CS 354 - Machine Organization & Programming
Tuesday Oct 24, and Thursday Oct 26,2023

Midterm Exam 2 - Thursday Nov 9th, 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,

Project p3: DUE on or before Friday, Oct 27

Project p4A: DUE on or before Friday Nov 3,

Project p4AQuestions: DUE on or before Monday Nov 6,

Project p4B: DUE on or before Friday Nov 10,

Learning Objectives

- able to determine hit or miss given address and cache contents
- able to determine set number (index) from s-bits
- able to determine if an address is within a given block
- understand the effect of cache configuration on given sequence of address (working set)
- understand difference btw direct mapped, fully associative, and set associative caches
- able to explain and implement Least Frequently Used replacement policies
- able to explain and implement Least Recently Used replacement policy
- understand diff btw write-through, write-back, no-write allocate, write allocate caches
- compare cache performance of different cache configurations for working set sequence
- describe the impact of stride and the scales of the memory mountain

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 Instr. 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

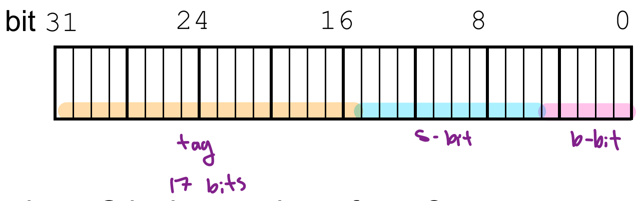
Direct Mapped Cache is a cache having S sets with 1 line / set where memory blocks map to exactly 1 set

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

B = 32 bytes = 5 b-bits

S = 1024 sets = 10 s-bits

32-bit Address Breakdown

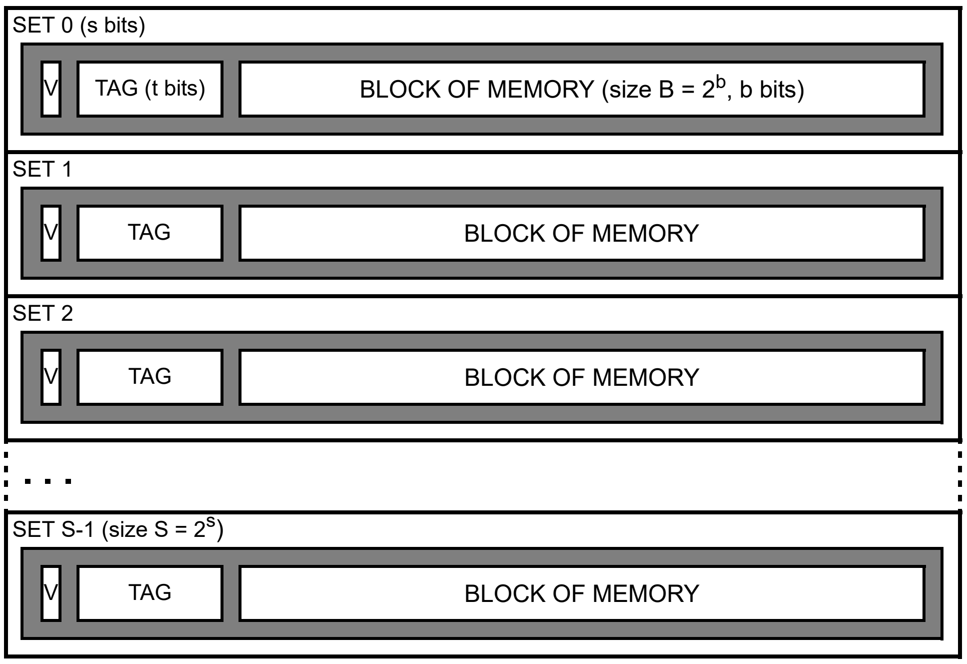


→ Is the cache operation fast O(1) or slow O(S) where S is the number of sets?

no search needed O(1) set selection + O(1) line matching (v-bit)

(+) has simple circuitry

if tag matches t-bits



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

"conflict miss" set's line stores one block and 1 line / set

"thrashing" can occur

✵ *Appropriate for* larger caches (L3)

Fully Associative Caches - Unrestrictive

Fully Associative Cache is a cache having 1 set with (E) lines per set

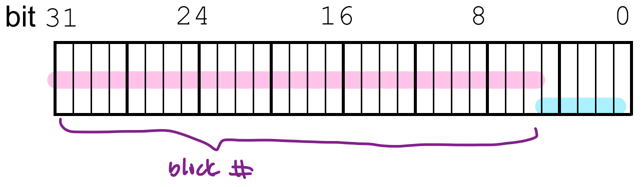
where mem. blocks can be stored in any line

