EE/CE 6301: Advanced Digital Logic

Bill Swartz

Dept. of EE
Univ. of Texas at Dallas

Session 02

Optimization / Overview of HDL-for-Synthesis

Credits

- *This presentation was adapted from work of Mehrdad Nourani of the University of Texas at Dallas.*

The Challenge of Optimization

Algorithm

- An algorithm defines a procedure for solving a computational problem
  - Examples:
    - Quick sort, bubble sort, insertion sort, heap sort
    - Dynamic programming method for the knapsack problem
- Definition of complexity
  - Run time on deterministic, sequential machines
  - Based on resources needed to implement the algorithm
    - Needs a cost model: memory, hardware/gates, communication bandwidth, etc.
    - Example: RAM model with single processor
      ➔ running time $\propto$ # operations

[©Gupta]

Runtime Complexity

- Runtime complexity: the time required by the algorithm to complete as a function of some natural measure of the problem size, allows comparing the scalability of various algorithms
- Complexity is represented in an asymptotic sense, with respect to the input size *n*, using big-Oh notation or O(…)
- Runtime $t(n)$ is order $f(n)$, written as $t(n) = O(f(n))$ when

  where *k* is a real number

- Example: $t(n) = 7n! + n^2 + 100$, then $t(n) = O(n!)$ because $n!$ is the fastest growing term as *n* approaches infinity.

$$\lim_{n\to\infty}\left|\frac{t(n)}{f(n)}\right| = k$$

Asymptotic Notions

- Idea:
  - A notion that ignores the "constants" and describes the "trend" of a function for large values of the input
- Definition
  - Big-Oh notation $f(n) = O(g(n))$
    if constants K and n_0 can be found such that:
    $\forall\ n \geq n_0,\ f(n) \leq K.\ g(n)$

    g is called an "upper bound" for f
    (f is "of order" g: f will not grow larger than g by more than a constant factor)

    Examples: $1/3\ n^2 = O(n^2)$ (also $O(n^3)$)
    $0.02\ n^2 + 127\ n + 1923 = O(n^2)$

Asymptotic Notions (cont.)

- Definition (cont.)
  - Big-Omega notation $f(n) = \Omega(g(n))$
    if constants K and n_0 can be found such that:
    $\forall\ n \geq n_0,\ f(n) \geq K.\ g(n)$

    g is called a “lower bound” for f
  - Big-Theta notation $f(n) = \Theta(g(n))$
    if g is both an upper and lower bound for f
    Describes the growth of a function more accurately than O or Ω
    Example:
    $n^3 + 4n \neq \Theta(n^2)$
    $4n^2 + 1024 = \Theta(n^2)$

[©Bazargan]

Asymptotic Notions (cont.)

- How to find the order of a function?
  - Not always easy, esp if you start from an algorithm
  - Focus on the “dominant” term
    - $4\ n^3 + 100\ n^2 + \log n \rightarrow O(n^3)$
    - $n + n \log(n) \rightarrow n \log(n)$
  - $n! = K^n > n^K > \log n > \log\log n > K$
    - $\Rightarrow n > \log n, \quad n \log n > n, \quad n! > n^{10}$.
- What do asymptotic notations mean in practice?
  - If algorithm A has “time complexity” $O(n^2)$ and algorithm B has time complexity $O(n \log n)$, then algorithm <u>B is better</u>
  - If problem P has a lower bound of $\Omega(n \log n)$, then there is NO WAY you can find an algorithm that solves the problem in $O(n)$ time.

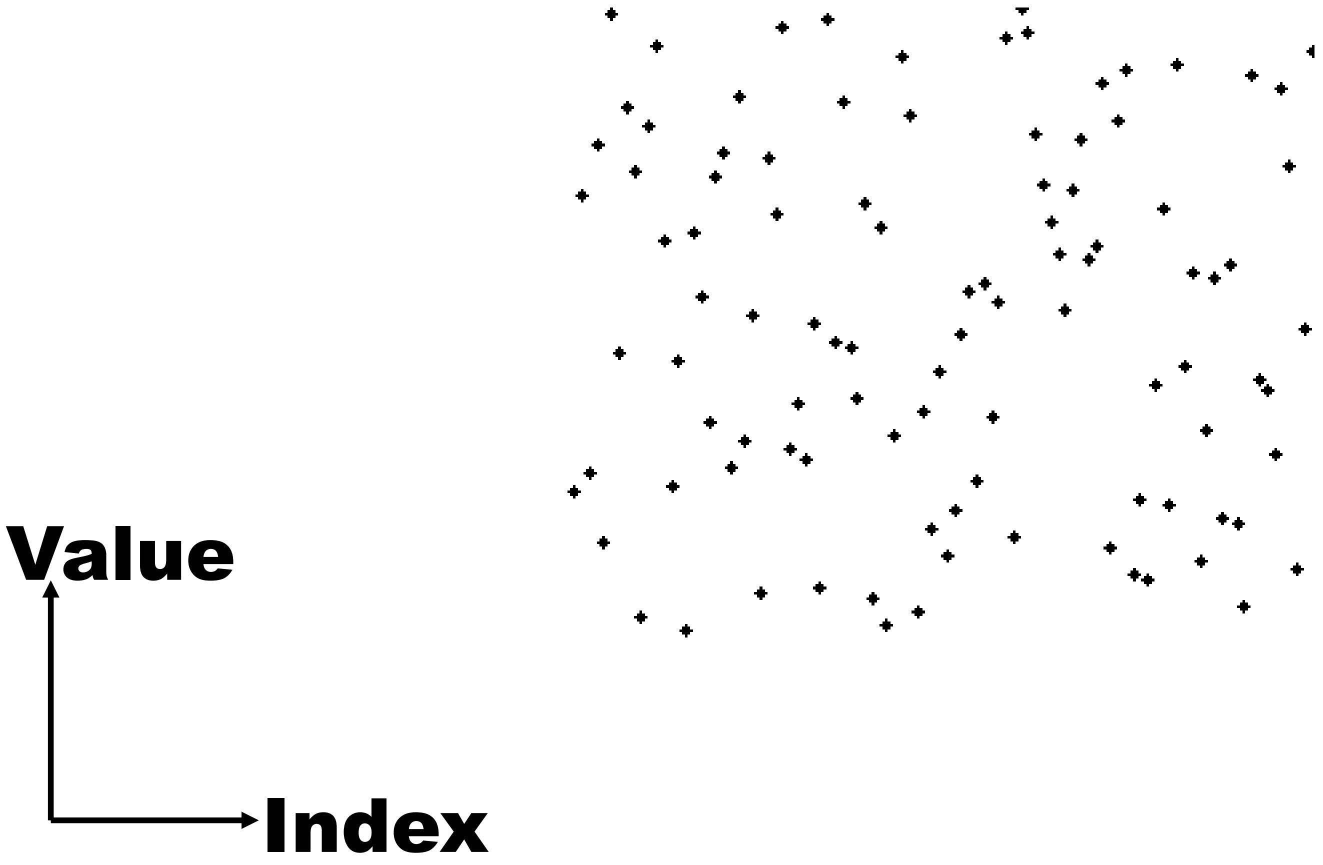
Algorithm (cont.)

- Definition of complexity (cont.)
  - Example: Bubble Sort
  - Scalability with respect to input size is important
    - How does the running time of an algorithm change when the input size doubles?
    - Function of input size (n).
      Examples: n^2+3n, 2^n, n log n, ...
    - Generally, large input sizes are of interest
      (n > 1,000 or even n > 1,000,000)
    - What if I use a better compiler?
      What if I run the algorithm on a machine that is 10x faster?

```
for (j=1 ; j< N; j++) {
   for (i=; i < N-j-1; i++) {
       if (a[i] > a[i+1]) {
         hold = a[i];
         a[i] = a[i+1];
         a[i+1] = hold;
        }
      }
   }
```

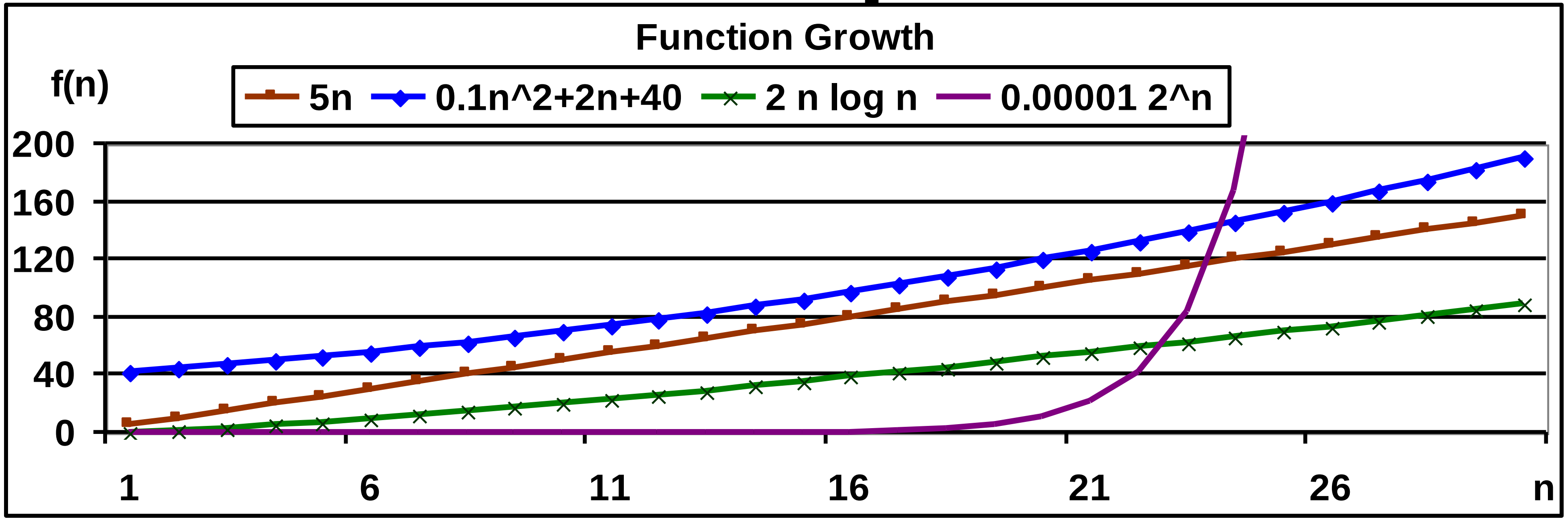
[©Gupta]

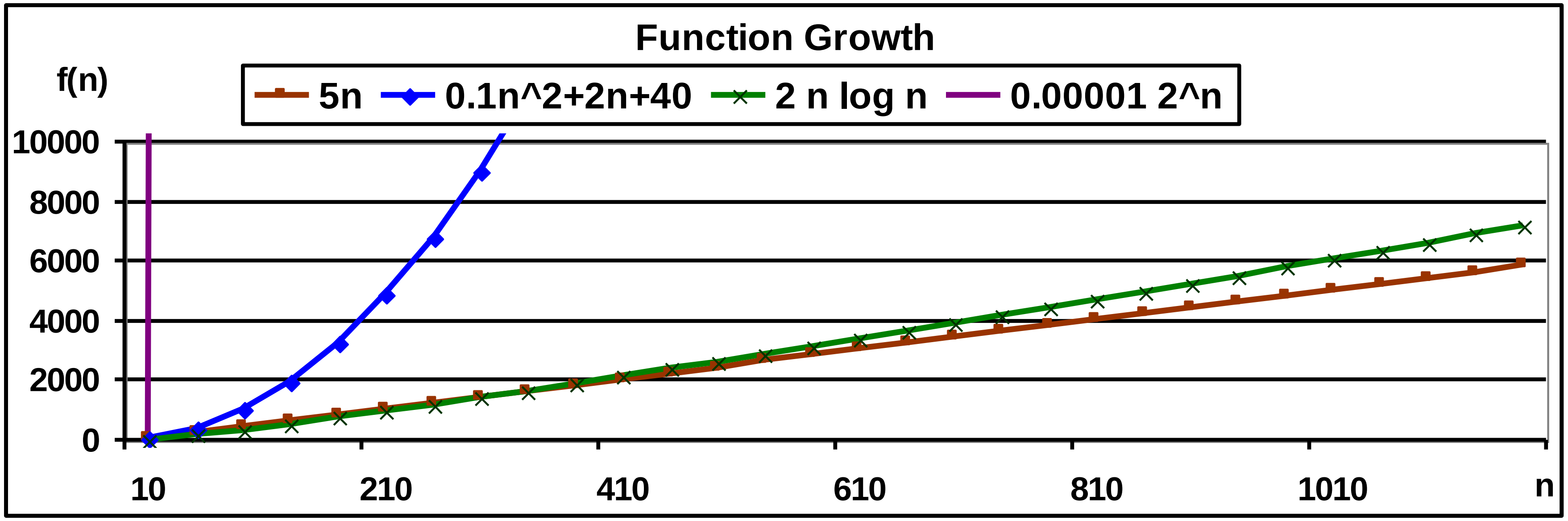
Bubble sort animation



[http://en.wikipedia.org/wiki/Image:Bubble_sort_animation.gif]

