CS 354 - Machine Organization & Programming
Tuesday Mar 12, and Thursday Mar 14,2024

Midterm Exam 2 - Thursday April 4th, 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**

Project p3: DUE on or before Friday, Mar 15

Homework hw4: DUE on or before Monday Mar 18,

Project p4A: DUE on or before Thurs Mar 21,

Project p4AQuestions: DUE on or before Thurs Mar 21,

Project p4B: DUE on or before Friday Apr 5,

Learning Objectives

- determine hit or miss given address and cache contents
- determine set number (index) from s-bits
- determine if an address is within a given block
- explain the effect of cache configuration on given sequence of address (working set)
- explain difference btw direct mapped, fully associative, and set associative caches
- implement Least Frequently Used replacement policies
- implement Least Recently Used replacement policy
- explain 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

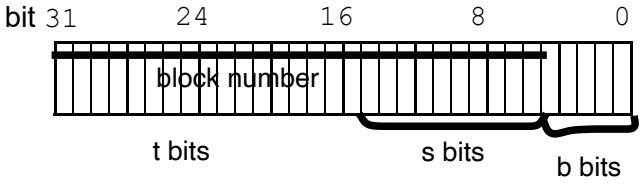
Direct Mapped Cache is a cache having S sets with 1 cache line per set

where memory blocks map to exactly 1 set

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

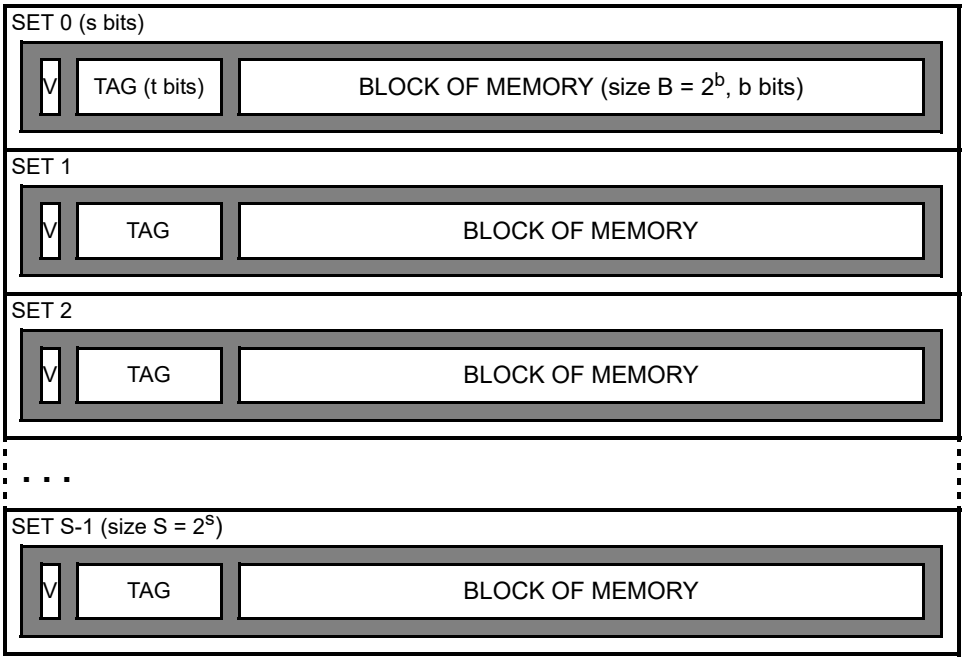
B = 32 5 b bits
S = 1024 10 s bits

32-bit Address Breakdown



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

No search, set selection O(1) + Line Matching O(1)
no search for line since 1 per set, no need search set since simple circuitry



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

CONFLICT MISS - same s bits but different t bits so diff block number — sets store one block only

can cause thrashing — continuously exchanging blocks

✵ *Appropriate for* larger caches

Fully Associative Caches - Unrestrictive

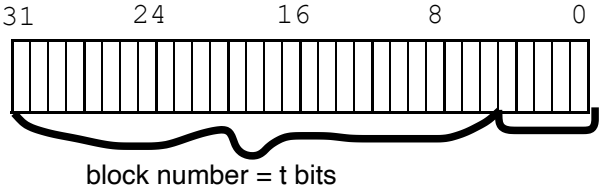
Fully Associative Cache is a cache having 1 set with E lines in that set

where memory blocks can be at any line in set

→ What is the address breakdown if blocks are 32 bytes and there are 1~~024~~ sets?

