

# On the Future of Computer Science and Engineering

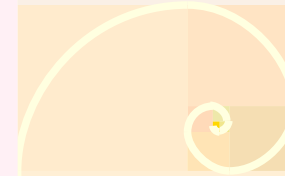
Dr. Pierre Vigneras

[pierre@giki.edu.pk](mailto:pierre@giki.edu.pk)

Assistant Professor

Faculty of Computer Science & Engineering,

GIKI



# Parallel Worlds

- Science

- Models
- Languages
- Complexity
- Computability, ...

Edsger Dijkstra: "Computer science is no more about computers than astronomy is about telescopes."

**Using a computer does not make one a computer scientist!**

- Engineering

- Hardware
  - Processors, Hard drives, Memory, ...
- Software
  - Design, Operating Systems, Project Management, ...

**Please don't ask me how to solve your Internet Explorer Problem! ;-)**

# Outline

- The Past
  - Hardware models
  - Computation models
  - Programming Models
- The Present
  - The Chip Multi-threading hardware trend
  - Influence on Computer Science & Engineering
- The Future
  - Heterogeneous or Homogeneous?
  - Operating Systems or Virtual Machines

# The Past

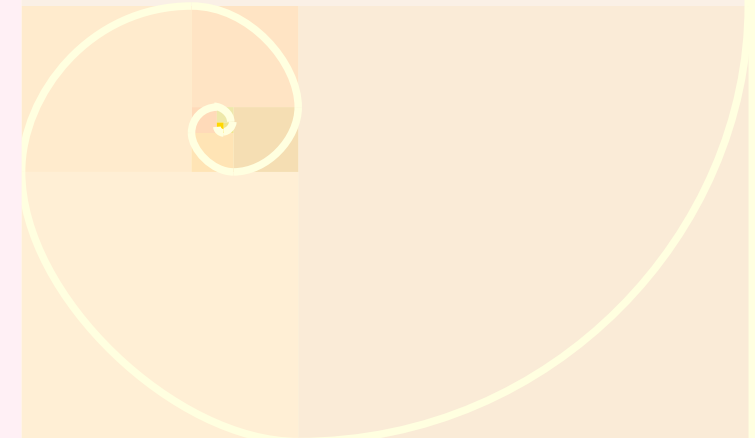
# Hardware Model

# Counting

- From the very beginning, men started off by counting on their digits for various reasons
  - Sharing food items fairly
  - Religion & ceremonies according to time
  - etc.
- Many possibilities (base): 10, 16, 12, 6, ...
- Some attempts to create counting machines
  - Help counting up to big numbers
  - Abacus (China), Pascaline (Blaise Pascal, 1642), The Difference Engine (Charles Babbage, 1812)

# Counting Machine Evolution

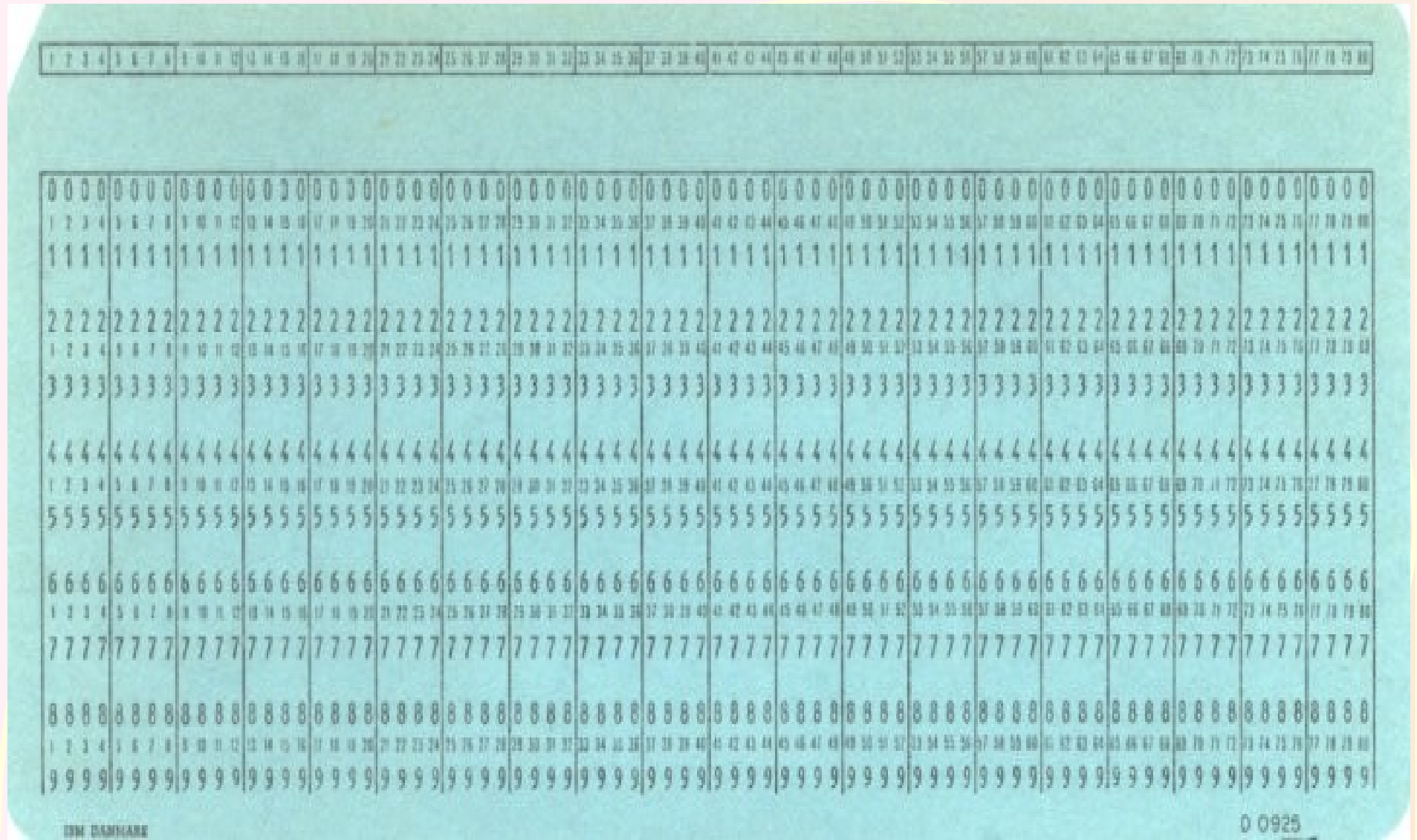
- 1890: Herman Hollerith presents
  - Punched Cards & Tabulating Machine
  - For U.S. Census Bureau
  - Used until the highly controversial United States presidential election of 2000
  - Tabulating Machine is a great success: creation of IBM by Herman Hollerith
- Limited to tabulation



# Cards

([http://en.wikipedia.org/wiki/Punch\\_card](http://en.wikipedia.org/wiki/Punch_card))