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

B = 32 , 5 b-bits

S =

32-bit Address Breakdown



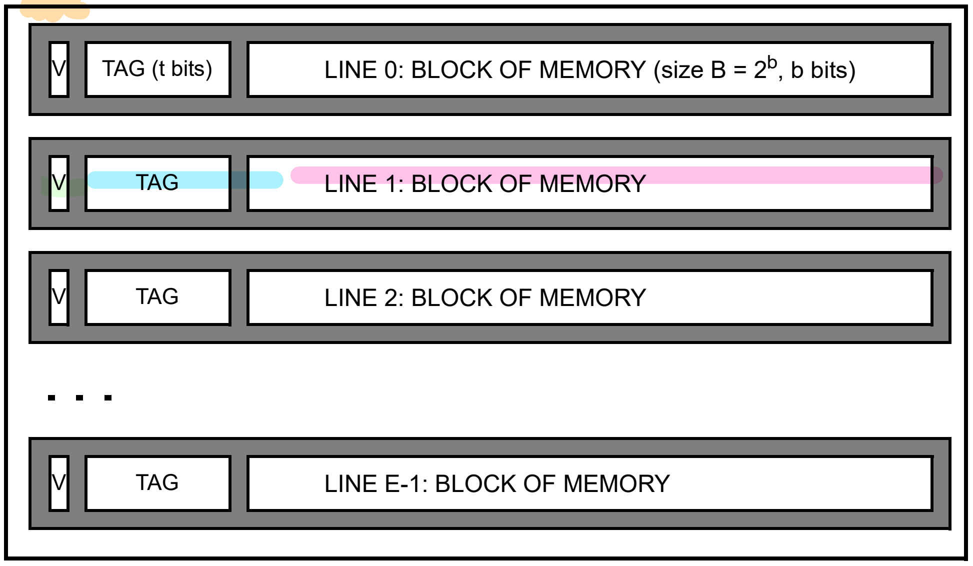
→ Is the cache operation fast O(1) or slow O(E) where E is the number of lines?

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

↑ any 1set

(–) complex circuitry to match t bits in addr with tag in each line

(–) more t-bits



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

choose a free line - reduces conflict miss

→ How many lines should a fully associative cache have?

cache size = $C = S \times B \times \overset{\text{lines}}{E}$

$1 \times B \times E$

$E = \frac{C}{B}$

✵ *Appropriate for* small caches (L1)

Set Associative Caches - Sweet!

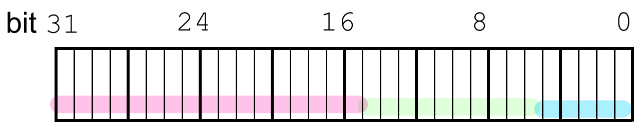
Set Associative Cache is a cache commonly used today