32 = 2^5 -> 5 b bits

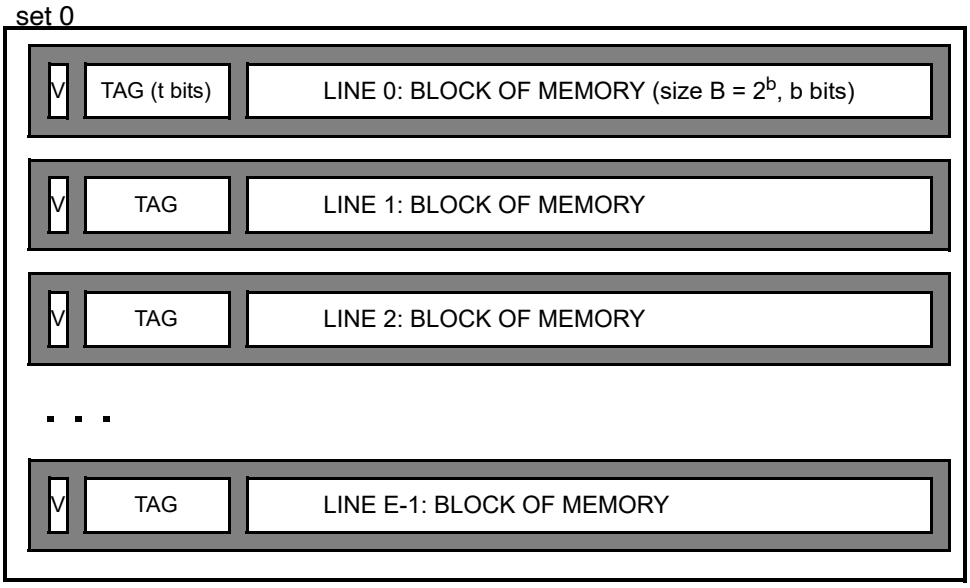
32-bit Address Breakdown



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

set selection O(1), Line Matching O(E)

- complex circuitry to make it such that every block goes to every line
- also have to match more t bits



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

they all map to the same single set — choose a free line

→ How many lines should a fully associative cache have?

1. Determine cache size and block size
C = S * E * B where E is number of lines
since S = 1 in this case, E = C/B -> number of lines in this fully associative cache

❋ *Appropriate for* small cache (L1)

Set Associative Caches - Sweet!

Set Associative Cache is a cache commonly used today

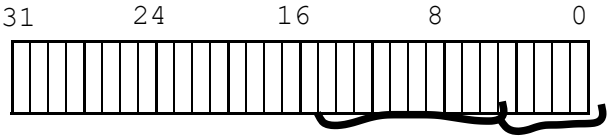
has S sets with E lines per set

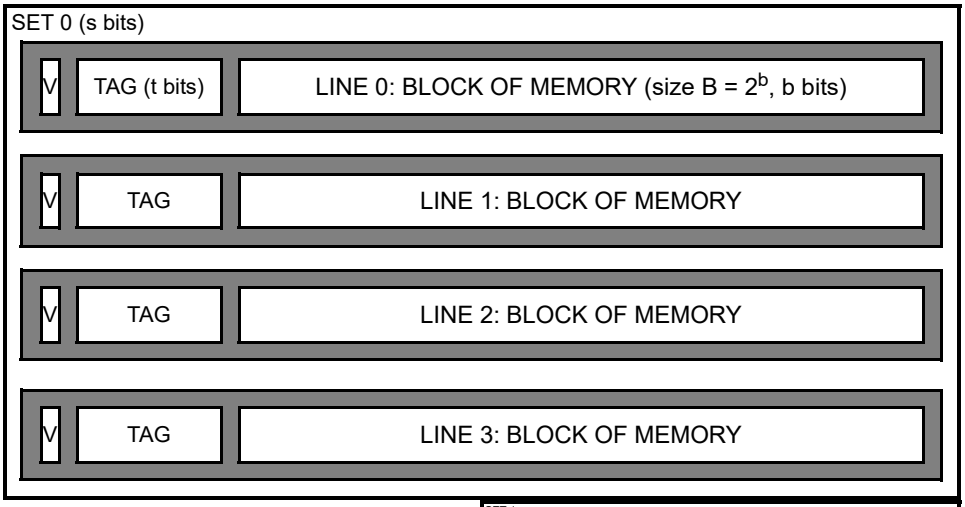
memory blocks map to a single set with E lines (can be any line in that set)