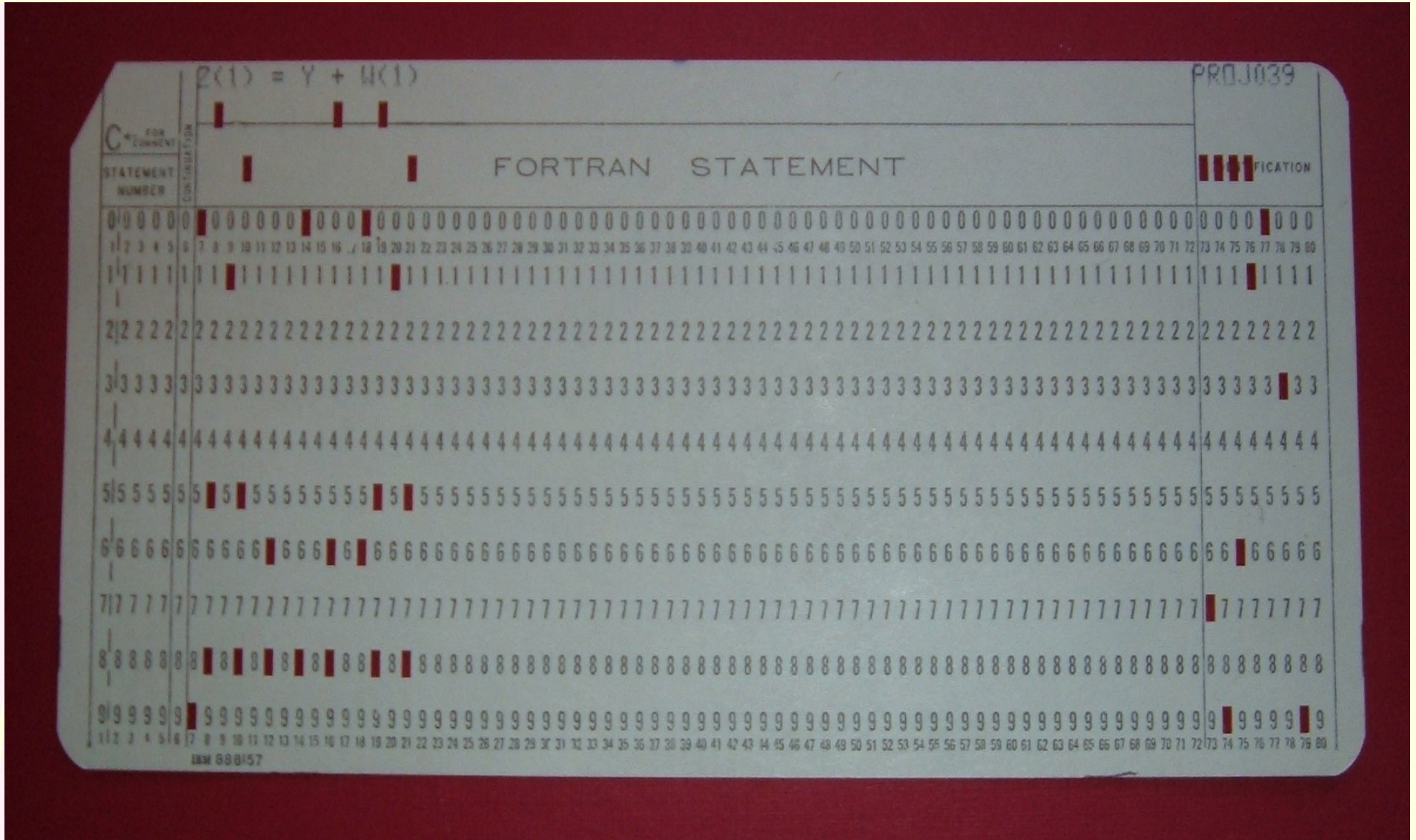
## II. History/Hardware



# Cards

([http://en.wikipedia.org/wiki/Punch\\_card](http://en.wikipedia.org/wiki/Punch_card))

## II. History/Hardware

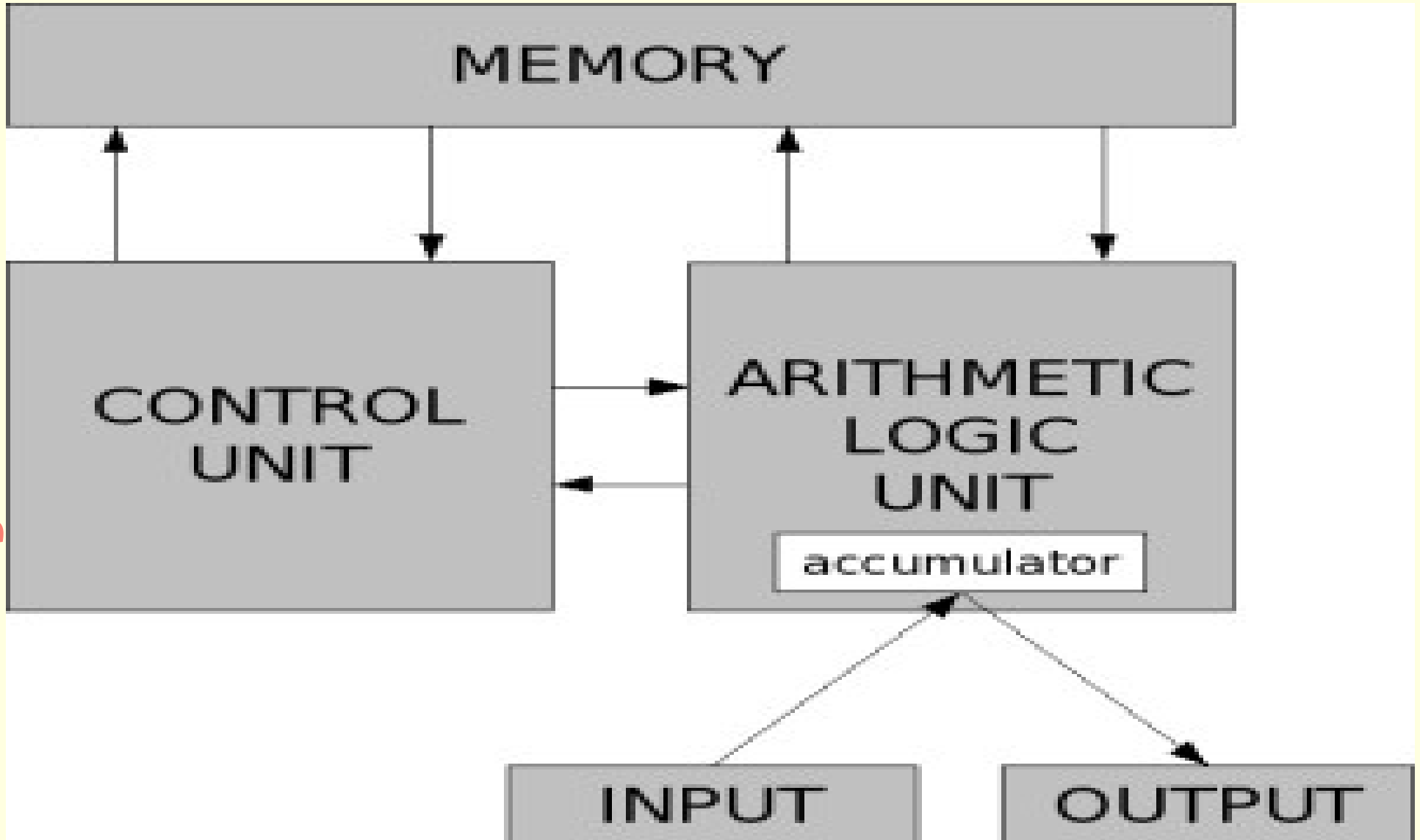


# John Von Neumann

([http://en.wikipedia.org/wiki/Von\\_Neumann\\_Architecture](http://en.wikipedia.org/wiki/Von_Neumann_Architecture))

- Data and Program can be stored in the same space
  - The machine itself can alter either its program or its internal data
- Conditional goto's to other point in the code
- Often used subroutines can be stored in memory
- First machine appears in 1947 (EDVAC & UNIVAC)
  - Still the overall architecture of computers today!

# Von Neumann Architecture



II. History/Hardware

# Major engineering advances

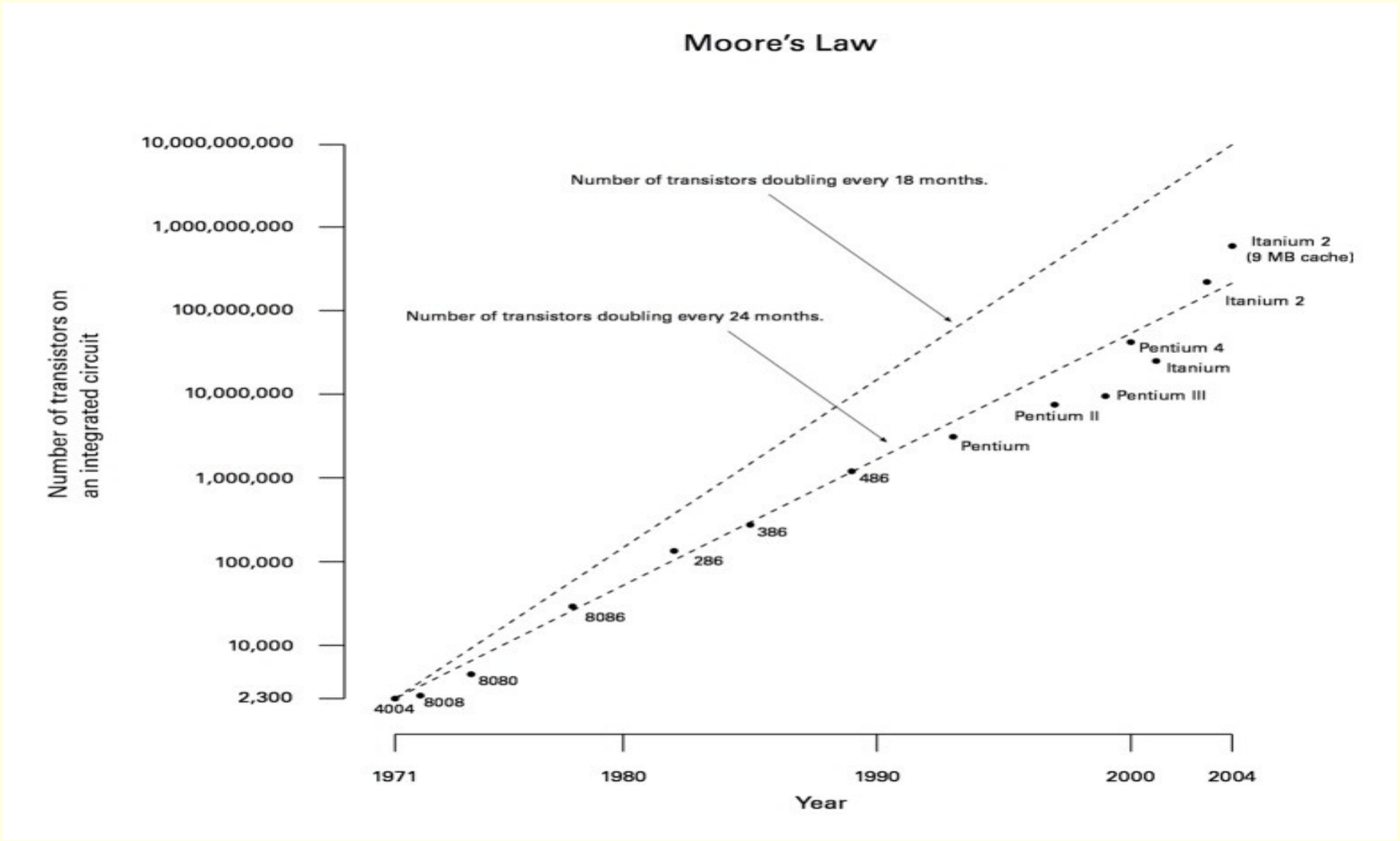
- Transistors (Shockley, Bardeen, Walter; 1947)
  - Freedom from vacuum tubes, which were bulky
- Integrated Circuits (Kilby, 1959)
  - Aka “chip”
  - collection of tiny transistors which are connected together
  - only connections were needed to other electronic components
- Machines becomes smaller and more economical to build and maintain.

