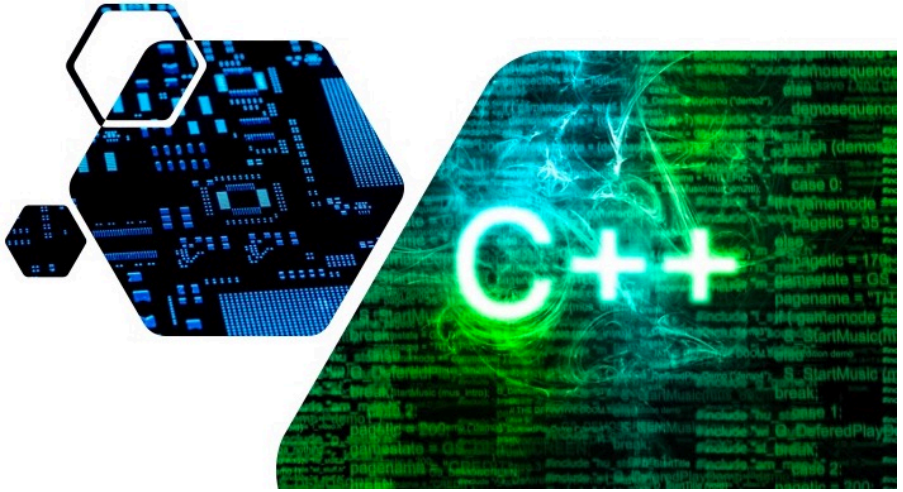
Lecture 1: Introduction to Computer Design Problems

Dr. Tsung-Wei Huang
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT

The course teaches you how to write "good" programs with a specific focus on solving computer design problems. You will gain "hands-on" experience in writing C++/Python code for implementing important data structures and algorithms …



~~*Advanced*~~ *Applied Programming for Computer Design Problems*

Class Logistics

- **Staff**
  - Instructor: Dr. Tsung-Wei Huang
  - TA: I will be helping you in person ☺
- **When/where to meet**
  - 3:30 AM – 6:30 PM every Thursday (excluding holiday)
  - MEB 2555
- **Office hour**
  - Tsung-Wei Huang: 1-3 PM every Wed (MEB 2124)
  - Feel free to email for additional appointments
- **Web: https://github.com/tsung-wei-huang/ece5960**

Scoring

- **Total 100 points**
  - In-class programming practice: 40 points
  - Programming Assignment 1: 20 points
  - Programming Assignment 2: 20 points
  - Programming Assignment 3: 20 points
  - **NO EXAM**
- **Academic integrity**
  - We trust you but don't take it for granted
  - Violation will be recorded in your transcript
  - https://regulations.utah.edu/academics/6-400.php

Learning Materials

- **You DON'T NEED to buy any textbook**
  - Slides will be available and are enough

Class Philosophy

- **Focus on software programming**
  - With applications on computer design problems
  - *Code review and refactor*
- **NOT to trouble you with**
  - Difficult homework
  - Tricky exams
  - Unreasonable learning and scoring curve
- **At the end of the class, I want you to**
  - Understand important computer design problems
  - Improve your coding skills and algorithm knowledge
  - Have more job opportunities in software companies

To this end

- **Each class is organized as two parts**
  - Lecture
  - Programming practice
- **We will start from very basic stuff and then move up**
  - Important data structure and algorithms
  - Important C++ coding techniques and best practice
- **Solve real-world computer design problems in C++**
  - Circuit partition
  - Layout floorplanning
  - Machine learning
  - …

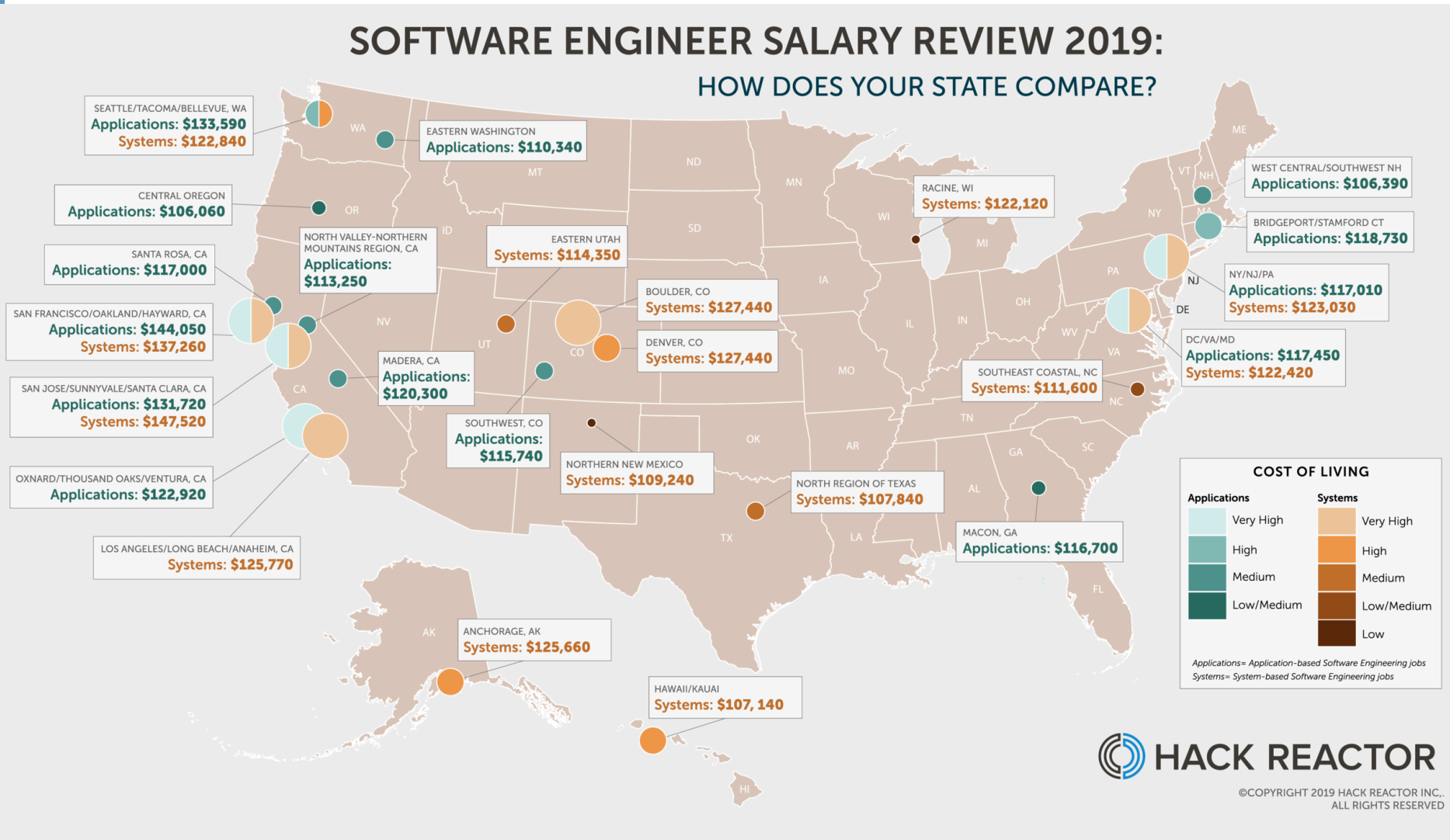
Utah CADE Machines

- **All of you should have access to CADE servers**
  - If not, go apply for one at: https://www.cade.utah.edu/
- **CADE machines have all the facilities we need**
  - Domain: labX-Y.eng.utah.edu
    - X = 1, 2, 3, … (# of lab space)
    - Y = 1, 2, 3, 4, 5, … (machine # of each lab space)
    - For example, lab2-20.eng.utah.edu
  - Account: your uid
  - Password: your uid login password
- **Remote login using ssh (the easiest way)**
  - ssh -x u6024634@lab2-20.eng.utah.edu

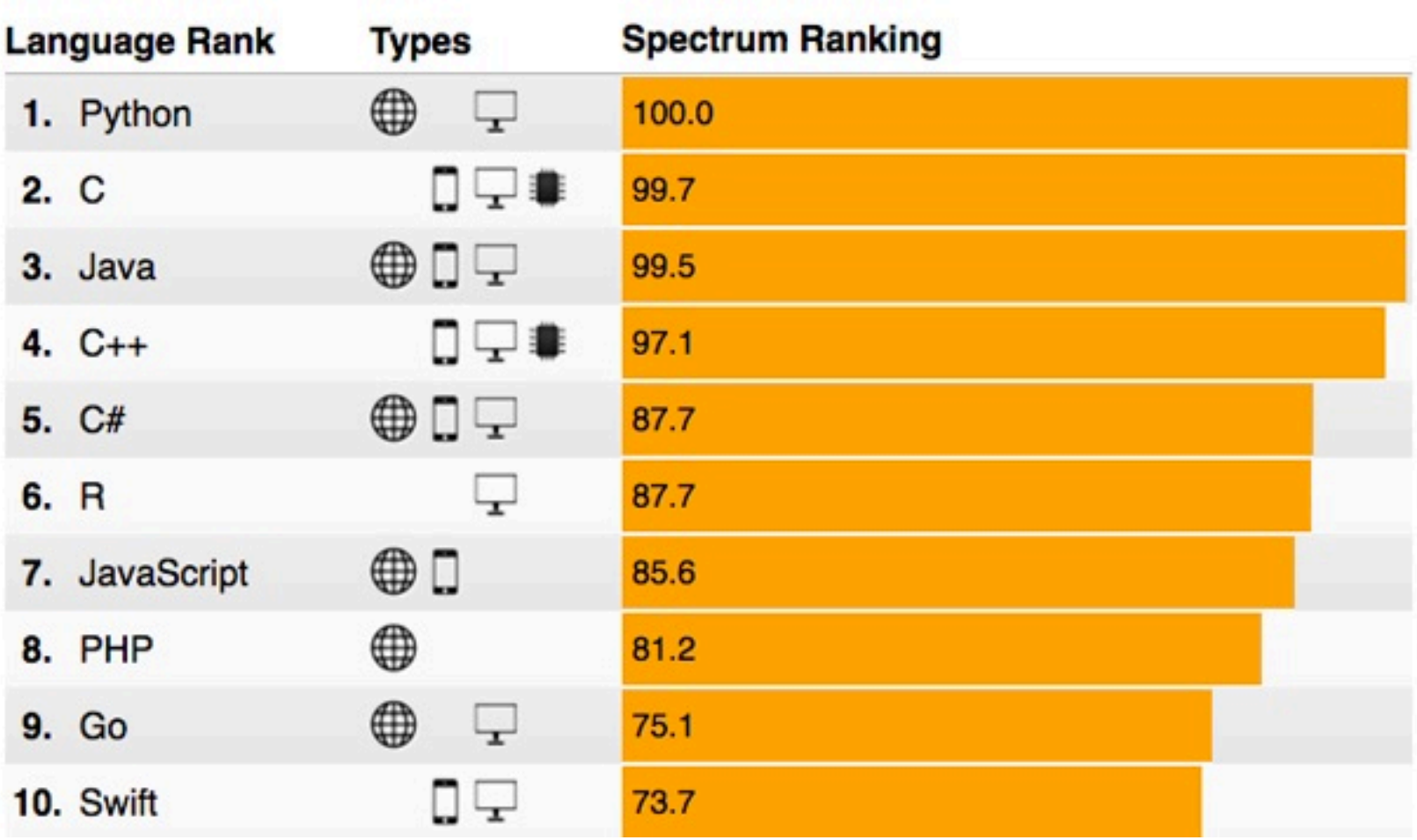
Login using NoMachine

Demo

Software Engineer Job Landscape



Top 10 Programming Languages



Targeted Programming Language

- **We will be using C++ most of the time**
  - Most large-scale problems are written in C/C++
  - Most AI backend engines are written in C/C++
  - Most performance-critical blocks are written in C/C++
- **C++ is advantageous in**
  - Being both high-level and low-level
  - Linking to hardware and computer architecture
  - Understanding memory hierarchy and cache effect

What are computer design problems?

