CS 354 - Machine Organization & Programming
Tuesday Mar 21st and Thursday Mar 23rd,2023

Midterm Exam 2 - Thursday April 6th, 7:30 - 9:30 pm

- **UW ID required**
- **#2 pencils required**
- **closed book, no notes, no electronic devices (e.g., calculators, phones, watches) see "Midterm Exam 2" on course site Assignments for topics**

Homework hw4: DUE on or before Monday, Mar 27 7th

Project p3: DUE on or before Friday, Mar 24th

Project p4A: DUE on or before Friday, Mar 31st

Project p4B: DUE on or before Friday, April 7th

Last Week (before Spring Break)

Locality & Caching Bad Locality Caching: Basic Idea & Terms Designing a Cache: Blocks	Rethinking Addressing Designing a Cache: Sets and Tags TODO Basic Cache Lines TODO Basic Cache Operation Basic Cache Practice-practice quiz L14-12

This Week

Finish L14 (bring W7 outline) Direct Mapped Caches - Restrictive Fully Associative Caches - Unrestrictive Set Associative Caches - Sweet! Replacement Policies	Writing to Caches Cache Performance Impact of Stride Memory Mountain C, Assembly, and Mach Code
Next Week: Assembly Language Instructions and Operands B&O Chapter 3 Intro 3.1 A Historical Perspective 3.2 Program Encodings 3.3 Data Formats 3.4 Accessing Information 3.5 Arithmetic and Logical Control 3.6 Control	

Direct Mapped Caches - Restrictive

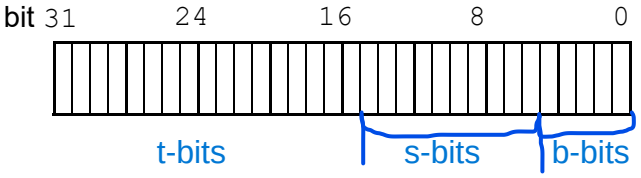
Direct Mapped Cache is a cache having S sets with 1 line per set

Each Block maps to one set

→ What is the address breakdown if blocks are 32 bytes and there are 1024 sets?

B = 32 bytes = 2^b => b = 5
S = 1024 bytes = 2 ^ s => s = 10

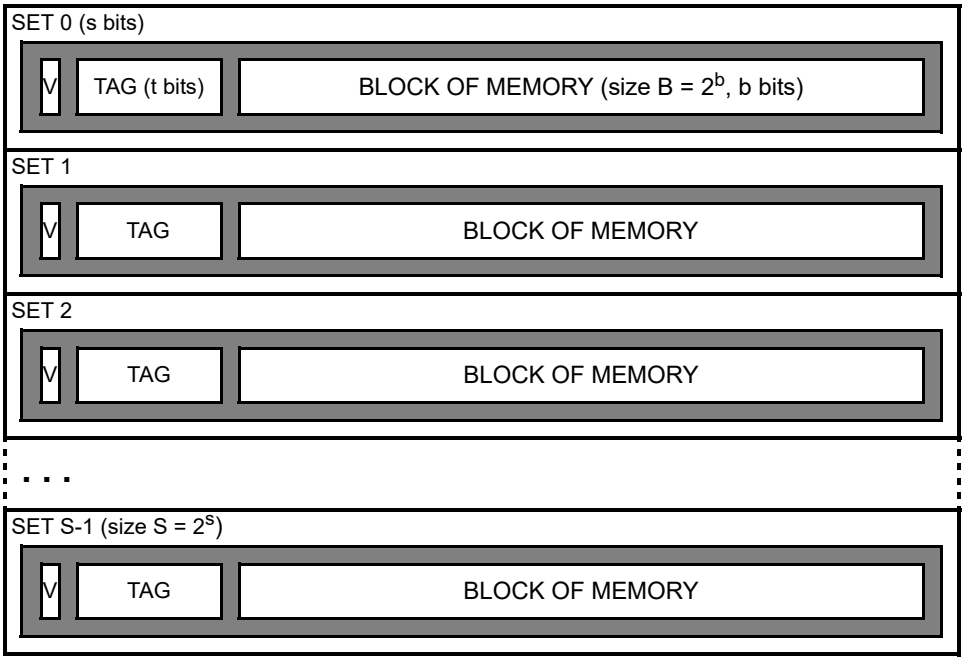
32-bit Address Breakdown



→ Is the cache operation fast O(1) or slow O(S) where S is the number of sets? FAST

No search O(1) set selections + O(1) line matching

+ simple circuitry to determine if tag matches t-bits



→ What happens when two different memory blocks map to the same set?

