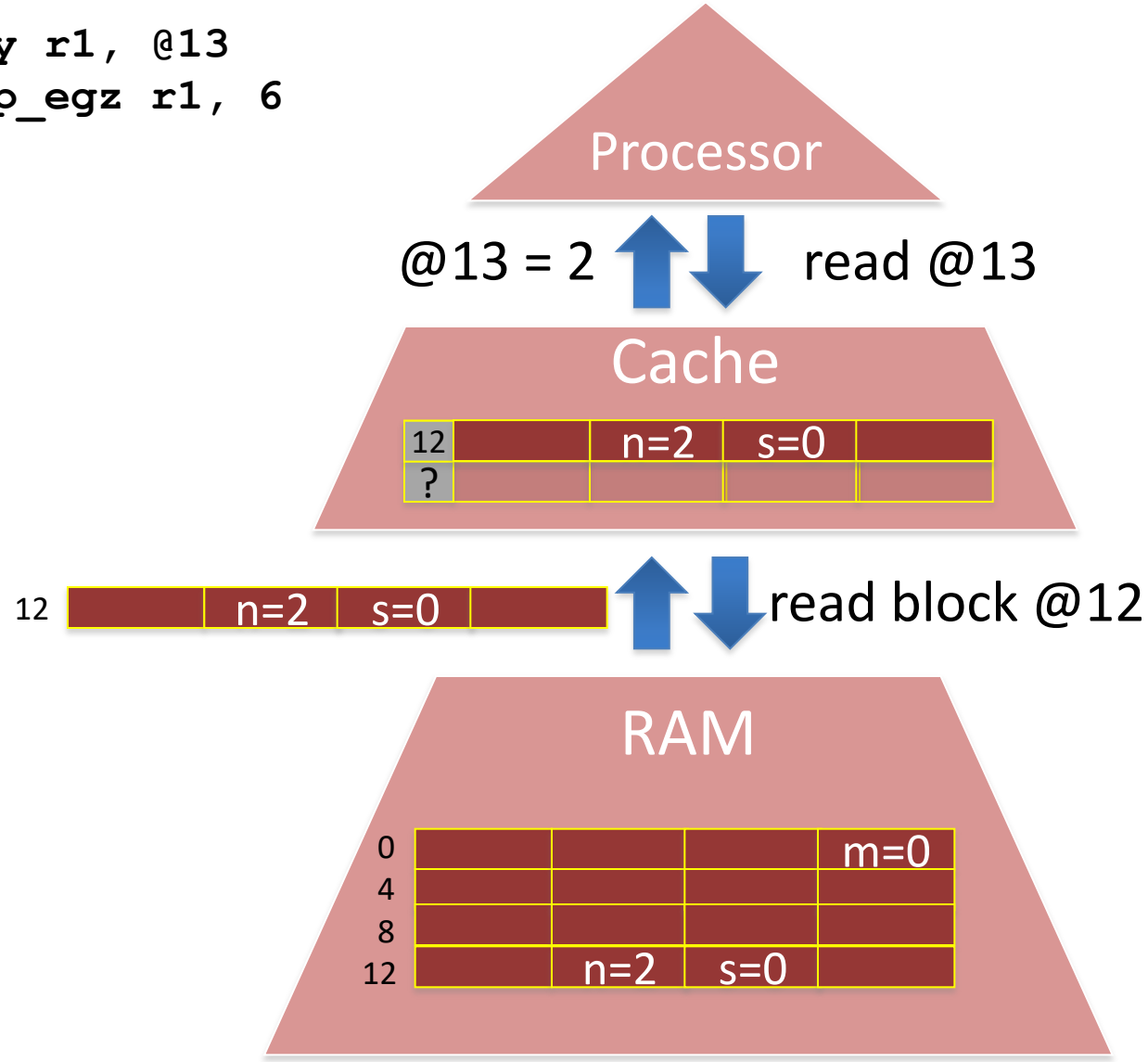
    n ← n – 1

m ← s

No of cache miss: ~~Ø~~ 1

No of memory access: ~~Ø~~ 1

```
copy r1, @13
jump_egz r1, 6
```



Execution

Sum of *n* integers

input: n

output: m

s ← 0

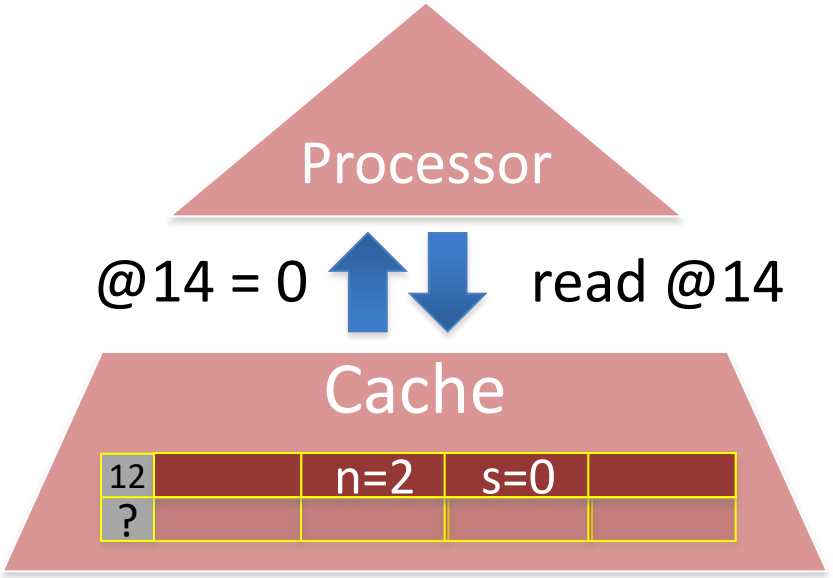
while n > 0

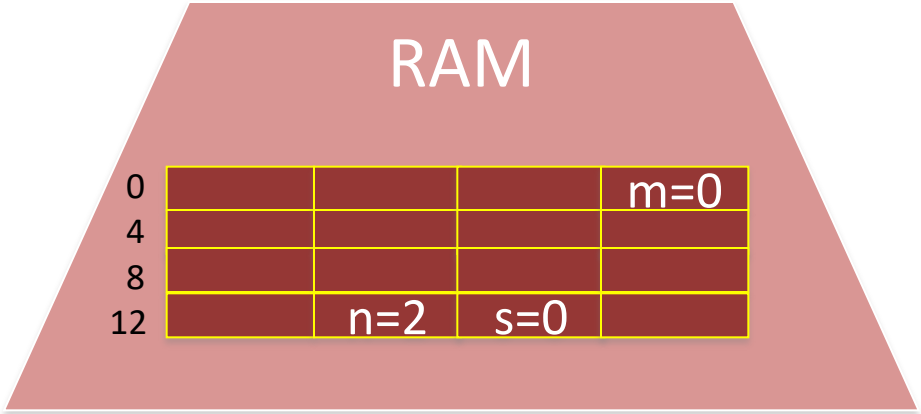
    s ← s + n

    n ← n – 1

m ← s

```
copy r1, @13
jump_egz r1, 6
copy r1, @14
```





No of cache miss: 1

No of memory access: ~~1~~ 2

Execution

Sum of *n* integers

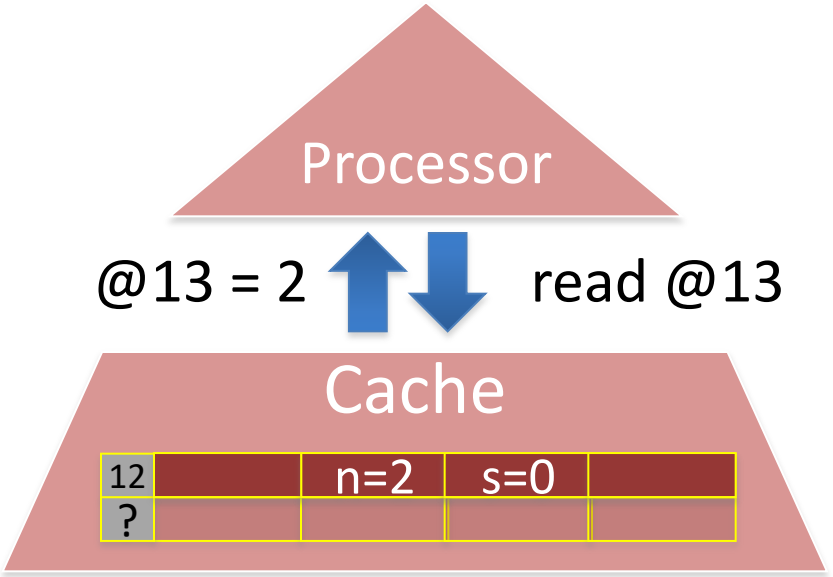
input: n