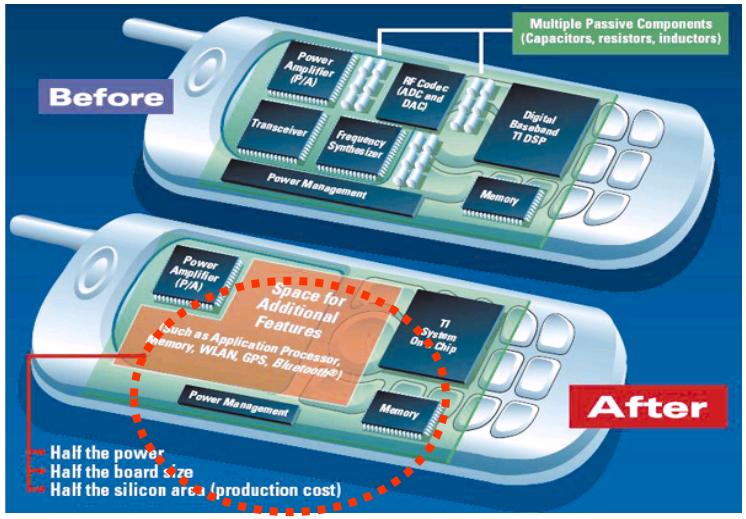
System Design Lifecycle

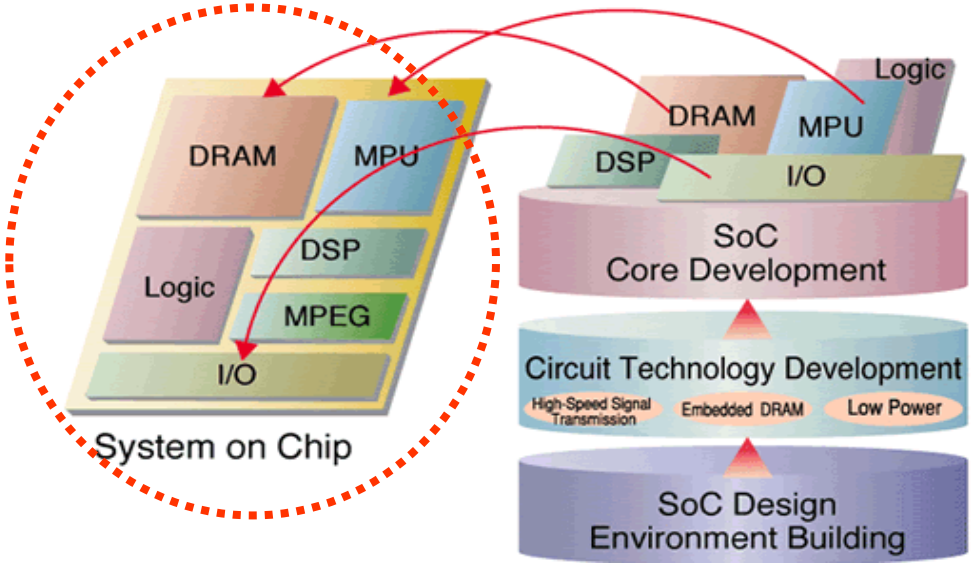
	1970	1980	1990	2002
Cost of 1MHz	$7,600.82	$103.40	$25.47	$0.17
Cost of 1 megabit storage	$5,256.90	$614.40	$7.85	$0.33
Cost of sending 1 trillion bits	$150,000.00	$129,166.67	$90.42	$0.12

Source: "The New Paradigm" Federal Reserve Bank of Dallas 1999 Report and 2002 Actuals

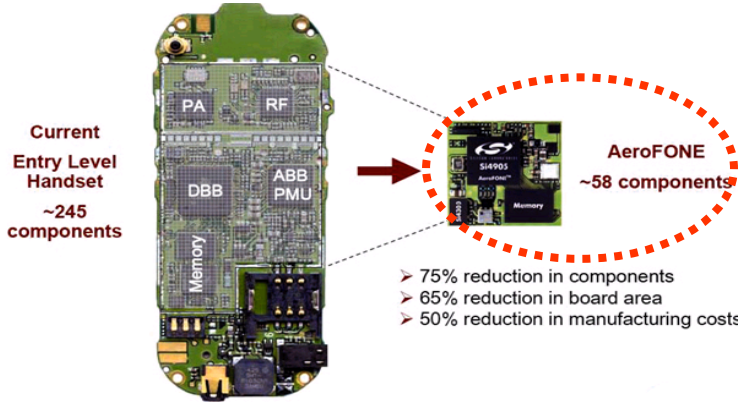
Very Large Scale Integration (VLSI)