"conflict miss" - set stores 1 block if multiple blocks map to same set

"thrashing" - continuously exchanging blocks

✵ *Appropriate for* larger caches like L3

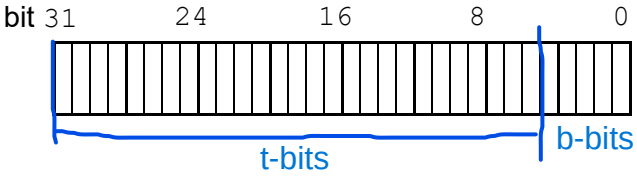
Fully Associative Caches - Unrestrictive

Fully Associative Cache is a cache having one set with E lines where Blocks can be stored in any line

→ What is the address breakdown if blocks are 32 bytes and there are ~~1024~~ 1 sets?

B = 32 b = 5
S = 1 s = 0

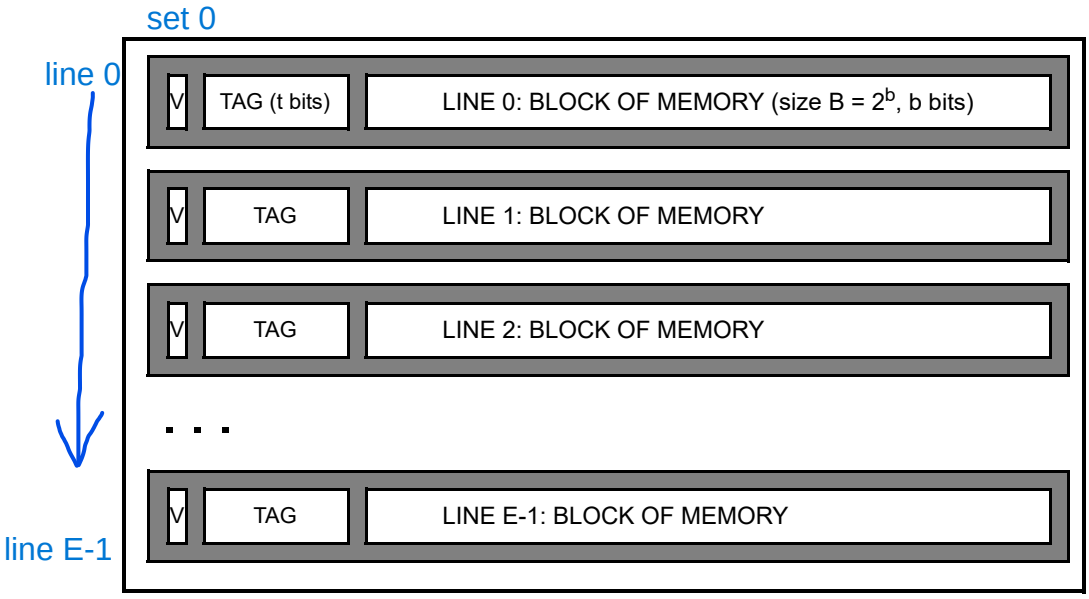
32-bit Address Breakdown



→ Is the cache operation fast O(1) or slow O(E) where E is the number of lines? SLOW!

must search each line

O(1) set selection + O(E) line matching



→ What happens when two different memory blocks map to the same set?

Choose a free line

+ reduces conflict misses

→ How many lines should a fully associative cache have?

C = S x E x B = 1 x E x B

C = E x B => E = C/B

❋ *Appropriate for* a small cache (L1)

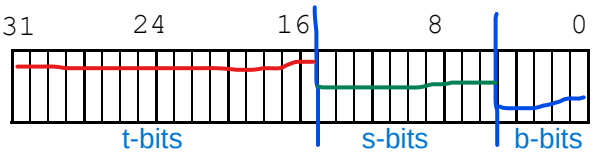
***** MOST IMPORTANT FOR TESTS/QUIZZES

Set Associative Caches - Sweet!