output: m

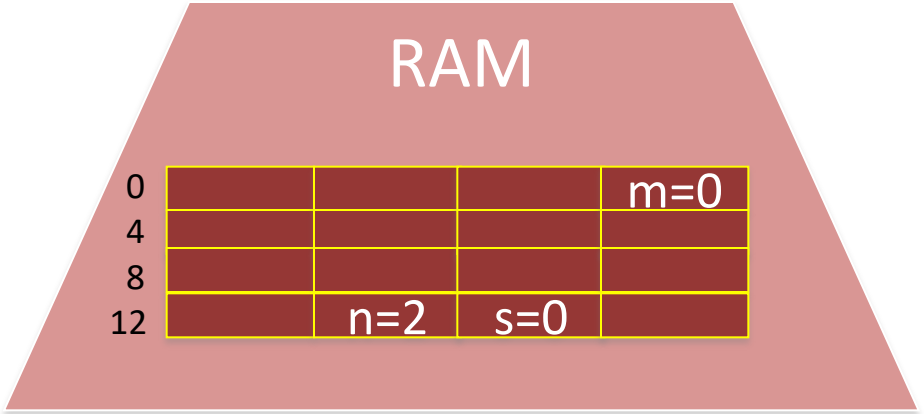
s ← 0

while n > 0
    s ← s + n
    n ← n – 1

m ← s

```
copy r1, @13
jump_egz r1, 6
copy r1, @14
copy r2, @13
```





No of cache miss: 1
No of memory access: ~~2~~ 3

Execution

Sum of *n* integers

input: n

output: m

s ← 0

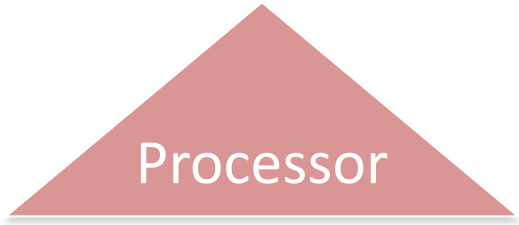
while n > 0

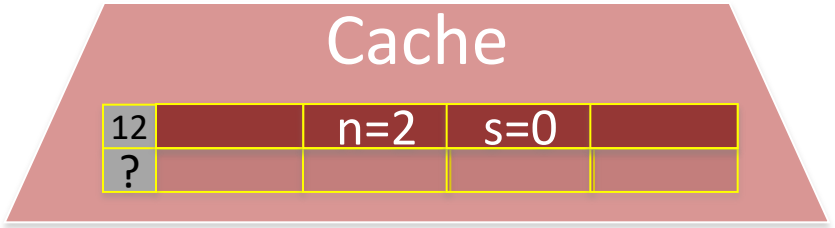
    s ← s + n

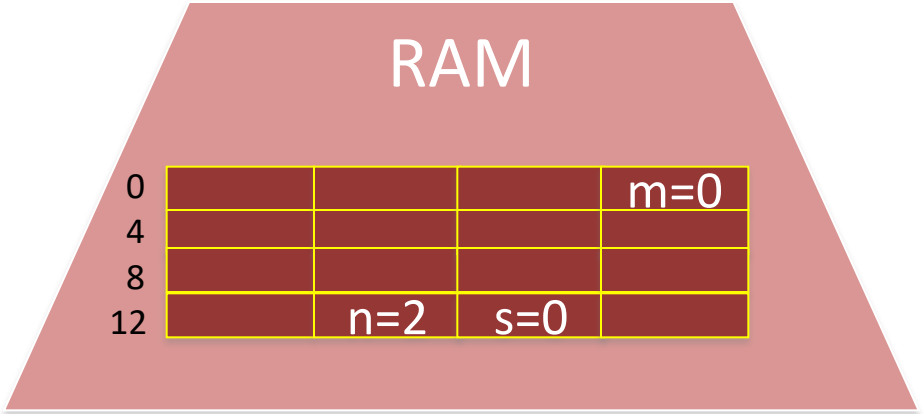
    n ← n – 1

m ← s

```
copy r1, @13
jump_egz r1, 6
copy r1, @14
copy r2, @13
add r1, r1, r2
```