having (S) sets with (E) lines / set

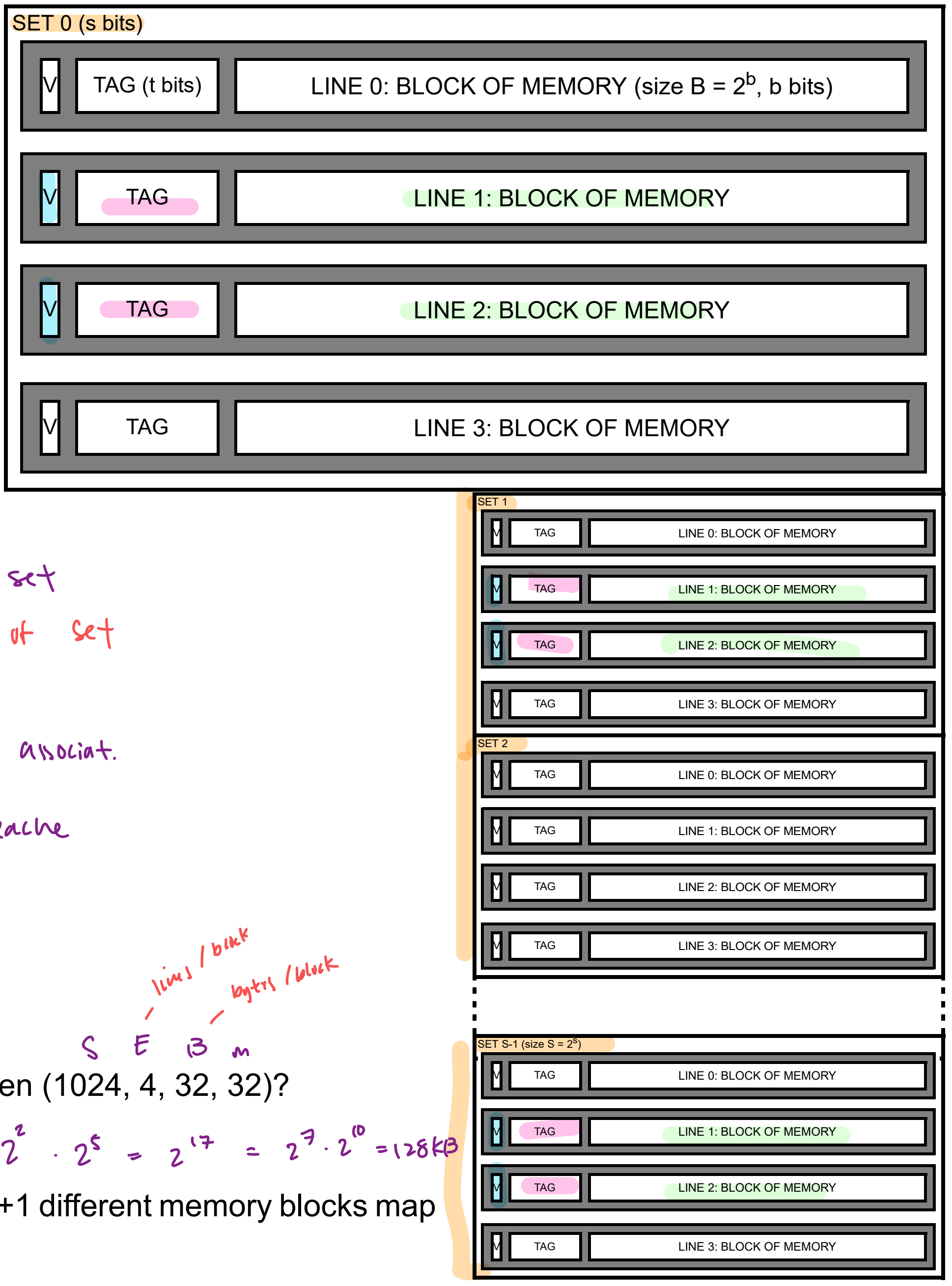
mem blocks map to 1 set and can be in any line w/in that set

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

32-bit Address Breakdown



$B = 32$ $b = 5$ $S = 1024$ $s = 10$



(+) fast

O(1) set matching

O(E) line matching

(+) reduces conflict miss

(−) more circuitry

(not as much as fully assoc.)

Let E be # of lines / set

associativity of set

E = 4 is 4 way set associat.

E = 1 is direct map cache

✵ ***C = (S, E, B, m)***

Let C be cache size

$C = S \times E \times B$ S $\overset{\text{lines / block}}{E}$ $\overset{\text{bytes / block}}{B}$ m

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

$C = S \times E \times B = 2^{10} \cdot 2^{2} \cdot 2^{5} = 2^{17} = 2^{7} \cdot 2^{10} = 128KB$

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

use replacement policy to choose line to be replaced

(block)

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:
`b1, b2, b3, b1, b3, b4, b4, b7, b1, b8, b4, b9, b1, b9, b9, b2, b8, b1`

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	~~b1~~	b8	NO
3.	b1	b4	b7	b3	Yes
4.	~~b1~~	b2	b8	~~b1~~	NO

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

2. *Least Recently Used* (LRU)

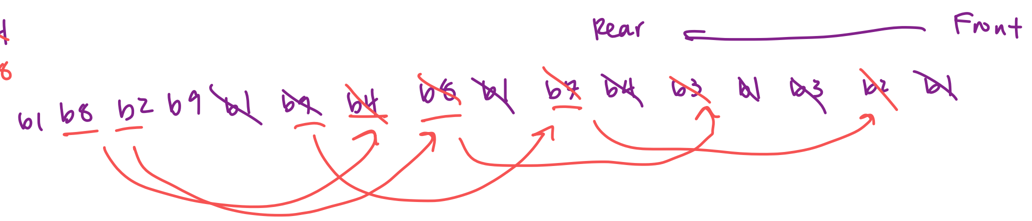
track when line last used

use LRU queue – when line used move to front

use status bits to track

→ What is the outcome after the sequence finishes?
Assume the initial placement is in ascending line order (left to right below). (LRU)

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



Rear
Front
b1 b8 b2 b9 ~~b1~~ ~~b9~~ ~~b4~~ ~~b8~~ ~~b1~~ ~~b7~~ ~~b4~~ ~~b3~~ ~~b1~~ ~~b3~~ ~~b2~~ ~~b1~~

