EE6094
CAD for VLSI Design

Chapter 1
Introduction to EDA

Spring 2021
Andy, Yu-Guang Chen
Assistant Professor, Department of EE
National Central University
andyygchen@ee.ncu.edu.tw

Course Website

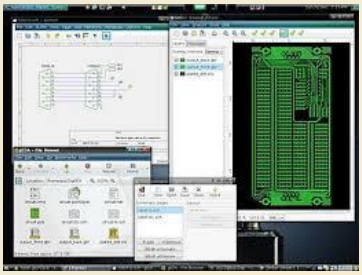
- https://ncueeclass.ncu.edu.tw/course/6662

Outline

- What is EDA?
- VLSI Design Flow
- Design Styles
- VLSI Design Cycle
  - Logic Synthesis
  - Physical Design
- Lithography
- Algorithms
  - Complexity Analysis
  - P, NP, NP-Complete, and NP-Hard
- IDEA Project

What is EDA?

- Electronic design automation (EDA)
  - 電子設計自動化
- VLSI computer-aided design (VLSI-CAD)
  - 積體電路電腦輔助設計



What is EDA?

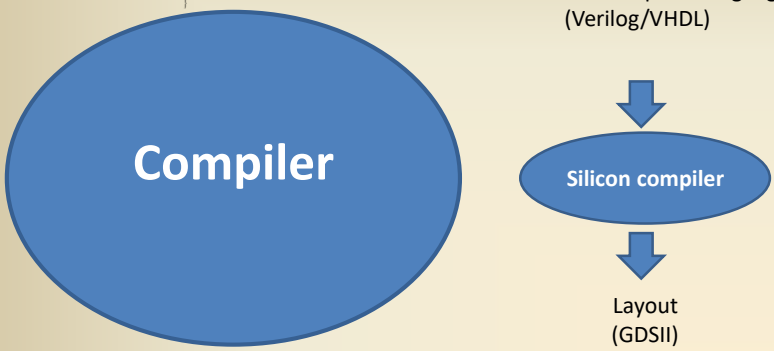
- A category of *software tools* for designing electronic systems such as integrated circuits and printed circuit boards.
- The tools work together in a design flow that chip designers use to design and analyze entire semiconductor chips.
- Since a modern semiconductor chip can have billions of components, *EDA tools are essential for their design*

Silicon Compiler

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Hardware description language (Verilog/VHDL)



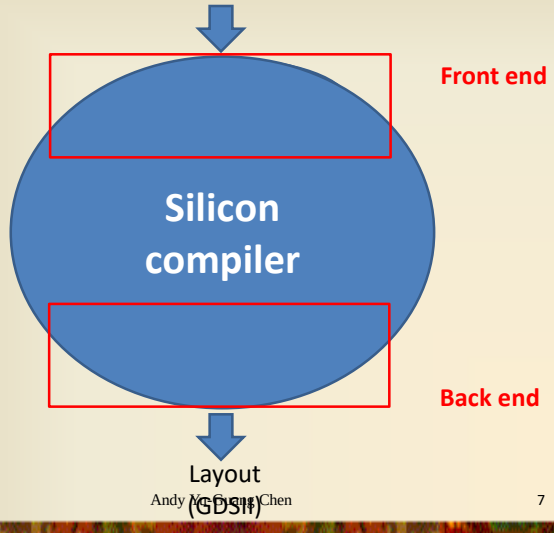
Layout (GDSII)

Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Silicon Compiler

Hardware description language
(Verilog/VHDL)



Four Stages in Creation of an IC

I. DESIGN	III. TESTING
Modeling	Testing for defects
Synthesis & optimization	
Validation	

II. FABRICATION	IV. PACKAGING
Mask fabrication	Slicing
Wafer fabrication	Packaging

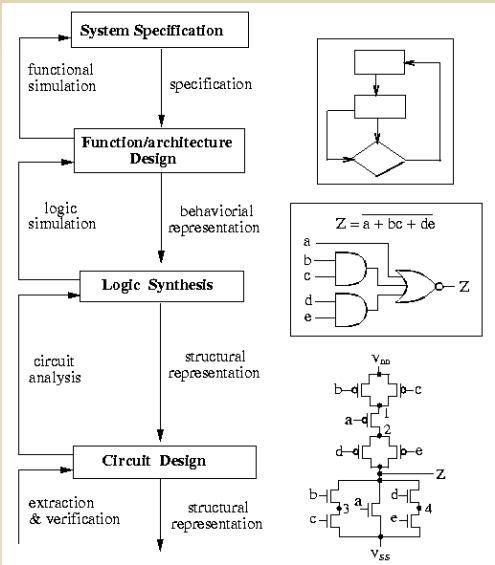
Traditional VLSI Design Cycles

1. System specification
2. Functional design
3. Logic synthesis
4. Circuit design
5. Physical design
6. Fabrication
7. Packaging

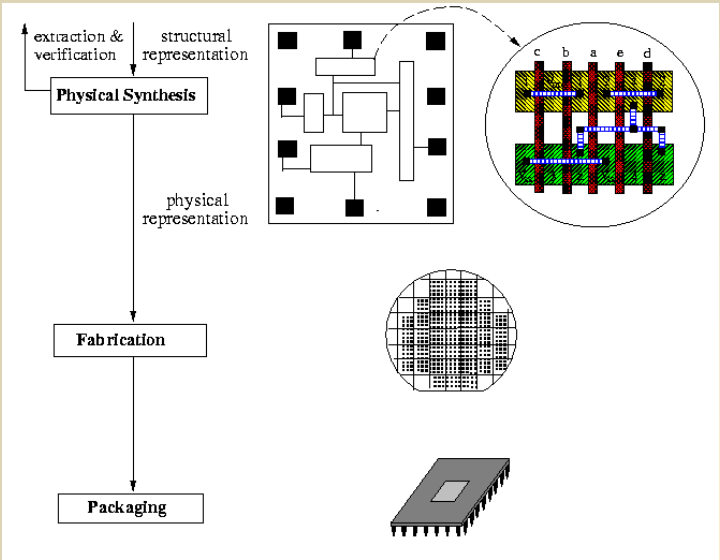
Micron tehcnology => 1um, 2um, 3um, etc
Sub-micron techology => 0.8um, 0.6um, 0.35um 0.25um etc
Deep sub-micro technology => 0.18um, 0.13um
Nanotechnoogy => 90nm, 65nm etc

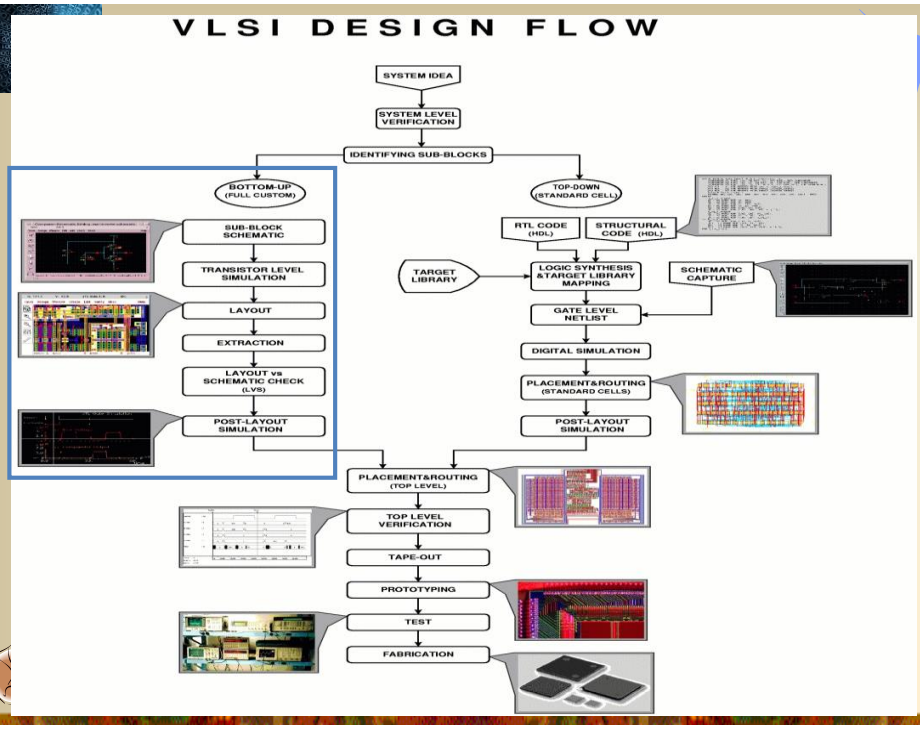
- Other tasks involved: verification, testing, etc.
- Design metrics: area, speed, power dissipation, noise, design time, testability, etc.
- Design revolution: interconnect (not gate) delay dominates circuit performance in deep submicron era.
  - Interconnects are determined in physical design.
  - Shall consider interconnections in early design stages.