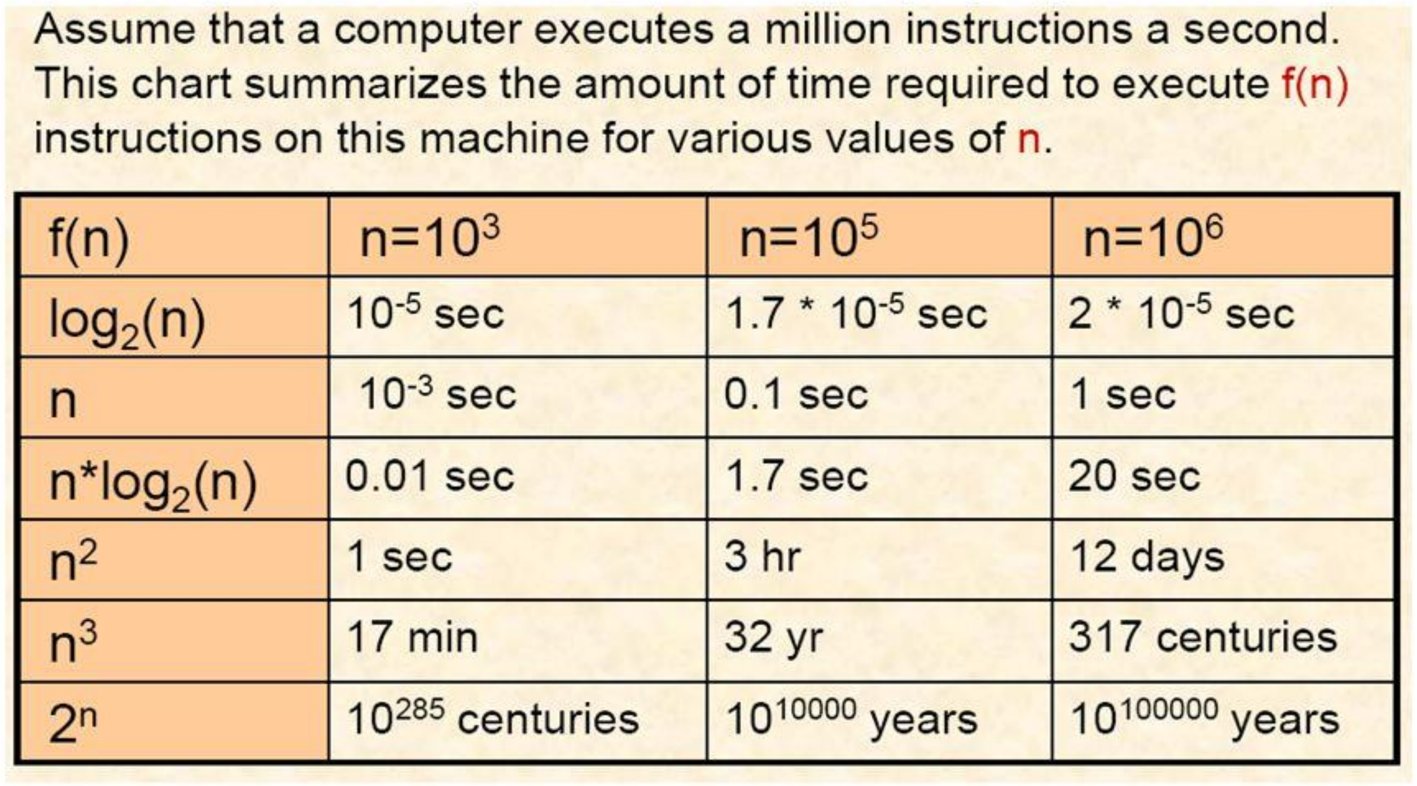
Function Growth Examples





[©Bazargan]

Importance of Asymptotic Analysis—Worst- & Average-Case



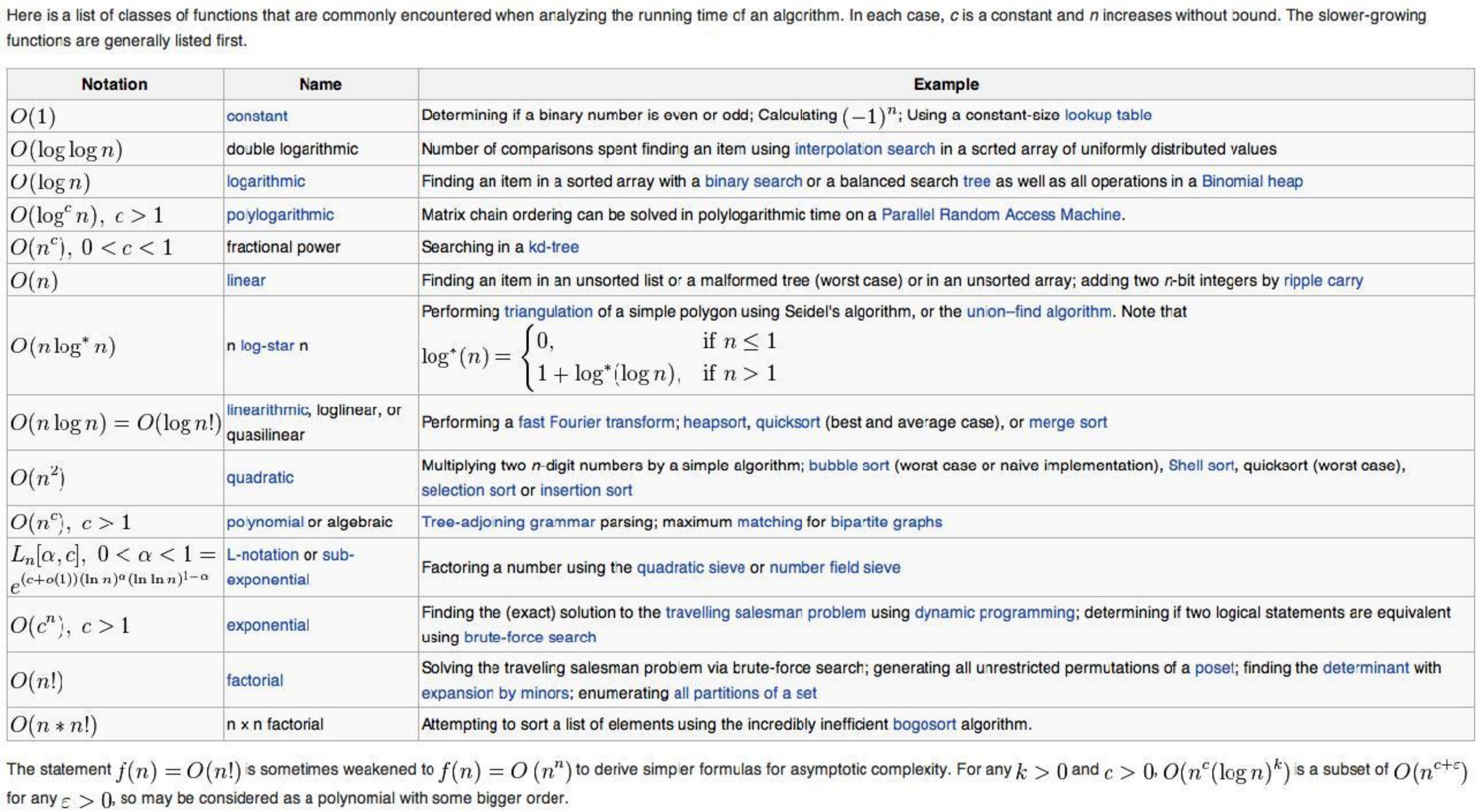
Assume that a computer executes a million instructions a second. This chart summarizes the amount of time required to execute f(n) instructions on this machine for various values of n.

f(n)	$n=10^3$	$n=10^5$	$n=10^6$
$\log_2(n)$	10^{-5} sec	$1.7 * 10^{-5}$ sec	$2 * 10^{-5}$ sec
n	10^{-3} sec	0.1 sec	1 sec
$n*\log_2(n)$	0.01 sec	1.7 sec	20 sec
n^2	1 sec	3 hr	12 days
n^3	17 min	32 yr	317 centuries
2^n	10^{285} centuries	10^{10000} years	10^{100000} years

- Asymptotic analysis tells us whether a technique/algorithm will be practical in all cases (worst-case analysis) or in the average-case (av.-case analysis) for problem sizes of interest

Ack: Table obtained from http://www.csd.uwo.ca/courses/CS1037a/notes/topic13_AnalysisOfAlgs.pdf

Asymptotic order of common functions



Here is a list of classes of functions that are commonly encountered when analyzing the running time of an algorithm. In each case, *c* is a constant and *n* increases without bound. The slower-growing functions are generally listed first.

Notation	Name	Example
$O(1)$	constant	Determining if a binary number is even or odd; Calculating $(-1)^n$; Using a constant-size lookup table
$O(\log\log n)$	double logarithmic	Number of comparisons spent finding an item using interpolation search in a sorted array of uniformly distributed values
$O(\log n)$	logarithmic	Finding an item in a sorted array with a binary search or a balanced search tree as well as all operations in a Binomial heap
$O(\log^c n),\ c>1$	polylogarithmic	Matrix chain ordering can be solved in polylogarithmic time on a Parallel Random Access Machine.
$O(n^c),\ 0<c<1$	fractional power	Searching in a kd-tree
$O(n)$	linear	Finding an item in an unsorted list or a malformed tree (worst case) or in an unsorted array; adding two *n*-bit integers by ripple carry
$O(n\log^* n)$	n log-star n	Performing triangulation of a simple polygon using Seidel's algorithm, or the union–find algorithm. Note that $\log^*(n) = \begin{cases} 0, & \text{if } n \le 1 \\ 1+\log^*(\log n), & \text{if } n > 1 \end{cases}$
$O(n\log n) = O(\log n!)$	linearithmic, loglinear, or quasilinear	Performing a fast Fourier transform; heapsort, quicksort (best and average case), or merge sort
$O(n^2)$	quadratic	Multiplying two *n*-digit numbers by a simple algorithm; bubble sort (worst case or naive implementation), Shell sort, quicksort (worst case), selection sort or insertion sort
$O(n^c),\ c>1$	polynomial or algebraic	Tree-adjoining grammar parsing; maximum matching for bipartite graphs
$L_n[\alpha,c],\ 0<\alpha<1 = e^{(c+o(1))(\ln n)^\alpha(\ln\ln n)^{1-\alpha}}$	L-notation or sub-exponential	Factoring a number using the quadratic sieve or number field sieve
$O(c^n),\ c>1$	exponential	Finding the (exact) solution to the travelling salesman problem using dynamic programming; determining if two logical statements are equivalent using brute-force search
$O(n!)$	factorial	Solving the traveling salesman problem via brute-force search; generating all unrestricted permutations of a poset; finding the determinant with expansion by minors; enumerating all partitions of a set
$O(n*n!)$	n x n factorial	Attempting to sort a list of elements using the incredibly inefficient bogosort algorithm.

The statement $f(n)=O(n!)$ is sometimes weakened to $f(n)=O\left(n^n\right)$ to derive simpler formulas for asymptotic complexity. For any $k>0$ and $c>0$, $O(n^c(\log n)^k)$ is a subset of $O(n^{c+\varepsilon})$ for any $\varepsilon>0$, so may be considered as a polynomial with some bigger order.

Algorithms and Complexity

- Example: Exhaustively Enumerating All Placement Possibilities
  - Given: n cells
  - Task: find a single-row placement of n cells with minimum total wirelength by using exhaustive enumeration.
  - Solution: The solution space consists of $n!$ placement options. If generating and evaluating the wirelength of each possible placement solution takes 1 µs and $n = 20$, the total time needed to find an optimal solution would be 77,147 years!
- A number of physical design problems have best-known algorithm complexities that grow exponentially with n, e.g., $O(n!)$, $O(n^n)$, and $O(2^n)$.
- Many of these problems are NP-hard (NP: non-deterministic polynomial time)
  - No known algorithms can ensure, in a time-efficient manner, globally optimal solution

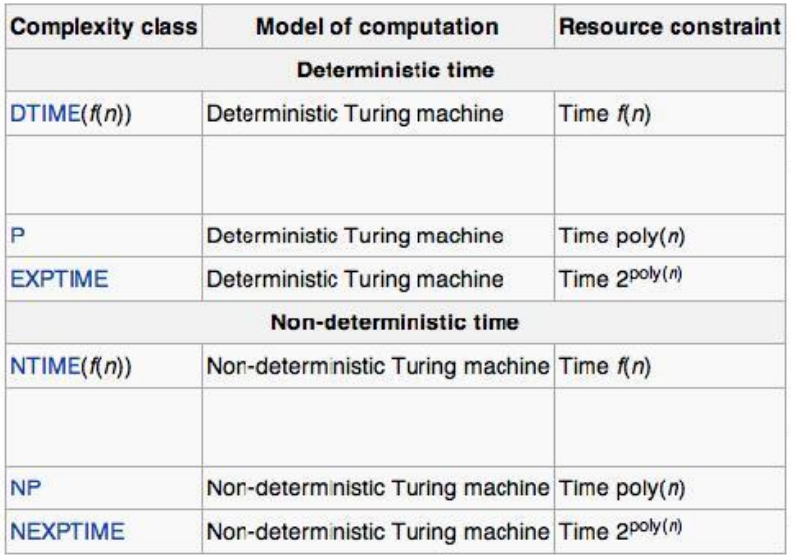
$\Rightarrow$ Heuristic algorithms are used to find near-optimal solutions

Problem Tractability

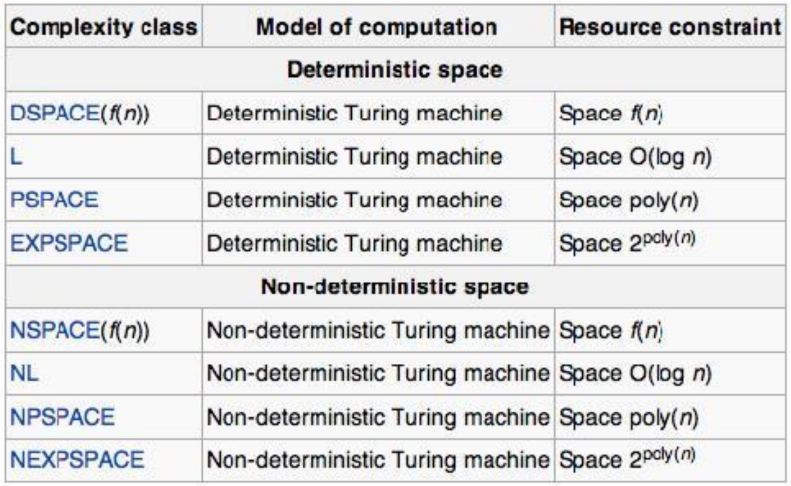
- Problems are classified into “easier” and “harder” categories
  - Class P: a polynomial time algorithm is known for the problem (hence, it is a tractable problem)
  - Class NP (non-deterministic polynomial time): a solution is verifiable in polynomial time
  - $P \subseteq NP$. Is $P = NP$? (Find out and become famous!)
  - Practically, for a problem in NP but not in P: polynomial solution not found yet (probably does not exist) ➔ exact (optimal) solution can be found using an algorithm with exponential time complexity
- NP-completeness, NP-hardness, etc.
  - Most CAD problems are NP-complete, NP-hard, or worse
  - Be happy with a “reasonably good” solution