3. *Least Frequently Used* (LFU)

must track how often a line is used

each line has a counter – zeroed when line gets new block
– incr. when line is accessed

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

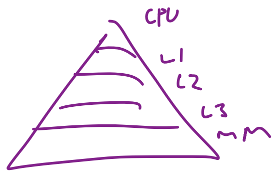
L0	L1	L2	L3
b1	b4	b9	b8

b1 ||||, ~~b2~~, ~~b3~~ |, b4 ||, ~~b7~~, ~~b8~~, b9 ||, ~~b2~~, b8

✵ *Exploiting replacement policies* to improve performance

is not easy or always improves

Writing to a Cache



✵ *Reading data copies* a block of mem into cache

✵ *Writing data requires that* there copies are consistent

Write Hits

occur when writing to a block that is in this cache

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

1. ***Write Through***: write to this and next lower cache level

   (−) must wait for lower level to do write

   (−) more bus traffic

2. ***Write Back***: write to the next lower level only when changing line "line is evicted"

   (+) faster – no wait

   (−) must track if changed, Dirty Bit is set

   if replaced and DB set, must write to lower level

Write Misses

occur when writing to a block that is not in cache

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

1. ***No Write Allocate***: write directly to next lower level bypassing "this" cache

   (−) must wait for lower level

   (+) less buss traffic (less copying back n. forth)

2. ***Write Allocate***: read block into cache 1st then write to it

   (−) must wait to read from lower level

   (−) more bus traffic

Typical Designs

1. **Write Through** paired with no write alloc – make sure next lower level
2. **Write Back paired** with write alloc – make sure this level is written

→ Which best exploits locality?

2. also symmetric w/ read

Cache Performance

Metrics

hit rate # hits / # mem access

hit time time to determine cache hit

miss penalty, additional time to process a miss

① **Larger Blocks** (S and E unchanged)

hit rate better, more spacial locality per block

hit time same

miss penalty worse, more time to transfer larger blocks

THEREFORE block size tend to be small, 32 bytes or 64 bytes

② **More Sets** (B and E unchanged)

hit rate better, more blocks in the cache → temporal locality ↑

hit time worse, slower set selection

miss penalty same

THEREFORE faster caches have fewer sets

slower caches are larger with more sets

③ **More Lines E per Set** (B and S unchanged)

hit rate better, ↑ temporal locality

(+) fewer conflict miss

hit time worse, slower line matching

miss penalty worse, harder to detect miss

THEREFORE faster caches have fewer line/set

slower caches have more lines/set

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

P4A
royal, snares,
rockhopper

Impact of Stride

Stride Misses % misses = min (1 , (word size * k)/B) * 100

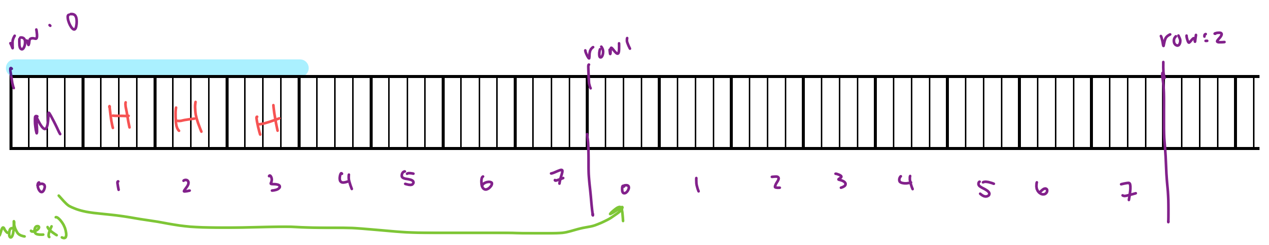
* where K is stride length in <u>WORDS</u> !!!

and B is block size in 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`:



MIDTERM

Assume: a is aligned with cache blocks 4. page on hard drive
is too big to fit entirely into the cache
words are 4 bytes, block size is 16 bytes
E = 1 → 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	1 M	2 H	3 H	4 H	5 M	6 H	7 H	8 H
1								
...								