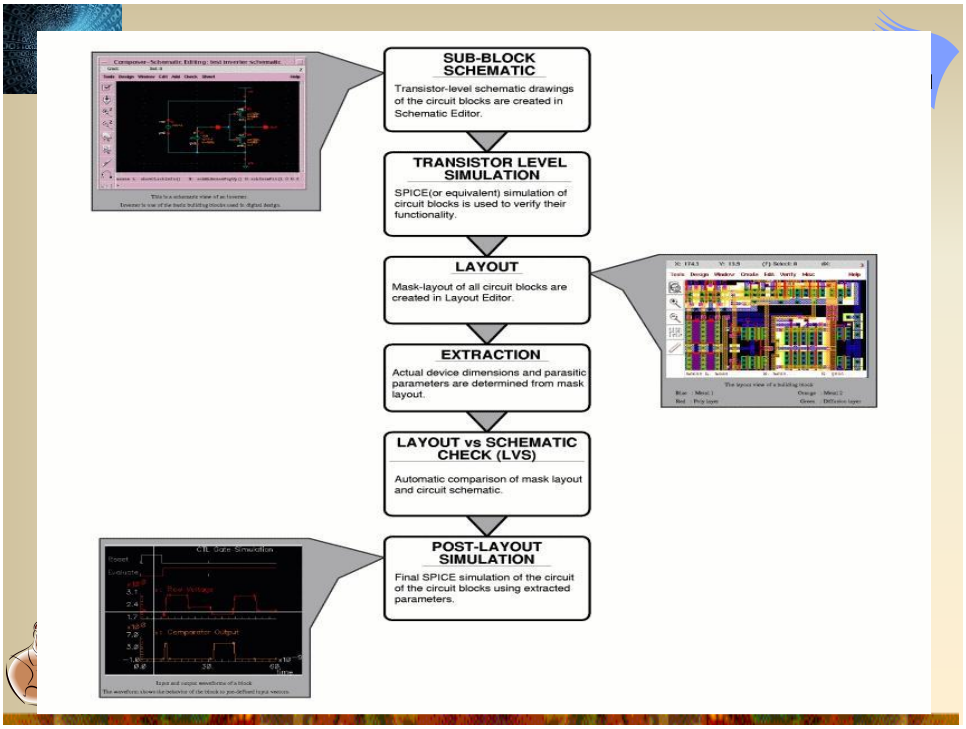
Traditional VLSI Design Cycle

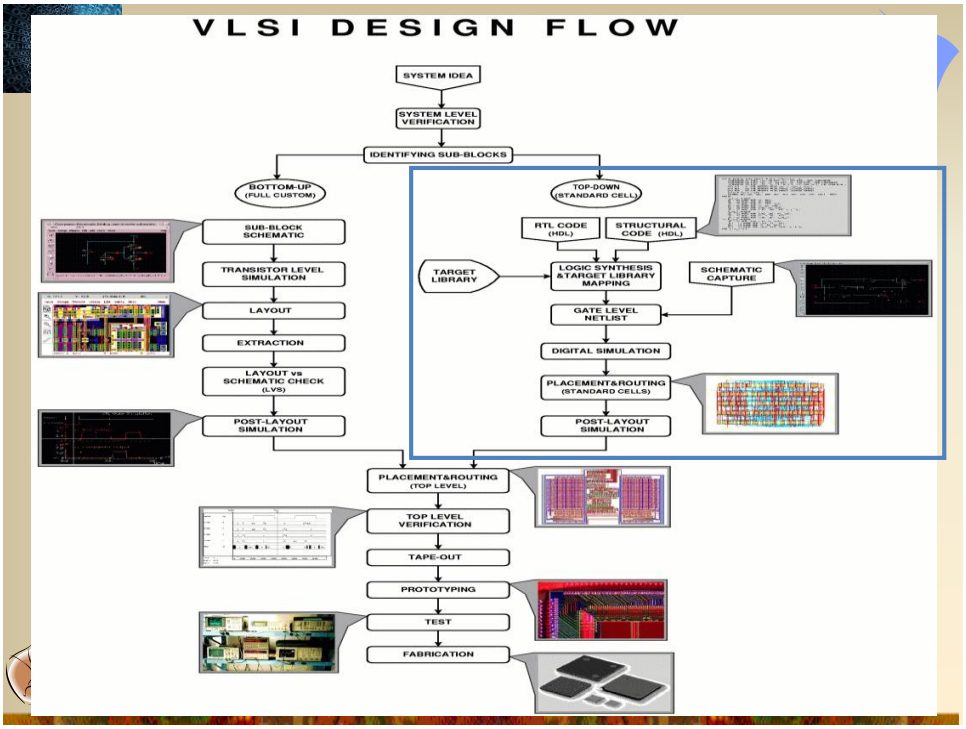


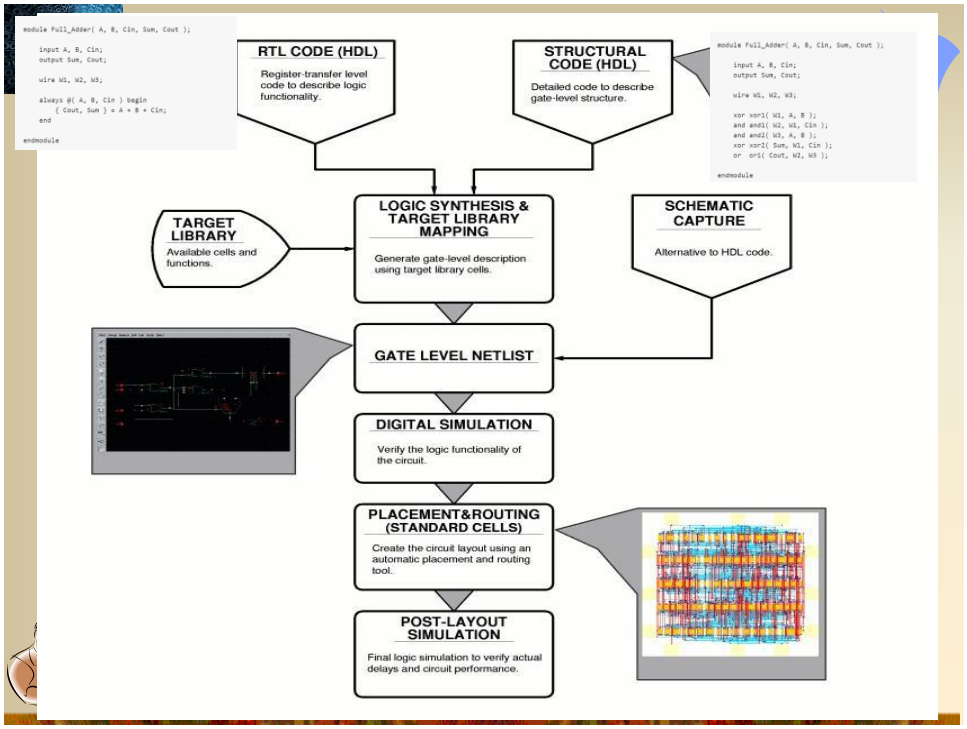
Traditional VLSI Design Flow



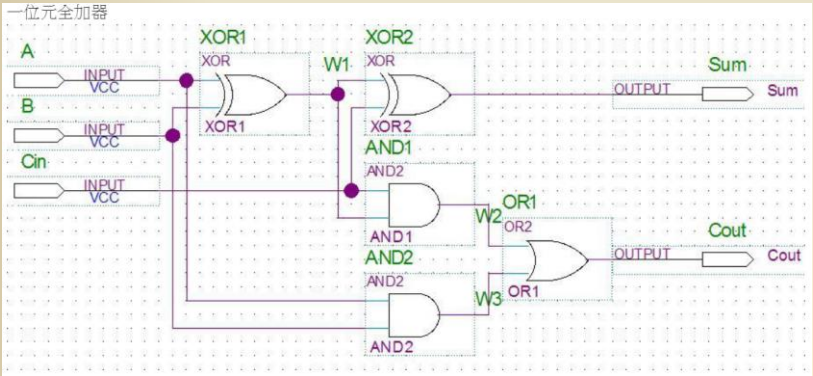


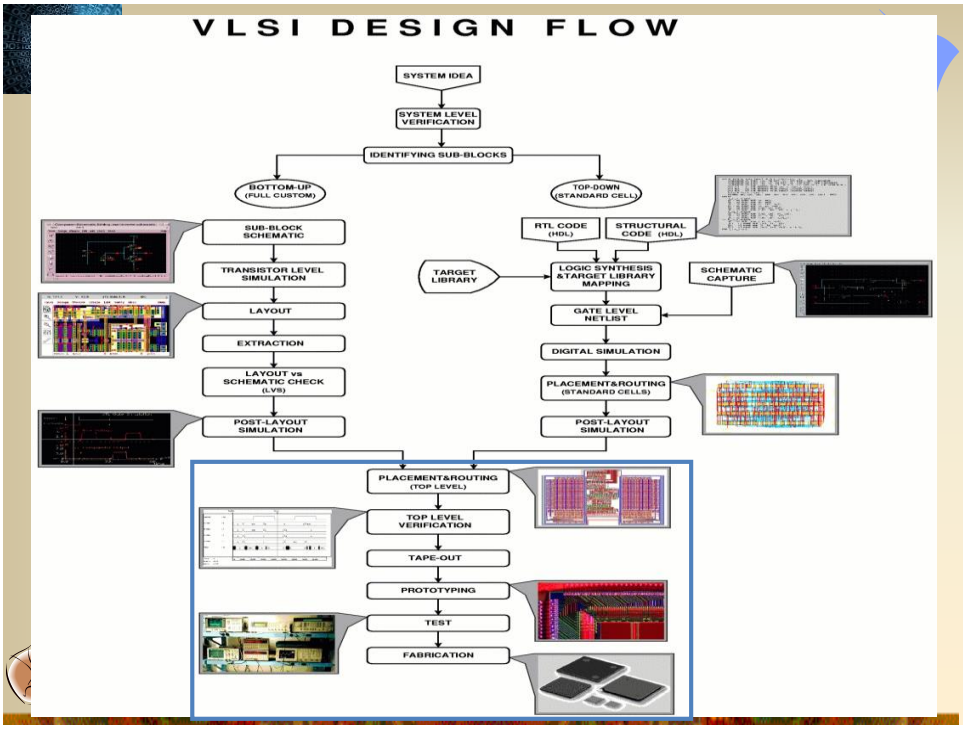

SUB-BLOCK SCHEMATIC
Transistor-level schematic drawings of the circuit blocks are created in Schematic Editor.
TRANSISTOR LEVEL SIMULATION
SPICE(or equivalent) simulation of circuit blocks is used to verify their functionality.
LAYOUT
Mask-layout of all circuit blocks are created in Layout Editor.
EXTRACTION
Actual device dimensions and parasitic parameters are determined from mask layout.
LAYOUT vs SCHEMATIC CHECK (LVS)
Automatic comparison of mask layout and circuit schematic.
POST-LAYOUT SIMULATION
Final SPICE simulation of the circuit of the circuit blocks using extracted parameters.

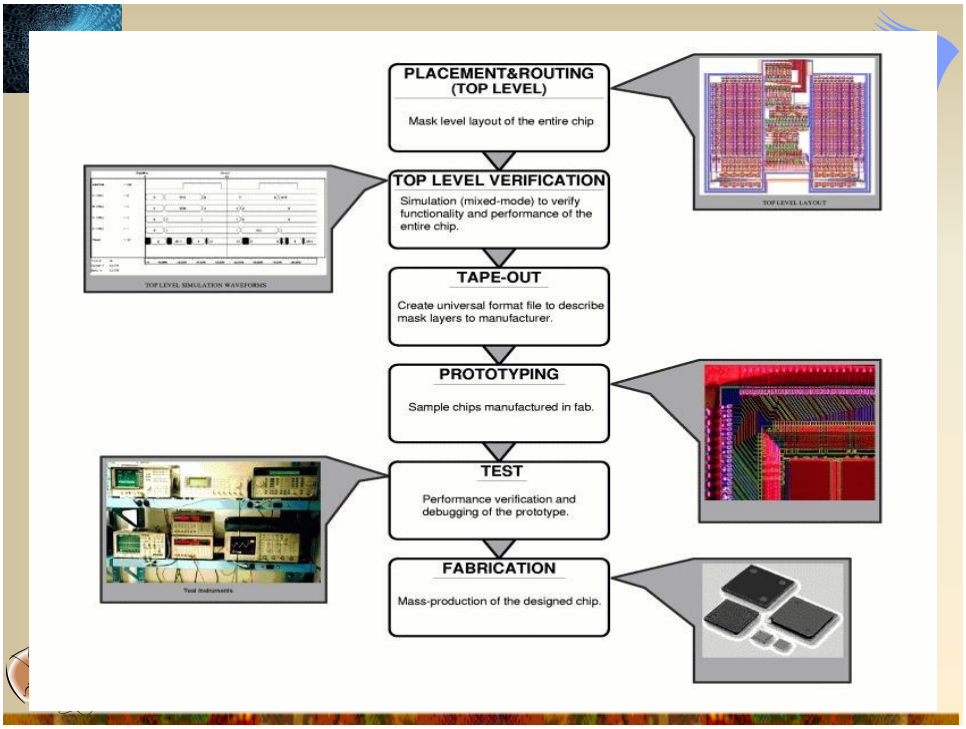

VLSI DESIGN FLOW
SYSTEM IDEA
SYSTEM LEVEL VERIFICATION
IDENTIFYING SUB-BLOCKS
BOTTOM-UP (FULL CUSTOM)
SUB-BLOCK SCHEMATIC
TRANSISTOR LEVEL SIMULATION
LAYOUT
EXTRACTION
LAYOUT vs SCHEMATIC CHECK (LVS)
POST-LAYOUT SIMULATION
TOP-DOWN (STANDARD CELL)
RTL CODE (HDL)
STRUCTURAL CODE (HDL)
TARGET LIBRARY
LOGIC SYNTHESIS &TARGET LIBRARY MAPPING
SCHEMATIC CAPTURE
GATE LEVEL NETLIST
DIGITAL SIMULATION
PLACEMENT&ROUTING (STANDARD CELLS)
POST-LAYOUT SIMULATION
PLACEMENT&ROUTING (TOP LEVEL)
TOP LEVEL VERIFICATION
TAPE-OUT
PROTOTYPING
TEST
FABRICATION