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

5 b bits, 10 s bits, 17 t bits

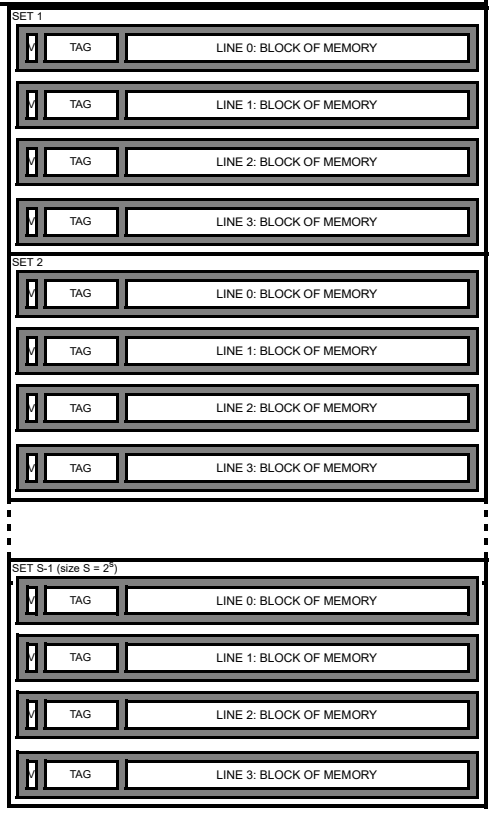
32-bit Address Breakdown





set selection O(1) compute from s bits
line matching O(E) where E is num lines
but E is "smaller" than before

+ reduce conflict miss
- but more circuitry than direct mapped
less than unrestricted



Let E be number of lines per set — "Associativity of the Set"

E = 4 is Four Way Set Associative

E = 1 is Direct Mapped Cache

❋ ***C = (S, E, B, m)*** — number of bits in address tell us number of tag bits

Let C be the cache size in bytes

C = S * E * B

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

C = 2^10 * 2^2 * 2^5 = 2^17 = 128Kb

m = 32 bits, 13 t bits

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

use replacement policy to pick a victim to kick out

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
2.	b1	b2	--	b8
3.	b1	b4	b7	b3
4.	b1	b2	b8	b1

L0 L1 L2 L3
b1 b2 b3 b4
b1 b2 b7 b4
b1 b2 b7 b8
b1 b2 b4 b8
b1 b2 b9 b8
— 1 is possible

2 is not possible
3 is possible
4 not possible

2. *Least Recently Used* (LRU)

must track the line last used, and have a least recently used LRU queue — when line is used, move it in queue

Hardware — use status bits to track

→ What is the outcome after the sequence finishes? b1 b9 b2 b8
Assume the initial placement is in ascending line order (left to right below).

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

LRU MRU
b1 b2 b3 b1 b3 b4 b7 b1 b8 b4 b9 b1 b9 b2 b8 b1

3. *Least Frequently Used* (LFU)

Must track how often each line is used
Each line has a count
-- set to 0 when line gets a new block
— increment 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
2	0	1	2	0	0	2	0	0

b1 b9 b8 b4

❋ *Exploiting replacement policies* to improve performance is NOT EASY
so programmers don't do it

Writing to a Cache

❋ *Reading data copies* a block of memory into cache levels

❋ *Writing data requires that* these copies are consistent

Write Hits

same question at each cache level

occur when writing to a block that is in "the" cache (cache K)

→ When should a block be updated in lower memory levels? updated to K+1 cache level

1. ***Write Through***: write to this level (K) and update next level (K+1)

   -ve must wait for lower level to write
   -ve more bus traffic passing all writes through
   +ve or add bypass cache with write buffer (smaller side cache to let it wait for K+1 to do write)

2. ***Write Back***: write to next level (K+1) only when line changes "block evicted"

   +ve faster, no wait
   -ve must track if line has been changed, so can update lower caches when evicted from cache K, add "DIRTY" bit
   +ve less bus traffic, only update when done with block, no write to K+1 until block removed from K