# Moore's Law

([http://en.wikipedia.org/wiki/Moore's\\_law](http://en.wikipedia.org/wiki/Moore's_law))

- 1965: Moore, a co-founder of Intel.
  - *The complexity of integrated circuits doubles every 24 months*
  - Quoted as “[...] doubles every **18 months**”!
  - Empirical observations and prediction
  - Goal for an entire industry.
- Moore's law means an average performance improvement in the industry as a whole of over 1% a week.
  - A new product expected to take three years to develop and just two months late is 10 to 15% slower, bulkier, or lower in storage capacity!

# Moore's law



# Moore's Law effect

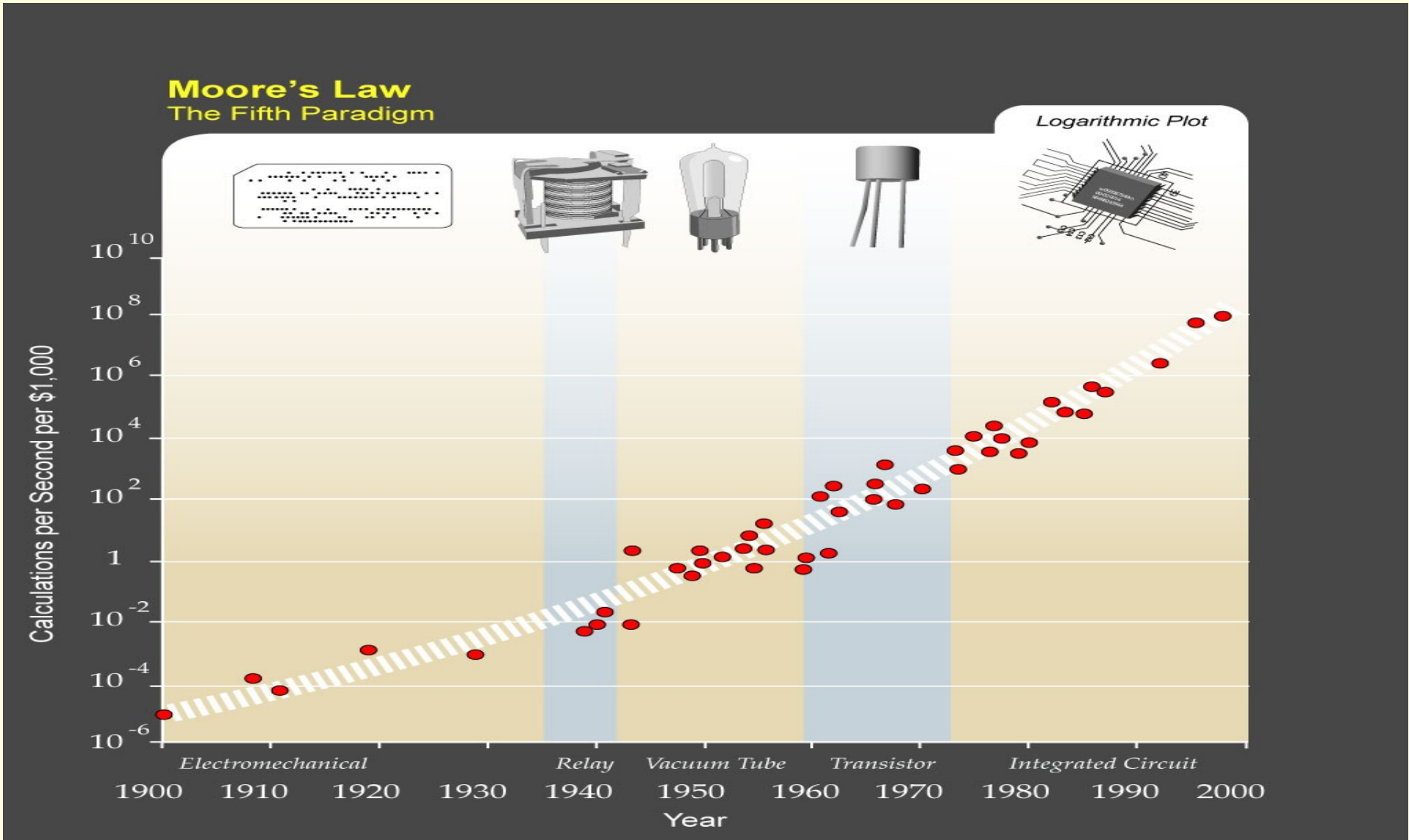


# Moore's Law limitation

- 2006 (IBM)
  - Announce 30 nm technology
- April 2005 (Gordon Moore)
  - transistors may reach the limits of atomic levels
  - the law may not hold valid for too long
- 2003 (Kurzweil)
  - Moore's Law of Integrated Circuits was not the first, but the fifth paradigm to provide accelerating price-performance.
  - New type of technology will replace current integrated-circuit technology, and that Moore's Law will hold true long after 2020.

# General Moore's Law

## II. History/Hardware



# 2007 Trend

([http://en.wikipedia.org/wiki/Multicore\\_CPU](http://en.wikipedia.org/wiki/Multicore_CPU))

- Major misunderstanding:
  - Performance is not equivalent to clock speed !
    - Performance= Instructions Per Clock tick \* Clock Speed
    - Big IPC: better performance for same clock speed (less heat)
      - AMD, Sun, IBM processors have an IPC > Intel (on average)
  - Moore's law is not about doubling performance anyway!
- Doubling the number of transistors:
  - Put two CPUs on the same silicon die: dual-core
- The future is multi-core systems
  - Intel, AMD, IBM, Sun are focusing on this
  - No need for a new CPU design: less risk
- Consequences?