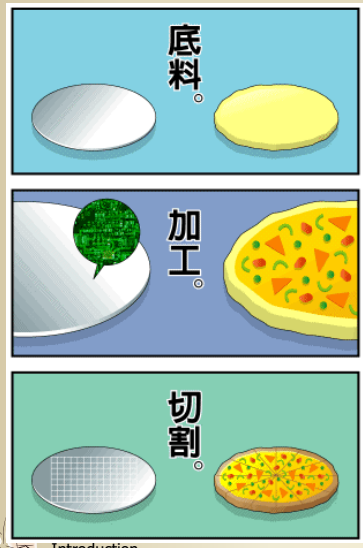
One Bit Full Adder

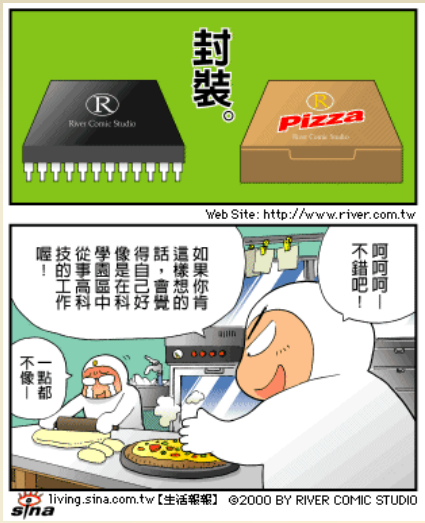


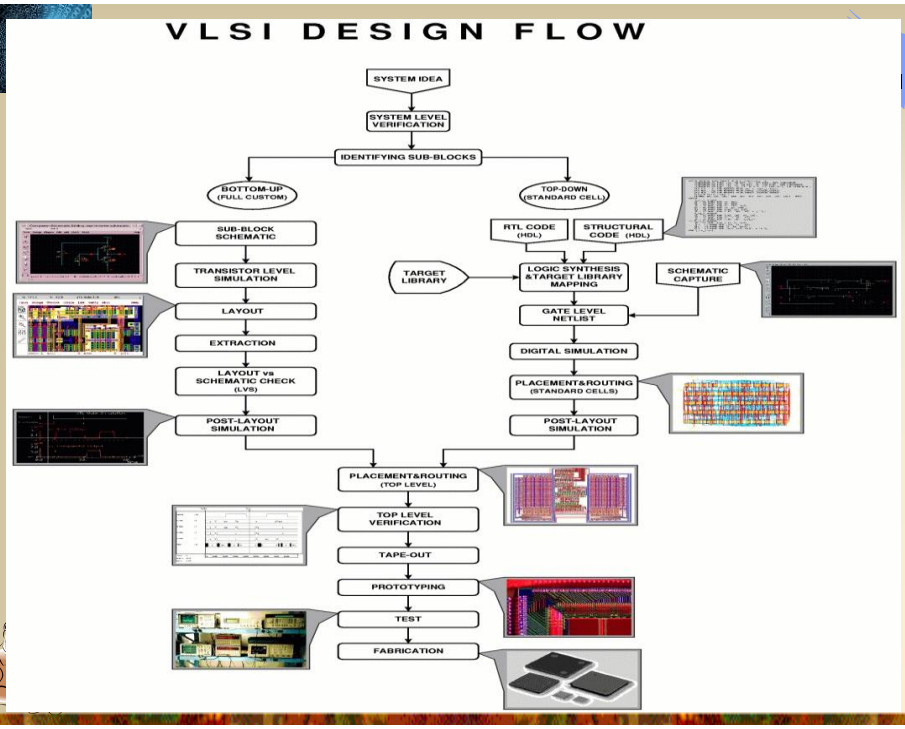

VLSI DESIGN FLOW
SYSTEM IDEA
SYSTEM LEVEL VERIFICATION
IDENTIFYING SUB-BLOCKS
BOTTOM-UP (FULL CUSTOM)
TOP-DOWN (STANDARD CELL)
SUB-BLOCK SCHEMATIC
TRANSISTOR LEVEL SIMULATION
LAYOUT
EXTRACTION
LAYOUT vs SCHEMATIC CHECK (LVS)
POST-LAYOUT SIMULATION
RTL CODE (HDL)
STRUCTURAL CODE (HDL)
TARGET LIBRARY
LOGIC SYNTHESIS &TARGET LIBRARY MAPPING
SCHEMATIC CAPTURE
GATE LEVEL NETLIST
DIGITAL SIMULATION
PLACEMENT&ROUTING (STANDARD CELLS)
POST-LAYOUT SIMULATION
PLACEMENT&ROUTING (TOP LEVEL)
TOP LEVEL VERIFICATION
TAPE-OUT
PROTOTYPING
TEST
FABRICATION


PLACEMENT&ROUTING (TOP LEVEL)
Mask level layout of the entire chip
TOP LEVEL LAYOUT
TOP LEVEL VERIFICATION
Simulation (mixed-mode) to verify functionality and performance of the entire chip.
TOP LEVEL SIMULATION WAVEFORMS
TAPE-OUT
Create universal format file to describe mask layers to manufacturer.
PROTOTYPING
Sample chips manufactured in fab.
TEST
Performance verification and debugging of the prototype.
Test Instruments
FABRICATION
Mass-production of the designed chip.

IC Fabrication







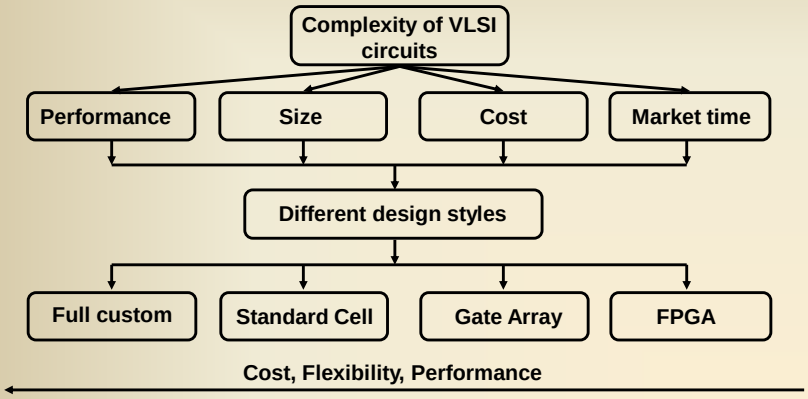
Design Actions

- **Synthesis:** increasing information about the design by providing more detail (e.g., logic synthesis, physical synthesis).
- **Analysis:** collecting information on the quality of the design (e.g., timing analysis).
- **Verification:** checking whether a synthesis step has left the specification intact (e.g., layout verification).
- **Testing:** checking whether a fabricated chip has left all functions intact

Design Actions

- **Optimization:** increasing the quality of the design by rearrangements in a given description (e.g., logic optimizer, timing optimizer).
- **Design Management:** storage of design data, cooperation between tools, design flow, etc. (e.g., database).

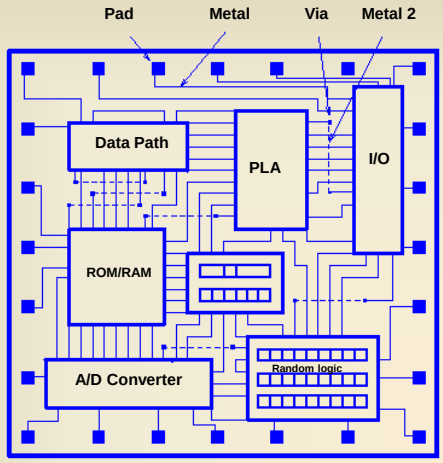
Design Styles



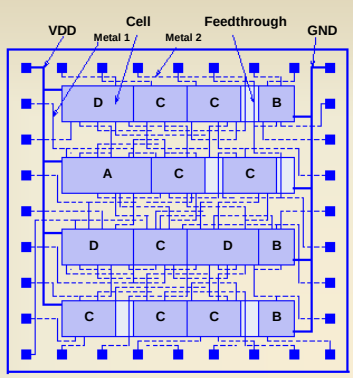
Design Styles

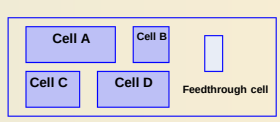
- Full Custom Design Style
  - Design every component from scratch
- Standard Cell Design Style
  - Selects pre-designed cells (of same height)
- Gate Array Design Style
  - Use arrays of prefabricated transistors
  - Needs wiring customization to implement logic
- Field Programmable Gate Arrays (FPGA)
  - Logic and interconnects are both prefabricated
  - Program the logic functions and interconnects

Full Custom Design Style

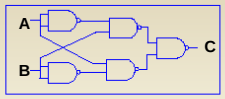


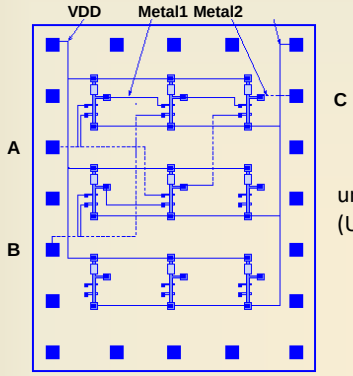
Standard Cell Design Style





Gate Array Design Style

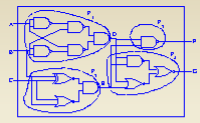


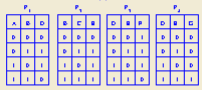


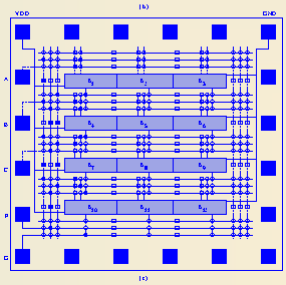
uncommitted logic array (ULA)