No of cache miss: 1

No of memory access: 3

Execution

Sum of *n* integers

input: n

output: m

s ← 0

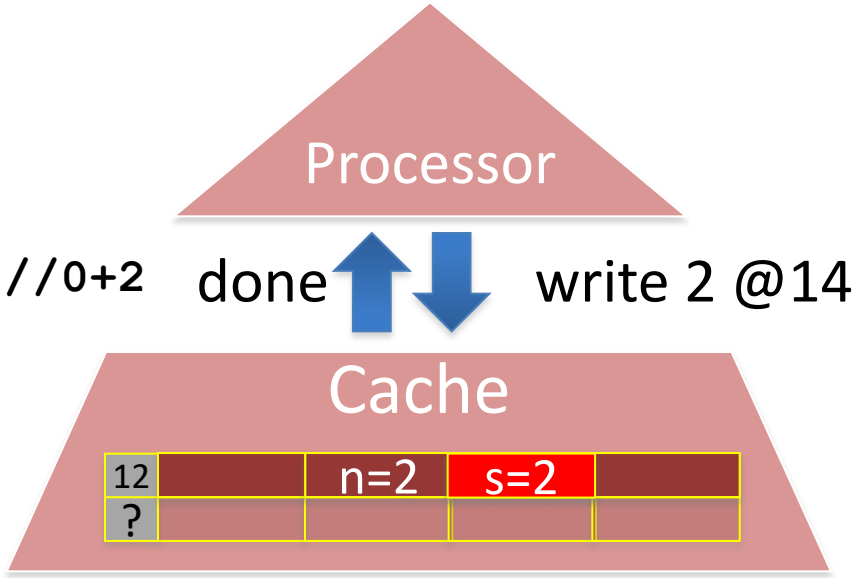
while n > 0

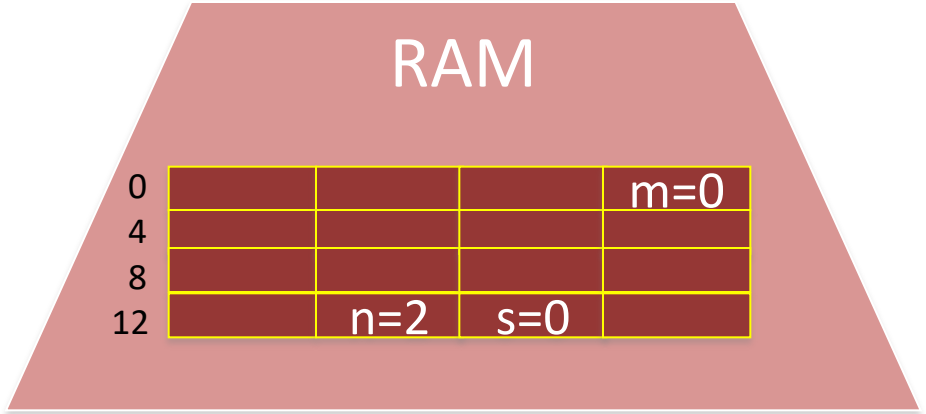
s ← s + n

n ← n – 1

m ← s

```
copy r1, @13
jump_egz r1, 6
copy r1, @14
copy r2, @13
add r1, r1, r2 //0+2
copy @14, r1
```





No of cache miss: 1

No of memory access: ~~3~~ 4

Execution

Sum of *n* integers

input: n

output: m

s ← 0

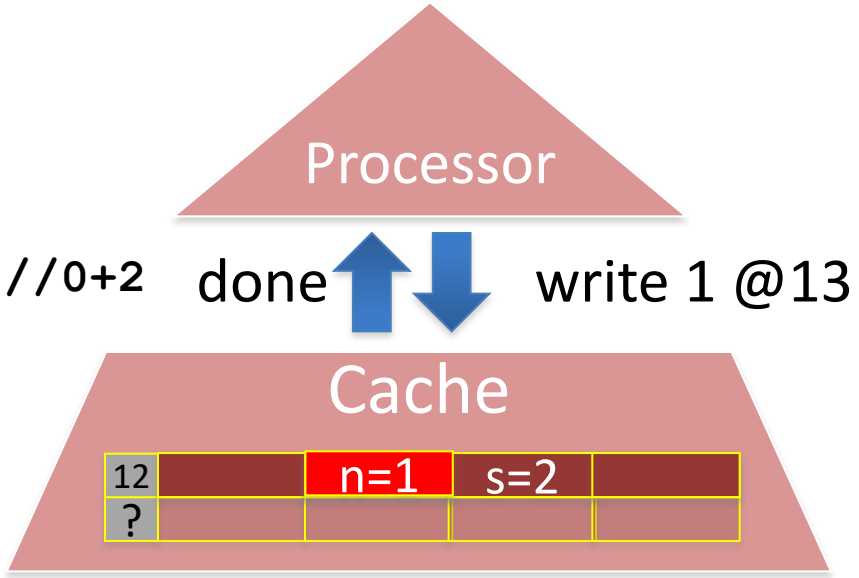
while n > 0

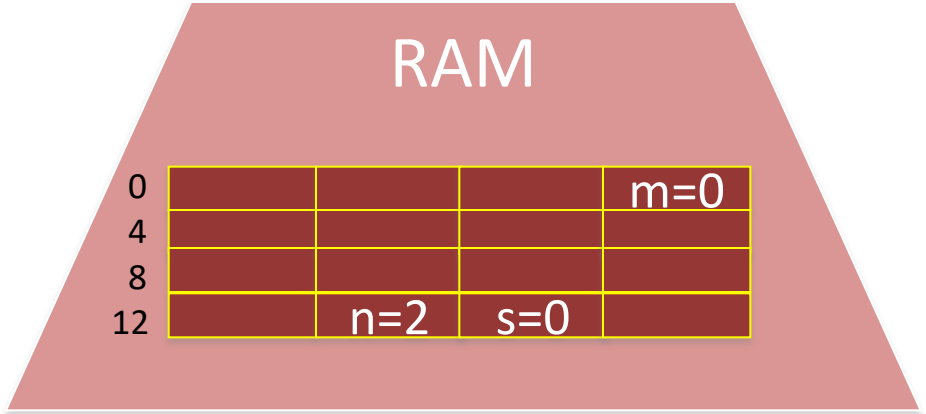
  s ← s + n

  n ← n – 1

m ← s

```
copy r1, @13
jump_egz r1, 6
copy r1, @14
copy r2, @13
add r1, r1, r2 //0+2
copy @14, r1
copy r1, @13
add r1, r1, -1
copy @13, r1
```





No of cache miss: 1

No of memory access: ~~4~~ ~~5~~ 6

Memory Accesses

- First loop iteration:
  - 6 memory accesses
  - 5 cache hits (request address was in cache)
  - 1 cache miss
- All subsequent iterations:
  - 6 memory accesses
  - 6 cache hits
  - 0 cache miss