# Computation Model

# Alan Turing Machine (1936)

([http://en.wikipedia.org/wiki/Turing\\_machine](http://en.wikipedia.org/wiki/Turing_machine))

- Infinite tape (divided into adjacent cells)
  - Each cell contains a symbol from some finite alphabet:  $\{a, \dots, z\}$ ;  $\{0, \dots, 9\}$ ;  $\{0, 1\}$
- Head
  - read/write symbols and move the tape left and right one cell at a time.
- Table of instructions that tells the machine:
  - what symbol to write, how to move the head and what its new state will be
- State register that stores the (finite) state of the table.
  - One special start state

# Turing Machine

([http://en.wikipedia.org/wiki/Turing\\_machine](http://en.wikipedia.org/wiki/Turing_machine))

- Example: increment function:  $\text{inc}(x) = x + 1$

- Tape alphabet:  $\{0, 1\}$

- Input: A number in base one enclosed by zero

$0_{10} : 0 \dots 0 \mathbf{1} 0 \dots 0$

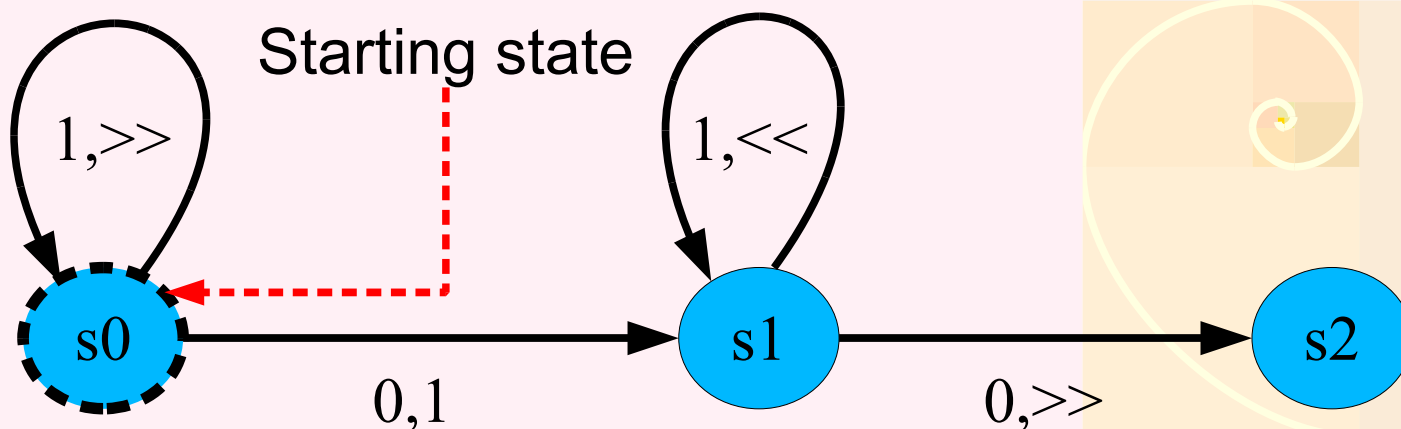
Starting points

$1_{10} : 0 \dots 0 \mathbf{1} 1 0 \dots 0$

$4_{10} : 0 \dots 0 \mathbf{1} 1 1 1 0 \dots 0$

$10_{10} : 0 \dots 0 \mathbf{1} 1 1 1 1 1 1 1 1 1 1 0 \dots 0$

- Output: written in place of the input (same form)



II. History/Theory

# Turing Machines and Algorithms

- A Turing Machine describes a “*mechanical procedure*”
  - Sort of **recipe**
  - Each step is **simple** and **well defined** so it can be followed by any human using a paper and a pencil
- Very complex procedures can be described using Turing machines
  - Addition, multiplication, ... (anything?)
- Turing Machines are the actual formalisation of what is called **algorithms**

# The Halting Problem (Turing, 1936)

*Given a TM, say  $t$ , and its initial input, say  $n$ , can we construct another TM that determines whether  $t$  halts on ' $n$ ' (the alternative is that  $t$  runs forever).*

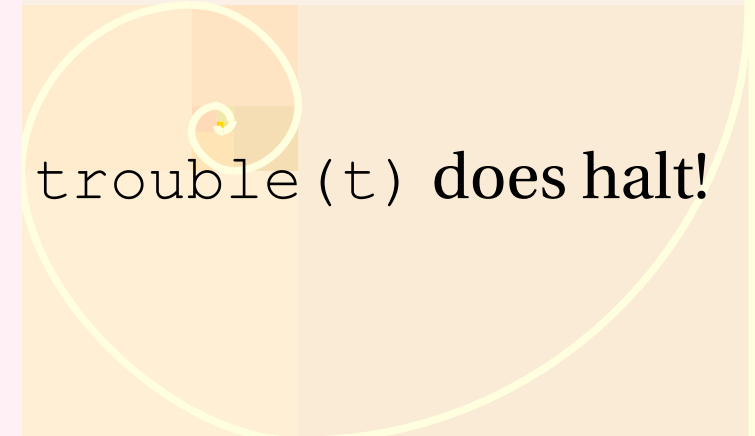
- `halt(t, n) == true` if  $t$  halts on  $n$ , `false` otherwise.
- Suppose that such a TM machine does exist.
- Consider:

String representing an algorithm

```
boolean trouble(s) {  
    if (halt(s, s) == true) {  
        loop forever; // Implementation?  
    }  
    return true;  
}
```

# Halting Problem (Turing, 1936)

- `trouble()` takes the string 's' (representing an algorithm) and gives it to the (supposed existing) function `halt()` both as the algorithm to check and as its initial input.
- Consider the string 't' that represents the algorithm `trouble()`
- If `trouble(t)` halts (it returns `true`)
  - `halt(t,t)` returns `false`, hence `trouble(t)` does not halt!
- If `trouble(t)` does not halt
  - `halt(t,t)` returns `true`, hence `trouble(t)` does halt!



# Halting Problem (Turing, 1936)

- The Halting Problem is **undecidable** in the Turing system
  - And also in any other computation model that is equivalent (Markov algorithms, Lambda calculus, ...)
- Many other problems are **undecidable**
  - **Informally, such problems cannot be solved in general by computers**
- Related to Godel's Theorem (1931)
  - Any theory is either:
    - incomplete (some truth cannot be proven)
    - inconsistent (can prove one statement and its contrary)

# Church's Lambda Calculus (1936)

([http://en.wikipedia.org/wiki/Lambda\\_calculus](http://en.wikipedia.org/wiki/Lambda_calculus))

- Formal system designed to investigate function definition, function application, and recursion.
- Every expression stands for a function with a single argument

-  $f(x) = x + 2 \dashrightarrow \lambda x. x+2$

Returns a *number*

$f(1)=3 \dashrightarrow (\lambda x. x+2) 1=1+2=3$

-  $g(x,y)=x-y \dashrightarrow \lambda x. (\lambda y. x-y)=\lambda xy. x-y;$

Returns a *function*

$g(3,2)=1 \dashrightarrow (\lambda xy. x-y) 3 2 = (\lambda y. 3-y) 2 = 3-2 = 1$

-  $g(f(1),f(0)) = f(1)-f(0) = (1+2)-(0+2) = 3-2 = 5$

$\dashrightarrow (\lambda xy. x-y) ((\lambda x. x+2) 1) ((\lambda x. x+2) 0)$

# Church's Lambda Calculus (1936)

([http://en.wikipedia.org/wiki/Lambda\\_calculus](http://en.wikipedia.org/wiki/Lambda_calculus))

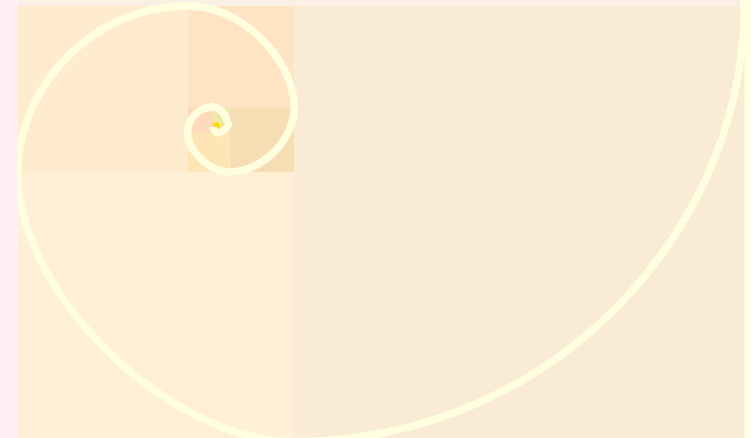
- $(\lambda xy. x - y) 7 2$ ;  $(\lambda y. 7 - y) 2$  and  $7 - 2$  are equivalent
- This equivalence of lambda expressions can not be *decided* by an *algorithm* in general
- Consider:  $(\lambda x. x x) (\lambda x. x x) \rightarrow$  No reduction!
- Logic and predicates

TRUE :=  $\lambda xy. x$ , FALSE :=  $\lambda xy. y$

AND :=  $\lambda pq. p q$  FALSE, OR :=  $\lambda pq. p$  TRUE  $q$

NOT :=  $\lambda p. p$  FALSE TRUE,

IFTHENELSE :=  $\lambda pxy. p x y$



# Church-Turing Thesis

([http://en.wikipedia.org/wiki/Church-Turing\\_thesis](http://en.wikipedia.org/wiki/Church-Turing_thesis))

- hypothesis about the nature of computers
  - digital computer
  - human with a pencil and a paper following a set of rules.
- “Any calculation that is possible can be performed by an algorithm running on a computer, provided that sufficient time and storage space are available”
  - may be regarded as a physical law or as a definition, as it has not been mathematically proven.

