

# Contents

Bits, Bytes, & Integers .....	1
bit level manipulations .....	1
integers .....	1
casting integers .....	2
byte order .....	2
Machine Programming .....	2
history .....	2
assembly/machine code view .....	2
machine code: control .....	4
machine code: procedures .....	4
machine code: data .....	5
machine code: advanced .....	5
the memory hierarchy .....	6
the memory abstraction .....	6
RAM .....	6
locality .....	7
memory hierarchy .....	7
memory mountain .....	9
storage technologies and trends .....	9

## Bits, Bytes, & Integers

### bit level manipulations

- binary: get more precision over n-ary or smth
- and (&), or (|), not (~), xor (^)
- shifts
  - $x \ll y$ 
    - throw away extra bits at left
    - fill with 0s on right
    - $x \cdot 2^y$
  - $x \gg y$ 
    - throw away extra bits at right
    - unsigned shift: uses logical shift: fill with 0s on left
    - signed shift: uses arithmetic shift: replicate sign bit on left
    - *undefined*: shift amtn  $< 0$  or  $\geq$  word size
    - $\lfloor x / 2^y \rfloor$  i.e. rounding to left
      - to round to zero ( $\lceil x / 2^y \rceil$ ):  $(x + (1 \ll y) - 1) \gg y$
- logical &&, ||, !
  - views 0 as false, nonzero as true
  - returns 0 or 1

### integers

- limits
  - $U_{\text{Max}} = 2^w - 1$

- $T_{\min} = -2^{w-1}$
  - $T_{\max} = 2^{w-1} - 1$
- $-x = \sim x + 1$  in two's complement
  - but if  $x = T_{\min}$  (most negative two's complement), you get back  $T_{\min}$

### casting integers

- constants are signed ints by default
  - specify 10U for unsigned or 24L for long
  - source of mistakes: make sure to, eg, `1ULL << 36`
- signed  $\leftrightarrow$  unsigned: maintain bit pattern
  - may add/subtract  $2^w$  (0b1000 is 8 unsigned, -8 signed.)
  - casting to larger? sign extend.
  - casting to smaller? drop significant bits.
- mix of signed and unsigned in expression (eg `==`)? implicitly casted and eval'd in unsigned.

### byte order

...	0x100	0x101	0x110	0x111	...
...	01	23	45	67	...

Table 1: big endian

...	0x100	0x101	0x110	0x111	...
...	67	45	23	01	...

Table 2: little endian

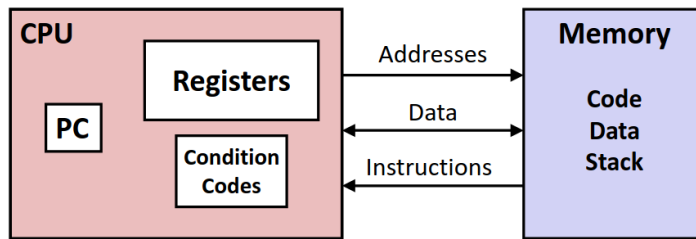
## Machine Programming

### history

- intel x86 processors
  - a Complex Instruction Set Computer (CISC), lots of instructions
  - Reduced: (RISC) can be fastish but esp good for low power
- architecture: processor design spec?? needed to know how to write assembly/machine code??
- microarchitecture: implementation of architecture
- machine code: byte-level programs processors exec.
- assembly code: text readable machine code

### assembly/machine code view

# Assembly/Machine Code View



## Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called "RIP" (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching
- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