Memory Accesses for this Program

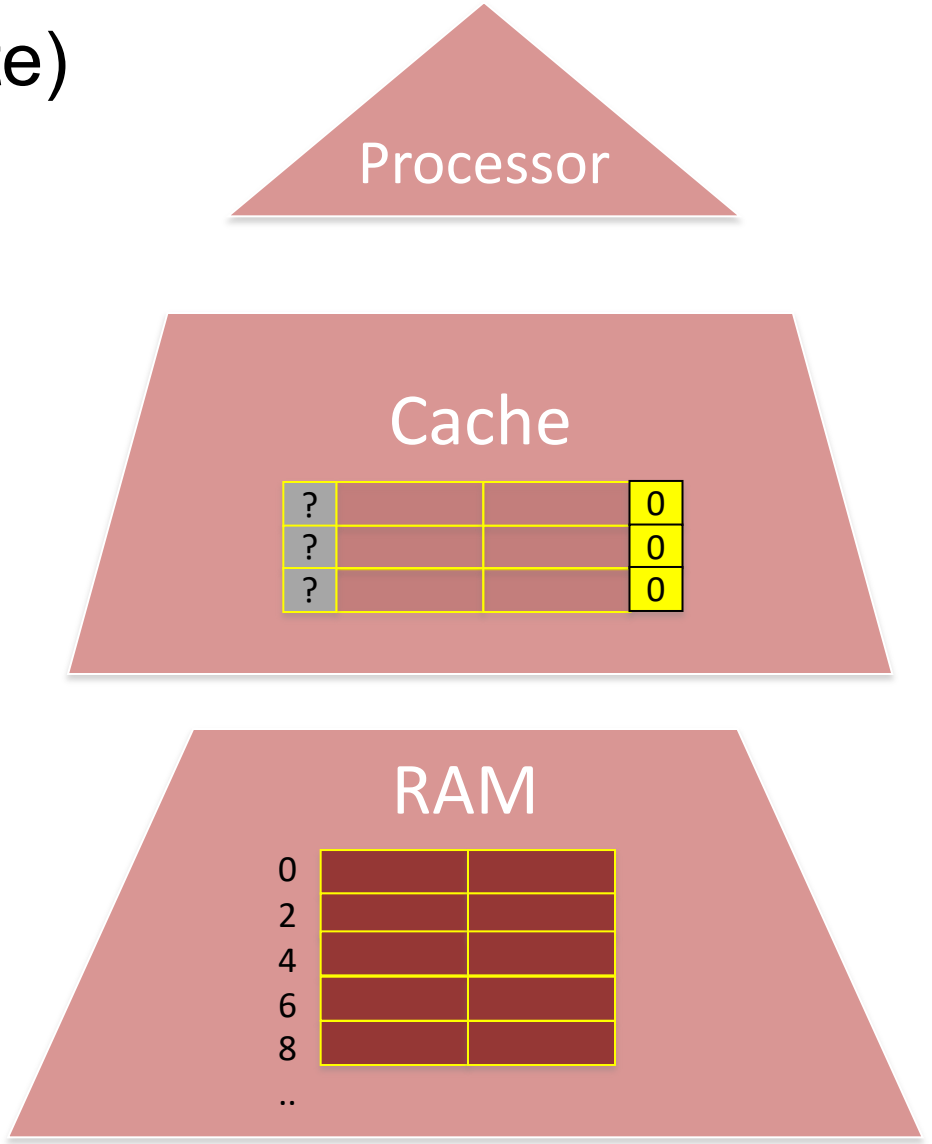
- Memory accesses: 6n (1ns/access)
- Cache miss: 1 access to the main memory (100ns)

- Total time with cache: 6n + 100 = 6n + 100 ns
- Total time without cache: 600n ns

Loop Iterations	Time with Cache	Time w/o Cache
10	160ns	6’000ns
100	700ns	60’000ns
1’000	7’100ns	600’000ns

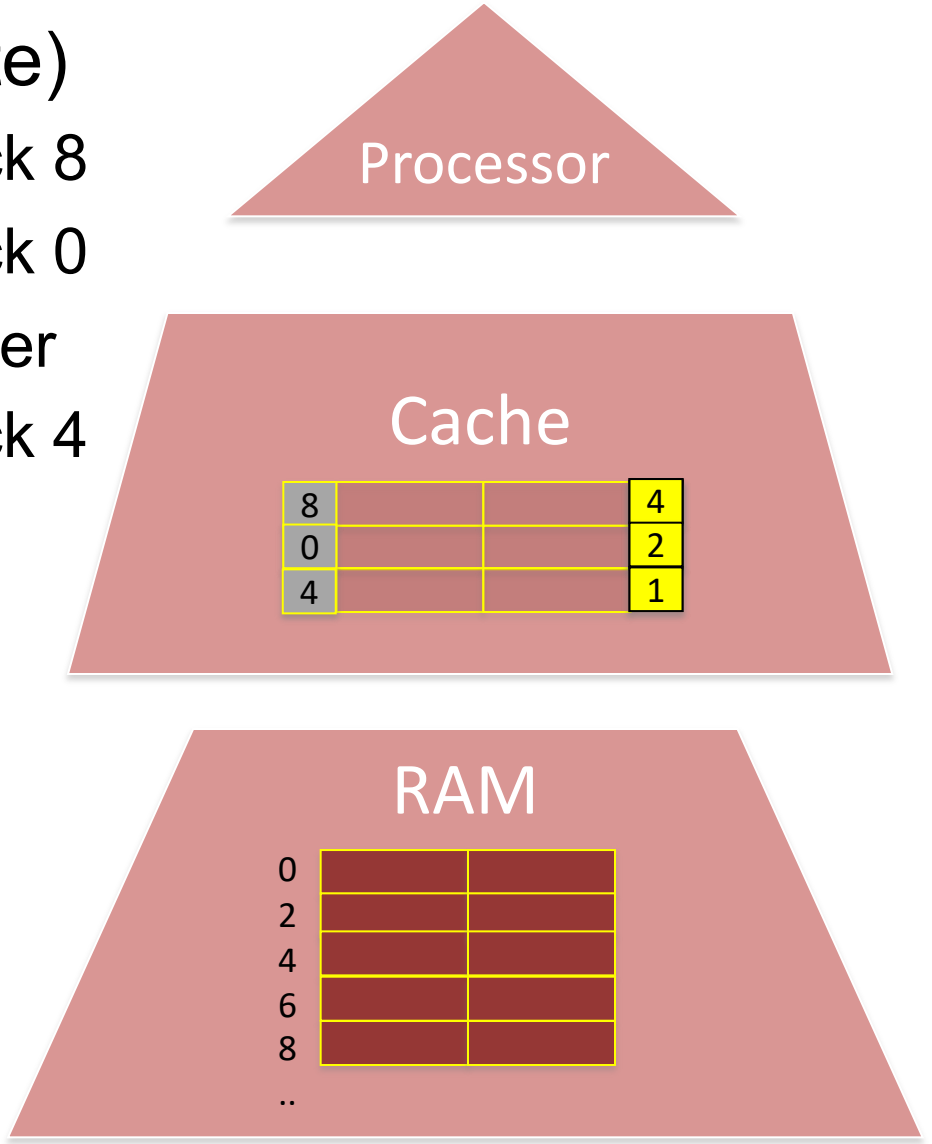
Another Example 1/5

- Memory accesses (read/write)
  - @ address 9
  - @ address 0
  - @ address 1
  - @ address 4
  - @ address 5
  - @ address 8
  - @ address 6
  - @ address 9
  - @ address 0