Structured ASICs are essentially gate array

FPGA Design Style







Comparisons of Design Styles

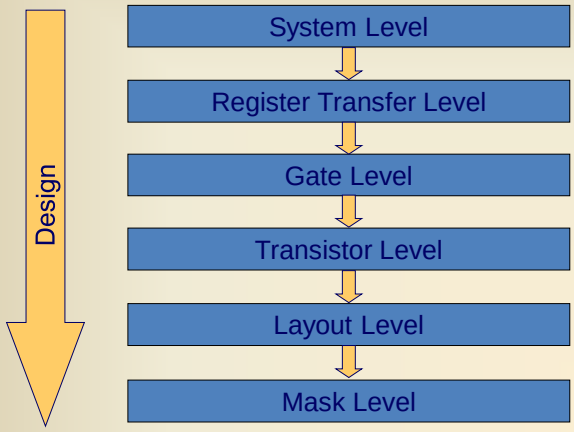
	Style			
	full-custom	standard cell	gate array	FPGA
cell size	variable	fixed height *	fixed	fixed
cell type	variable	variable	fixed	programmable
cell placement	variable	in row	fixed	fixed
interconnections	variable	variable	variable	programmable
design cost	high	medium	medium	low

* uneven height cells may be used

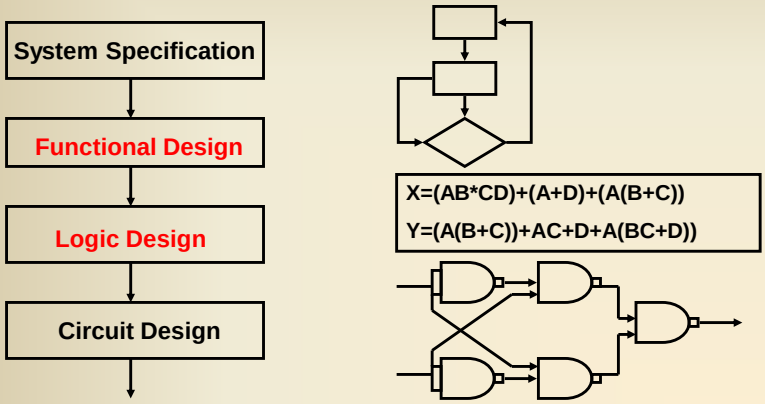
Comparisons of Design Styles

	style			
	full-custom	standard cell	gate array	FPGA
Area	compact	compact to moderate	moderate	large
Performance	high	high to moderate	moderate	low
Fabrication layers	all	all	routing layers	none

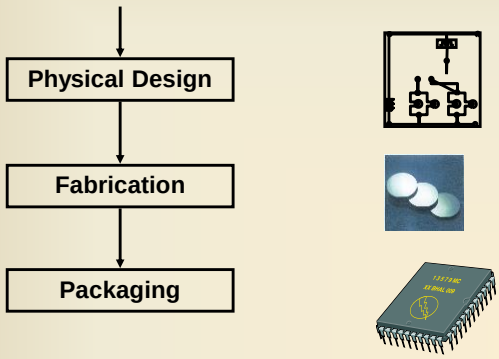
Design of Integrated Systems



VLSI Design Cycle



VLSI Design Cycle (cont.)



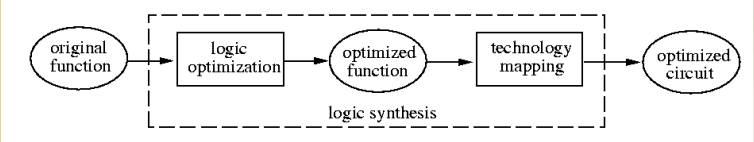
Focus of this Course (1)

- CAD tools for synthesis and verification at logic-level of abstraction
- Theory behind: functions representation and manipulation
  - representation ⇔ data structures
  - manipulation ⇔ algorithms
- In-depth course:
  - You should be able create a small CAD-tool

Why Logic Level?

- Logic-level synthesis is the core of today's CAD flows for IC and system design
  - course covers many algorithms that are used in a broad range of CAD tools
  - basis for other optimization techniques
  - basis for functional verification techniques
- Most algorithms are computationally hard
  - covered algorithms and flows are good example for approaching hard algorithmic problems
  - course covers theory as well as implementation details
  - demonstrates an engineering approaches based on theoretical solid but also practical solutions
    - very few research areas can offer this combination

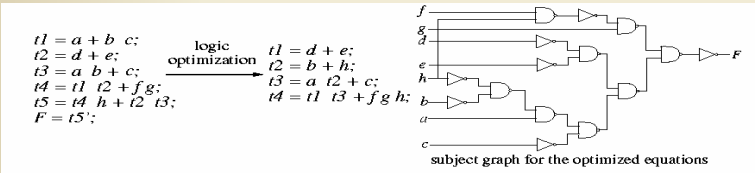
Logic Design/Synthesis



- **Logic synthesis** programs transform Boolean expressions into logic gate networks in a particular library.
- Optimization goals: minimize area, delay, power, etc
- **Technology-independent** optimization: logic optimization
  - Optimizes Boolean expression equivalent.
- **Technology-dependent** optimization: **technology mapping/library binding**
  - Maps Boolean expressions into a particular cell library.

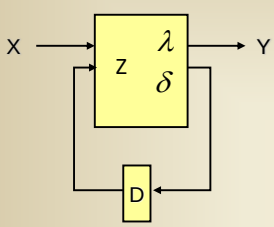
Logic Optimization Examples

- **Two-level:** minimize the # of product terms.
  - $F = \bar{x_1}\bar{x_2}\bar{x_3} + \bar{x_1}\bar{x_2}x_3 + x_1\bar{x_2}\bar{x_3} + x_1\bar{x_2}x_3 + x_1x_2\bar{x_3} \Rightarrow F = \bar{x_2} + x_1\bar{x_3}$.
- **Multi-level:** minimize the #'s of literals, variables.
  - E.g., equations are optimized using a smaller number of literals.



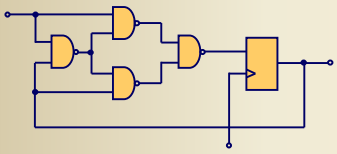
- Methods/CAD tools: Quine-McCluskey method (exponential-time exact algorithm), Espresso (heuristics for two-level logic), MIS (heuristics for multi-level logic), Synopsys, etc.

Logic Optimization Examples



Given: Finite-State Machine F(X,Y,Z,λ,δ) where:

X: Input alphabet
Y: Output alphabet
Z: Set of internal states
λ : X × Z → Y (output function)
δ : X × Z → Z' (next state function)



Target: Circuit C(G, W) where
G: set of circuit components g (Boolean gates flip-flops, etc}
W: set of wires connecting G

Finite State Machine

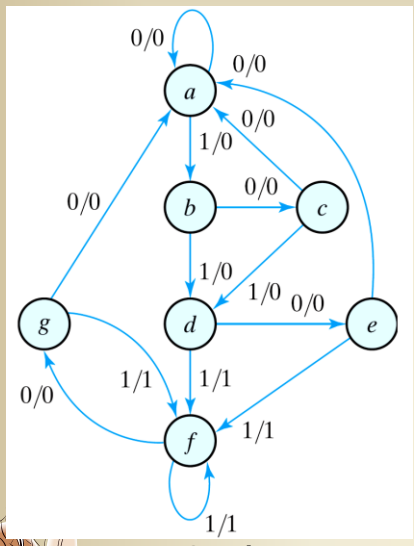


Fig. 5.25 State diagram

State:	**a**										
Input:	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**0**
Output:											

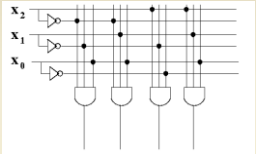
Typical Synthesis Scenario

RTL to Network Transformation	- Read Verilog/VHDL - Control/data flow analysis
Technology Independent Optimizations	- Basic logic restructuring - Crude measures for goals
Technology Mapping	- Use logic gates from target cell library
Technology Dependent Optimizations	- Timing optimization - Physically driven optimizations
Test Preparation	- Improve testability - Test logic insertion

Objective Function for Synthesis

- Minimize area
  - in terms of literal count, cell count, register count, etc.
- Minimize power
  - in terms of switching activity in individual gates, blocks, etc.
- Maximize performance
  - in terms of maximal clock frequency of synchronous systems, throughput for asynchronous systems
- Any combination of the above
  - combined with different weights
  - formulated as a constraint problem
    - Ex: minimize area for a clock speed > 300MHz

Constraints on Synthesis



- Given implementation style:
  - two-level implementation (PLA)
  - multi-level logic
  - FPGAs
- Given performance requirements
  - minimal clock speed requirement
- Given cell library
  - set of cells in standard cell library
  - fan-out constraints (maximum number of gates connected to another gate)

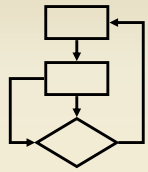
VLSI Design Cycle

System Specification

Functional Design

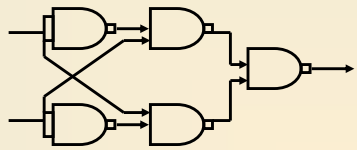
Logic Design

Circuit Design



X=(AB*CD)+(A+D)+(A(B+C))

Y=(A(B+C))+AC+D+A(BC+D))

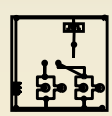


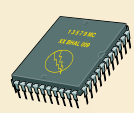
VLSI Design Cycle (cont.)

Physical Design

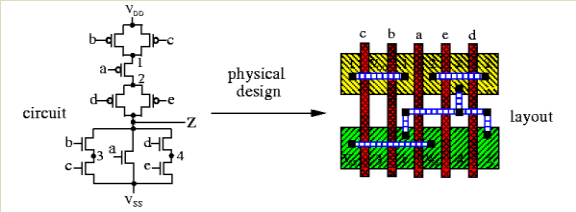
Fabrication

Packaging



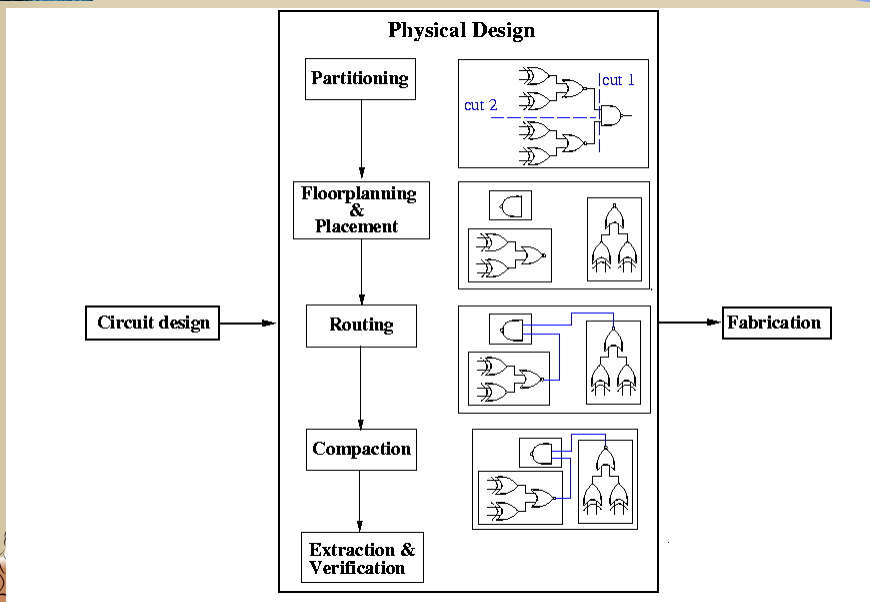


Physical Design



- Physical design converts a circuit description into a geometric description.
- The description is used to manufacture a chip.
- Physical design cycle:
  1. Logic partitioning
  2. Floorplanning and placement
  3. Routing
  4. Compaction
- Others: circuit extraction, timing verification and design rule checking