Also in case anybody cares, it is incorrect to describe an optimization problem as NP-complete. Only decision problems with "Yes/No" (e.g. "does a solution exist of size K") answers can properly be termed NP-complete. Optimization problems (e.g. "find the best solution") are usually "NP-Hard". In polite company (and most journals) incorrect but well intentioned uses of "NP-complete" are accepted. -Craig Chase

Computational Complexity Classes

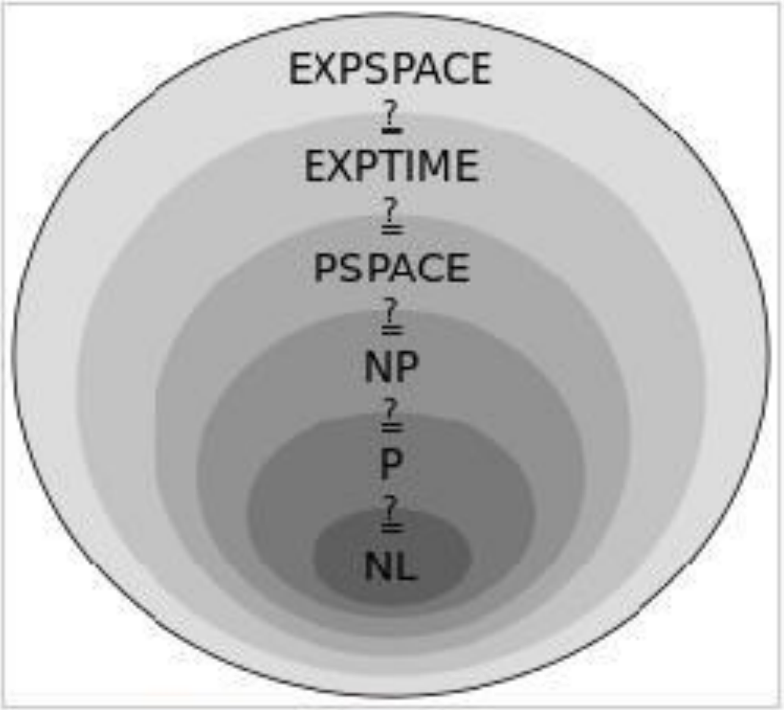


Complexity class	Model of computation	Resource constraint
Deterministic time		
DTIME($f(n)$)	Deterministic Turing machine	Time $f(n)$
P	Deterministic Turing machine	Time poly(n)
EXPTIME	Deterministic Turing machine	Time $2^{\mathrm{poly}(n)}$
Non-deterministic time		
NTIME($f(n)$)	Non-deterministic Turing machine	Time $f(n)$
NP	Non-deterministic Turing machine	Time poly(n)
NEXPTIME	Non-deterministic Turing machine	Time $2^{\mathrm{poly}(n)}$



Complexity class	Model of computation	Resource constraint
Deterministic space		
DSPACE($f(n)$)	Deterministic Turing machine	Space $f(n)$
L	Deterministic Turing machine	Space O(log n)
PSPACE	Deterministic Turing machine	Space poly(n)
EXPSPACE	Deterministic Turing machine	Space $2^{\mathrm{poly}(n)}$
Non-deterministic space		
NSPACE($f(n)$)	Non-deterministic Turing machine	Space $f(n)$
NL	Non-deterministic Turing machine	Space O(log n)
NPSPACE	Non-deterministic Turing machine	Space poly(n)
NEXPSPACE	Non-deterministic Turing machine	Space $2^{\mathrm{poly}(n)}$

Computational Complexity Classes



A representation of the relation among complexity classes

Examples of NP-complete problems

- Does a graph have a Hamiltonian cycle?
  - Hamiltonian cycle = simple cycle (no repeated vertices) that contains all nodes
  - Related – Traveling salesman problem (mincost Hamiltonian cycle)
- 3SAT: Given a Boolean expression expressed as POS with 3 literals in each sum, is it satisfiable?
  - 2SAT can be solved in polynomial time!
- Find a maximal clique in a graph
  - Clique = set of vertices so that every pair of vertices in the set is connected by an edge (complete subgraph)
- Find a maximal independent set in a graph
  - A set of vertices of the largest cardinality, so that no pair of vertices is connected by an edge
- "Bible" of NP-completeness:
  - M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Company, New York, NY, 1979.

Deterministic Algorithm Types

- Algorithms usually used for P problems
  - Exhaustive search! (aka exponential)
  - Dynamic programming
  - Divide & Conquer (aka hierarchical)
  - Greedy
  - Mathematical programming
  - Branch and bound
- Algorithms usually used for NP problems
  (not seeking "optimal solution", but a "good" one)
  - Greedy (aka heuristic)
  - Genetic algorithms
  - Simulated annealing
  - Restrict the problem to a special case that is in P

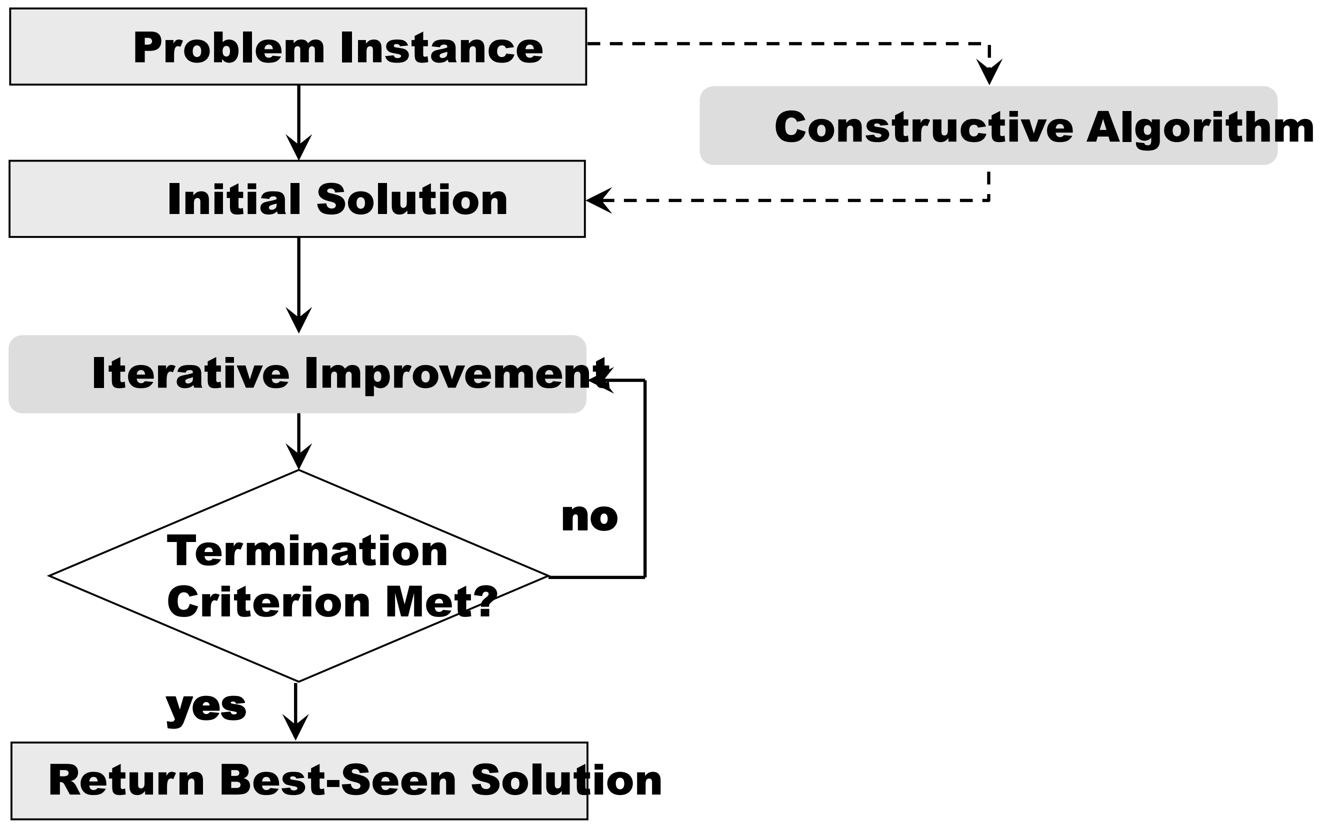
Heuristic algorithms

- Deterministic: All decisions made by the algorithm are repeatable, i.e., not random. One example of a deterministic heuristic is Dijkstra's shortest path algorithm.

- Stochastic: Some decisions made by the algorithm are made randomly, e.g., using a pseudo-random number generator. Thus, two independent runs of the algorithm will produce two different solutions with high probability. One example of a stochastic algorithm is simulated annealing.

- In terms of structure, a heuristic algorithm can be
  - Constructive: The heuristic starts with an initial, incomplete (partial) solution and adds components until a complete solution is obtained.
  - Iterative: The heuristic starts with a complete solution and repeatedly improves the current solution until a preset termination criterion is reached.

Flowchart of heuristic algorithms



Coping with NP-hard Problems

- In system level design we confront many NP-hard optimization problems.
- Simpler sub-problem based on dominate cost or special problem structure
- problems exhibit structure
  - optimal solutions found in reasonable time in practice
- approximation algorithms
- heuristic solutions
- high density of good/reasonable solutions?

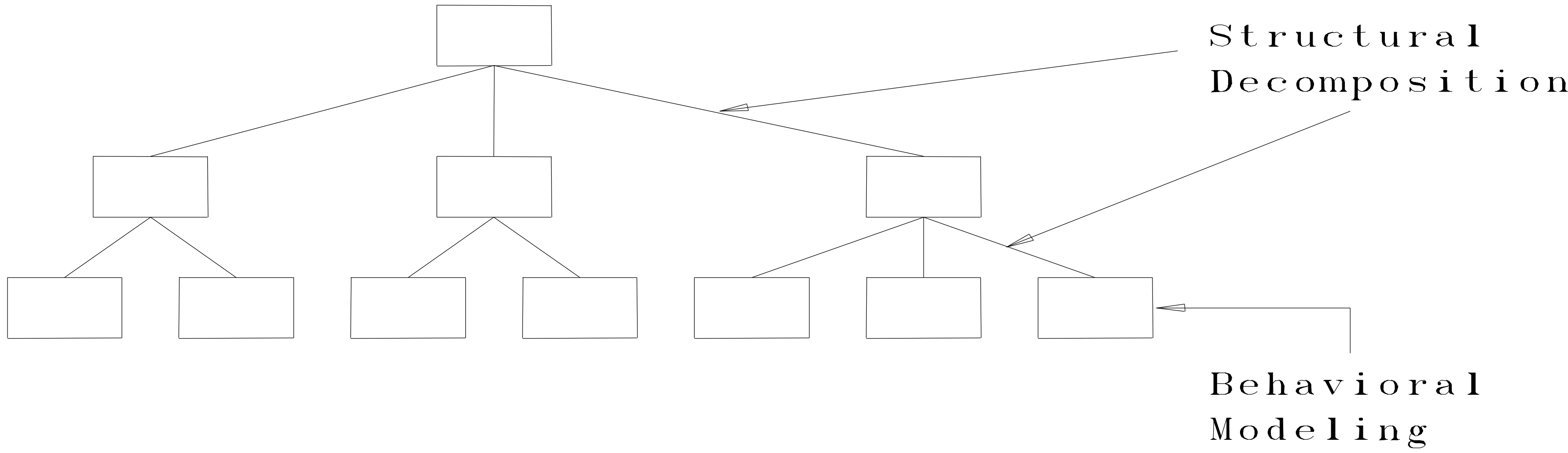
Not a Solved Problem

- NP-hard problems
  - —almost always solved in suboptimal manner
  - —or for particular special cases
- decomposed in suboptimal ways
- quality of solution changes as dominant costs change (relative costs are changing!)
- new effects and mapping problems crop up with new architectures, substrates

Decomposition

- Easier to solve
  - only worry about one problem at a time
- Less computational work
  - smaller problem size
- Abstraction hides important objectives
  - solving 2 problems optimally in sequence often not give optimal result of simultaneous solution
  - Question: Like what?

Decomposition to a Tree Hierarchy



Top-Down Design

- Begin at the top.
- Partition according to some objective criterion.
- No "priori" knowledge of available lower level components.

- Advantage: optimized partition.
- Disadvantage: unique level components.

Bottom-Up Design

- Begin at the bottom.
- Cluster components to take advantage of available lower level components.
- Lower level components were designed first.

- Advantage: use available components.
- Disadvantage: clustering is often non-optimal. Why?