# Programming Model

# Computation Models are Basis of Programming Languages

- Turing Machine leads to usual representation of algorithms
  - expressed in an iterative way, in sequences of well defined steps
- $\lambda$ -calculus leads to another way of expressing algorithms
  - expressed using *only functions*
  - Use *recursion* extensively
- Same expressive power than Turing Machine
  - Proven!
- Many other models...

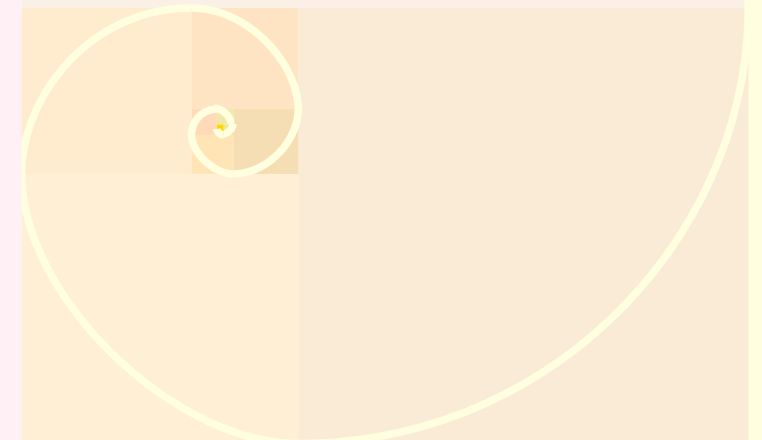
# Programming Paradigms

([http://en.wikipedia.org/wiki/Programming\\_paradigm](http://en.wikipedia.org/wiki/Programming_paradigm))

- Logic Programming
  - Define assertions to find a goal
  - Based (roughly) on the Boolean Algebra
- Imperative languages
  - List of statements that change a program state
  - Based on the Turing Machine model
- Functional languages
  - Sequence of stateless function evaluations
  - Based on the Church  $\lambda$ -calculus

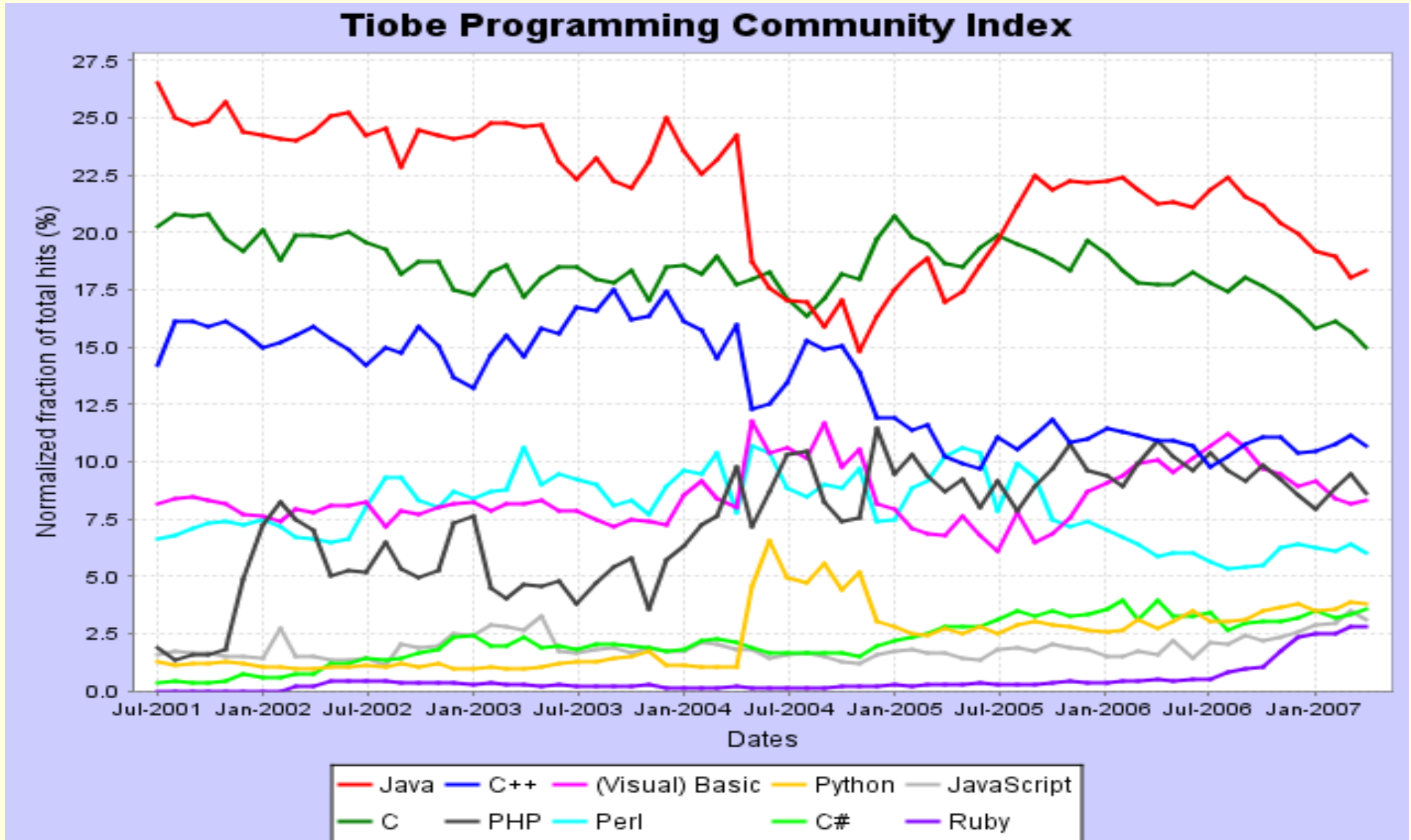
# Language Trend

- Mostly used languages today are based on either Turing Machines or Lambda Calculus.
  - C, C++, Java, C#: imperative
  - Ruby, Python, Perl, PHP: imperative
  - LISP, Common LISP: functional
  - Prolog: logical



# Language Trend

(<http://www.tiobe.com/tpci.htm>)