Write Misses

occur when writing to a block that is not in "the" cache (cache K)

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

1. ***No Write Allocate***: write directly to next lower level (K+1) bypassing this cache (K)

   -ve must wait for next lower level to write (K+1)
   +ve less bus traffic since no wait to get block

2. ***Write Allocate***: read block into curr level(K) first, then write (K+1)

   -ve must wait for block from level (K+1) to read block in
   -ve more bus traffic, must pass block to current cache level

Typical Designs

1. **Write Through** paired with No Write Allocate "get it to K+1"
2. **Write Back paired** with Write Allocate "keep it here in K"

→ Which best exploits locality? 2 exploits locality best,
it keeps in highest level before having to go lower level

Cache Performance

Metrics

hit rate number of hits / number of memory access, higher == better

hit time time to determine cache hit, set selection + line matching, lower == better

miss penalty, additional time to process a miss, lower == better

Larger Blocks (S and E unchanged) C = S * E * B

hit rate better, more spatial locality per block, more values from array in same block

hit time same, since set selection and line matching same number, so take same time

miss penalty worse, more time to transfer larger block

THEREFORE block sizes are small, typically 32 or 64 bytes

More Sets (B and E unchanged)

hit rate better, more temporal locality, more sets, so more diff blocks can be in cache at same time

hit time worse, since more sets, more of complexity (circuitry) to find correct matching set (still O(1))

miss penalty same, since block size the same

THEREFORE faster caches are smaller (L1), slower caches are larger (L3)

More Lines E per Set (B and S unchanged)

hit rate better, more temporal locality, more sets, so more diff blocks can be in cache at same time, fewer conflict misses

hit time worse, slower line matching O(E)

miss penalty worse, more lines, longer to detect match

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

lscpu to find out

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

Impact of Stride

measured in words (1 word is 4 bytes)

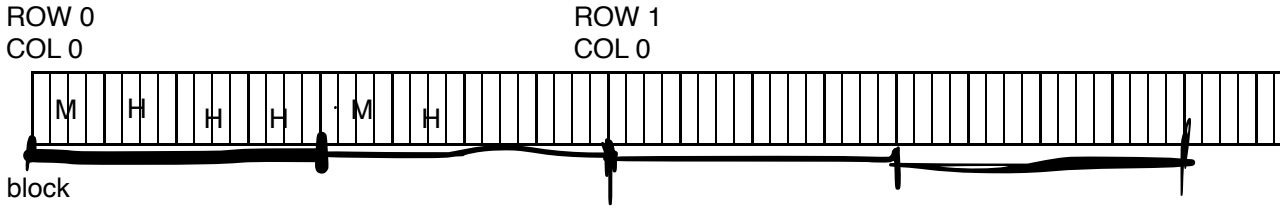
Stride Misses % misses = min(1, (word_size * k) / b) * 100

where K is stride length in WORD 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`:



Assume: `a` is aligned with cache blocks
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				
...								

% misses = min (1, (word_size * k)/ B)

(4 * 1 / 16) = 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							

% misses = min (1, (word_size * k)/ B)