14

- integer registers: prof: "compiler %rsp 64 bit, %esp 32 bit, compiler will spit out whichever is smaller and fits your data so b careful." also stuff like "%eax vs %ax vs %ah/%al"
- registers
  - eg, %rax for full 8 bytes of register, %eax right 4 bytes, %ax right 2 bytes, splitting further into %ah and %al for left and right halves of %ax.
  - see ref sheet
- memory addressing modes:  $D(\%Rb, \%Ri, S) = \text{Mem}[\%Rb + (S * \%Ri) + D]$ 
  - D: displacement of 1, 2, or 4 bytes. default: 0
  - Rb: base register (any of the 16 integer registers)
  - Ri: index register (any except %rsp).
  - S: scale of 1, 2, 4, or 8. default: 1
- lea addr dest instruction
  - sets dest to addr (eg mov instead dest to the value at that addr)
  - intended to calculate pointer to obj: eg array elem
  - compiler authors end up using it to do arithmetic
  - doesn't touch condition codes
- which registers are pointers?
  - %rsp (top of stack pointer) %rip (current instruction/program counter pointer) always pointers
  - pointers near stack pointer or program counter pointer *probably* also pointers.
  - mov (%rsi), %rsi: register used as pointer? value is probably pointer.
    - (%rsi, %rbx) one of these is a pointer, don't know which
    - (%rsi, %rbx, 2) rsi is a pointer, not rbx (why?)
    - 0x400570(, %rbx, 2) 0x is pointer, not rbx (why?) (assume blank, is 0)
    - lea (anything), %rax idk bro

## machine code: control

- control flow
  - lots of GOTOs. c0vm moment
- condition codes (status of recent tests): CF, ZF, SF, OF
  - set as side effect of arithmetic
  - Carry Flag: set if carry from unsigned overflow (or borrowing a 1 to make 0x0 - 0x1 work)
  - Zero Flag: get a 0
  - Sign Flag:  $t < 0$
  - Overflow Flag: signed overflow
  - in GDB as eflags register (a flag isn't showing up? is set to 0.)
  - compare instruction (cmp)
    - computes  $b - a$  without setting b, unlike sub
    - used for if statements
  - test instruction
    - computes  $b \& a$  (like and) without setting b
    - used to compare %rX to 0 (test %rX %rX)
    - used to check if 1-bits are same in two registers, like normal & usage
  - j... instructions: jump to different parts depending on condition codes
    - jmp, je, jne, jg, jge, etc
  - set... these correspond to j... instructions
    - sets only the low-order byte to 0 or 1 based on ...
    - usually use movzbl to alter remaining unaltered bytes.
- cmov conditional move
- for val = Test ? Then\_Expr : Else\_Expr;
  - used only when safe; both branches are computed
  - avoid bad performance, side effects, unsafety
- loops exist.
  - do while, while, for loops (beginning conditional is often optimized away)
- switch statements: jump tables
  - notice a jump to a jump table: jmpq \*0x4007f0(,%rdi,8) and a bunch of suspicious rets
  - to inspect a jump table in gdb, x /8xg 0x4007f0 (8 outputs, hex, giant aka quad word)

## machine code: procedures

- need to be able to
  - pass control to procedure and back to return
  - pass data: args, return
  - manage stack memory

- these are implemented via machine instructions, which are defined by Application Binary Interface (ABI)
- the stack
  - grows top to down, bottom address is the “top” of stack: %rsp
- first 6 args in registers, rest on stack
- note: things have alignments and sizes. same for primitives. for complex (structs, arrays), alignments are of their largest component
- note: when we see `sub $0x18, %rsp` for eg, we are “allocating” 18 bytes into the stack. or something. we will pop from stack later by adding 18 back to rsp.

### **machine code: data**

- going over arrays (1d, nd, multilevel), structs (alloc, access, alignment), floats
- arrays
  - contiguous region of `length * sizeof(type)` bytes
  - therefore, nd arrays get row-major ordered
- structs
  - fields ordered to declaration
  - each element must satisfy its own alignment requirement
  - entire struct (initial addr and its size) must be aligned to largest element’s size
  - tldr just save space by larger data types first
- alignment restrictions
  - 1 byte: `char`, ...
    - no restrictions on address
  - 2 bytes: `short`, ...
    - lowest 1 bit of address must be  $0_2$
  - 4 bytes: `int`, `float`, ...
    - lowest 2 bits of address must be  $00_2$
  - 8 bytes: `double`, `long`, `char *`, ...
    - lowest 3 bits of address must be  $000_2$
- floating point
  - args passed in `%xmm0`, `%xmm1`, ...
  - result in `%xmm0`
  - all XMM registers call-clobbered

