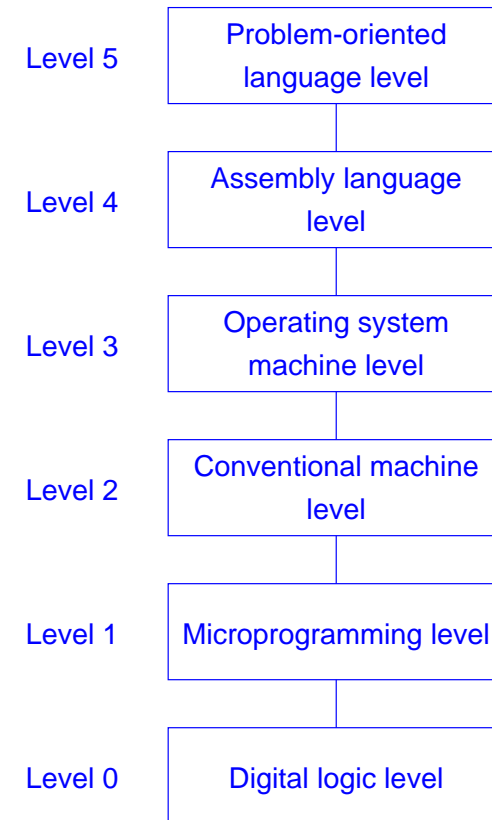


# Modern Computer Organization: Abstractions at Several Levels

- levels of computer architecture
- the Java Virtual Machine (level 5 - mainly)
- memory hierarchy and caches (level 1)
  - issues, and details
- the digital logic level (level 0)

# Levels of Computer Architecture

- Level 5: Problem solving language (e.g. Java, C?)
  - compilation
- Level 4: Assembler
  - translation
- Level 3: Operating system machine level
  - partial interpretation
- Level 2: Instruction set architecture
  - interpretation (microprogram) or direct execution
  - e.g. registers, (virtual) memory access
- Level 1: Microarchitecture level
  - hardware
  - e.g. datapaths, buses, 'hidden' registers (CI, MAR)
- Level 0: Digital logic level
  - e.g. circuits (wires and gates)



alt. [Null&Lobur, fig 1.3]  
– note the extra level!

## High-Level Language Level: the Java Virtual Machine (JVM)

Ref: [Null&Lobur, sect 3.5]

Fig 8.11, Fig 8.13

- JVM is a stack machine interpreter for Java bytecode (`.class` files)
- individual bytecode instructions are represented by numbers stored in a single byte
  - the Java code sequence `i = i + 1;` (where `int i;`) might translate into the bytecode sequence:

mnemonic	bytecode	operation
<code>iload_0</code>	1A	push local variable 0 ( <code>i</code> ) onto stack
<code>iconst_1</code>	04	push constant 0 onto stack
<code>iadd</code>	60	push variable <code>n</code> onto the stack
<code>istore_0</code>	3A	pop stack and store in local variable 0

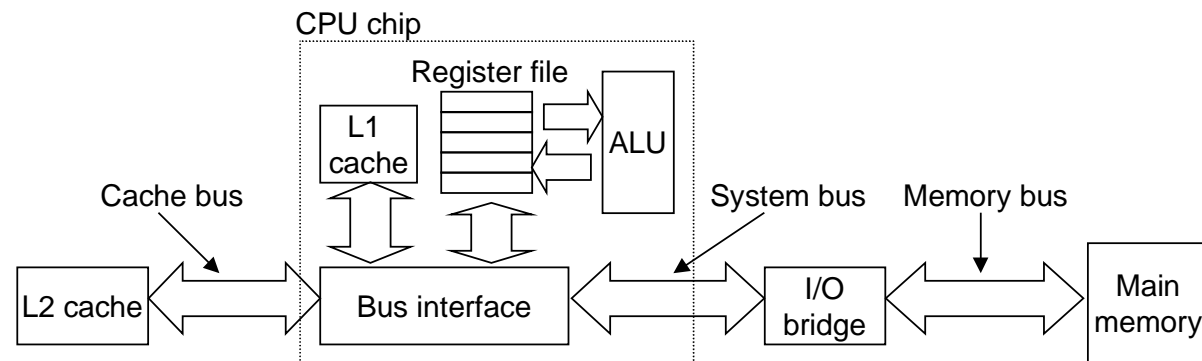
- a JVM *interprets* these, and can produce the same effects on many different physical computers
  - this stage corresponds to Level 2, but on a *virtual* (software) level
- OS level is abstracted via I/O-related methods (accessed via the JVM native methods, which accesses the C system calls)
- Q: why is it useful to design Java this way? why use a stack machine?