Partitioning

- Definition: Given a set of objects $O=\{o_1,\ldots,o_n\}$ determine a partition $P=\{p_1,\ldots p_m\}$ such that $p_1 U \ldots U p_m = O$, $p_i \cdot p_j = 0$ for all *i,j*, *i* != *j* and the cost determined by an objective function f(P) is minimal.
- NP-complete for general graphs/problems
- Many heuristics / approaches
- System designer must do two things:
  1. Selecting a set of system components (allocation)
  2. Partitioning the system's functionality among those components (partitioning).
- Partitioning Issues:
  - Abstraction level
  - Granularity
  - Estimation

Partitioning Heuristic

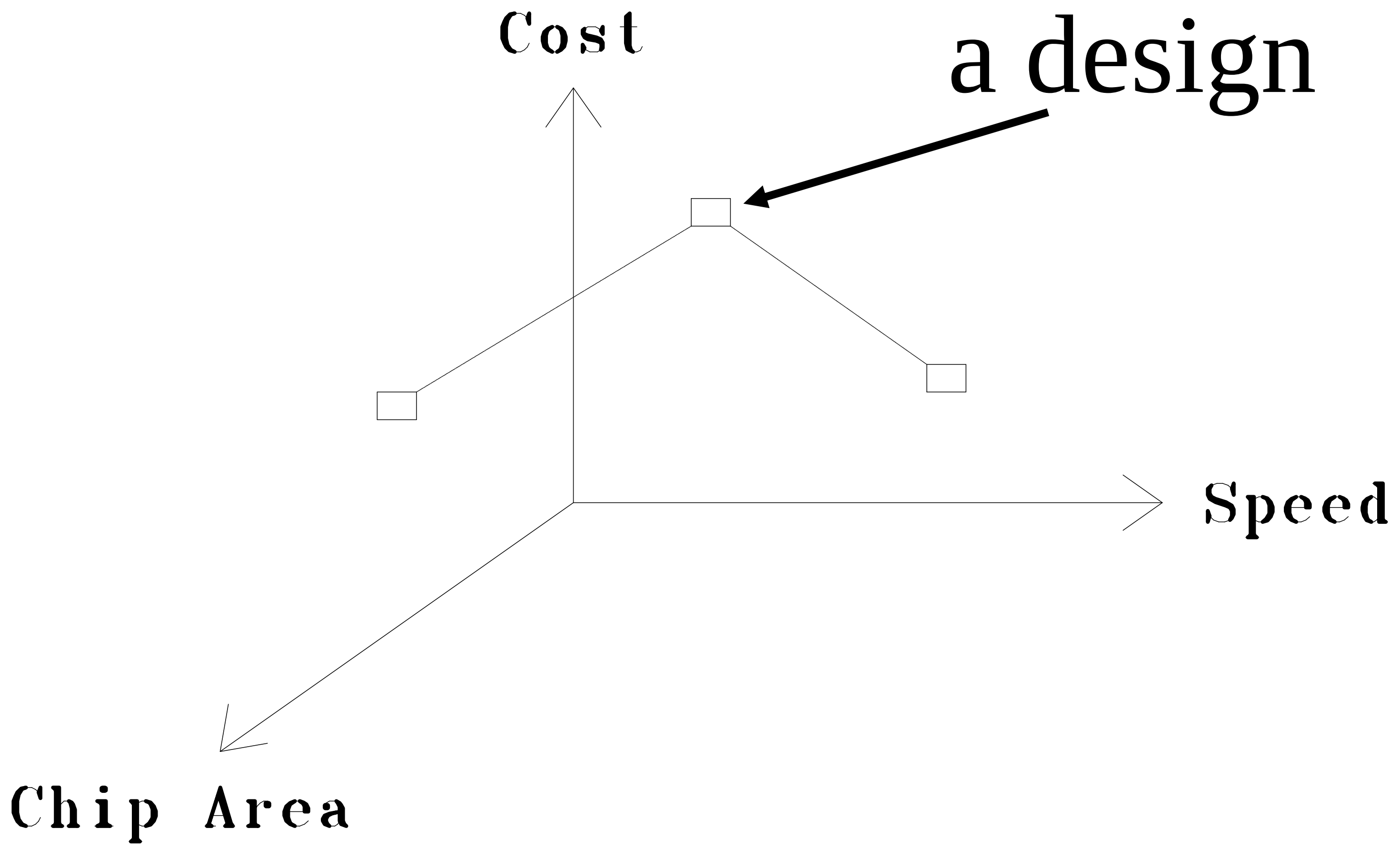
- Greedy, iterative
  - —pick one partition that decreases cost (i.e. a user defined metric) and move it
  - —repeat
- Small amount of:
  - —look past moves that make locally worse
  - —randomization
- Estimation Metrics:
  - —Fast (usually analytical) estimate of area,time,power,etc.
  - —Fidelity of estimation
- Quality Metrics:
  - —Hardware/software cost, performance, benchmarking

Design Space

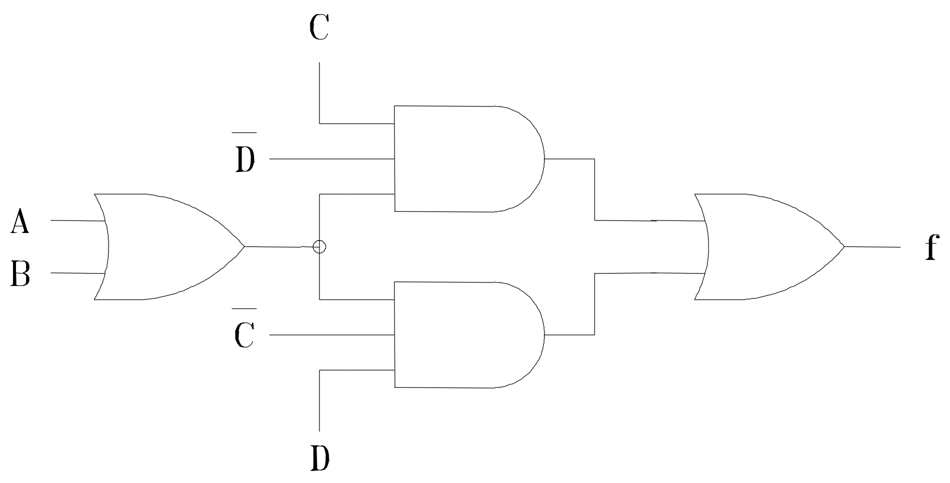
Concept of Design Space

- There exists no perfect/optimal algorithm for the design of complicated systems
- The designer moves around in a space
- The coordinates of the space are optimization criterion: speed, chip area, cost, power, pins, etc.
- Motion in the space involves tradeoffs

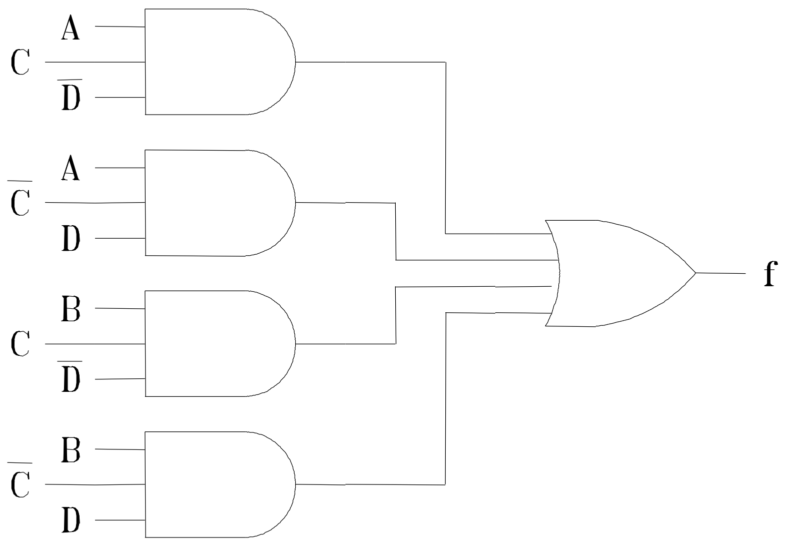
A 3-Dimensional Design Space



Example: Speed-Area Tradeoff

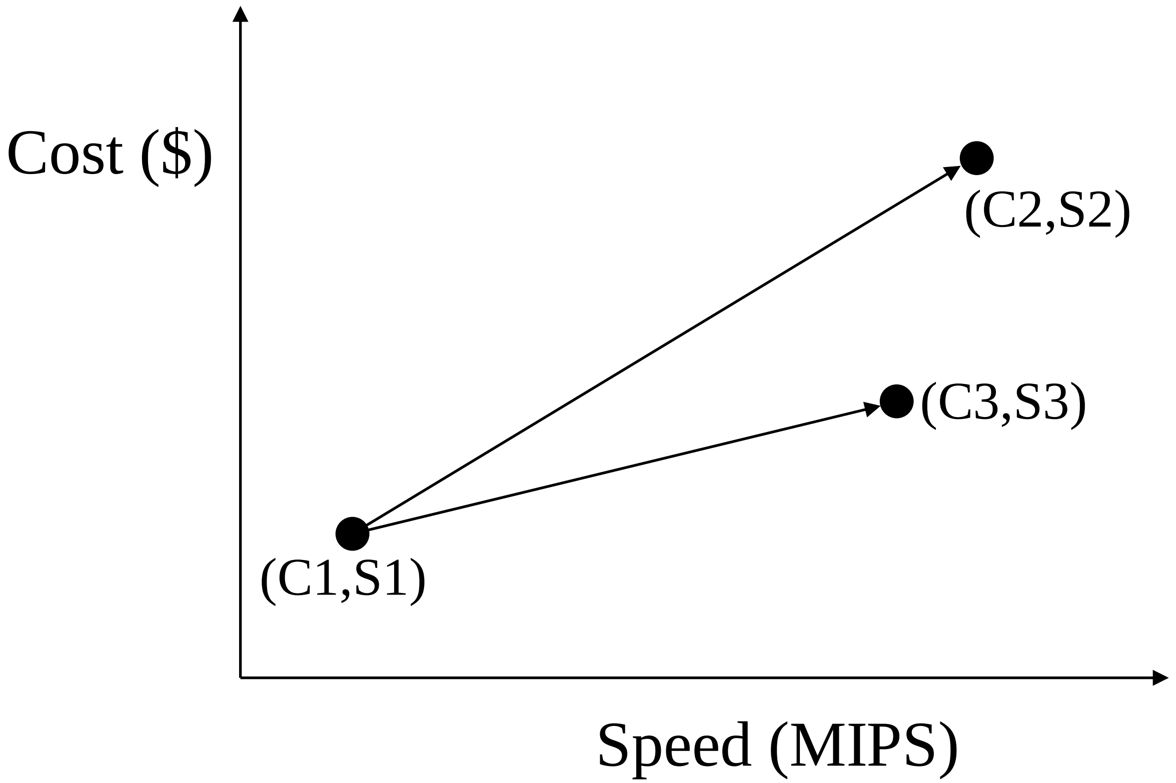


Circuit A



Circuit B

Example: Workstation Cost/Speed Tradeoff



C1	$ 5K
S1	50 MIPS
C2	$ 30K
S2	500 MIPS
C3	$ 10K
S3	280 MIPS

Lecture 9: Multi-Objective Optimization

Suggested reading: K. Deb, *Multi-Objective Optimization using Evolutionary Algorithms*, John Wiley & Sons, Inc., 2001

Multi-Objective Optimization Problems (MOOP)

- Involve more than one objective function that are to be minimized or maximized
- Answer is set of solutions that define the best tradeoff between competing objectives

General Form of MOOP

- Mathematically

$$
\begin{aligned}
&\text{min/max} \quad f_m(\boldsymbol{x}), && m = 1, 2, \ldots, M \\
&\text{subject to } g_j(\boldsymbol{x}) \geq 0, && j = 1, 2, \ldots, J \\
&\qquad h_k(\boldsymbol{x}) = 0, && k = 1, 2, \ldots, K \\
&\qquad \underbrace{x_i^{(L)}}_{\text{lower bound}} \leq x_i \leq \underbrace{x_i^{(U)}}_{\text{upper bound}}, && i = 1, 2, \ldots, n
\end{aligned}
$$

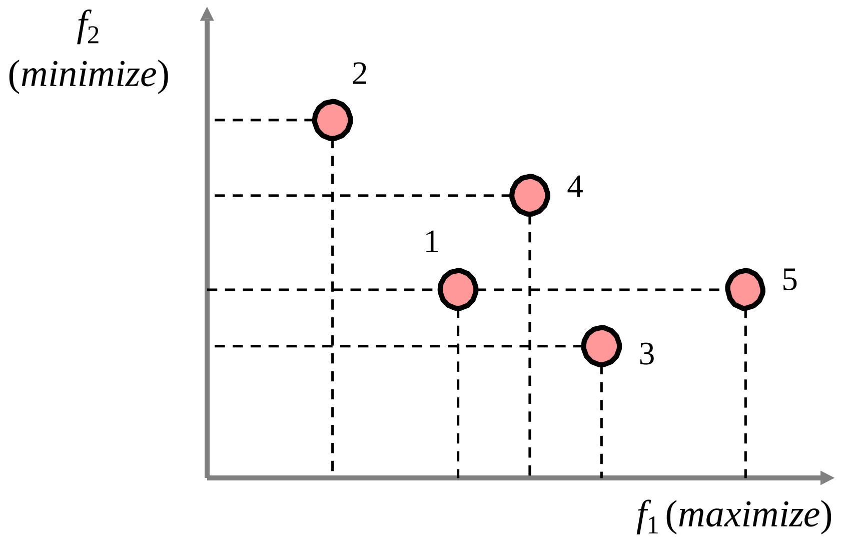
Dominance

- In the single-objective optimization problem, the superiority of a solution over other solutions is easily determined by comparing their objective function values
- In multi-objective optimization problem, the goodness of a solution is determined by the **dominance**

Definition of Dominance

- **Dominance Test**
  - $\boldsymbol{x}_1$ dominates $\boldsymbol{x}_2$, if
    - Solution $\boldsymbol{x}_1$ is no worse than $\boldsymbol{x}_2$ in all objectives
    - Solution $\boldsymbol{x}_1$ is strictly better than $\boldsymbol{x}_2$ in at least one objective
  - $\boldsymbol{x}_1$ dominates $\boldsymbol{x}_2$ $\Longleftrightarrow$ $\boldsymbol{x}_2$ is dominated by $\boldsymbol{x}_1$