Physical Design Flow



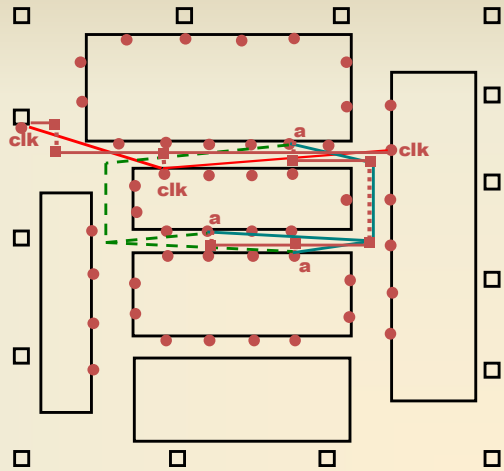
Physical Design Process

Design Steps:

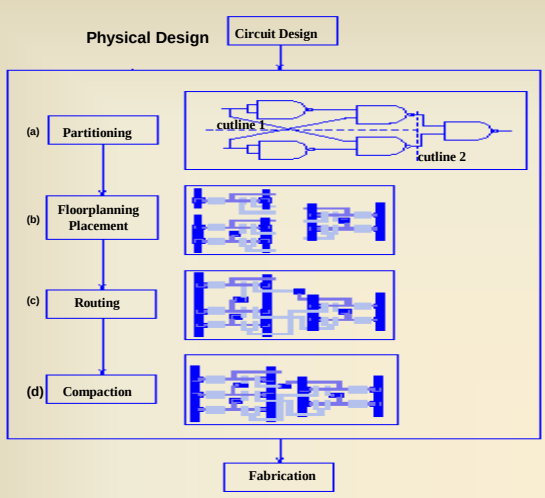
- Partition & Clustering
- Floorplan & Placement
  - Pin Assignment
- Global Routing
- Detailed Routing

Methodology:

- Divide-and-Conquer

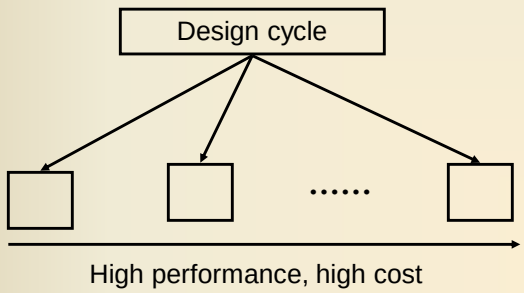


Physical Design Cycle



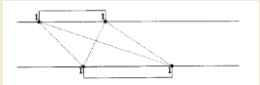
Complexities of Physical Design

- More than 100 million transistors
- Performance driven designs
- Power-constrained designs
- Time-to-Market



History of Physical Design

- Born in early 60's (board layout)
- Passed teenage in 70's (standard cell place and route)
- Entered early adulthood in 80's (over-the-cell routing)
- Declared dead in late 80's !!!
- Found alive and kicking in 90's
- Physical Design (PD) has become a dominant force in overall design cycle,
  - thanks to the deep submicron scaling
  - expand vertically with logic synthesis and interconnect optimization, analysis…. => Design closure!



Design Closure

- A part of the development workflow by which an integrated circuit design is modified from its initial description to meet a growing list of design constraints and objectives
- Looks at the overall design closure process, which takes a chip from its initial design state to the final form in which all of its design constraints are met.

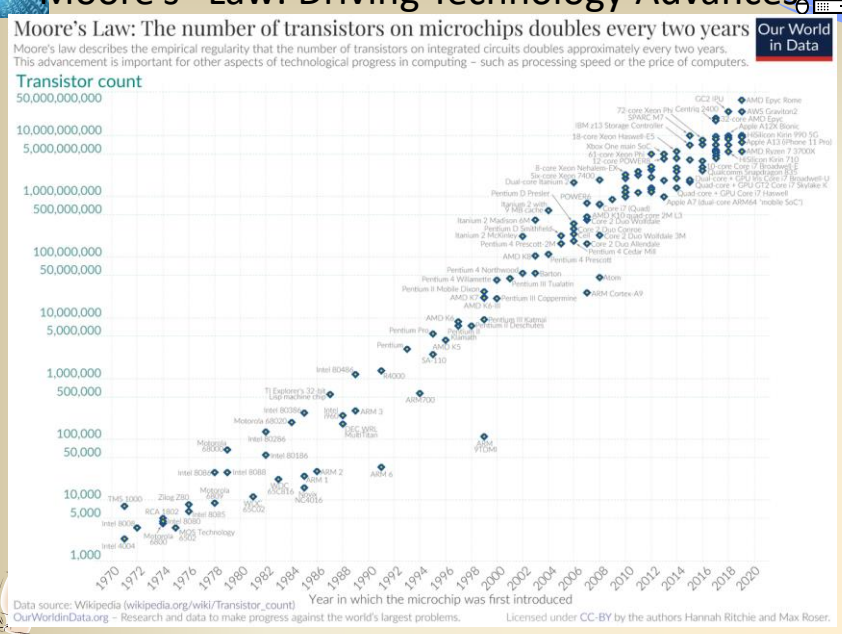
Focus of this Course (2)

- Many existing solutions are still very suboptimal
  - Ex: placement
- Interconnect dominates
  - No physical layout, no accurate interconnect
- More new physical and manufacturing effects pop up
  - Crosstalk noise, Electromigration, …
  - Optical Proximity Correction, OPC (manufacturability), etc.
- More vertical integration needed
- Physical design is the KEY linking step between higher level planning/optimization and lower level modeling

Moore's Law

- Logic capacity doubles per IC at a regular interval.
- Moore: Logic capacity doubles per IC every two years (1975).
- D. House: Computer performance doubles every 18 months (1975)
- Consequences of smaller transistors:
  - Faster transistor switching
  - More transistors per chip
- True for 40+ years!
- And it will be true in at least another 10 years
  - Need smarter and more powerful CAD tools than ever

"Moore's" Law: Driving Technology Advances



Moore's Law

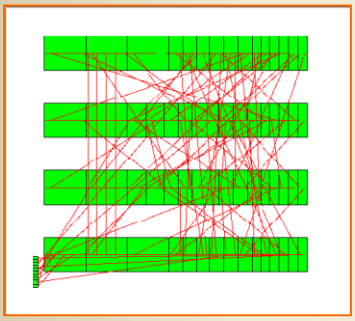
Moore's Law

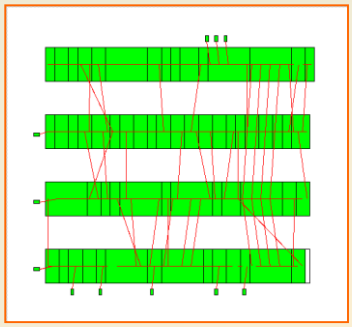
Placement Challenge

- Placement, to large extend, determines the overall interconnect
- If it sucks, no matter how well you interconnect optimization engine works, the design will suck
- Placement is a very old problem, but got renewed interest
  - Mixed-size (large macro blocks and small standard cells)
  - Optimality study shows that placement still a bottleneck
  - Not even to mention performance driven, and coupled with buffering, interconnect optimizations, and so on (all you name)

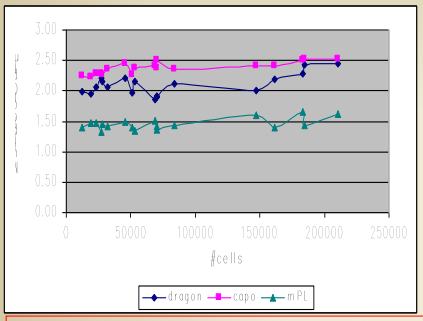
VLSI Global Placement Examples

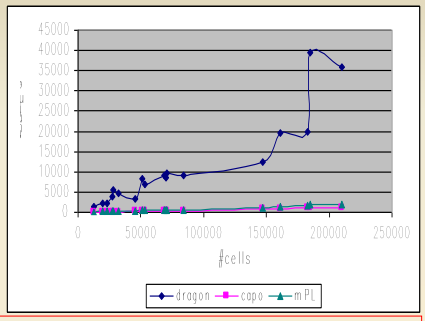
- Which one of the two placement results is better?





Comparison with Optimal



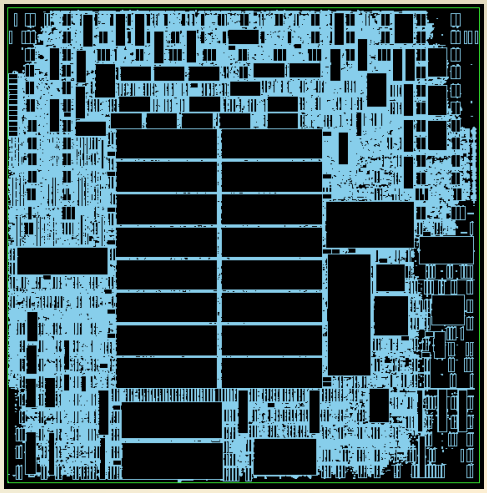


- Capo: Based on recursive min-cut (UCLA-UMich)
- Dragon: Recursive min-cut + SA refinement at each level (NWU-UCLA)
- mPL: multi-level placer (UCLA)

There is significant room for improvement in placement algorithms: existing algorithms are 50-150% away from optimal!