# The Microarchitectural Level Example: the Memory Hierarchy

Ref: [Null&Lobur, sect 6.3-6.4], [O'H&Bryant, sect 6.2-6.4]

- in a computer system, there is a memory hierarchy, because:
  - many different mediums for the storage of data
  - generally, there is a *trade-off* between speed and capacity
  - ◆ fast memories tend to be small; large memories tend to be slow

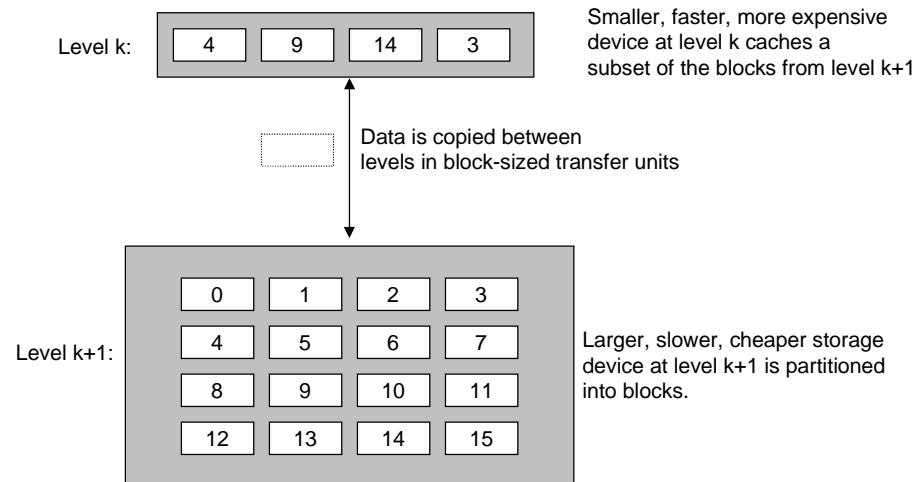
medium	access time	typical size
registers	~1 ns	< 1 KB
cache mem.	~ 20 ns	< 2 MB
main mem.	~ 100 ns	< 2 GB
disk	~ 10 <sup>7</sup> ns	> 10 GB



[O'H&Bryant, fig 6.24]

- idea of cache memory: store recently accessed data from main memory in a small, faster memory (closer to / inside the CPU)
  - Q: why might this be useful?

## Cache Memory: Issues



- [O'H&Bryant, fig 6.22]
- issue: cache must store the memory address (as a 'tag') of the data, as well as the data itself!
- issue: (considering the cache to be organized as an array) which cache entry ('index') holds the data for address  $X$  (if any)?
- idea (direct-mapped cache: for a cache of size  $C' = 2^c$  entries, all addresses with same  $a_1$  are mapped to the same entry

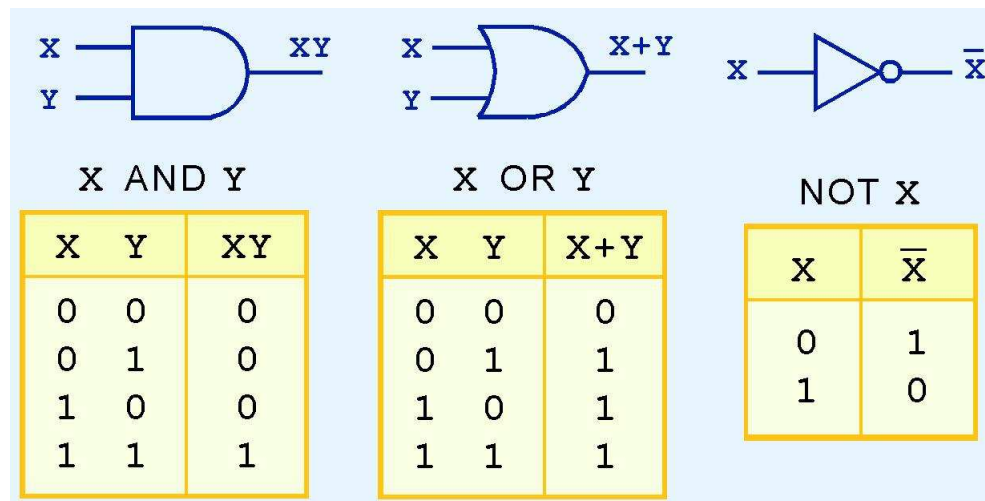
$$X = \begin{array}{|c|c|} \hline 31 & c-1 \\ \hline a_0 & a_1 \\ \hline \end{array}$$