- Improve performance, power, and area

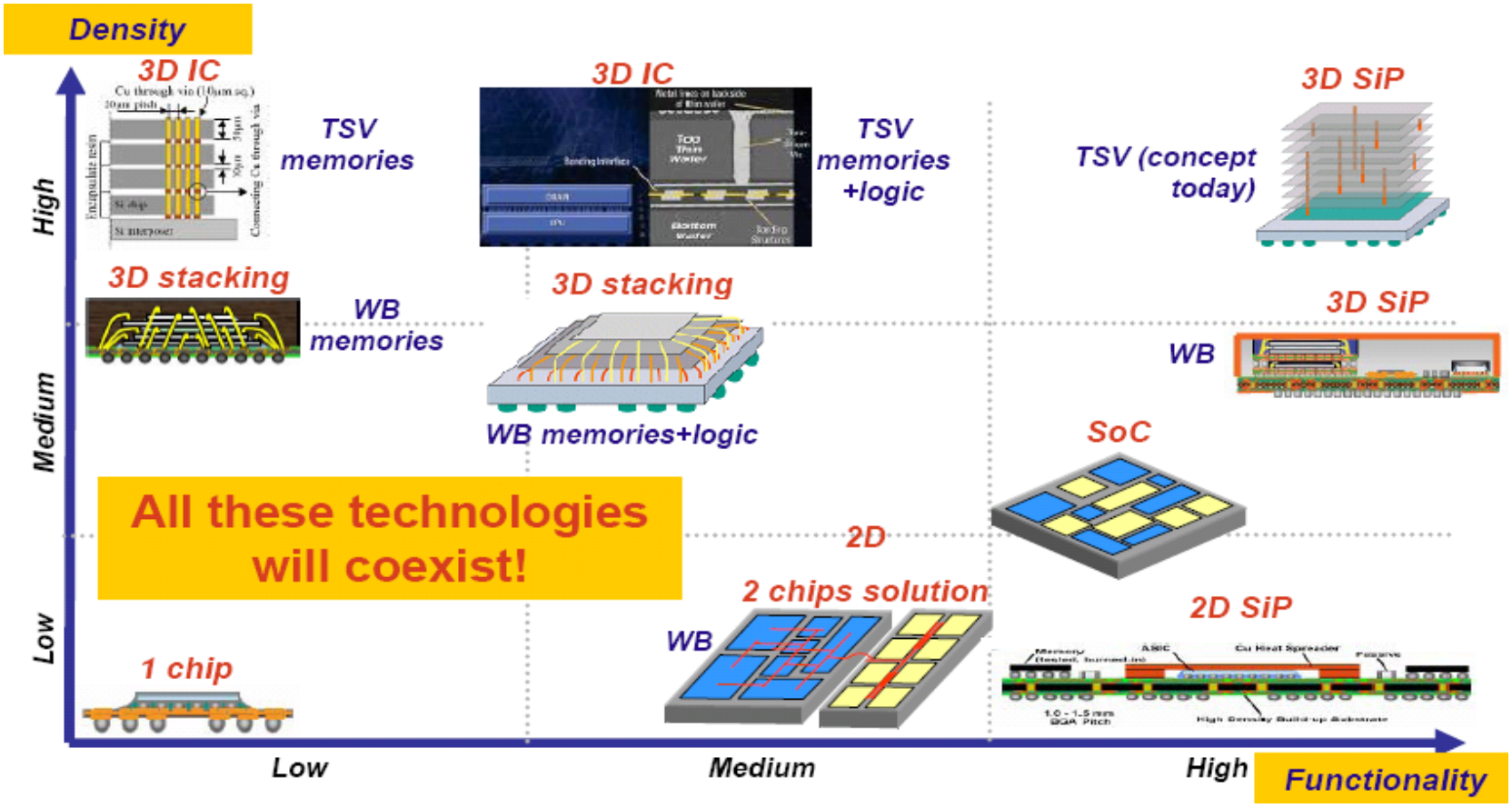




Smaller chip also improves the power, cost, performance, and many other things

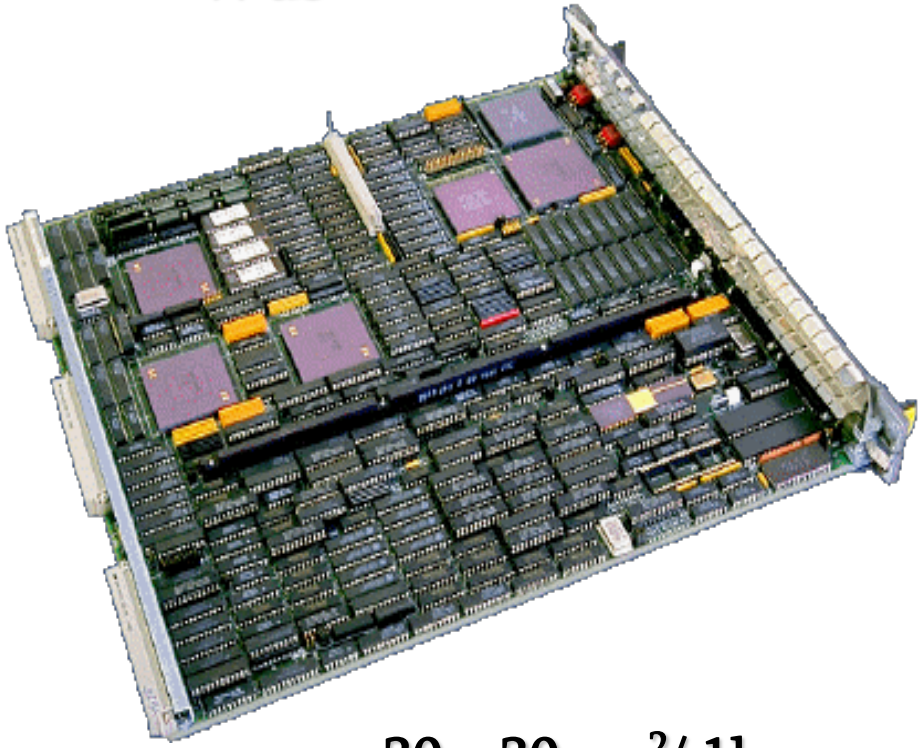


System on Chip or System in Package



Miniaturization

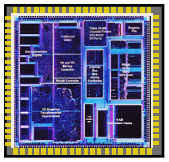
Was



30 x 30 cm^2/ 1kg

Now…

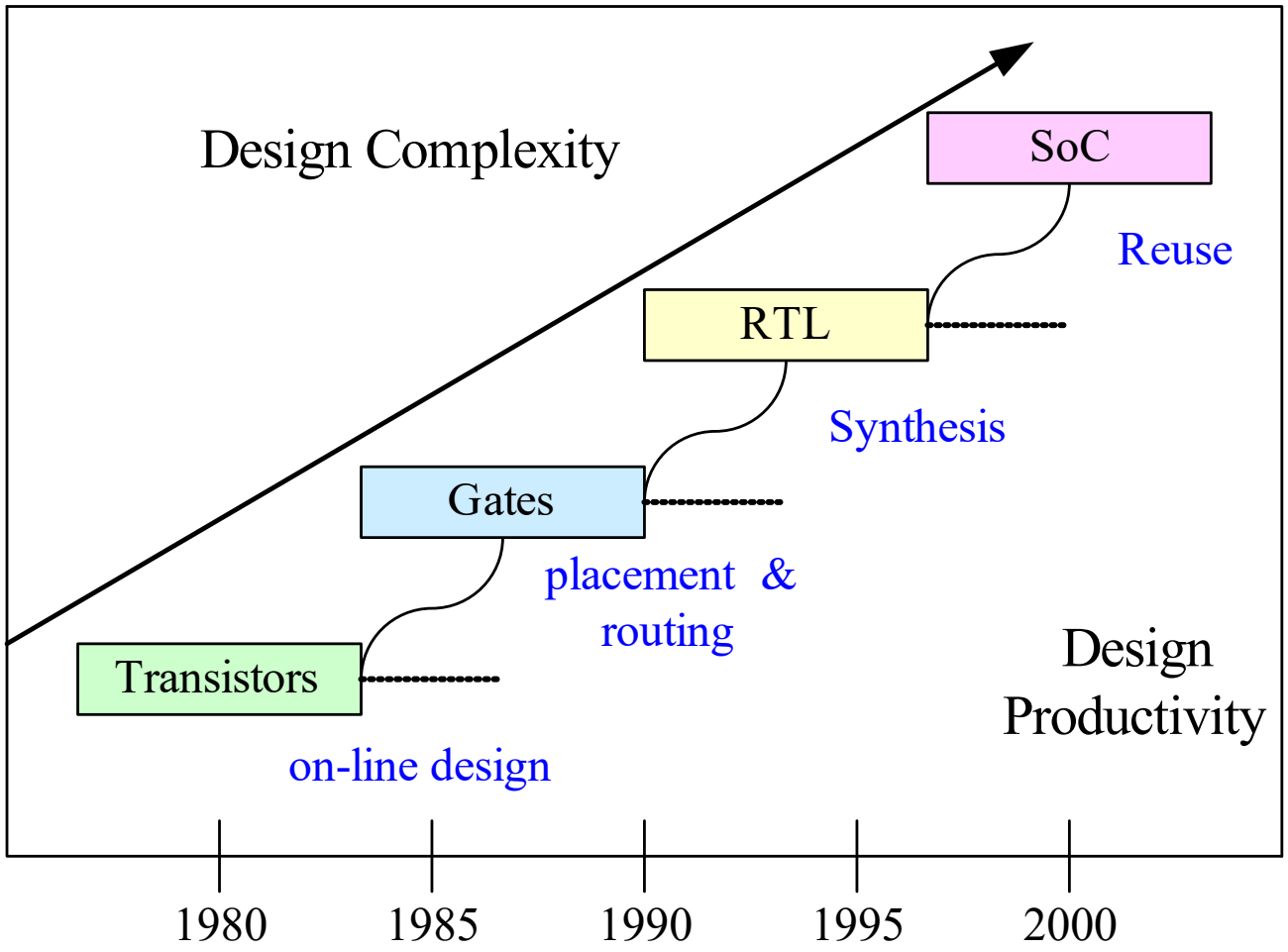
PC on Chip



2 x 2 cm^2 / 10g

Trends of VLSI design

- It must be simple and reusable



Representing a “Design”

- **Behavioral representation**
  - Describes the functionality of a target
- **Structural representation**
  - Describes the connectivity between components
- **Physical representation**
  - Describes the layout, polygons, and spacing rules

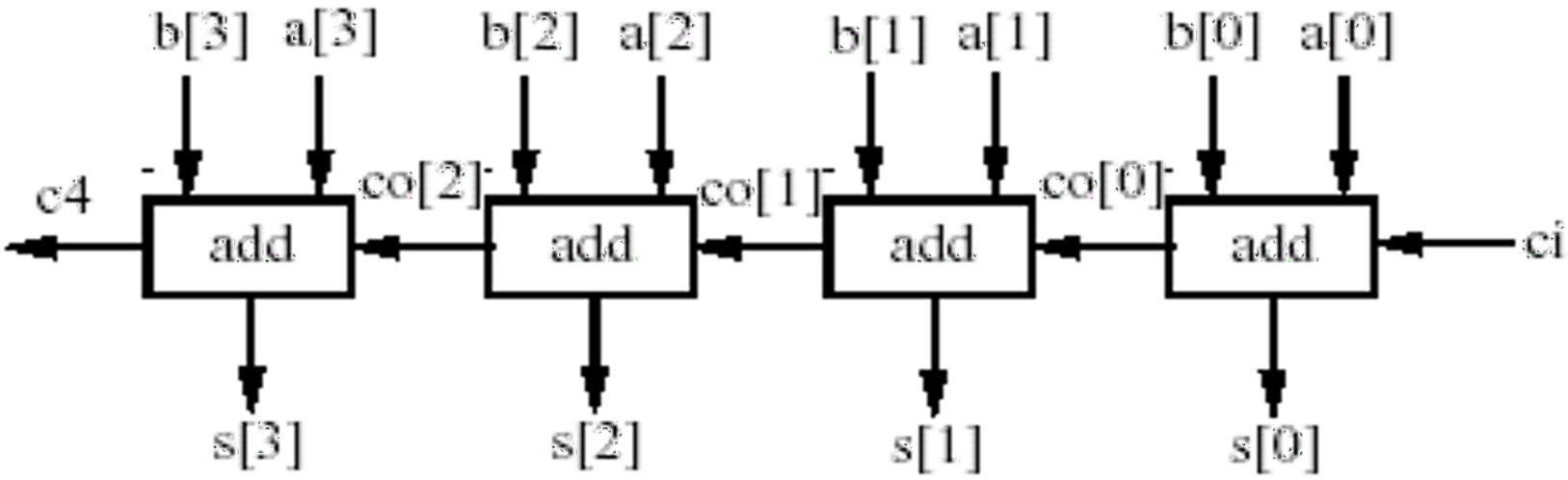
Behavioral Representation

- **High-level description language**
  - System-C
  - System-Verilog
  - VHDL

```
Module add4 (s, c4, ci, a, b);
    input [3:0] a, b;
    input ci;
    output [3:0] s;
    output c4;
  assign {c4,s} = a + b + ci;
endmodule
```

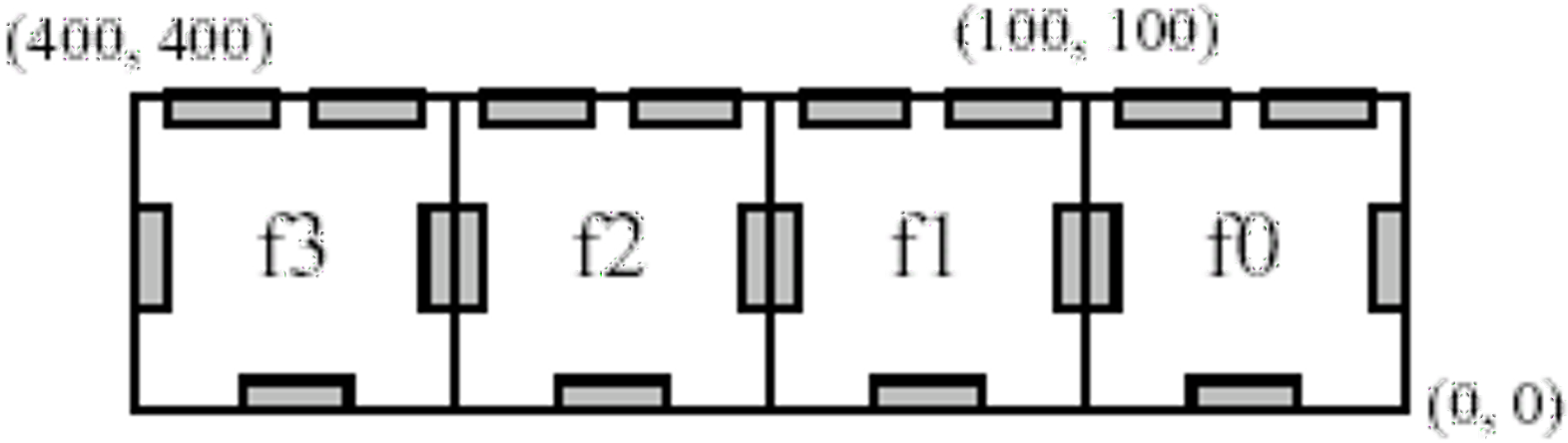
Structural Representation

- **Things are more concrete to implement the target**
  - Gates
  - IP blocks



Physical Representation

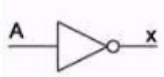
- **The actual geometry to tape-out the target**
  - Polygons
  - Metal layers
  - Spacing rules



The Design Style of Digital Circuits

- **Digital circuit's signal is at either one or two levels**
  - on/off, 0/1, true/false
  - Use transistors to create logic gates for Boolean ops

Logic Gates

NOT	AND	NAND	OR	NOR	XOR	XNOR
$\overline{A}$	AB	$\overline{AB}$	$A+B$	$\overline{A+B}$	$A \oplus B$	$\overline{A \oplus B}$
						