Set Associative Cache is a cache commonly used today having S sets and E lines / set

→ What is the address breakdown if blocks are 32 bytes and there are 1024 sets?

32-bit Address Breakdown

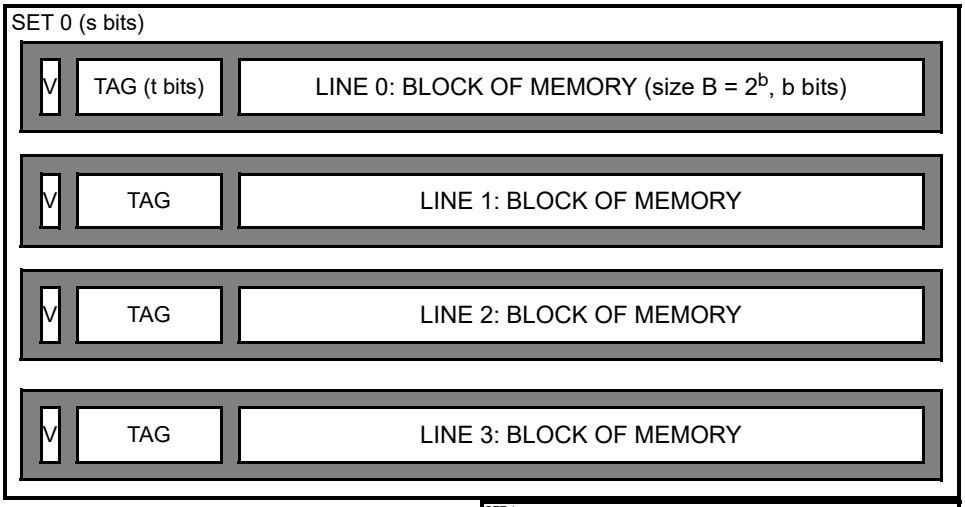


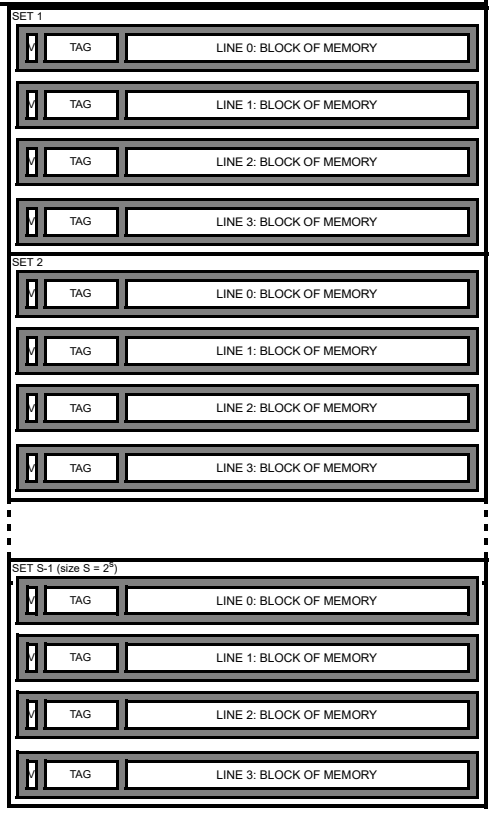
B = 32, b = 5
S = 1024, s = 10
E = 4 lines/set

+ FAST
  O(1) set selection
  O(E) line matching

+ reduces conflict misses

- more complex circuitry
  to match t-bits to tag





Let E be lines/set
the "associativity" of the set

E = 4 is Four-way set associative

E = 1 is Direct-mapped cache

❋ ***C = (S, E, B, m)***

m: # bits in address

Let C be the cache size in bytes

C = S x E x B

S E B M

→ How big is a cache given (1024, 4, 32, 32)?

C = 1024 x 4 x 32 = 2^10 x 2^2 x 2^5 = 2^17 = 128 k

→ What happens when E+1 different memory blocks map to the same set?

must evict ("kick out" or overwrite) and replace
"Replacement Policy"

Replacement Policies

Assume the following sequence of memory blocks

are fetched into the same set of a 4-way associative cache that is initially empty:

w.s. = b1, b2, b3, b1, b3, b4, b4, b7, b1, b8, b4, b9, b1, b9, b9, b2, b8, b1

M M M H H M H M H M M M H H H H H H

1. *Random Replacement*

→ Which of the following four outcomes is possible after the sequence finishes?