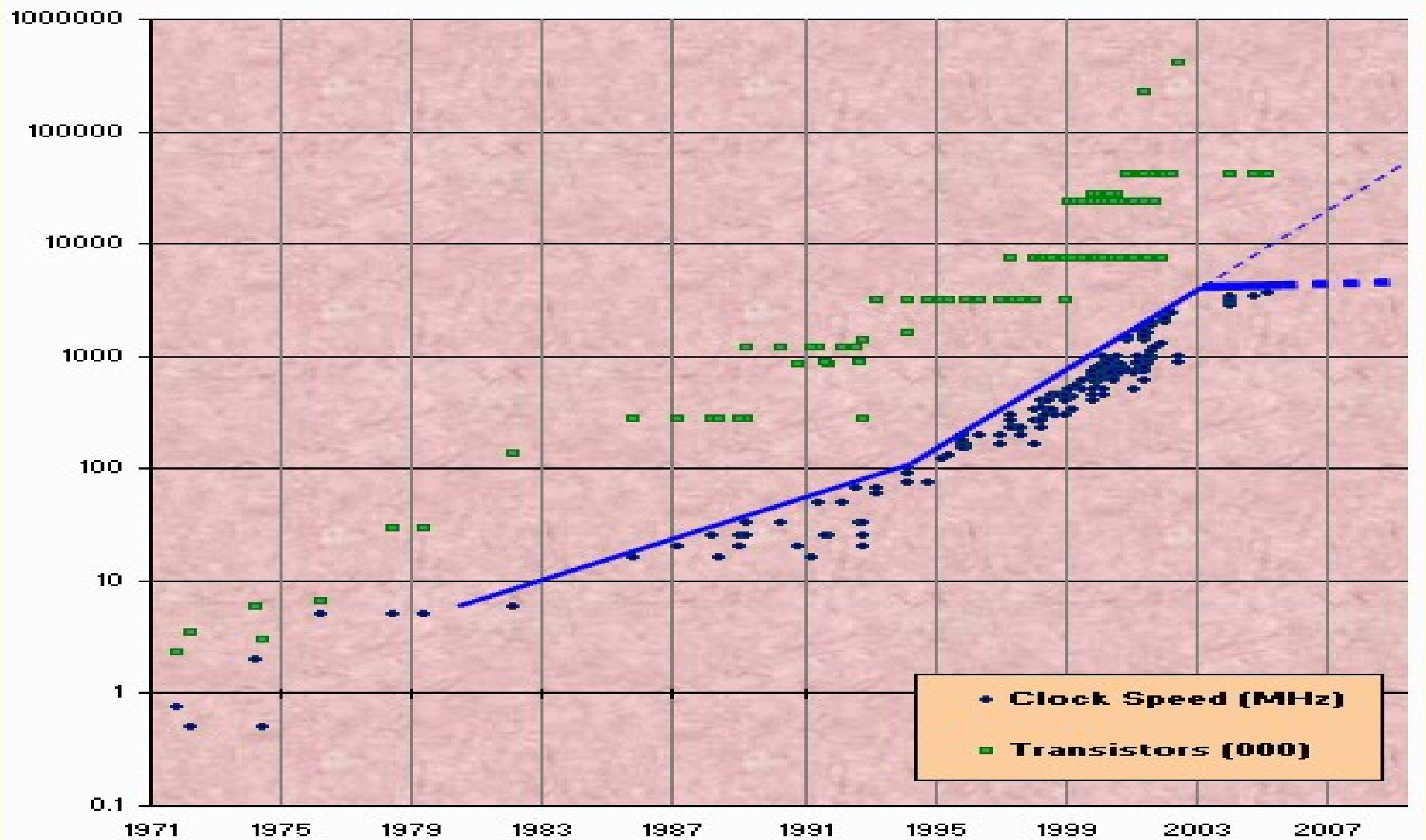


# **The Present**

# The Chip Multi-threading Trend

II. History

# Facts



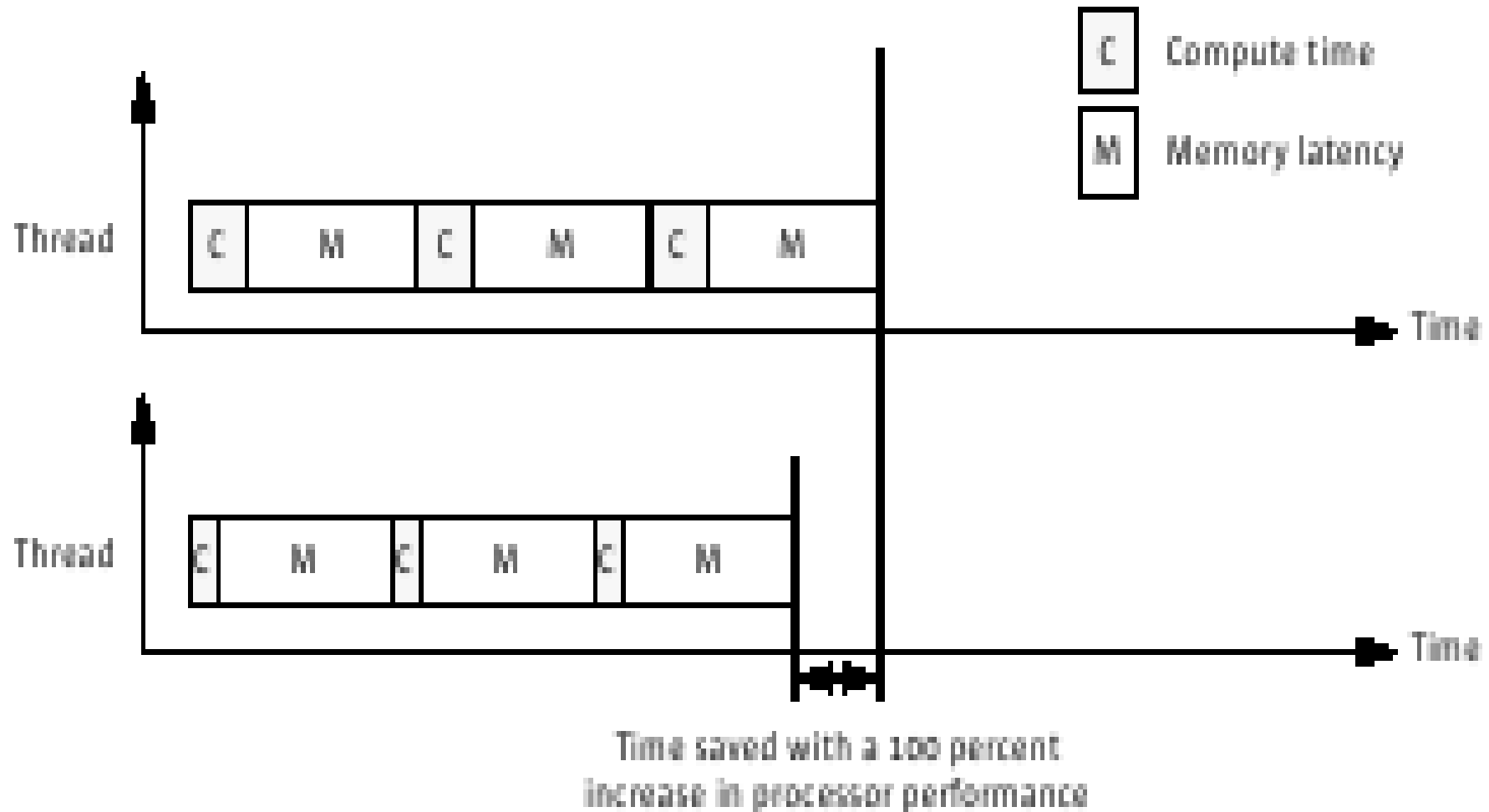
# Facts

- 2000: Intel provides the Pentium IV and bets on high clock speed
  - promise a 10 Ghz processor for 2005
- 2004: Intel is not able to pass beyond the 4Ghz limit due to heat problems
  - 2005: IBM, Sony and Thoshiba provide the Cell processor clocked at 4Ghz for the PlayStation 3
- Major breakdown
  - Intel revert back to the Pentium-m (based on a P3) and cell them in pair (Core Duo = 2 P-m roughly)
  - 2006: end of Pentium IV line!

# The Hardware Engineering Limit

- Von-Neuman architecture has reached a limit
  - Lots of engineering to make new processors more efficient than last ones
    - super-scalar, pre-fetching, out-of-order execution, branch-prediction
  - Needs a lot of transistors
    - fine, this is the Moore's law
- But small improvement (<10 %)
  - At same clock speed and with same cache P4~P3
  - Partly due to limit of the hardware model
  - Partly due to gap between memory and processor speed