A	X
0	1
1	0

B	A	X
0	0	0
0	1	0
1	0	0
1	1	1

B	A	X
0	0	1
0	1	1
1	0	1
1	1	0

B	A	X
0	0	0
0	1	1
1	0	1
1	1	1

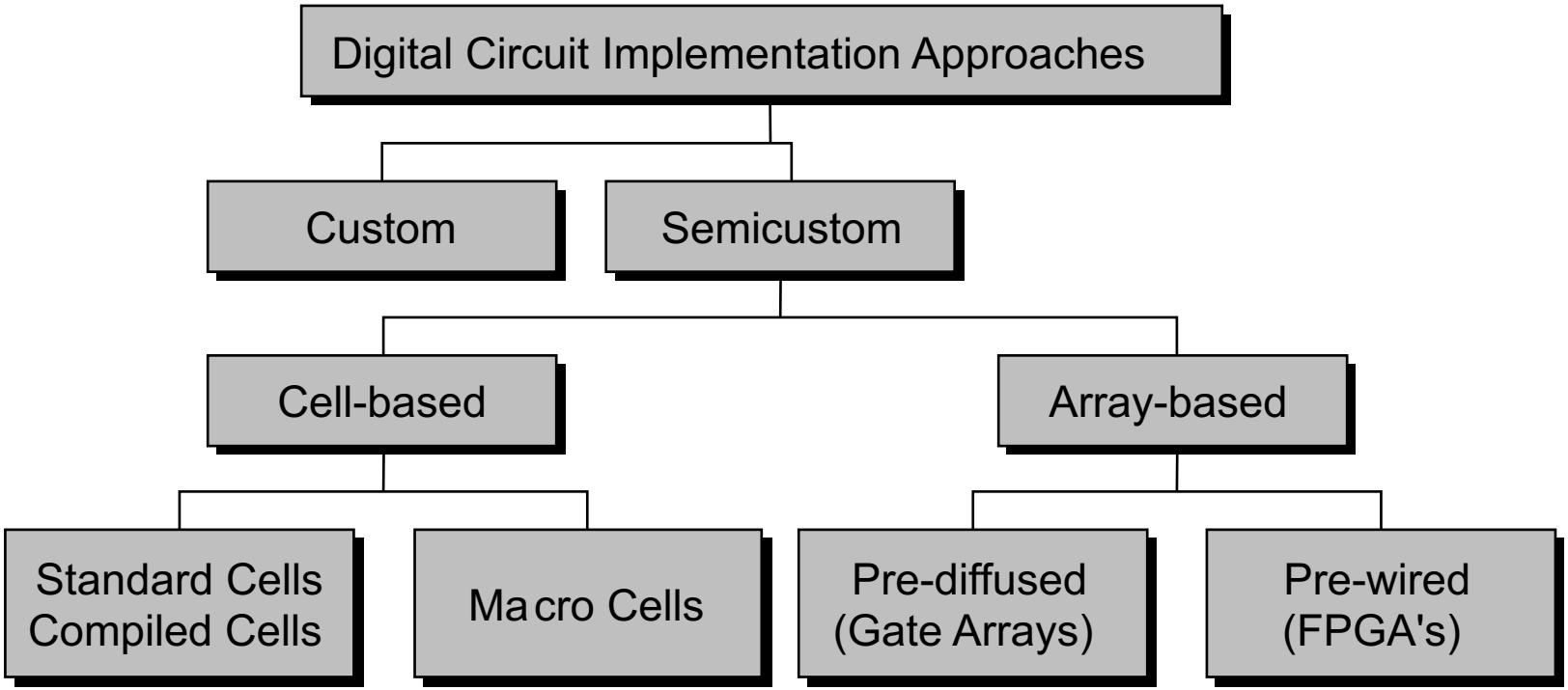
B	A	X
0	0	1
0	1	0
1	0	0
1	1	0

B	A	X
0	0	0
0	1	1
1	0	1
1	1	0

B	A	X
0	0	1
0	1	0
1	0	0
1	1	1

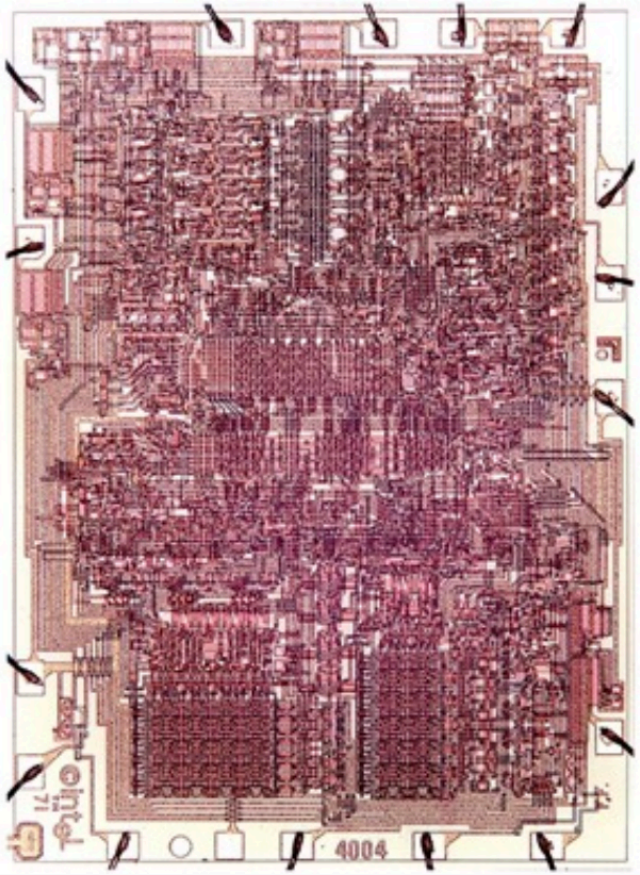
The Design Style of Digital Circuits

- **The rule of thumb (design principle)**
  - Systematic
  - Reusable



The Full-custom Approach (I)

- **Early day, people handcrafted everything …**



Courtesy Intel

Intel 4004 microprocessor
(108 KHz, 2300 transistors, 10um)

*When **performance** or **design density** are critical, handcrafting circuit topology and physical design seems to be the only option*

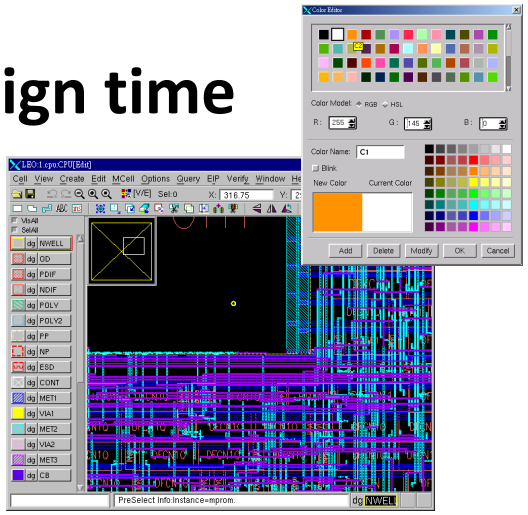
- *high cost*
- *long time-to-market*

It is OK when

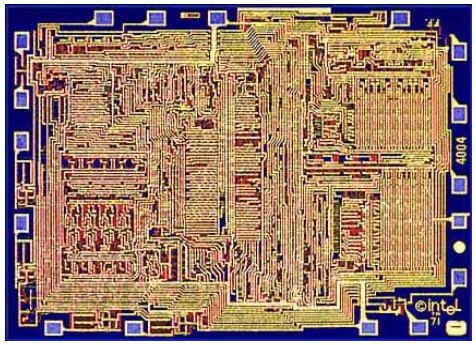
- *custom blocks can be reused*
- *cost can be amortized over a large volume (e.g., uP,memories)*
- *cost is not primary design criterion (e.g. Supercomputers)*

The Full-custom Approach (II)

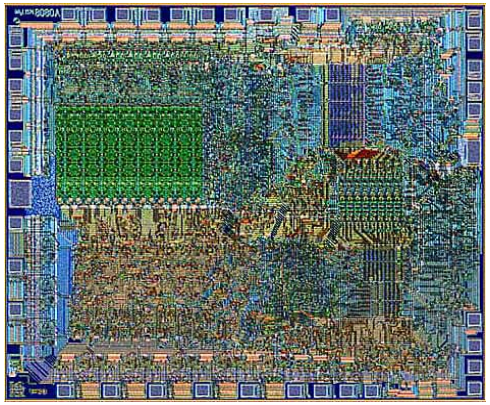
- **Performance at transistor level**
  - Utilize layout editing tools
  - Virtuoso (Cadence)
  - Laker (SpringSoft, now Synopsys)
- **Very expensive in design cost and design time**
  - 10-20 gates per week
- **Used for:**
  - Analog
  - Leaf cells - libraries, memory cells
  - Datapath in high performance designs



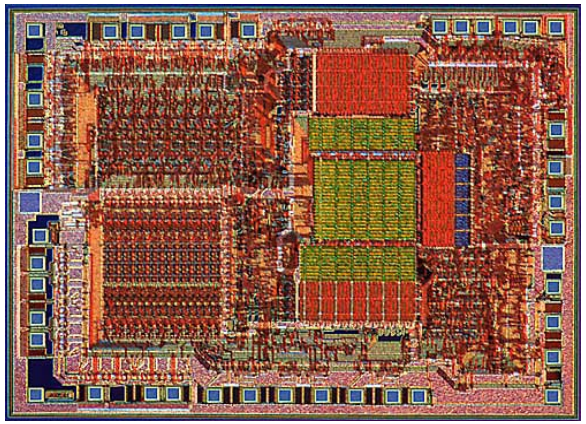
Toward Automation and Regular Structures



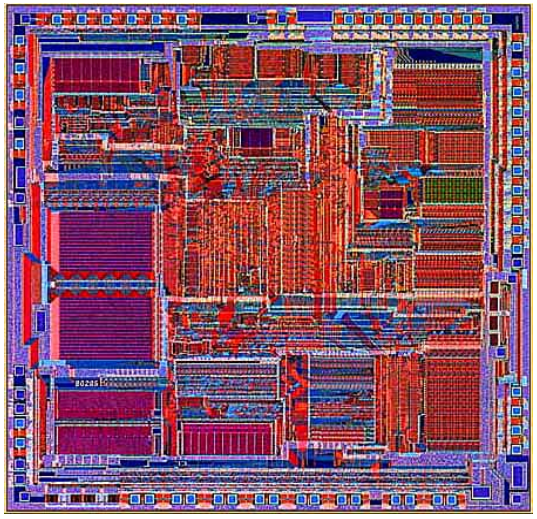
Intel 4004 (‘71)