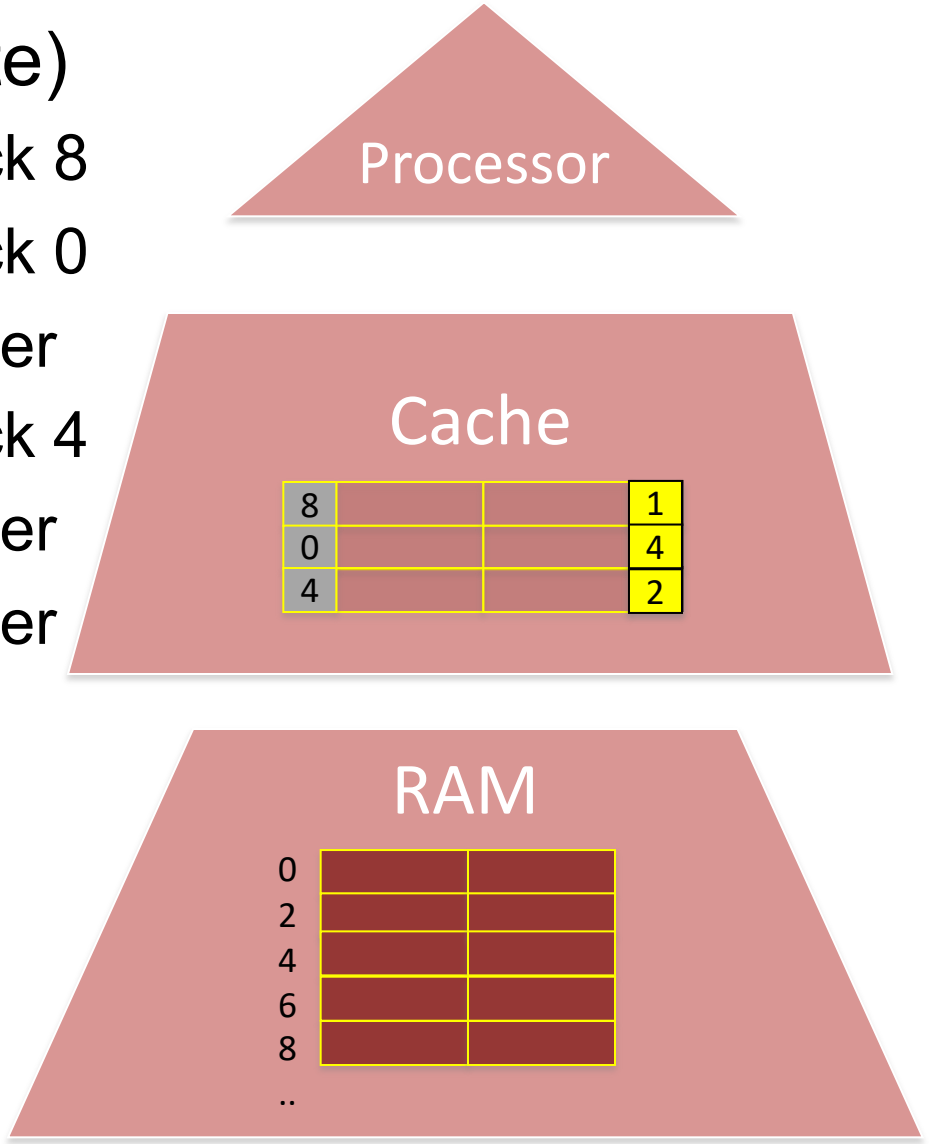
Another Example 2/5

- Memory accesses (read/write)
  - @ address 9 – miss, load block 8
  - @ address 0 – miss, load block 0
  - @ address 1 – hit, reset counter
  - @ address 4 – miss, load block 4
  - @ address 5
  - @ address 8
  - @ address 6
  - @ address 9
  - @ address 0