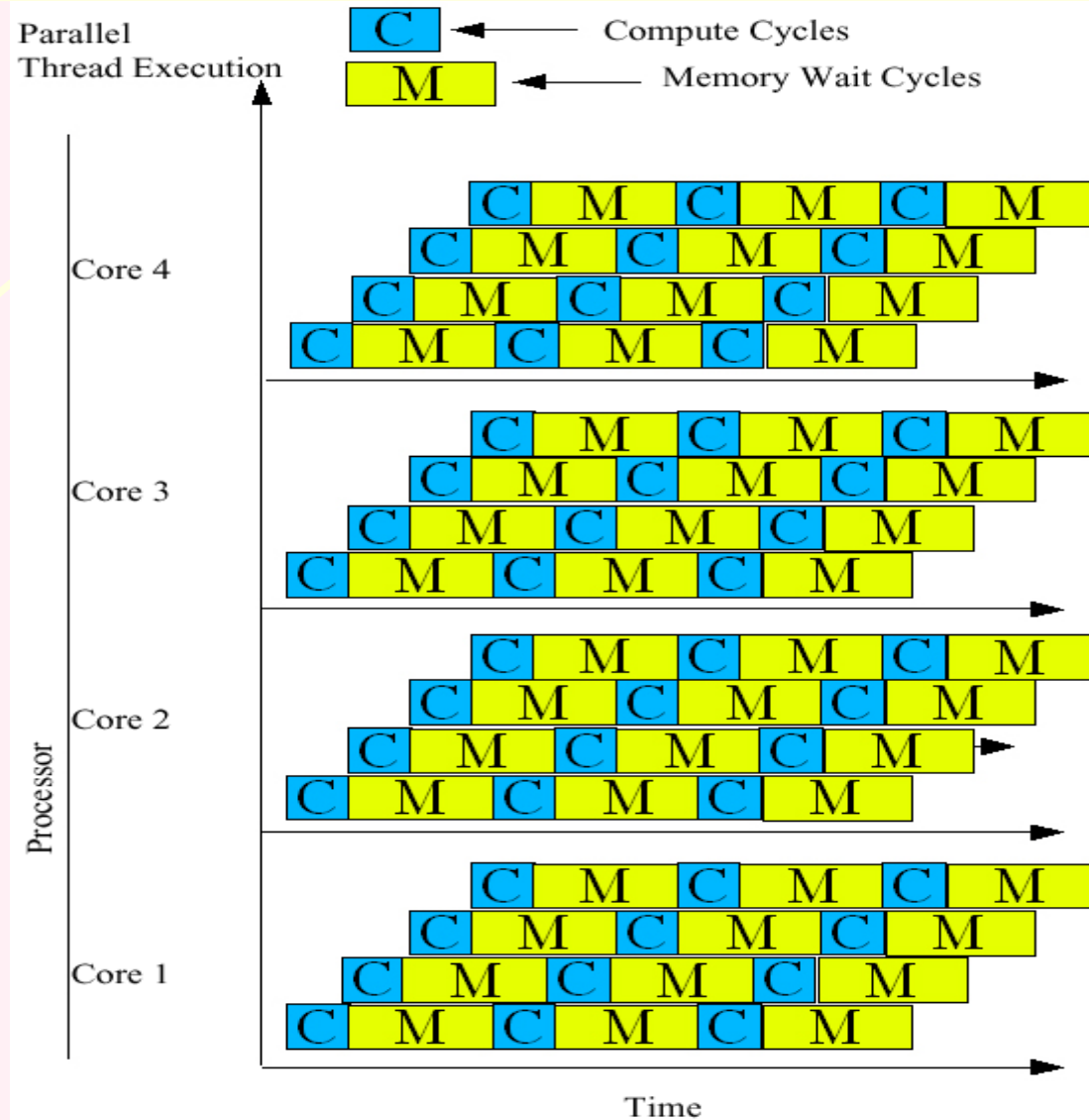
# Memory Bottleneck



# Solution: Multi-Core (CMP)

- Embedding multiple core on the same die:
  - is economic: no need for new design, no risk.
  - it follows Moore's law: doubling #transistors/die
- Engineering Improvement
  - Hardware Multi-threading (e.g.: Intel HT)
    - several flows of execution are executed in a single core
- Examples:
  - IBM Power 5: 2 core \* 2 threads = 4 ways processor
  - Sun T1: 8 cores \* 4 threads = 32 ways processor
- CMT encompasses any combination of CMP/HMT

# Chip Multi-Threading



# Influence on CS&E

# On the success of bad programming

- During ~ 3 decades, the doubling in processor performance/24 months has allowed programmers to write naïve algorithms
  - “Application is slow!”
    - Choice 1: ask an expert to improve your application
    - Choice 2: use the slow application right now; in 24 months you will be able to purchase a new processor that will double its speed! **Much cheaper!**
- This is no more valid with new processors!
  - Applications have to be concurrent!
  - How to get speedup from concurrency on new processors?

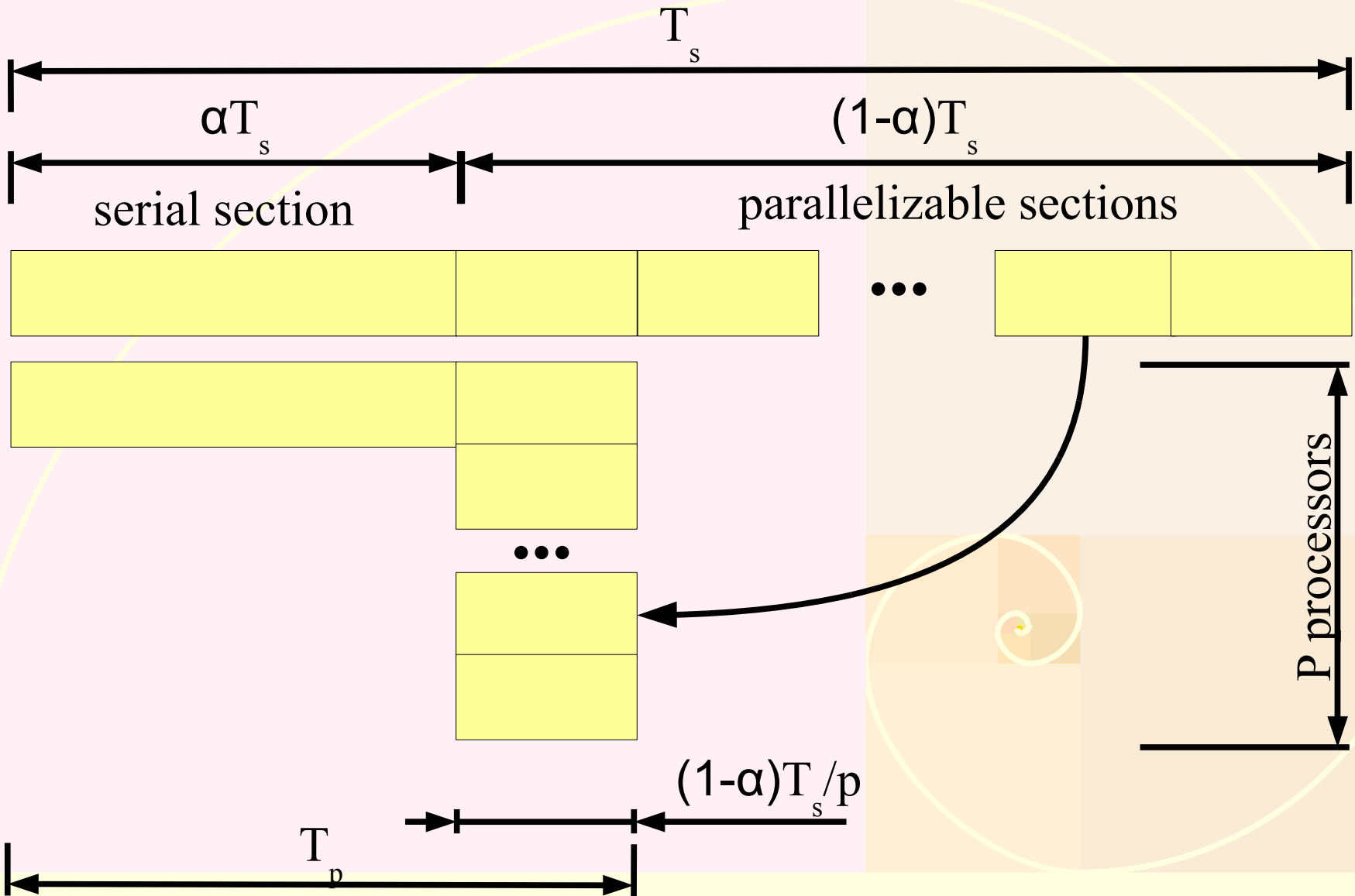
# Parallel Speedup

- For a given algorithm, let  $T_p$  be the time to perform the computation using  $p$  processors
- How much time we gain by using a parallel algorithm?
  - Speedup:  $S_p = T_1 / T_p$
  - Issue: what is  $T_1$ ?
    - Time taken by the execution of the **best sequential algorithm** on the same input data
    - We note  $T_1 = T_s$

# Speedup consequence

- Best Parallel Time Achievable is:  $T_s / p$
- Best Speedup is bounded by  $S_p \leq T_s / (T_s / p) = p$ 
  - Called Linear Speedup
- Conditions
  - Workload can be divided into  $p$  equal parts
  - No overhead at all
- Ideal situation, usually impossible to achieve!
- Objective of a parallel algorithm
  - Being as close to a linear speedup as possible