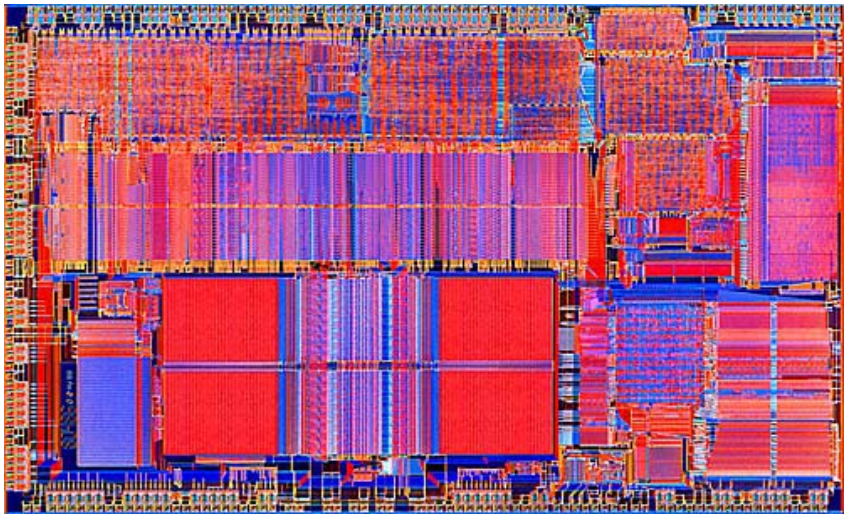
Intel 8080



Intel 8085



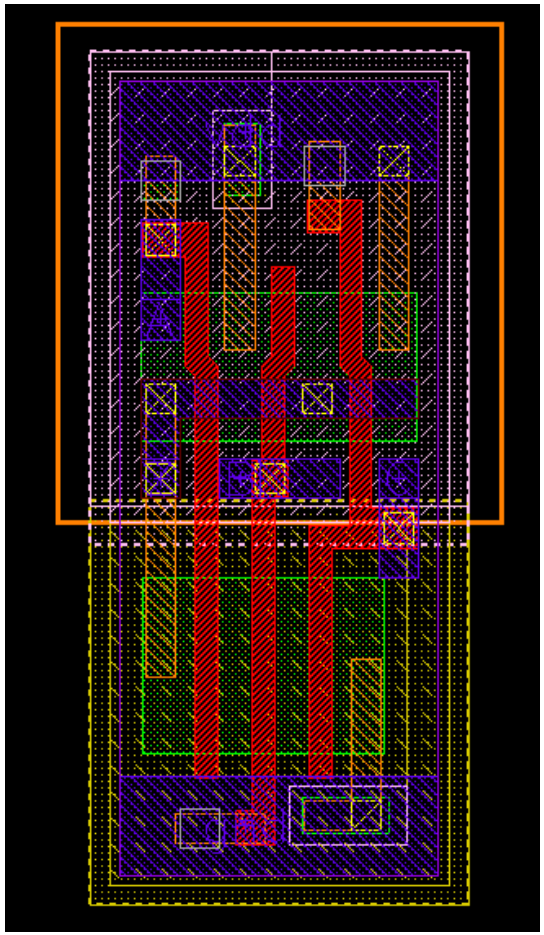
Intel 8286



Intel 8486

Reusability

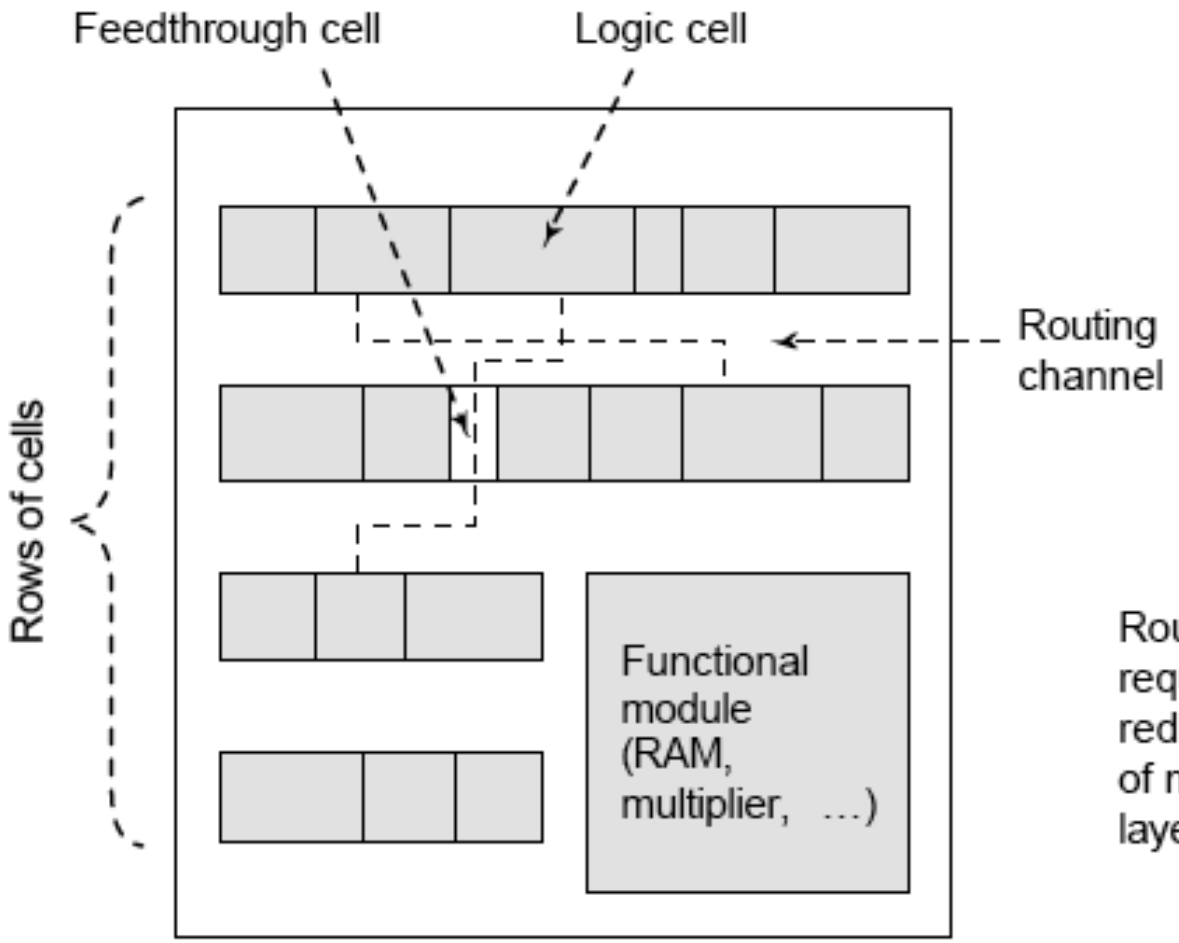
❑ Standard cell methodology



Path	1.2V - 125°C	1.6V - 40°C
$In1—t_{pLH}$	$0.073+7.98C+0.317T$	$0.020+2.73C+0.253T$
$In1—t_{pHL}$	$0.069+8.43C+0.364T$	$0.018+2.14C+0.292T$
$In2—t_{pLH}$	$0.101+7.97C+0.318T$	$0.026+2.38C+0.255T$
$In2—t_{pHL}$	$0.097+8.42C+0.325T$	$0.023+2.14C+0.269T$
$In3—t_{pLH}$	$0.120+8.00C+0.318T$	$0.031+2.37C+0.258T$
$In3—t_{pHL}$	$0.110+8.41C+0.280T$	$0.027+2.15C+0.223T$

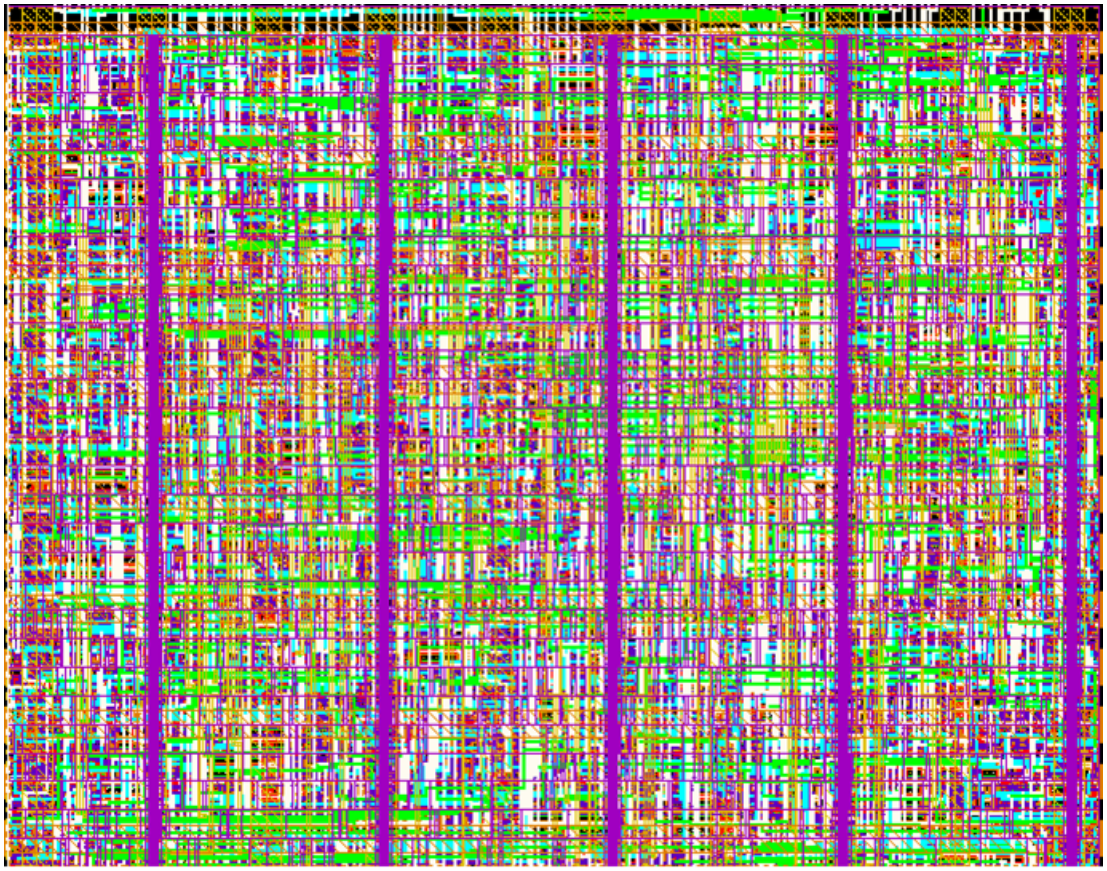
3-input NAND cell
(from ST Microelectronics):
C = Load capacitance
T = input rise/fall time