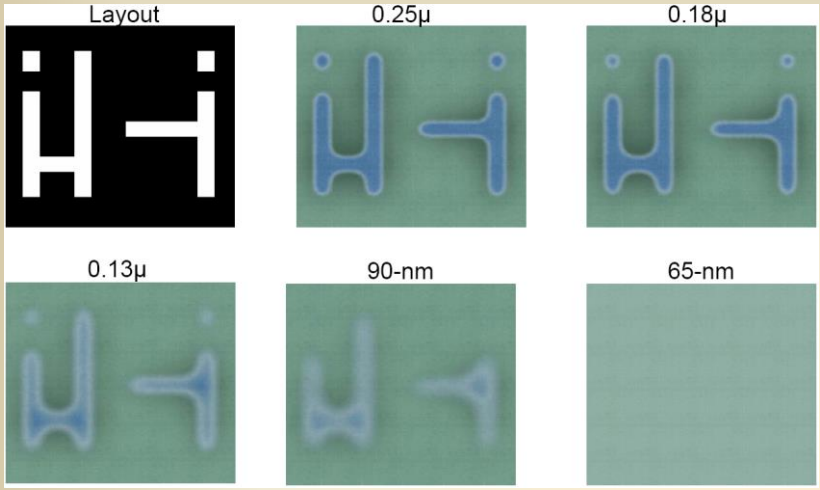
FloorPlacer (Mix-mode Placement)

- Many macros
- data paths + dust logic
- I/O constraint (area I/O or wirebond)

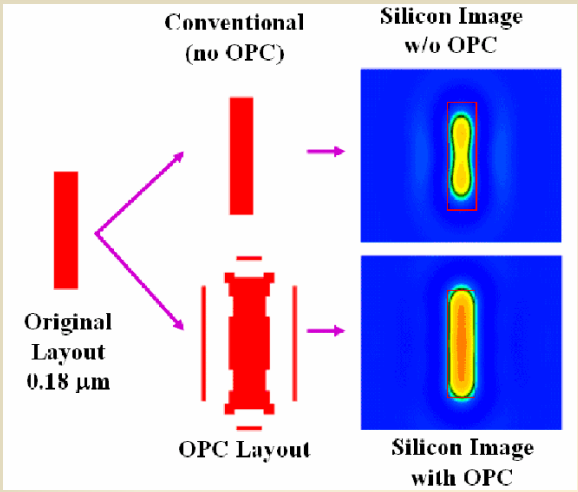


(source: IBM)

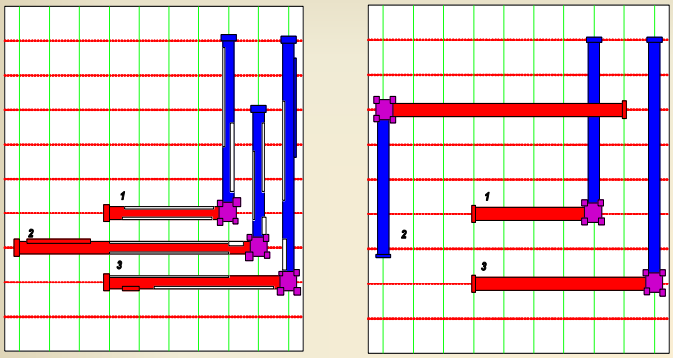
Lithography



Optical Proximity Correction (OPC)



OPC-Aware Routing



More OPC friendly

Design Issues and Tools

- System-level design
  - Partitioning into hardware and software, co-design, co-simulation, etc.
  - Cost estimation, design-space exploration
- Algorithmic-level design
  - Behavioral descriptions (e.g. in Verilog, VHDL)
  - High-level simulation
- From algorithms to hardware modules
  - High-level (or architectural) synthesis
- Logic design:
  - Schematic entry
  - Register-transfer level and logic synthesis
  - Gate-level simulation (functionality, power, etc)
  - Timing analysis
  - Formal verification

Design Issues and Tools (Cont'd)

- Transistor-level design
  - Switch-level simulation
  - Circuit simulation
- Physical (layout) design:
  - Partitioning
  - Floorplanning and Placement
  - Routing
  - Compaction
  - Layout editing
  - Design-rule checking
  - Layout extraction
- Design management
  - Data bases, frameworks, etc.
- Silicon compilation: *from algorithm to mask patterns*
  - The *idea* is approached more and more, but still far away from a single *push-button* operation

We Need Algorithms

- To optimize design among different objectives, area, power, performance, and etc.
- Fundamental questions: How to do it smartly?
- Definition of algorithm in a board sense: A step-by-step procedure for solving a problem. Examples:
  - Cooking a dish
  - Making a phone call
  - Sorting a hand of cards
- Definition for computational problem: A well-defined computational procedure that takes some value as input and produces some value as output

Computational Problem

- A mathematical object representing a collection of questions that computers might be able to solve
  - decision problem
    - Given a positive integer n, determine if n is prime
  - search problem
    - Find all even numbers in X=[1,2,3,4,5]
  - counting problem
    - Given a positive integer n, count the number of nontrivial prime factors of n.
  - optimization problem
    - Given a graph G, find an independent set of G of maximum size

Computational Problem

- A mathematical object representing a collection of questions that computers might be able to solve
  - function problem
    - a single output (of a total function) is expected for every input, but the output is more complex than that of a decision problem
    - travelling salesman: Given a list of cities and the distances between each pair of cities, find the shortest possible route that visits each city exactly once and returns to the origin city.

Computational Complexity

- Computational complexity is an abstract measure of the time and space necessary to execute an algorithm as function of its input size
  - The input is the graph G(V,E)
    - input size = $|V|$ and $|E|$
  - The input is the truth table of an n-variable Boolean function
    - input size = 2^n

Time and Space Complexity

- Time complexity is expressed in elementary computational steps
  - example: addition (or multiplication, or value assignment etc.) is one step
  - normally, by "most efficient" algorithm we mean the fastest
- Space complexity is expressed in memory locations
  - e.g. in bits, bytes, words

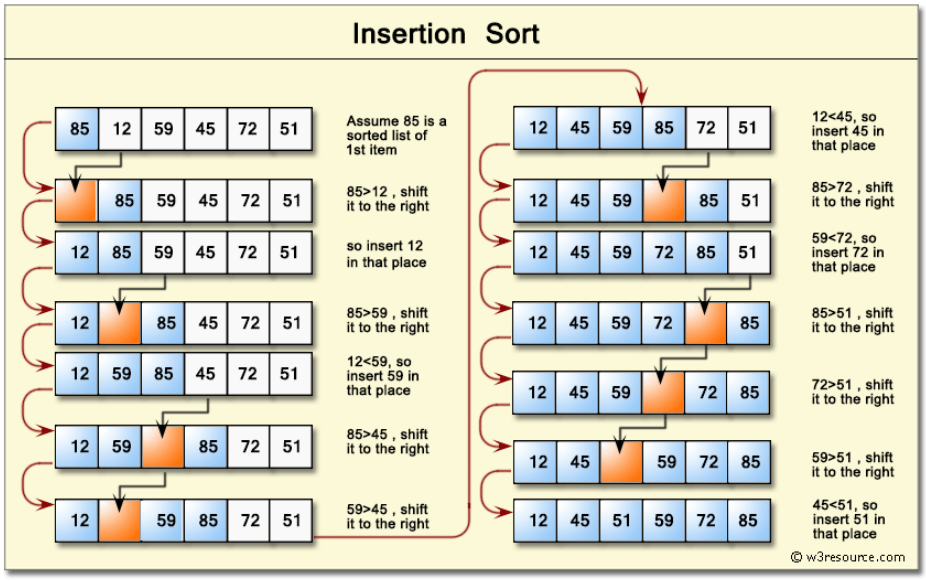
Example: Different Sorting Algorithms

- Input: An array of n numbers D[1]…D[n]
- Output: An array of n numbers E[1]…E[n] such that E[1] ≥E[2] ≥… ≥E[n]

- Solution I : Insertion sort
- Solution II: Selection sort

Sorting Arrays with Insertion Sort

- Sorting is to place the data into some particular order such as ascending or descending.
  - An important problem with many applications in computer science.
- Insertion sort—a simple, but inefficient, sorting algorithm.
- The first iteration of this algorithm looks at the second element and, if it's less than the first element, insert the second element in front of the first element.
- The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements.
  - All three elements are in order.
- At the ith *iteration of this algorithm, the first i elements in the original array will be sorted.*



Sorting Arrays with Insertion Sort

```
// Fig. 6.16: fig06_16.cpp
// This program sorts an array's values into ascending order.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   const int arraySize = 10; // size of array a
   int data[ arraySize ] = { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
   int insert; // temporary variable to hold element to insert

   cout << "Unsorted array:\n";

   // output original array
   for ( int i = 0; i < arraySize; i++ )
      cout << setw( 4 ) << data[ i ];

```

Fig. 6.16 | Sorting an array with insertion sort. (Part 1 of 3.)

Sorting Arrays with Insertion Sort

```
   // insertion sort
   // loop over the elements of the array
   for ( int next = 1; next < arraySize; next++ )
   {
      insert = data[ next ]; // store the value in the current element

      int moveItem = next; // initialize location to place element

      // search for the location in which to put the current element
      while ( ( moveItem > 0 ) && ( data[ moveItem - 1 ] > insert ) )
      {
         // shift element one slot to the right
         data[ moveItem ] = data[ moveItem - 1 ];
         moveItem--;
      } // end while

      data[ moveItem ] = insert; // place inserted element into the array
   } // end for

   cout << "\nSorted array:\n";

```

Fig. 6.16 | Sorting an array with insertion sort. (Part 2 of 3.)

Sorting Arrays with Insertion Sort

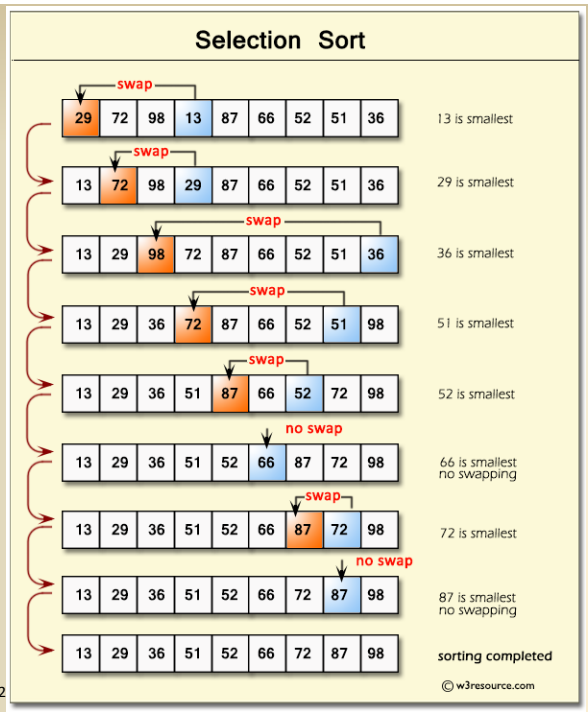
```
   // output sorted array
   for ( int i = 0; i < arraySize; i++ )
      cout << setw( 4 ) << data[ i ];

   cout << endl;
} // end main
```

```
Unsorted array:
  34  56   4  10  77  51  93  30   5  52
Sorted array:
   4   5  10  30  34  51  52  56  77  93
```

Fig. 6.16 | Sorting an array with insertion sort. (Part 3 of 3.)

Sorting Arrays with Selection Sort

- Easy-to-program, but inefficient, sorting algorithm.
- The first iteration of the algorithm selects the smallest element in the array and swaps it with the first element.
- The second iteration selects the smallest element of the remaining elements and swaps it with the second element.
- The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index.



Sorting Arrays with Selection Sort

```
// Fig. 7.13: fig07_13.cpp
// Selection sort with pass-by-reference. This program puts values into an
// array, sorts them into ascending order and prints the resulting array.
#include <iostream>
#include <iomanip>
using namespace std;

void selectionSort( int * const, const int ); // prototype
void swap( int * const, int * const ); // prototype

int main()
{
   const int arraySize = 10;
   int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   cout << "Data items in original order\n";

   for ( int i = 0; i < arraySize; i++ )
      cout << setw( 4 ) << a[ i ];

   selectionSort( a, arraySize ); // sort the array

   cout << "\nData items in ascending order\n";

```

Fig. 7.13 | Selection sort with pass-by-reference. (Part 1 of 3.)