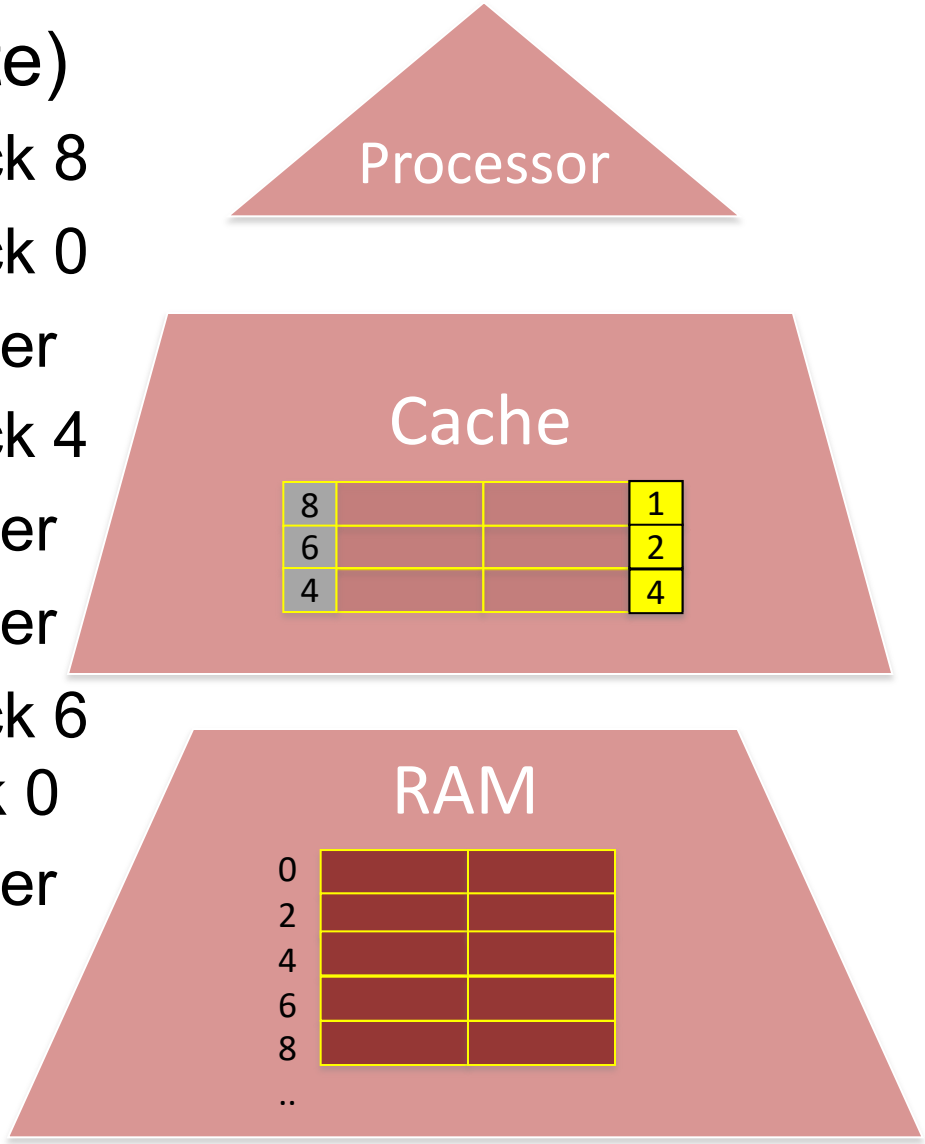
Another Example 3/5

- Memory accesses (read/write)
  - @ address 9 – miss, load block 8
  - @ address 0 – miss, load block 0
  - @ address 1 – hit, reset counter
  - @ address 4 – miss, load block 4
  - @ address 5 – hit, reset counter
  - @ address 8 – hit, reset counter
  - @ address 6
  - @ address 9
  - @ address 0



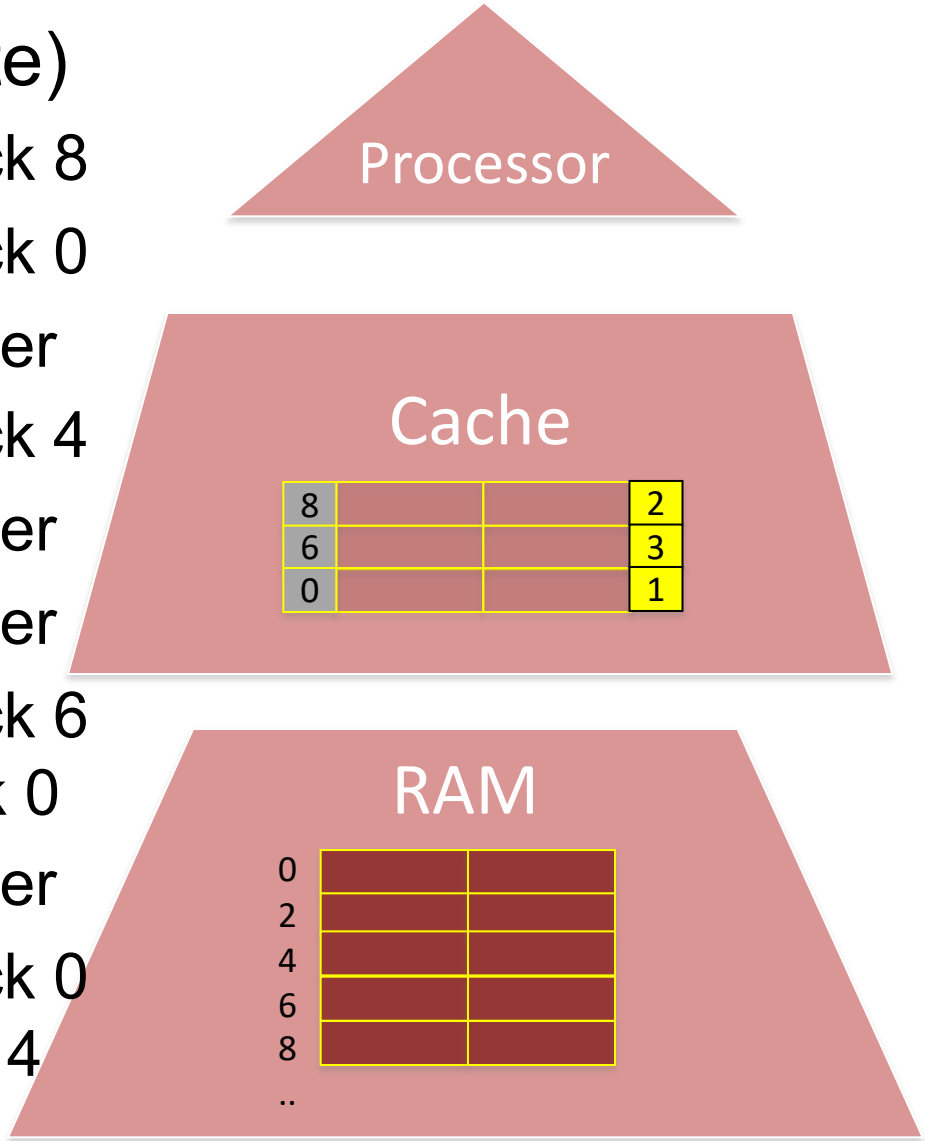
Another Example 4/5

- Memory accesses (read/write)
  - @ address 9 – miss, load block 8
  - @ address 0 – miss, load block 0
  - @ address 1 – hit, reset counter
  - @ address 4 – miss, load block 4
  - @ address 5 – hit, reset counter
  - @ address 8 – hit, reset counter
  - @ address 6 – miss, load block 6 overwrite block 0
  - @ address 9 – hit, reset counter
  - @ address 0



Another Example 5/5

- Memory accesses (read/write)
  - @ address 9 – miss, load block 8
  - @ address 0 – miss, load block 0
  - @ address 1 – hit, reset counter
  - @ address 4 – miss, load block 4
  - @ address 5 – hit, reset counter
  - @ address 8 – hit, reset counter
  - @ address 6 – miss, load block 6 overwrite block 0
  - @ address 9 – hit, reset counter
  - @ address 0 – miss, load block 0 overwrite block 4



Part 6: Locality Principle

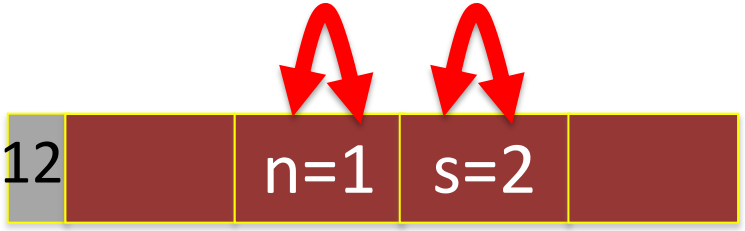
Why is the Cache useful?