Cell-based Design



Routing channel requirements are reduced by presence of more interconnect layers

The New Generation of Standard Cells



Cell-structure hidden under interconnect layers

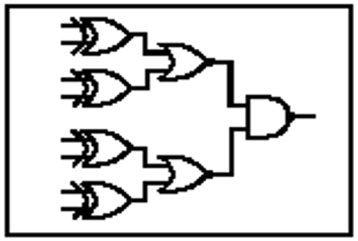
Advantages of Standard Cells

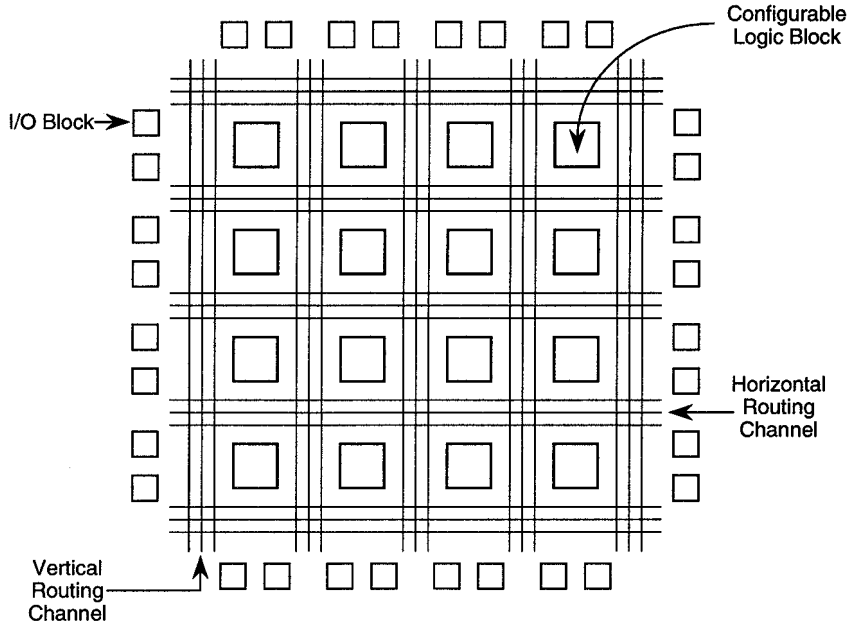
- **Cells are characterized and stored in library**
  - Standard cells: FFs, AND, OR..
  - Macro cells: Memory, PLA, ALU…
- **Need update when technology advances**
- **Easier to develop CAD tools for optimization**
- **Design process is simplified**
  - Manufacturing process is not simplified

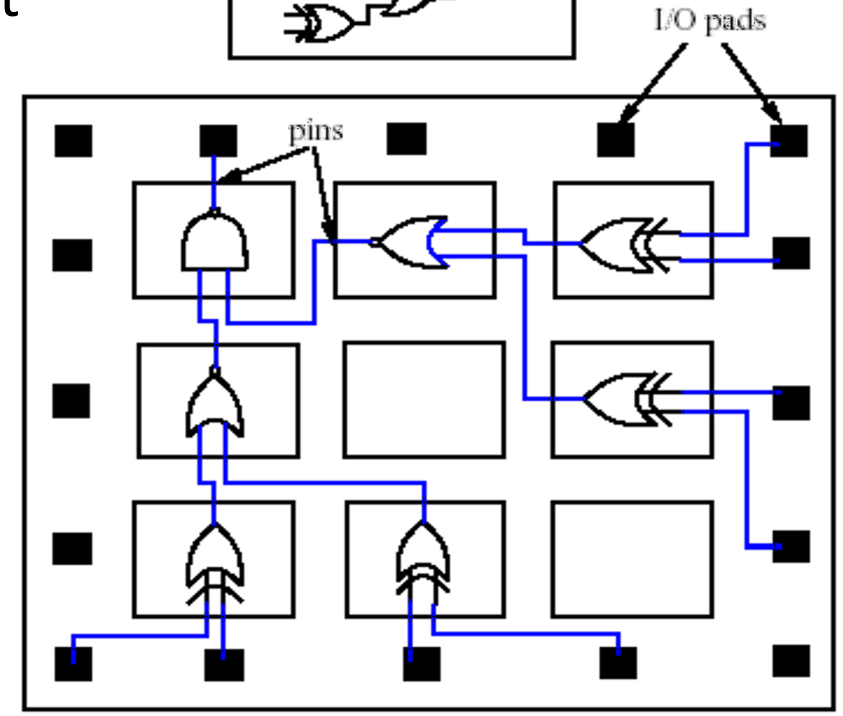
Manufacturing chips is never easy …

Gate Array Design Style

- **Arrays and sites are pre-manufactured**
  - NAND/NOR gates already there
  - Just need to deal with wiring
  - Reduces production cost







Field Programmable Gate Array (FPGA)

Arrays of programmable modules with the capability of implementing a generic logic function

Wires can be connected by programmable connection boxes (CB) and switch boxes (SB)

Reduce development and production time

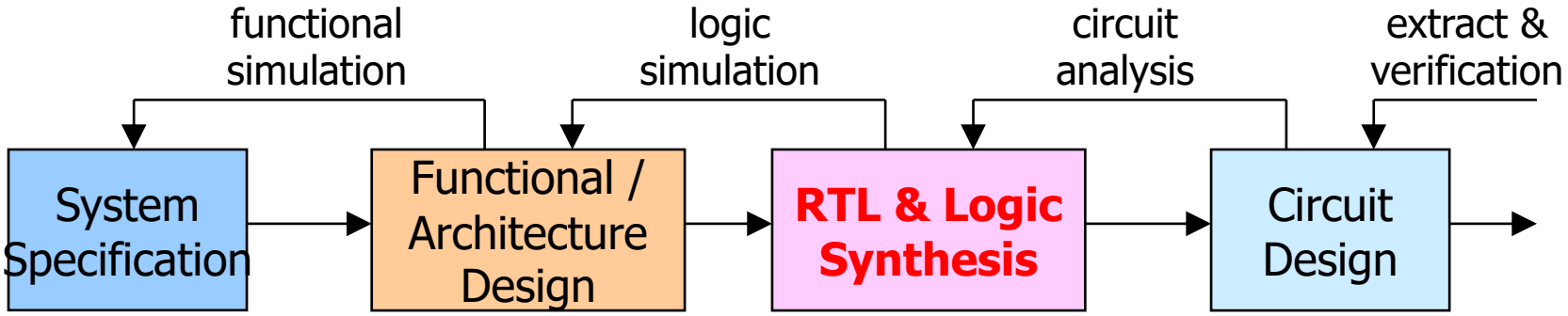
Low cost prototyping

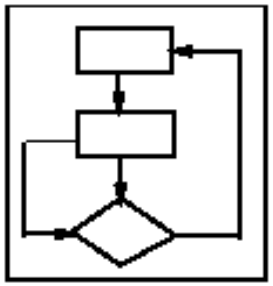
Comparison of Design Styles

	Full custom	*Standard cell*	*Gate array*	*FPGA*	*SPLD*
Density	Very high	High	High	Medium	Low
Performance	Very high	High	High	Medium	Low
Flexibility	Very high	High	Medium	Low	Low
Design time	Very long	Short	Short	Very short	Very short
Manufacturing time	Medium	Medium	Short	Very short	Very short
Unit cost -- small quantity	Very high	High	High	Low	Very Low
Unit cost – large quantity	Low	Low	Low	High	Very High

Standard cell methodology enables reusability, modularity, and facilitate the process of computer-aided design (CAD) or electronic design automation (EDA)

Standard-cell Design Automation Flow (I)

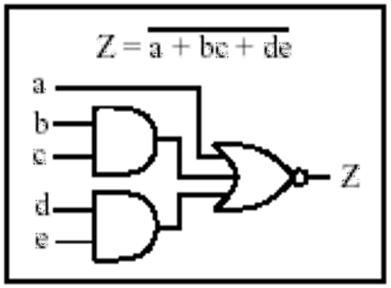




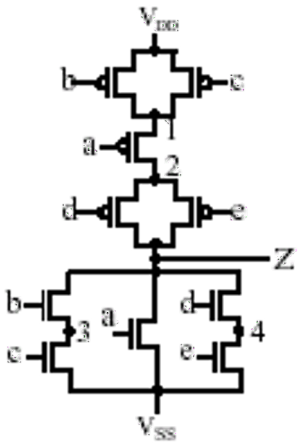
behavior representation

```
module fz(a,b,c,d,e,Z);
input a,b,c,d;
output Z;
assign Z =
  ~(a|(b&C)|(d&e));
endmodule
```