→ Assume the initial placement is random.

	L0	L1	L2	L3	
? → 1.	b9	b1	b8	b2	Yes
2.	b1	b2	--	b8	NO
3.	b1	b4	b7	b3	Yes (unlikely though)
4.	b1	b2	b8	b1	NO (b1 in 2 spots)

L0	L1	L2	L3
b1	b2	~~b3~~	~~b4~~
		~~b7~~	b8
		~~b4~~	
		b9	

2. *Least Recently Used* (LRU)

Must track when line was used
Use LRU queue "priority queue"
When line is used move to "front"
Simply use status bits to track LRU

→ What is the outcome after the sequence finishes?

Assume the initial placement is in ascending line order (left to right below).

L0	L1	L2	L3
b1	b2	b3	b4
	b7	b5	b8
	b9	b8	
		b9	
		b2	

rear ← front ← LRU end

b1 b8 b2 b9 b1 b9 b1 b9 b4 b8 b1 b7 b4 b3 b1 b3 b2 b1

L0 L3 L2 L1 L0 L1 L0 L1 L3 L2 L0 L2 L3 L2 L0 L2 L1 L0

LRU b9 | LRU b8 | LRU b7

3. *Least Frequently Used* (LFU)

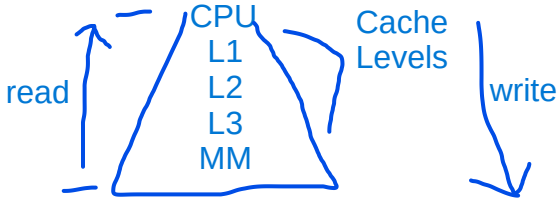
Must track how often line is used
Each line has a counter
- zero counter when line gets a new block
- increment counter when line is accessed
- if tie, choose randomly

→ Which blocks will remain in the cache after the sequence finishes?

L0	~~L1~~	L2	~~L3~~	L1	~~L1~~	L1	~~L2~~	L2
b1	~~b2~~	b3	~~b4~~	b7	~~b8~~	b9	~~b2~~	b8
counter: 4		1		2		2		

✵ *Exploiting replacement policies* to improve performance is NOT EASY, so we don't do it (as programmers)

Writing to a Cache



- *Reading data copies* a block of memory into cache level
- *Writing data requires that* these copies are consistent

Write Hits

occur when writing to a block that is in the cache

→ When should a block be updated in lower memory levels?

1. ***Write Through***: write to this cache and update next lower level
   - must wait for next lower level to write
   - more bus traffic, passing all writes through

(L2)

2. ***Write Back***: write to next lower level when block is evicted from (L1)

   \+ faster, no wait for next level

   - must track if line (block data) has changed

   add a status bit, a "dirty" bit ≡ changed

Write Misses

occur when writing to a block that is not in the cache level

→ Should space be allocated in this cache for the block being changed?

1. ***No Write Allocate***: write directly to next lower level cache bypassing this cache level
   - must wait for lower level to write

2. ***Write Allocate***: read block into current cache level and then write to it
   - must wait to get block from lower level
   - more bus traffic, must pass block from lower level to current level

Typical Designs

1. **Write Through** paired with NO WRITE ALLOCATE
2. **Write Back paired** with WRITE ALLOCATE

→ Which best exploits locality? pairing 2 (for both spatial and temporal locality)

Cache Performance

Metrics

hit rate # hits / # mem accesses, higher is better

hit time time to determine a hit, shorter is better
= set selection + line matching
O(1) O(E)

miss penalty, additional time to process a miss, shorter is better

Larger Blocks (S and E unchanged) increase size of B

hit rate Better, more spatial locality

hit time Same, essentially

miss penalty Worse, takes longer to transfer larger block

THEREFORE Block sizes are small, typically 32 bytes or 64 bytes

More Sets (B and E unchanged)

hit rate better, more temporal locality in cache

hit time worse, more sets more complex set selection

miss penalty same

THEREFORE faster caches (L1) are smaller (fewer sets)
slower caches (L3) are larger

More Lines E per Set (B and S unchanged)

hit rate Better, more temporal locality in cache

hit time WORSE, O(E) more lines to check for match

miss penalty worse, more lines slows choosing a victim to evict

THEREFORE faster caches have fewer lines (L1)
slower caches have more lines (L3)

Intel Quad Core i7 Cache (gen 7)

all: 64 byte blocks, use pseudo LRU, write back

L1: 32KB, 4-way Instruction & 32KB 8-way Data, no write allocate
L2: 256KB, 8-way, write allocate
L3: 8MB, 16-way (2MB/Core shared), write allocate

↳ 16-way = 16 lines per set; 8-way = 8 lines per set; ... etc.