Example Dominance Test

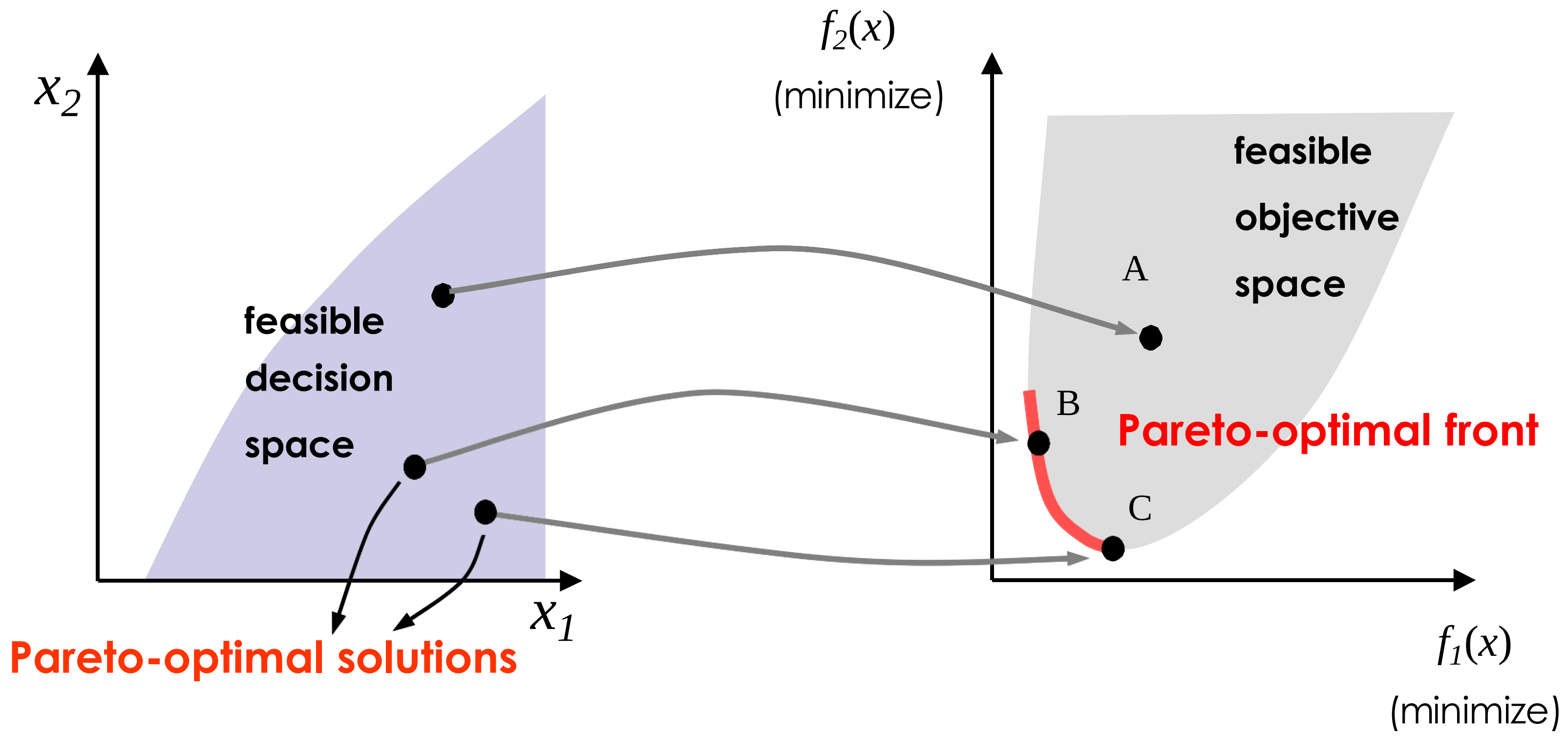


- 1 Vs 2: 1 dominates 2
- 1 Vs 5: 5 dominates 1
- 1 Vs 4: Neither solution dominates

Pareto Optimal Solution

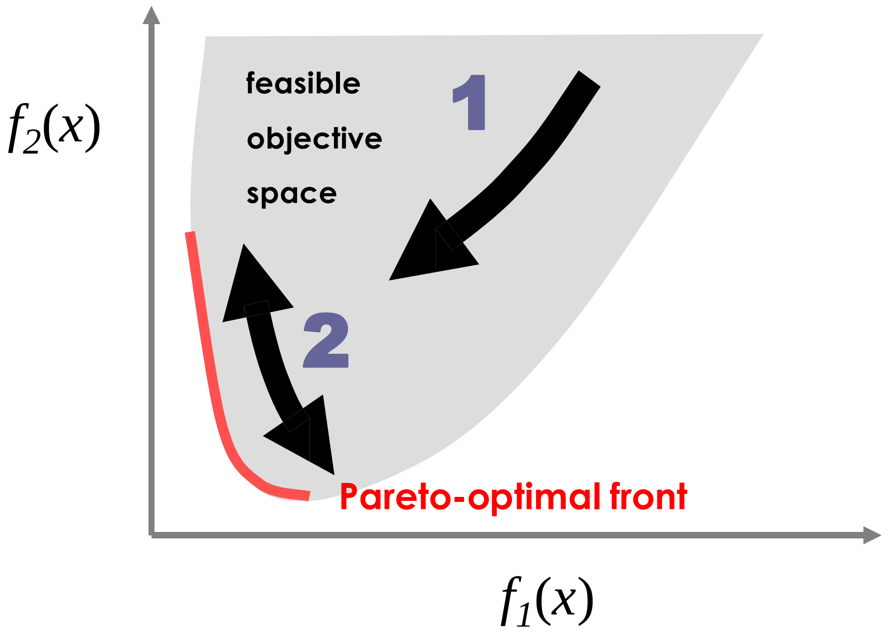
- **Non-dominated solution set**
  - Given a set of solutions, the non-dominated solution set is a set of all the solutions that are not dominated by any member of the solution set
- The non-dominated set of the entire feasible decision space is called the **Pareto-optimal set**
- The boundary defined by the set of all point mapped from the Pareto optimal set is called the **Pareto-optimal front**

Graphical Depiction of Pareto Optimal Solution



Goals in MOO

- Find set of solutions as close as possible to Pareto-optimal front
- To find a set of solutions as diverse as possible



Classic MultiObjectiveOptimization Methods

Weighted Sum Method

- Scalarize a set of objectives into a single objective by adding each objective pre-multiplied by a user-supplied weight

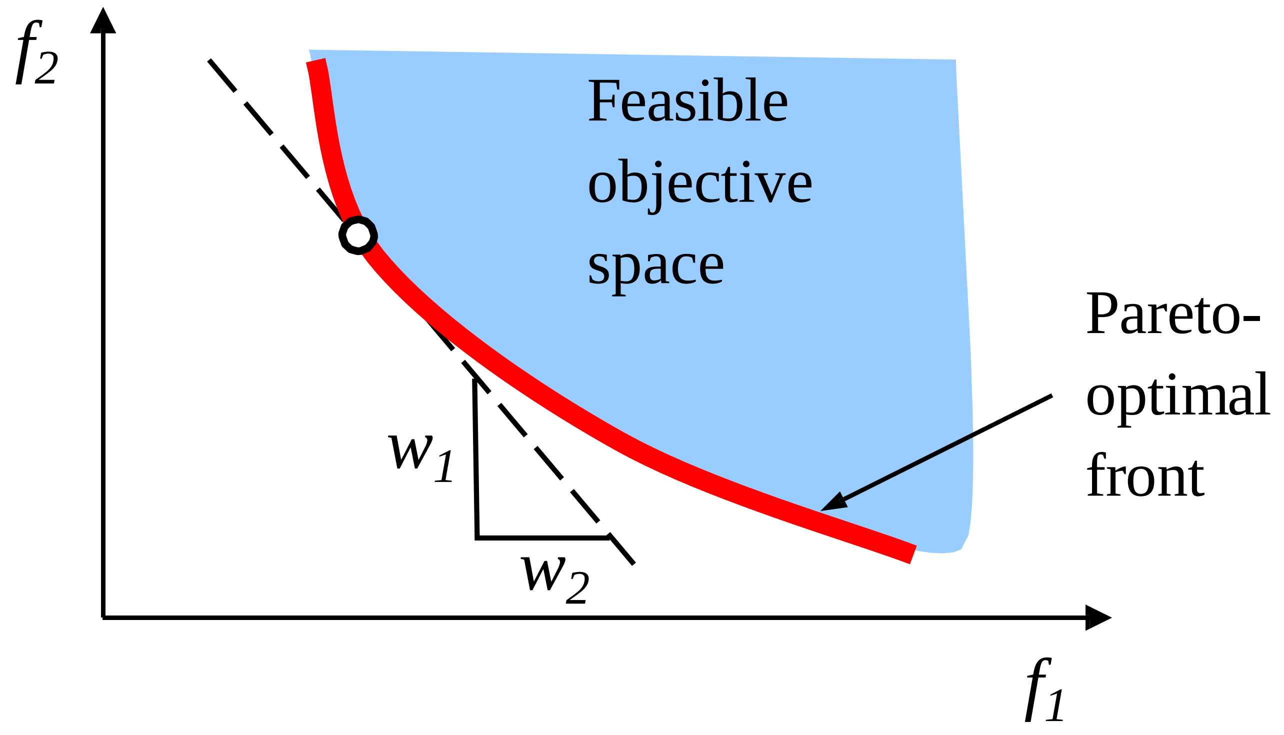
$$\begin{array}{lll} \textbf{minimize} & F(\boldsymbol{x}) = \sum_{m=1}^{M} w_m f_m(\boldsymbol{x}), & \\ \textbf{subject to} & g_j(\boldsymbol{x}) \geq 0, & j = 1, 2, \ldots, J \\ & h_k(\boldsymbol{x}) = 0, & k = 1, 2, \ldots, K \\ & x_i^{(L)} \leq x_i \leq x_i^{(U)}, & i = 1, 2, \ldots, n \end{array}$$

- Weight of an objective is chosen in proportion to the relative importance of the objective

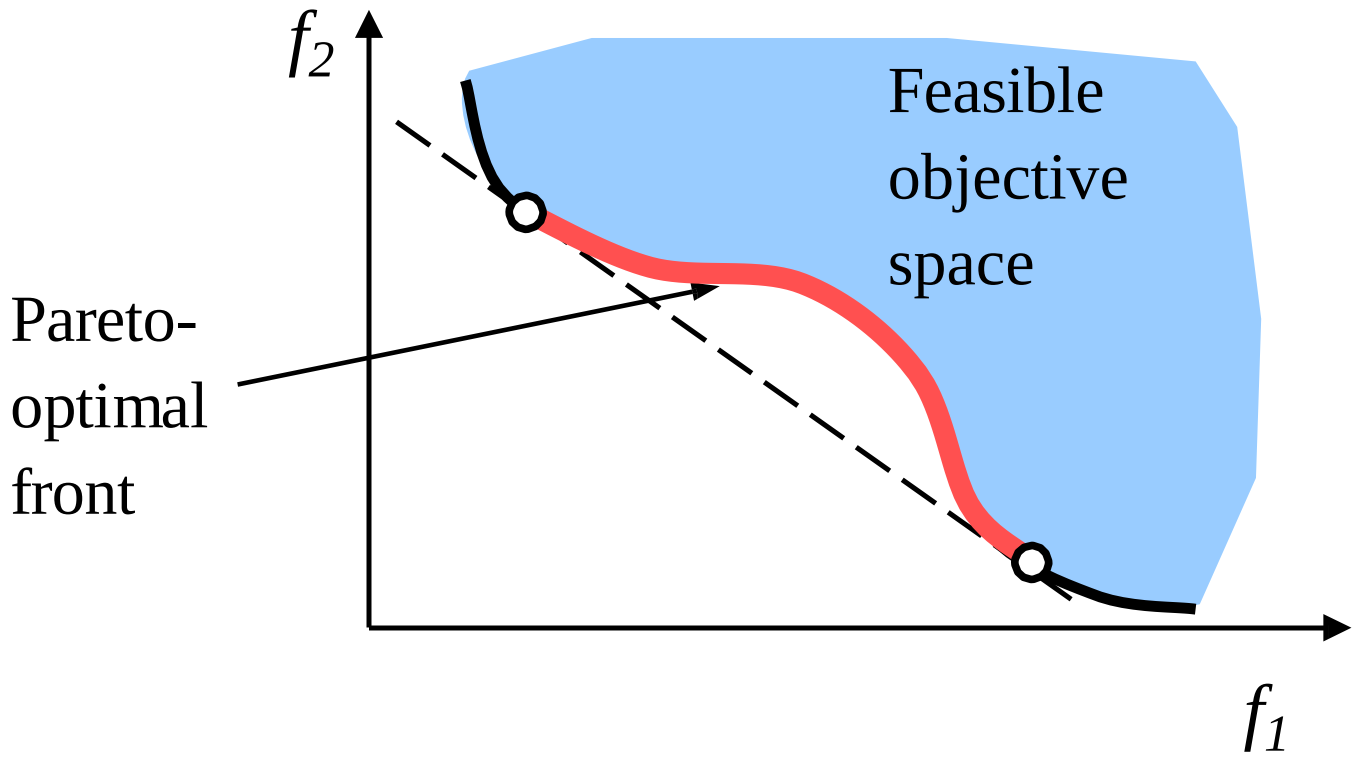
Weighted Sum Method

- Advantage
  - Simple
- Disadvantage
  - It is difficult to set the weight vectors to obtain a Pareto-optimal solution in a desired region in the objective space
  - It cannot find certain Pareto-optimal solutions in the case of a nonconvex objective space

Weighted Sum Method (Convex Case)



Weighted Sum Method (Non-Convex Case)