HDL description

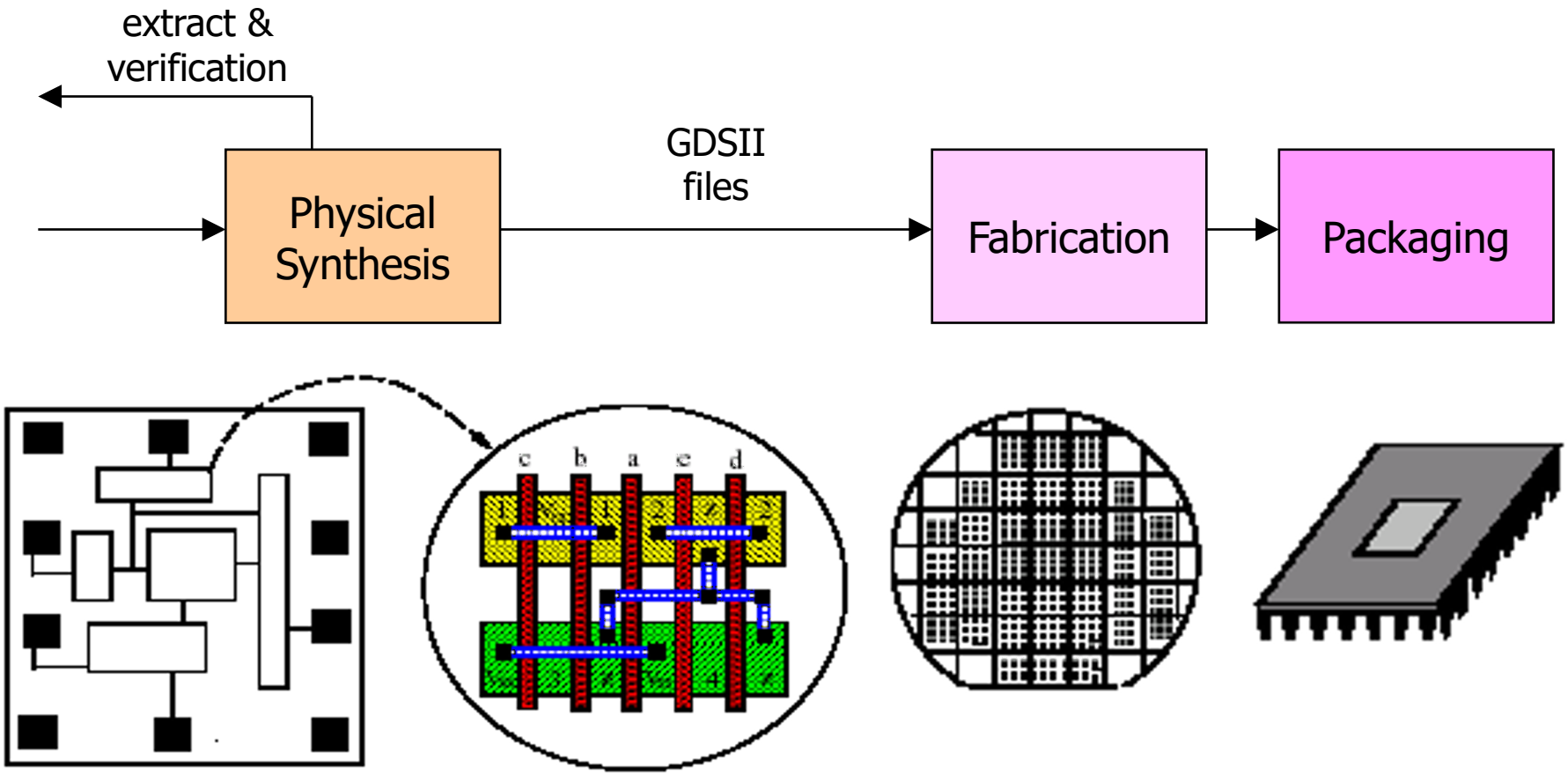


gate-level representation

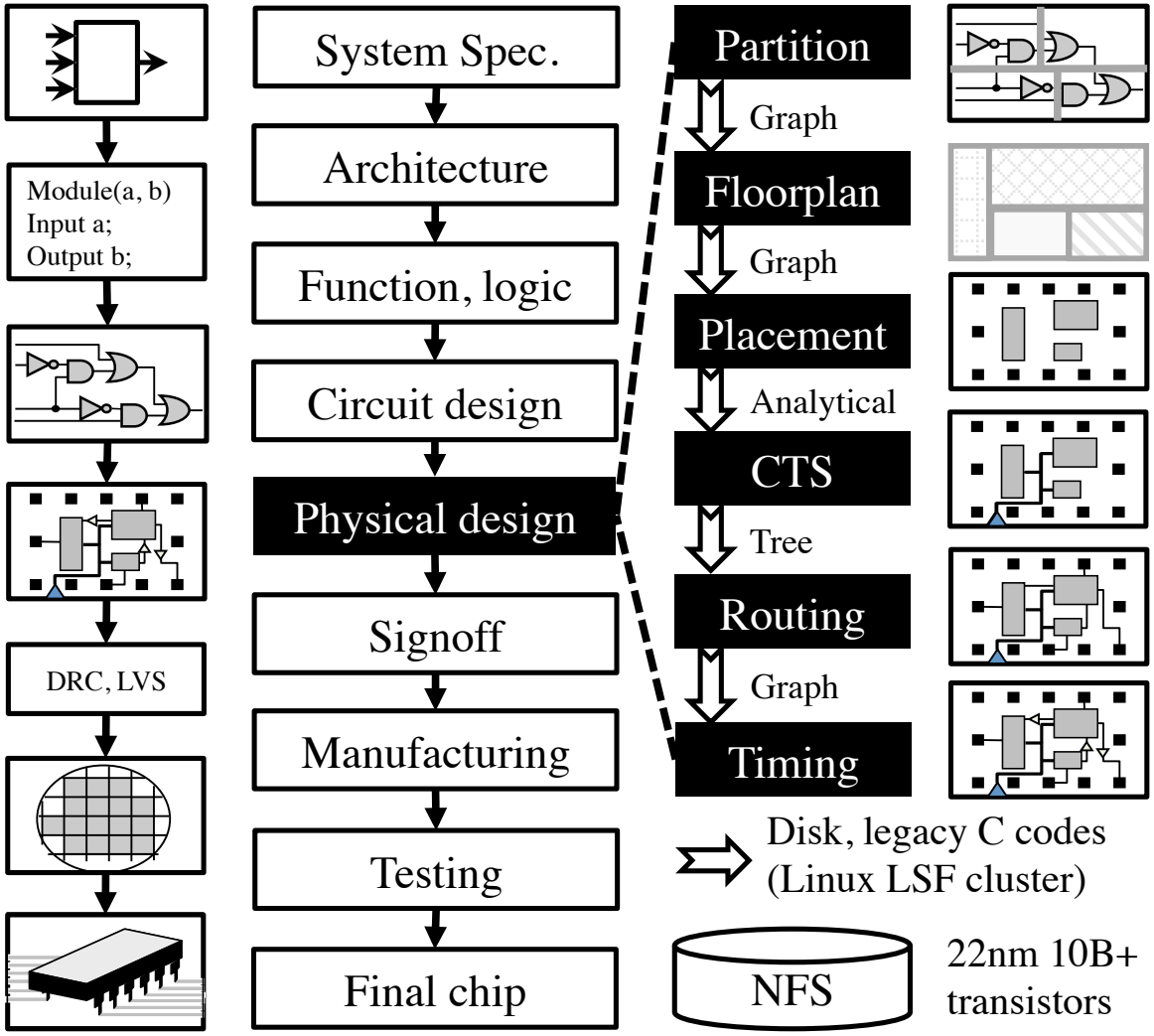


switch-level representation

Standard-cell Design Automation Flow (II)



Physical Design Details

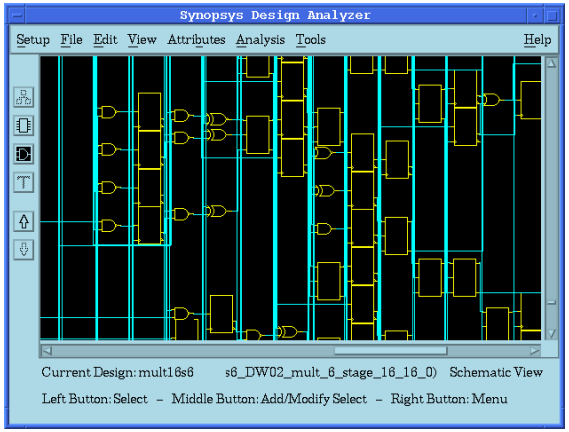


Hardware Description Language (HDL)

- **A "language" to describe a deign**
  - The structure of a system
  - Different methods to achieve a function
  - Logical structures that perform the architecture
- **Can describe the design at some levels of abstraction**
  - Behavioral, structural, gate-level
- **Can be used to document the system**
  - Simulation results
  - Functionalities report

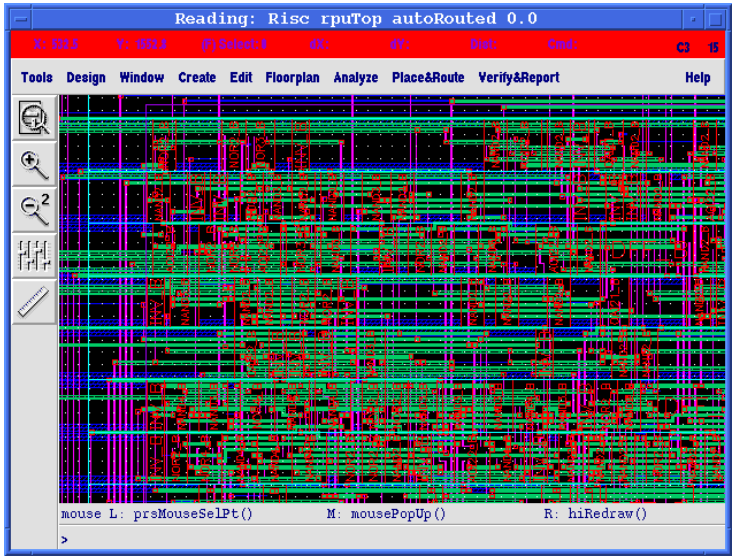
RTL/Logic Synthesis

- **Translate an HDL source code into a netlist**
  - Technology independent optimization (logic minimization)
  - Bind the netlist with user specified cell libraries (technology mapping)
  - Technology dependent optimization
- **Supported synthesis tools:**
  - HDL/Design Compiler (Synopsys)
  - RTL Compiler (Cadence)



Physical Synthesis

- **Transform a gate-level netlist into a layout**
  - Place circuit components
  - Route wires
  - Transform into a mask (GDSII file)
- **Supported P&R tools:**
  - Silicon Ensemble (Cadence)
  - Apollo (Synopsys)
  - SOC Encounter (Cadence)
  - Blast Fusion (Magma)



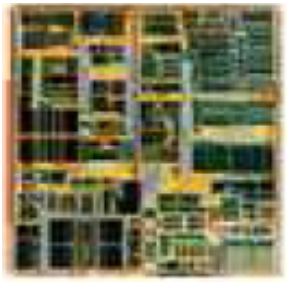
Design Automation is Driving Chip Evolution

1982 – Intel 80286
134K transistors
12MHz; 68.7 mm^2

1985 – Intel 80386
275K transistors
33MHz; 104 mm^2

1989 – Intel 80486
1.2M transistors
50MHz; 163 mm^2

1993 – Intel Pentium
3.1M transistors
66MHz; 264 mm^2



1997 – Intel Pentium II
7.5M transistors
300MHz; 209 mm^2

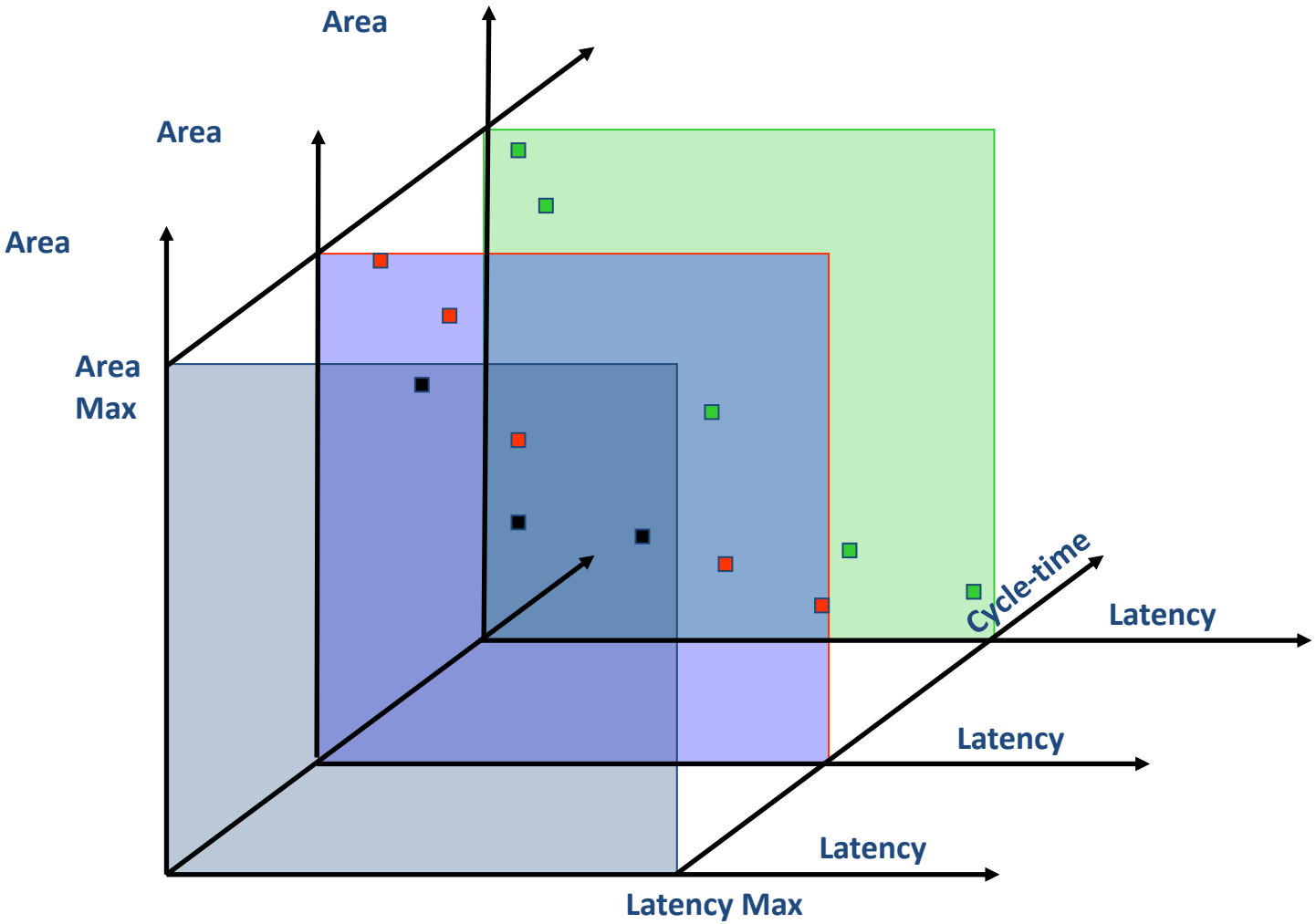
1999 – Intel Pentium III
28M transistors
733MHz; 140 mm^2