### **machine code: advanced**

- Memory Layout: see the pdf
- Buffer Overflow
  - Vulnerability
  - Protection
    - Avoid overflow vulnerabilities (eg validate: we know we need max n chars)
    - Employ system-level protections: randomized stack offsets, nonexecutable stack
    - Have compiler use “stack canaries”: right after buffer on stack; check for corruption

- Bypassing Protection: Return Oriented Programming attacks with gadgets
- Unions

## the memory hierarchy

- “Memory Wall” or Von Neumann bottleneck: performance gap between cpu computation and ram data storage
- Three approaches to work around it:
  - **Build a hierarchy** (covered in here in 213)
  - Find other stuff to do (346, 418)
  - Move computation (346, 7xx?)

toc:

- The memory abstraction
- RAM : main memory building block
- Locality of reference
- The memory hierarchy
- The memory mountain
- Storage technologies and trends

## the memory abstraction

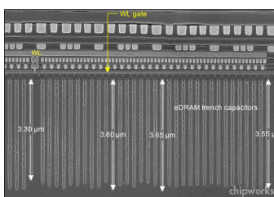
- bus: a collection of parallel wires that carry address, data, and control signals.
- buses are typically shared by multiple devices.
- buses do the literal running of info, eg, from cpu regs to ram

## RAM

- static (SRAM)
- dynamic (DRAM)

## RAM Technologies

### ■ DRAM

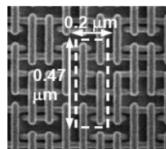


### ■ 1 Transistor + 1 capacitor / bit

- Capacitor oriented vertically

### ■ Must refresh state periodically

### ■ SRAM



### ■ 6 transistors / bit

### ■ Holds state indefinitely (but will still lose data on power loss)

## SRAM vs DRAM Summary

	Trans. per bit	Access time	Needs refresh?	Needs EDC?	Cost	Applications
SRAM	6 or 8	1x	No	Maybe	100x	Cache memories
DRAM	1	10x	Yes	Yes	1x	Main memories, frame buffers

EDC: Error detection and correction

### ■ Trends

- SRAM scales with semiconductor technology
  - Reaching its limits
- DRAM scaling limited by need for minimum capacitance
  - Aspect ratio limits how deep can make capacitor
  - Also reaching its limits