There are two cases in which a cache is useful:

1. **Temporal Locality:** multiple accesses to the same address within a short time span, e.g., variable n was access multiple times in our previous example.
2. **Spatial Locality:** accesses to variables in the same block, e.g., n and s were it the same block in our example

Temporal Locality

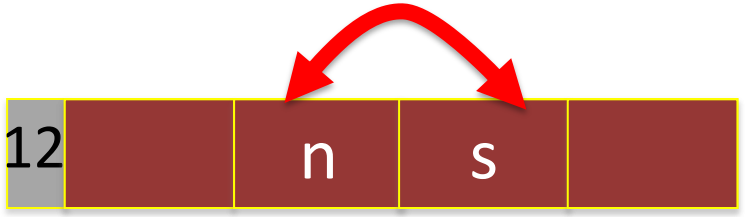
- In cache because there has been an access to the same address



- Is that realistic?
- All “interesting” algorithms have loops that access the same variables.

Spatial Locality

- In cache because there has been an access to an address in the same block



- Is that realistic?
- All “interesting” algorithms access variables that are linked (e.g., arrays). The compiler has to ensure that these variables will be close in memory.

Example of Cache Influence

- Addition of elements in a matrix
- Order of elements in the sum does not matter for result
- But order might matter for performance

M(0,0)	M(0,1)	M(0,2)	M(0,3)
M(1,0)	M(1,1)	M(1,2)	M(1,3)
M(2,0)	M(2,1)	M(2,2)	M(2,3)
M(3,0)	M(3,1)	M(3,2)	M(3,3)

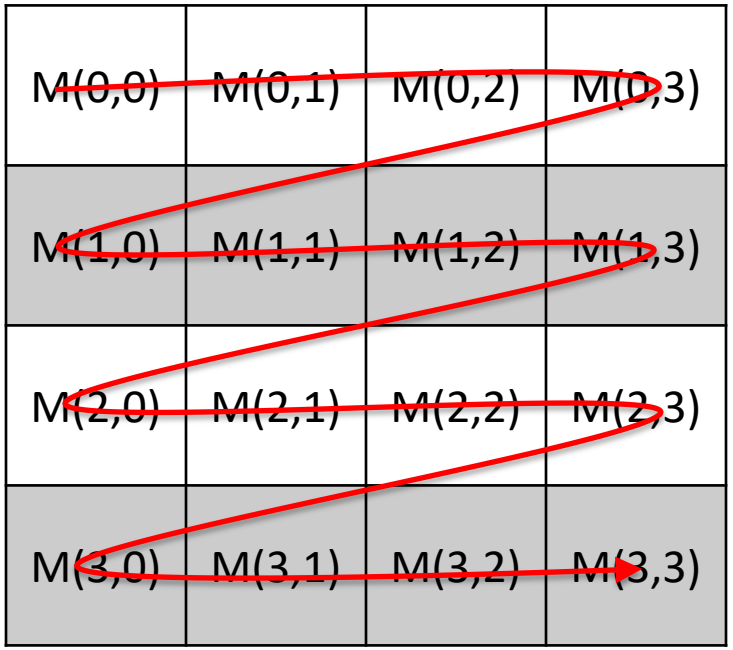
Addition per Line

MatrixSumPerLines

input: matrix M 4x4
output: sum of elems

```
s ← 0

for r from 0 to 3
    for c from 0 to 3
        s ← s + M(r,c)
return s
```