2000 – Intel Pentium4
42M transistors
1.5MHz; 224 mm^2

Automation is an Optimization

- **Automation with no optimization has no value**
  - Of course, why are you wasting time!?
- **Optimization aims to outperform manual designs**
  - Too much expensive to rely on human
  - Want the process to be automated
- **Objectives**
  - Low Power
  - Faster timing
  - Smaller area
  - …

There is NO Perfect Solution



We will Learn How to Make Trade-offs

- Multi-criteria optimization
- Pareto points
- Heuristics
- Parameter tuning
- …

How to make a tradeoff though?

You Need to Know Algorithms …

- **Exhaustive search:** Search the entire solution space.
- **Branch and bound:** A search technique with pruning.
- **Greedy method:** Pick a locally optimal solution at each step.
- **Dynamic programming:** Partition a problem into a collection of sub-problems, the sub-problems are solved, and then the original problem is solved by combining the solutions.
- **Simulated annealing:** An adaptive, iterative, non-deterministic algorithm that allows "uphill" moves to escape from local optima.
- **Multilevel framework:** The bottom-up approach followed by the top-down one; good for handling large-scale designs.
- **Mathematical programming:** A system of solving an objective function under constraints.

In-class Practice 1

- **Big Mod**
  - Input: a, b, c (3<a, c<5000000, 1<b<2147483647)
  - Output: $x = a^b \pmod c$
- **Naïve method**

```
for(i=1, j=1; i<=b; i++)
        j = (j*a)%c;
printf("%d\n", j);
```

- **Problem of naïve method**
  - need b iterations
  - Very slow execution for large b

Reflect on Big Mod Problem

- **Naïve method**
  - $2^{16} = 2*2*2*2*2*2*\ldots*2$ total 15 calculation
- **Can we do better?**

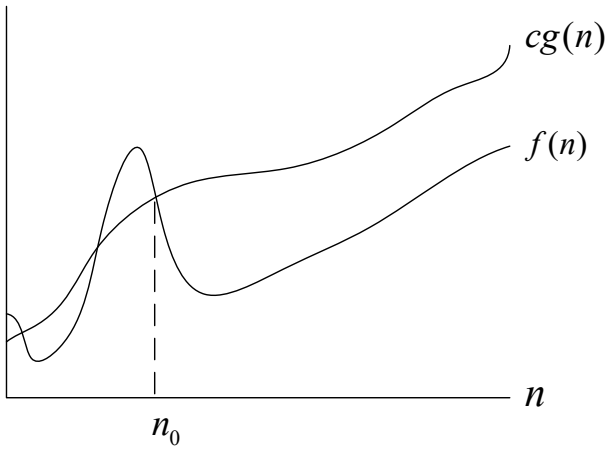
In fact, there is a name for the refined method:

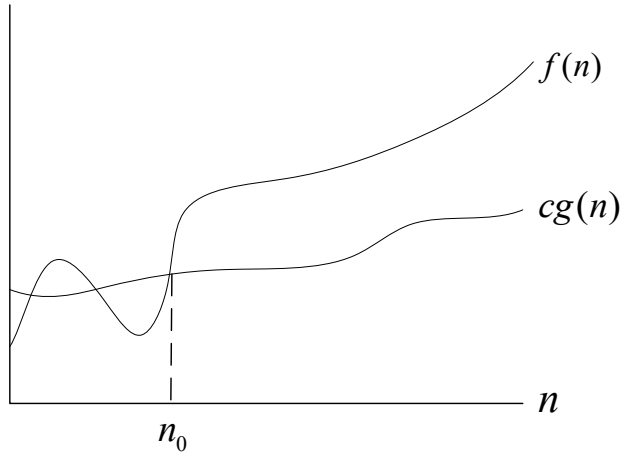
Divide and Conquer algorithm

(used to solve >50% computer design problems, including Google's distributed system solutions, e.g., MapReduce)

Get a Sense of Runtime Performance

- **Time complexity analysis**
  - Runtime plays the most important role in performance
  - In algorithm theory, we use big O worst case analysis
    - Upper bound of the runtime with respect to input size N





In this course, we will use TW's empirical analysis as with modern computer architectures

Empirical Analysis

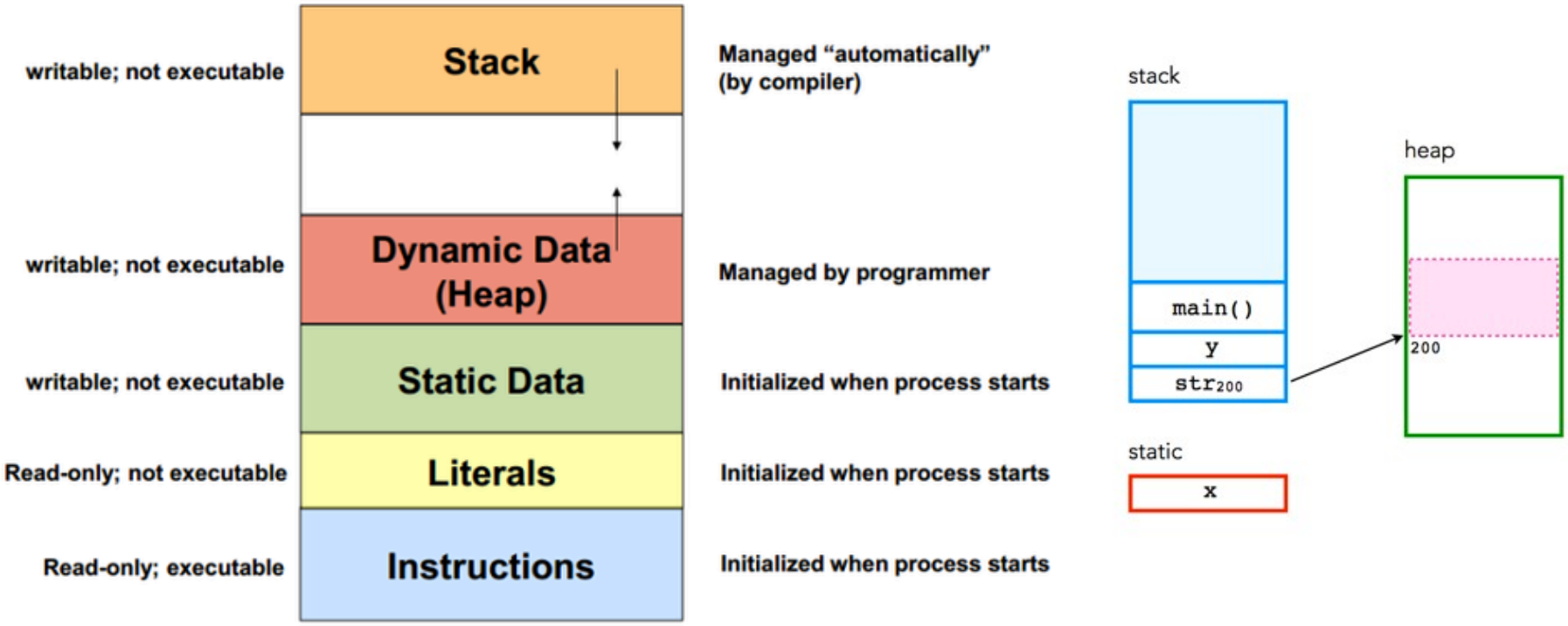
- **Technical Analysis(Input Data Size = N)**
  - O(N)
    - for(int i=1; i<=N; i++) some_work();
  - $O(N^2)$
    - for(int i=1; i<=N; i++)
      for(int j=1; j<=N; j++)
        some_work();
  - $O(N^3)$, $O(N^4)$…
- **On modern computers, input Data Size = N**
  - Timing Constraint
    - O(N) = 1000000 ~ 8000000 equals to Run Time < 1s
    - Generally true on modern PCs

Empirical Analysis Examples

- **A Problem with N = 1000, Time Limit = 1s**
  - Could a O(N) Algorithm Pass the Time Limited Constraint?
  - What about the O(NlogN)?
  - What about the $O(N^2)$?
  - What about the $O(N^2 \log N)$?
  - What about the O(1) time ?
- **A Problem with N = 1000000, Time Limit = 1s**
  - Could a O(N) Algorithm Pass the Time Limited Constraint?
  - What about the O(logN) ?
  - What about the O(logN logN) ?
  - What about the $O(N^2)$?

What about Space Complexity?

- **Same as the empirical runtime analysis**
  - However, it depends on your RAM size
- **Stack vs Heap**



In-class Practice 2

Problem Description

A sequence of n > 0 integers is called a jolly jumper if the absolute values of the difference between successive elements take on all the values 1 through n-For instance,

1 4 2 3

is a jolly jumper, because the absolutes differences are 3, 2, and 1 respectively. The definition implies that any sequence of a single integer is a jolly jumper. You are to write a program to determine whether or not each of a number of sequences is a jolly jumper.

Input

Each line of input contains an integer 2<= n <= 3000 followed by n integers representing the sequence.

Output

Each line of input, generate a line of output saying "Jolly" or "Not jolly".

In-class Practice 2

Sample Input

4 1 4 2 3

5 1 4 2 -1 6

2 1 2

6 11 7 4 2 1 6

10 1 2 5 7 3 8 49 8 9 10

Sample Output

Jolly

Not jolly

Jolly

Jolly

Not jolly

Summary

- **Design automation flow to handle computer designs**
  - High-level source code
  - Logic synthesis
  - Physical synthesis
  - Testing and verification
  - Tape out
- **Software method to replace manual effort**
  - Need efficient algorithms and data structures
- **Empirical space/time complexity analysis**
- **Two practice problems**
- **Next time: data structure and C++ STL**