% misses = $\frac{1}{4}$ = min (1 , word size · k (B)) · 100

= min (1, (4 · 1)/ 16) · 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	1 M	M						
1	2 M	M						
...	3 M	M						

4 M M

⋮

4096 M

% miss = 100%

stride is 8 words, and

Memory Mountain

Independent Variables

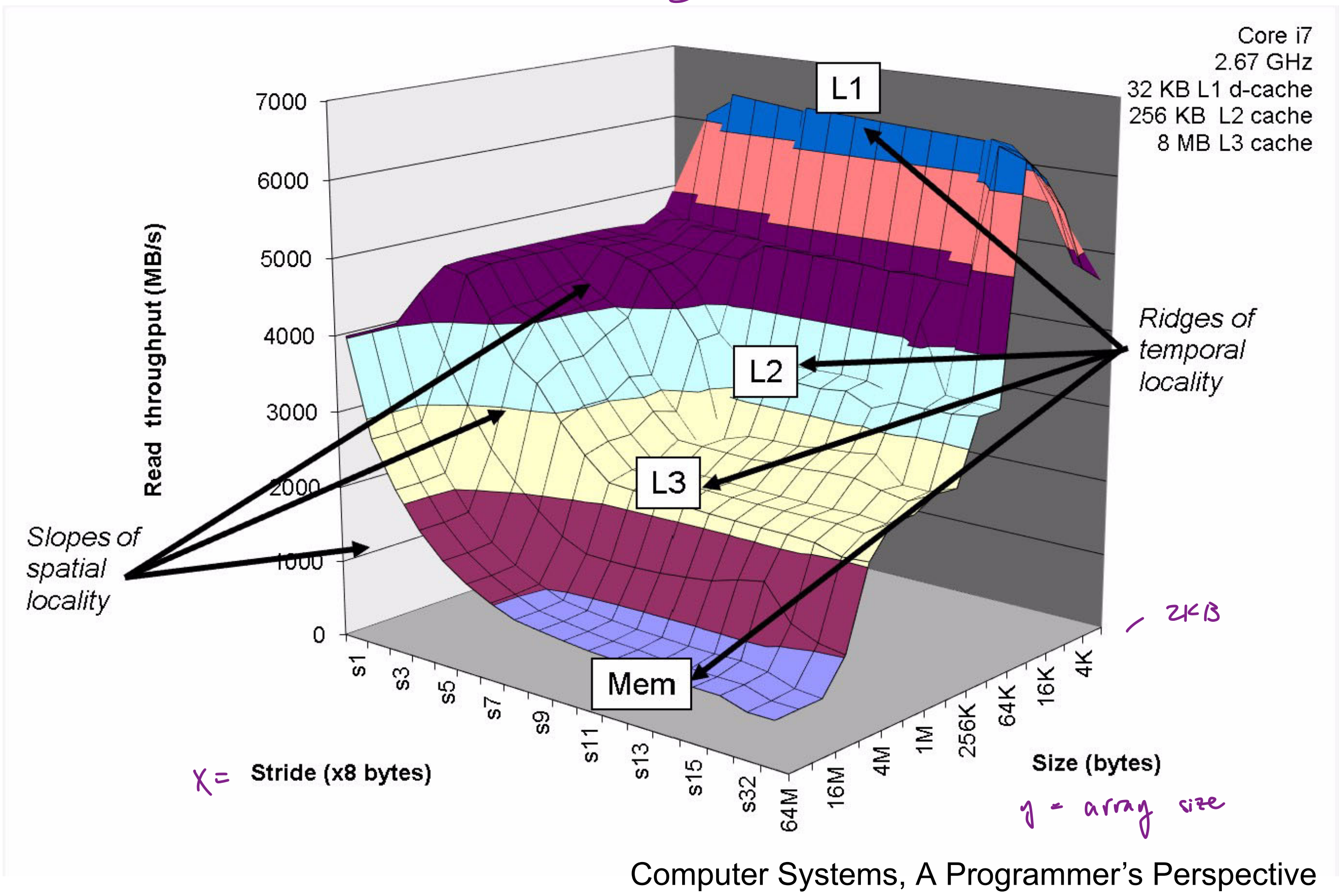
x = stride - 1 to 16 double words step size used to scan through array

y = size - 2K to 64 MB arraysize

Dependent Variable

read throughput - 0 to 7000 MB/s performance

z =



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 & spatial locality

C, Assembly, & Machine Code

in the begining

```
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

- is a HLL that enables more prod coding
- easier to write correct code
- can be compiled and run on diff machine

→ What aspects of the machine does C hide from us? machine instr.

addressing modes

registers, , conditional code

Assembly (ASM)

- human readable representation of machine code
- very mach dependent

→ What ISA (Instruction Set Architecture) are we studying? IA-32 x86-64

→ What does assembly remove from C source? HLL constructs

logical control : if , switch, else

local variables (data types

composite structs

→ Why Learn Assembly?

1. to understand the stack
2. to identify inefficiencies
3. to understand compiler optimization

Machine Code (MC) **is**

- elementary cpu instructions and data (binary)
- the encoding that a part. machine understands

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

1 to 15 bytes