ε-Constraint Method

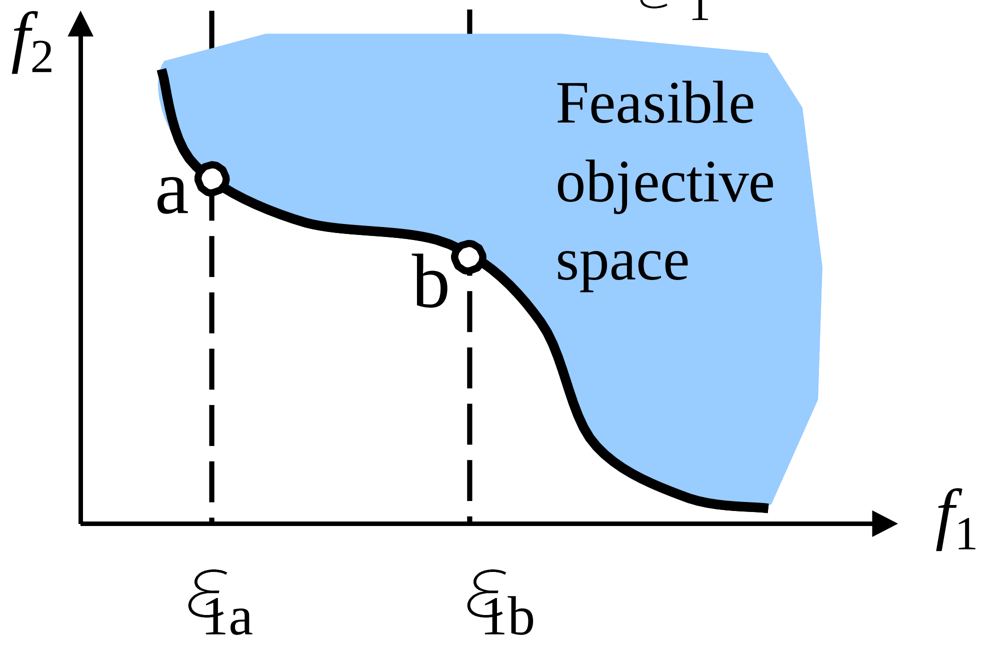
- Haimes et. al. 1971
- Keep just one of the objective and restricting the rest of the objectives within user-specific values

$$\begin{array}{lll}
\textbf{minimize} & f_\mu(\boldsymbol{x}), & \\
\textbf{subject to} & f_m(\boldsymbol{x}) \le \varepsilon_m, & m = 1, 2, \ldots, M \text{ and } m \ne \mu \\
& g_j(\boldsymbol{x}) \ge 0, & j = 1, 2, \ldots, J \\
& h_k(\boldsymbol{x}) = 0, & k = 1, 2, \ldots, K \\
& x_i^{(L)} \le x_i \le x_i^{(U)}, & i = 1, 2, \ldots, n
\end{array}$$

ε-Constraint Method

Keep f_2 as an objective **Minimize** $f_2(\boldsymbol{x})$

Treat f_1 as a constraint $f_1(\boldsymbol{x}) \leq \varepsilon_1$



ε-Constraint Method

- Advantage
  - Applicable to either convex or non-convex problems

- Disadvantage
  - The ε vector has to be chosen carefully so that it is within the minimum or maximum values of the individual objective function

Weighted Metric Method

- Combine multiple objectives using the weighted distance metric of any solution from the ideal solution z^*

minimize $$l_{\mathbf{p}}(\boldsymbol{x}) = \left(\sum_{m=1}^{M} w_m \left| f_m(\boldsymbol{x}) - z_m^* \right|^p \right)^{1/p},$$

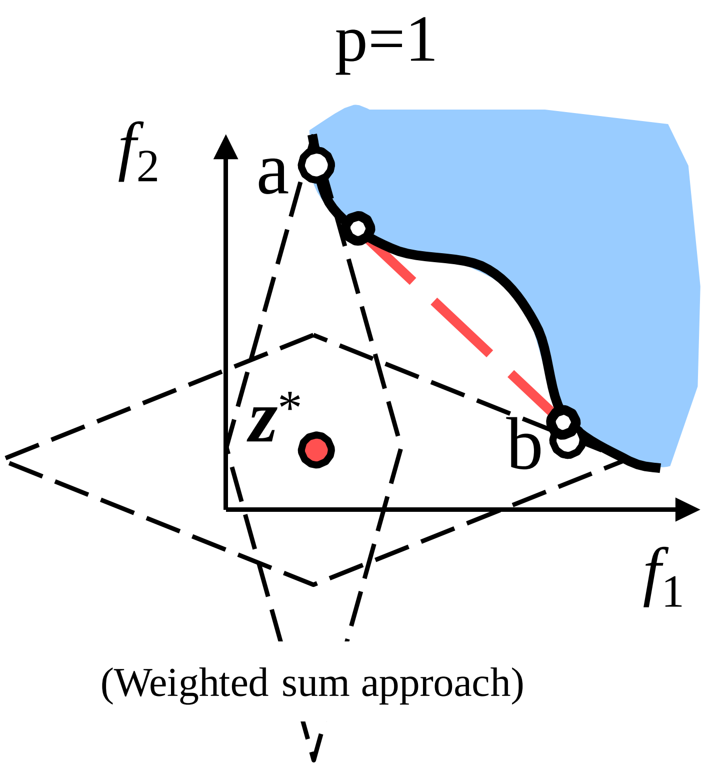
subject to

$$g_j(\boldsymbol{x}) \geq 0, \qquad j = 1, 2, \ldots, J$$

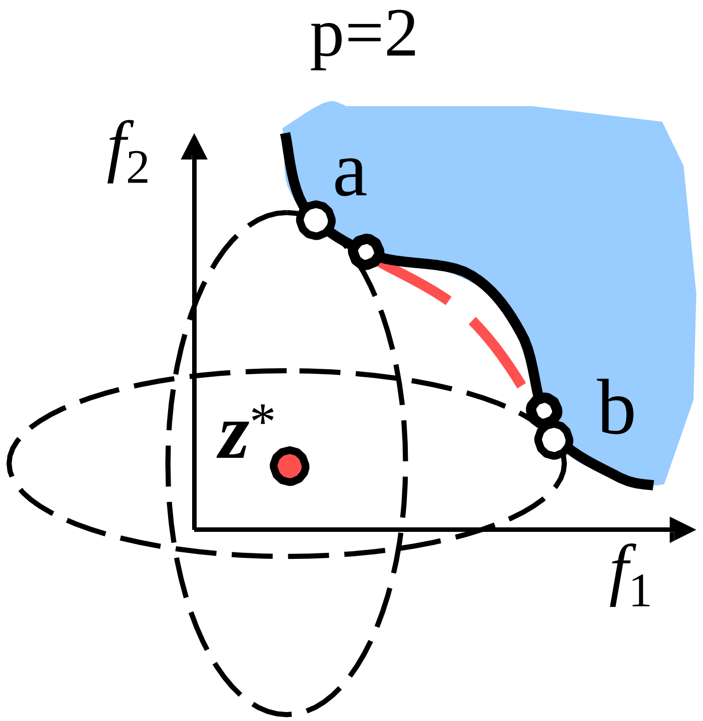
$$h_k(\boldsymbol{x}) = 0, \qquad k = 1, 2, \ldots, K$$

$$x_i^{(L)} \leq x_i \leq x_i^{(U)}, \qquad i = 1, 2, \ldots, n$$

Weighted Metric Method

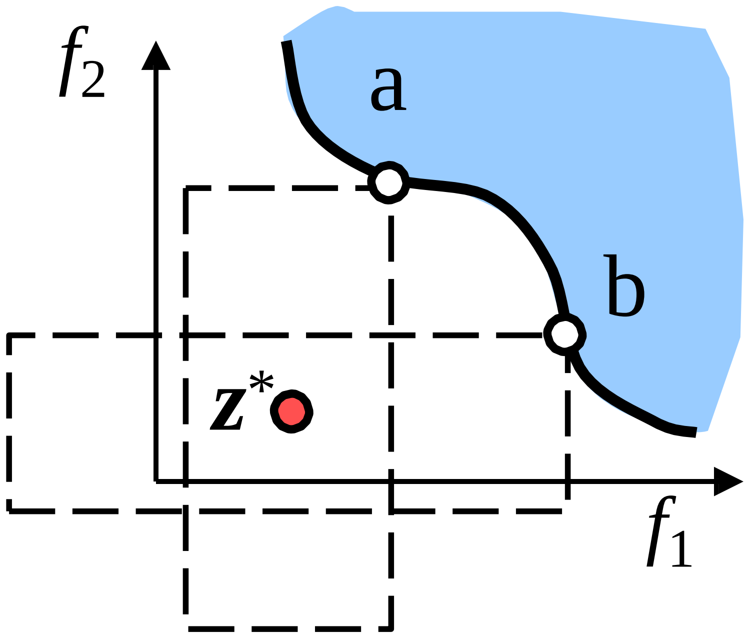


(Weighted sum approach)



Weighted Metric Method

$p=\infty$



(Weighted Tchebycheff problem)

Weighted Metric Method

- Advantage
  - Weighted Tchebycheff metric guarantees finding all Pareto-optimal solution with ideal solution z^*
- Disadvantage
  - Requires knowledge of minimum and maximum objective values
  - Requires z^* which can be found by independently optimizing each objective functions
  - For small p, not all Pareto-optimal solutions are obtained
  - As p increases, the problem becomes non-differentiable

Overview of HDL-for-Synthesis

Fundamental Concepts

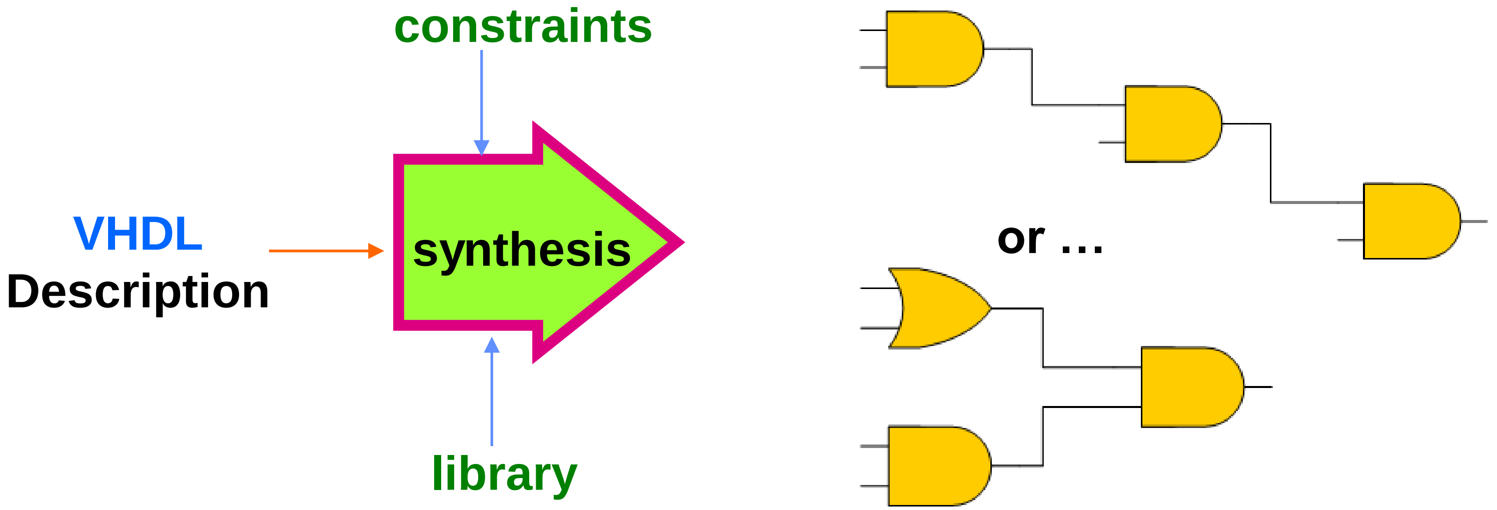
Hardware Modeling Using HDL

- **HDL: Hardware Description Language -** A high level programming language used to model hardware.

- Hardware Description Languages
  - have special hardware related constructs.
  - can be used to build models for **simulation**, **synthesis** and **test**
  - have been extended to the system design level
  - **VHDL: V**HSIC **H**ardware **D**escription **L**anguage
    - VHSIC – Very High Speed Integrated Circuit Program
    - Mostly used in academia
  - **Verilog** HDL
    - Mostly used in commercial electronics industry

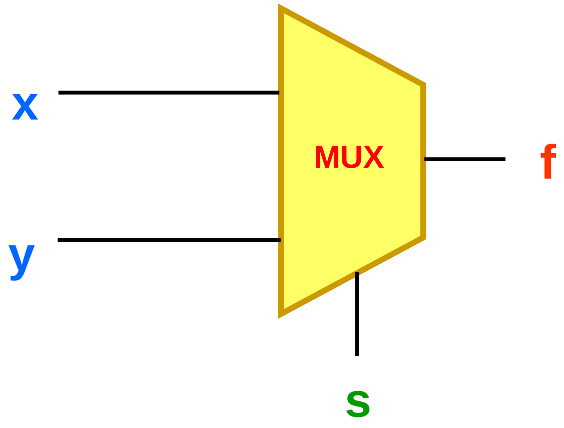
Concept of Synthesis

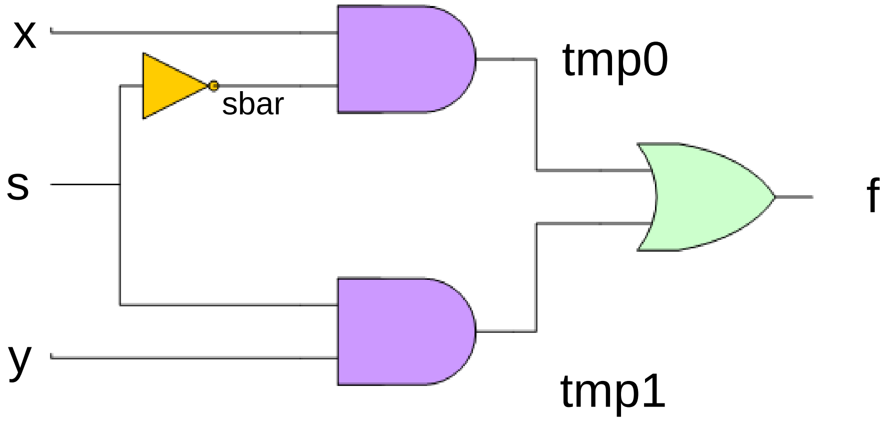
- Logic synthesis
  - A program that **"designs" logic from abstract descriptions** of the logic
    - takes constraints (e.g. size, speed)
    - uses a library (e.g. 3-input gates)
  - The aim of synthesis is to produce hardware which will do what the concurrent statements specify.
    - This includes processes as well as other concurrent statements.
- How?
  - You write an "abstract" HDL description of the logic
  - The synthesis tool provides alternative implementations