(4 * 8 / 16) = 100%

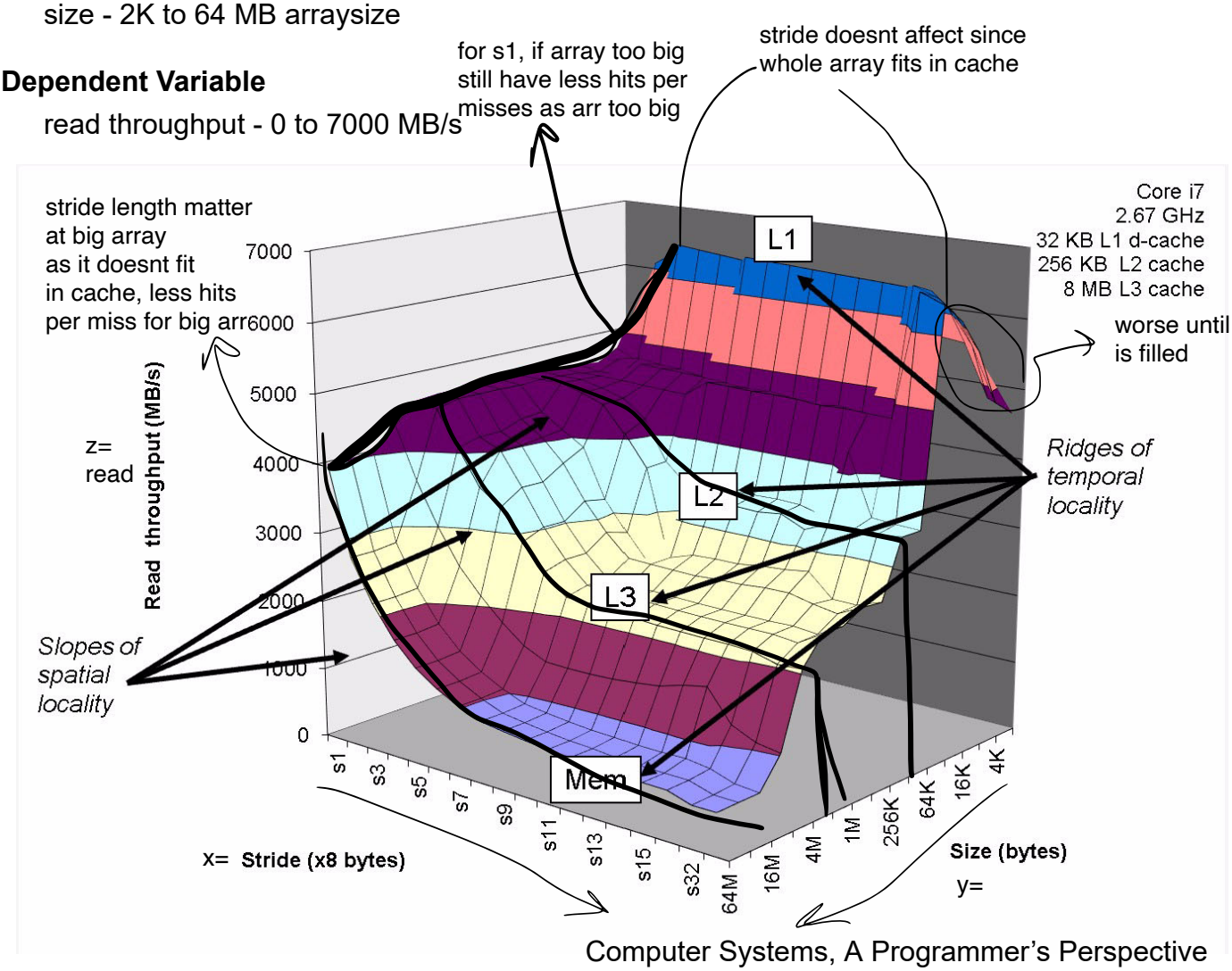
Memory Mountain

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

factor of ~40 between L1 and MM

as array size got bigger, lost about 40% of throughput

Spatial Locality Impacts stride

factor of ~7 from top to bottom (4000:600 MB/s)

❋ *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

```
int accum = 0;
int sum(int x, int y)
{


   int t = x + y;
   accum += t;
   return t;
}
```

Assembly (AT&T)

```
sum:
   pushl %ebp
   movl %esp, %ebp
   movl 12(%ebp), %eax
   addl 8(%ebp), %eax
   addl %eax, accum
   popl %ebp
   ret
```

Machine (hex)

1 hex digit is 4 bits, 4 bits + 4 bits = 1 byte

```
55              1byte
89 e5           2byte
8b 45 0C        3byte
03 45 08
01 05 ?? ?? ?? ??    ?byte
5D
C3
```

2 hex values is 1 byte in machine code

C

- a high level language to enable us to be more productive coders
- helps us write correct code, with syntax and type checking
- can compile and run on different machines

→ What aspects of the machine does C hide from us? low-level machine details

- machine code
- addressing modes
- registers, etc.

Assembly (ASM)

- Human readable machine code
- Machine dependent

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

→ What does assembly remove from C source?

high level language constructs
- logical control structures
- local variable names and data types
- composites, arrays, structs, union

→ Why Learn Assembly?

1. better understand the stack
2. identify code inefficiencies and vulnerabilities
3. understand compiler optimisations better

Machine Code (MC) **is**

- elementary CPU instructions and data in binary
- the unique encoding that a particular machine understands

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