# Cache Memory: Details: Associativity and Replacement Policy

- cache hit rates: % of (word) accesses in program when data is in cache
    - needs to be **high** (e.g. > 95%) for good performance!
  - problem: the data for addresses with the same  $a_1$  cannot be in the cache simultaneously!
  - solution (K-way set associative caches): each entry can hold a set of  $K$  data items (blocks) (typically  $K = 1, 2, 4, 8$ )
    - addresses with the same  $a_1$  can **map** into any of the  $K$  items in the set
  - issue (replacement policy): if all items in the set contain data, what happens when we access a new address  $X'$  with the same  $a_1$ ?
    - random: choose any item, and put data and tag for  $X'$  there
    - least recently used: choose the item that was *least recently accessed* to replace
- Q: which is likely to be better? any downsides?
- cache behavior (hit rates are not always simple to analyze!)
    - the Random Cache\$ for Beer Challenge!

## The Digital Logic Level: Gates

- Ref: [Null&Lobur, sect 2.3–2.4], [Tanenbaum, ch 3]
- digital logic is lowest level of computer organization!
- digital circuit elements (including 1-bit memory cells) are made from gates  
(in turn, from 1 or more transistors or switches, [Tanenbaum, fig 3.1])
- [Null&Lobur, figs 3.1-2]: AND and OR gates manipulate voltage levels (0- low, 1 - high) according to truth tables (Boolean logic)

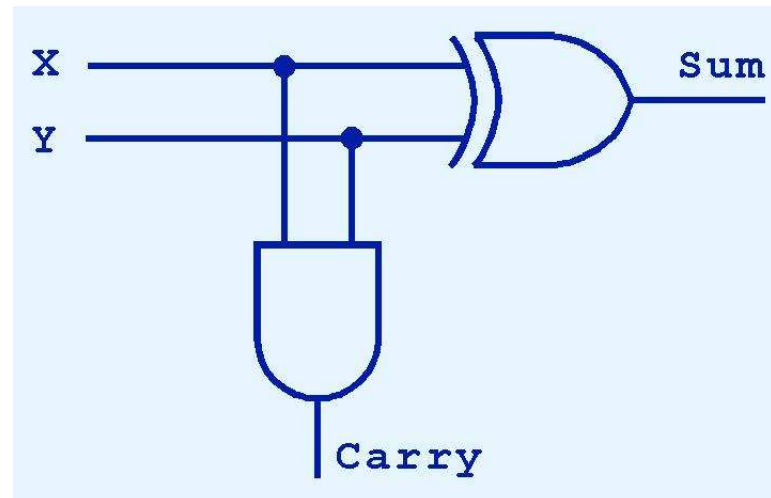


- [O'H&Bryant, p 45]: set of colors forming an 8-element boolean algebra

## The Digital Logic Level: Half Adders

- from gates, higher-level components such as a half-adder are constructed ([Null&Lobur, figs 3.10,3.11]:  $\text{Sum} = X \text{ XOR } Y$ ,  $\text{Carry} = X \text{ AND } Y$ )

Inputs		Outputs	
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



- this can be extended to a full adder ([Null&Lobur, fig 3.13])  
(truth table, circuit)
  - how could this be used to implement an integer add circuit?
- discussion point: why is binary the representation used in computers?