Goal

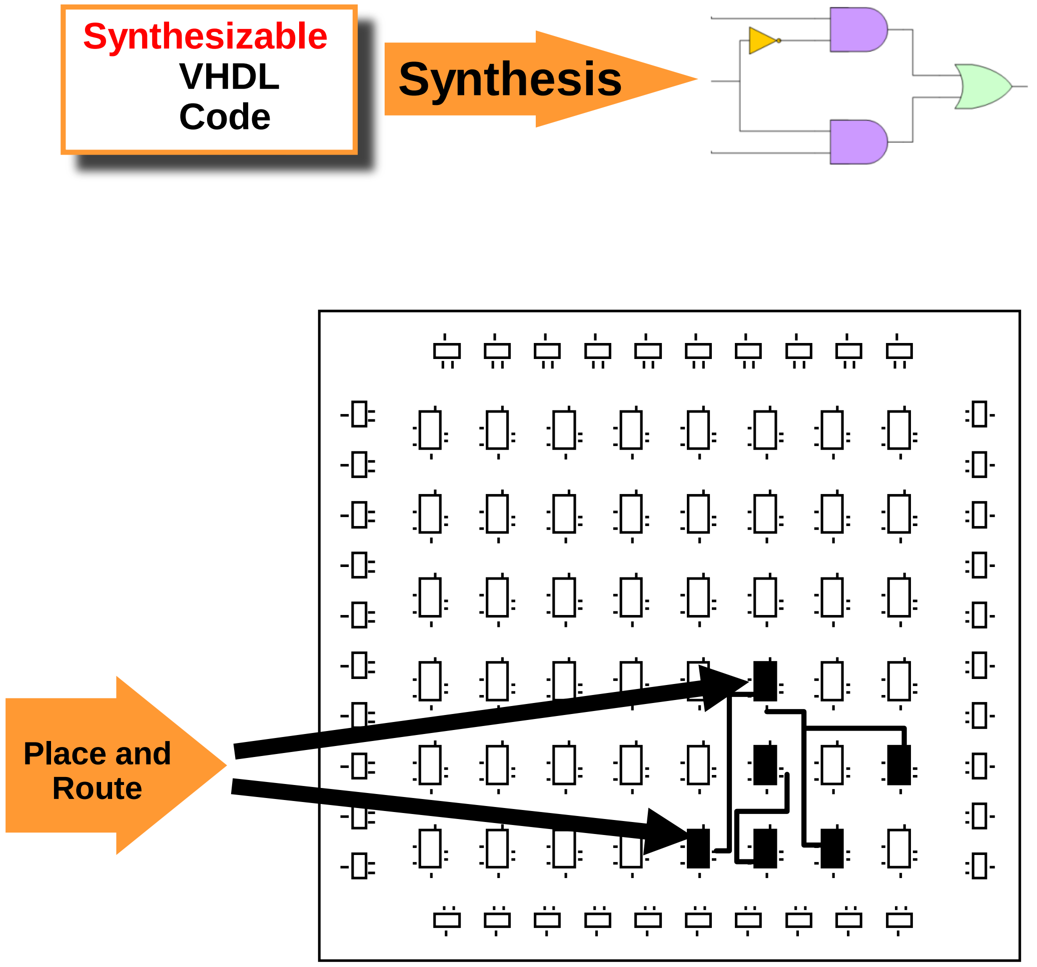
- We know the function we want, and can specify in C-like form.
  - … but we don't always know the exact gates (nor logic elements)…
  - … we want the tool to figure this out…





Importance of Synthesis

- In order to map an HDL code on a **FPGA**, **the code should be synthesizable!**
  - An HDL code that functions correctly in simulation, does not necessarily mean it is synthesizable.
  - Once you have gone through the synthesis tool for your HDL code **without errors**, your code is synthesizable.



VHDL Statements

- Concurrent
  - Signal assignment
  - Instantiation
  - when-else
  - with-select-when
  - process (as a wrapper for sequential statements)
- Sequential
  - Signal assignment - ONLY statement that is concurrent and sequential.
  - if-then-elsif-else - ONLY within a process
  - case-when - ONLY within a process

General VHDL Programming Flow

- **LIBRARY** and **USE** statements

Entity declaration:

- **ENTITY** entity_name **IS**
  - Identify the input and output **PORT**s and their data types
- **END** [entity_name];

Provide design description:

- **ARCHITECTURE** architecture_name **OF** entity_name **IS**
  - [SIGNAL declarations]
  - [CONSTANT declarations]
  - [TYPE declarations]
  - [COMPONENT declarations]
  - [ATTRIBUTE specifications]
- **BEGIN**
  - {COMPONENT instantiation statement ;}
  - {CONCURRENT ASSIGNMENT statement ;}
  - {PROCESS statement ;}
  - {GENERATE statement ;}
- **END** [architecture_name] ;

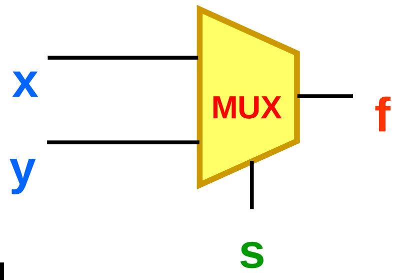
VHDL Syntax

- The basis of most of the VHDL is the logical interactions between signals in the modules.
  - Most of this is very intuitive, representative of logical functions.
- Another commonly used form of syntax is the conditional statements.
  - These work very much like the conditional statements of procedural programming that you should be used to.
- Keywords in VHDL are not case-sensitive.
- Names that user defines are case-sensitive.
- **END** statements do not require name of design entity or architecture to be followed.

Introductory Example

Introductory Example

- **2-1 Multiplexer**



- Characteristic table:

s	f
0	x
1	y

- Truth table:

s	x	y	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

- Boolean Equation:

$$f = \bar{s} \cdot x + s \cdot y$$

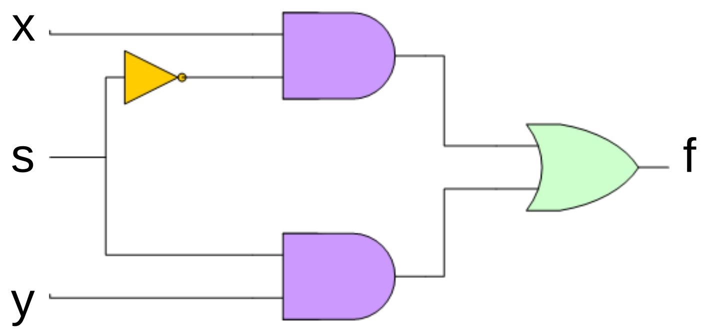
Introductory Example

- **2-1 Multiplexer**

- VHDL Code:

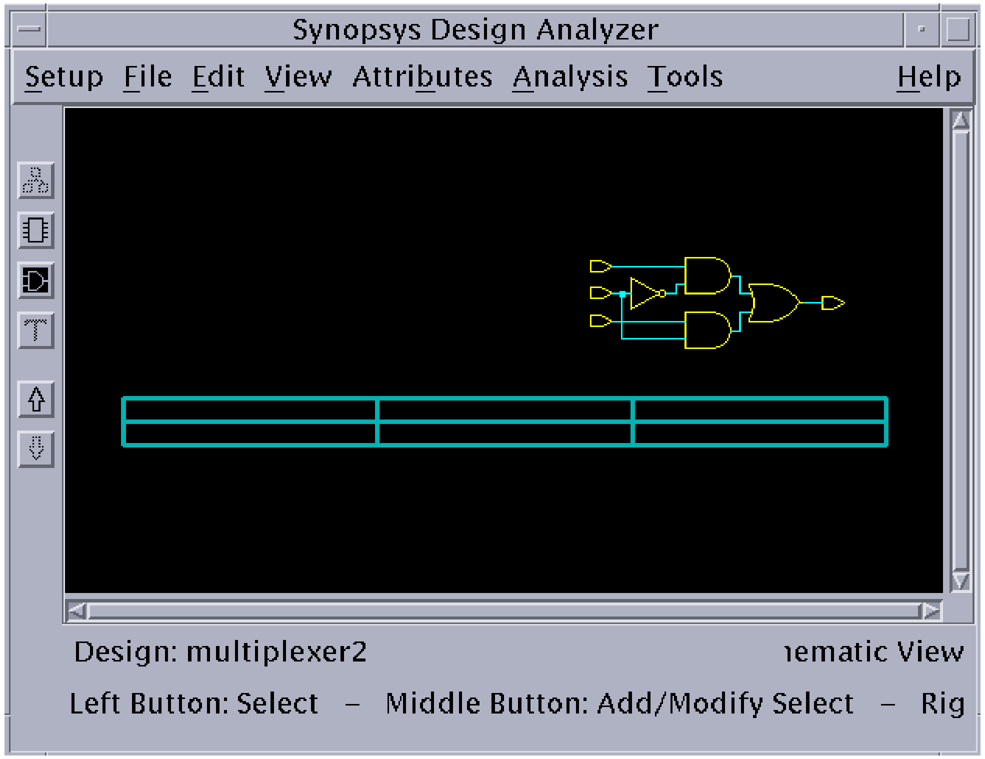
```
--Example 1: 2-1 Mux in VHDL
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY  multiplexer2 IS
PORT ( x, y, s        : IN   BIT ;
             f        : OUT  BIT ) ;
END multiplexer2 ;
ARCHITECTURE multiplexer2_arch OF multiplexer2 IS
BEGIN
f <= (x AND NOT s) OR (y AND s) ;
END multiplexer2_arch ;
```

- Gate-level description:



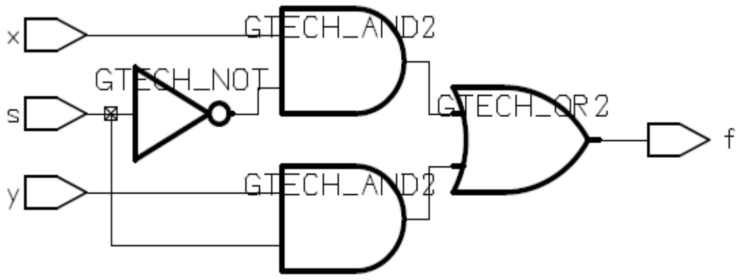
Introductory Example: Synopsys Synthesis

- Unoptimized circuit
  - Full Schematic View in Synopsys Design Analyzer graphical environment:
  - Gates used from Synopsys libraries:



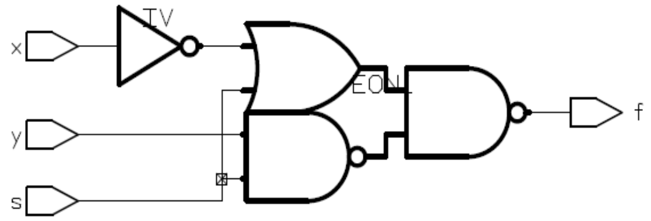
Design: multiplexer2 ematic View

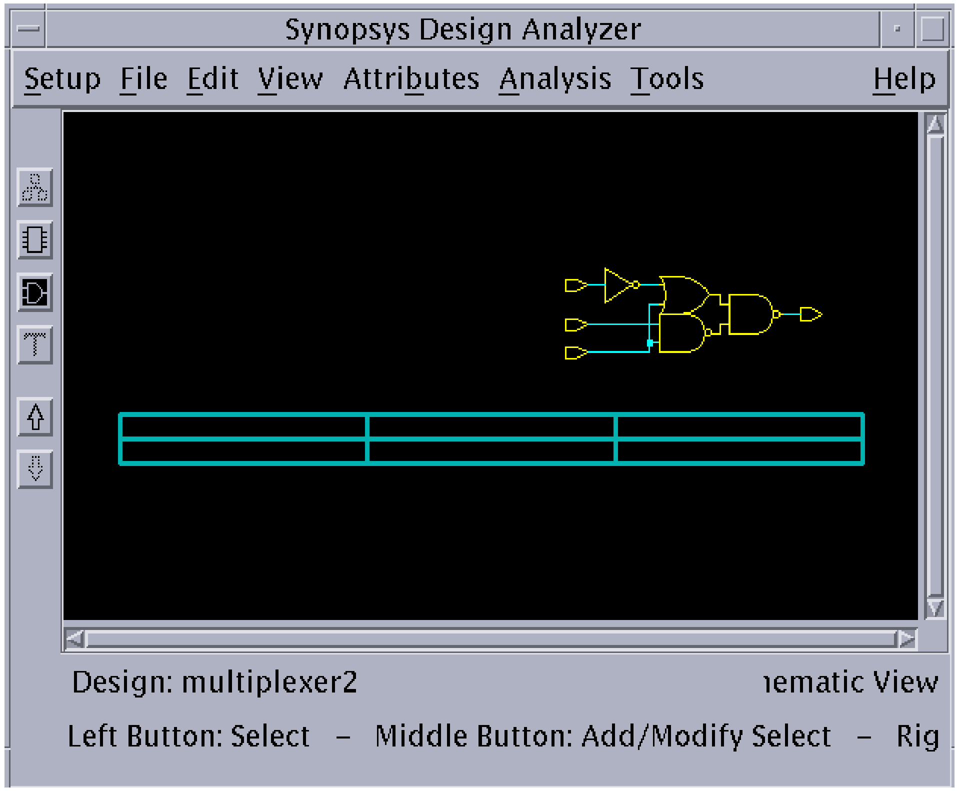
Left Button: Select – Middle Button: Add/Modify Select – Rig



Introductory Example: Synthesis (cont'd)

- Schematic of circuit after compilation and design optimization:





Design: multiplexer2 	ematic View

Left Button: Select – Middle Button: Add/Modify Select – Rig

- After synthesis and compilation, the tool picks different gate configuration for the HDL code we have written

Importance of Simulation

- The aim of simulation is to produce outputs (signals, integers, etc.) from specified input signals.
- Concurrent statements are evaluated whenever any input changes.
- If the evaluation of any concurrent statement results in an input signal change for any concurrent statement then that concurrent statement is evaluated.
- An important aspect of all designs is to do simulation. A motto that has been proven again and again is:

  If you don't simulate it, it won't work.
  If you do simulate it, it might work!