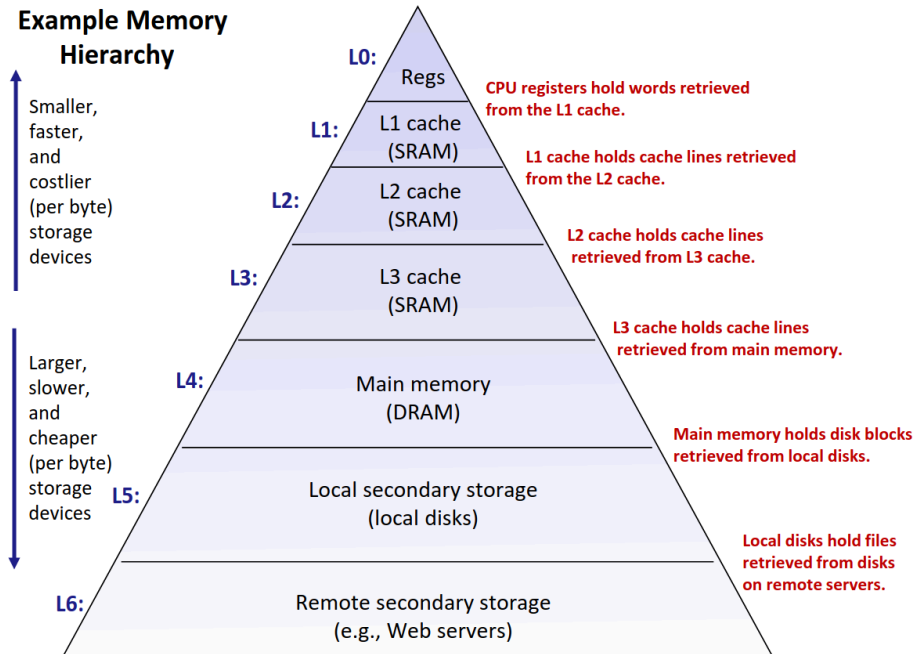
### locality

- a key to bridging CPU-memory gap
- principle of locality: progs use addresses near previously used addresses (data, instructions)
- temporal locality: same items likely to be referenced again
- spatial locality: nearby addresses likely to be referenced together
- fun fact: can have  $\sum 3d$  array with better spatial locality by making  $j$  the inner loop

### memory hierarchy

- properties of hard/software:
  - faster = more expensive, hotter, less capacity
  - CPU-memory gap widening
  - good programs usually have good locality
- this together implies a memory hierarchy **that gives the illusion of both large and fast memory. as cheap as the large memory, but fast as the memory at the top of the ladder:**

## Example Memory Hierarchy



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

34

## Examples of Caching in the Mem. Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 byte words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

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

43

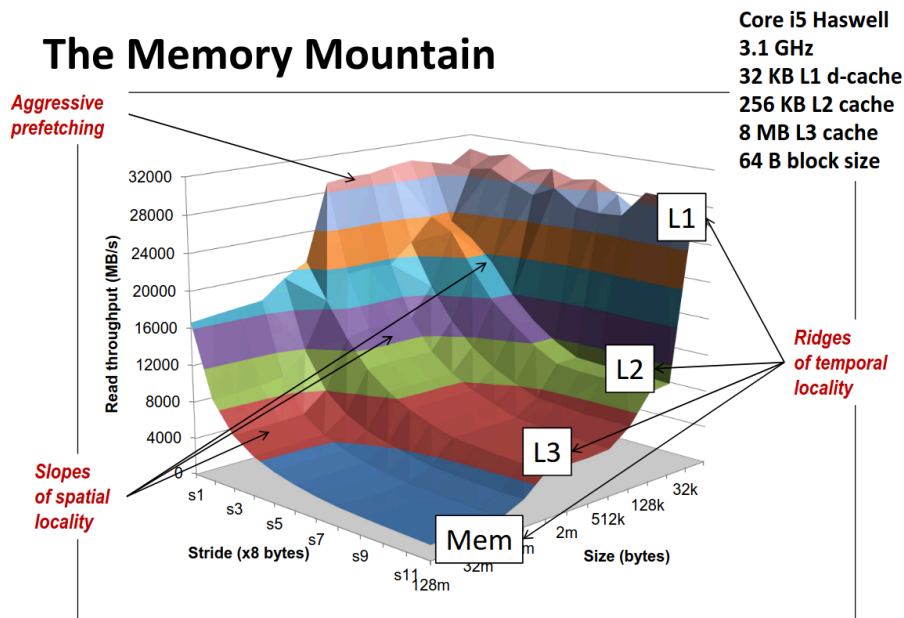
- caches are hardware controlled (invisible to software!), memory is software controlled
- transfer unit sizes are called “blocks”
- cache hit: data is needed from a cache **and it is there!**
- cache miss: data is needed from a cache **and it is NOT there!**
  - the block is fetched from memory
  - the block is stored in cache:
  - placement policy: where does new block go?



- replacement policy: what gets kicked out? (the victim)
- types:
  - cold (compulsory) miss: first reference to block
  - capacity miss: # of active cache blocks (size of “working set”) > size of cache
  - conflict miss: like a hash collision

### memory mountain

- read throughput/bandwidth: bytes read from memory per second
- memory mountain: read bandwidth as a function of spatial and temporal locality
  - characterizes memory system performance



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

### storage technologies and trends

TODO lol