Sorting Arrays with Selection Sort

```
   for ( int j = 0; j < arraySize; j++ )
      cout << setw( 4 ) << a[ j ];

   cout << endl;
} // end main

// function to sort an array
void selectionSort( int * const array, const int size )
{
   int smallest; // index of smallest element

   // loop over size - 1 elements
   for ( int i = 0; i < size - 1; i++ )
   {
      smallest = i; // first index of remaining array

      // loop to find index of smallest element
      for ( int index = i + 1; index < size; index++ )

         if ( array[ index ] < array[ smallest ] )
            smallest = index;

      swap( &array[ i ], &array[ smallest ] );
   } // end if
} // end function selectionSort
```

Fig. 7.13 | Selection sort with pass-by-reference. (Part 2 of 3.)

Sorting Arrays with Selection Sort

```
// swap values at memory locations to which
// element1Ptr and element2Ptr point
void swap( int * const element1Ptr, int * const element2Ptr )
{
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
} // end function swap
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Fig. 7.13 | Selection sort with pass-by-reference. (Part 3 of 3.)

Analysis of Algorithm

- There can be many different algorithms to solve the same problem
- Need some way to compare 2 algorithms
- Usually the run time is the criteria used
- However, difficult to compare since algorithms may be implemented in different machines, use different languages, etc.
- Also, run time is input-dependent. Which input to use?
- Big-O notation is used

Big-O Notation

- Consider run time for the worst input
  - upper bound on run time
- Express run time as a function input size n
- Interested in the run time for large inputs
- Therefore, interested in the growth rate
- Ignore multiplicative constant
- Ignore lower order terms

Big-O Notation

- $f = O(g)$, if two constants n_0 and K can be found such that for all $n \geq n_0$:

$$f(n) \leq K \cdot g(n)$$

- Examples:

  $2n^2 = O(n^2)$

  $2n^2 + 3n + 1 = O(n^2)$

  $n^{1.1}$+10000000000n is O($n^{1.1}$)

  $n^{1.1}$ = O(n^2)

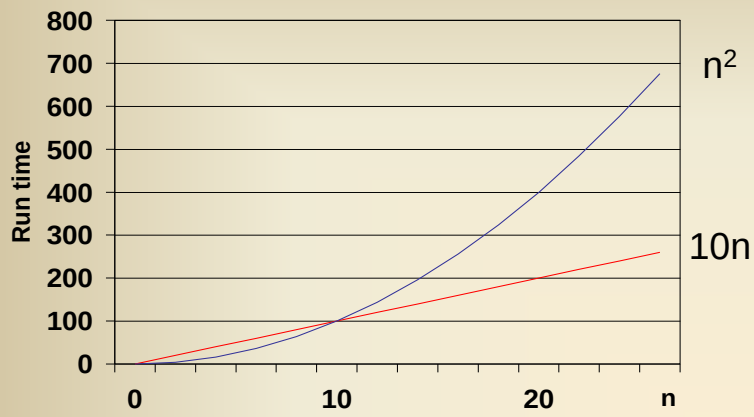
Sorting Algorithm

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Some Algorithm Design Techniques

- Greedy
- Divide and Conquer
- Dynamic Programming
- Network Flow
- Mathematical Programming (ex: linear programming, integer linear programming, quadratic programming, and etc.)

Effect of Multiplicative Constant



Growth Rates of some Functions

$$O(\log n) < O(\log^2 n) < O(\sqrt{n}) < O(n)$$
$$< O(n \log n) < O(n \log^2 n) < O(n^{1.5}) < O(n^2)$$
$$< O(n^3) < O(n^4)$$

Polynomial Functions

$$O(n^c) = O(2^{c \log n}) \text{ for any constant } c$$
$$< O(n^{\log n}) = O(2^{\log^2 n})$$
$$< O(2^n) < O(3^n) < O(4^n)$$
$$< O(n!) < O(n^n)$$

Exponential Functions

Problem of Exponential Function

- Consider 2^n, value doubled when n is increased by 1.

n	2^n	$1\mu s \times 2^n$
10	10^3	0.001 s
20	10^6	1 s
30	10^9	16.7 mins
40	10^{12}	11.6 days
50	10^{15}	31.7 years
60	10^{18}	31710 years

- If you borrow $10 from a credit card with APR 18%, after 40 yrs, you will owe $12700!

Exponential Time Complexity

- An algorithm has an exponential time complexity if its execution time is given by the formula

$$\text{execution time} = k_1 \cdot (k_2)^n$$

where *n* is the size of the input data and k_1, k_2 are constants

Exponential Time Complexity

- The execution time grows so fast that even the fastest computers cannot solve problems of practical sizes in a reasonable time
- The problem is called intractable if the best algorithm known to solve this problem requires exponential time
- Many CAD problems are intractable

Solution Type of Algorithms

- Polynomial time algorithms
- Exponential time algorithms
- Special case algorithms
- Approximation algorithms
- Heuristic algorithms

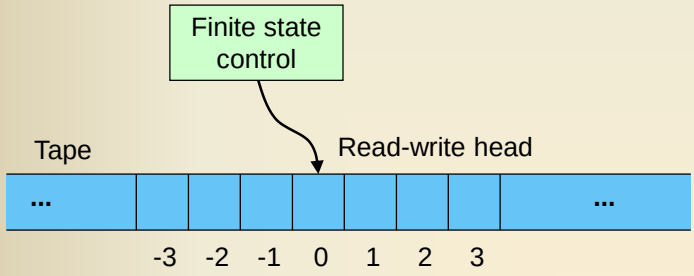
Complexity Classes

- Class P contains those problems that can be solved in polynomial time (the number of computation steps necessary can be expressed as a polynomial of the input size n).
- The computer concerned is a deterministic Turing machine

Deterministic Turing Machine

- Turing machine is a mathematical model of a universal computer
- Any computation that needs polynomial time on a Turing machine can also be performed in polynomial time on any other machine
- Deterministic means that each step in a computation is predictable (only one possible solution)

Deterministic One-tape Turing Machine



Undecidable Decision Problem

- The Halting Problem
  - The problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running (i.e., halt) or continue to run forever.