Testbench for HDL Simulation

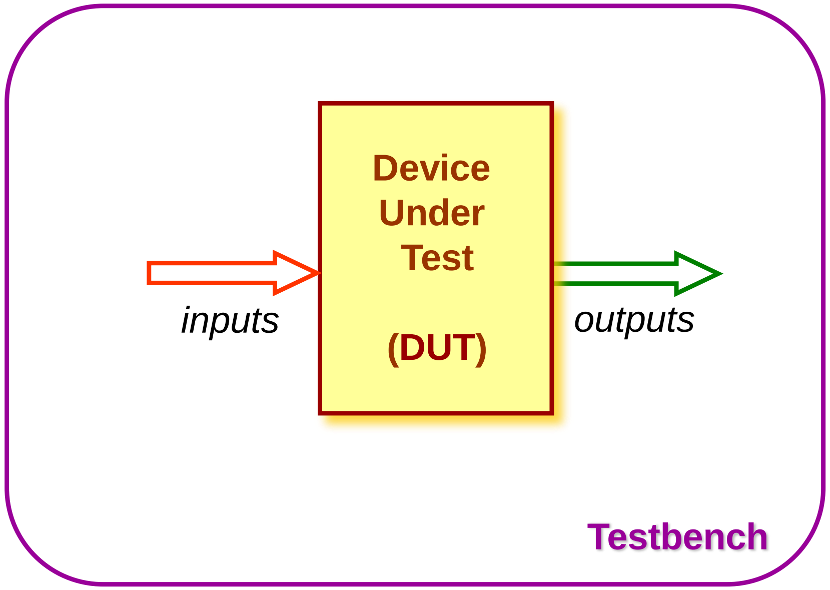
- Processes are a little different in that you must list the conditions that initiate evaluation of the process.
  - This can be done by WAIT statements or by a SENSITIVITY LIST.
- Synthesis usually ignores the sensitivity list.

- Testbench is used for generating stimulus for the entity under test.

- Different values are given to the primary input(s), output(s) are then observed in a wave graph or textual format to test the correctness of the design.

VHDL Testbench: One Approach

- Only the DUT is instantiated into test bench.
- Stimulus is generated inside the test bench
- Poor reusability.
- Suitable only for relatively simple designs.



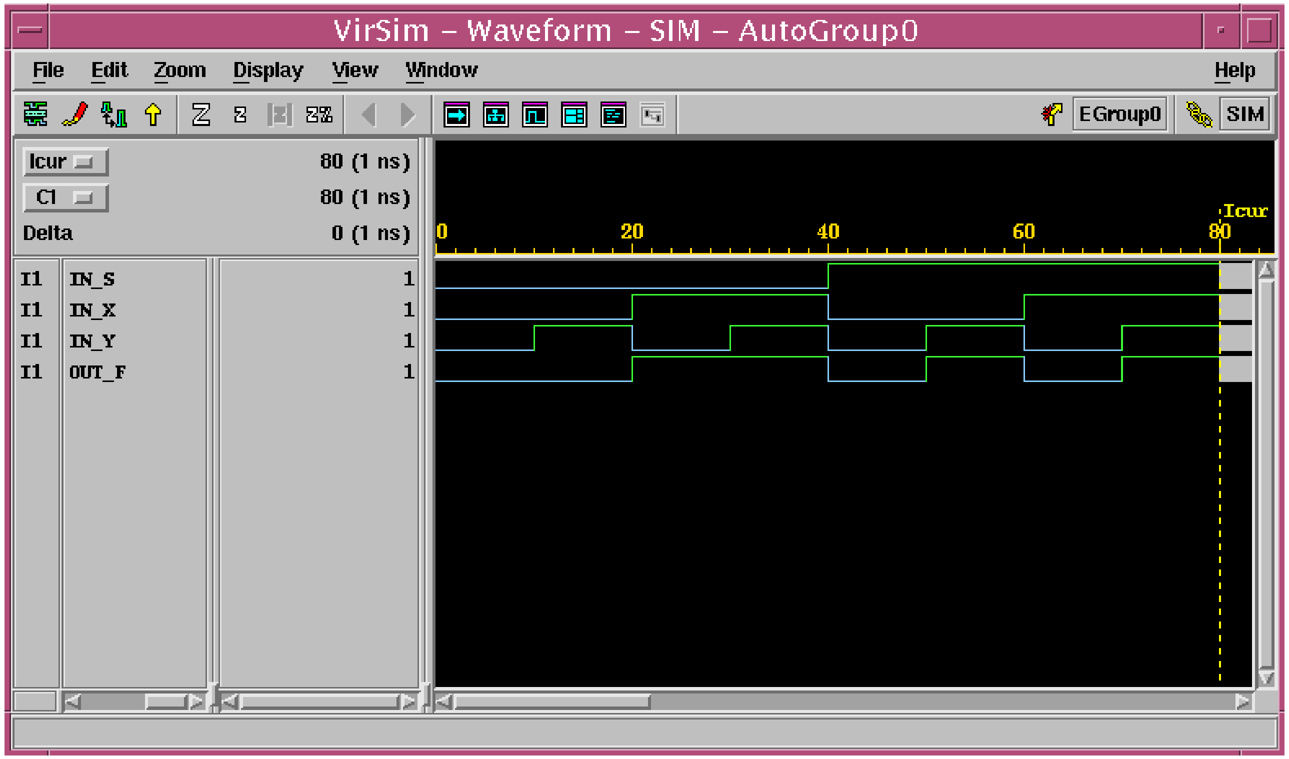
Introductory Example: Testbench1

- Testbench code for 2-1 multiplexer:

```
--Test bench1 for Example 1: 2-1 Mux
library IEEE;
USE IEEE.std_logic_1164.all;
entity tbmultiplexer2 is
end tbmultiplexer2;
architecture tbmultiplexer2_arch of tbmultiplexer2 is
component multiplexer2
PORT ( x, y, s       : IN   BIT ;
      f    : OUT   BIT ) ;
end component;
signal in_x, in_y, in_s, out_f: bit := '0';
begin
imultiplexer2:multiplexer2 port map(x=>in_x, y=>in_y, s=>in_s, f=>out_f);
in_x<='0', '1' after 20 ns, '0' after 40 ns, '1' after 60 ns;
in_y<='0', '1' after 10 ns, '0' after 20 ns, '1' after 30 ns, '0' after 40 ns,
           '1' after 50 ns, '0' after 60 ns, '1' after 70 ns;
in_s<='0', '1' after 40 ns;
end tbmultiplexer2_arch;
configuration cf_multiplexer2 of tbmultiplexer2 is
for tbmultiplexer2_arch
for imultiplexer2:multiplexer2
use entity WORK.multiplexer2 (multiplexer2_arch);
end for;
end for;
end cf_multiplexer2;
```

Introductory Example: Simulation1

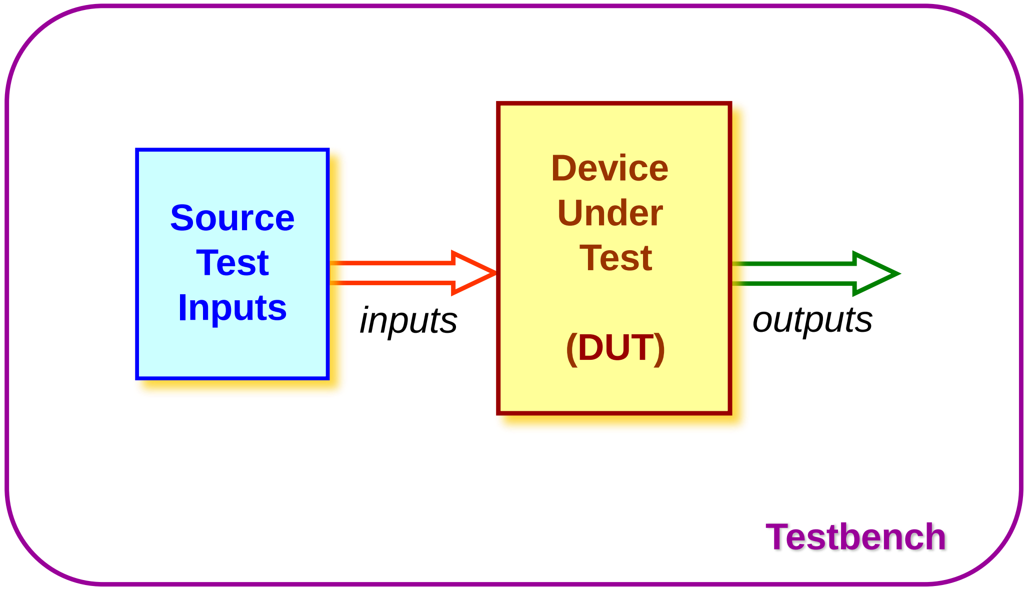
- Simulation Waveforms for 2-1 Mux
  - Scirocco Virsim Waveform Graph from Synopsys is invoked:



- IN_S is the select line.
- IN_X and IN_Y are the inputs.
- OUT_F is the output -> Correct functionality achieved

VHDL Testbench: Another approach

- Source and DUT instantiated into testbench.
- For designs with complex input and simple output.
- Source can be for instance an entity or a process or directly the stimulus.



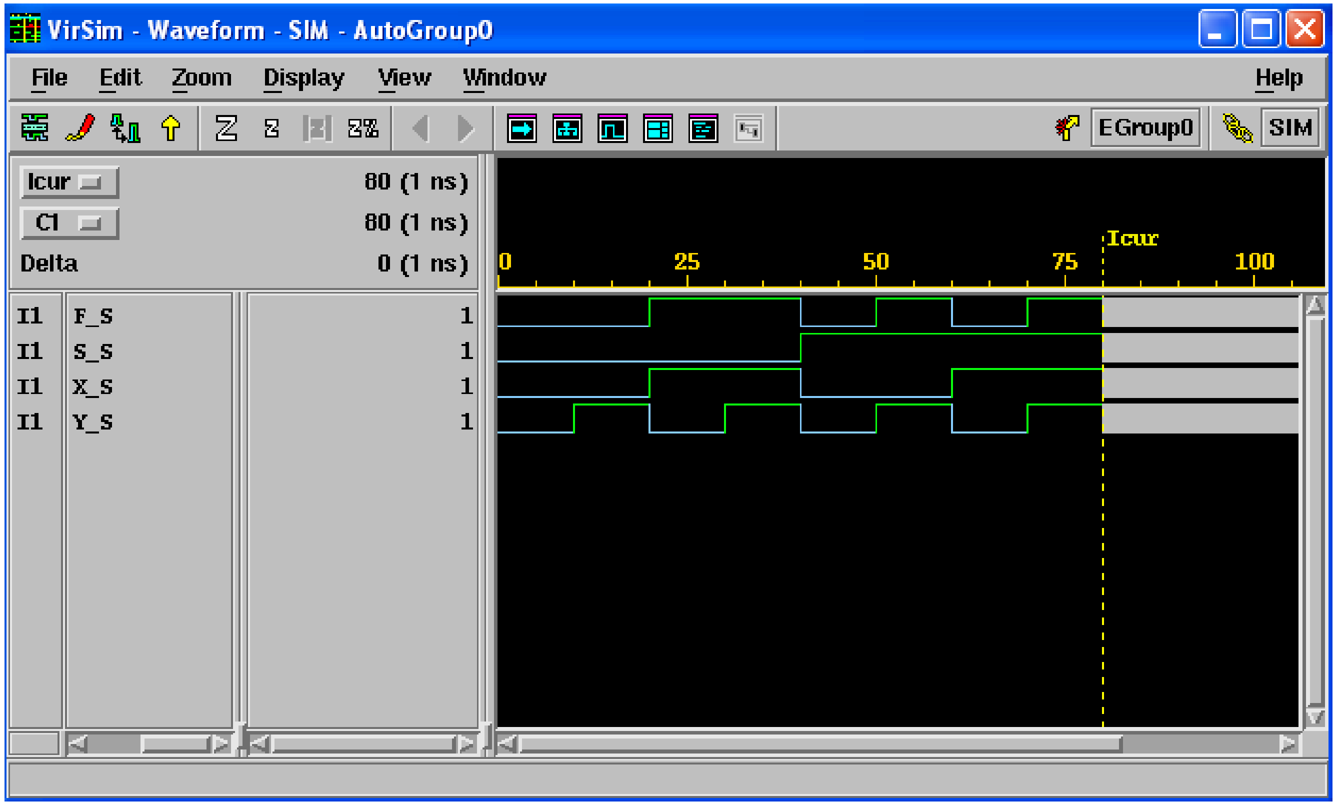
Introductory Example: Testbench2

- Testbench code for 2-1 multiplexer (another approach):

```
--Test bench 2 for Example 1: 2-1 Mux
library IEEE;
use IEEE.std_logic_1164.all;
ENTITY  mux2test IS
PORT ( ff     : IN   BIT ;
     xx, yy, ss              : OUT   BIT ) ;
END mux2test ;
ARCHITECTURE mux2test_arch OF mux2test IS
BEGIN
xx<='0', '1' after 20 ns, '0' after 40 ns, '1' after 60 ns;
yy<='0', '1' after 10 ns, '0' after 20 ns, '1' after 30 ns, '0' after 40 ns, '1' after 50 ns, '0' after 60
     ns, '1' after 70 ns;
ss<='0', '1' after 40 ns;
END mux2test_arch ;
--------------------------------------------
library IEEE;
USE IEEE.std_logic_1164.all;
entity tbmux2 is
end tbmux2;
architecture tbmux2_arch of tbmux2 is
component multiplexer2
PORT ( x, y, s               : IN   BIT ;
          f   : OUT   BIT ) ;
end component;
component  mux2test
PORT ( ff     : IN   BIT ;
     xx, yy, ss              : OUT   BIT ) ;
END component ;
signal x_s, y_s, s_s, f_s: bit;
begin
imultiplexer2:multiplexer2 port map(x=>x_s, y=>y_s, s=>s_s, f=>f_s);
mux2test1:mux2test port map(ff=>f_s, xx=>x_s, yy=>y_s, ss=>s_s);
end tbmux2_arch;
configuration cf_multiplexer2 of tbmux2 is
for tbmux2_arch
for imultiplexer2:multiplexer2
use entity WORK.multiplexer2 (multiplexer2_arch);
end for;
end for;
end cf_multiplexer2;
```

Introductory Example: Simulation2

- Simulation Waveforms for 2-1 Mux
  - Scirocco Virsim Waveform Graph from Synopsys is invoked:



- S_S is the select line.
- X_S and Y_S are the inputs.
- F_S is the output -> Correct functionality achieved