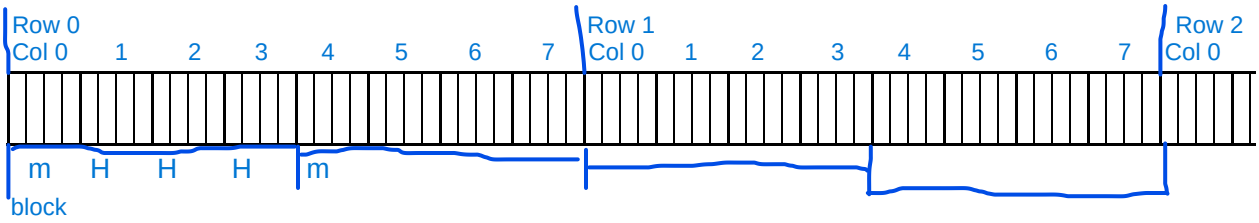
Impact of Stride

Stride Misses % misses = min(1, (word size * k) / B) * 100
where k is stride length in words 1 word => k = 1
and B is block size in bytes (4 bytes)

Example:

```
int initArray(int a[][8], int rows) {
   for (int i = 0; i < rows; i++)
      for(int j = 0; j < 8; j++)
         a[i][j] = i * j;
}
```

→ Draw a diagram of the memory layout of the first two rows of `a`:



Assume: `a` is aligned with cache blocks (and heap pages)
is too big to fit entirely into the cache
words are 4 bytes, block size is 16 bytes
direct-mapped cache is initially empty, write allocate used

→ Indicate the order elements are accessed in the table below and mark H for hit or M for miss:

a[i][j]	j = 0	1	2	3	4	5	6	7
i = 0	m	H	H	H	m	H	H	H
1	m	H	H	H	m	H	H	H
...								

% misses = min(1, (wordsize * k) / B) * 100
= min (1, (4 * 1) / 16) * 100
= 1/4 * 100 = 25 %

→ Now exchange the `i` and `j` loops mark the table again:

a[i][j]	j = 0	1	2	3	4	5	6	7
i = 0	m	m						
1	m	m						
...	m	m						

% miss = min (1, 4 * 8 / 16) * 10
= 1 * 100 = 100 %

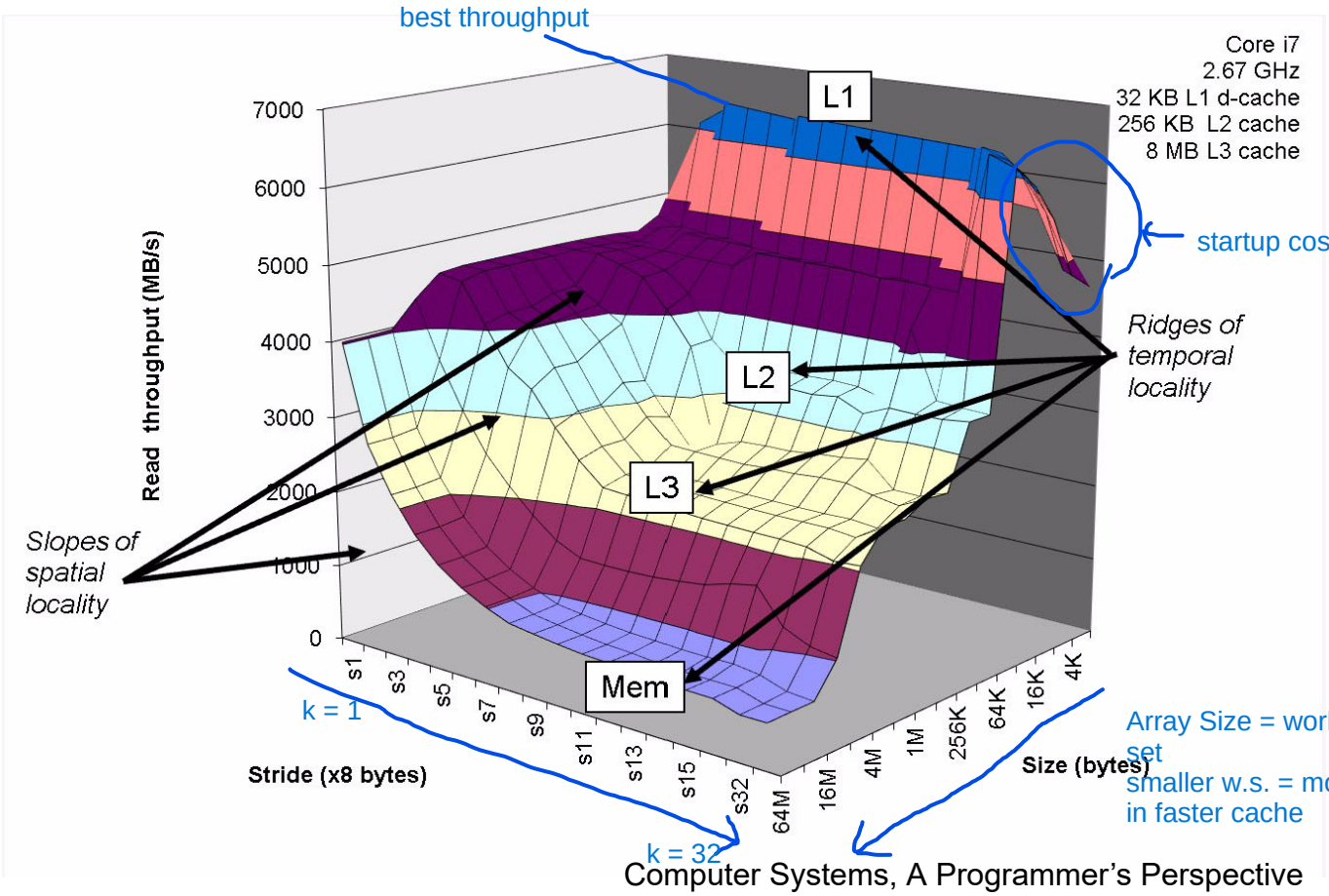
Memory Mountain

Independent Variables

stride - 1 to 16 double words step size used to scan through array
size - 2K to 64 MB arraysize

Dependent Variable

read throughput - 0 to 7000 MB/s



Computer Systems, A Programmer's Perspective
Second Edition, Bryant and O'Hallaron

Temporal Locality Impacts (size)

Spatial Locality Impacts (stride)

❋ *Memory access speed is not characterized* by a single value. It is a landscape that can be exploited by temporal and spatial locality

C, Assembly, & Machine Code

C Function	Assembly (AT&T)	Machine (hex)
`int accum = 0;`		
`int sum(int x, int y)`	`sum:`	
`{`	`  pushl %ebp`	`55`
	`  movl %esp, %ebp`	`89 e5`
	`  movl 12(%ebp), %eax`	`8b 45 0C`
`  int t = x + y;`	`  addl 8(%ebp), %eax`	`03 45 08`
`  accum += t;`	`  addl %eax, accum`	`01 05 ?? ?? ?? ??`
`  return t;`	`  popl %ebp`	`5D`
`}`	`  ret`	`C3`

C

- a HLL enables prod coding
- helps write correct code
- can be compiled and run on different machines

→ What aspects of the machine does C hide from us?
machine instructions, addressing modes, registers, condition code registers

Assembly (ASM)

- human readable representation of machine code
- very machine dependent

→ What ISA (Instruction Set Architecture) are we studying? IA-32

→ What does assembly remove from C source?
no logical control structures (if, while loop)
no local variable names and no data types
no composite data structures (arrays, structs, unions)

→ Why Learn Assembly?
1. To better understand the stack
2. To better recognize inefficiencies
3. To better understand compiler optimizations

Machine Code (MC) **is**

- elementary CPU instructions and data in binary
- the encodings that a part. machine understands

→ How many bytes long is an IA-32 instructions? 1 - 15 bytes