```
num ← 10
REPEAT UNTIL (num = 0) {
  DISPLAY(num)
  num ← num - 1
}
```

That program will halt, since `num` eventually becomes 0.

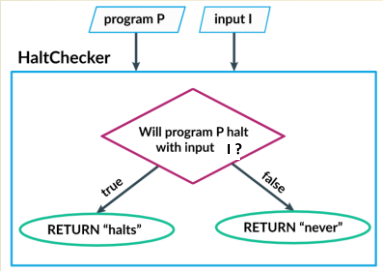
```
num ← 1
REPEAT UNTIL (num = 0) {
  DISPLAY(num)
  num ← num + 1
}
```

It counts up *forever*, since `num` will never equal 0.

Source: https://www.khanacademy.org/computing/ap-computer-science-principles/algorithms-101/solving-hard-problems/a/undecidable-problems

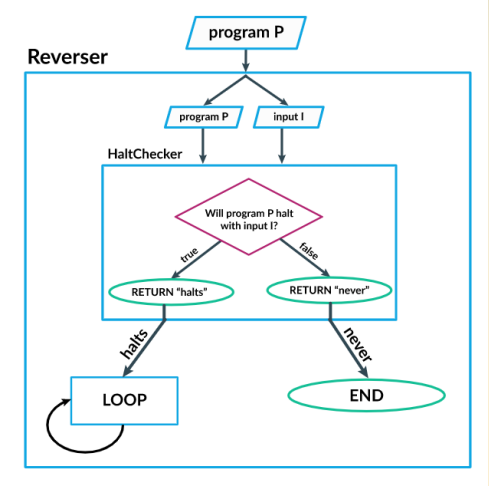
Undecidable Decision Problem

- Proof of contradiction
- We propose a program called HaltChecker
  - takes two inputs, a program's code and an input for that program.
  - It then uses that hypothetical haltability algorithm to return either "halts" or "never".

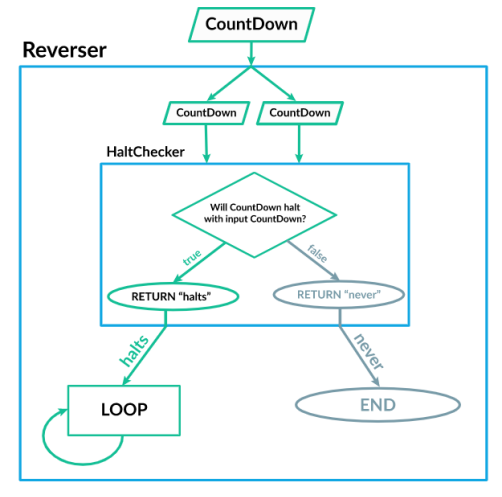


Undecidable Decision Problem

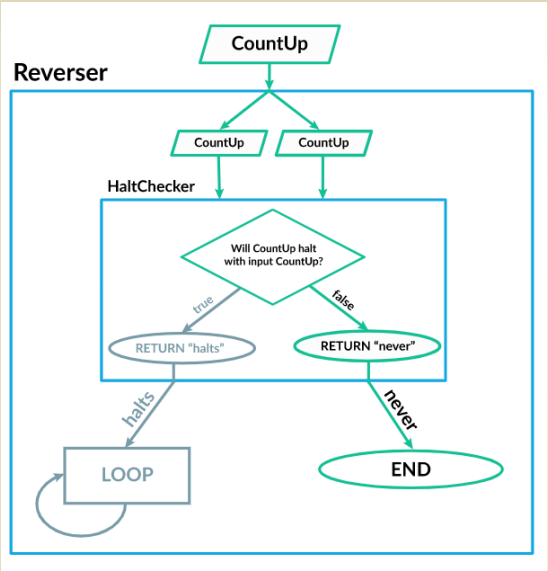
- We propose a program called Reverser



Undecidable Decision Problem

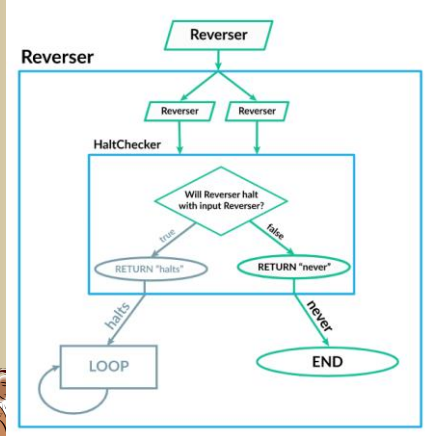


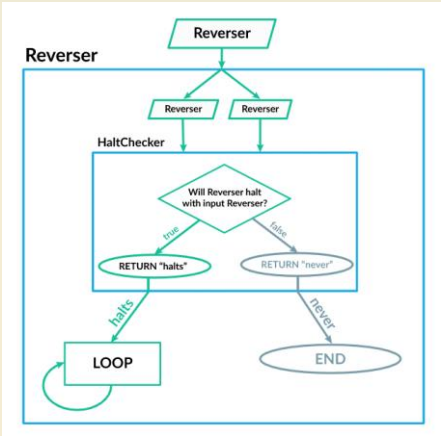
Undecidable Decision Problem



Undecidable Decision Problem

- What happens if we input Reverser itself into Reverser?





Undecidable Decision Problem

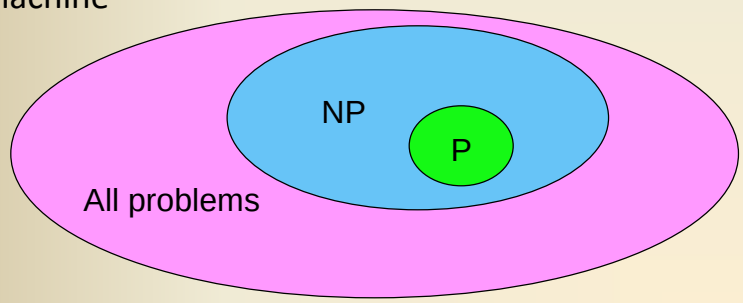
- For the left figure, HaltChecker just claimed that Reverser never halts, and then it went ahead and halted.
  - HaltChecker did not give us a correct answer.
- For the right figure, HaltChecker just claimed that Reverser halts, and yet, it went on forever.
  - Once again, HaltChecker did not give us a correct answer. I

Non-deterministic Turing Machine

- If solution checking for some problem can be done in polynomial time on a deterministic machine, then the problem can be solved in polynomial time on a non-deterministic Turing machine
- non-deterministic - 2 stages:
  - make a guess what the solution is
  - check whether the guess is correct

NP-class

- Class NP contains those problems that can be solved in polynomial time on a non-deterministic Turing machine



NP-complete Problems

- A question which is still not answered:

$$P \subset NP \text{ or } P \neq NP$$

- There is a strong belief that $P \neq NP$, due to the existence of NP-complete problems (NPC)
  - All NPC problems have the same degree of difficulty: if one of them could be solved in polynomial time, all of them would have a polynomial time solution.

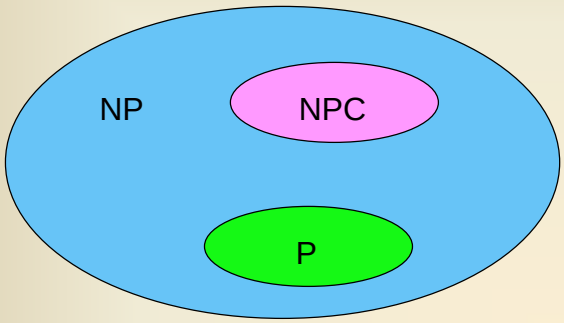
NP-complete Problems

- A problem is NP-complete if and only if
  - It is in NP
  - Some known NP-complete problem can be transformed to it in polynomial time
- Cook's theorem:
  - SATISFIABILITY is NP-complete

Reduction

- Idea: If we can solve problem A, and if problem B can be transformed into an instance of problem A, then we can solve problem B by reducing problem B to problem A and then solve the corresponding problem A.
- Example:
  - Problem A: Sorting
  - Problem B: Given n numbers, find the i-th largest numbers.
  - Polynomial-time Reducible

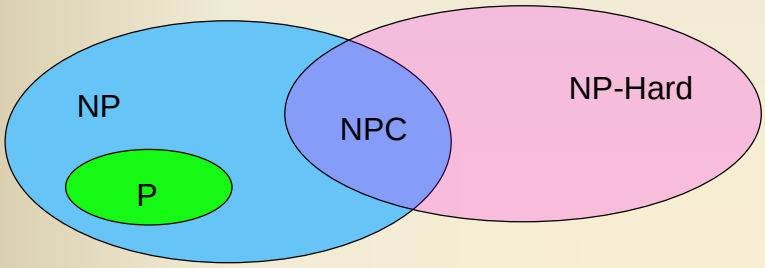
World of NP, Assuming P ≠ NP



NP-hard Problems

- Any decision problem (inside or outside of NP) to which we can transform an NP-complete problem to it in polynomial time will have a property that it cannot be solved in polynomial time, unless P = NP
- Such problems are called NP-hard
  - "as hard as the NP-complete problems"

NP-Hard, NP, and NPC



Practical Consequences

- Many problems in CAD for VLSI are NP-complete or NP-hard. Therefore:
  - Exact solutions to such problems can only be found when the problem size is small
  - One should otherwise be satisfied with sub-optimal solutions found by:
    - Approximation algorithms: they can guarantee a solution within e.g. 20% of the optimum
    - Heuristics: nothing can be said a priori about the quality of the solution (experience-based)

Example

- Tractable and intractable problems can be very similar:
  - the SHORTEST-PATH problem for undirected graphs is in
  - the LONGEST-PATH problem for undirected graphs is

Example

- Tractable and intractable problems can be very similar:
  - the SHORTEST-PATH problem for undirected graphs is in P
  - the LONGEST-PATH problem for undirected graphs is NP-complete

Examples of NP-complete Problems

- Clique:
  - Instance: graph G = (V,E), positive integer $K \leq |V|$
  - Question: does G contain a clique of size K or more?
- Minimum cover
  - Instance: collection C of subsets of a finite set S, positive integer $K \leq |C|$
  - Question: does G contain a cover for S of size K or less?

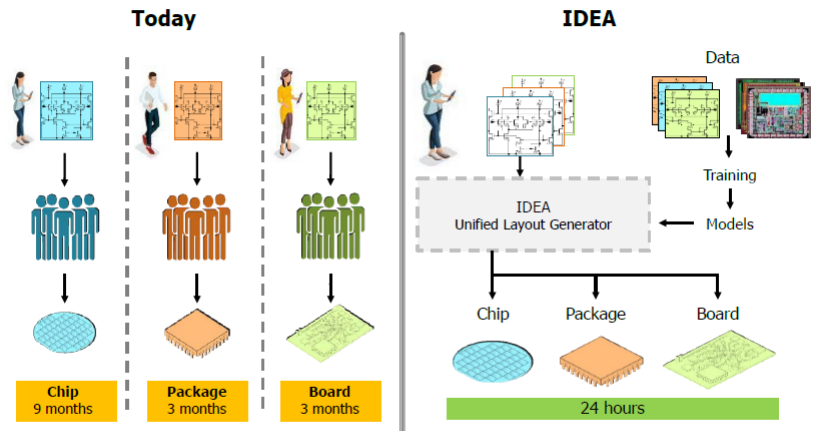
Brief Summary

- The class NP-complete is a set of problems which we believe there is no polynomial time algorithms
- Therefore, it is a class of hard problems
- NP-hard is another class of problems containing the class NP-complete
- If we know a problem is in NP-complete or NP-hard, there is nearly no hope to solve it efficiently

Intelligent Design of Electronic Assets (IDEA)

- Defense Advanced Research Projects Agency (DARPA)
- IDEA aims to create a “no human in the loop” 24 hour turnaround layout generator for System-On-Chips, System-In-Packages, and Printed Circuit Boards.

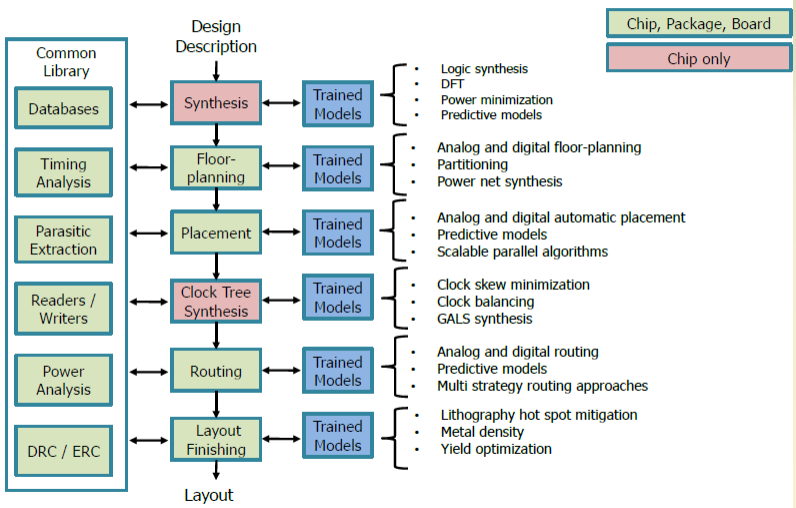
A unified electrical circuit layout generator

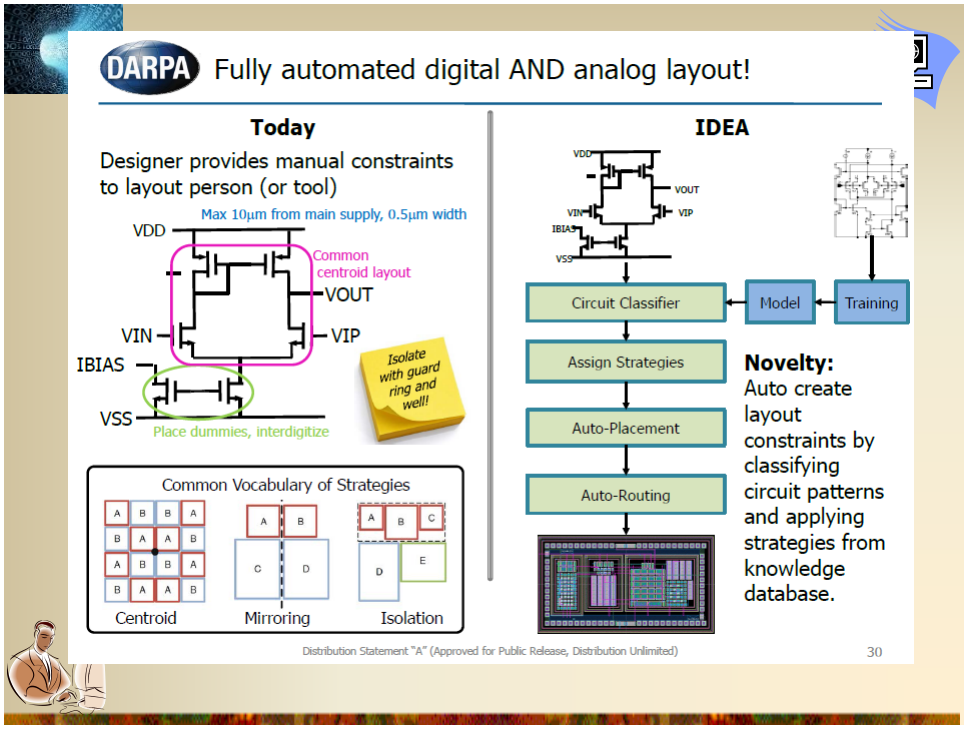


- Knowledge embedded in humans
- Limited knowledge reuse
- Reliance on scarce resources

- Knowledge embedded in software
- 100% automation
- 24 hour turnaround

Open EDA modularity




DARPA
Fully automated digital AND analog layout!
Today
Designer provides manual constraints to layout person (or tool)
Max 10μm from main supply, 0.5μm width
VDD
Common centroid layout
VOUT
VIN
VIP
IBIAS
VSS
Place dummies, interdigitize
Isolate with guard ring and well!
Common Vocabulary of Strategies
Centroid
Mirroring
Isolation
IDEA
Circuit Classifier
Model
Training
Assign Strategies
Auto-Placement
Auto-Routing
Novelty:
Auto create layout constraints by classifying circuit patterns and applying strategies from knowledge database.
Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)
30


Q&A
Lecture01
Slide 122