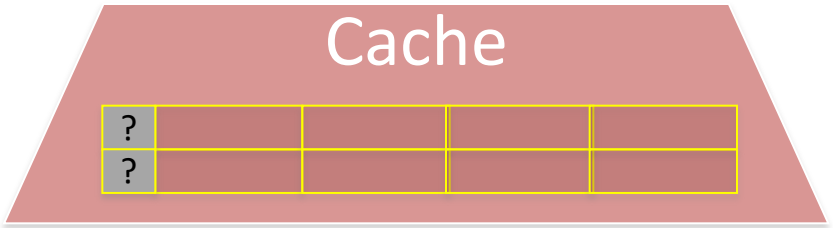


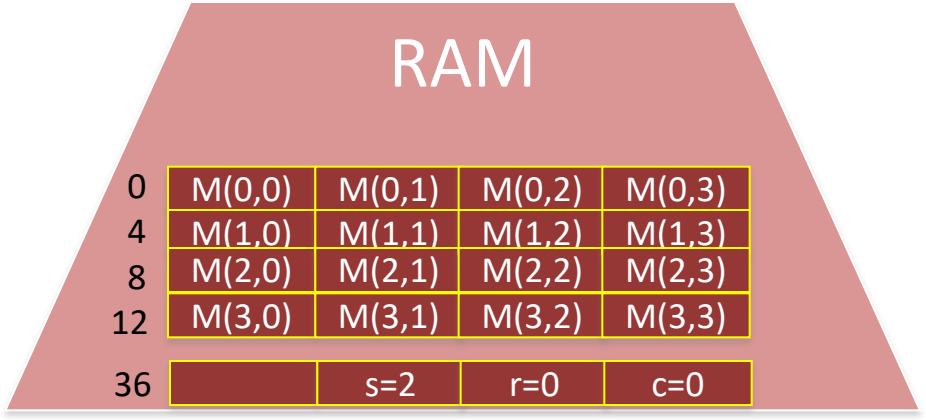
Addition per Column

MatrixSumPerCols
input: matrix M 4x4 output: sum of elems

s ← 0
 	
for c from 0 to 3
  for r from 0 to 3
  s ← s + M(r,c)
return s

Example Data Placement in Memory



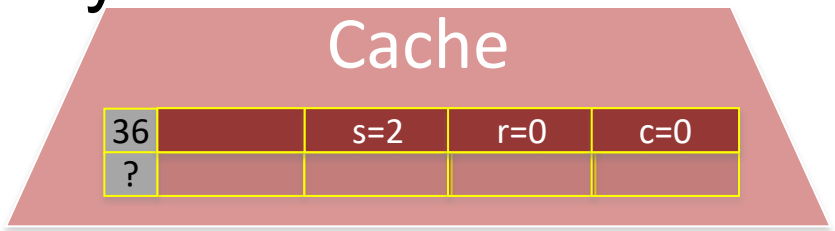


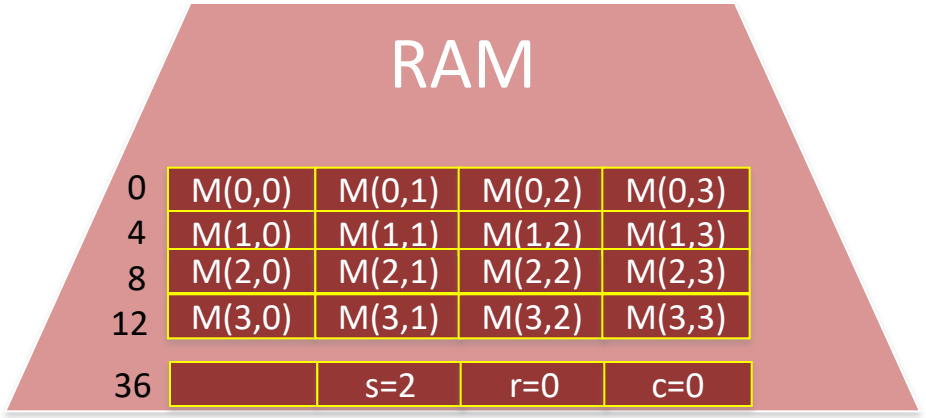
Exercise

- How many cache misses in ”addition per line”?
- How many cache misses in “addition per column”?
- Which one is more efficient?
  (assuming accessing the cache takes 1ns and accessing the main memory takes 100ns)

Analysis

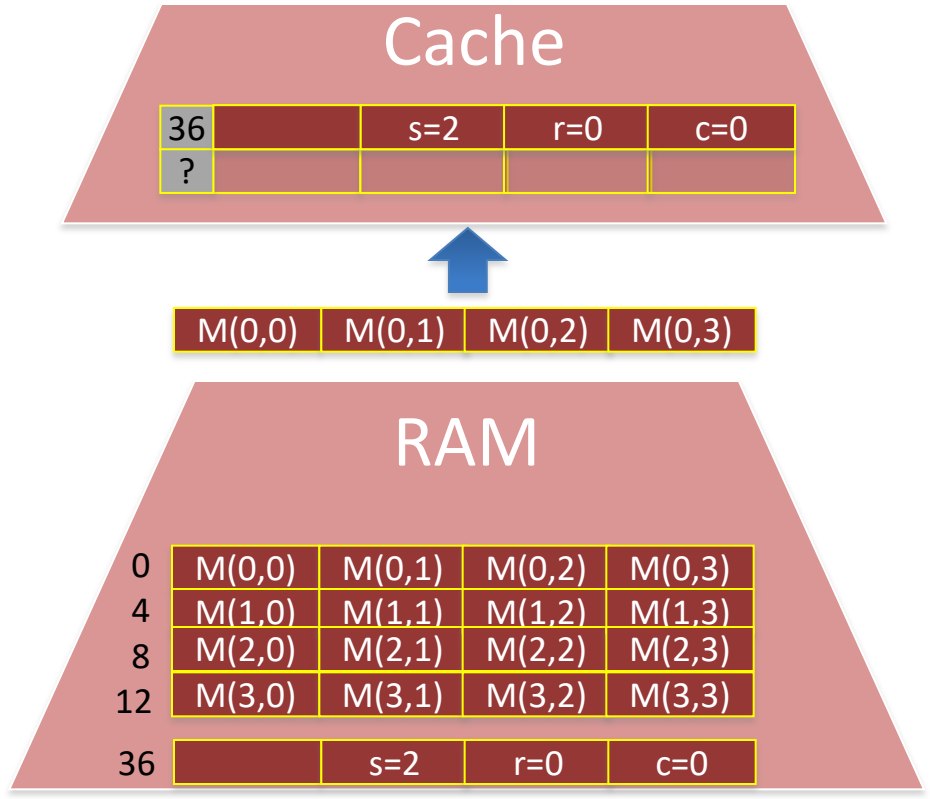
- Once s, r, and c are in cache, they will stay there due to temporal locality





Analysis

- 1 cache miss per line



Memory Access – Addition per Line

MatrixSumPerLines
input: matrix M 4x4 output: sum of elems

s ← 0
 	
for r from 0 to 3
  for c from 0 to 3
  s ← s + M(r,c)
return s

Access to cache: 1ns
Access to main memory: 100ns

- One iteration of the outer loop:
  - 1 cache miss to load the line
  - 22 accesses:
    - read and write access to r
    - 4 x read and write access to s
    - 4 x read access to M(r,c)
    - 4 x read and write access to c
- Total (4 iterations of outer loop):
  - 5 cache misses (1 initial + 4 lines)
  - 4 x 22 accesses
  - 4 x 22 + 5 x 100 ≈ **600ns**

Memory Access – Addition per Column

MatrixSumPerCols
input: matrix M 4x4 output: sum of elems

s ← 0
 	
for c from 0 to 3
  for r from 0 to 3
  s ← s + M(r,c)
return s

- One iteration of the outer loop:
  - 4 cache misses (one per line)
  - 22 accesses:
    - read and write access to r
    - 4 x read and write access to s
    - 4 x read access to M(r,c)
    - 4 x read and write access to c
- Total:
  - 17 cache misses (1 initial + 4x4 lines)
  - 4 x 22 accesses
  - 4 x 22 + 17 x 100 ≈ **1800ns**

About 3 times slower!

Try it Yourself!

```cpp
#include <vector>
using namespace std;

int main() {
    int size = 10000;
    vector<vector<double> > tab(size, vector<double>(size, 0));

    double sum = 0.0;

    for (int r = 0; r < size; r++) {
        for (int c = 0; c < size; c++) {
            sum = sum + tab[r][c];
        }
    }
    return 0;
}
```

```cpp
#include <vector>
using namespace std;

int main() {
    int size = 10000;
    vector<vector<double> > tab(size, vector<double>(size, 0));

    double sum = 0.0;

    for (int c = 0; c < size; c++) {
        for (int r = 0; r < size; r++) {
            sum = sum + tab[r][c];
        }
    }
    return 0;
}
```

```
gcc lines.c -o lines

time ./lines
cpu 1.818s
```

```
gcc columns.c -o columns

time ./columns
cpu 4.342s
```

About 2.4 times slower on my computer!

Summary

- Technology (register, RAM, Flash, Hard drives, tapes)
- Characteristics (latency, bandwidth, capacity, costs, volatile/non-volatile)
- Cache Principle
  - Keep data that you need close, so it can be accessed fast
  - LRU (Least Recently Used) Strategy
  - Works because of spatial and temporal locality principle
- When there is a good locality, there will be a lot of cache accessed and a good performance