# Amdhal's Law

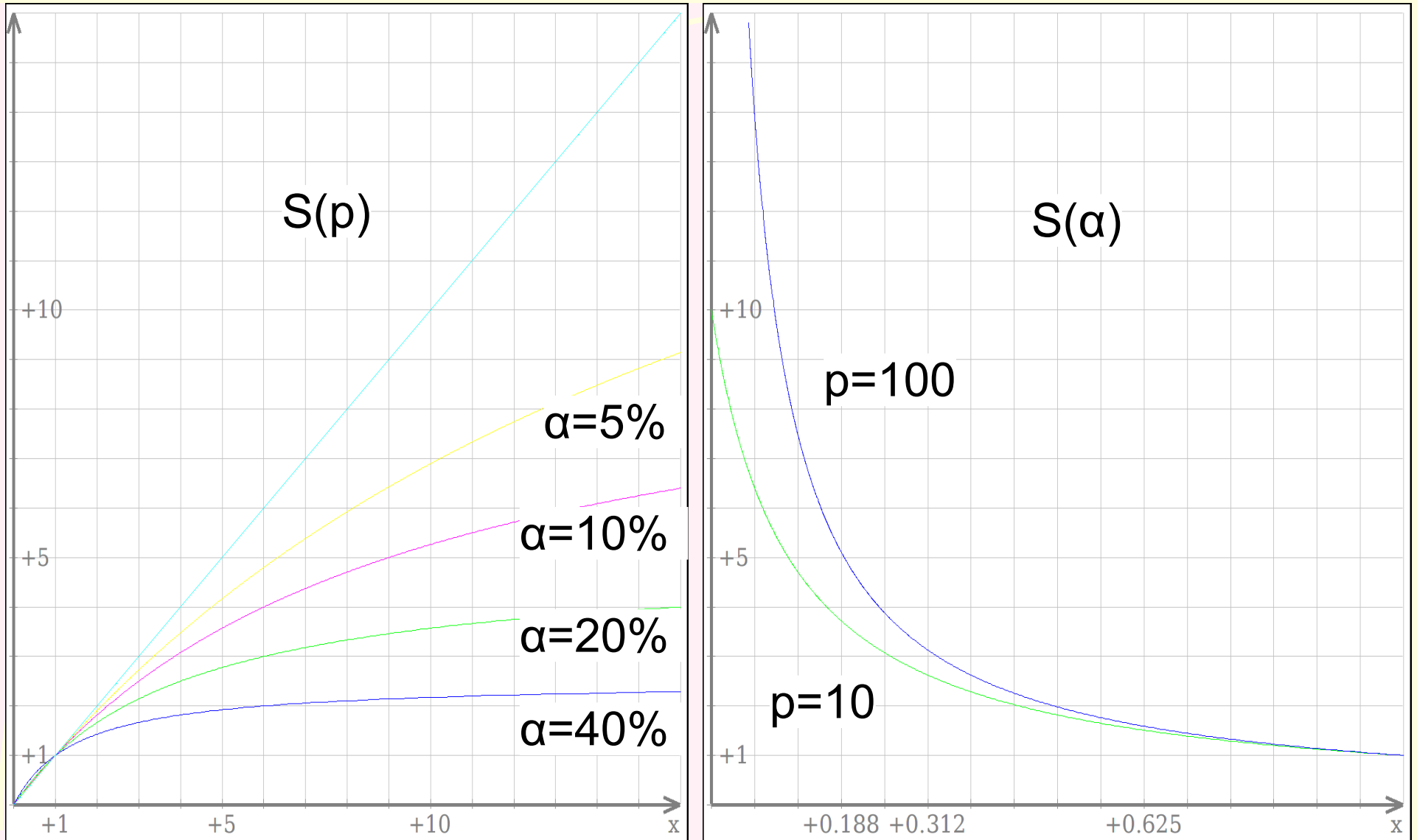


# Amdhal's Law (1967)

- For a problem with a workload  $W$ , we assume that we can divide it into two parts:
  - $W = \alpha W + (1-\alpha)W$   
 $\alpha$  percent of  $W$  **must be executed sequentially**, and the remaining  $1-\alpha$  **can be executed** by  $n$  nodes simultaneously with 100% of efficiency
  - $T_s(W) = T_s(\alpha W + (1-\alpha)W) = T_s(\alpha W) + T_s((1-\alpha)W) = \alpha T_s(W) + (1-\alpha)T_s(W)$
  - $T_p(W) = T_p(\alpha W + (1-\alpha)W) = T_s(\alpha W) + T_p((1-\alpha)W) = \alpha T_s(W) + (1-\alpha)T_s(W)/p$

$$S_p = \frac{T_s}{\alpha T_s + \frac{(1-\alpha)T_s}{p}} = \frac{p}{1 + (p-1)\alpha} \rightarrow \frac{1}{\alpha} \quad (p \rightarrow \infty)$$

# Amdhal's Law Consequences



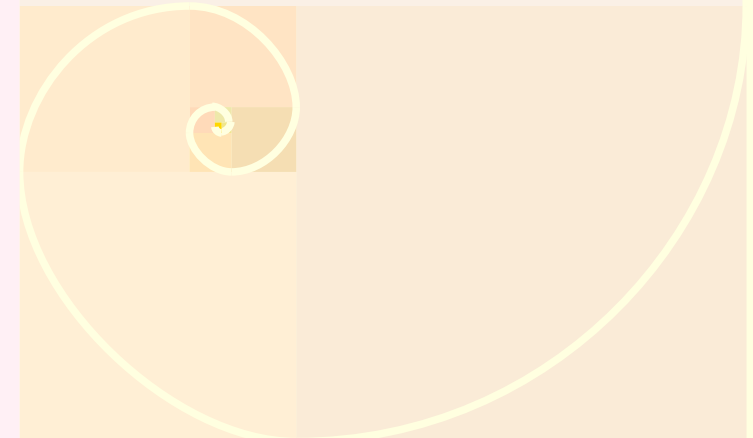
# Amdhal's Law Consequences

- For a fixed workload, and without any overhead, the maximal speedup as an upper bound of  $1/\alpha$
- The sequential bottleneck cannot be solved just by increasing the number of processors in a system.
- So, to get some speedup from new processors, applications will have to be **highly concurrent**
  - sequential part reduced to the strict minimum!

**The free lunch is over! [Sutter 2007]**

# Concurrency today

- The vast majority of software are designed:
  - according to the Turing Machine Model
  - with threads
- CS&E Students are still learning sequential programming up to graduate levels
- Any change on the horizon?



# Lesson from the past: Object Oriented Programming

- 60's: Simula then Smalltalk are first object oriented languages
- 1990: Wide adoption of OOP
  - Gap of 30 years! Why?
    - Many reasons
      - No real need before: applications were too simple in the vast majority of cases
      - Main stream languages were C & Pascal not Smalltalk
    - 1990: GUIs appeared on desktops
      - OOP fits very well in the development of GUIs
- OOP was a revolution: teachers, students, industry and researchers had to adapt

# Dealing with concurrency

- Parallel and Concurrent Programming not new
  - A lot of results
  - Never widely used! (limited to narrow areas: HPC)
- Concurrent Programming is provably harder than sequential programming
  - Non-determinism is hard to tackle with!
  - Debugging almost impossible with conventional techniques

We have threads for concurrent programming so where is the problem?

- Threads cannot be implemented as a library [Boehm 2005]
  - Threads are provably bad [Lee 2006]

# Consequences: end-users

- More bugs in applications
  - More deadlocks: application freeze
  - More inconsistencies: application data corrupted
- Purchasing new processor will not improve the performance by a significant speedup!
  - Beware of advertising!
    - If there is an improvement for a single threaded application it is because of faster clock, cache, bus and so on...
    - Not due to the actual processor design

# Consequences: industry

- Programming skills
  - Programmers that can deal with concurrent programming will get good jobs in the next future!
- Write/Compile/Test/Debug cycle will probably be changed
  - Proving formally that the written application is doing what it is supposed to do
  - Find bugs automatically
- Getting an efficient and safe software will take longer than before
  - Software Development cost may increase

# Consequences: students (CS or CE, Software of System)

- Keep good basic knowledges
  - Foundations of CS: Turing Machines, Lambda Calculus, Von Neumann
  - Mainstream languages: C, Java
- Stay on the edge
  - Rising languages (ruby, C#)
  - New hardware features (HT, CMT, ...)
- Start learning concurrent programming
  - Multi-threading (at least)
  - Other paradigms (Actors, Active Objects, Separates, ...)

# Consequences: research

- New computation models Required
  - Turing Machine & Von Neumann are outdated!
- Good programming paradigm & languages
  - Already some propositions:
    - E language (HP)
    - Fortress (Sun)
    - Polyphonic C#, Haskell/STM (Microsoft Labs)
- Many research workgroups
  - Berkeley, MIT, ...



# The Future

# Homogeneous or Heterogeneous

- Intel bet on homogeneous multi-core
  - promise 100 cores in 2010
- AMD bet on heterogeneous multi-core
  - Graphics Processing Unit (GPU) from ATI will be incorporated into the chip along with other cores
- Sun bet on high CMT (homogeneous)
- IBM bet on high heterogeneous processor
  - Cell processor is already composed of:
    - One dual-core Power processor
    - 8 Vector Processors with their own memory
- Impact on the whole industry!

# The promise of virtualization

- Virtual machine
  - Sun Java and .NET are already in mainstream
  - Low-level virtualization is the current trend
    - Execute several OS at the same time on the same machine (e.g: VMWare, IBM AS/400, Sun Dzone, Linux KVM, Xen, etc.)
- Next step maybe the removal of the OS
  - Applications are written directly on top of a virtual machine
  - The processor embed part of the functions provided today by the OS
- FPGA a promising approach [Berkeley 2006]

# Conclusion

# Conclusion

- CS is majority based on old models (computation, hardware, software)
- The free lunch is over
  - Applications will have to be highly concurrent to get speedup from new processors
  - Concurrent programming is hard
- Concurrent programming is not new
  - but still not widely adopted
- More research required
  - provide new models (computation, hardware, software)